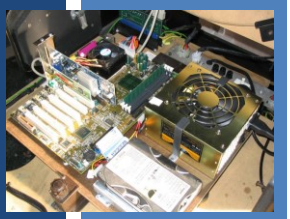
TCSS 422: OPERATING SYSTEMS



Processes & The Process API

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

1

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- 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 have switched to using ARM-based CPUs
  - Motivation: faster, less expensive than Intel-based CPUs

3

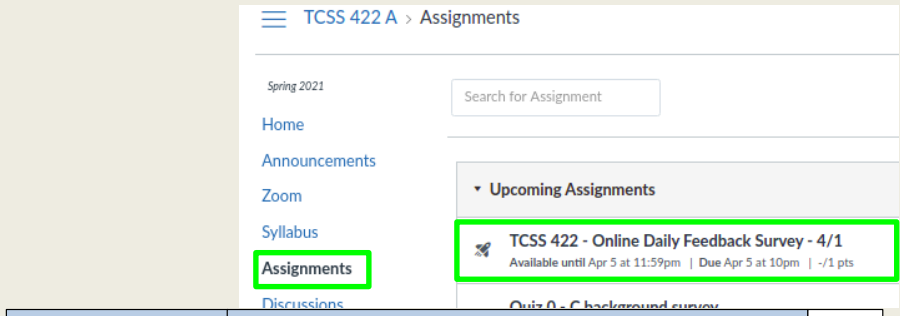
TEXT BOOK COUPON

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

4

ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 9p, closes 11:59p
- Thursday surveys: due ~ Mon @ 9p, closes 11:59p



5

TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

On a scale of 1 to 10, please classify your perspective on material covered in today's class:

1	2	3	4	5	6	7	8	9	10
Mostly Review To Me				Equal New and Review					Mostly New to Me

Question 2 0.5 pts

Please rate the pace of today's class:

1	2	3	4	5	6	7	8	9	10
Slow				Just Right					Fast

6

MATERIAL / PACE

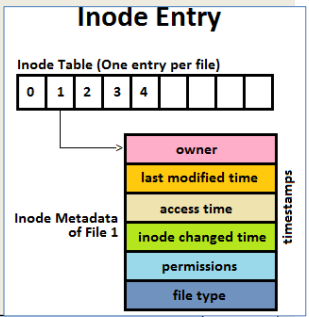
- Please classify your perspective on material covered in today's class (52 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.77 (↑ - previous 6.18)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.71 (↓ - previous 5.91)**

7

FEEDBACK FROM 3/30

- ***I'm still having trouble understanding what inodes are and how they work.***
- An inode is a data structure that tracks all of the files and directories within a Linux or UNIX-based filesystem.
- Every file and directory stored on a disk in a filesystem is allocated an inode as a **FILE RECORD**
- File records are identified by a unique integer known as the "inode".
- File records store metadata about each file or directory.



8

FEEDBACK - 2

- ***What are the values of using threads compared to a processes? When should each be used?***

9

FEEDBACK - 3

- ***In our example of concurrent threads exceeding their time slice and causing wrong output, would there be a way to track whether a thread completed its task or if it timed out before completion?***
- It is necessary to monitor the # of context switches for each thread
- This is available from the virtual file: `/proc/[pid]/status`
- There are two lines: `cat status | grep ctxt`

```
voluntary_ctxt_switches:8084285
nonvoluntary_ctxt_switches:    39709
```

- This is very difficult though – must determine thread IDs
- Program is so fast, by the time the thread IDs are determined, there is not much time before pthread reaches its specified count
- This is an example of an OS monitoring problem
- It is easier to observe context switches if the threads run for awhile
- SEE example: **pthread-check.sh**

10

FEEDBACK - 4

- ***Can the number of pages a program has for it's stack and heap change during runtime?***
- Yes
- The heap will dynamically change to accommodate program memory requirements
  - Requests to malloc()/realloc() gradually increase the heap size
  - Requests to free() will shrink the heap size
- The stack segment could increase if the requirements increase for tracking data involved with function calls
- I checked a small C program
- The stack size was 132 KB (33 x 4KB pages)
- Check the stack size in KB for a process using the command:
  pidstat -p [pid] -l -s
- The stack can be seen in the process virtual memory map - must do hexadecimal math to calculate size:
  cat /proc/[pid]/maps | grep stack

11

WHY VIRTUAL MEMORY?

- ***Trying to conceptualize the reason for virtual addresses (in Operating Systems)***
  - **Security:** if physical addresses were exposed, an attacker could acquire the physical address and attempt to read, modify, write the data
  - **Program Relocation**: because users only see virtual addresses, the OS can physically move programs to new locations without changing any user pointers
  - **Memory defragmentation**: OS can dynamically reorganize memory for better efficiency. All user pointers are virtual. Virtual pointers still work and are translated to new addresses
  - **Shared Libraries**: Two programs can have a virtual address (pointer) to a shared library that is mapped by the OS to a single physical address. The sharing and library location are abstracted. Shared libraries are important to save memory.

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:

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
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/

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/

15

OBJECTIVES – 4/4

- Questions from 3/30
- **C Review Survey – Due Friday Apr 7**
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

16

C REVIEW SURVEY - AVAILABLE THRU 4/7

17

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- **Student Background Survey**
- Virtual Machine Survey: VM requests sent to S. Rondeau
- 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

- **47 of 59 Responses** as of 4/3 @ ~11pm
- Current Standings:
  - Best Office Hours times so far:
    - Rank #1: Friday 12 – 2pm √ (52%)
    - Rank #2: Tues/Thur before class (12 – 3:30p) √ (48%)
  - Best lecture format:
    - Rank #1: Hybrid synchronous w/ recordings √ (84%)
    - Rank #2: In-person w/ recordings

19

TCSS 422 – OFFICE HRS – SPRING 2023

- Fridays 1:30 – 2:30pm – Online (Zoom)
  - This session will be primarily by Zoom, but some days in-person will be available in CP 229
  - Zoom link and reminder will be sent weekly
- Tuesdays 2:30 - 3:30pm – Hybrid (In-Person/Zoom)
  - This session will be in person in CP 229. Zoom will be monitored when there is no student in CP 229.

20

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- **Virtual Machine Survey: VM requests sent to S. Rondeau**
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

21

VIRTUAL MACHINE SURVEY

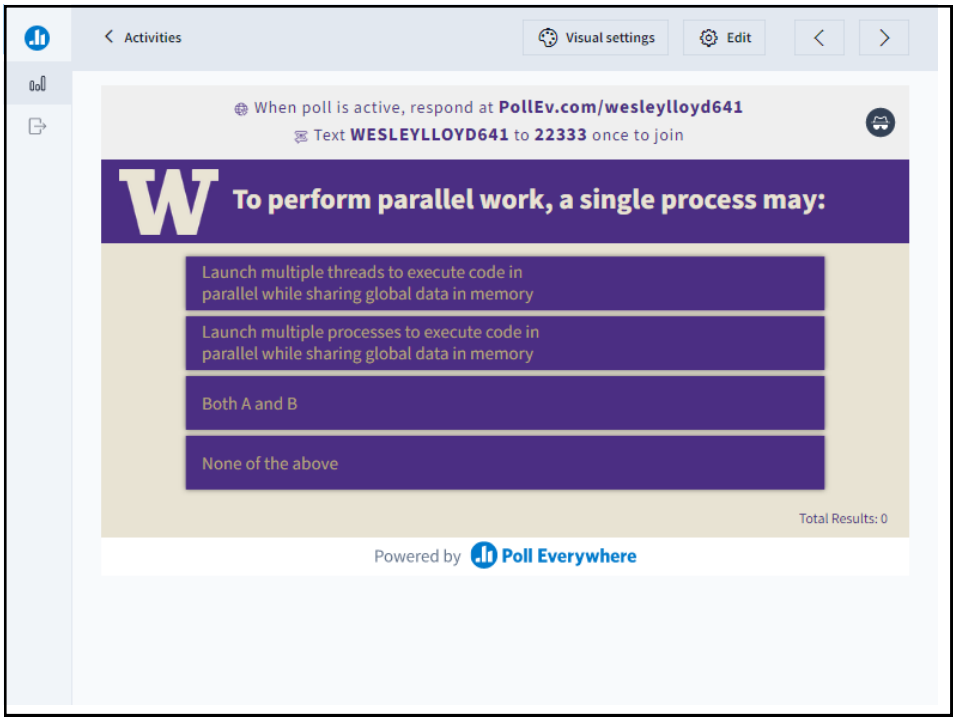
- Please complete the Virtual Machine Survey to request a "School of Engineering and Technology" remote hosted Ubuntu VM
- **https://forms.gle/V2sg4iW1awvhFx4W8**
- **Will close Thursday 4/6…**
- **VM requests will be sent to Stephen Rondeau for set up**

22

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- **Assignment 0**
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

23



24

PARALLEL PROGRAMMING

- To perform parallel work, a single process may:
- A. Launch multiple threads to execute code in parallel while sharing global data in memory
- B. Launch multiple processes to execute code in parallel without sharing global data in memory
- C. Both A and B
- D. None of the above

25

CHAPTER 2 SUMMARY : OPERATING SYSTEM DESIGN GOALS

- **ABSTRACTING THE HARDWARE**
  - Makes programming code easier to write
  - Automate sharing resources – save programmer burden
- **PROVIDE HIGH PERFORMANCE**
  - Minimize overhead from OS abstraction (Virtualization of CPU, RAM, I/O)
  - Share resources fairly
  - Attempt to tradeoff performance vs. fairness → consider priority
- **PROVIDE ISOLATION**
  - User programs can't interfere with each other's virtual machines, the underlying OS, or the sharing of resources

26

CHAPTER 2 SUMMARY : OPERATING SYSTEM DESIGN GOALS - 2

- **RELIABILITY**
  - OS must not crash, 24/7 Up-time
  - Poor user programs must not bring down the system:

    Blue Screen

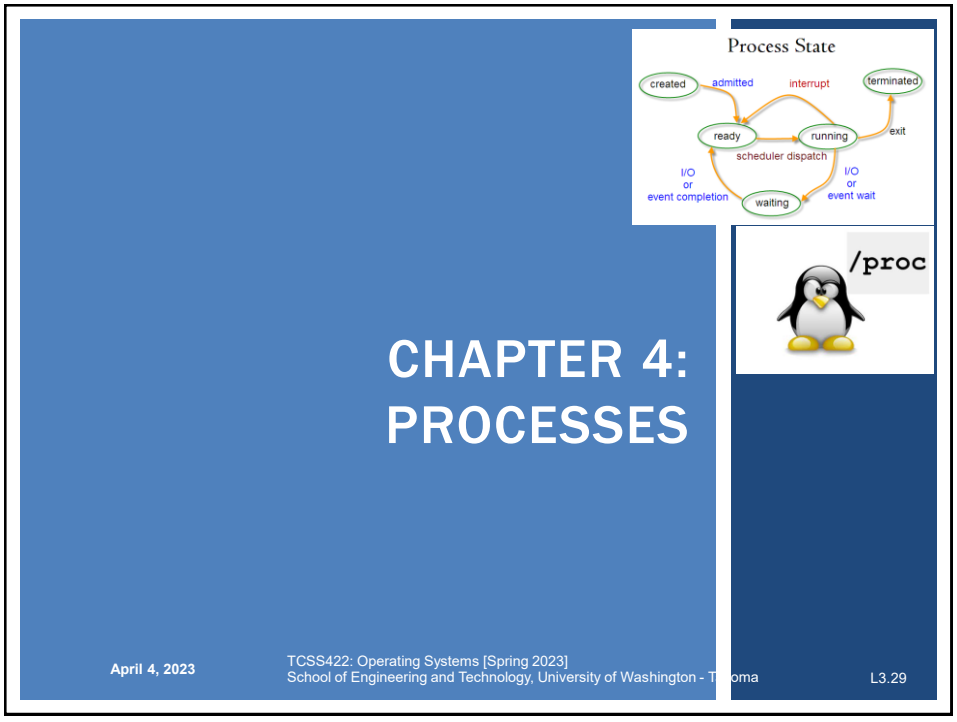
- ***Other Issues:***
  - Energy-efficiency
  - Security (of data)
  - Cloud: Virtual Machines

27

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- **Chapter 4: Processes**
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

28



29

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

30

PROCESS

A process is a running program.

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

31

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)

32

PROCESS API: CREATE

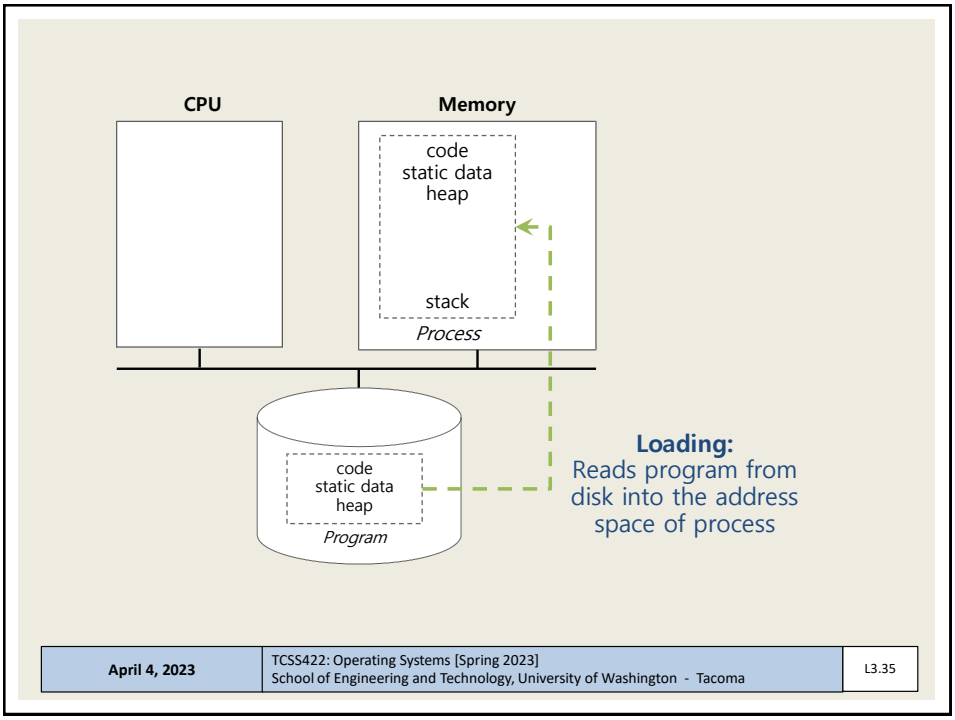
1. Load program code (and static data) into memory
   - Program executable code (binary): loaded from disk
   - Static data: also loaded/created in address space
   - **Eager loading**: Load entire program before running
   - **Lazy loading**: Only load what is immediately needed
     - Modern OSes: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

33

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

34


CPU
Memory
code
static data
heap
stack
Process
code
static data
heap
Program
Loading:
Reads program from disk into the address space of process
April 4, 2023
TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma
L3.35

35


WE WILL RETURN AT 4:50PM
April 4, 2023
TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington -
coma
L3.36

36

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - **Process states, context switches**
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

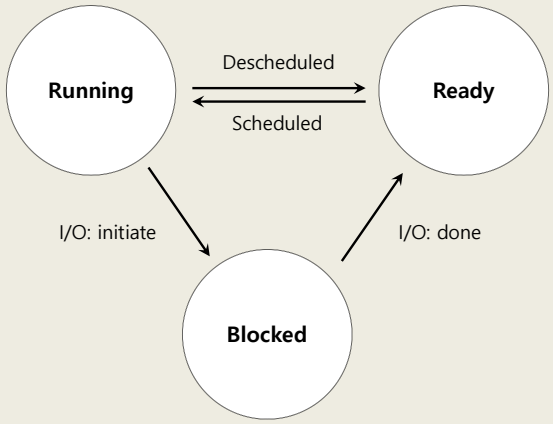
37

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

38

PROCESS STATE TRANSITIONS



39

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
```

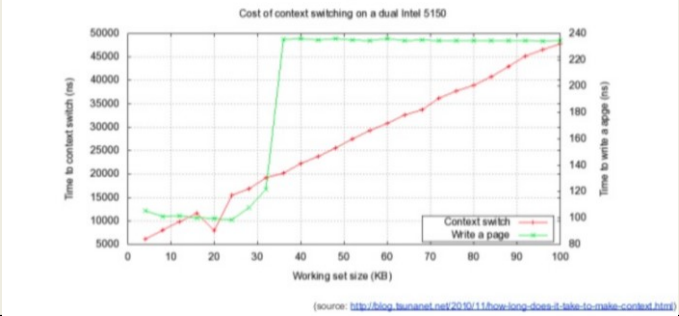
- proc "status" is a virtual file generated by Linux
- Provides a report with process related meta-data
- **What appears to happen to the number of context switches the longer a process runs? (mem.c)**

40

CONTEXT SWITCH

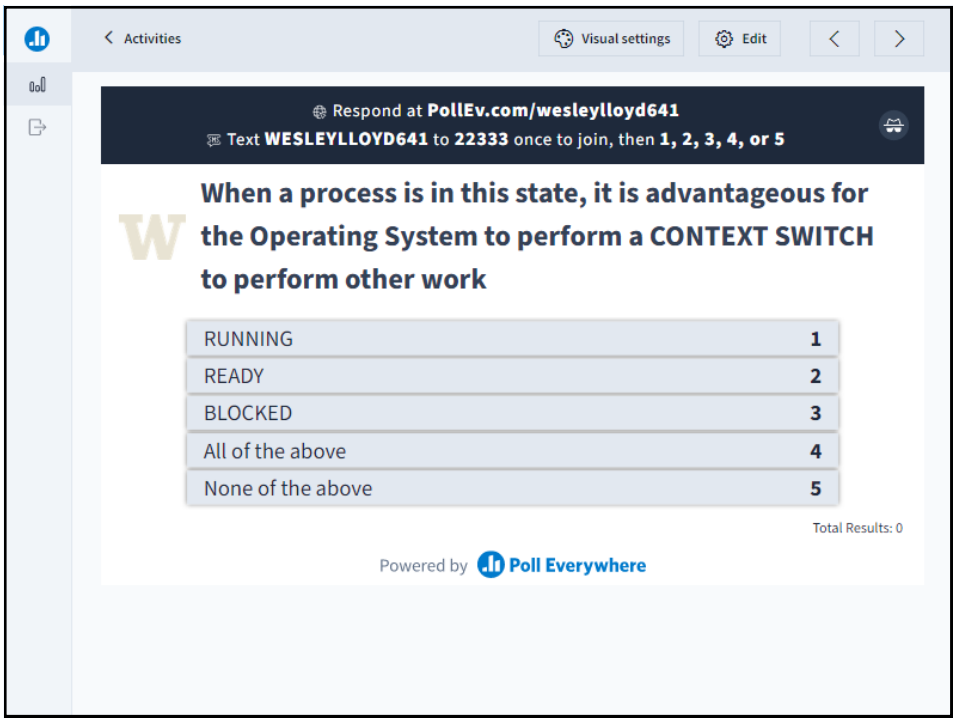
- ***How long does a context switch take?***
- 10,000 to 50,000 ns (.01 to .05 ms)
- 2,000 context switches is near 100ms

Without CPU affinity



(source: http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html)

41



42

QUESTION: WHEN TO CONTEXT SWITCH

- When a process is in 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

43

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0

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

44

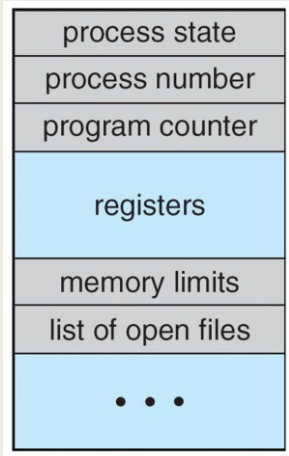
PROCESS DATA STRUCTURES

- 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

45

STRUCT TASK_STRUCT PROCESS CONTROL BLOCK

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



46

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 };
```


47

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
};
```


48

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 20.04 w/ kernel version 5.11, **LOC: 657 – 1394**
    - Ubuntu 20.04 w/ kernel version 4.4, **LOC: 1391 – 1852**

49

STRUCT TASK_STRUCT

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

50

STRUCT TASK_STRUCT - 2

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

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

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

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

51

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

52

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), wait(), exec()

53

CHAPTER 5: C PROCESS API

54

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - **fork()**, wait(), exec()

55

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

56

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;
}
```


FORK EXAMPLE - 2

- Non deterministic ordering of execution

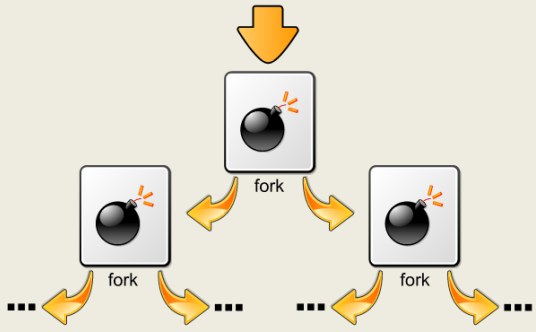
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

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

:(){ :|: & };:



59

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), **wait()**, exec()

60

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

61

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;
}
```


62

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>
```


63

FORK EXAMPLE

- Linux example

64

OBJECTIVES – 4/4

- Questions from 3/30
- C Review Survey – Due Friday Apr 7
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- **Chapter 5: Process API**
  - fork(), wait(), **exec()**

65

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

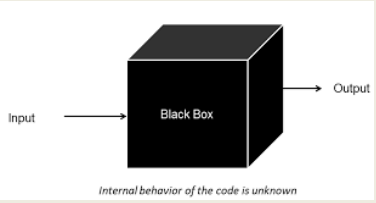
  Strings are null-terminated
  First argument is name of command being executed
  Fixed number of args passed in

66

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…



67

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
        …
```


68

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)

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

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

FILE MODE BITS

```
S_IRWXU
read, write, execute/search by owner
S_IRUSR
read permission, owner
S_IWUSR
write permission, owner
S_IXUSR
execute/search permission, owner
S_IRWXG
read, write, execute/search by group
S_IRGRP
read permission, group
S_IWGRP
write permission, group
S_IXGRP
execute/search permission, group
S_IRWXO
read, write, execute/search by others
S_IROTH
read permission, others
S_IWOTH
write permission, others
```

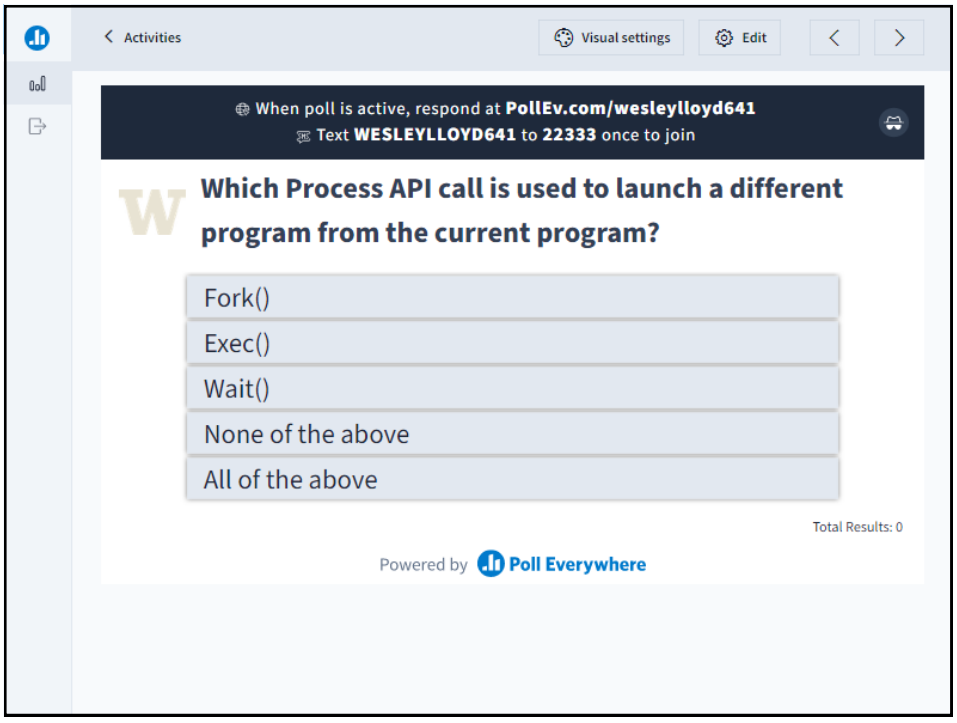

71

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>
```


72



73

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

74

QUESTIONS

75