TCSS 422: OPERATING SYSTEMS

Processes, Process API, Limited Direct Execution

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

October 3, 2018
TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

---

FEEDBACK FROM 10/1

- How do processes start threads?
  - Done in code or by compiler?
- VM Survey – results submitted
- Assignment 0 questions
- C Tutorial – to be posted

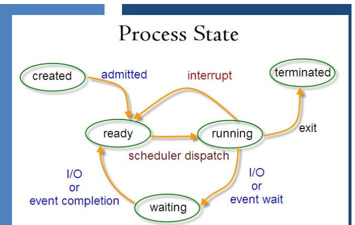
October 3, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L3.2

OBJECTIVES

- Chapter 4 – Processes
- Chapter 5 – Process API
- Chapter 6 – Limited Direct Execution

- Chapter 7 – Introduction to Scheduling
- Chapter 8 – Multi-level Feedback Queue

October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.3



CHAPTER 4: PROCESSES

October 3, 2018 TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma L3.4

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

October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.5

PROCESS

A process is a running program.

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

October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.6

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

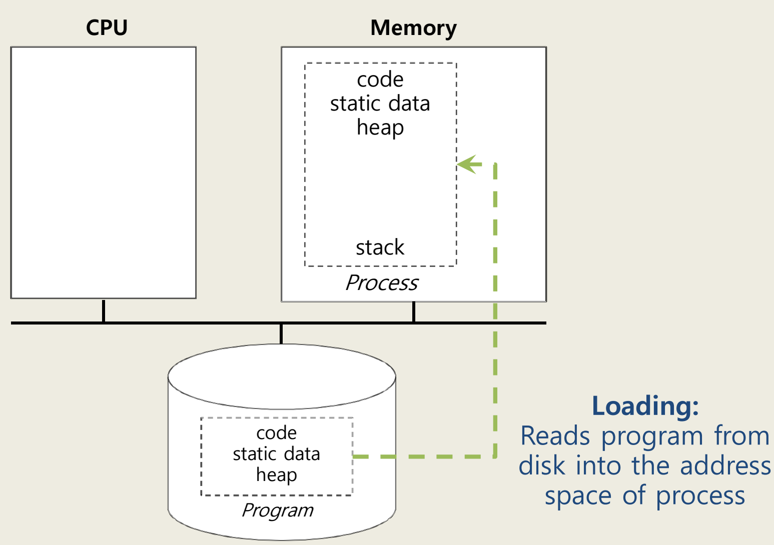
1. Load program code (and static data) into memory
   - Program executable code (binary): loaded from disk
   - Static data: also loaded/created in address space
   - <u>Eager loading</u>: Load entire program before running
   - <u>Lazy loading</u>: Only load what is immediately needed
     - Modern 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

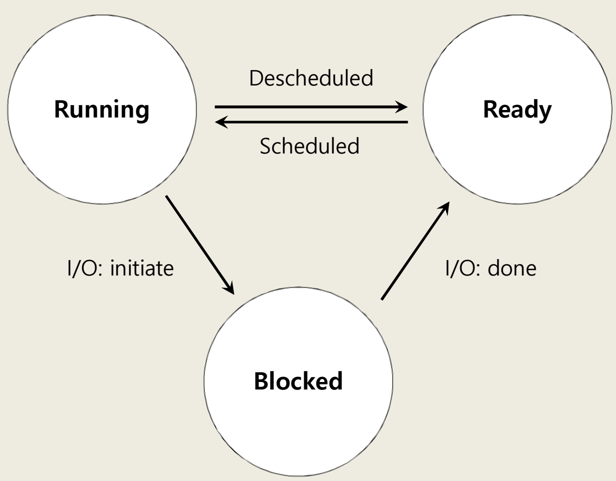
5. Start program running at the entry point: `main()`
   - OS transfers CPU control to the new process



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



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

October 3, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L3.13

XV6 KERNEL DATA STRUCTURES

- Process data structure - textbook: xv6 Pedagogical implementation of Linux

```
// 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
};
```

October 3, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L3.14

XV6 KERNEL DATA STRUCTURES - 2

- CPU register context data structure - textbook: xv6

```
// 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 };
```


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
    - Starts at 1391
- `struct thread_info`, provides "context"
  - thread_info.h is at:

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

LINUX: THREAD_INFO

- Linux thread data structure 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:
  http://www.makelinux.net/books/lkd2/ch03lev1sec1
  2nd edition is online (dated from 2005):

  Linux Kernel Development, 2nd edition
  Robert Love
  Sams Publishing

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

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

CHAPTER 5:
C PROCESS API

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

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

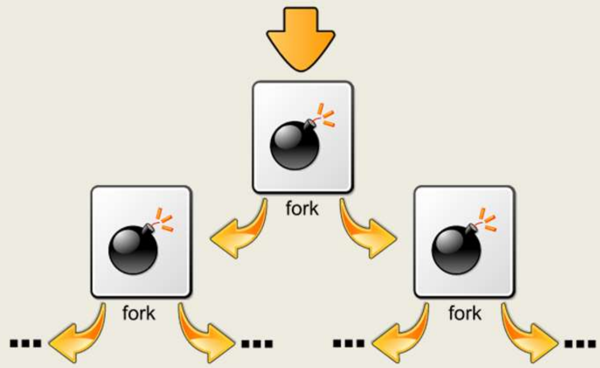
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

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

:(){ :|: & };:



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

exec()

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

- <u>execl(), execlp(), execle(): const char *arg</u>

  Command arguments provided as **<u>LIST of pointers</u>**
  to strings provided as arguments… (arg0, arg1, .. argn)
  (terminated by a null pointer)

- <u>execv(), execvp(), execvpe()</u>
  Command arguments provided as an **<u>ARRAY of pointers</u>** to strings as arguments

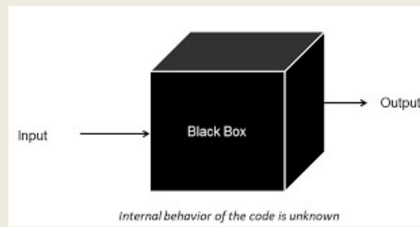
  Strings are null-terminated
  First argument is name of file being executed

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"

- May not want to know what is inside the black box… ☺

- FORTRAN ???



EXEC EXAMPLE

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {                    // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {            // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");          // program: "wc" (word count)
        myargs[1] = strdup("p3.c");        // argument: file to count
        myargs[2] = NULL;                  // marks end of array
        …
```

EXEC EXAMPLE - 2

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

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```


EXEC WITH FILE REDIRECTION (OUTPUT)

```
#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

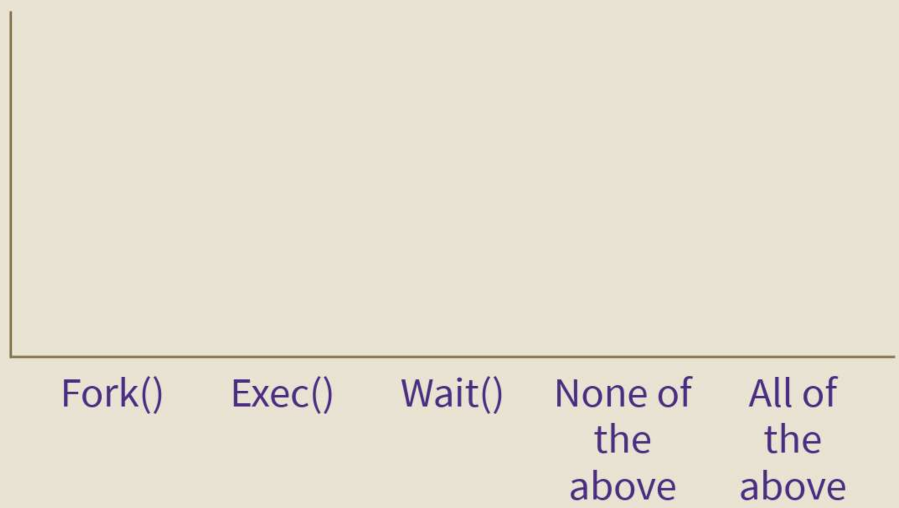
```
S_IRWXU
read, write, execute/search by owner
S_IRUSR
read permission, owner
S_IWUSR
write permission, owner
S_IXUSR
execute/search permission, owner
S_IRWXG
read, write, execute/search by group
S_IRGRP
read permission, group
S_IWGRP
write permission, group
S_IXGRP
execute/search permission, group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


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

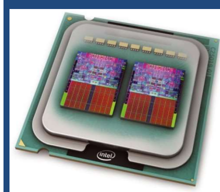
QUESTION: PROCESS API

- 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

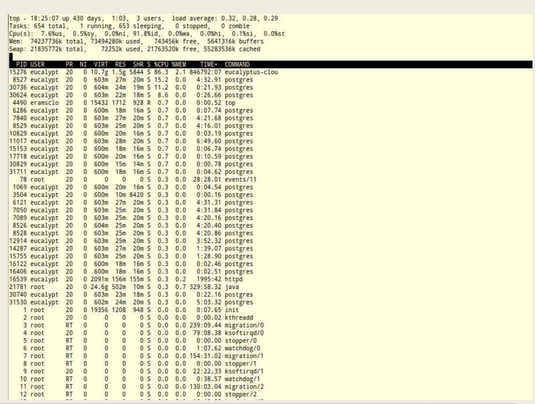
October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.38

CH. 6: LIMITED DIRECT EXECUTION



VIRTUALIZING THE CPU

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



```
top - 18:25:07 up 430 days,  1:03,  3 users,  load average: 0.32, 0.28, 0.29
Tasks: 654 total,   1 running, 653 sleeping,   0 stopped,   0 zombie
Cpu(s):  7.6%us,  0.5%sy,  0.0%ni, 91.8%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  74237736k total, 73494280k used,   743456k free,  5641316k buffers
Swap: 21835772k total,    72252k used, 21763520k free, 55283536k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
15276 eucalypt  20   0 10.7g 1.5g 5844 S 86.3  2.1 846792:07 eucalyptus-clou
 8527 eucalypt  20   0  603m  27m  20m S 15.2  0.0   4:32.91 postgres
30736 eucalypt  20   0  604m  24m  19m S 11.2  0.0   0:21.93 postgres
30624 eucalypt  20   0  603m  22m  18m S  8.6  0.0   0:26.66 postgres
 4490 eramsclo  20   0 15432 1712  928 R  0.7  0.0   0:00.52 top
 6286 eucalypt  20   0  600m  18m  16m S  0.7  0.0   0:07.74 postgres
 7840 eucalypt  20   0  603m  27m  20m S  0.7  0.0   4:21.68 postgres
 8529 eucalypt  20   0  603m  25m  20m S  0.7  0.0   4:16.01 postgres
10829 eucalypt  20   0  600m  20m  16m S  0.7  0.0   0:03.19 postgres
11017 eucalypt  20   0  603m  28m  20m S  0.7  0.0   6:49.60 postgres
15153 eucalypt  20   0  600m  18m  16m S  0.7  0.0   0:06.74 postgres
17718 eucalypt  20   0  600m  20m  16m S  0.7  0.0   0:10.59 postgres
30829 eucalypt  20   0  600m  15m  14m S  0.7  0.0   0:00.78 postgres
31711 eucalypt  20   0  600m  18m  16m S  0.7  0.0   0:04.62 postgres
   78 root      20   0     0    0    0 S  0.3  0.0  28:28.01 events/11
 1069 eucalypt  20   0  600m  20m  16m S  0.3  0.0   0:04.54 postgres
 3504 eucalypt  20   0  600m  10m 8420 S  0.3  0.0   0:00.16 postgres
 6121 eucalypt  20   0  603m  27m  20m S  0.3  0.0   4:31.31 postgres
 7050 eucalypt  20   0  603m  25m  20m S  0.3  0.0   4:31.84 postgres
 7089 eucalypt  20   0  603m  25m  20m S  0.3  0.0   4:20.16 postgres
 8526 eucalypt  20   0  604m  25m  20m S  0.3  0.0   4:20.40 postgres
 8528 eucalypt  20   0  603m  25m  20m S  0.3  0.0   4:20.86 postgres
12914 eucalypt  20   0  603m  25m  20m S  0.3  0.0   3:52.32 postgres
14287 eucalypt  20   0  603m  27m  20m S  0.3  0.0   1:39.07 postgres
15755 eucalypt  20   0  603m  25m  20m S  0.3  0.0   1:28.90 postgres
16122 eucalypt  20   0  600m  18m  16m S  0.3  0.0   0:02.46 postgres
16406 eucalypt  20   0  600m  18m  16m S  0.3  0.0   0:02.51 postgres
16539 eucalypt  20   0 2091m 156m 155m S  0.3  0.2   1995:42 httpd
21781 root      20   0 24.6g 502m  10m S  0.3  0.7 329:58.32 java
30740 eucalypt  20   0  603m  23m  18m S  0.3  0.0   0:22.16 postgres
31530 eucalypt  20   0  602m  24m  20m S  0.3  0.0   5:03.32 postgres
    1 root      20   0 19356 1208  948 S  0.0  0.0   0:07.65 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.02 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0 239:09.44 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0  79:08.38 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/0
    6 root      RT   0     0    0    0 S  0.0  0.0   1:07.62 watchdog/0
    7 root      RT   0     0    0    0 S  0.0  0.0 154:31.02 migration/1
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/1
    9 root      20   0     0    0    0 S  0.0  0.0  22:22.33 ksoftirqd/1
   10 root      RT   0     0    0    0 S  0.0  0.0   0:38.57 watchdog/1
   11 root      RT   0     0    0    0 S  0.0  0.0 130:03.04 migration/2
   12 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/2
```

COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

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

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	 7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

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

OS	Program
1. Create entry for process list 2. Allocate memory for `argv` 5. Clear registers 6. Execute call `main()`	 7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything
and would "just be a library"

DIRECT EXECUTION - 2

- **With direct execution:**

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

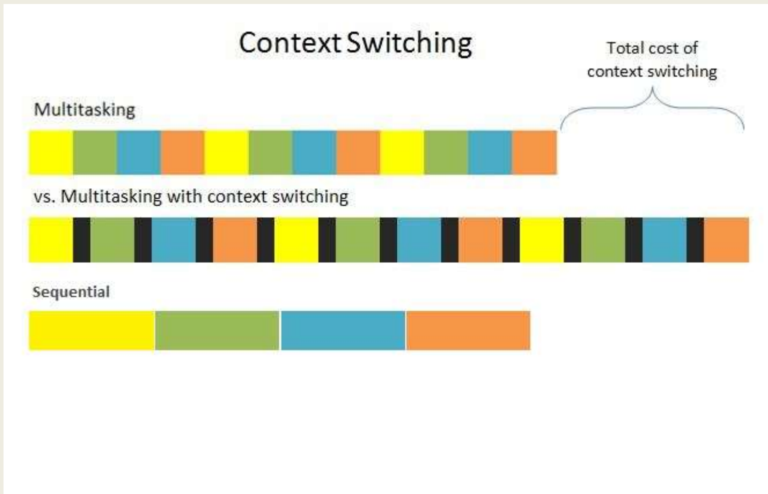
How do programs share disks and perform I/O if they are given direct control? Do they know about each other?

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

CONTROL TRADEOFF

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

CONTEXT SWITCHING OVERHEAD



LIMITED DIRECT EXECUTION

- OS implements LDE to support time/resource sharing
- Limited direct execution means "only limited" processes can execute DIRECTLY on the CPU in ***trusted*** mode
- TRUSTED means the process is trusted, and it can do anything… (e.g. it is a system / kernel level process)
- Enabled by ***protected (safe) control transfer***
- CPU supported context switch
- Provides data isolation

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

October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.47

CPU MODES

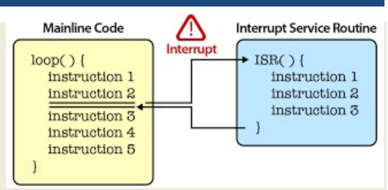
- <u>User mode: ring 3 - untrusted</u>
  - Some instructions and registers are disabled by the CPU
  - Exception registers
  - HALT instruction
  - MMU instructions
  - OS memory access
  - I/O device access
- <u>Kernel mode: ring 0 – trusted</u>
  - All instructions and registers enabled

October 3, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L3.48

SYSTEM CALLS

- Implement restricted "OS" operations
- Kernel exposes key functions through an API:
  - Device I/O (e.g. file I/O)
  - Task swapping: context switching between processes
  - Memory management/allocation: malloc()
  - Creating/destroying processes

TRAPS: SYSTEM CALLS, EXCEPTIONS, INTERRUPTS



- Trap: any transfer to kernel mode
- Three kinds of traps
  - **System call:** (planned) user → kernel
    - SYSCALL for I/O, etc.
  - **Exception:** (error) user → kernel
    - Div by zero, page fault, page protection error
  - **Interrupt:** (event) user → kernel
    - Non-maskable vs. maskable
    - Keyboard event, network packet arrival, timer ticks
    - Memory parity error (ECC), hard drive failure

EXCEPTION TYPES

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violation	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instruction	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunction	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

OS @ boot (kernel mode) | **Hardware**

initialize trap table

→ remember address of … syscall handler

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

Create entry for process list
Allocate memory for program
Load program into memory
Setup user stack with argv
Fill kernel stack with reg/PC
return-from -trap

→ restore regs from kernel stack
move to user mode
jump to main

→ Run main()
…
Call system
trap into OS

→ save regs to kernel stack
move to kernel mode
jump to trap handler

Handle trap
Do work of syscall
return-from-trap

→ restore regs from kernel stack
move to user mode
jump to PC after trap

→ …
return from main
trap (via `exit()`)

Free memory of process
Remove from process list

OS @ boot (kernel mode) | Hardware

initialize trap table

remember address of ...
syscall handler

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

Create entry for process list
Allocate memory for program
Load program into memory
Setup user stack with argv
Fill kernel stack with reg/PC

Computer BOOT Sequence: OS with Limited Direct Execution

move to kernel mode
jump to trap handler

Handle trap
Do work of syscall
return-from-trap

restore regs from kernel stack
move to user mode
jump to PC after trap

...
return from main
trap (via `exit()`)

Free memory of process
Remove from process list

MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- Cooperative multitasking (mostly pre 32-bit)
  - < Windows 95, Mac OSX
  - Opportunistic: running programs must give up control
    - User programs must call a special **yield** system call
    - When performing I/O
    - Illegal operations
  - (POLLEV)
    What problems could you for see with this approach?

QUESTIONS