CS3500-OPERATING SYSTEMS

O(1) SCHEDULER FOR XV6

Name	B.V.S Sudheendra
Roll No.	CS18B006
Department	Computer Science and Engineering
Email	bvssudhindra@gmail.com
Date	December 10th, 2020

Contents

1. Introduction

Following are all the files in which I am making changes, and a brief description about the changes I made :-

- **main.c** Added a call to a function which will initialize my queue arrays.
- **proc.c** This is where most of the code resides and hence, most of the changes are done here. Refer proc.c section more clarity.
- **proc.h** Modified the proc structure to hold more additional information about a process like priority, sleeping time etc.
- **defs.h** Added all the new function prototypes here.
- **sysproc.c** Added a couple of new syscalls.

I have also defined a few new files :-

- **ps.c** This will call the new system call which will print all the current processes and their status i.e, RUNNABLE/SLEEPING/RUNNING and their priorities.
- **queue.h** This contains all of my implementation for queue's holding processes.
- **test.c** Custom user-test case to check the working of my scheduler.

We boot our xv6 OS using the command `make qemu`.

2. The Logic

2.1 Queue

For making a O(1) scheduler, we need a queue type structure which stores proc structures in a FIFO (First In First Out) type of manner. I am implementing queue using arrays and storing appropriate indices to point to start and end of the queue. Hence inserting becomes O(1) using the pointer pointing to the rear of the queue, and first element can be accessed in O(1) using the front pointer. Refer this for more details.
Each item in the queue is a pointer to a proc structure. Now, in order to maintain lists of active(running) and expired(passive) queues, I have defined two arrays of type queue * of length 40. These 2 will serve as my active and expired queue lists. In order to find the first array index with non-zero queue in O(1), we will be using a bitmask. Bitmask is essentially a integer which tells me the first non-zero queue index in my active process list. In our case, bitmask is a 40-bit long integer which stores the indices of the array which are non-empty. If an index 'i' in the active queue list is non-empty, then bitmask[i] is set to 1, and the empty queue indices are set to 0. This way, we can find out the first non-empty queue in the queue list by simply iterating through the bitmask.

2.2 Proc Structure

We need to make some changes to the proc structure before we can start with the implementation of the scheduler. For starters, I have added an integer spriority, and dpriority which represent the static and dynamic priority of each process. Both of these will have a default value of 120 for all Normal processes. Next up, we need to store the sleeping time, and the number of times the process goes to sleep to find the avg. sleeping time as will be discussed below. Also required is the timeslice to be defined for each process. I have added all of these to the struct proc in proc.h.

2.3 Handling Priorities

There are two types of processes viz, Real Time and Normal. Real Time processes have priorities between 0 and 99, where as Normal processes have priorities from 100 to 139. We wont be dealing with Real Time processes in this scheduler, so we assume all processes to be scheduled as Normal processes. By default, a new process entering is given a priority 120. It is important to note that lesser the priority number, more is the actual priority given to that process. We can modify the priority of a process by using the *nice* command. Since the priorities are to be between 100 and 139, nice can only make changes to the priority by values ranging from -20(making the effective priority 100) to +19(making the effective priority 139).

Now, before moving on to the dynamic priorities, we define a *bonus* value for each process, which represents the avg. sleeping time of each process. The table below lists the bonus values for different avg. sleeping times :-

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

Bonus values are very helpful to prevent starvation. The longer a process waits/sleeps, the higher the priority should be, hence higher the bonus, the lower the dynamic priority number.Keeping all this in mind, the dynamic priority of a process is given by
dynamic_priority = max(100, min(static_priority - bonus + 5, 139))
Based on this dynamic priority, we will be adding processes to the expired queue list, as we will see shortly. An additional 5 is added to the priority to ensure that we take into account the fact that that process has already run for a while.

2.4 Timeslice

To prevent process starvation, we dont want to execute the next process only when the previous process has totally completed its execution. So, instead we make note of timeslices for each process such that, if the process ends up running for more than the corresponding timeslice, then we call a timer interrupt and move the process from the active queue list to the expired queue list. This way, processes wont starve and all processes will get a chance to execute.
Again, timeslice is different for different processes according to their priority. If the priority is < 120, then we need to give it a larger timeslice to be fair to the *more* important processes, as compared to the ones with priority between 120 and 139. We will be assigning timeslice to each process as follows :-
If priority < 120 :- timeslice = (140 - priority) * 20 ms.
Else if priority < 140 :- timeslice = (140 - priority) * 5 ms.

2.5 Scheduler logic

Now that we have defined and implemented all the stuff required, lets move to the scheduler itself. The scheduler basically consists of two queue lists i.e, active queue list(AQ), and the expired queue list(EQ). AQ, EQ are ordered collections of 40 queues (For processes with priorities from 100 to 139, 0th index indicating the queue of processes with priority 100, and 39th index the queue of processes with priority 139) . As discussed earlier, I have stored my queues in array's, each of length 40. AQ stores the process which are yet to be executed next, where as the processes which have done their time are stored in the EQ. We first take the first non-empty queue from AQ (first implies the queue with lowest priority, then get the first process in that queue, and schedule it. If the scheduled process finishes its execution before the end of its timeslice, then we dont need that process anymore. However if it is taking longer than the assigned timeslice, then we call a timer interrupt and move this process to the EQ with its dynamic priority as the priority number. Again, we choose the first non-empty queue in AQ, then schedule it and so on, until the AQ becomes empty.Whenever any of the queues in AQ become empty, we need to unset the bitmask at that index. When the AQ is empty, we need to swap the EQ, and the AQ i.e, the EQ becomes the new AQ, and the previous AQ (which was empty) becomes the new EQ. We should be mindful of the fact that we have to swap the bitmasks as well. This goes on until both the queue lists become empty, which essentially means that there are no more processes to be scheduled. When we want to include a new process, then we can simply insert it into the expired queue, and then the scheduler will take care of scheduling it by swapping AQ, EQ (AQ is empty, where as EQ has the new process) then popping the process and scheduling it.

2.6 Time Complexity

We still are yet to prove that all this is only O(1). Simply put, our scheduler consists of two parts :- Finding the first non-empty queue in the queue list, and then getting the first process in that queue. Step 2 is obviously constant time. For Step 1, we have maintained a bitmask, and finding the first non-empty bit here is simply iterating through the bits of our bitmask and hence our scheduler is O(1).

3. Code snippets

3.1 proc.h

In this file, we need to add the required variables into the proc structure.

```
// Static and dynamic priorities.
int spriority, dpriority;

// Timeslice allotted for the process.
uint64 timeslice;

// The number of sleeps for the process.
int s_no;

// Total sleep time and a helper variable to store
// the time when the process goes to sleep.
uint64 stime, last_time;
```

3.2 queue.h

This is the header file which contains all the queue structures and the helper function associated with it. The queue is going to be emulated using a circular array. Refer this for more details.

```
// The type - 'queue'
struct Queue{
    // Variables to store the start, end and size of the queue
    // which is being emulated by using an array
    int front, rear, size;

    // The array of struct proc * 's.
    struct proc* array[100];
};
```

The insert and del functions insert and remove a process from the queue.

```
void insert(struct Queue* queue, struct proc* item)
{
    // Rear points to the last item, hence increment it.
    queue->rear = (queue->rear + 1)%100;

    // Insert at that new rear position.
    queue->array[queue->rear] = item;

    // Size increases by 1.
    queue->size++;

    return;
}

struct proc* del(struct Queue* queue)
{
    // Get the first process inserted, by accsessing the front index.
    struct proc* item = queue->array[queue->front];

    // Updated front index will be the next index.
    queue->front = (queue->front + 1)%100;

    // Size decreases by 1.
    queue->size--;

    // Return the first process.
    return item;
}
```

The queue list is a struct of array of Queues of length 40, and a lock and bitmask on top of it.

```
// The type of list of 40 queues
struct queue_list{
  // To acquire and release the locks for the queue list.
  struct spinlock lock;

  // The list of 40 queues.
  struct Queue Q[40];

  // Bitmask, to store the indices having non-empty queues.
  uint64 bitmask;
};
```

3.3 proc.c

We need to initialize the variables we have defined in proc.h. This is done in procinit(), and freeproc().

```
// Initialize the variables inside the proc structure.
    p->spriority=p->dpriority=120;
    p->stime=p->s_no=0;
    p->timeslice=p->last_time=0;
```

I have defined two insert functions to insert the proc according to different conditions. We need to find the dynamic priority of the process every time we need to insert the process.

```
// This function will calculate the dynamic priority
// of the process, and then insert according to that
// priority. Used when the process has already been
// scheeduled atleast once
void
insert_expQ(struct queue_list* q,struct proc* p){

  // Acquire the lock for the queue list.
  acquire(&q->lock);

  // Bonus value for each process.
  int bonus=-1;

  // Avg Sleep time calculated as the total
  // sleep time divided by the number of sleeps.
  int AvgSleepTime = p->stime/p->s_no;

  // Find the bonus value for the correspnding
  // Avg sleep time.
  for(int i=1000000;i<10000000;){
    if(AvgSleepTime < i){
      bonus=(int)(i/1000000)-1;
      break;
    }
    i+=1000000;
  }
  if(bonus==-1)bonus=10;

  // Assign the dynamic priority according to the
  // static priority and the bonus value.
  p->dpriority = max(100,min(p->spriority-bonus+5,139));

  // Insert into the corresponding index which is
  // the dynamic priority - 100.
  insert(&(q->Q[p->dpriority-100]),p);

  // Set that bit of the bitmask as 1, which is done
  // by bitwise or with bit 1 at that position.
  q->bitmask = q->bitmask | ((uint64)1L << (p->dpriority-100));

  // Finally release the lock for that queue list.
  release(&q->lock);
}
```

When we want to insert for the first time, we use this function. This is called in userinit() and fork().

```
// This function will insert a process according to its
// static priority. Used when we are about to schedule
// a process for the first time.
void
insertQ(struct queue_list* q,struct proc* p){

  acquire(&q->lock);

  // Insert into the corresponding index which is
  // the priority - 100.
  insert(&(q->Q[p->spriority-100]),p);

  // Set that bit of the bitmask as 1, which is done
  // by bitwise or with bit 1 at that position.
  q->bitmask = q->bitmask | ((uint64)1L << (p->spriority-100));

  // Finally release the lock for that queue list.
  release(&q->lock);
  return;
}
```

Also, where ever we set p→state to RUNNABLE, we call insert_expQ i.e, in yield(), wakeup(), wakeup1(), and kill().
Other than all this, we make changes in the scheduler function which is listed in the next section.

3.4 scheduler()

After we acquire the locks for AQ, EQ, we check the bitmasks of both of them. If bitmask of AQ is $= 0$, EQ is $\neq 0$, then swap, else-if both bitmasks of AQ and EQ are 0, then release the locks and continue.

```
// If the active run queue is empty, and the expired run queue
// is non-empty, then swap the queue lists.
if(active_runQ.bitmask==0 && expired_runQ.bitmask!=0){
  struct Queue tempq;
  for(int i=0;i<40;i++){
    tempq=active_runQ.Q[i];
    active_runQ.Q[i]=expired_runQ.Q[i];
    expired_runQ.Q[i]=tempq;
  }
  uint64 temp=active_runQ.bitmask;
  active_runQ.bitmask=expired_runQ.bitmask;
  expired_runQ.bitmask=temp;
}

// If both are empty, then continue through the infinite loop
else if(active_runQ.bitmask==0 && expired_runQ.bitmask==0){
  release(&(active_runQ.lock));
  release(&(expired_runQ.lock));
  continue;
}
```

Next, we find the index of the first non empty queue in AQ, then get the first process. After we have the process to be scheduled, we calculate the timeslice for that process.

```
// Calculating the timeslice for the process this time
// based on its current dynamic priority.
// NOTE :- 1ms == 10000 units.
if(p->dpriority < 120){
  p->timeslice=(uint64)((140-p->dpriority)*200000); // 20ms
}
else p->timeslice=(uint64)((140-p->dpriority)*50000); // 5ms.
```

Next up we need to make changes to MTIME_CMP to make sure that a timer interrupt occours if the process exceeds its time quantum.

```
// Get the CPU id to increment MTIME_CMP field.
// MTIMT_CMP = MTIME + timeslice essentially means that
// a timer interrupt will be called after timeslice
// amount of time since then MTIME will exceed MTIME_CMP.
int id = cpuid();
*(uint64*)CLINT_MTIMECMP(id) = *(uint64*)CLINT_MTIME + p->timeslice;
```

4. Testing

4.1 Scheduling Time

To test that our scheduler indeed does schedule processes in O(1) time, I have made a user testcase called test.c. This function will take as argument the number of forks to be done.

```c
int main(int argc, char* argv[])
{
    set_flag_1();
    int pid;
    int i;
    for(i=0; i < atoi(argv[1]); i++){
        pid=fork();  // Create new processes.

        if(pid==0){
            // Some garbage calculations to keep the CPU busy.
            int x=0;
            x = i*3.14*1231*100 - x;
        }
    }
    set_flag_0();
    exit(0);

}
```

Also, in my scheduler, at the start of the loop, I store the current time as start_time, and just before we context switch using the swth() function, I print the current time - start_time. This essentially gives me the time taken to schedule a process. The set_flag's are system calls which will set a particular flags in the scheduler so that I dont print the scheduling time for every process every time, but only for this process.
Now, I will compare the time taken for my scheduler to schedule this process with the time taken by the default scheduler of xv6-riscv for different number of processes (Which essentially means different number of forks which is again equivalent to calling the process with different arguments).

4.1.1 Default Scheduler

While using the default scheduler, the time taken to schedule is as follows :-

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ test 1
 Time taken : 840
$ test 100
 Time taken : 1009
 Time taken : 1351
 Time taken : 104532
 Time taken : 168674
 Time taken : 113642
$ test 1000
 Time taken : 1385
 Time taken : 7009
 Time taken : 676860
 Time taken : 613498
 Time taken : 615395
 Time taken : 1678766
 Time taken : 1615901
 Time taken : 1620233
 Time taken : 2677037
 Time taken : 2613882
 Time taken : 2619575
 Time taken : 3677117
 Time taken : 3614432
 Time taken : 3617668
```

For 10,000, we get :-

```
 Time taken : 19000538
 Time taken : 48999665
 Time taken : 18997556
 Time taken : 20000104
 Time taken : 50000885
 Time taken : 20001801
 Time taken : 50997465
```

Clearly, as the number of forks keep on increasing the scheduling time keeps on increasing at almost around the same rate. This hints us that the default scheduler is O(N).

4.1.2 Our O(1) Scheduler

Doing the same tests with our O(1) scheduler, we get :-

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ test 1
Time taken : 437
$ test 100
Time taken : 370
Time taken : 317
$ test 1000
Time taken : 362
Time taken : 338
Time taken : 358
Time taken : 81
Time taken : 67
Time taken : 141
Time taken : 61
Time taken : 65
Time taken : 81
Time taken : 71
Time taken : 60
Time taken : 248
$ test 10000
Time taken : 324
Time taken : 334
Time taken : 352
Time taken : 67
Time taken : 92
Time taken : 116
Time taken : 99
Time taken : 57
Time taken : 67
Time taken : 63
Time taken : 61
Time taken : 116
Time taken : 82
Time taken : 72
Time taken : 114
Time taken : 70
Time taken : 249
```

In this case, as we can see the scheduling time is almost constant and in the same order for any number of forks which implies that our scheduler is O(1).
Hence our scheduler does indeed work in O(1) time.
.

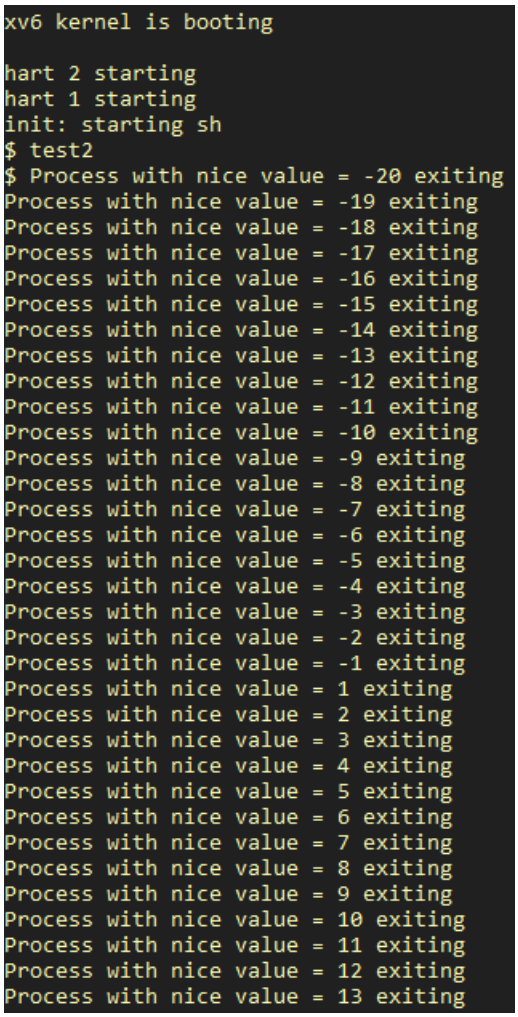
4.2 Nice values

I have made another testcase called test2.c which goes as follows :-

```c
int pid;
for(int i=-20; i<20; i++){
    pid = fork();
    if(pid==0){
        // Change the priority of different processes differently
        nice(i);

        // Consume CPU time
        for(int k=0;k<100;k++){
            for(int j=0;j<100000000;j++){}
        }
    }
}
exit(0);
```

We are looping from -20 to 19, calling fork and modifying the priority using the nice values. Next, we print in exit() the process with its nice value which is finishing its execution.



(the image is cropped)

For this I have added yet another variable called nice in the proc structure in proc.h, just to store the nice value of each process. The expected output would be that processes with lower nice values (Hence lower priority number, and higher actual priority) finish their execution first, followed by the processes with next higher nice values. The output we get is shown in the above figure. Clearly, this is exactly what was expected. This tests our nice system call and our scheduler logic.

— THE END —

5. References

1. ***Linux Kernel Development*** *by Robert Love*
2. ***Professional Linux Architecture*** *by Wolfgang Maurer*
3. https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation.
4. https://en.wikipedia.org/wiki/O(1)_scheduler