Agnes Duru

Dr. Angeliki Zavou

CSC112 Operating Systems

5/9/2020

CSC112 Operating System Assignment 2 -Written Part

```
                                    /* SNIPPET 1 */
1: long sum = 0;
2: void *square(void *arg) {
3:     int x = *(int *)arg;
4:     sum += x * x;
5:     pthread_exit(NULL);
6: }
7: int main(int argc, char * argv[]) {
8:     int NUM_THREADS = atoi(argv[1]);
9:     pthread_t threads[NUM_THREADS];
10:    int *param[NUM_THREADS];
11:    for (long i = 0; i < NUM_THREADS; i++) {
11:         param[i] = malloc(sizeof(int));
12:         *param[i] = i + 1;
13:         pthread_create(&threads[i], NULL, square, (void *)param[i]);
14:    }
15:    printf("Average: %.2lf\n", (double)sum/NUM_THREADS);
16:    pthread_exit(NULL);
17:}
```

---

i) The snippet above includes at least one concurrency bug. For each one of the concurrency bugs you identify provide a brief description of the following: type of bug(s), reason/lines of code that cause the bug(s), critical section(s) (for the data race bugs).

a. This is a data race because the threads are competing for concurrent access to the shared global variable sum. An order bug can be found because the average is getting calculated before all the threads have finished the sum calculations. The critical section is line 3. This section affects the value of the global variable sum which needs to finish its incrementing before the main thread prints the average.

ii) Update the code above to eliminate the concurrency bugs identified in (i) using the minimum number of synchronization primitives and without waiting on ("join-ing") threads. Your solution must ensure that all summations are executed properly and the final output is consistently printing the proper results. For example, the final output for NUM_THREADS = 5, should be à 11.00 The final output for NUM_THREADS = 10, should be à 38.50 - For each one of the concurrency bugs you need to clearly

describe which synchronization primitive you chose and why as well as their initial values (if applicable).

i. For the order bug, I would surround the for loop with binary semaphores where one is placed before the printing of the Average. For the race problem, I would surround the critical area of the global variable sum incrementing in the square function with binary semaphores as well. The initial value of the first semaphore being sem1 would be 1 and the rest would be 0 to ensure that the order is being kept. I decided to use binary semaphores instead of counting semaphores because multiple threads running at a time is not necessary in this case.

[Q2] Consider four concurrently executing threads in the same process as shown below:

i) Show the final possible output-sequences (after all four threads have executed), assuming that each one of the following printf-statements is executed to completion. Assume also that the output in each printf-statement is not buffered and is immediately printed.

Thread A → printf("is "); printf("there! "); | Thread B → printf("almost "); | Thread C →printf("is") | Thread D → printf("there "); printf("Summer ");

a. Being that the output is not buffered and are occurring in any way, there are thus 4! ways that these statements could be printed. Some ways include:
    a. Way 1:
        i. Thread A : "Hang in"
        ii. Thread A: " Here"
        iii. Thread B: "almost"
        iv. Thread C: "is"
        v. Thread D: "there!"
        vi. Thread D: "Summer"
    b. Way 2:
        i. Thread B: "almost"
        ii. Thread A: "Hang in"
        iii. Thread A: "here!"
        iv. Thread C: "is"
        v. Thread D: "there!"
        vi. Thread D: "Summer"
    c. Way 3:
        i. Threads: D;C; B;A
    d. Way 4:
        i. Threads: C;B;A;D
    e. Way 5:

i. Threads:B;D;C;A

b. And there are 19 more ways in which the a;b;c; and d order sequences can be uniquely manipulated.

ii) Using the minimum number of semaphores needed to make sure that the final output will always be the single line: “Hang in there! Summer is almost here!”? - Note: To get full points you need to justify a) the number and type of semaphores chosen b) their initial values and the 3) specific semaphore-operations (P() and V()) used to guarantee the final result.

a. I would do:

```
sem_t sem1;
sem_t sem2;
sem_t sem3;
sem_t sem4;


    sem_init(&sem1, 0, 1);

  sem_init(&sem2, 0, 0);
  sem_init(&sem3, 0, 0);
  sem_init(&sem4, 0, 0);



  sem_wait(&sem1); //acquires semaphore
    printf(“Hang in ”);
   sem_post(&sem2); //releases semaphore

sem_wait(&sem2);
prinft(“there!”);
sem_post(&sem3);

sem_wait(&sem3);
printf(“Summer “);
sem_post(&sem4);

sem_wait(&sem4);
printf(“is”);
sem_post(&sem5);

sem_wait(&sem5);
printf(“almost”);
sem_post(&sem6);

sem_wait(&sem6);
```

printf(“here”);

b. I decided to initialize one semaphore to hold 1 thread and the rest to hold 0 threads at a time so that the others could wait their turn to be executed. I used a binary semaphore with sem_wait as P() and sem_post as V() because I did not think it was necessary to run numerous threads at a time in this case. I used 6 semaphores because I thought that by pairing almost each print statement with a pair of semaphores I would be able to better control the order.

iii) iii) Update your code from (ii), so that the final output will always be the following two lines: “Hang in there! Summer is almost here!” “Hang in there! Summer is almost here!”

```
****I am initializing  the global  variable i
Int i =0;

While (i != 2) {
sem_wait(&sem1); //acquires semaphore
  printf(“Hang in ”);
    sem_post(&sem2); //releases semaphore

sem_wait(&sem2);
prinft(“there!”);
sem_post(&sem3);

sem_wait(&sem3);
printf(“Summer “);
sem_post(&sem4);

sem_wait(&sem4);
printf(“is”);
sem_post(&sem5);

sem_wait(&sem5);
printf(“almost”);
sem_post(&sem6);

sem_wait(&sem6);
printf(“here”);
*****Added this
sem_post(&sem7);
```

```
sem_wait(&sem7);
 i = i+1;
}
```

a. I instantiated an integer i. This integer is used in a condition to make the program print the saying twice. To ensure that i gets incremented last I added a sem_wait semaphore so that it could execute after Thread A's, "here".