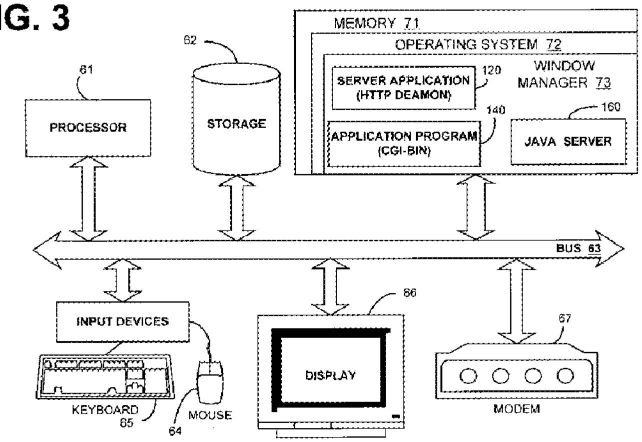
EXHIBIT 19: BRYANT IN VIEW OF APA-BACH

U.S. Patent No. 6,405,367
"Apparatus and Method for Increasing the Performance of Java Programs Running on a Server"
Inventors: *Bryant* et al.
Filing Date: June 5, 1998
Issue Date: June 11, 2002
Assignee: Hewlett-Packard Company
("*Bryant*")

M. J. Bach, The Design of the Unix Operating System, Bell Telephone Labs., Inc. (1986) ("*APA-Bach*")

U.S. Patent No. 7,426,720	*Bryant* in view of *APA-Bach*
1. A system for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Bryant* discloses a system for dynamic preloading of classes through memory space cloning of a master runtime system process. *Bryant* discloses a method to "speed up the actual execution of a particular Java application" by "preload[ing]" the potentially needed object *files* and classes. The goal of the *Bryant* disclosure is the streamlining and acceleration of the performance of a Java application execution. *Bryant* notes from the outset that "it is faster to connect up to the already running Java server and have the already running Java server fork a child server" than it is to execute the desired classes from scratch. *Bryant*, Abstract. Thus one of ordinary skill in the art seeking to improve or accelerate the performance of a Java operation would look to combine the disclosure of *Bryant* with the copy-on-write technology that was very well known in the art, as evidenced by the *APA-Bach* text. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application. The Java server accomplishes the execution of a particular Java application by forking itself and then having the child Java server run the Java class files in the already loaded Java virtual machine for the specific Java CGI-BIN script." *Bryant*, col.2 ll.46-48 (Summary of the Invention).

A processor; A memory	*Bryant* discloses a processor and a memory. FIG. 3  *Bryant*, Fig 3.
a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code;	*Bryant* discloses a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code. The apparatus and method of Bryan works by "preload[ing] all potentially needed object files," as quoted below. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col.2, ll.46–48 (Summary of the Invention) "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant*, col.6, l.66–col.7, l.4.

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;	*Bryant* discloses 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. The master runtime system process is merely the standard creation of a process form a set of instructions. *Bryant* discloses this as the initialization step and then the loading of the standard class files needed for Java application execution. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant*, col.6, l.66–col.7, l.4.
a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process;	*Bryant* discloses a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process. The execution of a child runtime process here is simply the well-known fork system call, which creates a duplication of a master runtime process. This is disclosed in *Bryant* as quoted below. "Java server 160 forks immediately to create a child Java server 180, upon the establishment of the pipe connection, so that the pipe connection from the application program 140 is connected to both the parent Java server 160 and to the child Java server 180. The child Java server 180 receives the program name execution arguments and environmental arguments sent on the pipe connection, sets up the file descriptors, maps to the requested class and method, executes the class and method, and writes the output to a stdout; which is then returned to application program 140." *Bryant*, col.5, ll.9–14. "Immediately upon being called by the application program 140, the Java server 160 forks a process of the child Java server 180 with the pipe connection, thereby establishing communication with the application program 140, with Java server 160 and with the process of the child Java server 180 at step 163. The Java server 160 then waits to receive the exit status from the process of the child Java server 180 that is forked to provide the requested service at step 164." *Bryant*, col.7, ll.7–14.

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.	*Bryant* in combination with *APA-Bach* discloses 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. As discussed in the Request for Reexamination, the copy-on-write technology was widely known in the art at the time of the purported invention of the '720 patent. The *APA-Bach* reference clearly discloses the utility and application of the copy-on-write method of cloning. One of ordinary skill in the art looking for a means to streamline and accelerate a Java machine could have combined the *Bryant* reference with *APA-Bach* to minimize resource usage by copying data only when needed, *i.e.*, employing copy-on-write. Thus, this claim element would have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of *Bryant*, in combination with *APA-Bach*. "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 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 *fork* 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 *exec* soon after the *fork* 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 *fork*. 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.

do this, the kernel turns on the 'copy on write' bit for every page table entry in private 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.

U.S. Patent No. 7,426,720	*Bryant*
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.	*Bryant* discloses a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process. *Bryant* discloses the preloading of all potentially needed object files of the Java virtual machine. The reason for this preloading is to speed up the actual execution of a particular Java application. Thus it is inherent that the *Bryant* disclosure would check whether the class or object file has been preloaded in a local cache, with the goal to speed up the execution of the class or object file. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col. 2, ll.46–49.

U.S. Patent No. 7,426,720	*Bryant*
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.	*Bryant* discloses a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. This limitation is also inherent in *Bryant*, for reasons similar to those described above with respect to claim 2. In order to speed up the actual execution of files it is inherent that *Bryant* would utilize a class locator to identify the files. Where the class locator is unable to locate an instantiated class definition in the local cache, the method of *Bryant* would revert to the standard operating procedure of locating a source definition and executing a class in the traditional manner. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col. 2, ll.46–49.

U.S. Patent No. 7,426,720	***Bryant***
4. A system according to claim 1, further comprising: a class resolver to resolve the class definition.	*Bryant* discloses a class resolver to resolve the class definition. This limitation is also inherent in *Bryant*, for reasons similar to those described above with respect to claim 2. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col. 2, ll.46–49.

U.S. Patent No. 7,426,720	***Bryant***
5. A system according to claim 1, further comprising: at least one of a local and remote file system to maintain the source definition as a class file.	*Bryant* discloses at least one of a local and remote file system to maintain the source definition as a class file. The local file system is preloaded with the potentially needed object files, as disclosed below. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col. 2, ll.46–49.

U.S. Patent No. 7,426,720	*Bryant*
6. A system according to claim 1, further comprising: 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.	*Bryant* discloses 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. The process cloning mechanism of *Bryant* is useful to load the object file only once and then execute the fork system call to create child runtime processes. "The Java server is accessed by an object file (proxy), that is setup to access the correct Java server process. Next, when an application is to be executed, the object file calls the Java server process that forks itself and then has the child server run the already loaded classes and methods. Thus, the Java classes and methods are loaded only once when the Java virtual machine is started. With large classes and methods, it is faster to connect up to the already running Java server and have the already running Java server fork a child server to execute the correct classes and methods than it is to start and load the Java virtual machine, and execute the original classes and methods." *Bryant*, abstract.

U.S. Patent No. 7,426,720	*Bryant*
7. A system according to claim 1, wherein the master runtime system process is caused to sleep relative to receiving the process request.	*Bryant* discloses the master runtime system process is caused to sleep relative to receiving the process request. The listening pipe of *Bryant* is idle until the Java server receives a request for service call, as explained below. "The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution. A listening pipe is setup to receive requests for service calls and the Java server waits to receive a call in step 162." *Bryant,* col.7, ll.1–6.

U.S. Patent No. 7,426,720	*Bryant*
8. A system according to claim 1, wherein the object-oriented program code is written in the Java programming language.	*Bryant* discloses object-oriented program code is written in the Java programming language. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant,* col.6, l.65–col.7, l.4.

U.S. Patent No. 7,426,720	*Bryant*
9. A system according to claim 8, wherein the master runtime system process and the child runtime system process are Java virtual machines.	*Bryant* discloses the master runtime system process and the child runtime system process are Java virtual machines as directly disclosed below. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution. Immediately upon being called by the application program 140, the Java server 160 forks a process of the child Java server 180 with the pipe connection, thereby establishing communication with the application program 140, with Java server 160 and with the process of the child Java server 180 at step 163." *Bryant,* col.6, l.65–col.7, l.12.

U.S. Patent No. 7,426,720	*Bryant*
10. A method for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Bryant* discloses a method for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application. The Java server accomplishes the execution of a particular Java application by forking itself and then having the child Java server run the Java class files in the already loaded Java virtual machine for the specific Java CGI-BIN script." *Bryant* at col.2, ll.46–48 (Summary of the Invention)
executing a master runtime system process;	*Bryant* discloses executing a master runtime system process. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant* col.6 l.66 – col.7 l.4

obtaining a representation of at least one class from a source definition provided as object-oriented program code;	*Bryant* discloses obtaining a representation of at least one class from a source definition provided as object oriented program code. *Bryant,* col.2, ll.46–48 (Summary of the Invention). "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant,* col.6, l.66–col.7, l.4. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution."
interpreting and instantiating the representation as a class definition in a memory space of the master runtime system process;	*Bryant* discloses interpreting and instantiating the representation as a class definition in a memory space of the master runtime system process. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant*, col.6, l.66–col.7, l.4.

and cloning the memory space as a child runtime system process responsive to a process request and executing the child runtime system process;	*Bryant* discloses cloning the memory space as a child runtime system process responsive to a process request and executing the child runtime system process. "Java server 160 forks immediately to create a child Java server 180, upon the establishment of the pipe connection, so that the pipe connection from the application program 140 is connected to both the parent Java server 160 and to the child Java server 180. The child Java server 180 receives the program name execution arguments and environmental arguments sent on the pipe connection, sets up the file descriptors, maps to the requested class and method, executes the class and method, and writes the output to a stdout; which is then returned to application program 140." *Bryant*, col. 5, ll.9–14. "Immediately upon being called by the application program 140, the Java server 160 forks a process of the child Java server 180 with the pipe connection, thereby establishing communication with the application program 140, with Java server 160 and with the process of the child Java server 180 at step 163. The Java server 160 then waits to receive the exit status from the process of the child Java server 180 that is forked to provide the requested service at step 164." *Bryant*, col.7, ll.7-14.

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 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 until the child runtime system process needs to modify the referenced memory space of the master runtime system process.	*Bryant* discloses a method 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 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 until the child runtime system process needs to modify the referenced memory space of the master runtime system process. Claim 10 seeks to claim a longhand version of the same technology relevant to claim 1: copy-on-write cloning. But, as explained above with respect to claim 1 and as detailed with respect to the copy-on-write limitation as outlined in claim 1, *APA-Bach* explicitly discloses the copy-on-write technology. Applicant pointed to the relevance of the *APA-Bach* text by incorporating Chapter 7 of the *APA-Bach* description of the fork system call. One of ordinary skill in the art could have kept reading in that same disclosure and come across the discussion of the copy-on-write technology in Chapter 9 of the *APA-Bach* disclosure. Also, as further discussed in the Request for Reexamination the copy-on-write technology was widely known in the art at the time of the purported invention of the '720 patent. The *APA-Bach* reference clearly discloses the utility and application of the copy-on-write method of cloning. One of ordinary skill in the art looking for a means to streamline and accelerate a Java machine could have combined the *Bryant* reference with the *APA-Bach* reference. Thus, this claim element would have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of *Bryant*, either by itself or in combination with *APA-Bach*. The relevant disclosures of *APA-Bach* are quoted above with reference to claim 1.

U.S. Patent No. 7,426,720	***Bryant***
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.	*Bryant* discloses determining whether the instantiated class definition is available in a local cache associated with the master runtime system process. "The present invention is generally directed to an apparatus and method for increasing the performance of Java application execution for tasks requiring fast execution of Java applications using Java language application software. In accordance with the preferred embodiment of the present invention, the invention is accomplished by moving the code that was in the individual Java CGI-BIN script into one Java server daemon process. The individual Java CGI-BIN scripts are replaced by an object file (the proxy) that calls the daemon process to execute the code that would be in the CGI-BIN script on the proxy's behalf. The proxy object file preferably is in C and executes Java code by invoking the Java virtual machine so very minor changes are needed to turn the Java applications into library routines. The Java server invokes the Java virtual machine . . ." *Bryant*, col. 2, ll.32–46.

U.S. Patent No. 7,426,720	***Bryant***
12. A method according to claim 11, further comprising: locating the source definition if the instantiated class definition is unavailable in the local cache.	*Bryant* discloses locating the source definition if the instantiated class definition is unavailable in the local cache. "The present invention is generally directed to an apparatus and method for increasing the performance of Java application execution for tasks requiring fast execution of Java applications using Java language application software. In accordance with the preferred embodiment of the present invention, the invention is accomplished by moving the code that was in the individual Java CGI-BIN script into one Java server daemon process. The individual Java CGI-BIN scripts are replaced by an object file (the proxy) that calls the daemon process to execute the code that would be in the CGI-BIN script on the proxy's behalf. The proxy object file preferably is in C and executes Java code by invoking the Java virtual machine so very minor changes are needed to turn the Java applications into library routines. The Java server invokes the Java virtual machine . . ." *Bryant*, col. 2, ll.32–46.

U.S. Patent No. 7,426,720	***Bryant***
13. A method according to claim 10, further comprising: resolving the class definition.	*Bryant* discloses resolving the class definition. "The present invention is generally directed to an apparatus and method for increasing the performance of Java application execution for tasks requiring fast execution of Java applications using Java language application software. In accordance with the preferred embodiment of the present invention, the invention is accomplished by moving the code that was in the individual Java CGI-BIN script into one Java server daemon process. The individual Java CGI-BIN scripts are replaced by an object file (the proxy) that calls the daemon process to execute the code that would be in the CGI-BIN script on the proxy's behalf. The proxy object file preferably is in C and executes Java code by invoking the Java virtual machine so very minor changes are needed to turn the Java applications into library routines. The Java server invokes the Java virtual machine . . ." *Bryant*, col. 2, ll.32–46.

U.S. Patent No. 7,426,720	***Bryant***
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.	*Bryant* discloses maintaining the source definition as a class file on at least one of a local and remote file system. "[W]hen the size of the Java class files gets large, the amount of time spent loading the Java code can be a performance limiter, since the Java virtual machine dynamically loads class files only when needed." *Bryant*, col. 2, ll.5–9.

U.S. Patent No. 7,426,720	***Bryant***
15. A method according to claim 10, further comprising: instantiating 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.	*Bryant* discloses instantiating 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. "In accordance with the invention, it has been determined that with large Java scripts, it is faster to connect up to the server and have it fork a child and execute the correct code than it is to start a new Java virtual machine, load the needed class files and execute the correct code." *Bryant*, col. 2, ll. 57–63.

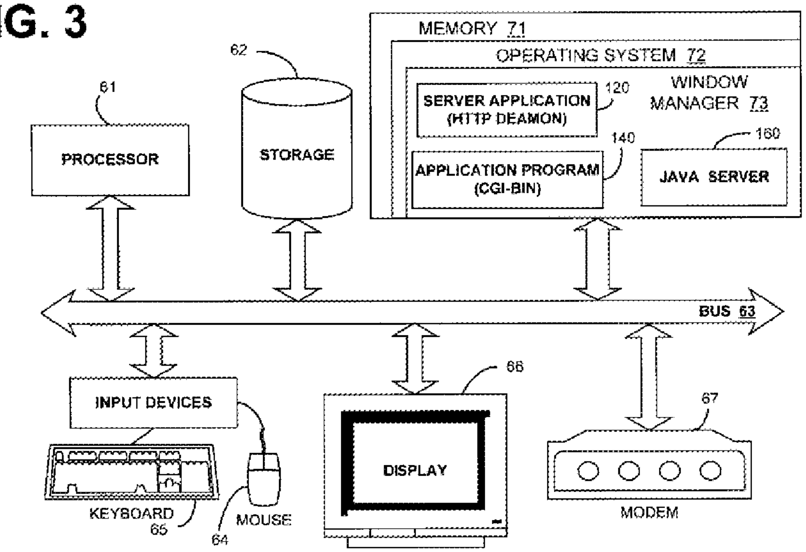
U.S. Patent No. 7,426,720	*Bryant*
16. A method according to claim 10, further comprising: causing the master runtime system process to sleep relative to receiving the process request.	*Bryant* discloses causing the master Java virtual machine to sleep relative to receiving the process request. "The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution. A listening pipe is setup to receive requests for service calls and the Java server waits to receive a call in step 162." *Bryant* col.7 ll.1-6.

U.S. Patent No. 7,426,720	*Bryant*
17. A method according to claim 10, wherein the object-oriented program code is written in the Java programming language.	*Bryant* discloses a method wherein the object-oriented program code is written in the Java programming language. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant* col.6 l.65 – col.7 l.4.

U.S. Patent No. 7,426,720	*Bryant*
18. A method according to claim 17, wherein the master runtime system process and the child runtime system process are Java virtual machines.	*Bryant* discloses a master runtime process and a child runtime process that are Java virtual machines. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution. Immediately upon being called by the application program 140, the Java server 160 forks a process of the child Java server 180 with the pipe connection, thereby establishing communication with the application program 140, with Java server 160 and with the process of the child Java server 180 at step 163." *Bryant* col.6 l.65 – col.7 l.12

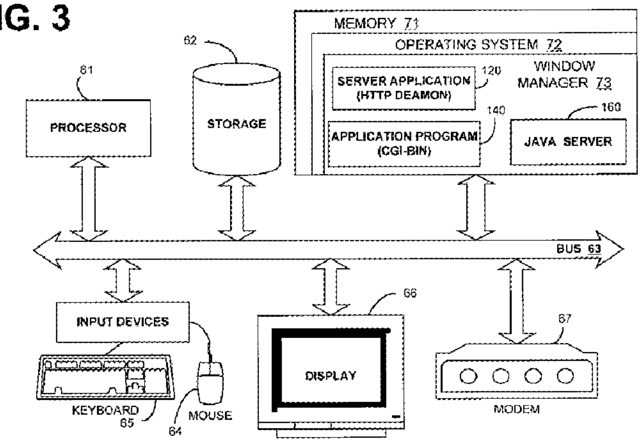
<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Bryant</th></tr>
<tr><td>19. A computer-readable storage medium holding code for performing the method according to claim 10.</td><td>Bryant discloses a computer-readable storage medium holding code for performing the method according to claim 10. See, e.g., the disclosure of "storage" (62) as depicted in Fig. 3.

FIG. 3



Bryant, Fig. 3.</td></tr>
</table>

U.S. Patent No. 7,426,720	*Bryant*
20. An apparatus for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Bryant* discloses a system for dynamic preloading of classes through memory space cloning of a master runtime system process. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application. The Java server accomplishes the execution of a particular Java application by forking itself and then having the child Java server run the Java class files in the already loaded Java virtual machine for the specific Java CGI-BIN script." *Bryant*, col. 2, ll.46–53.

A processor; A memory means for executing a master runtime system process;	The system disclosed in *Bryant* includes a network server or web server that includes a processor and a memory means for executing a master runtime system process. *See, e.g.*, the disclosure of a "processor" (61) and "storage" (62) in Fig. 3 of the *Bryant* reference. FIG. 3  *Bryant*, Fig 3.
means for obtaining a representation of at least one class from a source definition provided as object-oriented program code;	*Bryant* discloses a means for obtaining a representation of at least one class from a source definition provided as object-oriented program code. "The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application." *Bryant*, col.2 ll.46-48 (Summary of the Invention). "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant*, col.6 l.66 – col.7 l.4.

means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process;	*Bryant* discloses a means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process. "The process of the Java server 160 will now be discussed with respect to FIG. 8. First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." *Bryant*, col.6 l.66 – col.7 l.4.
and means for cloning the memory space as a child runtime system process responsive to a process request and means for executing the child runtime system process;	*Bryant* discloses a means for cloning the memory space as a child runtime system process responsive to a process request and means for executing the child runtime system process. "Java server 160 forks immediately to create a child Java server 180, upon the establishment of the pipe connection, so that the pipe connection from the application program 140 is connected to both the parent Java server 160 and to the child Java server 180. The child Java server 180 receives the program name execution arguments and environmental arguments sent on the pipe connection, sets up the file descriptors, maps to the requested class and method, executes the class and method, and writes the output to a stdout; which is then returned to application program 140." *Bryant*, col.5 ll.9-14. "Immediately upon being called by the application program 140, the Java server 160 forks a process of the child Java server 180 with the pipe connection, thereby establishing communication with the application program 140, with Java server 160 and with the process of the child Java server 180 at step 163. The Java server 160 then waits to receive the exit status from the process of the child Java server 180 that is forked to provide the requested service at step 164." *Bryant*, col.7 ll.7-14.

wherein the means for cloning the memory space is configured to clone the memory space of a child runtime system process using a copy-on-write process cloning mechanism that instantiates 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 that defers 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.	*Bryant* discloses an apparatus wherein the means for cloning the memory space is configured to clone the memory space of a child runtime system process using a copy-on-write process cloning mechanism that instantiates 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 that defers 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. The *APA-Bach* reference clearly discloses the utility and application of the copy-on-write method of cloning. One of ordinary skill in the art looking for a means to streamline and accelerate a Java machine could have combined the Bryant reference with *APA-Bach* to minimize resource usage by copying data only when needed, *i.e.*, employing copy-on-write. "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 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 *fork* 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 *exec* soon after the *fork* 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 *fork*. 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.

U.S. Patent No. 7,426,720	***Bryant***
21. A system according to claim 1, further comprising: a resource controller to set operating system level resource management parameters on the child runtime system process.	*Bryant*, either by itself or in combination with *APA-Bach*, discloses a resource controller to set operating system level resource management parameters on the child runtime system process. *Bryant* creates a child server by forking the parent server. This child server operates under resource management controls, executing specific tasks as instructed by the Java server. "The child Java server 180 is initialized at step 181. The child Java server 180 receives the information sent on the pipe connection created by the application program 140 at step 182. The child Java server 180 then maps, at step 183, to the specified application (i.e., class and method) identified in the information that was communicated over the pipe connection and received at step 182. The child Java server then executes the specified application (i.e. class and method) using the specified program name, execution arguments, and environment arguments present in the information received on the pipe connection at step 184." *Bryant*, col. 7, ll. 41-52.

U.S. Patent No. 7,426,720	***Bryant***
22. A method according to claim 10, further comprising: setting operating system level resource management parameters on the child runtime system process.	*Bryant*, either by itself or in combination with *APA-Bach*, discloses setting operating system level resource management parameters on the child runtime system process. *Bryant* creates a child server by forking the parent server. This child server operates under resource management controls, executing specific tasks as instructed by the Java server. "The child Java server 180 is initialized at step 181. The child Java server 180 receives the information sent on the pipe connection created by the application program 140 at step 182. The child Java server 180 then maps, at step 183, to the specified application (i.e., class and method) identified in the information that was communicated over the pipe connection and received at step 182. The child Java server then executes the specified application (i.e. class and method) using the specified program name, execution arguments, and environment arguments present in the information received on the pipe connection at step 184." *Bryant*, col. 7, ll. 41-52.