CONCURRENCY: INTRODUCTION

Shivaram Venkataraman
CS 537, Spring 2019
- Project 2b is out. Due Feb 27th, 11:59
- Project 2a grading in progress

Discussion:

  Makefile tutorial

  How to return values from a syscall
Virtual memory: Summary
Concurrency
  What is the motivation for concurrent execution?
  What are some of the challenges?
RECAP
SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk
   – Slower, cheaper backing store than memory
Process can run when not all pages are loaded into main memory
OS and hardware cooperate to make large disk seem like memory
   – Same behavior as if all of address space in main memory

Requirements:
   – OS must have mechanism to identify location of each page in address space in memory or on disk
   – OS must have policy for determining which pages live in memory and which on disk
VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address
  – if TLB hit, address translation is done; page in physical memory
 Else
  – Hardware or OS walk page tables
  – If PTE designates page is present, then page in physical memory
    (i.e., present bit is cleared)
 Else
  – Trap into OS (not handled by hardware)
  – OS selects victim page in memory to replace
    • Write victim page out to disk if modified (add dirty bit to PTE)
  – OS reads referenced page from disk into memory
  – Page table is updated, present bit is set
  – Process continues execution
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 cost of page fault for every newly accessed page

Prepaging (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)

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 Example

**Page reference string:** ABCABDADDBCBA

<table>
<thead>
<tr>
<th>Metric: Miss count</th>
<th>OPT</th>
<th>FIFO</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td></td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>ABC</td>
<td>ABD</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>ABC</td>
<td>ABD</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>DBC</td>
<td>ABD</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>DCA</td>
<td>ABD</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>DCA</td>
<td>ABD</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>DAB</td>
<td>ABD</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>CAB</td>
<td>CBD</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>CAB</td>
<td>CBD</td>
</tr>
</tbody>
</table>

**Three pages of physical memory**

- **OPT:** 5 misses
- **FIFO:** 7 misses
- **LRU:** 5 misses

**Metric:** Miss count
IMPLEMENTING LRU

Software Perfect LRU
- 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

Hardware Perfect LRU
- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- 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 very 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 awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear use bits as search
  - Stop when find page with already cleared use bit, replace this page
CLOCK: LOOK FOR A PAGE

Physical Mem:

0 1 2 3 ...

clock hand

if $\text{use-bit} = 1$
  clear $\text{use-bit}$
  move clock
else
  select page

Page 0 is accessed

\( \rightarrow \text{set use-bit to 1} \)

Approx CPU
Abstraction: Virtual address space with code, heap, stack

Address translation
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging
- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging, inverted page tables etc.

Larger address spaces: Swapping mechanisms, policies (LRU, Clock)
Virtualization

CPU

Context Switch

Schedulers

Allocation

Segmentation

Paging

Memory

TLBs

Multilevel

Swapping
CONCURRENCY
Moore's Law: Transistor speeds double in 1.5 years.
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

细粒度的 高性能
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?

Page directories, page tables? Shared
Instruction pointer? Not shared
What state do threads share?

- **Code segment**? Shared
- **Stack**? Not stack
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)
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
THREADS DEMO
balance = balance + 1;

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

Thread 1:
%eax: 1
%rip: 0x19d

Thread 2:
%eax: 10 2
%rip: 0x195

directory
control
blocks:

T1
• 0x195 mov 0x9cd4, %eax
• 0x19a add $0x1, %eax
• 0x19d mov %eax, 0x9cd4

T2
0x195 0x199
balance = balance + 1; balance at 0x9cd4

Thread Schedule #2

Thread 1:
- %eax: 100
- %rip: 0x195

Thread 2:
- %eax: 101
- %rip: 0x195

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

Final answer: 101 = Context switch, T1 resumes

State:
- 0x9cd4: 100
- %eax:
- %rip = 0x195
Thread 1
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

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

TIMELINE VIEW

3 increment
Concurrent leads to non-deterministic results

- Different results even with same inputs
- Race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!
WHAT DO WE WANT?

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

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

More general: Need mutual exclusion for critical sections
if thread A is in critical section C, thread B isn’t
(okay if other threads do unrelated work)
Synchronization

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

Motivation: Build them once and get them right
Concurrency is needed for high performance when using multiple cores

Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)

Context switches within a critical section can lead to non-deterministic bugs
LOCKS

Goal: Provide mutual exclusion (mutex)

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);
THREADS DEMO2
NEXT STEPS

Project 2b: Out now

Next class: How to implement locks?
Discussion:
  Makefile tutorial
  How to return values from a syscall