Week 8
Locking, Linked Lists, Scheduling Algorithms

Classes COP4610 / CGS5765
Florida State University
Concurrency Aspects of Project 2

- Synchronizing access to request queue(s)
  - Multiple producers may access request queue(s) at the same time
  - Multiple consumers may access request queue(s) at the same time

- Synchronizing access to other global data
Elevator Concurrency

- Passengers may appear on a floor at the same time as an elevator arrives at a floor.
- The procfs display may be read at the same time that you're updating:
  - Number of passengers that you've serviced
  - Total number of passengers on a given floor
  - Other stuff, depending on your implementation
- How do you guarantee correctness?
Global Data vs. Local Data

- **Global data** is declared at global scope, e.g. outside of any function body
  - Often necessary for kernel programming
- Particularly sensitive to concurrency issues
  - Be **extra** careful when handling globals
**Global Data vs. Local Data**

- *Local data* is declared within a function
- Local data is sensitive to concurrency issues when it depends on global data or when parallel access is possible
  - Think carefully about whether it needs to be synchronized
Synchronization Primitives

- Semaphores
  - User space
  - Kernel space

- Mutexes
  - User space
  - Kernel space

- Spin locks
  - Kernel space

- Atomic Functions
Synchronization Primitives
(We'll Only Cover These)

- Mutexes
  - User space
  - Kernel space

- Does anyone remember the differences between a mutex and semaphore?
The Mutex

Caught up in the mutex?
Mutexes

- Mutex – A construct to provide MUTual EXclusion
- Based on the concept of a semaphore
- Found at `<source_dir>/include/linux/mutex.h`
- Can be locked or unlocked
  - Only one thread of execution may hold the lock at a time
Kernel-Space Mutex - Initialization

- `mutex_init(&mutex)`

- Declare and initialize a mutex
  - Only initialize it once
Kernel-Space Mutex - Locking

- void mutex_lock(struct mutex *);
- int mutex_lock_interruptible(struct mutex *);

- mutex_lock() can wait indefinitely
- mutex_lock_interruptible() locks a mutex as long as it is not interrupted
  - returns 0 if locking succeeded, < 0 if interrupted
- Use of the interruptible version is typically preferred
Kernel-Space Mutex – Unlocking

- `void mutex_unlock(struct mutex *);`

- Guarantees that the mutex is unlocked
  - Why is there no interruptible version of this function?
/* Declare your mutex */
struct mutex my_mutex;

/* Initialize your mutex */
mutex_init(&my_mutex);

/* Lock */
if(mutex_lock_interruptible(&my_mutex))
    return -ERESTARTSYS;

    /* Do stuff to protected global variables */

/* Unlock */
mutex_unlock(&my_mutex);
User-Space Mutex

- Used with pthreads
- Might be useful if you are prototyping your elevator in user-space before porting to kernel
User-Space Mutex - Initialization

- `int pthread_mutex_init(pthread_mutex_t *, NULL);`
- `int pthread_mutex_destroy(pthread_mutex_t *);`

- `pthread_mutex_init()` dynamically allocates a mutex
- `pthread_mutex_destroy()` frees a mutex
User-Space Mutex - Locking

- `int pthread_mutex_lock(pthread_mutex_t *);`

- Returns 0 on locking, < 0 otherwise
User-Space Mutex - Unlocking

- `int pthread_mutex_unlock(pthread_mutex_t *);`

- Returns 0 on unlocking, < 0 otherwise
Kernel Linked Lists

More elegant than arrays, but takes some getting used to...
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 2
- Found in `<linux source>/include/linux/list.h`
Kernel Linked Lists

- **Pros**
  - Safer and saves time over implementing them yourself
  - Handy functions already defined for your use
  - Can dynamically represent a FIFO

- **Cons**
  - Pointers can be a bit tricky
A kernel linked list consists of only a `next` and `prev` pointer to a `list_head` structure.

Start with an empty `list_head`

Next connect to other `list_head` structures embedded into your structures
struct list_head

*Picture taken from Linux Device Drivers, 3 ed.*
Declarer and initializes an empty linked list called todo_list

Note the INIT_LIST_HEAD function takes a pointer
Embedding a list_head into your own structs

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

- We can define a struct that holds more valuable information with an empty list_head struct inside
- How do we add this struct to the main list?
/* Declare a struct * with an embedded list_head */
struct tasks1 *my_task;

/* Kmalloc and fill in my_task with data you care about here... */

/* Add the my_task struct to the last element in the list using list_add_tail() */
list_add_tail(&my_task->list, &todo_list);

- List_add_tail() can be used to implement a FIFO
Other Linked List Functions

- Deleting items from the list
  - `list_del(struct list_head *entry);`
  - `list_del_init(struct list_head *entry);`

- Go from list_head to surrounding struct
  - `list_entry(struct list_head *ptr, type_of_struct, field_name);`
Other Linked List Functions

- Iterate through a linked list
  - list_for_each(struct list_head *cursor, struct list_head *list)

Elevator Scheduling
General Advise

- Just make elevator work first
  - Use a very simple algorithm
- Optimize if there is time
FIFO

- **Method:**
  - Service requests in order they arrive

- **Pros:**
  - + No computational time expended to choose requests
  - + Implementation: As easy as it gets
  - + Fair: Every request gets a turn

- **Cons:**
  - - Terrible for random requests!
    - Elevator may backtrack back and forth several times in processing requests
  - - Low throughput potential
Shortest Seek Time First

- **Method:**
  - Service requests near elevator first

- **Pros:**
  - + Very low computational time expended
  - + Implementation: Fairly easy
  - + Throughput: Fairly high – may minimize seek time

- **Cons:**
  - - Unfair: May starve distant requests
  - - May ignore distant clusters of requests
**Method:**
- Service requests in one direction, then go backwards

**Pros:**
- + Very low computational time expended
- + Implementation: Fairly easy (go up, go down, go up)
- + Throughput: somewhat high
- + Fair: No starvation

**Cons:**
- - May take up to two elevator trips to collect a set of requests (up, down)
**Method:**
- Improvement on SCAN – same method (almost)
- Uses information of request locations to determine where to stop going in a certain direction.

**Pros:**
- Low computational time expended
- Implementation: Moderate (set upper bound, go up, set lower bound, go down)
- Throughput: somewhat high
- Fair: No starvation

**Cons:**
- May miss a new request at outer boundary just as direction change occurs
Hybrid

- Combine methods, come up with something new
- Up to your creativity
Project Demos and Deliverables
Basic Information

- Project 2 is due on October 26th
  - Due at demo time
- Please sign up for a time to demo your elevator project
- If you wish to use slack days, make an appointment with me on the day you wish to turn in your project
Project Deliverables

- Before demo, please compress and send me the following
  - README
  - Project report
  - Part 1 – source and Makefile
  - Part 2 – source and Makefile
  - Part 3 – source and Makefile

  Just send me files you have added (for example, you don’t need to send me the syscall_table_32.S file)
Demo

- Will only have time to demo Part 3 – elevator
- Please look at grading sheet to understand what I will be looking for
Getting Help

- I will be holding extra help sessions again next week in LOV 16
  - Tuesday: 5:15pm-6:30pm
  - Thursday: 2:00pm-3:15pm

- Regular office hours
  - Alejandro -- M W 9:00am-10:00am and by appointment
  - Sarah – W 12:30pm-1:30pm, F 3:30pm-4:30pm and by appointment
Next Friday

- We are ahead in recitation lectures, so next recitation will be my office hours
  - Replacing the 3:30pm-4:30pm hours on October 22nd
Test Elevator Driver

- Test elevator driver will be posted before the end of the weekend
  - Will randomly test system calls
- Make sure system calls are registered to the following numbers in syscall_table_32.S:
  - `start_elevator` 337
  - `issue_request` 338
  - `stop_elevator` 339
Any Questions?