Memory Management

Anandha Gopalan
(with thanks to R. Kolcun and P. Pietzuch)
axgopala@imperial.ac.uk

Imperial College London

Basic Concepts

- Memory Allocation
- Swapping

Virtual Memory

Paging & Segmentation

- Demand Paging
- Page replacement algorithms
- Working set model

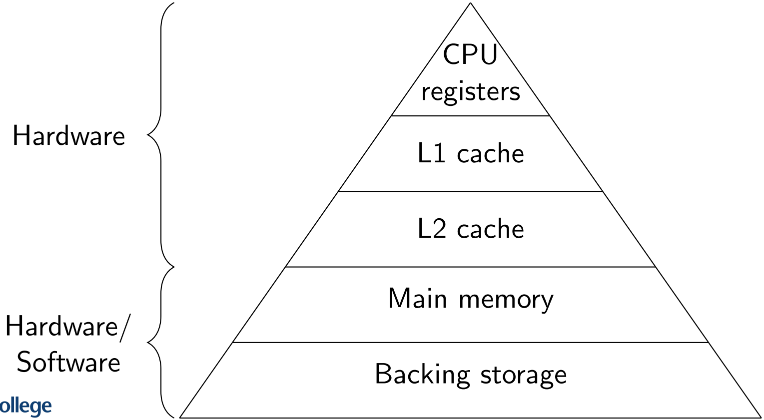
Linux Memory Management

Memory Hierarchy

Hardware: CPU registers and main memory

- Register access in one CPU clock cycle (or less)
- Main memory can take many cycles
- Caches sit between main memory and CPU registers

Managed by

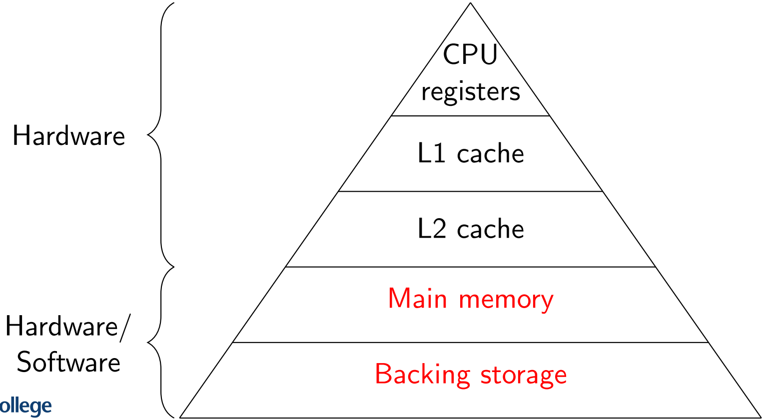


Memory Hierarchy

Hardware: CPU registers and main memory

- Register access in one CPU clock cycle (or less)
- Main memory can take many cycles
- Caches sit between main memory and CPU registers

Managed by



Memory Management

Memory is a key component of the computer

- e.g. every instruction cycle involves memory access $\Rightarrow$ process has to be loaded into memory before it can execute

Memory management needs to provide

- Memory allocation
- Memory protection

Characteristics

- No knowledge of how memory addresses are generated
  - e.g. instruction counter, indexing, indirection, ...
- No knowledge what memory addresses are used for
  - e.g. instructions or data
- True for simple case but may want protection with respect to read, write, execute, etc.

Logical vs. Physical Address Space

Memory management binds logical address space to physical address space

Logical address

- Generated by the CPU
- Address space seen by the process

Physical address

- Address seen by the memory unit
- Refers to physical system memory

Logical and physical addresses

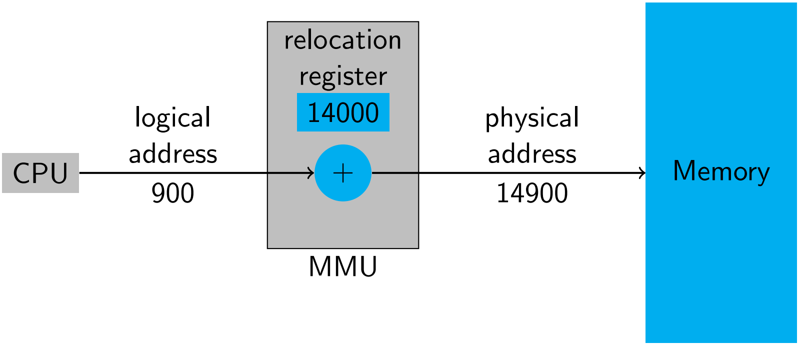
- Same in compile- and load-time address-binding schemes
- Different in execution-time address-binding schemes

How do you achieve this mapping?

Memory-Management Unit (MMU)

Hardware device for mapping logical to physical addresses

- e.g. add value in relocation register to every address generated by process when sent to memory
- User process deals with logical addresses only
- Has to be fast → implemented in hardware



Memory Allocation

Main memory is usually split into two partitions:

- Resident operating system (**kernel**)
  - Usually held in low memory with interrupt vector
- User processes (**user**)
  - Held in high memory

How do you decide where to load a new process?

Need to figure out the strategy for process to be loaded into the correct location

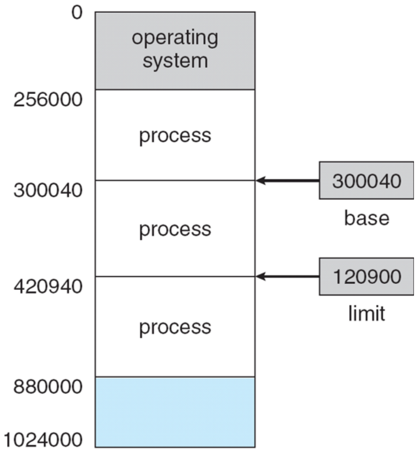
Contiguous Memory Allocation I

Contiguous allocation with relocation registers

- base register contains physical start address for process
- limit register contains maximum logical address for process
- MMU maps logical address dynamically
  - Physical address $=$ logical address $+$ base
  - If logical address $>$ limit then error

Contiguous Memory Allocation II

base and limit register define logical address space



e.g jmp 100 in program would go to physical location 300140

Hole

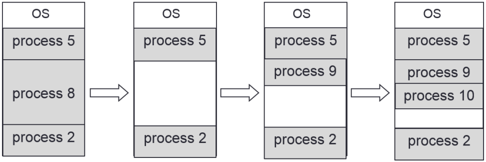
- Block of available memory
- Holes of various size scattered throughout memory

When new process arrives:

- allocate memory from hole large enough

OS maintains information about:

- Allocated partitions
- Free partitions (holes)



What is the best algorithm for allocation?

Dynamic Memory Allocation

First-fit $\rightarrow$ Allocate first hole that is big enough

Best-fit $\rightarrow$ Allocate smallest hole that is big enough

- Must search entire list, unless ordered by size
- Produces smallest leftover hole

Worst-fit $\rightarrow$ Allocate largest hole

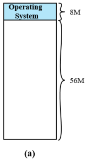
- Must also search entire list
- Produces largest leftover hole

Why best-fit or worst-fit?

First-fit and best-fit better than worst-fit in terms of speed and storage utilisation

Fragmentation

External fragmentation $\rightarrow$ memory exists to satisfy request, but not contiguous

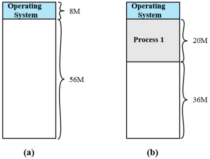


Reduce external fragmentation by compaction

- Shuffle memory contents to place all free memory together in one large block $\rightarrow$ leads to I/O bottlenecks

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous

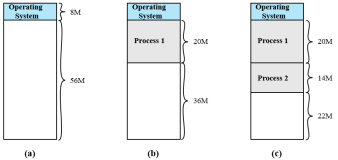


Reduce external fragmentation by compaction

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous

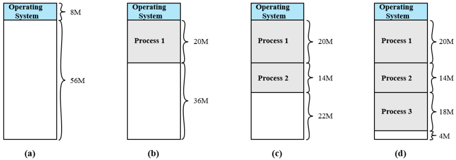


Reduce external fragmentation by <u>compaction</u>

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous

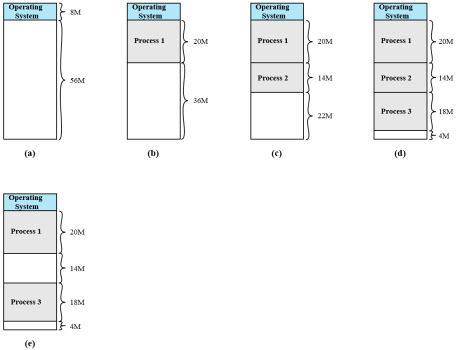


Reduce external fragmentation by compaction

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous

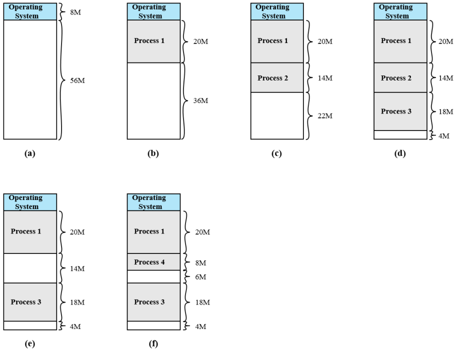


Reduce external fragmentation by <u>compaction</u>

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous



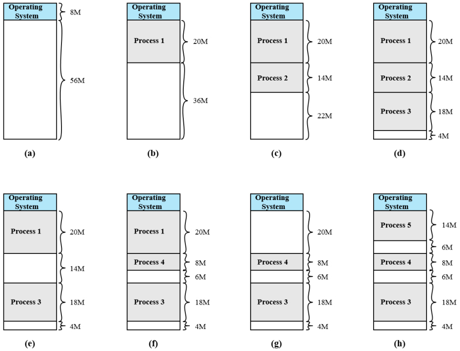
Reduce external fragmentation by compaction

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Imperial College London

Fragmentation

External fragmentation → memory exists to satisfy request, but not contiguous



Reduce external fragmentation by <u>compaction</u>

- Shuffle memory contents to place all free memory together in one large block → leads to I/O bottlenecks

Swapping

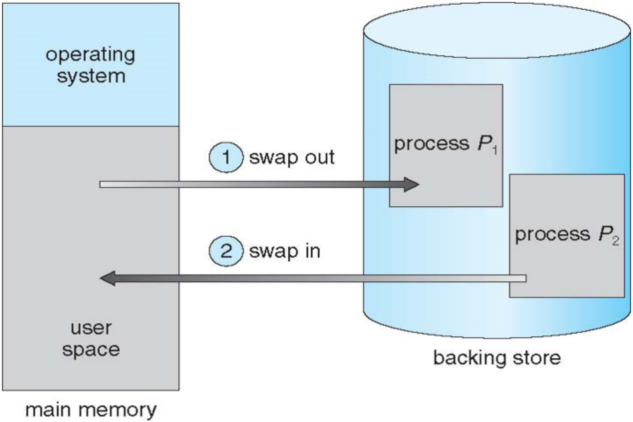
Problem: Number of processes limited by amount of available memory

- But ... only running processes need to be in memory

Solution:

- **Swap** processes temporarily out of memory to backing store
- Bring back into memory for continued execution
- Requires swap space → can be file or dedicated partition on disk
- Transfer time is major part of swap time

Swapping



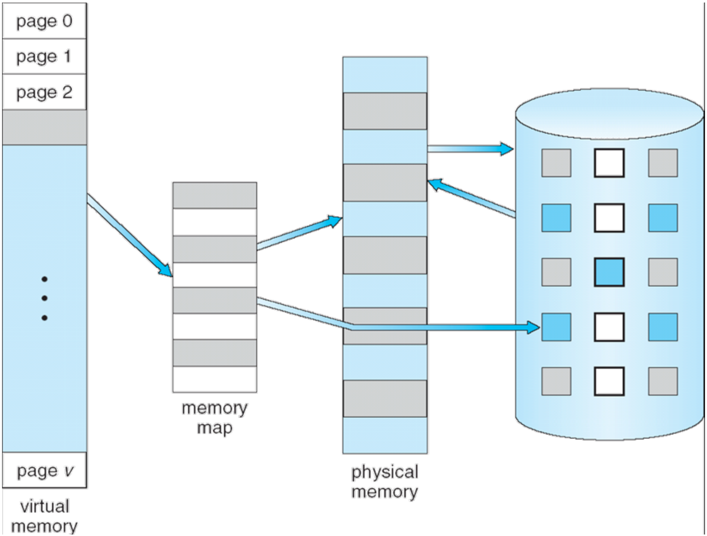
What if a process is "too large" to fit into memory ⇒ can only part of a process exist in memory?

Virtual Memory

Separation of user logical memory from physical memory

- Only part of process needs to be in memory for execution
- Logical address space can be <u>much</u> larger than physical address space
- Address spaces can be shared by several processes
- Allows for more efficient process creation

Virtual Memory



Virtual memory can be implemented via

- Paging
- Segmentation

Paging

Physical address space of process can be noncontiguous

- Process allocated physical memory when available
  - Avoid external fragmentation
  - Avoid problems of variable sized memory chunks

Frames

- Fixed-sized blocks of physical memory
- Keep track of all free frames

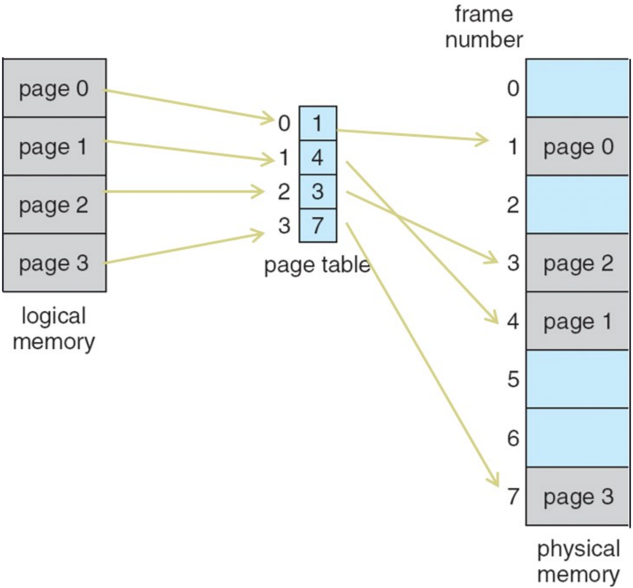
Pages

- Block of same size (as frame) of logical memory

To run program of size n pages

- Find n free frames and load program
- Set up **page table** to translate logical into physical addresses

Page Table Example



Address Translation I

How does logical address translate to physical address?

Hint: pages and frames are the same size $\Rightarrow$ address offset in the page will be the **same** as that in the frame

Address now consists of two parts: **page number** and **page offset**

- only need to translate page number into its corresponding frame address

Address Translation II

How do you calculate the page number?

Depends on address size and page/frame size

e.g. Consider a page/frame size of 64 bytes

- 64 bytes can be addressed $\Rightarrow$ total of 64 addresses
- Number of bits required for 64 addresses $= 6$ $(2^6 = 64)$

For a 10-bit **virtual** address we have:

- page offset requires 6 bits (based on above)
- page number has 4 bits (remaining bits) $\Rightarrow$ between $0 \ldots 15$

Address Translation III

Page number (p)

- Used as an index into page table
- Page table has base address of pages in physical memory

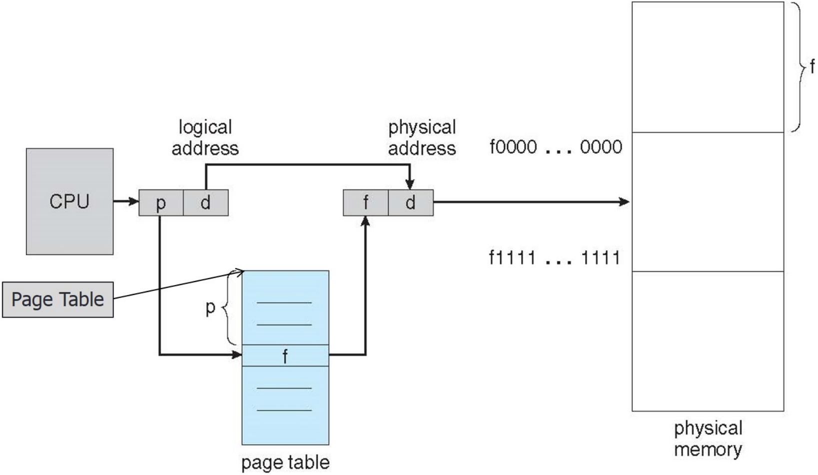
Page offset (d)

- Defines physical memory address sent to the memory unit
- Combined with base address

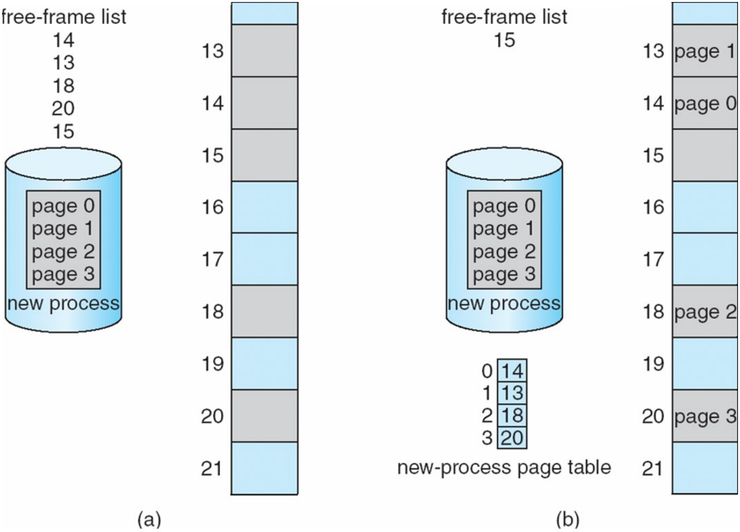
For given address size of m-bits and page size of 2^n

page number	page offset
p	d
$(m - n)$ bits	n bits

Paging Hardware



Free Frames



Before allocation

After allocation

Example Problem

Address Translation

Consider a 32-bit virtual memory address and a page size of 1 KB. How many pages can a process potentially have?

Example Problem

Address Translation

Consider a 32-bit virtual memory address and a page size of 1 KB. How many pages can a process potentially have?

1 KB page size $=$ 1024 bytes $\Rightarrow$ total of 1024 addresses
Number of bits needed for 1024 address $=$ 10 ($2^{10} = 1024$)

Example Problem

Address Translation

Consider a 32-bit virtual memory address and a page size of 1 KB. How many pages can a process potentially have?

1 KB page size $=$ 1024 bytes $\Rightarrow$ total of 1024 addresses
Number of bits needed for 1024 address $= 10$ ($2^{10} = 1024$)

For a 32-bit address you have:

- page offset requires 10 bits
- page number has 22 bits $\Rightarrow 2^{22}$ (4194304) potential pages

Fragmentation

Internal fragmentation $\rightarrow$ Allocated memory is larger than requested memory, but size difference internal to partition

Example - Calculating Internal Fragmentation

Page size $=$ 2048 bytes; Process size $=$ 72,766 bytes

Number of pages $= \frac{72766}{2048} = 35$

Bytes left-over $=$ 72766 % 2048 $=$ 1086

Internal fragmentation $=$ 2048 – 1086 $=$ 962 bytes

Worst-case fragmentation $\Rightarrow$ 1 frame $=$ 1 byte

Average-case fragmentation $\Rightarrow \frac{1}{2}$ frame size

Are small frames desirable?

- Each page table entry takes memory to track
- Page size growing over time → typically 4 KB but some architectures support variable page sizes up to 256 MB

Page Table Implementation

Page table kept in main memory

- Page-table base register (PTBR) points to page table
- Context switch requires update of PTBR for new process page table (if necessary)
- Page-table length register (PTLR) indicates size

Problem

- Inefficient, as every data/instruction access requires two memory accesses → one for page table and one for data/instruction

Associative Memory

Solution: use special fast-lookup hardware cache as associative memory

Associative memory $\rightarrow$ supports parallel search

- Called **Translation Look-aside Buffer (TLB)**

Page #	Frame #
1	4
3	74
9	50
7	7

- Address translation (p, d)
  - If p in associative register, get frame # out
  - Otherwise, get frame # from page table in memory

Translation Look-aside Buffers (TLBs) I

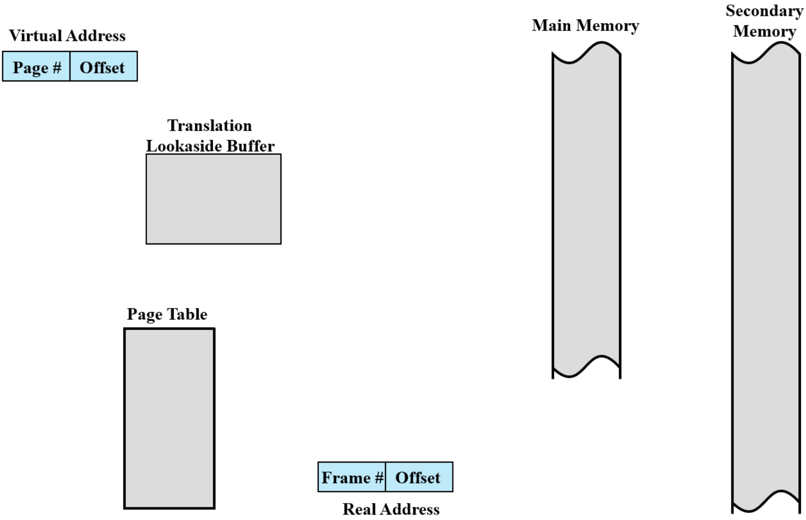
TLBs usually needs to be flushed after context switch

- Can lead to substantial overhead
- What about kernel pages for system calls?

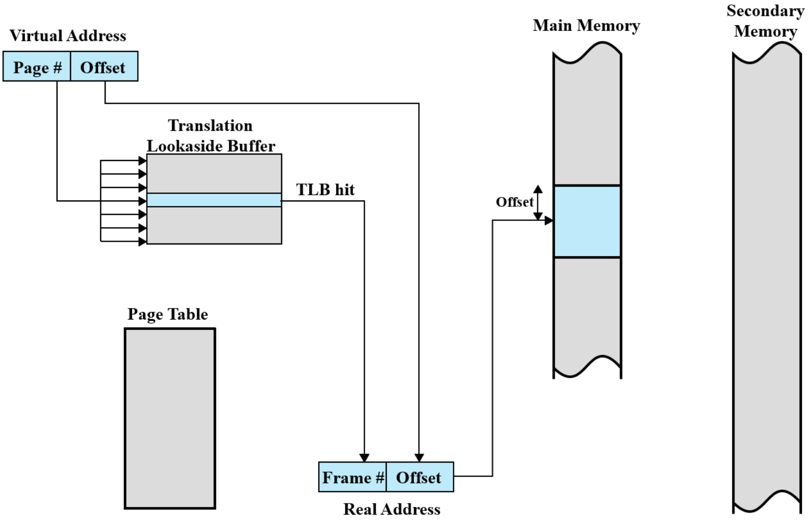
Some TLBs store address-space ids (ASIDs) in entries

- Uniquely identifies each process to provide address-space protection for that process

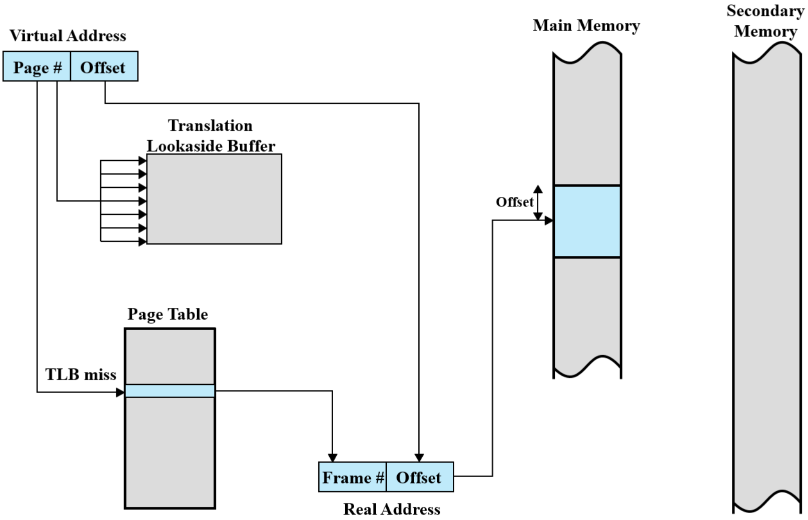
Translation Look-aside Buffers (TLBs) II



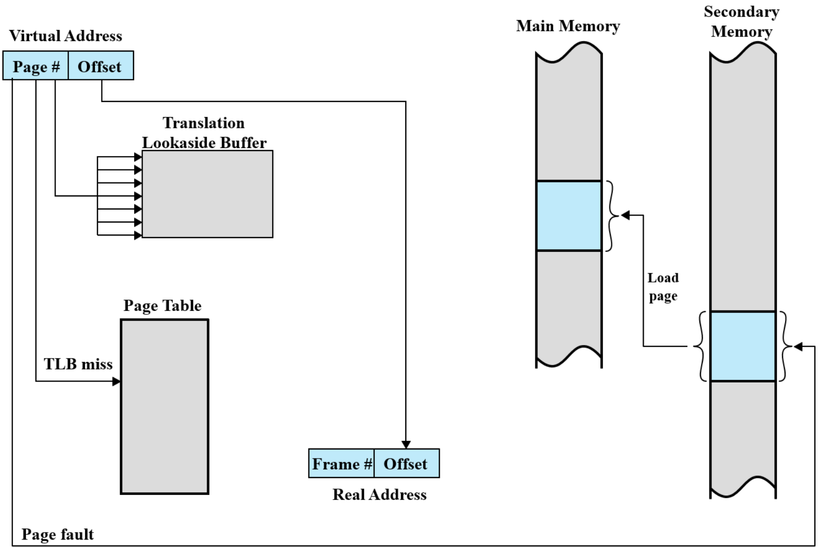
Translation Look-aside Buffers (TLBs) II



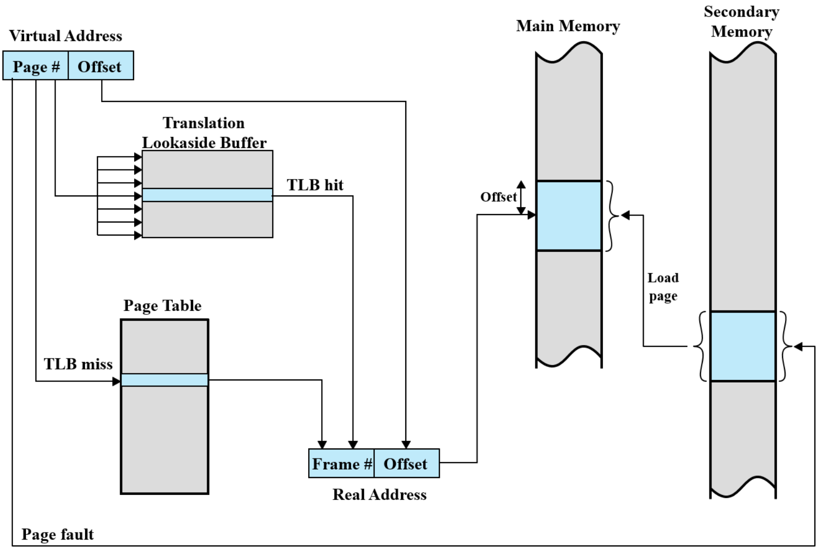
Translation Look-aside Buffers (TLBs) II



Translation Look-aside Buffers (TLBs) II



Translation Look-aside Buffers (TLBs) II



Example Problem

Effective Access Time

TLB Lookup $= \epsilon$ (can be $<$ 10% of memory access time *m*)
Hit Ratio $= \alpha$

- Fraction of times that page is found in associative registers
- Ratio related to number of associative registers

Effective Access Time (EAT) $= (\epsilon +$ m) x $\alpha + (\epsilon +$ 2m) x (1 - α)
$=$ 2m $+ \epsilon$ - mα

Consider $\alpha = 80\%$, $\epsilon = 10$ *ns* for TLB search, $m = 100$ *ns* for memory access

- EAT $=$ 110 x 0.80 $+$ 210 x 0.20 $=$ 130 *ns*

A more realistic hit ratio might be 99%

- EAT $=$ 110 x 0.99 $+$ 210 x 0.01 $=$ 111 *ns*

Page Table Size

Why do we need need to worry?

Page table can grow to be very large in size

On a 32-bit machine with a 4 KB page size:

- Number of page table entries $= \frac{2^{32}}{2^{12}} = 2^{20}$
- Size of each page table entry $=$ 32 bits
- Size of page table $= 2^{20}$ x 32 bits $=$ 4 MB

On 64-bit machine with 4 KB pages $\rightarrow$ page table needs 2^{52} entries

- with 8 bytes per entry, that's 30 million GB ...
- lot of memory to be allocated ☹

Page Table Types

Hierarchical page table

Hashed page table

Inverted page table

Hierarchical Page Table I

Idea: Let the page-table be broken-up and paged if it is too large

Simple technique → **two-level page table** for a machine with 32-bit addresses and a 4 KB page size

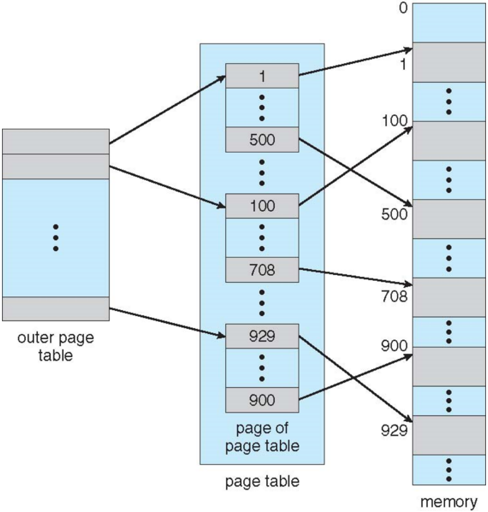
- page offset needs 12 bits
- page table size = 4 MB

How do you break the page table up?

Each part of the page table that is being paged must fit on a page

- Recall: page size = 4 KB
- Number of entries on one page = $\frac{Page\ \ size}{Address\ \ size} = \frac{4\ \ KB}{32\ \ bits} = 2^{10}$
- No of bits required for 2^{10} entries = 10
- Address bits left for top-level page table = 32 – 10 – 12 = 10

Hierarchical Page Table II



Fix outer page table in memory

Two-Level Paging I

Logical address divided

- Page number consisting of 20 bits
- Page offset consisting of 12 bits

Since page table paged, page number further divided

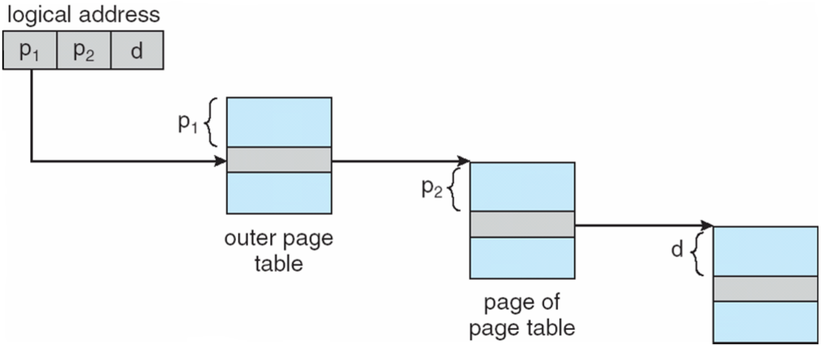
- 10 bit page number
- 10 bit page offset within 2^{nd} level page table

Thus, logical addresses as follows

page number		page offset
p_1	p_2	d
10	10	12

$p_1 \rightarrow$ index into the outer page table
$p_2 \rightarrow$ displacement within page pointed to by outer page table

Two-Level Paging II



Imperial College London

Page Table Addressing

Consider a paging system that uses a three-level page table. Virtual addresses are composed into four fields (a, b, c, d) with d being the offset. What is the maximum number of pages in a virtual address space?

Answer: 2^{a+b+c}, since there are a total of $2^{a+b+c+d}$ addresses in the address space and each page has 2^{d} addresses

Example Problem

Page Table Addressing

Consider a paging system that uses a three-level page table. Virtual addresses are composed into four fields (a, b, c, d) with d being the offset. What is the maximum number of pages in a virtual address space?

Answer: 2^{a+b+c}, since there are a total of $2^{a+b+c+d}$ addresses in the address space and each page has 2^d addresses

Page Table: Another Idea

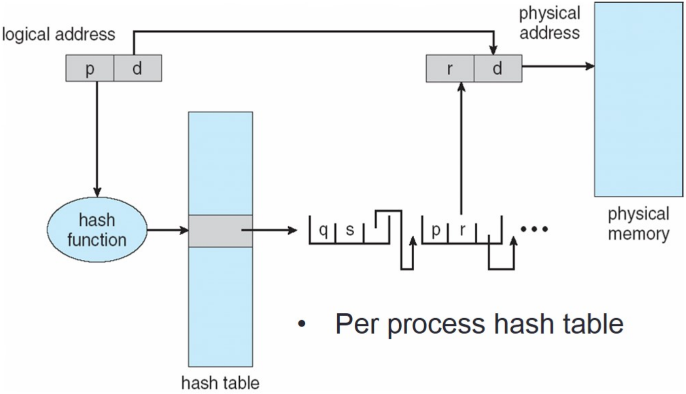
Don't store entry per page but per frame

- **Hashed page table**
- **Inverted page table**

Hashed Page Table

Hash virtual page number into page table

- Page table contains chain of elements hashing to same location
- Search for match of virtual page number in chain
- Extract corresponding physical frame if match found

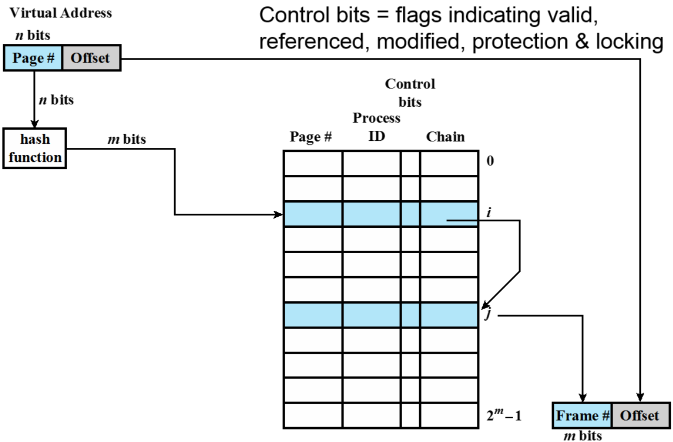


Inverted Page Table

One entry per physical frame

Decreases memory needed to store page table

- But increases time to search table when page reference occurs



Inverted Page Table (one entry for each physical memory frame)

Segmentation

Paging gives one-dimensional virtual address space $\rightarrow$ what about separate address spaces for code, data, stack?

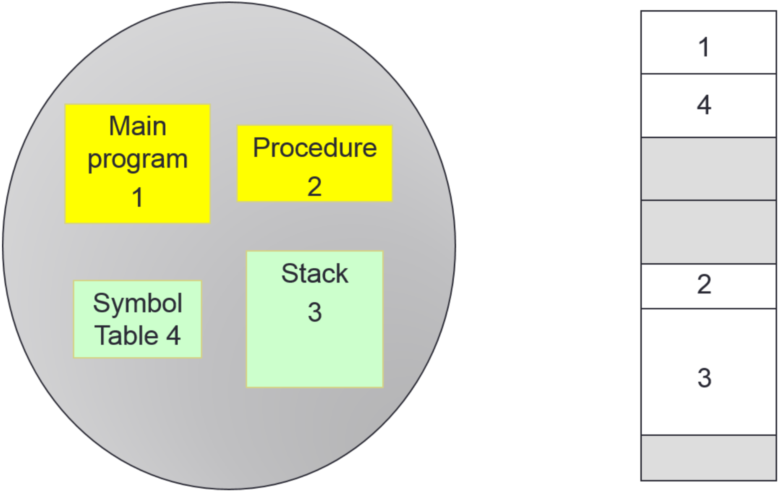
Segment

- Independent address space from 0 to some maximum
- Can grow/shrink independently
- Support different kinds of protection (read/write/execute)
- Unlike pages, programmers are aware of segments
- Segment corresponds to program, procedure, stack, object, array, etc.

Memory allocation harder due to variable size

- May need to move segment which grows
- May suffer from external fragmentation
- But good for shared libraries

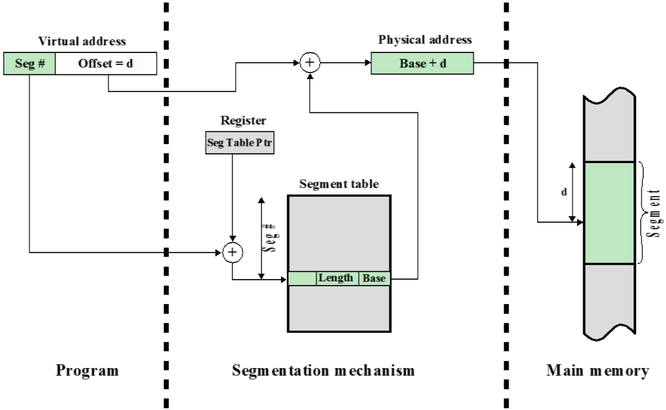
Logical View of Segmentation



User logical space

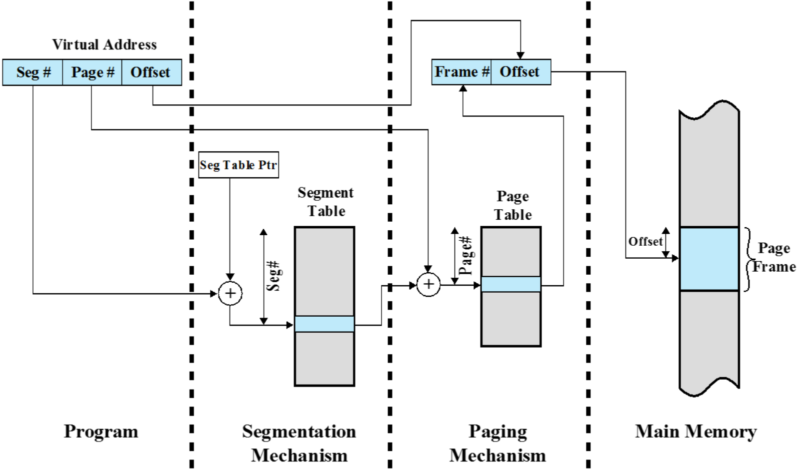
Physical memory space

Segmentation Address Translation



- One bit in table indicates whether segment is in memory
- Another bit indicates whether segment is modified

Hybrid Segmentation/Paging



Most OSs use only paging

Memory Control Bits

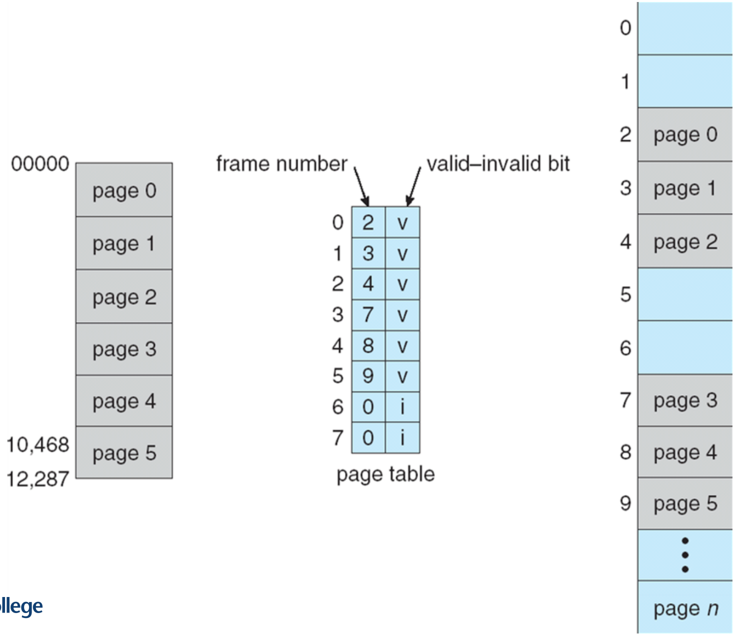
Protection bits → associated with a frame indicate read-only, read-write, execute only

Valid-invalid bit

- **Valid** → page present in physical memory
- **Invalid** → page missing in physical memory
  - Page fault is generated ⇒ kernel trap to bring in page from backing store

Page replacement bits → to indicate if page has been modified or referenced (used later). Also, lock bit to prevent page from being transferred out

Memory Validity



Demand Paging I

When do you bring the page into memory?

Bring page into memory only when needed

- Lower I/O load
- Less memory needed
- Faster response time
- Support for more users

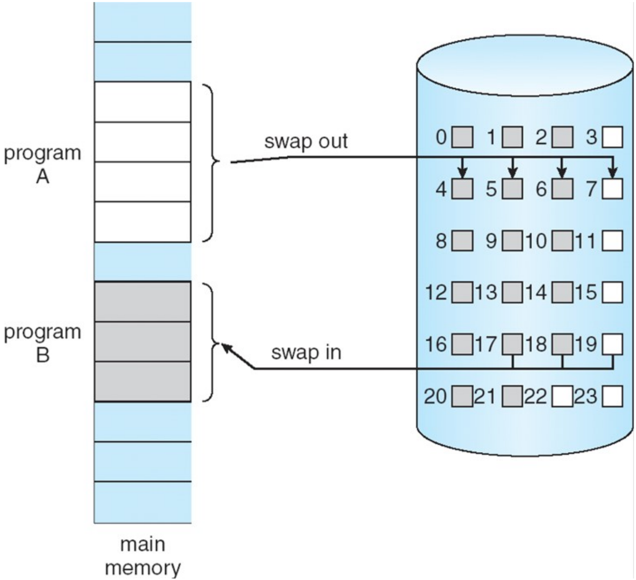
Page needed → reference it

- Invalid reference → abort
- Not-in-memory → bring into memory

Many page faults when process first starts

Eventually required pages are in memory so page fault rate drops

Demand Paging II

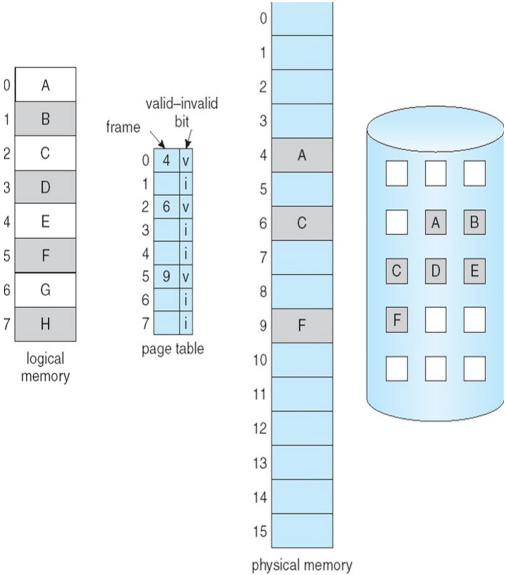


Demand Paging III

Use **valid-invalid** bit to check memory validity

- 1 → in memory
- 0 → not in memory
  - Initially set to 0 on all entries
  - If 0 during address translation → page fault

Demand Paging IV



Page Faults I

First reference, trap to OS $\rightarrow$ page fault

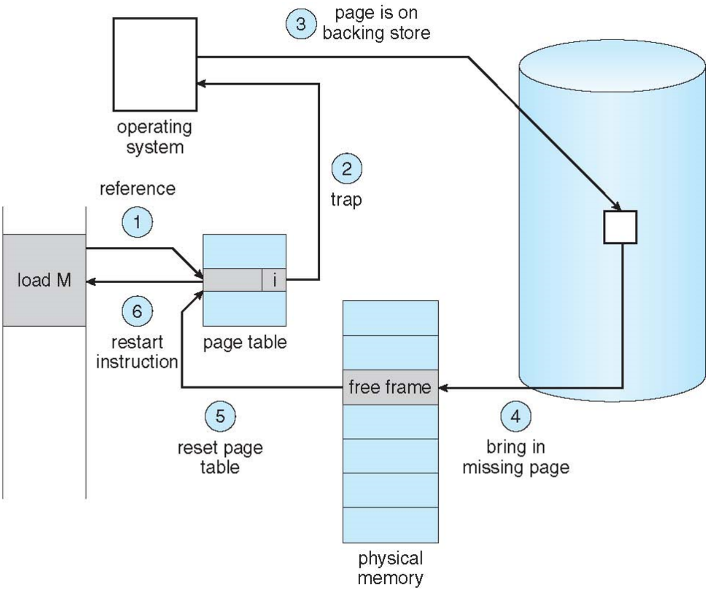
OS looks at another table to decide

- Invalid reference $\rightarrow$ abort
- Valid reference but just not in memory $\rightarrow$ handle request

To handle valid request

- Get empty frame
- Swap page into frame
- Reset tables, validation bit $= 1$
- Restart last instruction

Page Faults II



Performance: Demand Paging

Page Fault Rate (p), $0 \leq p \leq 1.0$

- If $p = 0$, no page faults
- If $p = 1$, every reference causes a page fault

Effective Access Time (EAT) $=$ (1 – p) x memory access $+$ p
x (page fault overhead
$+$ [swap page out]
$+$ swap page in
$+$ restart overhead)

Note: no need to swap page out if not modified

Virtual Memory Tricks

Copy-on-Write (COW)

- Allows parent and child processes to initially share same pages in memory → if either process <u>modifies</u> shared page, then <u>copy</u> page
- Efficient process creation: copy only modified pages
- Free pages allocated from pool of zeroed-out pages

Memory-mapped files

- Map file into virtual address space using paging
- Simplifies programming model for I/O

I/O Interlock

- Pages must sometimes be locked into memory
  - Pages used for DMA from disk

Example Problem

Demand Paging

Memory access time = 200 *ns*
Average page-fault service time = 8 *ms*

$$
\begin{aligned}
\text{EAT} &= (1 - p) \times 200 + p \times (8\ ms) \\
&= (1 - p) \times 200 + p \times 8{,}000{,}000 \\
&= 200 + p \times 7{,}999{,}800
\end{aligned}
$$

If one access out of 1,000 causes a page fault, then EAT = 8.2 *ms* → slowdown by a factor of 40!

If we want performance degradation < 10%

$$
\begin{aligned}
\text{EAT} &< \text{EAT} + 10\% \text{ of EAT} \\
200 + 7{,}999{,}800 \times p &< 220 \\
7{,}999{,}800 \times p &< 20 \\
p &< 0.0000025
\end{aligned}
$$

Less than one page fault in every 400,000 memory accesses

Page Replacement

No free frame? Replace page

How do you decide which page to replace?

Find some unused page in memory to swap out $\rightarrow$ need strategy for page replacement

Minimise number of page faults

- Avoid bringing same page into memory several times

Prevent over-allocation of memory

- Page-fault service routine should include page replacement

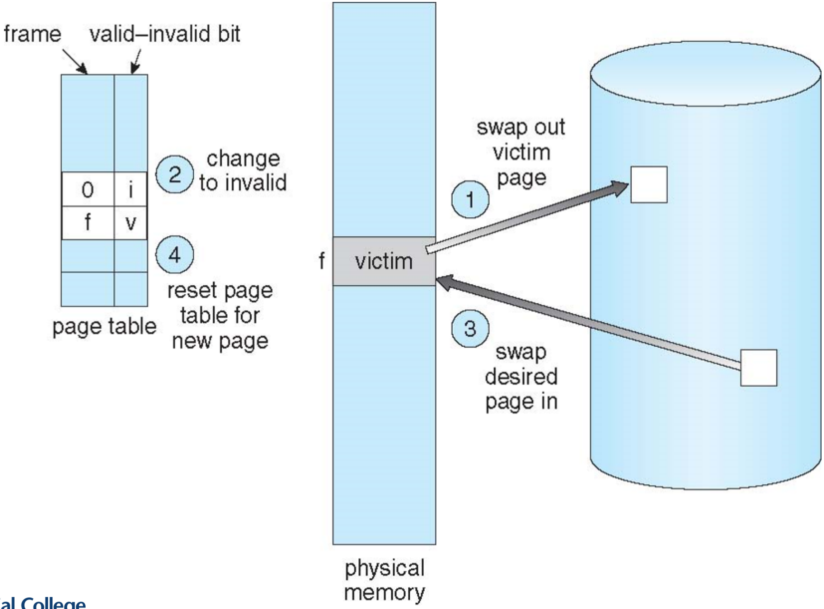
Use modify (dirty) bit to reduce overhead of page transfers

- Only modified pages written to disk

Basic Page Replacement I

1. Find location of desired page on disk
2. Find free frame
3. Frame found?
    - Yes? $\Rightarrow$ use it
    - No? $\Rightarrow$ use replacement algorithm to select **victim** frame
4. Load desired page into (newly) freed frame
5. Update page and frame tables
6. Restart process

Basic Page Replacement II



Page Replacement Algorithms

Aim: Lowest page-fault rate

How do we compare page replacement algorithms?

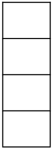
Use a Reference String → particular string of memory references and calculate number of page faults for each algorithm

E.g. 1, 2, 3, 3, 2, 4, 1, 4, 5, 5, 7, 2, 3, 1

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform



Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1
2

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1
2
3

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1
2
3
4

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1	
2	
3	
4	5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1	4
2	
3	
4	5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1	4
2	
3	
4	5

Assume 4 frames
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
6 page faults

Optimal Algorithm

Replace page that will not be used for the longest period of time

- Unimplementable, as knowledge of future references needed
- Useful for measuring how well other algorithms perform

1	4
2	
3	
4	5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

6 page faults

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

Total of 9 page faults

First-In-First-Out (FIFO) Algorithm

Replace oldest page

- May replace heavily used page

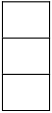
Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

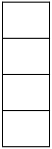
Total of 15 page faults

Heavily used pages, 0, 2, 3 are being swapped in and out

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)



Assume 3 frames



Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1

Assume 3 frames

1

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1
2

Assume 3 frames

1
2

Assume 4 frames

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1
2
3

Assume 3 frames

1
2
3

Assume 4 frames

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4
2	
3	

Assume 3 frames

1
2
3
4

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4
2	1
3	

Assume 3 frames

1
2
3
4

Assume 4 frames

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4
2	1
3	2

Assume 3 frames

1
2
3
4

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	
3	2	

Assume 3 frames

1	5
2	
3	
4	

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	
3	2	

Assume 3 frames

1	5
2	1
3	
4	

Assume 4 frames

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	
3	2	

Assume 3 frames

1	5
2	1
3	2
4	

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	3
3	2	

Assume 3 frames

1	5
2	1
3	2
4	3

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	3
3	2	4

Assume 3 frames

1	5	4
2	1	
3	2	
4	3	

Assume 4 frames

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	3
3	2	4

Assume 3 frames
9 page faults

1	5	4
2	1	5
3	2	
4	3	

Assume 4 frames
10 page faults

Belady's Anomaly

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (FIFO replacement)

1	4	5
2	1	3
3	2	4

Assume 3 frames
9 page faults

1	5	4
2	1	5
3	2	
4	3	

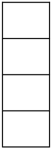
Assume 4 frames
10 page faults

Belady's Anomaly: More frames $\Rightarrow$ more page faults

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter



Assume 4 frames
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1

Assume 4 frames
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2

Assume 4 frames
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Imperial College London

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1
2
3
4

5

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1	
2	
3	5
4	3

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1		
2		
3	5	4
4	3	

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Least Recently Used (LRU) Algorithm

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1	5	
2		
3	5	4
4	3	

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

8 page faults

Each page entry has a counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter

1	5	
2		
3	5	4
4	3	

Assume 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

8 page faults

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

Total of 12 page faults

LRU Approximation Algorithms

Proper LRU is expensive $\rightarrow$ use approximations instead

Reference bit

- With each page associate reference bit r, initially r = 0
  - When page referenced, set r = 1
  - Replace page with r = 0 (if one exists)
- Periodically reset reference bits
- Does not provide proper order for LRU

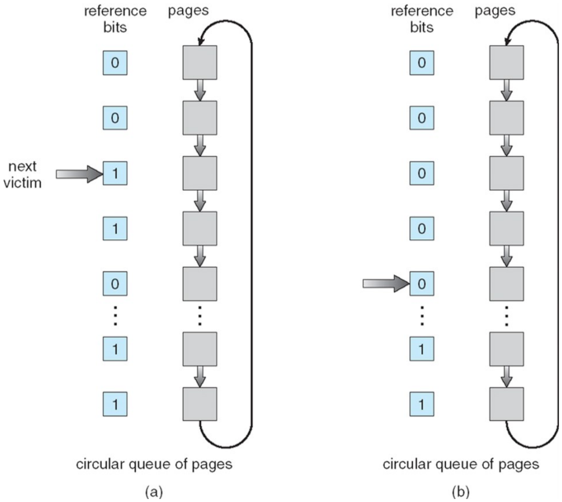
Clock Replacement Policy

- Needs reference bit r and uses clock replacement
- If page to be replaced (in clock order) has r = 1 then
  - Set r = 0 and leave page in memory
  - Continue till you find r = 0, and replace that page
  - If all r = 1, replace starting page

Clock Page Replacement

When page fault occurs, the page being pointed to is inspected

- If r = 0, evict page
- If r = 1, clear r, and advance pointer



Counting Algorithms

Keep counter of number of references made to each page

LFU (least frequently used) algorithm

- Replace page with smallest count
- May replace page just brought into memory
- Page with heavy usage in past will have high count
  - Reset counters or use **aging**

MFU (most frequently used) algorithm

- Replace page with largest count
- Page with smallest count probably just brought in and yet to be used

Example Problem

Page Replacement

Reference string: 1, 2, 1, 3, 2, 1, 4, 3, 1, 1, 2, 4, 1, 5, 6, 2, 1.
Assuming number of frames is 3, calculate the number of page faults for LRU and Clock page replacement algorithms.

Using LRU:

1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
	2	2	2	2	2	2	3	3	3	3	4	4	4	6	6	6
			3	3	3	4	4	4	4	2	2	2	5	5	5	1
Y	Y	N	Y	N	N	Y	Y	N	N	Y	Y	N	Y	Y	Y	Y

Total of 11 page faults

Using Clock:

1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
1	1	1	1	1	1	4	4	4	4	4	4	4	5	5	5	5
	2	2	2	2	2	2	2	1	1	1	1	1	1	6	6	6
			3	3	3	3	3	3	3	2	2	2	2	2	2	1
Y	Y	N	Y	N	N	Y	N	Y	N	Y	N	N	Y	Y	N	Y

Total of 9 page faults

Locality of Reference I

For program to run efficiently

- System must maintain program's favoured subset of pages in main memory

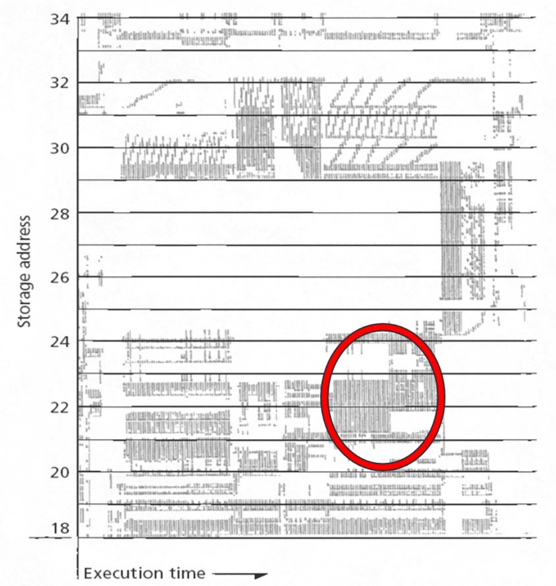
Otherwise thrashing

- Excessive paging activity causing low processor utilisation
- Program repeatedly requests pages from secondary storage

Locality of Reference

- Programs tend to request same pages in space and time

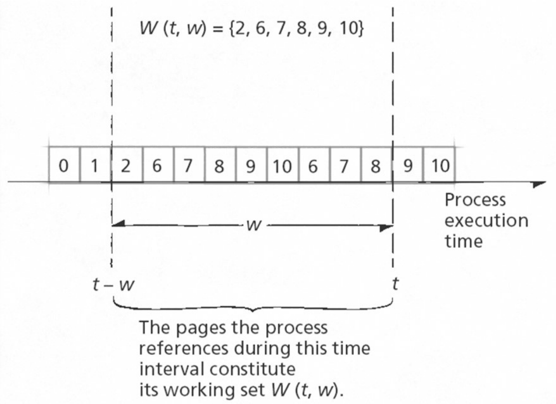
Locality of Reference II



Working Set Model

Working set of pages → W (t, w)

- Set of pages referenced by process during process-time interval (t – w) to t



Working Set Clock Algorithm

Idea: Add "time of last use" to Clock Replacement algorithm

- Keeps track if page in working set

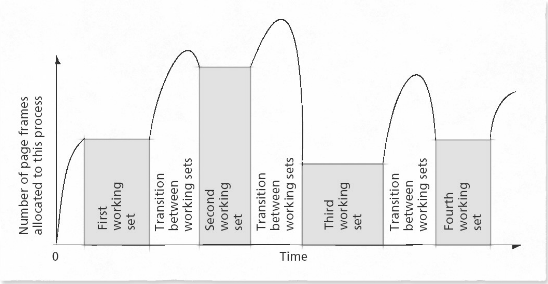
At each page fault, examine page pointed to

- If $r = 1$, then set $r = 0$ and move to next page
- If $r = 0$, calculate age
  - If age $<$ working set age w, continue (page in working set)
  - If age $>$ working set age w
    - If page is clean, replace
    - Otherwise trigger write-back, continue to next page

Working Set Size

Processes transition between working sets

- OS temporarily maintains in memory, pages outside of current working set
- Goal of memory management is to reduce mis-allocation



What about page fault frequency?

If many faults $\Rightarrow$ allocate more page frames

Global vs. Local Page Replacement

Local strategy

- Each process gets fixed allocation of physical memory
- Need to pick up changes in working set size

Global strategy

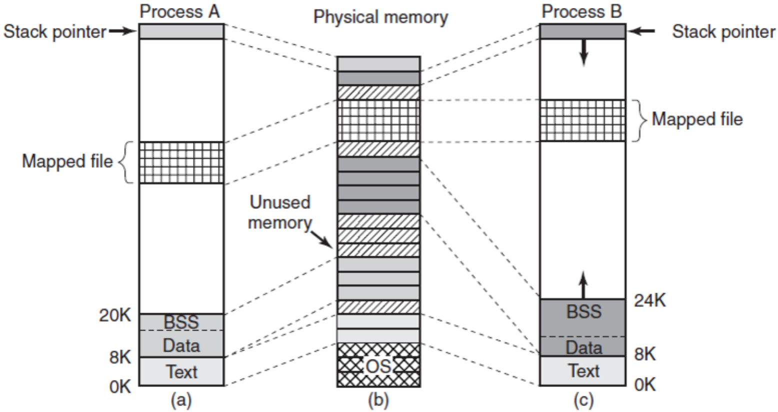
- Dynamically share memory between runnable processes
- Initially allocate memory proportional to process size
- Consider page fault frequency (PFF) to tune allocation
  - Measure page faults/per sec and increase/decrease allocation

No universally agreed solution

- Linux: global page replacement
- Windows: local page replacement
- Depends on scheduling strategy (i.e. round-robin, ...)

Linux Memory Management

Mapping and Sharing Memory



Memory Management System Calls

System call	Description
`s = brk(addr)`	Change data segment size
`a = mmap (addr,len,prot,flags,fd,offset)`	Map a file/device into memory
`s = munmap (addr,len)`	Unmap a file/device from memory

Return code `s` is -1 if error

`a` and `addr` are memory addresses

`len` is a length

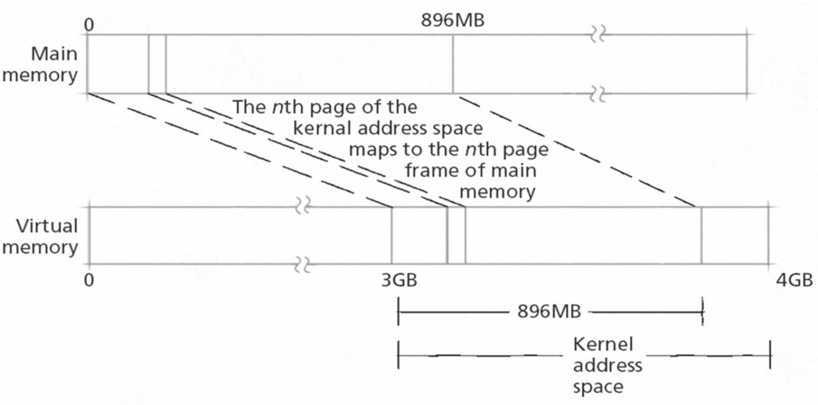
`prot` controls protection

`flags` are miscellaneous bits

`fd` is a file descriptor

`offset` is a file offset

Virtual Memory Layout I



Virtual Memory Layout II

On a 32-bit machine, process has 4 GB of space

Top 1 GB used for Kernel memory

- User processes can make system calls without TLB flush
- Kernel space not visible in user mode
- Kernel typically resides in 0 – 1 GB of physical memory

Kernel maps lower 896 MB of physical memory to its virtual address space

- All memory access must be virtual but need efficient access to user memory + DMA in low memory
- Create temporary mappings for $>$ 896 MB of physical memory in remaining 128 MB of virtual memory

Physical Memory Management

Linux memory zones

- ZONE_DMA and ZONE_DMA32: pages used for DMA
- ZONE_NORMAL: normal regularly mapped pages
- ZONE_HIGHMEM ($>$ 896 MB): pages with high memory addresses – not permanently mapped

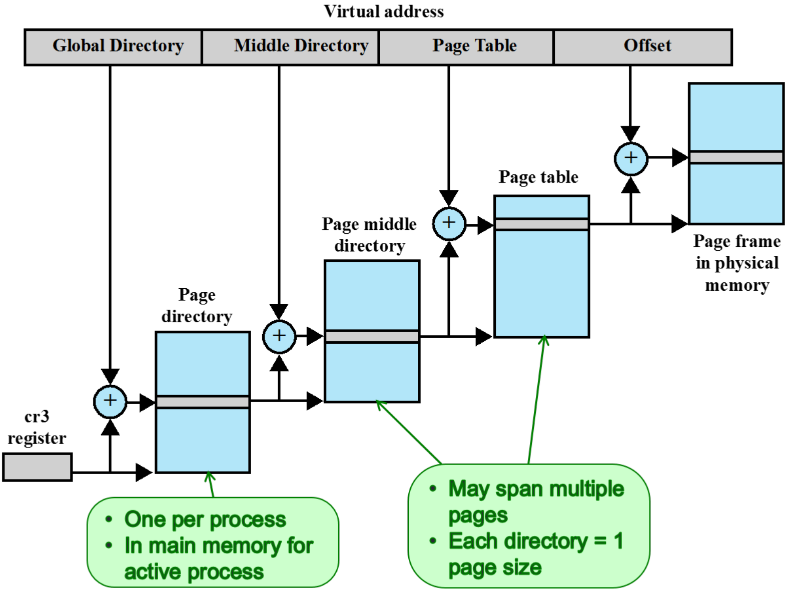
Kernel and memory map are pinned, i.e. never paged out

Paging

	Usually on IA-32	On x86-64
Page size	4 KB	Larger page sizes (e.g. 4 MB)
Virtual address space	4 GB	128 TB
Page-table	Two-level (or three levels with Physical Address Extension (PAE))	Up to four-level page table

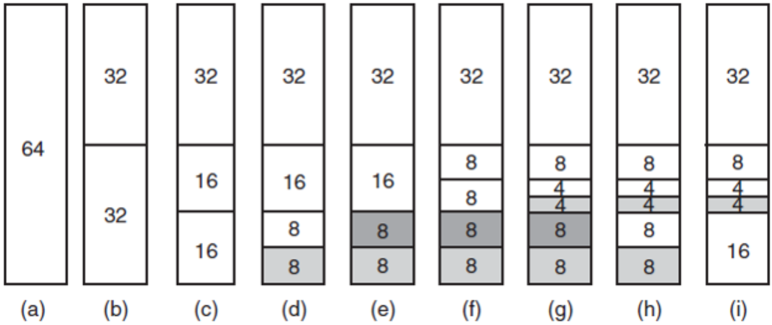
Offset bits contain page status: dirty, read-only, ...

3-level Paging



Imperial College London

Buddy Memory Allocation



- Tries to map contiguous pages to contiguous frames to optimise transfers
- Split and merge frames as required

Imperial College London

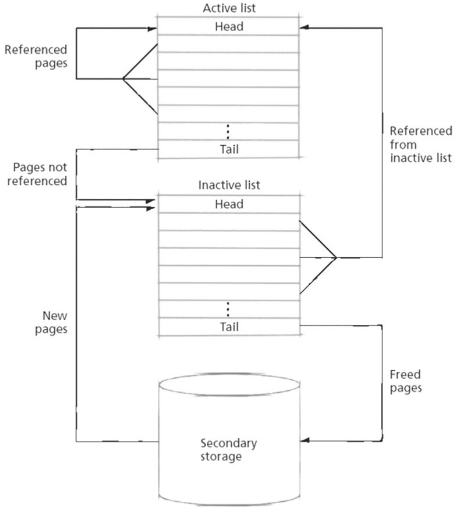
Page Replacement I

Linux uses variation of clock algorithm to approximate LRU page-replacement strategy

Memory manager uses two linked lists (and reference bits)

- **Active list**
  - Contains active pages
  - Most-recently used pages near head of active list
- **Inactive list**
  - Contains inactive pages
  - Least-recently used pages near tail of inactive list
- Only replace pages in inactive list

Page Replacement II



Page Replacement III

kswapd (swap daemon)

- Pages in inactive list reclaimed when memory low
- Uses dedicated swap partition or file
- Must handle locked and shared pages

pdflush kernel thread

- Periodically flushes dirty pages to disk