Synchronization Primitive - Semaphores

Lecture 11

Klara Nahrstedt

CS241 Administrative

- SMP2 Quiz on Monday, 2/12/07
- Read Chapter 5.3 in Stallings Book and Chapter 14 in R&R

Discussion

- In uni-processor
  - Concurrent processes cannot be overlapped, only interleaved
  - Process runs until it invokes system call, or is interrupted
  - To guarantee mutual exclusion, hardware support could help by allowing disabling interrupts

```
While(true) {
    /* disable interrupts */
    /* critical section */
    /* enable interrupts */
    /* remainder */
}
```

What's the problem with this solution?

Discussion

- In multi-processors
  - Several processors share memory
  - Processors behave independently in a peer relationship
  - Interrupt disabling will not work
  - We need hardware support
  - The hardware support is based on execution of multiple instructions atomically
  - Atomic execution of a set of instructions means that these instructions are treated as a single step that cannot be interrupted

Test and Set Instruction

```
boolean Test_And_Set(boolean* lock) atomic {
    boolean initial;
     initial  = *lock;
    *lock = true;
    return initial;
}
```

pseudo-code!

Usage of Test_And_Set for Mutual Exclusion

```
Pi
{
  while(1) {
   while(Test_And_Set(lock))
               {  };
      /* Critical Section */
      lock =0;
      /* remainder */}  }
```

```
Void main ()
{
  lock = 0;
  parbegin(P1,….,Pn);
}
```

Busy Waiting Issue !!!

Semaphore Concept

Fundamental Principle: two or more processes want to cooperate by means of simple signals

For signaling introduce

- Special Variable : **Semaphore** *s*
- Primitives:
  - `semSignal(s)` – transmit signal via semaphore s
  - `semWait(s)` – receive signal via semaphore s

Primitives semSignal and semWait must be ATOMIC!!!

Note: Different notation is used for semSignal and semWait (P for semWait, V for semSignal, ‘wait’ for semWait, ‘signal’ for semSignal)

Definition of Semaphore Primitives (Counting Semaphore)

```
struct semaphore{
    int count;
    queueType queue; };

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    { place this process in s.queue;
      block this process;
    }
}
```

```
void semSignal(semaphore s)
{
    s.count++;
    if (s.count ≤ 0)
     {
     remove a process P from
    s.queue;
     place process P on ready list;
    }
}
```

Definition of Binary Semaphore Primitives

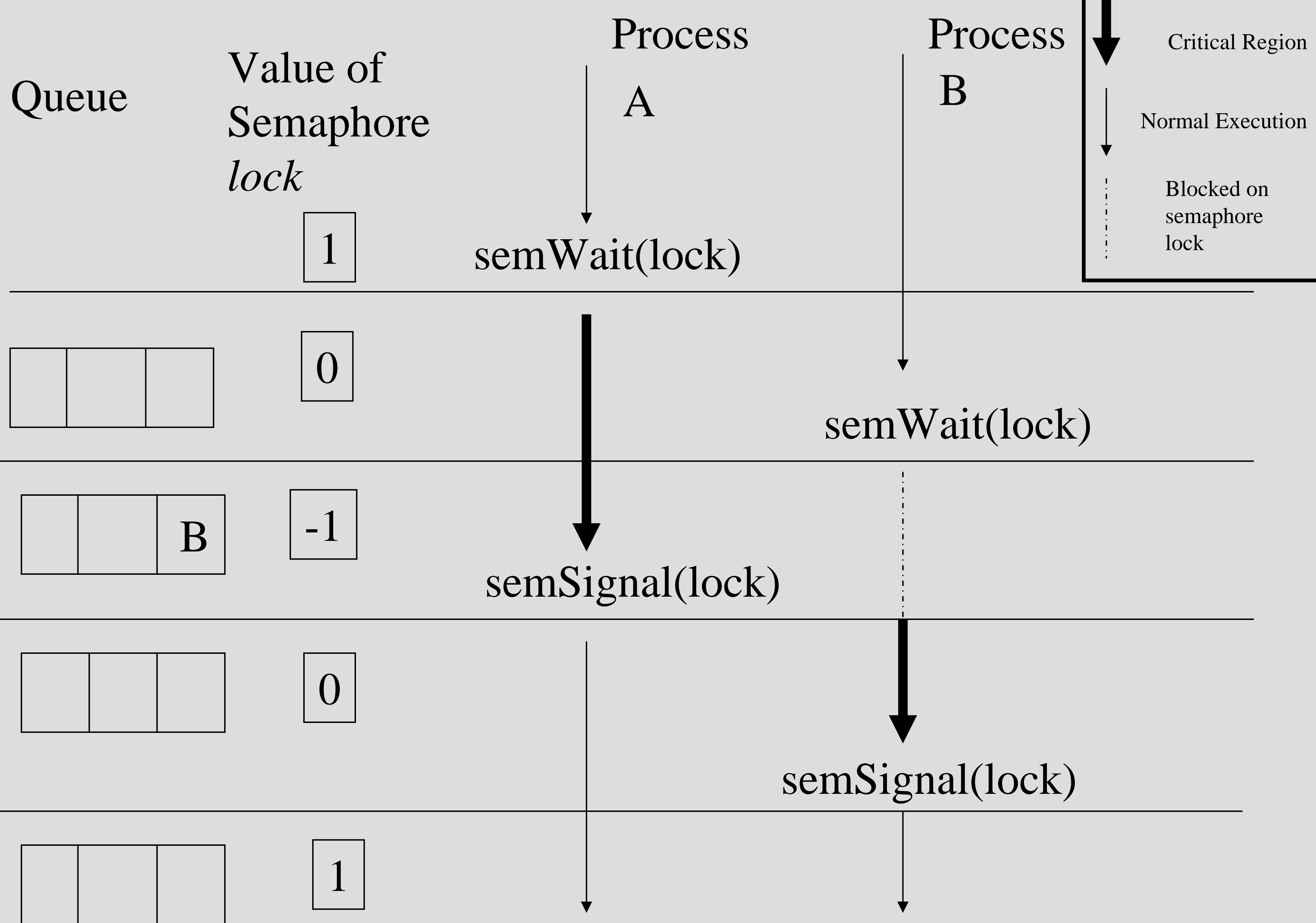
```
struct binary_semaphore {
enum {0,1} value;
queueType queue; };

void semWaitB(binary_semaphore s)
{
   if (s.value == 1)
    s.value = 0;
   else
    { place this process in s.queue;}
      block this process;
   }
}
```

```
void semSignalB(binary_semaphore s)
{
   if (s.queue is empty())
      s.value = 1;
   else
    {
    remove a process P from s.queue;
    place process P on ready list;
   }
```

Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi
{
   while(1)   {
   semWait(s);
     /* Critical Section */
   semSignal(s);
    /* remainder */
   }
}
```

```
lock = 0;
Pi
{
  while(1) {
   while(Test_And_Set(lock)) { };
      /* Critical Section */
      lock =0;
      /* remainder */
   }
}
```


Queue
Value of Semaphore *lock*
Process A
Process B
Critical Region
Normal Execution
Blocked on semaphore lock
1
semWait(lock)
0
semWait(lock)
B
-1
semSignal(lock)
0
semSignal(lock)
1

Exercise – Exam Question

Consider 2 processes sharing two semaphores S and Q to protect critical variables ‘a’ and ‘b’. What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?

process 1 executes

```
while(1) {
   semWait(S);
    a;
   semSignal(Q);
   }
```

process 2 executes

```
while(1) {
   semWait(Q);
      b;
   semSignal(S);
}
```

Implementation of Semaphores in POSIX

POSIX:SEM semaphore is
    variable of type sem_t

Atomic Operations:

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
Int sem_trywait(sem_t *sem);
Int sem_wait(sem_t *sem);
```

Use <semaphore.h>

Unnamed Semaphores Ch 14 pp 491-501

#include
<semaphore.h>
Sem_t sem;

You cannot make a copy of a semaphore variable!!!

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
unsigned value);

pshared == 0 only threads of process creating semaphore can use semaphore.

Sharing Semaphores

Sharing semaphores between threads within a process is easy, use pshared==0

Forking a process creates copies of any semaphore it has… this does not share the semaphore

Making **pshared** non zero allows any process that can access the semaphore to use it. This places the semaphore in global environment.

sem_init can fail!!!

In unsuccessful, sem_init returns -1 and sets errno.

errno	cause
EINVAL	Value > sem_value_max
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

Initialization Example

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror(“Failed to initialize semaphore semA”);
```

Semaphore Operations

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

Destroying a semaphore that's been destroyed gives undefined result.

Destroying a semaphore on which a thread is blocked gives undefined results.

Semaphore Operations

#include <semaphore.h>

int sem_post(sem_t *sem);

signal safe

can be used in signal handlers

if unsuccessful, returns -1 and sets errno

errno == EINVAL if semaphore doesnt exist

Semaphore Operations

int sem_trywait(sem_t *sem);

doesn't block

returns -1 and errno==EAGAIN if semaphore zero

can be interrupted by signal – errno == EINTR

int sem_wait(sem_t *sem);

blocks if semaphore zero

can be interrupted by signal – errno == EINTR

Summary

Semaphores

Semaphore implementation

POSIX Semaphore

Programming with semaphores

S: Chapter 5 pp 215-227,

RR:Chapter 14 pp488-497