Linux

Jiafeng

May 5, 2024

1 Introduction

The Linux operating system is an open-source, multi-user, multi-tasking operating system. It was first released by Linus Torvalds in 1991. Through the joint efforts of many developers around the world, it has become one of the most popular server operating systems in the world today. The main features of Linux include security, flexibility, and high performance while supporting a variety of hardware platforms. As one of the Linux distributions, Ubuntu is famous for its ease of use, and beautiful user interface. Since its first release in 2004, Ubuntu has become one of the first choices for individual users. This article will explore the design and implementation of Linux. The topic will include the design and implementation of process and thread management, CPU scheduling, and memory management, and compare and analyze it with the content learned in class.

1.1 new stuff list

1. sibling 2.1
2. task_traced 2.1
3. ”sub-reaper” mechanism 2.2 , similar to orphan

2 Process management mechanism

Linux’s process management mechanism has undergone many improvements and extensions based on the traditional Unix system to adapt to more complex and diverse application scenarios.[1] While these improvements improve system performance and flexibility, they also introduce some trade-offs.

First, Linux uses the task_struct structure to describe a process or thread, which contains many important process attribute information. For example [2],

(1) the process ID (pid) is used to uniquely identify a process

(2) the parent process ID (parent_pid) is used to maintain the hierarchical relationship between processes,

(3) the memory map (mm) describes the virtual address space of the process,

(4) the signal processing function (signal_struct) stores a pointer to the signal handling structure for the process

2.1 task strcut info

In addition, task_struct also contains some fields for maintaining process relationships, such as children maintaining a list of all child processes, and sibling is used to connect child processes with the same parent process. Sibling processes usually share some resources, and can easily find each other through the sibling field, communicate and synchronize operations [2]. These detailed process description information and relationship maintenance mechanisms make process management more efficient and conducive to the execution of complex tasks. However, the task_struct structure is relatively large, which may increase the system's memory overhead. Especially when creating a large number of processes, this overhead may become a performance bottleneck.

Compared with the classic process state model, Linux introduces some new process states to meet more complex process management needs. In addition to the traditional running state (task_running), interruptible sleep state (task_interruptible) and uninterruptible sleep state (task_uninterruptible), Linux also introduces the following states [3]:

(1) task_traced : The process is being tracked by the debugger so that the debugger can single-step or set breakpoints, which we don't talk about in class.

(2)task_stopped: The process has stopped executing, such as receiving a SIGSTOP signal.

(3)exit_zombies: The process has terminated, but its parent process has not yet obtained its exit status. At this time, the process occupies system resources but cannot be scheduled.

(4)exit_dead: The process has terminated and its exit status has been obtained by the parent process. At this time, the process resources can be reclaimed by the system.

2.2 trace & "subreaper"

These new process states provide the Linux kernel with a more sophisticated mechanism for controlling process behavior, but they also introduce some trade-offs. Take the task_traced state as an example. It allows the debugger to single-step and set breakpoints in the process. This is very useful for development and debugging and can help programmers find and solve problems. Without this state, the debugger cannot implement these functions, or can only implement them in a more complex and inefficient way. However, the introduction of the task_traced state also increases the complexity of the kernel. The kernel needs to consider this state in scenarios such as process switching and signal processing and interact with the debugger. This requires more code logic and may introduce new bugs and security vulnerabilities.

An orphan process is a process which the parent process exits before the child process, causing the child process to become an "orphan". What we learned in class is that orphan processes will be automatically adopted by the init process (a special process with PID 1). The init process will periodically call the wait() function to clean up these adopted zombie processes. This applies to traditional Unix systems, but this method also has some problems:

When the init process adopts an orphan process, the original parent process can no longer obtain information such as their exit status. This is a problem for applications that need to monitor

the running status of child processes (such as service managers). If there are too many orphan processes, it may put too much burden on the init process and affect the stability of the system. In some special cases (such as multi-threaded programs) [5], orphan processes may not be adopted by the init process, but by other processes, which makes the process relationship complicated. To solve these problems, modern Linux systems have introduced some new mechanisms, such as ”subreaper” [4]. A process can mark itself as a ”subreaper” through the prctl() system call and become the ”custodian” of its child processes. When the process exits, its child processes will be adopted by the ”custodian” process instead of the init process. This can avoid overloading the init process and improve system stability. At the same time, the ”custodian” process can also obtain information about the adopted process through functions such as wait() to monitor and manage it. However, these new mechanisms also introduce some additional overhead and complexity.

3 Thread management mechanism

Linux began to introduce the concept of ”kernel-level threads”, that is, to create a corresponding lightweight process (LWP) for each user-level thread in the kernel. In this way, the kernel can be responsible for thread scheduling and synchronization. In this model, the task_struct structure not only represents the process, but also the thread.

However, this one-to-one thread model also has some problems, such as the high overhead of creating threads and the need to frequently switch between user mode and kernel mode. To further optimize the performance of threads, Linux introduced NPTL (Native POSIX Thread Library) in version 2.6. NPTL uses a lighter data structure, which significantly reduces the overhead of thread operations.[7]

At the same time, to avoid storing too much thread-related information in task_struct, Linux introduced the thread_info structure. This structure stores some meta-information of the thread, such as a pointer to the corresponding task_struct, CPU affinity, expected scheduling class, etc. Therefore, in modern Linux systems, the implementation of threads is a hybrid model.[6]

4 CPU scheduling policies

Linux uses the Completely Fair Scheduler (CFS) as its default scheduling policy. CFS aims to allocate fair CPU time slices to each process or thread while minimizing scheduling overhead. It uses a red-black tree to track all runnable processes/threads and selects the process/thread with the smallest virtual running time to execute.

CFS itself is not a typical multi-level queue scheduler, but it achieves similar effects by introducing the concepts of dynamic priority and scheduling class. The dynamic priority mechanism is similar to the priority and aging techniques we learned in class. Each process/thread has a dynamic priority, whose initial value is based on the static priority (i.e. nice value) of the process/thread.[8&9] During operation, the dynamic priority is adjusted according to the actual behavior of the process/thread. The basic idea is to calculate the bonus value through variables such as the interactivity of the process/thread and the sleep time calculation. For interactive processes/threads, the bonus value is higher, thereby increasing their dynamic priority. In addition to dynamic priority, CFS also introduces the concept of scheduling classes. Several scheduling

classes are defined in Linux, such as real-time scheduling class (RT), fair scheduling class (fair), idle scheduling class (idle), etc. Each scheduling class has its priority range and scheduling policy. The priorities and scheduling policies of these scheduling classes are similar to the different queues in multi-level queue scheduling. Each scheduling class can be regarded as a separate queue with different priorities and scheduling algorithms.

For I/O-bound processes/threads, CFS will suspend the increment of their virtual run time. By doing this, it can prevent them from obtaining too many CPU time slices. When the I/O is completed, CFS will recalculate their virtual run time to ensure that they get fair scheduling opportunities.

5 Memory management

5.1 Memory Mapping

Memory Mapping is a powerful mechanism in Linux memory management. It allows processes to map files to their own virtual address space. This mechanism relies on the support of virtual memory and paging mechanisms. Compared with traditional file I/O, memory mapping provides a more efficient way to access data. In traditional file I/), processes need to use system calls such as read()/write() to read and write file data. That process requires data copying between user space and kernel space, which costs space and time. When using memory mapping, processes can directly access file data without going through system cells, thereby improving the efficiency of file I/O. [11]

Memory mapping needs mmap() system call.[10] When we use this system call, we need to provide parameters, such as starting address, length, and permissions. When we call the mmap() system calls for memory mapping, the operating system does not immediately read the data from the disk. Instead, it sets up a mapping area in the virtual address space of the process. And creates corresponding page table entries. These page table entries are initially not associated with any actual physical memory pages. When the process first accesses these virtual pages, a page fault exception is triggered. It will load the data from the file and update the page table. Memory mapping uses the lazy loading strategy of OS, which not only improves efficiency but also reduces unnecessary memory usage.

5.2 Access control mechanism

Linux kernel provides an access permission mechanism to ensure the security of memory. Each page has corresponding access permission bits, such as read, write, execute, etc. These permission bits are stored in the page table entry. When a process accesses a virtual page, the MMU will automatically check the access permissions of the page. If the access type(such as read, write, or execute)does not match the page permissions, the MMU will trigger an access exception. The kernel's exception handler will capture and handle it.

6 I/O& AI tech

In modern operating systems, I/O scheduling algorithms play a key role in improving system performance and response time. Traditional I/O scheduling algorithms, such as Noop, Deadline, CFQ, etc. [13], are mainly based on predefined rules and heuristics to sort and merge I/O requests. However, these algorithms are often difficult to adapt to complex and dynamically changing I/O loads, resulting in poor performance and waste of resources.

In recent years, with the development of machine learning technology, especially the success of deep learning in various fields, researchers have begun to explore the application of AI technology in I/O scheduling optimization. Through machine learning algorithms, especially deep neural networks, it is possible to learn and extract hidden patterns and rules from massive I/O tracking data, dynamically predict future I/O requests, and adjust scheduling strategies accordingly.

However, training these complex machine learning models often requires extensive computing resources and time. This is where the NPU (Neural Processing Unit) comes into play. NPU is a hardware accelerator specially designed for deep learning and artificial intelligence applications. It provides highly parallel and optimized matrix computing capabilities, which can significantly accelerate the training and inference process of neural networks [12].

References

1. Difference between UNIX and Linux - Guru99
2. A Brief on Linux Process by Amit Nadiger on LinkedIn
3. Process States in Linux Kernel - HumbleC
4. What are Subreapers in Linux - OpenSourceForU
5. What is a subreaper process? - Unix Stack Exchange
6. Difference between NPTL and POSIX threads - StackOverflow
7. Native POSIX Thread Library - Wikipedia
8. Process Scheduling - Scaler Topics
9. Fair Scheduling in Linux - OpenSource.com
10. mmap - The Open Group
11. Understanding Mapping - IBM Documentation
12. Neural Processing Unit (NPU) Explained - Utmel
13. Linux I/O Scheduler Tuning - CloudBees Blog