Operating Systems and Concurrency

Lecture 12: Memory Management I

G52OSC

Geert De Maere and Isaac Triguero
{Geert.DeMaere,Isaac.Triguero}@nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

- Contact details:
  - Name: Isaac Triguero
  - Office: C82 (office hour 10:00-11:00 on Tuesdays - please confirm by e-mail beforehand).
  - E-mail: Isaac.Triguero@nottingham.ac.uk

- About me:
  - Graduated in 2009, Msci in Computer Science and Engineering
  - Completed my PhD in CS in 2014 (Data mining)
  - Member of ASAP
  - 4 years experience as SysAdmin.
  - Specific interest in Data Science

Remember

Subjects We Will Discuss

Subject	#Lectures	By
Introduction to operating systems/computer design	1-2	GDM
Processes, process scheduling, threading, ...	3-4	GDM
Concurrency (deadlocks)	4-5	GDM
Revision	1	GDM
Memory management, swapping, virtual memory, ...	5-6	IT
File Systems, file structures, management, ...	4-5	IT
Virtualisation and the Cloud	1-2	IT
Revision	1	IT

Table: Module Structure

Important: From now on, we don't have more lectures on **Thursdays at 4pm**!

1. **Introduction** to memory management

2. **Modelling** of multi-programming
3. Memory management based on **fixed partitioning**

Memory Management

Memory Hierarchies

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

- "**Higher memory**" is faster, more expensive and volatile, "**lower memory**" is slower, cheaper, and non-volatile
- The operating system provides a **memory abstraction**

- 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
- **Distribute memory** between processes and simulate an "**infinitely large**" memory space
- **Control access** when multiprogramming is applied
- **Transparently** move data from **memory** to **disk** and vice versa

Memory Management
History of Memory Management

- Memory management has **evolved**: memory management in **modern computers** is very different from **early computers**

- **History repeats** itself:
  - Many of the **early ideas underpin** more **modern memory management** approaches (e.g. relocation)
  - Modern **consumer electronics** often require **less complex memory management** approaches

Partitioning

Approaches: Contiguous vs. Non-Contiguous

Figure: Contiguous

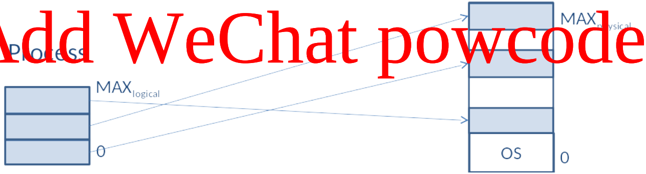


Figure: Non-contiguous

Models

Approaches: Contiguous vs. 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).

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)

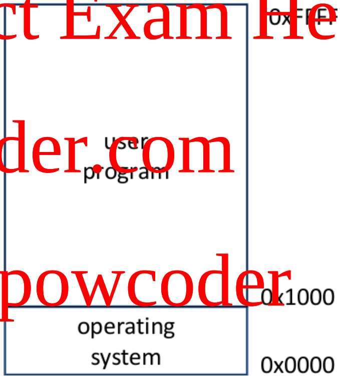


Figure: Mono-programming

Mono-Programming

No Memory Abstraction: Properties

- Every process is allocated **contiguous block of memory**, i.e. it contains no "holes" or "gaps" ($\Leftrightarrow$ **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**[1] enable the programmer to use more memory than available (burden on programmer)

[1] Tanenbaum - Section 3.3 (page 194)

Mono-Programming

No Memory Abstraction: Properties (Cont'ed)

- **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 are common in basic **embedded systems** and **modern consumer electronics**, e.g. washing machines, microwaves, car's ECUs, etc.

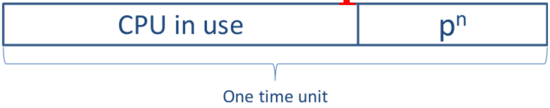
Mono-Programming
Simulating Multi-Programming

- Simulate multi-programming through **swapping**
  - **Swap process** out to the disk and load a new one (context switches would become **time consuming**)
  - Apply **threads** within the same process (limited to one process)
- Assumption that **multiprogramming** can **improve CPU utilisation**?
  - Intuitively, this is true
  - How do we model this?

Multi-Programming
A Probabilistic Model

- There are n **processes in memory**
- A process spends p percent of its time **waiting for I/O**
- **CPU Utilisation** is calculated as 1 minus the time that all processes are waiting for I/O: e.g., $p = 0.9$ then CPU utilisation = 1 - 0.9 $\Rightarrow$ 0.1 $(1 - p)$

- The probability that **all** n **processes are waiting for I/O** (i.e., the CPU is idle) is p^n, i.e. $p \times p \times p \ldots$
- 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**

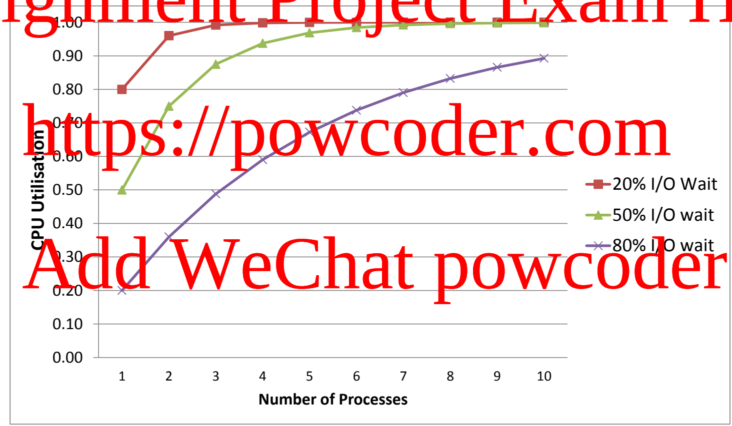
Multi-Programming

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

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**

  $\Rightarrow$ memory management should provide support for multi-programming

Multi-Programming

A Probabilistic Model

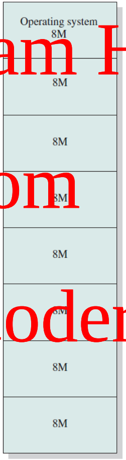
Caveats:

- This model assumes that **all processes are independent**, this is not true
- More complex models could be built using **queueing theory**, but we can still use this simplistic model to **make approximate predictions**

Partitioning

Fixed Partitions of equal size

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



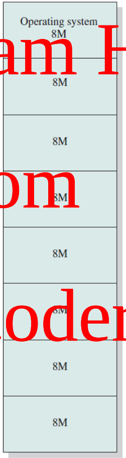
(a) Equal-size partitions

Figure: From Stallings

Partitioning

Fixed Partitions of equal size

- **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)



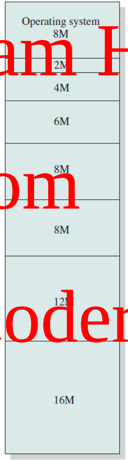
(a) Equal-size partitions

Figure: From Stallings

Partitioning

Fixed Partitions of non-equal size

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

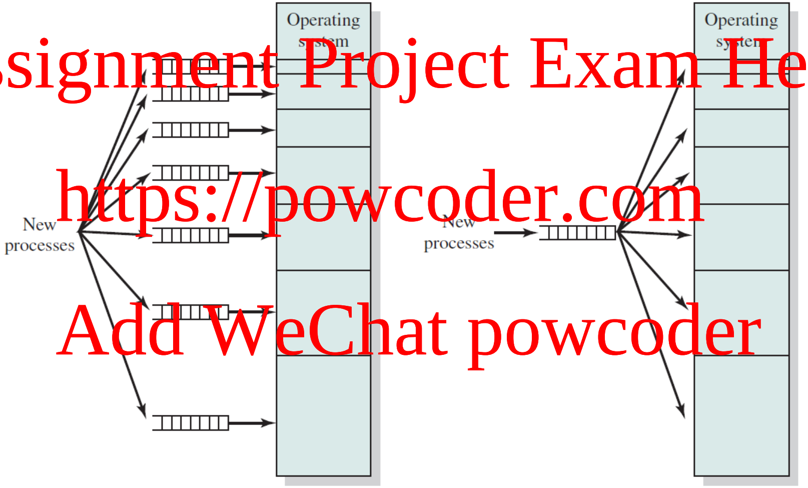


(b) Unequal-size partitions

Figure: From Stallings

Partitioning

Fixed Partitions (Allocation Methods)



(a) One process queue per partition (b) Single queue

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**

- **Mono-programming** and **absolute addressing (no memory abstraction)**

- Why multi-programming: CPU utilisation modelling.
- Memory management for **Multi-programming**: fixed (non-)equal **partitions**.

[2]Resources: Tanenbaum Section 3.1, 3.2. Stallings Section 7.1, 7.2