Module 05: Synchronization Tools and Classic Problems

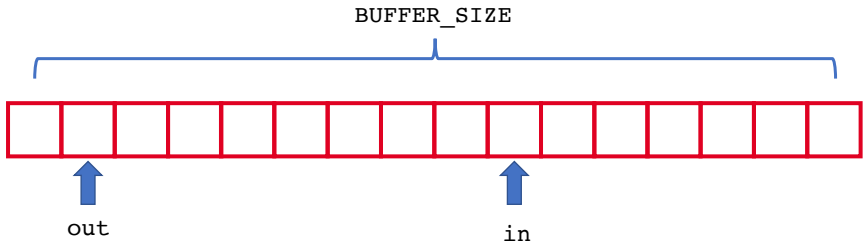
1

Producer Code

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Consumer Code

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```



2

Producer Code

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

$register_1 = \text{count}$
$register_1 = register_1 + 1$
$\text{count} = register_1$

Consumer Code

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

$register_2 = \text{count}$
$register_2 = register_2 - 1$
$\text{count} = register_2$

3

Producer Code

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

$register_1 = \text{count}$
$register_1 = register_1 + 1$
$\text{count} = register_1$

Consumer Code

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

$register_2 = \text{count}$
$register_2 = register_2 - 1$
$\text{count} = register_2$

THESE OPERATIONS ARE NOT ATOMIC AT THE LEVEL OF ACTUAL CODE EXECUTION AT MACHINE CODE LEVEL

4

Producer Code

```
register1 = count
register1 = register1 + 1
count = register1
```

Consumer Code

```
register2 = count
register2 = register2 – 1
count = register2
```

T_0	producer	$register_1$=count
T_1	producer	$register_1$=$register_1$+1
T_2	consumer	$register_2$=count
T_3	consumer	$register_2$=$register_2$-1
T_4	producer	count=$register_1$
T_5	consumer	count=$register_2$

5

Producer Code

```
lock(lock_variable)
register1 = count
register1 = register1 + 1
count = register1
unlock(lock_variable)
```

Consumer Code

```
lock(lock_variable)
register2 = count
register2 = register2 – 1
count = register2
unlock(lock_variable)
```

Sequence One

T_0	producer	$register_1$=count
T_1	producer	$register_1$=$register_1$+1
T_2	producer	count=$register_1$
T_3	consumer	$register_2$=count
T_4	consumer	$register_2$=$register_2$-1
T_5	consumer	count=$register_2$

Sequence Two

T_1	consumer	$register_2$=count
T_2	consumer	$register_2$=$register_2$-1
T_3	consumer	count=$register_2$
T_4	producer	$register_1$=count
T_5	producer	$register_1$=$register_1$+1
T_6	producer	count=$register_1$

Either of these two sequences is possible, and either is acceptable as they both represent outcomes we would expect

6

Producer Code

```
while (true) {
   /* Produce an item in next_produced */

   while (count == BUFFER_SIZE) ;

   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   lock(lock_variable);
   count++
   unlock(lock_variable);
}
```

Consumer Code

```
while (true) {
   while (count == 0) ;

   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   lock(lock_variable);
   count--;
   unlock(lock_variable);
}
```

The "lock variable" is a shared static variable. The lock() and unlock() calls are system calls that are promised to be "atomic"

7

The Critical Section Problem

A ***critical section*** is a section of code that is accesses data that is potentially being updated by code running in some other process or thread.

corresponding critical sections in different processes or threads are those that are accessing the same data.

IT IS VITAL THAT WHEN ANY PROCESS OR THREAD IS RUNNING IN A CRITICAL SECTION THAT NO OTHER PROCESS OR THREAD IS ALLOWED TO RUN IN ITS CORRESPONDING CRITICAL SECTION

8

The Critical Section Problem

We generally perceive threaded code as follows:

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Entry Section: The segment of code that implements the "request" to enter a critical section

Critical Section: That code that must be ran not in parallel with corresponding critical sections in other threads/processes.

Exit Section: The segment of code that implements the "notification" that a critical section as been left.

Remainder Section: The segment of code that can be ran in parallel with other threads / processes.

Again, "the problem" is that we need a mechanism that guarantees that no two threads / processes are simultaneously "in" corresponding critical sections.

9

The Critical Section Problem

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

We solve the problem by providing tools that allows a programmer to write correct entry and exit sections. We define correct solutions as those with these properties:

1. Mutual Exclusion: If process P1 is in its critical section, no other process may be in its corresponding critical section.
2. Progress: If no process is executing in its critical section, then only those processes NOT in their remainder sections may participate in who goes into the critical section next. The decision cannot be postponed indefinitely.
3. Bounded Waiting: There is a finite bound on how many times OTHER threads/processes are allowed to enter their critical sections before a process/thread is allowed to enter its own

10

The Critical Section Problem

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

There are multiple solutions to the critical section problem.

Software Solutions
- Peterson's Solution

Hardware Solutions (Support)
- Memory Barriers
- Hardware-Level Instructions

High-Level Constructs (Often these depend on hardware support)
- Mutex
- Semaphores
- Monitors

11

Peterson's Solution

This is the "classic" software solution. It is often used in textbooks to illustrate how one might solve the critical section problem and to illustrate how one might prove that the solution has the critical three properties.

However…. Changes in how we, as a society, build CPUs has rendered Peterson's Solution somewhat obsolete as its correctness cannot be guaranteed on modern hardware.

Like the book, we'll only consider Peterson's Solution to the extent that we can show why such purely software solutions to the critical section problem can be fraught with difficulties when attempted on modern hardware. This will set us up for what is actually done…

12

Peterson's Solution

```
int turn;
boolean flag[2];
```

There are TWO processes/threads. One called P_0 and the other called P_1

Both processes SHARE the variables turn and flag[]

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Both processes run the same code

in either of the two processes, `i` refers to the "PID" of the process that we are examining running the code and `j` refers to the "PID" of the OTHER process in the pair. For a process 0, `i == 0` and `j == 1` and for a process 1, `i == 1` and `j == 0`.

`turn` records whose turn it is to enter the critical section
`flag[x]` keeps track of if process `x` wants to enter its critical section

13

Peterson's Solution

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Presume that both threads/processes are running. Whenever EITHER of them get to the entry section, each will both try to:

1) Assert that it wants to get into its own critical section
2) Assert that it's "the other guy's turn" to enter its own critical section
3) Do a busy wait WHILE the "other guy wants in" and "it's the other guy's turn

Note, that even if both threads/processes want in at "about the same time", EACH only tries to give permission to the other guy only once. Inevitably, only ONE of them will "stick". Both of them may want to get in, but the "whose turn is it" can only be set one way…. So only one gets past the busy wait.

When a process leaves its critical section, it drops its "desire" to be in the critical section

14

Peterson's Solution

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

We'll leave the proofs for why Peterson's Solution provides the required properties for solution to the critical section problem to the book. They are pretty interesting, do read them.

HERE we will focus on an "unwritten assumption" in those proofs that didn't used to be broken "in the day" – but now is as a matter of course in modern architectures. This renders Peterson's Solution, and many like it, effectively non-operable and is why we need to ultimately rely on hardware support, often mediated through the kernel.

15

Peterson's Solution

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

We'll leave the proofs for why Peterson's Solution provides the required properties for solution to the critical section problem to the book. They are pretty interesting, do read them.

HERE we will focus on an "unwritten assumption" in those proofs that didn't used to be broken "in the day" – but now is as a matter of course in modern architectures. This renders Peterson's Solution, and many like it, effectively non-operable and is why we need to ultimately rely on hardware support, often mediated through the kernel.

16

Peterson's Solution – Whomp Whomp

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Modern processors ROUTINELY reorder memory accesses and updates on the fly to decrease access time and increase computational throughput.

At the hardware level, the only guarantee one gets is that the CPU won't reorder operations in a way that prevents what you asked for *eventually* being correct.

In effect, this means that, unless you take special steps (whatever that means), there is no promise that the ORDER you specify operations in your high level code is the ORDER of operations that the CPU actually uses when the code runs.

17

Peterson's Solution – Whomp Whomp

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Modern processors ROUTINELY reorder memory accesses and updates on the fly to decrease access time and increase computational throughput.

At the hardware level, the only guarantee one gets is that the CPU won't reorder operations in a way that prevents what you asked for *eventually* being correct.

In effect, this means that, unless you take special steps (whatever that means), there is no promise that the ORDER you specify operations in your high level code is the ORDER of operations that the CPU actually uses when the code runs.

18

Peterson's Solution – Whomp Whomp

```
int turn;
boolean flag[2];
```

```
// Thread One
while (true)
   { flag[i]=true;
     turn = j;
     while (flag[j] && turn == j)
        ;

     x = 100;

     flag[i]=false
   }
```

```
// Thread Two
while (true)
   { flag[i]=true;
     while (flag[j] && turn ==j)
       ;

     print(x);

     flag[i]=false;
   }
```

19

Peterson's Solution – Whomp Whomp

```
int turn;
boolean flag[2];
```

```
// Thread One
while (true)
   { flag[i]=true;
     turn = j;
     while (flag[j] && turn == j)
        ;

     x = 100;

     flag[i]=false
   }
```

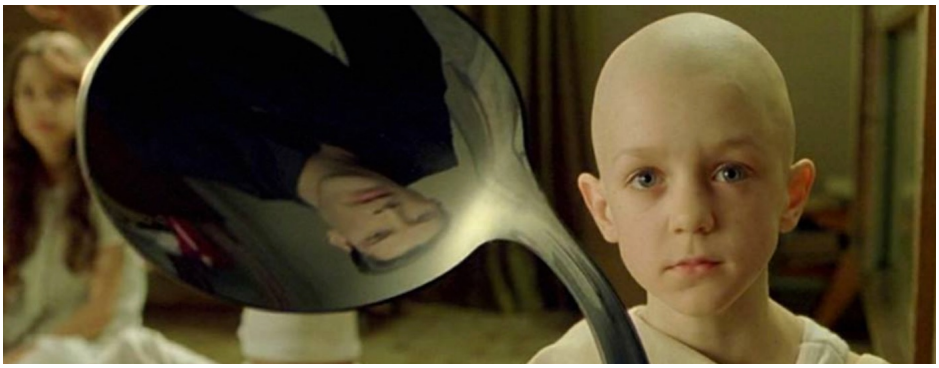
```
// Thread Two
while (true)
   { flag[i]=true;
     while (flag[j] && turn ==j)
       ;

     print(x);

     flag[i]=false;
   }
```

The CPU itself can decide to do the `x = 100;` and the `flag[i]=false` in the order you say OR in a different order if it thinks reordering it will make things faster due to caching concerns or other issues related to making memory access fast. In other words, the busy body CPU may or may not honor your ORDERING of memory operations. **Therefore, there exists a possibility that your critical section exclusion breaks because critical things get "reordered" outside the protected section!**

20

Peterson's Solution – Whomp Whomp

Ultimately, hardware problems require hardware solutions. We'll talk about those, but here's a good place to yet again talk about there not really being any spoons.



21

The Critical Section Problem

We generally perceive threaded code as follows:

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Entry Section: The segment of code that implements the "request" to enter a critical section

Critical Section: That code that must be ran not in parallel with corresponding critical sections in other threads/processes.

Exit Section: The segment of code that implements the "notification" that a critical section as been left.

Remainder Section: The segment of code that can be ran in parallel with other threads / processes.

Again, "the problem" is that we need a mechanism that guarantees that no two threads / processes are simultaneously "in" corresponding critical sections.

22

Mutex Locks

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

23

Mutex Locks

LOCK CONTENTION

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either ***high contention*** (a relatively large number of threads attempting to acquire the lock) or ***low contention*** (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

24

Mutex Locks

WHAT IS MEANT BY "SHORT DURATION"?

Spinlocks are often identified as the locking mechanism of choice on multi-processor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

25

Semaphores - Multiple Locking

```
S = number_of_resources;
while (true) {
        wait(S)
            critical section
        signal(S)
            remainder section
}
```

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

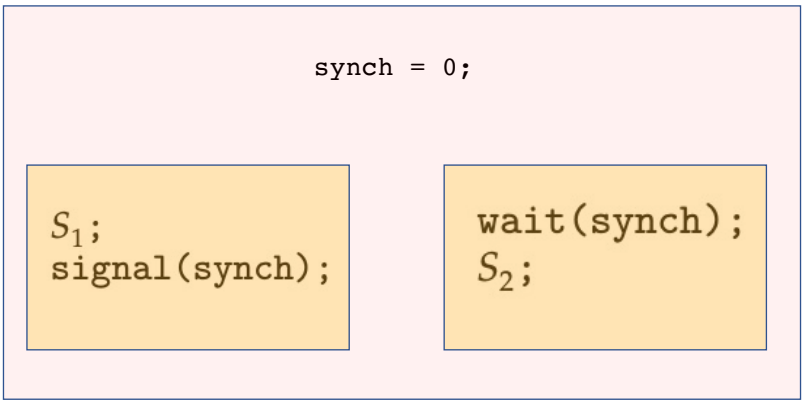
P(S)

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

V(S)

26

Semaphores – Synchronization



27

Semaphore and Mutex Implementation

Busy Wait:	We've seen this. Basically a spinlock.
Thread Sleep:	Each semaphore / mutex has a lock variable AND a list of threads/processes blocking on it. Waiting threads/processes are added to the list and put on the wait queue. When a lock variable / semaphore allows a pass, a thread/process is chosen from the list and woken up.
Hybrid:	A locked or waiting process/thread spins for a bit, and if it doesn't unlock during its spin allocation, it is put to sleep.

28

Readers-Writer Lock

In computer science, a **readers–writer** (**single-writer** lock,[1] a **multi-reader** lock,[2] a **push lock**,[3] or an **MRSW lock**) is a synchronization primitive that solves one of the readers–writers problems. An RW lock allows concurrent access for read-only operations, while write operations require exclusive access. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data. When a writer is writing the data, all other writers or readers will be blocked until the writer is finished writing. A common use might be to control access to a data structure in memory that cannot be updated atomically and is invalid (and should not be read by another thread) until the update is complete.

29

Readers-Writer Lock

Writer Starvation: Is it possible for a writer to NEVER be allowed to write? Why?

Read vs Write Priority: Are readers or writers considered more important? Does making one choice or the other affect lock contention?

30