

Processes & Threads (3 of 3)

Process Creation

There are two main models of process creation - the *fork/exec* and the *spawn* models. On systems that support *fork*, a new process is created as a copy of the original one and then explicitly executes (*exec*) a new program to run. In the *spawn* model the new program and arguments are named in the system call, a new process is created and that program run directly.

Fork is the more flexible model. It allows a program to arbitrarily change the environment of the child process before starting the new program. Typical *fork* pseudo-code looks like:

```
if ( fork() == 0 ) {
    /* Child process */
    change standard input
    block signals for timers
    run the new program
}
else {
    /* Parent process */
    wait for child to complete
}
```

Any parameters of the child process's operating environment that must be changed must be included in the parameters to *spawn*, and *spawn* will have a standard way of handling them. There are various ways to handle the proliferation of parameters that results, for example AmigaDOS® uses tag lists - linked lists of self-describing parameters - to solve the problem.

The steps to process creation are similar for both models. The OS gains control after the *fork* or *spawn* system call, and creates and fills a new PCB. Then a new address space (memory) is allocated for the process. *Fork* creates a copy of the parent address space, and *spawn* creates a new address space derived from the program. Then the PCB is put on the run list and the system call returns.

An important difference between the two systems is that the *fork* call must create a copy of the parent address space. This can be wasteful if that address space will be deleted and rewritten in a few instruction's time. One solution to this problem has been a second system call, *vfork*, that lets the child process use the parent's memory until an *exec* is made. We'll discuss other systems to mitigate the cost of *fork* when we talk about memory management.

Which is "better" is an open issue. The tradeoffs are flexibility vs. overhead, as usual.

Threads

Threads are lightweight processes. They improve performance by weakening the process abstraction. A process is one thread of control executing one program in one address space. A thread may have multiple threads of control running different parts of a program in one address space.

Because threads expose multitasking to the user (cheaply) they are more powerful, but more complicated. Thread programmers have to explicitly address multithreading and synchronization (which is the topic of our next unit).

User Threads

User threads are implemented entirely in user space. The programmer of the thread library writes code to synchronize threads and to context switch them, and they all run in one process. The operating system is unaware that a thread system is even running.

User-level threads replicate some amount of kernel level functionality in user space. Examples of user-level threads systems are Nachos and Java (on OSes that don't support kernel threads).

Because the OS treats the running process like any other there is no additional kernel overhead for user-level threads. However, the user-level threads only run when the OS has scheduled their underlying process (making a blocking system call blocks all the threads.)

Kernel Threads

Some OS kernels support the notion of threads and schedule them directly. There are system calls to create threads and manipulate them in ways similar to processes. Synchronization and scheduling may be provided by the kernel.

Kernel-level threads have more overhead in the kernel (a kernel thread control block) and more overhead in their use (manipulating them requires a system call). However the abstraction is cleaner (threads can make system calls independently).

Examples include Solaris LWPs, and Java on machines that support kernel threads (like solaris).