

Recitation Week #11

Finishing Project 3

Alejandro Cabrera
Operating Systems
COP4610 / CGS5765

Today's Recitation

- Project 3 Clarifications
- Project 3 Walk-through: Modeling Techniques
- Kernel Linked Lists
- Project 3 Hints
- Project 3 Demo Registration

Clarifications

- /proc/elevator
 - Module only reads state
 - Stores NO state, only reads state from kernel
 - procfs entry implementation goes here
 - Elevator system implementation does not go here
- LINUX_DIR/elevator/
 - Stores ALL state
 - Elevator system implementation goes here
 - As well as system calls
 - Does not include procfs entry implementation
 - Provides functions to access state from modules

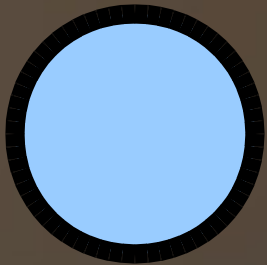
Walk-through: Modeling Technique

- Goal: Design an elevator scheduling method from start to finish.
- The design should take into account what an elevator will do at any given time in the system.
- Tool: state machine diagram
- Particular method: SCAN heuristic

State Machine Diagrams

- A diagram used to model the behavior of a dynamic system
- A node represents a given state
- An arrow between a node and another node represents a transition
- A transition means a condition has been met so we move to the next state

State Machine Diagrams: Conventions



A state node



A state node (current state)

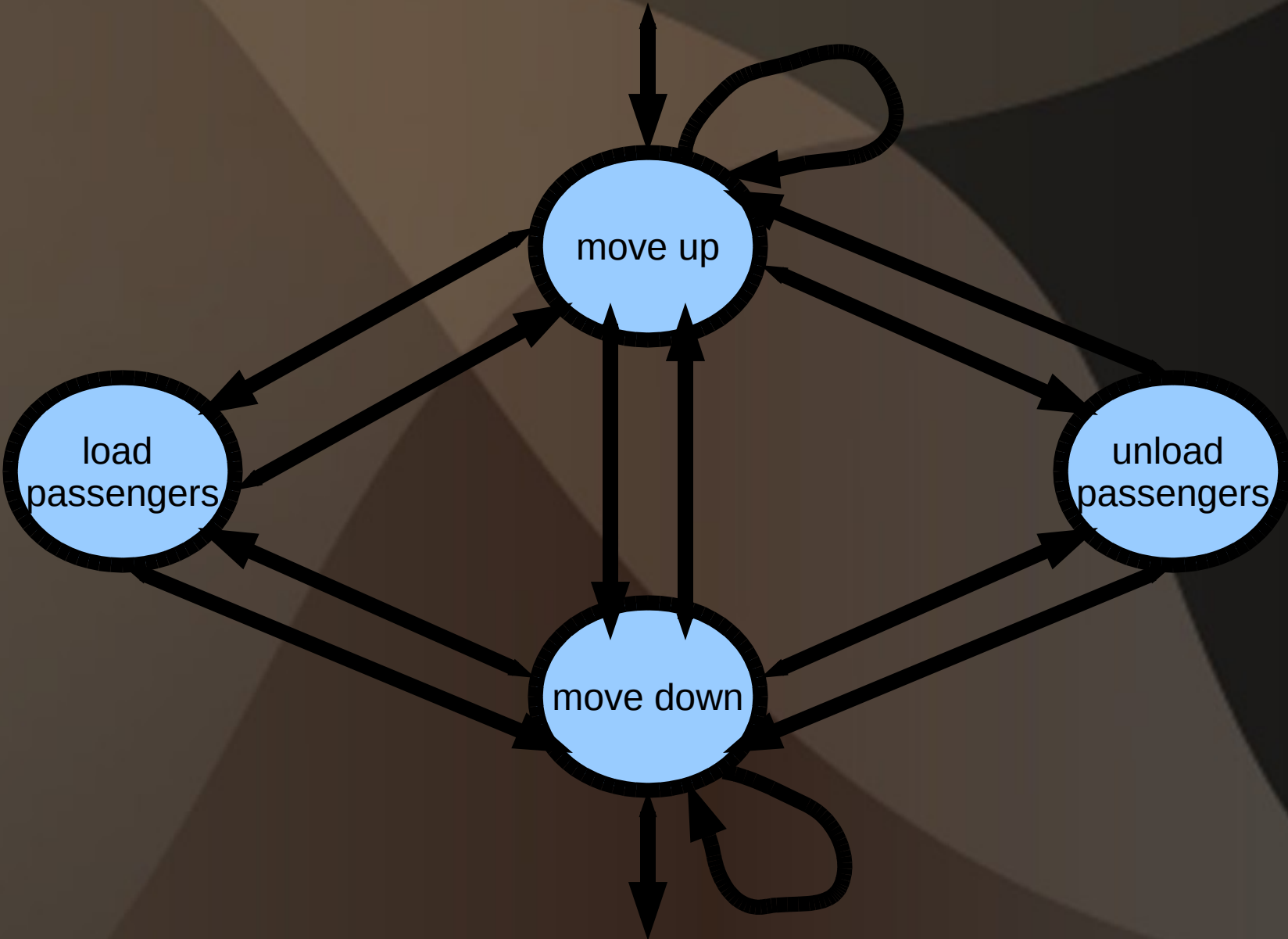


A transition

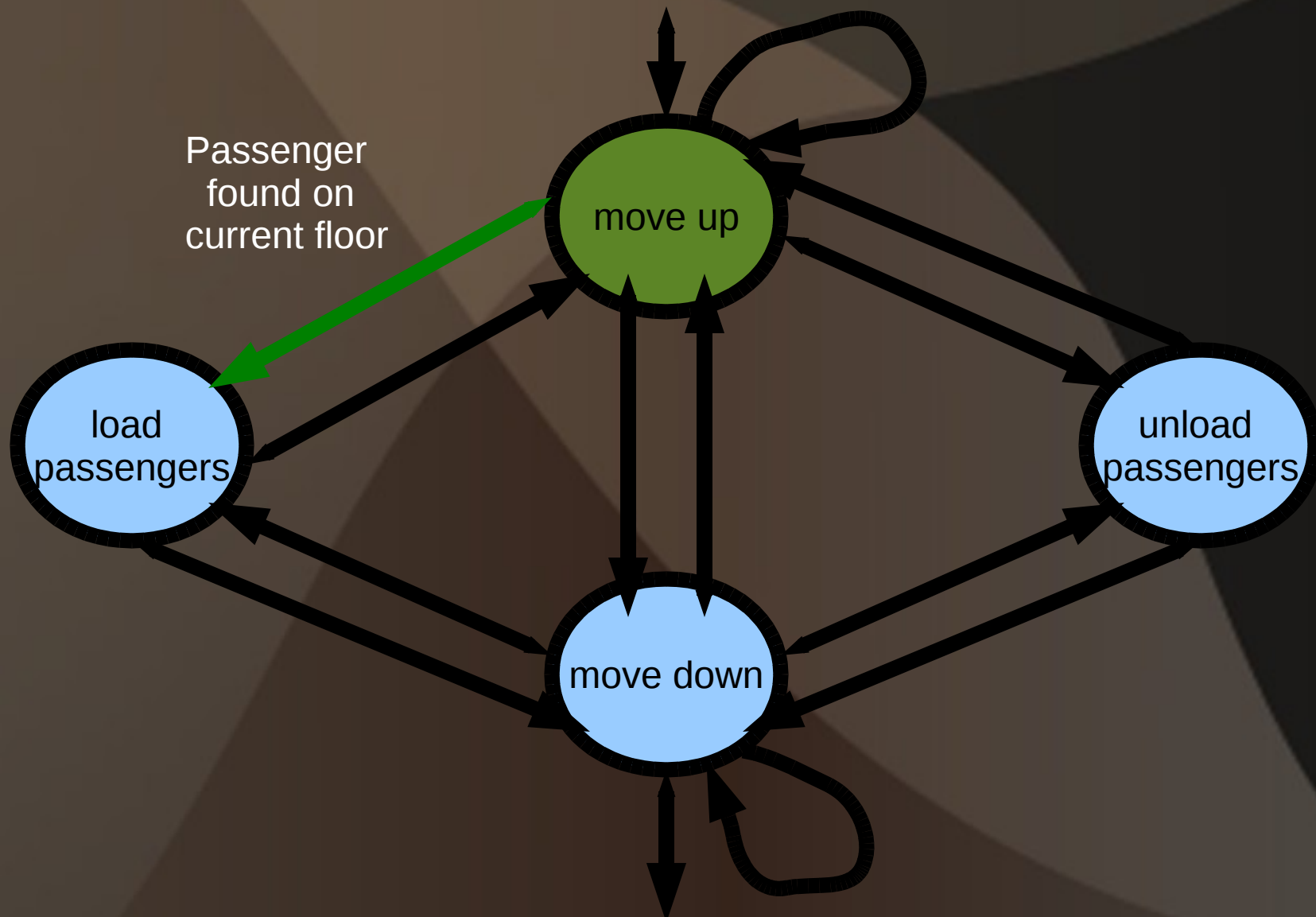


A transition (current transition)

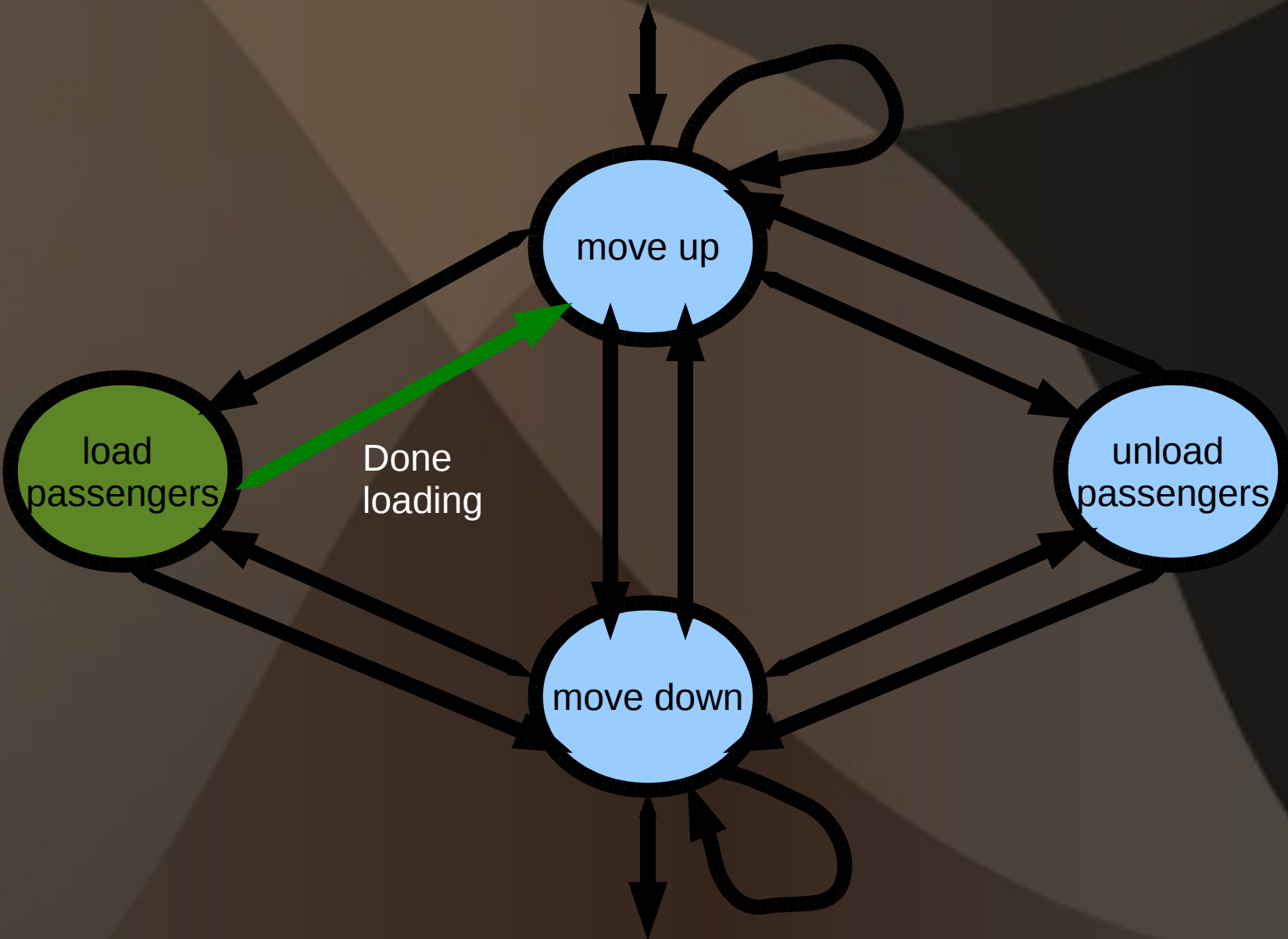
State Machine Diagram: SCAN



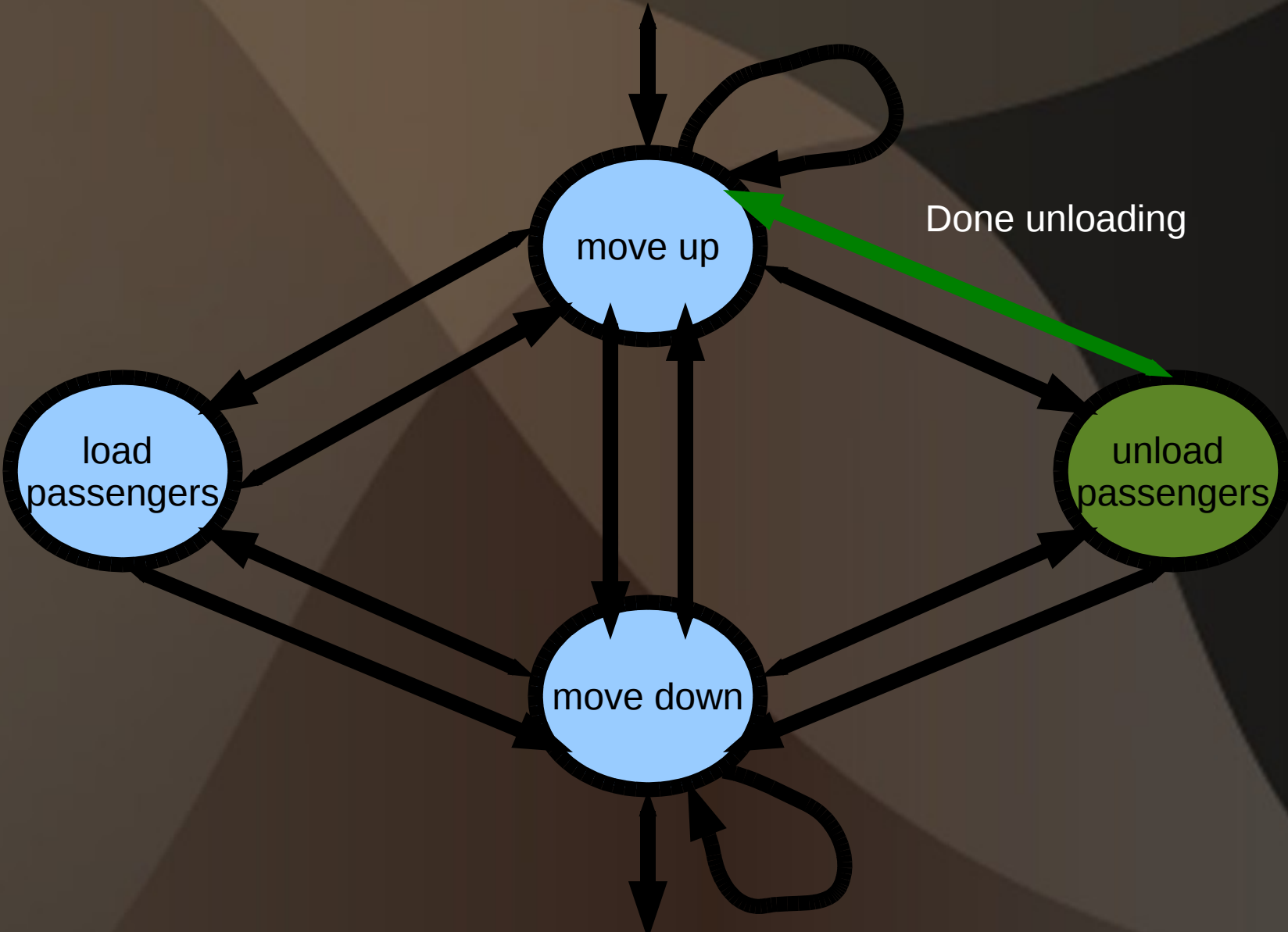
State Machine Diagram: SCAN



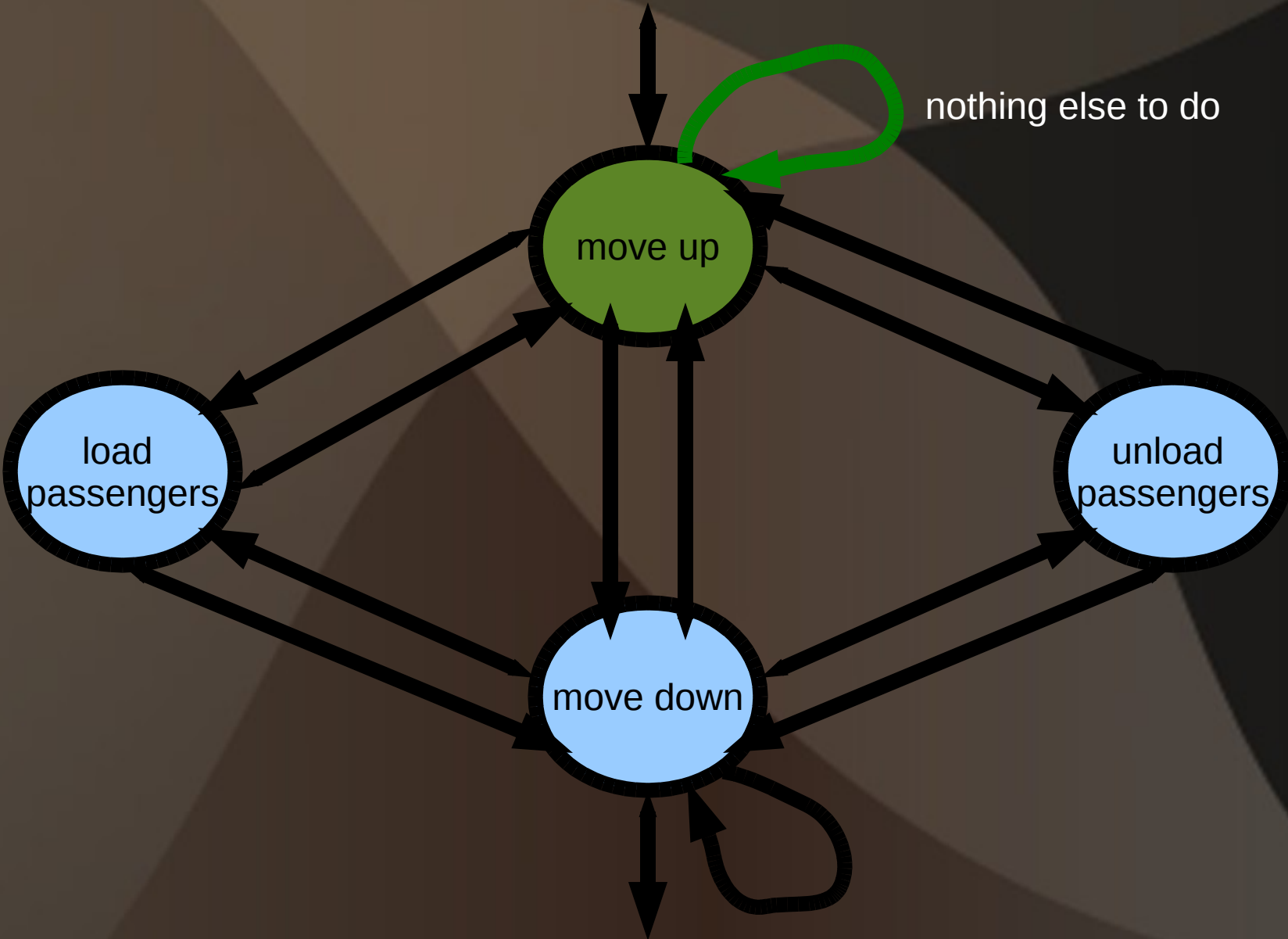
State Machine Diagram: SCAN



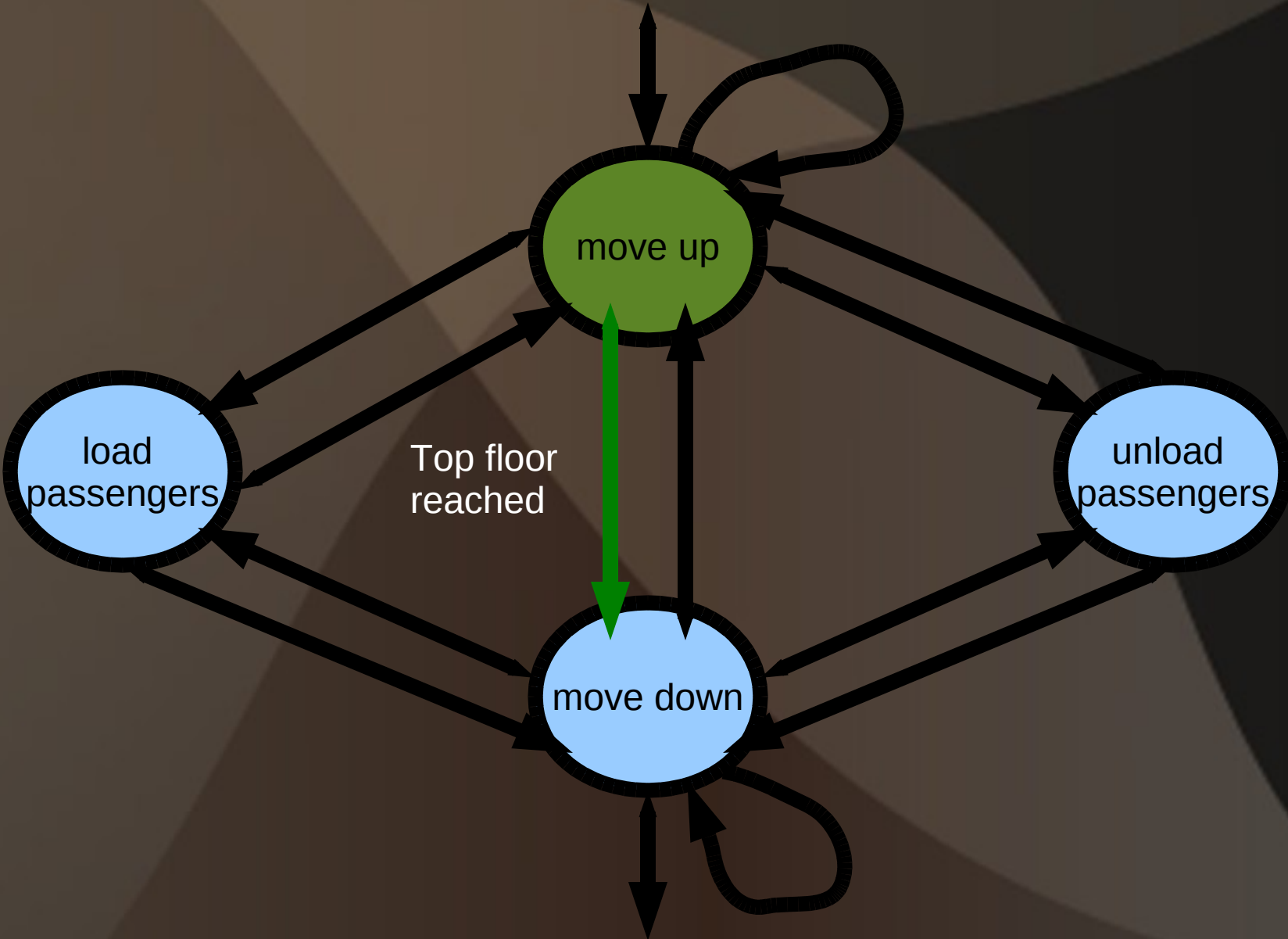
State Machine Diagram: SCAN



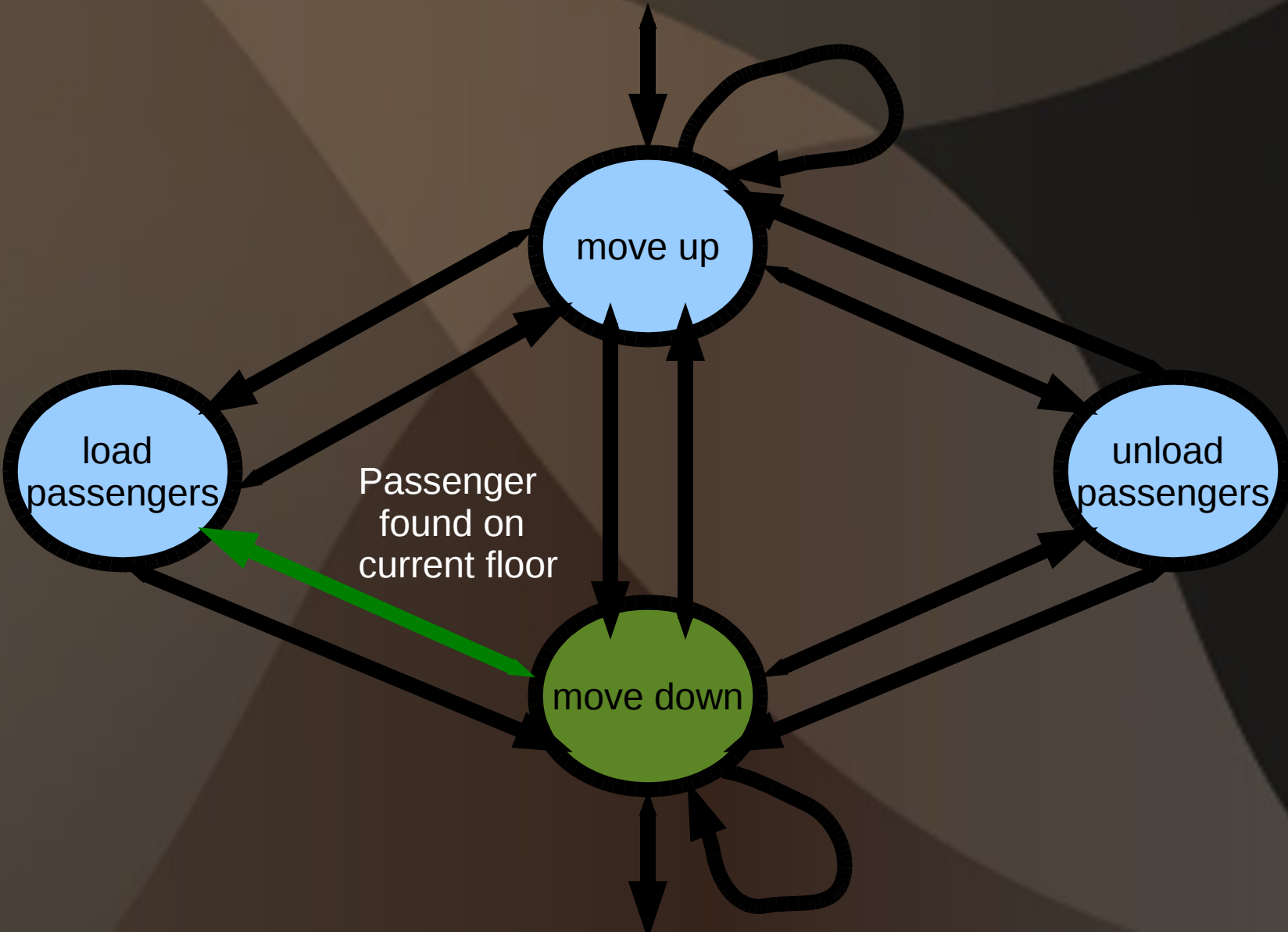
State Machine Diagram: SCAN



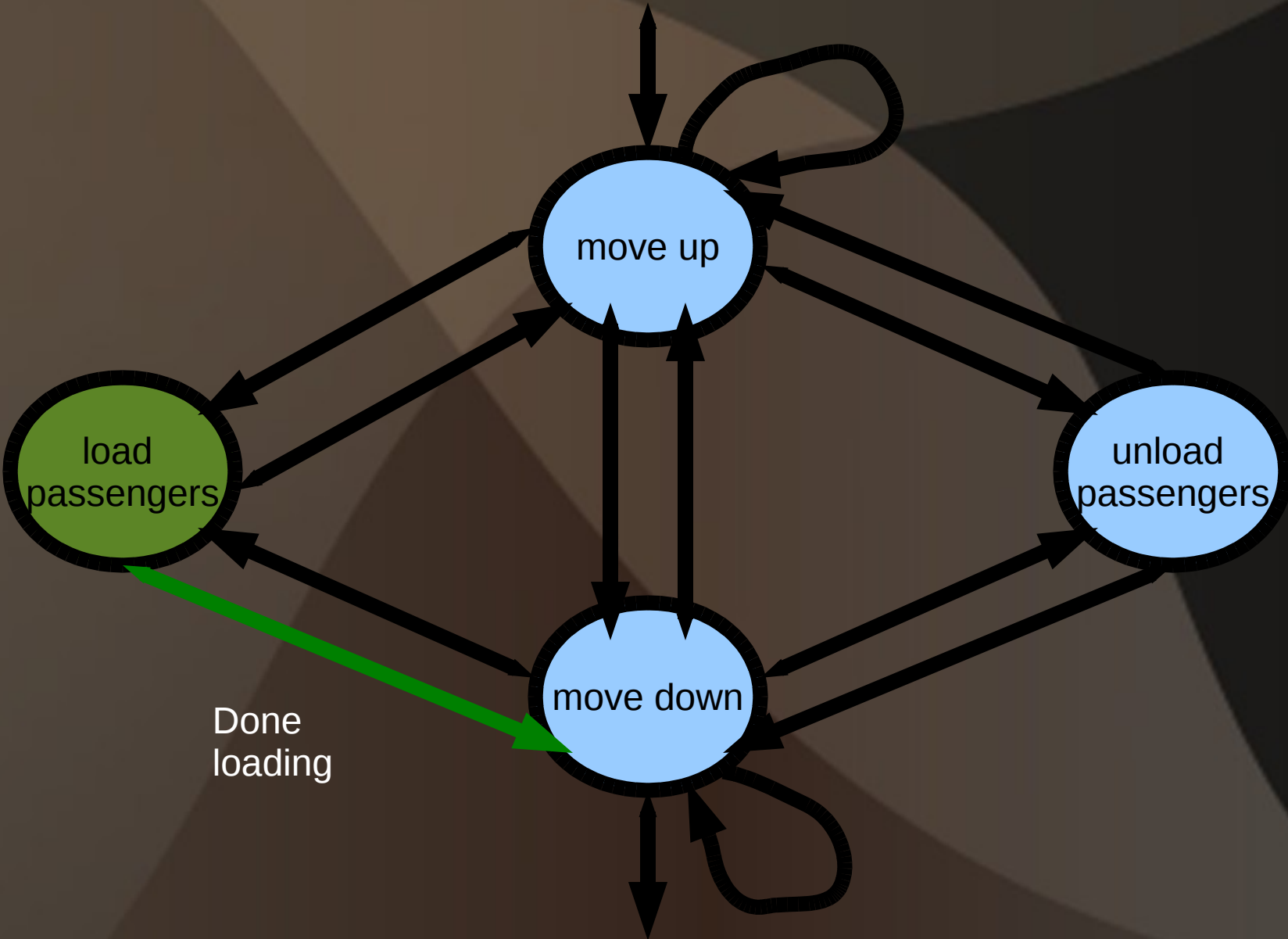
State Machine Diagram: SCAN



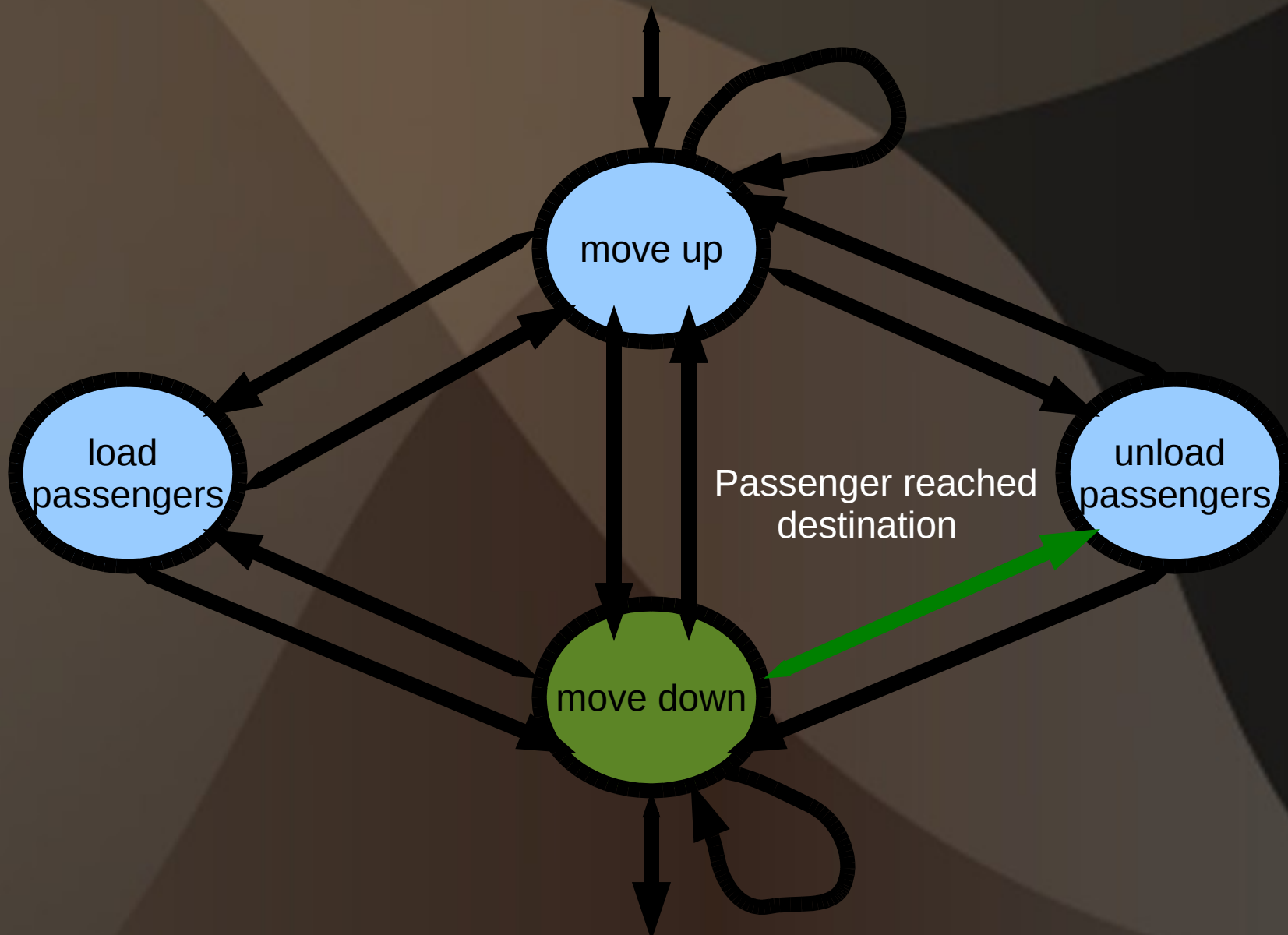
State Machine Diagram: SCAN



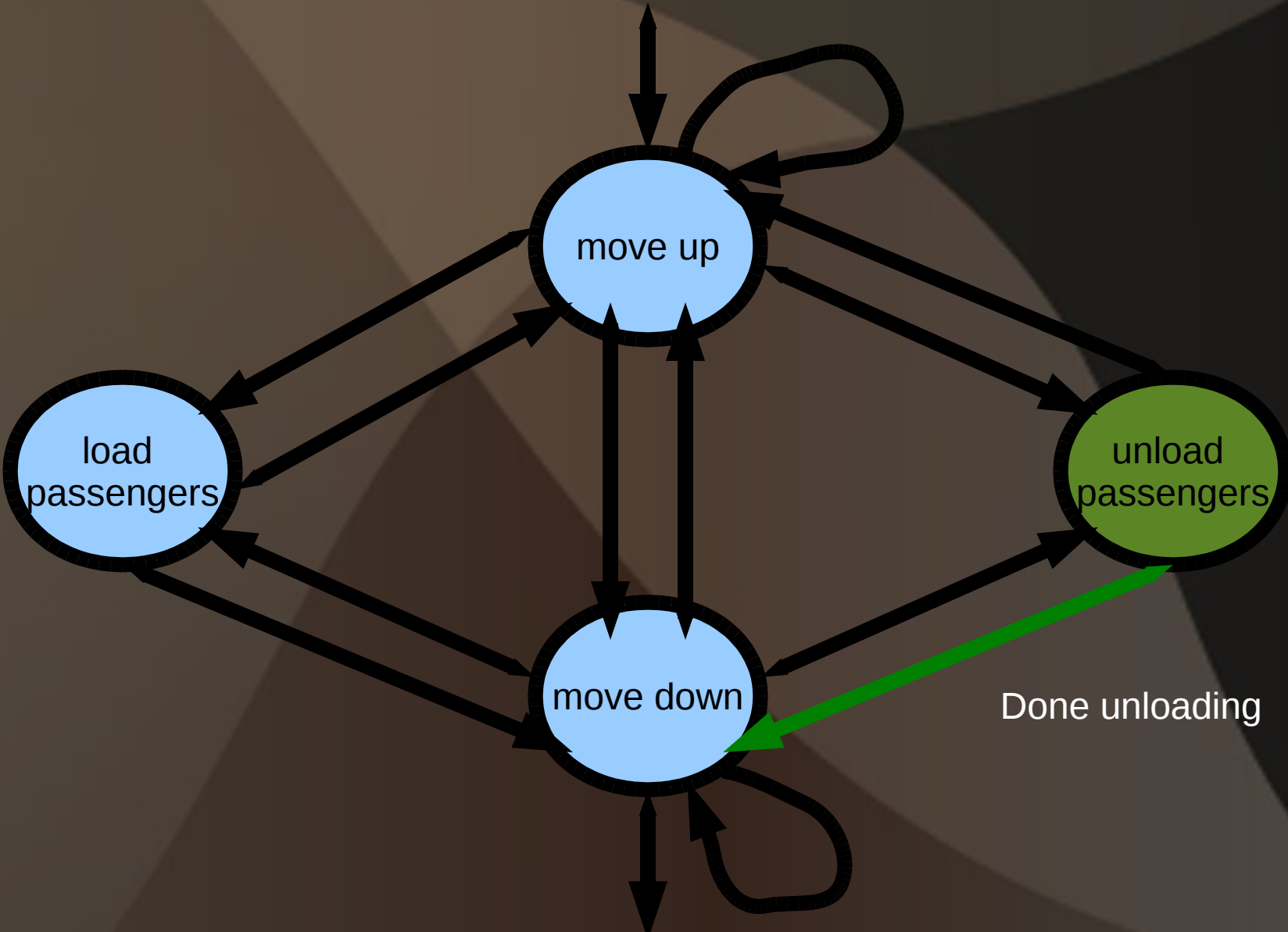
State Machine Diagram: SCAN



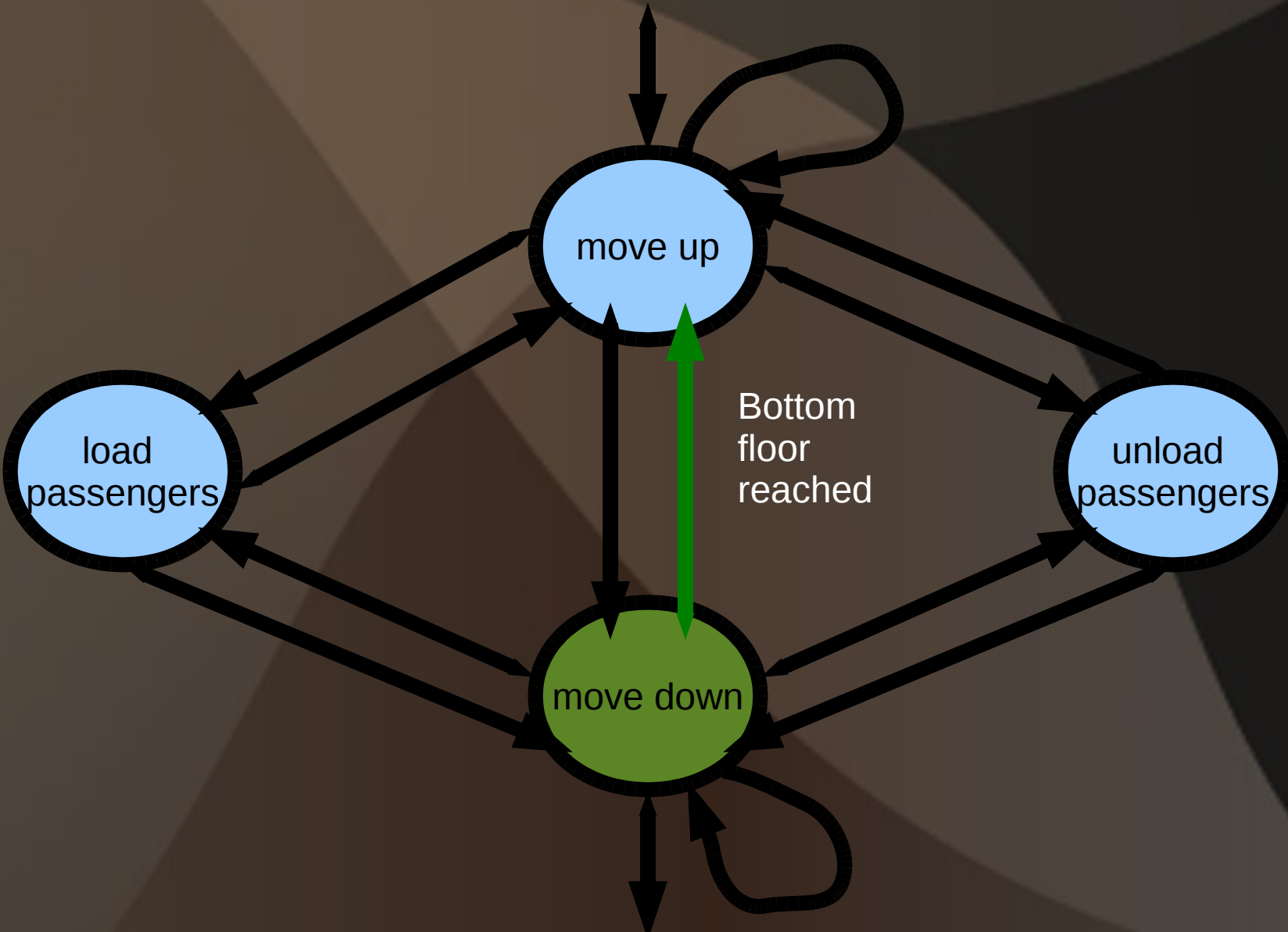
State Machine Diagram: SCAN



State Machine Diagram: SCAN



State Machine Diagram: SCAN



SCAN State Machine Diagram: What's Missing?

- States representing start, stop, and shutdown.
- Transitions from missing states and existing states to each other.
 - The tricky parts
- We continue with additional conventions!

State Machine Diagrams: Conventions 2



State Package

A state machine section, simplified



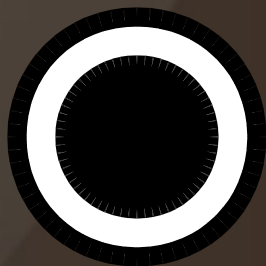
Current State Package

A state machine section, simplified (current)



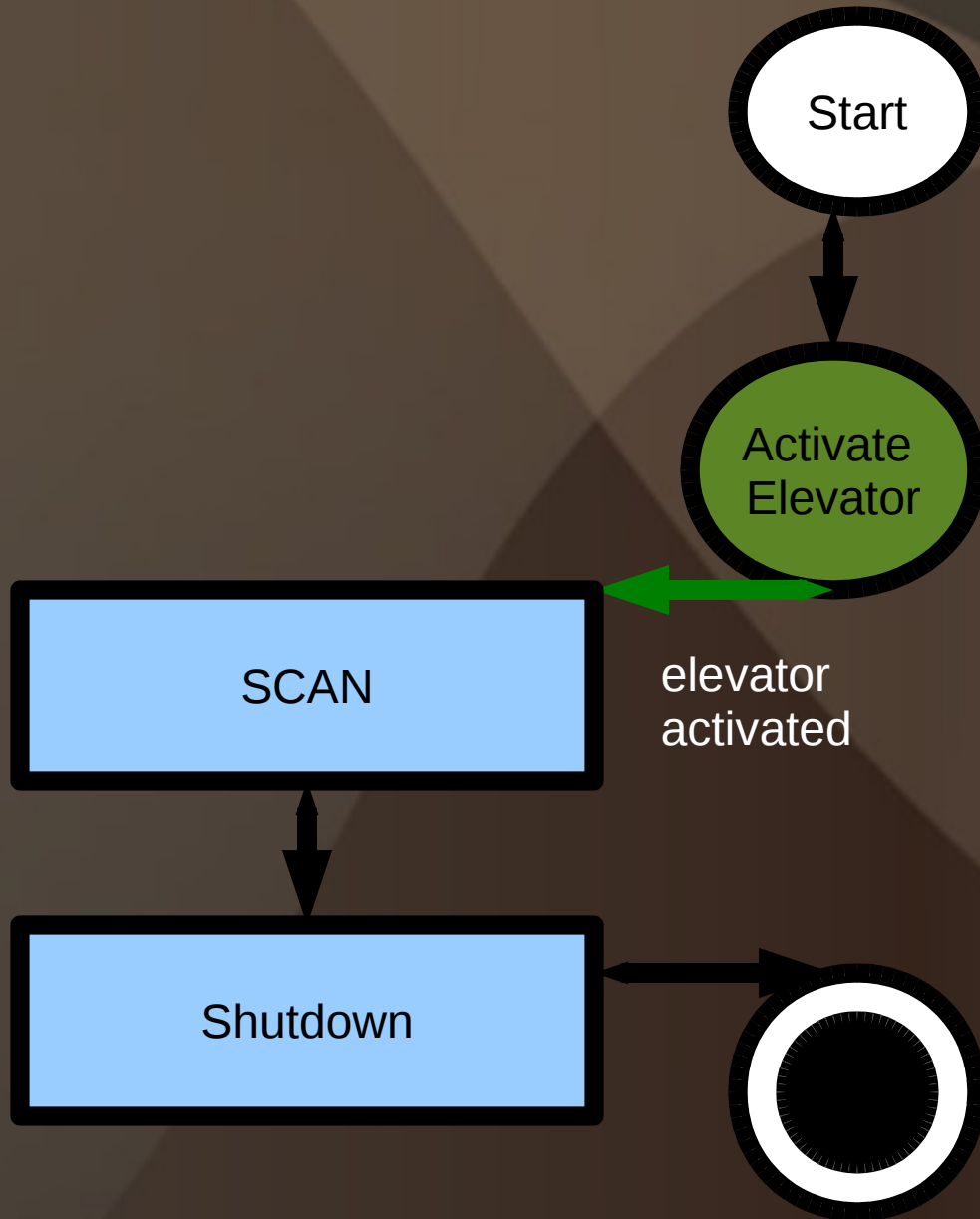
Start

Start node (begin system)

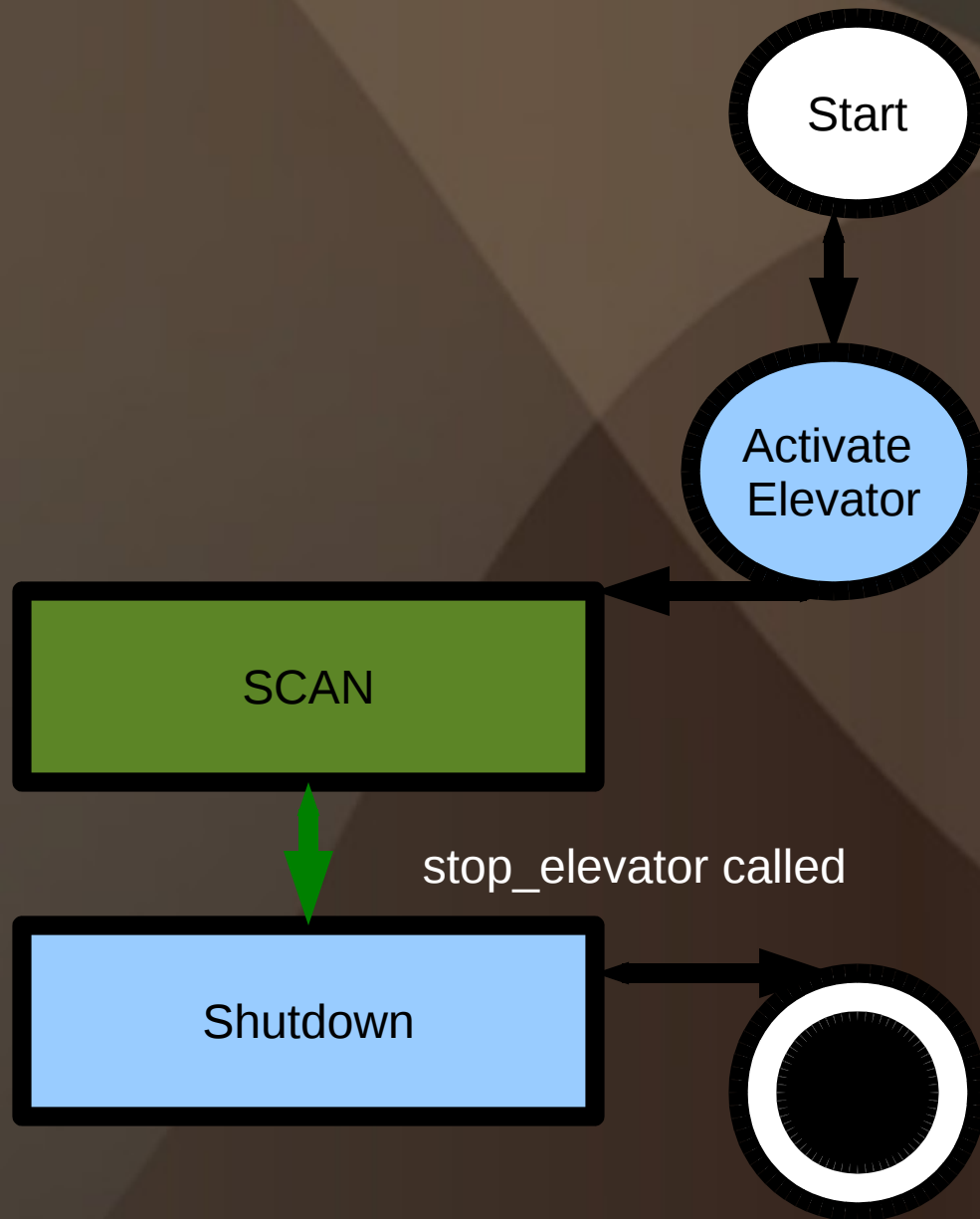


End node (exit system)

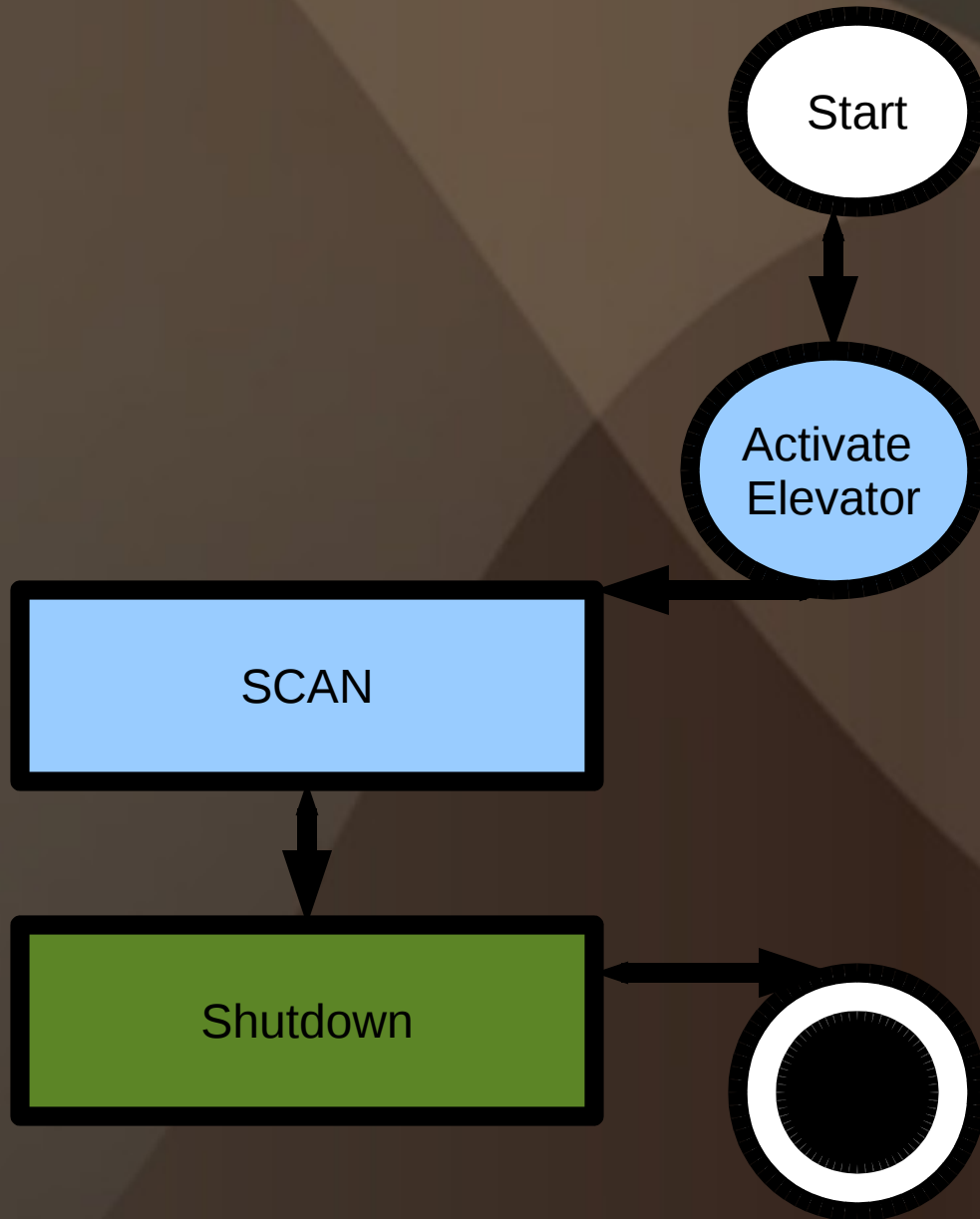
State Machine Diagram: SCAN + More



State Machine Diagram: SCAN + More

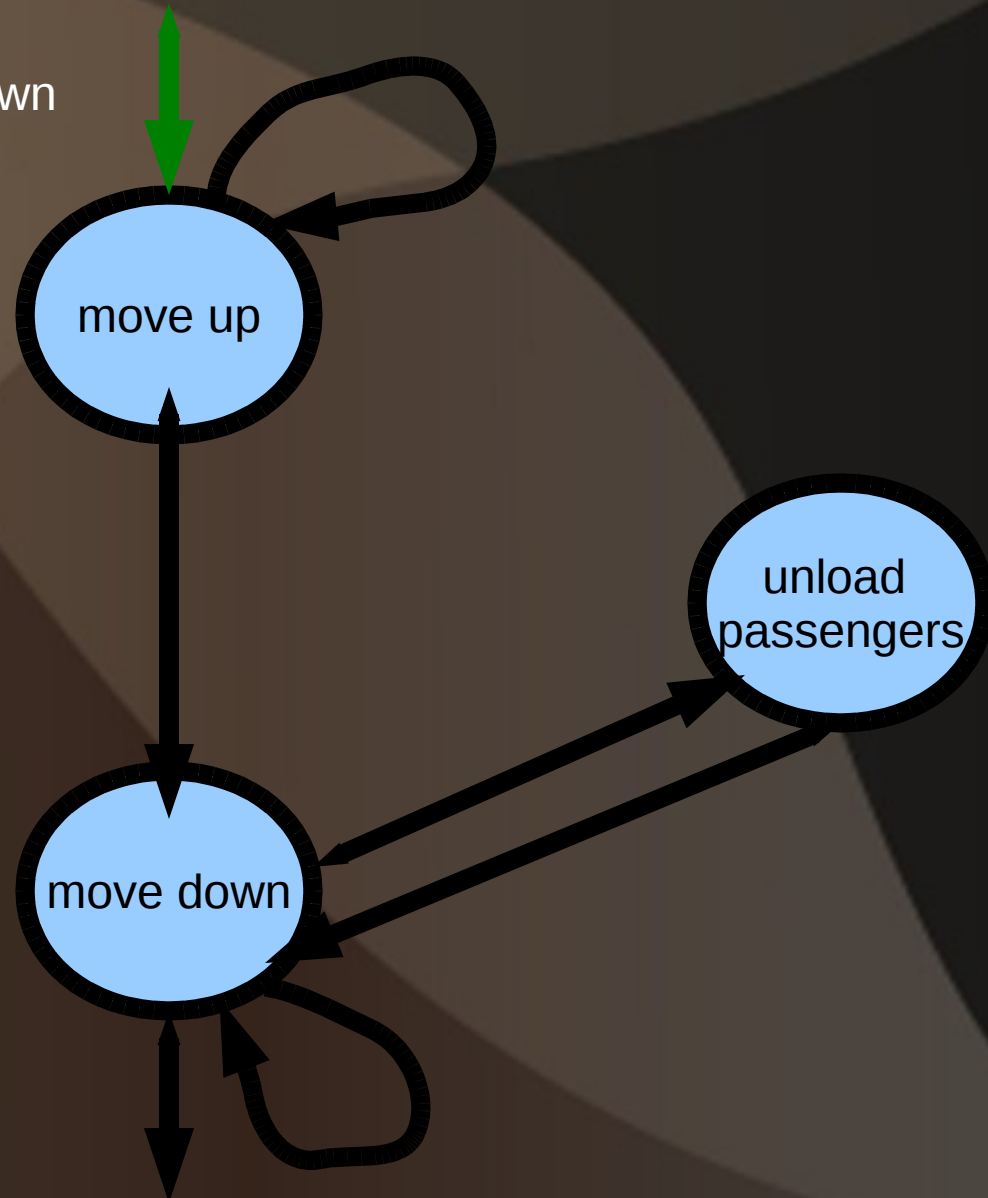


State Machine Diagram: SCAN + More

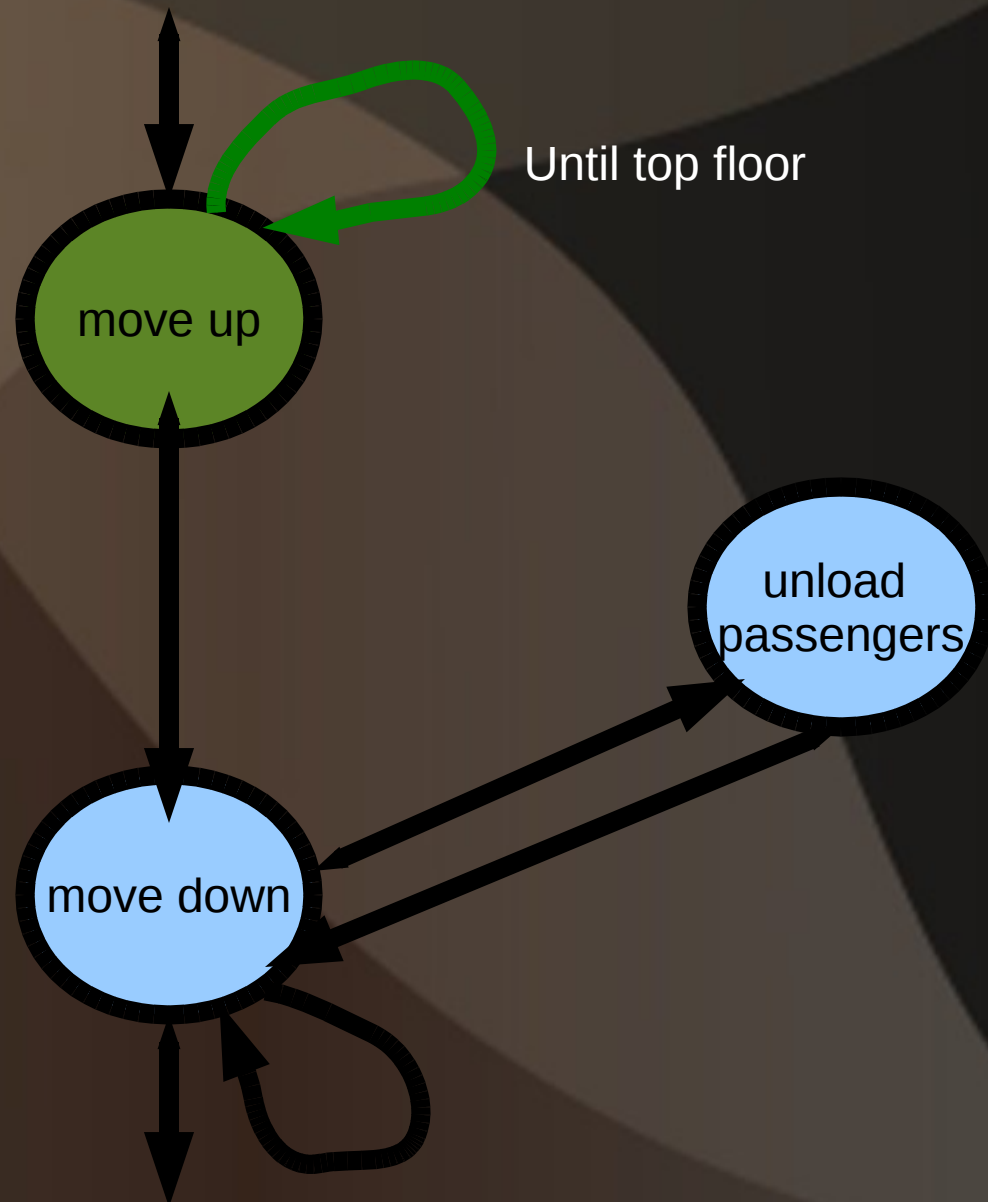


State Machine Diagram: Shutdown

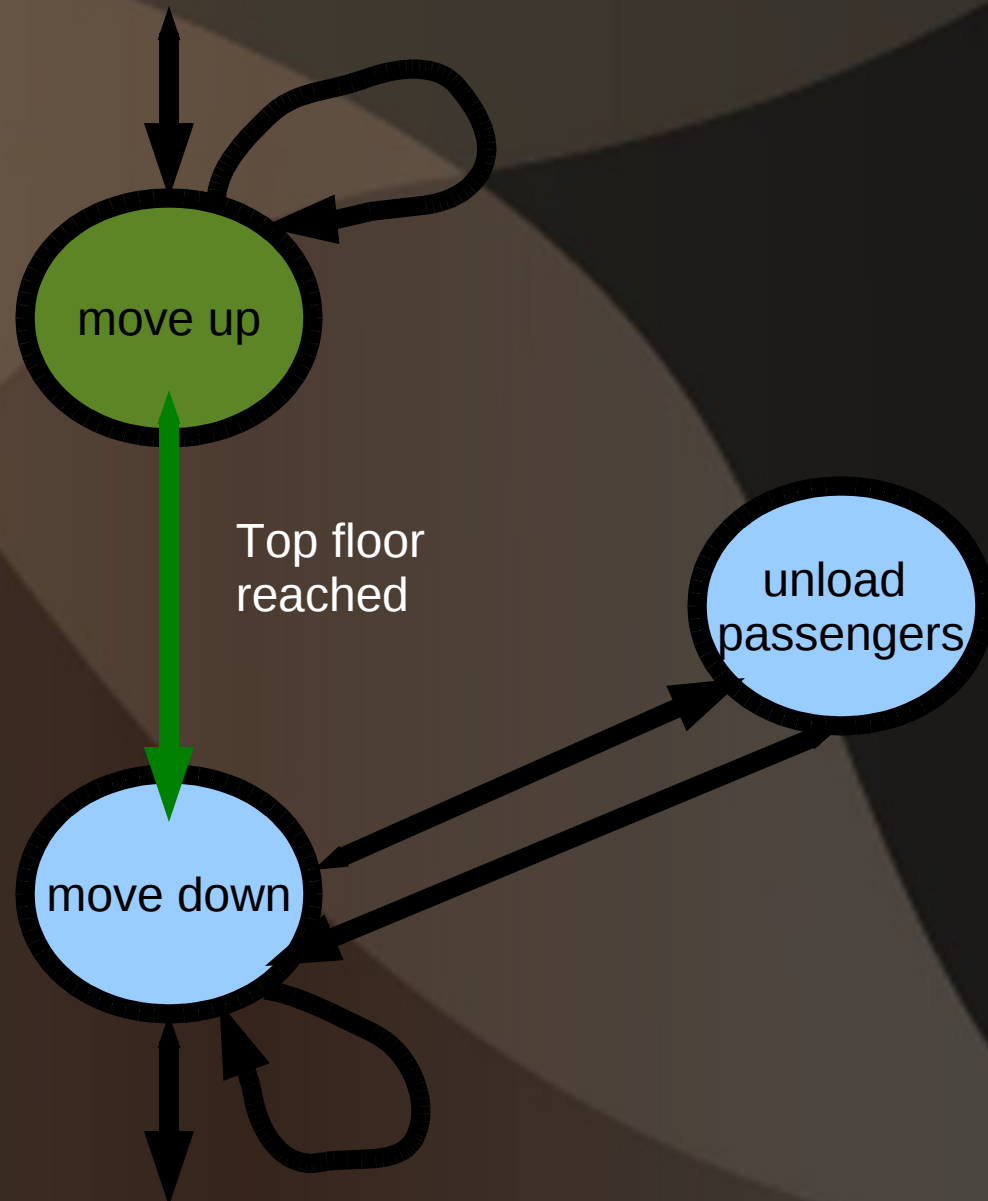
begin
shutdown



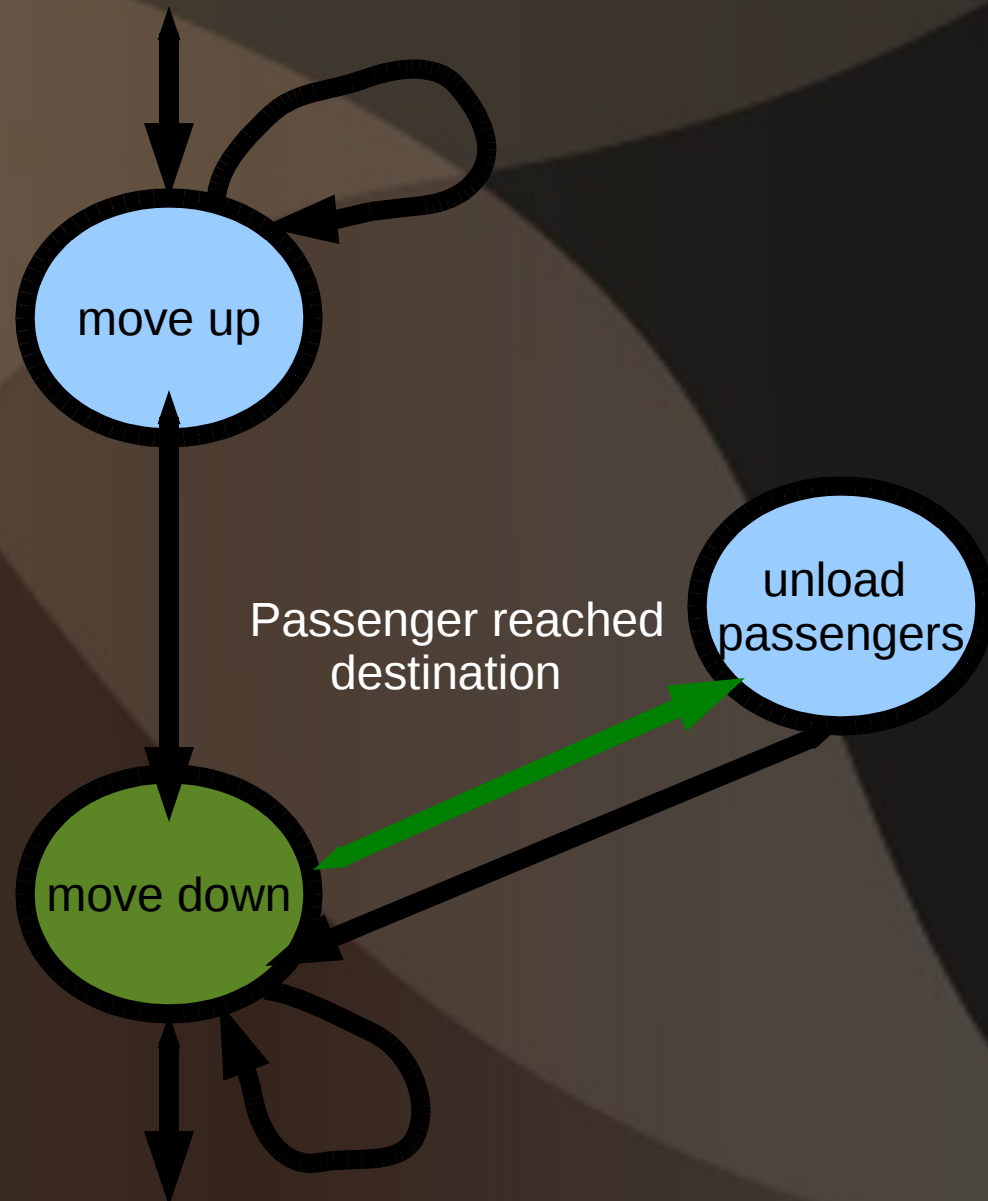
State Machine Diagram: Shutdown



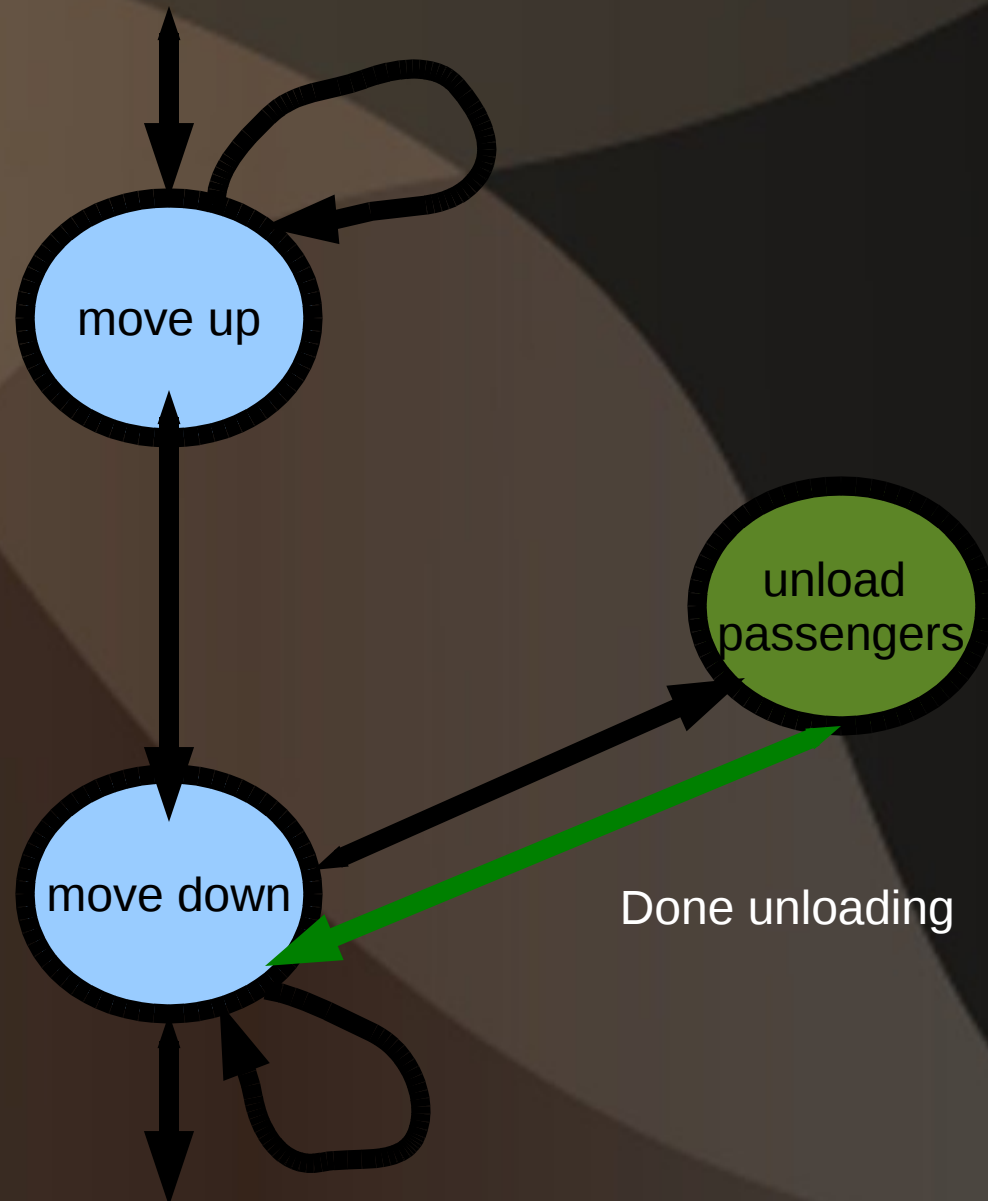
State Machine Diagram: Shutdown



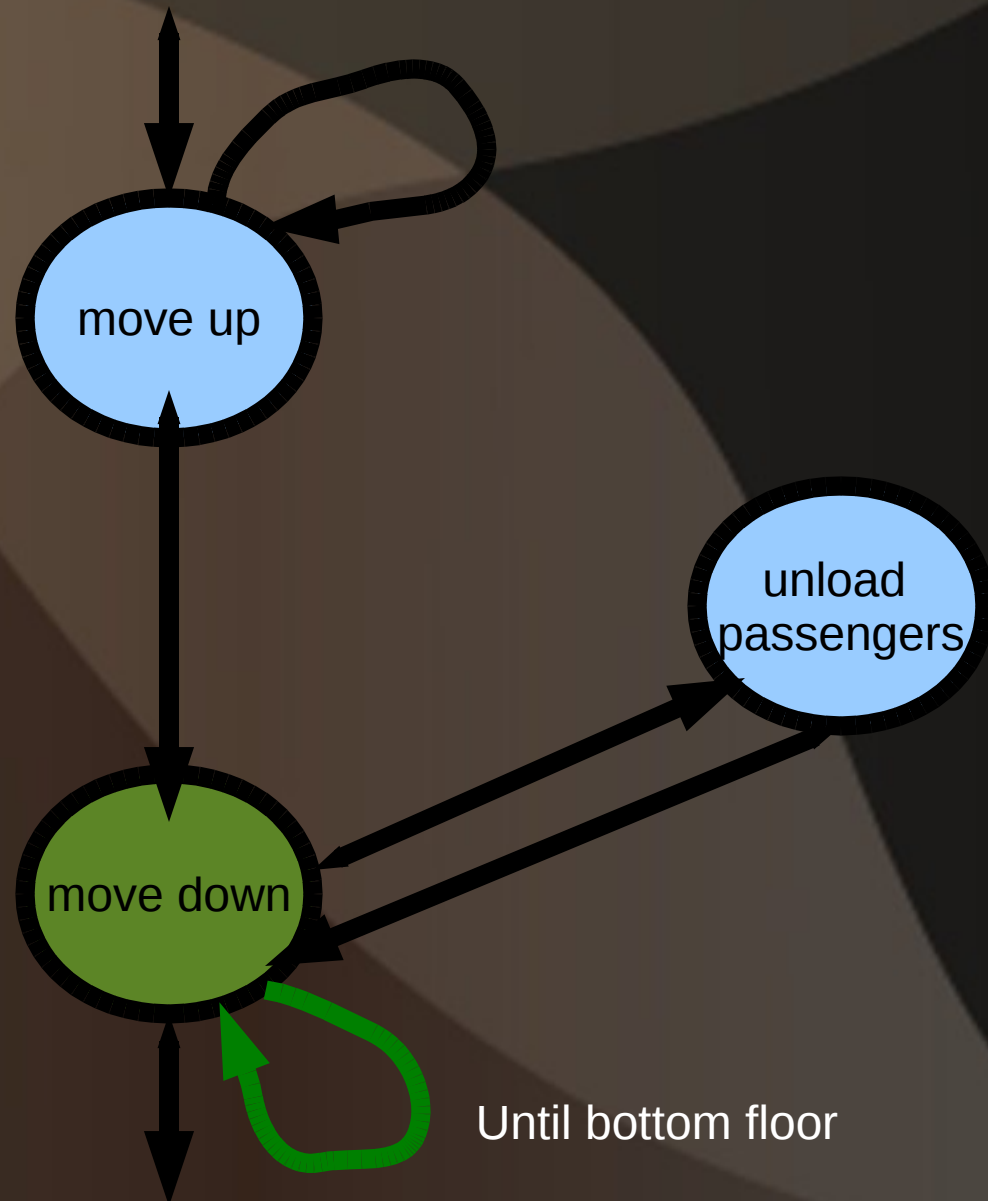
State Machine Diagram: Shutdown



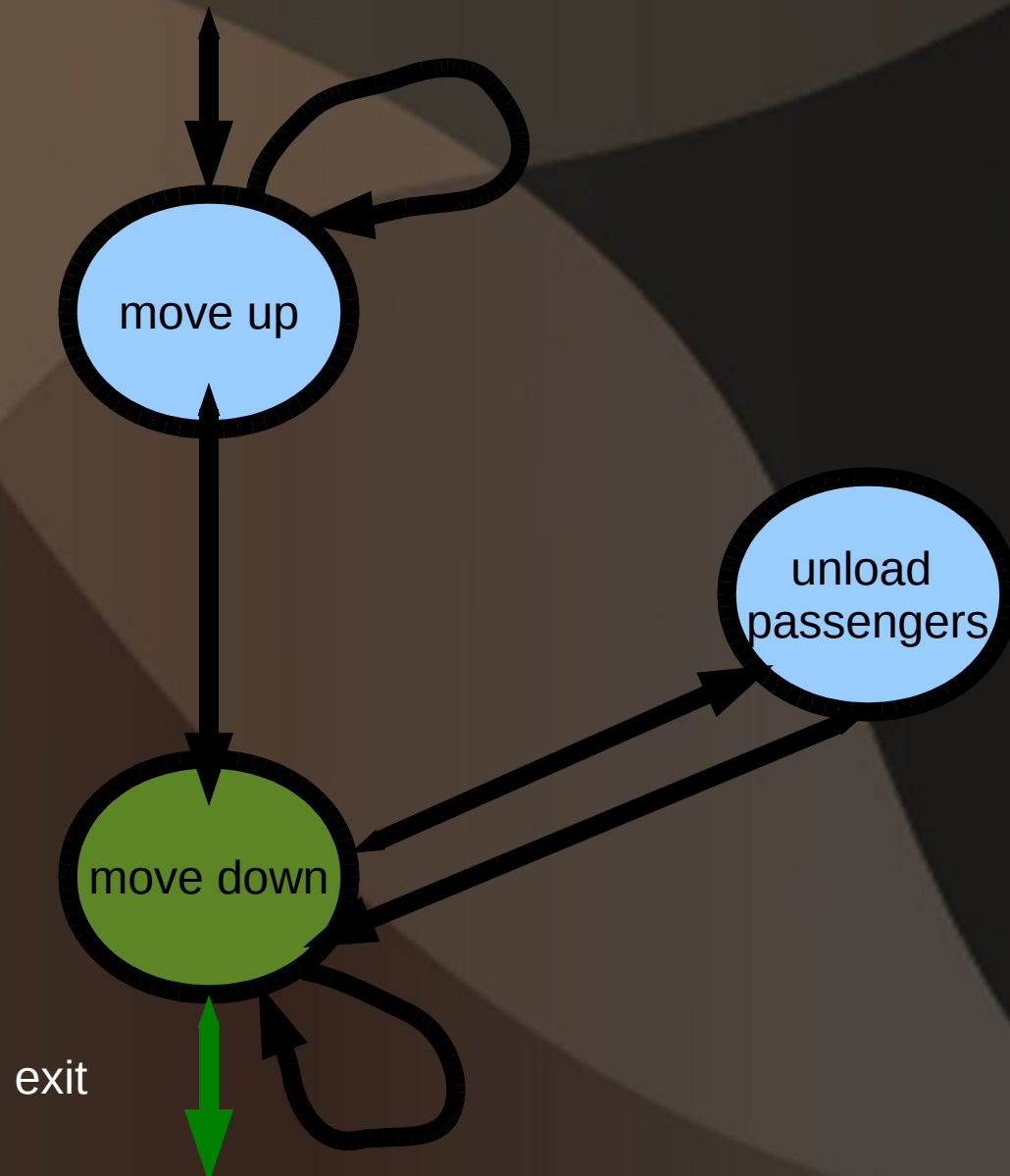
State Machine Diagram: Shutdown



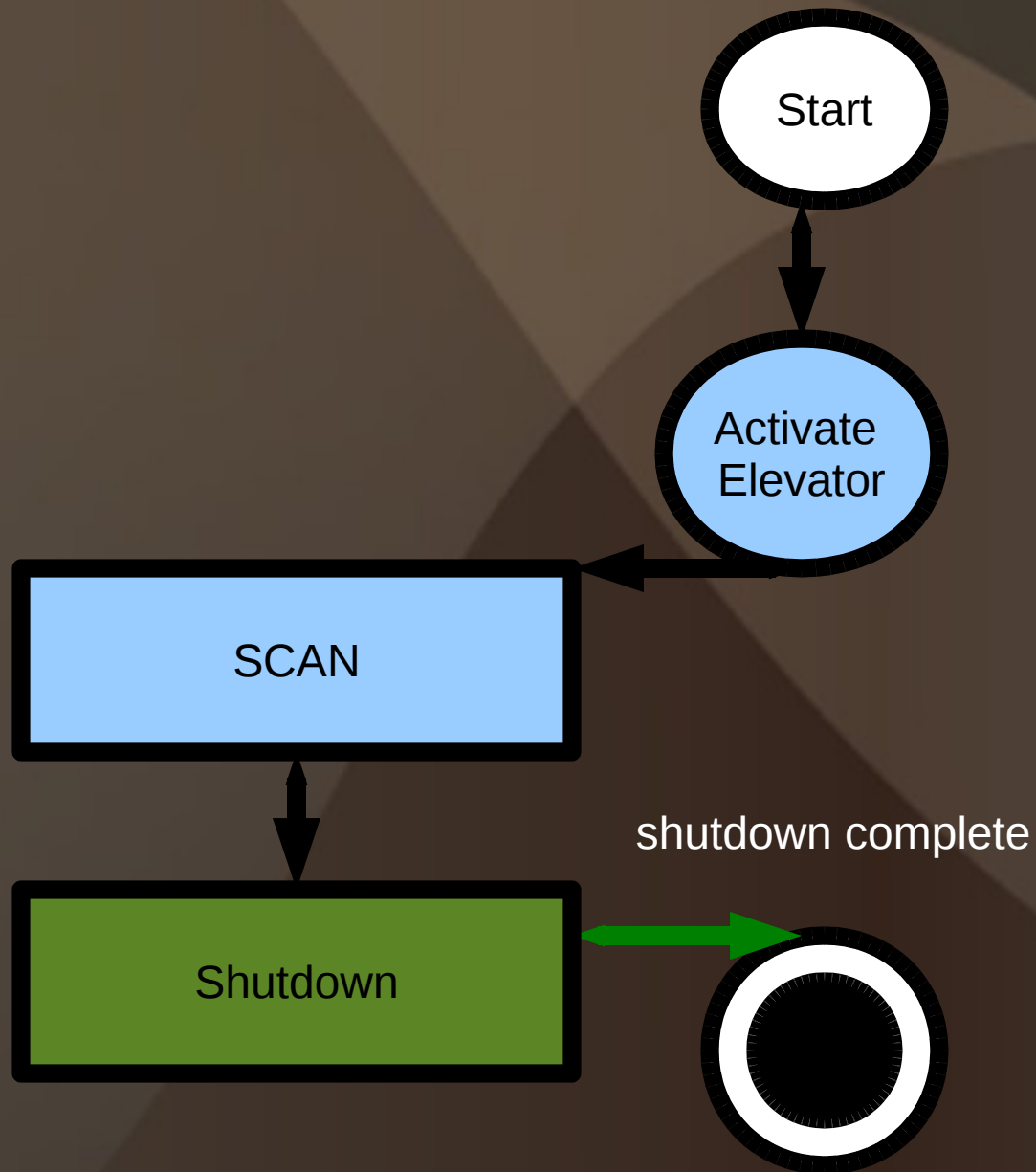
State Machine Diagram: Shutdown



State Machine Diagram: Shutdown



State Machine Diagram: SCAN + More



Kernel Linked Lists

- Linux kernel provides generic doubly- and singly-linked list implementations for use in kernel code
 - Take advantage of it!
- Likely to prove useful for implementing queue processing system in project 3

Why Use Kernel Linked Lists?

- Generic: can be used with any data type
- Portable across platforms
- Easy to Use
- Readable: the code written using the kernel linked list functions has the potential to be easily understood
- Saves Time: a.k.a. “don't reinvent the wheel”

struct list_head

```
struct list_head{  
    struct list_head *next, *prev;  
};
```

- A kernel linked list consists of only a next and prev pointer to a list_head structure.
- Implements a circular list structure

Embedding and Using struct list_head

```
struct tasks1{
    struct list_head todo_list;
    int task_id;
    void *task_data;
};
```

Using the above structure, we have access to a list of tasks!

- Each task node contains a task id and task data.

Embedding Multiple Lists

```
struct tasks2{
    struct list_head date_sorted_list;
    struct list_head priority_sorted_list;
    int task_id;
    void *task_data;
};
```

- Now, we can manage tasks in two lists:
 - One list, we append new tasks and keep them sorted by date added
 - For the other list, we insert new tasks to keep them sorted by priority

Initializing Lists

```
struct tasks1 my_tasks;  
INIT_LIST_HEAD(&my_tasks.todo_list);  
/* more code */
```

- `INIT_LIST_HEAD` properly initializes a `list_head` structure for use

Adding Elements

```
struct tasks1 *tmp =
    (struct tasks1 *) calloc(1, sizeof(struct tasks1));
list_add(&(tmp->todo_list), &(my_tasks.todo_list));
/* or, to add to tail of list */
list_add_tail(&(tmp->todo_list), &(my_tasks.todo_list));
```

- `list_add()` and `list_add_tail()` allow insertion of elements into kernel-supplied list structure
 - From C++, think `push_back()` and `push_front()`

Kernel Linked Lists: What's Missing?

- List traversal
- Removing elements from a list
- Moving elements
- Moving elements to another list
- Reference:
 - LINUX_DIR/include/linux/list.h
 - <http://isis.poly.edu/kulesh/stuff/src/klist/>
 - <http://lxr.free-electrons.com/source/include/linux/list.h>
 - LDD3 Chp. 11, pp. 295-299

Project 3 Final Hints

- Design your elevator algorithm using a state machine diagram
 - Make sure you understand where synchronization goes
 - Make sure you understand what happens in a state
 - Make sure you understand what happens when you transition
- Include a mutex inside of each structure you plan to implement in your file
- Expect to write between 500-1000 lines of code, maybe more (for elevator system alone)
- Understand the kernel linked list interface and proper use

Project 3 Final Hints

- Remember what your proc module must print:
 - Elevator number
 - Elevator direction
 - Elevator's current floor
 - Elevator's next floor
 - Elevator's load
 - ...
- These serve as powerful hints as to the minimum information an elevator object must store!

Next Time:

- Project 4

Any Questions?