Exceptional Control Flow: Processes
(Slides adapted from CSAPP)
Announcement

- HW 3 due: this Friday
Previous class summary

- Exceptions
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- This class: Processes
  - What is the process?
  - Systems calls for process creation and termination in Linux.
  - Understanding multitasking: Shell programs
Agenda

- What is a process?
- Concurrent process
- Linux process control
  - Creating process: fork
  - Terminating process: (exit)
  - Zombie
  - Reaping child process (wait)
  - Executing programs in a process (execenv)
- Understanding multitasking
  - Shell programs
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
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 as 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*

![Diagram showing context switching between two processes](image-url)

CEN591 Fall 2011
Process control in Unix/Linux

- Process states
  - Running
  - Stopped (Suspended is not scheduled)
  - Terminated

- Process control
  - Creating process
  - Terminating process
  - Reaping child process
  - Synchronizing with child process
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
pid_t 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

System call getpid returns the Process Id (PID)
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");
}
```

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);
}
```

Sample output:
Parent has x = 0
Bye from process 33020 with x = 0
Child has x = 2 (since child has private copy of x, parent’s subtraction does not affect the child’s variable)
Bye from process 33021 with x = 2
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

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

Sample output:
L0
Bye
L1
Bye
L2
Bye
Bye
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);
}
```

Output:
```
cleaning up
cleaning up
```
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

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 to reap a child?
  - The parent signals the OS that it no longer needs the zombie by using one of the `wait()` system calls.
Nonterminating Child Example

Child process still active even though parent has terminated

Must kill explicitly, or else will keep running indefinitely