lab7

编程题

1.分别编写基于UNIX System V IPC的管道、共享内存、信号量和消息队列的Linux应用程序，实现进程间的数据交换。

管道

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
  int pipefd[2];
  // pipe syscall creates a pipe with two ends
  // pipefd[0] is the read end
  // pipefd[1] is the write end
  // ref: https://man7.org/linux/man-pages/man2/pipe.2.html
  if (pipe(pipefd) == -1) {
    perror("failed to create pipe");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    // child process reads from the pipe
    close(pipefd[1]); // close the write end
    // read a byte at a time
    char buf;
    while (read(pipefd[0], &buf, 1) > 0) {
      printf("%s", &buf);
    }
    close(pipefd[0]); // close the read end
  } else {
    // parent process writes to the pipe
    close(pipefd[0]); // close the read end
    // parent writes
    char* msg = "hello from pipe\n";
    write(pipefd[1], msg, strlen(msg)); // omitting error handling
    close(pipefd[1]); // close the write end
  }

  return EXIT_SUCCESS;
}
```

共享内存

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>

int main(void) {
  // create a new anonymous shared memory segment of page size, with a
permission of 0600
  // ref: https://man7.org/linux/man-pages/man2/shmget.2.html
  int shmid = shmget(IPC_PRIVATE, sysconf(_SC_PAGESIZE), IPC_CREAT | 0600);
  if (shmid == -1) {
    perror("failed to create shared memory");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    // attach the shared memory into child process's address space
    char* shm = shmat(shmid, NULL, 0);
    while (!shm[0]) {
      // wait until the parent signals that the data is ready
      // WARNING: this is not the correct way to synchronize processes
      // on SMP systems due to memory orders, but this implementation
      // is chosen here specifically for ease of understanding
    }
    printf("%s", shm + 1);
  } else {
    // attach the shared memory into parent process's address space
    char* shm = shmat(shmid, NULL, 0);
    // copy message into shared memory
    strcpy(shm + 1, "hello from shared memory\n");
    // signal that the data is ready
    shm[0] = 1;
  }

  return EXIT_SUCCESS;
}
```

信号量

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <sys/sem.h>

int main(void) {
  // create a new anonymous semaphore set, with permission 0600
  // ref: https://man7.org/linux/man-pages/man2/semget.2.html
  int semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
  if (semid == -1) {
    perror("failed to create semaphore");
    exit(EXIT_FAILURE);
  }

  struct sembuf sops[1];
  sops[0].sem_num = 0; // operate on semaphore 0
  sops[0].sem_op  = 1; // increase the semaphore's value by 1
  sops[0].sem_flg = 0;
  if (semop(semid, sops, 1) == -1) {
    perror("failed to increase semaphore");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    printf("hello from child, waiting for parent to release semaphore\n");
    struct sembuf sops[1];
    sops[0].sem_num = 0; // operate on semaphore 0
    sops[0].sem_op  = 0; // wait for the semaphore to become 0
    sops[0].sem_flg = 0;
    if (semop(semid, sops, 1) == -1) {
      perror("failed to wait on semaphore");
      exit(EXIT_FAILURE);
    }
    printf("hello from semaphore\n");
  } else {
    printf("hello from parent, waiting three seconds before release
semaphore\n");
    // sleep for three second
    sleep(3);
    struct sembuf sops[1];
    sops[0].sem_num = 0; // operate on semaphore 0
    sops[0].sem_op  = -1; // decrease the semaphore's value by 1
    sops[0].sem_flg = 0;
    if (semop(semid, sops, 1) == -1) {
      perror("failed to decrease semaphore");
      exit(EXIT_FAILURE);
    }
  }

  return EXIT_SUCCESS;
}
```

消息队列

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>

struct msgbuf {
  long mtype;
  char mtext[1];
};

int main(void) {
  // create a new anonymous message queue, with a permission of 0600
  // ref: https://man7.org/linux/man-pages/man2/msgget.2.html
  int msgid = msgget(IPC_PRIVATE, IPC_CREAT | 0600);
  if (msgid == -1) {
    perror("failed to create message queue");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid == -1) {
    perror("failed to fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    // child process receives message
    struct msgbuf buf;
    while (msgrcv(msgid, &buf, sizeof(buf.mtext), 1, 0) != -1) {
      printf("%c", buf.mtext[0]);
    }
  } else {
    // parent process sends message
    char* msg = "hello from message queue\n";
    struct msgbuf buf;
    buf.mtype = 1;
    for (int i = 0; i < strlen(msg); i ++) {
      buf.mtext[0] = msg[i];
      msgsnd(msgid, &buf, sizeof(buf.mtext), 0);
    }
    struct msqid_ds info;
    while (msgctl(msgid, IPC_STAT, &info), info.msg_qnum > 0) {
      // wait for the message queue to be fully consumed
    }
    // close message queue
    msgctl(msgid, IPC_RMID, NULL);
  }

  return EXIT_SUCCESS;
}
```

2.分别编写基于UNIX的signal机制的Linux应用程序，实现进程间异步通知。

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

static void sighandler(int sig) {
  printf("received signal %d, exiting\n", sig);    // 打印接收到的信号值
  exit(EXIT_SUCCESS);    // 退出程序
}

int main(void) {
  struct sigaction sa;    // 信号处理配置的结构体

  sa.sa_handler = sighandler;    // 设置信号处理函数
  sa.sa_flags = 0;    // 信号处理标志，无特殊标志
  sigemptyset(&sa.sa_mask);    // 初始化信号屏蔽字为空

  // 注册信号处理函数 sighandler 为 SIGUSR1 的信号处理器
  if (sigaction(SIGUSR1, &sa, NULL) != 0) {
    perror("failed to register signal handler");    // 注册失败时打印错误信息
    exit(EXIT_FAILURE);    // 退出程序
  }

  int pid = fork();    // 创建子进程

  if (pid == -1) {
    perror("failed to fork");    // 创建子进程失败时打印错误信息
    exit(EXIT_FAILURE);    // 退出程序
  }

  if (pid == 0)
  {
    while (1) {
      // 循环等待信号
    }
  }
  else {
    // 向子进程发送 SIGUSR1 信号
    kill(pid, SIGUSR1);
  }

  return EXIT_SUCCESS;
}
```

这个程序的思路如下：

1. 定义一个静态函数 `sighandler`，它用于处理信号。当接收到信号时，它打印信号值并退出程序。
2. 在 `main` 函数中，声明一个 `struct sigaction` 结构体变量 `sa`，用于配置信号处理。
3. 设置 `sa` 的成员 `sa_handler` 为上面定义的 `sighandler` 函数，用于处理信号。
4. 将 `sa` 的成员 `sa_flags` 设置为0，表示没有特殊的标志位。
5. 使用 `sigemptyset` 函数将 `sa` 的成员 `sa_mask` 初始化为空的信号屏蔽字。

6. 调用 `sigaction` 函数将 `SIGUSR1` 信号的处理器注册为 `sighandler` 函数。如果注册失败，打印错误信息并退出程序。
7. 使用 `fork` 函数创建一个子进程。如果创建失败，打印错误信息并退出程序。
8. 在父进程中，判断当前进程是否为子进程。如果是子进程，进入一个无限循环，等待信号的到来。
9. 在父进程中，向子进程发送 `SIGUSR1` 信号，使用 `kill` 函数。
10. 程序正常退出，返回 `EXIT_SUCCESS`。

3.参考rCore Tutorial 中的shell应用程序，在Linux环境下，编写一个简单的shell应用程序，通过管道相关的系统调用，能够支持管道功能。

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

int parse(char* line, char** argv) { //用于解析输入的命令行，将其分割为标记并存储在一个字符指针数组中
  size_t len;
  // 从标准输入读取一行
  if (getline(&line, &len, stdin) == -1)
    return -1;
  // 移除行末尾的换行符
  line[strlen(line) - 1] = '\0';
  // 将行分割为标记
  int i = 0;
  char* token = strtok(line, " ");
  while (token != NULL) {
    argv[i] = token;
    token = strtok(NULL, " ");
    i++;
  }
  return 0;
}

int concat(char** argv1, char** argv2) { //执行两个命令的串联操作
    // 创建管道
    int pipefd[2];
    if (pipe(pipefd) == -1)
      return -1;

    // 执行第一个命令
    int pid1 = fork();
    if (pid1 == -1)
      return -1;
    if (pid1 == 0) {
      dup2(pipefd[1], STDOUT_FILENO);  // 重定向子进程1的标准输出到管道写入端
      close(pipefd[0]);  // 关闭管道读取端
      close(pipefd[1]);  // 关闭管道写入端
      execvp(argv1[0], argv1);  // 执行第一个命令
    } //将标准输出重定向到管道的写入端，使得输出通过管道传递给下一个命令
```

```
    // 执行第二个命令
    int pid2 = fork();
    if (pid2 == -1)
      return -1;
    if (pid2 == 0) {
      dup2(pipefd[0], STDIN_FILENO);  // 重定向子进程2的标准输入到管道读取端
      close(pipefd[0]);  // 关闭管道读取端
      close(pipefd[1]);  // 关闭管道写入端
      execvp(argv2[0], argv2);  // 执行第二个命令
    } //将标准输入重定向到管道的读取端，以便从管道中读取第一个命令的输出作为输入

    close(pipefd[0]);  // 关闭管道读取端
    close(pipefd[1]);  // 关闭管道写入端
    wait(&pid1); // 等待子进程退出
    wait(&pid2);
    return 0;
}

int main(void) {
  printf("[command 1]$ ");
  char* line1 = NULL;
  char* argv1[16] = {NULL};
  if (parse(line1, argv1) == -1) { //解析命令行参数
    exit(EXIT_FAILURE);
  }
  printf("[command 2]$ ");
  char* line2 = NULL;
  char* argv2[16] = {NULL};
  if (parse(line2, argv2) == -1) {
    exit(EXIT_FAILURE);
  }
  concat(argv1, argv2);  // 执行两个命令的串联
  free(line1);  // 释放内存
  free(line2);  // 释放内存
}
```

这段代码实现了两个命令的串联执行。代码的思路如下：

1. 定义了两个辅助函数 `parse` 和 `concat`，以及主函数 `main`。
2. `parse` 函数用于解析输入的命令行，将其分割为标记并存储在一个字符指针数组中。
3. `concat` 函数用于执行两个命令的串联操作。它首先创建一个管道，然后使用 `fork` 函数创建两个子进程。
4. 第一个子进程负责执行第一个命令，并将标准输出重定向到管道的写入端，使得输出通过管道传递给下一个命令。
5. 第二个子进程负责执行第二个命令，并将标准输入重定向到管道的读取端，以便从管道中读取第一个命令的输出作为输入。
6. 父进程在执行完 `fork` 后，关闭了管道的读取和写入端，并使用 `wait` 函数等待子进程的退出。
7. 主函数 `main` 提示用户输入第一个命令，并调用 `parse` 函数解析命令行参数，然后提示用户输入第二个命令，同样调用 `parse` 函数解析参数。
8. 最后，调用 `concat` 函数执行两个命令的串联操作，并释放动态分配的内存。

总体思路是通过创建管道和使用 `fork` 函数创建子进程，将第一个命令的输出通过管道传递给第二个命令的输入，实现两个命令的串联执行。

问答题

1.直接通信和间接通信的本质区别是什么？分别举一个例子。

本质区别是消息是否经过内核，如共享内存就是直接通信，消息队列则是间接通信。

2.请描述Linux中的无名管道机制的特征和适用场景。

无名管道用于创建在进程间传递的一个字节流，适合用于流式传递大量数据，但是进程需要自己处理消息间的分割。

3.请描述Linux中的消息队列机制的特征和适用场景。

消息队列用于在进程之间发送一个由type和data两部分组成的短消息，接收消息的进程可以通过type过滤自己感兴趣的消息，适用于大量进程之间传递短小而多种类的消息。

4.请描述Linux的bash shell中执行与一个程序时，用户敲击 Ctrl+C 后，会产生什么信号（signal），导致什么情况出现。

会产生SIGINT，如果该程序没有捕获该信号，它将会被杀死；若捕获了，通常会在处理完或是取消当前正在进行的操作后主动退出。

5.请描述Linux的bash shell中执行与一个程序时，用户敲击 Ctrl+Z 后，会产生什么信号（signal），导致什么情况出现。

会产生SIGTSTP，该进程将会暂停运行，将控制权重新转回shell。

6.请描述Linux的bash shell中执行 kill -9 2022 这个命令的含义是什么？导致什么情况出现。

向pid为2022的进程发送SIGKILL，该信号无法被捕获，该进程将会被强制杀死。

7.请指出一种跨计算机的主机间的进程间通信机制。

一个在过去较为常用的例子是Sun RPC。