NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 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?

   Answer: *Program 1 will count up, but some of the computations will be overwritten by another thread, thus we don't expect to see the global_variable to reach 3000, but a bit less. Program 2 creates processes instead of threads. Thus the memory is not shared. This results in an unmodified global_variable (=0) for the parent process, thus outputting 0. Program 3 uses a mutex and locks it before modifying the shared global variable and thus is the only program that will print 3000. Thus none of them will have 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?

   Answer: *All threads share the heap and the code regions. However, they use their own respective stacks. The stacks are nevertheless in the same address space, so in theory, threads could access each other's stack, as they all belong to the same process, and thus all virtual addresses are mapped using the same pagetable.*

3. Take a look at program 6. What will it output? Explain your answer. What would you change?

   Answer: *The problem with the code is that some of the threads will lock* `mut` *first while others lock* `mut2` *first before they try to lock the respective another lock. That results in a condition called a deadlock, where we have a circular dependency between the threads (Thread 1 waits for Thread 2 waits for Thread 1 ...). This is a minimal example, and thus the error should be relatively easy to spot. However, remember that these dependency rings can be huge, and many paths might not exhibit such behaviour. Thus it is not always easy to spot such bugs. In all cases, we can fix it by strictly adhering to a locking order, meaning we always try to lock in the same order. Another option (if you can't control the lock, because it might not be within your code, but it might be a device), you can always attempt to lock it, and if that fails, give up, go to sleep, and retry later. The amount that you sleep should not be constant*

to evade the case where you and another process/thread try to access the same resource at exactly the same intervals.

- ☐ 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?

   Answer: *A lock is either locked or not. If it is not locked, a single thread can lock it and proceed to access the protected region. Thus, only ever one thread can take the lock at a time. However, let's assume that we now have a resource that a fixed number n of threads can simultaneously handle. Let n be bigger than 1, but not unlimited. We know the size of n at setup time. In this case, we would need to create n locks to protect that resource and allow efficient use, such that n threads could acquire a lock. This would also mean for the threads to cycle through the locks to find one that is unlocked, and it would require an acquire function that does return control to the caller if it can't acquire a specific lock. In this case, it can be more useful to have a semaphore. A semaphore is initialized with the maximum amount of threads that can proceed to the protected section. A semaphore usually has two functions, wait and signal. A thread that wants to access the protected resource will call wait and either proceed if a slot is available or wait until a slot becomes available. Signalling on the other hand is used to increase the free slot counter and notify the next waiting thread, such that it can continue execution. That reduces a thread's useless work when trying to enter a region (e.g. no cycling on multiple locks).*

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?

```
// 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

Answer: *We can remove the locks in program 3 (as shown in the code block below). For program 4, we cannot just easily replace the value assignment with a CAS, as we have to change two different values. We need to keep a copy of the new value to add it. Keep in mind that this only works because the commutativity of the addition operation.*

```
int global_variable = 0;
pthread_mutex_t mut;
void *count(void *argp) {
    for (int i = 0; i < 1000; ++i)
    {
        int gv = global_variable;
        int cas_rv = CAS(&global_variable, gv, gv+1);
```

```
        while (!cas_rv)
        {
            gv = global_variable;
            cas_rv = CAS(&global_variable, gv, gv+1);
        }
    }
    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 5: Lockfree version of program 3

```
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 6: Program 6

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

Answer:

- ☐ u=1, v=1, Sequence: (a1, a2, b1, b2)
- ☐ u=0, v=1, Sequence: (a1, b1, b2, a2)
- ☒ u=1, v=0, Sequence that would end up in that state: (a2, b1, b2, a1), which breaks sequential consistency because we can't execute a2 before a1.
- ☐ u=0, v= 0, Sequence: (b1, b2, a1, a2)

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

Answer: *See a possible implementation below*

```
// 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;
}

typedef struct lock_
{
    uint8_t lock;
```

```
16      char name[16];
17  } lock;
18
19  // This function only returns when the lock has been acquired
20  void acquire(lock *lock)
21  {
22      // first busy loop on the lock if it becomes available soon.
23      int attempts = 0;
24      int backoff_time = 1;
25      uint8_t current_lock_state = lock->lock;
26      uint8_t took_lock = CAS(&lock->lock, current_lock_state, 1);
27      while ( !took_lock) {
28          current_lock_state = lock->lock;
29          took_lock = CAS(&lock->lock, current_lock_state, 1);
30          if (!took_lock && attempts > 100000) {
31              // we might need to wait longer for the lock
32              // so we yield the core instead of busy waiting
33              sleep(backoff_time);
34              // We might want to introduce a maximum for the backoff
35              backoff_time *=2;
36          }
37          attempts++;
38      }
39  }
40
41  void release(lock *lock)
42  {
43      uint8_t rv = CAS(&lock->lock, 1, 0);
44      assert(rv);
45  }
```

Listing 7: Exponential back-off lock