Assignment 5 (Inter-process communication)

24 March 2011

due 6 April at noon

In this assignment, we implement four system calls for allowing integer variables to be shared between processes:

```
int Open_Shared_Var(const char *name, int value);
int Read_Shared_Var(int var);
void Write_Shared_Var(int var, int value);
void Close_Shared_Var(int var);
```

Shared variables have globally unique names, so cooperating processes just need to know the names of the variables they are sharing. The first time a process opens a shared variable with a particular name, the kernel will set aside some space for it and return a handle (an index into an array kept in kernel memory). Other processes that open the variable with the same name will receive a handle to the same variable. Writes to the variable by one process can be read by other processes.

The main system call implementation details have been done for you… you did something similar in the last assignment. Now you just must implement these system calls, in `a5/src/geekos/syscall.c`. Look for the `SHARED VARIABLE IMPLEMENTATION` comment in that file.

Each shared variable is represented in the kernel by the following structure:

```
struct Shared_Var
{
   bool  free;    /* is this var free, or in use?  */
   char* name;    /* what is name of this var? */
   int   refs;    /* how many processes reference it? */
   int   value;   /* current value of var */
};
```

The kernel has a global array of some fixed number of these variables available for use by processes:

```
#define MAX_SHARED_VARS 25
struct Shared_Var g_sharedVars[MAX_SHARED_VARS];
```

The shared variables are initialized by the function `Init_SharedVars`, which does something like this for each variable:

```
g_sharedVars[i].free = true;
g_sharedVars[i].name = NULL;
g_sharedVars[i].refs = 0;
g_sharedVars[i].value = 0;
```

Your task is to implement the four system calls. **Open** is the hardest, followed by **close,*** **then *read** and **write.** Below I have some detailed specifications for all of them.

There are two user-mode test programs you can use. Typing `work` at the shell prompt will run the two counter processes, to 800 and 900. So ideally we'd like the counter to reach 1700, but you will find that it comes well short of that, and varies on each run. We'll get into that later. The other test program is called `hello`, and it should always produce the same output:

```
$ hello
vc == 8
vb == 10
va == 7
```

Also, you can use `Print()` to help you figure out what's going inside the system calls. Remember, it works just like `printf` in standard C, with percent codes like `%d`, `%s`, etc.

Open

```
/* Open (and possibly create) a named shared variable.
 * Params:
 *   state->ebx: user pointer to string containing variable name
 *   state->ecx: length of variable name
 *  state->edx: initial value for variable (if it doesn't exist yet)
 * Returns:
 *   the index of the shared variable
 */
static int Sys_OpenSharedVar(struct Interrupt_State* state)
```

One of the first things we need to do is copy the name string from user space to kernel space. Fortunately, there is a provided function for this, which makes it relatively easy:

```
/* Copy name from user to kernel space */
char* name = Malloc( state->ecx + 1 );
Copy_From_User( name, state->ebx, state->ecx );
name[state->ecx] = '\0';
```

Following that, there are two phases. First, we loop through all the shared variables in the array, looking for a variable with a matching name. The kernel provides a `strcmp` function to compare strings; it works just like in C:

```
strcmp( s1, s2 ) == 0
```

means they are the same. If you find an existing shared variable with the desired name, just increment its `refs` field and return its index.

Otherwise, we need to create a variable with this name. Now we loop through the array again, just looking for a free variable (check the `free` flag in `g_sharedVars[i]`). Once you find a free variable, we have to initialize it. Set its `free` flag to false, initialize its `name`, set `refs` to 1, and set its initial `value` from `state->edx`.

If you don't find any free variable, then we're in trouble. But we're not supporting proper error-handling at the moment, so just print an error message and return zero.

Read

```
/* Read from a shared variable.
 * Params:
 *   state->ebx: index of the shared variable
 * Returns:
 *   the value of the shared variable
 */
static int Sys_ReadSharedVar(struct Interrupt_State* state)
```

This is easy, just return the value of the indicated variable in the global array. You probably should do some sanity checking to make sure that `state->ebx` is within the bounds of the array, and that the indicated shared variable is not free. (It would be an error to read from a shared variable after it has been closed.)

Write

```
/* Write to a shared variable.
 * Params:
 *   state->ebx: index of the shared variable
 *   state->ecx: its new value
 * Returns:
 *   always returns zero.
 */
static int Sys_WriteSharedVar(struct Interrupt_State* state)
```

Almost as easy as reading the variable, but you set its new value from `state->ecx` and always return zero. You should do the same sanity checking as for the read.

Close

```
/* Closes (and possibly deallocates) a shared variable.
 * Params:
 *   state->ebx: index of the shared variable
 * Returns:
 *   always returns zero.
 */
static int Sys_CloseSharedVar(struct Interrupt_State* state)
```

Again, do sanity checks to make sure that `state->ebx` is a valid index, and indicates a variable that is not already freed. If everything is okay, then decrement the `refs` field. If `refs` reaches zero, it means that we're the last process to close this shared variable, so we should deallocate it: Set `free` to true, set `name` to `NULL` and zero out the integer fields. This allows the shared variable location to be reused the next time we open one.

Finally

For full credit, you should be sure to `Free()` the name strings allocated with `Malloc()` in `Sys_OpenSharedVar`. But be careful! If the pointer gets copied into the `g_sharedVars` array, then we cannot free it until that shared variable is deallocated. Leave this until the end, because calling `Free()` in the wrong places can cause obscure and hard-to-trace bugs.

That's it! Make sure your kernel compiles, and test it with the `work` and `hello` programs. Commit your changes before the deadline.