Outline

- Announcements
  - Lab 1 due today. Demos scheduled for tomorrow (Feb 03, 2000)
  - Lab 2 out today, due Feb 16, 2000
  - Questions?

- Process synchronization primitives
  - critical sections (review)
  - two-process solutions for mutual exclusion
  - higher-level primitives: locks, semaphores, condition variables
  - implementing the primitives

[ Silberschatz/Galvin: Sections 6.1-6.4]

(Review) Critical Sections

- Critical sections: General framework for process synchronization
  
  ENTRY-SECTION
  CRITICAL-SECTION-CODE
  EXIT-SECTION

  - the ENTRY-SECTION controls access to make sure no more than one process $P_i$ gets to access the critical section at any given time
    * acts as a guard
  - the EXIT-SECTION does bookkeeping to make sure that other processes that are waiting know that $P_i$ has exited

- How can we implement critical sections?
  - build on top of atomic memory load/store operations
  - provide higher-level primitives

(Review) Two-Process Solutions: Turn Counters

- Shared integer variable: $\text{turn}$ (initialized to 0)
  - for $i \in \{0, 1\}$: $P_i$ executes:
    
    while ($\text{turn} \neq i$) wait-a-bit;
    CRITICAL SECTION;
    $\text{turn} := j$;

  - the while loop is the entry section
    * process $P_i$ waits till its turn occurs
  - the single instruction $\text{turn} := j$ constitutes the exit section
    * informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores

- Drawbacks?
  - if $P_i$ never wants to execute the critical section, $P_0$ cannot reenter;
    * access must alternate
Two-Process Solutions: Array of Flags

- Boolean array flag (initialized to false), $P_i$ executes:
  1: $\text{flag}[i] := \text{true};$
  2: while $\text{flag}[j]$ wait-a-bit;
  3: $\text{flag}[i] := \text{false};$

- Mutual exclusion?

  - Is this good enough?
  - No: $P_0$ and $P_1$ can be looping on instruction 2 forever

Criteria for Correctness

- Three conditions
  - Mutual exclusion
  - Progress
    - at least one process requesting entry to a critical section will be able to enter it if there is no other process in it
  - Bounded waiting
    - no process waits indefinitely to enter the critical section once it has requested entry

Two-Process Solutions: Petersen’s Algorithm

- Combines the previous two ideas
  1: $\text{flag}[i] := \text{true}$
  2: $\text{turn} := j$
  3: while ($\text{flag}[j]$ and ($\text{turn} == j$)) wait-a-bit
  4: $\text{flag}[i] := \text{false}$

- Does the algorithm satisfy the three criteria?

Petersen’s Algorithm: Mutual Exclusion

- Suppose: $P_0$ is in its critical section, and $P_1$ is wanting to enter
  - This can happen only if either
    - (case 1) $P_0$ found $\text{flag}[1]$ false, or
    - (case 2) $P_0$ found $\text{turn} == 0$
    - in the first case: $P_1$ will set $\text{turn}$ after $P_0$ did, and find $\text{turn} == 0$
    - in the second case: $P_1$ has already set $\text{turn} == 0$
    - in both cases: $P_1$ will wait till $\text{flag}[0] == \text{false}$
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 P_i is not ready to enter the critical section
  • flag[i] will be false • P_0 can enter

To prove bounded waiting:
- let P_0 be in the critical section and P_1 be waiting on instruction 3 above
- if P_0 exits and goes elsewhere,
  • either P_1 will find flag[0] to be false
  • if not, P_0 will attempt to reenter the critical section, setting turn := j
  • in either case, P_1 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 6.2.2)
  • 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
    • atomic read-modify-write operations

Synchronization Primitives (1): Locks (Mutexes)

- Locks (also known as Mutexes)
  • 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 $S = 1$)
  
  $$\text{P}(S)$$
  
  CRITICAL SECTION
  
  $$\text{V}(S)$$

- Sequencing (initially $S = 0$)
  
  $$\text{P}_1, \text{P}_2$$

  Statement 1
  
  $$\text{V}(S) \text{ P}(S)$$

  Statement 2

  $$\text{V}(S)$$

- Detailed examples of its use in Lecture 8 (February 14)

Universality of Binary Semaphores

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

  $$\text{P}($$CountSem$$) \quad \text{V}($$CountSem$$)$$

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

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

Synchronization Primitives (3): Condition Variables

- Condition variables
  
  - an implicit process queue

  - three operations that must be performed within a critical section
    
    - $$\text{WAIT}$$
      
      associate self with the implicit queue
      suspend self

    - $$\text{SIGNAL}$$
      
      wake up exactly one suspended process on queue
      – has no effect if there are no suspended processes

    - $$\text{BROADCAST}$$
      
      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)
    
    - $$\text{SIGNAL}$$-ing process continues in the critical section
    
    - resumed process must re-enter (so, is not guaranteed to be the next one)

  - Hoare style
    
    - $$\text{SIGNAL}$$-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: $cv_0, cv_1, cv_2$

  - Process $P_i$ executes (in a critical section)

  $$\begin{align*}
  &\text{if } (\text{ turn } != i) \text{ WAIT}(cv_i) \\
  &\langle\text{do the operation}\rangle \\
  &\text{turn} := (\text{turn} + 1) \text{ mod 3}; \text{ SIGNAL}(cv_{\text{turn}})
  \end{align*}$$
Higher-level Synchronization Primitives

- Several additional primitives are possible
  - Built using locks, semaphores, and condition variables
- An example: Event Barriers (see Nachos Lab 2)

Implementing the Synchronization Primitives

- Need support for atomic operations from the underlying hardware
  - Applicable 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
    - 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

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

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

- Drawback
  - A process spins on this loop till it gets a chance to 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

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

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

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

- 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
  
  ```
  function test-and-set(var target:boolean) boolean;
  begin
  test-and-set := target;
  target := true;
  end;
  ```

- **Swap**
  - atomically exchange the values of given variables X and Y
    
    ```
    temp = X; X = Y; Y = temp;
    ```
  - can emulate test-and-set
    
    ```
    function test-and-set(var v: boolean): boolean
    var t := true;
    swap (v, t);
    return t;
    ```

Implementing Locks Using Test-and-Set

- **LOCK**
  
  ```
  L : boolean := false
  while test-and-set (lock) wait-a-bit
  ```

- **UNLOCK**
  
  ```
  lock := false
  ```

- **Properties of this implementation**
  - mutual exclusion?
    - first process Pi entering critical section sets lock := true
    - test-and-set (from other processes) evaluates to true after this
    - when Pi exits, lock is set to false, so the next process Pj 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 6.3 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, priority); -- wakes up process
    - **wake_up** (address);
  - typical code
    
    ```
    ENTRY: while (locked) sleep(bufaddr);
    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

Next Two Lectures

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
  - multiple-processor scheduling

Reading

- Silberschatz/Galvin: Chapter 5