Process & Thread Management II

CIS 657

Queues

- Run queues: hold threads ready to execute
  - Not a single ready queue; 64 queues
  - All threads in same queue are treated as same priority
- Sleep queues: hold threads waiting on events
  - Hashed data structure, optimized for searching on wait channel
  - In older versions of the OS, these held processes, not threads

Sleep() and Sleep Queues

- LOOKUP(x) hashes an identifier to a sleep queue (kern_synch.c)
- Ident can be a void *, int, or other (generally a pointer).
- The sleep routines do not determine how many identifiers there are in the system
- Threads can give up the CPU by making a sleep() call (passing wait channel ID and priority at which to be awakened)
Algorithm for Sleep

1. Acquire sched_mutex lock
2. Record the wait channel in the thread structure, and hash the channel to find a sleep queue
3. Set the thread’s priority for when it is awakened, set SLEEPING flag
4. Place thread at end of sleep queue
5. Call mi_switch() to context switch (sched_lock automatically released)

Wakeup() & Wakeup_one()

- Wakeup awakens all processes on a channel
  - Front to back
- Wakeup_one() awakens approximately one thread

Algorithm for Wakeup()

1. Remove thread from sleep queue
2. Recompute user-mode scheduling priority if sleep time > 1 sec
3. If in SLEEPING state,
   1. Make thread RUNNABLE
   2. If in memory, put in run queue; else start swapin process to get it
4. If in STOPPED state, do nothing
Context Switching

- Involuntary context switch: time slice expired or higher-priority process is ready to run
- Voluntary context switch: call to sleep()
- The routine mi_switch() does the dirty work, selecting the next process to run (see /usr/src/sys/kern/kern_synch.c).

Thread State

- A context switch requires changing both kernel- and user-mode context
- Saved in the thread structure (see proc.h)
- mi_switch():
  - Selects thread from highest-priority scheduling queue w/runnable thread
  - Loads process’s context from the TCB

Historic Synchronization:
Uniprocessor Architecture

- Kernel process was best priority
- No other process could enter top half of kernel (no preemption)
- Coordination by ensuring data in stable state before sleep
- Mask interrupts (if needed) to block bottom half processes from data conflicts
FreeBSD 3.0

- The origin of the Big Giant Lock (BGL)
- Acquired by every process on entry to the kernel.
- Serialized access to the kernel (good)
- …but limited to 1 process in kernel at a time (bad on multiprocessor)

FreeBSD 5.0

- First SMP-ready FreeBSD
- Most kernel access now outside the BGL
- 5 levels of locking

The 5 Lock Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Can Sleep</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest</td>
<td>Hardware</td>
<td>no</td>
<td>Memory-interlocked test-and-set</td>
</tr>
<tr>
<td></td>
<td>Spin mutex</td>
<td>no</td>
<td>Spin lock</td>
</tr>
<tr>
<td></td>
<td>Sleep mutex</td>
<td>no</td>
<td>Spin for a while, then sleep</td>
</tr>
<tr>
<td></td>
<td>Lock manager</td>
<td>yes</td>
<td>Sleep lock</td>
</tr>
<tr>
<td>Highest</td>
<td>Witness</td>
<td>yes</td>
<td>Partially ordered sleep locks</td>
</tr>
</tbody>
</table>
Hardware: Memory-interlocked test-and-set

- Memory-interlocked: only allow one processor to use this instruction on memory at one time
- Two component operations, done atomically
  - Read value of memory location, then
  - Write value to same location

Swap vs. Test-and-set

```c
Test-and-set(lock) {
    int one = 1;
    swap(one, lock);
}
```

Spin Lock Using Test-and-set

```c
spinlock(lock) {
    while (test-and-set(lock));
}
spinunlock(lock) {
    lock = 0;
}
```
Use of Spin Locks

- Busy waiting is BAD
- Only for low-contention resources
- Lock typically held for short time
- Result should be that very little time is wasted in the lock over the system
- Do spin locks allow deadlock? Starvation?

Sleep Locks

- Used when we expect resources to be held for significant lengths of time
  - E.g., disk buffers during I/O
- So, queue up using sleep() instead of spinning.
- Releasing the lock must wake up one thread

Sleep Mutex

- Sometimes we don’t know how long the resource will be held
  - Expect short
  - Could be long
- Spin a while, then sleep
- Lets other processes run
- Inverse exponential distribution
The Witness Module

- Above the other types of locks is a set of partially-ordered sleep locks
- This avoids deadlock
- Recall:
  - Circular wait, no preemption, hold-and-wait, mutual exclusion
  - Which condition does this avoid?
- Also makes sure thread doesn’t hold any spin locks

Rules for Acquiring Witness Locks

- A thread may acquire only one lock in each class
- A thread may only acquire a lock in a higher-numbered class than the highest-numbered lock the thread already holds