MP-2

General changes:

1. To add a systemcall the changes are made as shown here:
   -
   https://stackoverflow.com/questions/8021774/how-do-i-add-a-system-call-utility-in-xv6

1. GOTTA COUNT 'EM ALL

- Modification to proc struct in proc.h :

```
int syscall_count[NUM_SYSCALLS];
# Each array element corresponds to a systemcall.
# Each array element initialised to 0 in allocproc
```

- Modify syscall.c to increment the corresponding array element every time a syscall is made.
- In waitx, add the count of each syscall of the child to the parent before reaping.
- sys_getSysCount accesses the array for the proc which called it, takes an int num argument, and returns syscall_count[num]
- syscount.c simply forks a child which runs the argument function and calls getSysCount which in turn calls sys_getSysCount

2. WAKE ME UP WHEN MY TIMER ENDS

- Modified proc structure in proc.h to include the following fields:
  - alarmticks - Time period after which handler is called

    - alarmleft - The number of ticks remaining before handler is called
    - is_handler_active - To prevent sigalarm logic from implementing when the handler is already active
    - alarm_tf - For context switching back to where we left off after handler is done
- Initialize them appropriately in allocproc() - allocate alarm_tf and deallocate in freeproc()
- In trap.c , in usertrap(), if which_dev equals 2 - it signifies a timer interrupt. In this case, check if alarmleft == 0 and if so, given handler is not already running, store the current trapframe in alarm_tf, context switch, set trapframe epc to handler function, and context switch.
- In sig_return() which is another systemcall which the handler is expected to call, context switch back(by changing p→trapframe to the old trapframe stored in alarm_tf), mark that the handler as no longer active and return to userspace.

General changes for both scheduler

PSEUDOTIME - IMPORTANT

- Pseudotime is used as an indicator of the order in which processes arrived.
- Note that the global variable “
- Pseudo time is a global variable which follows the invariant that
    - Every process which was scheduled when the value of pseudotime was lower was actually scheduled earlier.
- Basically, we cant just use ticks for getting start time, as multiple processes may arrive at the same time
- Pseudotime is initialsied to 0, and incremented:
    - Every time a new process is created

- Every time a sleeping process is woken up
- Every time a process changes queue (if MLFQ)

2. Changed makefile to include define LSB or MLFQ flags when user chooses to do so

LBS

- Implemented rand() using LRG
- added tickets and start_time field to proc.h
- Tickets initialized to 1 in allocproc() and child inherits the parent tickets in fork() function(overwrites the one ticket which was allocated by default)
- settickets systemcall simply changes the tickets field of current process to the num given as its argument(can be called by the user process)
- start_time is initialized to pseudotime in fork() as that is where any process becomes runnable
- In scheduler function:
- Probability proportional to number of tickets. This is implemented as :
  - Chose a random number from 0 to total_no_tickets
  - When the prefix sum exceeds the winning ticket, it implies the winning ticket is found in a region of size equal to the number of tickets held by the current process and since the ticket was randomly sampled, the probability that this happened is proportional to number of tickets as required.
  - Once winner is found, iterate again and find processes with the same number of tickets but a lower start_time()
- If chosen, perform a context switch

MLFQ

- The approach simulates a queue without actually implementing the actual data structure
- Added the following fields to proc structure:

```
    int priority;             // Priority of the process (cou
    int entry_time;           // Time at which the process en
    int ticks_used;           // ticks used since entryt ot t
```

- int queue_time_slice[NUM_QUEUES] = {1,4,8,12}; to store time slices for each queue
- IN SCHEDULER() function :
- Acquire ticks and first perform priority boost if it is a multiple of PRIORITY_BOOST_INTERVAL

```
// Helper functions for MLFQ
void priority_boost()
{
  struct proc* p;
  for ( p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->state == RUNNABLE)
    {
      p->priority = 0;
      p->ticks_used = 0;
      // Entry time i have decided to retain the original entry
    }
    release(&p->lock);
  }
}
// Set priority zero for all processes.
// reinitialise all ticks_used
```

```
// Letting them retain their entry_time
// adds some randomness
```

- Iterate through all process and schedule the process:
  - First which has the lowest queue no (called “priority” in my code)
  - Then, look for the process with lowest entry_time
- As explained earlier, entry_time is the pseudotime, so even if multiple processes come at the same tick, the one which came earlier actually has a lower entry_time
- Schedule the process
- In Usertrap() - called at the end of ever tick used by any process:
  - increase ticks used for the process

```
// Update the MLFQ Part
        p->ticks_used++;
        // Punish p if more than equal to tinme slice was used
        // and if not push it to the end of the same queue

        pseudotime++;

        //  Punish processes using more time by pushing them dow
        // priority
        if (p->ticks_used >= queue_time_slice[p->priority])
        {

          if (p->priority < NUM_QUEUES - 1)
          {
            p->priority++;
          }
          p->ticks_used = 0;

          ////////////// IMPORTANT ///////////////
```

```
          // We want this process to have the highest time with
          // because it is at the back or end of the queue (choi
          // within the same queue is based on entry_time - this
          // incremented pseudotime and hence the highest )

          p->entry_time = pseudotime;

        }
```

- If we have a I/O interrupt ：

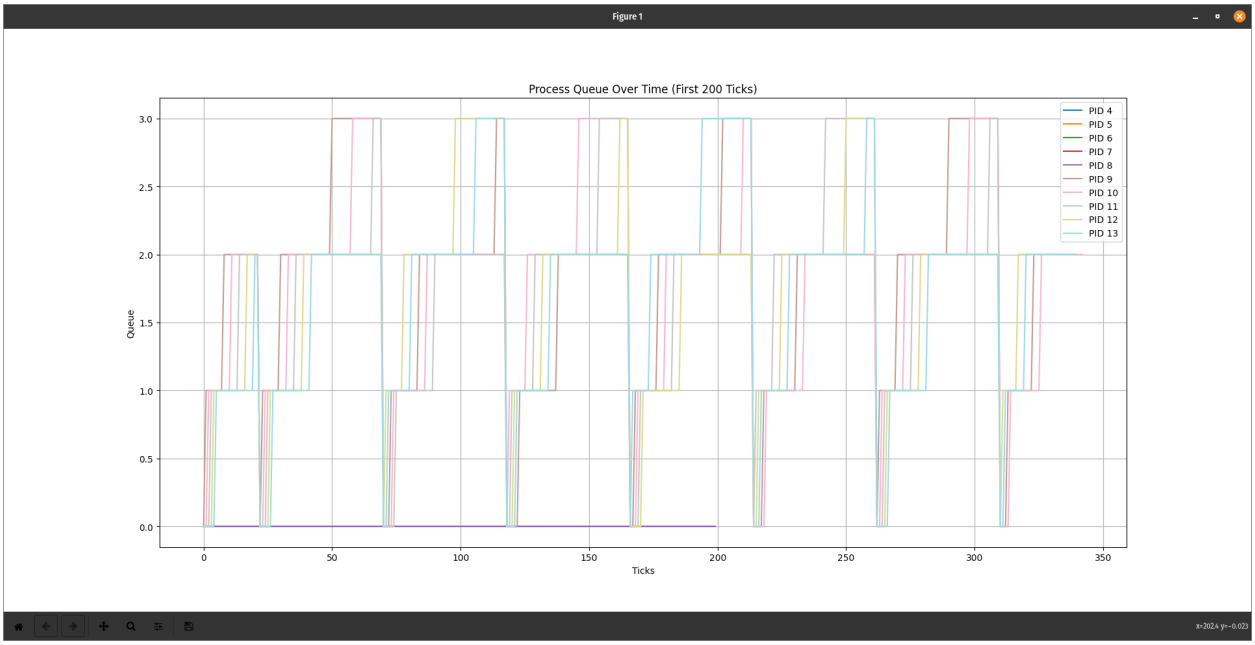
Notice how I do not set ticks used to zero here - I just push it to the end of the queue.

This is done to prevent gaming of the scheduler by relinquishing CPU just before time slice.

```
else if (which_dev == 1 || which_dev==3 || which_dev ==4)
 {
     // This doesnt change the queue number, just pushes it to
     // by increasing entry_time
     // ticks_used is not changed to prevent gaming of schedule
   pseudotime++;
   p->entry_time = pseudotime;

 }
```

MLFQ GRAPH:



TIME COMPARISON

- Columns a:b signify NFORK-a, I/O-b
- Each cell - a,b : Avg rtime: a , Avg wtime: b
- ***NOTE: These are values in the default schedulertest where I/O bound processes arrive first.***

	RR	LBS (with arrival time)	LBS without arrival time	MLFQ
10 : 5	27, 208	26, 157	26, 194	27 , 206
20 : 10	28, 350	27, 227	27, 319	28, 340
20 : 5	42, 636	41, 343	42, 543	41, 623

Kindly see the IMPORTANT- ARRIVAL TIME section for an explanation of why LBS with arrival time is performing better in this particular case.

QUESTION : Implications of arrival time on LBS

- With arrival times, the algorithm is essentially a FCFS among all processes which have the same number of tickets (and if that number is the winning ticket)

> The **convoy effect** refers to a situation where shorter processes get stuck waiting for a long-running process to complete, leading to poor performance.

- PITFALLS: The convoy effect (which was the primary issue with FCFS) still exists in a way. Consider the case where all processes have the same number of tickets, and the process which arrived earlier runs for a very long time. It hogs up the resources until it runs and causes other processes with the same number of tickets but which arrived later to starve.
- ***Even worse, say all processes have the same number of tickets and the first process to arrive never returns control (say by having an infinite loop). Then the scheduler is just stuck in an infinite loop - choosing the first process to arrive which never completes. The only solution is to reboot. This in fact happens with the "preempt" test of usertests which the current implementation of LBS scheduler fails. Simply removing the arrival time part from code passes all tests.***
- It is better than FCFS *if the number of tickets is different for different processes since* :
    - lottery scheduling still selects processes **randomly based on ticket distribution**. Thus, no single process (even an early one) can monopolize the CPU entirely unless it has many tickets. This ensures that all processes get a chance, preventing the convoy effect from being too severe.
- Another major pitfall of LBS in general is giving the power of setting tickets to the user application. This is essentially an open invitation to monopolize the scheduler

IMPORTANT - ARRIVAL TIME :

- Now, the above graph shows significantly lesser wait time for LBS with arrival time. However, that is because the default schedulertest code in XV6 spawns I/O bound processes first. I/O bound processes relinquish CPU early and hence finish earlier. In short shorter process arrive first. FCFS is the best algorithm in existence for such a case (it can be proven that scheduling shorter processes first is always beneficial). However, this need not always be the case as and we do not have apriori knowledge of process running times.

→ Schedulertest code:

```
#define NFORK 10
#define IO 5

int main()
{
  int n, pid;
  int wtime, rtime;
  int twtime = 0, trtime = 0;
  for (n = 0; n < NFORK; n++)
  {
    pid = fork();
    if (pid < 0)
      break;
    if (pid == 0)
    {
    ////////////////// VERY IMPORTANT ////////////////////////
    ////////////////////////////////////////////////////////
      if (n > IO)   // nOTICE THE N>I/O PART
      {
        sleep(200); // IO bound processes
      }
      else
      {
        for (volatile int i = 0; i < 1000000000; i++)
        {
```

```
        } // CPU bound process
      }
      printf("Process %d finished\n", n);
      exit(0);
    }
  }
  for (; n > 0; n--)
  {
    if (waitx(0, &wtime, &rtime) >= 0)
    {
      trtime += rtime;
      twtime += wtime;
    }
  }
  printf("Average rtime %d,  wtime %d\n", trtime / NFORK, twtime
  exit(0);
}
```

→ Simply changing the N<I/O to N>1/O will spawn CPU intensive (the useless ‘for’ loop part) processes first i.,e longer processes are scheduled first. This is the perfect setting for convoy effect and indeed testing the schedulertest with this set up for NFORK=10 and I/O = 5 gives, rtime = 27, wtime = 263 (far worse than the case where no arrival time was taken into consideration).

- In general, whether arrival time should be considered or not depends on the order in which processes arrived. It is beneficial to consider if shorter processes arrive first.
- We could implement a setup where number of ticks used is counted for each process and if it crosses a threshold, arrival time is not considered otherwise it is.