# Chapter 5: Mutual Exclusion and Semaphores

CSCI 3753 Operating Systems

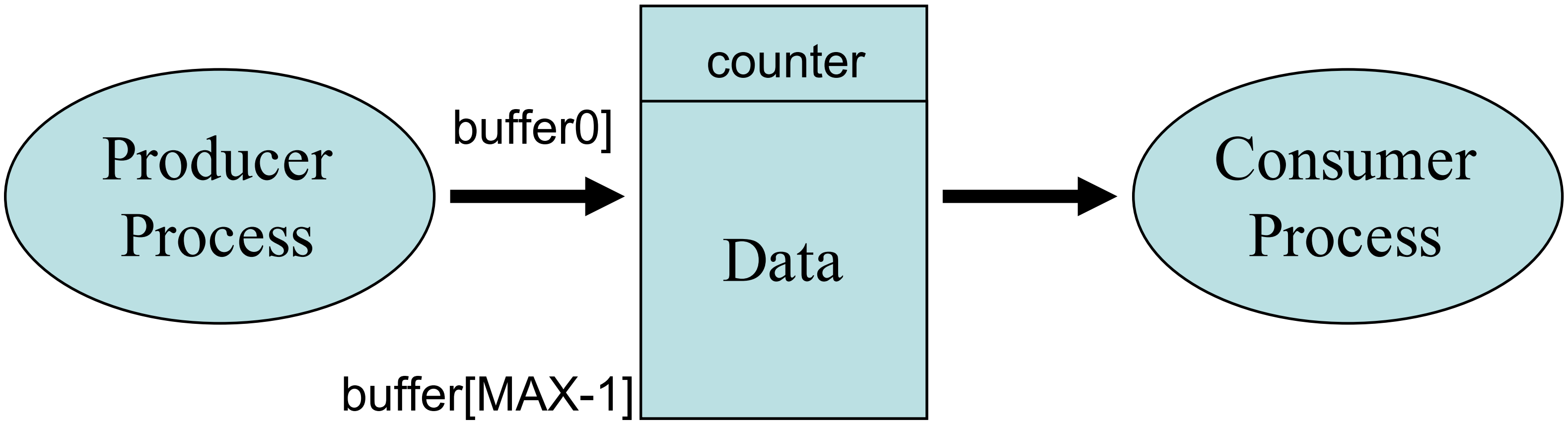
William Mortl, MS and Rick Han, PhD
University of Colorado at Boulder

# Recap

- Inter-Process Communication (IPC)
  - Shared Memory
  - Message Passing
    - Unix-domain sockets, Internet sockets, pipes, signals
- Chapter 5: Synchronization
  - Race conditions occur when 2 processes/threads try to access the same shared variable
  - Example

# Synchronization

## Bounded Buffer




```
while(1) {
    while(counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
}
```

```
while(1) {
    while(counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
}
```

Producer writes new data into buffer and increments counter

counter updates can conflict!

Consumer reads new data from buffer and decrements counter

# A Race Condition Example

- Let brackets [value] denote local value of counter in either the producer or consumer's process. counter=5 initially.

// counter++

```
(1)  reg1 = counter;     [5]
(3)  reg1 = reg1 + 1;    [6]
(5)  counter = reg1;     [6]
```

// counter--;

```
(2)  reg2 = counter;     [5]
(4)  reg2 = reg2 - 1;    [4]
(6)  counter = reg2;     [4]
```

- Counter should be 5 with 1 producer and 1 consumer, but counter = 4! Reversing steps (5) and (6) sets counter=6
- Undesirable and unpredictable *race condition*
- Basic Problem: unprotected access to a shared variable (counter)

# Critical Section

- Some kernel data structures could be subject to race conditions, e.g. access to list of open files
- Kernel developer must ensure that no such race conditions occur
- User or kernel developer identifies *critical sections* in code where each process accesses shared variables
  - access to critical sections is controlled by special *entry* and *exit* code

```
while(1) {

  entry section

    critical section (manipulate common var's)

  exit section

    remainder section code

}
```

# Critical Sections Need Mutual Exclusion

- How do we protect access to critical sections?
  - want to prevent another process from executing while current process is modifying this variable - *mutual exclusion*
- Approaches to mutual exclusion:
  - User space – see Peterson’s solution Chapter 5.2
  - Hardware-based - disable interrupts
    - before entering critical section, disable interrupts
    - after exiting critical section, reenable interrupts
    - This provides mutual exclusion

# Disabling Interrupts

```
shared int counter;
```

Code for $p_1$

```
disableInterrupts();
counter++;
enableInterrupts();
```

Code for $p_2$

```
disableInterrupts();
counter--;
enableInterrupts();
```

Achieves mutual exclusion:

- While P1 is in its critical section, P2 cannot enter, and vice versa
- Only when the first process is finished by enabling interrupts can another process attempt to enter the critical section
- modify the shared counter



example from Pearson slides

# Disabling Interrupts (2)

```
shared int counter;
```

Code for $p_1$

```
disableInterrupts();
counter++;
enableInterrupts();
```

Code for $p_2$

```
disableInterrupts();
counter--;
enableInterrupts();
```

Drawbacks?

- Interrupts could be disabled indefinitely
  - Due to a bug, e.g. user forgets to reenable interrupts
  - Unintentionally due to a loop



example from Pearson slides

# Disabling Interrupts (3)

```
shared int counter;
```

<u>Code for $p_1$</u>

```
disableInterrupts();
counter++;
enableInterrupts();
```

<u>Code for $p_2$</u>

```
disableInterrupts();
counter--;
enableInterrupts();
```

Drawbacks?

- Interrupts could be disabled a long time
  - Interrupts can be disabled too long, e.g. if the critical section has many lines of code
  - Can prevent useful I/O from being processed
  - Can prevent useful progress on other processes



example from Pearson slides

# Locks

- Goal: achieve mutual exclusion without the drawbacks of fully disabling interrupts
- Create a variable/flag called a *lock* to
  - The lock's value indicates whether the critical section (or access to its protected variables) is busy
  - Unset the lock or flag when the critical section is done
- Locks are often called *mutexes*

# Lock Semantics

- Each task calls `Acquire(lock)` before entering its critical section.
  - If a task successfully grabs the lock, this task can proceed to execute its critical section
  - If the lock is not available, the calling task blocks. So only 1 task at a time has the lock.
    - Multiple tasks can block waiting on a lock, forming a FIFO queue
- A task calls `Release(lock)` when done, so other processes can enter their critical sections too
  - 1 waiting task is woken (based on order) at each lock release

# A Lock Example

```
shared int counter;
shared Lock lock;
```

Code for $p_1$

```
Acquire(lock);
...
counter++;
...
Release(lock);
```

Code for $p_2$

```
Acquire(lock);
...
counter--;
...
Release(lock);
```

Advantage: Achieves mutual exclusion without blocking all processes

- Other tasks are blocked from executing their critical sections
- But if the owner of the lock is context-switched out, other tasks can keep executing (provided they have no interest in the lock)



example from Pearson slides

# A Lock Example

```
shared int counter;
shared Lock lock;
```

Code for $p_1$

```
Acquire(lock);
...
counter++;
...
Release(lock);
```

Code for $p_2$

```
Acquire(lock);
...
counter--;
...
Release(lock);
```

- Lock-based solutions only work if all processes participate and surround critical sections with entry and exit code to synchronize access to shared data
- Acquire and Release implemented as system calls



example from Pearson slides

# Locks

- Additional advantages:
  - user doesn't have control over disabling/reenabling interrupts – the OS does this.
  - user doesn't have to remember to reenable interrupts
  - So tasks with I/O that need to process interrupts can make progress
- Disadvantage:
  - user still must remember to release(lock), otherwise other tasks blocked on the lock will wait forever

# A Flawed Lock Implementation

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

<u>Code for $p_1$</u>

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

/* Execute critical
      section */
  counter++;

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

<u>Code for $p_2$</u>

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

/* Execute critical
      section */
   counter--;

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

Testing of the lock using a while() is subject to a race condition



example from Pearson slides

# A Flawed Lock Implementation

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

<u>Code for $p_1$</u>

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

/* Execute critical
      section */
  counter++;

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

- If P1 has just completed the *test* "while(lock)" and is just before the *setting* of "lock=TRUE", P1 can get context switched out
- Then P2 executes and sees lock=FALSE in its while(lock) test, and also drops through to the lock=TRUE statement
- Now both P1 and P2 are executing in critical section - BAD



example from Pearson slides

# A Flawed Lock Implementation

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

<u>Code for $p_1$</u>

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

/* Execute critical
        section */
  counter++;

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

- The problem: between *testing* the lock and *setting* the lock to FALSE, a task can be context-switched out
- Instead, want an uninterruptible or *atomic* test-and-set operation
  - Achieve by disabling interrupts
  - Achieved by hardware test-and-set instruction TS



example from Pearson slides

# A Correct Lock Implementation

```
Acquire(lock) {
  disableInterrupts();

/* Wait for lock */
  while(lock) {
     enableInterrupts();
     disableInterrupts();
  }
  lock = TRUE;
  enableInterrupts();
}
```

```
Release(lock) {
  disableInterrupts();
  lock = FALSE;
  enableInterrupts();
}
```

- Advantage: no race condition as test-and-set is atomic
- Disadvantage:
  - busy waiting to acquire lock (A busy waiting type of lock is also called a spinlock)
  - task is stuck in Acquire() and can't do other work



example from Pearson slides

# Test-and-Set Instruction

- CPU hardware provides an uninterruptible hardware instruction `TS`,
  - which atomically performs *both* the test and the set operations
  - called by the TestandSet() system call
- the boolean TestandSet() operation in the next slide is essentially a swap of values
  - The x86 CPU instruction set contains atomic instructions such as XCHG that are essentially swap statements
  - Can use atomic XCHG to implement spinlocks

# A Test-and-Set Lock Implementation

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

**Code for $p_1$**

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

/* Execute crit sect */
  counter++;

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

**Code for $p_2$**

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

/* Execute crit sect */
  counter--;

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

---

```
boolean TestandSet(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

This entire instruction (sequence) is atomic. The hardware enforces atomicity.



example from Pearson slides

# A Test-and-Set Lock Implementation

- 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
- Efficient:
  - The system is exclusively occupied for only a short time - the time to test and set the lock, and not for entire critical section
  - typically only about 10 instructions

# A Test-and-Set Lock Implementation

- Advantage:
  - Compared to spinning inside the Acquire() system call, this approach spins in user space
    - Allows the user task to do other execution while polling the TestandSet()
- Disadvantage:
  - requires user to remember to put in while()
  - while() means busy-waiting, i.e. this is a form of spinlock too except in user space
  - requires special hardware support in the form of a low level atomic machine instruction TS

# Blocking on Locks

- Simplicity of spinlocks:
  - don't require any other OS constructs, such as a queue of tasks blocked on the lock
  - If multiple tasks are blocked on the lock, they all spin
  - Can be useful if blocking time is short
- Locks can be augmented with queues to hold blocked tasks
  - OS scheduler must be involved
  - Useful when blocking time is long
  - We'll see an example of this with semaphores

# Semaphores

- more general solution to mutual exclusion proposed by Dijkstra
- Semaphore S is an integer variable that, apart from initialization, is accessed only through 2 standard atomic operations
  - P(), also called wait()
    - somewhat equivalent to a test-and-set, but also *decrements* the value of S
  - V(), also called signal()
    - *increments* the value of S
  - OS provides ways to create and manipulate semaphores atomically

# Semaphores

- Pseudo-code for classic semaphore
  - its value can't go below zero, i.e. classic semaphore is non-negative

“atomic”

```
P(S) {
    while(S<=0)
        {wait or no op;}
    S--;
}
```

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

atomic

P() is “atomic” in the sense that it tests S atomically, and if S<=0, then the process calling P() will *relinquish control*. Otherwise, the process continues forward and decrements S atomically.

# A Semaphore Implementation

- Based only disabling and reenabling of interrupts
  - semaphores can also be implemented using TestandSet() instructions

```
P(S) {
    disableInterrupts();
    while(S<=0) {
        enableInt();
        disableInt();
    }
    S--;
    enableInt()
}
```

```
V(S) {
    disableInt();
    S++;
    enableInt()
}
```

disable interrupts because S++ might cause race condition

Also a form of spinlock
S not decremented unless it is positive

# Mutual Exclusion with Semaphores

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in code
```

Process P1:

```
P(S);
   // execute crit sect
    counter++;
V(S);
```

Process P2:

```
P(S);
   // execute crit sect
    counter--;
V(S);
```

- Both processes atomically P() and V() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable counter

# Mutual Exclusion with Semaphores (2)

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in code
```

Process P1:

```
P(S);
   // execute crit sect
    counter++;
V(S);
```

Process P2:

```
P(S);
   // execute crit sect
    counter--;
V(S);
```

- The 1st process that calls P() on semaphore S will check if S==0 (it is not), and then will atomically decrement S to 0
- The next process calling P(S) blocks on semaphore S since S=0, hence mutual exclusion is achieved
- When 1st process is done, it calls V(S), atomically incrementing S to 1

# Mutual Exclusion with Semaphores (3)

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in code
```

Process P1:

```
P(S);
   // execute crit sect
    counter++;
V(S);
```

Process P2:

```
P(S);
   // execute crit sect
    counter--;
V(S);
```

- After a V(S), all waiting processes are busy-waiting, so there is a race to see which one calls P(S) first

# Binary Semaphores

- The previous example showed how to use the semaphore as a *binary* semaphore
  - its initial value was set to 1
  - a binary semaphore is also called a *mutex lock*, i.e. it can be used to provide mutual exclusion on some piece of critical code
  - Let's define a binary semaphore as a semaphore whose value can't exceed 1, even if many processes keep signal()'ing the semaphore
    - Additional logic would have to be added to the counting semaphore in its V()/signal() function to cap its maximum value to 1

# Counting Semaphores

- A semaphore can also be used more generally as a *counting* semaphore
  - its initial value is n, e.g. n=10
  - the value of the semaphore could be used to keep track of the number of instances of a finite resource that are still available
    - We'll see an example later using counting semaphores to solve the general producer/consumer bounded buffer problem

# Enforcing Order with Semaphores

- Enforce *order* of access between 2 processes P1 and P2
  - P1 contains code C1 and P2 contains code C2
  - Want to ensure that code C1 executes before code C2
  - Use semaphores to synchronize the order of execution of the two processes

**`Semaphore S=0;`** // initial value of semaphore = 0

Process P1:

```
C1;        // execute C1
signal(S); // V() semaphore
```

Process P2:

```
wait(S);  // P() semaphore
C2;       // execute C2
```

# Enforcing Order with Semaphores (2)

- if P1 executes 1st, then
  - C1 will execute 1st, then P1 V()'s semaphore, adding 1 to its value
  - Later, when P2 executes, it will call wait(S), which will decrement the semaphore to 0 – no waiting - followed by execution of C2
  - Thus C1 executes before C2

**`Semaphore S=0;`** // initial value of semaphore = 0

Process P1:

```
C1;         // execute C1
signal(S); // V() semaphore
```

Process P2:

```
wait(S);  // P() semaphore
C2;       // execute C2
```

# Enforcing Order with Semaphores (3)

- If P2 executes 1st, then
  - P2 blocks on semaphore (=0), so C2 will not be executed yet
  - Later, when P1 executes, it runs C1, then V()'s the semaphore
  - This awakens P2, which then executes C2
  - Thus C1 executes before C2

**`Semaphore S=0;`** // initial value of semaphore = 0

Process P1:

```
C1;        // execute C1
signal(S); // V() semaphore
```

Process P2:

```
wait(S);  // P() semaphore
C2;       // execute C2
```

# A Revised Semaphore Definition

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

atomic

```
P(semaphore *S) {
    S->value--;
    if (S->value<0) {
        add this process
           to S->list;
        block();
    }
}
```

atomic

```
V(semaphore *S) {
    S->value++;
    if (S->value<=0) {
        remove a process P
           from S->list;
        wakeup(P);
    }
}
```

- Efficiently sleep the process until it needs to be woken up by a V()/signal(), rather than spinlock

# A Revised Semaphore Definition

- New definition allows a semaphore's value to be negative, because the decrement occurs before the test in P()
  - The absolute value of the semaphore's negative amount can be used to indicate the # of processes blocked on the semaphore
- Processes now yield the CPU if the semaphore's value is negative, rather than busy wait
  - If more than one process is blocked on a semaphore, then use a FIFO queue to select the next process to wake up when a semaphore is V'ed
    - Why is LIFO to be avoided?

# Mutual Exclusion with Revised Semaphore

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in
                    //     code
```

Process P1:

```
P(S);
counter++;
V(S);
```

Process P2:

```
P(S);
counter--;
V(S);
```

Process P3:

```
P(S);
counter*=2;
V(S);
```

- The 1st process (say P2) that calls P() on semaphore S will set S==0 and move into its critical section
- The next process (say P3) calling P(S) decrements S to -1 and blocks on semaphore, hence mutual exclusion is achieved
- The next process (say P1) calling P(S) decrements S to -2 and blocks on S – *Note how |S| = # blocked processes!*

# Mutual Exclusion with Revised Semaphore

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in
                    //    code
```

Process P1:

```
P(S);
counter++;
V(S);
```

Process P2:

```
P(S);
counter--;
V(S);
```

Process P3:

```
P(S);
counter*=2;
V(S);
```

- P2 finishes, V()'s the semaphore S, increasing its value to -1, signaling a process to unblock (P3).
- P3 finishes, V()'s the semaphore S, increasing its value to 0, causing a process to unblock (P1)
- P1 finishes, V()'s the semaphore S, increasing its value to original value of 1. No more processes to unblock.

# Enforcing Order with Revised Semaphore

- If P1 hits its signal(S) first, before P2 hits wait(S), then
  - P1 will have executed C1 already, will then increment S to 1, and no process will be unblocked.
  - Later, P2 calls wait(S), decrements S from 1 to 0, and executes C2. Order is preserved: C1 executes before C2

**`Semaphore S=0;`** // initial value of semaphore = 0

Process P1:

```
C1;          // execute C1
signal(S);   // V() semaphore
```

Process P2:

```
wait(S);   // P() semaphore
C2;        // execute C2
```

# Enforcing Order with Revised Semaphore

- If P2 hits wait(S) before P1 hits signal(S), then
  - P2 will decrement S to -1, and block P2 on the semaphore.
  - Next, P1 calls signal(S) having executed code C1, incrementing S from -1 to 0 and unblocking P2.
  - P2 now executes C2. Order is preserved: C1 executes before C2

**`Semaphore S=0;`** // initial value of semaphore = 0

Process P1:

```
C1;         // execute C1
signal(S); // V() semaphore
```

Process P2:

```
wait(S);   // P() semaphore
C2;        // execute C2
```

# Deadlock

- Semaphores provide synchronization, but can introduce more complicated higher level problems like *deadlock*
  - two processes deadlock when each wants a resource that has been locked by the other process
  - e.g. P1 wants resource R2 locked by process P2 with semaphore S2, while P2 wants resource R1 locked by process P1 with semaphore S1

# Deadlock Example

```
Semaphore Q= 1;     // binary semaphore as a mutex lock
Semaphore S = 1;    // binary semaphore as a mutex lock
variable R1, R2;
```

*Process P1:*

```
P(S);                    (step 1)
P(Q);                    (step 3)
     modify R1 and R2;
V(S);
V(Q);
```

*Process P2:*

```
(step 2) P(Q);
(step 4) P(S);
Deadlock!     modify R1 and R2;
         V(Q);
         V(S);
```

If steps (1) through (4) are executed in that order, then P1 and P2 will be deadlocked after statement (4) - verify this for yourself by stepping thru the semaphore values

# Deadlock

- In the previous example,
  - Each process will sleep on the other process's semaphore
  - the V() signalling statements will never get executed, so there is no way to wake up the two processes from within those two processes
  - there is no rule prohibiting an application programmer from P()'ing Q before S, or vice versa - the application programmer won't have enough information to decide on the proper order
  - in general, with N processes sharing N semaphores, the potential for deadlock grows

# Other Deadlock Examples

- A programmer mistakenly follows a P() with a second P() instead of a V(), e.g.

  P(mutex)

      critical section

  P(mutex)

  – This causes a self-deadlock!

- A programmer forgets and omits V(mutex). Can cause deadlock.

| P1: | P2: |
|---|---|
| *P(mutex)* | P(mutex) |
| critical section 1 | critical section 2 |
| V(mutex) | |

  – P2 calls P(mutex) and executes crit sect 2. Then P1 blocks on P(mutex), then P2 blocks on P(mutex)

# Other Synchronization Errors

- A programmer forgets and omits P(mutex). Can violate mutual exclusion if P(mutex) is omitted.

| P1: | P2: |
| --- | --- |
| *P(mutex)* | |
| critical section 1 | critical section 2 |
| *V(mutex)* | V(mutex) |

- A programmer reverses the order of P() and V(), e.g.

```
V(mutex)
    critical section    <---- this violates mutual exclusion,
P(mutex)                      but is not deadlock
```