Exceptional Control Flow and Processes
Exceptional control flow

• Most control flow is sequential
  – Minor exceptions: jumps and procedure calls
    • Caused by changes in internal program state (and thus predictable)
  – However, we have also seen violations of this rule
    • Control flow changes in response to external factors
    • (e.g., exceptions in Java or segfaults in C)
Exceptional control flow

- **Exceptions** violate sequential control flow
  - Unconditional transfer to another location in code
    - Partially implemented in hardware, partially in software
  - Often the result of an error condition
    - But not necessarily – we can also use exceptions for time-sharing!
  - Categorized as **asynchronous** vs. **synchronous**
    - Whether it happens as a result of an external source or not
  - Categorized by **recovery possibility**
    - Always returns, sometimes returns, or never returns
  - If recovery is possible, further categorized by **recovery location**
    - Same instruction vs. next instruction
Interrupts

- **Interrupt**: communication mechanism
  - Asynchronous, always returns to next instruction
  - “Interrupts” execution as the result of an outside event
    - Example: an I/O operation has finished
    - Example: a process has finished its time slice
Traps

- **Trap**: intentional control transfer to kernel
  - Synchronous, (almost) always returns to next instruction
  - Like a function call, except the target runs in kernel mode
  - Also referred to as system calls
  - x86-64 instruction “syscall” w/ ID in %rax
  - Parameters are passed in %rdi-%r9; return value stored in %rax
  - Well-known standards (e.g., POSIX)
Faults

- **Fault**: error that is potentially correctable
  - Synchronous, sometimes returns to same instruction
  - **Page fault (#14)**: virtual memory cache miss
    - Recoverable – read the required page from slower memory
  - **Segmentation fault (#13)**: invalid memory access
    - Not recoverable – undefined behavior
  - **Divide-by-zero error (#0)**
    - Not recoverable – undefined result

(1) Current instruction causes a fault  \( I_{\text{curr}} \)
(2) Control passes to handler
(3) Fault handler runs
(4) Handler either reexecutes current instruction or aborts.
Aborts

- **Abort**: unrecoverable error
  - Synchronous, never returns
  - Machine check (#18): fatal hardware error
System calls

- In P4: `iotrap` instruction is a system call
  - Performs I/O operations using `stdin` and `stdout`
  - Input: single character or decimal integer
    - Destination memory address in `%rdi`
  - Output: single character, decimal integer, or string
    - Source memory address in `%rsi`

<table>
<thead>
<tr>
<th>Trap IDs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>charout</td>
</tr>
<tr>
<td>charin</td>
</tr>
<tr>
<td>decout</td>
</tr>
<tr>
<td>decin</td>
</tr>
<tr>
<td>strout</td>
</tr>
<tr>
<td>flush</td>
</tr>
</tbody>
</table>

In P4, you’ll simulate these system calls using standard C functions like `printf` and `scanf`
System calls

- Some of the functions we've been using in C are actually wrappers for a system call (or multiple system calls)
  - fopen, fread, malloc
    - System calls: open (id=2), read (id=0), mmap (id=9)
- System call interfaces are defined by standards
  - SysV vs. POSIX (IEEE standard: http://pubs.opengroup.org/onlinepubs/9699919799/)
- In general, system call wrappers are called system-level functions
- It is important to check for errors after calling these functions
  - Textbook uses wrapper functions (e.g., "Open") for this

```c
int fd = open("file.txt", O_RDONLY);
if (fd < 0) {
    fprintf(stderr, "Error opening file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}
```
Textbook notes

- Error handling is important!
  - Textbook provides error-handling wrappers; this is good practice
  - However, we'll omit error handling to simplify examples

- `envp` parameter to `main()` is not standard
  - `getenv()` is the only environmental mechanism defined by the POSIX C99 standard
Processes

- Exceptions enable **processes**
  - **Process**: a running program
    - One program, (possibly) many processes
  - Abstraction provided by OS kernel
    - One kernel, many user processes
  - Shared portion of virtual address space
    - Kernel memory (above stack)
    - This region is not visible to user programs
  - Toggle control (kernel and processes)
    - **Interrupts** – cycle through processes ("round robin")
    - **Traps** – function call from processes into kernel ("syscalls")
    - **Faults** – software error (recover or abort)
    - **Aborts** – stop process without taking down the machine
Implementing processes

- Processes are implemented by the OS kernel
  - Kernel maintains data structure w/ process information
    - Including an ID for each process (pid)
  - Multitasking via exceptional control flow
    - Periodic interrupt to switch processes
    - Called round-robin switching
  - Context switch: swapping current process
    - Save context of old process
    - Restore context of new process
    - Pass control to the restored process
Exception implementation

- Kernel exception table
  - Every exception is assigned a unique ID
  - Table translates exception ID to handler address
Processes

- **Process**: instance of an executing program
  - Independent single logical flow and private virtual address space
- **Logical flow**: sequence of executed instructions
- **Concurrency**: overlapping logical flows
- **Multitasking**: processes take turns
- **Parallelism**: concurrent flows on separate CPUs/cores
Process creation

• The `fork()` syscall creates a new process
  – Initializes new entry in the kernel data structures
  – To user code, the function call returns twice
    • Once for original process (parent) and once for new process (child)
    • Returns 0 in child process
    • Returns child pid in parent process
    • Both processes will continue executing concurrently
  – Parent and child have separate address spaces
    • Child's space is a duplicate of parent's at the time of the fork
    • They will diverge after the fork!
  – Child inherits parent's environment and open files
Process creation example

- Fork returns twice!

```c
int main ()
{
    printf("Before fork\n");
    pid_t pid = fork();
    printf("After fork: pid=%d\n", pid);
    return 0;
}
```
Process creation example

• What does this code do?

```c
int main ()
{
    printf("Before fork\n");

    pid_t pid1 = fork();

    printf("After fork: pid1=%d\n", pid1);

    pid_t pid2 = fork();

    printf("After second fork: pid1=%d pid2=%d\n", pid1, pid2);

    return 0;
}
```
Process creation example

- Fork returns twice! (every time)
  - Beware of non-determinism and I/O interleaving

```c
int main ()
{
  printf("Before fork\n");

  pid_t pid1 = fork();
  printf("After fork: pid1=%d\n", pid1);

  pid_t pid2 = fork();
  printf("After second fork: pid1=%d pid2=%d\n", pid1, pid2);

  return 0;
}
```

Exercise: Modify this program to fork a total of three processes
Parents can wait for children to finish

```c
int main ()
{
    printf("Before fork\n");

    pid_t pid = fork();
    if (pid != 0) {     // parent
        wait(NULL);
        printf("Child has terminated.\n");
    } else {            // child
        printf("Child is running.\n");
    }

    printf("After fork: pid=%d\n", pid);
    return 0;
}
```
Process control syscalls

- `#include <stdlib.h>`
  - `getenv`: get environment variable value
  - `setenv`: change environment variable value

- `#include <sys/types.h>`
  - `pid_t`: new type for PID value

- `#include <unistd.h>`
  - `fork`: create a new process
  - `getpid`: return current process id (pid)
  - `getppid`: return parent’s process id (pid)
  - `exit`: terminate current process
  - `execve`: load and run another program in the current process
  - `sleep`: suspend process for specified time period

- `#include <sys/wait.h>`
  - `waitpid`: wait for a particular child process to terminate (requires child’s PID)
  - `wait`: wait for any child process to terminate
Processes and shells

- A **shell** is an interactive application-level program that launches other programs (called **jobs** or **process groups**)
  - All spawned as a result of the same command
- **Foreground vs. background jobs**
  - A single foreground job (interactive I/O)
  - Zero or more background jobs
  - Use `&` to start something in the background
    - Ex: `./my_prog &`
  - Use **CTRL-Z** to send foreground job to background
  - Use **CTRL-C** to interrupt the foreground job
  - **fg**: promote background job to foreground
Shells use `fork()` and `execve()` to run commands.

```c
int main ()
{
    printf("Before fork\n");
    pid_t pid = fork();

    if (pid != 0) {   // parent
        wait(NULL);
        printf("Child has terminated.\n");
    } else {         // child
        printf("Child is running.\n");
        char *cmd = "/bin/uname";
        char *args[] = { "uname", "-a", NULL };
        char *env[]  = { NULL };
        execve(cmd, args, env);
        printf("This won't print unless an error occurs.\n");
    }

    printf("After fork: pid=%d\n", pid);
    return 0;
}
```
Linux process tools

- **ps** – list processes
  - "ps -fe" to see all processes on the system
  - "ps -fu <username>" to see your processes
- **top** – list processes, ordered by current CPU
  - Auto-updates
- **/proc** – virtual filesystem exposing kernel data structures
- **pmap** – display memory map of a process
- **strace** – prints a list of system calls from a process
  - Compile with "-static" to get cleaner traces