HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2
Course: Operating Systems

March 23, 2021

Goal: This lab helps student to understand the definition of process, and how an operating system can manage the execution of a process.

Content In detail, this lab reflects the theory of process into practical exercises. For example:

- How to identify the memory regions of process's data segment, including: Data segment, BSS segment, Stack, Heap?
- How to retrieve the information of running process? How is PCB table controlled by OS?
- Create a program with multiple processes.

Result After doing this lab, student can distinguish a program and a process. They can create a program with multiple process and retrieve the information of processes.

Requirement Student need to review the theory of process in operating system.

1. Process's memory regions

Traditionally, a Unix process is divided into segments. The standard segments are code segment, data segment, BSS (block started by symbol), and stack segment.

The code segment contains the binary code of the program which is running as the process (a "process" is a program in execution). The data segment contains the initialized global variables and data structures. The BSS segment contains the uninitialized global data structures and finally, the stack segment contains the local variables, return addresses, etc. for the particular process.
Under Linux, a process can execute in two modes - user mode and kernel mode. A process usually executes in user mode, but can switch to kernel mode by making system calls. When a process makes a system call, the kernel takes control and does the requested service on behalf of the process. The process is said to be running in kernel mode during this time. When a process is running in user mode, it is said to be "in userland" and when it is running in kernel mode it is said to be "in kernel space". We will first have a look at how the process segments are dealt with in userland and then take a look at the book keeping on process segments done in kernel space.
In Figure 1.1, blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. The distinct bands in the address space correspond to memory segments like the heap, stack, and so on.

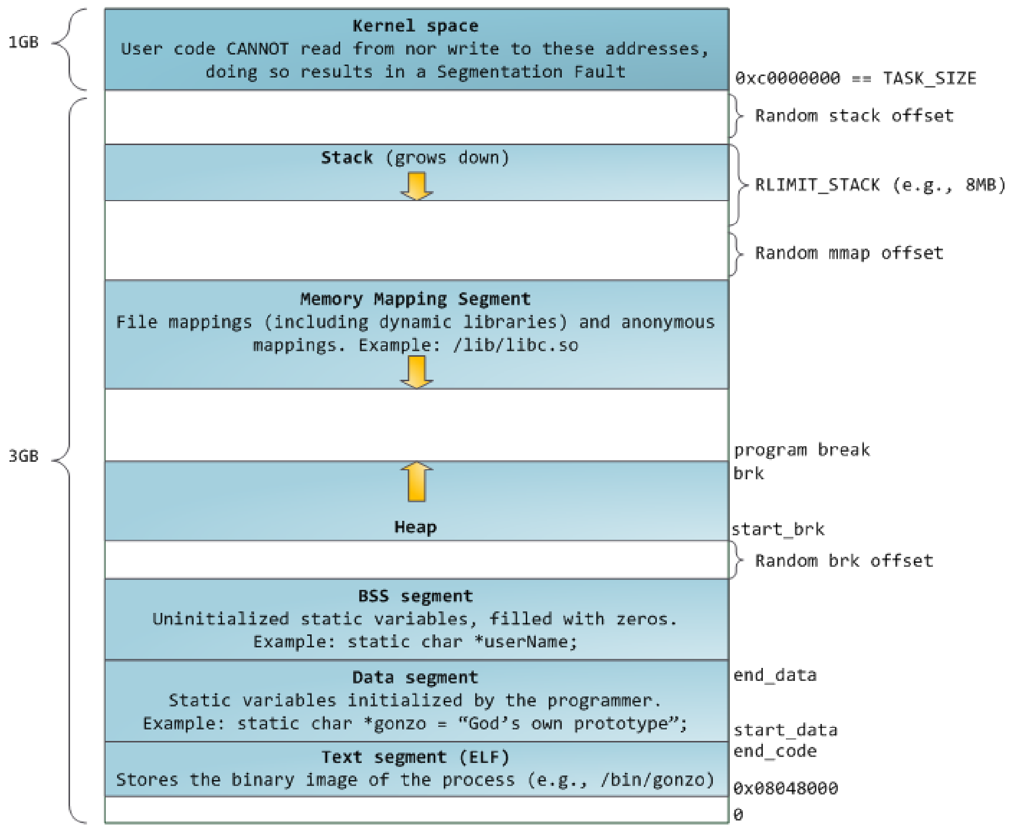
1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Figure 1.1: Layout of memory segments with process.

Userland's view of the segments

- The Code segment consists of the code - the actual executable program. The code of all the functions we write in the program resides in this segment. The addresses of the functions will give us an idea where the code segment is. If we have a function func() and let p be the address of func() (p = &func;). We know that p will point within the code segment.
- The Data segment consists of the initialized global variables of a program. The Operating system needs to know what values are used to initialize the global variables. The initialized variables are kept in the data segment. To get the address of the data segment we declare a global variable and then print out its address. This address must be inside the data segment.
- The BSS consists of the uninitialized global variables of a process. To get an address which occurs inside the BSS, we declare an uninitialized global variable, then print its address.
- The automatic variables (or local variables) will be allocated on the stack, so printing out the addresses of local variables will provide us with the addresses within the stack segment.
- A process may also include a heap,which is memory that is dynamically allocated during process run time.

1.1. Looking inside a process

Looking at the following C program with basic statements:

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

int glo_init_data = 99;
int glo_noninit_data;

void print_func(){
int local_data = 9;
printf("Process ID = %d\n", getpid());
printf("Addresses of the process:\n");
printf("1. glo_init_data = %p\n", &glo_init_data);
printf("2. glo_noninit_data = %p\n", &glo_noninit_data);
printf("3. print_func() = %p\n", &print_func);
printf("4. local_data = %p\n", &local_data);
}

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

```
print_func();
return 0;
}
```

Let's run this program many times and give the discussion about the segments of a process. Where is data segment/BSS segment/stack/code segment?

1.2. Dynamic allocation on Linux/Unix system

1.2.1. malloc

malloc() is a Standard C Library function that allocates (i.e. reserves) memory chunks. It compiles with the following rules:

- malloc allocates at least the number of bytes requested
- The pointer returned by malloc points to an allocated space (i.e. a space where the program can read or write successfully)
- No other call to malloc will allocate this space or any portion of it, unless the pointer has been freed before.
- malloc should be tractable: malloc must terminate in as soon as possible.
- malloc should also provide resizing and freeing.

1.2.2. Process's memory and heap

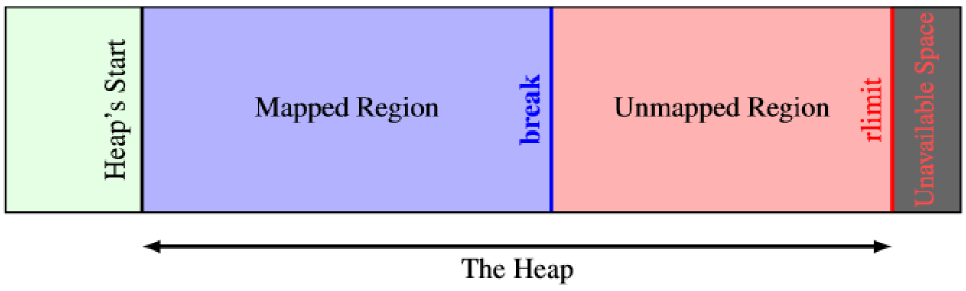

Figure 1.2: Heap region.

The heap is a continuous (in term of virtual addresses) space of memory with three bounds: a starting point, a maximum limit (managed through sys/ressource.h's functions getrlimit(2) and setrlimit(2)) and an end point called the break. The break marks the end of the mapped memory space, that is, the part of the virtual address space that has correspondence into real memory. Figure 1.2 sketches the memory organization.

Write a simple program to check the allocation of memory using malloc(). The heap is as large as the addressable virtual memory on computer architecture. The program checks the maximum usable memory per process.

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

int main(int argc,char* argv[]){
size_t MB = 1024*1024; // # of bytes for allocating
size_t maxMB = 0;
void *ptr = NULL;

do{
if(ptr != NULL){
printf("Bytes of memory checked = %zi\n",maxMB);
memset(ptr,0,maxMB); // fill the allocated region
}
maxMB += MB;
ptr = malloc(maxMB);
}while(ptr != NULL);

return 0;
}
```

In order to code a malloc(), we need to know where the heap begin and the break position, and of course we need to be able to move the break. This is the purpose of the two system calls brk() and sbrk().

1.2.3. brk(2) and sbrk(2)

We can find the description of these syscalls in their manual pages:

```c
int brk(const void *addr);
void* sbrk(intptr_t incr);
```

brk(2) places the break at the given address *addr* and return 0 if successful, -1 otherwise. The global *errno* symbol indicates the nature of the error.
sbrk(2) moves the break by the given increment (in bytes.) Depending on system implementation, it returns the previous or the new break address. On failure, it returns (void *)-1 and set *errno*. On some system sbrk() accepts negative values (in order to free some mapped memory.)
Implement a simple malloc() function with sbrk(). The idea is very simple, each time malloc is called we move the break by the amount of space required and return the previous address of the break. This malloc waste a lot of space in obsolete memory chunks. It is only here for educational purpose and to try the sbrk(2) syscall.

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

void *simple_malloc(size_t size)
{
void *p;
p = sbrk (0);
/* If sbrk fails , we return NULL */
if (sbrk(size) == (void*)-1)
return NULL;
return p;
}
```

2. Process Control Block

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor‘s registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is 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.

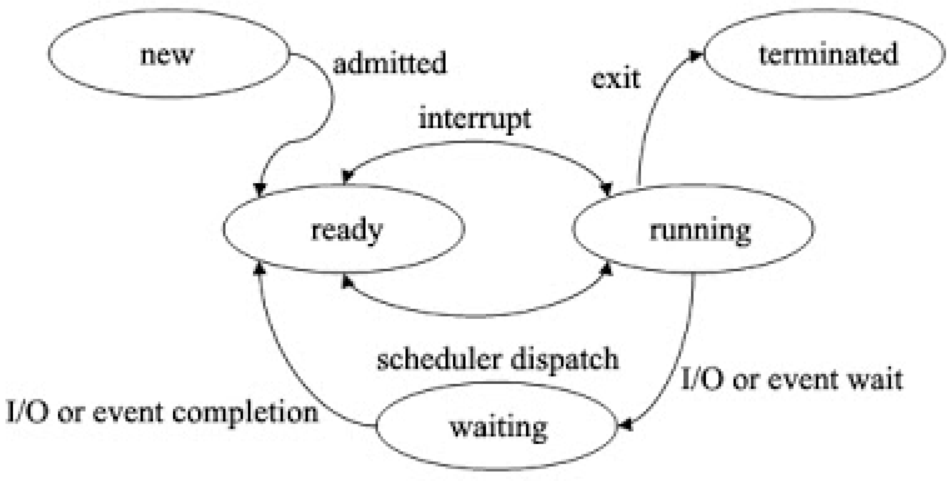

Figure 2.1: Diagram of process state.

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.

2.1. How do we find the process'ID and group?

A process can find its process ID with the *getpid* system call. It can find its process group number with the *getpgrp* system call, and it can find its parent's process ID with *getppid*. 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;
}
```

2.2. Creating a process

The *fork* system call clones a process into two processes running the same code. *Fork* returns a value of 0 to the child and a value of the process ID number (pid) to the parent. A value of -1 is returned on 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:
                printf("I am the parent: pid=%d\n", getpid());
                break;
        case -1:
                perror("Fork failed");
        }
        return 0;
}
```

2.3. Basics of Multi-process programming

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

```
#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 observing the output of the program and giving the conclusion about the order of letters “Hello, World!”.

This is because of the independent among porcesses, so that the order of execution is not guaranteed. It need a mechanisim to suspend the process until its child process finised. The system call `wait()` is used in this case. The example of `wait()` is presented in appendix A.
In case of expanding the result to implement the reverse order which child process is suspended until its father process is finished. This case is not recommended due to the `orphan process` status. An example of such programs is presented in appendix B

2.4. Retrieve the code segment of process

2.4.1. The information of process

You need a program with the execution time being enough long.

```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 (process ID - 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
```

Using 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
...
```

2.4.2. 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 file 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.

3. Exercise

3.1. Questions

(2 points)

1. What the output will be at LINE A? Does this value not change for every process executions? Explain your answer.
2. What the output will be at LINE C? Explain your answer.
3. Remove line B from the program. Observe your display result onto the screen and give your remark.

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

int value = 19;

int main()
{
   pid\_t pid;
   pid = fork();
   if (pid == 0) {        /* child process */
        value += 15;
        printf("PID of child process = %d \n",getpid()); /* LINE A */
        return 0;
   }
   else if (pid > 0) { /* parent process */
        wait(NULL); /* LINE B */
        printf("PARENT: value = %d \n",value);   /* LINE C */

        return 0;
   }
}
```

3.2. Programming exercises (required)

Problem 1 (5 points) Given a file named "*numbers.txt*" containing 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

```
15
18
8
10
21
33
32
35
```

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

```
Parent: <your_result>
Child: <your_result>
```

Problem 2 (3 points) The relationship between processes could be represented 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 could display their relationship by a tree in figure 3.1. B is a child process of A. A is the parent of both B and C.

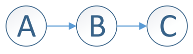

Figure 3.1: Creating processing with *fork*().

Write a program that uses *fork* system calls to create processes whose relationship is similar to the one showed in Figure 3.2. Note: If a process has multiple children then its children must be created from left to right. For example, process A must creates B first then create C and finally D.

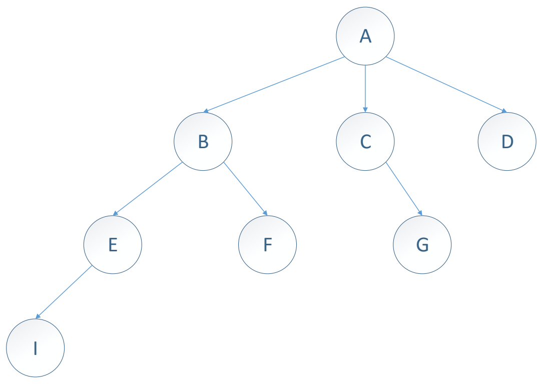

Figure 3.2: The tree of running processes.

Submission The source code for problem 1 and 2 must be written in two single files named "*ex*1.*c*" and "*ex*2.*c*, respectively. Those file are placed in a directory whose name is your STUDENTID-<ex...>.Before submitting your work, please compress this directory in ZIP format (has .zip extension) and name the compression file by your StudentID. You must submit the ZIP file to BKEL.

A. System Call wait()

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

```
#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 illustrate 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, ");
```

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

    return 0;
}
```