HY-345: Operating Systems
Winter Semester 2025
Assignment 4

Implementation of the “Least Tolerance First” Scheduling Policy in the Linux Operating System

Tutorial: 25/11/2025
Submission: 16/12/2025

Introduction

The kernel of the Linux operating system contains a scheduler that decides which process will be executed next on the computer's processor (CPU). The scheduler makes decisions according to the scheduling policy, which helps to ensure efficient use of the processor. In this assignment, you will implement a new scheduling policy as part of the Linux operating system.

Least Tolerance First

In this assignment, you are asked to implement the "Least Tolerance First" (LTF) scheduling algorithm. According to the algorithm, each process must declare its deadline **D** by which it wants to be completed, as well as its estimated execution time **E**. The "Least Tolerance First" algorithm dynamically prioritizes processes based on the time remaining until the deadline and how much execution time remains in the process.

According to the Least Tolerance First algorithm, each process is characterized by a tolerance value **T**, which determines which process will be executed first. The value T is defined as $T = \frac{D - Current\ Time}{Remaining\ Execution\ Time}$, where D is the deadline of the process, Current Time is the current system time, and Remaining Execution Time is the computational time remaining for the process to execute. The Least Tolerance First algorithm always chooses to execute the process with the smallest T value.

In each scheduling decision, the scheduler calculates the value T for all processes that have declared a deadline and execution time and selects the process with the smallest value T to be executed. A small T value means that the process is more critical because its deadline is approaching, so it must be executed immediately.

The Least Tolerance First algorithm is also responsible for the execution time of each process. When the algorithm has allowed a process to run for the time it has declared, then the algorithm must terminate (kill) the process and print an appropriate message. If a process fails to execute for the entire execution time it initially declared and the deadline it declared has passed, then the scheduler must terminate it and print an appropriate message. Finally, the algorithm is **preemptive**. This means that if there is a process that has a higher priority according to the Least Tolerance First policy, then the scheduler will interrupt the current process and replace it with the one with the higher priority.

Execution Example

Let us consider two different processes, P_1 and P_2. Process P_1 specifies an execution time of 5s and a deadline of 10s. Process P_2 specifies an execution time of 4s and a deadline of 11s. We assume that both processes start their execution simultaneously at time 0s. Therefore, the scheduler must now decide which process to execute.

1. At time t=0
   - Process P_1 has $\mathrm{T} = \frac{10-0}{5} = 2$
   - Process P_2 has $\mathrm{T} = \frac{11-0}{4} = 2.75$

2. The scheduler decides that process P_1 has higher priority, so it will select it for execution.

3. At time t=1
   - Process P_1 has $\mathrm{T} = \frac{10-1}{4} = 2.25$
   - Process P_2 has $\mathrm{T} = \frac{11-1}{4} = 2.5$

4. The scheduler decides that process P_1 still has higher priority, so it will continue to execute it.

5. At time t=2
   - Process P_1 has $\mathrm{T} = \frac{10-2}{3} = 2.666$
   - Process P_2 has $\mathrm{T} = \frac{11-2}{4} = 2.3$

6. The scheduler decides that process P_2 now has higher priority, so it must take time on the processor. Therefore, it interrupts process P_1 and replaces it with P_2.

The execution of the processes continues according to the rules of the Least Tolerance First algorithm until there are no more processes that can be executed (either because they have completed their execution time or because their deadline has passed).

Modifications to the Linux kernel

For this assignment, you will use the code from Assignment 3 as a basis. In addition, you will use the QEMU emulator and the virtual disk image you used in Assignment 3. Instructions on compiling the Linux kernel and using the virtual disk image can be found in the previous assignment.

For this assignment, you need to modify the operating system kernel and implement the new scheduling policy. For this policy to work properly, you will use the system calls you implemented in the previous assignment. The main function of the Linux Scheduler and the entry point is the `void __sched schedule(void)` function in the `kernel/sched.c` file.

The steps to follow are:

1. The scheduler identifies the processes that use the LTF policy by checking which of them have set a deadline D and execution time E.
    - These processes have priority over the others running on the system. If there is no "privileged" process, the scheduler does not need to change its behavior and therefore uses the default scheduling algorithm.
2. For each of the "privileged" processes, it calculates how much time they have already been running on the processor.
3. For each process, it calculates the remaining execution time R. This time depends on the time the process has already been running and the total time it has requested to run.
4. If the process has exceeded its deadline, the scheduler terminates it. If a process has already completed the execution time it requested, the scheduler terminates it.
5. For each of the processes, it calculates the value T. The process with the lowest value T is selected by the algorithm to be executed.
    - In this case, the scheduler will interrupt the lowest priority process already running on the processor in order to execute the selected process.

In the LTF algorithm, a process with a T value less than 1 has less time until the deadline than the execution time remaining. This means that it cannot meet its deadline. In this assignment, you can handle such cases in one of the following ways:

1. The scheduler terminates the process immediately because it cannot meet its deadline.
2. The scheduler ignores the specific process and selects another one that may have a higher T value.
3. The scheduler normally selects the specific process, knowing that it will not meet its deadline and that it may delay another one.

Implementation

The goal of this assignment is to implement the requested scheduling algorithm. The implementation you choose is up to you, and there is no single correct implementation. Here are some useful structures and functions that may help you.

File	Entity	Description
include/linux/sched.h	struct task_struct	Process descriptor. Each process is represented as such a struct. It offers all the information about one particular task (i.e. process) such as pid, state, parent process, children, opened files, etc.
	struct rq	Runqueue. It is the main data structure in process scheduling. It manages active processes by holding the tasks that are in a runnable state at any given moment of time.
	struct sched_entity	CFS works with more general entities than tasks. This struct contains attributes for accounting run time of processes.
	struct sched_class	The current Linux scheduler has been designed with an extensible hierarchy of modules in mind. These modules encapsulate scheduling policy details. Scheduling classes are implemented through the sched_class structure, which contains hooks to the functions that implement the policy.
kernel/sched.c	schedule(void)	Main function of the Linux scheduler. Responsible for implementing the process scheduling functionality.
	void context_switch(...)	Performs the actual context switch operation by switching from the old task_struct to the new one.
	pick_next_task(...)	Selects task_struct of the next process that will run on the processor. Iterates over the list of processes in the runnable state.

In addition, you can examine the `kernel/sched_rt.c` and `kernel/sched_fair.c` files, which implement the Real-Time Scheduling Class and the Completely Fair Scheduler, respectively. Their implementations can help you understand how the Linux scheduler works and how process management is done. Finally, the `include/linux/time.h` file contains various structs for measuring time, as well as various functions for converting between units of measurement.

Remarks

- Make sure that your implementation does not starve the other processes in the system. You can achieve this either by setting appropriate parameters via system calls or by switching between policies selected in the scheduler.

- For this assignment, you will use the Linux kernel 2.6.38.1. You can use the Elixir platform to browse the Linux Kernel code

- To get the task_struct of the current process that made the system call, you can look in the file `arch/x86/include/asm/current.h`

- The Linux kernel stores detailed information about all current processes in a list of task_struct objects. To access all system processes in the list, you can use the for_each_process macro.

- Use the printk function to check that your implementation is working correctly. To view these messages, you can use the `dmesg` utility tool or simply run the command "`cat /var/log/messages`".

Demo Programs

You must create and submit at least one demo program that uses the system calls from the previous assignment and demonstrates the correct implementation of the new scheduling algorithm. Here are some examples of cases you can test.

1. Build a simple program that calls set_proc_info (from the previous assignment) and sets the scheduling parameters. Check that your implementation handles these parameters correctly and can correctly calculate the value T as defined by the LTF algorithm.
2. Build a simple program that sets its parameters and make sure that your implementation of the algorithm is called correctly when it detects such a process.
3. Create multiple processes that define different parameters for each one and make sure that your implementation works correctly and correctly selects which process to execute first according to their priority.
    - Your processes must require the use of the processor (spinning) and not be in sleep state. You can achieve this by using a loop and arithmetic operations.
4. Set a specific execution time for a process and let it run for at least that long. Check that when the process runs for the requested time, the scheduler terminates it.
5. For each test program you create, use the appropriate prints that show the parameters set by each process and when it starts spinning. These prints will help you understand if your implementation is making the right decisions.

Submission

After completing the assignment, you must submit the following:

1. The new kernel image resulting from the compilation: linux-2.6.38.1/arch/x86/boot/bzImage.
2. All the files you needed to modify or create in the Linux kernel source code to implement the system calls. This means that you will submit all the .c, .h, Makefile, etc. files that you modified or created. Please note that you should **not** submit files that you did not need to modify for your implementation (e.g., the rest of the kernel source tree).
3. The code from all the test programs you wrote and ran in the guest Linux OS to test the system calls you implemented. Also, any header files you used for type and function definitions, as well as any Makefiles needed to compile these programs. You do not need to submit the executable files.
4. A README file in which you briefly (but comprehensively and clearly) describe all the steps you followed to add and implement the new system calls. You should also comment on what you observed from the test programs you ran. If you did anything different or more than what is mentioned in the assignment description at any step, you can also mention it in the README. Due to the complexity of this assignment, it is recommended that you mention in the README any implementation attempts you made, even if they did not lead anywhere or did not work properly.
5. You can create a directory with the modified kernel files (if you want, it would be good to keep the file structure within the kernel) as well as a directory with the test programs and header files from the guest OS.

Please note:

1. You do <u>**NOT**</u> need to submit the disk image (hy345-linux.img) even if it has been modified. The disk image may indeed change while you are using the guest OS, but you do not need to submit it.
2. You do <u>**NOT**</u> need to submit a file containing the entire Linux kernel source code. You should only note and submit the files that you have modified or created. The kernel image (bzImage), source and header files, and test programs you submit should be sufficient for your assignment to run with the original disk image and QEMU so that the correct implementation of the assignment can be seen.

Resources

You can use the following resources to better understand how the Linux Scheduler works, as well as its various components and functions.

[1] Completely Fair Scheduler
[2] Kernel Scheduler Documentation
[3] Columbia University - Linux Scheduler
[4] Linux Processes and Scheduling
[5] Linux Data Structures
[6] Process Management and Process Descriptor
[7] A study on Linux Kernel Scheduler version 2.6.32
[8] A complete guide to Linux process scheduling
[9] Kernel Scheduling Classes Documentation
[10] Completely Fair Scheduler Internals