CS 550:
Advanced Operating Systems
Processes and Threads

Ioan Raicu
Computer Science Department
Illinois Institute of Technology

CS 550
Advanced Operating Systems
February 17th, 2011
Outline for Today

• Motivation and definitions
• Processes
• Threads
• Synchronization constructs
• Speedup issues
  – Overhead
  – Caches
  – Amdahl’s Law
How can we make threads cooperate?

- If task can be completely decoupled into independent sub-tasks, cooperation required is minimal
  - Starting and stopping communication
- Trouble when they need to share data!
- Race conditions:

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread A</td>
</tr>
<tr>
<td>readX</td>
<td>readX</td>
</tr>
<tr>
<td>incX</td>
<td>incX</td>
</tr>
<tr>
<td>writeX</td>
<td>writeX</td>
</tr>
<tr>
<td>time --&gt;</td>
<td>time --&gt;</td>
</tr>
<tr>
<td>Thread B</td>
<td>Thread B</td>
</tr>
<tr>
<td>readX</td>
<td>readX</td>
</tr>
<tr>
<td>incX</td>
<td>incX</td>
</tr>
<tr>
<td>writeX</td>
<td>writeX</td>
</tr>
</tbody>
</table>

- We need to force some serialization
  - Synchronization constructs do that!
A lock (mutual exclusion, mutex) guards a critical section in code so that only one thread at a time runs its corresponding section
  – acquire a lock before entering crit. section
  – releases the lock when exiting crit. section
  – Threads share locks, one per section to synchronize

If a thread tries to acquire an in-use lock, that thread is put to sleep
  – When the lock is released, the thread wakes up with the lock! (blocking call)
Lock / mutex syntax example in PThreads

```c
pthread_mutex_t lock = PTHREAD_Mutex_INITIALIZER;
int x;

threadA() {
    int temp = foo(x);
    pthread_mutex_lock(&lock);
    x = bar(x) + temp;
    pthread_mutex_unlock(&lock);
    // continue…
}

threadB() {
    int temp = foo(9000);
    pthread_mutex_lock(&lock);
    baz(x) + bar(x);
    x *= temp;
    pthread_mutex_unlock(&lock);
    // continue…
}
```

• But locks don’t solve everything…

• Problem: potential deadlock!

```c
threadA() {
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    // continue…
}

threadB() {
    pthread_mutex_lock(&lock2);
    pthread_mutex_lock(&lock1);
    // continue…
}
```
A condition variable (CV) is an object that threads can sleep on and be woken from

- *Wait* or *sleep* on a CV
- *Signal* a thread sleeping on a CV to wake
- *Broadcast* all threads sleeping on a CV to wake
- I like to think of them as thread pillows…

Always associated with a lock!

- Acquire a lock before touching a CV
- Sleeping on a CV releases the lock in the thread’s sleep
- If a thread wakes from a CV it will have the lock

Multiple CVs often share the same lock
Outline for Today

- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl’s Law
More threads does not always mean better!
  – I only have two cores…
  – Threads can spend too much time *synchronizing* (e.g. waiting on locks and condition variables)

Synchronization is a form of overhead
  – Also communication and creation/deletion overhead
• Caches are often one of the largest considerations in performance
• For multicore, common to have independent L1 caches and shared L2 caches
• Can drive domain decomposition design
Applications can almost never be completely parallelized; some serial code remains.

- \( s \) is serial fraction of program, \( P \) is # of processors.
- Amdahl’s law:

\[
\text{Speedup}(P) = \frac{\text{Time}(1)}{\text{Time}(P)} \\
\leq \frac{1}{(s + \frac{(1-s)}{P})}, \text{ and as } P \rightarrow \infty \\
\leq \frac{1}{s}
\]

- Even if the parallel portion of your application speeds up perfectly, your performance may be limited by the sequential portion.
Super-linear speedup is possible

Multicore is hard for architecture people, but pretty easy for software

Multicore made it possible for Google to search the web
Quiz Answers!

- Super-linear speedup is possible
  True: more cores means simply more cache accessible (e.g. L1), so some problems may see super-linear speedup

- Multicore is hard for architecture people, but pretty easy for software
  False: parallel processors put the burden of concurrency largely on the SW side

- Multicore made it possible for Google to search the web
  False: web search and other Google problems have huge amounts of data. The performance bottleneck becomes RAM amounts and speeds! (CPU-RAM gap)
• Threads can be *awake and ready/running* on a core or *asleep for sync.* (or blocking I/O)

• Use PThreads to thread C code and use your multicore processors to their full extent!
  - `pthread_create()`, `pthread_join()`, `pthread_exit()`
  - `pthread_mutex_t`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`
  - `pthread_cond_t`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

• **Domain decomposition** is a common technique for multithreading programs

• Watch out for
  – Synchronization *overhead*
  – Cache issues (for sharing data, decomposing)
  – Amdahl’s Law and algorithm parallelizability

• Reading Ch. 3

• Programming Assignment – Part 1
Part 1: Peer-to-peer file sharing with centralized index

1. registry(node 1, foo.avi)
2. search(foo.avi)
3. Node1, node2
4. obtain (foo.avi)

Indexing server:
- Foo.avi: Node1
- Bar.c: Node 1
- Foo.avi: Node 2
- Mypic.gif: Node 3
Programming Assignment

• Two entities
  – Central indexing server
    • List of all files at peers
  – Peer (both client and server)
    • [client] Search for a file at the indexing server
    • Download file from a peer, update indexing server
    • [server] listen for download requests and service
  – Provide concurrency at the central indexing server and peer

• Feel free to use any prog language and any mechanism (threads, RPC, RMI, sockets, semaphores...)

CS550: Advanced Operating Systems