THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Week 10 Tutorial

Parallelism with POSIX threads and optimisation in C

Pthreads basics

`Pthreads` is a library specified by the POSIX standard, defining an API for creating and manipulating threads.

To use the `Pthreads` library:

1. Include the `pthread.h` header in your source code.
2. Specify `-pthread` in your compiler flags to tell the compiler to link and configure the Pthreads library.

   ```
   > clang -g -Wall -Werror -std=gnu11 program.c -o program -pthread
   ```

   Linker flags should be specified at the end of your usual compiler options.

Creating threads

To create a thread using the `Pthreads` library, we use the `pthread_create` function.

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

1. The first argument to this function takes a `pthread_t`, it will use it to store the thread ID.
2. The second argument is a pointer to a `pthread_attr_t` structure, you can use this to specify additional options in the creation of the thread, it's fine to leave this as `NULL`.
3. The third argument is a pointer to a function that the thread will execute once it spawns.
4. The last argument is the parameter passed to the thread function.
5. The function returns a non zero value if an error occurred.

Waiting on threads

The `pthread_join` function forces the calling thread to wait for a particular thread ID to finish.

```
int pthread_join(pthread_t thread, void** retval);
```

1. The first argument is the thread to wait for.
2. The second argument allows you to get the return value from the `start_routine`. If you don't need the return value you can just pass in `NULL`.
3. The function returns a non-zero value if an error occurred.

Question 1: Hello from `Pthreads`

The following code creates 4 threads, giving each of them an argument and waits for them to finish.

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

#define NTHREADS 4

void* worker(void* arg) {
    const int argument = *((int*) arg);
    printf("Hello from thread %d\n", argument);
    return NULL;
}

int main(void) {
    int args[NTHREADS] = { 1, 2, 3, 4 };
    pthread_t thread_ids[NTHREADS];

    // Create threads with given worker function and argument
    for (size_t i = 0; i < NTHREADS; i++) {
        if (pthread_create(thread_ids + i, NULL, worker, args + i) != 0) {
            perror("unable to create thread");
            return 1;
        }
    }

    // Wait for all threads to finish
    for (size_t i = 0; i < NTHREADS; i++) {
        if (pthread_join(thread_ids[i], NULL) != 0) {
            perror("unable to join thread");
            return 1;
        }
    }
}
```

Compile and run the code above. Make sure you understand exactly what each line is doing. Reminder that `printf` is thread safe by the POSIX standard, thus calling it from multiple threads will yield no side effects.

1. Why does the program output Hello from thread ... in different orderings?
2. What is the difference between a thread (`pthread_create`) and a process (`fork`)?

Question 2: Parallel Sum

In the following code, each thread is given a section of an array of numbers to sum.

1. How long does it take to run when you set `THREADS` to 1, 2, 3 ... etc?
   You can use `time ./sum` to see how long the program takes to run.
2. Why does using more threads make the program slower?
3. How can this be fixed? Hint: False sharing

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

#define LENGTH 100000000
#define NTHREADS 4
#define NREPEATS 10
#define CHUNK (LENGTH / NTHREADS)

typedef struct {
        size_t id;
        long* array;
        long result;
} worker_args;

void* worker(void* args) {

        worker_args* wargs = (worker_args*) args;

        const size_t start = wargs->id * CHUNK;
        const size_t end = wargs->id == NTHREADS - 1 ? LENGTH : (wargs->id + 1) * (

        // Sum values from start to end
        for (size_t i = start; i < end; i++) {
                wargs->result += wargs->array[i];
        }

        return NULL;
}
```

```c
int main(void) {

        long* numbers = malloc(sizeof(long) * LENGTH);
        for (size_t i = 0; i < LENGTH; i++) {
                numbers[i] = i + 1;
        }

        worker_args* args = malloc(sizeof(worker_args) * NTHREADS);
        for (size_t n = 1; n <= NREPEATS; n++) {
                for (size_t i = 0; i < NTHREADS; i++) {
                        args[i] = (worker_args) {
                                .id     = i,
                                .array  = numbers,
                                .result = 0,
                        };
                }

                pthread_t thread_ids[NTHREADS];

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

                // Wait for threads to finish
                for (size_t i = 0; i < NTHREADS; i++) {
                        pthread_join(thread_ids[i], NULL);
                }

                long sum = 0;

                // Calculate total sum
                for (size_t i = 0; i < NTHREADS; i++) {
                        sum += args[i].result;
                }

                printf("Run %2zu: total sum is %ld\n", n, sum);
        }

        free(args);
        free(numbers);
}
```

Question 3: Performance and benchmarking

The execution time of a program can be measured by using the `time` command. Taking the example from the next exercise, we can measure how long it takes to run by running the `time` command like so:

```
time ./mutex
4000000

real    0m0.420s
user    0m0.999s
sys     0m1.234s
```

The *real* time is the wall clock time taken to run the program. The *user* time is the total CPU time used, and the *system* time is the time spent in system calls or the kernel. Here you can see that the CPU time spent is larger than the wall clock time spent, so this was executing on multiple cores simultaneously.

Sometimes it is better to have finer-grained measures of time, for example if your program has a large setup phase, and you only want to measure some operation which occurs after that. For this you can use the `clock()` function. This provides you with microsecond accuracy. You can achieve nanosecond accuracy using by using operating system specific functions.

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

int main(void) {

    const clock_t tick = clock();

    int ops = 0;
    for (int i = 0; i < 10000000; i++) {
        ops += i;
    }

    const clock_t tock = clock();
    printf("Time elapsed: %fs\n", (double) (tock - tick) / CLOCKS_PER_SEC);

    return 0;
}
```

Question 4: Matrix multiplication

The code below performs matrix multiplication.

1. Analyse the program's performance:

To use `gprof`, you need to compile with profiling using `gcc` and the `-pg` compiler flag.

```
$ gcc -g -pg -Wall -Werror -std=gnu11 matrix.c -o matrix -pthread
```

(a) Ensure the that `-pg` flag has been used to compile the program.

(b) Execute the program as usual.

(c) After the program has finished, execute the profiler:

```
$ gprof ./matrix > mm.stats
```

(d) View the saved information using `less`, which function is the bottleneck in the code and why?

(e) You can use `gprof -l` to show statistics for each line of the source.

2. Swap of the order of the loops so that it is `y, k, x` instead of `y, x, k`.
   Why does this have any effect on the performance of the code?

3. Improve the performance by using `pthreads` for parallelism.

4. Measure how long it takes to run with and without parallelism using `time`. Recommend that you use the lab machines to benchmark as they are similar to the machines used for the upcoming assignments. When benchmarking, it's important that no other programs are running since they may affect the results.

5. Compare your performance with the cache optimised version in
   What every programmer should know about memory by Ulrich Drepper.

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

#define WIDTH 512
#define IDX(x, y) ((y) * WIDTH + (x))

/**
 * Returns the matrix multiplication of a and b.
 */
float* multiply(const float* a, const float* b) {

    float* result = calloc(WIDTH * WIDTH, sizeof(float));

    for (size_t y = 0; y < WIDTH; y++) {
        for (size_t x = 0; x < WIDTH; x++) {
            for (size_t k = 0; k < WIDTH; k++) {
                result[IDX(x, y)] += a[IDX(k, y)] * b[IDX(x, k)];
            }
        }
    }

    return result;
}

/**
 * Returns a Hadamard matrix, if H is Hadamard matrix, then
 * HH^T = nI, where I is the identity matrix and n is the width.
 * Easy to verify that the matrix multiplication was done correctly.
 *
 * Sylvester's construction implemented here only works
 * for matrices that have width that is a power of 2.
 *
 * Note that this construction produces matrices that are symmetric.
 */
float* hadamard(void) {

    // Ensure the width is a power of 2
    assert(((WIDTH - 1) & WIDTH) == 0);

    size_t w = WIDTH;
    size_t quad_size = 1;

    float* result = malloc(WIDTH * WIDTH * sizeof(float));

    result[0] = 1;
    while ((w >>= 1) != 0) {
        // Duplicate the upper left quadrant into the other three quadrants
```

```
        for (size_t y = 0; y < quad_size; ++y) {
            for (size_t x = 0; x < quad_size; ++x) {
                const float v = result[IDX(x, y)];
                result[IDX(x + quad_size, y)] = v;
                result[IDX(x, y + quad_size)] = v;
                result[IDX(x + quad_size, y + quad_size)] = -v;
            }
        }

        quad_size *= 2;
    }

    return result;
}

// Displays a matrix.
void display(const float* matrix) {

    for (size_t y = 0; y < WIDTH; y++) {
        for (size_t x = 0; x < WIDTH; x++) {
            printf("%6.2f ", matrix[IDX(x, y)]);
        }
        printf("\n");
    }
}


int main(void) {

    // Construct the matrices
    float* a = hadamard();
    float* b = hadamard();

    // Compute the result
    float* r = multiply(a, b);

    // Verify the result
    for (size_t y = 0; y < WIDTH; y++) {
        for (size_t x = 0; x < WIDTH; x++) {
            assert(x == y ? r[IDX(x, y)] == WIDTH : r[IDX(x, y)] == 0);
        }
    }

    puts("done");
```

```
    free(a);
    free(b);
    free(r);

    return 0;
}
```

Question 5: Critical sections and mutual exclusion

A critical section is a piece of code that accesses a shared resource that must not be be concurrently accessed by more than one thread. These are usually enforced using locks or other synchronisation measures.

Mutexes are a way of enforcing a critical section. In our threads, we can call `pthread_mutex_lock` in order to obtain ownership of the mutex before entering the critical section. If the thread cannot obtain ownership of the mutex object, it will wait until the mutex has been unlocked. The thread that owns the mutex must call `pthread_mutex_unlock` after it has exited the critical section.

The example below shows a typical use of mutexes in `pthreads`. This mutex is statically initialised using `PTHREAD_MUTEX_INITIALIZER`. You can also create them dynamically using `pthread_mutex_init`.

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

#define THREADS 4
#define LOOPS 1000000

static unsigned counter = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void* worker(void *arg) {

    for (unsigned i = 0; i < LOOPS; i++) {
        // attempt to lock the mutex...
        // thread will wait when mutex is already locked
        pthread_mutex_lock(&mutex);
        // only one thread will be able execute this code
        counter += 1;
        // unlock the mutex after the critical section
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

int main(void) {
```

```
    pthread_t thread_ids[THREADS];

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

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

    printf("%d\n", counter);
}
```

1. Run the code and verify that it outputs the result correctly.
2. Comment out the `pthread_mutex_unlock` line, does the program behave in a way you expect?
3. Uncomment out both `pthread_mutex_lock` and `pthread_mutex_unlock` lines, repeat the program until it outputs an incorrect result, why does this happen?

Question 6: Sockets in C

For more background and details, refer to Chapter 11 of the course textbook.

This is a practice exercise to play with sockets in C. A socket is one endpoint used by a computer to communicate over a network. In UNIX based systems, a socket is represented as a file. The file can be accessed using it's file descriptor with low level I/O to read and write operations already discussd in this course.

A network connection in TCP/IP is defined by 2 sockets (endpoints), it is the 4 tuple.

```
<address1:port1> <address2:port2>
```

In networking, there is a client and server relationship. A server is a computer system which provides (serves) a service on the network; file server, print server, audio streaming server, web server. A client connects to the server to request a service; fetch a file, print a document, play a radio station, fetch a webpage. The client must first establishes a connection, and thereafter, any data can be exchanged between the two computers.

To setup a client and server, there is an order of operations. The server must first bind to a port on it's IP address, and then listen (wait) for new connections. A client will connect to a server on a given IP address and port number. The server accepts a new connection via a new socket file descriptor and then two-way communication can commence with read and write operations.

An example of an echo server. It receives a message from the client and sends the same message in reply. It will begin and listen on a port. When a connection is accepted it will:

- read a message from the new connection (client socket)
- construct a reply of the same message
- send the message to the new connection (client socket)
- close the client socket

See the full code in utilties. Note the `bind(), listen(), and accept()` functions are just for server:

```c
int server_fd;
struct sockaddr_in address;
{ /* fill in struct  */ }

if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
{ /* error case */ }

if (listen(server_fd, 3) < 0) { /* error case */ }

/* start waiting for clients */
        int new_socket;
    socklen_t addrlen;
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (s
    { /* error case */ }
```

Compile with:

```
gcc -o server server.c
```

An example of an echo client. It sends a message to the server and receives the same message in reply. It will first attempt to connect to a server on a given address and port number, once a connection is established it will:

- send the message to the new connection (server socket)
- read a message from the new connection (client socket)
- close the server socket
- terminate

See the full code in utilties. Note the `connect()` function is just for clients:

```c
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0
{ /* error case */ }
```

Compile with:

```
gcc -o client client.c
```

Use these examples on your lab computer to send echo requests and receive replies on the loopback IP address 127.0.0.1 (the local computer) and a chosen port number from 1025 - 65535.

Next add an artificial delay in your server code for reply. Wait 10 seconds first, then send the reply. Using this modified server, attempt to connect two clients to the same server. What is the behaviour of the two clients attempting to access the server at the same time (within the 10 seconds)?

Next write an implementation that will service a client in the echo server by forking a new child process and executing the code for the new client socket. Allow any number of connections to be made in parallel with the server. Retry with two clients and report the behavioural change.

Next write an implementation that will use threads, not processes, for each new client socket accepted by the server.

As a discussion in your tutorial, what are the differences between the fork() vs thread based implementation of handling clients? what are the differences in resources allocated and attention required when they end?

Modify the echo server to accept messages of 4MB in length, and adjusted your buffer accordingly with dynamic memory. Are there any differences? Suppose you did this between two computers of different IP addresses, does anything change?