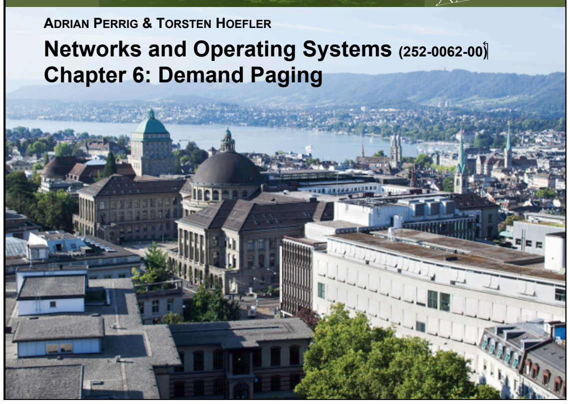
ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 6: Demand Paging



Page Table Structures

Page table structures

- **Problem: simple linear page table is too big**
- **Solutions:**
  1. Hierarchical page tables
  2. Virtual memory page tables
  3. Hashed page tables
  4. Inverted page tables

Page table structures

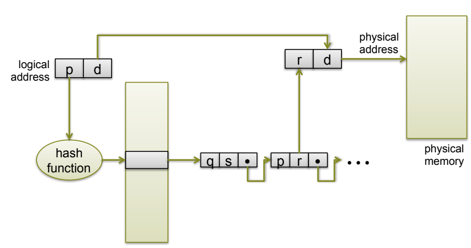
- **Problem: simple linear page table is too big**
- **Solutions:**
  1. Hierarchical page tables
  2. Virtual memory page tables (VAX)
  3. Hashed page tables
  4. Inverted page tables

Saw these last Semester.

#3 Hashed Page Tables

- **VPN is hashed into table**
  - Hash bucket has chain of logical->physical page mappings
- **Hash chain is traversed to find match.**
- **Can be fast, but can be unpredicable**
- **Often used for**
  - Portability
  - Software-loaded TLBs (e.g., MIPS)

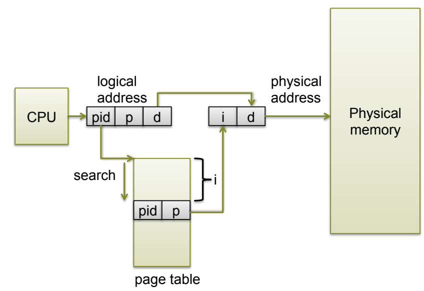
Hashed Page Table



#4 Inverted Page Table

- **One system-wide table now maps PFN -> VPN**
  - One entry for each real page of memory
  - Contains VPN, and which process owns the page
- **Bounds total size of all page information on machine**
  - Hashing used to locate an entry efficiently
- **Examples: PowerPC, ia64, UltraSPARC**

Inverted Page Table Architecture



The need for more bookkeeping

- **Most OSes keep their own translation info**
  - Per-process hierarchical page table (Linux)
  - System wide inverted page table (Mach, MacOS)
- **Why?**
  - Portability
  - Tracking memory objects
  - Software virtual → physical translation
  - Physical → virtual translation

TLB shootdown

TLB management

- **Recall: the TLB is a *cache*.**
- **Machines have many MMUs on many cores ⇒ many TLBs**
- **Problem: TLBs should be coherent. Why?**
  - Security problem if mappings change
  - E.g., when memory is reused

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

Process 0 on core 1 can only continue once shootdown is complete!

Keeping TLBs consistent

1. **Hardware TLB coherence**
   - Integrate TLB mgmt with cache coherence
   - Invalidate TLB entry when PTE memory changes
   - Rarely implemented
2. **Virtual caches**
   - Required cache flush / invalidate will take care of the TLB
   - High context switch cost!
     ⇒ Most processors use physical caches
5. **Software TLB shootdown**
   - Most common
   - OS on one core notifies all other cores - Typically an IPI
   - Each core provides local invalidation
6. **Hardware shootdown instructions**
   - Broadcast special address access on the bus
   - Interpreted as TLB shootdown rather than cache coherence message
   - E.g., PowerPC architecture

Our Small Quiz

- **True or false (raise hand)**
  - Base (relocation) and limit registers provide a full virtual address space
  - Base and limit registers provide protection
  - Segmentation provides a base and limit for each segment
  - Segmentation provides a full virtual address space
  - Segmentation allows shared libraries
  - Segmentation provides linear addressing
  - Segment tables are set up for each process in the CPU
  - Segmenting prevents internal fragmentation
  - Paging prevents internal fragmentation
  - Protection information is stored at the physical frame
  - Pages can be shared between processes
  - The same page may be writeable in proc. A and write protected in proc. B
  - The same physical address can be references through different addresses from (a) two different processes – (b) the same process?
  - Inverted page tables are faster to search than hierarchical (asymptotically)

17

Today

- **Uses for virtual memory**
- **Copy-on-write**
- **Demand paging**
  - Page fault handling
  - Page replacement algorithms
  - Frame allocation policies
  - Thrashing and working set

Recap: Virtual Memory

- **User logical memory ≠ physical memory.**
  - Only part of the program must be in RAM for execution
    ⇒ Logical address space can be larger than physical address space
  - Address spaces can be shared by several processes
  - More efficient process creation
- ***Virtualize* memory using software+hardware**

The many uses of address translation

- **Process isolation**
- **IPC**
- **Shared code segments**
- **Program initialization**
- **Efficient dynamic memory allocation**
- **Cache management**
- **Program debugging**
- **Efficient I/O**
- **Memory mapped files**
- **Virtual memory**
- **Checkpoint and restart**
- **Persistent data structures**
- **Process migration**
- **Information flow control**
- **Distributed shared memory**

and many more …

Copy-on-write (COW)

Recall `fork()`

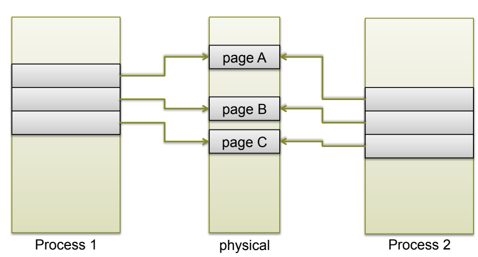
- **Can be expensive to create a complete copy of the process' address space**
  - Especially just to do `exec()`!
- **`vfork()`: shares address space, doesn't copy**
  - Fast
  - Dangerous – two writers to same heap
- **Better: only copy when you know something is going to get written**

Copy-on-Write

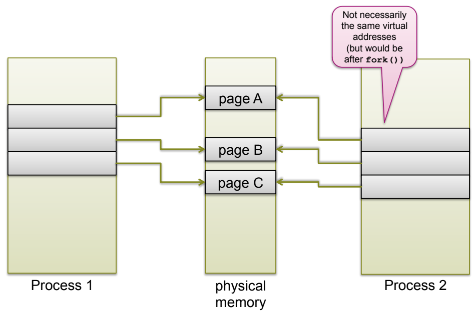
- **COW allows both parent and child processes to initially *share* the same pages in memory**

  If either process modifies a shared page, only then is the page copied
- **COW allows more efficient process creation as only modified pages are copied**
- **Free pages are allocated from a pool of zeroed-out pages**

Example: processes sharing an area of memory



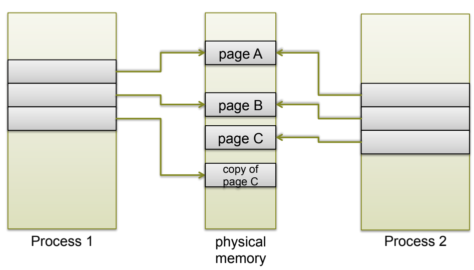
Example: processes sharing an area of memory



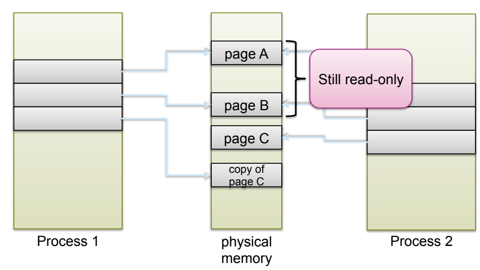
How does it work?

- **Initially mark all pages as read-only**
- **Either process writes ⇒ page fault**
  - Fault handler allocates new frame
  - Makes copy of page in new frame
  - Maps each copy into resp. processes writeable
- **Only modified pages are copied**
  - Less memory usage, more sharing
  - Cost is page fault for each mutated page

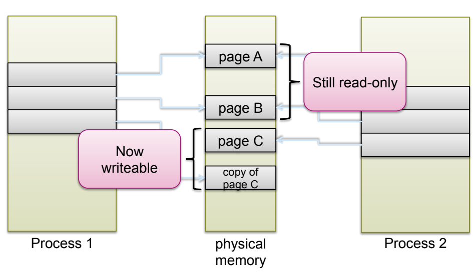
After process 1 writes to page C



After process 1 writes to page C



After process 1 writes to page C

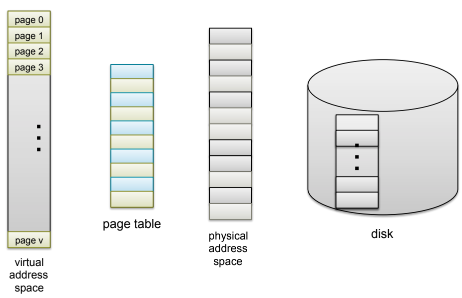


General principle

- **Mark a VPN as invalid or readonly ⇒ trap indicates attempt to read or write**
- **On a page fault, change mappings somehow**
- **Restart instruction, as if nothing had happened**
- **General: allows *emulation* of memory as well as *multiplexing*.**
  - E.g. on-demand zero-filling of pages
  - And…

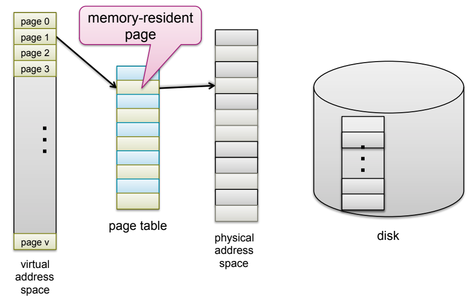
ETH zürich spcl.inf.ethz.ch @spcl_eth

Paging concepts



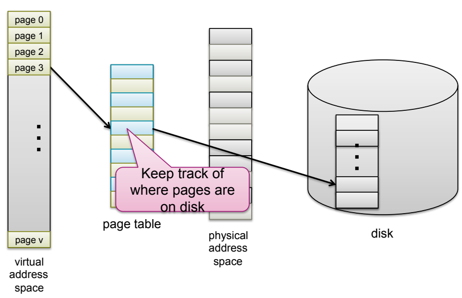
ETH zürich spcl.inf.ethz.ch @spcl_eth

Paging concepts



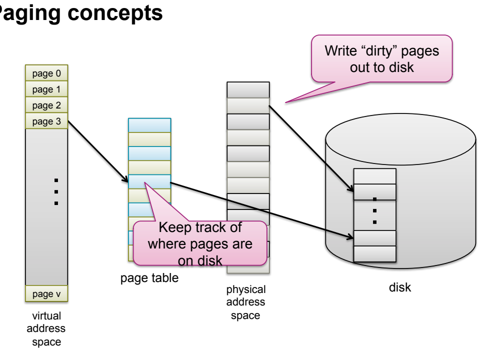
ETH zürich spcl.inf.ethz.ch @spcl_eth

Paging concepts



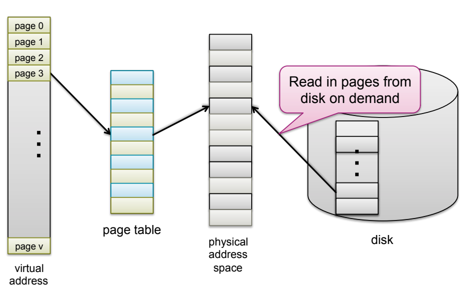
ETH zürich spcl.inf.ethz.ch @spcl_eth

Paging concepts



ETH zürich spcl.inf.ethz.ch @spcl_eth

Paging concepts



ETH zürich spcl.inf.ethz.ch @spcl_eth

Demand Paging

- **Bring a page into memory only when it is needed**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- **Turns RAM into a *cache* for processes on *disk*!**

Demand Paging

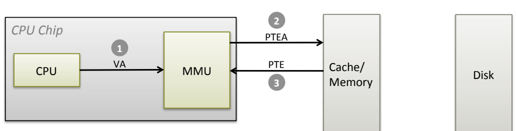
- Page needed ⇒ reference (load or store) to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- ***Lazy swapper*** **– never swaps a page into memory unless page will be needed**
  - Swapper that deals with pages is a pager
  - Can do this with segments, but more complex
- ***Strict demand paging*****: only page in when referenced**

Page Fault

- **If there is a reference to a page, first reference to that page will trap to operating system:**
  page fault

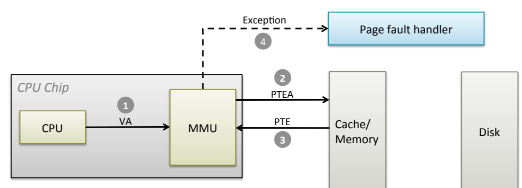
1. **Operating system looks at another table to decide:**
   - Invalid reference ⇒ abort
   - Just not in memory
2. **Get empty frame**
3. **Swap page into frame**
4. **Reset tables**
5. **Set validation bit = v**
6. **Restart the instruction that caused the page fault**

Recall: handling a page fault



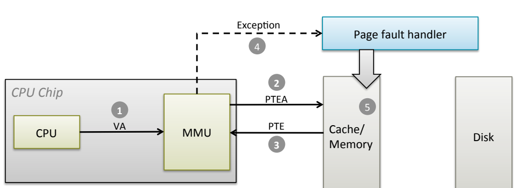
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory

Recall: handling a page fault



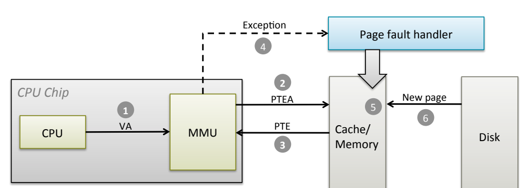
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception

Recall: handling a page fault



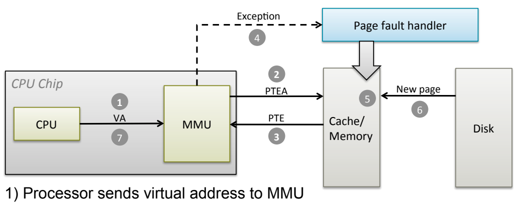
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler finds a frame to use for missing page

Recall: handling a page fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) **Handler finds a frame to use for missing page**
6) Handler pages in new page and updates PTE in memory

Recall: handling a page fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) **Handler finds a frame to use for missing page**
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Performance of demand paging

- **Page Fault Rate 0 ≤ *p* ≤ 1.0**
  - if *p* = 0 no page faults
  - if *p* = 1, every reference is a fault
- **Effective Access Time (EAT)**

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in
+ restart overhead
)

Demand paging example

- **Memory access time = 200 nanoseconds**
- **Average page-fault service time = 8 milliseconds**
- **EAT = (1 – p) x 200 + p (8 milliseconds)**
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- **If one access out of 1,000 causes a page fault, then**
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

Page Replacement

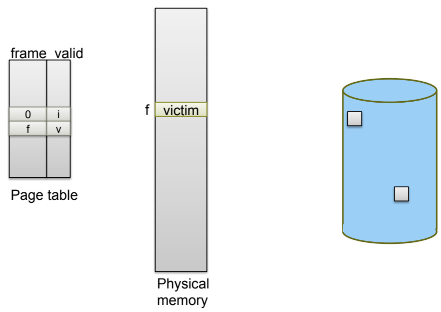
What happens if there is no free frame?

- ***Page replacement* – find "little used" resident page to discard or write to disk**
  - "victim page"
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- **Same page may be brought into memory several times**

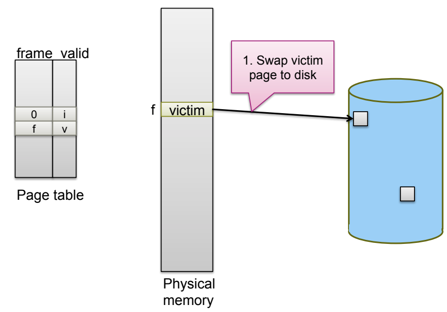
Page replacement

- **Try to pick a victim page which won't be referenced in the future**
  - Various heuristics – but ultimately it's a guess
- **Use "modify" bit on PTE**
  - Don't write "clean" (unmodified) page to disk
  - Try to pick "clean" pages over "dirty" ones (save a disk write)

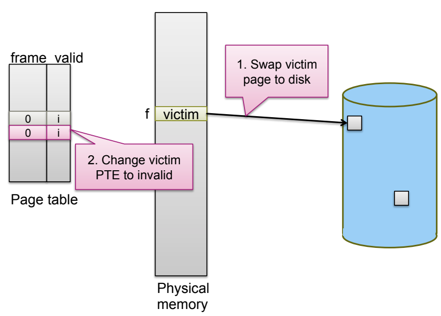
Page replacement



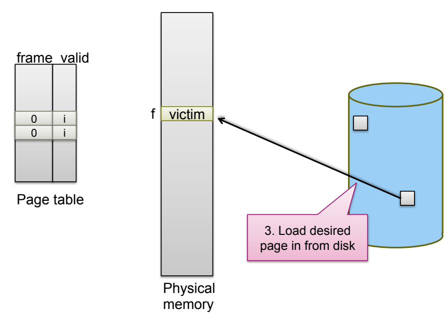
Page replacement



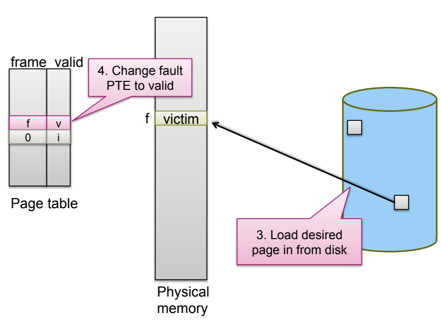
Page replacement



Page replacement



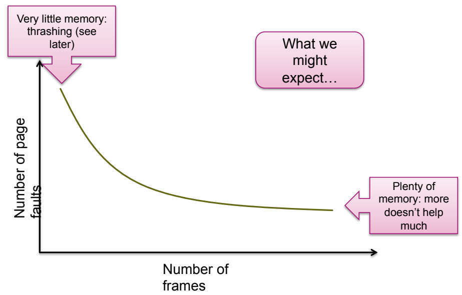
Page replacement



Page replacement algorithms

- **Want lowest page-fault rate**
- **Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string**
- **E.g.**
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7 , 0, 1

Page faults vs. number of frames



FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7	7
	0

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7	7	7
	0	0
		1

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7	7	7	2
	0	0	0
		1	1

ETH zürich spcl.inf.ethz.ch @spcl_eth

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

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

Here, 15 page faults.

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

▪

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

1	1	1
	2	2
		3

▪

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

1	1	1	4
	2	2	2
		3	3

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

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

9 page faults

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

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

9 page faults

- **4 frames:**

1	1	1	1
	2	2	2
		3	3
			4

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

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

9 page faults

- **4 frames:**

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

1	1	1	4	4	4	5	5	5	
	2	2	2	1	1	1	3	3	9 page faults
		3	3	3	2	2	2	4	

- **4 frames:**

1	1	1	1	5	5	5	5	4	4	
	2	2	2	2	1	1	1	1	5	10 page faults!
		3	3	3	3	2	2	2	2	
			4	4	4	4	3	3	3	

ETH zürich spcl.inf.ethz.ch @spcl_eth

More memory is better?

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**

1	1	1	4	4	4	5	5	5	
	2	2	2	1	1	1	3	3	9 page faults
		3	3	3	2	2	2	4	

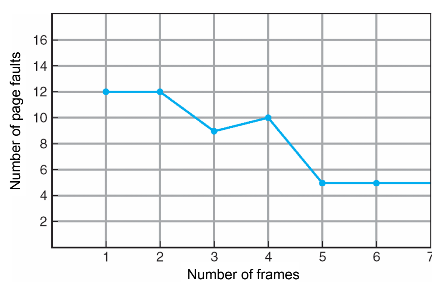
- **4 frames:**

1	1	1	1	5	5	5	5	4	4	
	2	2	2	2	1	1	1	1	5	10 page faults!
		3	3	3	3	2	2	2	2	
			4	4	4	4	3	3	3	

Belady's Anomaly: more frames ⇒ more page faults

ETH zürich spcl.inf.ethz.ch @spcl_eth

FIFO showing Belady's Anomaly



ETH zürich spcl.inf.ethz.ch @spcl_eth

Optimal algorithm

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

4 frames example:

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

1	1	4	
2	2	2	⇒ 6 page faults
3	3	3	
4	5	5	

How do you know this? – you can't!
Used for measuring how well your algorithm performs

ETH zürich spcl.inf.ethz.ch @spcl_eth

Optimal page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

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

Here, 9 page faults.

ETH zürich spcl.inf.ethz.ch @spcl_eth

Least Recently Used (LRU) algorithm

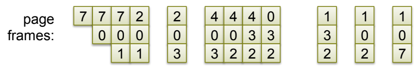
- **Reference string: 1 2 3 4 1 2 5 1 2 3 4 5**

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

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

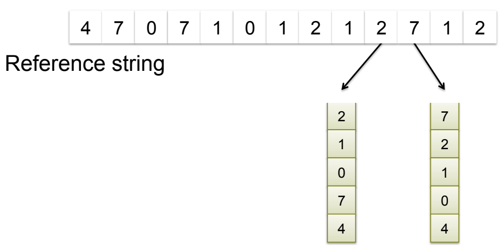


Here, 12 page faults.

LRU algorithm

- **Stack implementation – keep a stack of page numbers in a double link form:**
  - Page referenced:
    move it to the top
    requires 6 pointers to be changed
  - No search for replacement
- **General term: *stack algorithms***
  - Have property that adding frames always reduces page faults (no Belady's Anomaly)

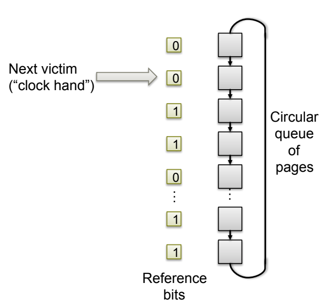
Use a stack to record most recent page references



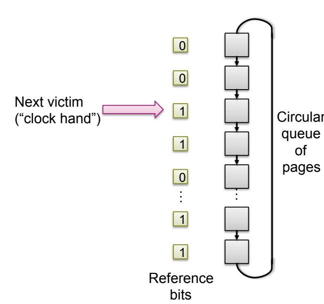
LRU approximation algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace a page which is 0 (if one exists)
    We do not know the order, however
- **Second chance**
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    set reference bit 0
    leave page in memory
    replace next page (in clock order), subject to same rules

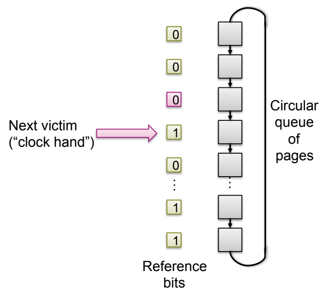
Second-chance (clock) page replacement algorithm



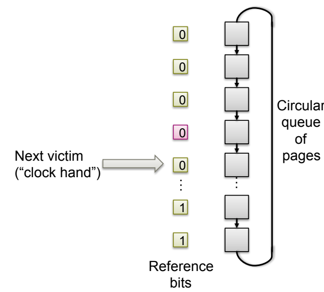
Second-chance (clock) page replacement algorithm



Second-chance (clock) page replacement algorithm



Second-chance (clock) page replacement algorithm



Frame allocation policies

Allocation of frames

- **Each process needs minimum number of pages**
- **Example: IBM 370 – 6 pages to handle SS MOVE instruction:**
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- **Two major allocation schemes**
  - fixed allocation
  - priority allocation

Fixed allocation

- **Equal allocation**
  - all processes get equal share.
- **Proportional allocation**
  - Allocate according to the size of process

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority allocation

- **Proportional allocation scheme**
- **Using priorities rather than size**
- **If process P_i generates a page fault, select:**
  1. one of its frames, or
  2. frame from a process with lower priority

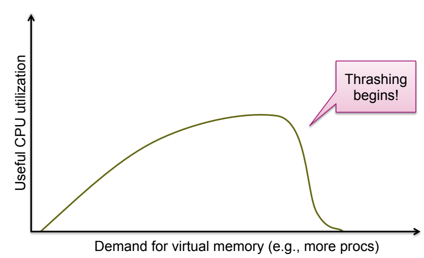
Global vs. local allocation

- **Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another**
- **Local replacement – each process selects from only its own set of allocated frames**

Thrashing

- **If a process does not have "enough" pages, the page-fault rate is very high. This leads to:**
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing ≡ a process is busy swapping pages in and out**

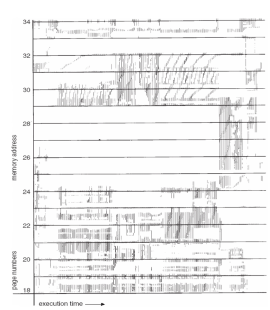
Thrashing



Demand paging and thrashing

- **Why does demand paging work?**
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- **Why does thrashing occur?**
  Σ size of locality > total memory size

Locality in a memory reference pattern



Working-set model

- **Δ ≡ working-set window**
  ≡ a fixed number of page references
  - Example: 10,000 instruction
- **WSS_i (working set of Process P_i) =**
  total number of pages referenced in the most recent Δ (varies in time)
  - Δ too small ⇒ will not encompass entire locality
  - Δ too large ⇒ will encompass several localities
  - Δ = ∞ ⇒ will encompass entire program

Allocate *demand frames*

- **D = Σ WSS_i ≡ total demand frames**
  - Intuition: how much space is really needed
- **D > m ⇒ Thrashing**
- **Policy: if D > m, suspend some processes**

Working-set model

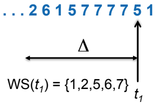
Page reference string:

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4

Working-set model

Page reference string:

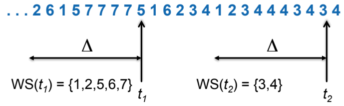
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4



Working-set model

Page reference string:

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4



Keeping track of the working set

- **Approximate with interval timer + a reference bit**
- **Example: Δ = 10,000**
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts shift+copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- **Why is this not completely accurate?**
  - Hint: Nyquist-Shannon!

Keeping track of the working set

- **Approximate with interval timer + a reference bit**
- **Example: Δ = 10,000**
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts shift+copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- **Why is this not completely accurate?**
- **Improvement = 10 bits and interrupt every 1000 time units**

Page-fault frequency scheme

- **Establish "acceptable" page-fault rate**
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame