Preview

- 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

What are Pthreads?

- Hardware vendors had implemented their own proprietary versions of threads.
- For a requirement of standardized programming interface, this interface has been specified by the IEEE POSIX 1003.1c standard (1995) (POSIX thread).
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a header file <pthread.h>

The Thread ID

- A Process ID (**pid_t**) for a process is unique in the system.
- But a thread ID (**pthread_t**) has significance only within the context of the process where it belongs.
- Even though unsigned long (Linux), unsigned integer (Solaries 9) is used represent a pthread_t, a function **ptread_equal**() must be used to compare thread ID.

The Thread ID

```
#include <phread.h>

/*returns non-zero if equal, return 0 otherwise */
int pthread_equal(pthread_t t1, pthread_t t2);
```

```
#include <phread.h>
/*returns thread ID of the calling thread */
pthread_t pthread_self(void);
```

The thread Creation

- Initially, your main() program comprises a single, default thread.
- All other threads must be explicitly created by the programmer
- **pthread_create()** function creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- Once created, threads are peers, and may create other threads. **There is no implied hierarchy or dependency between threads**

The thread Creation

```
#include <pthread.h>
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *att,
                void *(*start_routine)(void*), void *arg);
```

- **att** point to structure of pthread atribute. If att is NULL, a default attribute will be used
- **start_routing** point to address of a void function with no parameter or
- We can save parameters to typeless pointer arg and be able to pass to the function.
- When multiple threads are created, there is **no guarantee which runs first**. It is depends on the thread scheduler.

The thread Creation

- On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.
  - **EAGAIN** Insufficient resources to create another thread, or a system-imposed limit on the number of threads was encountered.
  - **EINVAL** Invalid settings in *attr*.
  - **EPERM** No permission to set the scheduling policy and parameters specified in *attr*.

The thread Creation

- pthread_attr_t
  - int flags
  - int stacksize
  - int contentionscope
  - int inheritsched
  - int detachstate
  - int sched
  - struct sched_param param
  - struct timespec starttime deadline period

Compile with pthread()

- To compile a program with pthread, we need provide pthread library to linker.

Ex)

Program file Name: example.c

Executable file name: example

Step 1) create object cord example.o

gcc –c example.c

Step 2) create an executable code example

gcc –pthread –o example example.o

```
/***********************************************
  createTh.c: Demonstrate creation of threads
***********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t<NUM_THREADS;t++)
   {
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc)
      {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
      }
   }
   pthread_exit(NULL);
   exit (0);
}
```

```
 /*********************************************
  createTh2.c: Demonstrate creation of threads
*********************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS  5
static pthread_t getCurrentThreadID()
{
   return pthread_self();
}
// the function pointed by each thread by passing different parameter
void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   pthread_t id = getCurrentThreadID(); //get a thread id
   printf("Hello World! It's me, thread #%d! with thread id %u\n", tid, id);
   pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   //create five threads pointing same function with different argument
   for(t=0;t<NUM_THREADS;t++)
   {
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc!=0)
      {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(1);
      }
   }
   pthread_exit(NULL);
   exit (0);
}
```

```
/*******************************************************************
  threadrace.c: Demonstrate race condition with multiple threads
*******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

int count =0;
void *printHi(void *); /* function prototype for threads function*/

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t<NUM_THREADS;t++) /* create five threads */
   {
      printf("In main: creating thread %d\n", t);
         rc = pthread_create(&threads[t], NULL, printHi, (void *)t);
      if (rc!=0)
      {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(1);
      }
   }
   pthread_exit(NULL);
   exit (0);
}
```

```
void *printHi(void *threadid)
{
   int tid, i;

   tid = (int)threadid;

   if (tid % 2 == 0)
   {
        for (i =0; i<10; i++)
          {
              sleep (1);
              count++;
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);

          }
   }
   else
   {
          for (i =0; i<10; i++)
          {
              count++;
              sleep(1);
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);
          }
    }

   pthread_exit(NULL);
}
```

The thread Termination

- If any thread within a process call exit or _exit system call, then the entire process terminate.
- A single thread inside a process can be terminated by three ways.
  - The thread can simply return from the start routine
  - The thread can be cancelled by another thread by calling **pthread_cancel**() function in the same process.
  - The thread can call **phread_exit()** with some condition

The thread Termination

```
#include <pthread>
void pthread_exit(void *rval_ptr);
```

- The rval_ptr: typeless pointer. This pointer is available to other threads in the process by calling the pthread_join() system call.

Thread Synchronization

- Joining" is one way to accomplish synchronization between threads.
- There are three methods to synchronization between threads
  - Mutexes
  - Joining
  - Condition variables

Mutexes

- A **Mutex** is a variable that can be in one of two state: unlock (0), lock(1).
- Modification to the value of a **Mutex** in the unlock and lock operations are executed indivisibly.
  - Lock operation check mutex value and if mutex =1, set mutex=0; if mutex = 0; sleep on the mutex queue
  - Unlock operation set mutex =1;
- **Mutexes** are used to prevent data inconsistencies due to race conditions
- **Mutexes** are used for serializing shared resources.
- If a global resource is accessed by several thread, the global resource should have a **Mutex** associated with it.

Mutexes

- Scenario to avoid race condition without mutex (using variable lock)
  - Each thread need check lock variable before entering a critical region.
  - If lock =0 then a thread can enter to a critical region.
  - If lock = 1 then other thread cannot enter the critical region.
  - Once a thread finish its job in the critical region, set lock = 0 and let other thread enter the critical region.

Mutexes

```
repeat
   while lock ≠ 0 do
         ; (no-operation)
      lock = 1
        Critical Section

     lock = 0;
        Remainder Section

   until false
```

Scenario)

1. Initially lock = 0.
2. A process (or thread) P_1 tries get into critical section. A process check lock value = 0.
3. Process P_1 CPU time is over and go to ready state, before updating lock = 1.
4. Process P_2 tries get into critical section. P_2 check lock value lock = 0
5. P_2 set lock = 1 and go to critical section.
6. P_2 CPU time is over and P_1 is rescheduled.
7. P_1 already read lock = 0, P_1 set lock = 1 and go to Critical section. Now P_1 and P_2 are in the critical section at the same time

Mutexes

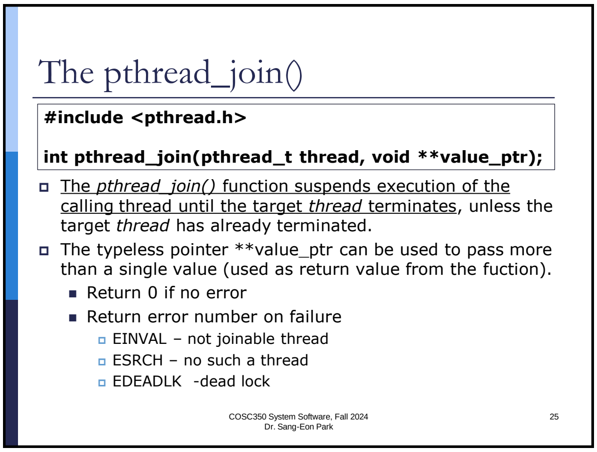
```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER
repeat
   pthread_mutex_lock( &mutex1 );
        Critical Section

   pthread_mutex_unlock( &mutex1 );

        Remainder Section

   until false
```

```c
/*******************************************************************
  threadrace.c: Demonstrate race condition with multiple threads
*******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

int count =0; /* global variable shared by multiple threads*/
void *printHi(void *); /* function prototype for threads function*/

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t<NUM_THREADS;t++) /* create five threads */
   {
      printf("In main: creating thread %d\n", t);
          rc = pthread_create(&threads[t], NULL, printHi, (void *)t);
      if (rc!=0)
      {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(1);
      }
   }
   pthread_exit(NULL);
   exit (0);
}
```

```c
void *printHi(void *threadid)
{
   int tid, i;

   tid = (int)threadid;

   if (tid % 2 == 0) /* for a thread with even number*/
   {
        for (i =0; i<10; i++)
          {
              sleep (1);
              count++;
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);

          }
   }
   else /* for a thread with odd number*/
   {
          for (i =0; i<10; i++)
          {
              count++;
              sleep(1);
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);
          }
    }

   pthread_exit(NULL);
}
```

```c
/*******************************************************************
  threadrace1.c:
  Demonstrate to use mutex to avoid race condition between multiple threads
*******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   5
/* mutex for shared variable count is initiated*/
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
int count =0;  /* shared variable between threads */
void *printHi(void *);
int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t<NUM_THREADS;t++)
   {
      printf("In main: creating thread %d\n", t);
          rc = pthread_create(&threads[t], NULL, printHi, (void *)t);
      if (rc!=0)
      {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(1);
      }
   }
   pthread_exit(NULL);
   exit (0);
}
```

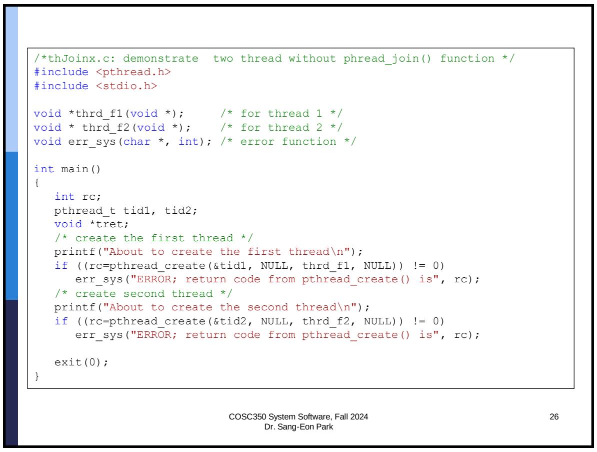
```c
void *printHi(void *threadid)
{
   int tid, i;

   tid = (int)threadid;

   if (tid % 2 == 0) /* for a thread with even number*/
   {

       for (i =0; i<10; i++)
       {
              pthread_mutex_lock(&count_mutex);
              sleep (1);
              count++;
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);
              pthread_mutex_unlock(&count_mutex);

       }
   }
   else /* for a thread with odd number*/
   {

        for (i =0; i<10; i++)
        {
              pthread_mutex_lock(&count_mutex);
              count++;
              sleep(1);
              printf("Hi!, thread #%d! shared variable count = %d\n", tid, count);
              pthread_mutex_unlock(&count_mutex);

        }
    }

   pthread_exit(NULL);
}
```



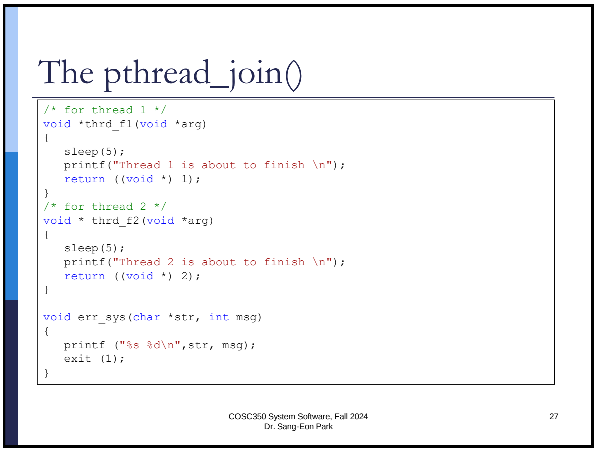
The pthread_join()

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

- The *pthread_join()* function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.
- The typeless pointer **value_ptr can be used to pass more than a single value (used as return value from the fuction).
  - Return 0 if no error
  - Return error number on failure
    - EINVAL – not joinable thread
    - ESRCH – no such a thread
    - EDEADLK -dead lock

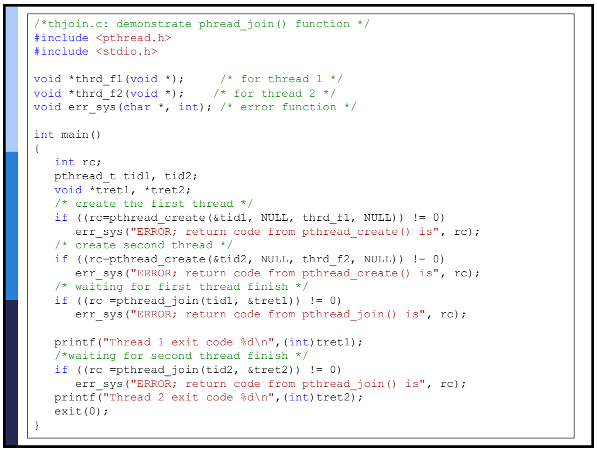


```
/*thJoinx.c: demonstrate  two thread without phread_join() function */
#include <pthread.h>
#include <stdio.h>

void *thrd_f1(void *);     /* for thread 1 */
void * thrd_f2(void *);    /* for thread 2 */
void err_sys(char *, int); /* error function */

int main()
{
   int rc;
   pthread_t tid1, tid2;
   void *tret;
   /* create the first thread */
   printf("About to create the first thread\n");
   if ((rc=pthread_create(&tid1, NULL, thrd_f1, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   /* create second thread */
   printf("About to create the second thread\n");
   if ((rc=pthread_create(&tid2, NULL, thrd_f2, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);

   exit(0);
}
```

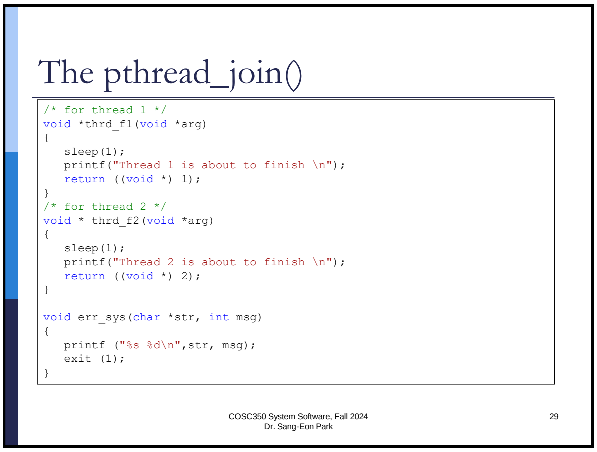


The pthread_join()

```
/* for thread 1 */
void *thrd_f1(void *arg)
{
   sleep(5);
   printf("Thread 1 is about to finish \n");
   return ((void *) 1);
}
/* for thread 2 */
void * thrd_f2(void *arg)
{
   sleep(5);
   printf("Thread 2 is about to finish \n");
   return ((void *) 2);
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
```

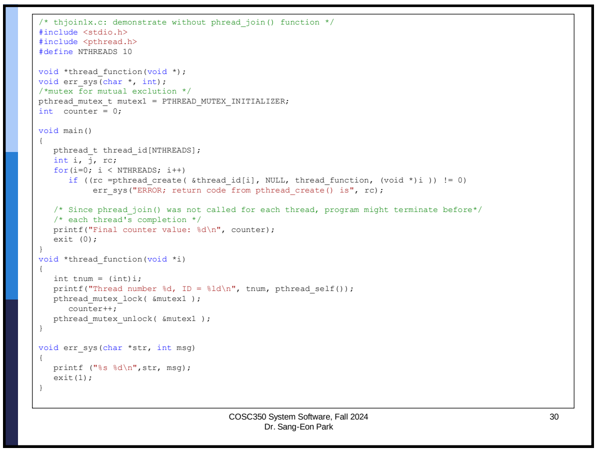


```
/*thjoin.c: demonstrate phread_join() function */
#include <pthread.h>
#include <stdio.h>

void *thrd_f1(void *);     /* for thread 1 */
void *thrd_f2(void *);    /* for thread 2 */
void err_sys(char *, int); /* error function */

int main()
{
   int rc;
   pthread_t tid1, tid2;
   void *tret1, *tret2;
   /* create the first thread */
   if ((rc=pthread_create(&tid1, NULL, thrd_f1, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   /* create second thread */
   if ((rc=pthread_create(&tid2, NULL, thrd_f2, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   /* waiting for first thread finish */
   if ((rc =pthread_join(tid1, &tret1)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);

   printf("Thread 1 exit code %d\n",(int)tret1);
   /*waiting for second thread finish */
   if ((rc =pthread_join(tid2, &tret2)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);
   printf("Thread 2 exit code %d\n",(int)tret2);
   exit(0);
}
```

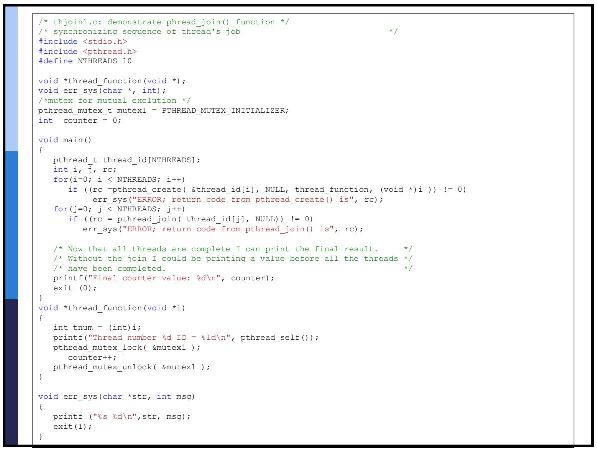


The pthread_join()

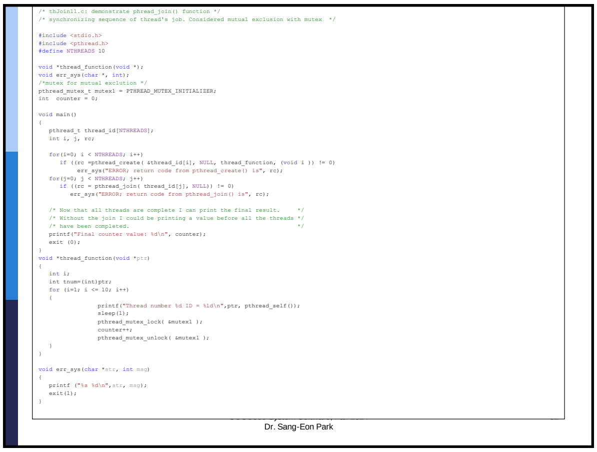
```
/* for thread 1 */
void *thrd_f1(void *arg)
{
   sleep(1);
   printf("Thread 1 is about to finish \n");
   return ((void *) 1);
}
/* for thread 2 */
void * thrd_f2(void *arg)
{
   sleep(1);
   printf("Thread 2 is about to finish \n");
   return ((void *) 2);
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
```



```
/* thjoin1x.c: demonstrate without phread_join() function */
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10

void *thread_function(void *);
void err_sys(char *, int);
/*mutex for mutual exclution */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;
   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *)i )) != 0)
           err_sys("ERROR; return code from pthread_create() is", rc);

   /* Since phread_join() was not called for each thread, program might terminate before*/
   /* each thread's completion */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *i)
{
   int tnum = (int)i;
   printf("Thread number %d, ID = %ld\n", tnum, pthread_self());
   pthread_mutex_lock( &mutex1 );
      counter++;
   pthread_mutex_unlock( &mutex1 );
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```

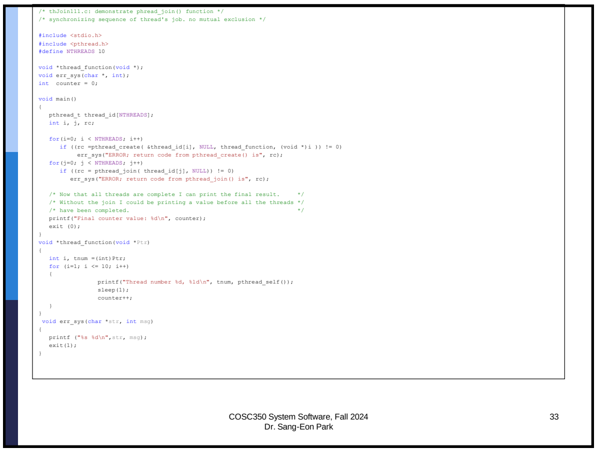


```c
/* thjoin1.c: demonstrate phread_join() function */
/* synchronizing sequence of thread's job                          */
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10

void *thread_function(void *);
void err_sys(char *, int);
/*mutex for mutual exclution */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;
   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *)i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);
   for(j=0; j < NTHREADS; j++)
      if ((rc = pthread_join( thread_id[j], NULL)) != 0)
        err_sys("ERROR; return code from pthread_join() is", rc);

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *i)
{
   int tnum = (int)i;
   printf("Thread number %d ID = %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
      counter++;
   pthread_mutex_unlock( &mutex1 );
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```



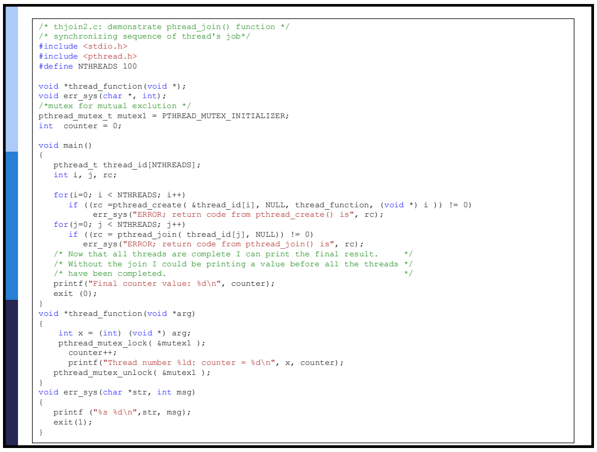
```c
/* thJoin11.c: demonstrate phread_join() function */
/* synchronizing sequence of thread's job. Considered mutual exclusion with mutex  */

#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10

void *thread_function(void *);
void err_sys(char *, int);
/*mutex for mutual exclution */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;

   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);
   for(j=0; j < NTHREADS; j++)
      if ((rc = pthread_join( thread_id[j], NULL)) != 0)
        err_sys("ERROR; return code from pthread_join() is", rc);

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *ptr)
{
   int i;
   int tnum=(int)ptr;
   for (i=1; i <= 10; i++)
   {
                printf("Thread number %d ID = %ld\n",ptr, pthread_self());
                sleep(1);
                pthread_mutex_lock( &mutex1 );
                counter++;
                pthread_mutex_unlock( &mutex1 );
   }
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```

Dr. Sang-Eon Park

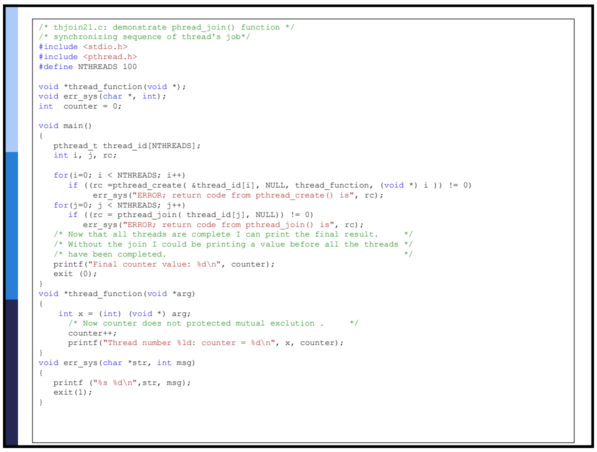


```c
/* thJoin111.c: demonstrate phread_join() function */
/* synchronizing sequence of thread's job. no mutual exclusion */

#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10

void *thread_function(void *);
void err_sys(char *, int);
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;

   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *)i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);
   for(j=0; j < NTHREADS; j++)
      if ((rc = pthread_join( thread_id[j], NULL)) != 0)
        err_sys("ERROR; return code from pthread_join() is", rc);

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *Ptr)
{
   int i, tnum =(int)Ptr;
   for (i=1; i <= 10; i++)
   {
                printf("Thread number %d, %ld\n", tnum, pthread_self());
                sleep(1);
                counter++;
   }
}
 void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```

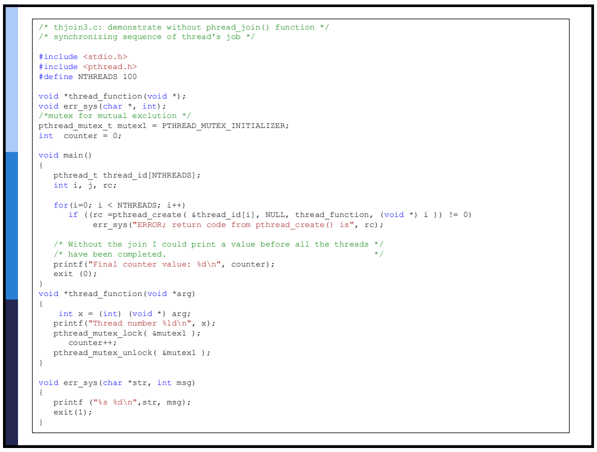
COSC350 System Software, Fall 2024 33
Dr. Sang-Eon Park



```c
/* thjoin2.c: demonstrate phread_join() function */
/* synchronizing sequence of thread's job*/
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 100

void *thread_function(void *);
void err_sys(char *, int);
/*mutex for mutual exclution */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;

   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *) i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);
   for(j=0; j < NTHREADS; j++)
      if ((rc = pthread_join( thread_id[j], NULL)) != 0)
        err_sys("ERROR; return code from pthread_join() is", rc);
   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *arg)
{
    int x = (int) (void *) arg;
    pthread_mutex_lock( &mutex1 );
      counter++;
      printf("Thread number %ld: counter = %d\n", x, counter);
   pthread_mutex_unlock( &mutex1 );
}
void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```



```c
/* thjoin21.c: demonstrate phread_join() function */
/* synchronizing sequence of thread's job*/
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 100

void *thread_function(void *);
void err_sys(char *, int);
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;

   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *) i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);
   for(j=0; j < NTHREADS; j++)
      if ((rc = pthread_join( thread_id[j], NULL)) != 0)
        err_sys("ERROR; return code from pthread_join() is", rc);
   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *arg)
{
    int x = (int) (void *) arg;
      /* Now counter does not protected mutual exclution .     */
      counter++;
      printf("Thread number %ld: counter = %d\n", x, counter);
}
void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```



```c
/* thjoin3.c: demonstrate without phread_join() function */
/* synchronizing sequence of thread's job */

#include <stdio.h>
#include <pthread.h>
#define NTHREADS 100

void *thread_function(void *);
void err_sys(char *, int);
/*mutex for mutual exclution */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void main()
{
   pthread_t thread_id[NTHREADS];
   int i, j, rc;

   for(i=0; i < NTHREADS; i++)
      if ((rc =pthread_create( &thread_id[i], NULL, thread_function, (void *) i )) != 0)
          err_sys("ERROR; return code from pthread_create() is", rc);

   /* Without the join I could print a value before all the threads */
   /* have been completed.                                          */
   printf("Final counter value: %d\n", counter);
   exit (0);
}
void *thread_function(void *arg)
{
    int x = (int) (void *) arg;
   printf("Thread number %ld\n", x);
   pthread_mutex_lock( &mutex1 );
      counter++;
   pthread_mutex_unlock( &mutex1 );
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit(1);
}
```

The pthread_cancel()

- The **pthread_cancel**() function requests cancellation of the target thread.
- The target thread is cancelled, based on it's ability to be cancelled.
- When cancel ability is deferred, all cancels are held pending in the target thread until the thread changes the cancel ability, calls a function that is a cancellation point.

The pthread_cancel()

- Cancellation points:
  - pthread_cond_timedwait()
  - pthread_cond_wait()
  - pthread_delay_np()
  - pthread_join()
  - pthread_join_np()
  - pthread_extendedjoin_np()
  - pthread_testcancel()

```
/* thcancel.c demonstrate pthread_cancel() function */
#include <pthread.h>
#include <stdio.h>

void *threadfunc(void *);
void err_sys(char *, int );

int main(int argc, char **argv)
{
   pthread_t tid;
   int rc=0;

   printf("Entering testcase\n");
   /* create a thread */
   if ((rc = pthread_create(&tid, NULL, threadfunc, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   sleep(2);
   printf("Now Canceling the thread\n");
   /* try to calcel the thread created */
   if ((rc = pthread_cancel(tid)) != 0)
      err_sys("ERROR; return code from pthread_cancel() is", rc);
   if ((rc = pthread_join( tid, NULL)) != 0)
         err_sys("ERROR; return code from pthread_join() is", rc);
   sleep(3);
   printf("Main completed\n");
   return 0;
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
void *threadfunc(void *parm)
{
   printf("Entered secondary thread\n");
   while (1)
  {
     printf("Secondary thread is looping\n");
     pthread_testcancel(); /* cancel point */
     sleep(1);
  }
}
```

The pthread_cleanup_push()

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
```

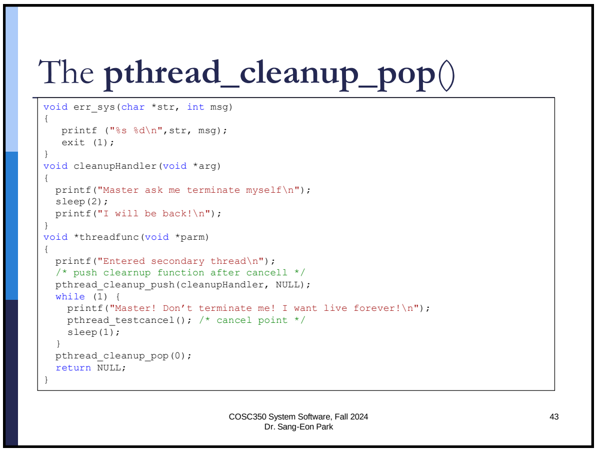
- The **pthread_cleanup_push**() function pushes a clean up function routine, to be called with the single argument, arg, when the thread performs one of the following actions.
  - pthread_exit()
  - pthread_cancel()

The pthread_cleanup_pop()

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

- The **pthread_cleanup_pop**() function pops the last cleanup handler from the cancellation cleanup stack.
- If the *execute* parameter is nonzero, the handler is called with the argument specified by the **pthread_cleanup_push**() call with which the handler was registered.

```
/*thpushpop.c: demonstrate
  pthread_cleanup_push() and pthread_cleanup_pop() */

#include <pthread.h>
#include <stdio.h>
void err_sys(char *, int );
void cleanupHandler(void *);
void *threadfunc(void *);

int main()
{
   pthread_t tid;
   int rc=0;

   printf("Entering testcase\n");
   /* now creating a thread */
   if ((rc = pthread_create(&tid, NULL, threadfunc, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   sleep(2);

   printf("Now Canceling the thread\n");
   /* now cancelling the created thread */
   if ((rc = pthread_cancel(tid)) != 0)
      err_sys("ERROR; return code from pthread_cancel() is", rc);
   sleep(3);
   if ((rc = pthread_join( tid, NULL)) != 0)
         err_sys("ERROR; return code from pthread_join() is", rc);
   printf("Main completed\n");
   return 0;
}
```



The **pthread_cleanup_pop()**

```c
void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
void cleanupHandler(void *arg)
{
  printf("Master ask me terminate myself\n");
  sleep(2);
  printf("I will be back!\n");
}
void *threadfunc(void *parm)
{
  printf("Entered secondary thread\n");
  /* push clearnup function after cancell */
  pthread_cleanup_push(cleanupHandler, NULL);
  while (1) {
    printf("Master! Don’t terminate me! I want live forever!\n");
    pthread_testcancel(); /* cancel point */
    sleep(1);
  }
  pthread_cleanup_pop(0);
  return NULL;
}
```