Processes and Wrap-up

CSCI 237: Computer Organization
35th Lecture, Friday, December 6, 2024

Kelly Shaw

Administrative Details

- Lab #6 due today at 5pm
- "Sample" final on Glow (from S18, not all questions are relevant)
- Extra practice problems added to webpage (solns on Glow)
- Review session on Tues 4-5:30pm in TCL 202
  - Bring questions!
- Final exam
  - Wednesday, December 11, 9:30am – 11:30am (2 hours)
  - Clark Hall 105
- Colloquium talk on Friday at 2:35pm in Wege
  - "Systems research to address societal problems"
  - Aruna Balasubramanian, Stony Brook University
- WICS + Allies Hot Cocoa
  - Monday at 3pm CS Common Area

Last Time

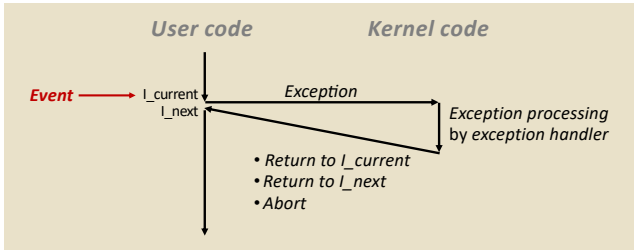
- Dynamic Memory Allocation (Ch 9.9)
  - Tracking Free Blocks
    - Implicit Lists
    - Explicit Lists
    - Segregated Lists
- Exceptional Control Flow
- Exceptions

Today: Processes and Wrap-up

- Exceptions
- Processes
- Current Computer Architecture
- Wrap up

Exceptions

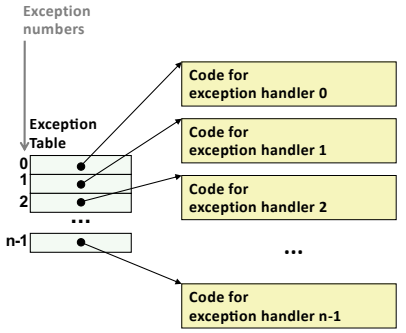
- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



5

5

Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

6

6

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

7

7

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

8

8

System Calls

- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

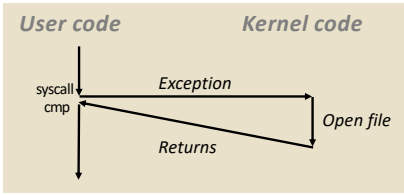
9

9

System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
0000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in
%rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

10

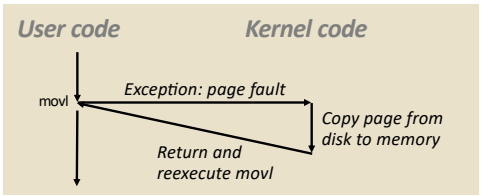
10

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:    c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



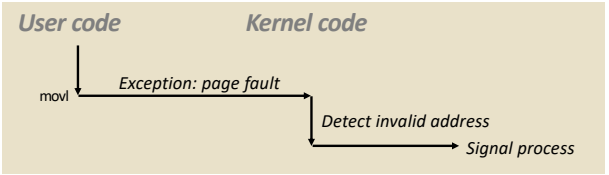
11

11

Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

12

12

Today: Processes and Wrap-up

- Exceptions
- Processes
- Current Computer Architecture
- Wrap up

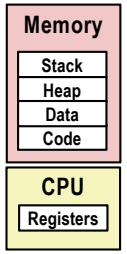
13

13

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"

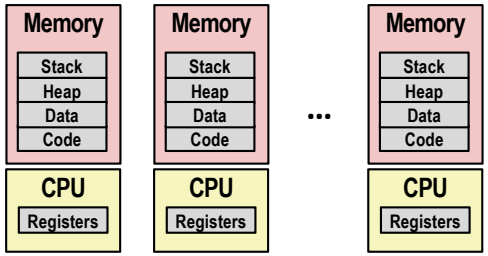
- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*



14

14

Multiprocessing: The Illusion

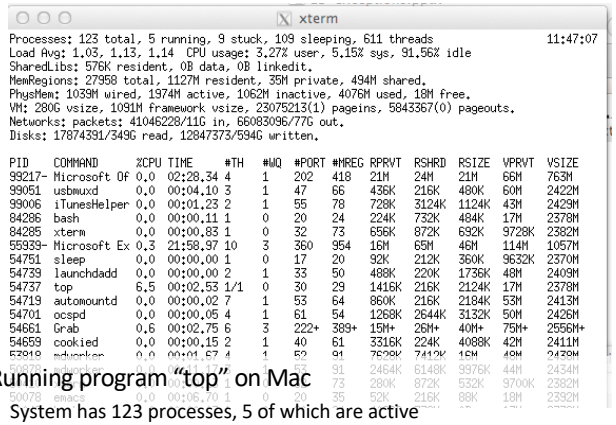


- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

15

15

Multiprocessing Example

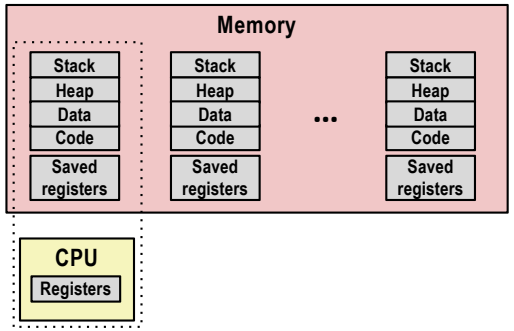


- Running program "top" on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

16

16

Multiprocessing: The (Traditional) Reality

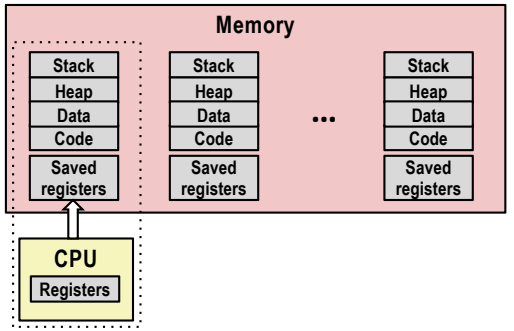


- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for non-executing processes saved in memory

17

17

Multiprocessing: The (Traditional) Reality

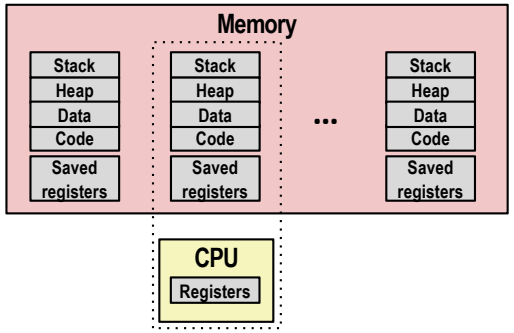


- Save current registers in memory

18

18

Multiprocessing: The (Traditional) Reality

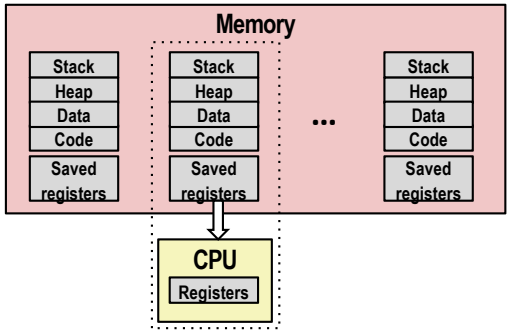


- Schedule next process for execution

19

19

Multiprocessing: The (Traditional) Reality

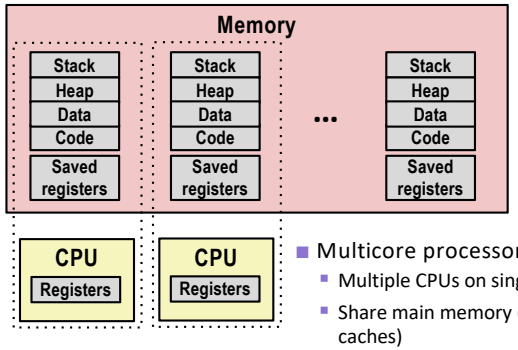


- Load saved registers and switch address space (context switch)

20

20

Multiprocessing: The (Modern) Reality



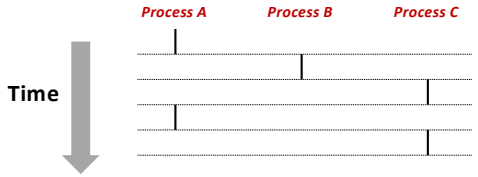
- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel (OS)

21

21

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

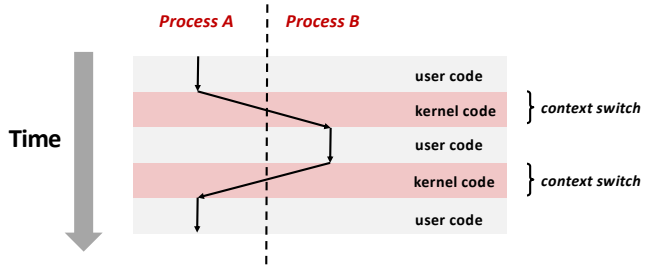


22

22

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



24

24

Today: Processes and Wrap-up

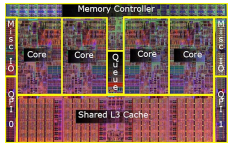
- Exceptions
- Processes
- Current Computer Architectures
- Wrap up

25

25

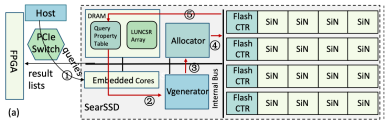
Current Computer Architecture Trends

- Chip multiprocessors
  - Small number of independent computation
  - Data sharing challenging
- GPUs
  - Thousands of almost identical calculations
- Accelerators
  - Specialized computation hardcoded into hardware



Intel multiprocessor


NVIDIA architecture



NDSearch: Accelerating Graph-Traversal-Based Approximate Nearest Neighbor Search through Near Data Processing

Start by Writing a Program in C

```
kshaw — ssh -l kshaw limia.cs.williams.edu — 80×24
File Edit Options Buffers Tools C Help
#include <stdio.h>

void printMessage(char **strings, int num)
{
  for(int i = 0; i < num; i++){
    printf("%s ", strings[i]);
  }
  printf("\n");
}

int main(int argc, char *argv[])
{
  char *array[] = {"Have", "a", "safe", "break!"};

  printMessage(array, 4);

  return 0;
}

-UUU:----F1  hello.c        All L9     (C/*l Abbrev) ----------
```

- Learned how to write and debug C code
  - Ah, pointers, we love you! ❤
- Learned how to use gcc, gdb, make, valgrind, address sanitizer

Compile it to an Executable

gcc! make!

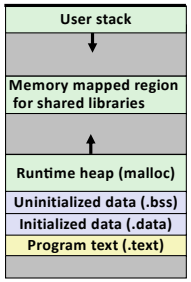
- Learned how to represent data
  - Twos complement
  - Floating point
- Learned how to manipulate data
  - Logical and bitwise manipulation
  - Arithmetic operations (including shifting for ÷ and ✖)
- How to convert C code to assembly code
  - Registers/memory
  - `if`/`else`, `for`/`while`/`do-while`, `switch`
  - Function calls
  - Layout of arrays and structs in memory
- How executables are created
  - Text segment holds instructions
    - Encoded in binary (Y86-64)
  - Data segments hold global data

A Process is Created to Run Our Executable

```
kshaw — ssh -l kshaw limia.cs.williams.edu — 80×24
-> ls
hello*  hello.c  hello.c~
-> ./hello
```

Linux commands!

- Each process has its instructions and data loaded from the executable into a virtual address space
- Each process thinks it has all of memory to itself
  - Enabled by virtual memory
- Each process thinks it has the CPU to itself
  - Exceptional control flow allows process context switching



30

30

Process Starts Running (1)


```
kshaw — ssh -l kshaw limia.cs.williams.edu — 80×24
-> hello
Have a safe break!
->
```

- PC register is set with address of first instruction
- Register storing page table address is set
- Instructions executed on pipelined datapath
  - Stalling or data forwarding pipeline used depending on processor
  - Branch prediction used to reduce control hazards
  - Every virtual address (instructions and data) translated to physical address and sent to memory hierarchy
    - Access TLB, page table, potentially causes page fault
    - Physical addresses sent to L1/L2/DRAM

31

31

Process Starts Running (2)

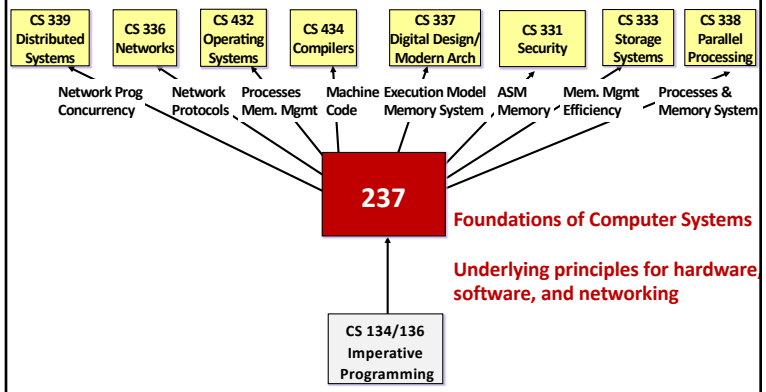
```
kshaw — ssh -l kshaw limia.cs.williams.edu — 80×24
-> hello
Have a safe break!
->
```

- Operating system invoked on traps/faults/syscalls/interrupts
  - Examples: grow heap, page fault, timer interrupt, divide by 0, read virtual address 0
  - If instruction causes exception, flag is set and carried down pipeline until instruction reaches WB stage
    - Process state (i.e. registers, condition codes) saved into memory
    - Exception table used to find starting address in kernel of exception handler
    - Handler code starts running on processor
- Calls to dynamic memory allocator just normal execution of instructions in library code until system calls to grow heap invoke OS

32

32

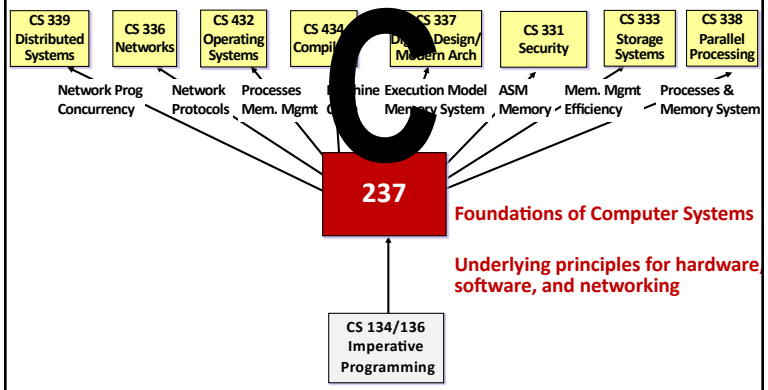
Role within CS Curriculum



33

33

Role within CS Curriculum



34

34

Final Exam

- Closed book, closed notes
- Cumulative with emphasis on material not tested on yet
- Will provide info w/ instructions and registers similar to midterm
- Things to think about
  - Midterm, quizzes, practice problems, in-lecture questions
  - Short answer conceptual questions

35

35

Thank You!

- It's been a challenging semester with the ongoing pandemic.
- This is a tough course with a **lot** of material to learn.
- Be proud of all you accomplished! You learned lots of conceptual knowledge and lots of skills. Both will serve you well in future classes and beyond.
- Thanks for being so great throughout!
- It's been fun getting to know you! Please don't be strangers.

36

36

Student Course Surveys

- Please complete them!

*To access the online evaluations, log into **Glow** (glow.williams.edu) using your regular Williams username and password (the same ones you use for your Williams email account). On your Glow dashboard you'll see a course called "**Course Evaluations**." Click on this and then follow the instructions you see on the screen. If you have trouble finding the evaluation, you can ask a neighbor for help or reach out to ir@williams.edu.*

37

37