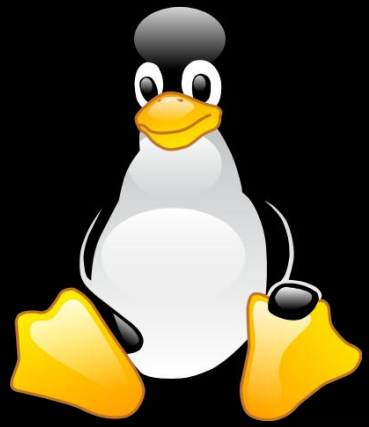
*Tushar B Kute,*
Assistant Professor,
Sandip Institute of Technology and Research Centre, Nashik (INDIA)
tbkute@gmail.com

University of Pune
T.E. I.T.
Subject code: 314441

# OPERATING SYSTEM

## Part 18: POSIX Threads



# Thread

- Multiple strands of execution in a single program are called threads. A more precise definition is that a thread is a sequence of control within a process.
- When we create a new thread in a process, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.

# POSIX

- Linux first acquired thread support around 1996, with a library often referred to as “LinuxThreads.” This was very close to the POSIX standard
- POSIX: "Portable Operating System Interface for Unix" is the name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces, for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

# Advantages of threads

- Sometimes it is very useful to make a program appear to do two things at once. The classic example: MS-Word.
- The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads
- Now that multi-cored CPUs are common even in desktop and laptop machines, using multiple threads inside a process can, if the application is suitable, enable a single process to better utilize the hardware resources available.
- In general, switching between threads requires the operating system to do much less work than switching between processes.

# Drawbacks of thread

- Writing multithreaded programs requires very careful design. The potential for introducing. Alan Cox (the well respected Linux guru) has commented that threads are also known as “how to shoot yourself in both feet at once.”
- Debugging a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine.

# POSIX Threads

- Based on IEEE POSIX 1003.1c standard.
- Almost all vendors support POSIX threads in one way or another
- Shared memory based
- Characteristics
  - Thread creation, management and destruction
  - Synchronization
- Many people refer them as light weight processes
  - Independent flow of control
    - Maintain stack pointer, registers, scheduling, set of pending and block signals and thread specific data.
  - Share any other process resources such as (shared) memory.

# POSIX threads

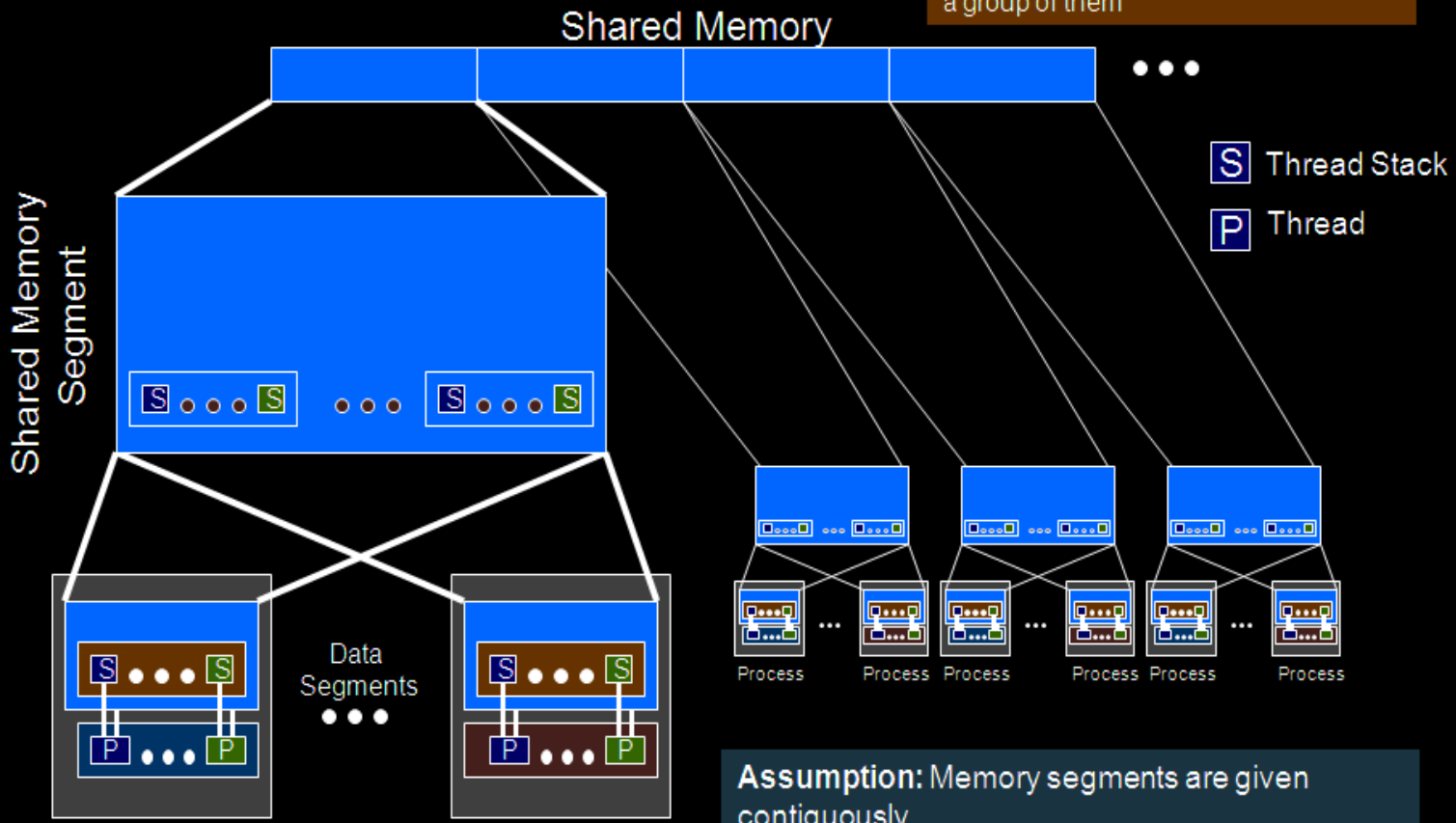
## A Logical View

**System V IPC**

A main structure that contains all pages in the shared memory

**POSIX IPC**

Memory is defined as a file system and each page can be accessed as a file or a group of them




**Assumption:** Memory segments are given contiguously.

# POSIX Threads

- Independent flow of control → Functions
- Function parameter → Data
- Data can be passed by value or by reference
  - In Pthreads all arguments are passed as a void pointer. Thus it is usually to be considered pass by reference, even though you can trick this!!!!

# POSIX Threads

## *Basic Datatypes:*

| | |
|---|---|
| **pthread_t** *var*; | A variable that will be used as an ID for a thread |
| **pthread_attr_t** *var*; | A variable that defines the thread behavior (i.e. scheduling, priority, stack size, scope, etc) |
| **pthread_mutex_t** *var*; | The definition of a Mutually Exclusive lock in Pthreads |
| **pthread_mutexattr_t** *var*; | Defines the mutex behavior. Either PTHREAD_PROCESS_PRIVATE (the default) and PTHREAD_PROCESS_SHARED |
| **pthread_cond_t** *var*; | The definition of a Conditional variable in Pthreads |
| **pthread_condattr_t** *var*; | Defines the conditional variable behavior. Either PTHREAD_PROCESS_PRIVATE (the default) and PTHREAD_PROCESS_SHARED |

# POSIX Threads

## *A Side Note:*

Threads keep an independent stack pointer

Stacks are not shared among threads

Auto variables are as if they were local to the thread that declare them

If the parameters are passed by reference to the thread, then they will be considered as shared

# POSIX Threads

## *Thread creation:*

Return a non zero value in success

```
int pthread_create(pthread_t *thr, const pthread_attr_t *attr,
                   void *(*start_routine)(void), void *arg)
```

| | |
|---|---|
| pthread_t *thr | Will contain the newly created thread's id. Must be passed by reference |
| const pthread_attr_t *attr | Give the attributes that this thread will have. Use NULL for the default ones |
| void *(*start_routine)(void) | The name of the function that the thread will run. Must have a void pointer as its return and parameters values |
| void *arg | The argument for the function that will be the body of the Pthreads |

Pointers of the type void can reference ***ANY*** type of data, but they ***CANNOT*** be used in any type of operations that reads or writes its data without a cast

# POSIX Threads

## *Miscellaneous Useful functions:*

*pthread_t* **pthread_self**(*void*)

Return the id of the calling thread. Returns a pthread_t type which is usually an integer type variable

***OpenMP Counterpart***

*int* **omp_get_thread_num**(*void*);

*void* **pthread_exit**(void **arg*);

This function will indicate the end of a Pthread and the returning value will be put in *arg*

# POSIX Threads

*Example 1:*

```
#include <pthread.h>
#define NUM_THREADS 4
void *work(void *i){
  printf("Hello, world from %i\n", pthread_self());
  pthread_exit(NULL);
}
int main(int argc, char **argv){
  int i;
  pthread_t id[NUM_THREADS];
  for(i = 0; i < NUM_THREADS; ++i){
    if(pthread_create(&id[i], NULL, work, NULL)){
      printf("Error creating the thread\n"); exit(19);
    }
  }
  printf("After creating the thread. My id is: %i\n",
            pthread_self());
return 0;}
```

```
Hello, world from 2
Hello, world from 3
After creating the thread.
My id is: 1
Hello, world from 4
```

***What happened to thread 5???***

```
Hello, world from 2
Hello, world from 3
Hello, world from 4
After creating the thread.
My id is: 1
Hello, world from 5
```

# POSIX Threads

- *Single Argument Passing*
  - Cast its value as a void pointer (a tricky pass by value)
  - Cast its address as a void pointer (pass by reference).
    - The value that the address is pointing should NOT change between Pthreads creation
- *Multiple Argument Passing*
  - Heterogonous: Create an structure with all the desired arguments and pass an element of that structure as a void pointer.
  - Homogenous: Create an array and then cast it as a void pointer

# POSIX Threads

## *Example 2a:*

```
#include <pthread.h>
#define NUM_THREADS 10
void *work(void *i){
  int f = *((int *)(i));
  printf("Hello, world from %i with value %i\n",
            pthread_self(), f);
  pthread_exit(NULL);
}
int main(int argc, char **argv){
  int i;
  pthread_t id[NUM_THREADS];
  for(i = 0; i < NUM_THREADS; ++i){
    if(pthread_create(&id[i], NULL, work, (void *)(&i))){
      printf("Error creating the thread\n"); exit(19);}
  }
 return 0;}
```

```
Hello, world from 2 with value 1
Hello, world from 3 with value 2
Hello, world from 6 with value 5
Hello, world from 5 with value 5
Hello, world from 4 with value 4
Hello, world from 8 with value 9
Hello, world from 9 with value 9
Hello, world from 10 with value 9
Hello, world from 7 with value 6
Hello, world from 11 with value 10
```

Wrong Method!!!!

# POSIX Threads

***Example 2b:***

```
#include <pthread.h>
#define NUM_THREADS 10
void *work(void *i){
        int f = (int)(i);
        printf("Hello, world from %i with value %i\n",
                pthread_self(), f);
        pthread_exit(NULL);
}
int main(int argc, char **argv){
        int i;
        pthread_t id[NUM_THREADS];
        for(i = 0; i < NUM_THREADS; ++i){
                if(pthread_create(&id[i], NULL, work, (void *)(i))){
                        printf("Error creating the thread\n");
                        exit(19);
                }
        }
        return 0;
}
```

```
Hello, world from 2 with value 0
Hello, world from 3 with value 1
Hello, world from 4 with value 2
Hello, world from 5 with value 3
Hello, world from 6 with value 4
Hello, world from 7 with value 5
Hello, world from 8 with value 6
Hello, world from 10 with value 8
Hello, world from 11 with value 9
```

Right Method 1

# POSIX Threads

***Example 2c:***

```
#include <pthread.h>
#define NUM_THREADS 10
void *work(void *i){
        int f = *((int *)(i));
        printf("Hello, world from %i with value %i\n",
                pthread_self(), f);
        pthread_exit(NULL);}
int main(int argc, char **argv){
        int i;
        int y[NUM_THREADS];
        pthread_t id[NUM_THREADS];
        for(i = 0; i < NUM_THREADS; ++i){
                y[i] = i;
                if(pthread_create(&id[i], NULL, work, (void *)(&y[i]))){
                        printf("Error creating the thread\n");
                        exit(19);
                }
        }
        return 0;}
```

```
Hello, world from 2 with value 0
Hello, world from 4 with value 2
Hello, world from 5 with value 3
Hello, world from 6 with value 4
Hello, world from 7 with value 5
Hello, world from 8 with value 6
Hello, world from 9 with value 7
Hello, world from 3 with value 1
Hello, world from 10 with value 8
Hello, world from 11 with value 9
```

Right Method 2

# POSIX Threads

## *The Joining of All Loose Ends: pthread_join*

*int* **pthread_join**(**pthread_t** *id*, **void** ***tr*);

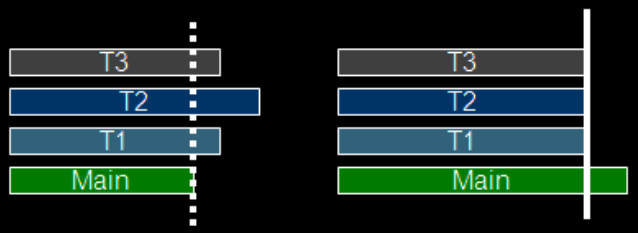
Make sure that the thread that has this *id* returns. Otherwise waits for it

| | |
|---|---|
| pthread_t id | The id of a created thread |
| void **tr | A pointer to the result of the thread |

OpenMP Counterpart

#pragma omp barrier

Returns a non zero value in success




Premature thread death

Join point

**Why use it?** *If the main thread dies, then all other threads will die with it. Even if they have not completed their work*

# POSIX Threads

***Example 3:***

```
#include <pthread.h>
#define NUM_THREADS 4
void *work(void *i){
        printf("Hello, world from %i\n", pthread_self());
        pthread_exit(NULL);
}
int main(int argc, char **argv){
        int i;
        pthread_t id[NUM_THREADS];
        for(i = 0; i < NUM_THREADS; ++i){
                if(pthread_create(&id[i], NULL, work, NULL)){
                        exit(19);
                }
        }
        printf("After creating the thread. My id is: %i\n", pthread_self());
        for(i = 0; i < NUM_THREADS; ++i){
                if(pthread_join(id[i], NULL)){
                        exit(19);
                }
        }
        printf("After joining\n");
        return 0;
}
```

```
Hello, world from 2
Hello, world from 3
Hello, world from 4
After creating the thread.
My id is: 1
Hello, world from 5
After joining
```

# POSIX Threads

## *Attributes Revisited: Useful Functions (1)*

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

Initialize an attribute with the default values for the attribute object

- **Default Schedule**: SCHED_OTHER **(?)**
- **Default Scope:** PTHREAD_SCOPE_SYSTEM **(?)**
- **Default Join State:** PTHREAD_CREATE_JOINABLE **(?)**

*int* **pthread_attr_destroy**(**pthread_attr_t** *attr);

De-allocate any memory and state that the attribute object occupied. It is safe to delete the attribute object after the thread has been created

*int* **pthread_attr_setdetachstate**(**pthread_attr_t** **attr*, **int** *JOIN_STATE*);

Set the attached parameter on the attribute object with the *JOIN_STATE* variable

- **PTHREAD_CREATE_JOINABLE**: It can be joined at a join point. State must be saved after function ends
- **PTHREAD_CREATE_DETACHED**: It cannot be joined at a join point. State and resources are de-allocated immediately

# POSIX Threads

## *Attributes Revisited: Useful Functions (2)*

*int* **pthread_attr_setschedpolicy**(**pthread_attr_t** **attr*, **int** *policy*)

Set the scheduling policy of the thread:

- **SCHED_OTHER** → Regular scheduling
- SCHED_RR → Round-robin (**SU**)
- SCHED_FIFO → First-in First-out (**SU**)

*int* **pthread_attr_setschedparam**(**pthread_attr_t** **attr*, **const struct sched_param** **pr*)

Contains the schedule priority of the thread

**Default: 0**

*int* **pthread_attr_setinheritsched**(**pthread_attr_t** **attr*, **int** *inherit*)

Tell if the scheduling parameters will be inherit from the parent or the ones in the attribute object will be used

PTHREAD_EXPLICIT_SCHED → Scheduling parameters from the attribute object will be used.

**PTHREAD_INHERIT_SCHED** → inherit the attributes from its parent.

*int* **pthread_attr_setscope**(**pthread_attr_t** **attr*, **int** *scope*)

Contention parameter

- **PTHREAD_SCOPE_SYSTEM**
- PTHREAD_SCOPE_PROCESS

# POSIX Threads

```
#include <pthread.h>
#define NUM_THREADS 4
struct args{int a; float b; char c;};
void *work(void *i){
        struct args *a = (struct args *)(i);
        printf("(%3i, %.3f, %3c) --> %i\n", a->a, a->b, a->c, pthread_self());
        pthread_exit(NULL);
}
int main(int argc, char **argv){
        int i;
        struct args a[NUM_THREADS];
        pthread_t id[NUM_THREADS];
        pthread_attr_t ma;
        pthread_attr_init(&ma);
        pthread_attr_setdetachstate(&ma, PTHREAD_CREATE_JOINABLE);
        for(i = 0; i < NUM_THREADS; ++i){
                a[i].a = i; a[i].b = 1.0 /(i+1); a[i].c = 'a' + (char)(i);
                pthread_create(&id[i], &ma, work, (void *)(&a[i]));
        }
        pthread_attr_destroy(&ma);
        for(i = 0; i < NUM_THREADS; ++i){pthread_join(id[i], NULL);}
        return 0;}
```

*Example 4:*

```
( 0, 1.000,  a) --> 2
( 3, 0.250,  d) --> 5
( 2, 0.333,  c) --> 4
( 1, 0.500,  b) --> 3
```

# POSIX Threads

## *Synchronization Types*

| | | |
|---|---|---|
| pthreads | Mutex | Mutual Exclusion Lock. Only the thread that has the lock can access the protected region |
| | Semaphores | |
| Java | Monitors | Act as a guard of some resource. Consists of a mutex with some kind of notification scheme |
| pthreads | Conditional Variables | Synchronization occurs when a condition is met. Always used in conjunction with a mutex. Inter-thread (process) communication. |
| pthreads | Reader / Writer Locks | Permits only reads on a data or writes on data in a group. In other words, lock out the writers when the readers are on the shared data and vice versa. |

# POSIX Threads

## *Synchronization (1): Mutex*

| Pthread | OpenMP |
| --- | --- |
| *int* **pthread_mutex_init**(**pthread_mutex_t** **m*, **const pthread_mutexattr_t** **ma*); | *int* **omp_lock_init**(**omp_lock_t** **lkc*); |
| *int* **pthread_mutex_lock**(**pthread_mutex_t** *m); | *int* **omp_lock_set**(**omp_lock_t** **lkc*); |
| *int* **pthread_mutex_unlock**(**pthread_mutex_t** **m*); | *int* **omp_lock_unset**(**omp_lock_t** **lkc*); |
| *int* **pthread_mutex_destroy**(**pthread_mutex_t** **m*); | *int* **omp_lock_destroy**(**omp_lock_t** **lkc*); |

1→ Initialization  2 → Setting  3 → Unsetting  4 → Destroying

**pthread_mutexattr_t** * **ma** can be left NULL for the default values

# POSIX Threads

*Example 5*

```
#include <pthread.h>
#define NUM_THREADS 2
#define CYCLE 100
double b;
void *deposit(void *i){ double m = *(double *)(i); int j;
        for(j = 0; j < CYCLE; ++j){ b += m; sleep(1); }}
void *withdraw(void *i){double m = *(double *)(i); int j;
        for(j = 0; j < CYCLE; ++j){ b -= m; sleep(1); }}
int main(int argc, char **argv){
        int i; double bi; double q = 10.0;  pthread_t id[NUM_THREADS];
        b = 100000; bi = b;
        pthread_create(&id[0], NULL, deposit , (void *)(&q));
        pthread_create(&id[1], NULL, withdraw, (void *)(&q));
        for(i = 0; i < NUM_THREADS; ++i){pthread_join(id[i], NULL);}
        printf("The Initial Balance: %.2lf\nThe   Final Balance: %.2lf\n",
bi, b);
        return 0; }
```

The Initial Balance: 100000.00
The Final Balance: 100010.00

# POSIX Threads

## *Example 5*

The Initial Balance: 100000.00
The Final Balance: 100000.00

```
...
pthread_mutex_t mt;
void *deposit(void *i){
        double m = *(double *)(i);
        int j;
        for(j = 0; j < CYCLE; ++j){
                pthread_mutex_lock(&mt); b += m; pthread_mutex_unlock(&mt);
sleep(1); }
}
void *withdraw(void *i){
        double m = *(double *)(i);
        int j;
        for(j = 0; j < CYCLE; ++j){
                pthread_mutex_lock(&mt); b -= m; pthread_mutex_unlock(&mt);
sleep(1); }
}
int main(int argc, char **argv){
        ...
        pthread_mutex_init(&mt, NULL);
        ...
        pthread_mutex_destroy(&mt);
        return 0;
}
```

# POSIX Threads

## *Synchronization (2): Conditional Variables*

*int* **pthread_cond_init**(**pthread_cond_t** **cond*, **pthread_condattr_t** **attr*)

Initialize a conditional variable *cond* with the attributes given by *attr*. The *attr* can be left NULL so it will use the default variable

*int* **pthread_cond_destroy**(**pthread_cond_t** **cond*)

De-allocated any resources associated with the conditional variable *cond*

*int* **pthread_condattr_init** (**pthread_condattr_t** **attr*)

Initialize the conditional attribute variable *attr* with the default value.

*int* **pthread_condattr_destroy** (**pthread_condattr_t** **attr*)

De-allocate the conditional attribute variable *attr*. It is safe to de-allocate the variable just after the conditional variable has been initialized

# POSIX Threads

## *Synchronization (2): Conditional Variables*

### Tricky Conditional Variables

*int* **pthread_cond_wait**(**pthread_cond_t** **condition*, **pthread_mutex_t** **m*)

Make the calling thread wait for a signal in the conditional variable. Must be called when the associated mutex is locked and it will unlock it while the thread blocks. It will also unlock the mutex if the signal has been received.

*int* **pthread_cond_signal**(**pthread_cond_t** **condition*)

Signal a thread that has been blocked waiting for condition to become true. It must be called **after** its associated mutex has been **locked** and the mutex must be **unlocked** after the signal has been **issued**

*int* **pthread_cond_broadcast** (**pthread_cond** **condition*)

Similar to pthread_cond_signal but it signals all the waiting threads for this conditional variable

# POSIX Threads

## *Example 6*

```
#include <pthread.h>
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6
pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;
int  count = 0;
```

Statically initialize the mutexes and the conditional variables

# POSIX Threads

```
void *f1(){
  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 %i (f1): %d\n",pthread_self(), count);
    pthread_mutex_unlock( &count_mutex );
    if(count >= COUNT_DONE) pthread_exit(NULL);
  }
}
```

**First function:** If the count variable is between COUNT_HALT1 and COUNT_HALT2, wait for a signal from f2. Otherwise increment count

# POSIX Threads

```
void *f2(){
   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 %i (f2): %d\n",pthread_self(), count);
      pthread_mutex_unlock( &count_mutex );
      if(count >= COUNT_DONE) pthread_exit(NULL);
   }
}
```

**Second function**: Send a signal to f1 if count is less than COUNT_HALT1 or greater than COUNT_HALT2. Later, increment the count

# POSIX Threads

```
Counter value 2 (f1): 1
Counter value 2 (f1): 2
Counter value 2 (f1): 3
Counter value 3 (f2): 4
Counter value 3 (f2): 5
Counter value 3 (f2): 6
Counter value 3 (f2): 7
Counter value 3 (f2): 8
Counter value 3 (f2): 9
Counter value 3 (f2): 10
Counter value 2 (f1): 11
```

```
int main(int argc, char **argv)
{
  pthread_t thread1, thread2;
  pthread_create( &thread1, NULL, &f1, NULL);
  pthread_create( &thread2, NULL, &f2, NULL);
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  return(0);
}
```

F1 is effectively halted between COUNT_HALT1 and COUNT_HALT2. Otherwise random

*Example taken from YoLinux.com. Copyright of Greg Ippolito*

# POSIX Threads

## *Miscellaneous Useful Functions*

*int* **pthread_attr_getstackaddr** (**const pthread_attr_t** **attr*, **void** ***stackaddr*)

Return the stack address that this P-thread will be using

*int* **pthread_attr_getstacksize** (**const pthread_attr_t** **attr*, **size_t** **stacksize*)

Return the stack size that this P-thread will be using

*int* **pthread_detach** (**pthread_t** *thr*, **void** ***value_ptr*)

Make the thread that is identified by *thr* not joinable

*int* **pthread_once** (**pthread_once_t** **once_control*, **void (****init_routine***)(void)**);

Make sure that the init_routine is executed by a single thread and only once. The once_control is a synchronization mechanism that can be defined as:
**pthread_once_t** *once_control* = **PTHREAD_ONCE_INIT**;

OpenMP Counterpart

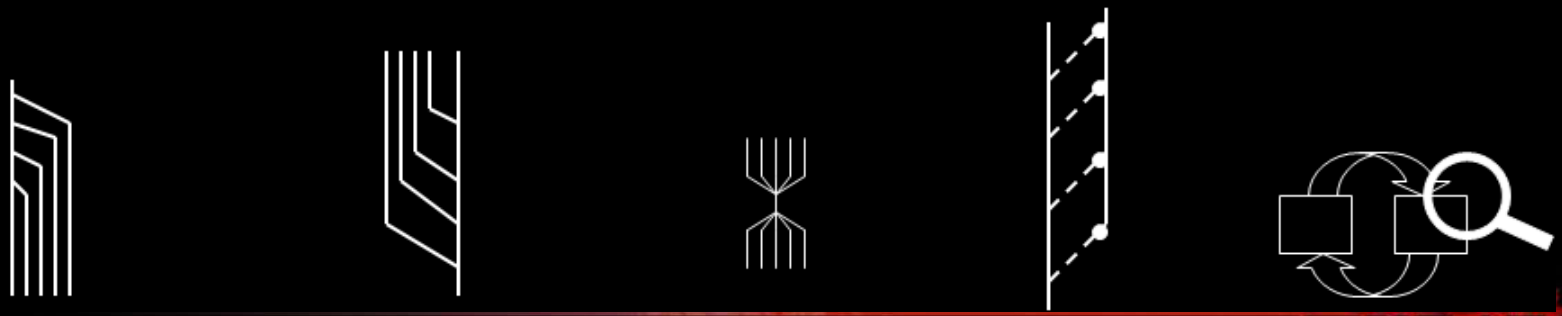
#pragma omp single

*void* **pthread_yield** ()

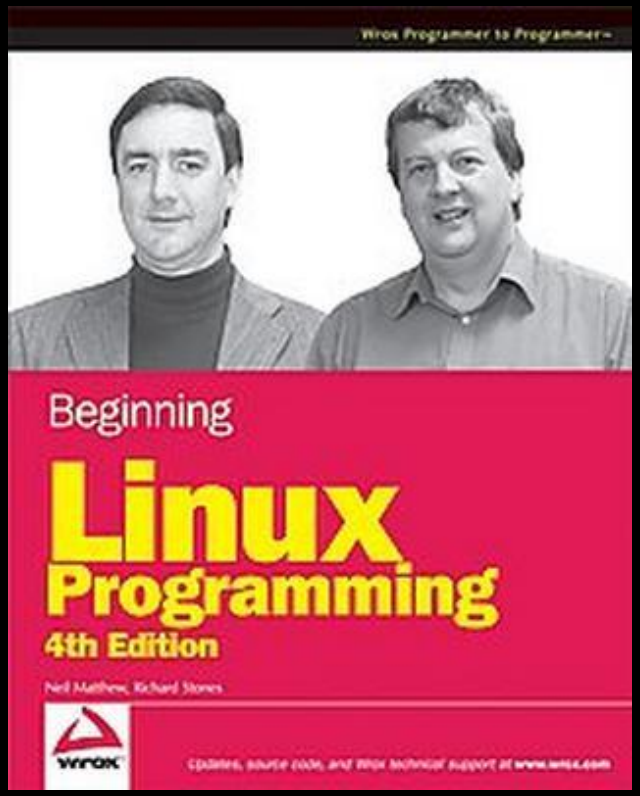
Relinquish the use of the processor

# Resources

- LLNL Pthread Tutorial: http://www.llnl.gov/computing/tutorials/pthreads/
- The YoLinux Pthread tutorial: http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#SYNCHRONIZATION
- Linux Parallel Processing HOWTO by Hank Dietz: http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html#toc2



# Reference Books




- "Beginning Linux Programming", 4th Edition, Neil Mathew, Richard Stones, Wrox Publication.
- Rating: