# Assignment 03

## Computer Systems

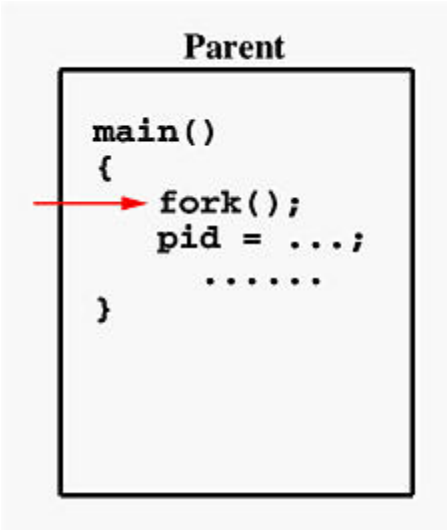
2023PCS0034

MOHAMMAD FAISAL SAYED

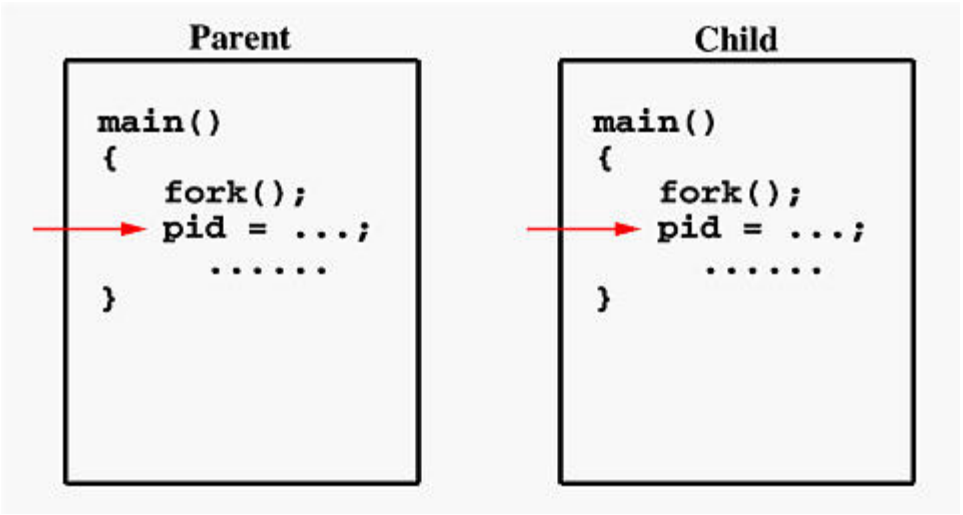
1. FORK()

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

If fork() returns a negative value, the creation of a child process was unsuccessful.
fork() returns a zero to the newly created child process.
fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.
Therefore, after the system call to fork(), a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.
Suppose the above program executes up to the point of the call to **fork()** (marked in red color):




- If the call to **fork()** is executed successfully, Unix will
- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:

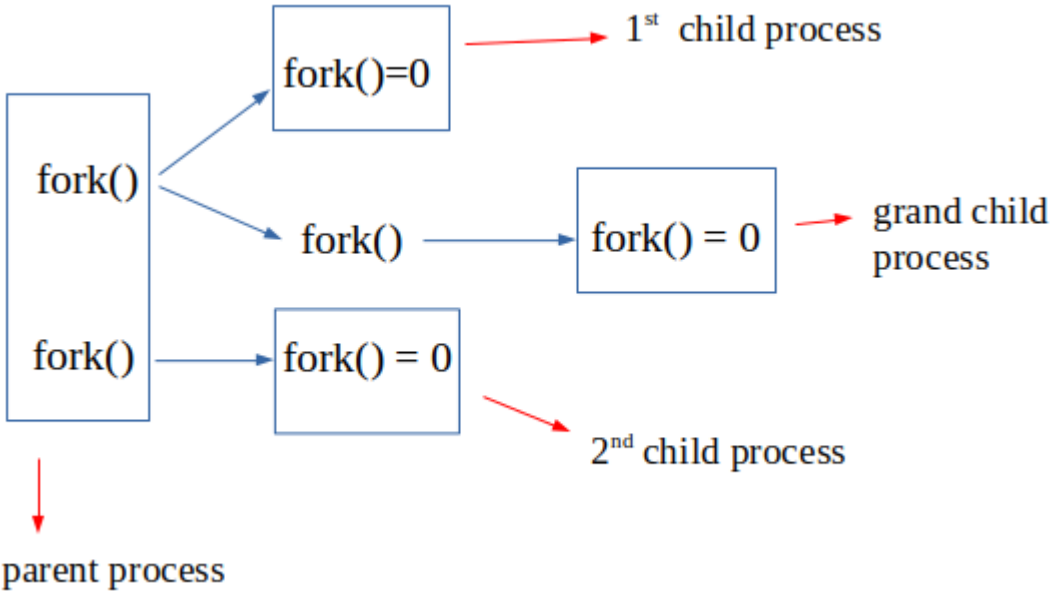



2. WAIT()

The system call wait() is easy. This function blocks the calling process until one of its child processes exits or a signal is received. For our purpose, we shall ignore signals. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.



## 3. THREADS()

Threads are "light weight processes" (LWPs). The idea is a process has five fundamental parts: code ("text"), data (VM), stack, file I/O, and signal tables. "Heavy-weight processes" (HWPs) have a significant amount of overhead when switching: all the tables have to be flushed from the processor for each task switch. Also, the only way to achieve shared information between HWPs is through pipes and "shared memory". If a HWP spawns a child HWP using fork(), the only part that is shared is the text.

Threads reduce overhead by sharing fundamental parts. By sharing these parts, switching happens much more frequently and efficiently. Also, sharing information is not so "difficult" anymore: everything can be shared. There are two types of threads: user-space and kernel-space.

User-Space Threads

User-space avoids the kernel and manages the tables itself. Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher. Also, a timer signal can force switches. User threads typically can switch faster than kernel threads [however, Linux kernel threads' switching is actually pretty close in performance].

Disadvantages. User-space threads have a problem that a single thread can monopolize the timeslice thus starving the other threads within the task. Also, it has no way of taking advantage of SMPs (Symmetric MultiProcessor systems, e.g. dual-/quad-Pentiums). Lastly, when a thread becomes I/O blocked, all other threads within the task lose the timeslice as well.

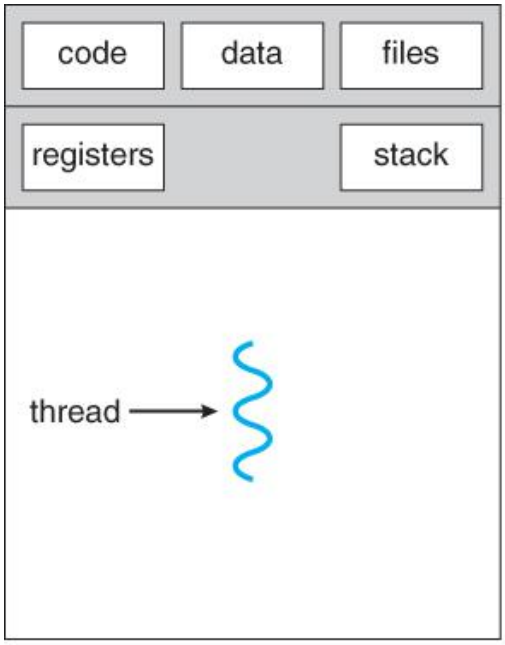
Solutions/work arounds. Some user-thread libraries have addressed these problems with several work-arounds. First timeslice monopolization can be controlled with an external monitor that uses its own clock tick. Second, some SMPs can support user-space multithreading by firing up tasks on specified CPUs then starting the threads from there [this form of SMP threading seems tenuous, at best]. Third, some libraries solve the I/O blocking problem with special wrappers over system calls, or the task can be written for nonblocking I/O.

Kernel-Space Threads

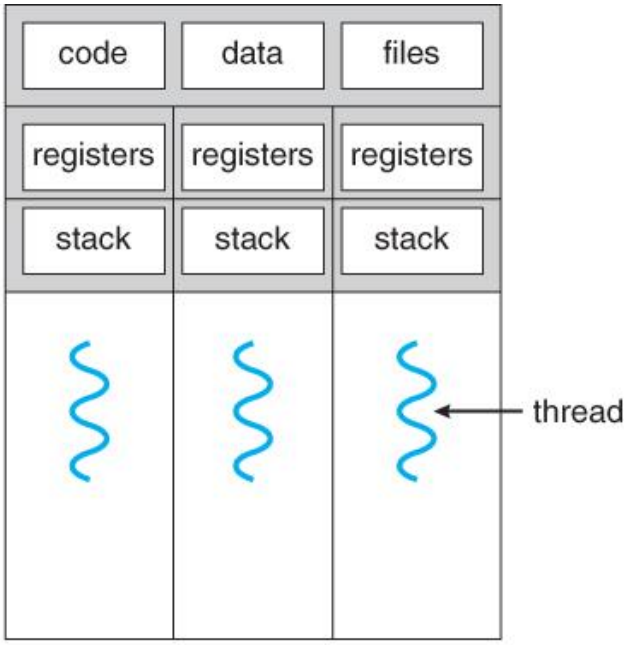
Kernel-space threads often are implemented in the kernel using several tables (each task gets a table of threads). In this case, the kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user->kernel-> user and loading of larger contexts, but initial performance measures indicate a negligible increase in time.

Advantages. Since the clocktick will determine the switching times, a task is less likely to hog the timeslice from the other threads within the task. Also I/O blocking is not a problem.

Lastly, if properly coded, the process automatically can take advantage of SMPs and will run incrementally faster with each added CPU.




single-threaded process




multithreaded process