הטכניון – מכון טכנולוגי לישראל
מעבדה במערכות הפעלה 046210
תרגיל בית מס' 2

תאריך הגשה: 17.03.2024, עד 23:55

Introduction

In this assignment you will learn about the scheduling algorithm and how to update it. This assignment builds on the previous one, so be sure to finish that before starting this one.

In the previous assignment you implemented a way for processes to play a role-playing game and form parties. However, processes in parties had to wait a lot while playing: after each process in the party got the CPU, there were a lot of other processes receiving CPU time before the next process in the party could get in.

Your mission in this assignment is to update the scheduling algorithm and make it schedule only processes from a single party, starting whenever the first process from this party first got a time slice and until no process in the party is in the RUNNING state. In order words, the processes will be scheduled as *parties*, with all other processes waiting for the current party to finish its business.

Background Information

Introduction to the scheduling can be found at the end of the file. More details are in the "The O(1) Scheduler" document in Moodle.

Detailed Description

This assignment builds on the previous assignment. For this work you should use the RPG system you implemented in the previous assignment and implement a scheduling algorithm for it. The updated scheduling algorithm should include the following rules:

1. For any process A which has an RPG character, if A is currently running on the CPU, the next process B to get the CPU will be some member of A's party that is currently in the RUNNING state.
    a. B can be in the expired queue. In this case it will be moved back to the active queue.
    b. B can equal A, for example if A isn't associated with a party (meaning it's a party of one)
2. All processes in a single party $P_1, P_2, \ldots \in P$, should get about the same CPU time and not starve each other.
3. Processes can join and leave parties at any time using the calls from the previous exercise. The game state that counts is the one closest to the scheduler's decision on the next process. For example, assume process A is in party P_1 when it's scheduled and during its timeslice it moves to party P_2. After A's timeslice is done, the next process to be run will belong to P_2.

Useful Information

- You can assume that the system is with a single CPU.
- An introduction to task scheduling is given in the Background Information section below
- More on system calls and task scheduling can be found in the “Understanding The Linux Kernel” book.
- Use **printk** for debugging.
- You are not allowed to use **syscall** functions to implement code wrappers, or to write the code wrappers for your system calls using the macro **_syscall1**. You should write the code wrappers according to the example of the code wrapper given above.
- Use Bootlin (see link in Moodle) to easily find where some function/variable is defined in the kernel

Tips for the Solution

- Spend time on understanding how the kernel chooses the next process to be scheduled.
- Use the functions **enqueue_task** and **dequeue_task** (definded in ‘sched.c’) to move processes between queues in the **runqueue.**
- You can write your own function for manipulating the **runqueue**.
- To check if a process is running (in the **runqueue**) you can check the **array** field of its **task_struct** (if the field is set to NULL, then the process is not in the **runqueue**). To check if a process is in the **active** array, compare its **array** field with the **array** field of a process that’s currently active (e.g. **current**).
- The queues themselves are regular kernel linked lists, and can be modified using **list_add**, **list_del**, etc.
- Take the time to think where to put your changes to avoid breaking existing functionality.

Testing Your Custom Kernel

You should test your new kernel thoroughly (all functionality and error messages that you can simulate). Note that your code will be tested using an automatic tester. This means that you should pay attention to the exact syntax of the wrapper functions, their names and the header file that defines them. You can use whatever file naming you like for the source/header files that implement the system calls themselves, but they should compile and link using the kernel make file. To do so add your source file in the following line inside the “Makefile” file located in the “kernel” folder:

obj-y   = sched.o dma.o … <your_file_name>.o

Submission Procedure

1. Submissions allowed in pairs only.
2. You should submit through the moodle website (one submission per pair).
3. You should submit one zip file named according to the format ID1_ID2.zip. It should contain:
   a. All files you added or modified in your custom kernel (including relevant files from the previous exercise). The files should be arranged in folders that preserve their relative path to the root path of the kernel source, i.e:
      ```
      zipfile -+
      ```

```
         |
         +- submitters.txt
         |
         +- rpg_api.h
         |
         +- kernel/ -+
         |           |
         |           +-...
         |
         +- include/ -+
         |            |
         |            +-...
        ...
```

b. The wrapper functions file “rpg_api.h”, as in the previous excercise.
c. A file named “submitters.txt” which lists the name **email** and ID of the participating students. The following format should be used:

   ploni almoni ploni@t2.technion.ac.il 123456789
   john smith john@gmail.com 123456789

   Note: that you are required to include your email.

Emphasis Regarding Grade

- Your grade for this assignment makes 35% of final grade.
- Your submissions will be checked using an automatic checker, pay attention to the submission procedure.
- You are allowed (and encouraged) to consult with your fellow students but you are not allowed to copy their code. You can share tests.
- Your code should be adequately documented and easy to read.
- Incorrect submission format and late submissions will be penalized.
- Obviously, you should free all dynamically allocated memory.

Background Information

This assignment requires basic understanding of the task scheduling in the Linux kernel. Below is some information to get you started. More information can be found in the recommended links in the lab Moodle and the OS course.

Linux is a multitasking operating system. A multitasking operating system achieves the illusion of concurrent execution of multiple processes, even on systems with single CPU. This is done by switching from one process to another very quickly. Linux uses **Preemptive Multitasking.** This means that the kernel decides when a process is to cease running and a new process is to begin running. Tasks can also intentionally **block** or **sleep** until some event occurs (keyboard press, passage of time etc.). This enables the kernel to better utilize the resources of the system and give the user a responsive feeling.

Task States

The state field of the process descriptor describes what is currently happening to the process. The process can be in one of the following states:

TASK_RUNNING: The process is either executing on a CPU or waiting to be executed.

TASK_INTERRUPTIBLE: The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK_RUNNING).

TASK_UNINTERRUPTIBLE: Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used.

TASK_STOPPED: Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.

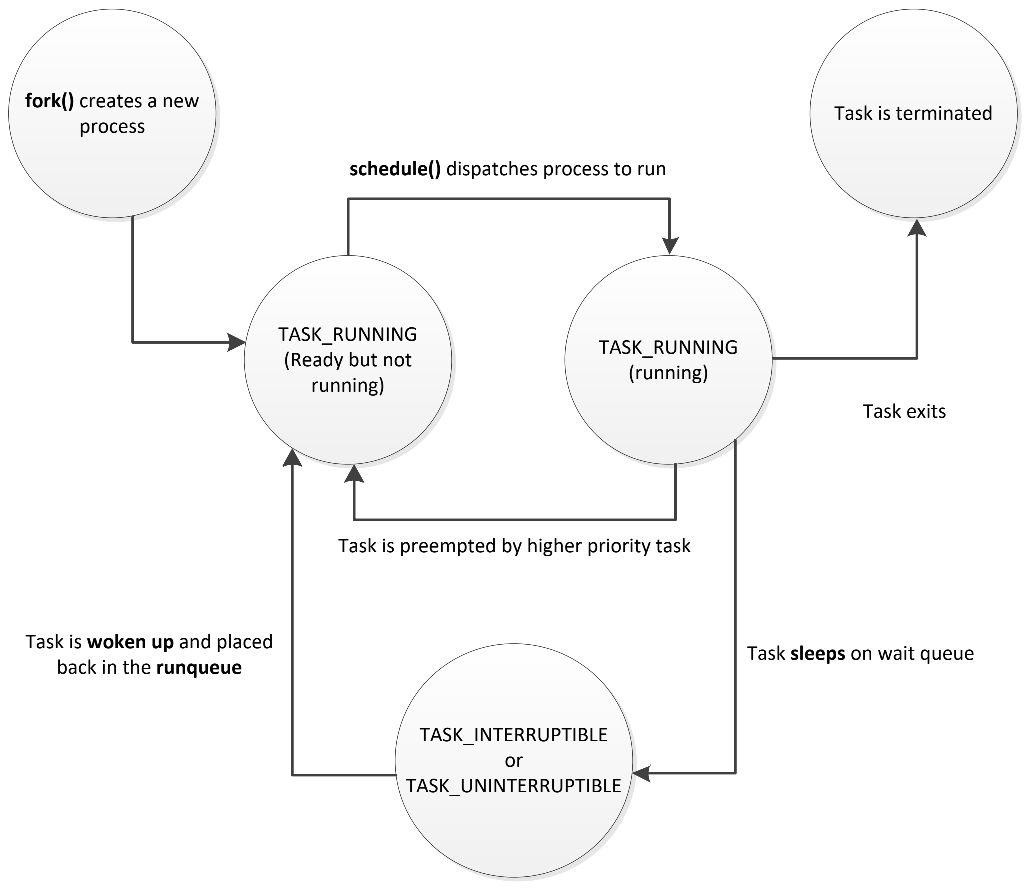
TASK_ZOMBIE: Process execution is terminated, but the parent process has not yet issued a **wait()** or **waitpid()** system call to return information about the dead process.[*] Before the wait()-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.

The value of the state field can be set using simple assignment, i.e,

```
p->state = TASK_RUNNING;
```

or using the macros **set_current_state** and **set_task_state**.

The life cycle of a process is shown in the following diagram:


fork() creates a new process
Task is terminated
schedule() dispatches process to run
TASK_RUNNING (Ready but not running)
TASK_RUNNING (running)
Task exits
Task is preempted by higher priority task
Task is **woken up** and placed back in the **runqueue**
Task **sleeps** on wait queue
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE

Task Scheduling

Note: The scheduling algorithm, called the O(1) scheduler, in our kernel (version 2.4) is taken from a kernel version 2.6. If you want to look online for information on the algorithm, look for information relevant for kernel version 2.6 (or use the algorithm's name).

The scheduling algorithm is implemented in "kernel/sched.c", and most of the logic is in the **scheduler_tick** and **schedule** functions. Linux scheduling is based on "time sharing" technique. The CPU time is divided into *slices*, one for each runnable process (processes with the **TASK_RUNNING** state). A duration of the time slice depends on the priority of the process and ranges between 10ms to 300ms. Each CPU runs only one process at a time. The kernel keeps track of time using timer interrupts. When the time slice of the currently running process expires, the kernel scheduler is invoked and another task is set to run for the duration of its time slice. Switching between tasks is done through **context** switch. Switching of the currently running task can also occur before the expiration of its time slice. This can occur due to interrupts that wake up processes with higher priority or when the currently running process yields execution to the kernel (e.g. **blocks** or **sleeps**).

The kernel holds all runnable processes (processes with the **TASK_RUNNING** state) in a data structure called a **runqueue**. The runnable processes are further divided to two arrays: processes that have yet to exhaust their time slice are in the **active** array, processes those whose time slice has expired are in the **expired** array. Each array is a collection of linked lists, one for each possible priority. The **runqueue** also points to the current running process (which obviously resides in the **active** array). Each time the **schedule()** function is called it selects the next running process by taking the first process from the first non-empty list in the **active** array. Once the time slice of a process expires it is moved to the **expired** array. When the **active** is empty, the **expired** and **active** arrays are switched, and the **active** processes are assigned new time slices.

Kernel Timing

The kernel keeps track of time using the *timer interrupt*. The timer interrupt is issued by the system timer (implemented in hardware). The period of the system timer is called 'tick'. The *timer interrupt* advances the tick counter (called **jiffies**), and initiates time dependent activities in the kernel (decrease the time slice of the current running process, wake up processes that **sleep** waiting for a timer event etc.). The **jiffies** variable (defined in "include/linux/sched.h") counts the system 'tick's event since start up. The 'tick' period duration depends on the specific linux version. The **HZ** variable is use for converting the 'tick's to seconds:

$$\boldsymbol{time\ in\ seconds = \frac{jiffies}{HZ}}$$