Today

- Exceptional Control Flow
- Processes

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**

```
<startup>
inst_1
inst_2
inst_3
...
inst_n
<shutdown>
```
Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return
  Both react to changes in *program state*

- **Insufficient for a useful system:**
  difficult 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

- **System needs mechanisms for “exceptional control flow”**

---

Exceptional Control Flow

- **Exists at all levels of a computer system**

- **Low level mechanisms**
  - Exceptions
    - change 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 (context switch and signals)
    - C language runtime library (nonlocal jumps)
Exceptions

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

\[ \text{User Process} \rightarrow \text{OS} \]

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

Interrupt Vectors

- Each type of event has a unique exception number \( k \)
- \( k \) = index into exception table (a.k.a. interrupt vector)
- Handler \( k \) is called each time exception \( k \) occurs
Asynchronous Exceptions (Interrupts)

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

- **Examples:**
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - clicking a mouse button or tapping a touch screen
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - 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
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program
Trap Example: Opening File

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

```c
open(filename, options)
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804d070</td>
<td><code>_libc_open</code></td>
<td></td>
</tr>
<tr>
<td>804d082</td>
<td><code>cd 80</code></td>
<td><code>int $0x80</code></td>
</tr>
<tr>
<td>804d084</td>
<td><code>5b</code></td>
<td><code>pop %ebx</code></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**User Process**

- `int` pop
- `returns`

**OS**

- `exception`
- `open file`

- 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;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483b7</td>
<td><code>c7 05 10 9d 04 08 0d</code></td>
<td><code>movl $0xd,0x8049d10</code></td>
</tr>
</tbody>
</table>

**User Process**

- `movl`
- `exception: page fault`
- `returns`

**OS**

- `Create page and load into memory`

- Page handler must load page into physical memory
- Returns to faulting instruction
- 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>

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 program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

- **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
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as executing in parallel (only an illusion)

```
+---------------------+---------------------+---------------------+
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
+---------------------+---------------------+---------------------+
```

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

```
+---------------------+---------------------+---------------------+
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
|                      |                      |                      |
+---------------------+---------------------+---------------------+
```
**fork: Creating New Processes**

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child’s process ID (pid) to the parent process

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

- Fork is interesting (and often confusing) because it is called *once* but returns *twice*

---

**Understanding fork**

**Process n**

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

**Child Process m**

```cpp
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`
- `Which one is first?`
- `hello from child`
Fork Example #1

- **Parent and child both run same code**
  - Distinguish parent from child by return value from `fork`
- **Start with same state, but each has private copy**
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```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);
}
```

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 #4

- 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");
}
```

exit: Ending a process

- void exit(int status)
  - exits a process
    - Normally return with status 0
  - atexit() registers functions to be executed upon exit

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

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

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

- Reaping
  - Performed by parent on terminated child (terrible nomenclature!)
  - 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
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

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

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

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

```c
int wait(int *child_status)
{
  suspends current process until one of its children terminates
  return value is the pid of the child process that terminated
  if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated
```
wait: Synchronizing with Children

```c
void fork9() {
    int child_status;
    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```

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);
}
```

**execve**: Loading and Running Programs

- `int execve(`
  - char *filename,
  - char *argv[],
  - char *envp)
- Loads and runs
  - Executable `filename`
  - With argument list `argv`
  - And environment variable list `envp`
- Does not return (unless error)
- Overwrites process, keeps pid
- Environment variables:
  - “name=value” strings
execve: Example

```
envp[n] = NULL
envp[n-1] -> "PWD=/homes/iws/perkins"
...
envp[0]  -> "USER=perkins"
```

```
argv[argc] = NULL
argv[argc-1] -> "/usr/include"
...
argv[0]   -> "ls"
```

execl and exec Family

- int execl(char *path, char *arg0, char *arg1, ..., 0)
- Loads and runs executable at path with args arg0, arg1, ...
  - path is the complete path of an executable object file
  - By convention, arg0 is the name of the executable object file
  - “Real” arguments to the program start with arg1, etc.
  - List of args is terminated by a (char *) 0 argument
  - Environment taken from char **environ, which points to an array of “name=value” strings:
    - USER=perkins
    - LOGNAME=perkins
    - HOME=/homes/iws/perkins

- Returns -1 if error, otherwise doesn’t return!
- Family of functions includes execv, execve (base function), execvp, execl, execle, and execlp
Summary

- **Exceptions**
  - Events that require non-standard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time, however,
  - Each process appears to have total control of the processor + has a private memory space

Summary (cont’d)

- **Spawning processes**
  - Call to `fork`
  - One call, two returns

- **Process completion**
  - Call `exit`
  - One call, no return

- **Reaping and waiting for Processes**
  - Call `wait` or `waitpid`

- **Loading and running Programs**
  - Call `exec` (or variant)
  - One call, (normally) no return