Memory Management

OPS Lecture 9, G53OPS/G52OSC

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

University Of Nottingham
United Kingdom

2015

Recall
Last Lecture

- **Dynamic relocation** and protection $\Rightarrow$ **base** (offset) and **limit** registers (logical/physical address)
- **Dynamic partitioning** $\Rightarrow$ **internal** and **external fragmentation**
- **Memory management** using **linked lists** and bitmaps

Dynamic Partitioning
Example

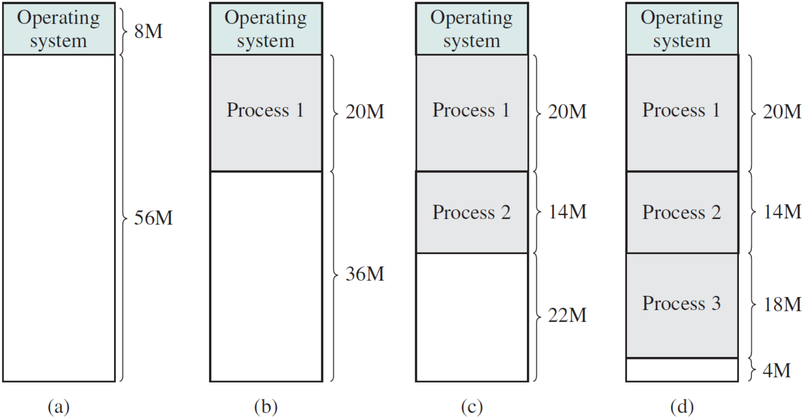

Figure: Dynamic partitioning, a problem occurs when process 2 grows (from Stallings)

Dynamic Partitioning

Swapping: Example

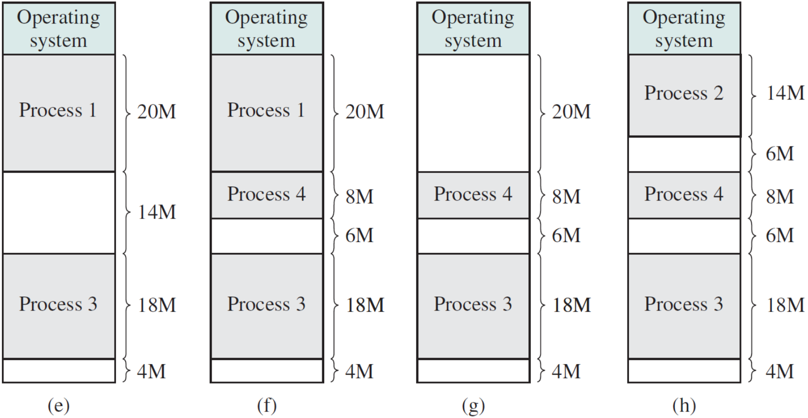

Figure: Dynamic partitioning, swapping results in external fragmentation (from Stallings)

Dynamic Partitioning

Memory Management

- How to keep track of **available memory**
  - Bitmaps
  - **Linked lists**
- The operating system is responsible for:
  - Applying strategies to (quickly) **allocate processes** to available memory ("holes")
  - Managing **free space**

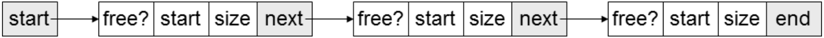

Figure: Memory management with linked lists

Dynamic Partitioning

Allocating Available Memory: First Fit

- The linked list is **initialised** to a **single link** of the entire memory size, flagged as “free”
- **First fit** starts scanning **from the start** of the linked list 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**”

Dynamic Partitioning

Allocating Available Memory: Next Fit

- As a minor variation of first fit, the **next fit algorithm** maintains a record of where it got to:
  - The next time a block is requested the algorithms **restarts its scan** from **where it stopped 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

Allocating Available Memory: 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

Allocating Available Memory: 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

Allocating Available Memory: Summary

- **First fit**: allocate **first block** that is **large enough**
- **Next fit**: allocate **next block** that is large enough, i.e. **starting from the current location**
- **Best fit**: choose block that **matches** required size **closest** - O(N) complexity
- **Worst fit**: choose the **largest possible block** - O(N) complexity

Dynamic Partitioning

Allocating Available Memory: 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**

Dynamic Partitioning

Allocating Available Memory: 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**
- **Finding neighbours** for coalescing becomes more difficult/time consuming

Dynamic Partitioning

Managing Available Memory: 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
- Both **neighbours** are examined when a **block is freed**
  - If either (or both) are **also free**
  - Then the two (or three) **entries are combined** into one larger block by adding up the sizes
    - The earlier block in the linked list gives the **start point**
    - The **separate links are deleted** and a single link inserted

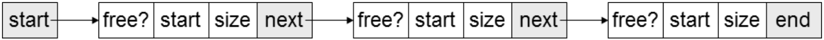

Figure: Memory management with linked lists

Dynamic Partitioning

Managing Available Memory: Compacting

- Even with coalescing happening automatically, **free blocks** may still **distributed across memory**
    - $\Rightarrow$ **Compacting** can be used to join free and used memory (but is **time consuming**)
- **Compacting is more difficult and time consuming** to implement than coalescing (processes have to be moved)
    - Each process is **swapped out** & **free space coalesced**
    - Process swapped back in at lowest available location

Contiguous Allocation Schemes

Overview and Shortcomings

- Different contiguous memory allocation schemes have different advantages/disadvantages
  - **Mono-programming** is easy but does result in **low resource utilisation**
  - **Fixed partitioning** facilitates **multi-programming** but results in **internal fragmentation**
  - **Dynamic partitioning** facilitates **multi-programming**, reduces **internal fragmentation**, but results in **external fragmentation** (allocation methods, coalescing, and compacting help)
- Can we design a memory management scheme that **resolves the shortcomings** of contiguous memory schemes

Paging
Principles

- **Paging** uses the principles of **fixed partitioning** and **code re-location** to devise a new **non-contiguous management scheme**, however:
  - Memory is split into much **smaller blocks** and **one or more blocks** are allocated to a process
    - e.g., a 11kb process would take up 3 blocks of 4 kb
  - These blocks **do not have to be contiguous in main memory**, but the process still **perceives them to be contiguous**
- Benefits compared to contiguous schemes include:
  - **Internal fragmentation** is reduced to the **last "block"** only
  - There is **no external fragmentation**, since physical blocks are **stacked directly onto each other** in main memory

Paging
Principles (Cont'ed)

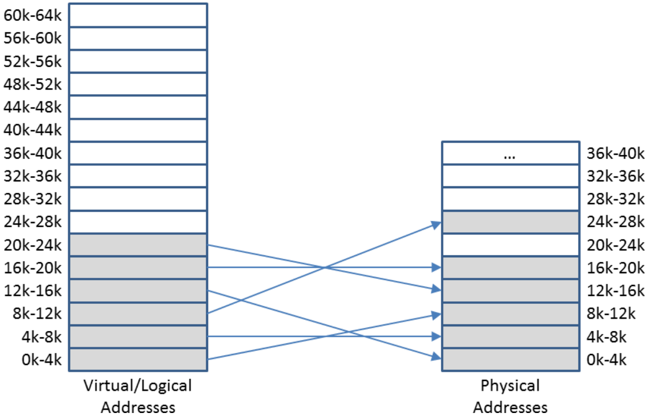

Figure: Paging in main memory with multiple processes

Paging

Principles (Cont'ed)

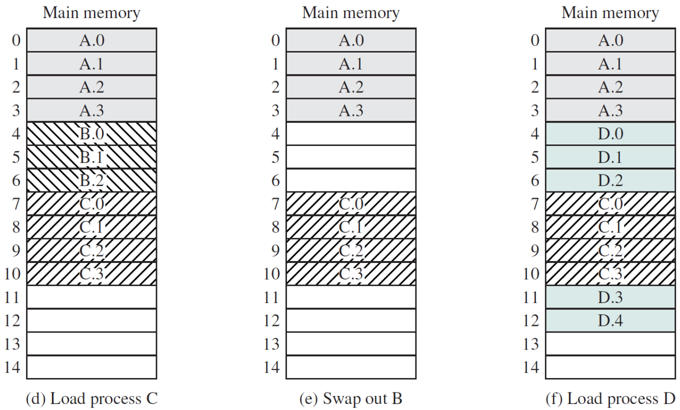

Figure: Concept of Paging (Stallings)

Paging

Principles: Definitions

- A **page** is a small block of **contiguous memory** in the **logical address space**, i.e. as seen by the process
- A **frame** is a **small contiguous block** in **physical memory**
- Pages and frames (usually) have the **same size**:
  - The size is usually a power of 2
  - Sizes range between 512 bytes and 1Gb

Paging
Relocation

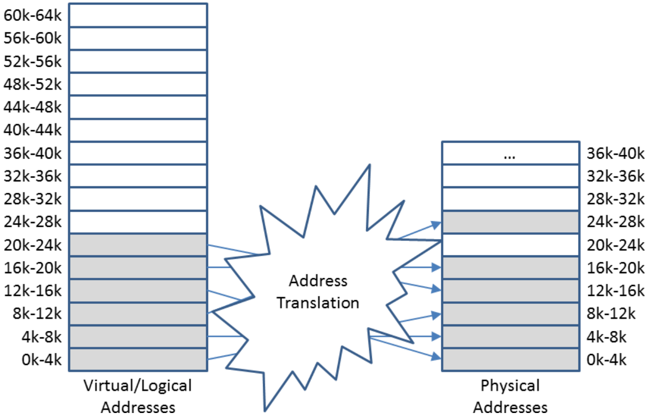

Figure: Address Translation

Paging
Relocation

- **Logical address** (page number, offset within page) needs to be **translated** into a **physical address** (frame number, offset within frame)
- **Multiple “base registers”** will be required:
  - Each logical page needs **a separate “base register”** that specifies the start of the associated frame
  - I.e, a **set of base registers** has to be maintained for each process
- The base registers are stored in the **page table**

Paging

Relocation: Address Translation

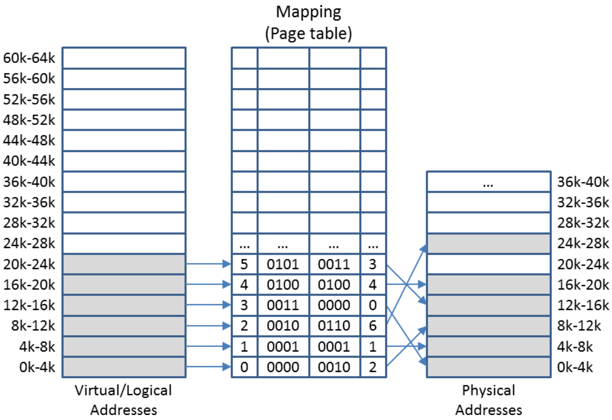

Figure: Address Translation

Paging

Relocation: Page Tables

- The page table can be seen as **a function**, that **maps the page number** of the logical address **onto the frame number** of the physical address
  - frameNumber=f(pageNumber)
- The **page number** is used as **index to the page table** that lists the **number of the associated frame**, i.e. it contains the location of the frame in memory
- Every process has its **own page table** containing its own “base registers”
- The **operating system** maintains a **list of free frames**

Paging

Address Translation: Implementation

- A **logical (physical) address** is relative to the start of the **program (memory)** and consists of two parts:
  - The **left most** *n* **bits** that represent the **page (frame) number**
    - e.g. 4 bits for the page number allowing 16 (2^4) pages (frames)
  - The **right most** *m* **bits** that represent the **offset within the page (frame)**
    - e.g. 12 bits for the offset, allowing up to 4096 (2^{12}) bytes per page (frame)
- The **offset** within the page and frame **remains the same** (they are the same size)
- The page number to frame number mapping is held in the **page table**

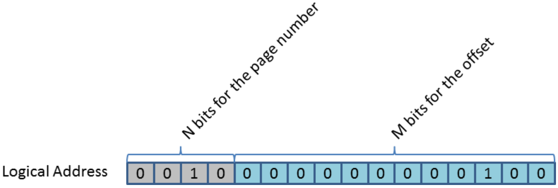

Figure: Logical Address

Paging

Address Translation: Implementation

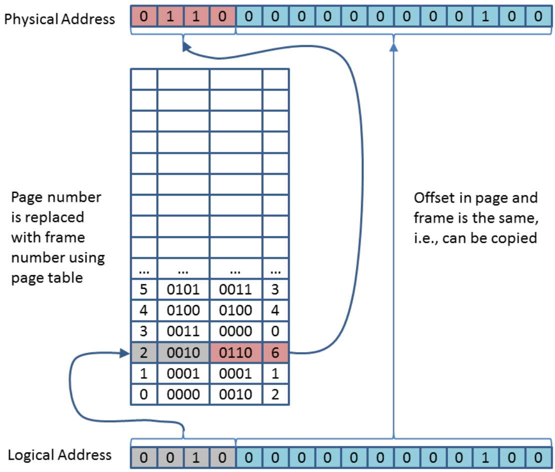

Figure: Address Translation

Paging

Relocation: Address Translation

- Steps in **address translation**:
  1. Extract the page number from logical address
  2. Use page number as an index to retrieve the frame number in the page table
  3. Add the “logical offset within the page” to the start of the physical frame
- **Hardware implementation** of address translation
  1. The CPU’s memory management unit intercepts logical addresses
  2. Address translation using page table as above
  3. The resulting physical address is put on the memory bus

Recap

Take-Home Message

- Memory **allocation**, **coalescing** and **compacting** in dynamic partitioning
- **Paging**, **page tables**, and **address translation**

Paging

Address Translation: Examples

- Let us assume 4KB pages (2^{12}), leaving 2^4 = 16 pages
- What is the physical address of 0, 8192, 20500 using the page table below

	Pages	Frames	
0	0000	0010	2
1	0001	0001	1
2	0010	0110	6
3	0011	0000	0
4	0100	0100	4
5	0101	0011	3
...	...	...	...
...	...	...	...
...	...	...	...
9	1001	0101	5
...	...	...	...
11	1011	0111	7
...	...	...	...

Table: Page Table

Paging

Address Translation: Examples

- Virtual address 0 falls in page with index 0 which is mapped onto frame with index 2, starting at 2×4096, i.e., the physical address is 8192
- Virtual address 8192 falls in page with index 2 which is mapped onto frame with index 6, starting at 6×4096, i.e., the physical address is 24576
- Virtual address 20500 falls in page with index 5 ($20500 > 5 \times 4096$) which is mapped onto page with index 3, starting at 3×4096, i.e., the physical address is $12288 + 20 = 12308$