Program and OS interactions: Exceptions and Processes

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
User-level programs vs. OS

safari  httpd  bomb

Send/recv packets, Open/close files

OS kernel

…
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”, ...
Ways user-level programs interact with 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

Achieved via h/w’s exceptional control flow
Topics

- Exceptional Control Flow
- Processes
Control Flow

- A CPU core reads and executes a series of instructions, one at a time
  - This sequence is the CPU’s control flow

*Physical control flow*

<startup>

\[ \text{inst}_1 \]
\[ \text{inst}_2 \]
\[ \text{inst}_3 \]
\[ \ldots \]
\[ \text{inst}_n \]

<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*

- **“exceptional control flow” – 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
Types of Exceptions

- **Asynchronous (interrupts):** caused by events external to CPU
  - hitting Ctrl-C at the keyboard
  - arrival of a packet from a network
  - arrival of data from a disk

- **Synchronous:** caused by executing an instruction
  - Intentional (*Traps*): e.g. INT 0x80 (syscall)
  - Unintentional but recoverable (*Faults*): e.g. page or protection faults
  - unintentional and unrecoverable (*Aborts*): e.g. memory error
Exceptions

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

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

- **fault**: return to $l_{\text{current}}$
- **trap, interrupt**: return to $l_{\text{next}}$
- **Abort**: machine reboot, process termination
Each type of exception has a unique exception number \( k \), used to index into exception table (a.k.a. interrupt vector).

- Handler \( k \) is called each time exception \( k \) occurs.

- Handled by the OS, not user-level programs.
Trap Example: Syscall (System call)

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

```
0804d070 <__libc_open>:
  ... 
804d082:  cd 80  int $0x80  # interrupt 80
804d084:  5b    pop  %ebx
  ... 
```

- 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 address is not in memory (on disk)

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

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

User Process

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

OS

```
Create page and load into memory
returns
```

- OS loads 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;
}
```

- OS detects invalid address
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
Topics

- Exceptional Control Flow
- Processes
Processes

- A **process** is an instance of a running program.
  - Not the same as “program” or “processor”

- **OS and 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
  - Virtual Memory managed by OS
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*
- Control flow passes from one process to another via a *context switch*

![Diagram showing context switching between two processes, A and B. Each process alternates between user code and kernel code, represented by arrows. Time flows from top to bottom.](image-url)
Basic UNIX syscalls for managing processes

- **fork**
  - Create a new process

- **exit**

- **wait**
  - Synchronize among processes

- **execve**
  - Load a new process image

Syscalls are documented in man pages section 2: `man -2 fork`

Standard C library provides wrapper functions for many syscalls
fork: Creating New Processes

- int fork(void)
  - creates a new process (child process) that is identical to the calling process (parent process)
- Fork is called once but returns twice

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

Return 0 to the child process

Return child’s pid to the parent
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");
}
```

**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");
}
```

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

- Just parent 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

- Just 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");
}
```
Beware: the fork bomb

Both parent and child can continue forking... indefinitely

```c
void fork_bomb()
{
    int cnt = 0;
    while(1) {
        printf("L%d\n", cnt++);
        fork();
    }
}
```

CAUTION: DO NOT RUN ON A SYSTEM YOU DO NOT OWN!
THIS WILL PROBABLY HOSE YOUR SYSTEM!!!
**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();
    fork();
    exit(0);
}
```
Zombies

- **Idea**
  - When process terminates, still consumes system resources
    - Why? So that parents can learn of children’s exit status
  - Called a “zombie” - dead, but still taking up resources

- **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
    - Ancestor of all
Zombie Example

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**

- How do you handle them without killing parent process?
**wait: Synchronizing with Children**

- `int wait(int *child_status)`
  - Blocks until some child exits
  - Return value is the `pid` of terminated child
  - If multiple children completed, will take in arbitrary order (use `waitpid` to wait for a specific child)

```c
void fork9() {
    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        printf("HP: hello from parent\n");
        wait(NULL);
        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 (Wait If Exited) 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 terminate abnormally\n", wpid);
    }
}
```
waitpid(): Waiting for a Specific Process

- `waitpid(pid, &status, options)`
  - suspends current process until specific process terminates
  - various options (see textbook)

```c
void forkl1()
{
    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 = N-1; i >= 0; 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: Execute a new program

- int execve(
  char *filename, //executable
  char *argv[], //arguments
  char *envp[]   //environment
)

- Does not return (unless error)

- overwrites current process code, data, and stack with new executable
  - keeps pid, open files and signal context

Kernel virtual memory

- Stack
- shared libraries
- Heap
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused
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);
    }
}
```

```
argv[argc] = NULL
argv[argc-1]  
...           
argv[0]       
```

- `/usr/include`
- `-lt`
- `"ls"

```
envp[n] = NULL
envp[n-1]  
...           
envp[0]       
```

- `"PWD=/home/cso"
- `"PATH=/usr/bin:/bin:..."
- `"USER=cso"
```
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`

- **Environment variables:**
  - “name=value” strings
  - `getenv()` and `putenv()`
Summary

- **Exceptions**
  - Used for events that require non-standard control flow
  - Hardware mechanism: exception control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **OS abstraction of Processes**
  - At any given time, system has multiple active processes
  - Each process appears to have total control of CPU + private memory space
  - Programs interface with OS using syscalls/interrupts:
    - fork, exit, wait, exec