Semaphores

Steve Goddard
goddard@cse.unl.edu

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

Using Semaphores
Solving the critical section problem
◆ Use a binary semaphore for mutual exclusion

```
var mutex : binary_semaphore := 1

process P1
    begin
        down(mutex)
        <critical section>
        up(mutex)
    end P1

process P2
    begin
        down(mutex)
        <critical section>
        up(mutex)
    end P2
```

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">%20while count = n do
NOOP
end while
buf[nextIn] := c
nextIn := nextIn+1 mod n
down(mutex)
count := count + 1
up(mutex)
end loop
end Producer

process Consumer
begin
loop
while count = 0 do
NOOP
end while
c := buf[nextOut]
nextOut := nextOut+1 mod n
down(mutex)
count := count - 1
up(mutex)
<consume a character "c”>
end loop
end Consumer

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
down(mutex)
count := count + 1
up(mutex)
<consume a character "c”>
end loop
end Producer

process Consumer
begin
loop
while count = 0 do
NOOP
end while
c := buf[nextOut]
nextOut := nextOut+1 mod n
down(mutex)
count := count - 1
up(mutex)
<consume a character "c”>
end loop
end Consumer
Condition Synchronization
Producer/Consumer system with counting semaphores

process Producer
begin
loop
<produce a character ‘c’>
down(emptyBuffers)
buf[nextIn] := c
nextIn := nextIn+1 mod n
up(fullBuffers)
end loop
end

process Consumer
begin
loop
<consume a character ‘c’>
down(fullBuffers)
c := buf[nextOut]
nextOut := nextOut+1 mod n
up(emptyBuffers)
end loop
end

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

Implementing Semaphores
Hardware-based solutions

- Disabling interrupts

down(var sem : semaphore)
begin
loop
DISABLE_INTS
exit when(sem > 0)
ENABLE_INTS
end loop
sem := sem - 1
ENABLE_INTS
end

toa
toa
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

Implementing Semaphores

Using test-and-set

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

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

    up(var sem : binary_semaphore)
    begin
        sem := 1
        end up
Implementing Semaphores
Using test-and-set

- General semaphores
  - use 2 binary semaphores

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

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

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

Implementing Semaphores
Using an operating system kernel

- OS kernel functions
  - suspend the currently executing process
  - resume a ready process
  - manage a queue
Implementing Semaphores
Using an operating system kernel

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

down(var sem : semaphore) begin
  down(mutex)
  if (sem = 0) then
    num_waiting += 1
    DISABLE_INTS
    insert_queue(sem, runningProcess)
  next := remove_queue(readyQueue)
  up(mutex)
  dispatch(next)
  ENABLE_INTS
  end if
  sem := sem - 1
  up(mutex)
end

up(var sem : semaphore) begin
  up(mutex)
  if (num_waiting > 0) then
    next := remove_queue(sem)
    insert_queue(readyQueue, next)
  end if
end
```