CSI500/400 Operating Systems

Prof. Seth Chaiken

Aug. 25, 2014
Organization

First Homeworks

500/400 Subject

Addressable Memory and a Complete program that our OS+HW system can run

Lecture 02
   First Homeworks
   Some Course Emphases

What’s a Process?

Lecture 03

Lecture 04 (under construction

More About This Course
GET A LAB ACCOUNT!

(Unless you have one from a previous CS course or CS dept project)
FILL OUT THE WEB FORM:
“Flipped classes”

▶ Pre-class prep homework: Focussed readings & Google forms due BEFORE a topic is discussed in class or practiced in lab. (Maybe some videos!)

▶ Traditional lectures are ineffective. We do 3 “Lecture meetings”/week to question, discuss, demonstrate; NOT to hear something first time.

▶ Web site: http://www.cs.albany.edu/~sdc/CSI500
First Homework HW1-reading for next class

First Homework HW1-reading for next class..due Wed.
Sept 3

Part 2: Skim Ch. 1 of ULK and Ch. 1 of MOS4. Write one useful and accurate sentence (preferably in your own words) about each concept below. By the end of the course, successful members will be able to explain technically how an OS implements or uses each one, and account for the behavior of a working computer system in terms of them.

shell, kernel or supervisor mode, user mode, architecture, multiplexing, spooling, timesharing, CPU, registers, program counter, stack pointer, PSW, memory hierarchy, RAM, virtual memory, MMU, context switch, device driver, busy waiting, interrupt, interrupt vector, process, address space or core image, process table, command interpreter or shell (Was your first sentence about shell accurate?), path name, root directory (or folder), working directory, file descriptor, rwx bits, shell prompt, system call, TRAP (like int 0x80) instruction, fork, exec.
Our Subject

▫ “Operating System” = Software to help “users” run “application software” (or “apps”) on computer hardware.
▫ Operating System (for 500/400) = the kernel.
   ▪ The kernel is copied into addressable memory by the bootloader.
   ▪ It stays in memory until power off, reset or reboot.
▫ The kernel PLUS the hardware form the integrated system which CSI500/400 graduates will eventually be able to customize and answer technical questions about.
The 500/400 Subject System

- Hardware (CSI333+404)
- Kernel or supervisor mode system software. 500/400
  - User mode system software (CSI402)
  - User mode application software (most CS courses)
Example of what our 500/400 HW+kernel system system does for our user

The user (person, like me) writes a program in assembly language.
/*This program prints 7 on the standard output*/
#include <asm/unistd.h>
.comm buff,2 /*Prepare a 2-byte buffer in addressable memory for the 2 characters we will write*/
Addressable MEMORY is CRITICAL for a systems person to know about but it is (on purpose) HIDDEN from Java, Web, Python/Perl, and most other high level application development systems today.

The C language exposes addressable memory.

```c
char buff[2]; /*Allocates 2 bytes of memory*/
/*We just declared a length 2 char array VARIABLE*/
/*We now DECLARE AND DEFINE an int VARIABLE*/
int IVar = 3; /*Allocates typically 4 bytes.*/
/* ... */
printf("%p", &(buff[0]));
/*This will print the ADDRESS OF the first byte in the two byte buffer in memory.*/
printf("%p", &(buff[1])); /*The second byte..*/
printf("%p", Ivar ); /*Prints 0x3, the VALUE OF IVar.*/
printf("%p", &IVar ); /*Prints the ADDRESS OF IVar*/
```
.section .text
.globl _start
_start:
    movb $7,%al
    addb '$0',%al
    movb %al,buff
    movb '$\n',buff+1
    movl $__NR_write,%eax
    movl $1,%ebx
    movl $buff,%ecx
    movl $2,%edx
    int $0x80
    movl $__NR_exit,%eax
    movl $0,%ebx
    int $0x80
/*Documented with MY PURPOSES for writing each line of code*/

.comm buff,2 /*Prepare a 2-byte buffer in addressable memory for the 2 characters we will write*/
#include <asm/unistd.h>
/*Define the __NR_write & other kernel syscall numbers.*/

.section .text /*What’s assembled below is used for machine instructions*/
.globl _start /*So the linker knows about our symbol spelled _start*/
_start: /*So _start symbolizes the address where the next instruction will be when the program is eventually loaded into addressable memory*/

movb $7,%al /*Instruction to supply integer value 7 for further computing */
addb $'0',%al /*Add the ASCII code for digit 0 to that integer 7 to make the ASCII code for 7*/
movb %al,buff /*Copy that ASCII code into addressable memory because the write syscall takes WHAT TO WRITE from ADDRESSABLE MEMORY*/
movb $'\n', buff+1 /*Copy the ASCII code for newline next to the code for '7'*/

movl $__NR_write, %eax
    /*Supply the syscall number for writing*/
movl $1, %ebx  /*STANDARD OUTPUT is file descriptor 1. Supply the file descriptor describing where to write*/
movl $buff, %ecx
    /*Supply the address locating WHAT to write*/
movl $2, %edx  /*Supply the count of HOW MANY chars or bytes to write*/
int $0x80  /*Ask the OS kernel to write; print the 7 on a line by itself*/
movl $__NR_exit,%eax
    /*Supply the syscall number to exit cleanly*/
movl $0,%ebx
    /*Supply the exit code that means no errors*/
int $0x80
    /*Ask the OS kernel to end our process*/
Lecture 02
GET A LAB ACCOUNT!

(Unless you have one from a previous CS course or CS dept project)
FILL OUT THE WEB FORM:
MOS4: 2.1 Processes and begin 2.2 (through 2.2.3)
Part 2: Skim Ch. 1 of ULK and Ch. 1 of MOS4. Write one useful and accurate sentence (preferably in your own words) about each concept below. By the end of the course, successful members will be able to explain technically how an OS implements or uses each one, and account for the behavior of a working computer system in terms of them.

shell, kernel or supervisor mode, user mode, architecture, multiplexing, spooling, timesharing, CPU, registers, program counter, stack pointer, PSW, memory hierarchy, RAM, virtual memory, MMU, context switch, device driver, busy waiting, interrupt, interrupt vector, process, address space or core image, process table, command interpreter or shell (Was your first sentence about shell accurate?), path name, root directory (or folder), working directory, file descriptor, rwx bits, shell prompt, system call, TRAP (like int 0x80) instruction, fork, exec.
CSI500/400 Emphasis

Day-to-day course/homework/lab/project content

c-programming, OS principles, Linux kernel internals as our example of the organization of a large, modern system.

Introduce practices used in

- Free/Open Source development.
  - GIT

- Embedded Systems
  - (shell) command line computer usage by programmers
  - Cross-development: Write and compile software on one (big, fast, desktop or server) system and run it on another (embedded) system.
What our hardware+kernel system does

It creates, runs (and destroys) processes
UA ICSI500/400 is about how a modern computer system (OS+hardware) runs application software supplied by a user in “binary” form: **machine language**

Assembly language is machine language coded with human-recognizable **symbols** and **mnemomics** (plus some explicit numbers) instead of numeric values and machine instruction codes.
_start:
    movb $7,%al
    addb $'0',%al

Disassembly of section .text:

08048074 <_start>:
   8048074:  b0 07       mov  $0x7,%al
   8048076:  04 30       add  $0x30,%al

| | immediate operand (data to copy or add)
| |--- opcode (move or add) and which register
|---address where each instruction will be located in memory

ADDRESS:  (instr. in hex.) (same instr. in asm.)
Note: mov is 2 bytes long, so the address of the
add instruction is 8048074+2
Some ATT/Gnu x386 (gas) language rules

- The destination is on the RIGHT: `movb $7,%al`
- `$anything` means the operand VALUE is immediately here. An actual 7 is copied to a register above.
- `% means the operand is in a REGISTER.
- Plain number operands: A NUMBER like 0x80490a8 in `movb %al,0x80490a8` or a SYMBOL like `buff` SYMBOLIZING a MEMORY ADDRESS NUMBER as in `movb %al,buff` means that MEMORY is addressed, and the operand is IN MEMORY, at the LOCATION (ADDRESS) located by that ADDRESS NUMBER.

In `movb %al,0x80490a8` copies one byte FROM register %al INTO MEMORY.

`movb buff,%al` does the reverse: Load a value from memory and copy it into register %al.
What’s important for 500/400

Instructions just like data are stored bit-by-bit in addressable memory. Assembly and disassembly listings show the instructions in hexadecimal with the addresses where they are at also in hexadecimal.

The CPU (effectively) does the instructions in the numeric order they are stored in memory unless modified by branches, jumps, calls, etc. That is called **program order**. Conditional branches cause program order to be determined by often unpredictable run-time data.

Unlike MIPS (taught in CSI333), x86 instructions are variable length and their encodings are very complicated and messy. If you need those details, consult the Intel or AMD architecture manuals, plus a manual on GNU/AT&T x86 assembly language.
.section .text
.globl _start
_start:
    movb $7,%al
    addb $’0’,%al
    movb %al,buff
    movb $’\n’,buff+1
    movl $__NR_write,%eax
    movl $1,%ebx
    movl $buff,%ecx
    movl $2,%edx
    int $0x80
    movl $__NR_exit,%eax
    movl $0,%ebx
    int $0x80
/*Documented with MY PURPOSES for writing each line of code*/
.comm buff,2 /*Prepare a 2-byte buffer in addressable memory for the 2 characters we will write*/
#include <asm/unistd.h>
/*Define the __NR_write & other kernel syscall numbers.*/

.section .text /*What’s assembled below is used for machine instructions*/
.globl _start /*So the linker knows about our symbol spelled _start*/
_start: /*So _start symbolizes the address where the next instruction will be when the program is eventually loaded into addressable memory*/

movb $7,%al /*Instruction to supply integer value 7 for further computing */
addb $’0’,%al /*Add the ASCII code for digit 0 to that integer 7 to make the ASCII code for 7*/

movb %al,buff /*Copy that ASCII code into addressable memory because the write syscall takes WHAT TO WRITE from ADDRESSABLE MEMORY*/
movb $'\n', buff+1 /*Copy the ASCII code for newline next to the code for '7'*/

movl __NR_write, %eax
    /*Supply the syscall number for writing*/
movl $1, %ebx  /*STANDARD OUTPUT is file descriptor 1. Supply the file descriptor describing where to write*/
movl $buff, %ecx
    /*Supply the address locating WHAT to write*/
movl $2, %edx  /*Supply the count of HOW MANY chars or bytes to write*/
int $0x80    /*Ask the OS kernel to write; print the 7 on a line by itself*/
The int $0x80A$ instruction transfers control of the CPU to the operating system kernel code.

When the CPU executes int (interrupt) instructions or syscall instructions, the CPU switches its MODE from user mode to KERNEL or SUPERVISOR mode.

VERY IMPORTANT TO RECOGNIZE: A protection boundary is crossed when this happens!

What this course will address: Some details of how the kernel software makes the computer copy those two bytes from user to kernel space and then to the I/O device to make the output happen.
shell prompt $ ./print7
7
shell prompt $
shell prompt $ ./print7
7
shell prompt $

1. (shell) process printed “.. $” and WAITED (BLOCKED).
2. CONCURRENTLY I typed ./print7ENTER-KEY
3. The OS switched the shell process from BLOCKED to RUNNING.
4. The shell program analyzed my input (../print7) and called SYSTEM CALLS fork() to make a second process.
5. The second process did SYSTEM CALL exec() to make itself load into VM and run my program (print7).
6. The second PROCESS ran the print7 PROGRAM.
7. Besides CPU instructions, that PROCESS ran (1) the Linux write system call and (2) the exit system call.
8. The OS destroyed the print7 process and made the first (shell) process RUN again.
9. The shell printed “shell prompt $ ” and BLOCKED again, waiting for more input from me.
Prove some of that please?

```shell
shell prompt $ ./print7 #command shell to run my program
7               #print7 process printed 7
NEW-LINE
shell prompt $              #shell waits BLOCKED for input
shell prompt $               #I typed ENTER a few times.
shell prompt $
shell prompt $ strace ./print7 #I typed a command
execve("./print7", ["./print7"], [/* 40 vars */]) = 0
write(1, "7\n", 27<==THIS IS THE 7\n that print7 printed!)            = 2
_exit(0)                     = ?
Process 7896 detached
shell prompt $
```

Evidence of TWO processes: tracing process output INTERLEAVED WITH print7 output!
Dramatization of two CPUs (or) cores, where one is time-shared between two-three processes:
Person one continually wiggles his left-hand fingers.
Person two, the prof, wiggles his left-hand fingers. When another student “Run wiggle!”, the prof. stops wiggling, he talks and then starts wiggling his right-hand fingers. So two hands are wiggling and one is still, not wiggling.
When another student shouts “Dong!” simulating the scheduler’s clock tick interrupt, the prof.:
1. Holds both hands still. (Meanwhile, person one still wiggles his hand.)
2. Talks about the scheduler choosing to give his other hand a chance.
3. Says the scheduler chooses the right hand.
4. Wiggles his right hand, keeping the left still.
5. When other students shout “Dong!” again, the prof. stops both hands, says “the scheduler chooses again” and resumes wiggling his left hand, keeping the right hand still.
The three famous process states MOS4 p93

**RUNNING**  One of the CPUs or “Cores” is ACTUALLY RUNNING machine code stored in the process’ VIRTUAL ADDRESS SPACE (or MEMORY) (or kernal code belonging to a system call that the process made.)

**BLOCKED**  Unable to run until some event external to the process happens, like a person finishing keyboard input by pressing the ENTER-KEY, or a disk drive’s arm settling into place, reading the magnetized spots, and the controller transferring data to physical memory.

**READY**  Runnable; temporarily stopped to let another process run. The SCHEDULER (part of the OS) chooses WHICH OF (MANY) READY processes to make RUNNING.
Lecture 03
Implementation of Processes

ULK: “From the kernel’s point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.” CPU time is shared (or sliced or multiplexed) by the SCHEDULER, an OS software component.

Software Engineering: “Entities” are real or simulated world things for which software will keep track of, maintain information about, operate on, etc.

HOW? The engineer designs one or more data structures or class definitions. WHEN THE SOFTWARE RUNS, one INSTANCE of these structures is LAID OUT IN MEMORY. The DATA in that MEMORY area keeps track of the “Entity”. For example,

```java
class Employee {
    String name;
    int salary;  // in cents, to avoid roundoff errors.
    Employee supervisor;
    Department department;
    // Methods
    public giveRaise( int raise )
    {
        this.salary = this.salary + raise;
    }
}
```
Process Descriptor or Linux task-struct

Names for the same concept:

- Process Table ENTRY.
- Process Descriptor.
- In Linux, struct task-struct defined in include/linux/scheduler.h

IT IS A STRUCTURE i.e. OBJECT, in the memory used by the kernel, one instance for each process.

(conceptually) most important FIELDS

- Process (scheduling) STATE: RUNNING, BLOCKED, READY (In a REAL OS like Linux, it’s different and more complex...)
- Data structures to SAVE the contents of CPU REGISTERS, including the PROGRAM COUNTER and STACK POINTER: for WHEN the process is NOT RUNNING. (When the process is RUNNING, that important data is kept, rapidly changing, in the HARDWARE CPU registers.)
Show the process descriptor from MOS4, pointing out the state field.
What if there are READY processes in addition to the RUNNING processes?
What if whatever some BLOCKED processes are waiting for actually happens?
The CPUs are busy running running processes. Some interrupt occurs. Even if no hardware devices or keyboards are generating interrupts, a TIMER makes interrupts 100-1000 times per second. In interrupt FORCES the CPU to stop running user process code and run KERNEL code instead. It give the KERNEL software a chance to change a RUNNING process to READY, a BLOCKED process to READY, and a READY process to RUNNING.
Process 2: READY->RUNNING and Process 1: RUNNING->READY (or blocked) is called a CONTEXT SWITCH.
Show Process Desc. in MOS4.
Show task structure in Linux
http://lxr.free-electrons.com
First guided treasure hunting:

1. Identifier search for task_struct
   1.1 All but one of the results are lines like struct task_struct;
   1.2 Know to look for it in include/linux/sched.h
Program
(Static) Machine instructions and initial data that might someday be copied into a process and run.

Process
(Dynamic) One single, particular run of a program.

Thread
(Dynamic) One single flow of control, running machine instructions one after the other, within one process.
Beginner programs (like the ones you have seen already in this course) make their processes run only one thread. But multithreading programs are very common now.
What is threading when a thread runs:

```
.section .text
.globl _start
_start: nop
        pushl $printbuf
        ...     .L11
.L8:    popl    %ebx
        popl    %ebp
        ret
     .section .data
printbuf: .ascii  "12345678\n"
```
What is an OS?

1. Operating System
2. Is Software
3. The OS interfaces application software with people, storage and communication media, and exotic devices like robots (through the hardware).

We’ll study HOW OPERATING SYSTEMS WORK

1. What they do.
2. How to use them.
Why study OS?

1. Create better applications, deploy and manage systems better, and fix problems more efficiently.
2. Gain (maybe you’re first) experience with a highly complex computing system. Become able to work with complexity.
3. Learn to think system-wide.
4. Know some technology good for a few years (like 5?).
5. Maybe build device drivers or perhaps even lower-level OS components, perhaps in embedded applications like cell phones and robots.
C is a big part of this course.

Reason ONE

- The interface between popular OS/hardware combinations and application software is the C language and execution model. POSIX says so!
- Unlike Java, C exposes the addressable memory.

Reason TWO

- Most currently active OS are coded mostly in C.
- C with GNU C’s \texttt{asm} extension provides the kind of hardware access that OS code needs. There are very few assembly language files in the Linux kernel.
The Big Picture

- (When system is not idle,) most of the time, the computer hardware executes **machine instructions**.
- The CPU keeps track of the memory address of the current machine instruction in the Program Counter register (PC).
- The CPU keeps track of a small amount of data in a few General Purpose and/or Index Registers.
- The CPU activates RAM to fetch machine instructions.
- Each machine instruction tells the CPU to sometimes activate RAM to fetch data, sometimes fetch and/or change data in (CPU) registers, sometimes activate RAM to store data there. Also the new value for the PC is determined.

This part of **beautiful interface** operates most often.
The address space ranges from 0 to $2^{32} - 1$, which is 4 Gigabinary -1, approximately 4 billion!
Virtualization

OS + hardware system features implements
ONE, SEPARATE, mostly ISOLATED, INDEPENDENT
Virtual Computer
CPU (1 or more with REGISTERS) + Addressable Memory
one FOR EACH PROCESS
currently running a program

Good Idea!
Imagine that ASM program from lab, by itself, in a bare hardware computer!