TCSS 422: OPERATING SYSTEMS

Three Easy Pieces, Processes, Process API

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

January 8, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Feedback from 1/3
- Chapter 2: Operating Systems – Three Easy Pieces
  - √ Easy piece #1: CPU Virtualization
  - √ Easy piece #2: Memory Virtualization
  - Easy piece #3: I/O Virtualization
  - Operating system design goals
- Processes – Ch. 4
- C Linux Process API – Ch. 5
- Limited Direct Execution – Ch. 6 (*wed*)
  - Virtualizing the CPU

January 8, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L2.2

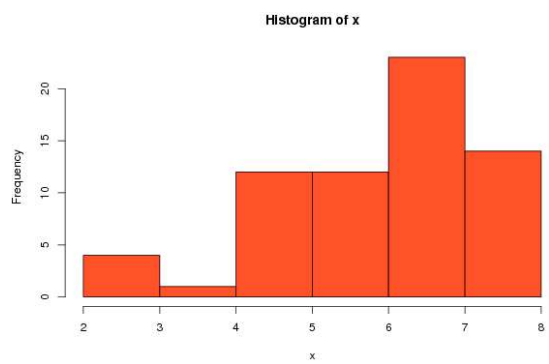
VIRTUAL MACHINE SURVEY

- Please complete the Virtual Machine Survey is wanting an Institute of Technology hosted Ubuntu 16.04 VM
- https://goo.gl/forms/XAxxuZtut5o707Ip1

QUIZ 0 – C REVIEW

Descriptive Statistics	
mean	6.34848484848485
standard deviation	1.45170906433046

6.348	79.36%	avg
2.000	25.00%	min
8.000	100.00%	max
7.000	87.50%	mode



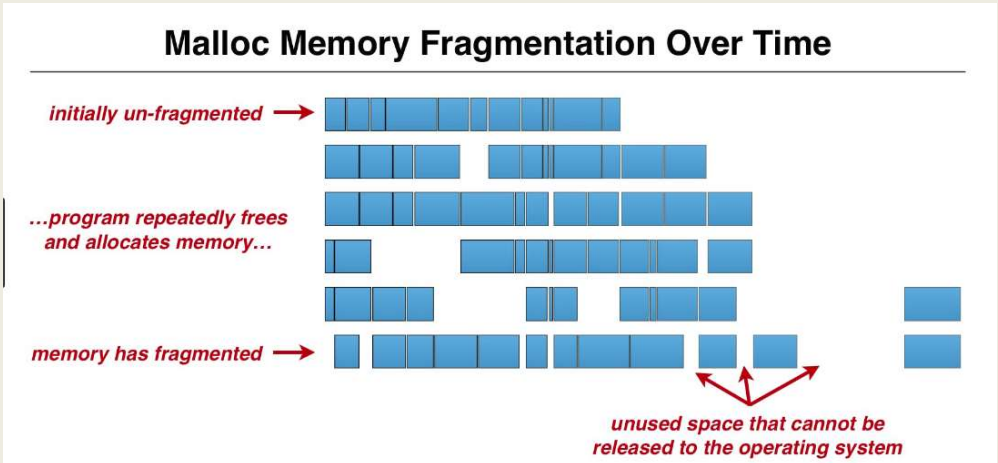
New Window | Postscript

SELECTED FEEDBACK FROM 1/3

- What causes threads to interrupt each other?
  - The OS interrupts threads when:
    - Threads YIELD (e.g. give up) the CPU because they're waiting on I/O (disk or network)
    - They exceed their "time slice" of execution (e.g. 10ms)
- Is kernel-mode code written in a unified language, or does it vary from OS-to-OS?
  - In Linux, kernel code is written in C and Assembly Language

FEEDBACK - 2

- What is "memory" fragmentation?



FEEDBACK - 3

- Are the resources given to a program static?
  - NO
- Can programs request more memory?
  - YES, In C, dynamic memory is allocated on the "HEAP" using the malloc() command
- What is a device driver?
  - A software program written in C and/or assembly language that facilitates transferring data to/from a device (e.g. Disk or Network Card) to the Computer
  - Block vs. char drivers, see: http://www.linuxjournal.com/article/2890

FEEDBACK - 4

- Are we allowed to publicly post source code for assignments later on in another term, or after graduation?
- Or should source code for programs be kept private indefinitely?
- During the term it could be grounds for an honor code violation if code pops up elsewhere
- FREE SPEECH – you're always "allowed" to post code
- However, it is discouraged…

CH. 2: INTRODUCTION TO OPERATING SYSTEMS

January 8, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L2.9

VIRTUALIZATION

- Operating systems present **physical resources** as **virtual representations** to the programs sharing them
  - Physical resources: CPU, memory, disk …
- The virtual form is "**abstract**"
- The OS presents an illusion that each user program runs in isolation on its own hardware
- This virtual form is general, powerful, and easy-to-use

January 8, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L2.10

ABSTRACTIONS

- What form of abstraction does the OS provide?
  - **CPU**
    - Processes and threads
  - **Memory**
    - Address space
    - → large array of bytes
    - All programs see the same "size" of RAM
  - **Disk**
    - Files

WHY ABSTRACTION?

- Allow applications to reuse common facilities
- Make different devices look the same
  - Easier to write common code to use devices
    - Linux/Unix Block Devices
- Provide higher level abstractions
- More useful functionality

ABSTRACTION CHALLENGES

- What is the right "level" of abstraction?
  - How much of the underlying hardware should be exposed?
    - What if **too much**?
    - What if **too little**?
- What are the correct abstractions?
  - Security concerns

To perform parallel work, a single process may:

Launch multiple threads to execute code in parallel while sharing global data in memory

Launch multiple processes to execute code in parallel without sharing global data in memory

Both A and B

None of the above

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

Total Results

PERSISTENCE

- DRAM: Dynamic Random Access Memory: DIMMs/SIMMs
  - Stores data while power is present
  - When power is lost, data is lost (*volatile*)
- Operating System helps "persist" data more **permanently**
  - I/O device(s): hard disk drive (HDD), solid state drive (SSD)
  - File system(s): "catalog" data for storage and retrieval

PERSISTENCE - 2

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
        int fd = open("/tmp/file", O_WRONLY | O_CREAT
                        | O_TRUNC, S_IRWXU);
        assert(fd > -1);
        int rc = write(fd, "hello world\n", 13);
        assert(rc == 13);
        close(fd);
        return 0;
}
```

- open(), write(), close(): OS system calls for device I/O
- Note: man page for open(), write() require page number: "man 2 open", "man 2 write", "man close"

PERSISTENCE - 3

- To write to disk, OS must:
  - Determine where on disk data should reside
  - Perform sys calls to perform I/O:
    - Read/write to file system (*inode record*)
    - Read/write data to file
- Provide fault tolerance for system crashes
  - Journaling: Record disk operations in a journal for replay
  - Copy-on-write: *see ZFS*
  - Carefully order writes on disk

SUMMARY: OPERATING SYSTEM DESIGN GOALS

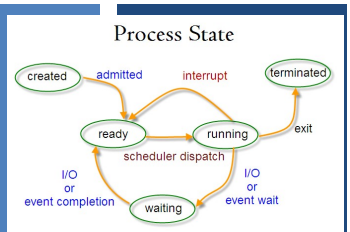
- **ABSTRACTING THE HARDWARE**
  - Makes programming code easier to write
  - Automate sharing resources – save programmer burden
- **PROVIDE HIGH PERFORMANCE**
  - Minimize overhead from OS abstraction (Virtualization of CPU, RAM, I/O)
  - Share resources fairly
  - Attempt to tradeoff performance vs.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

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



CHAPTER 4: PROCESSES

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

PROCESS

A process is a running program.

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

PROCESS API

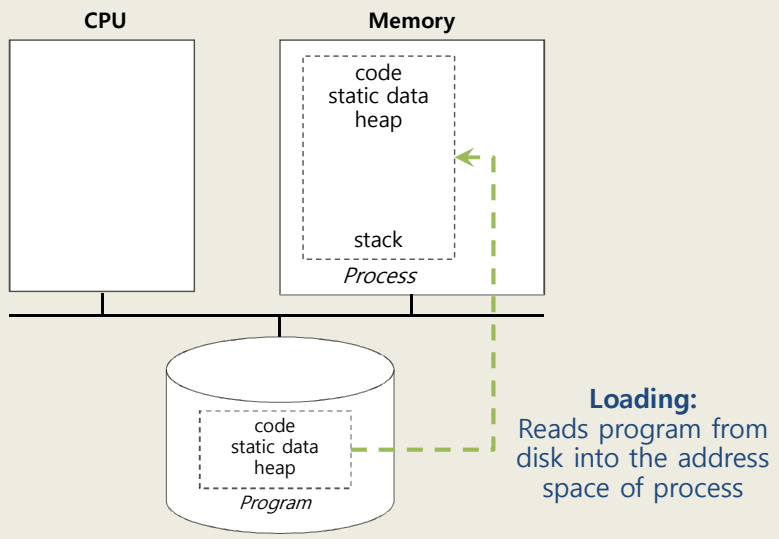
- Modern OSes provide a Process API for process support
- Create
  - Create a new process
- Destroy
  - Terminate a process (ctrl-c)
- Wait
  - Wait for a process to complete/stop
- Miscellaneous Control
  - Suspend process (ctrl-z)
  - Resume process (fg, bg)
- Status
  - Obtain process statistics: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

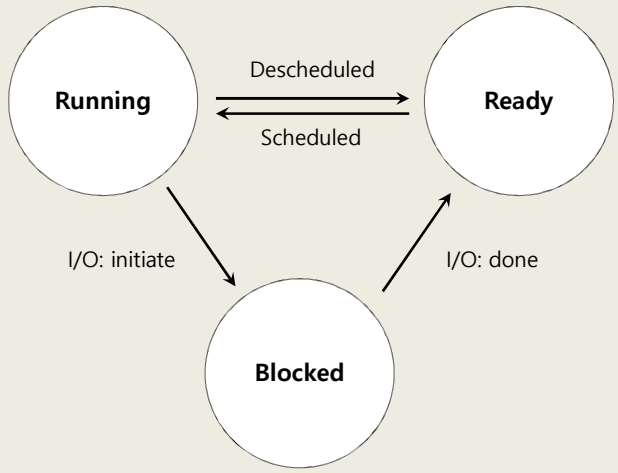


Loading:
Reads program from disk into the address space of process

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

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 };
```

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
  - 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/

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:
  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

RUNNING
READY
BLOCKED
All of the above
None of the above

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

Total Results

CHAPTER 5: C PROCESS API

January 8, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L2.36

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

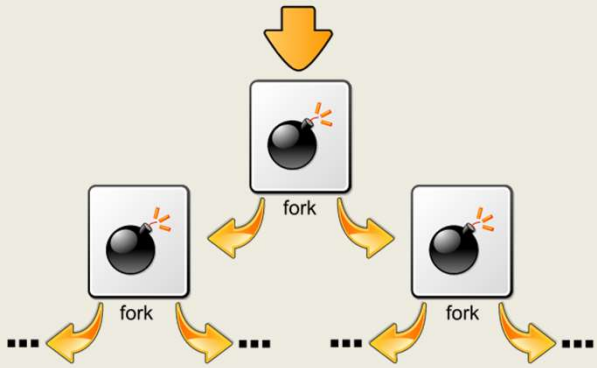
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

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()

- execl(), execlp(), execle(): const char *arg

  List of pointers (terminated by null pointer)
  to strings provided as arguments… (arg0, arg1, .. argn)

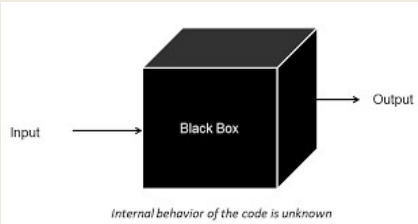
- Execv(), execvp(), execvpe()
  Array of pointers to strings as arguments

  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"

- We don't want to know what is inside… ☺



QUESTIONS