Announcements

- Assignment 3 online: when should it be due?
  - Monday, March 29.
- Changed earlier today (new clarifications), so re-download.
- Java tips:
  - `catch (Exception ex) {}` is a very bad idea.
  - Close all streams you open.
  - Thread termination and JVM.
  - `instanceof`.
  - Shifting and bit operations.
Chapter 7: Process Synchronization

- Background.
- The Critical-Section Problem.
- Synchronization Hardware.
- Semaphores.
- Classical Problems of Synchronization.
- Monitors.
- Java Monitors.
- Critical Regions.
Concurrent access to shared data may result in data inconsistency.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Shared-memory solution to bounded-butter problem in book allows at most $n - 1$ items in buffer at the same time.

Our solution for $n$ items was correct but needed two counters $pcount$ and $ccount$. Can we use just one counter?
Bounded Buffer

- Shared data:
  ```c
  #define B_SIZE 5
  char buffer[B_SIZE];
  int count=0;
  ```

- Producer:
  ```c
  int in=0;
  while (1) {
      /* produce an item in char nextProduced */
      while (count==B_SIZE)
          /* wait while buffer full */;
      buffer[in]=nextProduced;
      in=(in+1)%B_SIZE; count++; }
  ```

- Consumer:
  ```c
  int out=0;
  while (1) {
      while (count==0)
          /* wait while buffer empty */;
      nextConsumed=buffer[out];
      out=(out+1)%B_SIZE; count--;
      /* consume the item in char nextConsumed */ }
Bounded Buffer (Cont.)

- The statements
  
  \[
  \begin{align*}
  &\text{count++;} \\
  &\text{count--;}
  \end{align*}
  \]

  must be performed *atomically*, meaning that the process cannot be interrupted partway through their execution.

- But they may not be atomic on some CPUs:

  - count++ may be implemented as:

    \[
    \begin{align*}
    &\text{register1= count} \quad \text{LOAD \hspace{1em} R1, @count} \\
    &\text{register1= register1 + 1} \quad \text{INC \hspace{1em} R1} \\
    &\text{count= register1} \quad \text{STORE \hspace{1em} @count, R1}
    \end{align*}
    \]

  - count-- may be implemented as:

    \[
    \begin{align*}
    &\text{register2= count} \quad \text{LOAD \hspace{1em} R2, @count} \\
    &\text{register2= register2 - 1} \quad \text{DEC \hspace{1em} R2} \\
    &\text{count= register2} \quad \text{STORE \hspace{1em} @count, R2}
    \end{align*}
    \]
Bounded Buffer (Cont.)

- If both the producer and consumer attempt to update the buffer concurrently, the CPU instructions may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.
- Assume count is initially 5. One interleaving of statements is:

  producer: register1=count (register1 =5)
  producer: register1=register1+1 (register1 =6)
  consumer: register2=count (register2 =5)
  consumer: register2=register2-1 (register2 =4)
  producer: count=register1 (count =6)
  consumer: count=register2 (count =4)

- The value of count may be 4 (or 6 if you swap the last two lines), where the correct result should be 5.
Race Condition

- *Race condition*: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be synchronized.

- Extremely hard to write correct multi-threaded code:
  - Must think of all possible combinations in execution paths.
  - Or come up with formal proof of correctness (see textbook).
  - Or run a lot of tests *hoping* a bug will show up (nondeterminism).
  - Only *experience* helps.
  - Hence such a skill is highly valued by employers.
Java volatile

- Shared variables *not* always kept in sync:

```java
class Test {
    static int i=0, j=0;
    static void incr() { i++; j++; }
    static void print() { System.out.println("i:"+i+" j:"+j); }
}
```

- Thread T0 calls `incr()` repeatedly.
- Thread T1 calls `print()` repeatedly.
- Assume T0, T1 scheduled as if `incr()`, `print()` had been mutually exclusive.
- Output may read `i:1 j:2` because threads can keep local copies of shared variables and sync up the master copies at will (almost...). So `j` can be updated before `i` by T0.

- volatile declaration: update shared variables in the same order the local copies were updated:

```
static volatile int i=0, j=0;
```

- Details: Java Language Specification 8.3.1.4, 17.
The Critical-Section Problem

- Two or more processes all competing to use some shared data.

- Each process has a code segment, called *critical section*, in which the shared data is accessed.

- Problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
**Solution Requirements**

1. **Mutual Exclusion**: if process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.
   
   *Only one dog can eat the bone at a time.*

2. **Progress**: if no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
   
   *If no dog is eating and many are hungry, a hungry one should get the bone.*

3. **Bounded Waiting/No Starvation**: a bound must exist on the number of times that other processes may enter their critical sections after a process has made a request to enter its critical section and before that request is granted:
   
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the processes.

   *If there are many hungry dogs, they all get to eat eventually.*
Initial Attempts to Solve Problem

- Only 2 processes, $P_0$ and $P_1$.
- Processes may share some common variables to synchronize their actions.
- General structure of every process $P_i$:

```c
while (1) {
    /* entry section */
    /* critical section */
    /* exit section */
    /* remainder section */
}
```

- We’ll refer to the other process as $P_j$ ($j=1-i$).
Algorithm 1

- **Shared variables:**
  
  ```
  int turn=0;
  turn==i => P_i can enter its critical section.
  ```

- **Process P_i:**

  ```
  while (1) {
    while (turn!=i) /* do nothing */;
    /* critical section */
    turn=1-i;
    /* remainder section */
  }
  ```

- Satisfies mutual exclusion, but not progress: if P_o is fast and wants to go again, it cannot even if P_i is just spending a long time in its remainder section.
Algorithm 2

- **Shared variables:**
  
  - boolean flag[2]; flag[0]=flag[1]=false;
  - flag[i]==true \( \Rightarrow \) \( P_i \) wants to enter its critical section.

- **Process \( P_i \):**

  ```
  while (1) {
    flag[i]=true;
    while (flag[1-i]);
    /* critical section */
    flag[i]=false;
    /* remainder section */
  }
  ```

- **Satisfies mutual exclusion, but not progress;** if both execute
  flag[i]=true before either starts waiting in their while loops, then neither ever moves forward! Processes are too polite: “No, you go first.”
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$:

```c
while (1) {
    flag[i]=true;
    turn=1-i;
    while (flag[1-i] && turn==1-i);
    /* critical section */
    flag[i]=false;
    /* remainder section */
}
```

- Meets all three requirements; solves the critical-section problem for two processes.
Bakery Algorithm

- Critical section for n processes.
- Shared variables:
  - boolean c[n]; /* All false. */
  - int t[n]; /* All 0s. */
- Process $P_i$:
  
  while (1) {
    c[i]=true;
    t[i]=1+max(t[0],...,t[n-1]);
    c[i]=false;
    for (j=0; j<n; j++) {
      while (c[j]);
      while (t[j]!=0 && (t[j],j)<(t[i],i));
    } /* critical section */
    t[i]=0;
  } /* remainder section */

Choosing a ticket number.
Done choosing.
Wait for $i$ to choose.

Obtain ticket number; will get no smaller than another process but might get the same.

$j$ wants to enter (or already is inside) its critical section, and so it has nonzero ticket.

$t[j]<t[i] \text{ || } (t[j]==t[i] \text{ && } j<i)$

$i$ waits for $j$ to go through if $j$ has smaller ticket number; or if same ticket, and $j < i$. 

Obtain ticket number; will get no smaller than another process but might get the same.
Bakery Algorithm (Cont.)

- Why doesn’t this work? (Eliminated $c[]$).

```plaintext
while (1) {
    t[i]=1+max(t[0],...,t[n-1]);
    for (j=0; j<n; j++) {
        while (t[j]!=0 && (t[j],j)<(t[i],i));
    } /* critical section */
    t[i]=0;
    /* remainder section */
}
```

- Answer:

$P_0$ reads $t[0], t[1]$.
$P_1$ reads $t[0], t[1]$.
$P_1$ writes $t[1]=1$.
$P_1$ enters critical section because $t[1]=1$ is false (first for-loop iteration), and $(t[1],1)<(t[1],1)$ is false (second for-loop iteration).
$P_0$ writes $t[0]=1$.
$P_0$ enters critical section because $(t[0],0)<(t[0],0)$ is false (first for-loop iteration), and $(t[1],1)<(t[0],0)$ is false (second for-loop iteration).
Bakery Algorithm (Cont.)

- Why doesn’t this work? (Replaced `while` with `if` and `while`.)

```c
while (1) {
    c[i]=true;
    t[i]=1+max(t[0],...,t[n-1]);
    c[i]=false;
    for (j=0;j<n;j++) {
        while (c[j]);
        if (t[j]!=0 && (t[j],j)<(t[i],i))
            while(t[j]!=0); }
    /* critical section */
    t[i]=0;
    /* remainder section */ }
```

- Intent: $P_1$ realizes that it has a lower ticket than $P_0$ ($if$) and waits while $P_0$ remains in its critical section.

- Answer: $P_0$ can finish, go back, and get a higher ticket number. Now $P_0$ is stuck and so is $P_1$ who still thinks that $P_0$‘s nonzero ticket number implies $P_0$ is in its critical section.
Synchronization Hardware

- Test and modify the content of a word *atomically* (i.e. without CPU *ever* preemption process in the middle).

- How? Disable interrupts or use special hardware.

```cpp
boolean testAndSet(boolean *target) {
    boolean result=*target;
    *target=true;
    return result; }
```

- Shared data:

```cpp
boolean lock=false;
```

- Process $P_i$:

```cpp
while (1) {
    while (testAndSet(&lock));
    /* critical section */
    lock=false;
    /* remainder section */ }
```

Bounded wait not satisfied!

If lock is true then nobody is in the critical section and this process can go ahead. *Atomically set* lock to prevent two processes from seeing false at the same time.
Atomically swap two variables.

```c
void swap(boolean *a, boolean *b) {
    boolean temp=*a; *a=*b; *b=temp;
}
```

Shared data:

```c
boolean lock=false;
```

Process $P_i$:

```c
while (1) {
    boolean key=true;
    while (key)
        swap(&lock,&key);
    /* critical section */
    lock=false;
    /* remainder section */
}
```

**Bounded wait not satisfied!**

*key acts like return value of testAndSet() as well as the true constant in its body.*
Synchronization Hardware (Cont.)

- **Starvation fix: shared data:**
  
  ```
  boolean waiting[n]; /* All false. */
  boolean lock=false;
  ```

- **Process** $P_i$:
  
  ```
  while (1) {
    waiting[i]=true;
    boolean key=true;
    while (waiting[i] && key) key=testAndSet(&lock);
    waiting[i]=false;
    /* critical section */
    j=(i+1)%n;
    while ((j!=i) && !waiting[j])) j=(j+1)%n;
    if (j=!i) waiting[j]=false;
    else lock=false;
    /* remainder section */ }
  ```

  Find first waiting process after $P_i$ in circular order (if any).

  Wait until either lock is available or some other process who held the lock passes it on to $P_i$ by setting its waiting[i] to false.

  If one is found, unblock it and give it the responsibility of unlocking (give it the key).

  Otherwise, nobody has been waiting, so release the lock for whoever tries to enter next.
Semaphores

- **Semaphore S**: an integer variable that can only be accessed via two partially indivisible operations. Shown as Java pseudo-class:

```java
class Semaphore {
    int mS = 1;
    
    wait() {
        while (mS <= 0);
        mS--; }
    
    signal() {
        mS++; }
}
```

- **Shared data**: Semaphore S;

- **Process P_i**: 
  ```java
  while (1) {
      S.wait();
      /* critical section */
      S.signal();
      /* remainder section */  }
  ```

Other values useful too.

Once one process is done waiting, it alone can proceed to finish.

Can't modify at the same time.

Busy-waiting (a.k.a. spin-lock).
Useful for short waits in multiple CPU systems (no context-switch).
Wasteful in single CPU systems.

Bounded wait not satisfied!
Semaphore Implementation

- Assume two OS system calls:
  - `block()` suspends the thread that invokes it.
  - `wakeup(T)` resumes the execution of a blocked thread `T`.
- Definition *without busy-waiting*:

```java
class Semaphore {
  int mS=1;
  LinkedList mL=new LinkedList();
  synchronized wait() {
    mS--;
    if (mS<0) {
      mL.addLast(Thread.currentThread());
      block();
    }
  }
  synchronized signal() {
    mS++;
    if (mS<=0)
      wakeup(mL.removeFirst());
  }
}
```

Calling thread goes off the CPU and into waiting (not ready) queue. No busy-waiting. Assume `block()` releases Java monitor.

Queue of blocked processes. Usually FIFO to prevent starvation. But can be priority queue too.
Semaphore as a General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$.
- Use semaphore `flag` initialized to 0.
- Code:

  \[
  \begin{align*}
  & P_i \\ & : \\ & A \\
  & \text{signal}(\text{flag}) \\
  & \text{wait}(\text{flag}) \\
  \end{align*}
  \quad
  \begin{align*}
  & P_j \\ & : \\ & \text{wait}(\text{flag}) \\
  & \text{signal}(\text{flag}) \\
  & B \\
  \end{align*}
  \]
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.

\[
\begin{align*}
P_0 & \quad P_1 \\
\text{wait}(S) & \quad \text{wait}(Q) \\
\text{wait}(Q) & \quad \text{wait}(S) \\
\vdots & \quad \vdots \\
\text{signal}(S) & \quad \text{signal}(Q) \\
\text{signal}(Q) & \quad \text{signal}(S)
\end{align*}
\]

- **Starvation**: indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Two Types of Semaphores

- *Counting* semaphore: integer value can range over an unrestricted domain.

- *Binary* semaphore: integer value can be at most 1 (only 0 or 1 in first, busy-waiting definition); can be simpler to implement on some CPUs (hardware constraint).

- Can implement a counting semaphore using binary semaphores.
Implementing a Counting Semaphore

- **Data structures:**
  
  binary-semaphore $S_1=1$, $S_2=0$;
  
  int $C$ = /* Initial value of counting semaphore. */;

- **Operations:**

  wait() {
    wait($S_1$);
    $C$--;
    if ($C<0$) {
      signal($S_1$);
      wait($S_2$); }
  signal($S_1$); }

  signal() {
    wait($S_1$);
    $C$++;
    if ($C<=0$) signal($S_2$);
    else signal($S_1$); }

  • $S_1$ protects $C$ from concurrent modification.
  • $S_2$ blocks process that calls wait() with $C<=0$.

  Must signal $S_1$ before waiting on $S_2$, otherwise nobody can pass through wait($S_1$) to signal($S_2$).

  If we signal $S_2$ instead of $S_1$, then this is because another process is waiting on $S_2$. That process will take care of signalling $S_1$ here:

  Intuitively, we pass on the key because we know somebody else is about to unlock.
Classical Problems of Synchronization

- Bounded-Buffer Problem.
- Readers and Writers Problem.
- Dining-Philosophers Problem.
Bounded-Buffer Problem

- **Shared data:**
  ```c
  semaphore full=0, empty=n, mutex=1;
  ```

- **Producer:**
  ```c
  int in=0;
  while (1) {
      /* produce an item in char nextProduced */
      wait(empty);
      wait(mutex);
      buffer[in]=nextProduced; in=(in+1)%B_SIZE;
      signal(mutex);
      signal(full);
  }
  ```

- **Consumer:**
  ```c
  int out=0;
  while (1) {
      wait(full);
      wait(mutex);
      nextConsumed=buffer[out]; out=(out+1)%B_SIZE;
      signal(mutex);
      signal(empty);
      /* consume the item in char nextConsumed */
  }
  ```

- **Binary semaphore:** only needed if buffer access is more complex than array access (e.g. shared linked list).

- **Counting semaphores**
Readers-Writers Problem

- **Shared data:**
  
  ```
  semaphore mutex=1, writer=1;
  int readCount=0;
  ```

- **Writer:** single one accessing shared file for output.
  ```
  while (1) {
    wait(writer);
    /* perform writing */
    signal(writer); }
  ```

- **Reader:** one or more can have shared read access to file.
  ```
  while (1) {
    wait(mutex);
    readCount++;
    if (readCount==1) wait(writer);
    signal(mutex);
    /* perform reading */
    wait(mutex);
    readCount--;
    if (readCount==0) signal(writer);
    signal(mutex); }
  ```

First reader locks out the writer by taking over `writer`.
Other readers stuck here: since first reader hasn't release `mutex`.

Last reader unlocks writer.

**Protects** `readCount`.

**Authorizes writer (if any; first reader otherwise) to proceed.**

**Writer may starve!**
Dining-Philosophers Problem

- Shared data:

  semaphore chopstick[5]; /* All 1. */

- Philosopher i:

  while (1) {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    /* eat */
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    /* think */
  }

Deadlock if each philosopher gets one chopstick!
Monitors

- Related, but not the same as Java monitors. Forget Java for now.
- High-level synchronization construct: at most one thread/process can be actively running any procedure within monitor. Rest are queued up.

```plaintext
monitor monitor-name
{
   /* shared variables */
   procedure body P1 (...) { ... }
   procedure body Pn (...) { ... }
   { /* initialization code */ }
}
```
To allow a process to suspend itself while executing in the monitor, a condition variable must be declared, as

```python
condition x;
```

Condition variable operations:
- `wait()`: if process P calls `x.wait()`, P is suspended until another process Q...
- `signal()`: ... runs `x.signal()` which resumes exactly one suspended process, if any; it is a no-op, otherwise.

When P is suspended, other procedures can run inside the monitor (like Q)...  
... but when Q signals P, who goes next (remember: only one process can run inside monitor)? No right/wrong answer.
Dining Philosophers Example

```java
monitor dp {
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  // also methods below
  void init() {
    for (int i=0;i<5;i++)
      state[i]=thinking;
  }

  void pickup(int i) {
    state[i]=hungry;
    test(i);
    if (state[i]!=eating)
      self[i].wait();
  }

  void putdown(int i) {
    state[i]=thinking;
    test((i+4)%5);
    test((i+1)%5);
  }

  void test(int i) {
    if ((state[(i+4)%5]!=eating) &&
        (state[i]==hungry) &&
        (state[(i+1)%5]!=eating)) {
      state[i]=eating;
      self[i].signal();
    }
  }
}
```

**Usage: Philosopher i:**
```java
while (1) {
  pickup(i);
  /* eat */
  putdown(i);
  /* think */
}
```

- **Deadlock impossible!**
- **Starvation possible!**

Philosopher i got hungry but couldn’t get both chopsticks at once, so she has to wait until another philosopher lets her have her chopsticks.

If i’s neighbors aren’t eating, and i wants to eat, then i can use their chopsticks.
Monitor Implementation with Semaphores

- **Shared data:**
  
  ```
  semaphore mutex=1, next=0, cSem=0;
  int nextCount=0, cCount=0;
  ```

- **External procedure proxy:**
  
  ```
  wait(mutex);
  /* call procedure */
  if (nextCount>0) signal(next);
  else signal(mutex);
  ```

- **Condition:**
  
  ```
  wait() {
    cCount=0;
    if (nextCount>0) signal(next);
    else signal(mutex);
    wait(cSem);
    cCount--; }
  ```

  ```
  signal() {
    if (cCount>0) {
      nextCount++;
      signal(cSem);
      wait(next);
      nextCount--; }
  }
  ```

- **1. Process S signals W. S has to wait while W unblocks, and until it leaves monitor.**
- **2. Number of S’s that are waiting on W’s.**
- **3. Coordinates S’s and W’s of a specific condition.**
- **4. Number of W’s waiting on the same condition for S’s to unblock them.**
- **5. Enable other processes to run within monitor. If an S had to wait after signalling, then unblock it. Otherwise, let anybody enter monitor.**
- **6. Must enable others to enter monitor or nobody will be able to signal us!**
- **7. W’s get stuck here**
- **8. S unblocks a W and lets W run while S blocks until W leaves**
- **9. Handles successive wait() s.**
Correct Monitor Use

- **Conditional-wait construct:**
  ```java
  x.wait(c)
  ```
  - Value of `c` is a *priority number*, stored with the name of the process that is suspended.
  - When `x.signal()` is executed, process with smallest priority number is resumed next.

- **Resource allocator:**
  ```java
  monitor ResourceAllocation {
    boolean busy=false;
    condition x;
    void acquire() {
      if (busy) x.wait(System.currentTimeMillis());
      busy=true; }
    void release() {
      busy=false;
      x.signal(); } }
  ```

- **Correctness of allocation technique requires:**
  - Users must always call `acquire()` once, then `release()` once.
  - User should not access resource directly.
Java synchronized

- Each object instance has a “monitor”:
  - It is combination of general monitor + a single condition variable.
  - Classes are objects (Class class). So each class has a monitor, too.
  - Ownership of instance monitor:
    ```
    void f() { synchronized (this) { /* owned */ } }
    ```
  - Ownership of class monitor:
    ```
    static synchronized void f() { /* owned */ }
    ```
  - General signal() is Java's notify(). notifyAll() is a Java extension.
  - To call general condition variable methods wait(), signal(), we must be inside general monitor. In Java, to execute wait(), notify(), notifyAll(), one must “own” the monitor.

- Avoids problem of entering critical section and forgetting to tell other threads we left it: syntax enforces this if-enter-must-leave requirement.

- Also solves volatile problem: make incr(), print() synchronized. Java forced to sync up master copies on monitor lock/unlock, and mutex is enforced.
Common mistake:

```java
mWaiting=false;
...
synchronized boolean waitIfFirst() throws InterruptedException {
    if (mWaiting) return false; // Somebody else is already waiting.
    mWaiting=true;
    synchronized (mEvent) { mEvent.wait(); }
    mWaiting=false;
    return true; } // Wait took place.
```

Intent: the first thread that calls `waitIfFirst()` must wait for an event; all others should just return without waiting. Method `synchronized` protects `mWaiting`.

Problem: when one thread starts waiting, it waits on `mEvent`’s monitor, not that of this so it still has the monitor of this. Hence all other threads block when they call `waitIfFirst()`.

Even in Java, multi-threading is tough:

Semaphore implementation is 60 lines of pure, hard-to-follow code.
Wrote 8000-line package to make it easier for average Stanford student. And, yes, first revision had bugs.
Critical Regions

- Another high-level synchronization construct.
- A shared variable $v$ of type $T$, is declared as:
  \[
  v : \text{shared } T
  \]
- Variable $v$ appears only inside statements of the form:
  \[
  \text{region } v \text{ when } B_i \ S_i;
  \]
  where $S_i$ is any other statement and $B_i$ is a boolean expression.
- While expression $B_i$ or statement $S_i$ are being executed, no other process can enter $B_i$ or $S_i$ to access variable $v$; in fact, it may not enter any such $B_j$ or $S_j$ protected by a region $v$. So code blocks referring to the same variable $v$ exclude each other in time.
- When a process tries to execute the region statement, it first blocks until it gains exclusive access to $v$, and then evaluates $B_i$:
  - If $B_i$ is true, statement $S_i$ is executed, and then exclusive access is relinquished.
  - If it is false, the process is blocked but it first relinquishes exclusive access; the process is also re-granted exclusive access to retest $B_i$ whenever another process successfully executes any $S_j$ (because its execution may have modified the value of $v$ and therefore $B_i$).
Bounded Buffer Example

- Shared data:
  ```
  #define B_SIZE 5
  char buffer[B_SIZE];
  count: shared int =0;
  ```

- Producer:
  ```
  int in=0;
  while (1) {
    /* produce an item in char nextProduced */
    region count when (count<B_SIZE) {
      buffer[in]=nextProduced;
      in=(in+1)%B_SIZE;
      count++;
    }
  }
  ```

- Consumer:
  ```
  int out=0;
  while (1) {
    region count when (count>0) {
      nextConsumed=buffer[out];
      out=(out+1)%B_SIZE;
      count--;
    }
    /* consume the item in char nextConsumed */
  }
  ```
wait(mutex);
while (!B) {
    firstCount++;
    if (secondCount>0) signal(secondDelay);
    else signal(mutex);
    wait(firstDelay);
    firstCount--;
    secondCount++;
    if (firstCount>0) signal(firstDelay);
    else signal(secondDelay);
    wait(secondDelay);
    secondCount--; } 

S;
if (firstCount>0) signal(firstDelay);
else if (secondCount>0) signal(secondDelay);
else signal(mutex);
If a single process P evaluates B to false, it receives ownership (responsibility to signal) of mutex because the first process to finish unblocks P instead of signalling mutex. mutex blocks other processes from evaluating B until P has had a chance to do so.

In general:
1. All processes that evaluate B to false block at firstDelay.
2. A process that evaluates B to true runs S and eventually unblocks another process P that was waiting on firstDelay.
3. P starts a cascade, letting all other processes waiting on firstDelay to go through. All these processes, P included, then block on secondDelay. But the last process through firstDelay signals Q, which is one of those blocked processes. Q can now move past secondDelay.
4. Q evaluates B again. If B is true, Q runs S and eventually signals another process to move past secondDelay. If B is false, Q goes around the loop and blocks at firstDelay again; but in doing so, it signals another process to move past secondDelay. And so on until either every process is blocked at firstDelay again (step 1) or one process evaluates B to true, runs S, and concludes by unblocking one of the processes waiting on firstDelay (step 2).
Mutually exclusive access to the critical section is provided by `mutex`.

If a process cannot enter the critical section because the `B` is false, it initially waits on the `firstDelay` semaphore; moved to the `secondDelay` semaphore before it is allowed to reevaluate `B`.

Keep track of the number of processes waiting on `firstDelay` and `secondDelay`, with `firstCount` and `secondCount` respectively.