Outline

• Announcements
  – Lab 2 demos today and tomorrow
  – No class on Wednesday, February 25\textsuperscript{th}
  – Questions?

• Process synchronization (cont’d)
  – Language support for synchronization
    • Critical regions
    • Monitors

• (Time permitting) CPU Scheduling
  – basic concepts
  – scheduling criteria
  – scheduling algorithms

[Silberschatz/Galvin/Gagne, Sections 7.5 – 7.8, Sections 6.1 – 6.3]
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

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

```
P_1      P_2      P_3
P( sem_1 ); P( sem_2 ); P( sem_3 );
// do stuff    // do stuff    // do stuff
V( sem_2 );   V( sem_3 );   V( sem_1 );
```

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)  critical regions
  – Concurrent Pascal  monitors
  – 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

```pascal
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;
var count: integer;
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 );

var first, second: semaphore;
var fcount, scount: integer;
fcount++ ;
if ( scount > 0 ) V( second );
else V( mutex );
P( first );
fcount-- ;
scount++ ;
if ( fcount > 0 ) V( first );
else V( second );
P( second );
scount-- ;
if ( fcount > 0 ) V( first );
else if ( scount > 0 ) V( second );
else V( mutex );
```

2/23/2004
Language Support (2): 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)
  – (Hoare semantics) signalleee Q continues
    • 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
  – (Mesa semantics) signaller P continues
    • 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);
  if (count==N) notfull.wait;
  buffer[in] := x;
  in := (in+1) mod N;
  count := count+1;
  notempty.signal;

begin
  in = 0; out = 0; count = 0;
end;

procedure entry remove(x: char);
  if (count==0) notempty.wait;
  x := buffer[out];
  out := (out+1) mod N;
  count := count-1;
  notfull.signal;

begin
  in = 0; out = 0; count = 0;
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
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);
  while (count==N) notfull.wait;
  buffer[in] := x;
in := (in+1) mod N;
count := count+1;
notempty.signal;
end;

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

begin
  in = 0; out = 0; count = 0;
end;
Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks

- Informally:
  - algorithm for Philosopher I
    ```plaintext
dp.pickup(i);
est;
dp.putdown(i);
```
  - use array to describe state
    ```plaintext
var state: array [0..4] of (thinking, hungry, eating);
```
  - use array of condition variables to block on when required resources are unavailable
    ```plaintext
var self: array [0..4] of condition;
```
  - pickup(i)
    - changes state to hungry
    - checks if neighbors are eating
    - if not, grabs chopsticks, 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
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);
   state[i] := hungry;
   test(i);
   while ( state[i] != eating )
      self[i].wait;

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

• Announcements
  – Lab 2 demos today and tomorrow
  – No class on Wednesday, February 25th
  – Questions?

• Process synchronization (cont’d)
  – Language support for synchronization
    • Critical regions
    • Monitors

• (Time permitting) CPU Scheduling
  – basic concepts
  – scheduling criteria
  – scheduling algorithms

[Silberschatz/Galvin/Gagne, Sections 7.5 – 7.8, Sections 6.1 – 6.3]
CPU Scheduling: Overview

• What is scheduling?
  – Simply deciding *which process to execute* and for how long

• Why do we need it?
  – Better resource utilization
  – Improve the system performance for desired load pattern
  – Support multitasking for interactive jobs
    • Example: Editing and compiling
  – Can enable providing of specific guarantees
Scheduling: Components

• Processes

• Scheduler
  – focus on short-term scheduling (of the CPU)
  – decide which process to give the CPU to next
    • rationale: utilize CPU resource better
    • can also be necessary because of other factors: fairness, priorities, etc.

• Dispatcher:
  – suspends previous process and (re)starts new process
    • context switch, including adjusting and updating the various process queues
    • switch to user mode from the scheduler's supervisor mode
    • jump to the appropriate point in user space and resume executing “running” process
Scheduling: Operation Details

- (Review) Queues associated with process states
  - Running, Ready, Waiting

- Scheduler invoked in the following situations (triggers)
  - process switches from running to waiting state
    - e.g., block for I/O, wait for child
  - process switches from running to ready state
    - e.g., expiration of timer
  - process switches from waiting to ready state
    - e.g., completion of I/O
  - process terminates
Preliminaries: Model of Process Behavior

- CPU versus I/O bursts
  - a given process’ behavior is broken into
    - a run of activity on the CPU referred to as a *CPU burst*
    - a run of non-CPU (usually I/O) activity or an *I/O burst*
  - the overall execution of a process is alternating CPU and I/O bursts

- CPU burst lengths typically characterized as *exponential* or *hyperexponential*
  - CPU bound processes: few, long CPU bursts
  - I/O bound processes: many, very-short CPU bursts

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>IO</th>
<th></th>
<th>CPU</th>
<th>IO</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>10</td>
<td>1000</td>
<td>15</td>
<td>4000</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Process 2</td>
<td>20</td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
Preliminaries: Preemption

- *Preemptive* versus *non-preemptive* scheduling
  - the corresponding scheduling *policy* is non-preemptive
    - if a process switches to a waiting state *only* as a function of its own behavior
      - i.e. when it invokes OS services, or when it terminates
  - it is preemptive
    - if its state can be switched otherwise

- Cost: Maintaining consistent system state while the processes are suspended in the midst of critical activity
  - suspension might need interrupts to be turned off
    - e.g., the process being suspended is updating sensitive kernel data-structures
    - however, interrupts cannot always be ignored
  - poses challenging problems to coordinate the states of processes interrupted in a preemptive way
## Preliminaries: Scheduling Metrics

### User Oriented

**Performance Related**
- *response time*: time it takes to produce the first response
- *turnaround time*: time spent from the time of “submission” to time of completion
- *deadlines*: the time within which the program must complete (the policy must maximize percentage of deadlines met)

**Other**
- *predictability*: expectation that the job runs the same regardless of system load

### System Oriented

**Performance Related**
- *waiting time*: time spent waiting to get the CPU
- *throughput*: the number of processes completed per unit time (directly affected by the waiting time)
- *CPU utilization*: percentage of time the CPU is busy

**Other**
- *fairness*: no process should suffer starvation
- *enforcing priorities*: higher priority processes should not wait
Scheduling Algorithms (1)
First-come First-served (FCFS)

- Non-preemptive

- Implementation
  - a queue of processes
  - new processes enter the ready queue at the end
  - when a process terminates
    - the CPU is given to the process at the beginning of the queue
  - (in practice) when a process blocks
    - it goes to the end of the queue
    - the CPU is given to the process at the beginning of the queue

- How does FCFS perform?
Performance of FCFS

• 3 processes P1, P2, and P3 with CPU requirements 24, 3, and 3 msecs
  - Arrive at the same time in that order

  \[
  \begin{array}{ccc}
  \text{P 1} & \text{P 2} & \text{P 3} \\
  \end{array}
  \]

  - Average waiting time = \(\frac{0+24+27}{3} = 17\)
  - Average turnaround time = \(\frac{24+27+30}{3} = 27\)
  - Average throughput = \(\frac{30}{3} = 10\)
  - Can we do better?

  \[
  \begin{array}{ccc}
  \text{P 2} & \text{P 3} & \text{P 1} \\
  \end{array}
  \]

  - Average waiting time = \(\frac{0+3+6}{3} = 3\) !!!
  - Average turnaround time = \(\frac{3+6+30}{3} = 13\) !!!
  - Average throughput = \(\frac{30}{3} = 10\)
Evaluation of FCFS

• *Pro*: Very simple code, data-structures and hence low overhead

• *Con*: Can lead to large average waiting times

• General disadvantage due to lack of preemption
  – when a poorly (long-term) scheduled collection has one large task with lots of CPU needs and a collection of others with I/O intensive needs
    • the CPU intensive process can cause very large delays for the processes needing (mostly) I/O