Memory Management

Learning Outcomes

- Appreciate the need for memory management in operating systems, understand the limits of fixed memory allocation schemes.

- Understand fragmentation in dynamic memory allocation, and understand basic dynamic allocation approaches.

- Understand how program memory addresses relate to physical memory addresses, memory management in base-limit machines, and swapping

- An overview of virtual memory management.

Process

- One or more threads of execution
- Resources required for execution
  - Memory (RAM)
    - Program code ("text")
    - Data (initialised, uninitialised, stack)
    - Buffers held in the kernel on behalf of the process
  - Others
    - CPU time
    - Files, disk space, printers, etc.

UNSW
SYDNEY

OS Memory Management

- Keeps track of what memory is in use and what memory is free
- Allocates free memory to process when needed
  - And deallocates it when they don't
- Manages the transfer of memory between RAM and disk.

Memory Hierarchy

- Ideally, programmers want memory that is
  - Fast
  - Large
  - Nonvolatile
- Not possible
- Memory management coordinates how memory hierarchy is used.
  - Focus usually on RAM ⇔ Disk

registers

cache

main memory

electronic disk

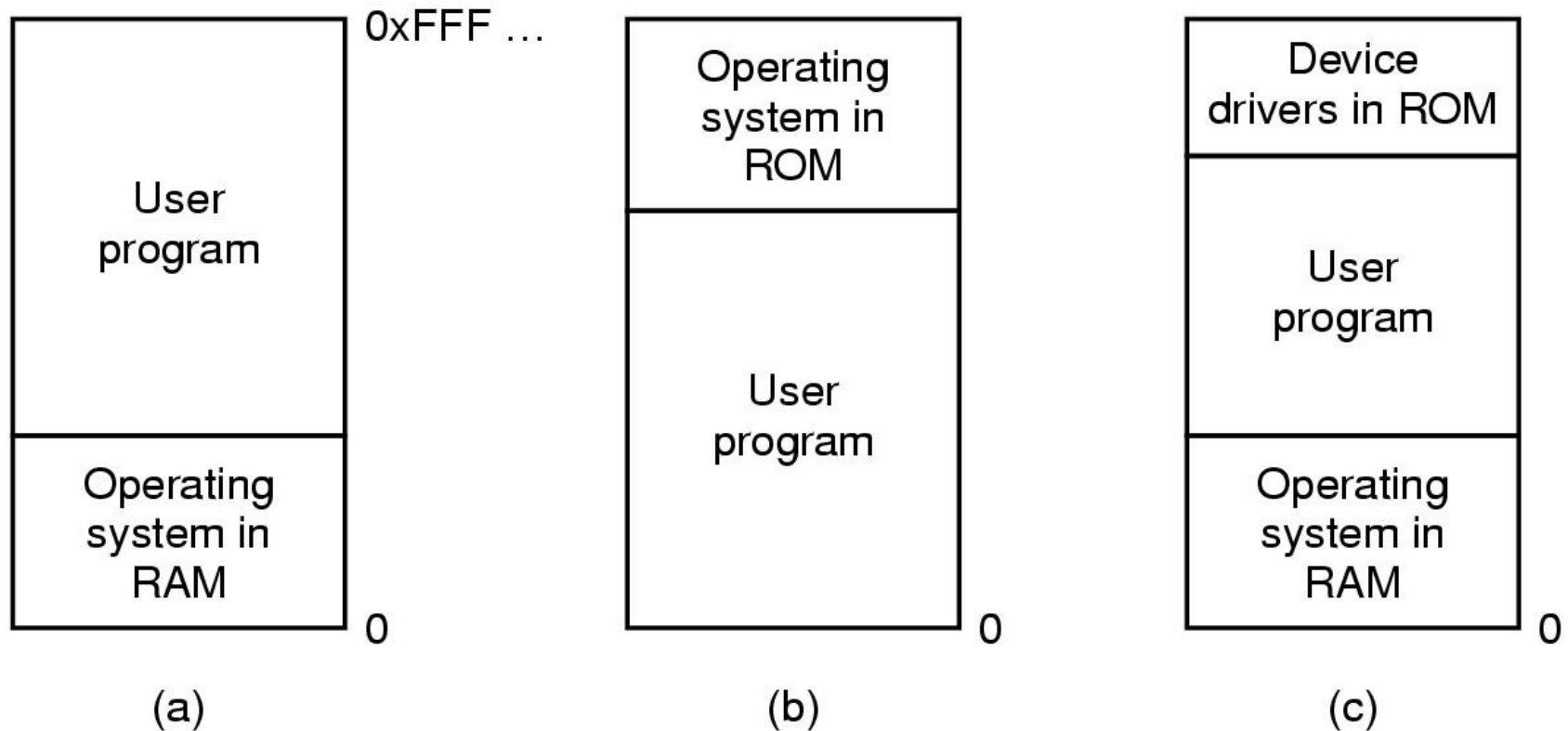magnetic disk

optical disk

magnetic tapes
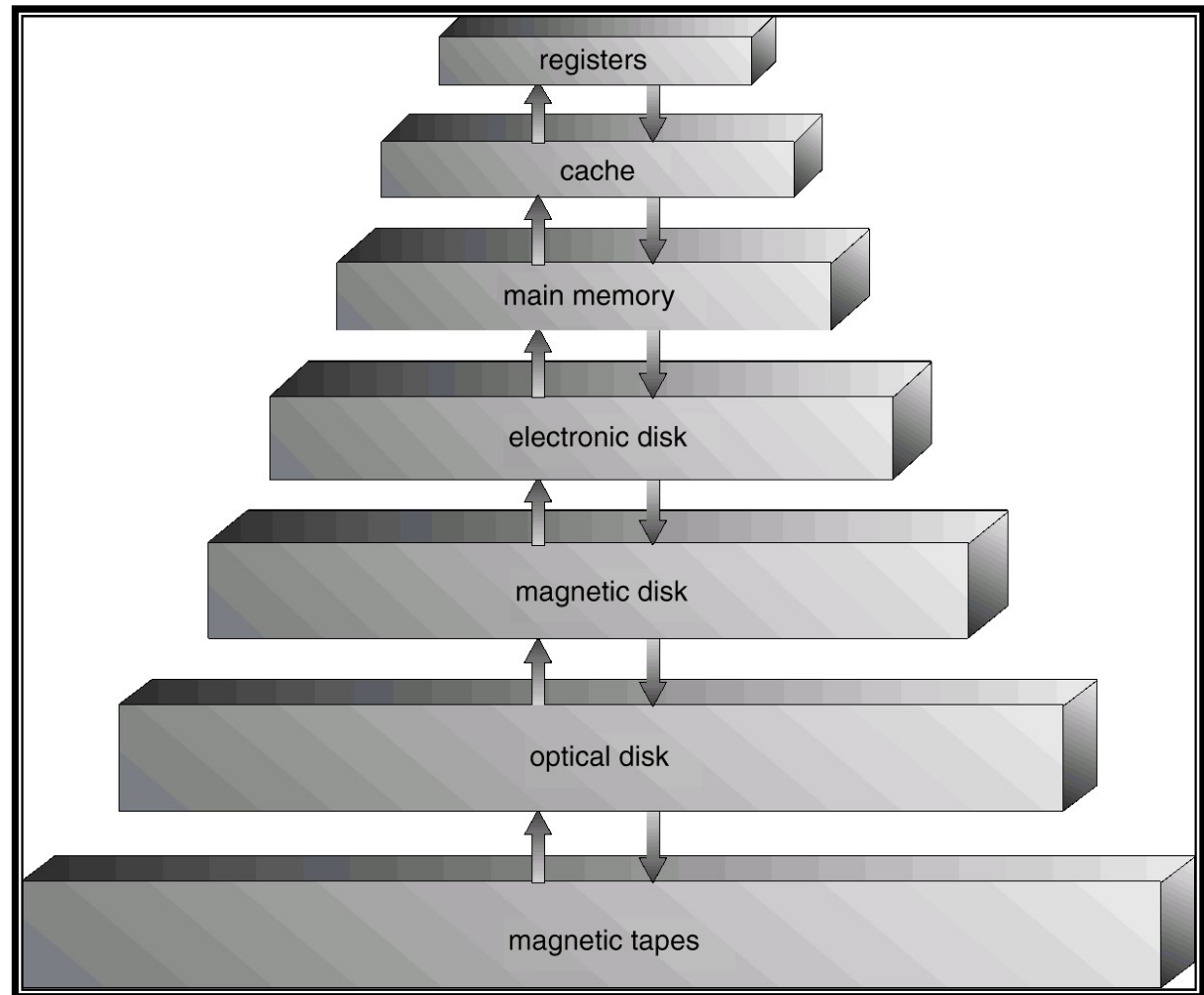
OS Memory Management

- Two broad classes of memory management systems
  - Those that transfer processes to and from external storage during execution.
    - Called swapping or paging
  - Those that don't
    - Simple
    - Might find this scheme in an embedded device, dumb phone, or smartcard.

Basic Memory Management
Monoprogramming without Swapping or Paging

User program	Operating system in ROM	Device drivers in ROM
	User program	User program
Operating system in RAM		Operating system in RAM
(a)	(b)	(c)

0xFFF …   0   0   0

Three simple ways of organizing memory

- an operating system with one user process

Monoprogramming

- Okay if
    - Only have one thing to do
    - Memory available approximately equates to memory required

- Otherwise,
    - Poor CPU utilisation in the presence of I/O waiting
    - Poor memory utilisation with a varied job mix

Idea

- Recall, an OS aims to
  - Maximise memory utilisation resource utilisation
  - Maximise CPU utilization
    - (ignore battery/power-management issues)
- Subdivide memory and run more than one process at once!!!!
 - Multiprogramming, Multitasking

Modeling Multiprogramming
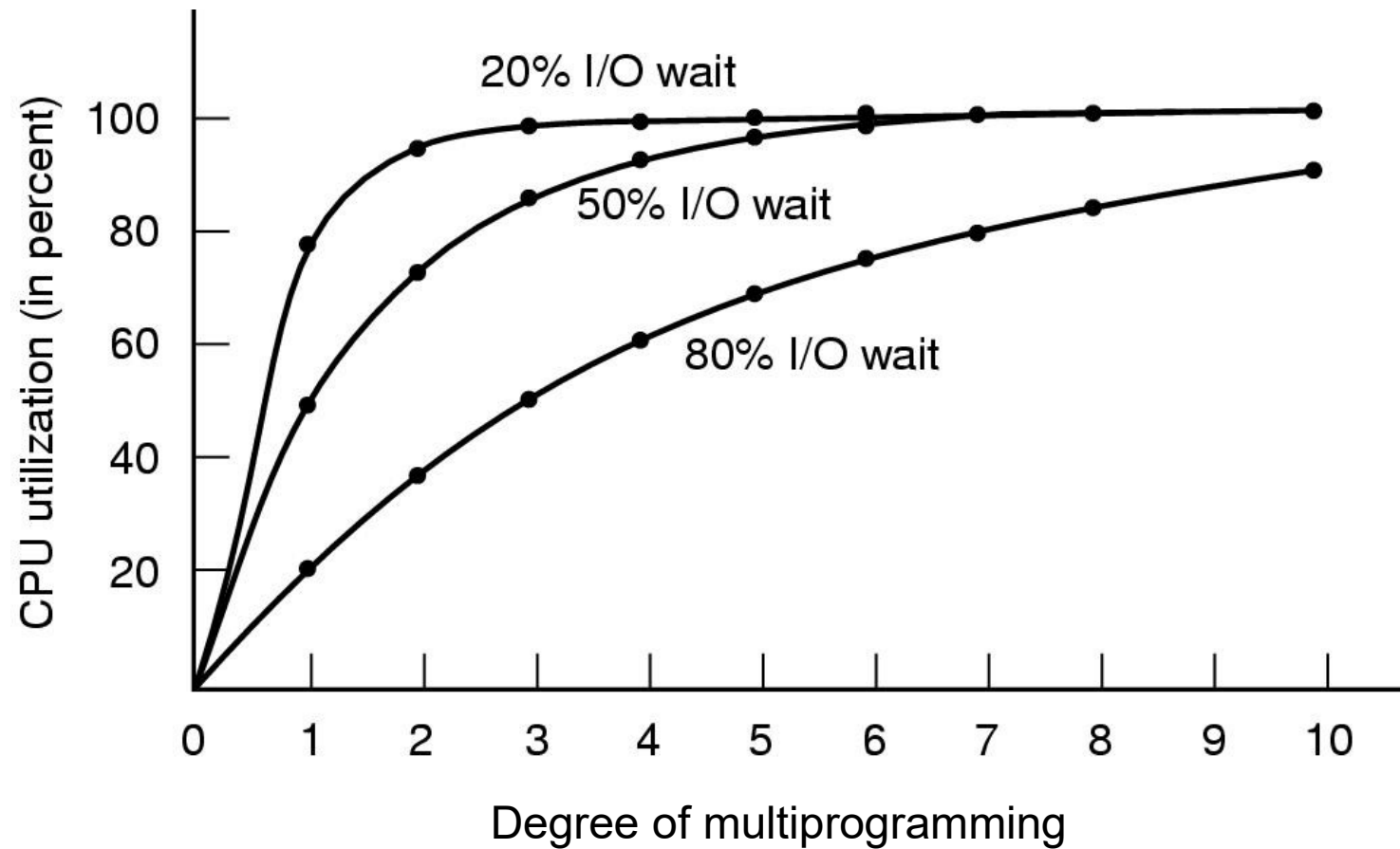
CPU utilization as a function of number of processes in memory

KE1   Kevin Elphinstone, 30/03/2020

General problem: How to divide memory between processes?

where is the free memory
how do we allocate the free memory for programs to run

- Given a workload, how to we
  - Keep track of free memory?
  - Locate free memory for a new process?
- Overview of evolution of simple memory management
  - Static (fixed partitioning) approaches
    - Simple, predicable workloads of early computing
  - Dynamic (partitioning) approaches
    - More flexible computing as compute power and complexity increased.
- Introduce virtual memory
  - Segmentation and paging

Process B

Process C

Process D

UNSW
SYDNEY
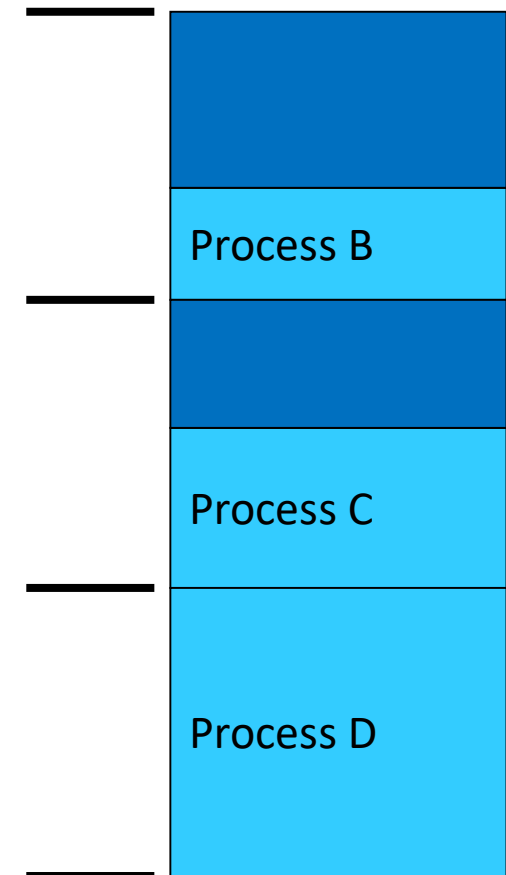
Problem: How to divide memory

- One approach
  - divide memory into fixed equal-sized partitions
  - Any process <= partition size can be loaded into any partition
  - Partitions are free or busy

Process A

internal fragmentation

Process B

Process C

Process D

UNSW
SYDNEY
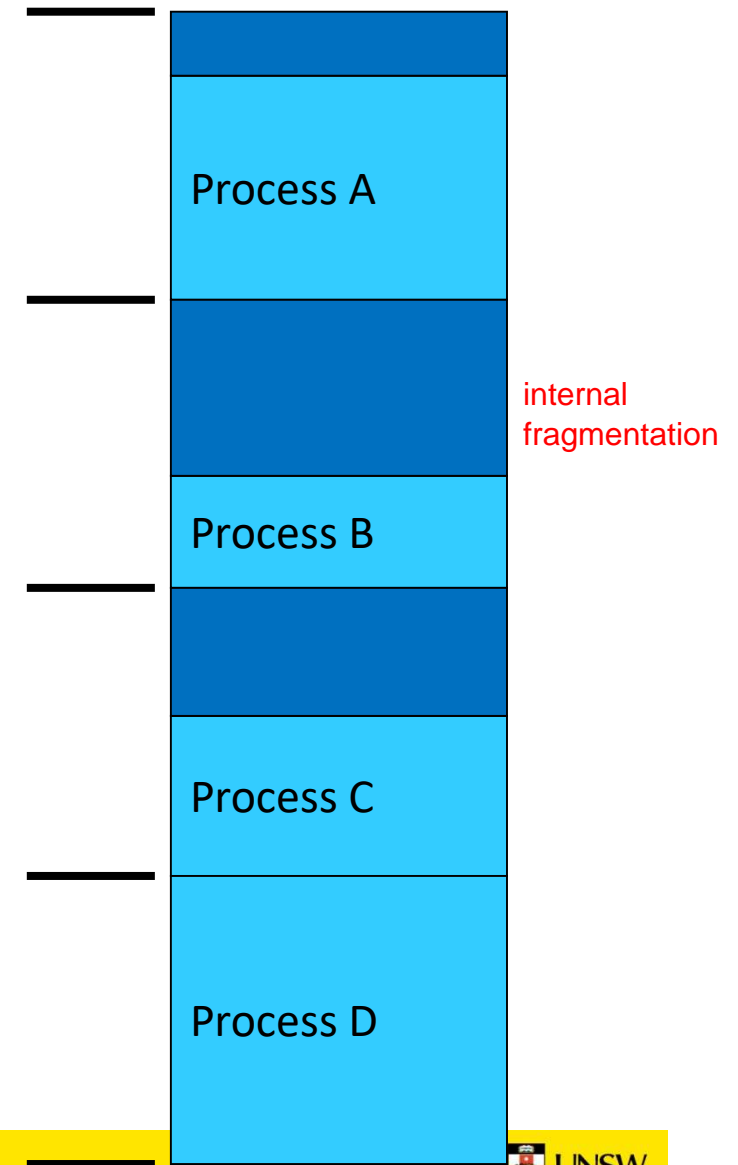
Simple MM: Fixed, equal-sized partitions

- Any unused space in the partition is wasted
  - Called internal fragmentation
- Processes smaller than main memory, but larger than a partition cannot run.

Process A

Process B

Process C

Process D

UNSW
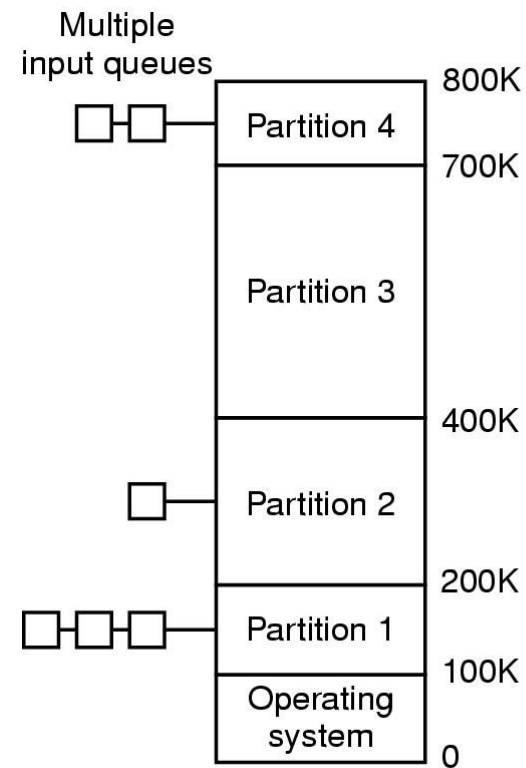SYDNEY
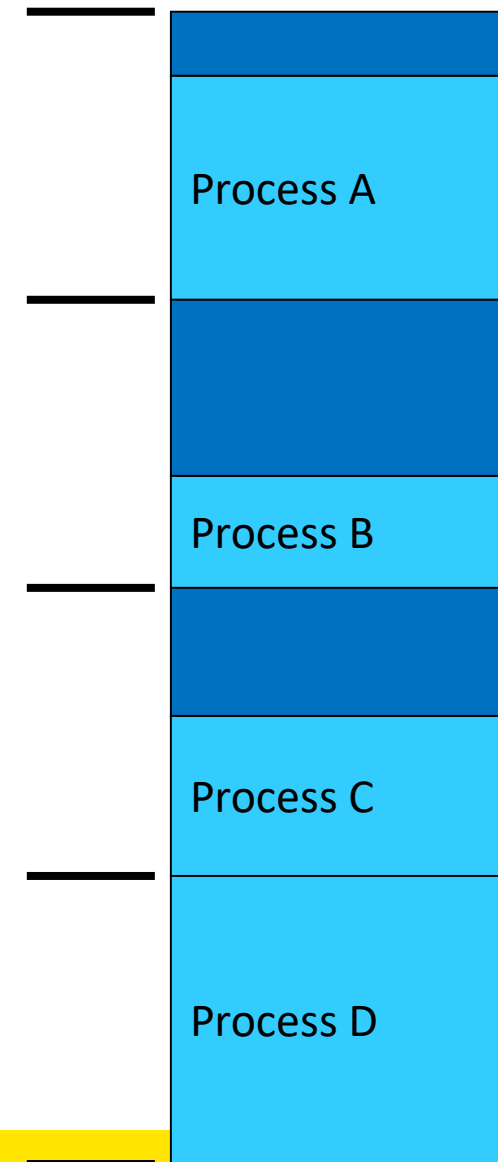
Simple MM: Fixed, variable-sized partitions

- Divide memory at boot time into a selection of different sized partitions
  - Can base sizes on expected workload
- Each partition has queue:
  - Place process in queue for smallest partition that it fits in.
  - Processes wait for when assigned partition is empty to start
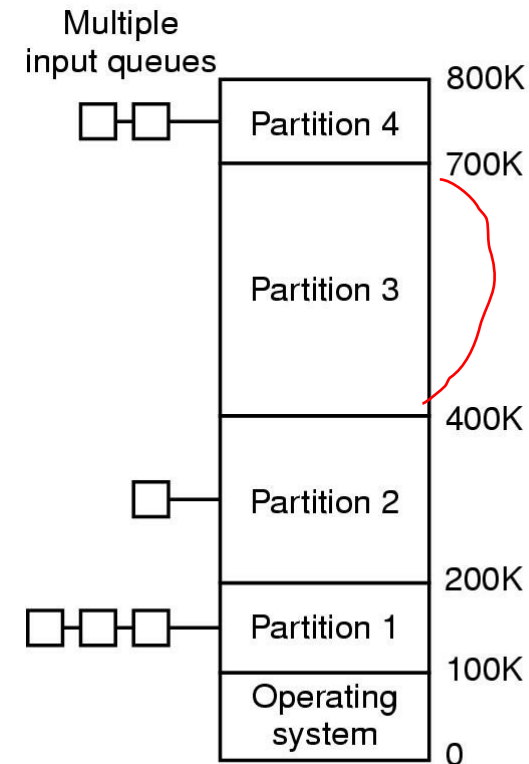
(a)

- Issue
 - Some partitions may be idle
    - Small jobs available, but only large partition free
    - Workload could be unpredictable

Multiple input queues

Partition 4 — 800K
700K

Partition 3
400K

Partition 2
200K

Partition 1
100K

Operating system
0

(a)

Alternative queue strategy

- Single queue, search for any jobs that fit
  - Small jobs in large partition if necessary
  - Increases internal memory fragmentation

Single
input queue

Partition 4

Partition 3

Partition 2

Partition 1

Operating
system

(b)
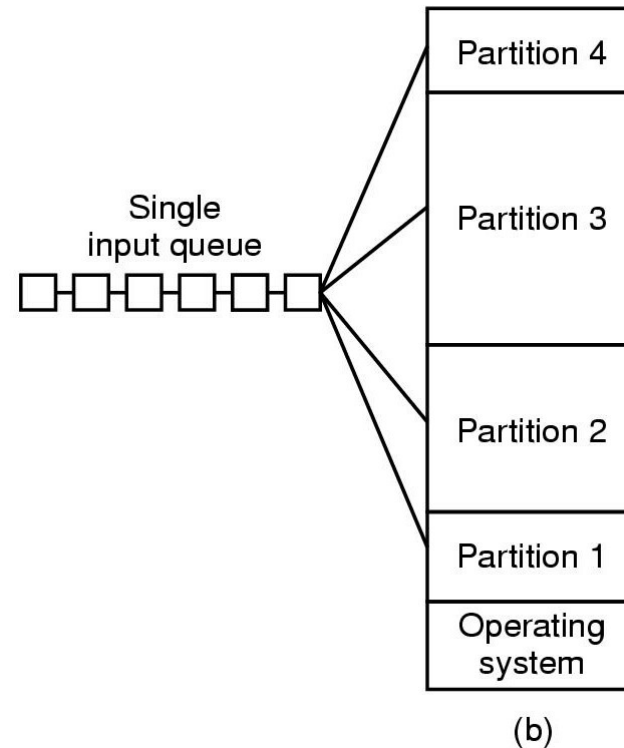
Fixed Partition Summary

- Simple

- Easy to implement

- Can result in poor memory utilisation
  - Due to internal fragmentation

- Used on IBM System 360 operating system (OS/MFT)
  - Announced 6 April, 1964

- Still applicable for simple embedded systems
  - Static workload known in advance

Dynamic Partitioning get rid of the internal fragmentation program

- Partitions are of variable length
  - Allocated on-demand from ranges of free memory

- Process is allocated exactly what it needs
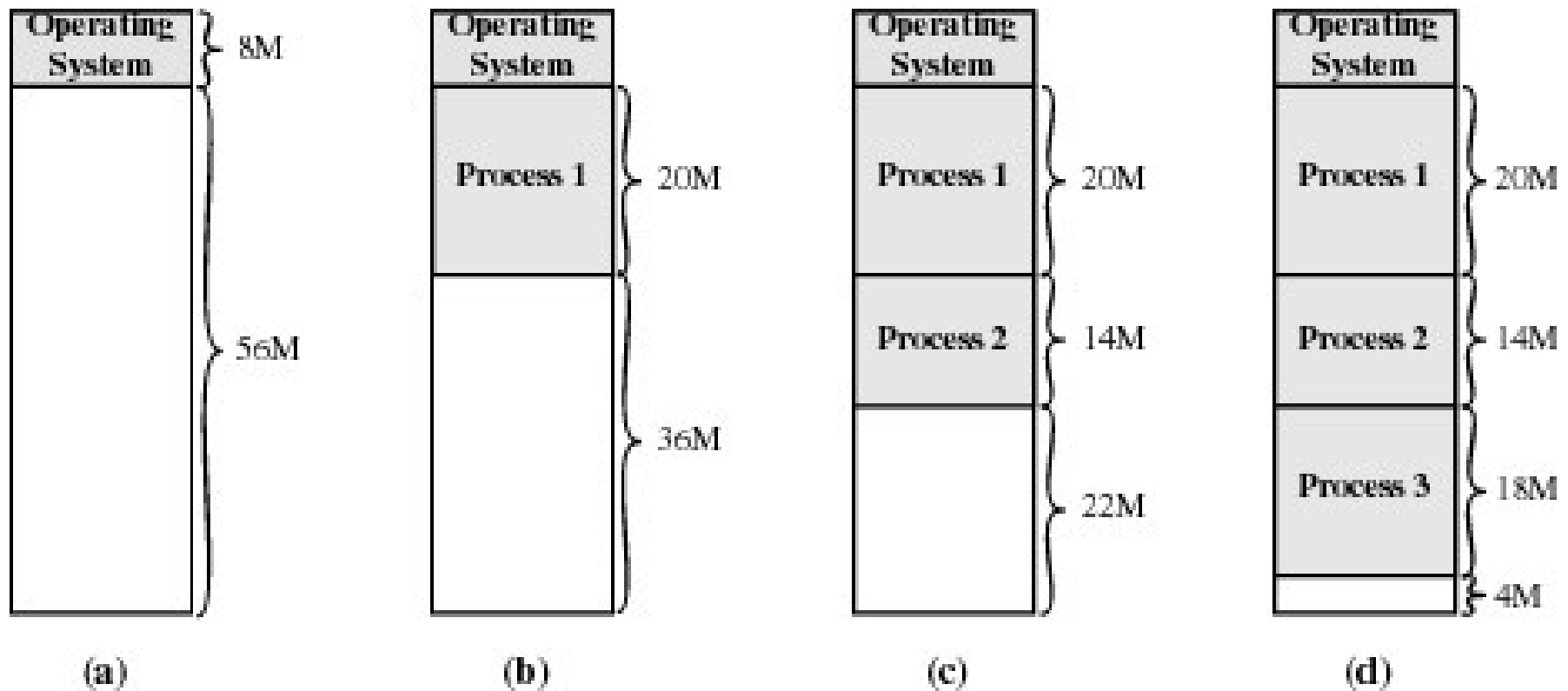 - Assumes a process knows what it needs

Figure 7.4 The Effect of Dynamic Partitioning

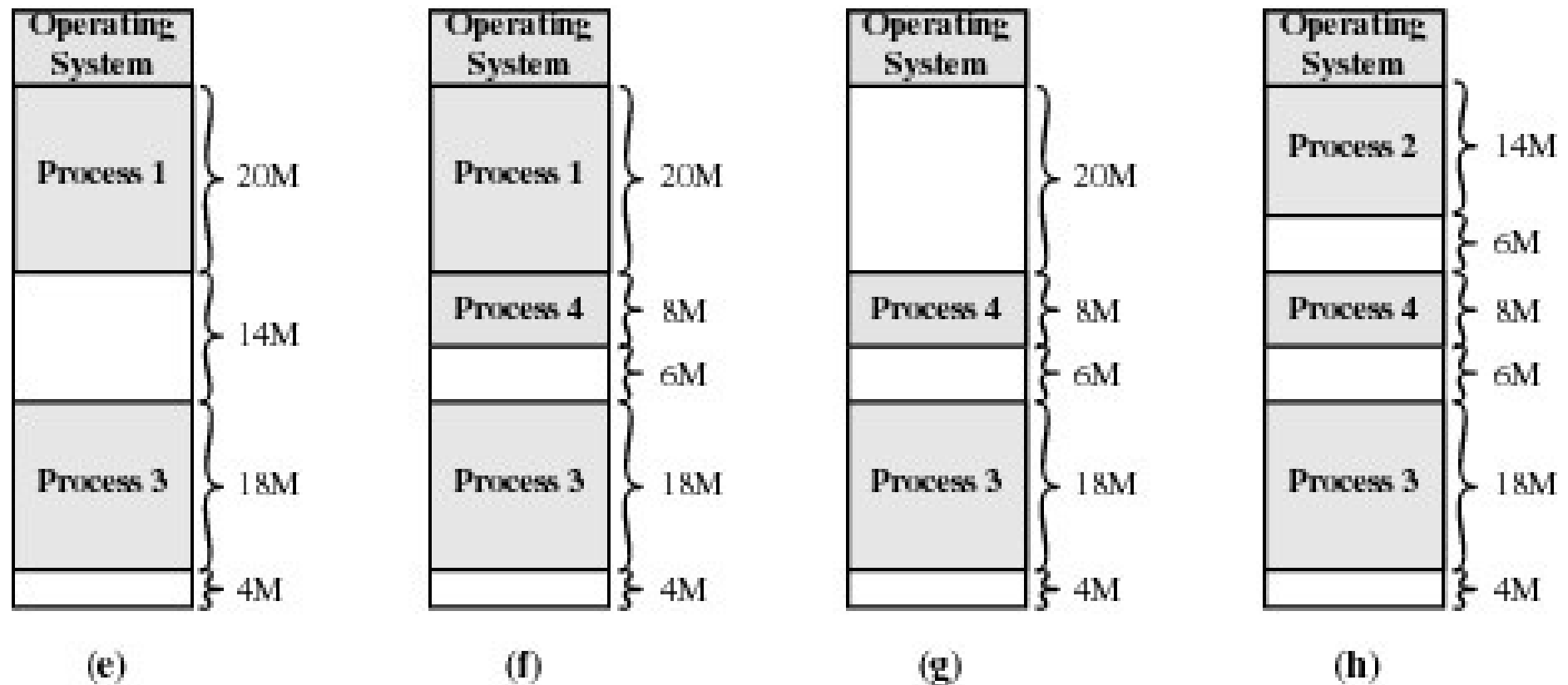this leaves a lot of small holes on our memory

Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning

- In previous diagram
  - We have 16 meg free in total, but it can't be used to run any more processes requiring > 6 meg as it is fragmented
  - Called *external fragmentation*
- We end up with unusable holes

Recap: Fragmentation

- **External Fragmentation**:
  - The space wasted external to the allocated memory regions.
  - Memory space exists to satisfy a request, but it is unusable as it is not contiguous.
- **Internal Fragmentation:**
  - The space wasted internal to the allocated memory regions.
  - allocated memory may be slightly larger than requested memory; this size difference is wasted memory internal to a partition.

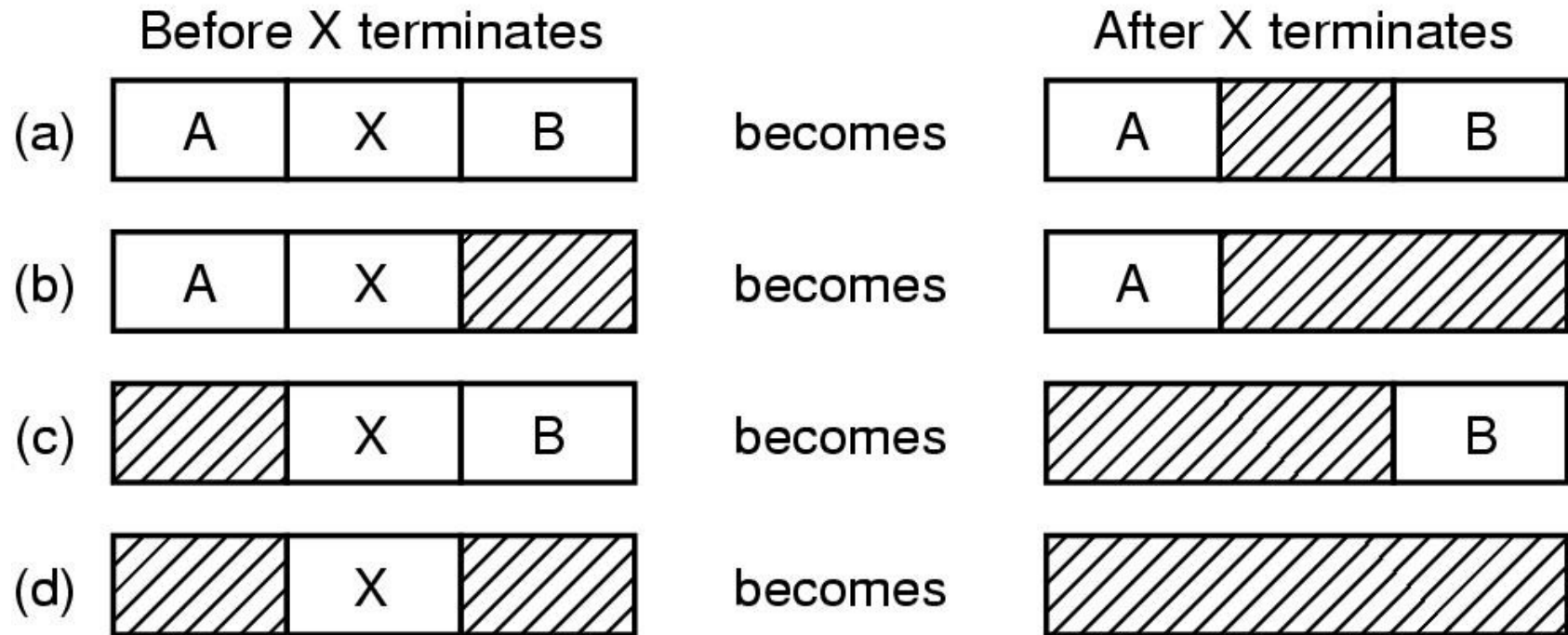# Dynamic Partition Allocation Algorithms

- Also applicable to `malloc()`-like in-application allocators

- Given a region of memory, basic requirements are:
  - Quickly locate a free partition satisfying the request
    - Minimise CPU time search
  - Minimise external fragmentation
  - Minimise memory overhead of bookkeeping
  - Efficiently support merging two adjacent free partitions into a larger partition

Classic Approach

- Represent available memory as a linked list of available "holes" (free memory ranges).
  - Base, size
  - Kept in order of increasing address
    - Simplifies merging of adjacent holes into larger holes.
  - List nodes be stored in the "holes" themselves
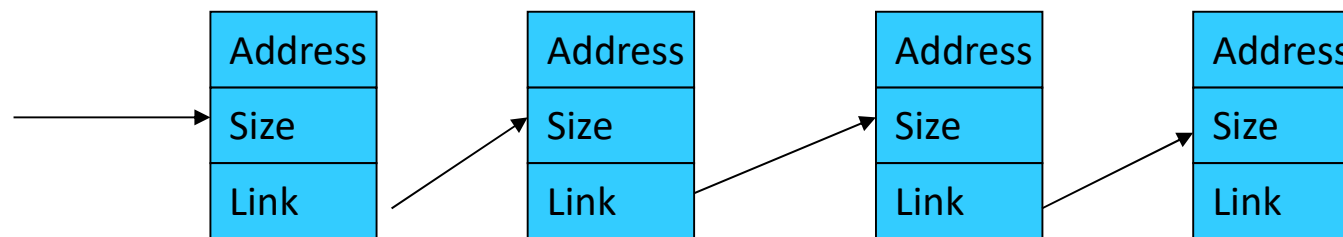
Coalescing Free Partitions with Linked Lists

Four neighbor combinations for the terminating process X

Dynamic Partitioning Placement Algorithm

- First-fit algorithm
  - Scan the list for the first entry that fits
    - If greater in size, break it into an allocated and free part
    - Intent: Minimise amount of searching performed
  - Aims to find a match quickly
  - Biases allocation to one end of memory
  - Tends to preserve larger blocks at the end of memory

Address		Address		Address		Address
Size	→	Size	→	Size	→	Size
Link		Link		Link		Link
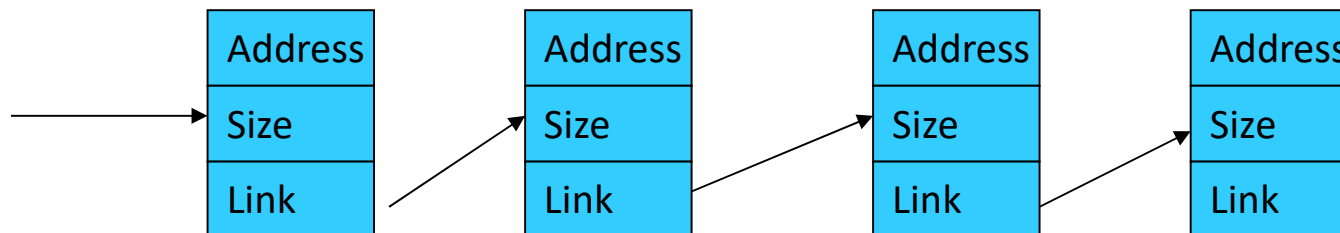
Dynamic Partitioning Placement Algorithm

- Next-fit
  - Like first-fit, except it begins its search from the point in list where the last request succeeded instead of at the beginning.
    - (Flawed) Intuition: spread allocation more uniformly over entire memory to avoid skipping over small holes at start of memory
    - Performs worse than first-fit as it breaks up the large free space at end of memory.

Address
Size
Link

Address
Size
Link

Address
Size
Link

Address
Size
Link
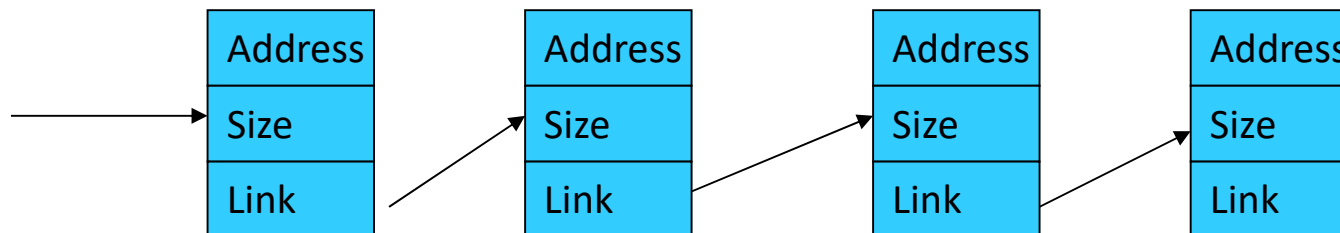
Dynamic Partitioning Placement Algorithm

- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Performs worse than first-fit
    - Has to search complete list
      - does more work than first-fit
    - Since smallest block is chosen for a process, the smallest amount of external fragmentation is left
    - Create lots of unusable holes
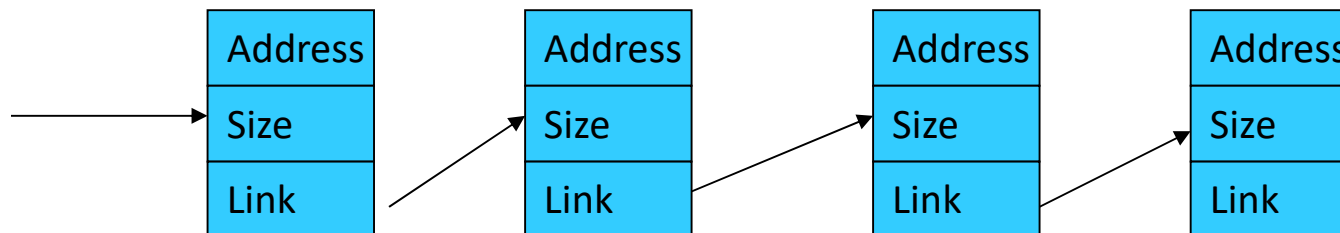
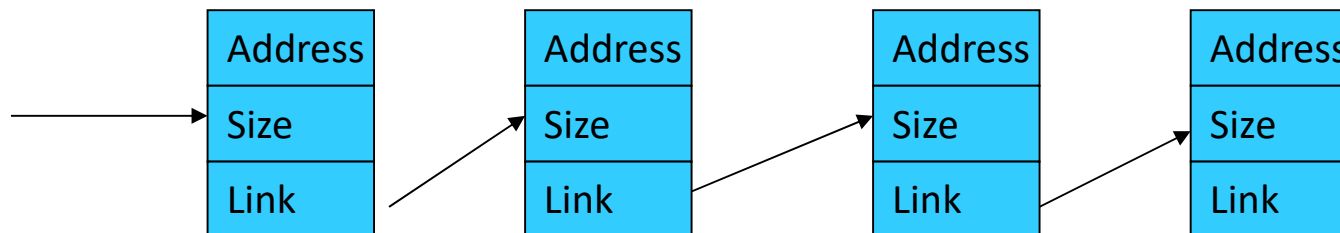# Dynamic Partitioning Placement Algorithm

- **Worst-fit algorithm**
  - Chooses the block that is largest in size (worst-fit)
    - (whimsical) idea is to leave a usable fragment left over
  - Poor performer
    - Has to do more work (like best fit) to search complete list
    - Does not result in significantly less fragmentation

Address
Size
Link

Address
Size
Link

Address
Size
Link

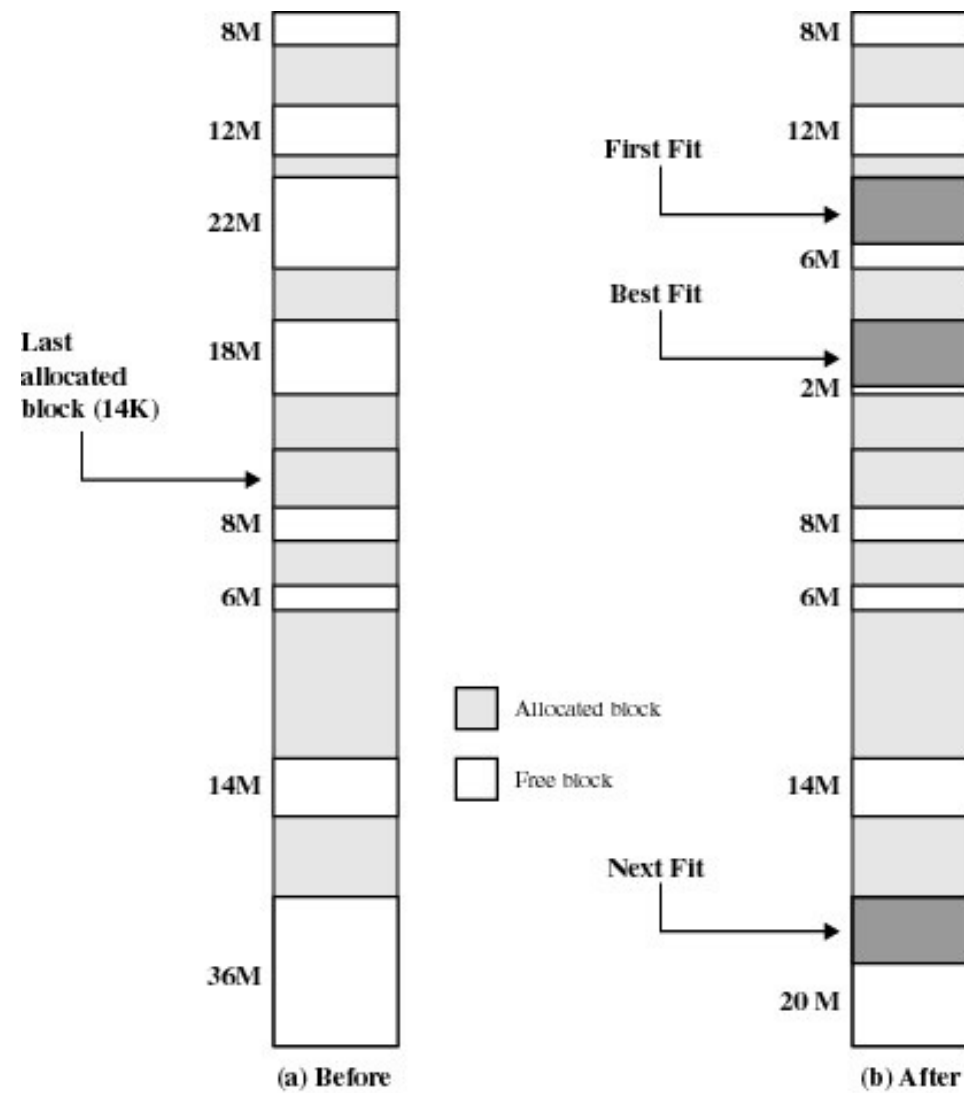| Address |
|---------|
| Size |
| Link |

UNSW
SYDNEY

Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Dynamic Partition Allocation Algorithm

- Summary
  - First-fit generally better than the others and easiest to implement

- You should be aware of them
  - They are simple solutions to a still-existing OS or application service/function – memory allocation.

- Note: Largely have been superseded by more complex and specific allocation strategies
  - Typical in-kernel allocators used are *lazy buddy,* and *slab* allocators

Compaction

- We can reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Only if we can relocate running programs?
    - Pointers?
  - Generally requires hardware support

this is not easy
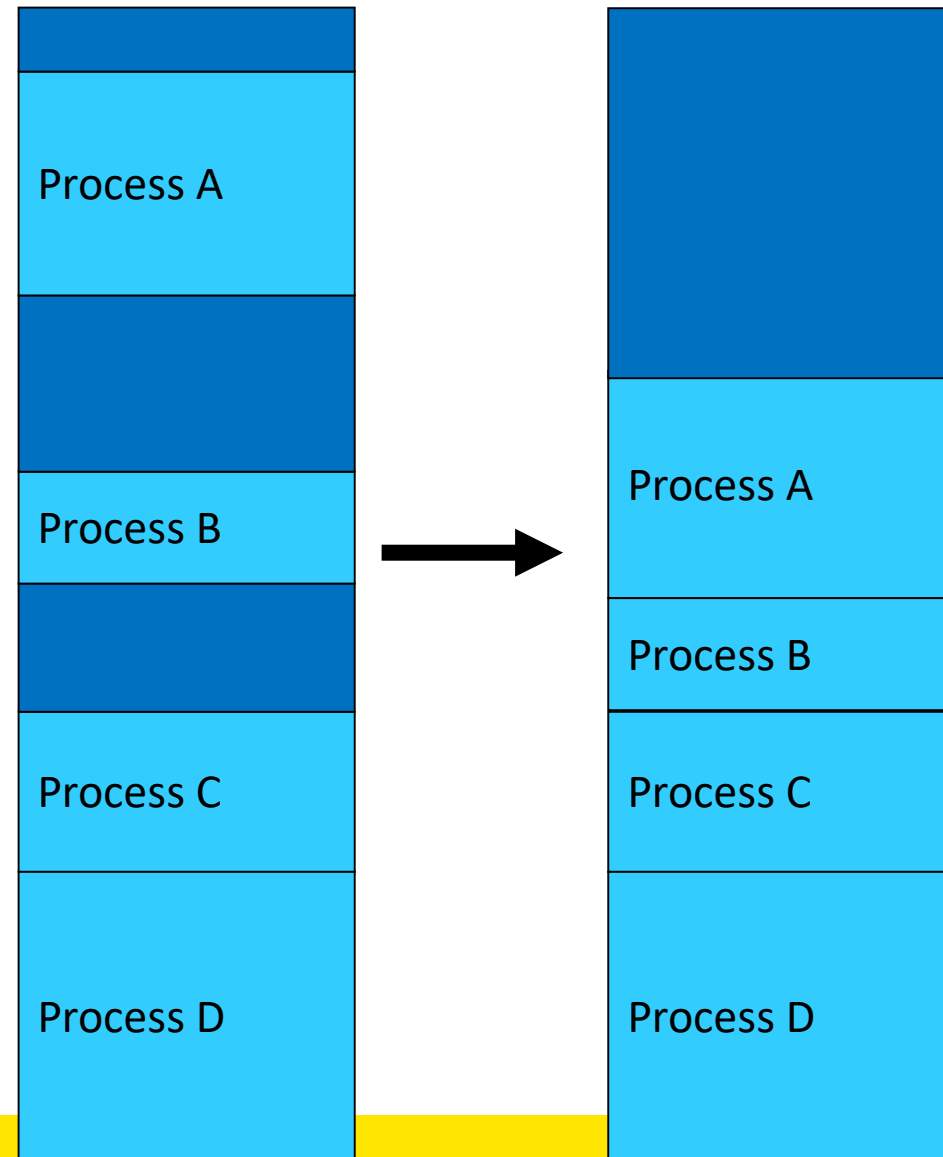
Some Remaining Issues with Dynamic Partitioning

- We have ignored
  - Relocation
    - How does a process run in different locations in memory?
  - Protection
    - How do we prevent processes interfering with each other

Process A

Process B

Process C

Process D
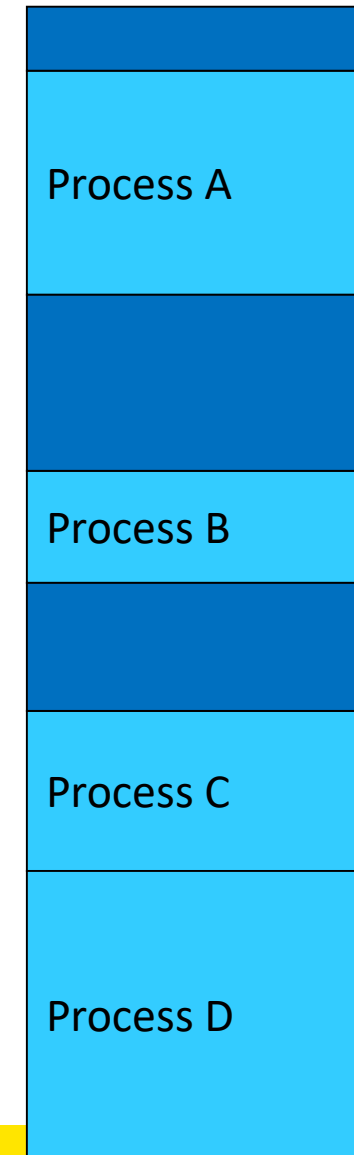
Example Logical Address-Space Layout

- Logical addresses refer to specific locations within the program

- Once running, these address must refer to real physical memory

- When are logical addresses bound to physical?

Process control information

Entry point to program

Increasing address values

Current top of stack

Process Control Block	0x0000
Program	Branch instruction
Data	Reference to data
Stack	0xFFFF

Figure 7.1   Addressing Requirements for a Process
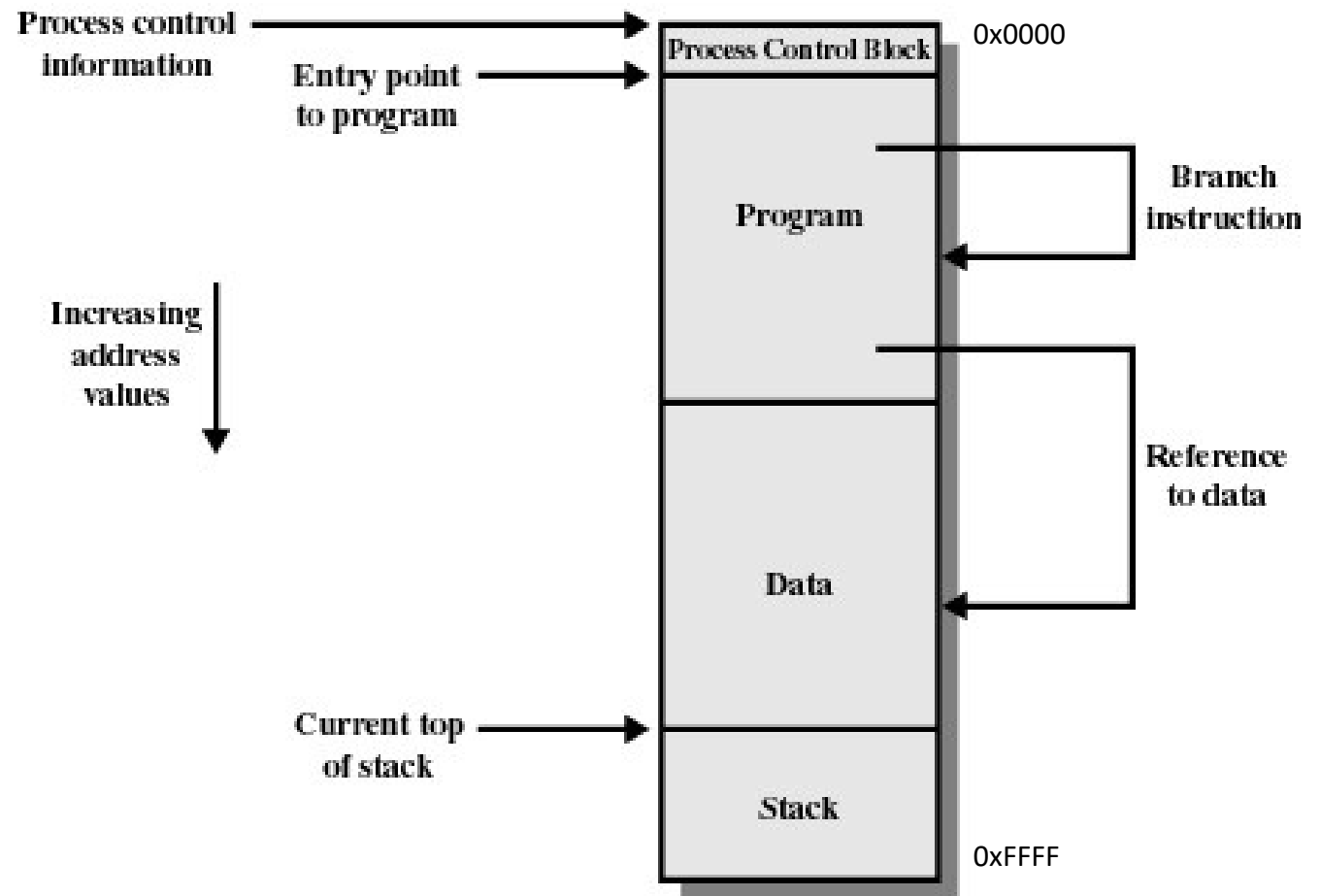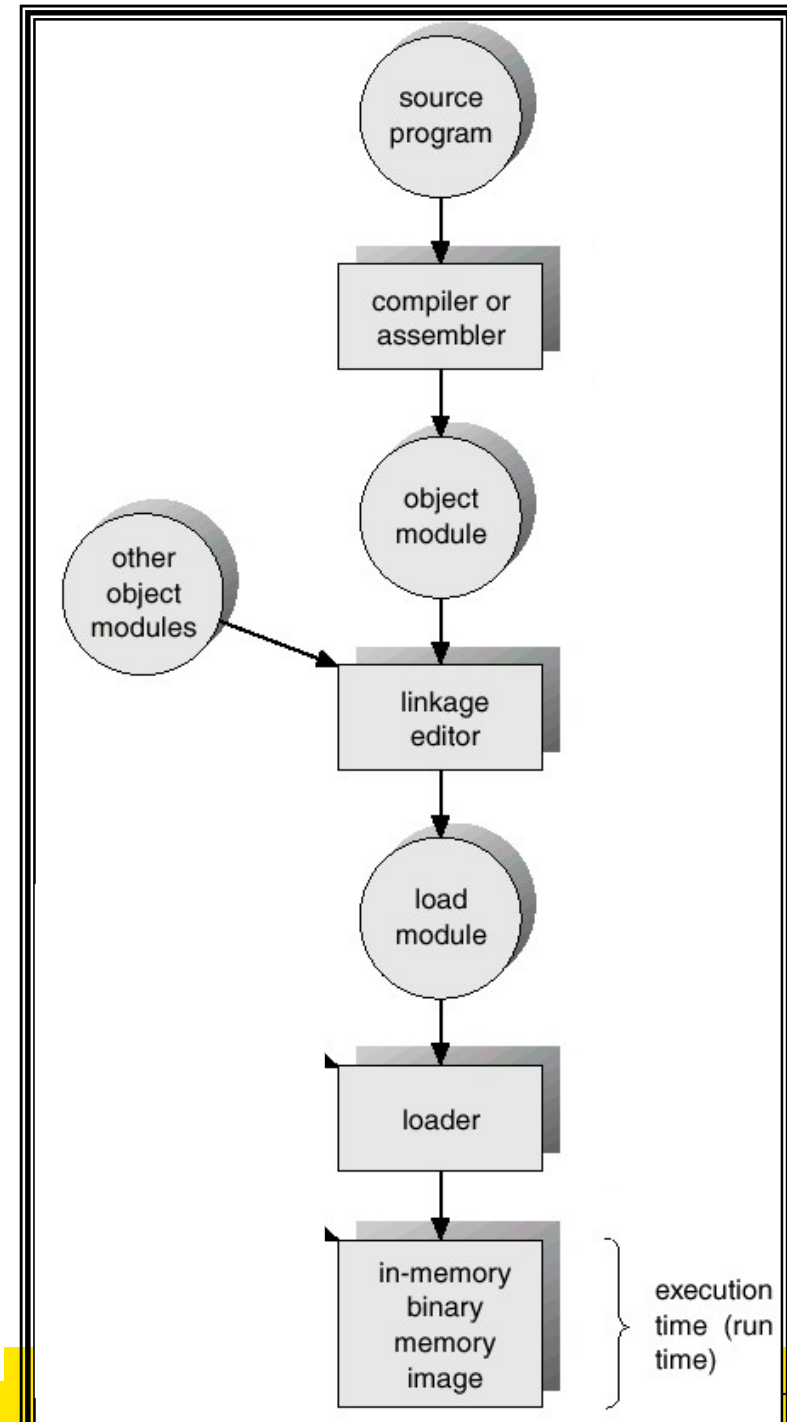
When are memory addresses bound?

- Compile/link time
  - Compiler/Linker binds the addresses
  - Must know "run" location at compile time
  - Recompile if location changes

- Load time
  - Compiler generates *relocatable* code
  - Loader binds the addresses at load time

- Run time
  - Logical compile-time addresses translated to physical addresses by *special hardware.*

Hardware Support for Runtime Binding and Protection

- For process B to run using logical addresses
  - Process B expects to access addresses from zero to some limit of memory size
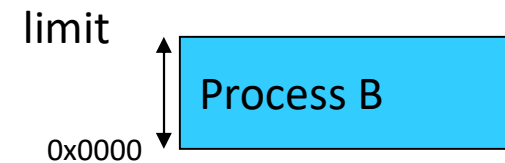
limit

Process B

0x0000

Hardware Support for Runtime Binding and Protection

- For process B to run using logical addresses
  - Need to add an appropriate offset to its logical addresses
    - Achieve relocation
    - Protect memory "lower" than B
  - Must limit the maximum logical address B can generate
    - Protect memory "higher" than B

0xFFFF

limit Process B base

0x0000

UNSW
SYDNEY
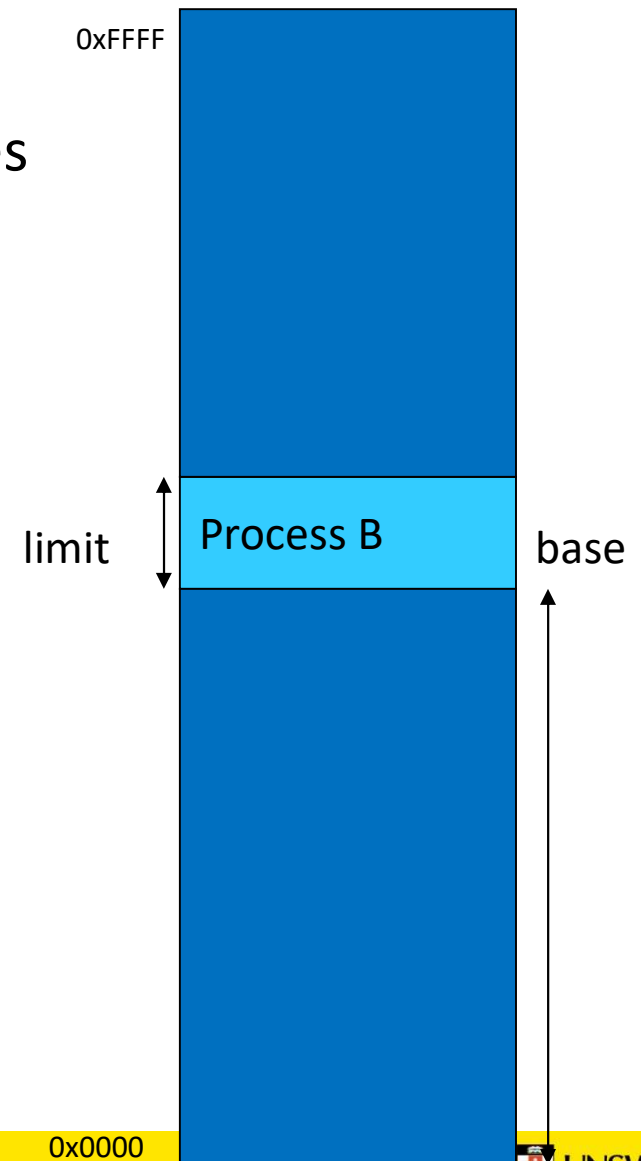
Hardware Support for Relocation and Limit Registers
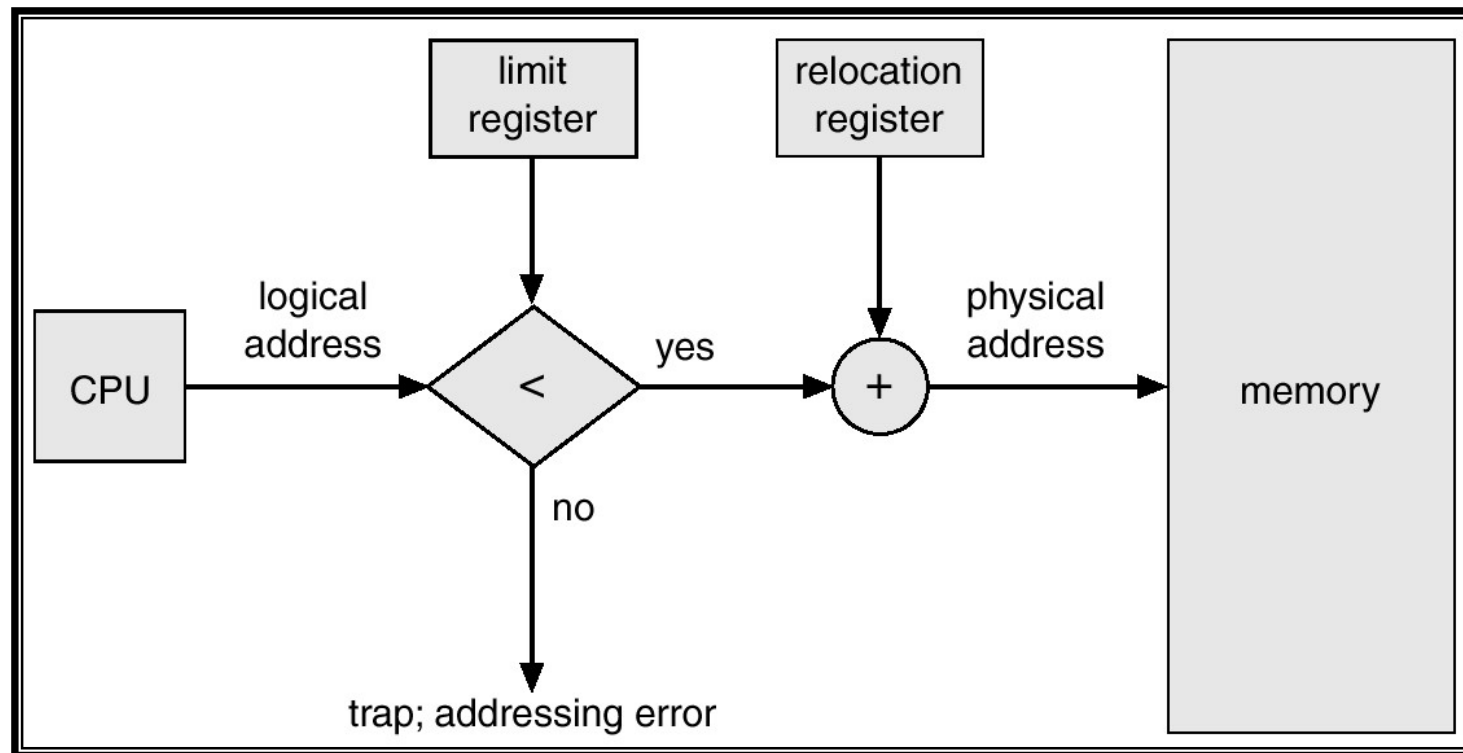
Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch

base=0x8000

limit = 0x2000

0xFFFF

Process A

0x9FFF
limit
0x8000

0x1FFF
0x0000   Process B   base

Process C

Process D

0x0000

UNSW
SYDNEY

Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch

base=0x4000

limit = 0x3000

0xFFFF

Process A

Process B

0x6FFF

0x2FFF

limit

Process C

0x0000

0x4000

base

Process D

0x0000

UNSW
SYDNEY
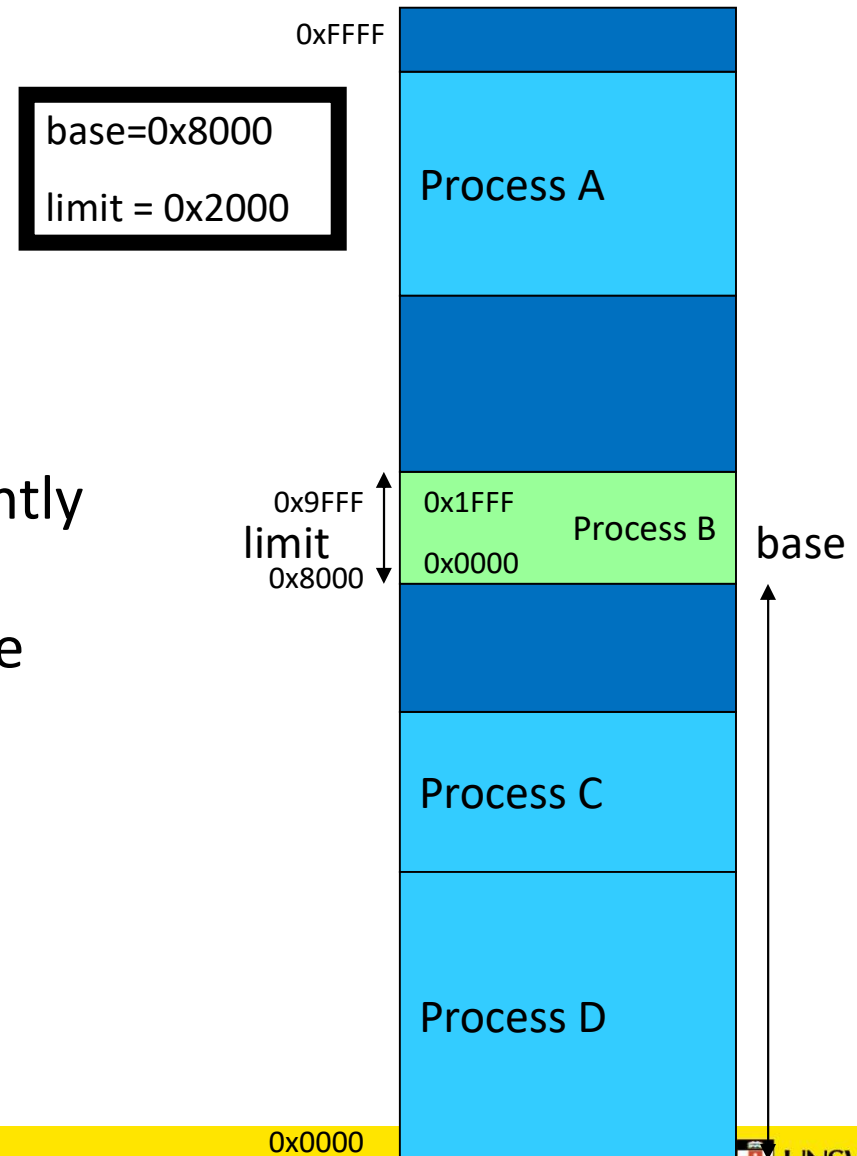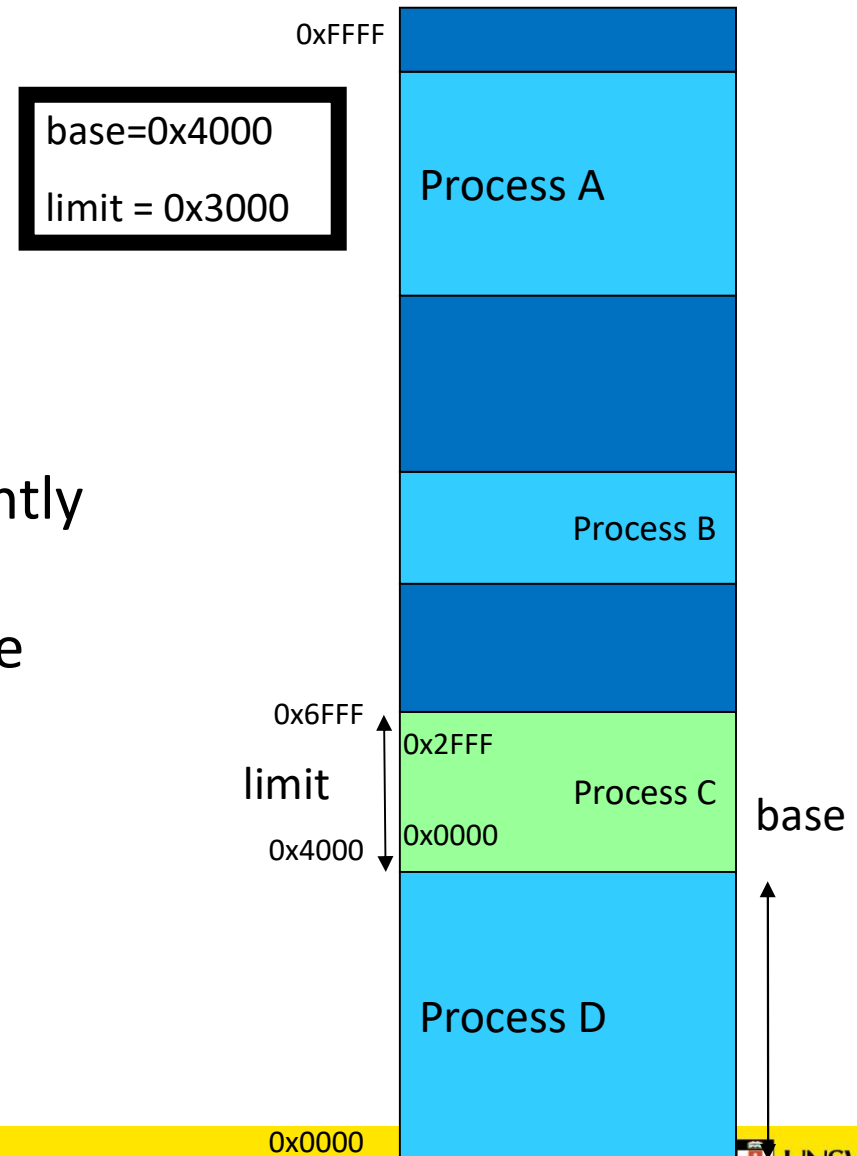
Base and Limit Registers

- Pro
  - Supports protected multi-processing (-tasking)

- Cons
  - Physical memory allocation must still be contiguous
  - The entire process must be in memory
  - Do not support partial sharing of address spaces
    - No shared code, libraries, or data structures between processes

Timesharing

- Thus far, we have a system suitable for a batch system
  - Limited number of dynamically allocated processes
    - Enough to keep CPU utilised
  - Relocated at runtime
  - Protected from each other
- But what about timesharing?
  - We need more than just a small number of processes running at once
  - Need to support a mix of active and inactive processes, of varying longevity

0xFFFF

Process A

Process B

Process C

Process D

0x0000

UNSW
SYDNEY
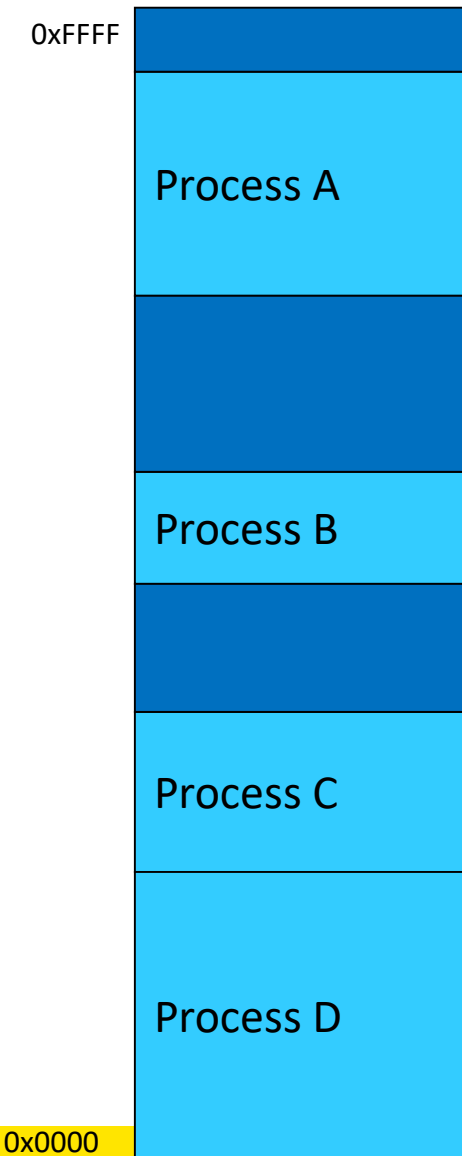
Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.

- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- Can prioritize – lower-priority process is swapped out so higher-priority process can be loaded and executed.

- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
  - slow

Schematic View of Swapping
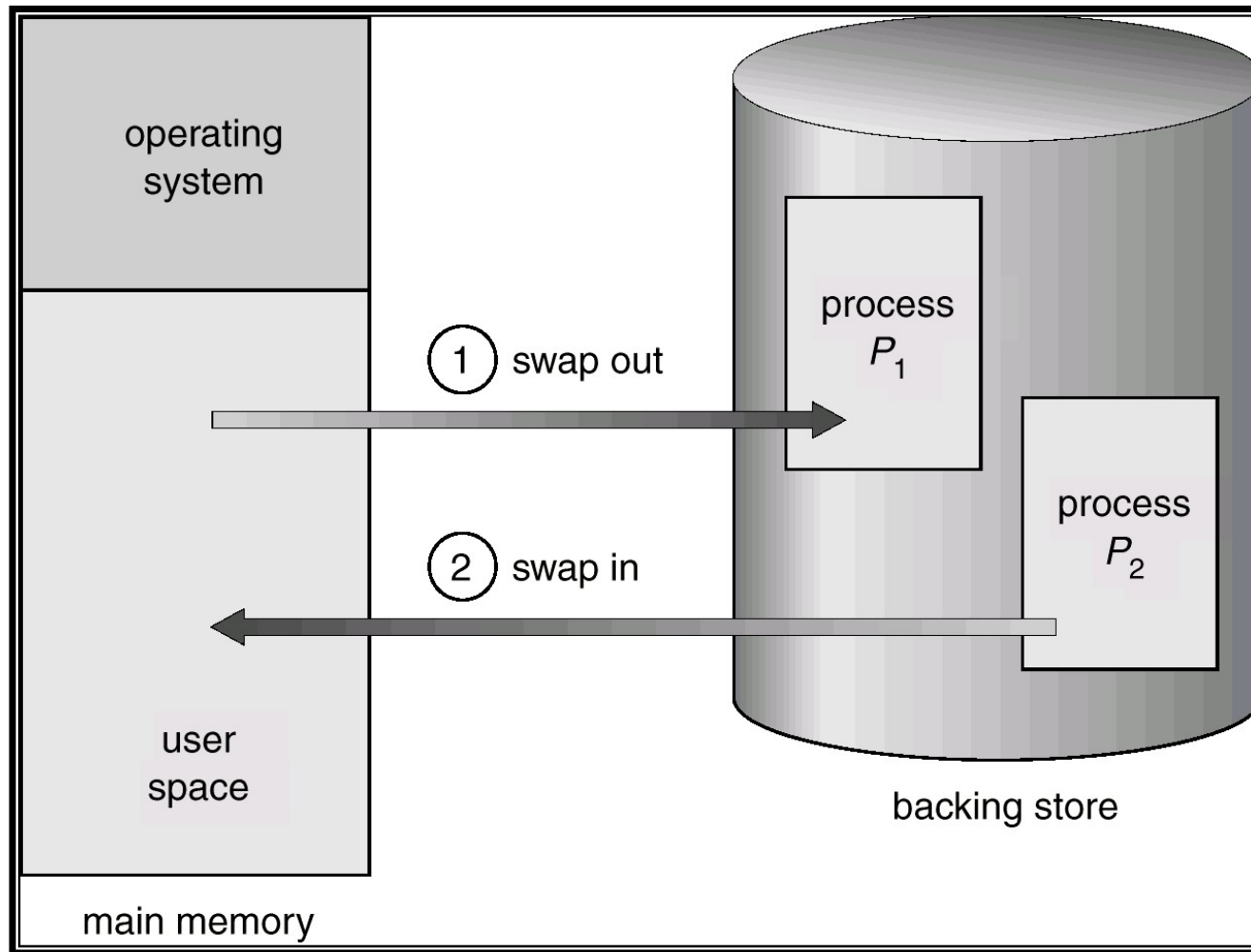
So far we have assumed a process is smaller than memory

- What can we do if a process is larger than main memory?

Virtual Memory

- Developed to address the issues identified with the simple schemes covered thus far.
- Two classic variants
  - Paging
  - Segmentation
    - (no longer covered in course, see textbook if interested)

- Paging is now the dominant one of the two
  - We'll focus on it
- Some architectures support hybrids of the two schemes
  - E.g. Intel IA-32 (32-bit x86)
    - Becoming less relevant

Virtual Memory – Paging Overview

- Partition physical memory into small equal sized chunks
  - Called *frames*
- Divide each process's virtual (logical) address space into same size chunks
  - Called pages
  - Virtual memory addresses consist of a *page number* and *offset* within the page
- OS maintains a *page table*
  - contains the frame location for each page
  - Used by *hardware* to translate each virtual address to physical address
  - The relation between virtual addresses and physical memory addresses is given by page table
- Process's physical memory does **not** have to be contiguous

Virtual address space

Range	Page
60K-64K	X
56K-60K	X
52K-56K	X
48K-52K	X
44K-48K	7
40K-44K	X
36K-40K	5
32K-36K	X
28K-32K	X
24K-28K	X
20K-24K	3
16K-20K	4
12K-16K	0
8K-12K	6
4K-8K	1
0K-4K	2

} Virtual page

Physical memory address

28K-32K
24K-28K
20K-24K
16K-20K
12K-16K
8K-12K
4K-8K
0K-4K

Page frame

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen Available Frames

	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process A

	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process B

Figure 7.9 Assignment of Process Pages to Free Frames

		Main memory				Main memory				Main memory
0		A.0		0		A.0		0		A.0
1		A.1		1		A.1		1		A.1
2		A.2		2		A.2		2		A.2
3		A.3		3		A.3		3		A.3
4		B.0		4				4		D.0
5		B.1		5				5		D.1
6		B.2		6				6		D.2
7		C.0		7		C.0		7		C.0
8		C.1		8		C.1		8		C.1
9		C.2		9		C.2		9		C.2
10		C.3		10		C.3		10		C.3
11				11				11		D.3
12				12				12		D.4
13				13				13		
14				14				14		

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

Process A page table: 0 → 0, 1 → 1, 2 → 2, 3 → 3

Process B page table: 0 → —, 1 → —, 2 → —

Process C page table: 0 → 7, 1 → 8, 2 → 9, 3 → 10

Process D page table: 0 → 4, 1 → 5, 2 → 6, 3 → 11, 4 → 12

Free frame list: 13, 14

Paging

- No external fragmentation

- Small internal fragmentation (in last page)

- Allows sharing by *mapping* several pages to the same frame

- Abstracts physical organisation
  - Programmer only deal with virtual addresses

- Minimal support for logical organisation
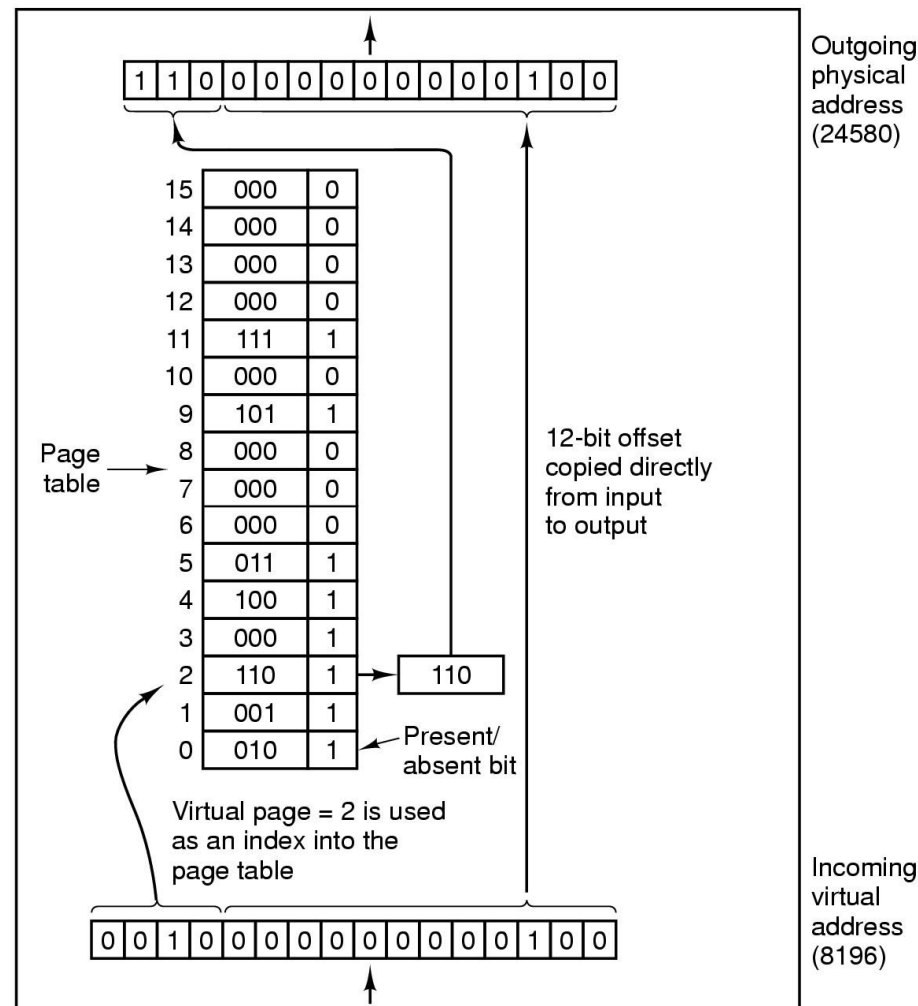 - Each unit is one or more pages

Memory Management Unit
(also called Translation Look-aside Buffer – TLB)
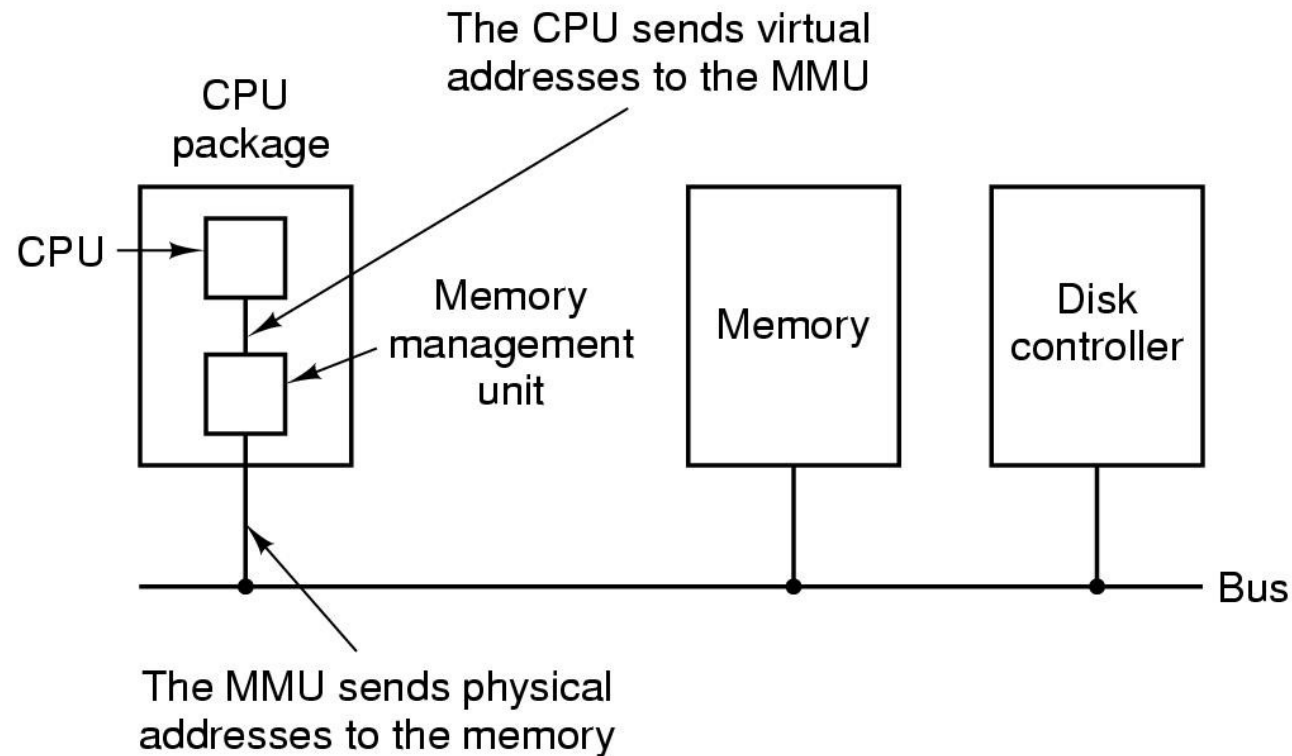
The position and function of the MMU

MMU Operation

Assume for now that the page table is contained wholly in registers within the MMU – in practice it is not

Internal operation of simplified MMU with 16 4 KB pages