1 Threads

- Threads are a mechanism to allow a program to do more than one thing at a time.
- A thread exists within a process. Threads are a finer-grained unit of execution than processes.
- 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.
- The child process can modify **its memory**, close file descriptors etc., without affecting its parent, and vice versa.
- Threads offer lower consumption of system resources and easier communication between processes. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original.
- There are many potential pitfalls to using threads or any other environment where the same memory space is shared by multiple processes.
  - You must be careful about more than one process using the same variables at the same time.
  - Many functions are not *reentrant*; that is, there cannot be more than one copy of that function running at the same time (unless they are using separate data segments).
  - Static variables declared inside functions are often a problem.
  - Returning a pointer to statically allocated storage inside the function is no good; another thread may execute that function and overwrite the return value before the first one is through using it.
  - Setting or using global variables may create problems in threads.
- Semaphores, mutexes, disabling interrupts, or similar means should be used to protect variables, particularly aggregate variables, against simultaneous access.
- POSIX threads (pthreads) provide a relatively portable implementation of lightweight processes.
- All thread functions and data types are declared in the header file $< pthread.h >$

- They are in **libpthread**, so you should add **-lpthread** to the command line when you link your program.

1.1 Thread Creation

- Each thread in a process is identified by a *thread ID*, 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.
- The **pthread_create** function creates a new thread. You provide it with the following:
  - A pointer to a **pthread_t** variable, in which the thread ID of the new thread is stored.
  - 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.
  - A pointer to the thread function. This is an ordinary function pointer, of this type:

    ```
    void* (*) (void*)
    ```

  - 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. thread-create.c (see Fig. 1)
- Under normal circumstances, a thread exits in one of two ways;
  - by returning from the thread function. The return value from the thread function is taken to be the return value of the thread.
  - a thread can exit explicitly by calling **pthread_exit**.

```
#include <pthread.h>
#include <stdio.h>
/* Prints x's to stderr.  The parameter is unused.  Does not return.  */
void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}
/* The main program.  */
int main ()
{
  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);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

Figure 1: Create a Thread

1.1.1 Passing Data to Threads

- Use the thread argument to pass a pointer to some structure or array of data.
- Define a structure for each thread function, which contains the parameters that the thread function expects.
- Using the thread argument, it is easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data.
- In the following program (see Fig. 2), the same thread function, **char_print**, is used by both threads, but each is configured differently using *struct* **char_print_parms**. thread-create2.c
- A serious bug in it. The main thread creates the thread parameter structures as local variables, and then passes pointers.

```
#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;
}
/* The main program.  */
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
  /* Create a new thread to print 30000 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 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
  return 0;
}
```

Figure 2: Create Two Threads

- if **main** finishes executing before either of the other two threads are done, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it.

1.1.2 Joining Threads

- The solution is to force main to wait until the other two threads are done.
- That function is **pthread_join**, which takes two arguments;
  - the thread ID of the thread to wait for.
  - 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.
- In the following code (see Fig. 3), main does not exit until threads are no longer using the argument structures.
- Make sure that any data you pass to a thread by reference is not deallocated;
  - for local variables, which are deallocated when they go out of scope.
  - for heap-allocated variables, which you deallocate by calling free.

1.1.3 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***.
- The following program (see Fig. 4) computes the n^{th} prime number in a separate thread. That thread returns the desired prime number as its thread return value. primes.c
- 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.

```
int main ()
{
pthread_t thread1_id;
pthread_t 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);
/* 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;
}
```

Figure 3: Revised Main Function for thread-create2.c

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

1.1.4 Thread Attributes

- Thread attributes provide a mechanism for fine-tuning the behavior of individual threads.
- 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;
  - Create a **pthread_attr_t** object. The easiest way is simply to declare an automatic variable of this type.
  - Call **pthread_attr_init**, passing a pointer to this object. This initializes the attributes to their default values.
  - Modify the attribute object to contain the desired attribute values.

```
#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;
  int n = *((int*) arg);
  while (1) {
    int factor;
    int 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;
    }
    ++candidate;
  }
  return NULL;
}
int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;
/* Start the computing thread, up to the 5000th 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 largest prime it computed.  */
  printf("The %dth prime number is %d.\n", which_prime, prime);
  return 0;
}
```

Figure 4: Compute Prime Numbers in a Thread

  - Pass a pointer to the attribute object when calling **pthread_create**.
  - 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**.

- 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 application programming tasks, only one thread attribute is typically of interest. 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 when it terminates. (until another thread calls **pthread_join** to obtain its return value. Only then are its resources released.)
  - A **detached thread** 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** or obtain its return value.

- The following code (see Fig. 5) creates a detached thread by setting the detach state thread attribute for the thread. detached.c

- Even if a thread is created in a joinable state, it may later be turned into a detached thread. To do this, call **pthread_detach**. Once a thread is detached, it cannot be made joinable again.

- Table 1 shows the thread attributes; the default values are marked with an asterisk.

1.2 Thread Cancellation

- 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.

```
#include <pthread.h>
void* thread_function (void* thread_arg)
{
  /* Do work here...  */
  return NULL;
}
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.  */
  return 0;
}
```

Figure 5: Skeleton Program That Creates a Detached Thread

- Often a thread may be in some code that must be executed in an all-or-nothing fashion.
- 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.
- A thread may be in one of three states with regard to thread cancellation.
  - **asynchronously cancelable**. The thread may be canceled at any point in its execution.
  - **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. These points are called *cancellation points*.
  - **uncancelable**. Attempts to cancel the thread are quietly ignored.
- When initially created, a thread is synchronously cancelable.

Table 1: Thread Attributes

Attribute	Value	Meaning
detachstate	PTHREAD_CREATE_JOINABLE*	Joinable state
	PTHREAD_CREATE_DETACHED	Detached state
schedpolicy	SCHED_OTHER*	Normal, non-realtime
	SCHED_RR	Realtime, round-robin
	SCHED_FIFO	Realtime, first in first out
schedparam	policy specific	
inheritsched	PTHREAD_EXPLICIT_SCHED*	Set by *schedpolicy* and *schedparam*
	PTHREAD_INHERIT_SCHED	Inherited from parent process
scope	PTHREAD_SCOPE_SYSTEM*	One system timeslice for each thread
	PTHREAD_SCOPE_PROCESS	Threads share same system timeslice (not supported under Linux)

1.2.1 Synchronous and Asynchronous Threads

- To make a thread asynchronously cancelable, use **pthread_setcanceltype**.
- 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.

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

- To create a cancellation point is to call **pthread_testcancel**. 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.

1.2.2 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 reenable cancellation.
  - The second argument, if not null, points to a variable that will receive the previous cancellation state.

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

- Using **pthread_setcancelstate** enables you to implement **critical sections**.
- If a thread begins executing the critical section, it must continue until the end of the critical section without being canceled. critical-section.c (see Fig. 6)
- 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**.

1.3 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, 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. 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 (thread-specific data area). The variables stored in this area are duplicated.

```
#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;
  /* 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;
}
```

Figure 6: Protect a Bank Transaction with a Critical Section

- Special functions for setting and retrieving values from the thread-specific data area.
- 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.
  - The second argument to **pthread_key_t** is a cleanup function.
- After you've created a key, each thread can set its thread-specific value corresponding to that key by calling **pthread_setspecific**.
- To retrieve a thread-specific data item, call **pthread_getspecific**, passing the key as its argument. tsd.c (see Fig. 7)

```
#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)
{
  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;
}
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;
}
```


Figure 7: Per-Thread Log Files Implemented with Thread-Specific Data

- Observe that **thread_function** does not need to close the log file. **close_thread_log** was specified as the cleanup function for that key. Whenever a thread exits, that function is called to pass the thread-specific value for the thread log key.

1.4 Synchronization and Critical Sections

- There's no way to know when the system will schedule one thread to run and when it will run another.
- If one thread is only partway through updating a data structure when another thread accesses the same data structure, chaos.
- These *bugs* are called **race conditions**; the threads are racing one another to change the same data structure.

1.4.1 Race Conditions

- Suppose that your program has a series of queued jobs that are processed by several concurrent threads. The queue of jobs is represented by a linked list of struct job objects. job-queue1.c (see Fig. 8)
  - Now suppose that two threads happen to finish a job at about the same time, but only one job remains in the queue. By unfortunate coincidence, we may have two threads executing the same job.
  - To make matters worse, one thread will unlink the job object from the queue, leaving **job_queue** containing null. When the other thread evaluates **job_queue->next**, a segmentation fault will result.
- This is an example of a race condition. Under lucky circumstances, this particular schedule of the two threads may never occur, and the race condition may never exhibit itself. Only under different circumstances, may the bug exhibit itself.
- To eliminate race conditions, you need a way to make operations **atomic**.
- In this particular example, you want to check **job_queue**; if it s not empty, remove the first job, all as a single atomic operation.

```
#include <malloc.h>
struct job {
  /* Link field for linked list.  */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs.  */
struct job* job_queue;
extern void process_job (struct job*);
/* Process queued jobs until the queue is empty.  */
void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* Get the next available job.  */
    struct job* next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

Figure 8: Thread Function to Process Jobs from the Queue

1.4.2 Mutexes

- The solution to the job queue race condition problem is to let only one thread access the queue of jobs at a time.
- Implementing this requires support from the operating system, **mutexes**, short for MUTual EXclusion locks.
- A mutex is a special lock that only one thread may lock at a time. If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is **blocked**, or put on hold.
- To create a mutex, create a variable of type **pthread_mutex_t** and pass a pointer to it to **pthread_mutex_init**.
- The second argument to **pthread_mutex_init** is a pointer to a mutex attribute object, which specifies attributes of the mutex.

- The mutex variable should be initialized only once.

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

- Another simpler way to create a mutex with default attributes is to initialize it with the special value **PTHREAD_MUTEX_INITIALIZER**. No additional call to **pthread_mutex_init** is necessary.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- A thread may attempt to lock a mutex by calling **pthread_mutex_lock** on it. If the mutex was unlocked, it becomes locked and the function returns immediately.

- More than one thread may be blocked on a locked mutex at one time. When the mutex is unlocked, only one of the blocked threads (chosen unpredictably) is unblocked and allowed to lock the mutex; the other threads stay blocked.

- A call to **pthread_mutex_unlock** unlocks a mutex. This function should always be called from the same thread that locked the mutex.

- The following code (see Fig. 9) shows another version of the job queue example. Now the queue is protected by a mutex. job-queue2.c

- Note that if the queue is empty (that is, **job_queue** is null), we don't break out of the loop immediately because this would leave the mutex permanently locked and would prevent any other thread from accessing the job queue ever again.

1.4.3 Mutex Deadlocks

- Mutexes provide a mechanism for allowing one thread to block the execution of another.This opens up the possibility of a new class of bugs, called *deadlocks*.

- A deadlock occurs when one or more threads are stuck waiting for something that never will occur.

- A simple type of deadlock may occur when the same thread attempts to lock a mutex twice. The behavior in this case depends on what kind of mutex is being used. Three kinds of mutexes exist;

```
#include <malloc.h>
#include <pthread.h>
struct job {
  /* Link field for linked list.  */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs.  */
struct job* job_queue;
extern void process_job (struct job*);
/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
/* Process queued jobs until the queue is empty.  */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Now it's safe to check if the queue is empty.  */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Get the next available job.  */
      next_job = job_queue;
      /* Remove this job from the list.  */
      job_queue = job_queue->next;
    }
    /* Unlock the mutex on the job queue, since we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);
    /* Was the queue empty?  If so, end the thread.  */
    if (next_job == NULL)
      break;
    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

Figure 9: Job Queue Thread Function, Protected by a Mutex

  - Locking a *fast mutex* (the default kind) will cause a deadlock to occur.
  - Locking a *recursive mutex* does not cause a deadlock. The mutex remembers how many times **pthread_mutex_lock** was called on it by the thread that holds the lock; that thread must make the same number of calls to **pthread_mutex_unlock** before the mutex is actually unlocked and another thread is allowed to lock it.
  - GNU/Linux will detect and flag a double lock on an *error-checking mutex*.

- By default, a GNU/Linux mutex is of the fast kind.To create a mutex of one of the other two kinds, first create a mutex attribute object by declaring a **pthread_mutexattr_t** variable and calling **pthread_mutexattr_init** on a pointer to it. Then set the mutex kind by calling **pthread_mutexattr_setkind_np**;

  - the first argument is a pointer to the mutex attribute object.
  - the second is **PTHREAD_MUTEX_RECURSIVE_NP** for a recursive mutex, or **PTHREAD_MUTEX_ERRORCHECK_NP** for an error-checking mutex.

- Pass a pointer to this attribute object to **pthread_mutex_init** to create a mutex of this kind, and then destroy the attribute object with **pthread_mutexattr_destroy**.

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```