CH
BA
OF]
VIEW
N
ASAN
NIN
SRI
21:
EXHIBIT

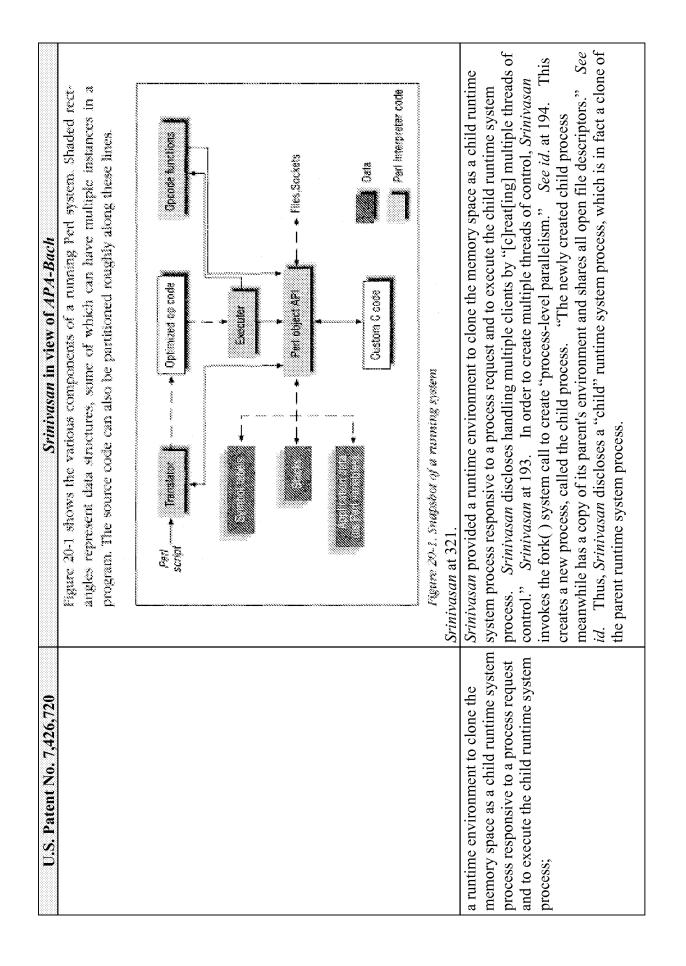
"Advanced Perl Programming" O'Reilly & Associates, Inc. Author: Sriram Srinivasan Publication Date: August 1997 ("Srinivasan") M. J. Bach, The Design of the Unix Operating System, Bell Telephone Labs., Inc. (1986) ("APA-Bach")

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
1. A system for dynamic preloading of	Srinivasan provided a system for dynamic preloading of classes through memory space
classes through memory space cloning	cloning of a master runtime system process.
of a master runtime system process,	
comprising:	
A processor;	Srinivasan provided a system that ran on a computer with a processor and memory. For
A memory	example, Srinivasan references an "in-memory cache of objects." Srinivasan at 178.
a class preloader to obtain a	Srinivasan provided a class preloader to obtain a representation of at least one class from a
representation of at least one class from	source definition provided as object-oriented program code. Srinivasan discusses object
a source definition provided as	oriented programming and its usefulness in conjunction with Perl code. Srinivasan at 101.
object-oriented program code;	Srinivasan notes that, in Perl, a "class is a package." Srinivasan at 389. Srinivasan then
	discusses two methods for the creation of a package from source definition provided as
	object oriented program code. See id. at 389-390 (discussing creation of the "employee"
	class). In order to preload the class, <i>Srinivasan</i> discloses specific functional code:
	"Using object package:
	use Employee;
	Semp = Employee->new ("Ada", 3 5);
	<pre>\$emp->set_salary(1000);</pre>
	<i>Dee la</i> . al <i>39</i> 0.

Srinivasan in view of APA-Bach	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer and construction of the statempt to provide similar isolation characteristics (and don't	Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." <i>Srinivasan</i> at 98.	"Object orientation (OO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chapter, we will study what the noise is all about and build objects using Perl. I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." <i>Srinivasan</i> at 99.	"Objects of a certain type are said to belong to a <i>class.</i> " <i>Srinivasan</i> at 101. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises hand-coded lexer (<i>toke.c</i>), the yacc-based parser (perly.y), and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in <i>toke.c</i> and compiled to an internal format in <i>regcomp.c</i> . Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern
U.S. Patent No. 7,426,720				

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, chop, push, index, rindex, grep,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." <i>Srinivasan</i> at 323-24.
	"Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of this writing. It can be asked to translate a script to C code, which can be compiled to form an executable; as it happens, this executable is not much faster than the interpreted script, because most of the action still takes place in opcode functions as they exist now. Static typing hints may usher in some aggressive optimizations. For example, if you say: my integer \$i;
	the compiler would use C's native integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler can alternatively produce a byte-code file and have the interpreter eval it subsequently, similar to the facilities provided by Python and Java. It also supports much better debugging options than those currently provided with $-D$." <i>Srinivasan</i> at 370.
a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime	<i>Srinivasan</i> provided a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process. <i>Srinivasan</i> discloses handling multiple clients by "[c]reat[ing] multiple threads of control." <i>Srinivasan</i> at 193. In order to create multiple threads of control, <i>Srinivasan</i> invokes the

7,426,720 Srinivasan in view of APA-Bach	fork() system call to create "process-level parallelism." <i>See id.</i> at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." <i>See id.</i> Thus, <i>Srinivasan</i> discloses a "parent" runtime system process, <i>i.e.</i> , a master runtime system process.	"The data structures described above are normally kept in global C variables. If Perl is compiled with <i>–DMULTIPL/CITY</i> , it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." <i>Srinivasan</i> at 323.
U.S. Patent No. 7,426,720	system process;	



Ś

"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered "Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports $fork$, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:	<pre># Forking server use IO:: Socket; sSIG(CHLD) sub {wait ()}; %SIG(CHLD) sub {wait ()}; %sain_sock = new IO::Socket::INET (LocalHost => 1200, Listen => 1200, Listen => 1, Reuse => 1, p; die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = \$main_sock->accept()) { %pid = fork(); die "cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process while (\$pid == 0) {</pre>
U.S. Patent No. 7,426,720	

	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute exit(), or a termination status otherwise), just in case the parent uses wait or waitpid to enquire about this status. The terminated child process is also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.	s cloningSrinivasan in view of APA -Bach provided the use of $fork$ in Unix/Linux, which includes a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a parace of the memory space of the master runtime system process into a memory space of the master runtime system process into a parace of the memory space of the master runtime system process. The memory space of the master runtime system process and to defer copying of the memory space of the master runtime system process. The he child runtime system process uncludes the child runtime system process. The fork() system call disclosed by Srinivasan was commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory; this is ess until the eactibed in detail in APA -Bach. One of ordinary skill in the art at the time of the invention, seeking to reduce the impact of a fork() system call on local memory, would look to the disclosure of APA -Bach for a description of the copy-on-write bit."The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call." APA -Bach at 192."The copy-on-write bit, used in the fork system call, indicates that the kernel must create a the fork system call, indicates that the kernel must create a
U.S. Patent No. 7,426,720			and a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

<i>Srinivasan</i> in view of <i>APA-Bach</i> new copy of the page when a process modifies its contents." <i>APA-Bach</i> at 287.	"9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the <i>fork</i> system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call <i>exec</i> soon after the <i>fork</i> call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions The page can now be reference through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during <i>fork</i> . If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." <i>APA-Bach</i> at 289–90.	<i>Srinivasan</i> provided a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process. For example, <i>Srinivasan</i> discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." <i>Srinivasan</i> at 179. In <i>Srinivasan</i> , "[o]bjects of a certain type are said to belong to a <i>class</i> ." <i>Srinivasan</i> at 101. Thus, the exemplar disclosure effects a search through the available classes. "This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
U.S. Patent No. 7,426,720		2. A system according to claim 1, further comprising: a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process.

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	"hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular
	expression-matching code, which is why regex matchers written in Java won't come anywhere close in performance (I'll reevaluate this claim once Sun's Java processors are
	freely available.) As you will see later on, the opcode functions look strikingly similar to the plue code output by xsubm/SWIG: this is because they interoperate using the aroument
	stack and obey the same parameter passing protocols." <i>Srinivasan</i> at 324. "This returns a list of object references that match the query criteria. Now if you reissue this
	query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that
	have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
	"The Adaptor: : File module does not have this problem because it maintains a list of all
	objects given to its store () method (for reasons to be explained in the next section); hence
	successive incluted queries return ruentical fists. Queries
	One big reason why object-oriented databases haven't caught on is the lack of a query
	language (or at least a standard query language). When you have a million objects in the database it would be a terrible thing to load every single object in memory to see whether it
	matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates
	queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded
	and cycles through all objects, matching them against the query specification.
	Schema Evolution
	Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are
	added to be able to reconcile old data with newer object implementations.
	Implementation
	This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
	attention to the design gotchas and unimplemented features as you do to the code.

<i>Srinivasan</i> in view of <i>APA-Bach</i> Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.	Fite Adaptor	Figure 11-2 Structure of file adeptor Srinivasan at 179-80.	<i>Srinivasan</i> provided a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. For example, <i>Srinivasan</i> discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." <i>Srinivasan</i> at 179. In <i>Srinivasan</i> , "[o]bjects of a certain type are said to belong to a <i>class</i> ." <i>Srinivasan</i> at 101. Thus, the exemplar disclosure effects a search through the available in the local cache. It is inherent that such a search could locate a class definition that is unavailable in the local cache.	"This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that
U.S. Patent No. 7,426,720			3. A system according to claim 2, further comprising: a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache.	

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence
	Queries
	One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the
	database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates
	queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded
	and cycles through all objects, matching them against the query specification.
	Schema Evolution
	Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has
	to be able to reconcile old data with newer object implementations.
	Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
	only the key procedures that perform query processing and file or database I/O. Pay as much
	attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File
	An Adaptor::File instance represents all objects stored in one file. When this adaptor is
	created (using new), it reads the entire file and translates the data to in-memory objects.
	Slurping the entire file into memory avoids the problem of having to implement fancy
	on-disk schemes for random access to variable-lengin data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large
	numbers of objects (over 1,000, to pick a number).
	The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their id) as chown in Figure 11.2
	111 value (alle illevies of their -1 uf , as shown in 113 13 $ur < 11-2$.

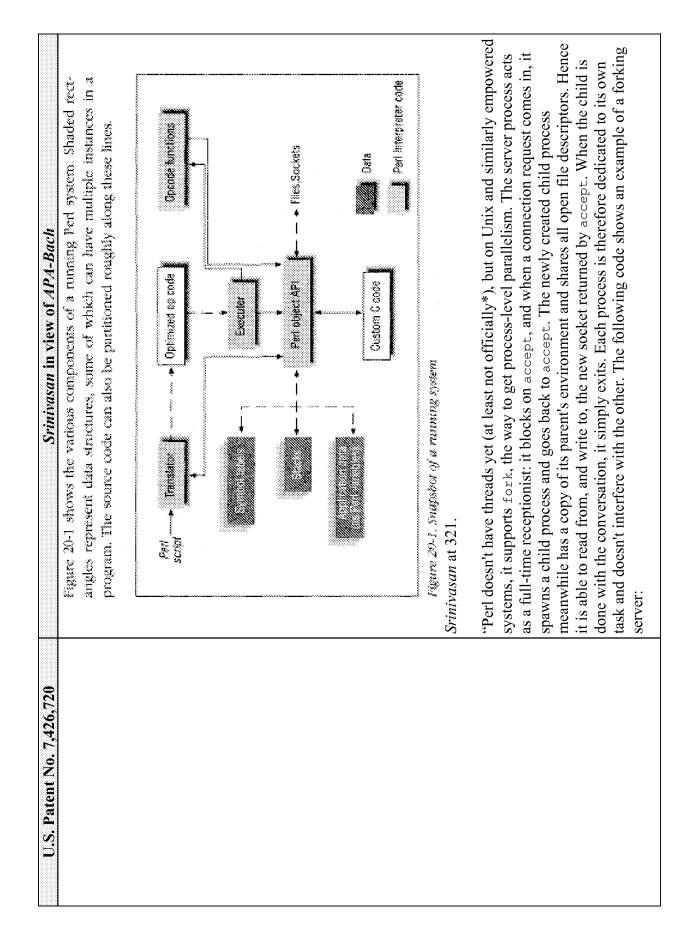
Srinivasan in view of APA-Bach	Fite Adaptor	Figure 11.2. Spracture of file adaptar	Srinivasan at 179-80.	"Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." <i>Srinivasan</i> at 184.	"retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other" <i>Srinivasan</i> at 186.	<i>Srinivasan</i> provided a class resolver to resolve the class definition. <i>Srinivasan</i> discloses that, in Perl, "[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design." <i>Srinivasan</i> at 107-108.	"This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
U.S. Patent No. 7,426,720						4. A system according to claim 1, further comprising: a class resolver to resolve the class definition.	

"The Adaptor: : File objects given to its st successive identical o Queries One big reason why o language (or at least database, it would be matches your criteria queries to equivalent	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists.
objects given to its st successive identical o Queries One big reason why o language (or at least database, it would be matches your criteria queries to equivalent	s store () method (for reasons to be explained in the next section); hence al queries return identical lists.
successive identical c Queries One big reason why c language (or at least database, it would be matches your criteria queries to equivalent	al queries return identical lists.
Queries One big reason why one big reason why clanguage (or at least database, it would be matches your criteria queries to equivalent	
One big reason why one big reason why one big reason why one and the set of t	
language (or at least a database, it would be matches your criteria queries to equivalent	One big reason why object-oriented databases haven't caught on is the lack of a query
database, it would be matches your criteria queries to equivalent	language (or at least a standard query language). When you have a million objects in the
matches your criteria queries to equivalent	database, it would be a terrible thing to load every single object in memory to see whether it
queries to equivalent	matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates
	queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded
scheme for file based	scheme for file based objects: it converts the query expression to an evalable Perl expression
and cycles through al	and cycles through all objects, matching them against the query specification.
Schema Evolution	
Let us say you have s	et us say you have sent your objects' data to a file, and tomorrow, some more attributes are
added to the object ir	dded to the object implementation. The schema is said to have evolved. The framework has
to be able to reconcil	to be able to reconcile old data with newer object implementations.
Implementation	
This section explains	his section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
only the key procedu	nly the key procedures that perform query processing and file or database I/O. Pay as much
attention to the desig	attention to the design gotchas and unimplemented features as you do to the code.
Adaptor::File	
An Adaptor::File inst	An Adaptor::File instance represents all objects stored in one file. When this adaptor is
created (using new),	created (using new), it reads the entire file and translates the data to in-memory objects.
Slurping the entire fil	e file into memory avoids the problem of having to implement fancy
on-disk schemes for 1	on-disk schemes for random access to variable-length data; after all, that is the job of DBM
and database impleme	and database implementations. For this reason, this approach is not recommended for large
numbers of objects (or	numbers of objects (over 1,000, to pick a number).
The file adaptor has a	he file adaptor has an attribute called all_instances, a hash table of all objects given to its store
method (and indexed	hethod (and indexed by their _id), as shown in Figure 11-2.

U.S. Patent No. 7,426,720 "This te client/se way that way that example be able way that example be able "This re evertes objects success oueries one big databas matchet quertes scheme and cyc Schema Let us s added to to be able Implem	<i>Strinivasan</i> in view of <i>APA-Bach</i> . "This technique is often employed in the guts of client/server libraries. In a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this" <i>Srinivasan</i> at 134. "This returns a list of object references that match the query eriteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Oueries and the store of objects in the database is a value object in the adatabase, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor:: DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded as and cycles through all objects, matching them against the query specification. Let us say unberentiation. The schema is said to have evolved. The framework has the mean added to the object implementation. The schema is said to have evolved. The framework has the mean is and to the object implementation.
Adapto	only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File

<i>Srinivasan</i> in view of <i>APA-Bach</i> An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.	File Adaptor	l. Figure 11-2. Structure of file adaptor	Srinivasan at 179-80.	"Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." <i>Srinivasan</i> at 184.	"retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other" <i>Srinivasan</i> at 186.	<i>Srinivasan</i> provided a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process. <i>Srinivasan</i> discloses handling multiple clients by "[c]reat[ing] multiple threads of control." <i>Srinivasan</i> at 193. In order
U.S. Patent No. 7,426,720						6. A system according to claim 1, further comprising: a process cloning mechanism to instantiate the child runtime system process by copying the

U.S. Patent No. 7,426,720 memory snace of the master mutime	Srinivasan in view of APA-Bach to create multimle threads of control. Srinivasan invokes the fork() system call to create
system process into a separate memory sugar for the child runtime system	"process-level parallelism." See id. at 194. This creates a new process, called the child "process "The newly created child process meanwhile has a conv of its parent's
process.	process. The new y created child process including that a copy of the parents of the parent of the parent in fact a clone of the parent runtime system process.
	"The data structures described above are normally kept in global C variables. If Perl is compiled with <i>–DMULTIPLICITY</i> , it lumps all these global variables into a structure called
	Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of
	type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum
	performance. The API remains the same in either case. You can use multimle intermeters to enforce commletely isolated namesnaces. Fach
	interpreter has its own "main" package and its own tree of loaded packages. I have not seen
	this feature used in production Perl applications, but Tel provides a framework called
	These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses
	a different mechanism, though the result (of isolated name spaces) is similar. More on this in
	the next section." Srinivasan at 323.



U.S. Patent No. 7,426,720	
	<pre># Forking server use IO:: Socket; cerr(curn) curb (; /;</pre>
	אסור
	Proto => 'tcp',
	Keuse => I,
	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>
	<pre>\$pid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ;</pre>
	if (\$pid == 0) {
	# UNIIα process while (defined (\$buf <\$new sock»)
	print \$new_sock "You said: \$buf\n";
	<pre>process exits when it is done. # else 'tis the parent process, which goes back to accept()</pre>
	} close (\$main_sock);
	The fork call results in two identical processes-the parent and child-starting from the
	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the
	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client
	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child
	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
	the child was able to execute exit (), or a termination status otherwise), just in case the
	parent uses wait or waitpid to enquire about this status. The terminated child process is
	also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait;

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.
7. A system according to claim 1, wherein the master runtime system process is caused to sleep relative to receiving the process request.	<i>Srinivasan</i> provided a system wherein the master runtime system process is caused to sleep relative to receiving the process request. For example, <i>Srinivasan</i> discloses a persistent process, "also known as a zombie process." <i>Srinivasan</i> at 195.
	"If you want a finer granularity, you can use the Time: : HiRes module available from CPAN, which gives microsecond resolution on Unix systems (gives access to the usleep and ualarm system calls) On Microsoft Windows systems, you can use the Win32 : : Timer call for millisecond-level timing." <i>Srinivasan</i> at 141.
	"The $fork$ call results in two identical processes the parent and child-starting from the statement following the $fork$. The parent gets a positive return value, the process ID (pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client.
	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute exit (), or a termination status otherwise), just in case the
	parent uses walt or waltpld to enquire about this status. The terminated child process is also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular." <i>Srinivasan</i> at 195.

Srinivasan in view of APA-BachSrinivasan provided a system wherein the object-oriented program code is written in the Java programming language. For example, Srinivasan discloses that "Java offers two learn about the notion of classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both."	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't	offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." <i>Srinivasan</i> at 98.	"The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (<i>toke.c</i>), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in <i>toke.c</i> and compiled to an internal format in <i>regcomp.c</i> . Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons.
U.S. Patent No. 7,426,720 8. A system according to claim 1, wherein the object-oriented program code is written in the Java programming language.			

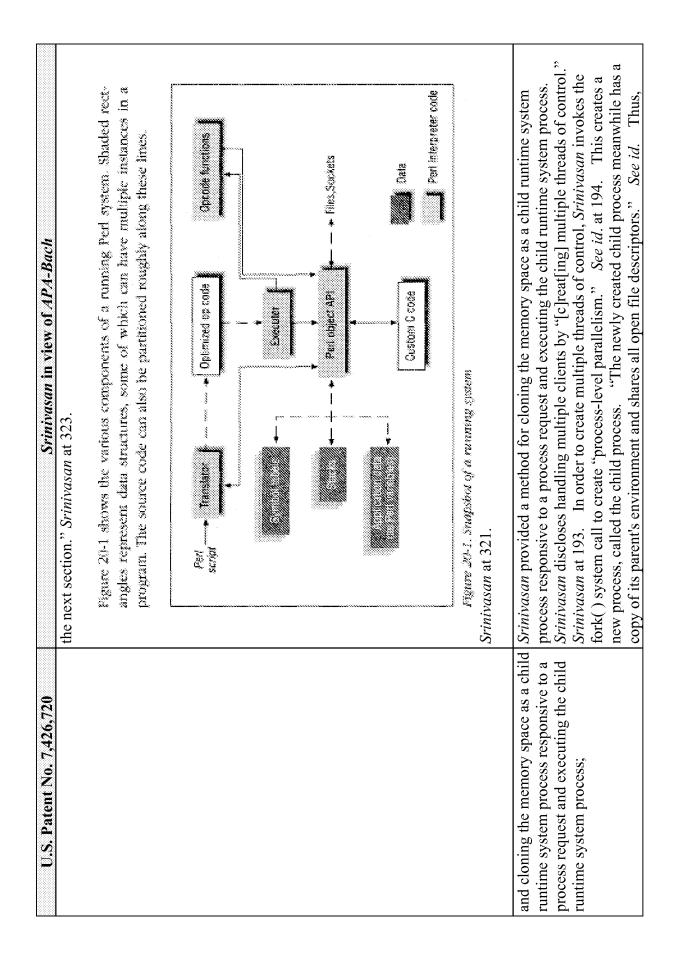
U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach contains a pointer to the next one to execute and a pointer to the data it is expected to work
	on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, chop, push, index, rindex, grep,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovinoly hand-ontimized e code. You can also see why it is hard to create a
	Perl-to-Java byte-code translator: there is no correspondence between the two sets." Srinivasan at 323-24.
10. A method for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	<i>Srinivasan</i> provided a method for dynamic preloading of classes through memory space cloning of a master runtime system process.
executing a master runtime system process;	<i>Srinivasan</i> provided a method for executing a master runtime system process. <i>Srinivasan</i> discloses handling multiple clients by "[c]reat[ing] multiple threads of control." <i>Srinivasan</i> at 193. In order to create multiple threads of control, <i>Srinivasan</i> invokes the fork() system call to create "process-level parallelism." <i>See id.</i> at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." <i>See id.</i> Thus, <i>Srinivasan</i> discloses a "parent" runtime system process, <i>i.e.</i> , a master runtime system process.
	"The data structures described above are normally kept in global C variables. If Perl is compiled with <i>–DMULTIPLICITY</i> , it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case.

<i>Srinivasan</i> in view of <i>APA-Bach</i> You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." <i>Srinivasan</i> at 323.	Figure 20-1 shows the various components of a running Perl system. Shaded rect- angles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned raughly along these lines.	Peri - Thanklatter - Diplomiced tra coults - Optimiced tra coults - Optimiced tra coults - Optimiced tra coults	Ant tablect API Arr - + Files, Suckets	Custom 6 code	Figure 20-1. Snapshea of a running system Srinivasan at 321.	ne Srinivasan provided a method for obtaining a representation of at least one class from a
U.S. Patent No. 7,426,720						obtaining a representation of at least one

U.S. Patent No. 7,426,720 class from a source definition provided	2 0000
as object-oriented program code;	oriented programming and its usefulness in conjunction with Perl code. <i>Srinivasan</i> at 101. <i>Srinivasan</i> notes that, in Perl, a "class is a package." <i>Srinivasan</i> at 389. <i>Srinivasan</i> then discusses two methods for the creation of a package from source definition provided as object oriented program code. <i>See id.</i> at 389-390 (discussing creation of the "employee"
	class). In order to preload the class, Srinivasan discloses specific functional code:
	use Employee; \$emp = Employee->new ("Ada", 3 5);
	<pre>\$emp->set_salary(1000);</pre>
	<i>See id.</i> at 390.
	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is
	equivalent to both. Java does not allow one package to mess around with another package s namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stonned
	determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't
	offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Develonment Kit ODK).
	Java has gained significant reflection capabilities, though nowhere near the amount of
	productly available information for gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to
	different models of programming. Men were sent to the moon while FORTRAN and COBOL miled the roost which provise that you can get a whole lot done if you don't indulge
	in language wars. Java allows you to dynamically "dispatch" a function call, by giving the
	function's name as a string, and to trap an exception if the function doesn't exist; this is like
	using Perl's symbolic references." Srinivasan at 98.

U.S. Patent No. 7.426.720	Srinivasan in view of APA-Bach
	"Object orientation COO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chanter, we will study what the noise is all about and build objects using Perl.
	I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." <i>Srinivasan</i> at 99.
	"Objects of a certain type are said to belong to a <i>class.</i> " <i>Srinivasan</i> at 101.
	"The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (<i>toke.c</i>), the yacc-based parser (perly.y), and the code generator (op.c).
	Regular expressions–which form a distinct sublanguage–are recognized in <i>toke.c</i> and compiled to an internal format in <i>regcomp.c</i> . Opcodes are similar in concept to machine
	code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern
	interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly
	contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these oncodes are not mere instruction types: they actually
	embody the exact unit of work expected at that point in that program.
	Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of
	abstraction. A large number of these opcodes directly correspond to the facilities available at
	the scripting level, such as regular expression matching and substitution, chop, push, index, rindex, grep,* and so on, which explains why there are 343 opcodes as of this writing! It
	work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets "
	Srinivasan at 323-24.
	"Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of
	this writing. It can be asked to translate a script to C code, which can be compiled to form an executable; as it happens, this executable is not much faster than the interpreted script.

U.S. Patient No. 7,426,720 Syminus of the action still takes place in opcode functions as they exist now. Static typing hins may taken in some aggressive optimizations. For example, if you say: wy integer 3 is any taken in some aggressive optimizations. For example, if you say: w integer 3 is the static integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler and alternatively produce a byte-code file and have the interpreter eval. it subsequently, similar to the facilities provided with $-D$." <i>Strinsaan</i> at 730. Strinsaan at 730. Strinsaan at 730. Strinsaan at 730. Strinsaan at 730. System process; Strinsaan at 730. System process; Strinsaan at 730. Strinsaan at 730. Strinsaan at 733. Strinsaan at 730. Strinsaan at 793. Strinsaan at 730. Strinsaan at 793. Strinsaan at 730. Strinsaan at 793. Strinsaan at 730. Strinsaan at 730. Strinsaan at 730. Strin task of 100000000000000000000000000000000000
--



U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	<i>Srinivasan</i> discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process.
	"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports fork, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it is the server process of the server process acts acts as a full-time reception between a server process.
	spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:
	# Forking server use IO:: Socket;
	<pre>\$SIG{CHLD} sub {wait ()}; \$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',</pre>
	5, tcp
); die "Socket could not be created. Reason: \$!\n" unless (\$sock);
	<pre>ew_sock = \$main_sock->accept()) {</pre>
	die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process
	while (defined (\$buf <\$new_sock») # do something with \$buf
	print \$new_sock "You said: \$buf\n";
	<pre></pre>
	close (\$main_sock);

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	The f_{ork} call results in two identical processes-the parent and child-starting from the statement following the f_{ork} . The parent gets a positive return value, the process ID (p_{id}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client.
	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute $exit()$, or a termination status otherwise), just in case the parent uses wait or waitpid to enquire about this status. The terminated child process is
	also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.
wherein cloning the memory space as a child runtime system process involves instantiating the child runtime system	<i>Srinivasan</i> in view of <i>APA-Bach</i> provided the use of <i>fork</i> in Unix/Linux, which includes a method wherein cloning the memory space as a child runtime system process involves instantiating the child runtime system process by copying references to the memory space of
process by copying references to the memory space of the master runtime system process into a separate memory	the master runtime system process into a separate memory space for the child runtime system process; and wherein copying references to the memory space of the master runtime system process defers copying of the memory space of the master runtime system process
space for the child runtime system process;	until the child runtime system process needs to modify the referenced memory space of the master runtime system process. The fork() system call disclosed by <i>Srinivasan</i> was
and wherein copying references to the memory space of the master runtime	commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory; this is described in detail in <i>APA-Bach</i> . One of ordinary skill in
system process defers copying of the memory space of the master runtime system process until the child runtime	the art at the time of the invention, seeking to reduce the impact of a fork() system call on local memory, would look to the disclosure of APA -Bach for a description of the copy-on-write bit.

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
system process needs to modify the referenced memory space of the master runtime system process.	"The only way for a user to create a new process in the UNIX operating system is to invoke the <i>fork</i> system call." APA - $Bach$ at 192.
	"The <i>copy-on-write</i> bit, used in the <i>fork</i> system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." <i>APA-Bach</i> at 287.
	"9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the <i>fork</i> system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call <i>exec</i> soon after the <i>fork</i> call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during <i>fork</i> . If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." $APA-Bach$ at 289–90.
11. A method according to claim 10, further comprising: determining whether the instantiated class definition is available in a local cache associated with the master runtime system process.	<i>Srinivasan</i> provided a method for determining whether the instantiated class definition is available in a local cache associated with the master runtime system process. For example, <i>Srinivasan</i> discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." <i>Srinivasan</i> at 179. In <i>Srinivasan</i> , "[o]bjects of a certain type are said to belong to a <i>class</i> ." <i>Srinivasan</i> at 101. Thus, the exemplar disclosure effects a search through the available classes. "This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their d), as shown in Figure 11-2.
	Fite Adaption
	Eggere 11-2. Structure of file adaptor Srinivasan at 179-80.
12. A method according to claim 11, further comprising: locating the source definition if the instantiated class definition is unavailable in the local cache.	<i>Srinivasan</i> provided a method for locating the source definition if the instantiated class definition is unavailable in the local cache. For example, <i>Srinivasan</i> discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." <i>Srinivasan</i> at 179. In <i>Srinivasan</i> , "[o]bjects of a certain type are said to belong to a <i>class</i> ." <i>Srinivasan</i> at 101. Thus, the exemplar disclosure effects a search through the available

11 C Datent No. 7 176 770	Cuintingen in view of 1D4 Rach
	classes. It is inherent that such a search could locate a class definition that is unavailable in the local cache.
	"This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists.
	One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it
	matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification.
	Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations.
	This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code.
	An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy

Srinivasan in view of APA-Bach on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all instances, a hash table of all objects given to its store	method (and indexed by their _id), as shown in Figure 11-2.	decry, dat and instances 23 23 23 23 23 23 23 23 23 23	kgure 11-2. Structure of file adaptor	<i>Srinivasan</i> at 179-80.	"Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." <i>Srinivasan</i> at 184.	"retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other" <i>Srinivasan</i> at 186.	<i>Srinivasan</i> provided a method for resolving the class definition. <i>Srinivasan</i> discloses that, in Perl, "[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design." <i>Srinivasan</i> at 107-108.	"This returns a list of object references that match the query criteria. Now if you reissue this
U.S. Patent No. 7,426,720							13. A method according to claim 10, further comprising: resolving the class definition.	

quer refei have	query, it is not too much to expect it to return an identical list of objects (the same object
have	references, that is). This means that Adaptor has to keep an in-memory cache of objects that
COTT	have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
Υhe	"The Adaptor: : File module does not have this problem because it maintains a list of all
obje	bjects given to its store () method (for reasons to be explained in the next section); hence accessive identical queries return identical lists.
Que	Queries
One	One big reason why object-oriented databases haven't caught on is the lack of a query
lang	language (or at least a standard query language). When you have a million objects in the
data	base, it would be a terrible thing to load every single object in memory to see whether it
mate	matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates
duer	ies to equivalent SQL queries, while Adaptor: : File implements a simple-minded
sche	scheme for file based objects: it converts the query expression to an evalable Perl expression
and	cycles through all objects, matching them against the query specification.
Sche	Schema Evolution
Let u	us say you have sent your objects' data to a file, and tomorrow, some more attributes are
adde	added to the object implementation. The schema is said to have evolved. The framework has
to be	to be able to reconcile old data with newer object implementations.
Impl	lementation
This	This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
only	only the key procedures that perform query processing and file or database I/O. Pay as much
atter	attention to the design gotchas and unimplemented features as you do to the code.
Aua	
An /	An Adaptor::File instance represents all objects stored in one file. When this adaptor is
Creat	ted (using new), it reads the entire file and translates the data to in-memory objects.
Slur	Slurping the entire file into memory avoids the problem of having to implement fancy
on-d	lisk schemes for random access to variable-length data; after all, that is the job of DBM
and	database implementations. For this reason, this approach is not recommended for large
num	numbers of objects (over 1,000, to pick a number).
The	he file adaptor has an attribute called all instances, a hash table of all objects given to its store

Srinivasan in view of APA-Bach method (and indexed by their _id), as shown in Figure 11-2.	File Adaptiv	Figure II-3. Spricture of file adapter	<i>Srinivasan</i> at 179-80.	"Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." <i>Srinivasan</i> at 184.	"retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other" <i>Srinivasan</i> at 186.	<i>Srinivasan</i> provided a method for maintaining the source definition as a class file on at least one of a local and remote file system. <i>Srinivasan</i> discloses, for example, that "[i]n a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this" <i>Srinivasan</i> at 134.	"You will not merely dabble with language syntax or the APIs of different modules as you read this book. You will spend just as much time dealing with real-world issues such as avoiding deadlocks during remote procedure calls and switching smoothly between data storage using a flat file or a database." <i>Srinivasan</i> at <i>xi</i> .
U.S. Patent No. 7,426,720						14. A method according to claim 10, further comprising: maintaining the source definition as a class file on at least one of a local and remote file system.	

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	"This technique is often employed in the guts of client/server libraries. In a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this" <i>Srinivasan</i> at 134.
	"This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Oneries
	One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression
	and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations.
	Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code.

<i>Srinivasan</i> in view of <i>APA-Bach</i> discloses handling multiple clients by "[c]reat[ing] multiple threads of control." <i>Srinivasan</i> at 193. In order to create multiple threads of control, <i>Srinivasan</i> invokes the fork() system call to create "process-level parallelism." <i>See id.</i> at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." <i>See id.</i> Thus, <i>Srinivasan</i> discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process.	"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports $fork$, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:	<pre># Forking server use IO:: Socket; \$SIG{CHLD} sub {wait ()}; \$SIG{CHLD} sub {wait ()}; \$main_sock = new IO::Socket::INET (LocalHost => 'goldengate', LocalPort => 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { while (defined (\$buf <\$new_sock>) while (defined (\$buf <\$new_sock>) while (defined (\$buf <\$new_sock>) while (defined (\$buf <\$new_sock>) % this while (defined (\$buf <\$new_sock>) % this which something with \$buf</pre>	print \$new sock "You said: \$buf\n";
U.S. Patent No. 7,426,720 copying the memory space of the master runtime system process into a separate memory space for the child runtime system process.			

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	<pre>provide the parent process exits when it is done. # else 'tis the parent process, which goes back to accept() } close (\$main_sock);</pre>
	The $fork$ call results in two identical processes-the parent and child-starting from the statement following the $fork$. The parent gets a positive return value, the process ID (pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client.
	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute exit(), or a termination status otherwise), just in case the parent uses wait or waitpid to enquire about this status. The terminated child process is
	also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.
	This limitation is also disclosed by <i>APA-Bach</i> , as below:
	"The only way for a user to create a new process in the UNIX operating system is to invoke the <i>fork</i> system call." $APA-Bach$ at 192.
	"The <i>copy-on-write</i> bit, used in the <i>fork</i> system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." <i>APA-Bach</i> at 287.

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute exit(), or a termination status otherwise), just in case the parent uses wait or waitpid to enquire about this status. The terminated child process is also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular." <i>Srinivasan</i> at 195.
17. A method according to claim 10, wherein the object-oriented program code is written in the Java programming language.	<i>Srinivasan</i> provided a method wherein the object-oriented program code is written in the Java programming language. For example, <i>Srinivasan</i> discloses that "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both." <i>Srinivasan</i> at 98.
	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'II learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it, everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." <i>Srinivasan</i> at 98.

U.S. Fatelit No. 7,420,720	Srinivasan in view of APA-Bach
	"The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (<i>toke.c.</i>), the yacc-based parser (perly.y), and the code generator (op.c). Regular expressions-which form a distinct sublanguage-are recognized in <i>toke.c</i> and compiled to an internal format in <i>regcomp.c.</i> Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes and substitution, chop, push, index, <i>r</i> ^{±index} , <i>r</i> ^{±index} , greep, [*] and so on, which explains why there are 343 opcodes as of this writing! It as so explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." <i>Srinivasan</i> at 323-24.
19. A computer-readable storage medium holding code for performing the method according to claim 10.	<i>Srinivasan</i> provided a computer-readable storage medium holding code for performing the method according to claim 10. For example, <i>Srinivasan</i> references an "in-memory cache of objects." <i>Srinivasan</i> at 178. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.

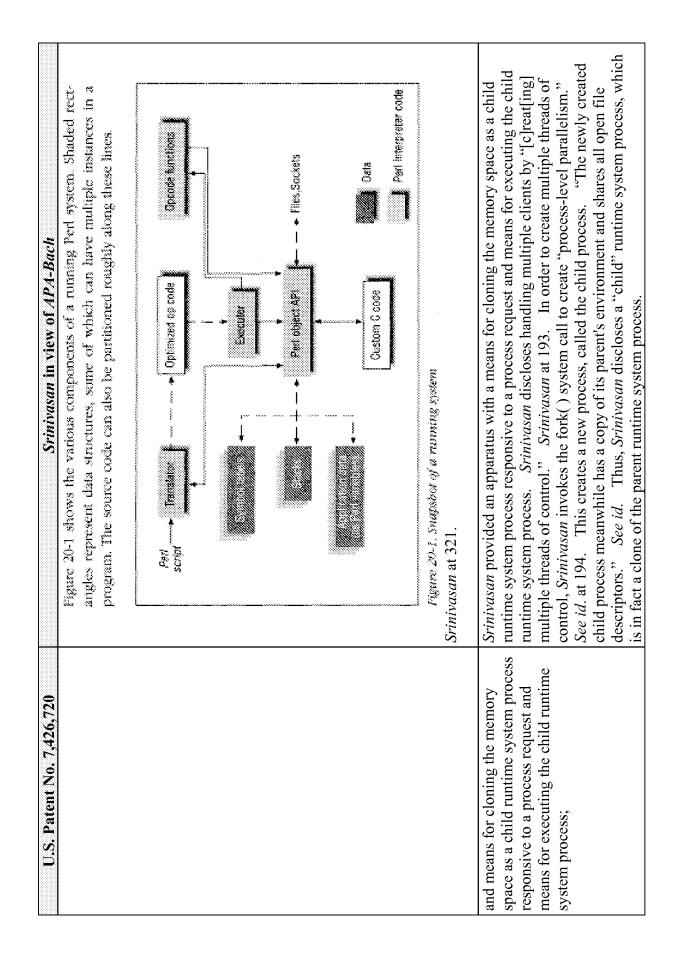
"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for teasons to be explained in the next section), hence successive identical queries return identical lists. Queries . One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database. Adaptor:: DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects, it converts the query very expression to an evalable. Peri expression and cycles through all objects, matching them against the query specification. Schema Evolution Schema Evolution Equation Whore conjects data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation. The section explains the implementation. The section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover object implementations. This section explains the implementation. The schema is add to the design greehas and unimplemented features as you data schema subtometers for a database. Joing the key procedures that perform query processing and file or database I/O. Pay as much adaptor::File. Instance represents all objects schema file or database I/O. Pay as much adaptor::File. Instance represents all objects schema file or database I/O. Pay as much attention to the design greehas and unimplemented features as you do to the code: Adaptor::File. Instance represents all objects schema file or database I/O. Pay as much attention to the design greehas and unimplemented features are obted. The file adaptor: included the adamase adaptor is an unders? Supplementation transplane the entire file and translates the data is included to the objects (oret 1,000, to pick a nunder).	U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
objects given to its store () method (for reasons to be explained in the next section successive identical queries return identical lists. Queries: Queries: Queries: Dubbig reason why object-oriented databases haven't caught on is the lack of a quary spin objects and approxed in the a terrible thing to load every single object in memory to see matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to oquivalent SQL queries, while Adaptor::File implements a simple- min scheme for file based objects: it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Scheme Forline and cycles through all objects data to a file, and tomorrow, some more att and cycles through all objects, matching them against the query specification. If say you bave sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fran to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor:: DBI and Adaptor:: File. We only the ksy procedures that perform query processing and file or database for able to ::File An Adaptor:: File An Adaptor:: File An Adaptor:: File instance represents all objects stored in one file. When this adaptor: for disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for muchts schemes for random access to variable-length data; after all, that is the job and database implementations. The file adaptor shore is not recommended for muchts schemes for random access to variable-length data; and it objects given the file adaptor has an attribute called all instances, a hash table of all objects given	"The Adaptor: : File	module does not have this problem because it maintains a list of all
Successive identical queries return identical lists. Successive identical queries return identical lists. One big reason why object-oriented databases haven't caught on is the lack of a queries of the big reason why object-oriented databases haven't caught on is the lack of a quartes your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor: :File implements a simple-min scheme for file based objects: it converts the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fram to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database <i>JO</i> . Pr attention to the design gotchas and unimplemented for the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor evented (using wow), it reads the entire file and translates the data to im-memory of Slurping the entire file into memory avoids the problem of having to implement fo on-disk schemes for random access to variable-length data; after all, that is the job and databose: fobjects (over 1.000, to pick a mumber). The file adaptor has an attribute called all_instances, a hash table of all objects given	objects given to its	ore () method (for reasons to be explained in the next section); hence
Queries One big reason why object-oriented databases haven't caught on is the lack of a q language (or at least a standard query language). When you have a million objects database, it would be a terrible thing to load every single object in memory to see matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor:: File implements a simple-min scheme for file based objects; it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fran to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::File. We only the key procedures that perform query processing and file or database I/O. P attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement for on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects and the piece and beyer by and by ander stored for numbers of objects and the subset.	successive identical	queries return identical lists.
One big reason why object-oriented databases haven't caught on is the lack of a quadbase, it would be a terrible thing to load very language). When you have a million objects database, it would be a terrible thing to load very single object in memory to see matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor::File implements a simple-mili scheme for file based objects; it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have cvolved. The fram to be able to reconcile old data with newer object implementations. Implementation. This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database. Jo. Pt attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File instance represents all objects stored in one file. When this adapt created (using new), it reads the entire file and translates the data to in-memory ob Slurping the entire file into memory avoids the problem of having to implement for and database implementations. The scheme all objects stored in one file. When this adapt created (using new), it reads the entire file and translates the data to in-memory ob Slurping the entire file into memory avoids the problem of having to implement for and database. Adaptor: File would near store the and translates the data to in-memory ob subset implementations. For this reason, this approach is not recommended for mumbers of objects and tuning the entire file into memory avoids the problem of having to implement for and database implementations. For this reason, this approach is not recommended for mumbers of objects have the southed and translates the data to in-memory observerse inth	Queries	
language (or at least a standard query language). When you have a million objects database, it would be a terrible thing to load every single object in memory to see matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor::File implements a simple-min scheme for file based objects: it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The frant to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation. This section explains the implementation of Adaptor::File. We only the key procedures that perform query processing and file or database I/O. Prattention This section explains the entire file and translares as you do to the code. Adaptor::File instance represents all objects stored in one file. When this adapter::File instance represents all objects stored in one file. When this adapter::File instance represents all objects stored in one file. When this adapter::File and translares the edite to implement for on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file dataptor is not recommended for numbers of objects (over 1,000, to pick a number).	One big reason why	object-oriented databases haven't caught on is the lack of a query
database, it would be a terrible thing to load every single object in memory to see matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor: File implements a simple-min scheme for file based objects; it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fran to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database. I/O. P attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory ob Slurping the entire file into memory avoids the problem of having to implement fi on database for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of fobjects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	language (or at leas	a standard query language). When you have a million objects in the
matches your criteria; this is a job best left to the database. Adaptor::DBI simply queries to equivalent SQL queries, while Adaptor:: File implements a simple-min scheme for file based objects: it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Externa Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fram to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database I/O. Pr attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File Adaptor::File instance represents all objects stored in one file. When this adaptor::File instance represents all objects stored in one file. When this adaptor::File instance represents all objects stored in one file. When this adaptor::File on-disk schemes for random access to variable-length data; after all, that is the job and database informations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	database, it would t	a terrible thing to load every single object in memory to see whether it
queries to equivalent SQL queries, while Adaptor: : File implements a simple-min scheme for file based objects: it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fran to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::Eile. We only the key procedures that perform query processing and file or database I/O. Pi attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File Adaptor::File Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of flaving to implement fo on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	matches your criter	; this is a job best left to the database. Adaptor::DBI simply translates
 scheme for file based objects: it converts the query expression to an evalable Perl and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fram to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database J/O. Pt attention This section explains the implementation of Adaptor::DBI and Adaptor::File. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adapter::File An Adaptor::File instance represents all objects stored in one file. When this adapter::File instance file internet of the and translates the data to in-memory of Slurping the entire file instance represents all objects stored in one file. When this adapter::File on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). 	queries to equivaler	SQL queries, while Adaptor: : File implements a simple-minded
and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fram to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::File. We only the key procedures that perform query processing and file or database I/O. Pe attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to im-memory of Slurping the entire file into memory avoids the problem of flaving to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	scheme for file base	l objects: it converts the query expression to an evalable Perl expression
Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The fran to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor:::DBI and Adaptor:::File. We only the key procedures that perform query processing and file or database I/O. Pa attention to the design gotchas and unimplemented features as you do to the code. Adaptor:::File An Adaptor:::File An Adaptor:::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to im-memory of Slurping the entire file into memory avoids the problem of having to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	and cycles through	Il objects, matching them against the query specification.
Let us say you have sent your objects' data to a file, and tomorrow, some more att added to the object implementation. The schema is said to have evolved. The frar to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database J/O. Pi attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory ol Slurping the entire file into memory avoids the problem of having to implement fa on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	Schema Evolution	
added to the object implementation. The schema is said to have evolved. The frar to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database J/O. Pi attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory ol Slurping the entire file into memory avoids the problem of having to implement fe on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	Let us say you have	sent your objects' data to a file, and tomorrow, some more attributes are
to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database <i>I</i> /O. Pa attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory ol Slurping the entire file into memory avoids the problem of having to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	added to the object	nplementation. The schema is said to have evolved. The framework has
Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database I/O. Pe attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	to be able to reconc	e old data with newer object implementations.
This section explains the implementation of Adaptor::DBI and Adaptor::File. We only the key procedures that perform query processing and file or database I/O. Pa attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File and Adaptor::File instance represents all objects stored in one file. When this adapter adaptor::File instance represents all objects stored in one file. When this adapter adaptor::File instance represents all objects stored in one file. When this adapter adaptor::File instance represents all objects stored in one file. When this adapter adaptor::File instance represents all objects stored in one file. When this adapter adapter instance represents all objects stored in one file. When this adapter adapter data to in-memory of Slurping the entire file into memory avoids the problem of having to implement file on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	Implementation	
only the key procedures that perform query processing and file or database I/O. Pa attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fa on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	This section explair	the implementation of Adaptor::DBI and Adaptor::File. We will cover
attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	only the key proced	res that perform query processing and file or database I/O. Pay as much
Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adap created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fa on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	attention to the desi	n gotchas and unimplemented features as you do to the code.
An Adaptor::File instance represents all objects stored in one file. When this adaption created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fa on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	Adaptor::File	
created (using new), it reads the entire file and translates the data to in-memory of Slurping the entire file into memory avoids the problem of having to implement fi on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	An Adaptor::File in	tance represents all objects stored in one file. When this adaptor is
Slurping the entire file into memory avoids the problem of having to implement fa on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	created (using new)	it reads the entire file and translates the data to in-memory objects.
on-disk schemes for random access to variable-length data; after all, that is the job and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	Slurping the entire	le into memory avoids the problem of having to implement fancy
and database implementations. For this reason, this approach is not recommended for numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	on-disk schemes for	random access to variable-length data; after all, that is the job of DBM
numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given	and database implem	intations. For this reason, this approach is not recommended for large
The file adaptor has an attribute called all_instances, a hash table of all objects given	numbers of objects (ver 1,000, to pick a number).
	The file adaptor has	n attribute called all_instances, a hash table of all objects given to its store
method (and indexed by their 1d), as shown in Figure 11-2.	method (and indexed	by their _id), as shown in Figure 11-2.

U.S. Patent No. 7,426,720 U.S. Patent No. 7,426,720
--

Semp = Employee->new ("Ada", 3 5);Semp->set_salary(1000);	<i>See id.</i> at 390.	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of	publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." <i>Srinivasan</i> at 98.	"Object orientation (OO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chapter, we will study what the noise is all about and build objects using Perl. I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." <i>Srinivasan</i> at 99.	"Objects of a certain type are said to belong to a <i>class</i> ." <i>Srinivasan</i> at 101. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises a
U.S. Patent No. 7,426,720					

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	hand-coded lexer (<i>toke.c</i>), the yacc-based parser (perly.y), and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in <i>toke.c</i> and
	compiled to an internal format in <i>regcomp.c</i> . Opcodes are similar in concept to machine code: while machine code is executed by hardware, oncodes (sometimes called byte-codes)
	or p-code) are executed by a "virtual machine." The similarity ends there. Modern interneters never emulate the workings of a hardware CDI1 for nerformance reasons
	Instead, they create complex structures primed for execution, such that each opcode directly
	contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually
	embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes
	resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction A large number of these oncodes directly correspond to the facilities available at
	the scripting level, such as regular expression matching and substitution, chop, push, index,
	rindex, grep,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the
	work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Derl-to-lava hyte-code translator: there is no correspondence between the two sets."
	Srinivasan at 323-24.
	"Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of this writing. It can be asked to translate a script to C code, which can be compiled to form an
	executable; as it happens, this executable is not much faster than the interpreted script, because most of the action still takes place in opcode functions as they exist now. Static
	typing hints may usher in some aggressive optimizations. For example, if you say:
	the compiler would use C's native integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler can alternatively produce a byte-code file and have
	the interpreter $eval$ it subsequently, similar to the facilities provided by Python and Java. It also supports much better debugging options than those currently provided with $-D$."
	Srinivasan at 370.

Srinivasan in view of APA-Bach	<i>Srinivasan</i> provided an apparatus with a means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process. <i>Srinivasan</i> discloses that, in Perl, "[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design." <i>Srinivasan</i> at 107-108.	"The data structures described above are normally kept in global C variables. If Perl is compiled with <i>-DMULTIPLICITY</i> , it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." <i>Srinivasan</i> at 323.
U.S. Patent No. 7,426,720	means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process;	



"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports $fork$, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:	<pre># Forking server use IO:: Socket; \$SIG{CHLD} sub {wait ()}; \$SIG{CHLD} sub {wait ()}; \$main_sock = new IO::Socket::INET (LocalHost => 1200, LocalPort => 1200, Listen => 5, Proto => 'tcp', Reuse => 1, ?; die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process which (\$bif <\$new_sock "You said: \$buf\n"; } print \$new_sock "You said: \$buf\n"; } } The fork (0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() } close (\$main_sock); } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid)</pre>
U.S. Patent No. 7,426,720	

7 H V		<i>Srinivasan</i> in view of <i>APA-Bach</i> of the child process check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute exit (), or a termination status otherwise), just in case the process exits (or terminates process, and it is always a good thing to remove it using wait; otherwise, the process lately keep filling up with junk. In the preceding oode, wait doesn't block, because it is called only when we know for sure that a child process has diod-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and sIGCHLD in particular.
of the master runtime system process "The only way for a user to create a new process in the UNIX operating system is to invoke	sf Is ce	<i>IPA-Bach.</i> One of ordinary skill in the art at the time of the invention, seeking to reduce a impact of a fork() system call on local memory, would look to the disclosure of <i>IPA-Bach</i> for a description of the copy-on-write bit. The only way for a user to create a new process in the UNIX operating system is to invoke

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	the <i>jork</i> system call. AFA-Bach at 192. "The <i>copy-on-write</i> bit, used in the <i>fork</i> system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." APA-Bach at 287.
	"9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the <i>fork</i> system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call <i>exec</i> soon after the <i>fork</i> call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply
	increments the region reference count of shared regions The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private
	regions of the parent and child processes during <i>fork</i> . If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." <i>APA-Bach</i> at 289–90.
to claim 1, source controller level resource s on the child	<i>Srinivasan</i> provided an apparatus with a resource controller to set operating system level resource management parameters on the child runtime system process. For example, "[o]n Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it." <i>Srinivasan</i> at 195.
	"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports fork, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is

		-
Ι τουνίανοι αρουνιαίνα γιαι τι τναι τι τνιαίτιο α οπιματι απιναπι να τατατιταίναι γιαν ναι οιαίαιο τη	Ιτουνίτνο αρουνιαίνα ψτιτι τι, σαι τι τνιαιτιρ α ριτιατι απινατι τα τιτνιτιταίναι γιαν νατι ριαίας τ	Ιτουνιάνου αρουντάνου τη τη τη τη τη τη τητάτη α ομιάτι αποινάτη τη τητοιτητάτου γαιν νότι οιαίας τ
TESOULCES associated with it. Dut it tetallis a shiali alloult of hitothianoh (hie exit status li	TESOULCES associated with it. Dut it tetaills a sitial allouit of information (the exit status it	[τενουτέες associated while it. Dut it tetalls a shial allount of πησιμιαιοίη (μις έλιε status it
resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if
resources associated with it. But it retains a small amount of information (the exit status it	resources associated with it. But it retains a small amount of information (the exit status it	resources associated with it. But it retains a small amount of information (the exit status if
resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if
resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if
resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if	resources associated with it. But it retains a small amount of information (the exit status if
process exits (or terminates abnormany), the system gets no of the memory, mes, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormany), the system gets no of the memory, mes, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormany), the system gets no of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
of the child process. Bout processes check this return value and execute their own rougle, the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Bout processes check this return value and execute their own rogic, the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Bout processes check this return value and execute their own rogic, the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Increase control in two numbers in two numbers processes-up parent and curre-starting nom up statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Increase control in two numbers in two numbers processes-up parent and curre-starting nom up statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Increase control in two numbers in two numbers processes and child statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (p_{pid}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (p_{pid}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (p_{pid}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The $fork$ call results in two identical processes-the parent and child-starting from the statement following the $fork$. The parent gets a positive return value, the process ID (p_{pid}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The $fork$ call results in two identical processes-the parent and child-starting from the statement following the $fork$. The parent gets a positive return value, the process ID (p_{pid}) of the child process. Both processes check this return value and execute their own logic; the main process goes back to $accept$, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	 close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if	The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
 ¹ close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 ¹ close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 ¹ close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
<pre># else 'tis the parent process, which goes back to accept()</pre>	<pre># else 'tis the parent process, which goes back to accept()</pre>	<pre># else 'tis the parent process, which goes back to accept()</pre>
<pre># else 'tis the parent process, which goes back to accept() } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>	 # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
 # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the menory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
 exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
 exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	 exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if 	<pre>exit(0); # Child process exits when it is done.</pre>
<pre>pxit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() j) close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value, the process ID (\$pid) of the child process goes back to accept, and the child process reads a line from the socket and main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>	<pre># exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() j) close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value, the process ID (\$pid) of the child process goes back to accept, and the child process reads a line from the socket and main process goes back to accept, and the child process reads a line from the socket and cehoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>	<pre># exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() j close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value, the process ID (\$pid) of the child process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>
<pre></pre>	<pre></pre>	<pre></pre>
<pre></pre>	<pre></pre>	<pre></pre>
<pre>print \$new_sock "You said: \$buf\n";</pre>	<pre>print \$new_sock "You said: \$buf\n";</pre>	<pre>print \$new_sock "You said: \$buf\n";</pre>
<pre>print \$new_sock "You said: \$buf\n";</pre>	<pre>print \$new_sock "You said: \$buf\n";</pre>	<pre>print \$new_sock "You said: \$buf\n";</pre>
<pre>print \$new_sock "You said: \$buf\n";</pre>	<pre># do someting with \$bufhn";</pre>	<pre>print \$new_sock "You said: \$buf\n";</pre>
<pre># do something with \$buf print \$new_sock "You said: \$buf\n";</pre>	<pre># do something with \$buf print \$new_sock "You said: \$buf\n";</pre>	<pre># do something with \$buf, print \$new_sock "You said: \$buf.n"; } print \$new_sock "You said: \$buf.n"; } aclose (\$main_sock); The fork call results in two identical processs, which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and excente their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if</pre>
<pre># do something with \Spuf print \Snew_sock "You said: \Spuf\n";</pre>	<pre># do something with \Spuf print \Snew_sock "You said: \Spuf\n";</pre>	<pre># do something with \Spuf print \Snew_sock "You said: \Spuf\n";</pre>
<pre>will construct the parent control control shuft # do something with Shuft print Shew_sock "You said: Shuft.n"; print Shew_sock "You said: Shuft.n"; # else 'tis the parent process exits when it is done. # else 'tis the parent process which goes back to accept()) close (\$main_sock); The fork call results in two identical processes the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>	<pre>miller (definition of the state (provide the state))</pre>	<pre>will define the state of the state of the state state in the stat</pre>
<pre>while (defined (\$buf <\$nww_sock*)</pre>	<pre>while (defined (\$buf <\$new_sock*)</pre>	<pre>while (defined (\$buf <\$nww_sock*)</pre>
<pre>while (defined (Sbuf <\$new sock>) # do something with \$buf print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; } t = do something with \$buf print \$new_sock "You said: \$buf\n"; print "= do something with \$buf # else 'tis the parent process exits when it is done. # else 'tis the parent process, which goes back to accept() } Close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the socket and choos it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information iter exit status if resources associated with it. But it retains a small amount of information iter exit status if resources associated with it. But it retains a small amount of information iter exit status if resources associated with it. But it retains a small amount of information iter exit status if resources associated with it. But it retains a small amount of information retain the status if</pre>	<pre>while (defined (\$buf <\$new sock») # do something with \$buf print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; p) t = 1000 (\$n + Child process exits when it is done. # else 'tis the parent process, which goes back to accept() f close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process Both processes check this return value, the process ID (\$pid) of the child process goes back to accept, and the child process reads a line from the socket and choes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information it and the socket and resources associated with it. But it retains a small amount of information it and retains a small amount of information it and retains a small amount of information it and retains a status it and</pre>	<pre>while (defined (Sbuf <\$new sock>) # do something with \$buf print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; p) t = lse 'tis the parent process exits when it is done. # else 'tis the parent process, which goes back to accept() p) close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process Both processes check this return value, the process ID (\$pid) of the child process goes back to accept, and the child process reads a line from the socket and cchoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information it and resources associated with it. But it retains a small amount of information it status if retains a small amount of information it and retains a small amount of information it and retains a status it retains a status it</pre>
<pre>while defined (\$buf <\$new_sock>) while defined (\$buf <\$new_sock>) print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; perint \$new_sock "You said"; process exits (or terminates abnormally, the system gets rid of the menory, files, and other resources associated with "Put it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information "You said"; retains a small suck "You said"; retains a sm</pre>	<pre>while defined (\$buf <\$new_sock>) while defined (\$buf <\$new_sock>) while defined (\$buf <\$new_sock>) print \$new_sock "You said: \$buf\n";</pre>	<pre>while (defined (sbuf <\$new_sock*)) # do something with \$buf print \$new_sock "You said: \$buf/n";</pre>
<pre># Child process while (defined (\$buf <\$new_sock") # do something with \$buf; print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; exit(0); # Child process exits when it is done. # else 'tis the parent process which goes back to accept() j close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status if resources associated with it. But it network and low of the crit status it network</pre>	<pre># Child process while (defined (\$buf <\$new_sock") # do something with \$buf # the something with \$buf # print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; exit(0); # Child process exits when it is done. # else 'tis the parent process which goes back to accept() j close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it network and low the child proces are status if resources associated with it. But it network and low the child mount of information (the exit status if resources associated with it. But it network and low the child mount of information (the exit status if resources associated with it. But it network and low the child mount of information (the exit status if resources associated with it. But it was a stall amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if retains a small status if the status if retains a small status if the status if</pre>	<pre># Child process while (defined (\$buf <\$new_sock") # do something with \$buf; print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; exit(0); # Child process exits when it is done. # else 'tis the parent process which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both process check this return value and execute their own logic; the main process goes back to accept, and the child process reads a line from the socket and cchoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the menory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it network and lamount of information (the exit status if resources associated with it. But it networks and lamount of information (the exit status if resources are associated with it. But it networks and and and and and and and and and and</pre>
<pre># Child process while (defined (\$buf <\$new_sock»)</pre>	<pre># Child process while (defined (\$buf <\$new_sock»)</pre>	<pre># Child process while (defined (\$buf <\$new_sock»)</pre>
<pre>the relation of the second state should be a something with Sbuf # control state should the sprint show sock "You said: Sbuf/n"; # control sprint Show sock "You said: Sbuf/n"; # exit(0); # child process exits when it is done. # else 'tis the parent process which goes back to accept()</pre>	<pre></pre>	<pre>try type = 0 to something with %buf # control of the (defined (%buf <%new sock) # conservation (%buf <%new sock) # conservation %buf for the formation of the form for the process exits when it is done. # exit(0); # child process exits when it is done. # else 'tis the parent process which goes back to accept() close (%main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (%pid) of the child process. Both process check this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>
<pre>if (\$pid == 0) { # Child process while described (\$buf <\$new_sock») # Child process which \$buf print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; print process exits when it is done. # else 'tis the parent process which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value, the process ID (\$pid) of the child process both processes check this return value and execute their own logic; the main process post to accept, and the child process reads a line from the socket and cchoes it back to the client. Incidentally, the cHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>	<pre>if (\$pid == 0) { # Child process while defined (\$but <\$new_sock») # Child process which \$buf print \$new_sock "You said: \$buf\n"; process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process Both processes check this return value. The process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>	<pre>if (\$pid == 0) { # Child process while (\$but <\$new_sock») # Child process which gbuf print \$new_sock "You said: \$buf\n"; process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process which goes back to accept() close (\$main_sock); The fork call results in two identical process-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process Both processes check this return value and execute their own logic; the main process pose back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the menory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>
<pre>if (\$pid == 0) { # Child process while (cffind (\$nd \$nd \$nd \$nd \$nd \$nd \$nd \$nd \$nd \$nd</pre>	<pre>if (\$pid == 0) { # Child process while (\$bid \$\number \$\numbe</pre>	<pre>if (\$pid == 0) { # Child process while (cffined (\$ht {\$new_sock}) # Child process while (defined (\$ht {\$new_sock}) print \$new_sock "You said: \$buf\n"; process exits when it is done. # else 'tis the parent process which goes back to accept() close (\$main_sock); The fork call results in two identical processes the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and execute their own logic; the main process lose to accept, and the child process reads a line from the socket and cchoos it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>
<pre>duf (\$pid = 0) { if (\$pid = 0) { f (\$pid = 0) { # Child process while (defined (\$buf <\$new sock)) # do something with \$buf # to so something with \$buf # to so something with \$buf # to so so which goes back to accept () * else 'tis the parent process which goes back to accept () close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process Both processes check this return value, the process ID (\$pid) of the child process cock to accept, and the child process reads a line from the socket and choose it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>	<pre>dif (\$pid=0) f</pre>	<pre>dif (\$pid=0) f if (\$pid=0) f</pre>
<pre>die "cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { f (fild process while (defined (\$buf <\$new_sock)) # do something with \$buf print \$new_sock "You said: \$buf\n"; p) print \$new_sock "You said: \$buf\n"; p) exit(0); # Child process exits when it is done. # else 'tis the parent process which goes back to accept() j) close (\$main_sock); The fork call results in two identical processe-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and exceute their own logic; the main process goes back to accept, and the child process reads a line from the socket and cchocs it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it but it retains a status if retains a status if resources associated with it retains a statu</pre>	<pre>die "cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process while (defined (\$buf <\$new_sock») # do something with \$buf print \$new_sock "You said: \$buf/n"; p) print \$new_sock "You said: \$buf/n"; p) exit(0); # Child process exits when it is done. # else 'tis the parent process which goes back to accept() p) close (\$main_sock); The fork call results in two identical processe-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process. Both processes check this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it beta in the status if retains a status in the status if resources associated with it beta in</pre>	<pre>die "cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { f (fild process while (defined (\$buf <\$new_sock)) # do something with \$buf print \$new_sock "You said: \$buf\n"; pint \$new_sock "You said: \$put\n"; pint \$new_soc</pre>
<pre>die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process # Child process # Date (Gefined (\$buf <\$new sock)) # do something with \$buf, print \$new_sock "You said: \$buf\n"; p) print \$new_sock "You said: \$buf\n"; p) f do something with \$buf, pint \$new_sock "You said: \$buf\n"; p) f do something with \$buf, pint \$new_sock "You said: \$buf\n"; p) f alse 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # for formation for \$cores, which goes back to accept() of the child process back this return value and excent their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if </pre>	<pre>die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # Child process while (defined (\$buf <\$new sock)) # do something with \$buf print \$new_sock "You said: \$buf\n"; p) print \$new_sock "You said: \$buf\n"; p) close (\$main_sock); The for (0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept() p) close (\$main_sock); The fork call results in two identical processes the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process back to accept, and the child process check this return value, the process ID (\$pid) of the child process check this return value, the process ID (\$pid) of the child process check this return value, the process ID (\$pid) of the child process check this return value, the process ID (\$pid) process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources areasociated with it but is the stated acceses and accese acces</pre>	<pre>die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) (# Child process while defined (\$buf <\$new sock)) # Child process which \$puf, print \$new_sock "You said: \$buf\n";</pre>
<pre>vprof for any formulation of the file "control for the "control for the "control for the file (\$pid == 0) { if (\$pid == 0) { # Child process while (defined (\$buf <\$nw_sock)) # do something with \$buf \n"; print \$new_sock "You said: \$buf \n"; process exits the parent process exits when it is done. # else 'tis the parent process exits when it is done. # else 'tis the parent process exits when it is done. # for for the process back the recess the parent and child-starting from the statement following the for. The parent gets a positive return value, the process ID (\$pid) of the child process back the sceles this return value and excente their own logic; the main process back to accept, and the child process reads a line from the socket and choes it back to the client. hoidentally, the crIID signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if } } </pre>	<pre>vprot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) {</pre>	<pre>vprove fork: \$!" unless defined (\$pid) ; if (\$pid == 0) {</pre>
<pre>\$\frac{\$\partial{PD}{Plane} = 10^{f}(); die "Cannot fork: \$!" unless defined (\$pid); if (\$pid==0); # Child process while (defined (\$buf <\$new_sock*) # do something with \$buf print \$new_sock "You said: \$buf\n"; print \$new_sock "You said: \$buf\n"; process exits when it is done. # else 'tis the parent process which goes back to accept() close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and choes it back to the client. Incidentally, the cHILD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a stared approxement of the other status if resources associated with it. But it retains appr</pre>	<pre>\$</pre>	<pre>\$</pre>
<pre>\$pid = fork();</pre>	<pre>\$pid = fork();</pre>	<pre>\$pid = fork();</pre>
<pre>prove the state of the sta</pre>	<pre>prove the state of the sta</pre>	<pre>print for the state of the</pre>
<pre>while (Snew sock = Smain_sock->accept()) { Spid = fork(); Spid = fork(); the cork (Struct Struct (Spid);</pre>	<pre>while (Snew sock = Smain_sock->accept()) { Spid = fork(); dia "cannot fork: \$!" unless defined (Spid) ; if (Spid == 0) { # Child process while (defined (Sbuf <snew socks))<br=""># do something with Sbuf print Snew_sock "You said: Sbuf/n"; print Snew_sock "You said: Sbuf/n"; print Snew_sock "You said: Sbuf/n"; print Snew_sock "You said: Spuf/n"; print Snew_sock "You said: Spuf(n"; print Snew_sock "You said: Spuf(n"; process check this return value, the process ID (Spid) of the following the fork. The parent gets a positive return value, the process ID (Spid) of the child process check this return value, the process ID (Spid) of the child process check this return value, the process ID (Spid) of the child process cose check this return value, the process ID (Spid) of the child process cose check this return value, the socket and choes it back to the client. Incidentally, the critD signal has nothing to do with IPC per se. On Unix, when a child process extist with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if</snew></pre>	<pre>while (Snew sock = Smain_sock->accept()) (</pre>
<pre>while (\$new_sock = \$main_sock->accept()) { Spid = fork(); Spid = fork(); Spid = fork(); spid = fork(); "totant fork: \$!" unless defined (\$pid) ; "for (\$pid == 0) { "for (\$pid for (\$pid ?), " "for (\$p</pre>	<pre>while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>while (\$new_sock = \$main_sock->accept()) { Spid = fork(); Spid = fork(); spid = fork(); if (\$pid == 0) { # Child process while (defined (\$puf <\$new_sock)) # Child process while (defined (\$puf <\$new_sock)) # Child process sxifts when it is done. # else 'tis the parent process, which goes back to accept()) close (\$main_sock); The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process between their own logic; the main process goes back to accept, and the child process reds a line from the socket and coloes it back to the client. Incidentally, the citLD signal has nothing to do with IPC per se. On Unix, when a child process associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information (the cxit status if resources associated with it. But it retains a small amount of information retains a status in retains a status in retains a status in retains a status in retains a status in</pre>
<pre>while (\$new_sock = \$main_sock>baccept()) { Sind = fork(); tit (\$pid = 0) {</pre>	<pre>while (\$new_sock = \$main_sock>baccept()) { Sidd = fork(); till = fork(); tilll = fork(); tilll = fork(); tilll = fork()</pre>	<pre>while (\$new_sock = \$main_sock=>accept()) { Sidd = fork(); tid = fork();</pre>
<pre>def = fork(); while (fshew sock = smain_sock->accept()) { spid = fork(); if (fshew sock = smain_sock->accept()) { ff (fshew sock = smain_sock->accept()) { ff (fshew sock = smain sock = sock); ff (fshew sock = fork (); ff (fshew sock = fork (); ff (fshew sock = fork fshew sock); ff (fshew sock = fork = fork fshew sock = fork fshew sock = fork fshew sock = fork = fork fshew sock = fork = fork fshew sock = fork fshew sock = fork = fork fshew sock = fork = fork fshew sock = fork f</pre>	<pre>due "Scoke" cound not be created: while (fenew sock = \$main_sock->accept()) { spid = fork(); die "cannot fork: \$!" unless defined (\$pid); if (\$pid == 0) { die "cannot fork: \$!" unless defined (\$pid); f (\$pid == 0) { f (\$pid == 0) {</pre>	<pre>due "Scheur cound not be created: Neadon: Wille (Senew sock = Smain_sock->accept()) { while (Senew sock = smain_sock->accept()) { spid = fork(); ti (Spid == 0) { dia "cannot fork: S!" unless defined (Spid); if (Spid == 0) {</pre>
<pre>die "Socket could not be created. Reason: \$'\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$'\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { spid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ; die "Cannot fork: \$!" unless defined (\$pid) ; die (\$pid == 0) { * # Child process * # child process * # child process * # child process * # child process = stits when it is done. * else 'tis the parent process, which goes back to accept() * else 'tis the parent process, which goes back to accept()</pre>
<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock-accept()) {</pre>
<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = \$main_sock->accept()) {</pre>
<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { %pid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid = 0) { # Child process while (defined (\$puf <\$new_sock)</pre>	<pre>); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (%new_sock = %main_sock->accept()) { \$pid = fork();</pre>	<pre>); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pid = fork();</pre>
<pre>(\$socket could not be created. Reason: \$!\n" unless (\$sock); while (\$spid = fork();</pre>	<pre>(\$socket could not be created. Reason: \$!\n" unless (\$sock); while (\$spid = fork();</pre>	<pre>(\$socket could not be created. Reason: \$!\n" unless (\$sock); while (\$sok sock = \$main_sock->accept()) { \$pid = fork(); if (\$pid = 0) { if (\$pid = 0) {</pre>
<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = fork())</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = fork())</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = fork())</pre>
<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre>Reuse => 1,); end = "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pidd = fork();</pre>	<pre>Russ => 1,); enser some the second not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pid = "Cannot fork: \$!" unless defined (\$pid) ; \$pid = "Cannot fork: \$!" unless defined (\$pid) ; if (\$pid == 0) { # child process defined (\$pid) ; # child process defined (\$pid) ; # child process exits when it is done. # else 'tis the parent process, which goes back to accept()] close (\$main_sock); # else 'tis the parent process, which goes back to accept() close (\$main_sock); The fork call results in two identical processes the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (\$pid) of the child process check this return value and excute their own logic, the main process goes back to accept, and the child process reads a line from the socket and closes (structurd), the system gets rid of the memory. Files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information the part of the part</pre>	<pre>Russ => 1,); such => 1,); such = (Snew sock = Smain_sock->accept()) { Spid = Carnot fork: \$!" unless defined (Spid) ; f (Spid == 0) { # Child procer: >= 0) { # Child proces; "Yuu ness defined (Spid) ; # Child proces; "Yuu said: Shuf.n"; print Snew_sock "You said: Shuf.n"; f else 'tis the parent process exits when it is done. # else 'tis the parent process which goes back to accept() close (Smain_sock); The fork call results in two identical processes the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID (Spid) of the child process clock this return value and excute their own logic, the main process goes back to accept, and the child process reads a line from the socket and choces the to the client. Incidentally, the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information stating at the states and otheres associated with it. But it retains a sm</pre>
<pre>Reuse => 1,</pre>	<pre>Reuse => 1,</pre>	<pre>Reuse => 1,</pre>
<pre>tense = > 1, **********************************</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre>proce > 'rcp', Reuse > 1,</pre>	<pre>proco > 'rcp', Reuse > 1, ;; die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = \$main.sock->accept()) { \$pid = fork(); die "cannot for: \$!" unless defined (\$pid) ; if (\$pid = 0) { f (\$pid process exits when it is done. } f = 1se 'tis the parent process exits when it is done. } f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. } f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is done. f = 1se 'tis the parent process exits when it is own logic, the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates anormally), the system ges rid of the memory. Fils, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>	<pre>rroot > 'rcp',</pre>
<pre>Proto >> 'tcp', Reuse >> 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); %pid= fock(); die "cannot fork: \$!" unless defined (\$pid) ; f (\$pid==0) { f (\$pid==0) { f (\$pid==0) {</pre>	<pre>Profo > 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); %pid = fork(); die "cannot fork: \$!" unless defined (\$pid) ; ff (\$pid = 0) {</pre>	<pre>Profo > 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); %pid = fork(); die "cannot fork: \$!" unless defined (\$pid) ; ff (\$pid = 0) { ff (\$pid = 0) { ff (\$pid = 0) { for something with \$pid, for something with \$pid, print \$new_sock "von said: \$put\n"; print \$new_sock "von said: \$put\n"; print \$new_sock "von said: \$put, felse 'tis the parent process exits when it is done. f else 'tis the parent process exits when it is done. f else 'tis the parent process, which goes back to accept() close (\$main_sock); The fork call results in two identical process check this return value and excent their own logic; the main process goes back to accept, and the child process reads a line from the socket and choosit back to accept, and the child process reads a line from the socket and choces to back to accept, and the child process reads a line from the socket and choces to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child procurces associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if retain the status if the status if the status if the status if the sta</pre>
<pre>Proto => 'tcp',</pre>	<pre>Proto => 'tcp',</pre>	<pre>Proto => 'tcp',</pre>
<pre>proto = 'tcp',</pre>	<pre>proto => 'tcp',</pre>	<pre>proto => 'tcp',</pre>
<pre>proto => 'tcp',</pre>	<pre>proto => 'tcp',</pre>	<pre>prote = > 1,</pre>
<pre>proto => 1,</pre>	<pre>proto => 1,</pre>	<pre>proto => 1,</pre>
<pre>Listen => 5, Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tc</pre>	<pre>Listen => 5, Proto => 'tcp', Proto = 'cond not be created. Reason: \$!\n" unless (\$sock); Proto = 'cond fork: \$!" unless defined (\$pid) ; Proto = 'cond fork: \$!" unless defined (\$pid) ; Proto = 'cond (\$bit (\$sock) * Proto = 'cond (\$bit (\$sock) * Proto = 'cond (\$bit (\$bit (\$sock) * Proto = 'cond (\$bit (\$bit (\$pid = 0) (Proto = 'cond (\$bit (\$pid : \$bit "; Proto = 'cond (\$bit (\$pid : \$bit "; Proto = 'cond (\$bit (\$bit (\$pid : \$bit "; Proto = 'cond (\$bit (\$bit (\$pid : \$bit "; Proto = 'cond (\$bit (\$bit "; Proto = 'cond (\$bit (\$bit "; Proto = 'cond (\$bit (\$bit "; Proto = 'cond "; Proto = 'cond "; Proto = 'cond (\$bit "; Proto = 'cond "; Proto = 'cond</pre>	<pre>Listen => 5,</pre>
<pre>Listen => 5,</pre>	<pre>Listen => 5,</pre>	<pre>Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>
<pre>Listen => 5, Proto => 'tcp', Proto => 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = \$main sock->accept()) {</pre>	<pre>Listen => 5,</pre>	<pre>Listen => 5,</pre>
<pre>die "Socket could not be created. Reasen: \$;\n" unless (\$sock);</pre>	<pre>die "Socket could not be created. Reasen: \$;\n" unless (\$sock);</pre>	<pre>die "Socket could not be created. Reason: \$', "</pre>
<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: %!\n" unless (%sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre>die "socket could not be created: Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre>Internet = > 1200, Intern => 5, Proto => 'tcp', Reuse => 1, Reuse section () (Reuse Section () () Reuse Section () () Reuse Section () Reuse Reuse Section () Reuse</pre>	<pre>Internet = > 1200, Intern => 5, Proto => 'tcp', Reuse => 1, die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) (\$etaid = fork(); if (\$pid = 0) (for the \$fined (\$pid); for the \$fined (\$pid); for the \$fined (\$pid); for the \$fined (\$pid * \$mew_sock)) for the \$fined (\$main_sock); for the \$fined (\$main_sock); for the \$fid \$pid * \$mew_sock) \$mich goes back to accept (\$pid of the \$mid process \$he parent and \$hild-starting from the statement following the \$fork. The parent gets a positive teum value, the process ID (\$pid) of the \$fid process \$he parent gets a positive teum value and \$cocket and cohoss it back to accept, and the child process reads a line from the socket and cohoss it back to accept, and the child process reads a line from the socket and cohoss it back to accept, and the child process reads a line from the socket and process exits (or terminates a shormally), the \$ystem gets rid of the memory, files, and other resources associated with it. But it retains a small annourt of information (the exit status if the process associated with it. But it retains a small annourt of finformation (the exit status if the process associated with it. But it retains a small annourt of finformation (the exit status if the process associated with it. But it retains a small annourt of finformation (the exit status if the process associated with it is but as the process asociated with it is but as the process as the proces aso</pre>	<pre>Internet = > 1200, Intern => 5, Proto => 'tcp', Reuse => 1, Reuse sock Mhile (\$number 5 main_sock->accept()) { If (\$pid = 0) { Reined (\$pid \$bif, Print \$new_sock) Print \$new_sock) Print \$new_sock "You said, \$bif, Print \$new_sock to accept, and the child process reads a line from the socket and choes the sock to accept, and the child process reads a line from the socket and choes the sock to accept, and the child process reads a line from the socket and process exits (or terminates a shormally), the system gets rdo of the memory, files, and other process exits (or terminates a shormally), the system gets rdo of the memory, files, and other process exits (or terminates a shormally), the system gets rdo of the memory find and other process exits (or terminates a shormally), the system gets rdo of the memory find and other process exits (or terminates a shormand other a</pre>
<pre>LocalPott = 1200, Listen = 5, Proto = > 'tcp', Proto = Proto = Proto</pre>	<pre>LocalPott = 1200, Listen = 5, Proto = > 'tcp', Proto = Proto = Proto</pre>	<pre>location: = > 1, listen => 5, proto => 'tep', proto => 'tep', proto => 'tep', proto => 'tep', proto => 'tep', proto == 'tepk(); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$rea_oock = #main_sock->accept()) { f(\$pid == 0); f(\$pid == 0; f(\$pid = 0; f(\$pid == 0; f(\$pid = 0; f(\$pid =</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$m_acock = \$main_sock-accept()) {</pre>	<pre>LocalPort = > 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$ma_ock->accept()) { %pid = fork(); dia "cannot forx: \$!" unless defined (\$pid) ; if (\$pid == 0) (%pid = process while (defined (\$pid form, seck); %pid == 0) (%pid = process) while (defined (\$pid form, seck); %pid == 0) (* form \$ form \$ for \$ for</pre>	<pre>localPott = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$ms_aock = \$main_sock-accept()) {</pre>
<pre>localPott => 1200, Listen => 5, Froto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (%new_sock = %min_sock-raccept()) {</pre>	<pre>localPott = > 1200, Listen => 5, Froto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$ms_sock=smain_sock=sacept()) {</pre>	<pre>locality of the control of the created is the control of the</pre>
<pre>localPort = > 1200,</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', 'tcp', Proto == 'tcp', 'tcp', Proto == 'tcp', 'tcp', 'tcp', Proto == 'tcp', 'tcp', 'tcp', Proto == 'tcp', 'tcp', 'tcp', Proto == 'tcp', 'tcp', 'tcp', 'tcp', 'tcp', Proto == 'tcp', 'tcp</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', 'tch', Proto == 'tch', 'tcp', 'tch', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tc</pre>
<pre>Indextrant = 1200, Intern = 5, Forture = 1200, Intern = 5, Forture = 5, Fortur</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Proto => Proto => 'tep', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto == 'ti', Proto Proto == 'ti', Proto ==</pre>	<pre>Listen => 5, troote => 'tcp',</pre>
<pre>Instance = 1200, Instance = 5, Proto = * 'tep', Proto = * 'tep', Proto = * 'tep', Proto = * 'tep', Proto = 'tep', Proto = 'tep', Proto = 'tep', Proto = 'teok'); Proto = 'teok'); Proto = Proto = Proto = 'teok', Proto = Proto =</pre>	<pre>Instance = 1200, Instance => 1, Proto => 'tep', Proto == 'tep', Proto ==</pre>	<pre>IncalPort => 1200, Instant => 5, Proto => 'tcp', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Prot</pre>
<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto Proto = 'tcp', Proto == 'tcp',</pre>	<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto = 't', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto = 't', Proto == 't', Proto == 't', Proto == 't', Proto == 't', Proto Proto = 't', Proto == '</pre>	<pre>IncalPort => 1200, Instant => 5, Proto => 'tep', Proto == 't, "unless defined (9pid) ; Proto == 't, "unless defined (9pid) ; Pr</pre>
<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto == 'tch', Proto == 'tch', 'tch', Proto == 'tch', 'tch', Proto == 'tch', 'tch', Proto == 'tch', 'tch', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tch</pre>	<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Proto => 'tcp', Nhile (\$new sock = \$main_sock->accept()) { \$pida = fork(), \$pida = fork(), \$pida = fork(), \$pida = fork(), \$pida = fork(), \$pida = fork(), \$pida = 0) { \$pida = 0, {</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Process Proto == 'tcp', Proto == 'tcp', Process Proto == 'tcp', Proto == 'tcp', Process Proto == 'tcp', P</pre>
<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', 'tch', Proto == 'tch', 'tcp', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tch', 'tch', 'tch', 'tch', 'tch', Proto == 'tch', 'tc</pre>	<pre>LocalPort => 1200, Lister => 5, Proto => 'tcp', Proto == 'tc', 'unless defined ('pld) ; Proto == 'tc', 'unless defined 'pld', 'u</pre>	<pre></pre>
<pre>localPott = > 1200, listen => 5, proto => 'tep', proto == 0, { proto => 'tep', proto == 'tep',</pre>	<pre>localPott = > 1200,</pre>	<pre>Index = 1200, Total port => 1200, Total port => 1200, Total port => 12, Proto => 'tcp', Proto == 'tcp', P</pre>
<pre>localPort => 1200, listen => 5, proto => 'tep', Reuse => 1, proto fork: S!' unless defined (Spid) ; f (Spid = 0) { f (Spid = 0) { f (Spid == 0) { f (Spid) {</pre>	<pre>localPort => 1200,</pre>	<pre>Index Prote => 1200, Index "socket could not be created. Reason: \$!\n" unless (\$sock); Prote => 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); spid = fork(); fie (\$pid = fork(); fie (\$pid = 0) {</pre>
<pre>incalPott => 1200, intern => 5, proto => 'tcp', proto == 'tcp', process proto == 'tcp', proto == 'tcp', process = 'tcp', 'tcp', proto == 'tcp', process = 'tcp', process = 'tcp', 'tcp', proto == 'tcp', 'tcp',</pre>	<pre>IncalPot = 1200, Insten = 5, Proto => 'tcp', Proto => 'tcp', Reuse => 1, Proto => 'tcp', Proto == 20th, Proto => 'tcp', Proto == 20th, Proto => 'tcp', Proto == 20th, Proto => 'tcp', Proto == 20th, Proto == 'tcp', Proto == 20th, Proto == 'tcp', Proto == 20th, Proto == 20th,</pre>	<pre>IncalPot = 1200, Insten = 5, Proto => 'tcp', Proto == 'tcp', Proto Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto</pre>
<pre>IncalPott => 1200, Title "Socket could not be created. Fasson: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) (\$pid = fork(); fif (\$pid = 0) (</pre>	<pre>IncalPott => 1200, Title "Socket could not be created. Fasson: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) (\$pid = fork(); fif (\$pid = 0) (</pre>	<pre>LocalPort => 1200, Lister => 5, Froto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { \$pid = fork(); \$pid = fork(); \$pid = fork(); \$child process while (defined (\$pid); \$child process while (defined (\$pid) print \$new_sock "You said: \$puf(n";) f (\$pid == 0) { * child process which goes back to accept() * child process which goes back to accept()) close (\$main_sock); # else 'tis the parent process which goes back to accept()) close (\$main_sock); The fork call results in two identical process exits when it is done. # else 'tis the parent process which goes back to accept()) close (\$main_sock); f do something with Fuch the process ID (\$pid) detechind process cock this return value, the process ID (\$pid) of the child process cock this return value, the process ID (\$pid) of the child process cock this return value and excente their own logic, the main process goes back to accept, and the child process reads a line from the socket and cochoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per se. On Unix, when a child process easis(or terminals a shormally), the system gest rid of the memory.files, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if</pre>
<pre>IncalPott => 1200, Titesten => 5, Froto => 'tcp', Reuse => 1, Spid = fork(); Reuse => 1, Spid = fork(); Reuse => 1, Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = 0) { for (Spid == 0) { for (Spid =</pre>	<pre>IncalPott => 1200, Tisten => 5, Froto => 'tep', Froto => 'tep', Reuse => 1, Spid = fork(); Reuse => 1, Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = fork(); Reuse sock = Smain_sock->accept()) { Spid = fork(); Reuse sock = Smain_sock); Reuse sock = Smain_sock = Smain_sock); Reuse sock = Smain_sock = Smain_sock); Reuse sock = Smain_sock = Smain_sock); Reuse sock = Smain_sock); Reuse sock = Smain_sock); The fork = The Parent Process = which goes back to accept() } Close (Smain_sock); The fork = Parent process = which goes back to accept() for the child process check this return value, the process ID (Spid) of the child process check this return value, the process ID (Spid) of the child process check this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes subs to the clent. Incidentally, the critD signal manout of information (the exit status if process exists of the monory. The system gets rid of the monory. Thes, and other resources associated with it. But it retains a small amount of information (the exit status if</pre>	<pre></pre>
<pre>IncalPot = 1200, Interen => 5, Proto => 'tcp', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new.sock = \$main_sock->accept()) { \$pid = fork(); \$pid = fork(); if (\$pid == 0) { fin (\$pid = 0) { fin fin (\$pid = 0) { fin fin (\$pid = 0) { fin fin (\$pid = 0) { fin (\$pi</pre>	<pre>IncalPot = 1200, Interen => 5, Proto => 'tep', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); %pile (\$new_sock = \$main_sock->accept()) { \$pid = fork(); \$pid = fork(); if (\$pid == 0) { fin (\$pid = 0) { fin for \$pid = 0, \$pi</pre>	<pre>IncalPot = 1200, Interen => 5, Proto => 'tep', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) { Spid = tork(); Spid = tork(); Spid = tork(); if (\$pid == 0) {</pre>
<pre>IncalPot = > 1200, Instance => 1, Proto => 'tep', Reuse => 1, Reuse => 1,</pre>	<pre>IncalPot = > 1200, Instance => 1, Proto => 'tep', Reuse => 1, Reuse => 1,</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto ==</pre>
<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto ==</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto ==</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto ==</pre>
<pre>LocalPort => 1200, Listen => 5, Proto => 'terp', Reuse => 1, , Reuse => 1, , Spid = fork(); , defined (Suff <amew sock)<br="">, if (Spid == 0) (# Child process which goes back to accept()</amew></pre>	<pre>localPott => 1200, listen => 5; Proto => 'tcp', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', Proto == 'tch', 'tcp', 'tch', Proto == 'tch', 'tcp', 'tch', 'tcp', 'tcp'</pre>	<pre></pre>
<pre>localPot = >10, listen = >5, proto => 'tcp', proto => 'tcp', proto => 'tcp', proto => 'tcp', proto => 'tcp', proto => 'tcp', proto => 'tcp', proto == 'tcp', proto == 'tcp',' proto == 'tcp', proto == 'tcp', proto == 'tcp', proto == 'tcp',' proto == 'tcp', proto == 'tcp', proto == 'tcp', proto == 'tcp',' proto == 'tcp', proto == 'tcp', proto == 'tcp', proto == 'tcp',' proto == 'tcp', proto == 'tcp', proto == 'tcp', proto == 'tcp',' proto == 'tcp', proto == 'tcp',</pre>	<pre>localPot = > 1200, listen => 5, proto => 'tcp', Reuse => 1, proto => 'tcp', Reuse => 1, proto => 'tcp', Reuse => 1, proto => 'tcp', Reuse => 1, proto => 'tcp', relation form: 9! unless defined (\$pid) ; fif (\$pid == 0) (fif (\$pid == 0) (fif</pre>	<pre>LocalPort = >1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new sock = \$main_sock->accept()) { Spid = fork(); f(\$pid = 0) { f(\$pid = 0) {</pre>
<pre>Incertibute = > 1200, Tisten => 5, Proto => 'tep', Reuse => 1,); Reuse sock = Smain_sock->accept()) { spid = fork(); # unless defined (Spid) ; if (Spid == 0) { ti (Spid == 0) {</pre>	<pre>location: = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); Reuse => 1,); Reuse => 1,); Rease == 0;); Rease == 0;</pre>	<pre>localPort => 1200, listen => 5, Proto => 'tep', Reuse => 1,); die "socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock-accept()) { %tad = Tock = \$main_sock.accept()) { %tad = Tock = \$main_sock.accept()) { %tad = Tock = \$main_sock.accept() { %tad = Tock = \$main_sock.accept()) { %tad = Tock = \$main_sock.accept() { %tad = Tock = \$main_sock.accept = Tock = \$main_sock.accept() { %tad = \$main_sock.accept.and \$main_sock.accept.and \$main_sock.accept.and \$main_sock.accept</pre>
<pre>localPort => 1200, listen => 5, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', read = fork(); die "Socket could not be created. Reason: \$:\n" unless (\$sock); %pid = fork(); die "Cannot fork: \$!" unless defined (\$pid); ff (\$pid == 0) (</pre>	<pre>localPort => 1200, listen => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); %pid = fork(); if (\$pid = 00, tesk (); if (\$pid = 0, tesk (); for a = 0, the client. for a social or (); for a soc</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock: \$!" unless defined (\$pid); if (\$pid == 0) (*) * fold process while (defined (\$pid * \$new_sock)) # fold process socks) print \$new_sock * You said; \$buf\n"; print \$new_sock * You said; \$new_sock * You \$new_sock * You \$new_sock * You process satis or the client. Process resock * Not \$new_sock * You \$new_sock</pre>
<pre>localPort => 1200, listen => 5, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', Reuse => 1, proto => 'tep', read = fork(); die "Socket could not be created. Reason: \$:\n" unless (\$sock); %pid = fork(); die "Cannot fork: \$!" unless defined (\$pid); ff (\$pid == 0) (</pre>	<pre>localPort => 1200, listen => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); %pid = fork(); if (\$pid = 00, tesk (); if (\$pid = 0, tesk (); for a = 0, the client. for a social or (); for a soc</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock: \$!" unless defined (\$pid); if (\$pid == 0) (*) * fold process while (defined (\$pid * \$new_sock)) # fold process socks) print \$new_sock * You said; \$buf\n"; print \$new_sock * You said; \$new_sock * You \$new_sock * You \$new_sock * You process satis or the client. Process resock * Not \$new_sock * You \$new_sock</pre>
<pre>LocalFort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reid = fork(); die "Cennfort fork: \$!" unless defined (\$pid); if (\$pid == 0) { # Child process # Child process # Child process # Child process exits when it is done.) # else 'tis the parent process, which goes back to accept() * else 'tis the parent process, which goes back to accept() close (%main_sock); The fork call results in two identical processes exits when it is done. # else 'tis the parent process which goes back to accept() close (%main_sock); The fork call results in two identical process exits when it is form the statement following the fork. The parent gots a positive teturn value, the process ID (\$pid) of the child process Both process echeck this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to accept, and the child process reads a line from the socket and colose it back to the client. Incidentally, the CHID signal has nothing to do with IPC per se. On Unix, when a child process exists of the trains a small amount of information (the exit status if</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tcp', Reuse => 1, Reuse == 0, Reuse == 1, Reuse == 1, R</pre>	<pre>LocalFort => 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); while (\$new_sock = \$min_sock->accept()) { stated = 0 of \$\$ in unless defined (\$pid) ; if (\$pid == 0) { f (\$nid process while (defined (\$pid \$\$ in \$\$ bif `) # f (\$nid process while (defined (\$pid \$\$ in \$\$ done.) if (\$pid == 0) { f (\$nid process while (defined (\$pif \$\$ bif `) # f (\$nid process which goes back to accept()) close (\$main_sock); The fork call realls in two identical processes exits when it is done. # else 'tis the parent process, which goes back to accept()) close (\$main_sock); The fork call realls in two identical processes the parent and child-starting from the statement following the fork. The parent process which goes back to accept() of the child process bolk process set and excute their own big; the main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists or terminates abnormally), the system gest rid of the memory. files, and other resources associated with it. But it retains a small amount of information (the exit status if procuces associated with it. But it retains a small amount of information (the exit status if procuces associated with it. But it retains a small amount of information (the exit status if procuces associated with it. But it retains a small amount of information (the exit status if procuces associated with it. But it retains a small amount of information (the exit status if procuces associated with it. But it retains a small amount of information (the exit status if protects associated with it. But it retains a small amount of the exit status if protects associated with it. But it retains a small amount of the exit status if process associated with it. But it retains a process as a status process associated with process process associated with process proces proces process</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse sock => 1,); Reuse sock => 1,); Reuse sock == 0) (); Reuse sock => 1,); Reuse sock => 1,]; Reuse => 1</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse scattering (Spid => 0) (); Reuse scattering (Spid (Spid Spid));); Reuse scatter which goes back to accept ()); Reuse scatter process which goes back to accept ()); Close (Smain_sock);); Close (Smain_sock);) The fork call results in two identical processes which goes back to accept ()); Close (Smain_sock);] The fork call results in two identical processes exists when it is done.); Close (Smain_sock); The fork call results in two identical process exists when it is done.); Close (Smain_sock); The fork call results in two identical process exists when it is done.); Close (Smain_sock); The fork call results in two identical process exists when it is done.); Close (Smain_sock); The fork call results in two identical process reads a line from the socket and echoes it back to accept, and the child process reads a line from the socket and echoes it back to accept, the child process reads a line from the socket and echoes it back to accept, the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists or tranmally), the system gets rid of the memory. Fles, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of finformation (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if resources associated with it But it retains a small amount of information (the exit status if resources associated with resources associated with resources associated with resources associat</pre>	<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse sock = \$main_sock->accept()) { if (\$pid == 0) { if (\$pid == 0) { f (\$pid == 0)</pre>
<pre>LocalPort => 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse == 0) (); Reuse == 0) (); Reuse == 0) (); Reuse => 1,); Reuse =</pre>	<pre>JocalPort = >1200, Jisten => 5, Proto => 'tep', Reuse => 1,); Reuse score = % main_gock->accept()) {</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse score = \$main_sock->accept()) { dia "Cannot fork: \$!" unless defined (\$pid) ; dia "Cannot fork: \$!" unless defined (\$pid) ; for (atone second) if (\$pid == 0) { tif (\$pid == 0) {</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse sock = % main_sock->accept()) { if (\$pid == 0) { if (\$pid == 0] { if (\$p</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(); % pid = fork(); % child process = % main_sock->accept()) { % pid = fork(); % child process for a complexing (% pid (% pid)); % for (a for (% pid (% pid % pid)); % for (a for (% pid (% pid % pid)); % for (a for (% pid (% pid % pid)); % for (a for (% pid % pid)); % for (a for (% pid for (% pid % pid)); % for (a for (% pid for (% pid % pid)); % for (a for (% pid for (% pid)); % for (a for (% pid)); % for (% pid); % fo</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); die "Socket could not be created. Reason: \$!/n" unless (\$sock); while (fnew sock = \$main_sock->accept()) {</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(1; %pid = fork(1; %pid = fork(1; %pid = fork(1; %pid = fork(1; %pid = = 0) { %formule starts show sock "You said: Spuf,n";) %form sock = Sman_sock "You said: Spuf,n";) %form the sock something with Spuf) %form the process which goes back to accept()) %form the process which goes back to accept()) %form sock is a positive term value, the process ID (\$pid) of the child process check this return value, the process ID (\$pid) of the child process check this return value and excute their own logi; the main process goes back to accept, and the child process reads a line from the sock and choos it back to accept, and the child process reads a line from the sock and choos it back to the client. Incidentally, the cHID signal has nothing to do with IPC per se. On Unix, when a child process exists of traning a bornmally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if the protes associated with it. But it retains a small amount of information (the exit starus if the protes associated with it. But it retains a small amount of information (the exit starus if the protes associated with its base and the protes associated with its base and the protes associated with its base and the protes associated with and the protes associated with a protes associate</pre>	<pre>location: = > 1200, listen => 5, proto => 'tep', neuse => 1,); die "Socket could not be created. Reason: \$!/n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); die "Socket could not be created. Reason: \$!/n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(1; %pid = fork(1;); %pid = = 0) { %formula (former set in the defined (form (former set)); %pid = fork(1;); %former sock->accept(1) { %former sock->accept(1); %former sock, "You said: form(n; % ender former sock); %form(n; former sock); %form the sock "You said: form the sock to main process which goes back to accept, and the child process reads a line from the sock and colors it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates a shormally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if protes associated with it. But it retains a small amount of information (the exit status if protes areas to be accept and the client and be accept and accept ac</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(); % pind = fork(); % pind = fork(); % pind = fork(); % pind == 0) { % folid process while (defined (\$buf <\$new_sock)) % folid process which goes back or accept () % folid process which goes back to accept () % print \$new_sock "You said: \$buf\n";) exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gots a positive return value, the process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per sc. On Unix, when a child process easis dormanally), the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of the process of</pre>	<pre>localPort = > 1200, listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(); % piot = 0) { % for the sock = % main_sock->accept()) { % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = sock = sock = sock = sock) % piot = 0) { % for the sock = sock =</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(1; %pid = fork(1;); %pid = = 0) { %formula (former set in the defined (form (former set)); %pid = fork(1;); %former sock->accept(1) { %former sock->accept(1); %former sock, "You said: form(n; % ender former sock); %form(n; former sock); %form the sock "You said: form the sock to main process which goes back to accept, and the child process reads a line from the sock and colors it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates a shormally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if protes associated with it. But it retains a small amount of information (the exit status if protes areas to be accept and the client and be accept and accept ac</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(); % pind = fork(); % pind = fork(); % pind = fork(); % pind == 0) { % folid process while (defined (\$buf <\$new_sock)) % folid process which goes back or accept () % folid process which goes back to accept () % print \$new_sock "You said: \$buf\n";) exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gots a positive return value, the process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per sc. On Unix, when a child process easis dormanally), the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of the process of</pre>	<pre>localPort = > 1200, listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(); % piot = 0) { % for the sock = % main_sock->accept()) { % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = sock = sock = sock = sock) % piot = 0) { % for the sock = sock =</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(1; %pid = fork(1;); %pid = = 0) { %formula (former set in the defined (form (former set)); %pid = fork(1;); %former sock->accept(1) { %former sock->accept(1); %former sock, "You said: form(n; % ender former sock); %form(n; former sock); %form the sock "You said: form the sock to main process which goes back to accept, and the child process reads a line from the sock and colors it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates a shormally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if protes associated with it. But it retains a small amount of information (the exit status if protes areas to be accept and the client and be accept and accept ac</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(); % pind = fork(); % pind = fork(); % pind = fork(); % pind == 0) { % folid process while (defined (\$buf <\$new_sock)) % folid process which goes back or accept () % folid process which goes back to accept () % print \$new_sock "You said: \$buf\n";) exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gots a positive return value, the process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per sc. On Unix, when a child process easis dormanally), the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of the process of</pre>	<pre>localPort = > 1200, listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(); % piot = 0) { % for the sock = % main_sock->accept()) { % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = sock = sock = sock = sock) % piot = 0) { % for the sock = sock =</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(1; %pid = fork(1;); %pid = = 0) { %formula (former set in the defined (form (former set)); %pid = fork(1;); %former sock->accept(1) { %former sock->accept(1); %former sock, "You said: form(n; % ender former sock); %form(n; former sock); %form the sock "You said: form the sock to main process which goes back to accept, and the child process reads a line from the sock and colors it back to the client. Incidentally, the cHD signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates a shormally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if protes associated with it. But it retains a small amount of information (the exit status if protes areas to be accept and the client and be accept and accept ac</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); adja = fork(); % pind = fork(); % pind = fork(); % pind = fork(); % pind == 0) { % folid process while (defined (\$buf <\$new_sock)) % folid process which goes back or accept () % folid process which goes back to accept () % print \$new_sock "You said: \$buf\n";) exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); # else 'tis the parent process, which goes back to accept ()) close (\$main_sock); } The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gots a positive return value, the process ID (\$pid) of the child process back to accept, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per sc. On Unix, when a child process easis dormanally), the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of the process of</pre>	<pre>localPort = > 1200, listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(); % piot = 0) { % for the sock = % main_sock->accept()) { % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = % main_sock = sock) % piot = 0) { % for the sock = sock = sock = sock = sock) % piot = 0) { % for the sock = sock =</pre>
<pre>LocalPort = > 1200, Listen => 5, Proto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); and = fork(1; %pid = fork(1; %pid = fork(1; %pid = fork(1; %pid = fork(1; %pid = = 0) { %formule starts show sock "You said: Spuf,n";) %form sock = Sman_sock "You said: Spuf,n";) %form the sock something with Spuf) %form the process which goes back to accept()) %form the process which goes back to accept()) %form sock is a positive term value, the process ID (\$pid) of the child process check this return value, the process ID (\$pid) of the child process check this return value and excute their own logi; the main process goes back to accept, and the child process reads a line from the sock and choos it back to accept, and the child process reads a line from the sock and choos it back to the client. Incidentally, the cHID signal has nothing to do with IPC per se. On Unix, when a child process exists of traning a bornmally, the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if resources associated with it. But it retains a small amount of information (the exit starus if the protes associated with it. But it retains a small amount of information (the exit starus if the protes associated with it. But it retains a small amount of information (the exit starus if the protes associated with its base and the protes associated with its base and the protes associated with its base and the protes associated with and the protes associated with a protes associate</pre>	<pre>location: = > 1200, listen => 5, proto => 'tep', neuse => 1,); die "Socket could not be created. Reason: \$!/n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>	<pre>LocalFort => 1200, Listen => 5, Froto => 'tep', Reuse => 1,); Reuse => 1,); Reuse => 1,); die "Socket could not be created. Reason: \$!/n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>
<pre>Idea = > 1; 1200, Iisten => 5, Proto => 'tcp', Reuse => 1,); Reuse == 0, { # Child process which Getined (\$but <\$new_sock) # Child process which goes back to accept()) # else 'tis the parent process, which goes back to accept())) Close (\$main_sock);) The fork call results in two identical processes which goes back to accept()) close (\$main_sock);) The fork call results in two identical processes which goes back to accept() of the child process both processes check this return value, the process ID (\$pid) of the child process both process check this return value and excute their own logic; the main process goes back to accept, and the child process reads a line from the sock tand echoes it back to the client. Incidentally, the cHID signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates a shormmally), the system gets rid of the memory. fles, and other resources associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if process associated with it. But it retains a small amount of information (the exit status if</pre>	<pre>LocalPort = > 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$'\n" unless (\$sock); while (\$new sock = \$main_sock->accept()) { %pid = fork(); %pid = fork(); die "cannof fork: \$!" unless defined (\$pid) ; %pid = fork(); if (\$pid == 0) {</pre>	<pre>LocalPort = > 1200, Listen => 5, Proto => 'tcp', Reuse => 1,); die "Socket could not be created. Reason: \$'\n" unless (\$sock); while (\$new sock = \$main_sock->accept()) { %pid = fork(); %pid = fork(); %pid = fork(); %pid = fork(); %pid = fork(); %pid = fork(); %child process while (defined (\$but <\$new sock») %hut solves %nich goes back to accept() % child process which goes back to accept() % child process exits when it is done. * else 'tis the parent process, which goes back to accept() close (\$main_sock); for the parent process, which goes back to accept() % close (\$main_sock); for the parent gots a positive return value, the process ID (\$pid) of the child process check this return value and execute their own logic, the main process goes back to accept, and the child process reads a line from the sock and choos it back to the client. Incidentally, the cHID signal has nothing to do with IPC per se. On Unix, when a child process exists of treminates abnormally, the system gots rid of the memory. files, and other resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with it. But it retains a small amount of information (the exist status if resources associated with its as a status if a status if retains a status if a status if resources associated</pre>
<pre>listen => 5,</pre>	<pre>Jocathort => 1200, Jisten => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Reason: \$!\n" unless (\$sock); %pid = fock(); die "Cann(); die "Cann(); fock = 3main_sock->accept()) { %pid = fock(); die "Cann() forcess while (defined (\$buf <\$new_sock)) if (\$pid == 0) { %fock == 0} { %fock == 0 { %fock == 0 { %fock == 0} { %fock == 0 { %fock == 0 { %fock == 0} { %fock == 0 {</pre>	<pre>JocatPort => 1200, Jisten => 5, Proto => 'tep', Reuse => 1,); die "Socket could not be created. Neason: \$!\n" unless (\$sock); while (\$new_sock = \$main_sock->accept()) {</pre>
<pre></pre>	<pre>dis "Socket could not be created. Fister => 5,</pre>	<pre>dis = 's' = 'te',</pre>
<pre>dise "socket could not be created. Farston => 1, '</pre>	<pre>dis = 2. tcut,</pre>	<pre>die "Socket could not be created. Reason: \$!\n" unless (\$sock);</pre>
<pre></pre>	<pre></pre>	<pre></pre>
<pre></pre>	<pre>distant => 5,</pre>	<pre>dis = "socket could not be created. Reason: \$1, " unless (\$sock);</pre>
<pre>dis "Socket could not be created. Reason: \$!\n" unless (\$sock);</pre>	<pre>die "Socket could not be created: Reason: \$1, " unless (\$sock);</pre>	<pre>die "Socket could not be created. Reason: \$1, " unless (\$sock);</pre>
<pre>dis = 's'' ' 's''''''''''''''''''''''''''</pre>	<pre>dis = 's'' = '12'',</pre>	<pre>dis = 's' = 'to',</pre>

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	parent uses wait or waitpid to enquire about this status. The terminated child process is also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait; otherwise, the process tables keep filling up with junk. In the preceding code, wait doesn't block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.
	"The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_cthc, pp_sys.c, and pp_hot.c. The last one contains the opcode functions that are "hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come
	anywhere close in performance. (I'll reevaluate this claim once Sun's Java processors are freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." <i>Srinivasan</i> at 324. "This returns a list of object references that match the query criteria. Now if you reissue this
	query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." <i>Srinivasan</i> at 178.
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	language (or at least a standard query language). When you have a million objects in the
	database, it would be a terrible thing to load every single object in memory to see whether it
	matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates
	queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded
	scheme for file based objects: it converts the query expression to an evalable Perl expression
	and cycles through all objects, matching them against the query specification.
	Schema Evolution
	Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are
	added to the object implementation. The schema is said to have evolved. The framework has
	to be able to reconcile old data with newer object implementations.
	Implementation
	This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
	only the key procedures that perform query processing and file or database I/O. Pay as much
	attention to the design gotchas and unimplemented features as you do to the code.
	Adaptor::File
	An Adaptor:: File instance represents all objects stored in one file. When this adaptor is
	created (using new), it reads the entire file and translates the data to in-memory objects.
	Slurping the entire file into memory avoids the problem of having to implement fancy
	on-disk schemes for random access to variable-length data; after all, that is the job of DBM
	and database implementations. For this reason, this approach is not recommended for large
	numbers of objects (over 1,000, to pick a number).
	The file adaptor has an attribute called all instances, a hash table of all objects given to its store
	method (and indexed by their _id), as shown in Figure 11-2.
	Fike Adjaption
	100 E source dat Datients
	t.
	unununununununununununaanaaaaaaaaaaaaa
	e ligare a a accessere o ji ter analysis

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	Srinivasan at 179-80.
	"Perl automatically garbage collects a data structure when its reference count drops to zero. If a data structure has been blessed into a module, Perl allows that module to perform some clean-up before it destroys the object, by calling a special procedure in that module called DESTROY and passing it the reference to the object to be destroyed:
	<pre>package Employee; sub DESTROY { my (\$emp) = @_; print "Alas, ", \$emp->{"name"}, "is now no longer with us \n"; }</pre>
	This is similar to C++'s destructor or the finalize () method in Java in that Perl does the memory management automatically, but you get a chance to do something before the object is reclaimed. (Unlike Java's finalize, Perl's garbage collection is deterministic; DESTROY is called as soon as the object is not being referred to any more.)" <i>Srinivasan</i> at 112.
22. A method according to claim 10, further comprising: setting operating system level resource management parameters on the child runtime system	<i>Srinivasan</i> provided an apparatus with a setting operating system level resource management parameters on the child runtime system process. For example, "[o]n Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it." <i>Srinivasan</i> at 195.
	"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports $fork$, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on accept, and when a connection request comes in, it spawns a child process and goes back to accept. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by accept. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

U.S. Patent No. 7,426,720	
	<pre># Forking server use IO:: Socket; cerr(curr) curb ()</pre>
	sud(сны) sup {wait ()}; \$main_sock = new IO::Socket::INET (LocalHost => 'goldengate', тосарокт => 1000
	Proto => 'tcp',
	keuse => 1,);
	be created. nain_sock->acc
	<pre>\$pid = fork(); die "Cannot fork: \$!" unless defined (\$pid) ;</pre>
	# UNIIα process while (defined (\$buf <\$new sock»)
	# do something with Sbuf
	print \$new_sock "You said: \$buf\n";
	<pre>% exit(0); # Child process exits when it is done. # else 'tis the parent process, which goes back to accept()</pre>
	} close (\$main_sock);
	The $fork$ call results in two identical processes-the parent and child-starting from the statement following the $fork$. The parent gets a positive return value, the process ID (pid)
	of the child process. Both processes check this return value and execute their own logic; the
	main process goes back to accept, and the child process reads a line from the socket and echoes it back to the client.
	Incidentally, the CHLD signal has nothing to do with IPC per se. On Unix, when a child
	process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if
	the child was able to execute exit(), or a termination status otherwise), just in case the
	parent uses wait or waitpid to enquire about this status. The terminated child process is
	also known as a <i>zombie</i> process, and it is always a good thing to remove it using wait;

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	block, because it is called only when we know for sure that a child process has died-the CHLD signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and SIGCHLD in particular.
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." <i>Srinivasan</i> at 194-95.
	"The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_ct, pp_sysc, and pp_hot. The last one contains the opcode functions that are
	"hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come
	freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." <i>Srinivasan</i> at 324.
	"This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the
	"The Adaptor: : File module does not have this problem because it maintains a list of all "The Sector is store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists.
	Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	queries to equivalent SQL queries, while Adaptor:: File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression
	and cycles through all objects, matching them against the query specification. Schema Evolution
	Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has
	to be able to reconcile old data with newer object implementations.
	This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover
	only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code.
	An Adaptor::File instance represents all objects stored in one file. When this adaptor is
	created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy
	on-disk schemes for random access to variable-length data; after all, that is the job of DBM
	and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number).
	The file adaptor has an attribute called all instances, a hash table of all objects given to its store method (and indexed by their id), as shown in Figure 11-2.
	File Auaptor
	ait instances and childents
	laguere 11-2. Stracture of file adaptor
	Srinivasan at 179-80.

U.S. Patent No. 7,426,720	Srinivasan in view of APA-Bach
	"Perl automatically garbage collects a data structure when its reference count drops to zero. If a
	data structure has been blessed into a module, Perl allows that module to perform some clean-up
	before it destroys the object, by calling a special procedure in that module called DESTRUY and
	passing it the reference to the object to be destroyed:
	package Employee;
	sub DESTROY {
	my ($\$emp$) = $@_$;
	print "Alas, ", \$emp->{"name"}, "is now no longer with us \n";
	This is similar to C++'s destructor or the finalize () method in Java in that Perl does the memory
	management automatically, but you get a chance to do something before the object is reclaimed.
	(Unlike Java's finalize, Perl's garbage collection is deterministic; DESTROY is called as soon as
	the object is not being referred to any more.)" Srinivasan at 112.