Q1 2025-2026 SO2 Project

The professor Baka Baka likes the potential of our Zeos system and wants to implement some kind of video game like the Dig Dug(™) 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 score 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 pollEvent(struct event_t *ev);

struct event_t {
      int pressed;     //1 if pressed, 0 if released
      char scancode;
}
```

It allows a user process to obtain an event (a pressed or released key) and store its scancode in structure `ev`. This is a **non-blocking** function, if there is no key pressed, then it will return a zero value. The keyboard device support implementation has to store the scancodes in a circular buffer.

Screen support

Current screen support just writes characters sequentially to the screen, we want to add support for interpreting some of the characters sent to the screen (something similar to https://wiki.bash-hackers.org/scripting/terminalcodes or https://en.wikipedia.org/wiki/ANSI_escape_code). In particular we want (at least) to:

- Change the cursor position
- Remove the current character
- Change the foreground and background color

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 clone(void (*function)(void* arg), void*parameter,
          char* stack)
```

This creates a new thread that will execute function *'function'* passing it the parameter ' *parameter'* as in '*function(parameter)*' using stack available at `stack` (pointing to the topmost address of the memory space set up for the child stack). This thread 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. Just the last thread of a process will release all resources.

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 must 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 create new memory slots in the user space:

```
char* getSlot(int num_bytes)
```

This call allocates enough physical pages to store `num_bytes` bytes and maps them to a consecutive region in the user address space. It returns the first usable logical address assigned to the slot. The memory slots must be inherited by child processes.

```
int delSlot(char* s)
```

This call deletes a previously allocated memory slot `*s*`, releasing all its resources.

Milestones

For every milestone you must deliver:

-The source code with elaborate comments to show that you fully understand what you have done and that you have not used any AI-based tool to write the code.

-A test benchmark that shows the correct behavior of your implementation covering all the cases described in this document

-A text file with a short description of all the decisions taken during the implementation of the feature (if any) and the modifications performed in other parts of the operating system.

1. (1 point) Keyboard support stores keys in a circular buffer.
2. (1 point) Functional *pollEvent* feature.
3. (,5 points) Functional *change cursor position* feature.
4. (,5 points) Functional *remove character* feature.
5. (1 point) Functional change foreground and background color features.
6. (2 points) Functional *clone* and *exit* system calls.
7. (2 point) Functional synchronization support
8. (1 points) Functional *getSlot* and *delSlot* features.
9. (1 point) Functional game using different implemented features.
10. [Optional] (1 point) Challenge: Implement a user level slab allocator on top of '*getSlot*'.