First slide

- Rest of project 2 due next Friday
  - Turnin code + writeup
- Today:
  - Project 2 parts 4-6 (quick)
  - Midterm sample questions & review

1

Project 2 – web server

- web/sioux.c – singlethreaded web server
  - Read in command line args, run the web server loop
- web/sioux_run.c – the webserver loop
  - Open a socket to listen for connections (`listen`)
  - Wait for a connection (`accept`)
  - Handle it
    - Parse the HTTP request
    - Find and read the requested file (www root is ./docs)
    - Send the file back
    - Close the connection
- web/web_queue.c – an empty file for your use

2

What you need to do

- Make the web server multithreaded
  - Create a thread pool
    - A bunch of threads waiting for work
    - Number of threads = command-line arg
  - Wait for a connection
  - Find an available thread to handle connection
    - Current request waits if all threads busy
  - Once a thread grabs onto connection, it uses the same processing code as before.

3

Hints

- Each connection is identified by a socket returned by `accept`
  - Which is just an int
  - Simple management of connections among threads
- Threads should sleep while waiting for a new connection
  - Condition variables are perfect for this
- Don't forget to protect any global variables
  - Use part 2 mutexes, CVs
- Develop + test with pthreads initially
- Mostly modify sioux_run.c and/or your own files
- Stick to the sthread.h interface!

4

Part 6 – Report

- Design discussion & functionality
  - Make it short
- Results
  - Run a few experiments with the new webserver
    - Use given web benchmark: /cse451/projects/webclient
  - Present results in a *graphical easy-to-understand* form.
  - Explain
    - Are the results what you expected?
    - Try to justify any discrepancies you see
  - Answer a few of our questions

5

Project 2 questions?

6

Midterm – top 3 topics

- Scheduling
- Synchronization
- Virtual Memory

7

Scheduling review

- FIFO:
  - \+ simple
  - \- short jobs can get stuck behind long ones; poor I/O device utilization
- RR:
  - \+ better for short jobs
  - \- hard to select right time slice
  - \- poor turnaround time when jobs are the same length
- SJF:
  - \+ optimal (ave. waiting time, ave. time-to-completion)
  - \- hard to predict the future
  - \- unfair
- Multi-level feedback:
  - \+ approximate SJF
  - \- unfair to long running jobs

8

A simple scheduling problem

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

- FIFO Turnaround time:
- FIFO Waiting Time:

9

A simple scheduling problem

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

- FIFO Turnaround Time:
  - A: (10-0) = 10
  - B: (15-1) = 14
  - C: (17-3) = 14
  - (10+14+14)/3 = 12.66
- FIFO Waiting Time:
  - A: 0
  - B: (10-1) = 9
  - C: (15-3) = 12
  - (10+9+12)/3 = 10.33

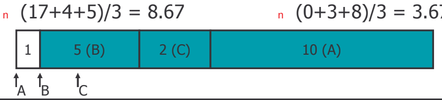
10

A simple scheduling problem

- What about SJF with 1 unit delay?

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

- Ave Turnaround Time:
  - B: 5
  - C: 7-3 = 4
  - A: 1+5+2+10 = 18
  - (17+4+5)/3 = 8.67
- Ave Waiting Time:
  - B: 0
  - C: 5-2 = 3
  - A: 1+5+2 = 8
  - (0+3+8)/3 = 3.67



11

Priority Inversion

- Have three processes
  - P1:Highest priority; P2:Medium; P3:Lowest
- Have this code:

```
P(mutex);
critical section;
V(mutex);
```

- P3 acquires mutex; preempted
- P1 tries to acquire mutex; blocks
- P2 enters the system at medium priority; runs
- P3 never gets to run; P1 never gets to run!!
- This happened on Mars Pathfinder in 1997!
- Solutions?

12

Deadlock-related questions

- Can there be a deadlock with only one process?
- Given two threads, what sequence of calls to transfer(...) causes the following to deadlock?

```
/* transfer x dollars from a to b */
void transfer(account *a, account *b, int x)
  P(a->sema);
  P(b->sema);
  a->balance += x;
  b->balance -= x;
  V(b->sema);
  V(a->sema);
```

13

Some synchronization issues

- Monitors
  - How should we use them?
  - Why is this weird inside a monitor?

    P(mutex);
    account+=balance;
    V(mutex);
- General notes
  - Always init your semaphores!
  - Say which variables are in shared state

14

Another synchronization problem

- File sharing problem
  - Processes can share a file as long as $\sum pid < n$
  - Write a monitor to coordinate the processes

15

File sharing – (almost) correct solution

```
type file = monitor
   var space_available: condition
       total: integer
   procedure file_open(id)
   begin
     if (total + id >= n)
       space_available.wait();
     total = total + id;
   end
   procedure file_close(id)
   begin
     total = total - id;
     space_available.signal();
   end
```

Find the bugs!

16

File sharing – correct solution

```
type file = monitor
   var space_available: conditional_wait
       total: integer
   procedure file_open(id)
   begin
     while (total + id >= n)
       space_available.wait(id);
     total = total + id;
     if (total < n - 1)
       space_available.signal();
   end
   procedure file_close(id)
   begin
     total = total - id;
     space_available.signal();
   end
```

17

Quick VM exercise

- Consider a virtual address space of 8 pages of 512 bytes each, mapped onto a physical memory of 32 frames
  - Virtual address size (in bits):
  - Physical address size (in bits):

18

Another VM sample question

- Given:
  - 32-bit architecture
    - Architecture only supports 30-bit physical addresses
  - 4K pages
  - Master page table has 4K entries
    - Maps 4K 2nd level page tables
  - Draw a picture of virtual address structure and how it gets translated…

19

Intel x86 Memory Architecture

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

Virtual Address Format

10 Bits	10 Bits	12 Bits
PDE Num	PTE Num	Page Offset

PDE/PTE Fromat

20 Bits	11 Bits	1 Bit
Physical Frame Num	Prot, Mod, Ref	Valid

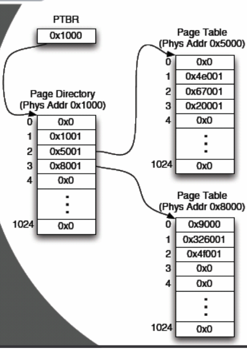
Translation

Describe the result of accessing the following virtual addresses:

0x0
0x00803024
0x00c00136

(2^{22} == 0x400000,
2^{12} == 0x1000)

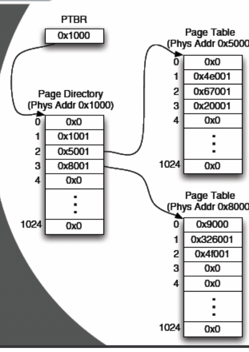
Answers: fault, 0x00020024, fault



Translation

What is the data stored at virtual address 0x00402004?

Answer: 0x0004e001



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, 0x5000, 0x8000, 0x326000, 0x4f000, 0x200000, x67000, 0x4e000. Ignoring kernel/user bits and write protection, the page tables have been made accessable to the address space (virtual addresses 0x00400000-0x004fffff), so a process running in this address space could map-in any physical frame it wanted to.