OS Lab No. 1

Date: 04 / 08 / 2025

Team Members:

1. Kashyap Pansuriya - 230123079
2. Gopal Singh - 230123083
3. Aditya Arun Gupta - 230123078
4. Nilarnab Sutradhar - 230123041

Task 1: Prime PID Assign

Implement the feature to assign prime number PIDs to newly created processes in xv6-riscv.

Snippet 1: kernel/proc.c

```
// Allocate only prime pid to processes
static int
isprime(int n)
{
  if(n < 2) return 0;
  if(n % 2 == 0) return n == 2;
  for(int d = 3; d*d <= n; d += 2)
    if(n % d == 0)
      return 0;
  return 1;
}

static int
nextprime(int start)
{
  int p = start;
  while(!isprime(p)) {
    p++;
  }
  return p;
}

int
allocpid()
{
  int pid;

  acquire(&pid_lock);
  pid = nextprime(nextpid);
  nextpid = pid + 1;
  release(&pid_lock);

  return pid;
}
```

- `isprime()` and `nextprime()` to determine prime numbers for PID allocation.
- Modified `allocpid()` so it skips non-prime PIDs.

Snippet 2: `user/primepidTest.c`

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

#define N 10000

int
main(void)
{
  for(int i = 0; i < N; i++){
    int pid = fork();
    if(pid < 0){
      printf("fork failed\n");
      exit(1);
    }
    if(pid == 0){
      printf("child %d: my PID %d\n", i+1, getpid());
      exit(0);
    }
    else {
        wait(0);
    }
  }

  exit(0);
}
```

- Test to verify newly spawned processes receive prime number PIDs.
- Test can be invoked from xv6 terminal using **primepidTest** command.

Task 2: Implement `top` Command

Expose process information via a new syscall and build a user-space `top` command.

Snippet 1: `kernel/proc.h`

```c
#include "procinfo.h"
...

int getprocinfo(struct procinfo *info);
...
```

Snippet 2: `kernel/proc.c`

```c
int getprocinfo(struct procinfo *info)
{
  struct proc *p;
  int idx = 0;

  for (int i = 0; i < NPROC; i++)
  {
    p = &proc[i];
    acquire(&p->lock);

    if (p->state != UNUSED)
    {
      info[idx].pid = p->pid;
      info[idx].state = p->state;
      info[idx].ticks = p->ticks;

      safestrcpy(info[idx].name, p->name, sizeof(info[idx].name));
      idx++;
    }

    release(&p->lock);
  }

  return idx;
}
```

- Defines helper function to get process information.
- **getprocinfo** function fills up struct procinfo which stores pid, state, ticks and name of the process

Snippet 3: `kernel/trap.c`

```c
...
struct spinlock tickslock;
uint ticks;
...

void
trapinit(void)
{
  initlock(&tickslock, "time");
}

void
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }

  w_stimecmp(r_time() + 1000000);
}

int
devintr()
{
...
if(myproc() != 0 && myproc()->state == RUNNING) myproc()->ticks++;
...
}
```

- Set up a 'ticks' counter protected by **tickslock** in **trapinit()** so we can safely track timer ticks.
- In **clockintr()**, we grab the lock, bump the global 'ticks', wake any waiting processes, and release the lock, then schedule the next interrupt.
- After each timer interrupt, if a user process is running, we do myproc()-ticks++ in **usertrap()** to count one CPU-time tick for that process.
- Together, these changes let us keep both a system-wide tick count and individual process tick counts for the **top** command

Snippet 4: `kernel/procinfo.h`

```c
#pragma once
#include "kernel/types.h"

#ifndef STATE_DEFINE
#define STATE_DEFINE
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
#endif

#ifndef PROCINFO
#define PROCINFO
struct procinfo
{
  int pid;
  enum procstate state;
  uint ticks;
  char name[16];
};
#endif
```

Snippet 5: `kernel/sysproc.c`

```c
...
extern struct proc proc[NPROC];
...


uint64
sys_getprocinfo(void)
{
  uint64 addr;
  struct procinfo info[NPROC];
  int count;

  if (argaddr(0, &addr) < 0)
    return -1;

  count = getprocinfo(info);

  if (copyout(myproc()->pagetable, addr, (char *)info, sizeof(info)) < 0)
    return -1;

  return count;
}
```

- Creating a inteface to be able to use getprocsinfo as a system call from user-space

Snippet 6: `kernel/syscall.c` and `kernel/syscall.h`

```
...
extern uint64 sys_getprocinfo(void);
...


static uint64 (*syscalls[])(void) = {
...
[SYS_getprocinfo] sys_getprocinfo,
  ...
};
```

Snippet 7: `user/user.h`

```
...
int getprocinfo(void *);
...
```

Snippet 8: `user/usys.pl`

```
...
entry("getprocinfo");
...
```

- Some necessary changes in order to define a new custom system call

Snippet 9: `kernel/top.c`

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

#define NPROC 64

void print_procs(){
    struct procinfo arr[NPROC];
    int n = getprocinfo(arr);

    if (n < 0)
    {
        printf("Failed to retrieve process info\n");
        exit(1);
    }


    printf("PID \t STATE \t\t TICKS \t NAME\n");
    printf("-----------------------------------------\n");

    for (int i = 0; i < n; i++)
    {
        char *state;

        switch (arr[i].state)
        {
            case UNUSED:   continue; // Skip UNUSED
            case USED:     state = "USED"; break;
            case SLEEPING: state = "SLEEPING"; break;
            case RUNNABLE: state = "RUNNABLE"; break;
            case RUNNING:  state = "RUNNING"; break;
            case ZOMBIE:   state = "ZOMBIE"; break;
            default:       continue; // Skip unknown
        }

        printf("%d \t %s \t %d \t %s\n", arr[i].pid, state, arr[i].ticks, arr[i].name);
    }

    printf("\n");
}
```

- Print function to print the formatted table of processes as per requirement

Conclusion and References

We implemented the Prime PID assignment and a user-level `top` utility in xv6-riscv, adding kernel data structures, syscalls, and user-space support. The tests demonstrate correct PID assignment, tick accounting, and process-table rendering.

References:

- xv6-riscv source code (MIT)
- Lab assignment description