Q1 2023-2024 SOA Project

The professor Baka Baka likes the potential of our Zeos system and wants to implement some kind of video game like the Lunar Landing(™) 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 two running threads to manage the videogame. One thread will be in charge of reading the user input from the keyboard and updating the state of the videogame. The other thread will draw all the user interface.

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 pollKey(char* b);
```

It allows a user process to obtain a pressed key and store it in *´b´*. This is a **non-blocking** function, if there is no key pressed, then it will return a negative value. The keyboard device support implementation has to store the keystrokes in a circular buffer.

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 to add a new memory region in the user space that works like a stack. This region will be empty at the beginning of the process and it will have a pointer stating the current address. This pointer can only be increased. Increasing the pointer may imply to reserve physical pages and map them to the user space. This memory region must be inherited by child processes.

```
char* memoryInc(int size)
```

This system call increases the pointer by size bytes and returns the initial logical address assigned to use this size bytes region. If there are no free frames or no free logical space it must return a NULL. This call must reserve any required physical page to map the increased region into the user address space.

Screen support

Current screen support just writes characters sequentially to the screen, we want to add support for sprites. A sprite is just a matrix of X rows and Y columns with the same format as the video memory (char+color). Since this an Sprite is a kernel structure, it is defined as:

```
typedef struct {
     int x;     //number of rows
     int y;     //number of columns
     char* content;  //pointer to sprite content matrix(X,Y)
} Sprite;
```

The system call to draw a sprite at a screen position is:

```
int spritePut(int posX, int posY, Sprite* sp);
```

In addition, the following system call to put the cursor at a given position must be implemented:

```
int gotoXY(int posX, int posY);
```

And for changing the color and background of the text:

```
int SetColor(int color, int background);
```

Thread support

In order to implement the architecture of the videogame, as defined at the beginning, thread support must be implemented. Therefore we want to add the system call:

```
int threadCreate( void (*function)(void* arg), void* parameter )
```

This creates a new thread with a fixed stack size of 4096 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 threadExit(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:

```
int semCreate(int initial_value);
```

Creates a semaphore with an initial counter of *initial_value*. When successful, it returns the identifier of the new semaphore.

```
int semWait(int semid);
```

Decrements the counter of the semaphore *semid* and, if negative, blocks the calling thread.

```
int semSignal(int semid);
```

Increments the counter of the semaphore *semid* and unblocks the first blocked thread, if any.

```
int semDestroy(int semid);
```

Destroys the semaphore *semid*. Only the thread that created a semaphore can destroy it.

Semaphores are local to processes.

Milestones

1. (1 point) Keyboard support stores keys in a circular buffer.
2. (1 point) Functional *pollKey* feature.
3. (1 point) Functional *memoryInc* feature.
4. (1 point) Functional *gotoXY* and SetColor features.
5. (0,5 points) Functional *spritePut* feature.
6. (2 points) Functional *threadCreate* and *threadExit* system_call.
7. (2 point) Functional synchronization support.
8. (1 point) Functional game using different implemented features.
9. (0,5 points) Remove the requirement of *threadExit* at the finalization of the thread.
10. [Optional] (1 point) Evaluate the performance of using a 1x1 sprite compared to a 80x25 sprite to clean the background of the screen. To improve the accuracy of the measurement, implement a mechanism to prevent a thread from being preempted in ZeOS (this mechanism can be pure user mode or kernel mode).