NWEN 303 Assignment 2

Daniel Braithwaite

August 12, 2016

1 Readers and Writers

1.1 a

For this problem we only need modify the StartR and EndR operations

```
procedure StartR
    if $nw$ != 0 then waitC(OKR)
    $nr \leftarrow nr + 1$
    $signalC(OKR)$
procedure EndR
    $nr \leftarrow nr + 1$
    if $nr == 0$ then signalC(OKW)
```

When a process starts a read and gains permission to read then nr is incremented by 1.

If there are currently processes reading then $nr > 0$. If another process attempts to start a read then as $nr > 0$ we know $nw = 0$ the process will be allowed to read even if there are writers waiting. So we see the rule is satisfy.

1.2 b

For this problem we only need to modify the EndW operations

```
procedure EndW
    $nw \leftarrow nw - 1$
    if $empty(OKW)$ then
        $signalC(OKR)$
    else
        $signalC(OKW)$
```

When ever we end write operation we first check if there are any processes waiting on the OKW condition, if there are none then we will signal a read process. Otherwise we signal a write process. Giving any waiting write process priority. Satisfying the rule.

1.3 c

For this problem we need to modify the monitor init, EndR and EndW operations.

```
procedure MONITOR RW
    $int\ nr \leftarrow 0$
    $int\ nw \leftarrow 0$
    $int\ writeCount \leftarrow 0$
    $cond\ OKR\ OKW$
procedure ENDR
    $nr \leftarrow nr - 1$
    $writeCount \leftarrow 0$
    if $nr == 0$ then
        $signalC(OKW)$
procedure ENDW
    $nw \leftarrow nw - 1$
    $writeCount \leftarrow writeCount + 1$
    if $writeCount\ >=\ 2$ or $empty(OKW)$ then
        $writeCount \leftarrow 0$
        $signalC(OKR)$
    else
        $signalC(OKW)$
```

When ever a write ends we increment the writeCount variable by 1. If this variable ever reaches 2 then we have reached the write limits so we reset it to 0 and signal a read process. We see this satisfy the rule as only two consecutive writes can occur.

2 Critical Section Problem

2.1 a

We define a monitor and two operations, Start and End. We will use the monitor to track how many processes are in there critical section at a time.

```
procedure MONITOR C
    $n \leftarrow 0$
    $cond\ OK$
procedure START
    if $n = M$ then
        $waitC(OK)$
    $n \leftarrow n + 1$
procedure END
    $n \leftarrow n - 1$
    $signalC(OK)$
```

We want to show that only M processes can be in there critical section at once. When a process calls the start operation, the count is incremented. If a process is uses the start operation when there are already M processes with access then the process waits for the OK condition to be signaled. The OK condition is signaled when one process calls the End operation. Meaning there can only be at most M processes in there critical section at once.

2.2 b

```
procedure SEMAPHRORE S
    $s.value \leftarrow M$
    $s.waitSet \leftarrow \{\}$
procedure PROCESS
    while true do
        non critical section
        $wait(S)$
        critical section
        $signal(S)$
```

We create a semaphore with a value of M (so there are M permissions). And we initialize the wait set to empty. We see this must satisfy the give rule. Any process attempting to gain a permission will be blocked (and added to the wait set) if the semaphore has already given out M permissions. When another process exits its critical section it will signal the semaphore, giving its permission to another process.

3 Producers and Consumers Revisited

We use a monitor to solve this problem, with a signaling policy of signal and continue. We have a buffer of size N. We want this buffer (of size K) to be circular, so when the head (or tail) reaches the end it should wrap around to overwrite old elements that we have "removed". So we will assume our monitor has two functions $increment(i, K)$ which will increment i and if it is equal to K reset it to 0. and size which will give us the number of elements in the buffer.

Starting a read or starting a write will give you an index to read from or write to. When a read or write is started the head or tail is updated so that the next process wont be given the same index.

Of course this courses an issue. What happens if a process starts a read operation and gets given index i. Then while it is still reading some operations occur causing the buffer to wrap around, since we have already incremented the head, position i can be written to while a process is still reading from it.

To solve this the monitor will keep track of whether a buffer index is being accessed and will have a condition variable for each of the buffer positions. So when starting a read or a write before we let the process access the buffer we check if there currently is a process accessing the given position, if there is then we wait for the condition variable assigned to that buffer position to be signaled. When finishing an operation we update a value to say that now that position doesn't have a process accessing it and we signal the condition variable for that buffer position.

```
buffer ← [K]
procedure Monitor RW
    head ← 0
    tail ← 0
    cond conds[K]
    cond notEmpty, notFull
    num[N]
procedure StartR
    pos ← head
    if length() == 0 then
        waitC(notEmpty)
    increment(head, K)
    signalC(notFull)
    if num[pos] == 1 then
        waitC(conds[pos])
    num[pos] ← 0
    returnpos
procedure EndR(int pos)
    num[pos] ← 0
    signalC(conds[pos])
procedure StartW
    pos ← tail
    if length() == N then
        waitC(notFull)
    increment(tail, N)
    signalC(notEmpty)
    if num[pos] == 1 then
        waitC(conds[pos])
    num[pos] ← 1
    returnpos
procedure EndW(int pos)
    num[pos] ← 0
    signalC(conds[pos])
```

Now we will make a more rigorous argument about why is solution satisfy the rule given. We see we can have multiple
We want to have multiple readers and writers accessing the buffer at the same time but ensure that no two processes can access the same position in

the buffer. If a process is reading or writing to position i then $num[i]$ will be equal to 1, indicating that that buffer position is currently being accessed. So if a new process attempts to start an operation that will involve index i it will be forced to wait until the $conds[i]$ condition is signaled. This condition is signaled when the process accessing position i ends its operation. For each of the condition variables in the *conds* array we will use a wait queue so we can ensure actions will happen in the order which they started waiting, this is so if we get a read, then a write the a read to the same index we want the operations to execute in that order otherwise the two read threads could read the same value. This argument guarantees that no two processes will be accessing the same.

If the buffer is full and a process attempts a write then the process will be forced to wait until a read is started at which point the buffer will not be full. Similar for if a read is attempted while the buffer is empty, the process will be waited until a write has started. Note that while it might not make sense to signal the waiting process when the operation starts but remember that the process will be forced to wait until the current operation at that index is finished.

4 Double Buffering

For this we use a monitor with a signal and continue policy, to handout permissions for two buffers. The monitor keeps track of whether reads and writes have occurred as to know when a process is allowed to read or write data, it will also handle swapping the buffers.

```
$readBuffer$
$writeBuffer$
**procedure** MONITOR RW
    $read \leftarrow true$
    $written \leftarrow false$
    cond OKR, OKW
**procedure** StartR
    **if** $read == true$ **then** waitC(OKR)
**procedure** EndR
    $read \leftarrow true$
    **if** $written == true$ **then**
        swap()
        signalC(OKW)
**procedure** StartW
    **if** $written == true$ **then**
        waitC(OKW)
**procedure** EndW
    $written \leftarrow true$
    **if** $read == true$ **then**
        swap()
        signalC(OKR)
**procedure** SWAP
    $readBuffer \leftarrow writeBuffer$
    $read \leftarrow false$
    $writeen \leftarrow false$
```

Now we show this solution satisfy the given correctness property's. The two variables read and written keep track of whether a read or write operation has occurred since the last swap. This is so if the consumer attempts to read but the data has already been read then we are forced to wait for a swap to occur (we don't want to read the same data twice), same for writes. Only when a read and a write have both occurred do we then swap the buffers around. So the read buffer can be read from and the write buffer can be written to at the same time.