TOPIC: Replacing round robin schedaling in XV6 OS with your own schedular.

XV6 :

* XV6 is a simple OS like model developed by MIT Students in 2006 for study purpose of students to understand the working of Operating System.

Schedular:

* Schedular is a program which selects the processes which are readily available to execute based on some strategy for selection.

* They are many kinds of schedulars, popularly,

* FIRST IN FIRST OUT, ie, FIRST COME FIRST SERVE
* SHORTEST JOB FIRST SCHEDULING
* SHORTEST REMAINING TIME SCHEDULING
* NON - PREEMPTIVE PRIORITY BASED SCHEDULING
* PREEMPTIVE PRIORITY BASED SCHEDULING
* ROUND ROBIN.

BASICALLY, XV6 is built-in with "Round Robin scheduling".

Round Robin Scheduling:

* In this scheduling, each process is given with a time quantum (time period) to execute its job on CPU.

* Basically, it gives equality to all process in the READY QUEUE to execute it's process on CPU.

Advantages:

* Every process in the READY QUEUE gets a chance to execute its operation on CPU, hence No STARVATION OF PROCESS.

Dis-advantages: Of

* IF Time Quantum ↓ ⇒ undergoes more context switching

* If Time Quantum ↑ ⇒ acts as FIRST COME FIRST SERVE

SCHEDULAR CHOSE TO REPLACE } : PRE-EMPTIVE PRIORITY SCHEDULING

PRE-EMPTIVE PRIORITY SCHEDULING :

* In this schedular, each process gets it's chance to operate on CPU based on it's priority value.

* But if any process comes with ~~the~~ higher priority than the running process on CPU, the process with higher priority gets the change to run on CPU (Pre-emptive). Hence Context switching happens, even when a process is running.

Why do we chose this priority ?

* We thought that, giving equal priority to each process is unnecessary, when some process need immediate action whereas some can be delayed.

* Gone with an idea, "EQUALITY IS BETTER THAN JUSTICE"

Advantages :

* As it is mentioned, the (importance) chance to run on CPU is given based on the importance of the process, not based on FIRST COME or SHORTEST JOB which gives the ~~pro~~ schedular a major upper hand on real-world application.

DIS-ADVANTAGE:

* STARVATION OF low priority process -

Work:

*We also thought that, we could clear this dis-advantage by implementing 'ageing' for the process available for execution. but when we tried to increment the running time for process, it takes the function to execute and called for each ~~second~~ clock tick,

* Being an small process, it takes, so, many errors while we implementing on calculating run time, but ~~soon~~ we tigedit. when we keep on working on it, we will clear it.

* We have modified the XV6 schedular to priority for which the code description is on the following page.

MODIFICATION IN XV6

PREREQUISITES

- Installing the **QEMU Emulator**.

Problem Faced :

It couldn't find the path to QEMU emulator.

Solution:

Manually added the path in the MAKEFILE.

```
	echo "*** Is the directory with i386-jos-elf-gcc in your PATH?" 1>&2; \
	echo "*** If your i386-*-elf toolchain is installed with a command" 1>&2; \
	echo "*** prefix other than 'i386-jos-elf-', set your TOOLPREFIX" 1>&2; \
	echo "*** environment variable to that prefix and run 'make' again." 1>&2; \
	echo "*** To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \
	echo "***" 1>&2; exit 1; fi)
endif

# If the makefile can't find QEMU, specify its path here
  QEMU = /usr/bin/qemu-system-i386

# Try to infer the correct QEMU
```

Referred:

- To install QEMU emulator:
  https://zoomadmin.com/HowToInstall/UbuntuPackage/qemu-system-i386
- To Solve the path Problem:
  https://www.cse.iitd.ernet.in/~sbansal/os/previous_years/2014/lec/l3-hw1.html

IMPLEMENTING PS SYSTEM CALL

PS : Displays the current process – Name, Pid, States , Priority

1. **Adding the priority** (as a datatype) **in the process table:**

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority                 // Priority (0-15)
};
```

2. Defining the cps() in proc.c :

```
-
int
cps()
{
	struct proc *p;

	sti();                                          //Enabling interrupts

	acquire(&ptable.lock);                          //Acquired Lock

	cprintf("name \t pid \t state \t \t priority \n ");
	for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){      //Printing Processes of Process table
	    if( p->state == SLEEPING )
	        cprintf("%s \t %d  \t SLEEPING \t %d \n ",p->name , p->pid , p->priority);
	    else if( p->state == RUNNING )
	        cprintf("%s \t %d  \t RUNNING \t %d \n ",p->name , p->pid , p->priority);
	     if( p->state == RUNNABLE )
	        cprintf("%s \t %d  \t RUNNABLE \t %d \n ",p->name , p->pid , p->priority);
	}

	release(&ptable.lock);                          //Released Lock

	return 22;                                      //Returning the System Call Default No.
}
```

This cps() –

- Iterates through the processes in process table
- Prints it's name, pid, states, priority.

3. Creating ps.c file to call cps() :

Open ps.c ~/Desktop/Modified_files Save

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(void)
{
	cps();

	exit();
}
```

4. Creating sys_cps() to call cps() in sysproc.c:

```
int
sys_cps (void)                         // Defining Cps as a System call
{
	return cps();
}
```

IMPLEMENTING CPRO SYSTEM CALL :

cpro : System call that creates N number of child processes.

1. **Creating cpro.c file to create child process:**

Open cpro.c ~/Documents/xv6-public-master2 Save

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
  int pid;
  int k, n;
  int x, z;

  if (argc != 2) {
    printf(2, "usage: %s n\n", argv[0]);            // default value
  }

  n = atoi(argv[1]);                                                    // Input from User - No. Of
Process

  for ( k = 0; k < n; k++ ) {
    pid = fork ();

    if ( pid < 0 ) {                                // Failed in creating a process

      printf(1, "%d failed in fork!\n", getpid());
      exit();
    }
    else if (pid == 0) {                            // child process

      printf(1, "Child %d created\n",getpid());

      for ( z = 0; z < 10000.0; z += 0.01 )         // Making the child process as CPU Bound
         x =  x + 1.23 * 45.67;
      exit();
    }
  }

  for (k = 0; k < n; k++) {                                             // parent process
    wait();
  }

  exit();
}
```

This program –

- Gets number of child process as input from command line(QEMU).
- Creates the child processes as input.
- Each child process does arithmetic operation, so that the child process is CPU bound.
- Also, Parent process wait until the child completes.

2. **Defining the default value of priority in allocproc() in proc.c:**

```
*proc.c
~/Desktop/xv6-public-master
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 5;                    //Default Priority for all Process

  release(&ptable.lock);
```

IMPLEMENTING CHPR SYSTEM CALL :

chpr : System call that changes the priority of the .

1. **Defining chpr() in proc.c:**

```
//Change Priority
int
chpr( int pid, int priority )
{
        struct proc *p;

        acquire(&ptable.lock);                              //Acquired Lock

        for( p = ptable.proc; p < &ptable.proc[NPROC]; p++ ){
          if( p -> pid == pid ){
            p -> priority = priority;                      //Changing the corresponding priority
            break;
          }
        }

        release(&ptable.lock);                              //Released Lock

        return pid;
}
```

This chpr() –

- Iterates through the processes in process table
- Checks if the input pid matches with any one of the process , if so changes its priority to given input priority.

DECLARING THE CPS() AND CHPR() :

Whenever new functions are created in **proc.c ,** there are certain protocols where the new function should be declared , so that will available it for usage.

- **Declaring in defs.h:**

defs.h
~/Desktop/xv6-public-master

```c
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             cps(void);
int             chpr(int pid,int priority);

// swtch.S
void            swtch(struct context**, struct context*);

// spinlock.c
void            acquire(struct spinlock*);
void            getcallerpcs(void*, uint*);
```

- **Declaring in user.h:**

user.h
~/Desktop/xv6-public-master

```c
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int cps(void);
int chpr(int pid,int priority);

// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *memmove(void*, const void*, int);
```

- **Declaring as a system call in usys.S:**

  - *.S files -> Assembly language file. Hence it can communicate directly to hardware .
  - Here **eax** is the register to store the system calls.

usys.S
~/Desktop/xv6-public-master

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(cps)
SYSCALL(chpr)
```

- **Declaring as a system call in syscall.h:**

syscall.h
~/Desktop/xv6-public-master

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_cps    22
#define SYS_chpr   23
```

- **Declaring as sys_cps (), sys_chpr() in syscall.c:**

Open Save syscall.c ~/Desktop/xv6-public-master

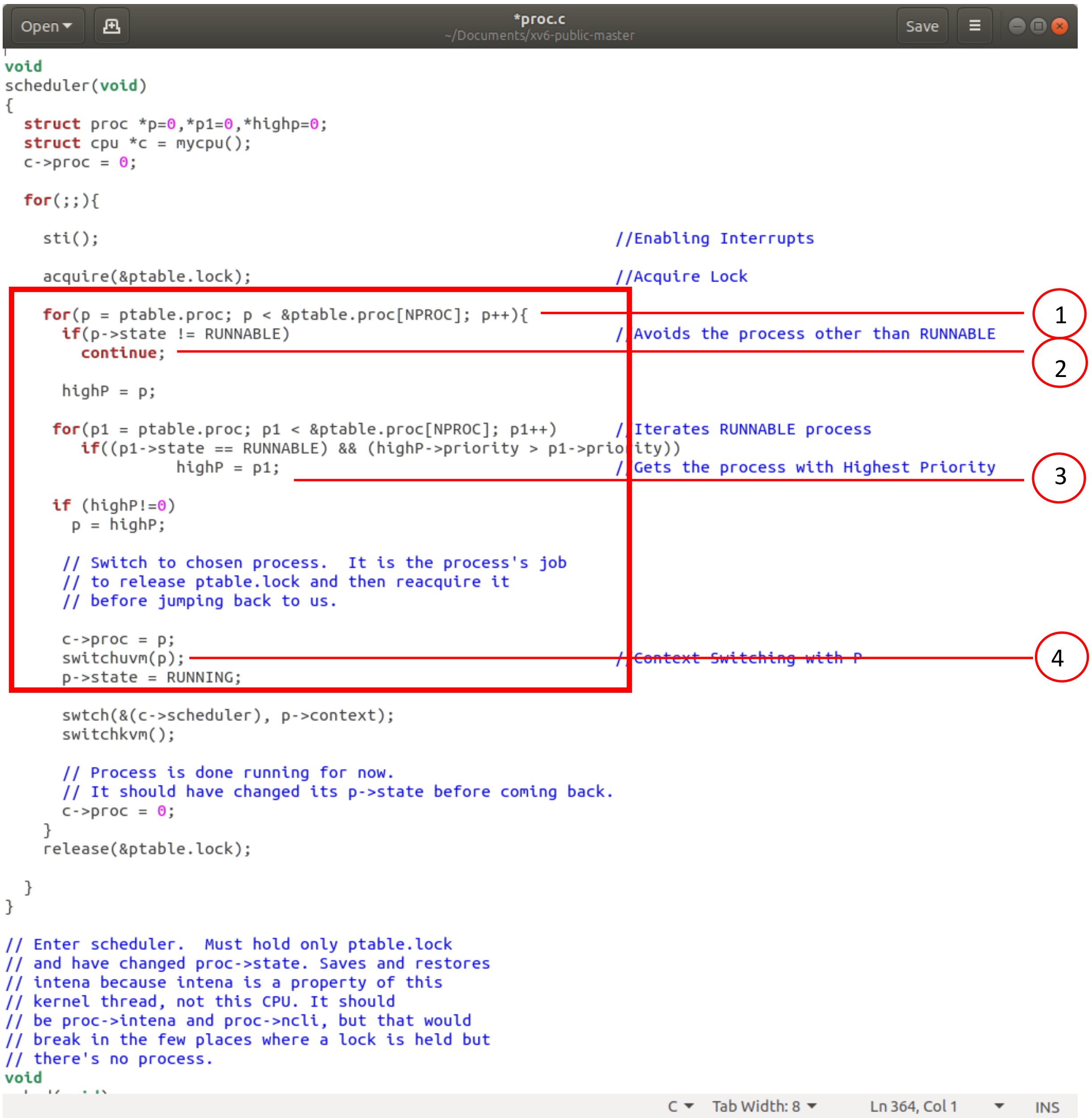
```
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}

extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_cps(void);
extern int sys_chpr(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_cps]     sys_cps,
[SYS_chpr]    sys_chpr,
};
```

IMPLEMENTING THE PRIORITY SCHEDULING:

To implement priority scheduling , the code has to be modified in scheduler() in proc.c file.



```
void
scheduler(void)
{
  struct proc *p=0,*p1=0,*highp=0;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){

    sti();                                                  //Enabling Interrupts

    acquire(&ptable.lock);                                  //Acquire Lock

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)                              //Avoids the process other than RUNNABLE
        continue;

      highP = p;

     for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)  //Iterates RUNNABLE process
        if((p1->state == RUNNABLE) && (highP->priority > p1->priority))
                  highP = p1;                               //Gets the process with Highest Priority

     if (highP!=0)
       p = highP;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      c->proc = p;
      switchuvm(p);                                         //Context Switching with P
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}

// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void
```


This scheduler() –

1. It iterates the processes in process table
2. Avoids non-runnable process.
3. highP gets the highest priority process among the runnable process.
4. Then, the high priority process is context switched with the running process.

ADDING PS, CPRO AND CHPR IN MAKEFILE

- when we make the xv6 files , Makefile in xv6 is executed and kept active - on to work on xv6.
- Hence it is needed to add the newly created in ps , cpro and chpr in Makefile.

➢ Adding the commands :

Open | Makefile ~/Desktop/xv6-public-master | Save

```
_forktest: forktest.o $(ULIB)
	# forktest has less library code linked in - needs to be small
	# in order to be able to max out the proc table.
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
	$(OBJDUMP) -S _forktest > forktest.asm

mkfs: mkfs.c fs.h
	gcc -Werror -Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_ps\
	_cpro\
	_chpr\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)

-include *.d
```

➢ Adding the corresponding c-files for the commands:

```
.gdbinit: .gdbinit.tmpl
	sed "s/localhost:1234/localhost:$(GDBPORT)/" < $^ > $@

qemu-gdb: fs.img xv6.img .gdbinit
	@echo "*** Now run 'gdb'." 1>&2
	$(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)

qemu-nox-gdb: fs.img xv6.img .gdbinit
	@echo "*** Now run 'gdb'." 1>&2
	$(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)

# CUT HERE
# prepare dist for students
# after running make dist, probably want to
# rename it to rev0 or rev1 or so on and then
# check in that version.

EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
	printf.c umalloc.c ps.c cpro.c chpr.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

Referred:

- To implement ps system call :
  Tutorial from VARSHA JENNI- https://www.youtube.com/watch?v=84OksVCw0AU
- To implement cpro and chpr system calls:
  Tutorial from FOO SO - https://www.youtube.com/watch?v=hIXRrv-cBA4
  Tutorial from VARSHA JENNI- https://www.youtube.com/watch?v=JbdDRdindbg