Tecnológico de Costa Rica 31 de agosto de 2017

Escuela de Ingeniería en Computación Samantha Arburola León
Principios de Sistemas Operativos 2013101697

Operating System Concepts

Chapter 5 - Practice Exercises

5.15 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
      int available;
} lock;
```

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the test and set() and compare and swap() instructions:

- **void acquire(lock *mutex)**
- **void release(lock *mutex)**

```
typedef struct{
      int available;
}lock;

void init(lock *mutex){
// available==0 indicates lock is available
// available==1 indicates lock  unavailable
      mutex->available=0;
}

int testTarget(int *target){
      int temp = *target;
      *target = true;
      return temp;
}

void acquire(lock *mutex){
      while(testTarget(&mutex->available,1)==1);
}

void release(lock *mutex){
      mutex->available=0;
}
```

Compare and Swap:

```
int compareSwap(int *punter,int expected,int new){
	int actual = *punter;
	if(actual == expected)
		*ptr = new;
	return actual;
}

void acquire(lock *mutex){
	while(compareSwap(&mutex->available,0,1)==1);

}

void release(lock *mutex){
	mutex->available=0;
}
```

5.17 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- **The lock is to be held for a short duration.**
- **The lock is to be held for a long duration.**
- **A thread may be put to sleep while holding the lock.**

Lock is to be held for a short duration: use a spinlock as it may in fact be faster than using a mutex lock which requires suspending the waiting process.

Lock is to be held for a long duration: a mutex lock is preferable because allows the other processing core to schedule another process while the locked process waits.

Thread may be put to sleep while holding the lock: you wouldn't want the waiting process to be spinning while waiting for the other process to wake up, so a mutex lock goes here

5.19 A multithreaded web server wishes to keep track of the number of requests it services (known as hits). Consider the two following strategies to prevent a race condition on the variable hits. The first strategy is to use a basic mutex lock when updating hits:

```
int hits;
mutex_lock_hit_lock;
hit lock.acquire();
hits++; hit lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

5.20 Consider the code example for allocating and releasing processes shown in Figure 5.23

a. **Identify the race condition(s).**
b. **Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).**
c. **Could we replace the integer variable `int number_of_processes = 0` with the atomic `integer atomic_t number_of_processes = 0` to prevent the race condition(s)?**

a. Race condition on the variable number of processes.

b. acquire() must be placed upon entering each function and release() immediately before exiting each function.

c. No,because the race occurs in the allocate process() function where number of processes is first tested in the if statement

5.37 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the decrease_count() function:

```
/* decrease available resources by count resources */
 /* return 0 if sufficient resources available, */
 /* otherwise return -1 */
int decrease_count(int count) {
        if (available_resources < count)
                return -1;
        else {
                available_resources -= count;
                return 0;
        }
 }
```

When a process wants to return a number of resources, it calls the increase_count() function:

```
 /* increase available resources by count */
int increase_(int count) {
        available_resources += count;
        return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. **Identify the data involved in the race condition.**
b. **Identify the location (or locations) in the code where the race condition occurs.**
c. **Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the decrease count() function so that the calling process is blocked until sufficient resources are available.**

The integer variable available_resources is shared between methods and threads, which allows for a race condition. It must be synchronized among threads, to make sure the threads always have to most recent value of the variable.

Between the reading and writing, the thread may be kicked of the CPU to allow another thread to also attempt to modify the value. Whereafter the variable value in the first thread is different than the variable value in memory.