Before The Test:

Exam 1 Preparation and Signals
Recitation for ECES 338
March 1, 1999

The Bakery Algorithm Revisited

In class, Tekin presented the *Bakery Algorithm*, with an eye towards preparing for the exam. The text of the algorithm follows:

```
repeat
    choosing[i] := true;
    number[i] := max (number[0], number[1], ..., number[n - 1]) + 1;
    choosing[i] := false;

    for j := 0 to n - 1
        do begin
            while choosing[j] do no-op;
            while number[j] <> 0
                and (number[j], j) < (number[i], i) do no-op;
        end;

    critical section

    number[i] := 0;

    remainder section

until false;
```

Now, the question (as it could appear on the test), is what happens to the algorithm if the first *while* loop is removed?

To answer this question (or any question of this type), it's best to think of an example that demonstrates that something breaks (It's usually pretty safe to assume that algorithms don't include non-essential code). So then, consider this example:

Time:	Action:
1	`P0` executes the `max()` function, adds 1 (total of 1), but **doesn't** assign this value to its array (it is suspended by the scheduler).
2	`P1` executes the `max()` function, adds 1 (total of 1 – remember, `P0` hasn't assigned it's value yet), and **does** assign the value.
3	`P1` continues execution, and eventually enters the *critical section*.
4	`P0` resumes execution, assigns it's number value, and proceeds into the `for` loop.

5	When `j` is zero, `number[0]` will be 1, thus evaluating to true. The final expression, `(number[0], 0) < (number[0], 0)` will be false, so the for loop will proceed to when `j` equals 1.
6	At `j = 1`, `number[1]` will be 1, so the first part of the while loop will evaluate to true. Then, the second part of the while will be evaluated. The expression `(number[1], 1) < (number[0], 0)` will be false, because `1 == 1`, but 1 is not greater than 0. So, `P0` will enter the critical section even though `P1` is still in it.

Signals

- Signals are asynchronous event messages that can be used to notify a process that something has occurred. These are akin to hardware interrupts, only they occur in software. All system signals (as defined in `<sys/signal.h>`) have a default behavior. Thus, if you do nothing, your program will adhere to this default behavior. The default behaviors are explained in section 5 of the manual page for `signal`. It is also possible to setup your process so that it ignores all signals (with the exception of `SIGSTOP` and `SIGKILL`), so that processing can continue uninterrupted. Finally, you can catch a signal, so that special interrupt-handling code (of your own design) can be executed whenever the specified signal is received.

- It is important to note that you are **not** required to use signals in your programs. Using signals in your programs will be considered to be a form of **error-handling**, which can give you back points that you have lost in other areas. Thus, using signals (as well as `perror`) is highly **recommended**.

- `int kill (pid_t pid, int sig)`
  - You can send a signal to a process by using the `kill()` system call. This system call is very straightforward – you simply give it a process ID, and an integer signal to send. You can play with the value of `pid`, in order to send the signal to more than one process at a time. This function will return `0` if successful, or `-1` if an error occurred.

- `void (*signal (int sig, void (*disp) (int))) (int)`
  - In order to catch (or ignore) a signal, the `signal()` system call must be used. In essence, this function will hook-up the reception of a certain signal to a function that you define. So, whenever a signal of the given type is received, your program will automatically execute the code in your function. The function prototype for `signal()` is rather confusing, but all you need to know is that the first parameter should be a signal, and the second should be a pointer to a function.

    Example:

    ```
    signal (SIGINT, sigint_handler);       /* Ctrl-C */
    ```

    Typically, this call will be located near the top of your `main()` function, so that the specified signal will be re-routed as early as possible. Furthermore, after this function is called, your program will now execute the function

`sigint_handler()` whenever a `Control-C` character is input from the terminal. Here is an example of a signal handling function:

```
void sigint_handler (int incoming_signal)
{
        fprintf (stderr, "SIGINT received, exiting.\n");
        fflush (NULL);        /* Flush all buffered I/O. */

        /* Remove any semaphores/shared memory/etc. */
        /* Clean-up any global open filehandles, etc. */

        exit (0);
} /* End sigint_handler(). */
```

In this example function, a message will be printed to `stderr`, and then various clean-up operations will be implemented, before the `exit()` call is made. If you wish to simply *ignore* a signal, then you don't need to call `exit()`.