HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2: Process
Course: Operating Systems

March 22, 2024

Goal This lab helps student to practice with the process in OS, and understand how we can manipulate process and perform multi-process interaction using communication mechanism.

Contents In detail, this lab requires student practice with examples `fork()` API to create processes and do interaction with these instances through the following experiments:

- retrieve the process information (with PID) and determine the process status in its life cycle
- examine the process memory regions
- create multi-process program and practice inter-process communication (IPC)
- create a multi-thread process using POSIX Pthread

Besides, the practices also introduces and includes some additional process interfaces i.e. environment, system call, arguments, IO and files.

Result After doing this lab, student can understand the definition of process and write a program with multi-process creation and communication.

Requirements Student need to review the theory of program and how to create a process from the associated program by executing it.

```
$ gcc -o hello hello.c
$ ./hello
$ ps auxf | grep hello
```

Contents

1 Background

In this section, we recall the basic background material which is related to the process experiment.

- Process concept: program in execution and each process has an unique PID
- Process control block and one partial implementation in task_struct.
- Process state and it life cycle diagram.
- Memory layout.
- A tree of process.
- Process environment

Process ID - pid Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or PID) which is typically an integer number.

Task struct the task_struct structure contains all the information about a process. Much information is investigated in this lab experiments.

- Represented by the C structure **task_struct**

```
pid t_pid;                          /* process identifier */
long state;                         /* state of the process */
unsigned int time_slice             /* scheduling information */
struct task_struct *parent;         /* this process's parent */
struct list_head children;          /* this process's children */
struct files_struct *files;         /* list of open files */
struct mm_struct *mm;       /* address space of this process */
```

Figure 1: Task struct representation

Process state The process state is represented in the following diagram. We can send a signal to a process to change it status. The detailed commands are introduced in section 2.3.2.

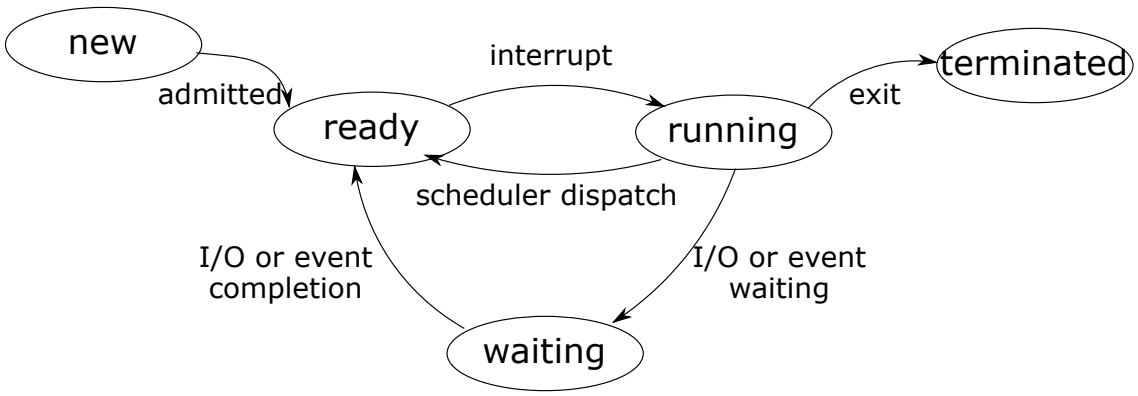

Figure 2: Diagram of process state

Memory layout The memory layout of a process is typically divided into multiple section

- Text section: the executable code
- Data section: global variables
- Heap: memory allocated dynamically during program running
- Stack: temporary data storage during function invoking

We hava an examination in each section using the variable declaration in different program scope to illustrated the memory layout as Figure.3 in section 2.4

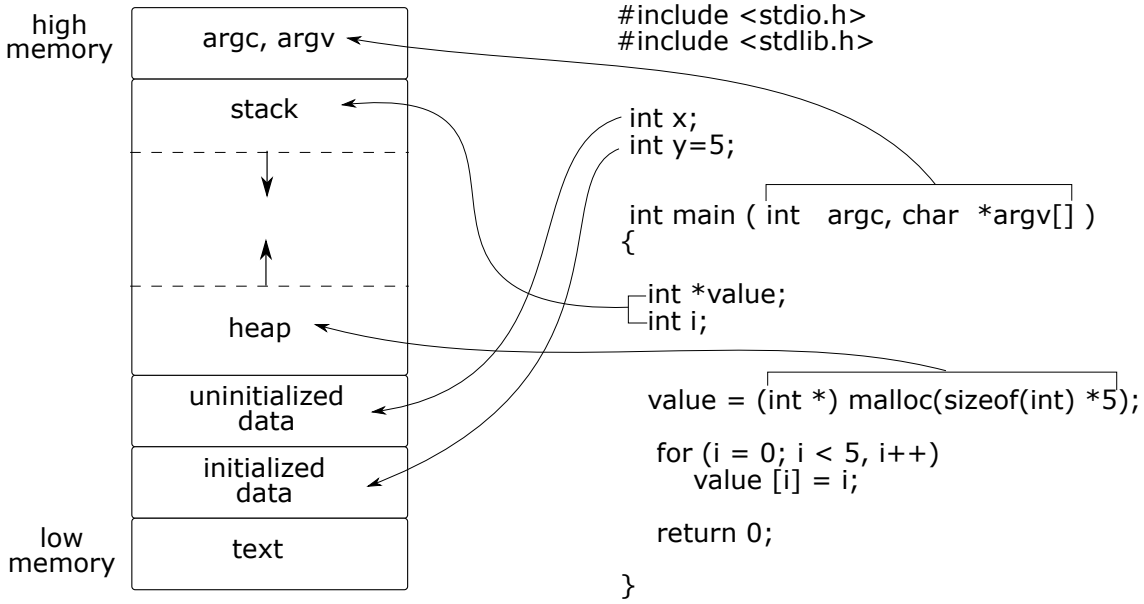

Figure 3: Memory layout of C program

A tree of process During the OS execution, a process, called parent process, may create several new processes which are referred as children of that process. Each of these new (child) processes may in turn create other processes, forming a tree of processes. An example of process tree is shown in Figure.4 and will be investigated in section 2.5

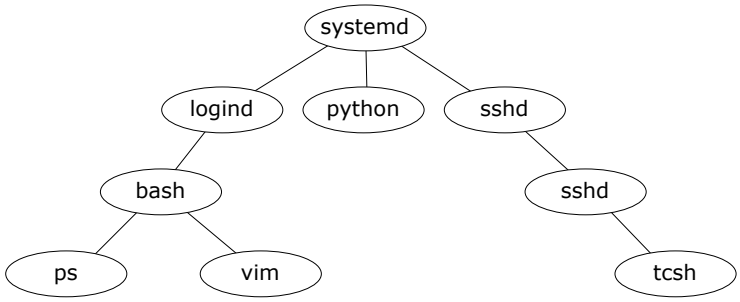

Figure 4: A tree of process in Linux

Multi processes and multi-thread process

IPC - InterProcess Communication There are two fundamental models of interprocess communication: shared memory and message passing.

In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

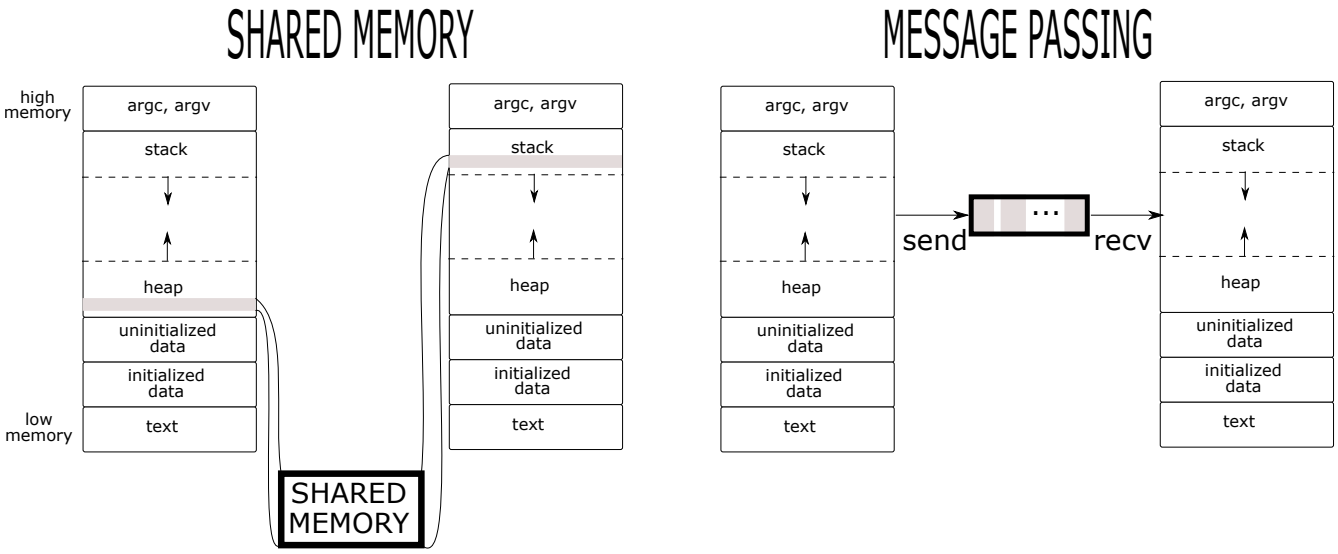

Figure 5: Two type of communication models

POSIX thread A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 6 illustrates the difference between a **traditional single-threaded process** and a **multi-threaded process**.

We use the POSIX thread library (Pthread) to create additional thread inside a traditional single-thread process. The details instructions and guidelines are at section 2.8.

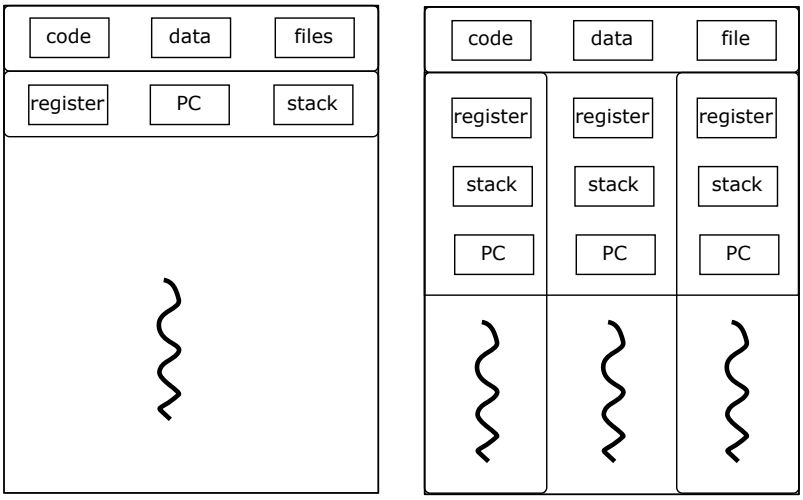

Figure 6: A traditional single-threaded process and a multi-threaded process

2 Programming interface

2.1 Fork API

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent

```
#include <unistd.h>
pid_t fork(void);
```

An example of fork calling program

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

int main(int argc, char* argv[])
{
    int pid;

    printf("Start of main...\n");

    pid = fork();
    if (pid > 0) {
        /*parent process*/
        printf("Parent section...\n");
    }
    else if (pid == 0) {
        /*child process*/
        printf("\nfork created...\n");
    }
    else {
        /*fork creation faile*/
        printf("\nfork creation failed!!!\n");
    }

    return 0;
}
```

2.2 Proc FS

ProcFS presents the information about processes and other system information. It provides a more convenience and standardized method for dynamically accessing process data held in kernel instead of tracing and direct acessing to kernel memory. For example, the GNU version of processing report ultility ps used the proc filesystem to obtain data, without using any specialized system calls.

We can retrieve various information in read-only part of /proc file system:

1. Process-specific subdirectories (/proc/PID)
2. Kernel info in /proc/
3. Network info in /proc/net

4. SCSI info in /proc/scsi
5. Parallel port info in /proc/parport
6. TTY info in /proc/tty
7. Miscellaneous kernel statistic in /proc/stat
8. Filesystem info in /proc/fs/¡FS_ID¿
9. Console info in /proc/console

Each process is mapped to a process-specific subdirectory a under the path associated with its Pid as **/proc/<pid>**

2.3 Process specification

By using the cat, more, or less commands on files within the /proc/ directory, users can immediately access enormous amounts of information about the system.

The specification of /proc entries:

File	Content
clear_refs	Clears page referenced bits shown in smaps output
cmdline	Command line arguments
cpu	Current and last cpu in which it was executed (2.4)(smp)
cwd	Link to the current working directory
environ	Values of environment variables
exe	Link to the executable of this process
fd	Directory, which contains all file descriptors
maps	Memory maps to executables and library files (2.4)
mem	Memory held by this process
root	Link to the root directory of this process
stat	Process status
statm	Process memory status information
status	Process status in human readable form
pagemap	Page table
stack	A symbolic trace of the process's kernel stack
...	...

2.3.1 Process state

filepath /proc/[pid]/status Provides much of the information in a format that's easier for humans to parse. The state is under file path /proc/<pid>/status

```
$ head -n 10 /proc/<pid>/status
Name:   helloworld
State:  S (sleeping)
Tgid:   1163
Ngid:   0
```

```
Pid:    1163
PPid:   1162
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
...
```

The fields are as follows:

Name Command run by this process

State Current state of the process "R (running)", "S (sleeping)", "D (disk sleep)", "T (stopped)", "t (tracing stop)", "Z (zombie)", or "X (dead)".

Pid Thread ID PPid PID of parent process.

2.3.2 Send signal to specified process

The command kill sends the specified signal to the specified processes or process groups. If no signal is specified, the TERM signal is sent.

```
$ kill -SIGNAL <pid>

$ killall -SIGNAL name
```

For example:

kill -SIGCONT 2378

killall hello

The list of support sign is as follows:

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

2.3.3 Process statistics environment

filepath /proc/[pid]/stat information about the process. This is used by **ps**.

```
$ cat /proc/<pid>/stat
```

filepath /proc/[pid]/statm Provides information about memory usage, measured in pages.

```
$ cat /proc/<pid>/statm
```

filepath /proc/[pid]/stack This file provides a symbolic trace of the function calls in this process's kernel stack.

```
$ cat /proc/<pid>/stack
```

filepath /proc/[pid]/environment This file contains the initial environment that was set when the currently executing program was started.

```
$ strings /proc/<pid>/environ
```

2.4 Proces memory layout

filepath /proc/[pid]/maps A file containing the currently mapped memory regions and their access permissions.

```
$ cat /proc/<pid>/maps
```

There are additional helpful pseudo-paths:
[stack] The initial process's (also known as the main thread's) stack.
[heap] The process's heap.
An example of the output

```
00400000-00401000 r-xp 00000000                  /home/.../multivar_heap
00600000-00601000 r--p 00000000                  /home/.../multivar_heap
00601000-00602000 rw-p 00001000                  /home/.../multivar_heap
024a2000-024c3000 rw-p 00000000                  [heap]
7f61f1996000-7f61f1b54000 r-xp 00000000          /lib/.../libc-2.19.so
7f61f1b54000-7f61f1d54000 ---p 001be000          /lib/.../libc-2.19.so
7f61f1d54000-7f61f1d58000 r--p 001be000          /lib/.../libc-2.19.so
7f61f1d58000-7f61f1d5a000 rw-p 001c2000          /lib/.../libc-2.19.so
7f61f1d5a000-7f61f1d5f000 rw-p 00000000
7f61f1d5f000-7f61f1d82000 r-xp 00000000          /lib/.../ld-2.19.so
7f61f1f78000-7f61f1f7b000 rw-p 00000000
7f61f1f80000-7f61f1f81000 rw-p 00000000
7f61f1f81000-7f61f1f82000 r--p 00022000          /lib/.../ld-2.19.so
7f61f1f82000-7f61f1f83000 rw-p 00023000          /lib/.../ld-2.19.so
7f61f1f83000-7f61f1f84000 rw-p 00000000
7ffe4ccea000-7ffe4cd0b000 rw-p 00000000          [stack]
7ffe4cd90000-7ffe4cd93000 r--p 00000000          [vvar]
7ffe4cd93000-7ffe4cd95000 r-xp 00000000          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 [vsyscall]
```

2.5 Process Tree

pstree is a Linux command that shows the running processes as a tree

```
$ pstree
init-+-cron
     |-dbus-daemon
     |-dhclient
     |-4*[getty]
     |-login---bash---pstree
     |-login---bash---msgrcv
     |-rsyslogd---3*[{rsyslogd}]
     |-sshd---2*[sshd---sshd---sftp-server]
     |-systemd-logind
     |-systemd-udevd
     |-upstart-file-br
     |-upstart-socket-
     `-upstart-udev-br
```

In addition, we can use the process-specific information retrieved from /proc filesystem to identify the parent pid and the list of child pid.

From child process, we can get the ppid (the PID of parent process)

```
$ head -n 10 <pid>
```

Meanwhile, the parent process can retrive the list of its child list under the pathname

```
$ cat /proc/<pid>/task/<pid>/children
```

By getting the ppid and the children list we can traverse through the process tree manually.

2.6 Inter-process Communication Programming Interfaces

POSIX Interprocess Communication (IPC) is a variation of System V interprocess communication. These interfaces are included in a set of programming interface which allow a programmer to coordinate activities among various program processes. We introduces the two basic illustrations of IPC mechanism which are Shared memory and Message Passing.

2.6.1 Shared Memory

shmget allocates a System V shared memory segment

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

shmat attaches the System V shared memory segment identified by *shmid* to the address space of the calling process

```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

shmdt detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process.

```
#include <sys/shm.h>

int shmdt(const void *shmadd
```

2.6.2 Message Passing - An illustration of Message Queue

To illustrate the message passing mechanism, we using here the message queue library.

msgget This system call creates or allocates a System V message queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

- The first argument, *key*, recognizes the message queue. The key can be either an arbitrary value.
- The second argument, *shmflg*, specifies the required message queue flags such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

msgbuf - The message buffer structure the structure of message is defined the following form:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

- The variable mtype is used for communicating with different message types
- The variable mtext is an array or other structure whose size is specified by msgsz (positive value).

msgsnd This system call sends/appends a message into the message queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgsnd(int msgid, const void *msgp, size_t msgsz,
           int msgflg)
```

- The first argument, **msgid**, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, **msgp**, is the pointer to the message, sent to the caller, defined in the structure of msgbuf
- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

msgrcv This system call retrieves the message from the message queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

- The first argument, **msgid**, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, **msgp**, is the pointer of the message received from the caller. It is defined in the structure of msgbuf
- The third argument, **msgsz**, is the size of the message received (message should end with a null character)
- The fourth argument, **msgtype**, indicates the type of message.
  - If msgtype is 0 (or NULL): Reads the first received message in the queue
- The fifth argument, **msgflg**, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes)

msgctl The system call performs control operations of the message queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf)
```

- The first argument, **msgid**, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, **cmd**, is the command to perform the required control operation on the message queue. Valid values for cmd are

2.7 Mapped memory

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

The **mmap**() function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor **fd** into memory, preferably at address **start**. If start is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. (In the case that start is not NULL, you can read more details in https://man7.org/linux/man-pages/man2/mmap.2.html).

The prot argument is used to determine the access permissions of this process to the mapped memory. The available options for prot are as below.

Option	Integer Value	Description
PROT_READ	1	Read access is allowed.
PROT_WRITE	2	Write access is allowed. Note that this value assumes PROT_READ also.
PROT_NONE	3	No data access is allowed.
PROT_EXEC	4	This value is allowed, but is equivalent to PROT_READ.

The **flags** argument is used to control the nature of the map. The following are some common options of flags.

Flag	Description
MAP_SHARED	This flag is used to share the mapping with all other processes, which are mapped to this object. Changes made to the mapping region will be written back to the file.
MAP_PRIVATE	When this flag is used, the mapping will not be seen by any other processes, and the changes made will not be written to the file.
MAP_ANONYMOUS / MAP_ANON	This flag is used to create an anonymous mapping. Anonymous mapping means the mapping is not connected to any files. This mapping is used as the basic primitive to extend the heap.
MAP_FIXED	When this flag is used, the system has to be forced to use the exact mapping address specified in the address If this is not possible, then the mapping will fail.

2.8 POSIX Thread (pthread) library

pthread_create - create a new thread

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of **start_routine**().

The new thread terminates in one of the following ways:

It calls pthread_exit(), specifying an exit status value that is available to another thread in the same process that calls pthread_join().

It returns from start_routine(). This is equivalent to calling pthread_exit() with the value supplied in the return statement.

It is canceled (see pthread_cancel()).

Some basic routines are available in pthread library:

- pthread_create()
- pthread_join()
- pthread_exit()

The calling procedure of POSIX library can be illustrated in Figure 7.

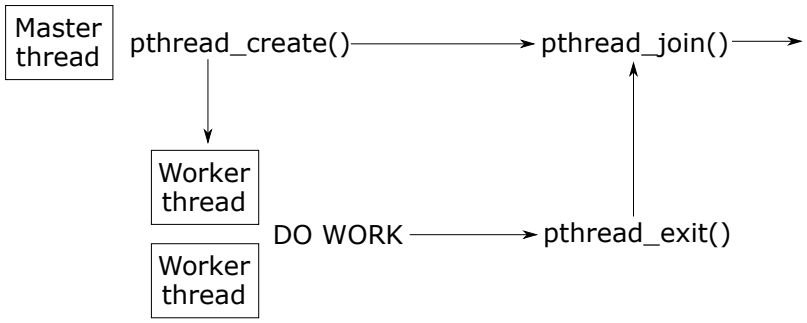

Figure 7: A calling procedure of pthread library's routines

An example of passing more than one-element argument to thread function.

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

struct student_t {
    char* name;
    int sid;
};

void *print_info(void *input) {
    printf("name: %s\n", ((struct student_t*)input)->name);
    printf("student ID: %d\n", ((struct student_t*)input)->sid);
}

int main() {
    struct student_t *John = (struct student_t *)malloc(sizeof(struct student_t));
    char jname[] = "John";
```

```
    John->name = jname;
    John->sid = 1122;

    pthread_t tid;
    pthread_create(&tid, NULL, print_info, (void *)John);
    pthread_join(tid, NULL);
    return 0;
}
```

3 Practices

3.1 Practice 1: Create process

Recall the experiment of creating a process with additional IO waiting

Step1 Create a program with source code "hello_wait.c"

```
#include <stdio.h>

int main(int argc, char* argv[])
{
        printf("Hello world\n");
        getc(stdin);

        return 0;
}
```

Step 2 Compile and execute the program to create process

```
$ gcc -o hello_wait hello_wait.c

$./hello_wait
```

Step 3 Retrieve the process Pid and it associated /proc folder

```
$ ps auxf | grep hello

$ ls /proc/<pid>
```

3.2 Practice 2: Traverse the tree of processes

We create a process based on our previous example and add additional call of fork.

Step 1 Implement the source code of "hello_fork.c"

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  fork();
  printf("Hello world\n");
  getc(stdin);

  return 0;
}
```

Step2 Compile and execute the created "hello_fork.c" program.

```
$ gcc -o hello_fork hello_fork.c
$./hello_fork
Hello world
Hello world
```

Step3 Get the Pid of the create process

```
$ ps auxf | grep hello_fork
oslab      3287  ...      \_ ./hello_fork
oslab      3288  ...         \_ hello_fork
```

Step4 Retrieve the information of parent and child processes:

```
$ head -n 10 /proc/<pid>/status
...
Pid: 3288
Ppid: 3287
...

$ cat /proc/<pid>/task/<pid>/children
3288
```

3.3 Practice 3: Examine the process memory regions

In this section, we try to touch the different regions in memory layout of the process

Step 1 Implement the source code of "multivar.c"

The source code of "multivar.c"

```
#include <stdio.h>

int glo_init_data = 99;
int glo_noninit_data;

void func(unsigned long number) {
  unsigned long local_data = number;

  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", &func);
  printf(" 4. local_data = %p\n", &local_data);
}
```

```
int main() {
  func(10);

  while (1)
    usleep(0);
}
```

Step 2 Compile and execute the two program separately.

```
$ gcc -o multivar multivar.c

$ ./multivar
Process ID = 1429
Addresses of the process :
 1. glo_init_data = 0x601058
 2. glo_noninit_data = 0x601068
 3. print_func ( ) = 0x40060d
 4. local_data = 0x7ffe08c57f68
```

Step 3 Get process memory mapping layout at /proc/<pid>/maps.

```
$ ps auxf | grep multivar

$ cat /proc/<pid>/maps
00400000-00401000 r-xp 00000000                  /home/.../multivar_heap
00600000-00601000 r--p 00000000                  /home/.../multivar_heap
00601000-00602000 rw-p 00001000                  /home/.../multivar_heap
024a2000-024c3000 rw-p 00000000                  [heap]
7f61f1996000-7f61f1b54000 r-xp 00000000          /lib/.../libc-2.19.so
7f61f1b54000-7f61f1d54000 ---p 001be000          /lib/.../libc-2.19.so
7f61f1d54000-7f61f1d58000 r--p 001be000          /lib/.../libc-2.19.so
7f61f1d58000-7f61f1d5a000 rw-p 001c2000          /lib/.../libc-2.19.so
7f61f1d5a000-7f61f1d5f000 rw-p 00000000
7f61f1d5f000-7f61f1d82000 r-xp 00000000          /lib/.../ld-2.19.so
7f61f1f78000-7f61f1f7b000 rw-p 00000000
7f61f1f80000-7f61f1f81000 rw-p 00000000
7f61f1f81000-7f61f1f82000 r--p 00022000          /lib/.../ld-2.19.so
7f61f1f82000-7f61f1f83000 rw-p 00023000          /lib/.../ld-2.19.so
7f61f1f83000-7f61f1f84000 rw-p 00000000
7ffe4ccea000-7ffe4cd0b000 rw-p 00000000          [stack]
7ffe4cd90000-7ffe4cd93000 r--p 00000000          [vvar]
7ffe4cd93000-7ffe4cd95000 r-xp 00000000          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 [vsyscall]
```

Results Notify the empty value of reader process output. Try to self explain based on the ordering of process execution to see something wrong when we execute the writer before the redear. Revese the order of reader/writer execution and see the updated result.

3.4 Practice 4: Inter Process Communication

3.4.1 Shared Memory

In this section, we implement 2 separated program called "reader.c" and "writer.c". These two programs are implemented in different source code files and have 2 main function. During their execution, we can get 2 different Pid and process information.

The purpose of this experiment is providing an illustration of the shared information through a **shared** memory region where each process can access separately. With a correct setting, we can transfer a message "hello world" between the two process.

Implement of message transferring we implement the writer process which set the value to the pre-shared memory region obtained by shmget() and shmat(). In the other side, we implement another process called reader get the value form the same memory region.

The experiment is performed following these steps:

Step 1 Implement the source code of "reader.c" and "writer.c"

The source code of "reader.c"

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Reader.c using the pre-shared key SHM_KEY 0x123
 *
 */

#define SHM_KEY 0x123

int main(int argc, char * argv[]) {
  int shmid;
  char * shm;
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("shmget");
    return 1;
  } else {
    printf("shared-memory-ID:--%d\n", shmid);
  }
  shm = (char * ) shmat(shmid, 0, 0);
  if (shm == (char * )- 1) {
    perror("shmat");
    exit(1);
  }
  printf("shared-memory-mm:--%p\n", shm);
  if (shm != 0) {
    printf("shared-memory-content:--%s\n", shm);
  }
  sleep(10);
  if (shmdt(shm) == -1) {
    perror("shmdt");
    return 1;
  }
  return 0;
}
```

The source code of "writer.c"

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>

/*
 * Writer.c using the pre-shared key SHM_KEY 0x123
 *
 */

#define SHM_KEY 0x123

int main(int argc, char * argv[]) {
  int shmid;
  char *shm;
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared-memory");
    return 1;
  } else {
    printf("Shared-memory-ID:--%d\n", shmid);
  }
  shm = (char *) shmat(shmid, 0, 0);

  if (shm == (char *) -1) {
    perror("shmat");
    exit(1);
  }
  printf("shared-memory-mm:--%p\n", shm);
  sprintf(shm, "hello--world\n");
  printf("shared-memory-content:--%s\n", shm);
  sleep(10);

  // detach from the shared memory
  if (shmdt(shm) == -1) {
    perror("shmdt");
    return 1;
  }
  // Mark the shared segment to be destroyed .
  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl");
    return 1;
  }
  return 0;
}
```

Step 2 Compile and execute the two program separately. Open one terminal for "reader.c"

```
$ gcc -o reader reader.c
$ ./reader
shared memory ID:   196608
shared memory mm:   0x7f45d3a73000
shared memory content:
```

Open another (different) terminal for "writer.c"

```
$ gcc -o writer writer.c
$ ./writer
Shared-memory ID:   131072
shared memory mm:   0x7f6f8c365000
shared memory content:   hello   world
```

Aftermath Recognize the different mm address of the different process since they are different program and hence, the local stack variable is placed in different layout. But by leveraging the shared memory technique, they "magically" can access to the exact same message content. Verify this result or further investigating by changing the message content and complete the experiment.

3.4.2 Message Passing - An illustration of Message Queue

We reproduce the same experiment in shared memory section except that the message is transferred using Message Queue in which it does not explicitly allocate memory to store the variable. Instead, it provides two basic operators called "send" and "receive" and the rest of data transferring mechanism is held by the system. Therefore, in this section, we use the two programs associated with their operation of sending/receiving and name "msgsnd.c"/"msgrcv.c"

Implement of message transferring we implement the writer process which set the value to the pre-shared memory region obtained by shmget() and shmat(). In the other side, we implement another process called reader get the value form the same memory region.

The experiment is performed following these steps:

Step 1 Implement the source code of "msgsnd.c" and "msgrcv.c"

The source code of "msgsnd.c"

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/*
 * Filename: msgsnd.c
 */

#define PERMS 0644
#define MSG_KEY 0x123
struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void) {
   struct my_msgbuf buf;
   int msqid;
   int len;
   key_t key;
   system("touch msgq.txt");

   if ((msqid = msgget(MSG_KEY, PERMS | IPC_CREAT)) == -1) {
      perror("msgget");
      exit(1);
   }
   printf("message queue: ready to send messages.\n");
   printf("Enter lines of text, ^D to quit:\n");
   buf.mtype = 1; /* we don't really care in this case */

   while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      len = strlen(buf.mtext);
      /* remove newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';
      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
      perror("msgsnd");
   }
   strcpy(buf.mtext, "end");
   len = strlen(buf.mtext);
   if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
   perror("msgsnd");

   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
   }
   printf("message queue: done sending messages.\n");
   return 0;
}
```

The source code of "msgrcv.c"

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

```c
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/*
 * Filename: msgrcv.c
 */

#define PERMS 0644
#define MSG_KEY 0x123
struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void) {
   struct my_msgbuf buf;
   int msqid;
   int toend;
   key_t key;

   if ((msqid = msgget(MSG_KEY, PERMS | IPC_CREAT)) == -1) { /* connect to the queue */
      perror("msgget");
      exit(1);
   }
   printf("message queue: ready to receive messages.\n");

   for(;;) { /* normally receiving never ends but just to make conclusion
             /* this program ends wuth string of end */
      if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
         perror("msgrcv");
         exit(1);
      }
      printf("recvd: \"%s\"\n", buf.mtext);
      toend = strcmp(buf.mtext,"end");
      if (toend == 0)
      break;
   }
   printf("message queue: done receiving messages.\n");
   system("rm msgq.txt");
   return 0;
}
```

Step 2 Compile and execute the two program separately. Open one terminal for "reader.c"

```
$ gcc -o msgsnd msgsnd.c
$ ./msgsnd
message queue: ready to send message
Enter lines of text, ^D to quit:
```

Open another (different) terminal for "writer.c"

```
$ gcc -o msgrcv msgrcv.c
$ ./msgrcv
message queue: ready to receive message
```

Results The input messages are sent from the msgsnd to the msgrcv.

Aftermath In this experiment, we have only one-way direction from msgsnd to msgrcv and this mode is officially called HALF-DUPLEX communication. There is another mode which support 2-way direction communication which we temporarily leave it to the exercise section.

3.5 Practice 5: Create thread using Pthread library

In this section, we use Pthread library to create a 2-thread program, then we execute it and listing the running thread.

Step 1 : We implement a program using Pthread to create 2 threads. Since the execution of these threads will be terminated at the end of the passed function, we insert an I/O waiting with **getc()** to keep them alive. Implement the source code of the program "hello_thread.c" as follows:

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

#define MAX_COUNT 10000
int count;

void *f_count(void *sid) {
  int i;
  for (i = 0; i < MAX_COUNT; i++) {
    count = count + 1;
  }
  printf("Thread %s: holding %d \n", (char *) sid, count);
  getc(stdin);
}

int main(int argc, char* argv[])
{
  printf("Hello world\n");
  pthread_t thread1,thread2;

  count = 0;
  /* Create independent threads each of which will execute function */
  pthread_create( &thread1, NULL, &f_count, "1");
  pthread_create( &thread2, NULL, &f_count, "2");


  // Wait for thread th1 finish
  pthread_join( thread1, NULL);

  // Wait for thread th1 finish
  pthread_join( thread2, NULL);
  getc(stdin);

  return 0;
}
```

Step2 Compile and execute the program "hello_thread.c". In this step, it need to remind that Pthread is 3rd party library in which it need an explicit declaration of library usage through *gcc* option **-pthread**

```
$ gcc -pthread -o hello_thread hello_thread.c
$./hello_thread
Hello world
Thread 2: holding 10000
Thread 1: holding 20000
```

Step3 Get the Pid of this process

```
$ ps auxf | grep hello_thread
oslab       3314   ...       \_ ./hello_thread
oslab       3353   ...       \_ grep --color=auto hello_thread
```

Results In this experiment, we expect to see that there is only one process hello_thread but it is existed two execution instances with the 3 different printing messages (remember the 3 messages of "Hello World", "Thread1:...", "Thread2*...". Verify this output and complete the experiment.

4 Exercise

4.1 Problem1

Firstly, downloading two text files from the url: https://drive.google.com/file/d/1fgJqOeWbJC4gh MKHkuxfIP6dh2F911-E These file contains the 100000 ratings of 943 users for 1682 movies in the following format:

userID <tab> movieID <tab> rating <tab> timeStamp
userID <tab> movieID <tab> rating <tab> timeStamp
...

Secondly, you should write a program that spawns two child processes, and each of them will read a file and compute the average ratings of movies in the file. You implement the program by using shared memory method.

4.2 Problem2

Given the following function:

$$\text{sum}(n) = 1 + 2 + \ldots + n$$

This is the sum of a large set including n numbers from 1 to n. If n is a large number, this will take a long time to calculate the sum(n). The solution is to divide this large set into pieces and calculate the sum of these pieces concurrently by using threads. Suppose the number of threads is numThreads, so the 1st thread calculates the sum of $\{1, \text{n/ numThreads }\}$, the 2 nd thread carries out the sum of $\{\text{n/ numThreads } +1, 2\text{n/ numThreads }\}, \ldots$

Write two programs implementing algorithm describe above: one serial version and one multi-thread version.

The program takes the number of threads and n from user then creates multiple threads to calculate the sum. Put all of your code in two files named "sum_serial.c" and "sum_multi-thread.c". The number of threads and n are passed to your program as an input parameter. For example, you will use the following command to run your program for calculating the sum of $1M$:

```
$./ sum_serial 1000000
$./sum_multi_thread 10 1000000
(#numThreads=10)
```

Requirement: The multi-thread version may improve speed-up compared to the serial version. There are at least 2 targets in the Makefile sum_serial and sum_multi-thread to compile the two program.

4.3 Problem3

Conventionally, message queue in the practice is used in a one-way communication method. However, we still can have some tricks to adapt it for two-way communication by using multi-thread mechanism.

4.4 Problem4

Use *mmap* to implement the mapping created file into local address space. After the address range mapping, use it as a demonstration for data sharing between the two processes.

Revision History

Revision	Date	Author(s)	Description
1.0	03.15	PD Nguyen	Document created
2	10.2022	HL La	Update lab content, practices and exercises
2.1	10.2023	PD Nguyen	Update message passing and exercises