

Ain Shams University Faculty of Engineering

Department of Computer Engineering and Software Systems

Operating Systems CSE_353

Modifying XV6 OS

Ahmed Salah Eldin Mohamed A.Maksoud	18P9076
Ahmed Emad Anwer Elsyed Mahmoud	18P3597
Omar Khaled Mahmoud Mohamed Mohamed Masoud	18P3067
Kareem Ayman Farouk	18P6994
Emad Mostafa Mohamed	18P1003
Kareem Amr Abdelsamie Abdelrahman	18P9093
Yehia Mohamed Hesham Mohamed	1804493

Modifying XV6 OS

Operating System CSE_353
Faculty of Engineering, Ain Shams University

Table of Contents

II.	Introduction.....	4
A.	Purpose	4
III.	R1 — CPU Scheduling Algorithms	5
A.	Briefing.....	5
B.	Round Robin (RR).....	5
1)	Algorithm	5
2)	XV6 Implementation	5
3)	Performance.....	6
C.	Shortest Remaining Job First (SJF)	7
1)	Theoretical background	7
2)	Implementation.....	7
3)	Performance.....	9
D.	Priority based.	10
1)	Theoretical background	10
2)	Implementation.....	10
3)	Performance.....	12
E.	Multi-Level Feedback Queue	13
1)	Theoretical background	13
2)	Implementation.....	14
3)	Performance.....	16
F.	Comparative Analysis.....	Error! Bookmark not defined.
IV.	R2 — Memory Management	19
A.	Briefing.....	19
1)	Memory accessing:	19
2)	Pages and page table flags:	19
B.	Page Replacement.....	20
1)	Supplied File Framework	20
2)	Memory Constrains for Process	21
3)	Retrieving a page.....	21
C.	Page Replacement Algorithms.....	21
1)	First in First Out Algorithm (FIFO)	21
2)	LRU	23
3)	FIFO & LRU Performance test	25
V.	R3 — File Management.....	26
A.	Briefing.....	26
B.	Using User-defined Extents	26

1) Implementation:.....	26
2) Performance.....	29
C.....	30
D. Analysis	31

I. INTRODUCTION

A. *Purpose*

The document has been created as a project report for the CSE335 Operating Systems Course, the project covers all of the implementational details needed for the project accompanied with a theoretical background explanation, a comparative analysis, and a code walkthrough.

This document is specifically submitted to Prof. Gamal Abdelshafy, and Eng. Sally Shaker as they are the intended readers of this report. The intended audience of this report is any student or engineer that is editing the XV6 operating system and require guidance to implement the algorithms which are aforementioned in the table of contents.

The flow of the document will follow the flow of each requirement as specified in the project statement. The submission of this report is accompanied with the code of each requirement submitted externally to the LMS.

II. R1 — CPU SCHEDULING ALGORITHMS

A. Briefing

CPU scheduling algorithms are short-term schedulers (STS), used to allocate the CPU to a process queued in the ready queue. What process will be allocated the CPU next is depending on the CPU scheduling algorithm. The STS is invoked very rapidly, often once every 100 milliseconds, and no operating system can function without implementing a STS of some sort.

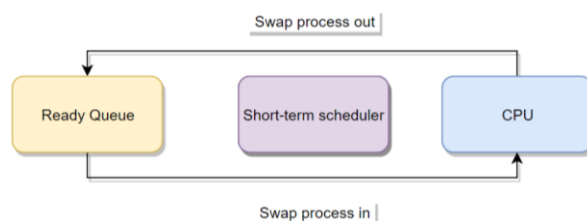


Figure II.1 CPU scheduler

Depending on the scheduling algorithm and the type of system, performance can be drastically affected. Each algorithm affects the process time parameters (such as turnaround time, waiting time, response time...) which in return affects how the system will perform.

B. Round Robin (RR)

1) Algorithm

The main concept of this algorithm is that all processes have the same maximum amount of time in the CPU called quantum. If the process takes less time than the quantum, the process is not preempted, otherwise, the process is preempted back to the ready queue to be scheduled again later. Turnaround time is hurt when using RR; however, the response time is minimized compared to other algorithms.

2) XV6 Implementation

The RR code implementation can be found in `[proc.c → Line 323 → void scheduler(void)]`. The scheduler loop is finding a process in the ready state from the process table, run the process until it finishes or exceeds time quantum, repeat.

First we enable interrupts via the `"sti()"` function, this is done to preempt an idle CPU in case no process was in the ready state. Without interrupts the CPU will keep looking for a process to run and never releasing the lock, thus the process table cannot be accessed, and no context switch can take place.

Next we acquire the lock for the process table so that no 2 CPUs run the same ready process, and now we can start looking for a runnable process. The for loop starts at the first process in the process table, and loops until the last process in the process table. Inside the for loop we check each process if it is runnable or not, if no process is runnable we exit the for loop and repeat the process again (the second for loop is within another infinite for loop).

However, if a process is found to be in the runnable state, we allocate the CPU to the process, call `"switchvm(p)"` to signal the hardware to use process's page table, change the process state to RUNNING, call `"swtch"` to perform a context switch. Now that the process is done running, we continue executing the rest of the `"scheduler()"` code, and we call the `"switchkvm()"` function to switch back to the kernel's page table. Then we de-allocate the process from the CPU, and after this is done for all of the page table we release the lock of the page table.

```
322 void
323 scheduler(void)
324 {
325     struct proc *p;
326     struct cpu *c = mycpu();
327     c->proc = 0;
328
329     for(;;){
330         // Enable interrupts on this processor.
331         sti();
332
333         // Loop over process table looking for process to run.
334         acquire(&table.lock);
335         for(p = table.proc; p < &table.proc[NPROC]; p++){
336             if(p->state != RUNNABLE)
337                 continue;
338
339             // Switch to chosen process. It is the process's job
340             // to release ptable.lock and then reacquire it
341             // before jumping back to us.
342             c->proc = p;
343             switchvm(p);
344             p->state = RUNNING;
345
346             swtch(&c->scheduler, p->context);
347             switchkvm();
348
349             // Process is done running for now.
350             // It should have changed its p->state before coming back.
351             c->proc = 0;
352         }
353         release(&table.lock);
354     }
355 }
356 }
```

Figure II.2 RR code

3) Performance

We measure performance using 3 main metric

1. Turnaround time (TAT)
 - a. Measured starting from the arrival time until it terminates
2. Waiting time (WT)
 - a. Measured as the time the process spends outside of the CPU
 - b. $WT = TAT - \text{running time}$
3. Response time (RT)
 - a. Measured as the time from arrival time until the process enters the CPU for the first time.
 - b. First waiting period.

a) Example

Here is an example of the round robin scheduling algorithm.

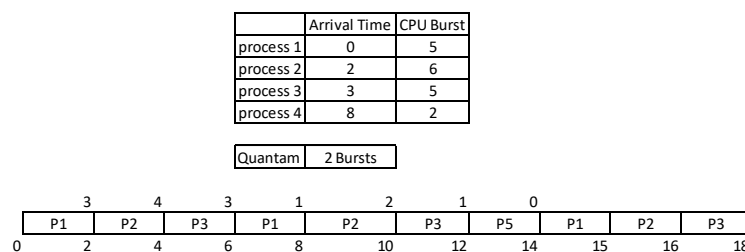


Figure II.3 RR example

$$\begin{aligned} \text{Average TAT} &= \frac{(15 - 0) + (16 - 2) + (18 - 3) + (14 - 8)}{4} = 12.5 \text{ time units} \\ \text{Average WT} &= \frac{(15 - 5) + (14 - 6) + (15 - 5) + (6 - 2)}{4} = 5.5 \text{ time units} \\ \text{Average RT} &= \frac{(0) + (2 - 2) + (4 - 3) + (14 - 8)}{4} = 1.75 \text{ time units} \end{aligned}$$

b) Measurement

When measuring performance for the round robin algorithm on the XV6 for the “ls” command. We can see the average of the children created by this command.

```
#####
Process Terminated
PID: 3
TAT: 25
WT: 8
```

Figure II.4 LS command performance using RR

The we know that the RT is always less than or equal to the WT; therefore, comparing the average TAT and to the average RT we can see that the TAT is 3 times larger than the RT. Which is expected as the RR produced a large TAT but a small RT.

This is because the process has a fixed amount of time to run in (quantum), which is usually smaller than the number of CPU bursts needed by the process. As we are always switching between processes rabidly, a single process will not wait for a long time before it gets the CPU again, which explains the small RT. However, this also means that a process will not have the CPU for a long time enough for it to finish most of its needed bursts, thus the process will have to stay queued longer increasing the termination time and thus the TAT.

C. Shortest Remaining Job First (SJF)

1) Theoretical background

Shortest remaining time first, is considered to be the preemptive version of the shortest job first algorithm, in this algorithm the job that has the minimum CPU needed cycles, will have the CPU allocated to it, while keeping track of the other processes required time, when another job CPU cycle count gets smaller than the current process allocated to the CPU, the operating system will perform context switching between those two processes so that the CPU is allocated to the shortest job right now, and after each cycle, this processes is repeated so that the CPU will always be allocated to shortest job, however this process requires more overhead time than the traditional shortest job first because the OS is required to monitor the CPU time of all the jobs in the ready queue and performance of context switching between processes also adds to the overhead time, but on the other hand the shortest remaining job first make the overall processing faster than the traditional shortest job algorithm.

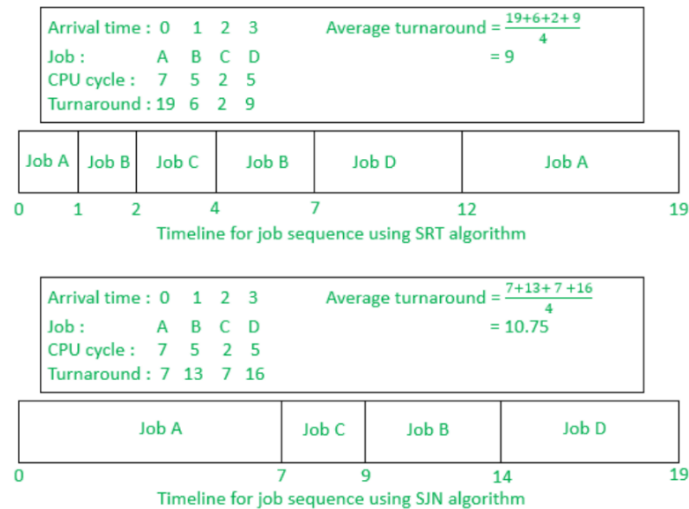


Figure II.5 SJN VS SRT

2) Implementation

In this section we will explain our implementation of the shortest remaining job first algorithm, first we made two pointers that points to the processes and a pointer to the CPU as shown below.

```

322 void
323 scheduler(void)
324 {
325     struct proc *p,*p1;
326     struct cpu *c = mycpu();
327     c->proc = 0;
328
329     for(;;){
330         // Enable interrupts on this processor.
331         sti();
332     }

```

Figure II.6 SRT part 1

*p will be used to find any runnable process in the process table and *c is used to point to the CPU, and it is process is first initialized by null, then for (; ;) is used to make an infinite loop, where sti() is enable interruption for the process, thus our shortest remaining job first scheduler deals with preemptive processes.

```

329     for(;;){
330         // Enable interrupts on this processor.
331         sti();
332
333         // Loop over process table looking for process to run.
334         acquire(&ptable.lock);
335         struct proc *shortestJob = 0;
336         uint maxSize = __UINT32_MAX__;
337         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
338             if(p->state != RUNNABLE)
339                 continue;
340
341             if(p->sz < maxSize)
342             {
343                 shortestJob = p;
344             }

```

Figure II.7 SRT part 2

After entering the infinite loop and enabling interruption for the processes, now we should acquire the lock for the table so that no other CPU can modify in the same table of processes.

The *shortestJob pointer initialized to null that will be used to point to the process having the least time later in the code, afterwards we initialize an unsigned variable to the max size of the process, and the reason will be explained later in the code.

On line 337 we loop on the process table, this loop is made to find a runnable a process within the table, which we were able to find by using the if condition in line 346, in case if the process is not runnable, we will continue to the next iteration, until we meet a runnable process then the loop will exit, and we will have pointer p pointing to this runnable process.

Then the shortestJob pointer will currently point to p since it is the smallest job we currently have, but it will be modified later.

```
341     if(p->sz < maxSize)
342     {
343         shortestJob = p;
344     }
345     for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
346         if(p1->state != RUNNABLE)
347             continue;
348         if(shortestJob->sz > p1->sz)
349         {
350             shortestJob = p1;
351         }
352     }
```

Figure II.8 SRT part 3

After line 344, now have *p and *shortestJob pointing to the same process, now the next loop will be used to determine the process that has the shortest CPU time.

On line 345, we have another loop, looping on the process table, and on like 346 we are trying to find runnable processes within the table.

In case of runnable processes they will pass the first condition and jump to the next condition on line 348, in this condition we are comparing the processes size with each other.

But before we proceed why are we comparing the sizes? In our implementation we assumed that the shortest job would have the smallest size, this is our method for predicting the time that the job will need.

On line 348, we are comparing the current shortest job size with the other runnable processes' size within the table, by the end of this loop we should have the shortestJob pointing to the process requiring the least time within the table.

```
353     // Switch to chosen process. It is the process's job
354     // to release ptable.lock and then reacquire it
355     // before jumping back to us.
356     cprintf("shortestJob %d \n", shortestJob->sz);
357     p = shortestJob;
358     c->proc = p;
359     switchvm(p);
360     p->state = RUNNING;
361
362     swtch(&(c->scheduler), p->context);
363     switchvm();
364
365     // Process is done running for now.
366     // It should have changed its p->state before coming back.
367     c->proc = 0;
368 }
369 release(&ptable.lock);
370 }
371 }
```

Figure II.9 SRT part 4

Line 356, printing the shortest job size is used for testing purposes, after that we will have pointer p, pointing to the shortest job, and switching will be performed in order to run the process, after a quantum time, the process will switch back, and the CPU is assigned back to null and the whole process will repeat, after ptable loop finishes, the CPU will release its lock.

3) Performance

As we mentioned before the shortest time remaining scheduler has more overhead time than the basic shortest time scheduler, this is due to the context switching, however it is generally better, in terms when looking at the overall processing time.

When comparing our implementation to the how the shortest remaining time scheduler should perform, we can see that our implementation has more overhead time than the usual this due to switching over a constant quantum of time rather than monitoring the processes CPU time, but since we only have two CPUs running in our system, monitoring all the processes time will not make much a difference.

Our implementation for the shortest remaining time scheduler might also seem close to round robin implementation, however we increased the quantum time for the SRT algorithm, so that we lower our overhead time and to make it as effective as possible. In the figure below, we can see that our SRT algorithm is functioning properly since the jobs of smaller size are being printed first and after they finish their execution, the CPU is allocated to the next bigger one.

```
#####  
Process Terminated  
PID: 3  
TAT: 50  
WT: 15  
PRIORITY BEFORE EXIT:17  
#####
```

Figure II.10 SRT testing wt/tat

[illegible]

Figure II.11 SRT testing

D. Priority based.

1) Theoretical background

The main characteristics of the priority scheduling algorithm is that the processes are assigned different priorities, a default high priority for kernel level processes, and a low priority in the user level processes. There are two types of priority-based algorithm, preemptive, and non-preemptive.

The preemptive priority based is works in a way such that whenever a new process enters the system, it interrupts the currently executing process, and attempts to take its place according to its priority (if the new process has a higher process that the existing one, then a context switch will occur, and the new process will acquire the CPU). If two jobs have the same priority are in the ready state, they will execute on FIRST COME FIRST SERVE (FCFS) fashion.

The non-preemptive priority-based works in a way such that whenever a new process enters the system, no interrupts occur and hence, the existing process will remain running on the CPU until it voluntary leaves the CPU even if the newly arrived process has a higher priority than the existing one.

There are two implementation types of this algorithm, the top-bottom priority approach, in which the high number represents low priority; and the bottom-up approach in which the high number represents high priority. In our proposed implementation, we have adopted the preemptive bottom-up approach.

Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

Figure II.12 example of priority scheduling

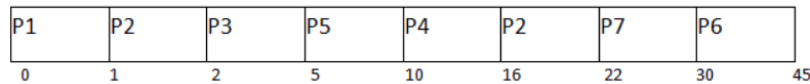


Figure II.13 Priority method Gantt chart

2) Implementation

As aforementioned, the implementation approach follows a preemptive bottom-up approach in the range 0 to 20. The implementation guide will follow a logical order.

First, we start with the priority assigned to the process when it arrives. We will implement a default priority for the user level processes of 10, and a default priority for the kernel level processes of 17.

The following files needed to be modified to implement the aforementioned setup:

- exec.c added a default priority for kernel level processes at line 102
- proc.c added a default priority for the user level processes at line 93

```
99  curproc->sz = sz;  
100  curproc->tf->eip = elf.entry; // main  
101  curproc->tf->esp = sp;  
102  curproc->priority = 17; //assign high priority for a process loaded from shell  
103  switchvm(curproc);
```

Figure II.14 exec.c modification

```

90     p->pid = nextpid++;
91     /* Code start */
92     /* for the priority sched */
93     p->priority = 10; // Default priority of 10
94     p->starttime = ticks;
95     p->rtime = 0;
96     p->endtime = 0;

```

Figure II.15 proc.c modification

Then, we have implemented the priority-based method itself. Our idea was to loop over the process table for a second time to fetch the process with the highest priority. After the process was found, a context switch happens that switches the CPU to the current process to run and save the context of the existing (running) process. We have used an aging technique to increase the priority of the process as long as it stays in the ready (RUNNABLE) state. The following code shows how the priority-based scheduling with aging have been implemented. The comments show the function of each implemented line.

```

C proc.c > scheduler(void)
345 void
346 scheduler(void)
347 {
348     struct proc *p, *p1;
349     struct cpu *c = mycpu();
350     c->proc = 0;
351
352     for(;;){
353         // Enable interrupts on this processor.
354         sti();
355         struct proc *high_priority_process = 0; // points to a high priority process
356         // Loop over process table looking for process to run.
357         acquire(&ptable.lock);
358         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
359             if(p->state != RUNNABLE)
360                 continue;
361
362             high_priority_process = p; // put p as a high prioity process
363
364             for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
365                 if(p1->state != RUNNABLE)
366                     continue;
367                 if (high_priority_process->priority < p1->priority) //found a process with higher priority (lower value)
368                     high_priority_process = p1;
369             }
370
371             // Switch to chosen process. It is the process's job
372             // to release ptable.lock and then reacquire it
373             // before jumping back to us.
374             p = high_priority_process;
375             cprintf("\n<Process: %d Priority: %d Size %d\n", p->pid, p->priority, p->sz);
376             c->proc = p;
377             switchvm(p);
378             p->state = RUNNING;
379
380             swtch(&c->scheduler, p->context); //perform context switch between the process in the schedule, and the process context
381             switchvm();
382             if(p->priority < 20) p->priority++; //when switch to RUNNABLE increment the priority
383             // Process is done running for now.
384             // It should have changed its p->state before coming back.
385             c->proc = 0;
386
387             release(&ptable.lock);
388         }
389     }
390 }

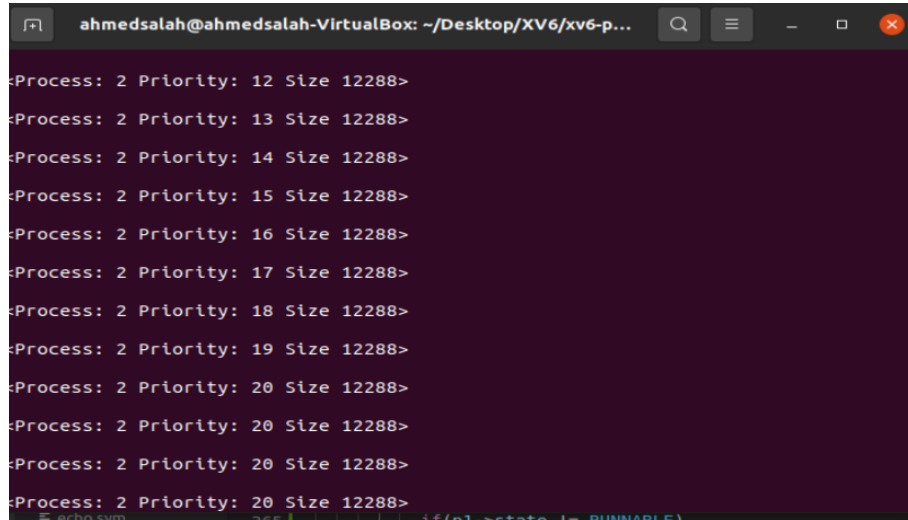
```

Figure II.16 code of priority-based scheduling

The basic idea is that we use the pointer **p1** to loop over the table once again and compare if the new fetched process priority (**p1->priority**) is greater than the process that reside in **high_priority_process**, if that is true, then **high_priority_process** will hold p1, and then **switch** will perform context switch to switch the CPU to the process with the highest priority.

3) Performance

We have mentioned before that the preemptive priority-based algorithm assigns the process with the highest priority to the CPU until a new process of higher priority enters the ready queue. The results of the priority-based algorithm that we have implemented in XV6 operating system illustrates the aforementioned behavior.

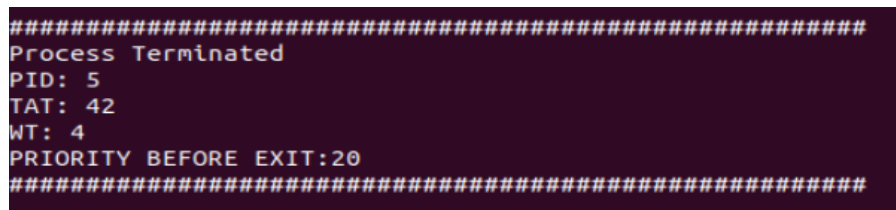


```
ahmedsalah@ahmedsalah-VirtualBox: ~/Desktop/XV6/xv6-p...
Process: 2 Priority: 12 Size 12288>
Process: 2 Priority: 13 Size 12288>
Process: 2 Priority: 14 Size 12288>
Process: 2 Priority: 15 Size 12288>
Process: 2 Priority: 16 Size 12288>
Process: 2 Priority: 17 Size 12288>
Process: 2 Priority: 18 Size 12288>
Process: 2 Priority: 19 Size 12288>
Process: 2 Priority: 20 Size 12288>
Process: 2 Priority: 20 Size 12288>
Process: 2 Priority: 20 Size 12288>
Process: 2 Priority: 20 Size 12288>
```

Figure II.17 shows the priority change of process 2

Note that process 2 has been switching between the RUNNING and the RUNNABLE (ready) states in which indicates that the process is recurrent and needs to have greater priority.

The following figure shows the TAT (turnaround time) and WT (waiting time) of the \$(ls) shell command using the priority-based scheduling algorithm, after the process has been terminated.



```
#####
Process Terminated
PID: 5
TAT: 42
WT: 4
PRIORITY BEFORE EXIT:20
#####
```

Figure II.18 shows the time analysis of the \$(ls) shell command

E. Multi-Level Feedback Queue

1) Theoretical background

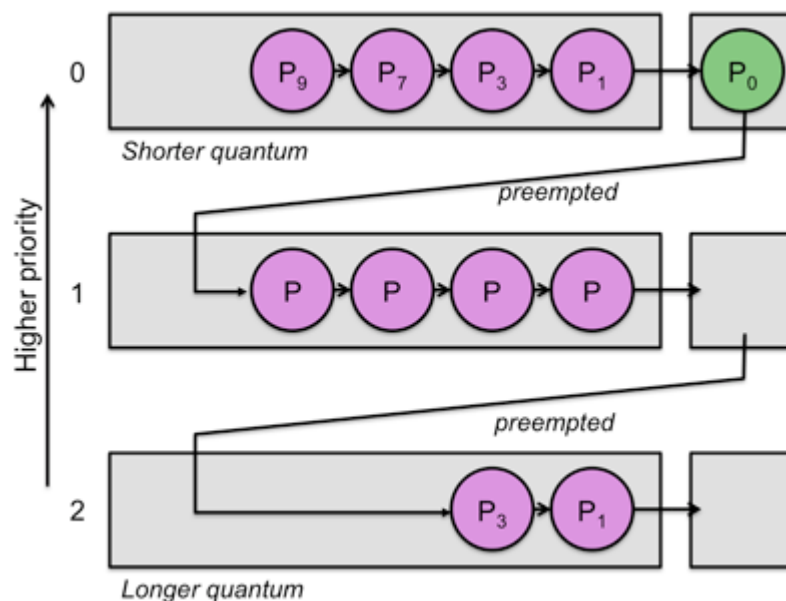
A CPU scheduler can't be either working only by a single scheduling algorithm nor by a single queue for outputting the processes to be scheduled.

A multilevel feedback queue indeed uses at 2 or more algorithms of queuing the processes into the processor most probably the first one is a round robin scheduler with a specified very low time quantum for the process then the processor preempts it out

The following queues might be also of type round robin but such a following queue with a round robin algorithm implementation must use a higher time quantum in order to serve the processes that didn't finish their required time in the first feedback queue

Following the round robin queues begins another type of scheduling algorithm which might be a FCFS algorithm, priority-based algorithm or even a SJF algorithm to finally serve the remaining processes till it finally terminates

Some of the multilevel feedback algorithms use a certain technique that switches between its queues and gives a certain time for the CPU to schedule in this queue as for example:



Give 80% of the time for the foreground queue which most probably is working with round robin scheduling

The next 20% of the time schedule the background processes which might be scheduled according to the priority of execution

Then return back to scheduling the foreground processes

In our implementation we see that small processes might end using the round robin scheduler and if they didn't end, they will move to a priority based one to schedule the highest priority one with demoting the ones scheduled at least once in the priority-based queue and promoting these who stayed much longer time without being executed (batch processes take highest priorities)

2) Implementation

At the beginning we found that we must add another attribute to the PCB to check if the process has been already scheduled in the round robin scheduler or not and it was added as an Enum called **isrr** in **proc.h** file at line 53

The main reason of using an enum is that it was already used in the process state and also that XV6 OS permitted us from using many things as booleans ,scanf and printf

```
35 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36 enum isrr{no,yes};
37 // Per-process state
38 struct proc {
39     uint sz; // Size of process memory (bytes)
40     pde_t* pgdir; // Page table
41     char *kstack; // Bottom of kernel stack for this process
42     enum procstate state; // Process state
43     int pid; // Process ID
44     struct proc *parent; // Parent process
45     struct trapframe *tf; // Trap frame for current syscall
46     struct context *context; // switch() here to run process
47     void *chan; // If non-zero, sleeping on chan
48     int killed; // If non-zero, have been killed
49     struct file *ofile[NOFILE]; // Open files
50     struct inode *cwd; // Current directory
51     char name[16];
52     int priority; //process priority for priority scheduling
53     enum isrr isRR; // used to determine if ran once in round robin or not
54 };
```

When the process is allocated to be put in the ptable it has to be set that it never entered the rqueue so is rr was set to no in allocproc method mentioned in **proc.c** at line 94

(alloc proc is referenced to allocate new processes into the ptable which is the queue in our schedulers)

```
86 found:
87     p->state = EMBRYO;
88     p->pid = nextpid++;
89     p->priority=10;
90
91
92 //set process at beginning of allocation that it didnt pass the RR scheduler
93
94     p->isRR=no;
95     release(&ptable.lock);
96
```

The next is to modify our implementation at scheduler () function that instead of only searching if the process is runnable but also that it didn't pass the round robin scheduler before so it can enter it now (line 353).

Once the process is allowed to enter the round robin scheduler after the context switching the referenced process is set to be have passed the round robin **isRR = yes** (line 366)

When all processes available in the ptable have already passed the round robin scheduler a break operation from its loop is done to release the lock and move to the priority based scheduler (**the counter check condition at line 352**)

```

333 scheduler(void)
334 {
335
336     struct proc *p;
337     struct proc *p1;
338     struct cpu *c = mycpu();
339     c->proc = 0;
340     struct proc *high_priority_process = 0; // points to a high priority process
341     // Loop over process table looking for process to run.
342     for(;;){
343
344         // Enable interrupts on this processor.
345         sti();
346
347         // Loop over process table looking for process to run.
348         acquire(&ptable.lock);
349         int count=0;
350         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
351         { count++;
352             if(count==NPROC)break;
353             if(p->state != RUNNABLE || p->isRR==yes)
354             {
355                 continue;
356             }
357             cprintf("process number %d with priority %d entered roundrobin scheduler\n",p->pid,p->priority);
358             // Switch to chosen process. It is the process's job
359             // to release ptable.lock and then reacquire it
360             // before jumping back to us.
361             c->proc = p;
362             switchvm(p);
363             p->state = RUNNING;
364             swtch(&(c->scheduler), p->context);
365             switchkvm();
366             p->isRR=yes;
367             // Process is done running for now.
368             // It should have changed its p->state before coming back.
369             c->proc = 0;
370         }
371         release(&ptable.lock);
372     }

```

Moving to the priority based scheduler that was mentioned in the section before this the only one change added is that when the process passes the priority scheduling it gets killed so when referenced again it is able to enter the round robin scheduler and if not killed but put on **EMBRYO** state it is also set to **isRR = no** so that the next time it gets scheduled using round robin scheduler (line 389)

```

364     acquire(&ptable.lock);
365     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
366         if(p->state != RUNNABLE)
367             continue;
368
369         high_priority_process = p; // put p as a high priority process
370
371         for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
372             if(p1->state != RUNNABLE)
373                 continue;
374             if (high_priority_process->priority < p1->priority) //found a process with higher priority (lower value)
375                 high_priority_process = p1;
376         }
377
378         // Switch to chosen process. It is the process's job
379         // to release ptable.lock and then reacquire it
380         // before jumping back to us.
381         p = high_priority_process;
382         cprintf("process number %d with priority %d entered priority based scheduler\n",p->pid,p->priority);
383         c->proc = p;
384         switchvm(p);
385         p->state = RUNNING;
386
387         swtch(&(c->scheduler), p->context);
388         switchkvm();
389         p->isRR=no;
390         if(p->priority < 20)
391             //change process priority as long as it was ran once before
392             p->priority++;
393
394         // Process is done running for now.
395         // It should have changed its p->state before coming back.
396         c->proc = 0;
397     }
398     release(&ptable.lock);
399
400 }
401
402 }

```


3) Performance

As has been illustrated, the multilevel feedback queue scheduling is where the ready queue diverges into several level queues (Round-robin and priority-based in our implementation) each of a unique scheduling algorithm. The results of the multilevel feedback queue scheduling algorithm that we have implemented in XV6 operating system illustrates the previously illustrated behavior.

Processes entering the round robin scheduler.

```
process number 1 with priority 12 entered roundrobin scheduler
process number 1 with priority 12 entered priority based scheduler
process number 1 with priority 13 entered roundrobin scheduler
process number 1 with priority 13 entered priority based scheduler
process number 1 with priority 14 entered roundrobin scheduler
process number 1 with priority 14 entered priority based scheduler
process number 1 with priority 15 entered roundrobin scheduler
process number 1 with priority 15 entered priority based scheduler
process number 1 with priority 16 entered roundrobin scheduler
process number 1 with priority 16 entered priority based scheduler
process number 1 with priority 17 entered priority based scheduler
process number 1 with priority 18 entered priority based scheduler
process number 1 with priority 19 entered roundrobin scheduler
process number 1 with priority 17 entered priority based scheduler
process number 1 with priority 18 entered roundrobin scheduler
process number 1 with priority 18 entered priority based scheduler
init: starting sh
process number 1 with priority 19 entered roundrobin scheduler
process number 1 with priority 19 entered priority based scheduler
process number 2 with priority 10 entered roundrobin scheduler
process number 2 with priority 10 entered priority based scheduler
process number 2 with priority 11 entered roundrobin scheduler
process number 2 with priority 11 entered priority based scheduler
process number 2 with priority 12 entered roundrobin scheduler
process number 2 with priority 12 entered priority based scheduler
process number 2 with priority 13 entered roundrobin scheduler
process number 2 with priority 13 entered priority based scheduler
process number 2 with priority 14 entered roundrobin scheduler
process number 2 with priority 14 entered priority based scheduler
process number 2 with priority 15 entered roundrobin scheduler
process number 2 with priority 15 entered priority based scheduler
```

Processes entering the priority-based scheduler.

```
process number 2 with priority 16 entered roundrobin scheduler
process number 2 with priority 16 entered priority based scheduler
process number 2 with priority 17 entered roundrobin scheduler
process number 2 with priority 17 entered priority based scheduler
process number 2 with priority 18 entered roundrobin scheduler
process number 2 with priority 18 entered priority based scheduler
process number 2 with priority 19 entered priority based scheduler
process number 2 with priority 20 entered roundrobin scheduler
process number 2 with priority 20 entered priority based scheduler
process number 2 with priority 20 entered priority based scheduler
process number 2 with priority 20 entered priority based scheduler
process number 2 with priority 20 entered roundrobin scheduler
process number 2 with priority 20 entered priority based scheduler
process number 2 with priority 20 entered roundrobin scheduler
process number 2 with priority 20 entered priority based scheduler
process number 2 with priority 20 entered roundrobin scheduler
process number 2 with priority 20 entered priority based scheduler
```


After typing ls command

```
kill process number 3 with priority 20 entered priority based scheduler
2 8 15128
process number 3 with priority 20 entered roundrobin scheduler
ln 2 process number 3 with priority 20 entered priority based scheduler
9 14980
process number 3 with priority 20 entered priority based scheduler
ls 2 10 17612
process number 3 with priority 20 entered roundrobin scheduler
mkdir process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered roundrobin scheduler
2 11 process number 3 with priority 20 entered priority based scheduler
15228
process number 3 with priority 20 entered roundrobin scheduler
process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered roundrobin scheduler
process number 3 with priority 20 entered priority based scheduler
rm process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered roundrobin scheduler
2 12 15204
sh process number 3 with priority 20 entered priority based scheduler
2 13 process number 3 with priority 20 entered roundrobin scheduler
process number 3 with priority 20 entered priority based scheduler
27840
process number 3 with priority 20 entered roundrobin scheduler
stressfs 2 process number 3 with priority 20 entered priority based scheduler
14 16116
process number 3 with priority 20 entered priority based scheduler
us process number 3 with priority 20 entered roundrobin scheduler
erests process number 3 with priority 20 entered priority based scheduler
2 15 67224
process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered priority based scheduler
wc process number 3 with priority 20 entered roundrobin scheduler
process number 3 with priority 20 entered priority based scheduler
2 16 16980
zombie process number 3 with priority 20 entered roundrobin scheduler
2 17 1 process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered priority based scheduler
4796
process number 3 with priority 20 entered roundrobin scheduler
process number 3 with priority 20 entered priority based scheduler
console process number 3 with priority 20 entered priority based scheduler
3 1 process number 3 with priority 20 entered priority based scheduler
process number 3 with priority 20 entered roundrobin scheduler
8 0
process number 2 with priority 19 entered priority based scheduler
process number 2 with priority 20 entered roundrobin scheduler
process number 2 with priority 20 entered priority based scheduler
```

The following figure shows the TAT (turnaround time) and WT (waiting time) of the \$(ls) shell command using the multilevel feedback queue scheduling algorithm, after the process has been terminated.

```
#####
Process Terminated
PID: 3
TAT: 92
WT: 33
PRIORITY BEFORE EXIT:20
#####
```

F. Comparative Analysis

Table II.1 Comparing different scheduling algorithms.

	Scheduler Algorithm			
Metric in time unit	RR	SJF	Priority	MLFQ
Average TAT	25	50	42	92
Average WT	8	15	4	33

Table II.2 Ranking scheduling algorithms.

Sort by ascending average TAT
RR
Priority
SJF
MLFQ

Sort by ascending average WT
Priority
RR
SJF
MLFQ

If measured performance accurately, we expect:

1. RR has the shortest average RT
2. SJF has the shortest average TAT
3. RR and Priority scheduling should have a high TAT due to the overhead caused by context switching often.

III. R2 — MEMORY MANAGEMENT

A. Briefing

XV6 uses page tables to manage its memory as the best memory management strategy in avoiding external fragmentations. In general page tables control memory addresses in a more secure way by having more than a single access address to. In general paging and page tables is a more secure way of managing the memory since it requires more than one address to instruction to fetch a certain memory allocation.

The simplest paging technique is the single page table indexing of a process to a memory allocation where in order to fetch a memory location we use a logical address of that process location in the page table then we use another address to index the page offset of that process to continue from

XV6 is an X86 based OS that uses Paging techniques hardware in both user and kernel space.

In x86 systems the paging hardware does the job of mapping the user logical space addresses to the kernel level space addresses by using a total of 32 bit for 2-level paging table hierarchy.

Page size is normally 4KB in XV6 which requires the least significant 12 bits of the page indexing address and the rest 20 bits themselves are used to index the page from the page table by using the other 2 additional addresses for accessing. So apparently XV6 has 2^{20} page table entries (logical ones) as they are the rest of the page address and each page table entry - PTE - has 2^{20} physical pages number indexing the real pages of processes left in the memory with their real physical addresses. When accessing the page using the 2-level hierarchy a request is sent for the operating system in order to translate virtual addresses into physical addresses that contain these 4KB contiguous random-access pages.

1) Memory accessing:

For a 2-level paging hierarchy it is divided as follows:

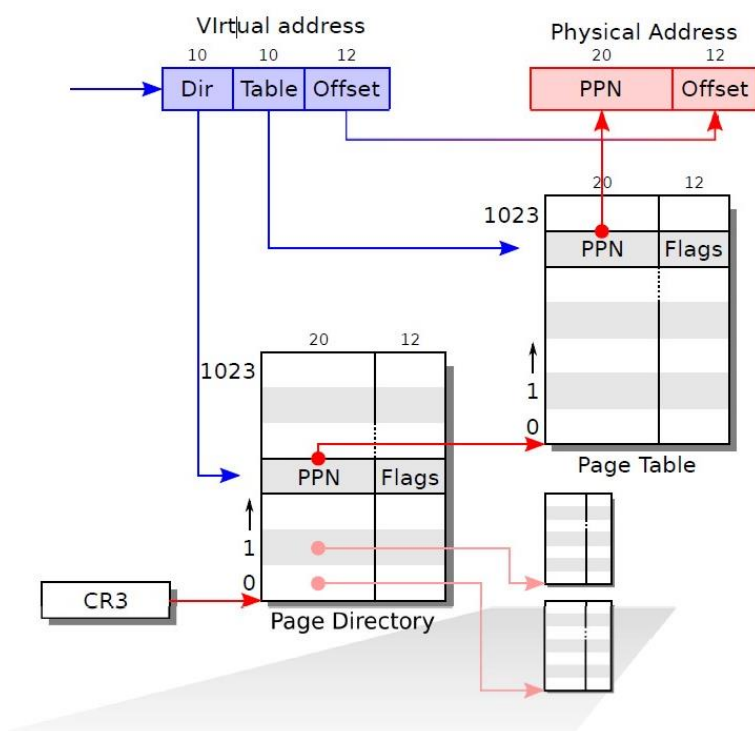
1. First 10 bits are used to index the page tables directory.
2. The second 10 bits are used to indicate which page to be used in this page table directory.
3. The rest 12 bits just represent the offset of the process state inside its allocated memory.

Each directory indexed by first 10 bits is called a page table entry -PTE- and it has references to page tables in the memory, it also has flags to indicate if this page table is Present **PTE_P**, it is used such that if a page table is present and the page indexed inside it is present it gives access to the offset inside this page, while if it is not present, a page fault is raised and

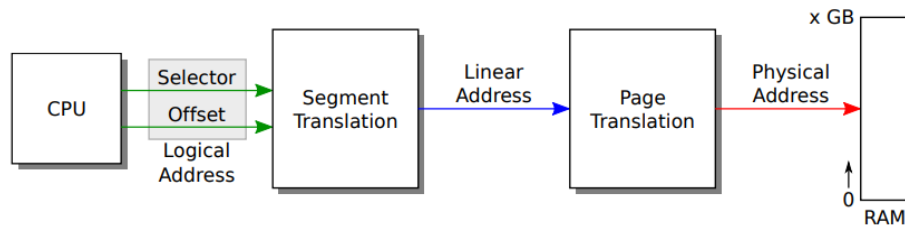
2-level paging enables removing of entire pages of a page table if there is a large waste of memory.

2) Pages and page table flags:

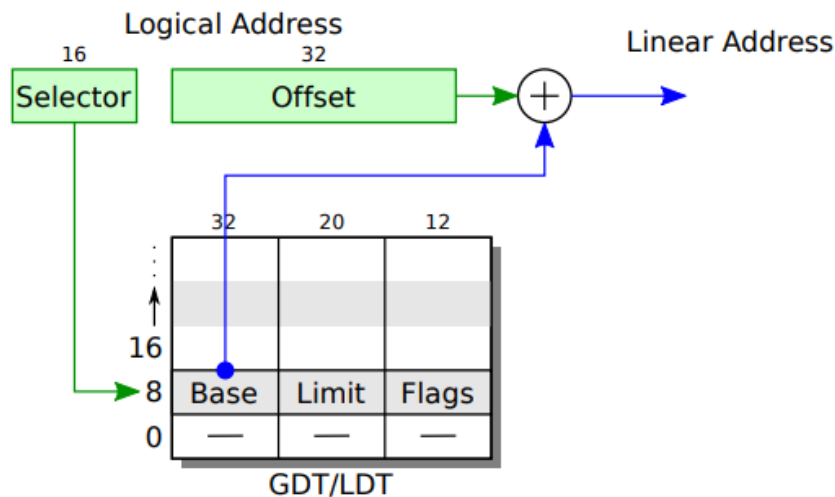
In XV6 the 2-level hierarchy gives both pages and page tables and their directories some flags to see if it is present **PTE_P** And if it can be accessed by user level processes **PTE_U** also if it is allowed to other processes to write in this page or only able to read **PTE_W**



3) Mapping logical address to physical address:



For a logical address consisting of a segment selector and an offset, if paging is enabled then the processor translates the linear address into physical one; otherwise, the processor uses linear address as physical address. Initially, the boot loader has the paging hardware disabled; hence, using - xv6 configured - segmentation hardware, it translates logical address to linear address without change; where both addresses are always equal while once paging gets enabled, the linear address is mapped to the physical address.



In protected mode, a segment register is an index into a segment descriptor table. For every table entry it specifies the minimum address (base physical address), maximum address (limit physical address) and permission bits for the segments; the permission bits correspond to the protection in the protection mode where it enables the kernel to use them as an insurance that a program uses only its own memory and does not exceed the determined allocated memory. Using the GDT table, the boot loader organizes that all segments have a base address of zero and maximum limit (4 GB); The GDT table has 3 entries, null entry, an entry for code and another for data. Using a flag, the code segment descriptor indicates that the code should run in 32-bit; Within this setup the boot loader is said to be in its protected mode where the logical addresses are mapped one-to-one to physical addresses.

B. Page Replacement

An important feature missing in XV6 is the ability to swap out pages in case of memory insufficiency to a backing store. That means, all the time, all processes are held in the same physical memory. We are going to implement a framework within which XV6 will be able to swap out pages and store them in disk. Moreover, it will swap back pages on demand.

First, we start by developing the process-paging framework. In this framework, each process is responsible for swapping in and out its own page. This means swapping is maintained locally for each process and not globally.

1) Supplied File Framework

We found a great framework which helps to create/delete/write/read a swap file. The framework was added to *fs.c* and a new *swapFile* pointer was added to the *proc* structure in *proc.h*. The swap files are named “*/.swap<id>*” where *id* is the process ID.

- `int createSwapFile(struct proc *p)` – Create a swap file for a given process *p* with ID *proc->pid*

- `int readFromSwapFile(struct proc *p, char* buffer, uint placeOnFile, uint size)` – Read size bytes into buffer `buffer` from `placeOnFile` index in a given process `p` swap file
- `int writeToSwapFile(struct proc *p, char* buffer, uint placeOnFile, uint size)` – Write size bytes into buffer `buffer` to `placeOnFile` index in a given process `p` swap file
- `Int removeSwapFile(struct proc *p)` - Delete the swap file for a given process `p`. Requires `p->pid` to be correctly initiated.

2) Memory Constrains for Process

In any given time, a process must not have physical pages **MAX_PSYC_PAGES =15** and no larger total pages **MAX_TOTAL_PAGES=30**. Whenever the max physical pages are violated, a swap occur.

This limit was chosen because the XV6 file system does not accept memory more than approx. 17 pages. That means user processes cannot have more than 30 pages. Shell and init processes are not affected or considered by our framework.

When a page is swapped out. It is marked in the page table to be not present and that can be done by **PTE_P** flag. But since the **PTE_P** can happen for other reasons rather a swapping out. We decided to add another flag :

```
#define PTE_PG 0x200
```

Now, whenever a page is swapped out, this secondary flag is set.

3) Retrieving a page

A running process might request a page which is swapped out. In this case, a trap is raised `trap(interrupt 14, T_PGFLT)`. After verifying the trap. We use the `%CR2` register to determine the fault address and retrieve the page.

For retrieving the page, we allocate a new physical page, copy its data from the file, and then pass it to the mapper to map it to the page table. After that, we return to the same command which caused the trap. We also check if the **MAX_PSYC_PAGES** is violated and swap out another page.

C. Page Replacement Algorithms

Now that we have a full swapping mechanism. We need to figure out which page to swap first. We will discuss to algorithms for that: FIFO and LRU (least recently used).

1) First in First Out Algorithm (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Our implementation differs slightly. We never have an actual queue, but we assign an order to each process and increment that order. When we check for the process to be swapped, we start with `loadOrder = 0xFFFFFFFF` and decrement to the first order.

To do that we need to add a data structure to handle the order, so we edit in “`proc.h`” to add a data structure.

```
1. enum page_struct_state {NOTUSED, USED};
2. struct pagecontroller {
3.     enum page_struct_state state;
4.     pde_t* pgdir;
5.     uint loadOrder;
6. };
```

```
1. struct proc {
2.     //SAME CODE AS BEFORE
3.     //Swap file. must initiate with create swap file
4.     struct file *swapFile;           //page file
5.     struct pagecontroller fileCtrlr[MAX_TOTAL_PAGES-MAX_PSYC_PAGES];
6.     struct pagecontroller ramCtrlr[MAX_PSYC_PAGES];
7.     uint loadOrderCounter; //load/creation
8. };
```

Where *ramCtrlr* and *fileCtrlr* are used to control pages non-swapped and swapped pages respectively. The *loadOrder* is the order of a page loaded inside a process. *Pgdir* is a pointer to the page table. *State* acts as a flag for whether the page is swapped or not.

For implementation of FIFO, we loop over the pages of a process and return the least *loadOrder*.

```
1. int getFIFO(){
2.     int i = 0;
3.     int pageIndex;
4.     uint loadOrder;
5.     recheck:
6.     pageIndex = -1;
7.     loadOrder = 0xFFFFFFFF;
8.     for ( i = 0; i < MAX_PYSC_PAGES; i++) {
9.         if (proc->ramCtrlr[i].state == USED && proc->ramCtrlr[i].loadOrder <= loadOrder){
10.            pageIndex = i;
11.            loadOrder = proc->ramCtrlr[i].loadOrder;
12.        }
13.    }
14.    return pageIndex;
15. }
```

We will talk about the overall swapping mechanism after LRU.

a) FIFO Test

We start by running a process and fill up 15 memory pages (1 code, 1 space, 1 stack, 14 sbrk):

```
1. int i, j;
2. char *arr[14];
3. char input[10];
4. // Allocate all remaining 12 physical pages
5. for (i = 0; i < 12; ++i) {
6.     arr[i] = sbrk(PGSIZE);
7.     printf(1, "arr[%d]=0x%x\n", i, arr[i]);
8. }
```

```
$ myMemTest
arr[0]=0x3000
arr[1]=0x4000
arr[2]=0x5000
arr[3]=0x6000
arr[4]=0x7000
arr[5]=0x8000
arr[6]=0x9000
arr[7]=0xA000
arr[8]=0xB000
arr[9]=0xC000
arr[10]=0xD000
arr[11]=0xE000
Called sbrk(PGSIZE) 12 times - all physical pages taken.
Press any key...
1 sleep 3 0 0 0 init 80105969 801056f1 80107229 801063ee 80107633 80107419
2 sleep 4 0 0 0 sh 80105969 801056f1 80107229 801063ee 80107633 80107419
3 sleep 15 0 0 0 myMemTest 80105969 80100a60 80102001 80101284 801065c4 8010639
56765/57054 free pages in the system
```

We can notice that our process (process 3) contains 15 allocated memory pages, 0 pages swapped out, 0 page faults, and 0 page swaps.

This allocation would cause page 0 to move to the swap file, but upon returning to user space, a PGFLT would occur and pages 0,1 will be hot swapped (page 0 contain code). Afterwards, page 1 is in the swap file, the rest are in memory.

```
1. arr[12] = sbrk(PGSIZE);
```



```
2. printf(1, "arr[12]=0x%x\n", arr[12]);
```

```
arr[12]=0xF000
Called sbrk(PGSIZE) for the 13th time, a page fault should occur and one page i.
Press any key...
1 sleep 3 0 0 0 init 80105969 801056f1 80107229 801063ee 80107633 80107419
2 sleep 4 0 0 0 sh 80105969 801056f1 80107229 801063ee 80107633 80107419
3 sleep 16 1 1 2 myMemTest 80105969 80100a60 80102001 80101284 801065c4 8010639
56765/57054 free pages in the system
```

Now we can notice that the physical pages number did not change (56765). While the process total pages increased to 16 with 1 page swapped out, 1-page fault (hot swapping of 0), and 2 pages swaps (page 0 and 1).

Allocate page 16. this would cause page 2 which contains the stack to move to the swap file, it would be hot swapped with page 3. Afterwards, pages 1 & 3 are in the swap file, the rest are in memory.

```
1. arr[13] = sbrk(PGSIZE);
2. printf(1, "arr[13]=0x%x\n", arr[13]);
```

```
arr[13]=0x10000
Called sbrk(PGSIZE) for the 14th time, a page fault should occur and two pages .
Press any key...
1 sleep 3 0 0 0 init 80105969 801056f1 80107229 801063ee 80107633 80107419
2 sleep 4 0 0 0 sh 80105969 801056f1 80107229 801063ee 80107633 80107419
3 sleep 17 2 2 4 myMemTest 80105969 80100a60 80102001 80101284 801065c4 8010639
56765/57054 free pages in the system
```

Again, we can notice that the physical pages number did not change (56765). While the process total pages increased to 17 with 2 pages swapped out, 2-page fault (hot swapping of 0 and 2), and 4 pages swaps (page 0, 1, 2 and 3).

Access page 3, causing a PGFLT, since it is in the swap file. It would be hot swapped with page 4. Page 4 is accessed next, so another PGFLT is invoked, and this process repeats a total of 5 times.

```
1. for (i = 0; i < 5; i++) {
2.     for (j = 0; j < PGSIZE; j++)
3.         arr[i][j] = 'k';
```

```
5 page faults should have occurred.
Press any key...
1 sleep 3 0 0 0 init 80105969 801056f1 80107229 801063ee 80107633 80107419
2 sleep 4 0 0 0 sh 80105969 801056f1 80107229 801063ee 80107633 80107419
3 sleep 17 2 7 9 myMemTest 80105969 80100a60 80102001 80101284 801065c4 8010639
56765/57054 free pages in the system
```

2) LRU

LRU (least recently used) is a great algorithm for swapping which acts as a good approximation to OPT (optimal page replacement algorithm). Instead of looking at the future and swapping the pages which will be mostly used then. It looks at the past and assumes that if a page is being used recently then it is more likely to be used at this moment.

The LRU algorithm is simple. We start with an *accessCount* for each page. This counter holds the number of times a page has been accessed. At every swap, we swap out the least accessed one. The swapped in page counter is then reset to 0.

To manage the accessCount we can add to our data structure *pagecontroller*

```
1. struct pagecontroller {
2.     //SAME CODE AS BEFORE
3.     uint accessCount;
4. };
```

Now that we our data structure is ready, we can implement the LRU. We start with *minAccess* = 0xffffffff and loop through the pages of the process to find the least a *accessCount*.

```
1. int getLRU(){
2.     int i;
3.     int pageIndex = -1;
```

```

4.  uint minAccess = 0xffffffff;
5.  uint loadOrder = 0xffffffff;
6.
7.  for (i = 0; i < MAX_PYSC_PAGES; i++) {
8.      if (proc->ramCtrlr[i].state == USED && proc->ramCtrlr[i].accessCount <= minAccess && proc->ramCtrlr[i].loadOrder < loadOrder) {
9.          minAccess = proc->ramCtrlr[i].accessCount;
10.         pageIndex = i;
11.         loadOrder = proc->ramCtrlr[i].loadOrder;
12.     }
13. }
14. return pageIndex;
15. }

```

We need to figure out when to increment the *accessCount* of the process. If page is used and is accessed, we increment *accessCount* and disable access flag.

```

1. void updateAccessCounters(struct proc * p){
2.     pte_t * pte;
3.     int i;
4.     for (i = 0; i < MAX_PYSC_PAGES; i++) {
5.         if (p->ramCtrlr[i].state == USED){
6.             pte = walkpgdir(p->ramCtrlr[i].pgdir, (char*)p->ramCtrlr[i].userPageVAddr,0);
7.             if (*pte & PTE_A) {
8.                 *pte &= ~PTE_A; // turn off PTE_A flag
9.                 p->ramCtrlr[i].accessCount++;
10.            }
11.        }
12.    }
13. }

```

Finally, we need to figure out when to call this function. A good implementation is to use the timer interrupt to do so.

a) LRU Testing

We repeat the whole steps in the FIFO test (please check code above). After inserting the 16th page, We notice that the space page is hotswapped (least used last order) and we end up with 16 pages with 1 page swapped out, zero faults, and 1 swap.

```

9 11:0 12:0 13:0 14:0 15:0 16:0 17:0 18:0 8:9 9:0 10:9 19:0 20:0 21:0 22:0
arr[12]=0xF000
Called sbrk(PGSIZE) for the 13th time, no page fault should occur and one page.
Press any key...
1 sleep  3 0 0 0 init 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
2 sleep  4 0 0 0 sh 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
5 sleep 16 1 0 1 myMemTest 8010596e 80100a60 80102001 80101284 801066c6 801061
56765/57054 free pages in the system

```

In the first row, 1st column indicate page index to swap. Then next 15 column are page indexes from 0 to 14, where page 8, 9, and 10 contain code, space, and stack respectively. Moreover, we can see the *loadOrder* and *accessCount* of each page

We then allocate 17th page. Again, no page faults

```

0 11:0 12:0 13:0 14:0 15:0 16:0 17:0 18:0 8:14 23:0 10:14 19:0 20:0 21:0 22:0
arr[13]=0x10000
Called sbrk(PGSIZE) for the 14th time, no page fault should occur and two page.
Press any key...
1 sleep  3 0 0 0 init 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
2 sleep  4 0 0 0 sh 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
5 sleep 17 2 0 2 myMemTest 8010596e 80100a60 80102001 80101284 801066c6 801061
56765/57054 free pages in the system

```

We then access pages 0, 1, and 2 to cause 3 page-faults.

```

1. arr[0][3] = 'k';
2. arr[1][3] = 'k';
3. arr[2][3] = 'k';

```



```

1 24:0 25:0 13:0 14:0 15:0 16:0 17:0 18:0 8:17 23:0 10:17 19:0 20:0 21:0 22:0
2 24:0 25:1 26:2 14:0 15:0 16:0 17:0 18:0 8:19 23:0 10:19 19:0 20:0 21:0 22:0
3 24:0 25:1 26:2 27:1 15:0 16:0 17:0 18:0 8:20 23:0 10:20 19:0 20:0 21:0 22:0
3 page faults should have occurred.
Press any key...
1 sleep 3 0 0 0 init 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
2 sleep 4 0 0 0 sh 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
3 sleep 17 2 3 5 myMemTest 8010596e 80100a60 80102001 80101284 801066c6 801061
56765/57054 free pages in the system

```

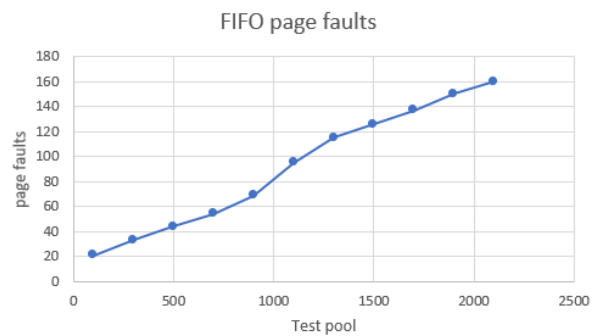
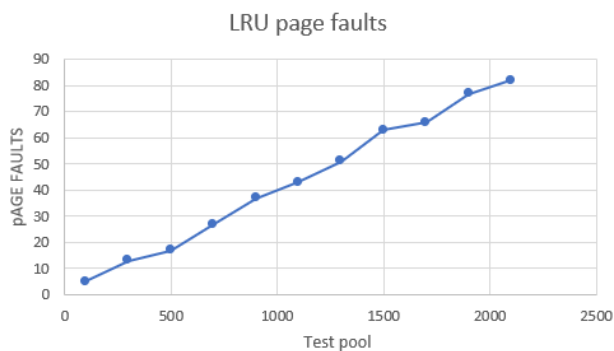
3) FIFO & LRU Performance test

In the first test, we wanted to have only 2 extra pages (total of 17 page). That means we only swaps if the code, stack, or extra pages are chosen. After that, we randomly chose a page and write to it. We can notice that the LRU did better because the code and stack pages were not hot-swapped as much as in the FIFO.

```

1. arr = malloc(ARR_SIZE); //allocates 14 pages (sums to 17)
2. for (i = 0; i < TEST_POOL; i++) {
3.     randNum = getRandNum(); //generates a pseudo random number between 0 and ARR_SIZE
4.     arr[randNum] = 'X'; //write to memory
5. }

```



In second test, we want to violate the assumption of the LRU (recently used pages will be used again). So, we will do a linear sweep. This will be done on a **TEST_POOL = 500**, with 30 pages.

```

1. arr = malloc(ARR_SIZE); //allocates 27 pages (sums to 30)
2. for (i = 0; i < TEST_POOL; i++) {
3.     for(j = 0; j < ARR_SIZE; j+= PGSIZE)
4.         arr[j] = 'X';
5. }

```

```

1 sleep 3 0 0 0 init 8010596e 801056f6 8010732b 801064f0 8010774b 80107531
2 runble 4 0 0 0 sh
3 zombie 30 15 7680 7695 myMemTest
56765/57054 free pages in the system

```

We can see a total of 7680-page faults, with comparison to random sweep 17-page faults.

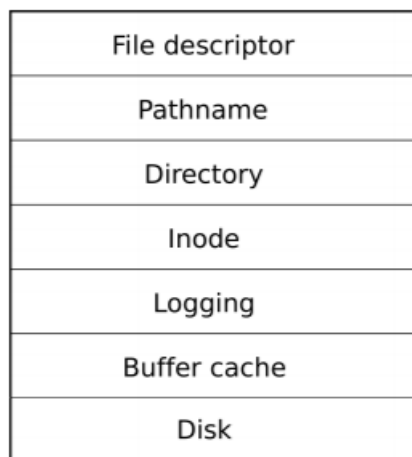
In conclusion, we can say that the LRU is better than the FIFO. But overall, there is no better swapping algorithm because in certain cases, an algorithm can perform very poorly. That's why some CPUs use multiple swapping algorithms and choose which one to use according to situation.

IV. R3 — FILE MANAGEMENT

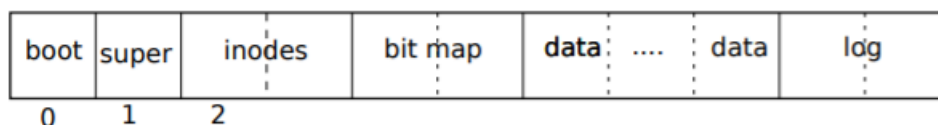
A. Briefing

The xv6 file system uses Unix-like pathnames, directories, and files. Its implementation is organized into the following seven layers as shown in the figure __:

1. The disk layer: reads and writes blocks on an IDE hard drive
2. The buffer cache: caches disk blocks and maintains access synchronization
3. The logging layer: as a journaling mechanism in the case of crashes
4. The inode layer: provides individual files, each as an inode structure
5. The directory layer: implements each directory as a special inode structure that contains a sequence of directory entries such as file name and I-number
6. The pathname layer: provides hierarchical path names
7. The file descriptor layer: provide abstraction such as pipes, devices, files



The disk is divided into blocks as shown in figure __. The first block (block 0), holds the boot sector of the system. Block 1 holds the superblock which stores metadata about the file system; It stores the size of the file system in blocks, the number of inodes, the number of data blocks, and the number of log blocks. Then starting from block 2 we have blocks holding inodes, after which come the bit map blocks, then the data blocks, and finally the log blocks.



In the management of empty space, XV6 holds a bitmap on disk with each bit representing the status of a block. If a bit is set to one, it indicates an occupied block, and if its set to zero, it indicates an available block. The bits representing the boot, super, inodes, and bitmap blocks are always set.

The system's block allocator module provides two functions: *balloc* for allocation of new disk blocks, and *bfree* for freeing of blocks. *Balloc* goes over the disk drive block by block starting from after the bitmap and checks the corresponding bit in the bitmap to check if the block is available. It returns the address of the block and sets the corresponding bitmap bit if the block is available. *Bfree* simply finds the appropriate bit in the bitmap and clears it.

When a new file is created, an inode is reserved for it. The inode holds the size of the file and an array of block numbers. The size of the array is $\text{NDIRECT} + 1$. The first NDIRECT entries represent the addresses of actual blocks on the disk. The last entry holds the address to a block that holds an array of NINDIRECT block numbers, such that the file can hold up to $\text{NDIRECT} + \text{NINDIRECT}$ disk blocks.

This summary covers the main concerns for integrating extents into XV6's file system.

B. Using User-defined Extents

1) Implementation:

Extents were integrated into the file system rather than becoming the predominant method of the system. The system's native files retain their inode structures such that they keep track of blocks using an array of block numbers. The extent inode structures keep track of the reserved blocks using an array of extents. Each extent is a 32-bit unsigned integer; the most significant 24 bits hold the address of the block at which the extent starts, while the least significant 8 bits hold the length of the extent.

Now when we need a block of data of any file, we can use the bmap function which will the file block number to the disk block number, we are going to explain our bmap function implementation.

```

377 // The first 24 bits are address
378 #define EXT_ADDR(addr) (addr >> (32-24))
379 // The last byte is the length
380 #define EXT_LEN(addr) (addr & 0x00ff)
381 // Generate address from block number and length
382 #define GEN_ADDR(bn,len) (bn << 8 | len)

```

Figure IV.1 file system code part 1

First we made three definitions, one of them is to extract the address, another to extract the length and the last one generates an address when given the block number and a length.

```

384 static uint
385 bmap(struct inode *ip, uint bn)
386 {
387     uint addr, *a;
388     struct buf *bp;
389     if (ip->type == T_EXT_FILE){
390         int n = 0;
391         uint current_length=0;
392         // check if the file type is extent based
393         while(ip->addrs[n]){
394             //extract length
395             uint length = EXT_LEN(ip->addrs[n]);
396             // check if the requested block is within the extent
397             if(bn>= current_length&& bn<length+current_length){
398                 //extract address
399                 addr = EXT_ADDR(ip->addrs[n]) + bn-current_length;
400                 return addr;
401             }
402             // if not within the extent, move to the next extent
403             current_length+= length;
404             n++;
405             if (n>NDIRECT) break;
406         }
407         if (n>NDIRECT){
408             panic("Extent file allocation exceeded\n");
409             return EXT_ADDR(ip->addrs[n-1]);
410         }
411     }

```

Figure IV.2 fs part 2- bmap part 1

In our bmap function we can see that it takes two parameters, one of them is a pointer to inode *ip and the other represents the block number bn , inside our function we first make a pointer to address *a and *bp is a pointer to a buffer.

On line 389, we have an if condition that checks if the type of the received file is an extent base file, now we will discuss how we deal with it in case it is an extent base.

On line 393, we will loop on our inode addresses, afterwards using our defined function EXT_LEN we can get the length of the block we are currently pointing to, then we start checking if the block that we requested lies within the extent on line 397, in that case we are going to return the address of that block using the defined function EXT_ADDR.

In case if the block is not within the extent, we move to the next block and check that that n does not exceed the NDIRECT.

In case n exceeds the NDIRECT, the loop will break and the condition at line 407 will be satisfied and we will return the last extent address and print the error message described in panic in line 408.

```

407     if (n>NDIRECT){
408         panic("Extent file allocation exceeded\n");
409         return EXT_ADDR(ip->addrs[n-1]);
410     }
411     // if the block is not found
412     // create a new block
413     uint newBlock = balloc(ip->dev);
414     //check if contiguous
415     uint len = EXT_LEN(ip->addrs[n-1]);
416     if( (newBlock == EXT_ADDR(ip->addrs[n-1]) + len) && EXT_LEN(ip->addrs[n-1]) < 0xff){
417         ip->addrs[n-1] += 1;
418     }
419     else{
420         // if not, add a new entry in the table
421         ip->addrs[n] = GEN_ADDR(newBlock, 1);
422     }
423     return newBlock;
424 }
425
426

```

Figure IV.3 bmap part 2

However, if we finished if the loop completes and the block is not found, we will need to allocate a new block using `ballocc` as in line 414, after that we extract the length and check if the new created block is contiguous by the if condition at line 417, but if the block is not contiguously allocated, in that case we will need to add a new entry to the table, using our defined function `GET_ADDR` and then we return the new block.

```

483 // Truncate inode (discard contents).
484 // Only called when the inode has no links
485 // to it (no directory entries referring to it)
486 // and has no in-memory reference to it (is
487 // not an open file or current directory).
488 static void
489 itrunc(struct inode *ip)
490 {
491     int i, j;
492     struct buf *bp;
493     uint *a;
494
495     // modified
496     /*truncate extent files*/
497     if (ip->type == T_EXT_FILE){
498         for(i = 0; i < NDIRECT; i++){
499             if(ip->addrs[i]){
500                 uint len = EXT_LEN(ip->addrs[i]);
501                 for(j = 0; j < len; j++){
502                     bfree(ip->dev, EXT_ADDR(ip->addrs[i]) + j);
503                 }
504                 ip->addrs[i] = 0;
505             }
506         }
507     }
508     return;
509 }
510 /*end code*/

```

Figure IV.4 fs part 3 - itrunc

Now that we finished explaining the `bmap()` function, we are going to explain how the data is de-allocated, the de-allocated of data is done by the function in figure 20, the `itrunc()` function, when it is time to delete a file, a certain system call called `unlink` calls the `itrunc` function de-allocate the files data.

As we see on line 497 we first check if the file type is a user-extent file, if so, then on line 498 we start looping on the file till we reach the `NDIRECT` part, that means we finished looping all over the extent-file.

Inside the loop, line 499, checks if the address exists, then we will extract that address and loop on the length, while looping we will free (de-allocating using `bfree()`) each block.

After the de-allocating process of an address is complete, on line 504, we assign that address to null and then we continue to the next iteration of the bigger loop and the whole process is repeated until all the blocks of the extent file are free.

```

6 struct stat {
7     short type; // Type of file
8     int dev; // File system's disk device
9     uint ino; // Inode number
10    short nlink; // Number of links to file
11    uint size; // Size of file in bytes
12    //modified
13    struct extents_details{
14        uint addr:24, // Extent Pointers
15        len:8; // Extent Length
16    }extents[12];
17 };
18

```

Figure IV.6 stat struct modification

```

537 void
538 stati(struct inode *ip, struct stat *st)
539 {
540     st->dev = ip->dev;
541     st->ino = ip->inum;
542     st->type = ip->type;
543     st->nlink = ip->nlink;
544     st->size = ip->size;
545
546     // Get more info for extent files
547     int i = 0;
548     if (st->type == T_EXT_FILE){
549         for(i = 0; i < NDIRECT; i++){
550             {
551                 st->extents[i].addr = EXT_ADDR(ip->addrs[i]);
552                 st->extents[i].len = EXT_LEN(ip->addrs[i]);
553             }
554         }
555     }

```

Figure IV.5 stati modification

Other details: in Figure IV.6 `stat` struct modification we added lines 13 to 16 for the extent files details and on Figure IV.5 `stati` modification we added lines 546 to 555 for collecting the extent file information.

2) Performance

To test the efficacy of the extent file system, a few test cases were carried out.

a) Test Case (1)

A file of extent type was created by setting the O_EXTENT flag in the open() system call. A buffer of 1024 bytes was created and filled to be used for writing into the file. In the loop, we write 1024 bytes 80 times, which gives a total file size of 80 KB if successful.

```
int fd = open("test.txt", O_CREATE|O_EXTENT|O_RDWR);
if (fd <= 0){
    printf(1, "File open failed\n");
    exit();
}

char x[1024];
int i;
for(i = 0; i < 1024; i++){
    x[i] = 'a';
}

for (i = 0; i < 80; i++){
    int size = write(fd, x, 1024);

    if (size < 1024)
    {
        printf(2, "Write failed. Return code %d\n", size);
        exit();
    }
}

printf(1, "Wrote 80 * 1024 bytes successfully\n");
```

Figure IV.7 fs test case 1

Results:

```
$ ./FileTesting
Wrote 80 * 1024 bytes successfully
File size 81920 bytes
Extent addr:706. Extent Len:160
$
```

Figure IV.8 fs test case 1 result

Analysis:

In the native file system, a maximum of a NDIRECT + NINDIRECT blocks can be held by any single file. If we have NDIRECT set to 12 and NINDIRECT set to 58, this gives a maximum of 70 KB per file (since a single block is 512 bytes). In this test case, we attempt to write 80 KB into an extent-based file. The write was successful and was completed in a single extent of length 160 blocks. This implies that the maximum size of an extent-based file is much larger than a native file.

b) Test Case (2)

Two extent-based files fd and fd2 were created and written to simultaneously.

```
int main(){

    int fd = open("test.txt", O_CREATE|O_EXTENT|O_RDWR);
    int fd2 = open("test1.txt", O_CREATE|O_EXTENT|O_RDWR);
    if (fd <= 0 || fd2 <= 0){
        printf(1, "File open failed\n");
        exit();
    }

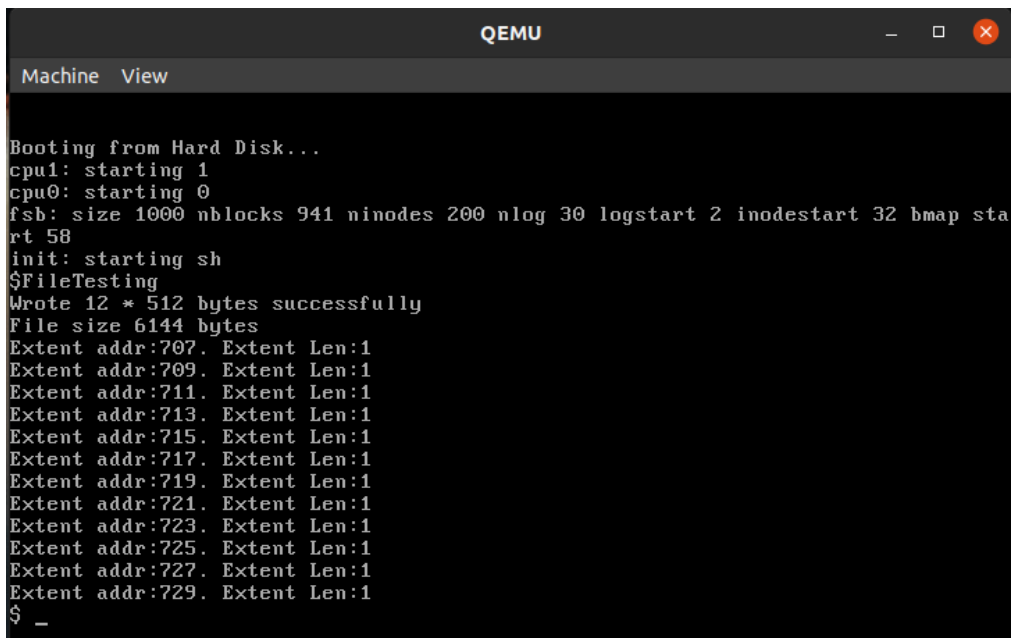
    char x[1024];
    int i;
    for(i = 0; i < 1024; i++){
        x[i] = 'a';
    }

    for (i = 0; i < 12; i++){
        int size = write(fd, x, 512);
        int size2 = write(fd2, x, 512);

        if (size < 512 || size2 < 512)
        {
            printf(2, "Write failed. Return code %d\n", size);
            exit();
        }
    }
}
```

Figure IV.9 fs test case 2

Results:



```
Machine View
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
fsb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
rt 58
init: starting sh
$FileTesting
Wrote 12 * 512 bytes successfully
File size 6144 bytes
Extent addr:707. Extent Len:1
Extent addr:709. Extent Len:1
Extent addr:711. Extent Len:1
Extent addr:713. Extent Len:1
Extent addr:715. Extent Len:1
Extent addr:717. Extent Len:1
Extent addr:719. Extent Len:1
Extent addr:721. Extent Len:1
Extent addr:723. Extent Len:1
Extent addr:725. Extent Len:1
Extent addr:727. Extent Len:1
Extent addr:729. Extent Len:1
$ _
```

Figure IV.10 fs test case 2 result

Analysis:

C.

Due to the nature of balloc (it searches the data blocks one by one to find an empty block), if we simultaneously write to two files, the extent length is limited by the number of blocks written in a single write() request. Furthermore, since extent files can have a maximum of NDIRECT extents, the benefits of extents diminish and perform worse than the native file type where the worst case scenario as shown in the results of this test case is that the extent-based file can hold a maximum of 12 KBs.

a) Test Case (3)

The underlying file system was tested to make sure it was not afflicted by the alterations made. Text2.txt represents a normal file, while text.txt represents the extent type file. The extent flag was not set for fd2.

```
int main(){
    int fd = open("test.txt", O_CREATE|O_EXTENT|O_RDWR);
    int fd2 = open("test1.txt", O_CREATE|O_RDWR);
    if (fd <= 0 || fd2 <= 0){
        printf(1, "File open failed\n");
        exit();
    }

    char x[1024];
    int i;
    for(i = 0; i < 1024; i++){
        x[i] = 'a';
    }

    for (i = 0; i < 12; i++){
        int size = write(fd, x, 512);
        int size2 = write(fd2, x, 512);

        if (size < 512 || size2 < 512)
        {
            printf(2, "Write failed. Return code %d\n", size);
            exit();
        }
    }
}
```

Figure IV.11 fs test case 3

Results:

```
Wrote 12 * 512 bytes successfully
File size 6144 bytes
Extent addr:707. Extent Len:1
Extent addr:709. Extent Len:1
Extent addr:711. Extent Len:1
Extent addr:713. Extent Len:1
Extent addr:715. Extent Len:1
Extent addr:717. Extent Len:1
Extent addr:719. Extent Len:1
Extent addr:721. Extent Len:1
Extent addr:723. Extent Len:1
Extent addr:725. Extent Len:1
Extent addr:727. Extent Len:1
Extent addr:729. Extent Len:1
```

Figure IV.12 fs test case 3 result.i

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16444
echo      2 4 15296
forktest  2 5 9608
grep      2 6 18660
init      2 7 15880
kill      2 8 15324
ln        2 9 15180
ls        2 10 17932
mkdir     2 11 15420
rm        2 12 15404
sh        2 13 28036
stressfs  2 14 16312
usertests 2 15 67420
wc        2 16 17176
zombie    2 17 14992
FileTesting 2 18 17128
console   3 19 0
test.txt  4 20 6144
test1.txt 2 21 6144
```

Figure IV.13 fs test case 3 result.ii

D. Analysis

Both writes were successful. The native file performed as expected.