Operating Systems Module: CO-562
Constructor University Date: 2024-09-27
Dr. Jürgen Schönwälder Due: 2024-10-04

Problem Sheet #4

Problem 4.1: *multi-threaded 100 prisoners problem* (10 points)

The 100 prisoners problem is stated by Philippe Flajolet and Robert Sedgewick as follows:

> The director of a prison offers 100 death row prisoners, who are numbered from 1 to 100, a last chance. A room contains a cupboard with 100 drawers. The director randomly puts one prisoner's number in each closed drawer. The prisoners enter the room, one after another. Each prisoner may open and look into 50 drawers in any order. The drawers are closed again afterwards. If, during this search, every prisoner finds his number in one of the drawers, all prisoners are pardoned. If just one prisoner does not find his number, all prisoners die. Before the first prisoner enters the room, the prisoners may discuss strategy — but may not communicate once the first prisoner enters to look in the drawers. What is the prisoners' best strategy?

A simple strategy is that every prisoner randomly chooses drawers while searching for his number. This strategy is not very effective since every prisoner has a 50% chance to find his number. This means that 100 prisoners have a chance of 0.5^{100} to find their numbers, which is almost zero.

A smart strategy is that every prisoner starts by opening the drawer with his own number. If the number in the drawer matches, he has been successful. If not, he next opens the drawer with the number found in the current drawer, i.e., the prisoner follows a pre-determined sequence of drawers to find his number. This strategy provides a surprisingly high chance for all prisoners to find their numbers.

Your task is to simulate this game. Implement a C program `prisoner` that first initializes 100 drawers and 100 prisoners and then runs different game strategies (at least random and smart). Implement the prisoners as concurrent threads. In order to open a drawer, a thread (prisoner) first has to acquire a lock. Implement for each game strategy two locking strategies:

(i) Using a global lock for all drawers: A prisoner first obtains the global lock and then executes the search for his number. When done, the prisoner releases the global lock. In other words, prisoners execute their searches sequentially.

(ii) Using a separate lock for each drawer: Prisoners inspect drawers concurrently but they have to obtain a lock for a specific drawer before opening it and they release the lock after closing the drawer. In other words, prisoners execute their search concurrently but they coordinate if they want to access the same drawer.

This yields four different methods:

1. Opening random drawers using a global lock (`random_global`)
2. Opening random drawers using locks for each drawer (`random_drawer`)
3. Opening a sequence of drawers using a global lock (`smart_global`)
4. Opening a sequence of drawers using locks for each drawer (`smart_drawer`)

The command line option `-n` determines how many games to simulate. The default is 100 games. The command line option `-s` seeds the random number generator with a non-static value. An example execution might produce the following output:

```
$ ./prisoner
random_global       0/100 wins/games =  0.00%  xxx.xxx/100 ms = x.xxx ms
random_drawer       0/100 wins/games =  0.00%  xxx.xxx/100 ms = x.xxx ms
smart_global       26/100 wins/games = 26.00%  xxx.xxx/100 ms = x.xxx ms
smart_drawer       31/100 wins/games = 31.00%  xxx.xxx/100 ms = x.xxx ms
```

The first column identifies the method, the second column shows the measured chance to win the game, while the third column shows the measured average execution time. Submit the times you measured. What do you observe?

Time measurement can be implemented in a simple (although not very accurate) manner. The `timeit()` function shown below takes as arguments the number of threads and a pointer to a function implementing one of the strategies. The `run_threads()` function creates `n` threads (each one executing `proc`) and joins them again. The calls of the `clock()` function obtain the a timestamp before and a timestamp after the execution of `run_threads()`. The timestamps are then used calculate the execution time (in microseconds).

```c
static double
timeit(int n, void* (*proc)(void *))
{
    clock_t t1, t2;
    t1 = clock();
    run_threads(n, proc);
    t2 = clock();
    return ((double) t2 - (double) t1) / CLOCKS_PER_SEC * 1000;
}
```