

- Deadlocks
 - Silberschatz Ch. 7
 - also Priority Inversion Problem

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
 - System has 2 disk drives.
 - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
 - semaphores A and B , initialized to 1

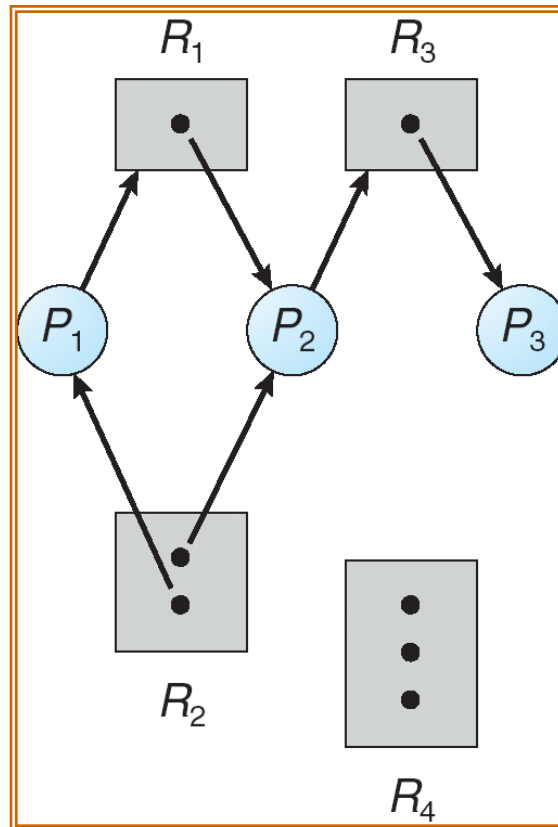
P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Deadlock Characterization

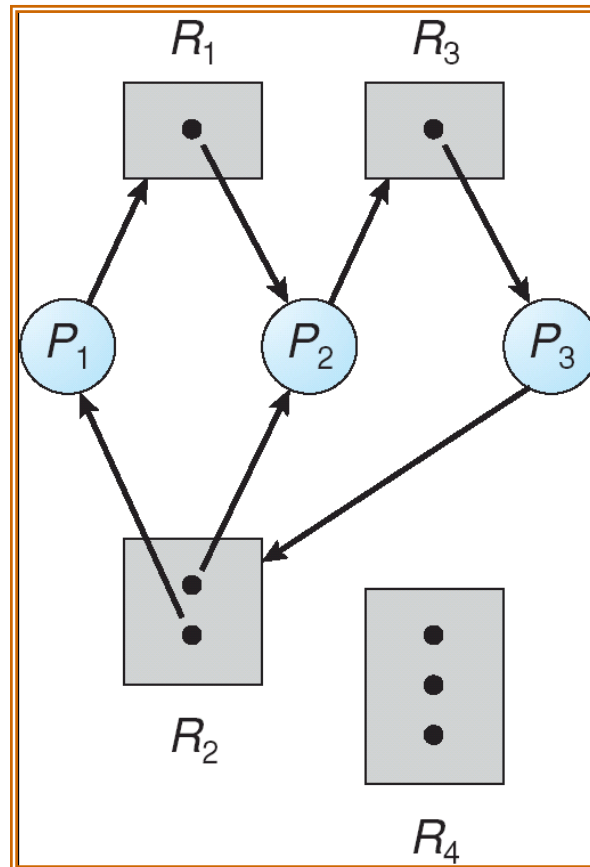
Deadlock can arise if four conditions hold simultaneously.

- **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 $\{P_0, P_1, \dots, P_{n-1}\}$ 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_0 , and P_0 is waiting for a resource that is held by P_0 .

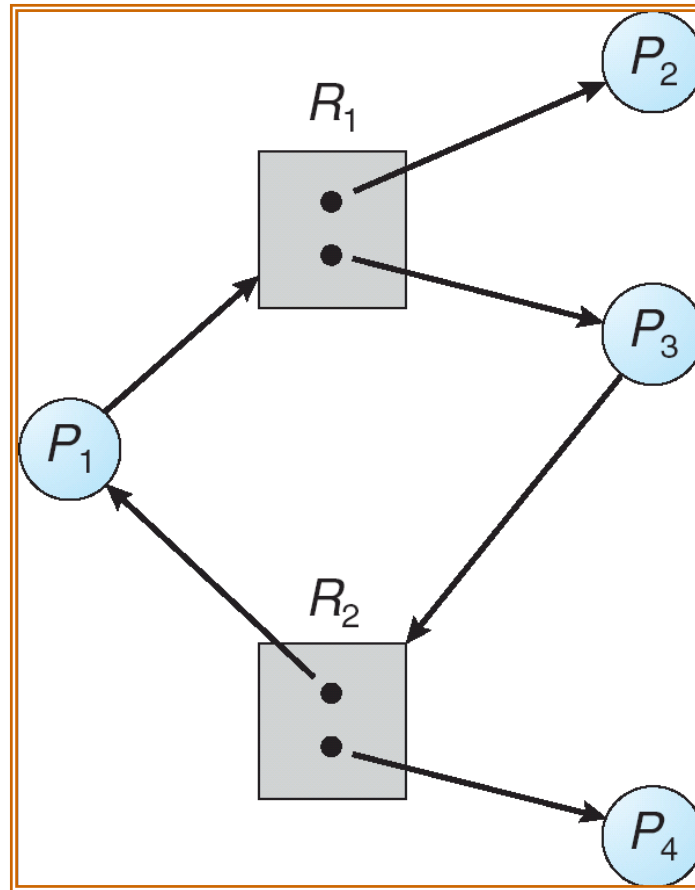
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



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.
- 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.
 - the “Ostrich” Algorithm

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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.
 - 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.
 - how do you preempt a robot movement?
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

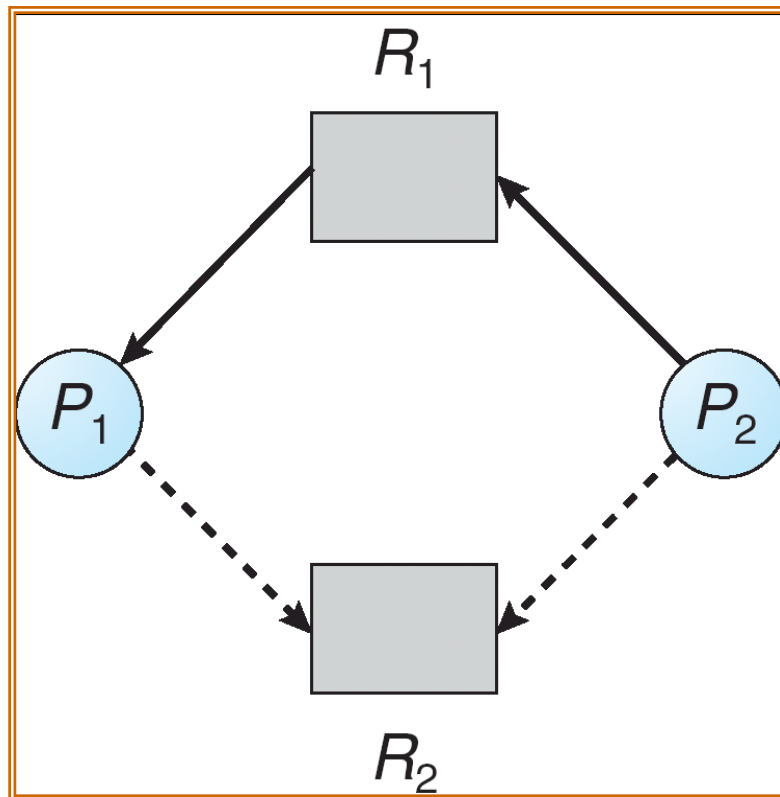
Deadlock Avoidance

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

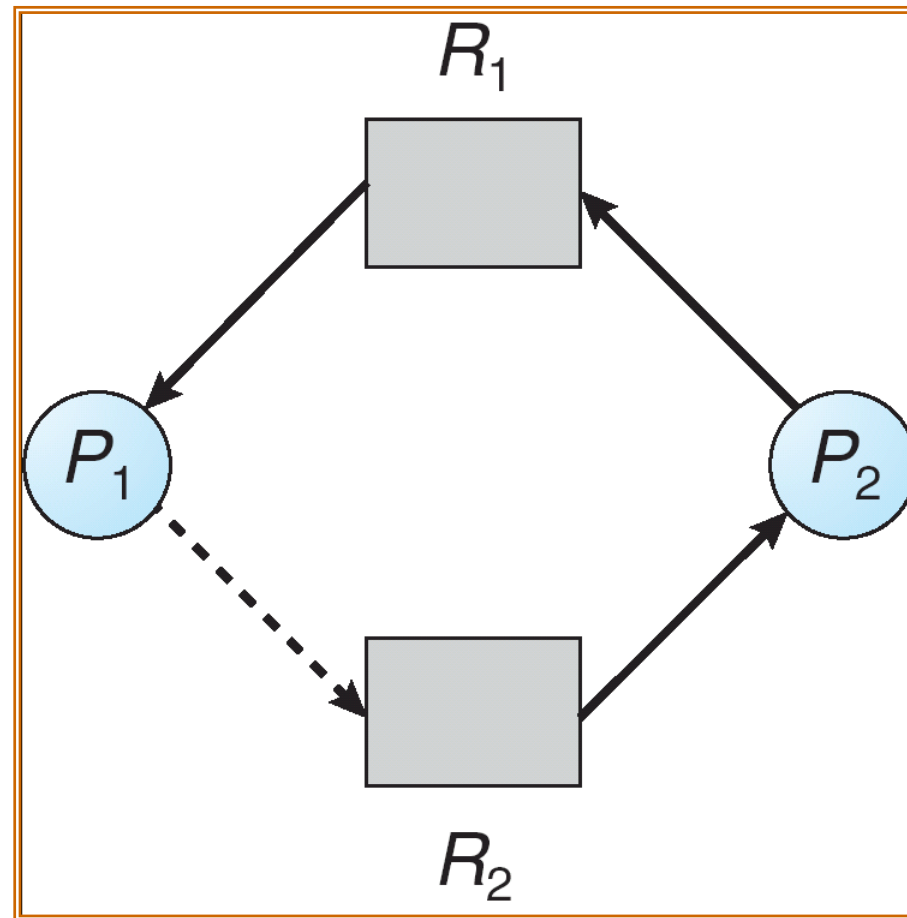
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation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

Resource-Allocation Graph

- single resource instance version of Bankers Algorithm (Dijkstra)
- multi-instance version similar objective – keep system in safe state



Unsafe State In Resource-Allocation Graph



Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
 - again, consider factors
- Rollback – return to some safe state, restart process for that state.
 - need logging facility/mechanism
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Livelock

- Different type of starvation problem:
 - Processes continue executing (as opposed to waiting, like in deadlock), but without making progress (or at least some of them don't).
 - E.g., High priority thread accepting requests runs too frequently, so low priority never gets a change to do rest of request processing.
 - E.g., processes call each other in a infinite loop, none makes progress, but they are both executing.

Priority Inversion

- Locking can have complex interactions with priorities
- Example:
 - low priority thread A locks mutex M
 - medium priority thread B runs for long
 - high priority thread C waits on M
- Problem: B will complete before C will ever get to run (because B has priority over A, and C waits on a resource that A has)
 - even worse: C may have a time deadline to complete, deadline may expire and C may determine system problem, deadlock, etc... and restart...

- Solution:
 - priority inheritance: once C blocks on M, A (which has M) gets C's priority (or higher)

What really happened on Mars?

- Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).
- The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.
- The spacecraft also contained a communications task that ran with medium priority.
- Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.