Assignment 3 Report - Use of Semaphores

Dean Tsankov, Kacper Lisik

February 16, 2025

Introduction

In this assignment, we learn how to use semaphores to synchronize threads in a concurrent program with shared variables as well as to use semaphores to control access to shared resources.

Problem 2 - The Bear and Honeybees Problem (multiple producers - one consumer)

Task description

```
Given are n honeybees and a hungry bear. They share a pot of honey. The
↪  pot is initially empty; its capacity is H portions of honey. The bear
↪  sleeps until the pot is full, eats all the honey, and goes back to
↪  sleep. Each bee repeatedly gathers one portion of honey and puts it
↪  in the pot; the bee who fills the pot awakens the bear.
Develop and implement a multithreaded program to simulate the actions of
↪  the bear and honeybees. Represent the bear and honeybees as
↪  concurrent threads (i.e., a "bear" thread and an array of "honeybee"
↪  threads) and the pot as a critical shared resource that can be
↪  accessed by at most one thread at a time. Use only semaphores for
↪  synchronization. Your program should print a trace of interesting
↪  simulation events. Is the solution fair (w.r.t. honeybees)?
```

Key points of solution

In this problem we make use of two binary semaphores. One to keep track of bees contributing to the pot and another for when the pot is full and the bear should wake up. Initially the ”bees can work” semaphore is set to one while the ”pot is full” is set to zero.

In the event when the bear is woken up the bees should stop filling the pot and as such at that point the bear thread is the one responsible for

letting the bees know that they should resume when the pot is emptied. Therefore the `Bees()` thread has the form:

```
(...)
void *Bees(void *arg)
{
    while (!end)
    {
        // add to jar enter critical section
        sem_wait(&beeWork);
        int curr = +++(((struct Arguments *)arg)->jarAmount);
        printf("bee id %d added to the jar, current: %d\n",
            (int)((struct Arguments *)arg)->id, curr);

        //when full do not continue beework, but let bear deal with it
        if (curr >= jarCapacity)
        {
            printf("bee id %d filled the jar\n",
                (int)((struct Arguments *)arg)->id);
            sem_post(&jarFull);
        }else{
            sem_post(&beeWork);
        }
        //simulate work being done and also help with thread fairness
        usleep(1000);
    }
    pthread_exit(NULL);
}
(...)
```

The sleep function call, while not crucial proves effective as not only to simulate work but to help all not yet initialized threads have a chance at joining in the work.

The `Bear()` thread is also fairly simple:

```
(...)
void *Bear(void *arg)
{
    while (!end)
    {
        sem_wait(&jarFull);
        *(((int *)arg)) = 0;
        printf("the bear ate the jar\n");
        itterations = itterations - 1;
```

```
            if (itterations <= 0)
            {
                end = 1;
            }
            //signal that bear is finished and the bees can resume
            sem_post(&beeWork);
        }
        pthread_exit(NULL);
    }
    (...)
```

Finally the main function has the rudimentary task to deploy the number of bee threads according to a passed parameter and the single bear thread, and later wait for their execution in view of a `join` routine.

Problem 4 -The Unisex Bathroom Problem (Similar to the Readers/Writers problem)

Task description

```
Suppose there is only one bathroom in your department. Any number of men
↪ or women can use it, but not simultaneously. Develop and implement a
↪ multithreaded program that provides a fair solution to this problem
↪ using only semaphores for synchronization. Represent men and women as
↪ threads that repeatedly work (sleep for a random amount of time) and
↪ use the bathroom. Your program should allow any number of men or
↪ women to be in the bathroom simultaneously. Your solution should
↪ ensure the required exclusion, avoid deadlock, and ensure fairness,
↪ i.e., ensure that any person (man or woman) waiting to enter the
↪ bathroom eventually gets to do so. Have the persons sleep for a
↪ random amount of time between visits to the bathroom and have them
↪ sleep for a smaller random amount of time to simulate the time it
↪ takes to be in the bathroom. Your program should print a trace of
↪ interesting simulation events.
```

Key points of solution

1. Arriving at the Bathroom

- A thread (representing a person) first waits in a FIFO queue using the `fifo_queue` semaphore. This ensures that requests are handled in the order they arrive.
- Then, it locks the shared state using `state_mutex` so that only one thread can change the shared variables at a time.

- If the bathroom is empty (`people_inside == 0`), the thread sets the current bathroom gender (`gender_inside`) to its own gender.

2. Entering the Bathroom

- If the current bathroom gender matches the thread's gender and the bathroom is not closed (i.e., not exceeding the maximum consecutive visits), the thread:
  - Increases the count of people inside (`people_inside`).
  - Increments the number of consecutive visits.
  - Prints a message indicating that it is entering.
  - Checks if the maximum consecutive visits is exceeded. If so, and if there is someone waiting from the opposite gender, it marks the bathroom as closed.
- Otherwise (if the bathroom is occupied by the opposite gender or is closed):
  - The thread increases its gender's waiting count.
  - It then releases the locks and waits on its specific queue (`g_queue[gender]`).
  - Once it is signaled, the thread re-acquires the lock, decreases the waiting count, increases the count of people inside, and prints a message indicating that it enters after waiting.

3. Using the Bathroom

- The thread simulates doing its business (printing a message such as "takes a pee" or "takes a poo").
- It then sleeps for a short random period to mimic the time spent in the bathroom.

4. Exiting the Bathroom

- When finished, the thread locks the shared state again and decreases `people_inside`.
- It prints an exit message.
- If the bathroom becomes empty (i.e., `people_inside` is 0), the code checks:
  - If there are waiting threads from the opposite gender, the bathroom's gender is switched, the consecutive visit counter is reset, and a batch of waiting threads from the opposite gender is signaled.
  - If no one from the opposite gender is waiting but there are threads of the same gender waiting, then a batch of these same-gender threads is signaled.
  - If no threads are waiting, the counters are simply reset.

5. Repeating the Cycle

- After exiting the bathroom, the thread goes back to "work" (sleeps for a bit) and will eventually try to use the bathroom again.

```
    (...)
    // 0-woman  1-man
void bathoroomer(bool gender)
{
    sem_wait(&fifo_queue);
    sem_wait(&state_mutex);
    if(people_inside == 0)  gender_inside = gender;

    if(gender_inside == gender && !bathroom_closed)
    {
        //entering wc

        people_inside++;
        consecutive_visits++;
        sem_wait(&cout_sem);
        cout<<"Thread: "<< omp_get_thread_num()<<" gender: "<<(gender?
        ("man"):("woman")) <<" entering the bathroom; "<<(gender_inside?
        ("man wc"):("woman wc"))<<"; people inside: "
        <<people_inside<<"\n";
        sem_post(&cout_sem);
        if(consecutive_visits > MAX_CONSECUTIVE_VISITS && waiting[!gender] > 0){

            sem_wait(&cout_sem);
            cout<<"bathroom closed Thread: "<<omp_get_thread_num()<<"\n";
            sem_post(&cout_sem);
            bathroom_closed = 1;
        }
        sem_post(&state_mutex);
        sem_post(&fifo_queue);
    }
    else
    {
        waiting[gender]++;
        sem_post(&state_mutex);
        sem_post(&fifo_queue);

        sem_wait(&g_queue[gender]);
        sem_wait(&state_mutex);
        waiting[gender]--;
        people_inside++;
```

```
        sem_wait(&cout_sem);
        cout<<"Thread: "<< omp_get_thread_num()<<" gender: "<<(gender?
        ("man"):("woman")) <<" entering the bathroom after waiting in
        the queue; "<<(gender_inside?("man wc"):("woman wc"))<<";
        people inside: "<<people_inside<<"\n";
        sem_post(&cout_sem);
        consecutive_visits++;
        if(consecutive_visits > MAX_CONSECUTIVE_VISITS &&
        waiting[!gender] > 0){

            sem_wait(&cout_sem);
            cout<<"bathroom closed Thread: "<<omp_get_thread_num()
            <<"\n";
            sem_post(&cout_sem);
            bathroom_closed = 1;
        }

        sem_post(&state_mutex);
    }

    //doing the bussines
    sem_wait(&cout_sem);
    cout<<"Thread: "<< omp_get_thread_num()<<" takes a "<<((rand()%10
    <2) ? ("poo"):("pee"))<<" \n";
    sem_post(&cout_sem);
    //
    usleep(rand()%1500);
    //
    //exiting the wc
    sem_wait(&state_mutex);
    people_inside--;
    sem_wait(&cout_sem);
    cout<<"Thread: "<< omp_get_thread_num()<<" gender: "<<(gender?
    ("man"):("woman")) <<" exiting the bathroom; "<<"people inside: "
    <<people_inside<<"\n";
    sem_post(&cout_sem);
    if(people_inside == 0)
    {
        if(waiting[!gender] > 0)
        {
            gender_inside = !gender;
            sem_wait(&cout_sem);
            cout<<"Thread: "<< omp_get_thread_num()<<" changing wc
            gender to "<<((!gender)? "man":"woman")<<"\n";
            sem_post(&cout_sem);
```

```
            consecutive_visits = 0;
            bathroom_closed = 0;
            for (int i = 0; i <
            min(MAX_CONSECUTIVE_VISITS,waiting[!gender]); i++)
            {
                sem_post(&g_queue[!gender]);
            }
        }
        else if(waiting[gender] > 0)
        {
            consecutive_visits = 0;
            bathroom_closed = 0;
            for (int i = 0; i <
            min(MAX_CONSECUTIVE_VISITS,waiting[gender]); i++)
            {
                sem_post(&g_queue[gender]);
            }
        }
        else
        {
            consecutive_visits = 0;
            bathroom_closed = 0;
        }
    }
    sem_post(&state_mutex);
}
    (...)
```