CIS-505 – Software Systems

Notes for 1/19/12

A quick reminder

- Homework 1 due Wednesday 1/25, 11:59pm
  - Yikes! Less than a week!
  - This is an *individual* assignment (not a group project)
- If you’re going to run experiments, please don’t use the Eniac cluster
  - use the “speclab” cluster for dangerous stuff!

Processes and Threads

- What's a process?
  - "a program in execution"
    - a memory address space (containing code & data)
    - various other resources (e.g. open files)
    - state information (pgm counter, registers, stack pointer etc)
      - the stuff stored in the process' PCB
  - really *two* categories of things
    - a *collection of resources*
      - the code & address space, open files, etc.
    - a *thread of execution*
      - the current state that operates on these resources
- We can think about these two things separately

Threads in the process model

- A lot of what the OS does is intended to keep processes from interfering with each other
  - every thread of execution is associated with its own grouping of resources (process)
  - I can’t write over your process’s address space
- This is nice, but it means that if you want to change threads, you must switch processes
  - requires OS intervention & expensive context switch
  - this is a shame, because some applications logically consist of more than one thread but need only one grouping of resources

Multi-threaded applications

- Could we let multiple threads share a common memory address space?
  - for applications with
    - a need to share data structures among threads
    - no need to for the OS to enforce resource separation (because the threads “trust” each other)
  - *not* for arbitrary code / general programs
- Some potentially *multi-threaded* applications:
  - web *server*
    - serves pages to several different clients at once
  - web *browser*
    - load different pages simultaneously

Can we implement multiple threads in a single process?

- We can do in user mode many of the functions usually handled by the OS
  - assumes the threads are cooperating so we don’t need hardware enforcement of separation
- Basic idea: a “dispatcher” subroutine (in the process) that is called when a thread is ready to relinquish control to another thread
  - manages stack pointer, program counter
  - can switch process’s internal state among threads

Inter-process communication

- The process model is a useful way to *isolate* running programs
  - separate resources, state, etc
  - narrow communication channel (wait, kill, etc)
  - vastly simplifies most programs - no need to worry about what other processes are doing
- Unfortunately, some applications work best if multiple threads are allowed to more tightly communicate and synchronize with each other
  - and this can make things complicated again

When might threads need to communicate?

- Many problems all over operating systems
  - threads with access to same data structures
  - kernel/OS access to user process data
  - processes sharing data via shared memory
  - processes sharing data via system calls
  - processes sharing data via file system
- …and computer science generally
  - database transactions
  - programming languages that support parallelism

What makes this hard?

- Process model allows more than one program to run “at the same time”
  – they’re not *actually* running at the same time (on uniprocessors), of course
  – “at the same time” means an arbitrary -- and unpredictable -- interleaving of the machine instructions of the “simultaneous” processes
- Problem: logical operations on shared data often involve more than one instruction
  – your process might be stopped (by the OS) and another process might run and alter shared data you were in the middle of operating on

Two “simultaneous” processes A (A1-A6) and B (B1-B5)

- instruction A1
- instruction A2
- instruction A3
- instruction A4
- instruction A5
- instruction A6

- instruction B1
- instruction B2
- instruction B3
- instruction B4
- instruction B5

…could be scheduled like this:

- instruction A1
- instruction A2
- instruction A3
- instruction A4
- instruction A5
- instruction A6

- instruction B1
- instruction B2
- instruction B3
- instruction B4
- instruction B5

… or maybe like this:

- instruction A1
- instruction A2
- instruction A3
- instruction A4
- instruction A5
- instruction A6

- instruction B1
- instruction B2
- instruction B3
- instruction B4
- instruction B5

Normally, this wouldn't bother us

- The two threads are isolated, after all
  - different memory, registers, etc
  - can't distinguish between different scheduling sequences, so no problem
- But what if the two threads share access to the same memory?
  - this is where the trouble begins…

Race conditions and Synchronization

- Operations on shared data structures often consist of short “bursts” of instructions
- When two processes/threads are executing concurrently, the result can depend on the precise interleaving of the two instruction streams (this is called a **race condition**)
  - race conditions cause bugs that are hard to reproduce!
- Besides race conditions, another issue is **synchronization** (one process is waiting for the results computed by another)
  - can we avoid busy waiting?

Say you want to count the total number of squirrels and birds…

- Matt: squirrel enthusiast
  - He gets excited (and increments a counter) whenever he sees a squirrel.
  - He ignores birds.
- Jonathan: bird enthusiast
  - He gets excited (and increments a counter) whenever she sees a bird.
  - He ignores squirrels

Matt & Jon have threads that share the same memory

- Two threads in loop incrementing a counter
  - Matt increments *event* when a squirrel arrives

```
while (TRUE) {
        wait_for_squirrel();
        event = event + 1;
}
```

  - Jonathan increments *event* when a bird arrives

```
while (TRUE) {
        wait_for_bird();
        event = event + 1;
}
```

Compiling “event=event+1”

Squirrel-watcher thread:

1. move event, R0
2. increment R0
3. move R0, event

- R0 is used as a temporary register for “event”

Bird-watcher thread:

1. move event, R1
2. increment R1
3. move R1, event

- notice how we were careful to avoid using the same register
- But does it work?

Does this work if squirrels and birds arrive at the same time?

Squirrel-watcher thread:

1. move event, R0
2. increment R0
3. move R0, event

Bird-watcher thread:

1. move event, R1
2. increment R1
3. move R1, event

Reasonable questions to ask at this point…

- At the end of this interleaving, by how much does *event* get incremented?
- Are there other “incorrect” interleavings of these instructions?
- Are there *any* “correct” interleavings?
  – if so, how can we ensure that they occur?

An alternative interleaving:

Squirrel-watcher thread:

1. move event, R0
2. increment R0
3. move R0, event

Bird-watcher thread:

1. move event, R1
2. increment R1
3. move R1, event

What's going on here?

- Things work as expected if these three instructions are executed *together:*
  1. move event, R1
  2. increment R1
  3. move R1, event
- Other threads that operate on *event* shouldn't interrupt during this operation
- We'd like a way to "group" these three instructions together, so that other relevant threads/processes won't interfere with them
- This is called the *mutual exclusion problem*

In other words…

1. BEGIN CRITICAL SECTION
   - (no one should interrupt)
2. move event, R1
3. increment R1
4. move R1, event
5. END CRITICAL SECTION
   - (OK, interrupt again)

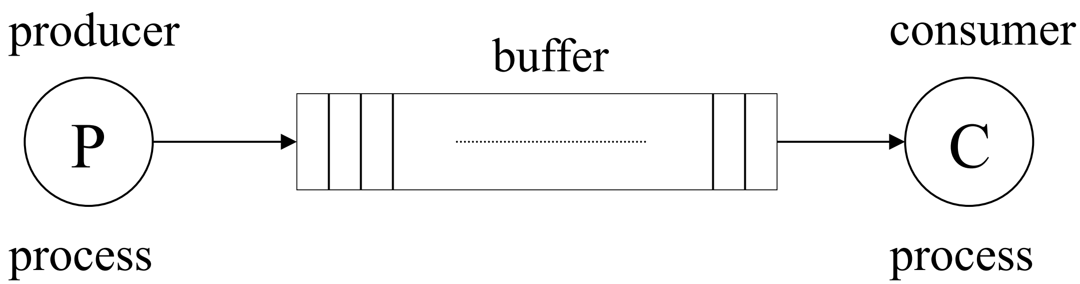
Mutual exclusion

- Need to ensure “atomic” execution of a sequence of instructions
  – at least as far as all the other threads accessing the data are concerned
- How might we do this?
  – an interrupt might occur at any time, giving control to another thread

Possible ways to implement

- Turn off interrupts
  - no help from OS or CPU
  - kind of drastic
- “TEST AND SET” instruction
  - “spin lock”
  - help from CPU
  - can be expensive
- Block until you can have exclusive access
  - Some kind of system call
  - help from OS

Classic example: Producer/Consumer Problems



- from time to time, the producer places an item in the buffer
- the consumer removes an item from the buffer
- careful synchronization required (they run simultaneously)
- the consumer must wait if the buffer empty
- the producer must wait if the buffer full
- typical solution would involve a shared variable count
- also known as the Bounded Buffer problem
- Example: in UNIX shell

cat myfile.txt | lpr

Hierarchy of Abstractions

Idealized Problems
- Producer-Consumer
- Dining Philosophers
- Readers-Writers

High-level Synchronization Primitives
- Monitors (Hoare, Brinch-Hansen)
- Synchronized method in Java

OS-level support (mutual exclusion and synchronization)
- Special variables: Semaphores, Mutexes
- Message passing primitives (send and receive)

Low-level (for mutual exclusion)
- Interrupt disabling
- Using read/write instructions
- Using powerful instructions (Test-and-set, Compare-and Swap…)

What on earth did that mean?

- Computer scientists like to find ways to think about problems at different *layers*
  - this is actually useful sometimes (not just as employment program for CS PhDs)
- Toward the bottom there are low-level *mechanisms*
  - use these to build a solution
- At the top there are high-level *problems*
  - you can try to adapt a solution to an *existing* standard problem to work for *your* problem
  - just find a way to “reduce” your problem to the standard problem

Dealing with Synchronization & Mutual Exclusion

- From the top: find similarities between problem at hand and high-level “ideal” problems with “standard” solutions
  - dining philosophers, producer/consumer, etc
- From the bottom: toolkit of low-level “primitives” that address various aspects
  - test and set instructions, disable interrupts, etc.
- In the middle: *abstractions* that link the two
  - monitors, semaphores, message-passing
    - OS and language interfaces to lower-level tools

The *Mutual Exclusion* Problem

- Problem: Allow access to shared data structures without *race conditions* caused by instruction interleaving
- Abstract Solution:
  - Identify *critical sections* (aka *critical regions*)
    - e.g., instruction sequence for "event = event + 1"
  - Give critical sections exclusive access (without "interference") over their entire execution
- Several different "standard" ways to implement critical section abstraction

Critical Section Abstraction

- Like a “guard” protecting regions of code:
  - BEGIN CRITICAL SECTION
    - if no one else in critical section, you can go in
    - otherwise wait
  - do critical section stuff
  - END CRITICAL SECTION
    - relinquish exclusive access, let someone else enter
- Assumption: *everyone else* is also playing by the same rules – all processes include explicit code around their critical sections

Requirements for a good critical section solution

- Safety
  - should really work - no two threads should ever simultaneously be in a critical section
- Generality
  - shouldn't depend on "fragile" assumptions
- Deadlock Freedom
  - no state where *everyone* is waiting for someone *else* to do something before *anyone* can proceed
- Starvation Freedom ("bounded liveness")
  - if a thread is waiting for access to a critical section, it should eventually be granted

Approach #1: Turning off interrupts

- Works by preventing pre-emptive scheduling via a clock or other interrupt
  - turn interrupts off at beginning, on again at end
- Used in low-level parts of the OS (e.g., during interrupt handling)
- Meets requirements, but not great for user processes
  - requires care: failure to yield may require reboot
  - overly powerful: prevents *all* other process from preempting, not just those entering critical sections
  - inflexible: all-or-nothing

Approach #2a: Use shared variables

- GL Peterson’s solution (1982)
- Everyone needing access to critical section shares special variables used to “flag” access
- Assumes that simple assignments to, and tests on, the shared variables are *atomic*
  - e.g., V doesn’t change *during* “V=1”
    - V might change right before or after, however
  - this is a pretty safe assumption on most uniprocessors

Peterson's solution

- Three shared Boolean variables
  - turn, flag[2] --- (supports 2 threads)
- Code to **enter** critical section:

```
flag[ME] = TRUE
turn = ME
while (turn==ME) && (flag[1-ME]==TRUE)
    ;   /* busy wait */
```

- Code to **exit** critical section:

```
flag[ME] = FALSE
```

Is it really that hard? What if we just do this:

```
while (turn != ME)
   ;  /* busy waiting */
CriticalSectionHere();
turn = 1-ME; /* be fair to other */
```

Ensures mutual exclusion, but requires *strict alternation*.
A process can't ever enter its CS twice in succession
if the other process doesn't enter CS!

How about this?

```
while (flag[1-ME])
  ; /* wait if other guy in CS */
flag[ME] = TRUE;  /* declare your entry */
CriticalSectionHere();
flag[ME] = FALSE; /* unblock other guy */
Non_CS();
```

Safety requirement violated (race condition)!

P0 tests flag[1] and finds it False

P1 tests flag[0] and finds it False

Both proceed, set their flags to True, and enter CS

Or this???

```
flag[ME] = TRUE;  /* declare entry first */
while (flag[1-ME])
  ;  /* wait while other is guy in CS */
CriticalSectionHere();
flag[ME] = FALSE; /* release */
Non_CS();
```

Vulnerable to deadlock (not deadlock-free)!

P0 sets flag[0] to TRUE
P1 sets flag[1] to TRUE
Both enter their loops and keep waiting

Sorry!

All that complexity in Peterson’s solution is actually necessary

Peterson's Solution
(starting to look good...)

```
flag[ME] = TRUE;/* declare interest */
turn = ME; /* for race condition */
while ((flag[1-ME]==TRUE)
          && (turn == ME))
  ; /* busy wait */
CriticalSectionHere();
flag[ME] = FALSE; /* release */
Non_CS();
```

other guy(1-ME)
is contending

detect and deal with
race condition

Approach 2b: Use the hardware (TSL)

- Peterson's solution is nice, but complex
- Much simpler when we have CPU support
  - reading and writing in one atomic operation
  - and fortunately, most modern CPUs have something like this
- Test-and-set-lock: TSL R, memory
  - R gets content of memory
  - memory gets value "1"
- Other powerful atomic instructions also work
  - swap, compare-and-swap, load-linked

Mutual Exclusion with the TSL instruction

- Enter critical section:
  1. TSL R0, lock
  2. if R0 == 1 jump to "1"
- Leave critical section:
  1. lock = 0;
- Does this work? Why (or not)?

So far:

- Solution 1: Disable interrupts
  - big hammer
- Solution 2: busy wait with shared variables
  - 2a: Peterson’s solution
    - complex but no hardware support
    - works on virtually all uniprocessors
  - 2b: TSL
    - simpler but assumes TSL (or equiv) instruction
    - works even on multiprocessors
  - Can be generalized to > 2 threads

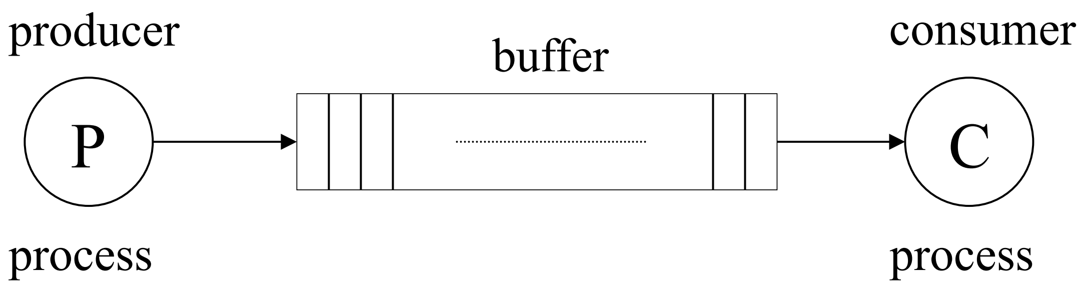
Are we done yet?

- Not quite!
- The shared variable solutions require *busy waiting*
  - bad: this can be inefficient - the locked out thread still needs to consume CPU while waiting
  - worse: can deadlock if the waiting process has higher priority in the scheduler
- We'd usually prefer a *blocking* solution

Approach 3: Sleep and Wakeup

- Two *abstract* system calls (these are not the names of actual Unix system calls!):
  - **sleep**
    - blocks until someone calls **wakeup**
  - **wakeup**(*process*)
    - unblocks *process*
    - process should have been previously blocked with **sleep**
- Remember the producer-consumer problem?
  - sleep and wakeup can help solve the full / empty buffer problem

Producer/Consumer Problems



- from time to time, the producer places an item in the buffer
- the consumer removes an item from the buffer
- careful synchronization required (they run simultaneously)
- the consumer must wait if the buffer empty
- the producer must wait if the buffer full
- typical solution would involve a shared variable count
- also known as the Bounded Buffer problem
- Example: in UNIX shell

cat myfile.txt | lpr

Producer/Consumer (partial solution)

- Producer:

```
while (TRUE) {
  produce X
  if (count == N)
          sleep;
  add_to_buffer(X)
  count = count + 1;
  if (count == 1)
     wakeup(Consumer);
}
```

- Consumer:

```
while (TRUE) {
  if (count==0)
     sleep;
  remove_from_buffer(X)
  count = count -1;
  if (count == N-1)
      wakeup(Producer);
  consume X
}
```

Doesn’t quite work

- Count is initially 0
  - Consumer reads the count
- Producer produces the item, inserts it, and increments count (to 1)
- Producer executes **wakeup**, but there is no waiting consumer (at this point)
- Now consumer continues its execution and calls **sleep***;* consumer blocks
- Consumer stays blocked forever
  - Main problem: race condition -- *wakeup* was lost

Semaphores (Dijkstra)

- A semaphore **S** has a non-negative integer value
- Two operations
  - **up(S)** (aka V(S)) : increments the value of S
  - **down(S)** (aka P(S)) : decrements the value of S if S is positive, else makes the calling thread/process wait
- When S==0, down(S) moves the thread to sleep (blocked) state
  - no busy waiting
- If S==0, up(S) also wakes up one sleeping process (if there are any)
  - uses an internal list of sleeping processes
- up and down calls are *atomic* actions

Can we do mutual exclusion via Semaphores?

- Semaphore value S initially set to 1
- Enter critical section
  - down(S)
- Exit critical section
  - up(S)
- Does this work?

Neat tricks with Semaphores

- Simplest semaphore examples are binary (semaphore value S is either 1 or 0),
  - used here as an “improved” (blocking) TSL
  - to get strict mutual exclusion, set initial S=1
    - maximum number in critical section = 1
- Initial values of S > 1 can be used to allow an arbitrary maximum number of threads to be active somewhere
  - other applications besides mutual exclusion (e.g., resource load control)

Mutual Exclusion Toolkit

- Solution 1: Disable interrupts
  - usually too big a hammer
- Solution 2: busy wait with shared variables
  - exploits standard instruction set features
  - 2a: Peterson's solution
    - complex but no hardware support needed
    - works on virtually all uniprocessors
  - 2b: TSL
    - simpler but assumes TSL (or equiv) instruction
    - works even on multiprocessors
- Solution 3: blocking system calls
  - needs OS support
  - 3a sleep/wakeup
    - simple; useful building block
    - "lost wakeup problem"
  - 3b Semaphores
    - generalized abstraction
    - can solve lost wakeup problem
  - can be built atop interrupt disabling and busy waiting inside OS