University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 15: Memory Management 1

University of Nottingham,
Ningbo China 2024

Remember

Subject	# Lectures
Introduction to operating systems/computer design	1-2
Processes, process scheduling, threading, ...	3-4
Concurrency, Deadlock	5-8
Memory management, swapping, virtual memory,…	6-7
File System, file structure, management,…	4-5
Revision	2

Table: Course structure

Goals for Today

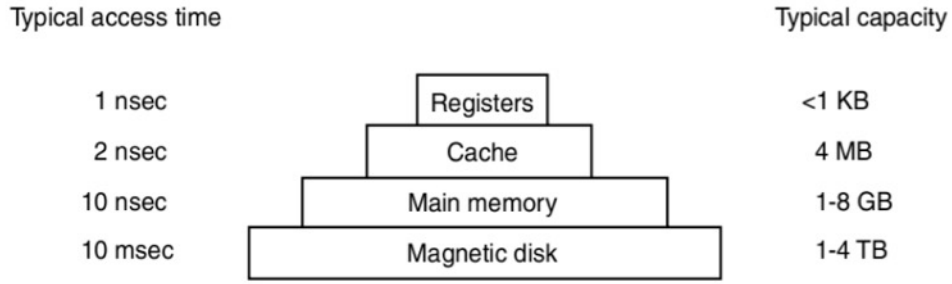
Overview

- **Introduction** to memory management
- **Modelling** of multi-programming
- Memory management based on **fixed partitioning**

Memory Management

Memory Hierarchies

- Computers typically have memory hierarchies:
  - Registers, L1/L2/L3 cache (volatile)
  - **Main memory (volatile)**
  - Disks (nonvolatile)



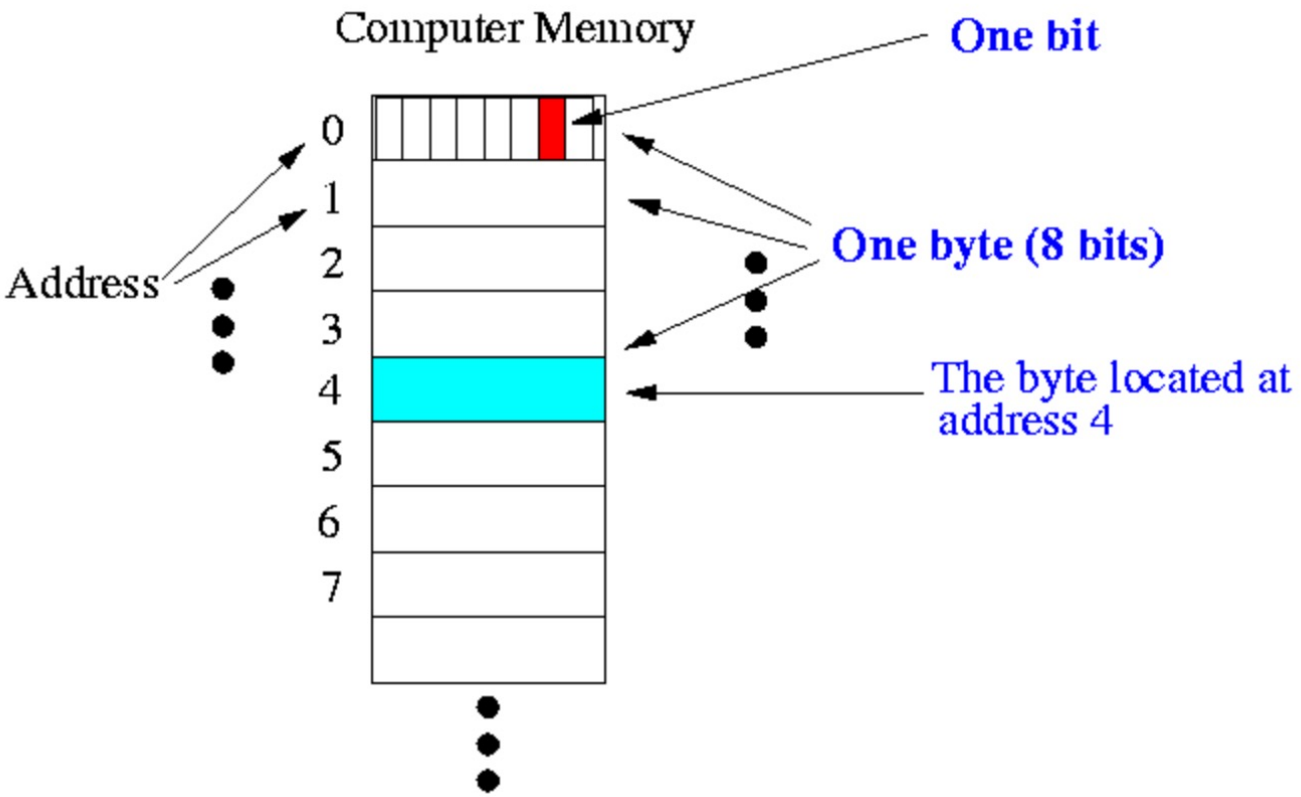
- "**Higher memory**" is faster, more expensive and volatile, "**lower memory**" is slower, cheaper, and non-volatile

- The operating system provides a **memory abstraction**

Memory Management

Memory Structure

- Memory can be seen as one **linear array** of bytes/words



Memory Management
OS Responsibilities

- **Allocate/deallocate** memory when requested by processes, **keep track** of **used/unused** memory
- **Transparently** move data from **memory** to **disk** and vice versa
- **Distribute memory** between processes and simulate an "**infinitely large**" memory space
- **Control access** when multiprogramming is applied

Figure: Representation of a process in memory

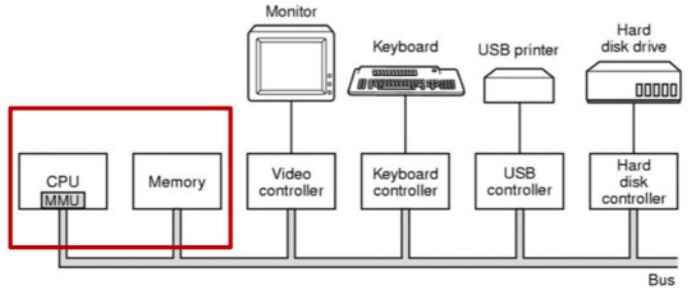


Figure: Simplified computer model (Tanenbaum, 2014)

Model
Approaches: Contiguous vs. Non-Contiguous

Figure: Contiguous

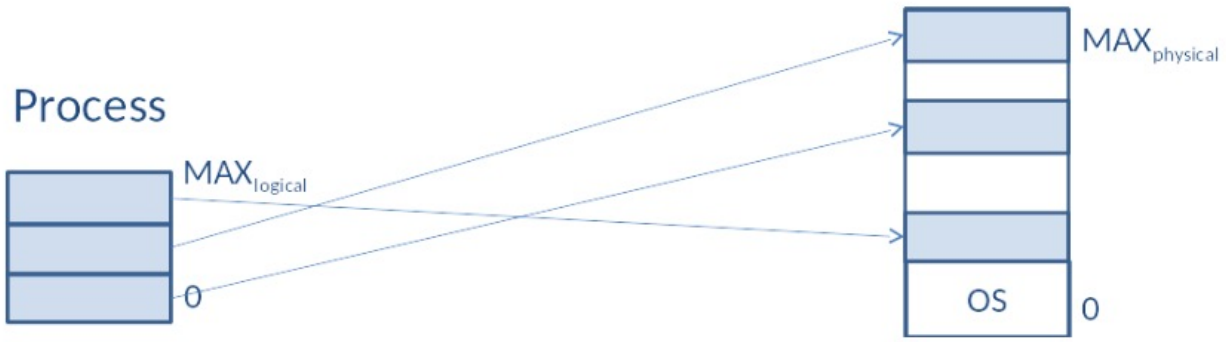


Figure: Non-contiguous

- **Contiguous memory management** models allocate memory in **one single block** without any **holes** or **gaps**
- **Non-contiguous memory management models** are capable of allocating memory in **multiple blocks**, or **segments**, which may be **placed anywhere in physical memory** (i.e., not necessarily next to each other)

Memory management has been evolved

- Approches
  - **Contiguous Memory**
    - ❖ **Mono-programming** [easy but low resource utilisation, cannot be used for modern multiprogramming machine]

      No relocation, physical address
    - ❖ **Multiprogramming** with **fixed partition** [multi-programming but internal fragmentation]

      Relocation (register), logical address
    - ❖ **Multiprogramming** with **dynamic partition** [low internal fragmentation but high external fragmentation]

      Relocation (register), logical address
  - Non-Contiguous Memory

    Relocation (table), logical address
    - ❖ **Paging** (fixed-partitioning/code re-location) [Internal fragmentation reduce to last block, no external fragmentation; Require page table to maintain page relocation]
    - ❖ **Virtual Memory** (locality →not all pages loaded) [more processes → CPU utilisation, no external fragmentation, more memory available; Need involve replacement algorithm, page table management, thrashing, variable/fixed resident set….]

Partitioning
Contiguous Approaches

- **Mono-programming:** one single partition for user processes
- **Multi-programming** with **fixed partitions**
  - Fixed **equal** sized partitions
  - Fixed **non-equal** sized partitions
- **Multi-programming** with **dynamic partitions**

Mono-Programming
No Memory Abstraction

- Only **one single user process** is in memory/executed at any point in time (no multi-programming)
- A fixed region of memory is allocated to the **OS/kernel**, the remaining memory is reserved for a **single process** (MS-DOS worked this way)
- This process has **direct access** to **physical memory** (i.e. no address translation takes place)

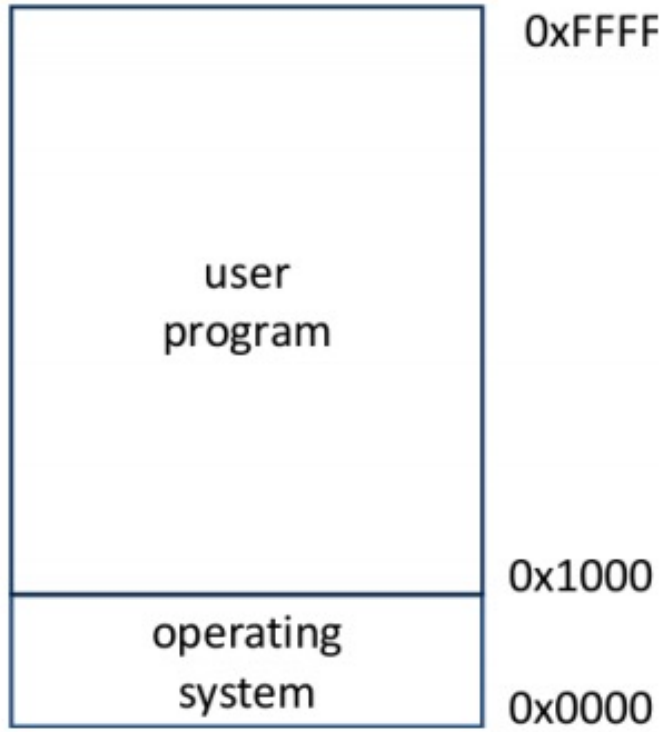


Figure: Mono-programming

Mono-Programming
No Memory Abstraction: Properties

- Every process is allocated **contiguous block of memory**, i.e. it contains no “holes” or “gaps” (⇔ **non-contiguous allocation**)
- **One process** is allocated the **entire memory space**, and the process is **always located in the same address space**
- **No protection** between different user processes required (one process)
- **Overlays** enable the programmer to use more memory than available (burden on programmer)

Mono-Programming
No Memory Abstraction: Properties (Cont.)

- **Shortcomings** of mono-programming:
  - Since a process has **direct access to the physical memory**, it may have **access to OS** memory
  - The operating system can be seen as a process - so we have **two processes** anyway
  - **Low utilisation** of hardware resources (CPU, I/O devices, etc.)
  - Mono-programming is unacceptable as **multiprogramming is expected** on modern machines
- Direct memory access and mono-programming is common in basic **embedded systems** and **modern consumer electronics**, e.g. washing machines, microwaves, etc.

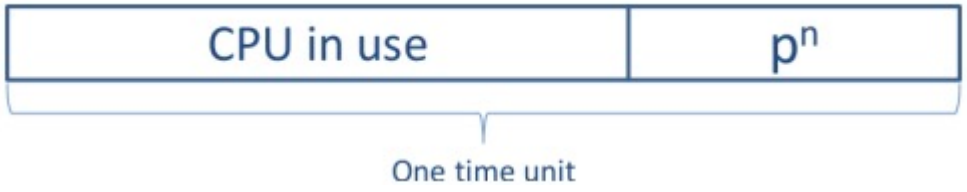
Why Multiprogramming
Simulating Multi-Programming

- What is multiprogramming? (Quiz!)

- Assuming that **multiprogramming** can **improve CPU utilisation**?
  - Intuitively, this is true
  - How do we model this?

Multi-Programming
A Probabilistic Model

- There are **n independent processes in memory**
- A process spends **p percent** of its time **waiting for I/O,** the probability that a single process is waiting for I/O is p
- The **probability** that all n processes are waiting for I/O (i.e., the CPU is idle) is p^n , i.e. p ×p ×p . . .
- **CPU Utilisation** is calculated as 1 minus the time that **all processes are waiting for I/O:** e.g., $p^n = 0.9$ then CPU utilisation = 1 - 0.9 ⇒ 0.1
- The **CPU utilisation** is given by $1-p^n$



Multi-Programming
A Probabilistic Model

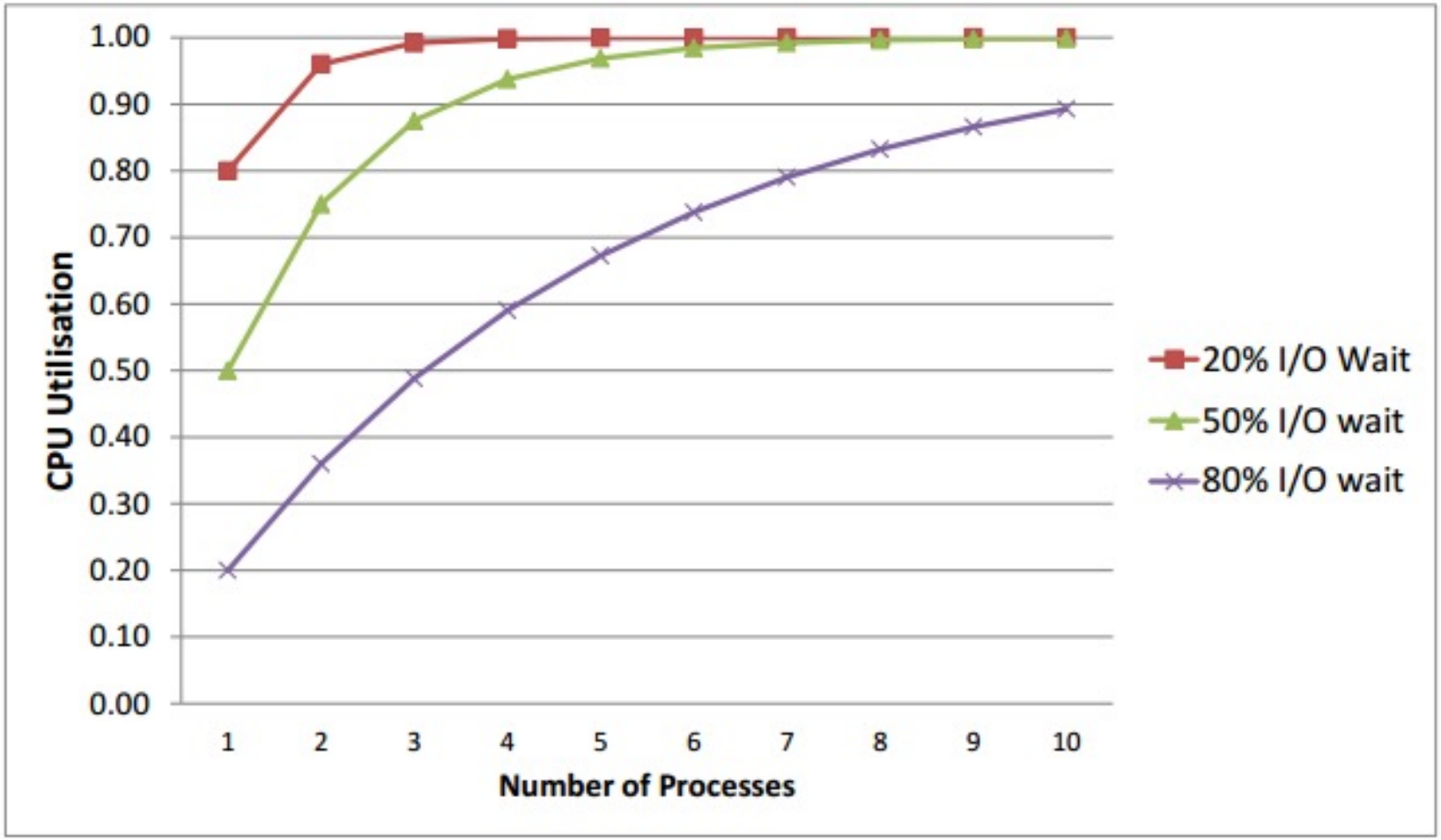
- With **an I/O wait time of 20%,** almost **100% CPU utilisation** can be achieved with four processes ($1-0.2^{4}$)
- With an **I/O wait time of 90%,** 10 processes can achieve about **65% CPU utilisation ($1-0.9^{10}$)**
- CPU **utilisation goes up** with **the number of processes** and down for **increasing levels of I/O ratio**

Multi-Programming
A Probabilistic Model

	I/O Ratio		
# Processes	0.2	0.5	0.8
1	0.80	0.50	0.20
2	0.96	0.75	0.36
3	0.99	0.88	0.49
4	1.00	0.94	0.59
5	1.00	0.97	0.67
6	1.00	0.98	0.74
7	1.00	0.99	0.79
8	1.00	1.00	0.83
9	1.00	1.00	0.87
10	1.00	1.00	0.89

Table: CPU utilisation as a function of the I/O ratio and the number of processes

Multi-Programming
A Probabilistic Model



Multi-Programming
A Probabilistic Model

- Assume that:
  - A computer has **one megabyte of memory**
  - The **OS takes up 200k**, leaving room for **four 200k processes**
- Then:
  - If we have an **I/O wait time of 80%,** then we will achieve just under **60% CPU utilisation** ($1-0.8^4$)
  - If we **add another megabyte of memory**, it would allows us to run **another five processes**
  - We can achieve about **87% CPU utilisation** ($1-0.8^9$)
  - If we add **another megabyte of memory** (fourteen processes) we will find that the CPU utilisation will increase to **about 96%** ($1-0.8^{14}$)
- Multi-programming does enable to improve resource utilisation
- **Memory management should provide support for multi-programming**

Partitioning
Contiguous Approaches

- **Mono-programming:** one single partition for user processes
- **Multi-programming** with **fixed partitions**
  - Fixed **equal** sized partitions
  - Fixed **non-equal** sized partitions
- **Multi-programming** with **dynamic partitions**

Partitioning
Fixed Partitions

- Divide memory into **static**, **contiguous** and **equal sized partitions** that have a **fixed size** and **fixed location**
  - Any process can take up **any (**large enough) **partition**
  - Allocation of **fixed equal sized partitions** to processes is **trivial**
  - Very **little overhead** and **simple implementation**
  - The operating system keeps a track of which partitions are being **used** and which are **free**

- **Disadvantages** of static equal-sized partitions:
  - **Low memory utilisation** and **internal fragmentation**: partition may be unnecessarily large
  - **Overlays** must be used if a program does not fit into a partition (burden on programmer)

Partitioning
Fixed Partitions

- Divide memory into **static** and **non-equal sized partitions** that have a **fixed size** and **fixed location**
  - **Reduces internal fragmentation**
  - The **allocation** or processes to partitions must be **carefully considered**

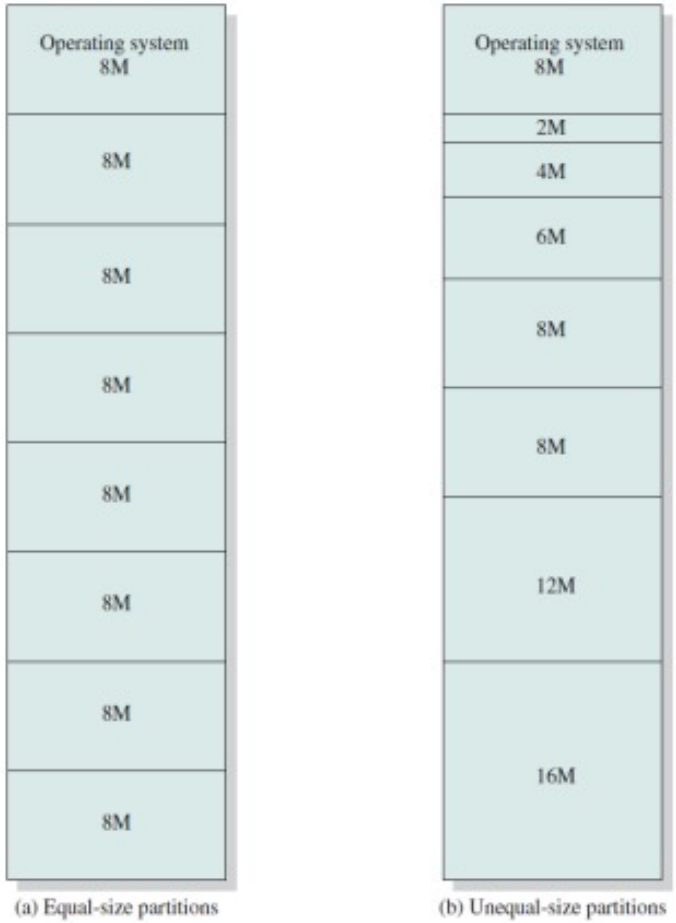


Figure: Fixed partitions (from Stallings)

Partitioning
Fixed Partitions (Allocation Methods)

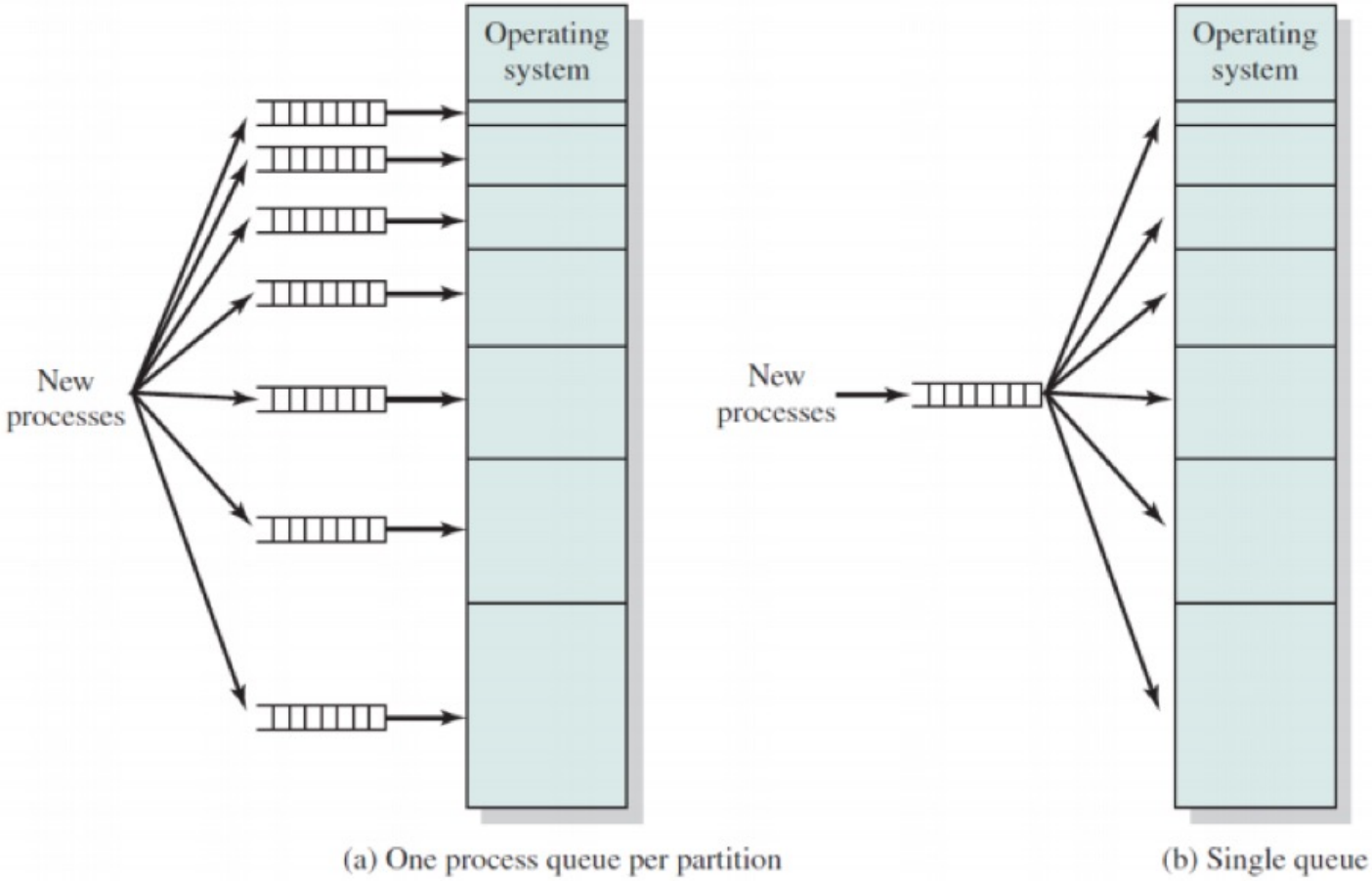


Figure: From Stallings

Partitioning
Fixed Partitions (Allocation Methods)

- **One private queue** per partition:
  - Assigns each process to the **smallest partition** that it would fit in
  - Reduces **internal fragmentation**
  - Can **reduce memory utilisation** (e.g., lots of small jobs result in unused large partitions) and result in **starvation**
- A **single shared queue** for all partitions can allocate small processes to **large partitions** but results **in increased internal fragmentation**
- **Cons of fixed partitioning:**
  - Limits the number of **active processes (overlays)**
  - Small job not **utilize partition space efficiently** (internal fragmentation)
  - Processes' memory requirement need to be **known beforehand**.

Memory management has been evolved

- Approches
  - **Contiguous Memory**
    - ❖ **Mono-programming** [easy but low resource utilisation, cannot be used for modern multiprogramming machine]

      No relocation, physical address
    - ❖ **Multiprogramming** with **fixed partition** [multi-programming but internal fragmentation]

      Relocation (register), logical address
    - ❖ **Multiprogramming** with **dynamic partition** [low internal fragmentation but high external fragmentation]

      Relocation (register), logical address
  - Non-Contiguous Memory

    Relocation (table), logical address
    - ❖ **Paging** (fixed-partitioning/code re-location) [Internal fragmentation reduce to last block, no external fragmentation; Require page table to maintain page relocation]
    - ❖ **Virtual Memory** (locality →not all pages loaded) [more processes → CPU utilisation, no external fragmentation, more memory available; Need involve replacement algorithm, page table management, thrashing, variable/fixed resident set….]

Recap
Take-Home Message

- **Mono-programming** and **absolute addressing**
- Why multi-programming: CPU utilisation modelling
- **Multi-programming**, fixed (non-)equal **partitions**, **CPU utilisation modelling**