HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2
Process & Multithreaded Process
Course: Operating Systems

October 10, 2022

Goal: This lab helps student to review the data segment of a process and distinguish the differences between thread and process.

Content In detail, this lab requires student identify the memory regions of process's data segment, practice with examples such as creating a multithread program, showing the memory region of threads:

- view process memory regions: Data segment, BSS segment, Stack and Heap.
- Write a multithread program.
- Solve an example which can run in parallel using thread.
- Show the differences between process and thread in term of memory region.

Result After doing this lab, students can understand the mechanism of distributing memory region to allocate the data segment for specific processes. In addition, they understand how to write a multithreaded program.

Requirement Student need to review the theory of process memory and thread.

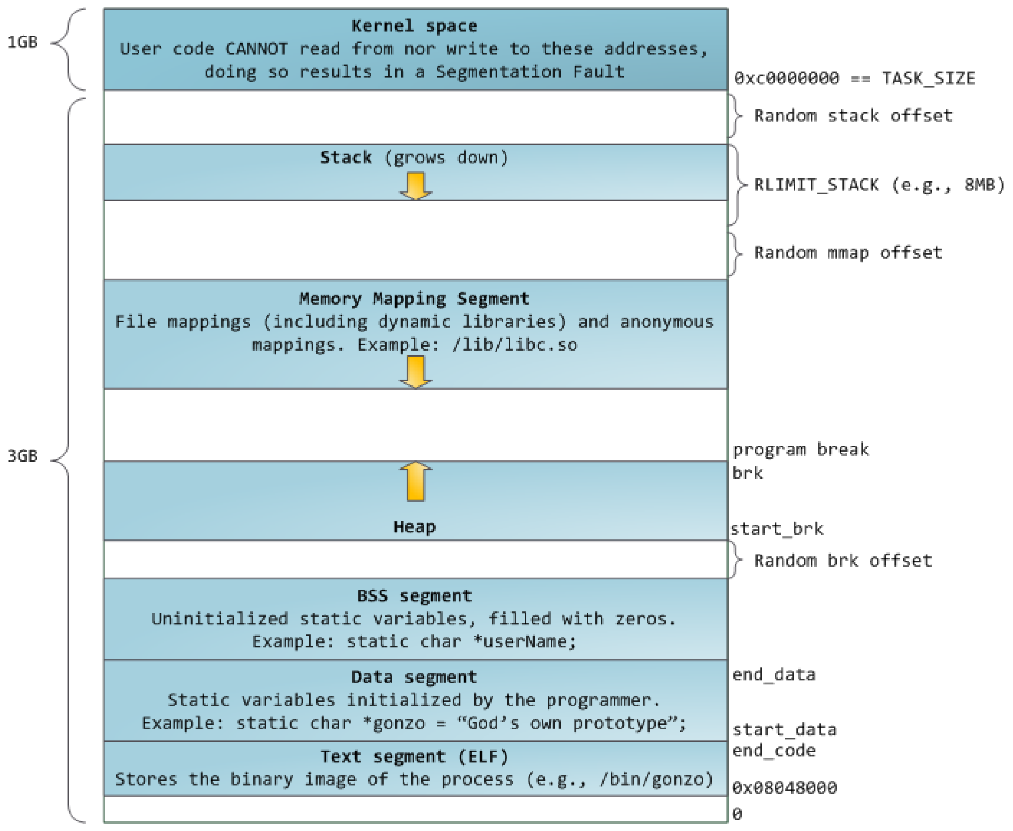
Contents

1. Introduction

1.1. Process 's memory

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
3GB
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
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.

1.2. Stack

Stack is one of the most important memory region of a process. It is used to store temporary data used by the process (or thread). The name "stack" is used to described the way data put and retrieved from this region which is identical to stack data structure: The last item pushed to the stack is the first one to be removed (popped).

Stack organization makes it suitable from handling function calls. Each time a function is called, it gets a new stack frame. This is an area of memory which usually contains, at a minimum, the address to return when it complete, the input arguments to the function and space for local variables.

In Linux, stack starts at a high address in memory and grows down to increase its size. Each time a new function is called, the process will create a new stack frame for this function. This frame will be place right after that of its caller. When the function returns, this frame is clean from memory by shrinking the stack (stack pointer goes up). The following program illustrates to identify the relative location of stack frames create by nested function calls.

Example: Trace function calls

```
#include <stdio.h>
void func (unsigned long number) {
        unsigned long local_f = number;
        printf("%2lu --> %p\n", local_f, &local_f);
        local_f--;
        if (local_f > 0) {
                func(local_f);
        }
}
int main() {
        func(10);
}
```

Similar to heap, stack has a pointer name stack pointer (as heap has program break) which indicate the top of the stack. To change stack size, we must modify the value of this pointer. Usually, the value of stack pointer is hold by stack pointer register inside the processor. Stack space is limited, we cannot extend the stack exceed a given size. If we do so, stack overflow will occurs and crash our program. To identify the default stack size, use the following command

```
ulimit −s
```

Different from heap, data of stack are automatically allocated and cleaned through procedure invocation termination, Therefore, in C programming, we do not need to allocate and free local variables. In Linux, a process is permitted to have multiple stack regions. Each regions belongs to a thread.

1.3. Interprocess Communication

Processes in OS are independent but we also need them cooperating. Cooperating process can affect or be affected by other processes, including sharing data. Some reasons for the need of cooperating processes are as follows:

- **Information sharing** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup** . If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Therefore, we need communication methods to transfer data among processes, which are also known as inter-process communication (IPC) protocols. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. The shell creates an *ls* process and a separate *lpr* process, connecting the two with a pipe, represented by the "|" symbol. A pipe permits one-way communication between two related processes. The *ls* process writes data into the pipe, and the *lpr* process reads data from the pipe. There are several IPC methods, in this lab, we will practice with two of them, namely shared memory and pipe. 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. The two communications models are contrasted in Figure 1.2. **In this lab, we will focus on the shared memory method.**

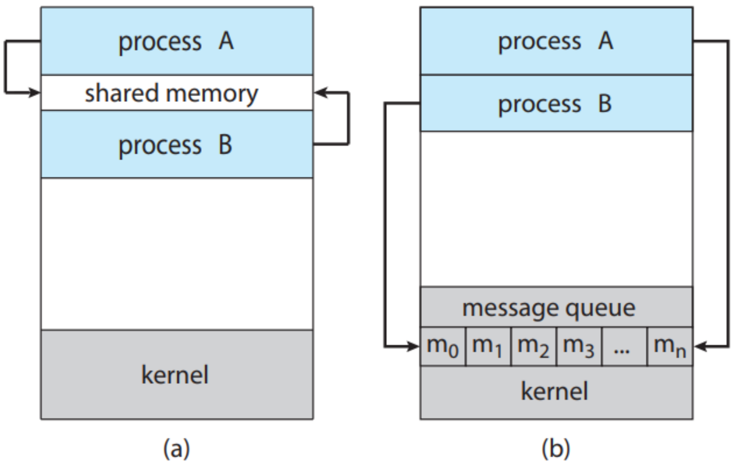


Figure 1.2: Shared memory vs message passing model.

1.4. Introduction to thread

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) 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.

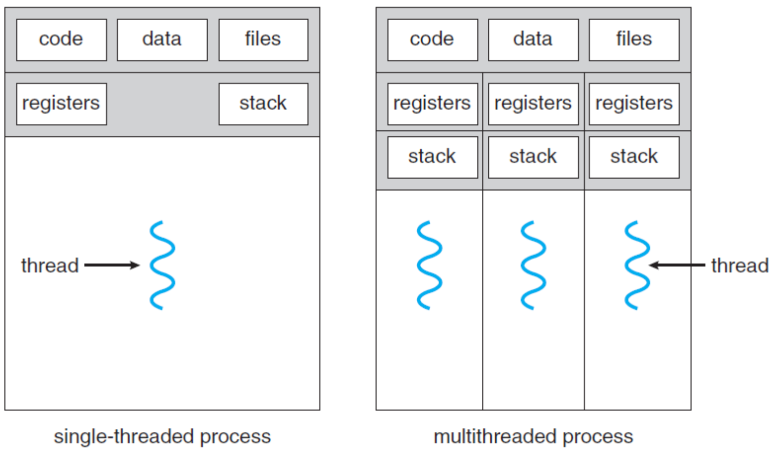


Figure 1.3: Single-threaded and multithreaded processes.

Figure 1.3 illustrates the difference between a traditional single-threaded process and a multithreaded process. The benefits of multithreaded programming can be broken down into four major categories:

- Responsiveness

- Resource sharing
- Economy
- Scalability

Question: What resources are used when a thread is created? How do they differ from those used when a process is created?

Multicore programming Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

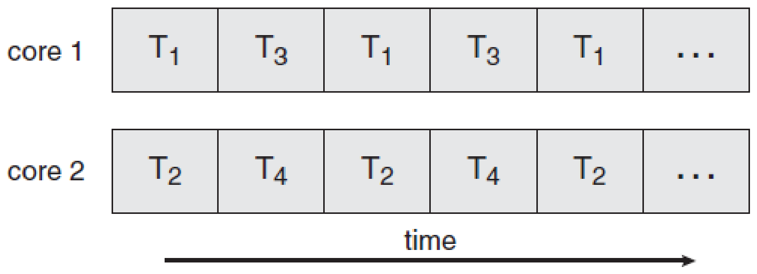

core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

Figure 1.4: Parallel execution on a multicore system.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core, as Figure 1.4 shown.
Question: Is it possible to have concurrency but not parallelism? Explain.

2. Practice

2.1. Looking inside a process

Looking at the following C program with basic statements:

```
#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?

2.2. How to transfer data between processes?

2.2.1. Shared Memory

A process allocates a shared memory segment using shmget ("SHared Memory GET").

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

- Its first parameter is an integer key that specifies which segment to create and unrelated processes can access the same shared segment by specifying the same key value. Moreover, other processes may have also chosen the same fixed key, which could lead to conflict. So that you should be careful when generating keys for shared memory regions. A solution is that you can use the special constant IPC_PRIVATE as the key value guarantees that a brand new memory segment is created.

- Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.

- The third parameter is the bitwise or of flag values that specify options to shmget. The flag values include these:
  - IPC_CREAT: This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.

  - IPC_EXCL: This flag, which is always used with IPC_CREAT, causes shmget to fail if a segment key is specified that already exists. If this flag is not given and the key of an existing segment is used, shmget returns the existing segment instead of creating a new one.
  - Mode flags: This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment.

To make the shared memory segment available, a process must attach it by calling shmat().

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

This function take three arguments as follows:

- the shared memory segment identifier SHMID returned by shmget().
- a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address.
- The third argument is a flag. You can read more details about this argument in Linux manual page. `https://man7.org/linux/man-pages/man3/shmat.3p.html`.

When you're finished with a shared memory segment, the segment should be detached using shmdt. Pass it the address returned by shmat. If the segment has been deallocated and this was the last process using it, it is removed. Examples: Run the two following processes in two terminals. At the writer process, you can type an input string and observe returns from the reader process.

- writer.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#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: %d\n", shmid);
  }

  shm = (char *)shmat(shmid, 0, 0);
  if (shm == (char *)-1) {
    perror("shmat");
    exit(1);
  }
  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;
}
```

- reader.c

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

#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: %d\n", shmid);
  }

  shm = (char *)shmat(shmid, 0,0);
  if (shm == (char *)-1) {
    perror("shmat");
    exit(1);
  }
  printf("shared memory: %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;
}
```

2.2.2. Synchronization Issues in Shared Memory

In the above programs, when there are only one writer and one reader, the problem may not occur. However, when there are several processes changing the shared memory at the same time, it can lead to **race condition** problems, which will be introduced more details in the next lab. In this lab, we will consider a solution for this problem by using Process Semaphore. The semaphore will lock the shared memory and only allow a process to access this segment at a particular time.
For example, the two following program will run in parallel. In particular, the writer1.c should be started first, and it will initialize a variable named data. This variable will contain a counter and a writerID. After both writers are started, they will print the values changed by each other.

- writer1.c

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

```
#include <semaphore.h>
#include <fcntl.h>
#include <stdbool.h>
#include<sys/stat.h>
#define SHM_KEY 0x123
#define SNAME "/semname"
struct shared_data{
    int counter;
    int writerID;
};
int main(int argc, char *argv[]){
  int shmid;
  struct shared_data *data;
  sem_t *sem = sem_open(SNAME, O_CREAT,0644);
  sem_init(sem, 0, 1);
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared memory");
    return 1;
  }else {
    printf("shared memory: %d\n", shmid);
  }
  if(sem == SEM_FAILED){
    printf("Sem failed\n");
    return -1;
  }

  data = (struct shared_data *)shmat(shmid, 0,0);
  if (data == (struct shared_data *)-1) {
    perror("shmat");
    exit(1);
  }
  data->counter=0;
  data->writerID=1;

  for(int i=0; i < 20; i++){
    sem_wait(sem);
    printf("Read from Writer ID: %d with counter: %d\n",
        data->writerID, data->counter);
    data->writerID = 1;
    data->counter++;
    sem_post(sem);
    sleep(1);
  }
  if (shmdt(data) == -1) {
    perror("shmdt");
    return 1;
  }

  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl");
    return 1;
  }
  // Remove the named semaphore
  if (sem_unlink(SNAME) < 0) perror("sem_unlink(3) failed");
  return 0;
}
```

- writer2.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>
#include <fcntl.h>
#include <stdbool.h>
#include<sys/stat.h>
#define SHM_KEY 0x123
#define SNAME "/semname"
struct shared_data{
    int counter;
    int writerID;
};

int main(int argc, char *argv[]){
  int shmid;
  struct shared_data *data;
  sem_t *sem = sem_open(SNAME, O_CREAT, 0644);
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared memory");
    return 1;
  }else {
    printf("shared memory: %d\n", shmid);
  }
  if(sem == SEM_FAILED){
    printf("Sem failed\n");
    return -1;
  }

  data = (struct shared_data *)shmat(shmid, 0,0);
  if (data == (struct shared_data *)-1) {
    perror("shmat");
    exit(1);
  }
  for(int i=0; i < 20; i++){
    sem_wait(sem);
    printf("Read from Writer ID: %d with counter: %d\n",
        data->writerID, data->counter);
    data->writerID = 2;
    data->counter++;
    sem_post(sem);
    sleep(1);
  }
  if (shmdt(data) == -1) {
    perror("shmdt");
    return 1;
  }
  sem_close(sem);
  return 0;
}
```

In the above programs, the semaphore named *sem* will be used to lock the code segment changing the values of shared variable. It 's noteworthy that we use semaphores' names to identify them between different processes.

2.2.3. Pipe

Pipe actually is very common method to transfer data between processes. For example, the "pipe" operator '|' can be used to transfer the output from a command to another command as in the following example:

```
# the output from "history" will be input to the grep command.
history | grep "a"
```

In terms of C programming, the standard library named "unistd.h" defined the following function to create a pipe. This function creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

```
int pipe(int pipefd[2]);
```

Pipe can be used for one-way communication as follows:

```
#include<stdio.h>
#include<unistd.h>
int main() {
    int pipefds[2];
    int returnstatus;
    int pid;
    char writemessages[20]= "Hello";
    char readmessage[20];
    returnstatus = pipe(pipefds);
    if (returnstatus == -1) {
        printf("Unable to create pipe\n");
        return 1;
    }
    pid = fork();

    // Child process
    if (pid == 0) {
        read(pipefds[0], readmessage, sizeof(readmessage));
        printf("Child Process: Reading, message is %s\n", readmessage);
        return 0;
    }
    //Parent process
    printf("Parent Process: Writing, message is %s\n", writemessages);
    write(pipefds[1], writemessages, sizeof(writemessages));
    return 0;
}
```

In the above program, firstly the parent process will create a pipline and call fork() to create a child process. Then, the parent process will write a message to the pipeline. At the same time, the child process will read data from the pipeline. Noticeably, both write() and read() need to know the size of the message.

2.3. How to create multiple threads?

2.3.1. Thread libraries

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java. In this lab, we use POSIX Pthread on Linux and Mac OS to practice with multithreading programming.

Creating threads

```
pthread_create (thread, attr, start_routine, arg)
```

Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
- `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or `NULL` for the default values.
- `start`: the C routine that the thread will execute once it is created.
- `arg`: A single argument that may be passed to texttstart_routine. It must be passed by reference as a pointer cast of type void. `NULL` may be used if no argument is to be passed.

Example: Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
        long tid;
        tid = (long)threadid;
        printf("Hello World! It's me, thread #%ld!\n", tid);
        pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t],NULL, PrintHello, (void *)t);
        if (rc){
          printf("ERROR; return from pthread_create() is %d\n", rc);
          exit(-1);
        }
```

```
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Passing argument to Thread We can pass a structure to each thread such as the example below. Using the previous example to implement this example:

```
struct thread_data{
    int  thread_id;
    int  sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *thread_arg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) thread_arg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
         (void *) &thread_data_array[t]);
    ...
}
```

Joining and Detaching Threads "Joining" is one way to accomplish synchronization between threads, For example:

- The `pthread_join()` subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if

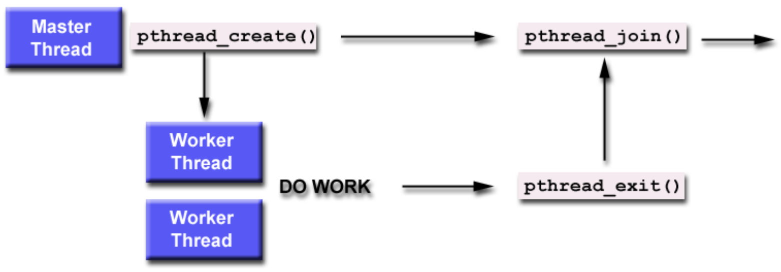


Figure 2.1: Joining threads.

it was specified in the target thread's call to `pthread_exit()`.

- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.

2.3.2. Multithread programming

Problem: Constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
        pthread_t tid; /* the thread identifier */
        pthread_attr_t attr; /* set of thread attributes */

        if (argc != 2) {
                fprintf(stderr,"usage: a.out <integer value>\n");
                return -1;
        }

        if (atoi(argv[1]) < 0) {
                fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
                return -1;
        }
        /* get the default attributes */
        pthread_attr_init(&attr);
        /* create the thread */
        pthread_create(&tid, &attr, runner, argv[1]);
        /* wait for the thread to exit */
        pthread join(tid,NULL);

        printf("sum = %d\n",sum);
        }

/* The thread will begin control in this function */
void *runner(void *param)
{
        int i, upper = atoi(param);
        sum = 0;
        for (i = 1; i <= upper; i++)
        sum += i;
        pthread exit(0);
}
```

3. Exercise (Required)

3.1. Problem 1

Firstly, downloading two text files from the url: `https://drive.google.com/file/d/1fgJqOeWbJC4ghMKHkuxfIP6dh2F911-E/view?usp=sharing` 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.

3.2. Problem 2

An interesting way of calculating `pi` is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows: Suppose you have a circle inscribed within a square, as shown in Figure 3.1.

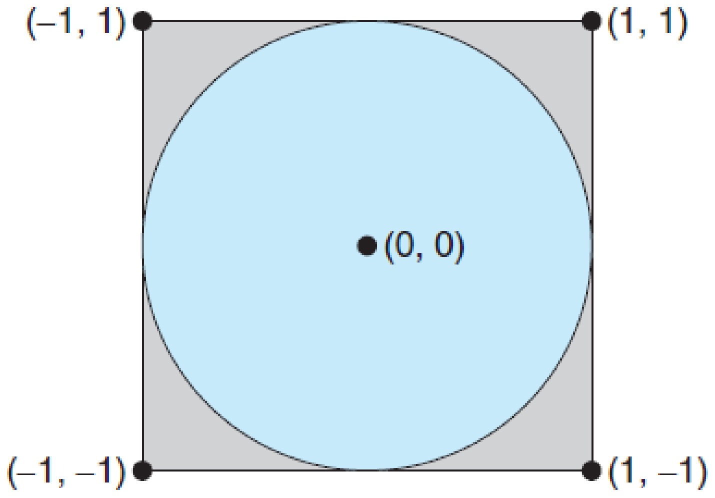


Figure 3.1: Monte Carlo technique for calculating pi.

(Assume that the radius of this circle is 1.) First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate `pi` by performing the following calculation:

```
pi = 4 x (number of points in circle) / (total number of points)
```

As a general rule, the greater the number of points, the closer the approximation to `pi`. However, if we generate too many points, this will take a very long time to perform our approximation. Solution for this problem is to carry out point generation and calculation concurrently. Suppose the number of points to be generated is *nPoints*. We create *N* separate threads and have each thread to create only *nPoints / N* points and count the number of points fall into the circle. After all threads have done their job we then get the total number of points inside the circle by combining the results from each thread. Since the total number of points has been generated equal to *nPoint*, the results of this method is equivalent to that of running a single process program. Furthermore, as threads run concurrently and the number of points each thread has to handle is much fewer than that of a singe process program, we can save a lot of time.

Write two programs implementing algorithm describe above: one serial version and one multi-thread version.
The program takes the number of points to be generated from user then creates multiple threads to approximate pi. Put all of your code in two files named *"pi_serial.c"* and *"pi_multi-thread.c"*. The number of points is passed to your program as an input parameter. For example, if your executable file is *pi* then to have your program calculate pi by generating one million points, we will use the follows command:

```
./pi_serial 1000000
./pi_multi-thread 1000000
```

Requirement: The multi-thread version must have some speed-up compared to the serial version. There are at least 2 targets in the Makefile *pi_serial* and *pi_multi-thread* to compile the two program.

3.3. Problem 3

Conventionally, pipe is a one-way communication method.(In the example at section 3, you can test by add a read() call after the writer() call at the parent process, a write() call after the read() call at the child process and observe what happens?). However, we still can have some tricks to adapt it for two-way communication by using two pipes. In this exercise, you should implement the TODO segment in the below program.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
static int pipefd1[2],pipefd2[2];

void INIT(void){
    if(pipe(pipefd1)<0 || pipe(pipefd2)<0){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}
void WRITE_TO_PARENT(void){
  /* send parent a message through pipe*/
   // TO DO
    printf("Child send message to parent!\n");
}
void READ_FROM_PARENT(void){
  /* read message sent by parent from pipe*/
  // TO DO
  printf("Child receive message from parent!\n");
}
void WRITE_TO_CHILD(void){
  /* send child a message through pipe*/
  // TO DO
  printf("Parent send message to child!\n");
}
void READ_FROM_CHILD(void){
  /* read the message sent by child from pipe */
    // TO DO
    printf("Parent receive message from child!\n");
}
int main(int argc, char* argv[]){
    INIT();
    pid_t pid;
    pid = fork();
    //set a timer, process will end after 1 second.
    alarm(10);
    if(pid==0){
        while(1){
            sleep(rand()%2+1);
            WRITE_TO_CHILD();
            READ_FROM_CHILD();
        }
    }else{
        while(1){
            sleep(rand()%2+1);
            READ_FROM_PARENT();
            WRITE_TO_PARENT();
        }
      }
    return 0;
}
```

A. Memory-related data structures in the kernel

In the Linux kernel, every process has an associated struct task_struct. The definition of this struct is in the header file include /linux/sched.h.

```
struct task_struct {
        volatile long state;
        /* -1 unrunnable, 0 runnable, >0 stopped */
        struct thread_info *thread_info;
        atomic_t usage;
        ...
        struct mm_struct *mm, *active_mm;
        ...
        pid_t pid;
        ...
        char comm[16];
        ...
};
```

Three members of the data structure are relevant to us:

- pid contains the Process ID of the process.
- comm holds the name of the process.
- The mm_struct within the task_struct is the key to all memory management activities related to the process.

The mm_struct is defined in include/linux/sched.h as:

```
struct mm_struct {
        struct vm_area_struct * mmap; /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache; /* list of VMAs */
        ...
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        ...
}
```

Here the first member of importance is the mmap. The mmap contains the pointer to the list of VMAs (Virtual Memory Areas) related to this process. Full usage of the process address space occurs very rarely. The sparse regions used are denoted by VMAs. The VMAs are stored in struct vm_area_struct defined in linux/mm.h:

```
struct vm_area_struct {
        struct mm_struct * vm_mm; /*The address space we belong to.*/
        unsigned long vm_start; /*Our start address within vm_mm.*/
        unsigned long vm_end; /*The first byte after our end
                                 address within vm_mm.*/
        ....
        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;
        ....
}
```

Kernel's view of the segments

The kernel keeps track of the segments which have been allocated to a particular process using the above structures. For each segment, the kernel allocates a VMA. It keeps track of these segments in the mm_struct structures. The kernel tracks the data segment using two variables: start_data and end_data. The code segment boundaries are in the start_code and end_code variables. The stack segment is covered by the single variable start_stack. There is no special variable to keep track of the BSS segment - the VMA corresponding to the BSS accounts for it.