ECE 391 Discussion Week 7

Announcements & Reminders

- MP2.2 due next Monday (Oct 10) at 5:59pm
- Please submit your MP3 group information by this Friday (Oct 7) at 11:59pm CT
  - Respond below the pinned piazza post.
  - Group member netids (there must be EXACTLY 4 people in your group)

Octrees

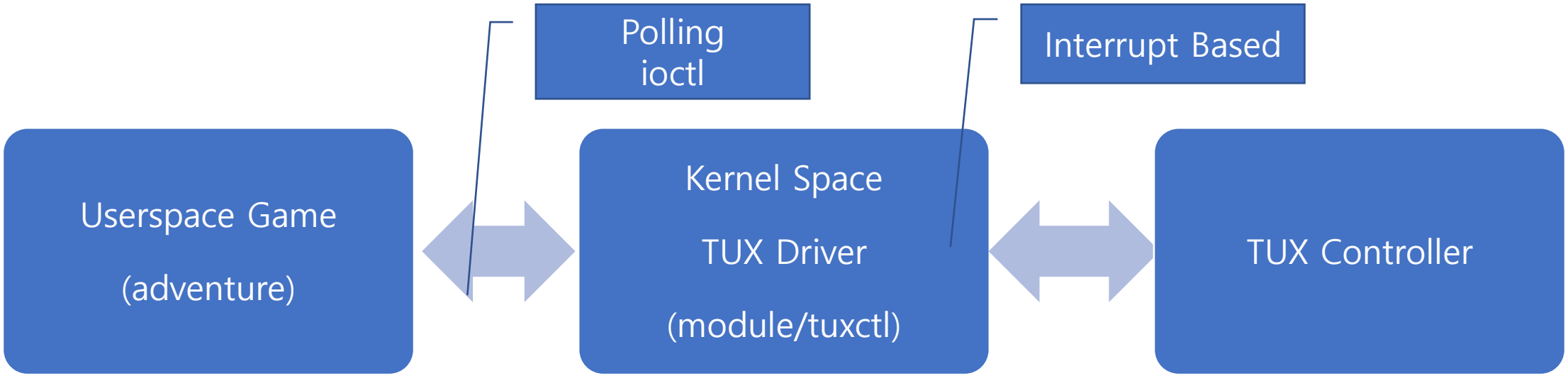
- Algorithm to display images with a multitude of colors on devices that can only display a limited number of colors (color quantizatio n)
- .photo files
  - Each pixel is 16-bits: RRRRRGGGGGGBBBBB
  - 1st 64/256 VGA palette colors already set up and used by game objects
  - Other 192 colors are for you to represent the room photos
- Use arrays, not a pointer-based data structure
- Use 64 colors for the 2nd level nodes and the remaining 128 to re present the nodes in the 4th level
- Don’t leave “holes”!

Octrees (continued)

1. Count the number of pixels in each node at level 4 of your octree
2. Sort the level 4 nodes based on the count and select the most frequent 128
   a. Need to keep track of the original order. How?
3. Calculate the averages for red, green, and blue separately f or the most frequent 128 level 4 nodes and assign them to the palette
   a. Note that red and blue are 5 bits while green is 6 bits!
   b. You should be able to figure out the VGA index from here
4. Repeat 1-3 for level 2 nodes
   a. Remember to remove the contribution of any pixels assigned to the level 4 nodes
   b. There's a more efficient way than just simply repeating steps 1-3 ag ain
5. Finally, reassign the colors to each pixel of the room photo

TUX Driver – Userspace Integration

TUX Driver


Polling
ioctl
Interrupt Based
Userspace Game
(adventure)
Kernel Space
TUX Driver
(module/tuxctl)
TUX Controller

TUX Controller Driver

- Enable control (read/write) of LED displays
- Report button presses and releases – be careful not to switch buttons!
- Handle device reset (save/restore the LED state!)
- Enable game play using the TUX
- Implement Ioctls
  - Do NOT implement TUX_READ_LED
- Testing
  - printk()
  - input.c

```
/* set to 1 and compile this file by itself to test functionality */
#define TEST_INPUT_DRIVER 0

/* set to 1 to use tux controller; otherwise, uses keyboard input */
#define USE_TUX_CONTROLLER 0
```

Synchronization

- Keyboard and Tux controller should both work at the same time
  - Update the same variables
  - Shared variables should be synced
- Look at status_thread() and show_status() to see how to properly integrate the TUX with the game

Definitions

- Thread
  - An independent stream of instructions scheduled to run by the oper ating system
  - Exists within a process and uses the process resources
  - Scheduled within a process, depending on scheduling policy
  - Can be scheduled simultaneously across processors/cores

- pthreads
  - Threading implementation adhering to POSIX standards

Using pthreads

- Creation

```
int pthread_create(thread_id, attributes, thread_function, args);
```

- Termination

```
void pthread_cancel(thread_id);
```

Thread functions

```
void* foo(void * arg){

        while (1) {
                // execute code here
        }

}
```

- Return type is generic pointer. Can be casted to any variable type.
- Argument is generic pointer. Any object can be passed in (and cas ted)

Adding TUX support to Adventure

- Main game loop

```
game_loop() {

        while (1) {
                // display new image/room
                // update status message
                // sleep for 1.5s
                // check keyboard command
                // update game state
        }

}
```

Adding TUX support to Adventure

- Approach 1 – Use main thread

```
game_loop() {

        while (1) {
                // display new image/room
                // update status message
                // sleep for 1.5s

                ioctl(fd, TUX_BUTTONS, &buttons); // poll driver
                switch (buttons) { // update game state }

                // check keyboard command
                // update game state
        }

}
```

Adding TUX support to Adventure

- Approach 2 – Use new thread

```
game_loop() {} // no changes to game_loop

static pthread_t tux_tid;

void* tux_thread(void * arg){
        int buttons = 0;
        while (1) {
                ioctl(fd, TUX_BUTTONS, &buttons); // poll driver
                switch (buttons) { // update game state }
        }
}

main() {
         …
         pthread_create(tux_tid, NULL, tux_thread, NULL);
         …
}
```

Synchronizing threads

- Use a semaphore/mutex!

```
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* foo(void * arg){

        while (1) {
        pthread_mutex_lock(&lock);
        // critical section
        if (data_available) {
                // do something
        }
        pthread_mutex_unlock(&lock);
        }

}
```

Synchronizing threads

- Condition variables – Put thread to sleep until wake up signal received

```
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;

void* foo(void * arg){

        while (1) {
        pthread_mutex_lock(&lock);
        while (!data_available) {
                pthread_cond_wait(&cv, &lock);
                // this also releases the lock
        }
        // data_available, do something
        pthread_mutex_unlock(&lock);
        }
}
```

  - When does the sleeping thread wakeup?
    - When signaling thread gives up lock
    - *pthread_cond_wait* reacquires lock before proceeding with thread

Synchronizing threads

```
int set_data_available(){
        pthread_mutex_lock(&lock);

        data_available = 1;
        pthread_cond_signal(&cv);

        pthread_mutex_unlock(&lock);
        data_available = 0;
}
```

- When does the sleeping thread wakeup?
  - When signaling thread gives up lock
  - *pthread_cond_wait* reacquires lock before proceeding with thread

Adding TUX support to Adventure

- Approach 3 – Use condition variables

```
game_loop() {
  while (1) {
        …
        // poll driver
        ioctl(fd, TUX_BUTTONS, &butto
ns);
        // determine if button presse
d
        pthread_mutex_lock(&lock);
        if (buttons_pressed){
                pthread_cond_signal(&
cv);
        }
        pthread_mutex_unlock(&lock);
        …
  }
}
```

```
void* tux_thread(void * arg){

  while (1) {
    pthread_mutex_lock(&lock);
    while (!buttons_pressed){
      pthread_cond_wait(&cv, &lock);
    }

    switch (buttons) {
      // update game state
    }
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

Good!!