COMP1521

W10 - Concurrency and Processes

Overview

Admin

Processes (Q2)

Threads (Q7, Q5, Q6)

Concurrency (Q8, Q9, Q10)

Admin

Assignment 2 due on Friday (tomorrow)

If you go to a help session, show up a bit early to make sure you can join the queue quickly. At this point, not all students will be seen to in help sessions considering how busy they are.

W9 weekly test due today.

W10 weekly test released today, due Thursday W11.

W10 labs due next monday.

Practice exam will be run in this weeks lab (although there is still a W10 lab). We will do 1 room is lab/assignment and the other is prac exam.

Processes

Running a process

You can run a process yourself pretty easily on the command line. For example, you can run `dcc prog.c -o prog` to invoke the dcc program.

You can also run a process from a C program, by taking advantage of some special functions.

posix_spawn

posix_spawn is a function that can be used to run processes from inside your C program.

```c
#include <spawn.h>
#include <stdlib.h>

extern char **environ;

int main(void) {
    pid_t pid;
    char *argv[] = {"/bin/echo", "hello", "world", NULL};       // Set up argv
    posix_spawn(&pid, "/bin/echo", NULL, NULL, argv, environ);  // Execute process
    waitpid(pid, NULL, 0);                                      // Wait for process to finish
    return 0;
}
```

posix_spawn

posix_spawn has a load of arguments, but only a few of them really matter for us.

pid_t *pid: a pointer to a pid_t variable which can be used to manage our spawned process.

char *path: path to the executable we should run

file_actions, attrp: just set to NULL, don’t care

char *argv[]: argv to be provided to new process

char *envp: environment variables

```
int
posix_spawn(pid_t *restrict pid, const char *restrict path, const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);
```

waitpid

We must wait for the spawned process to terminate before we return 0, otherwise it will be killed early when our program terminates.

We can use waitpid for this: just provide the pid as the first argument.

Second argument allows us to get the return status if we want by providing pointer to an int, but if we don't care can just give NULL.

Third argument is for options, but we don't care so just set to 0.

Tutorial Q2

2. Write a C program, now.c, which prints the following information:

    1. The current date.
    2. The current time.
    3. The current user.
    4. The current hostname.
    5. The current working directory.

```
$ dcc now.c -o now
$ ./now
29-02-2022
03:59:60
cs1521
zappa.orchestra.cse.unsw.EDU.AU
/home/cs1521/lab08
```

Threads

Threads

Normally a program just runs on one “thread”.

With special functions you can split your program between multiple “threads”.

This can potentially allow your program to take advantage of multi-core processors (which are capable of executing multiple threads simultaneously).

Even if you only have one core, it can still be a good idea.

For example, if one thread is waiting for user input, the CPU can just run the other thread in the meantime.

For the rest of the exercises we will be talking about threads on single core machines

Tutorial Q7

7. Concurrency can allow our programs to perform certain actions simultaneously that were previously tricky for us to do as COMP1521 students.

   For example, with our current C knowledge, we cannot execute any code while waiting for input (with, for example, `scanf`, `fgets`, etc.).

   Write a C program that creates a thread which infinitely prints the message `"feed me input!\n"` once per second (*sleep*), while the main (default) thread continuously reads in lines of input, and prints those lines back out to `stdout` with the prefix: `"you entered: "`.

pthread_create

pthread_create spawns a new thread allowing your program to split itself between threads.

thread: pointer to pthread_t which allows you to manage the state of the thread

attr: we don't care, just leave as NULL

start_routine: the name of a function that the thread should execute

arg: the argument to provide to the above function (as a void pointer)

- In order to use this argument you must cast it to the correct type of pointer

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

pthread_join

pthread_join is the equivalent of waitpid. It waits for thread to finish execution.

This must be done, otherwise when your program terminates (return 0) the thread will be abruptly killed.

thread: the pthread_t that was passed to pthread_create

value_ptr: a pointer which can be used to store a return value from the thread. We can just set to NULL

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

Tutorial Q5

5. Write a C program that creates a thread that infinitely prints some message provided by main (eg. `"Hello\n"`), while the main (default) thread infinitely prints a different message (eg. `"there!\n"`).

Tutorial Q6

6. The following C program attempts to say hello from another thread:

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

void *thread_run(void *data) {
    printf("Hello from thread!\n");

    return NULL;
}

int main(void) {
    pthread_t thread;
    pthread_create(
        &thread,    // the pthread_t handle that will represent this thread
        NULL,       // thread-attributes -- we usually just leave this NULL
        thread_run, // the function that the thread should start executing
        NULL        // data we want to pass to the thread -- this will be
                    // given in the `void *data` argument above
    );

    return 0;
}
```

However, when running this program after compiling with `clang`, the thread doesn't say hello.

```
$ clang -pthread program.c -o program
$ ./program
$ ./program
$ ./program
```

Why does our program exhibit such behaviour?

How can we fix it?

Concurrency

Concurrency

What if two threads try and increment (add one) a variable at the same time?

Think about the assembly instructions:

Thread 1	Thread 2
Variable = 0 Load value of variable to register: 0 -> switch execution Addi instruction: 1 Store variable to memory: store 1	Variable = 0 Load value of variable to register: 0 Addi instruction: 1 Store variable to memory: store 1 <- switch execution

Oh no! We lost an increment!

How to avoid?

Mutual exclusion: only one process can be editing the variable at any given time.

Create a single “lock”, which must be “acquired” before beginning to edit the variable.

Thread 1	Thread 2
Variable = 0 Try acquire LOCK: success Load value of variable to register: 0 -> switch execution Addi instruction: 1 switch execution -> Store variable to memory: store 1 Release LOCK switch execution ->	Variable = 0 Try acquire LOCK: failure, thread 1 has it <- switch execution Try acquire LOCK: failure, thread 1 has it switch execution <- Try acquire LOCK: success …

Mutual exclusion

Initialise a lock:

```
pthread_mutex_t global_total_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Acquire a lock:

```
pthread_mutex_lock(&global_total_mutex);
```

Release a lock:

```
pthread_mutex_unlock(&global_total_mutex);
```

Tutorial Q8

8. The following C program attempts to increment a global variable in two different threads, 5000 times each.

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

int global_total = 0;

void *add_5000_to_counter(void *data) {
    for (int i = 0; i < 5000; i++) {
        // sleep for 1 nanosecond
        nanosleep (&(struct timespec){.tv_nsec = 1}, NULL);

        // increment the global total by 1
        global_total++;
    }

    return NULL;
}

int main(void) {
    pthread_t thread1;
    pthread_create(&thread1, NULL, add_5000_to_counter, NULL);

    pthread_t thread2;
    pthread_create(&thread2, NULL, add_5000_to_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // if program works correctly, should print 10000
    printf("Final total: %d\n", global_total);
}
```

Since the global starts at 0, one may reasonably assume the value would total to 10000.

However, when running this program, it often gives differing values each individual execution:

```
$ dcc -pthread program.c -o program
$ ./program
Final total: 9930
$ ./program
Final total: 9983
$ ./program
Final total: 9994
$ ./program
Final total: 9970
$ ./program
Final total: 10000
$ ./program
Final total: 9996
$ ./program
Final total: 9964
$ ./program
Final total: 9999
```

Why does our program exhibit such behaviour?

Tutorial Q9

9. How can we use "mutual exclusion" to fix the previous program?

Mutual Exclusion

Deadlocks: trouble in paradise? Suppose there are two locks, A and B.

Suppose both threads need resources protected by A and B

Thread 1	Thread 2
Try acquire LOCK A: success switch execution -> Try acquire LOCK B: failure switch execution -> …	 Try acquire LOCK B: success <- switch execution Try acquire LOCK A: failure <- switch execution

Help! We are stuck!

Mutual exclusion

The solution is to always acquire locks in the same order.

EG, for three locks A, B, C:

If a process needs some of the resources represented by these locks, it should:

Always acquire LOCK A first

Always acquire LOCK B after LOCK A

Always acquire LOCK C after LOCK B and LOCK A

This makes “deadlocks” impossible to occur.

Atomic operations

The (kinda cheap) way out is to use what are known as “atomic” operations.

There exists an “atomic add” operation which is guaranteed to be only a single CPU instruction.

So, our example from before:

Thread 1	Thread 2
Variable = 0 Atomic Increment Variable (now v=1) switch execution->	Variable = 0 Atomic increment variable (now v=2)

We can see it’s not possible for anything to go wrong, even without using locks

Atomic operations

Declare atomic variable:

```
atomic_int global_total = 0;
```

Now we can write something like:

```
global_total++;
```

And it’s safe from concurrency bugs.

Concurrency bugs are actually called “race conditions”, since both threads “race” to do something at the same time when they should be taking turns.

Tutorial Q10

10. How can we use atomic types to fix the previous program?