Reminders

* Homework 4 & project 2
  * Project 2 code due midnight today
  * Project writeup & homework 4 tomorrow in lecture
* Midterm on Monday, November 8
  * Midterm review tomorrow in lecture
* Today’s office hours in 006
* Grading
  * Homework 3 back today (average: 84/100), solutions online
  * I still have some old homework/projects, pick up at the end
* Today:
  * Questions
  * Clear up issues on homework 3
  * Project 3 preview
  * Virtual memory stuff

Project 2 – last questions?

Homework 4 questions?

Clearing up synchronization issues

* Monitors
  * How should we use them?
  * Why is this weird inside a monitor?
    * P(mutex);
    * account+=balance;
    * V(mutex);
* Smoker problem
  * What’s wrong if we had:
    * Agent: V(ingr(a)); V(ingr(b));
    * Some smoker: P(ingr(a)); P(ingr(b));
  * General note: always init your semaphores!

Clearing up synchronization issues

* File sharing problem
  * Recall: processes can share a file as long as \( \sum \text{pid} < n \)
  * What’s wrong here:
    * file_open(pid) {
      * if(current + pid \geq n)
        * sum.wait();
      * current += pid;
    }

File sharing – (almost) correct solution

```plaintext
type file = monitor
var space_available: condition
total: integer
procedure file_open(id)
begin
while (total + id \geq n)
  space_available.wait();
total = total + id;
end
procedure file_close(id)
begin
  total = total - id;
  space_available.signal();
end
```
File sharing – correct solution

```plaintext
procedure file_open(id)
begin
    while (total + id >= n)
        space_available.wait(id);
        total = total + id;
    if (total < n - 1)
        space_available.signal();
end
```

```plaintext
procedure file_close(id)
begin
    total = total - id;
    space_available.signal();
end
```

Project 3 preview

- Out right after midterm, due Nov. 19
- Given: vmtrace
  - Takes a memory trace file (also given)
  - Outputs # of references, # of page faults, compulsory faults, page evictions, pageouts.
- Implement an LRU-like page replacement algorithm
- Design and perform an experiment on some aspect of virtual memory

Project 3 experiment

- Have a hypothesis
  - "Big pages are better"
  - "Algorithm y is better"
  - "Prefetching will reduce the number of page faults"
  - "If we understand why x happens, we can fix it"
- Two steps
  - Determine baseline behavior
  - New test
    - Change one aspect of the system, observe differences

Some Ideas

- What is the ideal page size for this trace under different amounts of main memory?
- How much better is page replacement algorithm X than LRU
  - "Real" LRU, FIFO, 2Q, ARC, etc
- How close can we come to LRU without doing any work between page faults?
  - No scanning, constant work per page fault
- How important is recency vs. frequency in predicting page re-use?

Not so good ideas

- What kind of music is made when I convert the address trace to notes?
- Can I make a fractal out of this data?

VM stuff

```
virtual address
```

```
physical memory
```

Often, first page table entry (page zero) is left invalid by the OS
- Any ideas why?
- How can we use paging to set up sharing of memory between two processes?
- TLBs

- Why?
  - No TLB: Average number of memory accesses per virtual addr ref: 2
  - With a TLB (99% hit rate): 0.99 + 0.01 * 2 = 1.01

- More page table/TLB examples
  - Intel x86
    - 4K pages (common) or 4M pages (jumbo pages)
    - Two-level page tables
    - Pentium 4: 64-entry TLB
  - AMD-64
    - still 4K or 2M pages
    - Four (1) PT levels for 4K pages; three for 2M pages
    - Two-level TLB (40 entries/512 entries)
  - Alpha
    - 8K page size
    - Three-level page table, each one page
    - Alpha 21264: 128-entry TLB

- Quick VM exercise
  - Consider a virtual address space of 8 pages of 1024 words each, mapped onto a physical memory of 32 frames
    - Virtual address size (in bits):
    - Physical address size (in bits):

- Intel x86 Memory Architecture
  - 2-Level Page Table
  - 4KB Page Size
  - 32 bit addresses
  - PDE/PTE of 32 bits

- Translation
  - Describe the result of accessing the following virtual addresses:
    - 0x0
    - 0x00803024
    - 0x00001101
    - 0x40000000
    - 2^{32} -- 0x40000000
    - 2^12 -- 0x1000
  - Answer: 0x0, 0x00000002, 0x04
Translation

What is the data stored at virtual address 0x00402004f?

Answer: 0x00300000

Translation

List the physical frames that this address space has direct access to. Is this address space properly isolated from accessing any other frames?

Answers: 0x1000, 0x2000, 0x3000, 0x4000, 0x5000, 0x6000, 0x7000, 0x8000, 0x9000, 0x10000, 0x11000, 0x12000, 0x13000, 0x14000, 0x15000, 0x16000, 0x17000, 0x18000, 0x19000, 0x1a000, 0x1b000, 0x1c000, 0x1d000, 0x1e000, 0x1f000, 0x20000, 0x21000, 0x22000, 0x23000, 0x24000, 0x25000, 0x26000, 0x27000, 0x28000, 0x29000, 0x2a000, 0x2b000, 0x2c000, 0x2d000, 0x2e000, 0x2f000, 0x30000, 0x31000, 0x32000, 0x33000, 0x34000, 0x35000, 0x36000, 0x37000, 0x38000, 0x39000, 0x3a000, 0x3b000, 0x3c000, 0x3d000, 0x3e000, 0x3f000, 0x40000, 0x41000, 0x42000, 0x43000, 0x44000, 0x45000, 0x46000, 0x47000, 0x48000, 0x49000, 0x4a000, 0x4b000, 0x4c000, 0x4d000, 0x4e000, 0x4f000, 0x50000, 0x51000, 0x52000, 0x53000, 0x54000, 0x55000, 0x56000, 0x57000, 0x58000, 0x59000, 0x5a000, 0x5b000, 0x5c000, 0x5d000, 0x5e000, 0x5f000, 0x60000, 0x61000, 0x62000, 0x63000, 0x64000, 0x65000, 0x66000, 0x67000, 0x68000, 0x69000, 0x6a000, 0x6b000, 0x6c000, 0x6d000, 0x6e000, 0x6f000, 0x70000, 0x71000, 0x72000, 0x73000, 0x74000, 0x75000, 0x76000, 0x77000, 0x78000, 0x79000, 0x7a000, 0x7b000, 0x7c000, 0x7d000, 0x7e000, 0x7f000, 0x80000, 0x81000, 0x82000, 0x83000, 0x84000, 0x85000, 0x86000, 0x87000, 0x88000, 0x89000, 0x8a000, 0x8b000, 0x8c000, 0x8d000, 0x8e000, 0x8f000, 0x90000, 0x91000, 0x92000, 0x93000, 0x94000, 0x95000, 0x96000, 0x97000, 0x98000, 0x99000, 0x9a000, 0x9b000, 0x9c000, 0x9d000, 0x9e000, 0x9f000, 0xa0000, 0xa1000, 0xa2000, 0xa3000, 0xa4000, 0xa5000, 0xa6000, 0xa7000, 0xa8000, 0xa9000, 0xaa000, 0xab000, 0xac000, 0xad000, 0xae000, 0xaf000, 0xb0000, 0xb1000, 0xb2000, 0xb3000, 0xb4000, 0xb5000, 0xb6000, 0xb7000, 0xb8000, 0xb9000, 0xba000, 0xbb000, 0xbc000, 0xbd000, 0xbe000, 0xbf000, 0xc0000, 0xc1000, 0xc2000, 0xc3000, 0xc4000, 0xc5000, 0xc6000, 0xc7000, 0xc8000, 0xc9000, 0xca000, 0xcb000, 0xcc000, 0xcd000, 0xce000, 0xcf000, 0xd0000, 0xd1000, 0xd2000, 0xd3000, 0xd4000, 0xd5000, 0xd6000, 0xd7000, 0xd8000, 0xd9000, 0xda000, 0xdb000, 0xdc000, 0xdd000, 0xde000, 0xdf000, 0xe0000, 0xe1000, 0xe2000, 0xe3000, 0xe4000, 0xe5000, 0xe6000, 0xe7000, 0xe8000, 0xe9000, 0xea000, 0xeb000, 0xec000, 0xed000, 0xee000, 0xef000, 0xf0000, 0xf1000, 0xf2000, 0xf3000, 0xf4000, 0xf5000, 0xf6000, 0xf7000, 0xf8000, 0xf9000, 0xfa000, 0xfb000, 0xfc000, 0xfd000, 0xfe000, 0xff000.

TLB Hit Rates

Consider a x86 program consisting of 33% load/store instructions. How many extra memory accesses per instruction executed does this program need when the TLB has a 0%, 95%, or 100% hit rate?

Answers: 1.33 base memory, 0.66 extra, 0% hit rate; 1.33 base memory, 0.66 extra, 95% hit rate; 0.66 base memory, 0 extra, 100% hit rate.