Lecture 6: Threads
Chapter 4
Single and Multithreaded Processes

![Diagram showing the difference between single-threaded and multithreaded processes. The single-threaded process has a single stack, while the multithreaded process has multiple stacks for each thread.]
Benefits of multi-threaded programming:
• Responsiveness
• Resource sharing
• Economy (Solaris example: 30X > expensive to create a process, 5X > expensive to context-switch processes)
• Scalability
Multicore Programming

- Each core seen by the OS as a separate processor

<table>
<thead>
<tr>
<th>single core</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>core 1</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>core 2</th>
<th>T₂</th>
<th>T₄</th>
<th>T₂</th>
<th>T₄</th>
<th>T₂</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Challenges in programming for multicore systems
  - Identifying parallelism (dividing activities)
  - Load balance
  - Data splitting
  - Data dependency
  - Testing and debugging
User- vs. Kernel-level Threads

From W. Stallings, Operating Systems, 6th Edition
User vs. Kernel-level Threads

User-level threads:
- Thread management done by user-level threads library:
  - create/destroy threads;
  - message passing or data sharing between threads;
  - scheduling threads;
  - saving/restoring threads contexts.
- Kernel not aware of the existence of threads: sees only the process
- Three primary thread libraries: POSIX Pthreads; Win32 threads; Java threads
- Advantages over kernel-level threads:
  - Thread switching
  - Application-specific scheduling
  - No need of special support from OS
- Disadvantages:
  - One blocking system call in a user-level thread blocks all threads in the same process
  - No advantage from multiprocessing
Kernel-Level Threads

- All work done in the kernel
- Most common OS have kernel-level threads
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

From W. Stallings, Operating Systems, 6th Edition
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many (with the variation Two-level Model)
Many-to-One Model

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

- Each user-level thread maps to kernel thread
- Disadvantage: creating a user thread requires creating a kernel thread
- Used in:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
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
Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread.

**Examples**
- IRIX
- HP-UX
- Tru64 UNIX
- Solaris 8 and earlier
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
Two Popular Thread Libraries

**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 and Implementing the Runnable interface
What is the output of the following code and why?

```c
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
main(int argc, char *argv[])
{
  pthread_t tid, tidl; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */
  if (argc != 2) {
    fprintf(stderr, "usage: a.out\integer value>\n\n"); exit();
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1])); exit();
  }
  pthread_attr_init(&attr);
  /* create the threads*/
  pthread_create(&tid,&attr,runner,argv[1]);
  pthread_create(&tidl,&attr,runner,argv[1]);
  pthread_join(tid,NULL);
  pthread_join(tidl,NULL);
  printf(sum = %d
,sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
  int upper = atoi(param);
  int i;
  int sum = 0;
  if (upper > 0) {
    for (i = 1; i <= upper; i++)
      sum += i;
  }
  pthread_exit(0);
}
```
Threading Issues in OS Design

- Semantics of `fork()` and `exec()` system calls
  - Does `fork()` duplicate only the calling thread or all threads?

- Thread cancellation of target thread. Challenge: thread has resources; thread in the middle of updating data.
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

- Signal handling
  - 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
Thread pools
- Create a number of threads in a pool where they await work
- Advantages?
- Ideas how to determine the size of the thread pool?

Thread-specific data
- Data specific to the thread (e.g., transaction id)
- Support provided by thread library

Scheduler activations
- Enable communication to maintain the appropriate number of kernel threads allocated to the application
- Upcalls: communication mechanism from kernel to thread library
Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Support for many-to-many model in the *Fiber* library
- 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 does not distinguish between threads and processes
- Therefore, tasks rather than threads or processes
- Thread creation is done through `clone()` system call (process creation through `fork()`)
- `clone()` allows a child task to share the address space of the parent task (process)
- How much sharing between process and thread determined by flags
- Implementation issue?

<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>