ECE437/CS481

M04C: PROCESS COORDINATION MONITOR & MUTEX LOCKS

CHAPTER 5.5 & 5.8

Xiang Sun

The University of New Mexico

Monitor

- **Applying semaphores is error-prone to programmers.**
  - The programmers have to put wait() before the critical section to issue a lock, and put signal() after the critical section to release the lock; otherwise, the process may not perform correctly, e.g.,

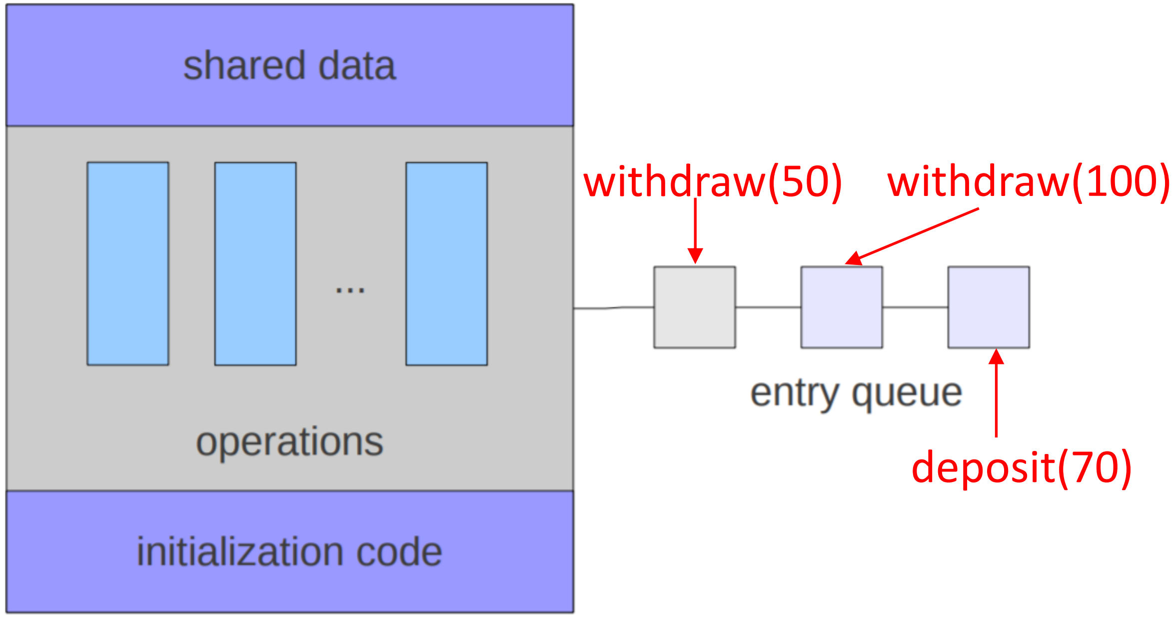
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Monitor**
  - A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
  - Monitor has been implemented as an object or module in Java, c++, Pascal,…

Monitor contains

- Shared data structures
- Procedures/operations that operate on the shared data structures.
- Synchronization between concurrent procedure invocations (i.e., processes/threads).

```
Monitor account {
    double balance;
    wait(s);
    double withdraw(amount) {
        balance = balance – amount;
        return balance;}
    signal(s);
    wait(s);
    double deposit(amount) {
        balance = balance + amount;
        return balance;}
    signal(s);
}
```



Monitor

- Monitor can achieve mutual exclusion among processes/threads by adding wait() and signal() for each procedure/operation.

- But what if a process/thread has to wait inside the monitor?
  - Recall that, in the producer/consumer problem, the producer is to generate a data item, put it into the queue, and the consumer is to consume a data item. The producer won't try to add data item into the queue if it's full.
  - The producer/consumer application can be implemented by a monitor.
  - Can this monitor-based solution achieve critical section to solve the producer/consumer problem?

```
Monitor Producer_Consumer {
    int itemCount = 0;

    producer(item) {
        while (itemCount==QUEUE_SIZE)
          {//wait until the queue not full};
        putItemIntoQueue(item);
        itemCount = itemCount + 1;}

    consumer() {
        while (itemCount==0)
          {//wait until the queue not empty};
        item = removeItemFromQueue();
        itemCount = itemCount - 1;
        return item;}
}
```

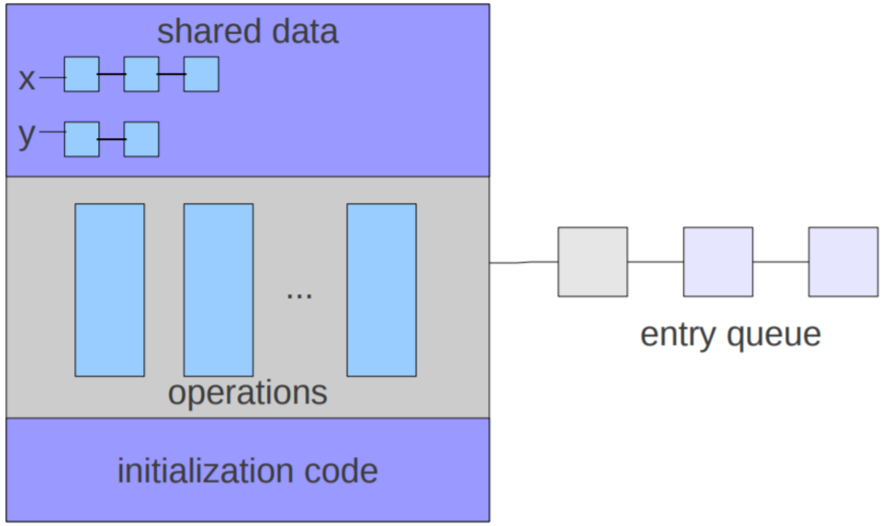
Monitor

- Condition variables are used in the monitor to provide a mechanism to enable processes/threads to release the CPU resource and wait for events.

- Denote x as a condition variable. One queue associated to x, and two atomic operations are defined on x:

  - x.wait(): release monitor lock, block the calling process, and add this process into condition variable queue.

  - x.signal(): remove a process from the condition variable queue, and resume this process.



Producer-Consumer using Monitors

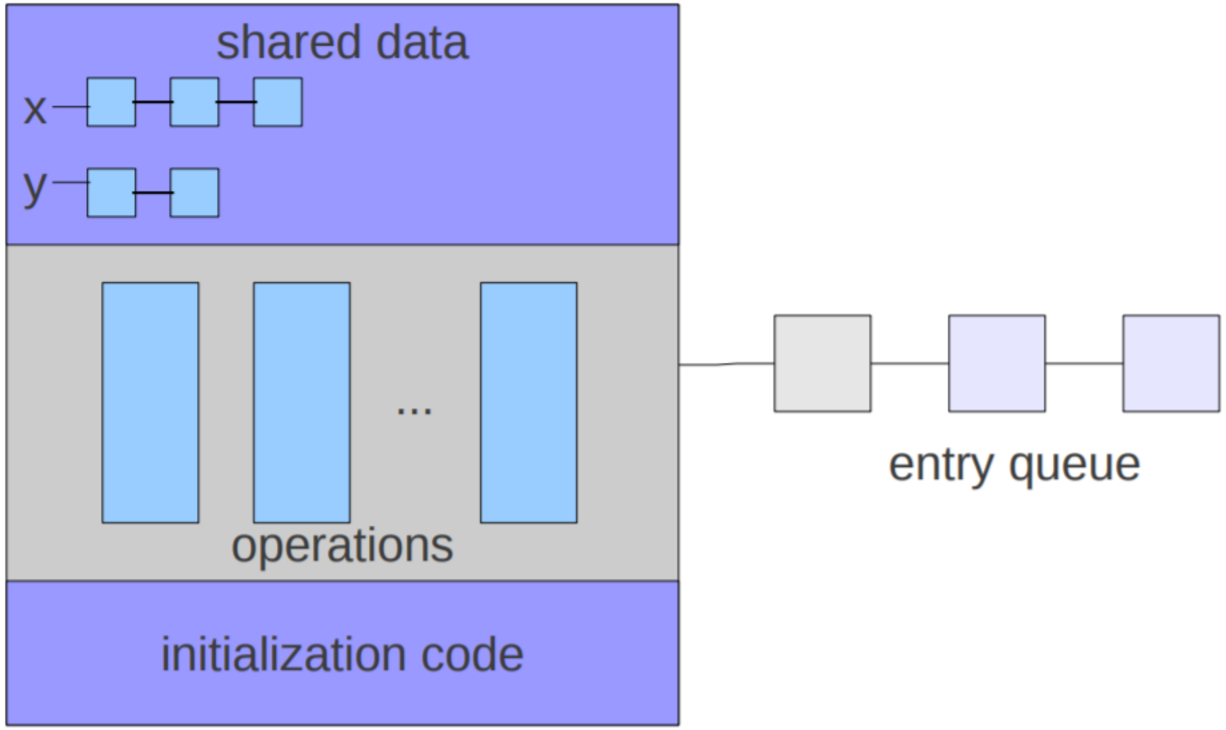
```
Monitor Producer_Consumer {
    int itemCount = 0;
    condition not_empty;
    condition not_full;

    producer(item) {
        if (itemCount== QUEUE_SIZE)
           {not_full.wait();}
        putItemIntoQueue(item);
        itemCount = itemCount + 1;
           not_empty.signal(); }

     consumer() {
         if (itemCount==0)
           {not_empty.wait();}
        item = removeItemFromQueue();
        itemCount = itemCount - 1;
            not_full.signal();
        return item;}
}
```

❑ Monitor

- ➢ Monitors have two kinds of "wait" queues
  - ✓ Entry to the monitor: has a queue of processes/threads waiting to obtain mutual exclusion so they can enter.
  - ✓ Condition variables: each condition variable has a queue of processes/threads waiting on the associated condition



- ➢ Monitor is considered as a non busy-waiting implementation

Mutex Locks

❑ Mutex Locks

- A mutual lock that allows one thread in accessing a critical section, and blocks other threads, which try to access the same critical section.

- Its function is very similar with binary semaphore, but can only achieve critical section among threads.

- Mutex lock is implemented based on atomic instructions (e.g., Test_and_Set and Comp_and_Swap).

Mutex Locks

❑ Thread API: mutex lock

- int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr):
  - ✓ initialize a lock
- int pthread_mutex_lock(pthread_mutex_t *mutex):
  - ✓ obtain lock; if the mutex lock is already locked, the calling thread blocks until the mutex becomes available.
- int pthread_mutex_unlock(pthread_mutex_t *mutex):
  - ✓ release exclusive lock
- int pthread_mutex_trylock(pthread_mutex_t *mutex):
  - ✓ obtain lock; if the mutex lock is currently locked (by any thread, including the current thread), the call returns immediately.
- int pthread_mutex_destroy(pthread_mutex_t *mutex):
  - ✓ delete lock

Mutex Locks

Thread API: mutex lock

- Demonstrate a simple use of pthread_mutex_trylock.

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

#define SPIN 10000000
pthread_mutex_t shared_mutex;
shared_counter=0;
time_t end_time;
```

```
int main (int argc, char *argv[]) {
 int s1,s2; pthread_t tid1, tid2;
 end_time = time(NULL) + 60; // run for 1 minute
 pthread_mutex_init(&shared_mutex,NULL);
 s1 = pthread_create(&tid1,NULL,(void *)counter_thread, NULL);
 s2 = pthread_create(&tid2,NULL,(void *)monitor_thread, NULL);
 if (s1==0) pthread_join(tid1,NULL);
 if (s2==0) pthread_join(tid2,NULL);
 exit(0);
}
```

- Initialize a mutex lock shared_mutex.
- Create two threads:
  - ✓ counter_thread: updates shared_counter at each interval;
  - ✓ monitor_thread: reports the current value of shared_counter, but only if the mutex is not locked by counter_thread;

Mutex Locks

❑ Thread API: mutex lock

- Demonstrate a simple use of pthread_mutex_trylock.
  - ✓ counter_thread updates a shared_counter at intervals

```
void counter_thread (void *arg){
while (time(NULL) < end_time) {
pthread_mutex_lock(&shared_mutex);
for (int spin=0; spin<SPIN; spin++) shared_counter++;
pthread_mutex_unlock(&shared_mutex);
sleep(1);
}
}
```

  - ✓ monitor_thread reports the current value of the counter, but only if the mutex is not locked by counter_thread.

```
void monitor_thread (void *arg) {
int misses, status; time_t thistime;
while (time(&thistime) < end_time) {
status = pthread_mutex_trylock(&shared_mutex);
if (status != EBUSY) {
printf("At time %ld Counter is %d\n", thistime, shared_counter/SPIN);
pthread_mutex_unlock(&shared_mutex); }
else {
misses++;
printf("At time %ld Counter is being LOCKED\n", thistime);
}
}
}
```

Mutex Locks

❑ Thread API: mutex lock

- ➢ Demonstrate a simple use of pthread_mutex_trylock.
  - ✓ Results

```
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is 0
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
At time 1538412754 Counter is being LOCKED
```

Mutex Locks

❑ Thread API: mutex lock

- Locking refers to short-duration holding of resources

- Mutex locks are static variables that are accessible by all the threads

- Mutex locks must be released/unlocked by the same thread that acquired it—which is different from semaphore.

Thread API: condition variable

- pthread_cond_init
  - initialize a condition variable
- pthread_cond_wait
  - block on a condition variable
- pthread_cond_timewait
  - wait with timeout
- pthread_cond_signal
  - signal one thread on waiting the condition variable
- pthread_cond_destroy
  - delete a condition variable
- pthread_cond_broadcast
  - signal all threads waiting on the condition variable

❑ Thread API: condition variable+ mutex lock

- ➢ There are two threads: thread 1 and thread 2 both try to update shared variables x and y. However, thread 1 can update x and y iff x>y.

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, null);
```

thread 1:

```
pthread_mutex_lock(&mutex);
while (x <= y) {;}
/* operate on x and y */
pthread_mutex_unlock(&mutex);
```

thread 2:

```
pthread_mutex_lock(&mutex);
/* modify x and y */
pthread_mutex_unlock(&mutex);
```

❑ Thread API: condition variable+ mutex lock

- Consider two shared variables x and y, protected by the mutex lock mutex, and a condition variable cond that is to be signaled whenever x > y.
- There are two threads: thread 1 is blocked when x<=y; thread 2 is to modify the values of x and y; if x>y, thread 2 will signal thread 1 to be wakeup.
- Thread 1 will wake up to see if the condition is satisfied.

Question-1:
In thread 1, if x<=y and mutex is locked, then how could thread 2 modify the value of x and y?

Question-2:
If we move “if (x>y)” after pthread_mutex_unlock in thread2, does it still work?

```
pthread_mutex_t mutex;
pthread_cond_t cond;
pthread_mutex_init(&mutex, null);
pthread_cond_init(&cond,null);//cond = PTHREAD_COND_INITIALIZER;
```

thread 1:

```
pthread_mutex_lock(&mutex);
if (x <= y) {
      pthread_cond_wait(&cond, &mutex);}
/* operate on x and y */
pthread_mutex_unlock(&mutex);
```

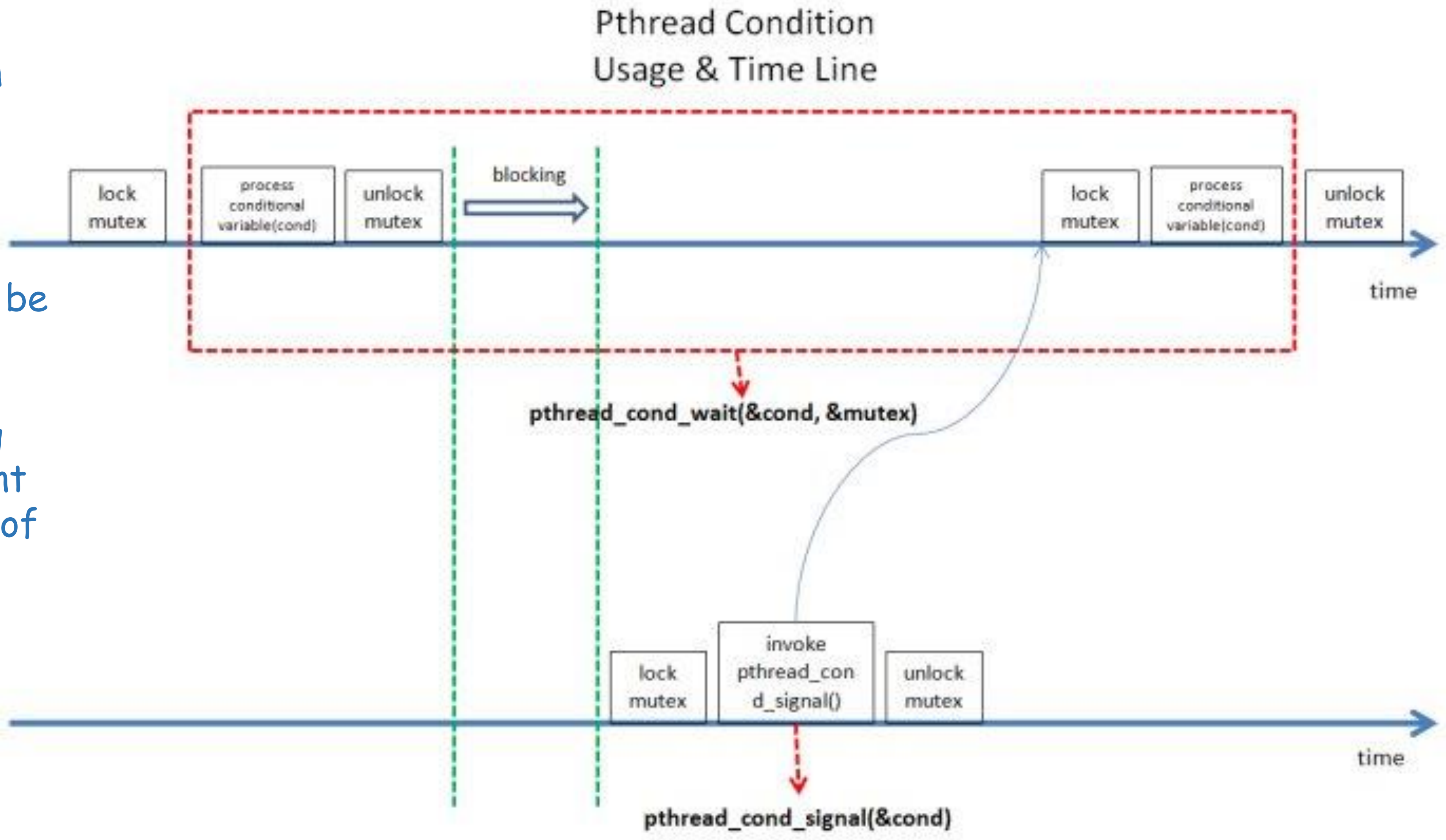
thread 2:

```
pthread_mutex_lock(&mutex);
/* modify x and y */
if (x > y) pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Mutex Locks

❑ Thread API: condition variable+ mutex lock

- If condition variable in thread 1 is satisfied, pthread_cond_wait() releases mutex and blocks thread 1.
- The mutex lock should be set before calling pthread_cond_wait(). The purpose of setting mutex lock is to prevent simultaneous requests of pthread_cond_wait().



Mutex Locks

❑ Thread API: condition variable+ mutex lock

- ➢ Example: one shared variables *i*, protected by the mutex lock **mutex**, and a condition variable **cond** that is to be signaled whenever *i* is a multiple of 3.
  - ✓ If variable i is Not a multiple of 3 (i.e., i%3!=0), thread 2 is blocked, and thread 1 will be executed by printing out "i: thread 2 is blocked".
  - ✓ If variable i is a multiple of 3 (i.e., i%3=0), thread 1 will wake up thread 2, and thread 2 will print out "i: thread 2 is executed"

```
shaun@shaun-VirtualBox:~/OS_code/condition_var$ ./cond_var
1:thread 2 is blocked
2:thread 2 is blocked
3:thread 2 is executed
3:thread 2 is executed
4:thread 2 is blocked
5:thread 2 is blocked
6:thread 2 is executed
6:thread 2 is executed
7:thread 2 is blocked
8:thread 2 is blocked
9:thread 2 is executed
```

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *thread1(void *);
void *thread2(void *);

int i=1;
int main(void){
         pthread_t t_a,t_b;
        pthread_create(&t_a,NULL,thread1,(void *)NULL);
        pthread_create(&t_b,NULL,thread2,(void *)NULL);
        pthread_join(t_b, NULL);
        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&cond);
        exit(0);
}

void *thread1(void *junk)
{
        for(i=1;i<=9;i++)
        {     pthread_mutex_lock(&mutex);
              if(i%3==0)
                  pthread_cond_signal(&cond);
               else
                   printf("%d:thread 2 is blocked\n",i);
               pthread_mutex_unlock(&mutex);
               sleep(1);}

}

void *thread2(void *junk)
{
        while(i<9)
        {
            pthread_mutex_lock(&mutex);
            if(i%3!=0)
              pthread_cond_wait(&cond,&mutex);
            printf("%d:thread 2 is executed\n",i);
            pthread_mutex_unlock(&mutex);
            sleep(1);}

}
```