Lecture 8

Deadlock & Main Memory I

Lecture Information

Ceng328 *Operating Systems* at April 13, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- **Reusable**: something that can be safely used by one process at a time and is not consumed by that use (processors, memory, files, devices, databases, and semaphores).

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- **Reusable**: something that can be safely used by one process at a time and is not consumed by that use (processors, memory, files, devices, databases, and semaphores).
- **Consumable**: these can be created and destroyed (interrupts, signals, messages, and information in I/O buffers).

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- **Reusable**: something that can be safely used by one process at a time and is not consumed by that use (processors, memory, files, devices, databases, and semaphores).
- **Consumable**: these can be created and destroyed (interrupts, signals, messages, and information in I/O buffers).
- A **preemptable resource** is one that can be taken away from the process owning it with no ill effects.(Memory, CPU)

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- **Reusable**: something that can be safely used by one process at a time and is not consumed by that use (processors, memory, files, devices, databases, and semaphores).
- **Consumable**: these can be created and destroyed (interrupts, signals, messages, and information in I/O buffers).
- A **preemptable resource** is one that can be taken away from the process owning it with no ill effects.(Memory, CPU)
- A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail (printer, CD-R(W)floppy disk).

Deadlocks & System Model I

- A system consists of a finite number of *resources* to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- **Reusable**: something that can be safely used by one process at a time and is not consumed by that use (processors, memory, files, devices, databases, and semaphores).
- **Consumable**: these can be created and destroyed (interrupts, signals, messages, and information in I/O buffers).
- A **preemptable resource** is one that can be taken away from the process owning it with no ill effects.(Memory, CPU)
- A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail (printer, CD-R(W)floppy disk).
- In general, deadlocks occur when sharing *reusable* and *nonpreemptable* resources.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource.
  3. **Release**. The process releases the resource.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource.
  3. **Release**. The process releases the resource.
- Request and release of resources that are not managed by the OS can be accomplished through the *wait*() and *signal*() operations on semaphores or through acquisition and release of a mutex lock.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource.
  3. **Release**. The process releases the resource.
- Request and release of resources that are not managed by the OS can be accomplished through the *wait*() and *signal*() operations on semaphores or through acquisition and release of a mutex lock.
- A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated.

Deadlocks & System Model II

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request**. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource.
  3. **Release**. The process releases the resource.
- Request and release of resources that are not managed by the OS can be accomplished through the *wait*() and *signal*() operations on semaphores or through acquisition and release of a mutex lock.
- A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated.
- A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again.

Deadlocks & System Model III

- One possible way of allowing user management of resources is to associate a semaphore with each resource. Mutexes can be used equally well.

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

Figure: Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Deadlocks & System Model III

- One possible way of allowing user management of resources is to associate a semaphore with each resource. Mutexes can be used equally well.

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
      down(&resource_1);
      use_resource_1( );
      up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

(b)

Figure: Using a semaphore to protect resources. (a) One resource. (b) Two resources.

- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

Deadlocks & System Model IV

- Deadlocks can occur in a variety of situations. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions.

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
    }
```

(b)

Figure: (a) Deadlock-free code. (b) Code with a potential deadlock.

Deadlocks & System Model IV

- Deadlocks can occur in a variety of situations. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions.
  - If process *A* locks record *R*1 and process *B* locks record *R*2, and then each process tries to lock the other one's record, we also have a deadlock (see Fig. 2).

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
    }
```

(b)

Figure: (a) Deadlock-free code. (b) Code with a potential deadlock.

Deadlocks & System Model V

- Deadlocks can occur on hardware resources or on software resources.

Deadlocks & System Model V

- Deadlocks can occur on hardware resources or on software resources.
- Unlike other problems in multiprogramming systems, *there is no efficient solution to the deadlock problem* in the general case.

Deadlocks & System Model V

- Deadlocks can occur on <u>hardware resources</u> or on <u>software resources</u>.
- Unlike other problems in multiprogramming systems, *there is no efficient solution to the deadlock problem* in the general case.
- A programmer who is developing multithreaded applications must pay particular attention to this problem.

Deadlocks & System Model V

- Deadlocks can occur on <u>hardware resources</u> or on <u>software resources</u>.
- Unlike other problems in multiprogramming systems, *there is no efficient solution to the deadlock problem* in the general case.
- A programmer who is developing multithreaded applications must pay particular attention to this problem.
- Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that

There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
    - P_0 is waiting for a resource held by P_1,

There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
    - P_0 is waiting for a resource held by P_1,
    - P_1 is waiting for a resource held by P_2 ,

There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
    - P_0 is waiting for a resource held by P_1,
    - P_1 is waiting for a resource held by P_2 ,
    - $\vdots$

  There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
    - P_0 is waiting for a resource held by P_1,
    - P_1 is waiting for a resource held by P_2 ,
    - $\vdots$
    - P_{n-1} is waiting for a resource held by P_n,

  There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Necessary Conditions I

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  - **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
    - P_0 is waiting for a resource held by P_1,
    - P_1 is waiting for a resource held by P_2 ,
    - $\vdots$
    - P_{n-1} is waiting for a resource held by P_n,
    - P_n is waiting for a resource held by P_0.

    There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Dr. Cem Özdoğan

Necessary Conditions II

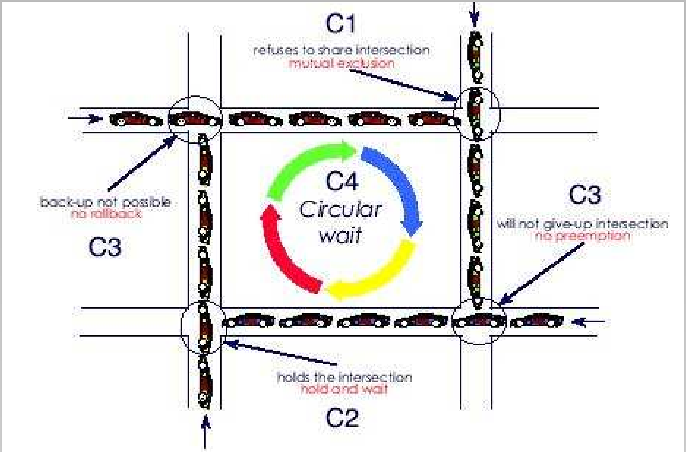


Figure: An example to Deadlock.

We emphasise that **all four conditions must hold for a deadlock** to occur.

Resource-Allocation Graph I

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.

Resource-Allocation Graph I

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.
- This graph consists of a set of vertices V and a set of edges E.

Resource-Allocation Graph I

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.
- This graph consists of a set of vertices V and a set of edges E.
- Pictorially, each process P_i is represented as a circle and each resource type R_j as a rectangle.

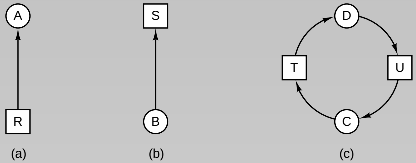


Figure: Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Resource-Allocation Graph I

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.
- This graph consists of a set of vertices V and a set of edges E.
- Pictorially, each process P_i is represented as a circle and each resource type R_j as a rectangle.

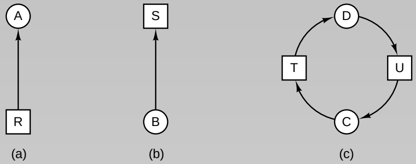


Figure: Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process (see Fig. 4).

Resource-Allocation Graph II

- Since resource type R_j may have more than one instance, each such instance is represented as a dot within the rectangle.

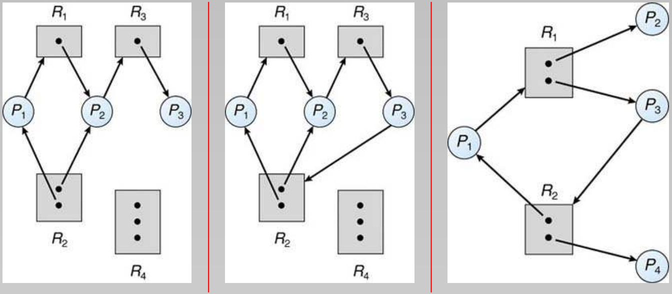


Figure: Left: Resource-allocation graph. Middle: Resource-allocation graph with a deadlock. Right: Resource-allocation graph with a cycle but no deadlock

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Given the definition of a resource-allocation graph, it can be shown that, 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.

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Given the definition of a resource-allocation graph, it can be shown that, 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.
- A cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock with resource types of several instances.

Resource-Allocation Graph III

- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Given the definition of a resource-allocation graph, it can be shown that, 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.
- A cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock with resource types of several instances.
- A *knot* must exist; a cycle with no non-cycle outgoing path from any involved node

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.
    - In this case, we also have a cycle. However, there is no deadlock.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.
  - In this case, we also have a cycle. However, there is no deadlock.
  - Observe that process P_4 may release its instance of resource type R_2.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.
  - In this case, we also have a cycle. However, there is no deadlock.
  - Observe that process P_4 may release its instance of resource type R_2.
  - That resource can then be allocated to P_3 , breaking the cycle.

Resource-Allocation Graph IV

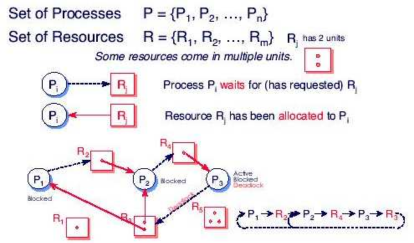
- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.
  - In this case, we also have a cycle. However, there is no deadlock.
  - Observe that process P_4 may release its instance of resource type R_2.
  - That resource can then be allocated to P_3 , breaking the cycle.
- In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.

Resource-Allocation Graph IV

- Suppose that process P_3 requests an instance of resource type R_2.
- Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is is added to the graph (see Fig. 5middle).
- At this point, two minimal cycles exist in the system.
- Now consider the resource-allocation graph in Fig. 5right.
    - In this case, we also have a cycle. However, there is no deadlock.
    - Observe that process P_4 may release its instance of resource type R_2.
    - That resource can then be allocated to P_3 , breaking the cycle.
- In summary, 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.

Resource-Allocation Graph V

- An example of resource allocation graphs (see Fig. 6);



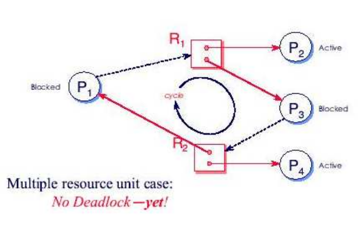


Figure: Resource Allocation Graphs. Lower; either P_2 or P_4 could relinquish (release) a resource allowing P_1 or P_3 (which are currently blocked) to continue.

Resource-Allocation Graph VI

Another example of how resource graphs can be used; three processes, *A*, *B*, and *C*, and three resources *R*, *S*, and *T* (see Fig. 7);.

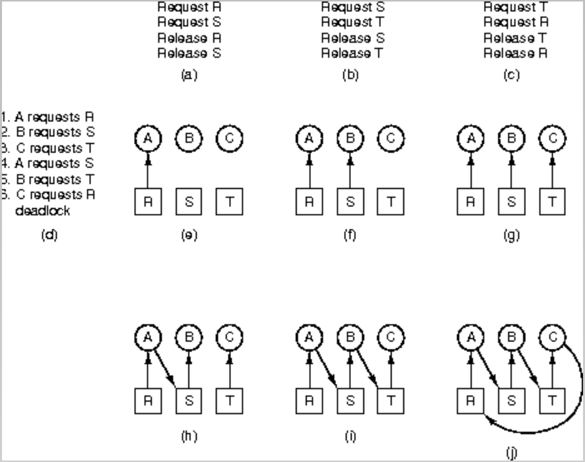


Figure: An example of how deadlock occurs.

Resource-Allocation Graph VII

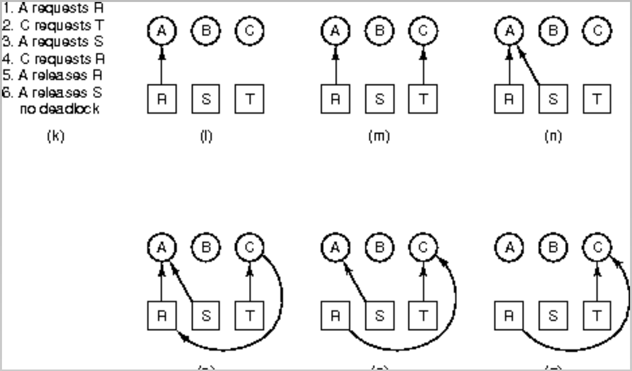


Figure: An example of how deadlock can be avoided.

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).
     - Deadlock avoidance requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime (run-time/dynamically, before it happens).

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).
     - Deadlock avoidance requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime (run-time/dynamically, before it happens).
  2. We can allow the system to enter a deadlock state, **detect** it, and **recover**.

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).
     - Deadlock avoidance requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime (run-time/dynamically, before it happens).
  2. We can allow the system to enter a deadlock state, **detect** it, and **recover**.
     - If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise.

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).
     - Deadlock avoidance requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime (run-time/dynamically, before it happens).
  2. We can allow the system to enter a deadlock state, **detect** it, and **recover**.
     - If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise.
     - In this environment, the system can provide an algorithm that examines the **state** of the system to **determine** whether a deadlock has occurred and an algorithm to **recover** from the deadlock (run-time/dynamically, after it happens)

Methods for Handling Deadlocks I

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  1. We can use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlock state.
     - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1) cannot hold (compile-time/statically, by design).
     - Deadlock avoidance requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime (run-time/dynamically, before it happens).
  2. We can allow the system to enter a deadlock state, **detect** it, and **recover**.
     - If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise.
     - In this environment, the system can provide an algorithm that examines the **state** of the system to **determine** whether a deadlock has occurred and an algorithm to **recover** from the deadlock (run-time/dynamically, after it happens)
  3. We can **ignore** (The Ostrich Algorithm; maybe if you ignore it, it will ignore you) the problem altogether and pretend that deadlocks never occur in the system.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
- Most OSs potentially suffer from deadlocks that are not even detected.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
- Most OSs potentially suffer from deadlocks that are not even detected.
  - Process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
- Most OSs potentially suffer from deadlocks that are not even detected.
  - Process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.
  - The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
- Most OSs potentially suffer from deadlocks that are not even detected.
  - Process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.
  - The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up.
  - Swap space on the disk is another limited resource. In fact, almost every table in the OS represents a finite resource.

Methods for Handling Deadlocks II

- The third solution is the one used by most OSs, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
- Most OSs potentially suffer from deadlocks that are not even detected.
  - Process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.
  - The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up.
  - Swap space on the disk is another limited resource. In fact, almost every table in the OS represents a finite resource.
- If deadlocks could be eliminated for free, there would not be much discussion.

Deadlock Prevention

- Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock?

Deadlock Prevention

- Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock?
- If we can ensure that at least one of the four following conditions is never satisfied, then deadlocks will be structurally impossible.

Deadlock Prevention

- Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock?
- If we can ensure that at least one of the four following conditions is never satisfied, then deadlocks will be structurally impossible.
- The various approaches to deadlock prevention are summarized in Fig. 9.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure: Summary of approaches, to deadlock prevention.

Mutual Exclusion

- **Attacking the Mutual Exclusion Condition**; Can a given resource be assigned to more than one process at once? Systems with only simultaneously shared resources cannot deadlock!

Mutual Exclusion

- **Attacking the Mutual Exclusion Condition**; Can a given resource be assigned to more than one process at once? Systems with only simultaneously shared resources cannot deadlock!
- The mutual-exclusion condition must hold for nonsharable resources (i.e., a printer).

Mutual Exclusion

- **Attacking the Mutual Exclusion Condition**; Can a given resource be assigned to more than one process at once? Systems with only simultaneously shared resources cannot deadlock!
- The mutual-exclusion condition must hold for nonsharable resources (i.e., a printer).
- Shareable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock (i.e.,read-only files). A process never needs to wait for a shareable resource.

Mutual Exclusion

- **Attacking the Mutual Exclusion Condition**; Can a given resource be assigned to more than one process at once? Systems with only simultaneously shared resources cannot deadlock!
- The mutual-exclusion condition must hold for nonsharable resources (i.e., a printer).
- Shareable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock (i.e.,read-only files). A process never needs to wait for a shareable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated <u>all its resources</u> before it begins execution.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated <u>all its resources</u> before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- Both these protocols have two main disadvantages.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- Both these protocols have two main disadvantages.
    - First, resource utilization may be low, since resources may be allocated but unused for a long period.

Hold and Wait

- **Attacking the Hold and Wait Condition**; Can a process hold a resource and ask for another? Can we require processes to request all resources at once?
- *Most processes do not statically know about the resources they need*.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- Both these protocols have two main disadvantages.
  - First, resource utilization may be low, since resources may be allocated but unused for a long period.
  - Second, starvation is possible (wait indefinitely).

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?
- If a process' requests (holding certain resources) is denied, that process must release its unused resources and request them again, together with the additional resource.

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?
- If a process' requests (holding certain resources) is denied, that process must release its unused resources and request them again, together with the additional resource.
- To ensure that this condition does not hold, we can use the following protocol.

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?
- If a process' requests (holding certain resources) is denied, that process must release its unused resources and request them again, together with the additional resource.
- To ensure that this condition does not hold, we can use the following protocol.
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?
- If a process' requests (holding certain resources) is denied, that process must release its unused resources and request them again, together with the additional resource.
- To ensure that this condition does not hold, we can use the following protocol.
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
  - The preempted resources are added to the list of resources for which the process is waiting.

No Preemption

- **Attacking the No Preemption Condition**; Can resources be preempted?
- If a process' requests (holding certain resources) is denied, that process must release its unused resources and request them again, together with the additional resource.
- To ensure that this condition does not hold, we can use the following protocol.
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
  - The preempted resources are added to the list of resources for which the process is waiting.
  - The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?
- Order resources by an index: $R_1, R_2, \ldots$; requires that resources are always requested in order.

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?
- Order resources by an index:$R_1, R_2, \ldots$; requires that resources are always requested in order.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?
- Order resources by an index: $R_1, R_2, \ldots$; requires that resources are always requested in order.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?
- Order resources by an index:$R_1, R_2, \ldots$; requires that resources are always requested in order.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Each process can request resources only in an increasing order of *enumeration*.

Circular Wait

- **Attacking the Circular Wait Condition**; Can circular waits exist?
- Order resources by an index:$R_1, R_2, \ldots$; requires that resources are always requested in order.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Each process can request resources only in an increasing order of *enumeration*.
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
  - For example, in a system with one tape drive and one printer,

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require <u>additional information</u> about how resources are to be requested.
    - For example, in a system with one tape drive and one printer,
        - the system might need to know that process *P* will request first the tape drive

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require <u>additional information</u> about how resources are to be requested.
  - For example, in a system with one tape drive and one printer,
    - the system might need to know that process *P* will request first the tape drive
    - and then the printer before releasing both resources,

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require <u>additional information</u> about how resources are to be requested.
  - For example, in a system with one tape drive and one printer,
    - the system might need to know that process *P* will request first the tape drive
    - and then the printer before releasing both resources,
    - whereas process *Q* will request first the printer

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require <u>additional information</u> about how resources are to be requested.
  - For example, in a system with one tape drive and one printer,
    - the system might need to know that process *P* will request first the tape drive
    - and then the printer before releasing both resources,
    - whereas process *Q* will request first the printer
    - and then the tape drive.

Deadlock Avoidance I

- Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require <u>additional information</u> about how resources are to be requested.
  - For example, in a system with one tape drive and one printer,
    - the system might need to know that process *P* will request first the tape drive
    - and then the printer before releasing both resources,
    - whereas process *Q* will request first the printer
    - and then the tape drive.
  - With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

Deadlock Avoidance II

- Each request requires that in making this decision the system consider

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,
  - the resources currently allocated to each process,

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,
  - the resources currently allocated to each process,
  - the future requests and releases of each process.

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,
  - the resources currently allocated to each process,
  - the future requests and releases of each process.
- The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,
  - the resources currently allocated to each process,
  - the future requests and releases of each process.
- The simplest and most useful model requires that each process declare the <u>maximum number</u> of resources of each type that it may need.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation <u>state</u> to ensure that a circular-wait condition can never exist.

Deadlock Avoidance II

- Each request requires that in making this decision the system consider
  - the resources currently available,
  - the resources currently allocated to each process,
  - the future requests and releases of each process.
- The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
  - A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence for the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
  - A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence for the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.
  - In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
  - A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence for the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.
  - In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
  - When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
  - A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence for the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.
  - In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
  - When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe State I

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
  - A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence for the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.
  - In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
  - When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
  - If no such sequence exists, then the system state is said to be **unsafe**.

Safe State II

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (see Fig. 10).

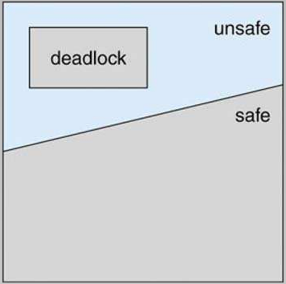


Figure: Safe, unsafe, and deadlock state spaces.

Safe State II

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (see Fig. 10).

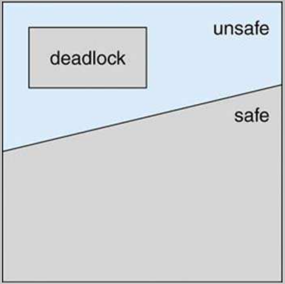


Figure: Safe, unsafe, and deadlock state spaces.

- An unsafe state may lead to a deadlock.

Safe State II

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (see Fig. 10).

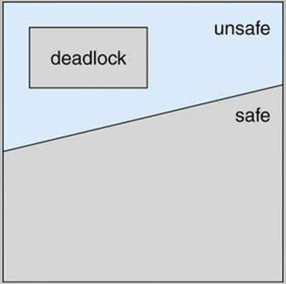


Figure: Safe, unsafe, and deadlock state spaces.

- An unsafe state may lead to a deadlock.
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

Safe State III

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2 .

P_i	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Safe State III

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2 .

P_i	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time t_0, the system is in a safe state. The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition.

Safe State III

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2 .

P_i	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time t_0, the system is in a safe state. The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition.
- A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state.

Safe State III

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2 .

P_i	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time t_0, the system is in a safe state. The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition.
- A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- Our mistake was in granting the request from process P_2 for one more tape drive.

Safe State IV

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Figure: Demonstration that the state in is safe (Upper), is not safe (Lower).

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
  - The request is granted only if the allocation leaves the system in a safe state.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
  - The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.

Safe State V

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
  - The idea is simply to ensure that the system will always remain in a safe state.
  - Initially, the system is in a safe state.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
  - The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may be lower than it would otherwise be.

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm then a deadlock situation may occur.

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm then a deadlock situation may occur.
- In this environment, the system must provide:

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm then a deadlock situation may occur.
- In this environment, the system must provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm then a deadlock situation may occur.
- In this environment, the system must provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - An algorithm to recover from the deadlock.

Single Instance of Each Resource Type I

- A *wait-for* graph.

Single Instance of Each Resource Type I

- A *wait-for* graph.
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

Single Instance of Each Resource Type I

- A *wait-for* graph.
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- For example, in Fig. 12, a resource-allocation graph and the corresponding wait-for graph are presented.

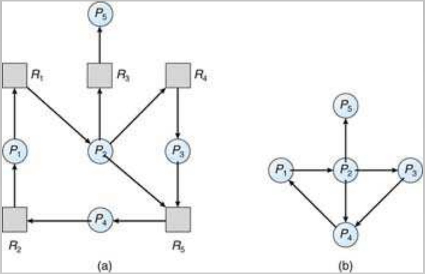


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Single Instance of Each Resource Type I

- A *wait-for* graph.
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- For example, in Fig. 12, a resource-allocation graph and the corresponding wait-for graph are presented.

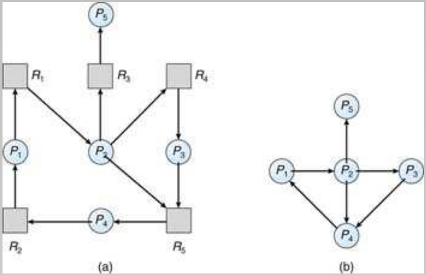


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

Single Instance of Each Resource Type II

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.

Single Instance of Each Resource Type II

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
- If this graph contains one or more cycles (knots), a deadlock exists.

Single Instance of Each Resource Type II

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
- If this graph contains one or more cycles (knots), a deadlock exists.
- Any process that is part of a cycle is deadlocked.

Single Instance of Each Resource Type II

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
- If this graph contains one or more cycles (knots), a deadlock exists.
- Any process that is part of a cycle is deadlocked.
- If no cycles exist, the system is not deadlocked.

Single Instance of Each Resource Type III

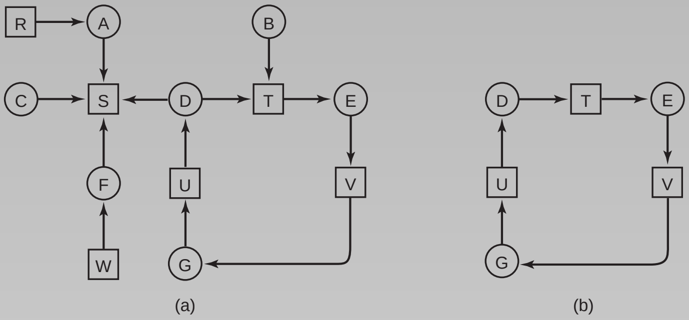


Figure: (a) A resource graph. (b) A cycle extracted from (a).

- Consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*.

Single Instance of Each Resource Type III

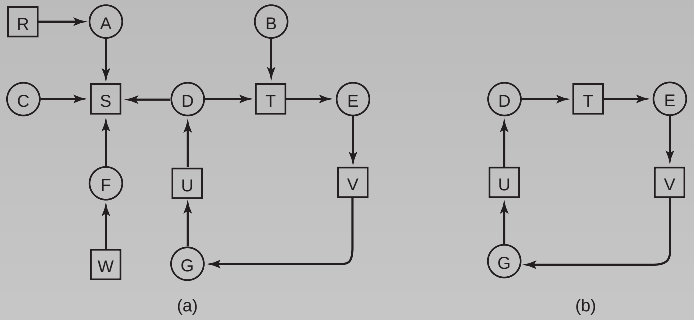


Figure: (a) A resource graph. (b) A cycle extracted from (a).

- Consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*.
- The state of which resources are known and the the resource graph is given in Fig. 13.

Single Instance of Each Resource Type III

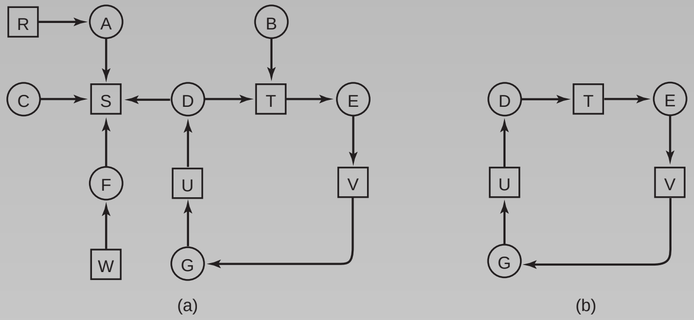


Figure: (a) A resource graph. (b) A cycle extracted from (a).

- Consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*.
- The state of which resources are known and the the resource graph is given in Fig. 13.
- The question is: "Is this system deadlocked, and if so, which processes are involved?"

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
- In the extreme, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately (considerable overhead).

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
- In the extreme, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately (considerable overhead).
- A less expensive alternative is simply to invoke the algorithm at less frequent intervals -for example, once per hour or whenever CPU utilization drops below 40 percent.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - One is simply to **abort** one or more processes to break the circular wait.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - One is simply to **abort** one or more processes to break the circular wait.
  - The other is to **preempt** some resources from one or more of the deadlocked processes.

Process Termination

- **Abort all deadlocked processes.**

Process Termination

- **Abort all deadlocked processes.**
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked.

Process Termination

- **Abort all deadlocked processes**.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked.
- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

Process Termination

- **Abort all deadlocked processes**.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked.
- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.

Process Termination

- **Abort all deadlocked processes**.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked.
- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.
- We should abort those processes whose termination will incur the minimum cost.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
    1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process?

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process?
     - *Checkpointing*; means that its state is written to a file so that it can be restarted later.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process?
     - *Checkpointing*; means that its state is written to a file so that it can be restarted later.
     - Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process?
     - *Checkpointing*; means that its state is written to a file so that it can be restarted later.
     - Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.
     - Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

Resource Preemption

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  1. **Selecting a victim**. Which resources and which processes are to be preempted? (minimum cost).
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process?
     - *Checkpointing*; means that its state is written to a file so that it can be restarted later.
     - Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.
     - Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
  3. **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Background I

- Memory is central to the operation of a modern computer system.

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,
  - to manage swapping between main memory and disk when main memory is too small to hold all the processes.

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,
  - to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory management systems can be divided into two classes:

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,
  - to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory management systems can be divided into two classes:
  1. Those that move processes back and forth between main memory and disk during execution (swapping and paging), (Memory Abstraction)

Background I

- Memory is central to the operation of a modern computer system.
- The part of the OS that manages the memory hierarchy is called the **memory manager**.
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and deallocate it when they are done,
  - to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory management systems can be divided into two classes:
  1. Those that move processes back and forth between main memory and disk during execution (swapping and paging), (Memory Abstraction)
  2. Those that do not. Simpler. (No Memory Abstraction)

Background II

- The CPU fetches instructions from memory according to the value of the program counter.

Background II

- The CPU fetches instructions from memory according to the value of the program counter.
- The memory unit sees only a stream of memory addresses;

Background II

- The CPU fetches instructions from memory according to the value of the program counter.
- The memory unit sees only a stream of memory addresses;
- It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).

Background II

- The CPU fetches instructions from memory according to the value of the program counter.
- The memory unit sees only a stream of memory addresses;
- It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Accordingly, we can ignore how a program generates a memory address.

Background II

- The CPU fetches instructions from memory according to the value of the program counter.
- The memory unit sees only a stream of memory addresses;
- It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Accordingly, we can ignore how a program generates a memory address.
- We are interested only in the sequence of memory addresses generated by the running program.

Basic Hardware I

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.

Basic Hardware I

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.

Basic Hardware I

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus.

Basic Hardware I

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus.
- Memory access may take many cycles of the CPU clock to complete (processor stalls).

Basic Hardware I

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus.
- Memory access may take many cycles of the CPU clock to complete (processor stalls).
- The remedy is to add fast memory between the CPU and main memory (**cache** memory).

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure
  - correct operation has to **protect** the *OS from access by user processes*

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure
  - correct operation has to **protect** the *OS from access by user processes*
  - and, in addition, to **protect** *user processes from one another*.

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure
  - correct operation has to **protect** the *OS from access by user processes*
  - and, in addition, to **protect** *user processes from one another*.
  - This protection must be provided by the CPU hardware.

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure
  - correct operation has to **protect** the *OS from access by user processes*
  - and, in addition, to **protect** *user processes from one another*.
  - This protection must be provided by the CPU hardware.
- **Compare every address generated in user mode with the registers.**

Basic Hardware II

- Not only we are concerned with the relative speed of accessing physical memory, but we also must ensure
  - correct operation has to **protect** the *OS from access by user processes*
  - and, in addition, to **protect** *user processes from one another*.
  - This protection must be provided by the CPU hardware.
- **Compare every address generated in user mode with the registers.**
- We first need to make sure that each process has a separate memory space.

Basic Hardware III

- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig. 14.

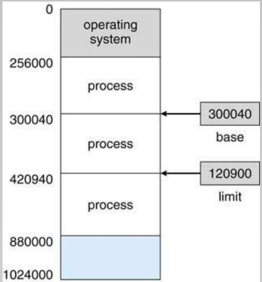


Figure: A base and a limit register define a logical address space.

Basic Hardware III

- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig. 14.
    - The **base register** holds the smallest legal physical memory address;

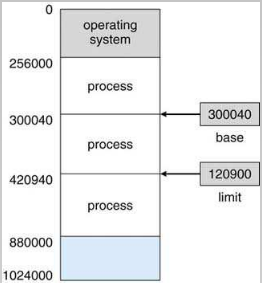


Figure: A base and a limit register define a logical address space.

Basic Hardware III

- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig. 14.
  - The **base register** holds the smallest legal physical memory address;
  - The **limit register** specifies the size of the range.

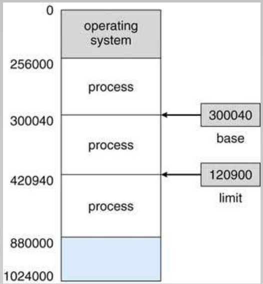


Figure: A base and a limit register define a logical address space.

Basic Hardware III

- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig. 14.
  - The **base register** holds the smallest legal physical memory address;
  - The **limit register** specifies the size of the range.
  - For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

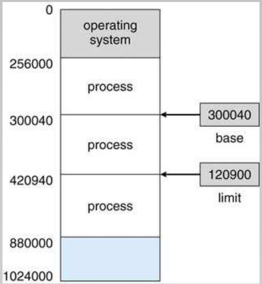


Figure: A base and a limit register define a logical address space.

Basic Hardware IV

- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the OS, which treats the attempt as a fatal error (see Fig. 15).

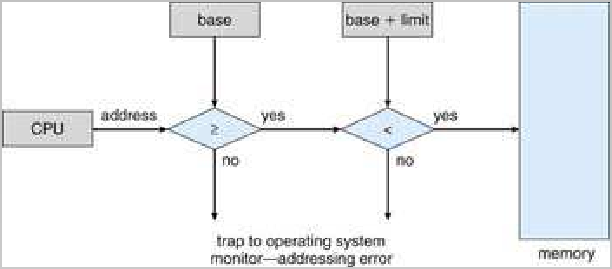


Figure: Hardware address protection with base and limit registers.

Basic Hardware IV

- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the OS, which treats the attempt as a fatal error (see Fig. 15).

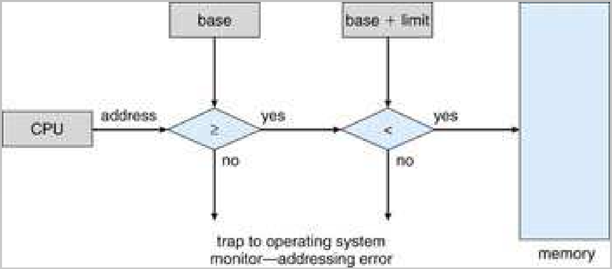


Figure: Hardware address protection with base and limit registers.

- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.