12. POSIX Threads

SOA, Deemed to be University
ITER, Bhubanewar

Book(s)

Text Book(s)

Kay A. Robbins, & Steve Robbins

Unix^{TM} Systems Programming

Communications, concurrency, and Treads

Pearson Education

Reference Book(s)

Brain W. Kernighan, & Rob Pike

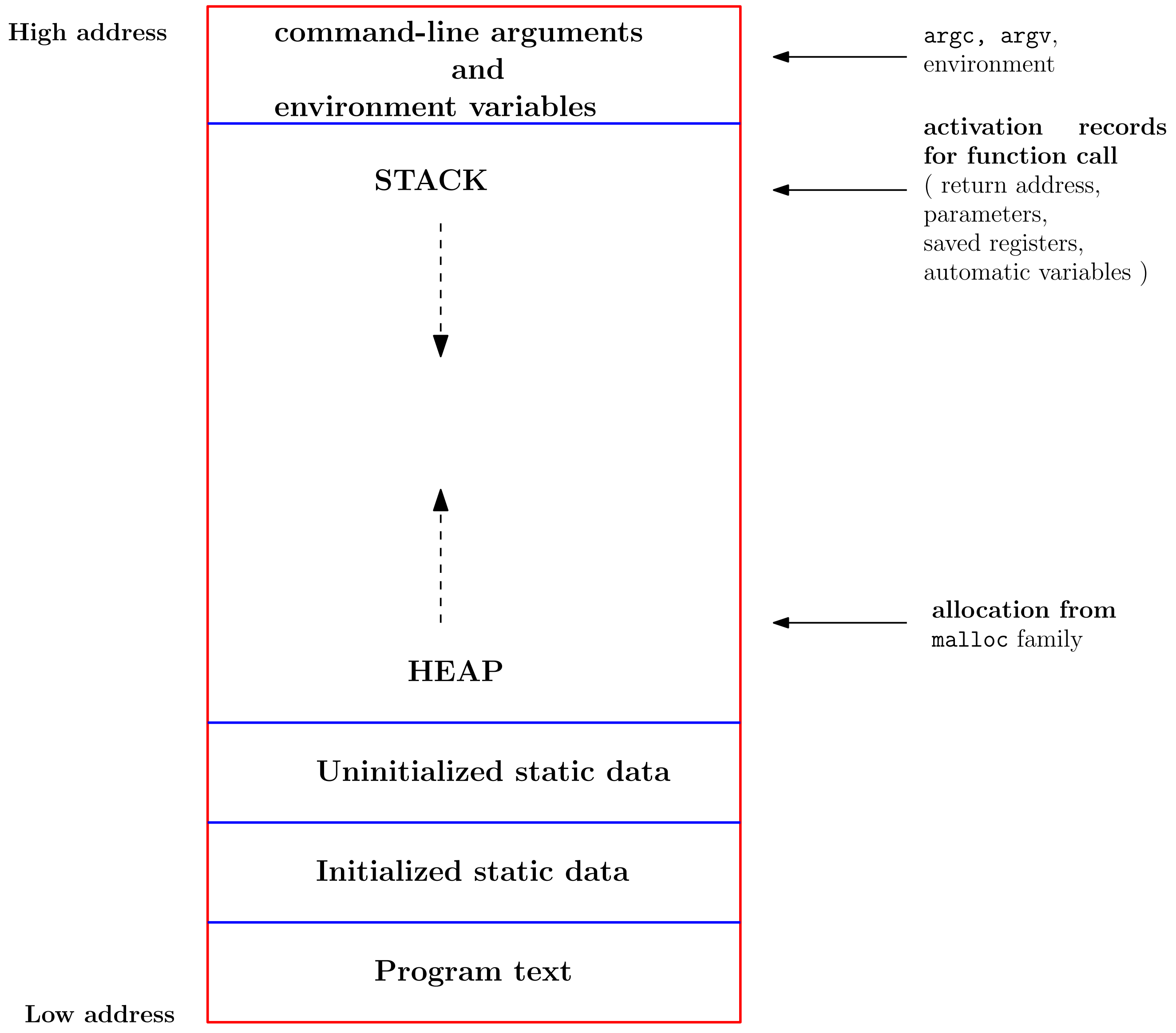
The Unix Programming Environment

PHI

Introduction

- ☞ One method of achieving parallelism is for multiple processes to cooperate and synchronize through shared memory or message passing.
- ☞ An alternative approach uses multiple threads of execution in **a single address space**.
- ☞ POSIX threads, described in the POSIX:THR Threads Extension.

Layout for a Program Image in Main Memory


High address
command-line arguments and environment variables
STACK
HEAP
Uninitialized static data
Initialized static data
Program text
Low address
argc, argv, environment
activation records for function call (return address, parameters, saved registers, automatic variables)
allocation from malloc family

Layout for a Program Image

- ☞ **Program image:** After loading, the program executable appears to occupy a contiguous block of memory called a program image.
- ☞ An ***activation record*** is a block of memory allocated on the top of the process stack to hold the execution context of a function during a call.
- ☞ Each function call creates a new activation record on the stack. The activation record is removed from the stack when the function returns, providing the last-called-first-returned order for nested function calls.
- ☞ The activation record contains the return address, the parameters (whose values are copied from the corresponding arguments), status information and a copy of some of the CPU register values at the time of the call.
- ☞ The process restores the register values on return from the call represented by the record.
- ☞ The activation record also contains automatic variables that are allocated within the function while it is executing. The particular format for an activation record depends on the hardware and on the programming language.

Layout for a Program Image Contd...

- ☞ The **`malloc`** family of functions allocates storage from a free memory pool called the **heap**. Storage allocated on the heap persists until it is freed or until the program exits.
- ☞ If a function calls **`malloc`**, the storage remains allocated after the function returns. The program cannot access the storage after the return unless it has a pointer to the storage that is accessible after the function returns.
- ☞ **Static variables** that are not explicitly initialized in their declarations are initialized to 0 at run time.
- ☞ Typically, the **initialized static variables** are part of the **executable module** on disk, but the uninitialized static variables are not.
- ☞ The **automatic variables** are not part of the executable module because they are only allocated when their defining block is called. The initial values of automatic variables are undetermined unless the program explicitly initializes them.

Example: Sizes of Executable Modules

Run the given below code with the command `$ size ./a.out` to look into different sections of a program image in C.

Program 1 : version1arrayinit.c

```c
int myarray[50000] = {1, 2, 3, 4};
int main(void) {
myarray[0] = 3;
return 0;
}
```

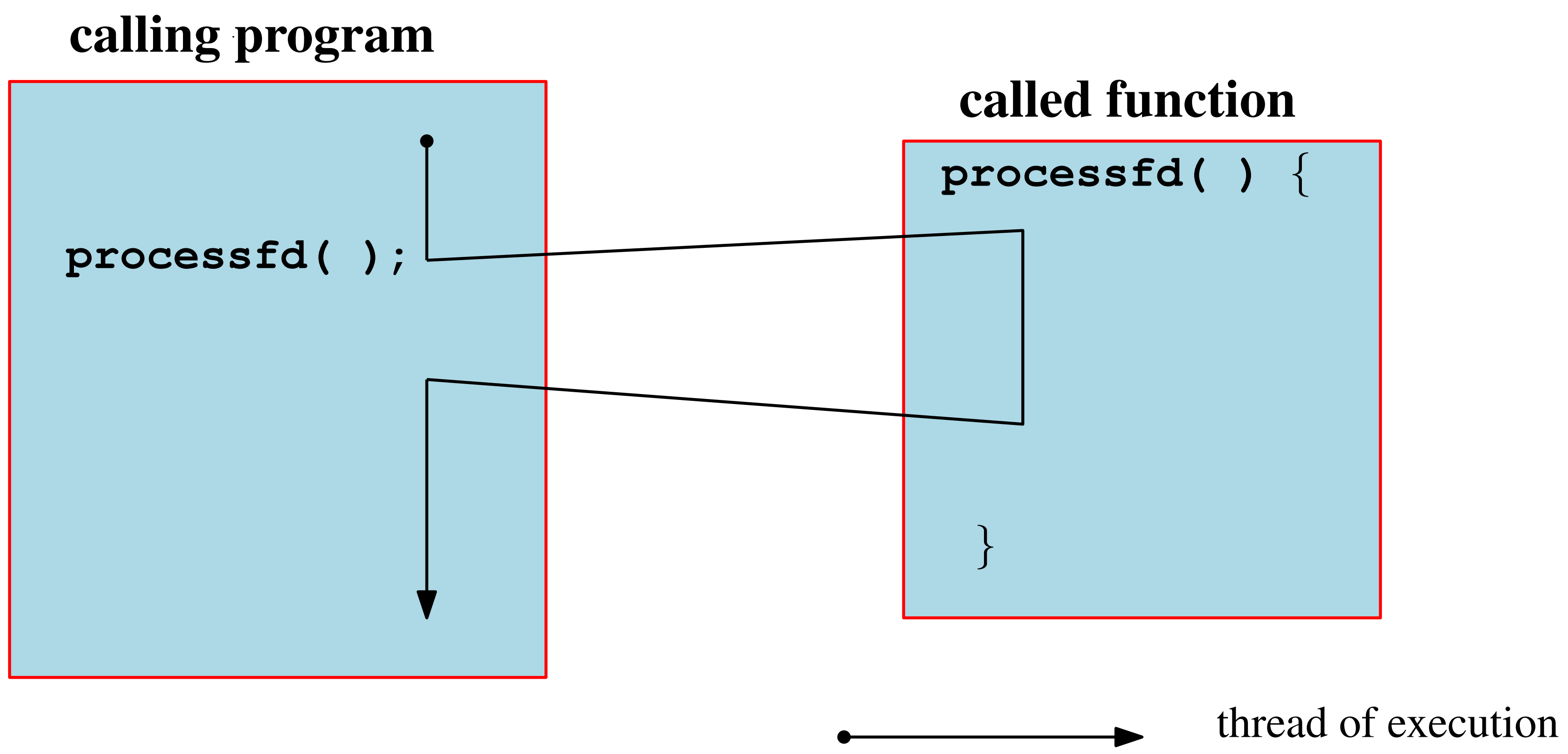
Program 2 : version2arrayinit.c

```c
int myarray[50000];
int main(void) {
myarray[0] = 3;
return 0;
}
```

Use `ls -l` to compare the sizes of the executable modules for the above two C programs. Explain the results.

Execution of a Normal Function Call

Program that makes an ordinary call to **`processfd`** has a single thread of execution.

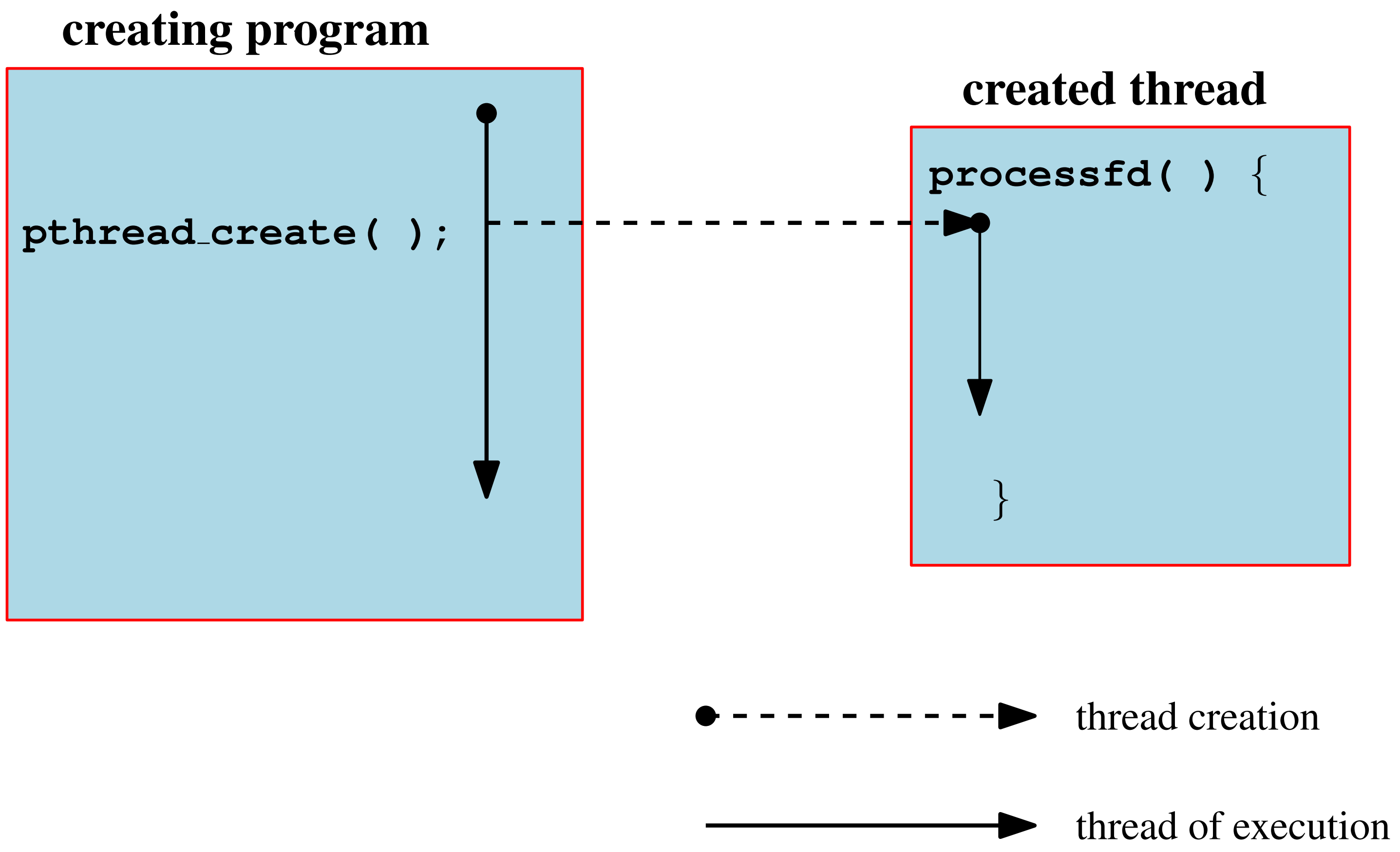


Execution of a Normal Function Call

- ☞ The calling mechanism creates an activation record (usually on the stack) that contains the return address.
- ☞ The thread of execution jumps to **`processfd`** when the calling mechanism writes the starting address of processfd in the processor's program counter.
- ☞ The thread uses the newly created activation record as the environment for execution, creating automatic variables on the stack as part of the record.
- ☞ The thread of execution continues in **`processfd`** until reaching a return statement (or the end of the function).
- ☞ The return statement copies the return address that is stored in the activation record into the processor program counter, causing the thread of execution to jump back to the calling program.

Thread Creation & Execution

Program that creates a new thread to execute **`processfd`** has two threads of execution.



Execution of a Thread

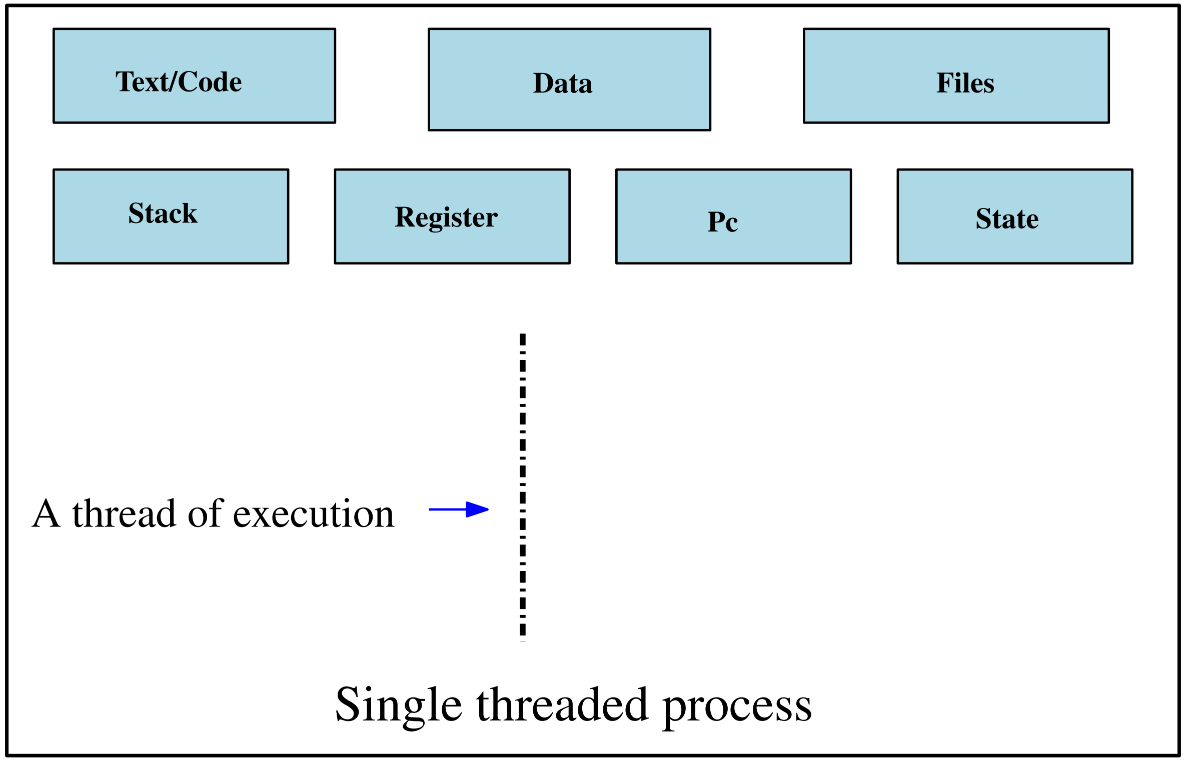
- The **`pthread_create`** call creates a new “schedulable entity” with its own value of the program counter, its own stack and its own scheduling parameters.

- The “schedulable entity” (i.e., thread) executes an independent stream of instructions, never returning to the point of the call.

- The calling program continues to execute concurrently.

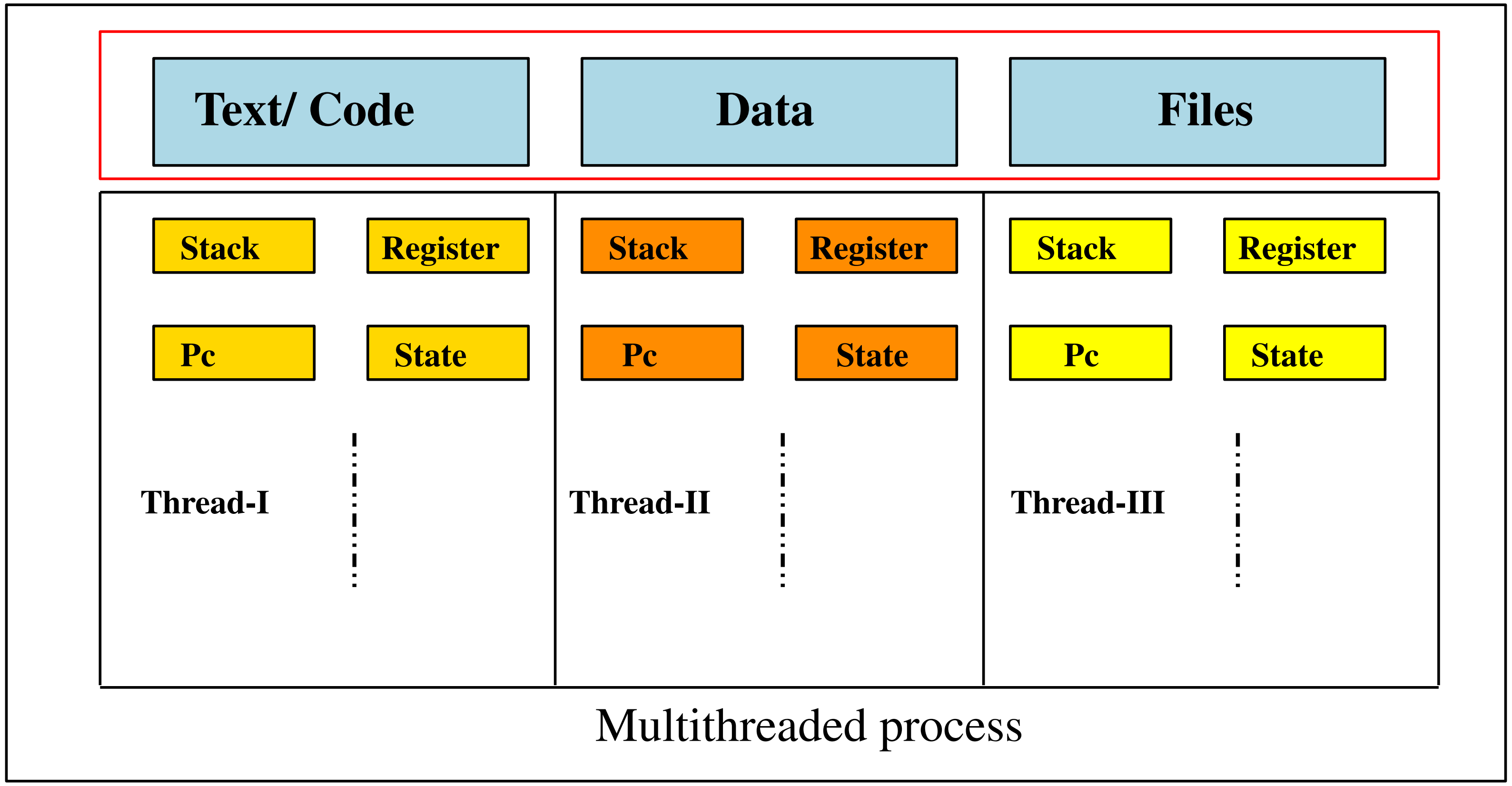
- In contrast, when **`processfd`** is called as an ordinary function, the caller’s thread of execution moves through the function code and returns to the point of the call, generating a single thread of execution rather than two separate ones.

Semantics of Single Thread



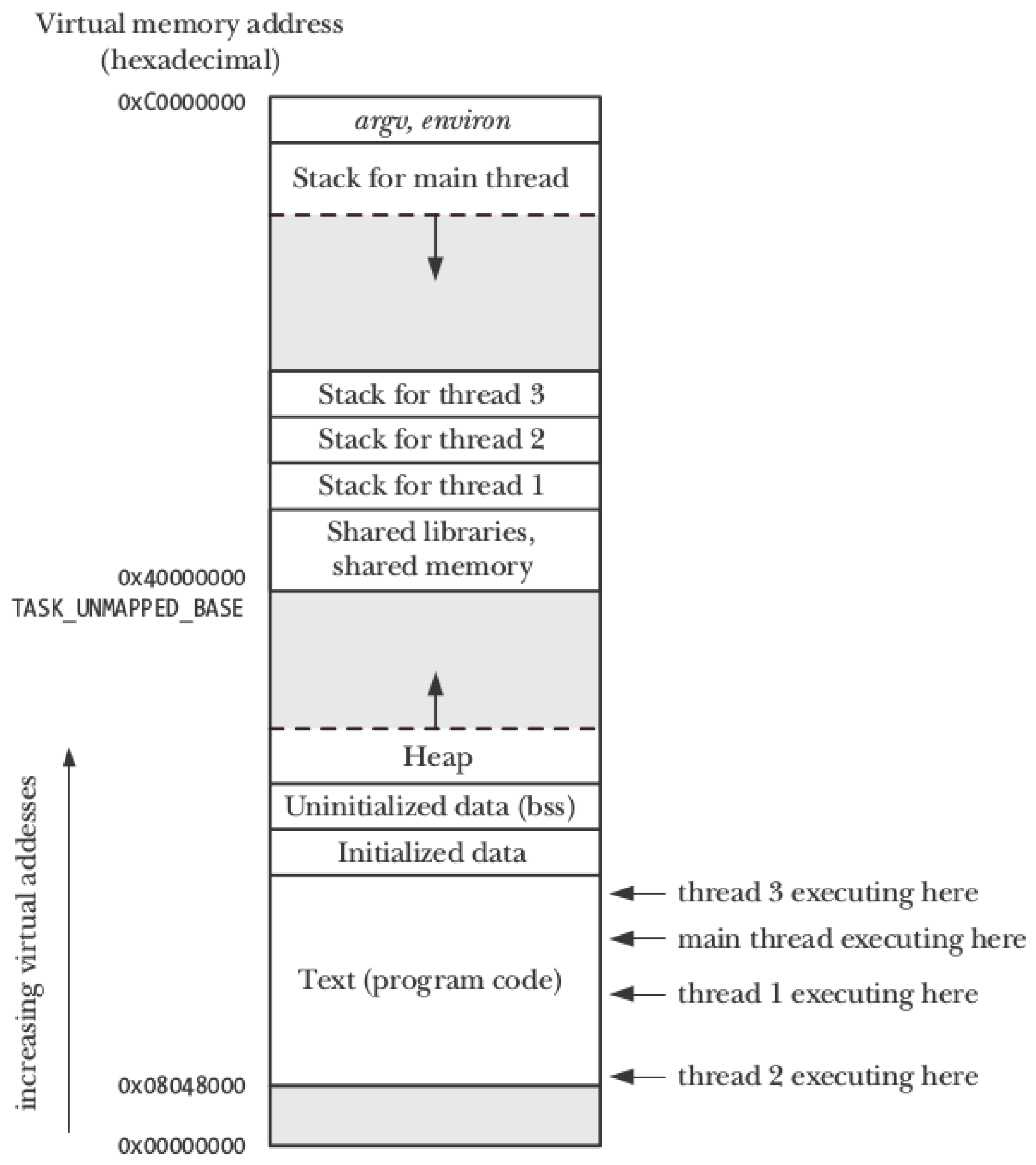
- A process starts with a single flow of control that executes a sequence of instructions.
- When a program executes, the value of the process program counter determines which process instruction is executed next. The resulting stream of instructions, called a **thread of execution**.
- A **thread of execution** can be represented by the sequence of instruction addresses assigned to the program counter during the execution of the program's code.

Semantics of MultiThread



- A thread is an abstract data type that represents a thread of execution within a process.
- A thread has its own execution **stack**, **program counter** value, **register set** and **state**. By declaring many threads within the confines of a single process, a programmer can write programs that achieve parallelism with low overhead. **While these threads provide low-overhead parallelism, they may require additional synchronization because they reside in the same process address space and therefore share process resources.**
- Processes are called **heavyweight** because of the work needed to start them. In contrast, threads are called **lightweight** processes.

Example: Four threads executing in a process



Thread Management

- A thread package usually includes functions for thread creation and thread destruction, scheduling, enforcement of mutual exclusion and conditional waiting.

- A typical thread package also contains a **runtime system** to manage threads transparently (i.e., the user is not aware of the runtime system).

- When a thread is created, the runtime system allocates data structures to hold the thread's ID, stack and program counter value.

- The thread's internal data structure might also contain scheduling and usage information.

- **The threads for a process share the entire address space of that process**. They can modify global variables, access open file descriptors, and cooperate or interfere with each other in other ways.

- POSIX threads are sometimes called **pthreads** because all the thread functions start with **`pthread`**.

POSIX Thread Management Functions

POSIX function	**Description**
`pthread_create`	Create a thread
`pthread_join`	Wait for a thread
`pthread_detach`	Set thread to release resources
`pthread_exit`	Exit a thread without exiting process
`pthread_self`	Find out own thread ID
`pthread_equal`	Test two thread IDs for equality
`pthread_kill`	Send a signal to a thread
`pthread_cancel`	Terminate another thread

NOTE:: Most POSIX thread functions return 0 if successful and a nonzero error code if unsuccessful. They do not set **`errno`**, so the caller cannot use **`perror`** to report errors. Programs can use **`strerror`** if the issues of thread safety are addressed.

Creating a Thread

```
#include <pthread.h>

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

```
Returns:

(1) If successful, pthread_create returns 0.

(2) If unsuccessful, pthread_create returns a nonzero error
     code.
```

The `pthread_create` Parameters

`pthread_create()`:	The **`pthread_create`** function creates a thread. The POSIX **`pthread_create`** automatically makes the thread runnable without requiring a separate start operation.
`thread`:	The parameter of **`pthread_create`** points to the ID of the newly created thread.
`attr`:	The **`attr`** parameter represents an attribute object that encapsulates the attributes of a thread. If **`attr`** is `NULL`, the new thread has the default attributes.
`start_routine`:	The third parameter, **`start_routine`**, is the name of a function that the thread calls when it begins execution.
`arg`:	The **`start_routine`** a single parameter specified by **`arg`**, a pointer to **`void`**. The **`start_routine`** returns a pointer to **`void`**, which is treated as an exit status by **`pthread_join`**.

Example: pthread_create()

```
    pthread_t tid;
    pthrread_create(&tid,NULL,add,NULL);
```

Example pthread_create() with error

```
  pthread_t tid;
  if (error = pthread_create(&tid, NULL, add, NULL))
      fprintf(stderr, "Failed to create thread: %s\n",
          strerror(error));
  else
      printf("Thread created\n");
```

```
void *add(void *arg){
   printf("Thread called function stmt\n");
}
```

Detaching Thread

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

```
Returns:

 (1) If successful, pthread_detach returns 0.

 (2) If unsuccessful, pthread_detach returns a nonzero error
      code.
```

- The **`pthread_detach`** function has a single parameter, **`thread`**, the thread ID of the thread to be detached.
- When a thread exits, it does not release its resources unless it is a detached thread.
- The **`pthread_detach`** function sets a thread's internal options to specify that storage for the thread can be reclaimed when the thread exits.
- Detached threads do not report their status when they exit.
- **Threads that are not detached are joinable** and do not release all their resources until another thread calls **`pthread_join`** for them or the entire process exits.
- The **`pthread_join`** function causes the caller to wait for the specified thread to exit, similar to **`waitpid`** at the process level.

Example: `pthread_detach()`

The following code segment creates and then detaches a thread to execute the function **`processfd`**.

```
#include<pthread.h>
void *processfd(void *arg);
int error;
pthread_t tid;
if (error = pthread_create(&tid, NULL, processfd, NULL))
   fprintf(stderr, "Failed to create thread: %s\n",
       strerror(error));
else if (error = pthread_detach(tid))
   fprintf(stderr, "Failed to detach thread: %s\n",
       strerror(error));
```

```
void *processfd(void *arg)
{
   printf("A detaching thread\n");
}
```

Example: pthread_detach()

When **detachfun** is executed as a thread, it detaches itself.

```
#include<pthread.h>
#include<stdio.h>
void *detachfun(void *arg)
{
   int i = *((int *)(arg));
   fprintf(stderr, "Before detached: argument is %d\n", i);
   if (!pthread_detach(pthread_self()))
      return NULL;
   fprintf(stderr, "If cond fails: argument is %d\n", i);
   return NULL;
}
int main()
{
   int x=10;
   pthread_t tid;
   pthread_create(&tid,NULL,detachfun,(void *)&x);
   sleep(1);
   return 0;
}
```

Thread Joining

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

```
Returns:

(1) If successful, pthread_join returns 0.

(2) If unsuccessful, pthread_join returns a nonzero error
    code.
```

thread: The **pthread_join** function suspends the calling thread until the target thread, specified by the first parameter, terminates.

value_ptr: The **value_ptr** parameter provides a location for a pointer to the return status that the target thread passes to **pthread_exit** or **return**. If **value_ptr** is NULL, the caller does not retrieve the target thread return status.

NOTE::Thread Joining to retrive value

The following code illustrates how to retrieve the value passed to **`pthread_exit`** by a terminating thread.

```
int error;
int *exitcodep;
pthread_t tid;
if (error = pthread_join(tid, &exitcodep))
   fprintf(stderr, "Failed to join thread: %s\n", strerror(
       error));
else
   fprintf(stderr, "The exit code was %d\n", *exitcodep);
```

☞ Calling **`pthread_join`** is not the only way for the main thread to block until the other threads have completed. The main thread can use a **semaphore** or any other methods to wait for all threads to finish.

Thread Exit

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

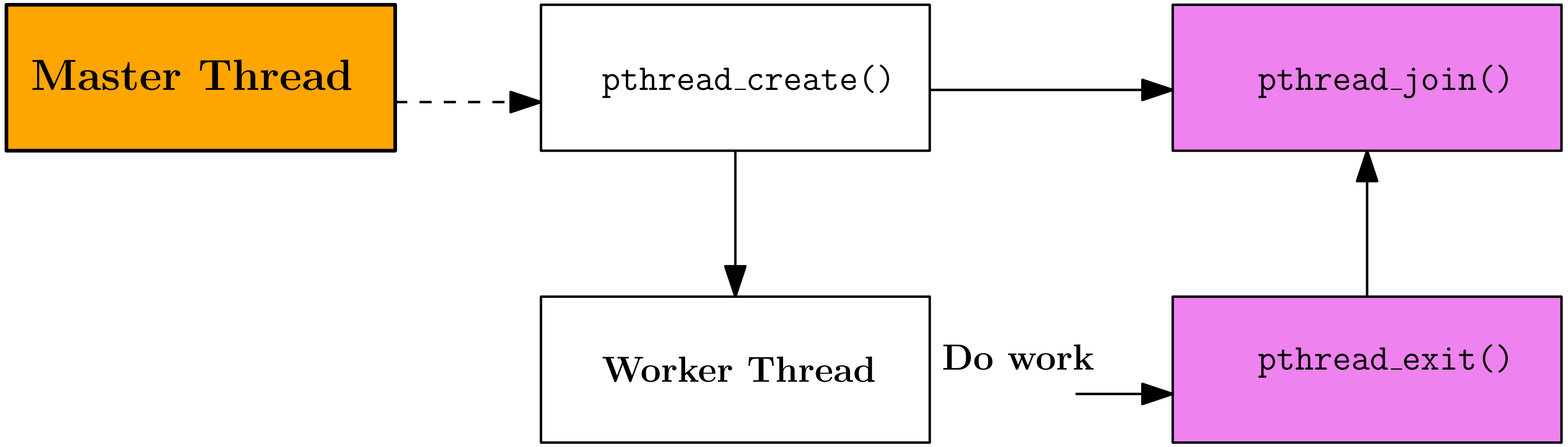
```
Returns:

POSIX does not define any errors for pthread_exit.
```

`pthread_exit:` A call to **`exit`** causes the entire process to terminate; a call to **`pthread_exit`** causes only the calling thread to terminate.

`value_ptr:` The **`value_ptr`** value is available to a successful **`pthread_join`**. Put `NULL` not to return any status. However, the **`value_ptr`** in **`pthread_exit`** must point to data that exists after the thread exits, so the thread should not use a pointer to automatic local data for **`value_ptr`**.

Thread Exit


Master Thread
pthread_create()
pthread_join()
Worker Thread
Do work
pthread_exit()

Note:::Thread Exit

- ☞ The process can terminate by calling **`exit`** directly, by executing **`return`** from main, or by having one of the other process threads call **`exit`**. In any of these cases, all threads terminate.
- ☞ If the main thread has no work to do after creating other threads, it should either block until all threads have completed or call **`pthread_exit(NULL)`**.
- ☞ A call to **`exit`** causes the entire process to terminate; a call to **`pthread_exit`** causes only the calling thread to terminate.
- ☞ A thread that executes **`return`** from its top level implicitly calls **`pthread_exit`** with the return value (a pointer) serving as the parameter to **`pthread_exit`**. **`return NULL`** is equivalent to **`pthread_exit(NULL)`** or **`return status`** is equivalent to **`pthread_exit(status).`**
- ☞ A process will exit with a return status of 0 if its last thread calls **`pthread_exit`**.

Passing Parameters to Threads

- The creator of a thread may pass a single parameter to a thread at creation time, using a pointer to `void`.
- To communicate multiple values, the creator must use a pointer to an array or a structure.

Example: Passing an Integer

```
void *passint(void *arg);
int main(){
 int x=20;pthread_t t;
 pthread_create(&t,NULL,passint,(void *)&x);
 pthread_join(t,NULL);
 return 0;
}void *passint(void *arg){
   int recv;
   recv=*((int *)(arg));
   printf("Parameter value=%d\n",recv);
   pthread_exit(NULL);
}
```

Passing a String

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

void *stringpass(void *arg);

int main(){
 char *msg="ITER";
 pthread_t t;
 pthread_create(&t,NULL,passint,(void *)msg);
 pthread_join(t,NULL);
 return 0;
}
void *stringpass(void *arg)
{
  char *str;
  str=(char *)(arg);
  printf("String received=%s\n",str);
  pthread_exit(NULL);
}
```

Passing an Integer Array

```
void *arraypass(void *arg);
int main()
{
 int arr[]={10,20,30,40};
 pthread_t tid;
 pthread_create(&tid,NULL,arraypass,(void *)arr);
 pthread_join(tid,NULL);
 printf("Bye....main thread\n");
 return 0;
}
void *arraypass(void *arg)
{
 int *ar,i;
 ar=(int *)arg;
 for(i=0;i<4;i++){
   printf("Received:arr[%d]=%d\n",i,*(ar+i));
   /*or  printf("Received:arr[%d]=%d\n",i,ar[i]); */
 }
 pthread_exit(NULL);
}
```

Passing an Array of Strings

```
#include<stdio.h>
#include<pthread.h>
void *stringarraypass(void *arg);
int main(){
 char *msg[]={"iter","soa","ibcs","sum"};
 pthread_t tid;
 pthread_create(&tid,NULL,stringarraypass,(void *)msg);
 pthread_join(tid,NULL);
 printf("Bye....main thread\n");
 return 0;
}
void *stringarraypass(void *arg){
 char **str;
 int i;
  str=(char **)arg;
 for(i=0;i<4;i++){
   printf("Received:arr[%d]=%s\n",i,str[i]);
 }
 pthread_exit(NULL);
}
```

Passing a Structure

```
void *structpass(void *arg);
struct pass{
    int a;
    int arr[10];
 };
int main(){
 struct pass pval; pthread_t tid;   int i;
 pval.a=6;
 for(i=0;i<pval.a;i++){
    pval.arr[i]=rand()%50+1;
 }
 pthread_create(&tid,NULL,structpass,(void *)&pval);
 pthread_join(tid,NULL);
 printf("Bye....main thread\n");
 return 0;
}
void *structpass(void *arg){
 struct pass *srcv;  int i;
 srcv=(struct pass *)arg;
 for(i=0;i<srcv->a;i++){
   printf("Received:arr[%d]=%i\n",i,srcv->arr[i]);
 }
 pthread_exit(NULL);
}
```

Returning Values from Thread-I

- ☞ The thread allocates memory space for returning the value since it is not allowed to return a pointer to its local variables.
- ☞ The thread called function returns the pointer using either **`return`** or **`pthread_exit()`**. The thread, who called **`pthread_join`** will get the return value by setting the 2nd parameter of the **`pthread_join`**.

Example: Returning an Integer

```
void *myfun(void *);
int main(){
 pthread_t tid;    int *p;
 pthread_create(&tid,NULL,myfun,NULL);
 pthread_join(tid,(void **)&p);
 printf("returned from thread=%d\n",*p);
 return 0;
}
void *myfun(void *arg){
  int *ret = malloc(sizeof(int));
  *ret=100;
  pthread_exit(ret); // or return ret;
}
```

Returning Values from Thread-II

- ☞ Pass the parameter to the thread an array of size one more to hold the return value.
- ☞ The address of extra array element is acts as the paramenter to **`pthread_exit()`** function. The extra array element stores the returned value and can be retrieved this value either through the array or through the second parameter of **`pthread_join`**.

Example: Returning a value

```
void *myfun(void *);
int main(){
 pthread_t tid; int *p; int a[2];
 pthread_create(&tid,NULL,myfun,(void *)a);
 pthread_join(tid,(void **)&p);
 printf("return=%d\n",*p);
 printf("using array=%d\n",a[1]);
 return 0;
}
void *myfun(void *arg){
  int *argint=(int *)arg;
  argint[1]=100;
  return argint+1;
}
```

Returning values from thread-III

- ☞ Value returned by manipulating the variable that is passed as an argument to the threaded function.
- ☞ The address of the passed parameter acts as the paramenter to **pthread_exit()** function. The variable stores the returned value and can be retrieved this value either through the variable or through the second parameter of **pthread_join**.

Example: Returning a value

```
void *myfun(void *);
int main(){
 pthread_t tid; int *p; int a;
 pthread_create(&tid,NULL,myfun,(void *)&a);
 pthread_join(tid,(void **)&p);
 printf("return=%d\n",*p);
 printf("using same variable=%d\n",a);
 return 0;
}
void *myfun(void *arg){
  int *passvar=(int *)arg;
  *passvar=100;
  pthread_exit(passvar);
}
```

Additional Reading

- Thread cancellation : **`pthread_cancel()`**
- Thread Safety
- User Threads versus Kernel Threads
- Thread Attributes.
  - attribute objects
  - state
  - stack
  - scheduling