Outline

- Announcements
  - Lab 1 demos today, tomorrow
  - Lab 2 due Feb 25th
    - Defer extra credit part (priority scheduler) to Lab 3

- Process synchronization
  - Locks, Semaphores, Condition variables
  - Implementing the primitives
  - Classical synchronization problems
    - Mutual exclusion
    - Sequencing
    - Producer consumer
    - Readers-writers
    - Dining philosophers

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

(Review) Synchronization Primitives (1): Locks

- 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)
  \[ \text{P}(S) \]
  \[ \text{CRITICAL SECTION} \]
  \[ \text{V}(S) \]

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

- Detailed examples of its use later in the lecture

Universality of Binary Semaphores

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

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

- \texttt{S1} ensures mutual exclusion for accessing \( C \)
- \texttt{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
    - \texttt{WAIT}
      - associate self with the implicit queue
      - suspend self
    - \texttt{SIGNAL}
      - wake up exactly one suspended process on queue
      - has no effect if there are no suspended processes
    - \texttt{BROADCAST}
      - wake up all suspended processes on queue

- Two types based on what happens to the process doing the \texttt{SIGNAL}
  - Mesa style (Nachos uses Mesa-style condition variables)
    - \texttt{SIGNAL}-ing process continues in the critical section
    - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - \texttt{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
    - \textbf{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: \texttt{turn}; three condition variables: \texttt{CV}_0, \texttt{CV}_1, \texttt{CV}_2
  - Process \( P_i \) executes (in a critical section)

\[
\text{if } (\text{turn} != i) \text{ WAIT}(\text{CV}_i) \\
<\text{do the operation}> \\
\text{turn} := (\text{turn} + 1) \mod 3; \text{ SIGNAL}(\text{CV}_{\text{turn}})
\]
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
  ```
  P(S):
  DISABLE-INTERRUPTS
  while S <= 0 do wait-a-bit <ENABLE-INTERRUPTS; YIELD CPU>
  S := S-1;
  ENABLE-INTERRUPTS
  end;
  end;

  V(S):
  DISABLE-INTERRUPTS
  S := S+1;
  [ wake up a waiting process ]
  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

```pascal
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;
  end;

V(S):
  S.value := S.value + 1;
  if ( S.value <= 0 )
    then begin
      remove P from S.L
      wakeup(P);
      end;
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
    ```java
    boolean Test-and-set( boolean &target ) {
        boolean rv = target;
        target = true;
        return rv;
    }
    ```
- **Swap**
  - atomically exchange the values of given variables X and Y
    ```java
    temp = X; X = Y; Y = temp;
    ```
  - can emulate test-and-set
    ```java
    boolean Test-and-set( boolean &target ) {
        boolean t := true;
        swap (target, 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
    - 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
    ```
    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

Outline

- Announcements
  - Lab 1 demos today, tomorrow
  - Lab 2 due Feb 25th
    - Defer extra credit part (priority scheduler) to Lab 3

- Process synchronization
  - Locks, Semaphores, Condition variables
  - Implementing the primitives
  - Classical synchronization problems
    - Mutual exclusion
    - Sequencing
    - Producer consumer
    - Readers-writers
    - Dining philosophers

{ 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
Classical Synchronization Problems (contd.)

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

Mutual Exclusion and Sequencing Using Semaphores

- Mutual exclusion: Semaphore initialized to 1
  \[
  \begin{align*}
  &P(S); \\
  &\text{CRITICAL SECTION} \\
  &V(S);
  \end{align*}
  \]

- Sequencing: Semaphore initialized to 0
  \[
  \begin{align*}
  &\text{process 1} \\
  &B();V(S); \\
  &P(S); \\
  &A(); \\
  &\text{process 2} \\
  &\end{align*}
  \]

Bounded-buffer Using Semaphores

- Three semaphores
  - \text{mutex}: provide mutual exclusion between processes (initial value = 1)
  - \text{empty}: count the number of empty slots (initial value = N)
  - \text{full}: count the number of full slots (initial value = 0)

**Producer(s)**

\[
\begin{align*}
\text{repeat} \\
// \text{produce an item in nextp} \\
P(\text{empty}); \\
P(\text{mutex}); \\
\text{// add nextp to buffer} \\
V(\text{mutex}); \\
V(\text{full}); \\
\text{until false;}
\end{align*}
\]

**Consumer(s)**

\[
\begin{align*}
\text{repeat} \\
P(\text{full}); \\
P(\text{mutex}); \\
\text{// remove item into nextc} \\
V(\text{mutex}); \\
V(\text{empty}); \\
\text{// consume item in nextc} \\
\text{until false;}
\end{align*}
\]

Readers-Writers Using Semaphores

To allow multiple readers, synchronize only the first/last reader with writers

**Reader(s)**

\[
\begin{align*}
P(s); \\
\text{rcount} := \text{rcount} + 1; \\
\text{if (rcount} == 1) \text{ then } P(\text{wsem}); \\
V(s); \\
\text{READ} \\
P(s); \\
\text{rcount} := \text{rcount} - 1; \\
\text{if (rcount} == 0) \text{ then } V(\text{wsem}); \\
V(s);
\end{align*}
\]

**Writer(s)**

\[
\begin{align*}
P(\text{wsem}); \\
\text{WRITE} \\
V(\text{wsem}); \\
P(s);
\end{align*}
\]

- stream of readers can starve writers
- can release either waiting readers or writers
Readers-Writers Using Semaphores: Writer-Priority

Have a writer block out subsequent readers (same as readers block out writers)

Reader

\[
\begin{align*}
P(rsem); \\
P(x); \\
rcount := rcount + 1; \\
\text{if } (rcount == 1) \text{ then } P(wsem); \\
V(x); \\
V(rsem); \\
\text{READ} \\
P(x); \\
rcount := rcount - 1; \\
\text{if } (rcount == 0) \text{ then } V(wsem); \\
V(x);
\end{align*}
\]

Writer

\[
\begin{align*}
P(y); \\
wcount := wcount + 1; \\
\text{if } (wcount == 1) \text{ then } P(rsem); \\
V(y); \\
V(wsem); \\
\text{WRITE} \\
P(y); \\
wcount := wcount - 1; \\
\text{if } (wcount == 0) \text{ then } V(rsem); \\
V(y);
\end{align*}
\]

Readers-Writers Using Semaphores: Writer-Priority (2)

Reader

\[
\begin{align*}
P(z); \\
P(rsem); \\
P(x); \\
rcount := rcount + 1; \\
\text{if } (rcount == 1) \text{ then } P(wsem); \\
V(x); \\
V(rsem); \\
V(z); \\
\text{READ} \\
P(x); \\
rcount := rcount - 1; \\
\text{if } (rcount == 0) \text{ then } V(wsem); \\
V(x);
\end{align*}
\]

Writer

\[
\begin{align*}
P(y); \\
wcount := wcount + 1; \\
\text{if } (wcount == 1) \text{ then } P(rsem); \\
V(y); \\
V(wsem); \\
\text{WRITE} \\
P(y); \\
wcount := wcount - 1; \\
\text{if } (wcount == 0) \text{ then } V(rsem); \\
V(y);
\end{align*}
\]

Philosopher (even i)

\[
\begin{align*}
P( \text{chopstick}[i] ); \\
P( \text{chopstick}[i+1 \mod 5] ); \\
\text{EAT} \\
V( \text{chopstick}[i] ); \\
V( \text{chopstick}[i+1 \mod 5] ); \\
\text{THINK}
\end{align*}
\]

Philosopher (odd i)

\[
\begin{align*}
P( \text{chopstick}[i+1 \mod 5] ); \\
P( \text{chopstick}[i] ); \\
\text{EAT} \\
V( \text{chopstick}[i+1 \mod 5] ); \\
V( \text{chopstick}[i] ); \\
\text{THINK}
\end{align*}
\]

• Deadlock
  
  a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set
  
  • details in Lectures 10 and 11.

Altimate solutions
  
  – allow at most 4 philosophers to sit simultaneously at the table
  
  – allow a philosopher to pick up chopsticks only if both are available

• All of these solutions suffer from the possibility of starvation!
Next Lecture (February 20, 2002)

- A larger synchronization example

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

Reading
- Silberschatz/Galvin/Gagne: Sections 7.6 – 7.8