ASSIGNMENT -1 OS CS 343, CS 344

SOME MUST KNOW THINGS ABOUT XV6(RISCV)

1) Every time you make some changes you have to rebuild xv6 { Make clean and Make qemu }
2) You cannot write the kernel commands in the kernel like you do in linux, so you have to make a user program and use system call in that. For example getpid() for the first assignment.
3) If you add something in the user folder then it should be added in UPROGS in Makefile and you have to build it again.
4) If you make any changes in the kernel files you need not to update Makefile, the kernel automatically starts behaving differently.

1.1 ASSIGN PRIME PIDS TO ALL THE PROCESS

STEPS:

1) In the kernel go inside proc.c there you see the initialization of nextpid like int nextpid = … here you can make changes like =2 for primes.
2) In allocproc() function update some lines like
   p->pid = next_prime(nextpid)
   Nextpid = p->pid+1; (this ensures that next prime number is allocated to next process)
3) Write a helper function next_prime(int n) and write it above allocproc().
4) This is the program for checkpid.c in user

—----------

```
#include "kernel/types.h"
#include "user/user.h"

int main(){
      int pid = getpid();
      printf("My PID is: %d\n", pid);
      exit(0);
}
```

—----------

NOTE:

1) *{ PID = 1 First user process created inside shell*
   PID = 2 sh the shell -> spawned by init.
   These process starts automatically and are essential for system to run }

2) *User process runs in user space but are still manages by kernel via the proc[] table.*

One problem You might face is process may fail after pid > 23 as all space is occupied and we are not using freed PIDs after exit()
Why this problem?
Xv6 uses a fixed size table proc[NPROC] usually NPROC = 64 , if you want you can increase it.
-> the size of NPROC can be increased in kernel/param.h to 128 or 256…

Extra - adding testfork() to fork a process

Write a testfork.c program in user

—----------

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main() {
  int pid = fork();
  if (pid == 0) {
    // what fork returns is  important if returns 0-> child process , if >0 -> parent process, if -1 -> fork failed
    printf("Child process, PID = %d\n", getpid());
    exit(0);
  } else {
    // Parent process
    wait(0);
    printf("Parent Process, PID = %d\n", getpid());
    exit(0);
  }
}
```

If pid = 0 runs for child else runs for parent
The wait and exit is so that the parent process should not be completed while its child is still running so it waits for it.

1.2 Writing a top program

STEPS TO MAKE A COMMAND AS A SYSTEM CALL

1) *Declare the system call prototype*
   Edit kernel/syscall.c here you can declare new system call functions prototype.
   extern int sys_ABC(void);
2) *Add systemcall to the syscall table*
   In kernel/syscall.c find the syscalls array and add an entry like
   [SYS_ABC] sys_ABC,

3) *Assign syscall number*
   Edit kernel/syscall.h and add a unique syscall number
   #define SYS_ABC xx;

4) *Implement the systemcall*
   In kernel/sysproc.c -> add a function
   Uint64
   sys_ABC(void)
   { // functionality of the command and return also as it is int }

5) *Expose to user space*
   In user/user.h int ABC(void) // this is declaration

6) *Then implement it in user/usys.S*
   SYSCALL(ABC)

7) *In the user directory now create a file eg rishabh.c and add your syscall command inside that, inside main use ABC() // use it according to the functionality you have designed it.*

8) *Now just rebuild it*

STEPS:

1) Created a file in kernel/process_info.h

(It is not necessary as you can define its content everywhere you use it but I want to write it at one place so I can access it anywhere by just including in the header)

```
struct process_info {
  int pid;
```

```
  int state;
  int ticks;
  char name[16];
};
```

I used this header in sysproc.c in the kernel and in top.c that I created in the user folder.

2) In **sysproc.c** there are many uint64 sys_.. functions so add one **uint64 sys_top** function where you will write the function to implement the system call.

3) **Update syscall mappings**
   In syscall.c you have to define the SYS_top
   -> goto the bottom and add #define SYS_top "next available number", and also to the extern uint64 sys_top(void)

4) In user/user.h
   -> define the top like int top(struct process_info *);

5) In user/usys.S
   -> add SYSCALL(top) // not required it is done already in rebuild

The thing is time is not tracked here so we have to define it

6) In kernel/proc.h add int rtime; //runtime (under the struct proc)

7) In proc.c under the allocproc() function initialise it p->rtime = 0;

8) Then write a program in user/top.c (You have to create the top.c)

9) Add in UPROGS in Makefile
   THEN
   Make clean
   Make qemu

Then when you write top in the terminal it will show the processes.
First and second are init and sh (shell) and the third is the top process that you called

Some Changes to make : (I was using aggdr as int , by default it is void)
In syscall.c change void agddr to int
In defs.h also scroll down to syscall.c and change agddr to int