Improved RTOS Project

ECE 5436 Advanced Microprocessor Systems

Team 13

Patrick Kelling

Christopher Andrew

Faiza Badarpura

Project Description

The goal of this project was to improve the RTOS we had been building across the in class labs with a selection of new features and improved versions of old features.

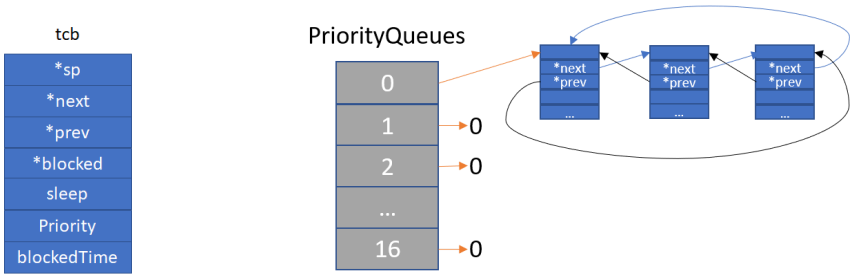
- MLFQ Implementation
- Priority Scheduling
- Bounded waiting
- Events, Queues, Gates
- Memory management
- POSIX Support (partial support)
    - Standard c File IO (fputC, fgetC, fopen with ‘r’ and ‘a’, fclose)
- Other improvements
    - Variable number of threads (up to MAXNUMTHREADS - 10 currently)
    - Improved File organization

Pseudo Code

MLFQ Implementation

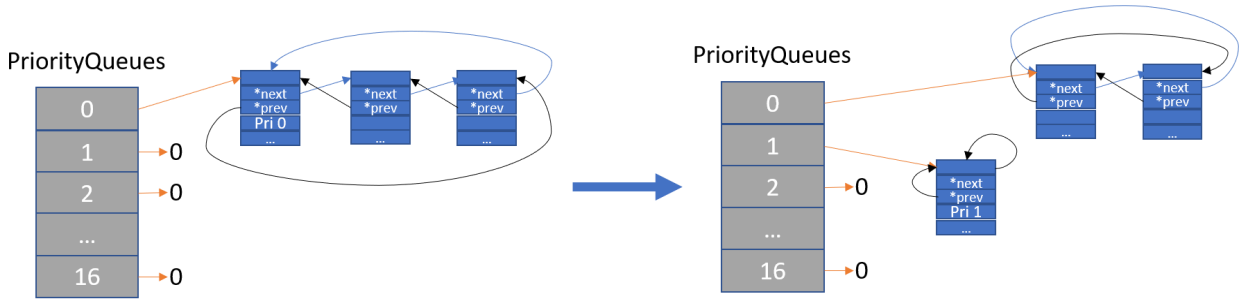
The Multilevel feedback queue allows for tasks to be automatically assigned a priority level. All of the tasks begin at the highest priority, but when a task uses it’s full timeslice, the priority is decremented. When a task suspends itself before using it’s full timeslice, it holds the same priority. Every so often, the priority for all tasks are boosted back to 0.

Thread Control Block structure and Priority Queues Array:

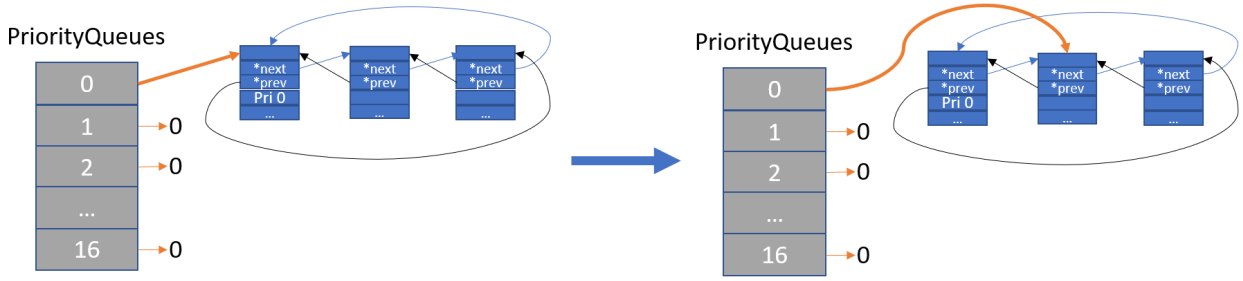


When the OS starts, all threads are added at the highest priority (0). Element 0 of the PriorityQueues array will point to the first threads tcb, and all of the tcbs will be linked together in a doubly linked list. The rest of the elements to the array will be set to zero. When a scheduled thread uses it’s whole timeslice, it is removed from it’s current linked list, and placed in the linked list of the lower priority. If that one is empty, it is linked to itself. The examples of these cases are shown below.

Thread using full timeslice and losing priority:



Thread suspending before full timeslice:



Functions and pseudo code:
updatePriorityQueues() - run by the scheduler to update Priority Queues

If thread used full timeslice:

-Remove tcb from current list, and point priority queue pointer to next tcb

-Add current tcb to next lowest priority linked list, if it is by itself, link it to itself

Else

-Keep current linked list the same

-Increment the PriorityQueues pointer to next tcb

boostPriority() - run every BOOSTPERIOD to put all threads back to highest priority

For all tcb's in tcbs[] array

- Set tcbs[i].next to tcbs[i+1] (if on last tcb, set to tcbs[0])
- Set tcb[i].prev to tcb[i-1] (if on first tcb, set to tcbs[last])

-Point PriorityQueues[0] to first thread's tcb

-Reset all PriorityQueues pointers to 0 (except for first one)

-Set RunPt to first thread tcb

runperiodicevents(void) - run every ms (only MLFQ pertinent code is included here)
-Increment boostCount value
-if (boostCount equals BOOSTPERIOD)
    boostPriority();
    boostCount = 0;

Priority Scheduling

Priority scheduling is achieved by starting at the highest priority queue and going through it until a thread that is not blocked or sleeping is found. If one is not found, the scheduler will go to the next queue and repeat the search. This means that higher priority threads will be scheduled and lower priority ones will only be scheduled if the higher priority ones are blocked.

Scheduler() - run every timeslice or when a thread suspends
-update Priority Queues
- set RunPt = 0
- currPriority = 0,
- pt = PriorityQueues[0] (start at beginning of highest priority linked list)
While RunPt is 0
    -If pt is !blocked and !sleeping
        RunPt = pt;
    -else pt = pt->next

    -if pt is equal to start of linked list, go to next Queue
        currPriority++

        -if currPriority < LOWESTPRIORITY
            Pt = PriorityQueues[0] (didn't find a thread to run, restart from beginning)
            currPriority = 0
        -else
            Pt = PriorityQueues[currPriority]

runperiodicevents(void) - run every ms (only Scheduling pertinent code is shown)
For all threads in tcbs[] array
    -decrement tcbs[i] sleep counter

Bounded waiting

Bounded waiting is achieved by keeping track of how long threads have been blocked in their tcb. When OS_Signal is called, it will search the tcbs[] array for the thread that has been blocked the longest (this is regardless of priority).

Functions:

OS_Signal(int32_t *semaPt) - signals blocked tcb's to unblock
maxBlockTime = 0, *longestWaitingThread = 0;
DisableInterrupts();
Increment *semaPt
If *semaPt <= 0
    For all tcb in tcbs[] array
        -find longest waiting tcb blocked on this semaPt
        -unblock it and set blockedTime to 0

runperiodicevents(void) - run every ms (only Bounded waiting code is shown)
For all threads in tcbs[] array
    -increment tcbs[i] blocked counter for blocked threads

Events, Queues, Gates

Events

The goal of events are to have threads that wait until they are signaled to run once then return to waiting. Comes in two forms, normal threads that are just waiting to be signaled, and threads that must be run at a fixed interval and return after completing.

int OS_AddPeriodicEventThread(void(*thread)(void), uint32_t period)

This function's goal is to add a thread to the array of periodic events if there is room in the list of periodic events and assign the provided period to that periodic function tcb.

void static runperiodicevents(void)

This function is ran every 1 [ms] and its function is to update the sleep, blocking and timers for normal threads. In addition it periodically resets the priority of all threads for the MLFQ implementation. For event threads this function decrements run counters and then runs the event if the timer reaches 0, followed by resetting the counter value.

Queues

Queues are a data structure similar to FIFO where the first entry is the first to leave and the last entry is the last to leave, the main difference is that a queue does not have a fixed size.

struct queue* OS_queue_create()

Creates the queue by creating the base queue structure, setting the initial to and bottom pointers to 0, and then returning the created structure.

void OS_queue_push(void* p_data, struct queue* p_workingQueue)

Adds an entry into the queue at the bottom of the list.

1. Wait to claim control of the queue to prevent multiple threads accessing.
2. Create a new data box and insert the p_data pointer into it.
3. Check if anything is in the queue, if nothing is setup this data box as the top and bottom positions
4. Add the created data box to the next entry for the bottom of the queue
5. Save the created data box to the bottom of the queue
6. Signal queue mutex to allow another to claim control

void* OS_queue_pop(struct queue* p_workingQueue)

Removes the top most piece of data from the queue and returns it.

1. Wait to claim control of the queue to prevent multiple threads accessing.
2. Check if the top of the queue does not exist
    a. If it doesn't signal mutex and return 0
3. Save the listDataBox in top to a temp pointer.
4. Save the data pointer from the box to a temp location
5. Assign the top->next to top
6. Free the now empty data box
7. Signal mutex
8. Return data stored in the box

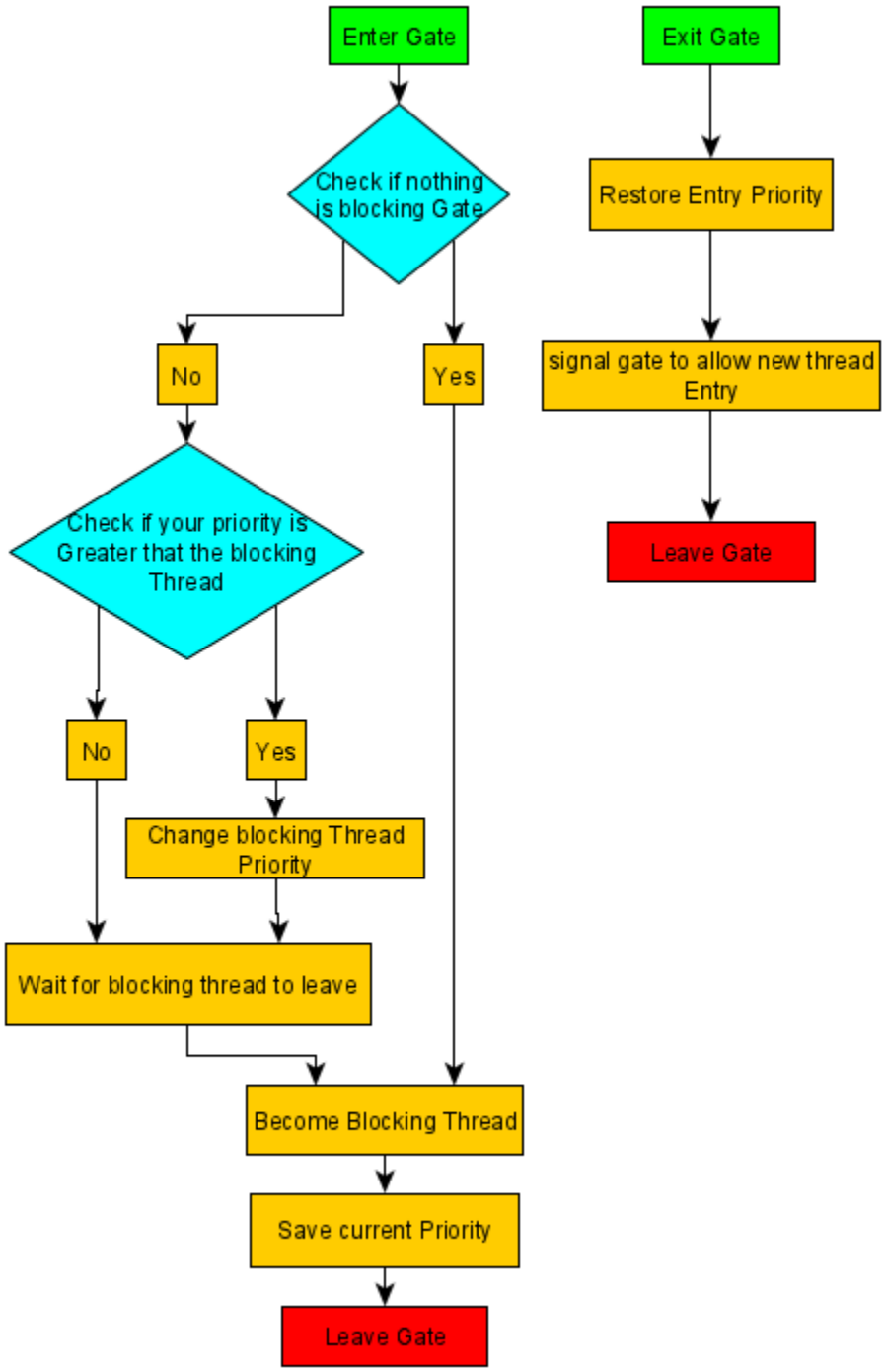
void* OS_queue_peak(struct queue* p_workingQueue)

This function allows the ability to access the top most entry in the queue without removing the item if it exists.

1. Check if the bottom of the queue exists
    a. Return the data entry contained in the data box
    b. Return 0;

Gates

Gates at their most basic level are a form of mutex that seeks to avoid the issue of priority inversion where a low priority thread can block and or reduce the effective priority of an high priority thread. Gates entry and exit functions are designed to be used in pairs and creates “blocks” of gate controlled code.



Gate Entry Exit general Flow Chart

void OS_gate_init(struct gate* gate)

This function initializes the gates internal variables.

void OS_gate_entry(struct gate* gate)

Controls entry into the gate and manages the priority of the blocking thread

1. Disable interrupts, this function is sensitive and should not have a thread switch while it is inside.
2. Check if the gate_Mutex == 1 and nothing is blocking the gate
    a. If this is true set gate_block == RunPt and set the restorePriority to be equal to the RunPt priority
3. Else if check to make sure the gate_block exists then check if RunPt priority is greater than gate_block priority
    a. If it is greater we need to change gate_blocks priority
    b. Remove it from the current priority queue and stitch it closed or remove it if that was the last entry
    c. Set gate_blocks priority to the new one
    d. Insert it into the new priority queue
4. Once all this priority manipulation is done we wait at the gate mutex
5. Once past the wait mutex this thread becomes the new gate_block and the new restore priority
6. Enable interrupts and return

void OS_gate_exit(struct gate* gate)

This function controls exits from the gate and restores thread priority to the value it had when being told to run by gate_entry().

1. Disable interrupts, being forced to swap threads while editing the priority queues would be bad
2. Remove the current thread from its priority queue
3. Reset its priority to its restorePriority value.
4. Set gate_block to 0
5. Set restorePriority to LOWESTPRIORITY
6. Signal the gate mutex to allow a new thread to finish entering
7. Enable interrupts

Memory management

This implementation includes three functions:

- void *malloc(size_t sizem)
- void free(void* ptr)
- void initialize()

Implementation of the malloc function:

void *malloc(size_t size)-

- Uses a structure to allocate initial memory
    - Pointer of type struct called next
    - integer of type size_t called size
    - Integer called free
-

void free(void *ptr)-

- Used to free up space that was allocated

void initialize(void)-

- Used to initialize initial memory chunk

POSIX Support (partial support)

For POSIX support we focus on File I/O and writing a stdio file that would work with our filesystem. (This involved making some filesystem changes).

FileSystem Changes:

-Use 2 character filenames (Ex- “aa”) - stored in sector 254
-Add Filenames[512] buffer when Loading directory from flash memory
-Allow for appending within sectors so no internal fragmentation occurs.

Note: This system only allows for 1 open file at a time, it is recommended to use a semaphore to control access to the system if multiple threads will use it.

Functions: (Filesystem.c - formerly eFile.c)

OS_MountDirectory() - Used to fill Directory, FAT, and Filenames buffers
-Read Sector 255 into buffer and copy that into Directory[] and FAT[] accordingly
-Read Sector 254 into Filenames buffer
-bDirectoryLoaded = 1

getFilenum(const char* filename) - Gets file number corresponding to a filename (2 characters)
while(Filenames[i:i+1] does not equal filename and i < 255)

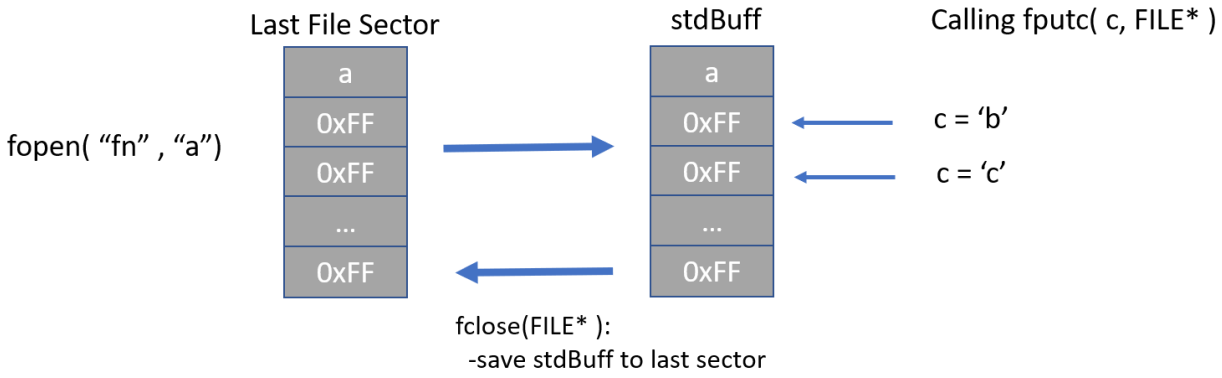
I++;
Return i; (255 means filename doesn't exist)

OS_File_RewriteLastSector(uint8_t num, uint8_t buf[512]) - Used for appending within sector
-Get last sector number of file (using lastsector function)
-call eDisk_WriteSector() function for buf and last sector of file

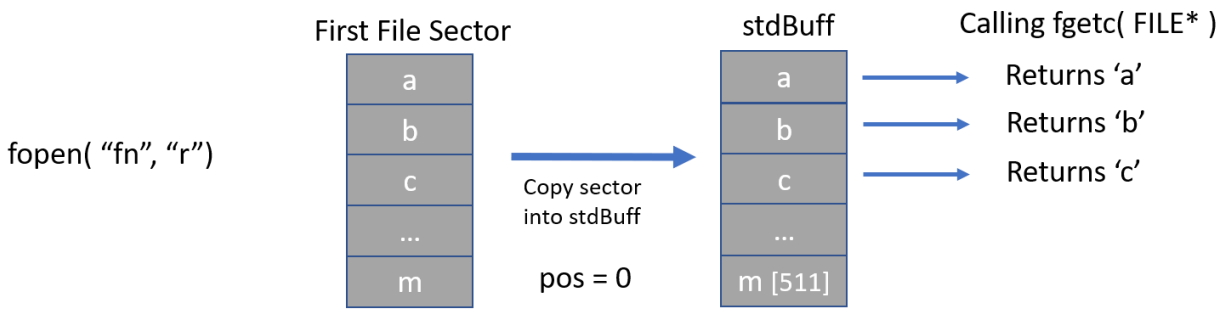
Note: Writing to a sector can only change 1 bits to 0, so this should only be used for appending. If it tries to change part of the sector that has already been written, the result will not be as expected.

Functions: (our_stdio.c)

Append Mode:
When the file is opened in append mode, it will search for the file in the Filenames array. If it is found, it will find the last sector of the file, and copy it into the stdBuff. When fputc() is called, all changes are made in the buffer, not in flash memory. When fclose() is called the stdBuff sector will be saved into non-volatile flash memory, as well as the Filenames, FAT, and Directory buffers, therefore, fclose must be called for changes to be saved.
If the filename was not found, the file will be created. If the stdBuff is full, it will be saved to flash memory, a new sector will be appended to the file, and the stdBuff will be emptied.



Read Mode:
Read mode allows a user to read from a file 1 character at a time. When fopen() is called, the filename provided is searched for in the Filenames array. If it is not found, a value of 0 is returned, if it is found, the first sector in the file is copied into the stdBuff. When the user calls fgetc(), the character at the current position is returned. If the end of the stdBuff is reached, the next sector will be pulled from memory into the stdBuff to be read.



FILE* fopen(const char *filename, const char *mode) - function to open a file (only ‘a’ and ‘r’ modes are supported)
-get file number using getFilenum function
-if file number equals 255, file does’n exist
    If (mode is append)
        -Create new file using OS_File_New(filename)
        -clear stdBuff elements to 0xFF (all 1 bits)
        -call OS_File_Append(file number, stdBuff)
    Else
        Error occurred, cannot read a file that doesn’t exist

-Set openFile structure info and fileIsOpen = 1

    Load file into stdBuf (OS_File_Read) - 0 for ‘r’ and (fileSize - 1) for ‘a’

-return &openFile;

FILE* fclose(FILE* fp) - closes the file connected to fp
-if fp is valid
    If mode == ‘a’, Rewrite Last Sector and call OS_File_Flush()
    Reset openFile structure info and set fileIsOpen = 0

int fputc(char c, FILE *fp) - put character into file
-Find next open position in file (stdBuff[i] == 0xFF) or position == 512 if none are open
-if position == 512
    -Save stdBuff using RewriteLastSector()
    -Clear the stdBuff (all values = 0xFF)
    -call OS_File_Append to add a new empty sector to the file
    -stdBuff[0] = c (character input will be first char of new sector)
-else
    -stdBuff[i] = c (note, new chars are only added to buffer, need to close file to save data)

char fgetc(FILE *fp) - get next character from file and increment position
-if currSectorPos == 512
	-open next sector using OS_File_Read()
	-if that returns 255, no other sector exists, so return EOF (0xFF)
Else
	-c = stdBuff[currSectorPos]
	-increment position (currSectorPos++)
Return c;