VIRTUAL MEMORY

the ultimate abstraction
VIRTUAL MEMORY

“In operating systems, when you see the word *virtual* substitute the word *slow*”
VIRTUAL MEMORY

The basic idea is to treat physical memory as a cache for the address space of a computer.
physical memory  →  Swap Space

Secondary Storage
Physical memory

Fast

Secondary Storage

Swap Space

Slow
Virtual Memory

※ invented in late 60s / early 70s -- memory > $10K/M
※ today memory < $0.10 --> less important to oversubscribe.
※ 70s - disk a lot slower than CPU or memory
※ today - disk much much much much much much much much slower
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute typical instruction</td>
<td>1 nanosecond</td>
</tr>
<tr>
<td>fetch from L1 cache</td>
<td>.5 nanoseconds</td>
</tr>
<tr>
<td>fetch from L2 cache</td>
<td>7 nanosec</td>
</tr>
<tr>
<td>fetch from main memory</td>
<td>100 nanosec</td>
</tr>
<tr>
<td>send 2k bytes over 1Gbps network</td>
<td>20,000 nanosec</td>
</tr>
<tr>
<td>read 1MB sequentially from memory</td>
<td>250,000 nanosec</td>
</tr>
<tr>
<td>fetch from new disk location</td>
<td>8,000,000 nanosec</td>
</tr>
<tr>
<td>read 1MB sequentially from disk</td>
<td>20,000,000 nanosec</td>
</tr>
<tr>
<td>send packet US to Europe and back</td>
<td>150,000,000 nanosec</td>
</tr>
</tbody>
</table>
Virtual Memory

* invented in late 60s / early 70s -- memory > $10K/M
* today memory < $0.10 -> less important to oversubscribe.
* 70s - disk a lot slower than CPU or memory
* today - disk much much much much much much much slower
* still, its convenient - can start 100s of shells @ 1 MB each w/o worrying.
My Macbook...
Virtual Memory

★ in 70s - difficult to invent

★ now that we know how to do it, not that hard so worth having around.
virtual memory

- demand paging
- process creation
- page replacement
- allocation of frames
- thrashing
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 address spaces to be shared among processes
- allows more programs to run
- allows for more efficient process creation
implementation

- can be implemented via
  - demand paging
  - demand segmentation
Virtual Memory can be larger than Physical M.
Virtual Address Space
VM has many uses

- separates logical from physical memory (abstraction)
- system libraries can be shared
- it can enable processes to share memory
- allow pages to be shared during process creation with fork()}
shared library using virtual memory
DEMAND PAGING

bring a page into memory only when it is needed
Demand Paging

- Less I/O needed
- less memory needed
- faster response
- more users / more applications
LAZY SWAPPER

In a paging system, processes reside in a disk. The whole process will not be placed into memory.
A LAZY SWAPPER

never swaps a page into memory unless that page will be needed.
A LAZY SWAPPER

swaps out unwanted pages onto the disk.
Transfer of a paged memory to contiguous disk space
need hardware support to implement page table
need hardware support to implement page table

<table>
<thead>
<tr>
<th>frame</th>
<th>valid - invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>
need hardware support to implement

<table>
<thead>
<tr>
<th>frame</th>
<th>valid - invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

initially all pages marked as invalid
need hardware support
to implement

<table>
<thead>
<tr>
<th>frame</th>
<th>valid - invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

initially all pages marked as invalid

during address translation, if bit 0 we page fault.
page table when some pages are not in main memory
page fault

* if there is ever a reference to a page, first reference will trap to OS (= page fault)

* OS looks @ another table to decide
  * invalid reference -> abort
  * just not in memory

* get empty frame

* swap page into frame

* reset table, validation bit = 1

* restart instruction
0091    movl 0x0092 %ecx
0094    movl 0x007b %edx
steps in handling page fault

1. reference
2. trap
3. page is on backing store
4. bring in missing page
5. reset page table
6. restart instruction

load M

physical memory

page table

free frame
implementation issues

- what happens if page is written?

- **write through** - send write immediately to lower level (disk)

- **write back** - send write to lower level when page evicted from higher level

- Which should we use here?

- How would we know a page needs to be written back?
dirty bit

* implemented in TLB - when TLB sees a write request to a page, it sets the dirty bit in TLB, when evicted from TLB need to copy dirty bit to page table and core map
what happens if there is no free frame?

* page replacement - find some page in memory, but not really in use, swap it out.
* need algorithm, that results in minimum number of page faults.
* same page may be brought into memory several times.
these schemes require

TEMPORAL LOCALITY
HVAC
HVAC
temporal locality
Initialization
Servicing requests
syncing w/ master server
temporal locality
performance

* page fault rate $0 < p < 1$
  * if $p = 0 \implies$ no page faults
  * if $p = 1 \implies$ every reference is a page fault
Effective Access Time

* $ma =$ memory access time typically 50 - 100 ns (if in L2 ~5)
* $pft =$ page fault time

$$EAT = (1 - p) ma + p(pft)$$

$pft = ?$
PFT

- time to trap to OS (save registers, determine input was page fault, etc.)
- swap page out (wait in queue until write, seek & latency time of HD)
- swap page in (wait in queue until read, seek & latency time of HD, transfer page to free frame)
- restart overhead.
example

- memory access time = 100 nanoseconds
- avg page fault service time 10 milliseconds
- EAT = $(1 - p)100 + p(10,000,000)$
- let’s say one access out of 1,000 leads to a page fault
EAT = (1 - p)100 + p(10,000,000)

EAT = (.999) 100 + .001(10,000,000)
    = 99.9 + 10000
    = 10099.9
\[ \text{EAT} = (1-p)\text{i}00 + p(10,000,000) \]

\[ \text{EAT} = (0.999)\text{i}00 + \text{.001}(10,000,000) \]

\[ = 99.9 + 10000 \]

\[ = 10099.9 \]

\[ \text{compared to no page faults} = 100\text{ns} \]

\[ \text{slowed down the computer by a factor of 100.} \]
example

• if I want only 10% degradation...

need one fault out of ?????
if I want only 10% degradation...

\[ 1.1(t_{\text{mem}}) = (1-p)t_{\text{mem}} + p(t_{\text{disk}}) \]

\[ p = (1.1 \times t_{\text{mem}}) / (t_{\text{mem}} + t_{\text{disk}}) \]

\[ \approx (1.1 \times 10^2) / (10^7 + 10^2) \]

\[ \approx 10^{-6} \]

at most one access out of 1,000,000 can be a page fault. (hit rate greater than 99.9999%)
VM BENEFITS DURING PROCESS CREATION

copy-on-write
copy-on-write

* both parent and child process initially share the same pages in memory
* if either modifies a shared page, only then it is copied.
* allows for efficient process creation as only modified pages are copied
* used by Windows, Linux, Solaris, Mac OS X
OVER-ALLOCATING MEMORY

when we increase multiprogramming we ‘overbook’ memory

(over-allocating memory)
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.
need for page replacement
basic page replacement

- find the location of the desired page on the disk
- find a free frame:
  - if there is a free frame, use it
  - if there is no free frame, use a page replacement algorithm to select the victim frame.
- read the desired page into the (newly) free frame. Update the page and frame tables.
- restart the process
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.

* in all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
team work

* come up with a page replacement algorithm
* assume a memory sizes of 1, 2, 3, 4, and 5 frames
* compute page faults

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>cont'd</td>
</tr>
</tbody>
</table>
Graph of page faults vs. number of frames
WHAT IS THE OPTIMAL SOLUTION?

How do you know it is optimal?
deliverables

* for ea. algorithm
  * for ea. memory size (1-5)
    * diagram showing memory contents @ ea. state of reference string (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5)
  * number of page faults
* graph
Optimal Algorithm?
Optimal Algorithm

- replace page that will not be used for the longest period of time.
- (4 frame example) -6 page faults
- used for measuring how well other algorithms perform
optimal page replacement
results -- Who did...

- FIFO
- Least recently used
- what others?
First-in-first-out FIFO

- reference string 1 2 3 4 1 2 5 1 2 3 4 5
- 3 frames: 9 page faults
- 4 frames: 10 page faults
BELADY’S ANOMALY
FIFO illustrating Belady’s Anomaly
LRU DEMO

memory size = 4, 3
LRU page replacement
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
use of a stack to record the most recent page refs.
LRU

⋆ is the most used page replacement algorithm
⋆ does not exhibit Belady’s anomaly
⋆ LRU belongs to a class of page replacement algorithms called stack algorithms
⋆ stack algorithms never exhibit Belady’s anomaly
⋆ a stack algorithm is an algorithm for which it can be shown that the set of pages in memory for $n$ frames is always a subset of the set of pages that are in memory with $n + 1$ frames
LRU

- Few Computer Systems provide sufficient hardware support for true LRU page replacements
LRU approximation

Algorithms

* base case: reference bit
* with each page associate a bit, initially 0
* when page is referenced bit set to 1
* replace the one which is 0 if one exists
additional reference bit algorithm

※ have 8 bit byte for each page
※ at certain interval (100ms) interrupt and transfer control to OS
※ shift bits to right. add current reference bit on left.
※ use frame with smallest number.
<table>
<thead>
<tr>
<th>fr 0</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr 1</td>
<td>o</td>
</tr>
<tr>
<td>fr 2</td>
<td>o</td>
</tr>
<tr>
<td>fr 3</td>
<td>i</td>
</tr>
<tr>
<td>fr 4</td>
<td>i</td>
</tr>
<tr>
<td>fr 5</td>
<td>o</td>
</tr>
<tr>
<td>fr 6</td>
<td>o</td>
</tr>
<tr>
<td>fr 7</td>
<td>i</td>
</tr>
</tbody>
</table>
### add current reference bit

<table>
<thead>
<tr>
<th>fr 0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fr 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fr 3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>fr 4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fr 5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>fr 6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fr 7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fr 0</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>fr 1</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>fr 2</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>fr 3</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>fr 4</td>
<td>O</td>
<td>I</td>
</tr>
<tr>
<td>fr 5</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>fr 6</td>
<td>I</td>
<td>O</td>
</tr>
<tr>
<td>fr 7</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>fr 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>fr 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fr 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>fr 3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fr 4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fr 5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fr 6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fr 7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**replace frame with smallest number**

Monday, October 29, 12
LRU approximation algorithms

- second chance algorithm
- a FIFO algorithm with a twist
- need a reference bit
- Think of a circular queue of pages (a clock)
Second Chance Algorithm

# examine pages in clock order
if page.refBit == 1:
    page.refBit = 0
    # leave page in memory
    # replace next page subject to
    # same rules.
else: # refBit == 0
    replace page
second chance algorithm
YOU TRY

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

memory sizes of 3 and 4 frames
enhanced 2nd chance

- use both reference and modify bits. With these bits as ordered pair we have the following 4 cases:

- \((0, 0)\) neither recently used nor modified - best page to replace.

- \((0, 1)\) not recently used but modified - not quite as good since page needs to be written out before replacement.

- \((1, 0)\) recently used but clean. it will probably be used again.

- \((1, 1)\) recently used and modified.

- replace 1st page encountered in the lowest non-empty class
counting algorithms

* keep counter of the number of references that have been made to each page.

* **LFU algorithm (least frequently used):** replaces page with smallest count.

* **MFU algorithm (most frequently used):** based on the argument that the page with the smallest count was probably just brought in.

*Neither algorithm used much.*
trade off

* clever algorithms which may work well
* clever algorithms are probably expensive
page buffering algorithms

- always maintain a pool of free frames
- when a page fault occurs you assign the desired page to a free frame from the pool
- at the same time you find a victim, write out the page, and put the victim frame in the pool.
</ REPLACEMENT ALGORITHMS>
ALLOCATION OF FRAMES

how do we allocate free memory?
ALLOCATION POLICY

How should memory be allocated among competing runnable processes?
ALLOCATION POLICY

say our page replacement algorithm is LRU

we have 3 processes A, B, and C
<table>
<thead>
<tr>
<th>process</th>
<th>age (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>10</td>
</tr>
<tr>
<td>A₁</td>
<td>7</td>
</tr>
<tr>
<td>A₂</td>
<td>5</td>
</tr>
<tr>
<td>A₃</td>
<td>4</td>
</tr>
<tr>
<td>A₄</td>
<td>6</td>
</tr>
<tr>
<td>A₅</td>
<td>3</td>
</tr>
<tr>
<td>B₀</td>
<td>9</td>
</tr>
<tr>
<td>B₁</td>
<td>4</td>
</tr>
<tr>
<td>B₂</td>
<td>6</td>
</tr>
<tr>
<td>B₃</td>
<td>2</td>
</tr>
<tr>
<td>B₄</td>
<td>5</td>
</tr>
<tr>
<td>B₅</td>
<td>6</td>
</tr>
<tr>
<td>B₆</td>
<td>12</td>
</tr>
<tr>
<td>C₁</td>
<td>3</td>
</tr>
<tr>
<td>C₂</td>
<td>5</td>
</tr>
<tr>
<td>C₃</td>
<td>6</td>
</tr>
</tbody>
</table>
2 strategies

- global: in using LRU consider all pages in memory
- local: in using LRU consider only pages for current process
consider A page faults requesting A6.

For global policy which frame gets replaced?

<table>
<thead>
<tr>
<th>process</th>
<th>age(time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>10</td>
</tr>
<tr>
<td>A₁</td>
<td>7</td>
</tr>
<tr>
<td>A₂</td>
<td>5</td>
</tr>
<tr>
<td>A₃</td>
<td>4</td>
</tr>
<tr>
<td>A₄</td>
<td>6</td>
</tr>
<tr>
<td>A₅</td>
<td>3</td>
</tr>
<tr>
<td>B₀</td>
<td>9</td>
</tr>
<tr>
<td>B₁</td>
<td>4</td>
</tr>
<tr>
<td>B₂</td>
<td>6</td>
</tr>
<tr>
<td>B₃</td>
<td>2</td>
</tr>
<tr>
<td>B₄</td>
<td>5</td>
</tr>
<tr>
<td>B₅</td>
<td>6</td>
</tr>
<tr>
<td>B₆</td>
<td>12</td>
</tr>
<tr>
<td>C₁</td>
<td>3</td>
</tr>
<tr>
<td>C₂</td>
<td>5</td>
</tr>
<tr>
<td>C₃</td>
<td>6</td>
</tr>
</tbody>
</table>
worst case thinking

※ let’s say we have move instruction
※ move x to z
※ worst case, how many pages might we need?
allocation of frames

-* each process needs minimum number of pages

-* some machines may need up to 6 pages to handle a single 2 operand instruction.

-* the instruction may straddle a page boundary

-* ea. operand may also straddle

-* (for ex., IBM 370)

-* if you allocation the process 5 frames the process cannot run.
2 major types of allocation

* Fixed: assign certain number of frames to a process and that number stays constant.

* Variable allocation
fixed allocation

* equal allocation - if 100 frames and 5 processes - give each 20 pages

* proportional allocation - allocate according to the size of the process
use a proportional allocation scheme using priorities rather than size.

if process P generates a page fault, select for replacement:

- one of its frames
- a frame from a process with lower priority
Global replacement schemes the most common method.
Consider LRU and memory size 3
What can you say about the page fault behavior?
If a process is spending more time paging than executing, the process is said to be ...
THRASHING
Thrashing

- if a process does not have ‘enough’ pages, the page-fault rate is very high and leads to
  - low CPU utilization
  - OS thinks that it needs to increase multiprogramming
  - another process is added to the system
Thrashing

* a process is busy swapping pages in and out
* not terrible if one process thrashing
* it all processes are thrashing we got a problem
Thrashing
no work gets done
processes spending their time page faulting.
locality model

★ one method of determining this need is the locality model.

★ the model states that as a process executes, it moves from one locality to another

★ a locality is a set of pages that are actively used together

★ a program contains several different localities which may overlap

★ assume access is not random

★ examples: text processing / compiling / etc.
locality in a memory-reference pattern
locality and thrashing

- why does paging work
- access not random pattern
- locality model
  - process migrates from one locality to another
- localities may overlap
why does thrashing occur

\[ \Sigma \text{ size of locality > total memory size} \]
working-set model

* Δ a working-set window - a fixed number of page references. For ex., 10,000 instructions

* WSS (working set of a process) = total number of pages referenced in the most recent Δ

  * if Δ too small - will not encompass entire locality
  * if Δ too large will encompass several localities
  * if Δ huge will encompass entire program

* D = Σ WSS = total frame demand
working set model

* OS monitors the working set of each process.
* it allocates to each process enough frames to fulfill its working set requirements
* if $D > m \implies$ thrashing
* Policy: if $D > m$ then suspend one of the processes
* thus preventing thrashing
working set model
keeping track of the working set

* approximate with interval time + reference bit

* example $\Delta = 10,000$

* timer interrupts after every 5,000 time units

* keep in memory 2 bits for ea. page

* whenever a timer interrupts copy and set the values of all reference bits to 0

* if one of the bits in memory = 1 $\Rightarrow$ page in working set

* improvement: 10 bits and interrupt every 1000 time units (however, cost of interrupts higher)
ANOTHER ANTI-THRASHING TECHNIQUE

page-fault frequency scheme
page fault frequency

• establish acceptable range for page faults
• if actual rate too low, process loses frame
• if too high, process gains frame
page fault frequency
demand paging is meant to be transparent but...

program 1
```
int A[][] = new int[1024][1024]
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        A[i, j] = 0
```

program 2
```
int A[][] = new int[1024][1024]
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        A[j, i] = 0
```

1024 x 1024 page faults

1024 page faults