Chapter 4: Threads
Logistics

- Hwk 2 posted Tuesday, due next Tuesday.
  - Any questions? Make sure you at least scan it over before the discussion to come armed with questions.

- Programming assignment 1 will be posted by 5pm today.
  - Two week assignment.
So far…

- We’ve rapidly covered topics on:
  - OS structure
  - How programs are executed in the form of processes.

- The process concept is critical.
  - Now is the time to ask any questions to clarify process concepts.
  - This discussion builds upon processes.
Threads

- Last two lectures: processes
  - Programs in execution
  - Communication between processes via OS-provided shared memory and messages.

- Threads are an extension of the process concept.
  - Born out of a desire for concurrency within a running program.
  - Like processes, they represent units of execution.
  - Unlike processes, they share state with the process that created them.

- Threads make concurrency programmable.
  - In two lectures, we talk about concurrency at a deeper level.
Sharing State

- Recall that a child forked from a parent process is a replica of the parent.
  - Modifications to the state (i.e., variables) are not visible to either side after forking.
  - Exchange of information must pass through an OS-provided channel.

- Threads are based on a model where threads spawned from a parent can see:
  - State residing within the parent.
  - State residing within other threads.
  - All through standard variables, not OS-provided mechanisms.

- From a programmability perspective, this makes building concurrent programs easier.
Single and Multithreaded Processes

Note!
These are shared!
Benefits

- Why thread?
  - Responsiveness
    - Example: Threads in a GUI.
  - Resource Sharing
  - Economy
  - Scalability
    - Threaded programs can scale to more cores.
    - Single processes w/out threads cannot.
Threads and parallel systems

- Threads fit naturally onto parallel computers.
  - Such as multicore.

- Allows a programmer to build one program to use all of the cores.
  - Instead of a flock of coordinated processes.

- The thread abstraction facilitates:
  - Shared address spaces.
  - Independent units of execution that can be scheduled onto the parallel processing elements (aka, cores).
Multicore Programming

- Multicore systems are putting pressure on programmers.
- How to utilize the parallel cores?

- Challenges facing a programmer include
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
Note that servers based on fork() do this too. Threads make this 1) faster, 2) more flexible by allowing child threads to share state with the parent w/out resorting to OS mechanisms.
Thinking about concurrency

- Say we have a program that we want to run that can be divided into a set of distinct tasks or subprograms.
  - Examples include: independent loop iterations, independent function calls, etc…

- We can write the program as a sequence of these tasks (see next slide)
- …or we can write the program where we split the blocks into groups that can legally run concurrently (two slides away).
  - Threads support the execution of these concurrent sequences.
Concurrent Execution on a Single-core System

Four distinct sequences that can execute concurrently: T1, T2, T3, T4
Instead of interleaving in a single sequence, we can actually run some tasks in parallel.
Splitting the program up

- How do we achieve this concurrency? How do we decide what each thread can do?
- This is a nontrivial task.

In a nutshell, it requires us to:
  - Identify what the program is to do – a set of tasks.
  - Identify dependencies between the tasks.
  - Partition the set of tasks into sequences that are not dependent on each other.
    - Think of the topological sort algorithm applied to a graph.
    - Graph is dependencies, topological sort reveals independent sequences.
Threads of multiple flavors

- So, given a program that we’ve decided we can split into distinct, separable sequences of execution, we need to write threads to do the work.

- We have two choices:
  - User-space threads
  - Kernel-space threads

- The distinction has consequences:
  - Performance
  - Flexibility
  - Portability
User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
Kernel Threads

- Supported by the Kernel

- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
User vs kernel

- Kernel threads are scheduled by the kernel.
- User threads are not.
  - Unless....

- A relationship is established mapping user threads onto kernel threads.
- A few ways exist for performing this mapping.
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
Many-to-One Model
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
One-to-one Model
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package
Many-to-Many Model
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
Two-level Model

user thread

kernel thread
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
Java Threads

- Java threads are managed by the JVM

- Typically implemented using the threads model provided by underlying OS

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
OpenMP

- Set of compiler directives for creating threads in C, C++ and Fortran programs.

- Very easy to use relative to pthreads.
  - Compiler responsible for creating and managing threads based on directives placed in source code.
  - Flexible, but not as flexible as explicit threads like pthreads, win32 threads, etc...

- Worth looking at if you have existing sequential code that you want to rapidly parallelize to use threads.
  - Supported in most compilers, including Visual Studio C++, Intel C/C++/Fortran, GNU compilers, etc...
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?

- This varies across systems.

- Some duplicate all threads in the process, some duplicate just the thread where `fork()` was called.
Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
Signals

- What signals exist?

- Off to the terminal for an example.

magnolia:examples matt$ kill -l
1) SIGHUP    2) SIGINT    3) SIGQUIT    4) SIGILL
  5) SIGTRAP   6) SIGABRT   7) SIGEMT    8) SIGFPE
  9) SIGKILL   10) SIGBUS   11) SIGSEGV   12) SIGSYS
 13) SIGPIPE   14) SIGALRM  15) SIGTERM   16) SIGURG
 17) SIGSTOP   18) SIGTSTP  19) SIGCONT   20) SIGCHLD
 21) SIGTTIN   22) SIGTTOU  23) SIGIO     24) SIGXCPU
 25) SIGXFSZ   26) SIGVTALRM 27) SIGPROF   28) SIGWINCH
 29) SIGINFO   30) SIGUSR1  31) SIGUSR2
Thread Pools

- Create a number of threads in a pool where they await work

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

- This is common in server applications.
  - Becoming common in desktop systems due to multicore.
  - Apple “Grand Central” feature in Snow Leopard looks like it has some thread-pool-like features. Not sure exactly what it is yet though beyond basic descriptions on the web.
Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library.
- This communication allows an application to maintain the correct number kernel threads.
Operating System Examples

- Windows XP Threads
- Linux Thread
Windows XP Threads

**ETHREAD**
- thread start address
- pointer to parent process

**KTHREAD**
- scheduling and synchronization information
- kernel stack

**TEB** (Thread environment block)
- thread identifier
- user stack
- thread-local storage

kernel space

user space
## Linux Threads

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>
Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)
Linux Threads

- Linux refers to them as *tasks* rather than *threads*

- Thread creation is done through `clone()` system call

- `clone()` allows a child task to share the address space of the parent task (process)
End of Chapter 4