Memory Management

OPS Lecture 8, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

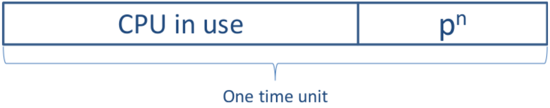
2015

Recall
Last Lecture

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

Recall
CPU utilisation

- A process **waits** p **percent/fractions** of its time **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:
  - Two processes **simultaneously wait for I/O** at any point in time is $p \times p$
  - n **processes** are simultaneously **waiting for I/O** is p^n
- The **CPU is used** when **not waiting for I/O**, i.e. $1 - p^n$ fractions of the time.



- It is useful to have multiple processes in memory to **maximise CPU utilisation**: mono-programming $\Rightarrow$ multi-programming
- Rather than allocating the full physical memory to one process, split it into **(non-)equal sized partitions** and **allocate a process to each partition**
- Fixed **equal sized partitions** result in **internal fragmentation**, **non-equal sized partitions** make **allocation** more difficult

Recall
Fixed Partitioning

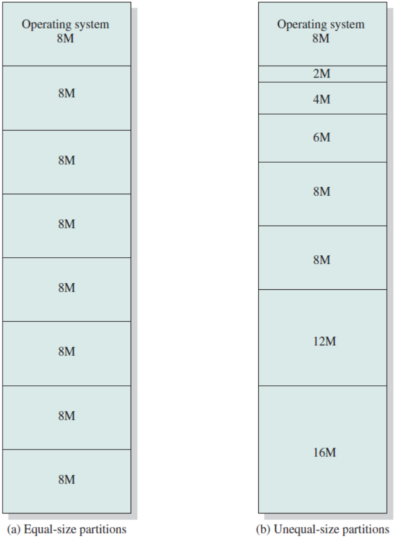


Figure: from Stallings

- Code **relocation** and **protection**
- **Dynamic partitioning**
- **Swapping**
- **Memory management**

Relocation and Protection

Principles

- **Relocation**: when a program is run, it **does not know in advance** which **partition/addresses** it will occupy
  - The program cannot simply generate **static addresses** (e.g. jump instructions) that are **absolute**
  - Addresses should be **relative to where the program has been loaded**
  - **Relocation must be solved** in an operating system that allows processes to run at **changing memory locations**
- **Protection**: once you can have two programs in memory at the same time, protection must be enforced

Relocation and Protection
Principles

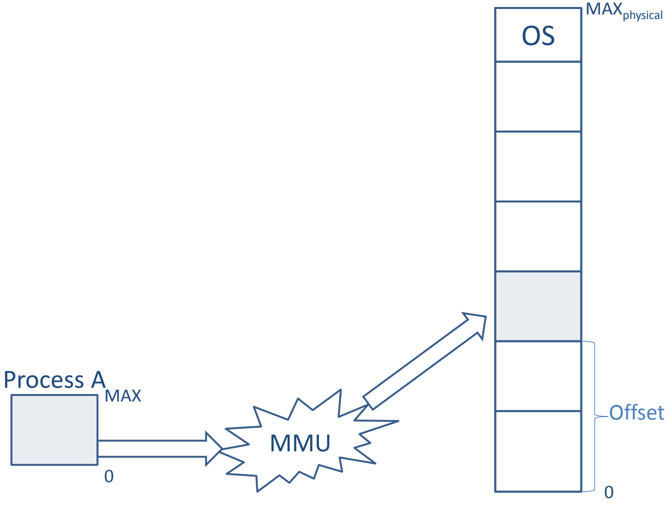


Figure: Address Relocation

Relocation and Protection
Address Types

- A **logical address** is a memory address **seen by the process**
  - It is **independent** of the current **physical memory** assignment
  - It is, e.g., **relative to the start of the program**
- A **physical address** refers to an **actual location** in **main memory**
- The **logical address space** must be mapped onto the machine's **physical address space**

Relocation and Protection
Approaches

1. **Static "relocation" at compile time**: a process has to be located at the same location every single time (impractical)
2. **Dynamic relocation** at **load time**
   - An **offset** is added to every logical address to **account for its physical location** in memory
   - **Slows down** the loading of a process, does not account for **swapping**
3. **Dynamic relocation** at **runtime**

Relocation and Protection

At Runtime: Base and Limit Registers

- Two special purpose registers are maintained in the CPU, containing a **base address** and **limit**
  - The **base register** stores the **start address** of the partition
  - The **limit register** holds the **size** of the partition

Relocation and Protection

Base and Limit Registers (Cont’ed)

- **At runtime**:
  - The **base register** is added to the **logical (relative) address** to generate the **physical address**
  - The resulting address is **compared** against the **limit register**
- This approach requires **hardware support** (was not always present in the early days!)

Relocation and Protection

Base and Limit Registers

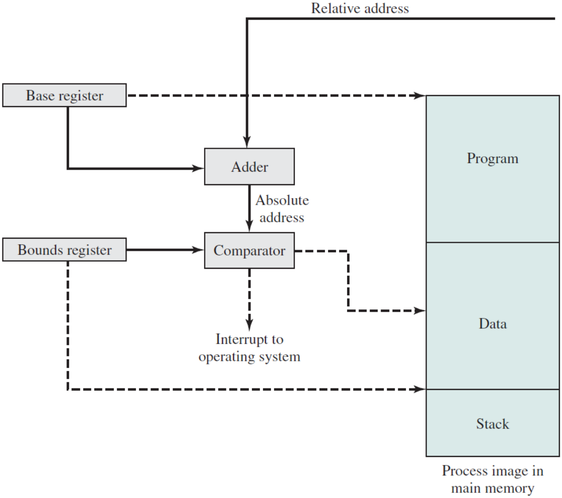


Figure: Address Relocation (Stallings)

Dynamic Partitioning
Context

- **Fixed partitioning** results in **internal fragmentation**:
  - An **exact match** between the requirements of the process and the available partitions **may not exist**
  - The partition may **not be used entirely**
- **Dynamic partitioning:**
  - A **variable number of partitions** of which the **size** and **starting address can change** over time
  - A process is allocated the **exact amount of contiguous memory** it requires, thereby preventing internal fragmentation

Dynamic Partitioning
Example

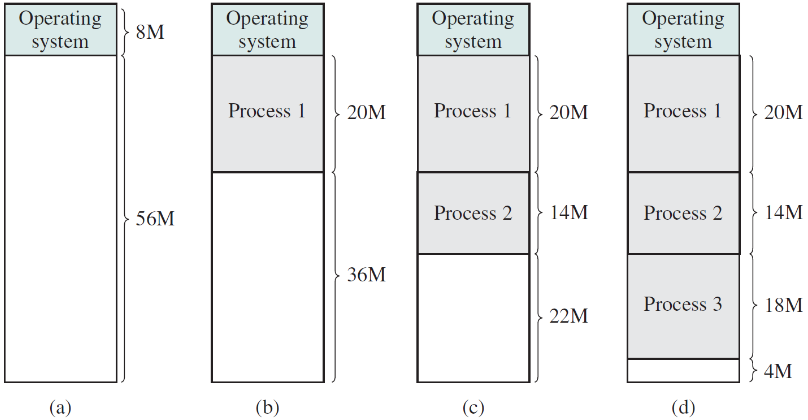


Figure: Dynamic partitioning (from Stallings)

Dynamic Partitioning

Swapping

- Swapping holds some of the **processes on the drive** and **shuttles processes** between the drive and main memory as necessary
- **Reasons** for swapping:
  - We have **more processes** than **partitions** (assuming fixed partitions)
  - The **total amount of memory that is required** for the processes **exceeds the available memory**
  - Some **processes** only **run occasionally**
  - A process's **memory requirements** have **changed**, e.g. increased

Dynamic Partitioning

Difficulties

- The exact **memory requirements** may **not be known** in advance (**heap** and **stack** grow dynamically)
  - $\Rightarrow$ Allocate the current requirements + “a bit extra”?

Figure: Memory organisation of a process

Dynamic Partitioning

Swapping: Example

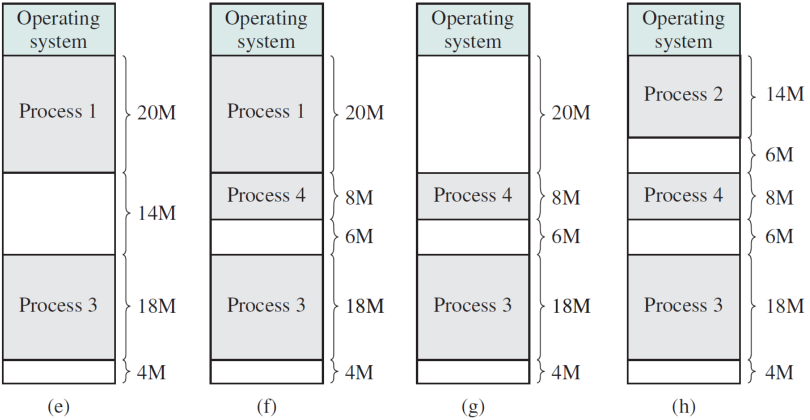


Figure: External fragmentation (from Stallings)

Dynamic Partitioning

Difficulties

- **External fragmentation**:
  - **Swapping** a process out of memory will create "**a hole**"
  - A new process may not use the entire "hole", leaving a small **unused block**
  - A new process may be **too large** for a given a "hole"
- The **overhead** of memory **compaction** to **recover holes** can be **prohibitive** and requires **dynamic relocation**

Dynamic Partitioning

Swapping: Questions

- **Memory management** becomes more complicated
- How to keep track of **available memory**
  - Linked lists
  - Bitmaps
- What strategies can I use to (quickly) **allocate** processes to available memory ("holes")

Dynamic Partitioning

Allocation Structures

- A more **sophisticated data structure** is required to deal with **a variable number** of **free and used partitions**
- A **linked list** is one such possible data structure
  - A linked list consists of a **number of entries** (“links”!)
  - Each link **contains data items**, e.g. **start of memory block**, **size**, free/allocated **flag**
  - Each link also contains a **pointer to the next** in the chain
- The **allocation** of processes to unused blocks becomes **non-trivial**

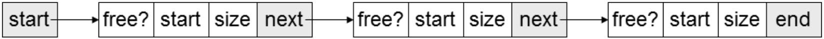


Figure: Memory management with linked lists

Dynamic Partitioning
Bitmaps

- There are other data structures that can be used in addition to linked lists, the simplest is a form of **bitmap**
- **Memory** is **split into blocks** of say 4K size
  - A bit map is set up so that each **bit is 0** if the **memory block is free** and **1** is the **block is used**, e.g.
    - 32Mb memory = 8192 * 4K blocks = 8192 bitmap entries
    - 8192 bits occupy 8192 / 8 = 1K bytes of storage (only!)
  - To find a hole of e.g. size 128K, then a group of **32 adjacent bits set to zero** must be found, typically a **long operation** (esp. with smaller blocks)

Dynamic Partitioning

Bitmaps (Cont'ed)

- A **trade-off exists** between the **size of the bitmap** and the **size of blocks** exists
  - The **size of bitmaps** can become prohibitive for small blocks and may **make searching** the bitmap **slower**
  - Larger blocks may increase **internal fragmentation**
- For this reason (& inflexibility) they are **rarely used**

Summary
Take Home Message

- **Contiguous** memory schemes: mono-programming, static and dynamic partitioning
- **Relocation** and protection $\Rightarrow$ principles underpin paging and virtual memory!
- Internal and external **fragmentation**
- **Memory management**

Next Lecture

Content

- **Virtual memory with paging** ☺

Fragmentation
Context

- We have seen examples of memory allocation algorithms with **variable partitions** using **linked list** representation schemes, all of them suffer from **fragmentation problems**
- As memory is allocated it is **split into smaller blocks** so that only the required amount is used
  - But when it is **freed** it stays the same size
  - Over time, the **blocks get smaller** and smaller
- As processes come and go the **free space** that is available is **split across various blocks**

Dynamic Partitioning

Allocation Methods: First Fit

- The linked list is **initialised** to a **single link** of the entire memory size, flagged as "free"
- **First fit:** whenever a block of memory is requested the **linked list is scanned** in order until a link is found which represents **free space of sufficient size**
  - If requested space is **exactly the size** of the free space, all the space is allocated (i.e., no **internal** fragmentation)
  - Else, the free link is **split into two**:
    - The first entry is set to the **size requested** and marked "**used**"
    - The second entry is set to **remaining size** and marked "**free**"
- When a block is **freed** the link is marked "free"

Dynamic Partitioning

Allocation Methods: Next Fit

- The **first fit algorithm** starts scanning **at the start** of the linked list whenever a block is requested
- As a minor variation, the **next fit algorithm** maintains a record of where it got to, in scanning through the list, each time an allocation is made
    - The next time a block is requested the algorithms **restarts its scan** from **where ever it left off last time**
    - The idea is to give an **even chance to all of memory to getting allocated**, rather than concentrating at the start
- However, simulations have shown that next fit actually gives **worse performance** than first fit!

Dynamic Partitioning

Allocation Methods: Best Fit

- **First fit** just looks for the **first available hole**
  - It doesn't take into account that there may be **a hole later** in the list that **exactly(-ish)** fits the requested size
  - First fit **may break up a big hole** when the right size hole exists later on
- The **best fit algorithm** always **searches the entire linked** list to find **the smallest hole big** enough to satisfy the memory request
  - However, it is **slower** than first fit because of searching
  - Surprisingly, it also results in **more wasted memory** because it tends to fill up memory with tiny, useless holes

Dynamic Partitioning

Allocation Methods: Worst Fit

- **Tiny holes** are created when **best fit** breaks a hole of nearly the exact size into the required size and whatever is left over
- To get around the **problem of tiny holes**
  - How about always **taking the largest available hole** and breaking that up
  - The idea being that the **left over part will still be a large** and therefore **potentially useful** size
  - This is the **worst fit algorithm**
- Unfortunately, simulations have also shown that worst fit is **not very good either**!

Dynamic Partitioning

Allocation Methods: Quick Fit and Others

- As yet another variation, **multiple lists of different (commonly used) size blocks** can be maintained
  - For example a separate list for each of 4K, 8K, 12K, 16K, etc., holes
  - **Odd sizes** can either go into the **nearest size** of into a **special separate list**
- This scheme is called **quick fit**, because it is **much faster** to find the required size hole, however it still has problem of creating **many tiny holes**
- Yet more sophisticated schemes can be used with **knowledge** of the **likely sizes of future requests**

Fragmentation

Coalescing

- **Coalescing** (joining together) takes place when two **adjacent entries** in the linked list **become free**
    - There may be three adjacent free entries if an in-use block that is in-between two free blocks is freed
- When a **block is freed**
    - Both **neighbours** are examined
    - If either (or both) are **also free**
    - Then the two (or three) **entries are combined** into one
        - The **sizes are added** up to give the total size
        - The earlier block in the linked list gives the **start point**
        - The **separate links are deleted** and a single link inserted

Fragmentation
Coalescing

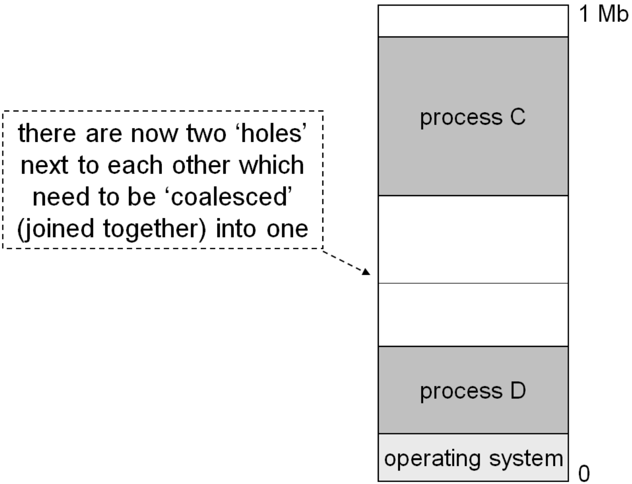


Figure: Coalescing

Fragmentation
Compacting

- Even with coalescing happening automatically the **free blocks** may still **be split up** from each other
    - Joining together all the free space available at any time so that all allocated memory is together (at the start) and all free memory is together (at the end) is **called compacting memory**
- Compacting is more difficult to implement than coalescing as **processes have to be moved**
    - Each process is **swapped out** & **free space coalesced**
    - Process swapped back in at lowest available location
- This is **time consuming** and so **done infrequently**

Fragmentation

Compacting

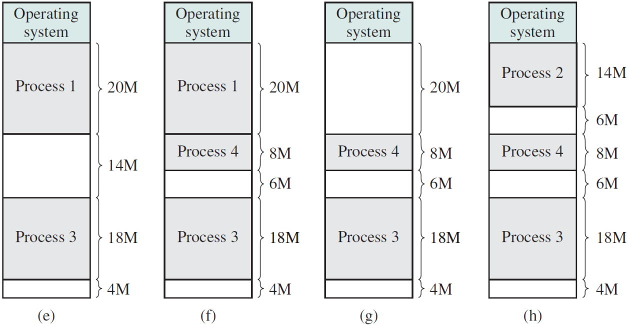


Figure: Compacting (from Stallings)

Dynamic Memory Allocation

Compacting

- If the correct size (only) of memory is allocated
  - What happens if the **process needs more memory?**
- The memory block can be expanded if there is **adjacent free space**
  - But if there is another process next to it
    - Either: one of the **processes** will have to be **moved**
    - Or: the process “in the way” will have to be **swapped out**
  - An amount of **extra memory** could always be allocated in case the **process grows a little**
    - But how much extra?
- We need a more **flexible scheme**