Possible Design HW5

Application starts

↓

Create a structure or a class for representing a matrix given the representation on the file.
For example:

```
struct matrix {
            int row;
            int col;
            bool decimal;
            QVector<QVector<double>>*  decVector;
};
```

↓

Application creates two threads to read the two input files:

*pthread_create(&readThreads[0], &attr, readFile, (void *)argv[1]);*

*pthread_create(&readThreads[1], &attr, readFile, (void *)argv[2]);*

Example: argv[1] is the input file name for the Matrix A

↓

Possible Design HW5

Wait for threads to finish and validate the values in the matrix. Always check the return values of pthread calls!!!

To fetch the matrices you can use either two global variables (no concurrency) or use the pthread_exit function to pass the just populated matrices (for examle: pthread_exit((void*) matrixObj);

Example:

```
int rc;
rc = pthread_join(readThreads[0], (void**)&matrixA);
if (rc) {
        printf("ERROR return code from pthread_join() is %s\n",
        strerror(rc));
        exit(-1);
}
```

Possible Design HW5

Activate as many threads as the size of the final matrix

EXAMPLE:

```
int status;
for(int i=0;i<numThreads;i++) {
        rc = pthread_join(mThreads[i], (void **)&status);
        if (rc) {
                printf("ERROR return code from pthread_join() is %s\n", strerror(rc));
                exit(-1);
        }
}
```

Each thread computes [$C_{i,j}$] and writes to the ”temp.txt” file: $C_{i,j}$=V on a new line

EXAMPLE

$C_{1,2}$=16.

FILE cannot be written concurrently! Mutual exclusion should be implemented to prevent more than one thread to write on the file.

Possible Design HW5

temp.txt will be something like:

$C_{1,2}$=16
$C_{3,1}$=54
$C_{4,1}$=87
$C_{1,3}$=6
…

The main thread joins the execution of all the processing threads and re-format the temp.txt file so that the output file will look like one of the input files. In other words, each line of the file describes each row of the output matrix.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0); // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

-
  ```
  typedef struct{
   int value;
   struct process *list;
   } semaphore;
  ```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
      P0                    P1
wait(S);              wait(Q);
wait(Q);              wait(S);
 ...                    ...
signal(S);             signal(Q);
signal(Q);             signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
       ...
        /* produce an item in next_produced */
       ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
     wait(full);
     wait(mutex);
         . . .
     /* remove an item from buffer to next_consumed */
         . . .
     signal(mutex);
     signal(empty);
         . . .
     /* consume the item in next consumed */
         . . .
 } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
      do {
             wait(rw_mutex);
                ...
             /* writing is performed */
                ...
          signal(rw_mutex);
    } while (true);
```

Readers-Writers Problem (Cont.)

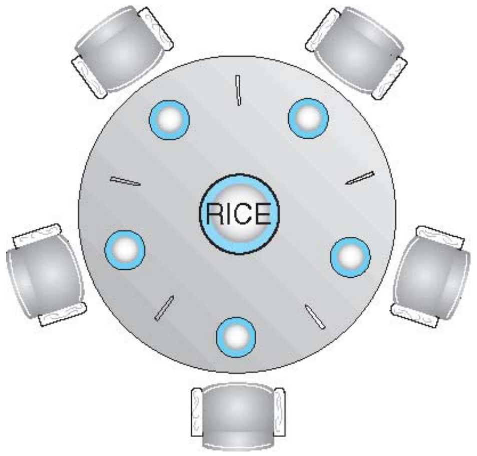
- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
          ...
        /* reading is performed */
          ...
        wait(mutex);
        read_count--;
        if (read_count == 0)
        signal(rw_mutex);
        signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
      do {
            wait (chopstick[i] );
             wait (chopStick[ (i + 1) % 5] );

                         //  eat

             signal (chopstick[i] );
             signal (chopstick[ (i + 1) % 5] );

                           //  think

      } while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.