Linux 进程控制操作应用编程

📌 **实验报告**

- **实验课题作业 4：**Linux 进程控制操作应用编程
- **姓名：**梁嘉嘉
- **学号：**23125240
- **课程：**高级操作系统
- **提交日期：**2024/10/24

1. 实验目的

本实验旨在通过编写 C 语言程序，掌握Linux操作系统中的进程控制机制，包括进程的创建、进程的同步、进程间的通信以及信号处理等核心概念。通过具体的实验任务，学习如何使用 `fork()` 函数创建父子进程，如何使用信号和软中断控制进程的终止，以及如何利用无名管道实现进程间的数据通信，深入理解Linux操作系统中进程管理的基本操作和工作原理。

2. 实验环境

硬件环境：

- **虚拟机平台：**VMware Workstation Pro 17
- **处理器：**AMD Ryzen 7 8845H with Radeon 780M Graphics
  - **架构：**x86_64
  - **CPU 核心数：**2 个核心
  - **每个核心的线程数：**4
  - **L1 缓存：**64 KiB (2 实例)
  - **L2 缓存：**2 MiB (2 实例)
  - **L3 缓存：**32 MiB (2 实例)
- **内存：**4 GiB
  - **可用内存：**2.6 GiB
  - **交换分区：**3.7 GiB
- **存储：**

- **硬盘大小**：80 GB
- **分区**：单一分区 `/` 挂载在 `sda2`

软件环境：

- **操作系统**：Ubuntu 24.04.1 LTS (noble)
- **内核版本**：6.8.0-45-generic
- **编译工具**：
  - **GCC 版本**：13.2.0 (Ubuntu)
  - **Make 版本**：4.3 (GNU Make)
  - **Git 版本**：2.43.0

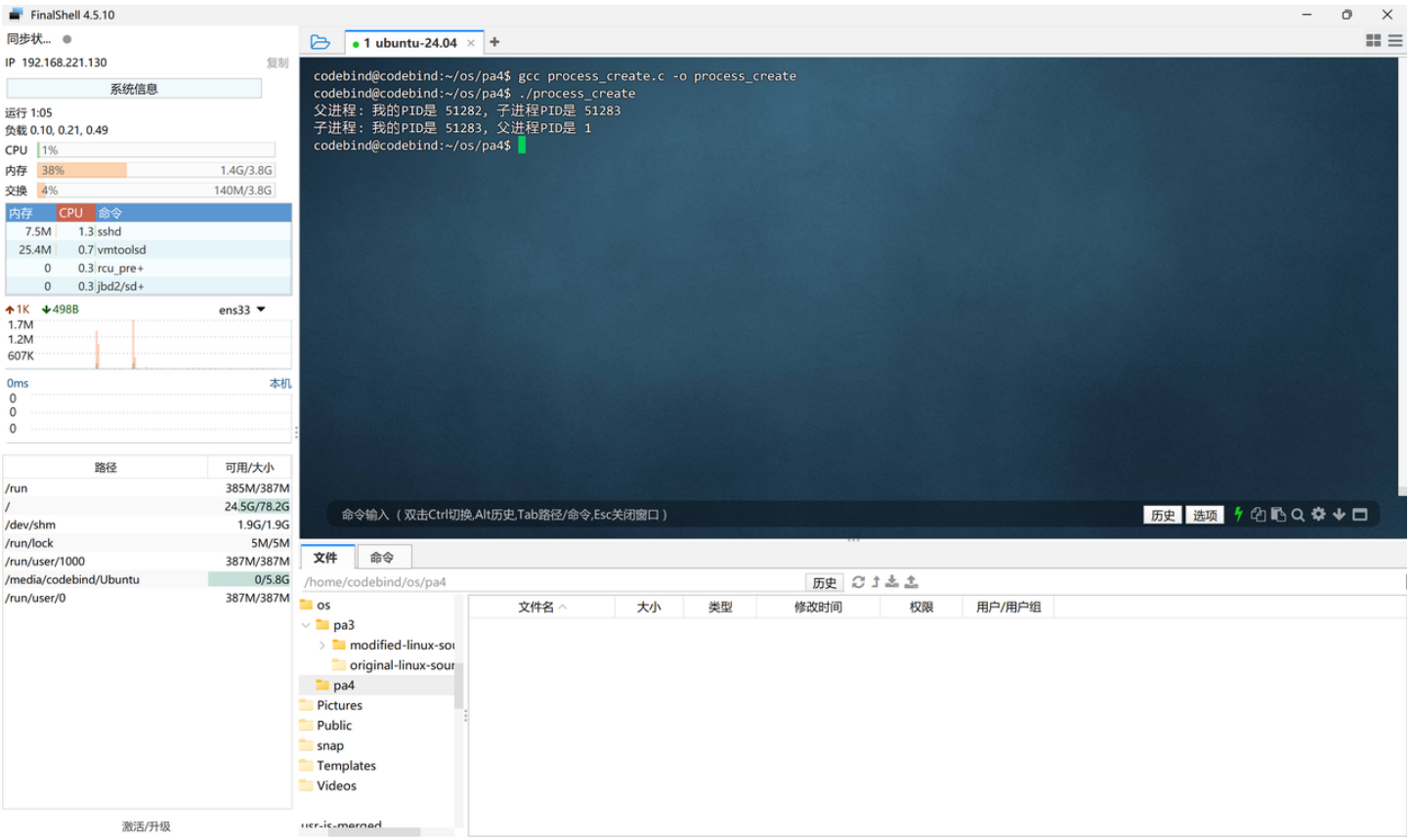
3. 实验步骤

3.1 进程创建实验

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork(); // 创建子进程

    if (pid < 0) {
        // 创建子进程失败
        printf("Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        // 子进程
        printf("子进程: 我的PID是 %d, 父进程PID是 %d\n", getpid(), getppid());
    } else {
        // 父进程
        printf("父进程: 我的PID是 %d, 子进程PID是 %d\n", getpid(), pid);
    }

    return 0;
}

```

```
$ gcc process_create.c -o process_create
```

```
2 $ ./process_create
```



3.2 字符串循环显示与终止

1. 声明了 `sig_atomic_t` 类型的全局变量 `stop` ，用于表示是否需要停止循环。

```
1 volatile sig_atomic_t stop;
```

> `volatile` 关键字告诉编译器，变量值可能会在程序运行时异步改变，因此每次访问该变量时，都需要重新读取值而不是使用缓存。

2. 信号处理函数：当程序收到 `SIGINT` 信号（即用户按下 `Ctrl+C` 时），系统会调用此函数。

在该函数中， `stop` 被设置为1，这将用于终止主循环。

```
1 void handle_sigint(int sig) {
2     stop = 1;
3 }
```

3. 主函数

```
```

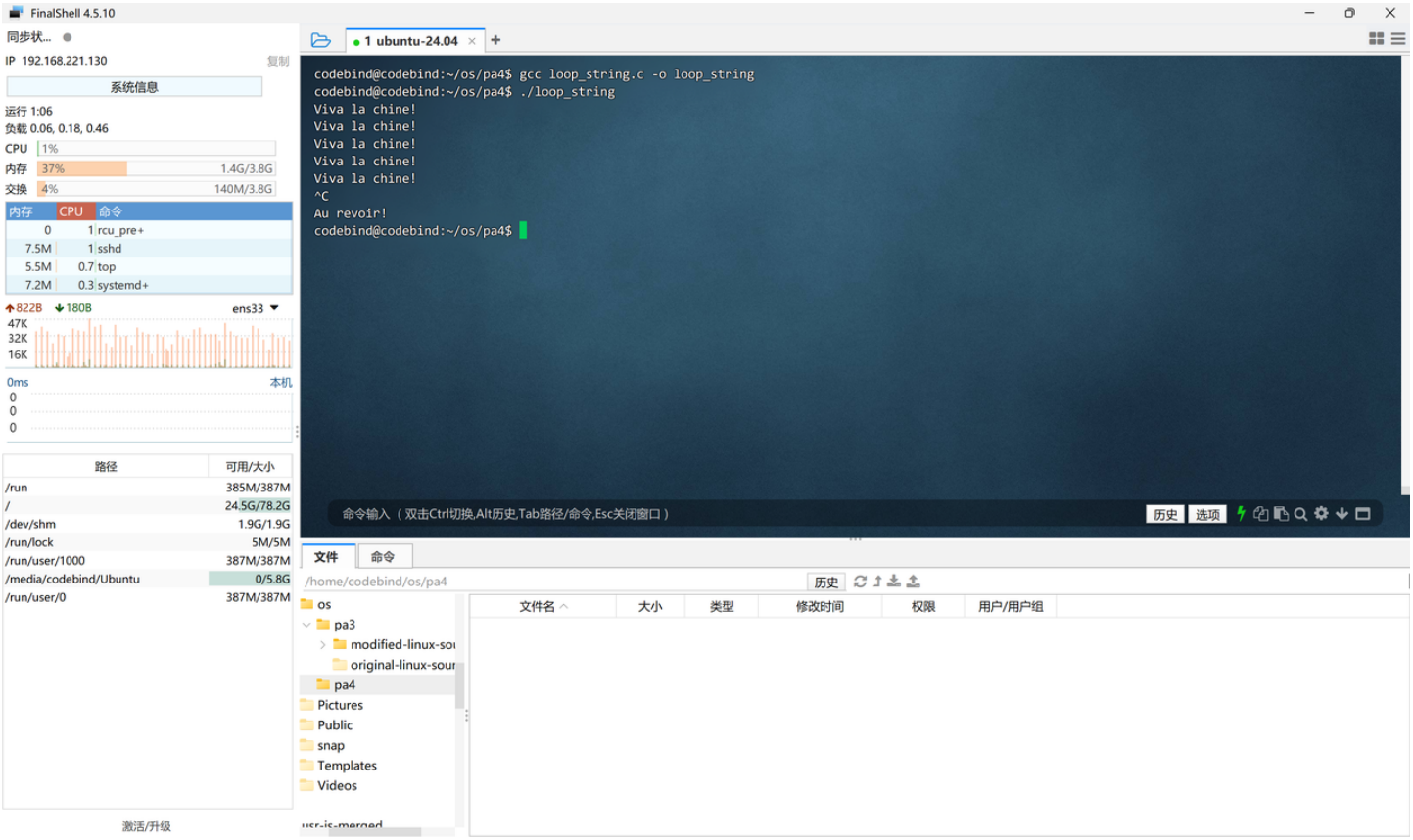
```c
int main() {
    signal(SIGINT, handle_sigint); // 捕捉 Ctrl+C 信号

    while (!stop) {
        printf("Viva la chine!\n");
        sleep(1); // 延迟1秒
    }

    printf("\nAu revoir!\n");
    return 0;
}
```

当用户按下 `Ctrl+C` 时，信号处理函数将 `stop` 设置为 1，这会使循环结束。

随后，程序打印 `Au revoir!` 并正常终止。

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t stop;

void handle_sigint(int sig) {
    stop = 1;
}

int main() {
    signal(SIGINT, handle_sigint); // 捕捉 Ctrl+C 信号

    while (!stop) {
        printf("Viva la chine!\n");
        sleep(1); // 延迟1秒
    }

    printf("\nAu revoir!\n");
    return 0;
}

```

```
$ gcc loop_string.c -o loop_string
$ ./loop_string
```

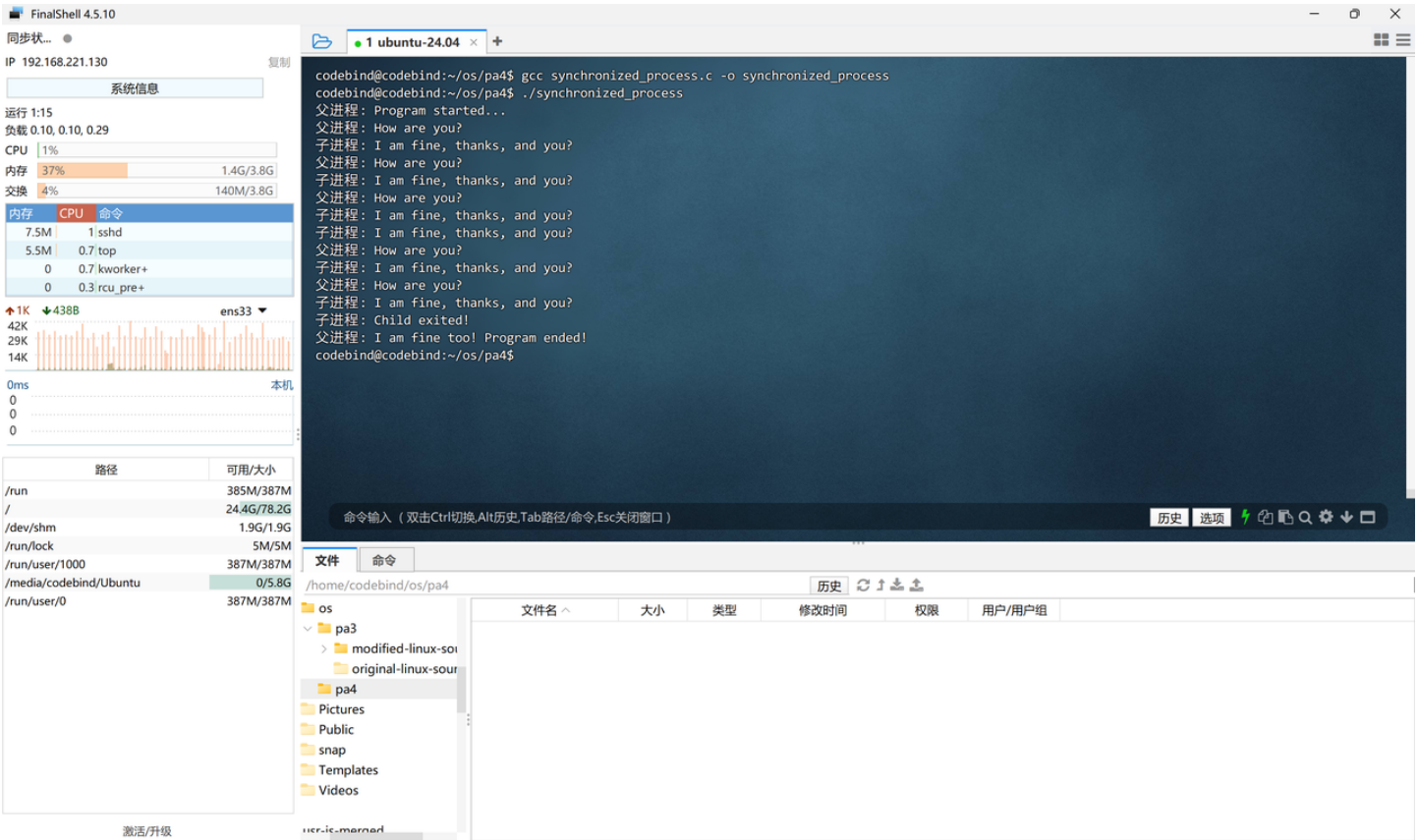


3.3 父子进程同步

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

volatile sig_atomic_t stop = 0;

void handle_sigusr1(int sig) {
    stop = 1;
}

int main() {
    signal(SIGUSR1, handle_sigusr1); // 捕捉软中断信号

    pid_t pid = fork(); // 创建子进程

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        // 子进程
        while (!stop) {
            printf("I am fine, thanks, and you?\n");
```

```c
            sleep(1);
        }
        printf("Child exited!\n");
    } else {
        // 父进程
        printf("Program started...\n");
        for (int i = 0; i < 5; i++) {
            printf("How are you?\n");
            sleep(1);
        }
        kill(pid, SIGUSR1); // 给子进程发送软中断信号
        wait(NULL); // 等待子进程终止
        printf("I am fine too! Program ended!\n");
    }

    return 0;
}
```

```
$ gcc synchronized_process.c -o synchronized_process
$ ./synchronized_process
```



3.4 基于无名管道的父子进程通信

📚 `pipe(fd)` 用于创建一个无名管道，`fd` 是一个大小为2的整数数组，其中：

- `fd[0]` 表示管道的读端。
- `fd[1]` 表示管道的写端。

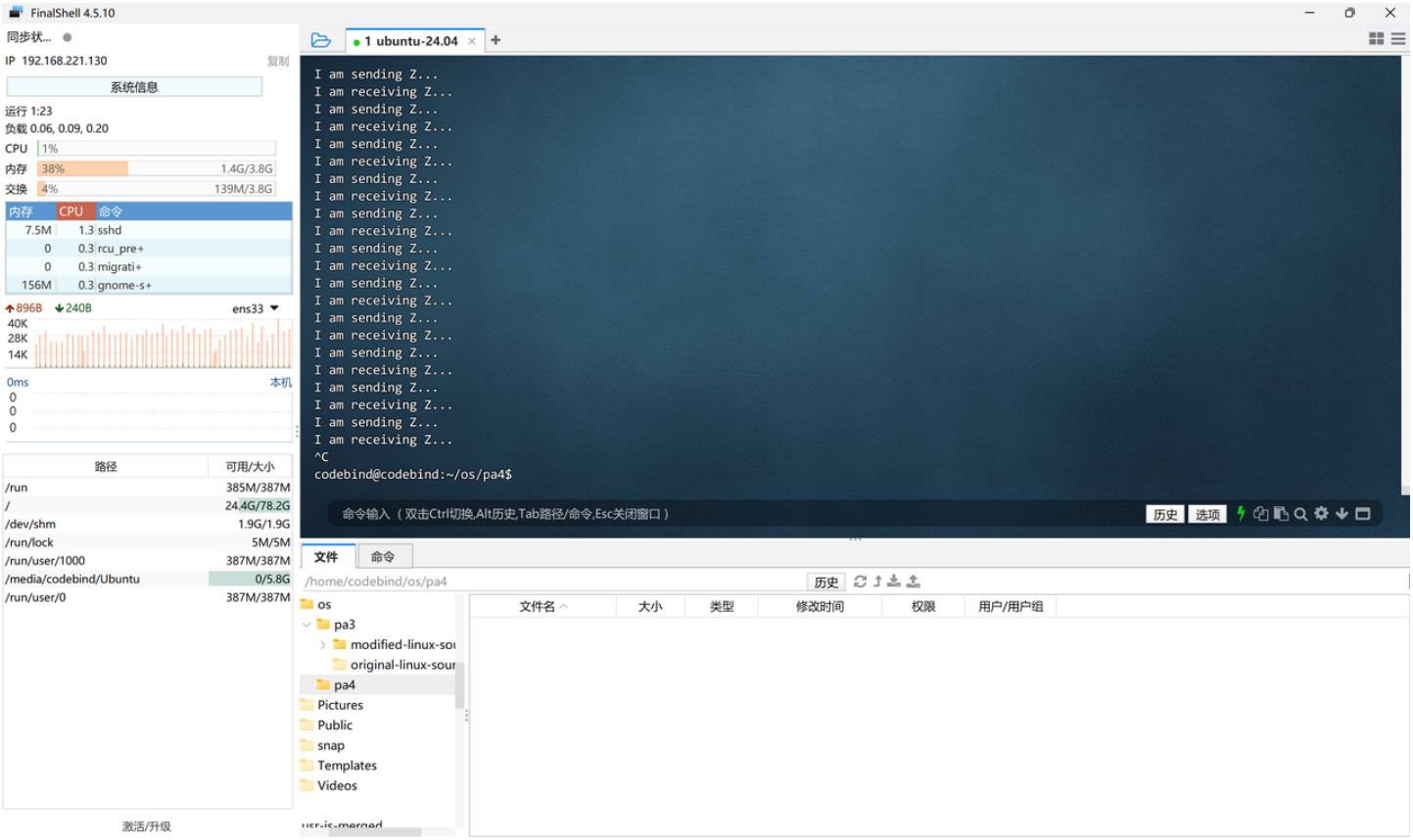
程序的整体流程：

1. **父进程** 先创建一个无名管道和一个子进程。
2. **父进程** 通过管道写入字母 `A` 到 `Z`，每个字母写 60 次。
3. **子进程** 从管道中读取数据并显示收到的字母。
4. **父进程** 完成所有字母的写入后，发送 `SIGUSR1` 信号给子进程，通知其停止工作。
5. **子进程** 接收到信号后，停止读取管道数据并终止，输出 “Child exited!”。
6. **父进程** 等待子进程结束后，输出 “Program ended!” 并终止。

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/wait.h>

volatile sig_atomic_t stop = 0;

void handle_sigusr1(int sig) {
    stop = 1;
}

int main() {
    int fd[2];
    char ch;
    pipe(fd); // 创建无名管道

    signal(SIGUSR1, handle_sigusr1); // 捕捉软中断信号

    pid_t pid = fork(); // 创建子进程

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        // 子进程：读取管道数据
        close(fd[1]); // 关闭写端
        while (!stop) {
```

```c
            if (read(fd[0], &ch, 1) > 0) {
                printf("I am receiving %c...\n", ch);
            }
        }
        printf("Child exited!\n");
        close(fd[0]); // 关闭读端
    } else {
        // 父进程：写入管道数据
        close(fd[0]); // 关闭读端
        char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        for (int i = 0; i < 26; i++) {
            for (int j = 0; j < 60; j++) {
                write(fd[1], &letters[i], 1);
                printf("I am sending %c...\n", letters[i]);
                usleep(10000); // 延迟10ms
            }
        }
        kill(pid, SIGUSR1); // 给子进程发送软中断信号
        wait(NULL); // 等待子进程终止
        printf("Program ended!\n");
        close(fd[1]); // 关闭写端
    }

    return 0;
}

```

```
$ gcc pipe_communication.c -o pipe_communication
$ ./pipe_communication
```



4. 疑难解惑与经验教训

无

5. 结论与体会

通过本次实验，我加深了对 Linux 进程控制机制的理解，学会了如何在父子进程间进行同步、通信以及如何通过信号控制进程的行为。使用 `fork()` 创建进程、`pipe()` 实现管道通信、以及捕捉和处理信号等操作让我更好地掌握了 C 语言在操作系统级别的编程技巧。实验中遇到的难点主要集中在父子进程间的同步和通信方面，经过调试，成功实现了进程之间的协作。此次实验不仅巩固了进程管理的理论知识，也提升了我解决实际编程问题的能力。