Operating System
CS 370
Spring 2019

Project 6
Building n-Butyllithium

Due Date: April 15th 2019
Must be submitted before class time(1:00pm)

Total Points: 100

Graduate Assistant: Pradip Maharjan
Office Hours: MoWe 12 PM- 2 PM
Teaching Assistant: Luis Maya Aranda
Office Hours: TuTh 10 AM – 12 PM

Course Web Page: http://osserver.cs.unlv.edu/moodle/

Objective: Become familiar with multi-threaded programming and synchronizing access to shared data with semaphores. Become familiar with the differences between mutexes and semaphores. Implement a semaphore in C.

Introduction:

Semaphores were invented by pioneering computer scientist Edsger Dijkstra in the early 1960s for an operating system, the Electrologica X8, he and his team were working on.* Semaphores are important synchronization variables that can be used to solve mutual exclusion as well as more general synchronization problems such as turn-taking, the Readers-Writers problem, and the Producer-Consumer problem. There are many classic problems that have been solved with semaphores, with many of the problems attributed to Dijkstra himself, such as the Dining Philosophers problem. In this exercise, you will use semaphores to build n-Butyllithium molecules from hydrogen producers, carbon producers, and lithium producers (threads).

Submission

Your program should compile using following command:

gcc -std=c11 chemistry.c -o chemistry -pthread -lm

or

gcc-5 -std=c11 chemistry.c -o chemistry -pthread -lm

- Submit chemistry.c (not c++, just one file, not zipped) via the class web page (assignment submission link) by class time (1:00 PM).

Submissions received after the due date/time will not be accepted.

* *https://en.wikipedia.org/wiki/Semaphore_(programming)*

Assignment

In this assignment, you will implement a semaphore. The pseudo code for the semaphore will be given. You will implement a driver program that will use the pthread library to create threads that produce hydrogen, threads that produce carbon, and threads that produce lithium. The three types of threads will synchronize to create n-Butyllithium. The test program will take the following input:

./chemistry [bondCount]

Where the **bondCount** is an unsigned integer that corresponds to the **number of molecules** your program will generate before exiting.

A Brief Review of n-Butyllithium Chemistry

You may recall from a chemistry course that acids donate protons (H^+) to a solution while bases accept protons. N-butyllithium, chemical formula C_4H_9Li, is a superbase used in the synthesis of organic compounds such as in the pharmaceutical industry. It is an unstable, colorless liquid at room temperature with a melting point of -105 F or 197 K. It is a superbase, meaning that it is a very strong base, with a pK_a of about 50. For reference, a common strong base is sodium hydroxide (NaOH) with a pK_a of about 14. One molecule of n-butyllithium is composed of 4 Carbons, 9 Hydrogens, and 1 Lithium atom. However, it often forms hexamers with the structure shown in the following figure.

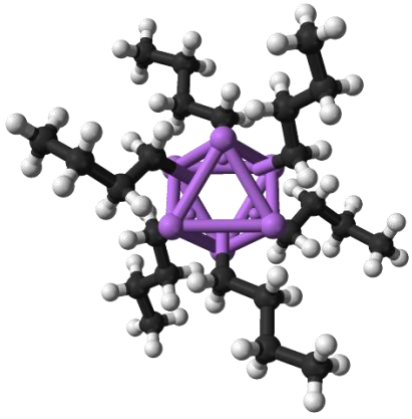

Figure 1: A hexamer of butyllithium $(C_4H_9Li)_6$

N-butyllithium is such a strong base that the amount of heat energy produced upon proton transfer is enough to cause spontaneous ignition and combustion in the presence of air. You can see videos of this effect on YouTube. For more information about n-butyllithium, see the Wikipedia page.*

*https://en.wikipedia.org/wiki/N-Butyllithium

An Overview of Semaphores

A semaphore is a synchronization variable with the following structure:

```
struct Semaphore {
  int value;
  int wakeups;
  Mutex *mutex;
  Cond *cond;
};
```

Semaphores are similar to mutexes in that they can both be used to solve mutual exclusion. You should already be familiar with mutexes from the previous assignment. Mutexes provide two basic functionalities: **lock** and **unlock**. The semaphore analogues are **wait** and **signal**. A thread can call **lock** on a mutex and will obtain ownership (acquire) the lock after some or no waiting depending on if the lock has already been acquired by another thread. Once a thread has completed the **lock** call, then that thread **owns** the lock and any other thread's call to **lock** on the same object must wait until the **owner** completes the **unlock** call. It is important to recognize the concept of ownership in a mutex lock. In semaphores, the concept of ownership does not apply. Any thread may **unlock** (**signal**) a semaphore even if that semaphore was locked by a different thread. This is a key difference that allows semaphores to have much more functionality than a mutex.

A semaphore is always initialized with an integer **value**. To distinguish semaphores from mutexes, "locking" a semaphore is called **wait()** or **P()** and "unlocking" a semaphore is called **post(), signal(), V()**. Note, **P()** does **not** stand for **post()**, it stands for **Proberen** (Dutch for "to test"). **V()** stands for **Verhogen** (Dutch for "increase"). Intuitively, semaphore is similar to a mutex with a **value** field that determines how many threads can concurrently use the critical section. However, semaphores also have much more functionality than this so it is limiting to view a semaphore only in this way. When a thread calls **wait()** on a semaphore object, it decrements the value field of the semaphore and will complete the **wait()** call if **value > 0** and if **value <= 0** the thread will wait on a condition variable until another thread calls **signal()**. When a thread calls **signal()** it will increment the semaphore's value and if **value > 0**, it will complete the call, else **value <= 0** and the thread will wake up a waiting thread before completing the **signal()** call. Often, semaphores are used similarly to mutexes in the following way:

```
ExampleFunction(){
  semaphore.wait();
  //Critical Section
  //. . .
  semaphore.signal();
}
```

In this way, the synchronized section will allow at most **value** threads into the critical section concurrently. If value is initialized to one, then the semaphore will behave exactly like a mutex in this scenario. However, if the value is initialized to, for example, five, then at most five threads can be in the critical section simultaneously. If the value is initialized to zero, then no threads can enter the critical section (deadlock).

A semaphore variable can be used as a sensor or signal to let another thread know that something has been completed or that a condition has been met. For this, we take advantage of the fact that any thread can call **signal()**, not just the thread that called **wait()**. Let sem1 be a semaphore with its value initialized to 0. Then consider the following code blocks:

Thread A

```
First(){
  //Thread A completes something
  //then signals thread B
  sem1.signal ();
  // Thread A can do something else…
}
```

Thread B

```
Second(){
  //Thread B is stuck until thread A completes
  //its task…
  sem1.wait();
  //Thread B can now continue…
}
```

Semaphores can also be used for turn-taking. We can think of this as an extension of the "signal" algorithm above. If we initialize the semaphore, **sem1**, with **value=0**, and semaphore **sem2** with **value=1** then consider the following code blocks:

Thread A

```
First(){
  while(true){
    sem2.wait();
    //Thread A's turn to do something
    //…
    sem1.signal ();
  }
}
```

Thread B

```
Second(){
  while(true){
    sem1.wait();
    //Thread B's turn to do something
    //. . .
    sem2.signal();
  }
}
```

In this example, thread A will run first since sem2 is initialized to 1, thread A's sem2.wait() call completes. Thread B's call to sem1.wait() will block thread B since sem1 is initialized to 0. It is only once thread A signals the sem1 mutex that Thread B will be able to complete its sem1.wait() call. After thread A signals sem1, it will call sem2.wait() again but the value of sem2 will remain as 0 until thread B finishes its turn and signals sem2. Convince yourself that this turn taking will work without deadlock forever.

Creating a Mutex

To implement a semaphore you will need to construct a mutex. A mutex is a tool to provide mutual exclusion. At any point in time, only one thread can hold a given mutex. Use the following C code algorithms to implement your mutex.

Algorithm 1 Make Mutex

```
Mutex *make_mutex()
{
    Mutex *mutex = malloc(sizeof(Mutex));
    int n = pthread_mutex_init(mutex, NULL);
    if(n != 0) {
        perror("make_mutex failed");
        exit(1);
    }
    return mutex;
}
```

Algorithm 2 Mutex Lock

```
void mutex_lock(Mutex *mutex)
{
    int n = pthread_mutex_lock(mutex);
    if(n != 0) {
        perror("mutex_lock failed");
        exit(1);
    }
}
```

Algorithm 3 Mutex Unlock

```
void mutex_unlock(Mutex *mutex)
{
    int n = pthread_mutex_unlock(mutex);
    if(n != 0) {
        perror("mutex_unlock failed");
        exit(1);
    }
}
```

Creating a Condition Variable

To implement a semaphore you will need to construct a condition variable. A condition variable is used to have a thread wait for a condition to be met. A condition variable is always used with a mutex that synchronizes access to the variable we would like to test as the condition. Conditions are checked in the following way:

1. Acquire the mutex to check the shared variable.
2. Check the shared variable to determine if the condition is met.
    a. If the condition is met, then complete the wait call.
    b. If the condition is not met, then unlock the mutex and sleep until signaled.

Once another thread calls signal, one waiting thread is woken up at which point it must repeat the above steps. You will create wrapper functions for the pthread condition variable data type, pthread_cond_t. You will need to typedef pthread_cond_t as Cond. Then use the following C code algorithms to implement your condition variables.

Algorithm 4 Make cond

```
Cond *make_cond()
{
    Cond *cond = malloc(sizeof(Cond));
    int n = pthread_cond_init(cond, NULL);
    if(n != 0) {
        perror("make_cond failed");
        exit(1);
    }
    return cond;
}
```

Algorithm 5 Cond Wait

```
void cond_wait(Cond *cond, Mutex *mutex)
{
    int n = pthread_cond_wait(cond, mutex);
    if(n != 0) {
        perror("cond_wait failed");
        exit(1);
    }
}
```

Algorithm 6 Cond Signal

```
void cond_signal(Cond *cond)
{
    int n = pthread_cond_signal(cond);
    if(n != 0) {
        perror("cond_signal failed");
        exit(1);
    }
}
```

Creating a Semaphore

Before synthesizing n-butyllithium molecules, you must first create the tool to synchronize them. You will need to implement a semaphore data type which will consist of a struct to hold the fields of the semaphore and three functions: **make_semaphore**, **sem_wait**, **sem_signal**. You must use your Test-And-Set lock as the mutex for the semaphore. The following algorithms should be used to create the semaphore variable.

Algorithm 7 Semaphore *make_semaphore(int value)

```
Semaphore * semaphore ← malloc(sizeof(Semaphore))
semaphore.value ← value
semaphore.wakeups ← 0
semaphore.mutex ← make_mutex()
semaphore.cond ← make_cond()
return semaphore
```

Algorithm 8 void sem_wait(Semaphore *semaphore)

```
mutex_lock(semaphore.mutex)
semaphore.value ← semaphore.value − 1
if semaphore.value < 0 then
    do
        cond_wait(semaphore.cond, semaphore.mutex)
    while semaphore.wakeups < 1
    semaphore.wakeups ← semaphore.wakeups − 1
end if
mutex_unlock(semaphore.mutex)
```

Algorithm 9 void sem_signal(Semaphore *semaphore)

```
mutex_lock(semaphore.mutex)
semaphore.value ← semaphore.value + 1
if semaphore.value ≤ 0 then
    semaphore.wakeups ← semaphore.wakeups + 1
    cond_signal(semaphore.cond)
end if
mutex_unlock(semaphore.mutex)
```

Creating n-Butyllithium

To create N n-Butyllithium molecules you will need to create 4N carbon threads and 9N hydrogen threads, and N lithium threads. Each thread will become a waiting carbon/hydrogen/lithium and then attempt to make an n-butyllithium molecule if there are at least 9 waiting hydrogens, 4 waiting carbons, and 1 waiting lithium. If not, then the thread will wait in the carbon, hydrogen, or lithium queue until signaled. To make the molecule, the making thread will move 4 waiting carbons to the assigned carbons variable, 9 waiting hydrogens to the assigned hydrogens variable and 1 waiting lithium to the assigned lithium variable. Then the making thread will signal the appropriate number of other hydrogens, carbons, and/or lithium to complete the molecule. The thread making the molecule will always be an atom in the molecule its making. Each thread will only contribute one atom to the reaction before exiting. The following pseudocode shows the algorithm for the lithium threads. You will need to design and implement the pseudocode for the lithium threads. You will need to create the code for the hydrogen and carbon threads that follow very similarly to the lithium example.

Algorithm 10 Lithium Algorithm

```
id ← thread_id
Lithium atom (id) has started
P(mutex)
Update number of waiting lithium atoms
while aL = 0 do
    if wC ≥ 4 and wH ≥ 9 and wL ≥ 1 then
        Make N-Butyllithium
        sleep(2)
        V(carbonQueue) x 4
        V(hydroQueue) x 9
    else
        Lithium atom (id) waiting
        V(mutex)
        P(lithiumQueue)
        P(mutex)
    end if
end while
Update number of assigned lithium atoms
V(mutex)
Lithium atom (id) done
return NULL
```

Testing Your Program

The following is an example of the program's execution creating one molecule. Your program should also **handle input error checking** as appropriate. To print with color you should print the following string before a statement of that color: Red is "\033[0;31m", Blue is "\033[0;34m", and Green is "\033[0;32m". To return to the default color, White, print "\033[0m".

```
→  Project-6 ./buty 1
Hydrogen atom (1) started
Hydrogen atom (1) waiting
Hydrogen atom (2) started
Hydrogen atom (2) waiting
Hydrogen atom (3) started
Hydrogen atom (3) waiting
Hydrogen atom (4) started
Hydrogen atom (4) waiting
Hydrogen atom (5) started
Hydrogen atom (5) waiting
Hydrogen atom (6) started
Hydrogen atom (6) waiting
Hydrogen atom (7) started
Hydrogen atom (7) waiting
Hydrogen atom (8) started
Hydrogen atom (8) waiting
Hydrogen atom (0) started
Hydrogen atom (0) waiting
Carbon atom (3) started
Carbon atom (3) waiting
Carbon atom (2) started
Carbon atom (2) waiting
Carbon atom (1) started
Carbon atom (1) waiting
Carbon atom (0) started
Carbon atom (0) waiting
Lithium atom (0) started
N-Butyllithium (1) made
Carbon atom (2) done
Carbon atom (1) done
Lithium atom (0) done
Carbon atom (0) done
Carbon atom (3) done
Hydrogen atom (2) done
Hydrogen atom (5) done
Hydrogen atom (6) done
Hydrogen atom (4) done
Hydrogen atom (7) done
Hydrogen atom (3) done
Hydrogen atom (8) done
Hydrogen atom (1) done
Hydrogen atom (0) done
→  Project-6
```