Chap.8
Memory Management

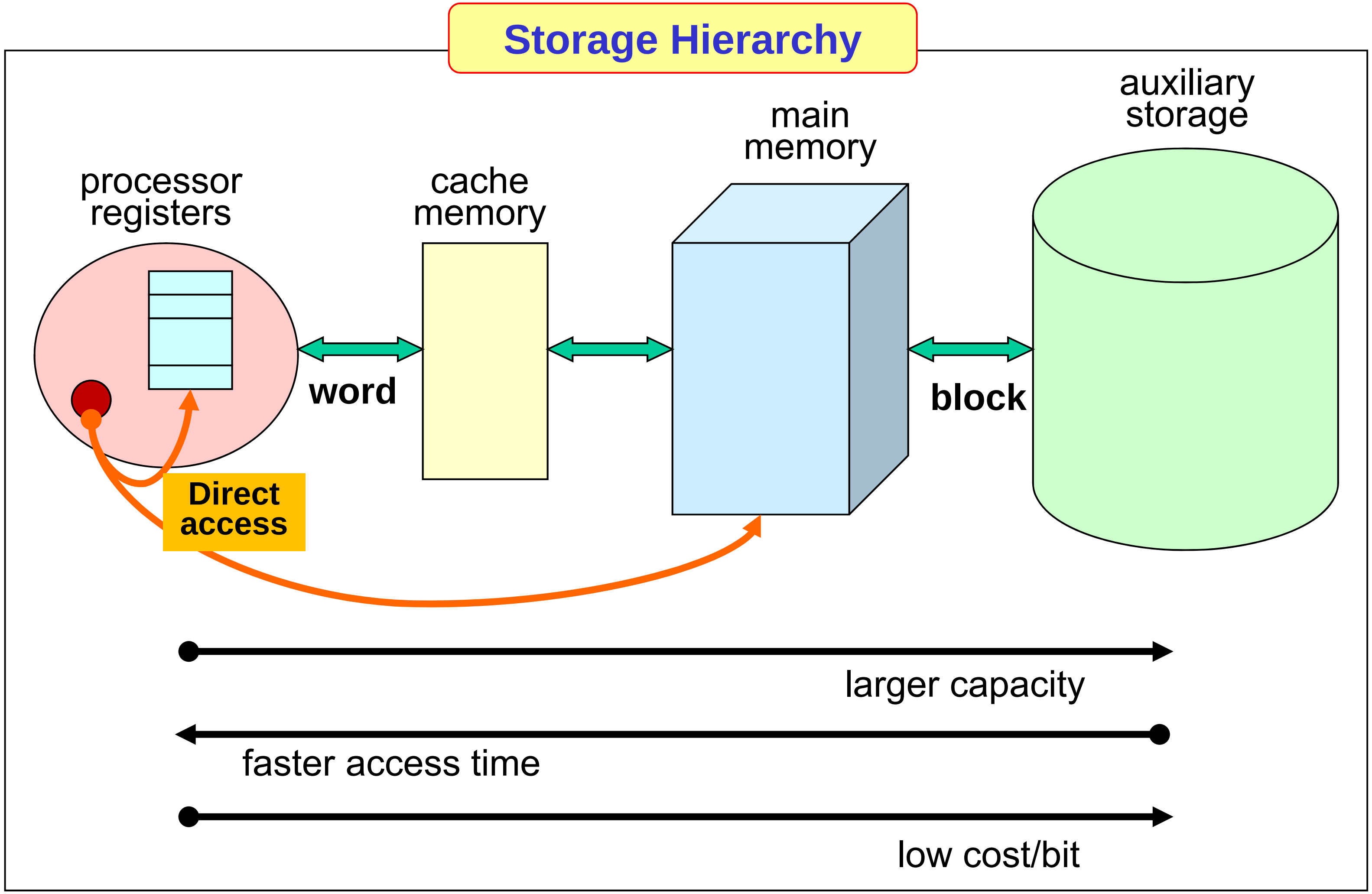
Young Ik Eom
DCLab@SKKU

Backgrounds

- **Types of memories in computer systems**
  - Processor registers
  - Cache memory
  - **Main memory**
  - **Auxiliary memory**

- **Notes**
  - Block
    - ✓ Data transfer unit between primary memory and secondary storage
    - ✓ Size: 1 ~ 4 KB (128B ~ 8MB)
  - Word
    - ✓ Data transfer unit between primary memory and CPU
    - ✓ Size: 16 ~ 64 bits

Backgrounds



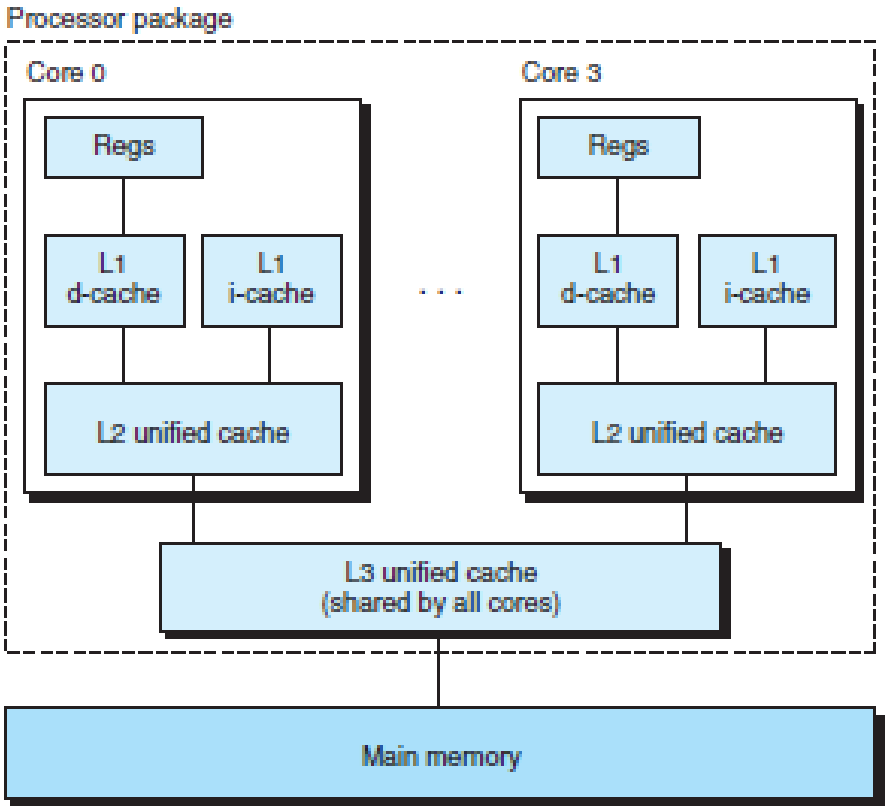
Backgrounds

Cache

- CPU register access
  - Generally takes 1~30 cycles of the CPU clock
- Memory access
  - Generally takes 50~200 cycles of the CPU clock
  - CPU normally needs to stall during the memory access
  - Intolerable because of the frequency of memory accesses
- Cache memory
  - Used to accommodate the speed differential

Backgrounds

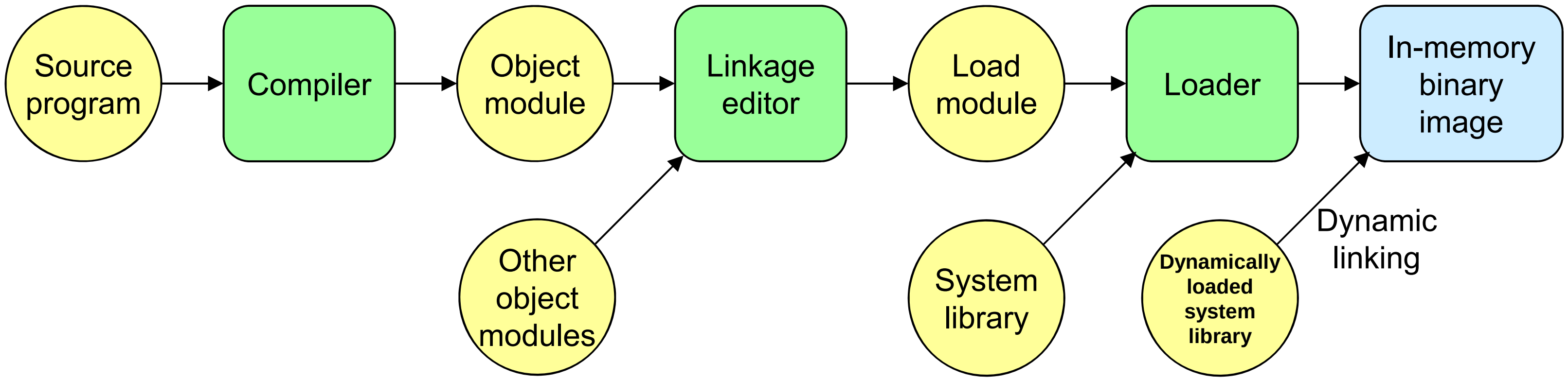
- **Cache**
  - Intel Core i7 cache



R. E. Bryant and D. R. O'Hallaron., Computer Systems, 2-ed., Pearson, 2011.

Backgrounds

Address binding



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Backgrounds

- **Address binding**
  - Compile time binding
    - When it is known at compile time where the process will reside in memory, then **absolute code** can be generated
    - Changing the starting location requires recompilation
    - MS-DOS .COM-format programs

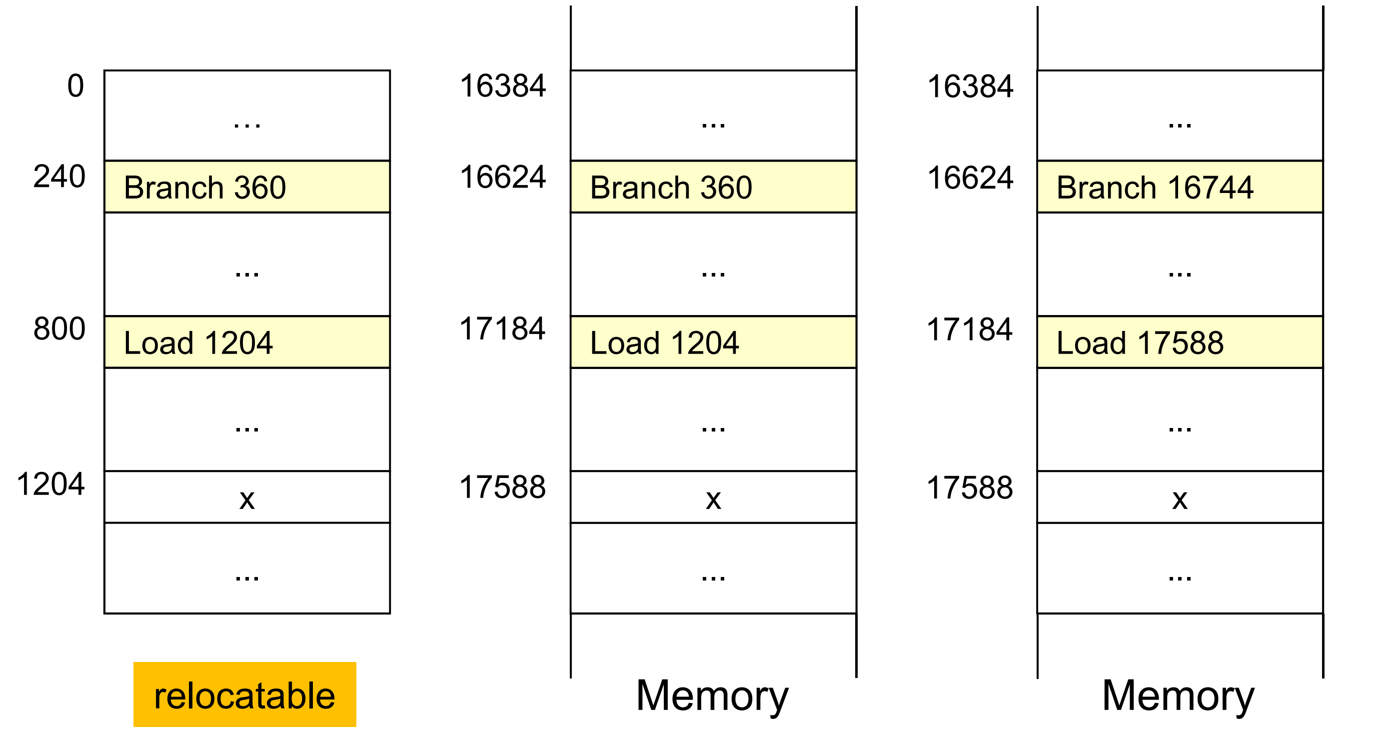
Backgrounds

- **Address binding**
  - Load time binding
    - When it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**
    - Final binding is delayed until load time
    - Changing the starting location requires reloading or **relocation** of the user code

Backgrounds

Load-time binding

(Allocation address = 16384)



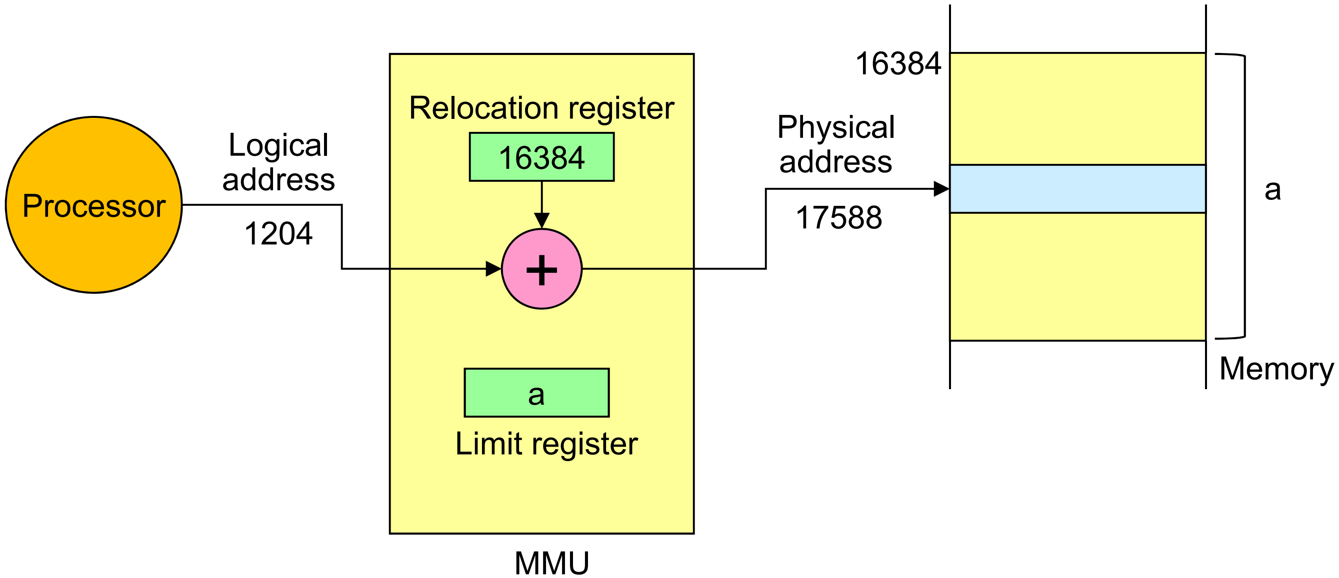
(a) program code (b) after loading (c) after **relocation**

Backgrounds

- **Address binding**
  - Run time binding
    - ✓ Processes can be moved during its execution from one memory segment to another
    - ✓ Special hardware is required
    - ✓ Used in most general-purpose operating systems

Backgrounds

- **Address binding**
  - Run time binding
    - ✓ Run-time mapping from logical(virtual) address to physical address is performed by a hardware device called MMU(Memory Management Unit)



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Backgrounds

Address binding

- Notes
  - Logical address (virtual address)
    - An address generated by the CPU
  - Physical address
    - An address seen by the memory unit
    - An address loaded into MAR

  - Relocation register (base register)
    - Holds the allocation address (smallest physical address)
  - Limit register
    - Contains the range of logical addresses

Backgrounds

- **Dynamic loading**
  - All routines are kept on disk in a relocatable load format
  - A routine is not loaded until it is called
    - ✓ When a routine needs to call another routine, the caller first checks to see whether the callee has been loaded
    - ✓ Calls the **relocatable linking loader** to load the callee into memory, if necessary
  - Advantages
    - ✓ Better memory space utilization
    - ✓ Unused routine is never loaded
    - ✓ Does not require special support from OS

Backgrounds

- **Dynamic linking**
  - Linking is postponed until execution time
  - Usually used with system libraries
    - ✓ Without this facility, each executable on a system must include a copy of its library
      - Wastes both disk space and main memory

Backgrounds

- **Dynamic linking**
  - Uses **stub** concept
    - ✓ Included in the image for each library routine reference
    - ✓ Small piece of code that indicates how to locate or load the appropriate library routine (in memory or from the library)
    - ✓ Replaces itself with the address of the routine and executes the routine
      - At the next time that the code segment is reached, the library routine is executed directly without further dynamic linking

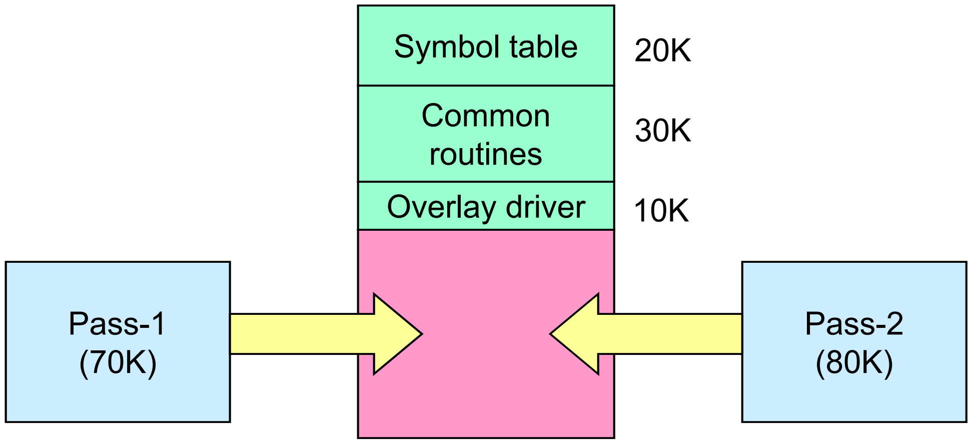
Backgrounds

- **Dynamic linking**
  - Code sharing
    - ✓ All processes that use a library execute only one copy of the library code
  - Supports the concept of shared libraries
  - Requires help from the OS

Backgrounds

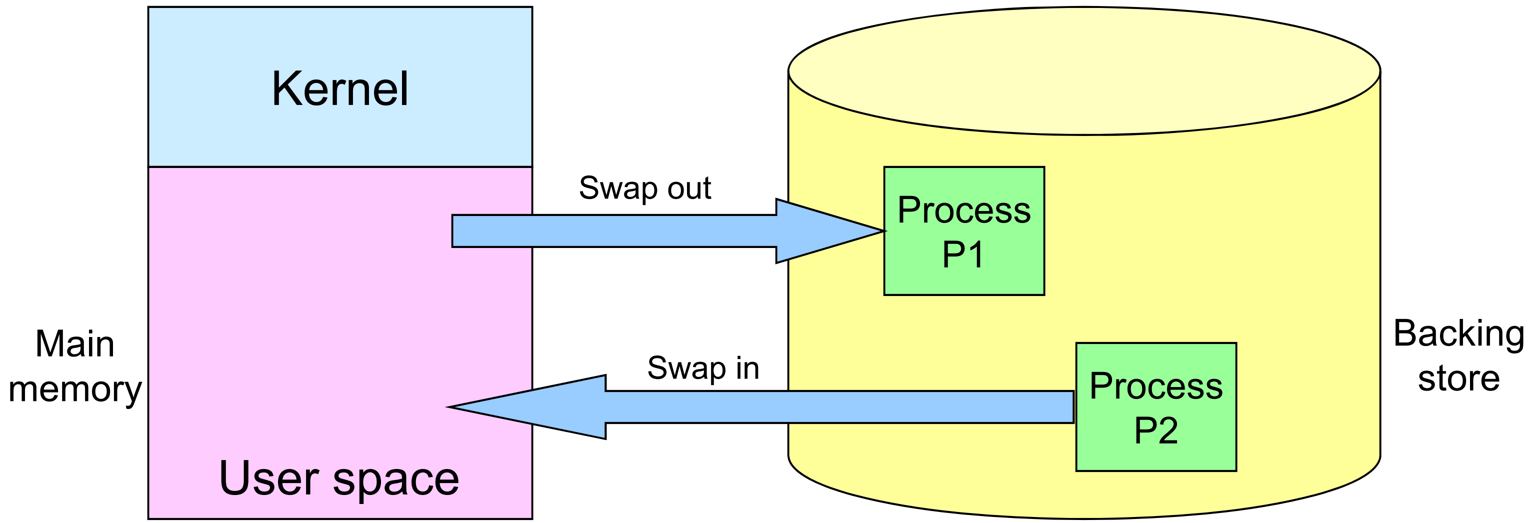
Overlay structure

- Keep in memory only those instructions and data that are needed at any given time
- Example) Assembler
  - ✓ Symbol table(20KB), Common routines(30KB), Pass-1(70KB), Pass-2(80KB)
  - ✓ Primary memory size: 150KB



Swapping

- **Swapping**
  - A process can be swapped temporarily out of memory to a **backing store** (**swap device**)



Swapping

- **Notes on swapping**
  - Time quantum vs swap time
    - ✓ Time quantum should be substantially larger than swap time (context switch time) for efficient CPU utilization
  - Memory areas to be swapped out
    - ✓ Swap only what is actually used
  - Pending I/O
    - ✓ If the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped
    - ✓ Solutions
      - Never swap a process with pending I/O
      - Execute I/O operations only into kernel buffers (and deliver it to the process memory when the process is swapped in)

Swapping

- **Notes on swapping**
  - Swap device (swap file system)
    - ✓ Swap space is allocated as a chunk of disk
      - Contiguous allocation of the process image
      - Fast access time

> ▪ Ordinary file system
> • Discontiguous allocation of the file data blocks
> • Focuses on the efficient file system space management

Contiguous Memory Allocation

Contiguous Memory Allocation

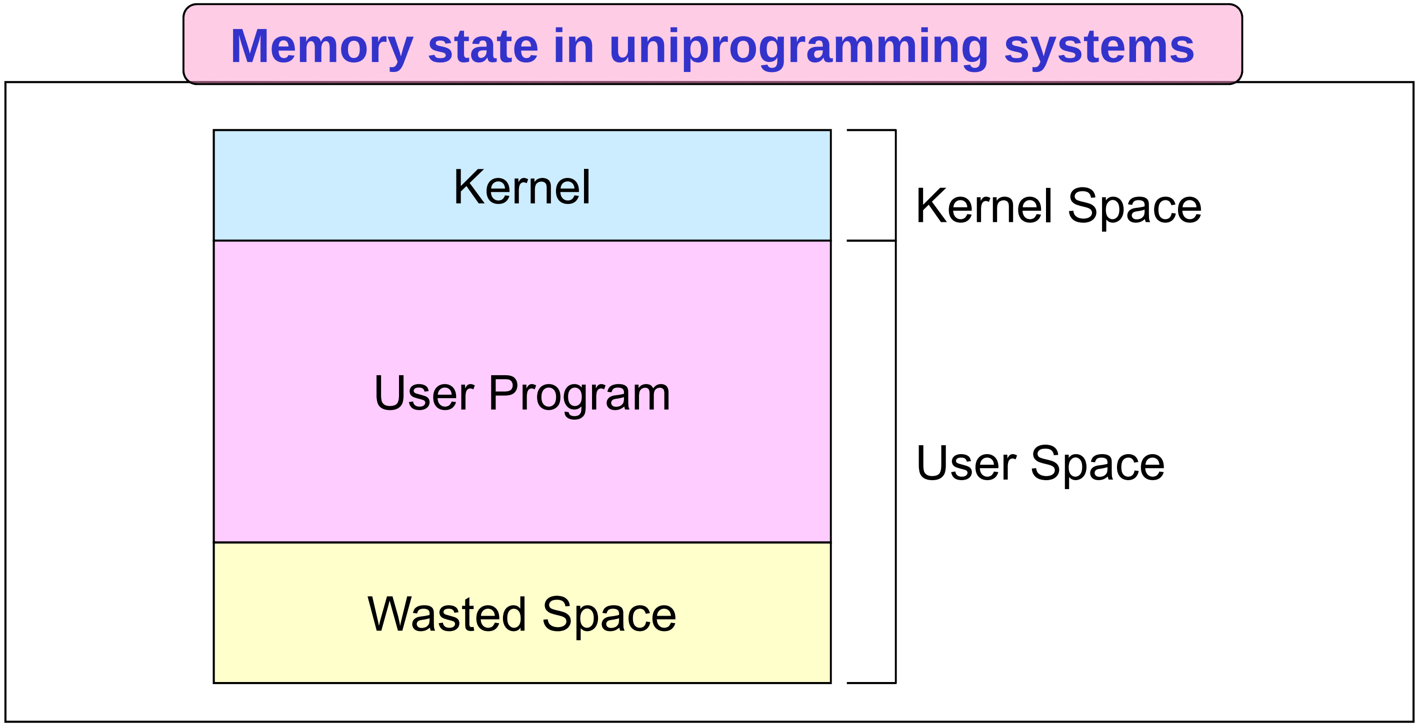
- **Basic policies**
  - Each process (context) is contained in a single contiguous section of memory
- **Policies for memory organization**
  - Number of processes in memory
    - ✓ Affects multiprogramming degree
  - Amount of memory space allocated for each process
  - Memory partition methods
    - ✓ Fixed(static) partition multiprogramming
    - ✓ Variable(dynamic) partition multiprogramming

Contiguous Memory Allocation

- **Uniprogramming**
- **Fixed partition multiprogramming**
- **Variable partition multiprogramming**

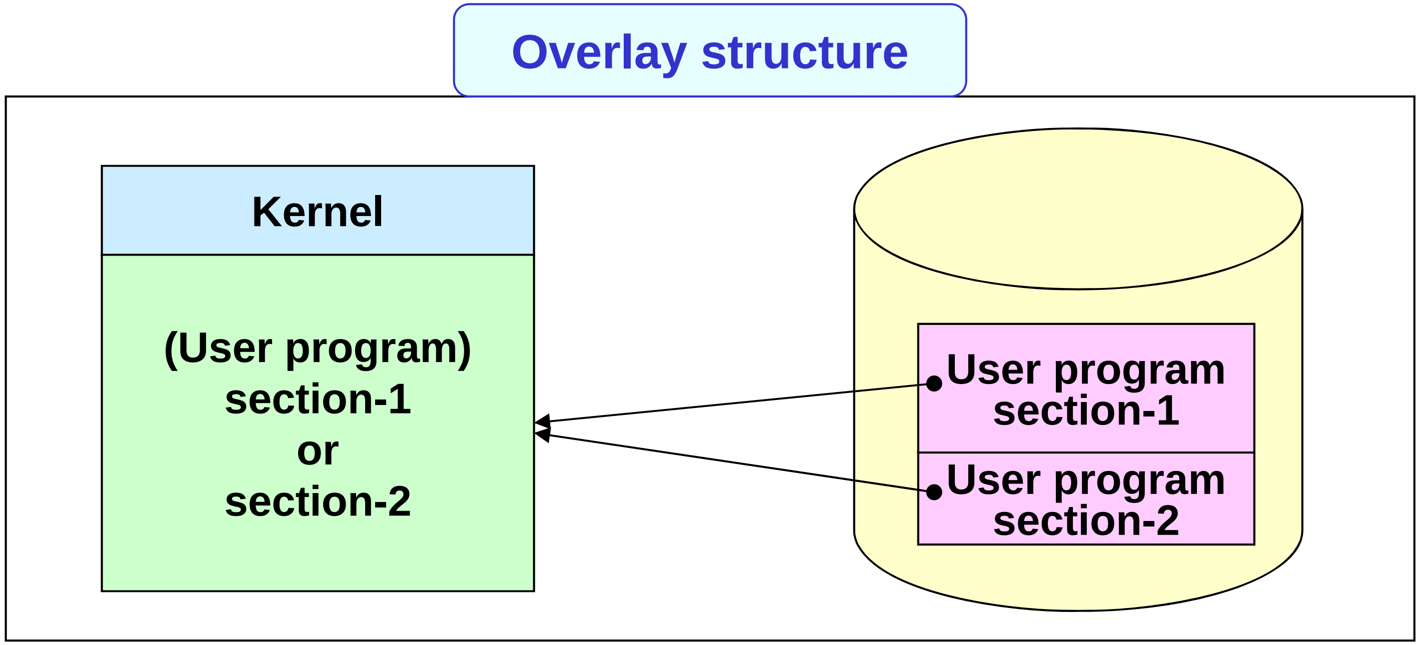
Uniprogramming

- **Uniprogramming system**
  - Only 1 process in memory
  - Simple memory management scheme



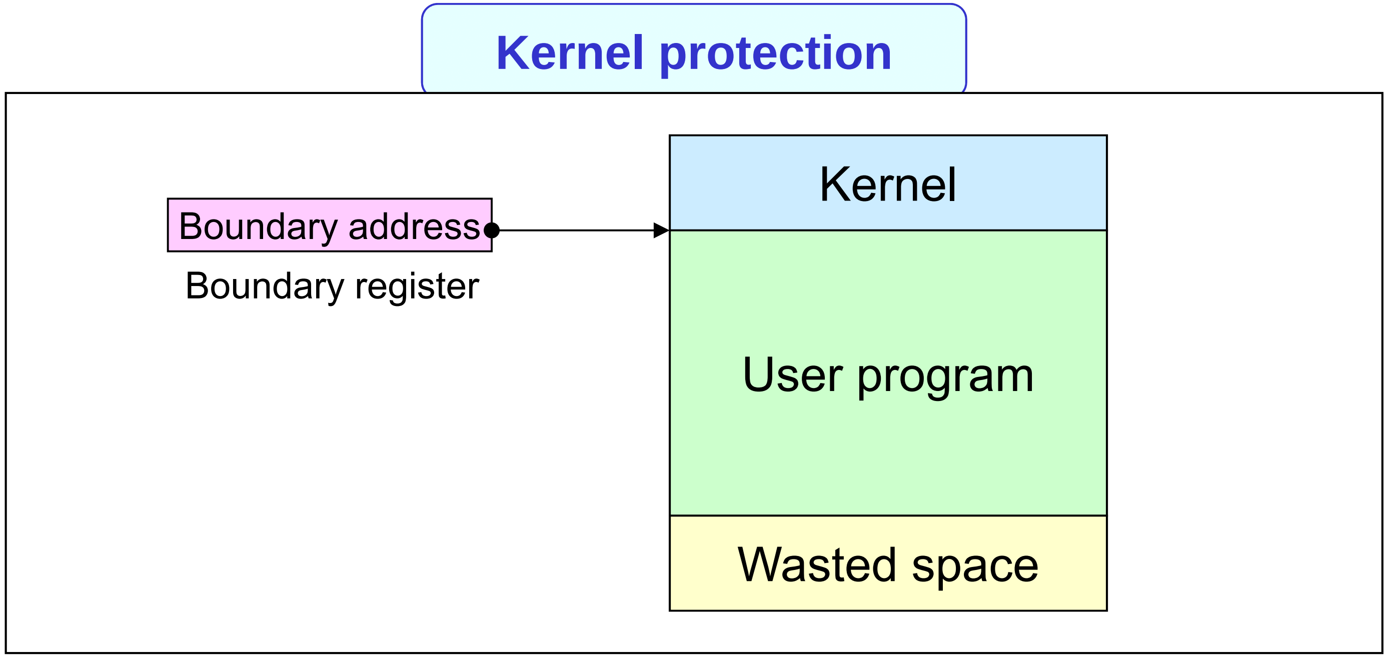
Uniprogramming

- **Issue-1**
  - Program-size > memory-size
    - ✓ Uses **overlay structure**
    - ✓ Requires support from compiler/linker/loader



Uniprogramming

- **Issue-2**
  - Kernel protection
    - ✓ Boundary register



Uniprogramming

- **Issue-3**
  - Low system resource utilization
  - Low system performance

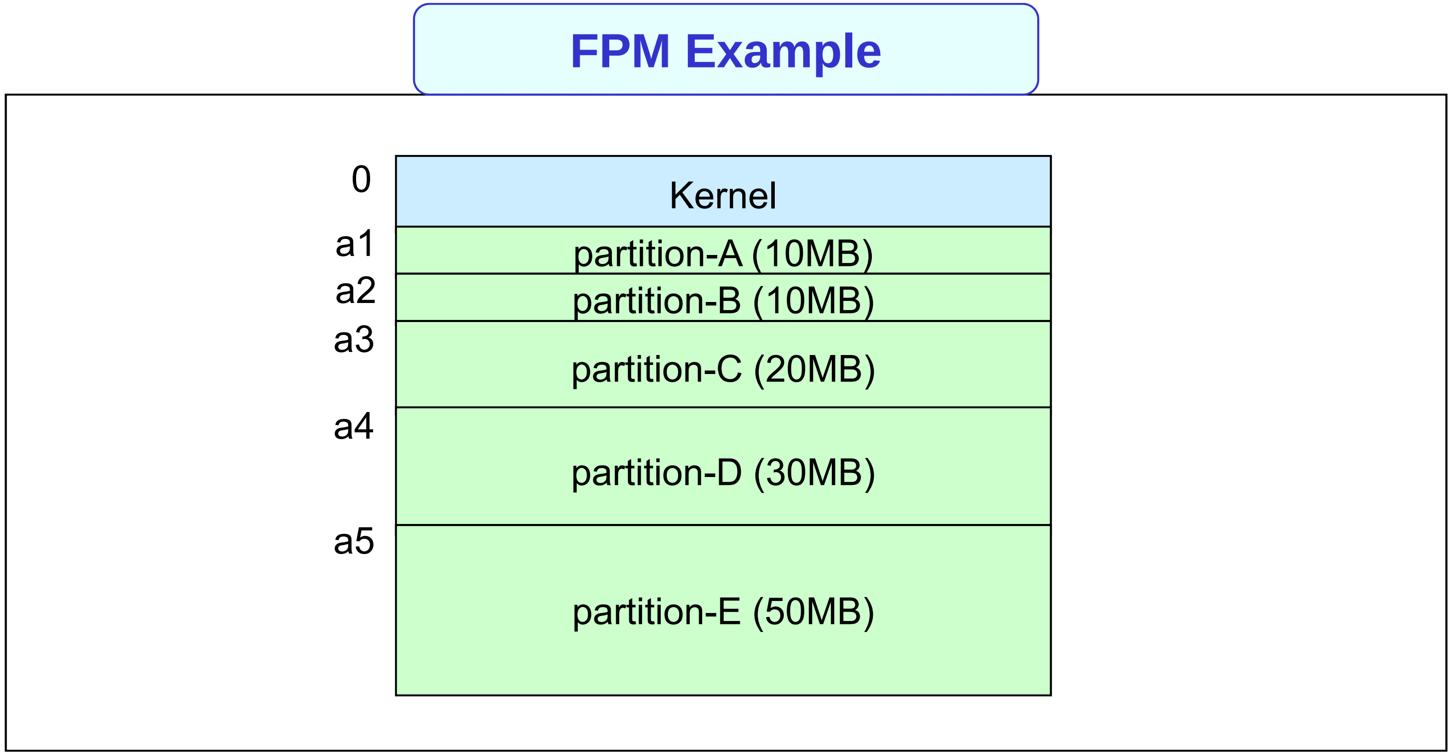
  - Solution
    - ✓ Multiprogramming

FPM

- **FPM: Fixed Partition Multiprogramming**
  - Divide memory into several fixed-size partitions
  - One process in one partition
  - When number of partitions = k
    - ✓ Maximum multiprogramming degree = k
  - IBM OS/360 MFT

FPM

FPM Example



FPM

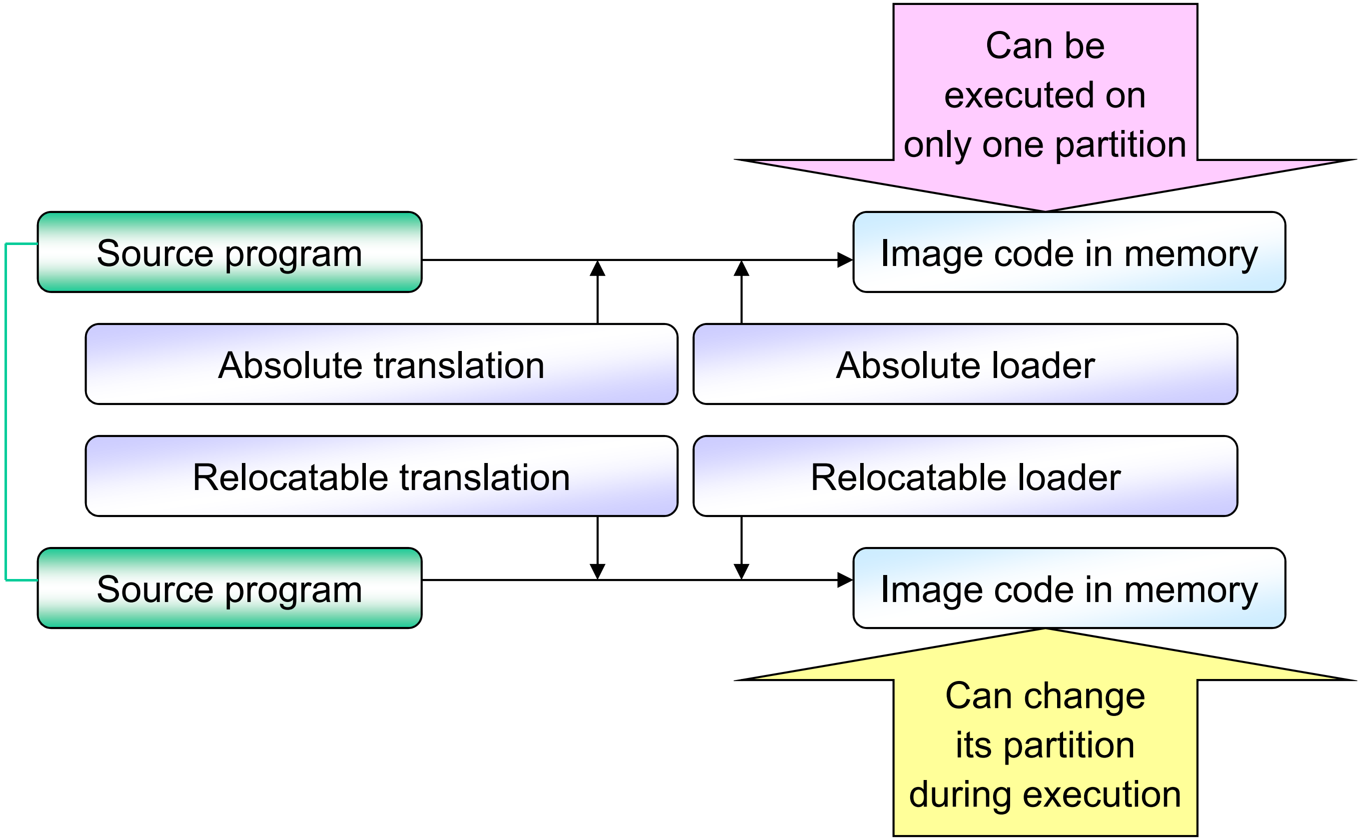
Data structure for FPM: Example

Data structure for FPM

partition	start address	size	current process ID	other fields
A	a1	10 MB	-	· · ·
B	a2	10 MB	-	· · ·
C	a3	20 MB	-	· · ·
D	a4	30 MB	-	· · ·
E	a5	50 MB	-	· · ·

FPM

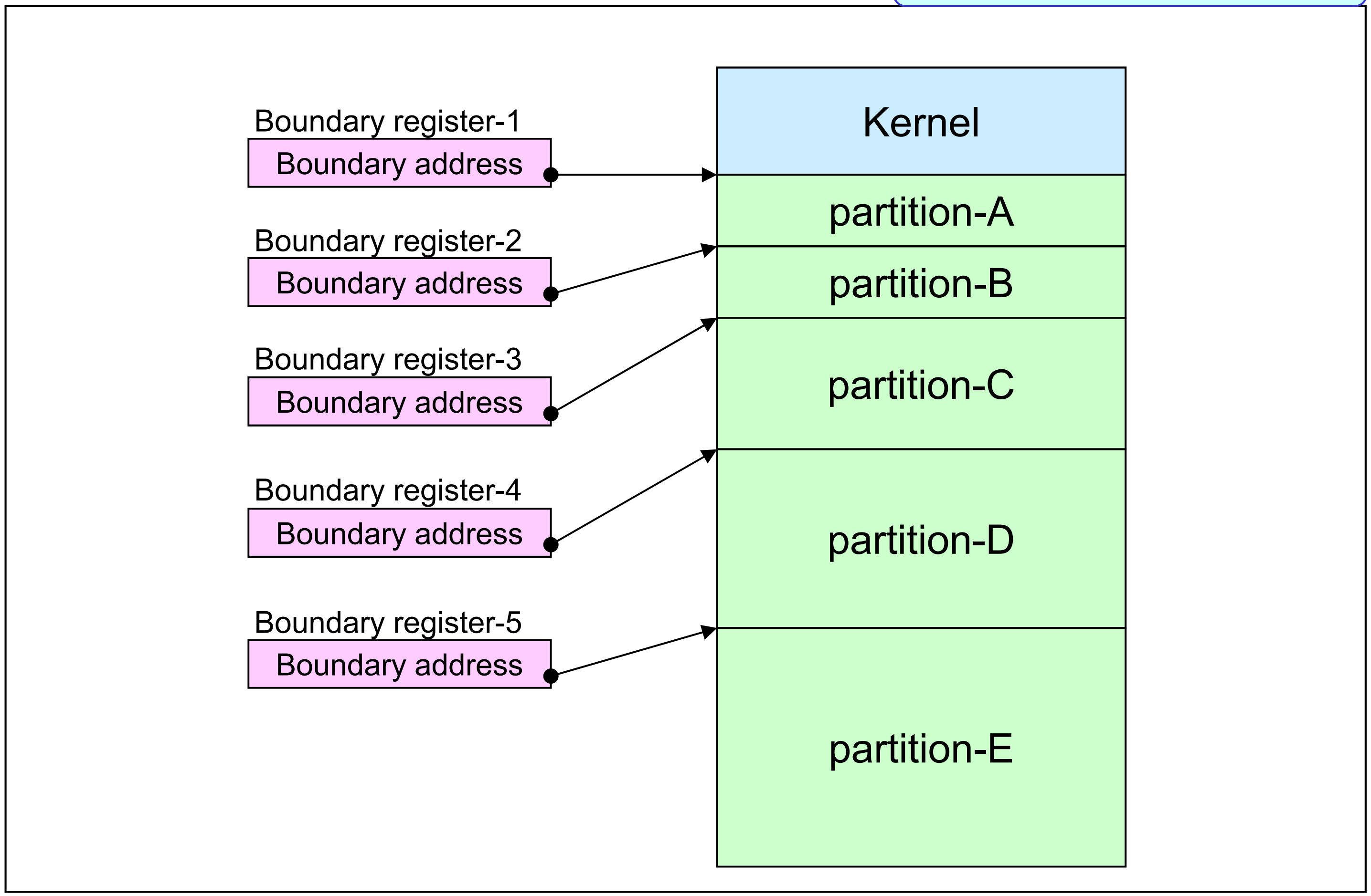
Program relocation



FPM

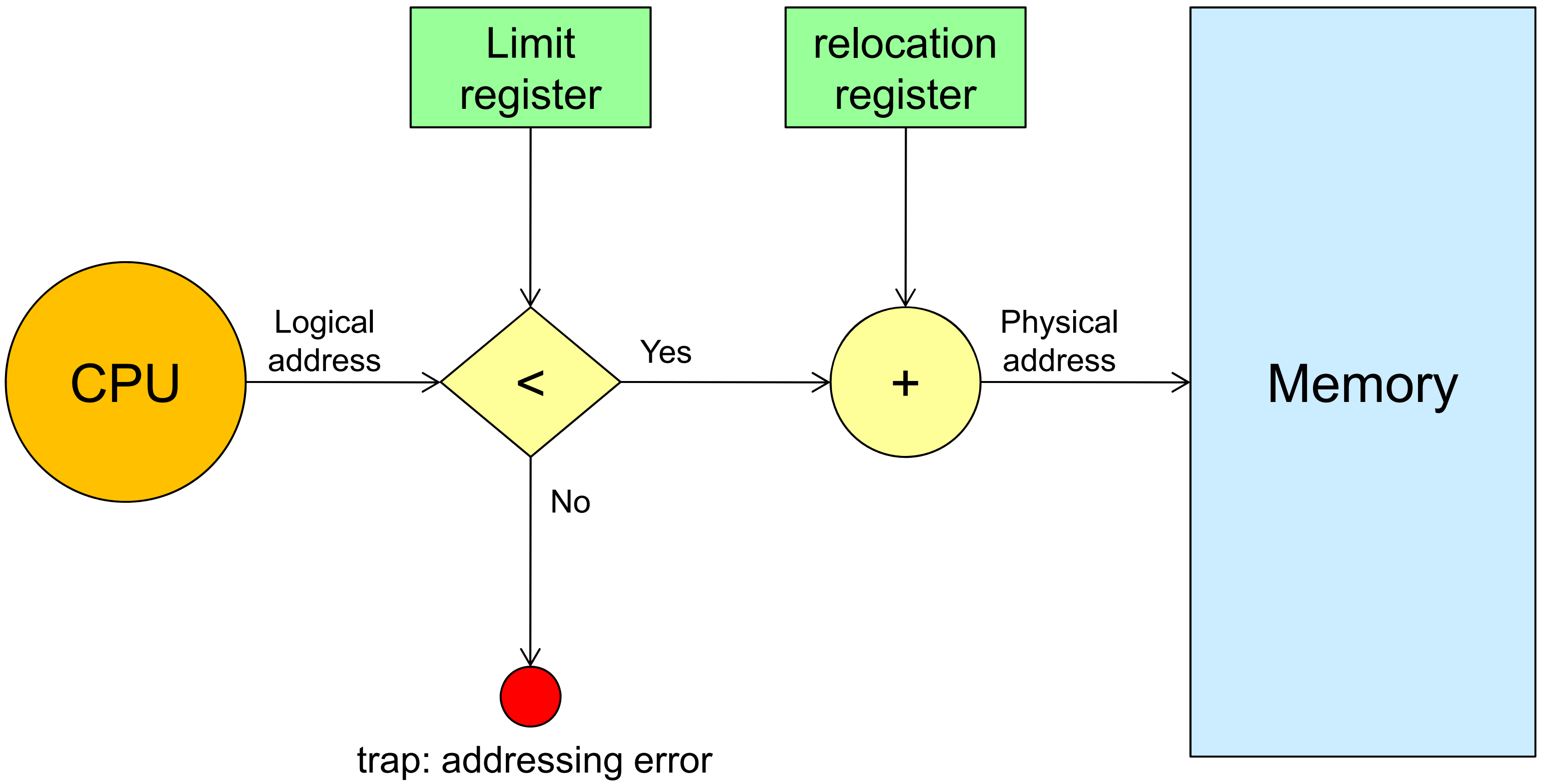
Protection method 1

Protection in FPM



FPM

Protection method 2



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

FPM

- **Protection method 2**
  - Loading the relocation and limit registers
    - ✓ During context switching
    - ✓ By the dispatcher
    - ✓ Using a special privileged instruction
      - Allows the OS to change the value of the registers
      - Prevents user programs from changing the registers

FPM

- **Fragmentation**
  - Storage space waste
    - ✓ Internal fragmentation
      - Exists when the memory space allocated is larger than the requested memory
    - ✓ External fragmentation
      - Exists when enough total memory space exists to satisfy the request, but it is not contiguous

- **Both types of fragmentation exist in FPM**

FPM

- **Summary**
  - Several fixed partitions in memory
  - Simple memory management
  - Low memory management overhead
  - May cause poor resource utilization
  - Internal/external fragmentation

VPM

- **VPM: Variable Partition Multiprogramming**
  - Initially, all memory is available as a single large block of available memory (hole, partition)
  - Memory partition state dynamically changes as a process enters (or exits) the system
  - No internal fragmentation in a partition
  - Contiguous allocation

VPM

Example

Memory allocation and partition scenario

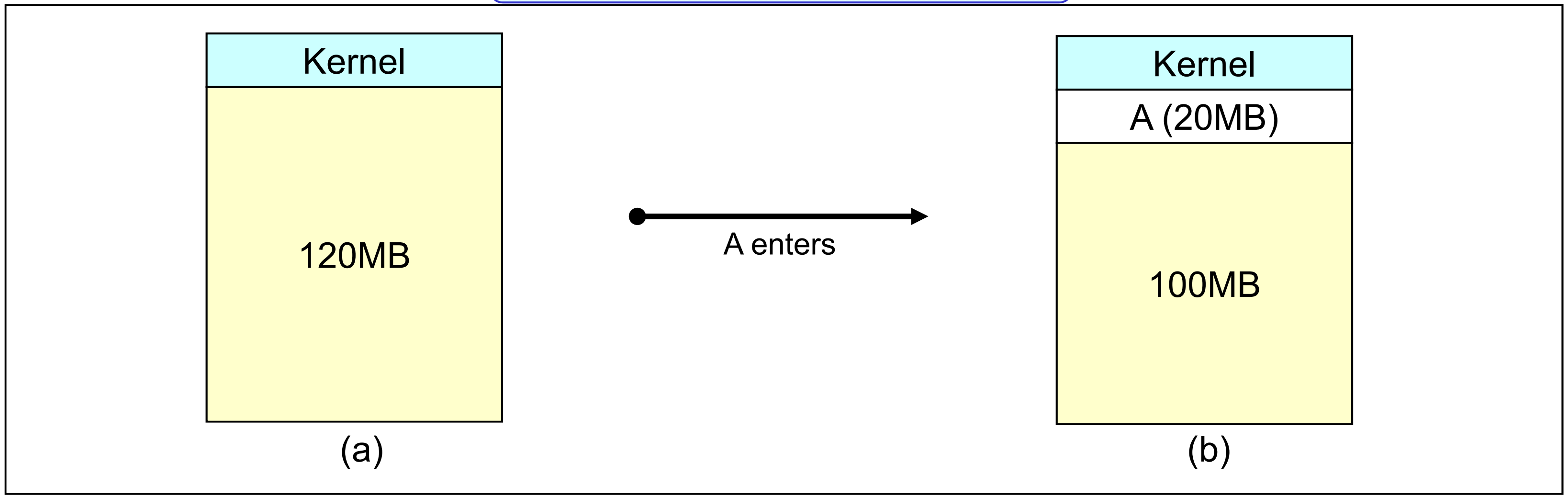
- Assumption

  - Memory space: 120 MB

  - (a) : Initial state
  - (b) : After loading process A(20MB)
  - (c) : After loading process B(10MB)
  - (d) : After loading process C(25MB)
  - (e) : After loading process D(20MB)
  - (f) : After process B releases memory
  - (g) : After loading process E(15MB)
  - (h) : After process D releases memory

VPM

VPM Example

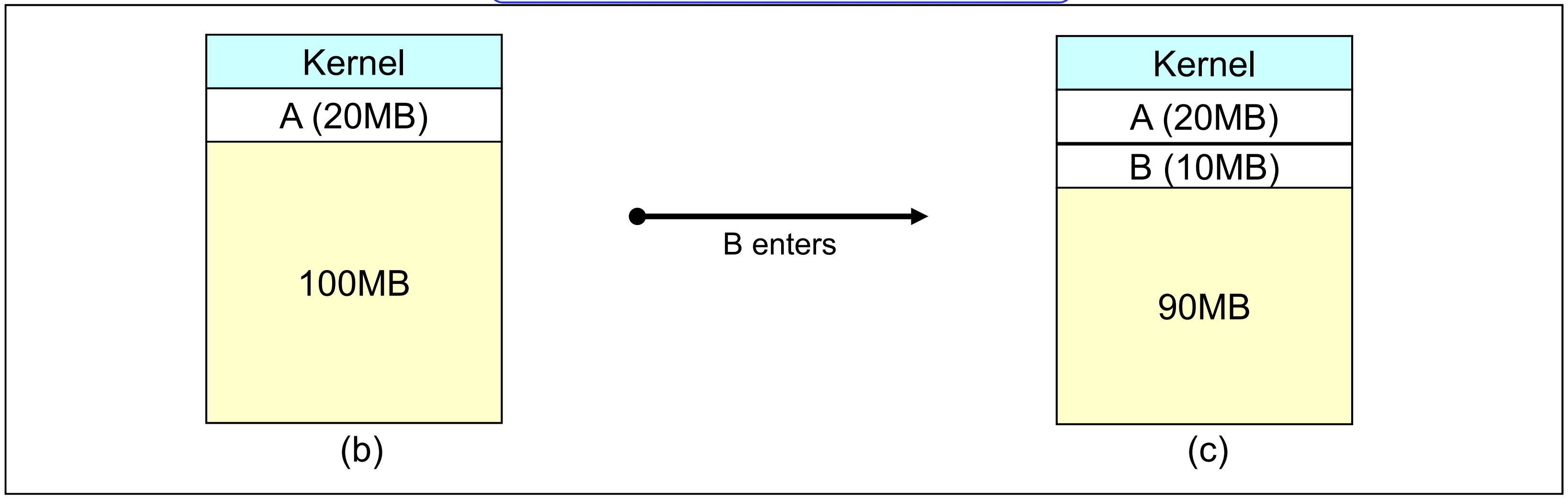


partition	start address	size	PID	other field
1	u	120	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	100	none	· · ·

VPM

VPM Example

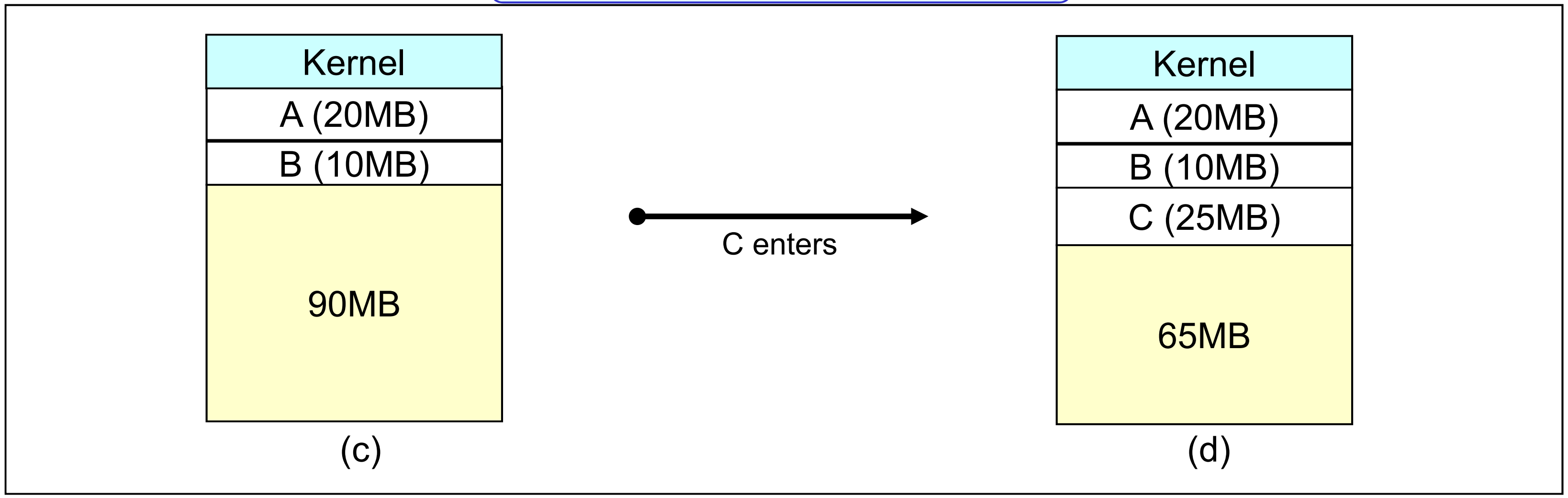


partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	100	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	90	none	· · ·

VPM

VPM Example

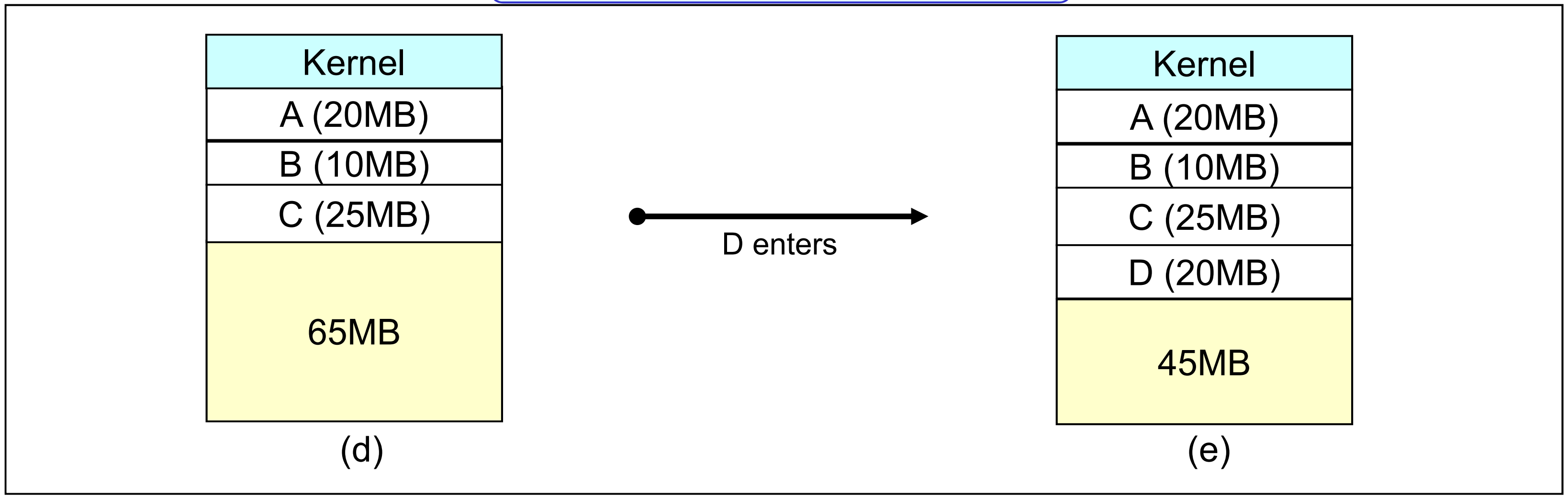


partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	90	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	25	C	· · ·
4	u+55	65	none	· · ·

VPM

VPM Example

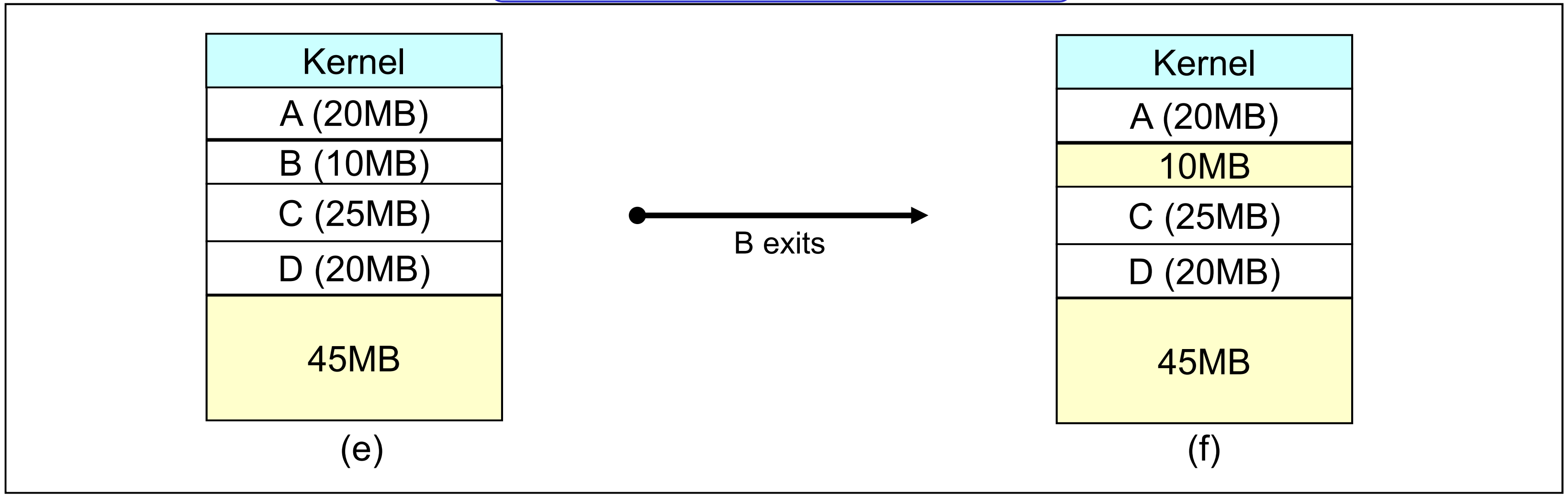


partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	25	C	· · ·
4	u+55	65	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	45	none	· · ·

VPM

VPM Example

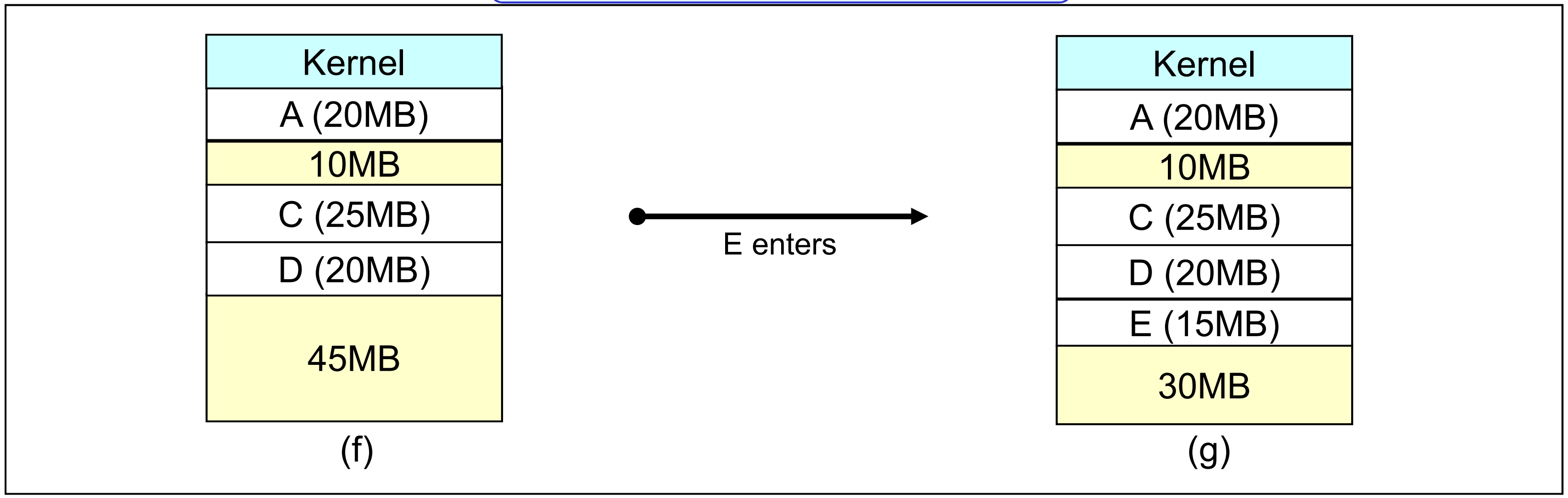


partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	B	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	45	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	none	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	45	none	· · ·

VPM

VPM Example

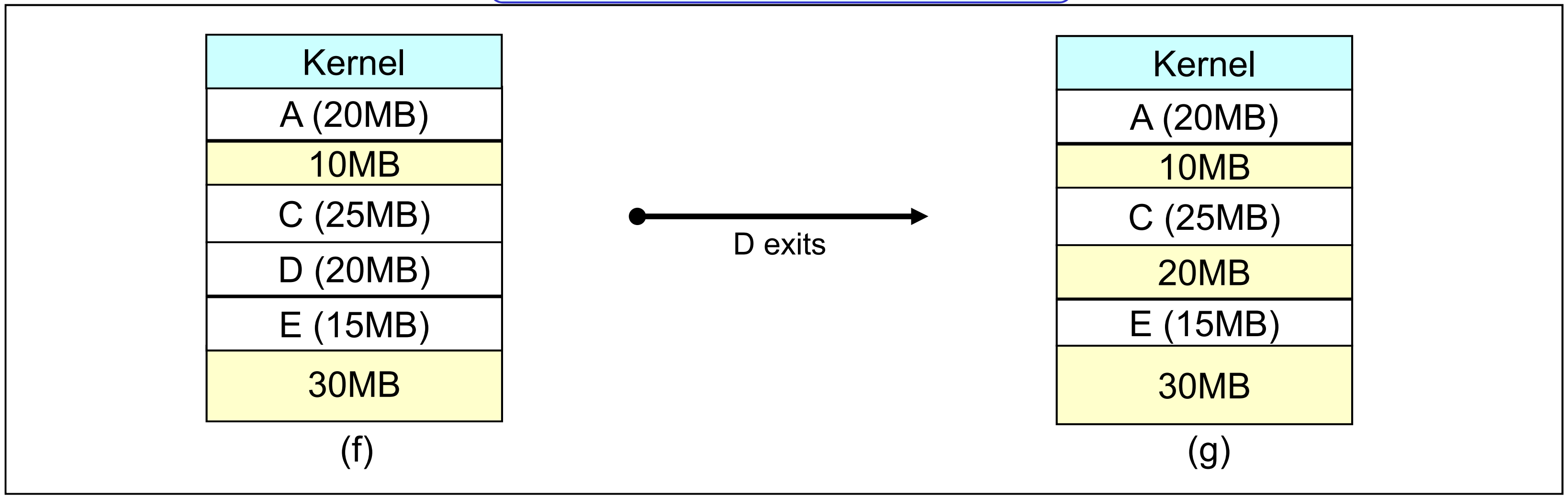


partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	none	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	45	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	none	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	15	E	· · ·
6	u+90	30	none	· · ·

VPM

VPM Example



partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	none	· · ·
3	u+30	25	C	· · ·
4	u+55	20	D	· · ·
5	u+75	15	E	· · ·
6	u+90	30	none	· · ·

partition	start address	size	PID	other field
1	u	20	A	· · ·
2	u+20	10	none	· · ·
3	u+30	25	C	· · ·
4	u+55	20	none	· · ·
5	u+75	15	E	· · ·
6	u+90	30	none	· · ·

VPM

- **Placement strategies**
  - First-fit
    - Start searching at the beginning of the state table
    - Allocate the first partition that is big enough
    - Simple and low overhead
  - Best-fit
    - Search the entire state table
    - Allocate the smallest partition that is big enough
    - Long search time
    - Can reserve large size partitions
    - May produce many small size partitions
    - External fragmentation

VPM

- **Placement strategies**
  - Worst-fit
    - ✓ Search the entire state table
    - ✓ Allocate the largest partition available
    - ✓ May reduce the number of small size partitions
  - Next-fit
    - ✓ Similar to first-fit method
    - ✓ Start searching from where the previous search ended
    - ✓ Circular search the state table
    - ✓ Uniform use for the memory partitions
    - ✓ Low overhead

VPM

Coalescing holes

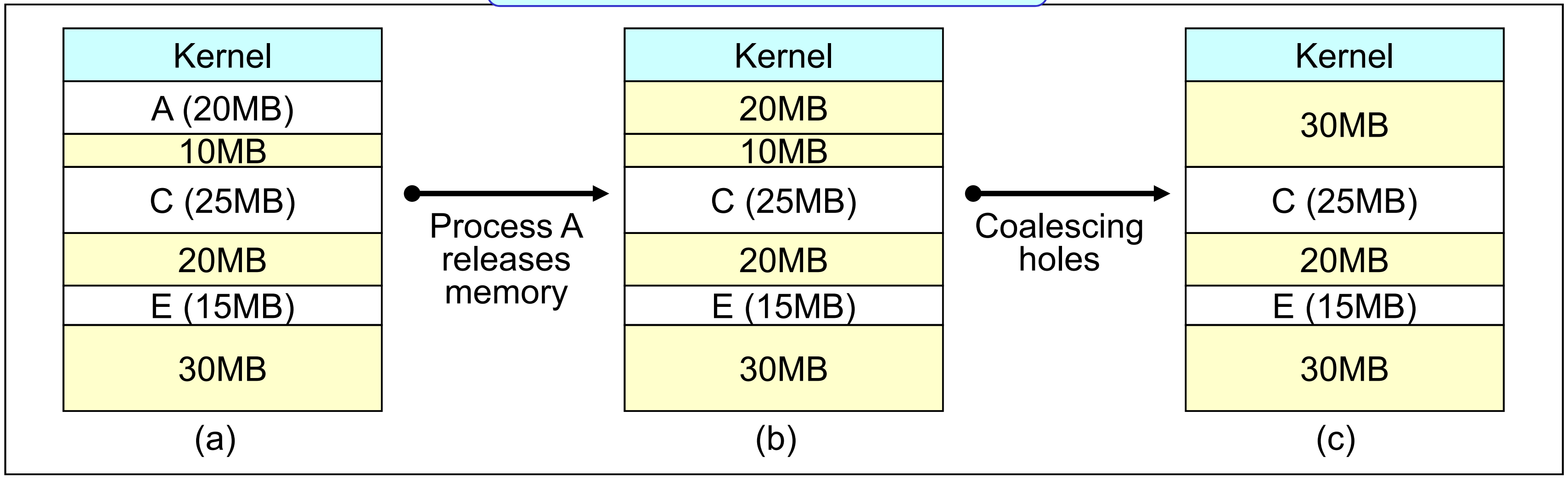
- Merge adjacent free partitions into one large partition

Storage compaction

- Shuffle the memory contents to place all free memory together in one large block (partition)
- Done at execution time
  - ✓ Can be done only if relocation is dynamically possible
- Consumes so much system resources
  - ✓ Consumes long CPU time

VPM

Coalescing holes (1)



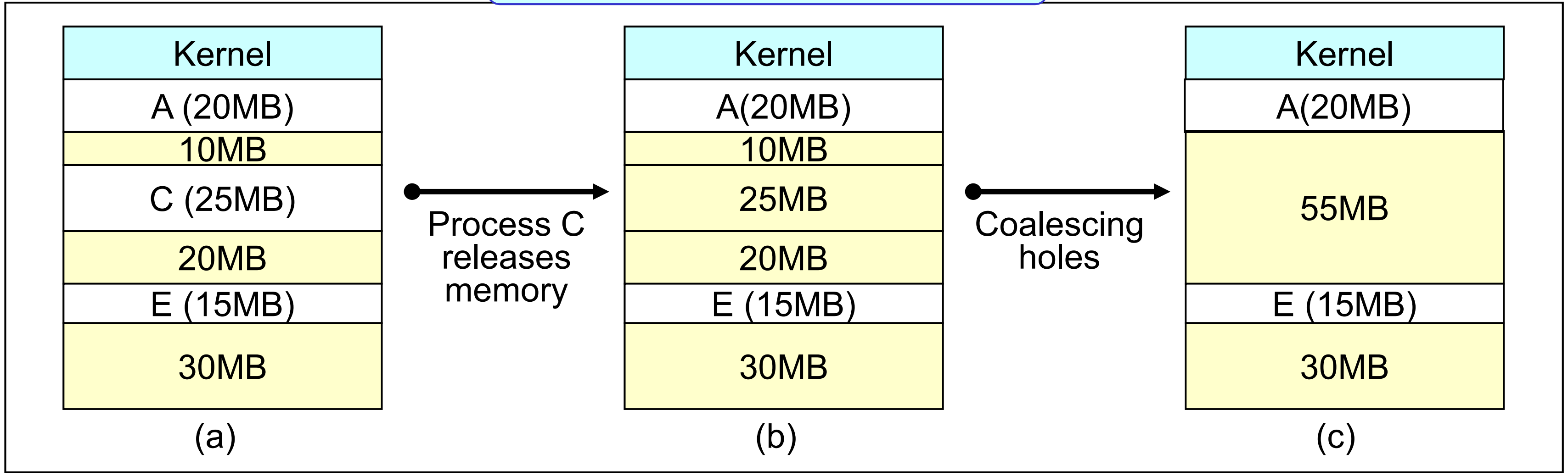
partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	none
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	30	none
2	u+30	25	C
3	u+55	20	none
4	u+75	15	E
5	u+90	30	none

VPM

Coalescing holes (2)



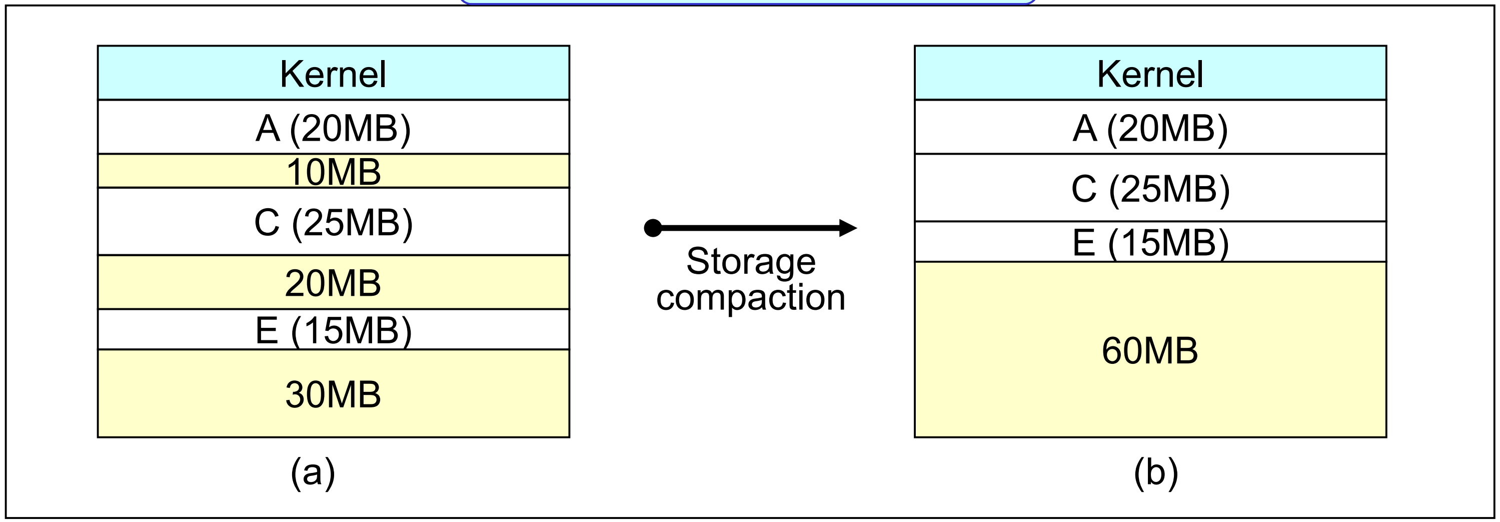
partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	none
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	55	none
3	u+75	15	E
4	u+90	30	none

VPM

Storage compaction



partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	25	C
3	u+45	15	E
4	u+60	60	none

Discontiguous Memory Allocation

Discontiguous Memory Allocation

- Paging
- Segmentation

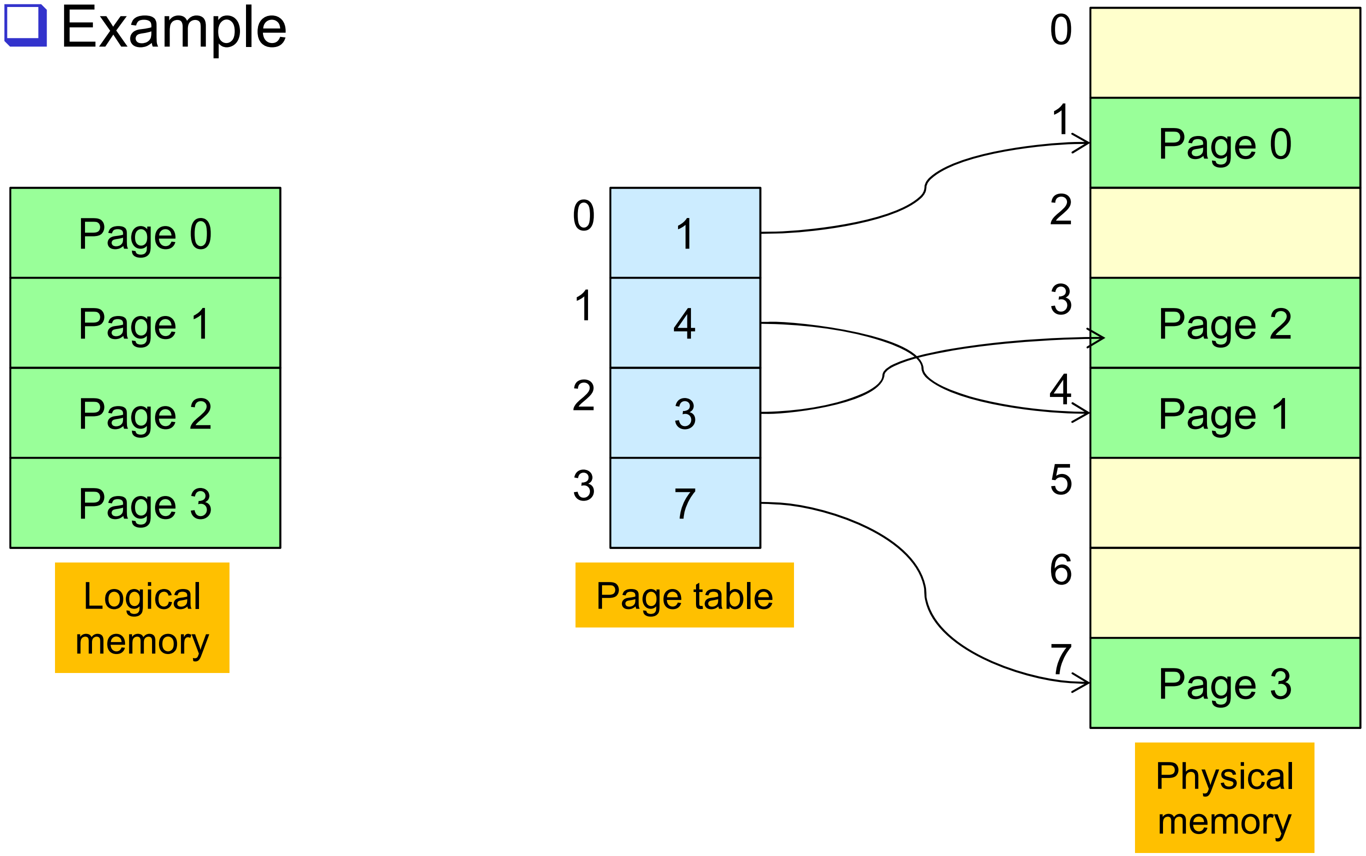
Paging

- **Paging**
  - Memory management scheme that permits the physical address space of a process to be noncontiguous
  - Implemented by closely integrating the hardware and operating system

Paging

Paging

Example



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Paging

Basic method

- Frames (page frames)
  - ✓ Breaking physical memory into fixed-size blocks
- Pages
  - ✓ Breaking logical memory into blocks of the same size
  - ✓ Page size
    - Typically power of 2
    - 512B ~ 16MB (typically 4KB ~ 8KB)
- Logical address
  - ✓ **v** = (**p**, **d**)
    - **p**: page number
    - **d**: page offset (displacement)

Paging

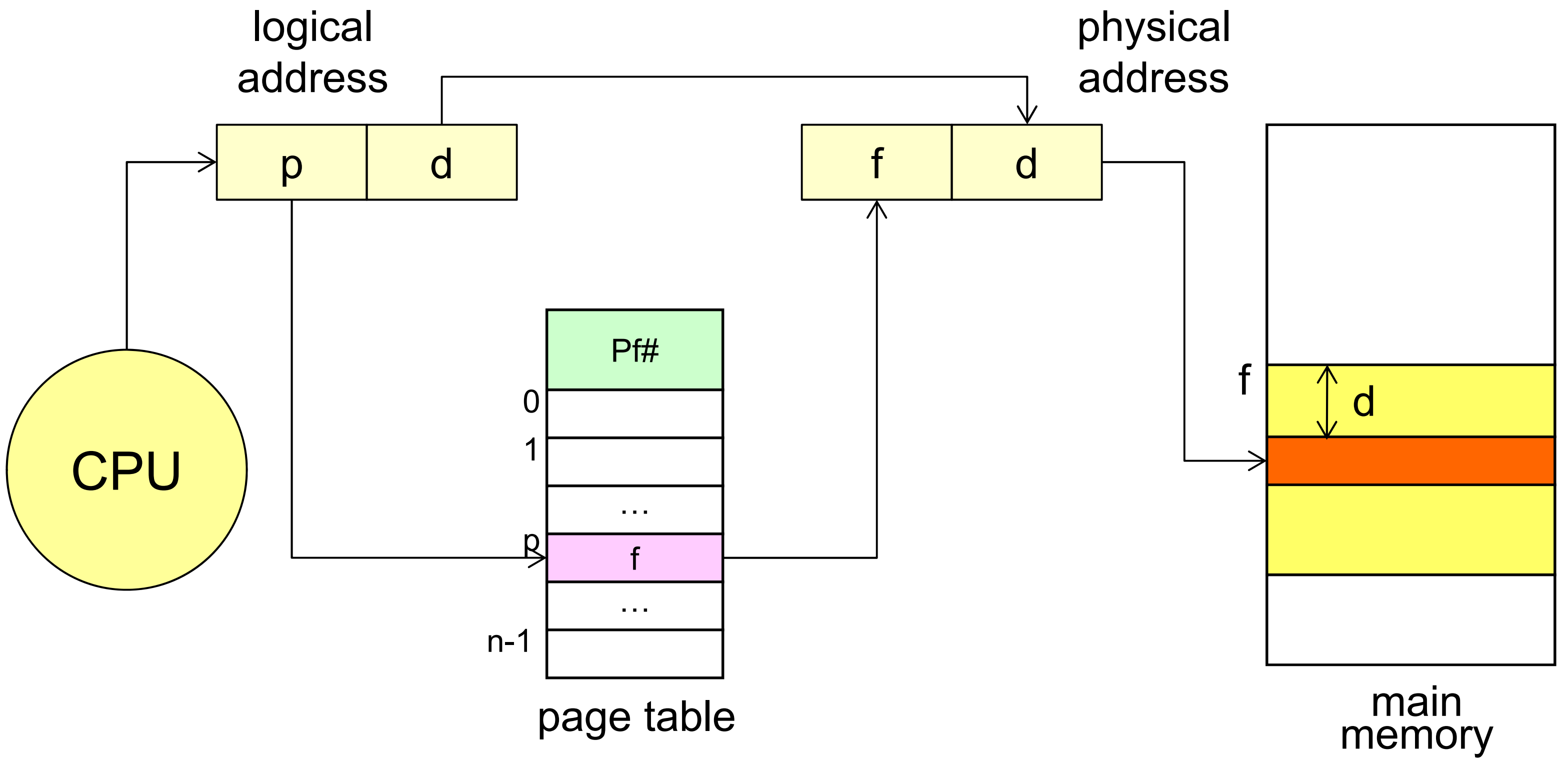
Basic method

- Address mapping
  - ✓ Logical address → physical address
  - ✓ Hidden from the user and controlled by the operating system
  - ✓ Uses page table
    - A page table for each process

Paging

Basic method

- Paging hardware and address mapping



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Paging

Basic method

- Page table implementation
  - ✓ Dedicated CPU registers
  - ✓ TLB (Translation Look-aside Buffer)

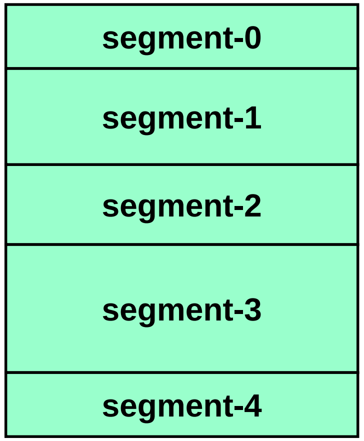
See [Virtual Memory Organization]

Segmentation

Basic concept

- View memory as a collection of variable-sized segments
- View program as a collection of various objects
  - ✓ Functions, methods, procedures, objects, arrays, stacks, variables, and so on

- Logical address
  - ✓ **v** = (**s**, **d**)
    - **s**: segment number
    - **d**: offset (displacement)



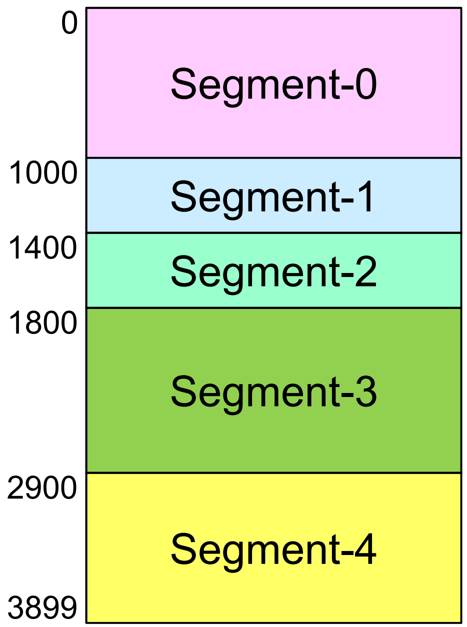
Segmentation

Basic concept

- Normally, when the user program is compiled, the compiler automatically constructs segments reflecting the input program
  - ✓ Code
  - ✓ Global variables
  - ✓ Heap
  - ✓ Stacks
  - ✓ Standard library

Segmentation

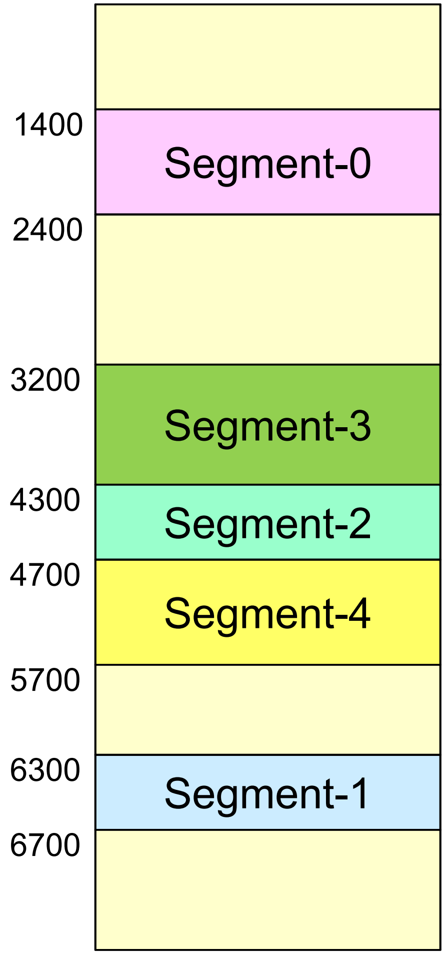
Example



Logical address space

	base	limit
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000

Segment table

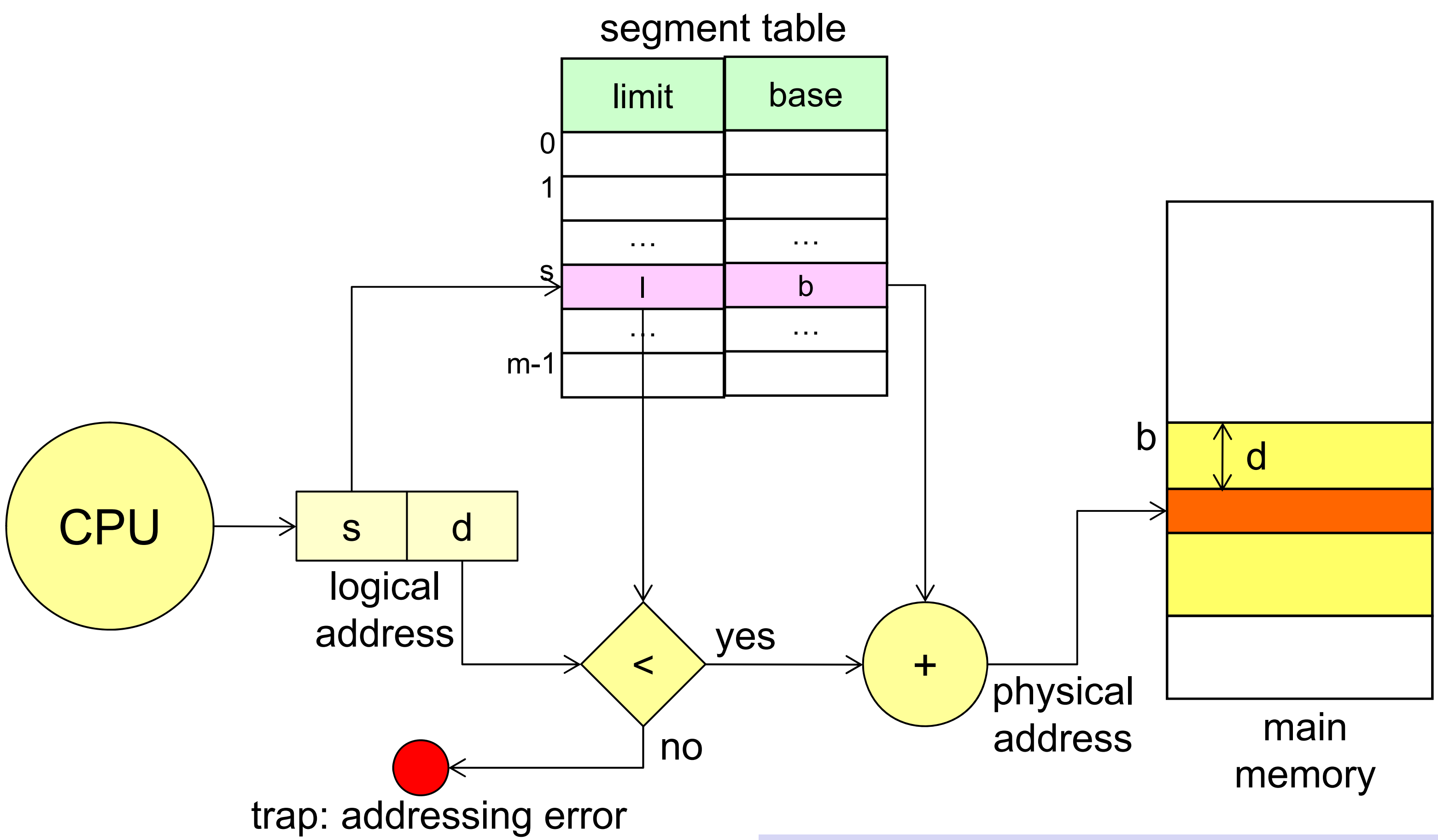


Physical memory

A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Segmentation

Address mapping



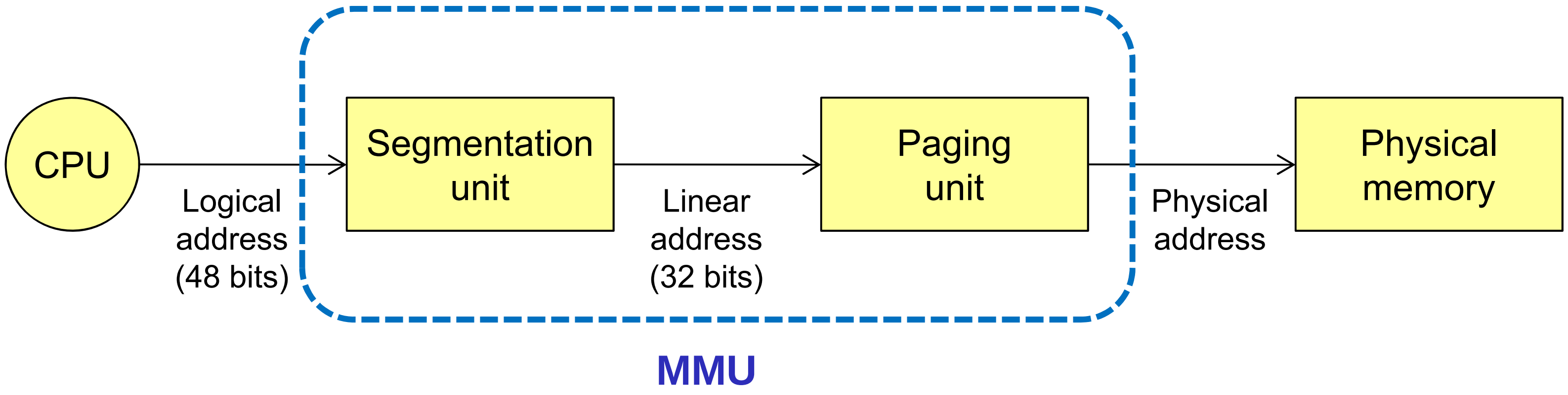
A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Intel Pentium Architecture

Intel Pentium Architecture

Intel Pentium architecture

- Supports both pure segmentation and segmentation with paging
- Address mapping



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Intel Pentium Architecture

Pentium segmentation

- Segment size: max 4GB
- Max number of segments per process: 2^{14}
- Logical address space of a process
  - ✓ 1st partition: 2^{13} private segments
    - LDT(Local Descriptor Table) keeps the information
  - ✓ 2nd partition: 2^{13} shared segments
    - GDT(Global Descriptor Table) keeps the information

- Each entry of LDT and GDT
  - ✓ 8-byte segment descriptor
  - ✓ Detailed information on a particular segment
    - Base location, limit of the segment, etc

Intel Pentium Architecture

- **Pentium segmentation**
  - 6 segment registers
    - ✓ Allows 6 segments to be addressed at any time
    - ✓ Points to the appropriate entry in LDT or GDT
  - 6 8-byte micro-program registers
    - ✓ Corresponding descriptors from either the LDT or GDT

Intel Pentium Architecture

- **Pentium segmentation**
  - Logical address: (**selector**, **offset**)
    - ✓ **selector**: (**s**, **g**, **p**)
      - **s**: segment number (13 bits)
      - **g**: whether the segment is in the LDT or GDT (1 bit)
      - **p**: deals with protection (2 bits)

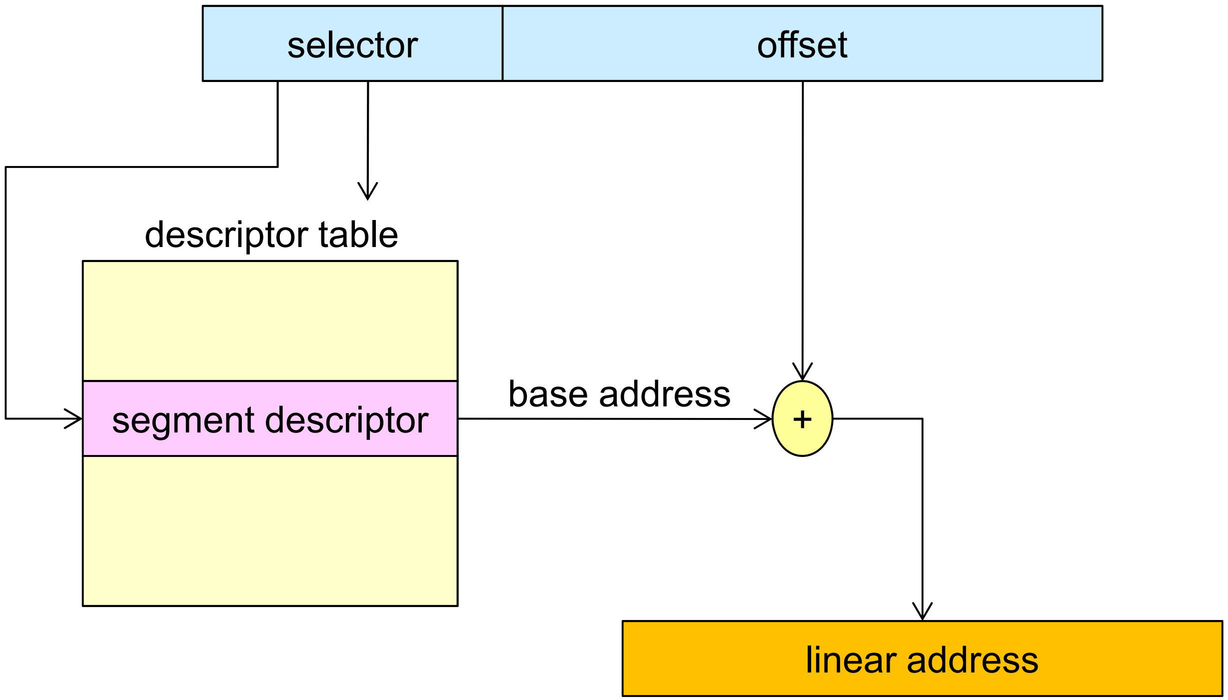
s	g	p
13	1	2

    - ✓ **offset**
      - 32-bit number (location within the segment)

Intel Pentium Architecture

Pentium segmentation

- Linear address generation



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Intel Pentium Architecture

Pentium paging

- Page size: 4KB or 4MB
- For 4KB pages
  - ✓ Two-level paging scheme is used

p1	p2	d
10	10	12
page number		page offset

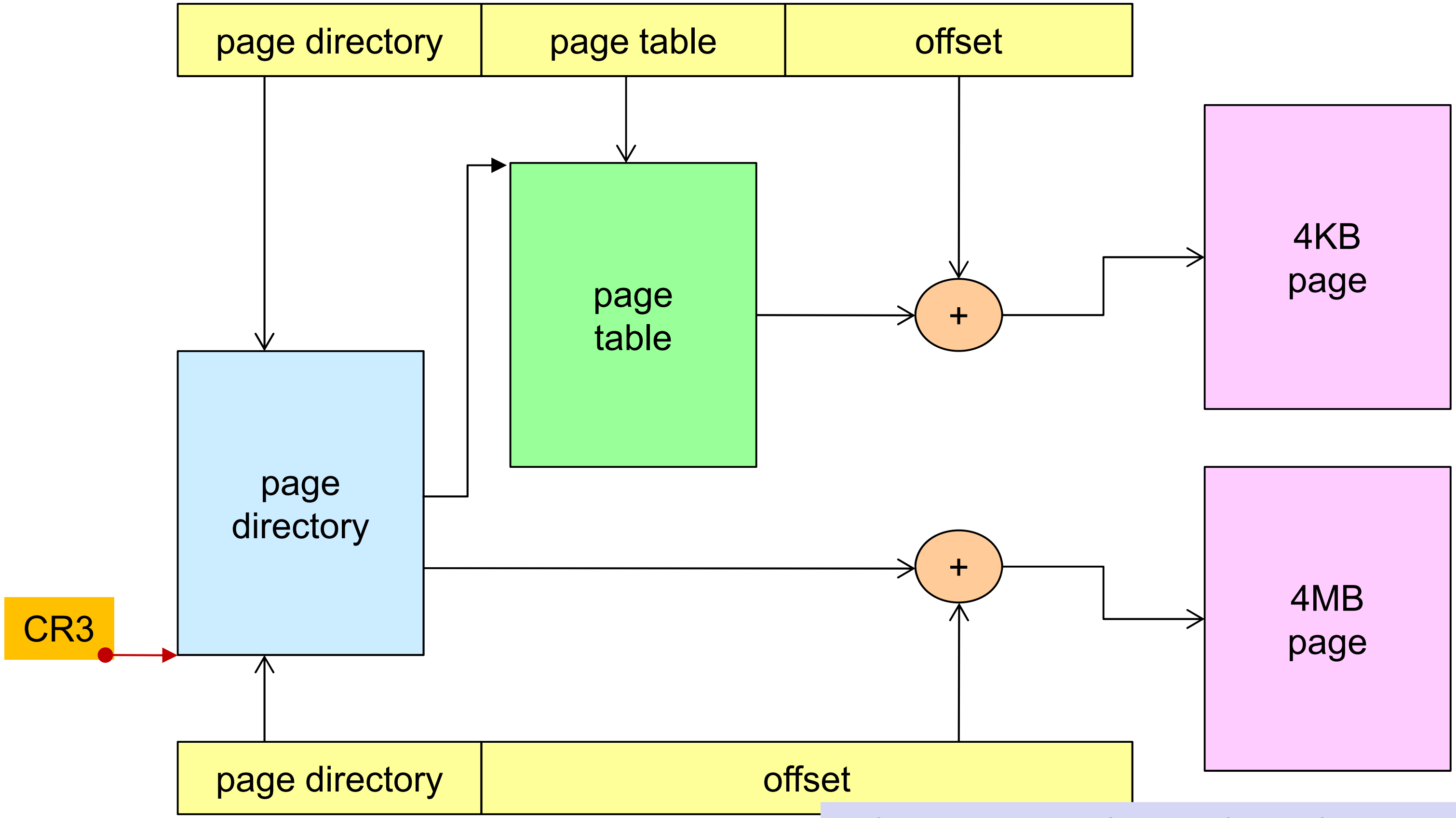
- For 4MB pages
  - ✓ Single-level paging scheme is used

p1	d
10	22
page number	page offset

Intel Pentium Architecture

Pentium paging

- Physical address generation



A. Silberschatz, et. al., Operating System Concepts, 8-ed., Wiley, 2010.

Intel Pentium Architecture

- **Pentium paging**
  - Notes
    - **Page size** flag in the page directory entry
      - Indicates whether the size of the page frame is 4KB or 4MB
    - **Invalid** bit in the page directory entry
      - Indicates whether the page table to which the entry is pointing is in memory or on disk
      - When the table is on disk, the OS can use the other 31 bits to specify the disk location of the table
      - The table is brought into memory on demand

Intel Pentium Architecture

- **Linux on Pentium systems**
  - Linux does not rely on segmentation and uses it minimally
  - Linux uses only 6 segments
    - ✓ Kernel code segment
    - ✓ Kernel data segment
    - ✓ User code segment
    - ✓ User data segment
    - ✓ Task state segment (TSS)
    - ✓ Default LDT segment

Intel Pentium Architecture

- **Linux on Pentium systems**
  - User code and user data segment
    - ✓ Shared by all processes running in user mode
  - One TSS for each process
    - ✓ Used to store H/W register context during context switches
  - Default LDT segment
    - ✓ Generally not used

Intel Pentium Architecture

Linux on Pentium systems

- 2-bit protection field in segment selector
  - Allows 4 levels of protection
  - But, Linux uses only 2 modes
    - User mode and kernel mode
- Linux adopted 3-level paging scheme
  - Works well in both 32-bit and 64-bit architectures
  - Linear address

global directory	middle directory	page table	offset

  - For Pentium systems, the size of the middle directory is zero bits

Summary

- **Contiguous memory allocation**
  - Uniprogramming
  - Multiprogramming
    - ✓ FPM
    - ✓ VPM
- **Discontiguous memory allocation**
  - Paging
  - Segmentation
- **Intel Pentium system**
  - Paging and segmentation
  - Linux on Pentium systems