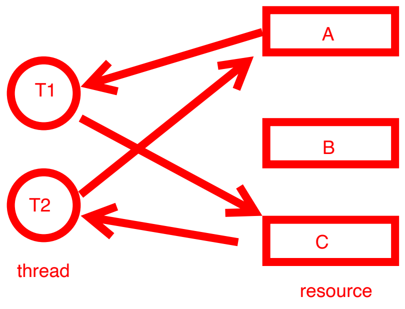
What techniques can we use about thread synchronisation?
- Mutex
- Semaphore
- Barrier
- Condition variables



THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Week 11 Tutorial

Synchronisation and Atomics

Pre-tutorial Questions

Question 1: Diagnosing deadlocks

There are four threads in the following program, labelled T_1 to T_4, which are (amongst other operations) locking and unlocking three mutexes labelled A, B, and C. The order of locking is shown below:

$$T_1 : lock(A), lock(B), lock(C), unlock(A), unlock(B), unlock(C)$$
$$T_2 : lock(A), lock(B), lock(C), unlock(C), unlock(B), unlock(A)$$
$$T_3 : lock(B), lock(C), unlock(B), unlock(C)$$
$$T_4 : lock(C), lock(A), unlock(A), unlock(C)$$

T1 lock(A)
T4 lock(C)

T1 lock(B)
T4 wait(A)

T1 wait(C)
T4 wait(A)

=> Deadlock

T1 lock(A)
T2 wait(A)
T3 lock(B)
T4 lock(C)

T1 wait(B)
T2 wait(A)
T3 wait(C)
T4 wait(A)

=> Deadlock

1. Which threads have the potential to deadlock here? T1: lockA lockB trylockC T4: lockC trylockA
   Show two possible interleavings of instructions that cause at least two threads to go into a deadlock.

2. Draw your interleavings from the above as a resource dependency graph.
   Verify that the deadlock exists by highlighting the cycle.

Question 2: Limits of parallelism

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1-p) + \frac{p}{n}}$$

Amdahl's law gives limits on the maximum possible speedup of programs.
What is the maximum possible speedup of the following programs:

1. 10% sequential, 90% parallel, with 4 processors used. 1 / (0.1 + 0.9/4) = 3.07

2. 90% sequential, 10% parallel, with 32 processors used. 1 / (0.9 + 0.1/32) = 1.107

3. 5% sequential, 95% parallel, with an unlimited number of processors. 1 / (0.05 + 0.95/inf) = 20

Discuss some reasons why the maximum theoretical speedup is not often achieved in computations.

Question 3: Dining philosophers

Assume that there are N philosophers sitting at a round table. A single chopstick is placed between two adjacent philosophers. Every philosopher is either thinking or eating. However, a philosopher needs both chopsticks (to the left and to the right) to start eating. They are not allowed to acquire chopsticks that are not immediately adjacent to them. Complete the following program so that each philosopher is able to eat.

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

#define THINKERS 5

static pthread_mutex_t chopsticks[THINKERS];

void* dine(void* arg) {

    const unsigned id = *((unsigned *) arg);

    while (true) {
        // TODO: Acquire two chopsticks first
        // the ith philosopher can only reach
        // the ith and (i + 1)th chopstick
        printf("Philosopher %u is eating\n", id);
    }

    return NULL;
}

int main(void) {

    unsigned args[THINKERS];
    pthread_t thinkers[THINKERS];

    // create the chopsticks
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_mutex_init(chopsticks + i, NULL) != 0) {
            perror("unable to initialize mutex");
            return 1;
        }
    }

    // launch threads
    for (size_t i = 0; i < THINKERS; i++) {
        args[i] = i;
        if (pthread_create(thinkers + i, NULL, dine, args + i) != 0) {
            perror("unable to create thread");
            return 1;
        }
    }

    // wait for threads to finish
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_join(thinkers[i], NULL) != 0) {
            perror("unable to join thread");
            return 1;
        }
    }

    // remove the chopsticks
    for (size_t i = 0; i < THINKERS; i++) {
        if (pthread_mutex_destroy(chopsticks + i) != 0) {
            perror("unable to destroy mutex");
            return 1;
        }
    }

    return 0;
}
```

Tutorial Questions

Question 4: Semaphore philosophers

We have previously solved the dining philosophers problem by using a locking hierarchy.
This time, use a semaphore for the table that only allows $N/2$ philosophers to eat at a time.

Question 5: Race conditions and condition variables

The following program creates two threads. The threads wait for each other in a back and forth manner that models an email conversation between `Alice` and `Bob`. The while loop around the `pthread_cond_wait` is used because `pthread_cond_wait` may return from spurious wake ups.

The code however is broken. It deadlocks.

1. Run `valgrind --tool=helgrind ./cond` to detect any synchronisation problems.

2. Why doesn't wrapping a mutex to protect `alice_has_mail` and `bob_has_mail` work? Hint: what happens to a `pthread_cond_signal` when no thread is waiting on it?

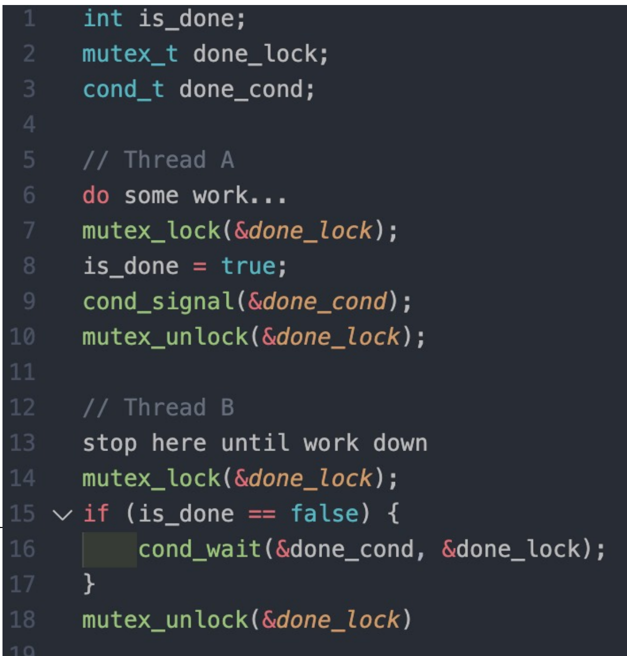
3. Fix the program so it works. You may remove the condition variables and use semaphores instead.

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

#define EXCHANGES 10000

bool alice_has_mail = false;
bool bob_has_mail = false;

pthread_mutex_t alice_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t alice_signal = PTHREAD_COND_INITIALIZER;

pthread_mutex_t bob_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t bob_signal = PTHREAD_COND_INITIALIZER;

void* alice(void* arg) {

    for (unsigned i = 0; i < EXCHANGES; ++i) {
        pthread_mutex_lock(&alice_mutex);
        while (!alice_has_mail) {
            puts("Alice: Waiting for Bob to signal");
            pthread_cond_wait(&alice_signal, &alice_mutex);
            puts("Alice: Got Bob's signal");
        }
        pthread_mutex_unlock(&alice_mutex);

        alice_has_mail = false;
        bob_has_mail = true;

        pthread_cond_signal(&bob_signal);
        printf("Alice: Got mail (%d) from Bob\n", i);
    }

    return NULL;
}
```

cond_signal:
notify one thread that is waiting on a condition

cond_wait:
1. make thread B wait
2. unlocks a mutex, thus will allow thread A to acquire the lock and eventually send the signal.
Therefore, the call blocks until the signal is received and thread B acquires the lock again.

```c
void* bob(void* arg) {

    for (unsigned i = 0; i < EXCHANGES; ++i) {
```



```c
int is_done;
mutex_t done_lock;
cond_t done_cond;

// Thread A
do some work...
mutex_lock(&done_lock);
is_done = true;
cond_signal(&done_cond);
mutex_unlock(&done_lock);

// Thread B
stop here until work down
mutex_lock(&done_lock);
if (is_done == false) {
    cond_wait(&done_cond, &done_lock);
}
mutex_unlock(&done_lock)
```

```
        pthread_mutex_lock(&bob_mutex);
        while (!bob_has_mail) {
            puts("Bob: Waiting for Alice to signal");
            pthread_cond_wait(&bob_signal, &bob_mutex);
            puts("Bob: Got Alice's signal");
        }
        pthread_mutex_unlock(&bob_mutex);

        alice_has_mail = true;
        bob_has_mail = false;

        pthread_cond_signal(&alice_signal);
        printf("Bob: Got mail (%d) from Alice\n", i);
    }

    return NULL;
}

int main(void) {

    pthread_t alice_tid;
    pthread_t bob_tid;

    pthread_create(&alice_tid, NULL, alice, NULL);
    pthread_create(&bob_tid, NULL, bob, NULL);

    // start the chain
    bob_has_mail = true;
    pthread_cond_signal(&bob_signal);

    pthread_join(alice_tid, NULL);
    pthread_join(bob_tid, NULL);

    return 0;
}
```

Question 6: Dancing threads

In following program uses threads to simulate dancers in a ballroom. Each thread is represents a person in the ballroom wearing a coloured dress. In the ballroom there is a main stage. However, dancers are only allowed onto the main stage if they are in a group of three. The second requirement to entering the main stage is that two of the dancers must be wearing a red dress and the other dancer must be wearing a white dress.

In the following program, each thread calls the function that corresponds to its colour after it spawns. i.e. a thread with a red dress calls the `red` function, a thread with a white dress calls the `white` function.

Your task is to add synchronisation code to `red` and `white` such that they block until all three can enter the main stage together, then the function should return. Suppose two red threads are blocked on `red`, and then a white thread calls `white`, the third thread should wake up the other two threads and they should all return.

This creates a barrier object at the passed address (pointer to the barrier object is in barrier), with the attributes as specified by attr (we'll just use NULL to get the defaults). The number of threads that must call pthread_barrier_wait() is passed in count.

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

#define NWHITE 1000
#define NRED   (NWHITE * 2)

void red(void);
void white(void);

void* red_thread(void* args) {
    // TODO
    red();
    return NULL;
}

void* white_thread(void* args) {
    // TODO
    white();
    return NULL;
}

int main(void) {
    pthread_t ids[NWHITE + NRED];

    for (size_t i = 0; i < NWHITE; i++) {
        for (size_t j = 0; j < 2; j++) {
            pthread_create(ids + i * 3 + j, NULL, red_thread, NULL);
        }

        pthread_create(ids + i * 3 + 2, NULL, white_thread, NULL);
    }
    for (size_t i = 0; i < NWHITE + NRED; i++) {
        pthread_join(ids[i], NULL);
    }
}
```

Question 7: Atomics (Extension)

In this exercise we will examine some ways to write concurrent programs using atomic instructions directly built into the processor. Like the SIMD instructions, they are exposed as compiler intrinsics. In `gcc`, they are available as part of the atomic builtins. These builtins are now declared legacy with the advent of C11 atomics. However you need the latest versions of `gcc` and `clang` to use them with the new names defined in `stdatomic.h`. Since `clang` maintains compatibility with `gcc`, these builtins are also available for `clang`.

These basic building blocks allow us to create faster data structures and synchronisation mechanisms.

1. The following example compares the speed of atomic increments with a mutex to protecting the same global variable. Remove the comments around `#define USEATOMIC` and compare differences in performance. Try varying the thread count to something much larger the number of available cores. How does the mutex version scale in comparison to the atomics version.

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

#define NTHREADS 2
#define REPEATS 10000000

// #define USEATOMIC

#ifndef USEATOMIC
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
#endif

unsigned counter = 0;

void* worker(void* arg) {

    for (unsigned i = 0; i < REPEATS; i++) {
        // NOTE: this code is here to benchmark the synchronisation
        // mechanism you should not attempt to atomically increment
        // a highly contended global counter if you can avoid it.
        // A much faster way would be save the results of the sum
        // on the stack and then add it atomically to the final value.

#ifdef USEATOMIC
    __sync_fetch_and_add(&counter, 1);
#else
```

```
        pthread_mutex_lock(&mutex);
        counter += 1;
        pthread_mutex_unlock(&mutex);
#endif
    }

    return NULL;
}







int main(void) {

    pthread_t thread_ids[NTHREADS];
    for (size_t i = 0; i < NTHREADS; i++) {
        pthread_create(thread_ids + i, NULL, worker, NULL);
    }

    for (size_t i = 0; i < NTHREADS; i++) {
        pthread_join(thread_ids[i], NULL);
    }

    printf("%u\n", counter);

    return 0;
}
```

2. The following code uses a barrier provided by `pthread`. Remove it and implement your own using the atomic builtins. You may need to use a spinlock in your implementation. Compare the performance.

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

#define NTHREADS 4
#define REPEATS 1000000

pthread_barrier_t barrier;

void* worker(void* arg) {

    for (unsigned i = 0; i < REPEATS; ++i) {
        pthread_barrier_wait(&barrier);
    }

    return NULL;
}
```

```
int main(void) {

    pthread_t thread_ids[NTHREADS];

    pthread_barrier_init(&barrier, NULL, NTHREADS);

    for (size_t i = 0; i < NTHREADS; ++i) {
        pthread_create(thread_ids + i, NULL, worker, NULL);
    }

    for (size_t i = 0; i < NTHREADS; ++i) {
        pthread_join(thread_ids[i], NULL);
    }

    pthread_barrier_destroy(&barrier);

    return 0;
}
```