Lecture 9
Synchronization

Concurrency

- Multiple processes/threads executing at the same time accessing a shared resource
  - Reading the same file/resource
  - Accessing shared memory

- Benefits of Concurrency
  - Speed
  - Economics

Concurrency

- **Concurrency** is the interleaving of processes in time to give the appearance of simultaneous execution.
  - Differs from parallelism, which offers genuine simultaneous execution.

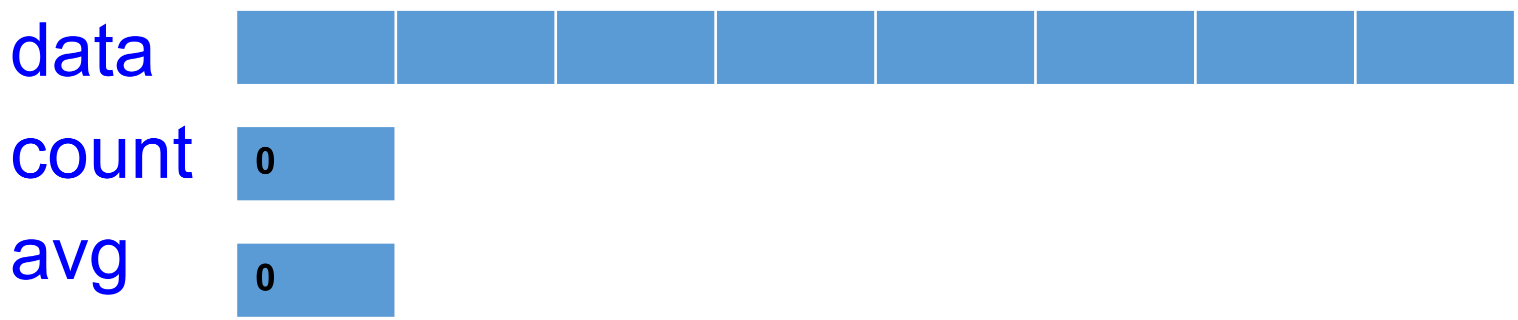
- **Parallelism** introduces the issue that different processors may run at different speeds
  - This problem is mirrored in concurrency as different processes progress at different rates

Condition for Concurrency

- Must give the same results as serial execution
- Using shared data requires synchronization

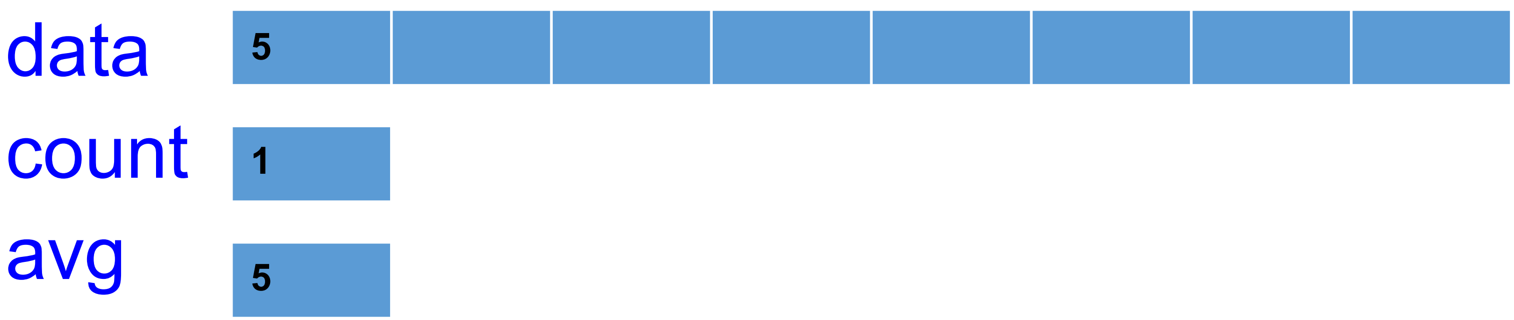
Example

Shared data/variables



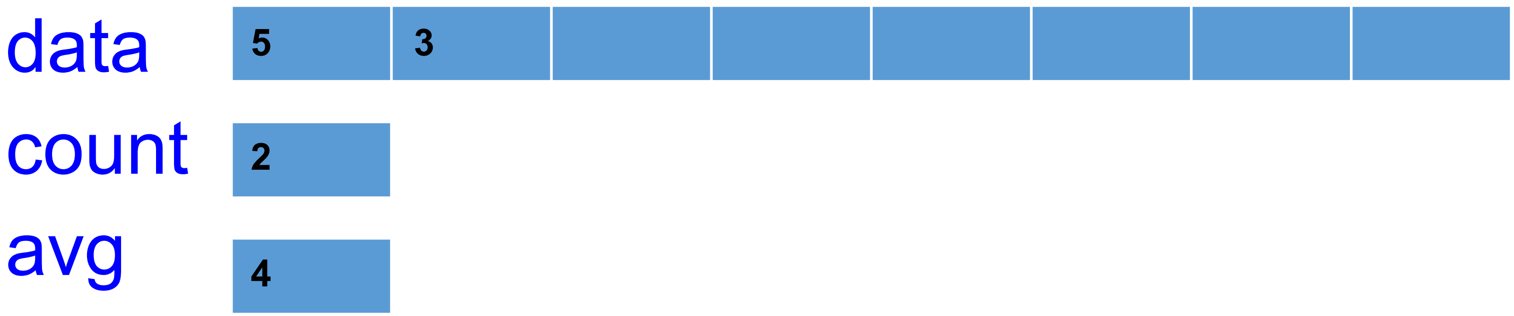
Example

Shared data/variables



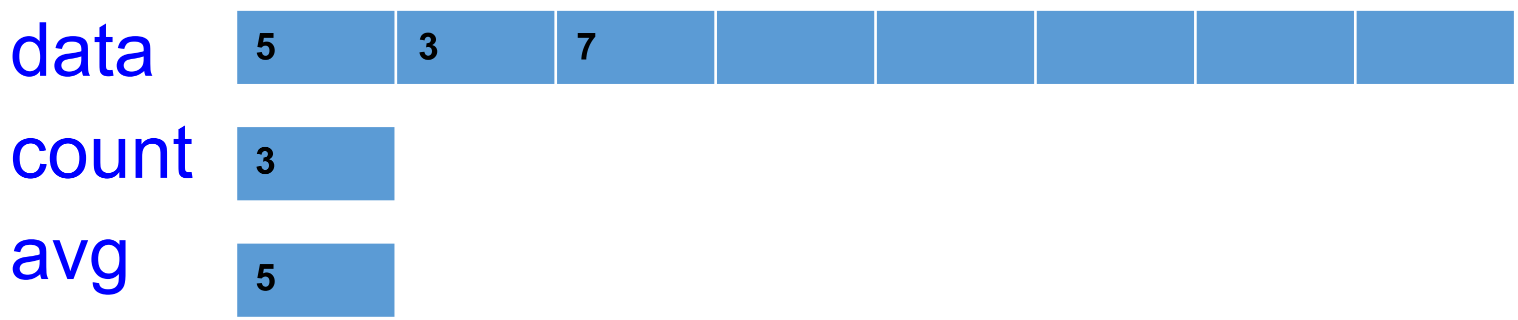
Example

Shared data/variables



Example

Shared data/variables



Serial Execution

```
// Waiting for data
while (true)
{
  v = get_value()
  add_new_value (v)
}
```

```
// Process data
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum + v) / count;
}
```

Concurrent Execution

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum  + v)/ count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum  + v)/ count;
}
```

Are we still going to have data consistency?

Correct values at all times in all conditions?

Concurrency

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum +v)/ count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = (sum +v)/ count;
}
```

Concurrency

More tricky issue:

C statements can compile into several machine language instructions

e.g. count++

```
mov R2, count
inc R2
mov count, R2
```

If these low-level instructions are *interleaved*, e.g. one process is preempted, and the other process is scheduled to run, then the results of the ***count*** value can be unpredictable

Concurrency

- Must give the same results as serial execution

- Using shared data requires synchronization

How can various mechanisms be used to ensure the **orderly execution of cooperating processes** that share address space so that data consistency is maintained?

Synchronization techniques:

- Mutex
- Semaphore
- Condition Variable
- Monitor

Race Condition

- Occurs in situations where:
  - Two or more processes (or threads) are accessing a shared resource
  - and the final result **depends on the order of instructions** are executed
- ***The part of the program where a shared resource*** is accessed is called *critical section*
- We need a mechanism to ***prohibit multiple processes from accessing a shared resource at the same time***

Critical Section

Process 1

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Process 2

```
while (true)
{
    v = get_value()
    add_new_value (v)
}
add_new_value (v) {
    int sum = avg * count + v;
    data[count] = v;
    count++;
    avg = sum / count;
}
```

Critical Section

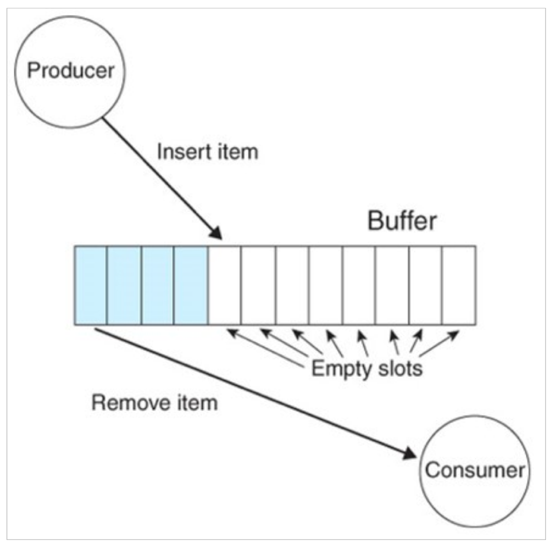
Where is the critical section in the ***add_new_value ()*** code?

Race Condition: Solution

- Solution must satisfy the following conditions:
  - **mutual exclusion**
    - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - **progress**
    - if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that wish to enter their critical sections can participate in the decision on which will enter its critical section next
    - this selection **cannot** be postponed indefinitely (OS must run a process, hence "progress")
  - **bounded waiting**
    - there exists a bound, or limit, on the number of times other processes can enter their critical sections after a process X has made a request to enter its critical section and before that request is granted **(no starvation)**

Producer-Consumer Problem (*Bounded Buffer Problem)*

- Two processes (producer and consumer) share a fixed size buffer
- Producer puts new information in the buffer
- Consumer takes out information from the buffer



Producer-Consumer Problem

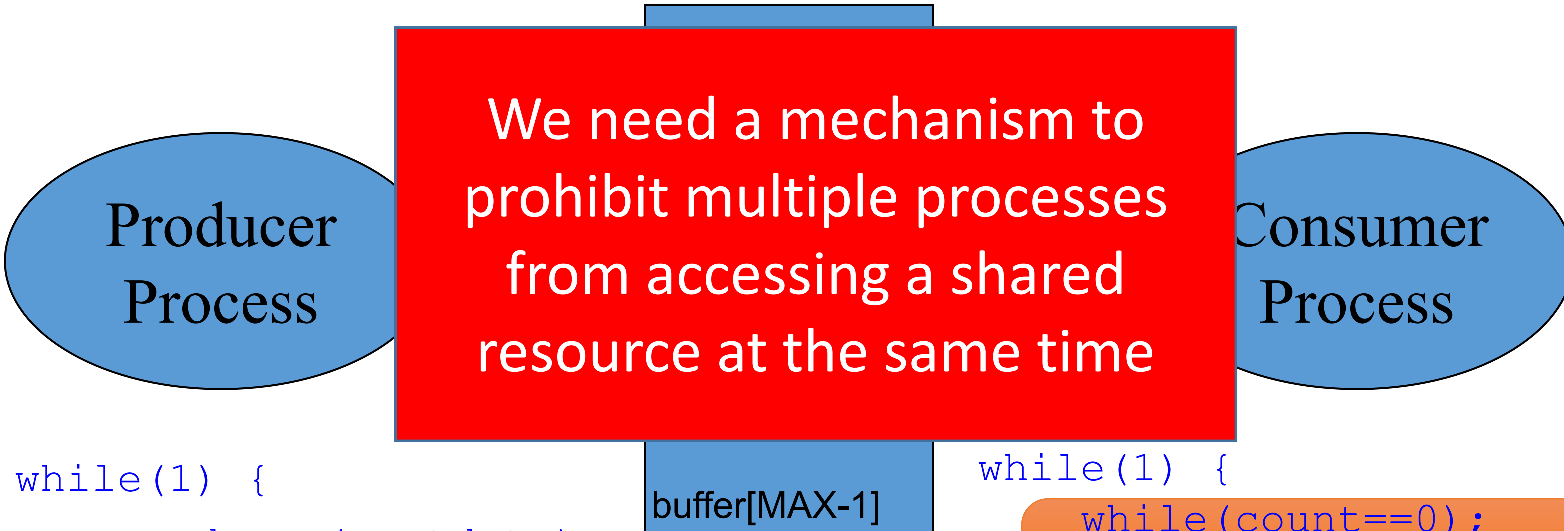
Shared data

data

count

Producer-Consumer Problem

Bounded Buffer




```
while(1) {
    produce (nextdata);
    while(count==MAX);
    buffer[count] = nextdata;
    count++;
}
```

Producer writes new data into buffer and increments counter

```
while(1) {
    while(count==0);
    getdata = buffer[count-1];
    count--;
    consume (getdata)
}
```

Consumer reads new data from buffer and decrements counter

Mutual Exclusion

- Mechanism to make sure no more than one process can execute in a critical section at any time
- How can we implement mutual exclusion?

Regular code

Entry critical section

Critical section

Access shared resource

Exit critical section

Regular code

Critical Section

// Producer

```
while(1) {
    produce (nextdata);
    while (count==MAX);
    Entry critical section
    buffer[count] = nextdata;
    count++;
    Exit critical section
}
```

// Consumer

```
while(1) {
    while (count==0);
    Entry critical section
    getdata = buffer[count-1];
    count--;
    Exit critical section
    consume (getdata)
}
```

Critical Section

Solution 1: Disabling interrupts

- Ensure that when a process is executing in its critical section, it cannot be preempted
- Disable all interrupts before entering a CS
- Enable all interrupts upon exiting the CS

```
shared int counter;
```

```
      producer code
disableInterrupts();
counter++;
enableInterrupts();
. . . remaining producer code
```

```
      consumer code
disableInterrupts();
counter--;
enableInterrupts();
. . . remaining consumer code
```

Solution 1: Disabling Interrupts

Problems:

- If a user forgets to enable interrupts???
- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; disabling interrupts blocks all processes
- Two or more CPUs???

Solution 2: **Software Only Solution**

```
shared boolean lock = FALSE;
shared int counter;
```

Code for producer

```
/* Acquire the lock */
  while(lock){ no_op;}(1)
  lock = TRUE; (3)

/* Execute critical
    section */
  counter++;

/* Release lock */
  lock = FALSE;
```

Code for consumer

```
/* Acquire the lock */
while(lock){ no_op;} (2)
lock = TRUE; (4)

/* Execute critical
      section */
   counter--;

/* Release lock */
lock = FALSE;
```

A flawed lock implementation:

Both processes may enter their critical section if there is a context switch just before the <lock = TRUE> statement

Solution 2: Software Only Solution

- Implementing mutual exclusion in software is not hard but it is not always work

- Need help from hardware

- Modern processors provide such support
  - Atomic test and set instruction
  - Atomic compare and swap instruction

Solution 2: Software Only Solution

- Peterson's solution:
  Restricted to only 2 processes.

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

```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- Turn indicates who will be run next
- Flag indicates who is ready to run next

- Need to prove:
  (1) Mutual exclusion is preserved
  (2) Progress is made
  (3) Bounded-waiting is met

Atomic Test-and-Set

- Need to be able to look at a variable and set it up to some value without being interrupted

  y = read (x); x = value;

- Modern computing systems provide such an instruction called *test-and-set (TS);*

```
boolean TS(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv; // returns original value of the target
}
```

- The entire sequence is a single instruction (atomic), implemented in hardware

Solution 3: Mutual exclusion using TS

```
shared boolean lock = FALSE;
shared int counter;
```

Code for p_1

```
/* Acquire the lock */
  while(TS(&lock)) ;

/* Execute critical section */
  counter++;

/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
while(TS(&lock)) ;

/* Execute critical section */
counter--;

/* Release lock */
lock = FALSE;
```

Atomic Test-and-Set

- The boolean TS () instruction is essentially a swap of values
- Mutual exclusion is achieved - no race conditions
  - If one process X tries to obtain the lock while another process Y already has it, X will wait in the loop
  - If a process is testing and/or setting the lock, no other process can interrupt it
- The system is exclusively occupied for only a short time - the time to test and set the lock, and not for entire critical section
- Don't have to disable and reenable interrupts
- Do you see any problems?
  - busy waiting.

```
while(TS(&lock)) ;
```

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

<u>Code for p_1</u>

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

<u>Code for p_2</u>

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

<u>Code for p_1</u>

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

<u>Code for p_2</u>

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

CPU is spinning & waiting

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

<u>Code for p_1</u>

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

<u>Code for p_2</u>

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

How can we eliminate the busy waiting?

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

<u>Code for p_1</u>

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

<u>Code for p_2</u>

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

Mutual exclusion using TS

shared boolean lock = FALSE;
shared int count;
Shared data_type buffer [MAX];

Code for p_1

```
while(1) {
    produce (nextdata)
    while(count==MAX);
    Acquire(lock);
    buffer[count] = nextdata;
    count++;
    Release(lock);
}
```

Code for p_2

```
while (1) {
    while(count==0);
    Acquire(lock);
    data = buffer[count-1];
    count--;
    Release(lock);
    consume(data);
}
```

- Look first at the busy wait when there is no room for data or no data
    - Need a way to pause a process/thread until the space is available or data is available.

Mutual Exclusion

- How can we eliminate the busy waiting?
- Need a way to pause a process/thread until the lock is available

sleep( ) and **wakeup( )** primitives

- *sleep()*: causes a running process to block

- *wakeup(pid)*: causes the process whose id is *pid* to move to ready state
    - No effect if process *pid* is not blocked

```
// producer – place data into buffer
while(1) {
   if (counter==MAX) sleep();
   buffer[in] = nextdata;
   in = (in+1) % MAX;
   counter++;
   if (counter == 1) wakeup (p2);
}

// consumer – take data out of buffer
while(1) {
   if (counter==0) sleep();
   getdata = buffer[out];
   out = (out+1) % MAX;
   counter--;
   if (counter == MAX – 1) wakeup (p1);
}
```

sleep( ) and wakeup( ) primitives

- Problem with counter++ and counter-- still exist
  - Can be solved using TS but it has busy waiting
- Possible problem with order of execution:
  - Consumer reads counter and counter == 0
  - Scheduler schedules the producer
  - Producer puts an item in the buffer and signals the consumer to wake up
    - Since consumer has not yet invoked sleep(), the wakeup() invocation by the producer has no effect
  - Consumer is scheduled, and it blocks
  - Eventually, producer fills up the buffer and blocks
  - How can we solve this problem?
    - Need a mechanism to count the number of sleep() and wakeup() invocations