1 Memory

1.1 Models

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)

2 Partitioning

2.1 Overview

- 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

2.2 Mono-Programming

- 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)

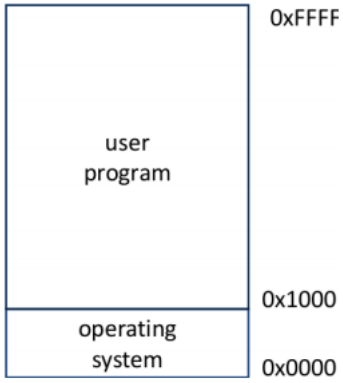

Figure: Mono-programming

- 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)

2.2.1 Shortcomings

- 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, cars ECUs, etc.

2.3 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)

To calculate *CPU utilisation*

- There are n processes in memory
- A process spends p percent of its time waiting for I/O
- The CPU utilisation is given by $1 - (p^n)$

Multi-programming does **enable to improve** resource utilisation `->` memory management should provide support for multi-programming.
Caveats:

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

2.4 Partitioning methods

2.4.1 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 a track of which partitions are being used and which are free

Disadvantages:

- **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)

2.4.2 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

2.4.3 Fixed partition 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*