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`
`value = atomic_read(&counter);`	`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

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

- 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

Without Mutex	With Mutex

`int counter=0;`
 	
`/* Function C */`
`void functionC()`
`{`
 	
`  counter++`
 	
`}`
	`/* Note scope of variable and mutex are the same */`
	`pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;`
	`int counter=0;`
	
	`/* Function C */`
	`void functionC()`
	`{`
	`  pthread_mutex_lock( &mutex1 );`
	`  counter++`
	`  pthread_mutex_unlock( &mutex1 );`
	`}`

Possible execution sequence

Thread 1	Thread 2	Thread 1	Thread 2
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable `counter`
			counter = 2

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 rc1, 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);
}
```

Condition Variables

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}
```

Condition Variables

```
void *functionCount1()
{
   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 functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

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

Condition Variables

```
void *functionCount2()
{
    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 functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

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

}
```