Your job this week

• Readings in Silberschatz
  – Chapter 7
• No homework this week instead there is a midterm exam on Friday
But First a Brief Word About Monitors  
(Section 6.7)

- The term *monitor* used in this context is not to be confused with *monitor mode* used to describe a hardware protection
- A synchronization monitor is a programming language construct that supports controlled access to shared data
- Essentially it is an ADT that encapsulates
  - Some shared data structures
  - Procedures/methods to access the data
  - Synchronization build into the procedures (using *condition variables*)

Today

- All these various synchronization methods are great for keeping concurrent processes/threads from mangling each other
- However they do introduce another problem
- Synchronization does not stop them from starving each other
- And that’s the topic of deadlocks
- We need to look at what deadlocks are and how to deal with them
Deadlock

- **Deadlock** is a problem that can exist when a group of processes compete for access to fixed resources.
- Def: deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
- Example: two processes share 2 resources that they must request (before using) and release (after using). Request either gives access or causes the proc. to block until the resource is available.

```plaintext
Proc1:
request tape
request printer
... <use them>
release printer
release tape

Proc2:
request printer
request tape
... <use them>
release tape
release printer
```

Four Conditions for Deadlock

- Deadlock can exist if and only if 4 conditions hold simultaneously:

1. **mutual exclusion**: at least one process must be held in a non-sharable mode.
2. **hold and wait**: there must be a process holding one resource and waiting for another.
3. **No preemption**: resources cannot be preempted.
4. **circular wait**: there must exist a set of processes [p1, p2, ..., pn] such that p1 is waiting for p2, p2 for p3, and so on....
Resource Allocation Graph

- Deadlock can be described through a resource allocation graph.
- The RAG consists of a set of vertices \( P = \{ P_1, P_2, \ldots, P_n \} \) of processes and \( R = \{ R_1, R_2, \ldots, R_m \} \) of resources.
- A directed edge from a process to a resource, \( P_i \rightarrow R_j \), implies that \( P_i \) has requested \( R_j \).
- A directed edge from a resource to a process, \( R_j \rightarrow P_i \), implies that \( R_j \) has been allocated by \( P_i \).
- If the graph has no cycles, deadlock cannot exist. If the graph has a cycle, deadlock may exist.

Resource Allocation Graph Example

There are two cycles here: \( P_1-R_1-R_2-P_2-P_3-R_3-P_1 \) and \( P_2-R_3-P_3-R_2-P_2 \), and there is deadlock.

Same cycles, but no deadlock.
Possible Approaches

- **Deadlock Prevention**: ensure that at least 1 of the necessary conditions cannot exist.
  - Mutual exclusion: make resources shareable (isn’t really possible for some resources)
  - hold and wait: guarantee that a process cannot hold a resource when it requests another, or, make processes request all needed resources at once, or, make it release all resources before requesting a new set
  - circular wait: impose an ordering (numbering) on the resources and request them in order

More Possible Approaches

- **Deadlock Avoidance**
  - General idea: provide information in advance about what resources will be needed by processes to guarantee that deadlock will not exist.
  - E.g., define a sequence of processes \(<P_1,P_2,..P_n>\) as **safe** if for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus the resources held by all \(P_j, j < i\).
    - This avoids circularities.
    - When a process requests a resource, the system grants or forces it to wait, depending on whether this would be an unsafe state.
Example:

• Processes p0, p1, and p2 compete for 12 tape drives

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

3 drives remain

• Current state is safe because a safe sequence exists: <p1,p0,p2>
  p1 can complete with current resources
  p0 can complete with current+p1
  p2 can complete with current +p1+p0

• If p2 requests 1 drive, then it must wait because that state would be unsafe.

The Banker’s Algorithm

• Banker’s algorithm decides whether to grant a resource request. Define data structures.

```c
int n;                // # of processes
int m;                // # of resources
int available[m];     // # of avail resources of type i
int max[n][m];        // max demand of each P_i for each R_j
int allocation[n][m]; // current allocation of resource R_j to P_i
int need[n][m];       // max # of resources that P_i may still request of R_j
```

let request[i] be a vector of the # of instances of resource R_j that Process P_i wants.
The Basic Banker’s Algorithm

if (request[i] > need[i]) {
    // error, asked for too much
}

if (request[i] > available[i]) {
    // wait until resources become available
}

// resources are available to satisfy request, assume
// that we satisfy the request, we would then have

available = available - request[i];
Allocation[i] = allocation[i] + request[i];
need[i] = need[i] - request[i];

// now check if this would leave us in a safe state
// if yes then grant the request otherwise the process
// must wait

Safety Check in Banker’s Algorithm

int work[m] = available; // to accumulate resources
boolean finish[n] = {FALSE,…}; // non finished yet

do {
    find an i such that (finish[i]==FALSE) && (need[i]<work)
    // process i can complete all of its requests

    finish[i] = TRUE; // done with this process

    work = work + allocation[i]; // assume this process gave
    // all its allocation back
}

} until (no such i exists);

if (all finish entries are TRUE) {
    // system is safe. i.e., we found a sequence a processes
    // that will lead to everyone finishing
}
Deadlock Detection

- If there is neither deadlock prevention nor avoidance, then deadlock may occur.
- In this case, we must have:
  - an algorithm that determines whether a deadlock has occurred
  - an algorithm to recover from the deadlock
- This is doable, but it’s costly

Deadlock Detection Algorithm

```c
int work[m] = available;       // to accumulate resources
boolean finish[n] = {FALSE,...}; // non finished yet

for (i = 0; i < n; i++) {
    if (allocation[i] is zero) { finish[i] = TRUE; }
}

do {
    find an i such that (finish[i]==FALSE && request[i]<work)
    // process I can finish with currently available resources
    finish[i] = TRUE;            // done with this process
    work = work + allocation[i]; // assume this process gave 
    // all its allocation back
} until (no such i exists);

if (finish[i] == FALSE for some i) {
    // System is deadlocked with Pi in the deadlock cycle
}
```
Deadlock Detection

- Deadlock detection algorithm is expensive. How often we invoke it depends on:
  - how often or likely is deadlock
  - how many processes are likely to be affected when deadlock occurs

Deadlock Recovers

- Once a deadlock is detected, there are 2 choices:
  1. abort all deadlocked processes (which will cost in the repeated computations necessary)
  2. abort 1 process at a time until cycle is eliminated (which requires re-running the detection algorithm after each abort)
- Or, could do process preemption: release resources until system can continue. Issues:
  - selecting the victim (could be clever based on R’s allocated)
  - rollback (must rollback the victim to a previous state)
  - starvation (must not always pick same victim)
- These are common database inspired methods, within an interactive OS none are really that acceptable
Real Life Deadlock Prevention

- Fewer resources (locks) means less deadlock potential, but also less potential concurrency. So there is a trade off here
- For really simple applications acquiring all the resources up front is fairly common, but not always practical.
- Programmers most often use common sense in the ordering of resources acquisition and releases
  - Resource levels is one area that helps development
- In complicated software systems resource levels are not practical. (e.g., memory management and the file system often recursively call each other), and deadlock prevention is far more a matter of fine tuning the locks and understanding the exact scenario in which locks are acquired

Important Points to Remember About Deadlocks

- When a deadlock does happen, by definition, it will not go away; therefore debugging deadlocks is somewhat simpler because all the processes are stuck and can’t squirm out of the way.
- Identifying a deadlock is sometimes easier then understanding how to prevent the deadlock
- No magic bullet here, but a lot of common sense
Next Time

- Review on Wednesday and
- Midterm on Friday
  - Closed book
  - Closed notes
  - Closed neighbor
  - Open mind
  - Roughly 10 questions covering
    - Hardware Support
    - OS Architecture
    - Processes and Threads
    - Scheduling Algorithms
    - Synchronization, and
    - The Linux Projects