Operating Systems: Memory

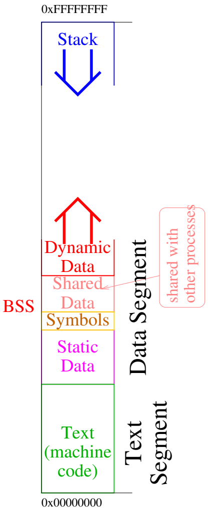
CIS*3110: Operating Systems

Dr. A. Hamilton-Wright

School of Computer Science
University of Guelph

2024-12-01

Unix-Style Process Image

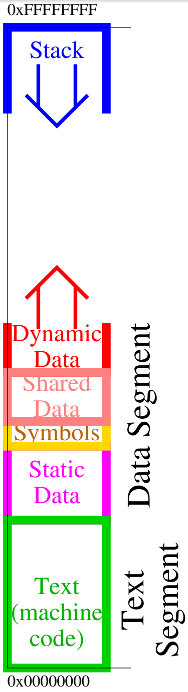


text – contains executable code

data – contains *static* data

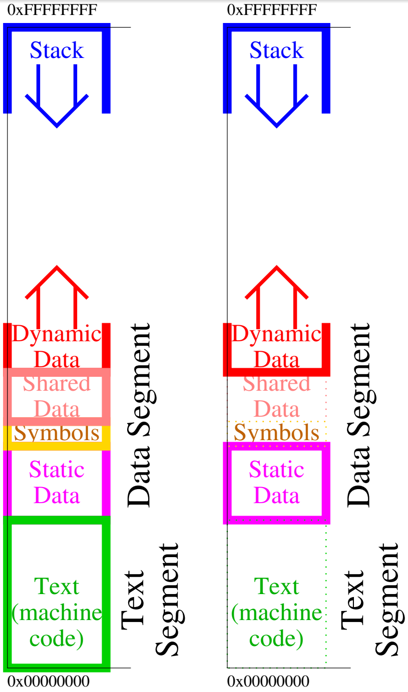
symbol table – symbol (literal) definitions
– strings, constants

Process Image: Unique process image “contains” all data



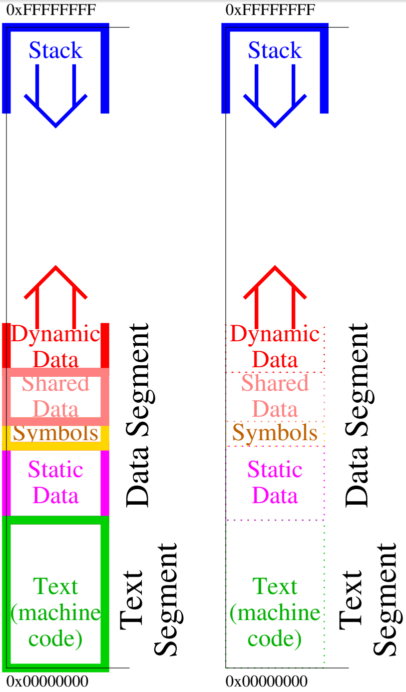
Program may clone itself using fork():

- most data **copied**
- **shared** (pointer) to shared libs, shared mem, readonly portions (text, symbols)

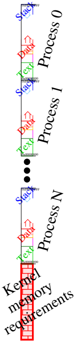


If a new POSIX **thread** is created:

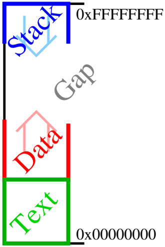
- most data is **shared**
- *only the stack* is copied → only **local variables** may be used without potential race conditions
- different thread implementation give different things
- **fibres** are threads that use **co-operative multitasking** instead of **preemptive multitasking**



Protected Virtual Addressing

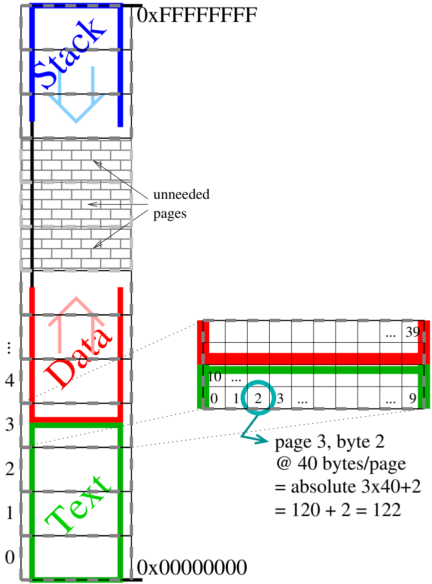


- program sees only **virtual addresses**
- kernel sees/manages <u>all</u> address spaces
- one program cannot ‘see’ or affect any other programs in memory
- large space requirements to store many programs
- program may largely consist of ‘empty space’ between stack and data
- each program is divided into **pages**; *only the pages currently needed* are represented in (or **loaded** into) in real memory



Process image and logical pages

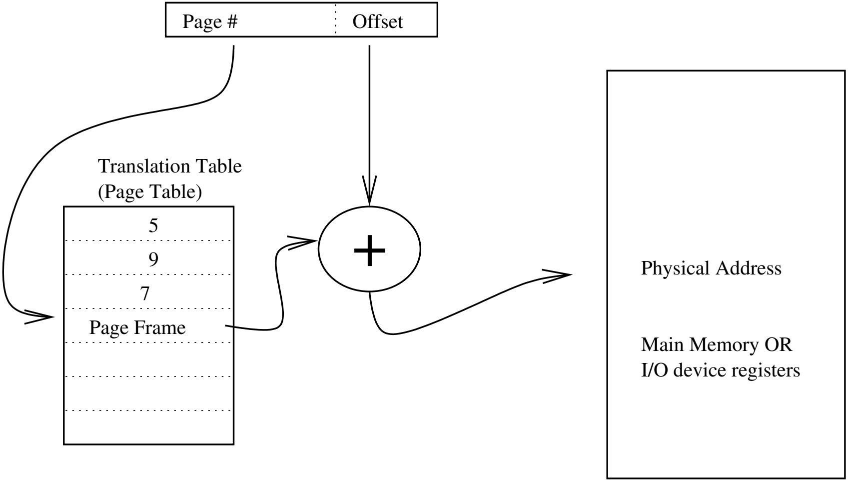
- process image is "sliced" into **pages**
- independent of data arrangement on pages
- "page + offset" $\rightarrow$ logical position of byte
- physical storage $\leftrightarrow$ logical?



MMU (Memory Management Unit)

Translates virtual $\Rightarrow$ physical addresses

A virtual address:



MMU - Address Translation

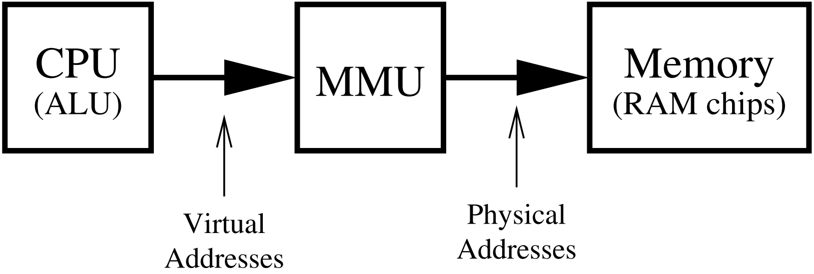
Processes are divided into **pages** (analogies: book, also slice of a loaf of bread)

$$\begin{aligned} \text{VirtualAddress}/\text{PageSize} &= \text{VirtualPageNumber} \\ \text{VirtualAddress}\%\text{PageSize} &= \text{PageOffset} \end{aligned}$$

Use virtual page number to look up the **frame number** for the **page number** in the **page table**.

$$(\text{FrameNumber} \times \text{PageSize}) + \text{Offset} = \text{PhysicalAddress}$$

Relationship between CPU and Memory



Valid and Invalid regions of Memory

- the *hardware page table* contains the following information for each *page* within a process
  - whether the page is “valid”, meaning “in memory” (called the `V` bit)
  - writeability: usually called `RO` or $\overline{\texttt{W}}$
  - dirty and used bits – more on this shortly
  - *page frame number*
- additionally, the in-memory page table contains an associated disk address if *paged out*

- the *valid* bit indicates whether page is currently in memory; the bottom of the *stack* and top of the *heap* indicate whether the page is legally part of the process

Example #1:

- Given a **page size** of 512_{10} bytes = $???_{16}$
- Read virtual address 0x072E (for process 1)
- $0x200 = 2^9 \Rightarrow 9$ bits of offset
- 0x072E / 0x200 = 0x3
- (0x12E remainder)

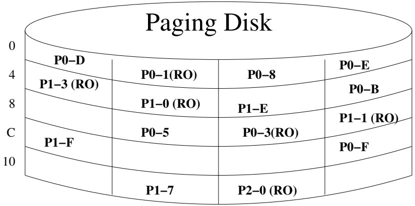
- page 0x3 $\Rightarrow$ frame 0xD
- 0xD $\times$ 0x200 = 0x1A00 + 0x12E
- = physical address 0x1B2E

Page Table

	V	RO	D	U	Page Frame	Disk Addr
F	0	0				C
E	1	0			11	6
D	0					
C	0					
B	0					
A	0					
9	0					
8	0					
7	0	0				11
6	1	0			1	
5	1	0			5	
4	1	0			10	
3	1	1			D	4
2	1	1			4	
1	0	1				B
0	0	1				5

Main Memory

0x16		P0–2 (RO)
0x14	P0–6	P2–0
0x12		P4–2
0x10	P1–4	P1–E
0x0E	PA–0	
0x0C	P0–A	P1–3 (RO)
0x0A	P5–1	PA–A
0x08	P0–C	P2–1
0x06	P0–4	P5–F
0x04	P1–2 (RO)	P1–5
0x02	P5–4	P5–0
0x00	P0–1 (RO)	P1–6



Example #2:

- Given a **page offset** of 11 bits
  - 11 wires of bus used for offset; $2^{11} = $ 0x800
- Read virtual address 0x77FF (for process 1)
- 0x77FF / 0x800 = 0xE
- (0x7FF remainder)

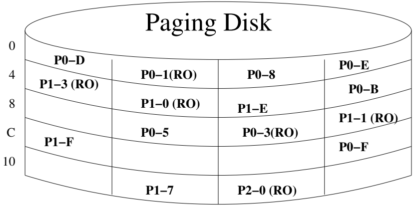
- page 0xE $\Rightarrow$ frame 0x11
- = physical address 0x8FFF

Page Table

	V	RO	D	U	Page Frame	Disk Addr
F	0	0				C
E	1	0			11	6
D	0					
C	0					
B	0					
A	0					
9	0					
8	0					
7	0	0				11
6	1	0			1	
5	1	0			5	
4	1	0			10	
3	1	1			D	4
2	1	1			4	
1	0	1				B
0	0	1				5

Main Memory

0x16		P0–2 (RO)
0x14	P0–6	P2–0
0x12		P4–2
0x10	P1–4	P1–E
0x0E	PA–0	
0x0C	P0–A	P1–3 (RO)
0x0A	P5–1	PA–A
0x08	P0–C	P2–1
0x06	P0–4	P5–F
0x04	P1–2 (RO)	P1–5
0x02	P5–4	P5–0
0x00	P0–1 (RO)	P1–6



Example #3:

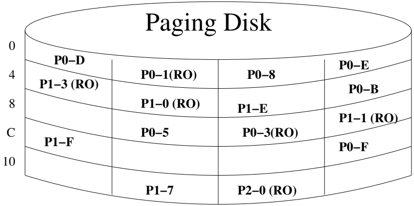
- Given a **page offset** of 11 bits (11 wires of bus used for off-set)
- 2^{11} = 0x800
- Read virtual address 0x77FF (for process 1)
- 0x77FF / 0x800 = 0x0E
- (0x7FF remainder)
- page 0xE $\Rightarrow$ frame 0x11
- 0x11 $\times$ 0x800 = 0x8800 + 0x7FF
- = physical address 0x8FFF

Page Table

	V	RO	D	U	Page Frame	Disk Addr
F	0	0				C
E	1	0			11	6
D	0					
C	0					
B	0					
A	0					
9	0					
8	0					
7	0	0				11
6	1	0			1	
5	1	0			5	
4	1	0			10	
3	1	1			D	4
2	1	1			4	
1	0	1				B
0	0	1				5

Main Memory

0x16		**P0–2 (RO)**
0x14	**P0–6**	**P2–0**
0x12		**P4–2**
0x10	**P1–4**	**P1–E**
0x0E	**PA–0**	
0x0C	**P0–A**	**P1–3 (RO)**
0x0A	**P5–1**	**PA–A**
0x08	**P0–C**	**P2–1**
0x06	**P0–4**	**P5–F**
0x04	**P1–2 (RO)**	**P1–5**
0x02	**P5–4**	**P5–0**
0x00	**P0–1 (RO)**	**P1–6**



Example #4:

- Given a **page offset** of 11 bits
- Read virtual address 0x072E (for process 1)
- 0x072E / 0x800 = 0
- (72E remainder)
- page 0x0 ⇒ page not in memory

- **page fault**
- page *is* on the paging disk
- we schedule a **read** to place the page in an empty **frame** (let's say 0x16)
- when I/O completes, we restart MMU access

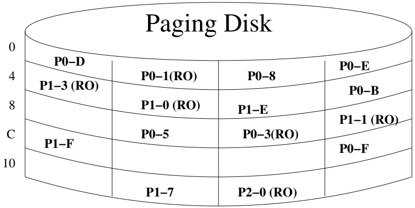
...

Page Table

	V	RO	D	U	Page Frame	Disk Addr
F	0	0				C
E	1	0			11	6
D	0					
C	0					
B	0					
A	0					
9	0					
8	0					
7	0	0				11
6	1	0			1	
5	1	0			5	
4	1	0			10	
3	1	1			D	4
2	1	1			4	
1	0	1				B
0	0	1				5

Main Memory

0x16		P0–2 (RO)
0x14	P0–6	P2–0
0x12		P4–2
0x10	P1–4	P1–E
0x0E	PA–0	
0x0C	P0–A	P1–3 (RO)
0x0A	P5–1	PA–A
0x08	P0–C	P2–1
0x06	P0–4	P5–F
0x04	P1–2 (RO)	P1–5
0x02	P5–4	P5–0
0x00	P0–1 (RO)	P1–6



Example #4b:

- Given a **page offset** of 11 bits
- Read virtual address 0x072E (for process 1)
- 0x072E / 0x800 = 0
- (72E remainder)
- page 0x0 (now in memory)

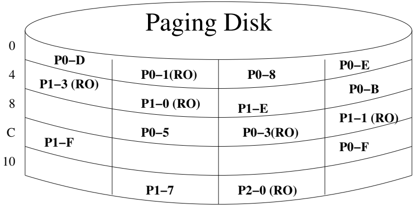
- page 0x0 $\Rightarrow$ frame 0x16
- 0x16 $\times$ 0x800 = 0xB800 + 0x72E
- = physical address 0xB72E

Page Table

	V	RO	D	U	Page Frame	Disk Addr
F	0	0				C
E	1	0			11	6
D	0					
C	0					
B	0					
A	0					
9	0					
8	0					
7	0	0				11
6	1	0			1	
5	1	0			5	
4	1	0			10	
3	1	1			D	4
2	1	1			4	
1	0	1				B
0	*1*	1			*16*	5

Main Memory

0x16	*P1–0 (RO)*	P0–2 (RO)
0x14	P0–6	P2–0
0x12		P4–2
0x10	P1–4	P1–E
0x0E	PA–0	
0x0C	P0–A	P1–3 (RO)
0x0A	P5–1	PA–A
0x08	P0–C	P2–1
0x06	P0–4	P5–F
0x04	P1–2 (RO)	P1–5
0x02	P5–4	P5–0
0x00	P0–1 (RO)	P1–6



Page Fault Handling

1. **IF** address invalid $\Rightarrow$ *terminate process*
2. **ELSE IF** page frame in process allocation is empty $\Rightarrow$ use this frame
3. **ELSE** choose page to be replaced
   - **IF** page is modified $\Rightarrow$ *save page on disk*
4. **IF** required page is saved on **paging area** $\Rightarrow$ *read in from disk paging area*
5. **ELSE IF** required page is a **text page** or an **initialized data page** $\Rightarrow$ *read in from executable program file*
6. **ELSE** initialize page to zeros (for security)
7. set up **page table** to point to the new page
8. mark process **RUNNABLE**

Page Fault Timeline

If address mapping in process $\mathcal{X}$ causes **page fault**:

- process $\mathcal{X}$ cannot be run – **BLOCKED** on I/O
  - waiting for needed page to be loaded into page frame
  - \+ possibly other steps
- other processes ($\mathcal{Y}, \mathcal{Z} \ldots$) *may* be run if **RUNNABLE**
- if no **RUNNABLE** process, system is **idle**
  - all processes are waiting on I/O
- (no matter what), fault causes major interruption in runtime of process $\mathcal{X}$
  - equivalent to **thousands** of instructions

$\therefore$ worthwhile to spend some processor effort trying to *manage* and *decrease* the number of page faults

Paged Virtual Memory

Paged Virtual Memory depends on the **locality principle**:

> *a large program only uses a small portion of its virtual address space at a given time.*

The set of pages that make up this portion is called the **working set**

The pages that are allocated page frames in physical memory is called the **resident set**.

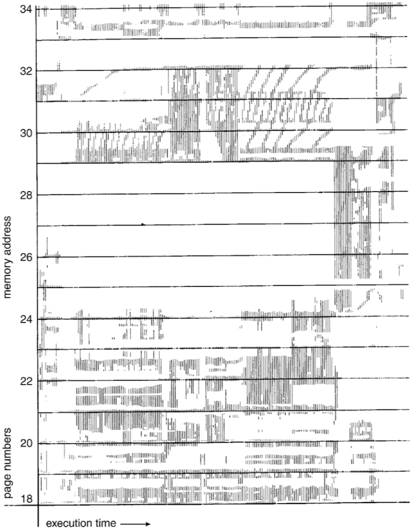


Figure 9.19 Locality in a memory-reference pattern.

*

*Silberschatz *et al*, *Operating System Concepts*, 8th ed. John Wiley & Sons. Fig 9.19, pp. 388.

Page Replacement Policy

A **page replacement policy** decides which page frames to replace

The ideal page replacement policy would achieve:

$$\textbf{resident set} \equiv \textbf{working set}$$

To *evaluate* a page replacement policy, you must calculate its **page fault rate** for the **page reference strings** of **real programs**.

Page Replacement Policies

Belady's Optimal Replacement replaces the page that occurs **furthest ahead** in the reference string

FIFO replace the page resident the longest

LFU (Least Frequently Used) replace the page that has been referenced the **fewest times**

LRU (Least Recently Used) replace the page that whose **last reference** is the **furthest in the past**

Use Bit

A **use bit** is a hardware bit that is set when a page is referenced, which is reset by a **software process**.
If the bit is **clear**, this means that the (page) has not been referenced since the software cleared it.
The bit will be **set to 1** when the hardware accesses the (page).

The page replacement algorithm becomes:

- at **regular** time intervals, clear **all reference bits**
- replace a page that has a **clear** (*i.e.*; unset, 0) reference bit.

This is a **crude approximation of LRU**.
Similar to the **dirty bit**, which is set whenever a page **is modified**.

Page Replacement Example #1: FIFO

If 64-bit machine with 64 kilobytes of RAM has a page size of 512 bytes, and the following page reference string is observed for a running program where $^{\triangledown}$ indicates the use bit has been cleared:

$$^{\triangledown} A\ B\ C\ D\ \ A\ B\ E\ ^{\triangledown} A\ B\ C\ D\ \ E\ B\ C\ F\ B\ B\ ^{\triangledown} B\ C\ D\ E\ \ F$$

$\mathcal{Q}$: How many page faults would occur using FIFO with 3 page frames?

FIFO(3) = 15

Frame 0	Frame 1	Frame 2
~~A~~	~~B~~	~~C~~
~~D~~	~~A~~	~~B~~
~~E~~	~~C~~	~~D~~
~~B~~	~~F~~	~~C~~
D	E	F

Page Replacement Example #2: Belady's

If 64-bit machine with 64 kilobytes of RAM has a page size of 512 bytes, and the following page reference string is observed for a running program where $^{\triangledown}$ indicates the use bit has been cleared:

$$^{\triangledown}A\ B\ C\ D\ \ A\ B\ E^{\triangledown}\ A\ B\ C\ D\ \ E\ B\ C\ F\ B\ B^{\triangledown}\ B\ C\ D\ E\ \ F$$

$\mathcal{Q}$: How many page faults would occur using Belady's Optimal Algorithm with 3 page frames and falling back to FIFO to break any ties?

Belady's(3) = 11

Frame 0	Frame 1	Frame 2
~~A~~	~~B~~	~~C~~
~~C~~	D	~~D~~
~~D~~		~~E~~
F		~~C~~
		E

If 64-bit machine with 64 kilobytes of RAM has a page size of 512 bytes, and the following page reference string is observed for a running program where $^\triangledown$ indicates the use bit has been cleared:

$$^\triangledown A\ B\ C\ D\ \ A\ B\ E\ ^\triangledown A\ B\ C\ D\ \ E\ B\ C\ F\ B\ B\ ^\triangledown B\ C\ D\ E\ \ F$$

$\mathcal{Q}$: How many page faults would occur using Belady's Optimal Algorithm with **4** *page frames and falling back to FIFO to break any ties?*

Belady's(4) = 8

F0	F1	F2	F3
~~A~~	~~B~~	C	~~D~~
D	E		~~E~~
			F

Page Replacement Example #4: LFU

$^{\triangledown}$ *A B C D A B E* $^{\triangledown}$ *A B C D E B C F B B* $^{\triangledown}$ *B C D E F*

$\mathcal{Q}$: How many page faults would occur using LFU with 4 page frames and falling back to FIFO to break any ties?

	F0	**F1**	**F2**	**F3**
counts:	. . .	. . . .	. .	.
	A	B	~~C~~	~~D~~
			~~E~~	~~C~~
LFU(4) = 13			~~D~~	~~E~~
			C	~~F~~
				~~D~~
				~~E~~
				F

$^{\triangledown}$ *A B C D A B E* $^{\triangledown}$ *A B C D E B C F B B* $^{\triangledown}$ *B C D E F*

Q: How many page faults would occur using LRU with 3 page frames and falling back to FIFO to break any ties?

LRU(4) = 12

F0	F1	F2	F3
~~A~~	~~B~~	~~C~~	~~D~~
~~E~~	F	~~E~~	C
D		~~D~~	
		~~F~~	
		E	

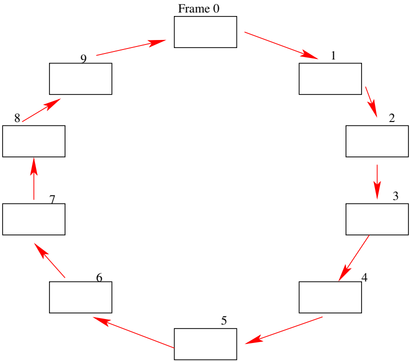
Clock Algorithm

Simple:

- read+clear use bits
- use first clear page found

Enhanced:

- (use, dirty)
  - (0,0) – best
  - (0,1) – write to store
  - (1,0) – probably needed
  - (1,1) – worst
- use first page found in lowest non-empty class
- may require several sweeps



Global versus Local Paging Policies

Selection for replacement:

- a **global page replacement policy** (*a.k.a.* a **paging policy**) is applied to all pages

versus

- a **local page replacement policy** is applied to pages for that process only

Allocation:

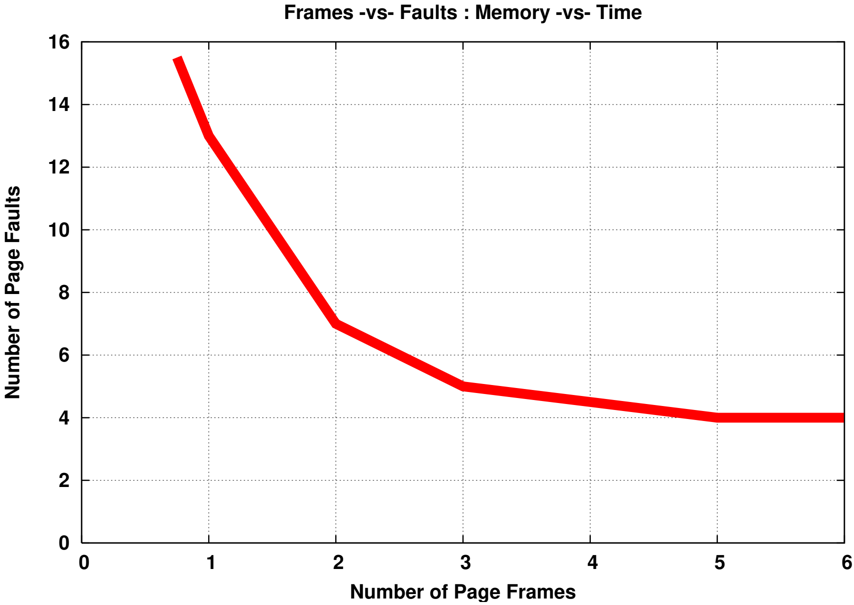
- a **fixed size partition policy** uses a fixed frame allocation for each process
- a **variable partition policy** varies the frame allocation for each process.

Variable Partition Policies

An algorithm may adjust the page **frame allocation** based on the observed **page fault rate**.

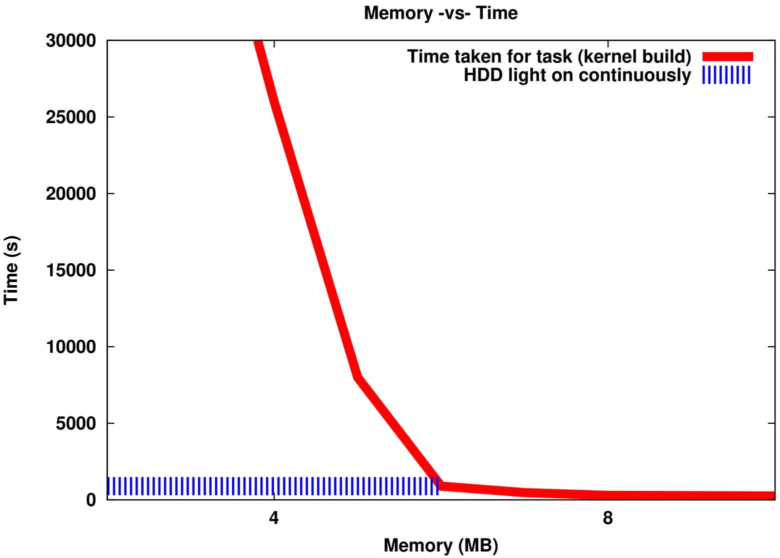
- **IF** fault rate is **High** for a process,
  - increase frame allocation by **quite a bit** ***
- **ELSE IF** fault rate is **Low**,
  - decrease frame allocation a **little**

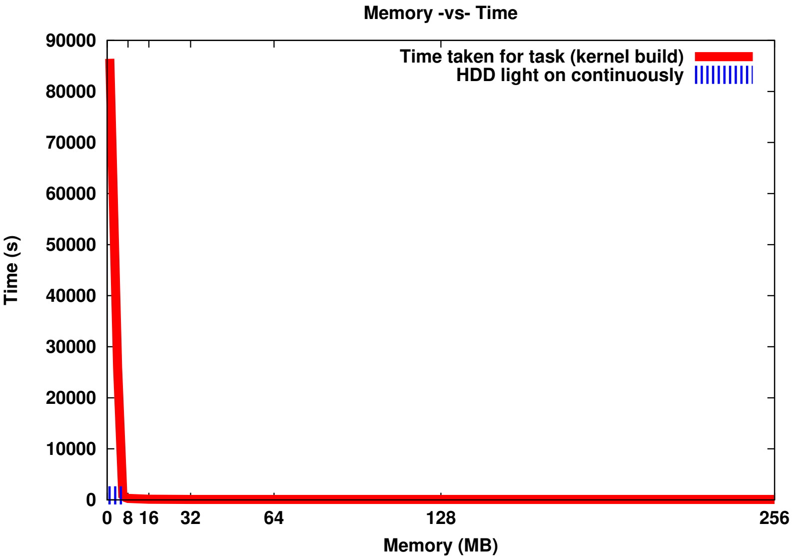
*** possibly even multiplicative or exponential instead of linear



†

†Silberschatz *et al*, *Operating System Concepts*, 8th ed. John Wiley & Sons. Fig 9.11, pp. 373.

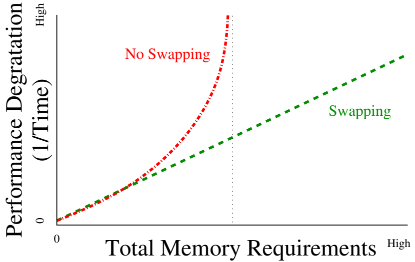

Memory -vs- Time
Time taken for task (kernel build)
HDD light on continuously
30000
25000
20000
15000
10000
5000
0
Time (s)
4
8
Memory (MB)


Memory -vs- Time
Time taken for task (kernel build)
HDD light on continuously
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Time (s)
0
8
16
32
64
128
256
Memory (MB)

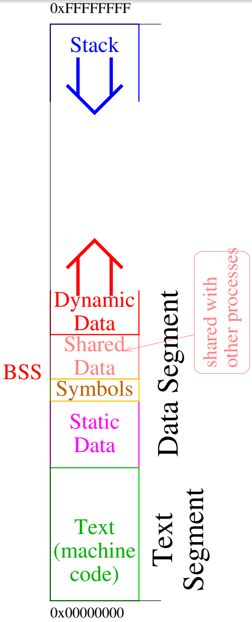
Swapping:

- transferring **page_size** blocks of memory data between memory ⇔ disk

- transferring *entire segments* of processes between memory ⇔ disk
- swapping is a *desperation* effort which attempts to provide a graceful (i.e.; linear) degradation of performance



Unix-Style Process Virtual Address Space



- for a **read-write** page, a separate copy of each page must be kept for each process, however the duplication can be delayed until a process modifies each page (this is known as **copy on write**), *unless it is a shared segment*
- for a read-only segment, only 1 copy of each page must be in physical memory for *all* processes

Adjusting the Address Space

There are two segments that may change size:

Stack : adjusted through (function) `call`

Data : adjusted when `malloc()`, `new` + co. are called

kernel routines to do this:

`brk(void *endp)` sets end of data segment to absolute (virtual) address `endp`

`sbrk(int incr)` increments data segment size by `incr` bytes (decrements if `incr` negative)

(Version 7 AT&T Unix – not POSIX, nor ANSI)

Dynamic Allocation (Segmentation)

There are several algorithms popular for (re-)allocation:

First Fit use first space found which is large enough

$$\text{space} - \text{size} \geq 0$$

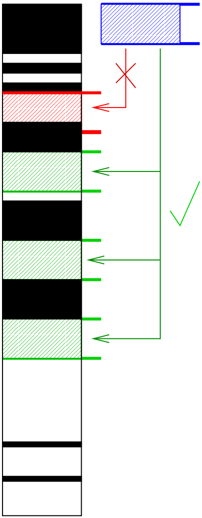
Best Fit use space found whose leftover space is **smallest**

$$\min \text{space} - \text{size} \geq 0$$

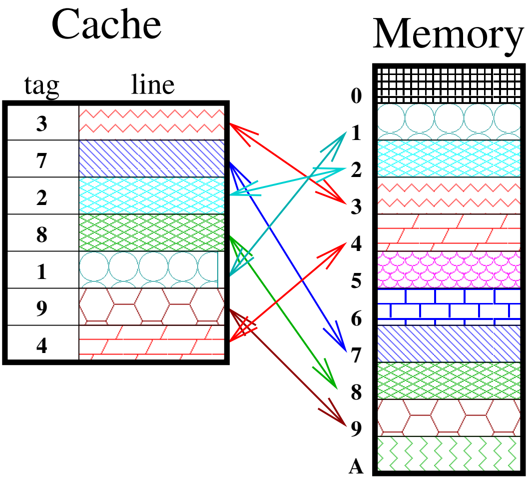
Worst Fit use space found whose leftover space is **largest**

$$\max \text{space} - \text{size} \geq 0$$

Knuth Buddy System complicated and unpopular, but used in Linux

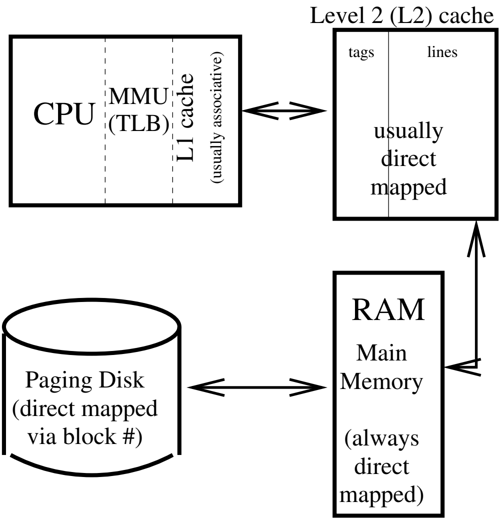


Memory Caches



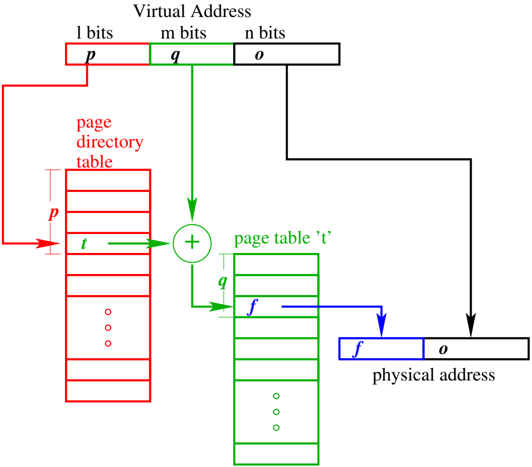
- similar to a paging system implemented entirely in hardware memory
- access to lines **3, 7, 2, 8, 1, 9, 4** $\Rightarrow$ **cache hits**
- access to lines **5, 6, 10** $\Rightarrow$ **cache miss**
- the **tags** are the page table entries and the **lines** are the pages

Memory Hierarchy



- within the CPU, the Level-1 (L1) cache records a small number of recent cache lines, supplied from the L2 cache
- in general, the **further** from the CPU storage is, the **larger**, **slower** and **cheaper** it is.

Hierarchical Page Table



- page table itself is large
- each 2nd order page table is 1 page in size
- allows us to page out parts of the page table we are not using