Page Replacement
Page Replacement

Assume a normal page table (e.g., BLITZ)
User-program is executing
A *PageInvalidFault* occurs!
- The page needed is not in memory
Select some frame and remove the page in it
  - If it has been modified, it must be written back to disk
    the “dirty” bit in its page table entry tells us if this is necessary
Figure out which page was needed from the faulting addr
Read the needed page into this frame
Restart the interrupted process by retrying the same instruction
Page Replacement Algorithms

Which frame to replace?

*Algorithms:*
  - The Optimal Algorithm
  - First In First Out (FIFO)
  - Not Recently Used (NRU)
  - Second Chance / Clock
  - Least Recently Used (LRU)
  - Not Frequently Used (NFU)
  - Working Set (WS)
  - WSClock
The Optimal Algorithm

Idea:
Select the page that will not be needed for the longest time
Optimal Page Replacement

Replace the page that will not be needed for the longest time.

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</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>Frames</td>
<td>a</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>
</tbody>
</table>

Page faults

Page faults

X
## Optimal Page Replacement

Select the page that will not be needed for the longest time

**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>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>Page</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>Frames</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>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>
</tr>
<tr>
<td>3</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>e</td>
</tr>
</tbody>
</table>

**Page faults**

<table>
<thead>
<tr>
<th>Page faults</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
</table>
Optimal Page Replacement

Idea:
Select the page that will not be needed for the longest time

Problem?
Optimal Page Replacement

Idea:
Select the page that will not be needed for the longest time

Problem:
Can’t know the future of a program
Can’t know when a given page will be needed next
The optimal algorithm is unrealizable
Optimal Page Replacement

However:

We can use it as a control case for simulation studies

- Run the program once
- Generate a log of all memory references
- Do we need all of them?
- Use the log to simulate various page replacement algorithms
- Can compare others to “optimal” algorithm
FIFO Algorithm

Always replace the oldest page ...

- Replace the page that has been in memory for the longest time
FIFO Algorithm

Replace the page that was first brought into memory

Example: Memory system with 4 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>a</td>
<td></td>
</tr>
</tbody>
</table>

| Page | 0 | a |   | a |   | a |   |   |   |   |
| Frames | 1 | b |   |   | b |   |   |   |   |   |
| 2 | c |   | c |   |   | c |   |   |   |   |
| 3 | d |   |   | c |   |   | d |   |   |   |

Page faults | x
FIFO Algorithm

Replace the page that was first brought into memory

Example: Memory system with 4 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>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

| Page | 0 | a | a | a | a | a | a | a | a | a | a |
| Frames 1 | b | b | b | b | b | b | b | b | b | b | b |
| 2 | c | c | c | c | c | c | c | c | c | c | c |
| 3 | d | d | d | d | d | d | d | d | d | d | d |

Page faults: X X
**FIFO Algorithm**

Replace the page that was first brought into memory

Example: Memory system with 4 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>a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>0</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames</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></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>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d</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>
<td></td>
</tr>
</tbody>
</table>

Page faults

```
x   x   x
```
**FIFO Algorithm**

Replace the page that was first brought into memory

**Example: Memory system with 4 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>a</td>
<td></td>
</tr>
<tr>
<td>Page</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>c</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Frames 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></td>
</tr>
<tr>
<td>2</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>
<td>e</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d</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>
<td></td>
</tr>
<tr>
<td>Page faults</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIFO Algorithm

Always replace the oldest page.
  - *Replace the page that has been in memory for the longest time*

Implementation
  Maintain a linked list of all pages in memory
  Keep it in order of when they came into memory
  The page at the tail of the list is oldest
  Add new page to head of list
FIFO Algorithm

Disadvantage?
FIFO Algorithm

Disadvantage:
- The oldest page may be needed again soon
- Some page may be important throughout execution
- It will get old, but replacing it will cause an immediate page fault
How Can We Do Better?

Need an approximation of how likely each frame is to be accessed in the future

- If we base this on past behavior we need a way to track past behavior
- Tracking memory accesses requires hardware support to be efficient
Referenced and Dirty Bits

Each page table entry (and TLB entry!) has a
- Referenced bit - set by TLB when page read / written
- Dirty / modified bit - set when page is written
- If TLB entry for this page is valid, it has the most up to date version of these bits for the page
- OS must copy them into the page table entry during fault handling

Idea: use the information contained in these bits to drive the page replacement algorithm
Referenced and Dirty Bits

Some hardware does not have support for the dirty bit. Instead, memory protection can be used to emulate it.

Idea:

- Software sets the protection bits for all pages to “read only”
- When program tries to update the page...
  - A trap occurs
  - Software sets the Dirty Bit in the page table and clears the ReadOnly bit
  - Resumes execution of the program
Not Recently Used Algorithm

Uses the Referenced Bit and the Dirty Bit

Initially, all pages have
  - Referenced Bit = 0
  - Dirty Bit = 0

Periodically... (e.g. whenever a timer interrupt occurs)
  - Clear the Referenced Bit
  - Referenced bit now indicates “recent” access
Not Recently Used Algorithm

When a page fault occurs...

Categorize each page...

- **Class 1:** Referenced = 0   Dirty = 0
- **Class 2:** Referenced = 0   Dirty = 1
- **Class 3:** Referenced = 1   Dirty = 0
- **Class 4:** Referenced = 1   Dirty = 1

Choose a victim page from class 1 ... why?
If none, choose a page from class 2 ... why?
If none, choose a page from class 3 ... why?
If none, choose a page from class 4 ... why?
Second Chance Algorithm

An implementation of NRU based on FIFO
Pages kept in a linked list (oldest at the front)
Look at the oldest page
   If its “referenced bit” is 0...
       - Select it for replacement

   Else
       - It was used recently; don’t want to replace it
         - Clear its “referenced bit”
         - Move it to the end of the list

Repeat

What if every page was used in last clock tick?
Implementation of Second Chance

Maintain a circular list of pages in memory
Set a bit for the page when a page is referenced
Search list looking for a victim page that does not have the referenced bit set
- If the bit is set, clear it and move on to the next page
- Replaces pages that haven’t been referenced for one complete clock revolution
Least Recently Used Algorithm

A refinement of NRU that orders how recently a page was used
- Keep track of when a page is used
- Replace the page that has been used least recently
# Least Recently Used Algorithm

Replace the page that hasn’t been referenced in 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>0</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames 1 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></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>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</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>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Page faults</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
</tbody>
</table>
Least Recently Used Algorithm

But how can we implement LRU?
Least Recently Used Algorithm

But how can we implement LRU?

Idea #1:
- Keep a linked list of all pages
- On every memory reference, Move that page to the front of the list
- The page at the tail of the list is replaced
Least Recently Used Algorithm

But how can we implement LRU?
... without requiring every access to be recorded?

Idea #2:
- MMU (hardware) maintains a counter
- Incremented on every clock cycle
- Every time a page table entry is used
  - MMU writes the value to the page table entry
  - This timestamp value is the time-of-last-use
- When a page fault occurs
  - OS looks through the page table
  - Identifies the entry with the oldest timestamp
Least Recently Used Algorithm

What if we don’t have hardware support for a counter?

Idea #3:
- Maintain a counter in software
- One every timer interrupt...
  - Increment counter
  - Run through the page table
  - For every entry that has “ReferencedBit” = 1
    * Update its timestamp
    * Clear the ReferencedBit
- Approximates LRU
- If several have oldest time, choose one arbitrarily
Not Frequently Used Algorithm

Bases decision of frequency of use rather than recency
Associate a counter with each page
On every clock interrupt, the OS looks at each page.
  - If the reference bit is set increment that page’s counter &
    clear the bit
The counter approximates how often the page is used
For replacement, choose the page with lowest counter
Not Frequently Used Algorithm

Problem:

Some page may be heavily used
  - Its counter is large
The program’s behavior changes
  - Now, this page is not used ever again (or only rarely)
This algorithm never forgets!
  - This page will never be chosen for replacement!

We may want to combine frequency and recency somehow
NFU With Aging

Associate a counter with each page
On every clock tick, the OS looks at each page.
  - Shift the counter right 1 bit (divide its value by 2)
  - If the reference bit is set...
    - Set the most-significant bit
    - Clear the Referenced Bit

\[
\begin{align*}
T_1 & \quad 100000 = 32 \\
T_2 & \quad 010000 = 16 \\
T_3 & \quad 001000 = 8 \\
T_4 & \quad 000100 = 4 \\
T_5 & \quad 100010 = 34
\end{align*}
\]
The Working Set

Demand paging
- Pages are only loaded when accessed
- When process begins, all pages marked INVALID
The Working Set

Demand paging
  - Pages are only loaded when accessed
  - When process begins, all pages marked INVALID

Locality of reference
  - Processes tend to use only a small fraction of their pages
The Working Set

Demand paging
- Pages are only loaded when accessed
- When process begins, all pages marked INVALID

Locality of reference
- Processes tend to use only a small fraction of their pages

Working Set
- The set of pages a process needs
- If working set is in memory, no page faults
- What if you can’t get working set into memory?
The Working Set

Thrashing

- If you can’t get working set into memory page faults occur every few instructions
- Little work gets done
- Most of the CPU’s time is going on overhead
Working Set Algorithm

Based on prepaging (prefetching)
- Load pages before they are needed

Main idea:
- Try to identify the process’s working set based on time
- Keep track of each page’s time since last access
- Assume working set valid for T time units
- Replace pages older than T
Working Set Algorithm

*Current Virtual Time*
- Only consider how much CPU time this process has seen

*Implementation*
- On each clock tick, look at each page
- Was it referenced since the last check?
  - Yes: make a note of Current Virtual Time
- If a page has not been used in the last T msec,
  - Assume it is not in the working set!
  - Evict it
  - Write it out if it is dirty
Working Set Algorithm

Scan all pages examining R bit:

- if \( R = 1 \)
  - set time of last use to current virtual time
- if \( R = 0 \) and age > \( \tau \)
  - remove this page
- if \( R = 0 \) and age \( \leq \tau \)
  - remember the smallest time
Page Fault Frequency

If $T$ is too small, page fault frequency will be high
If you make it bigger page fault frequency will decline
Page Fault Frequency

Too High: Need to give this process some more frames!

Too Low: Take some frames away and give to other processes!
Page Fault Frequency

Measure the page fault frequency of each process
Count the number of faults every second

May want to consider the past few seconds as well
Page Fault Frequency

Measure the page fault frequency of each process
Count the number of faults every second

May want to consider the past few seconds as well

Aging:
  Keep a running value
  Every second
    - Count number of page faults
    - Divide running value by 2
    - Add in the count for this second
Which Algorithm is Best?
Modeling Algorithm Performance

Run a program
- Look at all memory references
- Don’t need all this data
- Look at which pages are accessed
  0000001222333300114444001123444
- Eliminate duplicates
  012301401234

This defines the *Reference String*
- Use this to evaluate different algorithms
- Count page faults given the same reference string
Proactive Replacement

Replacing victim frame on each page fault typically requires two disk accesses per page fault.

Alternative → the O.S. can keep several pages free in anticipation of upcoming page faults.

In Unix: low and high water marks

\[
\text{low water mark} \quad \text{high water mark}
\]

\[
\text{low} < \# \text{ free pages } < \text{high}
\]
UNIX Page Replacement

Clock algorithm for page replacement
- If page has not been accessed move it to the free list for use as allocatable page
- If modified/dirty \(\rightarrow\) write to disk (still keep stuff in memory though)
- If unmodified \(\rightarrow\) just move to free list

High and low water marks for free pages
- Pages on the free-list can be re-allocated if they are accessed again before being overwritten
Local vs. Global Replacement

Assume several processes: A, B, C, ...
Some process gets a page fault (say, process A)
Choose a page to replace.

*Local page replacement*
- Only choose one of A’s pages

*Global page replacement*
- Choose any page
Local vs. Global Replacement

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th></th>
<th>Local</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>10</td>
<td>A0</td>
<td></td>
<td>A6</td>
</tr>
<tr>
<td>A1</td>
<td>7</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>5</td>
<td>A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>4</td>
<td>A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>6</td>
<td>A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td>9</td>
<td>B0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>4</td>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>6</td>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>5</td>
<td>B4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>6</td>
<td>B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>12</td>
<td>B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
<td>C1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
<td>C2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>6</td>
<td>C3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Global
Local vs. Global Replacement

Assume we have
- 5,000 frames in memory and 10 processes

Idea: Give each process 500 frames

Is this fair?
- Small processes do not need all those pages
- Large processes may benefit from even more frames

Idea:
- Look at the size of each process (... but how?)
- Give them a pro-rated number of frames with some minimum
Load Control

Assume:
- The best page replacement algorithm
- Optimal global allocation of page frames
Load Control

Assume:
- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!
Load Control

Assume:
- The best page replacement algorithm
- Optimal global allocation of page frames

*Thrashing is still possible!*
- Too many page faults!
- No useful work is getting done!
- Demand for frames is too great!
Load Control

Assume:
- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!
- Too many page faults!
- No useful work is getting done!
- Demand for frames is too great!

Solution:
- Get rid of some processes (temporarily swap them out)
- Two-level scheduling (swapping with paging)