

Linux 进程调度相关功能及性能测试

郭小康¹, 翟高寿¹, 寇思旭¹, 罗琼²

(1. 北京交通大学 计算机科学与技术学院; 2. 北京交通大学软件评测实验室, 北京 100044)

摘要: 进程调度是操作系统的核心功能, 为整个系统的质量保障提供关键支撑作用, 开展相关功能及性能测试非常必要。现有进程调度测试主要基于黑盒测试方法和 glibc 库函数, 存在测试用例不够完备及冗余问题, 而依托于假想进程创建的性能测试结果也不够精准。鉴于此, 提出基于系统调用的进程调度功能测试方案, 采用白盒测试方法及判定覆盖准则, 综合考虑相关函数调用链中的所有内核函数, 并通过功能归类处理和测试用例精简策略, 进而在保证测试完备性的同时还减少了测试用例冗余。基于实际进程开展性能测试, 系统分析进程状态转换时间节点, 利用 eBPF 的 kprobes 与 tracepoint 机制实现了对应内核函数及跟踪点地址的精准探测, 从而改善了各类时延指标的测试结果。其中, kworker、gnome、xdg 等进程及其子进程的上下文切换时延明显较长, 说明相关系统程序代码或系统配置存在一定优化空间。

关键词: Linux; 进程调度; 系统调用; 功能测试; 性能测试

DOI: 10.11907/rjdk.241836

中图分类号: TP316

文献标识码: A

文章编号: 1672-7800(2026)001-0095-08

扫描二维码阅读全文:



Functional and Performance Testing Related to Linux Process Scheduling

GUO Xiaokang¹, ZHAI Gaoshou¹, KOU Sixu¹, LUO Qiong²

(1. School of Computer Science and Technology, Beijing Jiaotong University;

2. Software Evaluation and Testing Laboratory of Beijing Jiaotong University, Beijing 100044, China)

Abstract: Process scheduling is the core function of the operating system, providing critical support for the overall system's quality assurance. It is essential to carry out relevant functional and performance testing. Existing process scheduling test is mainly based on black box testing methods and glibc library functions, which has the problems of incomplete and redundant test cases, and the performance test results based on the creation of hypothetical processes are not accurate enough. Therefore, the process scheduling functional test scheme based on system calls is proposed in this paper. The scheme adopts white box testing methods and decision coverage criteria, comprehensively considers all kernel functions in the relevant function call chain. Meanwhile, through functional classification processing and test case streamlining strategies, it reduces the redundancy of test cases while ensuring the completeness of the test. This paper conducts performance tests based on actual processes, systematically analyzes the process state transition time nodes, and uses the kprobes and tracepoint mechanisms of eBPF to accurately detect the corresponding kernel functions and tracepoint addresses, thereby improving the test results of various latency indicators. Among them, context switch latency of processes such as kworker, gnome and xdg, as well as their child processes, is significantly longer, indicating that there is room for optimization in the relevant system program code or system configuration.

Key Words: Linux; process scheduling; system call; functional testing; performance testing

0 引言

操作系统是计算机系统的基础系统软件, 为保证操作

系统的高效性、可靠性以及真正做到操作系统自主可控, 必须开展完备的功能测试和性能测试^[1-3]。进程调度是操作系统的核心功能, 为整个系统的质量保障提供关键支撑作用, 开展相关功能及性能测试尤为必要。现有对应功能

收稿日期: 2024-11-25

基金项目: CCF-深信服伏羲基金项目 (CCF-SANGFOR OF 2022001)

作者简介: 郭小康 (2001-), 女, 北京交通大学计算机科学与技术学院硕士研究生, 研究方向为 Linux 内核测试; 翟高寿 (1971-), 男, 博士, 北京交通大学计算机科学与技术学院副教授、硕士生导师, 研究方向为操作系统、系统安全与测试; 寇思旭 (2000-), 男, 北京交通大学计算机科学与技术学院硕士研究生, 研究方向为 Linux 内核测试; 罗琼 (1983-), 女, 北京交通大学软件评测实验室高级工程师, 研究方向为系统与软件测试、云测试、敏捷测试、铁路软件测试等。本文通讯作者: 翟高寿。

测试如 stress-ng 主要基于 glibc 库函数如 sched_setscheduler、sched_setattr、sched_setparam 等开展黑盒测试,且存在测试用例不够完备和冗余问题^[4]。有关性能测试如 SystemTap、Perf 通常局限于特定场景或假想进程的创建,测试结果不够精准^[5-6]。为此,本文基于系统调用(涵盖函数调用链中的所有内核函数)设计进程调度的功能测试方案,采用白盒测试方法,通过设置系统调用的参数以覆盖相应函数的判断语句并根据判定覆盖准则设计测试用例,同时依据功能归类处理和测试用例精简策略,从而保证测试的完备性且减少了测试冗余。在性能测试方面,本文系统分析进程调度有关指标计算时间节点与进程状态转换节点之间的对应关系,利用 eBPF(extended Berkeley Packet Filter)的 kprobes 与 tracepoint 机制对相应内核函数及跟踪点地址进行探测,利用 eBPF 提供的 hash 数据结构记录对应的时间戳和进程 pid,进而得到更为精准的测试结果。

1 相关工作

目前,进程调度测试的主要对象为 CFS 完全公平调度器,且大多为性能测试。Lan 等^[7]、Wu 等^[8]实现了 ScheduleBench,但有关方法依赖于进程创建与进程间管道通信。McVoy 等^[9]构建了 Lmbench,测试进程创建销毁时间开销与上下文切换性能,但上下文切换时延仍依赖于假想进程创建。Molnar 等^[6]设计了 Perf,可以跟踪和记录进程之间的上下文切换次数与进程调度延迟等。Oliveira 等^[10]跟踪线程调用栈,对实时 Linux 系统进行调度延迟测试,并提供了一种机制以评估和改进实时 Linux 系统的性能,但存在从栈中读取内容不全面和跟踪复杂等问题。当前,进程调度机制功能测试研究较少,主要集中于新调度算法研究。余飞^[11]指出在系统内部开展调度算法研究受到源码的使用限制和代码结构复杂所带来的高昂实现成本的限制,提出一种模块化调度器框架,通过将调度器代码封装到 10 个模块中,以减少各部分代码间的相互关联,提高代码清晰度。Jiang 等^[12]通过变异测试以验证 HRNN 调度算法的正确性,但由于 Linux 系统默认调度器为 CFS,源码量大且数据结构复杂,故此方法并不适用。stress-ng 对进程调度相关系统调用封装的库函数采用黑盒测试,未深入内核源码进行代码验证^[4]。

本文拟定的性能测试技术路线不再依赖于假想进程创建,而是基于系统实际进程,并建立了性能测试指标记录时间点与进程状态相关内核函数的对应关系。Kprobes、Tracepoint、PerfEvent 等技术可以实现内核探测,但均依赖于可加载内核模块,可能引入新的时间开销且给内核带来不安全问题^[13]。相比之下,eBPF 可以在操作系统内核中运行沙盒程序,在不修改内核源代码的情况下,通过加载和执行特定的 eBPF 程序以扩展内核的功能。其最初应用于网络数据包捕获与分析,现可用于安全监控、

系统跟踪、性能分析等^[14]。Gebai 等^[15]指出与其他探测内核的工具相比,eBPF 为用户提供了多种数据结构,例如简单的数组和哈希,这使其成为实时监控的绝佳工具。eBPF 内部集成了 Kprobes 与 Tracepoint 探测机制,可深入内核精准探测。同时,eBPF 程序运行在内核态却无法对内核函数进行调用与修改,使其广泛应用于 Linux 安全方面^[16]。昌武洋等^[17]基于 eBPF 与 LSTM 构造了一套 DDos 攻击检测系统,为网络异常流量检测提供了一种新的解决方法。Tian 等^[18]提出了 LBM(Linux (e)BPF Module),利用 eBPF 包过滤机制,提供了一个统一的 API 以加强对 Linux 内核中恶意外设的保护。王凯龙等^[19]提出一种结合 eBPF 与 KFENCE 的 Linux 内核态内存漏洞检测方法。王子成等^[20]提出一种基于 eBPF 的内核堆漏洞动态缓解框架,该框架充分利用 eBPF 的动态性、探测性及安全特性,实现对内核数据的跟踪,在运行时,它会将空间随机化对象注入内核,使得攻击者无法准确放置攻击。在进程调度性能测试层面,eBPF 的前端工具 BCC(BPF Compiler Collection)对进程调度相关函数进行探测,测试进程调度时延^[21]。综上所述,eBPF 可为进程调度性能测试提供轻量级可靠支持。

2 进程调度功能测试

2.1 功能测试整体设计

目前,进程调度相关功能测试仅针对系统调用相关部分 glibc 库函数开展黑盒测试,并未深入研究内核源码,且对功能相同的库函数重复测试,存在测试不够完备且测试冗余问题。为此,本文针对进程调度机制相关系统调用开展功能测试,更具通用性。首先,对功能相同系统调用进行归类,对同类系统调用内部函数链与源码的判断语句进行分析;其次,设置参数以覆盖函数调用链中所有可控的判断语句,依据判定覆盖准则开展白盒测试,同时对函数调用链相同的同类系统调用精简测试用例,仅测试能涵盖所有且最多测试用例的系统调用;最后,设计并实现了自动化执行程序,其中对需要 root 权限的某些测试用例,提供交互接收密码方式以保障程序自动化执行。

2.2 相关系统调用功能分类

进程调度相关系统调用共 15 个,本文根据功能将其划分为以下 6 类:与进程 nice 值相关、与进程调度属性相关、进程与 CPU 亲和力相关,以及与调度策略优先级相关、与 CPU 为进程分配时间片相关、与进程主动放弃 CPU 相关。系统调用原型及功能与参数说明如下:

(1) 与进程 nice 值相关系统调用。

int nice(int increment);

功能说明:修改进程 nice 值,使其在现有基础增加 increment

参数说明:increment 为 nice 的增加值

int setpriority(int which, int who, int niceval);

功能说明:修改指定进程的 nice 值为 niceval

参数说明:which 和 who 共同指定进程(which 为 PRIO_PRO-

CESS、PRIO_PGRP、PRIO_USER 分别表明 who 为进程、进程组或用户标识符,且 who 为 0 时为当前进程、当前进程组或当前用户的标识符);niceval 为进程 nice 值,取值为-20~19

int getpriority(int which, int who);

功能说明:获取进程 nice 值

参数说明:which 和 who 共同指定进程(具体含义同上)

(2)与进程调度属性相关系统调用。

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);

功能说明:设置指定进程调度策略与实时优先级

参数说明:设置指定 pid 进程的进程调度策略为 policy(policy 为 SCHED_NORMAL、SCHED_FIFO、SCHED_RR、SCHED_BATCH、SCHED_IDLE、SCHED_DEADLINE 分别为 CFS、FIFO、RR、BATCH、IDLE、DEADLINE 调度策略)、进程调度参数为 param,sched_param 结构体定义如下

```
struct sched_param {
```

```
    int sched_priority;
```

```
};
```

sched_priority 保存实时调度策略 SCHED_FIFO、SCHED_RR 进程实时优先级值,值限定为 1~99,1 表示优先级最低,99 表示最高;若为 SCHED_DEADLINE 策略设置为-1,其他调度策略设置为 0

int sched_setparam(pid_t pid, const struct sched_param *param);

功能说明:设置指定进程实时优先级为 param 中 sched_priority

参数说明:设置指定 pid 进程的实时优先级,param 具体含义同上

int syscall(SYS_sched_setattr, pid_t pid, struct sched_attr *attr, unsigned int flags);

功能说明:设置指定进程调度属性

参数说明:设置指定 pid 进程的调度属性为 attr,flags 无意义留作拓展使用

sched_attr 结构体定义如下:

```
struct sched_attr {
```

```
    __u32 size; // sched_attr 结构体大小
```

```
    __u32 sched_policy; // 调度策略
```

```
    __u64 sched_flags; // 调度标志
```

```
    /* SCHED_NORMAL, SCHED_BATCH */
```

```
    __s32 sched_nice; // nice 值
```

```
    /* SCHED_FIFO, SCHED_RR */
```

```
    __u32 sched_priority; // 实时优先级
```

```
    /* SCHED_DEADLINE */
```

```
    __u64 sched_runtime; // DEADLINE 调度策略进程运行时间
```

```
    __u64 sched_deadline; // DEADLINE 调度策略进程运行截止时间
```

```
    time
```

```
    __u64 sched_period; // DEADLINE 调度策略进程运行周期
```

```
    __u32 sched_util_min; // CPU 最大利用率
```

```
    __u32 sched_util_max; // CPU 最小利用率
```

```
};
```

```
int sched_getscheduler(pid_t pid);
```

功能说明:获取指定进程调度策略

参数说明:进程 pid

```
int sched_getparam(pid_t pid, struct sched_param *param);
```

功能说明:获取指定进程实时优先级

参数说明:获取指定 pid 进程的实时优先级封装在 param 中返回,param 含义同上

```
int syscall(SYS_sched_getattr, pid_t pid, struct sched_attr *attr, unsigned int size, unsigned int flags);
```

功能说明:获取指定进程调度属性

参数说明:获取指定 pid 进程调度属性封装在参数 attr 中返回,attr、flags 含义同上,size 为 attr 大小

(3)进程与 CPU 亲和力相关系统调用。

```
int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

功能说明:将指定进程绑定 CPU 的掩码 mask

参数说明:将指定 pid 进程绑定掩码值为 mask 的 CPU,cpusetsize 为掩码 mask 长度

```
int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

功能说明:获取指定进程绑定的 CPU 的掩码 mask

参数说明:获取指定 pid 进程绑定的 CPU 掩码值封装在 mask 中返回,cpusetsize 为掩码长度

(4)进程调度相关系统调用分类。

```
int sched_get_priority_max(int policy);
```

功能说明:获取指定调度策略的最大优先级

参数说明:policy 为指定进程调度策略

```
int sched_get_priority_min(int policy);
```

功能说明:获取指定调度策略的最小优先级

参数说明:policy 为指定进程调度策略

```
int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

功能说明:获取指定进程单次运行时间片

参数说明:获取指定 pid 进程单次运行时间片封装在参数 tp 中返回,timespec 结构体定义如下:

```
struct timespec {
```

```
    time_t tv_sec; /* seconds */
```

```
    long tv_nsec; /* nanoseconds */
```

```
};
```

```
int sched_yield(void);
```

功能说明:进程主动放弃 CPU 运行

2.3 同类系统调用函数调用链与判定覆盖分析

本文采用白盒测试方法及判定覆盖准则,综合考虑了相关函数调用链中的所有内核函数,并精简函数调用链相同的系统调用测试用例,再保证测试完备性的同时还减少了测试用例冗余。下文通过对 sched_setscheduler、sched_setparam、sched_setattr 的分析以详细描述本文测试方案设计。

如图 1 所示,内核文件 kernel/sched/core.c 中,系统调用 sched_setscheduler 与 sched_setparam 调用了同一函数 do_sched_setscheduler,该函数内部调用链为:do_sched_setscheduler->sched_setscheduler->_sched_setscheduler->

__sched_setscheduler。sched_setscheduler、sched_setparam、sched_setattr 等系统调用功能类似且源码内部均是将参数封装到 sched_attr 类型的变量 attr 中,将 attr 作为参数传递给 __sched_setscheduler 函数。

```

SYSCALL_DEFINE3(sched_setscheduler, pid_t, pid, int, policy, struct sched_param __user *, param)
{
    if (policy < 0)
        return -EINVAL;
    return do_sched_setscheduler(pid, policy, param);
}
SYSCALL_DEFINE2(sched_setparam, pid_t, pid, struct sched_param __user *, param)
{
    return do_sched_setscheduler(pid, SETPARAM_POLICY, param);
}
static int __sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param, bool check)
{
    .....
    if ((policy != SETPARAM_POLICY) && (policy & SCHED_RESET_ON_FORK)) {
        attr.sched_flags |= SCHED_FLAG_RESET_ON_FORK;
        policy &= ~SCHED_RESET_ON_FORK;
        attr.sched_policy = policy;
    }
    return __sched_setscheduler(p, &attr, check, true);
}

```

Fig. 1 Codes of sched_setscheduler and sched_setparam

图1 sched_setscheduler、sched_setparam 源码

从参数层面查看测试用例设计,系统调用 sched_setscheduler 的参数 1 进程 pid,测试用例为当前 pid 与非法 pid,参数 2 代表设置调度策略。因此,测试用例需要涵盖上文提出 5 种调度策略与非法调度策略。参数 3 中定义 sched_priority,调度策略为 SCHED_RR、SCHED_FIFO 的取值是 1~99,测试应该包含边界值内、边界值与边界值外;调度策略为 SCHED_DEADLINE 的取值为-1,测试用例应设置-1 与非-1 值,其他策略应设置 0 或者非 0 值进行验证。sched_setparam 系统调用的参数与 sched_setscheduler 参数大致相同,调用链中 policy 设置为 SETPARAM_POLICY 表示仅设置 sched_param 而不修改 policy。

从判定覆盖层面看,if 语句(1)主要是对 policy 的合法性进行检查,进程调度 5 种调度算法对应关系如下:测试用例涵盖 policy 为合法与非法的情况,if 语句(2)中 (policy != SETPARAM_POLICY) 是对 policy 的合法性进行检查,内核中 SETPARAM_POLICY 设置为-1, (policy & SCHED_RESET_ON_FORK) 是对标志进行检查,如果父进程指定 SCHED_RESET_ON_FORK 标志,则对其创建子进程。有如下规定:①若父进程调度策略为 SCHED_FIFO 或 SCHED_RR,则该策略将在子进程中重置为 SCHED_NORMAL;②若父进程的 nice 值为负值,则子进程中的 nice 值将重置为零;③若父进程为 SCHED_DEADLINE 调度策略,则无法创建子进程。

因此,在设置调度策略系统调用的 SCHED_FIFO、SCHED_RR 调度策略应该考虑规定(1),对 nice 系统调用测试用例设计应对应涵盖 SCHED_RESET_ON_FORK 标志涉及的规定(2),同时单独测试规定(3)加以验证。

同类系统调用 sched_setattr(位于 kernel/sched/core.c) 内部调用函数链为 sched_setattr(p, &attr) -> __sched_setscheduler(p, attr, true, true),依据内核函数调用链与判定覆盖准则对 sched_setattr 系统调用参数进行限制以设计测试方案。pid 设置与上文相同,flags 无含义为后续拓展

使用,attr 为调度相关属性,内部不同变量的取值与调度策略设定相关,本文在涉及测试用例时按调度策略分类。sched_attr 为调度属性信息(见 2.2 节)。从内核源码分析 sched_attr 中的变量约束,sched_flags 为控制调度行为的标志,取值如下:①SCHED_FLAG_RESET_ON_FORK:fork 子进程时重置调度属性;②SCHED_FLAG_RECLAIM:任务正在被回收或释放;③SCHED_FLAG_DL_OVERRUN:实时任务的 deadline 超时;④SCHED_FLAG_KEEP_POLICY:保持现有策略不变;⑤SCHED_FLAG_KEEP_PARAMS:保持 sched_param 不变;⑥SCHED_FLAG_UTIL_CLAMP_MIN:用于限制进程的 CPU 利用率下限,若设置该标志,进程的 CPU 利用率将不会低于指定的最小值(与 sched_attr 中 sched_util_min 相关);⑦ SCHED_FLAG_UTIL_CLAMP_MAX:用于限制进程的 CPU 利用率上限,若设置该标志,进程的 CPU 利用率将不会超过指定的最大值(与 sched_attr 中 sched_util_max 相关);⑧SCHED_FLAG_SUGOV:表示使用特定的 CPU 频率调整调度器(它通常用于内核级别的调度,而不是用户进程)。sched_flags 可测试标志位 SCHED_FLAG_RESET_ON_FORK、SCHED_FLAG_KEEP_POLICY、SCHED_FLAG_KEEP_PARAMS、SCHED_FLAG_UTIL_CLAMP_MIN、SCHED_FLAG_UTIL_CLAMP_MAX,同时 SCHED_FLAG_UTIL_CLAMP_MIN、SCHED_FLAG_UTIL_CLAMP_MAX 与 sched_util_min、sched_util_max 进行关联测试。

综上所述,sched_setattr 系统调用参数 sched_attr 中属性的设置涵盖了 sched_setscheduler、sched_setparam 参数与判定覆盖的测试用例,且三者系统调用底层调用函数均为将参数封装 sched_attr 类型的 attr,调用 __sched_setscheduler 函数,因此对 sched_setattr 进行测试,可包含 sched_setscheduler 与 sched_setparam 所有测试用例。由于 sched_attr 内其他变量与调度策略具有相关性,在测试过程中根据 sched_setattr 的参数 sched_attr.sched_policy 分类开展测试。此外,由于 sched_flags 取值较复杂,需单独测试。

本文对系统调用的功能测试上,测试用例设计不应只是关注系统调用定义函数 sched_setattr,应分析函数调用链中所有涉及函数如 __sched_setscheduler(位于 kernel/sched/core.c)的所有判断语句中的逻辑条件,依据判定覆盖准则设计测试用例。

2.4 功能测试用例设计

根据上文对 sched_setattr 系统调用函数链中全部判断语句中逻辑条件的分析,整合可控参数约束如表 1 所示,由于 sched_attr 内属性与 sched_policy 相关,按 sched_policy 分类,对 sched_flags 单独测试。

依据判定覆盖准则,对可控的参数约束进行了真假值验证,进而对进程调度相关系统调用展开测试。其他类系统调用测试方案设计思路与之类似,均基于参数约束的真假覆盖,实现对函数调用链内部判断语句的覆盖,进行开始完备测试并精简用例。对于 yield 这类无参系统调用,无

Table 1 Parameter constraints of sched_setattr system call		
表 1 sched_setattr 系统调用参数约束		
sched_attr	分类	参数约束
sched_attr	SCHED_NORMAL/ SCHED_BATCH	普通用户可增加 nice 值
		普通用户无法降低 nice 值
	SCHED_FIFO/ SCHED_RR	sched_priority=0
		(1)sched_priority(限制范围 1~99)
		(2)sched_flags 中含有 SCHED_FLAG_RESET_ON_FORK, 会在父进程 fork 子进程时, 子进程在继承父进程的调度参数时, 会将调度策略重置为 SCHED_NORMAL, 同时将 nice 值重置为 0
		(1)sched_runtime、sched_deadline、sched_period 三者关系:
		①sched_runtime≤sched_deadline≤sched_period
		②sched_runtime≥1024
		③sched_deadline!=0
		(2)sched_priority 为-1
		(3)对 SCHED_FLAG_RESET_ON_FORK 标志:
sched_policy	SCHED_DEADLINE	①SCHED_DEADLINE 调度策略的进程无法 fork 新进程;
		②若含有 SCHED_FLAG_RESET_ON_FORK 标志, fork 后子进程调度策略重置为 SCHED_NORMAL
	SCHED_IDLE	普通用户无法修改调度策略为其他策略
	SCHED_FLAG_KEEPPOLICY	保持调度策略不变
	SCHED_FLAG_KEEPPARAMS	保持实时优先级 sched_param 参数不变
	sched_flags	若包含该标志表明可根据用户设置的 sched_util_min 与 sched_util_max 设置 CPU 利用率限制;
		SCHED_FLAG_UTIL_CLAMP_MIN
	SCHED_FLAG_UTIL_CLAMP_MAX	sched_util_max 与 sched_util_min 限制:
		(1)sched_util_max≤1024
		(2)sched_util_min≤sched_util_max

法从参数覆盖测试用例, 本文采取的方法是创建一批进程, 让进程运行一段时间后主动放弃 CPU, 并观察进程切换状态。

2.5 测试自动化程序设计

本文对系统调用功能测试部分进行整合并设计自动化测试程序, 如图 2 所示。

第一层目录是需要测试的系统调用分类, 如设置调度属性类、进程 nice 值类、CPU 亲和力等。此外, 该层目录还包含 Makefile 文件, 用于对相关文件进行编译及配置测试环境; 同时包含 auto.sh 脚本, 用于执行自动化测试。该层的 Makefile 文件可递归调用第二层目录中的 Makefile 文件, 同时编译同一类系统调用中的所有测试程序。第二层目录是分调度策略对 sched_setattr 进行测试。由于某些测试用例需 root 权限, 因此在测试脚本编写过程中, 采取交互接收密码方式以保障程序自动化执行。

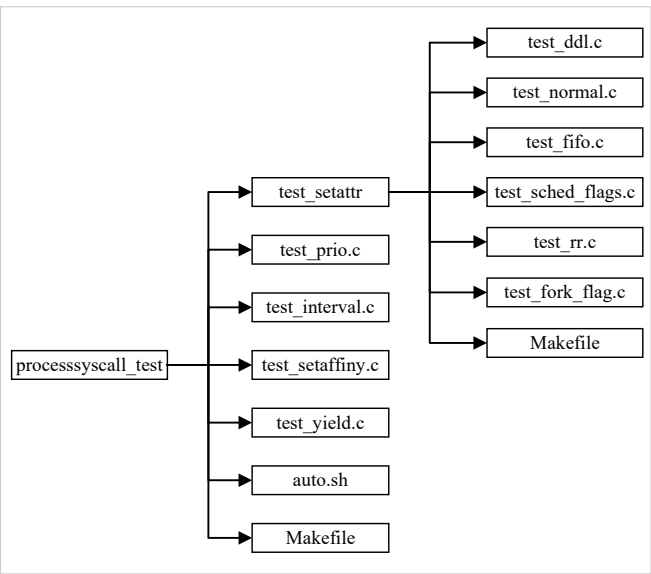


Fig. 2 Directory structure of system call test automation program
图 2 系统调用测试自动化程序目录结构

3 进程调度性能测试

3.1 性能测试整体设计

本文利用 eBPF 的 Kprobes 与 Tracepoint 机制, 精准开展性能测试工作。具体而言, 从宏观和微观两个维度确定性能测试指标, 并将这些测试指标与进程状态转换节点对应。通过探测与状态转换相关的内核函数及其地址, 利用 eBPF 内置的 Hash 数据结构存储探测指标的开始与结束时间戳, 进而计算出相应的测试指标。这种测试方法不仅更加精准, 而且由于 eBPF 虽运行于内核态, 但仅具备探测功能, 无法对内核进行修改, 因此在保证测试精准性的同时, 不会给内核带来安全风险。

3.2 进程状态转换与性能指标相关性分析

本文将进程状态转换作为探测时间节点, 如图 3 所示, 在内核的进程控制块结构体 task_struct 中, 其字段 state 定义了 7 种状态。进程状态转换与触发事件如下:

(1)TASK_RUNNING。进程创建后进入就绪队列处于该状态, 表示正在运行或准备运行(就绪、运行), 为区分 TASK_RUNNING 就绪与 CPU 运行状态, 两种状态的区分以调度函数 schedule 执行为节点。

(2)TASK_INTERRUPTIBLE。其表示可中断的睡眠状态, 进程正在等待某个事件, 当事件发生时或者有信号时会被唤醒, 进程在 CPU 运行状态与该状态的切换节点为 wait_event_interruptible 函数的执行。

(3)TASK_UNINTERRUPTIBLE。其表示不可中断的睡眠状态。进程正在等待某个事件, 必须等待资源到来才会被唤醒, 只有信号是无法被唤醒的, 进程在 CPU 运行状态与该状态的切换节点为 wait_event 函数的执行。

(4)TASK_STOPPED。向进程发送 SIGSTOP/SIGTIN/SIGSTP/SIGTOU 等信号, 它就会响应该信号并进入

TASK_STOPPED 状态;向进程发送一个 SIGCONT 信号,可以让其从 TASK_STOPPED 状态恢复到 TASK_RUNNING 状态。

(5) TASK_TRACED。进程暂停下来,等待跟踪它的进程对其进行操作。

(6) TASK_DEAD。进程退出,资源被回收,剩 task_struct 未被回收。

(7) EXIT_DEAD。进程所有资源被回收,进程销毁。

本文将进程性能测试指标划分为两类,以更全面地展

示进程运行效率和系统整体性能。从宏观角度关注整个系统的综合性能和进程在整个生命周期中的性能表现,包括周转时间、响应时间等指标;从微观角度关注进程层面的性能细节,包括进程调度时延、进程上下文切换时延、进程唤醒时延等。本文测试涵盖以上指标,利用进程状态转换时间节点以计算性能测试指标,进程状态的改变对应性能测试指标计算的开始与结束时间,如周转时间为进程创建处于就绪状态到进程销毁状态的时间差,进程调度时延为进程由就绪状态转换为运行状态时间差。

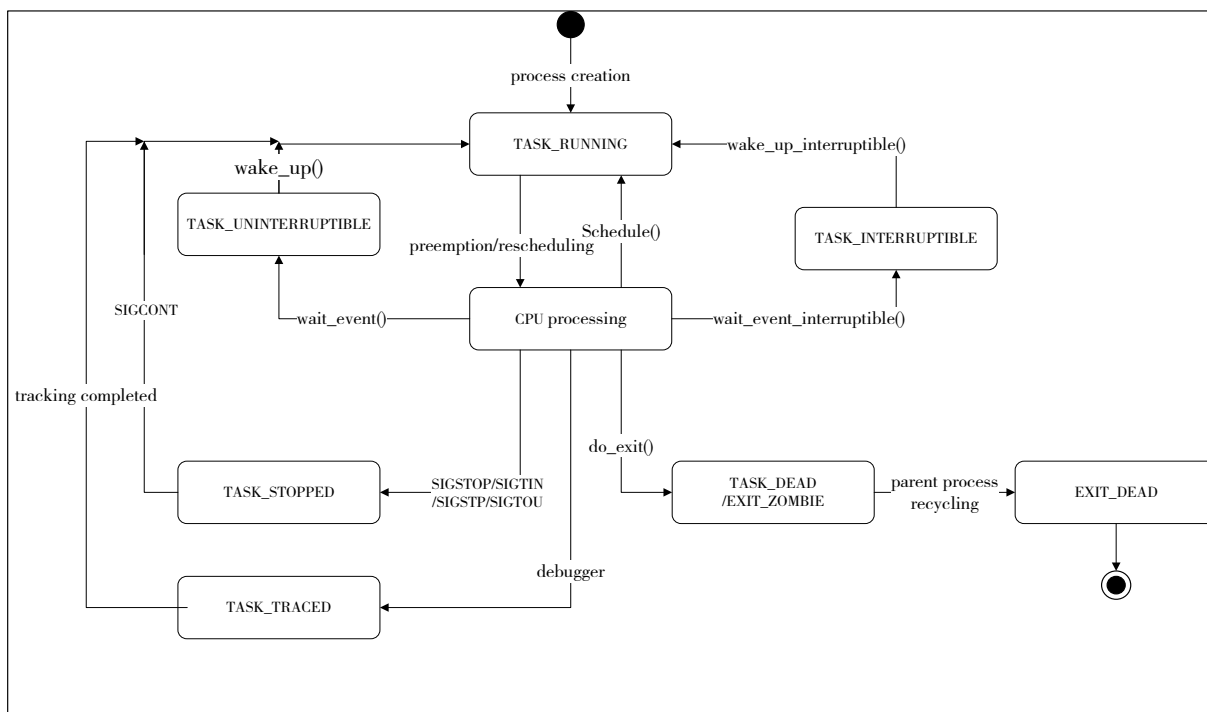


Fig. 3 Process state transition diagram

图3 进程状态转换图

3.3 基于静态追踪与 Kprobes 探测机制的性能指标测试

以进程上下文切换时延为例,如图4所示,将静态追踪点 SCHED_SWITCH (对应内核函数为 trace_sched_switch) 作为切换开始节点,将 Kprobes 探测 finish_task_switch 函数的返回时刻作为切换完成节点,计算时间间隔得到进程上下文切换时延。

```

static void __sched notrace __schedule(unsigned int sched_mode)
{
    if (likely(prev != next)) {
        .....
        trace_sched_switch(sched_mode & SM_MASK_PREEMPT, prev, next);
        rq = context_switch(rq, prev, next, &rf);
        .....
    }
    .....
}

static __always_inline struct rq *context_switch(struct rq *rq,
struct task_struct *prev, struct task_struct *next, struct rq_flags *rf)
{
    .....
    return finish_task_switch(prev);
}
  
```

Fig. 4 Function __schedule and context_switch

图4 __schedule 函数与 context_switch 函数

3.4 性能指标计算方法设计

如图5所示,以进程上下文切换时延为例,将基于静态追踪点 SCHED_SWITCH 探测进程切换为其他进程时间节点,利用 eBPF 内置函数 bpf_ktime_get_ns 获取时间戳,并从 ctx 寄存器中获取 prev 与 next 进程,使用 eBPF 提供的数据结构 hash 记录,在切换完成后从 Hash 查找开始时间节点,作差即可。其他指标类似,如进程唤醒时延为 Kprobe 探测 dequeue_task 函数或静态追踪点 SCHED_SWITCH 记录该进程出队时间,与 Kprobe 探测 wakeup 函数或静态追踪点 SCHED_WAKE_UP 记录进程被唤醒的时间差;进程周转时间为基于 Kprobe 探测 kernel_clone 函数记录进程入队时间,与 do_exit 函数记录进程运行结束的时间差;进程调度时延为进程入队 Kprobe 探测 enqueue_task 函数,与静态追踪点 SCHED_SWITCH 的时间差。

本文采取 eBPF 对内核实现精准探测,计算进程性能测试指标,并通过 BCC 前端内置 C 语言代码的方法,直接运行在内核态,与向内核添加内核模块相比开销更小。同时,eBPF 内部有安全检测机制,无法直接调用与修改内核

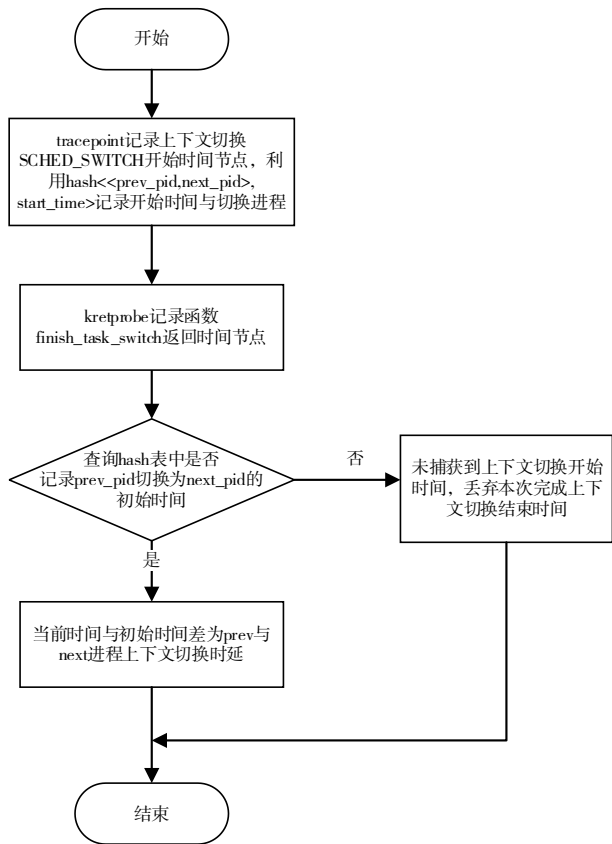


Fig. 5 Process context switch latency computation
图5 进程上下文切换时延计算

函数, 不会带来安全问题。

4 实验与结果分析

本文测试实验环境为 Ubuntu20.04 系统, 内核源码版本为 Linux5.15.0。

4.1 功能测试结果

如表 2 所示, 本文测试涵盖系统调用共 15 个, 设计并运行测试用例 62 个, 全部通过测试。通过 stress-ng 中的 stress_sched_policy 测试模块, 针对 glibc 库函数开展测试工作, 采用黑盒测试设计相应的测试用例, 不涉及内核源码审计与验证, 测试库函数为 8 个, 测试用例总数为 26 个, 对库函数的测试未覆盖所有参数的所有取值情况。本文系统调用测试方法创新点在于: 覆盖测试用例全面, 从内核源码出发, 依托参数与条件语句, 开展白盒测试, 覆盖函数调用链中的全部判断语句, 同时对功能相似内部函数调用链相同的系统调用测试用例进行简化, 在保证测试完备性的同时减少了测试用例冗余。

4.2 性能测试结果

本文对系统内进程的上下文切换时延、唤醒时延、调度时延、周转时间等测试指标进行了测试。以进程上下文切换时延为例, 如图 6 所示, 包含进程 pid 与名称 (python-1857)、进程运行的 cpu 编号 (001)、系统启动后时间戳

Table 2 Comparison with stress-ng on process scheduling function testing

表 2 与 stress-ng 进程调度功能测试工作对比		
测试分类	本文方法测试用例数	stress-ng 测试用例总数
进程 nice 值	8	0
进程调度属性	35(涉及所有调度策略与参数)	24(调度策略未涉及 SCHED_DEADLINE 等, 且未涉及对 sched_flags 等参数测试)
获取调度策略优先级限制大小	7	0
获取进程在每次使用 CPU 时分分配到时间片的长度	6(涉及所有调度策略与非法调度策略)	1(仅涉及 SCHED_RR)
进程主动放弃 CPU	1	1
进程绑定 CPU 的掩码	5	0
mask		
测试用例数总计	62	26

(325.196 945)、测试程序自定义输出进程上下文切换时延结果。



Fig. 6 Test result of process context switch latency
图6 进程上下文切换时延测试结果

由于进程上下文切换时延更能反映系统的并发处理能力, 本文额外对上下文切换时延作特殊点分析, 统计 1 小时内进程的平均上下文切换时延, 如图 7 所示, 进程上下文切换时延主要集中在 20 us 以内。

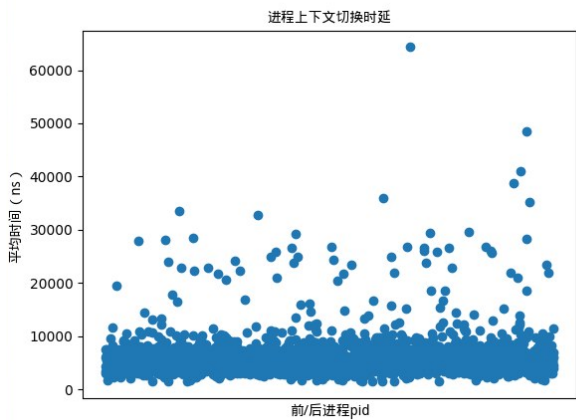


Fig. 7 Scatter plot of process context switch latency
图7 进程上下文切换时延散点图

通过多次实验对进程上下文切换时延特殊点位进行提取发现, 进程上下文切换时延较长的进程主要是与 kworker、gnome、xdg 相关的进程。kworker 是 Linux 内核的工作线程, 用于异步处理工作队列中的任务, 这些任务包括处理硬件中断、文件系统事件、管理系统内存等。可能

是该类进程正在处理大量的任务或等待某些资源,导致系统响应变慢。gnome 和 xdg 相关进程与 Linux 桌面环境相关,可能是因为桌面环境正在执行复杂的图形渲染或处理用户交互。此外,还有 systemd-journald 等进程,它负责收集、存储和管理系统日志,帮助用户查看系统运行状态和故障排查。同时,update-notifier 相关进程的上下文切换时延较大,此类进程主要用于通知用户系统中可用的软件更新。它会提示用户安装重要的系统更新和安全补丁。这类进程上下文切换时延较长,可能是由于它们正在执行资源密集型任务或等待某些资源。为降低时延,后续需要优化这些进程相关代码或调整系统配置。

传统方法对进程调度性能测试依托假想进程的创建,可能引入系统调用等额外开销,测试不够准确。本文性能测试的创新之处在于直接对系统内已有进程进行测试,基于 eBPF 的 Kprobe 与 Tracepoint 机制探测内核中的函数并计算指标,且 eBPF 程序直接运行在内核态,测试更加精准。

5 结语

本文针对已有进程调度测试存在测试用例不够完备及冗余问题,提出了基于系统调用的进程调度测试方案,并对进程调度相关功能与性能开展了测试。在功能测试方面,本文方法建立在系统调用的基础上,采用白盒测试方法及判定覆盖准则,对函数调用链中的全部判断语句进行分析并开展测试,保证了测试的完备性,并通过功能归类处理和测试用例精简策略,减少了测试用例冗余。在性能测试方面,本文将进程性能指标与状态转换时间节点相关内核函数对应,利用 eBPF 的 Kprobes 与 Tracepoint 机制对其进行精准探测,改善了各类时延结果,并对上下文切换时延较长的进程展开分析,为后续进程代码优化与系统配置提供了支持。

后续可瞄准 Linux 进程调度性能提升目标,通过分析进程上下文切换时延等性能测试结果,研究相关内核代码优化方案。对进程调度相关功能测试可在系统调用测试的基础上,扩展到面向进程全生命周期内核函数链的白盒测试,使测试更加完备且具有普遍性。此外,本文测试还可扩展到 openEuler、KylinOS 和 UOS,为国产操作系统测试平台构建提供有力支撑。

参考文献:

- [1] ZHANG J L. Analysis of the industrialization status and discussion on the development of domestic operating systems[J]. Science and Technology & Innovation, 2016, 3(20): 27-29.
张京林. 国产操作系统产业化现状分析和探讨[J]. 科技与创新, 2016, 3(20): 27-29.
- [2] MOGOSANU L, CARABAS M, CONDURACHE C, et al. Evaluating architecture-dependent Linux performance[C]//Conference on Control Systems and Computer Science, 2015: 499-505.
- [3] BORAS M, BALEN J, VDOVJAK K. Performance evaluation of Linux operating systems[C]//International Conference on Smart Systems and Technologies, 2020: 115-120.
- [4] ColinlanKing. stress-ng[EB/OL]. <https://github.com/ColinlanKing/stress-ng>.
- [5] EIGLER F C. Problem solving with systemtap[C]//Ottawa Linux Symposium, 2006: 261-268.
- [6] MOLNAR I. Perf[EB/OL]. <https://perf.wiki.kernel.org/index.php>.
- [7] LAN Y Q, XU H, LIU X H. The research of performance test method for Linux process scheduling[C]//International Symposium on Information Science and Engineering, 2012: 216-219.
- [8] WU W, YAO X, FENG W, et al. Research on improving fairness of Linux scheduler[C]//Yinchuan: IEEE International Conference on Information and Automation (ICIA), 2013.
- [9] MCVOY L. Lmbench[EB/OL]. <http://www.bitmover.com/lmbench>.
- [10] OLIVEIRA B, DANIEL C, DANIEL O, et al. Demystifying the real-time Linux scheduling latency[C]//Paris: Euromicro Conference on Real-Time Systems (ECRTS), 2020.
- [11] YU F. Research on the structure and algorithm of operating system scheduler[D]. Changsha: National University of Defense Technology, 2011.
余飞. 操作系统调度器结构及算法研究[D]. 长沙: 国防科学技术大学, 2011.
- [12] JIANG M Y, CHEN T Y, KUO F C, et al. Testing central processing unit scheduling algorithms using metamorphic testing[C]//Beijing: IEEE 4th International Conference on Software Engineering and Service Science, 2013.
- [13] Kprobes/Tracepoint/Perfevent: Kprobes/Tracepoint/Perfevent [EB/OL]. <https://docs.kernel.org/trace>.
- [14] JIANG O N. Design and implementation of network packet capture and analysis system based on eBPF[D]. Wuhan: Huazhong University of Science and Technology, 2020.
姜欧涅. 基于 eBPF 的网络数据包捕获与分析系统的设计与实现[D]. 武汉: 华中科技大学, 2020.
- [15] GEBAI M, DAGENAIS M. Survey and analysis of kernel and userspace tracers on Linux: design, implementation, and overhead[J]. ACM Computing Surveys, 2018, 51(2): 1-33.
- [16] LIU W, LIAO P. Research on the application of eBPF technology in dynamic tracking of operating system[J]. China Financial Computer, 2022(8): 81-84.
刘伟, 廖平. eBPF 技术在操作系统动态跟踪中的应用研究[J]. 中国金融电脑, 2022(8): 81-84.
- [17] CHANG W Y, FU X, WANG J C. DDoS attack detection system based on eBPF and LSTM[J]. Journal of Chongqing Technology and Business University (Natural Science Edition), 2023, 40(2): 36-43.
昌武洋, 付雄, 王俊昌. 基于 eBPF 与 LSTM 的 DDoS 攻击检测系统[J]. 重庆工商大学学报(自然科学版), 2023, 40(2): 36-43.
- [18] TIAN D J, HERNANDEZ G, CHOI J I, et al. LBM: a security framework for peripherals within the Linux kernel[C]//San Francisco: IEEE Symposium on Security and Privacy, 2019.
- [19] WANG K L. Research on a Linux kernel memory vulnerability detection method combining eBPF and KFENCE[J]. Electronic World, 2021(16): 95-97.
王凯龙. 一种结合 eBPF 与 KFENCE 的 Linux 内核态内存漏洞检测方法的研究[J]. 电子世界, 2021(16): 95-97.
- [20] WANG Z C, GUO Y G, ZHONG B N, et al. Kernel heap vulnerability dynamic mitigation mechanism based on eBPF[J]. Journal of Software, 2024, 35(7): 3332-3354.
王子成, 郭迎港, 钟炳南, 等. 基于 eBPF 的内核堆漏洞动态缓解机制[J]. 软件学报, 2024, 35(7): 3332-3354.
- [21] BCC. BCC[EB/OL]. <https://github.com/iovisor/bcc>.

(责任编辑: 孙娟)