Outline

- Announcements
  - No class on Monday, Feb 17th
  - Lab 2 due Feb 24th
    - Defer extra credit part (priority scheduler) to Lab 3

- Process synchronization
  - Critical sections (cont’d)
    - Petersen’s 2-process solution
  - Locks, Semaphores, Condition variables
  - Implementing the primitives
  - Classical synchronization problems

{ Silberschatz/Galvin/Gagne: Sections 7.2 – 7.5}
Petersen’s Algorithm: Mutual Exclusion

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

- Suppose: P₀ is in its critical section, and P₁ is wanting to enter
- This can happen only if either
  - (case 1) P₀ found flag[1] false, or
  - (case 2) P₀ found turn == 0
    - in the first case: P₁ will set turn after P₀ did, and find turn == 0
    - in the second case: P₁ has already set turn = 0
    - in both cases: P₁ will wait till flag[0] == false

Petersen’s Algorithm: Progress and Bounded Waiting

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

- To prove progress:
  - if P₁ is not ready to enter the critical section
    - flag[1] will be false ● P₀ can enter
- To prove bounded waiting:
  - let P₀ be in the critical section and P₁ be waiting on instruction 3 above
  - if P₀ exits and goes elsewhere,
    - either P₁ will find flag[0] to be false
    - if not, P₀ will attempt to reenter the critical section, setting turn := j
    - in either case, P₁ will find the condition for waiting in (3) to be false and will enter the critical section

Can These 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 $P$)
      
      \[
      \text{while } S <= 0 \text{ do } \text{wait-a-bit}; \\
      S := S - 1;
      \]
    - SIGNAL (sometimes denoted by $V$)
      
      \[
      S := S + 1; \\
      \text{wake up a waiting process (if any)}
      \]
  - 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 $S = 1$)
  
  \[
  \text{P} (S) \text{ CRITICAL SECTION } V(S) 
  \]

- Sequencing (initially $S = 0$)
  
  \[
  \text{P}_1 \quad \text{P}_2 \\
  \text{Statement 1 } V(S) \text{ P} (S) \text{ Statement 2}
  \]

- Detailed examples of its use in Lecture 7

Universality of Binary Semaphores

- Implement operations on a (counting) semaphore $\text{CountSem}$
  - use binary semaphores $S_1 = 1, S_2 = 0$
  - integer $C = \text{initial value of counting semaphore}$

  \[
  \begin{align*}
  \text{P(CountSem)} & \quad \text{V(CountSem)} \\
  \text{P}(S_1); & \quad \text{P}(S_1); \\
  C := C - 1; & \quad C := C + 1; \\
  \text{if } (C < 0) \text{ then } & \quad \text{if } (C <= 0) \text{ then } \text{V}(S_2); \\
  \text{begin } \text{V}(S_1); \text{P}(S_2); \text{ end } & \quad \text{else } \text{V}(S_1); \text{V}(S_1); \\
  \end{align*}
  \]

  - $S_1$ ensures mutual exclusion for accessing $C$
  - $S_2$ is used to block processes when $C < 0$
  - is a race condition possible after $V(S_1)$ but before $P(S_2)$?

Synchronization Primitives (3): Condition Variables

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

- 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
  <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: turn; three condition variables: cv0, cv1, cv2
  - Process Pi executes (in a critical section)

    ```
    if (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 2)

Outline

- Announcements
  - No class on Monday, Feb 17th
  - Lab 2 due Feb 24th
    • Defer extra credit part (priority scheduler) to Lab 3

- Process synchronization
  - Critical sections (cont’d)
    • Petersen’s 2-process solution
  - Locks, Semaphores, Condition variables
  - Implementing the primitives
  - Classical synchronization problems

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

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

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

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

- 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 \textit{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 \textit{wait queue} organized as a scheduler structure

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

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

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

- 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

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

- Swap
  - atomically exchange the values of given variables X and Y
  - can emulate test-and-set

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

**LOCK:**

\[ L : \text{boolean} := \text{false} \]

\[ \text{while Test-and-set}(\text{lock}) \text{ wait-a-bit} \]

**UNLOCK**

\[ \text{lock} := \text{false} \]

- Properties of this implementation
  - Mutual exclusion?
    - first process \( P_i \) entering critical section sets \( \text{lock} := \text{true} \)
    - 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
    - \text{sleep} (address);
    - \text{wake_up} (address); -- wakes up all processes sleeping on address
  - typical code
    
    ENTRY: while (locked) sleep(bufaddr);
    locked = true;
    EXIT: locked = false; wake_up (bufaddr);

Outline

- Announcements
  - No class on Monday, Feb 17th
  - Lab 2 due Feb 24th
    - Defer extra credit part (priority scheduler) to Lab 3
- Process synchronization
  - Critical sections (cont’d)
    - Petersen’s 2-process solution
  - Locks, Semaphores, Condition variables
  - Implementing the primitives
  - Classical synchronization problems

{ Silberschatz/Galvin/Gagne: Sections 7.2 – 7.5}
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

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