


TCSS 422: OPERATING SYSTEMS

**The Abstraction:
The Process, Process API,
Limited Direct Execution**



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

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

FEEDBACK FROM 1/4/2017

- What point(s) remain least clear...?
 - Why CentOS? (for the programming assignments) Can I get away with Debian?
 - Difference between a thread and a process
 - How to create threads, etc.?
 - Concurrency- what is the main idea/concept?
 - The example of the non-atomic threading
 - What is atomic?
 - Synchronization of the variable between threads
 - EXAMPLE PROGRAM:** How are the random incorrect values determined when the counter is not in sync?

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.2

FEEDBACK - 2

- What point(s) remain least clear...?
 - Visualization of memory
 - If one program has the same virtual memory – it still has a different physical memory?
 - Each program has its own virtual memory address space that is the entire address space of the physical machine (e.g. 4GB)
 - Yes, each program's virtual memory address space map's to a different physical address location...
 - If so, how does the OS map to this memory?
 - Through address translation, a feature of the OS with hardware (on CPU) support

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.3

FEEDBACK - 3

- What point(s) remain least clear...?
 - What is a context switch?
 - How is OS scheduling determined?
 - OS abstraction
 - Specifics of (the) OS

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.4

OBJECTIVES

- Process states
- Process data structures
- Process API – Ch. 5
- Limited Direct Execution – Ch. 6

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.5

CPU VIRTUALIZING

- How should the CPU be shared?
- Time Sharing:
Run one process, pause it, run another
- How do we SWAP processes in and out of the CPU efficiently?
 - Goal is to minimize **overhead** of the swap

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.6

PROCESS

A process is a running program.

Process comprises of:

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

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.7

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 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.8

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 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.9

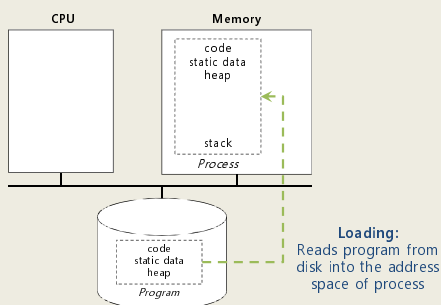
PROCESS API: CREATE

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 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.10



January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.11

PROCESS STATES

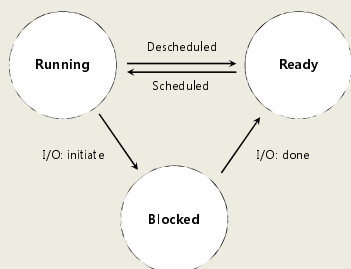
- **Running**
 - Currently executing instructions
- **Ready**
 - Process is ready to run, but has been preempted
 - CPU is presently allocated for other tasks
- **Blocked**
 - Process is **not** ready to run. It is waiting for another event to complete:
 - Process has already been initialized and run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

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.12

PROCESS STATE TRANSITIONS



January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.13

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

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.14

XV6 KERNEL DATA STRUCTURES

- xv6: pedagogical implementation of Linux
- Simplified structures

```

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

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

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.15

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 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.16

LINUX: STRUCTURES

- struct task_struct, equivalent to struct proc
 - Provides process description
 - Large: 10,000+ bytes
 - /usr/src/linux-headers-{kernel version}/include/linux/sched.h
 - 1227 - 1587
- Struct thread_info, provides "context"
 - thread_info.h is at:
 - /usr/src/linux-headers-{kernel version}/arch/x86/include/asm/

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.17

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;
};
    
```

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.18

LINUX STRUCTURES - 2

- List of Linux data structures:
<http://www.tldp.org/LDP/tlk/ds/ds.html>
- Description of process data structures:
<http://www.makelinux.net/books/lkd2/ch03lev1sec1>
2nd edition is online (dated from 2005):
Linux Kernel Development, 2nd edition
Robert Love
Sams Publishing

January 9, 2017

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.19

OBJECTIVES

- Process API – Ch. 5
- Limited Direct Execution – Ch. 6

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.20

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 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.21

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;
}
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.22

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>
```

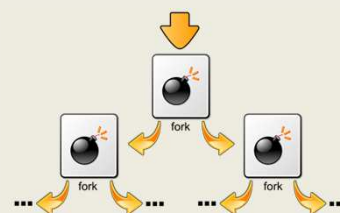
- CPU scheduler determines which to run first

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.23

:(){ :|: & }::



January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.24

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



January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.25

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;
}
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.26

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>
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.27

FORK EXAMPLE

- Linux example

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.28

exec()

- Supports running an external program
- 6 types: execl(), execlp(), execl(), execvp(), execvp(), execvpe())
- execl(), execlp(), execl(): const char *arg
List of pointers (terminated by null pointer)
to strings provided as arguments... (arg0, arg1, .. argn)
- Execvp(), execvp(), execvpe()
Array of pointers to strings as arguments
Strings are null-terminated
First argument is name of file being executed

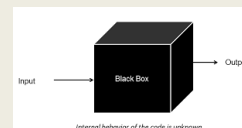
January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.29

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 "black box"
- We don't want to know what is inside... 😊



January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.30

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
        -
    } else { // parent goes down this path (main)
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    }
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
    return 0;
}
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.31

EXEC EXAMPLE - 2

```
-
execvp(myargs[0], myargs); // runs word count
printf("this shouldn't print out");
} else {
    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>
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.32

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);
        -
    } else { // parent goes down this path (main)
        execvp(myargs[0], myargs);
    }
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
    return 0;
}
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.33

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
```

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.34

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 {
    int wc = wait(NULL);
}
return 0;
}
```

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

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.35

LIMITED DIRECT EXECUTION



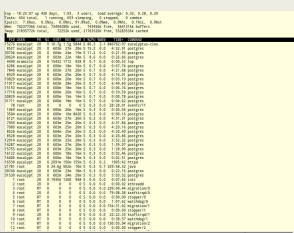
January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.36

VIRTUALIZING THE CPU

- How does the CPU support running so many jobs simultaneously?
- **Time Sharing**
- Tradeoffs:
 - Performance
 - Excessive overhead
 - Control
 - Fairness
 - Security
- Both HW and OS support is used



January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.37

DIRECT EXECUTION

- What if programs could directly control the CPU / system?

OS	Program
1. Create entry for process list	
2. Allocate memory for	
6. Execute call main()	8. Execute return from main()
9. Free memory of process	
10. Remove from process list	

**Computer BOOT Sequence:
OS with Direct Execution**

January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.38

DIRECT EXECUTION - 2

- With direct execution:

How does the OS stop a program from running, and switch to another to support **time sharing**?

How do programs share disks and perform I/O if they are given direct control? Do they know about each other?

With direct execution, how can dynamic memory structures such as linked lists grow over time?

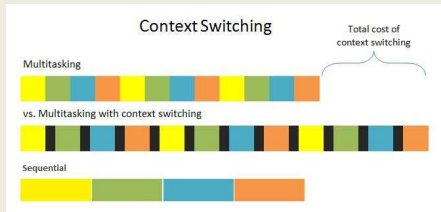
January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.39

CONTROL TRADEOFF

- Too much control:
 - No security
 - No time sharing
- Too little control:
 - Too much OS overhead
 - Poor performance for compute & I/O
 - Complex APIs (system calls), difficult to use

January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.40

CONTEXT SWITCHING OVERHEAD



January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.41

LIMITED DIRECT EXECUTION

- OS implements LDE to support time/resource sharing
- Enabled by **protected (safe) control transfer**
- CPU supported context switch
- Provides data isolation

January 9, 2016
TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma
L2.42

CPU MODES

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

access ← no access

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

January 9, 2016

TCSS422: Operating Systems [Winter 2017]
Institute of Technology, University of Washington - Tacoma

L2.43

QUESTIONS

