

Lecture 9

Chapter 12 of Robbins Book

POSIX Threads



POSIX Threads

- 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.
- This lecture explains how threads are created, managed and used to solve simple problems.
- An overview of basic thread management under the POSIX standard is presented



Threads vs. Processes

- Processes
 - Programs running in their own address space
 - Code
 - Data
 - Stack
- Threads
 - Sections of code within a process that run independently
 - Share address space
 - Share global data
 - Use of less resources



A Motivating Problem: Monitoring File Descriptors

- A blocking read operation causes the calling process to block until input becomes available. Such blocking creates difficulties when a process expects input from more than one source, since the process has no way of knowing which file descriptor will produce the next input.
- The multiple file descriptor problem commonly appears in client-server programming because the server expects input from multiple clients.
- There are a couple of approaches to solve this task, we will concentrate on how to use threads...

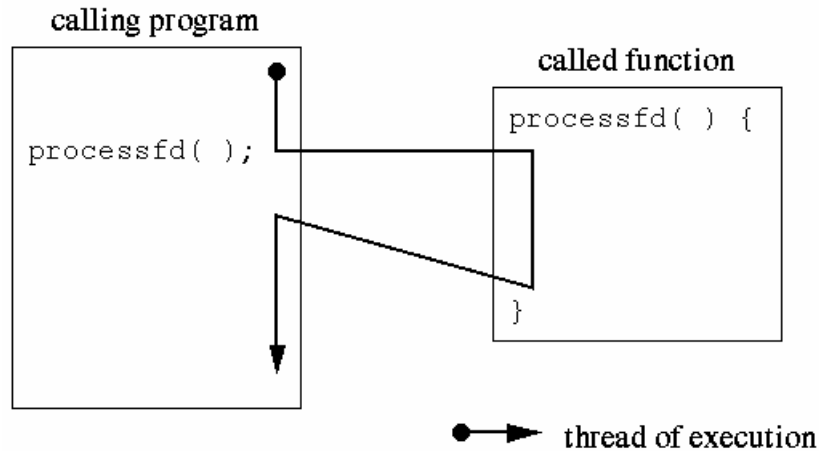


Monitoring File Descriptors

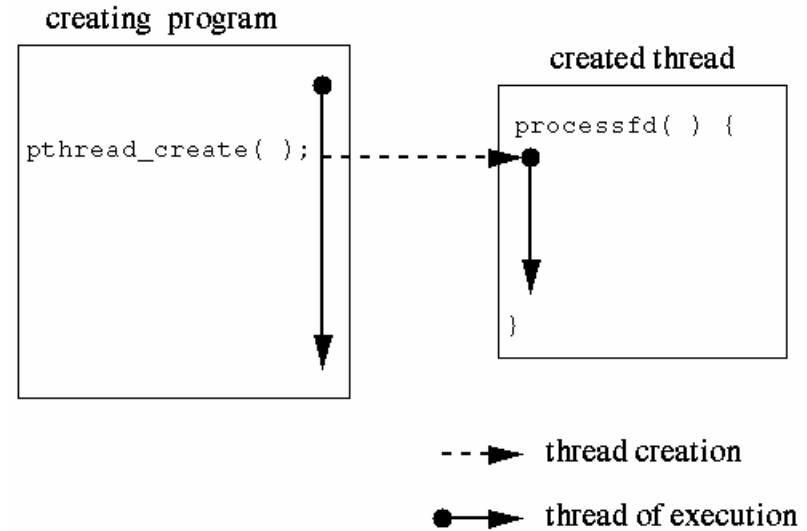
- Multiple threads can simplify the problem of monitoring multiple file descriptors because a dedicated thread with relatively simple logic can handle each file descriptor.
- Threads also make the overlap of I/O and processing transparent to the programmer
- We begin by comparing the execution of a function by a separate thread to the execution of an ordinary function call within the same thread of execution



Function and 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.



Monitoring File Descriptors

- 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.



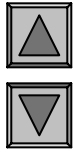
Function call and thread example

```
#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.



POSIX thread management functions

POSIX function	description
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	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



Referencing threads by ID

- 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(pthread_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



Creating a thread

- The `pthread_create` function creates a thread. POSIX `pthread_create` automatically makes the thread runnable without requiring a separate start operation.

```
#include <pthread.h>

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

- The `thread` parameter of `pthread_create` points to the ID of the newly created thread. 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. The third parameter, `start_routine`, is the name of a function that the thread calls when it begins execution. The `start_routine` takes 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`



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.



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

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

void *detachfun(void *arg) {
    int i = *((int *)arg);
    if (!pthread_detach(pthread_self()))
        return NULL;
    fprintf(stderr, "My argument is %d\n", i);
    return NULL;
}
```



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 return status

```
#include <pthread.h>

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



Exiting and cancellation

- 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`.
- A process will exit with a return status of 0 if its last thread calls `pthread_exit`.



Exiting

- The `value_ptr` value is available to a successful `pthread_join`. 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`.

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

- POSIX does not define any errors for `pthread_exit`



Cancelling

- Threads can force other threads to return through the cancellation mechanism.
- A thread calls `pthread_cancel` to request that another thread be canceled. The target thread's type and cancellability state determine the result.
- The single parameter of `pthread_cancel` is the thread ID of the target thread to be canceled. The `pthread_cancel` function does not cause the caller to block while the cancellation completes. Rather, `pthread_cancel` returns after making the cancellation request.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

- If successful, `pthread_cancel` returns 0. If unsuccessful, `pthread_cancel` returns a nonzero error code



labelling

- What happens when a thread receives a cancellation request depends on its state and type.
- If a thread has the `PTHREAD_CANCEL_ENABLE` state, it receives cancellation requests. On the other hand, if the thread has the `PTHREAD_CANCEL_DISABLE` state, the cancellation requests are held pending. By default, threads have the `PTHREAD_CANCEL_ENABLE` state.
- The `pthread_setcancelstate` function changes the cancellability state of the calling thread

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```



Passing parameters to threads and returning values

- 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



This program creates a thread to copy a file

```
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#define PERMS (S_IRUSR | S_IWUSR)
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_TRUNC)

void *copyfilemalloc(void *arg);

int main (int argc, char *argv[]) {          /* copy fromfile to tofile */
    int *bytesptr;
    int error;
    int fds[2];
    pthread_t tid;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
        return 1;
    }
    if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
        ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
        perror("Failed to open the files");
        return 1;
    }
    if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)) {
        fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
        return 1;
    }
    if (error = pthread_join(tid, (void **)&bytesptr)) {
        fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
        return 1;
    }
    printf("Number of bytes copied: %d\n", *bytesptr);
    return 0;
}
```



The copyfilemalloc function copies the contents of one file to another by calling the copyfile function

```
#include <stdlib.h>
#include <unistd.h>
#include "restart.h"

void *copyfilemalloc(void *arg) { /* copy infd to outfd with return value */
    int *bytesp;
    int infd;
    int outfd;

    infd = *((int *)(arg));
    outfd = *((int *)(arg) + 1);
    if ((bytesp = (int *)malloc(sizeof(int))) == NULL)
        return NULL;
    *bytesp = copyfile(infd, outfd);
    r_close(infd);
    r_close(outfd);
    return bytesp;
}
```



Thread Management

- 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 designated with an `_r` suffix.



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.
- [Chapter 13](#) explains how to control this type of interaction by using synchronization primitives such as mutex locks and condition variables.



User Threads versus Kernel Threads

- The two traditional models of thread control are user-level 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 more efficient than for processes but less than for user-level threads.



Thread Attributes

- POSIX takes an object-oriented approach to representation and assignment of properties by encapsulating properties such as stack size and scheduling policy into an object of type `pthread_attr_t`
- The attribute object affects a thread only at the time of creation.
- You first create an attribute object and associate properties, such as stack size and scheduling policy, with the attribute object.
- You can then create multiple threads with the same properties by passing the same thread attribute object to `pthread_create`



Thread Attributes

- Attributes behave like objects that can be created or destroyed.
 - Create an attribute object (initialize it with default properties).
 - Modify the properties of the attribute object.
 - Create a thread using the attribute object.
 - The object can be changed or reused without affecting the thread.
 - The attribute object affects the thread only at the time of thread creation.
- Initialize or destroy an attribute with:

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```



The thread state

- The state is either `PTHREAD_CREATE_JOINABLE` (the default) or `PTHREAD_CREATE_DETACHED`.

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                               int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                               int detachstate);
```

- If successful, these functions return 0. If unsuccessful, they return a nonzero error code



The following code segment creates a detached thread to run processfd.

```
int error, fd;
pthread_attr_t tattr;
pthread_t tid;

if (error = pthread_attr_init(&tattr))
    fprintf(stderr, "Failed to create attribute object: %s\n",
            strerror(error));
else if (error = pthread_attr_setdetachstate(&tattr,
            PTHREAD_CREATE_DETACHED))
    fprintf(stderr, "Failed to set attribute state to detached: %s\n",
            strerror(error));
else if (error = pthread_create(&tid, &tattr, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
```



The thread stack

- You can set a location and size for the thread stack.

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr,
                          void *stackaddr,
                          size_t stacksize);
```

- Some systems allow you to set a guard for the stack so that an overflow into the guard area can generate a SIGSEGV signal

```
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                              size_t guardsize);
```



Thread scheduling

- The *contention scope* of an object controls whether the thread competes within the process or at the system level for scheduling resources
- The `contentionscope` can be `PTHREAD_SCOPE_PROCESS` or `PTHREAD_SCOPE_SYSTEM`

```
int pthread_attr_getscope(const pthread_attr_t *restrict attr,  
                          int *restrict contentionscope);  
int pthread_attr_setscope(pthread_attr_t *attr,  
                          int contentionscope);
```



The following code segment creates a thread that contends for kernel resources

```
int error;
int fd;
pthread_attr_t tattr;
pthread_t tid;

if (error = pthread_attr_init(&tattr))
    fprintf(stderr, "Failed to create an attribute object:%s\n",
            strerror(error));
else if (error = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM))
    fprintf(stderr, "Failed to set scope to system:%s\n",
            strerror(error));
else if (error = pthread_create(&tid, &tattr, processfd, &fd))
    fprintf(stderr, "Failed to create a thread:%s\n", strerror(error));
```



Inheritance of scheduling policy:

- With `PTHREAD_INHERIT_SCHED` the scheduling attributes of the attribute object are ignored and the created thread has the same scheduling attributes of the creating thread.
- With `PTHREAD_EXPLICIT_SCHED` the scheduling is taken from the attribute.

```
int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,  
                                int *restrict inheritsched);  
int pthread_attr_setinheritsched(pthread_attr_t *attr,  
                                int inheritsched);
```



Thread scheduling

- Scheduling parameters and policy. Typical scheduling policies are `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`.
- What is supported is system dependent.
- Each policy is modified by scheduling parameters stored in a struct `sched_param`
- This may contain a priority or a quantum value.

```
int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                              struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                              const struct sched_param *restrict param);
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                               int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

