Implementing a Priority Scheduler

We will modify the present round robin scheduler of xv6 to create a priority scheduler, where each process will have a priority number assigned to it, beginning from 1 (highest priority) till 10 (lowest priority). This lecture explains:

- How the concept of priority can be introduced
- How to assign priorities to different processes from the user space
- How to track the running periods of processes
- How to modify the scheduler to implement a priority scheduling algorithm

1. Why Priority Scheduling?

We can extend xv6 to support **priority-based scheduling**, where:

- Priority **1** = highest
- Priority **10** = lowest
- A process with higher priority runs **before** any lower-priority process

Since we will assume **all processes have unique priorities**, we can keep the algorithm simple.

2. Step 1 — Add Fields to `struct proc`

We add two fields:

1. `priority` — integer from 1 to 10
2. `rounds` — counts how many times the scheduler chose this process

Open `kernel/proc.h` and modify:

```
struct proc {
  // ... existing fields ...

  int priority;   // 1 = highest, 10 = lowest
  int rounds;      // how many times the scheduler has run this
process
};
```

3. Step 2 — Initialize Priority and Turns

Modify `allocproc()` in `kernel/proc.c`:

```
p->priority = 10;   // default lowest priority
p->rounds = 0;
```

Special rule: first process gets priority 1

`userinit()` sets up the very first process (`init`). After it is created, add:

```
p->priority = 1;
```

This ensures xv6 boots properly.

4. Step 3 — Make Children Inherit Parent Priority

Modify `kfork()` in `kernel/proc.c`:

Look for the section:

```
np->priority = p->priority;
np->rounds = 0;
```

This ensures consistency across parent-child relationships.

5. Step 4 — Create a System Call to Set Priority

We need a syscall:

```
int setpriority(int newprio);
```

(1) Declare it in user space

Add to `user/user.h`:

```
int setpriority(int);
```

(2) Add syscall number

In `kernel/syscall.h`:

```
#define SYS_setpriority  22  // choose next number
```

(3) Add entry in syscall table

In `kernel/syscall.c`:

```
extern uint64 sys_setpriority(void);

static uint64 (*syscalls[])(void) = {
    // ...
    [SYS_setpriority] sys_setpriority,
};
```

(4) Implement the syscall wrapper

In `kernel/sysproc.c`:

```
uint64
sys_setpriority(void)
{
  int pr;
  argint(0, &pr);

  if(pr < 1 || pr > 10)
    return -1;

  struct proc *p = myproc();
  acquire(&p->lock);
  p->priority = pr;
  release(&p->lock);
```

```
  return 0;
}
```

Now users can call:

```
setpriority(3);
```

6. Step 5 — Modifying the Scheduler for Priority Scheduling

Now we replace Round Robin with **Strict Priority Scheduling**:

Algorithm

1. Scan all processes to find the **highest-priority RUNNABLE process**
2. Run *only that* one
3. When it yields, exit, or sleeps → repeat from step 1

Key design choice

Unlike the original xv6 RR scheduler, **we will run exactly one process per outer loop iteration**.

This stops the scheduler from accidentally running a lower-priority process before noticing a newly awakened high-priority one.

Replace the scheduler with this logic

Open `kernel/proc.c`, find `scheduler()`, and modify it as:

```
void
scheduler(void)
{
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    intr_on();
    intr_off();

    struct proc *bestproc = 0;
    int bestpriority = 11; // priorities are 0-10
```

```
    for(struct proc *p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE && p->priority < bestpriority) {
        if (bestproc != 0) {
          release(&bestproc->lock);
        }
        bestpriority = p->priority;
        bestproc = p;
      } else {
      release(&p->lock);
      }
    }

    if (bestproc == 0) {
      // nothing to run
      asm volatile("wfi");
      continue;
    }

    bestproc->rounds++;
    c->proc = bestproc;
    bestproc->state = RUNNING;
    swtch(&c->context, &bestproc->context);

    c->proc = 0;
    release(&bestproc->lock);
  }
}
```

What we achieve:

- The scheduler runs exactly **one** highest-priority RUNNABLE process per iteration
- `rounds` increments each time the scheduler chooses a process
- Priority ordering is strictly respected
- New high-priority processes preempt low-priority ones correctly

7. Step 6 — Modify Procdump

Modify procdump so that it shows the priority and the number of turns taken by each process. We can call procdump asynchronously by pressing **Ctrl + P** at any time to check the priority and turns taken by each process.

```
for(p = proc; p < &proc[NPROC]; p++) {
      if(p->state == UNUSED)
      continue;
      if(p->state >= 0 && p->state < NELEM(states) &&
states[p->state])
            state = states[p->state];
      else
            state = "???";
      printf("%d %s %s %d %d", p->pid, state, p->name, p->priority,
p->rounds);
      printf("\n");
}
```

7. Step 7 — Testing the Scheduler

Create a simple user test program named test_priority:

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

int main(int argc, char *argv[]) {
  int priority = atoi(argv[1]);
  setpriority(priority);
  while(1) {}
  return 0;
}
```

Create a test_priority process in the background using

```
$ test_priority 5 &
```

This starts one test_priority process with priority 5. Then, press **Ctrl + P** multiple times to observe the increasing count of ticks. We use the & at the end so that we do not get stuck in the while loop ourselves.

Press Ctrl + P to observe output like:

```
1 sleep  init 1 38
2 sleep  sh 1 18
6 run    test_priority 5 23
```

This tells test_priority has received 23 rounds till now. Press Ctrl + P to observe its round count has increased.

We can start another process with a higher in the background using

```
$ test_priority 4 &
```

Now if we press **Ctrl + P** after some time, we will observe the previous process's round count gets stuck, i.e. it does not get any more rounds in the CPU, as there is a new process with priority 4.

Sample output of Ctrl + P:

```
1 sleep  init 1 39
2 sleep  sh 1 23
6 runble test_priority 5 129
8 run    test_priority 4 14
```

Sample output of Ctrl + P again after some time:

```
1 sleep  init 1 39
2 sleep  sh 1 23
6 runble test_priority 5 129
8 run    test_priority 4 27
```

This shows that the second process with priority 4 is running ahead of the first one with priority 5. Even though init and sh have priorities 1, they are not run as they are in the **sleep** state. We can repeat the experiment by spawning another process with priority 3.