Homework 3: Written Problems

Problem 1: Exercise 5.4, page 255

a.) Suppose there is no signal_all primitive. Modify the solution so that it uses only signal.

Add a variable with the number of waiting readers. Modify request_read() to increment before waiting. When you reach place in release_write() to signal all readers, increment through all waiting readers and signal each of them to put on the entry queue.

```
int waitingReaders = 0;
procedures request_read() {
      while (nw > 0)
         waitingReaders = waitingReaders + 1;
         wait(oktoread);
      nr = nr + 1;
   } // request_read
procedure release_write() {
      nw = nw - 1;
      signal(oktowrite);       // awaken one writer and
        for (i: waitingReaders) signal(oktoread);
   } // release_write
```

b.) Modify the solution to give writers preference instead of readers.

```
Int dw = 0; #Delayed writers
procedures request_read() {
      while (nw > 0 || dw > 0)
         wait(oktoread);
      nr = nr + 1;
   } // request_read

   procedure release_read() {
      nr = nr - 1;
      if ( nr == 0 )
         signal(oktowrite); // awaken one writer
   } // release_read
   procedure request_write() {
      while ( nr > 0 || nw > 0 || dw > 0)
        dw = dw + 1
       wait(oktowrite);
      nw = nw + 1;
      dw = dw - 1
   } // request_write
   procedure release_write() {
      nw = nw - 1;
      signal(oktowrite);       // awaken one writer and
      if (dw == 0)
        signal_all(oktoread);    // all readers
   } // release_write
```

Add a new counter that tracks the number of delayed writers. If any delayed writers are waiting, they will be awakened first and will go before any readers.

c.) Modify the solution so that readers and writers alternate if both are trying to access the database.

```
int dr = 0, dw = 0;

procedure request_write() {
   while ( nr > 0 || nw > 0 )
      dw = dw + 1
      wait(oktowrite);
      dw = dw - 1
   nw = nw + 1;
} // request_write
procedure release_write() {
   nw = nw - 1;
   if (dr > 0)
     signal_all(oktoread);    // all readers
   else
     signal(oktowrite);       // awaken one writer and

} // release_write
```

In each release function, summon the other delayed processes first.

d.) Modify the solution so that readers and writers are given permission to access the database in FCFS order. Allow readers concurrent access when that does not violate the FCFS order of granting permission.

```
procedures request_read() {
      while (nw > 0)
        wait(oktoread, time);
      nr = nr + 1;
   } // request_read


   procedure request_write() {
      while ( nr > 0 || nw > 0 )
         wait(oktowrite, time);
      nw = nw + 1;
   } // request_write
```

This works because with using wait(cv, rank) the processes are automatically ordered in a priority wait queue. However, wait is only called when there is some one currently waiting on a process to finish

Problem 2: Exercise 5.8, page 256-257

The Savings Account Problem. A savings account is shared by several people (processes). Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative. A deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

I asked this as a question on a test one semester. My solution to part a.) is shown below.

You are to do parts b.) and c.) of the problem.

a.) Develop a monitor to solve this problem. The monitor should have two procedures: deposit(amount) and withdraw(amount). First, specify a monitor invariant. Assume the arguments to deposit and withdraw are positive. Use the Signal and Continue discipline.

Solution to part a.):

Here is an easy solution that simply wakes up all blocked processes when a deposit is made. Each process then gets a shot at making the withdrawal. It is possible that no one will be able to if the balance is not large enough for any of the pending withdrawals. It is possible that multiple processes will succeed at completing their withdrawals. This solution is not fair; that is, a process might never get to make a withdrawal (it might starve).

```
Invariant: balance >= 0

monitor bank {

   int balance = 1000000;

   cond delay;

   proc deposit( int amount ) {

      balance += amount;

      signal_all( delay );

   } # deposit

   proc withdraw( int amount ) {

      while ( amount > balance ) {

         wait( delay );

      }

      balance -= amount;

      return amount;

   } # withdraw

} # bank monitor
```

b.) Write a monitor that services withdrawals FCFS. For example, suppose the current balance is $200, and one customer is waiting to withdraw $300. If another customer arrives, he must wait, even if he wants to withdraw at most $200. Assume there is a magic function amount(cv) that returns the value of the amount parameter of the first process delayed on cv. Use the Signal and Continue discipline.

```
Invariant: balance >= 0

monitor bank {

   int balance = 1000000;

   cond delay;

   proc deposit( int amount ) {

      balance += amount;

      if (balance >= amount(delay)) {

            signal(delay)

      }

   } # deposit

   proc withdraw( int amount ) {

      while ( amount > balance ) {

         wait( delay );

      }

      balance -= amount;

      return amount;

   } # withdraw

} # bank monitor
```

c.) Suppose the magic amount function does not exist. Modify your answer to b.) to simulate it in your solution.

```
Invariant: balance >= 0

monitor bank {

   int balance = 1000000;

   int withdrawAmount = 0

   cond delay;

   proc deposit( int amount ) {

      balance += amount;
```

```
      if (balance >= withdrawAmount) {
            signal(delay)
      }
   } # deposit
   proc withdraw( int amount ) {
      while ( amount > balance ) {
         while (withdrawAmount == 0) {
            withdrawAmount = amount
         }
         wait( delay );
      }
      balance -= amount;
      withdrawAmount = 0
      return amount;
   } # withdraw
} # bank monitor
```

Problem 3: Exercise 5.10, page 257

Atomic Broadcast. Assume one producer process and n consumer processes share a bounded buffer having b slots. The producer deposits messages in the buffer; consumers fetch them. Every message deposited by the producer is to be received by all n consumers. Furthermore, each consumer is to receive the messages in the order they were deposited. However, consumers can receive messages at different times. For example, one consumer could receive up to b more messages than another if the second consumer is slow.

Develop a monitor that implements this kind of communication. Use the Signal and Continue discipline.

```
Monitor atomic_broadcast {
      Cond delay;
      Message buffer[b];
      Int pos = 0;
      Procedure writeToBuffer(Message data) {
            Buffer[pos] = data;
            Pos = pos + 1;
```

```
            Signal_all(delay);
      }
      Procedure requestRead () {
            For (I = 0; I < b; I ++) {
                  While (I == pos) {
                        Wait(delay);
                  }
                  …read buffer[i]…
            }
      }
}
```

Problem 4: Exercise 5.13, page 258

Search/Insert/Delete. Three kinds of processes share access to a singly linked list: searchers, inserters, deleters. Searchers merely examine the list; hence, they can execute concurrently with each other. Inserters add new items to the end of the list. Insertions must be mutually exclusive to preclude inserting two items at the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Develop a monitor to implement this kind of synchronization. First specify a monitor invariant. Use the Signal and Continue discipline.

```
Monitor sid_monitor {
      # Really crappy monitor with quite a bit of context switching ☹
      # Invariant: for searcher, there can be any amount of concurrent
      searchers and at most 1 insert in parallel
      # For deleters numSearchers = 0 && numInserters == 0 && numDeleters ==
      0
      Int numSearchers = 0, numDeleters = 0, numInserters = 0;
      Cond insert, search, delete;
      Procedure search(data) {
            While (numDeleters > 0) {
                  Delay(search)
```

```
        }
        numSearchers = numSearchers + 1;
        …search through linked list…
        numSearchers = numSearchers – 1;
        signal_all(insert);
        signal_all(delete);
        signal_all(search);
}
Procedure insert(data) {
        While (numInserters > 0 || numDeleters > 0) {
                Wait(insert);
        }
        numInseters = numInserters + 1;
        …insert data in linked list…
        numInserters = numInserters – 1;
        signal_all(insert);
        Signal_all(delete);
        Signal_all(search);
}
Procedure delete(data) {
        While (numInserters > 0 || numDeleters > 0 || numSearchers > 0) {
                Wait(delete);
        }
        numDeleterss = numDeleters + 1;
        …delete data in linked list…
        numDeleters = numDeleters – 1;
        signal_all(insert);
        Signal_all(delete);
        Signal_all(search);
}
```

Problem 5: Exercise 5.16, page 259

Suppose a computer center has two printers, A and B, that are similar but not identical. Three kinds of processes use the printers: those that must use A, those that must use B, and those that can use either A or B.

Develop a monitor to allocate the printers, and show the code that the processes execute to request and release a printer. Your solution should be fair, assuming that printers are eventually released (that is, you do not have to worry about processes that request a printer and then never return it). Use the Signal and Continue discipline.

Due to the monitor, I don't think this solution prints concurrently, but it does allocate correctly.

```
Monitor print {
	Cond delayA;
	Int delayedOnA = 0;
	Int delayedOnB = 0
	Procedure printToPrinter(data) {
		# if data can only be printed on printer A
		If (data.ablePrinterA || !data.ablePrinterB) {
			# request A
			while (delayedOnA > 0) {
				delayedOnA++;
				wait(delayA);
				delayedOnA--;
			}
			printOnA(data);
			# release A
			signal(delayA);
		}
		# if data can only be printed on printer B
		Else If (data.ablePrinterB || !data.ablePrinterA) {
			#request B
			while (delayedOnB > 0) {
				delayedOnB++;
				wait(delayB);
				delayedOnB--;
```

```
                }
                printOnB(data);
                # release B
                signal(delayB);
          }
    # data can be printed on either
    Else {
          # if there are less delayed jobs on A, go to printer A, else, go
          to B
          If (delayedA <= delayed) {
                # request A
                while (delayedOnA > 0) {
                      delayedOnA++;
                      wait(delayA);
                      delayedOnA--;
                }
                printOnA(data);
                # release A
                signal(delayA);
          } Else {
                #request B
                while (delayedOnB > 0) {
                      delayedOnB++;
                      wait(delayB);
                      delayedOnB--;
                }
                printOnB(data);
                # release B
                signal(delayB);
          }
    }
}
```