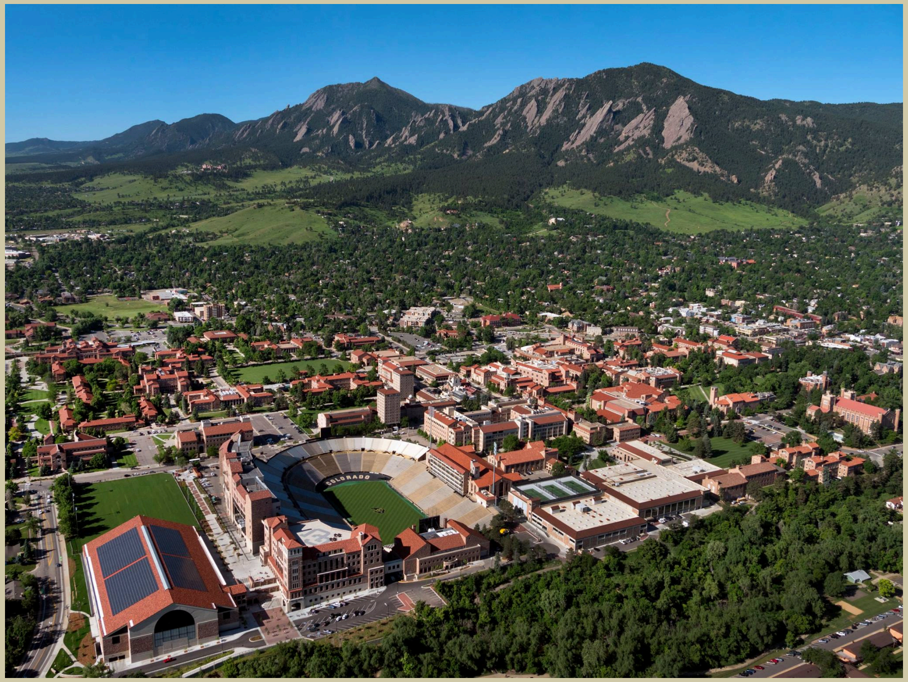
Design and Analysis of Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado Boulder

Mutexes and Semaphores



Design and Analysis of
Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado
Boulder

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

How can we eliminate the busy 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);
}
```

- 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 the data 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

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

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

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

Our solution using sleep( ) and wakeup( )

- Still a problem with mutual exclusion of the critical sections containing counter++ and counter-- still exist
  - Could use TS, but that has busy waiting (we will solve this later)
- Possible problem with order of execution:
  - Consumer reads count and count == 0
  - Scheduler schedules the producer to run
  - 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
  - Scheduler now schedules the Consumer to continue, and the consumer blocks because it missed the wakeup() by the producer
  - Eventually, producer fills up the buffer and blocks
- How can we solve this problem?
  - Could use a mechanism to count the number of sleep() and wakeup() invocations

Semaphores

- Dijkstra proposed more general solution to mutual exclusion

- Semaphore S is an abstract data type that is accessed only through two standard atomic operations
  - wait( ) (also called P(), from Dutch word *proberen* "to test")
    - somewhat equivalent to a test-and-set
    - also involves *decrementing* the value of S
  - signal( ) (V(), from Dutch word *verhogen* "to increment")
    - *increments* the value of S

- OS provides ways to create and manipulate semaphores atomically

Semaphores

```
typedef struct {
    int value;
    PID *list[ ];
} semaphore;
```

Both wait() and signal() operations are atomic

```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
signal(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Mutual Exclusion using Semaphores
Binary Semaphore

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

Process P1:

```
wait(S);
   // execute critical section
   counter++;
signal(S);
```

Process P2:

```
wait(S);
   // execute critical section
   counter--;
signal(S);
```

- Both processes atomically wait() and signal() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable ***counter***
- This will solve the mutual exclusion like a mutex lock, but will also eliminate the busy waiting

Problems with Semaphores

shared R1, R2;

semaphore Q = 1;   // binary semaphore as a mutex lock for R1

semaphore S = 1;   // binary semaphore as a mutex lock for R2

Process P1:		*Process P2:*	
wait(S);	(1)	wait(Q);	(2)
wait(Q);	(4)	wait(S);	(3)
modify R1 and R2;		modify R1 and R2;	
signal(S);		signal(Q);	
signal(Q);		signal(S);	

- Potential for Deadlock

Deadlock

- In the previous example,
  - Each process will block on a semaphore
  - The signal() statements will never get executed, so there is no way to wake up the two processes
  - There is no rule about the order in which wait( ) and signal( ) operations may be invoked
  - In general, with N processes sharing N semaphores, the potential for deadlock grows

Other problematic scenarios for Semaphores

- A programmer mistakenly follows a wait() with a second wait() instead of a signal()

- A programmer forgets and omits the wait(mutex) or signal(mutex)

- A programmer reverses the order of wait() and signal()

Another problem with synchronization

shared R1, R2;

semaphore S = 1;   // binary semaphore as a mutex lock for R1 & R2

Process P1:	*Process P2:*	*Process P3:*
wait(S);	wait(S);	wait(S);
modify R1 and R2;	modify R1 and R2;	modify R1 and R2;
signal(S);	signal(S);	signal(S);

Potential for starvation

Starvation

- The possibility that a process would never get to run
- For example, in a multi-tasking system the resources could switch between two individual processes
- Depending on how the processes are scheduled, a third process may never get to run
- The third task is being starved of accessing the resource

Semaphore Solution for Mutual Exclusion

```
semaphore lock = 1;  // initial value of semaphore is 1
shared int count;
shared data_type buffer[MAX];
```

Code for p_1

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

Code for p_2

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

- Busy waiting removed from the mutual exclusion when waiting on lock
- Will show a complete solution in another video

Does the Kernel need Synchronization?

- At any time, many kernel mode processes may be active
  - Share kernel data structures
  - Notice that even though user processes have their own address spaces, race conditions can still arise when they execute in kernel mode, e.g. executing a system call

- Preemptive and non-preemptive kernels
  - Preemptive kernel: allows a process to be preempted while running in kernel mode
    - Race conditions can occur

  - Non-preemptive kernel: does not allow a process to be preempted while running in kernel mode
    - Race conditions cannot occur

Windows Synchronization

- Kernel level
  – Single processor system: temporarily mask interrupts for all interrupt handlers that may also access a shared resource
  – Multiprocessor system: use spin lock (busy waiting)

- User level
  – Dispatcher objects: mutex locks, semaphores, …

Linux Synchronization

- Kernel level
  - Prior to version 2.6, non-preemptive kernel, but later versions are fully preemptive
  - Atomic integers: all math operations on atomic integers are performed without interruptions

```
atomic_t counter;
atomic_set(&counter, 5);
atomic_add(10, &counter);
…
```

  - Mutex locks, spin locks and semaphores, enabling/ disabling interrupts on single processor systems

- User level
  - futex, semop( ): system call

pthread Synchronization

- Mutex locks
  - Used to protect critical sections
- Some implementations provide semaphores through POSIX SEM extension
  - Not part of Pthread standard

```
#include <pthread.h>
pthread_mutex_t m; //declare a mutex object
pthread_mutex_init (&m, NULL); // initialize mutex object
```

```
//thread 1
pthread_mutex_lock (&m);
   //critical section code for th1
pthread_mutex_unlock (&m);
```

```
//thread 2
pthread_mutex_lock (&m);
   //critical section code for th2
pthread_mutex_unlock (&m);
```

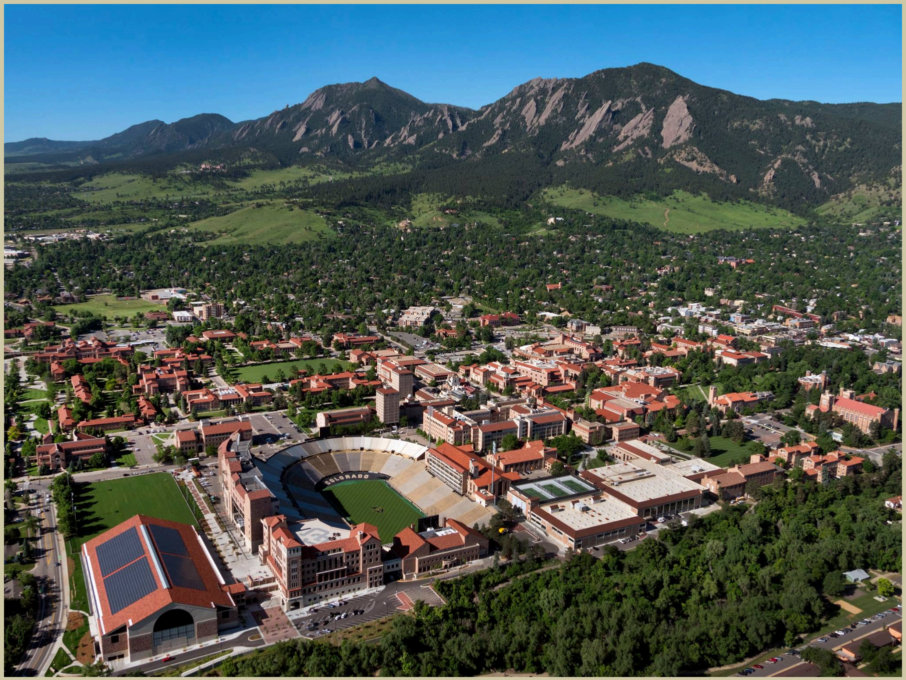
pthread mutex

- pthread mutexes can have only one of two states: lock or unlock
- Important restriction
  - Mutex ownership: Only the thread that locks a mutex can unlock that mutex
  - So, mutexes are strictly used for mutual exclusion while binary semaphores can also be used for synchronization between two threads

POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
//  pshared: 0 (among threads); 1 (among processes)

int sem_wait(sem_t *sem); //same as wait( )

int sem_post(sem_t *sem); //same as signal( )

sem_getvalue( ), sem_close( )
```

Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox

University of Colorado
Boulder