STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CS 492: Operating Systems

Inter Process Communication (4)

Monitors

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Announcement

- Hw1 due tonight!
- Teams have been formed
- Make sure you coordinate with your CAs (compilers etc.)
  - Try with them a hello world application.
- (Prob. next week) I will also introduce GIT.
  - You should create a repo, and add your CA

Announcement 2

- Next Tuesday (Feb 13): I will introduce the lab assignment
- Next Wednesday (Feb 14): Pthreads tutorial (required for Assignment 1, and Lab)
  - Introduction of thread programming
- Next Friday (Feb 16): Lab session (Inter-process communication)

Recap: Semaphores

- Semaphore: a count variable that counts the numbers of wakeups saved
- Operations:
  - *Down (for sleep):*
    - decrements the value by 1 (used 1 saved wakeup).
    - Process sleeps when it equals zero
  - *Up (for wakeup):* increments the value by 1 (save a wakeup)

Recap: Mutex

- A simplified version of semaphores
  – Two states: blocked and unblocked

- Implementation of mutual exclusion by using mutex

Reader-Writer Problem

- Readers: read data
- Writers: write data
- Rules:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot be in critical section together.
- Locking table: whether any two processes can be in the critical section simultaneously

	Reader	Writer
Reader	OK	No
Writer	No	No

Readers-Writers Problem

- Allow multiple processes do reading concurrently
- Allow one single process do writing at a time

- Semaphore mutex = 1; /* control critical section */
- Semaphore wrt = 1; /* # of writer process */
- int readcount = 0; /* reader counter */

Reader-Writer Solution

```
int readCount = 0;
semaphore mutex = 1;
semaphore wrt = 1;
```

```
reader() {
  while(TRUE) {
    <other stuff>;
    down(mutex);
    readCount++;

    if(readCount == 1)
      down(wrt);
    up(mutex);

    /* Critical section */
       access(resource);

    down(mutex);
    readCount--;
    if(readCount == 0)
      up(wrt);
    up(mutex);
  }
}
```

First reader blocks writers

Last reader unblocks writers

```
writer() {
  while(TRUE) {
    <other computing>;
    down(wrt);
    /* Critical section */
    access(resource);
    up(wrt);
  }
}
```

Writer blocks all readers

Is Semaphores the Final Solution?

- Semaphores accomplish 2 purposes:
  1) Mutual exclusion of the shared memory (mutex)
  2) Scheduling constraints: *down* and *up*
- But tricky to use
  - if down() and up() are scattered among several processes it may be difficult to use them correctly
    - We have shown that
      - Changing order of *down* may cause deadlocks
      - Changing order of *up* affects efficiency
- We need something that is simple and easy
  - Answer: monitors

Monitors

- Motivation
  - Would be nice if we had some clean, well-defined language support for synchronization...
  - Take complexity out of programmer's hands – let compiler deal with mutual exclusion
  - Cleaner idea
    - Use ***condition variables*** for scheduling constraints

- Rules:

1) Only one process can be active in a monitor at any time
2) Processes cannot access internal data of monitor
3) Put all critical sections inside monitor

Implementation of Monitors

- A package that consists of a collection of procedures, variables and data structures

```
Monitor
    Integer I
    Condition c
    Procedure producer()
    Procedure consumer()
End Monitor
```

- A programming language construct
- Compiler (not the programmer!) implements the mutual exclusion on monitor entries

Condition Variables in Monitors

- Mutual Exclusion is not enough. (req 2)
- Operations:
  - `Wait(&condVar)`: Causes the calling process to block. Re-acquire lock later, before returning.
  - `Signal(&condVar)`: Wake up one blocked process, if any
- Rule: Must hold Condition Variable when doing condition variable operations!

Condition Variables in Monitors (2)

- Signal: how do we make sure we don't have 2 active processes in the monitor?
- *Wait*, *signal*: similar to *sleep* and *wakeup.* Lost wakeUp calls in case of context switch.
  - Does this hold for Monitor Condition variables too?

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .      .      .
        end;
end monitor;
```

Condition Variables in Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;

    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Question

Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, *waituntil*, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example, waituntil $x < 0$ or $y + z < n$.

The signal primitive would no longer be needed. This scheme is clearly more general than that of Monitor Condition Variables, but it is not used.

Why not?

Monitor: Reader-Writer Problem

- The monitor solution uses two variables:
  - int readercount: Number of active readers; initially = 0
  - bool busy: indicate whether a writer is active initially = false

And two condition variables:

- Condition variable OKtoread= NIL
- Condition variable OKtowrite = NIL

Readers, and Writers

```
readers-writers : monitor;
begin
readercount : integer;
busy : boolean;
OKtoread, OKtowrite : condition;

procedure startread;

        begin
        if busy then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        (* Once one reader can start, they all can *)
        end startread;

procedure endread;

        begin
        readercount := readercount - 1;
        if readercount = 0 then OKtowrite.signal;
        end endread;

procedure startwrite;

        begin
        if busy OR readercount != 0 then OKtowrite.wait;
        busy := true;
        end startwrite;

procedure endwrite;

        begin
        busy := false;
        if OKtoread.queue then OKtoread.signal
                else OKtowrite.signal;
        end endwrite;

begin (* initialization *)
readercount := 0;
busy := false;
end;

end readers-writers;
```

Readers, and Writers

```
procedure startread;

        begin
        if busy then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        (* Once one reader can start, they all can *)
        end startread;
```

Readers, and Writers

```
procedure endread;

        begin
        readercount := readercount - 1;
        if readercount = 0 then OKtowrite.signal;
        end endread;
```

Readers, and Writers

```
procedure startwrite;

        begin
        if busy OR readercount != 0 then OKtowrite.wait;
        busy := true;
        end startwrite;
```

Readers, and Writers

```
procedure endwrite;

        begin
        busy := false;
        if OKtoread.queue then OKtoread.signal
                else OKtowrite.signal;
        end endwrite;
```