

操作系统：实验大作业

王天一

320200931301

日期：2023 年 6 月 13 日

1 实验内容

- **进程间通信**: Linux 环境下编写一个 C 语言程序, 读取一个数据文件 (自定义一个大的整数文件), 对每一个数据进行某种运算, 再在屏幕输出计算结果。要求以上工作用 3 个进程实现, 进程 1 负责读文件, 进程 2 进行计算, 进程 3 负责将计算结果输出到文件 `result.txt` 中。进程间通信分别使用管道、共享内存机制, 并比较不同方式的时间性能。
- **内存管理**: Linux 下编写一个程序, 利用内存映象文件, 实现 `less` 工具的功能 (多屏显示)

2 实验环境

本次实验使用的环境为一台装有 Ubuntu Server 22.04.2 LTS 操作系统的机器。其中配置如图 1 所示。

```
wty@wtymicro ~/tmp/share neofetch
```

```
.-/+00SSSS00+/-.  
`:+ssssssssssssssssss+:`  
-+ssssssssssssssssssyyssst+-  
.oSSssssssssssssssssdMMMMyssso.  
/ssssssssshdmNNmYnMMMHsssss/  
+ssssssshmydMMMMMMNdoddysssssss/  
/ssssssshNMMyhyhyyyhmNMMNHssssss/  
.sssssssdMMMNhsSSssssssshNMMDssssss.  
+ssshhhyNMMysssssssssyNMMMyssssst+  
ssyNMMyNmHssssssssssshmmhssssssso  
ssyNMMyNmHssssssssssshmmhssssssso  
+ssshhhyNMMysssssssssyNMMMyssssst+  
.sssssssdMMMNhsSSssssssshNMMDssssss.  
/ssssssshNMMyhyhyyyhdNMNHssssss/  
+sssssssdmydMMMMMMNdoddysssssss/  
/ssssssssshdmNNNNmYnMMMHsssss/  
.oSSssssssssssssssdMMMMyssso.  
-+ssssssssssssssssyyssst+-  
`:+ssssssssssssssssss+:`  
.-/+00SSSS00+/-.
```

```
wty@wtymicro  
-----  
OS: Ubuntu 22.04.2 LTS x86_64  
Kernel: 5.15.0-73-generic  
Uptime: 8 days, 5 hours, 7 mins  
Packages: 1195 (dpkg)  
Shell: zsh 5.8.1  
Resolution: 1024x768  
Terminal: /dev/pts/4  
CPU: Intel i5-5200U (4) @ 2.70GHz  
GPU: Intel HD Graphics 5500  
Memory: 817MiB / 3839MiB
```




图 1: 实验设备配置

注：为了确保实验环境的准确性，实验中的数据均使用`taskset`命令进行绑定核心的操作，故可以去掉机器核心数对结果的影响。

3 算法设计

3.1 进程间通信

对于进程间通信，我们需要设计我们的程序，使用三个进程实现以下工作：

- 进程一读取本地的文件，将读取出来的数据通过某种进程间通信手段传输给进程二。

- 进程二接收进程一发送的数据，进行某种运算（这里我们选择对 214 取模），并将结果发送给进程三。
- 进程三接收进程二发送的数据，将其写入结果的文件中。

3.1.1 基于管道的实现

管道（pipe）是一种常用的进程间通信（IPC）机制，用于在两个相关进程之间传输数据。它提供了一种简单的单向通信方式，其中一个进程充当写入数据的发送者，而另一个进程则充当接收数据的接收者。在本次实验中我使用 Linux 提供的 `pipe()` 函数来创建进程之间的匿名管道来进行进程间通信。

3.1.2 基于共享内存的实现

共享内存是一种进程间通信的高效方式，它允许多个进程共享同一块内存区域，从而避免了数据复制和操作系统内核的介入。本次实验中，我使用了 `shmget` 创建进程中的共享内存，并使用信号量机制对共享内存这一临界资源进行管理，保证其运行的正确性。

3.2 自制 less 工具

less 是一个功能强大的用于浏览文本文件的命令行工具。它可以在终端中按页显示文件内容，并提供了很多方便的导航和搜索功能。本次实验中我主要实现了 less 的以下几个功能：

- **分屏浏览**：一屏一屏的加载数据，避免全部加载数据导致的卡顿。
- **自适应窗口**：根据当前终端的高度自动调成程序的输出，保证程序可以一直占满整个终端。
- **禁用缓冲**：在运行的过程中禁用终端的输入缓冲区，实现按下某个按键程序可以直接相应的效果，而不是需要按下回车才会有反应。
- **分行移动**：可以一行一行的向前或者向后移动屏幕。

3.2.1 分屏浏览

分屏浏览的功能实现核心思路是对 `mmap` 内存映射函数的使用。在程序的开始，我们需要先将文件从硬盘映射到内存中，将文件映射成一个字符数组。

然后根据要进行的不同的操作，使用 `memcpy` 函数将文件中的内容拷贝到缓冲区，最后使用 `printf` 函数将缓冲区的内容打印到终端上。

3.2.2 自适应窗口

为了获取终端窗口的大小，我使用 `termios.h` 头文件提供的 `ioctl` 函数动态获取当前终端的高度，并将其赋值给一个全局变量，其他的函数通过此全局变量的值来调节其输出的高度。

3.2.3 禁用缓冲

为了禁用输入缓冲区，我使用 `termios.h` 头文件提供的 `tcsetattr` 函数动态设置当前终端属性，禁用标准模式和回显，实现禁用输入缓冲区的功能。

3.2.4 分行移动

为了实现分行移动，我添加了两个分支，分别是 `j` 和 `k`（对应 `vim` 的键位），当按下 `j` 时，会向下移动一行，当按下 `k` 时，会向上移动一行。

4 实验过程

4.1 进程间通信

我使用 Python 编写了一个脚本 `gen_data.py`，生成了一百万个整数，其大小为 6.9 兆字节，我们以此作为实验的测试数据。脚本的核心循环如下所示：

```
def gen_data():
    # 生成1到1000的数据
    with open('testdata.txt', 'w') as f:
        for i in range(1, 1000001):
            f.write(str(i) + '\n')
```

我使用如表 1 所示的环境对我自己的程序进行性能分析。

表 1: 测试过程中的各种选项

参数	数值
Make 版本	4.3
gcc 版本	11.3.0
编译选项	-O3（未附加-g 符号表）
计时方法	timeval（精度到微妙）
编写方式	所有函数均没进行特殊处理（static inline、Likely 等优化手段）

为了保证实验结果的可靠性，我们编译完程序源代码后，使用 `taskset` 命令将其锁到一个核心上运行五次，取其运行时间的平均值，最后两种方案的平均值如表 2 所示。

表 2: 实验结果

实现方法	第一次实验	第二次实验	第三次实验	第四次实验	第五次实验	均值
管道	8.093155s	7.985894s	8.019982s	7.972469s	8.008781s	8.0160562s
共享内存	0.234102s	0.239479s	0.215798s	0.247622s	0.228334s	0.233067s

通过结果可以看出，共享内存的结果远优于管道，这里我猜测是由于使用 `pipe()` 导致了过多的陷入（trap）导致浪费很多时间，所以我又使用了 Linux 下的 `time` 命令测试内核态和用户态花费的时间，其结果如表 3 所示，可以看到在使用管道的程序中，内核态花费了大量时间（69%），但在共享内存的程序中，内核态只花费了很少的时间（3%），符合我们的猜测。

表 3: time 测试实验结果（五次均值）

实现方法	用户态时间	用户态占百分比	内核态时间	内核态占百分比
管道	2.476s	30.9%	5.538s	69.1%
共享内存	0.226s	97.0%	0.008	3.0%

结合上面的实验和猜测，我们可以分析出管道的效率低下是反复陷入内核造成的。

4.2 自制 less 工具

在进行完进程间通信的试验后，我们可以直接使用其生成的文件作为测试文件来测试我们自制 less 工具的性能。首先是最基本的打开文件，其命令如下所示，结果如图 2 所示，可以看出打开文件的功能正常。

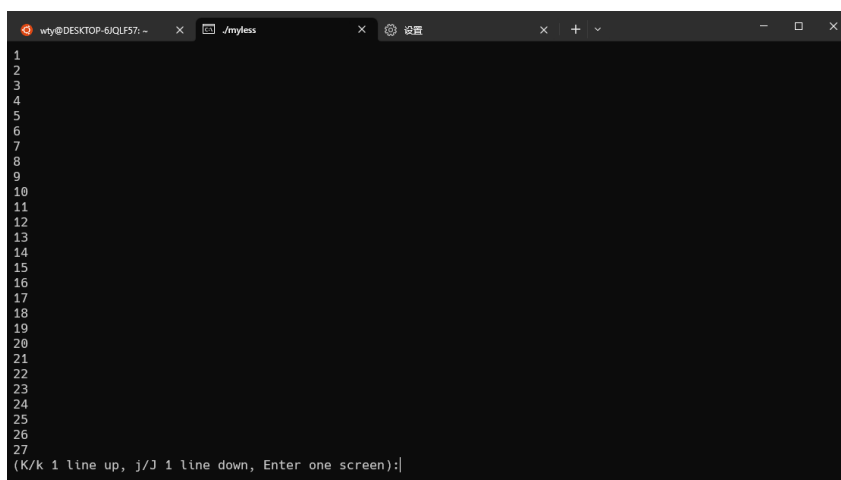


图 2: 自制 less 工具打开文件

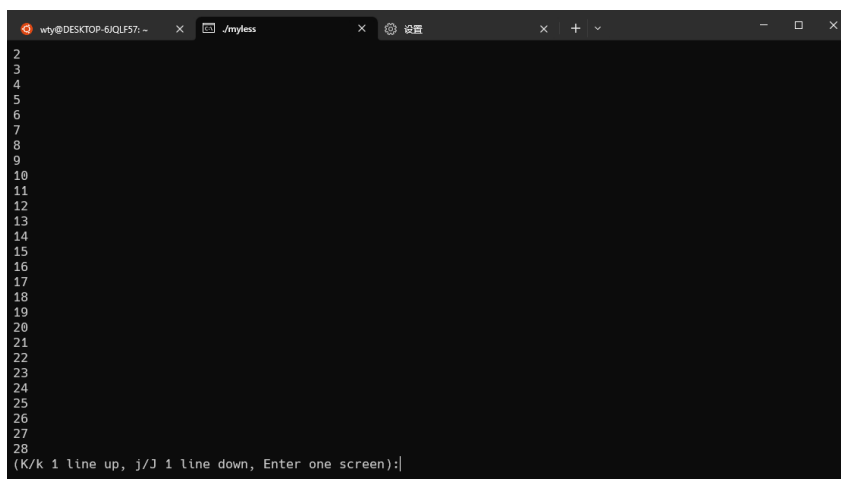


图 3: 自制 less 工具向下移动一行

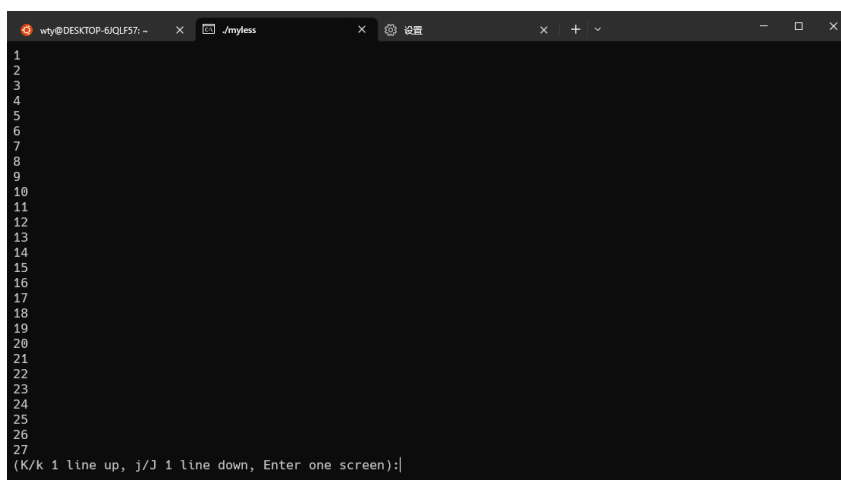


图 4: 自制 less 工具向上移动一行

```
./myless ../shared_mem.txt
```

然后测试其翻页功能，首先我们按下 j 键，测试其向下移动一页的功能，结果如图 3 所示，可以看出整个窗口显示的数据向下移动了一行，然后再按下 k 键向上移动一行，结果如图 4 所示，可以看出整个窗

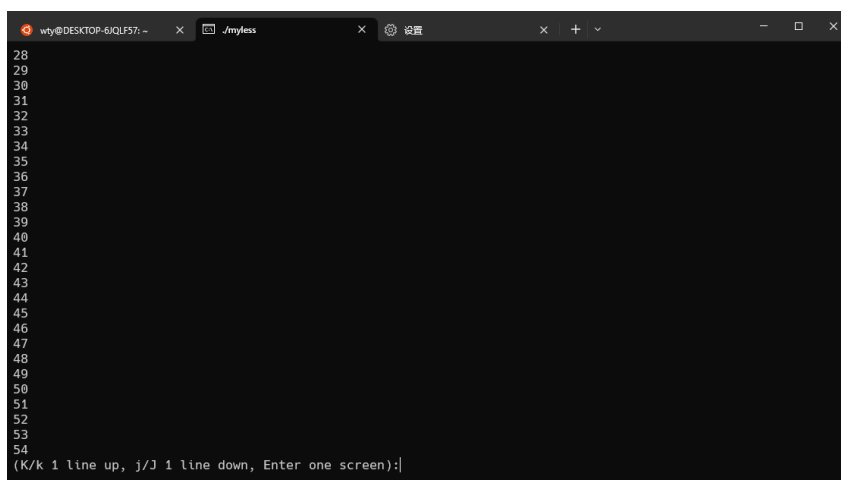


图 5: 自制 less 工具向后翻一整页

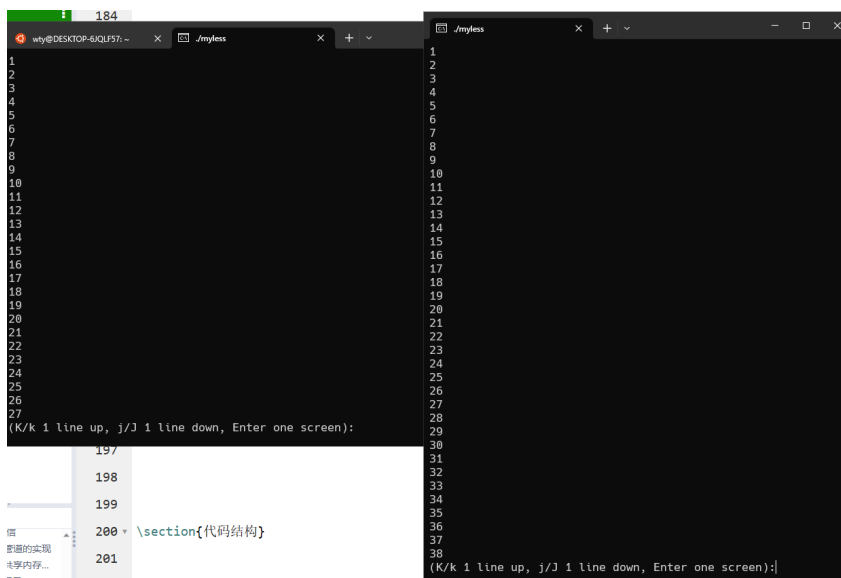


图 6: 自制 less 工具自适应高度

口显示的数据向下移动了一行，说明翻页功能正常。

再按下回车翻一整页，结果如图 5 所示，可以看出的确移动了一整页。说明翻一整页的功能也正常。

最后再测试自适应功能，打开两个终端连接机器的 ssh，然后以不同的高度打开自制 less 工具，如图 6 所示，可以看到在不同高度下，程序均能占满整个终端。

5 代码结构

本次作业的代码结构如下：

- **data**：包含生成测试数据的 Python 脚本和生成好的测试数据。
- **less**：包含自制 less 文件的源代码和 Makefile 文件，以及在 X86 平台上编译好的二进制可执行文件。
- **pipe**：包含使用管道实现实验第一部分的所有源代码和 Makefile 文件，以及在 X86 平台上编译好的二进制可执行文件。
- **result**：存储结果的文件夹，这里存放了我进行实验时候生成的数据。
- **shared_memory**：包含使用共享内存实现实验第一部分的所有源代码和 Makefile 文件，以及在 X86

平台上编译好的二进制可执行文件。

- **report**: 包含报告的所有 `tex` 源代码文件和编译好的 PDF 版本, 如果需要从源码构建, 请使用 `xelatex` 编译链进行编译。

注: 本次作业的所有代码均已开源至 [github](#), 仓库内有详细的 `git` 提交记录可作为原创证据。

6 执行结果及分析

本次实验中我首先分别基于管道和共享内存的方法进行了进程间通信的实验。实验结果显示共享内存的性能远优于进程间通信。我们通过猜测和实验证明这是频繁的使用管道会导致过多的陷入内核, 频繁的切换浪费了大量的时间。除此之外, 我还自制了一个 `less` 工具, 通过 `mmap` 函数实现了一些 `less` 工具的功能, 加深了对于内存管理的理解。