Q1 2022-2023 SOA Project

The respectable professor Baka Baka wants to implement a video game on our Zeos system. He would like to design an aim-and-shot game in which a dot moves randomly in the screen. The player aims and shots the dot with the keyboard device. Unfortunately, Zeos currently does not provide the necessary device access support (keyboard and screen).

Videogame architecture

The main thread of the videogame is the responsible of reading the keyboard and update accordingly the status of the match. So, in every iteration, this thread will allocate a memory page for the resulting frame, read the mouse status, draw the resulting frame in the previously allocated page, and pass that frame to a secondary thread. The secondary thread will take the oldest pending frame, draw it on the screen and deallocate the corresponding memory page.

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

Finally, some statistics about the performance of the videogame will be desirable. In the topmost line of the screen, show:

-the crosshair current position
-the frames per second
-the number of pending frames to be drawn

Keyboard support

A new system call must be implemented to read from keyboard:

```
int get_key(char* c);
```

It allows a user process to obtain one of the keys pressed and store it in ´c´. This system call is not blocking and, thus, returns -1 if there is no key available. If different processes execute this call, the keys must be served in strict sequential order (FIFO). The keyboard device support implementation stores the keystrokes in a circular buffer.

Memory management

There must be two new system calls in ZeOS to allocate and deallocate memory pages:

```
void *alloc();
```

```
int dealloc(void *address);
```

The system call alloc will allocate and return the initial address of one free logical page from the user space. If no free logical page is available, (void*)-1 will be returned and errno will hold the error code.

The system call dealloc will deallocate one previously allocated logical free page which starting address is passed as a first parameter.

Thread management

ZeOS must be extended to support threads following the implementation described in lectures. Two system calls for spawning and terminating a thread must be provided:

```
int createthread(int (*function)(void *param), void *param);

int terminatethread();
```

Screen management

A new system call to dump a frame to the video memory must be provided:

```
int dump_screen(void *address);
```

The parameter of the system call is an address corresponding to an 80x25 matrix in which a frame is contained.

Synchronization management

Semaphores will be used to execute certain code fragments in mutual exclusion or block processes. The data structures needed for each semaphore, are basically a counter and a queue for the blocked processes.

The related system calls for semaphores follow:

```
int sem_init (int n_sem, unsigned int value);
```

where:
n_sem: identifier of the semaphore to be initialized
value: initial value of the counter of the semaphore
returns: -1 if error, 0 if OK

This function initializes a semaphore identified by the number *n_sem*. It initializes the semaphore's counter to *value*. At the same time, it initializes the blocked processes queue in this semaphore and the data necessary for its correct use.

The return value 0 indicates a correct execution. If *n_sem* is already initialized, the returned value will be -1.

```
int sem_wait (int n_sem);
```

where:
n_sem: identifier of the semaphore
returns: -1 if error, 0 if OK

If the counter for semaphore *n_sem* is less than or equal to zero, this function will block the process that has called it in this semaphore. If the counter is greater than zero, this function will decrement the value of the semaphore.
A return value 0 indicates a correct execution. If *n_sem* is not a valid identifier for a semaphore, the returned value will be -1.

```
int sem_signal (int n_sem);
```

where:
n_sem: identifier of the semaphore
returns: -1 if error, 0 if OK

This function increments the counter of the semaphore *n_sem*. If there are one or more blocked processes in *n_sem*, this call will unblock the first process.

The return value 0 indicates a correct execution. If *n_sem* is not a valid identifier for a semaphore, the returned value will be -1.

```
int sem_destroy (int n_sem)
```

where:
n_sem: identifier of the semaphore to destroy
returns: -1 if error, 0 if OK

This call destroys the semaphore *n_sem*. The return value 0 indicates a correct execution. If there are blocked processes, they will be unblocked and their corresponding *sem_wait* will return -1.

If *n_sem* is not a valid identifier for a semaphore, or the semaphore is not initialized, the returned value will be -1.

Last comments

There are several details missing in this documentation. Take any decision you want about those missing details. If you are not sure enough about your decision, please, talk to the professor.

Milestones

1. (1 point) Memory management implemented and working.
2. (2 point) Thread management implemented and working.
3. (2 point) Keyboard management implemented and working.
4. (1 point) Screen management implemented and working.
5. (1 point) Synchronization management implemented and working.
6. (2 point) Functional videogame.
7. (1 point) Performance statistics