CS 31: Intro to Systems C Programming
L24: Synchronization and Race Conditions

Vasanta Chaganti & Kevin Webb
Swarthmore College
Dec 7, 2023
Reading Quiz
Threads

• Modern OSes separate the concepts of processes and threads.
  – The process defines the address space and general process attributes (e.g., open files)
  – The thread defines a sequential execution stream within a process (PC, SP, registers)

• A thread is bound to a single process
  – Processes, however, can have multiple threads
  – Each process has at least one thread (e.g. main)
This is the picture we’ve been using all along:

A process with a single thread, which has execution state (registers) and a stack.
Kernel-Level Threads

Kernel Context switching over threads

Each process has explicitly mapped regions for stacks
Synchronization

• Synchronize: to (arrange events to) happen such that two events do not overwrite each other’s work.

• Thread synchronization
  – When one thread has to wait for another
  – Events in threads that occur “at the same time”

• Uses of synchronization
  – Prevent race conditions
  – Wait for resources to become available (only one thread has access at any time - deadlocks)
### Synchronization: Too Much Milk (TMM)

<table>
<thead>
<tr>
<th>Time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00</td>
<td>Arrive home</td>
<td></td>
</tr>
<tr>
<td>3.05</td>
<td>Look in fridge, no milk</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Leave for the grocery store</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Arrive at the grocery store</td>
<td></td>
</tr>
<tr>
<td>3.25</td>
<td>Buy Milk</td>
<td></td>
</tr>
<tr>
<td>3.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive Home</td>
</tr>
<tr>
<td>3.40</td>
<td></td>
<td>Look in fridge, find milk</td>
</tr>
<tr>
<td>3.45</td>
<td></td>
<td>Cold Coffee (nom)</td>
</tr>
</tbody>
</table>

What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?
How many cartons of milk can we have in this scenario? (Can we ensure this somehow?)

<table>
<thead>
<tr>
<th>Time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00</td>
<td>Arrive home</td>
<td></td>
</tr>
<tr>
<td>3.05</td>
<td>Look in fridge, no milk</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Leave for the grocery store</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Arrive at the grocery store</td>
<td></td>
</tr>
<tr>
<td>3.25</td>
<td>Buy Milk</td>
<td></td>
</tr>
<tr>
<td>3.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive Home</td>
</tr>
<tr>
<td>3.40</td>
<td></td>
<td>Look in fridge, find milk</td>
</tr>
<tr>
<td>3.45</td>
<td></td>
<td>Cold Coffee (nom)</td>
</tr>
</tbody>
</table>

A. One carton (you)
B. Two cartons
C. No cartons
D. Something else
### Synchronization: Too Much Milk (TMM): One possible scenario

**What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?**

<table>
<thead>
<tr>
<th>Time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00</td>
<td>Arrive home</td>
<td></td>
</tr>
<tr>
<td>3.05</td>
<td>Look in fridge, no milk</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Leave for grocery</td>
<td>Arrive Home</td>
</tr>
<tr>
<td>3.15</td>
<td></td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3.20</td>
<td>Arrive at grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3.25</td>
<td>Buy Milk</td>
<td></td>
</tr>
<tr>
<td>3.30</td>
<td></td>
<td>Arrive at grocery</td>
</tr>
<tr>
<td>3.35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive home, put milk in fridge</td>
</tr>
<tr>
<td>3.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.45</td>
<td></td>
<td>Oh No!</td>
</tr>
</tbody>
</table>
• Coordination required:
  – Which thread goes first?
  – Threads in different regions must work together to compute new value for boundary cells.
  – Threads might not run at the same speed (depends on the OS scheduler). Can’t let one region get too far ahead.
  – Context switches can happen at any time!
Thread Ordering
(Why threads require care. Humans aren’t good at reasoning about this.)

- As a programmer you have no idea when threads will run. The OS schedules them, and the schedule will vary across runs.

- It might decide to context switch from one thread to another at any time.

- Your code must be prepared for this!
  – Ask yourself: “Would something bad happen if we context switched here?”

- hard to debug this problem if it is not reproducible
Example: The Credit/Debit Problem

- Say you have $1000 in your bank account
  - You deposit $100
  - You also withdraw $100

- How much should be in your account?

- What if your deposit and withdrawal occur at the same time, at different ATMs?
Credit/Debit Problem: Race Condition

Thread $T_0$

```java
Credit (int a) {
    int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
}
```

Thread $T_1$

```java
Debit (int a) {
    int b;
    b = ReadBalance ();
    b = b - a;
    WriteBalance (b);
    PrintReceipt (b);
}
```
Credit/Debit Problem: Race Condition

Thread T₀
Credit (int a) {
    int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
}

Thread T₁
Debit (int a) {
    int b;
    b = ReadBalance ();
    b = b - a;
    WriteBalance (b);
    PrintReceipt (b);
}

Say T₀ runs first
Read $1000 into b
Credit/Debit Problem: Race Condition

**Thread T\(_{0}\)**
Credit (int a) {
    int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
}

**Thread T\(_{1}\)**
Debit (int a) {
    int b;
    b = ReadBalance ();
    b = b - a;
    WriteBalance (b);
    PrintReceipt (b);
}

Say T\(_{0}\) runs first
Read $1000 into b
Switch to T\(_{1}\)
Read $1000 into b
Debit by $100
Write $900

CONTEXT SWITCH
Credit/Debit Problem: Race Condition

Say $T_0$ runs first
- Read $1000$ into $b$
- Switch to $T_1$
- Read $1000$ into $b$
- Debit by $100$
- Write $900$

$T_0$ Credit (int $a$) {
    int $b$;
    $b$ = ReadBalance ()
    $b$ = $b$ + $a$
    WriteBalance ($b$);
    PrintReceipt ($b$);
}

Switch back to $T_0$
- Read $1000$ into $b$
- Credit $100$
- Write $1100$

Say $T_1$ runs first
- Read $1000$ into $b$
- Switch to $T_0$
- Read $1000$ into $b$
- Debit by $100$
- Write $900$

$T_1$ Debit (int $a$) {
    int $b$;
    $b$ = ReadBalance ()
    $b$ = $b$ - $a$
    WriteBalance ($b$);
    PrintReceipt ($b$);
}

Bank gave you $100$!

What went wrong?
Thread $T_0$

Credit (int $a$) {
    int $b$;
    
    $b = \text{ReadBalance}()$;
    $b = b + a$;
    $\text{WriteBalance} (b)$;
    
    $\text{PrintReceipt} (b)$;
}

Thread $T_1$

Debit (int $a$) {
    int $b$;
    
    $b = \text{ReadBalance} ()$;
    $b = b - a$;
    $\text{WriteBalance} (b)$;
    
    $\text{PrintReceipt} (b)$;
}

Bank gave you $\$100$!

What went wrong?
To Avoid Race Conditions

1. Identify critical sections

2. Use synchronization to enforce mutual exclusion
   – Only one thread active in a critical section
Critical Section and Atomicity

• Sections of code executed by multiple threads
  – Access shared variables, often making local copy
  – Places where order of execution or thread interleaving will affect the outcome
  – Follows: read + modify + write of shared variable

• Must run atomically with respect to each other
  – Atomicity: runs as an entire instruction or not at all. Cannot be divided into smaller parts.
Which code region is a critical section?

Thread A

```c
main ()
{
    int a,b;

    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    a += 1
    return a;
}
```

Thread B

```c
main ()
{
    int a,b;

    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    a += 1
    return a;
}
```
Which code region is a critical section?
read + modify + write of shared variable

Large enough for correctness + Small enough to minimize slow down
Which values might the shared $s$ variable hold after both threads finish?

**Thread A**

```c
main ()
{
    int a, b;
    a = getShared();
b = 10;
a = a + b;
saveShared(a);
    a += 1
    return a;
}
```

**Thread B**

```c
main ()
{
    int a, b;
a = getShared();
b = 20;
a = a - b;
saveShared(a);
    a += 1
    return a;
}
```

**shared memory**

$s = 40$;
If A runs first

Thread A

```c
main ()
{
  int a, b;
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  a += 1
  return a;
}
```

Thread B

```c
main ()
{
  int a, b;
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  a += 1
  return a;
}
```

$s = 40$
$s = 50$
B runs after A Completes

Thread A

```c
main ()
{
    int a,b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    a += 1
    return a;
}
```

Thread B

```c
main ()
{
    int a,b;
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    a += 1
    return a;
}
```
What about interleaving?

One of the threads will overwrite the other’s changes.
Is there a race condition?

Suppose `count` is a global variable, multiple threads increment it:

```
count++;  
```

A. Yes, there’s a race condition (`count++` is a critical section).
B. No, there’s no race condition (`count++` is not a critical section).
C. Cannot be determined

How about if compiler implements it as:

```
movq (%rdx), %rax  // read count value
addq $1, %rax     // modify value
movq %rax, (%rdx)  // write count
```

How about if compiler implements it as:

```
incq (%rdx)       // increment value
```
Atomicity

• The implementation of acquiring/releasing critical section must be atomic.
  – An atomic operation is one which executes as though it could not be interrupted
  – Code that executes “all or nothing”

• How do we make them atomic?
  – Atomic instructions (e.g., test-and-set, compare-and-swap)
  – Allows us to build “semaphore” OS abstraction
Four Rules for Mutual Exclusion

1. No two threads can be inside their critical sections at the same time (one of many but not more than one).

2. No thread outside its critical section may prevent others from entering their critical sections.

3. No thread should have to wait forever to enter its critical section. (Starvation)

4. No assumptions can be made about speeds or number of CPU’s.
Railroad Semaphore
- Help trains figure out which track to be on at any given time.
Railroad Semaphore
- Help trains figure out which track to be on at any given time.

O.S. Semaphore:
- Construct that the OS provides to processes.
- Make system calls to modify their value
Mutual Exclusion with Semaphores

mutex = 1; //lock and unlock mutex atomically.

\[ T_0 \]
\[
lock (mutex);
\]
\[
< \text{critical section} >
\]
\[
unlock (mutex);
\]

\[ T_1 \]
\[
lock (mutex);
\]
\[
< \text{critical section} >
\]
\[
unlock (mutex);
\]

Atomicity: run the entire instruction without interruption.
Mutual Exclusion with Semaphores

mutex = 1; //unlocked.

T₀
lock (mutex);
< critical section >
unlock (mutex);

T₁
lock (mutex);
< critical section >
unlock (mutex);

Atomicity: run the entire instruction without interruption.

T₀: Wants to execute the critical section

T₀: Reads the value of mutex,
Changes the value of mutex = 0 (acquires lock)
Enteres critical section.
Mutual Exclusion with Semaphores

mutex = 0; //locked.

\[\begin{align*}
T_0 & : \text{Wants to execute the critical section} \\
T_0 & : \text{Reads the value of mutex,} \\
& \quad \text{Changes the value of mutex} = 0 \ (\text{acquires lock}) \\
& \quad \text{Enters critical section.}
\end{align*}\]

Atomicity: run the entire instruction without interruption.
Mutual Exclusion with Semaphores

```plaintext
mutex = 0; //locked.

T₀
lock (mutex);
< critical section >
unlock (mutex);

T₁ (blocked)
lock (mutex);
< critical section >
unlock (mutex);
```

Atomicity: run the entire instruction without interruption.

T₀: In the critical section
T₁: Wants to enter the critical section.
    Reads the value of mutex (mutex = 0)
    Cannot enter critical section.
    Blocked.
Mutual Exclusion with Semaphores

mutex = 0; //locked.

T₀
lock (mutex);
< critical section >
unlock (mutex);

T₁ (blocked)
lock (mutex);
< critical section >
unlock (mutex);

Atomicity: run the entire instruction without interruption.

T₀: Completes execution of critical section
Updates mutex value = 1. (release lock)
Atomic Execution

Mutual Exclusion with Semaphores

mutex = 1; //unlocked.

\( T_0 \)
\[
\text{lock (mutex);} \\
\text{< critical section >} \\
\text{unlock (mutex);} \\
\]

\( T_1 \) (blocked)
\[
\text{lock (mutex);} \\
\text{< critical section >} \\
\text{unlock (mutex);} \\
\]

Atomicity: run the entire instruction without interruption.

\( T_0 \): Completes execution of critical section
Updates mutex value = 1. (release lock)
Mutual Exclusion with Semaphores

mutex = 1; //locked.

\[ T_0 \]
lock (mutex);
< critical section >
unlock (mutex);

\[ T_1 \]
lock (mutex);
< critical section >
unlock (mutex);

Atomicity: run the entire instruction without interruption.

\[ T_1 \]: Can now acquire lock atomically and
Enter the critical section
Mutual Exclusion with Semaphores

- Use a “mutex” semaphore initialized to 1
- Only one thread can enter critical section at a time.
- Simple, works for any number of threads

Atomicity: runs as an entire instruction or not at all.
Semaphores

- Semaphore: OS synchronization variable
  - Has integer value
  - List of waiting threads
- Works like a gate
- If sem > 0, gate is open
  - Value equals number of threads that can enter
- Else, gate is closed
  - Possibly with waiting threads
Semaphores

• Associated with each semaphore is a queue of waiting threads
• When wait() is called by a thread:
  – If semaphore is open, thread continues
  – If semaphore is closed, thread blocks on queue
• Then signal() opens the semaphore:
  – If a thread is waiting on the queue, the thread is unblocked
  – If no threads are waiting on the queue, the signal is remembered for the next thread
Semaphore Operations

```plaintext
sem s = n;  // declare and initialize

wait (sem s)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)  // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

(*) With help from special hardware instructions.
Semaphore Operations

```plaintext
sem s = n; // declare and initialize

wait (sem s) // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s) // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

Based on what you know about semaphores, should a process be able to check beforehand whether `wait(s)` will cause it to block?

A. Yes, it should be able to check.
B. No, it should not be able to check.
Semaphore Operations

```
sem s = n;  // declare and initialize

wait (sem s)  // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)  // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

- No other operations allowed
- In particular, semaphore’s value can’t be tested!
  - No thread can tell the value of s
Synchronization: More than Mutexes

• “I want to block a thread until something specific happens.”
  – Condition variable: wait for a condition to be true

• “I want all my threads to sync up at the same point.”
  – Barrier: wait for everyone to catch up.
Barriers

• Used to coordinate threads, but also other forms of concurrent execution.

• Often found in simulations that have discrete rounds. (e.g., game of life)
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Barrier (0 waiting)
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads make progress computing current round at different rates.
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads that make it to barrier must wait for all others to get there.
Barrier Example, N Threads

```c
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}
```

Barrier allows threads to pass when N threads reach it.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads compute next round, wait on barrier again, repeat...
Synchronization: More than Mutexes

• I want all my threads to sync up at the same point.
  – Barrier: wait for everyone to catch up.

• I want to block a thread until something specific happens.
  – Condition variable: wait for a condition to be true

• I want my threads to share a critical section when they’re reading, *but still safely write*.
  – Readers/writers lock: distinguish how lock is used
Synchronization: Beyond Mutexes
Message Passing

- Operating system mechanism for IPC
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization: can’t receive until message is sent
Summary

• We have no idea when OS will schedule or context switch our threads.
  – Code must be prepared, tough to reason about.

• Threads often must synchronize
  – To safely communicate / transfer data, without races

• Synchronization primitives help programmers
  – Kernel-level semaphores: limit # of threads that can do something, provides atomicity
  – User-level locks: built upon semaphore, provides mutual exclusion (usually part of thread library)
Additional Slides: Solution to the Race Condition
Solution with mutexes

Thread A

```c
main ()
{
    int a,b;

    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);

    return a;
}
```

Thread B

```c
main ()
{
    int a,b;

    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);

    return a;
}
```

Shared memory

```c
s = 40;
```
Using Locks

Thread A

```c
main ()
{ int a,b;

    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);

    return a;
}
```

Thread B

```c
main ()
{ int a,b;

    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);

    return a;
}
```

shared memory

s = 40;
Using Locks

Thread A

```c
main ()
{
  int a, b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```c
main ()
{
  int a, b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

Lock Held by: Nobody
Using Locks

Thread A

```c
main ()
{ int a,b;

    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);

    return a;
}
```

Thread B

```c
main ()
{ int a,b;

    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);

    return a;
}
```

Lock held by:
Thread A
Using Locks

Thread A

```c
main ()
{
    int a,b;

    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);

    return a;
}
```

Thread B

```c
main ()
{
    int a,b;

    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);

    return a;
}
```

s = 40;
Lock 1;
Using Locks

Thread A

```c
main ()
{ int a,b;

    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);

    return a;
}
```

Thread B

```c
main ()
{ int a,b;

    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);

    return a;
}
```

Lock held by:
Thread A
Using Locks

Thread A

```c
main ()
{
    int a,b;
    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);
    return a;
}
```

Thread B

```c
main ()
{
    int a,b;
    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);
    return a;
}
```

Lock Held by: Nobody
Using Locks

Thread A

```c
main ()
{
    int a, b;

    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);

    return a;
}
```

Thread B

```c
main ()
{
    int a, b;

    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);

    return a;
}
```

s = 40;
Lock 1;

Lock held by:
Thread B
Using Locks

Thread A

```c
main ()
{
    int a,b;

    acquire(l);
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    release(l);

    return a;
}
```

Thread B

```c
main ()
{
    int a,b;

    acquire(l);
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    release(l);

    return a;
}
```

s = 40;
Lock 1;

Lock Held by: Nobody
Using Locks

- No matter how we order threads or when we context switch, result will always be 30, like we expected (and probably wanted).