Q1 2024-2025 SO2 Project

The professor Baka Baka likes the potential of our Zeos system and wants to implement some kind of video game like the Dangerous Dave(™) game in which different objects are displayed on the screen and you control your player with the keyboard. But, he realizes that Zeos currently lacks the required support to do it effectively and wants you to improve it.

Videogame architecture

There will be three running threads to manage the videogame. One thread of the videogame is responsible for reading the keyboard. Another thread will manage the movement of enemies, the placement of the player and finally the last one will draw the resulting state on the screen.

Notice that in ZeOS, a screen can be viewed as a matrix of 80 columns by 25 rows. Review the implementation of printc to understand how the video memory is accessed.

Keyboard support

A new system call is required:

```
int getKey(char* b, int timeout);
```

It allows a user process to obtain a pressed key and store it in *´b´*. This is a **blocking** function, it will wait until a key is pressed. If no key is pressed after '*timeout'* seconds, then it will unblock the process and return a negative value. The keyboard device support implementation has to store the keystrokes in a circular buffer.

Screen support

Current screen support just writes characters sequentially to the screen, we want to add support to:

- Change the cursor position: `int gotoXY(int x, int y)`
- Change the foreground and background color: `int changeColor(int fg, int bg)`
- Clear screen with a default screen *b* (matrix of 80x25 chars+color, if b is NULL just clear screen with an empty char) : `int clrscr(char* b)`

Thread support

In order to use the keyboard support from the previous section and do other work while blocked, we need some thread support. Therefore we want to add the system call:

```
int threadCreateWithStack( void (*function)(void* arg), int N, void* parameter )
```

This creates a new thread with a dynamically allocated stack size of 4096**N* bytes that will execute function *'function'* passing it the parameter ' *parameter'* as in '*function(parameter)*'. This thread and its stack must be freed after calling the system call:

```
void exit(void)
```

This call must be called to finish any running thread. Returning from a thread without calling this function is undefined behavior.

Synchronization support

In order to allow different threads to synchronize each other, we want to add semaphores support, therefore implement the following system calls:

```
sem_t* semCreate(int initial_value);
int semWait(sem_t* s);
int semSignal(sem_t* s);
int semDestroy(sem_t* s);
```

These functions will create an initial semaphore with an initial counter of *initial_value*; decrement the semaphore's counter and block the current thread if the counter is negative; increase the semaphores's counter and unblock the first blocked thread in the semaphore's queue; and destroy the semaphore (only the thread that created a semaphore can destroy it) respectively. The returned '*sem_t*' address can not be used from user space and any access to it should fail as the semaphore is a kernel structure and the pointer is just used as a handler to it.

Memory support

Due to the dynamic nature of a game, we will need to create/destroy different objects to represent the elements in the game. Therefore we want a new system call to add new memory regions in the user space:

```
char* memRegGet(int num_pages)
```

This call allocates *num_pages* pages of physical memory and maps them to a consecutive region in the user address space. It returns the initial logical address assigned to the region. This memory region must be inherited by child processes.

```
int memRegDel(char* m)
```

This call deletes a previously allocated memory region *m*, releasing all its resources.

Milestones

1. (1 point) Keyboard support stores keys in a circular buffer.
2. (1 point) Functional *getKey* feature.
3. (,5 points) Functional *gotoXY* feature.
4. (1 points) Functional *changeColor* and *clrscr* features.
5. (2 points) Functional *create_thread* and *exit* system_call.
6. (2 point) Functional synchronization support
7. (1 points) Functional *memRegGet* and *memRegDel* features.
8. (1 point) Functional game using different implemented features.
9. (,5 point) Remove the requirement of *exit* at the finalization of the thread.
10. [Optional] (1 point) Challenge: Implement a user level slab allocator on top of '*memRegGet*' regions.