TCSS 422: OPERATING SYSTEMS

Processes & The Process API

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

1

OBJECTIVES – 1/15

- **Questions from 1/13**
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

2

VIRTUAL MACHINE SUPPORT ON APPLE M1

- Installing a Ubuntu Virtual Machine on Apple M1 MacBooks:
- FREE
- https://mac.getutm.app/

- MACs use Apple Silicon ARM-based CPUs
  - Motivation: faster, less expensive than Intel-based CPUs

3

TEXT BOOK COUPON

- 15% off textbook code: AAC72SAVE15

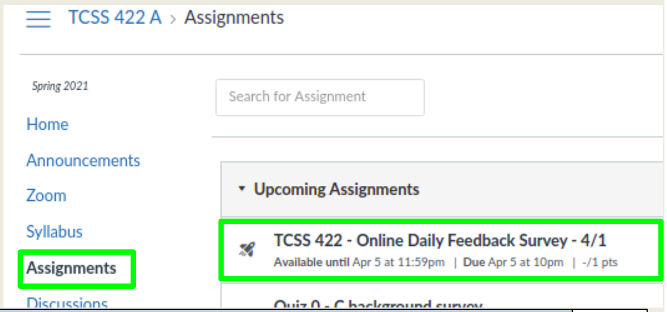
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-hardcover-version-110/hardcover/product-15gjeeky.html?q=three+easy+pieces+operating+systems&page=1&pageSize=4

- With coupon textbook is only ~ $33.79 + tax & shipping

4

FEEDBACK SURVEYS

- Feedback Survey in Class and on Canvas
- **All Quarter: 1-point Extra Credit** for completing online
- **Weeks 1-6: 2-points Extra Credit** completing in class
- **Weeks 7-9: 3-points Extra Credit**, 4-points (week 10)
- 46 points possible
- 2.5% added to final course grade for (46/46)
  - There will be other opportunities (seminars, etc.) to earn survey pts


TCSS 422 A › Assignments

Spring 2021
Home
Announcements
Zoom
Syllabus
Assignments
Discussions

Search for Assignment

Upcoming Assignments
TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts
Quiz 0 - C background survey

January 15, 2026	TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.5

5

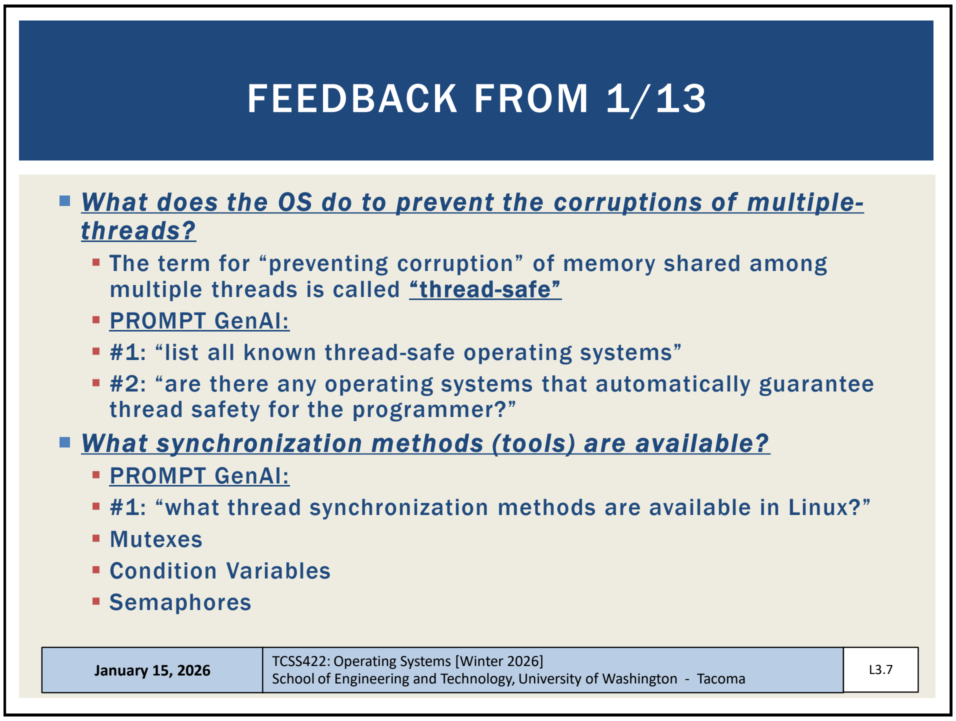
MATERIAL / PACE

- Please classify your perspective on material covered in today's class:
  - 45 of 46 respondents – 97.83%!!
  - 36 in-person, 9 online
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.34 (↑ - previous 5.83)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.13 (↓ - previous 5.21)**

January 15, 2026	TCSS422: Computer Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.6

6

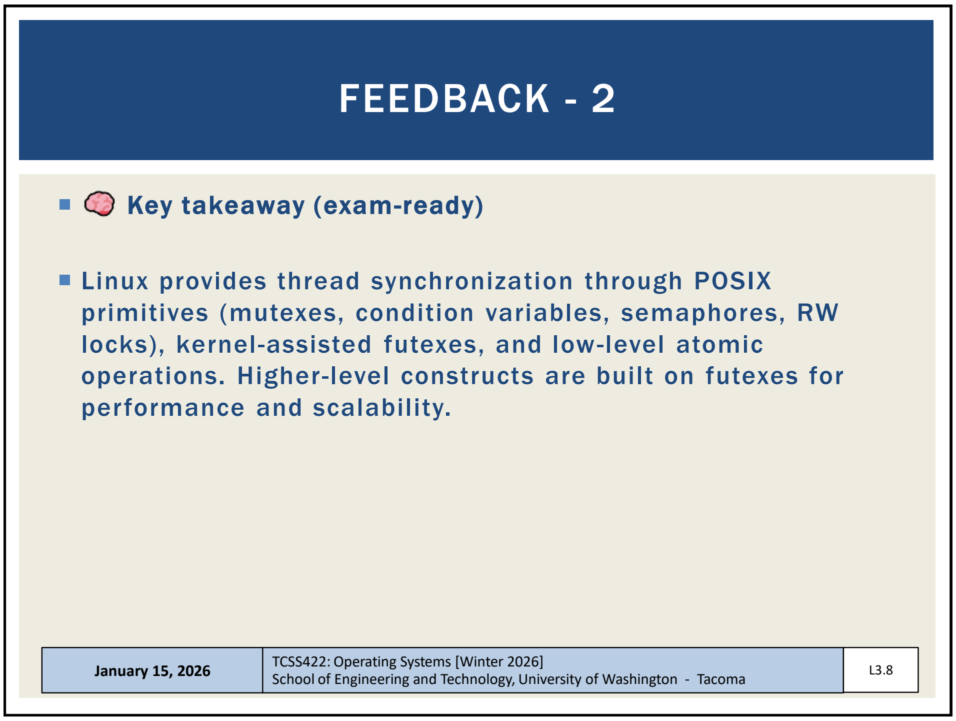


FEEDBACK FROM 1/13

- ***What does the OS do to prevent the corruptions of multiple-threads?***
  - The term for "preventing corruption" of memory shared among multiple threads is called **"thread-safe"**
  - PROMPT GenAI:
  - #1: "list all known thread-safe operating systems"
  - #2: "are there any operating systems that automatically guarantee thread safety for the programmer?"
- ***What synchronization methods (tools) are available?***
  - PROMPT GenAI:
  - #1: "what thread synchronization methods are available in Linux?"
  - Mutexes
  - Condition Variables
  - Semaphores

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.7

7



FEEDBACK - 2

- 🧠 Key takeaway (exam-ready)

- Linux provides thread synchronization through POSIX primitives (mutexes, condition variables, semaphores, RW locks), kernel-assisted futexes, and low-level atomic operations. Higher-level constructs are built on futexes for performance and scalability.

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.8

8

FEEDBACK - 3

- ***If two programs see some virtual addresses, how does the OS ensure that it does touch its physical memory?***
- Interpretation: 'how does the OS enable you to use variables stored in physical memory?'
- You can print the address of anything with %p and '&' the address of operator:

```
int x = 1;
printf("x = %d addr=%p\n",x,&x);
```

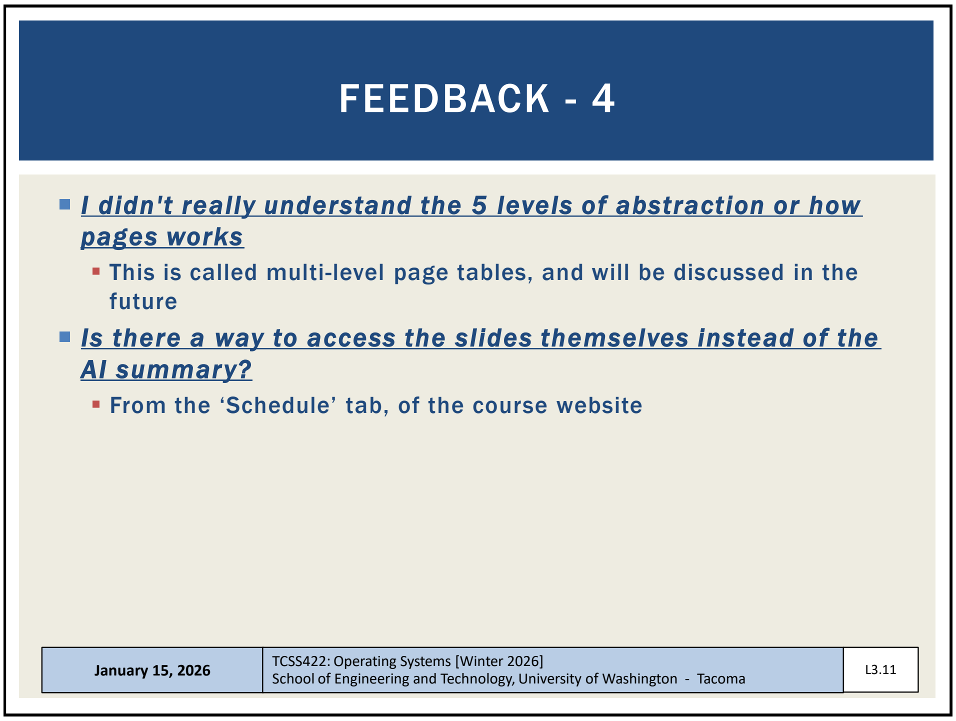
- When you modify or print 'int x', the OS automatically translates the virtual addr to the physical addr behind the scenes to support working with the variable
- int x virtual addr can be 0x7ffffffdf54
- x is a local variables stored on the program's stack
- notice the stack is near the end of the address range

VIRTUAL ADDRESS SPACE 64-BIT LINUX OS

- 48-bit Virtual Address Space (Standard)
- This is the most common configuration, providing a total usable space of 256 TB.

Region	Start Address	End Address	Size
User Space	0x0000000000000000	0x00007FFFFFFFFFFF	128 TB
Unused Gap	0x0000800000000000	0xFFFF7FFFFFFFFFFF	~16 EB
Kernel Space	0xFFFF800000000000	0xFFFFFFFFFFFFFFFF	128 TB

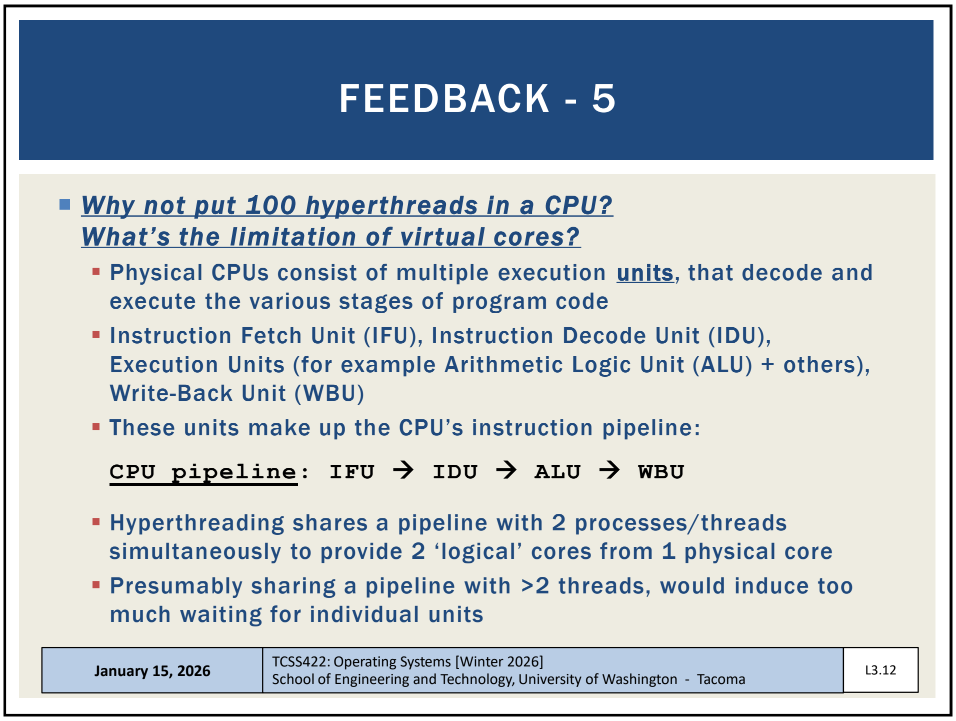
- A true 64-bit virtual address space can address 16,384,000 tera-bytes, which is 16,384 peta-bytes, which is 16.384 exa-bytes
- This much is not needed, so only 48-bits (3/4) of the address space is typically used
  - Larger servers may use a 57-bit address space (128 PB)



FEEDBACK - 4

- ***I didn't really understand the 5 levels of abstraction or how pages works***
  - This is called multi-level page tables, and will be discussed in the future
- ***Is there a way to access the slides themselves instead of the AI summary?***
  - From the 'Schedule' tab, of the course website

11



FEEDBACK - 5

- ***Why not put 100 hyperthreads in a CPU?***
  What's the limitation of virtual cores?
  - Physical CPUs consist of multiple execution **units**, that decode and execute the various stages of program code
  - Instruction Fetch Unit (IFU), Instruction Decode Unit (IDU), Execution Units (for example Arithmetic Logic Unit (ALU) + others), Write-Back Unit (WBU)
  - These units make up the CPU's instruction pipeline:

```
CPU pipeline: IFU → IDU → ALU → WBU
```

  - Hyperthreading shares a pipeline with 2 processes/threads simultaneously to provide 2 'logical' cores from 1 physical core
  - Presumably sharing a pipeline with >2 threads, would induce too much waiting for individual units

12

MOTIVATION FOR LINUX

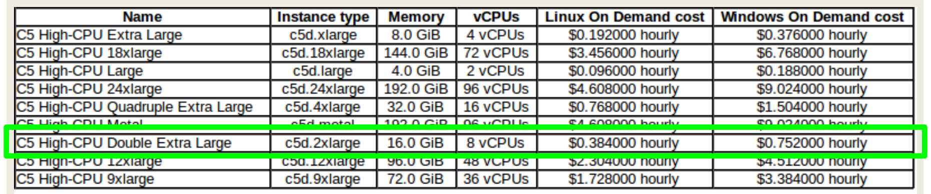
- It is worth noting the importance of Linux for today's developers and computer scientists.
- The CLOUD runs many virtual machines, recently in 2019 a key milestone was reached.
- Even on Microsoft Azure (the Microsoft Cloud), there were more Linux Virtual Machines (> 50%) than Windows.
- https://www.zdnet.com/article/microsoft-developer-reveals-linux-is-now-more-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:

13

MOTIVATION FOR LINUX - 2

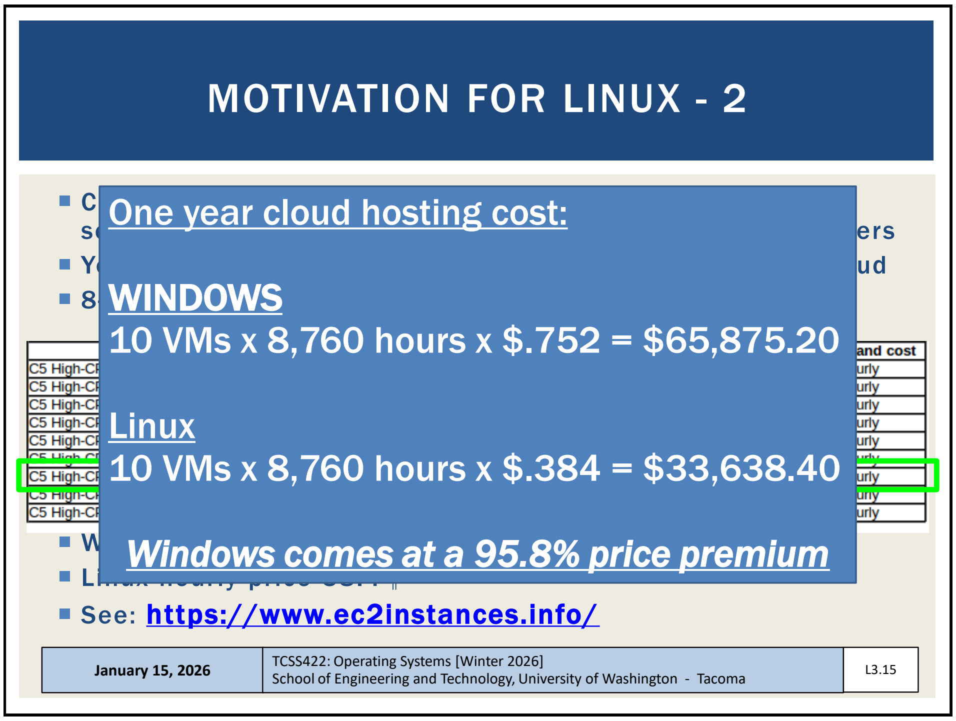
- Consider a pricing example where you're asked to develop a web services backend that requires 10 x 8-CPU-core virtual servers
- Your organization investigates hosting costs on Amazon cloud
- 8-core VM is "c5d.2xlarge"



Name	Instance type	Memory	vCPUs	Linux On Demand cost	Windows On Demand cost
C5 High-CPU Extra Large	c5d.xlarge	8.0 GiB	4 vCPUs	$0.192000 hourly	$0.376000 hourly
C5 High-CPU 18xlarge	c5d.18xlarge	144.0 GiB	72 vCPUs	$3.456000 hourly	$6.768000 hourly
C5 High-CPU Large	c5d.large	4.0 GiB	2 vCPUs	$0.096000 hourly	$0.188000 hourly
C5 High-CPU 24xlarge	c5d.24xlarge	192.0 GiB	96 vCPUs	$4.608000 hourly	$9.024000 hourly
C5 High-CPU Quadruple Extra Large	c5d.4xlarge	32.0 GiB	16 vCPUs	$0.768000 hourly	$1.504000 hourly
C5 High-CPU Metal	c5d.metal	192.0 GiB	96 vCPUs	$4.608000 hourly	$9.024000 hourly
C5 High-CPU Double Extra Large	c5d.2xlarge	16.0 GiB	8 vCPUs	$0.384000 hourly	$0.752000 hourly
C5 High-CPU 12xlarge	c5d.12xlarge	96.0 GiB	48 vCPUs	$2.304000 hourly	$4.512000 hourly
C5 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://instances.vantage.sh/

14



MOTIVATION FOR LINUX - 2

- C
  s ers
- Y ud
- 8

and cost
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly
C5 High-CP urly

One year cloud hosting cost:

WINDOWS
10 VMs x 8,760 hours x $.752 = $65,875.20

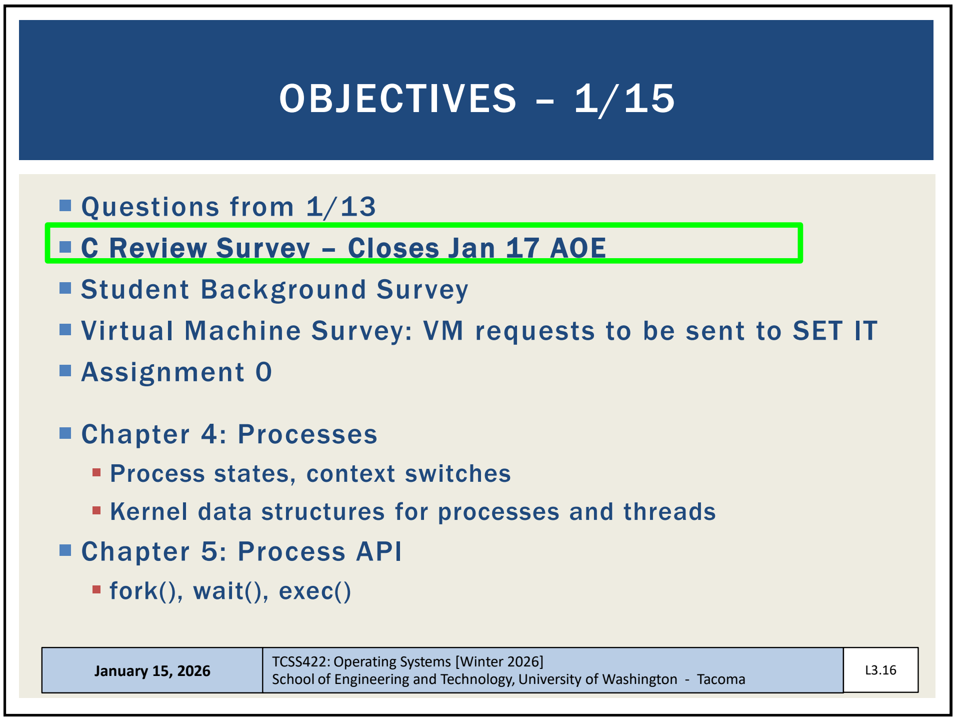
Linux
10 VMs x 8,760 hours x $.384 = $33,638.40

Windows comes at a 95.8% price premium

- W
- Linux hourly price cost
- See: **https://www.ec2instances.info/**

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.15

15



OBJECTIVES – 1/15

- Questions from 1/13
- **C Review Survey – Closes Jan 17 AOE**
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0

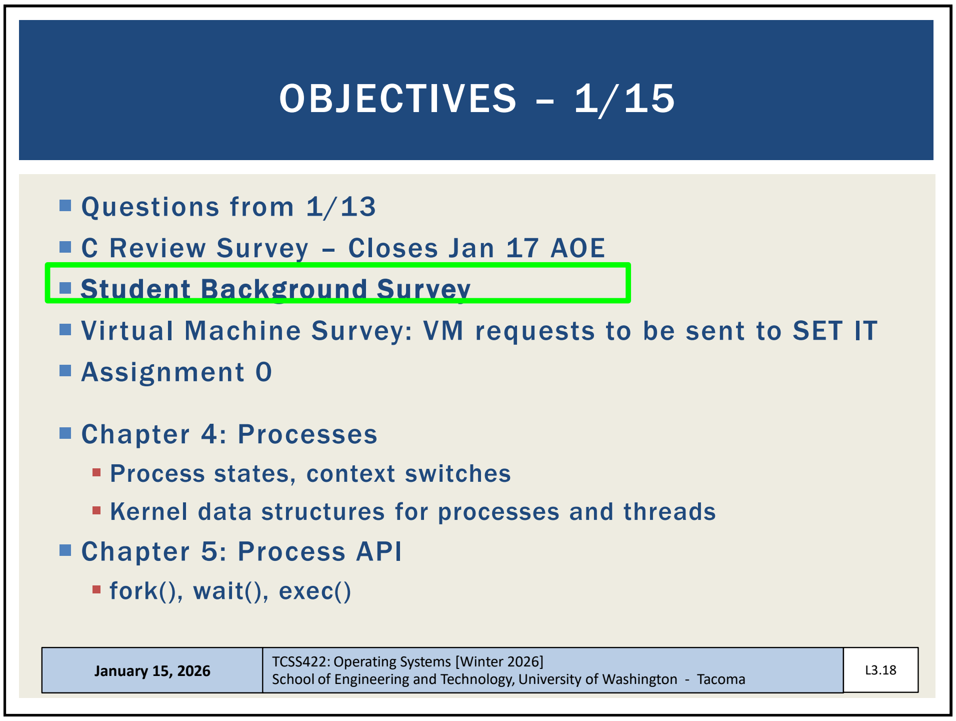
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.16

16

C REVIEW SURVEY - AVAILABLE THRU 1/17

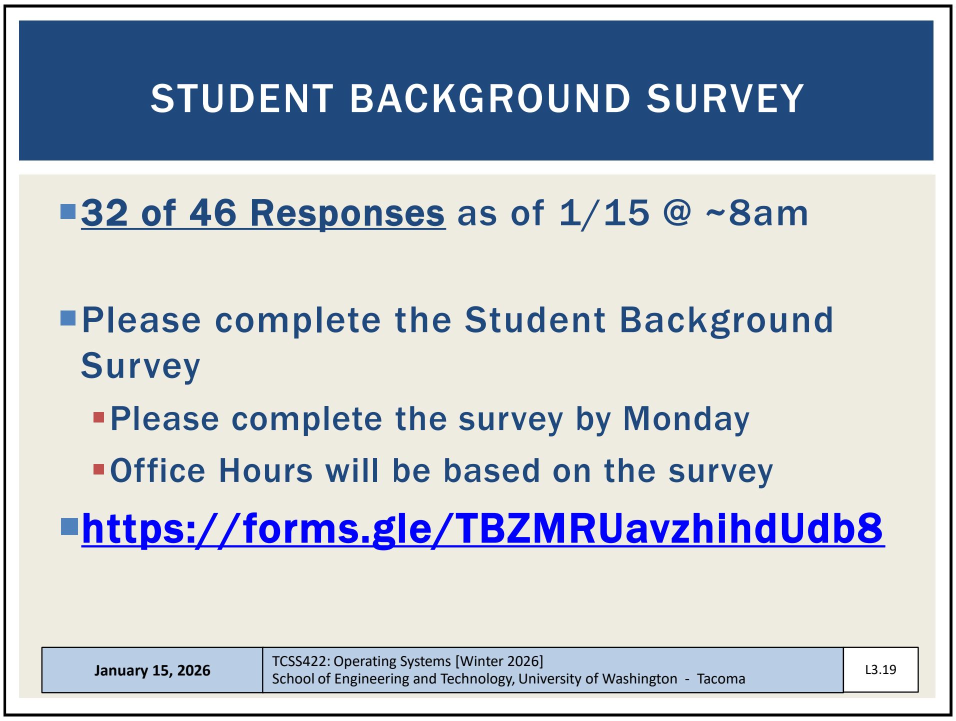
17



OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- **Student Background Survey**
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

18

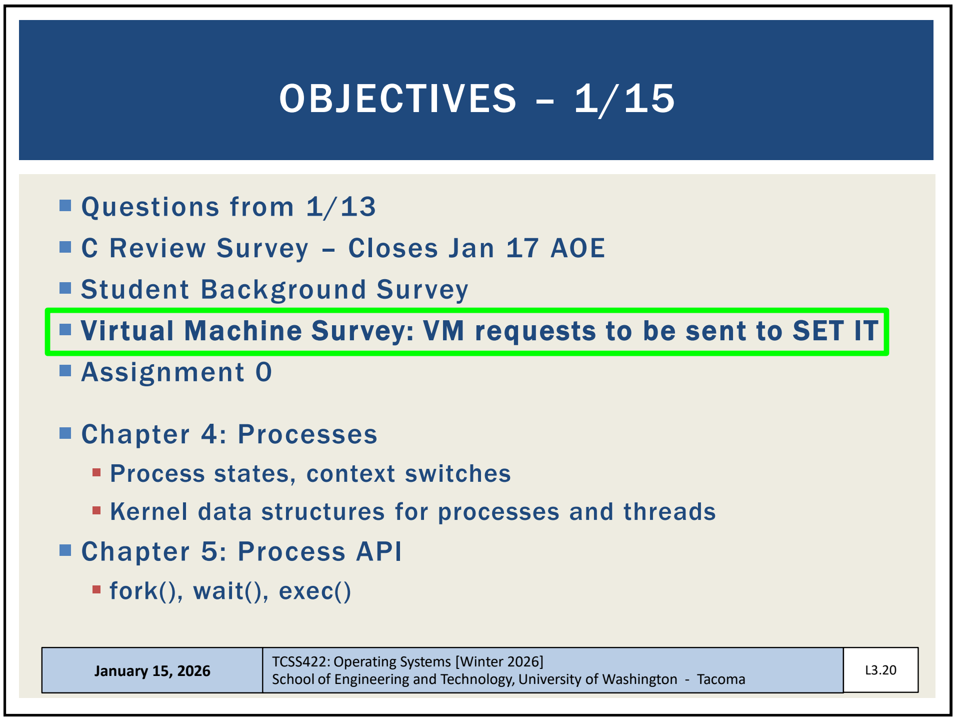


STUDENT BACKGROUND SURVEY

- **32 of 46 Responses** as of 1/15 @ ~8am
- Please complete the Student Background Survey
  - Please complete the survey by Monday
  - Office Hours will be based on the survey
- https://forms.gle/TBZMRUavzhihdUdb8

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.19

19

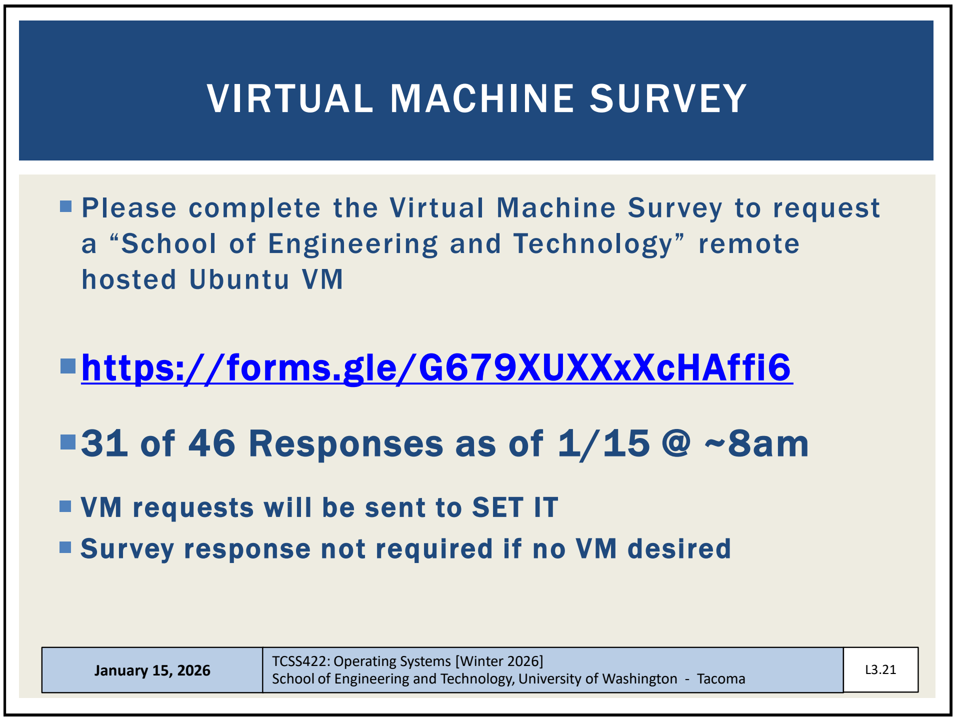


OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- **Virtual Machine Survey: VM requests to be sent to SET IT**
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.20

20

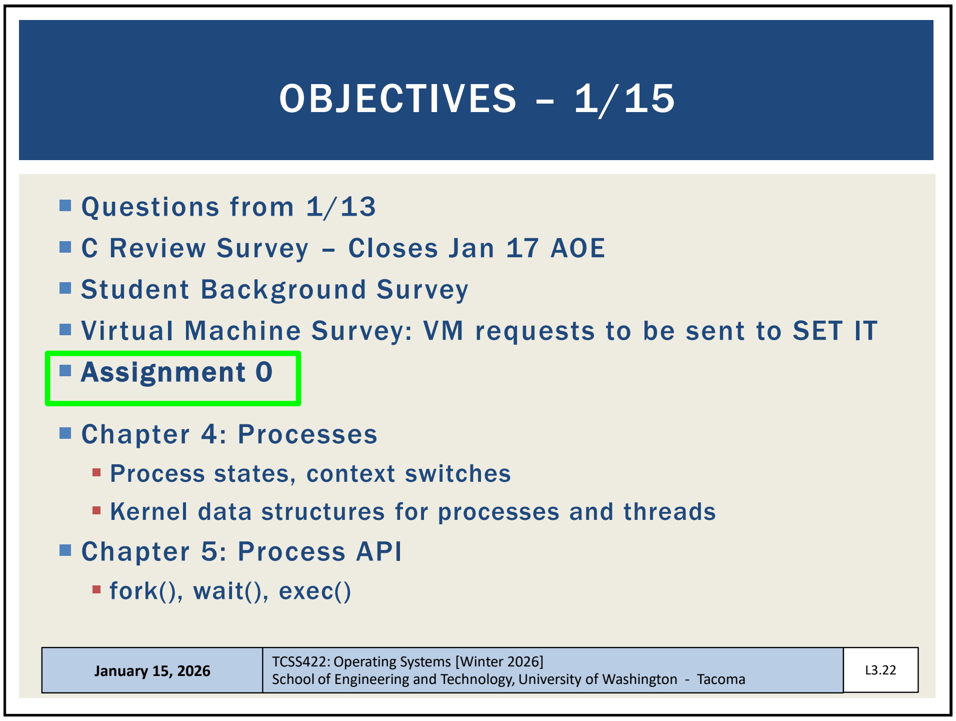


VIRTUAL MACHINE SURVEY

- Please complete the Virtual Machine Survey to request a "School of Engineering and Technology" remote hosted Ubuntu VM
- **https://forms.gle/G679XUXXxXcHAffi6**
- **31 of 46 Responses as of 1/15 @ ~8am**
- **VM requests will be sent to SET IT**
- **Survey response not required if no VM desired**

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.21

21



OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- **Assignment 0**
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.22

22

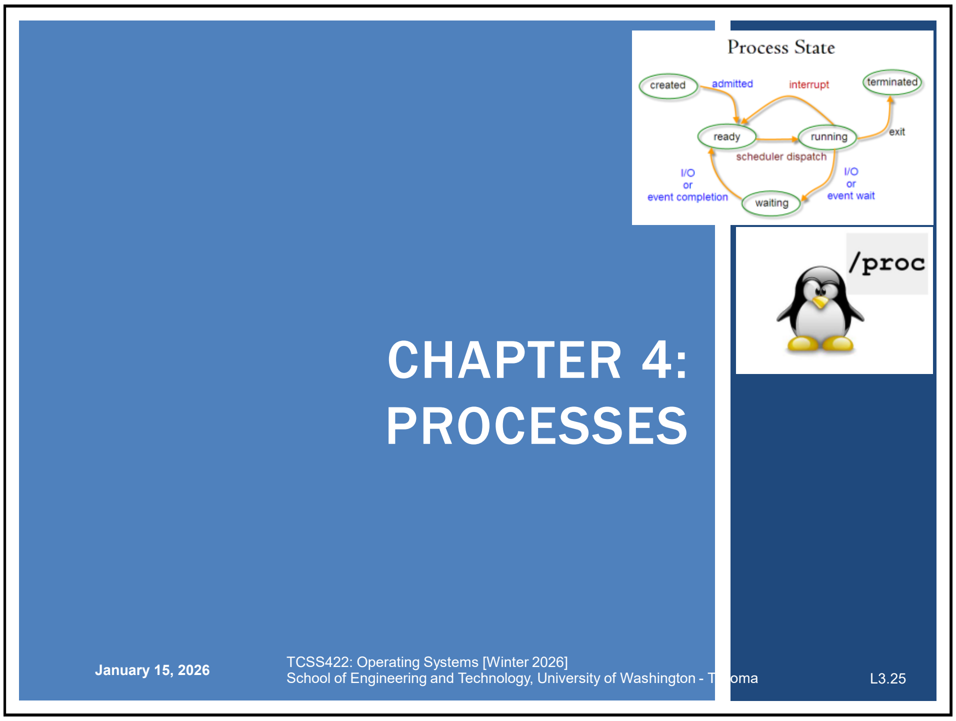
WE WILL RETURN AT 5:00PM

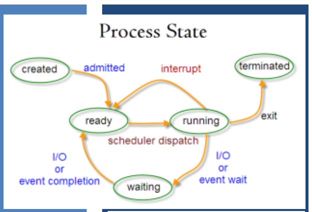
23

OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- **Chapter 4: Processes**
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

24

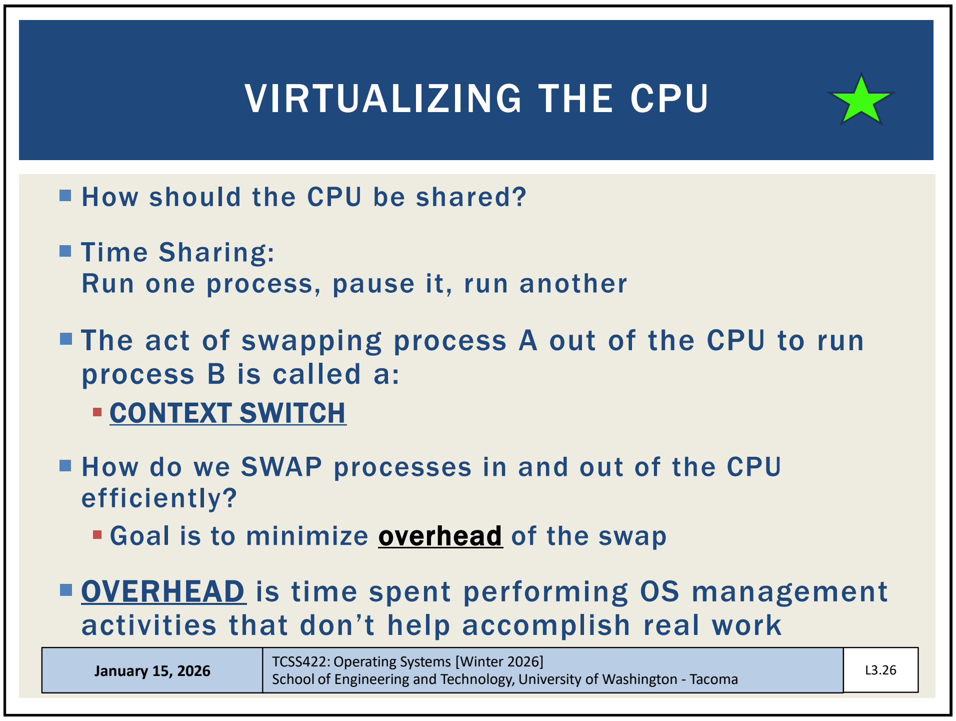




CHAPTER 4: PROCESSES

January 15, 2026 TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma L3.25

25

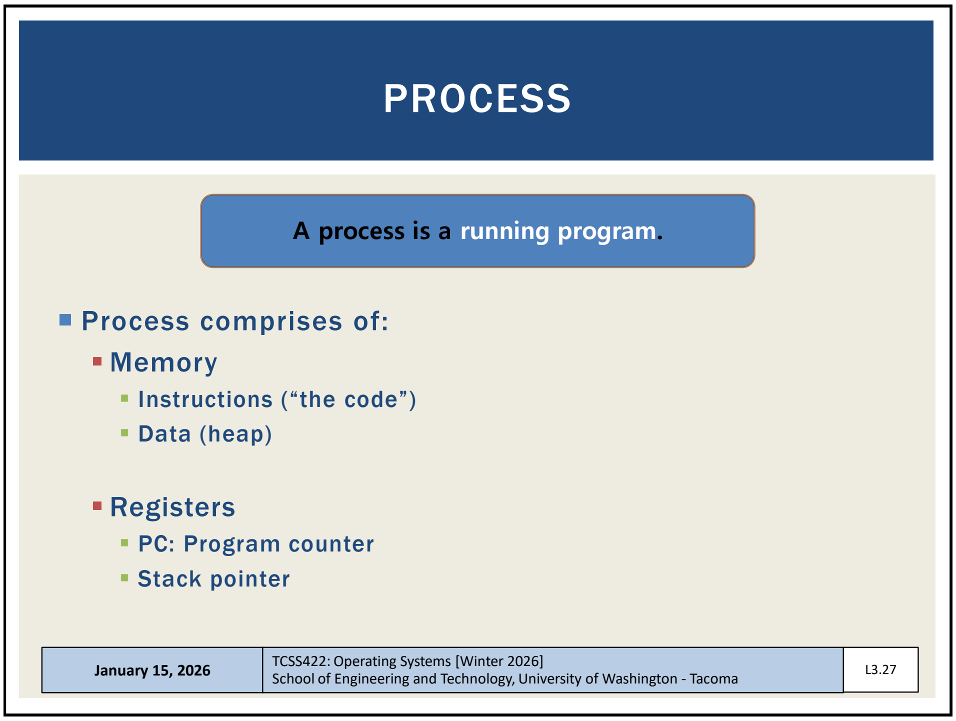


VIRTUALIZING THE CPU

- How should the CPU be shared?
- Time Sharing:
  Run one process, pause it, run another
- The act of swapping process A out of the CPU to run process B is called a:
  - **CONTEXT SWITCH**
- How do we SWAP processes in and out of the CPU efficiently?
  - Goal is to minimize **overhead** of the swap
- **OVERHEAD** is time spent performing OS management activities that don't help accomplish real work

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.26

26



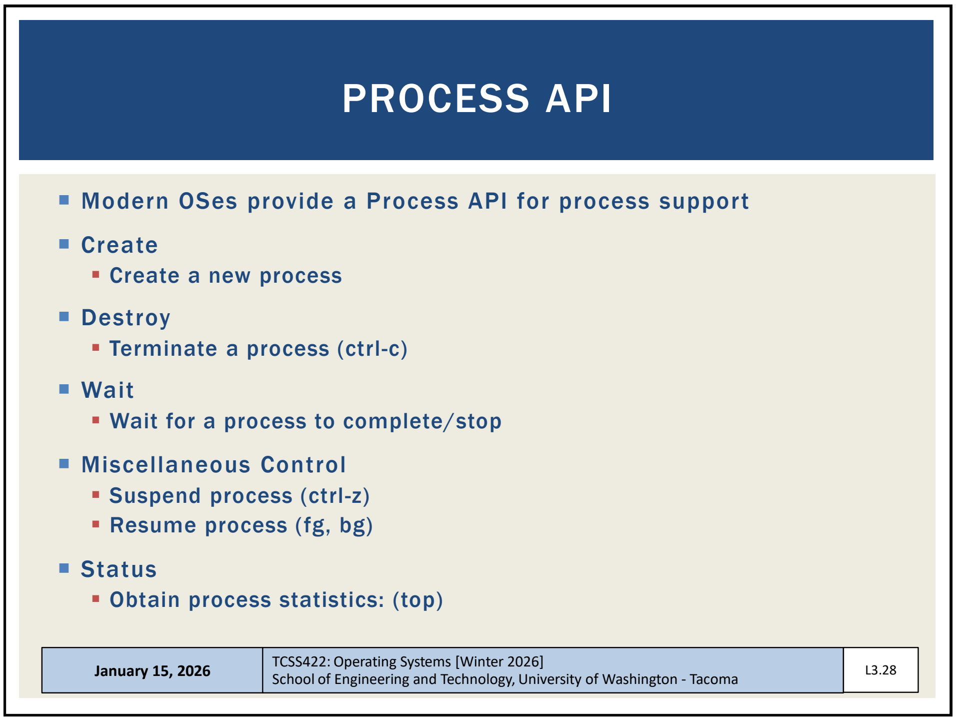
PROCESS

A process is a running program.

- Process comprises of:
  - Memory
    - Instructions ("the code")
    - Data (heap)
  - Registers
    - PC: Program counter
    - Stack pointer

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.27

27

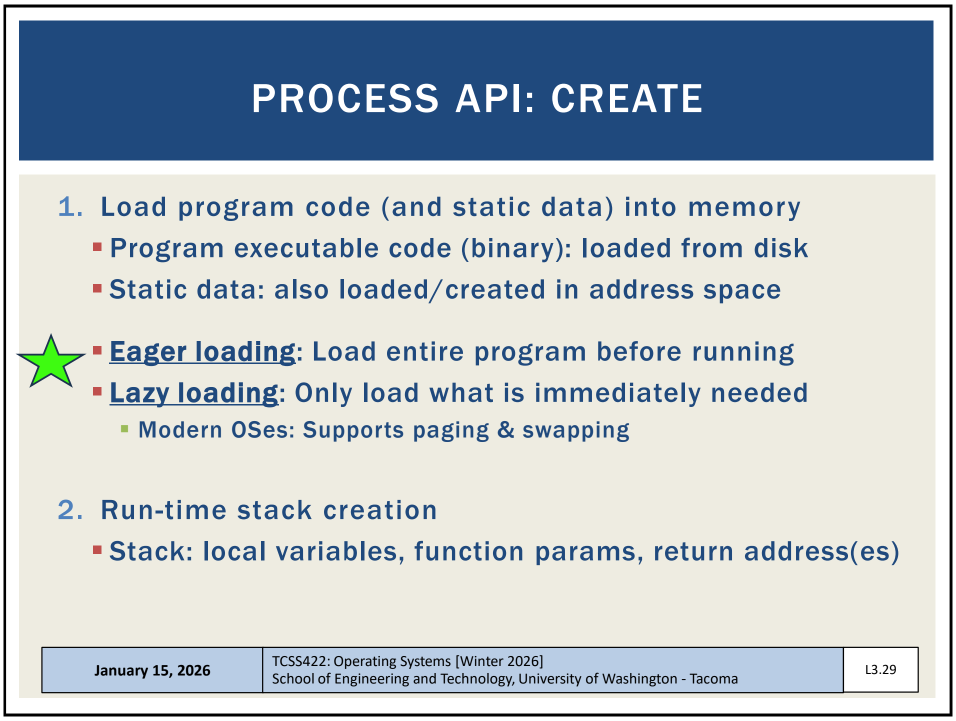


PROCESS API

- Modern OSes provide a Process API for process support
- Create
  - Create a new process
- Destroy
  - Terminate a process (ctrl-c)
- Wait
  - Wait for a process to complete/stop
- Miscellaneous Control
  - Suspend process (ctrl-z)
  - Resume process (fg, bg)
- Status
  - Obtain process statistics: (top)

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.28

28

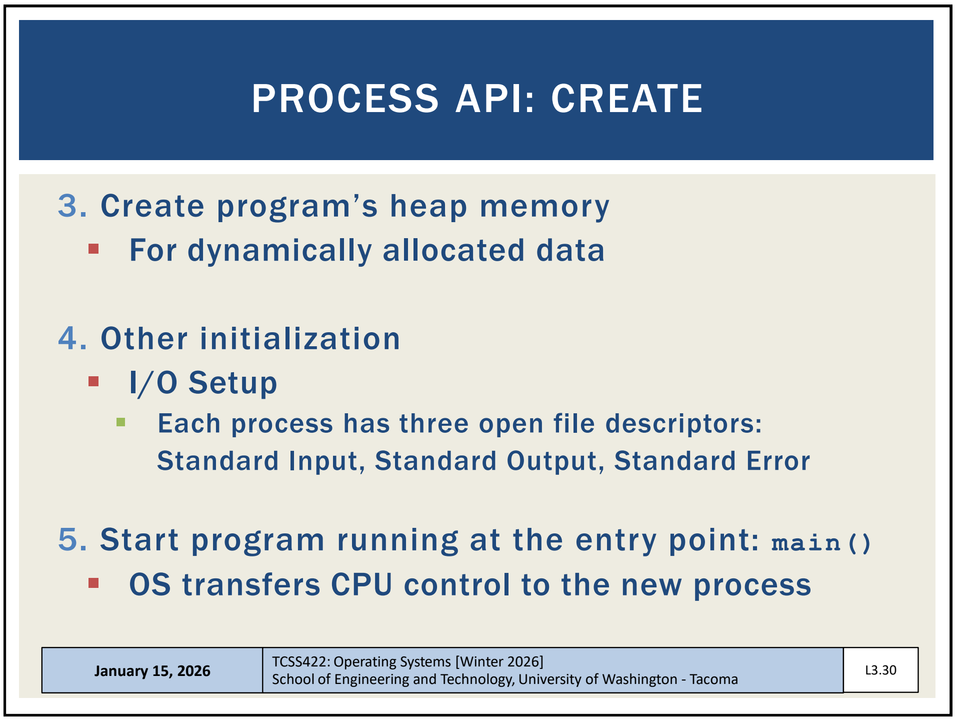


PROCESS API: CREATE

1. Load program code (and static data) into memory
   - Program executable code (binary): loaded from disk
   - Static data: also loaded/created in address space
   - **Eager loading**: Load entire program before running
   - **Lazy loading**: Only load what is immediately needed
     - Modern OSes: Supports paging & swapping
2. Run-time stack creation
   - Stack: local variables, function params, return address(es)

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.29

29

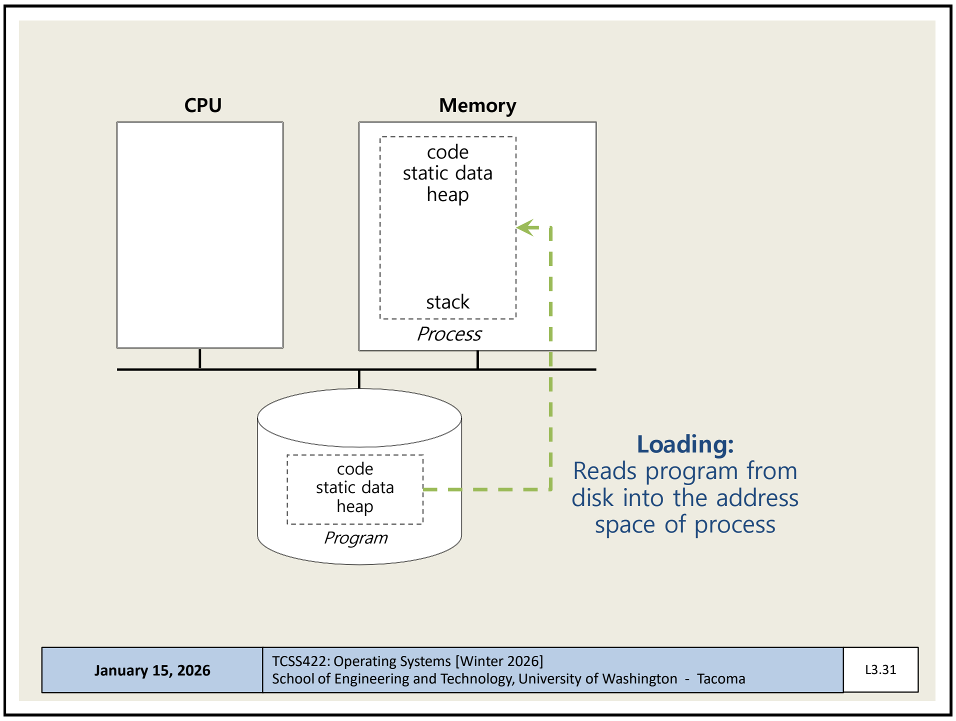


PROCESS API: CREATE

3. Create program's heap memory
   - For dynamically allocated data
4. Other initialization
   - I/O Setup
     - Each process has three open file descriptors: Standard Input, Standard Output, Standard Error
5. Start program running at the entry point: `main()`
   - OS transfers CPU control to the new process

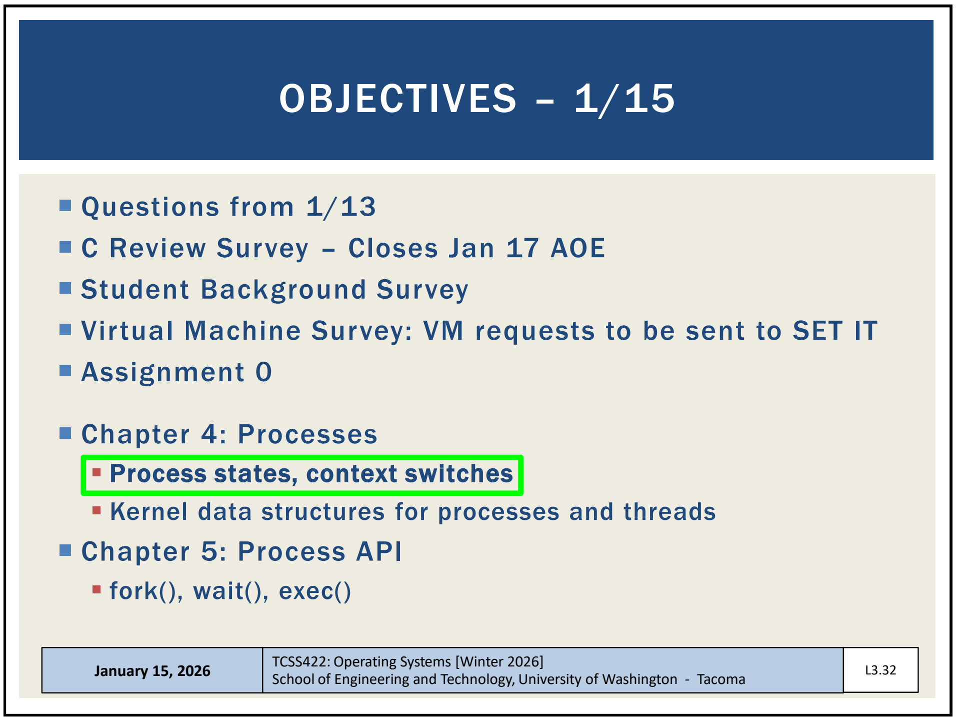
January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.30

30



Loading:
Reads program from disk into the address space of process

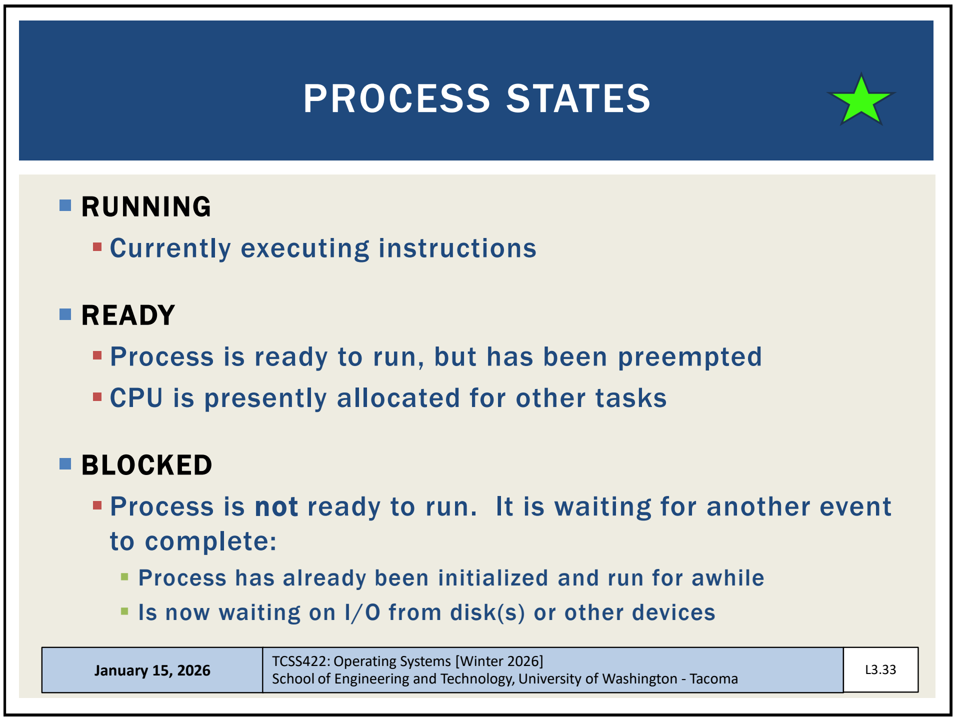
31



OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- Chapter 4: Processes
  - **Process states, context switches**
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

32

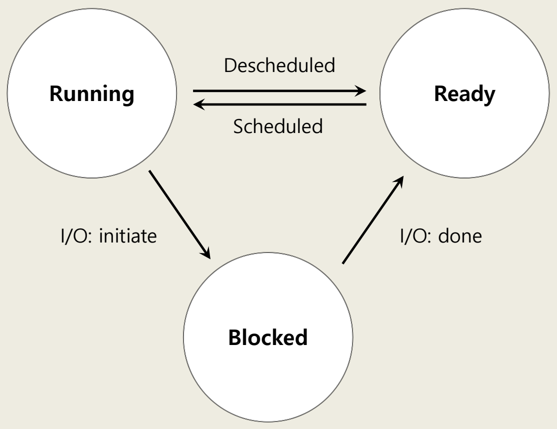


PROCESS STATES

- **RUNNING**
  - Currently executing instructions
- **READY**
  - Process is ready to run, but has been preempted
  - CPU is presently allocated for other tasks
- **BLOCKED**
  - Process is **not** ready to run. It is waiting for another event to complete:
    - Process has already been initialized and run for awhile
    - Is now waiting on I/O from disk(s) or other devices

33

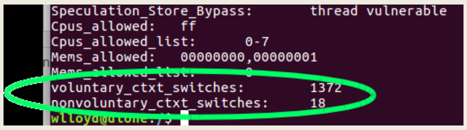
PROCESS STATE TRANSITIONS



34

OBSERVING PROCESS META-DATA

- Can inspect the number of **CONTEXT SWITCHES** made by a process
- Let's run mem.c (from chapter 2)
- cat /proc/{process-id}/status



```
Speculation_Store_Bypass:       thread vulnerable
Cpus_allowed:   ff
Cpus_allowed_list:      0-7
Mems_allowed:   00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        1372
nonvoluntary_ctxt_switches:     18
wlloyd@dione:/$
```

- proc "status" is a virtual file generated by Linux
- Provides a report with process related meta-data
- **What appears to happen to the number of context switches the longer a process runs? (mem.c)**

OBSERVING PROCESS META-DATA

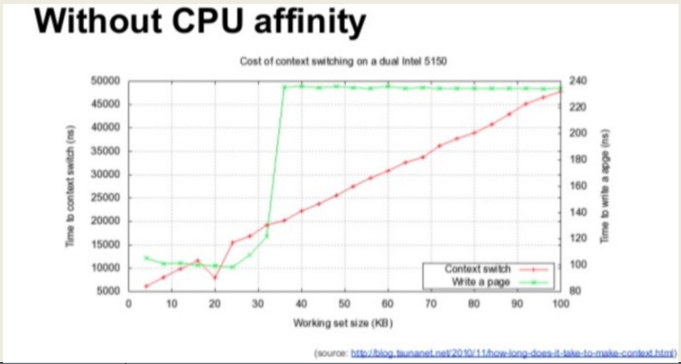
- Can inspect the number of **CONTEXT SWITCHES** made by a process
- Let's run mem.c (from chapter 2)

What is the difference between a voluntary and a non-voluntary context switch ?

- **What appears to happen to the number of context switches the longer a process runs? (mem.c)**

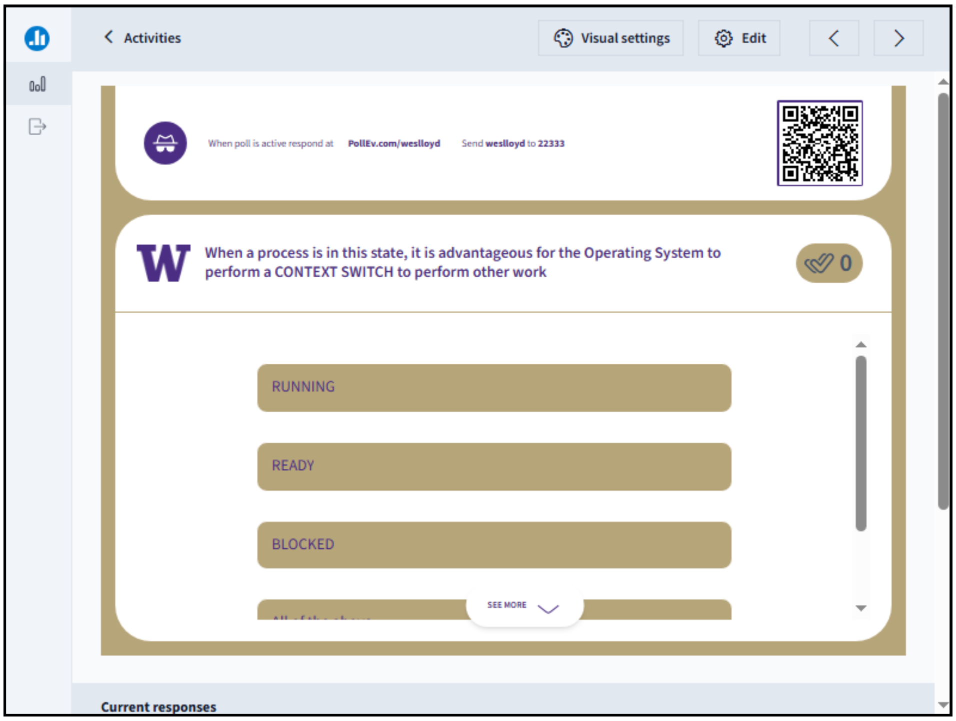
CONTEXT SWITCH

- *How long does a context switch take?*
- 10,000 to 50,000 ns (.01 to .05 ms)
- 2,000 context switches is near 100ms



January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.37

37



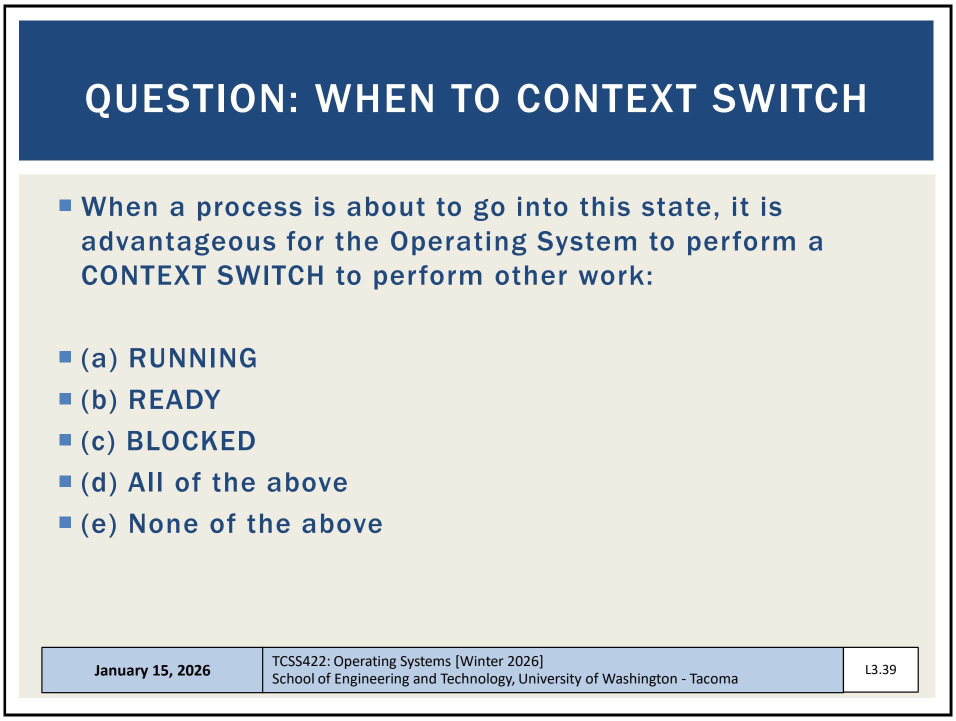
When poll is active respond at PollEv.com/weslloyd Send weslloyd to 22333

When a process is in this state, it is advantageous for the Operating System to perform a CONTEXT SWITCH to perform other work

0

- RUNNING
- READY
- BLOCKED
- All of the above

38



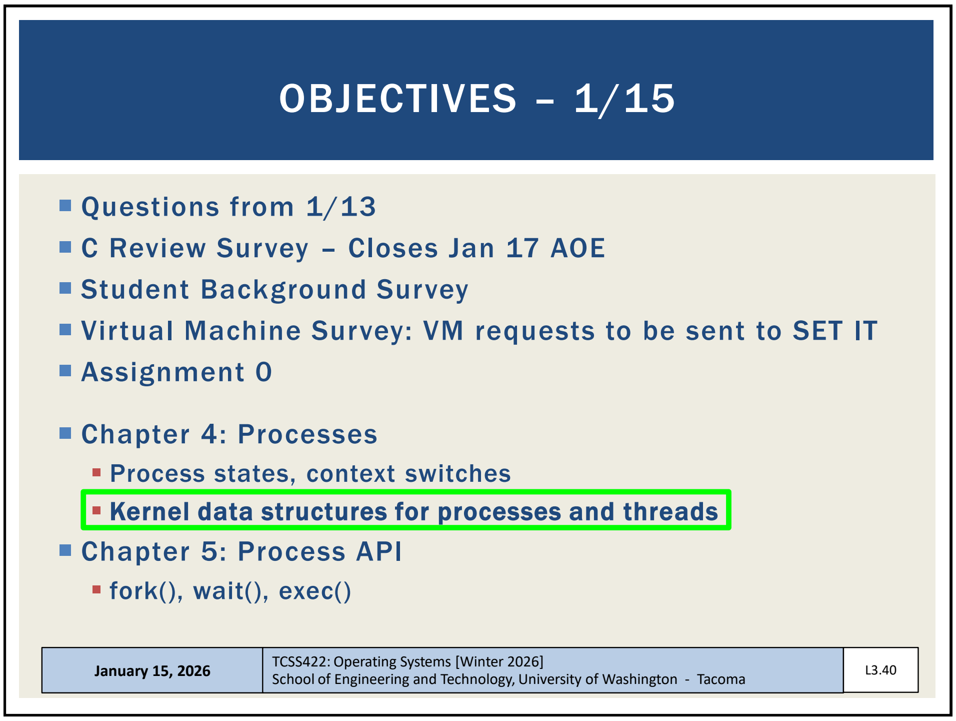
QUESTION: WHEN TO CONTEXT SWITCH

- When a process is about to go into this state, it is advantageous for the Operating System to perform a CONTEXT SWITCH to perform other work:

- (a) RUNNING
- (b) READY
- (c) BLOCKED
- (d) All of the above
- (e) None of the above

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.39

39



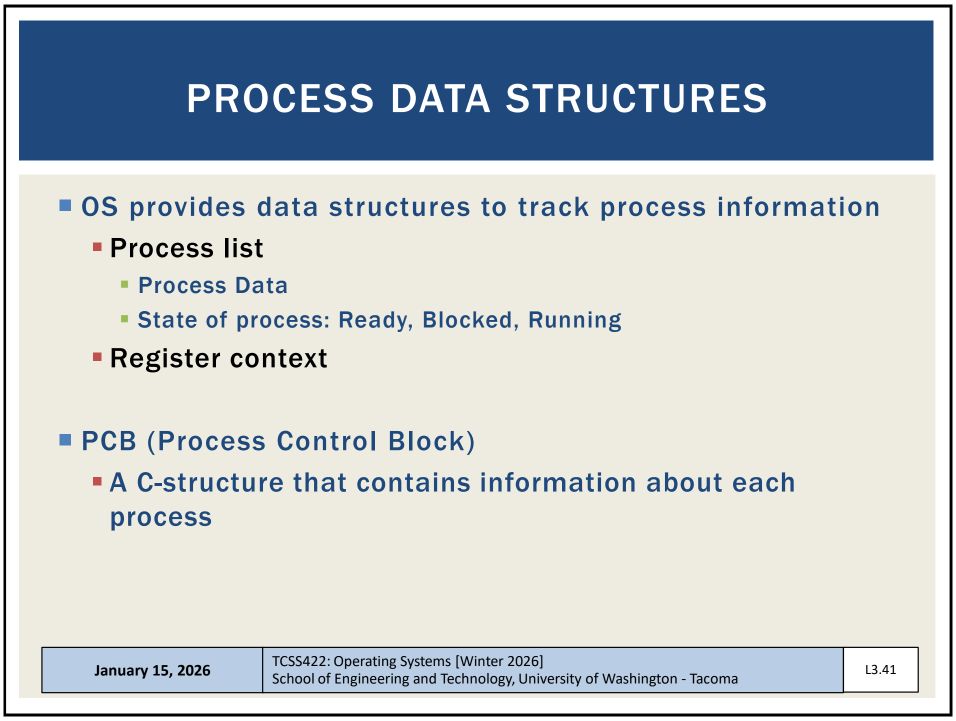
OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0

- Chapter 4: Processes
  - Process states, context switches
  - **Kernel data structures for processes and threads**
- Chapter 5: Process API
  - fork(), wait(), exec()

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.40

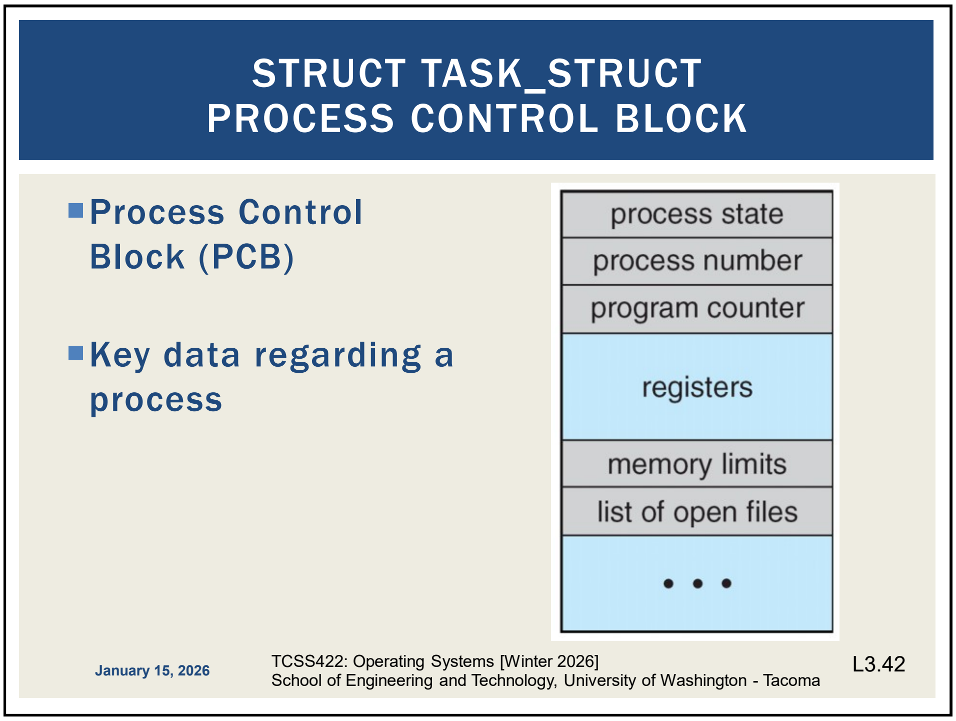
40



PROCESS DATA STRUCTURES

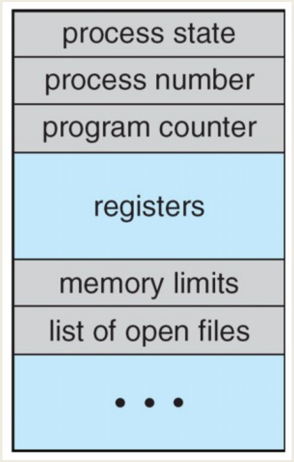
- OS provides data structures to track process information
  - Process list
    - Process Data
    - State of process: Ready, Blocked, Running
  - Register context
- PCB (Process Control Block)
  - A C-structure that contains information about each process

41

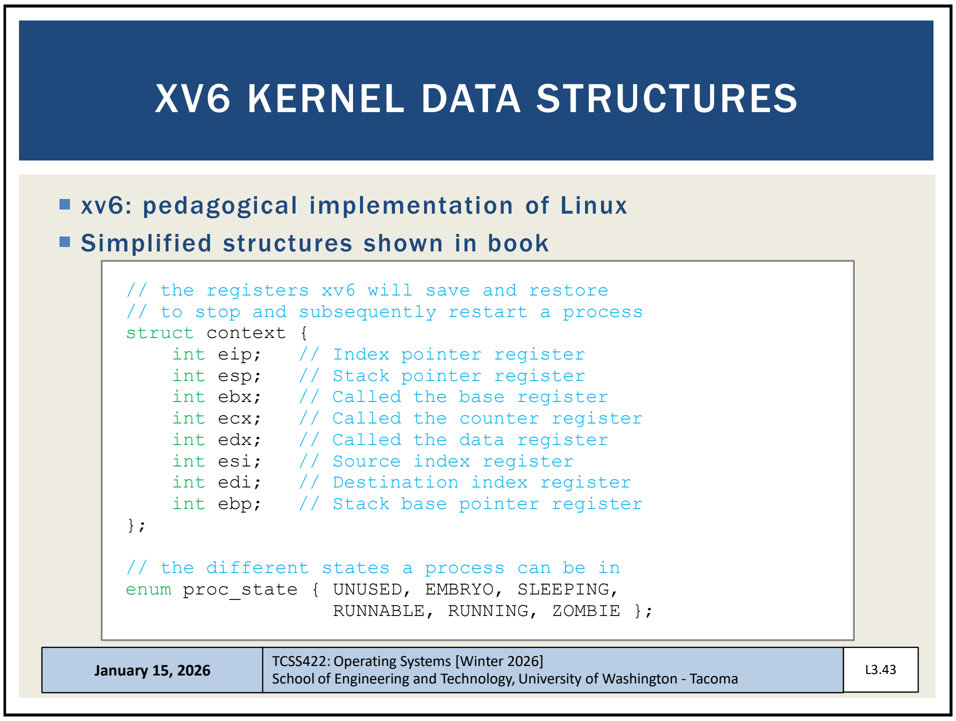


STRUCT TASK_STRUCT PROCESS CONTROL BLOCK

- Process Control Block (PCB)
- Key data regarding a process



42



XV6 KERNEL DATA STRUCTURES

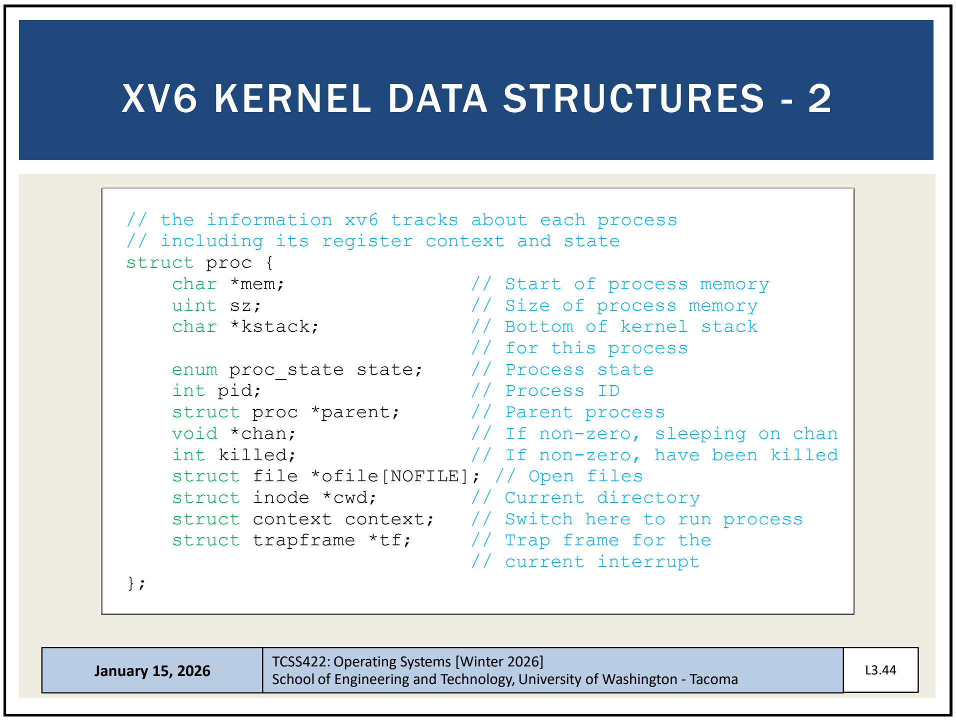
- xv6: pedagogical implementation of Linux
- Simplified structures shown in book

```
// 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 };
```

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.43

43

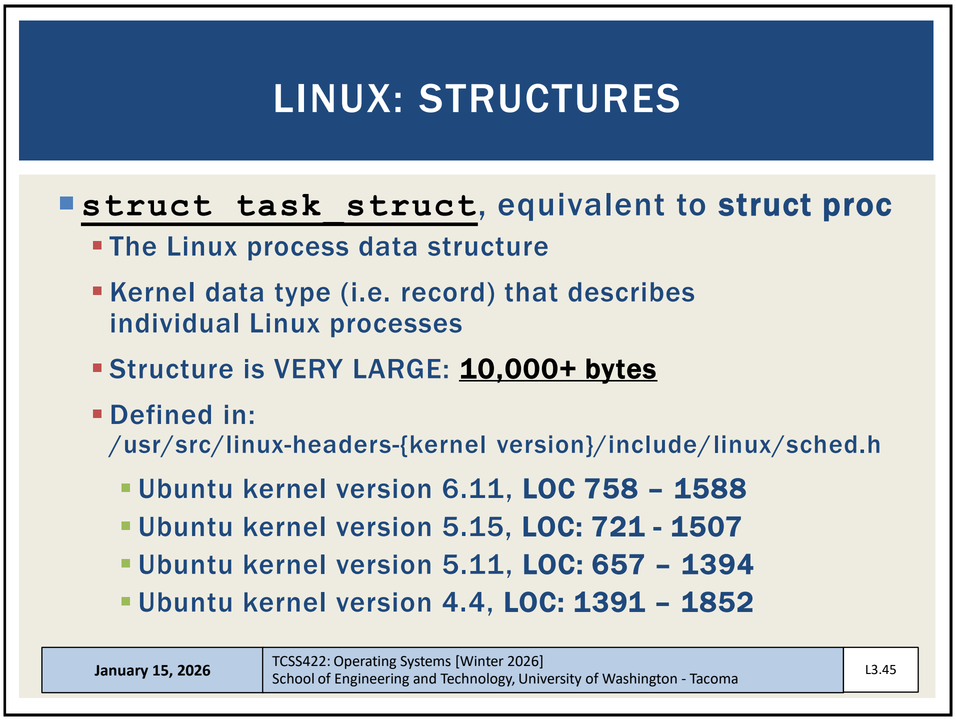


XV6 KERNEL DATA STRUCTURES - 2

```
// 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
};
```

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.44

44

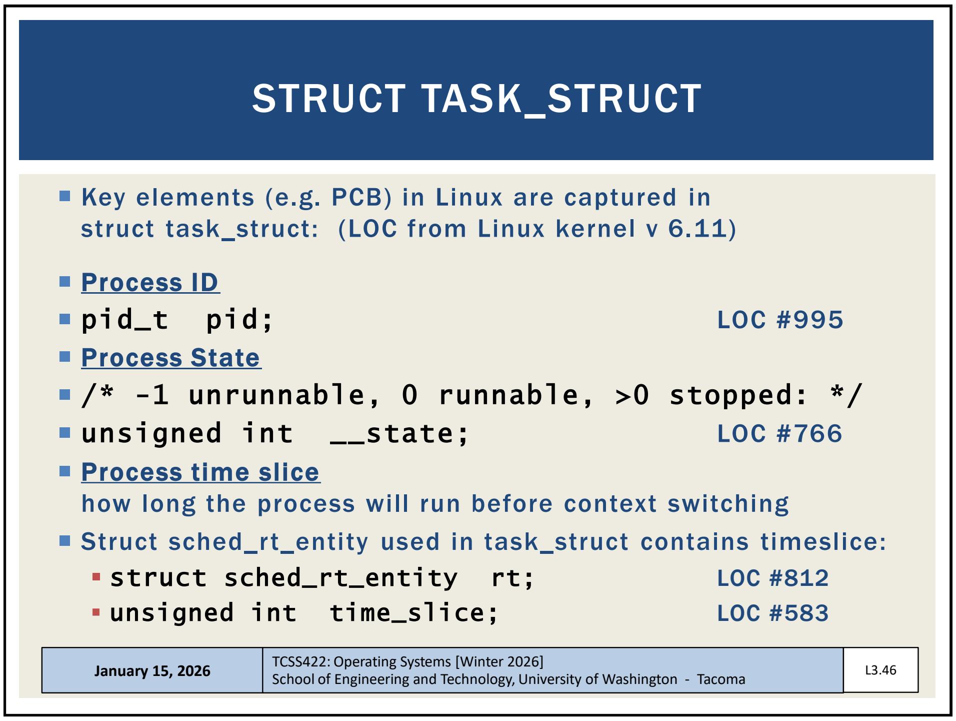


LINUX: STRUCTURES

- `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 6.11, **LOC 758 – 1588**
    - Ubuntu kernel version 5.15, **LOC: 721 - 1507**
    - Ubuntu kernel version 5.11, **LOC: 657 – 1394**
    - Ubuntu kernel version 4.4, **LOC: 1391 – 1852**

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.45

45



STRUCT TASK_STRUCT

- Key elements (e.g. PCB) in Linux are captured in struct task_struct: (LOC from Linux kernel v 6.11)
- **Process ID**
- `pid_t  pid;` LOC #995
- **Process State**
- `/* -1 unrunnable, 0 runnable, >0 stopped: */`
- `unsigned int  __state;` LOC #766
- **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 #812
  - `unsigned int  time_slice;` LOC #583

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.46

46

STRUCT TASK_STRUCT - 2

- **Address space of the process:**
- “mm” is short for “memory map”
- `struct mm_struct  *mm;` LOC #898

- **Parent process**, that launched this one
- `struct task_struct __rcu  *parent;` LOC #1009

- **Child processes** (as a list)
- `struct list_head  children;` LOC #1017

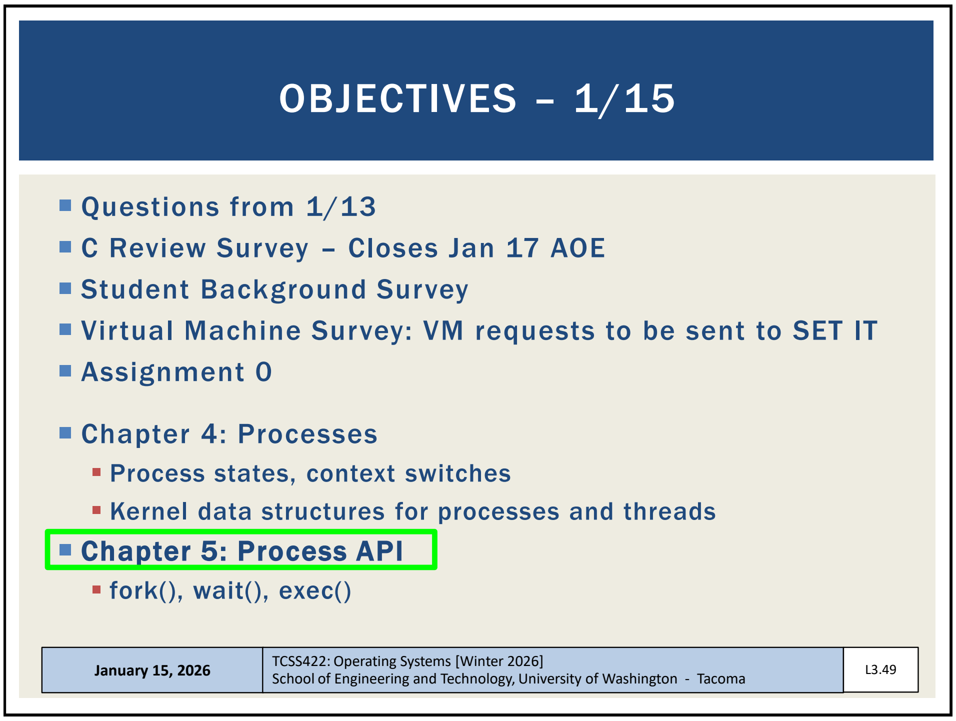
- **Open files**
- `struct files_struct  *files;` LOC #1121

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

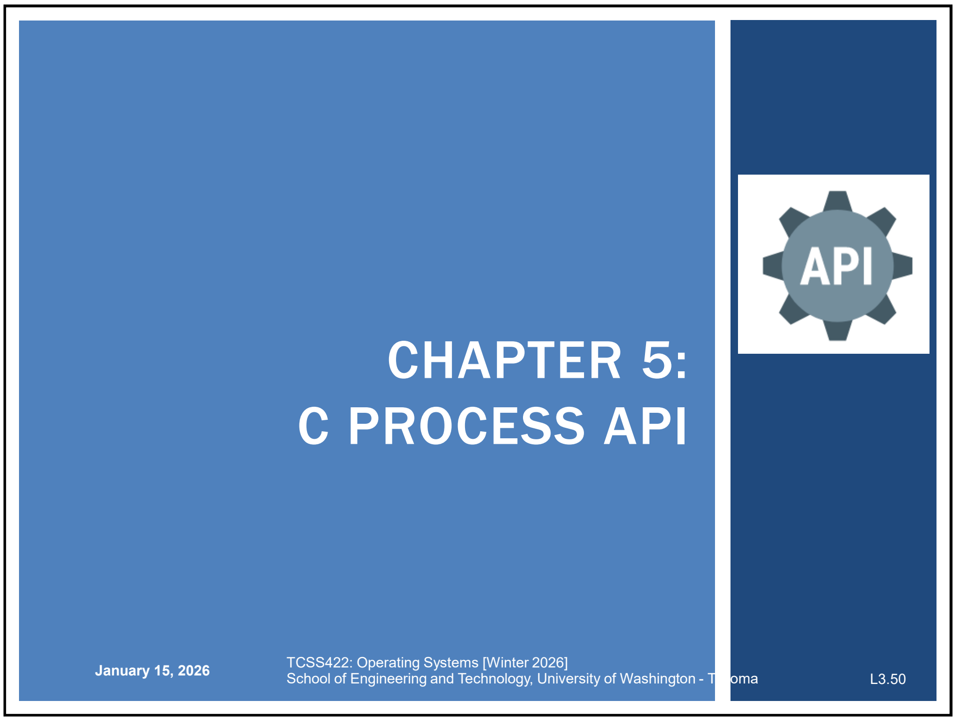


OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), wait(), exec()

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.49

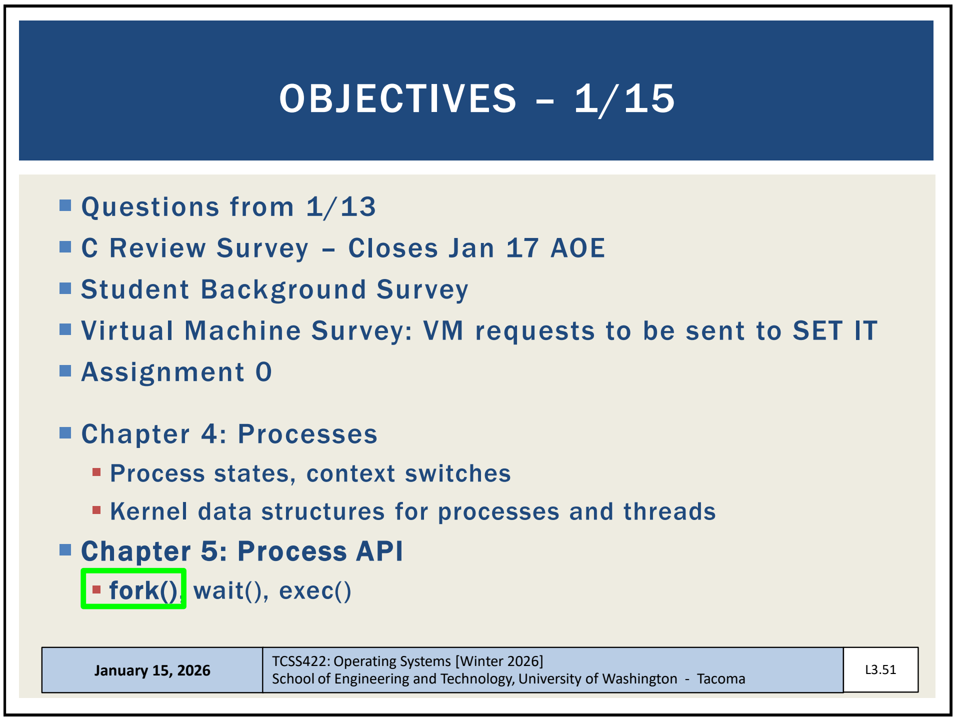
49



CHAPTER 5: C PROCESS API

January 15, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L3.50

50



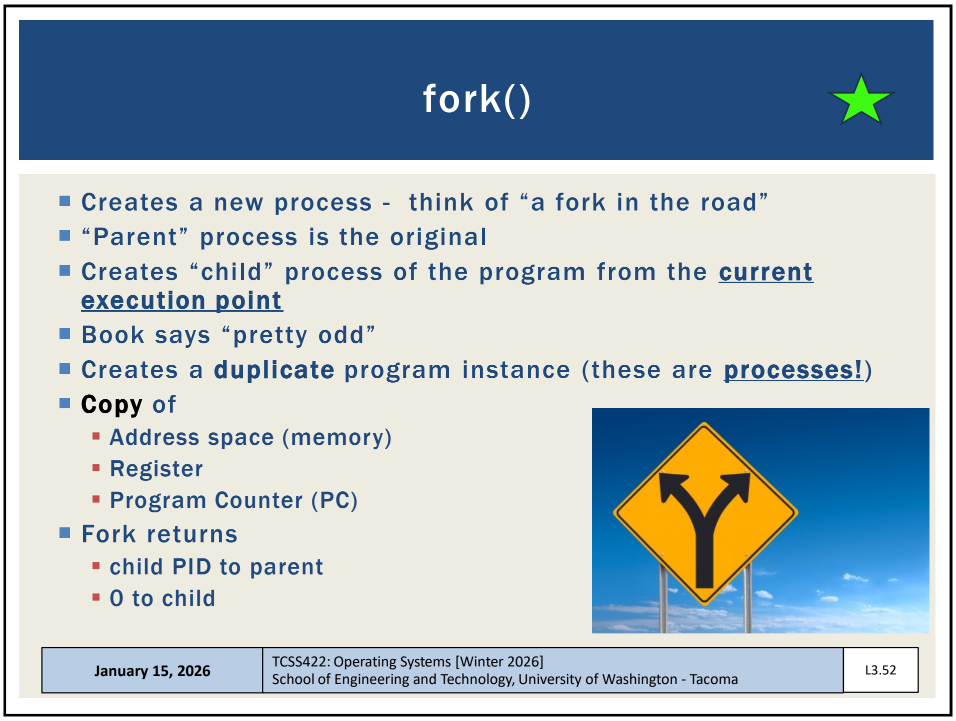
OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0

- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - **fork()**, wait(), exec()

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.51

51



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

January 15, 2026	TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma	L3.52

52

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;
}
```

53

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>
```

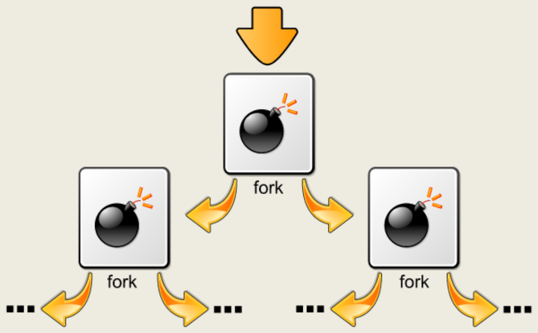
or

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

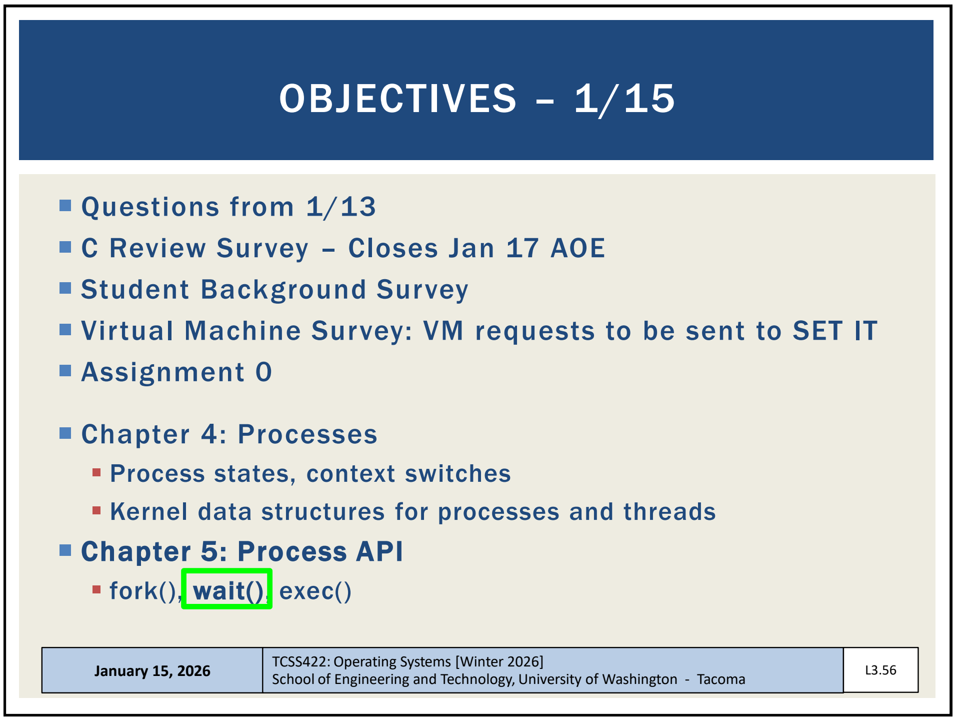
- CPU scheduler determines which to run first

54

:(){ :|: & };:



55



OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), **wait()**, exec()

56

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

57

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;
}
```

58

FORK WITH WAIT - 2

- Deterministic ordering of execution

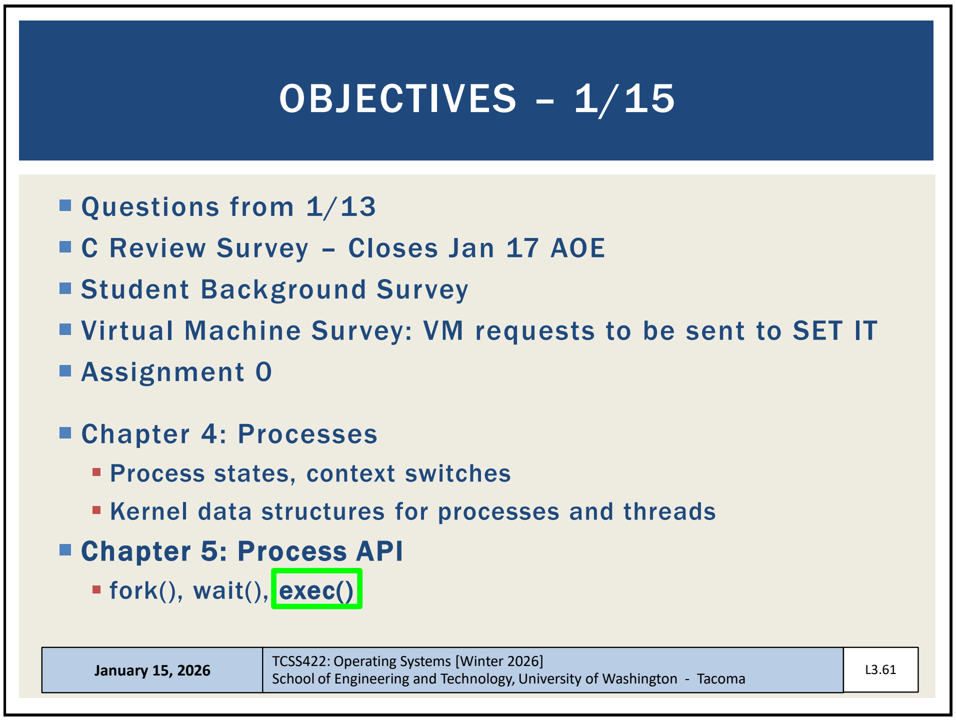
```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

59

FORK EXAMPLE

- Linux example

60

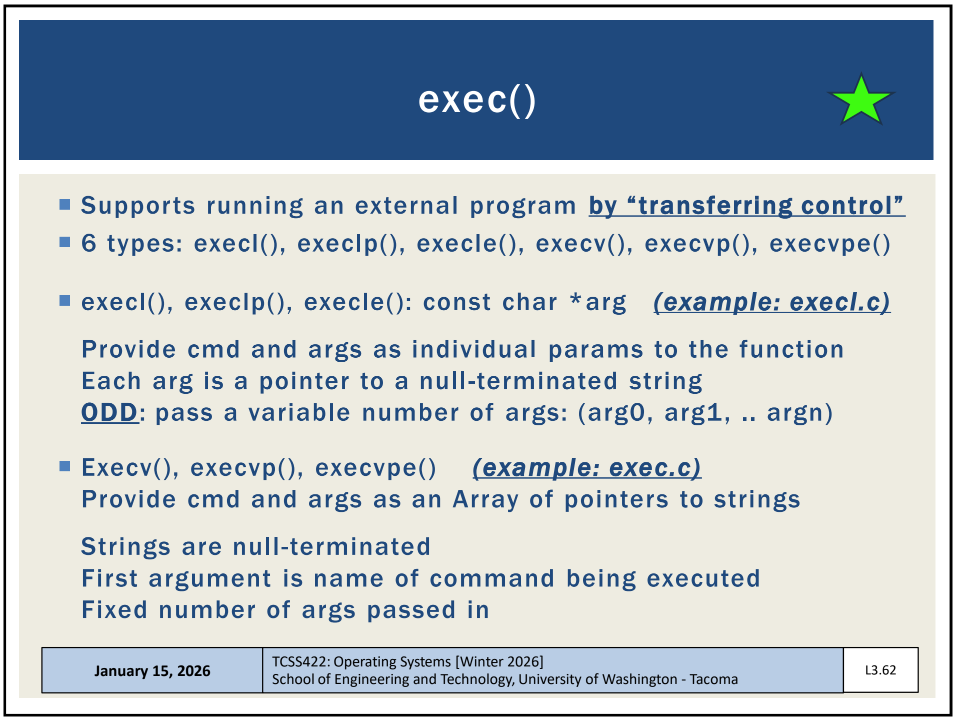


OBJECTIVES – 1/15

- Questions from 1/13
- C Review Survey – Closes Jan 17 AOE
- Student Background Survey
- Virtual Machine Survey: VM requests to be sent to SET IT
- Assignment 0

- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), wait(), **exec()**

61



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

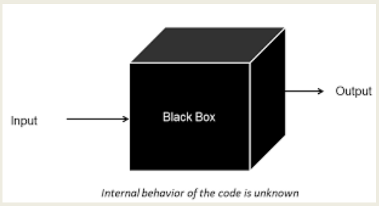
  Strings are null-terminated
  First argument is name of command being executed
  Fixed number of args passed in

62

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… ☺



63

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
        …
```

64

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>
```

65

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);
        …
```

66

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
```

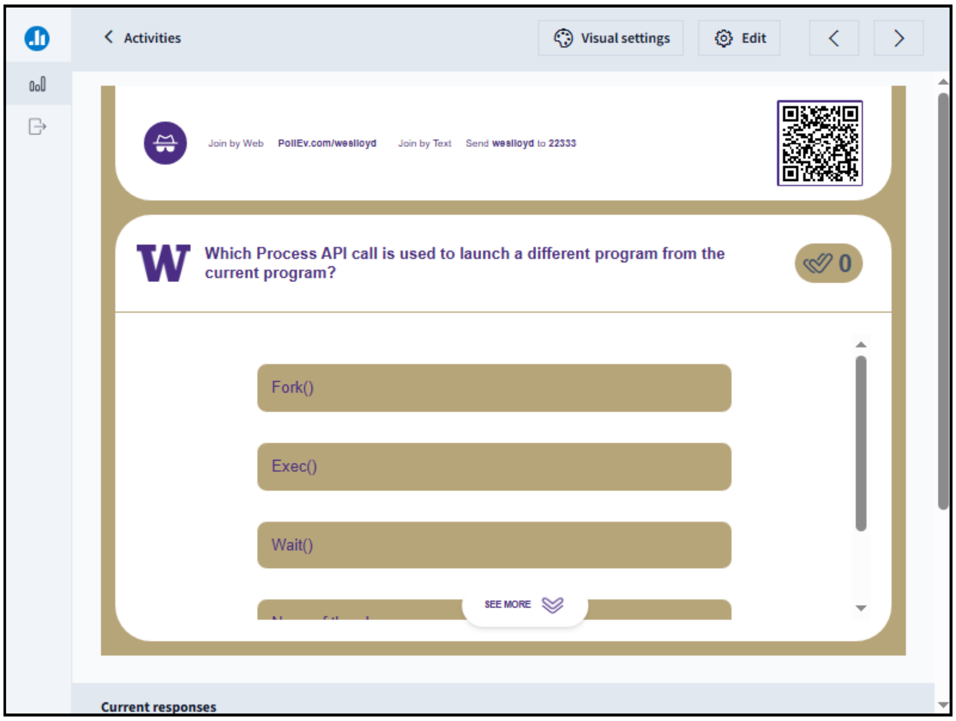
67

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>
```

68


Join by Web PollEv.com/weslloyd Join by Text Send weslloyd to 22333

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

- Fork()
- Exec()
- Wait()

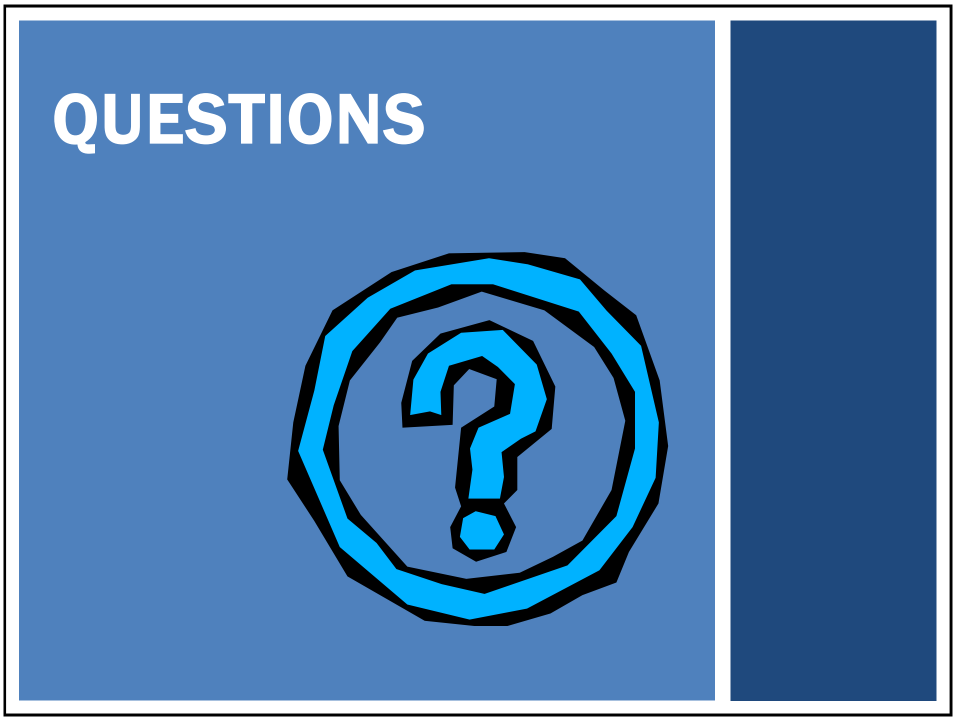
69

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

70



QUESTIONS