THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Week 12 Tutorial

Recursion, Aliasing, Thread Patterns

Question 1: Recursion Overhead

Some algorithms are much more easily thought about as being *recursive*, in that they call themselves, with a smaller version of the problem. This is great for describing algorithms, but sometimes not such a great way of implementing them. For example in C, every function call pushes a new *stack frame* onto the program stack. This stack frame is very large, at least in comparison to the bare minimum information needed to recurse. If too many frames get pushed on, the stack will overflow and a segmentation fault will occur.

A string is a palindrome if it reads the same forwards and backwards. For example, the strings `"noon"`, `"madam"`, and `"racecar"` are all palindromes. If we define the empty string `""` to also be a palindrome, this leads to a very nice recursive algorithm, by repeatedly trimming the ends of the string off.

```c
bool is_palindrome(char * s, size_t len) {

        if (len <= 1) {
                return true;
        } else if (s[0] != s[len - 1]) {
                return false;
        }
        return is_palindrome(s + 1, len - 2);
}
```

- Draw the recursive calls the algorithm makes to check the strings `"noon"` yes, `"madam"` yes and `"racecar"` no
- Copy the function into a C file, and make sure it works by running it on some test strings.
- Create a very long (around 100, 000 characters) palindrome. Check the algorithm works. How long can you make the string until you get a segmentation fault?
- Write an iterative version of the same function, by putting the function body inside a loop, and replacing the recursive call at the end with just a reassignment of the variables `str` and `len`. Ensure that it passes all of your previous tests, and does not cause a stack overflow for large strings.

- See if you can test the speed of the recursive and iterative functions. How much faster is the iterative version? Note that on higher optimisation levels, the compiler will attempt to automatically perform tail call optimisation if the function is marked as static and inline. Try different optimisation levels and examine the assembly to see if the function was optimised into a loop.

  If your input string was static, then the compiler may also precompute the answer and optimise out the function call entirely, this is possible because function is pure and does not introduce any side effects.

Sources of Performance Loss
1. Overhead (What is this? Where could it come from?)
   Communication - sequential solutions don't need to communicate between threads
   Synchronisation - waiting for mutex
   Computation - initialisation of counters when executing in parallel
   Memory - padding adds to the total memory of the program
2. Non-parallelizable computations
   Refers to Amdahl's low for computation on how much could be lost
3. Idle times
   Use thread pool to reduce idle times
4. Contention for resources
   e.g. highly contented mutex

Literals

Numeric literals in C are typed. By default, if you specify an integral value, it will be typed as an `int`. If your value is decimal, it will be typed as an `double`. You can also request your literal to be of a specific type by appending one of the following suffixes to your numeric constants.

SUFFIX	TYPE	EXAMPLE
`U`	`unsigned`	`1U`
`UL`	`unsigned long`	`1UL`
`ULL`	`unsigned long long`	`1ULL`
`L`	`long`	`1L`
`LL`	`long long`	`1LL`
`f`	`float`	`1.0f`

Note that the suffixes are case insensitive in C, so `1u` is also valid for `unsigned` literals.

Casting

In C, you can explicitly convert between integer types and pointer types by casting.

```
int x = (int) 3.9; // will be truncated to 3
float y = 2.3 + x; // some casts are implicit and happen automatically
```

In the example above, x is converted to a *double*, then added to *2.3* (which is also a double). The result is then converted to *float* and saved in *y*.

A cast from a signed integer to an unsigned integer results in no changes in the underlying bitpattern, only the interpretation of the data changes.

```
// will output -1 4294967295 when sizeof(int) is 4
printf("%d %u\n", -1, (unsigned) -1);
```

A cast from an *unsigned* integer to a *signed* integer is valid only if the value is within the range of the signed integer, otherwise the behaviour is implementation defined (decided by the compiler).

A non const pointer can be implicitly converted to a `const` pointer without an explicit cast. Conversion from `void *` to any pointer type does not require an explicit cast and vice versa.

Promotions

C can also change the type of your integer variables implicitly through promotions. The rules are as follows:

- If an integer type is used in an operation with another integer type of greater size (sizeof), the original type is *promoted* (casted) to the larger type
- If a signed integer type is used in an operation with an *unsigned* integer type, the *unsigned* integer type "wins" and the *signed* type is promoted (casted) to the *unsigned* type.

```
int x = 1;
unsigned y = x; // this works as you would expect

int z = (int) y; // requires cast
signed char w = (signed char) y; // requires cast and truncates to a char
```

Promotions can lead to some very unintuitive behaviour, thus you should avoid using different integer types in the same expression:

```
printf("%d", 1U > -1); // evaluates to 0 (false)
printf("%d", (unsigned short) 1 > -1); // evaluates to 1 (true)
printf("%d", -1L > 1U); // evaluates to 0 or 1 depending on your machine
```

Question 2: Casting and Promotions

- What implicit conversions are performed by the compiler for every line of code below?

```
char x = 1;          int to char
long long y = 1;     int to long long
int z = x;           char to int
float a = 0.1;       double to float
const char * ptr = &x;   char* to const char*
```

- What implicit type promotions and conversions are present in the code below?

```
int x = 1U - 1;          -1 (int) to unsigned
int y = 'a' - 1;         'a' (char) to int
unsigned y = 'a' < 1;    'a' (char) to int to unsigned
float z = 0.1 + 0.2f + 1;   1 (int) to float
```

- Take the C integer quiz to verify your understanding of casting and promotions.
- Why is there no implicit cast allowed between a `char**` to a `const char**`?

it violates const correctness http://c-faq.com/ansi/constmismatch.html

Aliasing and the `restrict` keyword

Two pointers are said to *alias* when they are used to access the same underlying region of memory. The C11 standard allows compilers to make assumptions on what pointers alias. The *strict aliasing* rule specifies that compilers are allowed to assume that the same underlying memory will not be accessed through pointers of different *types* with the exception of `char*` , that is, the compiler is allowed to assume that two pointers are of different types will not alias unless one of them is of type `char*`.

```
void f(float * x, int * y) {
        printf("%d\n", *y);
        *x = 1234.0f;
        printf("%d\n", *y);
}
```

It's possible for there to be two different outputs if `x` and `y` point to the same address or overlap in some way.

Suppose we follow the strict aliasing rule and the compiler assumes that the pointers do not alias. The compiler can cache the value of `*y` and avoid loading it from memory a second time. Alternatively, the compiler may choose the reorder instructions for performance. However, if the pointers do alias and the compiler performs this optimisation, then result will surprise the us. Suppose now there is a different function:

```
void y(char * x, char * y)
```

The compiler cannot assume that `x` and `y` do not alias since they are of the same type. However, if we know they will not alias, then we can tell the compiler this fact by using the `restrict` keyword.

```
void y(char * restrict x, char * restrict y)
```

In doing so, we tell compiler that the pointers will not alias and this allows the compiler to perform optimisations that previously would not be possible.

Question 3: Strict aliasing, restrict and optimisations

- Read the strict aliasing horror stories on StackOverflow.
- Does the following code violate strict aliasing?

```
union {
        float x;
        uint32_t y;
} u = { .x = 123.4f };

u.x = 2;
u.y = 123;
```

No using a union to type pun is fine.
Alternative you could use memcpy.
https://en.wikipedia.org/wiki/Type_punning

- Does the following code violate strict aliasing?

```
float x = 123.4f;
x = 2;
*(uint32_t *) &x = 123;
```

Yes. The same memory is accessed through a uint32_t* and a int*

- Does the following code violate strict aliasing?

```
float x = 123.4f;
x = 2;
uint32_t y = 123;
memcpy(&x, &y, sizeof(uint32_t));
```

No. Memcpy is the alternative way to type pun without violating strict aliasing.

- On -O1, the compiler optimises the following code to a call to memcpy.

```
void cpy1(char * restrict dest, char * restrict src, size_t n) {
        for (size_t i = 0; i < n; ++i) {
                dest[i] = src[i];
        }
}
```

- Why is it not allowed to optimise the following code to call memcpy.

```
void cpy2(char * dest, char * src, size_t n) {
        for (size_t i = 0; i < n; ++i) {
                dest[i] = src[i];
        }
}
```

- Why is it also invalid for the compiler to optimise the code above to use memmove?

The compiler cannot optimise cpy2 to memcpy because the memory addresses can interleave, and memcpy is declared using restrict pointers, implying that they will not interleave.
Memmove will not copy destructively as it copies backwards, however, in certain cases, the output will be different.

Question 4: Thread Pool

Starting and joining threads can be very expensive and in practice we want to minimise or completely remove recreating threads. You are tasked with creating a thread pool that will create and start a set number of threads using the function `thread_pool_new(size_t n)`. Each thread will be in a waiting state until they have received `work` to do. Implement a round-robin scheme, where when new work is given to the pool, it will allocate it to next thread in the round.

Use the following function and struct declarations to build a thread pool.

```
struct thread_data;

struct thread_pool;

struct thread_job* thread_job_new(void(*fn)(void*), void* data);

void thread_job_destroy(struct thread_job* j);

void* thread_pool_work(void* arg);

struct thread_pool* thread_pool_new(size_t n);

void thread_pool_execute(struct thread_pool* pool,
        struct thread_job* job);

void thread_pool_destroy(struct thread_pool* pool);
```

You may change the functions if you want.

After implementing your thread pool, try modifying it so work is allocated to the next available thread.

You may use the example on the next page as a way of simulating the threadpool.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "thread_pool.h"
#include <stdint.h>

#define MAIN_THREAD_TIMEOUT (60)
#define N_JOBS (200)

struct work_data {
        uint32_t time;
        uint32_t work_id;
};

void mysleep(void* arg) {
        struct work_data* data = (struct work_data*) arg;
        printf("is working on job %u\n", data->work_id);
        fflush(stdout);
        sleep(data->time);
        free(data);
}

int main() {

        struct thread_pool* pool = thread_pool_new(8);

        for(size_t i = 0; i < N_JOBS; i++) {
                struct work_data* d = malloc(sizeof(struct work_data));
                d->time = 1;
                d->work_id = i;
                    struct thread_job* job = thread_job_new(mysleep, d);
                thread_pool_execute(pool, job);
        }
        sleep(MAIN_THREAD_TIMEOUT);
        thread_pool_destroy(pool);
        puts("Finishing up now!");
        return 0;
}
```