POSIX Threads

- Introduction to Threads
- POSIX Threads
- The Pthread API
- Threads vs Processes

Threads

- Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently. A single process can contain multiple threads
- All threads are independently executing the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments.
- The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute in parallel. If one thread is blocked on I/O, other threads are still eligible to execute.

Threads in a Process

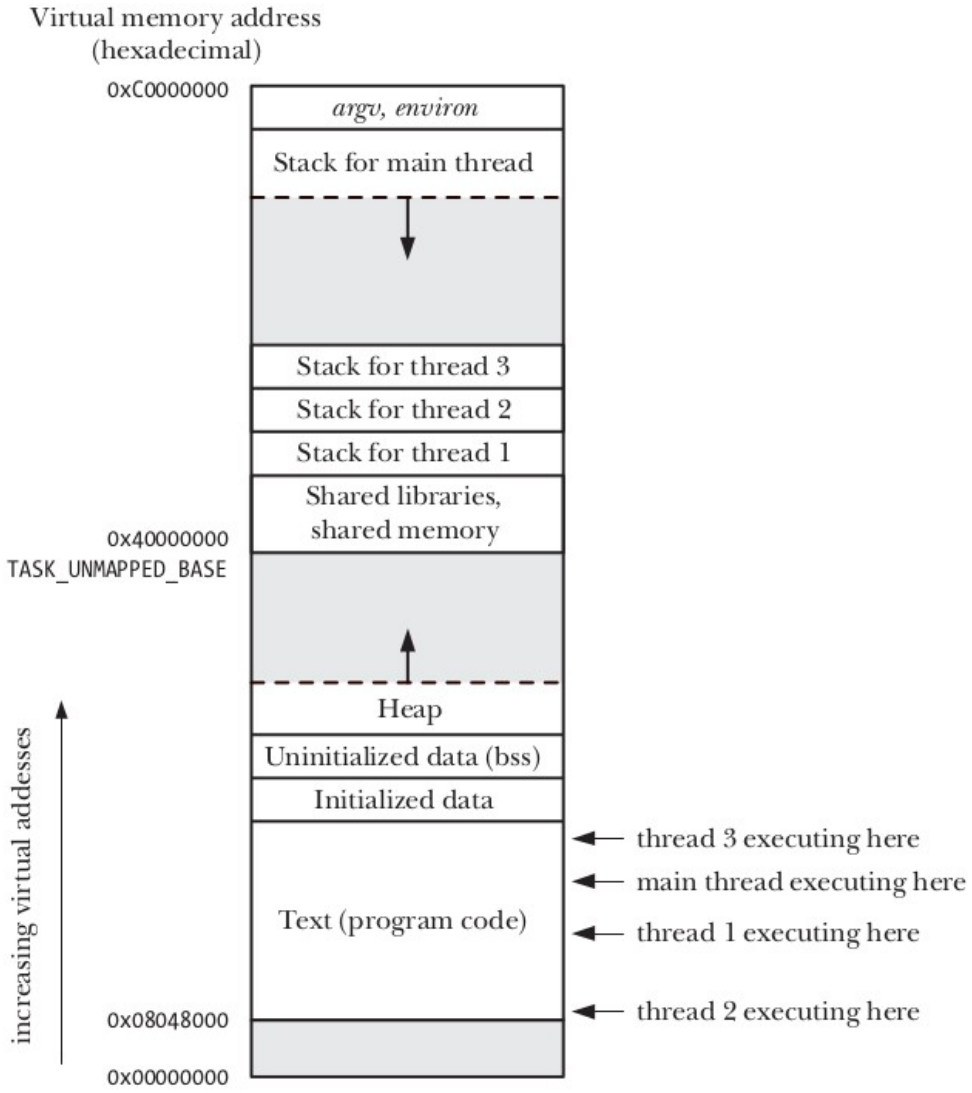


Figure 29-1: Four threads executing in a process (Linux/x86-32)

- As it can be seen from the figure all of the per-thread stacks reside within the same virtual address space. This means that, given a suitable pointer, it is possible for threads to share data on each other's stacks.
- This is occasionally useful, but it requires careful programming to handle the dependency that results from the fact that a local variable remains valid only for the lifetime of the stack frame in which it resides.
- Failing to correctly handle this dependency can create bugs that are hard to track down.

Threads : Introduction

Threads offer advantages over processes in certain applications.

To give an example consider a network server design in which a parent process accepts incoming connections from clients, and then uses *fork()* to create a separate child process to handle communication with each client

While this approach works well for many scenarios, it does have the following limitations in some applications :

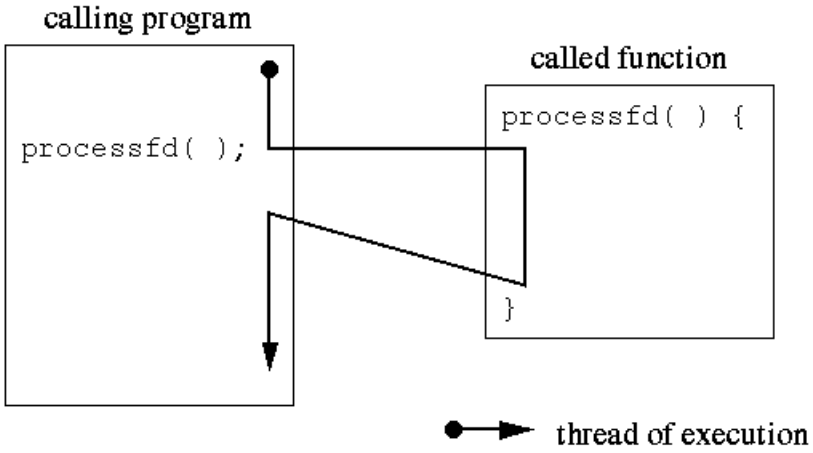
- It is difficult to share information between processes. Since the parent and child don't share memory, we must use some form of interprocess communication in order to exchange information between processes.
- Process creation with *fork()* is relatively expensive. The need to duplicate various process attributes such as page tables and file descriptor tables means that a *fork()* call is still time-consuming.

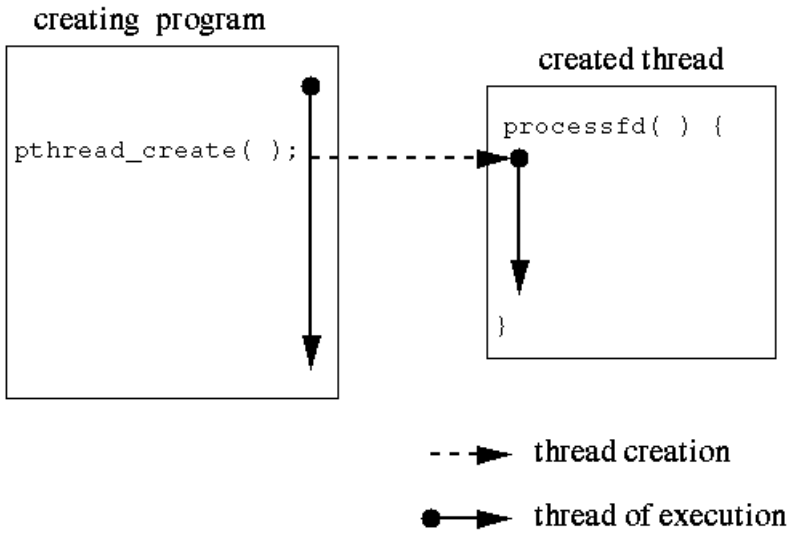
Threads : Motivation

Threads address both of these problems:

- Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables. However, in order to avoid the problems that can occur when multiple threads try to update the same information, we must employ some synchronization techniques.
- Thread creation is faster because many of the attributes that must be duplicated in a child created by *fork()* are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables

Threads : function call vs a Thread





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

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

- **When a new thread is created it runs concurrently with the creating process.**
  - **When creating a thread you indicate which function the thread should execute.**

Threads : function call vs a Thread

- A function that is used as a thread must have a special format.
- It takes a single parameter of type pointer to void and returns a pointer to void.
- The parameter type allows any pointer to be passed. This can point to a structure, so in effect, the function can use any number of parameters.
- The processfd function used above might have prototype:

```
void *processfd(void *arg);
```

- Instead of passing the file descriptor to be monitored directly, we pass a pointer to it.

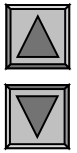
Function call and thread example

```
#include <stdio.h>
#include "restart.h"
#define BUFSIZE 1024

void docommand(char *cmd, int cmdsize);

void *processfd(void *arg) { /* process commands read from file descriptor */
   char buf[BUFSIZE];
   int fd;
   ssize_t nbytes;

   fd = *((int *)(arg));
   for ( ; ; )  {
      if ((nbytes = r_read(fd, buf, BUFSIZE)) <= 0)
         break;
      docommand(buf, nbytes);
   }
   return NULL;
}
```



Function call

```
void *processfd(void *);
int fd;

processfd(&fd);
```

Create a new thread to execute the function

```
void *processfd(void *arg);

int error;
int fd;
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd))
   fprintf(stderr, "Failed to create thread: %s\n",
strerror(error));
```

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.

Thread Managment

- When a thread allocates space for a return value, some other thread is responsible for freeing that space. *Whenever possible, a thread should clean up its own mess rather than requiring another thread to do it*.

- *When creating multiple threads, do not reuse the variable holding a thread's parameter until you are sure that the thread has finished accessing the parameter*. As the variable is passed by reference, it is a good practice to use a separate variable for each thread.

Thread Safety

- A hidden problem with threads is that they may call library functions that are not thread-safe, possibly producing spurious results
- A function is thread-safe if multiple threads can execute simultaneous active invocations of the function without interference.
- POSIX specifies that all the required functions, including the functions from the standard C library, be implemented in a thread-safe manner except for the specific functions
- Those functions whose traditional interfaces preclude making them thread-safe must have an alternative thread-safe version.

Thread Safety

- Another interaction problem occurs when threads access the same data.
- In more complicated applications, a thread may not exit after completing its assigned task. Instead, a worker thread may request additional tasks or share information.

User Threads versus Kernel Threads

- The two traditional models of thread control are user-level (green) threads and kernel-level threads.
- User-level threads usually run on top of an existing operating system. These threads are **invisible to the kernel** and compete among themselves for the resources allocated to their encapsulating process
- The threads are scheduled by a thread runtime system that is part of the process code
- Programs with user-level threads usually link to a special library in which each library function is enclosed by a jacket
- The jacket function calls the thread runtime system to do thread management before and possibly after calling the jacketed library function.

User Level Threads

- Functions such as `read` or `sleep` can present a problem for user-level threads because they may cause the process to block.
- To avoid blocking the entire process on a blocking call, the user-level thread library replaces each potentially blocking call in the jacket by a nonblocking version.
- The thread runtime system tests to see if the call would cause the thread to block. If the call would not block, the runtime system does the call right away. If the call would block, however, the runtime system places the thread on a list of waiting threads, adds the call to a list of actions to try later, and picks another thread to run.
- All this control is invisible to the user and to the operating system.

User Level Threads

- User-level threads have low overhead, but they also have some disadvantages.
- The user thread model, which assumes that the thread runtime system will eventually regain control, can be thwarted by CPU-bound threads.
- A CPU-bound thread rarely performs library calls and may prevent the thread runtime system from regaining control to schedule other threads.
- The programmer has to avoid the lockout situation by explicitly forcing CPU-bound threads to yield control at appropriate points.

Kernel-level Threads

- With kernel-level threads, the kernel is aware of each thread as a schedulable entity and threads compete systemwide for processor resources
- The scheduling of kernel-level threads can be almost as expensive as the scheduling of processes themselves, but kernel-level threads can take advantage of multiple processors.
- The synchronization and sharing of data for kernel-level threads is less expensive than for full processes, but kernel-level threads are considerably more expensive to manage than user-level threads.

Hybrid threads

- Hybrid thread models have advantages of both user-level and kernel-level models by providing two levels of control
- The user writes the program in terms of user-level threads and then specifies how many kernel-schedulable entities are associated with the process
- The user-level threads are mapped into the kernel-schedulable entities at runtime to achieve parallelism. The level of control that a user has over the mapping depends on the implementation

User Threads versus Kernel Threads

- The user-level threads are called threads and the kernel-schedulable entities are called lightweight processes
- The POSIX thread scheduling model is a hybrid model that is flexible enough to support both user-level and kernel-level threads in particular implementations of the standard.
- The model consists of two levels of scheduling—threads and kernel entities. The threads are analogous to user-level threads. The kernel entities are scheduled by the kernel. The thread library decides how many kernel entities it needs and how they will be mapped.

User Threads versus Kernel Threads

- User-level threads run on top of an operating system
  - Threads are invisible to the kernel.
  - Link to a special library of system calls that prevent blocking
  - Have low overhead
  - CPU-bound threads can block other threads
  - Can only use one processor at a time.
- Kernel-level threads are part of the OS.
  - Kernel can schedule threads like it does processes.
  - Multiple threads of a process can run simultaneously on multiple CPUs.
  - Synchronization is more efficient than for processes but less than for user-level threads.

Pthreads API

Pthread data types :

The Pthreads API defines a number of data types, some of which are listed

Data type	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attributes object
pthread_key_t	Key for thread-specific data
pthread_once_t	One-time initialization control context
pthread_attr_t	Thread attributes object

Pthreads API

Threads and errno :

- In the traditional UNIX API, errno is a global integer variable. However, this doesn't suffice for threaded programs.
- If a thread made a function call that returned an error in a global errno variable, then this would confuse other threads that might also be making function calls and checking errno (race condition)
- **Starting from POSIX.1 errno is no longer a global int, and it has become thread-local. So you can handle it safely.**

Pthreads API

Return value from Pthreads functions:

- The traditional method of returning status from system calls and some library functions is to return 0 on success and –1 on error, with errno being set to indicate the error.
- The functions in the Pthreads API do things differently. All Pthreads functions return 0 on success or a positive value on failure. The failure value is one of the same values that can be placed in errno by traditional UNIX system calls

POSIX Thread management functions

POSIX function	description
`pthread_cancel`	terminate another thread
`pthread_create`	create a thread
`pthread_detach`	set thread to release resources
`pthread_equal`	test two thread IDs for equality
`pthread_exit`	exit a thread without exiting process
`pthread_kill`	send a signal to a thread
`pthread_join`	wait for a thread
`pthread_self`	find out own thread ID

- 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**

Pthreads API : Thread Creation

When a program is started, the resulting process consists of a single thread, called the initial or main thread.

The *pthread_create()* function creates an additional (new) thread.

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

The new thread commences execution by calling the function identified by start with the argument arg . The thread that calls *pthread_create()* continues execution with the next statement that follows the call.

Pthreads API : Thread Termination

The execution of a thread terminates in one of the following ways:

- The thread's start function performs a return specifying a return value for the thread.
- The thread calls *pthread_exit()* .
- The thread is canceled using *pthread_cancel()*
- Any of the threads calls *exit()*, or the main thread performs a *return* (in the main() function), which causes all threads in the process to terminate immediately.

Pthreads API : Thread Termination

The *pthread_exit()* function terminates the calling thread, and specifies a return value that can be obtained in another thread by calling *pthread_join()*.

```
include <pthread.h>

void pthread_exit(void *retval);
```

Calling *pthread_exit()* is equivalent to performing a return in the thread's start function, with the difference that *pthread_exit()* can be called from any function that has been called by the thread's start function.

If the main thread calls *pthread_exit()* instead of calling *exit()* or performing a return , **then the other threads continue to execute**

Pthreads API : Thread IDs

- POSIX threads are referenced by an ID of type `pthread_t`. A thread can find out its ID by calling `pthread_self`.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- Since `pthread_t` may be a structure, use `pthread_equal` to compare thread IDs for equality. The parameters of `pthread_equal` are the thread IDs to be compared.

```
#include <pthread.h>

pthread_t pthread_equal(thread_t t1, pthread_t t2);
```

- If `t1` equals `t2`, `pthread_equal` returns a nonzero value. If the thread IDs are not equal, `pthread_equal` returns 0

Pthreads API : Detaching and Joining

- 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)
- To prevent memory leaks, long-running programs should eventually call either `pthread_detach` or `pthread_join` for every thread.

Pthreads API : Joining

- A nondetached thread's resources are not released until another thread calls `pthread_join` with the ID of the terminating thread as the first parameter.
- **The `pthread_join` function suspends the calling thread until the target thread, specified by the first parameter, terminates.**
- 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's return status

```
#include <pthread.h>

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

Pthreads API : Joining

threads/simple_thread.c

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

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}

 int
 main(int argc, char *argv[])
 {
     pthread_t t1;
     void *res;
     int s;

     s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
     if (s != 0)
         errExitEN(s, "pthread_create");

     printf("Message from main()\n");
     s = pthread_join(t1, &res);
     if (s != 0)
         errExitEN(s, "pthread_join");

     printf("Thread returned %ld\n", (long) res);

     exit(EXIT_SUCCESS);
 }
```

threads/simple_thread.c

Pthreads API : Detaching

- The `pthread_detach` function has a single parameter, `thread`, the thread ID of the thread to be detached.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- If successful, `pthread_detach` returns 0. If unsuccessful, `pthread_detach` returns a nonzero error code
- Once a thread has been detached, it is no longer possible to use *pthread_join()* to obtain its return status, and the thread can't be made joinable again. Detaching a thread doesn't make it immune to a call to *exit()* in another thread or a return in the main thread. In such an event, all threads in the process are immediately terminated, regardless of whether they are joinable or detached. To put things another way, *pthread_detach()* simply controls what happens after a thread terminates, not how or when it terminates.

Pthreads API : Thread Attributes

- As mentioned earlier that the *pthread_create() attr* argument, whose type is *pthread_attr_t*, can be used to specify the attributes used in the creation of a new thread.
- We'll just mention that these attributes include information such as the location and size of the thread's stack, the thread's scheduling policy and priority and whether the thread is joinable or detached.

Thread Attributes Example code

from **threads/detached_attrib.c**

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");

s = pthread_attr_destroy(&attr);             /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_attr_destroy");
```

from **threads/detached_attrib.c**

Thread Attributes Example code

The code shown creates a new thread that is made detached at the time of thread creation.

This code first initializes a thread attributes structure with default values, sets the attribute required to create a detached thread, and then creates a new thread using the thread attributes structure.

Once the thread has been created, the attributes object is no longer needed, and so is destroyed.

Threads Versus Processes

This lecture we briefly considered some of the factors that might influence our choice of whether to implement an application as a group of threads or as a group of processes.

We begin by considering the advantages of a multithreaded approach :

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work .
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes

Threads Versus Processes

Using threads can have some disadvantages compared to using processes

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner. Multiprocess applications don't need to be concerned with this.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.

Threads Versus Processes

- Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large, this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory .

Threads Versus Processes

The following are some other points that may influence our choice of threads versus processes:

- Dealing with signals in a multithreaded application requires careful design. (**As a general principle, it is usually desirable to avoid the use of signals in multithreaded programs.**)
- In a multithreaded application, all threads must be running the same program. In a multiprocess application, different processes can run different programs.
- Aside from data, threads also share certain other information (e.g., file descriptors, signal dispositions, current working directory, and user and group IDs). This may be an advantage or a disadvantage, depending on the application.

Summary

- In a multithreaded process, multiple threads are concurrently executing the same program. All of the threads share the same global and heap variables, but each thread has a private stack for local variables. The threads in a process also share a number of other attributes, including process ID, open file descriptors, signal dispositions, current working directory, and resource limits.
- The key difference between threads and processes is the easier sharing of information that threads provide, and this is the main reason that some application designs map better onto a multithread design than onto a multiprocess design. Threads can also provide fast performance for some operations, but this factor is usually secondary in influencing the choice of threads versus processes.

Summary

- Threads are created using *pthread_create()*. Each thread can then independently terminate using *pthread_exit()* (If any thread calls exit(), then all threads immediately terminate).
- Unless a thread has been marked as detached, it must be joined by another thread using *pthread_join()*, which returns the termination status of the joined thread.