Mini-Project: Evaluating xv6 (2012) Default Scheduler with Minimal Benchmarks

Project Overview

This mini-project evaluates the **default xv6 (2012) Round-Robin scheduler** using a **minimal benchmarking suite** and a focused **performance analysis**. You will:

- Implement a lightweight **`ps`** command (via syscall).
- Implement a **`getpinfo`** syscall to extract per-process stats.
- Write simple **workloads** (CPU-bound, I/O-bound, mixed) and a **benchmark harness**.
- Measure three metrics: **Response Time**, **Turnaround Time**, **CPU Utilization**.

Learning Objectives

- Understand and compute **Response Time**, **Turnaround Time**, and **CPU Utilization** in an OS scheduler.
- Gain hands-on experience reading/modifying xv6 kernel code and wiring **syscalls**.
- Design a small, reproducible benchmark to reason about scheduler behavior.

What You Will Deliver

1. **Modified xv6** with:
    - `ps` command (syscall + simple user test).
    - `getpinfo` syscall + per-process stats in `struct proc`.
    - Minimal workloads and a benchmark harness.
2. **Short report** (1–2 pages) with the three metrics and a brief discussion of observations.

Part A — Basic Infrastructure (Foundation)

You'll add a **`ps`** syscall and a **`getpinfo`** syscall, and extend `struct proc` with simple counters/timestamps.

> Throughout, assume the classic xv6 (2012) file layout:
> `defs.h, param.h, proc.h, syscall.h, syscall.c, sysproc.c,`

> `usys.S, user.h, proc.c, exit() in proc.c`, etc.

A.0 Create a Shared Header for Stats

Create a header that both kernel and user code can include. Easiest is to add it in the **xv6 root** as `pstat.h`.

`pstat.h` (new file)

```
# ifndef _PSTAT_H_
# define _PSTAT_H_
# include "param.h"

struct pstat {
  int inuse[NPROC];       // 1 if this slot is in use
  int pid[NPROC];         // PID
  int ticks[NPROC];       // # times scheduled (proxy for CPU time)
  int wait_ticks[NPROC];  // total time spent RUNNABLE
  int start_tick[NPROC];  // creation time (ticks)
  int first_run[NPROC];   // first scheduled time (ticks), -1 if never
  int end_tick[NPROC];    // completion time (ticks), else 0
  char name[NPROC][16];   // process name
};

# endif
```

A.1 `ps` Command (Kernel function + Syscall + User test)

`proc.c` — add kernel function `ps()`

```
void
ps(void)
{
  struct proc *p;

  cprintf("PID\tState\t\tName\tSize\tParent\n");
  cprintf("---\t-----\t\t----\t----\t------\n");

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED) continue;

    cprintf("%d\t", p->pid);
    if(p->state == SLEEPING)      cprintf("sleeping\t");
    else if(p->state == RUNNABLE) cprintf("runnable\t");
    else if(p->state == RUNNING)  cprintf("running\t\t");
    else if(p->state == ZOMBIE)   cprintf("zombie\t\t");
    else                          cprintf("???\t\t");

    cprintf("%s\t%d\t", p->name, p->sz);
    if(p->parent) cprintf("%d\n", p->parent->pid);
    else          cprintf("-\n");
  }
  release(&ptable.lock);
}
```

defs.h

```
void ps(void);
```

sysproc.c

```
int
sys_ps(void)
{
  ps();
  return 0;
}
```

syscall.h

```
# define SYS_ps 22
```

syscall.c

```
extern int sys_ps(void);
[SYS_ps] sys_ps,
```

user.h

```
int ps(void);
```

usys.S

```
SYSCALL(ps)
```

`user/pstest.c`

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

int
main(int argc, char *argv[])
{
  printf(1, "Process Status:\n");
  ps();
  exit();
}
```

A.2 `getpinfo` Syscall + Extend `struct proc`

`proc.h` — extend `struct proc`

```
int numTicks;
int wait_ticks;
int creation_time;
int first_run_time;
int completion_time;
```

`allocproc()` initialization in `proc.c`

```
p->numTicks = 0;
p->wait_ticks = 0;
p->creation_time = ticks;
p->first_run_time = -1;
p->completion_time = 0;
```

`scheduler()` update in `proc.c`

```
for(struct proc *q = ptable.proc; q < &ptable.proc[NPROC]; q++){
  if(q->state == RUNNABLE) q->wait_ticks++;
}
```

Mark first run

```
if(p->first_run_time < 0)
  p->first_run_time = ticks;
p->numTicks++;
```

`exit()` completion time

```
curproc->completion_time = ticks;
```

Kernel function `getpinfo()`

```
# include "pstat.h"

int
getpinfo(struct pstat *ps)
{
  if(ps == 0) return -1;
  acquire(&ptable.lock);
  struct proc *p;
  int i = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++, i++){
    if(p->state == UNUSED){
      ps->inuse[i] = 0;
      continue;
    }
    ps->inuse[i] = 1;
    ps->pid[i] = p->pid;
    ps->ticks[i] = p->numTicks;
    ps->wait_ticks[i] = p->wait_ticks;
    ps->start_tick[i] = p->creation_time;
    ps->first_run[i] = p->first_run_time;
    ps->end_tick[i] = p->completion_time;
    int j=0; for(; j<16 && p->name[j]; j++) ps->name[i][j] = p->name[j];
    ps->name[i][j] = 0;
  }
  release(&ptable.lock);
  return 0;
}
```

Add to `defs.h`

```
int getpinfo(struct pstat *);
```

`sysproc.c` wrapper

```
int
sys_getpinfo(void)
{
  struct pstat *ps;
  if(argptr(0, (void*)&ps, sizeof(*ps)) < 0)
    return -1;
  return getpinfo(ps);
}
```

`syscall.h`

```
# define SYS_getpinfo 23
```

`syscall.c`

```
extern int sys_getpinfo(void);
[SYS_getpinfo] sys_getpinfo,
```

`user.h`

```
struct pstat;
int getpinfo(struct pstat *);
```

`usys.S`

```
SYSCALL(getpinfo)
```

Part C — Minimal Benchmarking Suite

C.1 Workloads

`user/cpubound.c`

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

int main(int argc,char*argv[]){
  int iters=800000; if(argc>1) iters=atoi(argv[1]);
  int i,x=0; for(i=0;i<iters;i++){ x+=i; if((i&1023)==0) x>>=1; }
  printf(1,"cpubound done pid=%d\n", getpid());
  exit();
}
```

`user/iobound.c`

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

int main(int argc,char*argv[]){
  int ops=60; if(argc>1) ops=atoi(argv[1]);
  int i; for(i=0;i<ops;i++){ printf(1,"io %d\n",i); sleep(5); }
  printf(1,"iobound done pid=%d\n", getpid());
  exit();
}
```

`user/mixed.c`

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

int main(int argc,char*argv[]){
  int cycles=40; if(argc>1) cycles=atoi(argv[1]);
  int i,j,x=0;
  for(i=0;i<cycles;i++){ for(j=0;j<120000;j++) x+=j; if((i%4)==0) sleep(3); }
  printf(1,"mixed done pid=%d\n", getpid());
  exit();
}
```

C.2 Benchmark Orchestrator

`user/benchmark.c`

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

# define NCHILD 5

```

```
static void spawn(char*prog,char*arg){
  char*argv[3]={prog,arg,0};
  if(fork()==0){ exec(prog,argv); printf(1,"exec %s failed\n",prog); exit(); }
}

int main(int argc,char*argv[]){
  struct pstat ps; int i;
  int start=uptime();
  spawn("cpubound","600000");
  spawn("cpubound","600000");
  spawn("iobound","40");
  spawn("iobound","40");
  spawn("mixed","30");
  for(i=0;i<NCHILD;i++) wait();
  int end=uptime(); int makespan=end-start;
  if(getpinfo(&ps)<0){ printf(1,"getpinfo failed\n"); exit(); }

  int total_resp=0,resp_cnt=0,total_turn=0,turn_cnt=0,busy_ticks=0;
  printf(1,"\nFINAL STATS (pid name ticks wait start first end)\n");
  for(i=0;i<NPROC;i++){ if(ps.inuse[i] && ps.pid[i]>2){
    int turnaround=(ps.end_tick[i]>0?ps.end_tick[i]:end)-ps.start_tick[i];
    int response=(ps.first_run[i]>=0?ps.first_run[i]:end)-ps.start_tick[i];
    printf(1,"%d %-12s %d %d %d %d %d\n",ps.pid[i],ps.name[i],ps.ticks[i],
           ps.wait_ticks[i],ps.start_tick[i],ps.first_run[i],ps.end_tick[i]);
    if(ps.end_tick[i]>0){ total_turn+=turnaround; turn_cnt++; }
    if(ps.first_run[i]>=0){ total_resp+=response; resp_cnt++; }
    busy_ticks+=ps.ticks[i];
  }}
  int avg_resp=(resp_cnt? total_resp/resp_cnt:-1);
  int avg_turn=(turn_cnt? total_turn/turn_cnt:-1);
  int util_permille=(makespan>0?(busy_ticks*1000)/makespan:0);
  printf(1,"\nMAKESPAN (ticks): %d\n",makespan);
  printf(1,"AVG RESPONSE (ticks): %d\n",avg_resp);
  printf(1,"AVG TURNAROUND (ticks): %d\n",avg_turn);
  printf(1,"CPU UTILIZATION: %d.%d%%\n",util_permille/10,util_permille%10);
  exit();
}
```

Part D — Performance Analysis

Metrics:

- **Response Time** = `first_run - start_tick`
- **Turnaround Time** = `end_tick - start_tick`
- **CPU Utilization** = `(sum of ps.ticks[]) / makespan * 100%`

Run:

```
make qemu-nox CPUS=1
benchmark
```

Include results in your report.

Makefile Update

Add new programs to `UPROGS`:

```
UPROGS=...         _pstest        _cpubound        _iobound        _mixed        _ben
```

Rubric

- Correct syscall wiring (`ps`, `getpinfo`): 40%
- Workloads & benchmark suite: 40%
- Report (3 metrics + discussion): 20%