Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
    pushq  %rbp
    movq  %rsp, %rbp
    ...
    popq  %rbp
    ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

Data & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Processes – another important abstraction

- First some preliminaries
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)

- Processes
  - Creating new processes
  - Fork and wait
  - Zombies
Control Flow

- So far, we’ve seen how the flow of control changes as a single program executes.
- A CPU executes more than one program at a time though – we also need to understand how control flows across the many components of the system.

Exceptional control flow is the basic mechanism used for:
- Transferring control between processes and OS
- Handling I/O and virtual memory within the OS
- Implementing multi-process applications like shells and web servers
- Implementing concurrency
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s *control flow* (or *flow of control*)

*Physical control flow*

- \(<\text{startup}>\)
- \(\text{inst}_1\)
- \(\text{inst}_2\)
- \(\text{inst}_3\)
- ... 
- \(\text{inst}_n\)
- \(<\text{shutdown}>\)
Altering the Control Flow

- **Up to now: two ways to change control flow:**
  - Jumps (conditional and unconditional)
  - Call and return
  Both react to changes in *program state*

- **Processor also needs to react to changes in *system state***
  - user hits “Ctrl-C” at the keyboard
  - user clicks on a different application’s window on the screen
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - system timer expires

- **Can jumps and procedure calls achieve this?**
  - Jumps and calls are not sufficient – the system needs mechanisms for “exceptional” control flow!
Exceptional Control Flow

- Exists at all levels of a computer system

- **Low level mechanisms**
  - Exceptions
    - change processor’s in control flow in response to a system event (i.e., change in system state, user-generated interrupt)
  - Combination of hardware and OS software

- **Higher level mechanisms**
  - Process context switch
  - Signals – you’ll hear about these in CSE451 and CSE466
  - Implemented by either:
    - OS software
    - C language runtime library
Exceptions

- An *exception* is transfer of control to the operating system (OS) in response to some *event* (i.e., change in processor state)

- **Examples:**
  - div by 0, page fault, I/O request completes, Ctrl-C

- *How does the system know where to jump to in the OS?*
Interrupt Vectors

- Each type of event has a unique exception number $k$
- $k = \text{index into exception table (a.k.a. interrupt vector)}$
- Handler $k$ is called each time exception $k$ occurs

Basically a jump table for exceptions...
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - Handler returns to “next” instruction

- Examples:
  - I/O interrupts
    - hitting Ctrl-C on the keyboard
    - clicking a mouse button or tapping a touchscreen
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button on front panel
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC
Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

- **Traps**
  - Intentional: transfer control to OS to perform some function
  - Examples: *system calls*, breakpoint traps, special instructions
  - Returns control to “next” instruction

- **Faults**
  - Unintentional but possibly recoverable
  - Examples: page faults (recoverable), segment protection faults (unrecoverable), integer divide-by-zero exceptions (unrecoverable)
  - Either re-executes faulting (“current”) instruction or aborts

- **Aborts**
  - Unintentional and unrecoverable
  - Examples: parity error, machine check (hardware failure detected)
  - Aborts current program
Trap Example: Opening File

- User calls: `open(filename, options)`
- Function `open` executes system call instruction `int`

```
0804d070 <__libc_open>:
  ...  
804d082:    cd     80  int  $0x80
804d084:    5b     pop  %ebx
  ...  
```

**User Process** → **OS**

- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```c
int a[1000];
main ()
{
    a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10

- Page handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
- Successful on second try
Fault Example: Invalid Memory Reference

```c
int a[1000];
main ()
{
    a[5000] = 13;
}
```

80483b7: c7 05 60 e3 04 08 0d movl $0xd,0x804e360

- Page handler detects invalid address
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
### Exception Table IA32 (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-127</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>System call</td>
<td>Trap</td>
</tr>
<tr>
<td>129-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>

Summary

Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception
What is a process?

- What is a program? A processor? A process?
What is a process?

- Why are we learning about processes?
  - Processes are another abstraction in our computer system – the process abstraction provides an interface between the program and the underlying CPU + memory.

- What do processes have to do with exceptional control flow (previous lecture)?
  - Exceptional control flow is the mechanism that the OS uses to enable multiple processes to run on the same system.
Processes

- Definition: A *process* is an instance of a running program
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with **two key abstractions:**
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of main memory

- Why are these illusions important?

- How are these illusions maintained?
  - Process executions interleaved (multi-tasking)
  - Address spaces managed by virtual memory system – next course topic
Concurrent Processes

- Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time.
- Otherwise, they are *sequential*.
- **Examples:**
  - Concurrent: A & B, A & C
  - Sequential: B & C

![Diagram showing concurrent and sequential processes]

**Processes**
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
- However, we can think of concurrent processes as executing in parallel
Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of a user process

- Control flow passes from one process to another via a *context switch*... (how?)

```
Process A          Process B
user code          user code
kernel code        kernel code
user code          user code
context switch
context switch
user code          user code
```
Creating New Processes & Programs

- **fork-exec model:**
  - `fork()` creates a copy of the current process
  - `execve()` replaces the current process’ code & address space with the code for a different program

- **`fork()` and `execve()` are system calls**
  - Note: process creation in Windows is slightly different from Linux’s fork-exec model

- **Other system calls for process management:**
  - `getpid()`
  - `exit()`
  - `wait()` / `waitpid()`
fork: Creating New Processes

- **pid_t fork(void)**
  - creates a new process (child process) that is identical to the calling process (parent process), including all state (memory, registers, etc.)
  - returns 0 to the child process
  - returns child’s process ID (pid) to the parent process

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

- **fork is unique (and often confusing) because it is called once but returns twice**
Understanding fork

**Process n**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

Process n

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Child Process m

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

**Process n**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Child Process m**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**pid = m**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**pid = 0**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

Process n

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Child Process m

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Which one is first?

```
hello from parent
hello from child
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

hello from parent

Which one is first?

hello from child
Fork Example

- **Parent and child both run the same code**
  - Distinguish parent from child by return value from `fork()`
  - Which runs first after the `fork()` is undefined

- **Start with same state, but each has a `private copy`**
  - Same variables, same call stack, same file descriptors, same register contents, `same program counter`...

```c
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
Exec-­ing a new program

Very high-level diagram of what happens when you run the command “ls” in a Linux shell:
execve: Loading and Running Programs

- **int execve(**
  - char *filename,
  - char *argv[],
  - char *envp[]
  )

- **Loads and runs in current process:**
  - Executable **filename**
  - With argument list **argv**
  - And environment variable list **envp**
    - Env. vars: “name=value” strings
      (e.g. “PWD=/homes/iws/bpw”)

- **execve does not return (unless error)**

- **Overwrites code, data, and stack**
  - Keeps pid, open files, a few other items
**execute: Example**

```
envp[n] = NULL
envp[n-1] → “PWD=/homes/iws/gaetano”
...
envp[0] → “PRINTER=ps581”
```

```
argv[argc] = NULL
argv[argc-1] → “/usr/include”
...
argv[0] → “-l”
argv[0] → “ls”
```
exit: Ending a process

- **void exit(int status)**
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
  - **atexit()** registers functions to be executed upon exit

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

**function pointer**
Zombies

- Idea
  - When process terminates, it still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - A living corpse, half alive and half dead

- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process

- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by **init** process (pid == 1)
  - But in long-running processes we need *explicit* reaping
    - e.g., shells and servers
wait: Synchronizing with Children

- **int wait(int *child_status)**
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the `pid` of the child process that terminated
    - On successful return, the child process is reaped
  - If `child_status` ! = NULL, then the int that it points to will be set to a status indicating why the child process terminated
    - NULL is a macro for address 0, the null pointer
    - There are special macros for interpreting this status – see `wait(2)`

- If parent process has multiple children, `wait()` will return when *any* of the children terminates
  - `waitpid()` can be used to wait on a specific child process
wait Example

```c
void fork_wait() {
    int child_status;
    pid_t child_pid;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        child_pid = wait(&child_status);
        printf("CT: child %d has terminated\n", child_pid);
    }
    printf("Bye\n");
    exit(0);
}
```
Process management summary

- **fork** gets us two copies of the same process (but **fork()** returns different values to the two processes)
- **execve** has a new process substitute itself for the one that called it
  - Two-process program:
    - First **fork()**
    - if (pid == 0) { /* child code */ } else { /* parent code */ }
  - Two different programs:
    - First **fork()**
    - if (pid == 0) { **execve()** } else { /* parent code */ }
    - Now running two completely different programs
- **wait** / **waitpid** used to synchronize parent/child execution and to reap child process
Summary

■ Processes
  ▪ At any given time, system has multiple active processes
  ▪ Only one can execute at a time, but each process appears to have total control of the processor
  ▪ OS periodically “context switches” between active processes
    ▪ Implemented using *exceptional control flow*

■ Process management
  ▪ `fork`: one call, two returns
  ▪ `exec`: one call, usually no return
  ▪ `wait` or `waitpid`: synchronization
  ▪ `exit`: one call, no return
Detailed examples
Fork Example #2

- Both parent and child can continue forking

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
Fork Example #3

- Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
Fork Example #4

- Both parent and child can continue forking

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Fork Example #5

- Both parent and child can continue forking

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Zombie Example

```c
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- **ps** shows child process as “defunct”
- Killing parent allows child to be reaped by **init**
Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
    PID  TTY         TIME CMD
   6585  ttyp9    00:00:00  tcsh
   6676  ttyp9    00:00:06  forks
   6677  ttyp9    00:00:00  ps
linux> kill 6676
linux> ps
    PID  TTY         TIME CMD
   6585  ttyp9    00:00:00  tcsh
   6678  ttyp9    00:00:00  ps
```
wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```
waitpid(): Waiting for a Specific Process

- **waitpid(pid, &status, options)**
  - suspends current process until specific process terminates
  - various options (that we won’t talk about)

```c
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```