“Process management 2” review

- the execution is broken down into processes; besides a program (code) a process has a *state* which is stored in *process control block* (PCB)
- the code for the user mode of the process is written by an application programmer; in user mode process does useful things
- the code for the kernel mode of the process is written by an OS designer and the collection of all this code is called an *operating system* (it’s not a definition but a good way of looking at it)
- kernel side of a process coordinates process execution (creates new processes, terminates current process, makes scheduling decisions - decided what process to run next)
- when the next process to run is selected the kernel side of a process calls the routine that does *context switch*; this routine saves the state of the current process in it’s PCB, loads the state of the next process from this process’ PCB and restarts it
- for the restarted process it appears that the context switch routine has just finished

Lecture 5 “Threads”

- process as a unit of scheduling and a unit of resource allocation
- processes vs. threads
- why use and what to program with threads
- two types of threads:
  - user-level threads
  - kernel-level threads

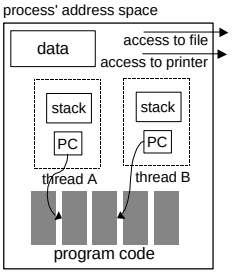
Two aspects of a process

- A process can be viewed two ways:
  - A unit of resource ownership
    - a process has an address space, containing program code and data
    - A process may have open files, may be using an I/O device, etc.
  - A unit of scheduling
    - the CPU scheduler dispatches one process at a time onto the CPU
    - associated with a process are values in the PC, SP, and other registers
- Insight (~1988) — these two are usually linked, but they don’t have to be; many recent operating systems attempt to separate these two aspects (modern Unices, Windows NT):
  - process = unit of resource ownership
  - thread = unit of scheduling

Processes vs. threads

- process = unit of resource ownership
  - a process (sometimes called a heavyweight process) has:
    - address space
    - program code
    - global variables, heap, stack
    - OS resources (files, I/O devices, etc.)
- thread = unit of scheduling
  - a thread (sometimes called a lightweight process - watch the meaning, the term is used for something else) is a single sequential execution stream within a process
  - a thread shares with other threads:
    - address space, program code
    - global variables, heap
    - OS resources (files, I/O devices)
  - a thread has its own:
    - registers, Program Counter (PC)
    - stack, stack pointer (SP)

Processes vs. threads (cont.)



- A thread is bound to a particular process
  - A process may contain multiple threads of control inside it
  - Threads can block, create children, etc.
- All of the threads in a process:
  - Share address space, program code, global variables, heap, and OS resources
  - Execute concurrently (have its own register, PC, SP, etc. values)

Why use threads

- **intuitive, easy to program**:
  a process with multiple threads makes a great server (e.g., printer server):
  - have one server process, many “worker” threads if one thread blocks (e.g., on a read), others can still continue executing
  - threads can share common data; don’t need to use inter-process communication
  - can take advantage of multiprocessors
- **efficiency of execution:**
  - cheap to create — only need a stack and storage for registers
  - use very little resources — don’t need new address space, global data, program code, or OS resources
  - context switches are fast — only have to save / restore PC, SP, and registers
- but… no protection between threads!

What to program with threads

- Good programs to multithread:
  - Programs with multiple independent tasks (debugger needs to run and monitor program, keep its GUI active, and display an interactive data inspector and dynamic call grapher)
  - Server which needs to process multiple requests simultaneously
  - Repetitive numerical tasks — break large problem, such as weather prediction, down into small pieces and assign each piece to a separate thread
- Programs difficult to multithread:
  - Programs that don't require any multiprocessing (99% of all programs)
  - Programs that require multiple processes (maybe one needs to run as root)

User-level threads

user-level threads - provide a library of functions to allow user processes to manage (create, delete, schedule) their own threads
OS is not aware of threads!

- advantages:
  - doesn't require modification to the OS
  - simple representation — each thread is represented simply by a PC, registers, stack, and a small control block, all stored in the user process' address space
  - simple management — creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
  - fast — thread switching is not much more expensive than a procedure call
  - flexible — CPU scheduling (among those threads) can be customized to suit the needs of the algorithm

User-level threads (cont.)

- disadvantages:
  - Lack of coordination between threads and OS kernel
    - Process as a whole gets one time slice
    - Same time slice, whether process has 1 thread or 1000 threads
    - Also — up to each thread to relinquish control to other threads in that process
  - Requires non-blocking system calls (i.e., a possibly multithreaded kernel)
    - Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process

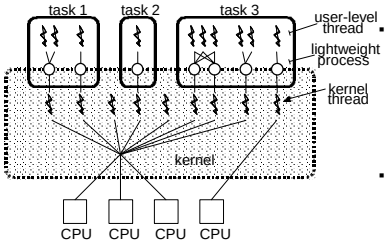
Kernel-level threads

- Kernel-level threads - kernel provides system calls to create and manage threads
  - advantages
    - Kernel has full knowledge of all threads - scheduler may choose to give a process with 10 threads more time than process with only 1 thread
    - Good for applications that frequently block (e.g., server processes with frequent interprocess communication)
  - disadvantages:
    - Slow — thread operations are 100s of times slower than for user-level threads
    - Significant overhead and increased kernel complexity — kernel must manage and schedule threads as well as processes - requires a full thread control block (TCB) for each thread

Multithreading Models

- Many-to-One(Solaris' green threads) – all user threads are mapped to one
  kernel threads same problems as with user threads
- One-to-One (Windows NT/2000, OS/2) – one user thread to one kernel thread
  - programmer has better control of concurrency
  - does not block the process on one thread blocking
  - may potentially waste resources
- Many-to-Many – multiplexes many user-level threads to a smaller or equal number of kernel-level threads
  - allocation is specific to a particular machine (more k. threads may be allocated on a multiprocessor)

Two-Level Thread Model (Digital UNIX, Solaris, IRIX, HP-UX)



ULTs can be multiplexed on lighweight processes
ULTs can be "pinned " to a processor
kernel thread may not have a corresponding LWP

- User-level threads for user processes
  - "Lightweight process" (LWP) serves as a "virtual CPU" where user threads can run
- Kernel-level threads for use by kernel
  - One for each LWP
  - Others perform tasks not related to LWPs
- OS supports multiprocessor systems