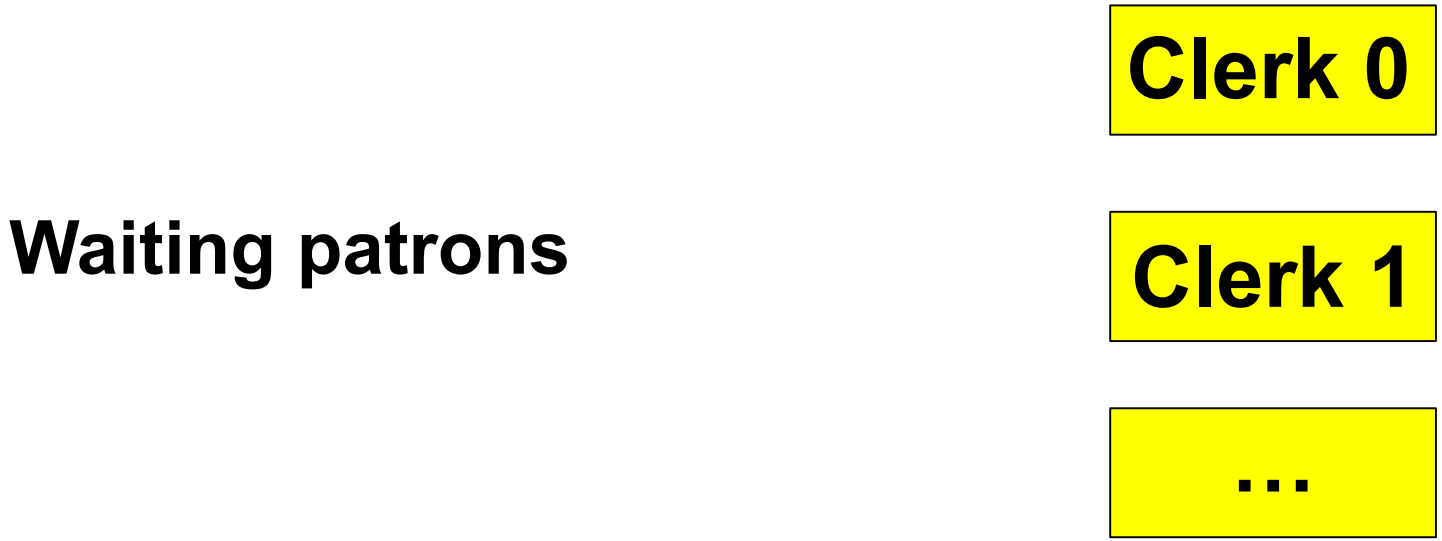
Due
April 29
11:59 PM

THE THIRD SPRING 2024 ASSIGNMENT EXPLAINED

Spring 2024

The problem (I)

- A post office with a single queue and several nameless clerks:



The problem (II)

- Must simulate the behavior of these patrons:
  - **Alice 0 10**
    Bob 3 5
    . . .
- Assuming there is only one clerk:
  - **Alice arrives at the post office at time = 0s.**
    Alice gets service.
    Bob arrives at the post office at time = 3s.
    Alice leaves the post office.
    Bob gets service.
    Bob leaves the post office.

The problem (III)

- With the same input:
  - **Alice 0 10**
    Bob 3 5
    . . .
- And two clerks:
  - **Alice arrives at the post office at time = 0s.**
    Alice gets service.
    Bob arrives at the post office at time = 3s.
    Bob gets service.
    Bob leaves the post office.
    Alice leaves the post office.

A first step

- Can represent the *n* clerks by a single number
  - `nFreeClerks`
- Its initial value will be the total number of clerks
- Its current value will indicate the number of free clerks

Your main program

- Will
  - Get the number of clerks from your program's argument vector
  - Repeatedly
    - Read patron's **`name`**, **`arrivalDelay`**, and **`serviceTime`**
    - Sleep for **`arrivalDelay`** seconds
    - Create a child thread
  - Wait until all threads have terminated
  - Print statistics

Your child threads

- Will
  - Print a message
  - Check if a clerk is available
    - Possibly wait
  - Print a message
  - Sleep for **`serviceTime`** seconds
  - Print a message
  - Signal a clerk has become available

Collecting statistics

- Before terminating, your program should display:
  - The total number of patrons that got serviced
    - Can be maintained by the main program
  - The number of customers that had to wait
    - Will require some extra coding
  - The number of customers that did not have to wait
    - Trivial

Patron threads

- Will
  - Print a message
  - If all clerks are busy
    - Increment the number of customers who had to wait
    - Wait
  - Print a message
  - Sleep for **`serviceTime`** seconds
  - Print a message
  - Signal that a clerk has become available

Using shared variasble

- The number of customers that have to wait and the number of free clerks are:
  - Global variables shared by all the threads
  - Must be declared **`static`**
  - Must be accessed inside a critical section

The final refinement

- Patron threads will
  - Enter a critical section
  - Print a message
  - If all clerks are busy
    - Increment the number of customers who had to wait
    - Wait
  - Print a message
  - Leave the critical section
  - Sleep for **`serviceTime`** seconds
  - Enter a critical section
  - Print a message
  - Signal that a clerk has become available
  - Leave the critical section

Creating Pthreads (I)

- Declare first a child function:
  - ```
    void *child_thread(void *arg) {
            int i;
            // must cast the argument
            i = (int) arg;
            …
            }
    ```
- Thread ends when their functions end
    ```

Creating Pthreads (II)

- Declare a thread ID array
  - `pthread_t tid[MAXTHREADS];`
- Start the thread:
  - `pthread_create(&tid[i], NULL, patron, (void *) pData);`
- Do not lose or overwrite the thread ID
  - You will need it again

Waiting for a specific thread

- Use **`pthread_join()`**
  - **`pthread_join(tid, NULL);`**

Passing arguments to a thread

- **`pthread_create()`** allows a single **`void`** * argument to be passed to the new thread

```
pthread_create(&tid, NULL, patron,(void *) &argList);
```

- You have to pass a string and an integer
  - Store them in a ***structure***
    - **`struct pData argList`**

Warning

- For some unknown reason, all calls to **`pthread_create()`** share the same memory locations for their argument
  - ***Get reused call after call***
  - !!!
- Your thread functions ***must*** save their input arguments into local variables
  - ***Failure to do so will result in very misleading outputs.***

The solution

- Do

```
struct pData *argptr;
char myName[MAXNAME];
int  myServiceTime;
argptr = (struct pData *) arg;
// required
strcpy(myName, argptr->name);
myServiceTime = argptr->serviceTime;
```

A small problem

- The Pthread library ***does not let*** you
  - Wait for an unspecified thread
    - `kidpid = wait(0);`
  - Do the equivalent of:
    - ```
      for (i = 0; i < nChildren; i++) {
          wait(0);
      }
      ```
      ```

A quick fix

- Parent will keep track of the thread id's of all its child threads:
  -
```
pthread_t tid[MAXTHREADS];
…
…
for (i = 0; i < nPatrons; i++)
    pthread_join(tid[i], NULL);
```

Pthread mutexes

- To create a Pthread mutex, use:
  - ```
    static pthread_mutex_t alone;
    // must be declared static
    …
    pthread_mutex_init(&alone, NULL);
    ```
- To request the mutex, use:
  - ```
    pthread_mutex_lock(&alone);
    ```
- To release the mutex, use:
  - ```
    pthread_mutex_unlock(&alone);
    ```
    ```

Pthread condition variables (I)

- The easiest way to create a condition variable is:
  - `pthread_cond_t freeClerks = PTHREAD_COND_INITIALIZER;`

Pthread condition variables (II)

- To wait on a condition:

```
pthread_mutex_lock(&alone);
    while (…)
        pthread_cond_wait(&freeClerks, &alone);
…
pthread_mutex_unlock(&alone);
```

A reminder

- Signals that are ***not caught*** by a waiting thread are ***lost***
- Before setting up a **`pthread_cond_wait(),`**
  - Patron threads ***must*** check that the resource they are waiting for is ***actually unavailable***
- Patron threads releasing a resource must always issue **`pthread_cond_signal()`**

Pthread condition variables (III)

- To signal a condition:

```
pthread_mutex_lock(&alone);
        …
        pthread_cond_signal(&freeclerks);
pthread_mutex_unlock(&alone);
```

- Critical section ***must*** use the same mutex as the one used around the corresponding **`pthread_cond_wait( )`**

Compiling your code

- ***All programs using Pthreads or POSIX semaphores must be compiled with the library flag*** `-lpthread` ***after the list of source code modules as in***
  - `g++ postoffice.cpp -lpthread`

Stating what should be obvious

- You will ***lose points*** if your program
  - Does not get the number of post office clerks from its argument vector.
  - Does not read its other input data from **`stdio`**.
  - Only reads fixed-format inputs
    -
```
Alice      0  10
Bob        3   5
Carolina   4   8
Dean       2   7
```

Notes

- For more details on Pthreads, you might want to look at the LLNL tutorial:
  - **POSIX Threads Programming**

    https://computing.llnl.gov/tutorials/pthreads/