Virtual Memory
Page Selection
Page Replacement

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

OPT
FIFO
LRU
Page Replacement Comparison

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:

| 0 | 1 | 2 | 3 | ...
|---|---|---|---|---
| use=1 | use=1 | use=0 | use=1 |
Clock:
Look For a Page

Physical Mem:

use= 0
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 use= 0 use= 1
Clock:
Look For a Page

Physical Mem:

0 1 2 3 ...

use= 0 0 0 1

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

...  

page 0 is accessed...
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:

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

...
Clock:
Look For a Page

Physical Mem:

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

`0 1 2 3 ...`

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

clock hand
Clock: Look For a Page

Physical Mem:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\text{use=} & 0 & 0 & 0 \\
\end{array}
\]

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 use bit (or dirty bit)?
  • Can the OS “emulate” these bits?

Think about this question:
  • Can the OS get control (i.e., generate a trap) every time use bit should be set? (i.e., when the 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
Concurrency
Concurrency

Questions answered:
- Why is concurrency useful?
- What is a thread and how does it differ from processes?
- What can go wrong if scheduling of critical sections is not atomic?
Motivation for concurrency: Blocking

- Operations proceeding at the same time: blocking for I/O, while doing other useful work
  - Example: web server
    - Serve the first request by reading a file from disk
    - Serve a second request by running computation
Motivation for Concurrency: Parallelism

http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext
Motivation for Concurrency: Parallelism
Motivation

CPU Trend: Same speed, but multiple cores
Goal: Write applications that fully utilize many cores

Option 1: Build apps from many communicating processes
  • Example: Chrome (process per tab)
  • Communicate via pipe() or similar

Pros?
  • Don’t need new abstractions; good for security

Cons?
  • Cumbersome programming
  • High communication overheads
  • Expensive context switching (why expensive?)
Concurrency: Option 2

New abstraction: **thread**

Threads are like processes, except:
- multiple threads of same process share an address space

Divide large task across several cooperative threads
Communicate through shared address space
Common Programming Models

Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads

- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread

- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)
What state do threads share?
What threads share page directories?
Do threads share Instruction Pointer?
Running thread 1
PTBR
IP

Running thread 2
PTBR
IP

PageDir A
PageDir B

Virt Mem (PageDir A)
CODE HEAP ...

...
Share code, but each thread may be executing different code at the same time

→ Different Instruction Pointers
Do threads share stack pointer?
CPU 1 running thread 1
CPU 2 running thread 2
RAM
PageDir A
PageDir B

Virt Mem (PageDir A)

CODE HEAP STACK 1 STACK 2
threads executing different functions need different stacks
THREAD VS. Process

Multiple threads within a single process share:

• Process ID (PID)
• Address space
  • Code (instructions)
  • Most data (heap)
• Open file descriptors
• Current working directory
• User and group id

Each thread has its own

• Thread ID (TID)
• Set of registers, including Program counter and Stack pointer
• Stack for local variables and return addresses
  (in same address space)
THREAD API

Variety of thread systems exist
  • POSIX Pthreads

Common thread operations
  • Create
  • Exit
  • Join (instead of wait() for processes)
OS Support:  Approach 1

User-level threads: Many-to-one thread mapping
- Implemented by user-level runtime libraries
  - Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads
  - OS thinks each process contains only a single thread of control

Advantages
- Does not require OS support; Portable
- Can tune scheduling policy to meet application demands
- Lower overhead thread operations since no system call

Disadvantages?
- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks
OS Support: Approach 2

Kernel-level threads: One-to-one thread mapping
- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages
- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages
- Higher overhead for thread operations
- OS must scale well with increasing number of threads
Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

Thread 1
%eax: ?
%rip: 0x195

Thread 2
%eax: ?
%rip: 0x195

T1

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:
%eax: ?
%rip: 0x195
%eax: ?
%rip: 0x195

T1

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

T1

process control blocks:

%eax: ?
%rip: 0x195
%eax: ?
%rip: 0x195

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
- $0x9cd4$: 101
- %eax: 101
- %rip = 0x1a2

process control blocks:
- %eax: ?
  - %rip: 0x195
- %eax: ?
  - %rip: 0x195

- 0x195  mov 0x9cd4, %eax
- 0x19a  add $0x1, %eax
- 0x19d  mov %eax, 0x9cd4

Thread Context Switch
Thread Schedule #1

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

process control blocks:
%eax: 101
%rip: 0x1a2
%eax: ?
%rip: 0x195

T2

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
0x9cd4: 101
%eax: 101
%rip = 0x19a

process control blocks:
%eax: 101
%rip: 0x1a2
%eax: ?
%rip: 0x195

T2

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4
Thread Schedule #1

State:
0x9cd4: 101
%eax: 102
%rip = 0x19d

process
%eax: 101
%rip: 0x1a2

control
%eax: ?
%rip: 0x195

blocks:

• 0x195  mov 0x9cd4, %eax
• 0x19a  add $0x1, %eax
• 0x19d  mov %eax, 0x9cd4
Thread Schedule #1

**State:**
- `0x9cd4`: 102
- `%eax`: 102
- `%rip = 0x1a2`

**process**
- `%eax: 101`
- `%rip: 0x1a2`

**control**
- `%eax: ?`
- `%rip: 0x195`

**blocks:**
- `0x195 mov 0x9cd4, %eax`
- `0x19a add $0x1, %eax`
- `0x19d mov %eax, 0x9cd4`

T2 ➔
**Thread Schedule #1**

**State:**
- 0x9cd4: 102
- %eax: 102
- %rip = 0x1a2

**Process Control Blocks:**
- %eax: 101
- %rip: 0x1a2
- %eax: ?
- %rip: 0x195

**Desired Result:**
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T2 ➔

Desired Result!
Let’s consider another schedule...
Thread Schedule #2

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T1 →
Thread Schedule #2

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:
• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4

T1
Thread Schedule #2

**State:**
- 0x9cd4: 100
- %eax: 101
- %rip = 0x19d

**Process Control Blocks:**

- 0x195  mov 0x9cd4, %eax
- 0x19a  add $0x1, %eax
- 0x19d  mov %eax, 0x9cd4

Thread Context Switch
Thread Schedule #2

**State:**
- 0x9cd4: 100
- %eax: ?
- %rip = 0x195

**Process Control Blocks:**
- 0x195: mov 0x9cd4, %eax
- 0x19a: add $0x1, %eax
- 0x19d: mov %eax, 0x9cd4

T2
Thread Schedule #2

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process
control
blocks:

%eax: 101
%rip: 0x19d
%eax: ?
%rip: 0x195

T2

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #2

**State:**
- 0x9cd4: 100
- %eax: 101
- %rip = 0x19d

**Process Control Blocks:**
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T2
Thread Schedule #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4
Thread Schedule #2

State:
- $0x9cd4$: 101
- $%eax$: 101
- $%rip = 0x1a2$

process
- %eax: 101
- %rip: 0x19d

control
- %eax: ?
- %rip: 0x195

blocks:
- $0x195$  mov  $0x9cd4$, %eax
- $0x19a$  add  $0x1$, %eax
- $0x19d$  mov  %eax, $0x9cd4$

Thread Context Switch

T2
Thread Schedule #2

**State:**
- `0x9cd4`: 101
- `%eax`: 101
- `%rip` = 0x19d

**Process control blocks:**
- `%eax`: 101
- `%rip`: 0x19d
- `%eax`: 101
- `%rip`: 0x1a2

**Thread Context Switch:**
- T1

**Instructions:**
- `0x195 mov 0x9cd4, %eax`
- `0x19a add $0x1, %eax`
- `0x19d mov %eax, 0x9cd4`
Thread Schedule #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

%eax: 101
%rip: 0x1a2

Thread 1

Thread 2

T1

• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4
Thread Schedule #2

**State:**
- 0x9cd4: 101
- %eax: 101
- %rip = 0x1a2

**process control blocks:**
- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T1

WRONG Result! Final value of balance is 101
How much is added to shared variable? 3: correct!
Timeline View

Thread 1
mov 0x123, %eax
add %0x1, %eax

mov %eax, 0x123

How much is added?

Thread 2
mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

2: incorrect!
Timeline View

Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

How much is added? 1: incorrect!
Timeline View

Thread 1

mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

Thread 2

mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

How much is added? 3: correct!
Timeline View

Thread 1

mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

Thread 2

mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

How much is added? 2: incorrect!
Non-Determinism

Concurrency leads to non-deterministic results
  • Not deterministic result: different results even with same inputs
  • race conditions: results depend on execution timing

Whether bug manifests depends on CPU schedule!

Passing tests means little

How to program: imagine scheduler is malicious
Assume scheduler will pick bad ordering at some point…
What do we want?

Want 3 instructions to execute as an uninterruptable group
That is, we want them to be atomic

\[
\begin{align*}
\text{mov} & \ 0x123, \ %\text{eax} \\
\text{add} & \ %0x1, \ %\text{eax} \\
\text{mov} & \ %\text{eax}, \ 0x123
\end{align*}
\]

critical section

More general:

Need mutual exclusion for critical sections
- if process A is in critical section C, process B can’t
- (okay if other processes do unrelated work)
Synchronization

Build higher-level synchronization primitives in OS
  - Operations that ensure correct ordering of instructions across threads

Why is this an OS (rather than app) concern?
Motivation: Build them once and get them right

Monitors
  - Locks
  - Semaphores
  - Condition Variables

Loads
  - Stores
  - Test&Set
  - Disable Interrupts
Locks

Goal: Provide mutual exclusion (mutex)

Three common operations:

• Allocate and Initialize
  • Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

• Acquire
  • Acquire exclusion access to lock;
  • Wait if lock is not available (some other process in critical section)
  • Spin or block (relinquish CPU) while waiting
  • Pthread_mutex_lock(&mylock);

• Release
  • Release exclusive access to lock; let another process enter critical section
  • Pthread_mutex_unlock(&mylock);