1 Sample user.c file (point 9 on tutorial sheet)

```c
/*
 ** ICOS (C) Kevin Twidle 1997
 ** Operating Systems Concepts (JMC2 / MSc Conv) 2005/'06
 ** Tutorial 6
 ** user.c: User's process for using semaphore to control access to a
 ** shared variable.
 ** Olav Beckmann 2003-11-24
 */

#include <icos/user.h>

semaphore s;

int global_shared = 'x';

/*
 * set_to_y
 * This function sets the shared variable to 'y' and then makes
 * the semaphore available.
 */
void set_to_y(void) {
    for (;;) {
        con_outc('1');
        global_shared = 'y';
        V(&s);
        delay(1);
    }
}

/*
 * print_value
 * This function prints the value of the shared variable, but only
 * when the semaphore indicates that it has been correctly set to
 * y. We release the semaphore afterwards to make sure that it can be
 * accessed by other processes.
 */
void print_value(void) {
    for (;;) {
        con_outc('2');
        P(&s);
        con_outc(global_shared);
        V(&s);
        delay(1);
    }
}

/*
 * set_to_n
 * This is a function that spoils the shared variable by setting it
 * to 'n'. The function takes but does not release the semaphore, thus
 * guaranteeing that it the value of the shared variable cannot be
 * printed until it has been reset.
 */
void set_to_n(void) {
    for (;;) {
        con_outc('3');
        P(&s);
        global_shared = 'n';
    }
}
*/
```
This exercise is meant to give you some simple experience of programming with semaphores in Simple Kernel.

- The basic idea is that the semaphore gets set by the function set_to_n, blocking access to the shared variable by print_value, and released by set_to_y, freeing access.

- Note that there are equal numbers of P and V operations in the program.

- This solution would not ensure that only y is printed without the delay() calls in each process: with no delay(), process set_to_y could iterate more than once, raising the value of the semaphore to a value higher than 1, which would mean that multiple P operations could take place before the value of the semaphore becomes zero and blocks a process.

- As an alternative to calling delay() in each process, we could have modified the semaphore mechanism in sem.c to prevent semaphores from reaching a value higher than 1.

- The scheduling priorities of the processes are equal here, if they were not, we would also have to consider the possibility of processes that free the semaphore being starved of access, thus deadlocking the entire system.