CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2020 L13
Deadlocks

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
• **Producer-consumer with bounded buffer**
  – Should the production and consumption rates be a perfect match?

• **Readers-Writers Problem**
  – Allow multiple readers to read at the same time (Q)
  – Only one single writer can access the shared data at the same time. No readers permitted when writer is accessing the data.

• **Monitor:**
  – Implements *mutual exclusion*: only one process may be active at a time
  – *Conditions* with associated queues where processes *wait* until *notified*
  – Our Monitor discussion is generic. See Self Exercise 5 for a Java example.
  – What about in C?
• Why not give each philosopher 2 chopsticks?
  – Nice and elegant solution. Widely used in Chinese restaurants. But takes all the fun away from the problem.

• What about this scenario?

Ph1: Hungary
Ph2: Thinking
Ph3: Eating
Ph4: Eating
Ph5: Thinking
Thus 3 and 4 hold 4 chopsticks, 1 thinks he can eat.
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable thus can be used to create a monitor
- Non-portable extensions include:
  - read-write locks
  - spinlocks
• A memory transaction is a sequence of read-write operations to memory that are performed atomically without the use of locks.

```c
void update()
{
    atomic{
        /* modify shared data*/
    }
}
```

May be implemented by hardware or software.
Alternative Approach: Open MP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.
CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Deadlocks

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
Chapter 8: Deadlocks

• System Model
• Deadlock Characterization
• Methods for Handling Deadlocks
  – Deadlock Prevention
  – Deadlock Avoidance resource-allocation
  – Deadlock Detection
  – Recovery from Deadlock
A Kansas Law

• Early 20th century Kansas Law
  – “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone”

• *Story of the two silly goats*: Aesop 6th cent BCE?
A contemporary example
System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  
  Resource may be CPU cycles, memory space, I/O devices, critical sections

- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set \( \{P_0, P_1, ..., P_n\} \) of waiting processes such that \( P_0 \) is waiting for a resource that is held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), ..., \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example
  - Dining Philosophers: each get the right chopstick first
  - we saw this example earlier

Let $s$ and $q$ be two semaphores initialized to 1

\[
\begin{align*}
P_0 & \quad & P_1 \\
\text{wait}(S); & \quad & \text{wait}(Q); \\
\text{wait}(Q); & \quad & \text{wait}(S); \\
& \quad & \ldots \\
& \quad & \text{signal}(S); \\
\text{signal}(Q); & \quad & \text{signal}(Q); \\
\text{signal}(S); & \quad & \\
\end{align*}
\]

$P_0$ executes $\text{wait}(s)$, $P_1$ executes $\text{wait}(Q)$

$P_0$ must wait till $P_1$ executes $\text{signal}(Q)$

$P_1$ must wait till $P_0$ executes $\text{signal}(S)$  Deadlock!
A set of vertices $V$ and a set of edges $E$.

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$
• Process

• Resource Type with 4 instances

• \( P_i \) requests instance of \( R_j \)

• \( P_i \) is holding an instance of \( R_j \)
Example of a Resource Allocation Graph

P1 **holds** an instance of R2, and is **requesting** R1..

Does a deadlock exist here?

P3 will eventually be done with R3, letting P2 use it.
Thus P2 will be eventually done, releasing R1. ...
Answer: No.

Observation: If the graph contains no cycles, then no process in the system is deadlocked.
If the graph does contain a cycle, then a deadlock **may** exist.
At this point, two minimal cycles exist in the system:

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes $P_1$, $P_2$, and $P_3$ are deadlocked.
Is there a deadlock?

*P4* will release its instance of resource type *R2*. That resource can then be allocated to *P3*, breaking the cycle. Thus there is no deadlock.

If a resource-allocation graph does not have a cycle, then the system is **not** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.
Basic Facts

• If graph contains no cycles ⇒ no deadlock

• If graph contains a cycle ⇒
  – if only one instance per resource type, then deadlock
  – if several instances per resource type, possibility of deadlock
Methods for Handling Deadlocks

• Ensure that the system will *never* enter a deadlock state:
  – Deadlock prevention
    • ensuring that at least one of the 4 conditions cannot hold
  – Deadlock avoidance
    • Dynamically examines the resource-allocation state to ensure that it will never enter an unsafe state, and thus there can never be a circular-wait condition

• Allow the system to enter a deadlock state
  – Detection: detect and then recover. Hope is that it happens rarely.

• Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
Methods for Handling Deadlocks

- **Deterministic**: Ensure that the system will *never* enter a deadlock state at any cost
- **Probabilistic view**: Hope it happens rarely. Handle if it happens: Allow the system to enter a deadlock state and then recover.
## Methods for Handling Deadlocks

<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource allocation policy</th>
<th>Scheme</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prevention</strong></td>
<td>Conservative, undercommits resources</td>
<td>Requesting all resources at once</td>
<td>Good for processes with a single burst of activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preemption</td>
<td>Good when preemption cost is small</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resource ordering</td>
<td>Compile time enforcement possible</td>
</tr>
<tr>
<td><strong>Avoidance</strong></td>
<td>midway</td>
<td>Find at least one safe path (dynamic)</td>
<td>Future max requirement must be known</td>
</tr>
<tr>
<td><strong>Detection</strong></td>
<td>Liberal</td>
<td>Invoked periodically</td>
<td>Preemption may be needed</td>
</tr>
</tbody>
</table>
Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all.

Advantages:
- Cheaper, rarely needed anyway
- Prevention, avoidance, detection and recovery
  - Need to run constantly

Disadvantages:
- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- Deterioration in system performance
  - Requires restart

To be fair to the ostriches, let me say that ...
For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

- **Mutual exclusion**: only one process at a time can use a resource.
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait**: there exists a set \(\{P_0, P_1, ..., P_n\}\) of waiting processes that are circularly waiting.
For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Restrain the ways request can be made:

- **Limit Mutual Exclusion** —
  - not required for sharable resources (e.g., read-only files)
  - (Mutual Exclusion must hold for non-sharable resources)
• **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  
  1. Require process to request and be allocated all its resources before it begins execution
Deadlock Prevention: Limit Hold and Wait

• **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  2. Allow a process to request resources when it is holding none.
     Ex: Copy data from DVD, sort file, and print
     – First request DVD and disk file
     – Then request file and printer

• Disadvantage: starvation possible
• **Limit No Preemption** –
  
  – If a process that is holding some resources, requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  
  – *Preempted resources* are added to the list of resources for which the process is waiting
  
  – Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
Deadlock Prevention: Limit Circular Wait

- **Limit Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

- Assign each resource a unique number
  - Disk drive: 1
  - Printer: 2 ...
  - Request resources in increasing order
    - *Example soon*
Midterm

Tuesday 3/10/2020
• Closed book, Approved calculators permitted
• Special seating
• IDs needed

• Example problems posted on Piazza. They will be discussed as apart of the review.
Dining philosophers problem: Necessary conditions for deadlock

- Mutual exclusion
  - 2 philosophers cannot share the same chopstick

- Hold-and-wait
  - A philosopher picks up one chopstick at a time
  - Will not let go of the first while it waits for the second one

- No preemption
  - A philosopher does not snatch chopsticks held by some other philosopher

- Circular wait
  - Could happen if each philosopher picks chopstick with the same hand first
Deadlock Example

/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
pthread_mutex_lock(&second_mutex);
/** * Do some work */
pthread_mutex_unlock(&second_mutex);
pthread_mutex_unlock(&first_mutex);
pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
pthread_mutex_lock(&first_mutex);
/** * Do some work */
pthread_mutex_unlock(&first_mutex);
pthread_mutex_unlock(&second_mutex);
pthread_exit(0);
}

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex.

Solution: Lock-order verifier “Witness” records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.
Deadlock may happen *even* with Lock Ordering

```java
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Ex: Transactions 1 and 2 execute concurrently.

Transaction 1 transfers $25 from account A to account B, and
Transaction 2 transfers $50 from account B to account A.

Deadlock is possible, even with lock ordering.

Lock ordering: First *from* lock, then *to* lock
Deadlock Avoidance

Manage resource allocation to ensure the system never enters an unsafe state.
Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes
Deadlock Avoidance

• Require additional information about how resources are to be requested
• Knowledge about sequence of requests and releases for processes
  – Allows us to decide if resource allocation could cause a future deadlock
    • Process P: Tape drive, then printer
    • Process Q: Printer, then tape drive
Deadlock Avoidance: Handling resource requests

• For each resource request:
  – Decide whether or not process should wait
    • To avoid possible future deadlock

• Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. *Future requests and releases of each process*
Avoidance: amount and type of information needed

• **Resource allocation state**
  – Number of available and allocated resources
  – Maximum demands of processes

• *Dynamically* examine resource allocation state
  – Ensure circular-wait cannot exist

• Simplest model:
  – Declare maximum number of resources for each type
  – Use information to avoid deadlock
Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

System is in **safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes such that

- for each $P_i$, the resources that $P_i$ can still request can be satisfied by
  - currently available resources +
  - resources held by all the $P_j$, with $j < i$
  - That is
    - If $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished and released resources
    - When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on

- If no such sequence exists: system state is **unsafe**
Deadlock avoidance: Safe states

• If the system can:
  – Allocate resources to each process in some order
    • Up to the maximum for the process
  – Still avoid deadlock
  – Then it is in a safe state

• A system is safe ONLY IF there is a safe sequence

• A safe state is not a deadlocked state
  – Deadlocked state is an unsafe state
  – Not all unsafe states are deadlock
Safe, Unsafe, Deadlock State

Examples of safe and unsafe states in next 3 slides
Example A: Assume 12 Units in the system

<table>
<thead>
<tr>
<th></th>
<th>Max need</th>
<th>Current holding</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

At time T0 (shown):
9 units allocated
3 (12-9) units available

A unit could be a drive, a block of memory etc.

- **Is the system at time T0 in a safe state?**
  - Try sequence \(<P1, P0, P2>\)
  - P1 can be given 2 units
  - When P1 releases its resources; there are now 5 available units
  - P0 uses 5 and subsequently releases them (10 available now)
  - P2 can then proceed.

- **Thus \(<P1, P0, P2>\) is a safe sequence, and at T0 system was in a safe state**
Example A: Assume 12 Units in the system (timing)

Is the state at T0 safe?

<table>
<thead>
<tr>
<th>Max need</th>
<th>Current holding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T0</td>
</tr>
<tr>
<td></td>
<td>T1</td>
</tr>
<tr>
<td></td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T3</td>
</tr>
<tr>
<td></td>
<td>T4</td>
</tr>
<tr>
<td></td>
<td>T5</td>
</tr>
<tr>
<td>av</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>10 done</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4 done</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>9 done</td>
</tr>
</tbody>
</table>

Thus the state at T0 is safe.