# Lecture 10: Introduction to Memory Management

Adam Hawley

February 21, 2019

## Contents

## 1 Background

### 1.1 Introduction

Programs must be brought (from storage) into memory and placed within a process for it to be run. The main memory and registers are the only storage entities that a CPU can access directly. The CPU fetches instructions from main memory according to the value of the program counter. A typical instruction execution cycle looks like this:

1. Fetch instruction from memory
2. Decode the instruction
3. Operand fetch

4. Possible storage of result in memory

Memory units only see a stream of one of the following:

- Read request + address
- Write request + data + address

Memory unit does not know how these addresses are generated. Register access can be done in one CPU clock while completing a memory access may take many cycles of the CPU clock. In this case the processor needs to **stall** since it does not have the data required to complete the instruction it is execution. (In reality not 100% true because modern CPUs use techniques such as *out-of-order execution.*

The **Cache** sits between the main memory and CPU registers to mitigate the *stall issue.* Protection of memory is required to ensure correct operation. User processes should not be able to access OS memory and one user should not be able to access the memory of another user process.

## 1.2 Address Spaces

A logical address space is a range of addresses that an operating system makes available to a process. It is up to the OS to enforce **memory protection**. Address space endpoints are a **base** register (holding the smallest legal physical address of the process in the memory) and a **limit** register (specifies the size of the address space). The CPU must check that every memory access generated in user mode is between the `base` and `base + limit` for that process.

A program residing on the disk needs to be brought into memory in order to execute. In general, we do not know a priori where the program is going to reside in memory.

- Addresses in the source program are generally symbolic
  - e.g `count`
- A compiler typically binds these symbolic addresses to relocatable addresses
  - e.g "14 bytes from the beggining of this module"
- **Linker** or **loader** will bind relocatable addresses to absolute addresses

Addresses are represented in different ways at different stages of a programs life:

**Compile Time** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.

**Load Time** If memory location is not known at compile time and no hardware support is available, **relocatable code** must be generated.

**Execution Time** (Most common in general computing) Binding delayed until run time if the process can be moved during its exectuion from one memory segment to another. This needs hardware support for address maps (e.g base and limit registers).

### 1.3 Logical vs. Physical Address Space

The concept of a **logical address space** that is bound to a separate **physical address space** is central to proper memory management.

**Logical Address** Issued by the CPU, within processs address space

**Physical Address** Address seen by the memory unit.

Logical and physical addresses are:

- The same in compile-time and load-time address-binding schemes
- Different in execution-time address-binding schemes.

In the latter case, the logical address can be referred to as the virtual address.

**Logical Address Space** The set of all logical addresses generated by a program.

**Physical Address Space** The set of all physical corresponding to a given logical address space.

### 1.4 Memory Management

A **Memory Management Unit (MMU)** is a hardware device that at runtime maps logical addresses to physical addresses. The user program deals with logical addresses and never sees the real physical addresses. Execution-time binding occurs when reference is made to location in memory. Logical addresses bound to physical addresses.

## 2 Contiguous Memory

### 2.1 Single-User Contiguous Memory

First computers: all memory assigned to a single job. Key points: contiguous + entirely assigned. Advantages:

- Very simple
- Address resolution: trivial (physical address = issued address)

Disadvantages:

- Only one job can run at a time so this cannot support multi-programming.
- Processor unused during I/O operations.

### 2.2 Fixed Contiguous Partitions

OS assigns one partition per process, size of partitions defined at boot time and never changes. Key point is that it has protection against memory intrusion. The OS must be assigned its own partition. Upon starting a new process, the OS has to:

1. Determine the relevent partition
2. Determine the start address within the active partition
3. Resolve addresses: `physicalAddress = issuedAddress + fixedBaseRegister`

The problem with this approach is that it is often difficult to choose the right partition sizes which can cause the following. **Internal fragmentation** is when a process may require less space than the available partition. Or process creation may fail even though there may be enough free memory due to wasted memory by small jobs.

### 2.3 Dynamic Contiguous Partitions

Partition size is selected when the job is loaded. Address resolution becomes:

```
physicalAddress = issuedAddress + variableBaseRegister
```

This approach alleviated the problems of fixed contiguous partitioning but does not solve it completely. It is still possible to get **External Fragmentation** where the OS has to keep track of free partitions.

### 2.3.1 Partition Allocation Problem

How to satisfy a request of size $n$ from a list of free partitions?

**First-fit** Allocate the first partition that is big enough

**Best-fit** Allocate the smallest partition that is big enough

- Must search entire list, unless the list is ordered by size
- Produces the smallest leftover partition

**Worst-fit** Allocate the largest partition

- Must also search entire list, unless the list is ordered by size
- Produces the largest leftover partition

**Random** As it sounds, take a random partition.

There is no clear winner as performance of each depends on the request patterns

### 2.3.2 Mitigation of External Fragmentation

External fragmentation can be mitigated by a **compaction** (or defragmentation) procedure. This requires **relocatable partitions** where the base register needs to be changed. The compaction algorithm needs spare memory space to operate efficiently (i.e to move small partitions out of the way before large partitions can be relocated). Compaction cannot be performed while I/O is in progress involving memory that is being compacted. Alternatively, the CPU can latch the process in memory while it is involved in I/O, or do I/O only into OS buffers (i.e double buffering).

## 2.4 Swapping

A process can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued exectution. This means that the total physical memory space of the processes can exceed physical memory.

The **Backing Store** is a fast disk large enough to accomodate binaries of all processes. A major part of swap time is transfer time and the total time is directly proportional to the amount of memory swapped.