CS 423 – Operating Systems Design

Lecture 9 – Lock Implementation

Klara Nahrstedt
Fall 2011

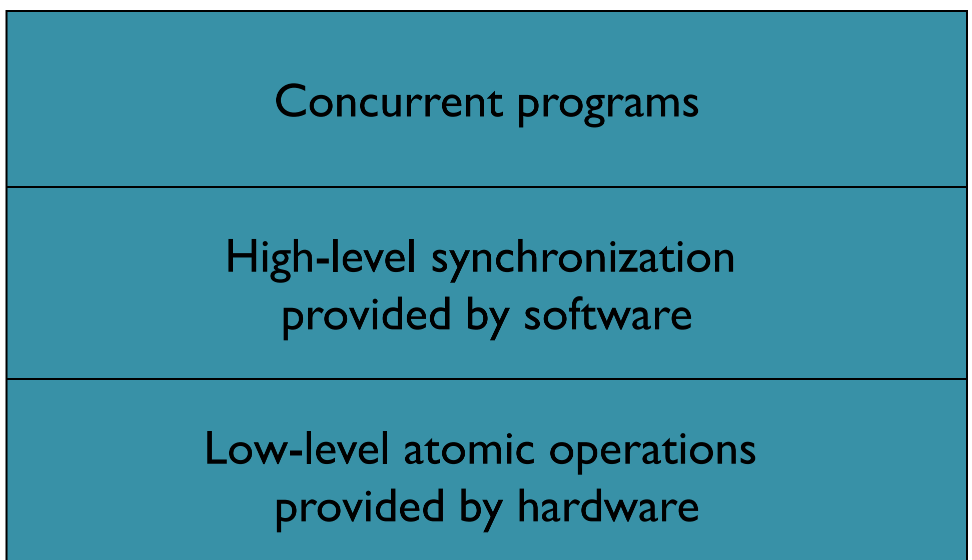
Based on slides by Sam King, Elsa Gunter and Andrew S. Tanenbaum

Overview

- Administrative Issues
  - MP2 will be posted today
- Lock Implementation
- Read Tanenbaum Section 2.3.3-2.3.6

Implementing locks

- Concurrent programs use high-level synchronization operations
  - Used with multiple threads so they need to worry about atomicity (e.g., they use data structures)
  - Can't use the high-level synchronization operations themselves



Interrupt enable/disable for atomicity

- On uniprocessor, operation is atomic as long as context switch does not occur
  - How does thread get context switched out?

  - Prevent context switches at wrong time by preventing these events

Interrupt enable/disable for atomicity

- With interrupt enable/disable, why do we need locks?
  - User program could call interrupt disable before entering critical section and call interrupt enable after leaving
  - What is wrong with this?

Lock impl. #1 (disable interrupts w/ busy waiting)

```
Lock() {
  disable interrupts
  while(value != FREE){
    enable interrupts
    disable interrupts
  }
  value = BUSY
  enable interrupts
}
```

```
Unlock() {
  disable interrupts
  value = FREE
  enable interrupts
}
```

- Why does lock() disable interrupts in the beginning of the function?
- Does this guarantee atomicity and allow progress?

Big picture

- The critical sections are the accesses to the "lock variable" (value on previous slide)
- We use interrupt enable/disable as a "lock" for this critical section
- Can't use higher-level primitives

Lock impl. #1

- Why ok to disable interrupts in lock()'s critical section and not user-mode code?
- Do we need to disable interrupts in unlock()?
- Why does body of while enable, then disable interrupts?

Lock impl. #1

- Why ok to disable interrupts in lock()'s critical section and not user-mode code?
  - Because we are in kernel mode and control everything, won't starve anyone
- Do we need to disable interrupts in unlock()?
  - Yes, because we are accessing protected data
- Why does body of while enable, then disable interrupts?
  - So that a context switch may occur for any pending interrupts so someone else may work

Read-modify-write instructions

- Interrupt disable works on a uniprocessor
  - Does not work on multiprocessor
- Could use atomic load / atomic store instructions (Too Much Milk #3)
- Modern processors provide an easier way with atomic read-modify-write instructions
  - Atomically {read value from memory into register, then write new value to memory location}
  - Test and Set

Test and set

- Var (say X) shared across all processors
- Atomically writes 1 to X memory location (set) and returns previous the value (test)

```
Test_and_set(x) {
  tmp = X
  X = 1
  return(tmp)
}
```

- Note that only 1 process can see a transition from 0 -> 1

Lock impl. #2 (test&set w/ busy waiting)

```
(value initially 0)
Lock() {
  while(test_and_set(value) == 1)
    ;
}
Unlock() {
  value = 0;
}
```

- If lock is free(value = 0), test_and_set sets value to 1 and return 0, so the while loop finishes
- If lock is busy(value = 1), test_and_set doesn't change the value and returns 1, so loop continues

Problem with lock impl. #1 and #2

- Waiting thread uses lots of CPU time waiting for lock to free (Busy Waiting)
- Better for thread to go to sleep and let other threads run
- Strategy for reducing busy-waiting: integrate the lock implementation with the thread dispatcher data structures and have the lock code manipulate thread queues

Interrupt disable / enable, Try 1

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 enable interrupts;
 add thread to lock wait queue;
 switch to next runnable thread;
}
```

Interrupt disable / enable, Try 1

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 enable interrupts;
 add thread to lock wait queue;
 switch to next runnable thread;
}
```

- If interrupt is handled as soon as enabled, lock might be freed, control returned, and thread would go on wait queue even though lock free

Interrupt disable / enable, Try 2

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 add thread to lock wait queue;
 enable interrupts;
 switch to next runnable thread;
}
```

Interrupt disable / enable, Try 2

- What is wrong with the following?

```
lock() {
disable interrupts;
...
if(lock is busy) {
 add thread to lock wait queue;
 enable interrupts;
 switch to next runnable thread;
}
```

- If interrupt is handled as soon as enabled, thread put on preemption queue
- Thread on two queues! Bad – wrong queue (wait) might be used first

Making add to (queue & switch) atomic

- Adding thread to wait queue and switching to the next thread must be **atomic**
- **Solution:**
  - Waiting thread leaves interrupts disabled when it calls switch
  - Next thread to run must be the responsibility of re-enabling interrupts before returning to user code
  - When waiting thread wakes up, it returns from switch with interrupts disabled (from last thread).

- Think of interrupt state as shared variable

Lock impl. #3 (interrupt disable/enable, no busy wait)

```
Lock() {
  disable interrupts
  if(value == FREE) {
    value = BUSY
  } else {
    add thread to wait
      queue for this lock
    switch to next thread
  }
  enable interrupts
}
```

```
Unlock() {
  disable interrupts
  value = FREE
  if(any thread waiting
     for this lock) {
     move wait thread to
       ready queue
     value = BUSY
  }
  enable interrupts
}
```

Handoff locks

- Thread calling unlock() gives lock to waiting thread to guarantee FIFO ordering of lock usage
  - No control over scheduling algorithm, but we do control access to locks
- What does it mean for a thread to add current thread to lock wait queue?
  - Is this the point at which it saves state?
- Why do we need a separate lock queue?

Lock impl. #3

- **Invariant:**
  - All threads promise to have interrupts disabled when they call schedule
  - All threads promise to re-enable interrupts after they get returned to from schedule

Invariant Example

Thread A	Thread B
yield() {	
disable interrupts;	
switch;	
	back from switch
	enable interrupts}
	<user code runs>
	lock() {
	disable interrupts
	...
	switch
back from switch	
enable interrupts}	
<user code runs>	
unlock() // move a to readyq	
yield()	
{disable interrupts	
switch	
	back from switch
	enable interrupts}

Lock impl. #4

- Can't implement locks using test&set without some amount of busy-waiting, but can minimize it
- Idea: use busy waiting only to atomically execute lock code. Give up CPU if busy

Lock impl. #4

```
lock() {
 while(test&set(guard){
 }
  if(value == FREE) {
  value = BUSY
 } else {
  add thread to wait_q
  switch to next run t
 }
 guard = 0
}
```

```
unlock() {
 while(test&set(guard)){
 }
 value = FREE
 if(wait_q nonempty) {
 move thread to ready_q
value = BUSY
}
guard = 0
}
```

Conclusion

- Careful consideration of lock implementation is important
- Consideration of hardware support is important
- Consideration of lock implementation for user and kernel space is important