Processes and control flow

- Is branches/calls the only way we can get the processor to “go somewhere” in a program?
- What is a process?

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 diagram]

  - Startup
  - inst₁
  - inst₂
  - instₙ
  - 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

- How do we deal with the above? Are branches/calls sufficient?

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)

  ![Exception diagram]

- Examples:
  - div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C
  - How does the system know where to jump to?
Interrupt Vectors

- Each type of event has a unique exception number.
- Each entry in the exception table points to a code for an exception handler.
- 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 pins
  - 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
- 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

Fault Example: Page Fault

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

Fault Example: Invalid Memory Reference

- 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>24</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>28</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 (Hex)</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>

http://download.intel.com/design/processors/manuals/753765.pdf

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
- Why are these illusions important?
- How are these illusions maintained?

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** (how?)
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

```c
int 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
```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");
}
```

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?**
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("E0\n");
}
```

Fork Example #3

- Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("E0\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("E0\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();
        } else {
            printf("Bye\n") ;
        }
    } else {
        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();
        }
    } else {
        printf("Bye\n") ;
    }
}
```

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 (horror movie?)
  - 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) {
        printf("Terminating child, PID = \%d\n" , getpid());
        wait(0);
    } else {
        printf("Parent Process, PID = \%d\n" , getpid());
    }
}
```

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") ;
}
```

```c
void fork6()
{
    atexit(cleanup);
    fork();
    exit(0);
}
```
Non-terminating Child Example

```c
#include <unistd.h>

void fork);
{
    pid_t pid = fork();
    int x = 0;
    int child_status;
    if (pid == 0) { // Child
        printf("Running Child, PID = %d", pid);
    }
    else { // Parent
        printf("Parent: Child PID = %d", pid);
        child_status = wait(NULL);
    }
}
```

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

Synchronization!

```c
wait: Synchronizing with Children

int wait(int *child_status)
{
    if (fork()) return 0;
    // Child
    child_status = 0;
}
```

- Suspend current process until one of its children terminates
- Return value is the pid of the child process that terminated
- If `child_status` is NULL, then object it points to will be set to status indicating why the child process terminated

```c
wait: Synchronizing with Children

void fork()
{
    int child_status;
    if (fork() == 0) { // Child
        printf("Hello from child!");
    }
    else { // Parent
        printf("Hello from parent!");
        wait(&child_status);
        printf("Child has terminated!");
        printf("%d", child_status);
        exit(0);
    }
}
```

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 fork()
{
    pid_t pid = fork();
    int x = 0;
    child_status;
    for (i = 0; i < N; i++)
    {
        if (pid[i] == 0) // Child
        {
            printf("Running Child, PID = %d", pid);
        }
        else // Parent
        {
            printf("Parent: Child PID = %d", pid);
            child_status = wait(child_status);
        }
    }
}
```

waitpid(): Waiting for a Specific Process

- Suspend current process until specific process terminates
- Various options that we won’t talk about

```c
void fork()
{
    pid_t pid = fork();
    int x = 0;
    child_status;
    for (i = 0; i < N; i++)
    {
        if (pid[i] == 0) // Child
        {
            printf("Running Child, PID = %d", pid);
            child_status = wait(child_status);
        }
        else // Parent
        {
            printf("Parent: Child PID = %d", pid);
            child_status = wait(child_status);
        }
    }
}
```
execute: Loading and Running Programs

- int execute(
  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

execute: Example

- env[0]: "PWD=/home/lw/luiscobre"
- env[1]: "USER=luisvareze"
- env[2]: "HOME=/home/lw"
- argv[0]: "ls"
- argv[1]: "/usr/include"
- argv[2]: "-1"
- argv[3]: NULL

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
  - Col to fork
    - One call, two returns
- Process completion
  - Col exit
    - One call, no return
- Reaping and waiting for Processes
  - Col wait or waitpid
- Loading and running Programs
  - Col exec (or variante)
    - One call, normally no return