COMP2129 Week 8 Tutorial

Sorting, threading

Exercise 1: Sorting Integers

The C standard library comes with a function for sorting arrays, called **`qsort`**. Look up the function prototype in the manpages: **`man qsort`**. The function takes four parameters: A pointer to where the array starts **`base`**, the number of elements in the array **`nel`**, the width of each element in the array **`width`**, and a function pointer to a comparison function **`compar`**.

```c
void
qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));
```

A *function pointer* is literally a pointer to the block of executable code corresponding to that function, along with some type information (such as the function's return type and arguments). Function pointers are used to pass functions (blocks of code) to other functions.

The comparison function is called on pairs of pointers to array elements. For example, when sorting an array of type `int`, the comparison function will be passed two integer pointers. Let's call the function f, it should satisfy the following rules:

$$\begin{aligned} f(a,b) < 0 &\quad \text{if } a < b \\ f(a,b) = 0 &\quad \text{if } a = b \\ f(a,b) > 0 &\quad \text{if } a > b \end{aligned}$$

Here is one such function which sorts signed integers in ascending order:

```c
int intcompare(const void *x, const void *y) {
  const int *a = x;
  const int *b = y;
  return (*a - *b);
}
```

Since the comparison function arguments have type `void *`, they need to be cast to the appropriate pointer type before dereferencing.

1. Create a static array containing both positive and negative integers, in no particular order.
2. Sort the integers in ascending order, and print them out.
3. Sort the integers in descending order, and print them out.
4. Sort the integers in increasing magnitude (absolute value), so that a sorted array might look like $[0, -1, 2, 5, -7, 9, -11, 13, 14]$.

Exercise 2: Sorting based on multiple attributes

The `qsort()` function can sort anything, given the right comparison function. Go to eLearning or Piazza, and download `marks.txt`, a file containing (fake) unikeys and corresponding subject marks (in no particular order). These could be stored in a struct:

```c
typedef struct {
  char unikey[9];
  int mark;
} MarkRow;
```

Use `fgets()` and `sscanf` to read the student data into an array of structs in a C program. Sort the array first by mark (highest first), then by unikey (alphabetical order, using `strcmp()`). This means that if two students have the same mark, then they should be listed next to each other, in alphabetical order.

Part of an example output might look like:

```
cwji3259 98
pyjk5649 98
pzrp2468 98
ebfa9170 89
fdlq4105 89
spsg7281 79
```

1. Implement the sorting program described above.
2. Try using the `strncmp` function to compare only the first 2 characters of the student's unikey. (You may have to make up some more data to see the effects of this).

Exercise 3: Searching

Sorted arrays can be searched quickly (in $O(\log n)$ time) using binary search. The standard library has an implementation of binary search, called `bsearch()`, which has a similar function signature to `qsort()`. The function signature differs in two ways:

1. There is an extra argument: the element (key) the caller is looking for.
2. The return type is a `void *`, which will be `NULL` if no match was found, or point to any match if a match was found.

Use the `bsearch()` function to search for any student with a particular mark from the previous exercise.

Hint: The comparison function given to `bsearch()` needs to be consistent (but perhaps not identical) to the comparison function used to sort the array. For example, given the sort from the previous question, a compatible comparison function would be only comparing the marks and disregarding the names. However, an incompatible comparison function would be comparing the names.

POSIX threads

POSIX threads are usually referred to as Pthread and are the POSIX standard for threads which define an API for creating and manipulating threads. To use the Pthreads library within a C program, the header **pthread.h** must be included, and the program must be linked with the pthread library by using the **-lpthread** compiler flag.

pthread.h defines the datatype **pthread_t**, which represents a unique thread ID. To create a thread, make space for a **pthread_t**, and then call the **pthread_create()** function.

```
pthread_t thread;
pthread_create(&thread, NULL, worker, NULL);
```

This statement creates a thread, stores the thread ID into variable **thread**, and starts the execution of the thread function **worker** with a **NULL** argument.

Note that the function **worker** must be a function that takes a **void** pointer as an argument and returns a void pointer. For example, the following function **f** increments an integer variable by 1, and would be a valid worker function.

```
void* increment(void* x) {
  int* i = (int *) x;
  (*i)++;
}
```

To wait for a worker to finish, the main thread can call **pthread_join()** on a particular **pthread_t**, which will cause the caller to block at that statement until the worker finishes.

```
pthread_join(thread, NULL);
```

It is essential to keep the main thread running while the workers are running, since if the main thread exits, then the program is terminated. Joining on all threads also acts as a synchronisation mechanism, since afterwards the main thread can be sure that all of the workers have finished and returned results.

Exercise 4: Off to the races

This exercise demonstrates a simple example of a *data race*, which can happen when operations do not happen atomically.

The following code is available on eLearning or Piazza as **race.c**.

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

#define NTHREADS (4)
#define BIGNUM   (10000000)

int counter = 0;

void *worker(void *arg) {
  for (int i = 0; i < BIGNUM; i++)
    counter++;
  return NULL;
}

int main(void) {
  pthread_t threads[NTHREADS];
  for (int i = 0; i < NTHREADS; i++)
    pthread_create(&threads[i], NULL, worker, NULL);

  for (int i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);

  printf("counter  = %d\n", counter);
  printf("expected = %d\n", NTHREADS * BIGNUM);
  return 0;
}
```

Compile the code:

```
$ gcc -o race race.c -lpthread -std=gnu99
```

Run the program multiple times, and observe the results.

Discuss the following:

- What does the **pthread_join** function do?
- When **NTHREADS** is set to 1, is the single-threaded behaviour recovered? How many threads are actually running in this case?
- When **BIGNUM** is set to a low value, like 5, what happens? Why?

Exercise 5: Parallelise work

One of the benefits of using threads on computers with multiple processors or cores is that work can be happening at the same time: for example an array could (potentially) be added up in $1/4$ the time, using 4 cores at once. (Often it is very hard to achieve this level of speedup, even for trivial problems).

Worker functions can be passed a single argument of type `void *`, and they can return a single argument of type `void *`. It is a common pattern to wrap multiple arguments in a `struct` to pass them into a thread worker function.

Sum the numbers from 1 to 1000 using four threads of execution. Initialise an array of type `int *` in the main thread, then pass the four worker threads arguments using a pointer to the following struct:

```
typedef struct {
  int *array;
  int start, end;
  int answer;
} WorkerArgs;
```

The `.array` member should hold a pointer to the underlying array, and the `.start` and `.end` members should hold the range of the array the worker is responsible for. The worker can then fill in the `.answer` member of the struct when finished, for the main thread to recombine.

A worker function might start like:

```
void *worker(void *arg) {
  WorkerArgs wargs = (WorkerArgs *) arg;
  // Sum wargs.start to wargs.end of wargs.array
  // return NULL, or return a pointer to wargs.answer.
}
```

Some code in the main thread might look like:

```
WorkerArgs wargs[NTHREADS];

// Initialise each struct

for (int i = 0; i < NTHREADS; i++)
  pthread_create(&threads[i], NULL, worker, (void *) &wargs[i]);

// Wait for the threads to finish

int sum = 0;
for (int i = 0; i < NTHREADS; i++)
  sum += wargs[i].answer;
```