PROJECT 2

SYSTEM ENVIRONMENT

- Operating System: Ubuntu 22.04.3 LTS
- Xv6 Version: xv6-public
- GCC Version: 11.4.0

IMPLEMENTATION

PART 1: UNIQ

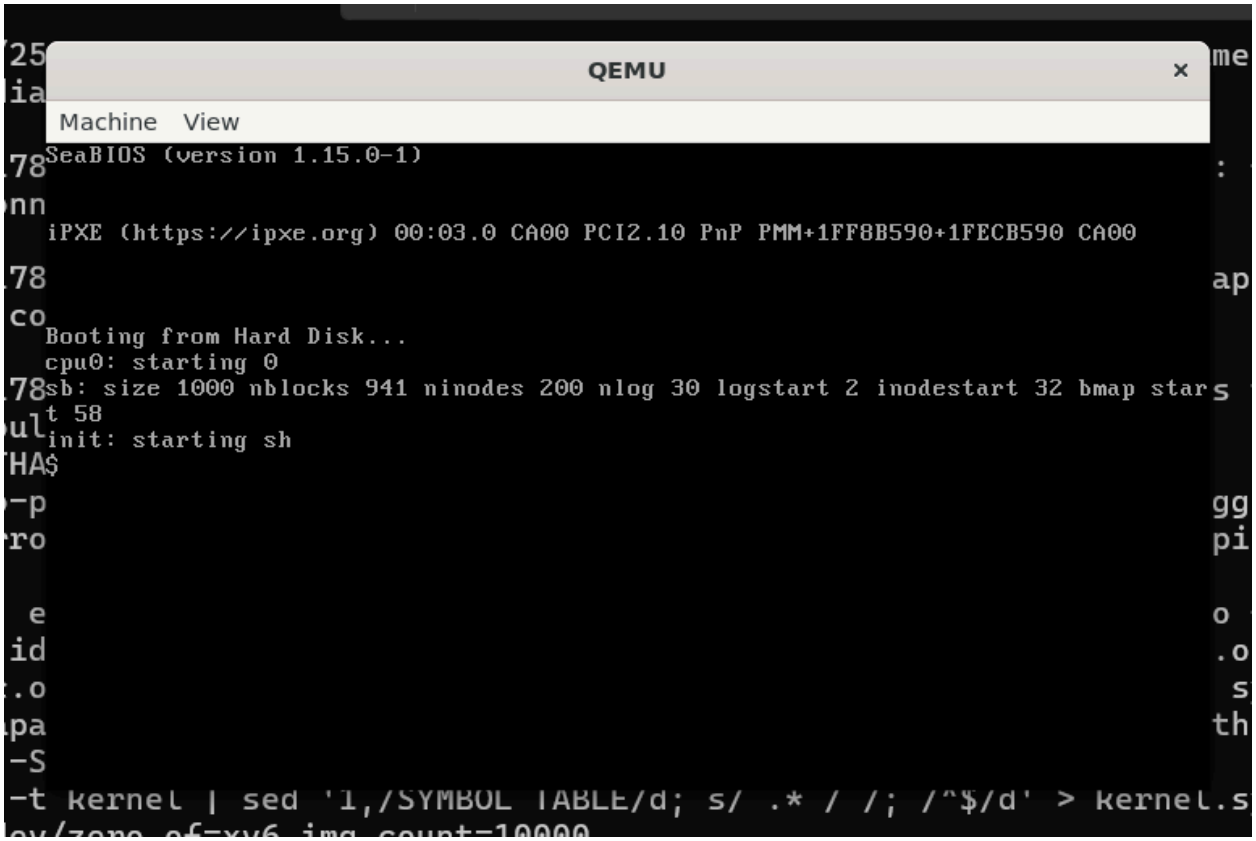
1. Created a file *uniq.c* that filters repetitive adjacent lines from the input file and implemented -c, -u, - w [N] functionality.
2. Added _uniq to the UPROGS in Makefile
3. Run the following commands
    a. make clean - to clean up all generated intermediate and binary files

```
root@GUTHA:~/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _use
rtests _wc _zombie _hello _sleep _uniq _find _tickstest _ps _dproc _priortyt
est
root@GUTHA:~/xv6-public#
```

    b. make - to compile the source code in `Makefile` and generate intermediate and binary files

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -D
DEFAULT -nostdinc -I. -c initcode.S
ld -m    elf_i386 -N -e start -Ttext 0 -o initcode.out initcode.o
objcopy -S -O binary initcode.out initcode
objdump -S initcode.o > initcode.asm
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file
.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pip
e.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o syspr
oc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0319572 s, 160 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 7.2294e-05 s, 7.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
414+1 records in
414+1 records out
212036 bytes (212 kB, 207 KiB) copied, 0.000833607 s, 254 MB/s
root@GUTHA:~/xv6-public#
```

c. make qemu - to bootup the compiled version of Xv6 in an environment



d. Create a file with some content using cat command and test uniq commands

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
$
$ cat > uniqtest
This is project 2
This is project 2
I like Windows OS
I like windows os
This is to test uniq implementation
This is a great project
An apple a day keeps the doctor away
All that glitters is not gold
lets go bulls
Happy diwali!
Happy diwali!     uniqtest
exec: failli!$ cat
exec catuniqtest failed
uniq uniqtest
This is project 2
I like Windows OS
I like windows os
This is to test uniq implementation
This is a great project
An apple a day keeps the doctor away
All that glitters is not gold
lets go bulls
Happy diwali!
$
```

Created uniqtest file, tested basic uniq functionality

```
$
$ uniq -c uniqtest
2 This is project 2
1 I like Windows OS
1 I like windows os
1 This is to test uniq implementation
1 This is a great project
1 An apple a day keeps the doctor away
1 All that glitters is not gold
1 lets go bulls
3 Happy diwali!
$
```

uniq -c uniqtest

```
$
$ uniq -u uniqtest
I like Windows OS
I like windows os
This is to test uniq implementation
This is a great project
An apple a day keeps the doctor away
All that glitters is not gold
lets go bulls
$
```

uniq -u uniqtest

```
$
$
$ uniq -w 4 uniqtest
This is project 2
I like Windows OS
This is to test uniq implementation
An apple a day keeps the doctor away
All that glitters is not gold
lets go bulls
Happy diwali!
$
```

uniq -w 4 uniqtest

```
$
$ cat uniqtest | uniq
This is project 2
I like Windows OS
I like windows os
This is to test uniq implementation
This is a great project
An apple a day keeps the doctor away
All that glitters is not gold
lets go bulls
Happy diwali!
$
```

cat uniqtest | uniq

PART 1: FIND

1. Created a file *find.c* with the functionality that find all the files in a directory tree with a specific name (given by the flag -name), -type f, -type d, -inum, -printi flags
2. Added _find in UPROGS in Makefile
3. Run the following commands
    a. make clean - to clean up all generated intermediate and binary files

```
$ root@GUTHA:~/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _use
rtests _wc _zombie _hello _sleep _uniq _find _tickstest _ps _dproc _priortyt
est
```

    b. make - to compile the source code in `Makefile` and generate intermediate and binary files

```
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file
.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pip
e.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o syspr
oc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0320039 s, 160 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 7.7303e-05 s, 6.6 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
414+1 records in
414+1 records out
212028 bytes (212 kB, 207 KiB) copied, 0.000892286 s, 238 MB/s
root@GUTHA:~/xv6-public#
```

    c. make qemu - to bootup the compiled version of Xv6 in an environment

```
s 941 total 1000
balloc: first 915 blocks have been allocated
balloc: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,for
mat=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
start 58
init: starting sh
$
$
$
```

Created some files, directories and tested find functionality with all options

```
$
echo > b
$ mkdir a
$ echo > a/b
$ mkdir a/aa
$ echo > a/aa/b
$
$ find . -name b
./b
./a/b
./a/aa/b
$ find a -name b
a/b
a/aa/b
$
$ find a/aa -name b
a/aa/b
$
```

Created few files and directories, tested find basic functionality

```
$
$ echo > a/aa/c
$ find . -name c -type f
./a/aa/c
$
$ mkdir c
$ find . -name c -type f
./a/aa/c
$
```

```
$
$ find . -name c -type d
c
$
```

Created a file and directory with name c and tested **-type f, d** options

```
$
$ find . -name b -inum 30
./a/b
$
$ find . -name b -inum +28
./a/b
./a/aa/b
$
$ find . -name b -inum -32
./b
./a/b
$
```

Tested with -inum flag

```
$
$ find . -name b -printi
28 ./b
30 ./a/b
32 ./a/aa/b
$
$
```

-printi flag

PART 2: ticks_running()

1. proc.h - Added a new field to the `struct proc` called burst_time to store the ticks for each process in RUNNING STATE.
2. proc.c - Updated burst_time in allocproc() to zero, calculated the ticks of a process in RUNNING STATE and added it to burst_time. Implemented helper for calculating ticks_running.
3. sysproc.c - Implemented sys_ticks_running() using helper ticks_running() in proc.c
4. Added new syscall number for ticks_running()
5. Syscall.c, usys.S, defs.h, user.h - Registered and declared the system call.
6. Added Written a program ticks_running_test.c to test ticks_running().
7. Run the following commands
    a. make clean - to clean up all generated intermediate and binary files

```
$ root@GUTHA:~/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _use
rtests _wc _zombie _hello _sleep _uniq _find _tickstest _ps _dproc _priortyt
est
root@GUTHA:~/xv6-public#
```

    b. make - to compile the source code in `Makefile` and generate intermediate and binary files

```
oc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0424986 s, 120 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000100937 s, 5.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
414+1 records in
414+1 records out
212028 bytes (212 kB, 207 KiB) copied, 0.00103958 s, 204 MB/s
root@GUTHA:~/xv6-public#
```

c. make qemu - to bootup the compiled version of Xv6 in an environment

```
start 58
init: starting sh
$
$
$ ticks_running_test 100
Current Process PID: 5
ticks_running() returned -1 for process 100 as it does not exist in process
table.
$
$ ticks_running_test 6
Current Process PID: 7
ticks_running() returned -1 for process 6 as it does not exist in process ta
ble.
$ ticks_running_test 8
Current Process PID: 8
Process 8 has run for 309 ticks.
$
```

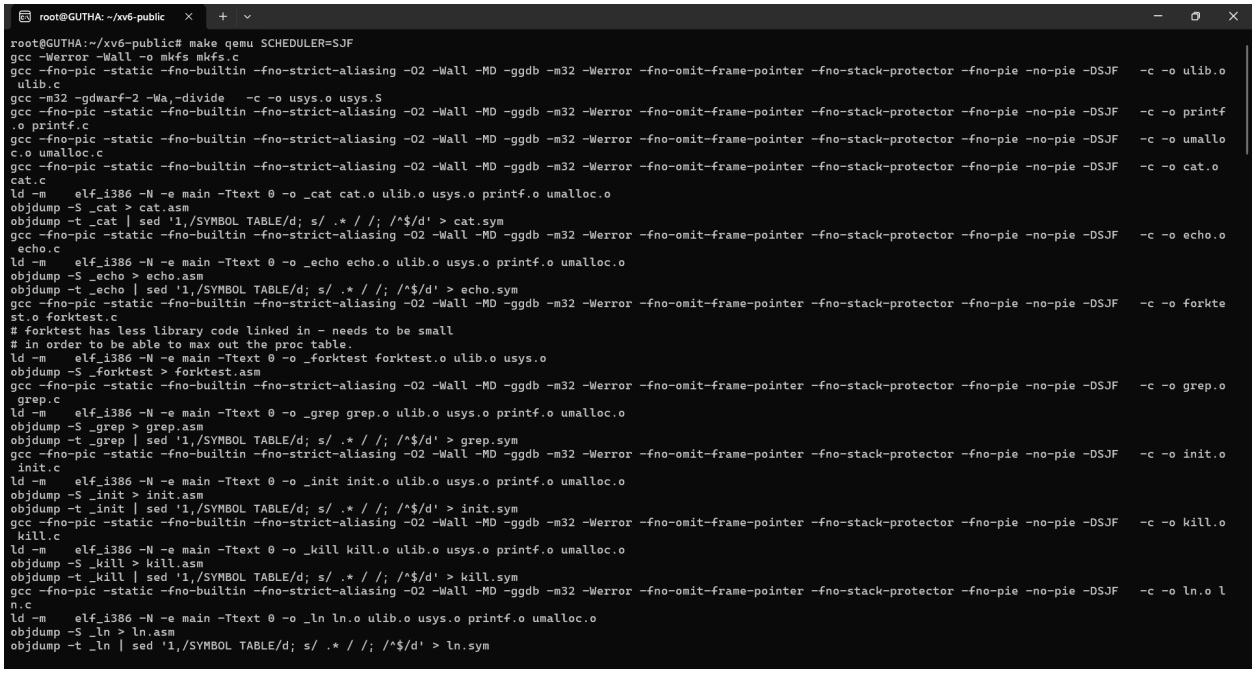
Tested ticks_running() system call by running ticks_running_test.c

PART 3: IMPLEMENTING A SIMPLE SCHEDULER

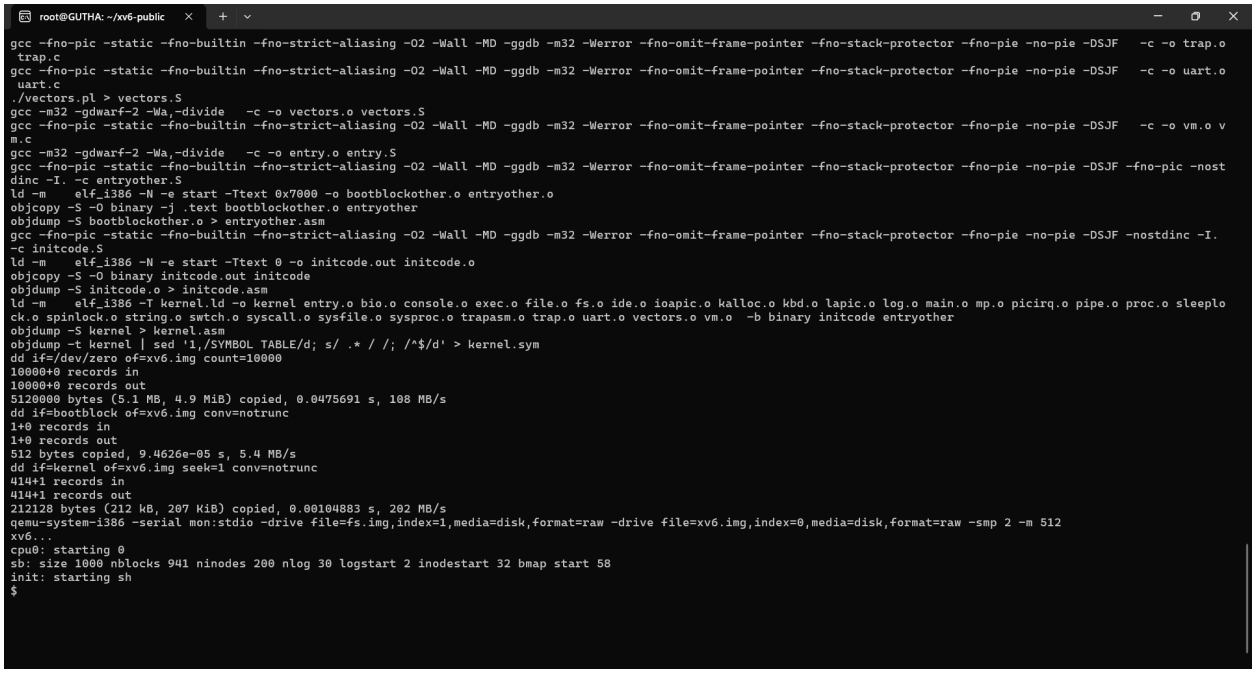
1. Makefile - Added support to switch between default and SJF schedulers at compile time using the SCHEDULER flag.
2. proc.h - Added a new field called predicted_burst to the proc structure to store the predicted job length.
3. proc.c
    a. Implemented rand() method that generates random numbers
    b. Assigned predicted_burst during the process allocation in allocproc() using rand() method
    c. Implemented SJF Scheduling logic in the scheduler() by selecting the process with the least predicted_burst.
    d. Implemented the sjf_job_length() for system call
4. sysproc.c - Added the sys_sjf_job_length() that can call the sjf_job_length() and return the predicted burst of the process with the given pid
5. syscall.h - Added syscall number for sjf_job_length()
6. Syscall.c, usys.S, defs.h, user.h - Registered and declared the system call.
7. trap.c - Modified it such that the process doesn't yield() after the timeslice. As in Simple Scheduler, there is no preemption, the process continues to execute till its completion.
8. Added a sjf_job_length_test.c to test the sjf_job_length() system call.
9. Run the following commands
    a. make clean - to clean up all generated intermediate and binary files

```
root@GUTHA:~/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _hello _sleep _uniq _find _tickstest _ps _dproc _priortytest
_ticks_running_test _sjf_job_length_test
root@GUTHA:~/xv6-public#
```

b. make qemu SCHEDULER=SJF - to compile the source code in `Makefile` and generate intermediate, binary files and bootup the xv6 environment with SIMPLE SCHEDULER



```
root@GUTHA: ~/xv6-public
root@GUTHA:~/xv6-public# make qemu SCHEDULER=SJF
gcc -Werror -Wall -o mkfs mkfs.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o ulib.o
 ulib.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o usys.o usys.S
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o printf
.o printf.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o umallo
c.o umalloc.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o cat.o
cat.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _cat cat.o ulib.o usys.o printf.o umalloc.o
objdump -S _cat > cat.asm
objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > cat.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o echo.o
 echo.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _echo echo.o ulib.o usys.o printf.o umalloc.o
objdump -S _echo > echo.asm
objdump -t _echo | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > echo.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o forkte
st.o forktest.c
# forktest has less library code linked in - needs to be small
# in order to be able to max out the proc table.
ld -m    elf_i386 -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
objdump -S _forktest > forktest.asm
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o grep.o
 grep.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _grep grep.o ulib.o usys.o printf.o umalloc.o
objdump -S _grep > grep.asm
objdump -t _grep | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > grep.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o init.o
 init.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _init init.o ulib.o usys.o printf.o umalloc.o
objdump -S _init > init.asm
objdump -t _init | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > init.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o kill.o
 kill.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _kill kill.o ulib.o usys.o printf.o umalloc.o
objdump -S _kill > kill.asm
objdump -t _kill | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kill.sym
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o ln.o l
n.c
ld -m    elf_i386 -N -e main -Ttext 0 -o _ln ln.o ulib.o usys.o printf.o umalloc.o
objdump -S _ln > ln.asm
objdump -t _ln | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > ln.sym
```



```
root@GUTHA: ~/xv6-public
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o trap.o
 trap.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o uart.o
 uart.c
./vectors.pl > vectors.S
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o vectors.o vectors.S
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF   -c -o vm.o v
m.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o entry.o entry.S
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF -fno-pic -nost
dinc -I. -c entryother.S
ld -m    elf_i386 -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
objcopy -S -O binary -j .text bootblockother.o entryother
objdump -S bootblockother.o > entryother.asm
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -DSJF -nostdinc -I.
-c initcode.S
ld -m    elf_i386 -N -e start -Ttext 0 -o initcode.out initcode.o
objcopy -S -O binary initcode.out initcode
objdump -S initcode.o > initcode.asm
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplo
ck.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0475691 s, 108 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 9.4626e-05 s, 5.4 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
414+1 records in
414+1 records out
212128 bytes (212 kB, 207 KiB) copied, 0.00104883 s, 202 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

```
414+1 records out
212128 bytes (212 kB, 207 KiB) copied, 0.00104883 s, 202 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw
 -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 5
8
init: starting sh
$
$ sjf_job_length_test
Current Process PID: 4
Usage: sjf_job_length_test <pid>
$
$ sjf_job_length_test 6
Current Process PID: 6
Process 6 is predicted to run for 5627 ticks.
$
$ sjf_job_length_test 100
Current Process PID: 8
Process with PID: 100. does not exist.
$
$ sjf_job_length_test 15
Current Process PID: 10
Process with PID: 15. does not exist.
$
$
$
```

Tested sjf_job_length() system call running sjf_job_length_test

Added a simple_scheduler_test.c that creates 10 children and demonstrates the simple scheduler algorithm.

```
init: starting sh
$ simple_scheduler_test
Parent process PID 3 started
Parent process checking ticks for all children...
Child: 0 PID: 4 has been created
Child: 1 PID: 5 has been created
Child: 2 PID: 6 has been created
Child: 3 PID: 7 has been created
Child: 4 PID: 8 has been created
Child: 5 PID: 9 has been created
Child: 6 PID: 10 has been created
Child: 7 PID: 11 has been created
Child: 8 PID: 12 has been created
Child: 9 PID: 13 has been created
CHILD    PID         PREDICTCED BURST    TICKS
 7       11              2749           530
 6       10              4086           530
 2       6               5627           522
 4       8               7419           574
 9       13              9084           529
 8       12              12767          532
 5       9               16212          548
 0       4               17515          528
 3       7               23010          532
 1       5               31051          529
Parent process: All children have finished execution.
Parent process took 5372 ticks to complete
$
```

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 5
8
init: starting sh
$ simple_scheduler_test
Parent process PID 3 started
Parent process checking ticks for all children...
Child: 0 PID: 4 has been created
Child: 1 PID: 5 has been created
Child: 2 PID: 6 has been created
Child: 3 PID: 7 has been created
Child: 4 PID: 8 has been created
Child: 5 PID: 9 has been created
Child: 6 PID: 10 has been created
Child: 7 PID: 11 has been created
Child: 8 PID: 12 has been created
Child: 9 PID: 13 has been created
CHILD    PID         PREDICTCED BURST    TICKS
 6       10              4086            6066
 9       13              9084            6058
 1       5               31051           6078
 4       8               7419            6074
 5       9               16212           6085
 8       12       3      7               23010           6091
         12767           6075
 2       6               5627            6098
 0       4               1751 7          11              2749            6090
5               6107
Parent process: All children have finished execution.
Parent process took 6111 ticks to complete
```

Output of the Simple_scheduler_test when run with default(round robin scheduler)

SIMPLE SCHEDULER VS DEFAULT SCHEDULER

Added func_exec_test.c to answer the questions

What effects do you see in the existing functionality of xv6, including those you implemented in the last project?

The functionality of commands like ls, stressfs, sleep, cat README remains the same except

1. In the Simple scheduler, processes with the shortest predicted job lengths are given priority.
2. This delays the execution of jobs with longer predicted bursts, which might lead to increased wait times for processes with longer predicted job lengths.
3. The order of process execution differs from the default round-robin scheduler, where each process gets equal time slices fairly.

How do the execution times of stressfs, ls, uniq, and find compare between the original scheduler and the new one you implemented?

DEFAULT SCHEDULER: Screenshots for exec times and functionality

```
Executing: uniq test.txt
This is to test the uniq implementation
apple
banana
bananashake
carrot
carrotsoup
Execution time for uniq test.txt: 3 ticks
```

```
Executing: find . -name README
./README
Execution time for find . -name README: 7 ticks
```

```
Executing: stressfs
stressfs starting
write 0
write 1
write 2
write 3
write 4
read
read
read
read
read
Execution time for stressfs: 35 ticks
```

```
Executing: ls
README          2 2 2286
test.txt        2 3 129
cat             2 4 15756
echo            2 5 14636
forktest        2 6 9080
grep            2 7 18600
init            2 8 15260
kill            2 9 14720
ln              2 10 14616
ls              2 11 17988
mkdir           2 12 14744
rm              2 13 14724
sh              2 14 28784
stressfs        2 15 15652
usertests       2 16 63156
wc              2 17 16180
zombie          2 18 14300
hello           2 19 14476
sleep           2 20 14772
uniq            2 21 19768
find            2 22 19944
tickstest       2 23 17676
ps              2 24 14172
ticks_running_ 2 25 15940
sjf_job_length 2 26 15948
simple_schedul 2 27 17380
func_exec_test 2 28 17100
console         3 29 0
stressfs0       2 30 10240
stressfs1       2 31 10240
stressfs2       2 32 10240
stressfs3       2 33 10240
stressfs4       2 34 10240
Execution time for ls: 20 ticks
```

```
We are also grateful for the bug reports and patches contributed by Silas
Boyd-Wickizer, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, eyalz800,
Nelson Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter
Froehlich, Yakir Goaron,Shivam Handa, Bryan Henry, Jim Huang, Alexander
Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin
Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,
Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".Execution time for cat README: 6 ticks
```

```
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,
Eldar Sehayek, Yongming Shen, Cam Tenn
y, tyfkda, Rafael Ubal, Warren
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, eve
n on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".
Execution time for cat README | uniq: 42 ticks
```

SIMPLE SCHEDULER: Screenshots for exec times and functionality

```
Executing: find . -name README
./README
Execution time for find . -name README: 7 ticks
```

```
Executing: stressfs
stressfs starting
write 0
write 1
write 3
write 4
read
read
write 2
read
read
read
Execution time for stressfs: 25 ticks
```

```
Executing: ls
README          2 2 2286
test.txt        2 3 129
cat             2 4 15756
echo            2 5 14636
forktest        2 6 9080
grep            2 7 18600
init            2 8 15260
kill            2 9 14720
ln              2 10 14616
ls              2 11 17988
mkdir           2 12 14744
rm              2 13 14724
sh              2 14 28784
stressfs        2 15 15652
usertests       2 16 63156
wc              2 17 16180
zombie          2 18 14300
hello           2 19 14476
sleep           2 20 14772
uniq            2 21 19768
find            2 22 19944
tickstest       2 23 17676
ps              2 24 14172
ticks_running_ 2 25 15940
sjf_job_length 2 26 15948
simple_schedul 2 27 17380
func_exec_test 2 28 17100
console         3 29 0
stressfs0       2 30 10240
stressfs2       2 31 10240
stressfs1       2 32 10240
stressfs4       2 33 10240
stressfs3       2 34 10240
Execution time for ls: 18 ticks
```

```
Will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".Execution time for cat README: 5 ticks
```

```
Executing: uniq test.txt
This is to test the uniq implementation
apple
banana
bananashake
carrot
carrotsoup
Execution time for uniq test.txt: 3 ticks
```

```
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, eve
n on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".
Execution time for cat README | uniq: 42 ticks
Benchmarking completed
```

COMMAND	TICKS: DEFAULT SCHEDULER	TICKS: SIMPLE SCHEDULER
stressfs	35	25
ls	20	19
cat README	6	8
uniq test.txt	3	3
find . -name README	7	7
Cat README | uniq	42	42

From the above observations, we can see that the execution times for simple processes like ls, uniq, and find remain almost the same whereas the execution times of complex process like stressfs are less in the simple scheduler when compared to the default scheduler.

What about pipes and fork? How does the OS behave when you execute complex commands such as ”cat filename | uniq” etc. with the new scheduler?

When executing piped command ‘cat README | uniq’, the functionality of the command remains the same while the execution times are less or equal compared to the default scheduler.

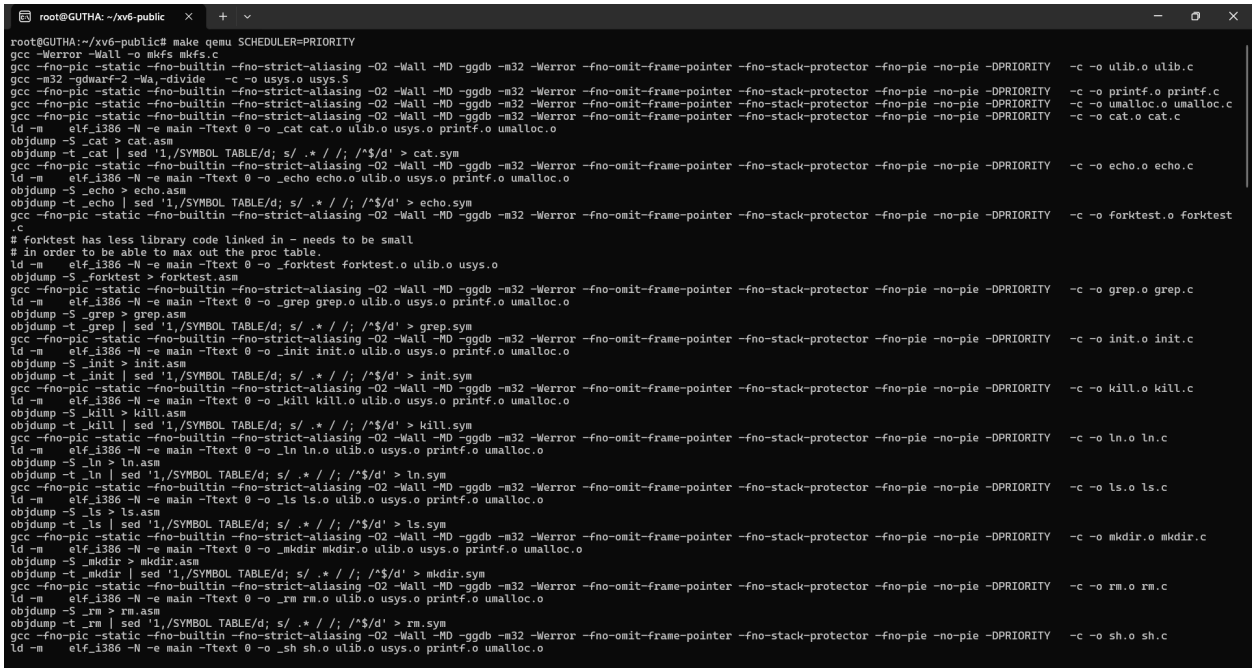
PART 4: A (More) Advanced Scheduler - PRIORITY SCHEDULER

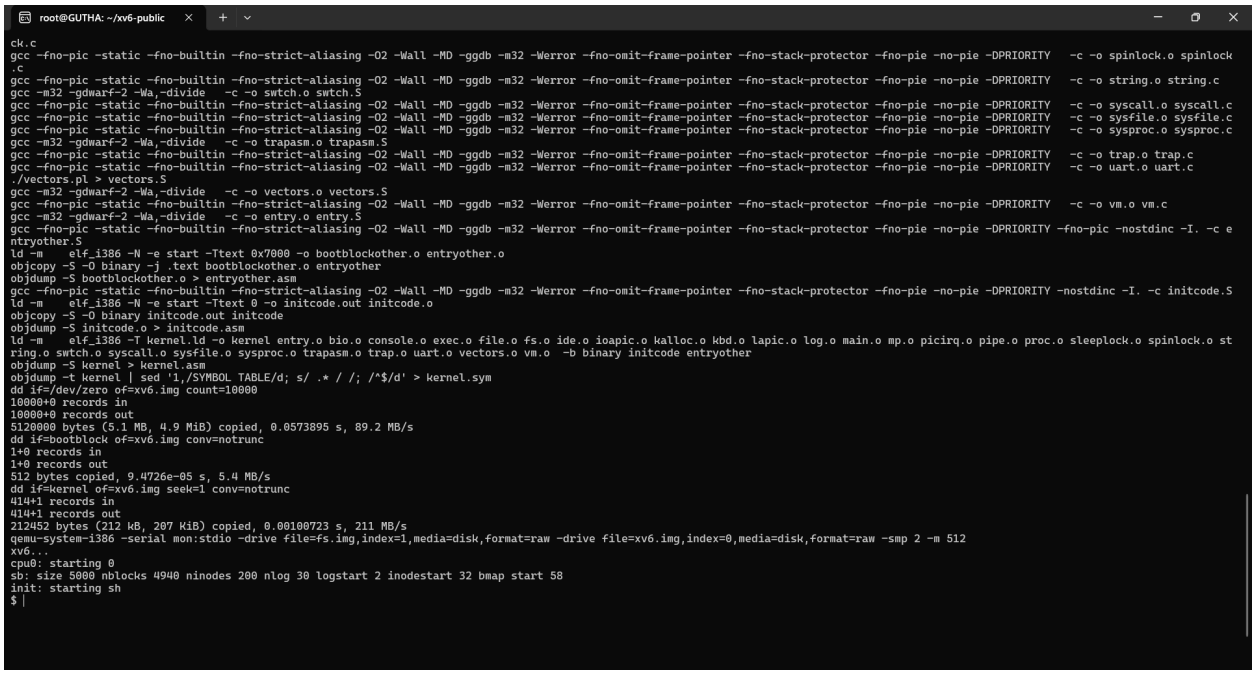
1. Makefile - Added support to switch between DEFAULT, SJF and PRIORITY schedulers at compile time using the SCHEDULER flag.
2. proc.h
    a. Added a new field called priority to the proc structure to store the priority of the process

    b. Defined 3 priority levels, PRIORITY_LOW, PRIORITY_MEDIUM, PRIORITY_HIGH
3. proc.c
    a. Implemented get_random(min, max) method that generates random numbers between min and max. (min inclusive, max exclusive)
    b. Assigned PRIORITY_LOW to priority in proc structure during the process allocation in allocproc()
    c. Implemented PRIORITY-SJF Scheduling logic in the scheduler() by selecting the process with the highest priority first, or the process with the least predicted_burst time if there are multiple processes with the same priority.
    d. Implemented the set_sched_priority(priority), and get_sched_priority(pid) system calls to set the priority of the current process and get the priority of the process with particular pid respectively.
4. exec.c - For a parent process to execute, its child processes must be executed first, therefore, the priority of the child processes must be less than the priority of the parent so i have assigned, all the child processes with priority as PRIORITY_MEDIUM in the exec file.
5. sysproc.c - Added the sys_set_sched_priority(), sys_get_sched_priority() that can call the set_sched_priority() and get_sched_priority() implemented in the proc.c file.
6. syscall.h - Added syscall number for sjf_job_length()
7. Syscall.c, usys.S, defs.h, user.h - Registered and declared the system calls.
8. trap.c - Modified it such that the process doesn't yield() after the timeslice. As in Priority Scheduler, there is no preemption, the process continues to execute till its completion.
9. Added a get_set_sched_priority_test.c to test the system calls.
10. Run the following commands
    a. make clean - to clean up all generated intermediate and binary files

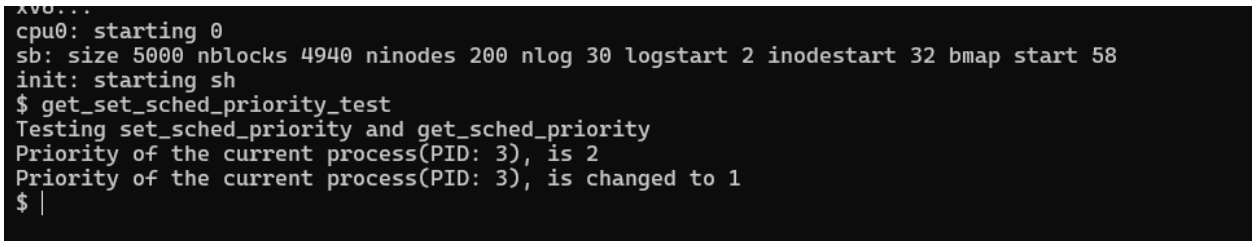
```
root@GUTHA:~/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests
_wc _zombie _hello _sleep _uniq _find _tickstest _ps _ticks_running_test _sjf_job_l
ength_test _simple_scheduler_test _func_exec_test _get_set_sched_priority
root@GUTHA:~/xv6-public#
```

    b. Make qemu SCHEDULER=PRIORITY, to the xv6 using PRIORITY scheduler





Added get_set_sched_priority_test to test get and set methods



NOTE:

1. To demonstrate the Priority scheduling, I have added set_sched_priority_for_pid(int pid, int priority) so that I can change the priority of any process with a particular pid that is currently in runnable or running state.
2. I have used the set_sched_priority_for_pid(int pid, int priority) in priority_sched_test.c to change the priority of all child processes with **odd pid** to **PRIORITY_HIGH**.

PRIORITY-SJF SCHEDULER

```
sb: size 5000 nblocks 4940 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ priority_sched_test
Parent process PID 3 started
Parent process checking ticks for all children...
Child: 0 PID: 4 has been created
Child: 1 PID: 5 has been created
Child: 2 PID: 6 has been created
Child: 3 PID: 7 has been created
Child: 4 PID: 8 has been created
Child: 5 PID: 9 has been created
Child: 6 PID: 10 has been created
Child: 7 PID: 11 has been created
Child: 8 PID: 12 has been created
Child: 9 PID: 13 has been created
CHILD    PID         PREDICTCED BURST    PRIORITY        TICKS
 7       11              2749            1              621
 9       13              9084            1              590
 5       9               16212           1              587
 3       7               23010           1              591
 1       5               31051           1              593
 6       10              4086            3              586
 2       6               5627            3              591
 4       8               7419            3              591
 8       12              12767           3              590
 0       4               17515           3              585
Parent process: All children have finished execution.
Parent process took 5947 ticks to complete
```

priority_sched_test.c

BASIC FUNCTIONALITY AND EXECUTION TESTING

```
$ func_exec_test
Benchmarking commands under the current scheduler...

Executing: stressfs
stressfs starting
write 0
write 1
write 2
write 4
read
read
read
write 3
read
read
Execution time for stressfs: 45 ticks
```

stressfs

```
Executing: ls
README          2 2 2286
test.txt        2 3 129
cat             2 4 15800
echo            2 5 14680
forktest        2 6 9132
grep            2 7 18644
init            2 8 15300
kill            2 9 14768
ln              2 10 14664
ls              2 11 18028
mkdir           2 12 14788
rm              2 13 14768
sh              2 14 28824
stressfs        2 15 15700
usertests       2 16 63196
wc              2 17 16224
zombie          2 18 14348
hello           2 19 14520
sleep           2 20 14816
uniq            2 21 19812
find            2 22 19984
tickstest       2 23 17716
ps              2 24 14216
ticks_running_ 2 25 15984
sjf_job_length 2 26 15996
simple_schedul 2 27 17420
func_exec_test 2 28 17144
get_set_sched_ 2 29 14840
priority_sched 2 30 17292
console         3 31 0
stressfs0       2 32 10240
stressfs1       2 33 10240
stressfs2       2 34 10240
stressfs4       2 35 10240
stressfs3       2 36 10240
Execution time for ls: 23 ticks
```

```
Kapsnak, Anders Kaseorg, kenao95, Wolfgang Ketter, Eddie Kohler, Austin
Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,
Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".Execution time for cat README: 9 ticks
```

```
Executing: uniq test.txt
This is to test the uniq implementation
apple
banana
bananashake
carrot
carrotsoup
Execution time for uniq test.txt: 4 ticks
```

```
Executing: find . -name README
./README
Execution time for find . -name README: 7 ticks
```

```
The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, eve
n on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".
Execution time for cat README | uniq: 41 ticks
Benchmarking completed.
```

SCHEDULERS REPORT

EXECUTION TIMES IN TICKS

COMMAND	DEFAULT SCHEDULER	SIMPLE SCHEDULER	PRIORITY-SJF SCHEDULER
stressfs	35	25	45
ls	20	19	23
cat README	6	8	9
uniq test.txt	3	3	4
find . -name README	7	7	7
Cat README | uniq	42	42	41

TOTAL TICKS to execute the same task(simple_scheduler_test) in different schedulers

DEFAULT SCHEDULER	SJF SCHEDULER	PRIORITY-SJF SCHEDULER
6107	5372	5947

COMMENTS/OBSERVATIONS

1. Total Turnaround time for processes in the round robin is high when compared to simple and priority schedulers as all the processes share timeslice.
2. Observed order of execution times **SJF <= PRIORITY-SJF <= DEFAULT** for most of the complex processes.
3. Simple processes like uniq, find etc, have almost no difference in execution times.
4. Complex processes are executed faster using Simple Scheduler and Priority Scheduler.
5. I observed that the Priority-SJF scheduler has slightly more execution times due to the scheduler's logic, as my implementation involves iterating over high to low priorities and finding the shortest predicted ones among the processes with the highest priority.

GROUP MEMBERS

1. Sai Mukesh Reddy Gutha
2. Sriram Mullapudi
3. Yaswanth Bellamkonda

Same group as Project 1: Yes