P2 tasks by file

os345.h

Define the structs and functions for a priority queue, this struct will be used to queue up the tasks that are ready to run, and for the blocked queues.

```
// priority queue structs
typedef struct node
{
    int tid;
    int priority;
    struct node *next;
} Node;

typedef struct pq
{
    Node *head;
} PQ;

int enq(PQ *q, int tid);
int deq(PQ *q);
int del_task(PQ *q, int tid);
void printq(PQ *q);
```

Add a blocked Q to the sempahore struct, each sempahore will keep track of the tasks that are blocked waiting for the resource that semaphore protects

```
typedef struct semaphore // semaphore
{
    struct semaphore *semLink; // semaphore link
    char *name;                // semaphore name
    int state;                 // semaphore state
    int type;                  // semaphore type
    int taskNum;               // semaphore creator task #
    PQ blockedQ;
} Semaphore;
```

os345.c

I will talk about the scheduler part of os345.c at the end.

Define 3 new global variables, a ready queue, a sempahjore that will be signaled every ten seconds in interrupts.c and a time varialbe to keep track of ten second intervals

```
Semaphore *tics10sec;   // 10 second semaphore
time_t oldTime10; // old 10sec time
PQ readyQ;
```

Initialize the semaphore in main and the other two in init_os

```
int main(int argc, char *argv[])
{
        .
        .
        .
        tics1sec = createSemaphore("tics1sec", BINARY, 0);
        tics10sec = createSemaphore("tics10sec", BINARY, 0);
        tics10thsec = createSemaphore("tics10thsec", BINARY, 0);
```

```
static int initOS()
{
        .
        .
        .
        // initialize queue to empty
        readyQ.head = NULL;

        // capture current time
        lastPollClock = clock(); // last pollClock
        time(&oldTime1);
        time(&oldTime10);
```

Make sure you free the ready queue memory when the os quits, we don't need to do the blocked q's here as that will be done by delete sempahore in semaphores.c

```
void powerDown(int code)
{
        // ?? free ready queue
        // ?? call deq until the readyQ is empty
```

```
    // ?? release any other system resources
    // ?? deltaclock (project 3)

    RESTORE_OS
    return;
} // end powerDown
```

Implement the queing functions defined in os345.h somewhere in os345.c

```
void enq(PQ *q, int tid)
{

}

int deq(PQ *q)
{

}

void del_task(PQ *q, int tid)
{

}

void printq(PQ *q)
{
    Node *cur = q->head;
    printf("Queue Contents:\n");
    while (cur)
    {
        printf("[%d] p:%d\n", cur->tid, cur->priority);
        cur = cur->next;
    }
    printf("\n");
}
```

---

os345tasks.c

bring in the global definition of the readyQ variable defined in os345.c

```
extern PQ readyQ;
```

modify createTask to put newly created tasks in the global readyQ

```
// ?? may require inserting task into "ready" queue
enq(&readyQ, tid);
```

modifty exitTask to make sure tasks that are killed can be scheduled for deletion. For instance if a task is on a blocked Q waiting for a resource and then I call killTask on the task, is needs to be removed from it's blocked Q, its state becomes exit and it needs to go on the ready Q do that it can delete itself when schedueld and clean up all of its memory.

```
static void exitTask(int taskId)
{
    assert("exitTaskError" && tcb[taskId].name);

    // 1. find task in system queue
    // 2. if blocked, unblock (handle semaphore)
    // 3. set state to exit

    // ?? add code here
    Semaphore *s = tcb[taskId].event;
    if (s)
    {
        // delete from blocked Q
        // add to ready Q
    }

    tcb[taskId].state = S_EXIT; // EXIT task state
    return;
} // end exitTask
```

In system kill task make sure that the item being killed is not on the readyQ anymore and it will no longer be defined

```
int sysKillTask(int taskId)
{
    .
    .
    .

    // ?? delete task from system queue

    for (int i = 0; i < tcb[taskId].argc; i++)
    {
        free(tcb[taskId].argv[i]);
    }
    free(tcb[taskId].argv);
```

```
    tcb[taskId].name = 0; // release tcb slot
    return 0;
} // end sysKillTask
```

---

os345interrupts.c

bring in the definition of your time varaibles defined in os345.c

```
extern Semaphore *tics10sec;   // 10 second semaphore
extern time_t oldTime10; // old 1sec time
```

modify the timer interupt service routine to signal your semaphore everty ten seconds

```
static void timer_isr()
{
    .
    .
    .
    // one second timer
    if ((currentTime - oldTime1) >= 1)
    {
        // signal 1 second
        semSignal(tics1sec);
        oldTime1 += 1;
    }
    .
    .
    .
    // ?? add other timer sampling/signaling code here for project 2
    // make sure it is for ten seconds and not 1
    return;
} // end timer_isr
```

---

os345sempahores.c

bring in the readyQ

```
extern PQ readyQ;
```

modify createSempahore to initialize it's blockedQ

```
Semaphore *createSemaphore(char *name, int type, int state)
{
    .
    .
    .
    // set semaphore values
    sem->name = (char *)malloc(strlen(name) + 1);
    strcpy(sem->name, name); // semaphore name
    sem->type = type;        // 0=binary, 1=counting
    sem->state = state;      // initial semaphore state
    sem->taskNum = curTask;  // set parent task #
    //?? initialize blocked Q
    .
    .
    .
} // end createSemaphore
```

modify delete sempahore such that is cleans up it's own blockedQ when deleted. Note that if a process is waiting on a semaphore that gets deleted we will need to schedule it to exit. So when a semphore gets deleted we should go thorugh its blocked Q and set all of the tasks on it to be in the exit state and add them to the ready Q

```
bool deleteSemaphore(Semaphore **semaphore)
{
                        .
                        .
                        .
            // ?? remove items from sem->blockedQ
            // set tcb[item_id].state to S_EXIT
            // add items to readyQ
            free(sem->name);
            free(sem);

            return TRUE;
        }
        .
        .
        .
    }

    // could not delete
    return FALSE;
} // end deleteSemaphore
```

Implement semWait and semSignal, remember that if semWait is called and the state is 0 the task should be blocked on that semaphores Q

For semSignal we should remove the highest priority task from the semaphores blocked Q and put it on the ready Q. make sure you set the tasks states correctly

For the coutning semaphores everything is essentially the same except that instaed of setting the state to be 0 or 1 you will be incrementing and decrementing the state (note the state can never be negative)

Lastly in semSignal the value does not increment or get set to 1 if a blocked process was removed from the Q because that process immediatley comsumes the signal

---

os345p2.c

extern in an variables you think you made need (hint you created a new sempahore you will want)

declare a new function, you cna name it whatever you want

```
int timeTask(int, char **);
```

implement this function as follows

```
char *myTime(char *svtime)
{
    time_t cTime; // current time

    time(&cTime); // read current time
    strcpy(svtime, asctime(localtime(&cTime)));
    svtime[strlen(svtime) - 1] = 0; // eliminate nl at end
    return svtime;
} // end myTime

int timeTask(int argc, char *argv[])
{
    while (1)
    {
        // ?? print the task id and the current time using the above
function
        // ?? wait for ten seconds using a semaphore
    }
```

```
    return 0; // terminate task
}
```

create ten instances of your timer task with a loop that start when you type p2 in your shell

```
int P2_main(int argc, char *argv[])
{
        .
        .
        .
    //?? looop ten times and create 10 timer tasks
    //?? tasks have an inifnite loop that wait on tics10sec
    //?? in the loop they print thier id and the current time
    .
    .
    .
    createTask("I'm Alive",  // task name
               ImAliveTask,  // task
               LOW_PRIORITY, // task priority
               2,            // task argc
               aliveArgv);   // task argument pointers
    return 0;
} // end P2_project2
```

---

os345.c again

you should be able to run p2 at this point without chaning the scheduler. I would use print statements and your printq function to verify that your q holds that values you are expecting before actually changing the scheduler to use your readyQ

now change the scheduler function to use your ready Q

you also might want to change the initial value for curTask in init_os to -1

```
static int scheduler()
{
        // 1. the curTask should go on the readyQ if it is valid and not
blocked
        // 2. Get a new task to run (nextTask) from the top of the readyQ
        // 3. If the task stop signal is set return it to the Q and set
nextTask to -1
        // 4. return the next task
}
```

os345p2.c optional

If you would like the easiest way to debug is to modify the P2_listTasks function to print out all of the queues as described in the prohject description. this way you can type "lt" and see the queues anytime you want.