Condition Variables

Another type of sychronization

- Locks allow one type of synchronization between threads – mutual exclusion
- Another common requirement in multi-threaded applications – waiting and signaling

  -> E.g., Thread T1 wants to continue only after T2 has finished some task
- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: condition variables

Condition Variables

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition
- Another thread that makes the condition true can signal the CV to wake up a waiting thread
- Pthreads provides CV for user programs

  –> OS has a similar functionality of wait/signal for kernel threads
- Signal wakes up one thread, signal broadcast wakes up all waiting threads

Condition Variables

- Condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.
- The condition itself is protected by a mutex. A thread must first lock the mutex to change the condition state.

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

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- If successful, the `pthread_cond_init()` and `pthread_cond_destroy()` functions return zero.
- Otherwise, an error number is returned to indicate the error.

Condition Variables

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
                      *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
                      *mutex, const struct timespec *abstime);
```

- The `pthread_cond_wait()` and `pthread_cond_timedwait()` functions are used to block on a condition variable.
- They are called with mutex locked by the calling thread.
- Upon successful completion, a value of zero is returned.
- Otherwise, an error number is returned to indicate the error.

Condition Variables

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- These two functions are used to unblock threads blocked on a condition variable.
- The `pthread_cond_signal()` call unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).
- The `pthread_cond_broadcast()` call unblocks all threads currently blocked on the specified condition variable cond.
- If successful, the `pthread_cond_signal()` and `pthread_cond_broadcast()` functions return zero. Otherwise, an error number is returned to indicate the error.

Condition Variables: parent waits for child

```
int done

pthread_mutex_t     m      =
PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t      c      =
PTHREAD_COND_INITIALIZER;

void thr_exit() {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&C),;
    pthread_mutex_unlock (&m);
}

void child (void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
```

```
void thr_join() {
    pthread_mutex_lock (&m);
    while (done == 0)
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock (&m);
}

int main() {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Why check condition in while loop?

- In the example code, why do we check condition before calling wait?
  - ✓ In case the child has already run and done is true, then no need to wait
- Why check condition with “while” loop and not “if”?
  - ✓ To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

```
if (condition)
wait (condvar)
```

//small chance that condition may be false when wait returns

```
while (condition)
wait (condvar)
```

//condition guaranteed to be true since we check in while-loop

Why use lock when calling wait?

- What if no lock is held when calling wait/signal?

```
void thr_exit() {
        done = 1;
        pthread_cond_signal (&c);
}
void thr_join () {
        if (done ==0)
        pthread_cond_wait (&c);
}
```

- Race condition: missed wakeup
  - Parent checks done to be 0, decides to sleep, interrupted
  - Child runs, sets done to 0, signals, but no one sleeping yet
  - Parent now resumes and goes to sleep forever
- Lock must be held when calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

Example: Producer/Consumer problem

- A common pattern in multi-threaded programs
- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a queue. Worker threads get requests from this queue and process them.
- Setup: one or more producer threads, one or more consumer threads, a shared buffer of bounded size.

Producer/Consumer with 2 CVs

```
pthread_cond_t empty, fill;
pthread_mutex_t mutex;
void *producer (void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX)
           pthread_cond_wait(&empty, &mutex);
        put (i);
        thread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void consumer(void *arg) {
    int i; for (i=0; i <loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count ==0)
           pthread_cond_wait(&fill, &mytex);
        int tmp=get ();
        pthread_cond_signal(&empty)
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Producer Consumer with Condition Variables

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

#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}
```

Producer Consumer with Condition Variables

```
void *consumer(void *ptr)                                   /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                                 /* take item out of buffer */
                pthread_cond_signal(&condp);        /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

References

- https://www.cse.iitb.ac.in/~mythili/os/