Q2 2024-2025 SOA Project

Professor Baka Baka likes the potential of our Zeos system and wants to implement some kind of video game. He would like to design a Pacman(™) like game (or any other astonishing videogame) in which different objects (ghosts) 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. The main thread of the videogame is responsible for reading the keyboard. For that, it will read the status of the keyboard, update the status of the placement of Pacman and block during a given amount of time for not consuming CPU.

Another thread will manage the movement of enemies and other elements and finally draw the resulting frame on the screen.

Both threads must work in mutual exclusion when updating/accessing the placement of Pacman.

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.

Finally, some statistics about the performance of the videogame will be desirable. In the topmost line of the screen, you must show the frames per second.

Keyboard support

Implement a new syscall to read the keyboard:

```
int GetKeyboardState(char *keyboard);
```

GetKeyboardState returns in *keyboard* the status of all keys in the keyboard. *Keyboard* is an array with as many positions as keys are in the keyboard and, for every position, 1 means if the corresponding key is pressed. GetKeyboardState returns 0 if everything is OK or -1 if there is any error. *Keyboard* is a preallocated user buffer. GetKeyboardState is a not blocking syscall.

To block the current thread, implement the following syscall:

```
int pause(int milliseconds);
```

That blocks the calling thread *milliseconds* milliseconds. This syscall returns 0 if successful.

Screen support

The mechanism to draw a frame to the screen shares a physical page between the process and the operating system. For that, a new system call must be created:

```
void * StartScreen();
```

This system call starts the screen support for the process that calls it. This system call maps a physical page between the operating system and the process and returns the logical address of that physical page or (void*)-1 if not successful. One process may have at most, one of these shared pages.

From the user perspective, this shared page represents the 80x25 screen matrix and the user must write directly to this page the contents of the frame it wants to draw to screen.

The operating system implements a first-free algorithm to map the physical page to the user logical page. To update the screen, every clock tick, the operating system dumps the contents of the shared physical page, if any, for the current process to the screen.

Thread support

To implement threads, you must implement the clone syscall:

```
int clone(int what, void *(*func)(void*), void *param, int stack_size);
```

The first parameter of the syscall, *what*, indicates if either a process (CLONE_PROCESS) or a thread (CLONE_THREAD) must be created. In case a process is created, the other parameters are meaningless. If a new thread must be created, the second parameter indicates the function of the thread, the third is the parameter of the function and the last parameter is the user stack size of the thread.

This system call supersedes fork. So, you must remove sys_fork and do all the process and thread creation in the same system call: sys_clone.

In addition, the scheduler must be changed to implement thread priorities with immediate preemption. For that, the following system call must be implemented to change the priority of a thread:

```
int SetPriority(int priority);
```

This system call sets the priority of the current thread. The higher the value of the *priority* parameter, the higher the priority of this thread.

Semaphores

To implement mutual exclusion, semaphores (local to the process) must be implemented with the following system calls:

```
int sem_init(int value);
int sem_wait(int sem_id);
int sem_post(int sem_id);
int sem_destroy(int sem_id);
```

The sem_init system call creates a semaphore with an initial value and returns its identifier for lately be used in the sem_wait, sem_post and sem_destroy system calls. Remember that a semaphore is a system structure with a counter and a queue of blocked threads.

Milestones

1. (2 points) Functional keyboard feature.
2. (2 points) Functional screen feature.
3. (2 points) Functional thread feature.
4. (2 points) Functional semaphore feature.
5. (2 points) Functional videogame.

NOTE: All Milestones must be tested thoroughly, meaning that all system calls must have a test case to test its correct functionality.