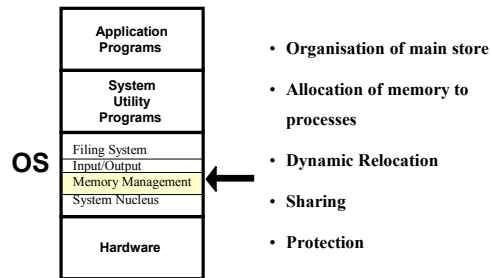


## Chapter 5: Memory Management



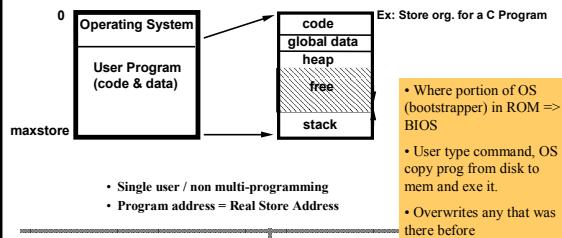
## Overview

• **Memory Manager** – administer use of primary memory, inc automatic movement of programs/data between primary and secondary storage

- Aims:
  - Access time must be as small as possible (hw and sw design)
  - Must be as large as possible (or at least appear so)
  - Cost effective (small % of overall cost of computer of early 90's)
- Registers are memory at CPU speeds
  - Normal primary mem can be around 1.5 - 4 times slower!
- Functions of memory manager:
  - Allocate mem to process
  - Process is allocated address space - Map process address space to actual memory
  - Minimise access times
- History repeating! – simplest memory man not used today (or is it?)

palm, embedded, smart card systems!!

## Single Contiguous Store Allocation

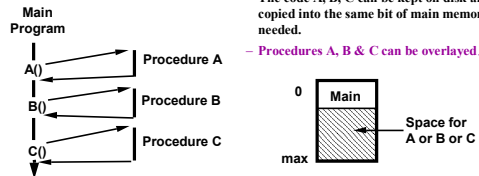


- Advantages**
- Simple - no special H/W
  - No need for address relocation

- Disadvantages**
- No protection - Single User
  - Programs must fit into store (or use overlays see next slide)
  - Waste of unused store

## Overlays - example

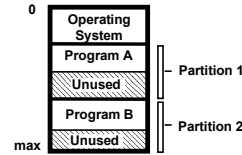
Procedures A, B, C do not call each other – so only Main & A, Main & B, or Main & C need be resident in memory at any one time.  
– The code A, B, C can be kept on disk and copied into the same bit of main memory when needed.  
– Procedures A, B & C can be overlaid.



- Disadvantages**
- Programmer responsible for organising overlay structure of program.
  - OS only provides the facility to load files into overlay region.

- What about multi programming?

## Fixed Partition Store Allocation



- Permits multi-programming.
- Need **relocating loader**.
- Usually some H/W protection between partitions.

### Disadvantages

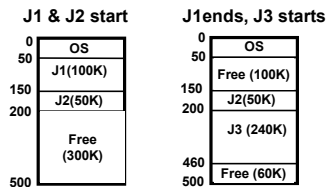
- Wasted space - pre-defined partitions
- Jobs must fit into partition (or use overlays).
- Could be enough unused store to run a large job but not enough in any one partition.

## Dynamic Partition Store Allocation

Store organised as before but partition created dynamically at the start of a job. i.e. OS attempts to allocate a contiguous chunk of store = job size. Eg OS MVT (IBM)

**Example:**  
Total store = 500K

J1 requires 100K  
 J2 requires 50K  
 J3 requires 240K  
 J4 requires 120K



**Store Fragmentation** ⇒ cannot start J4 because even though 160K of free store, there is not a contiguous 120K chunk.

## Compaction

The **fragmentation problem** could be solved by moving running programs to close free gaps and create larger ones

### Dynamic Program relocation

#### Problem:

- When a program is loaded into store by the operating system, program addresses = **real** store locations.
- Programs use **absolute address** to access operands from real store locations.
  - To move a program the operating system would have to modify these addresses.
- **But** the relocation information produced by the assembler/linker/loader is not maintained in main store.
- In general:-  
Programs are NOT dynamically relocatable if **program address = real store address**.

## Virtual Addressing

- Permits dynamic relocation
- Program uses virtual addresses which the H/W maps to real store locations.

$$R = \text{Amap}(A)$$

**R** = Real Address or physical store location.  
**Amap** = address mapping function.  
**A** = Virtual Address or program address.

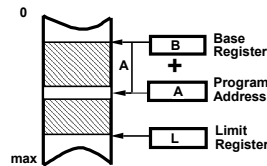
Modification of NARC to handle Virtual Addressing:

```

void fetch() {
    int W;
    W = Mem[Amap(PC)];
    Op = Opcode(W);
    D = Operand(W);
}

void execute() {
    switch (Op) {
        % only the modified cases are shown here
        case 2:  Acc = Mem[Amap(D)]; break; % loadm
        case 3:  Mem[Amap(D)] = Acc; break; % storem
        case 5:  Acc = Acc + Mem[Amap(D)]; break; % addm
        case 7:  Acc = Acc - Mem[Amap(D)]; break; % subm
    }
}
    
```

## Base & Limit Register - Virtual Addressing



### Dynamic Relocation:

- Hardware registers.
- When process scheduled base register loaded with the address of the start of its partition (limit end)
- Every memory address generated auto has base added to it before being sent to memory.
- Original instruction not modified

$$\text{Amap}(A) = A + B \quad \text{protection error if } A < 0 \text{ or } A + B > L$$

#### Advantages

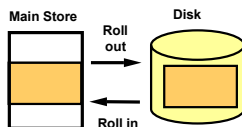
- Dynamic partitions + relocation
- Relocation permits compaction
- Simple hardware
- Programs can request and release store dynamically

#### Disadvantages

- Fragmentation requires compaction which consumes CPU cycles
- Virtual address space  $\leq$  real address space

## Swapping - Roll-in / Roll-out

A program occupies store even when it is waiting for I/O or some other event.



- Swap (roll-out) user programs to backing store (disk) when they are waiting. Only if waiting for slow I/O i.e. wait time  $\gg$  swap time.
- Swap back into store (roll-in) when event occurs. (May swap back into a different partition).

#### Advantages

- Allows higher degree of multi-programming.
- Better utilisation of store.
- Less CPU time wasted on compaction

#### Disadvantages

- Virtual address space  $\leq$  real address space
- Whole program must be in store when it is executing

## So far.....

### Mechanisms

- Fixed Partition  $\rightarrow$  Wasted Space
- Dynamic partition  $\rightarrow$  Fragmentation
- Relocatable (base & limit) V. addressing  $\rightarrow$  Compaction overhead to defrag
- Swapping  $\rightarrow$  Swap entire process

### Unresolved Problems:

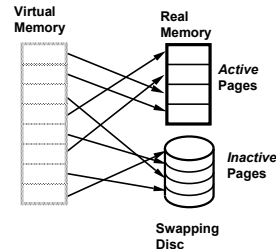
- (1) The entire program must be resident in store when executing.
- (2) Maximum program size ≤ available physical store.
- (3) Programs must occupy contiguous real store.

#### Solution:

- ✉ Map contiguous virtual address space onto **discontiguous chunks of real store**. solves (3) and reduces requirement for compaction.
- ✉ Make secondary or backing store (disks) appear as a logical extension of primary (main) memory. Chunks of program which are not currently needed can be kept on disk. solves (1) & (2)

### Paging

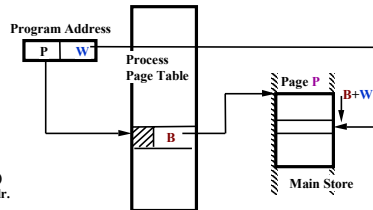
Virtual address space is divided into *pages* of equal size.  
Main Memory is divided into *page frames* the same size.



- Running or ready process
  - some pages in main memory
- Waiting process
  - all pages can be on disk
- Paging is transparent to programmer

**Paging Mechanism**  
(1) Address Mapping  
(2) Page Transfer

### Paging - Address Mapping

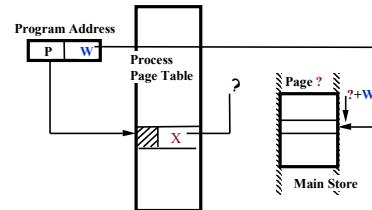


**Example:** Word addressed machine, W = 8 bits, page size = 256  

$$Amap(P, W) = PPT[P] * 256 + W$$

**Note:** The Process Page Table (PPT) itself can be paged

### Paging - Page Transfer



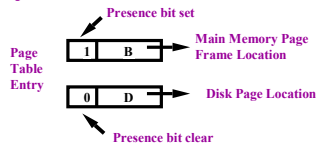
- Means that the page is not currently in memory and must be fetched from disk.

## Paging - Page Transfer

What happens when we access a page which is currently not in main memory (i.e. the page table entry is empty)?

- ✉ **Page Fault**
- Suspend running process
  - Get page from disk
  - Update page table
  - Resume process (re-execute instruction)

The location of a page on disk can be recorded in a separate table or in the page table itself using a **presence bit**.



Note: We can run another *ready* process while the page fault is being serviced.

## What happens if there are no free page frames?

- ✉ Must swap out some other pages to disk.

Which pages? (*see later*)

- ✉ Decided by the **Page Turning Algorithm** based on information in the Page Table. e.g.

- (a) How many times page has been referenced (*use count*).
- (b) Time page was last referenced.
- (c) Whether page has been written to (*dirty/clean bit*).

**Problem so far -**

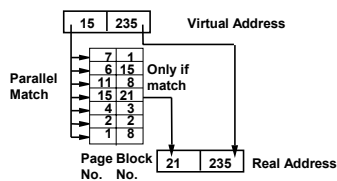
We have doubled memory access time since each memory access involves a page table access.

This could be solved by putting the page table in **fast store** such as registers but the size of the page table is proportional to the virtual address space.

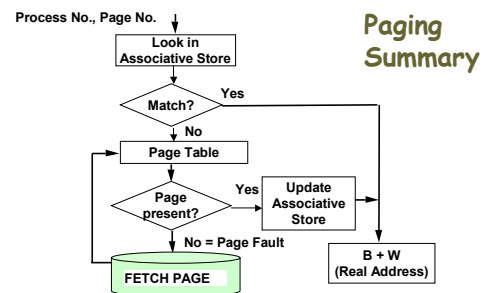
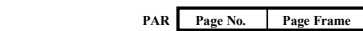
⇒ Too big!

## Associative Store (Content Addressable)

Translation  
Lookaside  
Buffers



- Page Address Registers (PARs) in Associative store map page no. to real block no.
- PARs contains only active page references.
- No. PARs = No. blocks in Real Memory
- With large real memories No. PARs ≪ No. blocks.
- ⇒ Use a small associative store to reference some of the most recently active pages.
- 16 -32 PARs often enough to give 98% hit rate.
- If not found in associative store, then have to use Process Page Table.



### ADVANTAGES OF PAGED SYSTEMS:

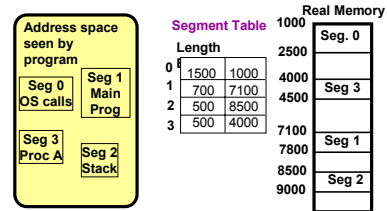
- Fully utilise available main store.
- Can execute programs where  
 $\text{Virtual Address Space} > \text{Main Memory Size}$ .
- No fragmentation or compaction.

### DISADVANTAGES:

- Complex hardware and software.
- Large overhead - not always worth it?
- Still only contiguous virtual store i.e. **not divided into logical units to reflect logical divisions in program and data.**

## Segmentation

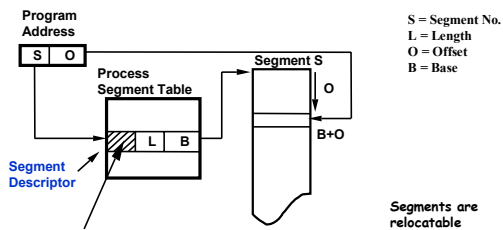
Program Address Space divided into **segments** to reflect the logical program structure  $\Rightarrow$  **functional division**:  
 i.e. procedures, program module, stack, heap, collection of data, etc.



Program sees a **two-dimensional** virtual address space (Segment, Offset)

## Segmentation

**Address Mapping**  
 $\text{Amap}(S,O) = \text{PST}[S].B + O$



### PROTECTION BITS:

- May be used to indicate – Code (Execute), Read Only data, Read/Write data, Shared Segment etc.
- **SHARED SEGMENTS:** Enter segment descriptor in segment table of each process which wants to share the segment (perhaps with different protection bits).

## Comparison of Paging & Segmentation:

### Paging

- Physical division of memory to implement one-level or virtual store. Transparent to programmer.
- Fixed size.
- Division of program address into page no., word no. is done by H/W, overflow of word no. increments the page no.

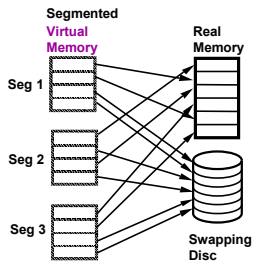
### Segmentation

- Logical division of program address space. Two dimensional store visible in assembly code.
- Variable size up to some limit.
- Division into segment number, offset is logical, consequently overflow of segment offset is a protection violation.

In practice, it is often not possible to fit all segments for a program into main memory at once (virtual memory > real memory), so -

**Swap Segments or Use Paging**

## Paged Segmented Store



Segments divided into fixed sized pages.

Whole segments need not be brought into main memory - pages brought in as accessed, so no overlaying.

Segmentation Performs: Limit and protection checks

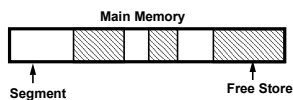
Paging Performs: Automatic swapping - invisible to programmer.

Segmented page table:  $\Rightarrow$  smaller page table, collapse unused parts of it.

## Memory Management Policies

- Allocation/ Placement
  - How to allocate memory
  - Where should information be loaded into main memory?
  - Trivial for paged systems
    - $\Rightarrow$  any available block (page frame) can be allocated
- Replacement
  - What should be swapped out of main memory to create free space?
- Fetch
  - When should information be loaded into main memory? e.g. -
    - on demand
    - in advance

## Placement - non-paged

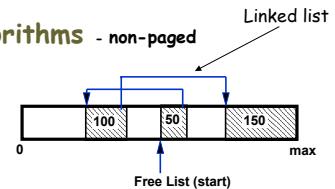


The Operating System maintains a list of free memory chunks usually called the **free list**.

- Allocation Tactics
  - If segment size < free chunk size locate segment at one end of the chunk to minimise fragmentation.
  - If segment size > free chunk size move segments to create larger free chunk (i.e. **compact store**).
- Deallocation Tactics
  - Coalesce free store - Join adjacent chunks to form one contiguous block of storage with **Size = sum of individual chunks**.

## Placement Algorithms - non-paged

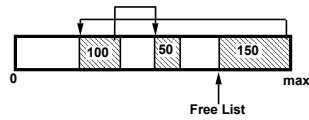
### 1) BEST FIT



- The free chunks are listed in order of **increasing size**.  
 $C_1 \leq C_2 \leq C_3 \leq \dots \leq C_n$
- Allocate Function:  
**Find smallest  $C_i$  such that the segment  $S \leq C_i$  by searching list.**
- Appears to waste least space, but leaves many **unusable small holes**.

## Placement Algorithms - non-paged

### 2) WORST FIT



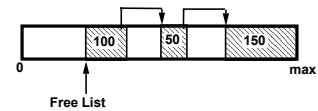
- The free chunks are listed in order of *decreasing* size.  
 $C_1 \geq C_2 \geq C_3 \geq \dots \geq C_n$

• Allocate Function:  
 Place segment in  $C_i$  and link remaining space back into free list, coalesce if possible.

- Allocating from a large chunk leaves a chunk big enough to use. Allocation from a chunk of nearly the right size is likely to leave an unusable small chunk.

## Placement Algorithms - non-paged

### 3) FIRST FIT



- The free chunks are listed in order of increasing base address.

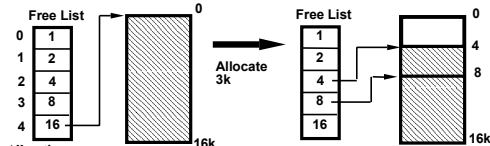
• Allocate Function:  
 Find first hole such that  $S \leq C_i$ .

- Compaction is easier if holes are held in address order.

## Placement Algorithms - non-paged / used by Linux

### (4) BUDDY

Allocation unit is power of 2 i.e. 1k, 2k, 4k ...  
 Separate list for each size in address order.  
 Initially - one large block (size = power of 2).



Allocation:

- Try to remove chunk from list  $i$ , where  $2^i$  is the smallest unit  $\geq S$ .
- If empty, split block list  $i+1$  - allocate half & chain remainder on list  $i$ .

Deallocation

- Return to appropriate list,  $i$
- If adjacent to another block on the *same* list coalesce and add to list  $i+1$ .

Advantages: Fast Allocation, Less Unusable chunks, Reduces Fragmentation.

## Replacement Algorithms - paged

- Decides which page to **remove** from main store
- If page frame has been modified (written to) it must be **written back** to disk, otherwise it can just be **discarded** as it already resides on disk. Therefore choose clean pages first.
- Objective - to replace page which is not going to be referenced for the longest time in the future.
- Difficult ! - the best we can do is to infer future behaviour from past behaviour.

## Replacement Algorithms - paged

### 1) First-in First-out

Replace page which has been resident the longest.

Must use **time stamps** or **maintain loading sequence**.

- Can replace heavily used pages e.g. editor
- FIFO anomaly: more page frames available in store may *increase* no. of faults. e.g. –
  - Try sequence of page refs: A B C D A B E A B C D E with 3 & 4 frames available, initially none resident.

### 2) Least Recently Used (LRU)

Replace page which has least recently been used.

- Usually best page to replace
- Time-stamp every page reference or record sequence of access to pages, so **high overhead**
- Page chosen could be next to be used, e.g. within a large loop

## Replacement Algorithms - paged

### 3) Least Frequently Used (LFU)

Replace page which has been least frequently used during the immediately preceding time interval.

- Keep a per page use count.
- Most recently brought in will have low count, so gets swapped out.

### 4) Not Used Recently

Approximates to LRU, but no time stamp.

Reference Bit: set on reference, cleared periodically.

Dirty Bit: set on write to page, cleared on page-in.

Priority order for page replacement:

- 1) unreferenced, unmodified
- 2) unreferenced, modified
- 3) referenced, unmodified
- 4) referenced, modified

Non-paged are similar, but need to consider segment size.

## Fetch Policies - when to swap in



### DEMAND

- fetch when needed is always required and is provided by the *page fault mechanism*



### ANTICIPATORY

- fetch in advance. This relies on being able to predict future program behaviour.

We can use:

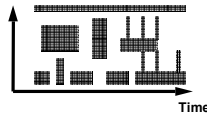
- knowledge of the nature of construction of programs.
- inference from past behaviour.

## Principle of Locality of Reference

If a program accesses a location  $n$  at some point in time it is likely to reference the same location  $n$  and locations in the immediate vicinity of  $n$  in the near future.

**Temporal Locality:**  
Storage locations referenced recently are likely to be referenced again.  
e.g. loops, stacks, variables (counts, pointers).

Storage  
Addresses



**Spatial Locality:**  
Clustered references - once a location is referenced it is likely a nearby location will be referenced.  
e.g. arrays, sequential code.

## Thrashing

(1) As the degree of multi-programming increases so does the percentage CPU utilisation. This is because the scheduler has a greater probability of finding a runnable process.

(2) After a threshold, the degree of multi-programming makes it impossible to keep sufficient pages in store for a process to avoid generating a large number of page faults. This causes:

- an increase in disk paging traffic;
- the disk channel becomes a bottleneck;
- most processes are blocked waiting for page transfer.

**Thrashing = processor spending more time paging than executing programs.**

© Imperial College, Department of computing Memory Management 37

## Working Set

- ☒ Only a subset of the program's code and data (*working set*) is referenced over a given time interval.
- ☒ To avoid thrashing, each process requires a minimum set of pages in store.

Minimum set = *Working Set*.

© Imperial College, Department of computing Memory Management 38

## Working Set Model

The working set is the set of pages a program references within a given time interval (window). From the principle of locality of reference we expect this set to change slowly.

To avoid thrashing: (1) Run a process only if its working set is in memory  
(2) Never remove a page which is part of a working set

**Note:**

- Working set applies to a particular process.
- The particular set of pages constituting the WS changes with time.
- Size of WS also changes with time.

Can increase accuracy by increasing history size, and/or interrupt frequency.  
- increased overheads.

*Note that all the page replacement algorithms discussed earlier implicitly tend to maintain process working sets.*

© Imperial College, Department of computing Memory Management 39

## Page Size Considerations

### Small Pages

- ☒ Less information brought in that is unreferenced.
- ☒ From locality of reference, smaller pages leads to **tighter working sets**.
- ☒ Internal fragmentation - Smaller pages lead to less wastage.

### Large Pages

- ☒ Reduce number of pages, hence page table size.
- ☒ Reduce I/O overheads due to disc seek time if transfer larger pages.
- ☒ Less page faults so reduce overheads of handling page faults.

| Computer   | Page Size (words) | Word Size (bits) |
|------------|-------------------|------------------|
| Multics    | 1024              | 36               |
| IBM 370    | 1024 or 512       | 32               |
| DEC PDP 20 | 512               | 36               |
| VAX        | 128               | 32               |

© Imperial College, Department of computing Memory Management 40

## Memory Management - Summary

### Mechanisms

- Fixed Partition → Wasted Space
- Dynamic partition → Fragmentation
- Relocatable (base & limit) → Compaction Overhead
- Swapping → Swap entire process
- Paging → Linear virtual address space
- Segmentation → 2-Dim. address space

### Policies

- Replacement (what to swap out)
- Fetch (when to load)
- Placement (where to load)

Memory Management means we must add to the process context:  
(1) copy of contents of base & limit registers  
or (2) pointer to page table  
or (3) pointer to segment table

## Module Section Summary

- Overview of the structure of the OS
- Device Drivers IO
- Processes
- Concurrency
- Scheduling
- Memory management