Review

- What is Pthreads
- The Thread ID
- The Thread Creation
- The thread Termination
- The pthread_join() function
- Mutex
- The pthread_cancel function
- The pthread_cleanup_push() function
- The pthread_cleanup_pop() function

Preview

- The **pthread_detach()**
- Condition Variable
  - pthread_cond_destroy()
  - pthread_cond_init()
  - pthread_cond_timedwait()
  - pthread_cond_wait()
  - pthread_cond_broadcast()
  - pthread_cond_signal

The **pthread_detach()**

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

- The **pthread_detach**() function indicates that system resources for the specified *thread* should be **reclaimed when the thread ends.**
- If the thread is already ended, resources are reclaimed immediately.
- If *thread* does not represent a valid undetached thread, **pthread_detach**() will return ESRCH

Functions for a process vs. a thread

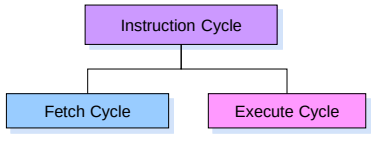
Process	Thread
fork()	pthread_create()
exit()	pthread_exit()
waitip()	pthread_join()
getpid()	ptherad_self()
abort()	pthread_cancel()

Thread Synchronization

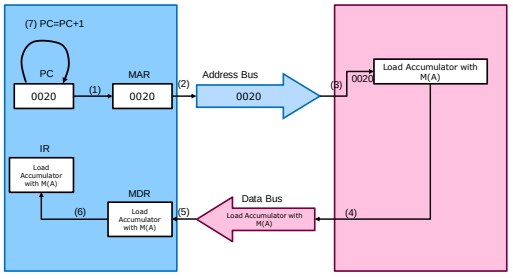
- If multiple threads uses variables that each thread can read and write, we need to synchronize the threads to ensure that they don't see and invalid value.
- In one processor system, the modification of a variable takes more than one memory patch cycle.

Thread Synchronization

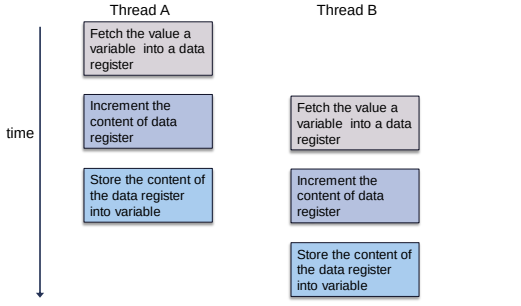
- Instruction cycle in the one processor system



Thread Synchronization



Thread Synchronization



Thread Synchronization

- We can protect our data and ensure access by only one thread at a time by using mutex.
- The mutex variable is represented by the **phread_mutex_t** data type.
- Before use **phread_mutex_t** data type, we need initiate by
  - Calling a function pthread_mutex_init()
  - Initiated by PTHREAD_MUTEX_INITIALIZER

Thread Synchronization

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```


Condition Variable

- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. (block state)
- A condition variable is a variable of type **pthread_cond_t**.
- The condition itself need to be protected by **mutex**.
- A thread must first lock the **mutex** to change the condition state.
- Other threads will not notice the change until they acquire the **mutex**.

Condition Variable

- Creating/Destroying a condition variable

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);

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

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

•The pthread_cond_destroy() function shall destroy the given condition variable specified by cond;
•The pthread_cond_init function initiate a condition variable pointed by cont with an attribute pointed by att

Condition Variable

Restricted type pointer

- More than one pointer can access the same chunk of memory and modify it during the execution of a program.
- **The restrict type qualifier** is an indication to the compiler that, if the memory addressed by the restrict - qualified pointer is modified, no other pointer will access that same memory.

Condition Variable

Waiting on condition

```
#include <pthread.h>
int pthread_cond_timedwait (pthread_cond_t *restrict cond,
   pthread_mutex_t *restrict mutex,
   const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
   pthread_mutex_t *restrict mutex);
```

- Both functions shall block on a condition variable. They shall be called with mutex locked by the calling thread or undefined behavior results.
- These functions atomatically release mutex and cause the calling thread to block on the condition variable cond
- Both function shall be equivalent except that an error is returned if the absolute time specified by abstime passes

Condition Variable

```
sturct timespac {
   time_t tv_sec /* seconds*/
   long tv_nsec  /* nanoseconds  */
};
```


Condition Variable

- Waking thread based on condition:

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

- The pthread_cond_broadcast() function shall unblock all threads currently blocked on the specified condition variable cond.
- The pthread_cond_signal() function shall unblock at least one of the threads that are blocked on the specified condition variable cond


```
/*************************************************************
   condition.c: demonstrate condition variable with functions
   pthread_cond_wait(), pthread_cond_signal()
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* initiate mutex and condition variable */
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(); /* a function for a thread */
void *functionCount2(); /* a function for a thread */
void err_sys(char *, int); /* error function */
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

int main()
{
   int rc;
   pthread_t thread1, thread2;


   /* Create two thread */
   if ((rc=pthread_create( &thread1, NULL, &functionCount1, NULL)) !=0)
            err_sys("ERROR; return code from pthread_create() is", rc);
   if ((rc=pthread_create( &thread2, NULL, &functionCount2, NULL)) !=0)
            err_sys("ERROR; return code from pthread_create() is", rc);

   if ((rc =pthread_join( thread1, NULL)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);
   if ((rc =pthread_join( thread2, NULL)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);

   exit(0);
}
```



```
void *functionCount1()
{
   for(;;)
   {  /*lock the mutex for a conditional variable */
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
      {
         /* waiting for a signal*/
         pthread_cond_wait( &condition_cond, &condition_mutex );
      }
      pthread_mutex_unlock( &condition_mutex );
      /*lock the mutex for shared variable count */
      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

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

void *functionCount2()
{
   for(;;)
   {  /*lock the mutex for a conditional variable */
      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);
   }
}
```