1 Relocation and Protection

1.1 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

1.2 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 machines physical address space

1.3 Approaches

- *Static relocation* at **compile** time: a process has to be located at the same location every single time (impractical)
- *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
- *Dynamic relocation* at **runtime**

1.4 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!)

2 Dynamic partinioning

2.1 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.

2.2 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 processs memory requirements have changed, e.g. increased
- The total amount of memory that is required for the processes exceeds the available memory

2.3 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**

2.4 Allocation Structures: Bitmaps

The simplest data structure that can be used is a form of **bitmap**

- Memory is split into blocks of say 4 kilobyte 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*220 / 4K blocks => 8192 bitmap entries 8192 bits occupy 8192 / 8 = 1K bytes of storage (only!)
- The size of this bitmap will depend on the size of the memory and the size of the allocation unit

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

2.5 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**

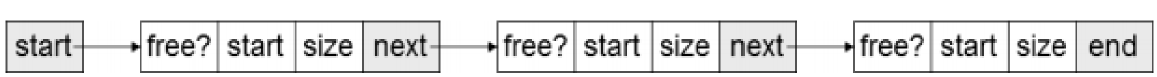

Figure: Memory management with linked lists

Reference section

placeholder