Operating Systems

Processes - II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Two Main Challenges

- How can OS implement virtualization of the CPU without adding excessive overhead to the system?

- How can the OS run processes efficiently while retaining control over the CPU?

Straightforward Solution: Direct Execution

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Q1: How does the OS guarantee that this program won't do anything nasty?

Q2: How can the OS stop this process from running and switch to another process?

Answer to Q1: Introducing Kernel mode and User mode

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

Trap table contains the addresses of the different trap handles (aka Interrupt Service Routines)

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

What Happened?

- Processes run at user mode so cannot access the hardware.
- If a process needs something from the hardware, it makes a system call with a specific system call number.
- This switches the machine to kernel mode and the OS starts executing the service routine specified by the number.
- Now, a process has limited power over the system and cannot do something that we don't want it to do.

Answer to Q2: We need help from the hardware

- If a process is using the CPU, this means the OS is not running (you can scale that to multi-processes with multicore).
- If so, how can an OS stop a process to switch to another process?
- Solution: timer interrupts

Timer Interrupt

- A device programmed to raise and interrupt every x milliseconds
  - x is set by the OS
  - OS starts the timer at boot time.
- When the interrupt occurs, the machine switches to kernel and the OS takes control.
  - The scheduler part of the OS
  - The scheduler decides: continue running the current process or context switch to another one.

APIs for dealing with Processes in C

Don't forget to include the following header files:

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
```

Basic Syscalls for Managing Processes

- **`fork`** spawns new process
  - Called once, returns twice
- **`exit`** terminates own process
  - Puts it into "zombie" status until its parent reaps
- **`wait`** and **`waitpid`** wait for and reap terminated children
- **`execve`** runs new program in existing process
  - Called once, never returns

`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*

Return 0 to the child process

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

Return child's pid to the parent

Understanding fork

Process n

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

pid = m

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

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

```
hello from parent
```

Child Process m

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

pid = 0

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

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

```
hello from child
```

Fork Example #1

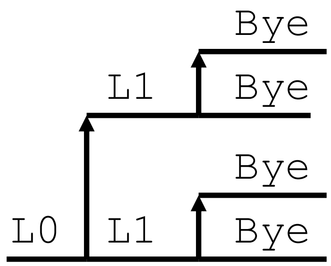
- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state, but each has private copy of memory
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
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

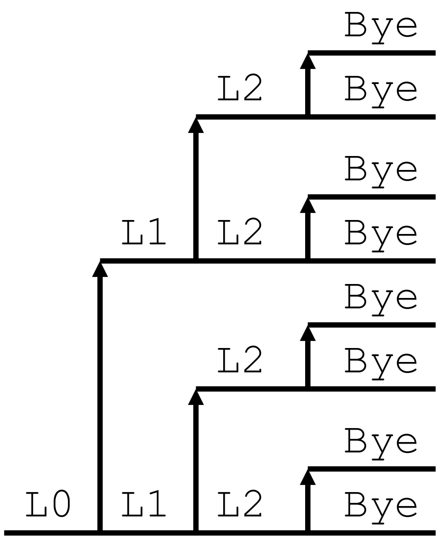
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Both parent and child can continue forking

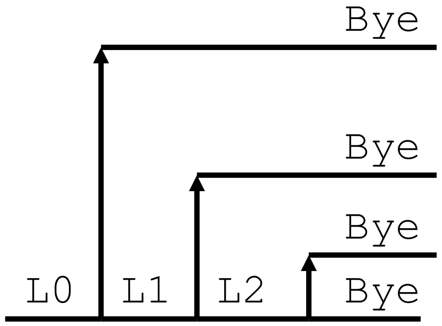
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Both parent and child can continue forking

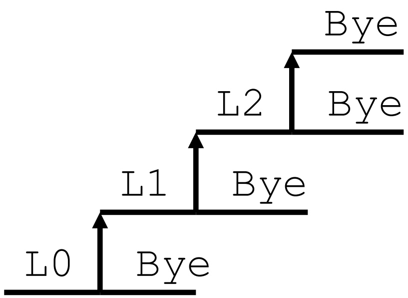
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



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");
}
```



`exit`: Ending a process

- `void exit(int status)`
  - exits a process
    - Normally return with status 0
  - `atexit(function_name)` make function_name execute upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies!

- Idea
  - When process terminates , still consumes system resources (i.e. an entry in process table)
    - Why? So that parents can learn of children's exit status
  - Called a "zombie"
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - OS discards process
- What if parent doesn't reap?
  - If parent has terminated, then child will be reaped by `init` process (the great-great-…-grandparent of all user-level processes)

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 */
    }
}
```

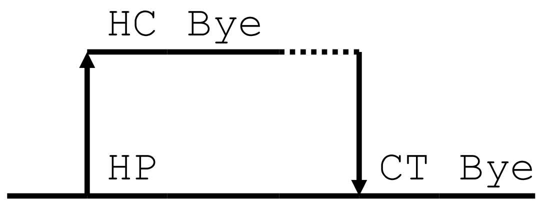
```
linux> ./forks7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`

`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)

```
void fork8() {
   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(0);
}
```



This is how child process is reaped by parent process.

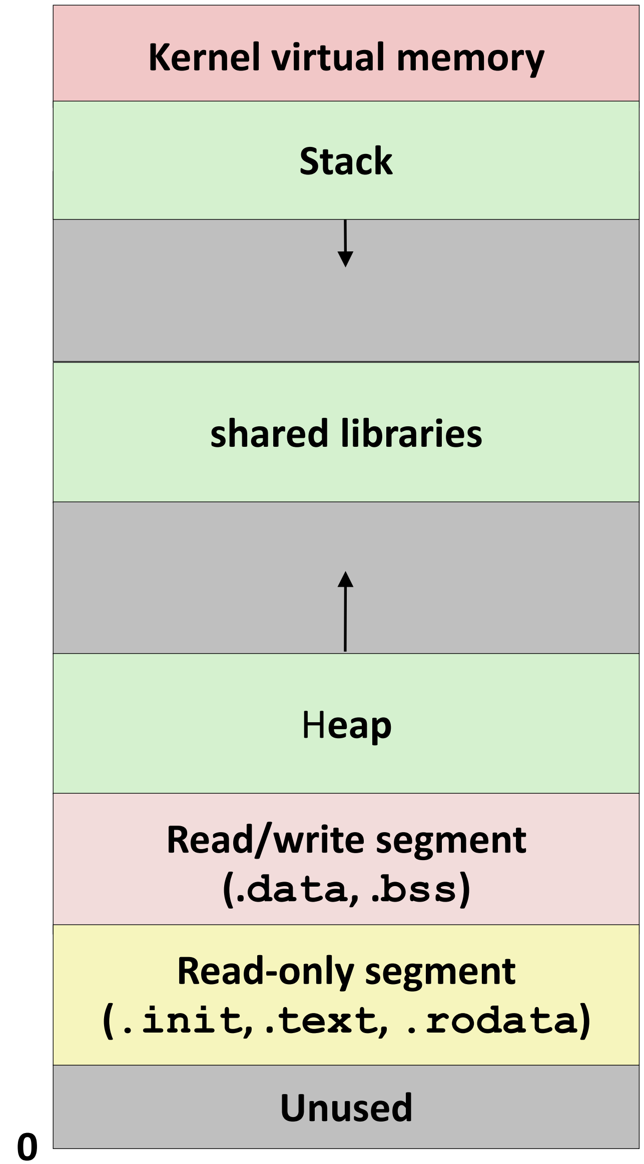
execve

- **int execve(char *fname, char *argv[],** `char *envp[])`
  - Executes program named by `fname`

```
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);
    }
}
```

`execve`: Load a new program image

- `execve` causes OS to overwrite code, data, and stack of process
  - keeps pid and open files



Conclusions

- OS must ensure that a running process will not cause any harm to the whole system.
- OS runs in kernel mode and has access to protected instructions (i.e., a set of machine code instructions that cannot be executed while the machine is in user mode).