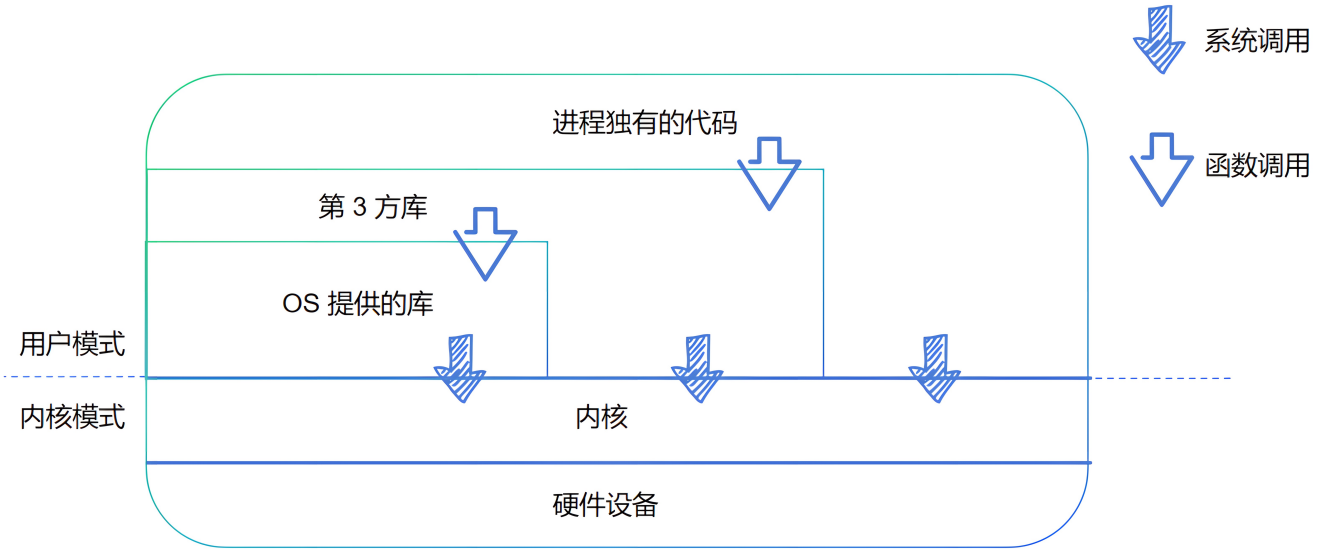
用户模式

OS 并非仅仅由内核(内核模式里运行的程序)构成，还包含许多在用户模式运行的程序。这些程序有的以库的形式存在，有的作为单独的进程而运行：

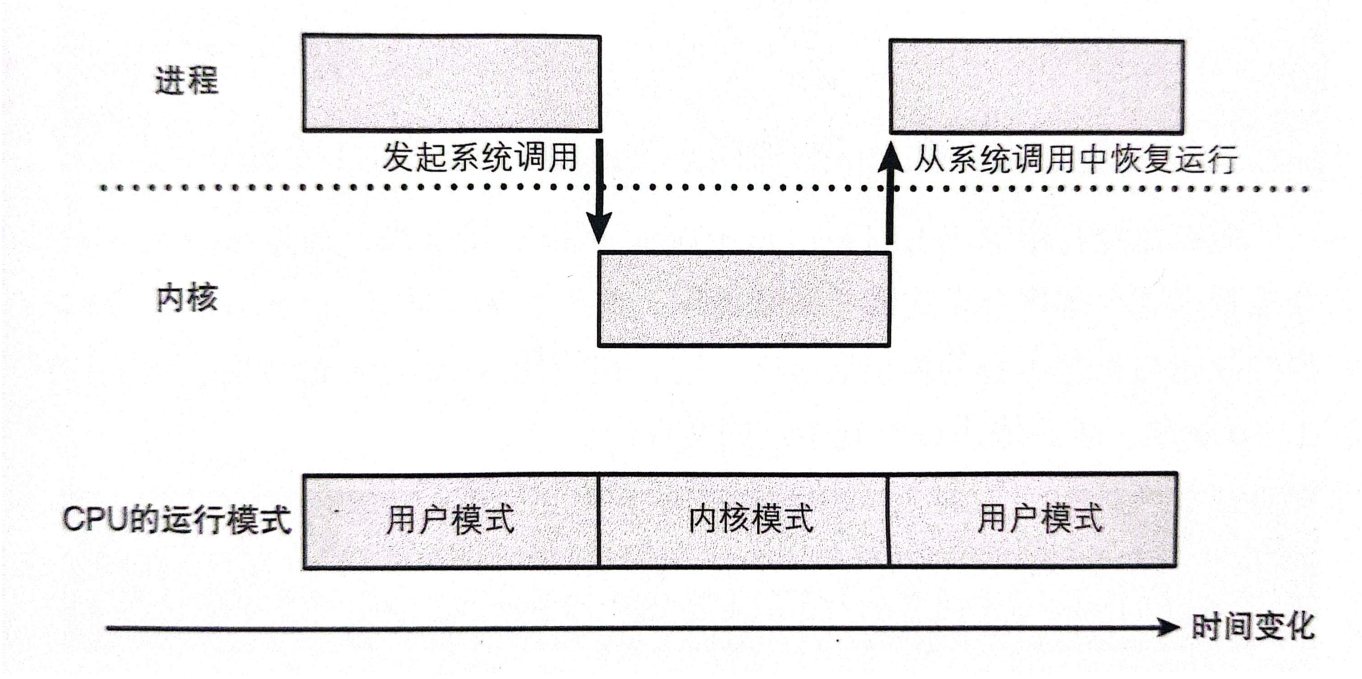


1. 系统调用

进程如果要执行需要依赖于内核的处理(操作硬件)，需要通过系统调用向内核发出请求，一般有如下类型：

- 进程控制(创建和删除)
- 内存管理(分配和释放)
- 进程间通信
- 网络管理
- 文件系统操作
- 文件操作(访问设备)

用户模式下系统调用向内核发送请求，CPU 会发生中断，从用户模式切换到内核模式，处理完系统调用后，切换回用户模式。



> 期间，内核会检查来自进程的请求是否合理，不合理则系统调用失败；用户进程必须经过系统调用才能切换到 CPU 运行模式

1.1 查看系统调用

使用 `strace` 命令可以对一个进程进行追踪。

例如，首先按照参考书，我们写一个简单的 hello.cpp:

```
syz@syz:~/projects$ cat hello.cpp
#include <iostream>

int main() {
        printf("hello, world!\n");
        return 0;
}
```

接着编译 `g++ -o hello hello.cpp` 得到可执行文件

依次执行：

- `./hello` : 直接执行
- `strace -o hello.log ./hello`: 执行并且使用 strace 追踪系统调用
- `cat hello.log` : 显示 hello.log 里的内容

```
syz@syz:~/projects$ ./hello
hello, world!
syz@syz:~/projects$ strace -o hello.log ./hello
hello, world!
syz@syz:~/projects$ cat hello.log
execve("./hello", ["./hello"], 0x7ffff05adf60 /* 29 vars */) = 0
brk(NULL)                               = 0x55cc79652000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe0c1e0b20) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe5d6553000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=16963, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16963, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe5d654e000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
```

strace 的运行结果的每一行对应一个系统调用，其中最直白且关键的内容是：

`write(1, "hello, world!\n", 14) = 14`

> 该命令使用 write() 系统调用将 hello world\n 打印到屏幕上，14 代表写入了14字节的数据

接下来，使用 `python` 完成类似的操作，使用 `python` 编写一个输出 `hello world` 的程序，接着使用：

`strace -o hello.py.log python3 ./hello.py`

```
syz@syz:~/projects$ cat hello.py
print("hello, world!")
syz@syz:~/projects$ strace -o hello.py.log python3 ./hello.py
hello, world!
syz@syz:~/projects$ cat hello.py.log
execve("/usr/bin/python3", ["python3", "./hello.py"], 0x7fff68d13b88 /* 29 vars */) = 0
brk(NULL)                               = 0x562b4531f000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdb4116150) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4bbf701000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=16963, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16963, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4bbf6fc000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
```

然后注意到也出现了：`write(1, "hello, world!\n", 14) = 14`

```
lseek(3, 0, SEEK_SET)                   = 0
read(3, "print(\"hello, world!\")\n", 4096) = 23
read(3, "", 4096)                       = 0
close(3)                                = 0
write(1, "hello, world!\n", 14)         = 14
rt_sigaction(SIGINT, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=SA_RESTORER|SA_ONSTACK, sa_restorer=0x7f4bbf3e152
0}, {sa_handler=0x562b437a0980, sa_mask=[], sa_flags=SA_RESTORER|SA_ONSTACK, sa_restorer=0x7f4bbf3e1520}, 8) = 0
munmap(0x7f4bbf0b8000, 151552)          = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

> strace 追踪 python 的运行结果事实上比追踪 c++ 的运行结果要多很多。

> 对于具体某个系统调用的作用，可以使用 `man 2 write`(2 指定是查询系统调用)

2. 实验

`sar` 命令可以获取进程在用户模式下与内核模式下的运行时间占比，给出每个 CPU 核心运行的大致内容

`sar -P ALL 1`: 每隔 1 s显示每个CPU核心的使用率的统计信息，直到按下 `Ctrl + C`退出，并显示已采集的所有数据的平均值。

```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      0.01      0.00      0.01      0.00      0.00     99.99
Average:          0      0.00      0.00      0.00      0.00      0.00    100.00
Average:          1      0.00      0.00      0.00      0.00      0.00    100.00
Average:          2      0.00      0.00      0.00      0.00      0.00    100.00
Average:          3      0.00      0.00      0.00      0.00      0.00    100.00
Average:          4      0.00      0.00      0.00      0.00      0.00    100.00
Average:          5      0.00      0.00      0.00      0.00      0.00    100.00
Average:          6      0.00      0.00      0.00      0.00      0.00    100.00
Average:          7      0.00      0.00      0.00      0.00      0.00    100.00
Average:          8      0.12      0.00      0.00      0.00      0.00     99.88
Average:          9      0.00      0.00      0.00      0.00      0.00    100.00
Average:         10      0.00      0.00      0.00      0.00      0.00    100.00
Average:         11      0.00      0.00      0.00      0.00      0.00    100.00
Average:         12      0.00      0.00      0.00      0.00      0.00    100.00
Average:         13      0.00      0.00      0.00      0.00      0.00    100.00
Average:         14      0.00      0.00      0.00      0.00      0.00    100.00
Average:         15      0.00      0.00      0.12      0.00      0.00     99.88
Average:         16      0.00      0.00      0.00      0.00      0.00    100.00
Average:         17      0.00      0.00      0.00      0.00      0.00    100.00
Average:         18      0.00      0.00      0.00      0.00      0.00    100.00
Average:         19      0.00      0.00      0.00      0.00      0.00    100.00
```

- all 字段表示 20 个 CPU 核心各项数据的平均值。
- **%user** 以及 **%nice** 字段值相加是用户模式运行时间占比。
- **%system** 为 CPU 核心执行系统调用等处理占用的时间比例。
- 其它字段之后解释。

`sar -P ALL 1 1`: 第 4 个参数可以指定采样的次数。

Test One

我们可以写一段不发起任何系统调用的仅执行循环的程序来观察其用户模式运行时间与内核模式运行时间占比。

```
syz@syz:~/projects$ cat loop.cpp
#include <iostream>

int main() {
        while(true);
}
```

编译运行，然后后台挂起进程，如下图这里的进程编号为 `457`：

```
syz@syz:~/projects$ g++ loop.cpp -o loop
syz@syz:~/projects$ ./loop &
[1] 457
syz@syz:~/projects$ sar -P ALL 1 1
```

观察 `sar` 指令的输出:

```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      5.00      0.00      0.00      0.00      0.00     95.00
Average:          0      0.00      0.00      0.00      0.00      0.00    100.00
Average:          1      0.00      0.00      0.00      0.00      0.00    100.00
Average:          2      0.00      0.00      0.00      0.00      0.00    100.00
Average:          3      0.00      0.00      0.00      0.00      0.00    100.00
Average:          4      0.00      0.00      0.00      0.00      0.00    100.00
Average:          5      0.00      0.00      0.00      0.00      0.00    100.00
Average:          6      0.00      0.00      0.00      0.00      0.00    100.00
Average:          7      0.00      0.00      0.00      0.00      0.00    100.00
Average:          8      0.00      0.00      0.00      0.00      0.00    100.00
Average:          9      0.00      0.00      0.00      0.00      0.00    100.00
Average:         10      0.00      0.00      0.00      0.00      0.00    100.00
Average:         11      0.00      0.00      0.00      0.00      0.00    100.00
Average:         12      0.00      0.00      0.00      0.00      0.00    100.00
Average:         13      0.00      0.00      0.00      0.00      0.00    100.00
Average:         14      0.00      0.00      0.00      0.00      0.00    100.00
Average:         15      0.00      0.00      0.00      0.00      0.00    100.00
Average:         16    100.00      0.00      0.00      0.00      0.00      0.00
Average:         17      0.00      0.00      0.00      0.00      0.00    100.00
Average:         18      0.00      0.00      0.00      0.00      0.00    100.00
Average:         19      0.00      0.00      0.00      0.00      0.00    100.00
```

可以看到，该进程(只进行循环)，用户模式的占比达到 **100%**;实验结束后，使用 `kill id` 杀掉该程序的进程，比如我这里是 `kill 457`，杀掉该进程后，用户模式占比为 **0**.

Test Two

首先介绍 **<unistd.h>** 库(C/C++)里面的系统调用函数 `getppid()`，用于获取 **父进程** 的进程 ID。

> 一段运行于 shell 下的程序，其父进程为运行该 shell 的进程，因此，getppid() 可以得到 shell 的进程 ID。

Small Test

首先你可以在 shell 里使用 `echo $$` 获取当前 shell 的 ID，然后编写一段 C/C++ 程序，大致为：

```
syz@syz:~/projects$ echo $$
341
syz@syz:~/projects$ cat ppid.cpp
#include <unistd.h>
#include <bits/stdc++.h>
#include <sys/types.h>

int main() {
        std::cout << getppid() << '\n';
}
syz@syz:~/projects$ g++ ppid.cpp -o ppid && ./ppid
341
syz@syz:~/projects$
```

> 证毕，这为我们接下来的实验提供了一点基础。

接下来，我们编写一段用户模式和内核模式交替的程序，运行挂起后并使用 `sar` 指令查看 CPU 情况：

```
syz@syz:~/projects$ cat ppidloop.cpp
#include <sys/types.h>
#include <unistd.h>

int main() {
        while (true) getppid();
}
syz@syz:~/projects$ g++ ppidloop.cpp -o ppidloop
syz@syz:~/projects$ ./ppidloop &
[1] 845
syz@syz:~/projects$ ps
    PID TTY          TIME CMD
    341 pts/0    00:00:00 bash
    845 pts/0    00:00:01 ppidloop
    846 pts/0    00:00:00 ps
syz@syz:~/projects$ sar -P ALL 1 1
```

> 注意，不要使用 `g++ ppidloop.cpp -o ppidloop && ./ppidloop &`，因为这会创建一个新的 bash 进程来运行 ppidloop 程序，这样之后，它将返回新创建的 bash 进程的 ID，你需要使用 `ps` 命令查看其它多余进程并将其杀掉。

在第 7 块 CPU 核心上：

- 运行 ppidloop 程序本身占用 36% 的运行时间
- getppid() 这一系统调用占用了 64% 的运行时间
- %idle 字段为 0

```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all      1.80      0.00      3.20      0.00      0.00     95.00
Average:          0      0.00      0.00      0.00      0.00      0.00    100.00
Average:          1      0.00      0.00      0.00      0.00      0.00    100.00
Average:          2      0.00      0.00      0.00      0.00      0.00    100.00
Average:          3      0.00      0.00      0.00      0.00      0.00    100.00
Average:          4      0.00      0.00      0.00      0.00      0.00    100.00
Average:          5      0.00      0.00      0.00      0.00      0.00    100.00
Average:          6      0.00      0.00      0.00      0.00      0.00    100.00
Average:          7     36.00      0.00     64.00      0.00      0.00      0.00
Average:          8      0.00      0.00      0.00      0.00      0.00    100.00
Average:          9      0.00      0.00      0.00      0.00      0.00    100.00
Average:         10      0.00      0.00      0.00      0.00      0.00    100.00
Average:         11      0.00      0.00      0.00      0.00      0.00    100.00
Average:         12      0.00      0.00      0.00      0.00      0.00    100.00
Average:         13      0.00      0.00      0.00      0.00      0.00    100.00
Average:         14      0.00      0.00      0.00      0.00      0.00    100.00
Average:         15      0.00      0.00      0.00      0.00      0.00    100.00
Average:         16      0.00      0.00      0.00      0.00      0.00    100.00
Average:         17      0.00      0.00      0.00      0.00      0.00    100.00
Average:         18      0.00      0.00      0.00      0.00      0.00    100.00
Average:         19      0.00      0.00      0.00      0.00      0.00    100.00
```

> 如果系统系调用很高，说明出现了类似如上这种导致系统负载过高的程序。

Tip

同时 `strace` 命令不仅可以追踪程序的系统调用，还可以得到其系统调用的时间：

```
syz@syz:~/projects$ strace -T -o hello.log ./hello
hello, world!
syz@syz:~/projects$ cat hello.log
execve("./hello", ["./hello"], 0x7ffe725144b8 /* 29 vars */) = 0 <0.000388>
brk(NULL)                               = 0x563276fdd000 <0.000023>
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe47ef96e0) = -1 EINVAL (Invalid argument) <0.000023>
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f75408d6000 <0.000069>
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory) <0.000026>
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3 <0.000026>
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=16963, ...}, AT_EMPTY_PATH) = 0 <0.000023>
```

```
brk(0x563276ffe000)                     = 0x563276ffe000 <0.000024>
futex(0x7f75408ce77c, FUTEX_WAKE_PRIVATE, 2147483647) = 0 <0.000054>
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0 <0.000025>
write(1, "hello, world!\n", 14)         = 14 <0.000040>
exit_group(0)                           = ?
```

可以看到调用 `write` 输出14字节的数据花了 40 微秒

3. 系统调用的包装函数

事实上对于 C/C++ 这种高级语言，并不能直接发起系统调用，只能通过汇编代码发起系统调用，对于 x86_64 架构，getppid() 通过如下代码发起：

```
mov $0x6e,%eax
syscall
```

> 首先将系统调用编号 0x6e 传递给 eax寄存器，然后 syscall 命令切换到内核模式，接下来执行负责处理 getppid 的内核代码

OS 提供了一些列被称为 **系统调用的包装函数** 的函数，从而避免了程序员必须为每一个系统调用编写相应的汇编代码，然后从高级语言调用这些代码，从而更好地提升了程序的可移植性，程序员只需要专注高级编程语言的书写即可。