

## Homework 2

### Assignment due:

**Problems 1, 2, 3:** Hand in at the start of class (9:30 am) on Tuesday, March 4th, or turnin electronically by the same time. Turnin typed (not hand-written) answers to the non-programming problems. If using electronic turnin, provide a pdf file named **answers.pdf**. Use the command: `turnin 422assign2 answers.pdf Problems 1-3` are worth 36 points (12 points ea.).

A reminder: Homework assignments are individual efforts. The two projects can be done in teams of two, not the homework. You may discuss the meanings of questions with classmates, but the answers and programs you turn in are to be yours alone. For the exercises from the book, explain your answers clearly and succinctly.

### Problem 1: Exercise 3.2, page 142

Suppose a computer has atomic decrement **DEC** and increment **INC** instructions that also return the value of the sign bit of the result. In particular, the decrement instruction has the following effect:

```
DEC(var, sign):
  < var = var - 1;
    if (var >= 0)
      sign = 0;
    else
      sign = 1; >
```

**INC** is similar, the only difference being that it adds 1 to var.

Using **DEC** and/or **INC**, develop a solution to the critical section problem for n processes. Do not worry about the eventual entry property. Describe clearly how your solution works and why it is correct.

### Problem 2: Exercise 4.13, page 194

Consider the following proposal for implementing **await** statements using semaphores:

```
sem e = 1, d = 0; # entry and delay semaphores
int nd = 0;      # delay counter
# implementation of <await (B) S;>
P(e);
while (!B) {
  nd = nd + 1;
  V(e);
  P(d);
  P(e);
}
S;
while (nd > 0) {
  nd = nd - 1;
  V(d);
}
V(e);
```

Does this code ensure that the **await** statement is executed atomically? Does it avoid deadlock? Does it guarantee that **B** is true before **S** is executed? For each of these questions, either give a convincing argument why the answer is “yes”, or give an execution sequence that illustrates why the answer is “no”.

### Problem 3: Exercise 4.21, pages 197-8

Consider the following solution to the readers/writers problem. It employs the same counters and semaphores as in Figure 4.13 (page 176), but uses them differently.

<pre> int nr = 0, nw = 0; # numbers of readers and writers  sem e = 1;          # mutual exclusion semaphore sem r = 1, w = 0;   # delay semaphores  int dr = 0, dw = 0; # delay counters </pre>	
<pre> process Reader[i = 1 to M] {   while (true) {     P(e);     if (nw == 0) {       nr = nr + 1;       V(r);     }     V(e);     P(r); # await read permission      read the database      P(e);     nr = nr - 1;     if (nr == 0 and dw &gt; 0) {       dw = dw - 1;       nw = nw + 1;       V(w);     }     V(e);   } } </pre>	<pre> process Writer[j = 1 to N] {   while (true) {     P(e);     if (nr == 0 and nw == 0) {       nw = nw + 1;       V(w);     }     else       dw = dw + 1;     V(e);     P(w);      write the database      P(e);     nw = nw - 1;     if (dw &gt; 0) {       dw = dw - 1;       nw = nw + 1;       V(w);     }     else       while (dr &gt; 0) {         dr = dr - 1;         nr = nr + 1;         V(r);       }     V(e);   } } </pre>

a.) Carefully explain how this solution works. What is the role of each semaphore? Show that the solution ensures that writers have exclusive access to the database and a writer excludes readers.

b.) What kind of preference does the solution have? Readers preference? Writers preference? Alternating preference?

c.) Compare this solution to the one in Figure 4.13 (p. 176). How many P and V operations are executed by each process in each solution in the best case? In the worst case?

**Programs:** Submit the program solutions electronically. See the end of this assignment for information on electronic turnin. The programming exercises are worth 64 points.

## Programming Exercises:

**Due:** Friday, March 2nd, 7 pm.

Write two bag-of-tasks solutions to the Quicksort problem: one in MPD and the other in C using pthreads.

**Common details:** Use the same algorithm for both solutions. This will allow you to better compare the execution times of MPD vs. C+threads. You are not required to use the quicksort algorithm on the MPD web page. As discussed before, it has several problems. You can use a quicksort from another source (document the source), or write your own variation. (I finally gave up on trying to get the algorithm on the MPD web page to work correctly in all cases, and switched to an algorithm from a data structures textbook). The main point: use the same algorithm for both programs.

Write your solutions so they work with 1 to 16 workers. The number of workers will be specified on the command-line. The other command-line arguments remain the same as in the previous assignment. The number of workers will be an additional argument at the end. For example:

```
sort-bag 5 gettysburg.txt getty.out 3
sort-tasks 5 gettysburg.txt getty.out 3
```

where the 5 is the cutoff value for using an n-squared sort, gettysburg.txt is the input file, getty.out is where the sorted output will be written, and there will be 3 workers.

Print to stdout the time, in milliseconds, for performing the sort. Do not include the time to read the unsorted file or the time to print the sorted file.

### MPD notes:

You are not required to use a dynamic linked-list for your bag. That is one way to represent the bag; there are other ways. However, use the same technique in both your MPD and C solutions. What follows here are some notes on how to create a structure in MPD and how to do dynamic memory allocation.

MPD supports a record data structure, which is analogous to a struct in C. This can be used to create an array of records, a linked-list of records, etc. Declarations of records start by declaring a new type. For example:

```
type person = rec( string[40] name, int height, int weight);
```

You can then declare individual instances of person:

```
person me, you.
```

You use 'dot' notation to reference individual fields of a record:

```
me.height = 185;
write("My weight is:", me.weight);
```

MPD also supports pointers as a type (in case you want to build a linked list for your bag):

```
ptr int qPtr;      # defines qPtr as a pointer to an integer
int xray;
qPtr = @xray;     # @ provides the address of its operand
```

```
qPtr^ = 17;          # will make 17 the value stored in xray
```

For a pointer to a record, you have to declare a new type:

```
type personPtr = ptr person;
personPtr people = null; # use null for an empty pointer.
```

Use the functions `new()` and `free()` to dynamically allocate and deallocate memory.

```
people = new(person);
people^.name = "Patrick";
people^.height = 185;
```

To use `free()`, you supply a pointer:

```
free(people);
```

### Naming:

Name your MPD version: `sort-bag.mpd`.

Name your C+threads version: `sort-tasks.c`.

### Timing tests:

Run timing tests on voltron for both programs. Run timing tests for 1, 2, 3, 4, etc. workers. Some questions that your timing results should be able to answer:

Do you see speed up as the number of workers increases?

How do the times for your MPD solution compare with your times for C+threads?

Do the times start to increase as you go beyond 4 workers?

Submit a file (using `turnin`) named `timing.txt`. This should contain the results of your timing runs. We should be able to figure out the answers to the above questions from the timing results that you supply.

**Turnin:** Use the `turnin` program to turn in three files: `sort-bag.mpd`, `sort-tasks.c`, `timing.txt`. The command is:

```
turnin 422assign2 sort-bag.mpd sort-tasks.c timing.txt
```

If you are using `turnin` for the answers to problems 1 to 3, the `turnin` command is:

```
turnin 422assign2 answers.pdf
```

See the man page for the `turnin` program for details on what `turnin` can do and how you can confirm that your file was turned in.

## Homework 2 Solutions

### Problem 1: Exercise 3.2, page 142

Suppose a computer has atomic decrement **DEC** and increment **INC** instructions that also return the value of the sign bit of the result. In particular, the decrement instruction has the following effect:

```
DEC(var, sign):
  < var = var - 1;
    if (var >= 0)
      sign = 0;
    else
      sign = 1; >
```

**INC** is similar, the only difference being that it adds 1 to var.

Using **DEC** and/or **INC**, develop a solution to the critical section problem for n processes. Do not worry about the eventual entry property. Describe clearly how your solution works and why it is correct.

Solution:

Still needed...

### Problem 2: Exercise 4.13, page 194

Consider the following proposal for implementing **await** statements using semaphores:

```
sem e = 1, d = 0; # entry and delay semaphores
int nd = 0;      # delay counter
# implementation of <await (B) S;>
P(e);
while (!B) {
  nd = nd + 1;
  V(e);
  P(d);
  P(e);
}
S;
while (nd > 0) {
  nd = nd - 1;
  V(d);
}
V(e);
```

Does this code ensure that the **await** statement is executed atomically? Does it avoid deadlock? Does it guarantee that **B** is true before **S** is executed? For each of these questions, either give a convincing argument why the answer is “yes”, or give an execution sequence that illustrates why the answer is “no”.

Solution needed:....

**Problem 3: Exercise 4.21, pages 197-8**

Consider the following solution to the readers/writers problem. It employs the same counters and semaphores as in Figure 4.13 (page 176), but uses them differently.

Consider the following writers' preference solution to the readers/writers problem [Courtois et al. 1971]:

<pre> int nr = 0, nw = 0; # numbers of readers and writers sem e = 1;         # mutual exclusion semaphore sem r = 1, w = 0;  # delay semaphores  int dr = 0, dw = 0; # delay counters </pre>	
<pre> process Reader[i = 1 to M] {   while (true) {     P(e);     if (nw == 0) {       nr = nr + 1;       V(r);     }     V(e);     P(r); # await read permission      read the database      P(e);     nr = nr - 1;     if (nr == 0 and dw &gt; 0) {       dw = dw - 1;       nw = nw + 1;       V(w);     }     V(e);   } } </pre>	<pre> process Writer[j = 1 to N] {   while (true) {     P(e);     if (nr == 0 and nw == 0) {       nw = nw + 1;       V(w);     }     else       dw = dw + 1;     V(e);     P(w);      write the database      P(e);     nw = nw - 1;     if (dw &gt; 0) {       dw = dw - 1;       nw = nw + 1;       V(w);     }     else       while (dr &gt; 0 {         dr = dr - 1;         nr = nr + 1;         V(r);       }     V(e);   } } </pre>

a.) Carefully explain how this solution works. What is the role of each semaphore? Show that the solution ensures that writers have exclusive access to the database and a writer excludes readers.

b.) What kind of preference does the solution have? Readers preference? Writers preference? Alternating preference?

c.) Compare this solution to the one in Figure 4.13 (p. 176). How many P and V operations are executed by each process in each solution in the best case? In the worst case?

Solution:

Still needed...

This is from a similar question from last year -- left here in case it helps for formatting the current answer:

Roles of the semaphores:

**m1**: Protects the **nr**, number of readers, variable. Controls reader access of **write** semaphore.

**m2**: Protects the **nw**, number of writers, variable. Controls writer access of **read** semaphore.

**m3**: Controls reader access of the **read** semaphore. This is necessary because we want to allow only one reader to wait on the read semaphore at any given time. This allows the writer's preference.

**read**: The **read** semaphore is held by the writers whenever there is  $\geq 1$  writers waiting. This stops any readers from trying to get the **write** semaphore. This gives the writers preference. As soon as one writer comes in and gets **P(read)**, no more readers can add to **nr** and the current readers will eventually finish and let go of **P(write)** allowing the writers to continue. Because **m3** allows only one reader to be waiting on **P(read)**, the writer will get the **read** semaphore in constant time.

**write**: The **write** semaphore protects the database such that no reader reads while a writer is writing, and no writer writes at the same time as another writer.

### Key assertions:

While there are readers reading the database,  $nr \geq 1$ , **P(write)** is held.

The number of readers, **nr**, can only be incremented if there are no writers waiting. Thus, if there are any writers waiting, **nr** can only go down.

While there are writers,  $nw \geq 1$ , no readers will make progress (**read** is held).

If  $nw \geq 1$ , new writers are free to increment **nw**, thus getting ahead of any readers in line.

Because of **m3**, only one reader can be waiting on **P(read)** at any given time. This means that the first writer to come in can halt new readers ( $nr = nr + 1$ ) almost immediately.

Reader processes try to grab **P(write)**, then read from the database, then release **P(write)**. They hold **P(write)** as long as  $nr \geq 1$ , but the key is that as soon as a writer begins execution, it will grab **P(read)**, which will prevent new readers from coming in.

The writer's have the ability to stop new readers from coming in, but readers do not have a similar ability. New writers can keep coming in and no reader will make progress (be able to read the database) until all writers have finished. Thus, in this solution, writer's have preference over readers.