

Recitation Week #8

Project 3 and procfs Programming

Alejandro Cabrera
Operating Systems
COP4610 / CGS5765

Today's Recitation

- Project 3
- procfs Programming

Project 3

- Overview
 - Part 1: Kernel Time
 - Part 2: Elevator Scheduling

Part 1: Kernel Time

- Implement a procfs entry to display the value of **xtime**.
- What's the point?
 - Writing a procfs module doesn't get much easier than this.
- Useful introduction before getting to part 2.

Part 2: Elevator Scheduling

- Implement a kernel module with procfs support to represent an elevator system.
- Implement a set of system calls to control your elevator system from user-space.
- Implement a set of user-space programs to exercise your system.

Why Elevator Scheduling?

- Elevator scheduling is one of the cleanest existing analogies to the storage system of the Linux kernel.
- Requests for reads and writes from programs are accumulated in the storage system.
- These programs issuing requests are the **producers**.
- The **consumers** are the disk-heads, for the case of hard-drives, moving data from sector to sector as needed.

Elevator Scheduling Algorithms

- A scheduling algorithm considers the state of the consumers and all requests and tries to optimize some metric.
- Potential metrics:
 - Throughput: Maximize total requests, minimize processing total time.
 - Priorities: Requests now have deadlines. Maximize number of requests meeting deadlines.
 - Burst throughput: Maximize peak requests that can be handled.
 - Energy: Minimize consumer action.

Elevator Kernel Module

- Needs to:
 - Set up a representation for elevators
 - Set up a representation for requests
 - Implement a scheduling algorithm
 - Implement an init function, a cleanup function.
 - Implement a procfs entry.
- Needs to worry about:
 - Concurrency (not the same as parallelism)

Elevator System Calls

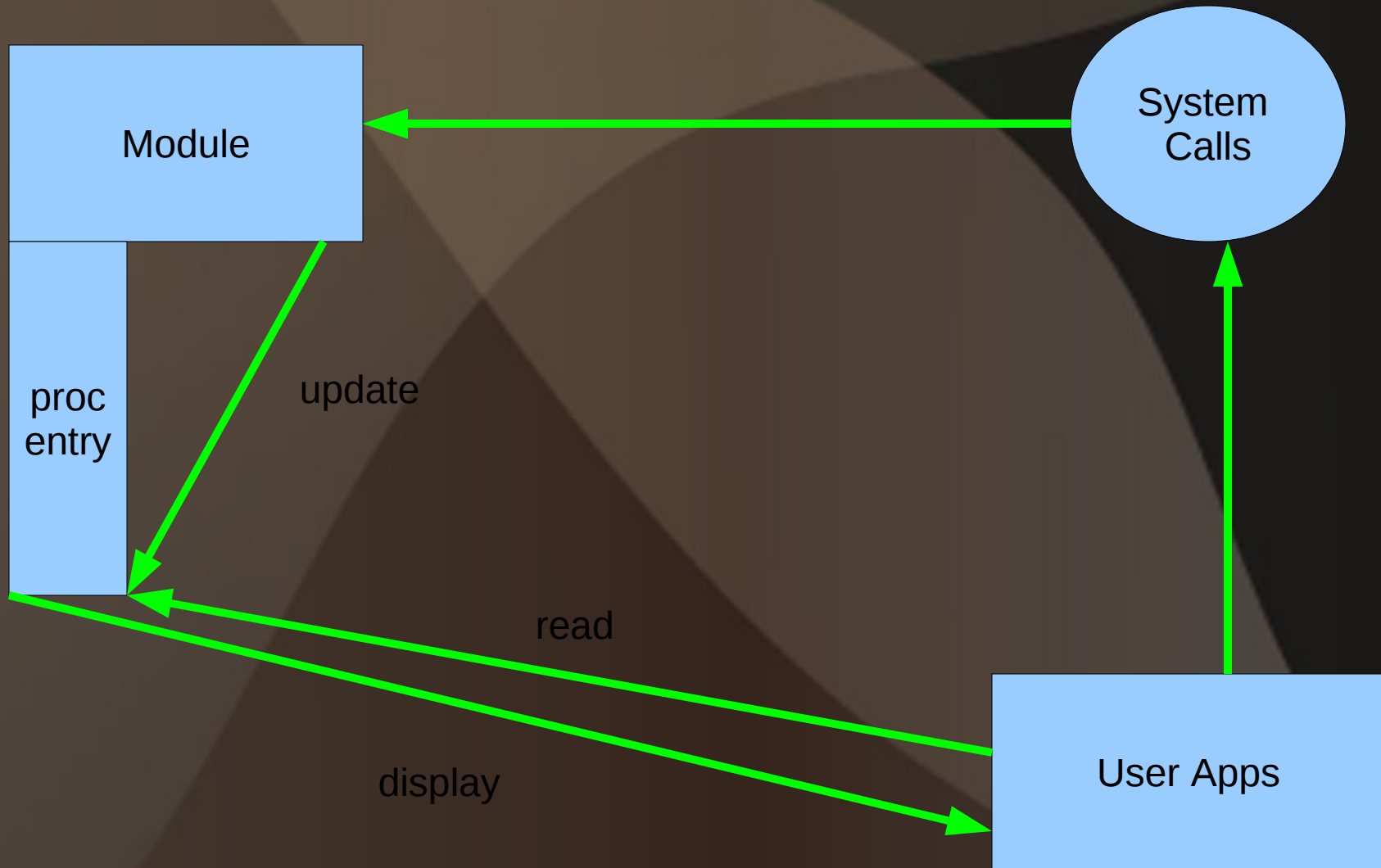
```
int start_elevator(const int
    elevator_number);
int issue_request(const int start_floor,
    const int dest_floor);
int stop_elevator(const int
    elevator_number);
```

- Each system call has a clear responsibility.
- Understand this responsibility perfectly before writing the system call code.

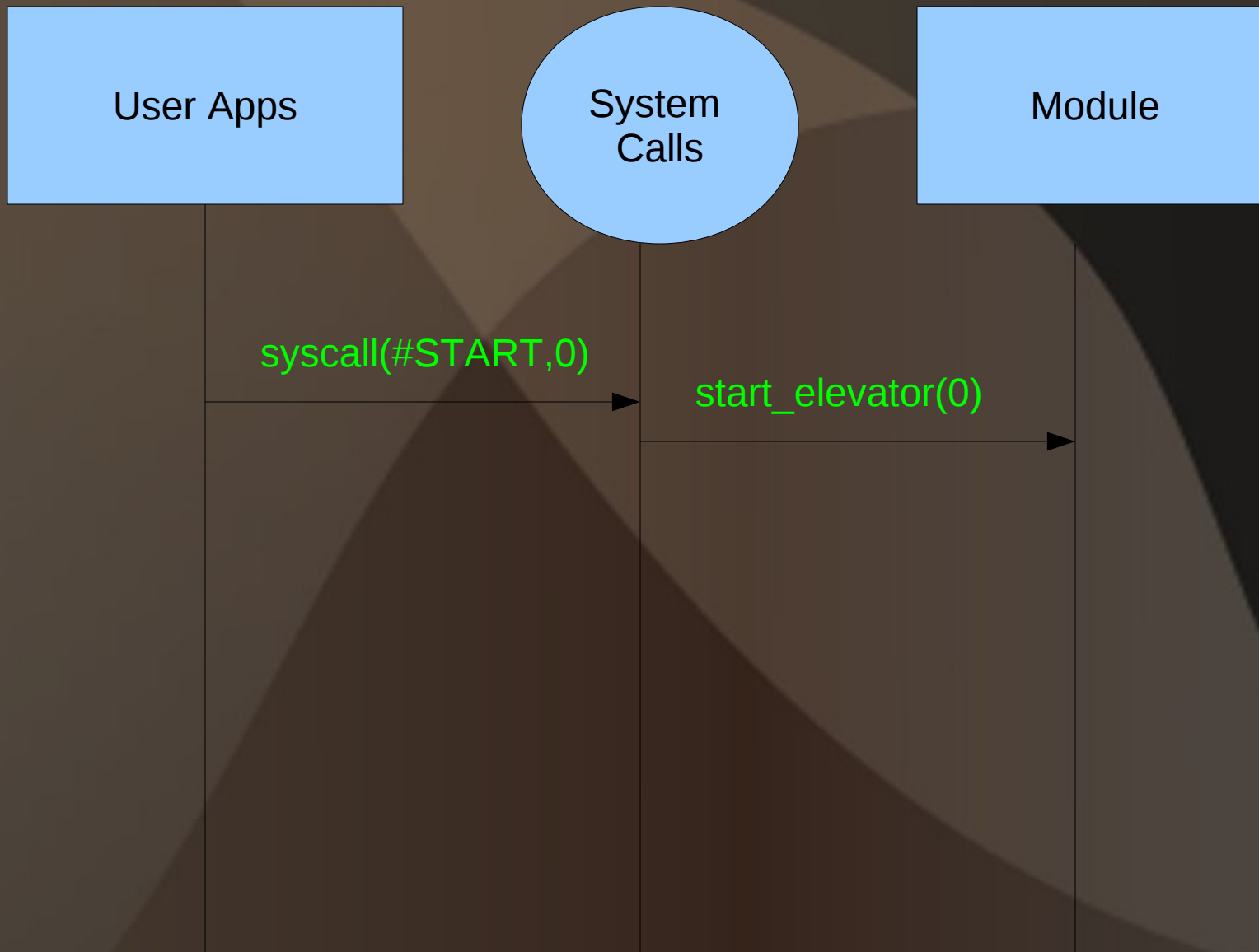
Elevator Applications

- **consumer.c:**
 - Runs in an infinite loop.
 - Issues K requests once per second.
- **producer.c:**
 - Takes an argument telling an elevator to stop or go.

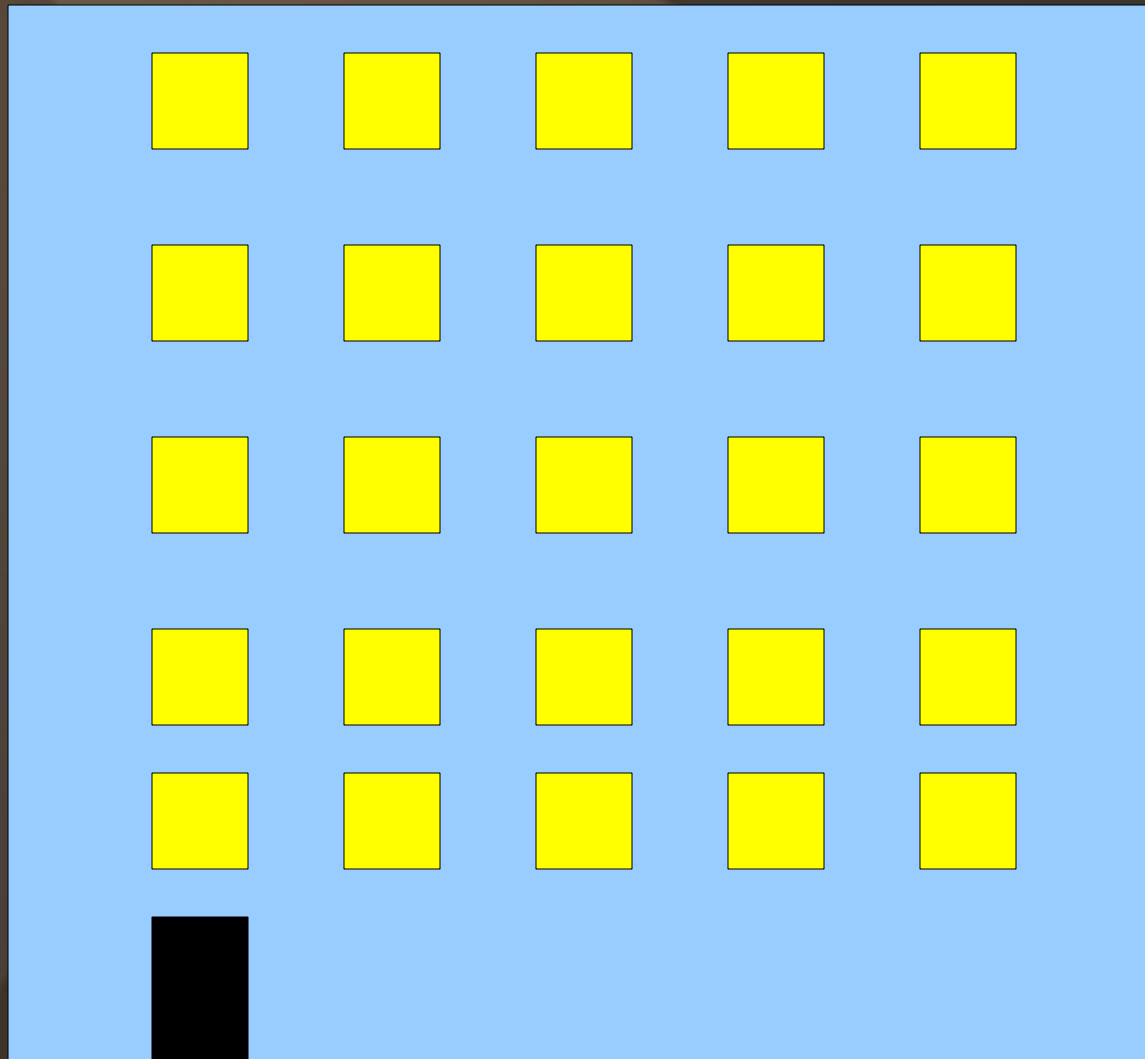
Elevator Component Relationships



Elevator State Diagram

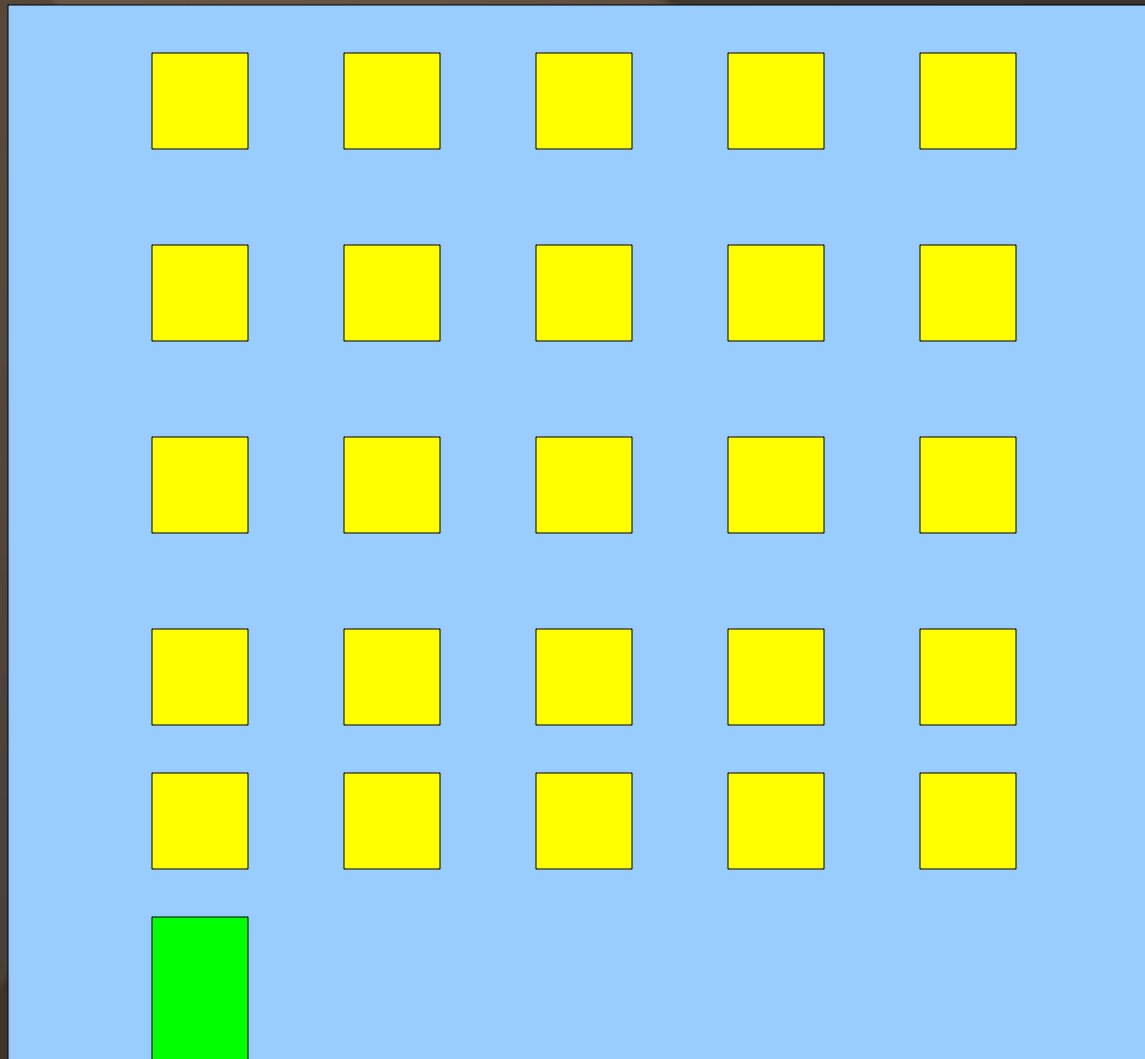


Module Internals

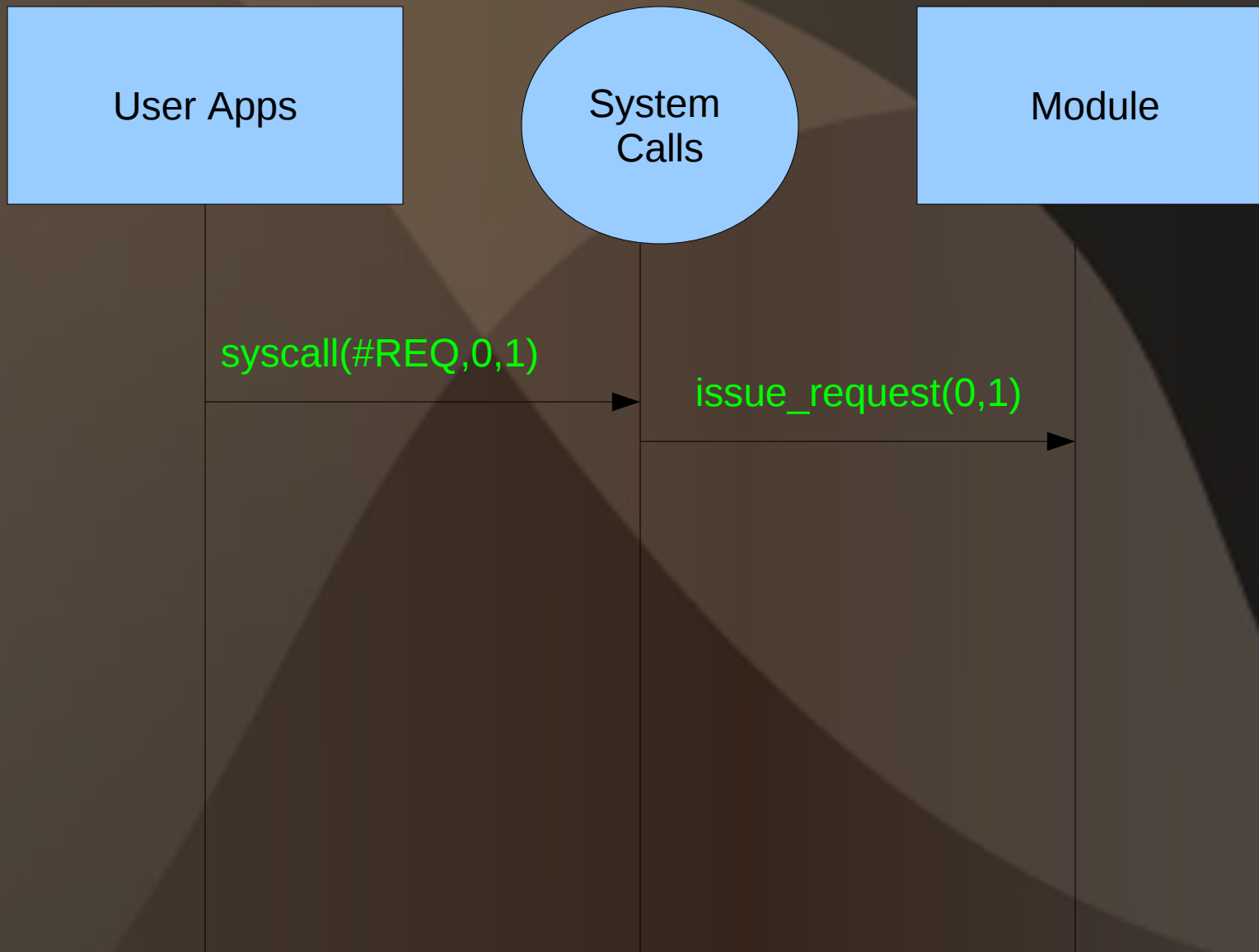


Module Internals

start_elevator(0)

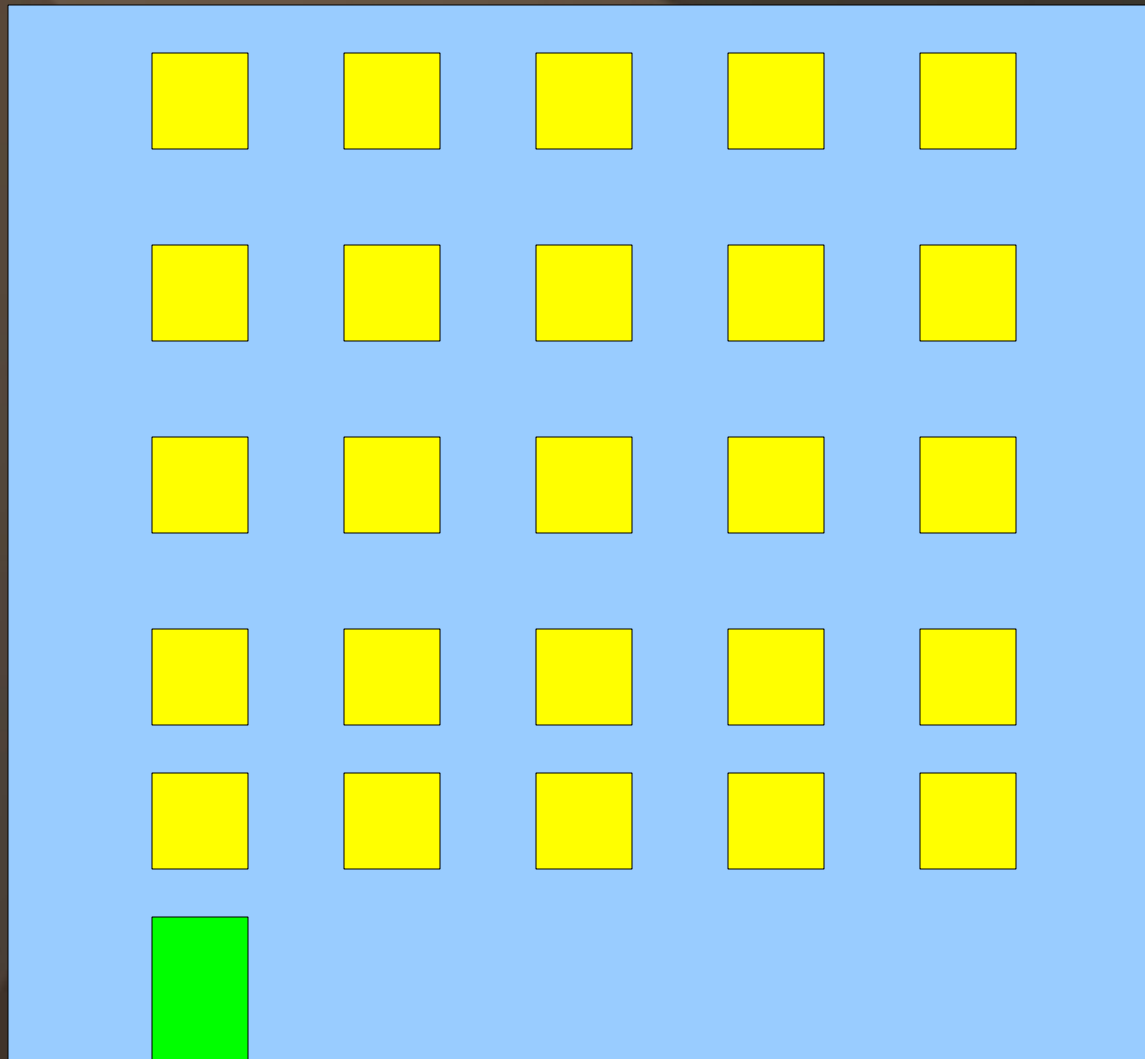


Elevator State Diagram



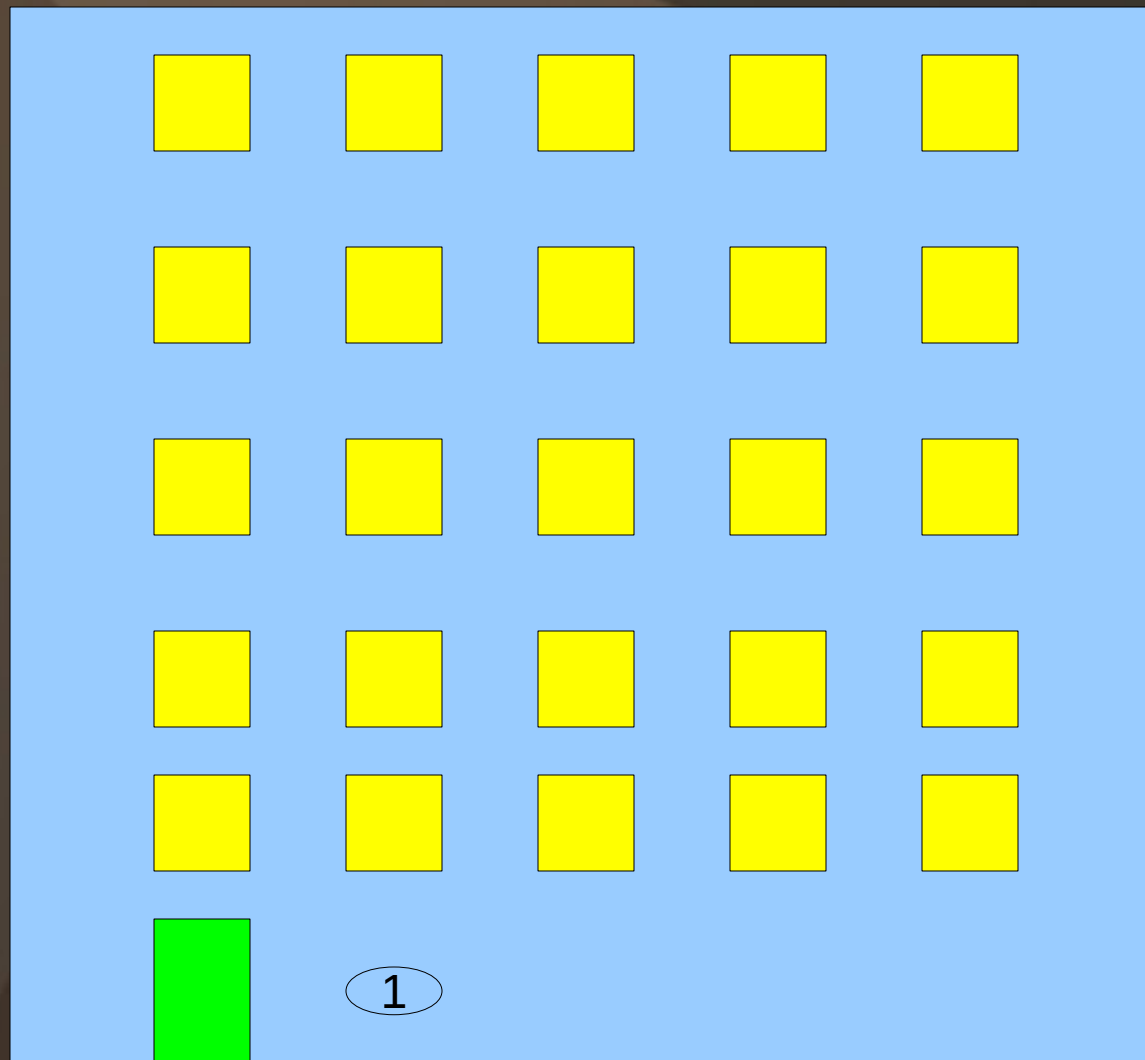
Module Internals

`issue_request(0,1)`

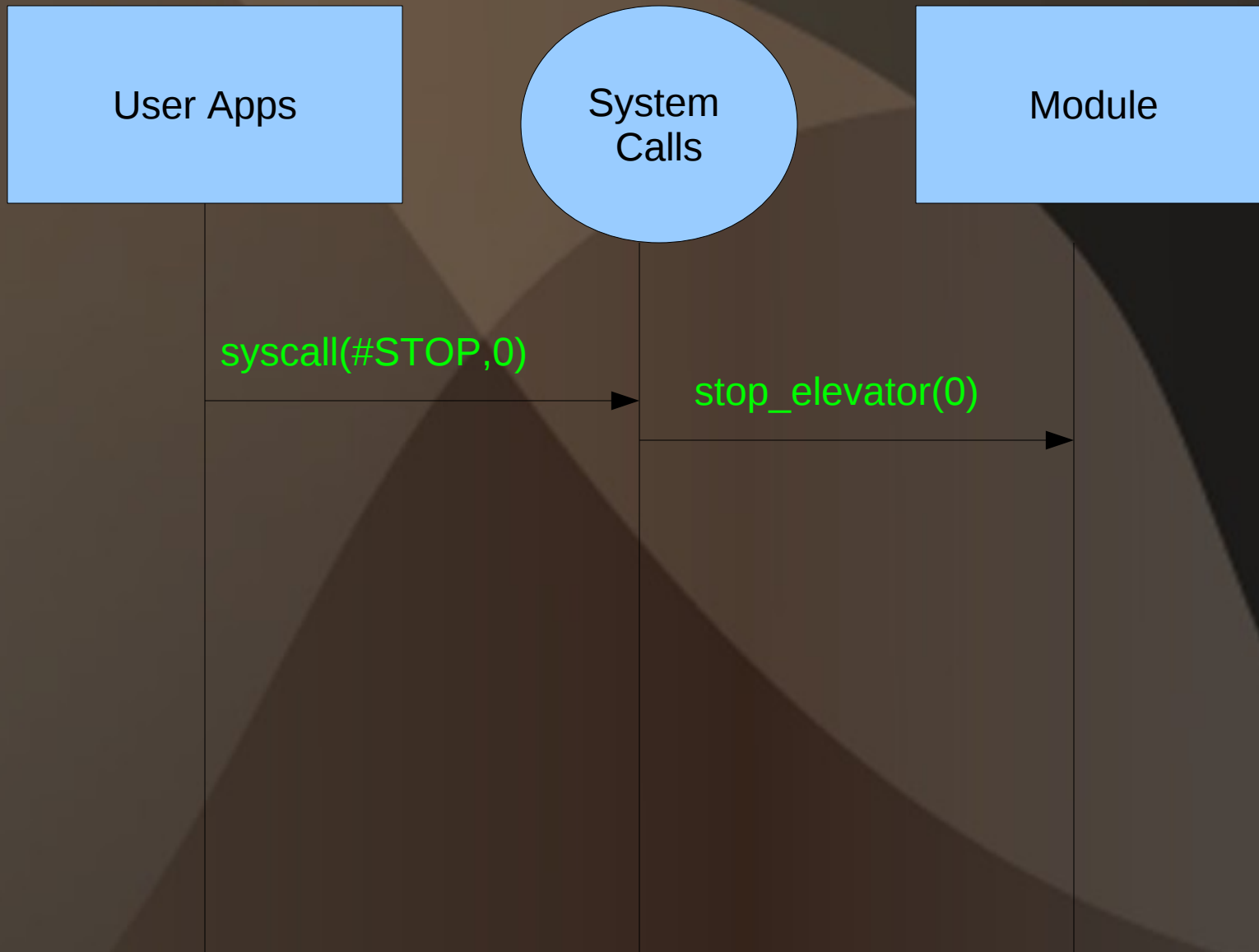


Module Internals

issue_request(0,1)

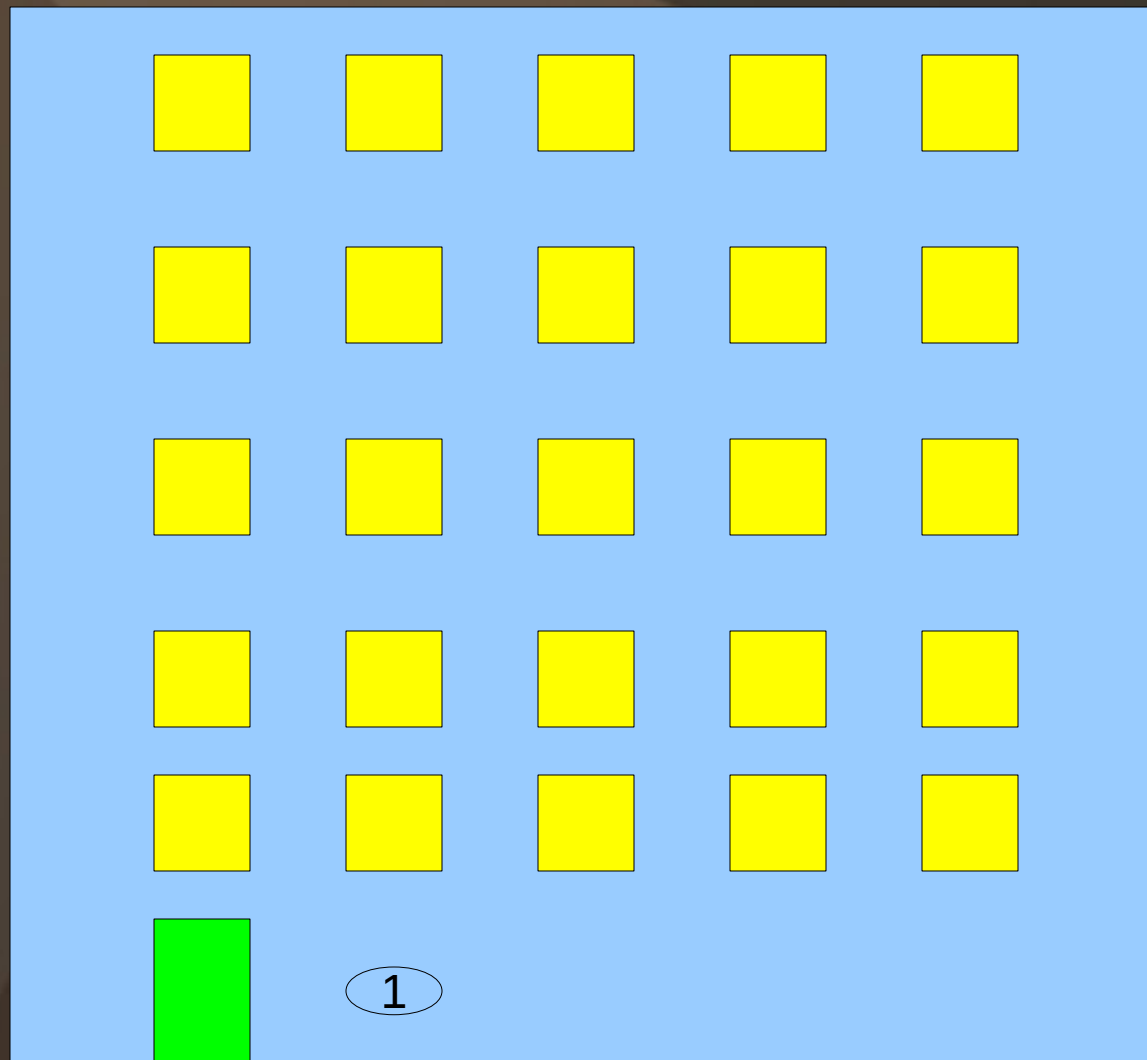


Elevator State Diagram



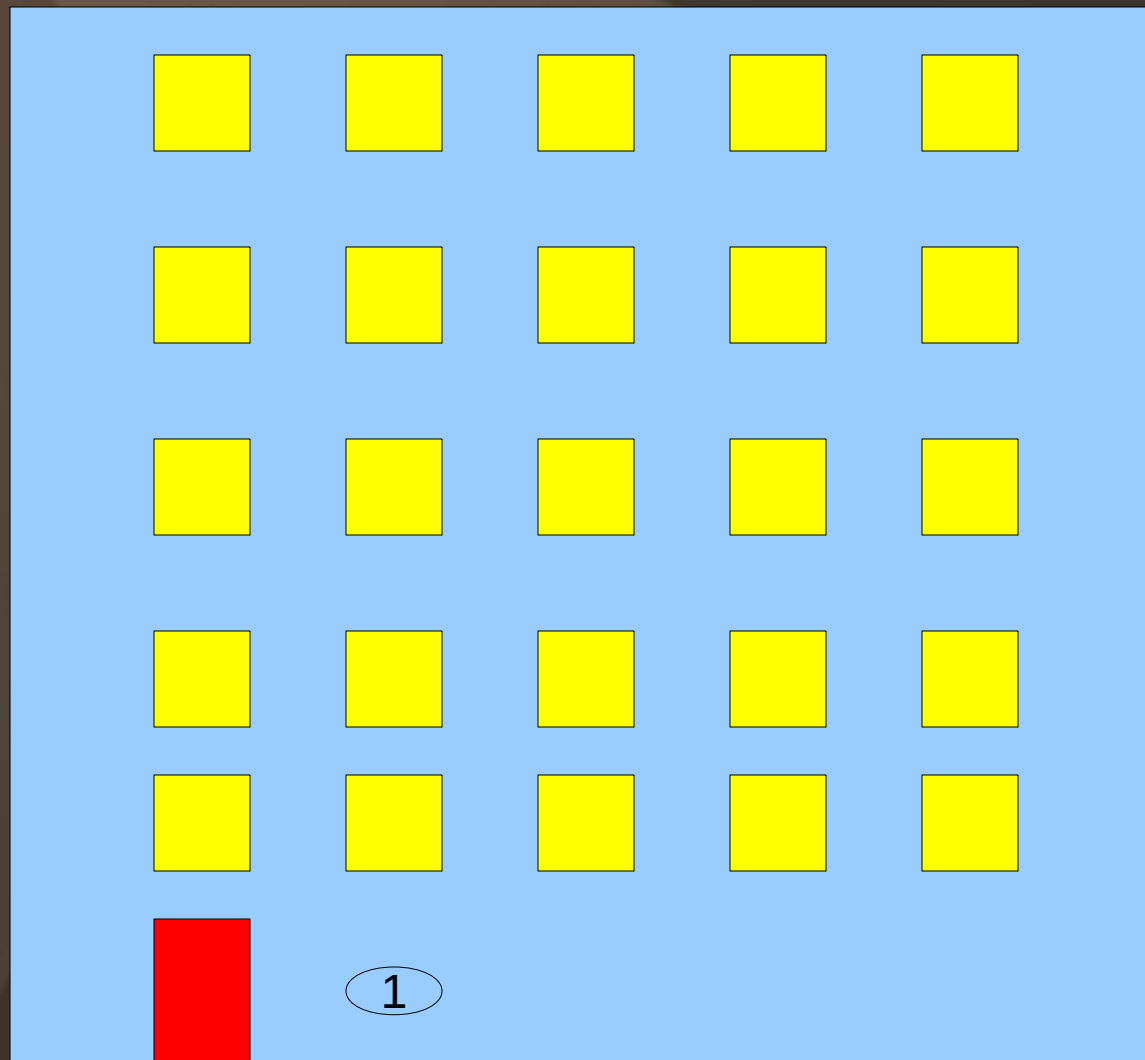
Module Internals

stop_elevator(0)



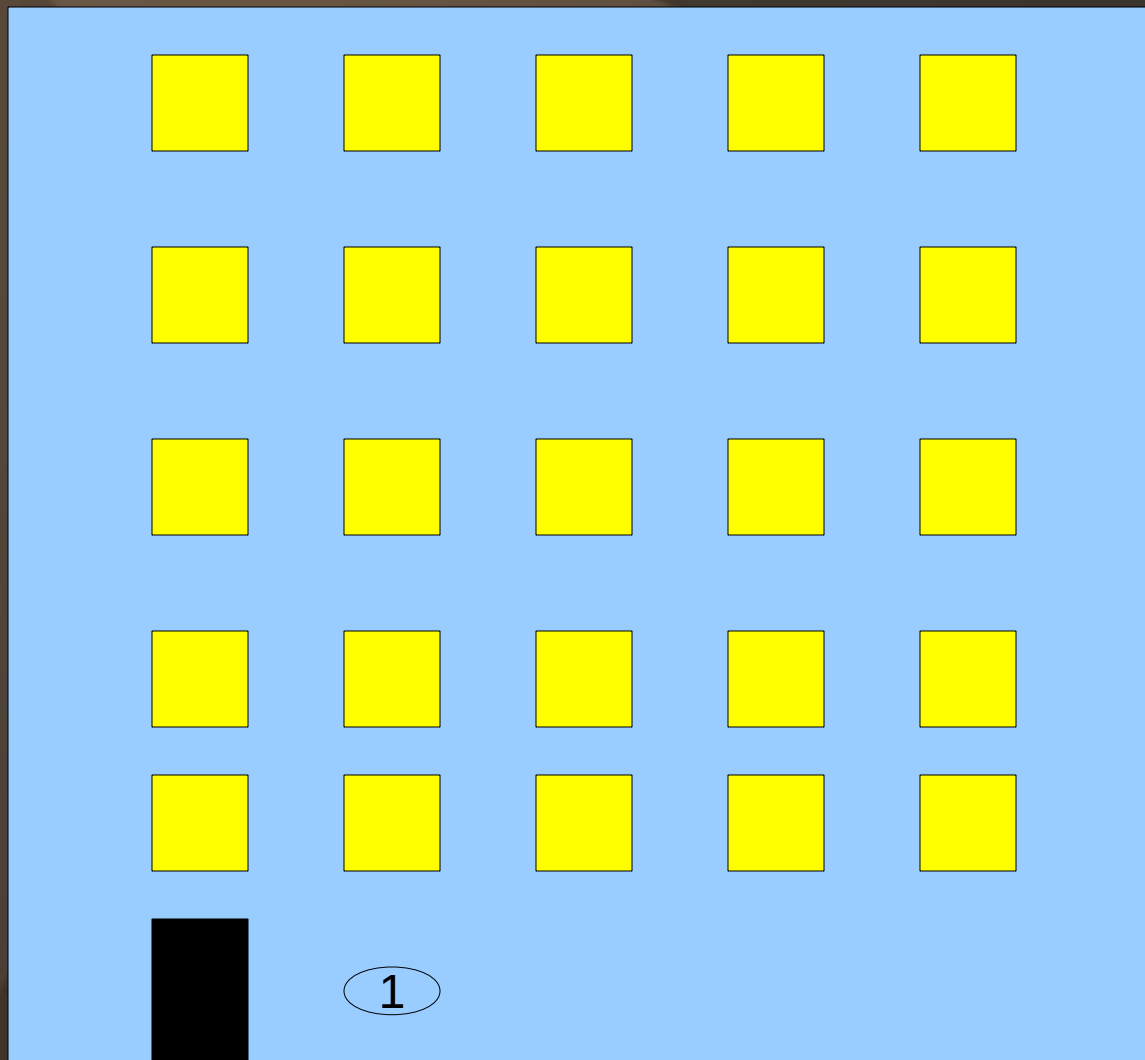
Module Internals

stop_elevator(0)



Module Internals

stop_elevator(0)



Implementing Time Constraints (User-Space)

```
#include <unistd.h>
unsigned int sleep(unsigned int
seconds);
```

- A call to `sleep` will have the program cease to the task scheduler for **seconds** number of seconds.
- You'll need this for several portions of the project.

Implementing Time Constraints (Kernel-Space)

```
#include <linux/delay.h>  
void ssleep(unsigned int seconds);
```

- A call to `ssleep` will have the program cease to the task scheduler for **seconds** number of seconds.
- Furthermore, several task states are changed in order to ensure proper behavior in an SMP system.

Additional Design Considerations

- How to move elevator?
- Which elevator to move, when more than 1 are active?
- What scheduling algorithm to use?
- How to handle incoming requests concurrently while scheduling elevators?
- What data needs to be synchronized?

procfs Programming

- procfs Hello World

procfs Hello World

- Much like the kernel module hello world example, we'll cover all steps from start to finish to create our own **read-only** procfs entry.
- Steps needed:
 - Create entry in module_init function
 - Register reading function with procfs_read
 - Delete entry in module_cleanup function
- Reference:
 - [Linux Kernel Module Programming Guide: Proc FS](#)

/proc/helloworld: Headers and Global Data

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
MODULE_LICENSE("GPL");

#define ENTRY_NAME "helloworld"
#define PERMS 0644
#define PARENT NULL

struct proc_dir_entry *proc_entry;
...
```

/proc/helloworld: Creation

```
int procfile_read(char *buf,
char **buf_location, off_t offset, int buffer_length, int *eof, void *data);

int hello_proc_init()
{
    proc_entry =
        create_proc_entry(ENTRY_NAME,
                          PERMS, PARENT);
    /* check proc_entry != NULL */
    proc_entry->read_proc = procfile_read;
    proc_entry->owner = THIS_MODULE;
    proc_entry->mode = S_IFREG | S_IRUGO;
    proc_entry->uid = 0;
    proc_entry->gid = 0;
    proc_entry->size = 11;

    printk("/proc/%s created\n", ENTRY_NAME);
    return 0;
}
```

/proc/helloworld: Reading

```
int procfile_read(char *buf,  
char **buf_location, off_t offset, int buffer_length,  
int *eof, void *data)  
{  
    int ret;  
    printk("/proc/%s read called.\n", ENTRY_NAME);  
  
    /* Setting eof. We exhaust all data in one shot */  
    *eof = 1;  
    ret = sprintf(buffer, "Hello World!\n");  
  
    return ret;  
}
```

/proc/helloworld: Deletion

```
void hello_proc_exit()  
{  
    remove_proc_entry(ENTRY_NAME,  
                      NULL);  
    printk("Removing /proc/%s.\n",  
          ENTRY_NAME);  
}
```

/proc/helloworld: Registration

```
module_init(hello_proc_init);  
module_exit(hello_proc_exit);
```

Next Time:

- More procfs Programming
- Scheduling Algorithms
- Kernel Debugging
- More Project 3 Hints

Any Questions?