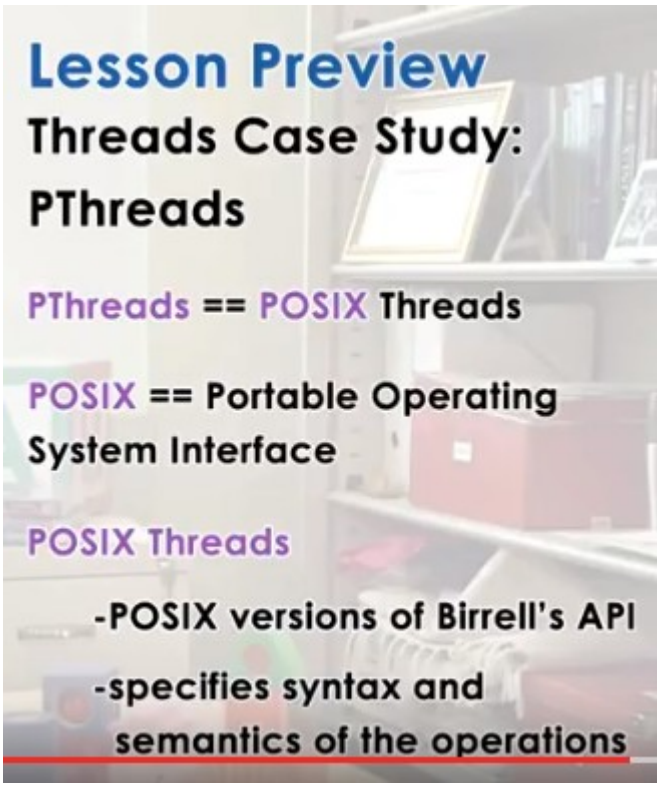
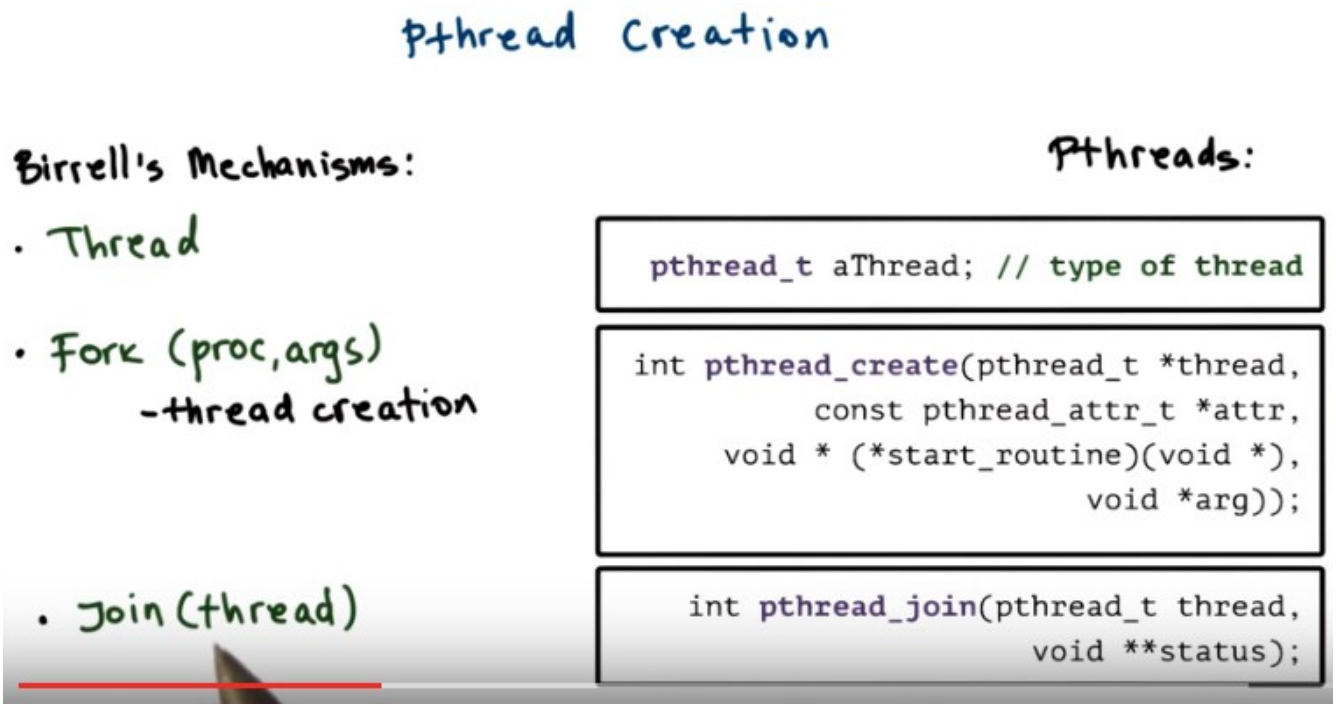
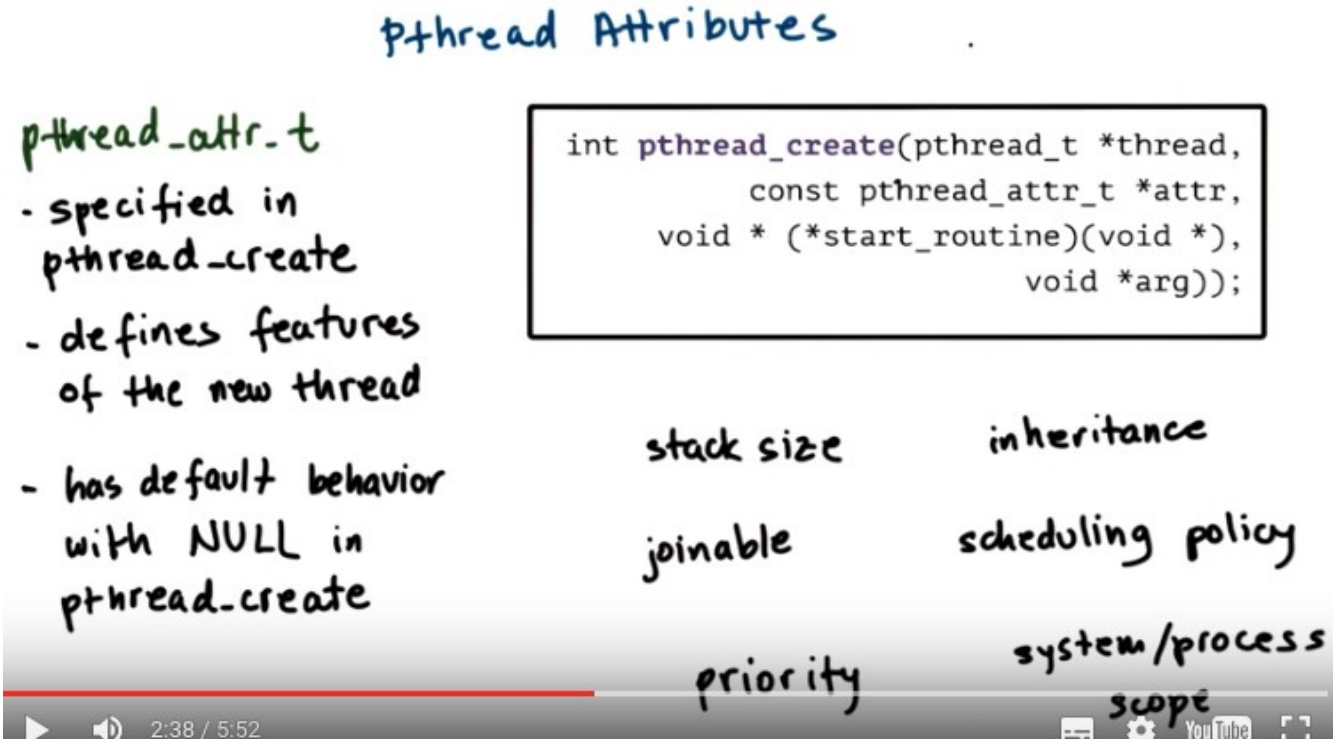


1. The last paper talked about threads in a generic way. It described multithreading concurrency synchronization in more generic terms. In this lecture, we will talk about PThreads, which is a very concrete multithreading system, and it's the de facto(實際上) standard in UNIX systems. First off, PThreads stands for POSIX Threads. POSIX stands for Portable Operating Systems Interface, and it basically describes the interface, the system call interface, that operating systems need to support. And its intent is to increase interoperability among OSes. Within POSIX, PThreads describes the thread and correlated API that operating systems need to support, in order for creating and usage and management of threads. And this includes both the threads themselves, as well as the synchronization and concurrency related construct such as mutex and condition variables.



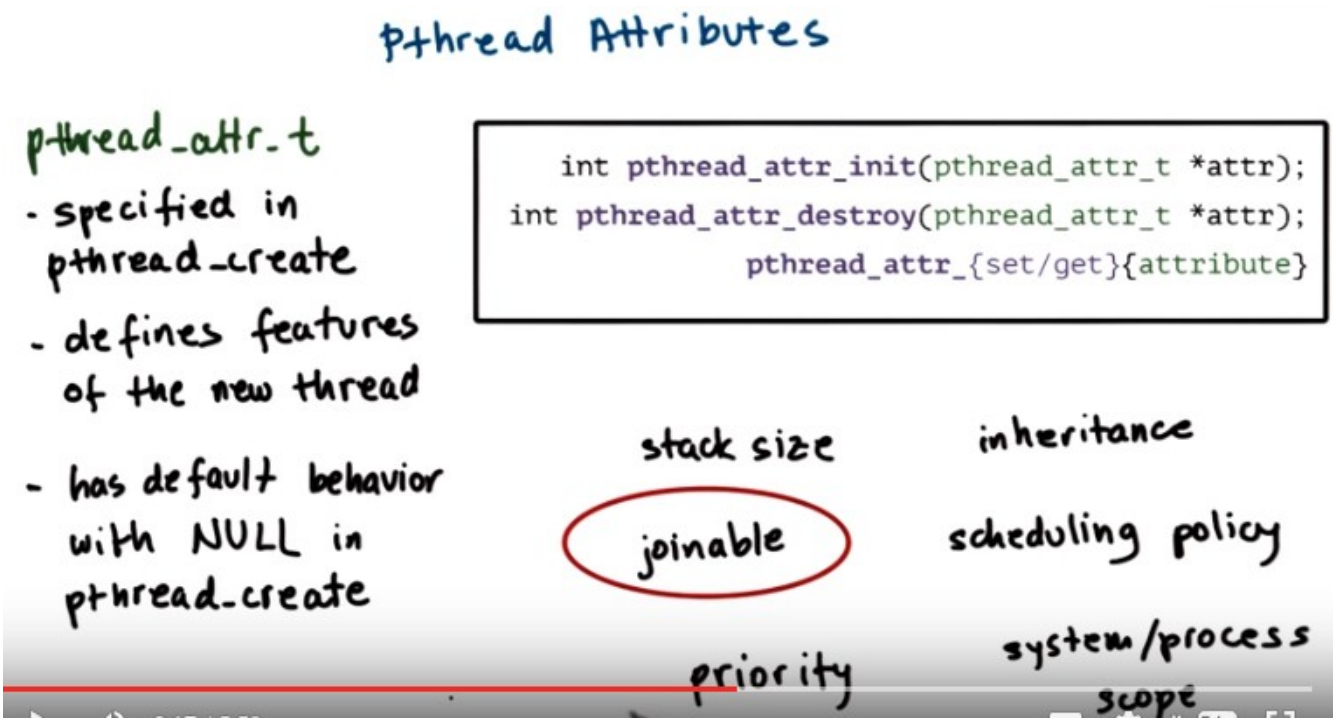
上圖中 pthread_create 的最後一個參數 arg 是 start_routine 這個函數的參數, 若 start_routine 函數無參數, 則 arg 為 NULL. 參見第 4 節和第 7 節. pthread_t 中的 _t 應該就是 type 的意思, 參見第 13 節的 pthread_mutex_t. 另外, void **status 是指向指針的指針, 參見 C 語言 p171.

2. First, let's look at the Pthread's thread abstraction and the thread creation mechanism that corresponds to the mechanisms that were proposed by Birrell. Birrell proposed the Thread abstraction and the Fork and join mechanisms that work with that abstraction. To represent threads, Pthread supports a pthread_t data type. That's the thread data type. Like Birrell thread data type, variables of this type will uniquely be identified with an identifier and will describe a thread, in this case a Pthreads thread. They'll have an ID, execution state, any other information that's relevant to the thread. Most of this is not something that you will directly see as a developer. Instead this is information that's used and maintained by the Pthread's library. The equivalent to the Fork that Birrell proposes is a more intuitive called pthread_create. It takes as arguments a start_routine. That's the equivalent to proc and Fork, as well as arguments equivalent to the arguments in Fork. It creates as a result and returns a new data structure that's of the type pthread type. And it's populated with all the relevant information so that this thread can start executing. It also takes as an input, a variable of type pthread attribute type. And this is a data structure that you can use to specify certain things about the thread that the pthreads library will need to take into consideration when managing the thread. We'll look at this in more detail shortly. Pthread_create also returns status information that indicates whether the creation was successful or a failure. Last, the alternative to Join is pthread_join. It takes two parameters, the thread structure, that's the thread that need to be joined, as well as the status variable. The status variable will capture all of the relevant return information, as well as the results that are returned from the Thread. The overall operation also returns a status that indicates whether the Join was successful or it failed. As you can see, these operations are fairly analogous to the operations that were proposed by Birrell.

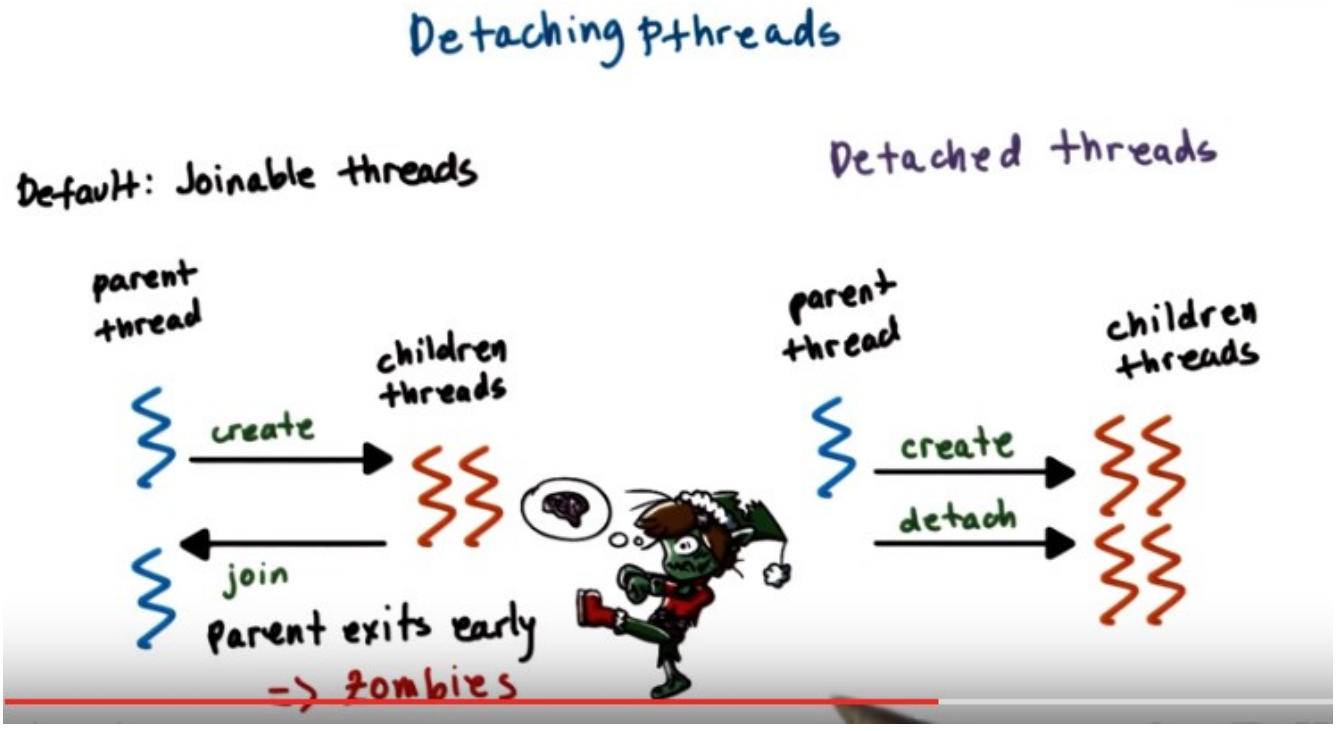


上圖中的 void * (* start_routine) (void *)實際上是一個指向函數的指針, 函數名為 start_routine, 返回值為 void *, 參數列表為(void *), 若省略參數列表, 則應寫為 void * (* start_routine), 這下就清楚多了 .

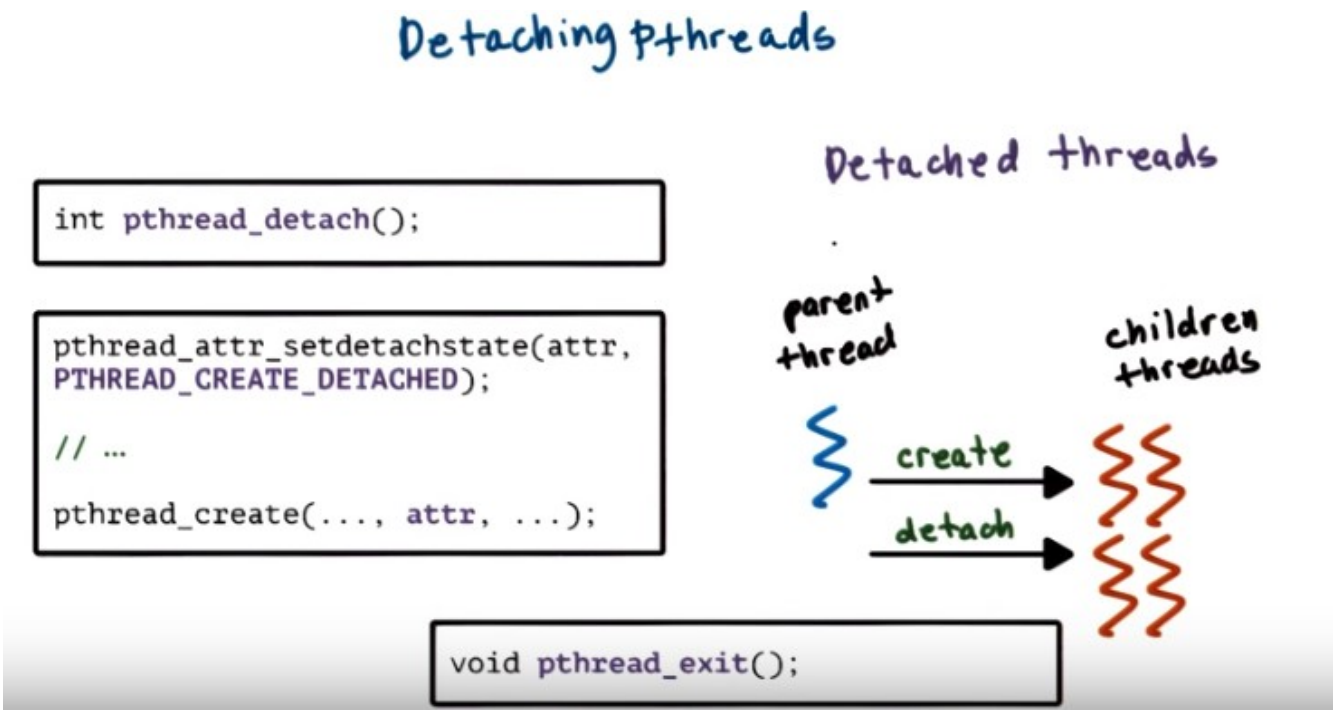
I would now like to talk, in a little more detail about Pthread Attributes. As we previously saw, there is a Pthread attribute-type argument to the pthread_create function. This argument gives us flexibility to specify certain features of the newly created thread. For instance, we can specify the stack size, scheduling policy or priority of the new thread. We can set the scope of the new thread, whether it's system or process, like what we explained in the previous lesson. Whether the threads should inherit attributes from the calling thread, whether it's joinable. Pthread establishes some default values for all of these parameters. And if you pass NULL as an argument to the the pthread_create function, you will achieve that default behavior.



There's several calls that support operations on Pthread Attributes. Pthread attribute init or destroy allow us to create and initialize the attribute data structure. Or to destroy it and free that data structure from memory. Pthread set/get allows us to either set the value of an attribute field or to read that value. One of these attributes requires particular attention, and that's joinable. To explain this, I will first need to explain some mechanisms that are supported by Pthreads, but were not considered by Birrell. The mechanism not considered by Birrell is detachable threads.



In Pthreads, the default behavior of thread creation is just like what Birrell described. The threads are joinable. With joinable threads the parent thread creates children threads, and can join them (children join parent, 見上圖中箭頭方向) at a later time. The parent thread should not terminate until the children threads have completed their execution and have been joined via the explicit join operation. If the parent thread exits early, the children threads can turn into zombies, because they may have completed or died, but not exited or have not been reaped properly. In Pthreads there is a possibility for the children threads to be detached from the parent. Once detached, these threads cannot be joined. If a parent exits, these children are free to continue their execution. This really makes the parent and the children equivalent to one another, with the exception that the parent threads have some additional information on the children that they have created.

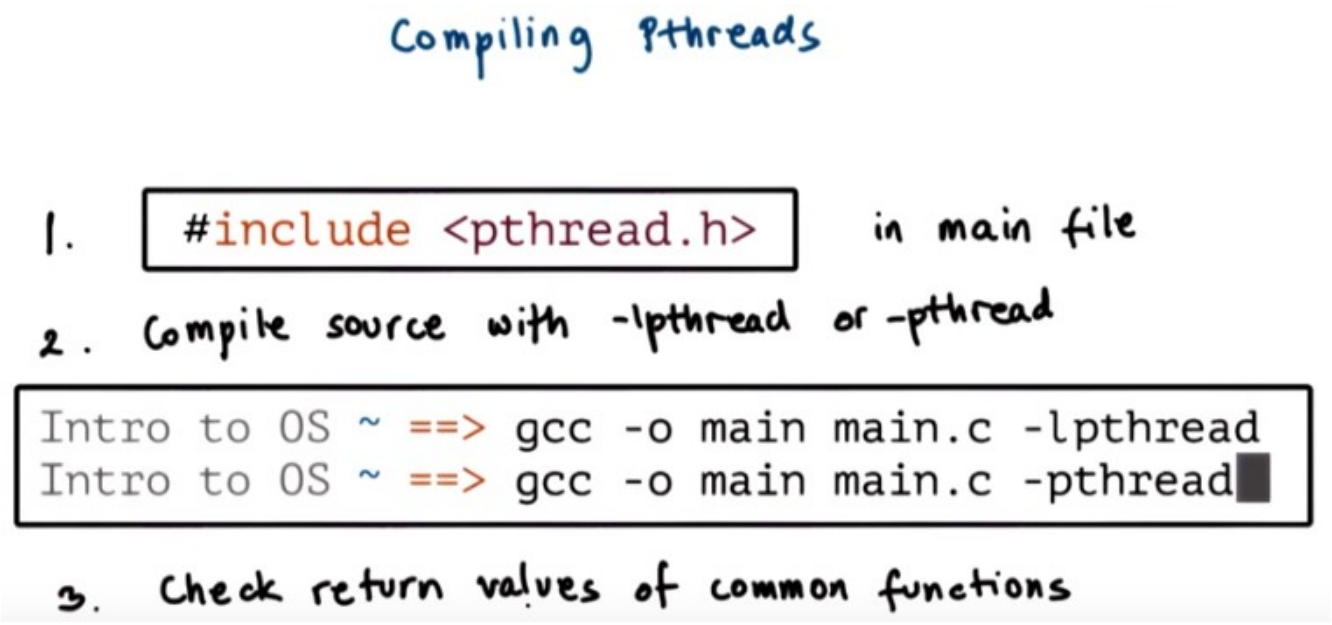


To detach threads, Pthread provides a pthread_detach operation. That takes the thread data structure, the thread that needs to be detached as an argument. Threads also can be created as detached threads using the attribute detach state. The value of this attribute first needs to be set to PTHREAD_CREATE_DETACHED, because otherwise the default behavior for Pthreads is for threads to be created as joinable threads. So with DETACHED thread since the parent thread doesn't need to stick around to wait for the children threads to complete, it can simply exit using pthread_exit.

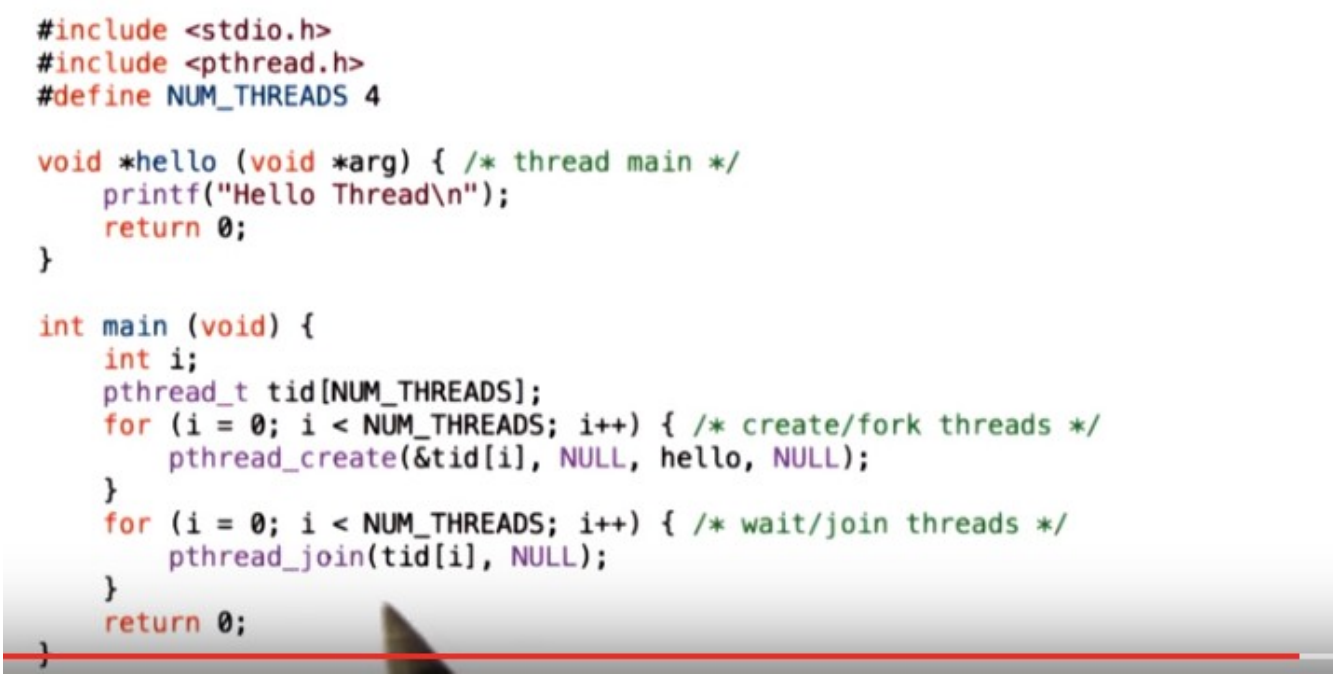
```c
#include <stdio.h>
#include <pthread.h>

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;
}
```

udacity 中指出上圖中有個 typo: 倒數第二句應為 pthread_create(&tid, &attr, foo, NULL);

Here is an example of using pthread attributes. First the pthread attribute data structure must be created and initialized. This will allocate the data structure with sufficient memory, and it will set its values to the default pthread parameters. We can also adjust the values of the attributes using calls like pthread_attr_setdetachstate or setscope, or which ever other attributes we want to adjust. For instance, here we are setting the detach state to be PTHREAD_CREATE_DETACHED like what we described just a minute ago. And we're setting the scope, the scheduling scope of the threading system, to be the system scope. This means that the newly created thread will share resources equally with all other threads in the system. Once the attributes have been initialized and set to the desired values, the resulting data structure is passed to the call pthread_create



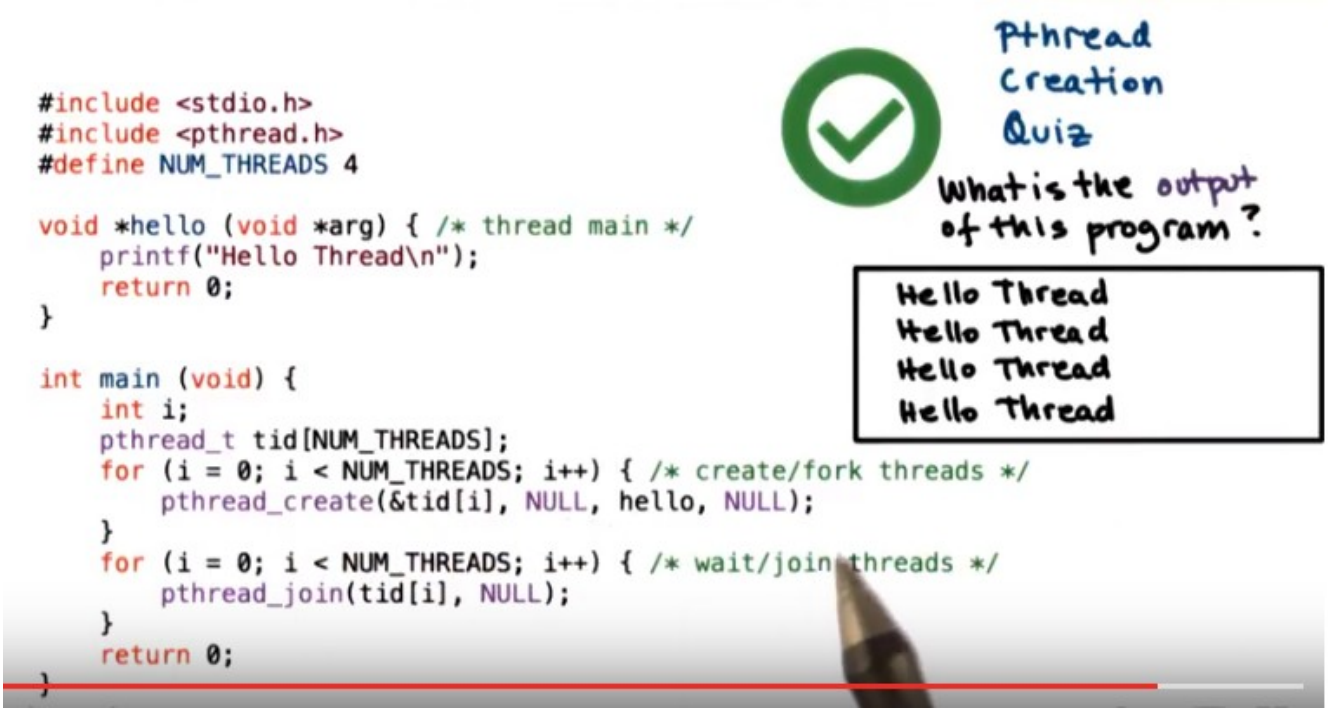
3. Before we look at some examples, there are few things that we need to consider when compiling threads. First make sure to include the pthread header file, pthread.h, in your main file that contains the pthreads code, otherwise your program will not compile. Second, make sure to link your program with 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 [INAUDIBLE] of writing multithreaded programs.



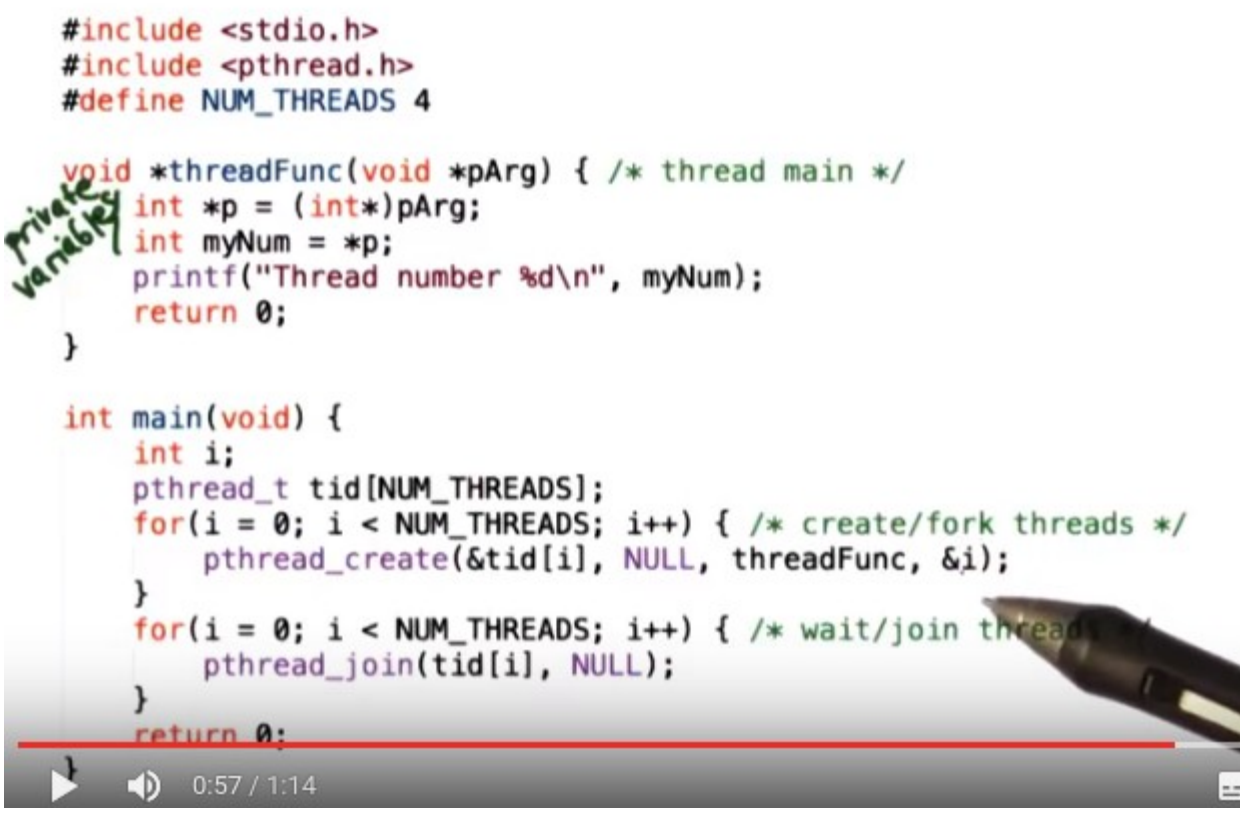
```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

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;
}
```

4. 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 (這四個 threads 是被 parent thread create 出來的, main 函數就是那個 parent thread, parent thread 的作用就是專門 create 四個 threads 讓它們跑. 參見本 note 的第 16 段之紅字, 或 P2L2 的第 11 段). All this function does is print Hello Thread, it doesn't take any arguments and that's why we pass NULL as a 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.

5. So, as a quick quiz, what do you think will be the output of this program? Type your answer in this text box.

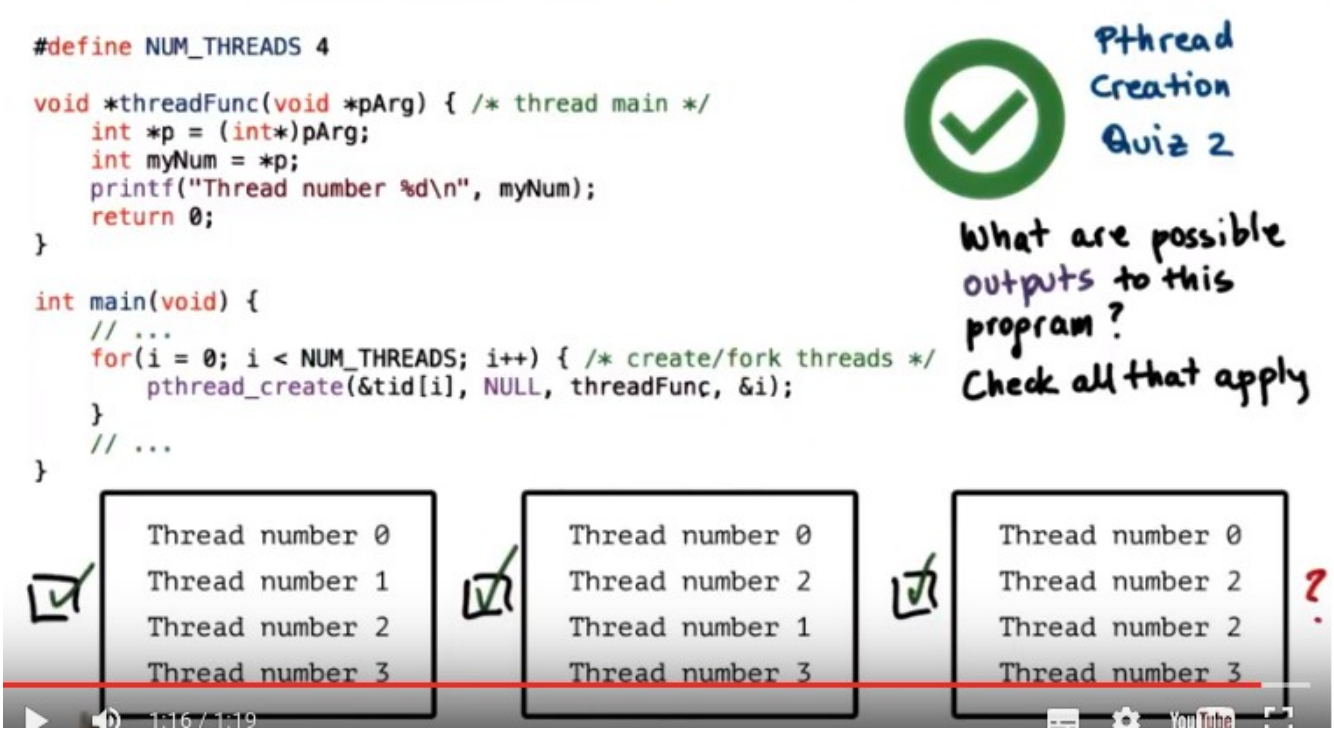


6. The end result here is straightforward. The string hello thread, will appear four times, will be printed four times once by every one of the threads we've created.



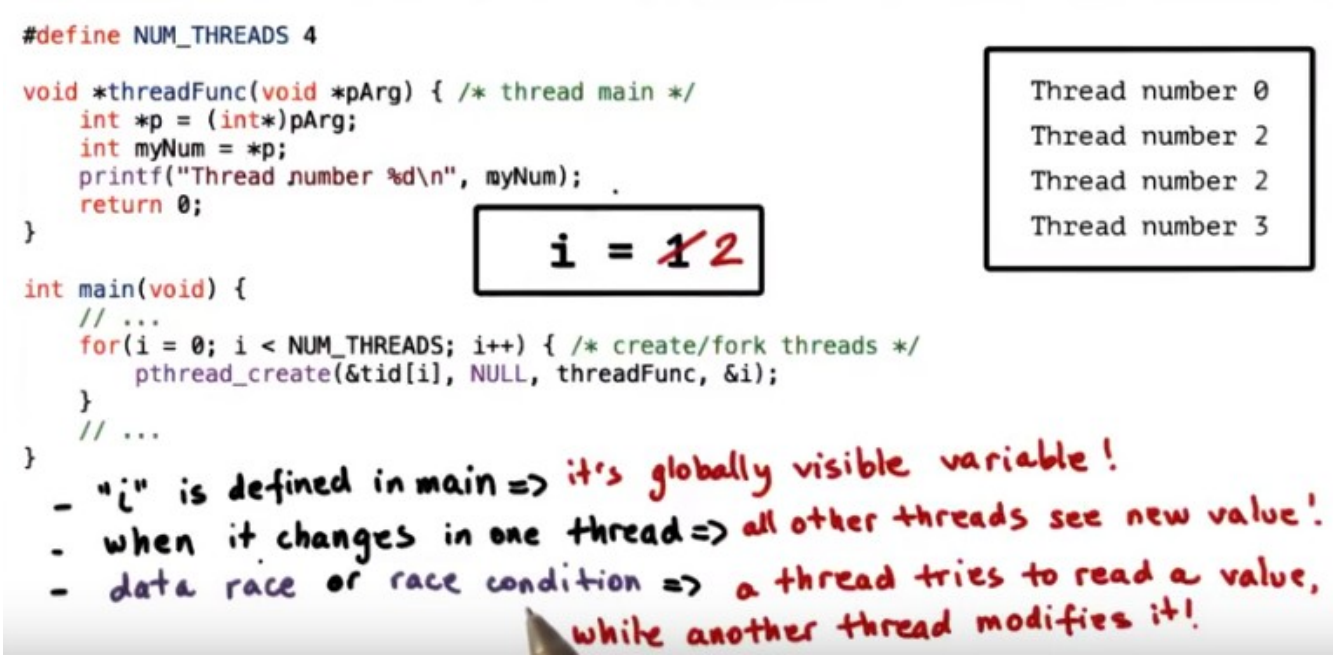
7. Let's look at a slightly different example. Here the threads need to execute a function, thread function. That's the function that's past the pthread create that takes in one argument. This is an integer argument and the function, what it does, it prints out thread number and then the number, the 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.

8. 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.



9. 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 1, 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.

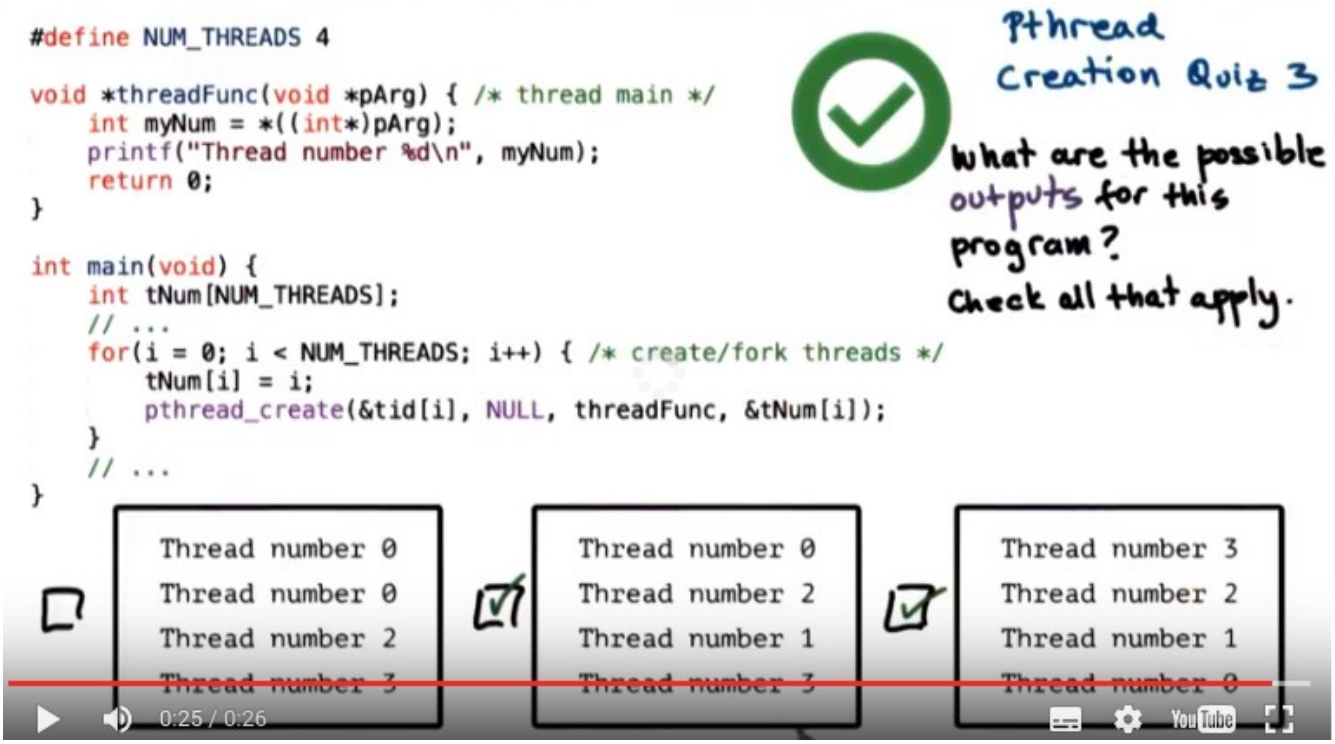


10. From the previous quiz the problem is that the variable i that's used in this thread creation operation is a globally visible variable that's defined in main. When its value changes in one thread, every one of the other threads will see the new value. In this particular case the second thread that 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 passes 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.

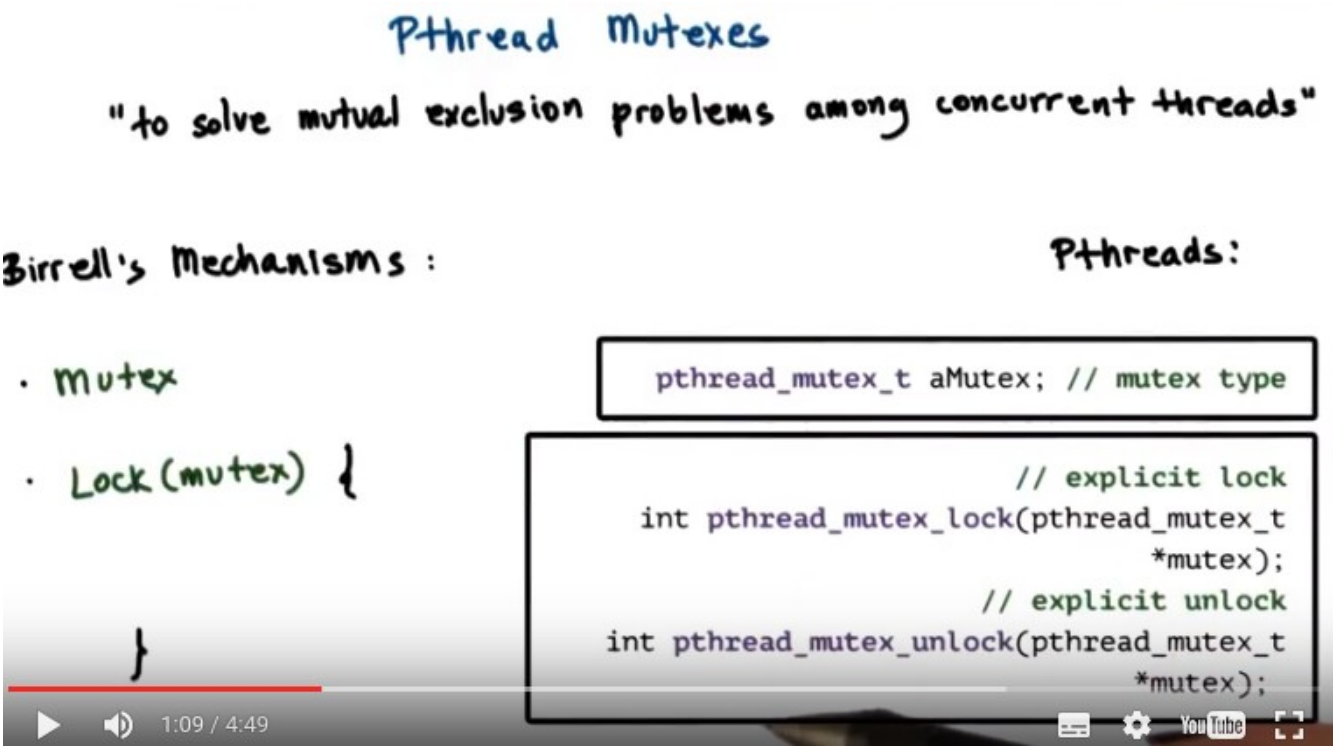
```c
#define NUM_THREADS 4

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

To correct the problem lets 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 s 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.

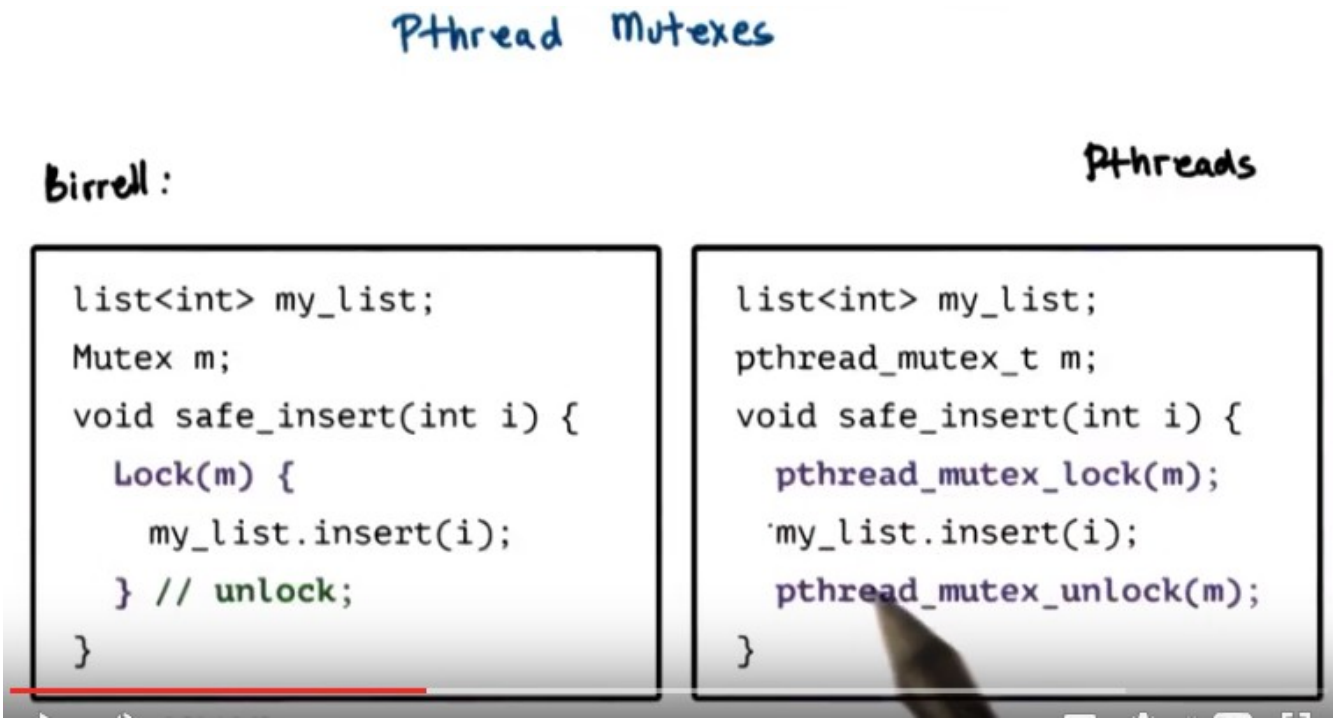
11. Now that we have fixed the error, we have one more quiz question. What are the possible outputs for this program? Here are your three choices. You should check all that apply.



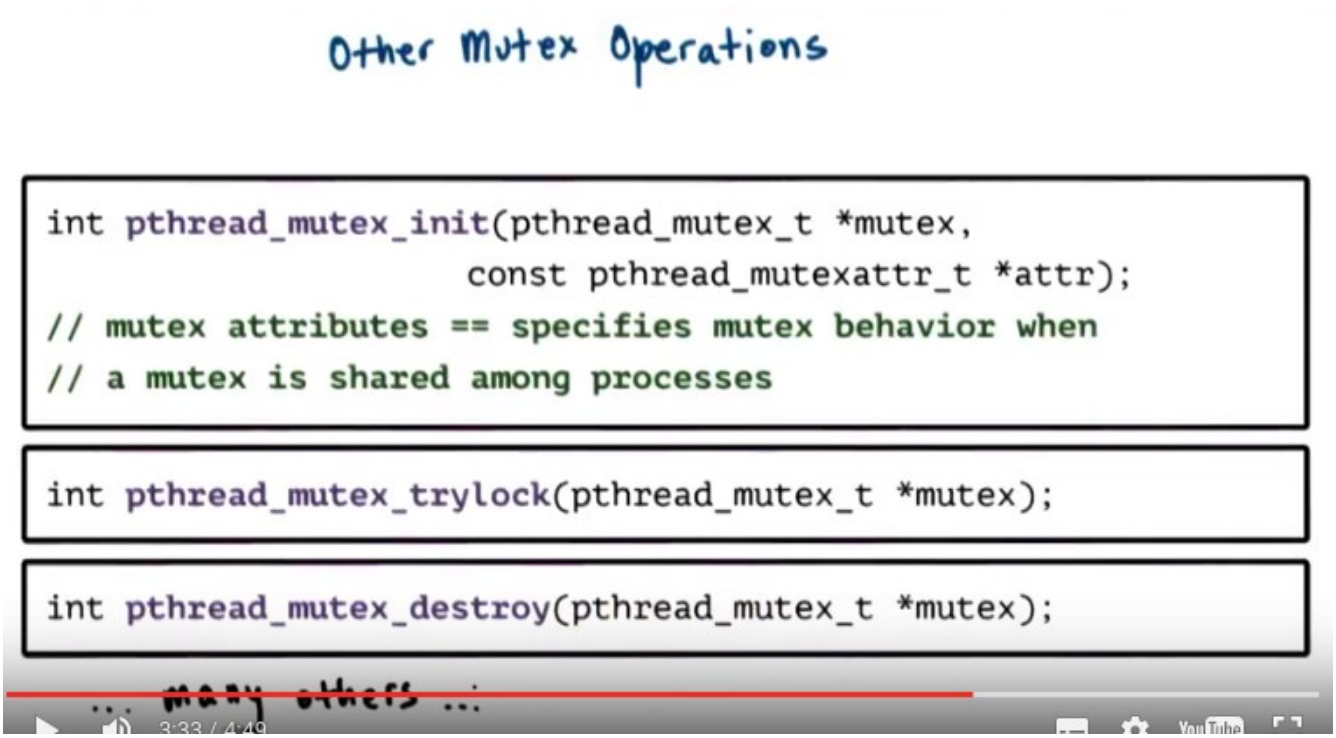
12. Now that we have fixed the error, and every one of the threads has its own private storage area to store the argument i, we expect to see the, line thread number, with the numbers 0, 1, 2, and 3 appear in the output. Given that, this first insert is not correct, and both of these two outputs, the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



13. 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 pthread mutex type (由此可知, pthread_mutex_t 中的 t 就是 type 的意思). For the log 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.

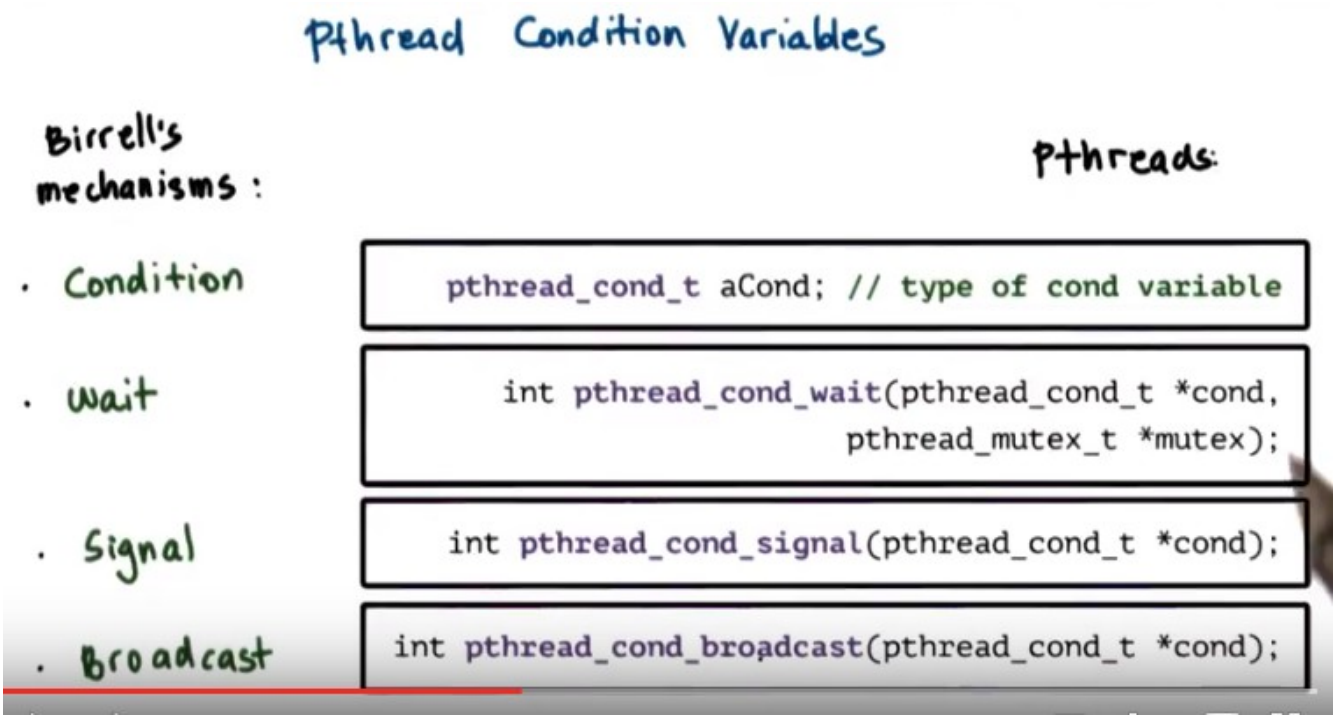


As an example, remember that in the thread introductory lecture, we implemented the safe_insert operation using Birrell's construct in the following way. With pthreads, the same safe_insert operation would be implemented as follows, we would be explicitly be locking and unlocking the mutex around the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



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 NULL 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 clocked. 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 dead locks 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.



14. As with 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. Weight, signal, and broadcast that can be performed on conditioned 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 weight, pthread has a pthread condition weight that takes two arguments, a condition variable and a mutex, just like what we saw in Birrell's weight. 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 re-acquire 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.

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);
```

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 straight forward, you have to use this operation in order to allocate the data structure for the condition and in order to initialize it's 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.

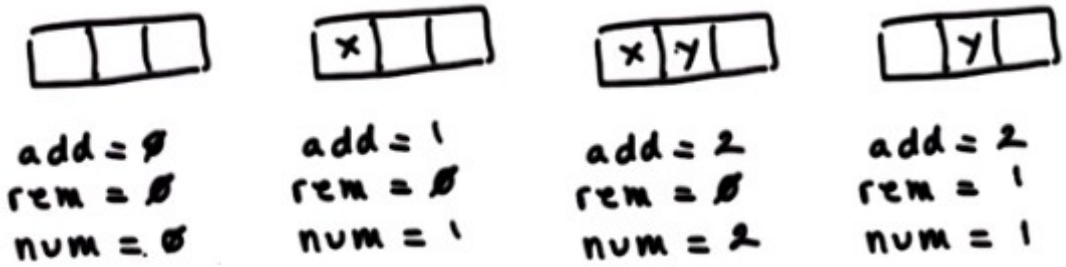
Condition Variable Safety Tips

- 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

And finally, a few pieces of advice regarding the use of condition variables. First make sure you don't forget to notify the waiting threads. Whenever any aspect off a predicate that some threads are waiting on change, 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 untill 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.

Producer - Consumer
Example in Pthreads

```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);
```

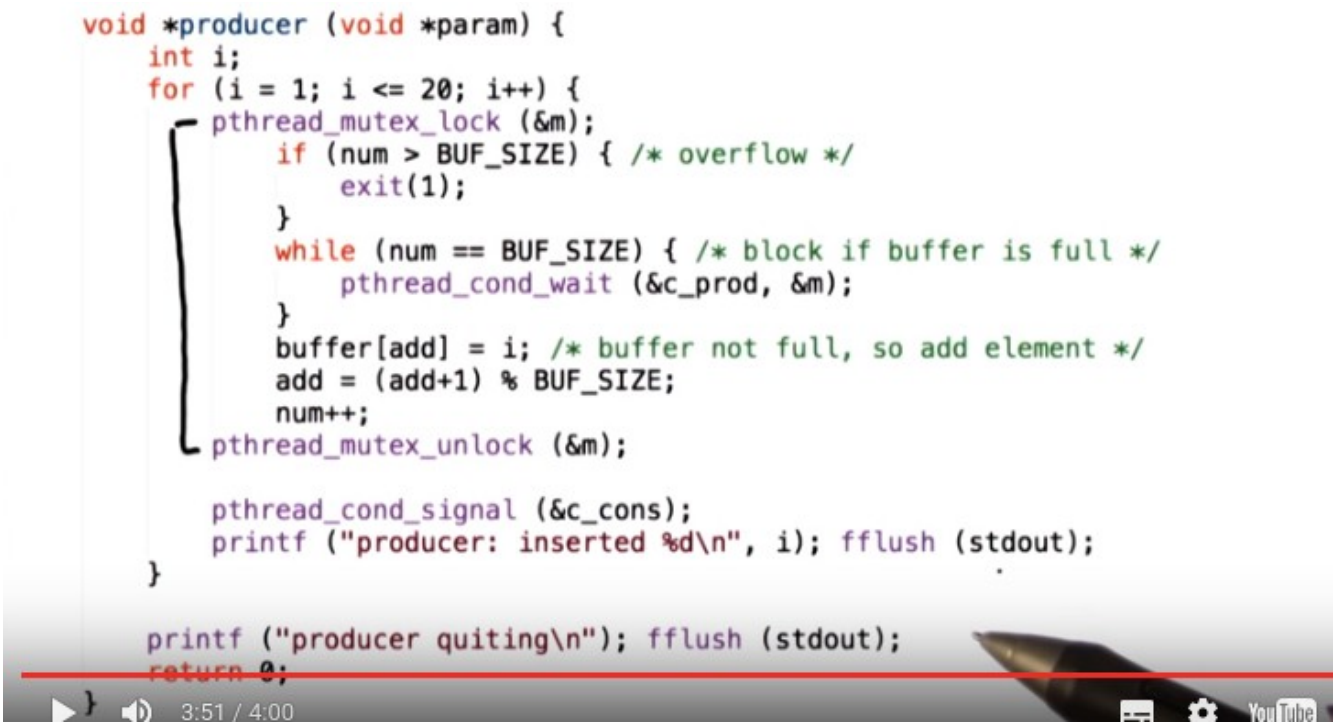


15. Now to tie everything together, we will look at an implementation of the classic producer-consumer problem that uses the pthreads library. We will look at the source code section by section. In this first, for instance, section this is the global scope where all of the different variables are defined. If you happen to get lost as I trace through this code, then please reference the source code link that's provided in the instructor's notes. So let's take a look at this. In this producer-consumer example, we have a shared buffer of size buffer size, and it happens to be three. There are three also shared variables where num refers to the number of elements in the buffer, and then add and rem refer to the indices that point to the element in this buffer where we need to add the next element or to remove the next element from. For instance if this is our shared buffer initially all of these variables would be zero. 這個 buffer 數組其實就是個 queue: 新元素加到右邊, 刪除元素時刪左邊的. When we add one element, that means that the total number of elements is one. Adding new elements will have to happen in the next field in the buffer array. And removing an element still remains to be zero, because this is the element we need to remove. Adding a second element changes the values of these shared variables as follows. So the total number will be two. And then new additions should be placed in the buffers of two element. And we still haven't removed anything. When we remove one element from the buffer, the x, that means that the total number of elements in the buffer is now one. Still, the next available slot in the buffer, that we can use to add an element, is two. And the slot that contains the next valid entry in the buffer that we should be removing next is one, the y. So, this illustrates how this buffer and these three shared variables are used to manage the producer consumer data. The shared variables are used in conjunction with a mutex, and we use this mutex initializer statement, that this basically automatically initializes the mutex. So it does the function of attribute init, essentially. And we're going to use two condition variables. One, c_cons, which will be used by the consumers, so the consumers will be waiting on this condition variable (這就是 condition variable). And then the other one, c_prod, and this one will be used by the producers. The producers will be waiting on this variable. There are also two functions, two procedures, the producer operation that will be executed by our producer threads, and the consumer operation and that will be executed by our consumer threads.

```
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");
}
```

由上圖的注釋可以看出 join 之作用(注意不要把 exit 看成 execute 了). 注意上圖中用 pthread_create()建了兩個 thread, 這兩個 thread 分別運行 producer 和 consumer 這兩個函數. 後面那兩個圖便是 producer 和 consumer 這兩個函數.

16. Now let's look at the main portion of the code of this producer consumer implementation. We'll be creating two threads, thread id1 and thread id2. The first thread will be created to execute the producer function, and then the second thread will be created to execute the consumer function. We're using the default behavior for these, so we will have to join them later in the main function, in the main thread. And we will look at the producer and consumer functions next, but they don't take any input so we're pasisng NULL as arguments. Note how we're checking for the return code of the pthread_create operation in order to help with debugging. So the thread, the main thread, the one that's executing command, will just create the producer and consumer threads, and then it will do nothing. It will wait for them to exit.

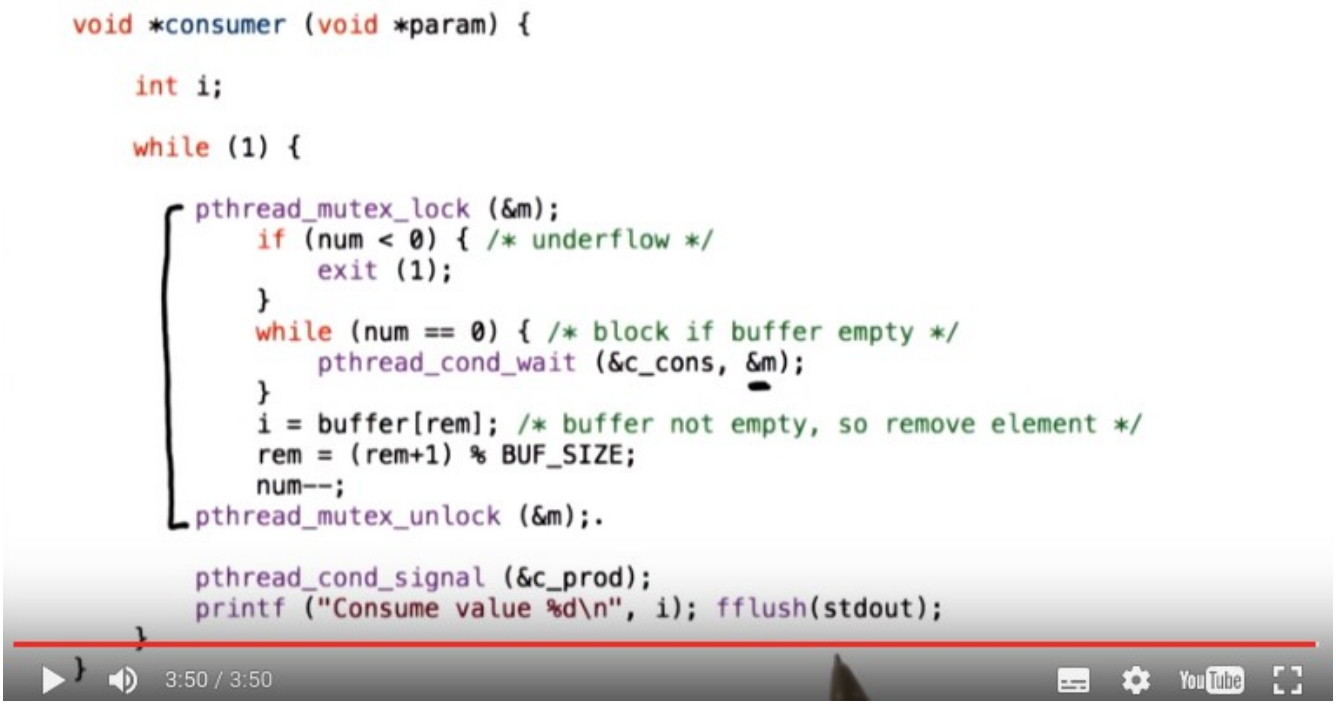


上圖中的 pthread_cond_wait(c_prod)中的 c_prod 是在第 15 節的代碼中定義的 condition variable. 當 consumer 用 pthread_cond_signal(&c_prod)來 signal 後, 上圖中的 producer 就可以跳出這個 wait. 而 Mutex m 的作用就是 lock, 比如上圖中, 當別人(consumer) acquire 了這個 mutex 後, 上圖中的 producer 就不能執行黑線包括的那段.

上圖中為何要在 for 之外定義 int i? 這是因為我在我電腦上也遇到過,說是 for(int i, ...)是某某標准, 我的編譯器不認.

The producer functions of the producer thread will try to execute for 20 times a loop in which it tries to produce an element for the shared buffer. During each pass through this loop, the producer thread will be trying to modify the shared buffer to add an element in that buffer, and then also to change the values of the shared variables, like add a num. Therefore, all of this has to happen within a mutex_lock, mutex_unlock operation. We first do some error checking to make sure that we don't have buffer overflow. Now we're trying to insert, we're trying to produce data for the shared buffer. If the number of elements that are currently in the buffer is equal to the buffer size, that means that the buffer is full. So we have to wait, we have to wait on the condition variable that associated with the producers, and this wait operation we have to use the mutex as part of it. Remember, this mutex has to be used as an argument of the wait call so that the pthreads library knows which particular mutex needs

to be freed and then reacquired after we complete the wait. Now, when we ultimately come out of the wait, so when the producer indeed comes out of the wait operation because the buffer is no longer full, so a consumer must have consumed some of the items in this buffer, then what do we do? Then the producer adds an element in the buffer, so it copies the value of i, this index i, copies the value of i into the element of the buffer that is indexed by the value of add, and then increments both the add, a variable, as well as the num, the total number of elements in the shared buffer. Note we may have a wrap around situation (即那個取%運算, 跟 leetcode wrtie4 中一樣). So, given that the buffer is a fixed size, buffer size, this add, we need to wrap around in case it becomes greater than buffer size. Once we perform this, then we're done. We've inserted an element. We've updated the variables in such a way that it reflects that now there is a new element in the buffer. And so, we can unlock the mutex. Now, the one thing that we didn't do is we didn't do any kind of signaling or broadcasting while we were performing this. It is possible that the buffer was empty when we performed this insert operation, and because of that, that currently there is some consumer threat that's waiting on a condition. We've inserted just one element. Therefore, only one consumer thread can proceed. So what we'll do is we will notify a thread that's waiting on the condition variable, and we will use for this the signal operation pthread condition signal because again, we inserted one element, no point waking up seven consumer threads. For sanity, I've inserted here some printout statements that will help you keep track of what's going on. These are not critical to the behavior of the multithreaded program.



上圖中的 rem 見第 15 節的代碼. i 的作用只是 printf.

17. Now if we look at the consumer code, so this is the code that's executed by every one of the consumer threads. There, what every one of the threads needs to do, it's going to, in a loop, so in a continuous loop, it will try to remove elements from the shared buffer. In every pass through this loop, the consumer thread will try to remove an element from the buffer and update the rem and num variables accordingly. To these kinds of modifications we have to lock the mutex, and then once we're done we have to unlock the mutex. Again, like in the producer case, we do some sanity error checking to make sure that our buffer doesn't have a negative number of elements. And then before we actually start removing elements from the buffer, we have to make sure that there are any elements in the buffer

to begin with, that the buffer isn't empty. If the buffer is empty, so if this variable num that indicates the total number of elements in the buffer is 0, the consumer thread has to wait. The wait is associated with the consumer condition variable. Remember this is the condition variable that the producer thread was signaling to, and also with this wait we have to use the mutex that protects this piece of code. Once we have successfully completed the wait part, so a producer has generated more data, the consumer has been notified, it has come out of the conditional wait, and yes indeed, re-verified that now there is data in the buffer that can be consumed. Then we can safely move on to execute this portion of the code. To remove an element from this buffer, what we really do is we read out the element of the buffer that's pointed by the rem variable. So rem points to the next valid field in the buffer where there is valid data that could be removed. We also make sure to print out that value i, just for sanity, to make sure that we're doing the correct thing. Now that that element that was pointed by rem is no longer valid, we have to make sure we increment rem to point to the next entry in the buffer. Just like in the case with the add index, we have to do some modular computation to deal with wrap-arounds around the end of the buffer. And we have to also make sure we decrement the total number of elements in the array. Now in the producer code, the producer was checking the value of this variable num against buffer size to determine whether or not the buffer is full. Whenever you determine that these are identical so that the buffer is full, the producer was waiting to be notified on a condition variable that's associated with the c producer. Therefore, in the consumer code, now that we've decremented this, variable num, now that we have consumed an element from the buffer and it's no longer full. We need to go ahead and notify a producer thread that there is now room in the buffer to insert other elements, to produce more data. The consumer consumed one element of this buffer, so it makes no sense to broadcast. Only one producer can insert one element in the buffer. Therefore, we will just use pthread_cond_signal. And note how here we are using this pthread_cond_signal outside of the lock-unlock operation. Given that we will always signal once we complete this code that this operation is not conditional upon some specific values of any of the shared data structure, we can release the mutex and then signal, and this will avoid the spurious wakeups issue that we talked about before. Again, the printf's are for debugging purposes.

18. In this lesson we took an in-depth look at the PThreads library. We talked about PThreads Creation,

Mutexes, Condition Variables, and we compared these constructs to the constructs that were proposed in Birrell's paper. We spent some time talking about common practices when it comes to PThreads, as well as some safety tips. And then we looked at how to compile PThread programs. And then walk through several programming examples.

19. As the final quiz, please tell us what you learned in this lesson. Also, we'd love to hear your feedback on how we might improve this lesson in the future.