

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 5: Synchronization with Monitors

- Readings for this topic: Section 6.7
 - Paper by Lampson and Redell, Communications of ACM, 23 (2), Feb. 1980.
- *Monitors* are a high-level data abstraction tool combining interesting features.
- Existing implementations of monitors are embedded in programming languages. Best existing implementation is in the Mesa/Cedar language at Xerox. In C, a queue manipulation monitor might look like:

```
monitor QueueHandler;  
struct {int first, last, buffer[200]} queue;
```

```
AddToQueue(val)
```

```
int val;
```

```
{ -- add val to end of queue -- }
```

```
int RemoveFromQueue()
```

```
{ -- remove value from queue, return it -- }
```

```
endmonitor;
```

- Monitor lock:
 - Is mutual exclusion implicit?
- Are monitors a higher-level concept than semaphores? Easier and safer to use?
- Monitors need more facilities than just mutual exclusion. Need some way to wait.
 - Example: when tried to get item from empty queue.

```

int RemoveFromQueue()
  if (QueueEmpty) wait;
  remove first element;
  return it;

```

- *Busy-wait inside monitor?*
 - *Put process to sleep inside monitor?*
- *Condition variables:*
 - *Wait(condition):*
 - *Signal(condition):*
 - *Broadcast(condition):*

```

monitor QueueHandler;
struct {int first, last, buffer[200]} queue;
  condition q;

```

```

  AddToQueue(val)
  int val;
  { -- add val to end of queue -- }
  signal(q);

```

```

int RemoveFromQueue()
  if (QueueEmpty)
    wait(q);
  remove first element;
  return it;
endmonitor;

```

- Several different variations on the wait/signal mechanism. They vary in terms of who gets the monitor lock after a signal. We use ‘‘Mesa semantics’’:
 - On signal, signaller keeps monitor lock.
 - Awakened process waits for monitor lock with no special priority (a new process could get in before it). This means that the thing you were waiting for could have come and gone: must check again and be prepared to sleep again if someone else took it.

```

int RemoveFromQueue()
  if (QueueEmpty)
    wait(q);
  remove first element;
  return it;

```

- Readers and writers problem with monitors:
 - Each synchronization operation gets encapsulated in a monitored procedure: *checkRead*, *checkWrite*, *doneRead*, *doneWrite*.
 - Conditions: *OKToRead*, *OKToWrite*.

- *checkRead*()

```

{
  while ((AW+WW) > 0) {
    WR += 1;
    wait(OKToRead);
    WR -= 1;
  }
  AR += 1;
}

```

- *doneRead()*

```
{  
    AR -= 1;  
    if (AR==0 & WW>0) {  
        signal(OKToWrite);  
    }  
}
```

- *checkWrite()*

```
{  
    while ((AW+AR) > 0) {  
        WW += 1;  
        wait(OKToWrite);  
        WW -= 1;  
    }  
    AW += 1;  
}
```

- *doneWrite()*

```
{  
    AW -= 1;  
    if (WW > 0) {  
        signal(OKToWrite);  
    } else {  
        broadcast(OKToRead);  
    }  
}
```

- Write the monitor ReadersWriters using those procedures above.

- Why are **whiles** needed above?
 - Could all of the *signals* be *broadcasts*?
 - How do wait and signal compare to P and V?
-
- Summary:
 - Semaphores use a single structure for both exclusion and scheduling. What about monitors?
 - Monitors enforce a style of programming where complex synchronization code doesn't get mixed with other code. Is it good?