.P2L3 - Threads Case Study: Pthreads

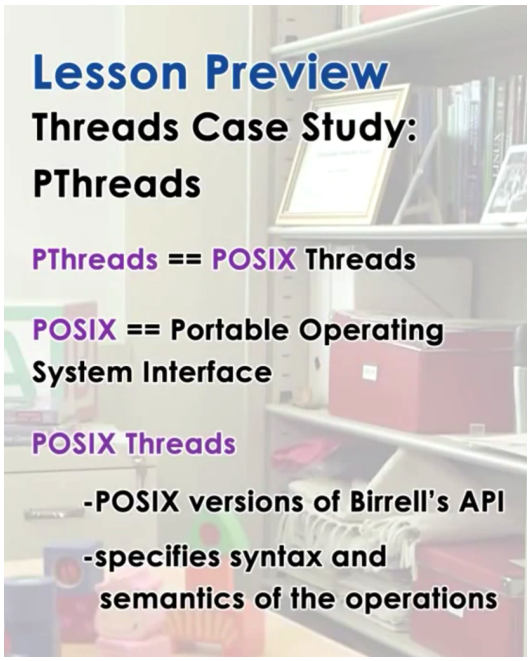
01 - Lesson Preview

PThread Links

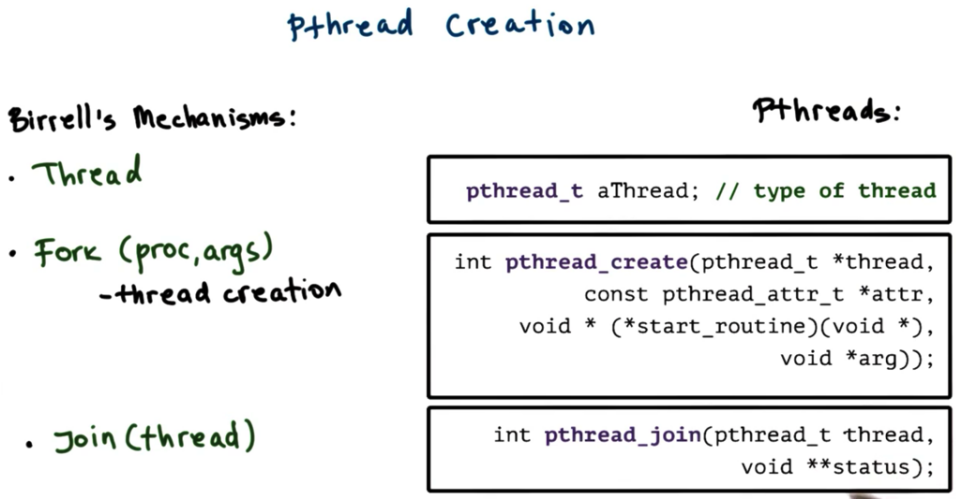
- "An Introduction to Programming with Threads"
- PThreads Programming Resource

PThread Lesson Resources

- PThread Coding Examples



02 - PThread Creation



pthread Attributes

pthread_attr_t

- specified in pthread_create
- defines features of the new thread
- has default behavior with NULL in pthread_create

```
int pthread_create(pthread_t *thread,
          const pthread_attr_t *attr,
      void * (*start_routine)(void *),
                          void *arg));
```

stack size
inheritance
joinable
scheduling policy
priority
system/process scope

pthread Attributes

pthread_attr_t

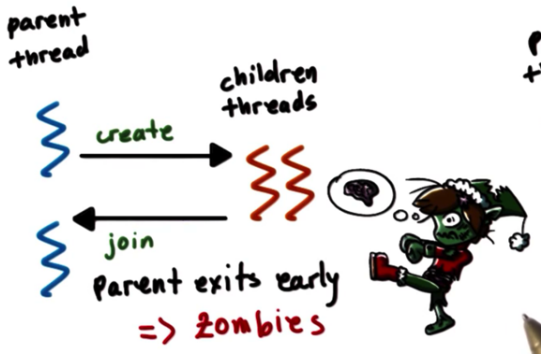
- specified in pthread_create
- defines features of the new thread
- has default behavior with NULL in pthread_create

```
   int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
              pthread_attr_{set/get}{attribute}
```

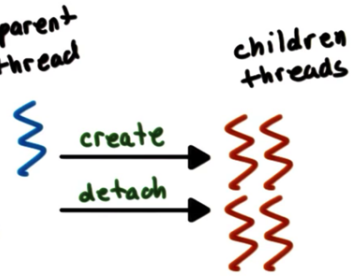
stack size
inheritance
joinable
scheduling policy
priority
system/process scope

Detaching pthreads

Default: Joinable threads



Detached threads

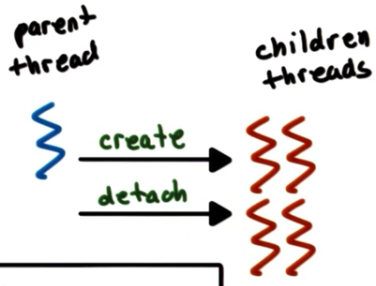


Detaching pthreads

```
int pthread_detach();
```

```
pthread_attr_setdetachstate(attr,
PTHREAD_CREATE_DETACHED);

// ...

pthread_create(..., attr, ...);
```

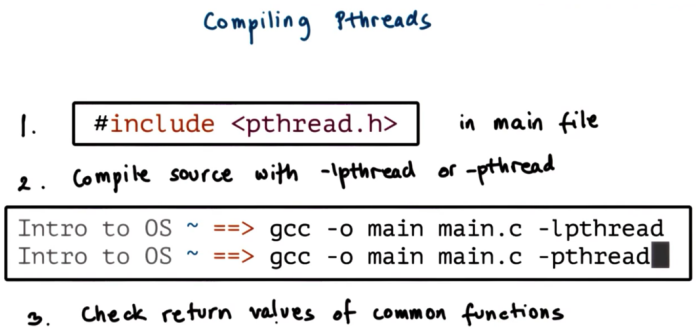
Detached threads




```
void pthread_exit();
```

```c
#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;
}
```

03 - Compiling PThreads



Before we look at some examples, there are few things that we need to consider when compiling threads. First make sure to include the pthread header file, pthread.h, in your main file that contains the pthreads code, otherwise your program will not compile. Second, make sure to link your program with the pthreads library by passing the lpthread flag at compile time. On certain platforms, the better option is to use a pthread instead as a flag, which tells the compiler to link the pthreads library as well as to configure the compilation for threads. If you don't link the library your program may not report certain compilation errors at compile time, but it will still fail. And finally, it's always a good idea to check the return values on common functions like when you're creating threads, creating variables, initializing certain data structures. This is a good programming practice in general, but it's extra useful when dealing with, all the hairy-ness of writing multithreaded programs.

04 - PThread Creation Example 1

Here is a simple example of creating threads with pthreads, let's look at the main function first. We see that in a loop we create pthread_create a number of times or, and we create four threads where each of the threads executes the function hello. All this function does is print Hello Thread, it doesn't take any arguments and that's why we pass NULL as an argument to

the pthread creation function. We also pass NULL in the attributes field because we're okay with just using the default pthread's behavior. That also means that these threads will be joinable, so then we have to, in a loop, join every single one of these threads.

05 - PThread Creation Quiz 1

06 - PThread Creation Quiz 1
admin

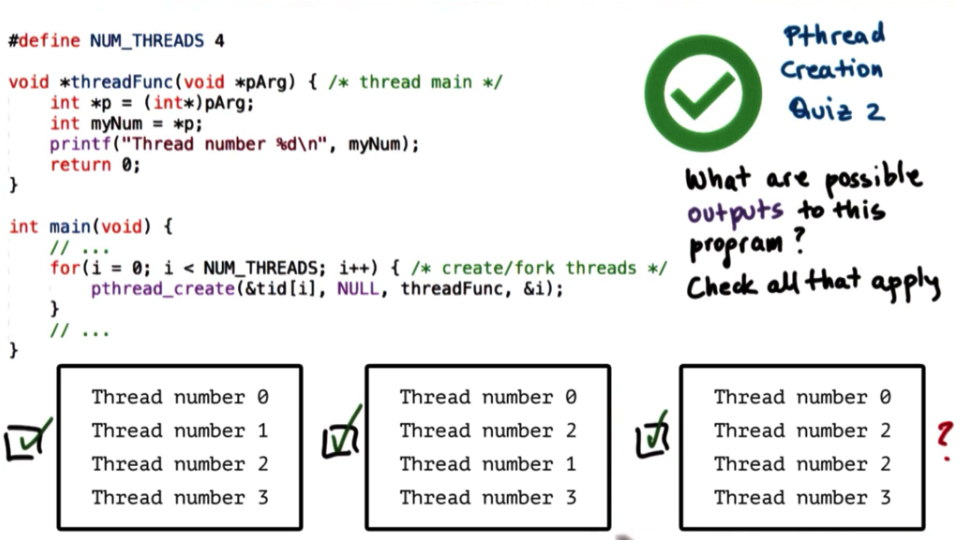
07 - PThread Creation Example 2

Let's look at a slightly different example. Here the threads need to execute a function, thread function. That's the function that's passed the pthread_create that takes in one argument. This is an integer argument and the function, what it does, it prints out thread number and then the number, the integer that was provided as an argument. The variables p and myNum are private to every one of the threads, so they are only valid in the scope of the thread function.Since we have multiple threads executing, four, every one of them will have its own private copies of these two variables, and they will potentially and in fact, likely be set to different values. When a thread is created, we see that the very first thing that happen are that it sets these private variables to values that depend on the input parameter. If you look at where the threads were created, we see that this input parameter, this argument is identical, that is, the index that's used in this loop. So once the thread sets these private variables, every one of them will print out a line, pthread number, and the value of the private variable, my number.

08 - PThread Creation Quiz 2

For this slightly modified example, what are the possible outputs? Instead of typing in your answers, here's some possible outputs, and you should check all that apply.

09 - PThread Creation Quiz 2



Feedback:

- Are you sure that you have included all options (exec order is not guaranteed)?
- Consider if a thread gets preempted in the middle of execution, what might this cause?

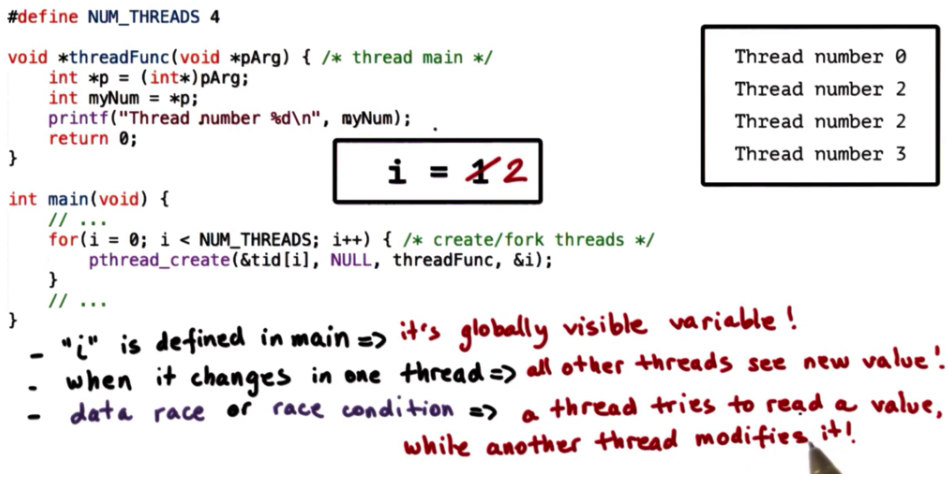
Hints:

- For the newly created threads, the order of execution is not guaranteed

The first output with sequential thread number 0, 1, 2, 3, is possible since I, whose values past this an argument to the thread creation function, has values that reach from 0 to 3. The next output, the print out, is a little bit arbitrary thread number 0, 2, 1, 3. But this is still possible because as we said earlier. We don't have control over how these newly created threads will be actually scheduled. So, it's possible that just the order in which their execution was scheduled, so the order in which every one of them performed the printf operation was slightly different that the order in which they were created. Now the last output that's actually also possible.Now, you may be asking yourself how since the print out thread number one, which appeared in the previous two options, doesn't even appear in this case. Is that an indication that that thread wasn't even created? If we look at this loop in main, we see that we must have really executed the printout operation for every one of the four created threads. So we really would expect that one of them would have printed out thread number one, when we pass the argument i equals 1. Let's explain what happened to that line in the next morsel.

10 - PThread Creation Example 3

From the previous quiz the problem is that the variable i that's used in this thread creation operation is a globally visible



variable that's defined in main. When its value changes in one thread, every one of the other threads will see the new value. In this particular case the second thread that was created in pthread_create was created with i equal 1. In the thread function, p will become equivalent to the address of i and myNum will then become equivalent to the actual value of i, so that's presumably 1. However, it is possible that before this thread had a chance to execute these operations and set the value of myNum to be 1, the main thread went into the next iteration of this for loop. And there it incremented i. So i is now 2. Since we pass as an argument the address of i, p will also correspond to the address of i. So it will point to the same i and then myNum will actually take as a value the new value of i so it will take as a value 2. So it's not like we lost the print out from that second thread that we were expecting with print out thread number 1, it's just that both the second and the third thread ended up seeing that the value of i is 2 and that's why then printing out thread number 2. We call this situation a data race, or a race condition. It occurs when one thread tries to read a variable that another thread is modifying. In this example the second thread that we created was trying to read the variable i, and we were expecting it that it would read i equal 1, however at the same time the main thread was modifying i, was incrementing it, and it became 2.

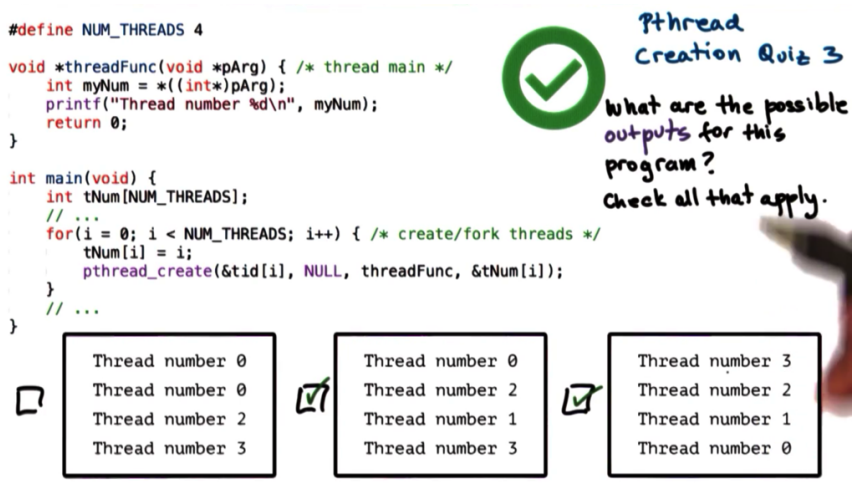
To correct the problem let's look at a slightly modified code here. We see that in the for looping main the value of i is first copied into an array. Into an element of an array tNum. The array has as many elements as there are threads and when we are creating a thread we pass as an argument the address of the particular element of the array that corresponds to that thread number. By creating this array then, it's like as if we created local storage, or private storage, for the arguments of every single one of the threads that we create. Now we don't have to worry about the ordering of how the new threads will execute the operations, because every one of them will have their own private copy of the input arguments that won't change.

```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]);
    }
    // ...
}
```

11 - PThread Creation Quiz 3

Now that we have fixed the error, we have one more quiz question. What are the possible outputs for tthis program? Here are your three choices. You should check all that apply.

12 - PThread Creation Quiz 3

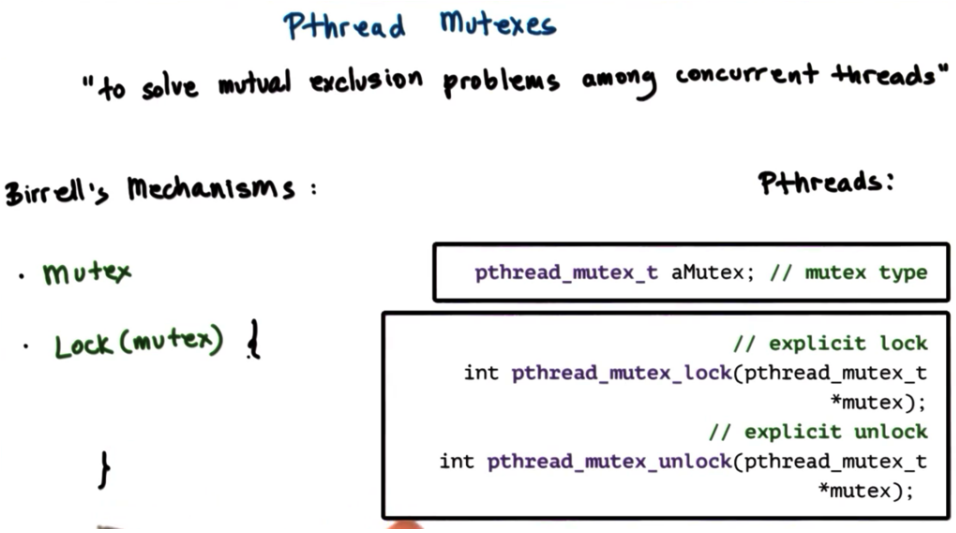


Feedback:

- The execution order is still not guaranteed, but will each value (0-3) be printed?

Now that we have fixed the error, and every one of the threads has its own private storage area to store the argument i, we expect to see the, line thread number, with the numbers 0, 1, 2, and 3 appear in the output. Given that, this first insert is not correct, and both of these two outputs, the second and third output, are correct answers to this question.

13 - PThread Mutexes



To deal with the mutual exclusion problem, pthread supports mutexes. As we explained when discussing Birrell's paper, mutexes provide a mechanism to solve the mutual exclusion problems among concurrent threads. Mutual exclusion lets us ensure that threads access shared state in a controlled manner. So that only one thread at a time can perform modifications or otherwise access that shared variable. Birrell proposed the use of the mutex itself and an operation to lock mutexes. In pthreads, the mutex data structure is represented via the pthread_mutex type. For the lock operation, remember that Birrell used the block construct where the critical section was protected by these curly brackets. Where the open curly bracket

meant that the mutex was being locked, and the closed curly bracket meant that the mutex was unlocked or free. In pthreads, this concept is supported explicitly, there is a separate pthread_mutex_lock operation and a separate pthread_mutex_unlock operation. Whatever code appears between these two statements will correspond to the critical section. As an example, remember that in the thread

Pthread Mutexes

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);
}
```

introductory lecture, we implemented the safe_insert operation using Birrell's construct in the following way. With pthreads, the same safe_insert operation would be implemented as follows, we would be explicitly be locking and unlocking the mutex around the insert operation in the shared list, my_list, and also note that the mutex is of appropriate type, pthread_mutex type.

Other Mutex Operations

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
// mutex attributes == specifies mutex behavior when
// a mutex is shared among processes
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

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

... many others ...

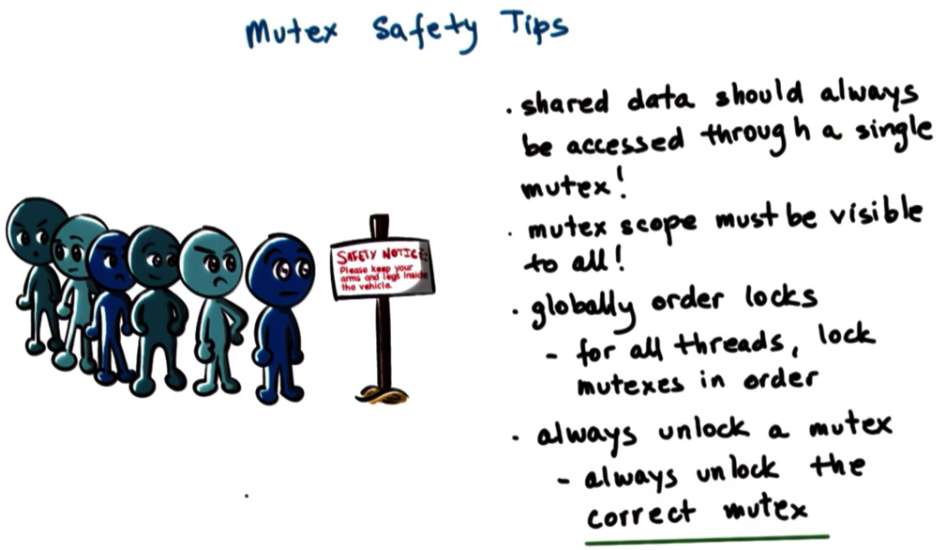
Pthread supports a number of other mutex related operations. Several of them are worth highlighting. First, mutexes must be explicitly initialized. This operation allocates a mutex data structure and also specifies its behavior. It takes as an argument a mutex attribute variable, and this is how we specify the mutex behavior. By passing now as this argument, we have an option to specify the default behavior from mutexes, or we can set one or more attributes that are associated with mutexes. For instance, pthreads permits mutexes and condition variables in general to be shared among processes. The default behavior would make a mutex private to a

process, so only visible among the threads within a process, whereas we can explicitly modify that behavior and make sure that the mutex can be shared with other processes. Another interesting operation is pthread_mutex_trylock. Unlike the lock operation which will block the calling thread if the mutex is in use, what trylock will end up doing is it will check the mutex, and if it is in use, it will actually return immediately, and it will notify the calling thread that the mutex is not available.

If the mutex is free, trylock will result in the mutex successfully being locked. But if the mutex is locked, trylock will not block the calling thread. This gives the calling thread an option to go and do something else and perhaps come back a little bit later to check if the mutex is free.

Also, you should make sure that you free up any pthread related data structures, and for mutex, for instance, you have the mutex_destroy operation. These are just some of the operations pthread support from mutexes. The ones we described here are enough to get your started with pthreads, and you can always refer to the pthreads documentation for information on the others.

In the previous lesson, we mentioned a number of common pitfalls where it comes to writing multithreaded programs. A few that are worth mentioning in the context of pthread mutexes include the following. Shared data should always be accessed through a single mutex. This is such a frequent error that it's worth reiterating. Next, the mutex scope must be visible to all

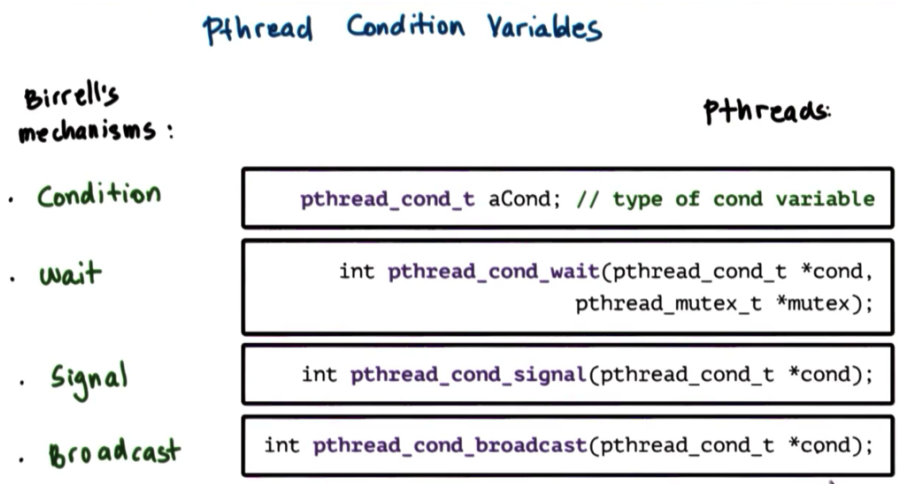


threads. Remember, a mutex cannot be defined as a private variable to a single thread, including main. You must declare all of your mutexes as global variables.

Another important tip is to globally order the locks. Once we establish an order between the locks, basically between the mutexes in the pthreads program, then for all threads we have to make sure that the mutexes are locked in that particular order. Remember, we said that this is a way to ensure that deadlocks don't happen. Finally, remember to always unlock a mutex. Moreover, make sure that you always unlock the correct mutex. Given that pthreads has

separate lock and unlock operations, it can be easy to forget the unlock, and compilers will not necessarily tell you that there is a problem with your code. So you have to make sure that you keep track of your locks and unlocks.

a14 - PThread Condition Variables



As was described in Birrell condition variables are synchronization constructs which allow block threads to be notified once a specific condition occurs. Birrell proposed the condition as condition variable abstraction as well as three operations. Wait, signal, and broadcast that can be performed on condition variables. In pthreads condition variables are represented via the designated condition variable data type. The remaining operations align really well with Birrell's mechanisms. For instance, for wait, pthread has a pthread condition wait that takes two arguments, a condition variable and a mutex, just like what we saw in Birrell's wait. The semantics of this operation is also identical to Birrell's wait. A thread that's entering the wait operation, a thread that must wait, will automatically release the mutex and place itself on the wait queue that's associated with the condition variable. When the thread is woken up, it will automatically reacquire the mutex before actually exiting the wait operation. This is identical to the behavior we saw in Birrell's wait. Identical to the signal and broadcast mechanisms in Birrell, PThreads has pthread condition signal and pthread condition broadcast, that we can use to either notify one thread that's waiting on a condition variable using the signal operation, or to notify all threads that are waiting on a condition variable using the pthread condition broadcast operation. There are also some other common operations that are used in conjunction with condition variables. These include the init and destroy functions. Pthread_condition_init is pretty straightforward, you have to use this operation in order to allocate the data structure for the condition and in order to initialize its attributes. Like what we saw with mutexes. The attributes can further specify the behavior that pthreads provides with conditions. For instance an example is whether or not the conditions variable will be used only within threads that belong to a single process or also shared across processes. And similar to what we saw with the mutex and threads attributes data structures. Passing null in this call will result in the default behavior that's supported by pthreads. That happens to be that the condition variable is private to a process. Just like threads condition variables should be explicitly freed and reallocated, we use the

condition destroy call for that. And finally, a few

Other Condition Variable Operations

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
// attributes -- e.g., if it's shared
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

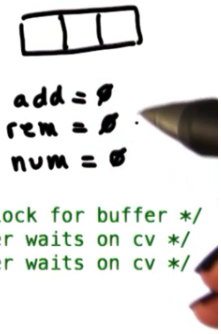
pieces of advice regarding the use of condition variables. First make sure you don't forget to notify the waiting threads. Whenever any aspect of a predicate that some threads are waiting on changes, make sure that you signal or broadcast the correct condition variables that these threads are waiting on. Next, if you're ever in doubt whether you should use signal or broadcast, use broadcast until you figure out what the desired behaviour is. Note that with broadcast you will lose performance. So, make sure you use the correct notification mechanism, signal or broadcast, when you need to wake up threads from a condition variable. Remember, since you don't actually need the mutex to signal and broadcast, it may be appropriate for you to remove that signal and broadcast operation until after you've unlocked the mutex, just like what we saw in the introductory lecture about threads. We will point out some of these options during the discussion of an actual pthreads example that we'll do next.

Condition Variable Safety Tips

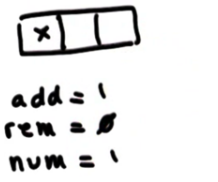
- do not forget to notify waiting threads!
  - predicate change => signal/broadcast correct condition variable
- When in doblt broadcast
  - but performance loss
- You do not need a mutex to signal / broadcast

15 - Producer and Consumer Example Part 1

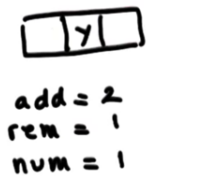
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#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
#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
#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);
```



16 - Producer and Consumer Example Part 2

```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");
}
```

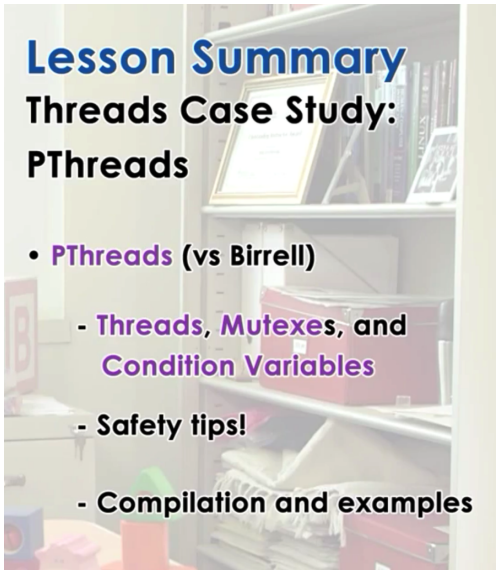
17 - Producer and Consumer Example Part 3

```
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;
}
```

```
void *consumer (void *param) {

    int i;

    while (1) {

        pthread_mutex_lock (&m);
            if (num < 0) { /* underflow */
                exit (1);
            }
            while (num == 0) { /* block if buffer empty */
                pthread_cond_wait (&c_cons, &m);
            }
            i = buffer[rem]; /* buffer not empty, so remove element */
            rem = (rem+1) % BUF_SIZE;
            num--;
        pthread_mutex_unlock (&m);.

        pthread_cond_signal (&c_prod);
        printf ("Consume value %d\n", i); fflush(stdout);
    }
}
```

(

18 - Lesson Summary



w