Memory Management

Basic Concepts

- Memory Allocation
- Swapping

Virtu

Pagi

- Demand Paging
- Page replacement algorithms

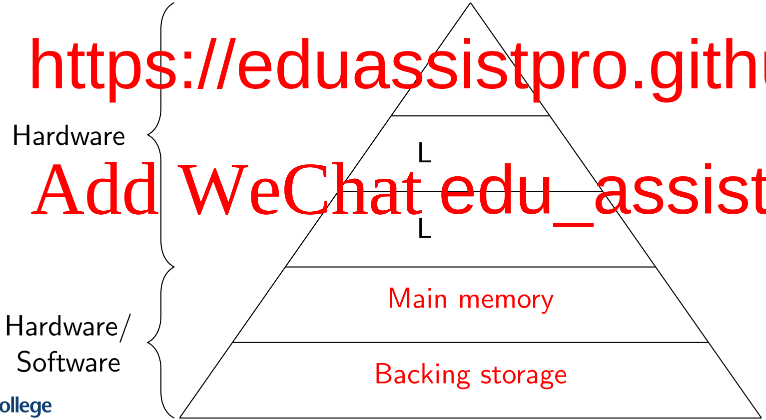
- Working set model

Linux Memory Management

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 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

-
-

Characteristics

- No knowledge of how memory addresses ar
  - e.g. instruction counter, indexing, ind

- No knowledge what memory addresses ar
  - 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

Phy

- Refers to physical system memory

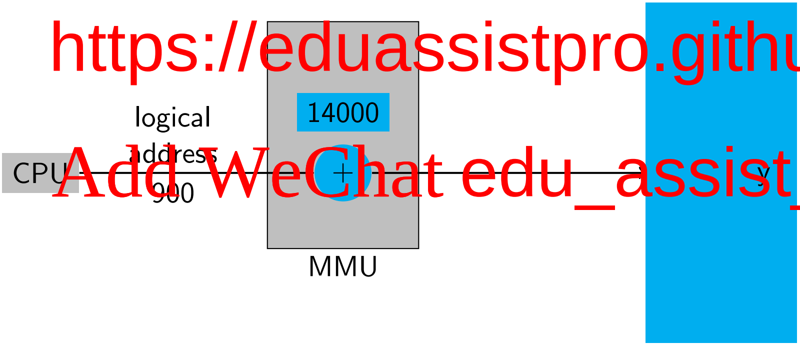
Logical and physical addresses

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

How do you achieve this mapping?

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



Main memory is usually split into two partitions:

- Resident operating system (**kernel**)
  - Usually held in low memory with interrupt vector
-
  -

How do you decide where to load a new process?

Need to figure out the strategy for process to be loade
correct location

Contiguous allocation with relocation registers

- base register contains physical start address for process
- [illegible]
- [illegible]
  - Physical address = logical address + bas
  - If logical address > limit then erro

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 m

What is the best algorithm for allocation?

First-fit → Allocate first hole that is big enough

Best-fit → Allocate smallest hole that is big enough

- Must search entire list, unless ordered by size
-

Wor

-
- 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

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

Swapping

Problem: Number of processes limited by amount of available memory

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

Solu

- Bring back into memory for continued exec
- Requires swap space → can be file or d disk

- Transfer time is major part of swap time

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

Separation of user logical memory from physical memory

- Only part of process needs to be in memory for execution
-
- Address spaces can be shared by several pro
- Allows for more efficient process creation

Virtual memory can be implemented via

-
-

Physical address space of process can be noncontiguous

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

Fra

-
-

Pages

- Block of same size (as frame) of l

To run program of size *n* pages

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

How does logical address translate to physical address?

Hint: he

page

Addr **e offset**

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

How do you calculate the page number?

Depends on address size and page/frame size

e.g. C

-
-)

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

- page offset requires 6 bits (based on above)
- page number has 4 bits (remaining bits) ⇒ between 0 . . . 15

Page number (p)

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

Pag

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

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

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 p

Num

For a 32-bit address you have:

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

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

Example - Calculating Internal Fragmentation

Page

Num

Byte

Internal fragmentation $= 2048 - 1086 = 962$ bytes

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

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

Are s

-
- architectures support variable page sizes

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
-

Problem

- Inefficient as every data/instruction acc memory accesses → one for page tabl data/instruction

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

Associative memory → supports parallel search

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

[illegible]	[illegible]
9	5
7	[illegible]

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

TLBs usually needs to be flushed after context switch

- Can lead to substantial overhead
-

Som

- Uniquely identifies each process to provid
  protection for that process

Effective Access Time

TLB Lookup = ϵ (can be $<$ 10% of memory access time *m*)

Hit Ratio = α

- Fraction of times that page is found in associative registers
-

Effec α)

= 2m

Consider α = 80%, ϵ = 10 *ns* for TLB se

memory access

- EAT = 110 x 0.80 + 210 x 0.20 = 130

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:

- [illegible] 32 [illegible]
- [illegible]

- Size of page table = 2 [illegible] x 32 bits = 4 MB

On 64-bit machine with 4 KB pages → [illegible] tries

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

- **Hierarchical** page table

- **Has**

- **Inverted** page table

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
-

How d

Each part of the page table that is being paged must __

- 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 $= 32 - 10 - 12 = 10$

Fix outer page table in memory

Logical address divided

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

Since page table paged, page number further divided

-
-

Thus, logical addresses as follows

page number		pa
p_1	p_2	
10	10	12

p_1 $\rightarrow$ index into the outer page table

p_2 $\rightarrow$ displacement within page pointed to by outer page table

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 virtu

Ans sses in the ad

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

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

Segment

- Independent address space from 0 to some maximum
-
-
-
- Segment corresponds to program, proce
  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

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

Most OSs use only paging

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

Valid-invalid bit

-
-
  - backing store

Page replacement bits → to indicate if p

or referenced (used later). Also, lock bit to prevent p being transferred out

When do you bring the page into memory?

Bring page into memory only when needed

- Lower I/O load
-
-
-

Page needed → reference it

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

Many page faults when process first starts

Eventually required pages are in memory so page fault rate drops

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

-
-
  -
  - If 0 during address translation

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

OS looks at another table to decide

- Invalid reference abort
- st

To ha

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

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

Eff

$+$ [swa

$+$ swa

$+$ resta

Note: no need to swap page out if not modified

Copy-on-Write (COW)

- Allows parent and child processes to initially share same pages in memory → if either process modifies shared page, then copy page
- Efficient process creation: copy only modified pages
-

Me

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

I/O Interlock

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

Demand Paging

Memory access time $= 200$ *ns*

Average page-fault service time $= 8$ *ms*

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

If one a [illegible] *ms* $\rightarrow$
slow

If we want performance degradation < 1 [illegible]

EAT	$<$ EAT $+$ 10% of EA [illegible]
$200 + 7{,}999{,}800 \times p$	< 220
$7{,}999{,}800 \times p$	< 20
p	< 0.0000025

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

No free frame? Replace page

How do you decide which page to replace?

Find some unused page in memory to swap out → need strategy for page replacement

Mini

-

Prevent over-allocation of memory

- Page-fault service routine should includ

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

- Only modified pages written to disk

1. Find location of desired page on disk

2. Find free frame
3.
   -
   - ame

4. Load desired page into (newly) freed frame

5. Update page and frame tables
6. Restart process

Aim: Lowest page-fault rate

How d

Use a R
and ca

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

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

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

Refe

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 ou

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

1	4	5

Assume 3 frames

2	1	5
3	2	
4	3	

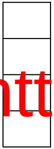
Assume 4 frames

10 page faults

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

Each page entry has a counter

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



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

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

Clock Replacement Policy

- Needs reference bit r and uses clock replace
- 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
-
-
    -

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
			[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	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

For program to run efficiently

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

Othe

- Program repeatedly requests pages from s

Locality of Reference

- Programs tend to request same pages in space and time

Working Set Model

Working set of pages → W (t, w)

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

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

- Keeps track if page in working set

At eac

-
-
    - If age $<$ working set age w, continue (
    - 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

Local strategy

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

Global strategy

  - Measure page faults/per sec and increa

No universally agreed solution

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

Assignment Project Exam Help

Li

https://eduassistpro.github.i

Add WeChat edu_assist_pr

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

Retu

`a` an

`len`

`prot` controls protection

`flags` are miscellaneous bits

`fd` is a file descriptor

`offset` is a file offset

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
-
-

Kern
address space

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

Linux memory zones

- ZONE_DMA and ZONE_DMA32: pages used for DMA
-
- addresses – not permanently mapped

Kernel and memory map are pinned, i.e. never pag

	Usually on IA-32	On x86-64
Page size	4 KB	Larger page sizes (e.g. 4 MB)
Vir spa		
Pa	levels with Physical Address Extension (PAE))	e

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

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

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

Memory manager uses two linked lists (and reference bits)

-
- **Inactive list**
  - Contains inactive pages
  - Least-recently used pages near tail of ina

- Only replace pages in inactive list

kswapd (swap daemon)

- Pages in inactive list reclaimed when memory low

pdflush kernel thread

- Periodically flushes dirty pages to disk