


TCCS 422: OPERATING SYSTEMS

The Process API & Limited Direct Execution

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



April 4, 2024 TCCS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma

1

OBJECTIVES – 4/4

- **Questions from 4/2**
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024 TCCS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma L4.2

2

TEXT BOOK COUPON

- 15% off textbook code: **POETRY15** (through Friday Apr 5)
- <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

April 4, 2024 TCCS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma L4.3

3

TCCS 422 – OFFICE HRS – SPRING 2024

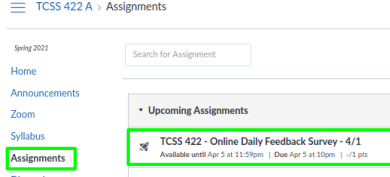
- ****Tuesdays after class until 7:00pm****
 Hybrid (in-person/Zoom)
 - This session will be in person in CP 229.
 - Zoom will be monitored when no student is in CP 229.
- Thursdays after class until 7:00pm – Hybrid (in-person/Zoom)
 - Additional office time will be held on Thursdays after class when there is high demand indicated by a busy Tuesday office hour
 - When Thursday Office Hours are planned, Zoom links will be shared via Canvas
 - Questions after class on Thursdays are always entertained even when the formal office hour is not scheduled

April 4, 2024 TCCS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma L4.4

4

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



April 4, 2024 TCCS422: Computer Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma L4.5

5

TCCS 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 dislike to me		Neutral						Mostly like 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		

April 4, 2024 TCCS422: Computer Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma L4.6

6

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (32 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.56 (↑ - previous 6.49)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.38 (↑ - previous 5.34)**

April 4, 2024 TCSS422: Computer Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.7

7

FEEDBACK FROM 4/2

- I am Interested In Differentiating the responsibilities between the run-time stack and the memory heap. I understand the gist of these ideas but still feel unfamiliar with their specifics.***
 - What data is stored on the heap?
 - What data is stored on the stack?
- What is the difference between voluntary and involuntary context switches (C/S)?***
 - A voluntary C/S occurs when a process performs privileged operations such as I/O that BLOCK and WAIT for a response
 - This is considered a voluntary C/S because the user program has elected to perform the I/O and needs to WAIT anyways.
 - It's a perfect time to for the CPU to C/S and perform other work
- UNCLEAR: The processes of going from running to blocked to ready then back to running.***

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.8

8

FEEDBACK FROM 4/4

- I understand that using malloc() while a program is running requires using free() if we want to prevent memory leaks, but isn't it true that most modern operating systems recover the allocated memory after a program exits?***
- YES, when the process ends, the operating system will claim all memory allocated for the code, stack, heap, and data segments
- If the program only runs for a short time, then it may be acceptable not to "free()" memory on the heap
- The issue is with programs that run forever (i.e. **servers**)
 - Web applications may "run forever"
 - if there is a memory leak in a web application, it could cause the web application server to eventually crash

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.9

9

FEEDBACK - 2

- I originally thought one of the main reasons we program in C on our Virtual Machines was so that we did not accidentally use malloc() and cause permanent damage to our memory by making it nonreusable.***
- When writing privileged kernel-level code, you may use "kmalloc()" which stands for "kernel malloc".
- Errors with dynamic memory allocation in the kernel may result in the corruption of the kernel's memory which is catastrophic if not recoverable
- If a user program fails, it is no big deal to the system
- If the kernel errors, the system may go down

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.10

10

FEEDBACK - 3

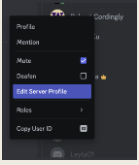
- We covered context switches quickly so I wonder how exactly they are implemented and better examples of where we use them?***
- A programmer can "use" a voluntary context switch by performing a blocking operation where the system must wait for I/O etc. In this case the CPU is not busy, and is reclaimed for some other process by the OS
- Otherwise the user does not **cause** or **enact** a context switch. Context switches are generated by the operating system when a process runs for more than a "time slice" which is from ~ 3 to 10 milliseconds depending how busy the system is
- We will cover context switches in more detail in Chapter 6

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.11

11

TCSS 422 DISCORD SERVER

- Please join the TCSS 422 A – Spring 2024 Discord Server
- <https://discord.gg/H7PPZ5ArFW>
- Under Edit Server Profile: Please update your 'Server Nickname' to your real name or UW NET ID
THANK YOU



April 4, 2024 TCSS422: Operating Systems (Spring 2024)
 School of Engineering and Technology, University of Washington - Tacoma L4.12

12


OBJECTIVES – 4/4

- Questions from 4/2
- **C Review Survey – Closes Friday April 5**
- Assignment 0
- 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

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.13

13

C REVIEW SURVEY - AVAILABLE THRU 4/7



April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.14

14

OBJECTIVES – 4/4


- Questions from 4/2
- C Review Survey – Closes Friday April 5
- **Assignment 0**
- 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

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.15

15

FEEDBACK ON 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



April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.16

16

TCSS 422 – SET VMS

▪ Request submitted for School of Engineering and Technology hosted Ubuntu 22.04 VMs for TCSS 422 – Spring 2024

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.17

17


OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024 TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma L4.18

18

CHAPTER 5: C PROCESS API




April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.19

19

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



April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.20

20

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;
}
    
```

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.21

21

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

```

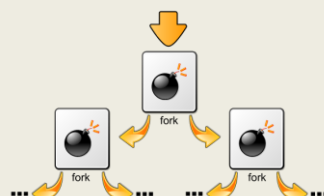
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
                
```

- CPU scheduler determines which to run first

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.22

22

:(){ :|: & };:



April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.23

23

OBJECTIVES - 4/4


- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- Chapter 5: Process API
 - fork(), **walk()**, 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**

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.24

24

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



April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.25

25

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;
}
    
```

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.26

26

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>
    
```

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.27

27

FORK EXAMPLE

- Linux example

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.28

28

OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.29

29

exec()

- Supports running an external program by **“transferring control”**
- 6 types: execl(), execlp(), execlx(), execv(), execvp(), execve()
- execl(), execlp(), execlx(): 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(), execve() (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

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.30

30

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
        ...
    }
}
```

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.31

31

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>
```

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.32

32

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);
        ...
    }
}
```

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.33

33

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
```

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.34

34

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>
```

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.35

35

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

Options: Fork(), Exec(), Wait(), SEE MORE

Current responses: Response options, Count, %

36

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

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.37

37


OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.38

38

CH. 6: LIMITED DIRECT EXECUTION



April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.39

39

OBJECTIVES – 4/4

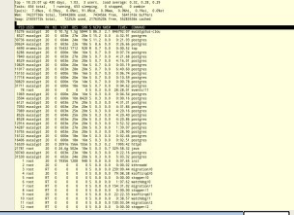
- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- Chapter 5: Process API
 - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
 - **Direct execution**
 - Limited direct execution
 - CPU modes
 - System calls and traps
 - Cooperative multi-tasking
 - Context switching and preemptive multi-tasking

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.40

40

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



April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.41

41

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()		
9. Free memory of process		
10. Remove from process list		
		7. Run main()
		8. Execute return from main()

April 4, 2024
TCSS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.42

42

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	
Without <i>limits</i> on running programs, the OS wouldn't be in control of anything and would "just be a library"	
5. Clear registers	7. Run main ()
6. Execute call main ()	8. Execute return from main ()
9. Free memory of process	
10. Remove from process list	

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.43

43

DIRECT EXECUTION - 2

▪ **With direct execution:**

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?

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.44

44

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

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.45

45

CONTEXT SWITCHING OVERHEAD

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.46

46

WE WILL RETURN AT 2:40PM

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.47

47

OBJECTIVES - 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024 TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L4.48

48

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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.49
---------------	---	-------

49

OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.50
---------------	---	-------

50

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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.51
---------------	---	-------

51

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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.52
---------------	---	-------

52

OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.53
---------------	---	-------

53

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

April 4, 2024	TCSS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma	L4.54
---------------	---	-------

54

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

Mainline Code

```
loop() {
  instruction 1
  instruction 2
  instruction 3
  instruction 4
  instruction 5
}
```

Interrupt Service Routine

```
ISR() {
  instruction 1
  instruction 2
  instruction 3
}
```

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.55

55

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
Kernel operating system	Synchronous	User request	Nonmaskable	Between	Resume
Trapping instruction execution	Synchronous	User request	User maskable	Between	Resume
Nonfatal	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
Illegal memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violation	Synchronous	Coerced	User maskable	Within	Resume
Using undefined instruction	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunction	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.56

56

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 args
 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

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.57

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 args
 Fill kernel stack with reg/PC
 return-from-trap

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

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.58

58

OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.59

59

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 see with this approach?

April 4, 2024
TCCS422: Operating Systems (Spring 2024)
School of Engineering and Technology, University of Washington - Tacoma
L4.60

60

MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- Cooperative multitasking (mostly pre-32-bit)
 - <
 - Op
 - A process gets stuck in an infinite loop.
 - **Reboot the machine**
 - When performing I/O
 - Illegal operations
- (POLLEV) What problems could you see with this approach?

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.61

61

62

QUESTION: MULTITASKING

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

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.63

63

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?

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.64

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?

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.65

65

66

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?

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.67

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?
 - 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)

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.68

68

OBJECTIVES – 4/4

- Questions from 4/2
- C Review Survey – Closes Friday April 5
- Assignment 0
- 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**

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.69

69

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.

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.70

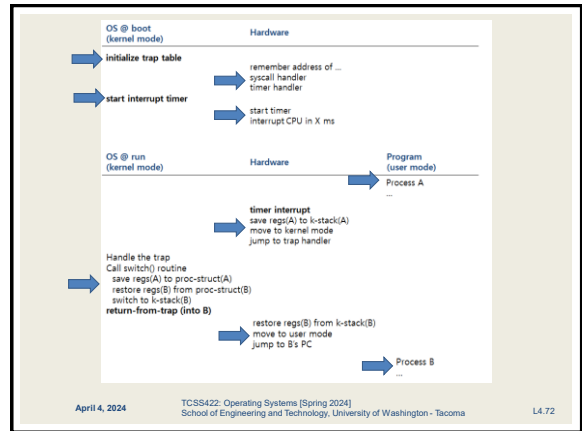
70

CONTEXT SWITCH - 2

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

April 4, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L4.71

71



72

The diagram illustrates the context switch process. It is divided into two main phases: 'OS @ boot (kernel mode)' and 'OS @ run (user mode)'. In the boot phase, the OS initializes a trap table (remembering syscall handler addresses), starts an interrupt timer, and interrupts the CPU. In the run phase, a context switch routine is called, which saves the current process's registers (A) and stack pointer (B), switches to the kernel stack, and returns from the trap into the user mode of the next process (B). The hardware and program components are also indicated.

Context Switch

Call switch() routine
 save reg(A) to proc-struct(A)
 restore reg(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)
 restore reg(B) from k-stack(B)
 move to user mode
 jump to B's PC

Process B

April 4, 2024 TCCS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma L4.73

73

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

April 4, 2024 TCCS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma L4.74

74

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

April 4, 2024 TCCS422: Operating Systems (Spring 2024) School of Engineering and Technology, University of Washington - Tacoma L4.75

75

QUESTIONS

A large blue question mark icon is centered on a dark blue background.

76