CFS Scheduler

Name: Christian Magnus Engebretsen Heimvik Kingo ID: 2025318314

How scheduling works in XV6 today

The scheduling on the XV6 before my implementation operated with 3 *levels* of context. First of all; all the way down is the user process, with the corresponding "lowest" context. Say process A is running user program with its corresponding cotnext and a timer tick happens. What happens then is that the timer interrupt is catched by the IDT (as explained in the previous report), corresponding vector in `vectors.S` is executed, which does `jmp alltraps` which jumps to `trapasm.S` where the building of the trapframe happens and the instruction to `call trap` is executed.

This changes our context from the lower user-level context to process A's kernel-context, and we are now in the kernel in `trap.c`. This can be regarded as a "middle" context, and under syscalls only these two contexts are used back and fourth. However, under a timer interrupt tick the following code statement in `trap.c` will be evalueated true

```
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER)
yield();
```

which invokes the call to `yield()`. This call to `yield()` will first lock the process table, change the process's A own state, for then to call `sched()`. This fucniton will after some condition checking, call `swtch()`. This effectively change the context once more (by switching cpu registers, changing instruction and stack pointer) from the "middle" context of process A to the "highest" context of the scheduler.

With this new highest context, the execution will appear out of `swtch()` in `scheduler()` as if nothing happend. After also switching the per-process page table the releasing the page table lock (for other cpus to grab it), the next process in the process list is chosen. Say process B time. The function `scheduler()` then calls `swtch()` which switches context down to "middle" level, but this time execution and context will come out of process B's kernel `swtch()` as if nothing happend. Execution will then return back to the "lowest" user context of process B, trough the interrupt return. User program B will then continue executing.

How the CFS scheduler works

The Completely Fair scheduler (CFS) is a CPU scheduling algorithm that aims to provide fair CPU time to all processes in the system, but also taking priorities into account. Each process has 3 major variables that determine its behaviour in the scheduler:

- **weight**: The weight of a process is calculated as the inverse of its niceness. The nicess, the lower the weight. The weight is hard coded due to its constant nature and given as an array, indexed by its corresponding niceness. The array of weights was generated by a simple python script and is defined as follows:

```
static const int cfs_weights[40] = {
    88818,     71054,     56843,     45475,     36380,
    29104,     23283,     18626,     14901,     11921,
    9537,     7629,     6104,     4883,     3906,
    3125,     2500,     2000,     1600,     1280,
    1024,     819,     655,     524,     419,
    336,     268,     215,     172,     137,
    110,     88,     70,     56,     45,
    36,     29,     23,     18,     15,
};
```

- **vruntime**: A variable that accumulates when the process is running, and it tracks the "effective" amount of time a process has been running, accordng to its priority. The vruntime for a given process needs to be updated after every time it has run, and the increment in vruntime of a given process p after each run is $$ \text{p->vruntime} += \text{p->vruntime}\frac{\text{weights}[20]}{\text{p->weight}}$$
- **timeslice**: The timeslice a process is allowed to run before has to yield. It must in all cases be larger than sched_granularity. The timeslice has to be calculated the when having picked the next one based of the vruntime of the processes, and before the process is set to RUNNING state by the following formula, where 10 is out sched_latency or our ticks: $$ \text{p->timeslice} = \text{10} + \frac{\text{p->weight}}{\sum_i \text{p[i]->runnable ? p[i]->weight : 0}}$$

The CFS scheduler will then schedule the processes as follows:

1. Pick the process with the lowest vruntime from ready processes.
2. Calculate the timeslice for the process according to the equation above.
3. Set the process to RUNNING state.
4. When done running, update the vruntime of the process according to the equation above.
5. If not done, set the process state to RUNNABLE. Else indicate finsihed.
6. Go to step 1.

Implemenation consideratins

For the sake of our implementation, we must consider the following:

- Each process that is running, will be interrupted each timer tick (each time the timer interrupt is called) and execution will end up in `scheduler()` with its context. Implement the scheduler algorithm here.
- Update ps correctly As the CFS scheduler relies on global state, these need to be maintained correctly through all process state changes. This circumferes:
- Newly forked processes: This is the default initialization of a new process. Child process `c` inherits runtime, vruntime and nice of parent `p`.

```
c->runtime = p->runtime
c->vruntime = p->vruntime
c->weight = p->weight
```

- Woken processes: The vruntime has to be set so no monopolization of the CPU can happen. This is done by

```
```

The `sched()` to the newly woken process is not included in the `wakeup()`, but this is the default implementation. This makes sence as well, the wakeup does and should to is to make the process **avalibale** for the `scheduler()`. It shouldn't force anything to happen. If we want thew woken up process to be run more recently after wakeup, give it higher priority then.