CS344 Assignment-2

Group-3, CSE

August 18, 2025

Team Members

- **230101024** (Baswa Dhanush Sai)
- **230101114** (Yeredla Koushik Reddy)
- **230101092** (Seella Mythresh Babu)
- **230101120** (Taes Padhihary)

Please find the Google Drive folder containing all the edited files here:

drive.google.com/drive/folders/1Ul5WB5L0lYgSJm6dPksa7rLtmEL7pat2?usp=sharing

Contents

[1] Task 2.1: Weighted Round Robin Scheduler

1.1. Problem Description

The default xv6 scheduler treats all processes equally. In this task, modify it to allow processes with higher priority to get more CPU time, while ensuring lower priority processes are not completely starved.

1. Add new system calls to allow setting and getting process priorities:
   - `set_priority(n)`: sets priority of the current process to a positive integer n.
   - `get_priority()`: returns priority of the calling process. MAX value is 1000.
2. Modify the xv6 scheduler to implement a Weighted Round Robin (WRR) algorithm using the priority values as weights:
   - A process with higher priority should get proportionally more CPU time than one with lower priority.
   - However, no process should be starved; all runnable processes must get some CPU time.
   - Set a default priority for all new processes (e.g., 10), unless explicitly changed.

1.2. Implementation in xv6

1. **Per-process add-ons:**
   - `priority` (default: 10, max: 1000) — higher means longer quantum.
   - a per-run counter (`ticks`) that counts how many timer ticks the process has used in its current quantum. Initialized to 0.
2. **Initialization:** New processes start with `priority=DEFAULT_PRIORITY` and begin a fresh quantum whose target length equals `priority`.
3. **Preemption** (main edit)**:** In both `usertrap()` and `kerneltrap()`, on each timer interrupt we:
   (a) increment the running process' `p->ticks`
   (b) if `p->ticks >= p->priority`, reset `p->ticks=0` and `yield()` (end of the time quantum)
   (c) otherwise, **do not** yield (the process continues running across ticks).
4. **System calls:** `setpriority(n)`, `getpriority()` allow user-space to set/read the weight. When setting a new priority we immediately reset the quantum.

1.2.1. `kernel` and `user` changes

- We define `MAX_PRIORITY = 1000` and `DEFAULT_PRIORITY = 10` in `kernel/defs.h`.
- In `kernel/proc.h`, we extend `struct proc` with `priority` (the WRR weight) and `ticks` (the number of ticks this process has consumed in the *current* quantum).
- In `kernel/proc.c`, set `p->priority=DEFAULT_PRIORITY` and `p->ticks=0` in the `allocproc()` and set `p->ticks=0` in `fork()` so it begins with a new quantum.

(Core idea) • In `kernel/trap.c`: xv6 originally *yields* on *every* timer tick — we change that to *yield only when the current process has used* `priority` *ticks* in both `usertrap()` and `kerneltrap()`.

• Add the system calls `int setpriority(int n)` and `int getpriority(void)` as numbers in `kernel/syscall.h`, tables in `kernel/syscall.c`, implementations in `kernel/sysproc.c` and prototypes in `user/user.h` and `user/user.pl`.

1.3. Testing with `task2.1Demo.c`

The following code (output in Figure 1) checks whether Weighted Round Robin is enforced. 'A' has been given priority 500 and hence can run upto 500 ticks uninterrupted. Then 'B' will run till the time quantum expires and will be preempted by 'C' which will run till 100 ticks/remaining time quanta uninterrupted if no higher priority process is present.

Listing 1. user/task2.1Demo.c

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

int priority_fork(int priority, char label) {
    int p = fork();
    if (p == 0) {
      setpriority(priority);

      for (long i = 0; i < 1000000000; i++) {
          if (i % 10000000 == 0) write(1, &label, 1);
      }

      fprintf(1, "\nProcess %d finished\n", priority);
      exit(0);
    }
    return p;
}

int main(void) {

  priority_fork(500, 'A');
  priority_fork(1, 'B');
  priority_fork(100, 'C');

  wait(0);
  wait(0);
  wait(0);

  exit(0);
}
```

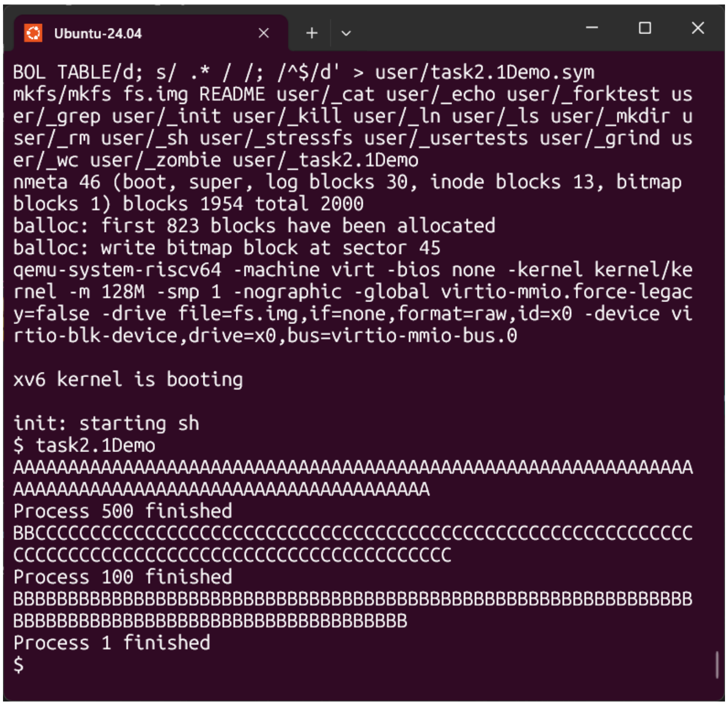

Figure 1. 'A' [500] runs till either 500 ticks or till it is completed (here, the latter happens). 'B' [1] runs till the end of time slice and then gets pre-empted by 'C' [100] which runs till it is finished. Then, process 'B' finishes uninterrupted. Hence, weighted Round Robin is enforced here.

[2] Task 2.2: Custom Scheduling Algorithm

2.1. Problem Description

Move beyond round-robin and weighted round-robin scheduling. Analyse and implement a scheduling algorithm that optimises one or more of the following goals:

- Minimise turnaround or response time
- Improve CPU utilisation
- Reduce waiting time
- Avoid starvation
- Ensure fairness

2.2. Implementation in xv6

We will implement Lottery Scheduling. The repository at github.com/avaiyang/xv6-lottery-scheduling provided information on the steps to be followed to implement the lottery scheduling algorithm.

2.2.1. Design and Tradeoffs: Lottery Scheduling

1. **Per–process state:** each process `p` stores an integer `tickets` (≥ 1). New processes start with `tickets = 10` by default. Children inherit the parent's tickets.

2. **Syscall:** `settickets(int n)` sets the caller's `tickets` to `n` if `n` $\geq$ `1`.

3. **Main scheduling loop (runs forever):**

   - Set `total` to 0. Scan the process table:

     For every `RUNNABLE` process `p`, let `t = max(1, p->tickets)` and add `t` to `total`.
   - If `total == 0`, no process is runnable: execute `wfi` and restart the loop.
   - Draw a winning number `w` by computing `w = rand32() % total`.
   - Set `prefix = 0`. Scan the process table again:
     - For each `RUNNABLE` process `p`, let `t = max(1, p->tickets)`.
     - If `w < prefix + t`, this process owns the winning ticket:
       * Set `p->state = RUNNING`, set `c->proc = p`, and perform `swtch` to `p`.
       * When `swtch` returns, clear `c->proc` and break out to start a new scheduling decision.
     - Otherwise, increase `prefix` by `t` and continue scanning.

4. **PRNG used by the scheduler (`rand32()`):**

   - Maintain a 64-bit nonzero global (or per-CPU) state `state`.
   - On each call: `state = state * 6364136223846793005ULL + 1ULL` (wraps mod 2^{64}). Return the upper 31 bits using `(uint)(state >> 33)`. This is the `winner` lottery value.

5. **Correctness:**

   - Over time the CPU share of process i converges to almost $t_i / \sum t_i$.
   - Any process with at least one ticket has non-zero probability of being chosen at each decision.
   - User space controls weights via settickets (e.g., interactive jobs should ideally get more tickets).

6. **Drawbacks:**

   - Random selection yields bursty service with the proportions stabilizing only over longer windows.
   - Without caps, a process can inflate its tickets and dominate which can lead to a potential abuse of the scheduler, as the user can set the tickets.
   - $O($`NPROC`$)$ per decision is not scalable.
   - Even with high tickets, a process can occasionally experience unlucky gaps; not suitable where strict minimum service guarantees are required.

2.3. Testing with `task2.2Demo.c`

Listing 2. user/task2.2Demo.c

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

// a CPU-bound function
void cpu_work(long iter) {
  for (long i = 0; i < iter; i++) {
    // A volatile variable prevents the compiler from optimizing the
       loop away.
    volatile int x = 1 + 2*3 - 4 / 1 + 91 - 35 * 1729 + 8;
    (void)x; // Suppress "unused variable" warning.
  }
}

// function to test lottery scheduling
int fork_lottery_child(const char *label, int tickets, int dur) {
  int p = fork();
  if (p==0) {
    if (settickets(tickets) < 0) printf("%s: settickets failed\n",
       label);

    uint64 start = uptime(), cnt = 0;
    while(uptime() < start + dur){
      cpu_work(5000);
      cnt++;
    }

    printf("%s tickets=%d cnt=%d\n", label, tickets, (int)cnt); // "
       cnt" will be proportional to the number of "tickets"
    exit(0);
  }
  return p;
}

int
main(void)
{
  int t1 = 10;
  int t2 = 100;
  int dur = 300;

  fork_lottery_child("A", t1, dur);
  fork_lottery_child("B", t2, dur);

  wait(0);
  wait(0);

  write(1, "\n", 1);
  exit(0);
}
```

The output in Figure 2 provides the expected outcome of the Lottery scheduling involving two processes.

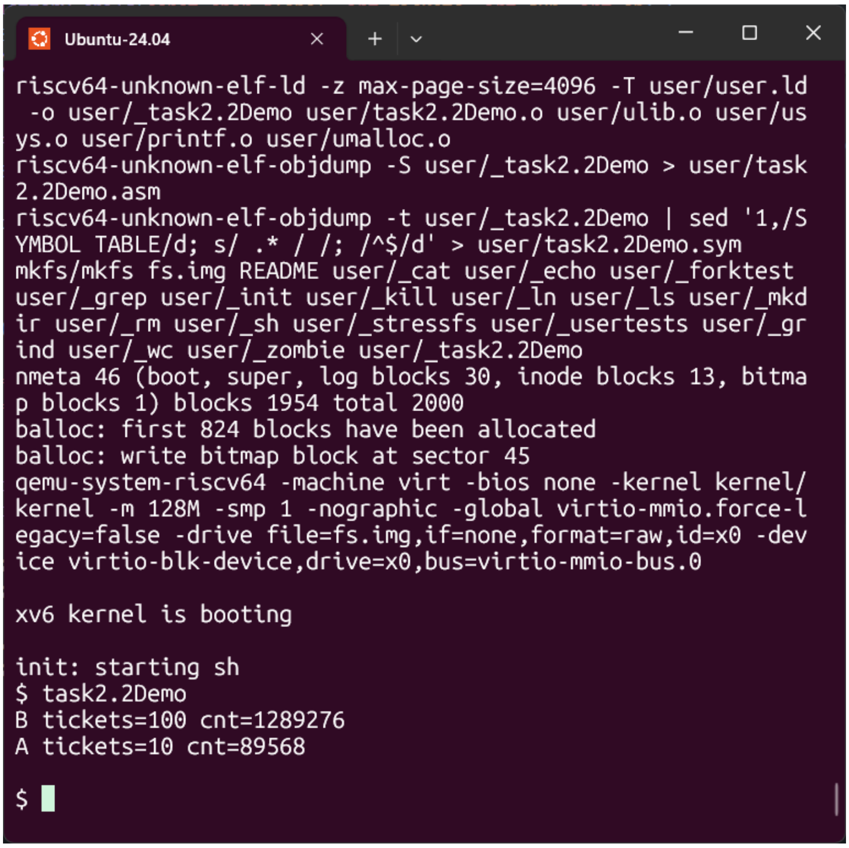

Figure 2. Fork labelled ‘A’ with 10 tickets ran around 89568 times, while fork labelled ‘B’ ran 1289276 ($> 11 \cdot 89568$) times, thus proving the expected lottery scheduling