The material on the page was taken from a tutorial develop by Guy Kerens which can be found here v1.0.1

Multi-Threaded Programming With POSIX Threads

Table Of Contents:

Before We Start...

This tutorial is an attempt to help you become familiar with multi-threaded programming with the POSIX threads (pthreads) library, and attempts to show how its features can be used in "real-life" programs. It explains the different tools defined by the library, shows how to use them, and then gives an example of using them to solve programming problems. There is an implicit assumption that the user has some theoretical familiarity with paralell programming (or multi-processing) concepts. Users without such background might find the concepts harder to grasp. A seperate tutorial will be prepared to explain the theoreticl background and terms to those who are familiar only with normal "serial" programming.

I would assume that users which are familiar with asynchronous programming models, such as those used in windowing environments (X, Motif), will find it easier to grasp the concepts of multi-threaded programming.

When talking about POSIX threads, one cannot avoid the question "Which draft of the POSIX threads standard shall be used?". As this threads standard has been revised over a period of several years, one will find that implementations adhering to different drafts of the standard have a different set of functions, different default values, and different nuances. Since this tutorial was written using a Linux system with the kernel-level LinuxThreads library, v0.5, programmers with access to other systems, using different versions of pthreads, should refer to their system's manuals in case of incompatibilities. Also, since some of the example programs are using blocking system calls, they won't work with user-level threading libraries (refer to our parallel programming theory tutorial for more information).
Having said that, i'd try to check the example programs on other systems as well (Solaris 2.5 comes to mind), to make it more "cross-platform".

What Is a Thread? Why Use Threads

A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

The advantage of using a thread group instead of a normal serial program is that several

operations may be carried out in parallel, and thus events can be handled immediately as they arrive (for example, if we have one thread handling a user interface, and another thread handling database queries, we can execute a heavy query requested by the user, and still respond to user input while the query is executed).

The advantage of using a thread group over using a process group is that context switching between threads is much faster then context switching between processes (context switching means that the system switches from running one thread or process, to running another thread or process). Also, communications between two threads is usually faster and easier to implement then communications between two processes.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes won't suffer. Another advantage of using processes is that they can run on different machines, while all the threads have to run on the same machine (at least normally).

Creating And Destroying Threads

When a multi-threaded program starts executing, it has one thread running, which executes the main() function of the program. This is already a full-fledged thread, with its own thread ID. In order to create a new thread, the program should use the `pthread_create()` function. Here is how to use it:

```
#include <stdio.h>       /* standard I/O routines                 */
#include <pthread.h>     /* pthread functions and data structures */

/* function to be executed by the new thread */
void*
do_loop(void* data)
{
    int i;

    int i;                      /* counter, to print numbers */
    int j;                      /* counter, for delay        */
    int me = *((int*)data);     /* thread identifying number */

    for (i=0; i<10; i++) {
        for (j=0; j<500000; j++) /* delay loop */
            ;
        printf("'%d' - Got '%d'\n", me, i);
    }

    /* terminate the thread */
    pthread_exit(NULL);
}
```

```
/* like any C program, program's execution begins in main */
int
main(int argc, char* argv[])
{
    int        thr_id;         /* thread ID for the newly created thread */
    pthread_t  p_thread;       /* thread's structure                     */
    int        a         = 1;  /* thread 1 identifying number            */
    int        b         = 2;  /* thread 2 identifying number            */

    /* create a new thread that will execute 'do_loop()' */
    thr_id = pthread_create(&p_thread, NULL, do_loop, (void*)&a);
    /* run 'do_loop()' in the main thread as well */
    do_loop((void*)&b);

    /* NOT REACHED */
    return 0;
}
```

A few notes should be mentioned about this program:

1. Note that the main program is also a thread, so it executes the `do_loop()` function in parallel to the thread it creates.
2. `pthread_create()` gets 4 parameters. The first parameter is used by `pthread_create()` to supply the program with information about the thread. The second parameter is used to set some attributes for the new thread. In our case we supplied a NULL pointer to tell `pthread_create()` to use the default values. The third parameter is the name of the function that the thread will start executing. The forth parameter is an argument to pass to this function. Note the cast to a 'void*'. It is not required by ANSI-C syntax, but is placed here for clarification.
3. The delay loop inside the function is used only to demonstrate that the threads are executing in parallel. Use a larger delay value if your CPU runs too fast, and you see all the printouts of one thread before the other.
4. The call to `pthread_exit()` Causes the current thread to exit and free any thread-specific resources it is taking. There is no need to use this call at the end of the thread's top function, since when it returns, the thread would exit automatically anyway. This function is useful if we want to exit a thread in the middle of its execution.

In order to compile a multi-threaded program using `gcc`, we need to link it with the pthreads library. Assuming you have this library already installed on your system, here is how to compile our first program:

```
gcc pthread_create.c -o pthread_create -lpthread
```

The source code for this program may be found in the pthread_create.c file.

Synchronizing Threads With Mutexes

One of the basic problems when running several threads that use the same memory space, is making sure they don't "step on each other's toes". By this we refer to the problem of using a data structure from two different threads.

For instance, consider the case where two threads try to update two variables. One tries to set both to 0, and the other tries to set both to 1. If both threads would try to do that at the same time, we might get with a situation where one variable contains 1, and one contains 0. This is because a context-switch (we already know what this is by now, right?) might occur after the first tread zeroed out the first variable, then the second thread would set both variables to 1, and when the first thread resumes operation, it will zero out the second variable, thus getting the first variable set to '1', and the second set to '0'.

What Is A Mutex?

A basic mechanism supplied by the pthreads library to solve this problem, is called a mutex. A mutex is a lock that guarantees three things:

1. Atomicity - Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread succeeded in locking this mutex at the same time.
2. Singularity - If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the thread until the original thread releases the lock.
3. Non-Busy Wait - If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources) until the lock is freed by the second thread. At this time, the first thread will wake up and continue execution, having the mutex locked by it.

From these three points we can see how a mutex can be used to assure exclusive access to variables (or in general critical code sections). Here is some pseudo-code that updates the two variables we were talking about in the previous section, and can be used by the first thread:

```
lock mutex 'X1'.
set first variable to '0'.
set second variable to '0'.
unlock mutex 'X1'.
```

Meanwhile, the second thread will do something like this:

```
lock mutex 'X1'.
set first variable to '1'.
set second variable to '1'.
```

```
unlock mutex 'X1'.
```

Assuming both threads use the same mutex, we are assured that after they both ran through this code, either both variables are set to '0', or both are set to '1'. You'd note this requires some work from the programmer - If a third thread was to access these variables via some code that does not use this mutex, it still might mess up the variable's contents. Thus, it is important to enclose all the code that accesses these variables in a small set of functions, and always use only these functions to access these variables.

Creating And Initializing A Mutex

In order to create a mutex, we first need to declare a variable of type `pthread_mutex_t` , and then initialize it. The simplest way it by assigning it the `PTHREAD_MUTEX_INITIALIZER` constant. So we'll use a code that looks something like this:

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
```

One note should be made here: This type of initialization creates a mutex called 'fast mutex'. This means that if a thread locks the mutex and then tries to lock it again, it'll get stuck - it will be in a deadlock.

There is another type of mutex, called 'recursive mutex', which allows the thread that locked it, to lock it several more times, without getting blocked (but other threads that try to lock the mutex now will get blocked). If the thread then unlocks the mutex, it'll still be locked, until it is unlocked the same amount of times as it was locked. This is similar to the way modern door locks work - if you turned it twice clockwise to lock it, you need to turn it twice counter-clockwise to unlock it. This kind of mutex can be created by assigning the constant `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` to a mutex variable.

Locking And Unlocking A Mutex

In order to lock a mutex, we may use the function `pthread_mutex_lock()`. This function attempts to lock the mutex, or block the thread if the mutex is already locked by another thread. In this case, when the mutex is unlocked by the first process, the function will return with the mutex locked by our process. Here is how to lock a mutex (assuming it was initialized earlier):

```
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
/* mutex is now locked - do your stuff. */
.
.
```

After the thread did what it had to (change variables or data structures, handle file, or whatever it intended to do), it should free the mutex, using the `pthread_mutex_unlock()` function, like this:

```
rc = pthread_mutex_unlock(&a_mutex);
if (rc) {
    perror("pthread_mutex_unlock");
    pthread_exit(NULL);
}
```

Destroying A Mutex

After we finished using a mutex, we should destroy it. Finished using means no thread needs it at all. If only one thread finished with the mutex, it should leave it alive, for the other threads that might still need to use it. Once all finished using it, the last one can destroy it using the `pthread_mutex_destroy()` function:

```
rc = pthread_mutex_destroy(&a_mutex);
```

After this call, this variable (a_mutex) may not be used as a mutex any more, unless it is initialized again. Thus, if one destroys a mutex too early, and another thread tries to lock or unlock it, that thread will get a `EINVAL` error code from the lock or unlock function.

Using A Mutex - A Complete Example

After we have seen the full life cycle of a mutex, lets see an example program that uses a mutex. The program introduces two employees competing for the "employee of the day" title, and the glory that comes with it. To simulate that in a rapid pace, the program employs 3 threads: one that promotes Danny to "employee of the day", one that promotes Moshe to

that situation, and a third thread that makes sure that the employee of the day's contents is consistent (i.e. contains exactly the data of one employee).
Two copies of the program are supplied. One that uses a mutex, and one that does not. Try them both, to see the differences, and be convinced that mutexes are essential in a multi-threaded environment.

The programs themselves are in the files accompanying this tutorial. The one that uses a mutex is employee-with-mutex.c. The one that does not use a mutex is employee-without-mutex.c. Read the comments inside the source files to get a better understanding of how they work.

Starvation And Deadlock Situations

Again we should remember that `pthread_mutex_lock()` might block for a non-determined duration, in case of the mutex being already locked. If it remains locked forever, it is said that our poor thread is "starved" - it was trying to acquire a resource, but never got it. It is up to the programmer to ensure that such starvation won't occur. The pthread library does not help us with that.

The pthread library might, however, figure out a "deadlock". A deadlock is a situation in which a set of threads are all waiting for resources taken by other threads, all in the same set. Naturally, if all threads are blocked waiting for a mutex, none of them will ever come back to life again. The pthread library keeps track of such situations, and thus would fail the last thread trying to call `pthread_mutex_lock()`, with an error of type `EDEADLK`. The programmer should check for such a value, and take steps to solve the deadlock somehow.

Refined Synchronization - Condition Variables

As we've seen before with mutexes, they allow for simple coordination - exclusive access to a resource. However, we often need to be able to make real synchronization between threads:

- In a server, one thread reads requests from clients, and dispatches them to several threads for handling. These threads need to be notified when there is data to process, otherwise they should wait without consuming CPU time.
- In a GUI (Graphical User Interface) Application, one thread reads user input, another handles graphical output, and a third thread sends requests to a server and handles its replies. The server-handling thread needs to be able to notify the graphics-drawing thread when a reply from the server arrived, so it will immediately show it to the user. The user-input thread needs to be always responsive to the user, for example, to allow her to cancel long operations currently executed by the server-handling thread.

All these examples require the ability to send notifications between threads. This is where condition variables are brought into the picture.

What Is A Condition Variable?

A condition variable is a mechanism that allows threads to wait (without wasting CPU cycles) for some even to occur. Several threads may wait on a condition variable, until some other thread signals this condition variable (thus sending a notification). At this time, one of the threads waiting on this condition variable wakes up, and can act on the event. It is possible to also wake up all threads waiting on this condition variable by using a broadcast method on this variable.

Note that a condition variable does not provide locking. Thus, a mutex is used along with the condition variable, to provide the necessary locking when accessing this condition variable.

Creating And Initializing A Condition Variable

Creation of a condition variable requires defining a variable of type `pthread_cond_t`, and initializing it properly. Initialization may be done with either a simple use of a macro named `PTHREAD_COND_INITIALIZER` or the usage of the `pthread_cond_init()` function. We will show the first form here:

```
pthread_cond_t got_request = PTHREAD_COND_INITIALIZER;
```

This defines a condition variable named 'got_request', and initializes it.

Note: since the `PTHREAD_COND_INITIALIZER` is actually a structure, it may be used to initialize a condition variable only when it is declared. In order to initialize it during runtime, one must use the `pthread_cond_init()` function.

Signaling A Condition Variable

In order to signal a condition variable, one should either the `pthread_cond_signal()` function (to wake up a only one thread waiting on this variable), or the `pthread_cond_broadcast()` function (to wake up all threads waiting on this variable). Here is an example using signal, assuming 'got_request' is a properly initialized condition variable:

```
int rc = pthread_cond_signal(&got_request);
```

Or by using the broadcast function:

```
int rc = pthread_cond_broadcast(&got_request);
```

When either function returns, 'rc' is set to 0 on success, and to a non-zero value on failure. In such a case (failure), the return value denotes the error that occured (`EINVAL` denotes that the given parameter is not a condition variable. `ENOMEM` denotes that the system has run out of memory.

Note: success of a signaling operation does not mean any thread was awakened - it might be that no thread was waiting on the condition variable, and thus the signaling does nothing (i.e. the signal is lost).
It is also not remembered for future use - if after the signaling function returns another thread starts waiting on this condition variable, a further signal is required to wake it up.

Waiting On A Condition Variable

If one thread signals the condition variable, other threads would probably want to wait for this signal. They may do so using one of two functions, `pthread_cond_wait()` or `pthread_cond_timedwait()`. Each of these functions takes a condition variable, and a mutex (which should be locked before calling the wait function), unlocks the mutex, and waits until the condition variable is signaled, suspending the thread's execution. If this signaling causes the thread to awake (see discussion of `pthread_cond_signal()` earlier), the mutex is automagically locked again by the wait funciton, and the wait function returns.

The only difference between these two functions is that `pthread_cond_timedwait()` allows the programmer to specify a timeout for the waiting, after which the function always returns, with a proper error value (ETIMEDOUT) to notify that condition variable was NOT signaled before the timeout passed. The `pthread_cond_wait()` would wait indefinitely if it was never signaled.

Here is how to use these two functions. We make the assumption that 'got_request' is a properly initialized condition variable, and that 'request_mutex' is a properly initialized mutex. First, we try the `pthread_cond_wait()` function:

```
/* first, lock the mutex */
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
/* mutex is now locked - wait on the condition variable.             */
/* During the execution of pthread_cond_wait, the mutex is unlocked. */
rc = pthread_cond_wait(&got_request, &a_mutex);
if (rc == 0) { /* we were awakened due to the cond. variable being signaled */
               /* The mutex is now locked again by pthread_cond_wait()      */
    /* do your stuff... */
```

```
    .
}
/* finally, unlock the mutex */
pthread_mutex_unlock(&a_mutex);
```

Now an example using the `pthread_cond_timedwait()` function:

```
#include <sys/time.h>     /* struct timeval definition           */
#include <unistd.h>       /* declaration of gettimeofday()       */

struct timeval  now;            /* time when we started waiting        */
struct timespec timeout;        /* timeout value for the wait function */
int             done;           /* are we done waiting?                */

/* first, lock the mutex */
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
/* mutex is now locked */

/* get current time */
gettimeofday(&now);
/* prepare timeout value */
timeout.tv_sec = now.tv_sec + 5
timeout.tv_nsec = now.tv_usec * 1000; /* timeval uses microseconds.         */
                                      /* timespec uses nanoseconds.         */
                                      /* 1 nanosecond = 1000 micro seconds. */

/* wait on the condition variable. */
/* we use a loop, since a Unix signal might stop the wait before the timeout */
done = 0;
while (!done) {
    /* remember that pthread_cond_timedwait() unlocks the mutex on entrance */
    rc = pthread_cond_timedwait(&got_request, &a_mutex, &timeout);
    switch(rc) {
        case 0:  /* we were awakened due to the cond. variable being signaled */
                 /* the mutex was now locked again by pthread_cond_timedwait. */
            /* do your stuff here... */
            .
            .
            done = 0;
            break;
        case ETIMEDOUT: /* our time is up */
            done = 0;
            break;
        default:        /* some error occurred (e.g. we got a Unix signal) */
            break;      /* break this switch, but re-do the while loop.   */
    }
}
/* finally, unlock the mutex */
```

```
pthread_mutex_unlock(&a_mutex);
```

As you can see, the timed wait version is way more complex, and thus better be wrapped up by some function, rather then being re-coded in every necessary location.

Note: it might be that a condition variable that has 2 or more threads waiting on it is signaled many times, and yet one of the threads waiting on it never awakened. This is because we are not guaranteed which of the waiting threads is awakened when the variable is signaled. It might be that the awakened thread quickly comes back to waiting on the condition variables, and gets awakened again when the variable is signaled again, and so on. The situation for the un-awakened thread is called 'starvation'. It is up to the programmer to make sure this situation does not occur if it implies bad behavior. Yet, in our server example from before, this situation might indicate requests are coming in a very slow pace, and thus perhaps we have too many threads waiting to service requests. In this case, this situation is actually good, as it means every request is handled immediately when it arrives.

Note 2: when the mutex is being broadcast (using pthread_cond_broadcast), this does not mean all threads are running together. Each of them tries to lock the mutex again before returning from their wait function, and thus they'll start running one by one, each one locking the mutex, doing their work, and freeing the mutex before the next thread gets its chance to run.

Destroying A Condition Variable

After we are done using a condition variable, we should destroy it, to free any system resources it might be using. This can be done using the `pthread_cond_destroy()`. In order for this to work, there should be no threads waiting on this condition variable. Here is how to use this function, again, assuming 'got_request' is a pre-initialized condition variable:

```
int rc = pthread_cond_destroy(&got_request);
if (rc == EBUSY) { /* some thread is still waiting on this condition variable */
    /* handle this case here... */
    .
    .
}
```

What if some thread is still waiting on this variable? depending on the case, it might imply some flaw in the usage of this variable, or just lack of proper thread cleanup code. It is probably good to alert the programmer, at least during debug phase of the program, of such a case. It might mean nothing, but it might be significant.

A Real Condition For A Condition Variable

A note should be taken about condition variables - they are usually pointless without some real condition checking combined with them. To make this clear, lets consider the server example we introduced earlier. Assume that we use the 'got_request' condition variable to signal that a new request has arrived that needs handling, and is held in some requests queue. If we had threads waiting on the condition variable when this variable is signaled, we are assured that one of these threads will awake and handle this request.

However, what if all threads are busy handling previous requests, when a new one arrives? the signaling of the condition variable will do nothing (since all threads are busy doing other things, NOT waiting on the condition variable now), and after all threads finish handling their current request, they come back to wait on the variable, which won't necessarily be signaled again (for example, if no new requests arrive). Thus, there is at least one request pending, while all handling threads are blocked, waiting for a signal.

In order to overcome this problem, we may set some integer variable to denote the number of pending requests, and have each thread check the value of this variable before waiting on the variable. If this variable's value is positive, some request is pending, and the thread should go and handle it, instead of going to sleep. Further more, a thread that handled a request, should reduce the value of this variable by one, to make the count correct.
Lets see how this affects the waiting code we have seen above.

```
/* number of pending requests, initially none */
int num_requests = 0;
.
.
/* first, lock the mutex */
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
/* mutex is now locked - wait on the condition variable */
/* if there are no requests to be handled.              */
rc = 0;
if (num_requests == 0)
    rc = pthread_cond_wait(&got_request, &a_mutex);
if (num_requests > 0 && rc == 0) { /* we have a request pending */
        /* do your stuff... */
        .
        .
        /* decrease count of pending requests */
        num_requests--;
    }
}
/* finally, unlock the mutex */
pthread_mutex_unlock(&a_mutex);
```

Using A Condition Variable - A Complete Example

As an example for the actual usage of condition variables, we will show a program that simulates the server we have described earlier - one thread, the receiver, gets client requests. It inserts the requests to a linked list, and a hoard of threads, the handlers, are handling these requests. For simplicity, in our simulation, the receiver thread creates requests and does not read them from real clients.

The program source is available in the file thread-pool-server.c, and contains many comments. Please read the source file first, and then read the following clarifying notes.

1. The 'main' function first launches the handler threads, and then performs the chore of the receiver thread, via its main loop.
2. A single mutex is used both to protect the condition variable, and to protect the linked list of waiting requests. This simplifies the design. As an exercise, you may think how to divide these roles into two mutexes.
3. The mutex itself MUST be a recursive mutex. In order to see why, look at the code of the 'handle_requests_loop' function. You will notice that it first locks the mutex, and afterwards calls the 'get_request' function, which locks the mutex again. If we used a non-recursive mutex, we'd get locked indefinitely in the mutex locking operation of the 'get_request' function.
   You may argue that we could remove the mutex locking in the 'get_request' function, and thus remove the double-locking problem, but this is a flawed design - in a larger program, we might call the 'get_request' function from other places in the code, and we'll need to check for proper locking of the mutex in each of them.
4. As a rule, when using recursive mutexes, we should try to make sure that each lock operation is accompanied by a matching unlock operation in the same function. Otherwise, it will be very hard to make sure that after locking the mutex several times, it is being unlocked the same number of times, and deadlocks would occur.
5. The implicit unlocking and re-locking of the mutex on the call to the `pthread_cond_wait()` function is confusing at first. It is best to add a comment regarding this behavior in the code, or else someone that reads this code might accidentally add a further mutex lock.

"Private" thread data - Thread-Specific Data

In "normal", single-thread programs, we sometimes find the need to use a global variable. It is frequently a bad practice to have global variables, but they sometimes do come handy. Especially if they are static variables - meaning, they are recognized only on the scope of a single file.

In multi-threaded programs, we also might find a need for such variables. We should note,

however, that the same variable is accessible from all the threads, so we need to protect access to it using a mutex, which is extra overhead. Further more, we sometimes need to have a variable that is 'global', but only for a specific thread. Or the same 'global' variable should have different values in different threads. For example, consider a program that needs to have one globally accessible linked list in each thread, but not the same list. Further, we want the same code to be executed by all threads. In this case, the global pointer to the start of the list should be point to a different address in each thread.

In order to have such a pointer, we need a mechanism that enables the same global variable to have a different location in memory. This is what the thread-specific data mechanism is used for.

Overview Of Thread-Specific Data Support

In the thread-specific data (TSD) mechanism, we have notions of keys and values. Each key has a name, and pointer to some memory area. Keys with the same name in two separate threads always point to different memory locations - this is handled by the library functions that allocate memory blocks to be accessed via these keys. We have a function to create a key (invoked once per key name for the whole process), a function to allocate memory (invoked separately in each thread), and functions to de-allocate this memory for a specific thread, and a function to destroy the key, again, process-wide. we also have functions to access the data pointed to by a key, either setting its value, or returning the value it points to.

Allocating Thread-Specific Data Block

The `pthread_key_create()` function is used to allocate a new key. This key now becomes valid for all threads in our process. When a key is created, the value it points to defaults to NULL. Later on each thread may change its copy of the value as it wishes. Here is how to use this function:

```
/* rc is used to contain return values of pthread functions */
int rc;
/* define a variable to hold the key, once created.         */
pthread_key_t list_key;
/* cleanup_list is a function that can clean up some data   */
/* it is specific to our program, not to TSD                */
extern void* cleanup_list(void*);

/* create the key, supplying a function that'll be invoked when it's deleted. */
rc = pthread_key_create(&list_key, cleanup_list);
```

Some notes:

1. After `pthread_key_create()` returns, the variable 'list_key' points to the newly created key.
2. The function pointer passed as second parameter to `pthread_key_create()`, will be automatically invoked by the pthread library when our thread exits, with a pointer to the key's value as its parameter. We may supply a NULL pointer as the function pointer, and then no function will be invoked for key. Note that the function will be invoked once in each thread, even thought we created this key only once, in one thread.
   If we created several keys, their associated destructor functions will be called in an arbitrary order, regardless of the order of keys creation.
3. If the `pthread_key_create()` function succeeds, it returns 0. Otherwise, it returns some error code.
4. There is a limit of `PTHREAD_KEYS_MAX` keys that may exist in our process at any given time. An attempt to create a key after `PTHREAD_KEYS_MAX` exits, will cause a return value of EAGAIN from the `pthread_key_create()` function.

Accessing Thread-Specific Data

After we have created a key, we may access its value using two pthread functions: `pthread_getspecific()` and `pthread_setspecific()`. The first is used to get the value of a given key, and the second is used to set the data of a given key. A key's value is simply a void pointer (void*), so we can store in it anything that we want. Lets see how to use these functions. We assume that 'a_key' is a properly initialized variable of type `pthread_key_t` that contains a previously created key:

```
/* this variable will be used to store return codes of pthread functions */
int rc;

/* define a variable into which we'll store some data */
/* for example, and integer.                          */
int* p_num = (int*)malloc(sizeof(int));
if (!p_num) {
    fprintf(stderr, "malloc: out of memory\n";
    exit(1);
}
/* initialize our variable to some value */
(*p_num) = 4;

/* now lets store this value in our TSD key.    */
/* note that we don't store 'p_num' in our key. */
/* we store the value that p_num points to.     */
rc = pthread_setspecific(a_key, (void*)p_num);

.
.
/* and somewhere later in our code... */
```

```
.
.
/* get the value of key 'a_key' and print it. */
{
    int* p_keyval = (int*)pthread_getspecific(a_key);

    if (p_keyval != NULL) {
        printf("value of 'a_key' is: %d\n", *p_keyval);
    }
}
```

Note that if we set the value of the key in one thread, and try to get it in another thread, we will get a NULL, since this value is distinct for each thread.

Note also that there are two cases where `pthread_getspecific()` might return NULL:

1. The key supplied as a parameter is invalid (e.g. its key wasn't created).
2. The value of this key is NULL. This means it either wasn't initialized, or was set to NULL explicitly by a previous call to `pthread_setspecific()`.

Deleting Thread-Specific Data Block

The `pthread_key_delete()` function may be used to delete keys. But do not be confused by this function's name: it does not delete memory associated with this key, nor does it call the destructor function defined during the key's creation. Thus, you still need to do memory cleanup on your own if you need to free this memory during runtime. However, since usage of global variables (and thus also thread-specific data), you usually don't need to free this memory until the thread terminate, in which case the pthread library will invoke your destructor functions anyway.

Using this function is simple. Assuming list_key is a `pthread_key_t` variable pointing to a properly created key, use this function like this:

```
int rc = pthread_key_delete(key);
```

the function will return 0 on success, or EINVAL if the supplied variable does not point to a valid TSD key.

Example

thrd_specific.c

pthread_key_create

pthread_once

Thread Cancellation And Termination

As we create threads, we need to think about terminating them as well. There are several issues involved here. We need to be able to terminate threads cleanly. Unlike processes, where a very ugly method of using signals is used, the folks that designed the pthreads library were a little more thoughtful. So they supplied us with a whole system of canceling a thread, cleaning up after a thread, and so on. We will discuss these methods here.

Canceling A Thread

When we want to terminate a thread, we can use the `pthread_cancel` function. This function gets a thread ID as a parameter, and sends a cancellation request to this thread. What this thread does with this request depends on its state. It might act on it immediately, it might act on it when it gets to a cancellation point (discussed below), or it might completely ignore it. We'll see later how to set the state of a thread and define how it acts on cancellation requests. Lets first see how to use the cancel function. We assume that 'thr_id' is a variable of type `pthread_id` containing the ID of a running thread:

```
pthread_cancel(thr_id);
```

The `pthread_cancel()` function returns 0, so we cannot know if it succeeded or not.

Setting Thread Cancellation State

A thread's cancel state may be modified using several methods. The first is by using the `pthread_setcancelstate()` function. This function defines whether the thread will accept cancellation requests or not. The function takes two arguments. One that sets the new cancel state, and one into which the previous cancel state is stored by the function. Here is how it is used:

```
int old_cancel_state;
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);
```

This will disable canceling this thread. We can also enable canceling the thread like this:

```
int old_cancel_state;
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old_cancel_state);
```

Note that you may supply a NULL pointer as the second parameter, and then you won't get the old cancel state.

A similar function, named `pthread_setcanceltype()` is used to define how a thread responds to a cancellation request, assuming it is in the 'ENABLED' cancel state. One option is to handle the request immediately (asynchronously). The other is to defer the request until a cancellation point. To set the first option (asynchronous cancellation), do something like:

```
int old_cancel_type;
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &old_cancel_type);
```

And to set the second option (deferred cancellation):

```
int old_cancel_type;
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &old_cancel_type);
```

Note that you may supply a NULL pointer as the second parameter, and then you won't get the old cancel type.

You might wonder - "What if i never set the cancellation state or type of a thread?". Well, in such a case, the `pthread_create()` function automatically sets the thread to enabled deferred cancellation, that is, `PTHREAD_CANCEL_ENABLE` for the cancel mode, and `PTHREAD_CANCEL_DEFERRED` for the cancel type.

Cancellation Points

As we've seen, a thread might be in a state where it does not handle cancel requests immediately, but rather defers them until it reaches a cancellation point. So what are these cancellation points?

In general, any function that might suspend the execution of a thread for a long time, should be a cancellation point. In practice, this depends on the specific implementation, and how conformant it is to the relevant POSIX standard (and which version of the standard it conforms to...). The following set of pthread functions serve as cancellation points:

- `pthread_join()`
- `pthread_cond_wait()`
- `pthread_cond_timedwait()`
- `pthread_testcancel()`
- `sem_wait()`
- `sigwait()`

This means that if a thread executes any of these functions, it'll check for deferred cancel requests. If there is one, it will execute the cancellation sequence, and terminate. Out of these functions, `pthread_testcancel()` is unique - it's only purpose is to test whether a cancellation request is pending for this thread. If there is, it executes the cancellation sequence. If not, it returns immediately. This function may be used in a thread that does a lot of processing without getting into a "natural" cancellation state.

Setting Thread Cleanup Functions

One of the features the pthreads library supplies is the ability for a thread to clean up after itself, before it exits. This is done by specifying one or more functions that will be called automatically by the pthreads library when the thread exits, either due to its own will (e.g. calling `pthread_exit()`), or due to it being canceled.

Two functions are supplied for this purpose. The `pthread_cleanup_push()` function is used to add a cleanup function to the set of cleanup functions for the current thread. The `pthread_cleanup_pop()` function removes the last function added with `pthread_cleanup_push()`. When the thread terminates, its cleanup functions are called in the reverse order of their registration. So the the last one to be registered is the first one to be called.

When the cleanup functions are called, each one is supplied with one parameter, that was supplied as the second parameter to the `pthread_cleanup_push()` function call. Lets see how these functions may be used. In our example we'll see how these functions may be used to clean up some memory that our thread allocates when it starts running.

```
/* first, here is the cleanup function we want to register.        */
/* it gets a pointer to the allocated memory, and simply frees it. */
void
cleanup_after_malloc(void* allocated_memory)
{
    if (allocated_memory)
        free(allocated_memory);
}

/* and here is our thread's function.      */
/* we use the same function we used in our */
/* thread-pool server.                     */
void*
```

```
handle_requests_loop(void* data)
{
    .
    .
    /* this variable will be used later. please read on...         */
    int old_cancel_type;

    /* allocate some memory to hold the start time of this thread. */
    /* assume MAX_TIME_LEN is a previously defined macro.          */
    char* start_time = (char*)malloc(MAX_TIME_LEN);

    /* push our cleanup handler. */
    pthread_cleanup_push(cleanup_after_malloc, (void*)start_time);
    .
    .
    /* here we start the thread's main loop, and do whatever is desired.. */
    .
    .
    .

    /* and finally, we unregister the cleanup handler. our method may seem */
    /* awkward, but please read the comments below for an explanation.     */

    /* put the thread in deferred cancellation mode.      */
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &old_cancel_type);

    /* supplying '1' means to execute the cleanup handler */
    /* prior to unregistering it. supplying '0' would     */
    /* have meant not to execute it.                      */
    pthread_cleanup_pop(1);

    /* restore the thread's previous cancellation mode.   */
    pthread_setcanceltype(old_cancel_type, NULL);
}
```

As we can see, we allocated some memory here, and registered a cleanup handler that will free this memory when our thread exits. After the execution of the main loop of our thread, we unregistered the cleanup handler. This must be done in the same function that registered the cleanup handler, and in the same nesting level, since both `pthread_cleanup_pop()` and `pthread_cleanup_pop()` functions are actually macros that add a '{' symbol and a '}' symbol, respectively.

As to the reason that we used that complex piece of code to unregister the cleanup handler, this is done to assure that our thread won't get canceled in the middle of the execution of our cleanup handler. This could have happened if our thread was in asynchronous cancellation mode. Thus, we made sure it was in deferred cancellation mode, then unregistered the cleanup handler, and finally restored whatever cancellation mode our thread was in previously. Note that we still assume the thread cannot be canceled in the execution of `pthread_cleanup_pop()` itself - this is true, since `pthread_cleanup_pop()` is not a cancellation point.

Synchronizing On Threads Exiting

Sometimes it is desired for a thread to wait for the end of execution of another thread. This can be done using the `pthread_join()` function. It receives two parameters: a variable of type `pthread_t`, denoting the thread to be joined, and an address of a `void*` variable, into which the exit code of the thread will be placed (or `PTHREAD_CANCELED` if the joined thread was canceled). The `pthread_join()` function suspends the execution of the calling thread until the joined thread is terminated.

For example, consider our earlier thread pool server. Looking back at the code, you'll see that we used an odd `sleep()` call before terminating the process. We did this since the main thread had no idea when the other threads finished processing all pending requests. We could have solved it by making the main thread run a loop of checking if no more requests are pending, but that would be a busy loop.

A cleaner way of implementing this, is by adding three changes to the code:

1. Tell the handler threads when we are done creating requests, by setting some flag.
2. Make the threads check, whenever the requests queue is empty, whether or not new requests are supposed to be generated. If not, then the thread should exit.
3. Make the main thread wait for the end of execution of each of the threads it spawned.

The first 2 changes are rather easy. We create a global variable named 'done_creating_requests' and set it to '0' initially. Each thread checks the contents of this variable every time before it intends to go to wait on the condition variable (i.e. the requests queue is empty).
The main thread is modified to set this variable to '1' after it finished generating all requests. Then the condition variable is being broadcast, in case any of the threads is waiting on it, to make sure all threads go and check the 'done_creating_requests' flag.

The last change is done using a `pthread_join()` loop: call `pthread_join()` once for each handler thread. This way, we know that only after all handler threads have exited, this loop is finished, and then we may safely terminate the process. If we didn't use this loop, we might terminate the process while one of the handler threads is still handling a request.

The modified program is available in the file named thread-pool-server-with-join.c. Look for the word 'CHANGE' (in capital letters) to see the locations of the three changes.

Detaching A Thread

We have seen how threads can be joined using the `pthread_join()` function. In fact, threads that are in a 'join-able' state, must be joined by other threads, or else their memory resources will not be fully cleaned out. This is similar to what happens with processes whose parents didn't clean up after them (also called 'orphan' or 'zombie' processes).

If we have a thread that we wish would exit whenever it wants without the need to join it, we should put it in the detached state. This can be done either with appropriate flags to the `pthread_create()` function, or by using the `pthread_detach()` function. We'll consider the second option in our tutorial.

The `pthread_detach()` function gets one parameter, of type `pthread_t`, that denotes the thread we wish to put in the detached state. For example, we can create a thread and immediately detach it with a code similar to this:

```
pthread_t a_thread;   /* store the thread's structure here             */
int rc;               /* return value for pthread functions.           */
extern void* thread_loop(void*); /* declare the thread's main function. */

/* create the new thread. */
rc = pthread_create(&a_thread, NULL, thread_loop, NULL);

/* and if that succeeded, detach the newly created thread. */
if (rc == 0) {
    rc = pthread_detach(a_thread);
}
```

Of-course, if we wish to have a thread in the detached state immediately, using the first option (setting the detached state directly when calling `pthread_create()` is more efficient.

Threads Cancellation - A Complete Example

Our next example is much larger then the previous examples. It demonstrates how one could write a multi-threaded program in C, in a more or less clean manner. We take our previous thread-pool server, and enhance it in two ways. First, we add the ability to tune the number of handler threads based on the requests load. New threads are created if the requests queue becomes too large, and after the queue becomes shorter again, extra threads are canceled.

Second, we fix up the termination of the server when there are no more new requests to handle. Instead of the ugly sleep we used in our first example, this time the main thread waits for all threads to finish handling their last requests, by joining each of them using `pthread_join()`.

The code is now being split to 4 separate files, as follows:

1. requests_queue.c - This file contains functions to manipulate a requests queue. We took the `add_request()` and `get_request()` functions and put them here, along with a data structure that contains all the variables previously defined as globals - pointer to

queue's head, counter of requests, and even pointers to the queue's mutex and condition variable. This way, all the manipulation of the data is done in a single file, and all its functions receive a pointer to a 'requests_queue' structure.
2. handler_thread.c - this contains the functions executed by each handler thread - a function that runs the main loop (an enhanced version of the 'handle_requests_loop()' function, and a few local functions explained below). We also define a data structure to collect all the data we want to pass to each thread. We pass a pointer to such a structure as a parameter to the thread's function in the `pthread_create()` call, instead of using a bunch of ugly globals: the thread's ID, a pointer to the requests queue structure, and pointers to the mutex and condition variable to be used.
3. handler_threads_pool.c - here we define an abstraction of a thread pool. We have a function to create a thread, a function to delete (cancel) a thread, and a function to delete all active handler threads, called during program termination. we define here a structure similar to that used to hold the requests queue, and thus the functions are similar. However, because we only access this pool from one thread, the main thread, we don't need to protect it using a mutex. This saves some overhead caused by mutexes. the overhead is small, but for a busy server, it might begin to become noticeable.
4. main.c - and finally, the main function to rule them all, and in the system bind them. This function creates a requests queue, creates a threads pool, creates few handler threads, and then starts generating requests. After adding a request to the queue, it checks the queue size and the number of active handler threads, and adjusts the number of threads to the size of the queue. We use a simple water-marks algorithm here, but as you can see from the code, it can be easily be replaced by a more sophisticated algorithm. In our water-marks algorithm implementation, when the high water-mark is reached, we start creating new handler threads, to empty the queue faster. Later, when the low water-mark is reached, we start canceling the extra threads, until we are left with the original number of handler threads.

After rewriting the program in a more manageable manner, we added code that uses the newly learned pthreads functions, as follows:

1. Each handler thread created puts itself in the deferred cancellation mode. This makes sure that when it gets canceled, it can finish handling its current request, before terminating.
2. Each handler thread also registers a cleanup function, to unlock the mutex when it terminates. This is done, since a thread is most likely to get canceled when calling `pthread_cond_wait()`, which is a cancellation point. Since the function is called with the mutex locked, it might cause the thread to exit and cause all other threads to 'hang' on the mutex. Thus, unlocking the mutex in a cleanup handler (registered with the `pthread_cleanup_push()` function) is the proper solution.
3. Finally, the main thread is set to clean up properly, and not brutally, as we did before. When it wishes to terminate, it calls the 'delete_handler_threads_pool()' function, which calls `pthread_join` for each remaining handler thread. This way, the function

returns only after all handler threads finished handling their last request.

Please refer to the source code for the full details. Reading the header files first will make it easier to understand the design. To compile the program, just switch to the thread-pool-server-changes directory, and type 'gmake'.

Using Threads For Responsive User Interface Programming

One area in which threads can be very helpful is in user-interface programs. These programs are usually centered around a loop of reading user input, processing it, and showing the results of the processing. The processing part may sometimes take a while to complete, and the user is made to wait during this operation. By placing such long operations in a seperate thread, while having another thread to read user input, the program can be more responsive. It may allow the user to cancel the operation in the middle.

In graphical programs the problem is more severe, since the application should always be ready for a message from the windowing system telling it to repaint part of its window. If it's too busy executing some other task, its window will remain blank, which is rather ugly. In such a case, it is a good idea to have one thread handle the message loop of the windowing systm and always ready to get such repaint requests (as well as user input). When ever this thread sees a need to do an operation that might take a long time to complete (say, more then 0.2 seconds in the worse case), it will delegate the job to a seperate thread.

In order to structure things better, we may use a third thread, to control and synchronize the user-input and task-performing threads. If the user-input thread gets any user input, it will ask the controlling thread to handle the operation. If the task-performing thread finishes its operation, it will ask the controlling thread to show the results to the user.

User Interaction - A Complete Example

As an example, we will write a simple character-mode program that counts the number of lines in a file, while allowing the user to cancel the operation in the middle.

Our main thread will launch one thread to perform the line counting, and a second thread to check for user input. After that, the main thread waits on a condition variable. When any of the threads finishes its operation, it signals this condition variable, in order to let the main thread check what happened. A global variable is used to flag whether or not a cancel request was made by the user. It is initialized to '0', but if the user-input thread receives a cancellation request (the user pressing 'e'), it sets this flag to '1', signals the condition

variable, and terminates. The line-counting thread will signal the condition variable only after it finished its computation.

Before you go read the program, we should explain the use of the `system()` function and the 'stty' Unix command. The `system()` function spawns a shell in which it executes the Unix command given as a parameter. The `stty` Unix command is used to change terminal mode settings. We use it to switch the terminal from its default, line-buffered mode, to a character mode (also known as raw mode), so the call to `getchar()` in the user-input thread will return immediatly after the user presses any key. If we hadn't done so, the system will buffer all input to the program until the user presses the ENTER key. Finally, since this raw mode is not very useful (to say the least) once the program terminates and we get the shell prompt again, the user-input thread registers a cleanup function that restores the normal terminal mode, i.e. line-buffered. For more info, please refer to stty's manual page.

The program's source can be found in the file line-count.c. The name of the file whose lines it reads is hardcoded to 'very_large_data_file'. You should create a file with this name in the program's directory (large enough for the operation to take enough time). Alternatively, you may un-compress the file 'very_large_data_file.Z' found in this directory, using the command:

```
uncompress very_large_data_file.Z
```

note that this will create a 5MB(!) file named 'very_large_data_file', so make sure you have enough free disk-space before performing this operation.

Side-Notes

water-marks algorithm
: An algorithm used mostly when handling buffers or queues: start filling in the queue. If its size exceeds a threshold, known as the high water-mark, stop filling the queue (or start emptying it faster). Keep this state until the size of the queue becomes lower then another threshold, known as the low water-mark. At this point, resume the operation of filling the queue (or return the emptying speed to the original speed).