System Programming

Module 3

EGCI 252
System Programming

Computer Engineering Department
Faculty of Engineering

What is a Thread?

A thread in computer science is short for a thread of execution. Threads are a way for a program to split itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does.

Threads compared with Processes

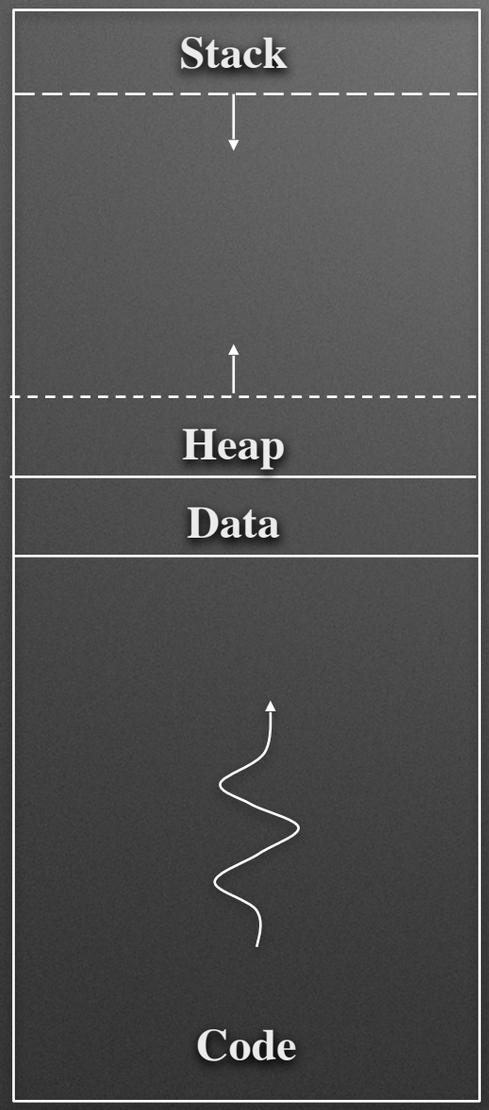
Threads are distinguished from traditional multi-tasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, and share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes; in other operating systems there is not so great a difference.

Thread Overview

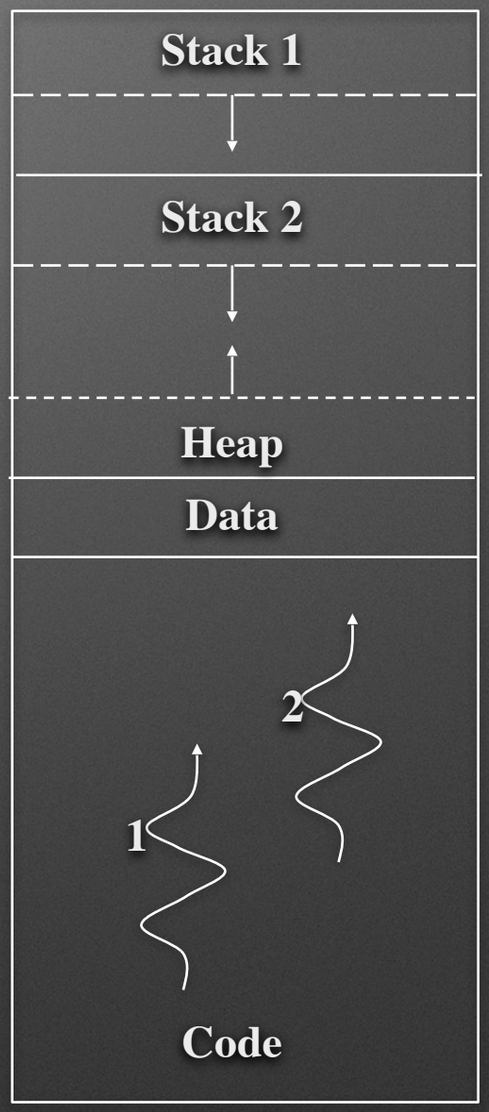
THREADS, LIKE PROCESSES, ARE A MECHANISM TO ALLOW A PROGRAM to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them synchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

We've seen how a program can fork a child process. The child process is initially running its parent's program, with its parent's virtual memory, file descriptors, and so on copied. The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa. When a program creates another thread, though, nothing is copied.

Thread Overview (Cont.)



Single-Threaded Process



Multiple-Threaded Process

Thread Overview (Cont.)

The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor. Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the exec functions, all the other threads are ended (the new program may, of course, create new threads).

GNU/Linux implements the POSIX standard thread API (known as *pthreads*). All thread functions and data types are declared in the header file <pthread.h>. The pthread functions are not included in the standard C library. Instead, they are in libpthread, so you should add -lpthread to the command line when you link your program.

Thread Creation

Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, use the type pthread_t.

Upon creation, each thread executes a thread function. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type void*, and have a void* return type. The parameter is the thread argument: GNU/Linux passes the value along to the thread without looking at it. Your program can use this parameter to pass data to a new thread. Similarly, your program can use the return value to pass data from an exiting thread back to its creator.

Thread Creation (Cont.)

The pthread_create function creates a new thread. You provide it with the following:

1. A pointer to a pthread_t variable, in which the thread ID of the new thread is stored.
2. A pointer to a thread attribute object.This object controls details of how the thread interacts with the rest of the program. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes.
3. A pointer to the thread function.This is an ordinary function pointer, of this type: void* (*) (void*)
4. A thread argument value of type void*. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to pthread_create returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.

Thread Creation - Example

```
//The program creates two threads that print a character
//continuously to standard error. The first thread prints
//Xs and the other prints Ys. After calling pthread_create,
//the main thread prints Zs continuously to standard error.
//gcc -o thread-create thread-create.c -lpthread


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


/* Prints Xs to stderr. */
void *print_Xs(void *unused)
{
   int n;
   for (n = 100; n > 0; n--)
   {
      fputc (‘X', stderr); usleep(1);
   }
   return NULL;
}

/* Prints Ys to stderr. */
void* print_Ys(void* unused)
{
   int n;
   for (n = 100; n > 0; n--)
   {
      fputc (‘Y', stderr); usleep(1);
   }
   return NULL;
}
```

Thread Creation – Example (Cont.)

```
/* The main program. */
int main()
{
   int n;
   pthread_t thread_id;

   /* Create a new thread. The new thread will run the print_xs function. */
   pthread_create (&thread_id, NULL, &print_xs, NULL);
   /* Create a new thread. The new thread will run the print_ys function. */
   pthread_create (&thread_id, NULL, &print_ys, NULL);

   /* Print Zs to stderr. */
   for (n = 100; n > 0; n--)
   {
      fputc(‘Z', stderr); usleep(1);
   }

   return 0;
}
```

Thread Exit

Under normal circumstances, a thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function. The return value from the thread function is taken to be the return value of the thread. Alternately, a thread can exit explicitly by calling pthread_exit. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to pthread_exit is the thread's return value.

Passing Data to Threads

The thread argument provides a convenient method of passing data to threads. Because the type of the argument is void*, though, you can't pass a lot of data directly via the argument. Instead, use the thread argument to pass a pointer to some structure or array of data. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.

Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data. The following program is similar to the previous example. This one creates two new threads, one to print x's and the other to print o's. The same thread function, char_print, is used by both threads, but each is configured differently using struct char_print_parms.

Passing Data to Threads (Cont.)

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


/* Parameters to print_function. */
struct char_print_parms
{
  /* The character to print. */
  char character;
  /* The number of times to print it. */
  int count;
};


/* Prints a number of characters to stderr, as given by PARAMETERS,
which is a pointer to a struct char_print_parms. */
void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type. */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;
  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
```

Passing Data to Threads (Cont.)

```
/* The main program. */
int main ()
{
   pthread_t thread1_id, thread2_id;
   struct char_print_parms thread1_args;
   struct char_print_parms thread2_args;

    /* Create a new thread to print 30,000 x's. */
   thread1_args.character = 'x';
   thread1_args.count = 30000;
   pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

   /* Create a new thread to print 20,000 o's. */
   thread2_args.character = 'o';
   thread2_args.count = 20000;
   pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
   return 0;
}
```

Potential Bug

The program has a serious bug in it. The main thread (which runs the main function) creates the thread parameter structures (thread1_args and thread2_args) as local variables, and then passes pointers to these structures to the threads it creates. What's to prevent Linux from scheduling the three threads in such a way that main finishes executing before either of the other two threads are done? If this happens, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it.

Joining Threads

One solution is to force main to wait until the other two threads are done. What we need is a function similar to wait that waits for a thread to finish instead of a process.

That function is pthread_join, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a void* variable that will receive the finished thread's return value. If you don't care about the thread return value, pass NULL as the second argument.

Revised Program

```
// This program shows the corrected main function for the buggy example
// in the previous program. In this version, main does not exit
// until both of the threads printing x's and o's have completed, so they are
// no longer using the argument  structures.

int main ()
{
   pthread_t thread1_id, thread2_id;
   struct char_print_parms thread1_args, thread2_args;
   /* Create a new thread to print 30,000 x's. */
   thread1_args.character = 'x'; thread1_args.count = 30000;
   pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
   /* Create a new thread to print 20,000 o's. */
   thread2_args.character = 'o'; thread2_args.count = 20000;
   pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

   /* Make sure the first thread has finished. */
   pthread_join (thread1_id, NULL);
   /* Make sure the second thread has finished. */
   pthread_join (thread2_id, NULL);
   /* Now we can safely return. */
   return 0;
}
```

Revised Program (Cont.)

The moral of the story: Make sure that any data you pass to a thread by reference is not deallocated, *even by a different thread*, until you're sure that the thread is done with it. This is true both for local variables, which are deallocated when they go out of scope, and for heap-allocated variables, which you deallocate by calling free (or using delete in C++).

Thread Return Values

If the second argument you pass to pthread_join is non-null, the thread's return value will be placed in the location pointed to by that argument.The thread return value, like the thread argument, is of type void*. If you want to pass back a single int or other small number, you can do this easily by casting the value to void* and then casting back to the appropriate type after calling pthread_join.

The following program computes the n^{th} prime number in a separate thread. That thread returns the desired prime number as its thread return value. The main thread, meanwhile, is free to execute other code.

The n^{th} prime number Computation

```
#include <pthread.h>
#include <stdio.h>
//Compute successive prime numbers (very inefficiently).
//Return the Nth prime number, where N is the value pointed to by *ARG.

void* compute_prime (void* arg)
{
    int candidate = 2, n = *((int*) arg);
    while (1)
    {
      int factor, is_prime = 1;
      /* Test primality by successive division. */
      for (factor = 2; factor < candidate; ++factor)
        if (candidate % factor == 0)
        { is_prime = 0; break; }
      /* Is this the prime number we're looking for? */
      if (is_prime)
      {
        if (--n == 0)
        /* Return the desired prime number as the thread return value. */
          return (void*) candidate;
      }
```

The nth prime number Computation (Cont.)

```
      ++candidate;
    }
    return NULL;
}

int main ()
{
    pthread_t thread;
    int which_prime = 5133, prime;

    /* Start the computing thread, up to the 5,133th prime number. */
    pthread_create (&thread, NULL, &compute_prime, &which_prime);

    /* Do some other work here... */

    /* Wait for the prime number thread to complete, and get the result. */
    pthread_join (thread, (void*) &prime);

    /*Print the 5,133th prime it computed. */
    printf("The %dth prime number is %d.\n", which_prime, prime);
    return 0;
}
```

Multi-thread Assignment

- Modify the previous program to allow 5 threads to concurrently search the prime number between 1 - 50,000
- Each thread only needs to analyse 10,000 numbers
- Print the all prime numbers on the display with the main thread
- Is the run-time of five threaded program faster than a single threaded program?

Thread IDs

Occasionally, it is useful for a sequence of code to determine which thread is executing it. The pthread_self function returns the thread ID of the thread in which it is called. This thread ID may be compared with another thread ID using the pthread_equal function.

These functions can be useful for determining whether a particular thread ID corresponds to the current thread. For instance, it is an error for a thread to call pthread_join to join itself. (In this case, pthread_join would return the error code EDEADLK.) To check for this beforehand, you might use code like this:

```
if (!pthread_equal (pthread_self (), other_thread))
    pthread_join (other_thread, NULL);
```

Thread Attributes

Thread attributes provide a mechanism for fine-tuning the behavior of individual threads. Recall that pthread_create accepts an argument That is a pointer to a thread attribute object. If you pass a null pointer, the default thread attributes are used to configure the new thread. However, you may create and customize a thread attribute object to specify other values for the attributes.

To specify customized thread attributes, you must follow these steps:

1. Create a pthread_attr_t object. The easiest way is simply to declare an automatic variable of this type.
2. Call pthread_attr_init, passing a pointer to this object. This initializes the attributes to their default values.
3. Modify the attribute object to contain the desired attribute values.
4. Pass a pointer to the attribute object when calling pthread_create.
5. Call pthread_attr_destroy to release the attribute object. The pthread_attr_t variable itself is not deallocated; it may be reinitialized with pthread_attr_init.

Thread Attributes (Cont.)

A single thread attribute object may be used to start several threads. It is not necessary to keep the thread attribute object around after the threads have been created. For most GNU/Linux application programming tasks, only one thread attribute is typically of interest (the other available attributes are primarily for specialty real-time programming). This attribute is the thread's *detach state*. A thread may be created as a *joinable thread* (the default) or as a *detached thread*. A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates. Instead, the thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls pthread_join to obtain its return value. Only then are its resources released. A detached thread, in contrast, is cleaned up automatically when it terminates. Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using pthread_join to obtain its return value.

Detach State

To set the detach state in a thread attribute object, use pthread_attr_setdetachstate. The first argument is a pointer to the thread attribute object, and the second is the desired detach state. Because the joinable state is the default, it is necessary to call this only to create detached threads; pass PTHREAD_CREATE_DETACHED as the second argument.

```
#include <pthread.h>
void* thread_function (void* thread_arg)
{
  /* Do work here... */
}

int main ()
{
  pthread_attr_t attr;
  pthread_t thread;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_create(&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy(&attr);

  /* Do work here... */

  /* No need to join the second thread.  */
  /* However the main thread needs to make sure that it will finish after the other threads */
  return 0;
}
```

Thread Cancellation

Under normal circumstances, a thread terminates when it exits normally, either by returning from its thread function or by calling pthread_exit. However, it is possible for a thread to request that another thread terminate. This is called *canceling* a thread. To cancel a thread, call pthread_cancel, passing the thread ID of the thread to be canceled. A canceled thread may later be joined; in fact, you should join a canceled thread to free up its resources, unless the thread is detached. The return value of a canceled thread is the special value given by PTHREAD_CANCELED.

Thread Cancellation (Cont.)

Often a thread may be in some code that must be executed in an all-or-nothing fashion. For instance, the thread may allocate some resources, use them, and then deallocate them. If the thread is canceled in the middle of this code, it may not have the opportunity to deallocate the resources, and thus the resources will be leaked. To counter this possibility, it is possible for a thread to control whether and when it can be canceled.

A thread may be in one of three states with regard to thread cancellation.

- The thread may be *asynchronously cancelable*. The thread may be canceled at any point in its execution.
- The thread may be *synchronously cancelable*. The thread may be canceled, but not at just any point in its execution. Instead, cancellation requests are queued, and the thread is canceled only when it reaches specific points in its execution.
- A thread may be *uncancelable*. Attempts to cancel the thread are quietly ignored. When initially created, a thread is synchronously cancelable.

Synchronous and Asynchronous Threads

An asynchronously cancelable thread may be canceled at any point in its execution. A synchronously cancelable thread, in contrast, may be canceled only at particular places in its execution.These places are called *cancellation points*. The thread will queue a cancellation request until it reaches the next cancellation point.

To make a thread asynchronously cancelable, use pthread_setcanceltype. This affects the thread that actually calls the function. The first argument should be PTHREAD_CANCEL_ASYNCHRONOUS to make the thread asynchronously cancelable, or PTHREAD_CANCEL_DEFERRED to return it to the synchronously cancelable state. The second argument, if not null, is a pointer to a variable that will receive the previous cancellation type for the thread. This call, for example, makes the calling thread asynchronously cancelable.

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Synchronous and Asynchronous Threads (Cont.)

What constitutes a cancellation point, and where should these be placed? The most direct way to create a cancellation point is to call pthread_testcancel. This does nothing except process a pending cancellation in a synchronously cancelable thread. You should call pthread_testcancel periodically during lengthy computations in a thread function, at points where the thread can be canceled without leaking any resources or producing other ill effects.

Certain other functions are implicitly cancellation points as well. These are listed on the pthread_cancel man page. Note that other functions may use these functions internally and thus will indirectly be cancellation points.

Uncancelable Critical Sections

A thread may disable cancellation of itself altogether with the pthread_setcancelstate function. Like pthread_setcanceltype, this affects the calling thread. The first argument is PTHREAD_CANCEL_DISABLE to disable cancellation, or PTHREAD_CANCEL_ENABLE to re-enable cancellation. The second argument, if not null, points to a variable that will receive the previous cancellation state. This call, for instance, disables thread cancellation in the calling thread.

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

Using pthread_setcancelstate enables you to implement *critical sections*. A critical section is a sequence of code that must be executed either in its entirety or not at all; in other words, if a thread begins executing the critical section, it must continue until the end of the critical section without being canceled.

Uncancelable Critical Sections (Cont.)

For example, suppose that you're writing a routine for a banking program that transfers money from one account to another. To do this, you must add value to the balance in one account and deduct the same value from the balance of another account. If the thread running your routine happened to be canceled at just the wrong time between these two operations, the program would have spuriously increased the bank's total deposits by failing to complete the transaction. To prevent this possibility, place the two operations in a critical section.

You might implement the transfer with a function such as process_transaction, shown in the following program. This function disables thread cancellation to start a critical section before it modifies either account balance.

Protect a Bank Transaction with a Critical Section

```
// critical-section.c
#include <pthread.h>
#include <stdio.h>
#include <string.h>

/* An array of balances in accounts, indexed by account number. */
float* account_balances;
/* Transfer DOLLARS from account FROM_ACCT to account
TO_ACCT. Return 0 if the transaction succeeded, or 1 if the balance
FROM_ACCT is too small. */

int process_transaction (int from_acct, int to_acct, float dollars)
{
   int old_cancel_state;
   /* Check the balance in FROM_ACCT. */
   if (account_balances[from_acct] < dollars)
     return 1;
```

Protect a Bank Transaction with a Critical Section

```
/* Begin critical section. */
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
/* Move the money. */
account_balances[to_acct] += dollars;
account_balances[from_acct] -= dollars;
/* End critical section. */
  pthread_setcancelstate (old_cancel_state, NULL);
  return 0;
}
```

Note that it’s important to restore the old cancel state at the end of the critical section rather than setting it unconditionally to PTHREAD_CANCEL_ENABLE. This enables you to call the process_transaction function safely from within another critical section—in that case, your function will leave the cancel state the same way it found it.

Thread-Specific Data

Unlike processes, all threads in a single program share the same address space. This means that if one thread modifies a location in memory (for instance, a global variable), the change is visible to all other threads. This allows multiple threads to operate on the same data without the use interprocess communication mechanisms.

Each thread has its own call stack, however. This allows each thread to execute different code and to call and return from subroutines in the usual way. As in a single-threaded program, each invocation of a subroutine in each thread has its own set of local variables, which are stored on the stack for that thread.

Sometimes, however, it is desirable to duplicate a certain variable so that each thread has a separate copy. GNU/Linux supports this by providing each thread with a *thread-specific data* area. The variables stored in this area are duplicated for each thread, and each thread may modify its copy of a variable without affecting other threads. Because all threads share the same memory space, thread-specific data may not be accessed using normal variable references. GNU/Linux provides special functions for setting and retrieving values from the thread-specific data area.

Thread-Specific Data (Cont.)

You may create as many thread-specific data items as you want, each of type void*. Each item is referenced by a key. To create a new key, and thus a new data item for each thread, use pthread_key_create. The first argument is a pointer to a pthread_key_t variable. That key value can be used by each thread to access its own copy of the corresponding data item. The second argument to pthread_key_t is a cleanup function. If you pass a function pointer here, GNU/Linux automatically calls that function when each thread exits, passing the thread-specific value corresponding to that key. This is particularly handy because the cleanup function is called even if the thread is canceled at some arbitrary point in its execution. If the thread-specific value is null, the thread cleanup function is not called. If you don't need a cleanup function, you may pass null instead of a function pointer.

After you've created a key, each thread can set its thread-specific value corresponding to that key by calling pthread_setspecific. The first argument is the key, and the second is the void* thread-specific value to store. To retrieve a thread-specific data item, call pthread_getspecific, passing the key as its argument.

Thread-Specific Data (Cont.)

Suppose, for instance, that your application divides a task among multiple threads. For audit purposes, each thread is to have a separate log file, in which progress messages for that thread's tasks are recorded. The thread-specific data area is a convenient place to store the file pointer for the log file for each individual thread.

The following program shows how you might implement this. The main function in this sample program creates a key to store the thread-specific file pointer and then stores it in thread_log_key. Because this is a global variable, it is shared by all threads. When each thread starts executing its thread function, it opens a log file and stores the file pointer under that key. Later, any of these threads may call write_to_thread_log to write a message to the thread-specific log file. That function retrieves the file pointer for the thread's log file from thread-specific data and writes the message.

Per-Thread Log Files Implemented with Thread-Specific Data

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

/* The key used to associate a log file pointer with each thread. */
static pthread_key_t thread_log_key;

/* Write MESSAGE to the log file for the current thread. */
void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, "%s\n", message);
}

/* Close the log file pointer THREAD_LOG. */
void close_thread_log (void* thread_log)
```

Per-Thread Log Files Implemented with Thread-Specific Data (Cont.)

```
{
  fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;

  /* Generate the filename for this thread's log file. */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());

  /* Open the log file. */
  thread_log = fopen (thread_log_filename, "w");

  /* Store the file pointer in thread-specific data under thread_log_key. */
  pthread_setspecific (thread_log_key, thread_log);
  write_to_thread_log ("Thread starting.");

  /* Do work here... */
  return NULL;
}
```

Per-Thread Log Files Implemented with Thread-Specific Data (Cont.)

```
int main ()
{
   int i;
   pthread_t threads[5];

   /* Create a key to associate thread log file pointers in
   thread-specific data. Use close_thread_log to clean up
   the file pointers. */
   pthread_key_create (&thread_log_key, close_thread_log);

   /* Create threads to do the work. */
   for (i = 0; i < 5; ++i)
      pthread_create (&(threads[i]), NULL, thread_function, NULL);

   /* Wait for all threads to finish. */
   for (i = 0; i < 5; ++i)
      pthread_join (threads[i], NULL);
   return 0;
}
```

Per-Thread Log Files Implemented with Thread-Specific Data (Cont.)

Normal Global Variable Usage:
A single global variable is associated to only one value
log_file ———> threadxxxx.log

Thread-Specific Data Implementation:
A single global variable is associated to multiple thread specific log files

```
thread_log_key  ———> threadxxx1.log for thread ID xxx1
                |———> threadxxx2.log for thread ID xxx2
                |———> threadxxx3.log for thread ID xxx3
                |———> threadxxx4.log for thread ID xxx4
                |———> threadxxx5.log for thread ID xxx5
```

Per-Thread Log Files Implemented with Thread-Specific Data (Cont.)

Observe that thread_function does not need to close the log file. That's because when the log file key was created, close_thread_log was specified as the cleanup function for that key. Whenever a thread exits, GNU/Linux calls that function, passing the thread-specific value for the thread log key. This function takes care of closing the log file.

Cleanup Handlers

The cleanup functions for thread-specific data keys can be very handy for ensuring that resources are not leaked when a thread exits or is canceled. Sometimes, though, it's useful to be able to specify cleanup functions without creating a new thread-specific data item that's duplicated for each thread. GNU/Linux provides *cleanup handlers* for this purpose.

A cleanup handler is simply a function that should be called when a thread exits. The handler takes a single void* parameter, and its argument value is provided when the handler is registered—this makes it easy to use the same handler function to deallocate multiple resource instances.

A cleanup handler is a temporary measure, used to deallocate a resource only if the thread exits or is canceled instead of finishing execution of a particular region of code. Under normal circumstances, when the thread does not exit and is not canceled, the resource should be deallocated explicitly and the cleanup handler should be removed.

Cleanup Handlers (Cont.)

To register a cleanup handler, call pthread_cleanup_push, passing a pointer to the cleanup function and the value of its void* argument. The call to pthread_cleanup_push must be balanced by a corresponding call to pthread_cleanup_pop, which unregisters the cleanup handler. As a convenience, pthread_cleanup_pop takes an int flag argument; if the flag is nonzero, the cleanup action is actually performed as it is unregistered.

The program fragment in the next slide shows how you might use a cleanup handler to make sure that a dynamically allocated buffer is cleaned up if the thread terminates.

Program Fragment Demonstrating a Thread Cleanup Handler

```
// Cleanup.c
#include <malloc.h>
#include <pthread.h>

/* Allocate a temporary buffer. */
void* allocate_buffer (size_t size)
{
   return malloc (size);
}

/* Deallocate a temporary buffer. */
void deallocate_buffer (void* buffer)
{
   free (buffer);
}

void do_some_work ()
{
   /* Allocate a temporary buffer. */
```

Program Fragment Demonstrating a Thread Cleanup Handler

```
void* temp_buffer = allocate_buffer (1024);
/* Register a cleanup handler for this buffer, to deallocate
it in case the thread exits or is cancelled. */
pthread_cleanup_push (deallocate_buffer, temp_buffer);

/* Do some work here that might call pthread_exit or
might be cancelled... */
   /* Unregister the cleanup handler. Because we pass a
   nonzero value, this actually performs the cleanup by
   calling deallocate_buffer. */
   pthread_cleanup_pop (1);
}
```

Because the argument to pthread_cleanup_pop is nonzero in this case, the cleanup function deallocate_buffer is called automatically here and does not need to be called explicitly. In this simple case, we could have used the standard library function free directly as our cleanup handler function instead of deallocate_buffer.

Processes Vs. Threads

For some programs that benefit from concurrency, the decision whether to use processes or threads can be difficult. Here are some guidelines to help you decide which concurrency model best suits your program:

- All threads in a program must run the same executable.A child process, on the other hand, may run a different executable by calling an exec function.
- An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread. An errant process, on the other hand, cannot do so because each process has a copy of the program’s memory space.

Processes Vs. Threads (Cont.)

- Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.
- Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.
- Sharing data among threads is trivial because threads share the same memory. (However, great care must be taken to avoid race conditions.) Sharing data among processes requires the use of IPC mechanisms. This can be more cumbersome but makes multiple processes less likely to suffer from concurrency bugs.

End of Module