0.1 Deadlock Avoidance

- Deadlock avoidance, allows the necessary conditions but makes sensible choices to ensure that a deadlock-free system remains free from deadlock
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Deadlock avoidance thus requires knowledge of future requests for process resources
- Ways to avoid deadlock by careful resource allocation:
  - Resource trajectories.
  - Safe/unsafe states.
  - Dijkstra's Banker's algorithm.

0.1.1 Resource Trajectories (See Fig. 1)

- The horizontal (vertical) axis represents the number of instructions executed by process **A** (**B**)
- Every point in the diagram represents a joint state of the two processes
- If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6
- At this point, **A** is requesting the plotter and **B** is requesting the printer, and both are already assigned
- The entire box is unsafe and must not be entered
- At point t the only safe thing to do is run process **A** until it gets to I_4. Beyond that, any trajectory to u will do
- At point t **B** is requesting a resource. The system must decide whether to grant it or not
- If the grant is made, the system will enter an unsafe region and eventually deadlock

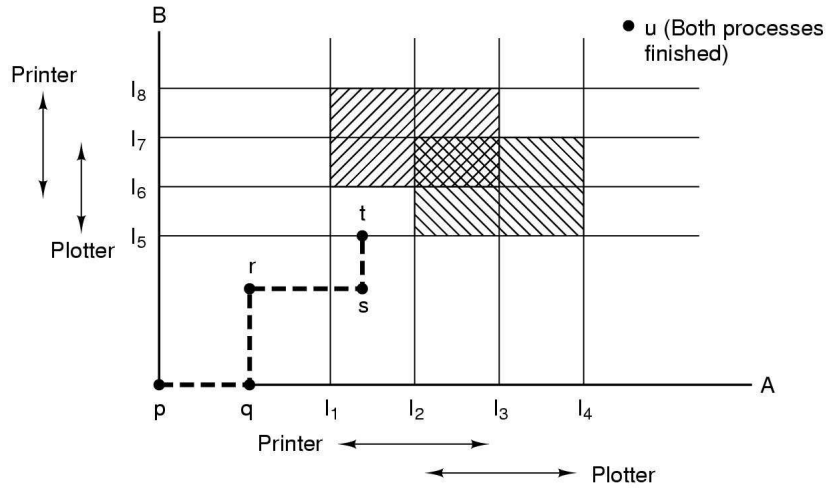

Figure 1: Two process resource trajectories.

0.1.2 Safe and Unsafe States (See Fig. 2)

- $\sum_{i=1}^{n} C_{ij} + A_j = E_j$
  - **C**: Current Allocation Matrix
  - **A**: Resources Available
  - **E**: Resources in Existence
- Add up all the instances of the resource j that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist
- At any instant of time, there is a current state consisting of **E**, **A**, **C**, and **R** (Request Matrix)
- A state is said to be **safe** if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free
- The upper state of Fig 2 is safe because there exist a sequence of allocations (scheduler runs B) that allows all processes to complete; by careful scheduling, can avoid deadlock

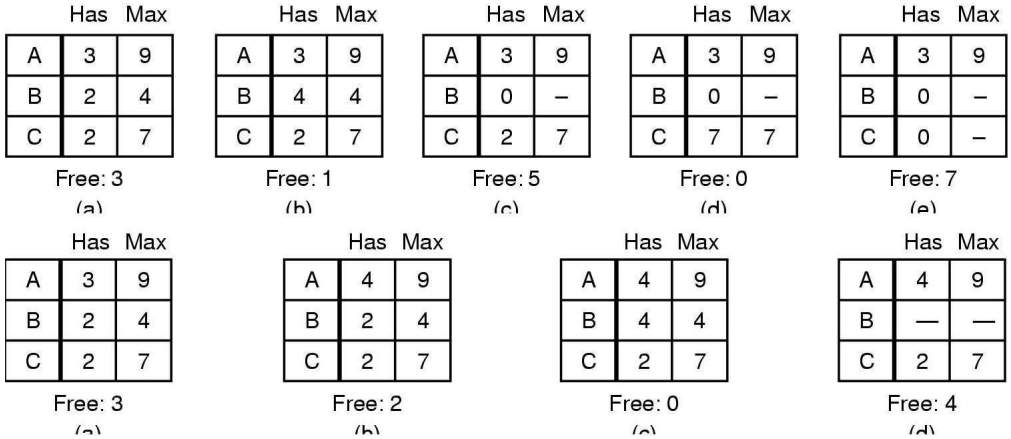

Figure 2: Demonstration that the state in is safe (upper), and in is not safe (lower).

- The lower state of Fig 2 is not safe because this time scheduler runs **A** and **A** gets another resource
- There is no sequence that guarantees completion
- An unsafe state is not a deadlock state
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes wili finish; from an unsafe state, no such guarantee can be given

0.1.3 The Banker's Algorithm for Deadlock Avoidance

- Assume **N** Processes P_i, **M** Resources R_j
- Availability vector $Avail_j$, units of each resource (initialized to maximum, changes dynamically)
- Let Max_{ij} be an $N \times M$ matrix
- $Max_{ij} = L$ means Process P_i will request at most L units of R_j
- $Hold_{ij}$ Units of R_j currently held by P_i
- $Need_{ij}$ Remaining need by P_i for units of R_j

- $Need_{ij} = Max_{ij} - Hold_{ij}$, for all i, j
- Resource Request
  - At any instance, P_i posts its request for resources in vector REQ_j (i.e., no hold-and-wait)
  - **Step 1:** verify that a process matches its needs.
    if $REQ_j > Need_{ij}$ **abort** *–error, impossible*
  - **Step 2:** check if the requested amount is available.
    if $REQ_j > Avail_j$ **goto Step 1** *–Pi must wait*
  - **Step 3:** provisional allocation (i.e., guess and check).
    $Avail_j = Avail_j - REQ_j$
    $Hold_{ij} = Hold_{ij} + REQ_j$
    $Need_{ij} = Need_{ij} - REQ_j$
    if isSafe() then grant resources (system is safe) **else** cancel allocation; **goto Step 1***–Pi must wait*
- isSafe
  - Find out whether the system is in a safe state. **Work** and **Finish** are two temporary vectors.
  - **Step 1:** initialize.
    $Work_j = Avail_j$ for all j; $Finish_i = false$ for all i
  - **Step 2:** find a process P_i such that
    $Finish_i = false$ and $Need_{ij} \leq Work_j$, for all j
    if no such process, **goto Step 4**
  - **Step 3:** $Work_j = Work_j + Hold_{ij}$
    (i.e., pretend it finishes and frees up the resources)
    $Finish_i = true$ **goto Step 2**
  - **Step 4:** **if** $Finish_i = true$ for all i
    then return true*–yes, the system is safe*
    else return false*–no, the system is NOT safe*
- What is safe?
  - Safe with respect to some resource allocation
    - very safe
      $NEED_i \leq AVAIL$ for all Processes P_i. *Processes can run to completion in any order.*

* safe (but take care)
  $NEED_i > AVAIL$ for some P_i
  $NEED_i \leq AVAIL$ for at least one P_i such that *There is at least one correct order in which the processes may complete their use of resources.*
* unsafe (deadlock inevitable)
  $NEED_i > AVAIL$ for some P_i
  $NEED_i \leq AVAIL$ for at least one P_i *But some processes cannot complete successfully.*
* deadlock
  $NEED_i > AVAIL$ for all P_i *Processes are already blocked or will become so as they request a resource.*

0.2 Deadlock Prevention

- The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded *a priori*
- Methods for preventing deadlock are of two classes:
  - *indirect methods* prevent the occurrence of one of the necessary conditions listed earlier.
  - *direct methods* prevent the occurrence of a circular wait condition.
- Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes
- Make it impossible that one of the four conditions for deadlock arise
- Mutual exclusion
  - In general, this condition cannot be disallowed
  - we can avoid assigning resources when not absolutely necessary
  - as few processes as possible should claim the resource
- Hold-and-wait
  - The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously
  - Can we require processes to request all resources at once?

  - Most processes do not statically know about the resources they need
  - Wasteful, but works

- No preemption
  - One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource
  - Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering
  - order resources by an index:$R_1, R_2, \ldots$
  - requires that resources are always requested in order
  - P_1 holds R_i and requests R_j, and P_2 holds R_j and requests R_i is impossible
  - sometimes a feasible strategy, but not generally efficient

0.3 Summary of Deadlock strategies

Table 1: Summary of Deadlock strategies

Principle	***Resource Allocation Strategy***	***Different Schemes***	***Major Advantages***	***Major Disadvantages***
DETECTION	Very liberal; grant resources as requested	Invoke periodically to test for deadlock	**1-** Never delays process initiation **2-** Facilitates on-line handling	Inherent preemption losses
PREVENTION	Conservative; undercommits resources	Requesting all resources at once	**1-**Works well for processes with single burst of activity **2-** No preemption is needed	**1-** Inefficient **2-** Delays process initiation
		Preemption	Convenient when applied to resources whose state can be saved and restored easily	**1-** Preempts more often then necessary **2-**Subject to cyclic restart
		Resource ordering	**1-** Feasible to enforce via compile time checks **2-** Needs no run-time computation	**1-** Preempts without immediate use **2-** Disallows incremental resource requests
AVOIDANCE	Selects midway between that of detection and prevention	Manipulate to find at least one safe path	No preemption necessary	**1-** Future resource requirements must be known **2-** Processes can be blocked for long periods

1 Memory Management

- The CPU fetches instructions and data of a program from memory; therefore, both the program and its data must reside in the main (RAM and ROM) memory

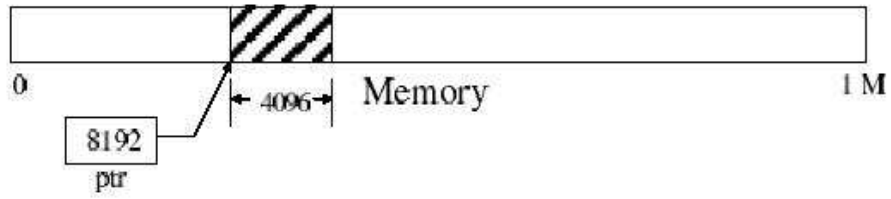

Figure 3: Allocating Memory.

- What is memory Huge linear array of storage
- **malloc** library call
  - used to allocate and free 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
- Example of allocation (see the Fig. 3)

```
char *ptr=malloc(4096); //char* is address of a single byte
```

- Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory
- A fundamental task of the **memory management** component of an operating system is to ensure safe execution of programs by providing:
  - Sharing of memory; issues are
    - *Transparency*; Several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of processes.

    - *Efficiency*; CPU utilization must be preserved and memory must be fairly allocated. Want low overheads for memory management.
    - *Relocation* Ability of a program to run in different memory locations.
  - Memory protection; processes must not corrupt each other (nor the OS!)

- Information stored in main memory can be classified in a variety of ways:
  - Program (*code*) and data (*variables, constants*).
  - Read-only (*code, constants*) and read-write (*variables*).
  - Address (*e.g., pointers*) or data (*other variables*); binding (when memory is allocated for the object): static or dynamic
  - The compiler, linker, loader and run-time libraries all cooperate to manage this information.

- Before a program can be executed by the CPU, it must go through several steps:
  - **Compiling (translating)**–generates the object code.
  - **Linking**–combines the object code into a single self-sufficient *executable code.*
  - **Loading**–copies the executable code into memory. May include run-time linking with libraries.
  - **Execution**–dynamic memory allocation.

- The process of associating program instructions and data (addresses) to physical memory addresses is called *address binding*, or *relocation*
  - **Static**–new locations are determined *before* execution
    - Compile time: The compiler or assembler translates *symbolic* addresses (e.g., variables) to *absolute* addresses.
    - Load time: The compiler translates symbolic addresses to *relative* (*relocatable*) addresses. The loader translates these to absolute addresses.
  - **Dynamic**–new locations are determined *during* execution
    - Run time: The program retains its relative addresses. The absolute addresses are generated by hardware.

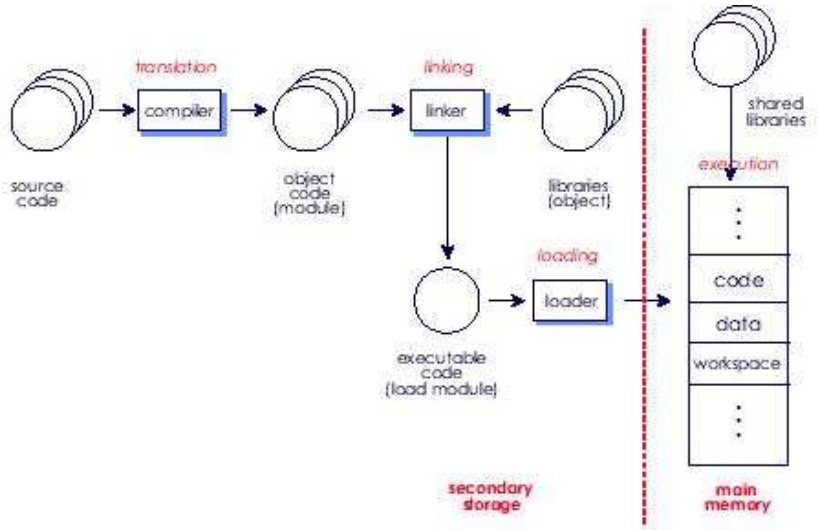

Figure 4: From source to executable code.

1.1 Basic Memory Management

- An important task of a memory management system is to bring (load) programs into main memory for execution. The following *contiguous memory allocation* techniques were commonly employed by earlier operating systems:
  - Direct placement
  - Overlays
  - Partitioning
  - Techniques similar to those listed above are still used by some modern, dedicated special-purpose operating systems and real-time systems
- Memory management systems can be divided into two classes:
  1. those that move process back and forth between main memory and disk during execution (swapping and paging)
  2. those that do not
- starting with the second one

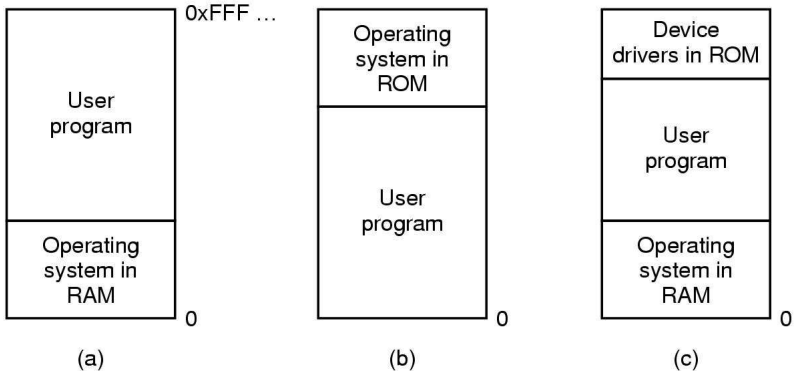

Figure 5: Three simple ways of organizing memory with an operating system and one user process.

1.1.1 Monoprogramming without Swapping or Paging (see the Fig. 5)

- The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the opreating system
- Memory allocation is trivial. No special relocation is needed, because the user programs are always loaded (one at a time) into the same memory location (*absolute loading*). The linker produces the same loading address for every user program
- Examples:Early batch monitors, MS-DOS

1.1.2 Multiprogramming with Fixed Partitions (see the Fig. 6)

- Except on simple embedded systems, monoprogramming is hardly used any more
- A simple method to accommodate several programs in memory at the same time (to support multiprogramming) is partitioning
- The easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions during system generation or startup
- When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it

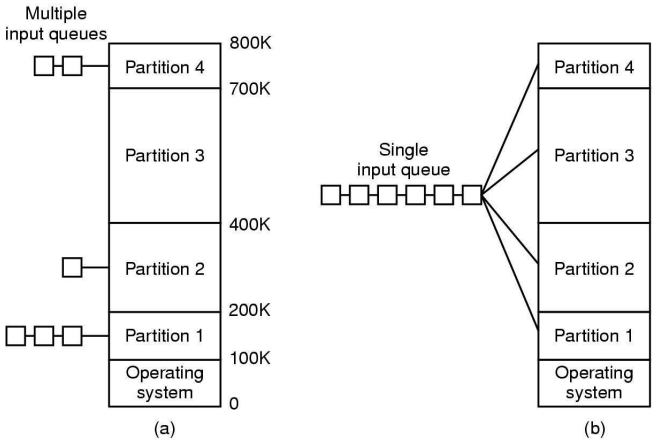

Figure 6: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queues.

- The aim of multiprogramming is to increase the CPU utilization
- $CPU\ utilization = 1 - p^n$, where n is the number of processes in the memory, p is waiting-time fraction that a process spends for I/O.
- say $p = 0.8$, means process spend 80 percent of their time waiting for I/O and n=10; then $CPU\ utilization = 89\%$, in other words CPU wasted 11 percent.

1.1.3 Relocation and Protection (see the Fig. 7)

- when a program is linked, the linker must know at what address the program will begin in the memory
- In order to provide basic protection among programs sharing the memory, partitioning techniques use a hardware capability known as *memory address mapping*, or address translation
- suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker
- if this program is loaded in partition 1 (at address 100 K, see the Fig. 6), that instruction will jump to to absolute address 100, which is inside the operating system
- what is needed is a call to 100K+100

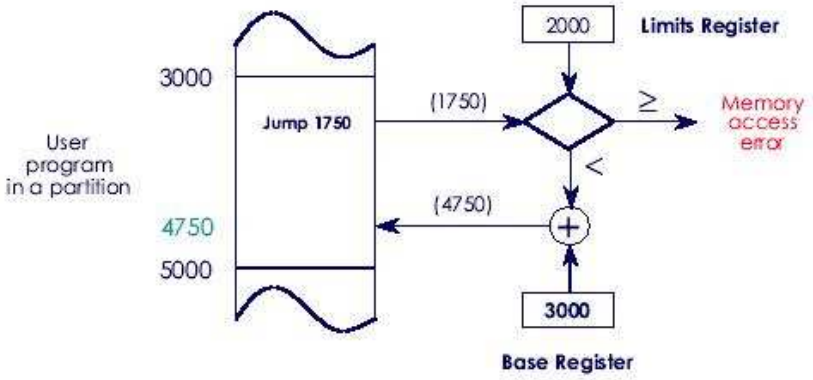

Figure 7: Address Translation.

- this problem is known as the relocation problem possible solution is is to modify the instructions as the program loaded into memory
- relocation during loading does not solve the protection problem
- A solution to both the relocation and protection problems is tp equip the mackine with two special hardware registers, called the **base** and **limits** registers

1.2 Swapping

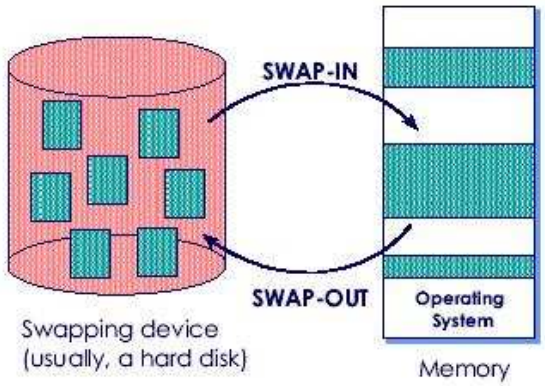

Figure 8: Swapping.

- Two general approaches to memory management can be used, depending (in part) on the available hardware
- The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk
- The other strategy, called **virtual memory**, allows programs to run even they are only partially in main memory
- The basic idea of swapping is to treat main memory as a *preemptable* resource
- A high-speed swapping device is used as the backing storage of the preempted processes
- *Fragmentation* refers to the unused memory that the memory management system cannot allocate
  - *Internal fragmentation*; Waste of memory *within* a partition, caused by the difference between the size of a partition and the process loaded. Severe in static partitioning schemes (Multiprogramming with Fixed Partitions (MFT)).
  - *External fragmentation*; Waste of memory *between* partitions, caused by scattered noncontiguous free space. Severe in dynamic partitioning schemes (Multiprogramming with Variable Partitions (MVT), swapping).
- *Compaction* (aka relocation) is a technique that is used to overcome external fragmentation
- The responsibilities of a swapper include:
  - Selection of processes to swap out *criteria*: suspended/blocked state, low priority, time spent in memory
  - Selection of processes to swap in *criteria*: time spent on swapping device, priority
  - Allocation and management of swap space on a swapping device. Swap space can be:
    * system wide
    * dedicated (e.g., swap partition or disk)