XXXIII Неделя науки СПбГПУ. Материалы межвузовской научно-технической конференции. Ч.IX: С.76-77, 2005.
© Санкт-Петербургский государственный политехнический университет, 2005.

УДК 681.3.07

Wei Wang (graduate student, Intelligent Systems Program),
M.V.Khloudova, PhD, associate prof.

POSIX THREADS PROGRAMMING

Threads are cousins to UNIX processes though they are not processes themselves. In UNIX, a process contains both an executing program and a bundle of resources such as the file descriptor table and address space. In Mach, a task contains only a bundle of resources; threads handle all execution activities. All threads associated with a given task share the task's resources. Thus a thread is essentially a program counter, a stack, and a set of registers -- all the other data structures belong to the task. Since threads are very small compared with processes, thread creation is relatively cheap in terms of CPU costs. As processes require their own resource bundle, and threads share resources, threads are likewise memory frugal. Additionally, threads can increase performance in a uniprocessor environment when the application performs operations that are likely to block or cause delays, such file or socket I/O. For UNIX systems, a standardized programming interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library – though the library may be part of another library, such as libc. There are several drafts of the POSIX threads standard. It is important to be aware of the draft number of a given implementation; because there are differences between drafts that can cause problems POSIX (Portable Operating System Interface) threads are a great way to increase the responsiveness and performance of your code. The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. That saves a lot of CPU time, making thread creation ten to a hundred times faster than new process creation. Because of this, you can use a whole bunch of threads and not worry too much about the CPU and memory overhead incurred.

The only active entities in a UNIX system are the processes. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Most versions of UNIX allow a process to create additional threads once it starts executing. UNIX is a multiprogramming system, so multiple, independent processes may be running at the same time. Each user may have several active processes at once; so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the user is absent, dozens of background processes, called daemons**,** are running. These are started automatically when the system is booted. UNIX is a multiprogramming system, so multiple, independent processes may be running at the same time. Each user may have several active processes at once; so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the user is absent, dozens of background processes, called daemons, are running. These are started automatically when the system is booted. Processes are named by their PIDs. When a process is created, the parent is given the child’s PID, as mentioned above. If the child wants to know its own PID, there is a system call, getpid, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.

When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. When a process creates another new process, using fork (), the new process is considered the child and the original process is considered the parent. This creates a hierarchical relationship that can be handy, especially when waiting for child processes to terminate. That's because with POSIX threads this hierarchical relationship doesn't exist. While a main thread may create a new thread, and that new thread may create an additional new thread, the POSIX threads standard views all your threads as a single pool of equals. So the concept of waiting for a child thread to exit doesn't make sense. The POSIX threads standard does not record any "family" information. This lack of genealogy has one major implication: if you want to wait for a thread to terminate, you need to specify which one you are waiting for by passing the proper tid to pthread_join.