# xv6 OS Scheduling Algorithm Assignment with Statistics Tracking

This README provides a detailed guide on how to modify the xv6 operating system to implement two scheduling algorithms, namely First-Come First-Serve (FCFS) and Priority-Based Scheduling. Additionally, we will create a new program test.c that runs these scheduling scenarios and tracks relevant process statistics.

## Table of Contents

1. [Introduction](#introduction)
2. [System Call and Utility Overview](#overview)
3. [Modification Steps](#modification-steps)
   1. [Update proc.h](#update-proch)
   2. [Update proc.c](#update-procc)
   3. [Update Makefile](#update-makefile)
   4. [Create head.c](#create-headc)
   5. [Create uniq.c](#create-uniqc)
   6. [Create test.c](#create-testc)
   7. [Create pstat.h](#create-pstat)
4. [Building and Running](#building-and-running)
5. [Conclusion](#conclusion)

## 1. Introduction

This assignment focuses on extending the xv6 operating system by implementing two scheduling algorithms, FCFS and Priority-Based Scheduling. Additionally, we will track and report process statistics including waiting time, turnaround time, and others. The goal is to test these scheduling algorithms in different scenarios and report the performance statistics.

## 2. Assignment Description

* **System calls**:
  + customfork: sets up the priority for each process.
  + procstat: calculates turnaround and waiting time of a process.
* **Utilities**:
  + head: Displays the first N lines of a file.
  + uniq: Filters repeated lines from a file, displaying only unique lines.

### Test Scenarios:

* **FCFS Scenarios**: When user processes arrive before the kernel process. When the kernel process arrives before user processes.
* **Priority Scenarios**: When the User process has the highest priority. When the kernel processes have the highest priority.

### Statistics Tracking:

For each scenario, the assignment requires tracking and reporting the following process statistics:

* **FCFS**: Creation time (ctime) End time (etime) Total time (ttime) Average Wait time (wtime) Average Turnaround time (tatime)
* **PBS**: Creation time (ctime) End time (etime) Total time (ttime) Priority of the process (priority) Average Wait time (wtime) Average Turnaround time (tatime)

To achieve this, we will modify few files and implement a system call to calculate average wait and turnaround times.

## 3. Modification Steps

### 3.1. Update proc.h

Edit the proc.h file and add the following lines:

// New fields for extended proc struct  
int tatime;  
int priority; // Turnaround time for the process

### 3.2. Update proc.c

In the proc.c file, add the following lines as instructed:

* **For FCFS**

void  
scheduler(void)  
{  
 //struct proc \*p;  
 struct cpu \*c = mycpu();  
 c->proc = 0;  
  
 for(;;){  
 // Enable interrupts on this processor.  
 sti();  
  
 // Loop over process table looking for process to run.  
 acquire(&ptable.lock);  
  
 //Assignment3 FCFS  
 struct proc \*minP = 0, \*p = 0;  
 // Loop over process table looking for process to run.  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
 if(p->state != RUNNABLE)  
 continue;  
  
 // ignore init and sh processes from FCFS  
 if(minP != 0){  
 // here I find the process with the lowest creation time (the first one that was created)  
 if(p->ctime < minP->ctime)  
 minP = p;  
 }  
 else  
 minP = p;  
 }  
  
 if(minP != 0){  
 // Switch to chosen process. It is the process's job  
 // to release ptable.lock and then reacquire it  
 // before jumping back to us.  
 c->proc = minP;  
 switchuvm(minP);  
 minP->state = RUNNING;  
 // cprintf("cpu %d, pname %s, pid %d, rtime %d\n", c->apicid, minP->name, minP->pid, minP->rtime);  
 swtch(&(c->scheduler), minP->context);  
 switchkvm();  
  
 // Process is done running for now.  
 // It should have changed its p->state before coming back.  
 c->proc = 0;  
 }  
 //}  
 release(&ptable.lock);  
  
 }  
}

* **For PBS:**

void scheduler(void)  
{  
 struct proc \*p, \*p1;  
  
 struct cpu \*c = mycpu();  
 c->proc = 0;  
  
 for (;;)  
 {  
 // Enable interrupts on this processor.  
 sti();  
  
 struct proc \*highP = 0;  
 // Loop over process table looking for process to run.  
 acquire(&ptable.lock);  
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
 {  
  
 if (p->state != RUNNABLE)  
 continue;  
 // Choose the process with highest priority (among RUNNABLEs)  
 highP = p;  
 for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)  
 {  
 if ((p1->state == RUNNABLE) && (highP->priority > p1->priority))  
 highP = p1;  
 }  
  
 if (highP != 0)  
 p = highP;  
  
 if (p != 0)  
 {  
  
 // Switch to chosen process. It is the process's job  
 // to release ptable.lock and then reacquire it  
 // before jumping back to us.  
 c->proc = p;  
 switchuvm(p);  
 p->state = RUNNING;  
  
 swtch(&(c->scheduler), p->context);  
 switchkvm();  
  
 // Process is done running for now.  
 // It should have changed its p->state before coming back.  
 c->proc = 0;  
 }  
 }  
 release(&ptable.lock);  
 }  
}

### System calls

* **FCFS :**

// Calculate the average wait time and average turn around time of all processes  
int  
procstat(int processid, struct pstat \*pstat)  
{  
 struct proc \*p;  
 int havechild;  
 struct proc \*curproc = myproc();  
 acquire(&ptable.lock);  
 for (;;)  
 {  
 havechild = 0;  
  
 // Loop through all processes in the process table  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
 {  
 if(p->parent != curproc)  
 continue;  
 havechild = 1;  
 // Check if the current process matches the specified process ID  
 if (p->pid == processid && p->state == ZOMBIE)  
 {  
  
 if (pstat != 0)  
 {  
 pstat->ctime = p->ctime;  
 pstat->etime = p->etime;  
 pstat->ttime = p->etime - p->ctime;  
 //Assignment3  
  
 pstat->tatime = pstat->ttime;  
 }  
 kfree(p->kstack);  
 p->kstack = 0;  
 freevm(p->pgdir);  
 p->pid = 0;  
 p->parent = 0;  
 p->name[0] = 0;  
 p->killed = 0;  
 p->state = UNUSED;  
 release(&ptable.lock);  
 return processid;  
 }  
  
 }  
 curproc->etime = ticks;  
  
 // No point waiting if we don't have any children.  
 if(!havechild || curproc->killed)  
 {  
 release(&ptable.lock);  
 return -1;  
 }  
  
 sleep(curproc, &ptable.lock);  
 }  
}

* **PBS :**

### In test.c

sum\_of\_wtime += wtime;  
 sum\_of\_tatime += pstat\_info.tatime;  
 wtime += pstat\_info.tatime;

### 3.3. Update Makefile

Edit the Makefile file and add the following lines in the user section:

\_ps\  
\_head\  
\_uniq\  
\_test\

### 3.4. Create head.c

Create a new file named head.c and add the following code:

#include "types.h"  
#include "stat.h"  
#include "user.h"  
  
#define MAX\_LINE\_LENGTH 1024  
#define DEFAULT\_N\_LINES 10  
  
void head(int fd, int n) {  
 char line[MAX\_LINE\_LENGTH];  
 int line\_count = 0;  
  
 while (1) {  
 int bytesRead = read(fd, line, sizeof(line));  
 if (bytesRead <= 0) {  
 break;  
 }  
  
 for (int i = 0; i < bytesRead; i++) {  
 if (line[i] == '\n') {  
 line\_count++;  
 if (line\_count > n) {  
 break;  
 }  
 }  
  
 printf(1, "%c", line[i]);  
  
 if (line\_count >= n) {  
 break;  
 }  
 }  
  
 if (line\_count >= n) {  
 break;  
 }  
 }  
}  
  
int main(int argc, char \*argv[]) {  
 int n = DEFAULT\_N\_LINES;  
 int fd = 0; // Initialize to standard input (0)  
  
 if (argc > 1 && argv[1][0] == '-') {  
 // Parse the number of lines from the command-line argument  
 n = atoi(argv[1] + 1);  
  
 if (n <= 0) {  
 printf(2, "Usage: head [-N] [file]\n");  
 exit();  
 }  
  
 // Open the file if provided  
 if (argc > 2) {  
 fd = open(argv[2], 0);  
 }  
 } else {  
 // No option provided, use default number of lines  
 if (argc > 1) {  
 fd = open(argv[1], 0);  
 }  
 }  
  
 if (fd < 0) {  
 printf(2, "head: cannot open '%s'\n", argv[argc - 1]);  
 exit();  
 }  
  
 head(fd, n);  
 close(fd);  
 exit();  
}

### 3.5. Create uniq.c

Create a new file named uniq.c and add the following code:

#include "types.h"  
#include "stat.h"  
#include "user.h"  
#include "fcntl.h"  
  
#define MAX\_LINE\_LENGTH 1024  
  
void uniq(int input\_fd) {  
 char line[MAX\_LINE\_LENGTH];  
 char prev\_line[MAX\_LINE\_LENGTH] = ""; // Store the previous line  
  
 while (1) {  
 int n = read(input\_fd, line, sizeof(line));  
  
 if (n <= 0) {  
 break; // End of file or an error  
 }  
  
 line[n] = '\0'; // Null-terminate the line  
  
 // If the current line is different from the previous line, print it  
 if (strcmp(line, prev\_line) != 0) {  
 printf(1, "%s", line);  
 strcpy(prev\_line, line); // Update the previous line  
 }  
 }  
}  
  
int main(int argc, char \*argv[]) {  
 int input\_fd = 0; // Default to standard input (file descriptor 0)  
  
 if (argc > 1) {  
 // Open the file if provided  
 input\_fd = open(argv[1], O\_RDONLY);  
  
 if (input\_fd < 0) {  
 printf(2, "uniq: cannot open %s\n", argv[1]);  
 exit();  
 }  
 }  
  
 uniq(input\_fd);  
  
 if (input\_fd != 0) {  
 close(input\_fd);  
 }  
  
 exit();  
}

### 3.6. Create test.c

Create a new file named test.c and add the following code:

#include "types.h"  
#include "stat.h"  
#include "user.h"  
#include "fcntl.h"  
  
struct pstat {  
 int ctime;  
 int etime;  
 int ttime;  
 int tatime;  
 int priority;  
};  
  
int main() {  
 char \*commands[] = {"uniq", "head"};  
 char \*arguments[] = {"input.txt", "example.txt"};  
 int prior[] = {2,1};  
 int num\_commands = sizeof(commands) / sizeof(commands[0]);  
 int wtime = 0, sum\_of\_wtime = 0, sum\_of\_tatime = 0;  
  
 for (int i = 0; i < num\_commands; i++)  
 {  
 int cpid;  
 struct pstat pstat\_info;  
  
 // creating a child process  
 cpid = customfork(prior[i]);  
 if (cpid < 0)  
 c {  
 printf(1, "fork failed to create\n");  
 exit();  
 }  
 if (cpid == 0)  
 {  
 // This is the child process  
 char \*args[] = {commands[i], arguments[i], 0};  
 exec(args[0], args);  
 printf(1, "exec %s failed for the process\n", commands[i]);  
 exit();  
 }  
 else  
 {  
 if (procstat(cpid, &pstat\_info) < 0)  
 {  
 printf(1, "procstat failed\n");  
 exit();  
 }  
 }  
  
 //printing the statistics of the processes  
 printf(1, "\nProcess statistics for '%s %s':\n", commands[i], arguments[i]);  
 printf(1, " Creation time: %d\n", pstat\_info.ctime);  
 printf(1, " End time: %d\n", pstat\_info.etime);  
 printf(1, " Total time: %d\n\n", pstat\_info.ttime);  
 printf(1," Priority of this process is: %d\n",pstat\_info.priority);  
 sum\_of\_wtime += wtime;  
 sum\_of\_tatime += pstat\_info.tatime;  
 wtime += pstat\_info.tatime;  
 }  
  
 printf(1, " Average Turnaround time using FCFS: %d\n", (sum\_of\_tatime/num\_commands));  
 printf(1, " Average Wating time using FCFS: %d\n\n", (sum\_of\_wtime/num\_commands));  
 exit();  
}

### 3.7. Create pstat.h

Create a new file named pstat.h and add the following code:

struct pstat {  
 int ctime;  
 int etime;  
 int ttime;  
 int tatime;  
 int priority;  
};

## 4. Building and Running

1. Build xv6 with the new modifications. Follow the build instructions provided in the xv6 documentation or repository.
2. After building, make sure to include the newly created utility programs (head, uniq, test) in the xv6.img file using the mkfs command.
3. Boot xv6 in a virtual machine or emulator (e.g., QEMU).
4. You can now run the new utility programs from the xv6 shell. For example:
   * test will execute the test program and will report all the scheduler statistics.

## 5. Conclusion

You have successfully modified the xv6 operating system to provide all the scheduler statistics of the processes including FCFS and PBS.