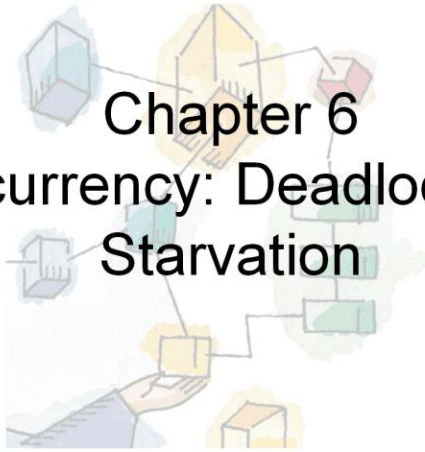


*Operating Systems:
Internals and Design Principles, 6/E*
William Stallings



Chapter 6 Concurrency: Deadlock and Starvation

Dave Bremer
Otago Polytechnic, N.Z.
©2008, Prentice Hall

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.

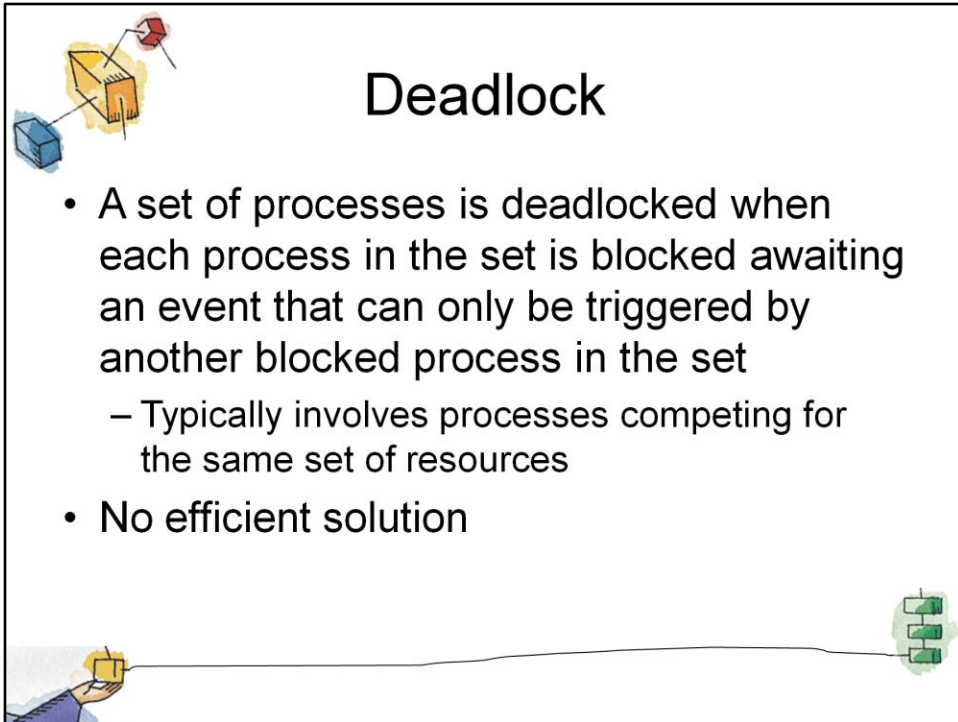
Roadmap



→ Principals of Deadlock

- Deadlock prevention
- Deadlock Avoidance
- Deadlock detection
- An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





Deadlock

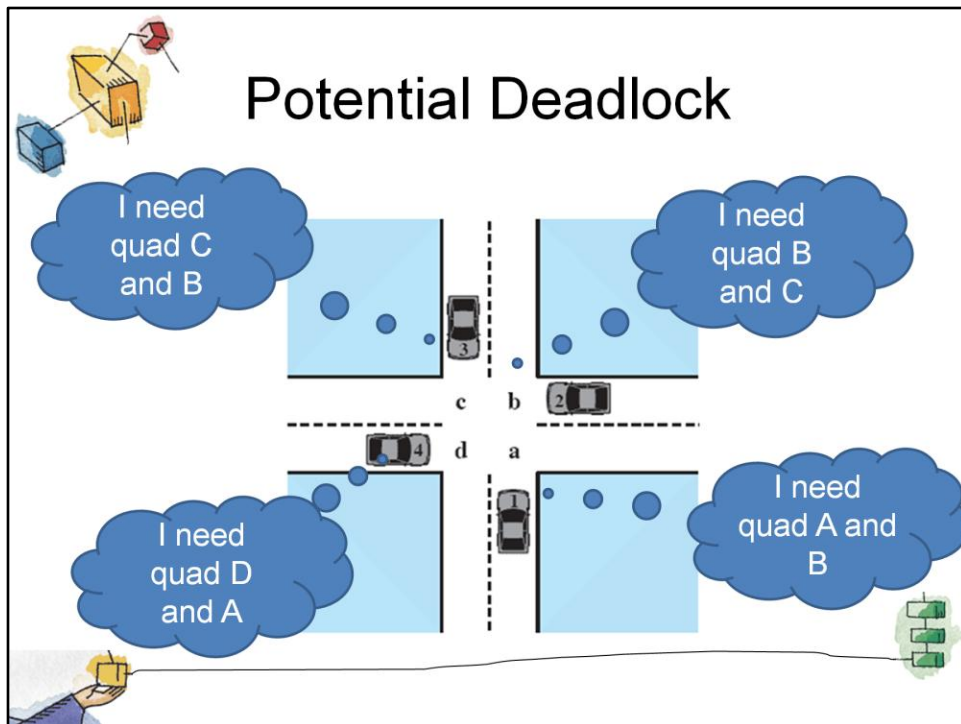
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
 - Typically involves processes competing for the same set of resources
- No efficient solution

A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

typically processes are waiting the freeing up of some requested resource.

Deadlock is permanent because none of the events is ever triggered.

Unlike other problems in concurrent process management, there is no efficient solution in the general case.



Animated Slide

Click 1 Cars approach intersection

Then Cars announce their resource needs

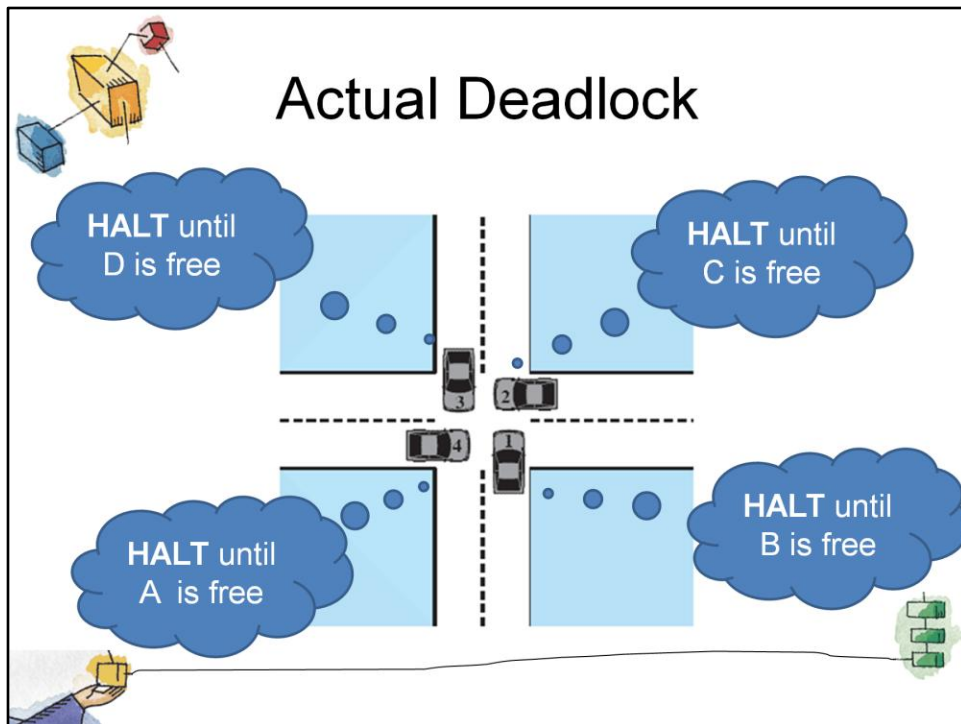
All deadlocks involve conflicting needs for resources by two or more processes. A common example is the traffic deadlock.

The typical rule of the road in the United States is that a car at a four-way stop should defer to a car immediately to its right.

This rule works if there are only two or three cars at the intersection.

If all four cars arrive at about the same time, each will refrain from entering the intersection, this causes a **potential deadlock**.

- The deadlock is only potential, not actual, because the necessary resources are available for any of the cars to proceed.
- If one car eventually does proceed, it can do so.



Animated Slide

Click 1 Cars move to deadlock

Then Cars announce their resource need

But if all four cars ignore the rules and proceed (cautiously) into the intersection at the same time, then **each car seizes one resource** (one quadrant) but cannot proceed because the required second resource has already been seized by another car.

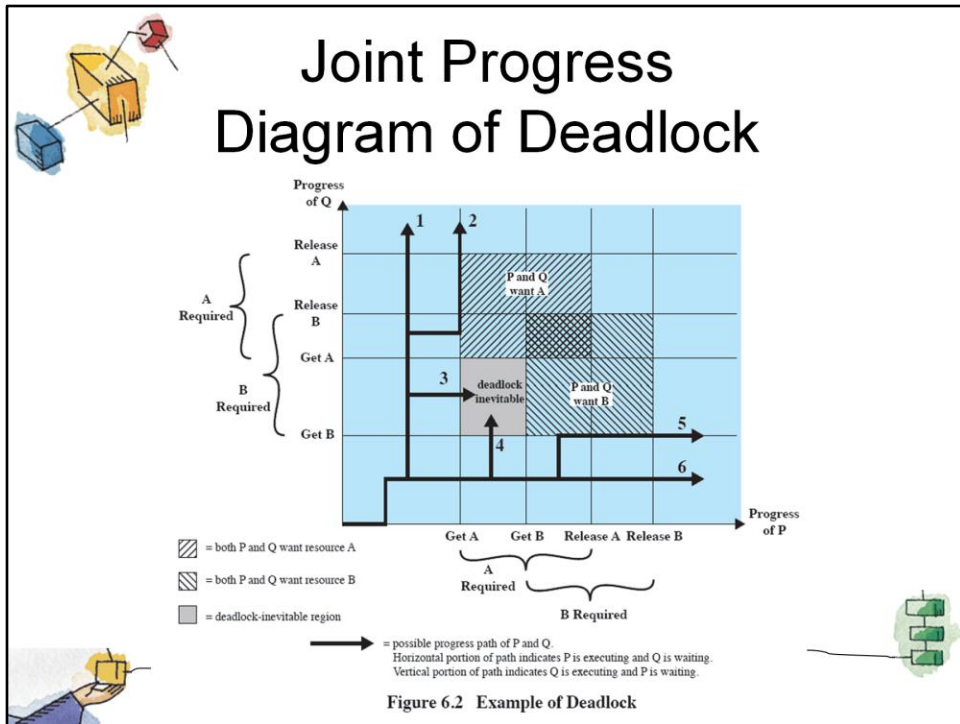
This is an actual deadlock.

Two Processes P and Q

- Lets look at this with two processes P and Q
- Each needing exclusive access to a resource A and B for a period of time

Process P	Process Q
...	...
Get A	Get B
...	...
Get B	Get A
...	...
Release A	Release B
...	...
Release B	Release A
...	...

Each process needs exclusive use of both resources for a certain period of time.
 Two processes, P and Q, have the following general form:



This illustrates the progress of two processes competing for two resources.

- The x-axis represents progress in the execution of P
- The y-axis represents progress in the execution of Q.

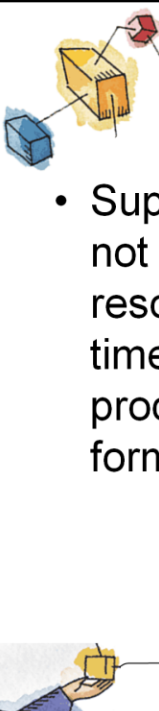
Six different paths of execution are shown.

The gray-shaded area can be referred to as a **fatal region**, applies to paths 3 and 4.

- If an execution path enters this fatal region, then deadlock is inevitable.

Note that the existence of a fatal region depends on the logic of the two processes.

- However, deadlock is only inevitable if the joint progress of the two processes creates a path that enters the fatal region.

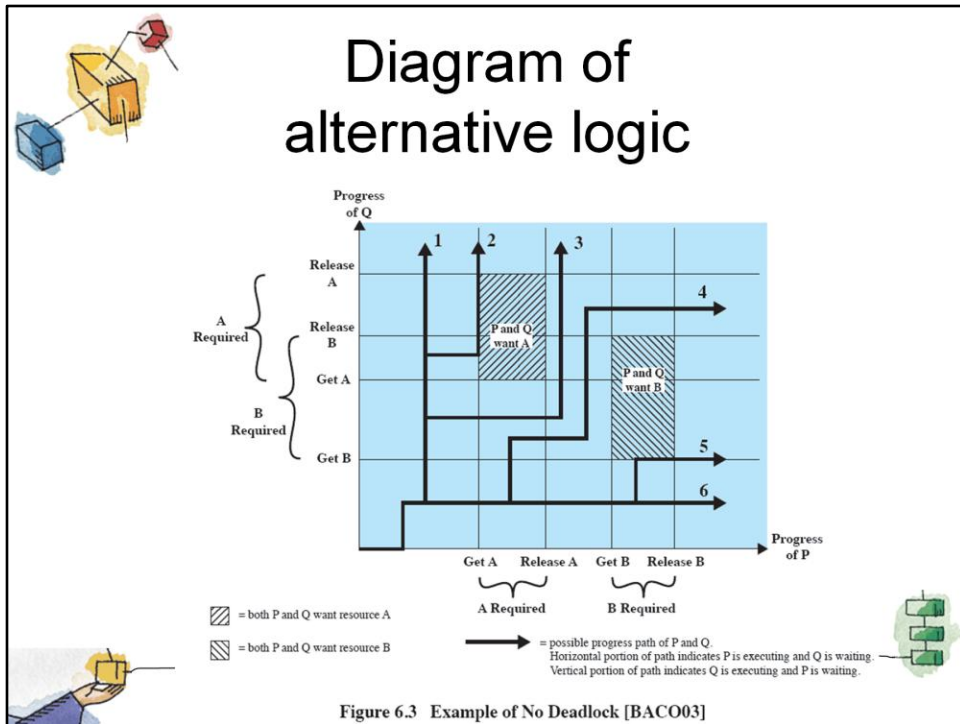


Alternative logic

- Suppose that P does not need both resources at the same time so that the two processes have this form

Process P	Process Q
...	...
Get A	Get B
...	...
Release A	Get A
...	...
Get B	Release B
...	...
Release B	Release A
...	...

Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application.



Discuss this diagram with students until they see that regardless of the relative timing of the two processes, deadlock cannot occur.



Resource Categories

Two general categories of resources:

- Reusable
 - can be safely used by only one process at a time and ***is not depleted*** by that use.
- Consumable
 - one that can be created (***produced***) and destroyed (***consumed***).





Reusable Resources

- Such as:
 - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other





Example of Reuse Deadlock

- Consider two processes that compete for exclusive access to a disk file D and a tape drive T.
- Deadlock occurs if each process holds one resource and requests the other.





Reusable Resources Example

Process P

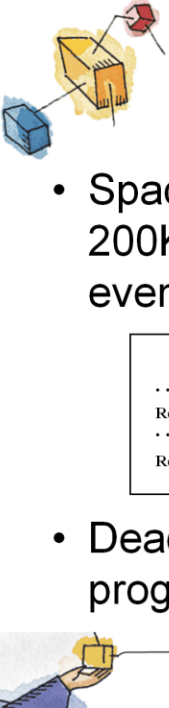
Process Q

Step	Action
P ₀	Request (D)
P ₁	Lock (D)
P ₂	Request (T)
P ₃	Lock (T)
P ₄	Perform function
P ₅	Unlock (D)
P ₆	Unlock (T)

Step	Action
q ₀	Request (T)
q ₁	Lock (T)
q ₂	Request (D)
q ₃	Lock (D)
q ₄	Perform function
q ₅	Unlock (T)
q ₆	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources






Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
...	...
Request 80 Kbytes;	Request 70 Kbytes;
...	...
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request



If the amount of memory to be requested is not known ahead of time, it is difficult to deal with this type of deadlock by means of system design constraints.


The best way to deal with this particular problem is, in effect, to eliminate the possibility by using virtual memory, which is discussed later (ch 8)



Consumable Resources

- Such as Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock






Example of Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

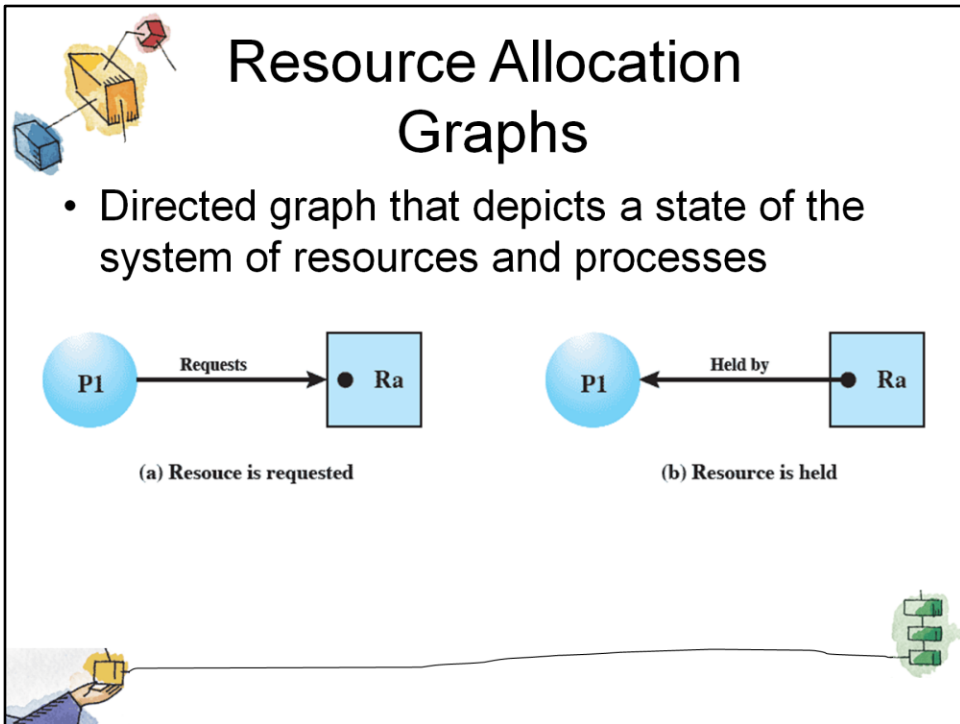


Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

A design error is the cause of the deadlock.

Such errors may be quite subtle and difficult to detect.


Furthermore, it may take a rare combination of events to cause the deadlock; thus a program could be in use for a considerable period of time, even years, before the deadlock actually occurs.



A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.



Within a resource node, a dot is shown for each instance of that resource.

A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted

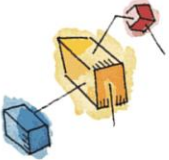


Conditions for *possible* Deadlock

- Mutual exclusion
 - Only one process may use a resource at a time
- Hold-and-wait
 - A process may hold allocated resources while awaiting assignment of others
- No pre-emption
 - No resource can be forcibly removed from a process holding it





All three must be present for deadlock to occur.



Actual Deadlock Requires ...

All previous 3 conditions plus:

- Circular wait
 - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

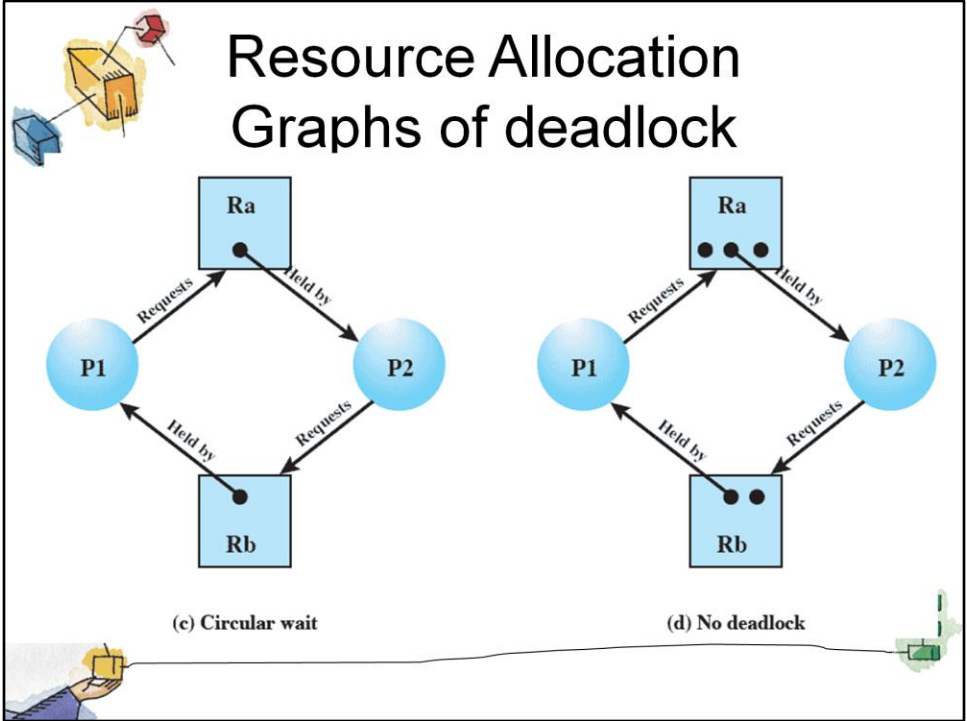


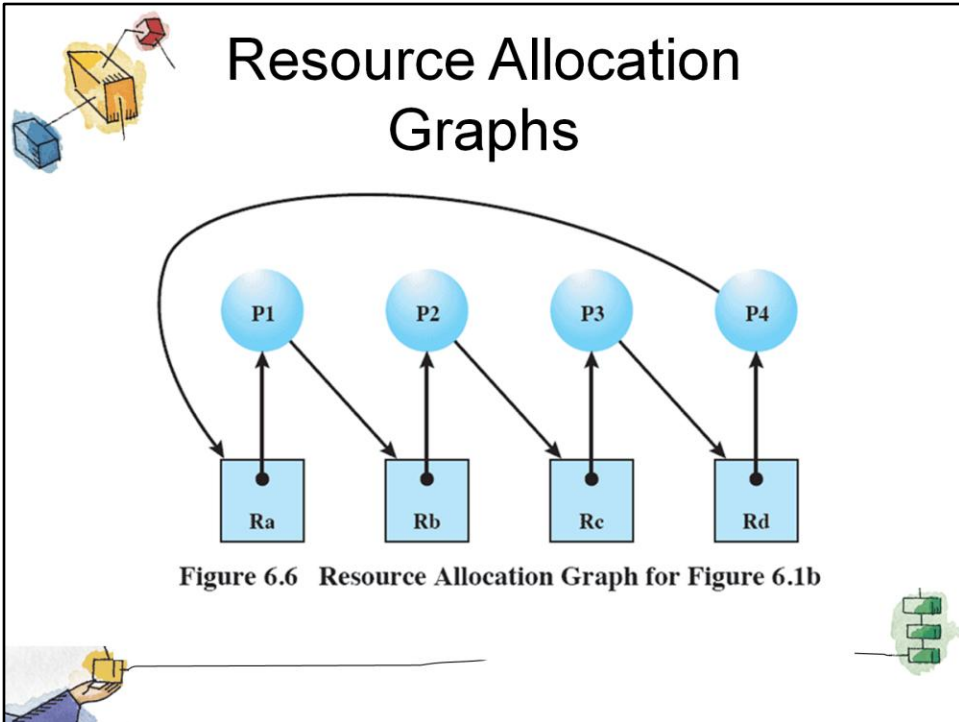
This is actually a potential consequence of the first three.

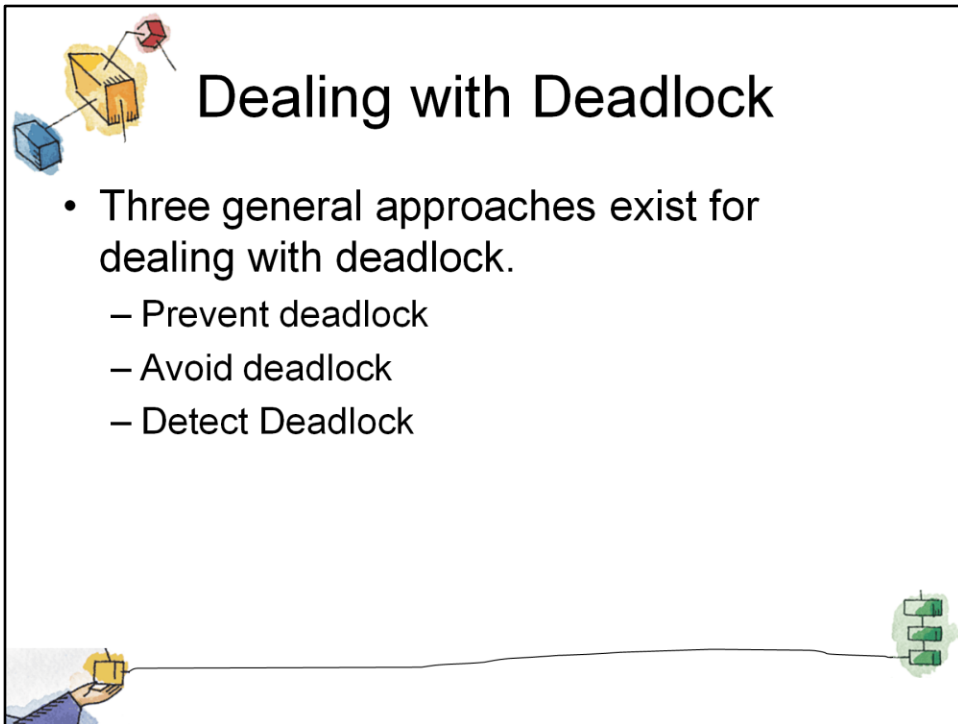
Given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait.

The unresolvable circular wait is in fact the definition of deadlock.

- The circular wait listed as condition 4 is unresolvable because the first three conditions hold.
- Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock.







Dealing with Deadlock

- Three general approaches exist for dealing with deadlock.
 - Prevent deadlock
 - Avoid deadlock
 - Detect Deadlock

Three general approaches exist for dealing with deadlock.

prevent deadlock

adopt a policy that eliminates one of the conditions (conditions 1 through 4).

avoid deadlock

by making the appropriate dynamic choices based on the current state of resource allocation.

detect the presence of deadlock

(conditions 1 through 4 hold) and take action to recover.

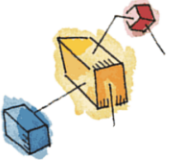
We discuss each of these approaches in turn.



Roadmap



- Principals of Deadlock
 - ➔ Deadlock prevention
 - Deadlock Avoidance
 - Deadlock detection
 - An Integrated deadlock strategy
 - Dining Philosophers Problem
 - Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.
- Two main methods
 - Indirect – prevent all three of the necessary conditions occurring at once
 - Direct – prevent circular waits




Deadlock prevention is strategy simply to design a system in such a way that the possibility of deadlock is excluded.

We can view deadlock prevention methods as falling into two classes.


- **indirect** method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3).
- **direct** method of deadlock prevention is to prevent the occurrence of a circular wait (item 4).

We now examine techniques related to each of the four conditions.



Deadlock Prevention Conditions 1 & 2

- Mutual Exclusion
 - Must be supported by the OS
- Hold and Wait
 - Require a process request all of its required resources at one time



Mutual Exclusion

The first of the four listed conditions cannot be disallowed (in general).

- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes.
- Even in this case, deadlock can occur if more than one process requires write permission.

Hold an Wait

Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.


This approach is inefficient in two ways.

- 1) a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources.
- 2) resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

Another problem is that a process may not know in advance all of the resources that it will require.



There is also the practical problem created by the use of modular programming or a multithreaded structure for an application.

An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.



Deadlock Prevention Conditions 3 & 4

- **No Preemption**
 - Process must release resource and request again
 - OS may preempt a process to require it releases its resources
- **Circular Wait**
 - Define a linear ordering of resource types



No Preemption

can be prevented in several ways.

- 1) If a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.
- 2) If a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.

This latter scheme would prevent deadlock only if no two processes possessed the same priority.

This approach is practical only with resources whose state can be easily saved and restored later, as is the case with a processor.

Circular Wait

Can be prevented by defining a linear ordering of resource types.


As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.



Roadmap



- Principals of Deadlock
 - Deadlock prevention
 - Deadlock Avoidance
 - Deadlock detection
 - An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests



Deadlock avoidance allows the three necessary conditions

but makes judicious choices to assure that the deadlock point is never reached.

Avoidance allows more concurrency than prevention.

With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.

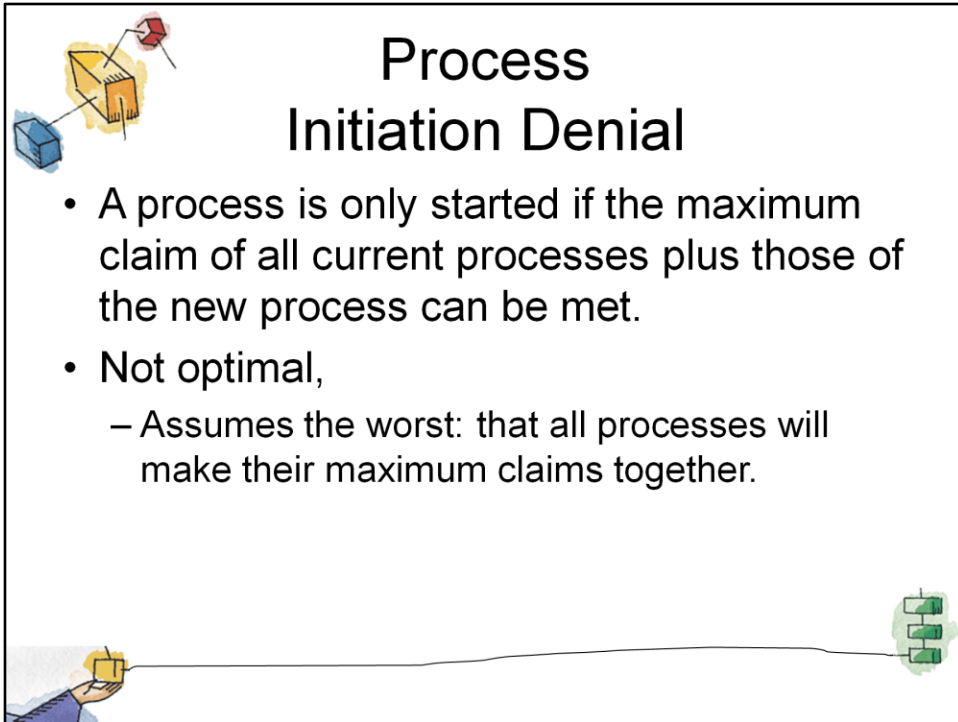
Deadlock avoidance requires knowledge of future process resource requests.



Two Approaches to Deadlock Avoidance

- **Process Initiation Denial**
 - Do not start a process if its demands might lead to deadlock
- **Resource Allocation Denial**
 - Do not grant an incremental resource request to a process if this allocation might lead to deadlock



A hand holding a laptop is shown at the bottom left. A network diagram with a yellow box, a blue box, and a red box is in the top left. A green server rack icon is in the bottom right. A thin line connects the laptop to the server rack.

Process Initiation Denial

- A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal,
 - Assumes the worst: that all processes will make their maximum claims together.

A process is only started if the maximum claim of all current processes plus those of the new process can be met.



This strategy is hardly optimal, because it assumes the worst:

that all processes will make their maximum claims together.



Resource Allocation Denial



- Referred to as the banker's algorithm
 - A strategy of resource allocation denial
 - Consider a system with fixed number of resources
 - **State** of the system is the current allocation of resources to process
 - **Safe state** is where there is at least one sequence that does not result in deadlock
 - **Unsafe state** is a state that is not safe
- 
- 

Movie button goes to <http://gaia.ecs.csus.edu/~zhangd/oscal/Banker/Banker.html>



Determination of Safe State

- A system consisting of four processes and three resources.
- Allocations are made to processors
- ***Is this a safe state?***

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
Resource vector R	9	3	6

	R1	R2	R3
Available vector V	0	1	1

(a) Initial state

Animation: Callouts explain resource parts of figure

This figure shows the state of a system consisting of four processes and three resources.

Total amount of resources

- R1 = 9
- R2 = 3
- R3 = 6

In the current state allocations have been made to the four processes, leaving available

- 1 unit of R2
- 1 unit of R3

Is this a safe state?

To answer this question, we ask an intermediate question:

- Can any of the four processes be run to completion with the resources available?
- That is, can the difference between the maximum requirement and current allocation for any process be met with the available resources?



Process i

- $C_{ij} - A_{ij} \leq V_j$, for all j
- This is not possible for P1,
 - which has only 1 unit of R1 and requires 2 more units of R1, 2 units of R2, and 2 units of R3.
- If we assign one unit of R3 to process P2,
 - Then P2 has its maximum required resources allocated and can run to completion and return resources to 'available' pool





After P2 runs to completion

- Can any of the remaining processes can be completed?

Note P2 is completed

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

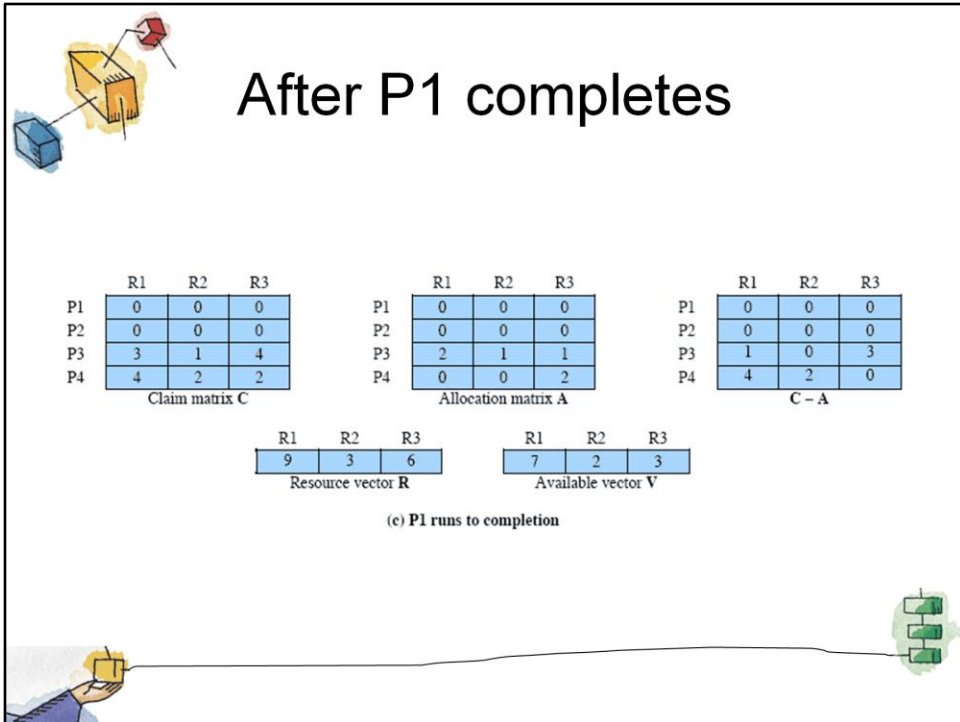
Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

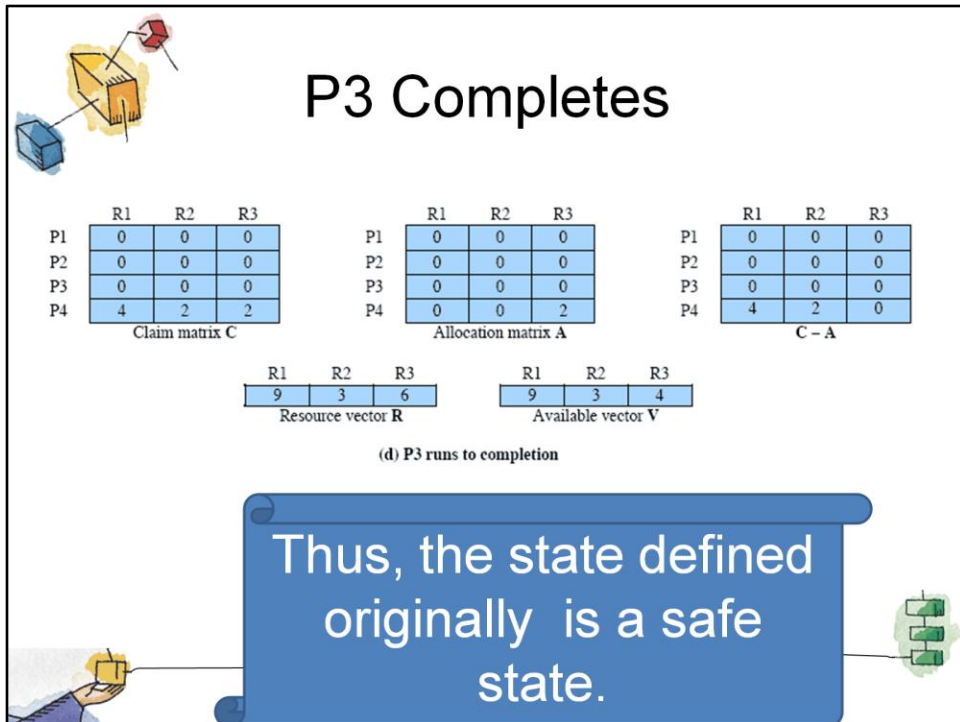
In this case, each of the remaining processes could be completed as shown on the next slides



Suppose we choose P1,

- allocate the required resources,
- complete P1,
- and return all of P1's resources to the available pool.

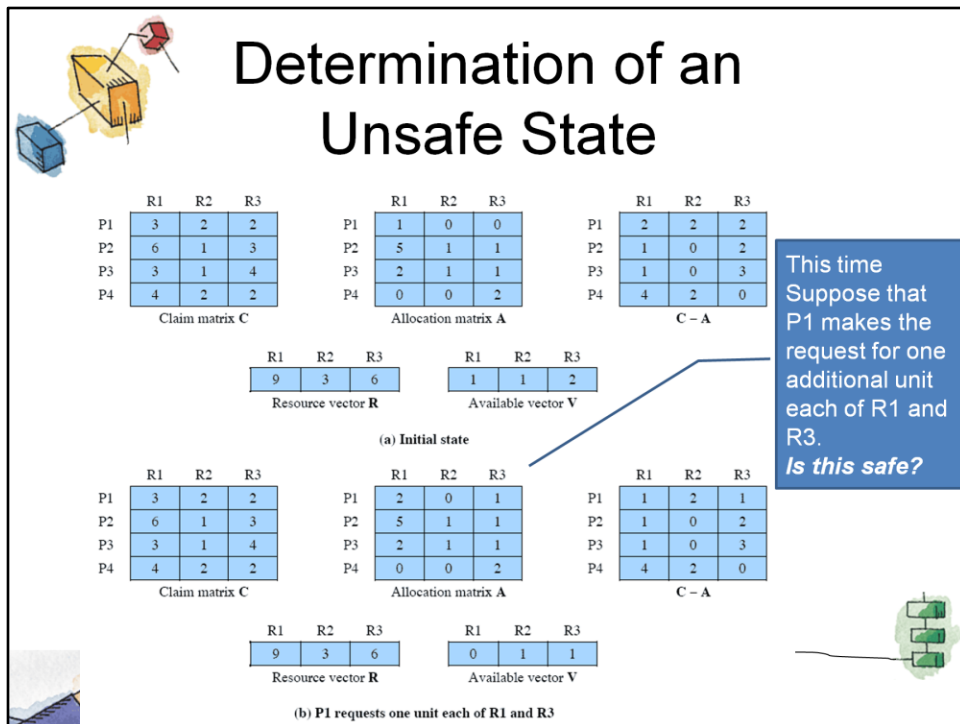
We are left in the state shown in Figure 6.7c on this slide



P3 completes, resulting in the state of Figure 6.7d shown on this slide

Finally, we can complete P4. At this point, all of the processes have been run to completion.

Thus, the state defined by Figure 6.7a is a safe state.



Suppose that P1 makes the request for one additional unit each of R1 and R3; if we assume that the request is granted,

Is this a safe state?

- No,
- because each process will need at least one additional unit of R1, and there are none available.

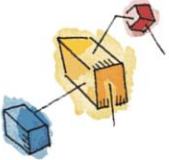
Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

NOTE: This is *not* a deadlocked state.

It merely has the potential for deadlock.



It is possible, for example, that if P1 were run from this state it would subsequently release one unit of R1 and one unit of R3 prior to needing these resources again.

- If that happened, the system would return to a safe state.
- Thus, the deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.



Deadlock Avoidance

- When a process makes a request for a set of resources,
 - assume that the request is granted,
 - Update the system state accordingly,
- Then determine if the result is a safe state.
 - If so, grant the request and,
 - if not, block the process until it is safe to grant the request.

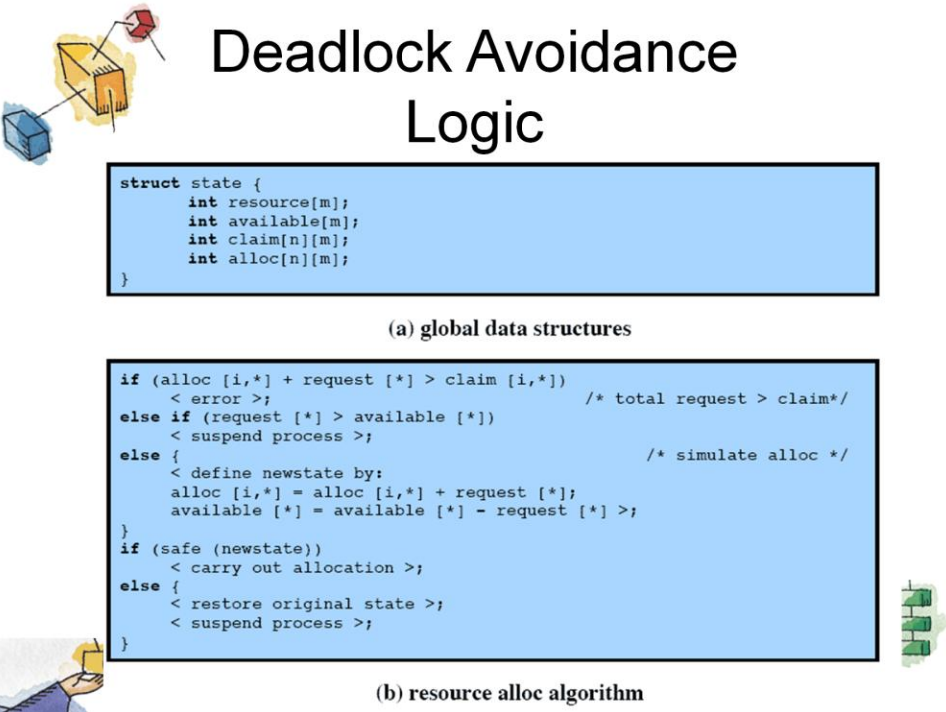


This suggests the following deadlock avoidance strategy,

which ensures that the system of processes and resources is always in a safe state.

When a process makes a request for a set of resources, assume that the request is granted, update

the system state accordingly, and then determine if the result is a safe state. If so, grant the request and, if not, block the process until it is safe to grant the request.



Deadlock Avoidance Logic

```

struct state {
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}

```

(a) global data structures

```

if (alloc [i,*] + request [*] > claim [i,*])
    < error >; /* total request > claim*/
else if (request [*] > available [*])
    < suspend process >;
else {
    /* simulate alloc */
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
    }
    if (safe (newstate))
        < carry out allocation >;
    else {
        < restore original state >;
        < suspend process >;
    }
}

```

(b) resource alloc algorithm


This slide gives an abstract version of the deadlock avoidance logic.

The main algorithm is shown in part (b).

With the state of the system defined by the data structure state, request[*] is a vector defining the resources requested by process i.

First, a check is made to assure that the request does not exceed the original claim of the process.

- If the request is valid, the next step is to determine if it is possible to fulfill the request (i.e., there are sufficient resources available).
 - If it is not possible, then the process is suspended.
 - If it is possible, the final step is to determine if it is safe to fulfill the request. To do this, the resources are tentatively assigned to process i to form newstate.



Deadlock Avoidance Logic

```
boolean safe (state S) {
  int currentavail[m];
  process rest[<number of processes>];
  currentavail = available;
  rest = {all processes};
  possible = true;
  while (possible) {
    <find a process Pk in rest such that
      claim [k,*] - alloc [k,*] <= currentavail;>
    if (found) { /* simulate execution of Pk */
      currentavail = currentavail + alloc [k,*];
      rest = rest - {Pk};
    }
    else possible = false;
  }
  return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)



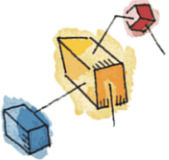



Figure 6.9 Deadlock Avoidance Logic

Then a test for safety is made using the algorithm in Figure 6.9c.

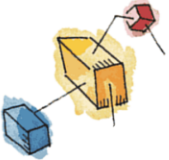


Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection,
- It is less restrictive than deadlock prevention.





Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection, and is less restrictive than deadlock prevention.



Deadlock Avoidance Restrictions

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources



However, it does have a number of restrictions on its use:

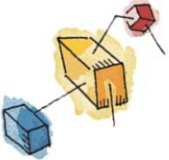
- The maximum resource requirement for each process must be stated in advance.
- The processes under consideration must be independent;
that is, the order in which they execute must be unconstrained by any synchronization requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.



Roadmap



- Principals of Deadlock
 - Deadlock prevention
 - Deadlock Avoidance
 - ➔ – Deadlock detection
 - ➔ – An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





Deadlock Detection

- Deadlock prevention strategies are very conservative;
 - limit access to resources and impose restrictions on processes.
- Deadlock detection strategies do the opposite
 - Resource requests are granted whenever possible.
 - Regularly check for deadlock

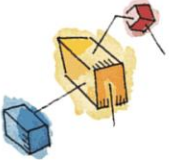


Deadlock checks can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.

Checking at each resource request has two advantages:



- it leads to early detection,
- the algorithm is relatively simple because it is based on incremental changes to the state of the system.

On the other hand, such frequent checks consume considerable processor time.



A Common Detection Algorithm

- Use a Allocation matrix and Available vector as previous
- Also use a request matrix Q
 - Where Q_{ij} indicates that an amount of resource j is requested by process i
- First 'un-mark' all processes that are not deadlocked
 - Initially that is all processes



A common algorithm for deadlock detection is one described in [COFF71].

The Allocation matrix and Available vector described in the previous section are used.

In addition, a request matrix Q is defined such that Q_{ij} represents the amount of resources of type j requested by process i .


The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked.



Detection Algorithm

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector \mathbf{W} to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the i th row of Q is less than or equal to \mathbf{W} .
 - i.e. $Q_{ik} \leq W_k$ for $1 \leq k \leq m$.
 - If no such row is found, terminate







Detection Algorithm cont.

4. If such a row is found,
 - mark process i and add the corresponding row of the allocation matrix to W .
 - i.e. set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$

Return to step 3.

- A deadlock exists if and only if there are unmarked processes at the end
- Each unmarked process is deadlocked.



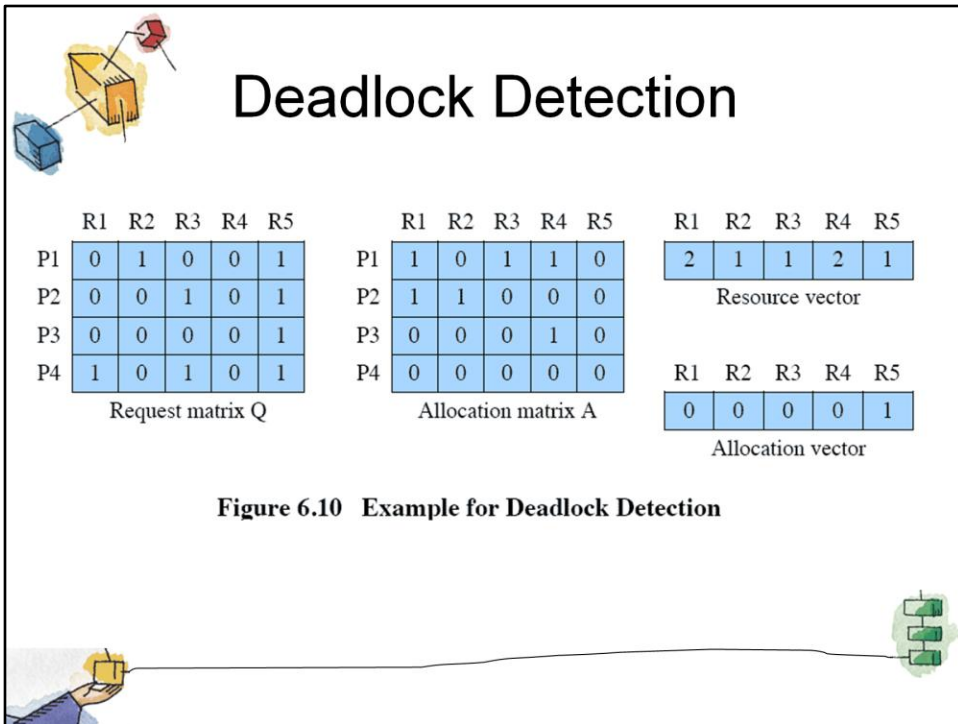
The strategy in this algorithm is to find a process whose resource requests can be satisfied with the available resources,

and then assume that those resources are granted and that the process runs to completion and releases all of its resources.

The algorithm then looks for another process to satisfy.

Note that this algorithm does not guarantee to prevent deadlock;

- That will depend on the order in which future requests are granted.
- All that it does is determine if deadlock currently exists.



1. Mark P4, because P4 has no allocated resources.
2. Set $\mathbf{W} = (0\ 0\ 0\ 0\ 1)$.
3. The request of process P3 is less than or equal to \mathbf{W} , so mark P3 and set $\mathbf{W} = \mathbf{W} + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1)$.
4. No other unmarked process has a row in Q that is less than or equal to \mathbf{W} .
Therefore, terminate the algorithm.


The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.



Recovery Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
 - Risk of deadlock recurring
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists







Advantages and Disadvantages

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> • Works well for processes that perform a single burst of activity • No preemption necessary 	<ul style="list-style-type: none"> • Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> • Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> • Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> • Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> • Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> • No preemption necessary 	<ul style="list-style-type: none"> • Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> • Never delays process initiation • Facilitates online handling 	<ul style="list-style-type: none"> • Inherent preemption losses

There are strengths and weaknesses to all of the strategies for dealing with deadlock.

Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations.



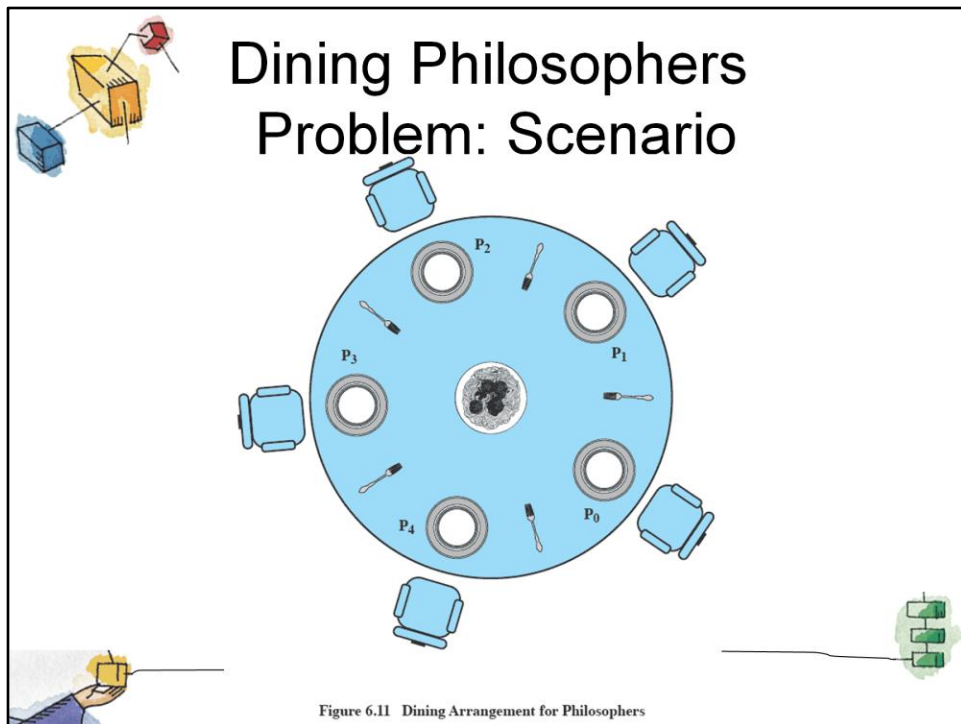
Roadmap

- Principals of Deadlock
 - Deadlock prevention
 - Deadlock Avoidance
 - Deadlock detection
 - An Integrated deadlock strategy

→ Dining Philosophers Problem

- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





Five philosophers live in a house, where a table is laid for them.

The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti.

Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.


A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.



The Problem

- Devise a ritual (algorithm) that will allow the philosophers to eat.
 - No two philosophers can use the same fork at the same time (mutual exclusion)
 - No philosopher must starve to death (avoid deadlock and starvation ... literally!)





A first solution using semaphores

```
/* program    diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
}
```




Figure 6.12 A First Solution to the Dining Philosophers Problem

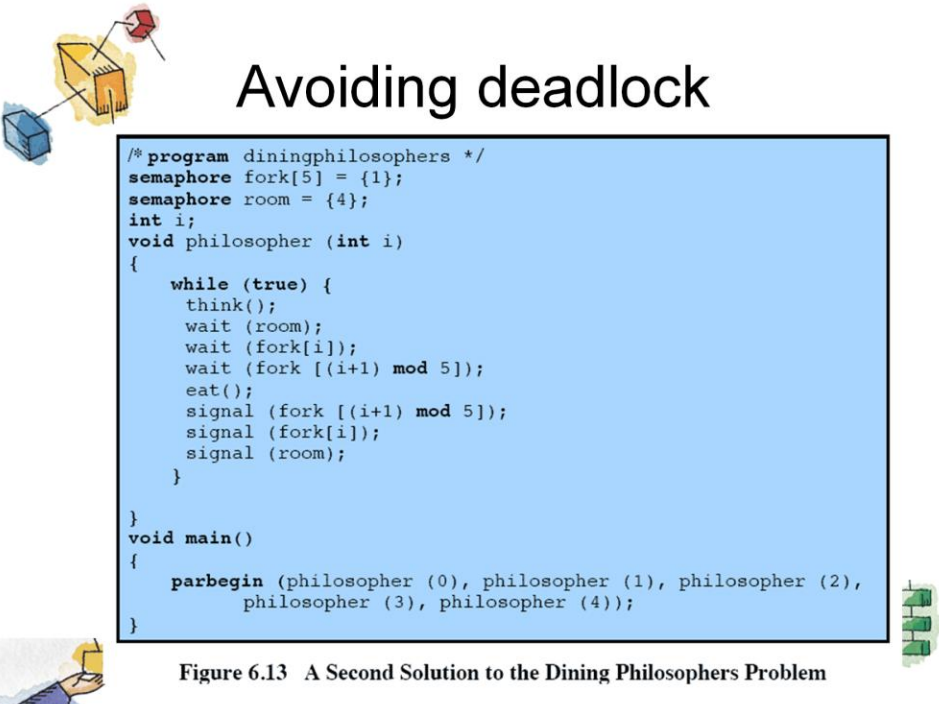
Each philosopher picks up first the fork on the left and then the fork on the right.

After the philosopher is finished eating, the two forks are replaced on the table.

This solution, alas, leads to deadlock:

- If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there.

In this undignified position, all philosophers starve.



Avoiding deadlock

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
             philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

We could consider adding an attendant who only allows four philosophers at a time into the dining room.

With at most four seated philosophers, at least one philosopher will have access to two forks.

This slide shows such a solution, again using semaphores. This solution is free of deadlock and starvation.

Solution using Monitors

```

monitor dining_controller;
cond ForkReady[5]; /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork(left))
        cwait(ForkReady[left]); /* queue on condition variable */
    fork(left) = false;
    /*grant the right fork*/
    if (!fork(right))
        cwait(ForkReady[right]); /* queue on condition variable */
    fork(right) = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left]) /*no one is waiting for this fork */
        fork(left) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right]) /*no one is waiting for this fork */
        fork(right) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}

```

This slide shows a solution to the dining philosophers problem using a monitor – and is continued on next slide

A vector of five condition variables is defined, one condition variable per fork.

These condition variables are used to enable a philosopher to wait for the availability of a fork.

- In addition, there is a Boolean vector that records the availability status of each fork (true means the fork is available).

The monitor consists of two procedures.

- The **get_forks** procedure is used by a philosopher to seize his or her left and right forks.
 - If either fork is unavailable, the philosopher process is queued on the appropriate condition variable.
 - This enables another philosopher process to enter the monitor.
- The **release-forks** procedure is used to make two forks available.

Note that the structure of this solution is similar to that of the semaphore solution proposed in Figure 6.12.

In both cases, a philosopher seizes first the left fork and then the right fork.

Unlike the semaphore solution, this monitor solution does not suffer from deadlock, because only one process at a time may be in the monitor.

For example, the first philosopher process to enter the monitor is guaranteed that it can pick up the right fork after it picks up the left fork before the next philosopher to the right has a chance to seize its left fork, which is this philosopher's right fork.



Monitor solution cont.

```
void philosopher[k=0 to 4]      /* the five philosopher clients */
{
  while (true) {
    <think>;
    get forks(k);                /* client requests two forks via monitor */
    <eat spaghetti>;
    release forks(k);           /* client releases forks via the monitor */
  }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor





Roadmap

- Principals of Deadlock
 - Deadlock prevention
 - Deadlock Avoidance
 - Deadlock detection
 - An Integrated deadlock strategy
- Dining Philosophers Problem

→ Concurrency Mechanisms in UNIX, Linux, Solaris and Windows

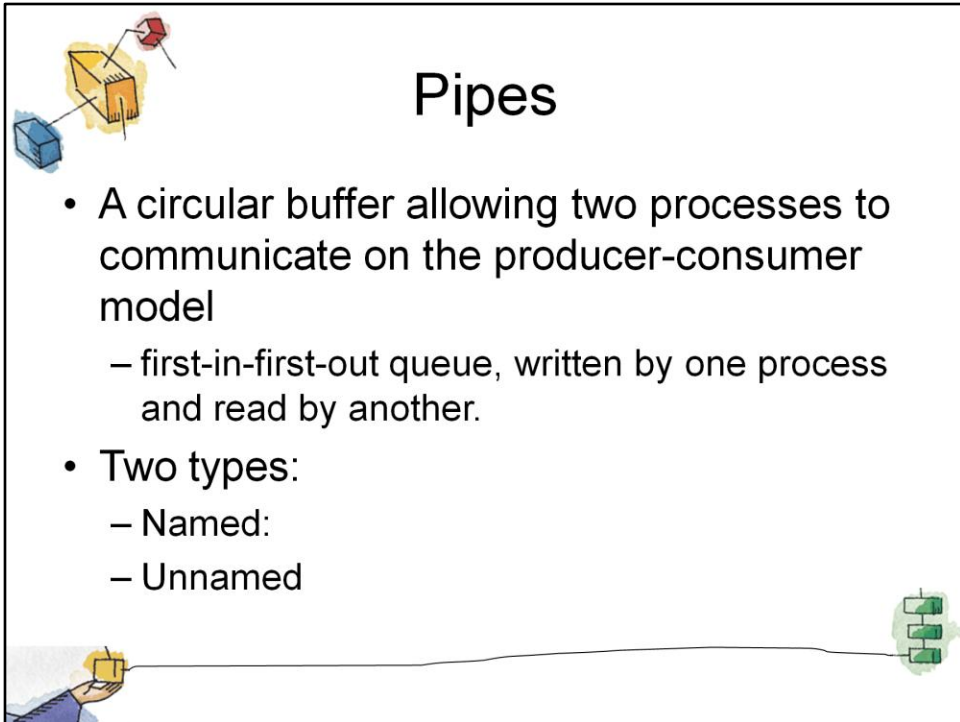




UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:
 - Pipes
 - Messages
 - Shared memory
 - Semaphores
 - Signals





Pipes

- A circular buffer allowing two processes to communicate on the producer-consumer model
 - first-in-first-out queue, written by one process and read by another.
- Two types:
 - Named:
 - Unnamed

a pipe is a circular buffer allowing two processes to communicate on the producer-consumer model.

Thus, it is a first-in-first-out queue, written by one process and read by another.

There are two types of pipes: named and unnamed.

- Only related processes can share unnamed pipes,
- while either related or unrelated processes can share named pipes.



Messages

- A block of bytes with an accompanying type.
- UNIX provides *msgsnd* and *msgrcv* system calls for processes to engage in message passing.
- Associated with each process is a message queue, which functions like a mailbox.





Shared Memory

- A common block of virtual memory shared by multiple processes.
- Permission is read-only or read-write for a process,
 - determined on a per-process basis.
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory.

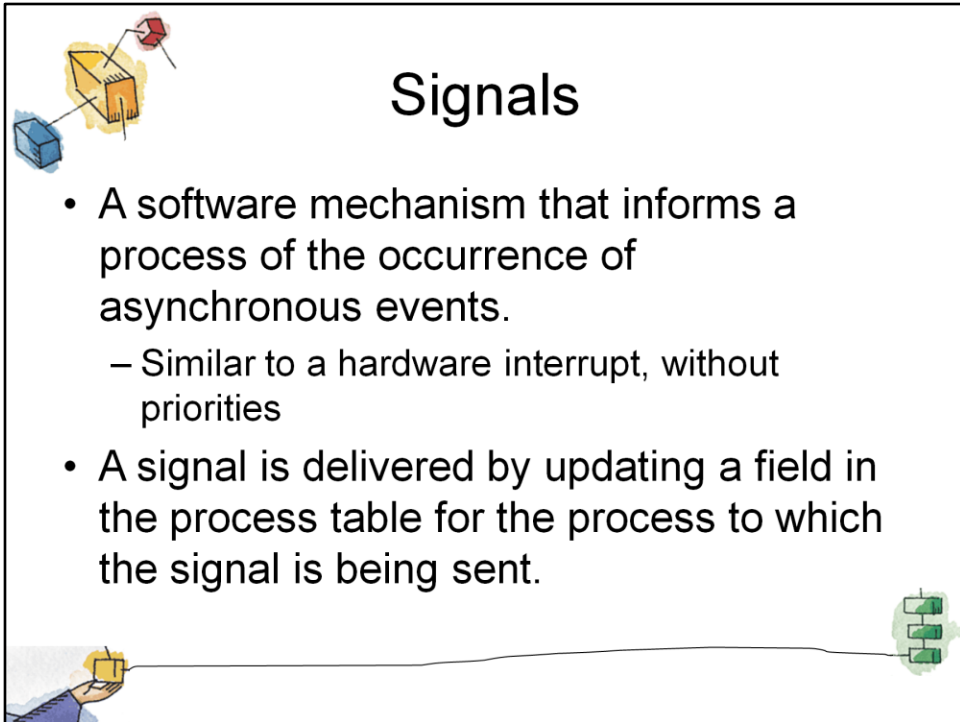




Semaphores

- SVR4 uses a generalization of the ***semWait*** and ***semSignal*** primitives defined in Chapter 5;
- Associated with the semaphore are queues of processes blocked on that semaphore.





Signals

- A software mechanism that informs a process of the occurrence of asynchronous events.
 - Similar to a hardware interrupt, without priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent.

A software mechanism that informs a process of the occurrence of asynchronous events.

- A signal is similar to a hardware interrupt but does not employ priorities.
- i.e., all signals are treated equally; signals that occur at the same time are presented to a process one at a time, with no particular ordering.

Processes may send each other signals, or the kernel may send signals internally.

A signal is delivered by updating a field in the process table for the process to which the signal is being sent. Because each signal is maintained as a single bit, signals of a given type cannot be queued.

A signal is processed just after a process wakes up to run or whenever the process is preparing to return from a system call. A process may respond to a signal by performing some default action (e.g., termination), executing a signal handler function, or ignoring the signal.



Signals defined for UNIX SVR4.

Value	Name	Description
01	SIGHUP	Hang up, sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure






Linux Kernel Concurrency Mechanism



- Includes all the mechanisms found in UNIX plus
 - Atomic operations
 - Spinlocks
 - Semaphores (slightly different to SVR4)
 - Barriers





Atomic Operations

- Atomic operations execute without interruption and without interference
- Two types:
 - Integer operations – operating on an integer variable
 - Bitmap operations – operating on one bit in a bitmap



Two types of atomic operations are defined in Linux:

- integer operations, which operate on an integer variable, and
- bitmap operations, which operate on one bit in a bitmap



Linux Atomic Operations

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
<code>ATOMIC_INIT (int i)</code>	At declaration: initialize an <code>atomic_t</code> to <code>i</code>
<code>int atomic_read(atomic_t *v)</code>	Read integer value of <code>v</code>
<code>void atomic_set(atomic_t *v, int i)</code>	Set the value of <code>v</code> to integer <code>i</code>
<code>void atomic_add(int i, atomic_t *v)</code>	Add <code>i</code> to <code>v</code>
<code>void atomic_sub(int i, atomic_t *v)</code>	Subtract <code>i</code> from <code>v</code>
<code>void atomic_inc(atomic_t *v)</code>	Add 1 to <code>v</code>
<code>void atomic_dec(atomic_t *v)</code>	Subtract 1 from <code>v</code>
<code>int atomic_sub_and_test(int i, atomic_t *v)</code>	Subtract <code>i</code> from <code>v</code> ; return 1 if the result is zero; return 0 otherwise
<code>int atomic_add_negative(int i, atomic_t *v)</code>	Add <code>i</code> to <code>v</code> ; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
<code>int atomic_dec_and_test(atomic_t *v)</code>	Subtract 1 from <code>v</code> ; return 1 if the result is zero; return 0 otherwise
<code>int atomic_inc_and_test(atomic_t *v)</code>	Add 1 to <code>v</code> ; return 1 if the result is zero; return 0 otherwise






Linux Atomic Operations



Atomic Bitmap Operations	
<code>void set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr
<code>void clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr
<code>void change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr
<code>int test_and_set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_bit(int nr, void *addr)</code>	Return the value of bit nr in the bitmap pointed to by addr





Spinlock

- Only one thread at a time can acquire a spinlock.
 - Any other thread will keep trying (spinning) until it can acquire the lock.
- A spinlock is an integer
 - If 0, the thread sets the value to 1 and enters its critical section.
 - If the value is nonzero, the thread continually checks the value until it is zero.



Only one thread at a time can acquire a spinlock.

Any other thread attempting to acquire the same lock will keep trying (spinning) until it can acquire the lock.

In essence a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section.

- If the value is 0, the thread sets the value to 1 and enters its critical section.
- If the value is nonzero, the thread continually checks the value until it is zero.

The spinlock is easy to implement but has the disadvantage that locked-out threads continue to execute in a busy-waiting mode.

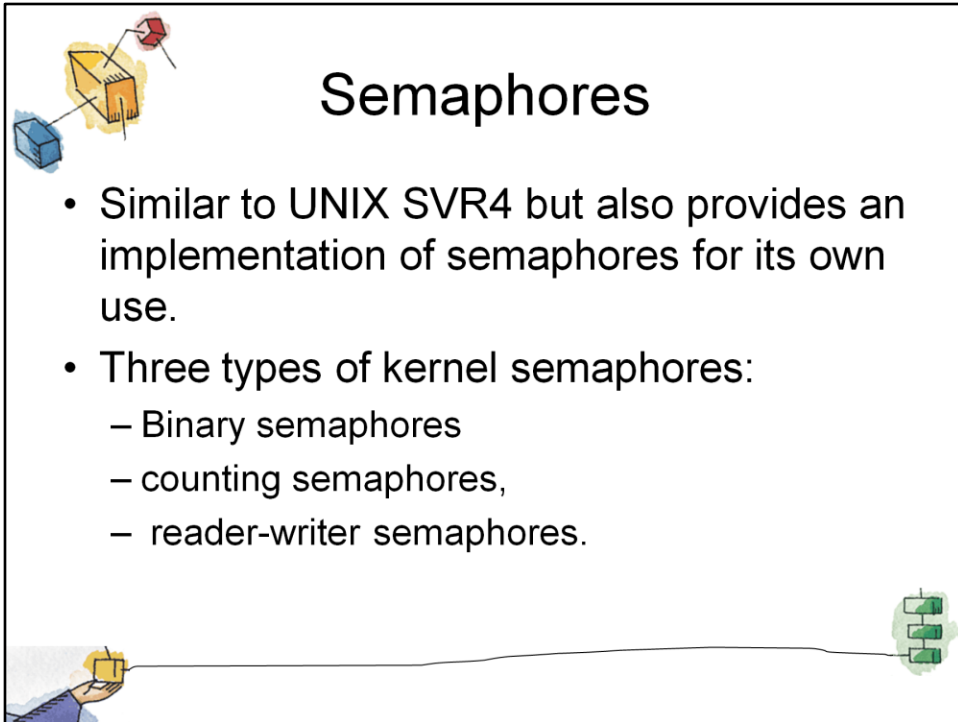
- Thus spinlocks are most effective in situations where the wait time for acquiring a lock is expected to be very short, say on the order of less than two context changes.



Linux Spinlocks

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like <code>spin_lock_irq</code> , but also saves the current interrupt state in flags
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise





Semaphores

- Similar to UNIX SVR4 but also provides an implementation of semaphores for its own use.
- Three types of kernel semaphores:
 - Binary semaphores
 - counting semaphores,
 - reader-writer semaphores.

Linux provides a semaphore interface corresponding to that in UNIX SVR4.

Internally, Linux provides an implementation of semaphores for its own use.

That is, code that is part of the kernel can invoke kernel semaphores.

These kernel semaphores cannot be accessed directly by the user program via system calls.

They are implemented as functions within the kernel and are thus more efficient than user-visible semaphores.

Linux provides three types of semaphore facilities in the kernel:

- binary semaphores,
- counting semaphores, and
- reader-writer semaphores.



Linux Semaphores

Traditional Semaphores	
<code>void sema_init(struct semaphore *sem, int count)</code>	Initializes the dynamically created semaphore to the given count
<code>void init_MUTEX(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 0 (initially locked)
<code>void down(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
<code>int down_interruptible(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
<code>int down_trylock(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
<code>void up(struct semaphore *sem)</code>	Releases the given semaphore
Reader-Writer Semaphores	
<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	Initializes the dynamically created semaphore with a count of 1
<code>void down_read(struct rw_semaphore, *rwsem)</code>	Down operation for readers
<code>void up_read(struct rw_semaphore, *rwsem)</code>	Up operation for readers
<code>void down_write(struct rw_semaphore, *rwsem)</code>	Down operation for writers
<code>void up_write(struct rw_semaphore, *rwsem)</code>	Up operation for writers





Barriers

- To enforce the order in which instructions are executed, Linux provides the memory barrier facility.


Table 6.6 Linux Memory Barrier Operations

<code>rmb()</code>	Prevents loads from being reordered across the barrier
<code>wmb()</code>	Prevents stores from being reordered across the barrier
<code>mb()</code>	Prevents loads and stores from being reordered across the barrier
<code>Barrier()</code>	Prevents the compiler from reordering loads or stores across the barrier
<code>smp_rmb()</code>	On SMP, provides a <code>rmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_wmb()</code>	On SMP, provides a <code>wmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_mb()</code>	On SMP, provides a <code>mb()</code> and on UP provides a <code>barrier()</code>




SMP = symmetric multiprocessor
UP = uniprocessor





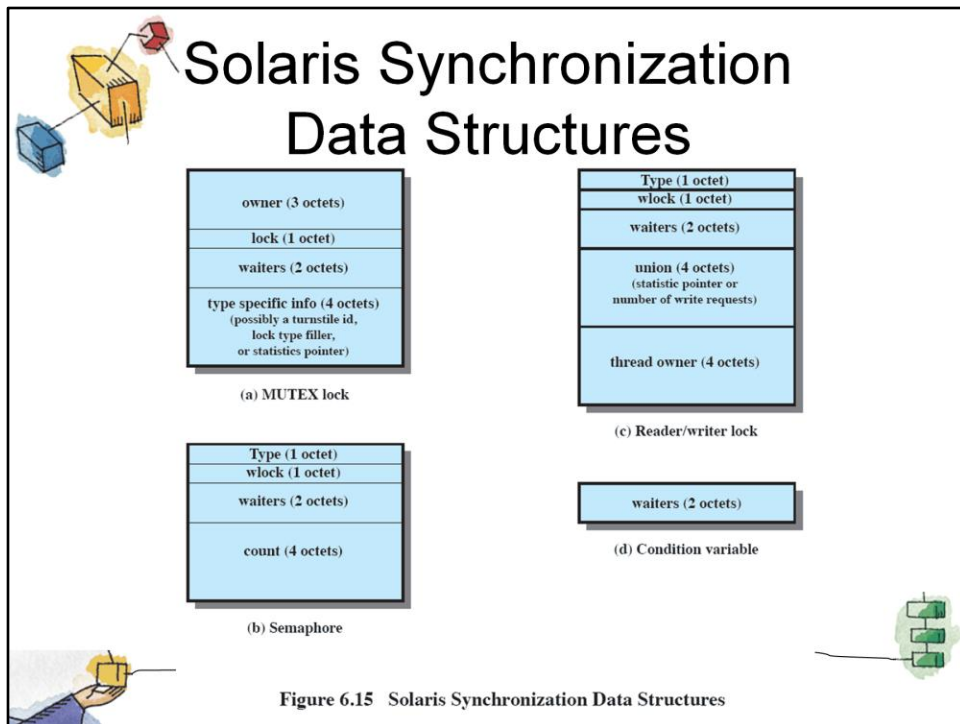
Solaris Thread Synchronization Primitives

- In addition to the concurrency mechanisms of UNIX SVR4
 - Mutual exclusion (mutex) locks
 - Semaphores
 - Multiple readers, single writer (readers/writer) locks
 - Condition variables



In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables



The initialization functions for the primitives fill in some of the data members.

Once a synchronization object is created, there are essentially only two operations that can be performed:

- enter (acquire lock) and
- release (unlock).

There are no mechanisms in the kernel or the threads library to enforce mutual exclusion or to prevent deadlock.

- If a thread attempts to access a piece of data or code that is supposed to be protected but does not use the appropriate synchronization primitive, then such access occurs.
- If a thread locks an object and then fails to unlock it, no kernel action is taken.

All of the synchronization primitives require the existence of a hardware instruction that allows an object to be tested and set in one atomic operation.



MUTEX Lock

- A mutex is used to ensure only one thread at a time can access the resource protected by the mutex.
- The thread that locks the mutex must be the one that unlocks it.



Semaphores and Read/Write locks

- Solaris provides classic counting semaphores.
- The readers/writer lock allows multiple threads to have simultaneous read-only access to an object protected by the lock.
 - It also allows a single thread to access the object for writing at one time, while excluding all readers.


Movie button links to Solaris RW Lock animation/simulation
<http://gaia.ecs.csus.edu/~zhangd/oscal/SolarisRWLock/SolarisRWLock.html>



Condition Variables



- A condition variable is used to wait until a particular condition is true.
- Condition variables must be used in conjunction with a mutex lock.





Windows concurrency mechanisms

- Windows provides synchronization among threads as part of the object architecture.
- Important methods of synchronization are
 - Executive dispatcher objects (using Wait functions),
 - user mode critical sections,
 - slim reader-writer locks, and
 - condition variables.



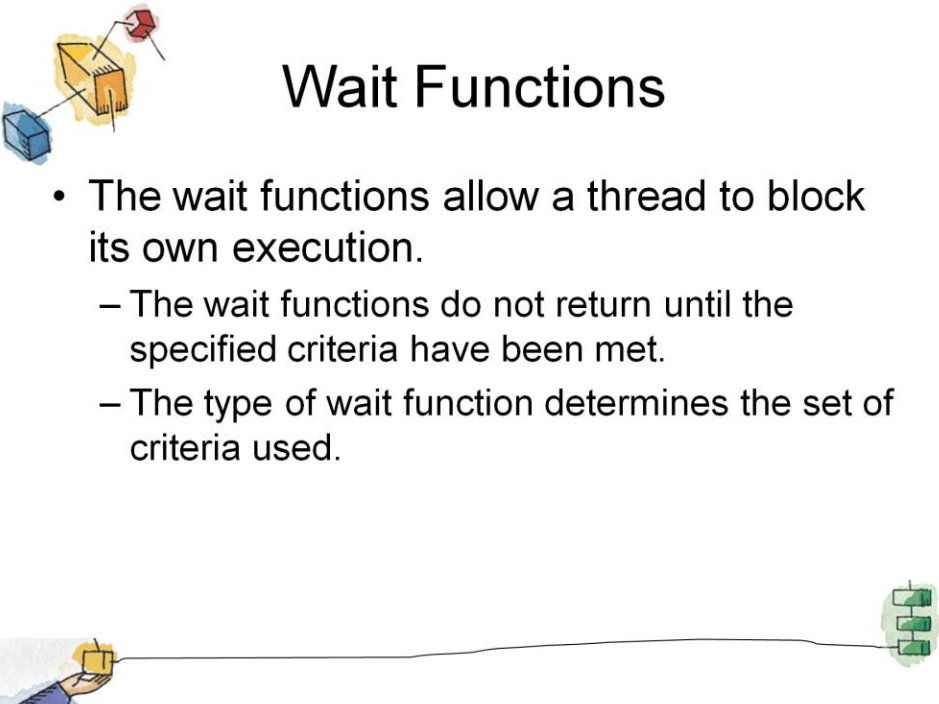
Windows provides synchronization among threads as part of the object architecture.

The most important methods of synchronization are

- Executive dispatcher objects,
- user mode critical sections,
- slim reader-writer locks, and
- condition variables.

Dispatcher objects make use of wait functions.

We first describe wait functions and then look at the synchronization methods.



Wait Functions


- The wait functions allow a thread to block its own execution.
 - The wait functions do not return until the specified criteria have been met.
 - The type of wait function determines the set of criteria used.

The wait functions allow a thread to block its own execution.

- The wait functions do not return until the specified criteria have been met.
- The type of wait function determines the set of criteria used.

When a wait function is called, it checks whether the wait criteria have been met.



- If the criteria have not been met, the calling thread enters the wait state.
- It uses no processor time while waiting for the criteria to be met.



Dispatcher Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.





The first five object types in the table are specifically designed to support synchronization.

The remaining object types have other uses but also may be used for synchronization.



Each dispatcher object instance can be in either a signaled or unsignaled state.

A thread can be blocked on an object in an unsignaled state;
the thread is released when the object enters the signaled state



Critical Sections

- Similar mechanism to mutex
 - except that critical sections can be used only by the threads of a single process.
- If the system is a multiprocessor, the code will attempt to acquire a spin-lock.
 - As a last resort, if the spinlock cannot be acquired, a dispatcher object is used to block the thread so that the Kernel can dispatch another thread onto the processor.



Critical sections provide a synchronization mechanism similar to that provided by mutex objects,

except that critical sections can be used only by the threads of a single process.

Event, mutex, and semaphore objects can also be used in a single process application, but critical sections provide a much faster, more efficient mechanism for mutual-exclusion synchronization.

Critical sections use a sophisticated algorithm when trying to acquire the mutex.

- If the system is a multiprocessor, the code will attempt to acquire a spin-lock.
- Effectively the spinlock optimizes for the case where the thread that currently owns the critical section is executing on another processor.
- If the spinlock cannot be acquired within a reasonable number of iterations, a dispatcher object is used to block the thread so that the Kernel can dispatch another thread onto the processor.
 - The dispatcher object is only allocated as a last resort.

Most critical sections are needed for correctness, but in practice are rarely contended. By lazily allocating the

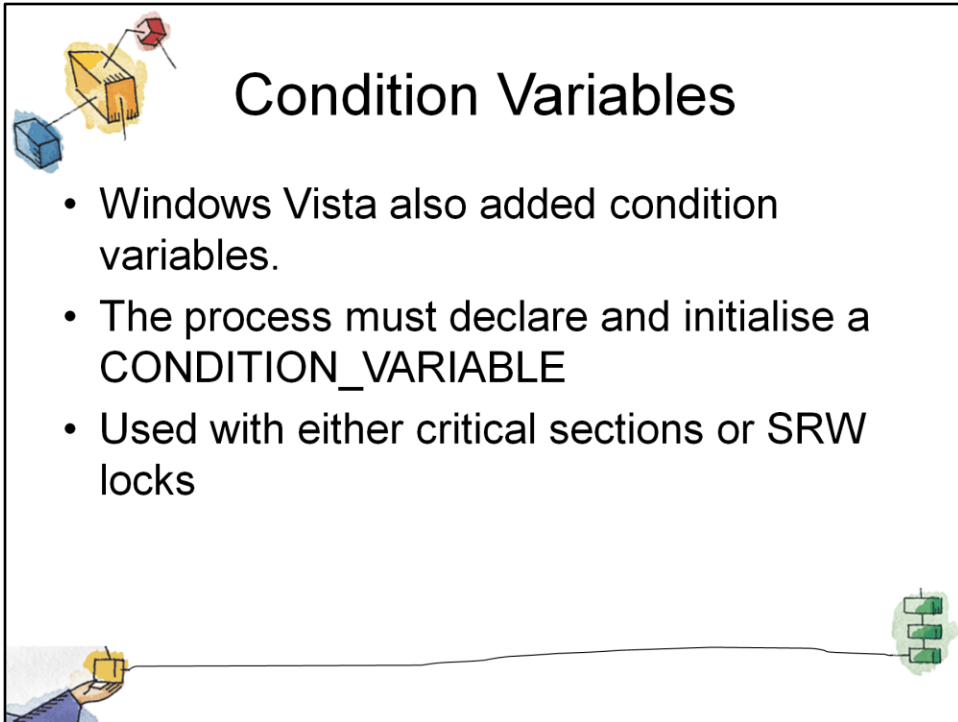
dispatcher object the system saves significant amounts of kernel virtual memory.



Slim Read-Writer Locks

- Windows Vista added a user mode reader-writer.
- The readerwriter lock enters the kernel to block only after attempting to use a spin-lock.
- 'Slim' as it normally only requires allocation of a single pointer-sized piece of memory.





Condition Variables

- Windows Vista also added condition variables.
- The process must declare and initialise a `CONDITION_VARIABLE`
- Used with either critical sections or SRW locks

The process must declare a `CONDITION_VARIABLE` and initialize it in some thread by calling `InitializeConditionVariable`.

Condition variables can be used with either critical sections or SRW locks, so there are two methods which sleep on the specified condition and releases the specified lock as an atomic operation:

- `SleepConditionVariableCS`
- `SleepConditionVariableSRW`,

There are two wake methods, which wake one or all of the sleeping threads:

- `WakeConditionVariable` and
- `WakeAllConditionVariable`,



Windows/Linux Comparison

Windows	Linux
Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying wait/signal mechanism	Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying sleep/wakeup mechanism
Many kernel objects are also dispatcher objects, meaning that threads can synchronize with them using a common event mechanism, available at user-mode. Process and thread termination are events, I/O completion is an event	
Threads can wait on multiple dispatcher objects at the same time	Processes can use the select() system call to wait on I/O from up to 64 file descriptors
User-mode reader/writer locks and condition variables are supported	User-mode reader/writer locks and condition variables are supported
Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported	Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported
A non-locking atomic LIFO queue, called an SLIST, is supported using compare-and-swap; widely used in the OS and also available to user programs	





Windows/Linux Comparison cont.

Windows	Linux
A large variety of synchronization mechanisms exist within the kernel to improve scalability. Many are based on simple compare-and-swap mechanisms, such as push-locks and fast references of objects	
Named pipes, and sockets support remote procedure calls (RPCs), as does an efficient Local Procedure Call mechanism (ALPC), used within a local system. ALPC is used heavily for communicating between clients and local services	Named pipes, and sockets support remote procedure calls (RPCs)
Asynchronous Procedure Calls (APCs) are used heavily within the kernel to get threads to act upon themselves (e.g. termination and I/O completion use APCs since these operations are easier to implement in the context of a thread rather than cross-thread). APCs are also available for user-mode, but user-mode APCs are only delivered when a user-mode thread blocks in the kernel	Unix supports a general signal mechanism for communication between processes. Signals are modeled on hardware interrupts and can be delivered at any time that they are not blocked by the receiving process; like with hardware interrupts, signal semantics are complicated by multi-threading
Hardware support for deferring interrupt processing until the interrupt level has dropped is provided by the Deferred Procedure Call (DPC) control object	Uses tasklets to defer interrupt processing until the interrupt level has dropped

