CS162
Operating Systems and
Systems Programming
Lecture 3

Processes (con’t), Fork,
Introduction to I/O

August 29th, 2018
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: Four fundamental OS concepts

- **Thread**
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack

- **Address Space** with translation
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine

- **Process**
  - An instance of an executing program is a *process consisting of an address space and one or more threads of control*

- **Dual Mode** operation/protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses
Recall: A simple address translation w/ Base & Bound

- Can the program touch OS?
- Can it touch other programs?
Tying it together: Simple B&B: OS loads process

- Processes: Proc 1, Proc 2, ..., Proc n
- OS

- Memory Layout:
  - Code
  - Static Data
  - Heap
  - Stack

- System Mode: 1
- Base: xxxx ...
- Bound: xxxx ...
- uPC: xxxx ...
- PC: xxxx ...
- Regs: ...

- Addressing:
  - Base + Offset = Address
  - Offset = Address - Base

- Memory Locations:
  - 0000 ...
  - 1000 ...
  - 1100 ...
  - 3000 ...
  - 3080 ...
  - FFFF ...
  - 0000 ...
  - 1000 ...
  - 1100 ...
  - 3000 ...
  - 3080 ...
  - FFFF ...

- Registers:

- Instruction Set:
  - Load
  - Store

- Process Execution:
  - Instruction Fetch
  - Instruction Decode
  - Address Calculation
  - Memory Access
  - Register Update

- System Calls:
  - I/O
  - Memory Management
  - File System

- Security:
  - Authentication
  - Authorization
  - Auditing

- Performance:
  - Throughput
  - Latency
  - Resource Utilization

- Reliability:
  - Error Detection
  - Error Correction
  - Fault Tolerance

- Scalability:
  - Resource Allocation
  - Load Balancing
  - Dynamic Configuration

- Portability:
  - Compiler Generation
  - Library Management
  - Interoperability

- Maintainability:
  - Code Analysis
  - Metrics Collection
  - Documentation

- Usability:
  - User Interface
  - Help System
  - Accessibility

- Maintainability:
  - Code Analysis
  - Metrics Collection
  - Documentation

- Usability:
  - User Interface
  - Help System
  - Accessibility
Simple B&B: OS gets ready to execute process

- Privileged Inst: set special registers for Proc 2
- RTU
Simple B&B: User Code Running

- How does kernel switch between processes?
- First question: How to return to system?
3 types of Mode Transfer

• Syscall
  – Process requests a system service, e.g., exit
  – Like a function call, but “outside” the process
  – Does not have the address of the system function to call
  – Marshall the syscall id and args in registers and exec syscall

• Interrupt
  – External asynchronous event triggers context switch
  – e.g., Timer, I/O device
  – Independent of user process

• Trap or Exception
  – Internal synchronous event in process triggers context switch
  – e.g., Protection violation (segmentation fault), Divide by zero, …

• All 3 are an UNPROGRAMMED CONTROL TRANSFER
How do we get the system target address of the “unprogrammed control transfer?”
Example: Interrupt Vector

- Where else do you see this dispatch pattern?
Simple B&B: User => Kernel

• How to return to system?
• How to save registers and set up system stack?
Simple B&B: Switch User Process

- How to save registers and set up system stack?
Simple B&B: “resume”

- How to save registers and set up system stack?
Process Control Block

(Assume single threaded processes for now)

• Kernel represents each process as a process control block (PCB)
  – Status (running, ready, blocked, …)
  – Registers, SP, … (when not running)
  – Process ID (PID), User, Executable, Priority, …
  – Execution time, …
  – Memory space, translation tables, …

• Kernel Scheduler maintains a data structure containing the PCBs

• Scheduling algorithm selects the next one to run
Recall: give the illusion of multiple processors?

• Assume a single processor. How do we provide the illusion of multiple processors?
  – Multiplex in time!
  – Multiple “virtual CPUs”

• Each virtual “CPU” needs a structure to hold, i.e., PCB:
  – Program Counter (PC), Stack Pointer (SP)
  – Registers (Integer, Floating point, others…?)

• How switch from one virtual CPU to the next?
  – Save PC, SP, and registers in current PCB
  – Load PC, SP, and registers from new PCB

• What triggers switch?
  – Timer, voluntary yield, I/O, other things
Simultaneous MultiThreading/Hyperthreading

• Hardware technique
  – Superscalar processors can execute multiple instructions that are independent
  – Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run

• Can schedule each thread as if were separate CPU
  – But, sub-linear speedup!

• Original technique called “Simultaneous Multithreading”
  – SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5
Scheduler

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

```c
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
```
Putting it together: web server

Client

Request

Reply
(retrieved by web server)

Web Server
Putting it together: web server

1. network socket read
   syscall
   2. copy arriving packet (DMA)
      interrupt
      3. kernel copy
      4. parse request
      5. file read
      6. disk request
      7. disk data (DMA)
      8. kernel copy
      9. format reply
      10. network socket write
         syscall
      11. kernel copy from user buffer to network buffer
         RTU
      12. format outgoing packet and DMA
         RTU

Server

Request

Reply

Network interface

Disk interface

Kernel

Hardware
Recall: User/Kernel (Privileged) Mode

User Mode

Kernel Mode

Limited HW access

Full HW access

exec

syscall

interrupt

exception

rtn

rfi

exit
Implementing Safe Kernel Mode Transfers

• Important aspects:
  – Separate kernel stack
  – Controlled transfer into kernel (e.g., syscall table)

• Carefully constructed kernel code packs up the user process state and sets it aside
  – Details depend on the machine architecture

• Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself
Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)
Before

User-level Process

code:
foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers

Kernel
code:
handler() {
pusha
    ...
}

Exception Stack

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

EAX, EBX, ...
During

User-level Process

code:

```c
foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}
```

stack:

```

```

Registers

```

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
```

Kernel

code:

```c
handler() {
    pusha
    ...
}
```

Exception Stack

```

SS
ESP
EFLAGS
CS
EIP
error
```

SS: ESP
User-level Process
Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory
How Does the Kernel Provide Services?

• You said that applications request services from the operating system via syscall, but …
• I’ve been writing all sort of useful applications and I never ever saw a “syscall” !!!

• That’s right.
• It was buried in the programming language runtime library (e.g., libc.a)
• … Layering
OS Run-Time Library

Tasks

OS

Applications

Login

Window Manager

Operating System Library
A Kind of Narrow Waist

Compilers

Web Browsers

Email

Web Servers

Databases

Word Processing

Portable OS Library

System Call Interface

Portable OS Kernel

Platform support, Device Drivers

x86

PowerPC

ARM

Ethernet (1Gbs/10Gbs)

802.11 a/g/n/ac

SCSI

Graphics

Thunderbolt

Application / Service

User

System

Software

Hardware

OS
Hardware support: Interrupt Control

• Interrupt processing not visible to the user process:
  – Occurs between instructions, restarted transparently
  – No change to process state
  – What can be observed even with perfect interrupt processing?

• Interrupt Handler invoked with interrupts ‘disabled’
  – Re-enabled upon completion
  – Non-blocking (run to completion, no waits)
  – Pack up in a queue and pass off to an OS thread for hard work
    » wake up an existing OS thread
Hardware support: Interrupt Control

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    » e.g., kernel segmentation fault
Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled
How do we take interrupts safely?

- **Interrupt vector**
  - Limited number of entry points into kernel
- **Kernel interrupt stack**
  - Handler works regardless of state of user code
- **Interrupt masking**
  - Handler is non-blocking
- **Atomic transfer of control**
  - "Single instruction"-like to change:
    » Program counter
    » Stack pointer
    » Memory protection
    » Kernel/user mode
- **Transparent restartable execution**
  - User program does not know interrupt occurred
Administrivia: Getting started

• **THIS** Friday (8/31) is early drop day! Very hard to drop afterwards…

• Work on Homework 0 **due on Tuesday**!
  – Get familiar with all the cs162 tools
  – Submit to autograder via git

• Participation: Attend section! Get to know your TA!

• Group sign up via autograder then TA form next week
  – Get finding groups of 4 people ASAP
  – Priority for same section; if cannot make this work, keep same TA
5 min break
Can a process create a process?

- Yes! Unique identity of process is the “process ID” (or PID)
- `fork()` system call creates a copy of current process with a new PID
- Return value from `fork()`: integer
  - When > 0:
    » Running in (original) Parent process
    » return value is pid of new child
  - When = 0:
    » Running in new Child process
  - When < 0:
    » Error! Must handle somehow
    » Running in original process

- All state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc…
fork1.c

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

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    size_t readlen, writelen, slen;
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
        exit(1);
    }
    exit(0);
}
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {               /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
}  else if (cpid == 0) {      /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
}
...

Process Races: fork3.c

int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    for (i=0; i<10; i++) {
        printf("[%d] parent: %d\n", mypid, i);
        // sleep(1);
    }
} else if (cpid == 0) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    for (i=0; i>-10; i--) {
        printf("[%d] child: %d\n", mypid, i);
        // sleep(1);
    }
}

• Question: What does this program print?
• Does it change if you add in one of the sleep() statements?
UNIX Process Management

- UNIX **fork** — system call to create a copy of the current process, and start it running
  - No arguments!

- UNIX **exec** — system call to *change the program* being run by the current process

- UNIX **wait** — system call to wait for a process to finish

- UNIX **signal** — system call to send a notification to another process

- UNIX man pages: **fork**(2), **exec**(3), **wait**(2), **signal**(3)
UNIX Process Management

```c
main () {
...
}
```

```c
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```

```c
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```

```c
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```
Shell

• A shell is a job control system
  – Allows programmer to create and manage a set of programs to do some task
  – Windows, MacOS, Linux all have shells

• Example: to compile a C program
  cc –c sourcefile1.c
  cc –c sourcefile2.c
  ln –o program sourcefile1.o sourcefile2.o
  ./program
Signals – infloop.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
    printf("Caught signal %d - phew!\n", signum);
    exit(1);
}

int main()
{
    signal(SIGINT, signal_callback_handler);

    while (1) {}
}
```
Summary

• Process: execution environment with Restricted Rights
  – Address Space with One or More Threads
  – Owns memory (address space)
  – Owns file descriptors, file system context, …
  – Encapsulate one or more threads sharing process resources

• Interrupts
  – Hardware mechanism for regaining control from user
  – Notification that events have occurred
  – User-level equivalent: Signals

• Native control of Process
  – Fork, Exec, Wait, Signal