TCSS 422: OPERATING SYSTEMS

**Three Easy Pieces
Process API,
Limited Direct Execution,
Scheduling Introduction**

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

April 2, 2018

TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Assignment 0 – Introduction to Linux
- Active Reading Quiz – Chapter 7
- Feedback from 3/28

- Processes – Ch. 4
- C Linux Process API – Ch. 5
- Limited Direct Execution – Ch. 6
  - Virtualizing the CPU
- Introduction to Scheduling – Ch. 7
- Multi-level Feedback Queue Scheduler – Ch. 8

April 2, 2018

TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L3.2

VIRTUAL MACHINE SURVEY

- **Please complete the Virtual Machine Survey is wanting an Institute of Technology hosted Ubuntu 16.04 VM**

- **https://goo.gl/forms/w9VWqkX756yXBUBt1**

- **Submitting results today…**

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.3

SELECTED FEEDBACK FROM 3/28

- **What is context switching?**

- **Most bash scripts I have seen begin with #!/bin/bash**
  - **You did not include this in your sample, yet it still worked.**
  - **Why did it work? and/or why is this usually included if it is not needed?**

- **What is fork used for? Such as in real-world applications?**

- **What is CPU virtualization?**

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.4

FEEDBACK - 2

- How do you schedule processes manualy?
  - Check out the "nice" command

  - Does this command, schedule processes?

  - Why? Why not?

- What's an example of a process state that's blocked?

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.5

CHAPTER 5: C PROCESS API

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.6

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 <u>current execution point</u>**
- **Book says "pretty odd"**
- **Creates a duplicate program instance (these are <u>processes!</u>)**
- **Copy of**
  - **Address space (memory)**
  - **Register**
  - **Program Counter (PC)**
- **Fork returns**
  - **child PID to parent**
  - **0 to child**

FORK EXAMPLE

- **p1.c**

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

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

FORK EXAMPLE - 2

■ **Non deterministic ordering of execution**

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

or

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

■ **CPU scheduler determines which to run first**

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.9

:(){ :|: & };:

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.10

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.11

FORK WITH WAIT

```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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.12

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.13

FORK EXAMPLE

- Linux example

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.14

exec()

- Supports running an external program
- 6 types: execl(), execlp(), execle(), execv(), execvp(), execvpe()

- execl(), execlp(), execle(): const char *arg

  List of pointers (terminated by null pointer)
  to strings provided as arguments… (arg0, arg1, .. argn)

- Execv(), execvp(), execvpe()
  Array of pointers to strings as arguments

  Strings are null-terminated
  First argument is name of file being executed

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.15

EXEC() - 2

- Common use case:
- Write a new program which wraps a legacy one
- Provide a new interface to an old system: Web services
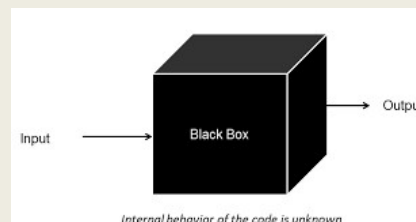- Legacy program thought of as a "black box"

- We don't want to know what is inside… 🙂

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.16

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)

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

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.19

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.20

EXEC W/ FILE REDIRECTION (OUTPUT) - 2

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

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.21

W 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 |

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

CH. 6:
LIMITED DIRECT EXECUTION

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 2, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L3.25 |

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 2, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L3.26 |

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 "**just be a library**"

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.27

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.28

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.29

CONTEXT SWITCHING OVERHEAD

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.30

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.31

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.32

CPU MODES

- **<u>User mode: ring 3 - untrusted</u>**
  - Some instructions and registers are disabled by the CPU
  - Exception registers
  - HALT instruction
  - MMU instructions
  - OS memory access
  - I/O device access

- **<u>Kernel mode: ring 0 – trusted</u>**
  - All instructions and registers enabled

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.33

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.34

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.35

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.36

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

April 2, 2018   TCSS422: Operating Systems [Spring 2018]
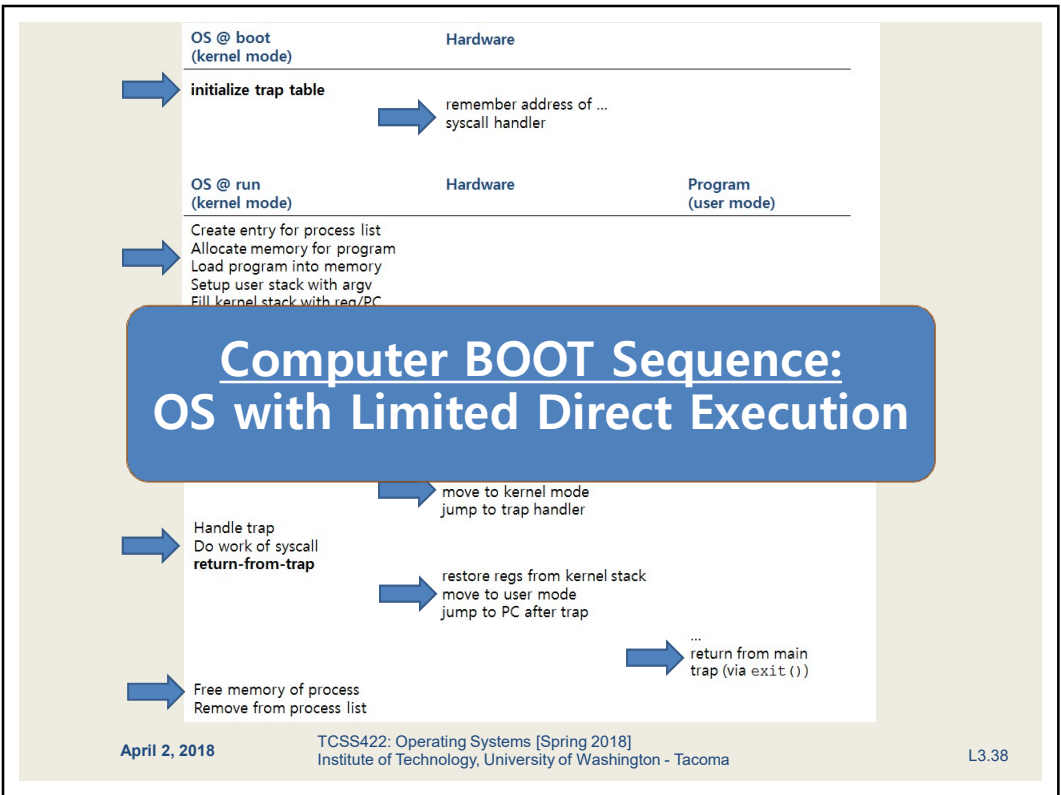   Institute of Technology, University of Washington - Tacoma   L3.37

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

Computer BOOT Sequence:
OS with Limited Direct Execution

                                 move to kernel mode
                                 jump to trap handler

Handle trap
Do work of syscall
return-from-trap
                                 restore regs from kernel stack
                                 move to user mode
                                 jump to PC after trap
                                                       ...
                                                       return from main
                                                       trap (via exit())

Free memory of process
Remove from process list

April 2, 2018   TCSS422: Operating Systems [Spring 2018]
                     Institute of Technology, University of Washington - Tacoma   L3.38
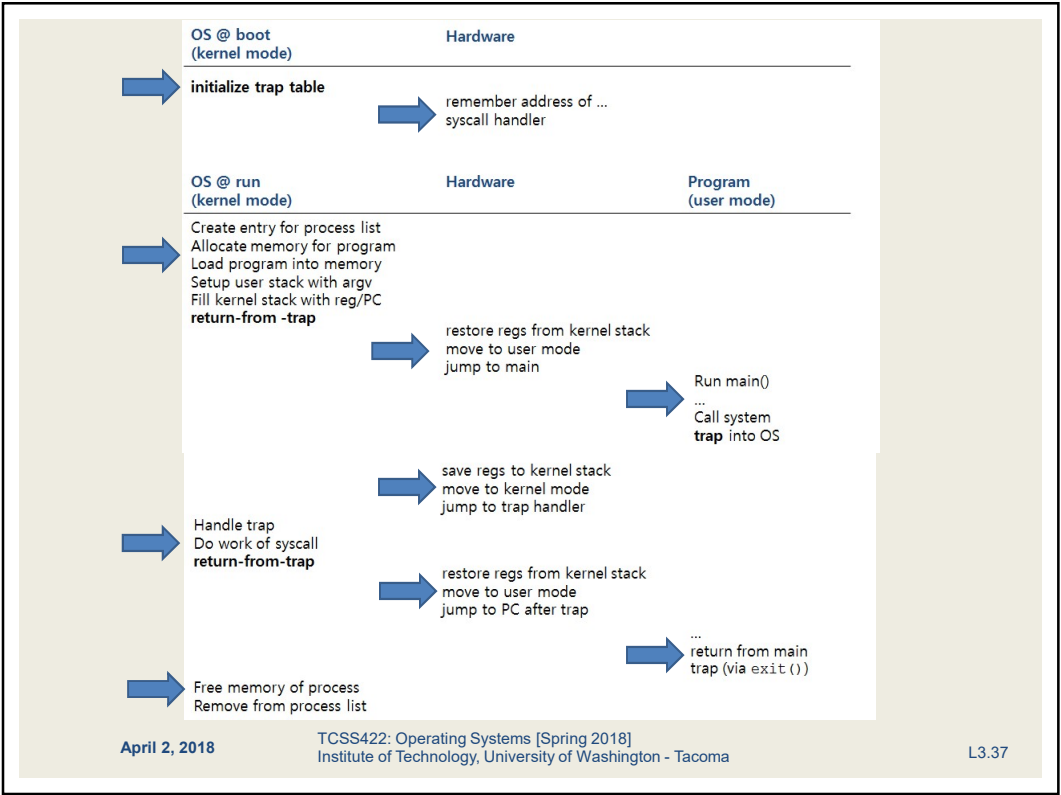
MULTITASKING

- 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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.39

MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?

- Cooperative multitasking (mostly pre 32-bit)
  - <
  - Op

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

    - When performing I/O
    - Illegal operations

  - (POLLEV)
    What problems could you for see with this approach?

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.40

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

QUESTION: MULTITASKING

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

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.42

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.43

MULTITASKING - 2

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

- Timer

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

  - Rais
  - Inter
    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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.44

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

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?

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 <u>context switch</u> swapping out the current process for a new one.

April 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.47

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.48

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   Program
(kernel mode)   Hardware   (user mode)

                                                                Process A
                                                                ...

                                       timer interrupt
                                       save regs(A) to k-stack(A)
                                       move to kernel mode
                                       jump to trap handler

Handle the trap
Call switch() routine
   save regs(A) to proc-struct(A)
   restore regs(B) from proc-struct(B)
   switch to k-stack(B)
return-from-trap (into B)

                                       restore regs(B) from k-stack(B)
                                       move to user mode
                                       jump to B's PC

                                                                Process B
                                                                ...

April 2, 2018   TCSS422: Operating Systems [Spring 2018]
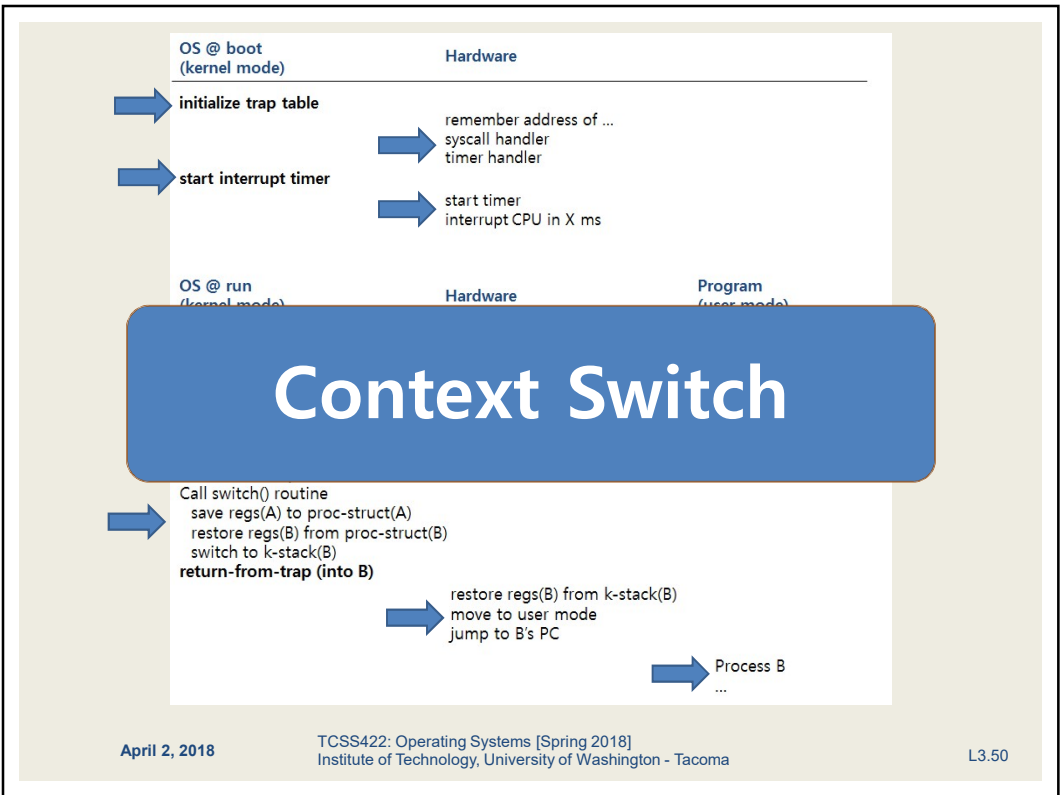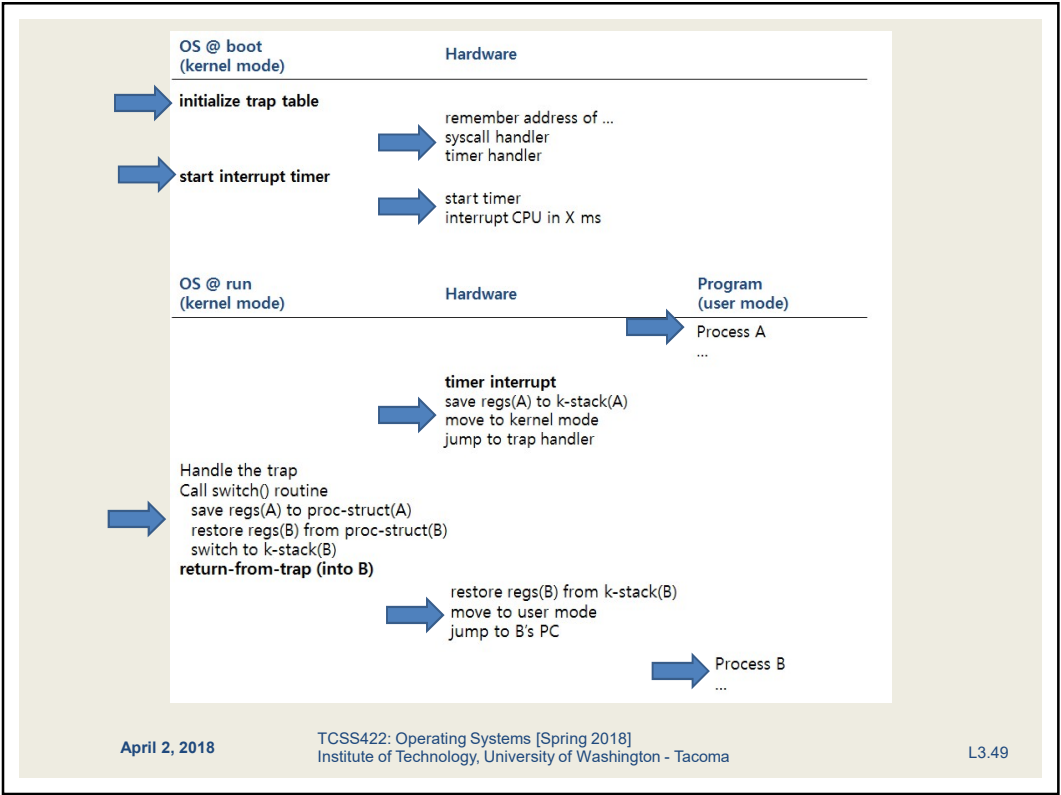   Institute of Technology, University of Washington - Tacoma   L3.49

OS @ boot
(kernel mode)   Hardware

initialize trap table

                                       remember address of ...
                                       syscall handler
                                       timer handler

start interrupt timer

                                       start timer
                                       interrupt CPU in X ms

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

Context Switch

Call switch() routine
   save regs(A) to proc-struct(A)
   restore regs(B) from proc-struct(B)
   switch to k-stack(B)
return-from-trap (into B)

                                       restore regs(B) from k-stack(B)
                                       move to user mode
                                       jump to B's PC

                                                                Process B
                                                                ...

April 2, 2018   TCSS422: Operating Systems [Spring 2018]
                       Institute of Technology, University of Washington - Tacoma   L3.50

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.51

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 2, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L3.52

QUESTIONS