POLITECNICO
DI TORINO

Process Synchronization

Operating Systems – Sarah Azimi

POSIX Synchronization

- POSIX.1b standard was adopted in 1993
- Pthreads API is OS-independent

- It provides:

  - **Semaphores**
  - **Mutex Locks**
  - **Condition Variables**

Semaphore

- Semaphores are useful for process synchronization and multithreading.

- The POSIX system in Linux presents its own built-in semaphore library.

  - Including <semaphore.h>
  - Compile the code by linking with –pthread -lrt

Semaphore Operations

- Semaphore variable is of type `sem_t`.
- Semaphore functions start with `sem_`

```
#include <semaphore.h>

int sem_init (sem_t    *sem, int    pshared,  unsigned    int value);

int sem_destroy (sem_t   *sem);

int sem_wait (sem_t    *sem);

int sem_trywait (sem_t   *sem);

int sem_post (sem_t   *sem);

int sem_getvalue (sem_t   *sem, int   *sval);
```

Semaphore Operations

- Semaphore variable is of type `sem_t`.
- Semaphore functions start with `sem_`

```
#include <semaphore.h>

int sem_init (sem_t    *sem, int    pshared,  unsigned    int value);

int sem_destroy (sem_t   *sem);

int sem_wait (sem_t    *sem);

int sem_trywait (sem_t   *sem);

int sem_post (sem_t   *sem);

int sem_getvalue (sem_t   *sem, int   *sval);
```

All Semaphore functions:

- If successful, return 0.
- Otherwise, they return -1 and sets `errno` to indicate the error.

Semaphore Operations

- To initialize a semaphore

Syntax:

```
int sem_init (sem_t   *sem,   int  pshared,  unsigned int value);
```

- `sem` specifies the semaphore to be initialized.
  - Semaphore variable is of type `sem_t`
- `pshared` specifies whether or not the newly initialized semaphore is shared between processes or between threads.
  - A non-zero value means the semaphore is shared between processes.
  - A value of zero means the semaphore is shared between threads.
- `value` specifies the value to assign to the newly initialized semaphore.

Semaphore Operations

- To lock a semaphore or wait

Syntax:

```
int sem_wait (sem_t   *sem);
```

- `sem_wait` is a standard semaphore wait operation.
- If the semaphore value is 0, `sem_wait` blocks until it can successfully decrement value.

Semaphore Operations

- To lock a semaphore or wait

Syntax:

```
int sem_wait (sem_t   *sem);

int sem_trywait (sem_t   *sem);
```

- `sem_wait` is a standard semaphore wait operation.
- If the semaphore value is 0, `sem_wait` blocks until it can successfully decrement value.

- `sem_trywait` is similar to `sem_wait`, except that instead of blocking on 0, it returns -1 and sets `errno` to `EAGAIN`

Semaphore Example

```
main() {
  sem_t my_semaphore;
  int value, rc;

  sem_init(&my_semaphore, 0, 1);
  sem_getvalue(&my_semaphore, &value);

  printf("The initial value of the semaphore is %d\n", value);
  sem_wait(&my_semaphore);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the wait is %d\n", value);
  rc = sem_trywait(&my_semaphore);
  if ((rc == -1) && (errno == EAGAIN))
        printf("Return value: %d  - trywait did not decrement the semaphore\n",
rc);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after trywait is %d\n", value);

}
```

Output:

The initial value of the semaphore is 1
The value of the semaphore after the wait is 0
Return value: -1 - trywait did not decrement the semaphore
The value of the semaphore after the trywait is 0

Semaphore Operations

- To get the value of semaphore

Syntax:

```
int sem_getvalue (sem_t   *sem, int  *sval);
```

- Allows the user to examine the value of a semaphore.
- Sets the integer referenced by `sval` to the value of the semaphore.

Semaphore Operations

- To release or signal a semaphore

Syntax:

```
int sem_post (sem_t   *sem);
```

- `sem_post` increments the semaphore value and is the classical semaphore signal operation.

Semaphore Example

```
main() {
  sem_t my_semaphore;
  int value, rc;

  sem_init(&my_semaphore, 0, 1);
  sem_getvalue(&my_semaphore, &value);
  printf("The initial value of the semaphore is %d\n", value);
  sem_wait(&my_semaphore);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the wait is %d\n", value);
  sem_post(&my_semaphore);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the signal %d\n", value);
  sem_post(&my_semaphore);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the signal %d\n", value);
  rc = sem_trywait(&my_semaphore);
  if  (((rc == -1) && (errno == EAGAIN)) || (rc == 0) )
      printf("return value from trywait = %d\n", rc);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the trywait %d\n", value);
  sem_wait(&my_semaphore);
  sem_getvalue(&my_semaphore, &value);
  printf("The value of the semaphore after the wait %d\n", value);
}
```

Output:

```
The initial value of the semaphore is 1
The value of the semaphore after the wait is 0
The value of the semaphore after the signal 1
The value of the semaphore after the signal 2
return value from trywait = 0
The value of the semaphore after the trywait 1
The value of the semaphore after the wait 0
```

Semaphore Operations

- To destroy a semaphore

Syntax:

```
int sem_destroy (sem_t   *sem);
```

- If `sem_destroy` attempts to destroy a semaphore that is being used by another process, it may return -1.

Semaphore Example

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t S1;

int main()
{
    sem_init(&S1, 0, 1);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_destroy(& S1);
    return 0;
}
```

```
void* thread(void* arg)
{
    //wait
    sem_wait(& S1);
    printf("\nEntered..\n");

    //critical section
    sleep(4);
    //signal
    printf("\nJust Exiting...\n");
    sem_post(& S1);
}
```

Semaphore Example

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t S1;

int main()
{
    sem_init(&S1, 0, 1);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_destroy(& S1);
    return 0;
}
```

Defining Semaphore Variable

Initializing semaphore variable
(0 as the second parameter means shared between threads)

Destroying the semaphore

```c
void* thread(void* arg)
{
    //wait
    sem_wait(& S1);
    printf("\nEntered..\n");

    //critical section
    sleep(4);
    //signal
    printf("\nJust Exiting...\n");
    sem_post(& S1);
}
```

Semaphore Example

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t S1;

int main()
{
    sem_init(&S1, 0, 1);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_destroy(& S1);
    return 0;
}
```

Defining Semaphore Variable

Initializing semaphore variable (0 as the second parameter means shared between threads)

Destroying the semaphore

```
void* thread(void* arg)
{
    //wait
    sem_wait(& S1);
    printf("\nEntered..\n");

    //critical section
    sleep(4);
    //signal
    printf("\nJust Exiting...\n");
    sem_post(& S1);
}
```

Until the value of S1 is 0, sem_wait blocks the operation.
Then, it decrement the value

Sem-post it the semaphore signal operation which increment the semaphore value

Semaphore Example

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t S1;

int main()
{
    sem_init(&S1, 0, 1);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thre
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_destroy(& S1);
    return 0;
}
```

Defining Semaphore Variable

Initializing semaphore variable (0 as the second parameter means shared between threads)

Destr

```
void* thread(void* arg)
{
    //wait
    sem_wait(& S1);
    printf("\nEntered..\n");

    //critical section
    sleep(4);
    //signal
    printf("\nJust Exiting...\n");
    sem_post(& S1);
}
```

Until the value of S1 is 0, sem_wait blocks the operation.
Then, it decrement the value

Sem-post it the semaphore signal

OUTPUT:

Correct	Incorrect
Entered..	Entered..
Just Exiting	Entered..
Entered..	Just Exiting
Just Exiting	Just Exiting

Semaphore Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <semaphore.h>

sem_t   s1;
char running = 1;
long long counter = 0;

void * process()
{
        while (running)
        {
                sem_wait(&s1);
                counter++;
                sem_post(&s1);
         }
        printf("Thread: exit\n");
pthread_exit(NULL);
}
```

```
int main()
{
        int i;
        long long sum = 0;
        pthread_t thread_Id;
        sem_init(&s1, 0, 1);
        if (pthread_create(&thread_Id, NULL, process, NULL))
        {
                        printf("ERROR in pthread_create()\n");
                        exit(-1);
        }
        for (i=0 ; i < 10 ; i++)
        {
                sleep(1);
                sem_wait(&s1);
                printf("counter = %lld\n", counter);
                sum += counter;
                counter = 0;
                sem_post(&s1);
        }
        sem_wait(&s1);
        running = 0;
        sem_post(&s1);
        pthread_join(thread_Id,  NULL);
        printf("Average Instructions = %lld \n", sum/10);
        return 0;
}
```

Exercise

- Write a synchronization protocol to allow the following order:

Thread 1	Thread 2
Count = 5	
	Count ++;
…	

Exercise

- Write a synchronization protocol to allow the following order:

T 1	T 2
Count = 5	
	Count ++;
…	

Algorithm:
2 semaphores are used (S1, S2), initialized to 0

T1:	T2:
count = 5	wait (S1)
signal (S1)	count ++;
wait (S2)	signal (S2)

Semaphores with processes

- If semaphores are shared by different processes, they need to be allocated in a shared memory

Exercise

- Write a synchronization protocol to allow the following order:

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SEGMENTPERM 0666

int main(int argc, char* argv[])
{       sem_t * sp;
        int retval;
        int id, err;
        int i;
        int nloop = 4;
/* Make shared memory segment. */
        id = shmget (IPC_PRIVATE, sizeof(sem_t), SEGMENTPERM);
        if (id == -1) perror("Creation");
        sp = (sem_t *) shmat(id,(void*) 0, 0);
/* Initialize the semaphore. */
        retval = sem_init(sp,1,1);
        if (retval != 0) {
        perror("Couldn't initialize.");
        exit(3);
}
```

```
if (fork() == 0) { /* child process*/
    for (i = 0; i < nloop; i++) {
      sem_wait(sp);
      printf("child entered critical section: %d\n", i);
      sleep(2);
      printf("child leaving critical section\n");
      sem_post(sp);
      sleep(1);
    }
    exit(0); /* end of child process*/}
/* back to parent process */
  for (i = 0; i < nloop; i++) {
    sem_wait(sp);
    printf("parent entered critical section: %d\n", i);
    sleep(2);
    printf("parent leaving critical section\n");
    sem_post(sp);
    sleep(1);
  }
sem_destroy(sp);
/* Remove segment. */
err = shmctl(id, IPC_RMID, 0);
if (err == -1) perror("Removal.");
return 0;
}
```

Exercise

- Write a synchronization protocol to allow the following order:

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SEGMENTPERM 0666

int main(int argc, char* argv[])
{       sem_t * sp;
        int retval;
        int id, err;
        int i;
        int nloop = 4;
/* Make shared memory segment. */
        id = shmget (IPC_PRIVATE, sizeof(sem_t), SEGMENTPERM);
        if (id == -1) perror("Creation");
        sp = (sem_t *) shmat(id,(void*) 0, 0);
/* Initialize the semaphore. */
        retval = sem_init(sp,1,1);
        if (retval != 0) {
        perror("Couldn't initialize.");
        exit(3);
}
```

```
if (fork() == 0) { /* child process*/
    for (i = 0; i < nloop; i++) {
      sem_wait(sp);
      printf("child entered critical section: %d\n", i);
      sleep(2);
      printf("child leaving critical section\n");
      sem_post(sp);
      sleep(1);
    }
    exit(0); /* end of child process*/}
/* back to parent process */
  for (i = 0; i < nloop; i++) {
    sem_wait(sp);
    printf("parent entered critical section: %d\n", i);
    sleep(2);
    printf("parent leaving critical section\n");
    sem_post(sp);
    sleep(1);
  }
sem_destroy(sp);
/* Remove segment. */
err = shmctl(id, IPC_RMID, 0);
if (err == -1) perror("Removal.");
return 0;
}
```

Output:

Output:
parent entered critical section: 0
parent leaving critical section
child entered critical section: 0
child leaving critical section
parent entered critical section: 1
parent leaving critical section
child entered critical section: 1
child leaving critical section
parent entered critical section: 2
parent leaving critical section
child entered critical section: 2
child leaving critical section
parent entered critical section: 3
parent leaving critical section
child entered critical section: 3
child leaving critical section

Mutex

- Another popular way to achieve synchronization is by using **Mutex**.
  - A Mutex is a lock that we set before CS and release it after.
  - When the lock is set, no other thread can access the locked region of the code.

```
While (true) {
    Acquire lock
        critical section
    Release lock
        remainder section
}
```

Mutex Operations

- Mutex variable is of type `pthread_mutex_t`.

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

- Initializing of a mutex variable by default attributes
  `int pthread_mutex_init (pthread_mutex_t  *lock,  const  pthread_mutexattr_t  * attr)`

- Lock operation
  `int pthread_mutex_lock (pthread_mutex_t  *lock)`

- Unlock operation
  `int pthread_mutex_unlock (pthread_mutex_t  *lock)`

- Destroy mutex
  `int pthread_mutex_destroy (pthread_mutex_t  *lock)`

Mutex

- Initialization of a Mutex variable

Syntax:

```
int pthread_mutex_init (pthread_mutex_t  *lock,   const pthread_mutexattr_t   *attr);
```

`lock` : The mutex Variable

`attr`: Mutex attrbiutes soecified by attr: if the attr is NULL, the default mutex attributes is used.

If successful, `pthread_mutex_init()` returns 0, and the state of the mutex becomes initialized and unlocked.
If unsuccessful, `pthread_mutex_init()` returns -1.

Mutex

- Locking a mutex

Syntax:

```
int pthread_mutex_lock (pthread_mutex_t  *lock);
```

`lock` : The mutex Variable

If successful, `pthread_mutex_lock()` returns 0. If unsuccessful, `pthread_mutex_lock()` returns -1.

Mutex

- Release the lock on mutex

Syntax:

```
int pthread_mutex_unlock (pthread_mutex_t  *lock);
```

`lock` : The mutex Variable

If successful, `pthread_mutex_unlock()` returns 0. If unsuccessful, `pthread_mutex_unlock()` returns -1.

Mutex

- Delete the mutex

Syntax:

```
int pthread_mutex_destroy (pthread_mutex_t  *lock);
```

`lock` : The mutex Variable

If successful, `pthread_mutex_destroy()` returns 0. If unsuccessful, `pthread_mutex_destroy()` returns -1.

Mutex Example

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;
int main(void)
{
    int i = 0;
    int error;
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return (1); }
    while (i < 2) {
         pthread_create(&(tid[i]), NULL, &trythis, NULL);
         i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
     return 0;
}
```

```
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

Mutex Example

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;
int main(void)
{
    int i = 0;
    int error;
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return (1); }
    while (i < 2) {
         pthread_create(&(tid[i]), NULL, &trythis, NULL);
         i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
     return 0;
}
```

Defining a mutex

Initializing the mutex “lock” with default attributes

Destroying the Mutex

```
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

locking

Unlocking

Mutex Example

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;
int main(void)
{
    int i = 0;
    int error;
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return (1); }
    while (i < 2) {
         pthread_create(&(tid[i]), NUL
         i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
     return 0;
}
```

```
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

OUTPUT:

✓	✗
Job 1 started	Job 1 has started
Job 1 finished	Job 2 has started
Job 2 started	Job 2 has finished
Job 2 finished	Job 2 has finished

Mutex Example

```
#include <semaphore.h>


int a=1, b=1;
pthread_mutex_t m;
void* thread1(int *arg) {
	pthread_mutex_lock(&m);
	printf("First thread (parameter: %d)\n", *arg);
	a++;
	b++;
	pthread_mutex_unlock(&m);
}
void* thread2(int *arg) {
	pthread_mutex_lock(&m);
	printf("Second thread (parameter: %d)\n",
*arg);
	b=b*2;
	a=a*2;
	pthread_mutex_unlock(&m);
}
```

```
main() {
	pthread_t threadid1, threadid2;
	int i = 1, j=2;

	pthread_mutex_init(&m, NULL);

	pthread_create(&threadid1, NULL, thread1, &i);
	pthread_create(&threadid2, NULL, thread2, &j);
	pthread_join(threadid1, NULL);
	pthread_join(threadid2, NULL);
	pthread_mutex_destroy(&m);

	printf("Final Values: a=%d b=%d\n", a, b);
}
```

Condition Variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

- In a critical section, a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.

  - It will suspend by waiting on a condition variable
  - It will release the critical section lock (MUTEX)
  - When that condition variable is signaled, it will become ready again. It will attempt to reacquire that critical section lock and only then it will be able to proceed.

Condition Variables

- To declare a condition variable

- Syntax:

```
pthread_cond_t   cond;
```

Condition Variables

- To create a condition variable
- Syntax:

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

- `Cond` represent the condition variable
- `Attr` permits the setting of condition variable object attributes. It the `attr` is NULL; the default attribute is used.

If successful, `pthread_cond_init()` returns 0.
If unsuccessful, an error code is returned.

Condition Variables

- To wait on a condition variable

- Syntax:

  int pthread_cond_wait (pthread_cond_t *cond, pthrea_mutex_t *mutex)

- `Cond` represent the condition variable
- `mutex` refers to the associated mutex parameter

It returns 0 if successful and an error code if not successful.

Condition Variables

- To signal condition variable

- Syntax:

  int pthread_cond_signal (pthread_cond_t *cond)

- `Cond` represent the condition variable

It returns 0 if successful and an error code if not successful.

General Scheme of Condition Variables

P0

pthread_mutex_lock(&mutex);

while(condition_is_false)
    pthread_cond_wait(&condition, &mutex);

Critical Section

pthread_mutex_unlock(&mutex);

P1

pthread_mutex_lock(&mutex);

Critical Section

"When" condition is verified
    pthread_cond_signal(&condition);

pthread_mutex_unlock(&mutex);

Condition Variable Example

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;

int done = 1;
int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, foo, NULL);
   // sleep for 1 sec so that thread 1
   // would get a chance to run first
    sleep(1);
    // Create thread 2
    pthread_create(&tid2, NULL, foo, NULL);
    // wait for the completion of thread 2
    pthread_join(tid2, NULL);
     return 0;
}
```

Declaring Mutex

Declaring the Condition Variable

```c
void* foo()
{
     // acquire a lock
    pthread_mutex_lock(&lock);
    if (done == 1) {
     done = 2;
      printf("Waiting on condition variable cond1\n");
      pthread_cond_wait(&cond1, &lock);
    }
    else {

        printf("Signaling condition variable cond1\n");
        pthread_cond_signal(&cond1);
    }

      pthread_mutex_unlock(&lock);
      printf("Returning thread\n");

    return NULL;
}
```

Locking by Mutex

Waiting a condition variable cond1

Signaling a condition variable cond1

Releasing the lock

Condition Variable Example

- Thread waits that a condition is satisfied before continuing.

```
int done = 0;
pthread_mutex_t mutex;
pthread_cond_t cv;

void *child (void *arg)
{
  printf("Child Begins\n");
  pthread_mutex_lock(&mutex);
  printf("Child is Sleeping ....\n");
  sleep(5);
  done = 1;
  printf("Child: Done....\n");
  pthread_cond_signal(&cv);
  sleep(5);
  printf("Child terminates the Critical Section ....\n");
  pthread_mutex_unlock(&mutex);
  printf("Child unlocks mutex....\n");
  return ;
}
```

```
int main()
{
pthread_t c;

printf("Parent Begins\n");
pthread_mutex_init(&mutex, NULL);
pthread_cond_init (&cv, NULL);
pthread_create(&c, NULL, child, NULL);
pthread_mutex_lock(&mutex);
while (done == 0)
   {
       printf("Parent is Waiting\n");
       pthread_cond_wait(&cv, &mutex);
       printf("Condition OK & Mutex Unlock: Parent can continue \n");
}
pthread_mutex_unlock(&mutex);
printf("Parent: out\n");
sleep(5);
printf("Parent: end\n");
return 0;
}
```

Condition Variable Example

- Thread waits that a condition is satisfied before continuing.

```
int done = 0;
pthread_mutex_t mutex;
pthread_cond_t cv;

void *child (void *arg)
{
  printf("Child Begins\n");
  pthread_mutex_lock(&mutex);
  printf("Child is Sleeping ....\n");
  sleep(5);
  done = 1;
  printf("Child: Done....\n");
  pthread_cond_signal(&cv);
  sleep(5);
  printf("Child terminates the Critical Section ....\n");
  pthread_mutex_unlock(&mutex);
  printf("Child unlocks mutex....\n");
  return ;
}
```

```
int main()
{
pthread_t c;

printf("Parent Begins\n");
pthread_mutex_init(&mutex, NULL);
pthread_cond_init (&cv, NULL);
pthread_create(&c, NULL, child, NULL);
pthread_mutex_lock(&mutex);
while (done == 0)
   {
      printf("Parent is Waiting\n");
      pthread_cond_wait(&cv, &mutex);
      printf("Condition OK & Mutex Unlock: Parent can continue \n");
}
pthread_mutex_unlock(&mutex);
printf("Parent: out\n");
sleep(5);
printf("Parent: end\n");
return 0;
}
```

Output:

```
./a.out
Parent Begins
Parent is Waiting
Child Begins
Child is Sleeping ....
Child: Done....
Child terminates the Critical
Section ....
Child unlocks mutex....
Condition OK & Mutex Unlock:
Parent can continue
Parent: out
Parent: end
```

Condition Variable Example

- The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

```
#include <semaphore.h>
#include <stdio.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

void *inc_count(void *);
void *watch_count(void *);



int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
}
```

```
int main (int argc, char *argv[])
{  int i, rc;
  pthread_t threads[3];
  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);
  pthread_create(&threads[0],
                          NULL,
                          inc_count,
                          (void *)&thread_ids[0]);
  pthread_create(&threads[1],
                          NULL,
                          inc_count,
                          (void *)&thread_ids[1]);
  pthread_create(&threads[2],
                          NULL,
                          watch_count,
                           (void *)&thread_ids[2]);
```

```
/* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n",
NUM_THREADS);

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
} /* of main */
```

Condition Variable Example

- The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

```
void *inc_count(void *idp)
{  int j,i;
  double result=0.0;
  int *my_id = idp;
  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
    /* Check the value of count and signal waiting thread when condition is
       reached.  Note that this occurs while mutex is locked. */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %d, count = %d  Threshold reached.\n", *my_id, count);}
else
        printf("inc_count(): thread %d, count = %d, unlocking mutex\n",*my_id, count);
    pthread_mutex_unlock(&count_mutex);
    /* Do some work so threads can alternate on mutex lock */
    for (j=0; j<1000; j++) result = result + (double)random();
  }
  pthread_exit(NULL);
}
```

Condition Variable Example

- The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

```
void *watch_count(void *idp)
{
  int *my_id = idp;
  printf("Starting watch_count(): thread %d\n", *my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the test will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal received.\n", *my_id);
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Condition Variable Example

- The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

Output:

```
inc_count(): thread 0, count = 1, unlocking mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking mutex
inc_count(): thread 0, count = 3, unlocking mutex
inc_count(): thread 1, count = 4, unlocking mutex
inc_count(): thread 0, count = 5, unlocking mutex
inc_count(): thread 0, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 0, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 0, count = 10, unlocking mutex
inc_count(): thread 1, count = 11, unlocking mutex
inc_count(): thread 0, count = 12  Threshold reached.
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 13, unlocking mutex
inc_count(): thread 0, count = 14, unlocking mutex
inc_count(): thread 1, count = 15, unlocking mutex
inc_count(): thread 0, count = 16, unlocking mutex
inc_count(): thread 1, count = 17, unlocking mutex
inc_count(): thread 0, count = 18, unlocking mutex
inc_count(): thread 1, count = 19, unlocking mutex
inc_count(): thread 1, count = 20, unlocking mutex
Main(): Waited on 3  threads. Done.
```