CS214-system programming

- Section 03/08 recitation 04

Yunhe Gao
yg397@scarletmail.rutgers.edu

Processes

- An instance of an executing program (running program), including the current values of the program counter, registers, and variables.
- The CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. This rapid switching back and forth is called **multiprogramming**.

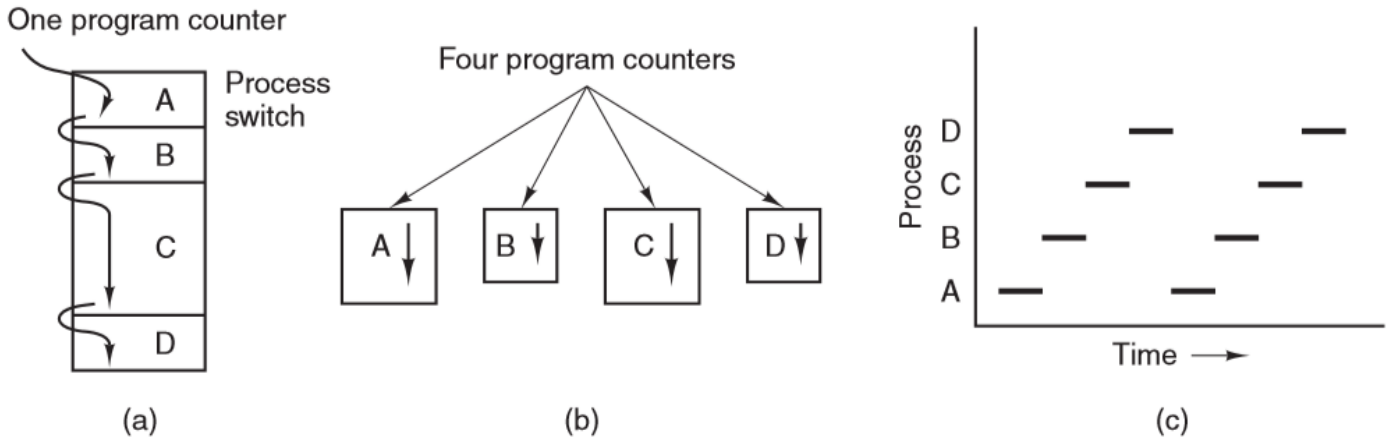
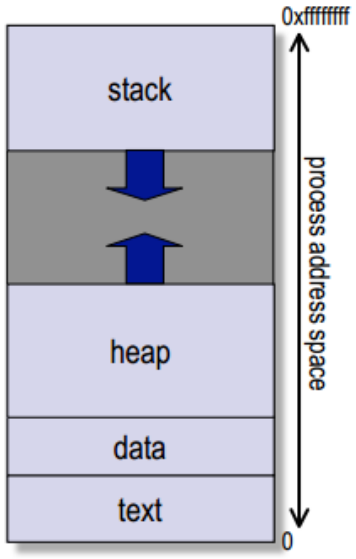


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Processes

- A process is an instance of a computer program that is being executed, it contains the program code and its current activity
- Each process has a **Process Control Block (PCB)** to describe its current execution state and some resources it is now processing
  - Identification information (e.g. parent process, user, process ID)
  - Execution contexts such as separate threads of execution
  - Address space (virtual memory)
  - I/O state (file handles, network communication endpoints, etc.)
  - etc.
- PCBs are stored as entries in a process table



Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a Process Control Block

Processes

- `ps <s|l|o|t|m|a>`
  - Process status command. Provides information about currently running processes.
  - http://man7.org/linux/man-pages/man1/ps.1.html

- `kill [PID]`
  - Send a SIGKILL signal to a process with the provided process ID.
  - You can send any kind of signal with the kill command (e.g. `kill -2 <pid>`)
  - http://man7.org/linux/man-pages/man2/kill.2.html

- `top`
  - Displays a monitor on processes and system resources. Allows you to send signals to processes. Also see '`htop`'.
  - https://linux.die.net/man/1/top

Various ways to create a process

- System initialization (boot, init process)
- Execution of a process-creation system call (fork) by a running process
- A user request to create a new process (click on an icon, run a .out file)
- Initiation of a batch job. (Batch job: A sequence of commands to be executed by the operating system is listed in a file and submitted for execution as a single unit.)

exec

- The exec() family of functions **replaces** the current process image with a new process image. It loads the program into the current process space and runs it from the entry point.
- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- Args:
  - The path to the program to execute
  - The arguments to the program

- exec only returns when error occurs
  - If success, the process is replaced by a new one. There is no place to return.

Fork

```
int pid = fork();
```

- fork() is a system call that creates another process.
- The new process is called the **child** and the creating process is called the **parent**.
- The child is a copy of the parent process except for identification and scheduling state.
  - SAME: They share process image and process control structure
    - That is, they share the same code, variables, file descriptors, signal disposition (reaction to signals)
  - DIFFERENT: PID, parent PID (PPID), and address space
    - Parent and child run in two different address spaces. There is **no memory sharing** by default.

Fork

```
int pid = fork();
```

- fork() is called once, but returns twice.
- Once for the parent process, and once for the child.

- Returns an integer.
  - negative -- Error.
  - zero -- Zero is returned to the child process.
  - positive -- The PID of child process is returned to the parent process

Wait

When a process terminates execution, the OS releases most of the resources and information related to that process.

- It does keep data about resource utilization and termination status in case the parent process wants to know.
- The process still takes up space in the OS's process table.

By default, the OS sends a signal (SIGCHLD) to alert the parent that here is information about the child process available.

After the parent receives the information, the OS releases the rest of the resources tied to the terminated process and removes its pid from the resource table.

Thus, the parent process should wait on the child process to terminate.

Fork

```
int pid = fork();

if(pid == 0){
        printf(“This is the child process.\n”);
        exec(something);        // call a program different from the parent program
        exit(2);
        }
else{
        printf(“This is the parent process.\n”);
        int status;
        int pid = wait(&status); // write the exit information into status
        printf(“Got status: %d\n”, WEXITSTATUS(status));  //use macro to inspect the return
                                                          //status of child process
        }
```

fork vs exec

- fork starts a new process which is a copy of the one that calls it, Both parent and child processes are executed simultaneously, while exec replaces the current process image with another one.
- in case of fork(), parent and child processes have different PID, while exec remain the same PID when load a new program.

Zombies and orphans

What if the parent doesn't wait for the child to terminate?

- The OS cannot free the remaining resources allocated to the terminated process. The terminated process's pid is still in the OS's process table. That terminated process is called a **zombie process**

What if the parent process terminates before the child is done running?

- The child process is an **orphan process**. It is adopted by init (PID 1)

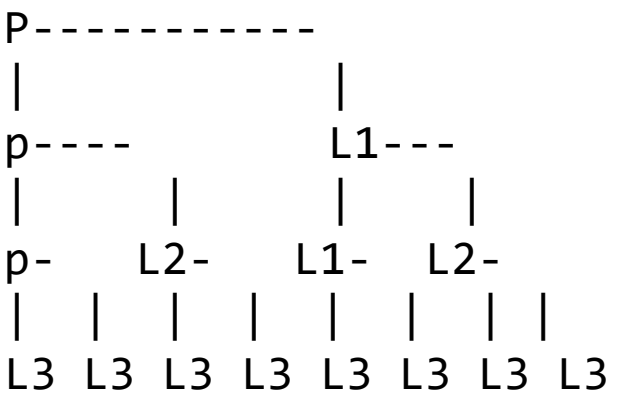
Some examples

How many hello will be printed?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();

    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
fork();        // Line 1
fork();        // Line 2
fork();        // Line 3
```



Some examples

What is the expect output?

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

void forkexample()
{
    int x = 1;
    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

```
Parent has x = 0
Child has x = 2
     (or)
Child has x = 2
Parent has x = 0
```

Variable changes in one process does not affected two other processes because data/state of two processes are different. And also parent and child run simultaneously so two outputs are possible.

Thanks