COS 318: Operating Systems

Mutex Implementation

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Announcements

- Project 1 due tomorrow.
  - Tonight's precept is open questioning.

- A few words about Independent Work: Why you should strongly consider starting it during your junior year:

  1) Helps you get internships between jr and sr year.

  2) Improves the detail of the reference letter a prof can write for you during fall of your senior year.

  3) Let's us nominate you for awards with fall deadlines like this one:

     http://cra.org/awards/undergrad/

Roadmap: Where are we & how did we get here?

- OS: Abstractions & resource management
  - 1 Abstraction: Process
  - 1 type of resource management: CPU scheduling

- Scheduling processes involves preempting and interleaving them.
- This arbitrary interleaving requires special thought about critical sections and mutual exclusion

- And that is how we got to the discussion of how to buy milk.
- Today: How to implement Mutual Exclusion?

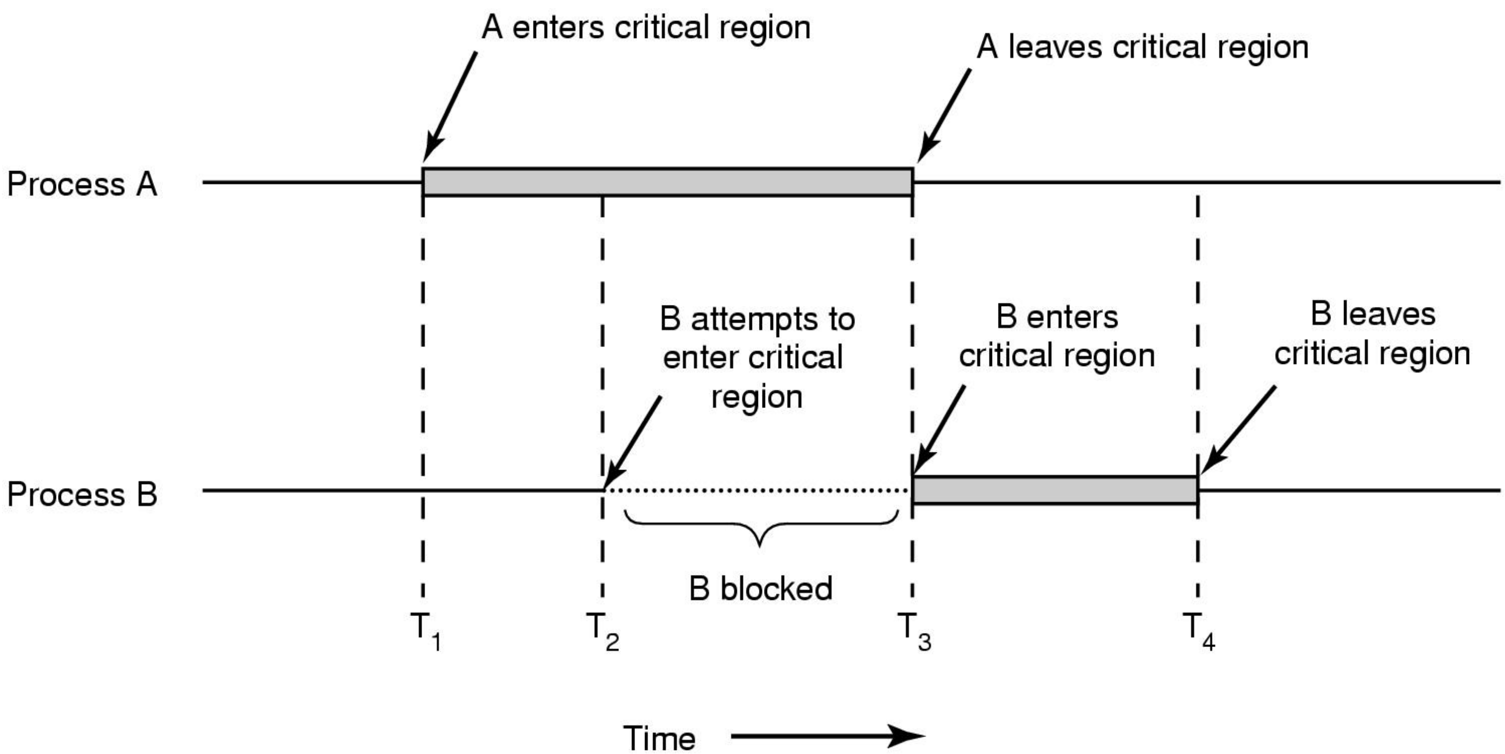
Mutual Exclusion and Critical Sections

- A critical section is a piece of code in which a process or thread accesses a common (shared or global) resource.

- Mutual Exclusion algorithms are used to avoid the simultaneous use of a common resource, such as a global variable.

- In the buying milk example, what is the portion that requires mutual exclusion?

Pictorially…



Conditions for a good Mutex solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Mutex: Implementation Possibilities

- Proposals for achieving mutual exclusion:

  - Lock variables
  - Disabling interrupts
  - Strict alternation
  - Peterson's solution
  - The TSL instruction

Simple, user-level lock variables

```
if (!lock) {
  lock = 1;
  {critical section}
  lock = 0;
}
```

Problem?

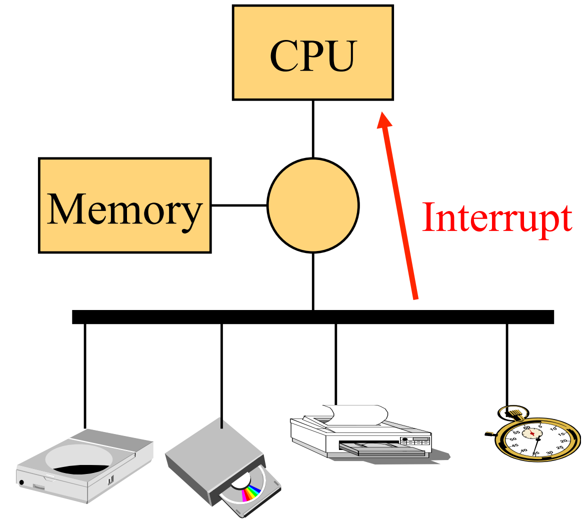
Mutex: Implementation Possibilities

- Proposals for achieving mutual exclusion:

  - Lock variables
  - Disabling interrupts
  - Strict alternation
  - Peterson's solution
  - The TSL instruction

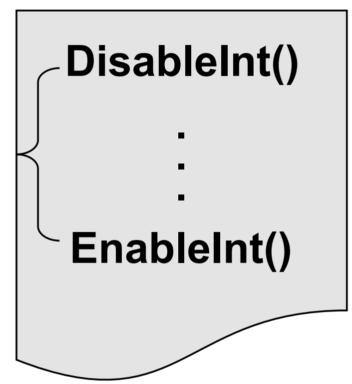
Use and Disable Interrupts

- Use interrupts
  - Implement preemptive CPU scheduling
  - Internal events to relinquish the CPU
  - External events to reschedule the CPU



- Disable interrupts
  - Introduce uninterruptible code regions
  - Think sequentially most of the time
  - **Delay** handling of external events

Uninterruptible region



A Simple Way to Use Disabling Interrupts

Acquire()

critical section?

Release()

```
Acquire() {
     disable interrupts;
}

Release() {
     enable interrupts;
}
```

- Issues with this approach?

One More Try

```
Acquire(lock) {
  disable interrupts;
  while (lock.value != FREE)
       ;
  lock.value = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  lock.value = FREE;
  enable interrupts;
}
```

- Issues with this approach?

Another Try

```
Acquire(lock) {
  disable interrupts;
  while (lock.value != FREE){
    enable interrupts;
    disable interrupts;
    }
  lock.value = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  lock.value = FREE;
  enable interrupts;
}
```

- Does this fix the “wait forever” problem?

Yet Another Try

```
Acquire(lock) {
  disable interrupts;
  while (lock.value == BUSY)
  {
    enqueue me for lock;
    Yield();
  }
  lock.value = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  if (anyone in queue) {
    dequeue a thread;
    make it ready;
  }
  lock.value = FREE;
  enable interrupts;
}
```

- Any issues with this approach?

Mutex: Implementation Possibilities

- Proposals for achieving mutual exclusion:
  - Lock variables
  - Disabling interrupts
  - Strict alternation
  - Peterson's solution
  - The TSL instruction

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Which condition does Strict Alternation violate?:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
      int other;                          /* number of the other process */

      other = 1 – process;                /* the opposite of process */
      interested[process] = TRUE;         /* show that you are interested */
      turn = process;                     /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
      interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Tanenbaum calls this “simpler than Dekker’s”, but still…

Atomic Memory Load orStore

- Assumed in in textbook (e.g. Peterson's solution)

```
int turn;
int interested[N];

void enter_region(int process)
{
    int other;

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE);
}
```

Current machines make promises regarding ordering and atomicity of individual reads or writes at the memory controller. But ordering between unrelated reads and writes is more difficult

- *L. Lamport, "A Fast Mutual Exclusion Algorithm," ACM Trans. on Computer Systems, Feb 1987.*
  - 5 writes and 2 reads

Other Issues: Memory reference ordering between CPUs in a multiprocessor…

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

- CPUs can make promises about memory ordering within one processor core. But harder to make promises across the whole system.
  - => Create special instructions with stronger ordering promises.

One last tragic example……

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

- What is programmer trying to do here?
- What could go wrong?

HARDWARE SUPPORT FOR MUTUAL EXCLUSION

Atomic Read-Modify-Write Instructions

- Basic Abstraction: Test and Set (TAS)
  - Assembly instruction that operates on a memory address
  - TAS memaddress, status
  - Or “TAS Reg7 reg4” where Reg7 contains a memory address, and reg4 is the register where you want the result placed

  - Read memaddress. If contents == 1, that’s it.
  - If contents == 0, atomically set to 1.

- Read and write are performed together in a manner that looks atomic to all processes.
- Return (ie place in a register)
  - If successfully set, return 1 (you just were able to obtain the lock)
  - If not successfully set, return 0 (you were unable to obtain the lock)

Other Atomic Read-Modify-Write Instructions

- LOCK prefix in x86
  - Make a specific set instructions atomic
  - Together with BTS to implement Test&Set
- Exchange (xchg, x86 architecture)
  - Swap register and memory
  - Atomic (even without LOCK)
- Fetch&Add or Fetch&Op
  - Atomic instructions for large shared memory multiprocessor systems
- Load link and conditional store
  - Read value in one instruction (load link)
  - Do some operations;
  - When store, check if value has been modified. If not, ok; otherwise, jump back to start

A Simple Solution with Test&Set

- Define TAS(lock)
  - If successfully set, return 1;
  - Otherwise, return 0;
- Any issues with the following solution?

```
Acquire(lock) {
  while (!TAS(lock.value))
    ;
}

Release(lock) {
  lock.value = 0;
}
```

What About This Solution?

```
Acquire(lock) {
  while (!TAS(lock.guard))
    ;
  if (lock.value) {
    enqueue the thread;
    block and lock.guard = 0;
  } else {
    lock.value = 1;
    lock.guard = 0;
  }
}
```

```
Release(lock) {
  while (!TAS(lock.guard))
    ;
  if (anyone in queue) {
    dequeue a thread;
    make it ready;
  } else
    lock.value = 0;
  lock.guard = 0;
}
```

- How long does the “busy wait” take?

Example: Protect a Shared Variable

```
Acquire(lock)
count++;
Release(lock)
```

- Acquire(mutex) system call
  - Pushing parameter, sys call # onto stack
  - Generating trap/interrupt to enter kernel
  - Jump to appropriate function in kernel
  - Verify process passed in valid pointer to mutex
  - Minimal spinning
  - Block and unblock process if needed
  - Get the lock
- Executing "**count++;**"
- Release(mutex) system call

Available Primitives and Operations

- Test-and-set
  - Works at either user or kernel
- System calls for block/unblock
  - **Block** takes some token and goes to sleep
  - **Unblock** "wakes up" a waiter on token

Block and Unblock System Calls

Block(lock)

- Spin on lock.guard
- Save the context to TCB
- Enqueue TCB to lock.q
- Clear lock.guard
- Call scheduler

Unblock(lock)

- Spin on lock.guard
- Dequeue a TCB from lock.q
- Put TCB in ready queue
- Clear lock.guard

◆ Questions

- Do they work?
- Can we get rid of the spin lock?

Always Block

```
Acquire(lock) {
  while (!TAS(lock.value))
    Block( lock );
}
```

```
Release(lock) {
  lock.value = 0;
  Unblock( lock );
}
```

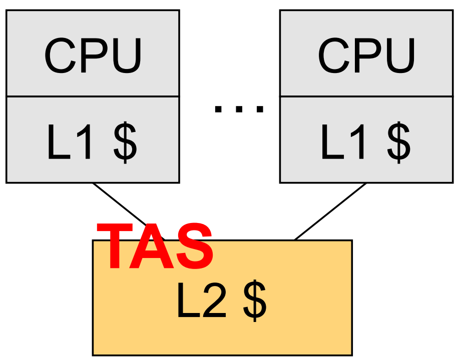
- What are the issues with this approach?

Always Spin

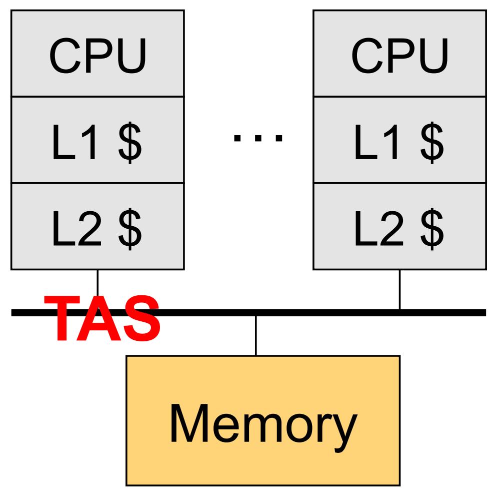
```
Acquire(lock) {
  while (!TAS(lock.value))
    while (lock.value)
    ;
}
```

```
Release(lock) {
  lock.value = 0;
}
```

- Two spinning loops in **`Acquire()`**?



Multicore

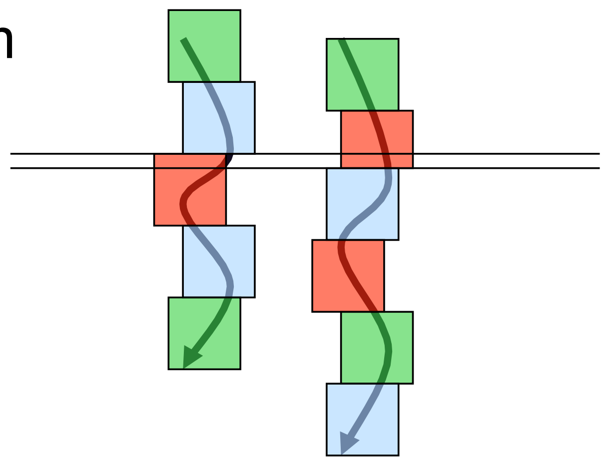


SMP

COMPETITIVE SPINNING

Optimal Algorithms

- What is the optimal solution to spin vs. block?
  - Know the future
  - Exactly when to spin and when to block
- But, we don't know the future
  - There is **no** online optimal algorithm



- Offline optimal algorithm
  - Afterwards, derive exactly when to block or spin ("what if")
  - Useful to compare against online algorithms

Classic Competitive Algorithms Example

- When to rent skis and when to buy?

Competitive Algorithms

- An algorithm is c-competitive if for every input sequence σ

$$C_A(\sigma) \leq c \times C_{opt}(\sigma) + k$$

  - c is a constant
  - $C_A(\sigma)$ is the cost incurred by algorithm A in processing σ
  - $C_{opt}(\sigma)$ is the cost incurred by the optimal algorithm in processing σ

- What we want is to have c as small as possible
  - Deterministic
  - Randomized

Constant Competitive Algorithms

```
Acquire(lock, N) {
  int i;

  while (!TAS(lock.value)) {
    i = N;
    while (!lock.value && i)
      i--;

    if (!i)
      Block(lock);
  }
}
```

- Spin up to N times if the lock is held by another thread
- If the lock is still held after spinning N times, block

- If spinning N times is equal to the context-switch time, what is the competitive factor of the algorithm?

Approximate Optimal Online Algorithms

- Main idea
  - Use past to predict future
- Approach
  - Random walk
    - Decrement N by a unit if the last Acquire() blocked
    - Increment N by a unit if the last Acquire() didn't block
  - Recompute N each time for each Acquire() based on some lock-waiting distribution for each lock
- Theoretical results

  $E\, C_A(\sigma(P)) \leq (e/(e-1)) \times E\, C_{opt}(\sigma(P))$

  The competitive factor is about 1.58.

Empirical Results

	Block	Spin	Fixed C/2	Fixed C	Opt Online	3-samples	R-walk
Nub (2h)	1.943	2.962	1.503	1.559	1.078	1.225	1.093
Taos (24h)	1.715	3.366	1.492	1.757	1.141	1.212	1.213
Taos (M2+)	1.776	3.535	1.483	1.750	1.106	1.177	1.160
Taos (Regsim)	1.578	3.293	1.499	1.748	1.161	1.260	1.268
Ivy (100m)	5.171	2.298	1.341	1.438	1.133	1.212	1.167
Ivy (18h)	7.243	1.562	1.274	1.233	1.109	1.233	1.141
Galaxy	2.897	2.667	1.419	1.740	1.237	1.390	1.693
Hanoi	2.997	2.976	1.418	1.726	1.200	1.366	1.642
Regsim	4.675	1.302	1.423	1.374	1.183	1.393	1.366

Table 1: Synchronization costs for each program relative to the optimal off-line algorithm

	Max spins	Elapsed time (seconds)	Improvement
Always-block	N/A	10529.5	0.0%
Always-spin	N/A	8256.3	21.5%
Fixed-spin	100	9108.0	13.5%
	200	8000.0	24.0%
Opt-known	1008	7881.4	25.1%
Opt-approx	1008	8171.2	22.3%
3-samples	1008	8011.6	23.9%
Random-walk	1008	7929.7	24.7%

Table 3: Elapsed times of Regsim using different spinning strategies.

A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," Proceedings of the 13th ACM Symposium on Operating Systems Principle, 1991.

The Big Picture

	OS codes and concurrent applications			
High-Level Atomic API	Mutex	Semaphores	Monitors	Send/Recv
Low-Level Atomic Ops	Load/store	Interrupt disable/enable	Test&Set	Other atomic instructions
	Interrupts (I/O, timer)	Multiprocessors		CPU scheduling

Summary

- Disabling interrupts for mutex
  - There are many issues
  - When making it work, it works for only uniprocessors
- Atomic instruction support for mutex
  - Atomic load and stores are not good enough
  - Test&set and other instructions are the way to go
- Competitive spinning
  - Spin at the user level most of the time
  - Make no system calls in the absence of contention
  - Have more threads than processors