TCSS 422: OPERATING SYSTEMS

The Process API & Limited Direct Execution

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

TEXT BOOK COUPON

- 15% off textbook code: AAC72SAVE15
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-hardcover-version-110/hardcover/product-15gjeeky.html?q=three+easy+pieces+operating+systems&page=1&pageSize=4
- With coupon textbook is only $33.79 + tax & shipping

3

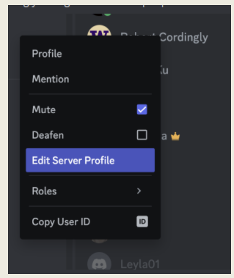
TCSS 422 – OFFICE HRS – WINTER 2026

- **Office Hours plan for Winter:**
- **Tuesday 2:30 - 3:30 pm Instructor Wes, Zoom**
- **Tue/Thur 6:00 - 7:00 pm Instructor Wes, CP 229/Zoom**
- **Tue 6:00 – 7:00 pm GTA Robert, Zoom/Room TBA**
- **Wed 1:00 – 2:00 pm GTA Robert, Zoom/Room TBA**
- Instructor is available after class at 6pm in CP 229 each day

4

TCSS 422 DISCORD SERVER

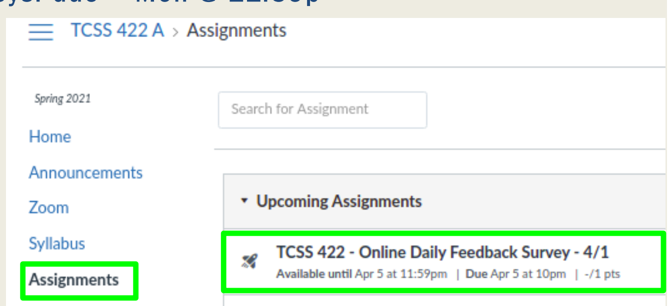
- Please join the TCSS 422 A – Winter 2026 Discord Server
- https://discord.gg/rR2yUDhgmq
- Under Edit Server Profile:
  Please update your 'Server Nickname'
  to your real name or UW NET ID
  THANK YOU



5

ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



6

TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 — 0.5 pts

On a scale of 1 to 10, please classify your perspective on material covered in today's class:

1	2	3	4	5	6	7	8	9	10
Mostly Review To Me				Equal New and Review					Mostly New to Me

Question 2 — 0.5 pts

Please rate the pace of today's class:

1	2	3	4	5	6	7	8	9	10
Slow				Just Right					Fast

7

MATERIAL / PACE

- Please classify your perspective on material covered in today's class
  - 41 of 46 respondents – 89.13%!!
  - 30 in-person, 11 online
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.54 (↑ - previous 6.34)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 4.73 (↓ - previous 5.13)

8

FEEDBACK FROM 1/15

- ***How does "2>&1" work? – redirection of stderr***
- Each process in Linux has 3 files:
- filehandle=0 for standard input (stdin)
- filehandle=1 for standard output (stdout)
- filehandle=2 for standard error (stderr)
- redirect stdin with "<"
- redirect stdout with ">"
- redirect stderrr with "2>"
- &0 refers to stdin, &1 refers to stdout, &2 refers to stderr

```
./a0.sh >output.txt 2>output.err
./a0.sh >output.txt 2>&1
```


9

FEEDBACK - 2

- **time** command – creates a separate process which times the "internal" child command
- **time** command writes time output to /dev/stderr
- ***Confusion***: **time does not write output to internal command's stderr stream**

```
time ./test4 >/dev/null 2>&1
```

- Timing results still go to console because test4's stderr was redirect to /dev/null, not the time command's output

```
{ time ./test4; } 2>/dev/null
```

- To hide the timing output, we need to isolate the time command with {}'s, to redirect time's stderr to /dev/null

10

FEEDBACK - 3

- ***Besides C programs, do programs in other languages like C++ and Java also have a stdin, stdout, and stderr in Linux?***
- YES
- ***In operating systems, what defines fair CPU sharing?***
- Processes with the same priority-level will receive roughly an equal share of time to run on the CPU (called 'CPU timeshare')
- ***Are page faults part of the mechanisms used for lazy-loading?***
- A page fault occurs when a memory page (e.g. 4k) is needed, but it is not present in the physical RAM
  - This could be caused by lazy-loading, because the OS initially loaded only the few pages that were required to run a program

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.11

11

OBJECTIVES – 1/20

- Questions from 1/15
- **C Review Survey – Closed Jan 17 AOE**
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.12

12

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- **Assignment 0 - Update**
- Chapter 4: Linux process data structure - task_struct
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

ASSIGNMENT 0

- ***In the homework, it specifies to use “non-interactive” commands. What does this mean exactly?***
- An non-interactive command does not require any input from the user (i.e. from the keyboard)
- Non-interactive commands and scripts can run entirely on their own without intervention
- These commands are considered “headless” in that they don’t feature a USER INTERFACE, either a GUI, or TUI
- **What is a TUI?**
  - ****Text-based User Interface***
    - **TUI is also a bird** →

TCSS 422 – SET VMS

- Request submitted for School of Engineering and Technology hosted Ubuntu 24.04 VMs for TCSS 422 – Winter 2026

FINISH CHAPTER 4

- Switch to Lecture 3 Slides
- Slides L3.37 to L3.48

CHAPTER 5:
C PROCESS API

17

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - **fork()**, wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

18

fork()

- Creates a new process - think of "a fork in the road"
- "Parent" process is the original
- Creates "child" process of the program from the **current execution point**
- Book says "pretty odd"
- Creates a **duplicate** program instance (these are **processes!**)
- **Copy** of
  - Address space (memory)
  - Register
  - Program Counter (PC)
- Fork returns
  - child PID to parent
  - 0 to child

FORK EXAMPLE

- **p1.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

FORK EXAMPLE - 2

- Non deterministic ordering of execution

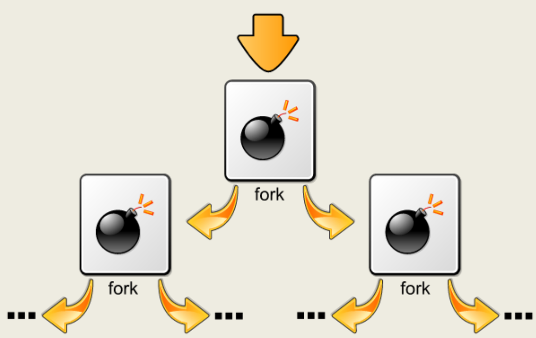
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

- CPU scheduler determines which to run first

:(){ :|: & };:



CLASS BREAK - QUESTION

- ***What is bootstrapping?***
- 'bootstrapping' refers to initialization steps and start-up activities to get a program or system up and ready to run
- For operating systems, bootstrapping is referred to as 'booting'
- For a Linux OS, bootstrapping is the loading of the Linux kernel (at /boot/vmlinuz), and all associated start-up activities like launching the init process (PID 1), etc.

- ***Can you find the size of your Linux kernel in MB ?***

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.23

23

WE WILL RETURN AT 5:05PM

January 20, 2026 TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - coma L4.24

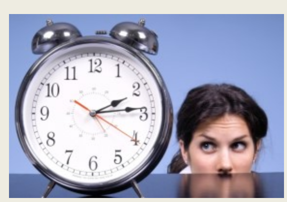
24

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), **wait(),** exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

wait()

- wait(), waitpid()
- Called by parent process
- Waits for a child process to finish executing
- Not a sleep() function
- Provides some ordering to multi-process execution



FORK WITH WAIT

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```


FORK WITH WAIT - 2

- Deterministic ordering of execution

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

FORK EXAMPLE

- Linux example

29

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), **exec()**
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

30

exec()

- Supports running an external program **by "transferring control"**
- 6 types: execl(), execlp(), execle(), execv(), execvp(), execvpe()
- execl(), execlp(), execle(): const char *arg ***(example: execl.c)***

  Provide cmd and args as individual params to the function
  Each arg is a pointer to a null-terminated string
  ODD: pass a variable number of args: (arg0, arg1, .. argn)
- execv(), execvp(), execvpe() ***(example: exec.c)***
  Provide cmd and args as an Array of pointers to strings

  Strings are null-terminated
  First argument is name of command being executed
  Fixed number of args passed in

31

EXEC EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {                  // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {          // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");          // program: "wc" (word count)
        myargs[1] = strdup("p3.c");        // argument: file to count
        myargs[2] = NULL;                  // marks end of array
        …
```


32

EXEC EXAMPLE - 2

```
…
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {                     // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```


EXEC WITH FILE REDIRECTION (OUTPUT)

- **Example:**
https://faculty.washington.edu/wlloyd/courses/tcss422/examples/exec2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[]){
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
        …
```

FILE MODE BITS

```
S_IRWXU
read, write, execute/search by owner
S_IRUSR
read permission, owner
S_IWUSR
write permission, owner
S_IXUSR
execute/search permission, owner
S_IRWXG
read, write, execute/search by group
S_IRGRP
read permission, group
S_IWGRP
write permission, group
S_IXGRP
execute/search permission, group
S_IRWXO
read, write, execute/search by others
S_IROTH
read permission, others
S_IWOTH
write permission, others
```

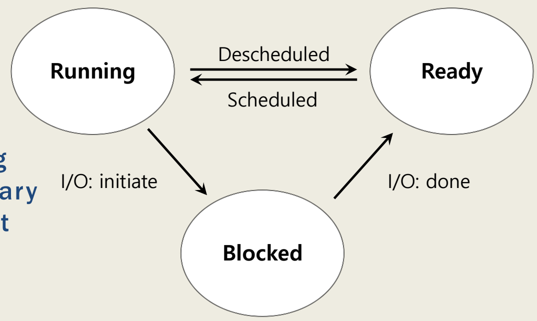

EXEC W/ FILE REDIRECTION (OUTPUT) - 2

```
        …
        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");          // program: "wc" (word count)
        myargs[1] = strdup("p4.c");        // argument: file to count
        myargs[2] = NULL;                  // marks end of array
        execvp(myargs[0], myargs);         // runs word count
    } else {                           // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

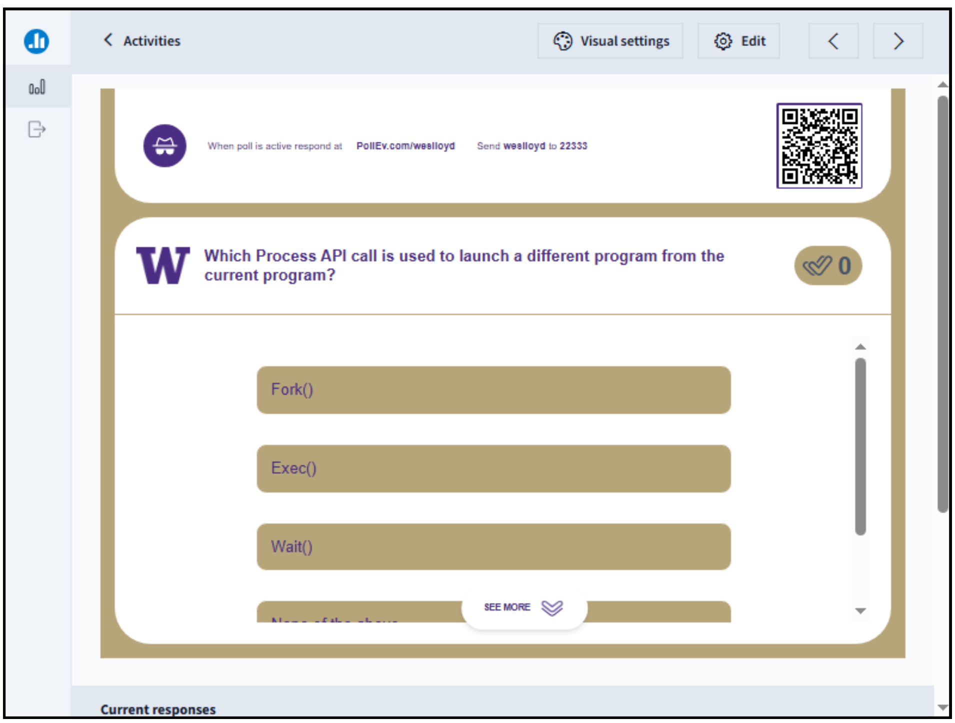
```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

BLOCKING API CALL

- **Blocking API** calls transfer control of the CPU to a kernel thread and force the user process from RUNNING to BLOCKED to wait for a response/outcome
- What blocking APIs have we identified thus far ?
- Does making a blocking API call create a voluntary or non-voluntary context switch ?



37



38

QUESTION: PROCESS API

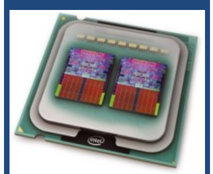
- Which Process API call is used to launch a different program from the current program?

- (a) Fork()
- (b) Exec()
- (c) Wait()
- (d) None of the above
- (e) All of the above

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- **Chapter 6: Limited Direct Execution**
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

CH. 6: LIMITED DIRECT EXECUTION



January 20, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L4.41

41

OBJECTIVES – 1/20

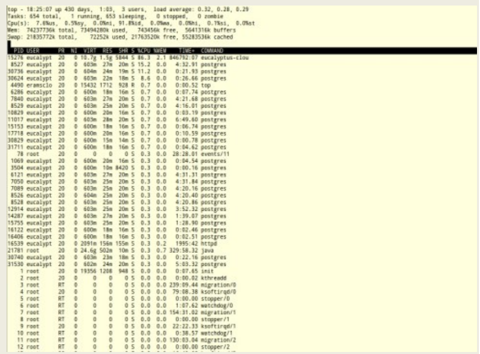
- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - **Direct execution**
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

January 20, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L4.42

42

VIRTUALIZING THE CPU

- How does the CPU support running so many jobs simultaneously?
- **Time Sharing**
- Tradeoffs:
  - Performance
    - Excessive overhead
  - Control
    - Fairness
    - Security
- Both HW and OS support is used



COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

- What if programs could directly control the CPU / system?

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

- What if programs could directly control the CPU / system?

OS	Program
1. Create entry for process list 2. Allocate memory for	
argv 5. Clear registers 6. Execute call `main()`	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

**Without *limits* on running programs,
the OS wouldn't be in control of anything
and would "<u>just be a library</u>"**

DIRECT EXECUTION - 2

- **<u>With direct execution:</u>**

  How does the OS stop a program from running, and switch to another to support **time sharing?**

  How do programs share disks and perform I/O if they are given direct control? Do they know about each other?

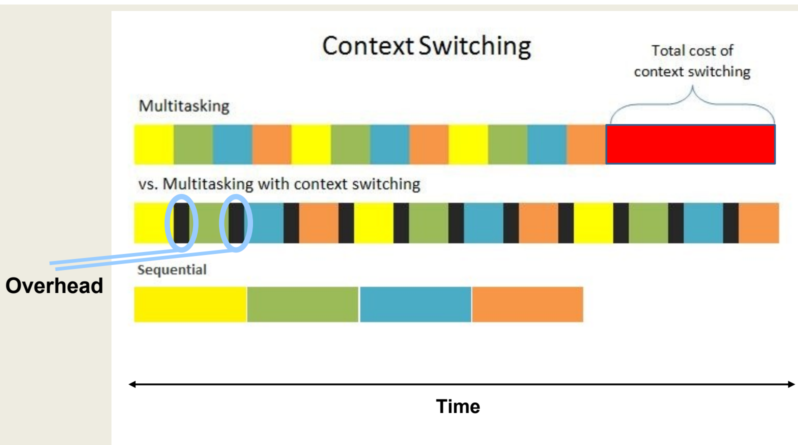
  With direct execution, how can dynamic memory structures such as linked lists grow over time?

CONTROL TRADEOFF

- **Too little control:**
  - No security
  - No time sharing
- **Too much control:**
  - Too much OS overhead
  - Poor performance for compute & I/O
  - Complex APIs (system calls), difficult to use

47

CONTEXT SWITCHING OVERHEAD



48

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - **Limited direct execution**
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

LIMITED DIRECT EXECUTION

- OS implements LDE to support time/resource sharing
- Limited direct execution means "only limited" processes can execute DIRECTLY on the CPU in ***trusted*** mode
- TRUSTED means the process is trusted, and it can do anything… (e.g. it is a system / kernel level process)
- Enabled by ***protected (safe) control transfer***
- CPU supported context switch
- Provides data isolation

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - **CPU modes**
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

51

CPU MODES

- Utilize CPU Privilege Rings (Intel x86)
  - rings 0 (kernel), 1 (VM kernel), 2 (unused), 3 (user)

access ← no access

- **User mode**:
  Application is running, but w/o direct I/O access
- **Kernel mode**:
  OS kernel is running performing restricted operations

52

CPU MODES

- **User mode: ring 3 - untrusted**
  - Some instructions and registers are disabled by the CPU
  - Exception registers
  - HALT instruction
  - MMU instructions
  - OS memory access
  - I/O device access
- **Kernel mode: ring 0 – trusted**
  - All instructions and registers enabled

OBJECTIVES – 1/20

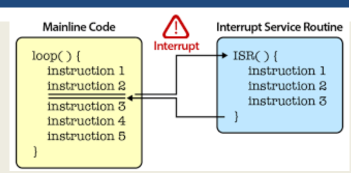
- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - **System calls and traps**
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

SYSTEM CALLS

- Implement restricted "OS" operations
- Kernel exposes key functions through an API:
  - Device I/O (e.g. file I/O)
  - Task swapping: context switching between processes
  - Memory management/allocation: malloc()
  - Creating/destroying processes

55

TRAPS: SYSTEM CALLS, EXCEPTIONS, INTERRUPTS



- Trap: any transfer to kernel mode
- Three kinds of traps
  - **System call:** (planned) user → kernel
    - SYSCALL for I/O, etc.
  - **Exception:** (error) user → kernel
    - Div by zero, page fault, page protection error
  - **Interrupt:** (event) user → kernel
    - Non-maskable vs. maskable
    - Keyboard event, network packet arrival, timer ticks
    - Memory parity error (ECC), hard drive failure

56

EXCEPTION TYPES

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violation	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instruction	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunction	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

57

OS @ boot (kernel mode) | **Hardware**

initialize trap table

remember address of … syscall handler

OS @ run (kernel mode) | **Hardware** | **Program (user mode)**

Create entry for process list
Allocate memory for program
Load program into memory
Setup user stack with argv
Fill kernel stack with reg/PC
return-from -trap

restore regs from kernel stack
move to user mode
jump to main

Run main()
…
Call system
trap into OS

save regs to kernel stack
move to kernel mode
jump to trap handler

Handle trap
Do work of syscall
return-from-trap

restore regs from kernel stack
move to user mode
jump to PC after trap

…
return from main
trap (via `exit()`)

Free memory of process
Remove from process list

58

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of ... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC		
	move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main trap (via `exit()`)
Free memory of process Remove from process list		

Computer BOOT Sequence: OS with Limited Direct Execution

OBJECTIVES – 1/20

- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - **Cooperative multi-tasking**
  - Context switching and preemptive multi-tasking

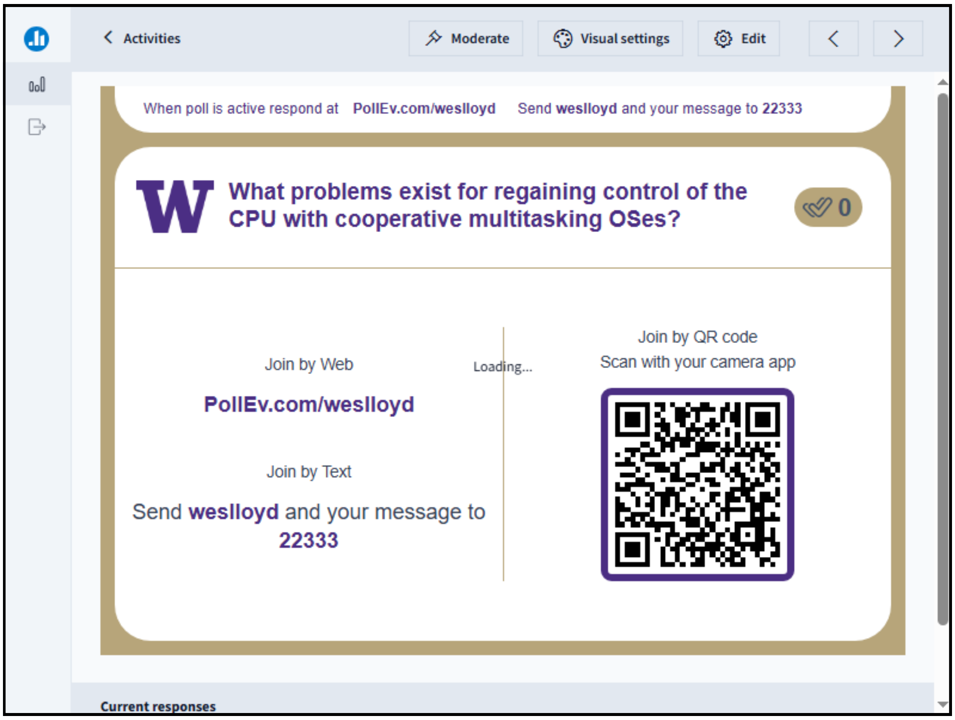
MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- **Cooperative multitasking** (mostly pre 32-bit)
  - < Windows 95, Mac OSX
  - Opportunistic: running programs must give up control
    - User programs must call a special **yield** system call
    - When performing I/O
    - Illegal operations
  - (POLLEV)
    What problems could you for see with this approach?

MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- Cooperative multitasking (mostly pre 32-bit)
  - < W
  - Op
    - Illegal operations
  - (POLLEV)
    What problems could you for see with this approach?

A process gets stuck in an infinite loop.
→ **Reboot the machine**



63

QUESTION: MULTITASKING

- What problems exist for regaining the control of the CPU with cooperative multitasking OSes?

64

MULTITASKING - 2

- **Preemptive multitasking** (32 & 64 bit OSes)
- >= Mac OSX, Windows 95+

- Timer interrupt
  - Raised at some regular interval (in ms)
  - Interrupt handling
    1. Current program is halted
    2. Program states are saved
    3. OS Interrupt handler is run (kernel mode)

- (PollEV) What is a good interval for the timer interrupt?

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.65

65

MULTITASKING - 2

- Preemptive multitasking (32 & 64 bit OSes)
- >= Mac OSX, Windows 95+

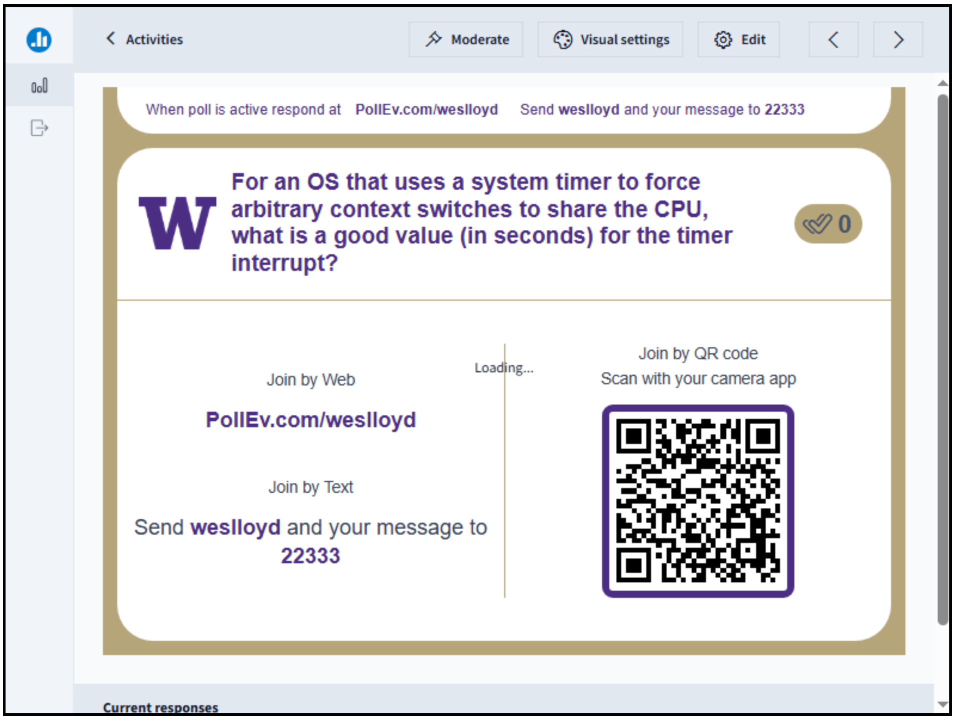
- Timer
  - Rais
  - Inter
    1. Current program is halted
    2. Program states are saved
    3. OS Interrupt handler is run (kernel mode)

A **timer interrupt** gives OS the ability to run again on a CPU.

- (PollEV) What is a good interval for the timer interrupt?

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.66

66



67

QUESTION: TIME SLICE

- For an OS that uses a system timer to force arbitrary context switches to share the CPU, what is a good value (in seconds) for the timer interrupt?

January 20, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L4.68

68

QUESTION: TIME SLICE

- For an OS that uses a system timer to force arbitrary context switches to share the CPU, what is a good value (in seconds) for the timer interrupt?
  - Typical time slice for process execution is **10 to 100 milliseconds**
  - Typical context switch overhead is (*switch between processes*) **0.01 milliseconds**
    - 0.1% of the time slice (1/1000th)

OBJECTIVES – 1/20

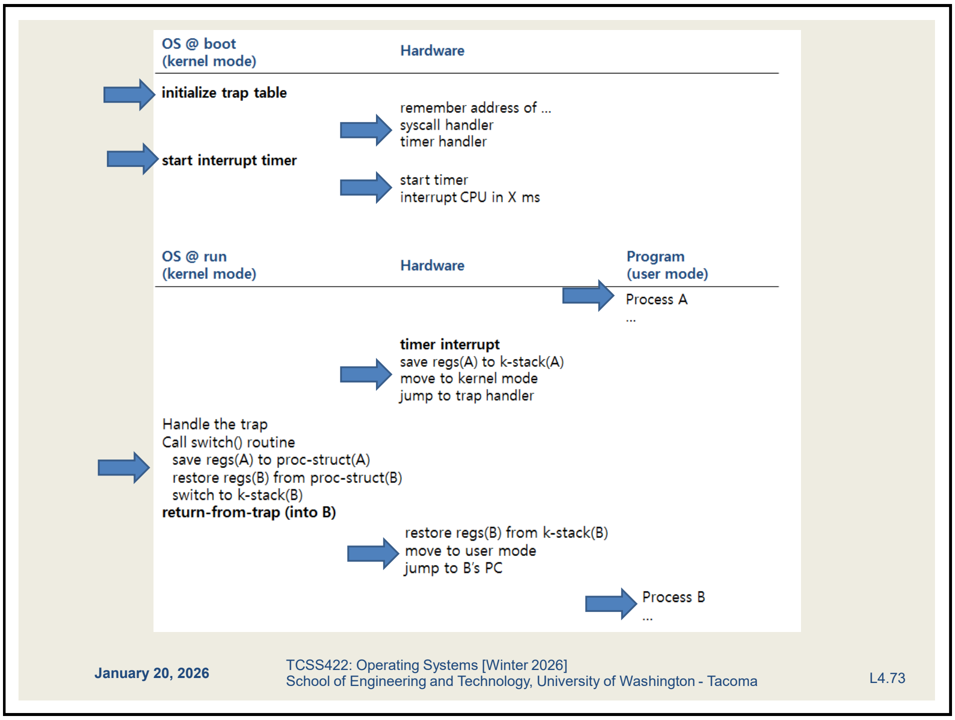
- Questions from 1/15
- C Review Survey – Closed Jan 17 AOE
- Assignment 0 - Update
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - **Context switching and preemptive multi-tasking**

CONTEXT SWITCH

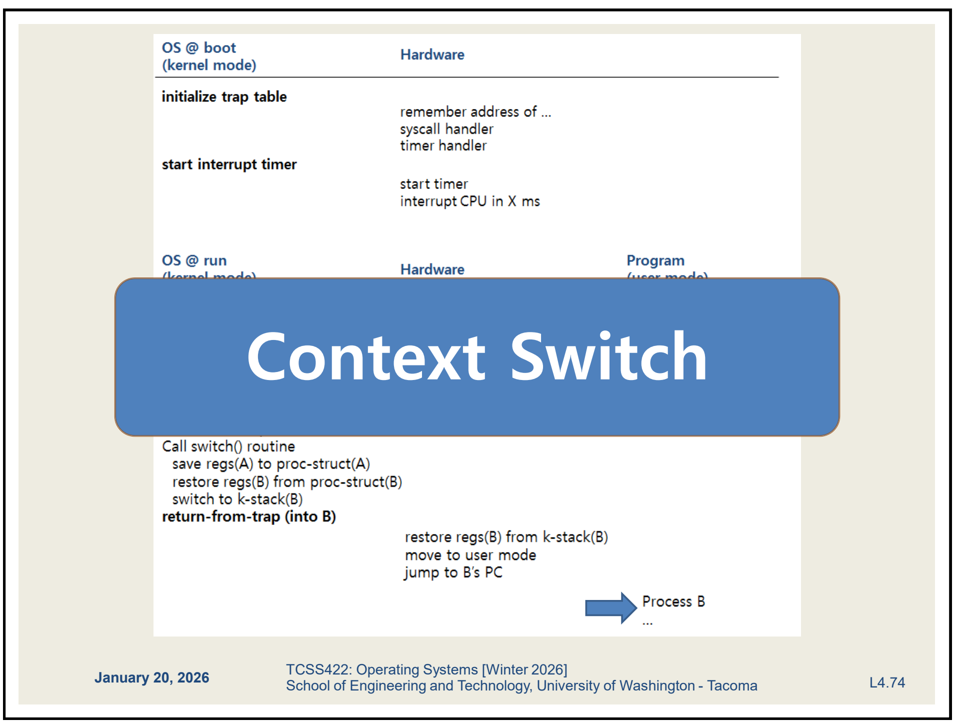
- **Preemptive multitasking** initiates "trap" into the OS code to determine:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS performs a **context switch** swapping out the current process for a new one.

CONTEXT SWITCH - 2

1. Save register values of the current process to its kernel stack
   - General purpose registers
   - PC: program counter (instruction pointer)
   - kernel stack pointer
2. Restore soon-to-be-executing process from its kernel stack
3. Switch to the kernel stack for the soon-to-be-executing process


OS @ boot
(kernel mode)
Hardware
initialize trap table
remember address of …
syscall handler
timer handler
start interrupt timer
start timer
interrupt CPU in X ms
OS @ run
(kernel mode)
Hardware
Program
(user mode)
Process A
…
timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler
Handle the trap
Call switch() routine
save regs(A) to proc-struct(A)
restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)
restore regs(B) from k-stack(B)
move to user mode
jump to B's PC
Process B
…
January 20, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L4.73

73


OS @ boot
(kernel mode)
Hardware
initialize trap table
remember address of …
syscall handler
timer handler
start interrupt timer
start timer
interrupt CPU in X ms
OS @ run
Hardware
Program
Context Switch
Call switch() routine
save regs(A) to proc-struct(A)
restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)
restore regs(B) from k-stack(B)
move to user mode
jump to B's PC
Process B
…
January 20, 2026
TCSS422: Operating Systems [Winter 2026]
School of Engineering and Technology, University of Washington - Tacoma
L4.74

74

INTERRUPTED INTERRUPTS

- What happens if during an interrupt (trap to kernel mode), another interrupt occurs?
- Linux
  - < 2.6 kernel: non-preemptive kernel
  - >= 2.6 kernel: preemptive kernel

75

PREEMPTIVE KERNEL

- Use “locks” as markers of regions of non-preemptibility (non-maskable interrupt)
- Preemption counter (`preempt_count`)
  - begins at zero
  - increments for each lock acquired (not safe to preempt)
  - decrements when locks are released
- Interrupt can be interrupted when `preempt_count=0`
  - It is safe to preempt (maskable interrupt)
  - the interrupt is more important

76

QUESTIONS

77