***MANAGE.C***

#include "defs.h"

st\_memory \*shared\_memory;

st\_message \*message;

int memID;

int msgID;

void terminate(int signo){

**//SEND INTR signal to all running computes**

for (int i =0 ; i<20 ; i++){

if(shared\_memory->process[i].pid !=0 ) {

kill(shared\_memory->process[i].pid, SIGINT);

}}

sleep(5);

**//delloactes the shared memroy**

shmdt(shared\_memory);

shmctl(memID, IPC\_RMID, NULL);

msgctl(msgID, IPC\_RMID, NULL);

free(message);

exit(0);

}

**void initializeSharedMemory()**{

//clear bitmap

for (int i = 0 ; i < BITMAPSIZE; i++){

shared\_memory->bitmap[i] = 0;

}

//clear all processes

for (int i = 0; i < 20; i++){

shared\_memory->process[i].pid = 0;

shared\_memory->process[i].num\_notest = 0;

shared\_memory->process[i].num\_perfec = 0;

shared\_memory->process[i].num\_test = 0;

}

//clear perfect num array

for(int i = 0 ; i < 20; i++){

shared\_memory->perfectfound[i]=0;

}}

**int findIndex(){**

for (int i = 0; i<20; i++){

if(shared\_memory->process[i].pid == 0){

return i;}

} return -1;}

**int main(){**

int currentIndex;

**//connect shared memroy**

memID = shmget(MEMKEY, sizeof(st\_memory), IPC\_CREAT | IPC\_EXCL | 0666);

if(memID == -1){ perror("manage shmget error"); exit(1); } //error

shared\_memory = shmat(memID, NULL, 0);

initializeSharedMemory();

**// Alternative way to clear shared memory**

// memset(shared\_memory->bitmap, 0, sizeof(shared\_memory->bitmap));

**//signal handling**

struct sigaction sigact;

memset(&sigact, 0, sizeof(sigaction));

sigact.sa\_handler = terminate;

sigaction(SIGINT, &sigact, NULL);

sigaction(SIGQUIT, &sigact, NULL);

sigaction(SIGHUP, &sigact, NULL);

**//connect msg q**

msgID = msgget(MSGKEY, IPC\_CREAT | 0666);

if(msgID == -1) { perror("manage msgget error"); exit(1); }

message = malloc(sizeof(st\_message));

**//Get messages in loop**

while(1){

memset(message, '\0', sizeof(st\_message));

**//recieve message**

if((msgrcv(msgID, message, sizeof(message), 0, 0)) == -1) {perror("manage msgrcv error"); exit(1);}

long messagetype = message->msg\_type;

int perfectfoundindex = -1;

switch(messagetype){

case GET\_PROCESSID :

//find an avaliable row index

currentIndex = findIndex();

//attach to a shared index in memory

shared\_memory->process[currentIndex].pid = message->msg\_number;

break;

case FOUND\_PERFECT\_NUM :

//find index in perfect found

for(int i = 0 ; i < 20 ;i++)

if(shared\_memory->perfectfound[i] == 0){ perfectfoundindex = i; }}

if(perfectfoundindex == -1){ printf("perfect found array full");}

else{//puts the perfect num in

shared\_memory->perfectfound[perfectfoundindex] = message->msg\_number;}

break;

case KILL\_ALL:

terminate(0); break; } }

terminate(0);

}

***COMPUTE.C***

#define SETBIT( x, i ) ( x |= (1 << i) )

#define TESTBIT( x, i ) (( x & (1 << i)) != 0 )

#define CLRBIT( x, i ) ( x &= ~(1 << i) )

#define FLIPBIT( x, i ) ( x ^= (1 << i) )

st\_memory \*shared\_memory;

st\_message \*message;

pid\_t currentpid;

int memID;

int msgID;

int rowindex;

int main(int argc, char\*\* argv){

int N, testnum; //the numbers to be tested

rowindex = -1;

//Get the N to compute

if(argc < 2){ N = 3; }

else{ N = atoi(argv[1]);}

if(N>BITMAPSIZE\*32){ printf("Compute too large.\n"); return 1; }

if( N < 3 ){ N = 3; }

//connecting shared memory segment

//connecting to msg q with MSGKEY

currentpid = getpid();

**//Send the pid to manage**

message->msg\_type = GET\_PROCESSID;

message->msg\_number = currentpid;

if(msgsnd(msgID, message, sizeof(message), 0) != 0 ) {perror("compute pid send error"); return 1;}

sleep(1);

**//Find the index to work on**

for(int i = 0 ; i<20 ; i++){

if(shared\_memory->process[i].pid == currentpid){ rowindex = i; break;

}}

if(rowindex == -1){

printf("Processes are full");return 0;}

/**/computing Perfect num in loop**

int compute = 1;

int loop = 0;

for(testnum = N; compute != 0; testnum++){

**//end call report -k**

if(testnum == N && loop == 1){

compute = 0;

char \*command = "./report";

char \*arguments[] = { "./report", "-k", NULL, NULL };

execvp(command, arguments); }

else if(testnum == BITMAPSIZE\*32){

loop = 1; testnum = 2; }

//find the bit to operate on

int x = whichInt(testnum);

int i = whichBit(testnum);

**//TESTED? PERFECT? SEND MSG**

if(TESTBIT(shared\_memory->bitmap[x],i) == 0){

SETBIT(shared\_memory->bitmap[x],i);

shared\_memory->process[rowindex].num\_test++; //increase tested count

if(isPerfect(testnum) == 1){ shared\_memory->process[rowindex].num\_perfect++;

message->msg\_type = FOUND\_PERFECT\_NUM;

message->msg\_number = testnum;

if(msgsnd(msgID, message, sizeof(message), 0) != 0 ) {perror("compute perfect num send error"); return 1;}}}

else{ //TESTBIT is 1, already computed

shared\_memory->process[rowindex].num\_notest++; //increase not tested count}

} return 0; }

int isPerfect(int n){//exclude 0 and 1 from calculation

if(n == 0 || n == 1){return 0;}

int sum = 1;

for(int i=2; i\*i<=n; i++){

if(n%i == 0) {sum = sum + i + n/i; }}

if (sum == n && n !=1 ) return 1; //1 = perfect number

return 0; //0 = not perfect

}

int whichInt(int N) { return N/32; }

int whichBit(int N) { return N%32;}

***REPORT.C***

st\_memory \*shared\_memory;

st\_message \*message;

int memID;

int msgID;

int main(int argc, char\*\* argv){

//connect to a shared memory

//connect tp to msg q with MSGKEY

int totalperfectnum = 0;

int totalnumbertested = 0;

int individualperfectsum = 0;

int individualnumbertested = 0;

printf("pid\t#perfect\t#skipped\t\t#tested\n\n");

for(int i = 0; i < 20 ; i++){

//if the process is running

if(shared\_memory->process[i].pid != 0){

individualnumbertested = shared\_memory->process[i].num\_test;

individualperfectsum = shared\_memory->process[i].num\_perfect;

printf("%d\t%d\t\t%d\t\t%d\n\n",

shared\_memory->process[i].pid,

individualperfectsum,

shared\_memory->process[i].num\_notest,

individualnumbertested);

totalnumbertested += individualnumbertested;

totalperfectnum += individualperfectsum; } }

printf("The Perfect Numbers: \n");

for (int i=0; i<20; i++){

if(shared\_memory->perfectfound[i] != 0){

printf("%d ", shared\_memory->perfectfound[i]); } }

printf("Total Number Tested: %d\n", totalnumbertested);

printf("Total Perfect Found: %d\n\n", totalperfectnum);

if(argc >= 2){ **//report -k handling**

if((strcmp(argv[1], "-k")) == 0){

message->msg\_type = KILL\_ALL;

if(msgsnd(msgID, message, sizeof(message), 0) != 0 ) {perror("report KILL ALL err"); return 1;}}}}

***DEFS.h***

#define BITMAPSIZE 1048576 // 2^25/32 == 1,048,576

#define MEMKEY 26677

#define MSGKEY 46677

#define GET\_PROCESSID 1

#define FOUND\_PERFECT\_NUM 2

#define KILL\_ALL 3

typedef struct{

pid\_t pid; //pid

int num\_perfect; //num of perfect found

int num\_test; //num tested

int num\_notest; //num not tested

} st\_process;

typedef struct{

int bitmap[BITMAPSIZE];

int perfectfound[20];

st\_process process[20];

} st\_memory;

typedef struct{

long msg\_type;

int msg\_number;

} st\_message;

***UNIQIFY.C***

int main(int argc, char \*\*argv) {

int afd[2]; //fd for piping process A

int bfd[2]; //fd for piping process B

pipe(afd); //make pipe A

pipe(bfd); //make pipe B

pid\_t apid, bpid; //identifies the first and second fork children

FILE \*fp; //for parse stdin

FILE \*stream; //for surpress stdin

**apid = fork(); //first child**

// printf("forked");

if(apid == -1){ printf("fork error");}

else if(apid == 0){ //child = handle sort

//close irrelevant pipes

close(afd[1]); //close write on A

close(bfd[0]); //close read on B

//relevant pipes

dup2(afd[0], STDIN\_FILENO); //redirecting pipe A readint to stdin

dup2(bfd[1], STDOUT\_FILENO); //redirecting pipe B writing to stdout

close(afd[0]);

close(bfd[1]);

execl("/usr/bin/sort", "sort", (char \*) NULL);

}

**bpid = fork(); //second child**

if(bpid == -1){printf("fork error");}

else if(bpid == 0){ //child **== surpress**

char currentword[100];

char nextword[100];

int count = 1;

//irrelevant pipes

close(afd[0]);

close(afd[1]);

close(bfd[1]);

//relevant pipes

dup2(bfd[0], STDIN\_FILENO); **//redirect the input from stdin**

close(bfd[0]);

//printf("\n");

fgets(currentword, 100, stdin);

while(fgets(nextword, 100, stdin)!= NULL){

int length = strlen(currentword);

// multiple of first word

if(strcmp(currentword, nextword) == 0){

count++; }

else{

if(length > 5 && length <= 35){

printf("Word: %s", currentword);

printf("Multiplicity: %-5d \n\n", count);

strcpy(currentword, nextword);

count = 1; }

else if(length > 35){

printf("Word: %.\*s\n", 35, currentword);

printf("Multiplicity: %-5d \n\n", count);

strcpy(currentword, nextword);

count = 1;}

else{

strcpy(currentword, nextword);

count = 1; } } }

int length = strlen(currentword);

if(length > 5 && length <= 35){

printf("Word: %s", currentword);

printf("Multiplicity: %-5d \n\n", count);

}

else if(length > 35){

printf("Word: %.\*s\n", 35, currentword);

printf("Multiplicity: %-5d \n\n", count);

}

exit(0);

}

**else{ //parent handles parsing**

//close irrelevant pipes

close(bfd[0]);

close(bfd[1]);

close(afd[0]);

char c;

//open a stream for writing into

FILE \*parse\_stream = fdopen(afd[1], "w");

//get each character from stdin until EOF

while((c=fgetc(stdin)) != EOF){

c = tolower(c);

if(isalpha(c)!=0){

fputc(c, parse\_stream);

}

else if(c == ' '||c == '\n'||isalpha(c)==0){

fputc('\n', parse\_stream);

}

}

fclose(parse\_stream);

close(afd[1]);

while(wait(NULL)>0){}

} return 0;}

***MYAR.c***

#define BUFFER 1024

#define AR\_HEADER\_SIZE 60

typedef struct ar\_hdr Header;

int main(int argc, char \* argv[]) {

if(argc < 3) print\_synopsis();

exit(-1);

}

//key = qxtA

char key = argv[1][0];

//name of archive

char\* archiveName = argv[2];

//single file found

char\* file = argv[3];

//to check if archive exists

int fd;

//F\_OK = file exists

if(access(archiveName, F\_OK) != -1){

//Opens file with append or R/W

fd = open(archiveName, O\_APPEND | O\_RDWR, 0666);

}

//doesn't exist

else{

if(key == 'q' || key =='A') {

//create a file with permission 666

fd = open(archiveName, O\_RDWR | O\_CREAT, 0666);

write(fd, ARMAG, 8);

}

else{

///command x && t

printf("File doesn't exist");

exit(-1);

}

}

int x;

switch (key) {

case 'q': append(fd, argc, argv);

break;

case 'x': extract(fd, argv);

break;

case 't': printt(fd, argv);

break;

case 'A': appendAll(fd, argv);

break;

default:

print\_synopsis();

break;

}

//close file descriptor so it can be reused.

close(fd);

exit(0);

}

void **append**(int fd, int argc, char \*\*argv){

int numofFiles= argc - 3 ;

int currentfile = 3;

while(numofFiles>0){

//Append the fstat info into archive

struct ar\_hdr \*file\_header = malloc(sizeof(struct ar\_hdr));

struct stat \*file\_stat = malloc(sizeof(struct stat));

char\* filename = argv[currentfile++];

int temp\_fd = open(filename, O\_RDONLY);

char realname[strlen(filename)+1];

strcpy(realname, filename);

strcat(realname, "/");

if(temp\_fd == -1){

printf("file not found");

close(fd);

exit(-1);

}

fstat(temp\_fd, file\_stat);

snprintf(file\_header->ar\_name, 16, "%-16s", realname);

snprintf(file\_header->ar\_date, 12, "%-12d", (int)file\_stat->st\_mtime);

snprintf(file\_header->ar\_uid, 6, "%-6d", (int)file\_stat->st\_uid);

snprintf(file\_header->ar\_gid, 6, "%-6d", (int)file\_stat->st\_gid);

snprintf(file\_header->ar\_mode, 8, "%-8o", file\_stat->st\_mode);

snprintf(file\_header->ar\_size, 10, "%-10d", (int)file\_stat->st\_size);

snprintf(file\_header->ar\_fmag, SARMAG, "%-2s", ARFMAG);

//Write the header information into the file

write(fd, file\_header, AR\_HEADER\_SIZE);

//Content of the file

char tempbuf[(int)file\_stat->st\_blksize];

int k = atoi(file\_header->ar\_size);

while(k>0){

if(k>BUFFER){

read(temp\_fd, tempbuf, BUFFER);

write(fd, tempbuf, BUFFER);

k = k - BUFFER;

}

else{

read(temp\_fd, tempbuf, k);

write(fd, tempbuf, k);

k = 0;

}

}

//offset

int j=file\_stat->st\_size;

if((j%2)==1){

write(fd,"\n",1);

}

close(temp\_fd);

numofFiles--;

}

close(fd);

exit(0);

}

void **extract**(int fd, char \*\*argv){

struct ar\_hdr \* fileHeader = malloc(sizeof(struct ar\_hdr));

struct stat \*file\_stat = malloc(sizeof(struct stat));

int found = 0;

int filesize;

char\* archiveName = argv[2];

char\* file = argv[3];

char\* name;

time\_t time;

printf("File:%s", file);

lseek(fd, SARMAG, SEEK\_SET);

fstat(fd, file\_stat);

filesize = file\_stat->st\_size;

while(read(fd, fileHeader, AR\_HEADER\_SIZE)!=0){

int j;

for(j=16; j>=0; j--){

if(fileHeader->ar\_name[j] == '/'){

fileHeader->ar\_name[j] = '\0';

}

}

name = fileHeader->ar\_name;

filesize= (int)strtol(fileHeader->ar\_size,NULL,10);

if(strcmp(name,file)!=0){

if(filesize%2 ==1){

filesize++;

}

lseek(fd, filesize, SEEK\_CUR);

}

else{

time = strtol(fileHeader->ar\_date, NULL,10);

found = 1;

printf(" found\n");

break;

}

}

if(found ==0){

printf(" doesn't exist\n");

}

//read content

int a;

int createdFile = open(name, O\_RDWR | O\_CREAT, 0666);

char contentbuffer[file\_stat->st\_size];

int readcount = atoi(fileHeader->ar\_size);

int i;

while(i<readcount){

a = read(fd, contentbuffer, 1);

write(createdFile, contentbuffer, a);

i++;

}

**//modify time**

struct utimbuf t\_buffer;

t\_buffer.actime = time;

t\_buffer.modtime= time;

utime(file, &t\_buffer);

}

void **printt**(int fd, char \*\*argv){

struct ar\_hdr \* fileHeader = malloc(sizeof(struct ar\_hdr));

struct stat \*file\_stat = malloc(sizeof(struct stat));

int found = 0;

int filesize;

char\* archiveName = argv[2];

char\* file = argv[3];

char\* name;

//printf("%s%s", archiveName, file);

lseek(fd, SARMAG, SEEK\_SET);

fstat(fd, file\_stat);

filesize = file\_stat->st\_size;

while(read(fd, fileHeader, AR\_HEADER\_SIZE)!=0){

int j;

for(j=16; j>=0; j--){

if(fileHeader->ar\_name[j] == '/'){

printf("%s\n", fileHeader->ar\_name);

}

}

filesize= (int)strtol(fileHeader->ar\_size,NULL,10);

name = fileHeader->ar\_name;

if (filesize%2==1) {

filesize++;

}

lseek(fd, filesize, SEEK\_CUR);

}

printf("done t");

}

void **appendAll**(int fd, char \*\*argv){

DIR \*d;

struct dirent \*dir;

char\* newargv[10000];

newargv[0] = argv[0]; //./myar

newargv[1] = "q";

newargv[2] = argv[2]; ///The name of the new archive file

int newargc = 2;

char myarfile[9];

d = opendir(".");

if (d) {

while ((dir = readdir(d)) != NULL) {

if((dir->d\_type == DT\_REG) && (strcmp(dir->d\_name,argv[2])!=0)){

char\* filename = dir->d\_name;

struct stat \* file\_stat = malloc(sizeof(struct stat));

stat(filename, file\_stat);

time\_t timer;

int seconds;

time(&timer);

seconds = difftime(timer,(int)file\_stat->st\_mtime);

if(seconds < 36000){

newargv[++newargc] = filename;

} } }

append(fd, newargc+1, newargv);

closedir(d);

} }

***JohnSem.c***

main() {

        int sid;        /\* segment id of shared memory segment \*/

        int \*array;     /\* pointer to shared array, no storage yet\*/

        int j;          /\*loop counter \*/

        int semid;      /\* semaphore id \*/

        struct sembuf sb;       /\* semaphore buffer \*/

                        /\* create shared segment if necessary \*/

        if ((sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660))== -1) {

                perror("shmget");

                exit(1);

                }

                        /\* map it into our address space\*/

        if ((array=((int \*) shmat(sid,0,0)))== (int \*) -1) {

                perror("shmat");

                exit(2);

                }

                        /\* Now fill it up \*/

        for (j=0;j<=100;j++) array[j]=j;

                        /\* get semaphore id\*/

        if ((semid=semget(KEY,1 ,IPC\_CREAT |0660))== -1) {

                perror("semget");

                exit(1);

                }

        sb.sem\_op = 1;  /\* set up for a unlock operation\*/

        sb.sem\_num = 0;

        sb.sem\_flg = 0;

        if (semop(semid, &sb, 1) == -1) { /\* should not block \*/

                perror("sem unlock");

                exit(1);

                }}

***MarySem.c***

struct sembuf sb;       /\* semaphore buffer \*/

main() {

        int sid;        /\* segment id of shared memory segment \*/

        int \*array;     /\* pointer to shared array, no storage yet\*/

        int j;          /\*loop counter \*/

        int sum;        /\*running sum\*/

        int semid;      /\* semaphore id \*/

                        /\* create shared segment if necessary \*/

        if ((sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660))== -1) {

                perror("shmget");

                exit(1);

                }

                        /\* map it into our address space\*/

        if ((array=((int \*) shmat(sid,0,0)))== (int \*)-1) {

                perror("shmat");

                exit(2);

                }

                        /\* get semaphore id\*/

        if ((semid=semget(KEY,1 ,IPC\_CREAT |0660))== -1) {

                perror("semget");

                exit(1);

                }

        sb.sem\_op =-1;  /\* set up for a lock operation\*/

        sb.sem\_num =0;

        sb.sem\_flg =0;

        if (semop(semid, &sb, 1) == -1) { /\* will block if locked \*/

                perror("sem lock");

                exit(1);

                }

                        /\* Now add it up \*/

        sum=0;

        for (j=0;j<=100;j++) sum+=array[j];

        printf("Mary says arrary sun is %d\n",sum);

        /\* Unmap and deallocate the shared segment \*/

        if (shmdt( (char \*) array) == -1) {

                perror("shmdt");

                exit(3);

        }

        if (shmctl(sid,IPC\_RMID,0) == -1) {

                perror("shmctl");

                exit(3);

        } }

***Signals.c && Jumping***

void perfect(int);

sigjmp\_buf jmpenv; /\* environment saved by setjmp\*/

int n; /\* global variable indicating current test point \*/

int main() {

        int begin; /\* starting point for next search\*/

                /\* interrupt routines\*/

        void status();

        void query();

        sigset\_t mask;

        struct sigaction action;

        if (sigsetjmp(jmpenv)) {

                printf("Enter search starting point (0 to terminate): ");

                scanf("%d",&begin);

                if (begin==0) exit(0);

                sigprocmask(SIG\_UNBLOCK, &mask, NULL);

                }

        else begin=2;

        /\* Status Routine will handle timer and INTR \*/

        sigemptyset(&mask);

        sigaddset(&mask, SIGINT);

        sigaddset(&mask, SIGALRM);

        sigaddset(&mask, SIGQUIT);

        action.sa\_flags=0;

        action.sa\_mask=mask;

        action.sa\_handler=status;

        sigaction(SIGINT,&action,NULL);

        sigaction(SIGALRM,&action,NULL);

        action.sa\_handler=query;

        sigaction(SIGQUIT,&action,NULL);

/\* start alarm clock \*/

        alarm(20);

                perfect(begin);

                }

void perfect(start)

        int start;

{

        int i,sum;

        n=start;

while (1) {

        sum=1;

        for (i=2;i<n;i++)

                if (!(n%i)) sum+=i;

        if (sum==n) printf("%d is perfect\n",n);

        n++;

        }

}

void

status(signum)

        int signum;

{

        alarm(0); /\* shutoff alarm \*/

        if (signum == SIGINT) printf("Interrupt ");

        if (signum == SIGALRM) printf("Timer ");

        printf("processing %d\n",n);

        alarm(20);      /\*restart alarm\*/

}

void query() {siglongjmp(jmpenv,1);}

***Find all files in current dir***

int main(void) {

DIR \*d;

struct dirent \*dir;

d = opendir(".");

if (d) {

while ((dir = readdir(d)) != NULL) {

printf("%s\n", dir->d\_name);

}

closedir(d);

}

return(0);

}

***one a.c b.c < x | two > y***

pipe(pfd){

if(!fork()){

close(pfd[0]);

dup2(pfd[1], 1);

close(pfd[1]);

fd = open("x", O\_RDONLY);

dup2(fd, 0);

close(fd);

execl("one","one","a.c","b.c", (char\*)0);

}

if(!fork()){

close(pfd[1]);

dup2(pfd[0],0);

close(pfd[0]);

fd=open("y", O\_WRONLY);

dup2(fd, 1);

close(fd);

execl("two", "two", (char\*) 0);

}

close(pfd[0]);

close(pfd[1]);

wait(&stat);

wait(&stat);

***"grep Villanova < scores > out"***

int main(int argc, char \*\*argv)

{

int in, out;

char \*grep\_args[] = {"grep", "Villanova", NULL};

// open input and output files

in = open("scores", O\_RDONLY);

out = open("out", O\_WRONLY | O\_TRUNC | O\_CREAT, S\_IRUSR | S\_IRGRP | S\_IWGRP | S\_IWUSR);

// replace standard input with input file

dup2(in, 0);

// replace standard output with output file

dup2(out, 1);

// close unused file descriptors

close(in);

close(out);

// execute grep

execvp("grep", grep\_args);

}

***Sort reg file in dir***

int main() {

int fds[2];

pid\_t pid;

char cwd[1024];

char \* buffer;

struct dirent \* DIR;

struct DIR \* dirp;

FILE \*stream;

pipe(fds);

int statues;

pid = fork();

if (pid == 0) { //child

close(fds[1]); //close writing on child

dup2(fds[0], STDIN\_FILENO);

//changes read end of pipe to stdin

close(fds[0]); //closes read on pipe

execlp("sort", "sort", NULL);

} else { //parent

close(fds[0]); //close read end

dup2(fds[1], STDOUT\_FILENO);

//changes write end to stdout

stream = fdopen(STDOUT\_FILENO, "w");

//write stdout

getcwd(cwd, sizeof(cwd));

dirp = opendir(cwd);

while ((DIR = readdir(dirp)) != NULL) {

if (DIR -> d\_type == DT\_REG) {

fputs(DIR -> d\_name, stream);

fputs("\n", stream);

}

}

fflush(stream);

fclose(stream);

close(fds[1]);

waitpid(pid, NULL, 0);

}

return 0;

}

cat < y.txt | sort > x.txt

int main() {

int fds[2];

int fd;

int pd;

pid\_t pid;

pipe(fds);

pid = fork();

if(pid == 0){

close(fds[1]); //close write

fd = open("x.txt", O\_RDWR | O\_CREAT, 0666);

dup2(fds[0], 0);

close(fds[0]);

dup2(fd, 1);

execlp("sort","sort", NULL);

}else {

close(fds[0]);

pd = open("y.txt", O\_RDONLY);

dup2(pd, 0);

dup2(fds[1], 1);

close(fds[1]);

execlp("cat", "cat", NULL);

waitpid(pid, 0, NULL);

}

}