Last Class: Synchronization Problems

• Reader Writer
  – Multiple readers, single writer
  – In practice, use read-write locks

• Dining Philosophers
  – Need to hold multiple resources to perform task

Dining Philosophers

• It’s lunch time in the philosophy dept
• Five philosophers, each either eats or thinks
• Share a circular table with five chopsticks
• Thinking: do nothing
• Eating => need two chopsticks, try to pick up two closest chopsticks
  – Block if neighbor has already picked up a chopstick
• After eating, put down both chopsticks and go back to thinking
Dining Philosophers v1

Semaphore chopstick[5];

do{
    wait(chopstick[i]);   // left chopstick
    wait(chopstick[(i+1)%5 ]);  // right chopstick
    // eat
    signal(chopstick[i]);  // left chopstick
    signal(chopstick[(i+1)%5 ]);  // right chopstick
    // think
} while(TRUE);

Dining Philosophers (semaphores)

#define N 5           /* number of philosophers */
#define LEFT (i+N-1)%N/* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0    /* philosopher is thinking */
#define HUNGRY 1      /* philosopher is trying to get forks */
#define EATING 2      /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */{
    while (TRUE) /* repeat forever */{
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}
Dining Philosophers (contd)

```c
void take_forks(int i)  /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);        /* enter critical region */
    state[i] = HUNGRY;   /* record fact that philosopher i is hungry */
    test(i);            /* try to acquire 2 forks */
    up(&mutex);         /* exit critical region */
    /* block if forks were not acquired */
}
void put_forks(i)       /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);        /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);         /* see if left neighbor can now eat */
    test(RIGHT);        /* see if right neighbor can now eat */
    up(&mutex);         /* exit critical region */
}
void test(i)            /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&state[i]);
    }
}
```

Real-world Examples

- **Producer-consumer**
  - Audio-Video player: network and display threads; shared buffer
  - Web servers: master thread and slave thread
- **Reader-writer**
  - Banking system: read account balances versus update
- **Dining Philosophers**
  - Cooperating processes that need to share limited resources
    - Set of processes that need to lock multiple resources
      - Disk and tape (backup),
    - Travel reservation: hotel, airline, car rental databases
Today: Deadlocks

- What are deadlocks?
- Conditions for deadlocks
- Deadlock prevention
- Deadlock detection

Deadlocks

- **Deadlock**: A condition where two or more threads are waiting for an event that can only be generated by these same threads.
- Example:

  ```
  Process A:           Process B:
  printer.Wait();     disk.Wait();
disk.Wait();         printer.Wait();

  // copy from disk    // copy from disk
  // to printer        // to printer

  printer.Signal();   printer.Signal();
disk.Signal();       disk.Signal();
  ```
Deadlocks: Terminology

- **Deadlock** can occur when several threads compete for a finite number of resources simultaneously.
- **Deadlock prevention** algorithms check resource requests and possibly availability to prevent deadlock.
- **Deadlock detection** finds instances of deadlock when threads stop making progress and tries to recover.
- **Starvation** occurs when a thread waits indefinitely for some resource, but other threads are actually using it (making progress).
  => Starvation is a different condition from deadlock.

Necessary Conditions for Deadlock

Deadlock can happen if all the following conditions hold.

- **Mutual Exclusion**: at least one thread must hold a resource in non-sharable mode, i.e., the resource may only be used by one thread at a time.
- **Hold and Wait**: at least one thread holds a resource and is waiting for other resource(s) to become available. A different thread holds the resource(s).
- **No Preemption**: A thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource.
- **Circular wait**: A set of waiting threads \( \{t_1, ..., t_n\} \) where \( t_i \) is waiting on \( t_{i+1} \) \( (i = 1 \text{ to } n) \) and \( t_n \) is waiting on \( t_1 \).
Deadlock Detection Using a Resource Allocation Graph

- We define a graph with vertices that represent both resources \( \{r_1, ..., r_m\} \) and threads \( \{t_1, ..., t_n\} \).
  - A directed edge from a thread to a resource, \( t_i \rightarrow r_j \), indicates that \( t_i \) has requested that resource, but has not yet acquired it (Request Edge).
  - A directed edge from a resource to a thread \( r_j \rightarrow t_i \), indicates that the OS has allocated \( r_j \) to \( t_i \) (Assignment Edge).

- If the graph has no cycles, no deadlock exists.
- If the graph has a cycle, deadlock might exist.

What if there are multiple interchangeable instances of a resource?
  - Then a cycle indicates only that deadlock might exist.
  - If any instance of a resource involved in the cycle is held by a thread not in the cycle, then we can make progress when that resource is released.
Detect Deadlock and Then Correct It

- Scan the resource allocation graph for cycles, and then break the cycles.
- Different ways of breaking a cycle:
  - Kill all threads in the cycle.
  - Kill the threads one at a time, forcing them to give up resources.
  - Preempt resources one at a time rolling back the state of the thread holding the resource to the state it was in prior to getting the resource. This technique is common in database transactions.
- Detecting cycles takes $O(n^2)$ time, where $n = |T| + |R|$. When should we execute this algorithm?
  - Just before granting a resource, check if granting it would lead to a cycle? (Each request is then $O(n^2)$.)
  - Whenever a resource request can't be filled? (Each failed request is $O(n^2)$.)
  - On a regular schedule (hourly or ...)? (May take a long time to detect deadlock)
  - When CPU utilization drops below some threshold? (May take a long time to detect deadlock)
- What do current OS do?
  - Leave it to the programmer/application.

Deadlock Prevention

**Prevent deadlock**: ensure that at least one of the necessary conditions doesn't hold.

**1. Mutual Exclusion**: make resources sharable (but not all resources can be shared)

**2. Hold and Wait**:  
  - Guarantee that a thread cannot hold one resource when it requests another  
  - Make threads request all the resources they need at once and make the thread release all resources before requesting a new set.

**3. No Preemption**:  
  - If a thread requests a resource that cannot be immediately allocated to it, then the OS preempts (releases) all the resources that the thread is currently holding.  
  - Only when all of the resources are available, will the OS restart the thread.  
  - *Problem*: not all resources can be easily preempted, like printers.

**4. Circular wait**: impose an ordering (numbering) on the resources and request them in order.
Deadlock Prevention with Resource Reservation

• Threads provide advance information about the maximum resources they may need during execution.
• Define a sequence of threads \( \{t_1, \ldots, t_n\} \) as safe if for each \( t_i \), the resources that \( t_i \) can still request can be satisfied by the currently available resources plus the resources held by all \( t_j, j < i \).
• A safe state is a state in which there is a safe sequence for the threads.
• An unsafe state is not equivalent to deadlock, it just may lead to deadlock, since some threads might not actually use the maximum resources they have declared.
• Grant a resource to a thread is the new state is safe.
• If the new state is unsafe, the thread must wait even if the resource is currently available.
• This algorithm ensures no circular-wait condition exists.

Example

• Threads \( t_1 \), \( t_2 \), and \( t_3 \) are competing for 12 tape drives.
• Currently, 11 drives are allocated to the threads, leaving 1 available.
• The current state is safe (there exists a safe sequence, \( \{t_1, t_2, t_3\} \) where all threads may obtain their maximum number of resources without waiting)
  - \( t_1 \) can complete with the current resource allocation
  - \( t_2 \) can complete with its current resources, plus all of \( t_1 \)'s resources, and the unallocated tape drive.
• \( t_3 \) can complete with all its current resources, all of \( t_1 \) and \( t_2 \)'s resources, and the unallocated tape drive.

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>in use</th>
<th>could want</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>12</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Example (contd)

• If $t_3$ requests one more drive, then it must wait because allocating the drive would lead to an unsafe state.
• There are now 0 available drives, but each thread might need at least one more drive.

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>in use</th>
<th>could want</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$t_3$</td>
<td>12</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Deadlock Avoidance using Resource Allocation Graph

• Claim edges: an edge from a thread to a resource that may be requested in the future
• Satisfying a request results in converting a claim edge to an allocation edge and changing its direction.
• A cycle in this extended resource allocation graph indicates an unsafe state.
• If the allocation would result in an unsafe state, the allocation is denied even if the resource is available.
  – The claim edge is converted to a request edge and the thread waits.
• This solution does not work for multiple instances of the same resource.
Banker's Algorithm

- This algorithm handles multiple instances of the same resource.
- Force threads to provide advance information about what resources they may need for the duration of the execution.
- The resources requested may not exceed the total available in the system.
- The algorithm allocates resources to a requesting thread if the allocation leaves the system in a safe state.
- Otherwise, the thread must wait.

Preventing Deadlock with Banker's Algorithm

class ResourceManager {
    int n;        // # threads
    int m;      // # resources
    int avail[m], // # of available resources of each type
    max[n,m],   // # of each resource that each thread may want
    alloc[n,m],  //# of each resource that each thread is using
    need[n,m],  // # of resources that each thread might still request
}
Banker's Algorithm: Resource Allocation

```java
public void synchronized allocate (int request[m], int i) {
    // request contains the resources being requested
    // i is the thread making the request

    if (request > need[i])  // vector comparison
        error();  // Can't request more than you declared
    else while (request[i] > avail)
        wait();  // Insufficient resources available

    // enough resources exist to satisfy the requests
    // See if the request would lead to an unsafe state
    avail = avail - request;  // vector additions
    alloc[i] = alloc[i] + request;
    need[i] = need[i] - request;

    while ( !safeState () ) {
        // if this is an unsafe state, undo the allocation and wait
        <undo the changes to avail, alloc[i], and need[i]>
        wait();
        <redo the changes to avail, alloc[i], and need[i]>
    }
}
```

Banker's Algorithm: Safety Check

```java
private boolean safeState () {
    boolean work[m] = avail[m];  // accommodate all resources
    boolean finish[n] = false;   // none finished yet

    // find a process that can complete its work now
    while (find i such that finish[i] == false
        and need[i] <= work) { // vector operations
        work = work + alloc[i]
        finish[i] = true;
    }

    if (finish[i] == true for all i)
        return true;
    else
        return false;
}
```

- Worst case: requires $O(mn^2)$ operations to determine if the system is safe.
Example using Banker's Algorithm

System snapshot:

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Allocation</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>P₀</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P₃</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Example (contd)

- How many resources are there of type (A,B,C)?
- What is the contents of the Need matrix?
- Is the system in a safe state? Why?
Example: solutions

- How many resources of type (A,B,C)? \( (3,14,11) \)
  \[ \text{resources} = \text{total} + \text{avail} \]
- What is the contents of the need matrix?
  \[ \text{Need} = \text{Max} - \text{Allocation}. \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- Is the system in a safe state? Why?
  - Yes, because the processes can be executed in the sequence \( P_0, P_2, P_1, P_3 \), even if each process asks for its maximum number of resources when it executes.

Example (contd)

- If a request from process \( P_1 \) arrives for additional resources of \((0,5,2)\), can the Banker's algorithm grant the request immediately?
- What would be the new system state after the allocation?

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>( P_0 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( P_1 )</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( P_3 )</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is a sequence of process execution that satisfies the safety constraint?
Example: solutions

• If a request from process P₁ arrives for additional resources of (0, 5, 2), can the Banker's algorithm grant the request immediately? Show the system state, and other criteria.

Yes. Since
1. \((0, 5, 2) \leq (1, 5, 2)\), the Available resources, and
2. \((0, 5, 2) + (1, 0, 0) = (1, 5, 2) \leq (1, 7, 5)\), the maximum number P₁ can request.
3. The new system state after the allocation is:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P₀</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>P₁</td>
<td>1 5 2</td>
<td>1 7 5</td>
</tr>
<tr>
<td>P₂</td>
<td>1 3 5</td>
<td>2 3 5</td>
</tr>
<tr>
<td>P₃</td>
<td>0 6 3</td>
<td>0 6 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0</td>
</tr>
</tbody>
</table>

and the sequence P₀, P₁, P₂, P₃ satisfies the safety constraint.

Summary

• Deadlock:  situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
• Detection and recovery: recognize deadlock after it has occurred and break it.
• Avoidance: don't allocate a resource if it would introduce a cycle.
• Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds.
• Code concurrent programs very carefully. This only helps prevent deadlock over resources managed by the program, not OS resources.
• Ignore the possibility! (Most OSes use this option!!)