Caching and Demand-Paged Virtual Memory

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

The danger of computers becoming like humans is not as great as the danger of humans becoming like computers.
Definitions

Cache
- Copy of data that is faster to access than the original
- **Hit**: if cache has copy
- **Miss**: if cache does not have copy

Cache block
- Unit of cache storage (multiple memory locations)

Temporal locality
- Programs tend to reference the same memory locations multiple times
- Example: instructions in a loop

Spatial locality
- Programs tend to reference nearby locations
- Example: data in a loop
During any phase of execution, the process references only a relatively small fraction of pages.
Memory read requests are sent to the cache. The cache either returns the value stored at that memory location, or it forwards the request onward to the next level cache.
Cache Concept (Write)

- Memory requests are buffered and then sent to the cache in the background.
- Typically, the cache stores a block of data, so each write ensures that the rest of the block is in the cache before updating the cache.
- If the cache is write through, the data is then sent onward to the next level of cache or memory.

Cache write through:
Data is written to the cache and memory.
## Memory Hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache/first level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache/second level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>$100 \mu$s</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory</td>
<td>$100 \mu$s</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core
Main Points

- Can we provide the illusion of near infinite memory in limited physical memory?
  - Demand-paged virtual memory
  - Memory-mapped files
- How do we choose which page to replace?
  - FIFO, MIN, LRU, LFU, Clock
- What types of workloads does caching work for, and how well?
  - Spatial/temporal locality vs. Zipf workloads
Observations

- A complete program does not have to be in memory, because
  - error handling codes are not frequently used
  - arrays, tables, large data structures usually allocate memory more than necessary and many parts are not used at the same time
  - some options and cases may be used rarely

- If they are not needed, why must they be in memory?
Benefits

- Program length is not restricted to real memory size. That is, virtual address size can be larger than physical memory size.
- Can run more programs because space originally allocated for the un-loaded parts can be used by other programs.
- Save load/swap I/O time because we do not have to load/swap a complete program.
Virtual Memory

- **Virtual memory** is the separation of user logical memory from physical memory.
- This permits to have extremely large virtual memory, which makes programming large systems easier.
- Because memory segments can be shared, this further improves performance and save time.
- Virtual memory is commonly implemented with demand paging, demand segmentation or demand paging+segmentation.
Demand Paging

- Virtual memory
- Page table
- Physical memory
- Page frame table

Process A

Valid/invalid or present/absent bit
Demand Paging (Before)

- If a process accesses a page that is not in physical memory, a page fault (trap) is generated and trapped to the kernel.
- The kernel will find the needed page and load it into physical memory.
- The kernel also modifies the page table.
Demand Paging (After)

- The kernel finds the page in virtual memory, brings it into physical memory.
- If there is no available page frame available, the kernel finds an "occupied" one.
- Suppose page A was chosen. The kernel brings page B, which replaces page A.
- The kernel updates the page table.
Address Translation

- Address translation from a *virtual address* to a *physical address* is the same as a paging system.
- However, there is an additional check. If the needed page is not in physical memory (i.e., its valid bit is not set), a page fault (i.e., a trap) occurs.
- If a page fault occurs, we need to do the following:
  - Find an unused page frame. If no such page frame exists, a victim must be found and evicted.
  - **Write** the old page out and **load** the new page in.
  - Update both page tables.
  - Resume the interrupted instruction.
Details of Handling a Page Fault

Trap to the OS // a context switch occurs
Make sure it is a page fault;
If the address is not a legal one then
   address error, return
Find a page frame // page replacement algorithm
Write the victim page back to disk // page out (if modified)
Load the new page from disk // page in
Update both page tables // two pages are involved!
Resume the execution of the interrupted instruction
Hardware Support

- Page Table Base Register, Page Table Length Register, and a Page Table.
- Each entry of a page table must have a valid/invalid bit. *Valid* means that that page is in physical memory. The address translation hardware must recognize this bit and generate a page fault if the valid bit is not set.
- Secondary Memory: use a disk.
- Other hardware components may be needed and will be discussed later.
Too Many Memory Accesses?! 

- Each address reference may use at least two memory accesses, one for page table look up and the other for accessing the page. It may be worse!

See below:

How many memory accesses are there?
May be more than eight!
Performance Issue: 1/2

- Let $p$ be the probability of a page fault, the page fault rate, $0 \leq p \leq 1$.
- The effective access time is 
  \[(1-p) \times \text{memory access time} + p \times \text{page fault time}\]
- The page fault rate $p$ should be small, and memory access time is usually between 10 and 200 nanoseconds.
- To complete a page fault, three components are important:
  - Serve the page-fault trap
  - Page-in and page-out, a bottleneck
  - Resume the interrupted process
Suppose memory access time is 100 nanoseconds, paging requires 25 milliseconds (software and hardware). Then, effective access time is

\[(1-p)\times100 + p\times(25 \text{ milliseconds})\]

\[= (1-p)\times100 + p\times25,000,000 \text{ nanoseconds}\]

\[= 100 + 24,999,900\times p \text{ nanoseconds}\]

If page fault rate is 1/1000, the effective access time is 25,099 nanoseconds = 25 microseconds. It is 250 times slower!

If we wish only 10% slower, effective access time is no more than 110 and \(p=0.0000004\).
Three Important Issues in V.M.

- **Page tables can be very large.** If an address has 32 bits and page size is 4K, then there are \( \frac{2^{32}}{2^{12}} = 2^{20} = (2^{10})^2 = 1 \text{M} \) entries in a page table per process!

- **Virtual to physical address translation must be fast.** This is done with TLB. Remove any TLB entries (i.e., copies of now invalid page table entry).

- **Page replacement.** When a page fault occurs and there is no free page frame, a victim page must be found. If the victim is not selected properly, system degradation may be high.
How Do We Know If Page Has Been Modified?

- Every page table entry has some bookkeeping
  - Has page been modified?
    - Set by hardware on store instruction
    - In both TLB and page table entry
  - Has page been recently used?
    - Set by hardware on in page table entry on every TLB miss

- Bookkeeping bits can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used
Keeping Track of Page Modifications (Before)

Both TLB and page table have the same entry, which shows the address to page A.

There is a “dirty” bit indicating whether a page is modified.

Some systems have a reference bit indicating whether a page has been used since it was loaded into memory. Whenever a location in a page is used (e.g., load, save, etc.), the reference bit is set. Of course, it is modified, the dirty and reference bits are all set.
Both TLB and page table have the same entry, which shows the address to page A.

This page has to be written back to disk if it is replaced by another page.

page modified
Most machines keep dirty/use bits in the page table entry

Physical page is

- Modified if any page table entry that points to it is modified (Modified/Dirty bit)
- Recently used if any page table entry that points to it is recently used (Referenced/Used bit)

On MIPS, simpler to keep dirty/use bits in the core map

- Core map: map of physical page frames
Page Replacement: 1/2

- The following is a basic scheme
  - Find the desired page on disk
  - Find a free page frame in physical memory
    - if there is a free page frame, use it
    - if there is no free page frame, use a page-replacement algorithm to find a victim page
  - write this victim page back to disk and update the page table and page frame table
  - Read the desired page into the selected frame and update page tables and page frame table
  - Restart the interrupted instruction
Page Replacement: 2/2

- If there is no free page frame, two page transfers (i.e., page-in and page-out) may be required.

- A modified bit may be added to a page table entry. The modified bit is set if that page has been modified (i.e., storing info into it). It is initialized to 0 when a page is loaded into memory.

- Thus, if a page is not modified (i.e., modified bit = 0), it does not have to be written back to disk.

- Some systems may also have a referenced bit. When a page is referenced (i.e., reading or writing), its referenced bit is set. It is initialized to 0 when a page is brought in.

- Both bits are set by hardware automatically.
We shall discuss the following page replacement algorithms:

- First-In-First-Out - FIFO
- The Least Recently Used – LRU
- The Optimal Algorithm
- The Second Chance Algorithm
- The Clock Algorithm

The fewer number of page faults an algorithm generates, the better the algorithm performs.

Page replacement algorithms work on page numbers. A string of page numbers is referred to as a page reference string.
The FIFO Algorithm

- The FIFO algorithm always selects the “oldest” page to be the victim.

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

3 frames

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

Page fault = 9  Miss ratio = 9/12 = 75%  Hit ratio = 25%

4 frames

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

Page fault = 10  Miss ratio = 10/12 = 83.3%  Hit ratio = 16.7%
Belady Anomaly

- Intuitively, increasing the number of page frames should reduce the number of page faults.
- However, some page replacement algorithms do not satisfy this “intuition.” The FIFO algorithm is an example.
- **Belady Anomaly**: Page faults may increase as the number of page frames increases.
- FIFO was used in DEC VAX-78xx series and NT because it is easy to implement: append the new page to the tail and select the head to be a victim!
The LRU Algorithm: 1/2

- The LRU algorithm always selects the page that has **not** been used for the **longest** period of time.

<table>
<thead>
<tr>
<th>Frame</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Page fault** = 10  
- **Miss ratio** = 10/12 = 83.3%  
- **Hit ratio** = 16.7%

- **3 frames**

<table>
<thead>
<tr>
<th>Frame</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4.</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Page fault** = 8  
- **Miss ratio** = 8/12 = 66.7%  
- **Hit ratio** = 33.3%

- **4 frames**
The LRU Algorithm: 2/2

- The memory content of 3-frames is a subset of the memory content of 4-frames. This is the inclusion property. With this property, Belady anomaly never occurs. Why?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(extra)
The Optimal Algorithm: 1/2

- The optimal algorithm always selects the page that will not be used for the longest period of time.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- 3 frames
- Page fault = 7
- Miss ratio = 7/12 = 58.3%
- Hit ratio = 41.7%

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- 4 frames
- Page fault = 6
- Miss ratio = 6/12 = 50%
- Hit ratio = 50%
The Optimal Algorithm: 2/2

- The optimal algorithm always delivers the fewest page faults, if it can be implemented. It also satisfies the **inclusion** property (i.e., no Belady anomaly).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

(extra)
The Inclusion Property

Define the following notations:

- $P = <p_1, p_2, \ldots, p_n>$: a page trace
- $m$: the number of page frames
- $M_t(P, \alpha, m)$: the memory content after page $p_t$ is referenced with respect to a page replacement algorithm $\alpha$.

A page replacement algorithm satisfies the **inclusion property** if $M_t(P, \alpha, m) \subseteq M_t(P, \alpha, m+1)$ holds for every $t$.

**Homework:** Inclusion property means no Belady anomaly.
LRU Approximation Algorithms

- FIFO has Belady anomaly, the Optimal algorithm requires the knowledge in the future, and the LRU algorithm requires accurate info of the past.
- The optimal and LRU algorithms are difficult to implement, especially the optimal algorithm. Thus, LRU approximation algorithms are needed. We will discuss three:
  - The Second-Chance Algorithm
  - The Clock Algorithm
  - The Enhanced Second-Chance Algorithm
The second chance algorithm is a FIFO algorithm. It uses the referenced bit of each page.

The page frames are in page-in order (linked-list).

If a page frame is needed, check the oldest (head):

- If its referenced bit is 0, take this one
- Otherwise, clear the referenced bit, move it to the tail, and (perhaps) set the current time. This gives this page frame a second chance.

Repeat this procedure until a 0 referenced bit page is found. Do page-out and page-in if necessary, and move it to the tail.

**Problem:** Page frames are moved too frequently.
Second-Chance Algorithm: 2/3

new page = X

rc = referenced and changed/modified bit pair

before

after

page name

time

reset rc bits
Second-Chance Algorithm: 3/3

new page = X

before

A → B → D → A → F

reset the rc bit
new time

referenced bit = 0

after

F → B → D → X

new page
new time
reset rc bits
The Clock Algorithm: 1/2

- If the second chance algorithm is implemented with a *circular* list, we have the **clock algorithm**.
- A “next” pointer is needed.
- When a page frame is needed, we examine the page under the “next” pointer:
  - If its referenced bit is 0, take it
  - Otherwise, clear the reference bit and advance the “next” pointer.
- Repeat this until a 0 reference bit frame is found.
- Do page-in and page-out, if necessary
The Clock Algorithm: 2/2

new page = X

reset r bit

new page

reset rc bits
Enhanced Second-Chance Algorithm: 1/5

- Four page lists based on their reference-modify bits \((r,c)\) are used:
  - **Q00** - pages were not recently referenced and not modified, the best candidates!
  - **Q01** - pages were changed but not recently referenced. Need a page-out.
  - **Q10** - pages were recently used but clean.
  - **Q11** - pages were recently used and modified. Need a page-out.
Enhanced Second-Chance Algorithm: 2/5

- We still need a “next” pointer.
- When a page frame is needed:
  - Does the “next” frame have 00 combination? If yes, victim is found. Otherwise, reset the referenced bit and move this page to the corresponding list (i.e., Q10 or Q11).
  - If Q00 becomes empty, check Q01. If there is a frame with 01 combination, it is the victim. Otherwise, reset the referenced bit and move the frame to the corresponding list (i.e., Q10 or Q11).
  - If Q01 becomes empty, move Q10 to Q00 and Q11 to Q01. Restart the scanning process.
Enhanced Second-Chance Algorithm: \( \frac{3}{5} \)

\[
\begin{array}{c|c}
Q00 & Q01 & Q10 & Q11 \\
1 & 5 & 8 & 10 \\
2 & 6 & 9 & 11 \\
3 & 7 & 11 & 12 \\
4 & 11 & 11 & 11 \\
\end{array}
\]
Enhanced Second-Chance Algorithm: 4/5

<table>
<thead>
<tr>
<th>Q00</th>
<th>Q01</th>
<th>Q10</th>
<th>Q11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>

- The algorithm consists of two steps: Q00 and Q01.
- The output Q10 is calculated based on the input Q00 and Q01.
- The output Q11 is determined by the output Q10.
This algorithm was used in IBM DOS/VS and MacOS!
Other Important Issues

- Global vs. Local Allocation
- Locality of Reference
- Thrashing
- The Working Set Model
- The Working Set Clock Algorithm
- Page-Fault Frequency Replacement Algorithm
Global vs. Local Replacement

- **Global** replacement allows a process to select a victim from the set of all page frames, even if the page frame is currently allocated to another process.

- **Local** replacement requires that each process selects a victim from its own set of allocated frames.

- With a global replacement, the number of frames allocated to a process may change over time, and, as a result, paging behavior of a process is affected by other processes and may be unpredictable.
Global vs. Local: A Comparison

- With a **global** replacement algorithm, a process cannot control its own page fault rate, because the behavior of a process depends on the behavior of other processes. The same process running on a different system may have a totally different behavior.

- With a **local** replacement algorithm, the set of pages of a process in memory is affected by the paging behavior of that process only. A process does not have the opportunity of using other less used frames. Performance may be lower.

- With a global strategy, **throughput is usually higher**, and is commonly used.
During any phase of execution, the process references only a relatively small fraction of pages.
Thrashing

- **Thrashing** means a process spends more time paging than executing (*i.e.*, low CPU utilization and high paging rate).

- If CPU utilization is too low, the **medium-term scheduler** is invoked to swap in one or more swapped-out processes or bring in one or more new jobs. The number of processes in memory is referred to as the **degree of multiprogramming**.
Degree of Multiprogramming: 1/3

- We cannot increase the degree of multiprogramming arbitrarily as throughput will drop at certain point and thrashing occurs.

- Therefore, the medium-term scheduler must maintain the optimal degree of multiprogramming.
Degree of Multiprogramming: 2/3

1. Suppose we use a global strategy and the CPU utilization is low. The medium-term scheduler will add a new process.

2. Suppose this new process requires more pages. It starts to have more page faults, and page frames of other processes will be taken by this process.

3. Other processes also need these page frames. Thus, they start to have more page faults.

4. Because pages must be paged-in and out, these processes must wait, and the number of processes in the ready queue drops. CPU utilization is lower.
5. Consequently, the medium-term scheduler brings in more processes into memory. These new processes also need page frames to run, causing more page faults.

6. Thus, CPU utilization drops further, causing the medium-term scheduler to bring in even more processes.

7. If this continues, the page fault rate increases dramatically, and the effective memory access time increases. Eventually, the system is paralyzed because the processes are spending almost all time to do paging!
The Working Set Model: 1/4

- The working set of a process at virtual time $t$, written as $W(t, \theta)$, is the set of pages that were referenced in the interval $(t-\theta, t]$, where $\theta$ is the window size.

- $\theta = 3$. The result is identical to that of LRU:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

$page \text{ fault}=10 \quad miss \text{ ratio}=10/12=83.3\% \quad hit \text{ ratio} = 16.7\%$
The Working Set Model: 2/4

- However, the result of $\theta = 4$ is different from that of LRU.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- page fault = 8  miss ratio = 8/12 = 66.7%  hit ratio = 33.3%

only three pages here
The Working Set Model: 3/4

- The Working Set Policy: Find a good \( \theta \), and keep \( W(t, \theta) \) in memory for every \( t \).
- What is the best value of \( \theta \)? This is a system tuning issue. This value can change as needed from time to time.
Unfortunately, like LRU, the working set policy cannot be implemented directly, and an approximation is necessary.

But, the working set model does satisfy the inclusion property.

A commonly used algorithm is the Working Set Clock algorithm, WSClock. This is a good and efficient approximation.
Emulating Modified/Use Bits w/ MIPS Software Loaded TLB

- **MIPS TLB entries have an extra bit:** modified/unmodified
  - Trap to kernel if no entry in TLB, or if write to an unmodified page

- **On a TLB read miss:**
  - If page is clean, load TLB entry as read-only; if dirty, load as rd/wr
  - Mark page as recently used

- **On a TLB write to an unmodified page:**
  - Kernel marks page as modified in its page table
  - Reset TLB entry to be read-write
  - Mark page as recently used