CS450/550
Operating Systems

Lecture 4 memory

Palden Lama
Department of Computer Science
Review: Summary of Chapter 3

- Deadlocks and its modeling
- Deadlock detection
- Deadlock recovery
- Deadlock avoidance
  - Resource trajectories
  - Safe and unsafe states
  - The banker’s algorithm
- Two-phase locking
- More reading: Textbook 3.1 - 3.9
Chapter 4: Memory Management

4.1 Basic memory management
4.2 Swapping
4.3 Virtual memory
4.4 Page replacement algorithms
4.5 Modeling page replacement algorithms
4.6 Design issues for paging systems
4.7 Implementation issues
4.8 Segmentation
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
  - and cheap

- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage

- Memory manager handles the memory hierarchy

Memory is cheap and large in today’s desktop, why memory management still important?
Monoprogramming without Swapping or Paging

- One program at a time, sharing memory with OS

Three simple ways of organizing memory. (a) early mainframes. (b) palmtop and embedded systems. (c) early PC.
Multiprogramming with Fixed Partitions

- Fixed-size memory partitions, without swapping or paging
  - Separate input queues for each partition
  - Single input queue
  - Various job schedulers

What are disadvantages?
Modeling Multiprogramming

- **Degree of multiprogramming:** how many programs in memory?
  - *Independent* process model: CPU utilization = $1 - p^n$
    - A process spends a fraction $p$ of its time waiting for I/O

Assumption on independence is not true!
Example

- under the independent process model, a computer has 32 MB memory, with OS taking 16 MB and each program taking 4 MB. With an 80% average I/O wait, what is the CPU utilization? How much more CPU utilization if adding another 16 MB memory?

\[
1 - 0.8^4 = 60\%
\]
\[
1 - 0.8^8 = 83\%
\]
\[
1 - 0.8^{12} = 93\%
\]
Analysis of Multiprogramming System Performance

- Performance analysis in batching systems with R-R scheduling

(a) Arrival and work requirements of 4 jobs

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival time</th>
<th>CPU minutes needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10:00</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10:10</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>10:15</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10:20</td>
<td>2</td>
</tr>
</tbody>
</table>

(b) # Processes

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU idle</td>
<td>.80</td>
<td>.64</td>
<td>.51</td>
<td>.41</td>
</tr>
<tr>
<td>CPU busy</td>
<td>.20</td>
<td>.36</td>
<td>.49</td>
<td>.59</td>
</tr>
<tr>
<td>CPU/process</td>
<td>.20</td>
<td>.18</td>
<td>.16</td>
<td>.15</td>
</tr>
</tbody>
</table>

(c) CPU utilization for 1 – 4 jobs with 80% I/O wait

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval
Relocation and Protection

- **Relocation**: what address the program will begin in in memory
  - Linker includes a list/bitmap with the binary program during loading
- **Protection**: must keep a program out of other processes’ partitions
- **Use base and limit values**
  - address locations added to base value to map to physical address
  - address locations larger than limit value is an error

What are disadvantages?

Perform an addition and a comparison on every memory reference.
Swapping (1) – Memory Relocation

- Swapping: bring in each process in its *entirety*, M-D-M- ...
- Key issues: allocating and de-allocating memory, keep track of it

Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory (memory holes)

Why not memory compaction?
Swapping (2) – Memory Growing

(a) Allocating space for growing single (data) segment
(b) Allocating space for growing stack & data segment

Why stack grows downward?
Memory Management with Bit Maps

- Keep track of dynamic memory usage: bit map and free lists

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units (what is its desirable size?)
  - shaded regions are free

- Corresponding bit map (searching a bitmap for a run of n 0s?)

- Same information as a list (better using a double-linked list)
Memory Management with Linked Lists

- *De-allocating* memory is to update the list

<table>
<thead>
<tr>
<th>Before X terminates</th>
<th>After X terminates</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) A X B</td>
<td>(a) A B</td>
</tr>
<tr>
<td>(b) A X</td>
<td>(b) A</td>
</tr>
<tr>
<td>(c) X B</td>
<td>(c) B</td>
</tr>
<tr>
<td>(d) X</td>
<td>(d)</td>
</tr>
</tbody>
</table>

*Four* neighbor combinations for the terminating process X
Memory Management with Linked Lists (2)

- How to *allocate* memory for a newly created process (or swapping)?
  - First fit
  - Best fit; surely slow, but why could be more wasteful than *first fit*?
  - Worst fit
  - How about separate P and H lists for searching speedup?
- Example: a block of size 2 is needed for memory allocation

---

(a)

(b)

(c)
Virtual Memory

- Virtual memory: the combined size of the program, data, and stack may exceed the amount of physical memory available.
  - Swapping with overlays; but hard and time-consuming to split a program into overlays by the programmer
  - What to do more efficiently?
Logical program works in its contiguous virtual address space

Address translation done by MMU

Actual locations of the data in physical memory
Paging and Its Terminology

- **Terms**
  - Pages
  - Page frames
  - Page hit
  - Page fault
  - Page replacement

- **Examples:**
  - MOV REG, 0
  - MOV REG, 8192
  - MOV REG, 20500
  - MOV REG, 32780

Page table gives the relation between virtual addresses and physical memory addresses.

```
Virtual address space

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Virtual page

- 60K-64K: X
- 56K-60K: X
- 52K-56K: X
- 48K-52K: X
- 44K-48K: 7
- 40K-44K: X
- 36K-40K: 5
- 32K-36K: X
- 28K-32K: X
- 24K-28K: X
- 20K-24K: 3
- 16K-20K: 4
- 12K-16K: 0
- 8K-12K: 6
- 4K-8K: 1
- 0K-4K: 2

Physical memory address

- 28K-32K
- 24K-28K
- 20K-24K
- 16K-20K
- 12K-16K
- 8K-12K
- 4K-8K
- 0K-4K

Page frame

- 20K – 24K:
  - 20480 – 24575
- 24K – 28K:
  - 24576 – 28671
```
Finding a Page in Memory or in Disk

° Two data structures created by OS on creating a process
  • To track which virtual address(es) use each physical page (in PT)
  • To record where each virtual page is stored on disk (in PT or not)

In practice, could be in two tables.
What is page size?

Why no tags?
• indexed with the virtual page #

Often rounded
Page Tables

Internal operation of MMU with 16 4 KB pages

Two issues:

1. Page table can be large
   * Using registers?

2. Mapping must be fast
   * PT in the main mem.?

Who handles page faults?

OS!
Multi-level Page Tables

(a) 32 bit address with 2 page table fields.  
(b) Two-level page tables

Page table for the top 4M of memory
Structure of a Page Table Entry

Who sets all those bits?

better by hardware! But OS could reset, say for NRU.
Taking advantage of Temporal Locality:

A way to speed up address translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

<table>
<thead>
<tr>
<th>Virtual page number (virtual page #)</th>
<th>Cache</th>
<th>Ref/use</th>
<th>Dirty</th>
<th>Protection</th>
<th>Physical Address (physical page #)</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
</tbody>
</table>

TLB access time comparable to cache access time; much less than Page Table (usually in main memory) access time

Soft miss and Hard miss!

Who handles TLB management and handling, such as a TLB miss?

Traditionally, TLB management and handling were done by MMU Hardware, today, more in software / OS (many RISC machines)
# A TLB Example

A TLB to speed up paging (usually inside of MMU traditionally)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>
Page Table Size

Given a 32-bit virtual address,
4 KB pages,
4 bytes per page table entry (memory addr. or disk addr.)

What is the size of the page table?

The number of page table entries:

\[ 2^{32} / 2^{12} = 2^{20} \]

The total size of page table:

\[ 2^{20} \times 2^2 = 2^{22} \text{ (4 MB)} \]

When we calculate Page Table size, the index itself (virtual page number) is often NOT included!

What if the virtual memory address is 64-bit?
Inverted Page Tables

- Inverted page table: one entry per page frame in physical memory, instead of one entry per page of virtual address space.

Given a 64-bit virtual address, 4 KB pages, 256 MB physical memory

How many entries in the Page Table? How many page frames instead? How large is the Page Table if one entry 8B?

Comparison of a traditional page table with an inverted page table
Inverted Page Tables (2)

- Inverted page table: how to execute virtual-to-physical translation?
  - TLB helps! But what if a TLB miss?

Comparison of a traditional page table with an inverted page table
CA: Integrating TLB, Cache, and VM

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB:

- CPU
- TLB Lookup
- Cache
- Main Memory

VA → PA

TLB misses or Page fault, go to Page Table translation
Page Replacement Algorithms

- Like cache miss, a *page fault* forces choice
  - which page must be removed
  - make room for incoming page

- **Modified page must first be saved**
  - Modified/Dirt bit
  - unmodified just overwritten

- Better not to choose an often used page
  - will probably need to be brought back in soon
  - Temporal locality
Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
  - OS has to know when each of the pages will be referenced next
  - Good as a benchmark for comparison
    - Take two runs, the first run gets the trace, and the second run uses the trace for the replacement
    - Still, it is only optimal with respect to that specific program
Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
  - Example: 0 5 2 0 1 5

- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!!
    - finding, removing, and moving it to the front

- Special hardware:
  - Equipped with a 64-bit counter
  - keep a counter field in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter (NRU)
  - And more simulation alternatives
Hardware LRU (2)

- For a RAM with $n$ page frames, maintain a matrix of $n \times n$ bits; set all bits of row $k$ to 1, and then all bits of column $k$ to 0. At any instant, the row whose binary value is lowest is the least recently used.

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

(a) 

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

(b) 

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

(c) 

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

(d) 

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

(e) 

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

(f) 

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

(g) 

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

(h) 

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

(i) 

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

(j) 

LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3
Not Recently Used (NRU)

- Each page has R bit (referenced; r/w) and M bit (modified)
  - bits are set when page is referenced and modified
  - OS clears R bits periodically (by clock interrupts)

- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified

- NRU removes a page at random
  - From the lowest numbered non-empty class
FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced (the oldest one)
- Disadvantage
  - page in memory the longest (oldest) may be often used
Second Chance Page Replacement Algorithm

OS clears R bits periodically (by clock interrupts)

Second chance (FIFO-extended): looks for an *oldest and not referenced* page in the previous clock interval; if all referenced, FIFO

- (a) pages sorted in FIFO order
- (b) Page list if a page fault occurs at time 20, and A has R bit set (numbers above pages are loading times);
- (c) what if A has R bit cleared?
The Clock Page Replacement Algorithm

- The clock page replacement algorithm differs Second Chance only in the implementation
  - No need to move pages around on a list
  - Instead, organize a circular list as a clock, with a hand points to the oldest page

When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
- R = 0: Evict the page
- R = 1: Clear R and advance hand
Not Frequent Used (NFU)

- NFU (Not Frequently Used): uses a counter per page to track how *often* each page has been referenced, and chose the least to kick out
  - OS adds R bit (0 or 1) to the counter at each clock interrupt
  - Problem: never forgets anything
    - E.g multi-pass compiler
### Aging - Simulating LRU/NFU in Software

- **Aging:** the counters are each shifted right 1 bit before the R bit is added in; the R bit is then added to the leftmost
  - The page whose counter is the lowest is removed when a page fault

<table>
<thead>
<tr>
<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>1 0 1 0 1 1</td>
<td>1 1 0 0 1 0</td>
<td>1 1 0 1 0 1</td>
<td>1 0 0 0 1 0</td>
<td>0 1 1 0 0 0</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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>11000000</td>
<td>11100000</td>
<td>11110000</td>
<td>01111000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td>00100000</td>
<td>10001000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
<td>11000000</td>
<td>01100000</td>
<td>10110000</td>
<td>01011000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
<td>01000000</td>
<td>10100000</td>
<td>01010000</td>
<td>00101000</td>
<td></td>
</tr>
</tbody>
</table>

(a)   (b)   (c)   (d)   (e)

The *aging* algorithm simulates LRU in software, 6 pages for 5 clock ticks, (a) – (e)
The Working Set and Pre-Paging

- Demand paging vs. pre-paging
- Working set: the set of pages that a process is currently using
- Thrashing: a program causing page faults every few instructions
- Observation: working set does not change quickly due to locality
  - Pre-paging working set for processes in multiprogramming

![Diagram showing working set](image)

The working set is the set of pages used by the $k$ most recent memory references. $w(k,t)$ is the size of the working set at time $t$. 

Example: 0, 2, 1, 5, 2, 5, 4
The Working Set Page Replacement Algorithm

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

The working set algorithm
The WSClock Page Replacement Algorithm

Operation of the WSClock algorithm
# Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</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 (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that 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>
## Belady's Anomaly

- **More page frames of memory, fewer page faults, true or not?**
  - FIFO of 0 1 2 3 0 1 4 0 1 2 3 4 in 3-page and 4-page memory

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

9 Page faults

(a)

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

10 Page faults

(b)

(a) FIFO with 3 page frames. (b) FIFO with 4 page frames.
### Modeling Page Replacement: Stack Algorithms

#### Reference string

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

#### State of memory array, \( M \), after each item in reference string is processed

| 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
| 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 7 | 3 | 3 | 5 | 3 | 3 | 3 | 1 | 7 | 1 | 3 | 4 | 1 |
| 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 3 | 4 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 3 | 3 | 7 | 1 | 3 | 4 |
| 0 | 2 | 1 | 3 | 5 | 4 | 6 | 6 | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 |
| 0 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

#### Page faults

| P | P | P | P | P | P | P | P | P | P | P | P |

#### Distance string

| \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | \( \infty \) | 4 | 4 | 2 | 3 | 1 | 5 | 1 | 2 | 6 | 1 | 1 | 4 | 2 | 3 | 5 | 3 |

| 7 | 4 | 6 | 5 |
Design Issues for Paging Systems

- Local page replacement vs. global page replacement
  - How memory be allocated among the competing processes?

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

(a) Original configuration. (b) Local page replacement. (c) Global page replacement.
Design Issues for Paging Systems (2)

- Local page replacement: static allocation
  - What if the working set of some process grows?
  - What if the working set of some process shrinks?
  - The consideration: thrashing and memory utilization

- Global page replacement: dynamic allocation
  - How many page frames assigned to each process
    - Keep monitoring the working set size
    - Allocating an equal share of available page frames
    - Allocating a proportional share of available page frames
    - Or hybrid allocation, using PFF (page fault frequency)
Page Fault Frequency (PFF)

- **PFF**: control the size of allocation set of a process
  - when and how much to increase or decrease a process’ page frame allocation

- **Replacement**: what page frames to be replaced

![Graph showing the relationship between page fault rate and number of page frames assigned.](image)

Page fault rate as a function of the number of page frames assigned
Load Control

- Despite good designs, system may still have thrashing
  - When combined *working sets* of all processes exceed the capacity of memory

- When PFF algorithm indicates
  - some processes need more memory
  - but no processes need less

- Solution:
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming
    - CPU-bound and I/O-bound mixing

Reduce number of processes competing for memory
Small page size

° **Advantages**
  • less unused program in memory (due to *internal fragmentation*)
  • better fit for various data structures, code sections

° **Disadvantages**
  • programs need many pages, larger page tables
  • Long access time of page (compared to transfer time)
  • **Also Maybe more paging actions due to page faults**
Tradeoff: overhead due to page table and internal fragmentation

Where

- \( s \) = average process size in bytes
- \( p \) = page size in bytes
- \( e \) = page entry size in bytes

\[
\text{overhead} = \frac{s \cdot e}{p} + \frac{p}{2}
\]

Optimized/minimized when \( f'(p) = 0 \)

\[
p = \sqrt{2se}
\]
Separate Instruction and Data Spaces

- What if the single virtual address space is not enough for both program and data?
  - Doubles the available virtual address space, and ease page sharing of multiple processes
  - Both addr. spaces can be paged, each has own page table

Is it an issue in today’s 64-bit machines?
Shared Pages

- How to allow multiple processes share the pages when running the same program at the same time?
  - One process has its own page table(s)

Two processes sharing same program sharing its I-page table
Shared Pages (2)

- What to do when a page replacement occurs to a process while other processes are sharing pages with it?
- How share data pages, compared to share code pages?
- UNIX fork() and copy-on-write
  - Generating a new page table point to the same set of pages, but not duplicating pages until...
  - A violation of read-only causes a trap
Cleaning Policy

- Need for a background process, *paging daemon*
  - periodically inspects state of memory
  - To ensure plenty of free page frames

- When too few frames are free
  - selects pages to evict using a replacement algorithm
Implementation Issues

Four times when OS involved with paging

1. **Process creation**
   - determine program size
   - create page table

2. **Process execution**
   - MMU reset for new process
   - TLB flushed (as invalidating the cache)

3. **Page fault time**
   - determine virtual address causing fault
   - swap target page out, needed page in

4. **Process termination time**
   - release page table, pages
Page Fault Handling

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS brings schedules new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored
11. Program continues
Instruction Backup

° The instruction causing the page fault is stopped part way. After OS has fetched the page needed, it must restart the instruction, but \textit{where exactly} the page fault was due to?

- The value of the PC at the time of trap depends on which operand faulted and how the CPU’s microcode has been implemented, which is hard for OS to tell
- Hidden internal register copies PC

\begin{verbatim}
MOVE.L #6(A1), 2(A0)
\end{verbatim}

16 Bits

\begin{tabular}{|c|c|}
\hline
1000 & MOVE \\
\hline
1002 & 6 \\
\hline
1004 & 2 \\
\hline
\end{tabular}

An instruction causing a page fault, say if PC = 1002; how OS knows the content of 1002 is an opcode or an operand? or where the instruction begins?
Virtual memory and I/O occasionally interact

Proc issues call for read from device into buffer
  • while waiting for I/O, another processes starts up
  • That process has a page fault
  • If global page replacement, buffer for the first proc may be chosen to be paged out (in particular, partial DMA transfer)

Need to specify some pages locked
  • Pinning: lock pages engaged in I/O in memory so that they will not be removed (from being target pages)
How to allocate page space on the disk in support of VM?

- Static swap area (pages copied): adding the offset of the page in the virtual address space to the start of the swap area
- Dynamic swapping (page no copied, a table-per-process needed)

(a) Paging to static swap area.
(b) Backing up pages dynamically
Page fault handling with an external pager
Segmentation (1)

- Why to have two or more separate virtual address spaces?

![Diagram showing virtual address space with sections for call stack, parse tree, constant table, source text, and symbol table.]

- How to free a programmer from the issues of expanding and contracting tables?

- One-dimensional address space with growing tables for compiling, one table may bump/interfere into another.
Segmentation (2)

- Segments: many *independent* virtual address spaces
  - A logical entity, known and used by the programmer
  - **Two-dimensional memory**: the program must supply a two-part address, *a segment number* and *an address with an segment*

How about sharing, such as a shared library?

Allows each table to grow or shrink, independently
### Comparison of Segmentation and Paging

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection <strong>modularity</strong></td>
</tr>
</tbody>
</table>
Implementation of Pure Segmentation

- An essential difference of paging and segmentation
  - Segments have different sizes while pages are fixed size!

What is the key difference of segmentation and swapping?

(a)-(d) Development of checkerboarding (external fragmentation) in physical memory, if segments are small; (e) Removal of the checkerboarding by compaction
Segmentation with Paging: MULTICS

- What if the memory is not large enough for a single segment?

- MULTICS (Honeywell 6000)
  - Paged segment with word (4B) addressing
  - *Multi-dim* VM up to $2^{18}$ segments, each up to 64K (32-bit) words
  - 34-bit virtual address (seg #, page #, page offset)
  - Physical memory 16M words (24-bit physical address)
  - Page size: 1024 words; or 64 words (64-word alignment)
  - 18-bit segment number for page table address

---

A 34-bit MULTICS virtual address

- Segment number: 18
- Page number: 6
- Offset within the page: 10
MULTICS Virtual Memory

- One descriptor segment and 36-bit segment descriptors
  - What if the page table of a segment is not in the memory?

How many page tables?

Descriptor segment points to page tables.  

Segment descriptor.
MULTICS Virtual Address → Physical Address

What if the descriptor segment is paged (often it is)?
How to speed up the searching & conversion?

Conversion of a 2-part MULTICS address into a main memory address
### MULTICS TLB

<table>
<thead>
<tr>
<th>Comparison field</th>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
<th>Is this entry used?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Execute only</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Execute only</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

*Simplified version* of the MULTICS TLB (LRU replacement), which has 16 most recently referenced pages.
Segmentation in Pentium resembles MULTICS

- But 16K segments, each up to 1B words
Summary

- Mono-programming -> Multi-programming
- Swapping
- Virtual Memory
  - Paging
  - Page replacement algorithms
  - Design and implementation issues
- Segmentation

- More reading: Chapter 4.1 - 4.9
Homework Assignment 4

Homework (due one week later):
  • Chapter 4 problems: