HCMC UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

– CO2018 Operating Systems – Fall 2018 –

Lab 3
Process (Part 1/2)

September 14, 2018

GOAL In this lab, students are expected to understand the definition of a process, and how an operating system manage a process.

CONTENT

- What is a process?
- How to retrieve processes' information?
- Processes in C Programming.

PREREQUISITE REQUIREMENT Review the definition of processes in previous lectures.

GRADING POLICY

- 30% in-class performance
- 70% E-learning submission

1. Introduction

A process could be simply referred as a program in execution. Apart from the (1) program code, which is known as the text section, a process also includes (2) the current activity including program counter and processor registers, (3) process stack which contains temporary data (such as function parameters, return addresses, and local variables), (4) a data section which contains global variables and (5) a heap containing memory that is dynamically allocated during process run time.
As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

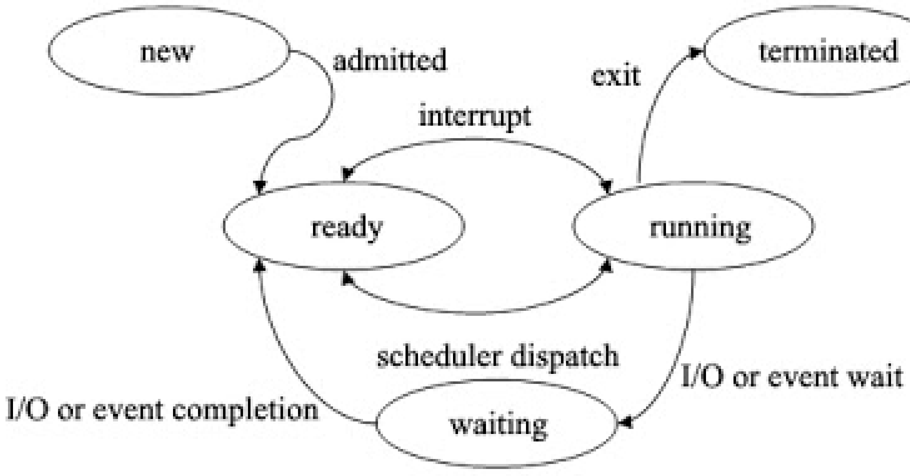

Figure 1.1: Diagram of process state.

To keep track of processes, the operating system maintains a process table (or list). Each entry in the process table corresponds to a particular process and contains fields with information that the kernel needs to know about the process. This entry is called a Process Control Block (PCB). Some of these fields on a typical Linux/Unix system PCB are:

- Machine state (registers, program counter, stack pointer)
- Parent process and a list of child processes
- Process state (ready, running, blocked)
- Event descriptor if the process is blocked
- Memory map (where the process is in memory)
- Open file descriptors

- Owner (user identifier). This determines access privileges & signaling privileges
- Scheduling parameters
- Signals that have not yet been handled
- Timers for accounting (time & resource utilization)
- Process group (multiple processes can belong to a common group)

A process is identified by a unique number called the process ID (PID). Some operating systems (notably UNIX-derived systems) have a notion of a process group. A process group is just a way to lump related processes together so that related processes can be signaled. Every process is a member of some process group.

2. How do we find the PID and group?

A process can retrieve its PID, group number, and parent's PID using the *getpid*, *getpgrp* and *getppid* system call, respectively. For example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        printf("Process ID: %d\n", getpid());
        printf("Parent process ID: %d\n", getppid());
        printf("My group: %d\n", getpgrp());

        return 0;
}
```

3. Creating a process

The *fork* system call clones a process into two processes running the same code. A value of -1 is returned upon failure.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        switch (fork()) {
        case 0:
                printf("I am the child: pid=%d\n", getpid());
                break;
        default:
```

```c
                printf("I am the parent: pid=%d\n", getpid());
                break;
        case -1:
                perror("Fork failed");
        }
        return 0;
}
```

4. Basics of Multi-process programming

Multi-process programming Implement a program using the `fork()` command to create child processes. Students can check the number of forked processes using the `ps` command.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>     /* defines fork(), and pid_t. */

int main(int argc, char ** argv) {

  pid_t child_pid;

  /* lets fork off a child process... */
  child_pid = fork();

  /* check what the fork() call actually did */
  if (child_pid == -1) {
        perror("fork"); /* print a system-defined error message */
        exit(1);
  }

  if (child_pid == 0) {
    /* fork() succeeded, we're inside the child process */
    printf("Hello, ");
    fflush(stdout);
  }
  else {
    /* fork() succeeded, we're inside the parent process */
    printf("World!\n");
    fflush(stdout);
  }

  return 0;
}
```

Compile and run the program Run the program multiple times and answer question 2 below.

5. Retrieve the code segment of process

5.1. The information of process

```c
/* Source code of loop_process.c */
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  int timestamp=0;
  while(1){
     printf("Time: %5d\n", timestamp++);
     sleep(1);
  }
  return 0;
}
```

```
$ gcc -o loop_process loop_process.c

$ ./loop_process
Time:     0
Time:     1
Time:     2
Time:     3
Time:     4
...
```

Find process ID `ps` command is used to find the PID of a process.

```
$ ps -a | grep loop_process
 7277 tc       ./loop_process
 7279 tc       grep loop_process
```

Retrieve PCB information of process on Linux system, each running process is reflected in the folder of filesystem **/proc**. Information of each process is associated with the folder called /proc/<pid>, where pid is process ID. For example, with the *pid* of the process above, **./loop_process** the directory containing the information of this process is **/proc/7277**.

```
$ ls /proc/<pid>
autogroup        environ      mountstats      smaps
auxv             exe          net/            stat
cgroup           fd/          ns/             statm
clear_refs       fdinfo/      oom_adj         status
cmdline          limits       oom_score       syscall
comm             maps         oom_score_adj   task/
coredump_filter  mem          pagemap
cpuset           mountinfo    personality
cwd              mounts       root
```

Use commands such as `cat`, `vi` to show the information of processes.

```
$ cat /proc/<pid>/cmdline
./loop_process

$ cat /proc/<pid>/status
Name:      loop_process
State:     S (sleeping)
Tgid:      7277
Pid:       7277
PPid:      6760
TracerPid:         0
Uid:       1001     1001     1001     1001
Gid:       50       50       50       50
FDSize:    32
...
```

5.2. Comparing the code segment of process and program

/proc/<pid>/ stores the information of code in `maps`

```
$ cat /proc/<pid>/maps
08048000-08049000 r-xp 00000000 00:01 30895 /home/.../loop_process
08049000-0804a000 rwxp 00000000 00:01 30895 /home/.../loop_process
b75e0000-b75e1000 rwxp 00000000 00:00 0
b75e1000-b76f8000 r-xp 00000000 00:01 646   /lib/libc-2.17.so
b76f8000-b76fa000 r-xp 00116000 00:01 646   /lib/libc-2.17.so
b76fa000-b76fb000 rwxp 00118000 00:01 646   /lib/libc-2.17.so
b76fb000-b76fe000 rwxp 00000000 00:00 0
b7705000-b7707000 rwxp 00000000 00:00 0
b7707000-b7708000 r-xp 00000000 00:00 0     [vdso]
b7708000-b7720000 r-xp 00000000 00:01 648   /lib/ld-2.17.so
b7720000-b7721000 r-xp 00017000 00:01 648   /lib/ld-2.17.so
```

```
b7721000-b7722000 rwxp 00018000 00:01 648    /lib/ld-2.17.so
bf9a8000-bf9c9000 rw-p 00000000 00:00 0      [stack]
```

Comparing with the program (executable binary file) using *ldd* to read executable binary files and *readelf* to list libraries that are used.

```
$ ldd loop_process
        linux-gate.so.1 (0xb77dc000)
        libc.so.6 => /lib/libc.so.6 (0xb76b6000)
        /lib/ld-linux.so.2 (0xb77dd000)

$ readelf -Ws /lib/libc.so.6 | grep sleep
 388: 000857f0 105 FUNC WEAK   DEFAULT 11 nanosleep@@GLIBC_2.0
 660: 000857f0 105 FUNC WEAK   DEFAULT 11 __nanosleep@@GLIBC_2.2.6
 803: 000bda48 121 FUNC GLOBAL DEFAULT 11 clock_nanosleep@@GLIBC_2.17
1577: 000bda48 121 FUNC GLOBAL DEFAULT 11 __clock_nanosleep@@GLIBC_PRIVATE
1651: 000aa8f8  43 FUNC GLOBAL DEFAULT 11 usleep@@GLIBC_2.0
1959: 00085564 542 FUNC WEAK   DEFAULT 11 sleep@@GLIBC_2.0
```

Following that, we can see the consistency of code segment between program and process.

6. Exercise

6.1. Questions

1. Describe the return values of *fork()*

2. Run the example in Section 4 multiple times and observe the output of each time. Investigate the stability of the outputs and provide reasons as well as solutions for this phenomenon. (Your solution must ensure that the output must always be "Hello, World!")

3. What will the output be at LINE A?

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

int value = 5;

int main()
{
    pid t pid;
    pid = fork();
    if (pid == 0) {        /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINE A */
        return 0;
    }
}
```

4. When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?
   A. Stack
   B. Heap
   C. Shared memory segments

6.2. Programming exercises (required)

Problem 1 Given a file named "*numbers.txt*" containing N (0 < N < 1000) multiple lines of text. Each line is a non-negative integer. Write a C program that reads integers listed in this file and stores them in an array (or linked list). The program then uses the $fork()$ system call to create a child process. The parent process will count the numbers of integers in the array that are divisible by 2. The child process will count numbers divisible by 3. Both processes then send their results to the **stdout**. For examples, if the file "*numbers.txt*" contains the following lines

```
12
3
4
10
11
```

After executing the program, we must see the following lines on the screen (in any order)

```
3
2
```

Problem 2 The relationship between processes could be presented by a tree. When a process uses $fork$ system call to create another process then the new process is a $child$ of this process. This process is the parent of the new process. For examples, if process A uses two $fork$ system calls to create two new processes B and C then we can illustrate their relationship by a tree in figure 6.1. B is a child process of A. A is the parent of both B and C.
Write a program that uses $fork$ system calls to create processes whose relationship is similar to the one showed in Figure 6.2. Note: If a process has multiple children then its children must

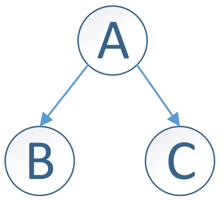

Figure 6.1: Creating processing with $fork()$.

be created from left to right. For example, process A must creates B first then create C and finally D.

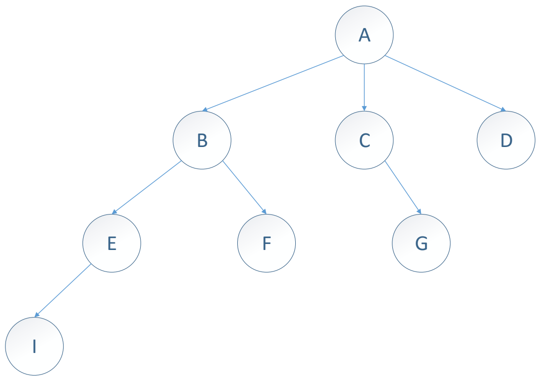

Figure 6.2: The tree of running processes.

SUBMISSION The source code for problem 1 and 2 must be written in two single files named "$ex1.c$" and "$ex2.c$, respectively. Those files are placed in a directory whose name is your Student ID. Students must also write a report and put it in that folder. Before submitting your work, please compress this directory using the ZIP format (has .zip extension) and name the compression file by your Student ID. You must submit the ZIP file to Sakai.

A. System Call wait()

This example uses system call to guarantee the order of running processes.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>      /* defines fork(), and pid_t. */

int main(int argc, char ** argv) {

  pid_t child_pid;

  /* lets fork off a child process... */
  child_pid = fork();

  /* check what the fork() call actually did */
  if (child_pid == -1) {
        perror("fork");
        exit(1);
  }

  if (child_pid == 0) {
    /* fork() succeeded, we're inside the child process */
    printf("Hello, ");
    fflush(stdout);
  }
  else {
    /* fork() succeeded, we're inside the parent process */
    wait(NULL);          /* wait the child
exit */
    printf("World!\n");
    fflush(stdout);
  }

  return 0;
}
```

B. Signal

This example illustrates the using of IPC `signal()` and `kill()` routines to suspend child process until its parent process is finished.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>      /* defines fork(), and pid_t. */

int main(int argc, char ** argv) {

  pid_t child_pid;
  sigset_t mask, oldmask;

  /* lets fork off a child process... */
  child_pid = fork();

  /* check what the fork() call actually did */
  if (child_pid == -1) {
        perror("fork"); /* print a system-defined error
message */
        exit(1);
  }

  if (child_pid == 0) {
    /* fork() succeeded, we're inside the child process */
    signal(SIGUSR1, parentdone);          /* set up a signal */
    /* Set up the mask of signals to temporarily block. */
    sigemptyset (&mask);
    sigaddset (&mask, SIGUSR1);

    /* Wait for a signal to arrive. */
    sigprocmask (SIG_BLOCK, &mask, &oldmask);
    while (!usr_interrupt)
      sigsuspend (&oldmask);
    sigprocmask (SIG_UNBLOCK, &mask, NULL);


    printf("World!\n");
    fflush(stdout);
  }
  else {
    /* fork() succeeded, we're inside the parent process */
    printf("Hello,␣");
```

```
        fflush(stdout);
        kill(child_pid, SIGUSR1);
        wait(NULL);
    }

    return 0;
}
```