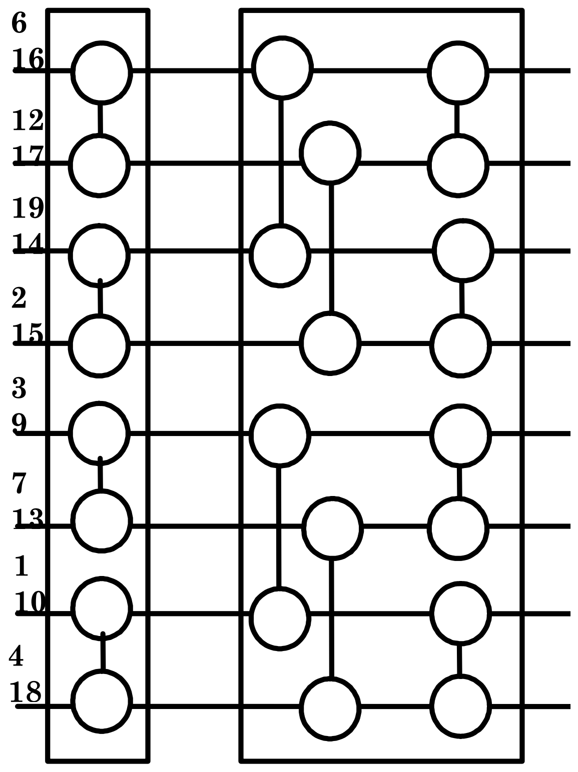
COP 4520 Fall 2005 **Midterm#2** **Name:** ____*KEY*__________________

1. The following is the first two or the three phases of an 8-node **bitonic sorting network** that we have **virtualized** to handle 16 numbers. This really is just the part that makes the list **bitonic**. One more phase completes the sort, but that would have added to your work without telling me any more about your knowledge.

3 **a.)** For each comparator, write a plus (+) or minus (–) to distinguish increasing from decreasing comparators.

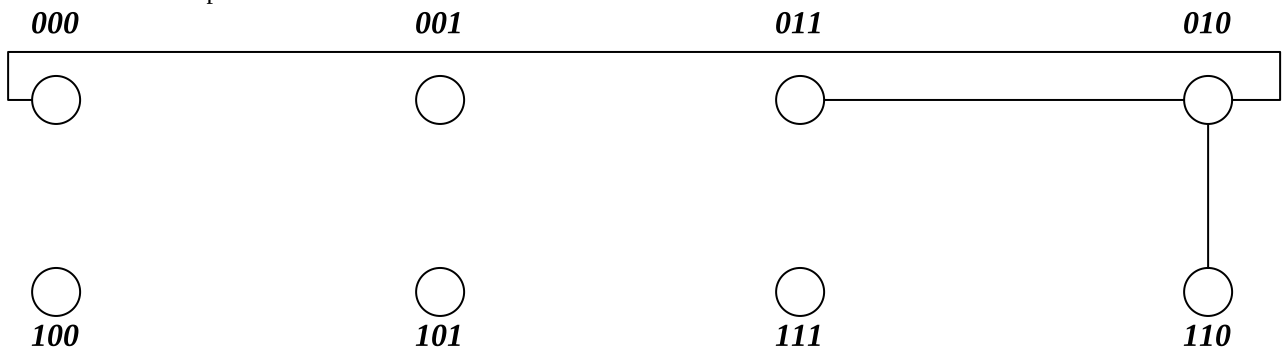
8 **b.)** Next show the values that are produced after each comparator performs its comparison swap. I have written the word ***Values*** under each column where you should be placing the eight pairs of values written on that communication line.



Values ***Values*** ***Values***

Answers to this are easy, but hard for me to edit into this document.

4 **c.)** Bitonic Sorting Networks can be mapped onto hypercubes in a very natural way. A 4 by 2 wraparound mesh can provide the same connectivity. Show this by numbering the nodes of the following such mesh, in a manner that achieves hypercube connectivity. Show the wires for your node numbered 010 that give it connectivity the nodes numbered 011, 000 and 110. Do not show any other wires. Of course, you cannot invent wires that do not already exist in such a wraparound 2d mesh.



2. This question concerns **ShearSort**.

3 **a.)** Give an example of two consecutive sorted dirty rows, each with 8 binary (0-1) values, and then show what these rows would look like after transpositions that occur with just one column step, assuming the rows are an even-odd pair.

There are lots of answers. Here's just one.

After row sort, but before column sort
00111111
10000000

After one step of column sort
00000000
10111111

2 **b.)** What two properties of a **ShearSort** allow us to focus on binary data in studying dirty rows?

Property 1: ***Oblivious***

Property 2: ***Comparison Exchange***

3 **c.)** The Rev in RevSort refers to what property of the row sorts? Give an example of what this means as regards of row 2 in a 16 by 16 mesh.

The row starting position is at the column numbered by the reverse of the row number. For example, row 2 starts at column 4.

PS: I apologize for saying row 0 in your question. Well, you get it for free!!

8 **3.** The following can be used to determine the length of a linked list in parallel, when one processor is associated with each node in the linked list.

```
plural int length = 1;
plural int partner = next; // linked list of processor numbers
while (partner != null) {
        length = length+ proc[partner].length;
        partner = proc[partner].partner;
}
```

What is the order of the execution time of the algorithm, given **N** nodes in the linked list? ***O(lg N)***

What is the order of the cost of the algorithm, again given **N** nodes in the linked list? ***O(N lg N)***

Consider a virtualization of this algorithm. Here we would have **p** processors and **N > p** elements in the linked list. Assuming that the we are really lucky and the elements are evenly distributed, every processor has no more that **N/p** elements. The processors first calculate the lengths of their local list, then run the above algorithm using only the length of the heads of their local lists as the initial values of length. Subsequently each processor propagates this length to the children of the head, assigning a length to each – this is the length of the list headed by the local root + k-1, where the node is the k-th element of the sublist.

What is the order of the execution time of the algorithm, given **N** and **p**? ***N/p + lg p + N/p = O(N/p + lg p)***

What is the order of the cost of the algorithm, given **N** and **p**? ***O(N + p lg p)***

What value of **p** gives you a cost that is comparable to the sequential algorithm, while gaining the speedup achieved when using **N** processor? ***P = N/lgN*** ***since N/p + lg p = N/(N/lg N) + lg(N/lg N) = O(lg N) and***

N + p lg p = N + N/lgN * lg (N/lg N) = O(N)

10 **4.** Write a monitor, **CP**, that is resource manager, serving producers and consumers of some resource. You do not have to actually keep track of the resource, just the number of units available. The monitor has two services, **produce** and **consume**, each with a single integer argument. **produce(n)** increases the number of units of the resource by **n**; **consume(m)** requests **m** units and cannot complete until those many are available. Of course, as a side effect, the completion of a **consume** reduces the available resources by the number of units requested.

Hint: There are only a few lines of code for each service, but you must be careful.

```
monitor CP {
// Shared data
  int N=0;
  cond w;
  procedure produce( int n ) {  // add n units of the resource
      N += n; // add new resources
      signal_all(w); // signal all waiting requestors
  }
  procedure consume( int m ) { // acquire m units of the resource
      while ( m >= N ) wait(w); // So long as request is not satisfiable, wait for more resources to be added
      N -= m; // satisfy request
  }
}
```

The following are required characteristics of the implementation:

a. If a request comes that can be immediately satisfied, it may not be delayed.
b. If a request comes that cannot be satisfied, the requestor is delayed.
c. When resources become available, delayed satisfiable requests must be honored in the order in which the requests were received. To assist in this, assume that the delay queue is managed in FIFO order.
d. Resources must be given out if there are any delayed requests that can be satisfied by amount of available units.

You must state your assumption about the signal semantics. That is, are you using signal and wait (**SW**) or signal and continue (**SC**) semantics, and is that critical to the correctness of your solution? If so, why?

Assume this uses SC semantics. Here is a precise statement of SC semantics:

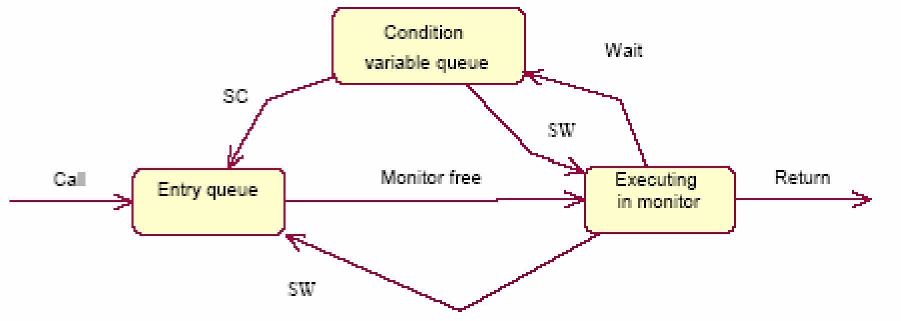
signaller continues executing, signalled processs(es) get placed on entry queue in order removed from wait queue.

There is some possible ambiguity. While all this is going on, the monitor stays busy so no new requests can come in from the outside, but can such an attempt lead to one of these new requesting processes getting intermingled on the entry queue with the older request processes? Most implementations of SC semantics do not preclude this. They just take the ready processes from the wait queue and place them in the entry queue. That will mean that they are all in the same order relative to each other, but new arrivals that came in while this was going on might have jumped ahead. Still it's a pretty good solution (sort of in the 90% realm that most people are happy with).

Can this work better with SW semantics? The problem is that SW semantics for signal_all are not well-defined. A useful and very reasonable definition is

signaller blocks, signalled processess move to ready queue, bypassing entry queue, getting placed in ready queue in the order in which they left wait queue. signaller process gets placed at end of entry queue. The first process in ready queue is immediately selected for execution.

This would mean that we could meet all our requirements with the above solution and have no ambiguity about the order of processes when returned to wait queue.



ALTERNATIVE ANSWER

10 **4.** Write a monitor, **CP**, that is resource manager, serving producers and consumers of some resource. You do not have to actually keep track of the resource, just the number of units available. The monitor has two services, **produce** and **consume**, each with a single integer argument. **produce(n)** increases the number of units of the resource by **n**; **consume(m)** requests **m** units and cannot complete until those many are available. Of course, as a side effect, the completion of a **consume** reduces the available resources by the number of units requested.

```
monitor CP {
// Shared data
  int N=0;
  cond w;
  procedure produce( int n ) {  // add n units of the resource
      N += n; // add new resources
      signal(w); // signal the oldest waiting requestor; use SW semantics
  }
  procedure consume( int m ) { // acquire m units of the resource
      if ( m >= N ) N -= m; // satisfied request immediately
      else { // need to patiently wait more resources
          boolean satisfied = false;
          while( !satisfied ) {  // keep trying until request honored
              wait(w);  // wait until more resources are produced
              if ( m >= N ) { N -= m; satisfied = true; }  // allocation needs are met
              if ( !empty(w) )  signal(w);  // wake up next requestor in queue; use SW semantics
          }
      }
  }
}
```

This uses SW semantics. Here is a precise statement of SW semantics:
signaller blocks, signalled process gets to execute immediately, signaller process gets placed at end of entry queue.

With SW, each process in queue is tried in order. It is also the case that, if we do not use SW then the solution above has the potential to get into an infinite loop. Think about it.

There is still an issue with how we handle property (c) if some requestor fails to get satisfied when awakened. When the final (oldest) requestor is taken from the queue and no signal is issued, we must now consider the policy for scheduling those who temporarily gave up their control of the monitor. Looking back at the definition of SW semantics, we see that the unsatisfied processes reenter the queue (issue a wait) in the same order in which they exited it.

There still is some possible ambiguity. While all this going on, the monitor stays busy so no new requests can come in from the outside, but can such an attempt lead to one of these new requesting processes getting intermingled on the entry queue with the older request processes? Most implementations preclude this by giving precedence to delayed processes, but it does mean this solution is implementation dependent.

By the way there is a signal and urgent wait. Its semantics are:
signaller blocks, signalled process gets to execute immediately, signaller process gets placed at head of entry queue.

12 **5.** Redo the resource manager from question 4, but using semaphores rather than a monitor.

```
// Shared data
 int N = 0, nw = 0; nd = 0;
 sem lock = 1, w = 0;
 procedure produce( int n ) {  // add n units of the resource
     P(lock);
     N += n; // add new resources
     if ( nw >0 ) {nd = nw;  for (int i=0; i<nw; i++)  V(w);  } } // wake 'em all up; don't release lock
     else V(lock); // there were no unsatisfied requests, so give up lock
 }
 procedure consume( int m ) { // acquire m units of the resource
     P(lock); // lock for critical section
     if ( m >= N ) { N -= m; V(lock); } // satisfied request immediately
     else {
         nw++;  // there is one more request needing to be satisfied
         boolean satisfied = false;
         V(lock);
         while( !satisfied ) { // keep trying until request honored
             P(w);  // wait until more resources are produced; assume FIFO queue
             // lock is already owned
             nd- -; // one more was awaken by latest production
              if ( m >= N ) { N -= m; nw- -; satisfied = true; } // allocation needs are met
             if (nd == 0) V(lock);  // all have awakened
         }
     }
 }
```

This also works perfectly if we never force release of a processor unless the current thread blocks or issues a delay(). That's really similar to SC (signal or signal_all) so far as the iterated V is concerned. It works since the lock is held, somewhat in the fashion of SW. This holding of lock prevents outside intervention, an issue with the monitors.

This was a bit longer than I had realized. Sorry.

ALTERNATIVE ANSWER

12 **5.** Redo the resource manager from question 4, but using semaphores rather than a monitor.

```
// Shared data
  int N = 0, nw = 0; nd = 0;
  sem lock = 1, w = 0, d = 0, awake = 0;
  procedure produce( int n ) {  // add n units of the resource
      P(lock);
      N += n; // add new resources
      if ( nw > 0 ) V(w); // pass baton to oldest requestor; don't release lock
      else V(lock); // there were no unsatisfied requests, so give up lock
  }
  procedure consume( int m ) { // acquire m units of the resource
      P(lock); // lock for critical section
      if ( m >= N ) { N -= m; V(lock); } // satisfied request immediately
      else {
          nw++;  // there is one more request needing to be satisfied
          boolean satisfied = false;
          V(lock);
          while( !satisfied ) { // keep trying until request honored
              P(w);  // wait until more resources are produced; assume FIFO queue
              // baton is passed to this requestor; lock is already owned
               if ( m >= N ) { N -= m; satisfied = true; } // allocation needs are met
              nw- -; // one fewer in w queue
               if (nw >0 ) {  // if more in w queue, must pass the baton to next
                  V(w);  // pass baton – note we assume we hold control until blocked
                  if ( !satisfied ) { nd++; P(d); nw++; V(awake); }  // delay until the queue is drained
              }
              else { // there's no left to whom we should pass the baton
                  while( nd>0 ) {  // queue is drained so wake up all those delayed
                      V(d); // wake up one; that is only one who can continue since we hold lock
                      P(awake); // be sure it actually is awake
                      nd- -; // count down to zero
                  }
                  V(lock);  // all have awakened
              }
          }
      }
  }
```

This works perfectly if we never force release of a processor unless the current thread blocks or issues a delay(). That's really similar to SC (signal, not signal_all) so far as V is concerned, but it works since the lock is held, somewhat in the fashion of SW. this holding of lock prevents outside intervention, an issue with the monitors.

10 **6.** Redo the resource manager from question 4, but using some combination of tuple space services (**read**, **readIfExists**, **take**, **takeIfExists**, **write**). Unlike in 4 and 5, you do **NOT** have to be concerned about fairness. In other words, it is okay to have a random requestor satisfied. Use a general method called **delay()** anytime you need to do a busy wait.

```
procedure produce( int n ) {  // add n units of the resource
    take("ASSETS", ?assetCount);  // see what's available
    assetCount += n; // add new resources
    write("ASSETS", assetCount);  // write back updated resource amount
}
procedure consume( int m ) { // acquire m units of the resource
    boolean satisfied = false; // not yet; must see what's available
    while ( !satisfied ) {
        take("ASSETS", ?assetCount);  // see what's available
        if (assetCount >= m) { // if can satisfy request, allocate them
            assetCount -= m; satisfied = true; // allocate recourses and indicate success
        }
        write("ASSETS", assetCount);  // write back updated resource amount
        if ( !satisfied ) delay();  // to avoid constantly taking and writing
    }
}

procedure init() {
    write("ASSETS", 0); // initial tuple indicate no instances of resource
}
```

BUTT UGLY VERSION

```
procedure produce( int n ) {  // add n units of the resource
    for (int i=0; i<n; i++)  write("ASSETS");  // write one tuple per unit of resource
}
procedure consume( int m ) { // acquire m units of the resource
    for (int i=0; i<m; i++)  take("ASSETS");  // creates unnecessary delays, perhaps of production
 }
```