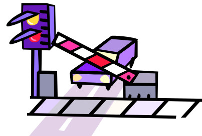
Synchronization and Semaphores



Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

1

CS241 Readings

- 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 where access to a memory location excludes any other access to that same location
  - The hardware support is based on execution of multiple instructions atomically (test and set)

Test and Set Instruction

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

atomic = *executed in a single shot without any interruption*

Note: this is more accurate than the textbook version

Using Test_And_Set for Mutual Exclusion

```
Pi {
   while(1) {
      while(Test_And_Set(lock)) {
         /* spin */
      }

      /* Critical Section */
      lock =0;
      /* remainder */
   }
}
```

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

Works, but has performance loss because of busy waiting.

Semaphores

- Fundamental Principle:
  - Two or more processes want to cooperate by means of simple signals
- Special Variable: **`semaphore s`**
  - A special kind of “int” variable
  - Can’t just modify or set or increment or decrement it

Semaphores

- Before entering critical section
  - **`semWait(s)`**
    - receive signal via semaphore **`s`**
    - “down” on the semaphore
    - Executed
- After finishing critical section
  - **`semSignal(s)`**
    - transmit signal via semaphore **`s`**
    - “up” on the semaphore
- Implementation requirements
  - **`semSignal`** and **`semWait`** must be atomic

Semaphores

- Different notation can be used
  - `semSignal`
    - `V` - verhogen
    - `signal`
    - `up`
  - `semWait`
    - `P` - proberen
    - `wait`
    - `down`

Semaphores vs. Test_and_Set

Semaphore

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

Test_and_Set

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

Inside a Semaphore

- Avoid busy waiting by suspending
  - Block if `s == False`
  - Wakeup on signal (`s = True`)
- Multiple process waiting on `s`
  - Keep a list of blocked processes
  - Wakeup one of the blocked processes upon getting a signal
- Semaphore data structure

```
typedef struct {
   int count;
    queueType queue;
   /* queue for procs.
   waiting on s */
} SEMAPHORE;
```

Inside a Semaphore

```
typedef struct {
   int count;
    queueType queue;
} SEMAPHORE;
```

`semSignal` and `semWait` must be atomic

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

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

Binary Semaphores

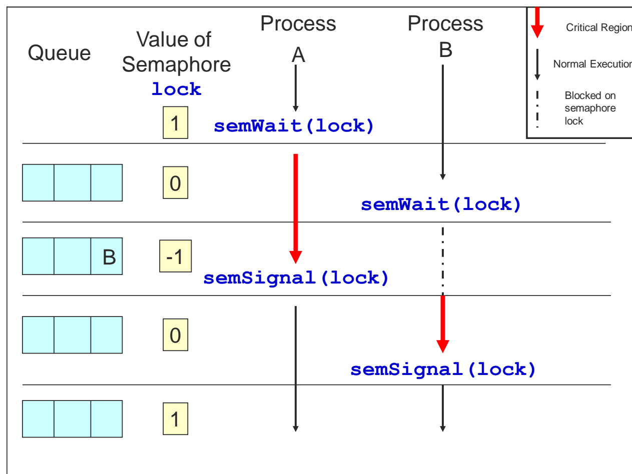
```
typedef struct bsemaphore {
    enum {0,1} value;
    queueType queue;
 } BSEMAPHORE;
```

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

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

Mutual Exclusion Using Semaphores

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



Semaphore Example 1

```
semaphore s = 2;
Pi {
    while(1) {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
    }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1) {
      semWait(s);
      /* CS */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
  - up to 2 processes can enter CS
- When might this be desirable?
  - allow up to 2 processes simultaneously inside CS,
    - e.g., limit number of processes reading a var
  - Be careful not to violate mutual exclusion inside CS!

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1) {
      semWait(s);
      /* CS */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1) {
      semWait(s);
      /* CS */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
  - No one can enter CS! Ever!
- When might this be desirable?
  - Never!

Semaphore Example 3

```
semaphore s = 0;
P1 {
   /* do some stuff */
   semWait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   semSignal(s);
   /* do some more stuff */
}
```

- What happens?
- When might this be desirable?

Semaphore Example 3

```
semaphore s = 0;
P1 {
   /* do some stuff */
   semWait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   semSignal(s);
   /* do some more stuff */
}
```

- What happens?
  - P1 waits until P2 signals
  - if P2 signals first, P1 does not wait
- When might this be desirable?
  - Having a process/thread wait for another process/thread

Semaphore Example 4

Process 1 executes:

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

Process 2 executes:

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

- Two processes
  - two semaphores: S and Q
  - Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?

The downside of using semaphores: Deadlocks!

- Two processes, P0 and P1, share two semaphores `S` and `Q`

```
Down  = semWait()
Up    = semSignal()
```

```
P0                                  P1
S:= 1; Q:=1;

Down(S); // S=0      ------->      Down(Q); //Q=0
Down(Q); // Q= -1    <-------
                     ------->      Down(S); // S=-1
       <-------------------------- DEADLOCK
// P0 blocked                      // P1 blocked

Up(S);                             Up(Q);
Up(Q);                             Up(S);
```

Be Careful when Using Semaphores!

- Violation of Mutual Exclusion
  - Up(mutex); — or — mutexUnlock();
  - critical section — or — criticalSection();
  - Down(mutex); — or — mutexLock();
- Violation of Mutual Exclusion (omit wait(mutex)/mutexLock())
  - critical section — or — critical Section();
  - Up(mutex); — or — mutexUnlock();
- Deadlock Possibility
  - Down(mutex); — or — mutexLock();
  - critical section — or — criticalSection();
  - Down(mutex); — or — mutexLock(P);
- Non-progress Situation (omit signal(mutex)/mutexUnlock())
  - Down(mutex); — or — mutexLock();
  - critical section — or — criticalSection();

POSIX Semaphores

- Named Semaphores
  - Provides synchronization between unrelated process and related process as well as between threads
  - Kernel persistence
  - System-wide and limited in number
  - Uses `sem_open`
- → Unnamed Semaphores
  - Provides synchronization between threads and between related processes
  - Thread-shared or process-shared
  - Uses `sem_init`

POSIX Semaphores

- Data type
  - Semaphore is a variable of type `sem_t`
- Include `<semaphore.h>`
- 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);
```

Unnamed Semaphores

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

- Initialize an unnamed semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno`
- Parameters
  - `sem`:
    - Target semaphore
  - `pshared`:
    - 0: only threads of the creating process can use the semaphore
    - Non-0: other processes can use the semaphore
  - `value`:
    - Initial value of the semaphore

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

Sharing Semaphores

- Sharing semaphores between threads within a process is easy, use `pshared==0`
  - Forking a process creates copies of any semaphore it has… `sem_t` semaphores are not shared across processes
- A non-zero `pshared` allows any process that can access the semaphore to use it
  - Places the semaphore in the global (OS) environment

`sem_init` can fail

- On failure
  - `sem_init` returns -1 and sets `errno`

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

```
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);
```

- Destroy an semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno`
- Parameters
  - `sem`:
    - Target semaphore
- Notes
  - Can destroy a `sem_t` only once
  - Destroying a destroyed semaphore gives undefined results
  - Destroying a semaphore on which a thread is blocked gives undefined results

Semaphore Operations

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

- Unlock a semaphore
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== EINVAL` if semaphore doesn't exist)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented
    - sem == 0: one blocked thread will be allowed to run
- Notes
  - `sem_post()` is reentrant with respect to signals and may be invoked from a signal-catching function

Semaphore Operations

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

- Lock a semaphore
  - Blocks if semaphore value is zero
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== EINTR` if interrupted by a signal)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread blocks

Semaphore Operations

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

- Test a semaphore's current condition
  - Does not block
- Returns
  - 0 on success
  - -1 on failure, sets `errno` (`== AGAIN` if semaphore already locked)
- Parameters
  - `sem`:
    - Target semaphore
    - sem > 0: thread acquires lock
    - sem == 0: thread returns

Summary

- Semaphores
- Semaphore implementation
- POSIX Semaphore
- Programming with semaphores
- Chapter 9 (S5.1-5.3)
- Chapter 13 (RR14.1 - 14.4)