TCSS 422: OPERATING SYSTEMS

The Process API & Limited Direct Execution

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

1

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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

2

TEXT BOOK COUPON

- 10% off textbook code: **TREAT10** (*through Friday Oct 15*)
- https://www.lulu.com/shop/remzi-arpaci-dusseau-and-andrea-arpaci-dusseau/operating-systems-three-easy-pieces-softcover-version-100/paperback/product-23779877.html?page=1&pageSize=4

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.3

3

OFFICE HOURS – FALL 2021

- **Tuesdays:**
  - 4:00 to 4:30 pm - CP 229
  - 7:00 to 7:30+ pm – ONLINE via Zoom
- **Thursdays**
  - 4:15 to 4:45 pm – ONLINE via Zoom
  - 7:00 to 7:30+ pm – ONLINE via Zoom
- Or email for appointment

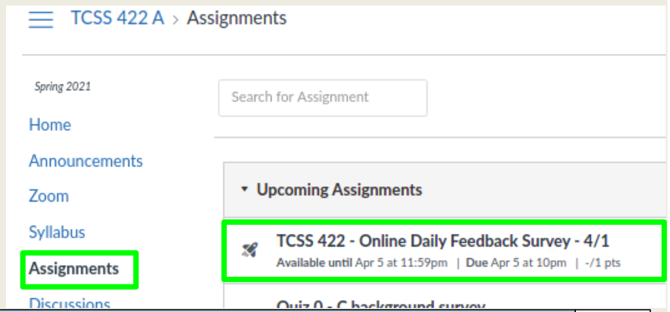
> Office Hours set based on Student Demographics survey feedback

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.4

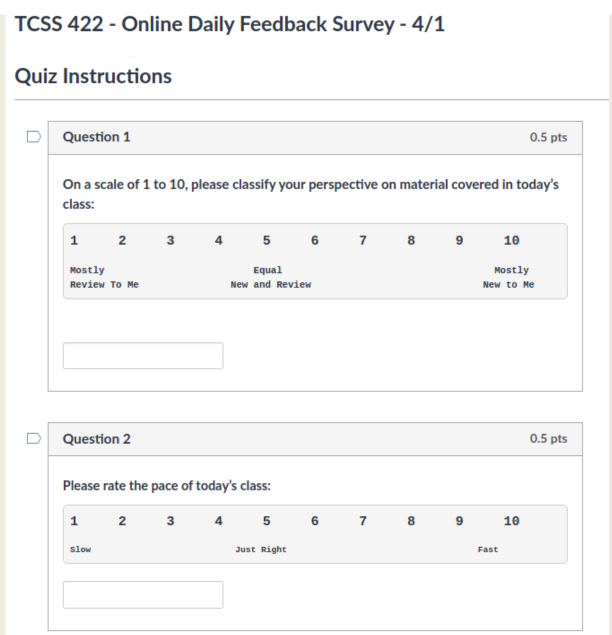
4

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



5



6

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (50 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.58 (↓ - previous 7.00)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.49 (↓ - previous 5.89)**

FEEDBACK

- ***What is the "kernel" and how is it closely related to the operating system?***
  - The kernel is the executable program that IS the operating system
  - On Ubuntu 20.04, the kernel is at:
    `ls -l /boot/vmlinuz-$(uname -r)`
  - Commands to display file information:
    ```
    file -v /boot/vmlinuz-$(uname -r)
    stat -v /boot/vmlinuz-$(uname -r)
    ```

FEEDBACK - 2

- ***What are fork(), wait(), and exec() used for and why are they beneficial?***
- **fork()** – API CALL to create a new process
  - Commonly used function to create new process in Linux C
  - Creates new process by cloning the parent and duplicating memory
  - Is still relatively efficient because **C**opy-**O**n-**W**rite (**COW**) is used to duplicate memory
  - Copy-on-write (COW) delays or altogether prevents duplication of process data. Initially, the parent and the child share a single copy. Data is marked if it is changed, and a duplicate is made, and each process receives a unique copy. Data duplication only occurs when data is changed, until then process data is shared read-only.
  - In this way the OS is lazy (again!)
  - ****What other approach mentioned in class involved the OS being lazy?***

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.9

9

FEEDBACK - 3

- ***What are fork(), wait(), and exec() used for and why are they beneficial?***
- **wait(), waitpid()** – API CALLS that wait for a child process to finish
- Two variants:
- **waitpid()** – provide the process ID to wait for
  ISSUE – if wrong ID is provided, can accidentally wait forever
- **wait()** – waits for the first child process to exit
  ISSUE – first process that exits may not be the desired one
  - Can check the ID and wait() again if not the right child

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.10

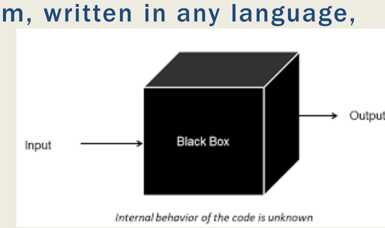
10

FEEDBACK - 4

- ***What are fork(), wait(), and exec() used for and why are they beneficial?***
- **execl, execlp, execle: transfer control of running process to external program (executable) using lists**
  - Variable number of arguments passed to function
- **Execv, execvp, execve: transfer control of running process to external program (executable) using arrays (vectors)**
  - Fixed number of arguments passed to function
- Once control is transferred to an external program, when the external program exits, the parent process can trap this, but control does not return to the original executable
- ***I may need to practice the different exec() calls. I'm still confused how to call them.***
- We will have a tutorial on using the exec routines

FEEDBACK - 5

- **How is the legacy program, when working with exec processes, like a "black box"?**
- When using exec, you're calling an executable program.
- With an executable, the source code may be unavailable, so it is not known exactly what the external program may do. (BE CAREFUL TO ONLY CALL TRUSTED EXECUTABLES)
- The executable can be any program, written in any language, even assembly language !
- Your C program acts only to generate the necessary inputs to invoke the external program
- C program acts like a "wrapper"



OBJECTIVES – 10/12

- Questions from 10/7
- **C Review Survey – Closes Thursday Oct 14**
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.13

13

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- **Assignment 0**
- Chapter 5: Process API
  - exec() with file redirection
- 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

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.14

14

FEEDBACK ON HOMEWORK 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**

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.15

15

TCSS 422 – SET VMS

- Request submitted for School of Engineering and Technology hosted Ubuntu 20.04 VMs for TCSS 422 – Fall 2021

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.16

16

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - **exec() with file redirection**
- 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

17

CHAPTER 5: C PROCESS API

18

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

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
        …
```

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
```


23

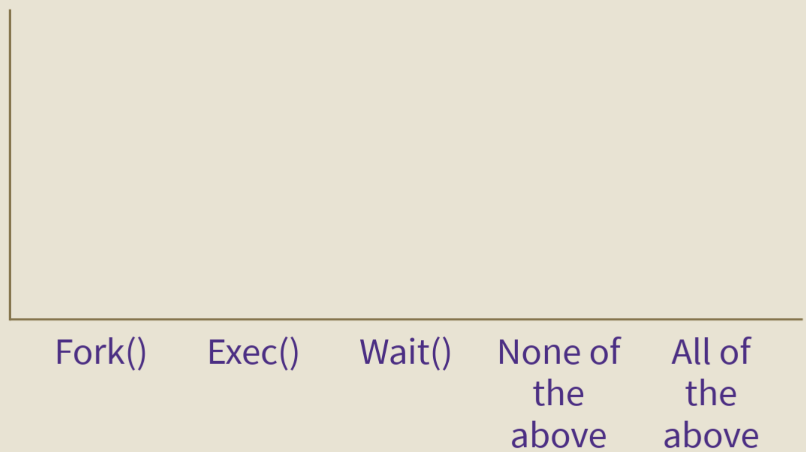
EXEC W/ FILE REDIRECTION (OUTPUT) - 2

```
        …
        // 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>
```


24

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



Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

25

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

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.26

26

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- **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

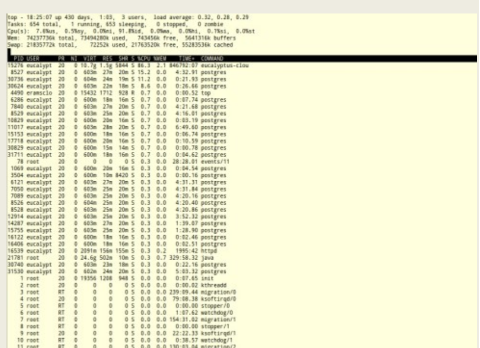
CH. 6: LIMITED DIRECT EXECUTION

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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

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	

31

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 *limits* on running programs, the OS wouldn't be in control of anything and would "<u>just be a library</u>"

OS	Program
`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	

32

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?

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.33

33

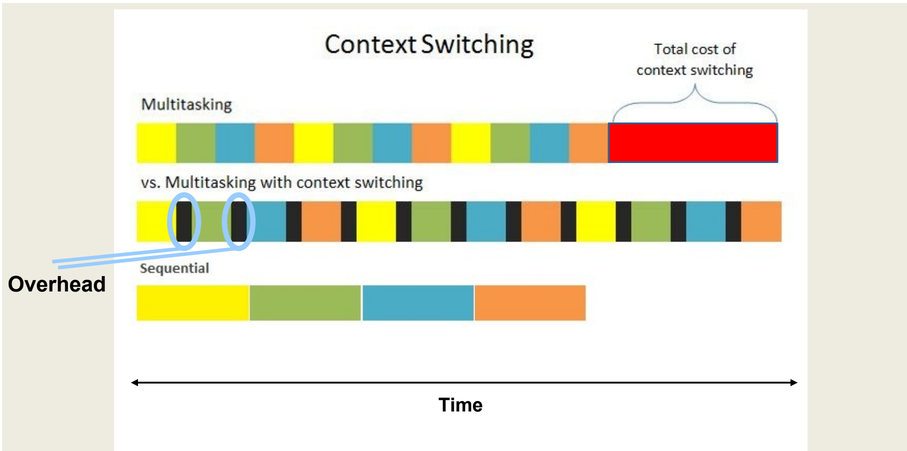
CONTROL TRADEOFF

- **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

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.34

34

CONTEXT SWITCHING OVERHEAD



October 12, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.35

35

WE WILL RETURN AT 2:40PM

October 12, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - coma | L4.36

36

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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 – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0

- Chapter 5: Process API
  - exec() with file redirection
- 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

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

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

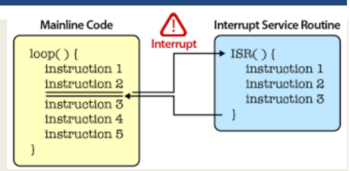
OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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

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

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

45

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

46

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

47

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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

48

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?

October 12, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L4.49

49

MULTITASKING

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

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

October 12, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L4.50

50

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

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results

51

QUESTION: MULTITASKING

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

October 12, 2021	TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma	L4.52

52

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?

53

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?

54

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?

55

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?

56

OBJECTIVES – 10/12

- Questions from 10/7
- C Review Survey – Closes Thursday Oct 14
- Assignment 0
- Chapter 5: Process API
  - exec() with file redirection
- 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:

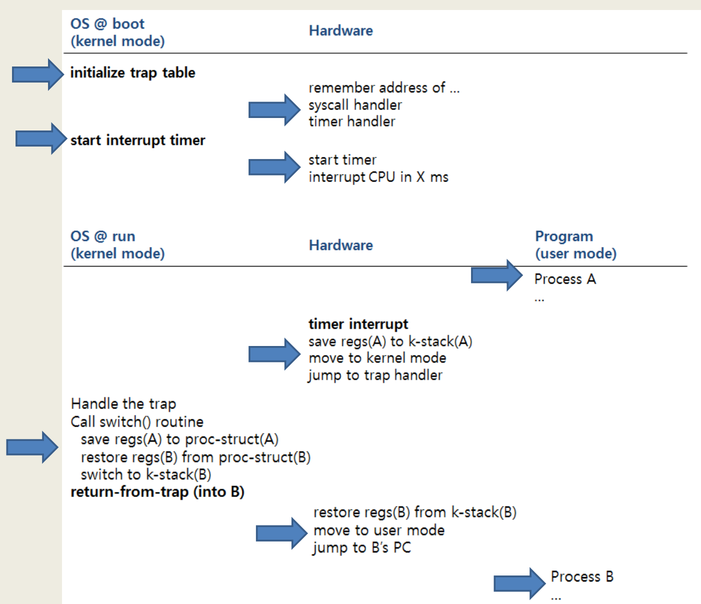
- Whether to continue running the **current process**, or switch to a **different one**.

- 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)
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 …

Context Switch

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

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

64

QUESTIONS

65