(Review)

Language Support (1): 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

  ```
  var $v$: shared $t$;
  region $v$ when $B$ 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
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) signallee 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

```delphi
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;
```

Use of Monitors: Bounded-buffer (Mesa Semantics)

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

procedure entry remove(x: char);
begin
  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 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

Dining Philosophers using Monitors - 2

```plaintext
type dining_philosophers = monitor

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

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;
end;
```

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;
    - send(mayconsume, pmsg);
  }

  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

```haskell
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);
```

- How to build a mailbox with bounded capacity?

Outline

- Announcements
  - Lab 2 due today, demos on Feb 27th, 28th
  - Lab 3 out today, due back March 18th (Monday after Spring Break)
    - Extra credit part (priority scheduler) from Lab 2
- Language support for synchronization
  - Critical regions
  - Monitors
  - Message passing
- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
[Silberschatz/Galvin/Gagne: Sections 7.5 – 7.8, 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: Operating 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

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>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>5</td>
</tr>
<tr>
<td>Process 2</td>
<td>20</td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

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

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)

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

- predictability: expectation that the job runs the same regardless of system load
- fairness: no process should suffer starvation
- enforcing priorities: higher priority processes should not wait
FCFS Scheduling

- 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
    - Average waiting time = (0+24+27)/3 = 17
    - Average turnaround time = (24+27+30)/3 = 27
    - Average throughput = (30)/3 = 10
    - Can we do better?
  - Average waiting time = (0+3+6) / 3 = 3 !!!
  - Average turnaround time = (3+6+30)/3 = 13 !!!
  - Average throughput = (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

Shortest Job First (SJF)

- The next process to be assigned the CPU is one that is ready and with smallest next CPU burst; FCFS is used to break ties
  - From the previous example,
    - P1, P2, P3 arrive at the same time in that order, needing CPU times 24, 3, 3
      - FCFS yielded an average waiting time of 17 units
      - SJF yields order P2, P3, P1, with average waiting time of 3 units
  - Another example
    - P1, P2, P3, P4 requiring bursts of 8, 9, 4, and 5 arrive 1 time unit apart
      - FCFS: Average waiting time = ( 0 + (8 – 1) + (17 – 2) + (21 – 3) )/4 = 10 units
      - SJF: Average waiting time = (0 + (17 – 1) + (8 – 2) + (12 – 3))/4 = 7.75 units
Evaluation of SJF

- **Pro**: If times are accurate, SJF gives *minimum* average waiting time

**Estimating the burst times**
- For long-term scheduling, user can be “encouraged” to give estimate
  - part of the job submission requirements
- For short-term scheduling, scheduler attempts to predict value
  - the approach assumes some *locality* in process CPU burst times
    - Use exponential averaging
      - $\tau_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot \tau_n$
    - where,
      - $\tau_n$ is the estimated value for the $n$'th CPU burst
      - $T_n$ is the actual most recent burst value
    - $\alpha = 0$ implies fixed estimate; $\alpha = 1$?; $\alpha = 0.5$?
      - the estimate lags the (potentially) sharper transitions of the CPU bursts

Next Lecture

- CPU Scheduling (contd.)
  - Scheduling algorithms: SJF, priority-based, round-robin
  - Practical implementations: Multi-level queues
  - Real-time scheduling

- Reading:
  - Silberschatz/Galvin/Gagne: Sections 6.3 – 6.6