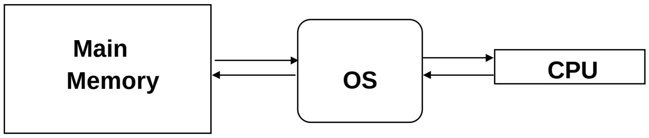
Memory Management

Chapter 7

Memory Management

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated (and de-allocated) to ensure a reasonable supply of ready processes to consume available processor time



Memory Management Requirements

- Relocation
  - – Programmer does not know where the program will be placed in memory when it is executed
  - – While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  - – Memory references must be translated in the code to actual physical memory address

Memory Management Requirements

- Protection
  - – Processes should not be able to reference memory locations in another process without permission
    - Impossible to check absolute addresses at compile time
    - Must be checked at run time
  - – Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
    - Operating system cannot anticipate all of the memory references a program will make

Memory Management Requirements

- Sharing
  - Allow several processes to access the same portion of memory
    - Better to allow each process access to the same copy of the program rather than have their own separate copy
- Logical Organization
  - Programs are written in modules
    - Modules can be written and compiled independently
    - Different degrees of protection given to modules (read-only, execute-only)
    - Share modules among processes

Memory Management Techniques

- Memory Management Techniques determine:
  - Where and how a process resides in memory
  - How addressing is performed
    - Binding:

identifiers --> compiled relative addresses
(relative to 0)
--> physical addresses

Memory Management Techniques

1) Single Contiguous
2) Overlays
3) Fixed (Static) Partitions
4) Relocation (Dynamic) Partitions
5) Paging
6) Demand Paging
7) Segmented
8) Segmented / Demand Paging

For each technique, observe:

- Algorithms
- Advantages / Disadvantages
- Special Requirements

I. Single Contiguous

```
While ( job is ready ) Do
      If ( JobSize <= MemorySize )
          Then  Begin
                Allocate Memory
                Load and Execute Job
                Deallocate Memory
                End
          Else  Error
```

I. Single Contiguous…

☺ *Advantages:*

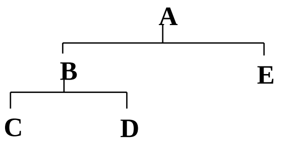
- Simplicity
- No special hardware

☹ *Disadvantages:*

- CPU wasted
- Main memory not fully used
- Limited job size

II. Overlays

- Programs can be sectioned into modules
- Not all modules need to be in main memory at the same time



- Programmer specifies which modules can overlay each other
- Linker inserts commands to invoke the loader when the modules are referenced
- The "parent" must stay in memory
- Used in DOS as an alternative to Expanded Memory.

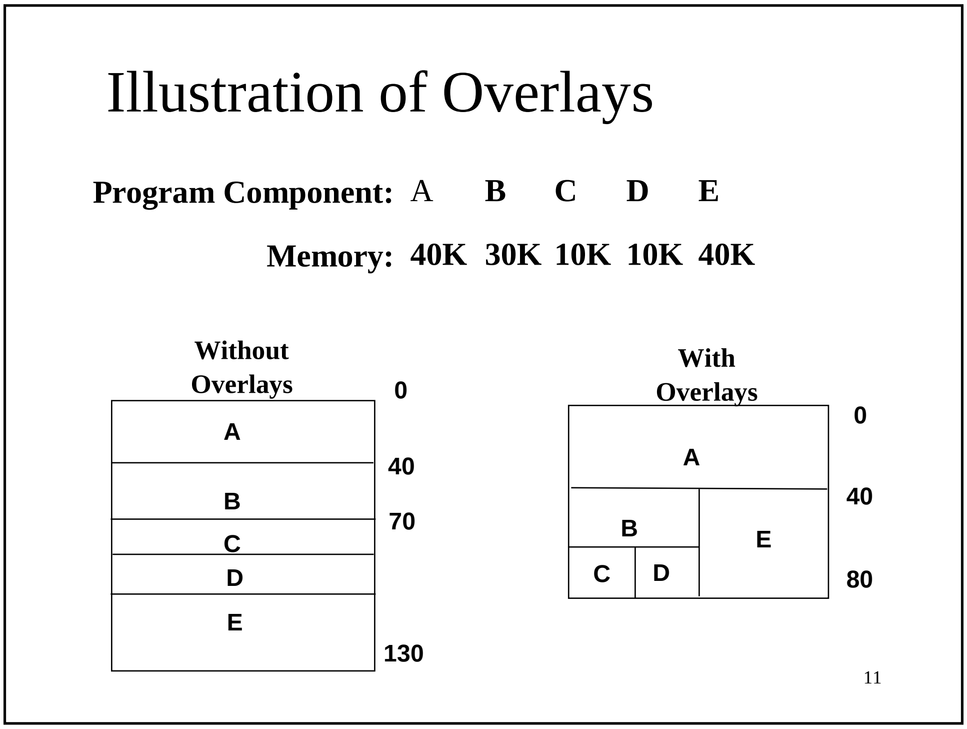
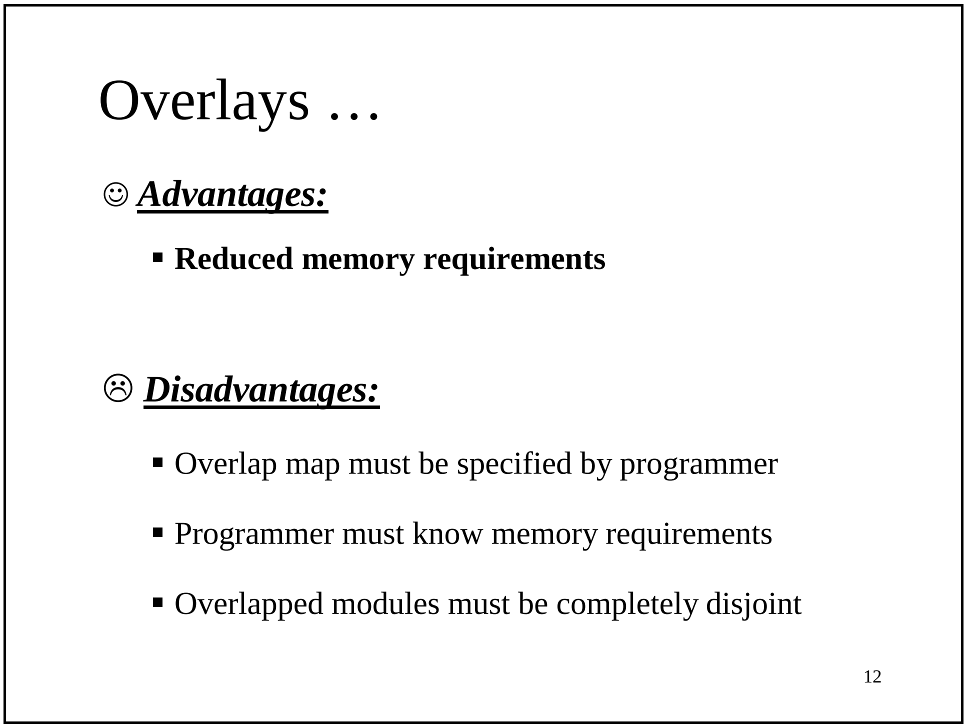


Illustration of Overlays

Program Component:	A	B	C	D	E
Memory:	40K	30K	10K	10K	40K



Overlays …

☺ ***<u>Advantages:</u>***

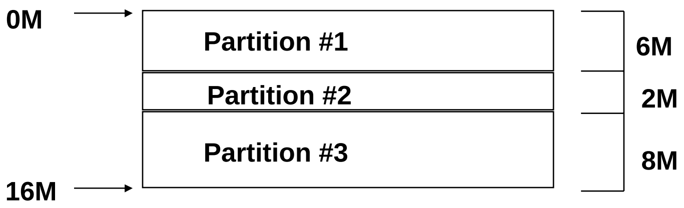
- **Reduced memory requirements**

☹ ***<u>Disadvantages:</u>***

- Overlap map must be specified by programmer
- Programmer must know memory requirements
- Overlapped modules must be completely disjoint

Fixed (Static) Partitioning with Absolute Translation

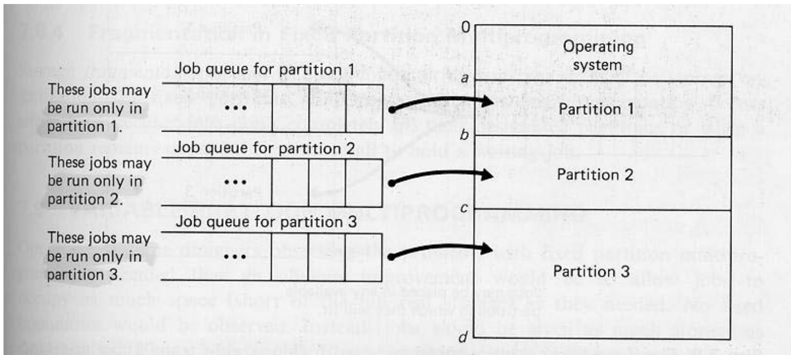
- Earliest attempt at multiprogramming
- Partition memory into fixed sized areas:



Fixed (Static) Partitioning with Absolute Translation …

- Each partition can hold ONE process
- Code generated using an ABSOLUTE address reflecting the starting address of the partition in which it is supposed to execute
  (relative to 0, 6M, or 8M in picture)
- Queue of processes waiting for each partition

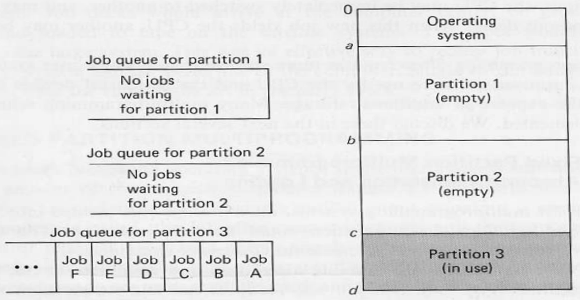
Fixed (Static) Partitioning with Absolute Translation



Job queue for partition 1
These jobs may be run only in partition 1.
Job queue for partition 2
These jobs may be run only in partition 2.
Job queue for partition 3
These jobs may be run only in partition 3.
0
a
b
c
d
Operating system
Partition 1
Partition 2
Partition 3

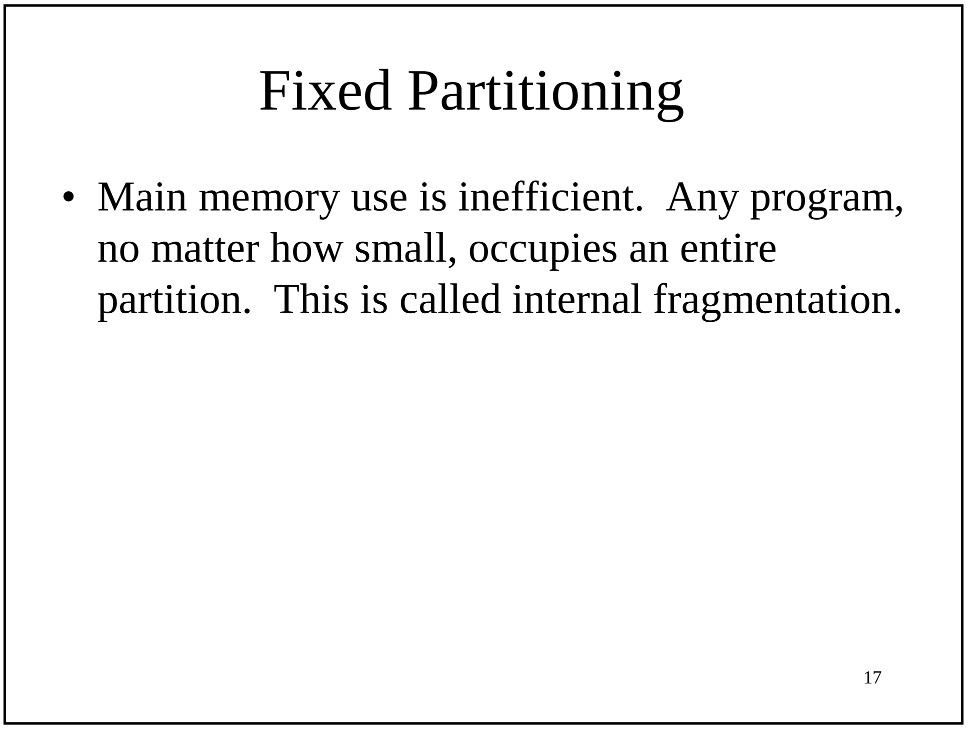
Fig. 7.6 Fixed partition multiprogramming with absolute translation and loading.

Fixed (Static) Partitioning with Absolute Translation…



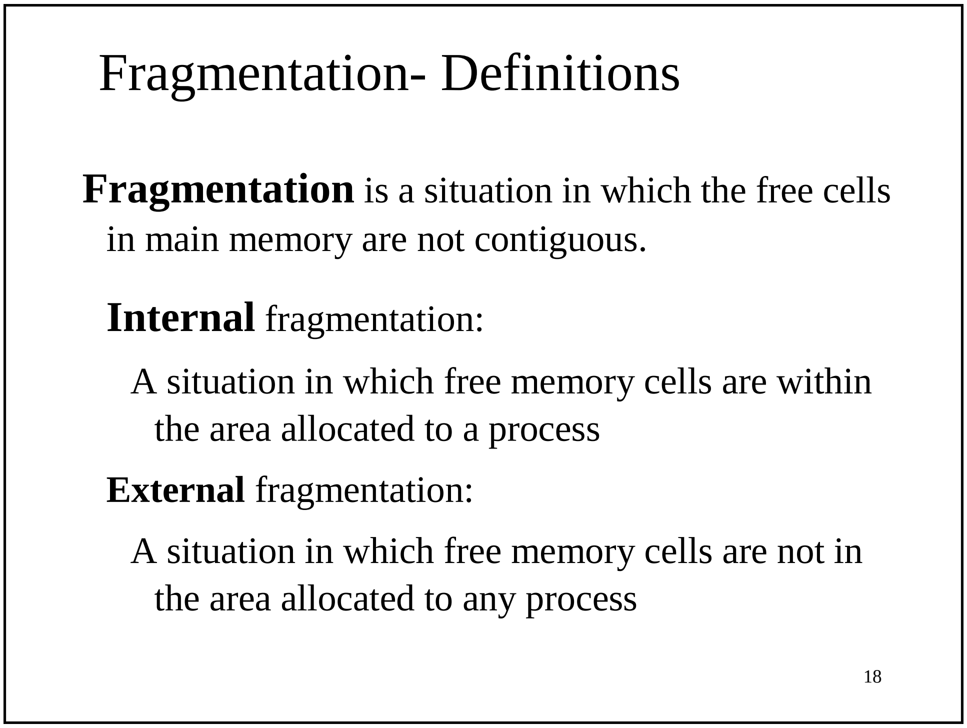
Job queue for partition 1
No jobs waiting for partition 1
Job queue for partition 2
No jobs waiting for partition 2
Job queue for partition 3
Job F
Job E
Job D
Job C
Job B
Job A
0
a
b
c
d
Operating system
Partition 1 (empty)
Partition 2 (empty)
Partition 3 (in use)

Fig. 7.7 An extreme example of poor storage utilization in fixed partition multiprogramming with absolute translation and loading. Jobs waiting for partition 3 are small and could "fit" in the other partitions. But with absolute translation and loading, these jobs may run only in partition 3. The other two partitions remain empty.



Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.



Fragmentation- Definitions

Fragmentation is a situation in which the free cells in main memory are not contiguous.

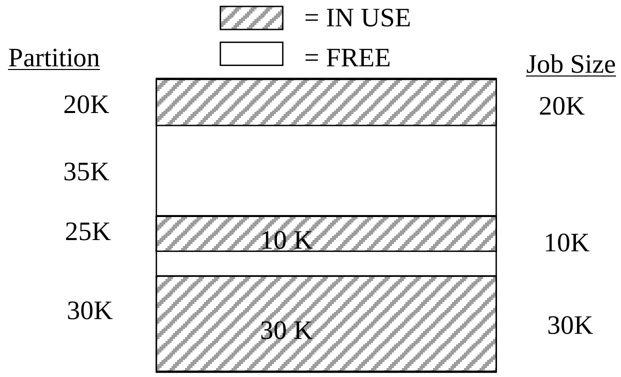
Internal fragmentation:

A situation in which free memory cells are within the area allocated to a process

External fragmentation:

A situation in which free memory cells are not in the area allocated to any process

Fixed Partition Fragmentation



▨ = IN USE

□ = FREE

Partition		Job Size
20K		20K
35K		
25K	10 K	10K
30K	30 K	30K

External fragmentation: 35K partition

Internal fragmentation: 25-10 => 15K wasted inside 25K partition

Fixed Partitioning with Absolute Translation: Pros/Cons

☺ ***Advantages:***

- Simplicity
- Multiprogramming now possible
- Works with *any* hardware (8088, 68000, etc)

Fixed Partitioning with Absolute Translation: Pros/Cons …

☹ *__Disadvantages:__*

- Job Size <= Max Partition Size <= MM Size
- Storage wasted due to *__internal fragmentation__*:

  process size < partition size
- Storage wasted due to *__external fragmentation:__*

  A partition may be idle because none of the jobs assigned to it are being run
- Once compiled a job can *only* be executed in designated partition

Fixed (Static) Partitions with __Relative__ Address Translation

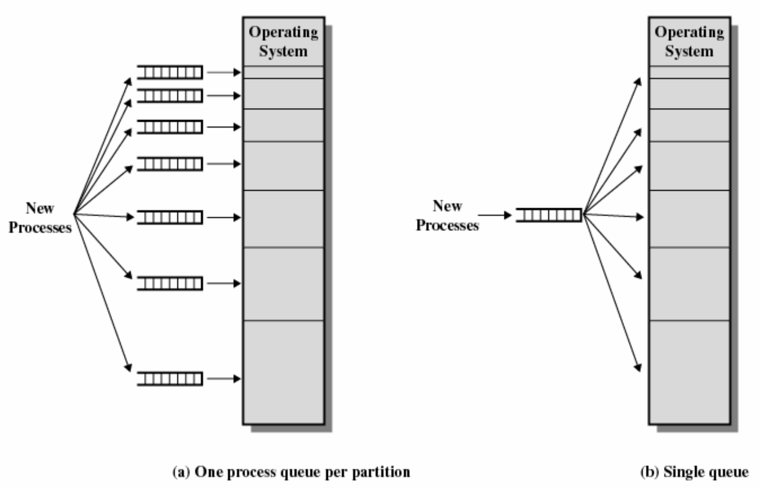
- Allows process to run in *any* free partition

- ALL Code generated using addresses *relative* to zero

Defining Partitions

- Equal-size partitions
  - Because all partitions are of equal size, it does not matter which partition is used
- Unequal-size partitions
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition

Allocating Processes to Partitions



(a) One process queue per partition

(b) Single queue

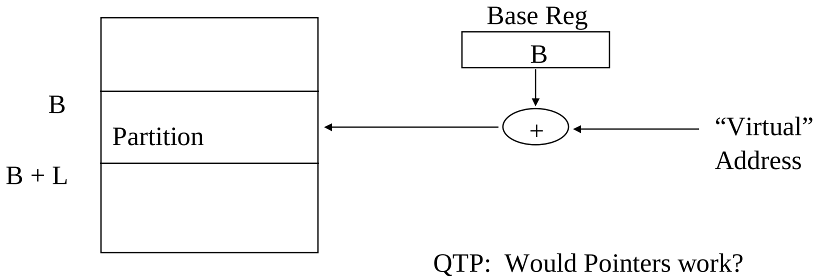
Fixed Partitions with Relative Address Translation...

Illustration:

Let:

B denote base (absolute) address of a partition

L denote partition length



QTP: Would Pointers work?

Multiprogramming Protection

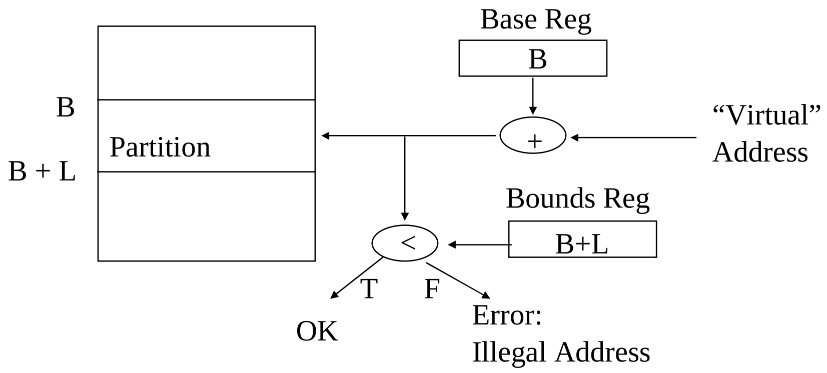
Fixed partitions with relative addressing supports multiprogramming protection

=> Ensure that one process does not access memory space dedicated to another process

Method:

Each relative address is compared to the bounds register

Multiprogramming Protection…



Fixed Partitioning with Relative Addressing: Pros/Cons

☺ *<u>Advantage compared to absolute addressing:</u>*

- Dynamic allocation of programs to partitions improves system performance

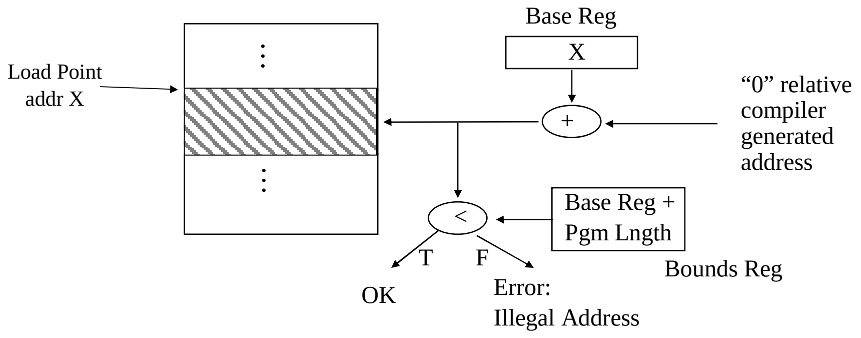
☺ *<u>Still some disadvantages:</u>*

- Partition sizes are fixed at boot time
- Can't run process larger than largest partition
- Partition selection algorithm affects system performance
- Still has internal and external fragmentation

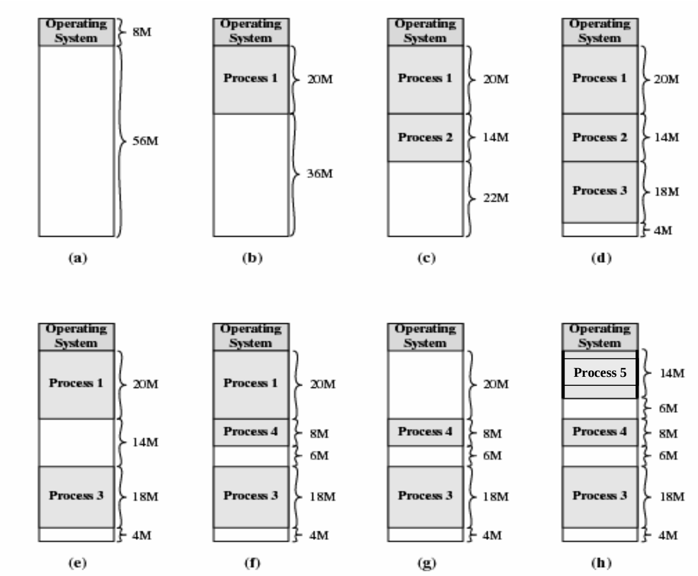
Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

Addressing Scheme in Dynamic Partitioning



Effects of Dynamic Partitioning



31

Dynamic Partitioning Placement Algorithm

Operating system must decide which free block to allocate to a process

- **Best-fit algorithm**
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often

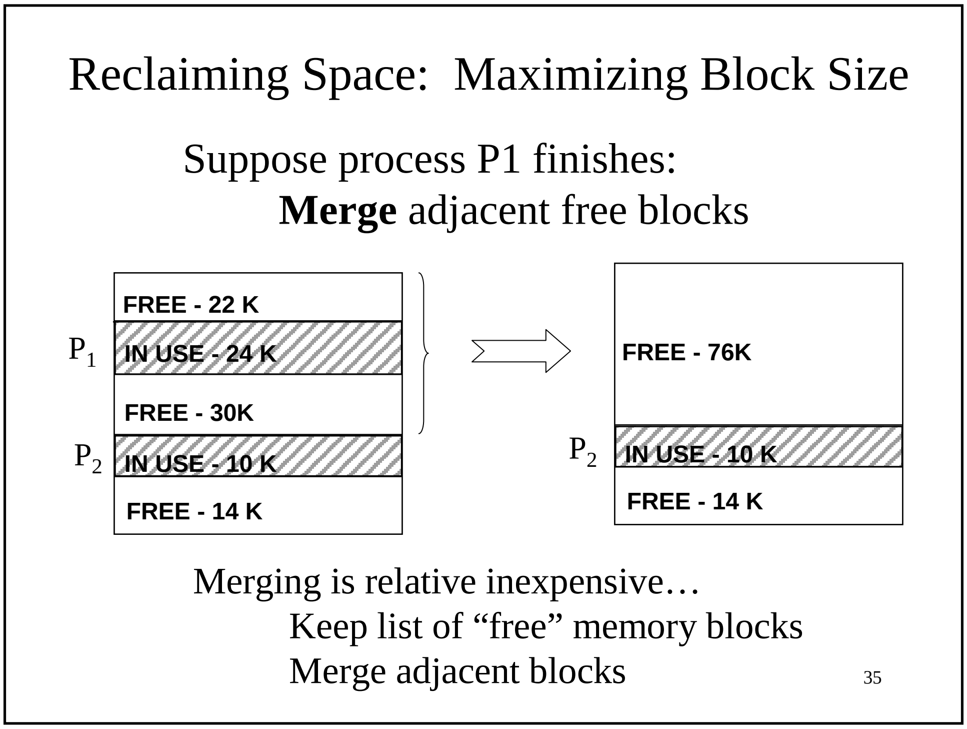
32

Dynamic Partitioning Placement Algorithm

- **First-fit algorithm**
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block

Dynamic Partitioning Placement Algorithm

- **Next-fit**
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory



Reclaiming Space: Maximizing Block Size

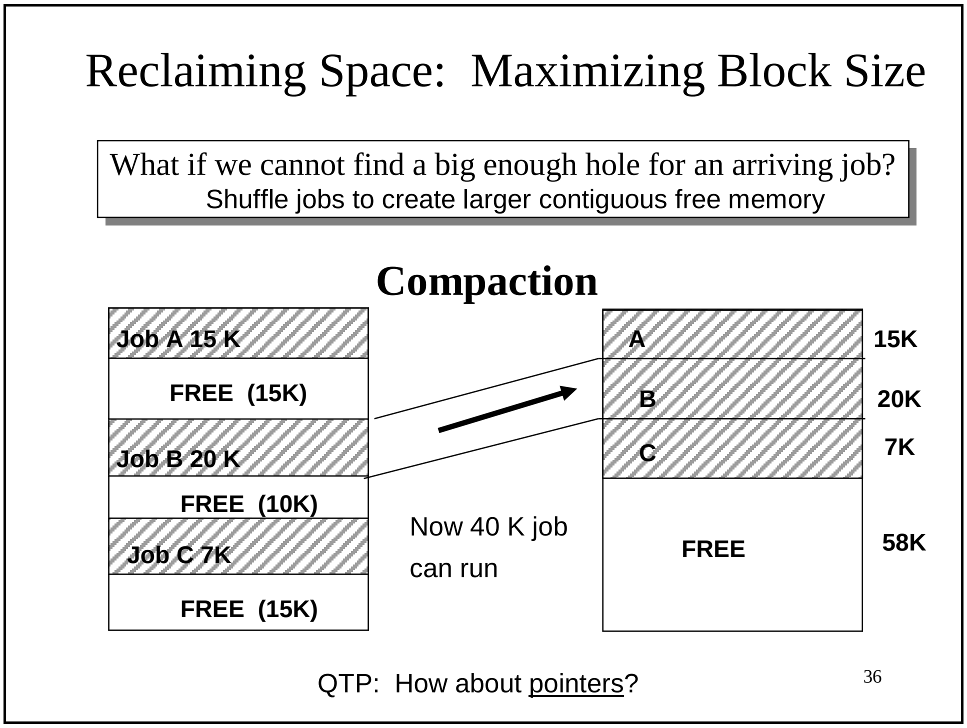
Suppose process P1 finishes:

Merge adjacent free blocks

Merging is relative inexpensive…

Keep list of “free” memory blocks

Merge adjacent blocks



Reclaiming Space: Maximizing Block Size

What if we cannot find a big enough hole for an arriving job?

Shuffle jobs to create larger contiguous free memory

Compaction

Now 40 K job can run

QTP: How about pointers?

Pros/Cons of Dynamic Partitions

☺ *<u>Advantages:</u>*

- Efficient memory usage

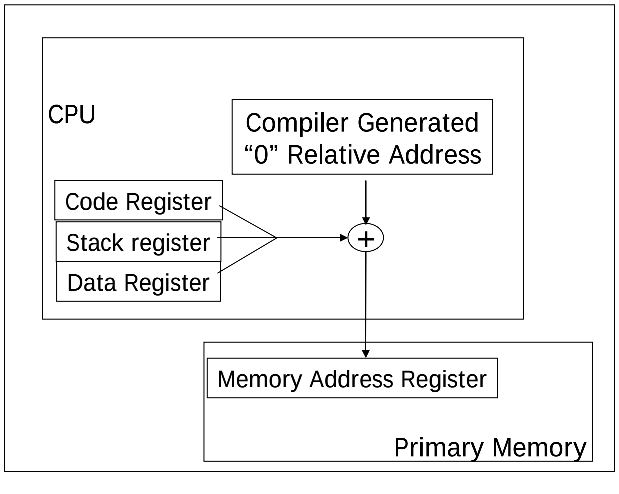
☹ *<u>Disadvantages:</u>*

- Partition Management
- Compaction *<u>or</u>* external fragmentation
- Internal fragmentation (if blocks composing partions are are always allocated in fixed sized units -- e.g. 2k)

The Move to Non-Contiguous Memory Space: **Multiple Segment Relocation Registers**

Must we have contiguous memory to run a program?

Consider:
Code
Stack
Data



CPU

Compiler Generated
"0" Relative Address

Code Register

Stack register

Data Register

+

Memory Address Register

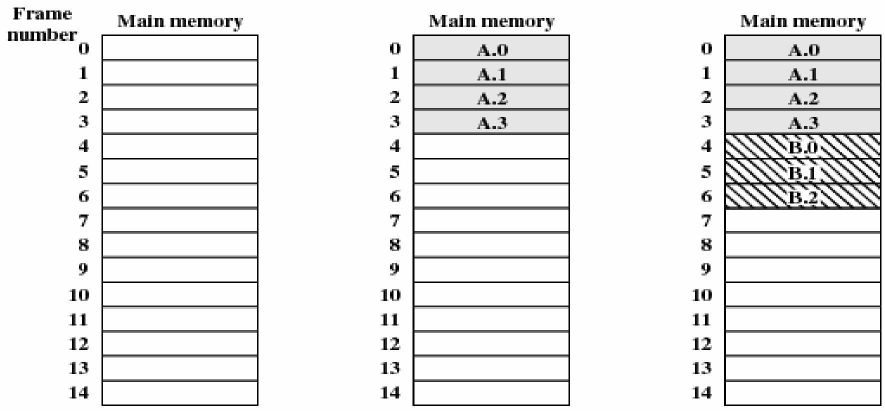
Primary Memory

An **Introduction** to Paging and Segmentation

Paging: Overview

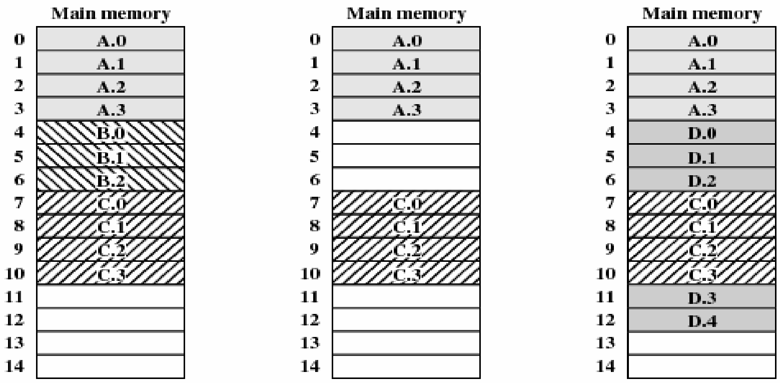
- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called **pages** and chunks of memory are called **frames**
- Operating system maintains a **page table** for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a **page number and offset** within the page

Assignment of Process Pages to Free Frames



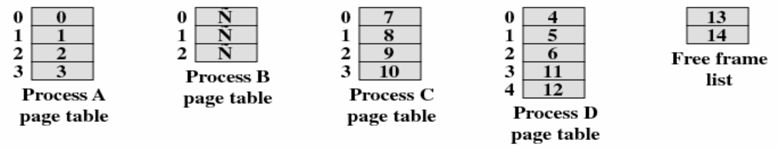
(a) Fifteen Available Frames (b) Load Process A (c) Load Process B

Assignment of Process Pages to Free Frames



(d) Load Process C (e) Swap out B (suspended) (f) Load Process D

Page Tables for Example



0	0
1	1
2	2
3	3

Process A page table

0	Ñ
1	Ñ
2	Ñ

Process B page table

0	7
1	8
2	9
3	10

Process C page table

0	4
1	5
2	6
3	11
4	12

Process D page table

13
14

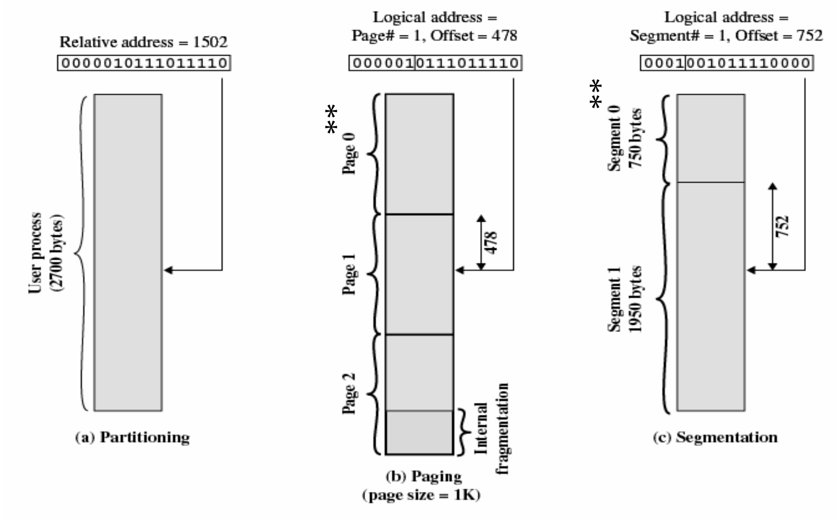
Free frame list

Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Segmentation Overview

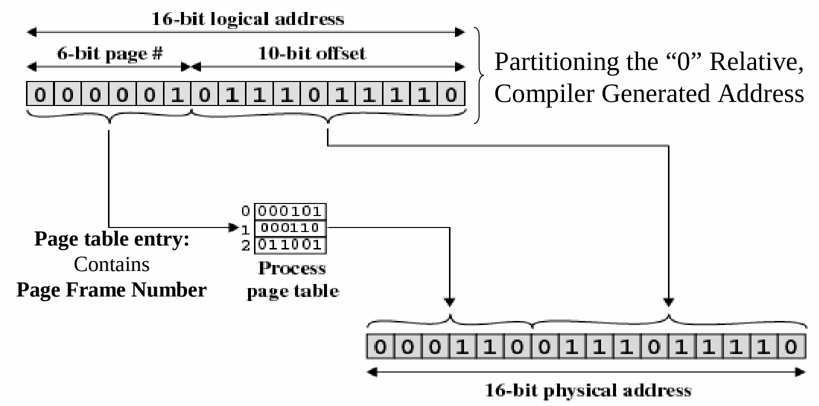
- All segments of all programs do not have to be of the same length
  - Segments usually correspond to program procedures
- There is a maximum segment length
- Addressing consist of two parts **- a segment number and an offset**
- Since segments are not equal, segmentation is similar to dynamic partitioning

Addressing Schemes



Paging:

Mapping the "0" Relative, Logical Address to a Physical Address



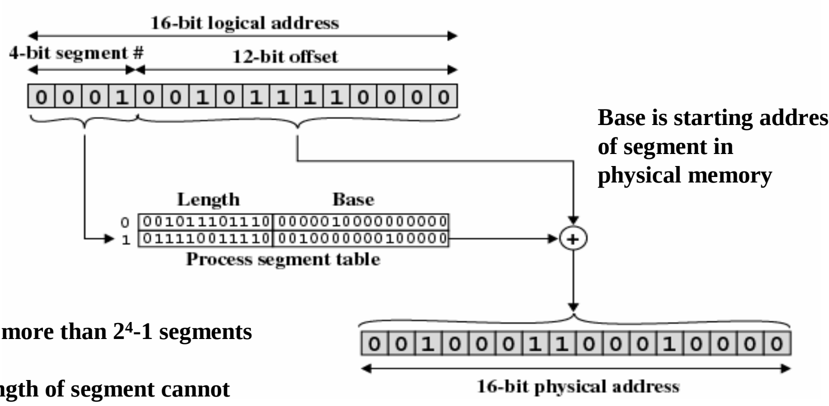
Partitioning the "0" Relative, Compiler Generated Address

Page table entry:
Contains
Page Frame Number

PFN || Offset == PFN * 2^{10} + Offset

Segmentation:

Mapping the “0” Relative, Logical Address to a Physical Address



Process segment table

	Length	Base
0	001011101110	0000010000000000
1	011110011110	0010000000100000

Base is starting address of segment in physical memory

No more than 2^4-1 segments

Length of segment cannot be larger than 2^{12}