


TCSS 422: OPERATING SYSTEMS

INTRODUCTION TO
OPERATING SYSTEMS,
PROCESSES



Wes J. Lloyd

School of Engineering and Technology

University of Washington - Tacoma

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

Tacoma

1

OBJECTIVES – 4/7

■ Questions from 4/2

■ Chapter 4: Processes

- Kernel data structures for processes and threads

■ Chapter 5: Process API

- fork(), wait(), exec()

■ Assignment 0

■ 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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.2

2

MATERIAL / PACE

■ Please classify your perspective on material covered in today's class (60 respondents):

■ 1-mostly review, 5-equal new/review, 10-mostly new

■ Average – 7.03 (↑ from 5.83)

■ Please rate the pace of today's class:

■ 1-slow, 5-just right, 10-fast

■ Average – 5.76 (↑ from 5.17)

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.3

3

FEEDBACK FROM 4/2

■ On Chapter 2:

■ *"I feel like I got lost at points but I can't drill down exactly what the issue was or what I was missing to fully understand what you were showing us. I just recall you saying this was an Intro instead of a deep dive into the material"*

■ YES! Chapter 2 provides a broad survey of the "three easy pieces" all in one lecture. We will revisit everything in more detail as we go along...

■ I also introduced "context switches" ahead of time with the mem.c example. We'll go over this again today in Ch. 6

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.4

4

FEEDBACK - 2

■ Processes have their own address space (i.e. memory). threads do not. A process crashing won't affect another process.

■ i.e. Chrome tabs are processes, and a crashed tab does not crash the entire browser

■ **** Are there any other differences between threads and processes?**

■ → Linux threads have a smaller data structure than processes

■ → Threads are owned by processes

■ → Creating a new thread is faster than creating a new process

■ → Threads share memory, and can communicate through RAM

■ → Inter-process communication involves opening a file or network stream (pipe) between processes (covered in TCSS 430)

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.5

5

FEEDBACK - 3

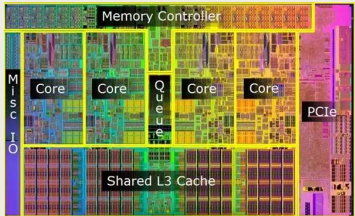
■ The overall process of concurrency is still a bit unclear to me.

■ Concurrency simply means doing multiple things at the same time on a computer.

■ Today's computer have multiple processing cores (i.e. CPU cores)

■ Each core can perform a different independent task "concurrently" at the same time

■ Check how many CPU cores in Ubuntu with "lscpu"



April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.6

6

FEEDBACK - 4

- Why Is It best for the CPU to perform a context switch when a thread is blocked?
- Which states is the process (job) idle?
- What are the consequences of suspending an idle process?
- a running process?

```
graph LR; Running((Running)) -- "I/O: initiate" --> Blocked((Blocked)); Blocked -- "I/O: done" --> Ready((Ready)); Ready -- "Scheduled" --> Running; Running -- "Deschedule" --> Ready;
```

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.7

7

FEEDBACK - 5

- Another student suggested that a single program that didn't have any concurrency would be able to run without an OS, however I am having trouble imagining a computer without an OS.
- In a sense, sort of, yes
- We talk about DIRECT EXECUTION today as part of Chapter 6, which is a bit like a computer without an OS.

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.8

8

MOTIVATION FOR LINUX

- It is worth noting the importance of Linux for today's developers and computer scientists.
- The CLOUD runs many virtual machines, recently in 2019 a key milestone was reached.
- Even on Microsoft Azure (the Microsoft Cloud), there were more Linux Virtual Machines (> 50%) than Windows.
- <https://www.zdnet.com/article/microsoft-developer-reveals-linux-is-now-more-used-on-azure-than-windows-server/>
- <https://www.zdnet.com/article/it-runs-on-the-cloud-and-the-cloud-runs-on-linux-any-questions/>
- The majority of application back-ends (server-side), cloud or not, run on Linux.
- This is due to licensing costs, example:

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.9

9

MOTIVATION FOR LINUX - 2

- Consider an example where you're asked to develop a web services backend that requires 10 x 8-CPU-core virtual servers
- Your organization investigates hosting costs on Amazon cloud
- 8-core VM is "c5d.2xlarge"

Name	Instance type	Memory	vCPUs	Linux On Demand cost	Windows On Demand cost
CS High-CPU Extra Large	c5d.xlarge	8.0 GiB	4 vCPUs	\$0.192000 hourly	\$0.376000 hourly
CS High-CPU 18xlarge	c5d.18xlarge	144.0 GiB	72 vCPUs	\$3.456000 hourly	\$6.768000 hourly
CS High-CPU Large	c5d.large	4.0 GiB	2 vCPUs	\$0.096000 hourly	\$0.188000 hourly
CS High-CPU 24xlarge	c5d.24xlarge	192.0 GiB	96 vCPUs	\$4.608000 hourly	\$9.024000 hourly
CS High-CPU Quadruple Extra Large	c5d.4xlarge	32.0 GiB	16 vCPUs	\$0.768000 hourly	\$1.504000 hourly
CS High-CPU Medium	c5d.medium	1.0 GiB	1 vCPU	\$0.024000 hourly	\$0.046000 hourly
CS High-CPU Double Extra Large	c5d.2xlarge	16.0 GiB	8 vCPUs	\$0.384000 hourly	\$0.752000 hourly
CS High-CPU 4xlarge	c5d.4xlarge	64.0 GiB	32 vCPUs	\$1.536000 hourly	\$3.008000 hourly
CS High-CPU 9xlarge	c5d.9xlarge	72.0 GiB	36 vCPUs	\$1.728000 hourly	\$3.384000 hourly

- Windows hourly price 75.2¢
- Linux hourly price 38.4¢
- See: <https://www.ec2instances.info/>

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.10

10

MOTIVATION FOR LINUX - 2

- One year cloud hosting cost:
- WINDOWS
- 10 VMs x 8,760 hours x \$.752 = \$65,875.20
- Linux
- 10 VMs x 8,760 hours x \$.384 = \$33,638.40
- Windows comes at a 95.8% price premium
- See: <https://www.ec2instances.info/>

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.11

11

"QUIZ" 0 - C PROGRAMMING BACKGROUND SURVEY

- Available via Canvas System
- Under: Assignments → Tutorials/Quizzes/In-class Activities
- Please disregard grade assigned by Canvas
- All submissions will receive 10 pts after assignment closes - (closes Thursday 4/9 @ 11:59p)

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.12

12

VIRTUAL MACHINE SURVEY

- Virtual Machine Survey
- Request for Ubuntu 18.04 VMs has been sent to the School of Engineering and Technology LABS
- Expect response soon regarding connection information
- Thank you!

April 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.13

13

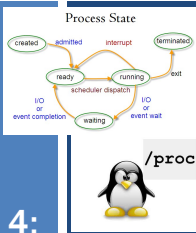
OBJECTIVES – 4/7

- Questions from 4/2
- Chapter 4: Processes**
 - Kernel data structures for processes and threads
- Chapter 5: Process API**
 - fork(), wait(), exec()
- Assignment 0**
- 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 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.14

14

CHAPTER 4: PROCESSES



April 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.15

15

PROCESS DATA STRUCTURES

- OS provides data structures to track process information
 - Process list
 - Process Data
 - State of process: Ready, Blocked, Running
 - Register context
- PCB (Process Control Block)
 - A C-structure that contains information about each process

April 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.16

16

XV6 KERNEL DATA STRUCTURES

- xv6: pedagogical implementation of Linux
- Simplified structures

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip; // Index pointer register
    int esp; // Stack pointer register
    int ebx; // Called the base register
    int ecx; // Called the counter register
    int edx; // Called the data register
    int esi; // Source index register
    int edi; // Destination index register
    int ebp; // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
    RUNNABLE, RUNNING, ZOMBIE };
```

April 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.17

17

XV6 KERNEL DATA STRUCTURES - 2

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem; // Start of process memory
    uint sz; // Size of process memory
    char *kstack; // Bottom of kernel stack
    // for this process
    enum proc_state state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    struct context context; // Switch here to run process
    struct trapframe *tf; // Trap frame for the current interrupt
};
```

April 7, 2020 TCS5422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L3.18

18

LINUX: STRUCTURES

- **struct task_struct**, equivalent to struct proc
 - Provides process description
 - Large: 10,000+ bytes
 - /usr/src/linux-headers-(kernel version)/include/linux/sched.h
 - ~ LOC 1391 – 1852 (4.4.0-170)
 - earlier was LOC 1227 – 1587
- **struct thread_info**, provides “context”
 - thread_info.h is at:
 - /usr/src/linux-headers-(kernel version)/arch/x86/include/asm/

April 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.19

19

LINUX: THREAD_INFO

```

struct thread_info {
    struct task_struct *task;          /* main task structure */
    struct exec_domain *exec_domain;  /* execution domain */
    u32 flags;                         /* low level flags */
    u32 status;                        /* thread synchronous flags */
    u32 cpu;                           /* current CPU */
    int preempt_count;                 /* 0 => preemptable,
                                         <0 => BUG */

    mm_segment_t addr_limit;
    struct restart_block restart_block;
    void __user *sysenter_return;

#ifdef CONFIG_X86_32
    unsigned long previous_esp;        /* ESP of the previous stack in
                                         case of nested (IRQ) stacks */
    u8 supervisor_stack[0];
#endif
    int uaccess_err;
};
    
```

April 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.20


20

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

April 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.21

21



CHAPTER 5: C PROCESS API

April 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.22

22

OBJECTIVES – 4/7


- Questions from 4/2
- Chapter 4: Processes
 - Kernel data structures for processes and threads
- Chapter 5: Process API
 - fork(), wait(), exec()
- Assignment 0
- 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 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.23

23

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 7, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L3.24

24

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.25

25

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

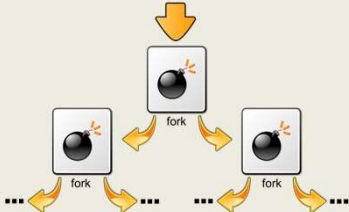
April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.26

26

:(){:|:&};:



April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.27

27

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.28

28

FORK WITH WAIT

■ p2.c

```
#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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.29

29

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.30

30

FORK EXAMPLE

- Linux example

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.31

31

exec()

- Supports running an external program
- 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

April 7, 2020

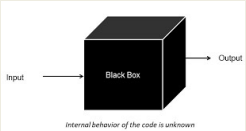
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.32

32

EXEC() - 2

- Common use case:
 - Write a new program which wraps a legacy one
 - Provide a new interface to an old system: Web services
 - Legacy program thought of as a "black box"
- We don't want to know what is inside... 😊



April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.33

33

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.34

34

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.35

35

EXEC WITH FILE REDIRECTION (OUTPUT)

```
#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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.36

36

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.37

37

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);          // parent goes down this path (main)
} else {
    int wc = wait(NULL);
}
return 0;
}
```

→

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

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.38

38

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

Fork()

Exec()

Wait()

None of the above

All of the above

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

Total Questions

39

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.40

40

TCSS 422 WILL RETURN AT 2:40PM



April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L1.41

41

OBJECTIVES – 4/7

■ Questions from 4/2

■ Chapter 4: Processes

- Kernel data structures for processes and threads

■ Chapter 5: Process API

- fork(), wait(), exec()

■ Assignment 0

■ 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

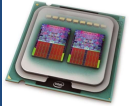
April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.42

42

CH. 6:
LIMITED DIRECT
EXECUTION



April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.43

43

OBJECTIVES – 4/7

- Questions from 4/2
- Chapter 4: Processes
 - Kernel data structures for processes and threads
- Chapter 5: Process API
 - fork(), wait(), exec()
- Assignment 0
- 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 7, 2020

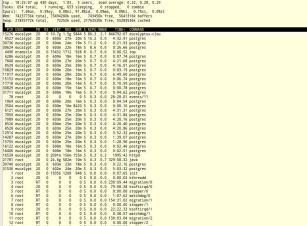
TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.44

44

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.45

45

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.46

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	
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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.47

47

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.48

48

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.49

49

CONTEXT SWITCHING OVERHEAD

Context Switching

Multitasking

vs. Multitasking with context switching

Sequential

Overhead

Total cost of context switching

Time

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.50

50

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.51

51

CPU MODES

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

access ← no access

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

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.52

52

CPU MODES

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

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.53

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.54

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

Interrupt Service Routine

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.55

55

EXCEPTION TYPES

Exception type	Synchronous vs. asynchronous	User request vs. occurred	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
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
Illegal undefined instruction	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunction	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.56

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

Handle trap

Do work of syscall

return-from-trap

save regs to kernel stack

move to kernel mode

jump to trap handler

restore regs from kernel stack

move to user mode

jump to PC after trap

Free memory of process

Remove from process list

return from main trap (via exit(1))

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.57

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

Computer BOOT Sequence:
OS with Limited Direct Execution

Handle trap

Do work of syscall

return-from-trap

move to kernel mode

jump to trap handler

restore regs from kernel stack

move to user mode

jump to PC after trap

Free memory of process

Remove from process list

return from main trap (via exit(1))

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.58

58

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?

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.59

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 for see with this approach?

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

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.60

60

Slides by Wes J. Lloyd

L3.10

W

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

61

QUESTION: MULTITASKING

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

April 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.62

62

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.63

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.64

64

W

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?

Start the presentation to see live content. Still no live content? Install the app or get help at [PollEV.com/app](#)

Total Results

65

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 7, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.66

66

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.67

67

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.68

68

The diagram illustrates the sequence of events during OS boot and execution. In the 'OS @ boot (kernel mode)' phase, the OS initializes a trap table (remembering the address of the syscall handler and timer handler) and starts an interrupt timer (interrupting the CPU in X ms). In the 'OS @ run (kernel mode)' phase, a timer interrupt occurs, saving registers (A) to the kernel stack (A) and moving to the kernel mode trap handler. The trap handler calls the switch() routine, saving registers (A) to the process structure (A), restoring registers (B) from the process structure (B), switching to the kernel stack (B), and returning from the trap into Process B. Finally, registers (B) are restored from the kernel stack (B), and the system moves to user mode, jumping to Process B's PC.

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.69

69

This diagram is identical to the one on slide 69, showing the OS boot and run phases and the context switch from Process A to Process B. A large blue box with the text 'Context Switch' is overlaid on the diagram.

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.70

70

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 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.71

71

PREEMPTIVE KERNEL

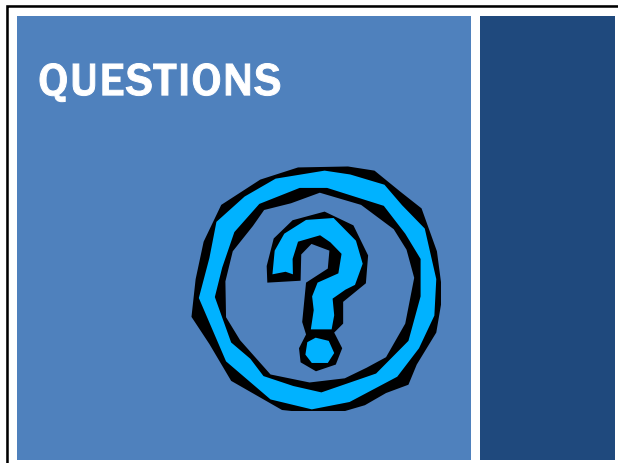
- Use “locks” as markers of regions of non-preemptibility (non-maskable interrupt)
- Preemption counter (`preempt_count`)
 - begins at zero
 - increments for each lock acquired (not safe to preempt)
 - decrements when locks are released
- Interrupt can be interrupted when `preempt_count=0`
 - It is safe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

April 7, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L3.72

72



73