Exceptional Control Flow Part I

Today
- Exceptions
- Process context switches
- Creating and destroying processes

Next time
- Signals, non-local jumps, …
Control flow

- Computers 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 system’s physical control flow (or flow of control).

![Physical control flow diagram]

- From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
- This sequence is the system’s physical control flow (or flow of control).
Altering the control flow

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

- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state
    - Data arrives from a disk or a network adapter
    - Instruction divides by zero
    - User hits ctl-c at the keyboard
    - System timer expires

- System needs mechanisms for “exceptional control flow” (ECF)
Exceptional control flow

Mechanisms for exceptional control flow exists at all levels of a computer system

- **Low level mechanism**
  - 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
Exceptions

- Exception – a transfer of control to the OS in response to some event (i.e., change in processor state)

- E.g. page fault, arithmetic overflow, I/O done, …
Interrupt vectors

- Each type of event has a unique exception number $k$.
- Index into jump table (a.k.a., interrupt vector).
- Jump table entry $k$ points to a function (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 pin
  - handler returns to “next” instruction.

Examples:

- I/O interrupts
  - hitting ctrl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector 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).
    - Either re-executes faulting (“current”) instruction or aborts.
  - Aborts
    - Unintentional and unrecoverable
    - Examples: parity error, machine check.
    - Aborts current program
Trap example

- Opening a File
  - User calls `open(filename, options)`
    
    Users calls `open(filename, options)`
    
    ```
    User Process
    OS
    ```
    
    - 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 #1

- Memory reference
  - 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

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

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

Diagram:
- User Process
- OS
- event → movl → page fault → return → Create page and load into memory
Fault example #2

- Memory reference
  - User writes to memory location
  - Address is not valid
  - Page handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with “segmentation fault”

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

![Diagram showing the process flow from user process to OS, including event detection, page fault, address detection, and signal process.]
Processes

- Def: 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 address space
    - Each program seems to have exclusive use of main memory.

- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
Logical control flows

Each process has its own logical control flow
Concurrent processes

- Two processes run concurrently (are concurrent) if their 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 running in parallel
Context switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Not a separate process, but runs as part of user process
- Control flow passes from one process to another via a *context switch*.

![Diagram showing context switching between Processes A and B]
Private address spaces

* Each process has its own private address space.

- kernel virtual memory (code, data, heap, stack)
- memory mapped region for shared libraries
- run-time heap (managed by malloc)
- read/write segment (.data, .bss)
- read-only segment (.init, .text, .rodata)
- unused

Memory invisible to user code

%esp (stack pointer)

brk

loaded from the executable file
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 process

```c
if (fork() == 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
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

- Key points
  - Both parent and child can continue forking

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

- Key points
  - 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

- Key points
  - 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

- Key points
  - 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: Destroying 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
  - Only need explicit reaping for long-running processes
    - E.g., shells and servers
Zombie - Example

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped

```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 */
    }
}
```
Nonterminating child example

```c
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
**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: 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++) { /*FIXME – whichever ends first*/
        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 terminate abnormally\n", wpid);
    }
}
```
waitpid

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```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);
    }
}
```
wait/waitpid example outputs

Using **wait (fork10)**

<table>
<thead>
<tr>
<th>Child PID</th>
<th>Exit Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3565</td>
<td>103</td>
</tr>
<tr>
<td>3564</td>
<td>102</td>
</tr>
<tr>
<td>3563</td>
<td>101</td>
</tr>
<tr>
<td>3562</td>
<td>100</td>
</tr>
<tr>
<td>3566</td>
<td>104</td>
</tr>
</tbody>
</table>

Using **waitpid (fork11)**

<table>
<thead>
<tr>
<th>Child PID</th>
<th>Exit Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3568</td>
<td>100</td>
</tr>
<tr>
<td>3569</td>
<td>101</td>
</tr>
<tr>
<td>3570</td>
<td>102</td>
</tr>
<tr>
<td>3571</td>
<td>103</td>
</tr>
<tr>
<td>3572</td>
<td>104</td>
</tr>
</tbody>
</table>
exec: Running new programs

- 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
    - arg0 becomes the name of the process
      - typically arg0 is either identical to path, or else it contains only the executable filename from path
    - “real” arguments to the executable start with arg1, etc.
    - list of args is terminated by a (char *)0 argument
  - returns -1 if error, otherwise doesn’t return!

```c
main() {
  if (fork() == 0) {
    execl("/usr/bin/cp", "cp", "foo", "bar", 0);
  }
  wait(NULL);
  printf("copy completed\n");
  exit();
}
```
Summarizing

- **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, though
  - Each process appears to have total control of processor + private memory space

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

- **Terminating processes**
  - Call `exit`: one call, no return

- **Reaping processes**
  - Call `wait` or `waitpid`

- **Replacing program executed by process**
  - Call `execl` (or variant): one call, (normally) no return