Virtual Memory: Policies

CS439: Principles of Computer Systems
March 11, 2019
Bringing It Together

• Processes are the OS’s main abstraction for protection
• A process defines an address space---but this is a virtual/logical address space
• Virtual address spaces are mapped onto physical address spaces in physical memory
• All virtual addresses range from 0->max for that process
• Paging describes how the OS provides the illusion of infinite memory
Today’s Additions

• Paging: Mechanisms
  – Issues of Efficiency: Space and Time

• Paging: Policy
  – Demand paging vs. pre-paging
  – Page Replacement Algorithms
  – Global vs. local
  – Page Sizes

• Multiprogramming
  – Swap
A system with 16-bit addresses

- 32 KB of physical memory
- 1024 byte pages
Virtual Address Translation:
Text Description

Steps to Virtual-Physical Memory Translation:

• The program gives a virtual address to the CPU to translate
• The MMU splits the virtual address into its page and offset numbers
• Since the offset is the same in Virtual and Physical memory it is sent along with no change
• The page number is translated into a frame number
  – Look into page table to find out if page exists in physical memory
    • The page number is the index of the correct entry in the page table (much like indexing an array)
  – If the page exists in memory the frame number is sent along
  – If it doesn’t exist in memory but exists on disk, the corresponding page is moved from disk into physical memory and its frame number is recorded
• Append offset to end of frame number to get the full physical address
Performance Issues: Speed

Problem: Virtual memory references require two memory references!
– One access to get the page table entry
– One access to get the data
Back to Architecture

Translation Lookaside Buffer/Memory Management Unit
Back to Architecture:  
Text Description

• There is a system bus that connects to the CPU, the Disk Controller, and the Memory Controller. It controls the flow of information between them.
• The Disk Controller connects to disks and allows the System Bus to interface with them. It creates the abstraction between the system and the disk hardware.
• The Memory Controller connects to memory and controls the abstraction between the system and the memory hardware.
• The System Bus interacts with the CPU: it gets a Physical Address from the TLB/MMU and sends/receives data from the Processor and/or Cache.
• The TLB (Translation Lookaside Buffer)/MMU (Memory Management Unit) are an intermediary between the System Bus and Processor for Physical Addresses. The Processor sends Virtual Addresses to the TLB/MMU, they are translated into physical addresses and then sent along the System Bus. The TLB is a cache for these translations, as we will see next.
Improving Speed:
Translation Lookaside Buffers (TLBs)

Cache recently accessed page-to-frame translations in a TLB
- For TLB hit, physical page number obtained in 1 cycle
- For TLB miss, translation is updated in TLB
- Has high hit ratio (why?)
Improving Speed: 
Translation Lookaside Buffers (TLBs) 
Text Description

• The TLB adds an extra step to getting a frame number from a page number.
  – The TLB holds recently used frame/page pairings.
  – TLBs have high hit ratio because they exploit locality within the system
  – For a TLB hit, the translation can be completed in 1 cycle

• The system simultaneously sends the page number to both the TLB and the page table looking for a frame number. If there is a hit in the TLB then it stops looking in the page table and sends the frame number along. If there is a miss in the TLB then it finds the frame number in the page table, sends the frame number along, and updates the TLB.
Performance Issues: Space

• Two sources:
  – data structure overhead (the page table)
  – fragmentation (How large should a page be?)

• Page table can be very large!
  – 32-bit addresses, 4096-byte pages... 1 million pages!
  • 32-bit addresses imply a VAS of $2^{32}$ bytes
  • Divide VAS into pages ($2^{32}/2^{12}$)
  • 1 million pages ($2^{20}$)
Dealing With Large Page Tables: Multi-Level Paging

Add additional levels of indirection to the page table by sub-dividing page number into $k$ parts
- Create a “tree” of page tables
- TLB still used, just not shown
- The architecture determines the number of levels of page table

Virtual Address

First-Level Page Table

Second-Level Page Tables

Third-Level Page Tables
Dealing with Large Page Tables: Multi-Level Paging

Text Description

• Add additional levels of indirection to the page table by subdividing page number into k parts. This creates a k-depth “tree” of page tables.
  – The architecture of the system determines k.

• Example: A system with 3 levels of page tables:
  – There are 3 bits for the first level page table
  – There are 4 bits for the second level page table
  – There are 5 bits for the third level page table
  – Note that if this were a single-level paged system, there would be 12 (3+4+5) bits for the page number

• Each entry in the first level page table points to a second level page table, and each entry in the second level page table points to a third level page table. The third level page table entries actually hold the bit/frame number information that is usually associated with a page table entry.

• Multi-Level page tables save space because you only have to allocate space for the page-table levels that you need. For example if you insanely only needed to hold information for 1 page table entry you would only need one first level page table, one second level table and one third level page table. It is important to note that you will always have only one first level page table.
Multi-Level Paging: Example

Example: Two-level paging
Multi-Level Paging: Example Text Description

• This example has two-level paging
• The first 3 bits are for the first level page table. The value of these bits is added to the PTBR (Page Table Base Register) which points to the beginning of the first level page table.
• The next 4 bits are for the second level page table. From the first level page table entry we get the address of the beginning of the second level page table, then we add the value from the second-level bits in the virtual address to this address. This gives us the address of the corresponding page table entry in the second level page table. Now we can access the relevant bits and frame number from the right second level page table entry.
• The TLB is very important now because each lookup from the page table requires a 2 step look up instead of just a single step.
64-bit Addresses

• Multi-level page tables work decently well for 32-bit address spaces... but what happens when we get to 64-bit?

• Too cumbersome (5 levels?)

• Rethink the page table:
  – Instead of making tables proportional to the size of the VAS, make them proportional to the size of the physical address space
  – One entry for each physical page---and hash!

Result: Inverted Page tables
Inverted Page Tables

• Each frame is associated with an entry
• Entry contains:
  – Residence bit: is the frame occupied?
  – Occupier: page number of the page in the frame
  – Protection bits
Inverted Page Tables: Virtual Address Translation

• As the CPU generates virtual addresses, where is the physical page?
  – Must search entire page table (32KB of Registers!)
  – Uh-Oh

• How can we solve that problem?
  – Cache it! (How did you know?)
  – Use the TLB to hold all the heavily used pages
  – But, um... the TLB is limited in size
  – So what about a miss in the TLB?
Use a Hash Table!

• Page $i$ is placed in slot $f(i)$ where $f$ is an agreed-upon hash function
  – Deal with collisions using linked list or rehashing

• To look up page $i$, perform the following:
  – Compute $f(i)$ and use it as an index into the table of page entries
  – Extract the corresponding page entry
Hashed Inverted Table In Use

Hash page numbers to find corresponding frame number
- Page frame number is not explicitly stored (1 frame per entry)
- Protection, dirty, used, resident bits also in entry

![Diagram of Hashed Inverted Table]

- CPU
- Virtual Address
- PID (running)
- Hash
- PTBR
- Inverted Page Table
- Memory
- Physical Addresses
- Tag check
- $h(PID, p)$
Hashed Inverted Table In Use: Text Description

- Hash page numbers to find corresponding frame number
  - page frame number is not explicitly stored (1 frame per entry)
  - protection, dirty, used, and resident bits also in entry
- The hash function takes in the PID of the process and the page number. The answer from this hash function is added to the PTBR to get the address of the right page table entry.
- Once at the right page table entry the PID and page number saved in the entry are checked against the given PID and page number. If they do not match, then the frame isn’t holding information for that page and it needs to be swapped in. If they do match then the right frame has been found and the frame number is sent along.
iClicker Question

Why use hashed/inverted page tables?

A. Regular (forward-mapped) page tables are too slow
B. Forward-mapped page tables don’t scale to larger virtual address spaces
C. Inverted page tables have a simpler lookup algorithm, so the hardware that implements them is simpler
D. Inverted page tables allow a virtual page to be anywhere in physical memory
Virtual Memory: The Bigger Picture

• A process’s VAS is part of its context contains its code, data, and stack

• Code pages are stored in a user’s file on disk
  – Some are currently residing in memory; most are not

• Data and stack pages are also stored in a file
  – Although this file is typically not visible to users
  – File only exists while a program is executing

• OS determines which portions of a process’s VAS are mapped in memory at any one time
Page Faults

References to non-mapped pages generate a *page fault*

Page fault handling steps:
1. Processor runs the interrupt handler
2. Page fault handler determines the reason for the page fault (exception or not resident)
3. If exception, process is killed
4. If not resident, it is a true page fault
5. Handler locates unmapped page and initiates the read of that page
6. Faulting process is blocked and another process is scheduled
7. Read of page completes, page is mapped into physical memory
8. Faulting process is unblocked
Paging: Policies
Paging: So Many Questions

• When should a process’s pages be loaded into memory?
• If memory is full but a page fault has just occurred (so a process needs another page!), what should happen?
  – If a page should be replaced, which one?
• What is a good page size?
• How many processes are too many?
Loading Pages

- **Demand paging**: A page is loaded the first time it is referenced
- **Pre-paging**: OS guesses in advance which pages the process will need and pre-loads them
What happens when we have a page fault.... but memory is full?
Page Replacement Algorithms: Goals

Improve OS performance by:
  – Reducing the frequency of page faults
  – Being efficient

Overarching goal: *Make good decisions quickly*
Page Replacement Algorithms: Optimal

- Look into the future and throw out the page that will be accessed farthest in the future
- Provably optimal
**Optimal Page Replacement**

**Clairvoyant replacement**

- Replace the page that won’t be needed for the longest time in the future

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time page needed next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimal Page Replacement: Text Description

- Clairvoyant replacement---impossible in real life
- Replace the page that won’t be needed for the longest time in the future
- Time and requests on x axis
- Page frames on the y axis
- Keep track of faults as you go
- At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.
- The request string is c, a, d, b, e, b, a, b, c, d.
- Each request occurs at another time tick, so requests are at times 1-10.
Optimal Page Replacement: Solution

Clairvoyant replacement

- Replace the page that won’t be needed for the longest time in the future

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

Faults: •

Time page needed next:
- a = 7
- b = 6
- c = 9
- d = 10
- a = 15
- b = 11
- c = 13
- d = 14
Optimal Page Replacement: Solution

**Text Description**

- **Performance with 4 frames and 10 requests:**
  - Request order: c a d b e b a b c d
    - We will keep this order in mind as we run the algorithm
  - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - Time 1: request page c, in memory so no page fault
  - Time 2: request page a, in memory so no page fault
  - Time 3: request page d, in memory so no page fault
  - Time 4: request page b, in memory so no page fault
  - Time 5: request page e which isn’t in memory, causes a page fault
    - Look at future requests: b a b c d, see that d is the one farthest in the future
    - Put e where d used to be
    - Current frame setup: a in frame 0, b in frame 1, c in frame 2 and e in frame 3
  - Time 6: request page b, in memory so no page fault
  - Time 7: request page a, in memory so no page fault
  - Time 8: request page b, in memory so no page fault
  - Time 9: request page c, in memory so no page fault
  - Time 10: request page d which isn’t in memory, causes a page fault
    - Don’t have any future requests so just pick something to kick out
    - Put d where a used to be
    - Final frame setup: d in frame 0, b in frame 1, c in frame 2 and e in frame 3
  - Total number of page faults: 2
Page Replacement Algorithms: LRU

*Least Recently Used*

- Works well when the recent past is a good predictor of the future
- Throw out the page that has not been used in the longest time
Least Recently Used Page Replacement
Use the recent past as a predictor of the near future

- Replace the page that hasn’t been referenced for the longest time

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faults</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Time page last used</th>
</tr>
</thead>
</table>
Least Recently Used Page Replacement Algorithm:

Text Description

- Use the recent past as a predictor of the near future
- Replace the page that hasn’t been referenced for the longest time
- Time and requests on x axis
- Page frames on the y axis
- Keep track of faults as you go
- At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.
- The request string is c, a, d, b, e, b, a, b, c, d.
- Each request occurs at another time tick, so requests are at times 1-10.
Least Recently Used Page Replacement: Solution

Use the recent past as a predictor of the near future

- Replace the page that hasn’t been referenced for the longest time

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a = 2</td>
<td>a = 7</td>
<td>a = 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time page last used</td>
<td>b = 4</td>
<td>b = 8</td>
<td>b = 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c = 1</td>
<td>e = 5</td>
<td>e = 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d = 3</td>
<td>d = 3</td>
<td>c = 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Least Recently Used Page Replacement: Solution

Text Description

- Performance with 4 frames and 10 requests:
  - request order: c a d b e b a b c d
  - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - Time 1: request page c, in memory so no page fault
  - Time 2: request page a, in memory so no page fault
  - Time 3: request page b, in memory so no page fault
  - Time 4: request page d, in memory so no page fault
  - Time 5: request page e which isn’t in memory, causes a page fault
    - look back in time and see that c was used least recently
    - past requests: a used at time 2, b used at time 4, c used at time 1 and d used at time 3
    - put e where c used to be
    - current frame setup: a in frame 0, b in frame 1, e in frame 2 and d in frame 3
  - Time 6: request page b, in memory so no page fault
  - Time 7: request page a, in memory so no page fault
  - Time 8: request page b, in memory so no page fault
  - Time 9: request page c which isn’t in memory, causes a page fault
    - look back in time and see that d was used least recently
    - past requests: a used at time 7, b used at time 8, e used at time 5 and d used at time 3
    - put c where d used to be
    - current frame setup: a in frame 0, b in frame 1, e in frame 2 and c in frame 3
  - Time 10: request page d which isn’t in memory, causes a page fault
    - look back in time and see that e was used least recently
    - past requests: a used at time 7, b used at time 8, e used at time 5 and c used at time 9
    - put d where e used to be
    - current frame setup: a in frame 0, b in frame 1, d in frame 2 and c in frame 3
    - NOTE: replaced d and had to immediately bring it back in. So the past is not a perfect prediction of the future
  - Total number of page faults: 3
Implementing LRU

Option 1: keep a time stamp for each page representing the last access

Problem: OS must record time stamp for each memory access and search all pages to find one to toss

Option 2: keep a list of pages, where the front of the list is the most recently used and the end is the least recently used. Move page to front on access. Doubly link the list.

Problem: Still too expensive, since OS must modify up to 6 pointers on memory access
iClicker Question

What is the goal of a page replacement algorithm?

A. Reduce the number of page faults
B. Reduce the penalty for page faults when they do occur
C. Minimize CPU time for a process
Approximating LRU: Clock

• Maintain a circular list of pages in memory
• Maintain pointer (clock hand) to oldest page
• Before replacing a page, check its reference bit
  – Remember the reference/clock bit?
  – It tracks whether the page was referenced recently
• If the reference bit is 1 (was referenced recently), clear bit and then check next page
• How does this algorithm terminate?
Clock

- Clock hand points to oldest page
- Clock hand sweeps over pages looking for one with reference bit = 0
  - Replace pages that haven’t been referenced for one complete revolution of the clock

func Clock_Replacement
begin
  while (victim page not found) do
    if (reference bit for cur page = 0) then
      replace current page
    else
      reset reference bit
      advance clock pointer
  end while
end

Page 0:
\[\begin{array}{c}
111\\
110
\end{array}\]

Page 1:
\[\begin{array}{c}
105
\end{array}\]

Page 2:
\[\begin{array}{c}
111
\end{array}\]

Page 3:
\[\begin{array}{c}
111
\end{array}\]

Page 4:
\[\begin{array}{c}
103
\end{array}\]

Page 5:
\[\begin{array}{c}
114
\end{array}\]

Page 6:
\[\begin{array}{c}
114
\end{array}\]

Page 7:
\[\begin{array}{c}
110
\end{array}\]

resident bit
reference bit
frame number
Clock: Text Description

- Pages are arranged in a (logical) circle
- Each page around the clock face keeps track of 3 bits: resident, reference and frame number
- Clock hand points to the oldest page
- Clock hand sweeps over pages looking for one with reference bit that is 0
  - Replace pages that haven’t been referenced for one complete revolution of the clock
- Pseudocode for algorithm:
  - func clock_replacement
    - begin
    - while(victim page not found) do
      - if(reference bit for current page is 0) then
        » replace current page
      - else
        » reset reference bit
      - advance clock pointer
    - end while
    - end clock_replacement
- Can be implemented with a circular list of all resident pages from the page table
- Is an approximation of the LRU algorithm
# Clock Page Replacement

## Example

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Faults

## Page table entries for resident pages:

<table>
<thead>
<tr>
<th>Page table entries for resident pages:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Clock Page Replacement: Example Text Description

• Same setup as earlier examples
• Time and requests on x axis
• Page frames on the y axis
• Keep track of faults as you go
• At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.
• The request string is c, a, d, b, e, b, a, b, c, d.
• Each request occurs at another time tick, so requests are at times 1-10.
• In earlier examples, we learned that no page replacements were necessary until time 5, so this example starts at time 5.
### Clock Page Replacement

#### Example: Solution

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>Requests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

#### Faults

- Time 1: Request is for frame 0 (d)
- Time 2: Request is for frame 1 (b)
- Time 3: Request is for frame 2 (c)

#### Page table entries for resident pages:

- Page table entries for resident pages:
  - Frame 0: Pages a, b, c, d
  - Frame 1: Pages e
  - Frame 2: Pages a, a, a (a repeated)
  - Frame 3: Pages d (d repeated)

- Frame 4: Pages b, b, b (b repeated)
- Frame 5: Pages a, a, a (a repeated)
- Frame 6: Pages d, d, d (d repeated)
- Frame 7: Pages e, e, e (e repeated)
- Frame 8: Pages b, b, b (b repeated)
- Frame 9: Pages a, a, a (a repeated)
- Frame 10: Pages d, d, d (d repeated)
Clock Page Replacement Example Solution: Text Description (1 of 2)

- Performance with 4 frames and 10 requests:
  - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
    - clock pointer at frame 0
  - Time 1: request page c, in memory so no page fault
    - update reference bit to be 1
  - Time 2: request page a, in memory so no page fault
    - update reference bit to be 1
  - Time 3: request page b, in memory so no page fault
    - update reference bit to be 1
  - Time 4: request page d, in memory so no page fault
    - update reference bit to be 1
  - Time 5: request page e which isn’t in memory, causes a page fault
    - have to run the clock algorithm
      - frame 0 (a) has reference bit of one, clear it and move pointer to frame 1
      - frame 1 (b) has reference bit of one, clear it and move pointer to frame 2
      - frame 2 (c) has reference bit of one, clear it and move pointer to frame 3
      - frame 3 (d) has reference bit of one, clear it and move pointer back to frame 0
      - since we cleared all the bits as we went frame 0 now has a reference bit of zero, this is the page to replace
      - put e in frame 0 and move pointer to frame 1
  - current frame setup: e in frame 0, b in frame 1, c in frame 2 and d in frame 3
Clock Page Replacement Example Solution: Text Description (2 of 2)

– Time 6: request page b which is in memory, no page fault
  • frame 1’s reference bit is set to one since it holds page b
– Time 7: request page a which isn’t in memory, causes a page fault
  • have to run the clock algorithm
    – frame 1 (b) has reference bit of one, clear it and move pointer to frame 2
    – frame 2 (c) has reference bit of zero, this is the page to replace
    – put a in frame 2 and move pointer to frame 3
  • current frame setup: e in frame 0, b in frame 1, a in frame 2 and d in frame 3
– Time 8: request page b, in memory so no page fault
  • update reference bit to be 1
– Time 9: request page c which isn’t in memory, causes a page fault
  • have to run clock algorithm
    – frame 3 (d) has reference bit of zero, this is the page to replace
    – put d in frame 3 and move pointer to frame 0
  • current frame setup: e in frame 0, b in frame 1, a in frame 2 and c in frame 3
– Time 10: request page d which isn’t in memory, causes a page fault
  • have to run clock algorithm
    – frame 0 (e) has reference bit of one, clear it and move to frame 1
    – frame 1 (b) has reference bit of one, clear it and move to frame 2
    – frame 2 (a) has reference bit of one, clear it and move to frame 3
    – frame 3 (c) has reference bit of one, clear it and move to frame 0
    – since we cleared all the bits as we went frame 0 now has a reference bit of zero, this is the page to replace
    – put d in frame 0 and move pointer to frame 1
  • final memory setup: d in frame 0, b in frame 1, a in frame 2 and c in frame 3
– total number of page faults: 4
Second Chance

- Cheaper to replace a page that has not been written since it need not be written back to disk
- Check both the *reference bit* and *modify bit* to determine which page to replace
  - (reference, modify) pairs form classes:
    - (0,0): not used or modified, replace!
    - (0,1): not recently used but modified: OS needs to write, but may not be needed anymore
    - (1,0): recently used and unmodified: may be needed again soon, but doesn’t need to be written
    - (1,1): recently used and modified
- On page fault, OS searches for page in the lowest nonempty class
Second Chance Implementation

The OS goes around at most three times searching for the (0,0) class:

1. If the OS finds (0,0) it replaces that page
2. If the OS finds (0,1) it
   - initiates an I/O to write that page,
   - locks the page in memory until the I/O completes,
   - clears the modified bit, and
   - continues the search in parallel with the I/O
3. For pages with the reference bit set, the reference bit is cleared
4. On second pass (no page (0,0) found on first), pages that were (0,1) or (1,0) may have changed
Second Chance Implementation
(another option)

The OS goes around at most three times searching for the (0,0) class:

1. If the OS finds (0,0) it replaces that page
2. If the OS finds (0,1), clear dirty bit and move on, but remember page is dirty. Write only if evicted.
3. For pages with the reference bit set, the reference bit is cleared
4. On second pass (no page (0,0) found on first), pages that were (0,1) or (1,0) may have changed
Optimizing Approximate LRU Replacement

The Second Chance Algorithm

- There is a significant cost to replacing “dirty” pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the dirty bit and the used bit to drive replacement
The Second Chance Algorithm: Text Description

• There is a significant cost to replacing “dirty” pages
• Modify the clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  – Use both the dirty bit and the used bit to drive replacement
• When a page is referenced the reference bit is set
  – If the page was modified, aka written to, the dirty bit is set
• During first “sweep” of the hand the used bit is cleared
• During second “sweep” of the hand the dirty bit is cleared
  – Assumption: OS remembers that the pages is really dirty
• Only replace pages that have both dirty and used bit equal to zero
  – So if a page is dirty have to wait full 2 sweeps for it to be replaced
How much physical memory do we allocate to a process?

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>–</td>
</tr>
</tbody>
</table>

Faults

Faults

Faults
Local Page Replacement: Text Description

- Two examples, one slide
- For both:
  - Time and requests on x axis
  - Page frames on the y axis
  - Keep track of faults as you go
  - The request string is a, b, c, d, a, b, c, d, a, b, c, d.
  - Each request occurs at another time tick, so requests are at times 1-12
- Example 1:
  - 3 page frames
  - At time 0, pages a, b, and c are loaded into frames 0, 1, and 2 respectively.
- Example 2:
  - 4 page frames
  - At time 0, pages a, b, and c are loaded into frames 0, 1, and 2 respectively. Frame 3 is empty.
### How much physical memory do we allocate to a process?

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faults</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faults</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

- **Requests:** a b c d a b c d a b c d
- **Page Frames:**
  - Page Frame 0: a, a, a, a, d, d, d, c, c, c, b, b, b
  - Page Frame 1: b, b, b, b, b, b, a, a, a, d, d, d, c, c
  - Page Frame 2: c, c, c, c, c, c, b, b, b, a, a, a, d
- **Faults:** • • • • • • • • • • • • • •
Local Page Replacement: Solution

Text Description (1 of 2)

• How much memory do we allocate to a process?

• Example with 3 page frames and 4 pages with FIFO replacement:
  – reference order: a b c d a b c d a b c d
  – Time 0: a in frame 0, b in frame 1 and c in frame 2
  – Time 1: request page a, in memory so no page fault
  – Time 2: request page b, in memory so no page fault
  – Time 3: request page c, in memory so no page fault
  – Time 4: request page d which isn’t in memory, causes a page fault
    • put d where a used to be since a was the oldest page in memory
    • current frame setup: d in frame 0, b in frame 1 and c in frame 2
  – Time 5: request page a which isn’t in memory, causes a page fault
    • put a where b used to be since b was the oldest page in memory
    • current frame setup: d in frame 0, a in frame 1 and c in frame 2
  – Time 6: request page b which isn’t in memory, causes a page fault
    • put b where c used to be since c was the oldest page in memory
    • current frame setup: d in frame 0, a in frame 1 and b in frame 2
  – Time 7: request page c which isn’t in memory, causes a page fault
    • put c where d used to be since d was the oldest page in memory
    • current frame setup: c in frame 0, a in frame 1 and b in frame 2
  – Time 8: request page d which isn’t in memory, causes a page fault
    • put d where a used to be since a is the oldest page in memory
    • current frame setup: c in frame 0, d in frame 1 and b in frame 2
Local Page Replacement: Solution

Text Description (2 of 2)

– Time 9: request page a which isn’t in memory, causes a page fault
  • put a where b used to be since b was the oldest page in memory
  • current frame setup: c in frame, d in frame 1 and a in frame 2
– Time 10: request page b which isn’t in memory, causes a page fault
  • put b where c used to be since c was the oldest page
  • current frame setup: b in frame, d in frame 1 and a in frame 2
– Time 11: request page c which isn’t in memory, causes a page fault
  • put c where d used to be since d was the oldest page
  • current frame setup: b in frame, c in frame 1 and a in frame 2
– Time 12: request page d which isn’t in memory, causes a page fault
  • put d where a used to be since a was the oldest page
  • final frame setup: b in frame, c in frame 1 and d in frame 2
– Total number of page faults: 9
  • page faulted on almost every request

• Example with 4 page frames and 4 frames and FIFO replacement
  – reference order: a b c d a b c d a b c d
  – Time 0: a in frame 0, b in frame 1, c in frame 2 and frame 3 is empty
  – Time 1 - 3: request pages a b and c which are already in memory
  – Time 4: request page d which isn’t in memory, causes a page fault
    • put d in the empty frame
    • current frame setup: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  – now all pages are resident in memory so the rest of the requests don’t cause any page faults
  – Total number of page faults: 1

• These examples show that allocating too few frames to a process can cause a massive jump in the number of page faults
So... how much physical memory should we allocate to each process?

A. 4 frames  
B. 8 frames  
C. 10% of the frames  
D. 50% of the frames  
E. It depends.

Do we really want to decide this ahead of time? Do we really want a fixed number per process? Does that make sense?
Local vs. Global Page Replacement

• *Local* page replacement algorithms only consider the pages owned by the faulting process
  — Essentially a fixed number of pages per process

• *Global* page replacement algorithms consider all the pages
Is there another way?

The Principle of Locality

Recall: programs exhibit *temporal* and *spatial* locality

– 90% of execution is sequential

– Most iterative constructs consist of a relatively small number of instructions

– When processing large data structures, the dominant cost is sequential processing on individual structure elements
Explicitly Using Locality: The Working Set Model

• The working set is:
  – informally, the pages the process is using right now
  – formally, the set of all pages that a process referenced in the last T seconds

• Assume that recently referenced pages are likely to be referenced again soon

• Only keep those pages in memory (the working set!)
  – Pages may be removed even when no page fault occurs
  – Number of frames allocated to a process will vary over time

• Also allows pre-paging.
Working Set Page Replacement

Implementation

- Keep track of the last $T$ references
  - The pages referenced during the last $T$ memory accesses are the working set
  - $T$ is called the window size

- Example: Working set computation, $T = 4$ references:

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>c</td>
<td>e</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Pages in Process</td>
<td>Page a</td>
<td>• $t=1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page b</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page c</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page d</td>
<td>• $t=2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page e</td>
<td>• $t=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Faults
Working Set Page Replacement: Text Description

• Implementation of working set
• keep track of the last T references
  – the pages referenced during the last T memory accesses are the working set
  – T is called the window size
• Example: working set computation with T = 4 references
  – Time and requests on x axis
  – Pages (not frames) on the y axis
  – Are keeping track of whether or not each page is in memory
    • instead of keeping track of what each frame is holding
    • also keep track of the “age” of each page in memory
      – when a page becomes 5 ticks old it is kicked out of memory
  – Keep track of faults as you go
  – Time 0: pages a, d and e in memory
    • a is 1 tick old, d is 2 ticks old and e is 3 ticks old
  – The request string is c, c, d, b, c, e, c, e, a, d.
  – Each request occurs at another time tick, so requests are at times 1-10.
Working Set Page Replacement
Implementation

- Keep track of the last \( T \) references
  - The pages referenced during the last \( T \) memory accesses are the working set
  - \( T \) is called the window size

- Example: Working set computation, \( T = 4 \) references:
  - What if \( T \) is too small? too large?

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>c</td>
<td>e</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Pages in Process</td>
<td>Page a</td>
<td>( t = 1 )</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Page b</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Page c</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Page d</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Page e</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Faults</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
Working Set Page Replacement: Solution

Text Description (1 of 2)

• Example: working set computation with $T = 4$ references
  – are keeping track of whether or not each page is in memory
    • also keep track of the “age” of each page in memory
      – when a page becomes 5 ticks old it is kicked out of memory
  – Time 0: pages a, d and e in memory
    • a is 1 tick old, d is 2 ticks old and e is 3 ticks old
  – Time 1: page c is requested which isn’t in memory, causes a page fault
    • pages in memory: a, c, d and e
  – Time 2: page c is requested, is in memory so no page fault
  – Time 3: page d is requested, is in memory so no page fault
    • at this time page e becomes 5 ticks old so is kicked out of memory
    • pages in memory: a, c and d
  – Time 4: page b is requested which isn’t in memory, causes a page fault
    • at this time page a becomes 5 ticks old so is kicked out of memory
    • pages in memory: b c and d
Working Set Page Replacement: Solution
Text Description (2 of 2)

– Time 5: page c is requested, is in memory so no page fault
– Time 6: page e is requested which isn’t in memory, causes a page fault
  • pages in memory: b c d and e
– Time 7: page c is requested, is in memory so no page fault
  • at this time page d becomes 5 ticks old so is kicked out of memory
  • pages in memory: b c and e
– Time 8: page e is requested, is in memory os no page fault
  • at this time page b becomes 5 ticks old so is kicked out of memory
  • pages in memory: c and e
– Time 9: page a is requested which isn’t in memory, causes a page fault
  • pages in memory: a c and e
– Time 10: page d is requested which isn’t in memory, causes a page fault
  • pages in memory: a c d and e
– Total number of page faults: 5
How do we choose the value of $T$?

What if $T$ is too big?
What if $T$ is too small?
Thrashing

- *Thrashing* occurs when the memory is over-committed and pages are tossed out while they are still in use.
- Many memory references cause pages to be faulted in.
  - Very serious and very noticeable loss of performance.

How do we limit thrashing in a multiprogrammed system?
How do we choose the value of T?

What if T is too big?
What if T is too small?

1 page fault = 10ms
10ms = 2M instructions
T needs to be a lot bigger than 2 million instructions
Load Control

- *Load control* refers to the number of processes that can reside in memory at one time.
- Working set model provides implicit load control by only allowing a process to execute if its working set fits in memory.
- BUT process frame allocations are variable.
- What happens when the total number of pages needed is greater than the number of frames available?  
  - Processes are swapped out to disk.
Load Control

When the multiprogramming level should be decreased, which process should be swapped out?
- Lowest priority process?
- Smallest process?
- Largest process?
- Oldest process?
- Faulting process?
Load Control: Text Description

• When a process is totally swapped out of memory it is put onto swap (aka the paging disk)

• Adds another stage to the process life cycle
  – This new stage is called suspended
    • We also saw this stage in relocation, when we also swapped out entire processes
  – Can go from any of the other states to suspended
    • Usually blocked or ready
  – Process can go from suspended to ready
Another Decision: Page Sizes

Page sizes are growing slowly but steadily. Why?

• Benefits for small pages: more effective memory use, higher degree of multiprogramming possible

• Benefits for large pages: smaller page tables, reduced I/O time, fewer page faults

•Growing because:
  – memory is cheap---page tables could get huge with small pages and internal fragmentation is less of a concern
  – CPU speed is increasing faster than disk speed, so page faults cause a larger slow down
iClicker Question

Can an application modify its own translation tables (however they are implemented)?

A. Yes
B. No
Summary:
Page Replacement Algorithms

• Unix and Linux use a variant of the second chance algorithm
• Windows NT uses FIFO replacement
• All algorithms do poorly on typical processes if processes have insufficient physical memory
• All algorithms approach optimal as the physical memory allocated to a process approaches virtual memory size
• The more processes running concurrently, the less physical memory each process can have
Summary: Paging

We’ve considered:

• Placement Strategies
  – None needed, can place pages anywhere

• Replacement Strategies
  – What to do when more jobs exist than can fit in memory

• Load Control Strategies
  – Determine how many jobs can be in memory at one time
Summary: Paging

The Good

The expense of memory accesses and the flexibility of paging make paging cost effective.
Announcements

• Discussion sections on Friday! Problem Set 6 is posted.

• Stack Check appointments today and tomorrow
  – You should have received email at your CS account telling you who to meet
    • If you didn’t, check that you didn’t give me your EID
  – If you haven’t signed up, see me

• Project 2 due Friday, 3/29

• You MUST get it working (except multi-oom)
Dynamic Relocation:
Address Translation

Program P’s logical address space

CPU

Logical Addresses

Base Register

≤

1000

≤

500

≤

500

Physical Addresses

MEMORY EXCEPTION

no

MAXprog

Instructions

Program P’s physical address space

MAXsys

1500

1000

0

0

no
Dynamic Relocation:
Text Description

• How we get from a program’s logical address space to physical memory.
• The bound register is used to make sure that the program isn’t trying to access memory outside of its space. For example, if the programs logical address space ranges from 0 to 500 then the bound register would hold a value of 500.
• The base register holds the beginning of that program’s addresses in physical memory. So if the programs memory started at address 1000 then the base register would hold 1000.
• Steps to Address Translation:
  – The program gives a logical address, or instruction, to the CPU.
  – The MMU now does 2 things at once:
    • It checks the address against the bound register, if the address is greater than the bound register a memory exception is thrown. This exception is a hardware interrupt that will be handled by the OS. This exception indicates the program was trying to access something that doesn’t belong to it.
    • It adds the base register to the logical address to get the physical address. For example logical address 8 would become 1008 if the base register was 1000.
• NOTE: Even when the address is successfully translated, you are not protected from memory errors: you could still accidentally try to access memory in your address space that hasn’t be initialized yet.
Virtual Address Translation

• A page table maps virtual pages to physical frames
Virtual Address Translation: Text Description

Steps to Virtual-Physical Memory Translation:
• The program gives a virtual address to the CPU to translate
• The MMU splits the virtual address into its page and offset numbers
• Since the offset is the same in Virtual and Physical memory it is sent along with no change
• The page number is translated into a frame number
  – Look into page table to find out if page exists in physical memory
    • The page number is the index of the correct entry in the page table (much like indexing an array)
  – If the page exists in memory the frame number is sent along
  – If it doesn’t exist in memory but exists on disk, the corresponding page is moved from disk into physical memory and its frame number is recorded
• Append offset to end of frame number to get the full physical address
Memory Management: Putting it all Together

• Dynamic Relocation with Base and Bounds:
  – Simple, but inflexible
  – Degree of multiprogramming limited, memory limited to physical memory size, no sharing of memory, memory allocation/deallocation difficult
  – Use compaction to solve external fragmentation

• Paging:
  – Process generates virtual addresses from 0 to Max
  – OS divides processes into pages
    • manages a page table for every process
    • manages the pages in memory
  – Simplifies memory allocation since any page can be allocated to any frame
  – Page tables can be very large
  – Page Replacement Algorithms
    • FIFO, Optimal, LRU, Clock, Enhanced Clock, Working Set
  – Design Considerations (page size, global vs. local, ...)
Today’s Agenda

• Page Faults
• Demand paging vs. pre-paging
• Paging: Policy
  – Page Replacement Algorithms
    • FIFO
    • LRU
    • Clock
  – Effect of increased memory
  – Global vs. local
  – Page Sizes
• Multiprogramming
  – Swap
• Segmentation
  – A new method of managing memory
  – How do we think about a process’s memory?
  – Let’s teach the OS about that!
• Segmentation vs. Paging
• Segmentation AND Paging
Segmentation

Segments take the user’s view of the program and gives it to the OS

– User views the programs in logical *segments*

  • code, global variables, stack, heap
  • NOT a linear array of bytes

– Convert logical view into physical partition

  • No longer a single address space
Segmentation
Advantages of Segmentation

• What are they?

• We already have protection, right?
  – Currently, protection is based on process address space. Can we do better?

• We already have sharing, right?
  – What if the shared code/data crosses page boundaries? or doesn’t begin at the beginning of a page at all?

• What about the stack and the heap?
Advantages of Segmentation

• Segments provide more fine-grained protection
  – Can provide policy per segment rather than for whole address space
  – Program text can be read only
  – Program data can be read/write

• Segments make sharing easier
  – Segment is shareable rather than they page

• Segments make it easier for the call stack and heap to grow dynamically
  – No need to guess how much room to allow
Implementing Segmentation

• Use virtual addresses that are segments and segment offsets
  – Compiler generates these addresses

• Memory mapping similar to paging
  – Segment table
    • Each entry contains a base address in memory, length of segment, and protection information
    – Hardware support: multiple base/limit registers

• If many segments exist, we’ll need something like the TLB
Implementing Segmentation

Add a segment table containing base & limit register values

Logical Addresses

STBR

Segment Table

Program $P$

CPU

$s_o^n$

0

32

0

Limit Register

$500$

$1000$

Base Register

$≤$

yes

no

MEMORY EXCEPTION

Physical Memory

0

1000

1500

Program $P$'s Segment

Program $P$

$n$

$s$

$s$

$base$

$limit$

$n$

$s$

Add a segment table containing base & limit register values.
Disadvantages of Segmentation

Contiguous piece in memory for each segment
  – Leads to...
    • External fragmentation
    • Bad utilization (what if you use a small part of a large piece?)
    • Allocation/deallocation of arbitrarily sized segments is complex
  – Similar problems to relocation algorithms (best fit, worst fit, first fit)
  – So... what should we do?
Segmentation vs. Paging

• Segmentation has what advantages over paging?
  – A. Fine-grained protection.
  – B. Easier to manage transfer of segments to/from the disk.
  – C. No external fragmentation

• Paging has what advantages over segmentation?
  – A. Fine-grained protection.
  – B. Easier to manage transfer of pages to/from the disk.
  – C. No external fragmentation.
Segmentation and Paging

- Treat virtual address space as a collection of segments of arbitrary sizes.
- Treat physical memory as a sequence of fixed size page frames.
- Segments are typically larger than page frames, so we map a logical segment onto multiple page frames by paging the segments.
Segmentation and Paging
Addresses in a Segmented Paging System

• A virtual address becomes a segment number, a page within that segment, and an offset within a page
  – segment number must be generated separately by the compiler
  – compiler can treat the page number and offset simply as an offset within the segment
• Segment number indexes into the segment table which yields base address for that segment
• Check the remainder of the address against the limit of the segment
• Use the page number to index the page table. The entry is the frame (same as paging)
• Frame and offset give physical address
Implementing Segmented Paging

- Add a segment table containing base & limit register values

Logical Addresses

Program $P$

CPU

Logical Addresses

STBR

Segment Table

Limit Register

Page Table

Physical Memory

Program $P$'s

Paged Segment

MEMORY EXCEPTION

Physical

Memory

≤

yes

f

no

0

32

0

n

p

o

s

base

limit

500

p

f
Given:

- a memory size of 256 addressable words
- a page table indexing 8 pages
- a page size of 32 words
- 8 logical segments

• How many bits is a physical address?
  A. 6  B. 8  C. 11  D. 13

• How many bits is a virtual address?
  A. 6  B. 8  C. 11  D. 13

• How many segment table entries do we need?
  A. 6  B. 8  C. 11  D. 13
Segmented Paging: Sharing

• Share individual pages by copying page tables entries
• Share whole segments by sharing segment table entries
  – Same as sharing the page table for that segment
• Need protection bits to specify and enforce read/write permission
Segmented Paging: More Implementation

• Where are the segment table and page tables stored?
  – Can store segment tables in small number of associative registers. Compilers then generate code for that number of segments. Page tables are in main memory with a TLB in hardware.
  – OR Both the segment and page tables can be in main memory with the segment index and page index combined used in the TLB lookup. In this case, hardware does not limit number of segments.

• What if they get too big?
Segmented Paging: More Implementation

• Valid and protection bits can go either on the segment or the page table entries.
• In a pure paging system, a forward page table contains \((\text{virtual address space})/\text{(page size)}\) entries. How many entries are required in segmented paging?
Questions

• What is the performance of address translation of segmented paging compared to relocation with base and bounds?

• Compared to pure paging?
Questions

• How does the external fragmentation of segmented paging compare with dynamic relocation?

• With pure paging?
Memory Management: Putting it all Together

• Segmentation
  – Compiler’s view presented to the OS
  – Segment tables tend to be small
  – Provides fine-grained protection, easier sharing, easier growth of stack and heap
  – Memory allocation/deallocation is expensive and complicated (back to first fit, worst fit, best fit)

• Segmented Paging
  – Only allocate the page table entries the process is using
  – Easy memory allocation, as any frame can be used
  – Sharing at either page or segment level
  – Increased internal fragmentation over paging
  – Two lookups per memory reference (may change depending on TLB usage)
Paging: The Mechanism

- Physical and virtual memory are partitioned into equal size units (pages and page frames)
- Size of VAS unrelated to size of physical memory
- Virtual pages are mapped to physical frames
- There is no external fragmentation
- Missing pages cause page faults
Inverted Page Tables: Example

- Physical Memory Size: 16MB
- Page Size: 4096 bytes
- Number of frames: 4096
- Space for page entries: 32 KB (8 bytes/entry)
- Percentage overhead: 0.2%
- Size of virtual memory: irrelevant
Initializing Memory: Pre-Paging with a TLB

1. A process needing $k$ pages arrives
2. The OS allocates all $k$ pages to the free frames.
   - The OS puts each page in a frame
   - The OS updates the entries in the page table
3. OS marks all TLB entries as invalid (flushes the TLB).
4. OS starts a process
5. As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.
Initializing Memory: Demand Paging with a TLB

1. A process arrives
2. The OS stores the process’s VAS on disk and does not put any of it into memory
3. OS marks all TLB entries as invalid (flushes the TLB).
4. OS starts the process
5. As process executes:
   1. Pages are faulted in as they are needed
      • The OS puts each page in a frame
      • The OS updates the entries in the page table
   2. OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.
Performance of Demand Paging

• Theoretically, a process could access a new page of memory with each instruction (expensive!)
• But processes tend to exhibit *locality of reference*
  – *temporal locality*: if a process accesses an item in memory, it will tend to reference the same item again soon
  – *spatial locality*: if a process accesses an item in memory, it will tend to reference an adjacent item soon
Page Replacement Algorithms: FIFO

First-In, First-Out

– Throw out the oldest page
– Simple to implement
– OS can easily throw out a page that is being accessed frequently
FIFO Replacement

- Implemented with a single pointer
- Performance with 4 page frames:

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

Frame List

3
0
2

0 1 2 3 4 5 6 7 8 9 10
FIFO Replacement: Text Description

• Implemented with a single pointer
  – pointer points to the oldest page

• Performance with 4 page frames:
  – keep track of time and requests made at each time
    • requests denoted with letters (a, b, c etc)
  – time shown on the x-axis
  – page frames kept track on the y-axis
  – keep count of faults as you go

• At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.

• The request string is c, a, d, b, e, b, a, b, c, d.

• Each request occurs at another time tick, so requests are at times 1-10.
FIFO Replacement: Solution

- Implemented with a single pointer
- Performance with 4 page frames:

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Page Frames</td>
<td>0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>Faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

Frame List

Faults
FIFO Replacement: Solution
Text Description

Performance with 4 page frames and 10 requests:

- single letters denote virtual pages
- Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - oldest page is a
- Time 1: request page c, already in memory so no fault
- Time 2: request page a, already in memory so no fault
- Time 3: request page d, already in memory so no fault
- Time 4: request page b, already in memory so no fault
- Time 5: request page e which isn’t in memory, causes a page fault
  - puts e where a used to be since a was the oldest page in memory
  - new frame setup: e in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - new oldest page is b
- Time 6: request page b, already in memory so no fault
- Time 7: request page a which isn’t in memory, causes a page fault
  - puts a where b used to be since b was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, c in frame 2 and d in frame 3
  - new oldest page is c
- Time 8: request page b which isn’t in memory, causes a page fault
  - puts b where c used to be since b was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, b in frame 2 and d in frame 3
  - new oldest page is d
- Time 9: request page c which isn’t in memory, causes a page fault
  - puts c where d used to be since d was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, b in frame 2 and c in frame 3
  - new oldest page is e
- Time 10: request page d which isn’t in memory, causes a page fault
  - puts d where e used to be since e was the oldest page in memory
  - final frame setup: d in frame 0, a in frame 1, b in frame 2 and c in frame 3
- Total number of page faults: 5
What happens when we have a page fault.... but memory is full?

The OS:

1. selects a page to replace (page replacement algorithm)
2. invalidates the old page in the page table
3. starts loading the new page into memory from disk
4. context switches to another process while I/O is being done
5. gets interrupt that the page is loaded in memory
6. updates the page table entry
7. continues faulting process