P2L3 PThreads

PThreads == POSIX Threads
POSIX = Portable Operating System Interface

1. PThread Creation

```
// Correspond to Birrell's Thread datatype
pthread_t aThread; // Pthread datatype

// Correspond to Birrell's Fork(proc, args) for thread creation

// pthread_attr_t is a data structure that holds information about -
// certain things the thread manager should take into consideration -
// when schedule the thread

// returns status representation about whether the creation was success or failure.
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
void * (*start_routine)(void *), void *arg)

// Correspond to Join(thread)

// status captures all relevant return information + results returned from the thread
int pthread_join(pthread_t thread, void **status);

```

1.1 PThread Attributes

Attributes:

- stack size
- inheritance
- joinable
- scheduling policy
- priority
- system/process scope

The attributes have default values, if we pass in NULL for attr argument in pthread_create(), then the thread will be created with default values for its attributes.

PThread Attribute Interface

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
// set or read a certain attribute value
pthread_attr_{set/get}{attribute};
```

- By default, when a child thread is created, it's a joinable thread. The parent can wait and join the child thread, hence reaping the child thread.

- If the parent thread exits before joining its child threads, the child thread can become a zombie thread. After the child thread exits, it won't be reaped by its parent, so the memory assigned to it will not be freed.
- Detached thread
  - once detached, the thread cannot be joined
  - if a thread is detached, it will become equivalent to its parent thread
  - detached child thread can go on to execute after parent exit

```
1  // to detach a thread
2  pthread_detach();
```

To create a thread as detached thread

```
1  pthread_attr_setdetachstate(attr, PTHREAD_CREATE_DETACHED);
2
3  // ...
4
5  pthread_create(..., attr, ...)
```

For a thread to exit execution:

```
1  pthread_exit();
```

Example to create detached thread:

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

void *foo (void *arg) { /* thread main */
    printf("Foobar!\n");
    pthread_exit(NULL);
}

int main (void) {

    int i;
    pthread_t tid;

    pthread_attr_t attr;
    pthread_attr_init(&attr); /* required!!! */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_create(NULL, &attr, foo, NULL);

    return 0;
}
```

- Note that the PTHREAD_SCOPE_SYSTEM means the newly created thread will share resources equally with all other threads in the system.
  The create statement should be:

```
pthread_create(&tid, &attr, foo, NULL);
```

1.2 Compile Pthreads

1. include header:

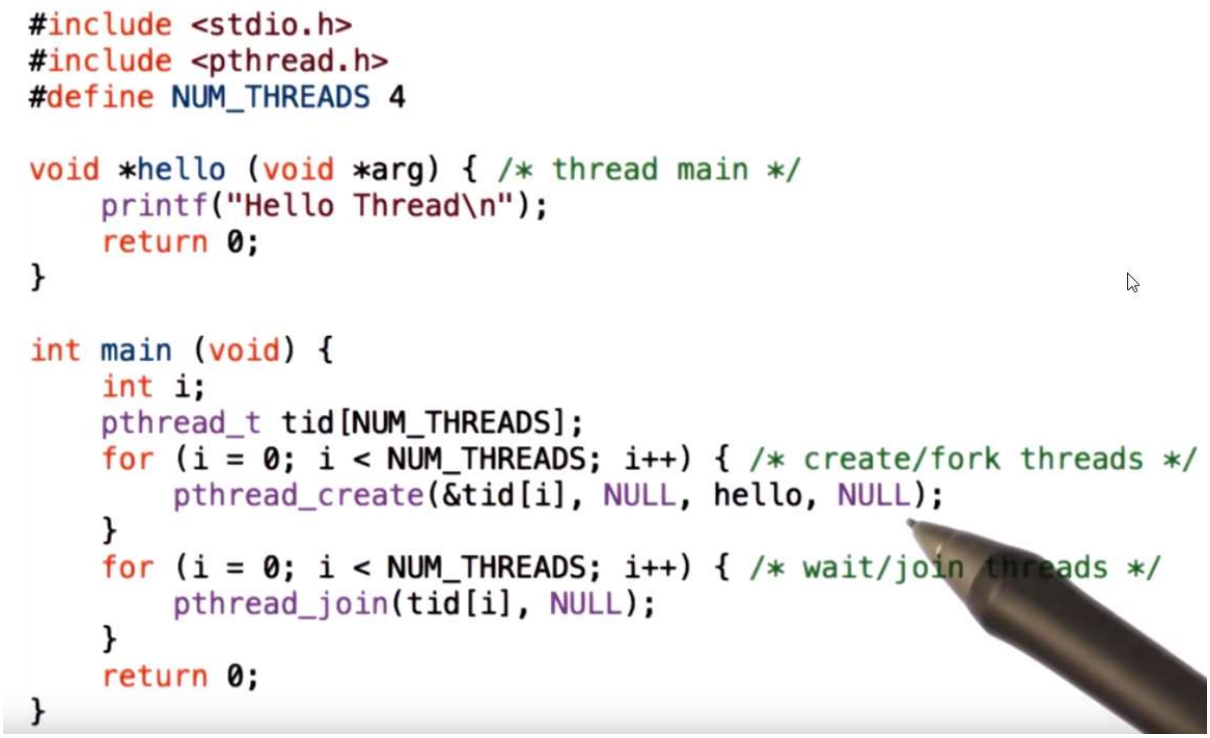
```
#include <pthread.h>
```

2. Compile source with -lpthread or -pthread

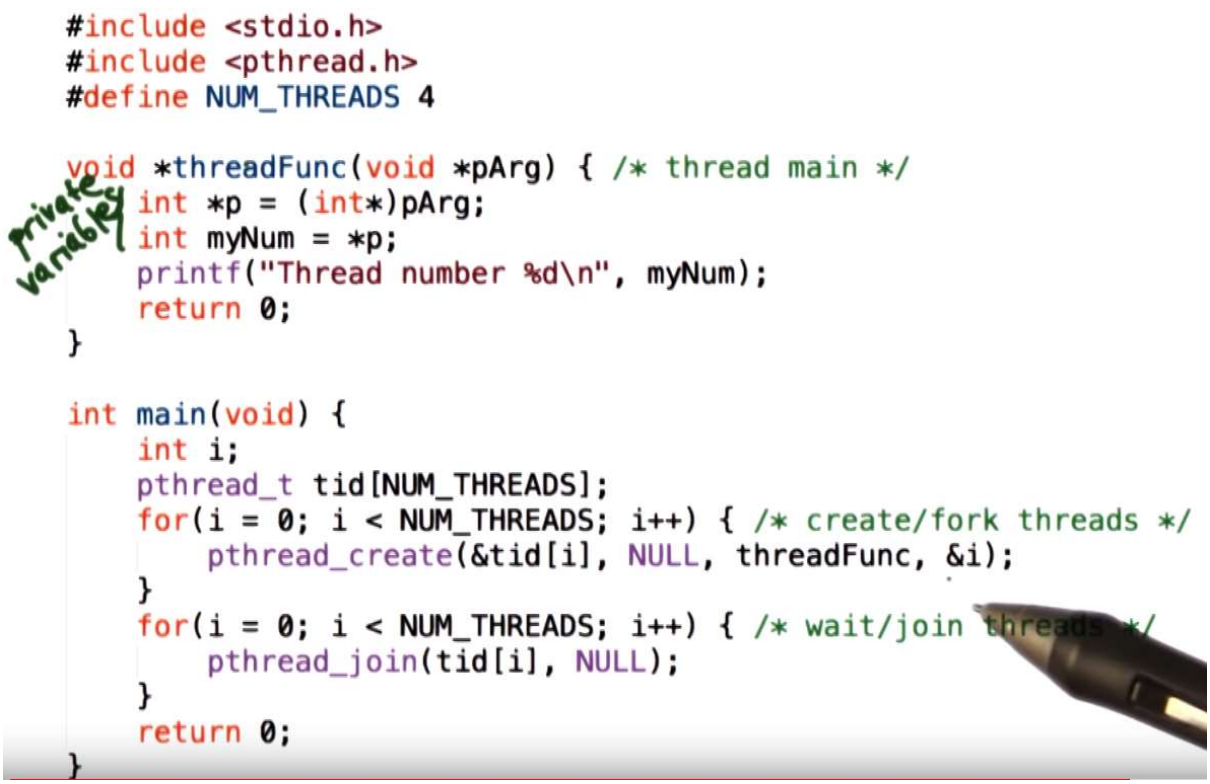
```
gcc -o main main.c -lpthread
gcc -o main main.c -pthread
```

3. Check return values of common functions

1.3 More Example

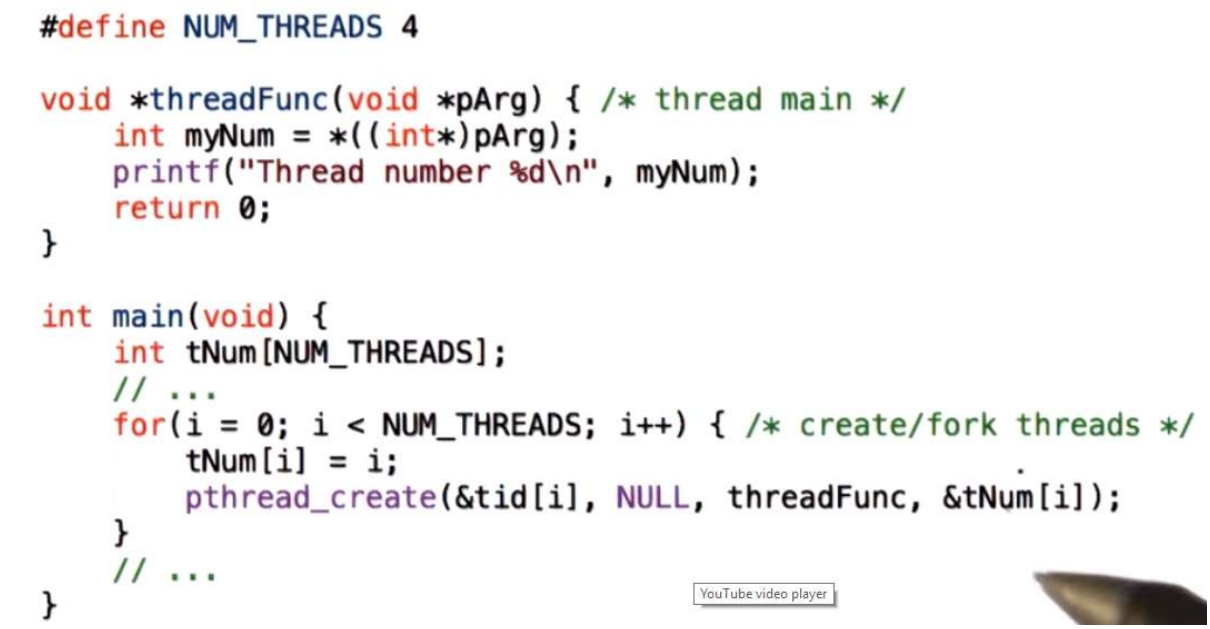


```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) { /* thread main */
    printf("Hello Thread\n");
    return 0;
}

int main (void) {
    int i;
    pthread_t tid[NUM_THREADS];
    for (i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        pthread_create(&tid[i], NULL, hello, NULL);
    }
    for (i = 0; i < NUM_THREADS; i++) { /* wait/join threads */
        pthread_join(tid[i], NULL);
    }
    return 0;
}
```




```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *threadFunc(void *pArg) { /* thread main */
    int *p = (int*)pArg;
    int myNum = *p;
    printf("Thread number %d\n", myNum);
    return 0;
}

int main(void) {
    int i;
    pthread_t tid[NUM_THREADS];
    for(i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        pthread_create(&tid[i], NULL, threadFunc, &i);
    }
    for(i = 0; i < NUM_THREADS; i++) { /* wait/join threads */
        pthread_join(tid[i], NULL);
    }
    return 0;
}
```

Note that in this program pArg is passed as a pointer to i, myNum can be tha value that i is when the child thread access it. It might have changed say from 1 to 2.
To correct this behavior, copy the i value each thread is supposed to take into a static array, then pass in pointers to array elements corresponding to each thread.




```c
#define NUM_THREADS 4

void *threadFunc(void *pArg) { /* thread main */
    int myNum = *((int*)pArg);
    printf("Thread number %d\n", myNum);
    return 0;
}

int main(void) {
    int tNum[NUM_THREADS];
    // ...
    for(i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        tNum[i] = i;
        pthread_create(&tid[i], NULL, threadFunc, &tNum[i]);
    }
    // ...
}
```

2. PThread Mutex

Pthread Mutex Syntax

```c
pthread_mutex_t aMutex; // mutex type
// lock operation
// explicit lock
int pthread_mutex_lock(pthread_mutex_t *mutex);
// explicit unlock
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Birrell :

```
list<int> my_list;
Mutex m;
void safe_insert(int i) {
  Lock(m) {
    my_list.insert(i);
  } // unlock;
}
```

Pthreads

```
list<int> my_list;
pthread_mutex_t m;
void safe_insert(int i) {
  pthread_mutex_lock(m);
  my_list.insert(i);
  pthread_mutex_unlock(m);
}
```

Other mutex operations

```
1  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
2  // mutex attributes specifies mutex behavior when a mutex is shared among processes
3  // the attr can be NULL, meaning follow the default behavior
4
5  // try to acquire the lock, if lock is free, then acquire it, if not it'll not block but return and
6  // notify the calling thread that the mutex is not available
7  int pthread_mutex_trylock(pthread_mutex_t *mutex);
8
9  // free mutex data structure
10 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mutex Safety Tips

- shared data should always be accessed through a single mutex!
- mutex scope must be visible to all!
- globally order locks
  - for all threads, lock mutexes in order
- always unlock a mutex
  - always unlock the correct mutex

3. Condition Variables

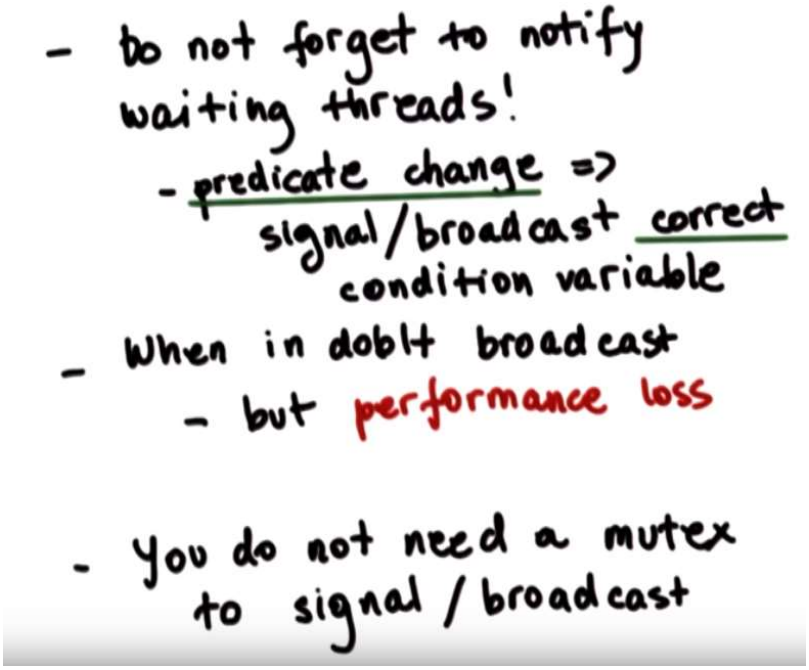
Basic PThread condition variable API:

```
pthread_cond_t aCond; // type of cond variable
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

```

Other condition variable operations:

```
// initiate a condition variable with attr attributes.
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Safety Tips:

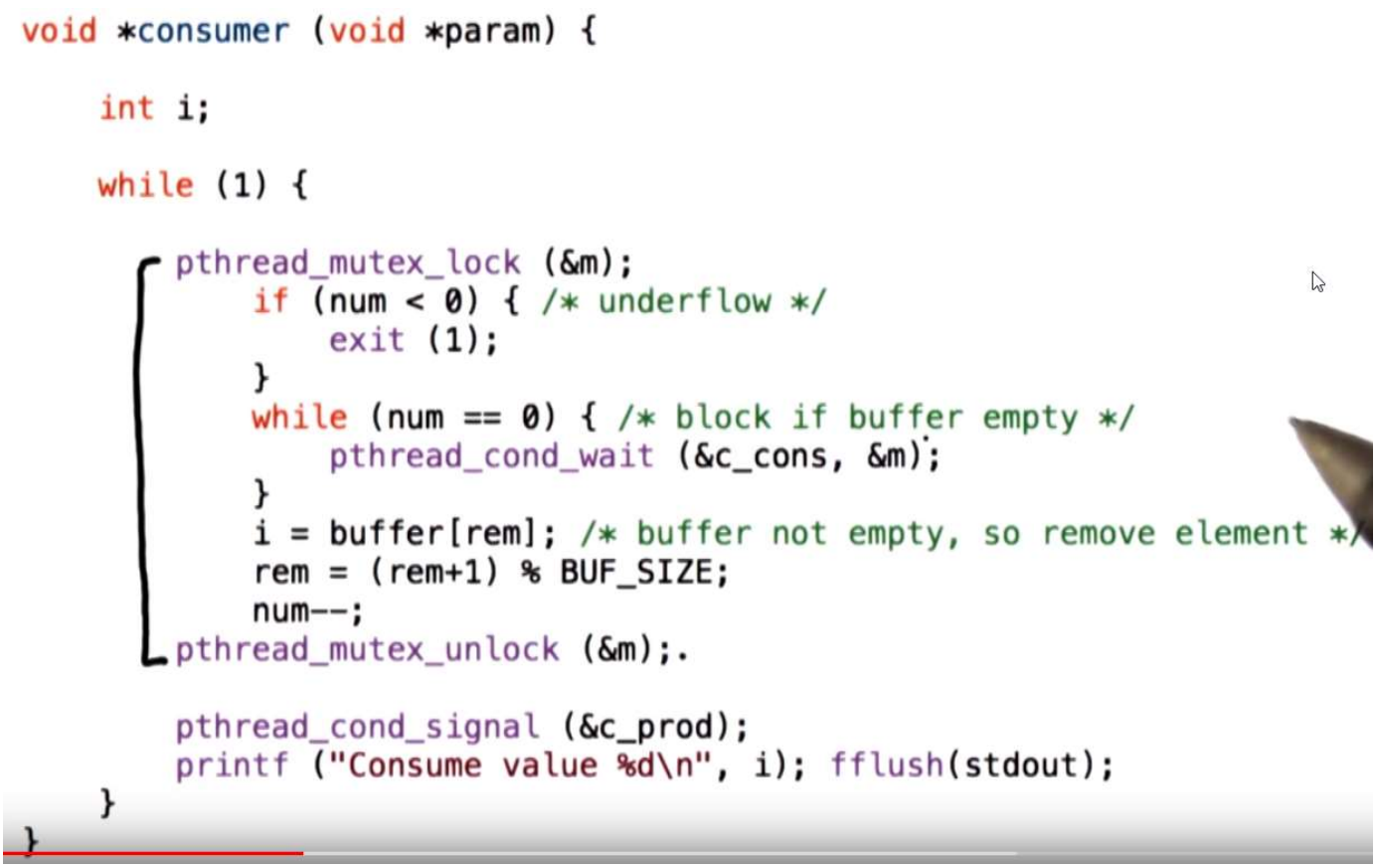


PThread Condition Variable Example — Producer Consumer Example:

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

#define BUF_SIZE 3      /* size of shared buffer */

int buffer[BUF_SIZE];   /* shared buffer */
int add = 0;            /* place to add next element */
int rem = 0;            /* place to remove next element */
int num = 0;            /* number elements in buffer */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;      /* mutex lock for buffer */
pthread_cond_t c_cons = PTHREAD_COND_INITIALIZER;   /* consumer waits on cv */
pthread_cond_t c_prod = PTHREAD_COND_INITIALIZER;   /* producer waits on cv */

void *producer (void *param);
void *consumer (void *param);
```

```c
int main(int argc, char *argv[]) {

    pthread_t tid1, tid2;  /* thread identifiers */
    int i;

    if (pthread_create(&tid1, NULL, producer, NULL) != 0) {
        fprintf (stderr, "Unable to create producer thread\n");
        exit (1);
    }

    if (pthread_create(&tid2, NULL, consumer, NULL) != 0) {
        fprintf (stderr, "Unable to create consumer thread\n");
        exit (1);
    }

    pthread_join(tid1, NULL); /* wait for producer to exit */
    pthread_join(tid2, NULL); /* wait for consumer to exit */
    printf ("Parent quiting\n");
}
```

```c
void *producer (void *param) {
    int i;
    for (i = 1; i <= 20; i++) {
        pthread_mutex_lock (&m);
            if (num > BUF_SIZE) { /* overflow */
                exit(1);
            }
            while (num == BUF_SIZE) { /* block if buffer is full */
                pthread_cond_wait (&c_prod, &m);
            }
            buffer[add] = i; /* buffer not full, so add element */
            add = (add+1) % BUF_SIZE;
            num++;
        pthread_mutex_unlock (&m);

        pthread_cond_signal (&c_cons);
        printf ("producer: inserted %d\n", i); fflush (stdout);
    }

    printf ("producer quiting\n"); fflush (stdout);
    return 0;
}
```



Note here the mutex is first released then signal condition variables to avoid spurious wakeups.