

Operating Systems Concepts: Chapter 8: Memory Management

Olav Beckmann
Huxley 449

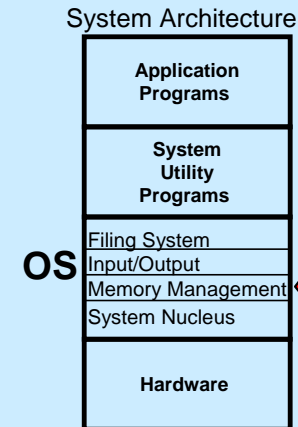
<http://www.doc.ic.ac.uk/~ob3>

Acknowledgements: There are lots. See end of Chapter 1.

- Home Page for the course:
<http://www.doc.ic.ac.uk/~ob3/Teaching/OperatingSystemsConcepts/>
- This is only up-to-date after I have issued printed versions of the notes, tutorials, solutions etc.

Chapter 8: Memory Management

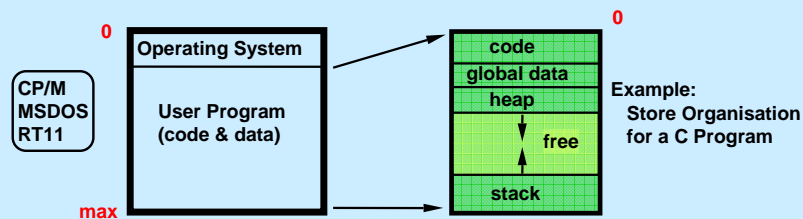
Ch 8a: Fixed Partitions, Dynamic Partitions, Placement, Swapping



Issues covered in Chapter 8

- Organisation of main store
- Allocation of memory to processes
- Dynamic Relocation
- Sharing
- Protection

Single Contiguous Store Allocation



- Single user / non multi-programming
- Program address = Real Store Address

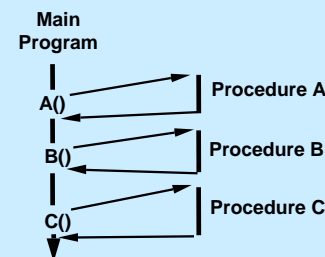
Advantages

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

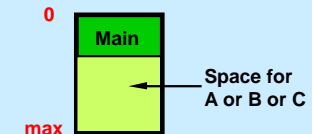
Disadvantages

- No protection - Single User
- Programs must fit into store (or use overlays)
- Waste of unused store

Overlays - Illustrated by example



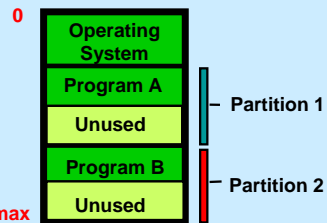
- Procedures A, B, C do not call each other - so only Main & A, Main & B, or Main & C need be resident in memory at any one time.
- The code A, B, C can be kept on disk and copied into the same bit of main memory when needed.
 - Procedures A, B & C can be *overlaid*.



Disadvantages

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

Fixed Partition Store Allocation



- Allocate fixed-size, strongly separated partitions for different tasks
- Permits multi-programming.
- Need relocating loader.
- Usually some H/W protection between partitions.

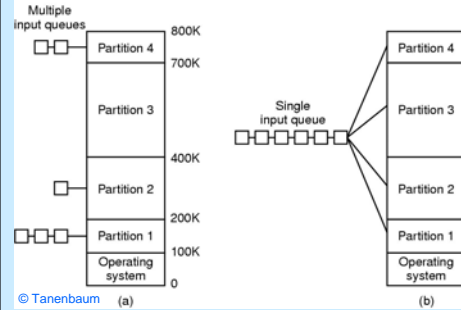
- Potential Disadvantages**
- Potentially sub-optimal use of overall available space
 - Constraints that would not be present if all resources were equally available

Real Example: IBM pSeries



- IBM pSeries offers "logical partitioning": multiprocessor server can be partitioned for multiple services.
- Strong isolation between partitions - good thing.
- Designed to run full OSs in partitions.
- Isolation enforced by hardware + special firmware.

Multiprogramming with Fixed Partitions



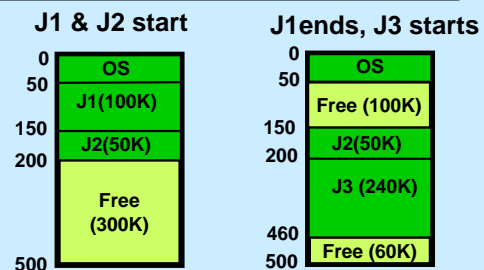
- Multiprogramming improves CPU utilisation:
 - Assume a job normally spends a fraction p of its time waiting for I/O.
 - Then, for one job, CPU utilisation is $1 - p$.
 - For n jobs in a multiprogramming system, approximate CPU utilisation is $1 - p^n$.

- Fixed partitions can be of different sizes.
- Different possible queuing strategies for jobs (see illustration above).
- OS/MFT (Multiprogramming with fixed partitions) used by OS/360 on IBM mainframes for many years.

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.
E.g. OS/MVT ("multiprogramming with a variable number of tasks", IBM 1967-)

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:

```

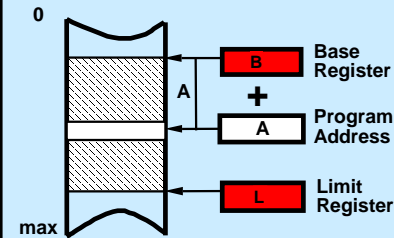
void fetch() {
    int W;
    W = Mem[Amap(PC)];
    Op = Opcode(W);
    D = Operand(W);
}

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 – One way of doing Virtual Addressing

$$\text{Amap}(A) = A + B$$

protection error if $A < 0$ or $A + B > L$



Dynamic Relocation:

Copy user program & data to new part of store and change base & limit registers.

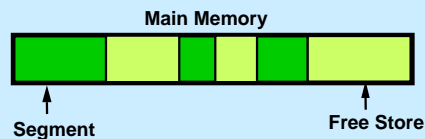
Advantages

- 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

Placement of Tasks in Memory (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.

$$C_1 \leq C_2 \leq C_3 \leq \dots \leq C_n$$

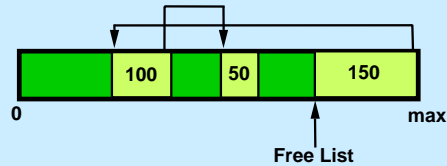
Allocate Function:

Find smallest C_i such that the segment $S \leq C_i$ by searching list.

- Appears to waste least space, but leaves many unusably small holes.

Placement Algorithms (non-paged)

2) WORST FIT



- The free chunks are listed in order of *decreasing* size.

$$C_1 \geq C_2 \geq C_3 \geq \dots \geq C_n$$

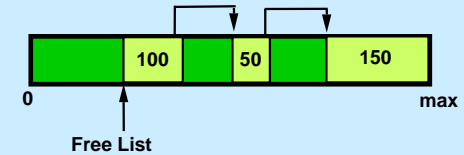
Allocate Function:

Place segment in C_1 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:

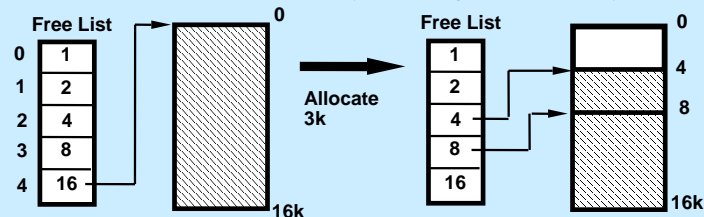
Find first free chunk such that $S \leq C_i$.

- Compaction is easier if free chunks are held in address order.

Placement Algorithms (non-paged)

(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 .

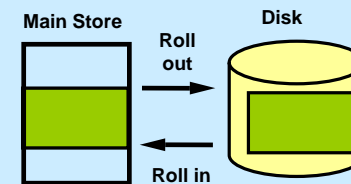
Deallocation

- 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.

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 (swap is DMA).

Disadvantages

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

Memory Management: Summary so far

- Dynamic partition store allocation: each job is allocated a contiguous region of memory
 - Advantage over fixed partition: avoids wasted space
 - Disadvantage: fragmentation – several problems
 - Cannot run jobs larger than available memory
 - Dynamic partition store management with compaction means that we need virtual addressing
 - Placement algorithms (first fit, best fit etc) are about where to allocate a job, given a number of memory regions that are large enough
- Paging deals with all these issues in a different way!

Unresolved Problems in Dynamic Partition Memory Management:

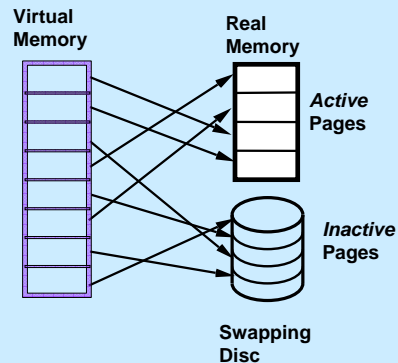
- (1) The entire program must be resident in store when executing.
- (2) Maximum program size \leq available physical store.
- (3) Programs must occupy contiguous real store.

Solution:

- ☛ Map contiguous virtual address space onto discontinuous 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
- Each process has its own virtual address space.

Paging Mechanism

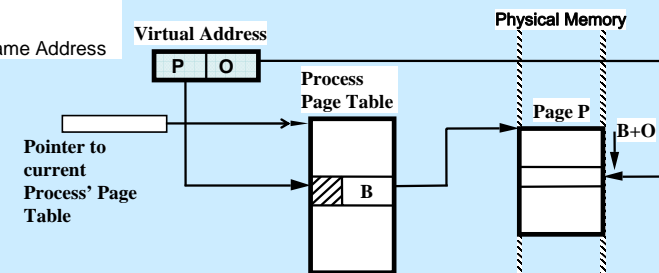
- (1) Address Mapping
- (2) Page Transfer

Paging - Address Mapping

P = Page No.

O = Offset

B = Page Frame Address



Example: Word addressed machine, 8-bit words, page size = 256

$$\text{Amap}(P, O) = \text{PPT}[P] * 256 + O$$

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

Paging: A Detailed Example

Consider a (very) simple computer system that has:

- 32 bytes of main (physical) memory
- 128 bytes of virtual memory
- 8 byte pages
- 256 bytes of disk storage

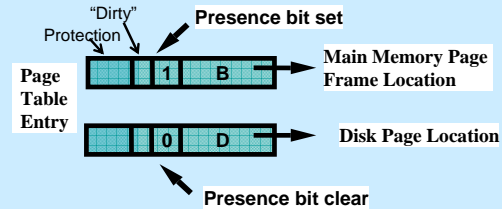
- How big are virtual memory addresses?
- How big are physical memory addresses?
- How big is a process page table entry?

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)
 - ? Can one instruction cause more than one page fault?

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

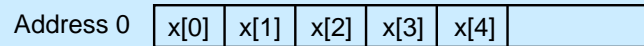


Note: We can run another *ready* process while the page fault is being serviced.

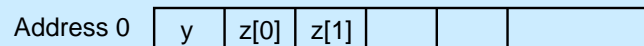
Some Useful Facts

- Kilo, Mega, Giga:
 - 1K = $2^{10} = 1024$
 - 1M = $2^{20} = 1024 * 1024 = \dots$
 - 1G = $2^{30} = 1024 * 1024 * 1024 = \dots$

- Storage layout of arrays in memory
`int x[5];`

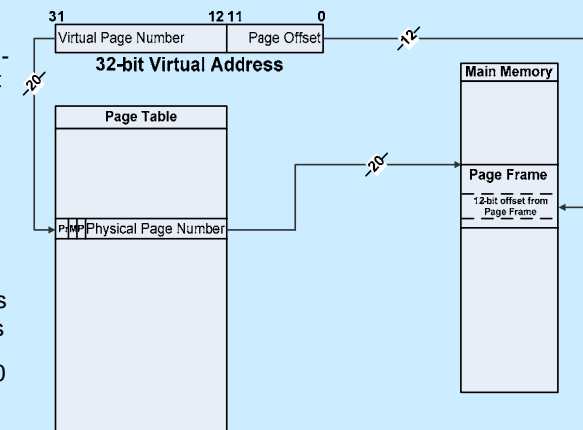


- Storage layout of structures
`typedef struct { int y, int z[2] } mydata;`

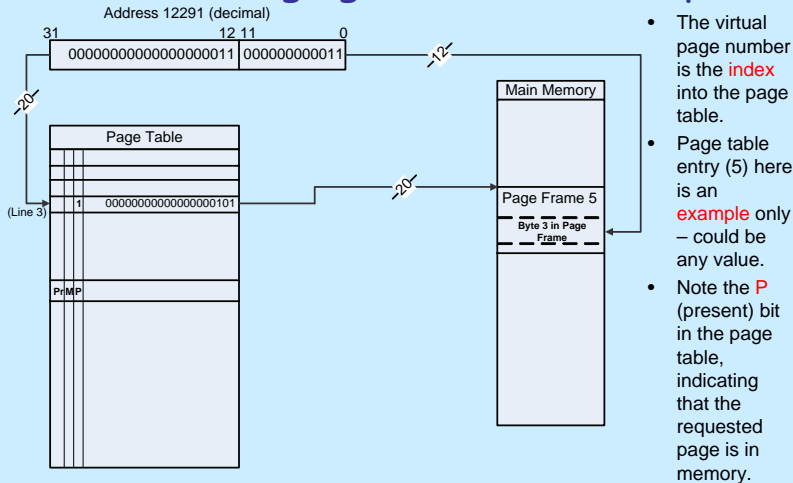


Basic Paging Scheme: A Closer Look

- 32-bit virtual and physical address
- Assume 4k page - 12-bit page offset
- 20-bit page number
- Each page table entry consists of 20-bit phys page number plus status bits
- There are 2^{20} page table entries
- Page table size is at least (20 + status) bits * 2^{20} = approx 3MB.



Basic Paging Scheme - Example



- The virtual page number is the **index** into the page table.
- Page table entry (5) here is an **example** only – could be any value.
- Note the **P** (present) bit in the page table, indicating that the requested page is in memory.

What happens if there are no free page frames?

☞ Must swap out some other pages to disk.

Which pages ?

☞ Decided by the **Page Replacement Algorithm** based on information in the Page Table. e.g.

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

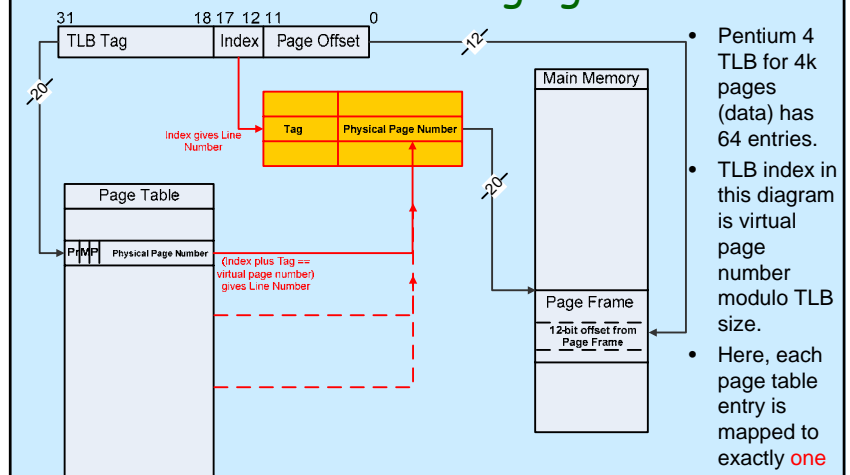
Problem so far -

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 !

Paging with TLB: Basic Idea

- The page table is not small.
- Basic paging scheme requires two memory reads for every virtual memory access.
- Idea: speed up the page table access by holding a small subset of page table entries on-chip, in a special very fast memory.
- This special fast memory is called a **TLB: Translation Lookaside Buffer** (Note: in the Tanenbaum book, it is called an **Associative Store** for reasons which I will explain later.)
- (The term TLB comes from the idea that while performing a virtual address translation, we “look aside” into this buffer first to check whether the result of the lookup is held there.)
- Typical TLB size: 64 entries. (How many bits to index?)
- This idea will pay off if we can make sure that a high proportion of page table accesses can be serviced by this small on-chip buffer.
- Chances should be good if the code we are executing mainly accesses a small number of different pages.

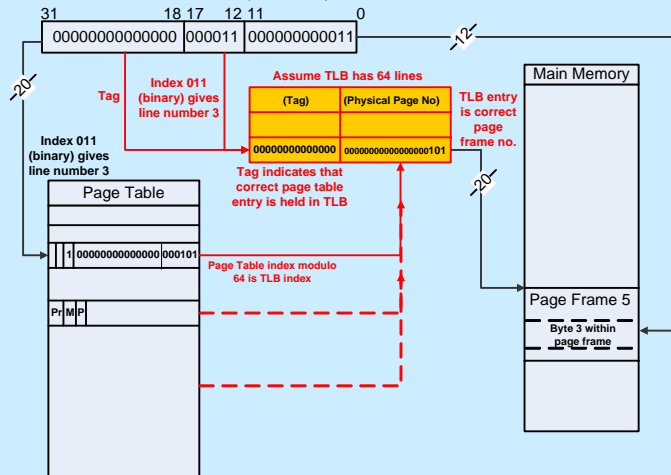
Paging with TLB



- Pentium 4 TLB for 4k pages (data) has 64 entries.
- TLB index in this diagram is virtual page number modulo TLB size.
- Here, each page table entry is mapped to exactly **one** TLB entry.

Paging with TLB: Example

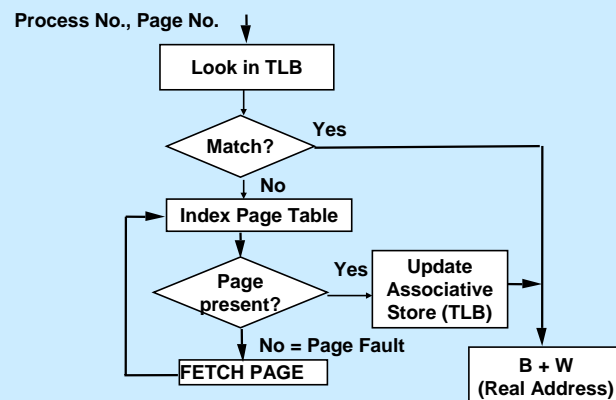
Address 12291 (decimal)



More about TLBs

- In the basic TLB scheme covered so far, each page table entry is mapped to exactly one TLB line: (virtual page number modulo TLB size).
- This can result in pathologically poor performance if we access virtual addresses in a stride that is a multiple of page size and TLB size.
- Better schemes have been developed to deal with this:
 - The idea is that every page table entry can be mapped to more than one TLB entry.
 - If, for example, every page table entry can be mapped to two different TLB entries, this is known as a **2-way set-associative TLB**.
 - The extreme case (not uncommon) is that each page table entry can be mapped to any one of the TLB entries. This is known as a **fully set-associative TLB**, or an **associative store** (in fact, Tanenbaum uses this term for TLBs).
 - Which TLB line is actually used for storing a page table entry can depend on, for example, most recent use etc.
- Important: Because each process has its own page table, and the TLB holds a snapshot of some entries of this page table, we must flush the TLB on a context switch. This makes context switches very expensive in paged operating systems.

Paging Summary



ADVANTAGES OF PAGED SYSTEMS:

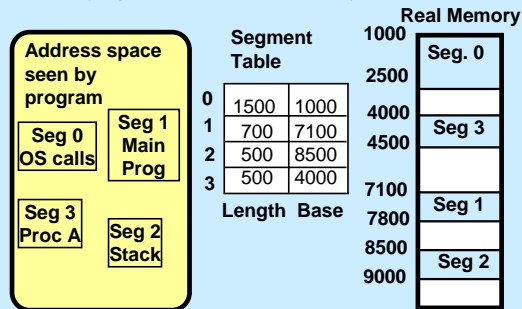
- Fully utilise available main store.
- Can execute programs where
Virtual Address Space > Main Memory Size.
- No fragmentation or compaction.

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.

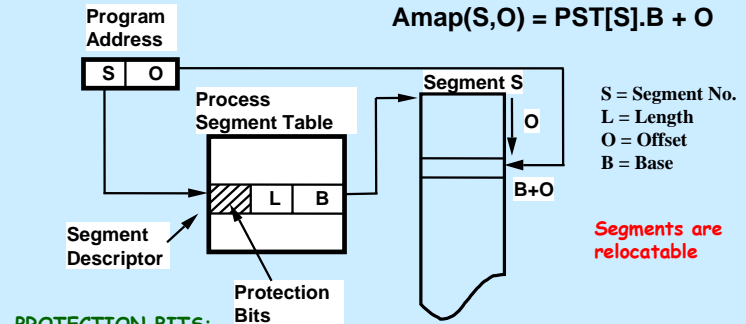


Program sees a *two-dimensional* virtual address space (Segment, Offset)

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.

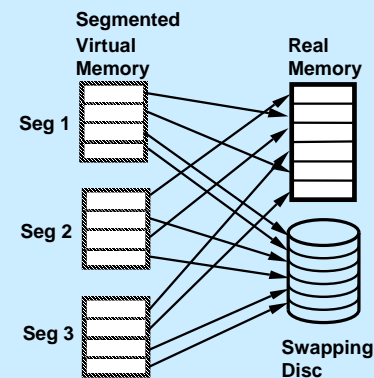
In practice, it is often not possible to fit all segments for a program into main memory at once (virtual memory > real memory), so -

Swap Segments or Use Paging

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.

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
⇒ 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

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*

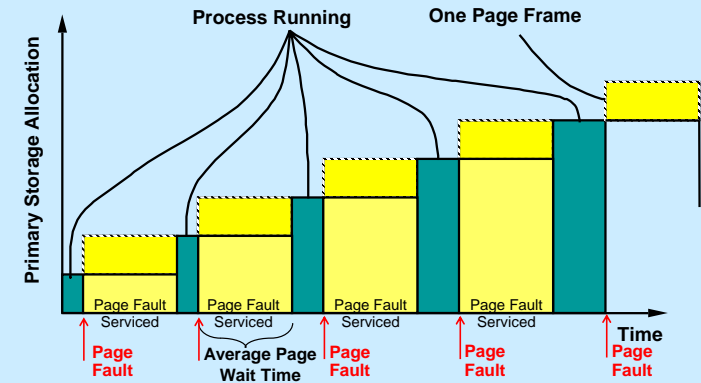
ANTICIPATORY

- 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.

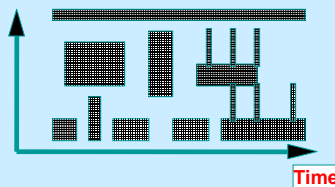
Demand Paging



Principle of Locality of Reference

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

Storage Addresses



Temporal Locality:

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

Spatial Locality:

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

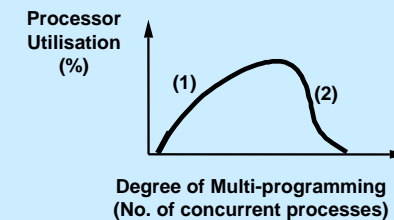
Thrashing

(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 ready process.

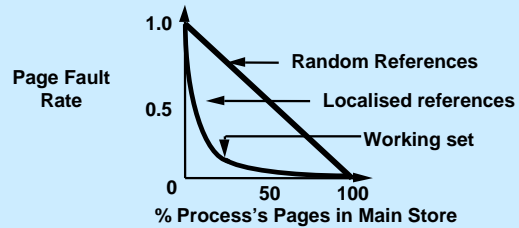
(2) After a threshold, the degree of multi-programming makes it impossible to keep sufficient pages in store for a process to avoid generating a large number of page faults. This causes:

- 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.

Working Set



☞ Only a subset of the program's code and data (*working set*) is referenced over a given time interval.

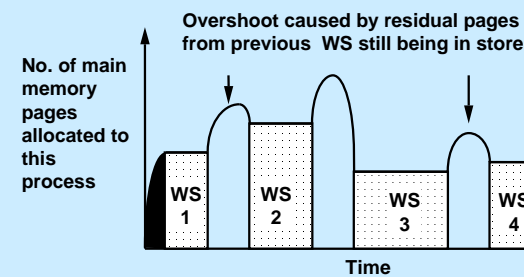
☞ To avoid thrashing, each process requires a minimum set of pages in store.
Minimum set = *Working Set*.

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.

$WS(t, w)$ = set of pages referenced in interval $\{t-w, t\}$, w = window

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



Notes:

- 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.

Determining the Working Set

Use reference bits which are set by H/W when a page is referenced.

e.g. Assume window $2t$ secs.

- Generate interrupt every t ms
- copy and clear reference bits for every page on interrupt
- maintain history of 2 bits for each page.

On page fault, check reference bits and 2 history bits. If any are set, page was accessed within past $2t-3t$ ms.

Page with at least 1 bit set considered to be in WS.

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.

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 - on average, half of last page of a segment is unused. 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 to bring in whole segment, so reduce overheads of handling page faults.

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

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