Outline

- Announcements
  - Lab 1 due today. Demos scheduled for Thursday and Friday
  - Lab 2 out today, due Feb 14, 2000
  - Questions?

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

[ Silberschatz/Galvin: Sections 6.1-6.4]

(Review) Need for Process Synchronization

- **Problem**: Unsynchronized execution can cause **race conditions**
  - Instruction sequences are usually not **atomic** or **uninterruptable**
  - Can be interleaved because of
    - Interrupts
    - Context switches
  - Can produce inconsistencies in shared data

- **Solution**: Processes need to **synchronize**
  - so that they do **not** access shared variables at the same time
  - this is called **mutual exclusion**
    - the **shared and critical** variables can be accessed by only one process at a time
    - access must be **serialized** even if the processes attempt concurrent access

- **General framework for achieving this**: Critical Sections

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?**
  - turn off interrupts around critical operations
  - build on top of atomic memory load/store operations
  - provide higher-level primitives
Two-Process Solutions: Turn Counters

- Shared integer variable: \( \text{turn} \) (initialized to 0)
  - for \( i \in \{0, 1\} \): \( P_i \) executes:
    ```
    while (\( \text{turn} \neq i \)) \text{ wait-a-bit;}
    \text{CRITICAL SECTION;}
    \text{turn} := j;
    ```
  - the while loop is the \textit{entry} section
    - process \( P_i \) waits till its turn occurs
  - the single instruction \( \text{turn} := j \) constitutes the \textit{exit} section
    - informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores

- Drawbacks?
  - if \( P_1 \) never wants to execute the critical section, \( P_0 \) cannot reenter;
    - access must alternate

Mutual exclusion?

Two-Process Solutions: Array of Flags

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

- Mutual exclusion?

- Is this good enough?
  - \textbf{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] := true
  2: \text{turn} := j
  3: \text{while} \text{(flag}[j] \text{ and (turn }== j)) \text{ wait-a-bit}
  \text{CRITICAL SECTION}
  4: \text{flag}[i] := false
  ```

- Does the algorithm satisfy the three criteria?
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 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: 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)
  \[
  \begin{align*}
  &P(S) \\
  &\text{CRITICAL SECTION} \\
  &V(S)
  \end{align*}
  \]

- Sequencing (initially S = 0)
  \[
  \begin{align*}
  &P_1 P_2 \\
  &\text{Statement 1} \\
  &V(S) P(S) \\
  &\text{Statement 2}
  \end{align*}
  \]

- Detailed examples of its use in Lecture 6

Universality of Binary Semaphores

- Implement operations on a (counting) semaphore \texttt{CountSem}
  - use binary semaphores S1 = 1, S2 = 0
  - integer C = initial value of counting semaphore

\[
\begin{align*}
&P \text{(\hspace{1cm}CountSem)} \hspace{1cm} V \text{(\hspace{1cm}CountSem)} \\
&P(S1); \hspace{1cm} P(S1); \\
&C := C - 1; \hspace{1cm} C := C + 1; \\
&\text{if ( C < 0 ) then} \hspace{1cm} \text{if ( C <= 0 ) then} \hspace{1cm} V(S2); \hspace{1cm} V(S1); \\
&\hspace{1cm} \text{begin} \hspace{1cm} \text{end} \hspace{1cm} \text{else} \hspace{1cm} \text{end} \\
\end{align*}
\]

- S1 ensures mutual exclusion for accessing C
- S2 is used to block processes when C < 0
- is a race condition possible after \texttt{V(S1)} but before \texttt{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
  <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: turn; three condition variables: cv0, cv1, cv2
  - Process P_i 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)

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)
    ```
    DISABLE-INTERRUPTS
    while S <= 0 do wait-a-bit <ENABLE-INTERRUPTS; YIELD CPU>
    S := S-1;
    ENABLE-INTERRUPTS
    ```
  - V(S)
    ```
    DISABLE-INTERRUPTS
    S := S+1;
    wake up a waiting process
    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

```
type semaphore = record
  value: integer;
  L: list of processes;
end;
```

```
P(S):  S.value := S.value - 1;
      if ( S.value < 0 ) then begin
        add process to S.L
        block;
      end;
```

```
V(S):  S.value := S.value + 1;
      if ( S.value <= 0 ) then begin
        remove P from S.L
        wakeup(P);
      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;
  return target;
end;
```

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

```
temp = X; X = Y; Y = temp;
```

```
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 P_i entering critical section sets lock := true
    - test-and-set (from other processes) evaluates to true after this
  - 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);
    - `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);
    ```

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

- Classical process synchronization problems
  - Mutual exclusion
  - Sequencing
  - Producer consumer
  - Readers-writers
  - Dining philosophers

- Language support for process synchronization
  - Critical regions
  - Monitors
  - Message passing

Reading

- Silberschatz/Galvin: Chapter 6