15-440 (Fall 2008)
Homework #1

1. Mutual Exclusion Through Memory Atomicity

The following is a potential solution to the challenge of properly ensuring mutually exclusive access to a critical section using only the atomicity of fundamental memory operations. Please argue for its correctness or provide an explained execution sequence that demonstrates a failure.

```
Process (int pid)
{
  boolean request_set[MAX_PIDS];
  boolean request_lock = FALSE;
  boolean in_cs = FALSE;


  while (FOREVER)
  {
    /* Make request */
    while (request_lock); request_lock = TRUE; // Acquire lock on request list
    request_set[pid] = WANT_CS;                // Add request to list
    request_lock = false;                      // Release list lock

    /* Wait for our turn */
    for (int index=(pid+1); index < MAX_PIDS; index++)
    {
      while (request_set[index]);
    }

    in_cs = TRUE; // Note that we're in the CS

    // Enter critical section
    << Critical Section >>

    // Get out of the CS
    in_cs = FALSE;

    << Non-Critical Section >>

    request_set[pid] = DONT_WANT_CS;
  }
}
```

2. Semaphores and Dinner (Page 1 of 2)

This problem asks you to solve a concurrency control situation using semaphores. Please do not use other concurrency control tools, such as mutexes, condition variables, atomic memory, telepathy, &c. Your solution should enforce the policies described below while avoiding deadlock, starvation, livelock, and other bad things.

At dinner time, students must visit three different stations in order to collect everything needed for a full meal:

- The station for the tray and utensils (TrayAndUtensils)
- The station for the salad or other side-dish (SaladOrSide)
- The station for the entree (Entree)

Each station can accomodate some number, which is known in advance, of concurrent students. Other students must wait in a single line at each station and advance in FCFS order. This gives the configuration of the stations a set of a parameters

- Number of concurrent students at TrayAndUtensils (N_{TU})
- Number of concurrent students at SaladOrSide (N_{SaSu})
- Number of concurrent students at Entree (N_E)

Please assume that each station is capable of reaching the concurrency level specified by the parameters. For example, please assume that any underlying buffer data structure can support the level of concurrency specified above.

The student thread calls a function, getMeal() to obtain a meal. The meat of this function is sketched out below:

```
void getMeal()
{
  // Pre condition: Student is hungry, but empty handed student

  // This must be first -- or SPILL!
  getTrayAndUtensils();

  // The order of these really doesn't matter, but the code
  // imposes this arbitrary order
  getSaladOrSide();
  getEntree();

  // Post condition: Time to chow down! The student has everything needed
  //                 This function returns, and the student grabs dinner
}
```

2. Semaphores and Dinner (Page 2 of 2)

You should implement the following four functions, which you saw in use in getMeal() above:

- void init (int Nta, int Nsasu, int Ne)

  This function initializes any necessary state and will be called before any significant portion of the cafeteria simulation. It should create and initialize semaphores, counters, &c.

- void getTrayAndUtensils();

  This function is called by a student when s/he wants to get the tray and utensils. The "work" within the function should be a simple comment, "// Get tray and utensils". Your real task is to manage the semaphores that ensure that the specified level of concurrency is not exceeded.

- void getSaladOrSide();

  This function is called by a student when s/he wants to get the salad or side. The "work" within the function should be a simple comment, "// Get salad or side". Your real task is to manage the semaphores that ensure that the specified level of concurrency is not exceeded.

- void getEntree();

  This function is called by a student when s/he wants to get the entree. The "work" within the function should be a simple comment, "// Get entree". Your real task is to manage the semaphores that ensure that the specified level of concurrency is not exceeded.

3. Mutual Exclusion with Monitors

Imagine a narrow bridge running North-South along a public roadway. The bridge is nominally two lanes, with one lane for each direction, and can accomodate two opposing cars simultaneously, but it is too narrow to properly accomodate even a single truck. As a result, a single truck must occupy both lanes in order for it to cross the bridge. Bicyclists can also use the bridge -- two bicycles can fit simultaneous per lane of travel.

Please assume the following in your answer:

- Cars and bicycles always travel in the right (proper) lane
- Cars and trucks cannot pass each other and, as a result, must be serviced in the relative order in which they arrive.
- Bicycles travel singly, or in pairs, across the bridge.
- Bicycles always travel in the right (proper) lane.
- As they approach the bridge, bicycles can ride in the curb-lane and, as a result, will pass cars and trucks that have not yet entered the bridge.

Please implement a monitor-paradigm solution to this problem using monitors. Your solution should be written in pseudo-code similar to that used within your textbook, C or C++, or Java. The goal of your solution is, of course, to provide a structure that maximizes the use of the bridge while preventing collisions.

Please also state the semantics of your solution: Mesa, Brinch Hanson, or Hoare.

Your monitor should include exactly the following entry procedures:

- EnterNorthBound (enum vehicle_type);
- EnterSouthBound (enum vehicle_type);
- ExitNorthBound (enum vehicle_type);
- ExitSouthBound (enum vehicle_type);

Where *enum vehicle_type* is one of CAR, BICYCLE, or TRUCK;

Note: If you are writing in pseudo-code, please assume that the monitor infrastructure exists and that you can use wait(), signal(), and broadcast(), as we did in class. If writing in Java, please write valid Java code using wait(), notify(), and notifyAll(), as shown in class. If writing in C or C++, please simulate monitors using mutexes and condition variables. Mutexes should be used to ensure that only one process is active within the monitor at a time and also to protect the predicate of the condition variables. Condition variables should be used to enqueue and wake-up blocked threads.

4. Parallel Quicksort Using Kernel Supported Threads (Page 2 of 2)

Below we have supplied quicksort code. Suppose that you've been given a machine with a massive number of processors. Please modify the code so that uses threads to take advantage of the multiple processors.

We understand that many things can be done to implement a parallel quicksort efficiently and effectively - many research papers have been published in this area. Please don't get fancy - a simple and intuitive solution will suffice. Furthermore, please don't concern yourself with idiosyncrasies of the memory model, &c. Don't think to hard – it is a question about threads, not parallel algorithms or architecture.

The thread package ensures that each thread is visible to the kernel and is independently schedulable. The thread package provides exactly the following interface:

- int thread_spawn (void * (*func)(void *), void *arg) /* returns tid */
- void thread_join (int tid); /*Suspends (blocks) calling thread, until the specified thread ends; similar to waitpid () for processes */

Hints:

- "func" is the function that will form the body of the thread. Its prototype should be something like: void *threadBody (void *pointer_to_parameter)
- "arg" is a pointer to data. This pointer will get passed to the "func" as its only parameter.
- The "arg" pointer can point to a struct that contains several elements.

4. Parallel Quicksort Using Kernel Supported Threads (Page 2 of 2)

The quick sort code:

```
void Quicksort (long long *hugeArray, long long left, long long right)
{
  int i, j;
  int pivot;


  if (right - left +1) < MIN_PARTITION
  {
   /* Too small to be worth a quicksort */
    ShellSort (hugeArray, left, right);
    return;
  }

  /* swap pivot w/right-most item */
  pivot = (left+right)/2;
  swap (hugeArray, pivot, right);
  pivot=right;

  /* i starts at the left and moves right;
     j starts at the right and moves left.
     each stops if the referenced item is on the wrong side of
     te pivot value. Then they are swapped
    */
  i = left;  j = pivot-1;

  do
  {
    while (hugeArray[i] < hugeArray[pivot])
      i++;

    while ((j > i) && (hugeArray[j] > hugeArray[pivot]))
      j--;

   swap (hugeArray, i, j);

  } while (i < j)

  /* Put the pivot back where it belongs -- in the middle */
  swap (hugeArray, i, pivot);

  /* sort the left and right sub-partitions recursively */
  Quicksort (hugeArray, left, i-1);
  Quicksort (hugeArray, i+1, right);
}

long long hugeArray[ARRAY_SIZE]; /* global */

void main()
{
  // Read in huge array

  QuickSort (hugeArray, 0, ARRAY_SIZE-1);
}
```