CMPT 300
Introduction to Operating Systems

Page Replacement Algorithms
Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk
**Illusion of Infinite Memory**

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency

- Principle: **Transparent Level of Indirection** (page table)
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue
Since Demand Paging is Caching, must ask:

- What is block size?
  - 1 page

- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual $\rightarrow$ physical mapping

- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal

- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation… (kinda LRU)

- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)

- What happens on a write? (write-through, write back)
  - Write-back. Need dirty bit!
Demand Paging Example

Since Demand Paging like caching, can compute average access time! ("Effective Access Time")

- EAT = Hit Rate x Hit Time + Miss Rate x Miss Time

Example:
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- Suppose p = Probability of miss, 1-p = Probably of hit
- Then, we can compute EAT as follows:
  
  \[
  EAT = (1 - p) \times 200\text{ns} + p \times 8,000,000\text{ns}
  \]
  
  \[
  = (1 - p) \times 200\text{ns} + p \times 7,999,800\text{ns}
  \]

If one access out of 1,000 causes a page fault, then

- EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!

What if want slowdown by less than 10%?
- 200ns x 1.1 < EAT \( \Rightarrow \) p < 2.5 \times 10^{-6}
- This is about 1 page fault in 400000!
What Factors Lead to Misses?

**Compulsory Misses:**
- Pages that have never been paged into memory before
- How might we remove these misses?
  - Prefetching: loading them into memory before needed
  - Need to predict future somehow! More later.

**Capacity Misses:**
- Not enough memory. Must somehow increase size.
- Can we do this?
  - One option: Increase amount of DRAM (not quick fix!)
  - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!

**Conflict Misses:**
- Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache

**Policy Misses:**
- Caused when pages were in memory, but kicked out prematurely because of the replacement policy
- How to fix? Better replacement policy
Replacement policy

Why do we care about Replacement Policy?
- Replacement is an issue with any cache
- Particularly important with pages
  - The cost of being wrong is high: must go to disk
  - Must keep important pages in memory, not toss them out

The simplest algorithm:
- Pick random page for every replacement
- Typical solution for TLB. Simple hardware
- Unpredictable – makes it hard to make real-time guarantees
Recall: What is in a Page Table Entry (PTE)?

- **Page frame number.** Physical memory address of this page
- **Present/absent bit,** also called **valid bit.** If this bit is 1, the page is in memory and can be used. If it is 0, the page is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault to get page from disk.
- **Protection bits** tell what kinds of access are permitted on the page. 3 bits, one bit each for enabling read, write, and execute.
- **Modified (M) bit,** also called **dirty bit,** is set to 1 when a page is written to
- **Referenced (R) bit,** is set whenever a page is referenced, either for reading or writing.
- M and R bits are very useful to page replacement algorithms
- **Caching disabled bit,** important for pages that map onto device registers rather than memory

![Diagram of Page Table Entry](image)
**R & M bits**

- **Referenced (R) bit** indicates if the page has been used recently.
  - Each time the page is referenced (read or written to), the R bit is set to 1.
  - OS defines a *clock period* for paging management. Every clock period, the R bit for each page is reset to 0.
    - R=0 → page is old (not used for some time)
    - R=1 → page is new (recently used)

- **Modified (M) bit** indicates if the page has been modified (written to) since it was last synced with disk.
  - The flag is reset when the page is saved to disk
  - When a page is removed from physical memory
    - M=1 → it will be saved to disk
    - M=0 → it will be abandoned and not saved to disk
Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Present $\Rightarrow$ Page in memory, PTE points at physical page
  - Absent $\Rightarrow$ Page not in memory; use info in PTE to find it on disk when necessary

- Suppose user references page with Absent PTE?
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a “Page Fault”
  - What does OS do on a Page Fault?:
    - Choose an old page to replace
    - If old page modified, write page contents back to disk
    - Change its PTE and any cached TLB to be invalid
    - Load new page into memory from disk
    - Update PTE, invalidate TLB for new entry
    - Continue thread from original faulting location
  - TLB for new page will be loaded when thread continues!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - Suspended process sits on wait queue
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Operating system
4. Bring in missing page
5. Reset page table
6. Restart instruction

Load M
Page Replacement Algorithms

- Optimal (OPT)
- Not recently used (NRU)
- First-In, First-Out (FIFO)
- Second chance (SC)
- Least recently used (LRU)
- Not frequently used (NFU)
- Aging algorithm
- Clock algorithm
- Working set
- WSClock
OPT page replacement

- Replace page that won’t be used for the longest time
- Optimal, but infeasible in practice, since can’t really know future…
- Makes good comparison case, however
Not recently used (NRU)

Use the referenced and modified bits in the page table entries. 4 possibilities:
1. Not referenced, not modified
2. Not referenced, modified (reference cleared on clock interrupt)
3. Referenced, not modified
4. Referenced, modified

When a page fault occurs, find any page in group 1, failing that any page in group 2, failing that any page in group 3, failing that any page in group 4. If there is more than one page in the lowest group, randomly choose one page from the group.

Rationale: replace the page that has not been referenced or modified.
FIFO

- Throw out oldest page.
  - Be fair – let every page live in memory for same amount of time.
  - Bad, because it tends to throw out heavily used pages instead of infrequently used pages
  - Second-chance algorithm avoids this problem by giving recently-used pages a second chance
Second-Chance Algorithm

- Give recently-used pages a second chance
  - If the oldest page has $R=0$, then choose it for replacement; if $R=1$, then move it to the end, and update its load time as through it's a new arrival.

(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.
Least Recently Used (LRU)

- Replace page that hasn’t been used for the longest time
- Programs have locality, so if something not used for a while, unlikely to be used in the near future.
- Seems like LRU should be a good approximation to OPT.
- How to implement LRU? Use a list!

On each use, remove page from list and place at head
- LRU page is at tail

Problems with this scheme for paging?
- List must be updated at every memory reference; List manipulation is expensive
- In practice, people approximate LRU (more later)
Consider a cache size of 3 page frames, and following reference stream of virtual pages:

- A B C A B D A D B C B

Consider FIFO Page replacement:

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</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>1</td>
<td>A</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: OPT

โร Suppose we have the same reference stream:
- A B C A B D A D B C B

Consider OPT Page replacement:
- 5 faults
- Where will D be brought in? Look for page not referenced farthest in future (C).

What will LRU do?
- Same decisions as OPT here, but won’t always be true!

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</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>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></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>3</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When will LRU perform badly?

Consider the following: A B C D A B C D A B C D

LRU Performs as follows (same as FIFO here):
Every reference is a page fault!

<table>
<thead>
<tr>
<th>Ref:</th>
<th>Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
**OPT Does much better**

- **But it’s not implementable**

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</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>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ref:
Page:
Exercise

Consider a cache size of 3 page frames, and following reference stream of virtual pages:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Run FIFO, OPT and LRU on this example.

Answer:

FIFO: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/fifopagereplacement.htm

OPT: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/optimalpagereplacement.htm

LRU: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/lrupagereplacement.htm
One desirable property: When you add memory the miss rate goes down

Does this always happen?

Seems like it should, right?

No: BeLady’s anomaly

Certain replacement algorithms (FIFO) don’t have this obvious property!
Does adding memory reduce number of page faults?
- Yes for LRU and OPT
- Not necessarily for FIFO! (Called Belady’s anomaly)

After adding memory:
- With FIFO, contents can be completely different
- In contrast, with LRU or OPT, contents of memory with X pages are a subset of contents with X+1 Page

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</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>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9 page faults

<table>
<thead>
<tr>
<th>Ref:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</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>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>E</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 page faults
Implementing LRU

Perfect:
- Timestamp page on each reference
- Keep list of pages ordered by time of reference
- Too expensive to implement in reality

Techniques for approximating LRU. Goal is to Replace an old page, not the oldest page

Hardware techniques
- 64-bit counter
- n x n matrix

Software techniques
- Not recently used (NRU)
- Aging Algorithm
- Clock Algorithm
LRU in hardware

Implementation #1:

- 64 bit counter, C, incremented after every instruction
- Each page also has a 64 bit counter
- When a page is referenced, C is copied to its counter.
- Page with lowest counter is oldest.
LRU in hardware

Implementation #2:
- Given n page frames, let M be a n x n matrix of bits initially all 0.
- Reference to page frame k occurs.
- Set all bits in row k of M to 1.
- Set all bits in column k of M to 0.
- Row with lowest binary value is least recently used.
LRU in hardware: implementation #2 example

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) 0, 0, 1, 1, 1, 0, 1, 0, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) 0, 0, 0, 0, 1, 1, 0, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) 0, 0, 0, 0, 0, 0, 1, 1, 0

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(e) 0, 1, 0, 0, 0, 0, 0, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(f) 0, 0, 0, 0, 0, 1, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(g) 1, 0, 0, 0, 0, 0, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(h) 1, 1, 0, 0, 0, 0, 0, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(i) 1, 1, 0, 0, 0, 0, 1, 1

<table>
<thead>
<tr>
<th>Page</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(j) 1, 1, 1, 0, 0, 1, 1

Figure 3-17. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.
A software counter associated with each page, initially zero. At end of each clock period, the operating system scans all the pages in memory.

For each page, the $R$ bit (0 or 1), is added to the counter (arithmetic addition), which roughly keeps track of how often each page has been referenced. When a page fault occurs, the page with the smallest counter is chosen for replacement.

Problem: It never forgets!

So pages that were frequently referenced (during initialization for example) but are no longer needed appear to be FU.
Aging algorithm

- Idea: Gradually forget the past
  - A k-bit software counter is associated with each page, the counter is initialized to 0
  - Shift all counters to right 1 bit before R bit is added in.
  - Then R bit is added to MSb (Most Significant (leftmost) bit)
  - Page with lowest counter value is chosen for removal.
# Aging algorithm example

Shown are six pages for five clock periods. The five clock periods are represented by (a) to (e).

<table>
<thead>
<tr>
<th>Page</th>
<th>R bits for pages 0-5, clock tick 0</th>
<th>R bits for pages 0-5, clock tick 1</th>
<th>R bits for pages 0-5, clock tick 2</th>
<th>R bits for pages 0-5, clock tick 3</th>
<th>R bits for pages 0-5, clock tick 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
<td>01111000</td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
</tr>
<tr>
<td>2</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td>00100000</td>
<td>10010000</td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
<td>01011000</td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
<td>01010000</td>
<td>00101000</td>
</tr>
</tbody>
</table>

(a) (b) (c) (d) (e)
Aging vs. LRU

- Aging has a finite history of memory
  - Consider aging with an 8-bit counter with value 0. It cannot distinguish between a page referenced 9 clock periods ago, and another referenced 1000 block periods ago.
  - If the counter has infinitely many bits, then it implements LRU exactly.

- 8 bits generally enough
  - If clock period is 20ms, a history of 160ms is perhaps adequate
**Clock Algorithm**

- A variant of second-chance algorithm
- Recall “R” (reference) bit in PTE:
  - Hardware sets R bit on each reference
  - Instead of clearing R periodically (with “clock period” mentioned before) driven by OS timer, clear it at page-fault events
- Arrange physical page frames in a circle with single clock hand. On each page fault:
  - Advance clock hand (not real-time)
  - Check R bit:
    - R=1 → used recently; clear and leave alone
    - R=0 → selected candidate for replacement
- Will always find a page or loop forever?
  - Even if all R bits set, will eventually loop around
    ⇒ FIFO
Clock Algorithm

What if hand moving slowly?
- Not many page faults and/or find page quickly

What if hand is moving quickly?
- Lots of page faults and/or lots of reference bits set

One way to view clock algorithm: Partitioning of pages into two groups: young and old

Animation: [http://gaia.ecs.csus.edu/~zhangd/oscal/ClockFiles/Clock.htm](http://gaia.ecs.csus.edu/~zhangd/oscal/ClockFiles/Clock.htm) (username/passwd: CSC139/csus.os.prin)

(Uncheck “use modified bit” button. Note that it uses “U” instead of “R” for the reference bit.)
N<sup>th</sup> Chance version of Clock Algorithm

**N<sup>th</sup> chance algorithm**: Give page N chances

- OS keeps counter per page: # sweeps
- On page fault, OS checks R bit:
  - R=1 → clear R bit and also set counter to N (ref’ed in last sweep)
  - R=0 → decrement count; if count=0, replace page
- Means that clock hand has to sweep by N times without page being used before page is replaced

**How do we pick N?**

- Why pick large N? Better approx to LRU
  - If N ~ 1K, really good approximation
- Why pick small N? More efficient
  - Otherwise might have to look a long way to find free page

**What about dirty pages?**

- Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
- Common approach:
  - Clean pages, use N=1
  - Dirty pages, use N=2
Allocation of Page Frames

How do we allocate memory (page frames) among different processes?

- Does every process get the same fraction of memory? Different fractions?
- Should we completely swap some processes out of memory?

Each process needs *minimum* number of pages

- Want to make sure that all processes that are loaded into memory can make forward progress
- Example: IBM 370: 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
Possible Replacement Scopes:

- Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Achieve effective utilization of memory through sharing
- Local replacement – each process selects from only its own set of allocated frames
  - Achieve memory isolation among processes
Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes ⇒ process gets 20 frames

- **Proportional allocation** (Fixed Scheme):
  - Allocate according to the size of process
  - Computation proceeds as follows:
    - \( s_i = \text{size of process } p_i \) and \( S = \sum s_i \)
    - \( m = \text{total number of frames} \)
    - \( a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \)

- **Priority Allocation**:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme
  - Possible behavior: If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number
Can we reduce Capacity misses by dynamically changing the number of pages/application?

Establish “acceptable” page-fault rate

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame

Question: What if we just don’t have enough memory?
If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- operating system spends most of its time swapping to disk

**Thrashing** ≡ a process is busy swapping pages in and out

Questions:

- How do we detect Thrashing?
- What is best response to Thrashing?
Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave well

- Not enough memory for Working Set $\Rightarrow$ Thrashing
  - Better to swap out process?
Working-Set Model

**Δ** ≡ working-set window ≡ fixed number of page references

- Example: 10 million references

**WS_i** (working set of Process *P_i*) = total set of pages referenced in the most recent **Δ** (varies in time)

- if **Δ** too small will not encompass entire locality
- if **Δ** too large will encompass several localities
- if **Δ** = ∞ ⇒ will encompass entire program

**D** = Σ|WS_i| ≡ total demand frames

- if **D** > *m* ⇒ Thrashing

  - Policy: if **D** > *m*, then suspend one of the processes
  - This can improve overall system behavior by a lot!

**Animation:**
[http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/workingset.htm](http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/workingset.htm)
What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in

- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages

- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set
Maintaining WS: A Simple Way

- Store page numbers in a shift register of length $k$, and with every memory reference, we do:
  - Shift the register left one position, and
  - Insert the most recently referenced page number on the right
  - The set of $k$ page numbers in the register is the working set.

- Too expensive to do this for each memory reference.

Diagram:

```
[p1, p2, ..., pk]  [p2, p3, ..., p(k+1)]
```

- The oldest page
- Page (k+1) is referenced
- The most recent page
Implementation:
Defining a working set

Since not practical to keep history of past \( \Delta \) memory references, use *working set window* of \( \tau \) ms.

- e.g., instead of defining working set as those pages used during previous 10 million references, define it as pages used during past working set window of 100ms
- Note: not wall-clock time! If a process starts running at time \( T \), and runs for 40ms at time \( T+100ms \), it’s execution time is 40ms. (the other 60ms is used for running other processes)

We use the term *current virtual time* to denote execution time of a process since its start

- Working set of a process is set of pages it referenced during the past \( \tau \) ms of virtual time
Working set algorithm

- Recall: the R bit of a PTE is cleared every clock period. Assume the working set window \( \tau \) ms spans multiple clock periods.

- On every page fault, the page table is scanned to look for a suitable page to evict. The R bit of each PTE is examined.
  - If R=1 the page has been accessed this clock period and is part of WS.
    - Its Time of last use is updated to the present time.
    - If R=1 for all pages in memory, a random page is evicted
  - If R=0 the age (difference between the present time and Time of last use) is determined.
    - If age > \( \tau \), then the page is no longer considered to be part of WS. It may be removed and replaced with the new page
    - If age \( \leq \tau \), then the page is still in WS. If all pages in physical memory are still in WS, the oldest one is chosen for eviction
Working set algorithm example

```
  2084     1
  2003     1
   1980     1
  1213     0
   2014     1
  2020     1
  2032     1
  1620     0
```

Information about one page

R (Referenced) bit

Time of last use

Scan all pages examining R bit:
  if (R == 1)
    set time of last use to current virtual time
  if (R == 0 and age > τ)
    remove this page
  if (R == 0 and age ≤ τ)
    remember the smallest time

Page referenced during this tick

Page not referenced during this tick

Page table

Current virtual time
WSClock algorithm

- Basic working set algorithm requires entire page table to be scanned at every page fault until a victim is located.

- WSClock algorithm is a combination of Clock algorithm and working set algorithm:
  - Instead of clearing R periodically driven by OS timer, clear it at page-fault events.
  - Arrange physical page frames in a circle with single clock hand. On each page fault:
    - Advance clock hand (not real time)
    - Check R bit:
      - R=1 ➔ used recently; clear and leave alone
      - R=0 ➔ additional checking for page age:
        - If age > τ, not in WS; selected candidate for replacement
        - If age ≤ τ, in WS. If all pages in physical memory are still in WS, the oldest one is chosen for eviction

- Worst-case same as working set algorithm, but average case much better

(Note: this is a simplified version of WSClock that does not consider the modified bit. The algorithm in textbook is more complex.)
Operations of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1. (c) and (d) give an example of R = 0 and age > τ.
Replacement algorithms
- **OPT**: Replace page that will be used farthest in future
- **FIFO**: Place pages on queue, replace page at end
- **Second-chance**: giving recently-used pages a second chance
- **LRU**: Replace page used farthest in past

Approximations to LRU
- **NFU & Aging**: Keep track of recent use history for each page
- **Clock Algorithm**:
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- **N\textsuperscript{th}-chance clock algorithm**
  - Give pages multiple passes of clock hand before replacing

Working Set:
- Set of pages touched by a process recently

Working set algorithm:
- Tries to keep each working set in memory

Thrashing: a process is busy swapping pages in and out
- Process will thrash if working set doesn’t fit in memory
- Need to swap out a process
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, good as benchmark</td>
</tr>
<tr>
<td>NRU</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>