Computer Systems Fundamentals

two_threads.c

```
 simple example which launches two threads of execution


$ gcc -pthread two_threads.c -o two_threads
$ ./two_threads|more

Hello this is thread #1 i=0

Hello this is thread #1 i=1

Hello this is thread #1 i=2

Hello this is thread #1 i=3

Hello this is thread #1 i=4

Hello this is thread #2 i=0

Hello this is thread #2 i=1
...
```

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

// this function is called to start thread execution
// it can be given any pointer as argument (int *) in this example

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 10; i++) {
        printf("Hello this is thread #%d: i=%d\n",  *p, i);
    }

    // a thread finishes when the function returns or  thread_exit is called
    // a pointer of any type can be returned
    // this can be obtained via thread_join's 2nd argument
    return NULL;
}

int main(void) {
    //create two threads performing almost the same task

    pthread_t thread_id1;
    int thread_number1 = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number1);

    int thread_number2 = 2;
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, run_thread, &thread_number2);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

two_threads_broken.c

```
_simple example which launches two threads of execution
_but demonstrates the perils of accessing non-local variables
_from a thread


$ gcc -pthread two_threads_broken.c -o two_threads_broken
$ ./two_threads_broken|more

Hello this is thread 2: i=0

Hello this is thread 2: i=1

Hello this is thread 2: i=2

Hello this is thread 2: i=3

Hello this is thread 2: i=4

Hello this is thread 2: i=5

Hello this is thread 2: i=6

Hello this is thread 2: i=7

Hello this is thread 2: i=8

Hello this is thread 2: i=9

Hello this is thread 2: i=0

Hello this is thread 2: i=1

Hello this is thread 2: i=2

Hello this is thread 2: i=3

Hello this is thread 2: i=4

Hello this is thread 2: i=5

Hello this is thread 2: i=6

Hello this is thread 2: i=7

Hello this is thread 2: i=8

Hello this is thread 2: i=9
$...
```

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

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 10; i++) {

        // variable thread number will probably have changed in main
        // before execution reaches here
        printf("Hello this is thread %d: i=%d\n", *p, i);

    }

    return NULL;
}

int main(void) {
    pthread_t thread_id1;
    int thread_number = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number);

    thread_number = 2;
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, run_thread, &thread_number);

    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

n_threads.c

```
 simple example of running an arbitrary number of threads
 for example:


$ gcc -pthread n_threads.c -o n_threads
$ ./n_threads 10

Hello this is thread 0: i=0

Hello this is thread 0: i=1

Hello this is thread 0: i=2

Hello this is thread 0: i=3


Hello this is thread 0: i=4

Hello this is thread 0: i=5

Hello this is thread 0: i=6

Hello this is thread 0: i=7
...
```

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

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 42; i++) {
        printf("Hello this is thread %d: i=%d\n", *p, i);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <n-threads>\n", argv[0]);
        return 1;
    }
    int n_threads = strtol(argv[1], NULL, 0);
    assert(n_threads > 0 && n_threads < 100);

    pthread_t thread_id[n_threads];
    int argument[n_threads];

    for (int i = 0; i < n_threads; i++) {
        argument[i] = i;
        pthread_create(&thread_id[i], NULL, run_thread, &argument[i]);
    }

    // wait for the threads to finish
    for (int i = 0; i < n_threads;i++) {
        pthread_join(thread_id[i], NULL);
    }

    return 0;
}
```

thread_sum.c

```
 simple example of dividing a task between n-threads

 compile like this:

$ gcc -O3 -pthread thread_sum.c -o thread_sum

 one thread takes 10 seconds

$ time ./thread_sum 1 10000000000
Creating 1 threads to sum the first 10000000000 integers
Each thread will sum 10000000000 integers
Thread summing integers          0 to 10000000000 finished sum is 49999999990067863552

Combined sum of integers 0 to 10000000000 is 49999999990067863552

real    0m11.924s
user    0m11.919s
sys 0m0.004s
$



Four threads runs 4x as fast on a machine with 4 cores

$
Creating 4 threads to sum the first 10000000000 integers
Each thread will sum 2500000000 integers
Thread summing integers 2500000000 to  5000000000 finished sum is  9374999997502005248
Thread summing integers 7500000000 to 10000000000 finished sum is 21874999997502087168
Thread summing integers 5000000000 to  7500000000 finished sum is 15624999997500696576
Thread summing integers          0 to  2500000000 finished sum is  3124999997567081472

Combined sum of integers 0 to 10000000000 is 49999999990071869440

real    0m3.154s
user    0m12.563s
sys 0m0.004s
$



Note result is inexact because we use values can't be exactly represented as double
and exact value printed depends on how many threads we use - becayse we break up
the computation differently depending on number of threads
```

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

struct job {
    long start;
    long finish;
    double   sum;
};

void *run_thread(void *argument) {
    struct job *j = argument;
    long start = j->start;
    long finish = j->finish;
    double sum = 0;

    for (long i = start; i < finish; i++) {
        sum += i;
    }

    j->sum = sum;

    printf("Thread summing integers %10lu to %11lu finished sum is %20.0f\n", start, finish, sum);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n-threads> <n-integers-to-sum>\n", argv[0]);
        return 1;
    }

    int n_threads = strtol(argv[1], NULL, 0);
    assert(n_threads > 0 && n_threads < 100);
    long integers_to_sum = strtol(argv[2], NULL, 0);
    assert(integers_to_sum > 0);

    long integers_per_thread = (integers_to_sum - 1)/n_threads + 1;

    printf("Creating %d threads to sum the first %lu integers\n", n_threads, integers_to_sum);
    printf("Each thread will sum %lu integers\n", integers_per_thread);

    pthread_t thread_id[n_threads];
    struct job jobs[n_threads];

    for (int i = 0; i < n_threads; i++) {
        jobs[i].start = i * integers_per_thread;
        jobs[i].finish = jobs[i].start + integers_per_thread;
        if (jobs[i].finish > integers_to_sum) {
            jobs[i].finish = integers_to_sum;
        }

        // create a thread which will sum integers_per_thread integers
        pthread_create(&thread_id[i], NULL, run_thread, &jobs[i]);
    }

    // wait for each threads to finish
    // then add its individual sum to the overall sum
    double overall_sum = 0;
    for (int i = 0; i < n_threads;i++) {
        pthread_join(thread_id[i], NULL);
        overall_sum += jobs[i].sum;
    }

    //
    printf("\nCombined sum of integers 0 to %lu is %.0f\n", integers_to_sum, overall_sum);
    return 0;
}
```

bank_account_broken.c

```
 simple example demonstrating unsafe access to a global variable from threads


$ gcc -O3 -pthread bank_account_broken.c -o bank_account_broken
$ ./bank_account_broken

Andrew's bank account has $108829
$
```

```c
#define _POSIX_C_SOURCE 199309L

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

int bank_account = 0;

// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {

        // execution may switch threads in middle of assignment
        // between load of variable value
        // and store of new variable value
        // changes other thread makes to variable will be lost
        nanosleep(&(struct timespec){.tv_nsec = 1}, NULL);
        bank_account = bank_account + 1;
    }

    return NULL;
}

int main(void) {
    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will probably be much less than $200000
    printf("Andrew's bank account has $%d\n", bank_account);
    return 0;
}
```

bank_account_mutex.c

```
 simple example demonstrating safe access to a global variable from threads
 using a mutex (mutual exclusion) lock


$ gcc -O3 -pthread bank_account_mutex.c -o bank_account_mutex
$ ./bank_account_mutex

Andrew's bank account has $200000
$
```

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

int andrews_bank_account = 0;

pthread_mutex_t bank_account_lock = PTHREAD_MUTEX_INITIALIZER;


// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {


        pthread_mutex_lock(&bank_account_lock);

        // only one thread can execute this section of code at any time

        andrews_bank_account = andrews_bank_account + 1;

        pthread_mutex_unlock(&bank_account_lock);
    }

    return NULL;
}

int main(void) {
    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will always be $200000
    printf("Andrew's bank account has $%d\n", andrews_bank_account);
    return 0;
}
```

bank_account_semphore.c

```
 simple example demonstrating ensuring safe access to a global variable from threads
 using a semaphore


$ gcc -O3 -pthread bank_account_semphore.c -o bank_account_semphore
$ ./bank_account_semphore

Andrew's bank account has $200000
$
```

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

int andrews_bank_account = 0;

sem_t bank_account_semaphore;

// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {

        // decrement bank_account_semaphore if > 0
        // otherwise wait until > 0
        sem_wait(&bank_account_semaphore);

        // only one thread can execute this section of code at any time
        // because  bank_account_semaphore was initialized to 1

        andrews_bank_account = andrews_bank_account + 1;

        // increment bank_account_semaphore
        sem_post(&bank_account_semaphore);
    }

    return NULL;
}

int main(void) {
    // initialize bank_account_semaphore to 1
    sem_init(&bank_account_semaphore, 0, 1);

    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will always be $200000
    printf("Andrew's bank account has $%d\n", andrews_bank_account);

    sem_destroy(&bank_account_semaphore);
    return 0;
}
```

bank_account_deadlock.c

simple example which launches two threads of execution which increment a global variable

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

int andrews_bank_account1 = 100;
pthread_mutex_t bank_account1_lock = PTHREAD_MUTEX_INITIALIZER;

int andrews_bank_account2 = 200;
pthread_mutex_t bank_account2_lock = PTHREAD_MUTEX_INITIALIZER;

// swap values between Andrew's two bank account 100,000 times
void *swap1(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&bank_account1_lock);
        pthread_mutex_lock(&bank_account2_lock);

        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;

        pthread_mutex_unlock(&bank_account2_lock);
        pthread_mutex_unlock(&bank_account1_lock);

    }

    return NULL;
}

// swap values between Andrew's two bank account 100,000 times
void *swap2(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&bank_account2_lock);
        pthread_mutex_lock(&bank_account1_lock);

        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;

        pthread_mutex_unlock(&bank_account1_lock);
        pthread_mutex_unlock(&bank_account2_lock);

    }

    return NULL;
}
int main(void) {
    //create two threads performing almost the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, swap1, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, swap2, NULL);

    // threads will probably never finish
    // deadlock will likely likely occur
    // with one thread holding  bank_account1_lock
    // and waiting for bank_account2_lock
    // and the other  thread holding  bank_account2_lock
    // and waiting for bank_account1_lock

    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    return 0;
}
```

COMP1521 20T2: Computer Systems Fundamentals is brought to you by the School of Computer Science and Engineering at the University of New South Wales, Sydney

at the University of New South Wales, Sydney.

For all enquiries, please email the class account at cs1521@cse.unsw.edu.au

CRICOS Provider 00098G