Operating Systems and Concurrency

Memory Management 2

COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recall

Last Lecture

- **Mono-programming** with **absolute addressing**
- Modelling **CPU utilisation**
- **Multi-programming** with fixed (non-)equal **partitions** to improve **CPU utilisation**
- **Internal fragmentation**

Overview

Goals for Today

- Code **relocation** and **protection**
- **Dynamic partitioning** & **segmentation**
- **Managing free/occupied** memory

Relocation and Protection

Example

```c
#include <stdio.h>
#include <unistd.h>
int iVar = 0;
int main() {
  while(iVar++ < 10) {
    printf("Addr:%p; Val:%d\n", &iVar, iVar);
    sleep(1);
  }
  return 0;
}
```

- If **running the code** twice (simultaneously):
  - Will the **same or different addresses** be displayed for *iVar*?
  - Will the value of *iVar* in the first run **influence** the value if *iVar* in the second run?
- Note that this may not work on “newer” OSs that use **Address Space Layout Randomization**

Relocation and Protection

Principles

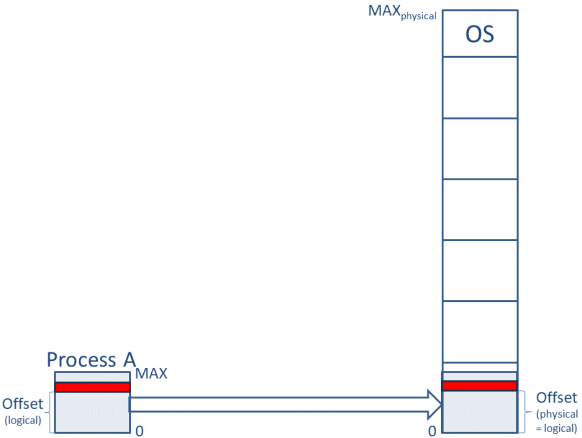

Figure: Address relocation

Relocation and Protection

Principles

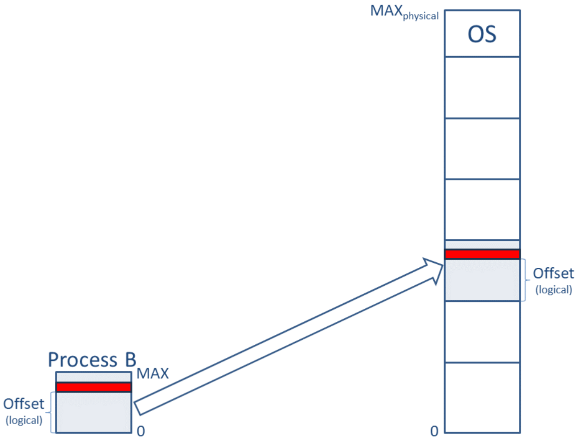

Figure: Address relocation

Relocation and Protection

Principles

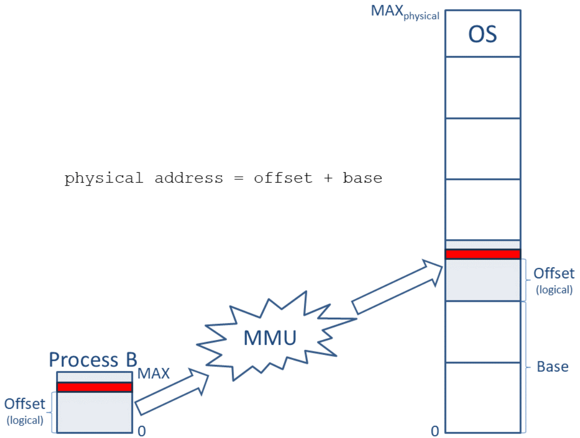

Figure: Address relocation

Relocation and Protection

Principles

- **Relocation:** One **does not know at compile time** which **partition/addresses** a process will occupy
  - **Compiler** assumes the process that it will run at 0
  - **Mapping** takes place once it is known where in physical memory the process resides
  - **Relocation** must be used in any operating system that allows processes to run at **changing locations** in **physical memory**
- **Protection**: once you can have multiple processes in memory at the same time, **protection** must be enforced

Relocation and Protection

Principles

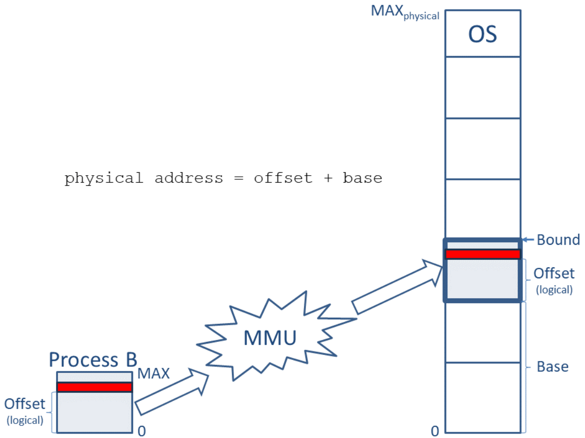

Figure: Address relocation

Relocation and Protection

Address Types

- A **logical address** is a memory address **seen by the process**
  - **Relative to the start of the program**
  - Assigned by the **compiler**
  - **Independent** of the current location in **physical memory**
- The **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** with **hardware support**

Relocation and Protection

At Runtime: Base and Bound Registers

- Two special purpose registers are maintained in the CPU (the **MMU**), containing a **base address** and **bound**
  - The **base register** stores the **start address** of the partition
  - The **bound register** holds the **size** of the partition
- **At runtime**:
  - The **base register** is added to the **logical (relative) address** to generate the **physical address**
  - The resulting address is **compared** against the **bound register**
- **Hardware support** was not always present in the early days!

Relocation and Protection

Base and Bound Registers

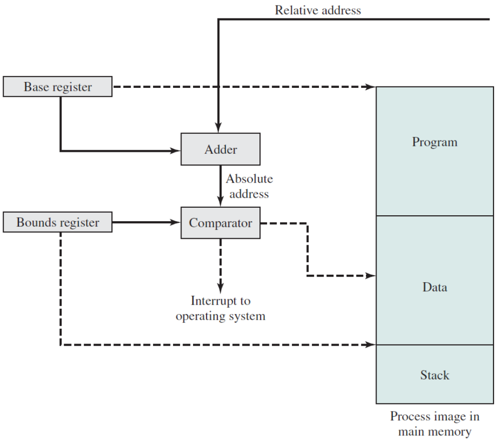

Figure: Address Relocation (Stallings)

Dynamic Partitioning

Context

- **Fixed partitioning** results in **internal fragmentation**:
  - An **exact match** for the process and may not always be **available**
  - The partition may **not be used completely**
- **Dynamic partitioning:**
  - A **variable number of partitions**
  - A process is allocated the **exact amount** of **contiguous** memory
  - **Reduces internal fragmentation**

Dynamic Partitioning

Example

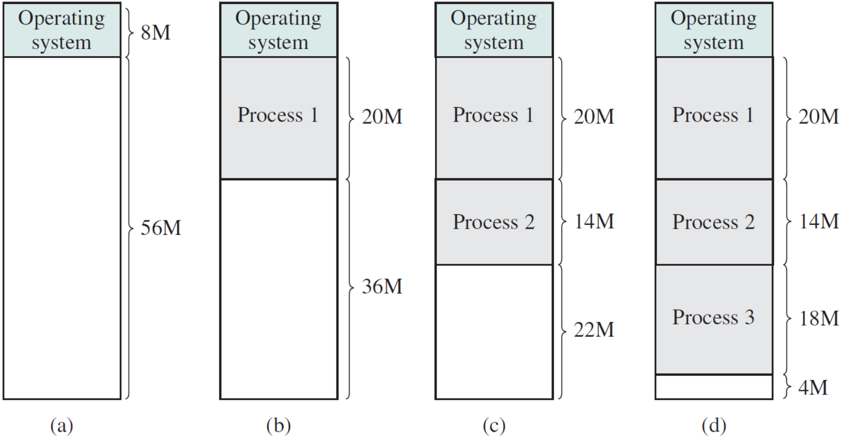

Figure: Dynamic partitioning (from Stallings)

Dynamic Partitioning

Swapping

- Swapping **moves processes** between the drive and main memory
- **Reasons** for swapping:
  - Some **processes** only **run occasionally**
  - The **total amount of memory** required **exceeds available memory**
  - **Memory requirements change** / cannot be **known in advance** (stack / heap)
  - We have **more processes** than **partitions** (assuming fixed partitions)
- Processes can be reloaded into a **different memory location** (base register changes)

Dynamic Partitioning

Swapping: Example

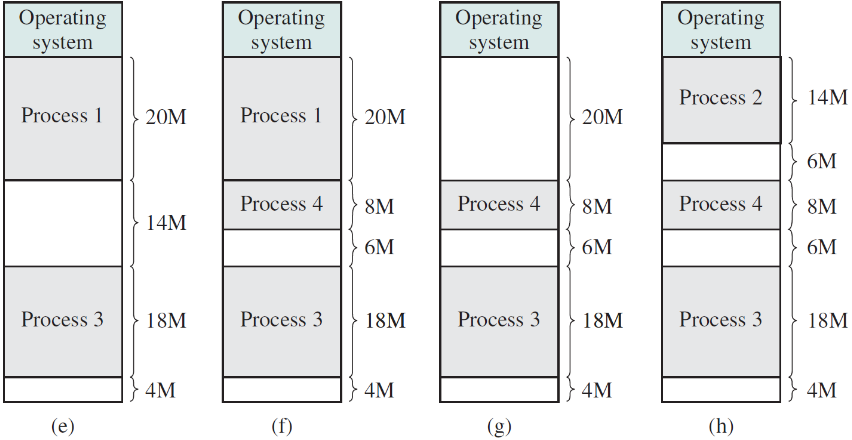

Figure: External fragmentation (from Stallings)

Dynamic Partitioning

Shortcomings

- **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

Shortcomings

Figure: Memory layout (logical address space)

Dynamic Partitioning

Shortcomings

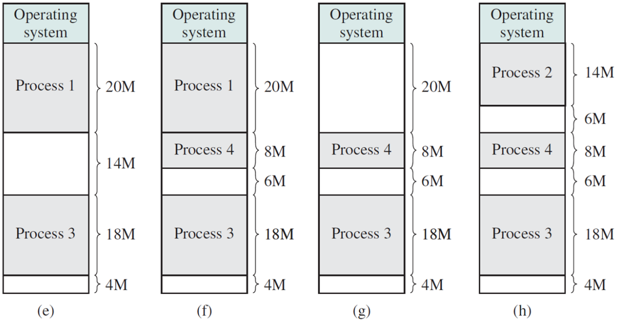

Figure: Fragmentation

Dynamic Partitioning

Shortcomings

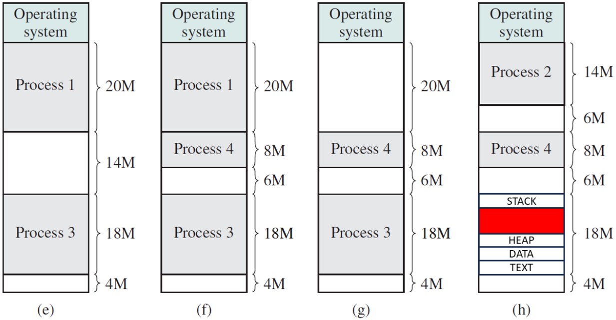

Figure: Fragmentation

Segmentation

Fine Grained Dynamic Partitioning

- **Dynamic partitioning** loads the **entire logical address space** into (**contiguous**) physical memory
  - Uses a **single base and bounds pair** per process
  - Results in **external fragmentation**
  - **Unused space** between stack and heap **takes up physical memory**
- **Segmentation** loads only the **relevant sections** into memory
  - Splits the logical address space into **separate contiguous segments** (code, data, heap, stack)
  - Each segment is **loaded separately** in (contiguous) memory
  - Each segment has a different **base** and **bound** pair
  - The base and bound pair are stored in a the **segmentation table**
  - **Part of the logical address** is used as an **index** into the segmentation table

Dynamic Partitioning

Shortcomings

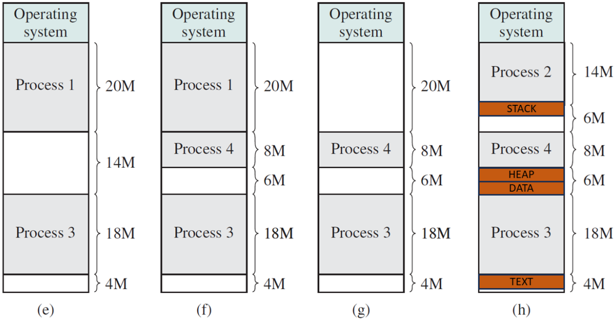

Figure: Segmentation

Segmentation

Segmentation Table

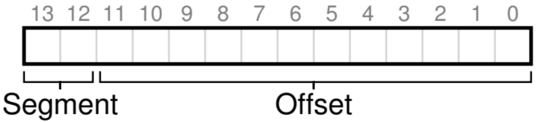

Figure: Logical address for segmentation (offset = position relative to start of segment)

Segment	**index**	**Base**	**Bound**	**RWX**
Code	00	...	...	...
Data	01	...	...	...
Heap	10	...	...	...
Stack	11	...	...	...

Segmentation

Segmentation Table & Challenges

- Segments can:
  - Have **protection bits** associated with them (RWX)
  - Be **shared** between processes (e.g. code segments)
- **MMU** must use the correct base / boud
- OS remains responsible for finding a sufficiently large **contiguous range of physical memory** for each segment (segments can be **fine-grained**)

Managing Free Space

Approaches

- How to keep track of **available memory**
  - Bitmaps
  - Linked lists
- What strategies can we use to (quickly) **allocate** processes to available memory (“holes”)?

Managing Free Space

Allocation Structures: Bitmaps

- The simplest data structure that can be used is a form of **bitmap**
- **Memory** is **split into blocks** of say 4KB 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 $\Rightarrow$ 32*2^{20} / 4KB blocks $\Rightarrow 32 * 2^8$ bitmap entries (8192)
    - 8192 bits occupy 8192 / 8 = 1KB of storage (only!)
  - The size of this bitmap will depend on the **size of the memory** and **the size of the allocation unit**.

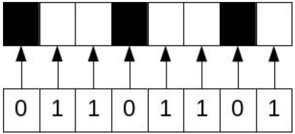

Figure: Memory management with bitmaps

Managing Free Space

Allocation Structures: Bitmaps (Cont'ed)

- To find a hole of e.g. size 128KB, then a group of **32 adjacent bits set to zero** must be found, typically a **long operation** (esp. with smaller blocks)
- 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**
- **Bitmaps** are **rarely used** for this reason

Managing Free Space

Allocation Structures: Linked List

- 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**

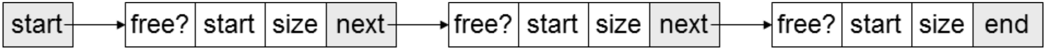

Figure: Memory management with linked lists

Managing Free Space

Allocation Structures

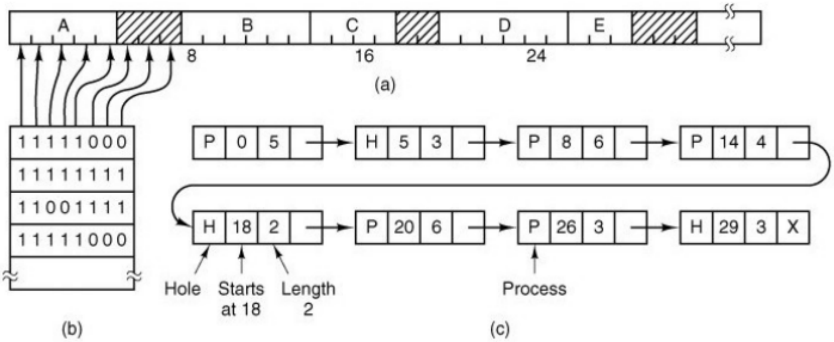

Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list. (from Tanenbaum)

Overview

Goals for Today

- Code **relocation** and **protection**
- **Dynamic partitioning** & **segmentation**
- **Managing free/occupied** memory

Test your Understanding

Tracking Free Memory

Exercise

Compare the **memory needed** to keep track of **free memory** using **bitmap** vs. **linked list**. The size of main memory is **8GB** and the block size is **1MB**. You can assume that exactly **half of the memory is in use**, and that it contains an **alternating sequence** of occupied and free blocks. The linked list only keeps track of **free blocks**, and each node requires a 32-bit memory address, a 16-bit length, and a 32-bit next-node field.

How many bytes of storage are required for each method?