Threads vs. Processes

Hank Levy

Processes and Threads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a "thread of control", which defines where the process is currently executing (basically, the PC and registers)
- Creating a new process is costly, because of all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly, because most communication goes through the OS

Parallel Programs

- Suppose I want to build a parallel program to execute on a multiprocessor, or a web server to handle multiple simultaneous web requests. I need to:
  - create several processes that can execute in parallel
  - cause *each* to map to the *same* address space (because they're part of the same computation)
  - give each its starting address and initial parameters
  - the OS will then schedule these processes, in parallel, on the various processors
- Notice that there's a lot of cost in creating these processes and possibly coordinating them. There's also a lot of duplication, because they all share the same address space, protection, etc……

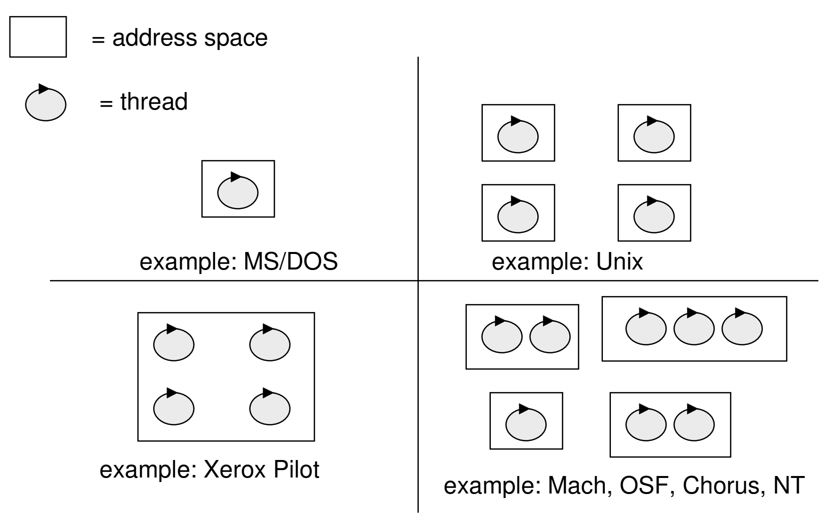
"Lightweight" Processes

- What's similar in these processes?
  - They all share the same code and data (address space)
  - they all share the same privileges
  - they share almost everything in the process
- What don't they share?
  - Each has its own PC, registers, and stack pointer
- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?

Threads and Processes

- **Some newer operating systems (Mach, Chorus, NT) therefore support two entities:**
  - **the <u>process</u>, which defines the <u>address space</u> and general process attributes**
  - **the <u>thread</u>, which defines a sequential execution stream within a process**
- **A thread is bound to a single process. For each process, however, there may be many threads.**
- **Threads are the unit of scheduling; processes are *containers* in which threads execute.**

How different OSs support threads



Separation of Threads and Processes

- Separating threads and processes makes it easier to support multi-threaded applications
- Concurrency (multi-threading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
- So, multi-threading is useful even on a <u>uni</u>processor.
- But, threads, even when supported separately from processes in the kernel, are too slow.

Kernel Threads

- Kernel threads still suffer from performance problems.
- Operations on kernel threads are slow because:
  - a thread operation still requires a kernel call
  - kernel threads may be overly general, in order to support needs of different users, languages, etc.
  - the kernel doesn't trust the user, so there must be lots of checking on kernel calls

User-Level Threads

- **To make threads really fast, they should be implemented at the <u>user</u> level**
- **A user-level thread is managed entirely by the run-time system (user-level code that is linked with your program).**
- **Each thread is represented simply by a PC, registers, stack and a little control block, managed in the user's address space.**
- **Creating a new thread, switching between threads, and synchronizing between threads can all be done without kernel involvement.**

(Old) Example of thread performance

	Ultrix	**Topaz**	**FastThreads**
Fork	11320	1208	39
Signal/Wait	1846	229	52

(perf. on a 3 MIPs processor)

•Ultrix: 1 thread per address space

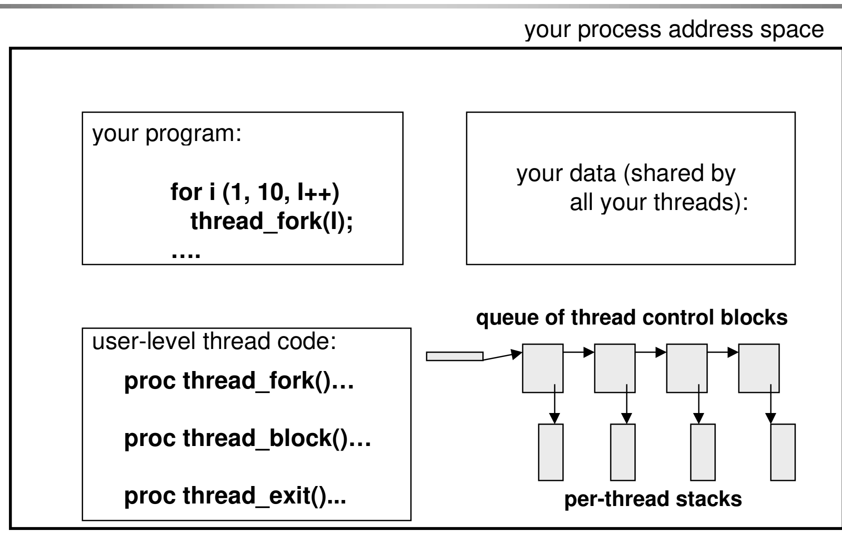
•Topaz: multiple threads per address space

•FastThreads: multiple user threads per address space

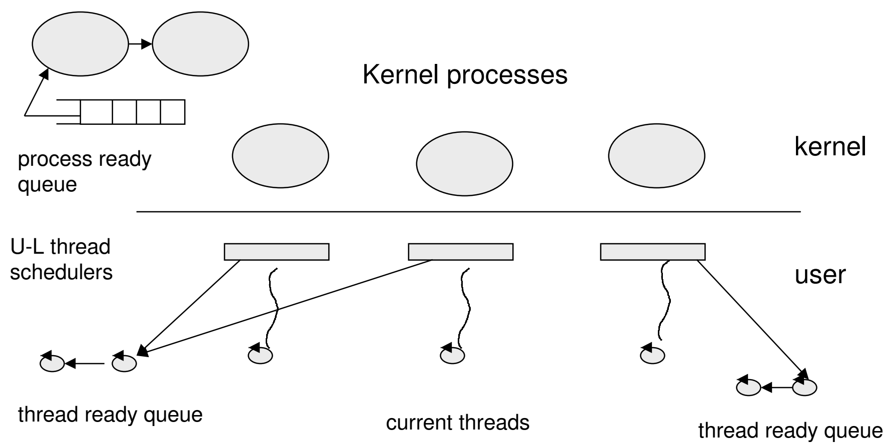
Example U-L Thread Interface

- t = thread_fork(initial context)
  - create a new thread of control
  - sometimes show as
    - thread_create(), thread_setstate(initial context)
- thread_stop()
  - stop the calling thread, sometimes called thread_block
- thread_start(t)
  - start the named thread
- thread_yield()
  - voluntarily give up the processor
- thread_exit()
  - terminate the calling thread, somtimes called thread_destroy.

Key Data Structures



User-Level Theads: Implementation



Threads

- **The OS is a huge multi-threaded program.**
- **Multi-threading is useful for applications as well.**
- **User-Level threads can be 100x faster than similar kernel thread operations.**
- **But user-leve threads suffer from a problem of poor integration; because they're invisible to the OS, the OS can make poor scheduling decisions, e.g.:**
  - **running a process that has no work (is running an idle thread)**
  - **blocking a process whose u-l thread has requested I/O, even though other threads can run in that process**
  - **removing a process whose u-l thread is holding a lock**
- **Solving this requires communication between the kernel and the u-l thread manager.**