Operating Systems and Concurrency

Lecture 13: Memory Management II

G52OSC

Geert De Maere and Isaac Triguero
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Recall

Last Lecture

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

- 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

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

Relocation and Protection

Recap

```
1  #include <stdio.h>
2
3  int iVar = 0;
4  void main() {
5      int i = 0;
6      while(i < 10) {
7          iVar++;
8          sleep(2);
9          printf("Address:%u; Value:%d\n",&iVar, iVar);
10         i++;
11     }
12 }
```


- If running the code twice (simultaneously):
  - Will the **same or different addresses** be displayed for *x* and will the value for *x* in the first run influence the value for *x* in the second run?
- Note that this may not work in many “new” OSs which use Address Space Layout Randomization[1] for security reasons – ASLR.

[1] Tanembaum Section 9.7 - Page 647

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

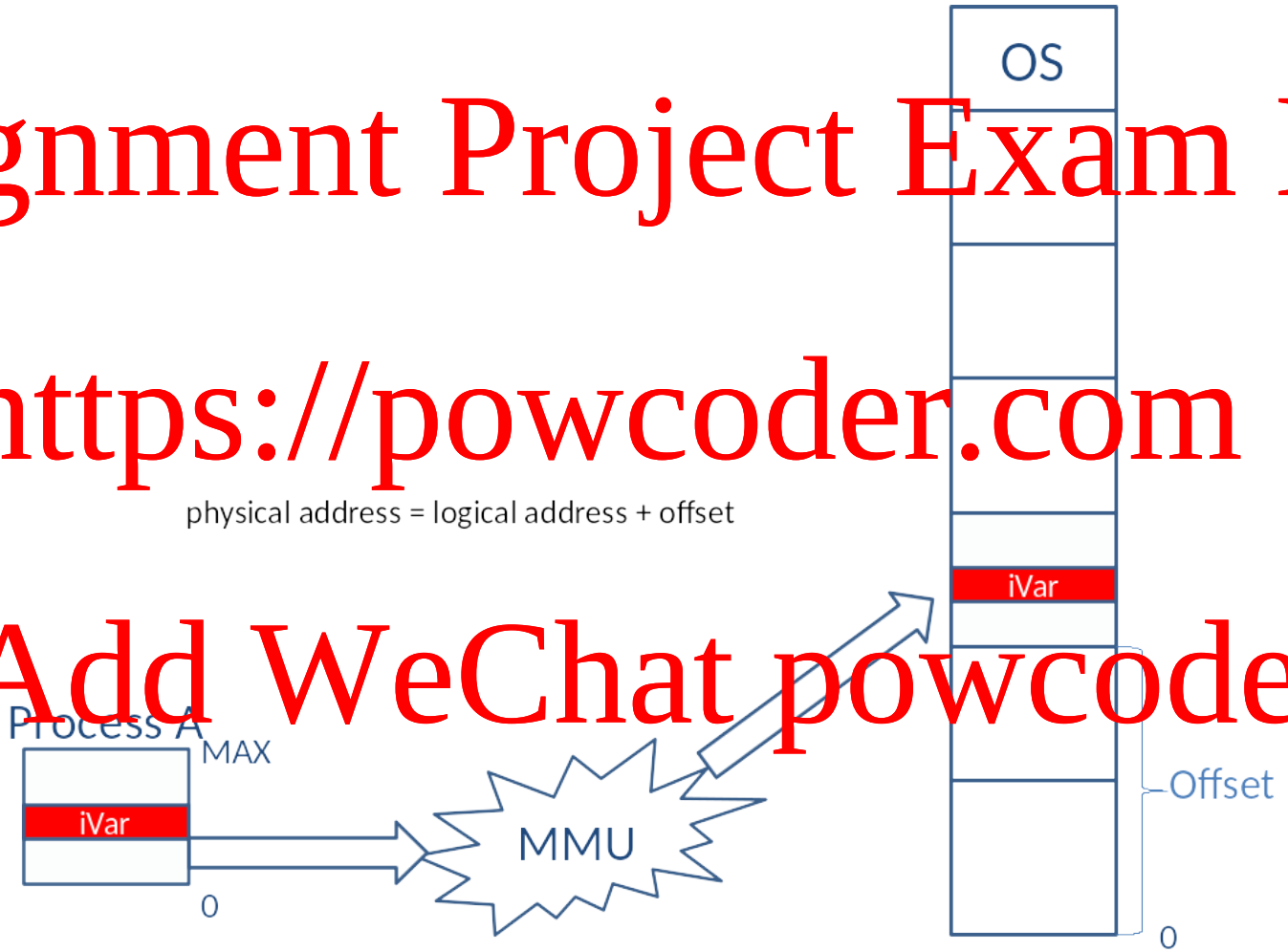

Figure: Address Relocation

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 (the **MMU**), 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
- **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

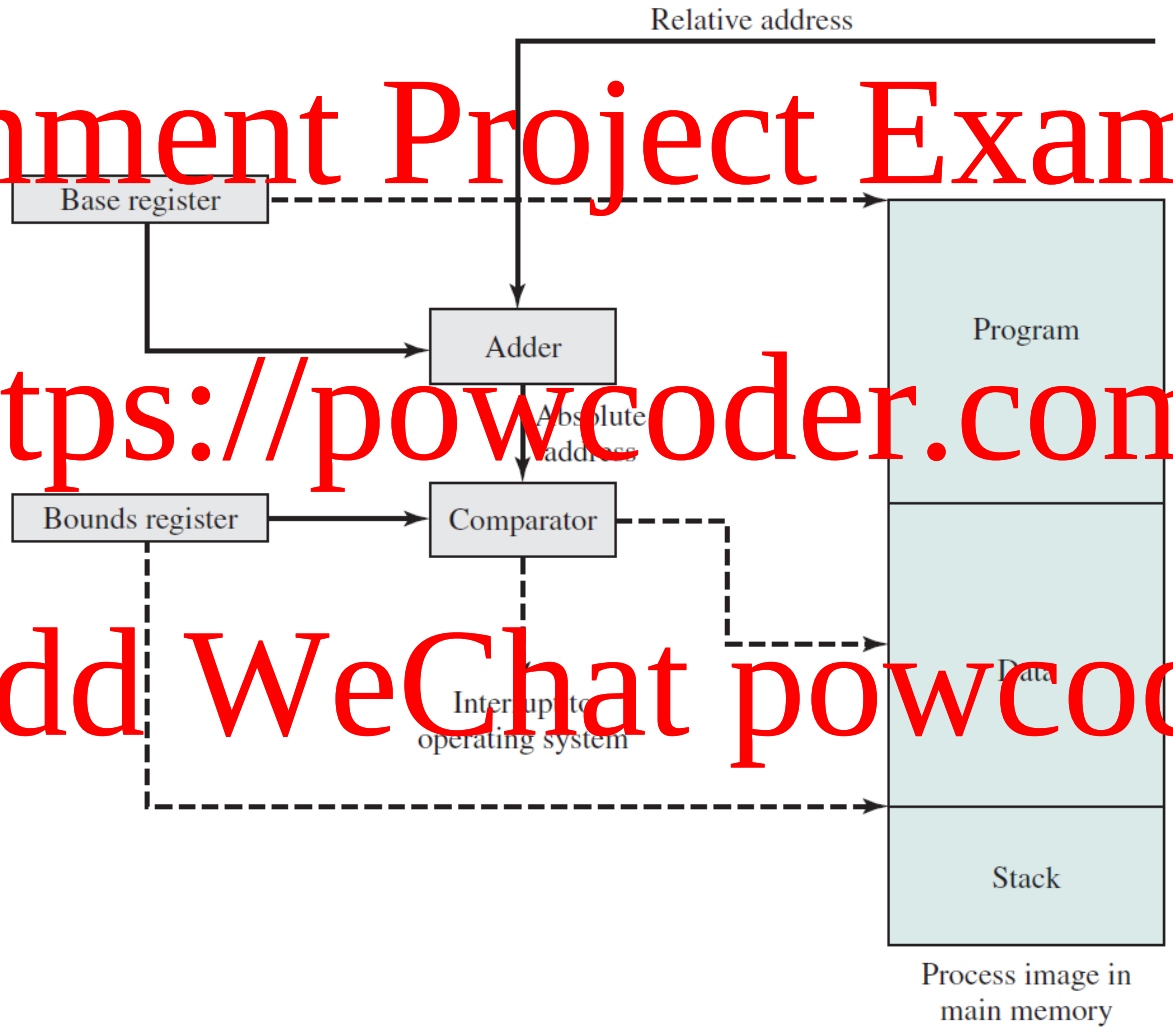

Figure: Address Relocation (Stallings)

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

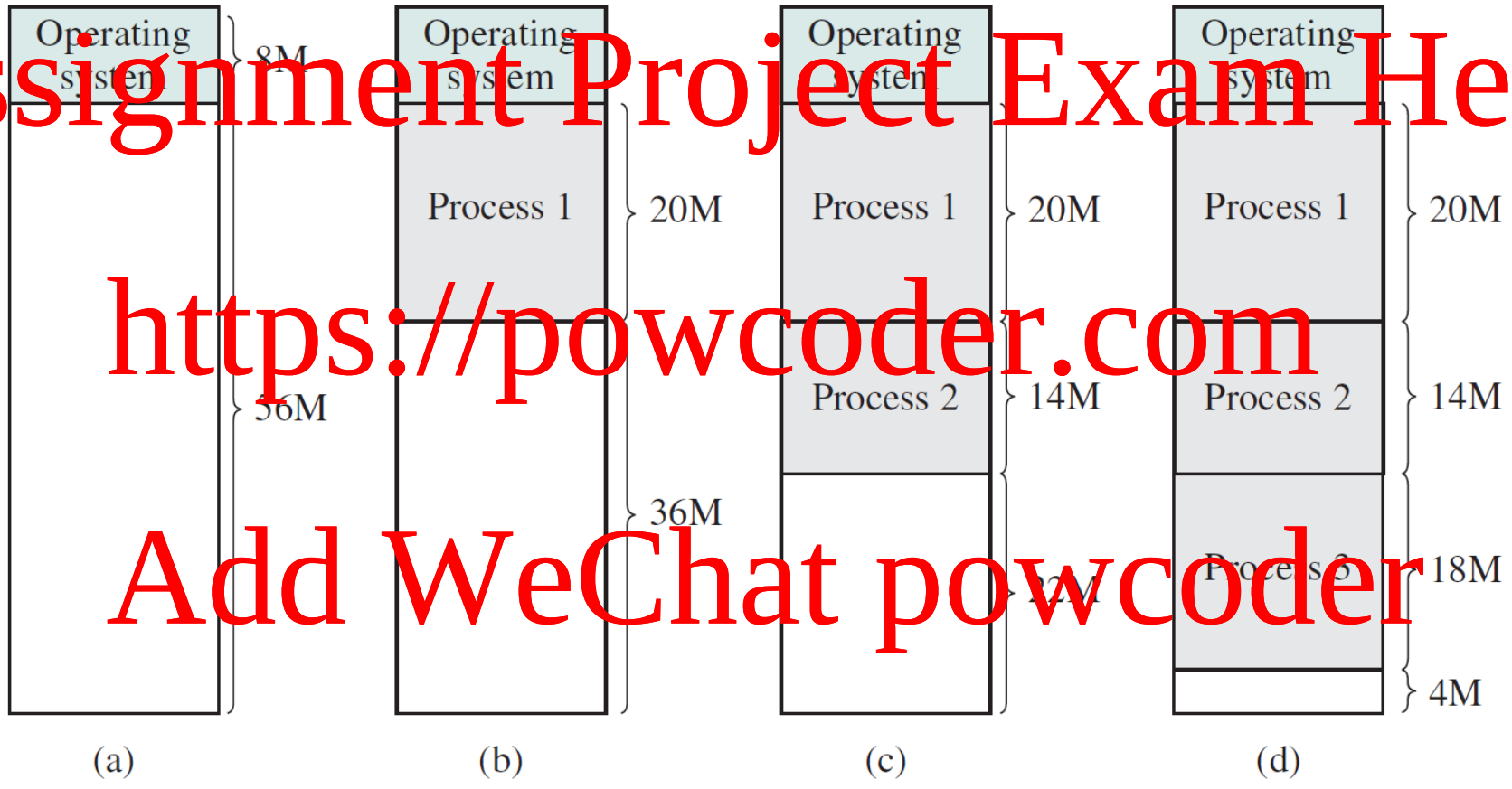

Figure: Dynamic partitioning (from Stallings)

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:
  - Some **processes** only **run occasionally**
  - We have **more processes** than **partitions** (assuming fixed partitions)
  - A process's **memory requirements** have **changed**, e.g. increased
  - The **total amount of memory that is required** for the processes **exceeds the available memory**

Dynamic Partitioning

Difficulties

- The exact **memory requirements** may **not be known** in advance (**heap** and **stack** grow dynamically)
  - ⇒ Allocate the current requirements + "a bit extra"?

Figure: Memory organisation of a process

Dynamic Partitioning

Swapping: Example

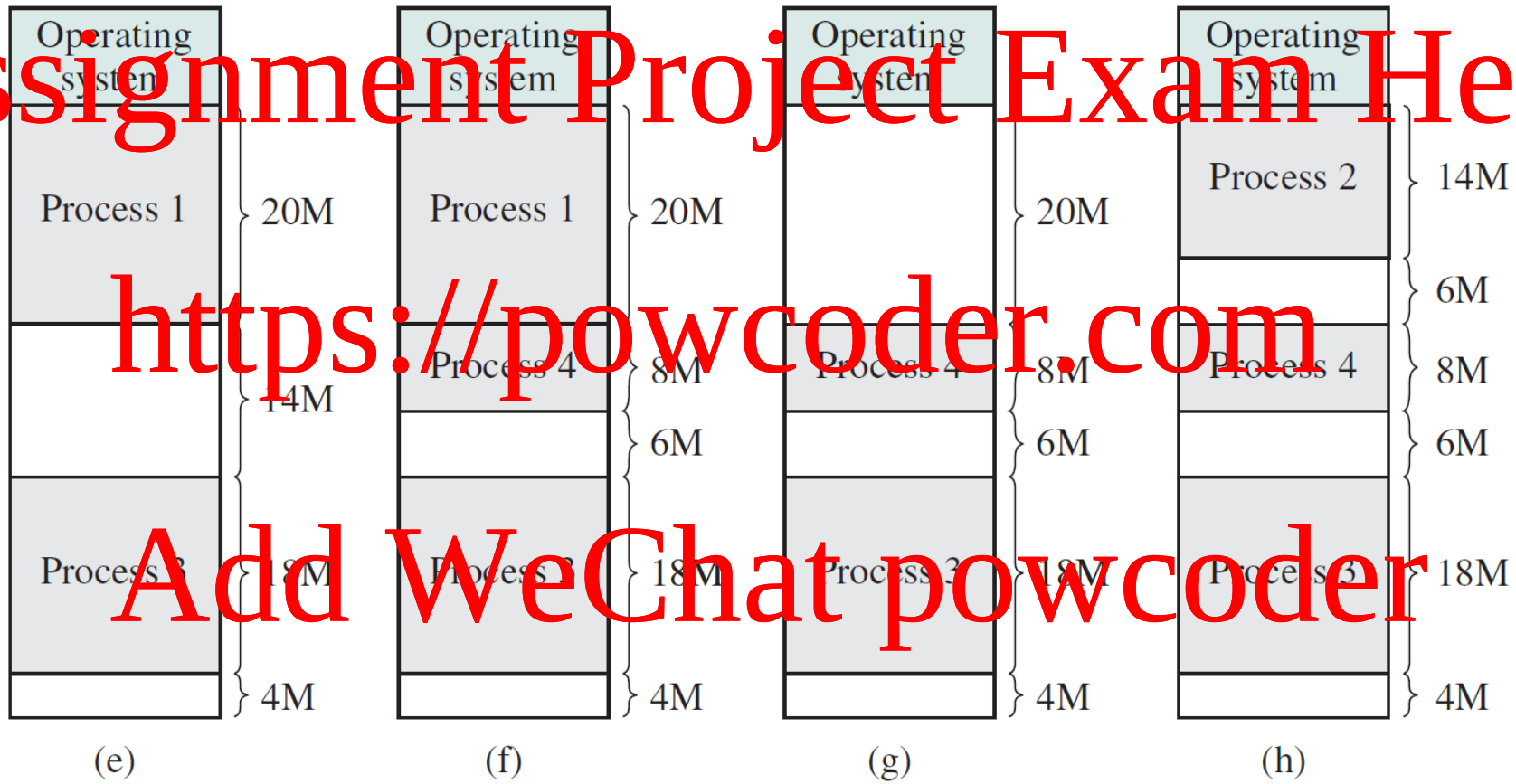

Figure: External fragmentation (from Stallings)

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**
  - Bitmaps
  - Linked lists
- What strategies can we use to (quickly) **allocate** processes to available memory ("holes")?

Dynamic Partitioning

Allocation Structures: Bitmaps

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

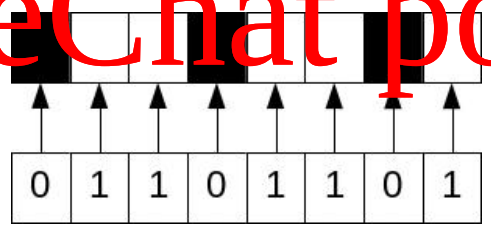

Figure: Memory management with bitmaps

Dynamic Partitioning

Allocation Structures: Bitmaps (Cont'ed)

- 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)
- 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

Dynamic Partitioning

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**

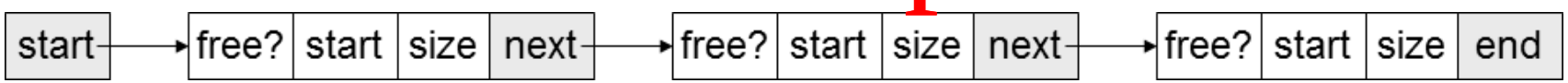

Figure: Memory management with linked lists

Dynamic Partitioning

Allocation Structures

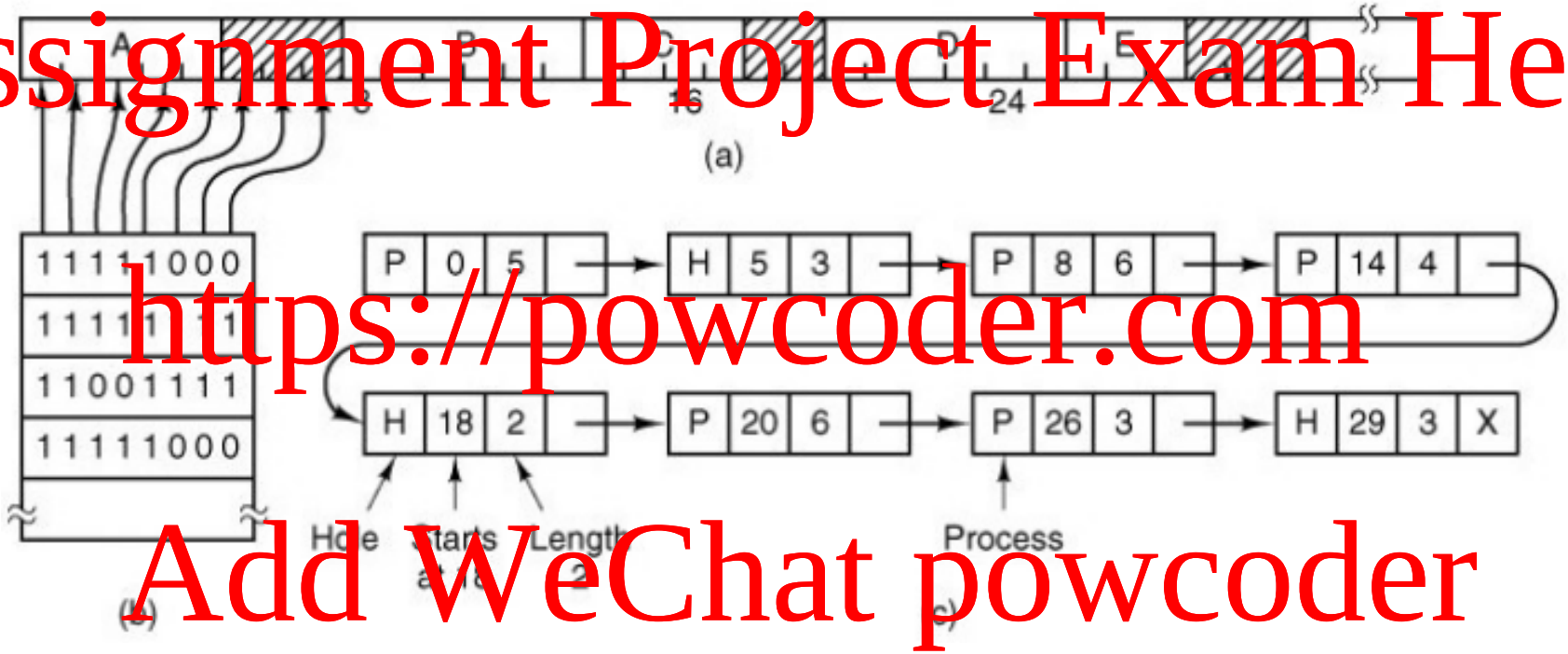

Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list. (from Tanenbaum)

Summary

Take Home Message[2]

- **Contiguous** memory schemes: mono-programming, static and **dynamic partitioning**
- **Relocation** and protection $\Rightarrow$ principles
- Internal and external **fragmentation**

[2] Resources: Tanenbaum Section 3.1, 3.2. Stallings Section 7.1, 7.2

Problem (modified from Tanenbaum)

Allocation Structures

- Compare the storage needed to keep track of free memory using bitmap vs. linked list with a main memory of 8 gygabytes, and a block size of 1 megabyte. For the linked list, assume that exactly half of the memory is in use, and that memory contains an alternating sequence of occupied blocks and free blocks. We will **only keep track of free blocks** with this list, assuming that each node needs a 32-bit memory address, a 16-bit length, and a 16-bit next-node field.
  How many bytes of storage are required for each method?

- Submit your answers at:
  `https://b.socrative.com/login/student/`
  Room name: G52OSC