CSE331 Fall 2023
Term Project Phase II:

The Scheduler

Goal: You will learn more about how process management is designed, particularly how the scheduler is implemented. The scheduler is the heart of the multiprogramming system; it is responsible for choosing a new task to run whenever the CPU is available. Then, you will learn about an alternative approach called the Stride scheduling policy. You will replace the standard scheduler with the Stride scheduler. Finally, you will learn how to observe the behavior of the Stride scheduler relative to the behavior of the standard scheduler.

Introduction

Process management is the part of the OS that creates and destroys processes, provides mechanisms to be used for synchronization, and multiplexes the CPU among runnable processes. You considered several aspects of process management throughout the course. This phase augments the discussion of the process descriptor and process creation and then focuses on the scheduler.

Process Management

Linux uses a different low-level view of how the OS is designed than the process model that appears at the system call interface. In this low-level view, when the hardware is started, a single hardware process begins to run. After the Linux kernel has been loaded, the initial process is created (to use up all otherwise unused CPU cycles) and then Linux processes are started according to how the machine is configured to operate. For example, **getty** processes are started for each login port and daemons are started as specified. Thus, when the system is ready for normal operation, several processes are in existence, most of which are blocked on various conditions. The runnable tasks (those whose states are TASK_RUNNING) begin competing for the CPU. In the kernel's view of multiprogramming, summarized in Figure 1, the initial process never really creates any separate computational units (processes). Instead, a **fork()** command defines a new kernel **struct task_struct** data structure. The initial process simply multiplexes among the programs that are loaded in different address spaces as it sees fit (and as defined by the task descriptor). That is, within the kernel, only one thread of execution exists that jumps from one runnable task to another. At the system call interface, this behavior has the appropriate UNIX process semantics.

The **fork()** system call (the main work is performed in **do_fork()** defined in **kernel/fork.c**) performs the process creation steps, covered in related lab hours.

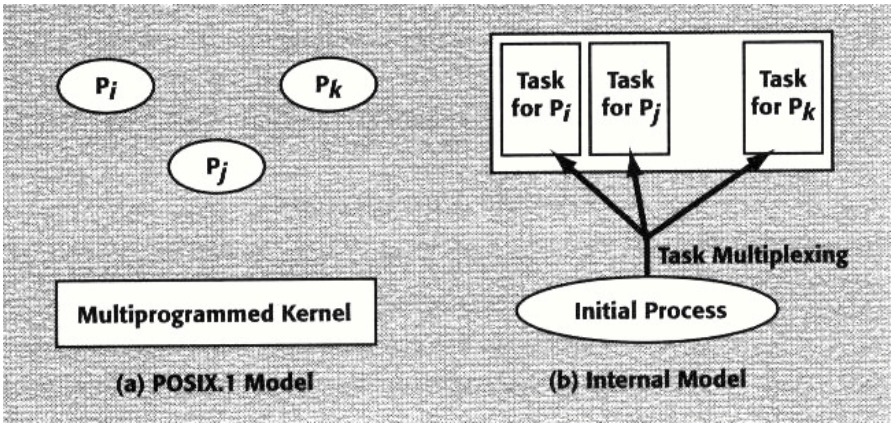

Figure 1: **Kernel Level Multitasking**

Following code fragment omits portions that implement concepts not discussed here. The remaining code fragments are annotated with C++ style comments to highlight the most critical steps.

```
int do_fork(
        unsigned long clone_flags, unsigned long usp,
        struct pt_regs *regs
)
{
    int nr;
```

```
    ...
    struct task_struct *p;
    ...
    if(clone_flags & CLONE_PID)
    {
        /* This is only allowed from the boot up thread */
        if(current->pid)
            return -EPERM;
    }

    current->vfork_sem = &sem
// Allocate the task descriptor entry
    p = alloc_task_struct();
    ...
// Copy the parent task descriptor entries to the child entry
    *p = *current;
    ...
// Find an empty slot in the task structure
    ...
// Modify/initialize child-specific fields
    ...
    p->swappable = 0;
    *(unsigned long *) p->kernel_stack_page = STACK_MAGIC;
    p->state = TASK_UNINTERRUPTIBLE;
    copy_flags(clone_flags, p);
    p->pid = get_pid(clone_flags);
    p->state = TASK_RUNNABLE;
    p->next_run = p;
    p->prev_run = p;

    p->p_pptr = p->p_opptr = current;
    p->p_cptr = NULL;
    init_waitqueue(&p->wait_chldexit);
// Set fields related to signals
```

```
    ...
    p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
    init_timer(&p->real_timer);
    p->real_timer.data = (unsigned long) p;
    ...
    p->times.tms_utime = p->itmes.tms_stime = 0;
    p->times.tms_cutime = p->times.tms_cstime = 0;
    ...
    p->start_time = jiffies;
    ...
    nr_tasks++;
    ...
    /* copy all the process information */
    if (copy_files(clone_flags, p))   // File table and files
        goto bad_fork_cleanup;
    if (copy_fs(clone_flags, p))      // User data segment
        goto bad_fork_cleanup_files;
    if (copy_sighand(clone_flags, p)) // Signals and handlers
        goto bad_fork_cleanup_fs;
    if (copy_mm(clone_flags, p))      // Virtual memory tables
        goto bad_fork_cleanup_sighand;
    retval = copy_thread(nr, clone_flags, usp, p, regs);
    if(retval)
        goto bad_fork_cleanup_mm;
    ...
    /* ok, now we should be set up.. */
    p->swappable = 1;
    p->exit_signal = clone_flags & CSIGNAL;
    ...

bad_fork_cleanup_mm:
    ...
bad_fork_cleanup_sighand:
    ...
bad_fork_cleanup_fs:
    ...
bad_fork_cleanup_files:
    ...
bad_fork_cleanup:
    ...
}
```

Each process descriptor has a link to parents and siblings. The **p_opptr** field points to the process's original parent process, and the **p_pptr** field points to the parent. The **p_cptr** field points to this process's youngest child (the last child process that it created), and the **p_ysptr** field points to this process's next younger sibling process. Finally, the p_osptr field points to the next older sibling process.

Process States

The **state** field in the task descriptor can have any of the following values:

-TASK_RUNNING -TASK_INTERRUPTIBLE -TASK_UNINTERRUPTIBLE
-TASK_ZOMBIE -TASK_STOPPED

As you go through the kernel code, you will often see statements of the form

current->state = TASK_...;

For example, in the **sleep_on()** code, **state** is set to TASK_UNINTERRUPTIBLE just before the process is placed on a wait queue. In **wake_up()**, **state** is set back to TASK_RUNNING after it is removed from the wait queue. Recall that the distinction between the TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE states relates to the sleeping task's ability to accept signals. TASK_ZOMBIE represents a state in which a process has terminated, but its parent has not yet performed a **wait()** system call to recognize that it has terminated. That is, the task does not disappear from the system until the parent is notified of its termination. The final possible value for the state field is TASK_STOPPED, which means that the task has been halted by a signal or through the ptrace mechanism. Thus most of the kernel code sets the state field to TASK_RUNNING, TASK_INTERRUPTIBLE, or TASK_UNINTERRUPTIBLE.

Outside of the kernel, a different set of process states is visible to users. These states are defined by the POSIX model rather than by the Linux implementation, though they are analogous to the internal states kept in the task descriptor. You can review these external states by inspecting the manual page for the STAT column in a **ps -t** command. The STAT output column has three fields. The first field may be one of the following:

- **R:** The process is running or runnable (its kernel state is TASK_RUNNING).
- **S:** The process is sleeping (its kernel state is TASK_INTERRUPTIBLE).
- **D:** The process is in an uninterruptible sleep (its kernel state is TASK_UNINTERRUPTIBLE).
- **T:** The process is either stopped or being traced (its kernel state is TASK_STOPPED).
- **Z:** The process is in the zombie state (its kernel state is TASK_ZOMBIE).

The second field of **ps** is either blank, or **W** if the process currently has no pages loaded. The third field is either blank, or **N** if the process is running at a reduced priority. Most of the kernel code transitions the state of the process among TASK_RUNNING, TASK_INTERRUPTIBLE, and TASK_UNINTERRUPTIBLE. Roughly speaking, whenever a process is to be blocked, it will be put on a queue (usually a wait queue) and its state will be

changed to one of TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE and then the scheduler will be called. A task/process, when awakened, is removed from the wait queue and its state is changed to TASK_RUNNING. This leaves the scheduler as the final critical piece of process management that this manual considers.

Scheduler Implementation

The scheduler is a kernel function, **schedule()** that gets called from various other system call functions (usually via **sleep_on()**) and after every system call and slow interrupt. Each time that the scheduler is called, it does the following.

1. Performs various periodic work (such as running device bottom halves).
2. Inspects the set of tasks in the TASK_RUNNING state.
3. Chooses one task to execute according to the scheduling policy.
4. Dispatches the task to run on the CPU until an interrupt occurs.

The scheduler contains three built-in scheduling strategies, identified by the constants SCHED_FIFO, SCHED_RR, and SCHED_OTHER. The scheduling policy for each process can be set at runtime by using the **sched_setscheduler()** system call (defined in **kernel/sched.c**). The task's current scheduling policy is saved in the policy field. The SCHED_FIFO and SCHED_RR policies are intended to be sensitive to real-time requirements, so they use scheduling priorities in the range [0:99], whereas SCHED_OTHER handles only the zero priority. Each process also has a priority field in its task descriptor. Conceptually, the scheduler has a multilevel queue with 100 levels (corresponding to the priorities from 0 to 99). Whenever a process with a priority that is higher than the currently running process becomes runnable, the lower-priorityprocess is preempted and the higher-priority process begins to run. Of course, in the normal timesharing (SCHED_OTHER) policy, processes do not preempt one another, since they all have a priority of zero.

Any task/process that uses the SCHED_FIFO policy must have superuser permission and have a priority of 1 to 99 (that is, it has a higher priority than all SCHED_OTHER tasks). Thus a SCHED_FIFO task that becomes runnable takes priority over every SCHED_OTHER task. The SCHED_FIFO policy does not reschedule on a timer interrupt. Once a SCHED_FIFO task gets control of the CPU, it will retain control until either it completes, blocks by an I/O call, calls **sched_yield()**, or until a higher priority task becomes runnable. If it yields or is preempted by another, higher-priority task, it is placed at the end of the task queue at the same priority level.

The SCHED_RR policy has the same general semantics as the SCHED_FIFO policy, except that it uses the time-slicing mechanism to multiplex the CPU among tasks that are in the highest-priority queue. A running SCHED_RR task that is preempted by a higher-priority task is placed back on the head of its queue so that it will resume when its queue priority is again the highest in the system. The preempted process will then be allowed to complete its time quantum.

The SCHED_OTHER policy is the default timesharing policy (which was explained in the lab hour). It uses the conventional time-slicing mechanism (rescheduling on each timer interrupt) to put an upper bound on the amount of time that a task can use the CPU continuously if other tasks are waiting to use it. This policy ignores the static priorities used to discriminate among tasks in the other two policies. Instead, a dynamic priority is computed on the basis of

the value assigned to the task by the **nice()** or **setpriority()** system call and by the amount of time that a process has been waiting for the CPU to become available. The counter field in the task descriptor becomes the key component in determining the task's dynamic priority. It is adjusted on each timer interrupt (when the interrupt handler adjusts the various timer fields for the task).

The following annotated code fragment from **kernel/sched.c** represents the main flow of the scheduler; most error checking and obscure cases have been removed from this fragment. The C++ style annotations have been added as explanation in this manual and do not appear in the Linux source code.

```
    /* if all runnable processes have "counter == 0", re-calculate
     * counters
     */
    if (!c)
        goto recalculate;
...
// Finally, you are ready to dispatch task next
    kstat.content_swtch++;
    get_mmu_context(next);
    switch_to(prev, next, prev);
    __schedule_tail(prev)—;
    ...
    return;

recalculate:
    ...
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
    goto repeat_schedule;

still_running:
    c = prev_goodness(prev, prev, this_cpu);
    next = prev;
    goto still_running_back;

handle_bh:
    do_bottom_half();
    goto handle_bh_back;
    ...
move_rr_last:
    if (prev->counter) {
        prev->counter = prev->priority;
        move_last_runqueue(prev);
    }
    goto move_rr_back;

    ...
}
```

```
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    int c;
    struct task_struct *prev, *next, *p;
    int this_cpu, c;

    ...
    prev = current;
    this_cpu = prev->processor;
    ...
// Here is where the bottom halves and task
// queues get executed
// Logically, this is part of ret_from_sys_call
    if (bh_active & bh_mask)
        goto handle_bh;
handle_bh_back:
    ...
    /* move an exhausted RR process to be last.. */
    if (prev->counter == SCHED_RR)
        goto move_rr_last;
move_rr_back:

    switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:
    }
    prev->need_resched = 0;

repeat_schedule:

/* this is the scheduler proper: */

    p = init_task.next_run;
    c = -1000;
    if(prev->state == TASK_RUNNING)
        goto still_running;

still_running_back:
    ...
// This loop finds the task that has the highest counter (dynamic
// priority value; that is the task that will be dispatched
    while (p != &init_task) {
        if (can_schedule(p)) {
            int weight = goodness(p, prev, this_cpu);
            if (weight > c)
                c = weight, next = p;
        }
        p = p->next_run;
    }
```

Stride Scheduling

A process requests a certain share of the total resource consumption allocating a corresponding fraction of tickets. Each ticket represents a minimum amount of CPU cycles. Possession of a share of the total number of allocated tickets results in the right to consume a corresponding fraction of processing time in the course of time. A process owning twice as many tickets as another process will receive a share twice as large. The amount of tickets allocated by a process is stored as a simple numerical value. This value is selected randomly in the creation of a process.

Each process is associated with a dynamic property which denotes an indicator for scheduling relative to other processes. This property is called the **pass** of a process. Allocations are performed by selecting the process with the currently minimal **pass** value. After consuming its quantum the process' **pass** value is advanced by an increment which is inversely proportional to the amount of tickets allocated. As a consequence, passes of processes advance with inversely proportional speed of their allocations. This increment is called a **stride**. To keep integer values, we multiply the inverse proportional with a large integer constant **int_multiplier**:

stride = int_multiplier / tickets

As several clients might feature equal passes at scheduling time, the criterion for selection of minimum pass must be extended. The process id therefore may serve as an adequate solution. A simple example may illustrate the basic principles of ***Stride*** scheduling. Let **int_multiplier** = 200 and three processes A, B, and C have ticket allocations according to the following table:

Process	**Tickets**	**stride**
A	**2**	**100**
B	**1**	**200**
C	**5**	**40**

With passes of all processes initially reset to zero and further assuming that process ids match the alphabetical order of the processes, allocations will be performed as illustrated in the table together with the figure. After seven allocations, the proportions of 2:1:5 have been achieved and the eighth allocation reflects the state before the first allocation.

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Every space is a quantum. Black color is process A, gray color is process B and white color is process C.

Problem Statement

You are to modify the Linux scheduler to support **Stride** scheduling and then measure the performance of the system with and without the modified scheduler so that you can compare the effect of the new scheduler.

Part A

Add a new scheduling policy to support **Stride** scheduling. To simplify the problem, have your new scheduler govern all processes using **Stride** scheduling. That is when your system is running the **Stride** scheduler, the **sched_setscheduler()** system call will have no effect.

Part B

Modify the kernel to add a new system call that enables or disables the utilization of the designed scheduler. Refer to the documentation files related to system calls.

Part C

Write instrumentation software so that you can obtain detailed performance data regarding the scheduler's behavior. Study the behavior of your **Stride** scheduler, and report on its performance. Discuss whether your implementation works as you expected during the design phase.

Planning a Solution

The introductory explanation of the scheduler gives you the background information that you need to solve this problem. You need to modify **kernel/sched.c** to implement your scheduler. You probably may need to modify other components, such as the task descriptor or user descriptor. Design your strategy for implementing the **Stride** scheduler so that it has as little impact on the existing code and data structures as possible, even if you have to sacrifice some performance. After you get the new scheduler to work properly, you can reconsider the design from the performance perspective.

Comparing Scheduler Performance

For performance inspections, set up an experiment using a few CPU-intensive processes (for example, processes that simply execute a very long ***for*** loop, doing arithmetic operations). Start several instances of the CPU-intensive process, and then try to see how the CPU time is distributed among the different processes.

For this purpose, you can exploit the timer information that is available to each process. Linux provides each process with task timers that can be read via **getitimer()** system call (see its man page). You can write your CPU-intensive load generator program so that it reads the task timers and then reports each timer value to a file. You can analyze the collective files after you have run your experiment. After you have exploited the explained approach, you should insert new instrumentation code into the kernel sources, along with a new module to compare and contrast the **Stride** scheduler with the Linux default scheduler.

Hints, Suggestions, and Recommendations

To be successful in this project, you need to understand the Linux scheduler well. Make sure that you can answer the following questions:

- What is the current scheduling mechanism? Be specific and thorough. (It should be possible to re-implement the behavior of the current scheduler from your description, which should be a kind of specification.)
- Is starvation possible with the current mechanism? If so, give an example of how it might occur.
- Is there aging? That is, are the priorities of processes that have low recent CPU consumption raised to avoid effective starvation?
- Are I/O bound processes (those whose ratio of I/O to CPU use is high) given any kind of favoritism?
- Suppose that all processes on the system are scheduled using SCHED_OTHER, and that all are in tight CPU loops. Give an expression that indicates the fraction of CPU time that will be allocated to a single process, as a function of its base ("nice") priority. (Note: The answer to this depends on at least one Linux configuration parameter.)

It is very inconvenient to have a hard bug in the scheduler - the machine simply won't run your copy of the kernel, and you'll spend days rebooting the system. For that reason, it is recommended that you

- think hard about your code before you recompile and install
- make modifications incrementally, not all at once. If the machine fails to boot or to run, it's a good guess that the last set of changes you made is the problem. (It's not certain, of course.)

Making sure that the set of changes is small can help find the problem. Additionally, you might find that instrumenting the code (i.e. printk's) helps you, and even building some additional tools (e.g., a new system call, and a user-level application that invokes it) is worth the effort. Before working on a source file, it would be wise to make a copy of it. For example, in the kernel directory, just use *"cp sched.c sched.c.orig"*.

Following is a nice CPU-intensive C code, if repeated continuously:

```
i = (i + 1) % 1000;
```

As a simple performance comparison, you would report a CPU time histogram of 10 concurrent running processes under both the original scheduling policy and the one you have implemented. Other nice metrics are throughput and average waiting time.

Submitting your work

The final demonstration date will be announced on the website of the course. Have your formal report ready at the demo date, including your solutions to the proposed problems, and the results (including several graphs, charts, etc.) of your solutions.

Zip the source codes you modify and upload them to the YULearn.

In order to demonstrate your effort on the project, carefully prepare and plan your final demo. Divide the demo content equally among your partners. You will not be explaining each and every line of code you have written.

At the final demo, each group member has to prepare a folded piece of paper, indicating the performance of the group members, by means of percentages of total work done, including herself/himself.

You are supposed to obey the academic honesty rules posted on the web page of the course. Discarding them may cause you trouble.

Happy codings,
Osman Kerem Perente