CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 24

Deadlocks

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

FAQ

- Resource allocation graph: what are resources?
  Memory space, IO devices, files, etc.
- Deadlock definition: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Deadlock Characterization: 4 *necessary* conditions
- Edges: request P -> R assignment R -> P
- If a cycle is present:
  - If each resource type has exactly one instance: Deadlock **has** occurred
  - If each resource type has multiple instances: A deadlock **may** have occurred

Notes

- Project topic, team, coordinator:
  - submission due today
  - Soon: select a group name and join the name
- PA4 Due 10/27. Early reward due 10/20
  - Note unlimited number of processes (use ArrayList). Also consider Scanner.
- Java Monitor example (Dining Philosophers) available as a self-exercise. See Piazza post.

Chapter 7: Deadlocks

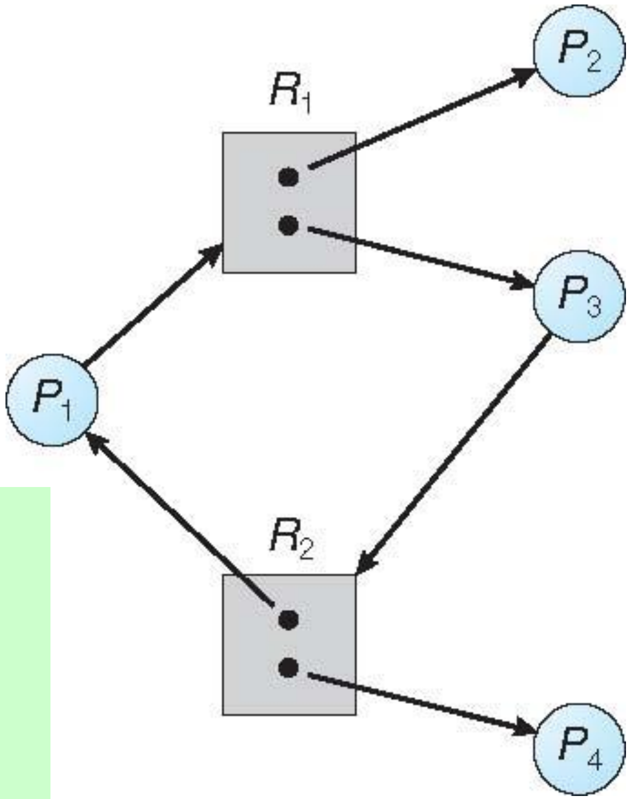
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance resource-allocation
  - Deadlock Detection
  - Recovery from Deadlock

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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Graph With A Cycle But No Deadlock



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

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 $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - 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 there can never be a circular-wait condition
- Allow the system to enter a deadlock state
  - Detect and then recover
- 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
- Handle if it happens: Allow the system to enter a deadlock state and then recover
- Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all .
  - My be acceptable if it happens only rarely (Probabilistic view)

Ostrich algorithm

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

Deadlock Prevention: **Limit Mutual Exclusion**

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

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
    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

Colorado State University

Deadlock Prevention: Limit No Preemption

- **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

Dining philosophers problem: Necessary conditions for deadlock

Relax conditions to avoid 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 Example with Lock Ordering

```
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);
}
```

Lock ordering:
First from lock, then to lock

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.

Deadlock Avoidance

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

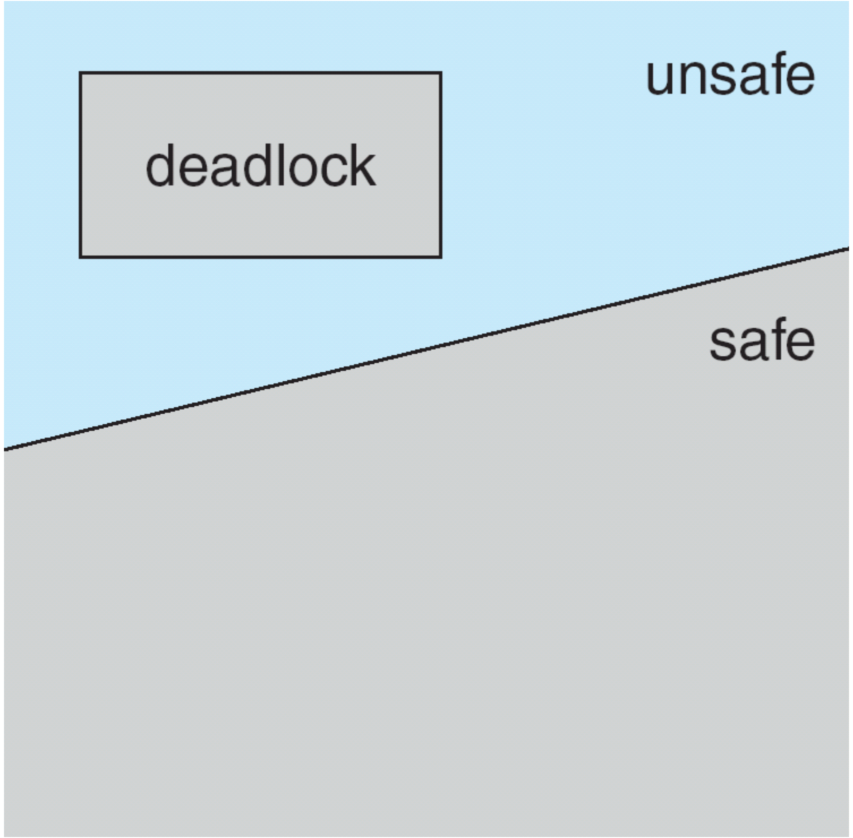
- for each P_i, the resources that P_i can still request can be satisfied by
  - currently available resources +
  - resources held by all the P_j, with $j < i$
  - That is
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished and released resources
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If no such sequence exists: system state is unsafe

Colorado State University

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



A unsafe state may lead to deadlock

Colorado State University