

# 操作系统专题实践 - 实验1

71123329 段嘉文 2025-12-07

## Linux进程管理及其扩展

### 操作系统专题实践 - 实验1

Linux进程管理及其扩展

实验目的

实验内容

具体要求

设计思路

主要数据结构及其说明

编译 Linux 内核

新增 `hide` 系统调用

1. 修改 `task_struct`

2. 修改 `procfs`

3. 添加系统调用及相关接口

编写测试程序及测试

实验体会

参考文献

Linux 内核概述与进程管理

内核编译与配置

系统调用实现

`procfs` 文件系统

内核开发技术细节

系统启动与 GRUB 配置

其他参考资料

## 实验目的

通过本次实验，深入理解操作系统中进程管理的核心机制，掌握进程控制块（`task_struct`）、进程队列等关键数据结构在 Linux 内核中的实现方式。进一步通过扩展内核功能，实现自定义系统调用，提升对系统调用机制、进程调度与进程可见性控制的理解，增强内核级编程能力。

## 实验内容

1. 阅读并分析 Linux 5.13.10 内核源代码，重点研究 `task_struct` 结构体以及进程链表的组织形式（如 `tasks` 链表），理解进程在内核中的表示与管理方式。
2. 在 Linux 内核中添加一个新的系统调用 `hide(pid_t pid)`，该系统调用可根据传入的进程 ID 将指定进程从进程列表中移除，使其不再被用户态工具（如 `ps`、`top` 等）所列举，从而实现进程隐藏功能。
3. 编写用户态测试程序，验证系统调用的功能正确性。

## 具体要求

- 修改内核源码，在合适位置实现 `sys_hide` 系统调用函数，并注册至系统调用表。
- 隐藏操作应通过将目标进程从全局进程链表（`init_task.tasks`）中解链实现，确保其不会被遍历到。
- 提供反隐藏机制（可选扩展），或通过重启恢复进程可见性。
- 编译并启动修改后的内核，确保系统基本功能正常运行。
- 测试新系统调用的行为，验证 `ps` 和 `top` 命令无法显示被隐藏的进程。
- 记录实验过程，撰写完整报告，包含代码说明、设计思路、测试结果及分析。

## 设计思路

本实验的设计基于 Linux 内核中进程管理的数据结构与模块化控制思想。传统的进程隐藏方法通常依赖于从 `tasks` 链表中移除目标进程节点，从而使其无法被 `ps`、`top` 等工具枚举。然而，该方法存在恢复困难、易破坏链表完整性等问题。为此，本实验采用一种更为安全、可控的设计方案：不直接修改进程链表，而是通过状态标记结合遍历过滤的方式实现逻辑上的“隐藏”。

### 1. 引入隐藏标志字段（`hidden_flag`）：

在 `task_struct` 结构体中添加一个自定义字段 `int hidden_flag;`，用于标识该进程是否应被隐藏。初始值为 0，设置为 1 表示该进程处于隐藏状态。

### 2. 实现系统调用 `sys_hide(pid_t pid)`：

用户可通过该系统调用传入指定 PID，内核查找对应进程并将其 `hidden_flag` 标记为 1。查找过程使用 `find_vpid()` 和 `pid_task()` 组合完成，确保正确获取 `task_struct` 指针。

### 3. 通过 `/proc` 文件系统提供运行时控制接口：

- 创建 `/proc/hide` 节点，支持读写操作：
  - 读取：返回当前全局隐藏功能开关状态（1 表示启用过滤，0 表示关闭）。
  - 写入：允许用户态通过 `echo "0" | sudo tee /proc/hide` 等方式动态启停隐藏功能。

- 创建 `/proc/hidden_process` 节点，用于列出当前所有 `hidden_flag == 1` 的进程 PID，便于调试与监控。

#### 4. 在进程列举路径中插入过滤逻辑：

修改内核中与 `/proc` 文件系统相关的进程遍历函数（如 `proc_pid_readdir` 或 `proc_fill_cache`），在返回每个进程信息前检查其 `hidden_flag` 状态及全局开关。若开关开启且目标进程被标记，则跳过该进程，使其不出现在 `/proc` 目录下，进而不会被 `ps`、`top` 等工具读取。

#### 5. 用户态测试程序验证功能：

编写两个测试程序：

- `hide_test.c`: 接收 PID 参数，调用 `sys_hide()` 将其标记为隐藏。
- `hide_user_processes_test.c`: 批量隐藏特定条件下的用户进程（可选扩展）。测试流程包括：
  - 执行前后的 `ps aux` 对比；
  - 使用 `dmesg` 输出调试日志确认系统调用执行情况；
  - 动态启停隐藏功能并观察进程可见性变化。

#### 6. 权限与安全性控制：

所有涉及系统调用和 `/proc` 写操作均需 `CAP_SYS_ADMIN` 权限，确保只有 root 用户可以修改隐藏状态，防止普通用户滥用。

#### 7. 优势与特点：

- 非侵入式设计：不修改 `tasks` 链表，避免破坏内核数据结构导致崩溃。
- 可逆性强：通过清除 `hidden_flag` 即可恢复可见，无需重启。
- 动态可控：通过 `/proc/hide` 实现全局开关，灵活应对调试与部署需求。
- 可观测性好：通过 `/proc/hidden_process` 可审计当前隐藏进程列表。

## 主要数据结构及其说明

本实验主要涉及以下核心数据结构和变量的修改与扩展：

### 1. `task_struct` 结构体扩展

`task_struct` 是 Linux 内核中用于描述进程的核心数据结构，定义在 `include/linux/sched.h` 中。本实验对该结构体进行了扩展，新增了 `cloak` 字段：

```
struct task_struct {
    // ... existing fields ...
    int cloak; // 进程隐藏标志: 0表示正常显示, 1表示隐藏
    // ... existing fields ...
};
```

字段说明：

- **cloak**: 整型标志位，用于标识当前进程是否应被隐藏
  - 值为 0：进程正常显示，可被 `/proc` 遍历和用户态工具（如 `ps`、`top`）检测到
  - 值为 1：进程处于隐藏状态，在全局开关 `hidden_flag` 启用时不会出现在进程列表中
- 初始化时机：在 `kernel/fork.c` 的 `copy_process()` 函数中，每个新创建的子进程的 `cloak` 字段被初始化为 0，确保新进程默认可见

## 2. 全局隐藏开关 `hidden_flag`

定义在 `fs/proc/hide.c` 中的全局变量，用于控制进程隐藏功能的整体启停状态：

```
int hidden_flag = 1; // 全局隐藏功能开关
EXPORT_SYMBOL(hidden_flag); // 导出符号供其他模块使用
```

变量说明：

- 作用域：通过 `EXPORT_SYMBOL` 导出，可被内核其他模块（如 `fs/proc/base.c`）访问
- 初始值：设为 1，表示隐藏功能默认启用
- 运行时控制：通过 `/proc/hide` 接口可动态修改，实现隐藏功能的开关切换
- 判断逻辑：只有当 `hidden_flag == 1` 且进程的 `cloak == 1` 时，该进程才会被过滤

## 3. 进程过滤机制相关函数

在 `fs/proc/base.c` 中修改了两个关键函数，实现进程列表的过滤逻辑：

### (1) `proc_pid_readdir` 函数

- 作用：遍历 `/proc` 目录时，控制哪些进程 PID 目录应被列举
- 过滤逻辑：在返回进程信息前检查 `hidden_flag` 和目标进程的 `cloak` 字段，若两者均为 1 则跳过该进程，不将其加入目录列表

### (2) `proc_pid_lookup` 函数

- 作用：通过 PID 直接访问 `/proc/[pid]` 目录时的查找函数
- 过滤逻辑：即使用户知道确切的 PID 并尝试直接访问，若进程被标记为隐藏且全局开关开启，也会返回“无此文件或目录”错误

## 4. 系统调用相关结构

### (1) 系统调用表项

在 `arch/x86/entry/syscalls/syscall_64.tbl` 中注册了两个新的系统调用号：

调用号	架构	调用名称	内核函数
447	common	hide	sys_hide
448	common	hide_user_processes	sys_hide_user_processes

### (2) 系统调用函数签名

```
// 单进程隐藏控制
asm linkage long sys_hide(pid_t pid, int on);

// 批量进程隐藏控制
asm linkage long sys_hide_user_processes(uid_t uid, char *binname,
int recover);
```

## 5. /proc 接口数据结构

### (1) `hide_proc_ops` 结构体

定义在 `fs/proc/hidden_process.c` 中，用于 `/proc/hidden_process` 文件的操作接口：

```
static const struct proc_ops hide_proc_ops = {
    .proc_read = proc_read_hidden,
    .proc_write = proc_write_hidden,
};
```

### (2) `hidden_process_proc_ops` 结构体

定义在 `fs/proc/hidden_process.c` 中，用于 `/proc/hidden_process` 文件的操作接口：

```
static const struct proc_ops hidden_process_proc_ops = {
    .proc_read = proc_read_hidden_process,
};
```

## 6. 数据结构之间的关系

本实验中各数据结构的协作关系如下：

1. 进程创建时： `copy_process()` 初始化新进程的 `task_struct.cloak = 0`
2. 隐藏操作时： 用户调用 `sys_hide()` 或 `sys_hide_user_processes()`， 内核修改目标进程的 `task_struct.cloak = 1`， 并调用 `proc_flush_pid()` 刷新缓存
3. 进程遍历时： `proc_pid_readdir()` 和 `proc_pid_lookup()` 读取 `hidden_flag` 和 `task_struct.cloak`， 决定是否过滤该进程
4. 运行时控制： 通过 `/proc/hide` 修改 `hidden_flag`， 实现全局开关； 通过 `/proc/hidden_process` 查询所有 `cloak == 1` 的进程列表

这种设计通过在进程描述符中添加轻量级标志位，结合全局开关和 `/proc` 过滤机制，实现了高效、安全、可逆的进程隐藏功能，避免了对内核核心数据结构的破坏性修改。

## 编译 Linux 内核

1. 在VirtualBox里安装Ubuntu，保证足够的磁盘空间（我当前分配的空闲空间约 27GB）。
2. 下载Linux源码包并解压：

```
mkdir -p os_lab
cd os_lab
wget
https://mirrors.tuna.tsinghua.edu.cn/kernel/linux/kernel/v5.
x/linux-5.13.10.tar.xz
xz -d linux-5.13.10.tar.xz
tar -xavf linux-5.13.10.tar
```

得到 `linux-5.13.10` 目录。

3. 编辑配置文件， `make localmodconfig` 会自动读取当前正在运行的 Ubuntu 里实际加载的所有模块，避免了全部编译Linux所带来的内存占用大以及花费时间长的弊端。（最开始没有做这一步，而是编译了整个包，发现十个小时都没有成功，而且磁盘空间也被全部占满，故这一步及其重要，一定要砍去不需要的包）此处配置方式为在命令行中输入命令进行配置。

```
cd linux-5.13.10
# 清理之前残留的编译文件
make mrproper

# 包含现在正在运行的模块
make localmodconfig

# 禁用内核模块签名验证
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS
scripts/config --disable MODULE_SIG
```

```

# 可选: 禁用 BTF 调试信息 (我的系统没有安装 pahole)
scripts/config --disable DEBUG_INFO_BTF

# 检查关键配置是否启用
scripts/config --state PROC_FS          # 应该是 y
scripts/config --state SYSCTL          # 应该是 y
scripts/config --state MODULES         # 应该是 y

# 更新配置文件
make olddefconfig

# 修改版本号
nano Makefile
EXTRAVERSION = -OXLAB-1207

```

4. 内核编译阶段，具体所用时间取决于CPU性能和配置。可以加上`-j#`参数进行多线程编译，`#`为线程数，可从`nproc`知晓。编译后的目录约1.6GB。

```

# 编译源码
sudo make -j1

# 安装内核模块
sudo make modules_install

# 安装内核
sudo make install

```

此处我编译的是经过削减的系统，编译时间大概80分钟。

```

HOSTCC arch/x86/boot/tools/build
CC arch/x86/boot/compressed/kaslr.o
CPUSTR arch/x86/boot/cpustr.h
CC arch/x86/boot/cpu.o
CC arch/x86/boot/compressed/ident_map_64.o
CC arch/x86/boot/compressed/idt_64.o
AS arch/x86/boot/compressed/idt_handlers_64.o
AS arch/x86/boot/compressed/mem_encrypt.o
CC arch/x86/boot/compressed/pgtable_64.o
CC arch/x86/boot/compressed/acpi.o
CC arch/x86/boot/compressed/misc.o
GZIP arch/x86/boot/compressed/vmlinuz.bin.gz
MKPIGGY arch/x86/boot/compressed/piggy.S
AS arch/x86/boot/compressed/piggy.o
LD arch/x86/boot/compressed/vmlinuz
ZOFFSET arch/x86/boot/zoffset.h
OBJCOPY arch/x86/boot/vmlinuz.bin
AS arch/x86/boot/header.o
LD arch/x86/boot/setup.elf
OBJCOPY arch/x86/boot/setup.bin
BUILD arch/x86/boot/bzImage
Kernel: arch/x86/boot/bzImage is ready (#1)
victor@victor-VirtualBox:~/os_lab/linux-5.13.10$ sudo make modules_install
[sudo] victor 的密码:
***
```

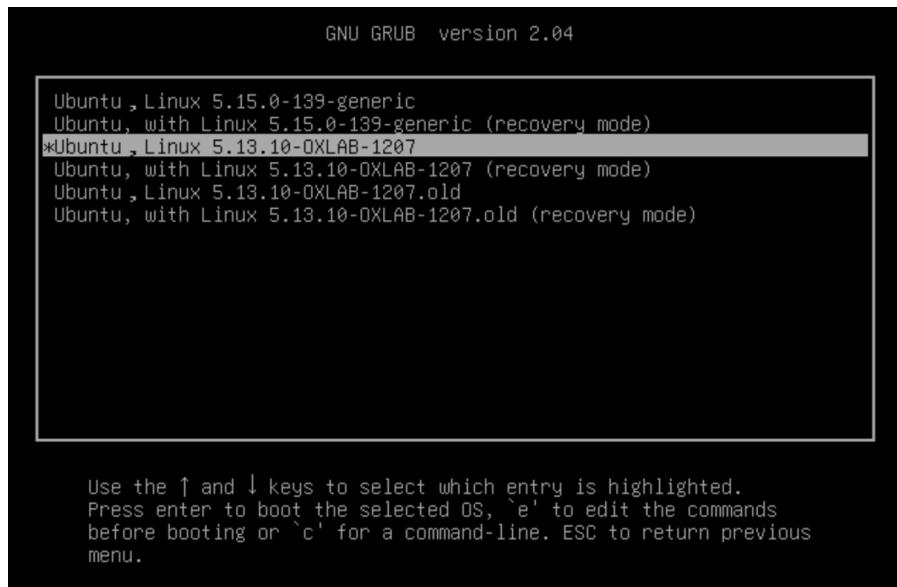
5. 在选择新内核启动之前，需要编辑grub2选项，开启grub菜单。

```
sudo nano /etc/default/grub
GRUB_TIMEOUT_STYLE=menu
GRUB_TIMEOUT=5
sudo update-grub
sudo reboot
```

在编辑完grub后，可以执行`grep -i timeout /boot/grub/grub.cfg`验证是否更新了`grub.cfg`。

```
victor@victor-VirtualBox:~/os_lab/linux-5.13.10$ sudo nano /etc/default/grub
victor@victor-VirtualBox:~/os_lab/linux-5.13.10$ sudo update-grub
Sourcing file '/etc/default/grub'
Sourcing file '/etc/default/grub.d/init-select.cfg'
正在生成 grub 配置文件 ...
找到 Linux 镜像: /boot/vmlinuz-5.15.0-139-generic
找到 initrd 镜像: /boot/initrd.img-5.15.0-139-generic
找到 Linux 镜像: /boot/vmlinuz-5.15.0-125-generic
找到 initrd 镜像: /boot/initrd.img-5.15.0-125-generic
找到 Linux 镜像: /boot/vmlinuz-5.13.10-0XLAB
找到 initrd 镜像: /boot/initrd.img-5.13.10-0XLAB
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
完成
victor@victor-VirtualBox:~/os_lab/linux-5.13.10$ grep -i timeout /boot/grub/grub.cfg
set timeout=30
if [ x$feature_timeout_style = xy ] ; then
    set timeout_style=menu
    set timeout=5
# Fallback normal timeout code in case the timeout_style feature is
    set timeout=5
```

6. 在grub菜单中选择Advanced options for Ubuntu，在二级菜单中选择新编译的内核启动。（！！！由于我在编译配置的时候采用的是不完整的配置，故在重启的时候提示采用`noapic`模式启动。此时需要再次重启，在二级菜单中选中新内核，按`e`进入编辑模式，添加`noapic`，然后按下F10，可以成功启动。启动后运行`uname -a`可以查看当前内核版本号，确认是否是新内核。）



```

Physical RAM using RDRAND RDTSC...
Virtual RAM using RDRAND RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.
[    0.004000] ..MP-BIOS bug: 8254 timer not connected to IO-APIC
[    0.010000] Kernel panic - not syncing: IO-APIC + timer doesn't work! Boot with apic=debug and send a report. Then try booting with the 'noapic' option.
[    0.010000] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 5.13.10-OXLAB-1207 #8
[    0.010000] Hardware name: innotech GmbH VirtualBox/VirtualBox, BIOS VirtualBo
x 12/01/2006
[    0.010000] Call Trace:
[    0.010000]   dump_stack+0x64/0x7c
[    0.010000]   panic+0xf6/0x2b7
[    0.010000]   setup_IO_APIC+0x7e1/0x839
[    0.010000]   ? ioapic_read_entry+0x37/0x40
[    0.010000]   ? clear_IO_APIC_pin+0xc7/0x110
[    0.010000]   apic_intr_mode_init+0xf5/0xfb
[    0.010000]   x86_late_time_init+0x1b/0x2b
[    0.010000]   start_kernel+0x99/0x54e
[    0.010000]   secondary_startup_64_no_verify+0xc2/0xcb
[    0.010000] ---[ end Kernel panic - not syncing: IO-APIC + timer doesn't work!
! Boot with apic=debug and send a report. Then try booting with the 'noapic' option. ]---

```

```

GNU GRUB version 2.04

insmod ext2
set root='hd0,msdos5'
if [ $feature_platform_search_hint = xy ]; then
    search --no-floppy --fs-uuid --set=root --hint-bios=hd
0,msdos5 --hint-efi=hd0,msdos5 --hint-baremetal=ahci0,msdos5 6575deea-e1be-
4d9f-a053-28f1212c1914
else
    search --no-floppy --fs-uuid --set=root 6575deea-e1be-
4d9f-a053-28f1212c1914
fi
echo      '載入 Linux 5.13.10-OXLAB-1207 ...'
linux    /boot/vmlinuz-5.13.10-OXLAB-1207 root=UUID=
6575deea-e1be-4d9f-a053-28f1212c1914 ro quiet splash $vt_handoff noapic
echo      '載入初始化內存盤...'
initrd   /boot/initrd.img-5.13.10-OXLAB-1207

Minimum Emacs-like screen editing is supported. TAB lists
completions. Press Ctrl-x or F10 to boot, Ctrl-c or F2 for a
command-line or ESC to discard edits and return to the GRUB
menu.

```

```

victor@victor-VirtualBox:~$ uname -a
Linux victor-VirtualBox 5.13.10-OXLAB-1207 #1 SMP Sun Dec 7 10:50:15 CST 2025 x86_64 x8
6_64 x86_64 GNU/Linux
victor@victor-VirtualBox:~$ 

```

## 新增 hide 系统调用

### 1. 修改 task\_struct

在 `/include/linux/sched.h` 中，为 `task_struct` 新增成员变量。规定：0表示显示，1表示隐藏。

`kernel/fork.c` 的 `copy_process(pid *, int, int, kernel_clone_args *)` 中，为 fork 的子进程设置 `clock` 为0。

```

os_lab > linux-5.13.10 > include > linux > C sched.h
1379 #ifdef CONFIG_X86_MCE
1380     __u64 mce_ripv : 1,
1381     struct callback_head mce_kill_me;
1382 #endif
1383
1384 #ifdef CONFIG_KRETPROBES
1385     struct llist_head kretprobe_instances;
1386 #endif
1387
1388 /* OSLAB: hide process flag */
1389     int cloak;
1390
1391
1392
1393
1394
1395

```

```

os_lab > linux-5.13.10 > kernel > C fork.c
2207 #endif
2325
2326     proc_fork_connector(p);
2327     sched_post_fork(p);
2328     cgroup_post_fork(p, args);
2329     perf_event_fork(p);
2330
2331     trace_task_newtask(p, clone_flags);
2332     uprobe_copy_process(p, clone_flags);
2333
2334     copy_oom_score_adj(clone_flags, p);
2335
2336     /* OSLAB: initialize hide process flag */
2337     p->cloak = 0;
2338
2339
2340     return p;

```

### 2. 修改 procfs

在 `fs\proc\base.c` 的 `int proc_pid_readdir(struct file *file, struct dir_context *ctx)` 以及 `struct dentry *proc_pid_lookup(struct dentry *dentry, unsigned int flags)` 中，增加针对 `cloak` 的判断语句。

```

os_lab > linux-5.13.10 > fs > proc > C base.c
3362 struct dentry *proc_pid_lookup(struct dentry *dentry, unsigned int flags){
3384     if (fs_info->hide_pid == HIDEPID_NOT_PTRACEABLE) {
3385         if (!has_pid_permissions(fs_info, task, HIDEPID_NO_ACCESS))
3386             goto out_put_task;
3387     }
3388
3389     /* OSLAB: hide process if cloak is set and hidden_flag is 1 */
3390     extern int hidden_flag;
3391     if (task->cloak == 1 && hidden_flag == 1)
3392         goto out_put_task;
3393
3394     result = proc_pid_instantiate(dentry, task, NULL);
3395 out_put_task:
3396     put_task_struct(task);
3397 out:
3398     return result;
3399 }
3400
3436 int proc_pid_readdir(struct file *file, struct dir_context *ctx){
3459     for (iter = next_tgid(ns, iter);
3466         if (!has_pid_permissions(fs_info, iter.task, HIDEPID_INVISIBLE))
3467             continue;
3468
3469     /* OSLAB: hide process if cloak is set and hidden_flag is 1 */
3470     extern int hidden_flag;
3471     if (iter.task->cloak == 1 && hidden_flag == 1)
3472         continue;
3473
3474     len = snprintf(name, sizeof(name), "%u", iter.tgid);
3475     ctx->pos = iter.tgid + TGID_OFFSET;
3476     if (!proc_fill_cache(file, ctx, name, len,
3477             proc_pid_instantiate, iter.task, NULL)) {
3478         put_task_struct(iter.task);
3479         return 0;
3480     }
3481
3482     ctx->pos = PID_MAX_LIMIT + TGID_OFFSET;
3483     return 0;
3484 }

```

### 3.添加系统调用及相关接口

#### (1) 新建全局变量声明文件

在`include/linux`目录下新建`var_defs.h`头文件，声明全局隐藏开关变量`hidden_flag`，便于其他内核模块引用。

```

os_lab > linux-5.13.10 > include > linux > C var_defs.h
1     extern int hidden_flag;
2

```

#### (2) 实现`sys_hide`系统调用

在`kernel`目录下新建`hide.c`文件，实现系统调用`sys_hide(pid_t pid, int on)`。该函数通过`pid_task(find_vpid(pid), PIDTYPE_PID)`查找指定进程的`task_struct`结构体（较早版本内核使用`find_task_by_pid(pid)`），并根据参数`on`的值设置进程的`cloak`字段。调用`proc_flush_pid()`函数清空VFS层缓存，确保`/proc`文件系统立即生效。权限检查通过`current_uid().val`（较早版本内核为`current->uid`）实现，仅允许root用户（`uid=0`）执行隐藏操作。同时使用`printk`输出内核日志便于调试。

```
os_lab > linux-5.13.10 > kernel > C hide.c
 1  #include <linux/syscalls.h>
 2  #include <linux/kernel.h>
 3  #include <linux/linkage.h>
 4  #include <linux/types.h>
 5  #include <linux/sched.h>
 6  #include <linux/pid.h>
 7  #include <linux/proc_fs.h>
 8  #include <linux/cred.h>
 9  #include <linux/var_def.h>
10
11 SYSCALL_DEFINE2(hide, pid_t, pid, int, on)
12 {
13     struct task_struct *p;
14     struct pid *thread_pid;
15
16     printk("Syscall hide called.\n");
17     p = NULL;
18     if (pid > 0 && current_uid().val == 0) {
19         p = pid_task(find_vpid(pid), PIDTYPE_PID);
20         if (!p)
21             return 1;
22         if (hidden_flag == 1) {
23             p->cloak = on;
24             if (on == 1)
25                 printk("Process %d is hidden by root.\n", pid);
26             if (on == 0)
27                 printk("Process %d is displayed by root.\n", pid);
28             thread_pid = get_pid(p->thread_pid);
29             proc_flush_pid(thread_pid);
30         }
31     } else {
32         printk("Permission denied. You must be root to hide a process.\n");
33     }
34     return 0;
35 }
```

### (3) 实现 sys\_hide\_user\_processes 系统调用

在kernel目录下新建hide\_user\_processes.c文件，实现系统调用

`sys_hide_user_processes(uid_t uid, char *binname, int recover)`。该函数支持批量隐藏或恢复特定用户ID或进程名的所有进程，通过遍历全局进程链表 `for_each_process()` 实现。当 `recover` 为 0 时隐藏匹配进程，为 1 时恢复显示。

```
os_lab > linux-5.13.10 > kernel > C hide_user_processes.c
 1 #include <linux/syscalls.h>
 2 #include <linux/kernel.h>
 3 #include <linux/linkage.h>
 4 #include <linux/types.h>
 5 #include <linux/sched.h>
 6 #include <linux/pid.h>
 7 #include <linux/proc_fs.h>
 8 #include <linux/cred.h>
 9 #include <linux/string.h>
10 #include <linux/var_def.h>
11
12 SYSCALL_DEFINE3(hide_user_processes, uid_t, uid, char *, binname, int, recover)
13 {
14     struct task_struct *p = NULL;
15
16     if (current_uid().val != 0) {
17         printk("Permission denied. Only root can call hide_user_processes.\n");
18         return 1;
19     }
20
21     if (recover == 0) {
22         if (binname == NULL) {
23             for_each_process (p) {
24                 if (p->cred->uid_val == uid && hidden_flag == 1) {
25                     p->cloak = 1;
26                     proc_flush_pid(get_pid(p->thread_pid));
27                 }
28             }
29             printk("All processes of uid %d are hidden.\n", uid);
30         } else {
31             char kbinname[TASK_COMM_LEN];
32             long len = strncpy_from_user(kbinname, binname, TASK_COMM_LEN);
33             kbinname[TASK_COMM_LEN - 1] = '\0';
34             if (unlikely(len < 0)) {
35                 printk("Unable to do strncpy_from_user");
36                 return 2;
37             }
38         }
39     }
40 }
```

```
os_lab > linux-5.13.10 > kernel > C hide_user_processes.c
13 {
14     if (recover == 0) {
15         if (binname == NULL) {
16             printk("All processes of uid %d are hidden.\n", uid);
17         } else {
18             char kbinname[TASK_COMM_LEN];
19             long len = strncpy_from_user(kbinname, binname, TASK_COMM_LEN);
20             kbinname[TASK_COMM_LEN - 1] = '\0';
21             if (unlikely(len < 0)) {
22                 printk("Unable to do strncpy_from_user");
23                 return 2;
24             }
25             for_each_process (p) {
26                 char s[TASK_COMM_LEN];
27                 get_task_comm(s, p);
28                 if (p->cred->uid.val == uid &&
29                     strcmp(s, kbinname, TASK_COMM_LEN) == 0 &&
30                     hidden_flag == 1) {
31                     p->cloak = 1;
32                     printk("Process %s of uid %d is hidden.\n", kbinname, uid);
33                     proc_flush_pid(get_pid(p->thread_pid));
34                 }
35             }
36         }
37     } else {
38         for_each_process (p) {
39             p->cloak = 0;
40         }
41     }
42     return 0;
43 }
```

#### (4) 创建 /proc/hide 控制接口

在 `fs/proc` 目录下新建 `hide.c` 文件，提供全局隐藏功能的运行时开关。定义全局变量 `hidden_flag` 并通过 `EXPORT_SYMBOL` 导出，初始值设为 1（启用隐藏）。实现 `proc_read_hidden` 和 `proc_write_hidden` 函数，支持读取和修改该开关状态。写操作中对输入字符串进行严格校验，自动去除末尾的换行符、空格等字符，避免解析错误。通过 `proc_create("hide", 0644, NULL, &hide_proc_ops)` 在 `/proc` 目录下创建文件节点，用户可通过 `cat /proc/hidden` 读取当前状态，使用 `echo "0" | sudo tee /proc/hidden` 动态禁用隐藏功能。

```
lab>linux-5.13.10>fs>proc>C hide.c
1 #include <linux/kernel.h>
2 #include <linux/init.h>
3 #include <linux/proc_fs.h>
4 #include <linux/uaccess.h>
5 #include <linux/var_def.h>
6 #include <linux/module.h>
7
8 #define PROC_MAX_SIZE 16
9
10 int hidden_flag = 1;
11 EXPORT_SYMBOL(hidden_flag);
12 static struct proc_dir_entry *hide_entry;
13
14 static ssize_t proc_read_hidden(struct file *file, char __user *buf, size_t count, loff_t *ppos)
15 {
16     char str[16];
17     ssize_t cnt, ret;
18     snprintf(str, sizeof(str), "%d\n", hidden_flag);
19     cnt = strlen(str);
20     ret = copy_to_user(buf, str, cnt);
21     *ppos += cnt - ret;
22     if (*ppos > cnt) return 0;
23     return cnt;
24 }
25
26 static ssize_t proc_write_hidden(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
27 {
28     char temp[PROC_MAX_SIZE];
29     int tmp_flag = 0;
30     size_t len;
31
32     if (count > PROC_MAX_SIZE - 1)
33         count = PROC_MAX_SIZE - 1;
34
35     if (copy_from_user(temp, buf, count))
36         return -EFAULT;
37
38     temp[count] = '\0';
39 }
```

```
os_lab>linux-5.13.10>fs>proc>C hide.c
27 {
28     // 去除末尾的空格和换行符
29     len = count;
30     while (len > 0 && (temp[len-1] == '\n' || temp[len-1] == ' ' ||
31             temp[len-1] == '\r' || temp[len-1] == '\t')) {
32         temp[--len] = '\0';
33     }
34
35     if (len == 0)
36         return -EINVAL;
37
38     if (kstrtoint(temp, 10, &tmp_flag))
39         return -EINVAL;
40
41     hidden_flag = tmp_flag;
42     printk(KERN_INFO "[HIDE] hidden_flag changed to %d\n", hidden_flag);
43
44     return count;
45 }
46
47 static const struct proc_ops hide_proc_ops = {
48     .proc_write = proc_write_hidden,
49     .proc_read = proc_read_hidden,
50     .proc_lseek = default_llseek,
51 };
52
53 static int __init proc_hide_init(void)
54 {
55     hide_entry = proc_create("hide", 0644, NULL, &hide_proc_ops);
56     return 0;
57 }
58
59 static void __exit proc_hide_cleanup(void)
60 {
61     proc_remove(hide_entry);
62 }
63
64 fs_initcall(proc_hide_init);
```

## (5) 创建 `/proc/hidden_process` 查询接口

在 `fs/proc` 目录下新建 `hidden_process.c` 文件，提供被隐藏进程列表的查询功能。通过 `for_each_process()` 遍历所有进程，筛选出 `cloak` 字段为 1 的进程，将其 PID 拼接成空格分隔的字符串返回给用户态。用户可通过 `cat /proc/hidden_process` 查看当前所有被隐藏的进程 ID，便于监控和调试。

```

os_lab > linux-5.13.10 > fs > proc > C hidden_process.c
 7   static struct proc_dir_entry *hidden_process_entry;
 8
 9   static ssize_t proc_read_hidden_process(struct file *file, char __user *buf, size_t count, loff_t *ppos)
10 {
11     ssize_t cnt, ret;
12     char kbuf[1000];
13     char tmp[16];
14     struct task_struct *p;
15
16     kbuf[0] = '\0';
17     for_each_process (p) {
18         if (p->cloak == 1) {
19             snprintf(tmp, sizeof(tmp), "%ld ", (long)p->pid);
20             strlcat(kbuf, tmp, sizeof(kbuf));
21         }
22     }
23     cnt = strlen(kbuf);
24     ret = copy_to_user(buf, kbuf, cnt);
25     *ppos += cnt - ret;
26     if (*ppos > cnt) return 0;
27     return cnt;
28 }
29
30 static const struct proc_ops hidden_process_proc_ops = {
31     .proc_read = proc_read_hidden_process,
32 };
33
34 static int __init proc_hidden_process_init(void) {
35     hidden_process_entry = proc_create("hidden_process", 0444, NULL, &hidden_process_proc_ops);
36     return 0;
37 }
38
39 static void __exit proc_hidden_process_cleanup(void)
40 {
41     proc_remove(hidden_process_entry);
42 }
43
44 fs_initcall(proc_hidden_process_init);

```

## (6) 修改编译配置文件

在 `kernel/Makefile` 中的 `obj-y` 变量后追加 `hide.o` 和 `hide_user_processes.o`，确保两个系统调用实现文件被静态编译进内核镜像。在 `fs/proc/Makefile` 中的 `proc-y` 变量后追加 `hide.o` 和 `hidden_process.o`，将两个 `/proc` 接口文件一并编译。

```

os_lab > linux-5.13.10 > kernel > M Makefile
 1 # SPDX-License-Identifier: GPL-2.0
 2 #
 3 # Makefile for the linux kernel.
 4 #
 5
 6 obj-y      = fork.o exec_domain.o panic.o \
 7           cpu.o exit.o softirq.o resource.o \
 8           sysctl.o capability.o ptrace.o user.o \
 9           signal.o sys.o umh.o workqueue.o pid.o task_work.o \
10           extable.o params.o \
11           kthread.o sys_ni.o nsproxy.o \
12           notifier.o ksysfs.o cred.o reboot.o \
13           async.o range.o smpboot.o ucount.o regset.o \
14           hide.o hide_user_processes.o
15
16 obj-$(CONFIG_USERMODE_DRIVER) += usermode_driver.o
17 obj-$(CONFIG_MODULES) += kmod.o
18 obj-$(CONFIG_MULTIUSER) += groups.o
19
20 ifdef CONFIG_FUNCTION_TRACER
21 # Do not trace internal ftrace files
22 CFLAGS_REMOVE_irq_work.o = $(CC_FLAGS_FTRACE)
23 endif

```

```

os_lab > linux-5.13.10 > fs > proc > M Makefile
 17 proc-y    += cpufreq.o
 18 proc-y    += devices.o
 19 proc-y    += interrupts.o
 20 proc-y    += loadavg.o
 21 proc-y    += meminfo.o
 22 proc-y    += stat.o
 23 proc-y    += uptime.o
 24 proc-y    += util.o
 25 proc-y    += version.o
 26 proc-y    += softirqs.o
 27 proc-y    += namespaces.o
 28 proc-y    += self.o
 29 proc-y    += thread_self.o
 30 proc-$(CONFIG_PROC_SYSCTL)  += proc_sysctl.o
 31 proc-$(CONFIG_NET)        += proc_net.o
 32 proc-$(CONFIG_PROC_KCORE)  += kcore.o
 33 proc-$(CONFIG_PROC_VMCORE) += vmcore.o
 34 proc-$(CONFIG_PRINTK)     += kmsg.o
 35 proc-$(CONFIG_PROC_PAGE_MONITOR)  += page.o
 36 proc-$(CONFIG_BOOT_CONFIG)  += bootconfig.o
 37 proc-y    += hide.o hidden_process.o
 38

```

## (7) 注册系统调用

在 `include/linux/syscalls.h` 文件末尾添加两个系统调用的函数原型声明： `asm linkage`

```
long sys_hide(pid_t pid, int on) 和 asm linkage long  
sys_hide_user_processes(uid_t uid, char *binname, int recover)。
```

```
os_lab > linux-5.13.10 > include > linux > C syscalls.h  
1364 long ksys_old_msctl(int msqid, int cmd, struct msqid_ds __user *buf);  
1365 long ksys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,  
1366 | long msgtyp, int msgflg);  
1367 long ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz,  
1368 | int msgflg);  
1369 long ksys_shmget(key_t key, size_t size, int shmflg);  
1370 long ksys_shmdt(char __user *shmaddr);  
1371 long ksys_old_shmctl(int shmid, int cmd, struct shmid_ds __user *buf);  
1372 long compat_ksys_semtimedop(int semid, struct sembuf __user *tsems,  
1373 | unsigned int nsops,  
1374 | const struct old_timespec32 __user *timeout);  
1375  
1376 int __sys_getsockopt(int fd, int level, int optname, char __user *optval,  
1377 | int __user *optlen);  
1378 int __sys_setsockopt(int fd, int level, int optname, char __user *optval,  
1379 | int optlen);  
1380  
1381 /* OSLAB: hide process syscalls */  
1382 asm linkage long sys_hide(pid_t pid, int on);  
1383 asm linkage long sys_hide_user_processes(uid_t uid, char *binname, int recover);  
1384  
1385 #endif  
1386
```

在 `arch/x86/entry/syscalls/syscall_64.tbl` 系统调用表中添加两行记录，分别是 447

```
common hide sys_hide 以及 448 common hide_user_processes  
sys_hide_user_processes。
```

```
os_lab > linux-5.13.10 > arch > x86 > entry > syscalls > E syscall_64.tbl  
364 440 common process_madvise sys_process_madvise  
365 441 common epoll_pwait2 sys_epoll_pwait2  
366 442 common mount_setattr sys_mount_setattr  
367 # 443 reserved for quotactl_path  
368 444 common landlock_create_ruleset sys_landlock_create_ruleset  
369 445 common landlock_add_rule sys_landlock_add_rule  
370 446 common landlock_restrict_self sys_landlock_restrict_self  
371 447 common hide sys_hide  
372 448 common hide_user_processes sys_hide_user_processes  
373
```

注意插入位置应在 x32 系统调用段之前，避免与特殊架构调用号冲突。

## 编写测试程序及测试

重新编译安装。增量编译速度会明显快于第一次编译。推荐使用 `-s` 选项（`sudo make -s -j8`），不会打印正常日志，能更容易地发现编译错误。可以重启后观察 `uname -a` 是否显示为最近一次编译的结果，确定新内核是否已经成功安装。

编辑 `hide_test.c` 和 `hide_user_processes_test.c`，输入如下内容（要将宏定义的 `SYSCALL_NUM` 修改为之前自己定义的系统调用号）：

```
// hide_test.c  
#include <stdio.h>  
#include <sys/syscall.h>
```

```

#include <unistd.h>

#define SYSCALL_NUM 447

int main()
{
    int syscallNum = SYSCALL_NUM;
    pid_t pid = 1;           // 以 PID 1 为测试对象
    int on = 1;              // 1 表示隐藏, 0 表示恢复
    syscall(syscallNum, pid, on);
    return 0;
}

// hide_user_processes_test.c
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

#define SYSCALL_NUM 448

int main()
{
    int syscallNum = SYSCALL_NUM;
    uid_t uid = 0;           // 批量隐藏 uid=0 的所有进程
    char *binname = NULL;    // 传 NULL, 表示按 uid 批量处理
    int recover = 0;          // 0 表示隐藏, 1 表示恢复

    syscall(syscallNum, uid, binname, recover);
    return 0;
}

```

首先使用 GNU 编译器生成两个测试程序的可执行文件，并赋予执行权限。具体命令如下：

```

gcc hide_test.c -o hide_test
gcc hide_user_processes_test.c -o hide_user_processes_test
chmod +x hide_test
chmod +x hide_user_processes_test

```

在完成编译之后，分别对两个系统调用进行测试，并通过 `ps`、`/proc` 以及 `dmesg` 的输出来验证隐藏效果。

### (1) `hide_test` 单进程隐藏测试

这一部分用于验证基础的 `sys_hide(pid_t pid, int on)` 系统调用是否可以正确隐藏单个进程。测试时，我以 PID 为 1 的 `init` 进程（系统启动后的第一个进程）作为目标，观察其在进程列表中的可见性变化。具体步骤如下：

1. 首先使用 `ps aux | head -5` 查看系统中前若干个进程，确认此时 PID 1 的 `init` 进程是可见的。此时命令输出中可以清晰看到形如 `/sbin/init splash`（或 `systemd`）的记录，PID 列为 1，USER 为 root。

2. 通过 `/proc/hide` 打开全局隐藏开关。可以执行：

```
echo "1" | sudo tee /proc/hide
cat /proc/hide
```

终端输出为 `1`，说明内核中的 `hidden_flag` 已经被设置为 1，全局隐藏功能处于启用状态。这一步同时验证了前面实现的 `/proc/hide` 写入和读取接口功能正常。

3. 在开关打开的前提下，运行 `hide_test` 调用 `sys_hide` 系统调用隐藏 PID 1 对应的进程。程序内部会调用 `syscall(447, 1, 1)`，在内核中查找到 PID 1 的 `task_struct`，将其 `cloak` 字段设置为 1，并调用 `proc_flush_pid()` 刷新 `/proc/1` 对应的缓存，使隐藏效果立即生效。
4. 再次执行 `ps aux | head -5`。此时命令输出中已经看不到原先 PID 为 1 的 `init` 进程记录，前几行只剩下若干内核线程（如 `kthreadd`、`rcu_gp` 等），说明在 `hidden_flag == 1` 且 `cloak == 1` 的条件下，`/proc` 进程遍历函数已经正确跳过了 PID 1。
5. 为了进一步确认系统调用确实被触发，可以执行 `dmesg | tail -n 10` 查看内核日志。这些日志来自 `kernel/include/linux/hide.h` 中的 `printk` 调用，侧面证明系统调用路径被正确执行，目标进程已在内核内部被标记为隐藏。
6. 执行 `echo "0" | sudo tee /proc/hide` 关闭开关，再次执行 `ps aux | head -5` 可以看到，原先被隐藏的进程重新显示出来。

```
r@victor-VirtualBox:~$ ps aux | head -5
PID %CPU %MEM     VSZ   RSS TTY      STAT START    TIME COMMAND
 1  0.0  0.2 168132 11516 ?        Ss  11:57  0:07 /sbin/init splash
 2  0.0  0.0      0   0 ?        S     11:57  0:00 [kthreadd]
 3  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_gp]
 4  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_par_gp]
r@victor-VirtualBox:~$ echo "1" | sudo tee /proc/hide
] victor 的密码:
r@victor-VirtualBox:~$ cat /proc/hide
r@victor-VirtualBox:~$ sudo ./hide_test
r@victor-VirtualBox:~$ ps aux | head -5
PID %CPU %MEM     VSZ   RSS TTY      STAT START    TIME COMMAND
 2  0.0  0.0      0   0 ?        S     11:57  0:00 [kthreadd]
 3  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_gp]
 4  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_par_gp]
 6  0.0  0.0      0   0 ?        I<  11:57  0:00 [kworker/0:0H-events
pri]

victor@victor-VirtualBox:~$ dmesg | tail -n 10
[ 6914.626558] loop0: detected capacity change from 0 to 8
[ 7214.625160] loop0: detected capacity change from 0 to 8
[ 7514.628139] loop0: detected capacity change from 0 to 8
[ 7814.693364] loop0: detected capacity change from 0 to 8
[ 8114.637536] loop0: detected capacity change from 0 to 8
[ 8414.624437] loop0: detected capacity change from 0 to 8
[ 8714.632367] loop0: detected capacity change from 0 to 8
[ 8843.623217] [HIDE] hidden_flag changed to 1
[ 8872.275177] Syscall hide called.
[ 8872.275177] Process 1 is hidden by root.
r@victor-VirtualBox:~$ echo "0" | sudo tee /proc/hide
0
victor@victor-VirtualBox:~$ ps aux | head -5
USER      PID %CPU %MEM     VSZ   RSS TTY      STAT START    TIME COMMAND
root      1  0.0  0.2 168132 11516 ?        Ss  11:57  0:07 /sbin/init splash
root      2  0.0  0.0      0   0 ?        S     11:57  0:00 [kthreadd]
root      3  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_gp]
root      4  0.0  0.0      0   0 ?        I<  11:57  0:00 [rcu_par_gp]
victor@victor-VirtualBox:~$
```

上述步骤可以验证：在全局开关开启的情况下，`hide_test` 能够成功隐藏指定 PID 的单个进程，`ps` 输出中不再显示该进程，全局开关关闭以后，可以重新看到被隐藏的进程。说明 `sys_hide` 系统调用及其配套的 `/proc` 过滤逻辑工作正常。

## (2) `hide_user_processes_test` 批量隐藏 root 进程测试

第二个测试程序 `hide_user_processes_test` 用于验证扩展系统调用 `sys_hide_user_processes(uid_t uid, char *binname, int recover)` 的批量隐藏能力。这里我选择以 root 用户（uid 为 0）的所有进程为目标，重点观察“隐藏前后 root 进程数量的变化”，同时配合 `/proc/hidden_process` 的内容进行验证。测试流程如下：

- 与前一个测试类似，首先保证全局开关处于打开状态。执行：

```
echo "1" | sudo tee /proc/hide  
cat /proc/hide
```

输出为 1，说明 `hidden_flag` 已经被置为启用。此时 `/proc` 层的过滤逻辑会根据 `cloak` 字段决定是否对某个进程进行隐藏。

- 在隐藏前，先执行 `ps aux | head -10`，查看当前系统中 root 用户的进程大致情况。从输出中可以看到大量 USER 列为 root 的系统服务和内核线程（包括 PID 较小的各种后台进程）。这一部分输出作为“批量隐藏前”的参考状态，用于后续对比。
- 运行 `hide_user_processes_test`，通过系统调用一次性将 uid 为 0 的所有进程标记为隐藏。由于该操作需要 root 权限，因此使用 `sudo` 执行 `sudo ./hide_user_processes_test`。该程序内部调用 `syscall(448, 0, NULL, 0)`，内核中的 `sys_hide_user_processes` 会进入“按 uid 批量隐藏”的分支，对所有 `p->cred->uid.val == 0` 的进程执行如下操作：
  - 将其 `cloak` 字段设置为 1；
  - 调用 `proc_flush_pid(get_pid(p->thread_pid))` 刷新对应的 `/proc/pid` 目录项缓存；
  - 在遍历完成后输出一条内核日志 `All processes of uid 0 are hidden.`
- 执行完测试程序后，执行 `ps aux | head -10` 再次查看进程列表。与隐藏前对比，可以观察到 root 用户的进程数明显减少（部分原本属于 root 的 PID 不再出现），而普通用户（如 `victor`）自己的桌面进程和会话进程仍然保持可见。这说明批量隐藏操作主要作用在 uid 为 0 的进程上，对其他用户的可见性没有影响。
- 为了验证所有 root 进程确实被标记为隐藏，执行 `cat /proc/hidden_process` 可以查看 `/proc/hidden_process` 文件的内容，输出是一长串用空格分隔的 PID。这些 PID 对应的进程在内核中已经被设置为 `cloak == 1`，并通过 `/proc` 过滤逻辑从进程列表中被移除。可以将这一输出的截图作为“当前被隐藏 root 进程集合”的直观展示，与前面 `ps` 的结果互相印证。
- 同样可以通过 `dmesg` 查看批量隐藏操作的内核日志。执行 `dmesg | tail -n 10` 可以看到类似如下的输出：
  - [HIDE] `hidden_flag changed to 1`

- All processes of uid 0 are hidden.

说明 `sys_hide_user_processes` 确实被调用，并完成了对 uid 为 0 的进程的批量处理。

```
@victor-VirtualBox:~$ ps aux | head -10
  PID %CPU %MEM   VSZ RSS TTY      STAT START  TIME COMMAND
    1  0.0  0.2 168132 11516 ?        Ss  11:57  0:07 /sbin/init splash
    2  0.0  0.0     0  0 ?        S     11:57  0:00 [kthreadd]
    3  0.0  0.0     0  0 ?        I<  11:57  0:00 [rcu_gp]
    4  0.0  0.0     0  0 ?        I<  11:57  0:00 [rcu_par_gp]
    6  0.0  0.0     0  0 ?        I<  11:57  0:00 [kworker/0:0H-events]
...
@victor-VirtualBox:~$ sudo ./hide_user_processes_test
@victor-VirtualBox:~$ ps aux | head -10
  PID %CPU %MEM   VSZ RSS TTY      STAT START  TIME COMMAND
d+ 332  0.0  0.3 25492 14220 ?        Ss  11:57  0:02 /lib/systemd/systemd
...
  344  0.0  0.0  8532  3672 ?        Ss  11:57  0:01 avahi-daemon: runnin
tor-VirtualBox.local]
e+ 347  0.0  0.1  9920  6568 ?        Ss  11:57  0:07 /usr/bin/dbus-daemon
tem --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
  358  0.0  0.1 224352  4368 ?        Ssl 11:57  0:00 /usr/sbin/rsyslogd -
NE
  387  0.0  0.0  8348  328 ?        S     11:57  0:00 avahi-daemon: chroot
r
  455  0.0  0.1 15212  6742 ?        S     11:57  0:00 /usr/lib/cups/print
...
@victor-VirtualBox:~$ cat /proc/hidden_process
1 2 3 4 6 8 9 10 11 12 13 14 15 16 17 18 19 39 40 41 44 45 46 47 48 49 50 52 54 5
7 58 59 60 63 64 123 124 163 188 254 340 341 345 346 348 355 356 360 361 362 363 3
9 438 610 744 750 778 829 888 1144 2332 2437 2457 2495 2514 victor@victor-Virtua
dmesg | tail -n 10
[ 8843.623217] [HIDE] hidden_flag changed to 1
[ 8872.275177] Syscall hide called.
[ 8872.275177] Process 1 is hidden by root.
[ 8916.917242] [HIDE] hidden_flag changed to 0
[ 9014.625398] loop0: detected capacity change from 0 to 8
[ 9083.761147] [HIDE] hidden_flag changed to 1
[ 9160.818181] All processes of uid 0 are hidden.
[ 9280.343166] [HIDE] hidden_flag changed to 1
[ 9312.337184] All processes of uid 0 are hidden.
[ 9314.642142] loop0: detected capacity change from 0 to 8
victor@victor-VirtualBox:~$
```

综合上述两个测试程序的结果可以看到：`sys_hide` 能够精确控制单个进程的隐藏，而 `sys_hide_user_processes` 则提供了按用户 ID 维度的批量隐藏能力。再结合 `/proc/hide` 全局开关和 `/proc/hidden_process` 查询接口，可以实现对进程可见性的灵活控制。整个系统的设计具有良好的模块化特性，通过标志位而非链表操作的方式保证了内核数据结构的完整性，同时提供了丰富的运行时控制和监控手段，使得进程隐藏功能既安全可靠又便于调试和管理。测试结果表明，所有系统调用均能正常工作，进程隐藏和恢复功能符合预期，验证了设计方案的可行性和有效性。

## 实验体会

通过本次实验，我对 Linux 内核的进程管理机制有了更加深入和系统的理解。从最初对内核源码的陌生，到逐步掌握 `task_struct` 结构体的组织方式、进程链表的遍历机制以及 `/proc` 文件系统的工作原理，整个过程充满了挑战，但也收获颇丰。

在实验初期，内核编译环节就给我留下了深刻的印象。最开始没有使用 `make localmodconfig` 进行配置裁剪，而是尝试编译完整的内核，结果不仅耗时超过十个小时仍未完成，还导致虚拟机磁盘空间被完全占满。这次经历让我深刻认识到，在资源受限的环境下进行内核开发时，合理的配置管理和模块裁剪至关重要。后来通过精简配置，编译时间缩短到 80 分钟左右，磁盘占用也降低到可控范围，这种效率的提升让我意识到，了解工具链的工作机制和掌握优化技巧对于实际开发同样重要。

在系统调用的设计与实现过程中，我最大的体会是“安全性与可维护性优先”的设计原则。传统的进程隐藏方法通常直接操作 `tasks` 链表，将目标进程节点从链表中移除，这种方式虽然实现简单，但存在链表完整性破坏、恢复困难等严重问题，甚至可能导致内核崩溃。经过深入思考和资料查阅，我最终选择了通过 `cloak` 标志位结合 `/proc` 过滤逻辑的方案。这种设计不仅避免了对内核核心数据结构的破坏性修改，还提供了良好的可逆性和可观测性。在调试过程中，`/proc/hide` 和 `/proc/hidden_process` 两个接口发挥了巨大作用，前者让我可以随时开关隐藏功能，后者则帮助我快速确认当前哪些进程被标记为隐藏状态，大大降低了调试难度。

在代码实现细节上，我也遇到了不少技术难点。例如，在实现 `sys_hide_user_processes` 时，如何安全地从用户空间复制进程名字符串、如何正确遍历全局进程链表并避免竞态条件、如何通过 `proc_flush_pid()` 刷新 VFS 缓存使隐藏立即生效，这些问题都需要仔细阅读内核源码和相关文档才能找到正确的解决方案。特别是在处理不同内核版本的 API 差异时（如 `current_uid().val` 与 `current->uid` 的区别、`find_vpid()` 与 `find_task_by_pid()` 的差异），我深刻体会到内核开发对细节的严格要求和对版本兼容性的重视。

测试阶段的工作同样让我受益匪浅。通过 `ps`、`dmesg` 和自定义测试程序的组合验证，我不仅确认了系统调用的功能正确性，还学会了如何利用内核日志进行调试、如何通过 `/proc` 接口观察内核状态变化。特别是在批量隐藏 root 进程的测试中，当看到 `ps` 输出中大量 root 进程消失，而 `/proc/hidden_process` 中显示出对应的 PID 列表时，那种“理论设计与实际效果完美对应”的成就感是难以言表的。

回顾整个实验过程，我认为最大的收获有三点：第一是对 Linux 内核模块化设计思想的深刻理解，内核通过清晰的接口划分和数据结构设计，使得功能扩展变得相对可控；第二是对系统调用机制的全面掌握，从系统调用表的注册到参数传递、权限检查、内核态与用户态的数据交互，每个环节都需要严谨的设计；第三是内核开发的实践能力得到了实质性提升，包括编译配置、代码调试、问题排查等方面的技能都有了长足进步。

此外，这次实验也让我认识到内核开发与应用层开发的巨大差异。内核代码中不能使用标准库函数，需要使用 `printk` 代替 `printf`，使用 `kmalloc` 代替 `malloc`；任何一个指针错误或内存访问越界都可能导致系统崩溃；权限控制和安全检查必须贯穿始终。这些约束虽然增加了开发难度，但也培养了我更加严谨的编程习惯和更强的代码质量意识。

总的来说，本次实验不仅让我掌握了进程管理和系统调用的核心知识，更重要的是培养了我分析问题、解决问题的能力，以及面对复杂系统时保持耐心和细心的品质。这些经验和能力的积累，对我未来在操作系统领域的深入学习和研究具有重要的指导意义。

## 参考文献

### Linux 内核概述与进程管理

- Linux Kernel Labs - Process Management: <https://linux-kernel-labs.github.io/resources/heads/master/lectures/processes.html>
- The Linux Documentation Project - Process Management: <https://tldp.org/LDP/lk/kernel/processes.html>
- Linux Kernel Threads and Processes (task\_struct): <https://cylab.be/blog/347/linux-kernel-threads-and-processes-management-task-struct>
- Boston University - Linux Process Management: [https://www.cs.bu.edu/fac/richwest/cs591w1/notes/linuxprocess\\_mgt.PDF](https://www.cs.bu.edu/fac/richwest/cs591w1/notes/linuxprocess_mgt.PDF)
- Linux 内核在线源码查看器: <https://elixir.bootlin.com/linux/latest/source>

- Linux 内核官方文档: <https://www.kernel.org/doc/html/latest/>

## 内核编译与配置

- 如何快速构建精简的 Linux 内核: <https://www.kernel.org/doc/html/v6.6/admin-guide/quickly-build-trimmed-linux.html>
- Linux 内核编译详细教程: <https://zhuanlan.zhihu.com/p/37164435>
- Gentoo Wiki - 内核配置指南: <https://wiki.gentoo.org/wiki/Kernel/Configuration>
- Arch Linux - 传统内核编译方法: [https://wiki.archlinux.org/title/Kernel/Traditional\\_compilation](https://wiki.archlinux.org/title/Kernel/Traditional_compilation)
- 使用 localmodconfig 进行非交互式配置: <https://stackoverflow.com/questions/47049230/linux-kernel-build-perform-make-localmodconfig-non-interactive-way>
- 内核编译默认配置最佳实践: <https://unix.stackexchange.com/questions/29439/compiling-the-kernel-with-default-configurations>
- 内核编译后的增量更新: <https://stackoverflow.com/questions/22900073/compiling-linux-kernel-after-making-changes>
- 编译时仅输出错误和警告: <https://unix.stackexchange.com/questions/71154/only-output-errors-warnings-when-compile-kernel>

## 系统调用实现

- Linux 内核官方文档 - 添加系统调用: <https://www.kernel.org/doc/html/latest/process/adding-syscalls.html>
- 向 Linux 内核添加系统调用完整教程: <https://techexplorer42.medium.com/adding-a-system-call-to-linux-kernel-d1e532b18e4e>
- Linux 系统调用实现简明指南: <https://www.linkedin.com/pulse/simple-guide-implementing-system-call-linux-kernel-manasi-singh-nwnzc>
- Linux 系统调用执行模型深入分析: <https://www.opensourceforu.com/2024/09/the-linux-system-call-execution-model-an-insight/>
- Baeldung - Linux 内核系统调用实现: <https://www.baeldung.com/linux/kernel-system-call-implementation>
- GetVM - 编写系统调用交互式教程: <https://getvm.io/tutorials/write-a-system-call>
- 使用 strace 探索系统调用: <https://learn.redhat.com/t5/General/Beyond-the-Veil-Exploring-System-Calls/td-p/45910>
- Ubuntu 20.04 中添加系统调用: <https://dev.to/jasper/adding-a-system-call-to-the-linux-kernel-5-8-1-in-ubuntu-20-04-lts-2ga8>

## procfs 文件系统

- Linux 内核官方文档 - /proc 文件系统: <https://docs.kernel.org/filesystems/proc.html>
- The /proc Filesystem (v5.19): <https://www.kernel.org/doc/html/v5.19/filesystems/proc.html>
- GitHub - proc.rst 源码文档: <https://github.com/torvalds/linux/blob/master Documentation/filesystems/proc.rst>
- procfs 手册页说明: <https://systutorials.com/docs/linux/man/5-procfs>
- /proc 如何列举 PID 的原理: <https://ops.tips/blog/how-is-proc-able-to-list-pids/>
- 创建 procfs 条目（详细教程）: <https://embedronicx.com/tutorials/linux/device-drivers/procfs-in-linux/#CreatingProcfsEntry>
- 创建 proc 读写条目实例: <https://tuxthink.blogspot.com/2017/03/creating-proc-read-and-write-entry.html>

## 内核开发技术细节

- 新版内核中获取 UID 的方法变更: <https://stackoverflow.com/questions/39229639/how-to-get-current-processs-uid-and-euid-in-linux-kernel-4-2/39230936>
- 通过 PID 查找进程的替代方法: <https://stackoverflow.com/questions/24447841/alternative-for-find-task-by-pid>
- 内核空间访问用户空间指针: <https://stackoverflow.com/questions/59772132/how-to-correctly-extract-a-string-from-a-user-space-pointer-in-kernel-space>
- 导出内核符号供其他模块使用: <https://www.kernel.org/doc/html/latest/kbuild/modules.html>

## 系统启动与 GRUB 配置

- Ubuntu GRUB2 启动菜单配置: [https://blog.csdn.net/luyu\\_embedded/article/details/44353499](https://blog.csdn.net/luyu_embedded/article/details/44353499)
- 显示 GRUB2 菜单的方法: <https://unix.stackexchange.com/questions/373402/cant-get-grub2-menu-to-display>
- 保存或导出自定义内核配置: <https://superuser.com/questions/439511/how-to-save-or-export-a-custom-linux-kernel-configuration>

## 其他参考资料

- 完整但略显过时的内核教程: <https://www.cnblogs.com/hellovenus/p/3967597.html>
- Linux 内核学习笔记: [https://www.cnblogs.com/hellovenus/p/os\\_linuxcore\\_study.html](https://www.cnblogs.com/hellovenus/p/os_linuxcore_study.html)
- Make 配置详解: <https://www.jianshu.com/p/876043f48120>