Virtual Memory
Virtual Memory Mechanisms

If page fault (i.e., present bit is cleared)
- Trap into OS (not handled by hardware. Why?)
- OS selects victim page in memory to replace
- Write victim page out to disk if modified. Add modified ("dirty") bit to PTE
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

What should scheduler do?
Mechanism for Continuing a Process

Continuing a process after a page fault is tricky

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
  - When instruction is being fetched
  - When data is being loaded or stored
- Requires hardware support
  - precise interrupts: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

Complexity depends upon instruction set

- Can faulting instruction be restarted from beginning?
  - Example: `move +(SP), R2`
  - Must track side effects so hardware can roll them back if needed
Virtual Memory Policies

Goal: Minimize number of page faults
- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions
- Page selection
  - When should a page (or pages) on disk be brought into memory?
- Page replacement
  - Which resident page (or pages) in memory should be thrown out to disk?
Average Memory Access Time (AMAT)

Hit% = portion of accesses that go straight to RAM
Miss% = portion of accesses that go to disk first
Tm = time for memory access
Td = time for disk access

AMAT = (Tm) + (Miss% * Td)
Page Selection

When should a page be brought from disk into memory?

**Demand paging:** Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay the cost of a page fault for every newly accessed page
Page Selection

When should a page be brought from disk into memory?

Pre-paging (anticipatory, prefetching): Load page before referenced

• OS predicts future accesses (oracle) and brings pages into memory early
• Works well for some access patterns (e.g., sequential)

• Problems?
Page Selection

When should a page be brought from disk into memory?

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don’t need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix
Page Replacement

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (“dirty” bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3, 2

Miss: 1,2,3

OPT

Metric: Miss count

Three pages of physical memory
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3, 2

Miss: 1,2,3

Metric: Miss count : 3

Three pages of physical memory
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3, 2

Three pages of physical memory

Compulsory misses

Miss: 1,2,3

Hit 1

Hit 2

OPT

Miss count : 3

1 2 3

1 2 3

1 2 3
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3, 2

OPT

Miss: 1,2,3

Hit 1

Hit 2

Miss: 4, Replace: 3

Hit 1

Metric:
Miss count: 4

Three pages of physical memory

capacity miss
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

OPT

Miss: 1,2,3
Hit 1
Hit 2
Miss: 4, Replace: 3
Hit 1
Hit: 4
Hit: 2

Metric:
Miss count: 4

Three pages of physical memory
OPT Replacement Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

Three pages of physical memory

Miss: 1,2,3
Hit 1
Hit 2
Miss: 4, Replace: 3
Hit 1
Hit: 4
Hit: 2
Miss: 3, Replace: 1
Hit: 2

Metric:
AMAT?
Miss count: 5
5 misses, 4 compulsory misses

AMAT = (Tm) + (Miss% * Td)

Assume Tm = 100ns
Assume Td = 1000000 ns (1millisec)

AMAT = ?
FIFO

FIFO: Replace page that has been in memory the longest
• Intuition: First referenced long time ago, done with it now
• Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
• Disadvantage: Some pages may always be needed
FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

OPT

Miss: 1,2,3

1 2 3

Three pages of physical memory

Metric:
Miss count: 3
FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

Three pages of physical memory

Miss: 1,2,3
Hit: 1
Hit: 2
Miss:4, Replace:1

Metric:
Miss count: 4
Page reference string: 1,2,3,1,2,4,1,4,2,3,2

OPT

Miss: 1,2,3

1 2 3

Hit: 1

1 2 3

Hit: 2

1 2 3

Miss: 4, Replace: 1

2 3 4

Miss: 1, Replace: 2

3 4 1

Hit: 4

3 4 1

Three pages of physical memory

Metric:
Miss count: 5
FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

Hit: 1
Hit: 2
Miss: 1,2,3
Miss: 4, Replace: 1
Miss: 1, Replace: 2
Miss: 2, Replace: 3
Miss: 3, Replace: 4

Metric:
Miss count: 7

Three pages of physical memory
FIFO Example

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

OPT

Miss: 1,2,3

Hit: 1

Hit: 2

Miss: 4, Replace: 1

Miss: 1, Replace: 2

Hit: 4

Miss: 2, Replace: 3

Miss: 3, Replace: 4

Hit: 2

Metric:
Miss count: 7

Three pages of physical memory
### FIFO Example

**Page reference string:** 1,2,3,1,2,4,1,4,2,3,2

<table>
<thead>
<tr>
<th>Miss: 1,2,3</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>Hit: 1</td>
<td>1 2 3</td>
</tr>
<tr>
<td>Hit: 2</td>
<td>1 2 3</td>
</tr>
<tr>
<td>Miss: 4, Replace: 1</td>
<td>2 3 4</td>
</tr>
<tr>
<td>Miss: 1, Replace: 2</td>
<td>3 4 1</td>
</tr>
<tr>
<td>Hit: 4</td>
<td>3 4 1</td>
</tr>
<tr>
<td>Miss: 2, Replace: 3</td>
<td>4 1 2</td>
</tr>
<tr>
<td>Miss: 3, Replace: 4</td>
<td>1 2 3</td>
</tr>
<tr>
<td>Hit: 2</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

**Three pages of physical memory**

7 total misses, 4 compulsory misses

**AMAT = (Tm) + (Miss% * Td)**

*Assume Tm = 100ns*

*Assume Td = 1000000 ns (1millisecond)*

**AMAT = ?**
LRU Example – Replace
Least Recently Used

Page reference string: 1,2,3,1,2,4,1,4,2,3,2

<table>
<thead>
<tr>
<th>Miss: 1,2,3</th>
<th>Hit: 1</th>
<th>Hit: 2</th>
<th>Miss:4, Replace:3</th>
<th>Hit: 1</th>
<th>Hit: 4</th>
<th>Hit: 2</th>
<th>Miss:3, Replace:1</th>
<th>Hit: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Three pages of physical memory

Metric:
Miss count
5 total misses
4 compulsory misses

In this example, same as OPT!
Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
  - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
  - **Stack property:** smaller cache a subset of bigger cache

- FIFO: Add more memory, usually have fewer page faults
  - Belady’s anomaly: but there are cases where we have **more** page faults!
Consider access stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

- 3 pages: 9 misses
- 4 pages: 10 misses
Problems with LRU-based Replacement

LRU does not consider frequency of accesses
- Is a page accessed once in the past equal to one accessed \( N \) times?
- Common workload problem:
  - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement
- Problem: LFU can never forget pages from the far past
Implementing LRU

Perfect LRU on Software
• OS maintains ordered list of physical pages by reference time
• When page is referenced: Move page to front of list
• When need victim: Pick page at back of list
• Trade-off: Slow on memory reference, fast on replacement

Perfect LRU on Hardware
• Associate timestamp with each page (e.g., PTE)
• When page is referenced: Associate current system timestamp with page
• When need victim: Scan through PTEs to find oldest timestamp
• Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU
• LRU is an approximation anyway, so approximate more
• Goal: Find an old page, but not necessarily the oldest
Clock Algorithm

Hardware
• Keep use (or reference) bit for each page frame
• When page is referenced: set use bit

Operating System
• Page replacement: Look for page with use bit cleared (has not been referenced for a while)
• Implementation:
  • Keep pointer to last examined page frame ("clock hand")
  • Traverse pages in circular fashion (like a clock)
  • Clear use bits as you search
  • Stop when find page with already cleared use bit, replace this page
Clock:
Look For a Page

Physical Mem:

use= 1
use= 1
use= 0
use= 1

0 1 2 3 ...

clock hand
Clock:
Look For a Page

Physical Mem:

0 1 2 3 ...

use= 0 1 use= 0 1

clock hand
Clock:
Look For a Page

Physical Mem:

use= 0
use= 0
use= 0
use= 1

0 1 2 3 ...

clock hand
Clock:
Look For a Page

Physical Mem: 

 evict page 2 because it has not been recently used
Clock: Look For a Page

Physical Mem:

```
use=0  use=0  use=0  use=1
0     1     2     3 ...
```

clock hand

page 0 is accessed...
Clock:
Look For a Page

Physical Mem:

0 - 1 - 2 - 3 - ...

use = 1
use = 0
use = 0
use = 1

clock hand
Clock:
Look For a Page

Physical Mem:

```
<table>
<thead>
<tr>
<th>use=1</th>
<th>use=0</th>
<th>use=0</th>
<th>use=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

clock hand
Clock:
Look For a Page

Physical Mem:

use=  
1  0  0  1

0 1 2 3  ...

clock hand
Clock:
Look For a Page

Physical Mem:

use= 1
use= 0
use= 0
use= 0

0 1 2 3 ...

clock hand
Clock:
Look For a Page

Physical Mem:

```
use= 0
use= 0
use= 0
use= 0
```

```
0 1 2 3 ...
```

clock hand
Clock: Look For a Page

Physical Mem:

0 1 2 3 ...

use= use= use= use=
0 0 0 0

evict page 1 because it has not been recently used
Clock Extensions

Use modified (“dirty”) bit to prefer to retain modified pages in memory
  • Intuition: More expensive to replace dirty pages
  • Modified pages must be written to disk, clean pages do not have to be
  • First replace pages that have use bit and modified bit cleared

Replace multiple pages at once
  • Intuition: Expensive to run replacement algorithm and to write single block to disk
  • Find multiple victims each time and track free list

Add software counter (“chance”) to track use frequency
  • Intuition: Want to differentiate pages by how much they are accessed
  • Increment software counter if use bit is 0
  • Replace when chance exceeds some specified limit
What if no hardware support?

What can the OS do if hardware does not have \texttt{use} bit (or \texttt{dirty} bit)?

- Can the OS “emulate” these bits?

Think about this question:

- Can the OS get control (i.e., generate a trap) every time \texttt{use} bit should be set? (i.e., when a page is accessed?)
Conclusion

Illusion of virtual memory: Processes can run when the sum of virtual address spaces is larger than physical memory

Mechanism:
- Extend page table entry with “present” bit
- OS handles page faults (or page misses) by reading in the desired page from disk

Policy:
- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) approximate LRU