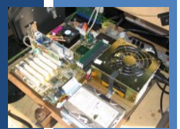
TCSS 422: OPERATING SYSTEMS

The Process API & Limited Direct Execution



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

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma

1

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.2

2

TEXT BOOK COUPON

- 15% off textbook code: **BCORP15!** (*through Friday Apr 7*)
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-softcover-version-100/paperback/product-14mjrrgk.html
- With coupon textbook is only $18.70 + tax & shipping

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.3

3

OFFICE HOURS – SPRING 2023

- **Tuesdays:**
  - 2:30 to 3:30 pm - CP 229 / Zoom
- **Fridays**
  - *1:30 to 2:30 pm – Zoom / (CP 229-on some days)
- Also available after class
- Or email for appointment

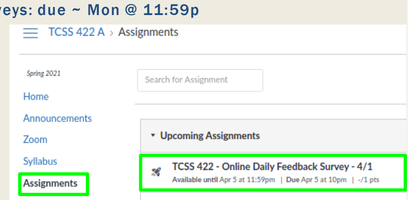
> Office Hours set based on Student Demographics survey feedback
** time may be occasionally rescheduled due to faculty meeting conflicts*

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.4

4

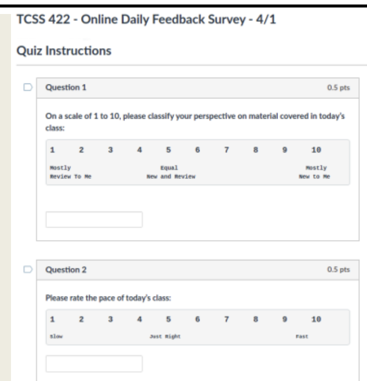
ONLINE DAILY FEEDBACK SURVEY

- 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 6, 2023 | TCSS422: Computer Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.5

5



April 6, 2023 | TCSS422: Computer Operating Systems [Spring 2023] 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 (45 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 7.20 (↑ - previous 6.77)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.42 (↓ - previous 5.71)**

7

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*

8

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

9

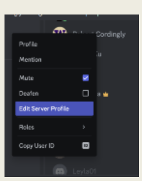
FEEDBACK - 3

- ***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

10

TCSS 422 DISCORD SERVER

- Please join the TCSS 422 A – Spring 2023 Discord Server
- **https://discord.gg/hqNanxEQ**
- Under Edit Server Profile:
  Please update your 'Server Nickname' to your real name or UW NET ID
  THANK YOU



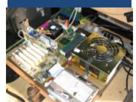
11

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

12

C REVIEW SURVEY - AVAILABLE THRU 4/7



April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.13

13

OBJECTIVES – 4/6

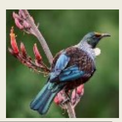
- Questions from 4/4
- C Review Survey – Closes Friday April 7
- **Assignment 0**
- Chapter 4: Linux process data structure - task_struct
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - Direct execution
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.14

14

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.15

15

FEEDBACK - 2

- ***My laptop is Apple M1 and the version of Ubuntu is 22.04.2 LTS. I was trying to look up the CPU model name from the VM, and it does not show up in my output. I'm wondering if it is due to M1, and is there any possible way for me to address the problem?***
- The ARM version of Ubuntu does not have the ability to identify the Model Name of M1/M2 Mac processors.
- You can likely find the CPU model from "About this Mac" from the MacOS.
- Additionally, you may be able to learn about the processor from the wikipedia pages:
- https://en.wikipedia.org/wiki/Apple_M1
- https://en.wikipedia.org/wiki/Apple_M2

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] 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 2023

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.17

17

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.18

18

LINUX: STRUCTURES

- `struct task_struct`, equivalent to **struct proc**
  - The Linux process data structure
  - Kernel data type (i.e. record) that describes individual Linux processes
  - Structure is VERY LARGE: **10,000+ bytes**
  - Defined in:
    /usr/src/linux-headers-{kernel version}/include/linux/sched.h
    - Ubuntu kernel version 5.15, **LOC: 723 - 1507**
    - Ubuntu kernel version 5.11, **LOC: 657 – 1394**
    - Ubuntu kernel version 4.4, **LOC: 1391 – 1852**

19

STRUCT TASK_STRUCT

- Key elements (e.g. PCB) in Linux are captured in struct task_struct: (LOC from Linux kernel v 5.11)
- Process ID
- `pid_t pid;` LOC #857
- Process State
- `/* -1 unrunnable, 0 runnable, >0 stopped: */`
- `volatile long state;` LOC #666
- Process time slice
  how long the process will run before context switching
- Struct sched_rt_entity used in task_struct contains timeslice:
  - `struct sched_rt_entity rt;` LOC #710
  - `unsigned int time_slice;` LOC #503

20

STRUCT TASK_STRUCT - 2

- Address space of the process:
- "mm" is short for "memory map"
- `struct mm_struct *mm;` LOC #779

- Parent process, that launched this one
- `struct task_struct __rcu *parent;` LOC #874

- Child processes (as a list)
- `struct list_head children;` LOC #879

- Open files
- `struct files_struct *files;` LOC #981

21

LINUX STRUCTURES - 2

- List of Linux data structures:
  http://www.tldp.org/LDP/tlk/ds/ds.html

- Description of process data structures:
  https://learning.oreilly.com/library/view/linux-kernel-development/9780768696974/cover.html
  3rd edition is online (dated from 2010):
  See chapter 3 on Process Management

  Safari online – accessible using UW ID SSO login
  Linux Kernel Development, 3rd edition
  Robert Love
  Addison-Wesley

22

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

23

CHAPTER 5: C PROCESS API

24

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

25

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;
}
```


26

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>
```

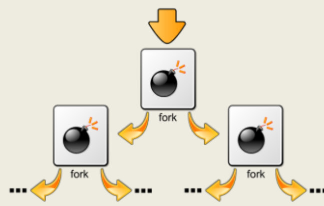
or

```
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

27

:(){ :|: & };:



28

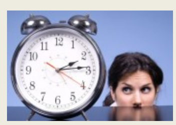
OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

29

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



30

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;
}
```


31

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>
```


32

FORK EXAMPLE

- Linux example

33

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

34

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

35

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
        …
```


36

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>
```


37

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);
        …
```


38

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
```

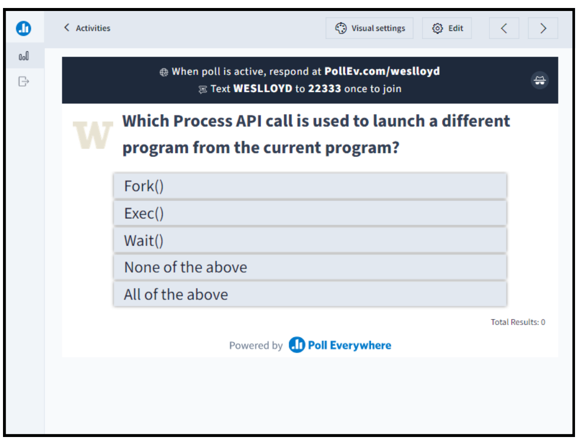

39

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>
```


40



41

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

42

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.43

43

CH. 6: LIMITED DIRECT EXECUTION

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.44

44

OBJECTIVES – 4/6

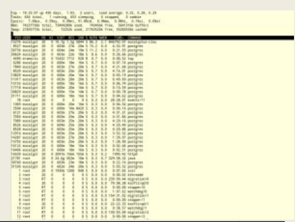
- Questions from 4/4
- C Review Survey – Closes Friday April 7
- Assignment 0
- Chapter 4: Linux process data structure - task_struct
- Chapter 5: Process API
  - fork(), wait(), exec()
- Chapter 6: Limited Direct Execution
  - **Direct execution**
  - Limited direct execution
  - CPU modes
  - System calls and traps
  - Cooperative multi-tasking
  - Context switching and preemptive multi-tasking

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.45

45

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.46

46

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	

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.47

47

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 "just be a library"

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.48

48

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.49

49

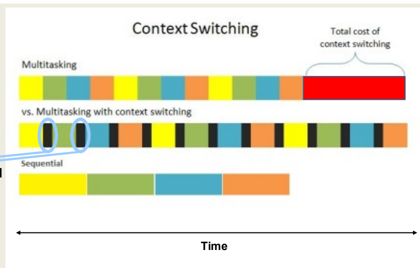
CONTROL TRADEOFF

- **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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.50

50

CONTEXT SWITCHING OVERHEAD

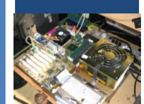


Overhead

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.51

51

WE WILL RETURN AT 2:40PM



April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.52

52

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.53

53

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.54

54

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

55

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

56

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

57

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

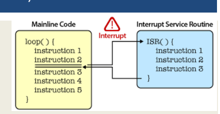
58

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

59

TRAPS: SYSTEM CALLS, EXCEPTIONS, INTERRUPTS



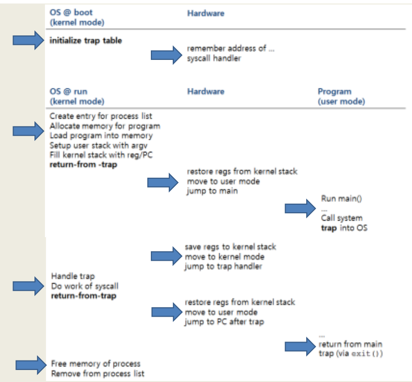
- Trap: any transfer to kernel mode
- Three kinds of traps
  - **System call:** (planned) user → kernel
    - SYSCALL for I/O, etc.
  - **Exception:** (error) user → kernel
    - Div by zero, page fault, page protection error
  - **Interrupt:** (event) user → kernel
    - Non-maskable vs. maskable
 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

60

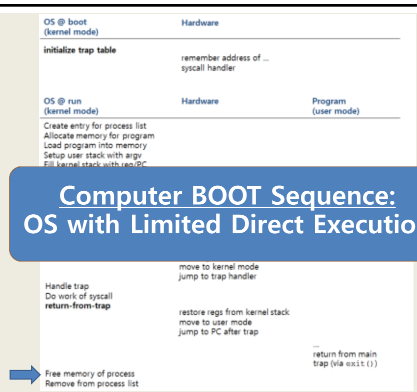
EXCEPTION TYPES

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

61



62



Computer BOOT Sequence: OS with Limited Direct Execution

63

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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

64

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?

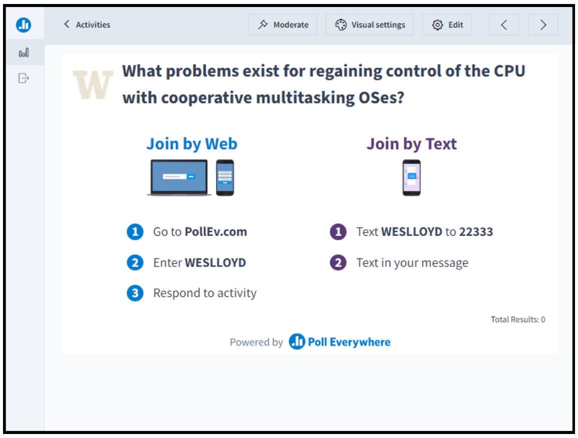
65

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**

66



67

QUESTION: MULTITASKING

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

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.68

68

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.69

69

MULTITASKING - 2

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

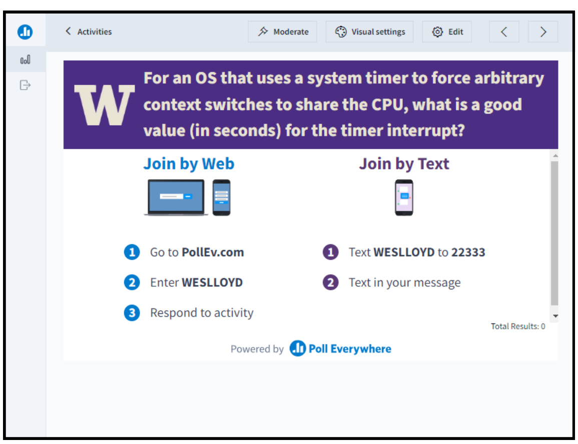
- Timer
  - Rais
  - Inter
    1. Current program is halted
    2. Program states are saved
    3. OS Interrupt handler is run (kernel mode)

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

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

April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.70

70



71

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.72

72

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.73

73

OBJECTIVES – 4/6

- Questions from 4/4
- C Review Survey – Closes Friday April 7
- 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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.74

74

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.75

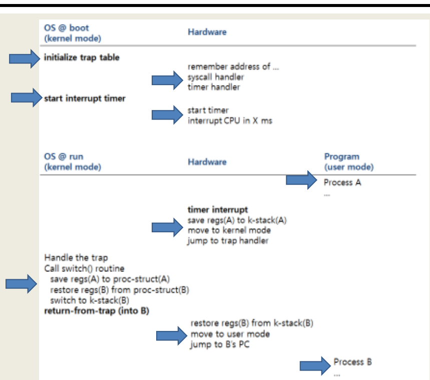
75

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

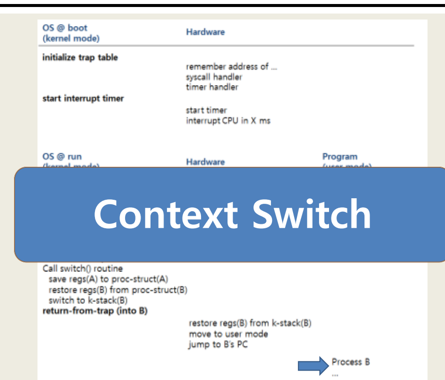
April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.76

76



April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.77

77



April 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.78

78

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.79

79

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 6, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L4.80

80

QUESTIONS

81