COMP1521

W10 - Concurrency and Processes

Overview

Admin

Processes (Q2)

Threads (Q7, Q5, Q6)

Concurrency (Q8, Q9, Q10)

Admin

Assignment 2 due on Friday

W9 weekly test due thursday.

W10 weekly test released thursday, 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.

Admin

Pls do myexperience 🙏

Processes

Running a process

```
z5420273@vx20:~/1521/labs/lab08$ which dcc
/usr/local/bin/dcc
z5420273@vx20:~/1521/labs/lab08$ dcc -o create_binary_file create_binary_file.c
```

You can invoke a program known to your system’s $PATH environment variable from the command line.

For example, you have run the process /usr/local/bin/dcc maybe once or twice.

Running a process: from another process

It’s possible to write a program to run a process for you. A simple example:

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

int main(void) {
    system("echo hello world");
    return 0;
}
```

This program runs /bin/echo as if it was invoked by a terminal.

Running a process: fork and execve

System is typically not used in “proper” C code, since it is brittle and easily can pose security issues when user input is included in the “system” command.

Another common method is the use of the execv family of functions.

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        char * const argv[] = {"/bin/echo", "hello", "world", NULL};
        execve("/bin/echo", argv, NULL);
    } else {
        waitpid(child_pid, NULL, 0);
        printf("child process finished executing\n");
    }
}
```

Running a process: fork and execve

Fork is a very confusing function. It creates a second copy of your process.

In one copy, fork returns the pid (process ID) of the child process - this is the parent.

In the other copy, fork returns 0 - this is the child.

This difference is used to make the parent and child process behave differently.

```c
if (child_pid == 0) {
    // this is the child
    char * const argv[] = {"/bin/echo", "hello", "world", NULL};
    execve("/bin/echo", argv, NULL);
} else {
    // this is the parent
    waitpid(child_pid, NULL, 0);
    printf("child process finished executing\n");
}
```

Running a process: fork and execve

Execve is also a little confusing. Unlike system, it does not create a new child process to run the new process - it replaces itself.

This is why fork must be used - otherwise we would replace the process with execve and never be able to do anything else.

Running a process: fork and execve

The parent and child will each execute a different branch of the if statement.

```
child_pid = fork()



|     ---CHILD PROCESS---      |      ---PARENT PROCESS---      |
|                              |                                |
|      if (child_pid == 0)     |              else              |
|                              |                                |
|         child_pid = 0        |        child_pid = 12446       |
|          execve(...)         |    waitpid(12446, NULL, 0)     |
|                              |          printf(...)           |
```

Running a process: posix_spawn

Execve is also quite brittle and, although better than system, it is not best practice.

Best practice is to use a function called posix_spawn, which doesn't require the use of fork().

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

int main(void) {
    pid_t pid;
    char * const argv[] = {"/bin/echo", "hello", "world", NULL};
    posix_spawn(&pid, "/bin/echo", NULL, NULL, argv, NULL);
    waitpid(pid, NULL, 0);
}
```

Running a process: waitpid

In two of the previous examples we used waitpid to “wait” for a process to terminate. This is often required to ensure things happen in a correct and consistent order.

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

int main(void) {
    pid_t pid;
    char * const argv[] = {"/bin/echo", "hello", "world", NULL};
    posix_spawn(&pid, "/bin/echo", NULL, NULL, argv, NULL);
    waitpid(pid, NULL, 0);
}
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        char * const argv[] = {"/bin/echo", "hello", "world", NULL};
        execve("/bin/echo", argv, NULL);
    } else {
        waitpid(child_pid, NULL, 0);
        printf("child process finished executing\n");
    }
}
```

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 is a benefit even on machines with only a single CPU core.

Actions that require the process to wait (like waiting for user input, or performing syscalls) can be performed in the background while a different thread is running.

Creating threads is like a lightweight version of spawning other processes.

Thread: pthread

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

void *thread_action(void *data) {
    printf("hello world!\n");
}

int main(void) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_action, NULL);
    pthread_join(thread, NULL);
}
```

pthread_create is used to spawn a thread. The thread runs the specified function, and you can optionally pass data as a void pointer via the fourth argument

pthread_join is used for the same reason as waitpid - it waits for the thread to finish.

Thread: pthread

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

void *thread_action(void *data) {
    char *str = (char*) data;
    printf("%s", str);
    return NULL;
}

int main(void) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_action, "hello world!\n");
    void **result = NULL;
    pthread_join(thread, result);
    printf("thread result: %p\n", result);
}
```

Above is an example of passing data to a thread and casting it to the correct type.

We can also see how to access the return value of the thread using pthread_join's second argument

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: "`.

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

There is a special class of bugs in multithreaded processes known as “race conditions”.

These bugs occur when the behaviour of a program can vary depending on the order that the threads execute.

```
~/Work/1521-materials/w10 | main !20 ?12
❯ ./race_condition
global=9999
~/Work/1521-materials/w10 | main !20 ?12
❯ ./race_condition
global=10000
~/Work/1521-materials/w10 | main !20 ?12
❯ ./race_condition
global=10000
~/Work/1521-materials/w10 | main !20 ?12
❯ ./race_condition
global=9999
```

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

int global;

void *add_one(void *data) {
    global += 1;
    return NULL;
}

int main(void) {
    global = 0;
    pthread_t *threads = malloc(sizeof(pthread_t) * 10000);
    for (int i = 0; i < 10000; i++) {
        pthread_create(&threads[i], NULL, add_one, NULL);
    }
    for (int i = 0; i < 10000; i++) {
        pthread_join(threads[i], NULL);
    }
    free(threads);
    printf("global=%d\n", global);
}
```

Concurrency: ???

Why would this happen? Consider the low level (assembly-level) function of this code.

The process scheduler will randomly assign CPU time between the threads:

```
|        Thread 1        |        Thread 2         |            Thread code:
|   lw    $t0, global    |                         |        C:
|         ------>        |                         |            global += 1;
|                        |   lw    $t0, global     |
|                        |         <------         |
|   addi $t0, $t0, 1     |                         |        MIPS:
|   sw    $t0, global    |                         |            lw    $t0, global
|        ----->          |                         |            addi  $t0, $t0, 1
|                        |    addi $t0, $t0, 1     |            sw    $t0, global
|                        |    sw    $t0, global    |
|                        |                         |
```

The final value of global only increases by 1 - we “lose” an increment.

Concurrency: How can we avoid this?

One solution is to never let two threads modify a variable simultaneously. We can achieve this using what are known as “mutexes” or “locks”.

```
|        Thread 1        |        Thread 2        |
|   lw    $t0, global    |                        |
|   addi $t0, $t0, 1     |                        |
|   sw    $t0, global    |                        |
|         ----->         |                        |
|                        |   lw    $t0, global    |
|                        |    addi $t0, $t0, 1    |
|                        |    sw    $t0, global   |
|                        |                        |
```

Desired behaviour

Concurrency: locks

Threads must acquire a lock before modifying or accessing a shared resource (IE, global variable).

```
LOCK=init_lock
aqr (aquire)
rel (release)

|        Thread 1        |        Thread 2        |
|   aqr   LOCK           |                        |
|     -SUCCESS-          |                        |
|   lw    $t0, global    |                        |
|        ----->          |     aqr    LOCK        |
|                        |     -FAILURE-          |
|                        |       <------          |
|   addi $t0, $t0, 1     |                        |
|   sw    $t0, global    |                        |
|   rel   LOCK           |                        |
|         ----->         |     aqr LOCK           |
|                        |    -SUCCESS-           |
|                        |   lw    $t0, global    |
|                        |   addi $t0, $t0, 1     |
|                        |   sw    $t0, global    |
|                        |   rel   LOCK           |
|                        |                        |
```

Concurrency: locks in C

Adding a lock to the C program is straightforward.

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

int global;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *add_one(void *data) {
    pthread_mutex_lock(&lock);
    global += 1;
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

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?

Deadlocks

With multiple locks, programmers should be careful to not cause "deadlocks".

If threads need multiple resources, it's possible for deadlocks to occur like shown:

```
LOCK_A=init_lock
LOCK_B=init_lock
aqr (aquire)
rel (release)

|        Thread 1        |        Thread 2        |
|   aqr   LOCK_A         |                        |
|     -SUCCESS-          |                        |
|          ----->        |                        |
|                        |   aqr   LOCK_B         |
|                        |      -SUCCESS-         |
|                        |       <-----           |
|   aqr   LOCK_B         |                        |
|     -FAILURE-          |                        |
|          ----->        |                        |
|                        |   aqr   LOCK_A         |
|                        |      -FAILURE-         |
|                        |       <-----           |
|          ----->        |                        |
|                        |       <-----           |
|          ----->        |                        |
...
```

Deadlocks: avoiding deadlocks

Fortunately avoiding deadlocks is simple. We just need to enforce a “global ordering” of locks.

That means that locks should always be acquired in the same order.

```
LOCK_A=init_lock
LOCK_B=init_lock
aqr (aquire)
rel (release)

GLOBAL ORDERING: LOCK_A then LOCK_B

|        Thread 1        |        Thread 2        |
|   aqr   LOCK_A         |                        |
|     -SUCCESS-          |                        |
|          ----->        |                        |
|                        |   aqr   LOCK_A         |
|                        |     -FAILURE-          |
|                        |      <-----            |
|   aqr   LOCK_B         |                        |
|     -SUCCESS-          |                        |
|        ...             |                        |
|   rel   LOCK_B         |                        |
|   rel   LOCK_A         |                        |
|          ----->        |                        |
|                        |   aqr   LOCK_A         |
|                        |     -SUCCESS-          |
|                        |   aqr   LOCK_B         |
|                        |     -SUCCESS-          |
...
```

Deadlocks: avoiding deadlocks

Locks should always be released in the opposite order to the global order.

This is in case we need to re-acquire the locks later.

```
LOCK_A=init_lock
LOCK_B=init_lock
aqr (aquire)
rel (release)

GLOBAL ORDERING: LOCK_A then LOCK_B

|        Thread 1       |        Thread 2        |
|   aqr   LOCK_A        |                        |
|     -SUCCESS-         |                        |
|         ----->        |                        |
|                       |    aqr   LOCK_A        |
|                       |       -FAILURE-        |
|                       |        <-----          |
|   aqr   LOCK_B        |                        |
|     -SUCCESS-         |                        |
|        ...            |                        |
|   rel   LOCK_A        |                        |
|         ------>       |                        |
|                       |   aqr   LOCK_A         |
|                       |     -SUCCESS-          |
|                       |        <------         |
|  aqr   LOCK_A         |                        |
|    -FAILURE-          |                        |
|         ------>       |                        |
|                       |   aqr   LOCK_B         |
|                       |     -FAILURE-          |
|                       |        <------         |
...
```

Concurrency: atomic operations

A boring way to avoid race conditions is using “atomic operations”. Clearly, the shown threads can never encounter a race condition.

```
|         Thread 1          |         Thread 2          |
|   addone    global        |                           |
|         ----->            |                           |
|                           |   addone    global        |
|                           |         <------           |
```

Concurrency: atomic operations in C

Declare a variable of type atomic_int for example, and use the atomic add operation to add one.

```c
#include <stdatomic.h>

atomic_int global;

void *add_one(void *data) {
    atomic_fetch_add(&global, 1);
    return NULL;
}
```

Tutorial Q10

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