HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 2
Process & Multithreaded Process
Course: Operating Systems

November 5, 2021

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.

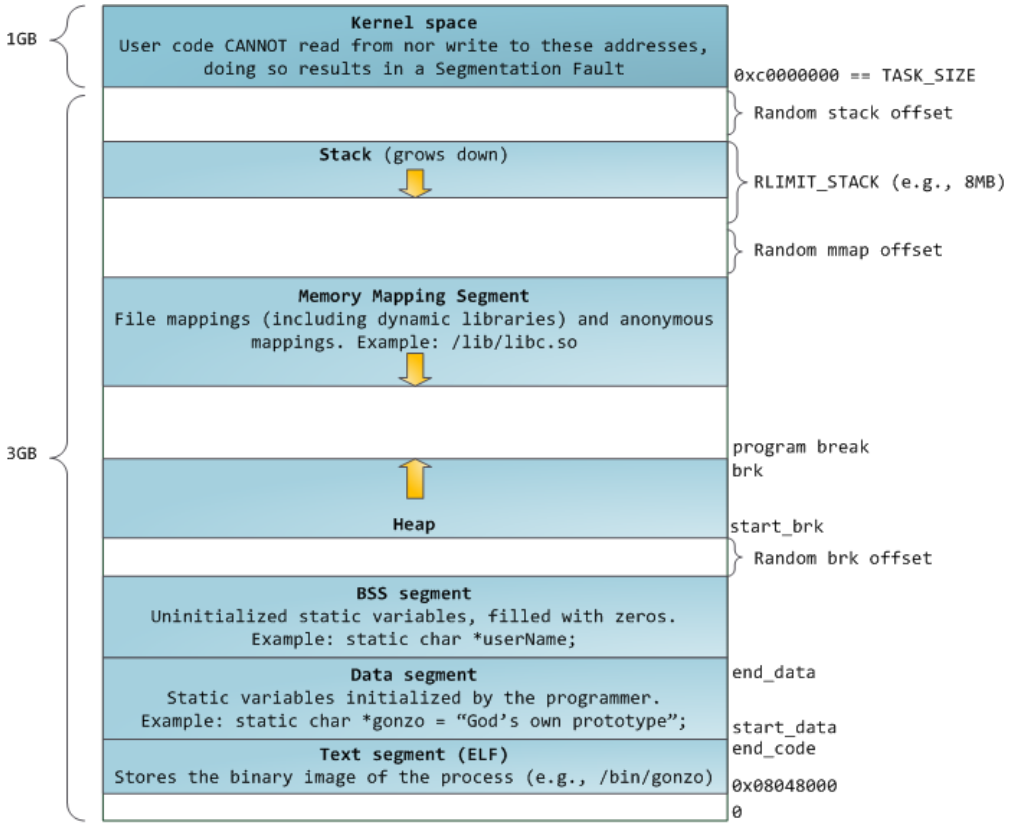

Figure 1.1: Layout of memory segments with process.

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

```
1  #include <stdio.h>
2  void func (unsigned long number) {
3          unsigned long local_f = number;
4          printf("%2lu ---> %p\n", local_f, &local_f);
5          local_f--;
6          if (local_f > 0) {
7                  func(local_f);
8          }
9  }
10 int main() {
11         func(10);
12 }
```

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

```
1 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. 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.

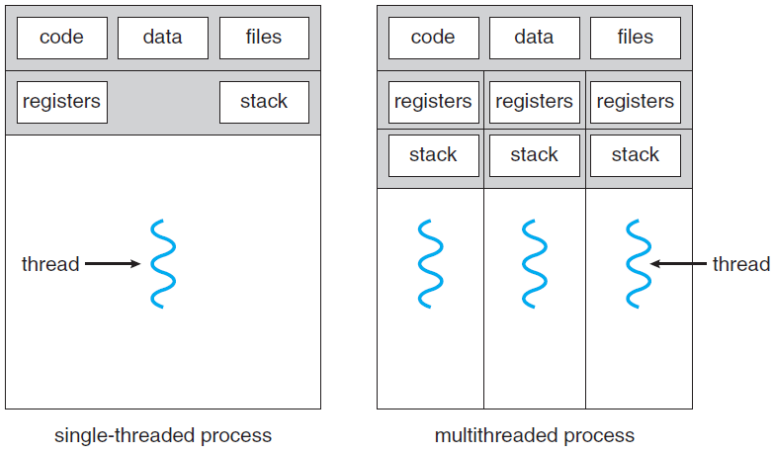

Figure 1.2: Single-threaded and multithreaded processes.

Figure 1.2 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

Q1. 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.

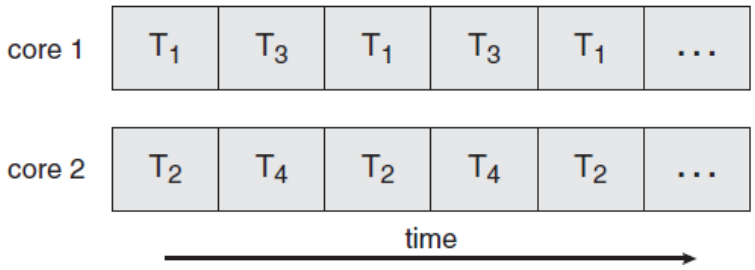

Figure 1.3: Parallel execution on a multicore system.

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 1.3 shown.

Q2. 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:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int glo_init_data = 99;
7  int glo_noninit_data;
8
9  void print_func(){
10         int local_data = 9;
11         printf("Process ID = %d\n", getpid());
12         printf("Addresses of the process:\n");
13         printf("1. glo_init_data = %p\n", &glo_init_data);
14         printf("2. glo_noninit_data = %p\n", &glo_noninit_data);
```

```
15          printf("3. print_func() = %p\n", &print_func);
16          printf("4. local_data = %p\n", &local_data);
17  }
18
19  int main(int argc, char **argv) {
20          print_func();
21          return 0;
22  }
```

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. Dynamic allocation on Linux/Unix system

2.2.1. 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.

2.2.2. Process's memory and heap

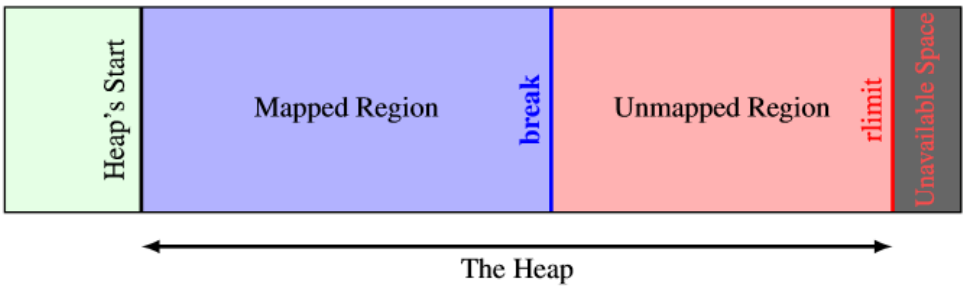

Figure 2.1: Heap region.

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 2.1 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.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc,char* argv[]){
6          size_t MB = 1024*1024; // # of bytes for allocating
7          size_t maxMB = 0;
8          void *ptr = NULL;
9
10         do{
11                 if(ptr != NULL){
12                         printf("Bytes of memory checked = %zi\n",maxMB);
13                         memset(ptr,0,maxMB); // fill the allocated region
14                 }
15                 maxMB += MB;
16                 ptr = malloc(maxMB);
17         }while(ptr != NULL);
18
19         return 0;
20 }
```

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().

2.2.3. BRK(2) AND SBRK(2)

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

```
1 int brk(const void *addr);
2 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.

```
1  #include <sys/types.h>
2  #include <unistd.h>
3
4  void *simple_malloc(size_t size)
5  {
6          void *p;
7          p = sbrk (0);
8          /* If sbrk fails , we return NULL */
9          if (sbrk(size) == (void*)-1)
10                 return NULL;
11         return p;
12 }
```

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 texttttstart_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

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #define NUM_THREADS 5
4
5  void *PrintHello(void *threadid)
6  {
7          long tid;
8          tid = (long)threadid;
9          printf("Hello World! It's me, thread #%ld!\n", tid);
10         pthread_exit(NULL);
11 }
12
13 int main (int argc, char *argv[])
14 {
15    pthread_t threads[NUM_THREADS];
16    int rc;
17    long t;
18    for(t=0; t<NUM_THREADS; t++){
19          printf("In main: creating thread %ld\n", t);
20          rc = pthread_create(&threads[t],NULL, PrintHello, (void *)t);
21          if (rc){
22            printf("ERROR; return from pthread_create() is %d\n", rc);
23            exit(-1);
24          }
25    }
26
27    /* Last thing that main() should do */
28    pthread_exit(NULL);
29 }
```

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:

```
1  struct thread_data{
2     int  thread_id;
3     int  sum;
4     char *message;
5  };
6
7  struct thread_data thread_data_array[NUM_THREADS];
8
9  void *PrintHello(void *thread_arg)
10 {
11    struct thread_data *my_data;
12    ...
13    my_data = (struct thread_data *) thread_arg;
14    taskid = my_data->thread_id;
15    sum = my_data->sum;
16    hello_msg = my_data->message;
17    ...
18 }
19
20 int main (int argc, char *argv[])
21 {
22    ...
23    thread_data_array[t].thread_id = t;
24    thread_data_array[t].sum = sum;
25    thread_data_array[t].message = messages[t];
26    rc = pthread_create(&threads[t], NULL, PrintHello,
27         (void *) &thread_data_array[t]);
28    ...
29 }
```

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

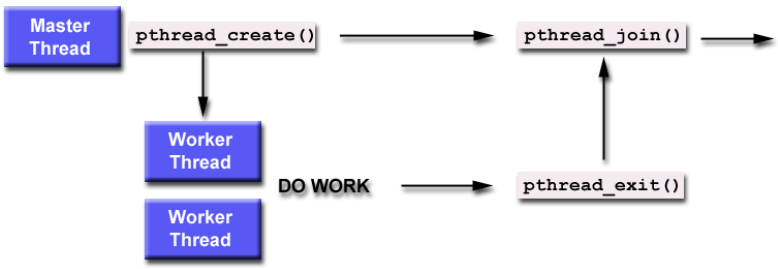

Figure 2.2: Joining threads.

- 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 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.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  int sum; /* this data is shared by the thread(s) */
5  void *runner(void *param); /* threads call this function */
6
7  int main(int argc, char *argv[])
8  {
9          pthread_t tid; /* the thread identifier */
10         pthread_attr_t attr; /* set of thread attributes */
11
12         if (argc != 2) {
13                 fprintf(stderr,"usage: a.out <integer value>\n");
14                 return -1;
15         }
16
17         if (atoi(argv[1]) < 0) {
18                 fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
19                 return -1;
20         }
21         /* get the default attributes */
22         pthread_attr_init(&attr);
23         /* create the thread */
24         pthread_create(&tid, &attr, runner, argv[1]);
25         /* wait for the thread to exit */
26         pthread join(tid,NULL);
27
28         printf("sum = %d\n",sum);
29         }
30
31 /* The thread will begin control in this function */
```

```
32 void *runner(void *param)
33 {
34         int i, upper = atoi(param);
35         sum = 0;
36         for (i = 1; i <= upper; i++)
37         sum += i;
38         pthread_exit(0);
39 }
```

3. Exercise (Required)

3.1. Problem 1

Implement the following function

```
void * aligned_malloc(unsigned int size, unsigned int align);
```

This function is similar to the standard *malloc* function except that the address of the allocated memory is a multiple of *align*. *align* must be a power of two and greater than zero. If the *size* is zero or the function cannot allocate a new memory region, it returns a NULL. For examples:

```
aligned_malloc(16, 64)
```

requires us to allocate a block of 16 bytes in memory and the address of the first byte of this block must be divisible by 64. This means if the function returns a pointer to $0x7e1010$ then it is incorrectly implemented because $0x7e1010$ (8261648 in decimal format) is not divisible by 64. However, $0x7e1000$ is a valid pointer since it is divisible by 64.
Associated with aligned_malloc() function, that is free() function to deallocate the memory region allocated in aligned_malloc() function. Along with the implementation of aligned_malloc() function, you have to implement aligned_free() below:

```
void * aligned_free(void *ptr);
```

Given pointer *ptr*, this function must deallocate the memory region pointed by this pointer.
Note: You must put the definition of those functions (*aligned_malloc*() and *aligned_free*()) in a file named *ex*1.*h* and their implementation in another file named *ex*1.*c*. DO NOT write *main* function in those files. You must write the test part of your function in another file.

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.
(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

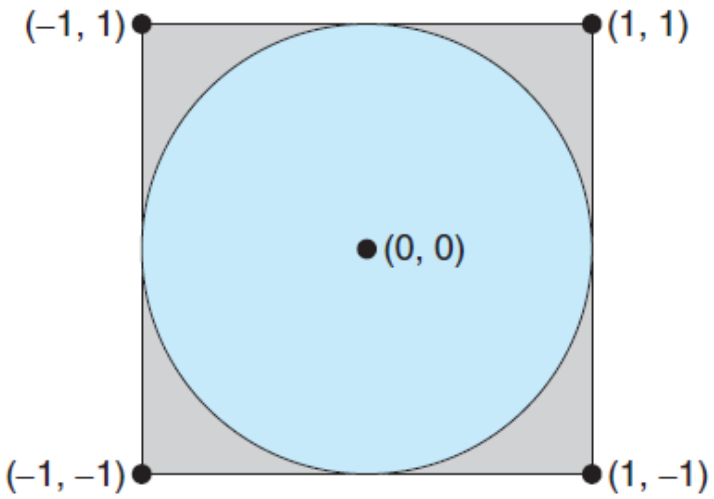

Figure 3.1: Monte Carlo technique for calculating pi.

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

Given the content of a C source file named *"code.c"*

```
1  #include <stdio.h>
2  #include <pthread.h>
3  void * hello(void * tid) {
4          printf("Hello from thread %d\n", (int)tid);
5  }
6  int main() {
7          pthread_t tid[10];
8          int i;
9          for (i = 0; i < 10; i++) {
10                 pthread_create(&tid[i], NULL, hello, (void*)i);
11         }
12         pthread_exit(NULL);
13 }
```

Since threads run concurrently, the output produced by threads will appear randomly and in an unpredictable way. Modify this program to have those threads to print their thread ID in ascending order every time we run the program as follows:

```
1  ## ./code
2  Hello from thread 0
3  Hello from thread 1
4  Hello from thread 2
5  Hello from thread 3
6  Hello from thread 4
7  Hello from thread 5
8  Hello from thread 6
9  Hello from thread 7
10 Hello from thread 8
11 Hello from thread 9
```

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.

```
1  struct task_struct {
2          volatile long state;
3          /* -1 unrunnable, 0 runnable, >0 stopped */
4          struct thread_info *thread_info;
5          atomic_t usage;
6          ...
7          struct mm_struct *mm, *active_mm;
8          ...
9          pid_t pid;
10         ...
11         char comm[16];
12         ...
13 };
```

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:

```
1  struct mm_struct {
2          struct vm_area_struct * mmap; /* list of VMAs */
3          struct rb_root mm_rb;
4          struct vm_area_struct * mmap_cache; /* list of VMAs */
5          ...
6          unsigned long start_code, end_code, start_data, end_data;
7          unsigned long start_brk, brk, start_stack;
8          ...
9  }
```

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:

```
1  struct vm_area_struct {
2          struct mm_struct * vm_mm; /*The address space we belong to.*/
3          unsigned long vm_start; /*Our start address within vm_mm.*/
4          unsigned long vm_end; /*The first byte after our end
5                                  address within vm_mm.*/
6          ....
7          /* linked list of VM areas per task, sorted by address */
8          struct vm_area_struct *vm_next;
9          ....
10 }
```

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.