Shahriar Attar - 810100186 Sobhan Alaeddini - 810100188 Matin Bazrafshan - 810100093

Repository Link: https://github.com/Shahriar-0/Operating-System-Lab-Projects-F2024
Latest Commit Hash: f9c495f3390642c0af7404174ab74af361c99b8e

Scheduling in xv6

1. **Why does calling the function sched lead to calling the function scheduler?**

   First, we explain what each of these functions does:

I. `sched()`: This function is responsible for relinquishing the CPU when a process determines that it's no longer runnable (like when it's waiting for I/O, its time slice is over, or it's yielding the CPU willingly). The `sched()` function doesn't choose the next process to run—instead, it sets up the environment so that the CPU can safely transition to run the scheduling algorithm (which will pick the next process). In the code, `sched()` is often the last function a process calls before it wants to stop running. It performs important bookkeeping tasks to save the current state of the process and prepare the system to switch the context to another process.

II. `scheduler()`: This function is the central piece of the scheduling mechanism. It runs perpetually as long as the system runs and is responsible for choosing which process to run next. It does this by looking at all the processes in the run queue and selecting the next one according to the scheduling algorithm implemented

In xv6, the kernel uses a two-stage process for scheduling:

I. **Transitioning from the current process to a safe point:** This is handled by the `sched()` function.

II. **Selecting the next process to run:** This is handled by the `scheduler()` function.

Now let's break down the steps. The kernel initially calls `mpmain()` and At the end of mpmain, the `scheduler()` function is called. This initiates the scheduling process for the kernel. the `sched` function is called on three occasions:

1. The process leaves the processor using the `exit()` system call.
2. The process enters the `SLEEPING` state using the `sleep()` system call.
3. The process is forced to leave the processor after the interrupt generated by the timer, in which case the `yield()` function is called, and the `sched()` function is also called in that function.

after checking various conditions, if all of them are met then it calls `swtch` to switch to the scheduler thread. Any function that calls `sched()` must do so with the `ptable.loc` held. This lock is held all during the context switch. While the `sched()` function handles the transition, the `scheduler()` function is responsible for the actual selection. The `scheduler` function is called by every CPU's scheduler thread at the start and loops in it forever. The scheduler function continues the for loop, finds a process to run, switches to it, and the cycle repeats. Therefore, calling `sched` results in calling the `scheduler` function because First `sched()` prepares the environment for selecting the next process, and second `sched()` ultimately calls `swtch()` to switch to the scheduler thread, where `scheduler()` resides.

This separation between `sched()` and `scheduler()` is a common pattern in operating systems. `sched()` concerns itself with transitioning from a running process to the point where the OS can safely choose another process, while `scheduler()` performs the selection of the next process.

Below you can see the implementation of `scheduler()` and `sched()` :

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

      for(;;) {
      // Enable interrupts on this processor.
      sti();
      // Loop over process table looking for process to run.
      acquire(&ptable.lock);
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->state != RUNNABLE)
                  continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
      }
      release(&ptable.lock);
      }
}
```

```
void sched(void) {
	int intena;
	struct proc *p = myproc();

	if(!holding(&ptable.lock))
	panic("sched ptable.lock");
	if(mycpu()->ncli != 1)
	panic("sched locks");
	if(p->state == RUNNING)
	panic("sched running");
	if(readeflags()&FL_IF)
	panic("sched interruptible");

	intena = mycpu()->intena;
	swtch(&p->context, mycpu()->scheduler);
	mycpu()->intena = intena;
}

// Give up the CPU for one scheduling round.
void yield(void) {
	acquire(&ptable.lock);
	myproc()->state = RUNNABLE;
	sched();
	release(&ptable.lock);
}
```

Scheduling

2. What is the data structure used to represent the execution queue in the CFS[1] used by the Linux kernel?

The CFS in the Linux kernel does not use a traditional queue to represent the execution queue. Instead, it utilizes a red-black tree, also known as a `rbtree`, to maintain a time-ordered list of runnable processes. Advantages over a traditional queue:

I. **Time-ordered:** Processes are sorted based on their virtual runtime, ensuring fairness and preventing starvation.
II. **Efficient insertions and deletions:** Red-black trees maintain their balance during insertions and deletions, offering a logarithmic time complexity for these operations.
III. **Fast lookup:** Finding the next runnable process involves only searching for the leftmost node in the tree.

Each node in the rbtree represents a sched_entity structure, which contains information about a specific process, including its `vruntime`[2]. The leftmost node in the tree always holds the `sched_entity` with the least amount of runtime, making it the next process to be scheduled. As processes run, their `vruntime` increases, causing them to move further down the tree, allowing other processes with lower `vruntime` to be scheduled. New processes are inserted into the tree based on their `vruntime`, ensuring their proper placement in the execution order.

This **time-ordered** structure ensures that all processes receive fair CPU time over time, preventing any process from being starved indefinitely. It also allows CFS to efficiently manage and schedule processes on multi-core systems.

[1] Completely Fair Scheduler
[2] virtual runtime

3. Compare these two operating systems from the point of view of scheduling queues.

xv6 uses a unique approach to manage process queues and state transitions. Unlike most operating systems, it doesn't maintain separate "ready queue" and "running queue" data structures. Instead, it uses a single list called `ptable` to track all processes in the system, regardless of their state.

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

Each process in the `ptable` has a corresponding state variable that indicates whether it is:

I. **Runnable (or Ready):** Ready to run but not currently assigned to the CPU.
II. **Running:** Currently executing on the CPU.
III. **Sleeping:** Waiting for an I/O operation or another event.
IV. **Zombie:** Finished execution but resources not yet released.

Process transitions between these states are managed as follows:

I. **Runnable to Running:** The scheduler selects a runnable process from the `ptable` and changes its state to running.
II. **Running to Sleeping:** A process calls the `sleep()` function when it requires an I/O operation or event, transitioning its state to sleeping and removing it from scheduling.
III. **Sleeping to Runnable:** When the awaited event occurs, the process is awakened, changing its state back to runnable and adding it back to the scheduling pool.
IV. **Running to Exit:** When a process finishes execution or calls `exit()`, it transitions to the zombie state, indicating it's finished but not yet cleaned up.
V. **Zombie to Free:** The parent process calls the wait function to collect resources from child processes in the zombie state, freeing them and removing them from the system.

This approach has a few advantages:

I. **Simplicity:** Eliminates the need for additional data structures, simplifying the overall design.
II. **Efficiency:** Checking for runnable processes becomes faster since the state is readily available in the `ptable`.

But also there exist a few disadvantages as well:

I. **Scalability:** Iterating through the entire ptable for scheduling may become inefficient for large numbers of processes.
II. **Fairness:** xv6's round-robin scheduling algorithm may not guarantee perfect fairness in all situations.
III. **Vulnerability:** In this approach when accessing `ptable.proc` we first need to `acquire` the `ptable.lock` and then release. If not implemented properly it can be used by malware.
IV. **Cache management:** Since every process has its high-level cache the main cache would be less efficient and useful.

	Scheduling Algorithm	Scalability	Complexity
xv6	**Uses a simple round-robin scheduling algorithm. Each runnable process receives a slice of CPU time in turn. May not guarantee perfect fairness under all circumstances.**	**Iterating through the entire process table for scheduling can become inefficient for large numbers of processes.**	**Simple and easy to understand. Suitable for educational purposes and small systems.**
CFS (Linux)	**Utilizes the Completely Fair Scheduler (CFS) algorithm. Assigns virtual runtime to each process based on its actual CPU time. Aims to provide all processes with equal access to CPU resources over time. More complex but ensures fairness and prevents starvation.**	**Red-black tree structure efficiently handles insertion and deletions, making it scalable for large numbers of processes.**	**More complex and requires an understanding of red-black trees and the CFS algorithm. Offers better performance and scalability for large systems.**

4. Cause of first interruption? Is it needed in single-core systems?

In xv6, interrupts are initially enabled during the execution of a loop. This is necessary even in a single-core system. When the `ptable` lock is activated, all interrupts are disabled by the `pushcli` function.

There might be a situation where the processor is in a state where several of its processes are waiting for I/O operations to complete, and none of the other processes are in the `RUNNABLE` state. In this case, no other process is being executed. If interrupts never get enabled, we won't be able to change the state of the relevant processes to `RUNNABLE` after the completion of I/O operations, which would allow them to be executed. As a result, the system would freeze.

That's why, in this loop, interrupts are enabled for a short period (just before locking the ptable) so that we can change the state of the processes if needed. The `pushcli` function is used in xv6 to disable interrupts while acquiring a `spin lock`. This function calls `cli` but also maintains a count of how many push calls have been made so far. Also, `popcli` is used to enable them. The `pushcli` function always disables interrupts, while the `popcli` function only enables interrupts if the nesting level of `cli` calls is zero.

5. Explain two two-level interrupt managers in Linux.

In Linux, interrupt handlers are typically divided into two parts: the "top half" and the "bottom half". This two-level structure allows for efficient handling of interrupts from hardware devices.

1. **Top Half:** This is the minimum necessary part of the interrupt handler, where communication with the hardware occurs. It is triggered immediately when an interrupt is raised, and its main job is to acknowledge the interrupt and quickly service the device to prevent further interrupts. This part of the handler is also referred to as the FLIH[3].
2. **Bottom Half:** The bottom half, or SLIH[4], performs the rest of the operations, such as processing the data and copying it into memory. The bottom half is deferred and can be scheduled to run later, allowing the top half to respond to new interruptions quickly.

[3] First-Level Interrupt Handler
[4] Second-Level Interrupt Handler

This two-level interrupt handling mechanism helps to balance the need for prompt response to hardware interrupts (handled by the top half) and the need for longer, more complex processing (handled by the bottom half). It ensures that the system can continue to respond to new interrupts even while previous interrupts are still being processed.

6. How was process starvation handled?

In xv6, process starvation is mitigated through a mechanism known as “aging”.

Aging is a technique used to gradually increase the priority of processes that wait in the system for a long time. If a process uses up its entire time slice (quantum), it is moved to a lower-priority queue. However, if a process waits too long in a lower-priority queue, its priority is gradually increased (or “aged”) until it is eventually scheduled for execution. This ensures that no process waits indefinitely, thus preventing starvation.

The aging mechanism adjusts the priorities of processes based on their behavior and resource usage. Processes that consume less CPU time (I/O-bound processes) are given higher priority, while processes that consume more CPU time (CPU-bound processes) are given lower priority. This dynamic adjustment of priorities helps to ensure a balance between different types of processes and prevents starvation.

It promotes fairness, as it prevents a process from monopolizing the CPU. By adjusting priorities based on resource usage, the aging mechanism ensures that all processes, regardless of their initial priorities, get a fair chance to execute.

Implementation

Multi-level Feedback Scheduling

To achieve multi-level feedback scheduling it is needed to implement a MLFQ[5]. A multi-level feedback queue is a CPU scheduling algorithm used by operating systems to manage the execution of processes. It involves dividing processes into multiple queues with different priority levels, and each queue uses a different scheduling algorithm.

In this project, three methods of scheduling are implemented, in order of priority, they are:

1. Round Robin
2. Last Come First Serve
3. Best Job First

Which means the round robin has the highest priority.

Added structures

To organize the workspace, some structures, and enums are declared:

A. schedqueue, is an enum to track a process's queue.

```
enum schedqueue {
    UNSET,
    ROUND_ROBIN,
    LCFS,
    BJF
};
```

B. `schedpriroty`, is another enum to determine the priority of a process. We considered five priorities in our scheduling.

```
enum schedpriroty {
    HIGH = 1,
    ABOVE_NORMAL,
    NORMAL,
    BELOW_NORMAL,
    LOW
};
```

[5] multi-level feedback queue

C. bjfparams, is a simple structure to keep BJF[6]-related parameters.

```
struct bjfparams {
    enum schedprioroty priority;
    float executed_cycle;
    int arrival_time;
    int process_size;

    float priority_ratio;
    float executed_cycle_ratio;
    float arrival_time_ratio;
    float process_size_ratio;
};
```

D. schedparams, is a structure to keep scheduling parameters. We will add this structure to `struct proc`, to let every process have its scheduling behavior.

Also, `last_exec` was added to be used in the aging mechanism and will be explained in detail later in this report.

```
struct schedparams {
    enum schedqueue queue;
    struct bjfparams bjf;
    int last_exec;
};
```

Scheduling

To schedule processes correctly, first we need to define each program's queue. It will be done in process initialization. The default queue is LCFS[7], but some critical processes, like `init,` need to be added to the RR[8] queue to make sure they will be executed whenever they are needed.

[6] Best Job First
[7] Last Come First Serve
[8] Round Robin

After initialization, in the `scheduler` function, we check each queue in order of their priority. If there is an available process to execute[9] in the first queue, which is round robin, it will be chosen and the context switching procedure will be started, but if there is not, we check the second queue and the same happens to the third queue, if there is not any RUNNABLE process, we keep checking processes in a never-ending loop.

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

    struct proc* last_scheduled = ptable.proc + NPROC - 1;

    for (;;) {
        sti();
        acquire(&ptable.lock);

        p = round_robin(last_scheduled);
        last_scheduled = (p != 0) ? p : last_scheduled;
        if (p == 0)
            p = last_come_first_serve();
        if (p == 0)
            p = best_job_first();
        if (p == 0) {
            release(&ptable.lock);
            continue;
        }

        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        p->sched.last_exec = ticks;
        p->sched.bjf.executed_cycle += 0.1f;

        swtch(&(c->scheduler), p->context);
        switchkvm();

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

[9] XV6 uses RUNNABLE state

Scheduling Methods

I. Round Robin

Round-robin is a CPU scheduling algorithm where each process is given a fixed time quantum to execute before being preempted and moved to the back of the queue. This ensures fair distribution of CPU time among processes and prevents any single process from monopolizing the CPU for too long.

To implement this algorithm, first, we need a queue, but according to XV6, there is already an array of processes called `ptable`, so we need to implement our queue within that array to avoid conflicts and other complexities. To achieve this, we just need a pointer to the last scheduled process, each time we need to check the next processes in the `ptable`, there are three different possibilities:

1. we reach the end of the array, so we just need to reset the pointer to the beginning of the `ptable.`
2. we reach a `RUNNABLE` process and that process queue is round robin, we choose this process to execute and we also must set the last scheduled process to this process.
3. we reach the last scheduled process, which means we did not find any available process.

```
struct proc* round_robin(struct proc* last_scheduled) {
    struct proc* p = last_scheduled + 1;
    for (;; p++) {

        if (p >= &ptable.proc[NPROC])
            p = ptable.proc;

        if (p->state == RUNNABLE && p->sched.queue == ROUND_ROBIN)
            return p;

        if (p == last_scheduled)
            return 0;
    }
}
```

II. LCFS

This algorithm has a simple rule. The last process entered in RUNNABLE state, must be executed. To do that, we just need to keep a process-created time, and each time we should choose a process with the highest created time value.

```
struct proc* last_come_first_serve(void) {
    struct proc* next_p = 0;
    int latest_time = -1;

    struct proc* p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != RUNNABLE || p->sched.queue != LCFS)
            continue;
        int time = p->ctime;
        if (next_p == 0 || time > latest_time) {
            next_p = p;
            latest_time = time;
        }
    }

    return next_p;
}
```

III. BJS

In this algorithm, we use a formula to rank the processes(lowest rank means highest priority), after that, we need to choose the lowest rank process among all processes to execute.

```
struct proc* best_job_first(void) {
    struct proc* next_p = 0;
    float best_rank;
    struct proc* p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != RUNNABLE || p->sched.queue != BJF)
            continue;
        float rank = procrank(p->sched.bjf);
        if (next_p == 0 || rank < best_rank) {
            next_p = p;
            best_rank = rank;
        }
    }
    return next_p;
}
```

Aging

Our aging mechanism is simple. Each process is initially placed in the second queue(which is LCFS), if a process is starved at 8000 cycles, it moves to the first queue. Please note that regardless of the current queue, it always goes to the first queue(no gradual aging or multi-level aging).

The aging function will be executed at every tick(about 10 milliseconds) and it will be called in `trap.c`.

```
void aging(int curr_time) {
    struct proc* p;
    acquire(&ptable.lock);

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == RUNNABLE && p->sched.queue != ROUND_ROBIN) {
            if (curr_time - p->sched.last_exec > MAX_AGE)
                change_queue(p->pid, ROUND_ROBIN);
        }
    }
    release(&ptable.lock);
}
```

Initializing queues

To prevent init and shell from being blocked by some other processes, we need to always keep them in the first queue(round robin). we can not add them in the second queue(LCFS), because they will be created at the very beginning, so they always will be blocked by other processes, and also we can not trust the third queue(BJS), according to the fact that a user can set parameters to have higher priority compared to shell or init.

```c
int init_queue(int pid) {
    struct proc* p;
    int queue;

    // init and shell
    if (pid == 1 || pid == 2)
        queue = ROUND_ROBIN;
    else if (pid > 2)
        queue = LCFS;
    else
        return -1;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            p->sched.queue = queue;
            break;
        }
    }
    release(&ptable.lock);

    return 0;
}
```

System Calls

I. Change queue

chqueue system call is defined to change queue of a process. changing queue is needed through the OS especially when it is used in the aging mechanism. Along with that, we declared another function called init_queue which initializes the queue of newly created processes.

After changing the queue, variable last_exec, which is defined to determine the last time the process is executed, will be updated to the current time, that may seem odd, but last_exec is used in aging to see how much a process is waiting **since entering queue**, so after changing queue, it must be reset for further trackings.

```
int change_queue(int pid, int new_queue) {
    struct proc* p;
    if (new_queue == UNSET) {
        return -1;
    }

    int is_process_exist = 0;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            is_process_exist = 1;
            p->sched.queue = new_queue;
            acquire(&tickslock);
            p->sched.last_exec = ticks;
            release(&tickslock);
            break;
        }
    }
    release(&ptable.lock);

    if (!is_process_exist)
        return -1;

    return 0;
}
```

II. Set BJS parameters of a process

It does set BJS parameters of a process, if the given `pid` does not exist, returns -1. Also it is noticeable that, ratios are float but xv6 terminal can only show integers, so scheduling calculations are done in float but we can only monitor them as float numbers.

```c
int set_bjs_proc(int pid, float priority_ratio, float
arrival_time_ratio,float executed_cycle_ratio, float process_size_ratio)
{
    struct proc* p;
    int is_pid_exist = 0;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            is_pid_exist = 1;
            p->sched.bjf.priority_ratio = priority_ratio;
            p->sched.bjf.arrival_time_ratio = arrival_time_ratio;
            p->sched.bjf.executed_cycle_ratio = executed_cycle_ratio;
            p->sched.bjf.process_size_ratio = process_size_ratio;
            break;
        }
    }
    release(&ptable.lock);

    if(!is_pid_exist)
        return -1;
    return 0;
}
```

III. Set BJS parameters of system

Sets all processes' BJS parameters. Processes which will be created after calling this function are not affected by these settings. The reason all processes have the same BJS parameters is because this function sets the same values for all processes.

```
int set_bjs_sys(float priority_ratio, float arrival_time_ratio,
                float executed_cycle_ratio, float process_size_ratio) {
    struct proc* p;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        p->sched.bjf.priority_ratio = priority_ratio;
        p->sched.bjf.arrival_time_ratio = arrival_time_ratio;
        p->sched.bjf.executed_cycle_ratio = executed_cycle_ratio;
        p->sched.bjf.process_size_ratio = process_size_ratio;
    }
    release(&ptable.lock);

    return 0;
}
```

IV. print scheduling information

A simple system call to log all processes' scheduling information on the terminal.

```
int print_processes_infos(void) {
    static char* states[] = {
        [UNUSED] "unused",
        [EMBRYO] "embryo",
        [SLEEPING] "sleeping",
        [RUNNABLE] "runnable",
        [RUNNING] "running",
        [ZOMBIE] "zombie"};
    static int columns[] = {16, 8, 12, 8, 8, 8, 8, 8, 8, 8, 8, 8};
    cprintf(
        "Process_Name    PID     State     Queue   Cycle   Arrival
Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank\n"

"------------------------------------------------------------------------
----------------------------------------\n");
    struct proc* p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == UNUSED)
            continue;
        const char* state;
        if (p->state >= 0 && p->state < NELEM(states) &&
states[p->state])
            state = states[p->state];
        else
            state = "???";
        cprintf("%s", p->name);
        printspaces(columns[0] - strlen(p->name));

        cprintf("%d", p->pid);
        printspaces(columns[1] - digitcount(p->pid));

        cprintf("%s", state);
        printspaces(columns[2] - strlen(state));

        cprintf("%d", p->sched.queue);
        printspaces(columns[3] - digitcount(p->sched.queue));

        cprintf("%d", (int)p->sched.bjf.executed_cycle);
        printspaces(columns[4] -
digitcount((int)p->sched.bjf.executed_cycle));
```

```
        cprintf("%d", p->sched.bjf.arrival_time);
        printspaces(columns[5] - digitcount(p->sched.bjf.arrival_time));

        cprintf("%d", p->sched.bjf.priority);
        printspaces(columns[6] - digitcount(p->sched.bjf.priority));

        cprintf("%d", p->sched.bjf.process_size);
        printspaces(columns[7] - digitcount(p->sched.bjf.process_size));

        cprintf("%d", (int)p->sched.bjf.priority_ratio);
        printspaces(columns[8] -
digitcount((int)p->sched.bjf.priority_ratio));

        cprintf("%d", (int)p->sched.bjf.arrival_time_ratio);
        printspaces(columns[9] -
digitcount((int)p->sched.bjf.arrival_time_ratio));

        cprintf("%d", (int)p->sched.bjf.executed_cycle_ratio);
        printspaces(columns[10] -
digitcount((int)p->sched.bjf.executed_cycle_ratio));

        cprintf("%d", (int)p->sched.bjf.process_size_ratio);
        printspaces(columns[11] -
digitcount((int)p->sched.bjf.process_size_ratio));

        cprintf("%d", (int)procrank(p->sched.bjf));
        cprintf("\n");
    }

    return 0;
}
```

User-Level Program

In order to run written system calls we wrote a user-level program called sched.

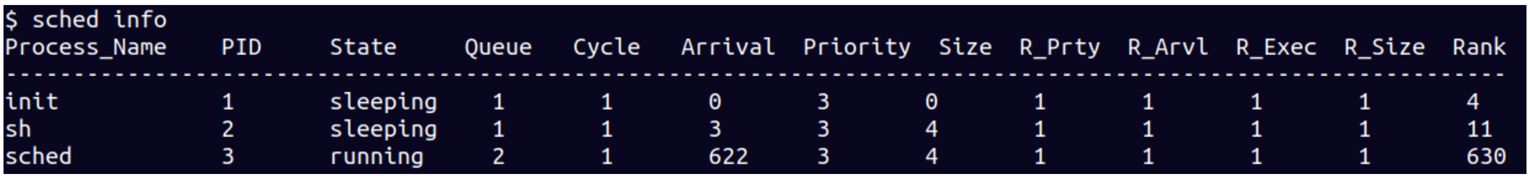
```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf(1, "invalid command\n");
        exit();
    }
    else if (!strcmp(argv[1], "chq")) {
        if (argc < 4) {
            printf(1, "invalid command\n");
            exit();
        }
        set_queue(atoi(argv[2]), atoi(argv[3]));
    }
    else if (!strcmp(argv[1], "bjsp")) {
        if (argc < 7) {
            printf(1, "invalid command\n");
            exit();
        }
        set_proc_bjs_params(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]),
atoi(argv[5]), atoi(argv[6]));
    }
    else if (!strcmp(argv[1], "bjss")) {
        if (argc < 6) {
            printf(1, "invalid command\n");
            exit();
        }
        set_sys_bjs_params(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]),
atoi(argv[5]));
    }
    else if (!strcmp(argv[1], "info")) {
        print_info();
    }
    else {
        printf(1, "%d\n", atoi(argv[1]));

    }
    exit();
}
```

We also wrote a simple foo program to perform prolonged computations.

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

#define PROCS_NUM 10

int randstate = 1;
int rand() {
    int a = 1664525;
    int b = 1013904223;
    int c = 1234567890;
    randstate = (randstate ^ a) * b + c;
    return randstate;
}

void perform_computation(int i) {
    int x = 1;
    for (int j = 0; j < i; j++)
        for (long k = 0; k < 1000000000; k++)
            x++;
}

int main() {
    for (int i = 0; i < PROCS_NUM; i++) {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0) {
            int sleep_time = rand() % 10000;
            sleep(sleep_time);
            perform_computation(i);
            exit();
        }
    }
    while (wait() != -1)
        ;

    exit();
}
```

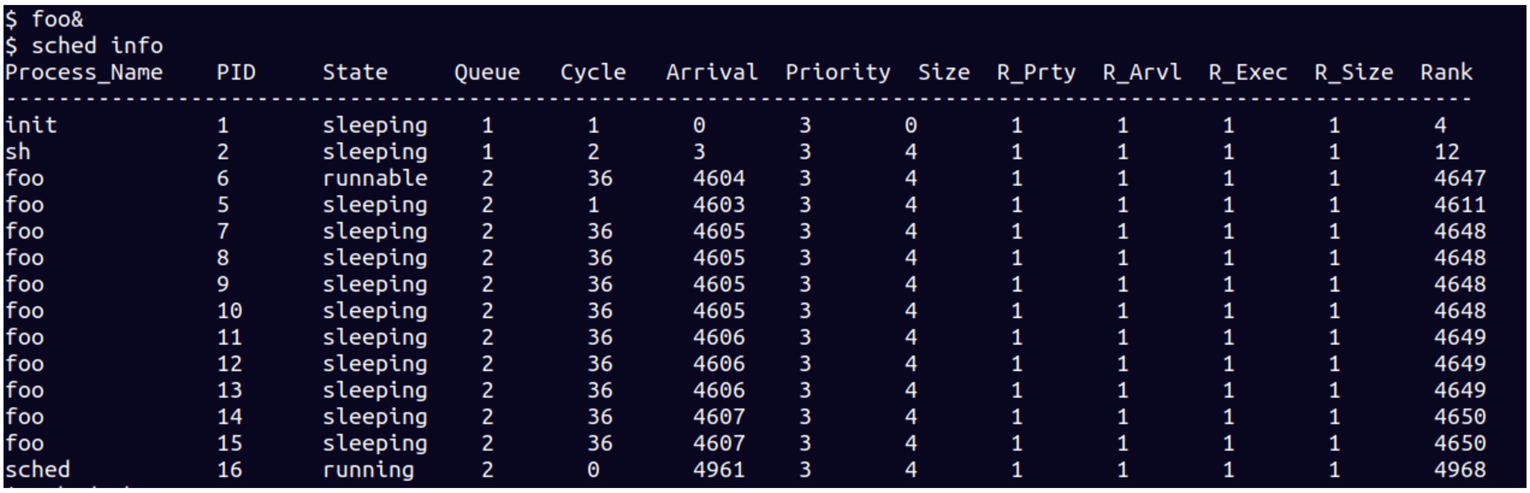
Testing

First lets test `sched info`



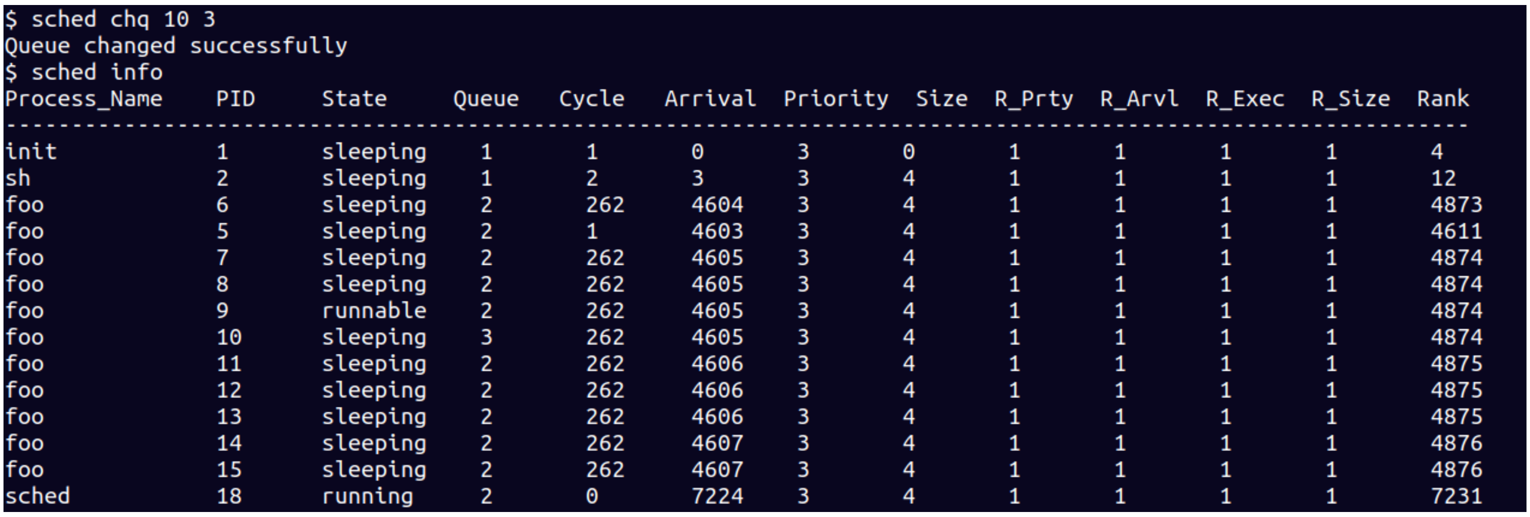
```
$ sched info
Process_Name    PID     State     Queue   Cycle   Arrival  Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank
------------------------------------------------------------------------------------------------------------------
init            1       sleeping   1       1       0        3         0     1       1       1       1       4
sh              2       sleeping   1       1       3        3         4     1       1       1       1       11
sched           3       running    2       1       622      3         4     1       1       1       1       630
```

now, we need to run foo process in the background, to do that, we just need to write `foo&`. As can be seen 10 process are created:



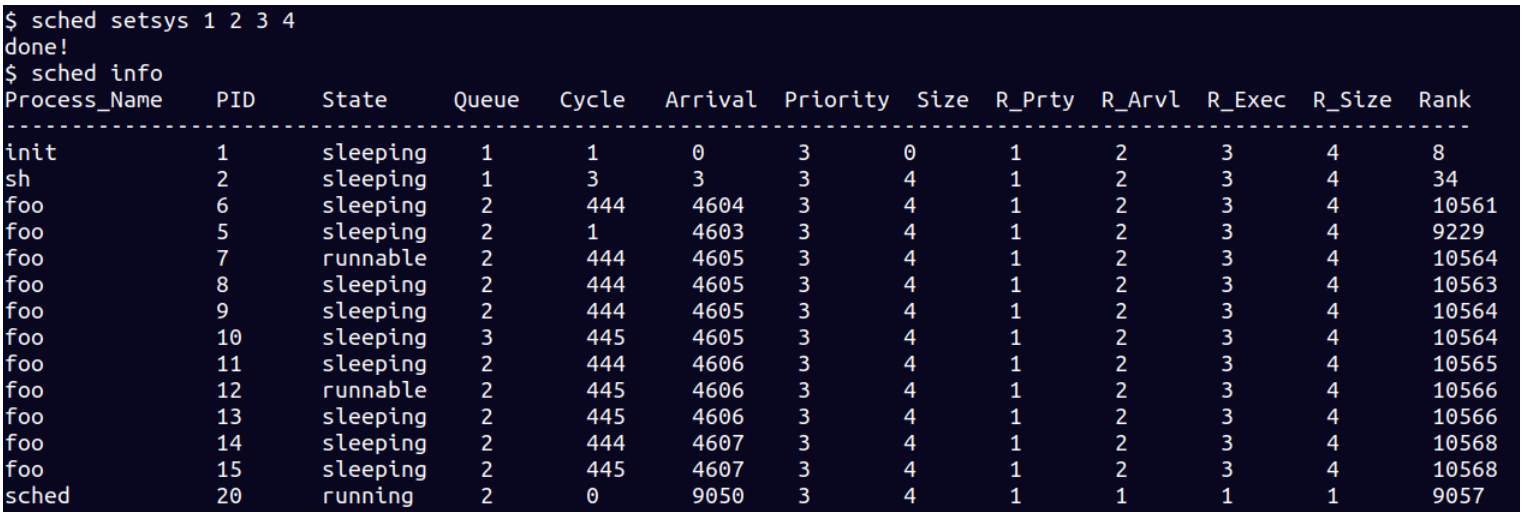
```
$ foo&
$ sched info
Process_Name    PID     State     Queue   Cycle   Arrival  Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank
---------------------------------------------------------------------------------------------------------------
init            1       sleeping   1       1       0        3         0     1       1       1       1       4
sh              2       sleeping   1       2       3        3         4     1       1       1       1       12
foo             6       runnable   2       36      4604     3         4     1       1       1       1       4647
foo             5       sleeping   2       1       4603     3         4     1       1       1       1       4611
foo             7       sleeping   2       36      4605     3         4     1       1       1       1       4648
foo             8       sleeping   2       36      4605     3         4     1       1       1       1       4648
foo             9       sleeping   2       36      4605     3         4     1       1       1       1       4648
foo             10      sleeping   2       36      4605     3         4     1       1       1       1       4648
foo             11      sleeping   2       36      4606     3         4     1       1       1       1       4649
foo             12      sleeping   2       36      4606     3         4     1       1       1       1       4649
foo             13      sleeping   2       36      4606     3         4     1       1       1       1       4649
foo             14      sleeping   2       36      4607     3         4     1       1       1       1       4650
foo             15      sleeping   2       36      4607     3         4     1       1       1       1       4650
sched           16      running    2       0       4961     3         4     1       1       1       1       4968
```

After using `chq` we can see that process queue is changed:

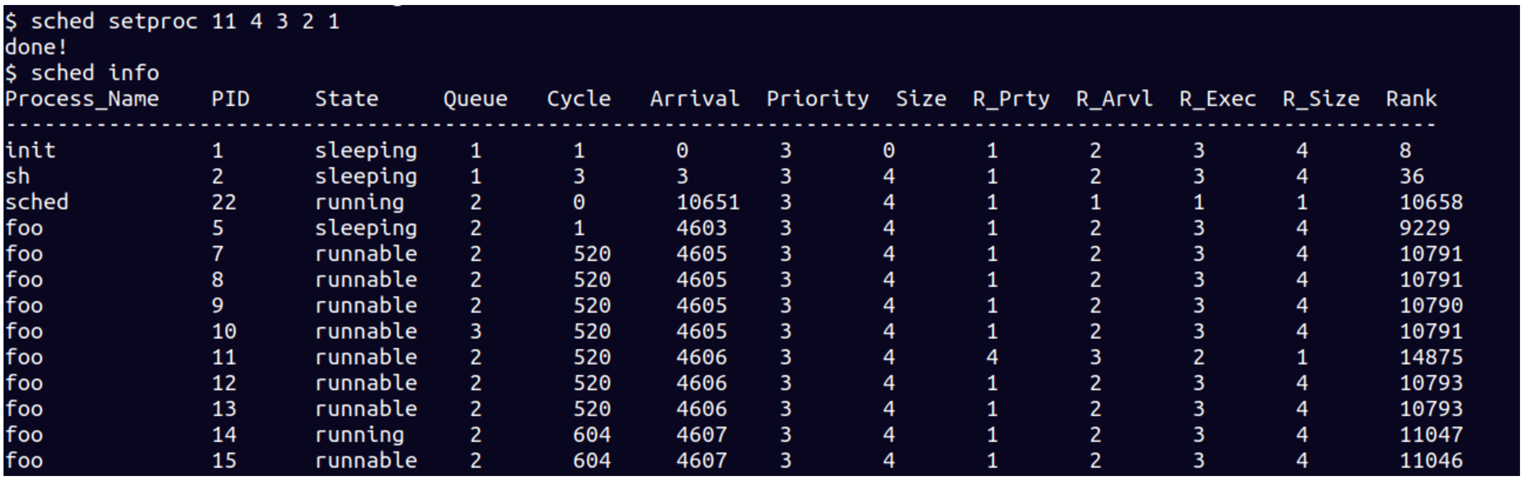


```
$ sched chq 10 3
Queue changed successfully
$ sched info
Process_Name    PID     State     Queue   Cycle   Arrival  Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank
---------------------------------------------------------------------------------------------------------------
init            1       sleeping   1       1       0        3         0     1       1       1       1       4
sh              2       sleeping   1       2       3        3         4     1       1       1       1       12
foo             6       sleeping   2       262     4604     3         4     1       1       1       1       4873
foo             5       sleeping   2       1       4603     3         4     1       1       1       1       4611
foo             7       sleeping   2       262     4605     3         4     1       1       1       1       4874
foo             8       sleeping   2       262     4605     3         4     1       1       1       1       4874
foo             9       runnable   2       262     4605     3         4     1       1       1       1       4874
foo             10      sleeping   3       262     4605     3         4     1       1       1       1       4874
foo             11      sleeping   2       262     4606     3         4     1       1       1       1       4875
foo             12      sleeping   2       262     4606     3         4     1       1       1       1       4875
foo             13      sleeping   2       262     4606     3         4     1       1       1       1       4875
foo             14      sleeping   2       262     4607     3         4     1       1       1       1       4876
foo             15      sleeping   2       262     4607     3         4     1       1       1       1       4876
sched           18      running    2       0       7224     3         4     1       1       1       1       7231
```

Other system calls were tested as well.



```
$ sched setsys 1 2 3 4
done!
$ sched info
Process_Name    PID     State     Queue   Cycle   Arrival  Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------------------
init            1       sleeping   1       1       0        3         0     1       2       3       4       8
sh              2       sleeping   1       3       3        3         4     1       2       3       4       34
foo             6       sleeping   2       444     4604     3         4     1       2       3       4       10561
foo             5       sleeping   2       1       4603     3         4     1       2       3       4       9229
foo             7       runnable   2       444     4605     3         4     1       2       3       4       10564
foo             8       sleeping   2       444     4605     3         4     1       2       3       4       10563
foo             9       sleeping   2       444     4605     3         4     1       2       3       4       10564
foo             10      sleeping   3       445     4605     3         4     1       2       3       4       10564
foo             11      sleeping   2       444     4606     3         4     1       2       3       4       10565
foo             12      runnable   2       445     4606     3         4     1       2       3       4       10566
foo             13      sleeping   2       445     4606     3         4     1       2       3       4       10566
foo             14      sleeping   2       444     4607     3         4     1       2       3       4       10568
foo             15      sleeping   2       445     4607     3         4     1       2       3       4       10568
sched           20      running    2       0       9050     3         4     1       1       1       1       9057
```



```
$ sched setproc 11 4 3 2 1
done!
$ sched info
Process_Name    PID     State     Queue   Cycle   Arrival  Priority  Size  R_Prty  R_Arvl  R_Exec  R_Size  Rank
----------------------------------------------------------------------------------------------------------------
init            1       sleeping   1       1       0        3         0     1       2       3       4       8
sh              2       sleeping   1       3       3        3         4     1       2       3       4       36
sched           22      running    2       0       10651    3         4     1       1       1       1       10658
foo             5       sleeping   2       1       4603     3         4     1       2       3       4       9229
foo             7       runnable   2       520     4605     3         4     1       2       3       4       10791
foo             8       runnable   2       520     4605     3         4     1       2       3       4       10791
foo             9       runnable   2       520     4605     3         4     1       2       3       4       10790
foo             10      runnable   3       520     4605     3         4     1       2       3       4       10791
foo             11      runnable   2       520     4606     3         4     4       3       2       1       14875
foo             12      runnable   2       520     4606     3         4     1       2       3       4       10793
foo             13      runnable   2       520     4606     3         4     1       2       3       4       10793
foo             14      running    2       604     4607     3         4     1       2       3       4       11047
foo             15      runnable   2       604     4607     3         4     1       2       3       4       11046
```