CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 23

Deadlocks

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Chapter 7: Deadlocks

Objectives:

- Description of deadlocks,
  - prevent sets of concurrent processes from completing their tasks
- Different methods for
  - preventing or
  - avoiding deadlocks

FAQ

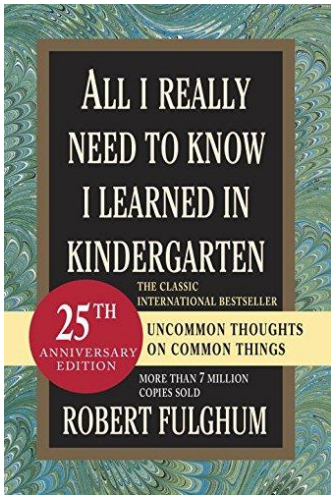
- Mutex vs condition variables: waiting for mutex vs condition variables
- Does a monitor always have two condition variables x and y? Not necessarily
- Why not always use transactional memory that makes a whole transaction atomic? May require CPU hardware support
- Why not look at details of transactional memory, Open MP etc.? Too many details. Likely coverd in other classes.
- Spinlocks efficient only for short waiting times vs Sleep waiting if needs to wait for a longer time, for example for a thread not currently running
- Single resource allocation vs locks: waiting with priority

Chapter 7: Deadlocks

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

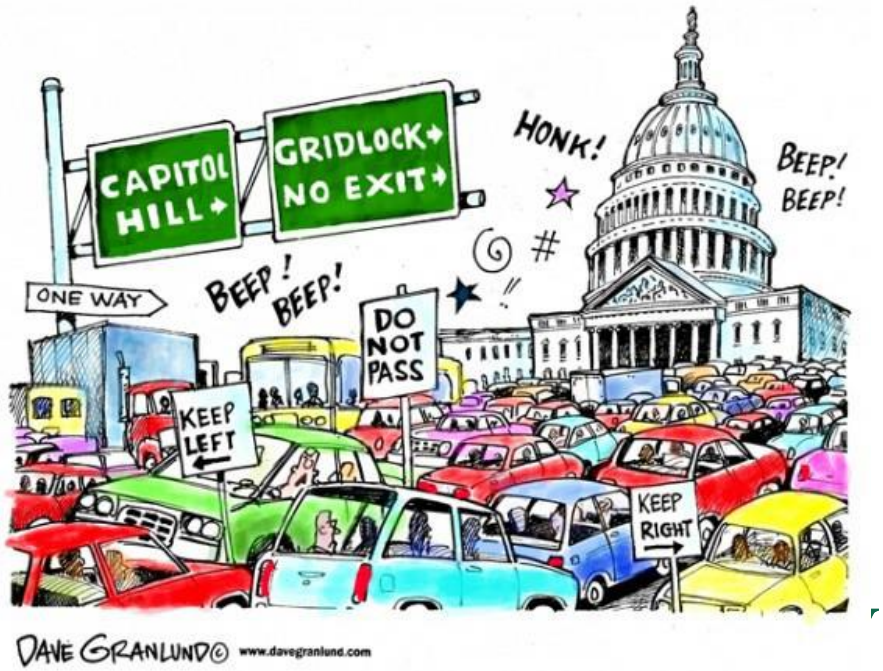
A Kansas Law

- 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?*



Colorado State University

A Gridlock


CAPITOL HILL
GRIDLOCK
NO EXIT
HONK!
BEEP! BEEP!
ONE WAY
BEEP! BEEP!
DO NOT PASS
KEEP LEFT
KEEP RIGHT
DAVE GRANLUND© www.davegranlund.com

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- 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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example
  - Dining Philosophers: each get the right chopstick first
  - box in text page 318 for mutex deadlock (we saw something similar earlier)

In this example, thread one attempts to acquire the mutex locks in the order (1) first mutex, (2) second mutex, while thread two attempts to acquire the mutex locks in the order (1) second mutex, (2) first mutex. Deadlock is possible if thread one acquires first mutex while thread two acquires second mutex.

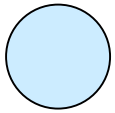
Resource-Allocation Graph

A set of vertices V and a set of edges E.

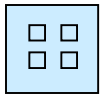
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

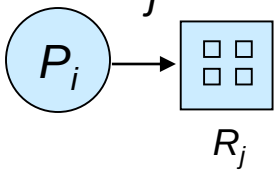
- Process



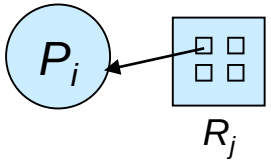
- Resource Type with 4 instances



- P_i requests instance of R_j



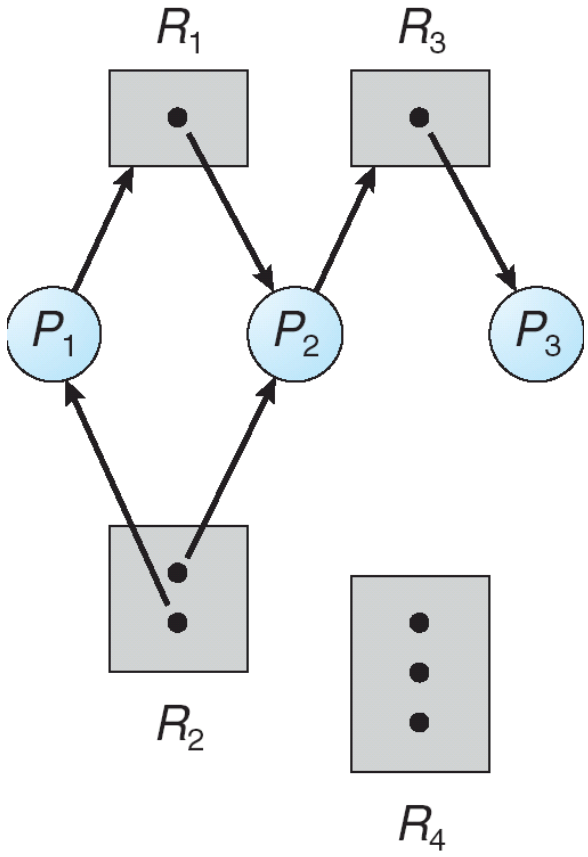
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

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

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.

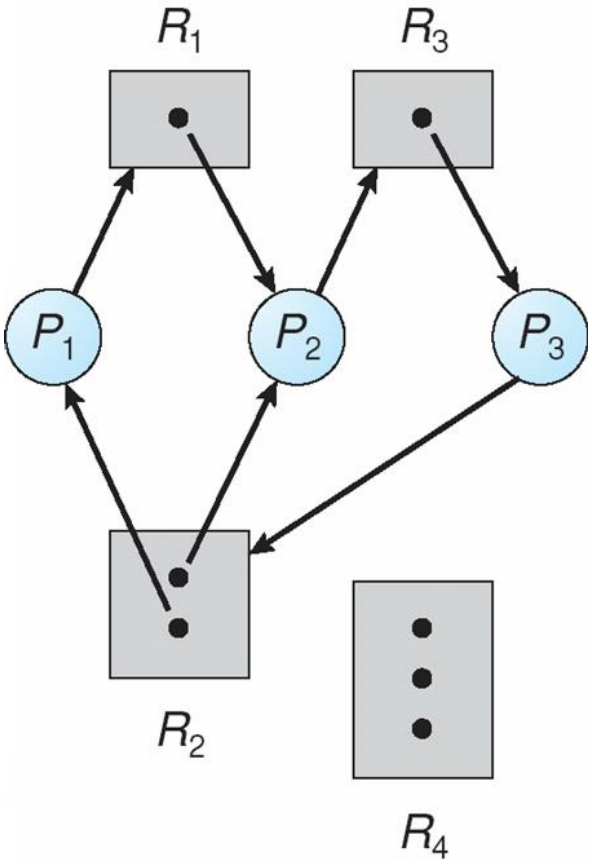


Does a deadlock exist here?

P3 will eventually be done with R3, letting P2 use it.

Thus P2 will be eventually done, releasing R1. …

Resource Allocation Graph With A Deadlock



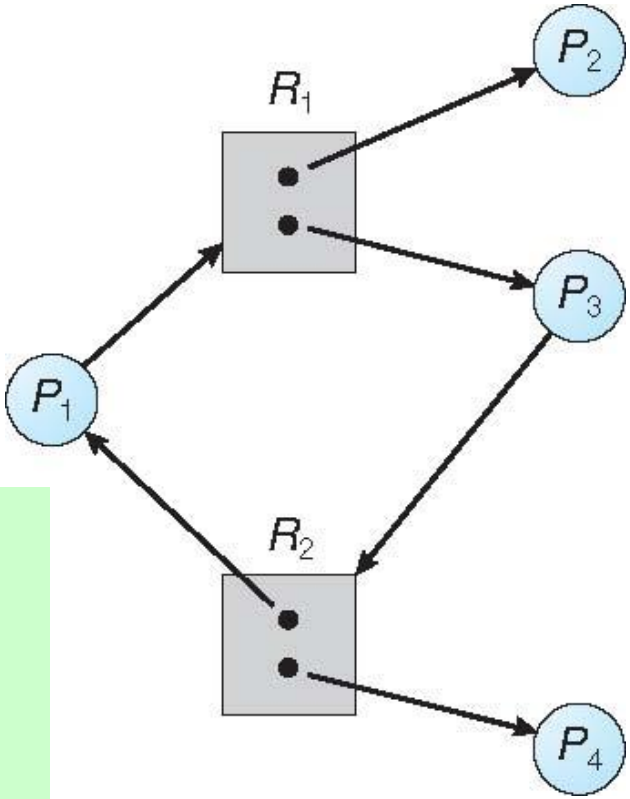
At this point, two minimal cycles exist in the system:

P1→ R1→ P2→ R3→ P3→ R2→ P1

P2→ R3→ P3→ R2→ P2

Processes *P1, P2,* and *P3* are deadlocked.

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 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)

Deadlock Prevention

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:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

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

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 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);
}
```

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.