NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Exercise 5

Processes, Threads & Synchronization

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman
Daniel Hansen

Handout: 14.03.02.2023

Introduction

In exercise five, we focus on concurrency, and how it might affect correctness. While concurrency and parallelism is taught in greater detail in other courses (e.g. TDT4200 - Parallel Programming), the operating system plays an essential role, as it needs to support parallelism and multi-threading to enable parallel programming concepts. Therefore, we are introducing the basic principles of synchronisation, processes, and threads in this course.

1 Parallel Execution

Throughout section 1, we refer to the following programs. We omitted the init and boilerplate parts of the program for brevity.

```
int global_variable = 0;
void *count(void *argp) {
    for (int i = 0; i < 1000; ++i)
        global_variable++;
    return 0;
}

int main() {
    pthread_t thread_ids[3];
    for (int i = 0; i < 3; i++)
        pthread_create(&thread_ids[i], NULL, count, NULL);
    for (int i = 0; i < 3; i++)
        pthread_join(thread_ids[i], NULL);
    printf("Parent global_variable: %d\n", global_variable);
    return 0;
}
```

Listing 1: Program Version 1

```
int global_variable = 0;

void count() {
    for (int i = 0; i < 1000; ++i)
        global_variable++;
}

int main()
{
    for (int i = 0; i < 3; i++)
    {
        pid_t pid = fork();
        if (pid == 0)
        {
            count();
            return 0;
        }
    }
    printf("Parent global_variable: %d\n", global_variable);
    return 0;
}
```

Listing 2: Program Version 2

```c
int global_variable = 0;
pthread_mutex_t mut;
void *count(void *argp) {
    for (int i = 0; i < 1000; ++i)
    {
        pthread_mutex_lock(&mut);
        global_variable++;
        pthread_mutex_unlock(&mut);
    }
    return 0;
}

int main() {
    pthread_t thread_ids[3];
    for (int i = 0; i < 3; i++)
        pthread_create(&thread_ids[i], NULL, count, NULL);
    for (int i = 0; i < 3; i++)
        pthread_join(thread_ids[i], NULL);
    printf("Parent global_variable: %d\n", global_variable);
    return 0;
}
```

Listing 3: Program Version 3

1. Which programs will give the same output?

   ☐ V1 and V2

   ☐ V2 and V3

   ☐ All of them will have the same output

   ☐ None of them will have the same output

2. What state is shared between the multiple threads created in Program 1 or 3?

3. Take a look at program 5. What will it output? Explain your answer. What would you change?

   ☐ It will crash because it detects multiple concurrent writes

   ☐ It will always output: `Parent global_variable: -2050885730`

   ☐ It will never complete

   ☐ It will output a different number every time, depending on how the threads are scheduled

   ☐ It will crash because it tries to lock an already locked lock

4. What is the difference between a lock and a semaphore? Can you find an example of when you would use which of those?

5. Given the following instruction (that will execute atomically, so no concurrency), how could we rewrite program 3 without using locks. Can you come up with a similar solution for program 4? What is the difference between program 3 and program 4?

```c
// an atomic compare-and-swap exists on many platforms
int CAS(int* location, int old_value, int new_value)
{
    int val = *location;
    if (val == old_value)
    {
        (*location) = new_value;
```

```
        return 1;
    }
    return 0;
}
```

Listing 4: Pseudo code describing the compare-and-swap instruction

```
int global_variable = 0;
int global_variable2 = 0;
pthread_mutex_t mut;
pthread_mutex_t mut2;

void *count1(void *argp)
{
    for (int i = 0; i < 1000; ++i)
    {
        pthread_mutex_lock(&mut2);
        pthread_mutex_lock(&mut);
        global_variable2++;
        global_variable += global_variable2;
        pthread_mutex_unlock(&mut);
        pthread_mutex_unlock(&mut2);
    }
    return 0;
}
void *count2(void *argp)
{
    for (int i = 0; i < 1000; ++i)
    {
        pthread_mutex_lock(&mut);
        pthread_mutex_lock(&mut2);
        global_variable++;
        global_variable2 += global_variable;
        pthread_mutex_unlock(&mut);
        pthread_mutex_unlock(&mut2);
    }
    return 0;
}

int main()
{
    pthread_t thread_ids[3];
    for (int i = 0; i < 3; i++)
    {
        if (i % 2)
            pthread_create(&thread_ids[i], NULL, count1, NULL);
        else
            pthread_create(&thread_ids[i], NULL, count2, NULL);
    }
    for (int i = 0; i < 3; i++)
        pthread_join(thread_ids[i], NULL);
    printf("Parent global_variable: %d\n", global_variable);
    return 0;
}
```

Listing 5: Program 5

2 Coherence, Consistency, and Synchronization

1. Sequential consistency guarantees us that the result is the same as if we would execute all the operations in sequential order on a single processor. It follows these two rules: 1. Instructions executed by a single processor appear in program order and 2. Instructions executed concurrently (i1 on processor 1 and i2 on processor 2) can appear in any interleaving (e.g. i1, i2 or i2, i1). Given the following two processors and their execution trace, which of the following options are impossible end-states under sequential consistency? Assume that all variables are initialized to 0.

Processor	1		2
Instr. ID		Instr. ID	
p1i1	*p = 1;	p2i1	u = *q;
p1i2	*q = 1;	p2i2	v = *p;

Table 1: Sequence of program execution on two cores

- ☐ u=1, v=1
- ☐ u=0, v=1
- ☐ u=1, v=0
- ☐ u=0, v=0

2. Using the CAS primitive introduced in the previous exercise, can you build a simple lock?