Part III
Synchronization
Semaphores

The bearing of a child takes nine months, no matter how many women are assigned.

Frederick P. Brooks Jr.
A semaphore is an object that consists of a private counter, a private waiting list of processes, and two public methods (e.g., member functions): signal and wait.
Semaphore Method: **wait**

```c
void wait(sem S) {
    S.count--;  
    if (S.count < 0) {
        add the caller to the waiting list;
        block();
    }
}
```

- After decreasing the counter by 1, if the new value becomes negative, then
  - add the caller to the waiting list, and
  - block the caller.
Semaphore Method: \texttt{signal}

```c
void signal(sem S)
{
    S.count++;   
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume(P);
    }
}
```

- After increasing the counter by 1, if the new value is not positive (e.g., non-negative), then
  - remove a process \texttt{P} from the waiting list,
  - resume the execution of process \texttt{P}, and return
- If $S\.count < 0$, $\text{abs}(S\.count)$ is the number of waiting processes.

- This is because processes are added to (resp., removed from) the waiting list only if the counter value is $< 0$ (resp., $\leq 0$).
The waiting list can be implemented with a queue if FIFO order is desired.

However, the correctness of a program should not depend on a particular implementation (e.g., ordering) of the waiting list.
**Important Note: 3/4**

<table>
<thead>
<tr>
<th>S.count--;</th>
<th>S.count++;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (S.count&lt;0) {}</td>
<td>if (S.count&lt;=0) {}</td>
</tr>
<tr>
<td>add to list;</td>
<td>remove P;</td>
</tr>
<tr>
<td>block();</td>
<td>resume(P);</td>
</tr>
</tbody>
</table>

- The caller may be blocked in the call to `wait()`.
- The caller is never blocked in the call to `signal()`. If `S.count > 0`, `signal()` returns and the caller continues. Otherwise, a waiting process is released and the caller continues. In this case, **two** processes continue.
The Most Important Note: 4/4

- wait() and signal() must be executed **atomically** (i.e., as one uninterruptible unit).
- Otherwise, **race conditions** may occur.
- **Homework**: use execution sequences to show race conditions if wait() and/or signal() is not executed atomically.
There are three typical uses of semaphores:

- **mutual exclusion:**
  Mutex (i.e., *Mutual Exclusion*) locks

- **count-down lock:**
  Keep in mind that a semaphore has a private counter that can count.

- **notification:**
  Wait for an event to occur and indicate an event has occurred.
Use 1: Mutual Exclusion (Lock)

**semaphore**: \( S = 1 \)

**int** count = 0;  // shared variable

**Process 1**

```c
while (1) {
    // do something
    S.wait();
    count++;
    S.signal();
    // do something
}
```

**Process 2**

```c
while (1) {
    // do something
    S.wait();
    count--;  // critical sections
    S.signal();
    // do something
}
```

- What if the initial value of \( S \) is zero?
- \( S \) is a **binary semaphore** (count being 0 or 1).
Use 2: Count-Down Counter

semaphore S = 3

Process 1
while (1) {
    // do something
    S.wait();
    S.signal();
    // do something
}

Process 2
while (1) {
    // do something
    S.wait();
    S.signal();
    // do something
}

After three processes pass through wait(), this section is locked until a process calls signal().

at most 3 processes can be here!!!
Use 3: Notification

semaphore S1 = 1, S2 = 0;

process 1
while (1) {
    // do something
    S1.wait();
    cout << "1";
    S2.signal();
    // do something
}

process 2
while (1) {
    // do something
    S2.wait();
    cout << "2";
    S1.signal();
    // do something
}

- Process 1 uses `S2.signal()` to notify process 2, indicating “I am done. Please go ahead.”
- The output is 1 2 1 2 1 2 1 2 ......
- What if `S1` and `S2` are both 0’s or both 1’s?
- What if `S1 = 0` and `S2 = 1`?
Dining Philosophers

- Five philosophers are in a thinking - eating cycle.
- When a philosopher gets hungry, he sits down, picks up his left and then his right chopsticks, and eats.
- A philosopher can eat only if he has both chopsticks.
- After eating, he puts down both chopsticks and thinks.
- This cycle continues.
Dining Philosopher: Ideas

- Chopsticks are shared items (by two neighboring philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1 (i.e., available).
- A philosopher calls `wait()` to pick up a chopstick and `signal()` to release it.

```c
Semaphore C[5] = 1;

C[i].wait();
C[(i+1)%5].wait();
C[(i+1)%5].signal();
C[i].signal();
```

- `left chop locked` to `right chop locked`
Dining Philosophers: Code

semaphore C[5] = 1;

philosopher \( i \)
while (1) {
    // thinking
    C\[i]\].wait();
    C\[(i+1)\%5\].wait();
    // eating
    C\[(i+1)\%5\].signal();
    C\[i]\].signal();
    // finishes eating
}

Does this solution work?
Dining Philosophers: Deadlock!

- If all five philosophers sit down and pick up their left chopsticks at the same time, this causes a *circular waiting* and the program deadlocks.
- An easy way to remove this deadlock is to introduce a weirdo who picks up his right chopstick first!
Dining Philosophers: A Better Idea

semaphore C[5] = 1;

philosopher \( i \) \((0, 1, 2, 3)\)

while (1) {
    // thinking
    C[i].wait();
    C[(i+1)\%5].wait();
    // eating
    C[(i+1)\%5].signal();
    C[i].signal();
    // finishes eating;
}

Philosopher 4: the weirdo

while (1) {
    // thinking
    C[(i+1)\%5].wait();
    C[i].wait();
    // eating
    C[i].signal();
    C[(i+1)\%5].signal();
    // finishes eating
}
Dining Philosophers: Questions

- The following are some important questions for you to think about.
  - We choose philosopher 4 to be the weirdo. Does this choice matter?
  - Show that this solution does not cause *circular waiting*.
  - Show that this solution does not cause *circular waiting* even if we have more than 1 and less than 5 weirdoes.
- This solution is not *symmetric* because not all threads run the same code.
**Count-Down Lock Example**

- The naïve solution to the dining philosophers problem causes circular waiting.
- If only **four** philosophers are allowed to sit down, deadlock cannot occur.
- **Why?** If all four sit down at the same time, the right-most one may have both chopsticks!
- **What if the right-most one could not eat?** Exercise!
Count-Down Lock Example

semaphore C[5] = 1;
semaphore Chair = 4;

while (1) {
    // thinking
    Chair.wait();
    C[i].wait();
    C[(i+1)%5].wait();
    // eating
    C[(i+1)%5].signal();
    C[i].signal();
    Chair.signal();
}

this is a count-down lock that only allows 4 to go!

geta chair

release my chair

this is our old friend
The Producer/Consumer Problem

- Suppose we have a **circular buffer** of $n$ slots.
- Pointer **in** (resp., **out**) points to the first **empty** (resp., **filled**) slot.
- **Producer** processes keep adding data into the buffer.
- **Consumer** processes keep retrieving data from the buffer.
Problem Analysis

- A producer deposits data into \texttt{Buf[in]} and a consumer retrieves info from \texttt{Buf[out]}.
- \texttt{in} and \texttt{out} must be advanced.
- \texttt{in} is shared among producers.
- \texttt{out} is shared among consumers.
- If \texttt{Buf} is full, producers should be blocked.
- If \texttt{Buf} is empty, consumers should be blocked.

\textit{buffer is implemented with an array \texttt{Buf[ ]}}
- A semaphore to protect the buffer.
- Another semaphore to block producers if the buffer is full.
- One more semaphore to block consumers if the buffer is empty.
Solution

number of slots

```c
semaphore NotFull=n, NotEmpty=0, Mutex=1;

while (1) {
    NotFull.wait();
    Mutex.wait();
    Buf[in] = x;
    in = (in+1)%n;
    Mutex.signal();
    NotEmpty.signal();
}

producer

while (1) {
    NotFull.wait();
    Mutex.wait();
    Buf[in] = x;
    in = (in+1)%n;
    Mutex.signal();
    NotEmpty.signal();
}

consumer

while (1) {
    NotEmpty.wait();
    Mutex.wait();
    x = Buf[out];
    out = (out+1)%n;
    Mutex.signal();
    NotFull.signal();
}
```

notifications

critical section
Question

- What if the producer code is modified as follows?
- **Answer:** a deadlock may occur. Why?

```c
while (1) {
    Mutex.wait();
    NotFull.wait();
    Buf[in] = x;
    in = (in+1)%n;
    NotEmpty.signal();
    Mutex.signal();
}
```
The Readers/Writers Problem

- Two groups of processes, readers and writers, access a shared resource by the following rules:
  - Readers can read simultaneously.
  - Only one writer can write at any time.
  - When a writer is writing, no reader can read.
  - If there is any reader reading, all incoming writers must wait. Thus, readers have a higher priority.
Problem Analysis

- We need a semaphore to block readers if a writer is writing.
- When a writer arrives, it must know if there are readers reading. A reader count is required and must be protected by a lock.
- This reader-priority version has a problem: if readers keep coming in an overlapping way, waiting writers have no chance to write.
When a reader arrives, it adds 1 to the counter.

If it is the first reader, waits until no writer is writing.

Reads data.

Decreases the counter.

If it is the last reader, notifies the waiting readers and writers that no reader is reading.

Readers
When a writer comes in, it waits until no reader is reading and no writer is writing.

Then, it writes data.

Finally, notifies waiting readers and writers that no writer is writing.
Solution

semaphore Mutex = 1, WrtMutex = 1;
int RdrCount;

reader
while (1) {
    Mutex.wait();
    RdrCount++;
    if (RdrCount == 1)
        WrtMutex.wait();
    Mutex.signal();
    // read data
    Mutex.wait();
    RdrCount--;
    if (RdrCount == 0)
        WrtMutex.signal();
    Mutex.signal();
}

writer
while (1) {
    Mutex.wait();
    WrtMutex.wait();
    // write data
    Mutex.wait();
    WrtMutex.signal();
    Mutex.signal();
}
The Roller-Coaster Problem: 1/5

- Suppose there are $n$ passengers and one roller coaster car. The passengers repeatedly wait to ride in the car, which can hold maximum $C$ passengers, where $C < n$.
- The car can go around the track only when it is full. After finishes a ride, each passenger wanders around the amusement park before returning to the roller coaster for another ride.
- Due to safety concerns, the car only rides $T$ times and then shut-down.
The Roller-Coaster Problem: 2/5

- The car always rides with exactly $C$ passengers
- No passengers will jump off the car while the car is running
- No passengers will jump on the car while the car is running
- No passengers will request another ride before they get off the car.
The Roller-Coaster Problem: 3/5

- A passenger makes a decision to have a ride, and joins the queue.
- The queue is managed by a gate keeper.
- Passengers check in one-by-one.
- The last passenger tells the car that all passengers are on board.
- Then, they have a ride.
- After riding passengers get off the car one-by-one.
- They go back to play for a while and come back for a ride.
The Roller-Coaster Problem: 4/5

- The car comes and lets the gate keeper know it is available so that the gate keeper could release passengers to check in.
- The car is blocked for loading.
- When the last passenger in the car, s/he informs the car that all passengers are on board, the car starts a ride.
- After this, the car waits until all passengers are off. Then, go for another round.
The Roller-Coaster Problem: 5/5

int count = 0;
Semaphore Queue = Boarding = Riding = Unloading = 0;
Semaphore Check-In = 1;

Passenger
Wait(Queue);
Wait(Check-In);
if (++count==Maximum)
Signal(Boarding);
Signal(Check-In);
Wait(Riding);
Signal(Unloading);

Unload passengers one-by-one
Is this absolutely necessary?
Can Unloading be removed? Ex.

Car
for (i = 0; i < #rides; i++) {
  count = 0;  // reset counter before boarding
  for (j = 1; j <= Maximum; j++)
    Signal(Queue);  // car available
    Wait(Boarding);
    // all passengers in car
    // and riding
    Signal(Riding);
    Wait(Unloading);
}
// all passengers are off

one ride
A Quick Summary: 1/2

- We have learned a few tricks in this component: lock, count-down lock and notification.
- Very often a counter is needed to determine if certain condition is met (e.g., number of readers in the readers-writers problem, check-in and boarding in the roller-coaster problem).
- Sometimes threads may have to be “paired-up” like the get-off process we saw in the roller-coaster problem.
- Use these basic and frequently seen patterns to solve other problems.
A Quick Summary: 2/2

- Using many semaphores could mean more locking and unlocking activities, and could be inefficient.

- Using only a few semaphores could produce very large critical sections, and a thread could stay in a critical section for a long time. Thus, other threads may have to wait very long to get in.

- Therefore, try your best to minimize the number of semaphores and reduce the length of locking time.
What Is a Pattern?

A pattern is simply a description/template for solving a problem that can be used in several situations.

However, a pattern is NOT a complete solution to a problem. It is just a template and requires extra work to make it a solution to a specific problem.

We will discuss a few patterns related to the use of semaphores.
This is the easiest one for enforcing mutual exclusion so that race conditions will not occur.

A semaphore is initialized to 1. Then, use the `Wait()` and `Signal()` methods to lock and unlock the semaphore, respectively.

```c
Semaphore Lock(1);
Wait(Lock);
    // critical section
Signal(Lock);
```

```c
Semaphore S(1);
int c = 0;
Wait(S);
    Wait(Mutex);
c++;
Signal(S);
if (c >= 3) {
    ...
} else {
    ...
}
```
Enter-and-Test: 1/2

In many applications, a thread may enter a critical section and test for a condition. If that condition is met, the thread does something\textsubscript{1}. Otherwise, its does something\textsubscript{2}.

Frequently, one of the two somethings may involve a wait.

Reader: Enter

\begin{verbatim}
Mutex.wait();
RdrCount++;
if (RdrCount == 1)
  WrtMutex.wait();
Mutex.signal();
\end{verbatim}

if the condition (i.e., RdrCount being 1), then wait until it is notified by another thread. In this case, the first reader does something.
Usually, a wait may be used in the entry part to wait for a particular condition to occur, and a signal is used upon exit to notify other threads.

```java
Reader: Exit
// read data
Mutex.wait();
RdrCount--;
if (RdrCount == 0)
    WrtMutex.signal();
Mutex.signal();
}
```

if the condition (i.e., `RdrCount` being 0), then tell someone, a reader or a writer, to continue. In this case, the last reader does something.
Exit-Before-Wait: 1/2

- In many applications, a thread exits a critical section and then blocks itself.
- Usually, a thread updates some variables in a critical section, and then waits for a resource from another thread.

```
Roller-Coaster: Passenger
Wait(Queue);
Wait(Check-In);
if (++count==Maximum)
    Signal(Boarding);
Signal(Check-In);
Wait(Riding);
Signal(Unloading);
```

if the condition (i.e., \(count\) being the maximum) holds, then notify some thread.

after exiting the critical section, wait for some event to happen.
Exit-Before-Wait: 2/2

- This **signaling** an event followed by **waiting** on another has to be used with care.
- A context switch can happen between the **signal** and the **wait**.
- For example, a thread enters the critical section, signals **s1** upon exit, and gets swapped out before reaches the wait. This could cause a deadlock. **Why?** So, be careful!

```c
Wait(s1);
// critical section
Signal(s1);
Wait(s2);
```

- a context switch could occur here!
Conditional Waiting/Signaling

- A thread waits or notifies another thread if a condition is satisfied.
- Make sure that no race condition will occur while the condition is being tested.

```c
if (count > 0)          if (count == 0)
    Signal(OK_to_GO);       Wait(Block_Myself);
```

are there other threads updating `count` at the same time?
If a thread is in its critical section, it holds the **baton** (i.e., the critical section).

That thread passes the **baton** (i.e., the critical section) to another thread.

If there are waiting threads for a condition that is now true, the **baton** (i.e., the critical section) is passed to one of them.

If no thread is waiting, the **baton** is passed to the next thread that tries to enter the CS.

This is a technique that can make the use of semaphores more efficient.
The **Waiting** thread waits on **Condition** if **Event** is not there. The **Signaling** thread sets **Event** and releases the **Waiting** thread.

Semaphore Mutex(1);
Semaphore Condition(0);
Bool Event = FALSE;

**Waiting Thread**
Wait(Mutex);
while (!Event) {
  Signal(Mutex);
  Wait(Condition);
  Wait(Mutex);
}

**Signaling Thread**
Wait(Mutex);
  Event = TRUE;
  Signal(Condition);
  Signal(Mutex);

try again!

critical section for protecting **Event**
Passing the Baton: 3/5

- **Waiting** does not acquire the CS. Instead, **Signaling** has the CS, does not release it, and gives the CS to **Waiting** (i.e., baton passed)

- **Signaling** must be sure that **Waiting** will not do any harm to the CS.

```
Waiting Thread
Wait(Mutex);
while (!Event) {
    Signal(Mutex);
    Wait(Condition);
    Wait(Mutex);
}

Signaling Thread
Wait(Mutex);
Event = TRUE;
Signal(Condition);
Signal(Mutex);
```
Passing the Baton: 4/5

Semaphore Mutex(1), Condition(0);
int Event = FALSE, Waiting = 0;

**Waiting Thread**

Wait (Mutex);
if (!Event) {
    Waiting++;
    Signal (Mutex);
    Wait (Condition);
}
...
if (Waiting > 0) {
    Waiting--;
    Signal (Condition);
} else

**Signaling Thread**

Wait (Mutex);
Event = TRUE;

a Mutex needed to protect Waiting
baton acquired

else
    Signal (Mutex);

baton passed

baton released

...
Passing the Baton: 5/5

- **Passing the baton** technically transfers the ownership of a critical section from a thread to another thread.

- The thread that has the baton does not need a signal to release it. Instead, the CS is directly given to another that needs it. The receiving thread does not need a wait for the CS.

- In this way, mutual exclusion may be destroyed; but, we reduce the number of entering and exiting a mutex.
Passing the Baton: Example

- We shall use the reader-priority version of the readers/writers problem as a more complex example.

- Note the following conditions:
  - If there is no writer writing, a reader can read.
  - If there is no readers reading and there are waiting writers, allow a writer to write (i.e., better!).
  - If there are readers reading OR a writer writing, no writer can write.
  - If there are waiting readers, a finishing writer should allow a reader to read (i.e. reader priority).
  - If there are waiting writers and no waiting reader, a finishing writer should allow a writer to write.
Passing the Baton: Example

- We will need counters for counting waiting readers and writers and active readers and writer.
- A semaphore for protecting all counters.
- A semaphore for blocking readers.
- A semaphore for blocking writers.

```c
int aReaders = 0; // number of active readers (>= 0)
int aWriters = 0; // number of active writer (0 or 1)
int wReaders = 0; // number of waiting readers
int wWriters = 0; // number of waiting writers

Semaphore Mutex(1); // semaphore for protecting counters
Semaphore Reader(0); // semaphore for blocking readers
Semaphore Writer(0); // semaphore for blocking writers
```
Developing the Entry Section for Readers

The first reader, who sees no writer writing and some readers waiting, releases the waiting readers.

The release of readers is in a cascading way (i.e., one after the other).

The first thread acquires the baton, and may pass it to each of the released thread.

Then, the last thread releases the baton!
Developing the Entry Section for **Readers**: **Passing the Baton**

The first reader, who gets the baton and sees no writer writing and some readers waiting, will release the waiting readers and pass its baton to the released reader. Each released reader, after releasing the next reader, passes the baton to it.

The last released returns the baton. Note that the critical section (i.e., the baton) is locked throughout this cascading releasing, and no other thread can pass through the first `Wait(Mutex)`.

```c
Wait(Mutex);
if (aWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
    Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
else
    Signal(Mutex);
```
Developing the Exit Section for **Readers**

Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
    wWriters++;
    [Signal(Mutex)];
    Signal(Writer);
}
else
    Signal(Mutex);

If this is the **last reader** and there are **waiting writers**, let one writer go.

Otherwise, simply go away.

This signal is not needed, because the last reader can pass the baton to the released writer!
### Developing the Exit Section for Readers: Passing the Baton

```c
Wait(Mutex);
aReaders--;
if (aReaders==0 & wWriters>0) {
    wWriters++;
    Signal(Mutex);
    Signal(Writer);
}
else
    Signal(Mutex);
```

If this is the last reader and there are waiting writers, let one writer go.
 Otherwise, simply go away.

The baton is passed to the released writer.

Would this work properly in terms of passing the baton? Why? Exercise!
Developing the Entry Section for **Writers**

```c
Wait(Mutex);
if (aReaders>0 | aWriters>0) {
    wWriters++;
    Signal(Mutex);
    Wait(Writer);
    Wait(Mutex);
}
```

A writer, who gets the baton and sees **some readers reading** OR **a writer writing**, will release the baton and wait. Later, this writer reacquires the baton to update **aWriters**.

Otherwise (i.e., no readers reading **AND** no writer writing), it increases the number of active writers, releases the baton, and starts writing.

**Passing the Baton: Example**
Passing the Baton: Example

Developing the Entry Section for **Writers**:

```java
Wait(Mutex);
if (aReaders>0 | aWriters>0) {
    wWriters++;
    Signal(Mutex);
    Wait(Writer);
    Wait(Mutex);
}
aWriters++;
Signal(Mutex);
```

A writer, who gets the baton and sees some readers reading **OR** a writer writing, will release the baton and wait. Otherwise (i.e., no readers reading **AND** no writer writing), it increases the number of active writers, releases the baton, and starts writing.

Exit Section of Readers: **Passing the Baton**

```java
Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
    wWriters++;
    Signal(Writer);
}
else
    Signal(Mutex);
```

The exiting reader acquires the baton, releases a writer w/o releasing the baton. Hence, the released writer has the baton, increases the active writers count, releases the baton, and writes.
Developing the Exit Section for **Writers**

- Lock the **Mutex** first.
- If there are waiting readers, let one of them go.
- If there is no waiting readers but there are waiting writers, let one of them go.
- If there is no waiting readers and no waiting writers, then do nothing.
- Don’t forget to release the **Mutex**.

```c
Wait(Mutex); acquire
aWriters--;
if (wReaders > 0) {
    wReaders--;
    Signal(Mutex);
    Signal(Readers);
}
else if (wWriters > 0) {
    wWriters--;
    Signal(Mutex);
    Signal(Writer);
}
else
    Signal(Mutex);
```
Developing the Exit Section for **Writers**: Passing the Baton

This Writer has acquired the baton.

It can pass the baton to a released Reader or a released Writer.

The baton is not passed if there is no waiting readers and there is no waiting writers.

In this case, this writer has to release the baton explicitly.
Summary
Passing the Baton

// READING
Wait(Mutex);
aReaders--;
if (aReaders = 0 & wWriters > 0) {
    wWriters++;
    Signal(Writer);
}
else
    Signal(Mutex);

// WRITING
Wait(Mutex);
aWriters--;
if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
else if (wWriters > 0) {
    wWriters--;
    Signal(Writer);
}
else // wReaders = wWriters = 0
    Signal(Mutex);
Wait(Mutex);
if (aWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
}
else
    Signal(Mutex);

// READING
Wait(Mutex);
aReaders--;
if (aReaders = 0 & wWriters > 0) {
    wWriters++;
    Signal(Reader);
}
else
    Signal(Mutex);

else // wReaders = wWriters = 0
    Signal(Mutex);

Wait(Mutex);
if (aReaders > 0 | aWriters > 0) {
    wWriters++;
    Signal(Mutex);
    Wait(Writer);
}
aWriters++;

// WRITING
Wait(Mutex);
else if (wWriters > 0) {
    wWriters--;
    Signal(Writer);
}
else
    Signal(Mutex);

Wait(Mutex);
else if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
else
    Signal(Mutex);

Wait(Mutex);
aWriters--;
if (wReaders > 0) {
    wWriters--;
    Signal(Reader);
}
else if (wWriters > 0) {
    wWriters--;
    Signal(Writer);
}
else // wReaders = wWriters = 0
    Signal(Mutex);
Passing the Baton: Example

- One of the several advantages of this solution is that it can easily be modified to achieve other goals. Here is a writer-priority version.

- A **writer-priority** version should satisfy the following conditions:
  1. New readers are blocked if a writer is waiting, and
  2. A waiting reader is released if no writer is writing.

Study the conditions for releasing readers and writers.
To meet **Condition (1)**, we have to change the first (enter) `if` statement of the reader thread:

```plaintext
if (aWriters > 0 || wWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
}
```

When a reader arrives, if there are writers waiting or writing, this reader blocks, and, hence yields to writers.
To meet **Condition (2)**, we have to strengthen the last (exit) `if` statement of the writer thread:

```c
if (wReaders > 0 && wWriters == 0) { // release readers
    wReaders--; // only if there
    Signal(Reader); // is no writers
} else if (wWriters > 0) { // if no readers
    wWriters--; // but some writers
    Signal(Writer); // release one
} else // no readers/writers
    Signal(Mutex);
```

Release a reader if there is no writer waiting. Otherwise (i.e., no waiting readers or some waiting writers), release a writer. Finally (i.e., no waiting readers and no waiting writers), do nothing.
Passing the Baton: Example

- A **fair** version should allow readers and writers take turns (i.e., no starvation).
- We assume the semaphores being used is implemented so that every blocked thread will be eventually released (i.e., no starvation).
- A fair version must satisfy the following:
  1. When a writer finishes, all waiting readers get a turn
  2. When all current readers finish reading, one waiting writer can write.

Study the conditions for releasing readers and writers.
Passing the Baton: Example

- We need to change the last (exit) if of the writer.
- A new `Bool` variable `Writing` is need to indicate whether a writer is writing. `Writing` is set to `TRUE` when a writer starts writing, and is set to `FALSE` when a reader starts reading.

```c
if (wReaders > 0 & (wWriters = 0 | Writing)) {
    wReaders--; // NEW: no waiting or writing writer
    Signal(Reader);
}
else if (wWriters > 0 & (wReaders = 0 | ¬Writing)) {
    wWriters--; // NEW: no waiting readers or no writing writer
    Signal(Writer);
}
else // wReaders = wWriters = 0
    Signal(Mutex);
```

- This is set by the current exiting writer. Because it exits, no writer is writing.
- a reader can read
- a writer can writer

NEW: no waiting readers or no writing writer
Passing the Baton: Example

This example, including its extensions and the passing the baton pattern, is due to Gregory R. Andrews. Refer to the following for more detailed discussions.


Semaphores with ThreadMentor
**Semaphores with ThreadMentor**

- **ThreadMentor** has a class `Semaphore` with two methods `Wait()` and `Signal()`.
- Class `Semaphore` requires a non-negative integer as an initial value.
- A name is optional.

```cpp
Semaphore Sem(“S”,1);
Sem.Wait();
// critical section
Sem.Signal();

Semaphore *Sem;
Sem = new Semaphore(“S”,1);
Sem->Wait();
// critical section
Sem->Signal();
```
Dining Philosophers: 4 Chairs

Semaphore Chairs(4);
Mutex Chops[5];

class phil::public Thread
{
  public:
    phil(int n, int it);
  private:
    int  Number;
    int  iter;
    void ThreadFunc();
};

Void phil::ThreadFunc()
{
  int i, Left=Number,
      Right=(Number+1)%5;
  Thread::ThreadFunc();
  for (i=0; i<iter; i++) {
    Chairs.Wait();
    Chops[Left].Lock();
    Chops[Right].Lock();
    // Eat
    Chops[Left].Unlock();
    Chops[Right].Unlock();
    Chairs.Signal();
  }
}

Count-Down and Lock!
The Smokers Problem: 1/6

- Three ingredients are needed to make a cigarette: tobacco, paper and matches.
- An agent has an infinite supply of all three.
- Each of the three smokers has an infinite supply of one ingredient only. That is, one of them has tobacco, the second has paper, and the third has matches.
- They share a table.
The Smokers Problem: 2/6

- The agent adds two randomly selected different ingredients on the table, and notifies the needed smoker.
- A smoker waits until agent’s notification. Then, takes the two needed ingredients, makes a cigarette, and smokes for a while.
- This process continues forever.
- How can we use semaphores to solve this problem?
The Smokers Problem: 3/6
The Smokers Problem: 4/6

- Semaphore Table protects the table.
- Three semaphores Sem[3] are used, one for each smoker:

<table>
<thead>
<tr>
<th>Smoker #</th>
<th>Has</th>
<th>Needs</th>
<th>Sem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 &amp; 2</td>
<td>Sem[0]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2 &amp; 0</td>
<td>Sem[1]</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0 &amp; 1</td>
<td>Sem[2]</td>
</tr>
</tbody>
</table>
```cpp
class A::public Thread {
private:
    void ThreadFunc();
};

class Smk::public Thread {
public:
    Smk(int n);
private:
    void ThreadFunc();
    int No;
};

Smk::Smk(int n) {
    No = n;
}

Void Smk::ThreadFunc() {
    Thread::ThreadFunc();
    while (1) {
        Sem[No]->Wait();
        Table.Signal();
        // smoker a while
    }
}
```

waiting for ingredients

agent thread

smoker thread

clear the table

The Smokers Problem: 5/6
void A::ThreadFunc()
{
    Thread::ThreadFunc();
    int Ran;
    while (1) {
        Ran = // random # // in [0,2]
        Sem[Ran]->Signal();
        Table.Wait();
    }
}

void main()
{
    Smk *Smoker[3];
    A Agent;
    Agent.Begin();
    for (i=0;i<3;i++) {
        Smoker = new Smk(i);
        Smoker->Begin();
    }
    Agent.Join();
}
The End