Linux Programming

Pthread

www.rt-ics.com

v13958190992@163.com

Topics

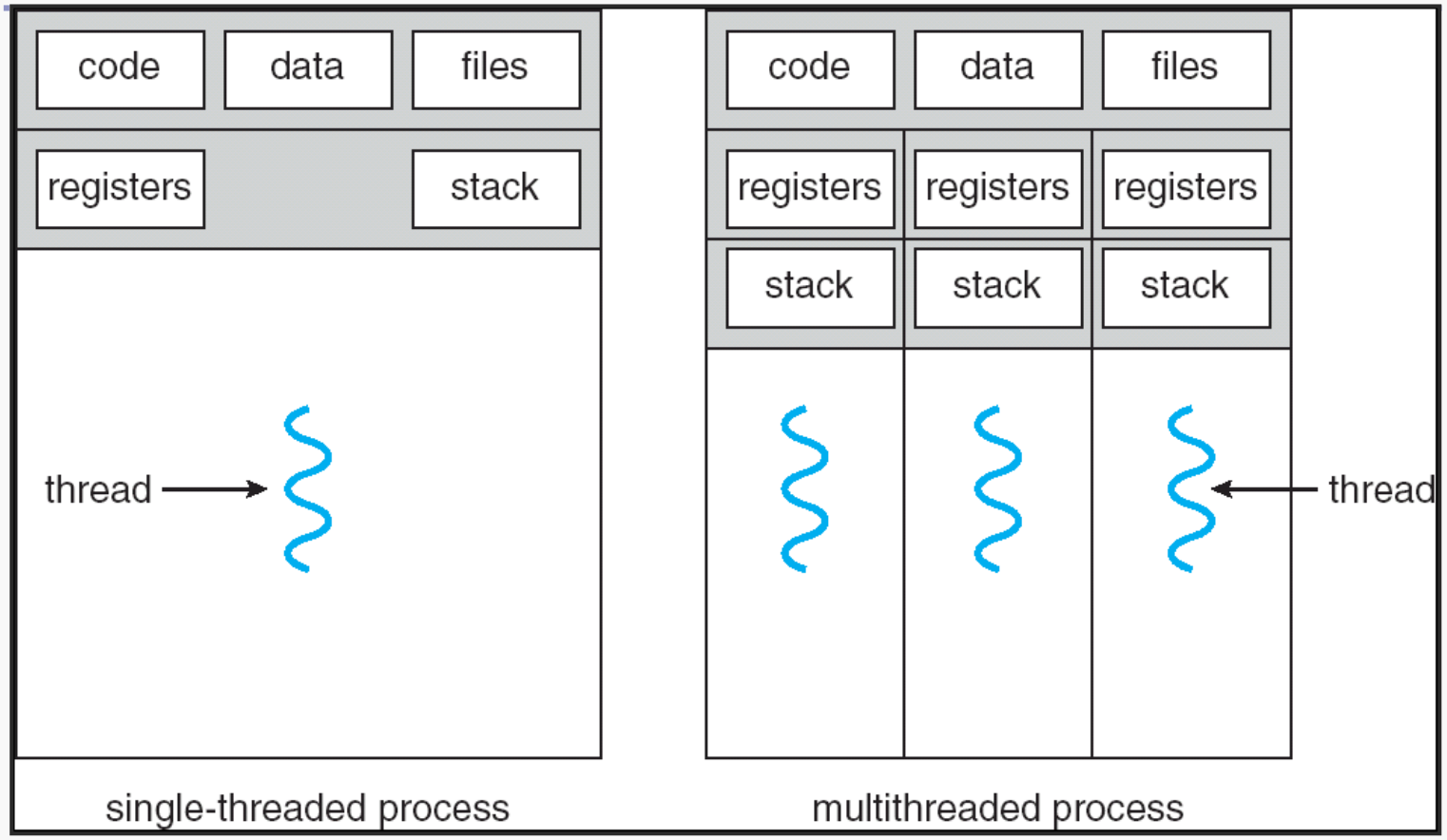
- Threads
- Pthread
- Pthread Mutex
- Pthread Condition variables
- Pthread Main API

Linux Programming
Threads

Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Single vs. Multithreaded Processes



Process: everything we've seen up to now
Thread: like a process, only shares memory, global variables, files, PID with other threads within the same process

use threads or processes?

- Why use threads instead of processes?
  - Faster context switch
  - Easier to share data
  - Uses less memory and resources
  - Thread creation faster than process creation
  - Less things to set-up
- Why use processes instead of threads?
  - Different code
  - Running on different machines
  - Different owners
  - Little communication
  - Sharing memory can lead to obscure bugs

User-level threads

- Threads can be provided at the **user** or **kernel level**
- User level: kernel knows nothing about threads
- Implemented in a library by somebody without touching the kernel
- User library handles
  - Thread creation
  - Thread deletion
  - Thread scheduling
- Benefits:
  - Faster creation and scheduling
- Drawbacks:
  - One thread blocking during I/O blocks **all** threads in process (even ready-to-run threads)

User-level threads (cont'd)

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

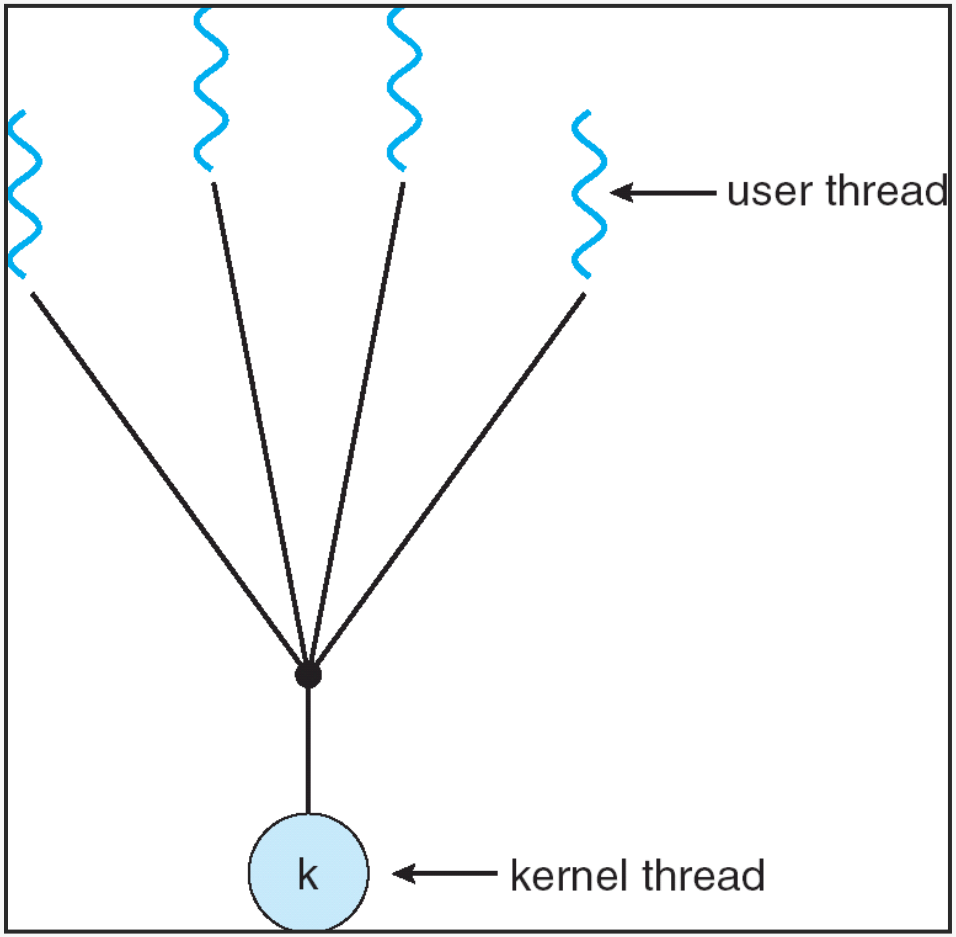
Kernel-level threads

- Kernel knows about threads
- Kernel handles thread creation, deletion, scheduling
- Benefits:
  - Kernel can schedule another thread if current one does blocking I/O
  - Kernel can schedule multiple threads on different CPUs on SMP multiprocessor
- Drawbacks:
  - Slower to schedule, create, delete than user-level
- Most modern OSes support kernel-level threads
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- How to map kernel-level threads to user-level threads?
  - Many-to-One
  - One-to-One
  - Many-to-Many

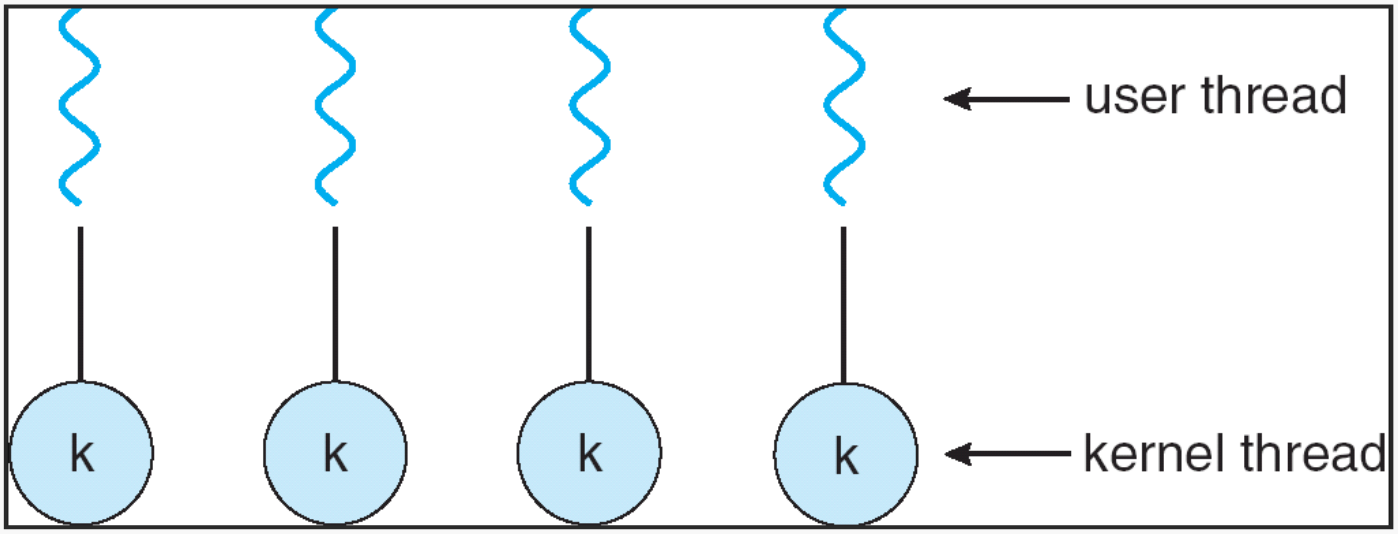
Many-to-One Model



Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- Any disadvantages?
  - All block when one blocks
  - All run on 1 CPU

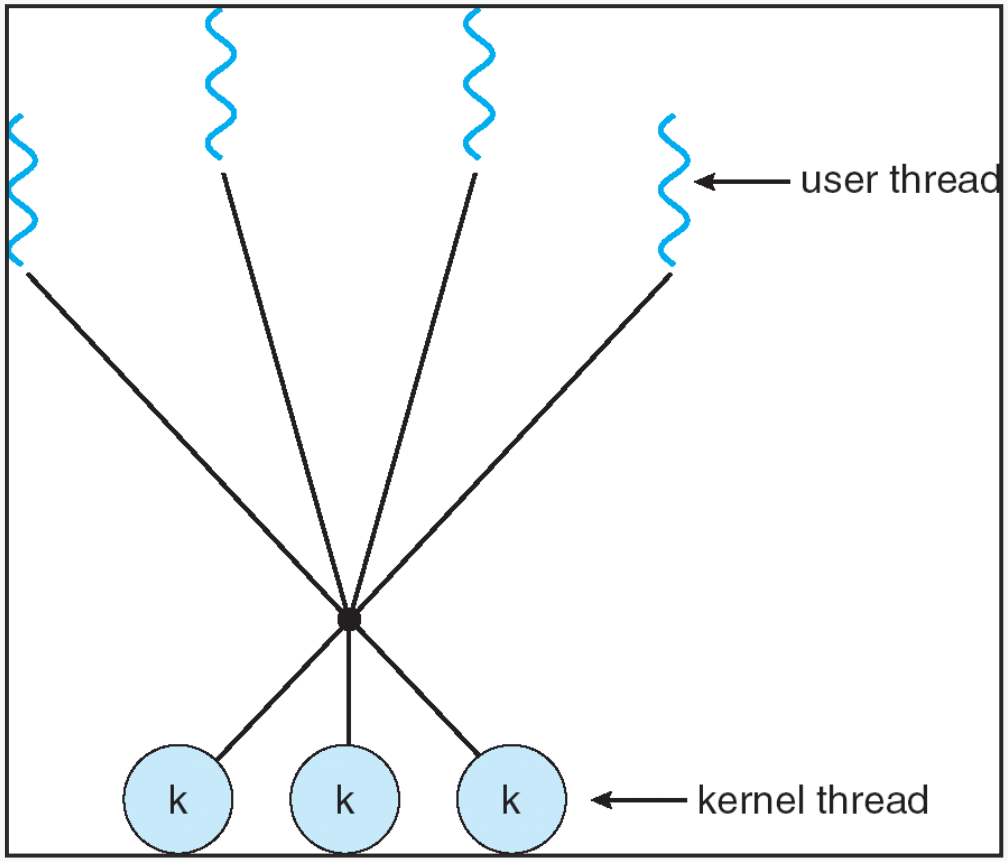
One-to-one Model



One-to-One

- Each user-level thread maps to a kernel thread
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
- Any disadvantages?
  - Overhead for creating threads
  - Many operating systems limit number of threads

Many-to-Many Model

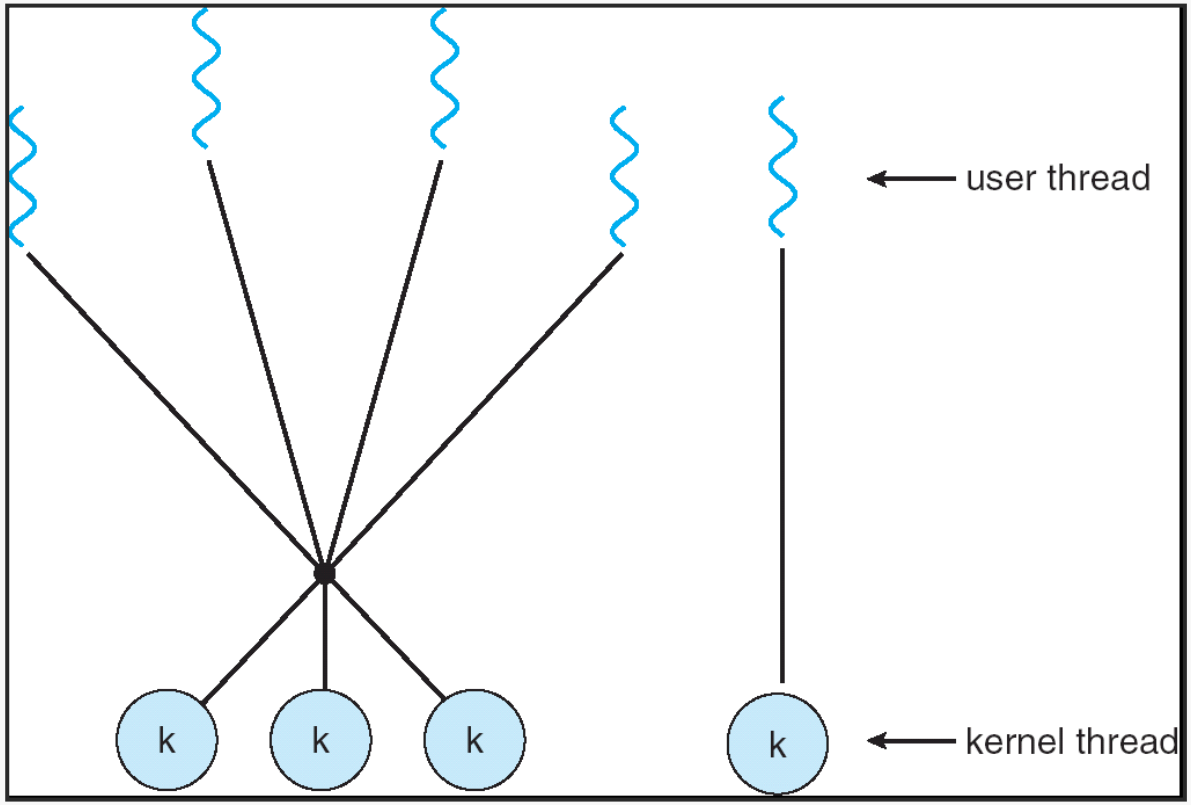


Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the flexibility of choosing the number of kernel threads allocated to a process
- “Best of both worlds”

- Solaris prior to version 9

Two-level Model



Two-level Model

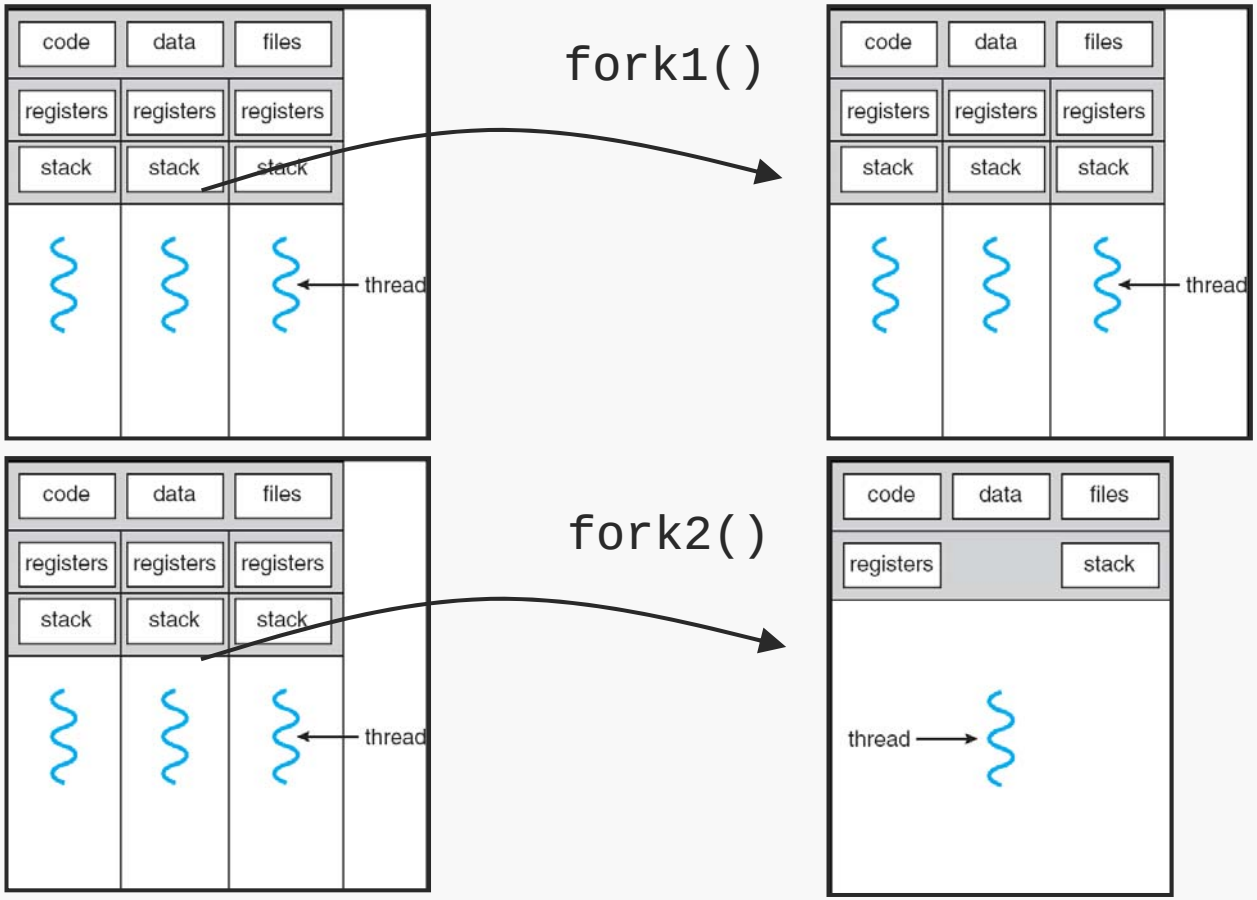
- Similar to many-to-many, but allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread-specific data

Semantics of `fork()` and `exec()`

- Does `fork()` duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of `fork()`
- `exec()` usually replaces all threads with new program



Thread Cancellation

- Terminating a thread before it has finished
- E.g., two cooperating threads, one discovers an error
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Why is this an issue?
  - What if a thread is in the middle of
    - Allocating resources
    - Performing I/O
    - Updating a shared data structure

Thread Cancellation (cont'd)

- Essentially, deferred cancellation = "target, please cancel yourself"
  - Occurs when target checks for cancellation signal
- Allows cancellation at "safe" points
  - Called **cancellation points** in Pthreads

Signal Handling

- Signals are used in UNIX to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by a particular event
  2. Signal is delivered to a process
  3. Signal is handled (or ignored/blocked)
- Options:
  - Deliver the signal to the thread to which the signal applies
    - Applicable with synchronous signals e.g., illegal memory access
  - Deliver the signal to **every** thread in the process
  - Deliver the signal to **certain** threads in the process
  - Assign a specific threat to receive **all signals** for the process

Thread Pooling

- When design situations arise that could benefit by using many short-lived threads, thread pooling is a useful technique.
- Rather than create a brand new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task.
- When the thread is finished with task, it adds itself back to the pool and waits for another assignment

Benefits of Thread Pooling

- You can reduce response time because a thread is already constructed and started and is simply waiting for its next task. In the case of an HTTP server, an available thread in the pool can deliver each new file requested
- Threads are fixed in size at the time of construction. All the threads are started, and then each goes into a wait state until a task is assigned to it. If all the threads are currently assigned a task, new service requests will put into a wait state until one of the threads finishes its task and returns itself to the pool.

Thread Pools

- Motivating example: a web server running on an SMP machine
- To handle each connection:
  1. Create a new process to handle it
     - too slow, inefficient
  2. Create a new thread to handle it
- Option 2 better but still has some problems:
  - Some overhead for thread creation/deletion
  - Thread will only be used for this connection
  - Unbounded number of threads might crash web server
- Better solution: use a thread pool of (usually) fixed size

Thread Pools (cont'd)

- Threads in pool sit idle
- Request comes in:
  - Wake up a thread in pool
  - Assign it the request
  - When it completes, return it to pool
  - If no threads in pool available, wait
- Advantages:
  - Usually slightly faster to wake up an existing thread than create a new one
  - Allows the number of threads in the application to be limited by the size of the pool
  - More sophisticated systems dynamically adjust pool size

Thread-specific Data

- Allows each thread to have its own copy of data
- Useful for implementing protection
  - For example, user connects to bank's database server
  - Server process responds, has access to all accounts
  - Multiple people may be accessing their accounts at the same time
  - Thread assigned to manipulate or view user's bank account
  - Using thread-specific data limits (unintentional or erroneous) access by other threads

Pthreads

- A POSIX **standard** (IEEE 1003.1c) API for thread creation and synchronization
  - NOT an implementation
  - Implementation in different OSes may be using user or kernel threads
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Fairly portable
- Available on `pyrite`
- Has man pages on Linux (check "`man pthread_create`")
- To use:
  - `#include <pthread.h>`
  - Link with pthread library: `g++ prog.cc -lpthread`

Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

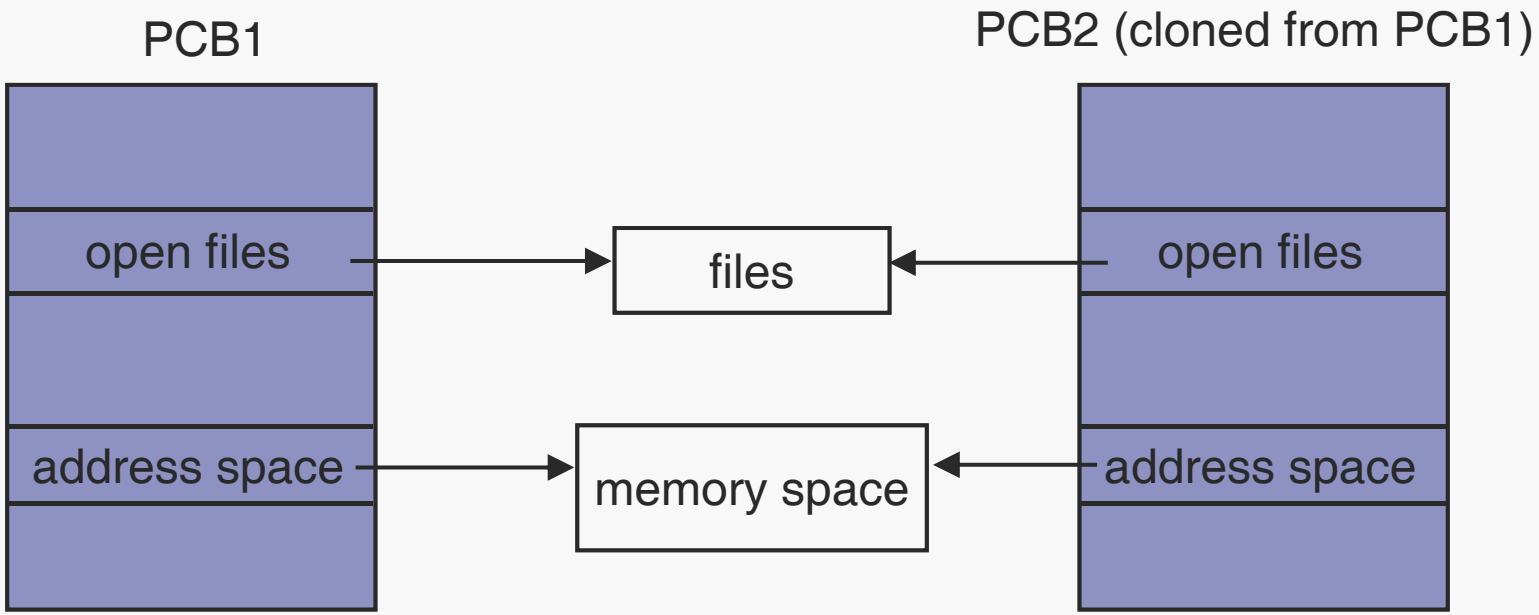
Linux Threads

- Linux refers to them as **tasks** rather than threads
- Thread creation is done through the `clone()` system call
- `clone()` allows a child task to share different things with the parent task such as:
  - the address space
  - the table of open files
  - the signal handlers
- If nothing is shared: same as `fork()`, creates essentially a process
- If everything is shared: creates a thread as we know it

Linux Threads (cont'd)

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

`clone()` is used to create kernel-level threads



To the user: PCB1 and PBC2 are
"kernel-level threads within the same process"

To the kernel: PCB1 and PCB 2 are processes (tasks)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending `Thread` class
  - Implementing the `Runnable` interface
- Details in textbook

Linux Programming

Pthread

Thread Basic

- Thread operations include thread creation, termination, synchronization, data management
- Threads in the same process share:
- Process address space, instructions and most data
- Opened files
- Signals and signal handles
- Each thread has a unique:
- Thread ID
- A register set
- Stack
- Local variables and return address

Pthreads: POSIX Threads

- Pthreads is a standard set of C library functions for multithreaded programming
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
  - Thread management: create, exit, detach, join, . . .
  - Thread cancellation
  - Mutex locks: init, destroy, lock, unlock, . . .
  - Condition variables: init, destroy, wait, timed wait, . . .
  - . . .
- Programs must include the file **`pthread.h`**
- Programs must be linked with the pthread library (**`-lpthread`**)

Pthreads Naming Convention

- Types: pthread[_object]_t
- Functions: pthread[_object]_action
- Constants/Macros: PTHREAD_PURPOSE

- Examples:
  - pthread_t: the type of a thread
  - pthread_create(): creates a thread
  - pthread_mutex_t: the type of a mutex lock
  - pthread_mutex_lock(): lock a mutex
  - PTHREAD_CREATE_DETACHED

Pthread – POSIX thread

- Include pthread header file
- # include <pthread.h>

- Compile and execution
- C compiler: gcc –o foo foo.c -lpthread
- C++ compiler: g++ -o foo foo.c –lpthread
- Execution: ./foo

Example

```
#include <pthread.h>
#include <stdio.h>
void *runner(void *param);
Int     sum;
main( ) {
    pthread_t  tid;  /* the thread ID */
    pthread_attr_t  attr;  /*set of thread attrs */
    pthread_attr_init(&attr);  /* get the default attr */
    pthread_create(&tid, &attr, runner, “10”);  /* create the thread */
    pthread_join(tid, NULL);  /* wait for the thread to exit */
    printf(“sum = %d\n”, sum);
}
void *runner(void *param) {
    int upper = atoi(param);
    sum = 0;
    for (i=0;i<upper;i++)  sum += i;
    pthread_exit(0);
}
```

Pthread – POSIX thread

- Basic pthread functions
  - pthread_create – create child thread
  - pthread_exit – thread termination
  - pthread_join – wait for thread termination
  - phtread_mutex_lock – lock critical section
  - pthread_mutex_unlock – unlock critical section
  - pthread_cond_wait – wait for a condition signal
  - phtread_cond_signal – wake up one waiting thread
  - pthread_cond_broadcast – wake up all waiting threads

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

void *thread_func( void *ptr ){
   char *message = (char *) ptr;
   printf("%s \n", message);
}
int main(int argc, char **argv){
   pthread_t thread1, thread2;
   char *msg1 = "Thread 1";
   char *msg2 = "Thread 2";
   int  ret1, ret2;

   ret1 = pthread_create(&thread1,NULL,thread_func,(void*)msg1);
   ret2 = pthread_create(&thread2,NULL,thread_func,(void*)msg2);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   printf("Thread 1 returns: %d\n",ret1);
   printf("Thread 2 returns: %d\n",ret2);
   return 0;
}
```

pthread_self()

- Returns the thread identifier for the calling thread
  - At any point in its instruction stream a thread can figure out which thread it is
  - Convenient to be able to write code that says: “If you’re thread 1 do this, otherwise do that”
  - However, the thread identifier is an opaque object (just a pthread_t value)
  - you must use pthread_equal() to test equality

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t id1, pthread_t id2);
```

pthread_create()

- Creates a new thread

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

  - Returns 0 to indicate success, otherwise returns error code
  - **`thread`**: output argument for the id of the new thread
  - **`attr`**: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
  - **`start_routine:`** function to use as the start of the new thread
    - must have prototype: void * foo(void*)
  - **`arg:`** argument to pass to the new thread routine
    - If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure

pthread_create() example

- Want to create a thread to compute the sum of the elements of an array

```
void *do_work(void *arg);
```

- Needs three arguments
  - the array, its size, where to store the sum
  - we need to bundle them in a structure

```
struct arguments {
        double *array;
        int size;
        double *sum;
}
```

pthread_create() example

```
int main(int argc, char *argv) {
  double array[100];
  double sum;
  pthread_t worker_thread;
  struct arguments *arg;

  arg = (struct arguments *)calloc(1,
                              sizeof(struct arguments));
  arg->array = array;
  arg->size=100;
  arg->sum = &sum;

  if (pthread_create(&worker_thread, NULL,
                     do_work, (void *)arg)) {
    fprintf(stderr,"Error while creating thread\n");
    exit(1);
  }
  ...
}
```

pthread_create() example

```
void *do_work(void *arg) {
 struct arguments *argument;
 int i, size;
 double *array;
 double *sum;

 argument = (struct arguments*)arg;

 size = argument->size;
 array = argument->array;
 sum = argument->sum;

 *sum = 0;
 for (i=0;i<size;i++)
    *sum += array[i];

 return NULL;
}
```

Comments about the example

- The “main thread” continues its normal execution after creating the “child thread”
- IMPORTANT: If the main thread terminates, then all threads are killed!
  - We will see that there is a join() function
- Of course, memory is shared by the parent and the child (the array, the location of the sum)
  - nothing prevents the parent from doing something to it while the child is still executing
  - which may lead to a wrong computation
  - we will see that Pthreads provide locking mechanisms
- The bundling and unbundling of arguments is a bit tedious

Memory Management of Args

- The parent thread allocates memory for the arguments
- Warning #1: you don't want to free that memory before the child thread has a chance to read it
  - That would be a race condition
  - Better to let the child do the freeing
- Warning #2: if you create multiple threads you want to be careful there is no sharing of arguments, or that the sharing is safe
  - For instance, if you reuse the same data structure for all threads and modify its fields before each call to pthread_create(), some threads may not be able to read the arguments destined to them
  - Safest way: have a separate arg structure for each thread

pthread_exit()

- Terminates the calling thread

```
void pthread_exit(void *retval);
```

  - The return value is made available to another thread calling a **`pthread_join()`** (see next slide)
  - My previous example had the thread just return from **`function do_work()`**
    - In this case the call to **`pthread_exit()`** is implicit
    - The return value of the function serves as the argument to the (implicitly called) **`pthread_exit()`**.

pthread_join()

- Causes the calling thread to wait for another thread to terminate

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

  - **`thread`**: input parameter, id of the thread to wait on
  - **`value_ptr`**: output parameter, value given to **`pthread_exit()`** by the terminating thread (which happens to always be a **`void *`**)
  - returns 0 to indicate success, error code otherwise
  - multiple simultaneous calls for the same thread are not allowed

pthread_kill()

- Causes the termination of a thread

```
int pthread_kill(
      pthread_t thread,
      int sig);
```

  - **`thread`**: input parameter, id of the thread to terminate
  - **`sig:`** signal number
  - returns 0 to indicate success, error code otherwise

pthread_join() example

```
int main(int argc, char *argv) {
  double array[100];
  double sum;
  pthread_t worker_thread;
  struct arguments *arg;
  void  *return_value;

  arg = (struct arguments *)calloc(1,sizeof(struct arguments));
  arg->array = array;
  arg->size=100;
  arg->sum = &sum;
  if (pthread_create(&worker_thread, NULL,
                     do_work, (void *)arg)) {
    fprintf(stderr,”Error while creating thread\n”);
    exit(1);
  }
  ...
  if (pthread_join(worker_thread, &return_value)) {
    fprintf(stderr,”Error while waiting for thread\n”);
    exit(1);
  }
}
```

pthread_join() Warning

- This is a common “bug” that first-time pthread programmers encounter
- Without the call to pthread_join() the previous program may end immediately, with the main thread reaching the end of main() and exiting, thus killing all other threads perhaps even before they have had a chance to execute

pthread_join() Warning

- When creating multiple threads be careful to store the handle of each thread in a separate variable
  - Typically one has an array of thread handles
- That way you'll be able to call pthread_join() for each thread
- Also, note that the following code is sequential!

```
for (i=0; i < num_threads; i++) {
        pthread_create(&(threads[i]),...)
        pthread_join(threads[i],...)
}
```

Thread Attributes

- One of the parameters to pthread_create() is a thread attribute
- In all our previous examples we have set it to NULL
- But it can be very useful and provides a simple way to set options:
  - Initialize an attribute
  - Set its value with some Pthread API call
  - Pass it to Pthread API functions like pthread_create()

pthread_attr_init()

- Initialized the thread attribute object to the default values

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

  - Return 0 to indicate success, error code otherwise
  - **`attr`**: pointer to a thread attribute

Detached Thread

- One option when creating a thread is whether it is joinable or detached
  - Joinable: another thread can call join on it
    - By default a thread is joinable
  - Detached: no thread can call join on it
- Let's look at the function that allows to set the "detached state"

pthread_attr_setdetachstate()

- Sets the detach state attribute

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

  - returns 0 to indicate success, error code otherwise
  - attr: input parameter, thread attribute
  - detachstate: can be either
    - PTHREAD_CREATE_DETACHED
    - PTHREAD_CREATE_JOINABLE (default)

Detach State

- Detached threads have all resources freed when they terminate
- Joinable threads have state information about the thread kept even after they finish
  - To allow for a thread to join a finished thread
  - So-called “no rush to join”
- So, if you know that you will not need to join a thread, create it in a detached state so that you save resources
- This is lean-and-mean C, as opposed to hand-holding Java, and every little saving is important

Creating a Detached Thread

```
#include <pthread.h>
#define NUM_THREAD 25

void *thread_routine (void *arg) {
  printf("Thread %d, my TID is %u\n",
     (int)arg, pthread_self());
  pthread_exit(0);
}
```

Creating a Detached Thread

```
int main() {
  pthread_attr_t attr;
  pthread_t tids[NUM_THREADS];
  int x;

  pthread_attr_init(&attr);

   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACH
   ED);
  phthread_create(&(tids[x]),
                  &attr,
                  thread_routine,
                  (void *)x);

  . . . // should take a while otherwise
  . . . // the child may not have time to run
}
```

Mutual Exclusion and Pthreads

- Pthreads provide a simple mutual exclusion lock
- Lock creation

```
int pthread_mutex_init(
        pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr);
```

  - returns 0 on success, an error code otherwise
  - **`mutex`**: output parameter, lock
  - **`attr`**: input, lock attributes
    - NULL: default
    - There are functions to set the attribute (look at the man pages if you're interested)

Linux Programming
pthread mutex

Pthread: Locking

- Locking a lock
  - If the lock is already locked, then the calling thread is blocked
  - If the lock is not locked, then the calling thread acquires it

```
int pthread_mutex_lock(
        pthread_mutex_t *mutex);
```

  - returns 0 on success, an error code otherwise
  - **`mutex`**: input parameter, lock

Pthread: Locking

- Just checking
  - Returns instead of locking

```
int pthread_mutex_trylock(
        pthread_mutex_t *mutex);
```

  - returns 0 on success, EBUSY if the lock is locked, an error code otherwise
  - **`mutex`**: input parameter, lock

Synchronizing pthreads

- Releasing a lock

```
int pthread_mutex_unlock(
        pthread_mutex_t *mutex);
```

  - returns 0 on success, an error code otherwise
  - **`mutex`**: input parameter, lock

- Pthreads implement exactly the concept of locks as it was described in the previous lecture notes

Mutex example

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main(){
    .. .. ..
}

void *thread_func()
{
   // protect shared variable counter
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Cleaning up memory

- Releasing memory for a mutex attribute

```
int pthread_mutex_destroy(
        pthread_mutex_t *mutex);
```

- Releasing memory for a mutex

```
int pthread_mutexattr_destroy(
        pthread_mutexattr_t *mutex);
```

Linux Programming
pthread Condition variables

Pthread Synchronization

- Condition variables –

  Condition variable is used with the appropriate functions for waiting and later continuation

Creating/Destroying:

```
pthread_cond_init()
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
Pthread_cond_destroy()
```

Waiting on condition:

```
pthread_cond_wait()
pthread_cond_timewait(timeout)
```

Waking thread based on condition:

`pthread_cond_signal()` – wake any one thread

`phread_cond_broadcast()` - wake up all threads blocked

Condition Variables

- Pthreads also provide condition variables as they were described in the previous lecture notes
- Condition variables are of the type pthread_cond_t
- They are used in conjunction with mutex locks

- Let’s look at the API’s functions

pthread_cond_init()

- Creating a condition variable

```
int pthread_cond_init(
      pthread_cond_t *cond,
      const pthread_condattr_t *attr);
```

  - returns 0 on success, an error code otherwise
  - **cond**: output parameter, condition
  - **attr**: input parameter, attributes (default = NULL)

pthread_cond_wait()

- Waiting on a condition variable

```
int pthread_cond_wait(
      pthread_cond_t *cond,
      pthread_mutex_t *mutex);
```

  - returns 0 on success, an error code otherwise
  - **cond**: input parameter, condition
  - **mutex**: input parameter, associated mutex

pthread_cond_signal()

- Signaling a condition variable

```
int pthread_cond_signal(
      pthread_cond_t *cond;
```

  - returns 0 on success, an error code otherwise
  - **cond**: input parameter, condition

- “Wakes up” one thread out of the possibly m any threads waiting for the condition
  - The thread is chosen non-deterministically

pthread_cond_broadcast()

- Signaling a condition variable

```
int pthread_cond_broadcast(
      pthread_cond_t *cond;
```

  - returns 0 on success, an error code otherwise
  - **cond**: input parameter, condition

- "Wakes up" ALL threads waiting for the condition
  - May be useful in some applications

Condition Variable: example

- Say I want to have multiple threads wait until a counter reaches a maximum value and be awakened when it happens

```
pthread_mutex_lock(&lock);
while (count < MAX_COUNT) {
  pthread_cond_wait(&cond,&lock);
}
pthread_mutex_unlock(&lock)
```

  - Locking the lock so that we can read the value of count without the possibility of a race condition
  - Calling **`pthread_cond_wait()`** in a while loop to avoid "spurious wakes ups"
  - When going to sleep the **`pthread_cond_wait()`** function implicitly releases the lock
  - When waking up the **`pthread_cond_wait()`** function implicitly acquires the lock
  - The lock is unlocked after exiting from the loop

pthread_cond_timed_wait()

- Waiting on a condition variable with a timeout

```
int pthread_cond_timedwait(
      pthread_cond_t *cond,
     pthread_mutex_t *mutex,
     const struct timespec *delay);
```

  - returns 0 on success, an error code otherwise
  - **cond**: input parameter, condition
  - **mutex**: input parameter, associated mutex
  - **delay:** input parameter, timeout (same fields as the one u sed for gettimeofday)

Synchronization – Condition example

```
void *thread_func_1() {
   for(;;) {
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
         pthread_cond_wait( &condition_cond, &condition_mutex );

      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value thread_func_1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}
```

Synchronization – Condition example(con't)

```
void *thread_func_2() {
    for(;;) {
       pthread_mutex_lock( &condition_mutex );
       if( count < COUNT_HALT1 || count > COUNT_HALT2 )
          pthread_cond_signal( &condition_cond );

       pthread_mutex_unlock( &condition_mutex );

       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value thread_func_2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
    }
}
```

Putting a Pthread to Sleep

- To make a Pthread thread sleep you should use the usleep() function

```
#include <unistd.h>

int usleep(usecond_t microseconds);
```

- Do not use the sleep() function as it may not be safe to use it in a multi-threaded program

PThreads: Conclusion

- Pthreads implement everything we talked about in the previous set of lecture notes almost directly
- There are many other useful functions, some which you may even need for your programming assignment
  - e.g., Read/Write locks

Beyond Locks and Cond. Vars

- At this point, we have everything we need to write concurrent programs
  - Locks for efficient mutual exclusion
  - Condition variables for non-wasteful thread synchronization and/or communication

- In the next lecture we'll see another abstraction for thread synchronization and mutual exclusion: Semaphores

Pthread Basic

- The same function is used in each thread, but the arguments of each thread are different
- Thread termination
  - thread itself calls *pthread_exit()* explictly
  - main process exits and all threads are terminated
- Be careful to the order of mutex posted or it may cause deadlock

Linux Programming
pthread main api

Main API

- pthread_atfork - register handlers to be called at fork(2) time
- pthread_attr_destroy [pthread_attr_init] - thread creation attributes
- pthread_attr_getdetachstate [pthread_attr_init] - thread creation attributes
- pthread_attr_getinheritsched [pthread_attr_init] - thread creation attributes
- pthread_attr_getschedparam [pthread_attr_init] - thread creation attributes
- pthread_attr_getschedpolicy [pthread_attr_init] - thread creation attributes
- pthread_attr_getscope [pthread_attr_init] - thread creation attributes

Main API

- pthread_attr_init - thread creation attributes
- pthread_attr_setdetachstate [pthread_attr_init] - thread creation attributes
- pthread_attr_setinheritsched [pthread_attr_init] - thread creation attributes
- pthread_attr_setschedparam [pthread_attr_init] - thread creation attributes
- pthread_attr_setschedpolicy [pthread_attr_init] - thread creation attributes
- pthread_attr_setscope [pthread_attr_init] - thread creation attributes
- pthread_cancel - thread cancellation

Main API

- pthread_cleanup_pop [pthread_cleanup_push] - install and remove cleanup handlers
- pthread_cleanup_pop_restore_np [pthread_cleanup_push] - install and remove cleanup handlers
- pthread_cleanup_push - install and remove cleanup handlers
- pthread_cleanup_push_defer_np [pthread_cleanup_push] - install and remove cleanup handlers

Main API

- pthread_condattr_destroy [pthread_condattr_init] - condition creation attributes
- pthread_condattr_init - condition creation attributes
- pthread_cond_broadcast [pthread_cond_init] - operations on conditions
- pthread_cond_destroy [pthread_cond_init] - operations on conditions
- pthread_cond_init - operations on conditions
- pthread_cond_signal [pthread_cond_init] - operations on conditions
- pthread_cond_timedwait [pthread_cond_init] - operations on conditions
- pthread_cond_wait [pthread_cond_init] - operations on conditions

Main API

- pthread_create - create a new thread
- pthread_detach - put a running thread in the detached state
- pthread_equal - compare two thread identifiers
- pthread_exit - terminate the calling thread

Main API

- pthread_getschedparam [pthread_setschedparam] - control thread scheduling parameters
- pthread_getspecific [pthread_key_create] - management of thread-specific data
- pthread_join - wait for termination of another thread
- pthread_key_create - management of thread-specific data
- pthread_key_delete [pthread_key_create] - management of thread-specific data
- pthread_kill_other_threads_np - terminate all threads in program except calling thread
- pthread_kill [pthread_sigmask] - handling of signals in threads

Main API

- pthread_mutexattr_destroy [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutexattr_getkind_np [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutexattr_init - mutex creation attributes
- pthread_mutexattr_setkind_np [pthread_mutexattr_init] - mutex creation attributes
- pthread_mutex_destroy [pthread_mutex_init] - operations on mutexes
- pthread_mutex_init - operations on mutexes
- pthread_mutex_lock [pthread_mutex_init] - operations on mutexes
- pthread_mutex_trylock [pthread_mutex_init] - operations on mutexes
- pthread_mutex_unlock [pthread_mutex_init] - operations on mutexes

Main API

- pthread_once - once-only initialization
- pthread_self - return identifier of current thread
- pthread_setcancelstate [pthread_cancel] - thread cancellation
- pthread_setcanceltype [pthread_cancel] - thread cancellation
- pthread_setschedparam - control thread scheduling parameters
- pthread_setspecific [pthread_key_create] - management of thread-specific data
- pthread_sigmask - handling of signals in threads
- pthread_testcancel [pthread_cancel] - thread cancellation

Reference

- PThread Tutorial: https://computing.llnl.gov/tutorials/pthreads/
- http://www.humanfactor.com/pthreads/
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- QNX 632 Help.chm

THANKS