1 Deadlocks

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- **Deadlock** is defined as the *permanent* blocking of a set of processes that **compete** for system resources.

1.1 System Model

- 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. Processes obtain resources that they later release for reuse by others (processors, memory, files, devices, databases, and semaphores).
- **Consumable**: these can be created and destroyed. When a resource is acquired by a process, the resource ceases to exist (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 (also CPU) is an example of a preemptable resource.
- 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. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another.

- A process must request a resource before using it and must release the resource after using it.
  - A process may request as many resources as it requires to carry out its designated task.
  - Obviously, the number of resources requested may not exceed the total number of resources available in the system.
- 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.
- The request and release of resources are system calls. Examples are the *request*() and *release*() device, *open*() and *close*() file, and *allocate*() and $free()$ memory system calls.
- 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.
- If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.
- A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again.
- One possible way of allowing user management of resources is to associate a semaphore with each resource. Mutexes can be used equally well.
- The three steps listed above are then implemented as a down on the semaphore to acquire the resource, using the resource, and finally an up on the resource to release it. These steps are shown in Fig. 1.

```
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 1: 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.
  - Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever.
  - None of the processes can run, none of them can release any resources, and none of them can be awakened.
  - This result holds for any kind of resource, including both hardware and software.
- To illustrate a deadlock state, consider a system with three CD RW drives.
  - Suppose each of three processes holds one of these CD RW drives.
  - If each process now requests another drive, the three processes will be in a deadlock state.
  - Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes.
- Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. 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 $R1$ and process B locks record $R2$, 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 2: (a) Deadlock-free code. (b) Code with a potential deadlock.

- 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.
- 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.

1.2 Deadlock Characterization

1.2.1 Necessary Conditions

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

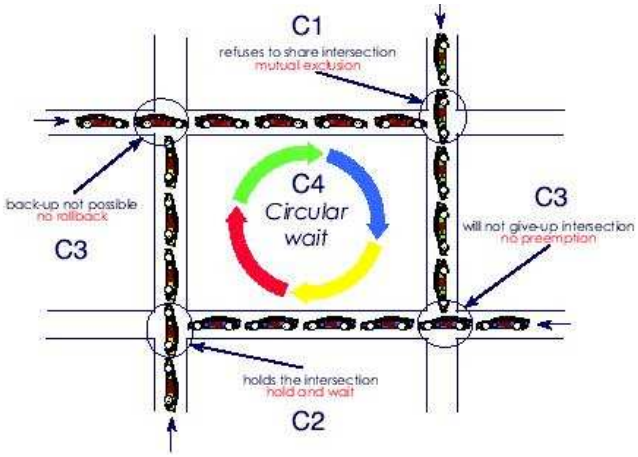

Figure 3: An example to Deadlock.

- **Mutual exclusion**. At least one resource must be held in a nonsharable mode;
  - That is, only one process at a time can use the resource.
  - If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **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.

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

1.2.2 Resource-Allocation Graph

- 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.
  - The set of vertices V is partitioned into two different types of nodes: P_is and R_is.
  - A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource (**request edge**).
  - A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i (**assignment edge**).
- Pictorially, each process P_i is represented as a circle and each resource type R_j as a rectangle.

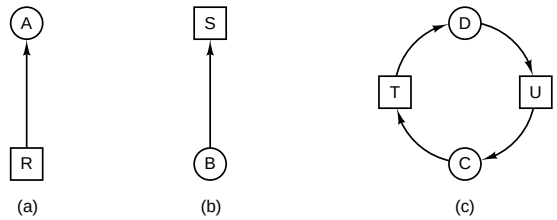

Figure 4: Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- 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).
- Since resource type R_j may have more than one instance, each such instance is represented as a dot within the rectangle.
- The resource-allocation graph shown in Fig. 5left depicts the following situation. The sets P, R, and E:

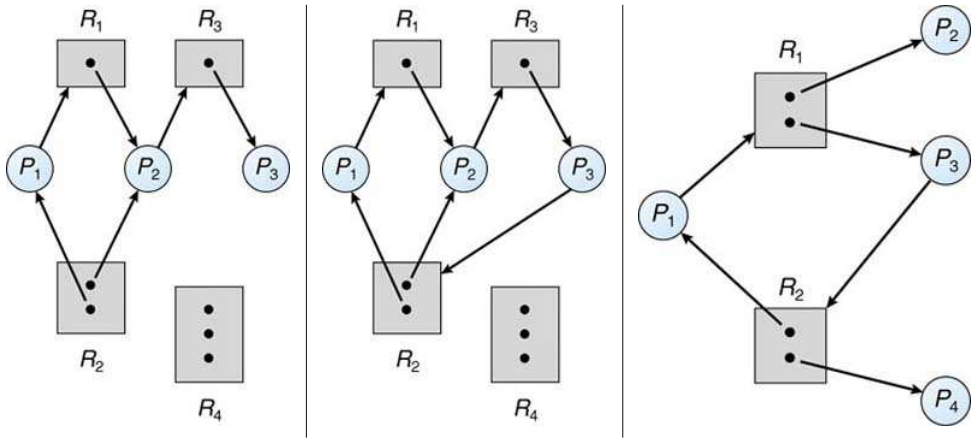

Figure 5: Left: Resource-allocation graph. Middle: Resource-allocation graph with a deadlock. Right: Resource-allocation graph with a cycle but no deadlock

  - $P = \{P_1, P_2, P_3\}$
  - $P = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- 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

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

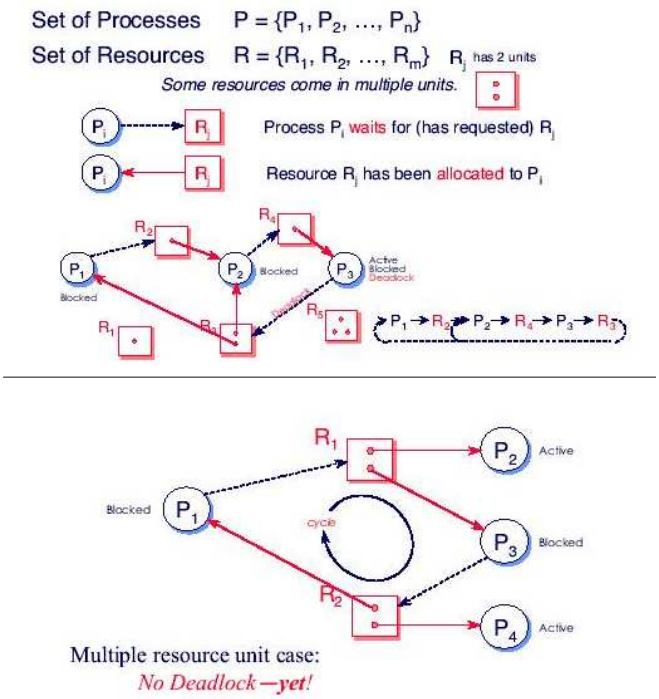

Figure 6: Resource Allocation Graphs. Lower; either P_2 or P_4 could relinquish (release) a resource allowing P_1 or P_3 (which are currently blocked) to continue.

- 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);.

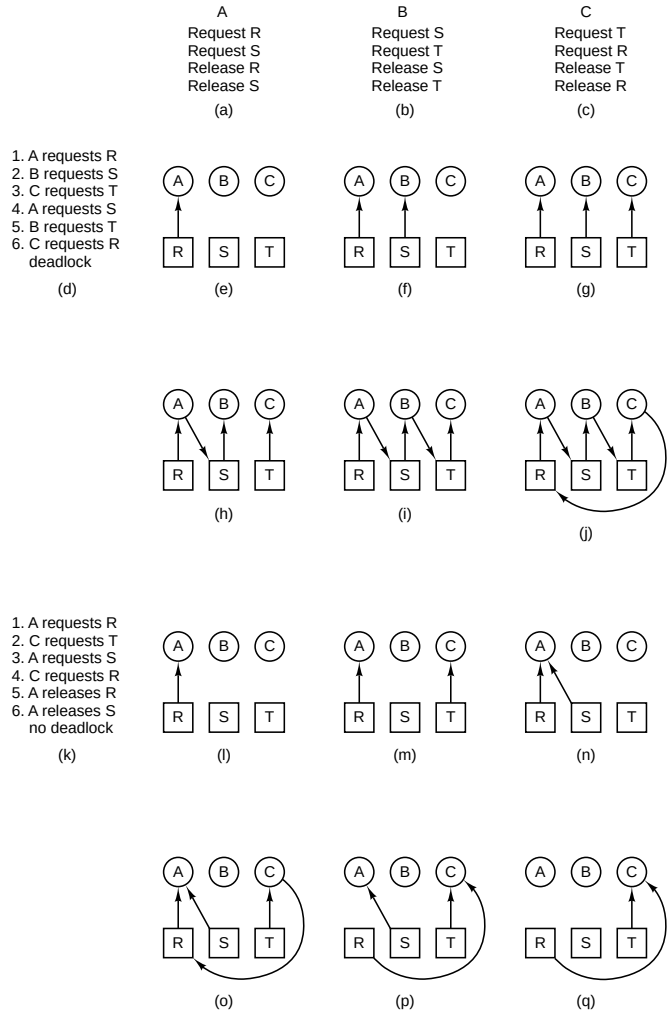

Figure 7: An example of how deadlock occurs and how it can be avoided.

1.3 Methods for Handling Deadlocks

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.

    - To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme.
    - Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions (Section 1.2.1) cannot hold.

- Design a system in such a way that the possibility of deadlock is excluded *a priori* (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. With this additional knowledge, it can decide for each request whether or not the process should wait.
- Make a decision dynamically checking whether the request will, if granted, potentially lead to a deadlock or not. (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.
   - Let deadlocks occur, detect them, and take action (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.

   - If a system neither ensures that a deadlock will never occur nor provides a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened.
   - Eventually, the system will stop functioning and will need to be restarted manually.
   - In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods.

- 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.

- 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.

- If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes.

- Thus we are faced with an unpleasant trade-off between convenience and correctness. Under these conditions, general solutions are hard to find.

1.4 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. 8.

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

Figure 8: Summary of approaches, to deadlock prevention.

1.4.1 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.

1.4.2 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.
- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- 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.
- To illustrate the difference between these two protocols, we 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. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

1.4.3 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.
- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- 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.

1.4.4 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.

1.5 Deadlock Avoidance

- Deadlock-prevention algorithms ensure that at least one of the necessary conditions for deadlock cannot occur and hence that deadlocks cannot hold.
- 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, 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.
- 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.
- Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- 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.

1.5.1 Safe State

- 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**.
- 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. 9).

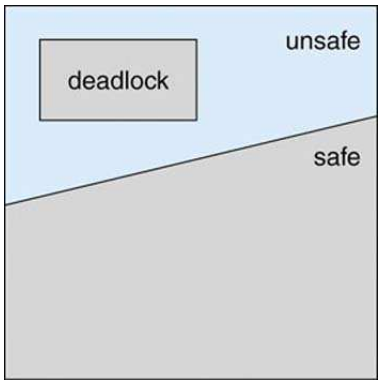

Figure 9: Safe, unsafe, and deadlock state spaces.

- An unsafe state may lead to a deadlock.

  - As long as the state is safe, the OS can avoid unsafe (and deadlocked) states.
  - In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.
  - 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.

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2 .
  - Process P_0 requires 10 tape drives, and holds 5 tape drives
  - Process P_1 may need as many as 4 tape drives, and holds 2 tape drive
  - Process P_2 may need up to 9 tape drives, and holds 2 tape drives
  - Thus, there are 3 free tape drives.

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.
  - At this point, only process P_1 can be allocated all its tape drives.
  - When it returns them, the system will have only 4 available tape drives.
  - Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process P_0 must wait.
  - Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

- Our mistake was in granting the request from process P_2 for one more tape drive.
- In Fig. 10 we have a state in which
  - A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.
  - A has 3 instances of the resource but may need as many as 9 eventually.
  - B currently has 2 and may need 4 altogether, later.
  - Similarly, C also has 2 but may need an additional 5.
  - The state of Fig. 10upper is safe because there exists a sequence of allocations that allows all processes to complete. By careful scheduling, can avoid deadlock.
  - Now suppose, this time A requests and gets another resource, giving Fig. 10lower. Eventually, B completes. At this point we are stuck.
  - We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that *guarantees* completion. A's request should not have been granted.
  - But, an unsafe state is not a deadlocked state. It is possible that A might release a resource before asking for any more, allowing C to complete and avoiding deadlock altogether.

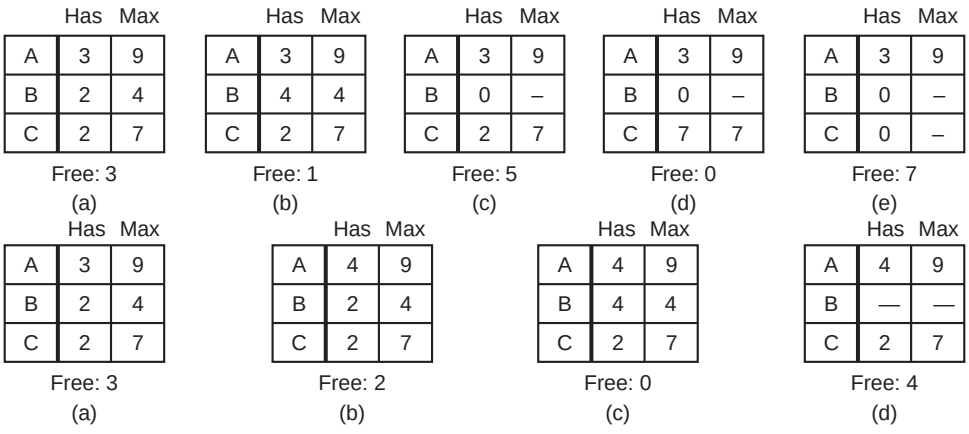

Figure 10: Demonstration that the state in is safe (Upper), is not safe (Lower).

- 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.

1.6 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.

1.6.1 Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph.
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- For example, in Fig. 11, a resource-allocation graph and the corresponding wait-for graph are presented.

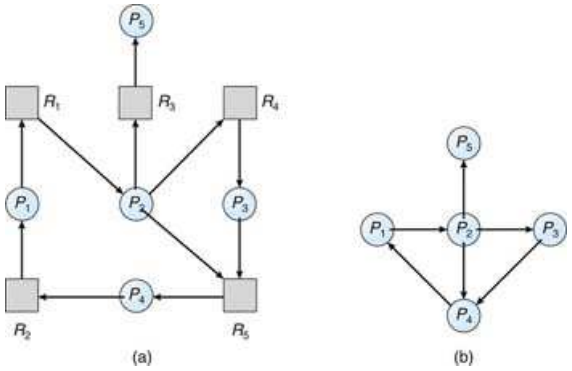

Figure 11: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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

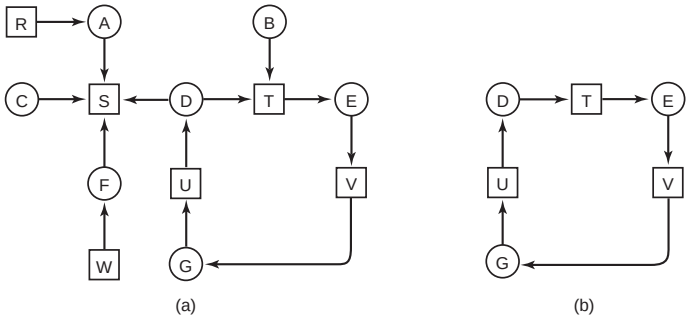

Figure 12: (a) A resource graph. (b) A cycle extracted from (a).

- 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. 12. The question is: "Is this system deadlocked, and if so, which processes are involved?"

- From this cycle, we can see that processes D, E, and G are all deadlocked. Processes A, C, and F are not deadlocked because S can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

1.6.2 Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock-detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures:
  - **Available**. A vector of length m indicates the number of available resources of each type.
  - **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
  - **Request**. An $n \times m$ matrix indicates the current request of each process.
    - If $Request[i][j]$ equals k, then process P_i is requesting k more instances of resource type R_j .
- The detection. algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let $Work$ and $Finish$ be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, \ldots, n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$, for some $i, 0 \leq i \leq n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

- Consider a system with five processes P_0 through P_4 and three resource types A, B, and C.
  - Resource type A has seven instances,
  - Resource type B has two instances,
  - Resource type C has six instances.
- Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation	Request	Available
P_i	ABC	ABC	ABC
P_0	010	000	000
P_1	200	202	
P_2	303	000	
P_3	211	100	
P_4	002	002	

- If the algorithm is executed, it will be found that the sequence $< P_0, P_2, P_3, P_1, P_4 >$ results in $Finish[i] == true$ for all i.
- Suppose now that process P_2 makes one additional request for an instance of type C.
- Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

1.6.3 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.

1.7 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.

1.7.1 Process Termination

- **Abort all deadlocked processes**. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **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 to determine whether any processes are still deadlocked.
- 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. Irrecoverable losses or erroneous results may occur, even if this is the least important process.

- 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.

1.7.2 Resource Preemption

- To eliminate deadlocks using resource preemption, we successively 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. Highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible.

  1. **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.
  2. **Rollback**. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource.
     - *Checkpointing*; means that its state is written to a file so that it can be restarted later.
     - The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process.
     - When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints.
     - 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?

2 Main memory

- Programs expand to fill the memory available to hold them. Consequently, most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage.
- It is the job of the OS to coordinate how these memories are used.
- Various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to **paging** and **segmentation** strategies.
- Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system.

2.1 Background

- 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 consists of a large array of words or bytes, each with its own address.
  - **malloc** library call
    - used to allocate memory,
    - finds sufficient contiguous memory,
    - reserves that memory,
    - returns the address of the first byte of the memory.
  - **free** library call

        * give address of the first byte of memory to free,
        * memory becomes available for reallocation.
    - Both **malloc** and **free** are implemented using the **brk** system call.

- The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

- 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.

- 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)

2.1.1 Basic Hardware

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses.

- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.

- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.

- 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).
- 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.

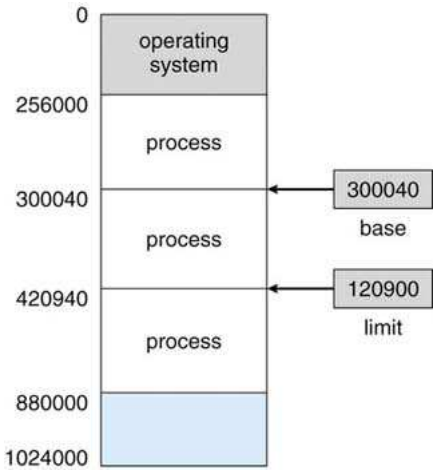

Figure 13: A base and a limit register define a logical address space.

- This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space.
- We can provide this protection by using two registers, usually a **base** and a **limit**, as illustrated in Fig. 13.
  - 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).
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

- 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. 14).
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.

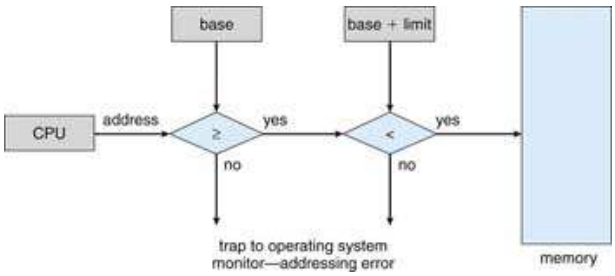

Figure 14: Hardware address protection with base and limit registers.