Outline

- Announcements
  - Lab 2 due today; Demos on Feb 22nd
  - Write-up 2 handed out today; due February 28th
  - Questions?

- Process synchronization (contd.)
  - Limitations of semaphores
  - Language support
    - Conditional critical regions
    - Monitors
  - Message passing

[Silberschatz/Galvin: Sections 6.6-6.7]

Limitations of Semaphores

- No abstraction and modularity
  - A process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
  - A process cannot be written in isolation
  - Why?

- Consider sequencing between three processes
  - \( P_1, P_2, P_3, P_1, P_2, P_3, \ldots \)

\[
\begin{align*}
P_1 & \rightarrow P(\text{sem}_1); \\
P_2 & \rightarrow P(\text{sem}_2); \\
P_3 & \rightarrow P(\text{sem}_3);
\end{align*}
\]

// do stuff // do stuff // do stuff

\[
\begin{align*}
P_1 & \rightarrow V(\text{sem}_1); \\
P_2 & \rightarrow V(\text{sem}_2); \\
P_3 & \rightarrow V(\text{sem}_3);
\end{align*}
\]

What happens if there are only two processes?
What happens if you want to use this solution for four processes?

Limitations of Semaphores (contd.)

- Very easy to write incorrect code
  - Changing the order of P and V can violate mutual exclusion requirements
    \[
    V(\text{mutex}); \text{CODE}; P(\text{mutex}); \text{instead of}
    P(\text{mutex}); \text{CODE}; V(\text{mutex});
    \]
  - Can cause deadlock
    \[
    P(\text{seq}); \text{instead of}
    V(\text{seq});
    \]

- Similar problems with omission

- Extremely difficult to verify programs for correctness
  - Need for still higher-level synchronization abstractions!
Language Support

- Helps simplify expression of synchronization
  - more convenient
  - more secure
  - less buggy

- We shall examine two fundamental constructs
  - conditional critical regions
  - monitors

- These constructs can be found in several concurrent languages
  - Communicating Sequential Processes (CSP)
  - Concurrent Pascal
  - object-oriented languages: Modula-2, Concurrent C, Java
  - Ada83, Ada95

Conditional Critical Regions

- A high-level language declaration
  - informally, it can be used to specify that while a statement $S$ is being executed, no more than one process can access a distinguished variable $v$
  - notation

\[
\text{var } v: \text{shared } t;\\
\text{region } v \text{ when } B \text{ do } S;\\
\]

- $v$ is shared and of type $t$
  - can only be accessed within a region statement
- $B$ is a Boolean expression
- $S$ is a statement
  - can be a compound statement

- Semantics
  - A process is guaranteed mutually exclusive access to the region $v$
  - Checking of $B$ and entry into the region happens atomically

Conditional Critical Regions: Benefits

Bounded-buffer producer/consumer

```plaintext
var buffer : shared record
pool: array [0..n-1] of item;
count, in, out: integer;
end;

Producer:
region buffer when count < n
do begin
  pool[in] := nextp;
in := (in + 1) mod n;
count := count + 1;
end;

Consumer:
region buffer when count > 0
do begin
  nextc := pool[out];
  out := (out + 1) mod n;
count := count - 1;
end;
```

- Guards against simple errors associated with semaphores
  - e.g., changing the order of $P$ and $V$ operations, or forgetting to put one of them

- Division of responsibility
  - the developer does not have to program the semaphore or alternate synchronization explicitly
  - the compiler "automatically" plugs in the synchronization code using predefined libraries
  - once done carefully, reduces likelihood of mistakes in designing the delicate synchronization code

Conditional Critical Regions: Implementation

```plaintext
var mutex: semaphore;
P( mutex );
while not B
do begin
  try-and-enter;
  end;
S;leave-critical-region;

var delay: semaphore;
count++ ;
V( mutex );
P( delay );
// check condition
if ( not B )
  if ( count > 1 )
    // release another
    V( delay );
P( delay );
else
  V( mutex );
P( delay );
else count--;
if ( count > 0 )
  then V( delay );
else V( mutex );
```

```plaintext
var first, second: semaphore;
var fcount, account: integer;

if ( account > 0 )
  V( second );
else V( mutex );
P( first );
if count-- ;
if ( fcount > 0 )
  V( second );
else V( mutex );
```

```plaintext
if ( fcount > 0 )
  V( first );
else if ( account > 0 )
  V( second );
else V( mutex );
```
Monitors

- An abstract data type
  - private data
  - public procedures
    - only one procedure can be in the monitor at one time
    - each procedure may have
      - local variables
      - formal parameters
  - condition variables
    - queues of processes
    - wait: block on a condition variable
    - signal: unblock a waiting process
      - no-op if no process is waiting

- Processes can only invoke the public procedures
  - raises the granularity of atomicity to a single user-defined procedure

Waiting in the Monitor

- Note that the semantics of executing a wait in the monitor is that several processes can be waiting “inside” the monitor at any given time but only one is executing
  - wait queues are internal to the monitor
  - there can be multiple wait queues

- Who executes after a signal operation? (say P signals Q)
  - signalee Q continues (advocated by Hoare)
    - logically natural since the condition that enabled Q might no longer be true when Q eventually executes
    - P needs to wait for Q to exit the monitor
  - signaller P continues (referred to as Mesa-style)
    - Q is enabled but gets its turn only after P either leaves or executes a wait
    - require that the signal be the last statement in the procedure
      - advocated by Brinch Hansen (Concurrent Pascal)
      - easy to implement but less powerful than the other two semantics

Use of Monitors: Bounded-buffer

```pascal
type bounded_buffer = monitor
var buffer: array [0..N] of char;
var in, out, count: integer;
var notfull, notempty: condition;
procedure entry append(x: char);
  begin
    if (count==N) notfull.wait;
    buffer[in] := x;
in := (in+1) mod N;
count := count+1;
notempty.signal;
  end;
procedure entry remove(x: char);
  begin
    if (count==0) notempty.wait;
    x := buffer[out];
    out := (out+1) mod N;
count := count-1;
notfull.signal;
  end;
end;
```

Is this solution correct under all monitor semantics? (P signals Q)
  Hoare: Q continues, P suspends ....................... YES
  Mesa: P continues, Q is put into ready queue ........... NO
  Brinch-Hansen: P exits monitor, Q continues ............ YES

Use of Monitors: Bounded-buffer (Mesa Semantics)

```pascal
procedure entry append(x: char);
  while (count==N) notfull.wait;
  buffer[in] := x;
in := (in+1) mod N;
count := count+1;
notempty.signal;
end;
```

```pascal
procedure entry remove(x: char);
  while (count==0) notempty.wait;
  x := buffer[out];
  out := (out+1) mod N;
count := count-1;
notfull.signal;
end;
```
Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks

- Informally:
  - algorithm for Philosopher I
    - dp.pickup(i);
    - eat;
    - dp.putdown(i);
  - use array to describe state
    - var state: array [0..4] of (thinking, hungry, eating);
  - use array of condition variables to block on when required resources are unavailable
    - var self: array [0..4] of condition;

- pickup(i)
  - changes state to hungry
  - checks if neighbors are eating
  - if not, grabs forks, and changes state to eating
  - otherwise, waits on self(i)

- putdown(i)
  - checks both neighbors
  - if either is hungry and can proceed, releases him/her

Dining Philosophers using Monitors - 2

type dining_philosophers = monitor

var state: array [0..4] of (thinking, hungry, eating);
var self: array [0..4] of condition;

procedure entry pickup ...
procedure entry putdown ...
procedure test ...

begin
  for i := 0 to 4 do
    state[i] := thinking;
  end;

procedure entry pickup(i: 0..4);

var state[i] := hungry;
test(i);
while ( state[i] != eating )
  self[i].wait;

procedure entry putdown(i: 0..4);

var state[i] := thinking;
test (ln(i));
test (rn(i));

procedure test(i: 0..4);
  if (state[ln(i)] != eating and state[i] == hungry and state(rn(i)) != eating)
    state[i] := eating; self[i].signal;

Dining Philosophers using Monitors - 3

- What is missing?
  - philosophers cannot deadlock but can starve
    - for example, we can construct timing relationships such that a waiting philosopher will be stuck in the “self” queue forever
  - monitors have to be enhanced with a fair scheduling policy to avoid starvation
    - both at the level of accessing the monitor
    - as well as to regulate “waking-up” those that are waiting inside
  - how can this be done?
    - use fair enqueue and dequeue policies

Monitors: Other Issues

- Expressibility: Are monitors more/less powerful than semaphores or conditional critical regions?
  - these three constructs are equivalent
    - the same kinds of synchronization problems can be expressed in each
  - the other two can be implemented using any one of the constructs
    - e.g., critical regions and monitors using semaphores
      - we talked about how critical regions can be implemented
      - in Lab 2, you built condition variables using semaphores
        - this implementation can be extended to build monitors

- Do monitors have any limitations?
  - absence of concurrency within a monitor
    - workarounds introduce all the problems of semaphores
  - monitor procedures will need to be invoked before and after
    - possibility of improper access, deadlock, etc.
Synchronization and Communication

- Synchronization primitives
  - assuming shared memory
    - locks
    - semaphores
    - monitors
- Synchronization can also be constructed using message-passing
  - message passing primitives combine data transfer and synchronization
    - a receive blocks for a message: equivalent to a `wait`
    - a send enables a process blocking on a receive to make progress: equivalent to a `signal`

Message Passing: Semantics

- A pair of primitives (available as system calls)
  
  ```
  send( destination, message )
  receive( source, message )
  ```

- Synchronization semantics
  - receiver cannot receive a message until it has been sent by another process
  - what happens to a process after it issues a send or receive primitive?
    - **Blocking send, blocking receive**: both sender and receiver are blocked until the message is delivered (also known as rendezvous)
    - **Nonblocking send, blocking receive**: sender can proceed, receiver blocks until the requested message arrives
    - **Nonblocking send, nonblocking receive**: neither party is required to wait.
      Receive returns success/failure

Message Passing: Addressing

- `send` must specify which process is to receive the message
  - most implementations also allow `receive` to specify the message source
- Direct addressing
  - use PIDs to indicate destination for `send` or source for `receive`
  - also possible to have an anonymous `receive`
    - return value indicates the source process
      - e.g., a print server process can accept a print request from any process
- Indirect addressing
  - messages are sent to a `shared data structure` from where they are retrieved

Uses of Message Passing

- Mutual exclusion
  
  ```
  receive( mutex, msg );  // mutex is a mailbox with an initial message
  CRITICAL-SECTION;
  send( mutex, msg );
  ```

- Bounded-buffer
  - `mayproduce` is a mailbox with N initial messages
  - `mayconsume` is a mailbox, which is empty initially

  ```
  Producer
  while (1) {
    receive( mayproduce, pmsg );
    pmsg := produce;
    consume( cmsg );
  }

  Consumer
  while (1) {
    receive( mayconsume, cmsg );
    consume( cmsg );
    send( mayproduce, null );
  }
  ```
Implementation of Message Passing Primitives

- Can implement message primitives using shared memory synchronization primitives
  - and vice-versa
- E.g., using monitors to build a mailbox

```pascal
type message_mailbox = monitor

var msgQ: queue of msg;
var notempty: condition;

procedure entry send(m: msg);
  msgQ.enqueue( m );
  notempty.signal;
procedure entry receive(m: msg);
  while ( msgQ.empty() )
    notempty.wait;
  msgQ.dequeue( m );

begin
  // initialize msgQ
end;
```

- How to build a mailbox with bounded capacity?

Next Lecture

- Process Deadlocks
  - system model
  - deadlock characterization
  - methods for handling deadlocks
  - deadlock prevention

Readings

- Silberschatz/Galvin: Sections 7.1 - 7.4

- Reminder: No class on Monday!