Outline

- Announcements
  - Lab 2 demos today (Feb 26th) and tomorrow (Feb 27th)

- Process synchronization (cont’d)
  - Language support for synchronization (cont’d)
    - Monitors

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms

[Silberschatz/Galvin/Gagne: Sections 7.7, 6.1 – 6.3]
Use of Monitors: Bounded-buffer

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

procedure entry remove(x: char);
    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)

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;

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

type dining_philosophers = monitor

procedure entry pickup(i: 0..4);
    state[i] := hungry;
    test(i);
    while (state[i] != eating ) self[i].wait;
end;

procedure entry putdown(i: 0..4);
    state[i] := thinking;
    test (ln(i));
    test (rn(i));
end;

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.

Outline

- Announcements
  - Lab 2 demos today (Feb 26th) and tomorrow (Feb 27th)

- Process synchronization (cont’d)
  - Language support for synchronization (cont’d)
    - Monitors

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms

Silberschatz/Galvin/Gagne: Sections 7.7, 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)

- **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

  - 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
Scheduling Algorithms (2)
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

\[
\begin{align*}
\text{P1} & \quad \text{P2} & \quad \text{P3} & \quad \text{P4} \\
\end{align*}
\]

FCFS: Average waiting time = \((0 + (8 – 1) + (17 – 2) + (21 – 3))/4\) = 10 units

\[
\begin{align*}
\text{P1} & \quad \text{P3} & \quad \text{P4} & \quad \text{P2} \\
\end{align*}
\]

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
    - \(\tau_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot \tau_n\)
    - \(\tau_n\) 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

Estimating the CPU Burst (contd.)

\[
\begin{align*}
\text{time} & \quad \text{CPU burst (t)} & \quad \text{“gusts” (t)} \\
& \quad 2 & \quad 4 & \quad 6 & \quad 8 & \quad 10 & \quad 12 \\
6 & \quad 4 & \quad 6 & \quad 13 & \quad 13 & \quad 13 & \quad \ldots \\
10 & \quad 8 & \quad 6 & \quad 6 & \quad 5 & \quad 9 & \quad 11 & \quad 12 & \quad \ldots \\
\end{align*}
\]

\text{Figure 5.3} \text{ Prediction of the length of the next CPU burst.}

Modifications to SJF

- Preemptive SJF (also called shortest remaining time first)
  - if the shortest estimated CPU burst among all processes in the ready queue is less than the remaining time for the one running,
    - preempt running process; add it to ready queue w/ remaining time
    - give CPU to process with the shortest CPU burst
  - policy prioritizes jobs with short CPU bursts

- Example: A, B, C, D with bursts 8, 9, 4, 5 arrive 1 time unit apart

\[
\begin{align*}
\text{A} & \quad \text{B} & \quad \text{C} & \quad \text{D} \\
\end{align*}
\]

SJF: Average waiting time = \((0 + (17 – 1) + (8 – 2) + (12 – 3))/4\) = 7.75 units

\[
\begin{align*}
\text{A} & \quad \text{C} & \quad \text{D} & \quad \text{A} & \quad \text{B} \\
\end{align*}
\]

Preemptive SJF: Average waiting time = \((0 – 0 + 9) + (17 – 1 + 0) + (2 – 2 + 0) + (6 – 3 + 0))/4 = 7 units
Scheduling Algorithms (3)
Priorities: A More General Scheduling Notion

- Elements of a priority-based scheduler
  - Process priorities (for example 0..100)
    - convention: a smaller number means higher priority
  - Tie-breaker mechanism
    - Example: FCFS
  - Map priority to considerations we have in mind
    - Internal
      - memory and other needs of the job
      - ratio of CPU to I/O burst times
      - number of open files etc.
    - External
      - the amount of money paid by the process owner
      - the importance of the user group running the process

- Priority-based scheduling
  - assign the CPU to the process with highest priority
  - may be used with or without preemption

Priority-based Scheduling: Example

- Consider five processes A, B, C, D, and E
  - With burst times: 10, 1, 2, 1, 5
  - With priorities: 3, 1, 3, 4, 2 (lower is better)
  - Arriving at times: 0, 0, 2, 2, 3

Without preemption:

\[
\text{Average waiting time: } \frac{1 - 0 + 0 + 16 - 2 + 18 - 2 + 11 - 3}{5} = 7.8
\]

With preemption:

\[
\text{Average waiting time: } \frac{1 - 0 + 7 + 0 - 0 + 16 - 2 + 18 - 2 + 3 - 3}{5} = 7.6
\]

Problems with Priority Schemes

- Process can be overtaken by higher priority processes arriving later
  - can happen continuously: leads to starvation
  - leads to better overall performance perhaps
    - but not from the point of view of the process in question

- Common solution: A process’ priority goes up with its age
  - FCFS is used to break ties between processes with equal priorities
  - For a process in ready queue, its priority will eventually be the highest

- A low-priority process holds resources required by a high-priority process? (priority inversion)

- Common solution: Priority inheritance
  - process with lock inherits priorities of processes waiting for the lock
  - priority reverts to original values when lock is released

Example of Priority Ageing: Unix

- Priority goes up with lack of CPU usage
  - process accumulates CPU usage
  - every time unit (~ 1 second)
    - recalculates priority
      \[
priority = \text{CPU usage} + \text{base priority}
\]
    - halves CPU usage carried forward
      \[
      \text{CPU usage} = \frac{(\text{CPU usage})}{2}
      \]
    - recall that smaller number implies a higher priority
    - base priority is settable by user
      - within limits
      - using “nice”

- Assuming all processes have the same base priority:
  - Are new processes prioritized over existing ones?
  - How does the priority of a process change over its lifetime?