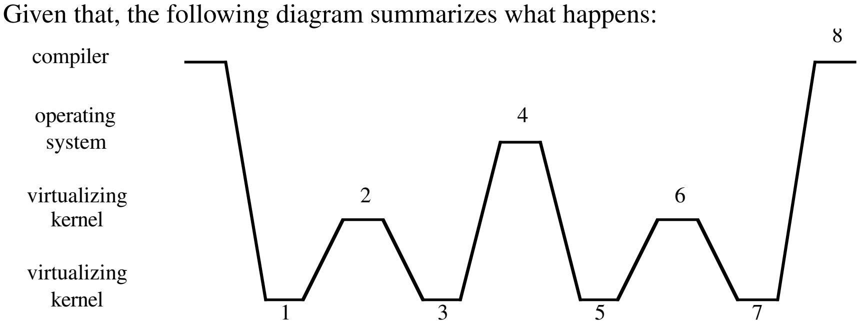
Answers to Homework #1

Due Date: January 16, 2000 **Points**: 60

1. (*20 points*) In the example of virtual machines, with a compiler above an operating system above two levels of virtualizing kernel, how many privileged instructions would be executed at each level if the instruction executed by the compiler can be emulated without use of privileged instructions by the operating system?
Answer: The critical observation is that even if the layer needs no privileged instruction to emulate an instruction in the next upper layer, it needs to return control to that layer, and this returning is a privileged instruction.
Given that, the following diagram summarizes what happens:



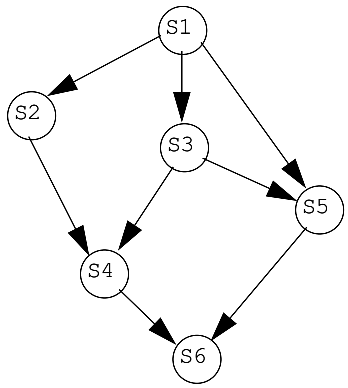
When the compiler tries to execute an instruction, it traps; this trap is acted upon by the lower virtual kernel (1). It determines that the instruction was not privileged, and returns control to the upper virtual kernel (2). That also determines the instruction is not privileged and tries to return control to the operating system; that being a privileged instruction, it traps to the lower virtual kernel (3). The lower virtual kernel updates the upper virtual kernel to make it appear that that layer successfully executed the privileged instruction, and then returns control to that layer. In doing so, control actually passes to the next upper layer, which is the operating system (4). This determines the instruction is to be emulated, and does so. No privileged instructions are involved. It then tries to return control to the compiler, and in doing so executes a privileged instruction.
The operating system therefore causes a trap to the lowermost virtual kernel (5), which determines that the instruction was privileged; then the lower virtual kernel updates the upper virtual kernel to make it appear that that layer successfully trapped the privileged instruction (that is, the operating system trapped to the upper, rather than the lower, virtual kernel), and then returns control to that layer (6). That layer determines that the operating system tried to return control to the compiler, and updates the operating system to make it appear that the operating system had done so. It then tries to return control to the operating system (and hence to the compiler); but this traps to the lower virtual kernel, as the instruction is privileged (7). The lower virtual kernel updates the upper virtual kernel to make it appear that that program had correctly executed the privileged instruction; it then returns control to the upper virtual kernel, which has caused control to be passed to the operating system, which has caused control to be passed to the compiler (8).
That means the lower virtual kernel executes 4 “return control” instructions, the upper virtual kernel 2, and the operating system 1. (In the diagram, each line following a short straight line represents an attempt to execute a privileged call.)

2. (*20 points*) Is the following program properly nested? Please either show that it is by rewriting the program using **parbegin** … **parend**, or prove that it is not properly nested. (The S*i* are statements.)

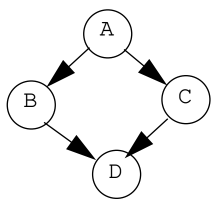
```
        c4 := 2;
        c6 := 2;
        S1;
        fork p1;
        S3;
        fork p2;
        S5;
        goto p4;
p1:     S2;
        goto p2;
p2:     join c4, p3;
```

```
        quit
p3:     S4;
p4:     join c6, p5;
        quit
p5:     S6
        quit
```

Answer: The precedence graph is the following:



For the precedence graph to be properly nested, it must have a sequential or parallel "innermost block". If it has a sequential innermost block, there will be some subgraph with an edge connecting two vertices. The source vertex will have one outgoing edge, and the destination vertex will have one incoming edge. No such subgraph of the above precedence graph exists. If the graph has a parallel innermost block, there will be some subgraph like:



In this subgraph, the vertices corresponding to B and C must have exactly one incoming edge (from the same source) and one outgoing edge (to the same destination). Checking the above precedence graph, we see there is no subgraph with this format. Hence the subgraph is that of a program that is not properly nested. Thus the original program is not properly nested and so cannot be written using **parbegin** … **parend**.

3. (*20 points*) Synchronization within monitors uses condition variables and two special operators, **wait** and **signal**. A more general form of synchronization would be to have a single primitive, **waituntil**, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

    waituntil *x* < 0 **or** *y* + *z* < *n*

   The **signal** primitive would no longer be needed.

   a. Use this more general form to solve the producer-consumer problem.
   b. Is this construct more, less, or as, powerful as using **wait** and **signal** (in Hoare's version of monitors)?
   c. Why do you think it is not used in practice?

   Answer:

   a. This solution is based on the monitor solution given in class.

```
buffer: monitor;
var   slots: array [0..n-1] of item;
              count, in, out: integer;
procedure entry deposit(data: item);
begin
        waituntil count <> n;
        slots[in] := data;
        in := in + 1 mod n;
        count := count + 1;
```

```
10  end;
11  procedure entry extract(var data: item);
12  begin
13        waituntil count <> 0;
14        data := slots[out];
15        out := out + 1 mod n;
16        count := count - 1;
17  end;
18  begin
19        count := 0; in := 0; out := 0;
20  end.
```

The idea is the same; the only difference is that the monitor checks whether the buffer is full in *deposit* (line 6), or empty in *extract* (line 13), by looking at the value of the *count* variable directly instead of waiting for signals to indicate that the appropriate condition is satisfied.

b. We need to show that we can implement the **wait** and **signal** primitives using **waituntil**. For each condition variable *x*, we make two new variables local to the monitor.

```
var x_waiting: integer, x_signaled: boolean;
```

They are initialized as follows.

```
x_waiting := 0;
x_signaled := false;
```

Each occurrence of `x.wait` is replaced with

```
x_waiting := x_waiting + 1;
waituntil x_signaled;
x_signaled := false;
```

Each occurrence of `x.signal` is replaced with

```
if x_waiting > 0 then begin
      x_waiting := x_waiting - 1;
      x_signaled := true;
end;
```

Thus, **waituntil** is at least as general as Hoare's scheme.

Conversely, we can implement **waituntil** using **wait** and **signal** as follows: for each statement of the form

```
waituntil expr
```

where *expr* is a boolean expression of variables $x_1, x_2, ..., x_n$, we make a new condition variable *x*. We replace the **waituntil** statement with

```
if not expr then x.wait;
```

After every statement in the monitor that assigns one of the variables $x_1, x_2, ..., x_n$, we insert

```
if expr then x.signal;
```

Thus, **waituntil** is equivalent to Hoare's scheme, not more general, since each can be expressed in terms of the other.

c. The implementation of **waituntil** in terms of **wait** and **signal** given in part (b) shows what the problem is. There could be a tremendous amount of time spent in rechecking the boolean expression after every assignment statement that could affect its value. Thus, **waituntil** is far less efficient than Hoare's scheme.