CS307 Operating Systems

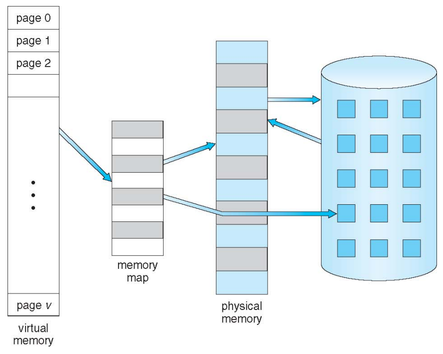
Virtual Memory

Fan Wu
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Spring 2020

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program could be larger than physical memory

Virtual Memory That is Larger Than Physical Memory



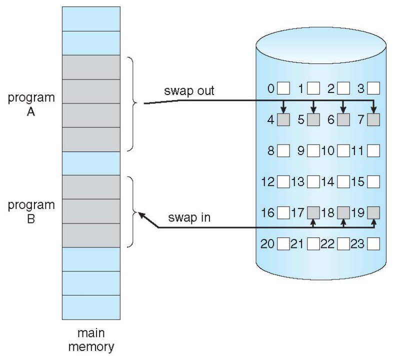
Virtual Memory

- **Virtual Memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows memory address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

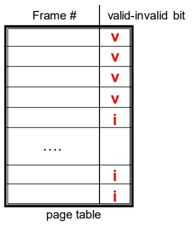
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper (pager)** – never swaps a page into memory unless page will be needed

Swap Paged Memory to Disk Space



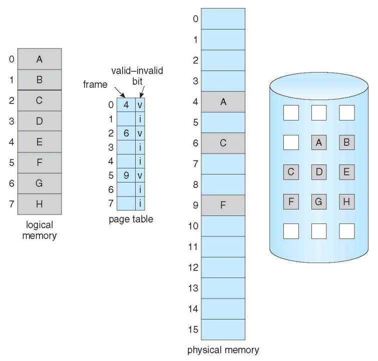
Valid-Invalid Bit

- With each page table entry a **valid–invalid** bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially, valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table with Pages Not in Main Memory

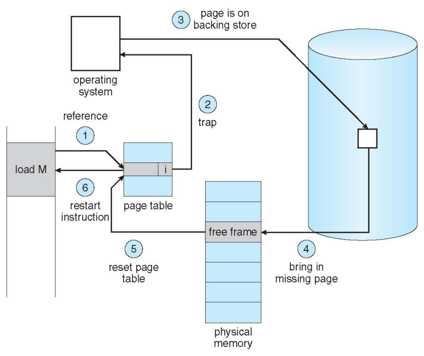


Page Fault

- If there is a reference to a page and the page is not in memory, the reference will trap to operating system:
  page fault

1. Operating system looks at page table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

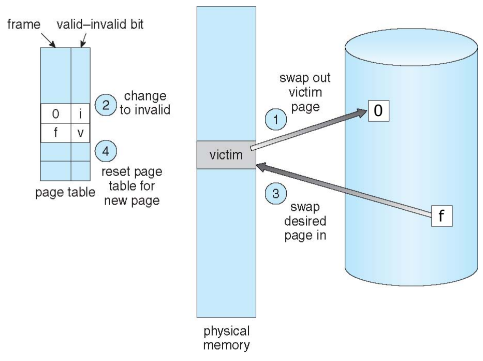
Steps in Handling a Page Fault



What Happens if There is no Free Frame?

- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

Page Replacement



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Page Replacement Algorithms

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page, which is still in memory, does not cause a page fault
- In all our examples, the reference string is

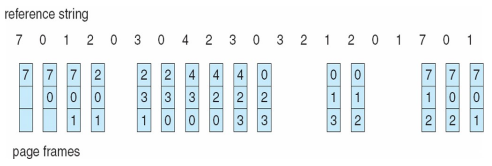
  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Page-Replacement Algorithms

- First-In-First-Out (FIFO) Page Replacement
- Optimal Page Replacement
- Least Recently Used (LRU) Page Replacement
- LRU Approximation Page Replacement
- Counting Page Replacement

FIFO Page Replacement

- When a page must be replaced, the oldest page is chosen.

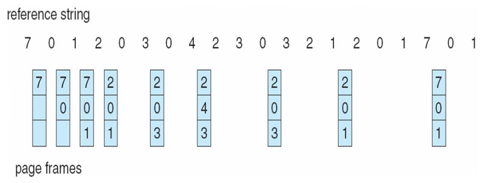


- Page faults: 15
- Consider the following reference string:

  0 1 2 3 0 1 2 3 0 1 2 3……

Optimal Page Replacement

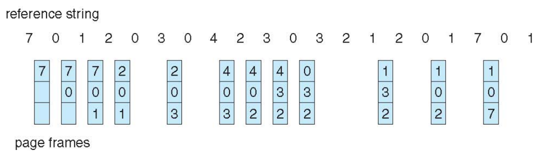
- Replace page that will not be used for longest period of time



- Page faults: 9
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Page Replacement

- Use past knowledge rather than future
- Replace page that has not been used in most amount of time
- Associate time of last use with each page

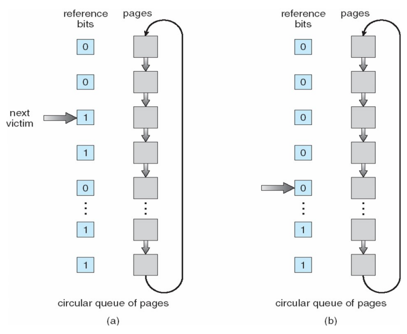


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

LRU Approximation Algorithms

- **Reference bit/ byte**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not specify the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Circular replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - Reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance Algorithm

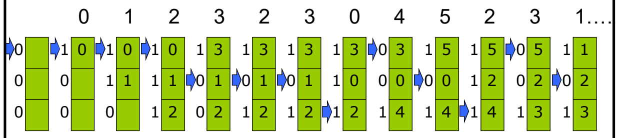


Pop Quiz

- A memory system has **three** frames. Consider the following reference string

0 1 2 3 2 3 0 4 5 2 3 1 4 3 2 6 3 2 1 2

Draw a diagram to show the page replacement using **Second-Chance Algorithm** and calculate the number of page faults.



Counting Algorithms

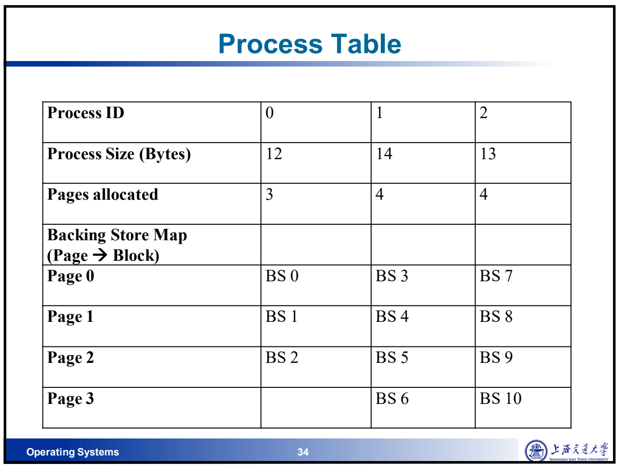
- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used (LFU) Algorithm**: replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Not commonly used

Homework

- Reading
  - Chapter 9
- Exercise
  - See course website

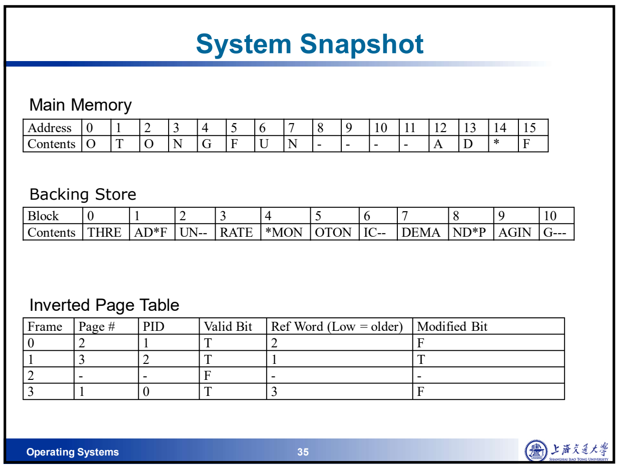
Demand Paging

System Characteristics	
Size of memory	16 bytes
Frame Size	4 bytes per frame
Memory Management Structure	Inverted Page Table
Replacement Policy	LRU, Global Replacement
Virtual Page Size	4 bytes per page
Logical Addressing Space Size	32 bytes
Backing Store Size	12 blocks
Backing Store Block Size	4 bytes per block



Process Table

Process ID	0	1	2
Process Size (Bytes)	12	14	13
Pages allocated	3	4	4
Backing Store Map (Page → Block)			
Page 0	BS 0	BS 3	BS 7
Page 1	BS 1	BS 4	BS 8
Page 2	BS 2	BS 5	BS 9
Page 3		BS 6	BS 10



System Snapshot

Main Memory

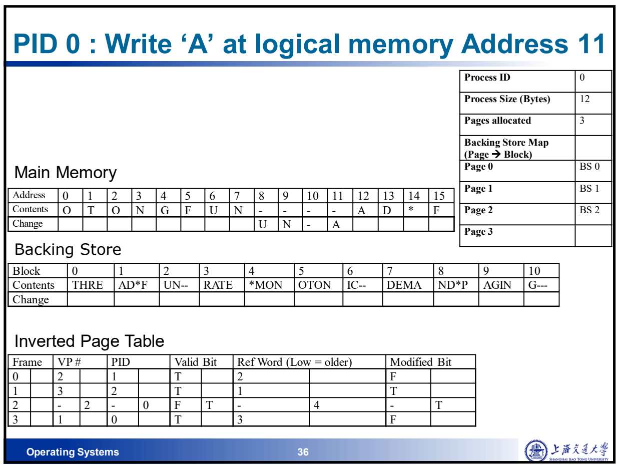
Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	O	T	O	N	G	F	U	N	-	-	-	-	A	D	*	F

Backing Store

Block	0	1	2	3	4	5	6	7	8	9	10
Contents	THRE	AD*F	UN--	RATE	*MON	OTON	IC--	DEMA	ND*P	AGIN	G---

Inverted Page Table

Frame	Page #	PID	Valid Bit	Ref Word (Low = older)	Modified Bit
0	2	1	T	2	F
1	3	2	T	1	T
2	-	-	F	-	-
3	1	0	T	3	F



PID 0 : Write 'A' at logical memory Address 11

Process ID	0
Process Size (Bytes)	12
Pages allocated	3
Backing Store Map (Page → Block)	
Page 0	BS 0
Page 1	BS 1
Page 2	BS 2
Page 3	

Main Memory

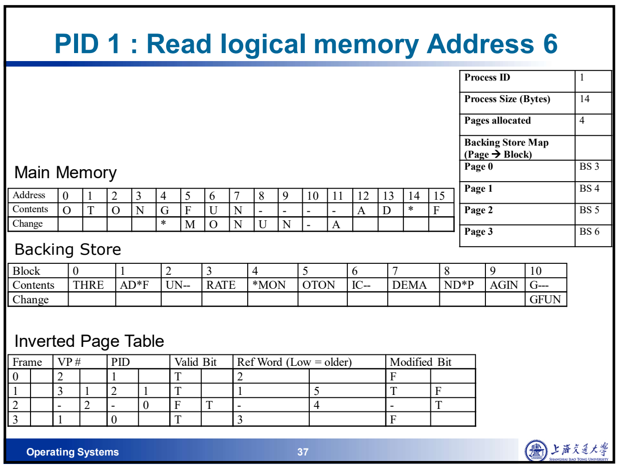
Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	O	T	O	N	G	F	U	N	-	-	-	-	A	D	*	F
Change									U	N	-	A				

Backing Store

Block	0	1	2	3	4	5	6	7	8	9	10
Contents	THRE	AD*F	UN--	RATE	*MON	OTON	IC--	DEMA	ND*P	AGIN	G---
Change											

Inverted Page Table

Frame		VP #		PID		Valid Bit		Ref Word (Low = older)		Modified Bit	
0		2		1		T		2		F	
1		3		2		T		1		T	
2		-	2	-	0	F	T	-	4	-	T
3		1		0		T		3		F	



PID 1 : Read logical memory Address 6

Process ID	1
Process Size (Bytes)	14
Pages allocated	4
Backing Store Map (Page → Block)	
Page 0	BS 3
Page 1	BS 4
Page 2	BS 5
Page 3	BS 6

Main Memory

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	O	T	O	N	G	F	U	N	-	-	-	-	A	D	*	F
Change					*	M	O	N	U	N	-	A				

Backing Store

Block	0	1	2	3	4	5	6	7	8	9	10
Contents	THRE	AD*F	UN--	RATE	*MON	OTON	IC--	DEMA	ND*P	AGIN	G---
Change											GFUN

Inverted Page Table

Frame		VP #		PID		Valid Bit		Ref Word (Low = older)		Modified Bit	
0		2		1		T		2		F	
1		3	1	2	1	T		1	5	T	F
2		-	2	-	0	F	T	-	4	-	T
3		1		0		T		3		F	