Lecture 8:

Memory Management I:
Introduction and Contiguous Allocation

`www.cl.cam.ac.uk/Teaching/2001/OSFounds/`

Next four lectures

To improve CPU utilisation, we'll keep several processes in memory.

- We need a **memory-management** scheme.

As main memory is too small

- We will use **secondary storage** (disk).

Next four lectures:

1. Memory management I
2. Memory management II
3. I/O
4. Filing systems

Today's Lecture

Today we'll cover:

- How do we manage memory when sharing the CPU between many processes?
  - Objectives,
  - Logical vs. Physical Addresses,
  - Contiguous allocation,
  - Fragmentation and Compaction.

Memory Management

In a multiprogramming system:

- many processes in memory simultaneously
- every process needs memory for:
  - instructions ("code" or "text"),
  - static data (in program), and
  - dynamic data (heap and stack).
- in addition, operating system itself needs memory for instructions and data.

$\Rightarrow$ must share memory between OS and k processes.

The memory magagement subsystem handles:

1. Relocation
2. Allocation
3. Protection
4. Sharing
5. Logical Organisation
6. Physical Organisation

The Address Binding Problem

Consider the following simple program:

```
int x, y;
x = 5;
y = x + 3;
```

We can imagine that this would result in some assembly code which looks something like:

```
str #5, [Rx]        // store 5 into 'x'
ldr R1, [Rx]        // load value of x from memory
add R2, R1, #3      // and add 3 to it
str R2, [Ry]        // and store result in 'y'
```

where the expression '`[ addr ]`' means "the contents of the memory at address addr".

Then the address binding problem is:

what values do we give `Rx` and `Ry`?

This is a problem because we don't know where in memory our program will be loaded when we run it:

- e.g. if loaded at 0x1000, then `x` and `y` might be stored at 0x2000, but if loaded at 0x5000, then `x` and `y` might be at 0x6000.

Address Binding and Relocation

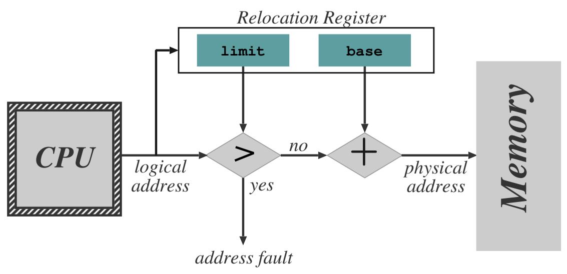
To solve the problem, we need to translate between program addresses and real addresses.

This can be done:

- at compile time:
  - requires knowledge of absolute addresses
  - e.g. DOS .com files
- at load time:
  - when program loaded, work out position in memory and update code with correct addresses
  - must be done every time program is loaded
  - ok for embedded systems / boot-loaders
- at run-time:
  - get some hardware to automatically translate between program and real addresses.
  - no changes at all required to program itself.
  - most popular and flexible scheme, providing we have the requisite hardware (MMU).

Logical vs Physical Addresses

Mapping of logical to physical addresses is done at run-time by Memory Management Unit (MMU), e.g.



1. **Relocation register** holds the value of the base address owned by the process.
2. Relocation register contents are added to each memory address before it is sent to memory.
3. e.g. DOS on 80x86 — 4 relocation registers, logical address is a tuple (s, o).
4. NB: **process never sees physical address — simply manipulates logical addresses**.
5. OS has privilege to update relocation register.

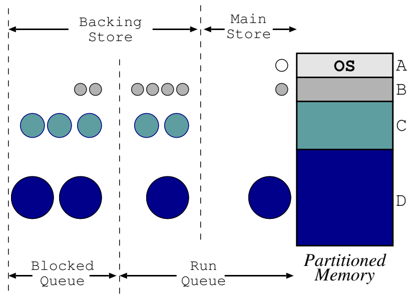
Contiguous Allocation

Given that we want multiple virtual processors, how can we support this in a single address space?

Where do we put processes in memory?

- OS typically must be in **low memory** due to location of interrupt vectors
- Easiest way is to statically divide memory into multiple fixed size **partitions**:
  - bottom partition contains OS, remaining partitions each contain exactly one process.
  - when a process terminates its partition becomes available to new processes.
    e.g. OS/360 MFT.
- Need to protect OS and user processes from malicious programs:
  - use base and limit registers in MMU
  - update values when a new processes is scheduled
  - NB: solving **both** relocation and protection problems at the same time!

Static Multiprogramming

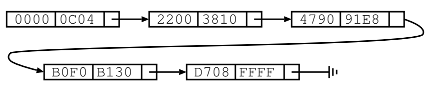


- Partition memory when installing OS, and allocate pieces to different job queues.
- associate jobs to a job queue according to size.
- swap job back to disk when:
  - blocked on I/O (assuming I/O is slower than the backing store).
  - time sliced: larger the job, larger the time slice
- run job from another queue while swapping jobs
- e.g. IBM OS/360 MVT, ICL System 4
- Problems: fragmentation, cannot grow partitions.

Dynamic Partitioning

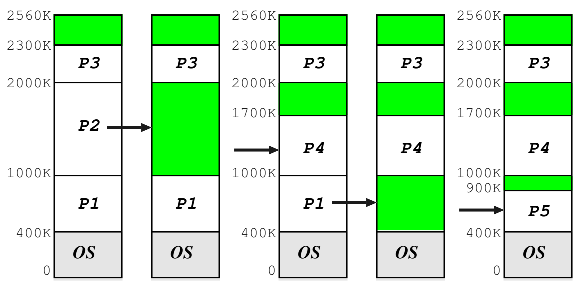
Get more flexibility if allow partition sizes to be **dynamically** chosen (e.g. OS/360 MVT) :

- OS keeps track of which areas of memory are available and which are occupied.
- e.g. use one or more *linked lists*:



- When a new process arrives the OS searches for a hole large enough to fit the process.
- To determine which hole to use for new process:
  - **first fit**: stop searching list as soon as big enough hole is found.
  - **best fit**: search entire list to find "best" fitting hole (i.e. smallest hole large enough)
  - **worst fit**: counterintuitively allocate largest hole (again must search entire list).
- When process terminates its memory returns onto the free list, coalescing holes where appropriate.

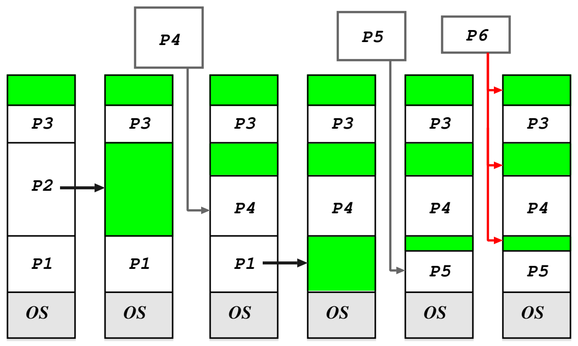
Scheduling Example



- Consider machine with total of 2560K memory.
- Operating System requires 400K.
- The following jobs are in the queue:

Process	Memory	Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

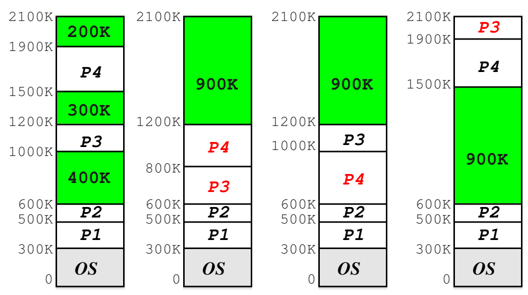
External Fragmentation



- Dynamic partitioning algorithms suffer from external fragmentation: as processes are loaded they leave little fragments which may not be used.
- **External fragmentation** exists when the total available memory is sufficient for a request, but is unusable because it is split into many holes.
- Can also have problems with tiny holes

Solution: compact holes periodically.

Compaction



Choosing optimal strategy quite tricky. . .

Note that:

- Require run-time relocation.
- Can be done more efficiently when process is moved into memory from a swap.
- Some machines used to have hardware support (e.g. CDC Cyber).

Also get fragmentation in *backing store*, but in this case compaction not really viable. . .

Summary

You should now understand:

- What memory management aims to achieve,
- Logical vs. Physical addresses,
- Contiguous allocation,
- External fragmentation,
- Compaction.

Next lecture: Memory Management II: Paging and Segmentation

Background Reading:

- Silberschatz et al.: – Sections 9.1–9.3 incl.