# Lab02: System Calls(写系统调用)

## System Call Tracing (Moderate)

### 1) 实验目的

本实验的目的是在 xv6 操作系统中添加系统调用追踪功能，以便在调试时提供帮助。通过实现一个新的 `trace` 系统调用，可以控制哪些系统调用被追踪，并在这些系统调用返回时打印出相关信息，包括进程 ID、系统调用名称和返回值。

### 2) 实验步骤

1. **添加系统调用原型和系统调用号**:
   - 在 `user/user.h` 中添加 `trace` 系统调用的原型:

     ```
     int trace(int mask);
     ```

   - 在 `kernel/syscall.h` 中添加 `SYS_trace` 的系统调用号:

     ```
     #define SYS_trace 22
     ```

2. **生成系统调用存根**:
   - 修改 `user/usys.pl`，添加 `trace` 系统调用:

     ```
     entry("trace");
     ```

3. **实现 `trace` 系统调用**:
   - 在 `kernel/sysproc.c` 中实现 `sys_trace` 函数:

     ```
     int sys_trace(void) {
       int mask;
       if (argint(0, &mask) < 0)
         return -1;
       myproc()->tracemask = mask;
       return 0;
     }
     ```

4. **修改 `fork` 函数**:
   - 在 `kernel/proc.c` 中，确保子进程继承父进程的追踪掩码:

```
np->tracemask = p->tracemask;
```

5. **修改 `syscall` 函数**：

   - 在 `kernel/syscall.c` 中，添加系统调用名称数组：

```
static char *syscallnames[] = {
  // ... existing syscall names ...
  [SYS_trace] "trace",
};
```

   - 修改 `syscall` 函数以打印追踪信息：

```
void syscall(void) {
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if (p->tracemask & (1 << num)) {
      printf("%d: syscall %s -> %d\n", p->pid, syscallnames[num], p-
>trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

6. **编译和运行**：

   - 在 `Makefile` 中将 `trace` 添加到 `UPROGS` 中：

```
UPROGS = ... _trace ...
```

   - 运行 `make qemu` 并测试 `trace` 程序：

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
```

### 3) 实验中遇到的困难和解决办法

1. **系统调用编号冲突**：
   - **困难描述**：在添加新的系统调用编号时，可能会与现有的编号冲突。
   - **解决办法**：确保在 `kernel/syscall.h` 中添加 `SYS_trace` 编号时使用未占用的编号。
2. **系统调用参数传递错误**：
   - **困难描述**：在实现 `sys_trace` 时，可能会由于参数传递错误导致系统调用无法正常工作。
   - **解决办法**：仔细检查 `argint` 函数的使用，确保正确传递和接收参数。
3. **文件描述符资源耗尽**：
   - **困难描述**：在实现 `fork` 函数时，如果不正确关闭不需要的文件描述符，会导致文件描述符资源耗尽。
   - **解决办法**：在 `fork` 中正确处理文件描述符，确保在适当时候关闭不需要的描述符。

### 4) 实验心得

通过本次实验，我深入理解了 xv6 操作系统中系统调用的实现过程，并掌握了如何通过添加自定义系统调用来扩展操作系统的功能。实验中遇到的问题让我更加熟悉了系统调用的参数传递和进程间通信机制。在调试过程中，我学会了如何利用系统调用追踪功能来排查问题，这对后续的实验和项目开发具有重要意义。整体上，这次实验不仅巩固了我对操作系统课程理论知识的理解，也提升了我在实际操作系统开发中的实践能力。

## Sysinfo 系统调用(Moderate)

### 1）实验目的

本次实验的目的是在 xv6 操作系统中添加一个名为 `sysinfo` 的系统调用，该调用收集运行系统的相关信息。具体来说，`sysinfo` 调用将返回系统中当前空闲内存的字节数以及非 `UNUSED` 状态的进程数量。我们提供了一个测试程序 `sysinfotest`，当其输出 "sysinfotest: OK" 时表示实验成功。

### 2）实验步骤

1. **添加系统调用声明和用户空间函数**
   - 在 `user/user.h` 中声明 `sysinfo` 系统调用。由于我们需要使用 `sysinfo` 结构体，因此需要提前声明：

     ```
     struct sysinfo;
     int sysinfo(struct sysinfo *);
     ```

   - 在 `user/usys.pl` 中添加 `sysinfo` 的声明以生成系统调用的汇编代码。
2. **定义 sysinfo 结构体**
   - 在 `kernel/sysinfo.h` 中定义 `sysinfo` 结构体，该结构体包含两个字段：`freemem` 和 `nproc`。

```
struct sysinfo {
    uint freemem;
    uint nproc;
};
```

3. **实现 sysinfo 系统调用**

   - 在 `kernel/sysproc.c` 中实现 `sysinfo` 系统调用。该调用需要填充 `sysinfo` 结构体，并将其拷贝到用户空间。

```
extern uint64 freemem();
extern uint64 proc_count();

uint64
sys_sysinfo(void)
{
    struct sysinfo info;
    uint64 addr;

    if(argaddr(0, &addr) < 0)
        return -1;

    info.freemem = freemem();
    info.nproc = proc_count();

    if(copyout(myproc()->pagetable, addr, (char *)&info, sizeof(info))
< 0)
        return -1;

    return 0;
}
```

4. **收集系统信息**

   - **收集空闲内存**： 在 `kernel/kalloc.c` 中添加 `freemem` 函数，该函数计算系统中当前空闲的内存量。

```
uint64
freemem(void)
{
    struct run *r;
    uint64 free = 0;

    acquire(&kmem.lock);
    for(r = kmem.freelist; r; r = r->next)
        free += PGSIZE;
    release(&kmem.lock);
```

```
    return free;
}
```

- **收集进程数量**： 在 `kernel/proc.c` 中添加 `proc_count` 函数，该函数计算非 `UNUSED` 状态的进程数量。

```
uint64
proc_count(void)
{
    struct proc *p;
    int count = 0;

    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->state != UNUSED)
            count++;
    }

    return count;
}
```

5. **测试和调试**

- 修改 Makefile，确保 `sysinfotest` 已添加到 `UPROGS` 中。
- 编译并运行 xv6，在 xv6 shell 中执行 `sysinfotest` 以验证实现是否正确。

```
make clean
make qemu
```

## 3）实验中遇到的困难和解决办法

1. **系统调用注册问题**

- **困难描述**：在实现 `sysinfo` 系统调用时，可能会遗漏在多个文件中进行必要的注册和声明，导致编译或运行时错误。
- **解决办法**：仔细检查所有涉及的文件，确保在 `user/user.h`、`user/usys.pl`、`kernel/syscall.h`、`kernel/sysproc.c` 等文件中正确声明和注册系统调用。

2. **内存信息收集**

- **困难描述**：在计算系统空闲内存时，可能会误用锁或计算错误。
- **解决办法**：仔细阅读 `kalloc.c` 中的代码，确保在计算空闲内存时正确使用锁，并且正确遍历空闲列表。

3. **进程状态统计**

- **困难描述**：在统计非 `UNUSED` 状态的进程数量时，可能会漏掉某些状态。
- **解决办法**：仔细检查 `proc` 数组中的所有状态，确保正确统计所有非 `UNUSED` 状态的进程。

### 4）实验心得

通过本次实验，我深入理解了 xv6 操作系统中系统调用的实现过程。添加一个新的系统调用需要在多个文件中进行修改和注册，包括用户空间和内核空间的函数声明、系统调用号的分配、实际功能的实现以及返回结果的拷贝等。通过实现 `sysinfo` 系统调用，我不仅熟悉了 xv6 中内存和进程管理的基本机制，还掌握了如何在内核中收集和返回系统信息。此外，在调试过程中，我学会了如何处理和解决常见的编译和运行时错误，这为我将来进行更多复杂的操作系统开发打下了坚实的基础。