Deadlock Avoidance

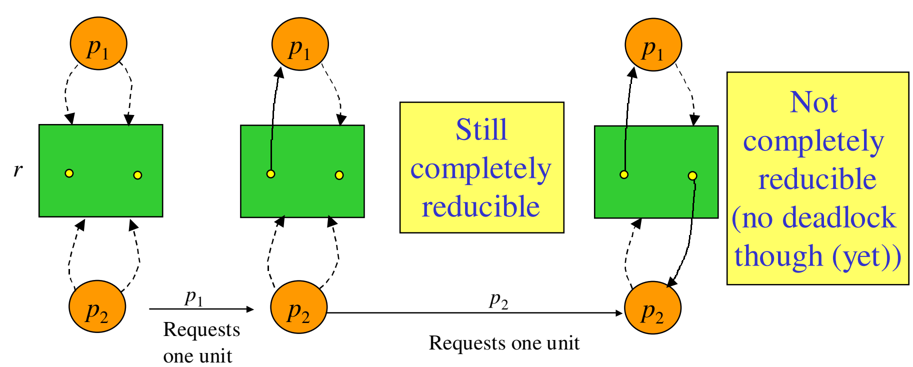
Determine whether, by allowing allocation, we could get a deadlock, i.e. check if by allowing allocation the system could enter a state that is not safe

Simple way to do it:

have each process declare the maximum number of resources of each type that it may need

Claims Graph

Claim graph: Resource graphs that also contains edges that represent *potential* requests (each process needs to specify max claim to each resource)



Banker's Algorithm (Dijkstra & Habermann)

Variables:

available[i]: units of r_i not allocated

c[a,i]: max claim of p_a on r_i

alloc[a,i]: number of units of r_i allocated to p_a

need[a,i]: number of potential requests: c[a,i] - alloc[a,i]

finish[i]: one entry per process, true if process can finish

work[i]: one entry per process

Let p_a attempt to allocate *request* units of r_i

if request > need[a,i] {error};
if request > available[i] {**wait**};
available[i] := available[i] - request;
alloc[a,i] := alloc[a,i] + request;
need[a,i] := need[a,i] - request;
if (not safe()) {**undo** request and **wait**};

safe()
  work[i] := available[i];
  while (∃ u: **not** finish[u] **and**
    (need(u,i) ≤ work[i])) {
      work[i] := work[i]+alloc[u];
      finish[u]:= true;
  }
  safe := ∀u: finish[u]

Solution: Priority Inheritance

If low-priority thread holds lock

and

A high-priority thread requests lock

then

set low-priority thread's priority to that of the high-priority thread until it gives up lock.

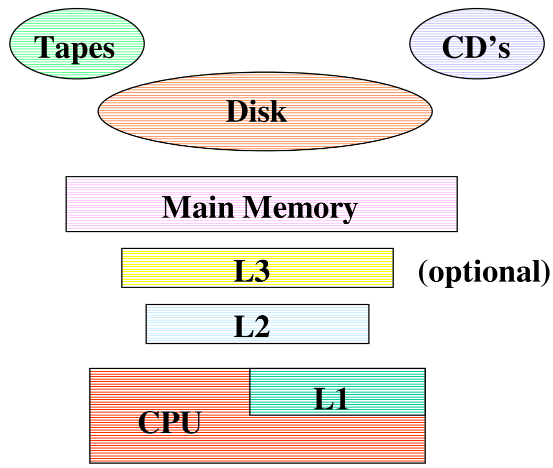
Priority Inversion

- Happens a lot
- vxWorks problems in Pathfinder
- Fundamental problem:
  - Low-priority thread holds lock, enters C.S.
  - Higher-priority thread blocks waiting for lock
  - A medium-priority thread comes along and

KA-BOOOOOM

STORAGE MANAGEMENT

The Storage Hierarchy



Properties

- Price/byte goes down as you move up the hierarchy:
  - L1 > L2 > L3 > Main memory > Disk > Tapes
- Speed goes down as well:
  - L1 > L2 > L3 > Main memory > Disk > Tapes
- Trends:
  - CPU getting much faster than storage
  - The space-time complexity is inherent

Fact

You can:

- Increase size and maintain same speed
- Increase speed and maintain same size

BUT

- Generally, you cannot do both at a reasonable cost

Programmer's View

- Memory is an array of bytes
- It is addressable in a linear fashion:
  - e.g. For a machine that has a 32-bit address space, memory is an array from 0 to $2^{32} - 1$
- Memory cannot hold values between program runs (or when machine powered down)

NonRelocatable Programs

Programs written such that:

- They must be loaded and run at some specific address or specific range, and/or
- Assume some specific region in memory will be theirs

Relocatable Programs

Programs written such that:

- They do not assume anything about where they will be loaded, and do not assume anything about where the data are placed
- A linker can prepare them to run anywhere

Example:

- UNIX, Windows NT, OS/2…

Relocatable vs. Nonrelocatable

```
ld    r0, 0x8000
add   r0, 4
ld    0x8004, r0
```

this assumes data at locations 0x8000 and 0x8004

```
ld    r0, *8000
add   r0, 4
ld    *8004, r0
```

this tells linker data should be fetched from offsets

Implementing Relocation

- Through a relocating loader
- Through a relocation register

A Relocating Loader

- Resolves the references at load time.
- More flexible, no need to know where program will run beforehand
- More overhead, because it had to be done every time the program is run

Relocation Register

- All addressing is relative to a base register
- Program can be relocated by changing the value of the base register

Example:

```
ld      r0, br(8000)
add     r0, 4
ld      br(8004), r0
```

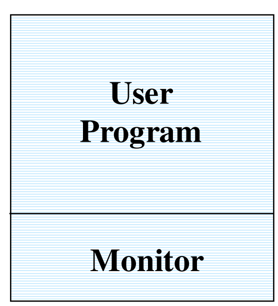
Memory Management

- Enable programmers to use memory
- Protect programs from one another
- Manage the physical resources efficiently

Early Systems

- Not much of an operating system
- Not much memory
- Program + monitor
- No protection, or relocation

Example:
DOS

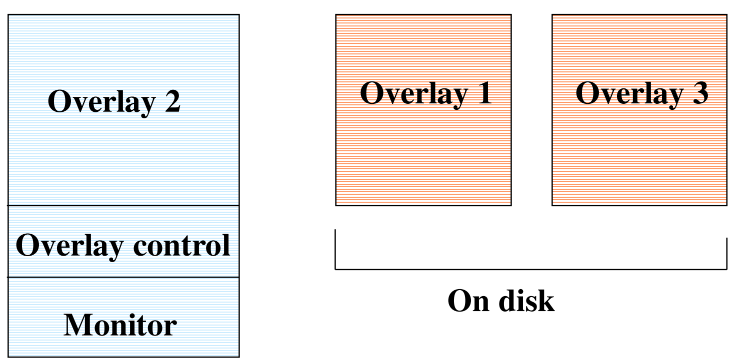


Overlaying

To overcome limited memory space:

- Program written in pieces
- After each piece finishes, it “loads” next piece and runs it
- Code in one overlay cannot access data or code in another overlay without loading it
- Can be very interesting to program (don’t try this at home!)

Example

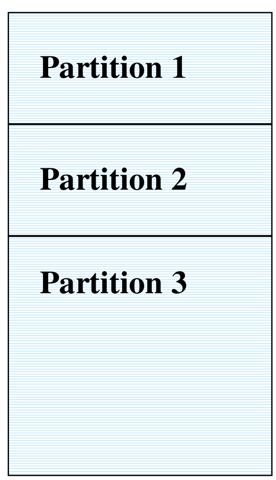


Multiple Partition Allocation

- Memory grew larger
- We want to have more programs in main memory at the same time
- Thus:
  - Create multiple partitions in main memory
  - Allocate one partition to each program

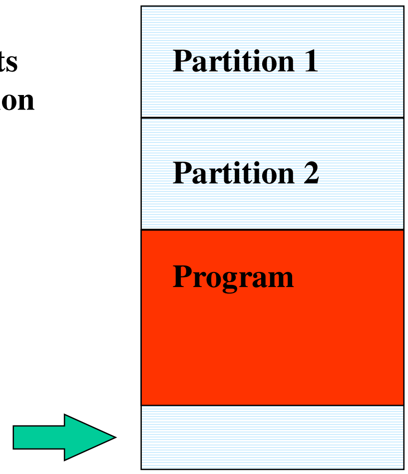
Fixed Partition Allocation

- **Divide memory into partitions of fixed sizes**
- **Load a program into a partition**
- **A program in a partition cannot access code in another partition**



Internal Fragmentation

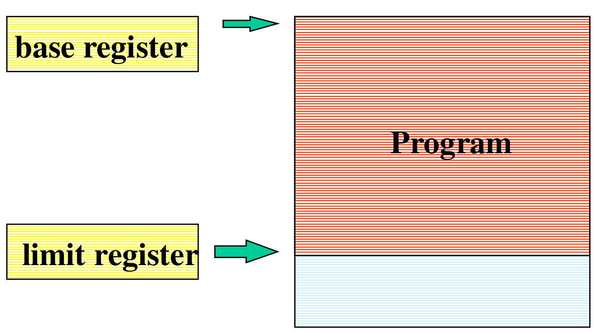
A program rarely fits exactly into a partition



This fragment is wasted

Protection: Limit Registers

In addition to the base register, we can have a limit register or a length register

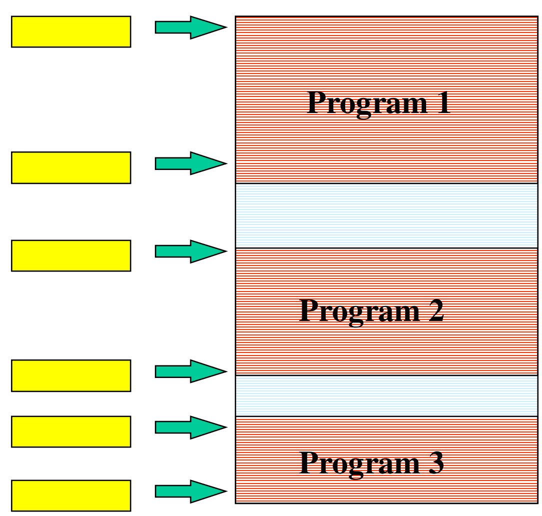


every access is checked in hardware to ensure it is within the allowable limits

Dynamic Partition Allocation

- Allocate memory depending on requirements
- Partitions adjust depending on memory size
- Requires relocatable code
  - Works best with relocation registers

Dynamic Partition Allocation



Implementation

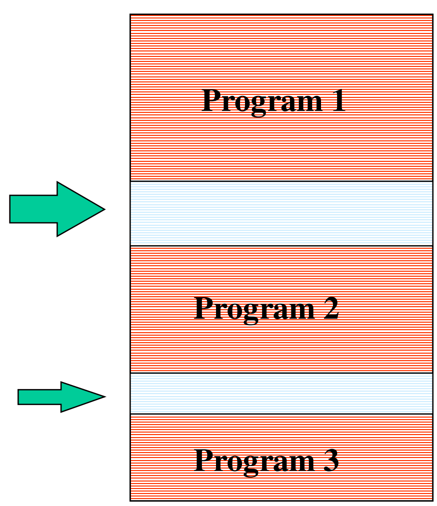
- OS maintains a list of free regions of memory (holes)
- Loads job in a hole and runs it

Selecting the hole

- first fit: allocate first hole that is large enough
- best fit: allocate smallest hole that is large enough
- worst fit: allocate largest hole
- rotating first fit: start next search from where you left off

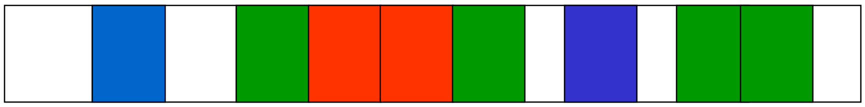
External Fragmentation

These fragments are not allocated to any program, and they are wasted.



How bad is external fragmentation? (Knuth)

H: # of holes S: # of used segments



blue: when released, H is decreased by 1
green: when released, H is unchanged
red: when released, H is increased

$S = b+g+r$ $H = (2b+y+\varepsilon)/2$

At equilibrium, $b = r$

$S = b+g+r = 2b+g = 2H$ or $H = S/2$

number of holes is half of number of segments!

Compaction (Burping)