CS 492: Operating Systems

Inter Process Communication (3)

Semaphores

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Mechanisms for Mutual Exclusion

1) Software solution
   - 1.1) Strict alternation
   - 1.2) Peterson's solution
   - 1.3) Sleep and wakeup

2) Hardware solution
   - 2.1) Interrupt disabling
   - 2.2) Test-and-Set lock (TSL)

3) Higher level solutions

Mechanisms for Mutual Exclusion

1) Software solution
   - 1.1) Strict alternation
   - 1.2) Peterson's solution
   - 1.3) Sleep and wakeup

2) Hardware solution
   - 2.1) Interrupt disabling
   - 2.2) Test-and-Set lock (TSL)

3) Higher level solutions

Why Solutions Discussed So Far are Not Good Enough?

- Basic ideas of software & hardware solutions
- Disadvantages
  - It requires **busy wait**, i.e., continuously testing a variable until some value appears
  - Wastes CPU time
  - Low priority job blocks high priority ones

How to Improve?

- Instead of keeping blocked processes busy wait, let them sleep
- System call *Sleep*
  - Caller gives up the CPU for some duration of time, until other processes wake it up
- System call *Wakeup*
  - Caller wakes up some sleeping process

Producer/Consumer Problem w/ Sleep/Wakeup

```
Shared variable:
    const int n;
    int  count=0;
```

Producer

```
    while (1){
        produce an  item A;
        if(count==N) //full buffer
           sleep();
        insert item;
        count++;
        if(count==1) //was buffer empty?
            wakeup(consumer)
}
```

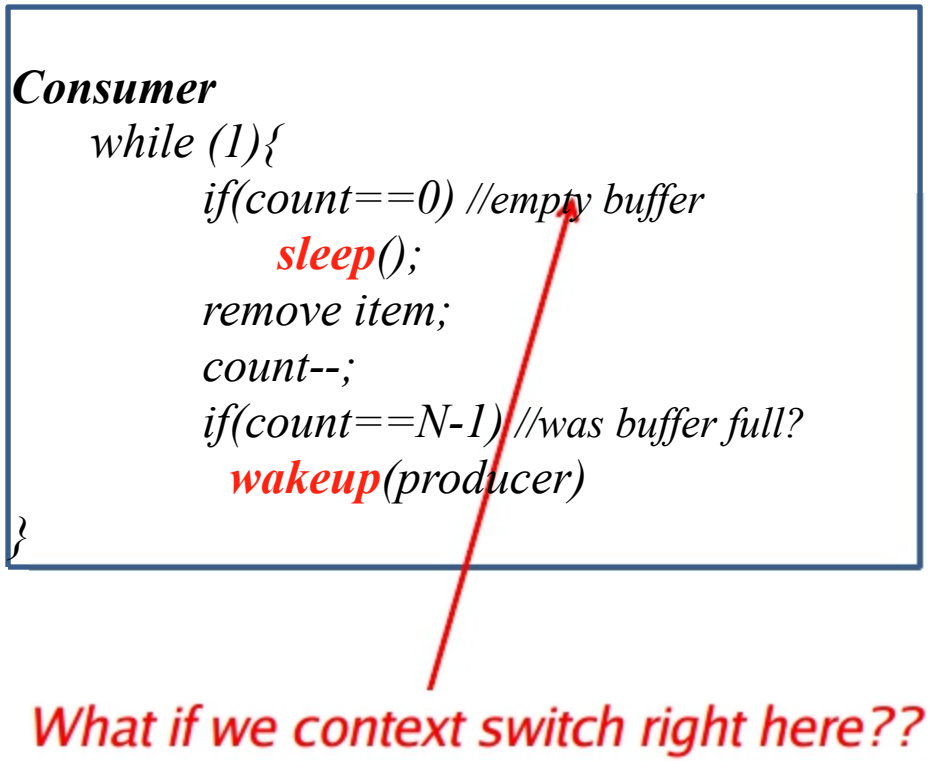
Consumer

```
    while (1){
        if(count==0) //empty buffer
            sleep();
        remove item;
        count--;
        if(count==N-1) //was buffer full?
          wakeup(producer)
}
```

Possible Chaos of Sleep&Wakeup

```
Producer
    while (1){
        produce an  item A;
        if(count==N) //full buffer
          sleep();
        insert item;
        count++;
        if(count==1) //was buffer empty?
          wakeup(consumer)
}
```

```
Consumer
    while (1){
        if(count==0) //empty buffer
          sleep();
        remove item;
        count--;
        if(count==N-1) //was buffer full?
          wakeup(producer)
}
```



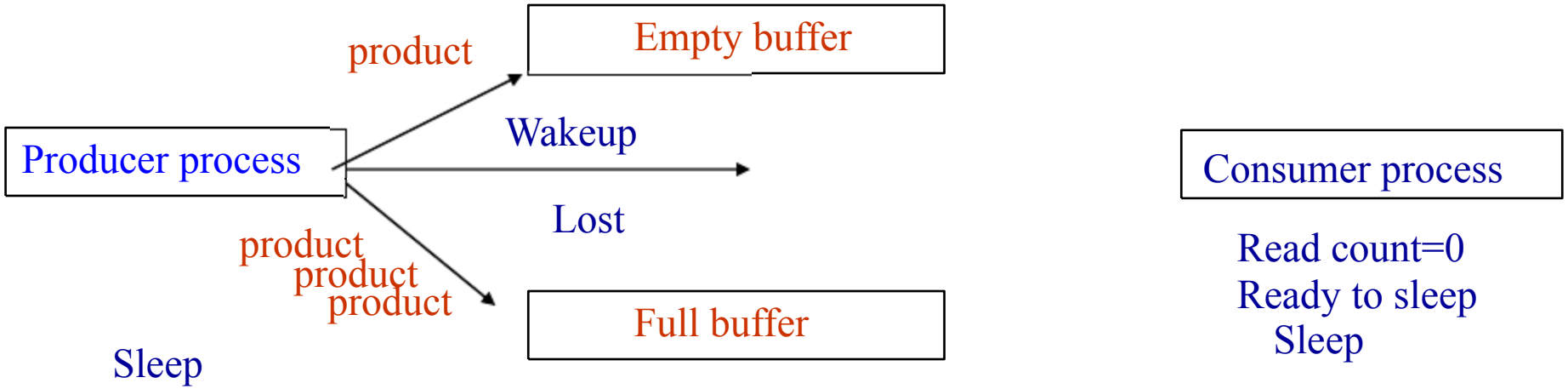
- What will happen?

Possible Chaos of Sleep&Wakeup

```
Producer
    while (1){
        produce an  item A;
        if(count==N) //full buffer
          sleep();
        insert item;
        count++;
        if(count==1) //was buffer empty?
          wakeup(consumer)
}
```

```
Consumer
    while (1){
        if(count==0) //empty buffer
          sleep();
        remove item;
        count--;
        if(count==N-1) //was buffer full?
          wakeup(producer)
}
```

What if we context switch right here??



Both producer and consumer processes sleep forever!

Lessons We Learned (1/2)

- Producers and consumers
  - Insert and remove items from the shared buffer in parallel
    - Producer: count++
    - Consumer: count--
- Requirement 1: Mutual exclusion!

Lessons We Learned (2/2)

- Fixed-size buffer
  - Full: the producers wait (or **sleep**).
  - Empty: the consumers wait (or **sleep**).
  - When the *producer* inserts items into the empty buffer: it **wakes up** the sleeping consumers
  - When the *consumer* removes items from the full buffer: it **wakes up** the sleeping producers
- Requirement 2: correct implementation of **paired** sleep and wakeup operations (scheduling constraints, or synchronization)

Goals for Today

- Mutual exclusion
  - Software-based solutions
  - Hardware-based solutions
  - Higher-level solutions
    - Semaphores

Semaphores

- Purpose: count the number of **wakeups** saved
- Solves the lost wakeup problem
- How does it work:
  - Define a count variable, or a *semaphore*
  - Semaphore is initialized to zero, i.e., no wakeup is saved
  - Operations:
    - ***Down (for sleep):*** decrements the value by 1 (use 1 saved wakeup)
    - ***Up (for wakeup):*** increments the value by 1 (save a wakeup)

Semaphores: Implementation

- Classic implementation (Dijkstra, ’65)
  - Down (val)
    - Checks to see if val > 0
    - If yes, val = val – 1;
    - If val == 0, process sleeps without completing *down*
  - Up(val)
    - val = val + 1;
    - If there are sleeping processes, wake up sleeper (chosen by scheduler)

Semaphores: More Discussion

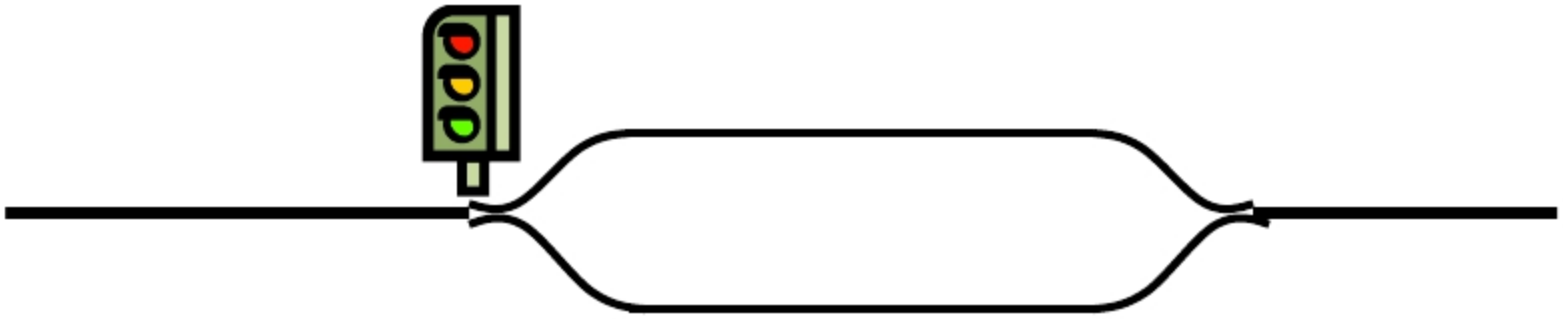
- *Down* and *up* are atomic
  - Checking values, changing values, and going to sleep, all is done as a single action (help of hardware)
  - Once the operation has started, no other process can access the semaphore until the operation has completed or blocked

- Thread going to sleep in Down won't miss wakeup from Up – even if they both happen at same time

- Why is this different from the processes busy waiting?

Semaphores like Integers

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are Down and Up – can't read or write value, except to set it initially
  - Operations must be atomic
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Types of Semaphores

- Two types of semaphores:
  - Counting semaphore
    - » Down - # of processes waiting on the resource
    - » Up - # of resources available
  - Binary semaphore
    - » Valued 0/1
    - » easier to implement at the expense of some additional functionality
    - » also called ***mutex***

Critical Section Using Semaphores

Shared semphore int mutex = 1;

Repeat

    Down(mutex);

    Critical section

    Up(mutex);

    Remainder section

Until false

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*.

Recall: Our Learned Lessons on Producer/Consumer Problem

- Requirement 1: Mutual exclusion on the shared buffer (for count-- and count++)

- Requirement 2: correct implementation of paired sleep and wakeup operations

Producer-Consumer Problem w/ Semaphores

- Requirement 1: Mutual exclusion on the shared buffer
  - Semaphore *mutex* initialized to 1
    - Only one process can be in the critical section at a time.
- Requirement 2: Scheduling constraints *wakeup* and *sleep*
  - Semaphore *full* initialized to 0
    - To block producer when buffer is full
  - Semaphore *empty* initialized to N
    - To block consumer when buffer is empty

Producer-Consumer Problem w/ Semaphores

```
semaphore mutex = 1;     // Mutex is binary semaphore
semaphore empty=N;       // counts empty buffer slots
semaphore full=0;        // counts occupied buffer slots
```

```
Producer:
  while(true){
        down (empty);
        down (mutex);
        Add item to buffer
        up (mutex);
        up (full);
}
```

```
Consumer:
 while(true){
        down (full);
        down (mutex);
        Remove item from buffer
         up (mutex);
         up (empty);
}
```

Game of Orders (1/2)

- The sequence of down and up operations:
  - Producer: down(empty), down(mutex), up(mutex), up(full)
  - Consumer: down(full), down(mutex), up(mutex), up(empty)

- Is order of *down* important?

```
Producer(item) {
    down(mutex);
    down(empty);
     // Add to buffers
    up(mutex);
    up(full);
}
```

(consumer is the same)

Game of Orders (2/2)

- The sequence of down and up operations:
  - Producer: down(empty), down(mutex), up(mutex), up(full)
  - Consumer: down(full), down(mutex), up(mutex), up(empty)

- Is order of *up* important?

```
Producer(item) {
    down(empty);
    down(mutex);
     // Add to buffers
     up(full);
     up(mutex);
}
```

(consumer is the same)

Subtle issue!

- How do the processes share lock variables (turn), semaphores, or a common buffer?
  - Shared data structures, can be stored in the kernel, and only accessed via system calls, or
  - Modern OSs offer a way for processes to share portion of their address space with other processes.

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (4)

```
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
```

Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (5)

```
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Using threads to solve the producer-consumer problem.

Question

- Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore.