CSCE 451/851
Operating Systems Principles

Semaphores

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE451

1

Problems with Proposed Mutual Exclusion Algorithms

- Algorithms are complex & brittle
- All employ "busy waiting"

2

Semaphores

A higher-level synchronization primitive

- An abstract data type
- A non-negative integer variable with two operations
  - » **`down(`***sem***`)`** (Also often called "**`P()`**", "**`wait()`**", ...)
    - Decrement *sem* by 1, if *sem* > 0. Otherwise "wait" until it is possible to do so and then decrement.
  - » **`up(`***sem***`)`** (Also often called "**`V()`**", "**`signal()`**", ...)
    - Increment *sem* by 1.
- Both operations are assumed to be *atomic*

3

Using Semaphores

Solving the critical section problem

- Use a *binary semaphore* for mutual exclusion

```
var mutex : binary_semaphore := 1
```

```
process P1
begin
   :
  down_b(mutex)
  <critical section>
  up_b(mutex)
   :
end P1
```

mutex

```
process P2
begin
   :
  down_b(mutex)
  <critical section>
  up_b(mutex)
   :
end P2
```

4

Using Semaphores

Producer/Consumer synchronization

```
globals
mutex : binary_semaphore := 1      nextIn,nextOut : 0..n-1 := 0
buf   : array [0..n-1] of char     count : 0..n := 0
```

```
process Producer
begin
  loop
    <produce a character "c">
    while count = n do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n

    downb(mutex)
    count  := count + 1
    upb(mutex)
  end loop
end Producer
```

```
process Consumer
begin
  loop
    while count = 0 do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n

    downb(mutex)
    count  := count - 1
    upb(mutex)

    <consume a character "c">
  end loop
end Consumer
```

5

Condition Synchronization

- Awaiting the development of a specific state within the computation

```
process Producer
begin
  loop
    <produce a character "c">
    while count = n do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn+1 mod n

    downb(mutex)
    count  := count + 1
    upb(mutex)
  end loop
end Producer
```

```
process Consumer
begin
  loop
    while count = 0 do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut+1 mod n

    downb(mutex)
    count  := count - 1
    upb(mutex)

    <consume a character "c">
  end loop
end Consumer
```

6

Condition Synchronization

Producer/Consumer system with *counting semaphores*

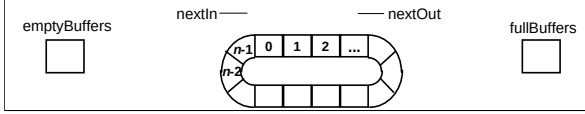
```
globals
fullBuffers  : semaphore := 0      buf   : array [0..n-1] of char
emptyBuffers : semaphore := n      nextIn, nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

    down(emptyBuffers)
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
    up(fullBuffers)
  end loop
end Producer
```

```
process Consumer
begin
  loop
    down(fullBuffers)
    c := buf[nextOut]
    nextOut := nextOut+1 mod n
    up(emptyBuffers)

    <consume a character "c">
  end loop
end Consumer
```



7

Implementing Semaphores

Hardware-based solutions

- Disabling interrupts

```
down(var sem : semaphore)
begin
  loop
    DISABLE_INTS
    exit when(sem > 0)
    ENABLE_INTS
  end loop
  <interrupts are disabled>

  sem := sem - 1
  ENABLE_INTS
end down
```

```
up(var sem : semaphore)
begin
  DISABLE_INTS
  sem := sem + 1
  ENABLE_INTS
end up
```

8

Implementing Semaphores
Hardware-based solutions

- Using special instructions: *test-and-set*
  - » perform a LOAD, COMPARE, and STORE in *one indivisible* operation

```
function TST(var flag : boolean) : boolean
begin
  TST  := flag
  flag := FALSE
end TST
```

9

Implementing Semaphores
Using *test-and-set*

- A binary semaphore (assume TRUE = 1, FALSE = 0)

```
downb(var sem : binary_semaphore)
begin
  while (NOT TST(sem)) do
    NOOP
  end while
end down


upb(var sem : binary_semaphore)
begin
  sem := 1
end up
```

10

Implementing Semaphores

Using *test-and-set*

- General semaphores
  - » use 2 binary semaphores

```
globals mutex       : binary_semaphore := 1
        delay       : binary_semaphore := 0
        num_waiting : integer          := 0
```

```
down(var sem : semaphore)
begin
  downb(mutex)
  if (sem = 0) then
    num_waiting += 1
    upb(mutex)
    downb(delay)
    num_waiting -= 1
  end if
  sem := sem - 1
  upb(mutex)
end down
```

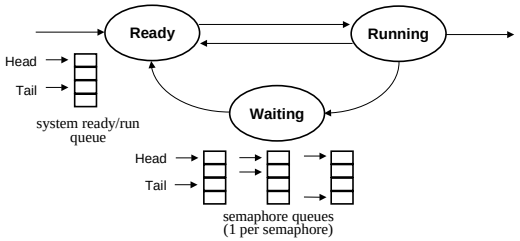
```
up(var sem : semaphore)
begin
  downb(mutex)
  sem := sem + 1
  if (num_waiting > 0) then
    upb(delay)
  else
    upb(mutex)
  end if
end up
```

11

Implementing Semaphores

Using an operating system kernel

- OS kernel functions
  - » suspend the currently executing process
  - » resume a ready process
  - » manage a queue



12

Implementing Semaphores

Using an operating system kernel

```
globals mutex          : binary_semaphore := 1
        num_waiting    : integer          := 0
        readyQueue     : system_queue
        runningProcess : process_id
```

```
down(var sem : semaphore)
begin
  down_b(mutex)
  if (sem = 0) then
    num_waiting += 1

    DISABLE_INTS
    insert_queue(sem,
runningProcess)
    next := remove_queue(readyQueue)
    up_b(mutex)
    dispatch(next)
    ENABLE_INTS
  end if

  sem := sem - 1
  up_b(mutex)
```

```
up(var sem : semaphore)
begin
  down_b(mutex)
  sem := sem + 1

  if (num_waiting > 0) then
    next := remove_queue(sem)
    insert_queue(readyQueue,
next)
  else
    up_b(mutex)
  end if
end up
```

13