Lab1 实验报告

Author: 刘佳隆

Student ID: 518010910009

Email: liujl01@sjtu.edu.cn

思考题 4

思考内核从完成必要的初始化到第一次切换到用户态程序的过程是怎么样的？尝试描述一下调用关系。

首先内核在完成初始化之后，通过`create_root_thread()`完成了第一个用户进程与用户线程的创建，而后通过`sched()`函数进行调度，由于此时只存在一个用户进程，因此会调度到我们刚刚创建的用户进程。然后通过`switch_context()`进行上下文的切换，最终将通过`eret_to_thread`切换到被选择的用户进程。

思考 7

尝试描述 printf 如何调用到 chcore_stdout_write 函数。

首先在`printf`函数中调用`vfprintf`

```
int printf(const char *restrict fmt, ...)
{
    int ret;
    va_list ap;
    va_start(ap, fmt);
    ret = vfprintf(stdout, fmt, ap);
    va_end(ap);
    return ret;
}
```

在`vprintf`函数中先完成格式化、检查等预处理，而后调用`printf_core(f, fmt, &ap2, nl_arg, nl_type)`，此时关联文件流 f 进行实际的格式化输出

```
    f->wpos = f->wbase = f->wend = 0;
}
if (!f->wend && __towrite(f)) ret = -1;
else ret = printf_core(f, fmt, &ap2, nl_arg, nl_type);
if (saved_buf) {
```

在`printf_core`中，通过调用`out`函数进行输出，由于多处调用了`out`函数且我们的目的是追踪如何调用到`chcore_stdout_write`函数，此处省略调用处的代码展示。

而后`out`函数调用`__fwritex`函数

```c
static void out(FILE *f, const char *s, size_t l)
{
    if (!(f->flags & F_ERR)) __fwritex((void *)s, l, f);
}
```

在 `__fwritex` 函数中，调用 `f->write` 函数，此处的 `f` 即为 `stdout`

```c
size_t __fwritex(const unsigned char *restrict s, size_t l, FILE *restrict f)
{
    size_t i=0;

    if (!f->wend && __towrite(f)) return 0;

    if (l > f->wend - f->wpos) return f->write(f, s, l);

    if (f->lbf >= 0) {
        /* Match /^(.*\n|)/ */
        for (i=l; i && s[i-1] != '\n'; i--);
        if (i) {
            size_t n = f->write(f, s, i);
            if (n < i) return n;
            s += i;
            l -= i;
        }
    }

    memcpy(f->wpos, s, l);
    f->wpos += l;
    return l+i;
}
```

通过 `user/chcore-libc/musl-libc/src/stdio/stdout.c` 可以看到 `stdout.write` 被定义为 `__stdout_write`

```c
static unsigned char buf[BUFSIZ+UNGET];
hidden FILE __stdout_FILE = {
    .buf = buf+UNGET,
    .buf_size = sizeof buf-UNGET,
    .fd = 1,
    .flags = F_PERM | F_NORD,
    .lbf = '\n',
    .write = __stdout_write,
    .seek = __stdio_seek,
    .close = __stdio_close,
    .lock = -1,
};
FILE *const stdout = &__stdout_FILE;
FILE *volatile __stdout_used = &__stdout_FILE;
```

在 `__stdout_write` 函数中调用到 `__stdio_write` 函数

```c
size_t __stdout_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct winsize wsz;
    f->write = __stdio_write;
    if (!(f->flags & F_SVB) && __syscall(SYS_ioctl, f->fd, TIOCGWINSZ, &wsz))
        f->lbf = -1;
    return __stdio_write(f, buf, len);
}
```

在 `__stdio_write` 函数中通过 `syscall(SYS_writev, ...)` 进行系统调用

```c
size_t __stdio_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct iovec iovs[2] = {
        { .iov_base = f->wbase, .iov_len = f->wpos-f->wbase },
        { .iov_base = (void *)buf, .iov_len = len }
    };
    struct iovec *iov = iovs;
    size_t rem = iov[0].iov_len + iov[1].iov_len;
    int iovcnt = 2;
    ssize_t cnt;
    for (;;) {
        cnt = syscall(SYS_writev, f->fd, iov, iovcnt);
        if (cnt == rem) {
            f->wend = f->buf + f->buf_size;
            f->wpos = f->wbase = f->buf;
            return len;
        }
```

在 `user/chcore-libc/musl-libc/src/chcore-port/syscall_dispatcher.c` 中可以看到，具有三个参数的 `SYS_writev` 调用进一步调用六参数的 `SYS_writev`，而后调用 `chcore_writev`

```c
case SYS_writev: {      messith, 5个月前 • initial commit
        return __syscall6(SYS_writev, a, b, c, 0, 0, 0);
}
```

```c
case SYS_writev: {      messith, 5个月前 • initial commit
        return chcore_writev(a, (const struct iovec *)b, c);
}
```

`chcore_writev` 调用 `chcore_write`

```c
ssize_t chcore_writev(int fd, const struct iovec *iov, int iovcnt)
{
        int iov_i;
        ssize_t byte_written, ret;

        if ((ret = iov_check(iov, iovcnt)) != 0)
                return ret;

        byte_written = 0;
        for (iov_i = 0; iov_i < iovcnt; iov_i++) {
                ret = chcore_write(fd,      messith, 5个月前 • initial co
                                   (void *)((iov + iov_i)->iov_base),
                                   (size_t)(iov + iov_i)->iov_len);
```

在`chcore_write`函数中，调用了`fd_dic[fd]->fd_op->write`，此处的`fd`即为`stdout`

```
ssize_t chcore_write(int fd, void *buf, size_t count)
{
        if (fd < 0 || fd_dic[fd] == 0)
                return -EBADF;
        return fd_dic[fd]->fd_op->write(fd, buf, count);
}
```

`stdout`对应的`fd_op`可以在`user/chcore-libc/musl-libc/src/chcore-port/stdio.c`看到，因此对其`write`函数的调用就是调用了`chcore_stdout_write`

```
struct fd_ops stdout_ops = {
        .read = chcore_stdio_read,
        .write = chcore_stdout_write,
        .close = chcore_stdout_close,
        .poll = chcore_stdio_poll,
        .ioctl = chcore_stdio_ioctl,
        .fcntl = chcore_stdio_fcntl,
};
```