# Linux Process Synchronization

#### In The Kernel

- Simplest unit is atomic integers
- All operations on atomic integers are atomic
- Particularly efficient in situations where an integer variable—such as a counter—needs to be updated

Atomic Operation	Effect
atomic_set(&counter,5);	counter = 5
atomic_add(10,&counter);	counter = counter + 10
atomic_sub(4,&counter);	counter = counter - 4
atomic_inc(&counter);	counter = counter + 1
<pre>value = atomic_read(&amp;counter);</pre>	value = 12

#### In The Kernel

- Mutex locks are available in Linux for protecting critical sections within the kernel
- Linux also provides spinlocks and semaphores for locking in the kernel
- On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations
- On single-processor machines, spinlocks are inappropriate. So, we use -

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

#### In The Kernel

- It provides two simple system calls— preempt disable() and preempt enable()
   —for disabling and enabling kernel preemption
- The kernel is not preemptible, however, if a task running in the kernel is holding a lock. In this case -
  - Implementation each task in the system has a thread-info structure containing a counter,
     preempt count , to indicate the number of locks being held by the task.
  - When a lock is acquired, preempt count is incremented. It is decremented when a lock is released
  - Can only be safely preempted if preempt count is zero

# POSIX Thread Synchronization API

- Creating PThread
- Joining PThread
- Mutex
- Semaphore
- Condition Variables

## POSIX Thread Implementation

 Next Slides have been taken from https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

## **Creating PThreads**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print message function( void *ptr );
main()
     pthread t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int iret1, iret2;
    /* Create independent threads each of which will execute function */
     iret1 = pthread create( &thread1, NULL, print message function, (void*) message1);
     iret2 = pthread create( &thread2, NULL, print message function, (void*) message2);
```

## Joining PThread

```
/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */
pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
```

#### **Start Routine**

```
printf("Thread 1 returns: %d\n",iret1);
  printf("Thread 2 returns: %d\n",iret2);
  exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

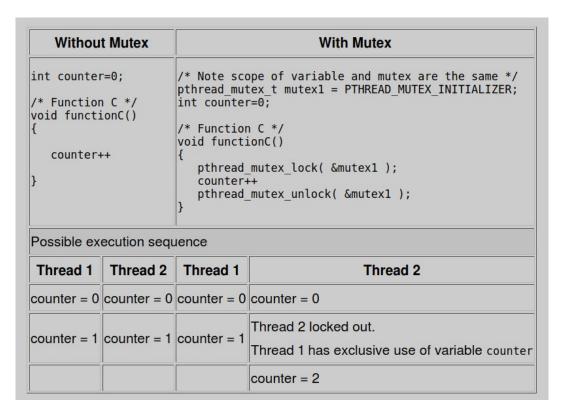
#### The pthread\_create Function

- thread returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr Set to NULL if default thread attributes are used. (else define members
  of the struct pthread attr t defined in bits/pthreadtypes.h)
- void \* (\*start\_routine) pointer to the function to be threaded. Function has a single argument: pointer to void.
- \*arg pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

## Explanation

- thread returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr Set to NULL if default thread attributes are used. (else define members
  of the struct pthread\_attr\_t defined in bits/pthreadtypes.h)
- void \* (\*start\_routine) pointer to the function to be threaded. Function has a single argument: pointer to void.
- \*arg pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

#### **Mutex Review**



## Mutex Implementation

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

## Mutex Implementation

```
void *functionC();
pthread mutex t mutex1 = PTHREAD MUTEX INITIALIZER;
int counter = 0;
main()
   int rcl, rc2;
   pthread t thread1, thread2;
   /* Create independent threads each of which will execute functionC */
   if( (rc1=pthread create( &thread1, NULL, &functionC, NULL)) )
      printf("Thread creation failed: %d\n", rc1);
   if( (rc2=pthread create( &thread2, NULL, &functionC, NULL)) )
      printf("Thread creation failed: %d\n", rc2);
   /* Wait till threads are complete before main continues. Unless we
  /* wait we run the risk of executing an exit which will terminate
   /* the process and all threads before the threads have completed.
   pthread join( thread1, NULL);
   pthread join( thread2, NULL);
   exit(0);
```