CSE451 Threads
Spring 2001

Gary Kimura

Lecture #5

April 4, 2001

Today

- Threads a versatile programming construct that supercharges up dreary old processes

Motivation for Threads

- An OS process includes numerous things:
  - An address space (defining all the code and data)
  - System resources and accounting information
  - A "*thread of control*" defining where the process is currently executing (basically, the PC and registers)
- Creating a new process can be costly, because of all of the structures that must be allocated
- And communicating among processes is costly, because most communication goes through the OS
- However a multiple *thread of control* paradigm is a wonderful programming tool that we really want to support in some useful way

Parallel Programs Without Threads

- Suppose we want to build a parallel program to execute on a multiprocessor, or a web server to handle multiple simultaneous web requests. We need to:
  - Create several processes that can execute in parallel
  - Cause each to share data, by possibly mapping to the same address space
  - Give each a starting address and initial set of parameters
  - The OS will then schedule these processes, in parallel, on the various processors, and maybe not even know they are really part of the same job

Cost of Doing Parallel Programs

- It can be costly doing parallel programs in an OS that does not support multi-threaded processes.
  - There is a cost with creating and coordinating all of the processes
  - There is also a lot of duplication, because they all share the same address space, opened files, protection, etc
- So any support that the OS can give for doing multi-threaded programs is a win

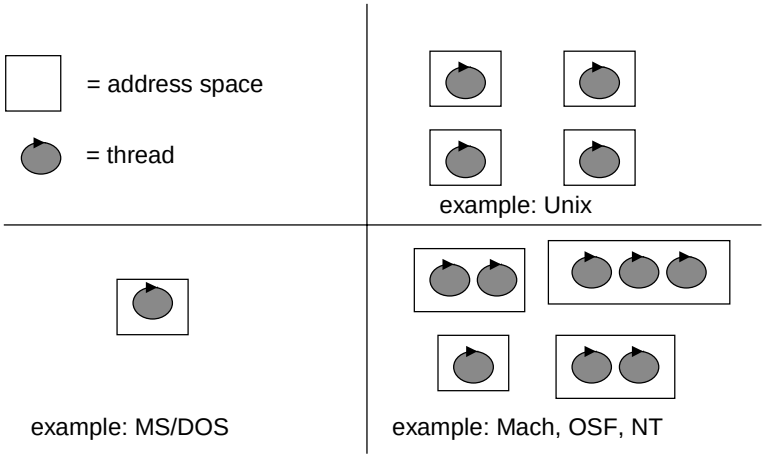
Lightweight Processes

- Looking at the previous slides, we need to ask ourselves, What's similar in all these processes?
  - They all share the same code and data (address space)
  - They share almost everything in the process (e.g., opened files)
- What don't they share?
  - Each has its own PC, registers, and stack pointer
- So the idea is to
  - Separate the process (address space, accounting, etc.) from that of the minimal "*thread of control*" (PC, SP, registers)?
- And we get multi-threaded processes

Adding Threads to Processes

- Some newer OSs (Mach, NT) support two entities:
  - The *process*, which defines the address space and general process attributes
  - The *thread*, 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

The Ways Different OS's do 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 uniprocessor.
- There are essentially two ways that people do multi-threading
  - Kernel threads – Direct OS implementation and support
  - User-level threads – Done at user level

Kernel Threads

- Each thread is a kernel object
  - It is a schedulable entity.
  - It goes through the same state transitions as we saw for processes (ready, running, and waiting)
- There are some performance issues
  - A thread cannot directly yield control to another thread in the same process
  - Kernel threads may be overly general, in order to support needs of different users, languages, etc.

User-Level Threads

- User level threads is essentially a way to mimic multi threading in a user level library
  - A user-level thread is managed entirely by the run-time system (user-level code that is linked to 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.
- It can be really fast provided the application is well behaved
- The original Windows running on DOS was done along this line

Example Kernel Thread Interface

- Some of the more common kernel thread interface calls are:
  - Create Thread – Add a new thread to the process
  - Terminate Thread – Destroy an existing thread
  - Impersonate Thread – This is particularly important for server applications where each thread in a process needs to perform the task in the context of the client
  - A wait or synchronization call – This is allows threads to synchronize among themselves and the OS in general
  - A set and query Thread Information – Provide an interface to modify and query a threads state/status.

User-Level threads Interface

- Some of the more common user level thread interface calls are:
  - Create Thread – Add a new thread to the process
  - Stop Thread – Allows a thread to block itself
  - Start Thread – Allows another thread to start a blocked thread
  - Yield Thread – Voluntarily gives up the CPU to another thread in the process
  - Exit Thread – Terminates the calling thread

What's the best thread approach?

- Choice either side, you'll have plenty of company
- Personally I like kernel threads
- What do I like about Kernel threads:
  - More robust than user-level threads
  - Allow impersonation
  - Easier to tune the OS CPU scheduler to handle multiple threads in a process
  - A thread doing a wait on a kernel resource (like I/O) does not stop the process from running
- What about user-level threads
  - A lot faster if programmed correctly
  - Can be better tuned for the exact application
- Note that user-level threads can be done on any OS

Some Things to Remember

- Each thread shares everything with all the other threads in the process
  - They can read/write the exact same variables, so they need to synchronize themselves
  - They can access each other's runtime stack, so be very careful if you communicate using runtime stack variables
  - Each thread should be able to retrieve a unique thread id that it can use to access *thread local storage*
- Multi-threading is great, but use it wisely

Next time

- So we have all these entities that want to run on the machine. How do we schedule them to run?