Exceptional Control Flow: Exceptions, Processes and Signals

User-level programs vs. OS

• Role of an OS:
  • Implements OS-level services: I/O, network
  • Enforce protection: no stomping on each other’s memory/files/packets…
  • Resource sharing: execute multiple programs “simultaneously”, ...

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)

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)
  • Combination of hardware and OS software
• Higher level mechanisms
  • Process context switch
  • Signals
  • Nonlocal jumps: setjmp()/longjmp()
• Implemented by either:
  • OS software (context switch and signals)
  • C language runtime library (nonlocal jumps)

How user-level programs interact with the OS?

• Invoke **syscalls**
  • Send/receive data
  • Create/delete files
  • Execute/kill other programs
• Generate **exceptions** (to be handled by OS)
  • Touch illegal memory
  • Divide by zero
• Get **interrupted** by OS
  • OS preempts a program to execute other programs
  • OS does “upcalls” to user-programs via signals

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**
  • data arrives from a disk or a network adapter
  • instruction divides by zero
  • user hits Ctrl-C at the keyboard
  • System timer expires
• System needs mechanisms for “exceptional control flow”
Exceptions

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

- 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
  - Handler returns to “next” instruction
- Examples:
  - I/O interrupts
  - hitting Ctrl-C at the keyboard
  - 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

Fault Example: Page Fault

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

- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Trap Example: Opening File

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

- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor
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>12-127</td>
<td>OS defined</td>
<td>Interrupt or trap</td>
</tr>
<tr>
<td>128-143</td>
<td>OS defined</td>
<td>Interrupt or 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 profound 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 (multitasking) or run on separate cores
  • Address spaces managed by virtual memory system
  • we’ll talk about this in a couple of weeks

Concurrent Processes

• Two processes run concurrently (are concurrent) if their flows overlap in time
• Otherwise, they are sequential

• Examples (running on single core):
  • 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 are running in parallel with each other

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 some 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 `pid` to the parent

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

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

---

**Understanding fork**

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

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

---

**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, ++x\n", ++x);
    } else {
        printf("Parent has x = %d, -->;\n", x);
        printf("Bye from process %d with x = %d, getpid(), x;\n",
    }
}
```
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");
}
```

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"
    • 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
  • So, only need explicit reaping in long-running processes
    • e.g., shells and servers
Zombie Example

• ps shows child process as “defunct”

Nonterminating Child Example

• Child process still active even though parent has terminated

wait: Synchronizing with Children

• 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() Example

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

waitpid(): Waiting for a Specific Process

• suspends current process until specific process terminates
• various options (see textbook)
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
- Does not return (unless error)
- Overwrites code, data, and stack
  - keeps pid, open files and signal context
- Environment variables:
  - "name=value" strings
  - getenv and putenv

execve Example

```c
if ((pid = Fork()) == 0) /* Child runs user job */
        if (execve(argv[0], argv, environ) < 0) {
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
```

Stack bottom

Summary

- Exceptions
  - Events that require nonstandard 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 on a single core, though
  - Each process appears to have total control of processor + private memory space

Summary (cont.)

- Spawning processes
  - Call 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 execve (or variant)
  - One call, (normally) no return

ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software
  - Process Context Switch
  - Hardware timer and kernel software
- Signals
  - Kernel software
- Nonlocal jumps
  - Application code

Today

- Multitasking, shells
- Signals
- Nonlocal jumps
The World of Multitasking

- System runs many processes concurrently
- Process: executing program
  - State includes memory image + register values + program counter
- Regularly switches from one process to another
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority

Programmer’s Model of Multitasking

- Basic functions
  - `fork` spawns new process
    - Called once, returns twice
  - `exit` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait` and `waitpid` wait for and reap terminated children
  - `execve` runs new program in existing process
    - Called once, (normally) never returns
- Programming challenge
  - Understanding the nonstandard semantics of the functions

Shell Programs

- A shell is an application program that runs programs on behalf of the user.
  - `sh` Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - `csh` BSD Unix C shell (`tcsh`: enhanced `csh` at CMU and elsewhere)
  - `bash” Bourne-Again” Shell

Signals

- A signal is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID’s (1-30)
  - only information in a signal is its ID and the fact that it arrived

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., &lt;c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Sending a Signal

- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process.

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD).
  - Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process.

Receiving a Signal

- A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.

- Three possible ways to react:
  - Ignore the signal (do nothing).
  - Terminate the process (with optional core dump).
  - Catch the signal by executing a user-level function called signal handler.

  Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Pending and Blocked Signals

- A signal is pending if sent but not yet received.

- There can be at most one pending signal of any particular type.

- Important: Signals are not queued.

  - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.

- A process can block the receipt of certain signals.

  - Blocked signals can be delivered, but will not be received until the signal is unblocked.

- A pending signal is received at most once.

Signal Concepts

- Kernel maintains pending and blocked bit vectors in the context of each process.

  - pending: represents the set of pending signals.

    - Kernel sets bit k in pending when a signal of type k is delivered.

    - Kernel clears bit k in pending when a signal of type k is received.

  - blocked: represents the set of blocked signals.

    - Can be set and cleared by using the sigprocmask function.

Process Groups

- Every process belongs to exactly one process group.

- Examples:

  - `/bin/kill -9 24818`

    Send SIGKILL to process 24818

  - `/bin/kill -9 -24817`

    Send SIGKILL to every process in process group 24817

Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process or process group.

Examples:

- `/bin/kill -9 24818`

  Send SIGKILL to process 24818

- `/bin/kill -9 -24817`

  Send SIGKILL to every process in process group 24817
Receiving Signals

• Suppose kernel is returning from an exception handler and is ready to pass control to process \( p \).

• Kernel computes \( pnb = \text{pending} \land \text{~blocked} \):
  - The set of pending nonblocked signals for process \( p \).

• If \( pnb \neq 0 \):
  - Pass control to next instruction in the logical flow for \( p \).
  - Else:
    - Choose the least nonzero bit \( k \) in \( pnb \) and force process \( p \) to receive signal \( k \).
    - The receipt of the signal triggers some action by \( p \).
    - Repeat for all nonzero \( k \) in \( pnb \).
    - Pass control to next instruction in logical flow for \( p \).

Default Actions

• Each signal type has a predefined default action, which is one of:
  - The process terminates.
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

Installing Signal Handlers

• The `signal` function modifies the default action associated with the receipt of signal `signum`:
  ```c
  handler_t *signal(int signum, handler_t *handler)
  ```

• Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`.
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, `handler` is the address of a signal handler:
    - Called when process receives signal of type `signum`.
    - Referred to as “installing” the handler.
    - Executing handler is called “catching” or “handling” the signal.
    - When the handler executes its return statement, control passes back to

Signals Handlers as Concurrent Flows

• A signal handler is a separate logical flow (not process) that runs concurrently with the main program.
  - “Concurrently” in the “not sequential” sense.

```
Process A
while (1)
{
    handler();
}
```

Summary

• Signals provide process-level exception handling:
  - Can generate from user programs.
  - Can define effect by declaring signal handler.