Types of Child Process:

1. Zoombie Process

When parent has to read or completely read the child process which it is not completes and excute parent process.

RESOLVE BY

Using Waite(NULL) command which waits for child process to complete its execution and then read its child process and remove it from process table

1. Orphan Process:

When child is running before its completion parent process is terminated

RESOLVE BY

Kernel by itself will make one parent process and assign that child and then it completes its process then it will kill it

1

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(){

printf("Before forking \n");

printf("Creating child proccess \n");

int i = fork (); //Here child process will be created

//and execute after the fork process

printf("%d\n",i);

if (i==0){

printf("Child Process=%d\n",getpid());

}

else{

printf("Parent Process=%d\n",getpid());

}

printf("After Fork Process \n ");

return 0;

}

2

#include<stdio.h>

//#include<sys/types.h>

#include<unistd.h>

void parent\_process(int cvar);

void child\_process(int pvar);

int y=10;

int main(){

int x=0;

printf("Before Forking \nCreating Child Process\n");

int i=fork();

if(i==0){

child\_process(x);

}

else{

parent\_process(x);

}

printf("After Forking\n");

return 0;

}

void child\_process(int a){

y+=2;

a=3;

printf("The value of child process variable = %d\n",a);

printf("In child process = %d\n",y);

}

void parent\_process(int b){

b=2;

y+=5;

printf("The value of parent process variable = %d\n",b);

printf("In process process = %d\n",y);

}

3

#include<stdio.h>

#include<unistd.h>

int main(){

printf("node:1 , pid = %d, ppid =%d\n",getpid(),getppid());

int i = fork();

if(i==0){

printf("node:3 , pid = %d, ppid =%d\n",getpid(),getppid());

}

else{

i = fork();

if(i==0){

printf("node:2 , pid = %d, ppid =%d\n",getpid(),getppid());

i=fork();

if(i==0){

printf("node:6 , pid = %d, ppid =%d\n",getpid(),getppid());

}

else{

i=fork();

if(i==0){

printf("node:5 , pid = %d, ppid =%d\n",getpid(),getppid());

}

}

}

}

return 0;

}

4

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

void fork7(){

if(fork()==0){

//Child

printf("Terminating Child Process\n");

printf("PID = %d\n",getpid());

exit(0);

}

else{

printf("Running Parent Process\n");

printf("PID =%d\n",getpid());

while(1){

//Inifinite Loop

}}

}

int main(){

printf("Main Function\n");

fork7();

return 0;

}

5

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(){

int pid = fork();

if(pid>0){

//parent process

printf("Parent Process\nPID is =%d\n",getpid());

}

if(pid==0){

printf("Child Process\nPID is =%d\n",getpid());

printf("Parent \nPID is =%d\n",getpid());

}

wait(NULL);

return 0;

}

6

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(){

execl("/bin/ls","dir",(char\*)0);

//We can only reach this code if execl returned with an error

perror("execl");

return 0;

}

7

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(){

int pid;

char cmd[256];

printf("Press e if you want to terminate\n");

while(1){

printf("cmd: ");

scanf("%s",cmd);

if(strcmp(cmd,"e")==0){ //Loop terminates if the user types e

exit(0);} //creates a new process. Parent gets the process ID. Child gets 0

if((pid==0)){ //Child process

wait(NULL);

}

else if(pid==0){ //child process

execlp(cmd,cmd,NULL);

//exec cannot return. If so do the following

fprintf(stderr,"Cannot create a new proccess\n");

exit(2);

}}}

8

#include <unistd.h>

#include<stdlib.h>

int main(void) {

execl("Output5","",NULL);

//system("Desktop/test1.sh 5");

return 0;

}

IPC and its Methagologies:  
 A program may have to feed another process for it to proceed. It is inherent in all the embedded systems.

In computing, it's common for one program or process to depend on the output or input of another process in order to proceed with its own tasks. This type of interaction between processes is known as inter-process communication (IPC).

IPC plays vital role in it. Folllowing are the commonly used mechanism.

1. Pipe
2. Named pipe
3. Messege Queue
4. Sheared Memory

Pipe:

Unidirectinol communication.

One send second reads like and vice versa, not read and send at one time.

Pipr only work one relative process parent child type process

Pipe() system calls

1. Returns two files discriptors it defines no which is read process and which is wrilte process.

File descriptors are commonly used in Unix-like operating systems, where many system resources are represented as files or streams, including network connections, pipes, and sockets.

file descriptors are used to manage and manipulate these resources, allowing programs to communicate with each other and with the operating system.

Hear first one process write the programe then other can read it .like it can see in down example by sleep command which waits till parent process complets its write command(send piped ends) then read process works(which is receiving end of pipe).

Example:

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include <sys/wait.h>

int main(void) {

int pipefd[2];

int pid;

char buffer[15];

pipe(pipefd);

pid = fork();

if(pid > 0) {

fflush(stdin);

printf("unamed\_pipe [INFO] Parent Process\n");

write(pipefd[1],"Hellow Mr.Linux",15);

}

else if(pid == 0) {

sleep(5);

fflush(stdin);

printf("unamed\_pipe [INFO] Child Process\n");

read(pipefd[0], buffer, sizeof(buffer));

write(1,buffer, sizeof(buffer));

printf("\n");

}

else {

printf("unamed\_pipe [ERROR] Error in creating child process\n");

}

if(pid > 0) wait(NULL);

return 0;

}

Named Pipe

When two different process which does not have (parent child relation) the communication is done by named pipe.

For this we have to create common file which is accessible by both the process.

After completion of communication that file can be deleted. That types of files referred as FIFO in linux.

System call mkfifo can be used to create a FIFO.

FIFO is c code file which has two processes.

1. fifo\_write.c (it is write program)
2. fifo-read.c (it is the read program)

hear write program always run first if in any case we (user) forced to run read program first so it will wait till the write program runs here comes the auto synchronization.

Char by char file value send in it

General Knowledge:

it. There are basically three kinds of users available in Linux and three kinds of permissions

associated with a file. Next question would arise in minds that can the permissions be change?

Yes, it can be altered. ‘chmod’ is the command meant for it.

Fifo\_write.c

#include<stdio.h>

#include<sys/stat.h>

#include<sys/types.h>

#include<fcntl.h>

#include<unistd.h>

int main(void) {

int fd, retval;

char buffer[] = "this is the operating system last lab before mid... mid paper will be on Saturday 30th October,2021";

fflush(stdin);

retval = mkfifo("/tmp/myfifo",0666);

fd = open("/tmp/myfifo",O\_WRONLY);

write(fd,buffer,sizeof(buffer));

close(fd);

return 0;

}

Fifo\_read.c

#include<stdio.h>

#include<sys/stat.h>

#include<sys/types.h>

#include<fcntl.h>

#include<unistd.h>

int main(void) {

int fd, retval;

char buffer[512] = {0};

fd = open("/tmp/myfifo",O\_RDONLY);

retval = read(fd, buffer, sizeof(buffer));

fflush(stdin);

write(1, buffer, sizeof(buffer));

printf("\n");

close(fd);

return 0;

}

Message Queue

Two or more processes can exchange information via access to a common system message queue.

Messge sends onto a queue by sending process which can be read by any other process.

Each message is given an identification or type so that processes can select the appropriate message.

Process must share a common key in order to gain access to the queue in the first place.

Message queues provide an asynchronous way of communication possible, meaning that the

sender and receiver of the message need not interact with the message queue at the same time.

Working:

A task which has to send the message can put message in the queue and other tasks. A

message queue is a buffer-like object which can receive messages from ISRs (Interrupt Service

Routine), tasks and the same can be transferred to other recipients. In short, it is like a pipeline.

Components

Message queue basically composed of few components. A message queue should

have a start and it should have an end as well. Starting point of a queue is referred as head of

the queue and terminating point is called tail of the queue. Size of the queue has to be decided

by the programmer while writing the code. And a queue cannot be read if it is empty.

Meanwhile, a queue cannot be written into if it is already full. And a queue can have some

empty elements as well.

System Calls.

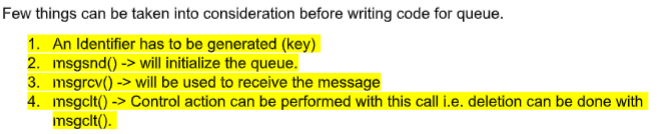
1. Creation/Deletion of Queue.
2. Sending /receiving of messege

2 different file

1. Sender
2. Reciver

Supports Automatic Syscncronization.

Memey is free after working



Message\_send.c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/types.h>

#include<sys/msg.h>

struct msgbuf {

long mtype;

char msgtxt[200];

};

int main(void) {

struct msgbuf msg;

int msgid;

key\_t key;

if((key = ftok("message\_send.c",'b')) == -1 ) {

perror("key");

exit(1);

}

if((msgid=msgget(key,0644|IPC\_CREAT)) == -1 ) {

perror("key");

exit(1);

}

printf("message\_send [INFO] The message ID is: %d\n", msgid);

printf("message\_send [PROMPT] Enter a text: ");

msg.mtype = 1;

while(fgets(msg.msgtxt,25, stdin)) {

if(msgsnd(msgid,&msg,sizeof(msg),0) == -1) {

perror("msgsnd");

exit(1);

}

}

if(msgctl(msgid,IPC\_RMID,NULL) == -1) {

perror("msgctl");

exit(1);

}

return 0;

}

Message\_recive.c

#include<stdio.h>

#include<stdlib.h>

#include<sys/ipc.h>

#include<sys/types.h>

#include<sys/msg.h>

struct msgbuf {

long mtype;

char msgtxt[200];

};

int main(void) {

struct msgbuf msg;

int msgid;

key\_t key;

if((key = ftok("message\_send.c",'b')) == -1 ) {

perror("key");

exit(1);

}

if((msgid=msgget(key,0644)) == -1 ) {

perror("msgid");

exit(1);

}

for(;;) {

if(msgrcv(msgid,&msg,sizeof(msg),1,0) == -1) {

perror("msgrcv");

exit(1);

}

printf("message\_receive [INFO] Message: %s\n",msg.msgtxt);

}

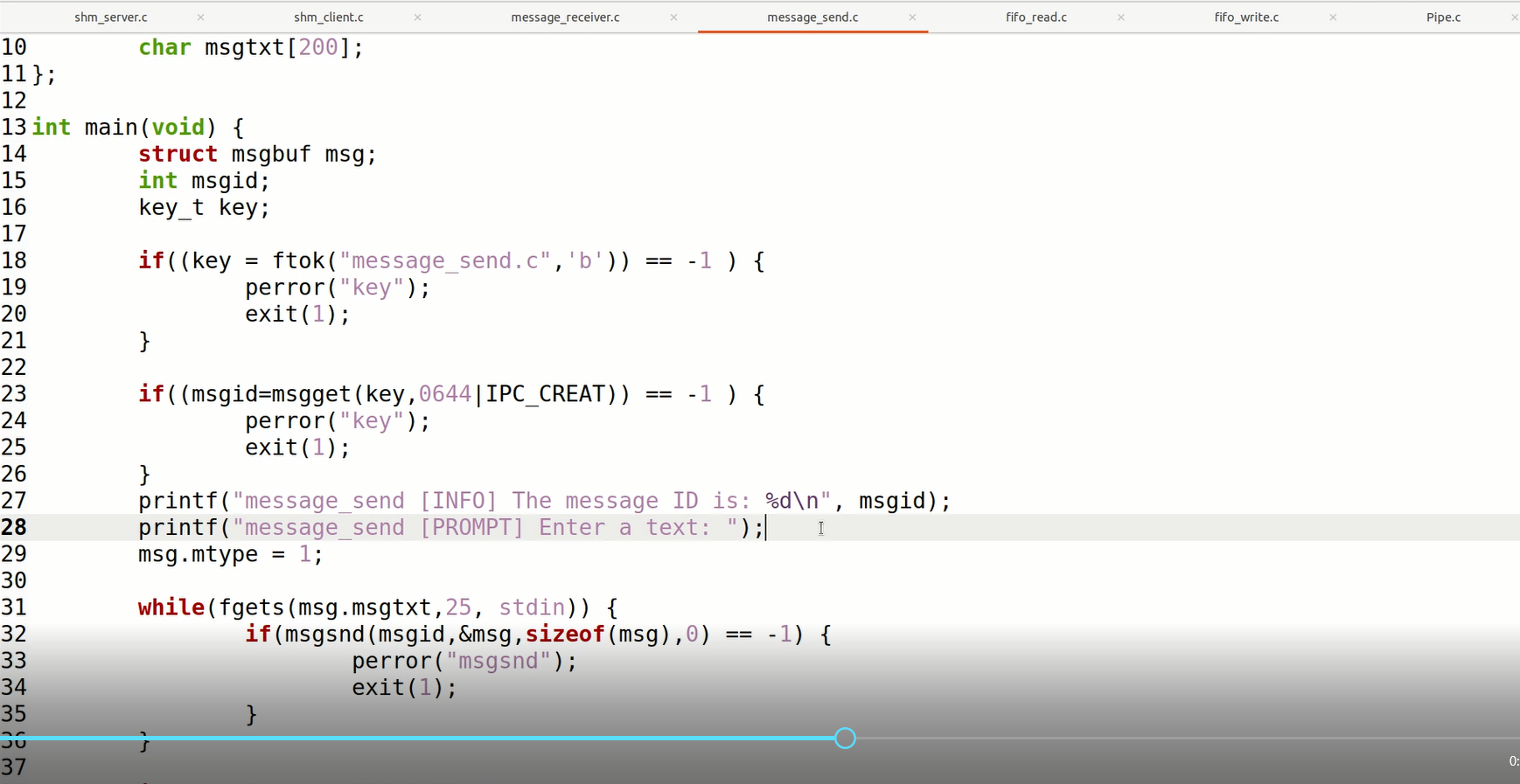
return 0;

}

Receiving end



Sending end



Sheared Memory

1. Initialization
2. Attached ( shmctl() )
3. Detached ( shmdt() )

**For Server**

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call shmget().
2. Attach this shared memory to the server’s address space with system call shmat().
3. Initialize the shared memory, if necessary.
4. Do something and wait for all clients’ completion.
5. Detach the shared memory with system call shmdt().
6. Remove the shared memory with system call shmclt().

**For Client**

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client’s address space
3. Use the memory
4. Detach all shared memory segments, if necessary
5. Exit

Server

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#include<stdlib.h>

#define MAXSIZE 27

int main(void) {

char c;

int shmid;

key\_t key;

char \*shm, \*s;

key = 2211;

if((shmid = shmget(key, MAXSIZE, IPC\_CREAT | 0666)) < 0)

exit(0);

if((shm = shmat(shmid, NULL, 0)) == (char\*) -1)

exit(0);

s = shm;

for(c = 'a'; c <= 'z'; c++)

\*s++ = c;

while(\*shm != '\0')

sleep(1);

exit(0);

}

Client

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#include<stdlib.h>

#define MAXSIZE 27

void die(char \*str) {

perror(str);

exit(1);

}

int main(void) {

int shmid;

key\_t key;

char \*shm, \*s;

key = 2211;

if((shmid = shmget(key, MAXSIZE, IPC\_CREAT | 0666)) < 0)

die("shmget");

if((shm = shmat(shmid, NULL, 0)) == (char\*) -1)

die("shmat");

for(s = shm; \*s != '\0'; s++)

putchar(\*s);

\*shm = '\0';

printf("\n");

exit(0);

}