Outline

• Announcements
  – No class on February 16th (President’s Day)
  – Lab 2 due on February 23rd
    • Demos will be on 23rd (Monday) and 24th (Tuesday)
  – Questions?

• Process Cooperation
  – Locks, Semaphores, Condition variables
  – Implementing the primitives
  – Classical synchronization problems

[Silberschatz/Galvin/Gagne, Sections 7.2 – 7.5]

(Review) Petersen’s 2-Process Algorithm

• Combines ideas from the turn counter and flag-based solutions

1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false

• Algorithm satisfies
  – Mutual Exclusion: Only one process at a time in critical section
  – Bounded Waiting: No process waits for an indefinite amount of time
  – Progress: At least one process enters critical section when both are trying

Can Such Solutions be Extended to >2 Processes?

• N-process solutions
  – do exist: Bakery Algorithm (see Section 7.2.2)
  – but reasoning gets even more complicated!

• So, we can implement critical sections using only support for atomic memory loads and stores
• But, there must be an easier way!

• Higher-level synchronization primitives
  – locks (mutexes), semaphores, condition variables
  – rely on more support from hardware
    • disabling of interrupts: only around the primitives
    • atomic read-modify-write operations
Synchronization Primitives (1): Locks (Mutexes)

- Locks
  - a single boolean variable L
  - in one of two states: AVAILABLE, BUSY
  - accessed via two atomic operations
    - LOCK (also known as Acquire)
      
      while ( L != AVAILABLE ) wait-a-bit
      L = BUSY;
    - UNLOCK (also known as Release)
      
      L = AVAILABLE:
      wake up a waiting process (if any)
  - process(es) waiting on a LOCK cannot “lock-out” process doing UNLOCK

- Critical sections using locks
  
  LOCK( L )
  CRITICAL SECTION
  UNLOCK( L )
  
  – Mutual exclusion? Progress? Bounded waiting?

Synchronization Primitives (2): Semaphores

- Semaphores
  - a single integer variable S
  - accessed via two atomic operations
    - WAIT (sometimes denoted by P)
      
      while S <= 0 do wait-a-bit;
      S := S-1;
    - SIGNAL (sometimes denoted by V)
      
      S := S+1;
      wake up a waiting process (if any)
  - WAITing process(es) cannot “lock out” a SIGNALing process

- Binary semaphores
  - S is restricted to take on only the values 0 and 1
  - WAIT and SIGNAL become similar to LOCK and UNLOCK
  - are universal in that counting semaphores can be built out of them

Uses of Semaphores

- Mutual exclusion (initially S = 1)
  
  P( S )
  CRITICAL SECTION
  V( S )

- Sequencing (initially S = 0)
  
  P1          P2
  Statement 1
  V( S )      P( S )
  Statement 2

- Detailed examples of its use in Lecture 8

Universality of Binary Semaphores

- Implement operations on a (counting) semaphore CountSem
  - use binary semaphores S1 = 1, S2 = 0
  - integer C = initial value of counting semaphore
    
    P(CountSem) V(CountSem)
    
    P(S1);         P(S1);
    C := C-1;      C := C+1;
    if ( C < 0 ) then
      begin V(S1); P(S2); end
      if ( C <= 0 ) then
        V(S1);         V(S1);
        else

    – S1 ensures mutual exclusion for accessing C
    – S2 is used to block processes when C < 0
    – is a race condition possible after V(S1) but before P(S2)?
Synchronization Primitives (3): Condition Variables

- Condition variables
  - an implicit process queue
  - three operations that must be performed within a critical section
    - WAIT
      - associate self with the implicit queue
      - suspend self
    - SIGNAL
      - wake up exactly one suspended process on queue
        - has no effect if there are no suspended processes
    - BROADCAST
      - wake up all suspended processes on queue

- Two types based on what happens to the process doing the SIGNAL
  - Mesa style (Nachos uses Mesa-style condition variables)
    - SIGNAL-ing process continues in the critical section
    - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - SIGNAL-ing process immediately exits the critical section
    - resumed process now occupies the critical section

Uses of Condition Variables

- Can be used for constructing
  - critical sections, sequencing, …

- Primary use is for waiting on an event to happen
  - after checking that it has not already happened
    - WHY IS THIS IMPORTANT?

- Example: Three processes that need to cycle among themselves
  - Example: Three processes that need to cycle among themselves
  - <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: turn; three condition variables: cv0, cv1, cv2
  - Process P executes (in a critical section)
    - while ( turn != i) WAIT(cv_i)
    - <do the operation>
    - turn := (turn + 1) mod 3; SIGNAL(cv_turn)

Higher-level Synchronization Primitives

- Several additional primitives are possible
  - Built using locks, semaphores, and condition variables

- An example: Event Barriers (see Nachos Lab 3)

Implementing the Synchronization Primitives

- Need support for atomic operations from the underlying hardware
  - applicable only to a small number of instructions
    - else, can implement critical sections this way

Three choices
- Use n-process mutual-exclusion solutions
  - complicated

✓ Selectively disable interrupts on uniprocessors
  - so, no unanticipated context switches ◆ atomic execution
  - solution adopted in Nachos (see Lab 2 for details)

✓ Rely on special hardware synchronization instructions

- Can implement one primitive in terms of another
  - Nachos Lab 2
Implementation Choices (1): Interrupt Disabling

- Semaphores

\[ \text{P(S)} \]
\[ \text{DISABLE-INTERRUPTS} \]
\[ \text{while } S \leq 0 \text{ do wait-a-bit } \text{<ENABLE-INTERRUPTS; YIELD CPU}> \]
\[ S := S-1; \]
\[ \text{ENABLE-INTERRUPTS} \]

\[ \text{V(S)} \]
\[ \text{DISABLE-INTERRUPTS} \]
\[ S := S+1; \]
\[ \text{[ wake up a waiting process ]} \]
\[ \text{ENABLE-INTERRUPTS} \]

- Drawback
  - a process spins on this loop (busy waiting) till it can enter critical section
  - can waste substantial amount of CPU cycles idling
    - Even if wait-a-bit is implemented as
      - give up CPU (i.e. put at the end of ready queue)
    - since there are still context switches
  - not a very useful utilization of valuable cycles

Efficient Semaphores

- Implement P and V differently
  - maintain an explicit wait queue organized as a scheduler structure

\[ \text{type semaphore = record} \]
\[ \text{value: integer;} \]
\[ \text{L: list of processes;} \]
\[ \text{end}; \]

\[ \text{P(S): } S.\text{value} := S.\text{value} - 1; \]
\[ \text{if } S.\text{value} < 0 \text{ then begin} \]
\[ \text{add process to } S.\text{L block;} \]
\[ \text{end; } \]

\[ \text{V(S): } S.\text{value} := S.\text{value} + 1; \]
\[ \text{if } S.\text{value} \leq 0 \text{ then begin} \]
\[ \text{remove } P \text{ from } S.\text{L \text{wakeup}();} \]
\[ \text{end; } \]

- still need atomicity: can use previously discussed solutions
  - can have spinning but only for a small period of time (~10 instructions)
  - queue enqueue/dequeue must be fair
    - not required by semantics of semaphores

Implementation Choices (2): Hardware Support

- Rationale: Hardware instructions enable simpler/efficient solutions to common synchronization problems
  - disabling interrupts is a brute-force approach
  - does not work on multiprocessors
    - simultaneous disabling of all interrupts is not feasible

- Two common primitives
  - test-and-set
  - swap

Semantics of Hardware Primitives

- Test-and-set
  - given boolean variables X, Y, atomically set X := Y; Y := true

\[ \text{boolean Test-and-set( boolean &target ) } \]
\[ \{ \]
\[ \text{boolean rv = target;} \]
\[ \text{target = true;} \]
\[ \text{return rv;} \]
\[ \} \]

- Swap
  - atomically exchange the values of given variables X and Y

\[ \text{temp = X; X = Y; Y = temp;} \]
  - can emulate test-and-set

\[ \text{boolean Test-and-set( boolean &target ) } \]
\[ \{ \]
\[ \text{boolean t := true;} \]
\[ \text{swap (target, t);} \]
\[ \text{return t;} \]
\[ \} \]
Implementing Locks Using Test-and-Set

```plaintext
LOCK:

L : boolean := false
while Test-and-set(lock) wait-a-bit

UNLOCK

lock := false
```

• Properties of this implementation
  – Mutual exclusion?
    • first process \( P_i \) entering critical section sets \( lock := \text{true} \)
    • test-and-set (from other processes) evaluates to true after this
    • when \( P_i \) exits, lock is set to false, so the next process \( P_j \) to execute the
      instruction will find test-and-set = false and will enter the critical section
  – Progress?
    • trivially true
  – Unbounded waiting
    • possible since depending on the timing of evaluating the test-and-set primitive,
      other processes can enter the critical section first
    • See Section 7.3 for a solution to this problem

Synchronization Primitives in Real OSes

• Unix: Single CPU OS
  – implement critical sections using interrupt elevation
    • disallow interrupts that can modify the same data
  – another possibility: interrupts never “force” a context switch
    • they just set flags, or wake up processes
  – primitives
    • sleep (address);
    • wake_up (address); -- wakes up all processes sleeping on address
  – typical code
    ```plaintext
    ENTRY: while (locked) sleep (bufaddr);
    locked = true;
    EXIT: locked = false; wake_up (bufaddr);
    ```

Synchronization Primitives in Real OSes (contd.)

• Solaris 2: multi-CPU OS
  – for brief accesses only
    • adaptive mutexes
    • starts off as a standard spinlock semaphore
      • if lock is held by running thread, continues to spin
      • valid only on a multi-CPU system
      • otherwise blocks
  – for long-held locks
    • condition variables
      • wait and signal
    • reader-writer locks
      • for frequent mostly read-only accesses

Classical Synchronization Problems

• Commonly encountered problems in operating systems
  – used to test any proposal for a new synchronization primitive

1. Mutual exclusion
  – only one process executes a piece of code (critical section) at any time
  – OS examples: access to shared resources
    • e.g., a printer

2. Sequencing
  – a process waits for another process to finish executing some code
  – OS examples: waiting for an event
    • e.g., recv suspends until there is some data to read on the network
Classical Synchronization Problems (cont’d)

3. **Bounded-buffer** (also referred to as the **Producer-Consumer** problem)
   - a pool of \( n \) buffers
   - *producer* process(es) put items into the pool
   - *consumer* process(es) take items out of the pool
   - issues: mutual exclusion, empty pool, and full pool
   - OS examples: buffering for pipes, file caches, etc.

4. **Readers-Writers**
   - multiple processes access a shared data object \( X \)
     - any number of *readers* can access \( X \) at the same time
     - no *writer* can access it at the same time as a *reader* or another *writer*
   - mutual exclusion is too constraining: WHY?
   - variations:
     - *reader-priority*: a reader must not wait for a writer
     - *writer-priority*: a writer must not wait for a reader
   - OS examples: file locks

Classical Synchronization Problems (cont’d)

5. **Dining Philosophers**
   - 5 philosophers
   - 5 chopsticks placed between them
     - to eat requires two chopsticks
   - philosophers alternate between thinking and eating
   - issues: deadlock, starvation, fairness
   - OS examples: simultaneous use of multiple resources
     - e.g., disk bandwidth and storage