Memory Management

G53OPS

Geert De Maere

University Of Nottingham
United Kingdom

2014

- Processes
- **Memory management**
- File management
- Input/output
- Security and protection
- …

- **OS responsibilities** in memory management
- **Basic memory management** schemes (partitioning)
- **Virtual** and **physical addresses**, **relocation and protection**
- **Swapping**
- **Administering** memory allocation

Memory Management

History of Memory Management

- Memory management has **evolved** together with operating systems
- **History repeats** itself:
  - Modern **consumer electronics** often require **less complex memory management** approaches
  - Many of the **early ideas underpin** more **modern memory management** approaches

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
- Memory can be seen as one **linear array** of bytes/words
- The operating system provides a **memory abstraction**

Memory Management

OS Responsibilities

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

Memory Management Models

Approaches

- **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**

Partitioning

Mono-programming Without Memory Abstraction

- Allow **one single user process** in memory/to be executed at the any one time
- Part of the memory is reserved for the OS/**kernel**, the remaining memory is reserved for one **single process** (the “**MS-DOS** way”)
- “No” **multi-programming**
- The process usually has direct access to the **physical memory**
- A fixed region of memory is allocated to the **OS**

Partitioning

Mono-programming Without Memory Abstraction

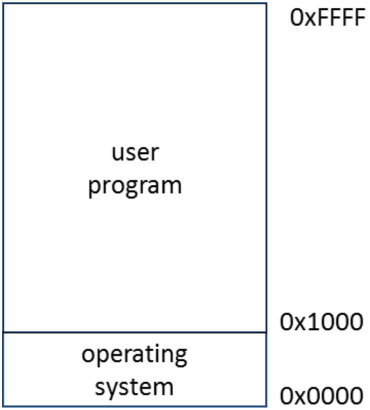


Figure: Mono-programming

Partitioning

Mono-programming Without Memory Abstraction: Properties

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

Partitioning

Mono-programming Without Memory Abstraction (Cont'ed)

- **Problems** with 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** in most cases
- Direct memory access and mono-programming is common in basic **embedded systems** and **modern consumer electronics**, e.g. washing machines, microwaves, car's ECUs, etc.

Partitioning

Multi-Programming in a Mono-Programming Environment

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

Partitioning

Modelling Multi-Programming: a Probabilistic Model

- A process spends p percent of its time **waiting for I/O**
- There are n **processes in memory**
- The probability that **all** n **processes are waiting for I/O** (i.e., the CPU is idle) is p^n
- The **CPU utilisation** is given by $1 - p^n$

Partitioning

Modelling Multi-Programming: an Example

- 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**

Partitioning

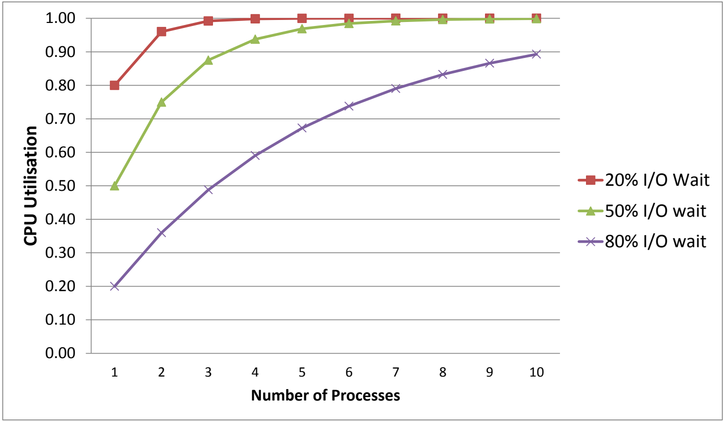
Modelling Multi-Programming: an Example

	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

Partitioning

Modelling Multi-Programming: an Example



Partitioning

Modelling Multi-Programming: an Example

- 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 mode to **make approximate predictions**

Partitioning

Modelling Multi-Programming: an Example

- 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%** $(1 - 0.8^{14})$

Partitioning

Modelling Multi-Programming: an Example

- 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**

Partitioning

Multi-Programming with 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**
  - Very **little overhead** and **simple implementation**
  - The operating system keeps a track of which partitions are being **used** and which are **free**
  - Allocation of **fixed equal sized partitions** to processes is **trivial**
- **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

Multi-Programming with Fixed Partitions (Cont'ed)

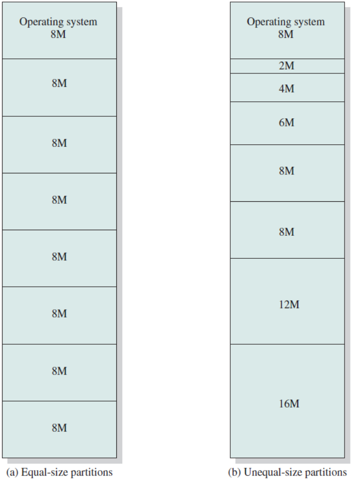


Figure: Fixed partitions (from Stallings)

Partitioning

Multi-Programming with Fixed Partitions (Cont'ed)

- **D**ivide 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**

Partitioning

Multi-Programming with Fixed Partitions (Cont'ed)

- How does one **allocate partitions** to jobs?
- How are **swap decisions** made
  - Priority based
  - State based
  - Space based

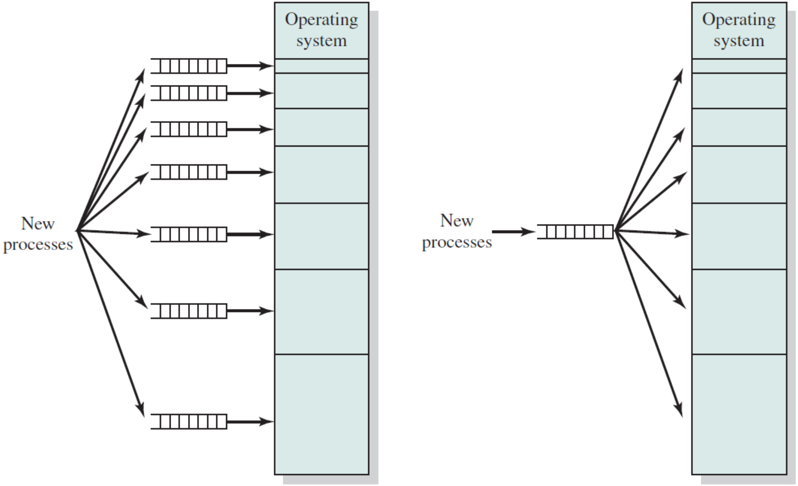
Partitioning

Multi-Programming with Fixed Partitions: Allocation Strategies

- 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**

Partitioning

Multi-Programming with Fixed Partitions: Allocation Strategies



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

Figure: From Stallings

Relocation and Protection
Context

- Memory addresses are calculated **relative to the start of the program**
- A job's memory **requirements may change** over time and an allocated partition may become too small
- Processes will **not always** occupy the **same partition**
    - It is not always known beforehand **which processes** will be **in memory** and which partitions they will occupy
    - Processes can be **swapped out**, and subsequently loaded into a **different partition**
    - **Memory requirements** of a process may **change overtime**, requiring a larger partition
- $\Rightarrow$ Code needs to be **relocatable**

Summary
Take Home Message

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