Lecture 14: Condition Variables

Mythili Vutukuru

IIT Bombay

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

Example: parent waits for child

```
int done  = 0;
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(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

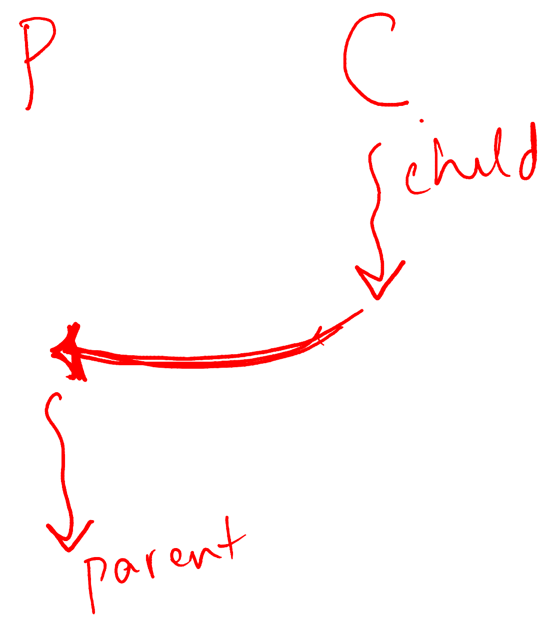


Figure 30.3: Parent Waiting For Child: Use A Condition Variable

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?

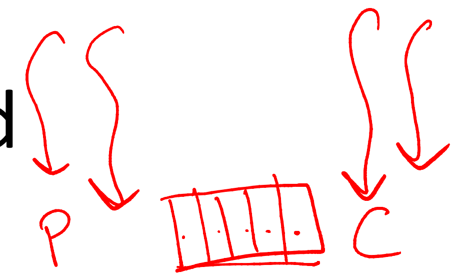
What if no lock is held when calling wait/signal?

```
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

```
cond_t empty, fill;
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);
        Pthread_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, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
```