• Last class:
  – Processes

• Today:
  – Threads
Why Threads?
Advantages of Threads

- Improve Responsiveness
  - Ideally, a thread is always ready

- Resource Sharing
  - All the stuff is easily accessible

- Economy of Resources
  - Thread resources are cheaper than process resources

- Utilization of Multiprocessors
  - Get all of them running
Multi-Threaded vs. Single-Threaded

<table>
<thead>
<tr>
<th></th>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**single-threaded process**

<table>
<thead>
<tr>
<th></th>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**multithreaded process**

thread → 

thread →
Terminology

• **Multiprogramming**
  – Run multiple processes *concurrently* on a single processor
  – OS choose which process to run out of multiple

• **Multiprocessing**
  – Run multiple processes on multiple processors
  – OS manages mapping of processes to processors

• **Multithreading**
  – Define multiple *execution contexts* in a single address space
  – OS manages mapping of contexts (threads) to an address space
  – OS manages mapping of threads to processor(s)
What’s a Thread?

- Thread of Execution on CPU
  - Program counter
  - Registers
- Memory
  - Address space (process)
  - Stack -- per thread
- I/O
  - Share files, sockets, etc. (process)
Working with Threads

• In a C program
  – `main()` procedure defines the first thread
  – C programs always start at `main`

• Create a second thread
  – Allocate resources to maintain a second execution context in same address space
    • Think about what process fields will be necessary for a thread
  – Supply a procedure name to start the new thread’s execution
Why Threads vs. Processes

• Easier to create than a new process
• Less time to terminate a thread than a process
• Less time to switch between two threads within the same process
• Less communication overheads
  – Communicating between the threads of one process is simple because the threads share everything: address space
Which is Cheaper?

- Create new process or create new thread (in existing process)
- Context switch between processes or threads
- Interprocess or inter-thread communication
- Sharing memory between processes or threads
- Terminate a process or terminate a thread (not last one)
Threading Models
Threading Models

• Programming: *Library or system call interface*
  – User-Space Threading
    • Thread management support in user-space library
    • Linked into your program
  – Kernel Threading
    • Thread management support in the kernel
    • Invoked via system call

• Scheduling: *Application or kernel scheduling*
  – May create user-level or kernel-level threads
    • NOTE: CPU only runs kernel threads!
User-Space Threads

• Thread management support in user-space library
  – Sets of functions for creating, invoking, and switching among threads

• Linked into your program
  – Thread libraries

• Examples
  – POSIX Threads (PThreads)
  – Win32 Threads
  – Java Threads
Kernel Threads

- Thread management support in kernel
  - Sets of system calls for creating, invoking, and switching among threads
- Supported and managed directly by the OS
  - Thread objects in the kernel
- Nearly all OS support a notion of threads
  - Linux -- thread and process abstractions are mixed
  - Solaris
  - Mac OS X
  - Windows XP
  - …
Many-to-one Thread Model

• Many user-level threads correspond to a single kernel thread
  – Kernel is not aware of the mapping
  – Handled by a thread library
• How does it work?
  – Create and execute a new thread
  – Upon *yield*, switch to another thread in the same process
    • Kernel is unaware
  – Upon *wait*, all threads are blocked
    • Kernel is unaware there are other options
    • Can’t wait and run at the same time
One-to-one Thread Model

- One user-level thread per kernel thread
  - A kernel thread is allocated for every user-level thread
  - Must get the kernel to allocate resources for each new user-level thread

- How does it work?
  - Create new thread, including system call to kernel
  - Upon *yield*, switch to another thread in system
    - Kernel is aware
  - Upon *wait*, another thread in the process may run
    - Only the single kernel thread is blocked
    - Kernel is aware there are other options in this process
Many-to-many Thread Model

- A pool of user-level threads maps to a pool of kernel threads
  - Pool sizes can be different (kernel pool is no larger)
  - A kernel thread is pool is allocated for every user-level thread
  - No need for the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread (may map to kernel thread dynamically)
  - Upon `yield`, switch to another thread in system
    - Kernel is aware
  - Upon `wait`, another thread in the process may run
    - If a kernel thread is available to be scheduled to that process
    - Kernel is aware of the mapping between process threads and kernel threads
What kind of problems would you solve with threads?

• Imagine you are building a web server
  – You could allocate a pool of threads, one for each client
    • Thread would wait for a request, get content file, return it
    – How would the different thread models impact this?

• Imagine you are building a web browser
  – You could allocate a pool of threads
    • Some for user interface
    • Some for retrieving content
    • Some for rendering content
  – What happens if the user decided to stop the request?
    – Mouse click on the stop button
Summary

• Threads
  – A mechanism to improve performance and CPU utilization

• Kernel and User-space threads
  – Kernel threads are real, schedulable threads
  – User-space may define its own threads (but not real)

• Threading Models and Implications
• Next time: More Threads