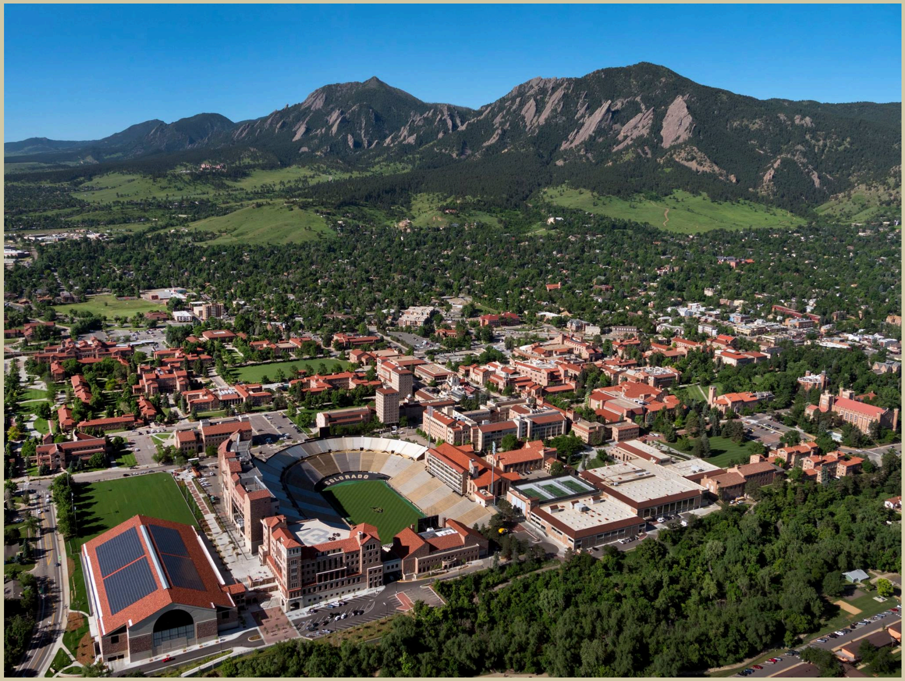
Design and Analysis of Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado Boulder

Synchronization



Design and Analysis of
Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado
Boulder

Concurrency

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

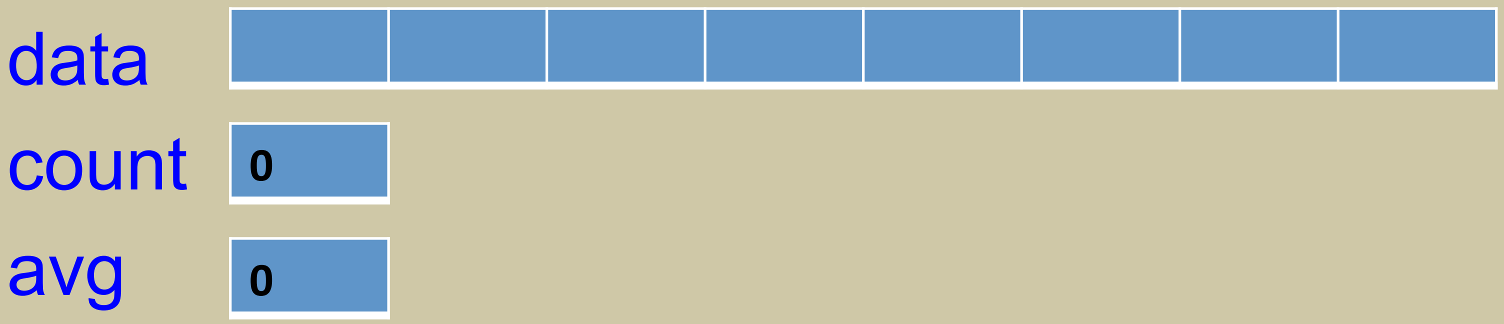
- Parallelism introduces the issue that different processors may run at different speeds
  - This problem is mirrored in concurrency as different processes progress at different rates

Concurrency

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

Concurrency

Shared data



Concurrency

```
while (true)
{
   v = get_value()
   add_new_value (v)
}
```

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

Concurrency

Process 1

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

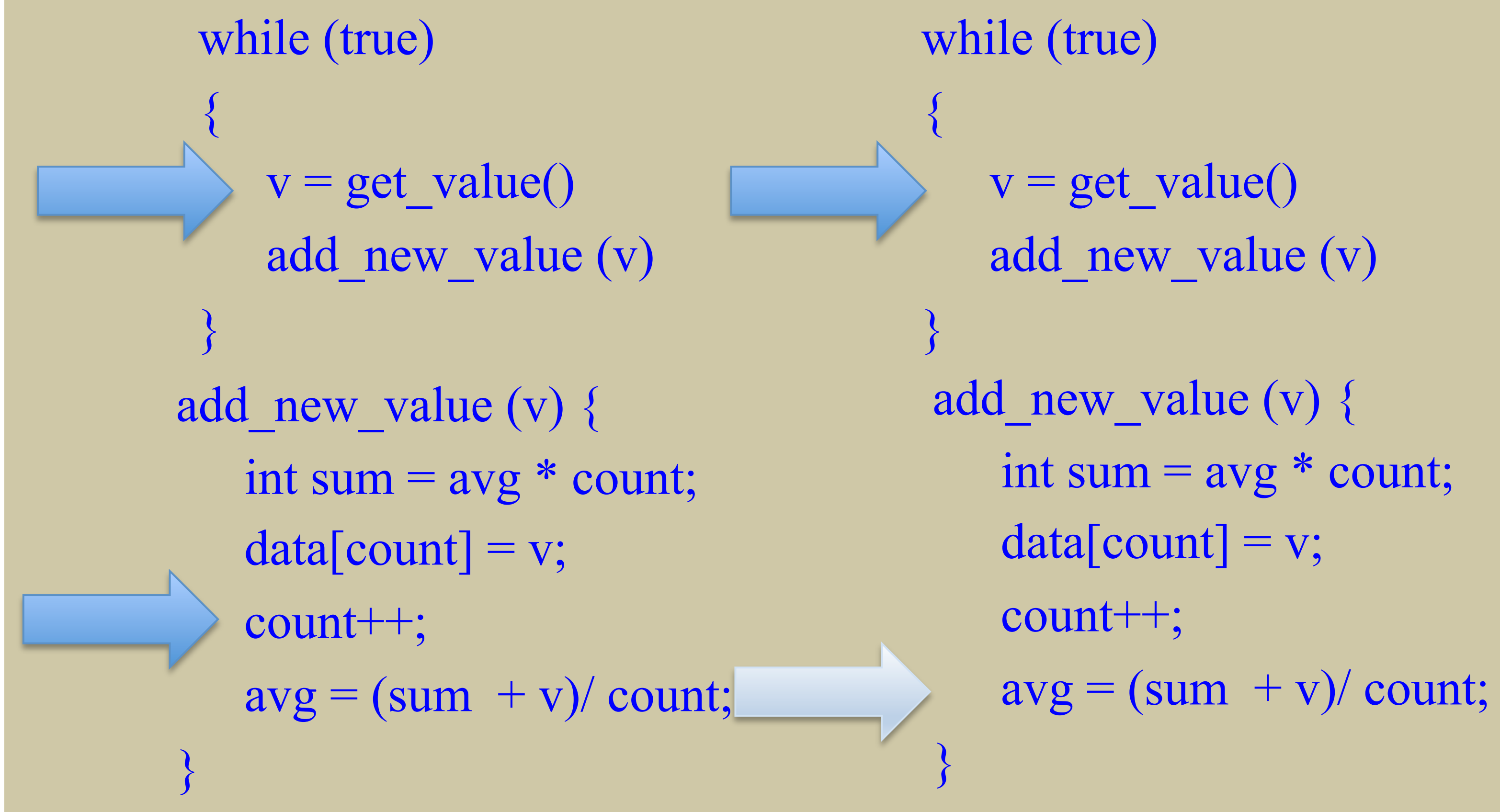
Process 2

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

Concurrency

C statements can compile into several machine language instructions

e.g. count++

```
reg2 = count;          mov R2, count
reg2 = reg2 + 1;       inc R2
count = reg2;          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

- Suppose we have the following sequence of interleaving.
- Let count = 5 initially.

// P1 - counter++	// P2 - counter++
(1) reg1 = count;	(2) reg2 = count;
(3) reg1 = reg1 + 1;	(4) reg2 = reg2 + 1;
(5) counter = reg1;	(6) count = reg2;

		1	2	3	4	5	6
count	5	5	5	5	5	6	6
reg1	?	5	5	6	6	6	6
reg2	?	?	5	5	6	6	6

RACE CONDITION

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

- Critical Section
- Mutex
- Semaphore
- Condition Variable
- Monitor

Race Condition

- Occurs in situations where:
  - two are 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*

Critical Section

Process 1

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

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

Producer-Consumer Problem

- Also known generically as a
  Bounded Buffer Problem

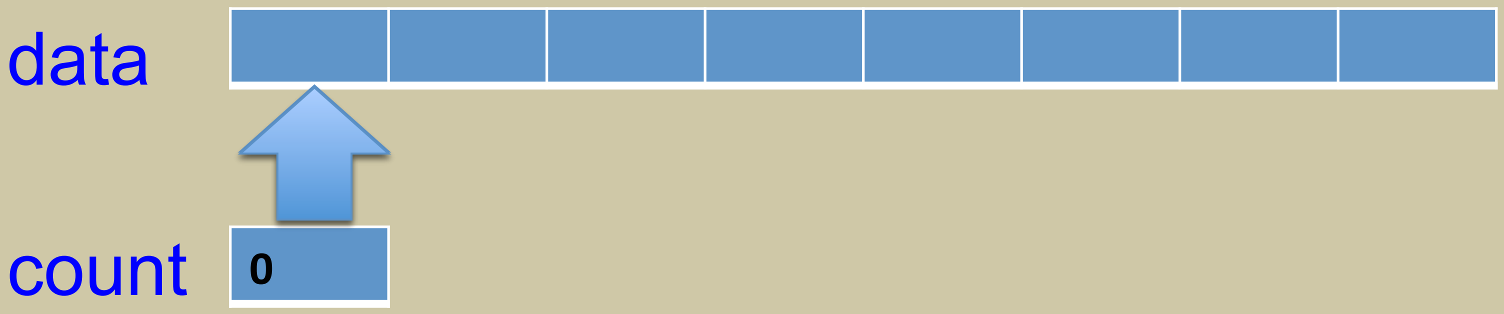
- Two processes (producer and consumer) share a fixed size buffer

- Producer puts new information in the buffer

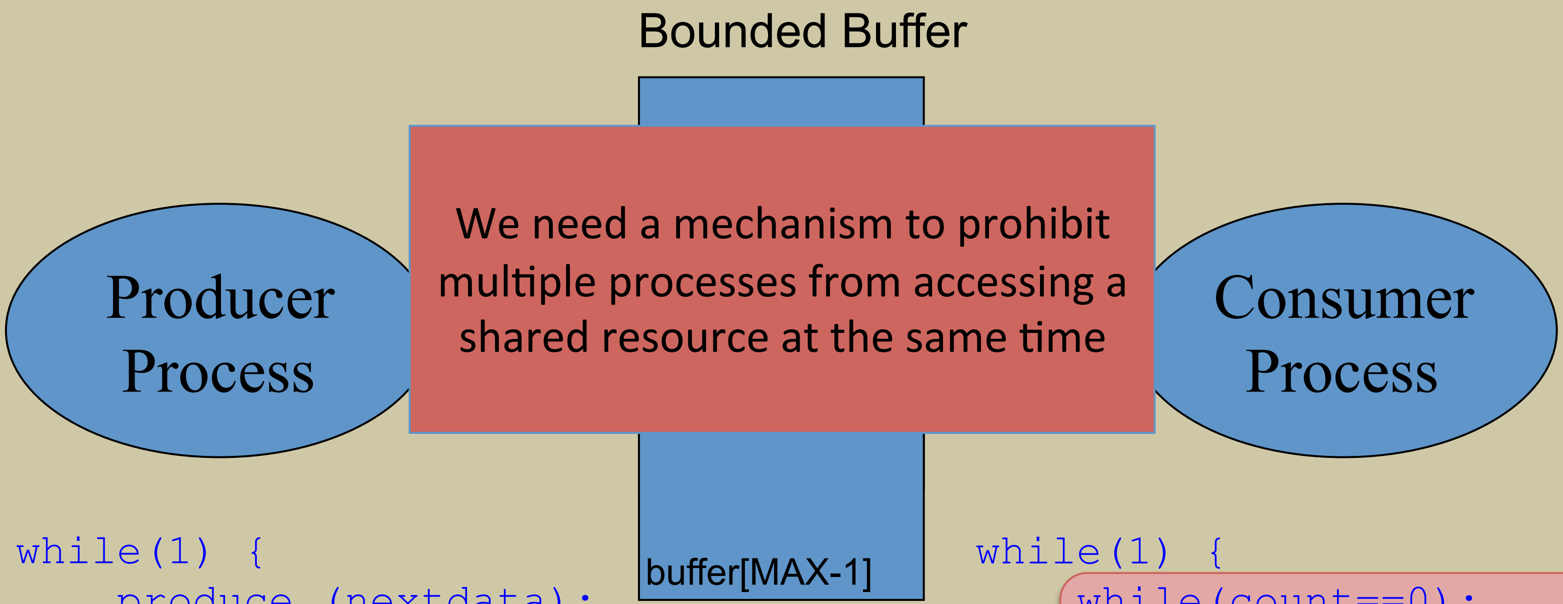
- Consumer takes out information from the buffer

Producer-Consumer Problem

Shared data



Producer-Consumer Problem



We need a mechanism to prohibit multiple processes from accessing a shared resource at the same time

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

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

Producer writes new data into buffer and increments counter

Consumer reads new data from buffer and decrements counter

Mutual Exclusion

- 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

Race Condition: Solution

- A 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)

How can we guarantee that only one process is executing within a critical section at any one time?

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                       consumer code
disableInterrupts();                 disableInterrupts();
counter++;                              counter--;
enableInterrupts();                  enableInterrupts();
. . . remaining producer code        . . . 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 extremely difficult

- *Read Section 5.3 for a software only solution*

- Need help from hardware
- Modern processors provide such support
    - Atomic test and set instruction
    - Atomic compare and swap instruction

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;
```

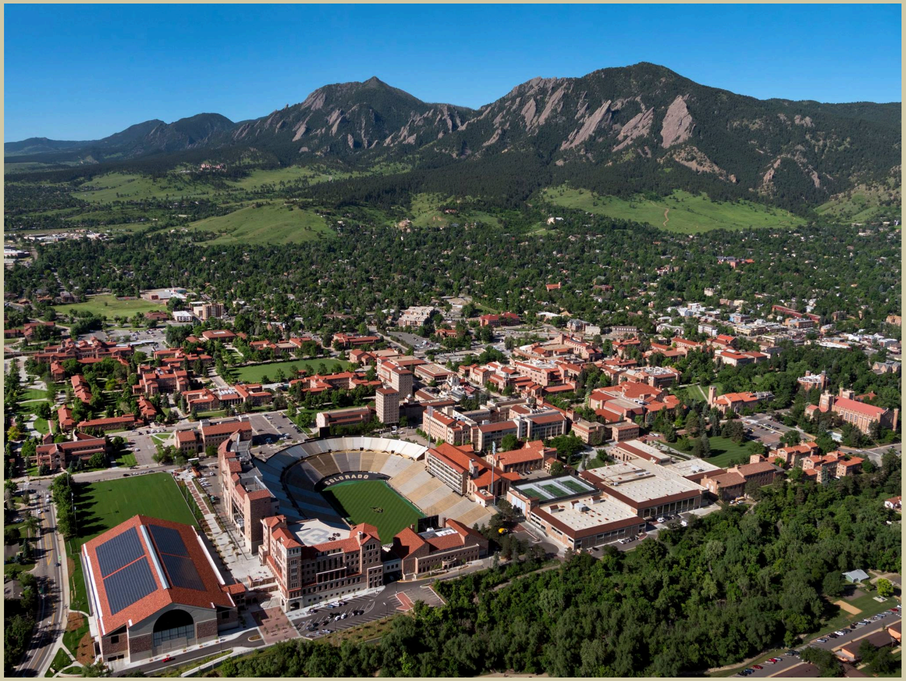
Code for p_2

```
    /* 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
  - The x86 CPU instruction set contains atomic instructions
  - Can use atomic XCHG to implement spinlocks
- Mutual exclusion is achieved - no race conditions
  - While another process Y already has it, X will wait in the loop to obtain it
  - When 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 disadvantages?
  - busy waiting. `while(TS(&lock)) ;`

Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox

University of Colorado
Boulder