Tutorial 3 - pthread

COMP3230 Principle of Operating System

Huang Songlin huangs0@hku.hk, 2023/10/12

Table of Content

POSIX Thread

POSIX := Portable Operating System Interface of UNIX

- High-Level Design & Thread Lifecycle
- Creation
- Thread Function
- Termination & Collection

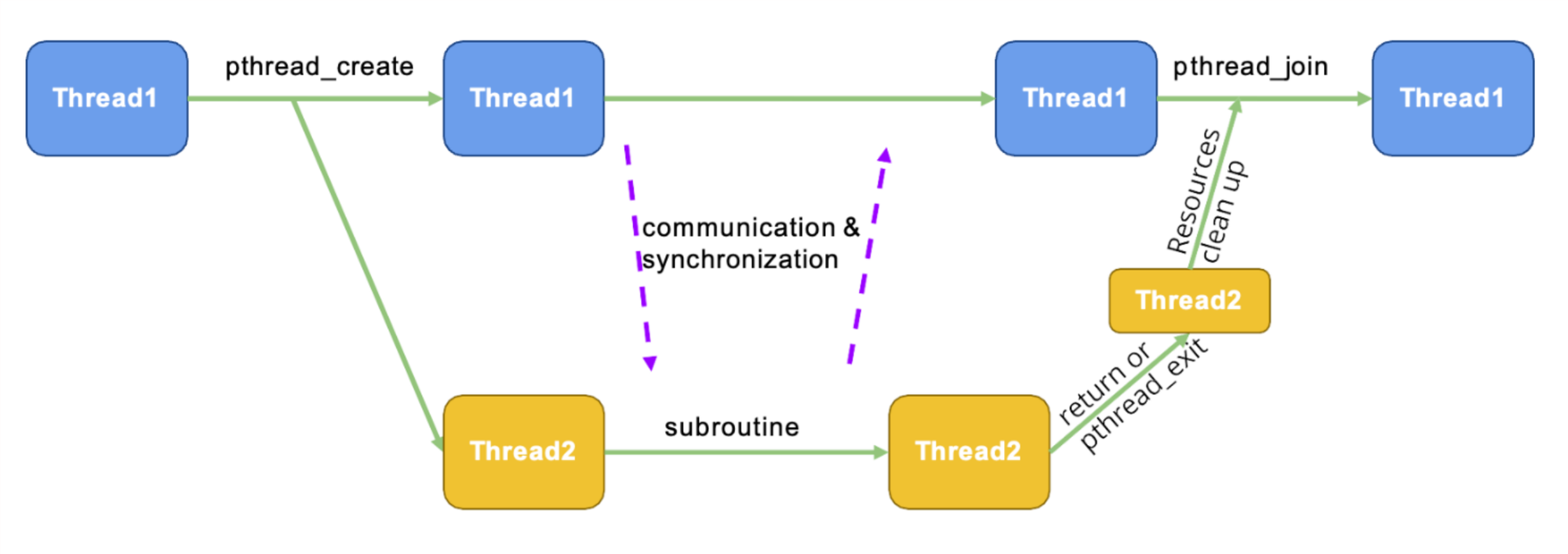
In code: **#include <pthread.h>**

In compile: **-pthread** (gcc main.c -pthread)

High-Level Design & Thread Lifecycle (Recap)

Threads is lightweighted process, with:

- **Shared heap (and global variables)**
- **Independent stack, register, and control (Thread Control Block)**



A Progressive Guide

Now you're given a task, multiply an array of 100 elements (0-99) with 2, a solution is:

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

int main() {
    int *a;
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    for (int i = 0; i < 100; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Computation is independent so multi-threading is a good option for acceleration.
- Multi-threading is like distributing tasks to workers (threads). For example, 2 threads:
  - Thread 1 $\longrightarrow$ 0 ~ 50
  - Thread 2 $\longrightarrow$ 50 ~ 100

Define Thread Functions

Threads are like workers, multi-threading is like distributing tasks to workers. For example, Thread 1 → 0 - 50, Thread 2 → 50 - 100

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int* a;
5
6  void* thr1_func(void* arg) {
7      for (int i = 0; i < 50; i++) { a[i] = a[i] * 2; }
8  }
9
10 void *thr2_func(void* arg) {
11     for (int i = 50; i < 100; i++) { a[i] = a[i] * 2; }
12 }
13
14 int main() {
15     int* a;
16     a = malloc(100 * sizeof(int));
17     for (int i = 0; i < 100; i++) { a[i] = i; }
18
19     for (int i = 0; i < 100; i++) {
20         a[i] = a[i] * 2;
21     }
22 }
```

- All thread function must have:
  - void pointer (**void ***) as input
  - void pointer (**void ***) as output

Questions:

- Both threads have **int i**, will they interrupt?
  - No, threads has **independent stack**.
- Why move line 17 to line 4?
  - If line 17, though **a** points to heap, **a** itself (the pointer) is on the stack of main function
  - If line 4, a become a global variable → all threads can access.

Create Threads

Now it's time to create and run threads

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

int* a;

void* thr1_func(void* arg) {
    for (int i = 0; i < 50; i++) { a[i] = a[i] * 2; }
}

void *thr2_func(void* arg) {
    for (int i = 50; i < 100; i++) { a[i] = a[i] * 2; }
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_t thr1, thr2;
    pthread_attr_t *attr = NULL;
    int iret;

    iret = pthread_create(&thr1, attr, thr1_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread 1"); }
    iret = pthread_create(&thr2, attr, thr2_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread 2"); }

    for (int i = 0; i < 100; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Include header file **#include <pthread.h>**
- Define pthread_t for each thread, respectively
  - **pthread_t** is ID of thread.
- Define **pthread_attr_t** to configure thread behaviour
  - Mostly use NULL to use default

Call **pthread_create**, 4 parameter:

- **pthread_t** * ***thread_id***
- ***pthread_attr_t*** ***attr***
- **void** *(**thr_func**)(void *): Just a function whose input and output type both are void pointer. Functional Pointer (Not Required)
- ***void*** ***arg**: the argument to be passed into **thr_func**

Will return 0 if creation is success and non-zero instead

Created threads start working immediately.

Terminate Threads

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

int* a;

void* thr1_func(void* arg) {
    for (int i = 0; i < 50; i++) { a[i] = a[i] * 2; }
    pthread_exit(NULL);
}

void *thr2_func(void* arg) {
    for (int i = 50; i < 100; i++) { a[i] = a[i] * 2; }
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_t thr1, thr2;
    pthread_attr_t *attr = NULL;
    int iret;

    iret = pthread_create(&thr1, attr, thr1_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread 1"); }
    iret = pthread_create(&thr2, attr, thr2_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread 2"); }

    iret = pthread_join(thr1, NULL);
    if (iret != 0) { perror("Cannot Collect Thread 1"); }
    iret = pthread_join(thr2, NULL);
    if (iret != 0) { perror("Cannot Collect Thread 2"); }
}
```

Similar to process, thread termination involves **two stages**:

1. Thread exit itself by **pthread_exit**
2. Main thread collect value by **pthread_join**

Thread function call **void pthread_exit(void* *retval*)** to exit:

- **void* *retval*** : return value to be collected by **pthread_join**
- No return

Main thread call **int pthread_join(pthread_t *thr*, void* *retval*)** to wait for termination:

- **pthread_join will block main thread (like waitpid)**
- **pthread_t *thr***: specify which thread to wait for
- **void** *retval***: return value from **pthread_exit**,
- use NULL if nothing to collect

Improvement - Accepting Parameters

Defining functions for threads respectively is not extensible. A better way is to identify themselves using parameters

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

int* a;

void* thr_func(void* arg) {
+   int id = * (int*) arg; // (int*) turn void ptr to int ptr
-   for (int i = 0; i < 50; i++) { a[i] = a[i] * 2; }
+   for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

-   pthread_t thr1, thr2;
+   pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret;
+   int *args = malloc(2 * sizeof(int)); args[0] = 0; args[1] = 1;

-   iret = pthread_create(&thr1, attr, thr1_func, NULL);
-   if (iret != 0) { perror("Cannot Create Thread 1"); }
+   for (int i = 0; i < 2; i++) {
+       iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
+       if (iret != 0) { perror("Cannot Create Thread %d", i); }
+   }

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], NULL);
        if (iret != 0) { perror("Cannot Collect Thread 1"); }
    }
}
```

We can use typecast on **void* arg** to receive parameters:

```
int id = * (int*) arg;
```

And define pthread_t(s) and argument(s) by **malloc**, and use the for loop to create threads.

Think question: Merge loop of creation and termination?

```
for (int i = 0; i < 2; i++) {
    iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
    if (iret != 0) { perror("Cannot Create Thread %d", i); }

    iret = pthread_join(thrs[i], NULL);
    if (iret != 0) { perror("Cannot Collect Thread 1"); }
}
```

Think question: More complicated parameters?

Improvement - Collecting Return Value

Let each thread return its id

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

int* a;

void* thr_func(void* arg) {
    int id = * (int*) arg; // (int*) turn void ptr to int ptr
    for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
-   pthread_exit(NULL);
+   pthread_exit((void *)&id); // similarly, return 0 if success
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret, * ret_val;
    int args[] = {0, 1};

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
        if (iret != 0) { perror("Cannot Create Thread"); }
    }

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], (void **) &ret_val);
        if (iret != 0) { perror("Cannot Collect Thread"); }
        printf("Thread %d returns %d\n", i, *ret_val);
    }
}
```

To collect the return value:

- In thread function, use pointer to return value (like id here) as the parameter of **pthread_exit**
- In main thread, initialize receiver, and use double pointer as the 2nd parameter of **pthread_join**

When it runs:

```
>>> Thread 0 returns 1700461583
>>> Thread 1 returns -734169344

>>> Thread 0 returns -1317724401
>>> Thread 1 returns 1991890688

>>> segmentation fault
```

What happened?

Collecting Return Value - Correct

Use Heap Variable - Won't be cleared after exit

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

int* a;

void* thr_func(void* arg) {
    int id = * (int*) arg; // (int*) turn void ptr to int ptr
    for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
-   pthread_exit((void *)&id); // similarly, return 0 if success
+   int *ret = malloc(1 * sizeof(int)); // use heap variable
+   *ret = id;
+   pthread_exit((void *)ret); // similarly, return 0 if success
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret, * ret_val;
    int args[] = {0, 1};

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
        if (iret != 0) { perror("Cannot Create Thread"); }
    }

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], (void **) &ret_val);
        if (iret != 0) { perror("Cannot Collect Thread"); }
        printf("Thread %d returns %d\n", i, *ret_val);
    }
}
```

Use pre-assigned global variable

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

int* a, * ret_vals;

void* thr_func(void* arg) {
    int id = * (int*) arg; // (int*) turn void ptr to int ptr
    for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
-   pthread_exit((void *)&id); // similarly, return 0 if success
+   ret_vals[id] = id;
+   pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret;
+   ret_vals = malloc(2 * sizeof(int));
    int args[] = {0, 1};

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
        if (iret != 0) { perror("Cannot Create Thread"); }
    }

    for (int i = 0; i < 2; i++) {
-       iret = pthread_join(thrs[i], (void **) &ret_val);
+       iret = pthread_join(thrs[i], NULL);
        if (iret != 0) { perror("Cannot Collect Thread"); }
-       printf("Thread %d returns %d\n", i, *ret_val);
+       printf("Thread %d returns %d\n", i, ret_vals[i]);
    }
}
```

Recap question: Why we can use stack variable of main function to pass in parameter?

Identification

For OS to manage threads, 1st is to identify them

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <unistd.h>

void* thr_func(void* arg) {
    int i = * (int*) arg; // (int*) turn void ptr to int ptr
    printf("Thread %d: \nMy pthread id is %d \nMy process id is %d\n\
My system id is %d\n", i, pthread_self(), getpid(), syscall(SYS_gettid));
    pthread_exit(NULL);
}

int main() {

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret, args[] = {0, 1};

    printf("Main Thread Process id %d", getpid());

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
        printf("Main Thread: \nCreate Thread %d with pthread_id %d\n\n", i, thrs[i]);
    }

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], NULL);
        if (iret != 0) { perror("Cannot Collect Thread"); }
    }
}
```

Normally threads has:

- Thread ID: obtain by **pthread_self()**, used by <pthread.h>, the same as pthread_create(pthread* *thread_id*)
- System ID: obtain by **syscall(SYS_gettid)**, used by OS, the same form as PID in Linux (Linux thread ~= process)
- Process ID: obtain by **getpid()**, all threads within process share the same pid
- (Manual) ID: pass in by the arguments for customization.

```
>>> Main Thread Process id 3078547
>>> Main Thread:
>>> Create Thread 0 with pthread_id 527197952
>>>
>>> Main Thread:
>>> Create Thread 1 with pthread_id 518805248
>>>
>>> Thread 0:
>>> My pthread id is 527197952
>>> My process id is 3078547
>>> My system id is 3078548
>>>
>>> Thread 1:
>>> My pthread id is 518805248
>>> My process id is 3078547
>>> My system id is 3078549
```

Cancellation

How to terminate (incorperative) thread from other thread?

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

void *thr1_func (void *arg){
   printf("func1 disable cancellation\n");
   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
   sleep(5);
   printf("func1 enable cancellation\n");
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
   pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
   while (1) { printf("..."); }
   // NO EXIT
}

void *thr2_func (void *arg){
   pthread_t tid = * (pthread_t *)arg;
   for (int i = 0; i < 4; i++) {
      printf("func2 sends a cancellation request to func1\n");
      pthread_cancel(tid);
      sleep(2);
   }
   pthread_exit(NULL);
}

int main() {
   pthread_t thread_id1, thread_id2;

   pthread_create(&thread_id1, NULL, thr1_func, NULL);
   sleep(2);
   pthread_create(&thread_id2, NULL, thr2_func,(void*)&thread_id1);

   pthread_join(thread_id1, NULL);
   printf("func1 has terminated\n");
   pthread_join(thread_id2, NULL);
   printf("func2 has terminated\n");
   return 0;
}
```

pthread_cancel(pthread_t *thread_id*) can cancel threads from other threads like Thread 2 or main. (Like kill API for process)

Threads can configure their behavior on cancellation by:

- **pthread_setcancelstate**(int *state*, int* *old_state*), states:
  - **PTHREAD_CANCEL_DISABLE**: ignore cancellation
  - **PTHREAD_CANCEL_ENABLE**: enable cancellation
  - int* *old_state*: for recording old state, mostly NULL
- **pthread_setcanceltype**(int *type*, int* *old_type*), type:
  - **PTHREAD_CANCEL_DEFERRED**: cancellation will be deferred until the thread next call a function that is a cancellation point. (Default)
  - **PTHREAD_CANCEL_ASYNCHRONOUS**: cancel at any time and mostly immediately. (for manually cancel)

Threads cancelled is terminated, and shall also be collected.

Takeaway - Comparison with process

In high-level, process models an instance of program, while threads are just stream of instructions, but they're highly similar.

	Process	Thread
Creation	fork() → create heap + stack + PCB	pthread_create() → create stack + TCB
Termination	exit() → clear all resources	pthread_exit() → only clear stack, heap alives
Force Termination	kill() / signal handler	pthread_cancel() / pthread_setcancelstate()
Collection	waitid() / waitpid() — collect PCB	pthread_join() — collect value
Communication	IPC — Signal/Pipe/Socket	Shared Memory — heap / global variable

Race Condition & Solution

- Race Condition — Multiple threads accessing same memory
- Atom Operation — Guarantee provided by hardware
- Mutex Lock
    - Create & Destroy
    - Lock & Unlock

Race Condition

Race condition arises if **multiple threads simultaneously read/write the shared resources (memory) → Unpredicted Outcome**

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

int a = 0, iret;

void* thr_func(void* arg) {
    for (int i = 0; i < 1000000; i++) { a += 1; }
    pthread_exit(NULL);
}

int main() {
    pthread_t thr1, thr2;

    iret = pthread_create(&thr1, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }
    iret = pthread_create(&thr2, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }

    iret = pthread_join(thr1, NULL);
    if (iret != 0) { perror("Cannot Collect Thread"); }
    iret = pthread_join(thr2, NULL);
    if (iret != 0) { perror("Cannot Collect Thread"); }

    printf("%d", a);
}
```

Correct: 2000000, Actual: 1016738, 1003869, 1019114, …

Question: Is the previous example race condition?

Reason: most C operations like `a += 1` involves:

```
thr_func:
.LFB6:
    .cfi_startproc
    ...
    jmp .L2
.L3:
    movl    a(%rip), %eax
    addl    $1, %eax
    movl    %eax, a(%rip)
    addl    $1, -4(%rbp)
```

Read Memory to Register	**movl a(%rip), %eax**
Do Operation on Register	**addl $1, %eax**
Write Register to Memory	**movl %eax, a(%rip)**

Thus, if Thread 1 and Thread 2 **read memory simultaneously**:

- They add 1 to their own register
- Then write back a + 1 to a
- a in memory will be a + 1 instead of a + 2

Atomic Operation

As discussed in lecture, the correct way is to allow at most one thread access each shared data

For this naive case, we can use the **fetch_and_add** atomic:

```
int FetchAndAdd(int* ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

🤔 Looks no diff w.r.t. previous code, still read/write memory.

🧐 YES, **above is just illustration, following is actual code**:

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

int a = 0, iret;

void* thr_func(void* arg) {
    for (int i = 0; i < 1000000; i++) {
-       a += 1;
+       __atomic_fetch_add(&a, 1, __ATOMIC_SEQ_CST);
    }
    pthread_exit(NULL);
}

int main() { ... }
```

Now the code will always leads to 2000000, correct answer.

What's the difference? Compare the assembly code:

With atomic_fetch_add

```
thr_func:
.LFB6:
    .cfi_startproc
    ...
    jmp .L2
.L3:
    lock addl   $1, a(%rip)
    addl    $1, -4(%rbp)
```

Without atomic_fetch_add

```
thr_func:
.LFB6:
    .cfi_startproc
    ...
    jmp .L2
.L3:
    movl    a(%rip), %eax
    addl    $1, %eax
    movl    %eax, a(%rip)
    addl    $1, -4(%rbp)
```

Correctness of Atomic is **guaranteed by hardware (CPU)** and requires the hardware support, not related with OS.

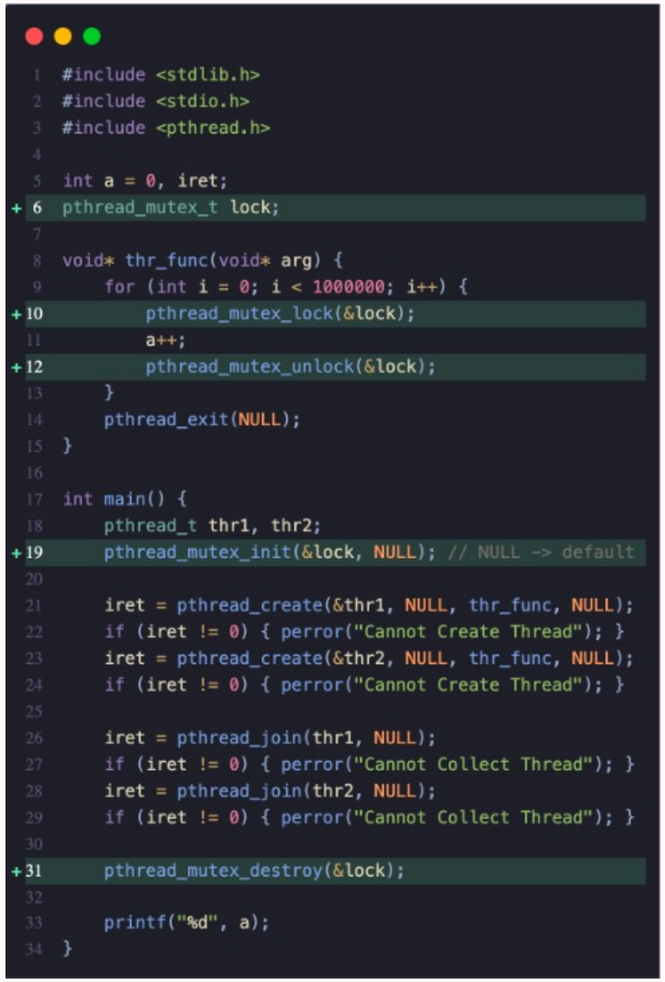
Other atomic operations supported in GCC(Linux):

- **__atomic_add_fetch**
- **__atomic_test_and_set**
- **__atomic_compare_exchange**

Reference: GCC Atomic Built-in

Mutex := Mutually Exclusive

Atomic only cover simple cases, for more complicated scenario we need to **lock the critical section**



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

int a = 0, iret;
pthread_mutex_t lock;

void* thr_func(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        a++;
        pthread_mutex_unlock(&lock);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t thr1, thr2;
    pthread_mutex_init(&lock, NULL); // NULL -> default

    iret = pthread_create(&thr1, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }
    iret = pthread_create(&thr2, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }

    iret = pthread_join(thr1, NULL);
    if (iret != 0) { perror("Cannot Collect Thread"); }
    iret = pthread_join(thr2, NULL);
    if (iret != 0) { perror("Cannot Collect Thread"); }

    pthread_mutex_destroy(&lock);

    printf("%d", a);
}
```

- Before init, define the lock variable **pthread_mutex_t**.
- Initialize lock by **pthread_mutex_init** with two parameter:
  - **pthread_mutex_t *mutex***: lock variable to be initialized
  - **pthread_mutexattr_t *attr***: configuration, usually NULL
- Acquire the lock by **pthread_mutex_lock**
  - **IMPORTANT: pthread_mutex_lock is a blocked call, thread will be blocked until lock is acquired**
  - **pthread_mutex_t *mutex***: lock to acquire
- Release the lock by **pthread_mutex_unlock**
  - **pthread_mutex_t *mutex***: lock to acquire
- Finally, destroy the lock by **pthread_mutex_destroy**
  - **pthread_mutex_t *mutex***: lock to acquire

Now the code will always leads to 2000000, correct answer.

Mutex lock gives more flexibility because you can do more between pthread_mutex_lock and pthread_mutex_unlock

trylock

pthread_mutex_lock block threads until lock acquired, but threads can works on other tasks with no race condition.

Want thread to work on another task b if a is occupied.

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

int a = 0, b = 0, iret;
pthread_mutex_t lock_a, lock_b;

void* thr_func(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        if (pthread_mutex_trylock(&lock_a) == 0) {
            a++;
            pthread_mutex_unlock(&lock_a);
        } else if (pthread_mutex_trylock(&lock_b) == 0) {
            b++;
            pthread_mutex_unlock(&lock_b);
        }
    }
    pthread_exit(NULL);
}

int main() { ... }
>>> a: 1029395, b: 970605
```

Sum of a and b is 2000000

pthread_mutex_lock will is not appropriate as it'll block thread until lock acquired.

Instead, **pthread_mutex_trylock**(pthread_mutex_t* *mutex*):

- **Immediately** return 0 if lock acquired and non-zero if not
- pthread_mutex_t *mutex*: lock try to acquire

pthread_mutex_trylock will not block thread, if lock is not acquired, thread can do other things → more efficient usage.

Synchronization & Condition Variable

- Synchronization — Controlling Execution Order
- Conditional Variable

Synchronization — Controlling Execution Order

Threads are executed independently (and in parallel), but we do need the control on execution order.

For example, in example of 0-99 × 2, need to know if Thread 1 and Thread 2 have finished their task → **pthread_join**

Question: a pthread_join but thread keep alive? i.e., just wait.

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

int* a, *state;
pthread_mutex_t lock; // not race, just for cond
pthread_cond_t cond;

void* thr_func(void* arg) {
    int id = * (int*) arg; // (int*) turn void ptr to int ptr
    for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
    pthread_mutex_lock(&lock);
    state[id] = 1;
    pthread_mutex_unlock(&lock);
    pthread_cond_signal(&cond);
    // can do more here
    sleep(3);
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_mutex_init(&lock, NULL); // NULL -> default
    pthread_cond_init(&cond, NULL); // NULL -> default

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret, args[] = {0, 1};
    state = malloc(2 * sizeof(int)); state[0] = 0; state[1] = 0;

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
    }

    pthread_mutex_lock(&lock);
    while (state[0] && state[1]) {
        pthread_cond_wait(&cond, &lock);
    }

    printf("All threads have done their job and still alive"); fflush(stdout);
    sleep(3);

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
}
```

Recap - Cond Variable

Cond variable involves an inner "queue", with two function:

- cond_wait: wait for cond, **put myself in "queue" and blocked**
- cond_signal: **wake up one thread in the "queue"** to work

Usage

IMPORTANT: Conditional variable must be used with mutex_lock (especially cond_wait) to protect thread safety

Similar to mutex_lock, we need to initialize **pthread_cond_t**:

1. Define the cond variable **pthread_cond_t cond;**
2. Do initialization by **pthread_cond_init(&cond, NULL)**
3. Finally destroy by **pthread_cond_destroy(&cond**)

Using cond variable involves two end:

- **pthread_cond_signal(&cond)**: wake up one thread in queue, no mutex protect need
- **pthread_cond_wait(&cond, &lock)**: (blocked) wait for a specific condition:
  - **Can only be called with mutex lock (&lock) acquired**
  - **Will release the mutex lock (&lock) once called**
  - **Will get back mutex lock once being waked up** (implicitly do pthread_mutex_lock)

Cond Variable - What's state here?

Other than pthread_cond_t, there's an **int * state**, but why?

```c
int* a, *state;
pthread_mutex_t lock; // not race, just for cond
pthread_cond_t cond;

void* thr_func(void* arg) {
    int id = * (int*) arg; // (int*) turn void ptr to int ptr
    for (int i = 50 * id; i < 50 * (id + 1); i++) { a[i] = a[i] * 2; }
    pthread_mutex_lock(&lock);
    state[id] = 1;
    pthread_mutex_unlock(&lock);
    pthread_cond_signal(&cond);
    // can do more here
    sleep(3);
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_mutex_init(&lock, NULL); // NULL -> default
    pthread_cond_init(&cond, NULL); // NULL -> default

    pthread_t * thrs = malloc(2 * sizeof(pthread_t));
    pthread_attr_t *attr = NULL;
    int iret, args[] = {0, 1};
    state = malloc(2 * sizeof(int)); state[0] = 0; state[1] = 0;

    for (int i = 0; i < 2; i++) {
        iret = pthread_create(&thrs[i], attr, thr_func, &args[i]);
    }

    pthread_mutex_lock(&lock);
    while (state[0] && state[1]) {
        pthread_cond_wait(&cond, &lock);
    }

    printf("All threads have done their job and still alive"); fflush(stdout);
    sleep(3);

    for (int i = 0; i < 2; i++) {
        iret = pthread_join(thrs[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
}
```

- Initialize the **state** to be `[0, 0]`
- Each thread **update their state before signal** main thread
    - States are shared among threads → need mutex_lock
- Waited thread (main) **will check state once they wake up**
    - Recall cond_wait acquire mutex_lock when wake up
    - If state matches, go to next stage

Reason: **Cond Variable is "Stateless"**!

- Only identify an even happened (like a thread finish task)
- In simple cases like sync with only one thread,
- solely use cond variable is fine → one signal, one wait
- But more complicated cases, need help of shared variables like **state** → thus need mutex_lock

Question: Replace while by if?

```c
pthread_mutex_lock(&lock);
while (state[0] && state[1]) {
    pthread_cond_wait(&cond, &lock);
}
```