Recall: Four Fundamental OS Concepts

- **Thread**: Execution Context
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with or w/o translation)
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine
    (in which case programs operate in a virtual address space)
- **Process**: an instance of a running program
  - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" has the ability to access certain resources
  - Combined with translation, isolates programs from each other and the OS from programs

Recall: OS Bottom Line: Run Programs

- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Recall: Protected Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine
Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Recall: Simple address translation with Base and Bound

- Can the program touch OS?
- Can it touch other programs?

Simple B&B: User => Kernel

- How to return to system?

Simple B&B: Interrupt

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

1. How to save registers and set up system stack?

**Simple B&B: “resume”**

1. How to save registers and set up system stack?

---

**Running Many Programs**

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other

- Questions ???
  - How do we represent user processes in the OS?
  - How do we decide which user process to run?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren’t we wasting a lot of memory?

---

**Multiplexing Processes: The Process Control Block**

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …

- Kernel Scheduler maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision

- Give out non-CPU resources
  - Memory/IO
  - Another policy decision
CPU Switch From Process A to Process B

Scheduler

if (readyProcesses(PCBs)) {
    nextPCB = selectProcess(PCBs);
    run(nextPCB);
} else {
    run_idle_process();
}

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

Simultaneous MultiThreading/Hyperthreading

• Hardware scheduling 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


• Up to 28 Cores, 56 Threads
  – 694 mm² die size (estimated)
• Many different instructions
  – Security, Graphics
• Caches on chip:
  – L2: 28 MiB
  – Shared L3: 38.5 MiB (non-inclusive)
  – Directory-based cache coherence
• Network:
  – On-chip Mesh Interconnect
  – Fast off-chip network directly supports 8-chips connected
• DRAM/chips
  – Up to 1.5 TiB
  – DDR4 memory
Is Base and Bound a Good-Enough Protection Mechanism?

- No: Too simplistic for real systems
- Inflexible/Wasteful:
  - Must dedicate physical memory for potential future use
  - (Think stack and heap!)
- Fragmentation:
  - Kernel has to somehow fit whole processes into contiguous block of memory
  - After a while, memory becomes fragmented!
- Sharing:
  - Very hard to share any data between Processes or between Process and Kernel
  - Need to communicate indirectly through the kernel…

Better: x86 – segments and stacks

Processor Registers

Code
Static Data
Heap
Stack

Start address, length and access rights associated with each segment

Better Alternative: Address Mapping

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Translation Map 1

<table>
<thead>
<tr>
<th>Code 2</th>
<th>Data 2</th>
<th>Stack 2</th>
<th>Heap 2</th>
<th>Code 1</th>
<th>Data 1</th>
<th>Stack 1</th>
<th>Heap 1</th>
<th>Code 1</th>
<th>Data 1</th>
<th>Stack 1</th>
<th>Heap 1</th>
</tr>
</thead>
</table>

Translation Map 2

<table>
<thead>
<tr>
<th>OS code</th>
<th>OS data</th>
<th>OS heap &amp; Stacks</th>
</tr>
</thead>
</table>

Administrivia: Getting started!

- Kubiatowicz Office Hours:
  - 1-2pm, Monday & Thursday
- Homework 0 Due Tomorrow!
  - Get familiar with the cs162 tools
  - configure your VM, submit via git
  - Practice finding out information:
    - How to use GDB? How to understand output of unix tools?
    - We don’t assume that you already know everything!
    - Learn to use “man” (command line), “help” (in gdb, etc), google
- Should be going to sections now – Important information there
  - Any section will do until groups assigned
- THIS Friday is Drop Deadline! HARD TO DROP LATER!
  - If you know you are going to drop, do so now to leave room for others on waitlist!
  - Why do we do this? So that groups aren’t left without members!
**Administrivia (Con’t)**

- Group sign up via autograder form next week
  - Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
  - Remember: Your TA needs to see you in section!

- Midterm 1: 2/17
  - 7-9PM in person
  - We will say more about material when we get closer…

- Midterm 1 conflicts
  - We will handle these conflicts after have final class roster
  - Watch for queries by HeadTA to collect information

---

**Recall: 3 types of Kernel 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
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall

- **Interrupt**
  - External asynchronous event triggers context switch
  - eg. 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, …

---

**Recall: User/Kernel (Privileged) Mode**

- **User Mode**
  - Limited HW access

- **Kernel Mode**
  - Full HW access

---

**Implementing Safe Kernel Mode Transfers**

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

- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
  - More on this next time

- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself!
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

Interrupt Vector

interrupt number () { intrpHandler_i () { ... } }

- Address and properties of each interrupt handler

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

Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled
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

During Interrupt/System Call

Recall: UNIX System Structure
A Narrow Waist

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

Putting it together: web server

Request

Reply

(retrieved by web server)
Recall: Processes

- How to manage process state?
  - How to create a process?
  - How to exit from a process?

- Remember: Everything outside of the kernel is running in a process!
  - Including the shell! (Homework 2)

- Processes are created and managed… by processes!

Bootstrapping

- If processes are created by other processes, how does the first process start?

  - First process is started by the kernel
    - Often configured as an argument to the kernel before the kernel boots
    - Often called the “init” process

  - After this, all processes on the system are created by other processes

Process Management API

- exit – terminate a process
- fork – copy the current process
- exec – change the program being run by the current process
- wait – wait for a process to finish
- kill – send a signal (interrupt-like notification) to another process
- sigaction – set handlers for signals
**pid.c**

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <unistd.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

    /* get current processes PID */
    pid_t pid = getpid();
    printf("My pid: %d\n", pid);

    exit(0);
}
```

Q: What if we let main return without ever calling exit?
- The OS Library calls exit() for us!
- The entrypoint of the executable is in the OS library
- OS library calls main
- You’ll see this in Project 0: init.c

**Process Management API**

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the program being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a signal (interrupt-like notification) to another process
- **sigaction** – set handlers for signals

**Creating Processes**

- **pid_t fork()** – copy the current process
  - New process has different pid
  - New process contains a single thread
- **Return value from fork()**: pid (like an 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
- **State of original process duplicated in both Parent and Child!**
  - Address Space (Memory), File Descriptors (covered later), etc...

**fork1.c**

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

    pid_t cpid, mypid;
    pid_t pid = getpid();
    printf("My 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");
    }
}
```
# fork1.c

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("Parent pid: %d\n", pid);
        printf("\n\nparent of [\%d]\n", cpid);
    }
    else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("Child pid: %d\n", mypid);
    }
    else {
        perror("Fork failed");
    }
}
```

1. What does this print?
2. Would adding the calls to `sleep()` matter?

# fork_race.c

```c
int i;
pid_t cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i); // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
} else {
    perror("Fork failed");
}
```

Recall: a process consists of one or more threads executing in an address space
- Here, each process has a single thread
- These threads execute concurrently

• What does this print?
• Would adding the calls to `sleep()` matter?

# Running Another Program

- With threads, we could call `pthread_create` to create a new thread executing a separate function
- With processes, the equivalent would be spawning a new process executing a different program
- How can we do this?
Process Management API

• exit – terminate a process
• fork – copy the current process
• exec – change the program being run by the current process
• wait – wait for a process to finish
• kill – send a signal (interrupt-like notification) to another process
• sigaction – set handlers for signals

fork3.c

```c
int cpid = fork();
if (cpid > 0) {
    tcpid = wait(&status);
} else if (cpid == 0) {
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    perror("execv");
    exit(1);
}
```

Process Management

```
child

main()

exec

pid=fork();
if (pid==0)
    exec(...);
else
    wait(&stat)

fork

parent

fork

wait

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

Process Management API

• exit – terminate a process
• fork – copy the current process
• exec – change the program being run by the current process
• wait – wait for a process to finish
• kill – send a signal (interrupt-like notification) to another process
• sigaction – set handlers for signals
fork2.c – parent waits for child to finish

```c
int status;
pid_t tcpid;

if (cpid > 0) {
    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) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
```

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the program being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a signal (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

inf_loop.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!\n");
    exit(1);
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}
}
```

Common POSIX Signals

- SIGINT – control-C
- SIGTERM – default for kill shell command
- SIGSTP – control-Z (default action: stop process)
- SIGKILL, SIGSTOP – terminate/stop process
  - Can't be changed with sigaction
  - Why?

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
- You will build your own shell in Homework 2…
  - … using fork and exec system calls to create new processes…
  - … and the File I/O system calls we’ll see next time to link them together

Process vs. Thread APIs

- Why have fork() and exec() system calls for processes, but just a pthread_create() function for threads?
  - Convenient to fork without exec: put code for parent and child in one executable instead of multiple
  - It will allow us to programmatically control child process’ state
    » By executing code before calling exec() in the child
  - We’ll see this in the case of File I/O next time
- Windows uses CreateProcess() instead of fork()
  - Also works, but a more complicated interface

Threads vs. Processes

- If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?
- Depends on how much isolation we want
  - Threads are lighter weight [why?]?
  - Processes are more strongly isolated

Conclusion

- 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