TCSS 422: OPERATING SYSTEMS

Processes & The Process API

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

April 6, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

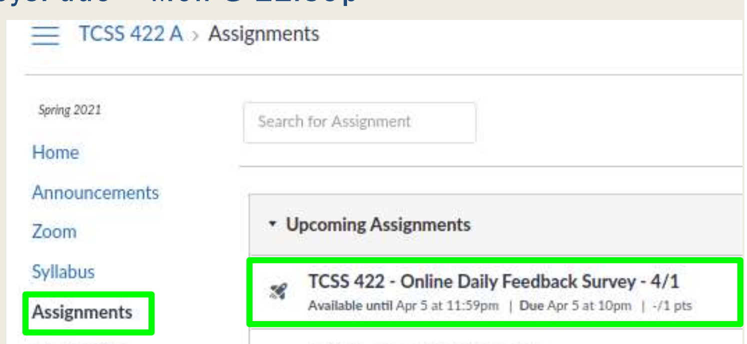
April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.2

TEXT BOOK COUPON

- 15% off textbook code: **INSPIRE15** (*through Friday April 9*)
- https://www.lulu.com/shop/remzi-arpaci-dusseau-and-andrea-arpaci-dusseau/operating-systems-three-easy-pieces-softcover-version-100/paperback/product-23779877.html?page=1&pageSize=4

ONLINE DAILY FEEDBACK SURVEY

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



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

April 6, 2021 TCSS422: Computer Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L3.5

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (61 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.92 (↑ - previous 5.59)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.57 (↑ - previous 5.33)

April 6, 2021 TCSS422: Computer Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L3.6

FEEDBACK

- ***When to use multi-thread and multi-core?***
  - "Embarrassingly parallel" programs
    - MAP-REDUCE, divide and conquer
    - Programs that process a large volume of data, but where processing can be decomposed into independent chunks
    - Chunks can be processed in parallel without coordination
  - Processing tasks that don't require shared memory
    - Web services where each user has separate state
  - Parallel algorithms and code
    - Requires coordination, but is manageable through known sharing and
- ***What does synchronization with processes and threads mean?***
  - Synchronization: coordinating access to shared memory
  - Applies to threads, as processes do not have shared memory

FEEDBACK - 2

- ***Implementation of the PIDs and threads is not yet clear***
  - Chapter 4 introduces processes, threads will follow
- ***What are the advantages of using the Linux /proc filesystem?***
  - Provides ability to inspect low-level details of how processes/threads are running (e.g. if you wanted to write you own top/htop utility)
  - Provides ability to inspect resource utilization and management being provided by the operating system

FEEDBACK - 3

- ***I feel like we went over the running of these programs a little fast, I am not sure if I could get these programs to run in a terminal as lots of commands were being used on the screen.***
- ***Can we get the today's lecture sample codes for reviewing?***
  - Code examples from class are linked from the schedule page:

    Source Code Examples

    Source code for examples from class are posted **[HERE]**.

    http://faculty.washington.edu/wlloyd/courses/tcss422/examples/
- ***Can we get lecture slides instead pdf? Some sample code pictures were overlapped.***
  - This has been corrected. Slides have been reposted. Thank you !

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://www.ec2instances.info/

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/

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

STUDENT BACKGROUND SURVEY

- Please complete the Student Background Survey
- https://forms.gle/yr6Dc9x9rX516U6t6

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

OBJECTIVES – 4/6

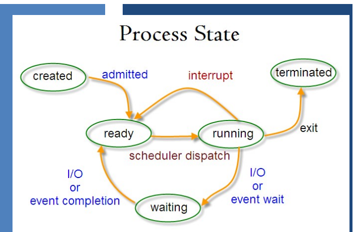
- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- **Assignment 0**
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

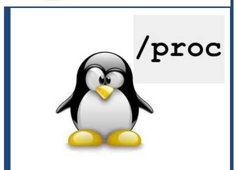
April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.17

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.18





CHAPTER 4: PROCESSES

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

PROCESS

A process is a running program.

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

PROCESS API

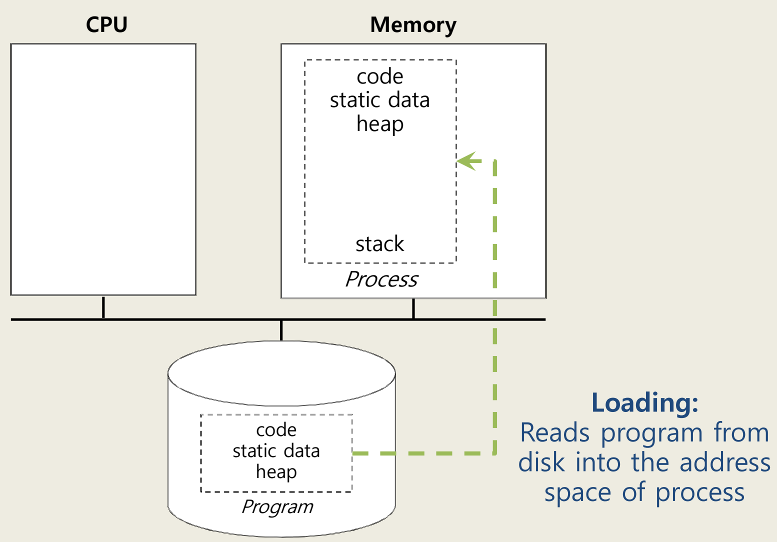
- Modern OSes provide a Process API for process support
- Create
  - Create a new process
- Destroy
  - Terminate a process (ctrl-c)
- Wait
  - Wait for a process to complete/stop
- Miscellaneous Control
  - Suspend process (ctrl-z)
  - Resume process (fg, bg)
- Status
  - Obtain process statistics: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

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)

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



April 6, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L3.25

OBJECTIVES – 4/6

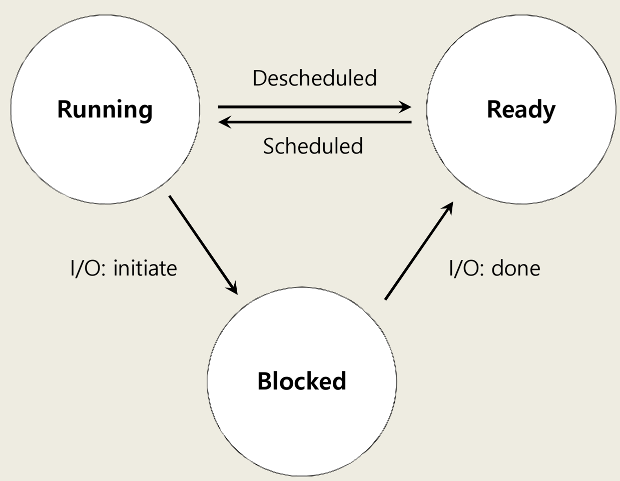
- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- Student Background Survey
- Virtual Machine Survey: VM requests sent to S. Rondeau
- Assignment 0
- Chapter 4: Processes
  - Process states, context switches
  - Kernel data structures for processes and threads
- Chapter 5: Process API
  - fork(), wait(), exec()

April 6, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L3.26

PROCESS STATES

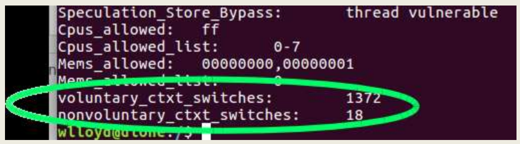
- **RUNNING**
  - Currently executing instructions
- **READY**
  - Process is ready to run, but has been preempted
  - CPU is presently allocated for other tasks
- **BLOCKED**
  - Process is not ready to run. It is waiting for another event to complete:
    - Process has already been initialized and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

PROCESS STATE TRANSITIONS



OBSERVING PROCESS META-DATA

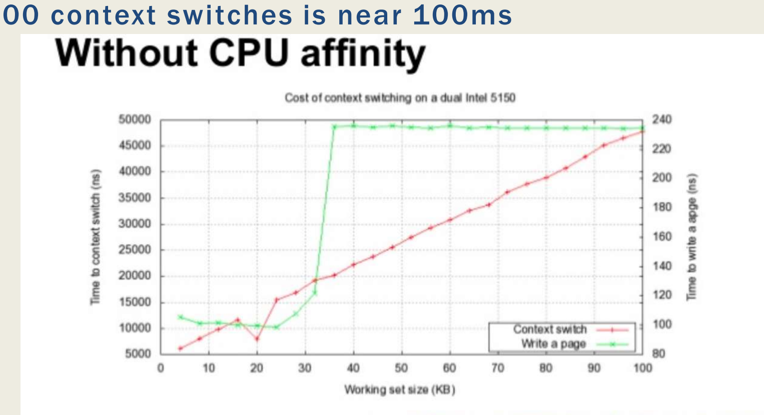
- Can inspect the number of **CONTEXT SWITCHES** made by a process
- Let's run mem.c (from chapter 2)
- cat /proc/{process-id}/status



- 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)**

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



W

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

RUNNING | READY | BLOCKED | All of the above | None of the above

April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.31

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

April 6, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.32

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

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

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 };
```


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
};
```

LINUX: STRUCTURES

- **struct task_struct**, equivalent to struct proc
  - The Linux process data structure
  - VERY LARGE: **10,000+ bytes**
  - /usr/src/linux-headers-{kernel version}/include/linux/sched.h
    - ~ LOC 1391 – 1852 (4.4.0-170)
  - **task_struct** originally stored in the kernel's stack space
    - Limited to 2 x 4KB pages = 8 KB
  - **task_struct** is LARGE, has been moved outside the kernel stack
  - The smaller **thread_info** struct is now stored on the kernel's stack & provides a ptr to **task_struct** allocated using the slab allocator
  - Slab allocator allocates memory for common data structures in Linux
- **struct thread_info**, provides ptr to task_struct
  - thread_info.h is at:

  /usr/src/linux-headers-{kernel version} /arch/x86/include/asm/

LINUX: THREAD_INFO

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        __u32                   flags;          /* low level flags */
        __u32                   status;         /* thread synchronous flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable,
                                                   <0 => BUG */
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        void __user             *sysenter_return;
#ifdef CONFIG_X86_32
        unsigned long           previous_esp;   /* ESP of the previous stack in
                                                   case of nested (IRQ) stacks
                                                */
        __u8                    supervisor_stack[0];
#endif
        int                     uaccess_err;
};
```

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

WE WILL RETURN AT 5:00PM

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

CHAPTER 5: C PROCESS API

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

April 6, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.43

fork()

- Creates a new process - think of "a fork in the road"
- "Parent" process is the original
- Creates "child" process of the program from the **current execution point**
- Book says "pretty odd"
- Creates a **duplicate** program instance (these are **processes!**)
- **Copy** of
  - Address space (memory)
  - Register
  - Program Counter (PC)
- Fork returns
  - child PID to parent
  - 0 to child

April 6, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.44

FORK EXAMPLE

- **p1.c**

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

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

April 6, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.45

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>
```

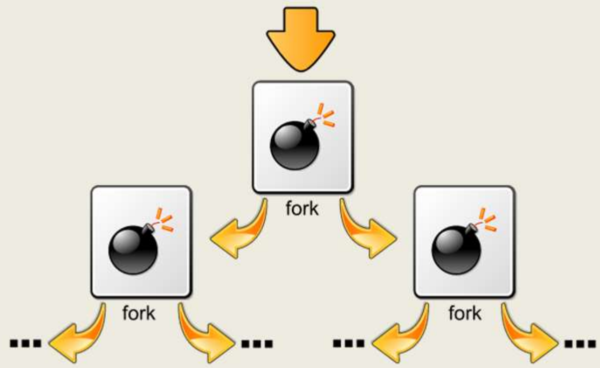
or

```
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 6, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L3.46

:(){ :|: & };:



OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

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

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;
}
```

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>
```


FORK EXAMPLE

- Linux example

OBJECTIVES – 4/6

- Questions from 4/1
- C Review Survey – Closes Friday Apr 9
- 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()

exec()

- Supports running an external program <u>by "transferring control"</u>
- 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
  <u>ODD</u>: pass a variable number of args: (arg0, arg1, .. argn)

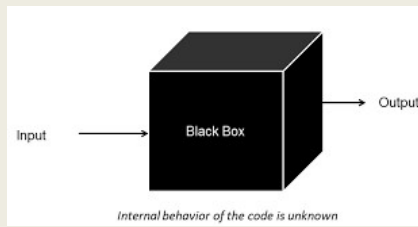
- Execv(), execvp(), execvpe() ***(example: exec.c)***
  Provide cmd and args as an Array of pointers to strings

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

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... ☺



EXEC EXAMPLE

```c
#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>
```

April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.57

EXEC WITH FILE REDIRECTION (OUTPUT)

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

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

April 6, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L3.58

FILE MODE BITS

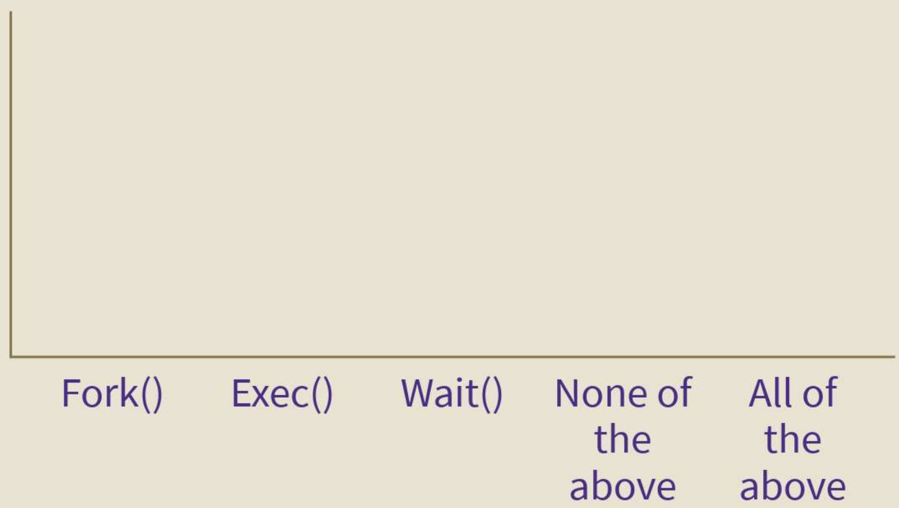
```
S_IRWXU
read, write, execute/search by owner
S_IRUSR
read permission, owner
S_IWUSR
write permission, owner
S_IXUSR
execute/search permission, owner
S_IRWXG
read, write, execute/search by group
S_IRGRP
read permission, group
S_IWGRP
write permission, group
S_IXGRP
execute/search permission, group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


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>
```

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



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

Total Results

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

QUESTIONS