Pthreads

Non of this material has been created by me! I have borrowed it all from books and from the net

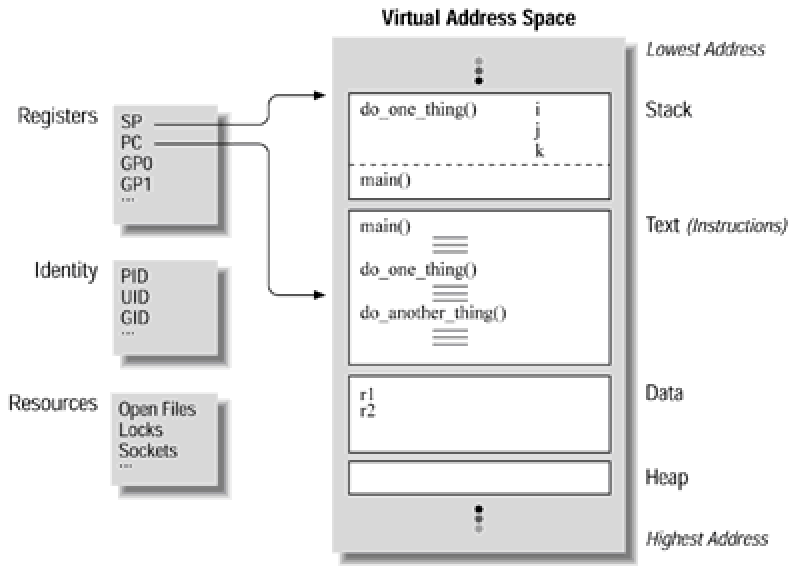
Threads Vs Processes

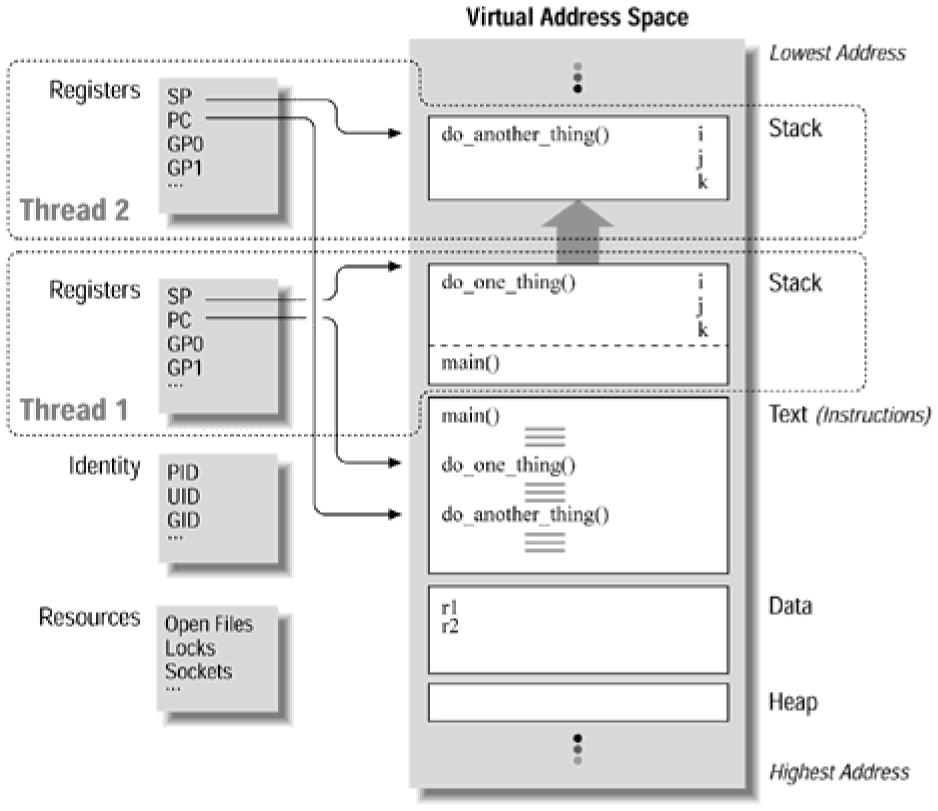
1. Threads require less program and system overhead to run than processes do. The operating system performs less work on behalf of a multithreaded program than it does for a multiprocess program. This translates into a performance gain for the multithreaded program.
2. You create a new process by using the UNIX *fork* system call, you create a new thread by calling the *pthread_create* Pthreads function
3. Threads don't have a parent/child relationship as processes do. So, any thread can cancel any other thread, as long as the canceling thread has the thread handle of its victim. Because you want your application to be solidly structured, you'll cancel threads only from the thread that initially created them. The only thread that has slightly different properties than any other is the first thread in the process, which is known as the *main* thread.
4. Wait = pthread_join

A Simple C Program with Concurrent Threads (simple_threads.c)

```
#include <stdio.h>
#include <pthread.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);
int r1 = 0, r2 = 0;
extern int
main(void)
{
  pthread_t       thread1, thread2;
  pthread_create(&thread1,
          NULL,
          (void *) do_one_thing,
          (void *) &r1);
  pthread_create(&thread2,
          NULL,
          (void *) do_another_thing,
          (void *) &r2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  do_wrap_up(r1, r2);
  return 0;
}
```

The pthread_t type may look a little strange to you if you're used to the data types returned by C language system calls on many UNIX systems. Because many of these types (like int) reveal quite a bit about the underlying architecture of a given platform (such as whether its addresses are 16, 32, or 64 bits long), POSIX prefers to create new data types that conceal these fundamental differences. By convention, the names of these data types end in _t.


Virtual Address Space
Lowest Address
Registers
SP
PC
GP0
GP1
...
do_one_thing()
i
j
k
main()
Stack
Text (Instructions)
main()
do_one_thing()
do_another_thing()
Identity
PID
UID
GID
...
Resources
Open Files
Locks
Sockets
...
r1
r2
Data
Heap
Highest Address


Virtual Address Space
Lowest Address
Registers
SP
PC
GP0
GP1
...
Thread 2
do_another_thing()
i
j
k
Stack
Registers
SP
PC
GP0
GP1
...
Thread 1
do_one_thing()
i
j
k
main()
Stack
Text (Instructions)
main()
do_one_thing()
do_another_thing()
Identity
PID
UID
GID
...
Resources
Open Files
Locks
Sockets
...
r1
r2
Data
Heap
Highest Address

The idea of threads is that multiple threads of execution can share a lot of resources; for instance, they generally operate in the same address space.

Switching from one thread to another is generally cheaper than switching from one process to another.

Furthermore, for processes using a lot of memory, threads may allow substantially more efficient use of memory.

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.

Upon creation, each thread executes a thread function and 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.

There are around 100 Pthreads procedures, all prefixed pthread_ and they can be categorized into four groups:

1. Thread management - creating, joining threads etc.
2. Mutexes
3. Condition variables
4. Synchronization between threads using read/write locks and barriers

The threads library provides three synchronization mechanisms:

mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

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. As with pthread_create, if the attribute pointer is null, default attributes are assumed.The mutex variable should be initialized only once. This code fragment demonstrates the declaration and initialization of a mutex variable.

```
pthread_mutex_t mutex;

pthread_mutex_init (&mutex, NULL);
```

OR

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Occasionally, it is useful to test whether a mutex is locked without actually blocking on it. For instance, a thread may need to lock a mutex but may have other work to do instead of blocking if the mutex is already locked.

GNU/Linux provides pthread_mutex_trylock for this purpose. If you call pthread_mutex_trylock on an unlocked mutex, you will lock the mutex as if you had called pthread_mutex_lock, and pthread_mutex_trylock will return zero. However, if the mutex is already locked by another thread, pthread_mutex_trylock will not block.

Example:

```
/* A mutex protecting job_queue. */

pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

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 because we’re done with the
queue for now. */
pthread_mutex_unlock (&job_queue_mutex);
```

Section: joins - Make a thread wait till others are complete (terminated).

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? *Nothing!* But if this happens, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it.

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.

The favored way to **correctly** shutdown a multithreaded program, it to make sure no thread is in a random state. Stop all the threads in some way or another, call a join on them where feasible, and from the last remaining thread call exit - or return if this happens in the main function.

condition variables - data type pthread_cond_t

While a mutex lets threads synchronize by controlling their access to data, a condition variable lets threads synchronize on the value of data.

Cooperating threads wait until data reaches a particular state or until a certain event occurs.

A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).

To declare such a condition variable, one simply writes something like this: pthread cond t c;, which declares c as a condition variable (note: proper initialization is also required). A condition variable has two operations associated with it: wait() and signal(). The wait() call is executed when a thread wishes to put itself to sleep; the signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition. Specifi- cally, the POSIX calls look like this:

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);

pthread_cond_signal(pthread_cond_t *c);

A mutex is required alone with the condition variable. The mutex is used to protect *the condition variable itself*. That's why you need it locked before you do a wait.

The wait will "atomically" unlock the mutex, allowing others access to the condition variable (for signalling). Then when the condition variable is signalled or broadcast to, one or more of the threads on the waiting list will be woken up and the mutex will be magically locked again for that thread.

Below we'll make the global variable count a shared resource that two threads increment and create the mutex count_mutex (in global scope) to protect it. We'll use the count_threshold_cv condition variable to represent an event≈the count variable's reaching a defined threshold value, WATCH_COUNT.

The main routine creates two threads. Each of these threads runs the inc_count routine. The inc_count routine locks count_mutex, increments count, reads count in a printf statement, and tests for the threshold value. If count has reached its threshold value, inc_count calls pthread_cond_signal to notify the thread that's waiting for this particular event. Before exiting, inc_count releases the mutex. We'll create a third thread to run the watch_count task. The watch_count routine waits for inc_count to signal our count_threshold_cv condition variable.

A Simple Condition Variable Example (cvsimple.c)

```
#include <stdio.h>
#include <pthread.h>
#define TCOUNT 10
#define WATCH_COUNT 12
int count = 0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_threshold_cv = PTHREAD_COND_INITIALIZER;
int  thread_ids[3] = {0,1,2};
extern int
main(void)
{
     int      i;
     pthread_t threads[3];
     pthread_create(&threads[0],NULL,inc_count, &thread_ids[0]);
     pthread_create(&threads[1],NULL,inc_count, &thread_ids[1]);
     pthread_create(&threads[2],NULL,watch_count,
&thread_ids[2]);
     for (i = 0; i < 3; i++) {
              pthread_join(threads[i], NULL);
     }
     return 0;
}
void watch_count(int *idp)
{
     pthread_mutex_lock(&count_mutex)
     while (count <= WATCH_COUNT) {
              pthread_cond_wait(&count_threshold_cv,
                                 &count_mutex);
              printf("watch_count(): Thread %d,Count is %d\n",
                   *idp, count);
     }
     pthread_mutex_unlock(&count_mutex);
}
void inc_count(int *idp)
{
     for (i =0; i < TCOUNT; i++) {
              pthread_mutex_lock(&count_mutex);
              count++;
              printf("inc_count(): Thread %d, old count %d,\
                   new count %d\n", *idp, count - 1, count );
              if (count == WATCH_COUNT)
                   pthread_cond_signal(&count_threshold_cv);
              pthread_mutex_unlock(&count_mutex);
     }
}
```

Section: Thread Cancellation

Cancellation allows one thread to terminate another.

One reason you may want to cancel a thread is to save system resources (such as CPU time) when your program determines that the thread's activity is no longer necessary

A simple example of a thread you might want to cancel would be a thread performing a read-only data search. If one thread returns the results you are looking for, all other threads running the same routine could be canceled.

The ability of a thread to go away or not go away when asked by another thread is known as its *cancelability state*. You must consider when the thread might go away≈maybe immediately, maybe a bit later. The degree to which a thread persists after it has been asked to go away is known as its *cancelability type*. OR you can set a given thread's cancel ability (that is, its ability to allow itself to be canceled).

Best Practice:

1. The simplest approach is to restrict the use of cancellation to threads that execute only in simple routines that do not hold locks or ever put shared data in an inconsistent state.
2. Another option is to restrict cancellation to certain points at which a thread is known to have neither locks nor resources.
3. Lastly, you could create a cleanup stack for the thread that is to be canceled; it can then use the cleanup stack to release locks and reset the state of shared data.
4. When a thread holds no locks and has no resources allocated, asynchronous cancellation is a valid option. When a thread must hold and release locks, it might temporarily disable cancellation altogether.

Cancelability of a Thread

Cancelability State	Cancelability Type	Description
PTHREAD_ CANCEL_ DISABLE	Ignored	Disabled. The thread can never be canceled. Calls to *pthread_cancel* have no effect. The thread can safely acquire locks and resources.
PTHREAD_ CANCEL_ ENABLE	PTHREAD_ CANCEL_ ASYNCHRONOUS	Asynchronous cancellation. Cancellation takes effect immediately.*
PTHREAD_ CANCEL_ ENABLE	PTHREAD_ CANCEL_ DEFERRED	Deferred cancellation (the default). Cancellation takes effect only if and when the thread enters a cancellation point. The thread can hold and release locks but must keep data in some consistent state. If a pending cancellation exists

at a cancellation point, the thread can terminate without leaving problems behind for the remaining threads.

The Simple Cancellation Example≈main (cancel.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#define NUM_THREADS 3
int count = NUM_THREADS;
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init_done=PTHREAD_COND_INITIALIZER;
int id_arg[NUM_THREADS] = {;0,1,2};
extern int
main(void)
{;
  int i;
  void *statusp;
  pthread_t threads[NUM_THREADS];
  /**** Create the threads ****/
  pthread_create(&(threads[0]), NULL, ask_for_it, (void *)
&(id_arg[0]));
  pthread_create(&(threads[1]), NULL, sitting_duck, (void *)
&(id_arg[1]));
  pthread_create(&(threads[2]), NULL, bullet_proof, (void *)
&(id_arg[2]));
  printf("main(): %d threads created\n",count);
  /**** wait until all threads have initialized ****/
  pthread_mutex_lock(&lock);
  while (count != 0) {;
    pthread_cond_wait(&init_done, &lock);
  }
  pthread_mutex_unlock(&lock);
  printf("main(): all threads have signaled that they're
ready\n");
  /**** cancel each thread ****/
  for (i = 0; i < NUM_THREADS; i++) {;
    pthread_cancel(threads[i]);
  }
  /**** wait until all threads have finished ****/
  for (i = 0; i < NUM_THREADS; i++) {;
    pthread_join(threads[i], &statusp);
    if (statusp == PTHREAD_CANCELED) {;
       printf("main(): joined to thread %d,
statusp=PTHREAD_CANCELED\n", i);
    } else {;
       printf("main(): joined to thread %d \n", i);
    }
  }
  printf("main(): all %d threads have finished. \n",
NUM_THREADS);
  return 0;
}
```

The Simple Cancellation Example≈bullet_proof (cancel.c)

```
void *bullet_proof(int *my_id)

{;

  int i=0, last_state;

  char *messagep;

  messagep = (char *)malloc(MESSAGE_MAX_LEN);

  sprintf(messagep, "bullet_proof, thread #%d: ", *my_id);

  printf("%s\tI'm alive, setting general cancelability OFF\n",
messagep);

  /* We turn off general cancelability here */

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &last_state);

  pthread_mutex_lock(&lock);

  {;

  printf("\n%s signaling main that my init is done\n", messagep);

  count -= 1;

  /* Signal to program that loop is being entered */

  pthread_cond_signal(&init_done);

  pthread_mutex_unlock(&lock);

  }

  /* Loop forever until picked off with a cancel */

  for(;;i++) {;

    if (i%10000 == 0)

      print_count(messagep, *my_id, i);
    if (i%100000 == 0)

      printf("\n%s This is the thread that never ends... #%d\n",
messagep, i);

  }

  /* Never get this far */

  return(NULL);

}
```

The Simple Cancellation Example≈ask_for_it (cancel.c)

```
void *ask_for_it(int *my_id)

{;

  int i=0, last_state, last_type;

  char *messagep;

  messagep = (char *)malloc(MESSAGE_MAX_LEN);

  sprintf(messagep, "ask_for_it, thread #%d: ", *my_id);

  /* We can turn on general cancelability here and disable async
cancellation. */

  printf("%s\tI'm alive, setting deferred cancellation ON\n",
messagep);

  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &last_state);

  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &last_type);

  pthread_mutex_lock(&lock);

  {;

  printf("\n%s signaling main that my init is done\n", messagep);

  count -= 1;

  /* Signal to program that loop is being entered */

  pthread_cond_signal(&init_done);

  pthread_mutex_unlock(&lock);
```

```
    }

    /* Loop forever until picked off with a cancel */

    for(;;i++) {;

      if (i%1000 == 0)

        print_count(messagep, *my_id, i);

      if (i%10000 == 0)

        printf("\n%s\tLook, I'll tell you when you can cancel
  me.%d\n", messagep, i);

        pthread_testcancel();

    }

    /* Never get this far */

    return(NULL);

  }
```

The Simple Cancellation Example≈sitting_duck (cancel.c)

```
  void *sitting_duck(int *my_id)

  {;

    int i=0, last_state, last_type, last_tmp;

    char messagep;

    messagep = (char *)malloc(MESSAGE_MAX_LEN);

    sprintf(messagep, "sitting_duck, thread #%d: ", *my_id);

    pthread_mutex_lock(&lock);

    {;

      printf("\n%s signaling main that my init is done\n",
  messagep);

      count -= 1;

      /* Signal to program that loop is being entered */
      pthread_cond_signal(&init_done);

      pthread_mutex_unlock(&lock);

    }

    /* Now, we're safe to turn on async cancelability */

    printf("%s\tI'm alive, setting async cancellation ON\n",
  messagep);

    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &last_type);

    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &last_state);

    /* Loop forever until picked off with a cancel */

    for(;;i++) {;

      if (i%1000) == 0)

        print_count(messagep, *my_id, i);

      if (i%10000 == 0) {;

        pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &last_tmp);

        printf("\n%s\tHum, nobody here but us chickens. %d\n",
  messagep, i);

        pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
  &last_tmp);

        }

    }

    /* Never get this far */

    return(NULL);

  }
```

Programs:

1. Write a program to take input from user for number of files to be scanned and word to be searched. write a multi threaded program to search the files and return pattern if found
2. Write a program to find number of CPUs, create that many threads and attach those threads to CPUs
3. Write a short program that creates 5 threads which print a tread "id" that is passed to thread function by pointer.