

CMPS 405: OPERATING SYSTEMS

Process related system calls
fork, getpid, getppid, wait, waitpid, execlp,
vfork, _exit, sleep

This material is based on the operating system books by Silberschatz and Stallings

Contents:

❖ Process related system calls

- fork
- getpid
- getppid
- wait
- sleep
- exec
- exit
- vfork
- pipe
- read
- write
- close
- dup
- dup2

Recall that system calls

- ❖ Allow user-level processes to request services of the operating system.
- ❖ Generally available as assembly-language instructions.
- ❖ Some Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++).
 - high-level languages support system calls by hidden run-time support system (the set of functions built into libraries with a compiler) provided by the operating system.
- ❖ System programs can be thought of as bundles of useful system calls. They provide basic functionality to users and so users do not need to write their own programs to solve common problems.

Process Creation

- ❖ Parent process create children processes, which, in turn create other processes, forming a tree of processes.
 - **Resource sharing:** Parent and children might share all resources, subset of parent's resources, or no resources.
 - **Execution:** Parent and children execute concurrently or can have the parent to wait until children terminate.
 - **Address space:** Child duplicate of parent or a child has a program loaded into it.
- ❖ UNIX/Linux examples: **fork** system call creates new process while an **exec** system call used after a fork to replace the process' memory space with a new program.

Unix Example

In UNIX the **init** program is the root of all of your processes.

- ❖ UNIX examples: **fork** system call creates new process which is a duplicate copy of the parent.
 - The result of successful **fork()** is two processes: the original process called the parent and a new process called the child.
 - **fork()** returns an integer value to each of these processes as follows:
 - a **positive** integer returned to the parent process which represents the child's ID.
 - **Zero** returned to the child process
 - If **fork()** fails it returns a **negative** integer returned to the original process (in this case there is only one process no new process was created)

Current Process Information

- ❖ To get process identification numbers:
 - `pid_t getpid(void);`
 - gets the current process id
 - `pid_t getppid(void);`
 - gets the parent's process id of the current process
- ❖ `pid_t` is defined in the `unistd.h` and `sys/types.h` header files.
The default max pid is 32,768.

Typically fork() Calling Sequence

```
if ( ( pid = fork () ) < 0 )  
  
    printf ("error forking\n");  
  
else if ( pid == 0 ) {  
  
    /* execute child stuff here */  
  
} else {  
  
    /* place parent stuff here */  
}
```

Unix Example

- ❖ **execve** system call used after a **fork** to replace the process' memory space with a new program.
 - Which means that the process having exec system call does nothing except executing the program specified by exec.
- ❖ **The parameters** passes to **execve** are:
 1. the path of the program to execute ended with the program name as well,
 2. the second parameter is the program name,
 3. the third until the parameter before last are the arguments you need to pass to the program you are executing,
 4. and finally the last argument is **NULL**.

wait() System Call

- ❖ Usage:

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *status);

Header files to
include

Prototype

- ❖ Function: temporary suspend the process. Often, called by a parent process.
- ❖ Parameters: one parameter a pointer to an integer. Useful when wait returns. Or NULL, the parameter is ignored.
- ❖ Returned value: usually is the process id of the exiting child. If the parent has no children -1 is returned.

wait

- ❖ Defined in the `sys/wait.h` and `sys/types.h` header files
- ❖ `pid_t wait(int *status);`
 - returns the PID of the child, returns -1 if an error occurred, returns 0 if no child was available.
 - if the value of `status` is not NULL then `status` points to the location of the status information.
- ❖ function used to block until a process terminates
- ❖ `wait()` waits until a child has exited
- ❖ in `wait()` if a child has already exited this function immediately returns.

waitpid

❖ `pid_t waitpid(pid_t pid, int *status, int options);`

- `waitpid()` waits until a child specified by `pid` returns.
- The option `WNOHANG` is the most commonly used returning 0 until the child exits.
- `WIFEXITED(status)` : macro that checks if the returned status is set.
- `WEXITSTATUS(status)`: macro that extract the returned status to an integer value.

Process Termination

- ❖ Process executes last statement and asks the operating system to delete it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- ❖ Parent may terminate execution of children processes (**abort**) for many reasons:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - Some operating system do not allow child to continue if its parent terminates
 - All children terminated - *cascading termination*

Zombies and premature exits

- ❖ **Zombie:** happens if a child exits when its parent is not currently executing wait. The exiting process becomes a zombie occupying a slot in a table maintained in the kernel for process control. Later when the parent executes wait, it can read the status of the child process and set it to rest.
- ❖ **Premature exit:** happens if the parent exits while one or more children are still running. In this case, the children (including zombies) are adopted by the init process (process-id = 1).

fork() Examples

```
#include <stdio.h>
#include <unistd.h>
int main(){
    int pid;
    printf("before fork");
    pid = fork();
    printf("after fork");
    exit(0);
}
```

Output:

If fork is successful the result will be:

before fork
after fork
after fork

If fork failed the result will be:

before fork
after fork

Describe all possible results of the following programs:

```
#include <unistd.h>
main(){
    pid_t pid;
    switch(pid = fork()){
        case -1: fatal("fork failed"); break;
        case 0: execlp("/bin/ls",ls,"-l",NULL); fatal("exec failed"); break;
        default: wait(NULL);
                printf("I am the parent: ls called by child is completed");
        exit(0);
    }
}
```

Modify any of the above programs to create more children for the parent and for these children to create their own children as well.

Examples

```
#include <unistd.h>
main(){
    pid_t pid;
    switch(pid = fork()){
        case -1: fatal("fork failed"); break;
        case 0: printf("child: after fork "); break;
        default: wait(NULL); printf("parent: after fork "); exit(0);
    }
}
```

```
#include <unistd.h>
main(){
    pid_t pid;
    switch(pid = fork()){
        case -1: fatal("fork failed"); break;
        case 0: printf("child: after fork "); break;
        default: printf("parent: after fork "); exit(0);
    }
}
```


Examples

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

int main ()
{
    int x, y;
    pid_t pid;
    int *status;

    x = y = 99;
    if ( (pid = fork ()) < 0 )
        perror ("fork.c");
    else if ( pid == 0 ) {
        x++;
        y++;
        printf ("child: %d %d\n", x, y);
        sleep (5);
        printf ("child died\n");
        return 0;
    }

    printf ("parent: %d %d\n", x, y);
    pid = wait (status);
    printf ("parent died with child status %d\n", *status);

    return 0;
} /* main () */
```

Additional Practice with Fedora

- ❖ Switch to Fedora in order to practice programming with the fork, exec, and wait system calls.
- ❖ Examples will be posted on Blackboard.
 - In Fedora, access the Blackboard using the Firefox browser.
 - Download the examples and try them.
 - You may need to play with the code for deeper understanding.

CMPS 405: OPERATING SYSTEMS


Inter-Process Communications (IPCs) with
unnamed pipes

Unnamed pipes related system calls
pipe, read, write, close, dup, dup2

This material is based on the operating system books by Silberschatz and Stallings

IPCs (Inter Process Communications): Pipes

- ❖ A pipe is a one-way communications channel which couples one related process to another.
 - Pipes are implemented by UNIX domain sockets.
- ❖ Example of pipes at command level:
 - **who | sort | lpr**
 - in this case the output of **who** is the input to sort and the output of **sort** is the input to **lpr**.
- ❖ In a program, a pipe is created by using the pipe system call:
 - **int pipe(int* pipefd);**
- ❖ Two file descriptors are returned and saved in:
 - **pipefd[0]** which is open for reading and
 - **pipefd[1]** which is open for writing.
 - If the call to pipe fails then -1 is returned.



Prototype

Important Statements

- ❖ **write(pipefd[1],buff, sizeof(buff));**
 - writes the contents of the buffer named **buff** to the write-end of the pipe in this case named **pipefd[1]**. The third argument is the size of the buffer to be written.
- ❖ **read(pipefd[0],buff, sizeof(buff));**
 - reads the information from the read-end of the pipe named **pipefd[0]** with the size given in the third argument and save it in the buffer **buff**.
- ❖ **close(pipefd[0]);**
 - closes the pipe's read-end.
- ❖ **close(pipefd[1]);**
 - closes the pipe's write-end.

Q. Using pipes read and write, write a program in which the child sends a string to the parent and the parent get it and display it.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int pfd[2];
    char buf[30];
    pipe(pfd);
    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfd[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfd[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
}
```

The output:

```
PARENT: reading from pipe
CHILD: writing to the pipe
CHILD: exiting
PARENT: read "test"
```


Example 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
int main()
{
    int pfd[2];
    char buf[30];
    if (pipe(pfd) == -1) {
        perror("pipe");
        exit(1);
    }
    printf("writing to file descriptor #%d\n", pfd[1]);
    write(pfd[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfd[0]);
    read(pfd[0], buf, 5);
    printf("read \"%s\"\n", buf);
}
```

Example 2:

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

```
main()
{
    int pipefd[2], n;
    char buff[100];
    if (pipe(pipefd) < 0) {
        error ("pipe error");
    }
    printf ("readfd = %d, writefd = %d\n", pipefd[0], pipefd[1]);
    if (write(pipefd[1], "hello world\n", 12) != 12) {
        error ("write error");
    }
    if ((n=read(pipefd[0], buff, sizeof(buff))) < 0) {
        error ("read error");
    }
    write (1, buff, n);
    exit (0);
}
```

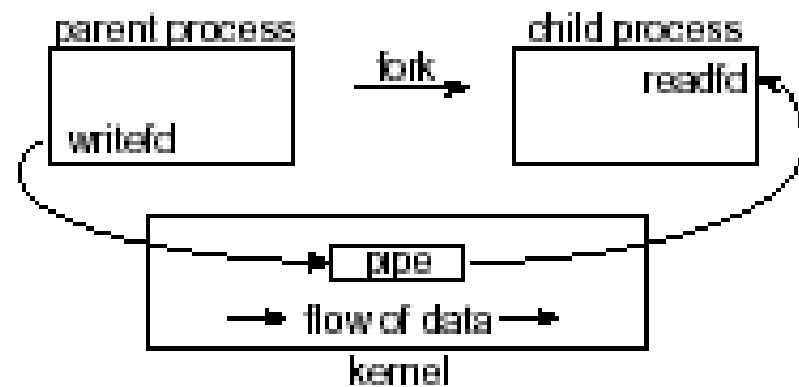
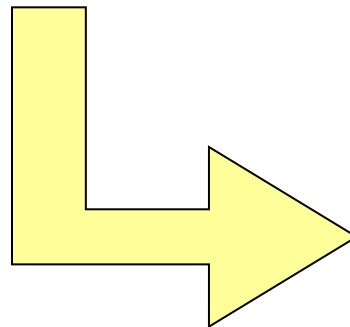
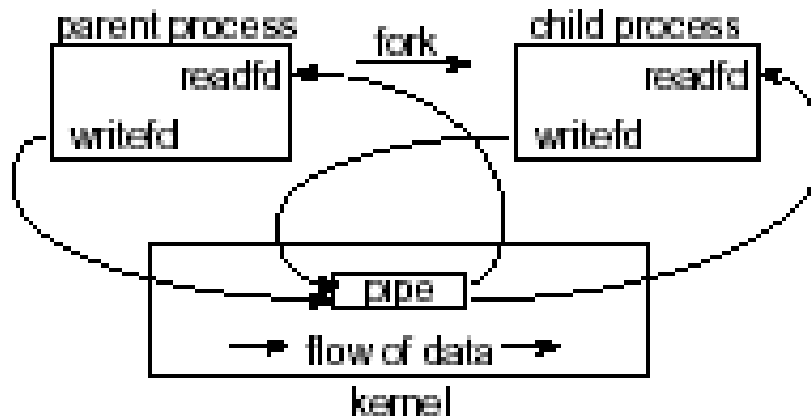


0: stdin
1: stdout
2: stderr

Pipes with fork

- ❖ First a process creates a pipe then calls fork to duplicate itself.
 - Not forgetting that pipes are one way buffered communication channel.
- ❖ **Example:** In a situation where parent opens files and child reads files the following must be done:
 - Parent closes read-end of the pipe and leaves write-end open while,
 - Child closes write-end of the pipe and leaves read-end open.

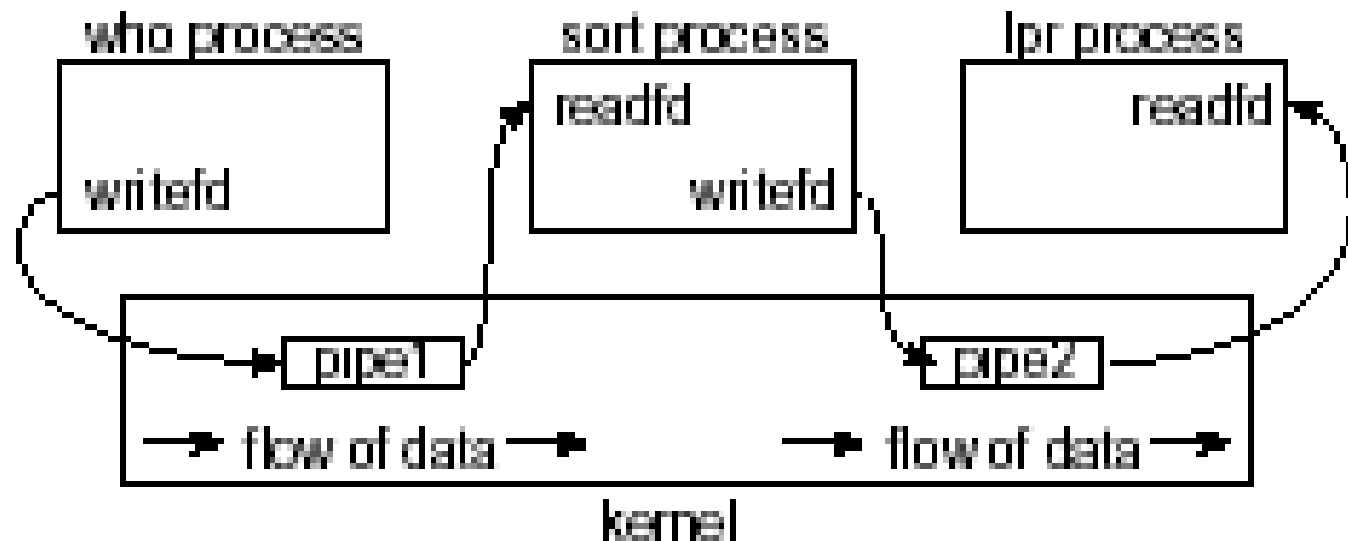
Pipes with fork



Examples:

❖ Q. How pipes are used to implement: `who | sort | lpr` ?

➤ Answer:

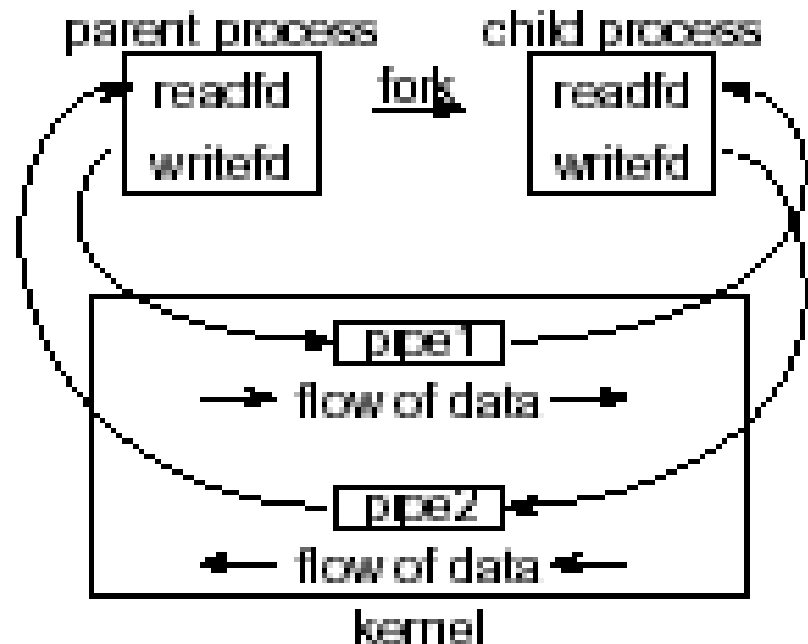


Examples:

- ❖ Q. Explain how to achieve the following: parent sends file name to child. Child opens and reads file, then returns content to parent.

➤ **Answer:**

Parent creates pipe1 and pipe2 then fork then parent closes read-end of pipe1 and write-end of pipe2 then child closes write-end of pipe1 then child closes read-end of pipe2.



The size of the pipe:

- ❖ The size of a pipe is finite, i.e., only a certain amount of bytes can remain in the pipe without being read. Typical size of a pipe is 512 bytes.
- ❖ If a write is made on a pipe and there is enough space, then the data is sent down the pipe and the call returns immediately.
- ❖ If, however, a write is made that will overflow the pipe, process execution is suspended until room is made by another process reading from the pipe.
- ❖ As a final comment: a parent process may have a number of children and pipes associated with it. In this case the problem of handling multiple pipes arises and one solution for it is the use of the ***select*** or ***poll*** system calls.

Q. Write a C program that implements: “ls | wc -l” .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    int pfd[2];
    pipe(pfd);
    if (!fork()) {
        close(1); /* close normal stdout */
        dup(pfd[1]); /* make stdout same as pfd[1] */
        close(pfd[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0); /* close normal stdin */
        dup(pfd[0]); /* make stdin same as pfd[0] */
        close(pfd[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }
}
```

Additional Practice with Fedora

- ❖ Switch to Fedora in order to practice programming with the fork, exec, and wait system calls.
- ❖ Examples will be posted on Blackboard.
 - In Fedora, access the Blackboard using the Firefox browser.
 - Download the examples and try them.
 - You may need to play with the code for deeper understanding.

CMPS 405: OPERATING SYSTEMS

Signals

Signals related system calls

signal, sigaction, kill, raise, pause, alarm

This material is based on the operating system books by Silberschatz and Stallings

Signals

- ❖ Signals are software generated interrupts that are sent to a process when an event happens.
- ❖ OS communicates an occurrence of an event to a process by sending a signal to the process.
- ❖ The process can respond by doing one of three things:
 - Take the default action which is mostly terminating the process.
 - Ignore the signal.
 - Trap the signal. Catching the signal by invoking a user-defined signal handling function.

Signals

- ❖ Each signal is identified by a number and is designed to perform a specific function .
- ❖ Signals are better presented by their symbolic names having the SIG prefix.
- ❖ There are two signals that a process can't ignore or run user-defined code to handle: SIGKILL terminates a process and SIGSTOP suspends a process.
- ❖ To view the list of all signal names and numbers that are available on your machine:
 - Use the command `kill -l`
 - View the file `/usr/include/sys/signal.h`

Listing signals in the shell

user > kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

user >

kill: premature termination of a process

- ❖ The kill command sends a signal usually with the intention of killing a process or multiple processes.
- ❖ SIGTERM is the default signal sent to the process using the kill command.
 - kill 44
 - kill -s TERM 44
 - kill 44 123 76 82
- ❖ If you kill a parent process, all of its children are terminated.

kill: premature termination of a process

- ❖ kill can be used to send a specific signal to a process by specifying the signal name without SIG after the option – s or using the option with signal number.
 - kill -s KILL 44
 - kill -9 44
 - sends the SIGKILL signal (signal number 9 in Linux) to the process with PID 44 as some processes may ignore the default kill with SIGTERM.
 - Kill -9 \$\$
 - Kills the shell

Signal sources

- ❖ Signals can be synchronized (predictable) or asynchronized (unpredictable) and can be originated from the following sources:
 - The keyboard: signals generated from the keyboard affect the current foreground jobs. [Ctrl-z] sends SIGTSTP to suspend a job.
 - The hardware: signals can be generated on account of an arithmetic exception like divide-by-zero (SIGFPE), an illegal instruction (SIGILL), or memory access violation (SIGSEGV).
 - A C program: the system call library offers some functions that generate signals. alarm generates SIGALRM after expiry of a specific time. The raise and kill functions can generate any signal.
 - Other sources: when a child dies, the kernel sends SIGCHLD to the parent. When background jobs try to read from the terminal, the terminal driver generates SIGTTIN.

Signal Handling

- ❖ Signals are delivered to the process once they are generated. The delivery of blocked signals is pending until they are unblocked.
- ❖ The original signal handling system was based on signal system call which is not reliable. Therefore, we also present the reliable POSIX signal handling mechanism.
 - POSIX (Portable Operating System Interface) initially known as IEEE-IX.
 - is a family of standards specified by the IEEE for maintaining compatibility between operating systems.
 - POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

The original signal Handling System

- ❖ `#include <signal.h>`
- ❖ `typedef void (*sighandler_t)(int);`
- ❖ `sighandler_t signal(int signo, sighandler_t handler);`
- ❖ The `signal()` system call installs a new signal handler for the signal with number `signo`.
- ❖ The signal handler is set to `sighandler` which may be a user specified function, or either `SIG_IGN` or `SIG_DFL`.
- ❖ Example:
 - `signal(SIGINT, SIG_IGN);`
 - `signal(SIGTSTP, SIG_DFL);`
 - `signal(SIGQUIT, quit_handler);` where `quit_handler` is a function.

The original signal Handling System: Example

```
#include <stdio.h>
#include <signal.h>
void tstp_handler(int signo);
int main (void){
    signal(SIGTSTP,tstp_handler);
    signal(SIGQUIT, SIG_IGN);
    signal(SIGINT,SIG_DFL);

    fprintf(stderr,"Press any: [Ctrl-z] or [Ctrl-q] or [Ctrl-c]\n");
    for(;;)
        pause();
    exit(0);
}

void tstp_handler(int signo){
    signal(SIGTSTP,tstp_handler);
    fprintf(stderr, "Can't stop this program\n");
}
```

The original signal Handling System: Example

```
#include <stdio.h>
#include <signal.h>
void tstp_handler(int signo);
void int_handler(int signo);
void quit_handler(int signo);

main(){
    int pid;
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }
    if (pid == 0){ /* child */
        signal(SIGTSTP, tstp_handler);
        signal(SIGQUIT, quit_handler);
        signal(SIGINT, int_handler);
        while(1);
    }
    else{/* parent. The pid hold id of child */
        sleep(3);
        printf("\nPARENT: sending SIGTSTP to child %d\n\n",pid);
        kill(pid,SIGTSTP);
        sleep(3);
        printf("\nPARENT: sending SIGINT to child %d\n\n",pid);
        kill(pid,SIGINT);
        sleep(3);
        printf("\nPARENT: sending SIGQUIT to child %d\n\n",pid);
        kill(pid,SIGQUIT);
        sleep(3);
        exit(0);
    }
    return 0;
}
```

The original signal Handling System: Example

```
void tstp_handler(int signo){
    signal(SIGTSTP,tstp_handler); /* reset signal */
    fprintf(stderr,"CHILD: I have received a SIGTSTP\n");
}

void int_handler(int signo){
    signal(SIGINT,int_handler); /* reset signal */
    printf("CHILD: I have received a SIGINT. Still alive ...\n");
}

void quit_handler(int signo){
    signal(SIGQUIT, quit_handler);
    printf("CHILD: My DADDY has Killed me!!!\n");
    exit(1);
}
```

Standard ways to send any signal

- ❖ The kill system call sends a signal to one or more processes. It returns 0 for a successful call, -1 otherwise.
 - `kill(pid_t pid, int sig);`
 - If pid is 0, the signal is sent to all processes, except system processes.
- ❖ The raise library function sends a signal to the process itself but not to other processes. It returns zero if successful, and non-zero otherwise.
 - `int raise(int sig)`
 - equivalent to `kill(getpid(), sig);`

The original signal Handling System

❖ Signal is unreliable because:

- The signal handling was reset to default before the handler is invoked. It resets the handler to SIG_DFL.
 - Solution: reinstall the handler at the beginning of the handling function. However, this may lead to a race condition.
 - Some signals might get lost.

POSIX Signal Handling

- ❖ The structure `sigaction` has at least the four members:

```
struct sigaction {  
    void (*sa_handler)(int);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*)(int, siginfo_t *, void *) sa_sigaction;  
}
```

- `sa_sigaction` can be the name of a function (the handler) or one of the following constants:
 - `SIG_IGN` Ignore the signal
 - `SIG_DFL` Restore default handler
- `sa_mask` specifies a set of signals that are blocked while the handler is executing.
- `sa_flags` specifies optional flags.
 - `SA_RSTARTHAND` sets the handler to the default handler after invocation of the handler.
- `sa_sigaction` is only used when `sa_flags` is specified.

Implementing POSIX signal handling in C

- ❖ `#include <signal.h>`
- ❖ Define a prototype of the handling function
 - `alarm_handler(int signo);`
- ❖ provide the code for the handling function
 - `void alarm_handler(int signo){ //.... The body of the function ...}`
- ❖ Define a sigaction structure
 - `struct sigaction act;`
- ❖ Optionally, can use the C library function `void *memset(void *str, int c, size_t n)` which copies the character `c` (an unsigned char) to the first `n` characters of the string pointed to by the argument `str`.
 - `memset(&act, '\0', sizeof(act));`

Implementing POSIX signal handling in C

- ❖ set the handler in the sigaction structure act
 - `act.sa_handler=alarm_handler; // or`
 - `act.sa_handler=SIG_IGN; // act.saction=SIG_DFL;`
- ❖ install the handler
 - `sigaction(SIGALRM, &act, NULL); // after this the handler will be called when SIGALRM is sent to the process.`
- ❖ It worth mentioning that the pause system call suspends the execution of the process until it receives any signal.

Additional Practice with Fedora

- ❖ Switch to Fedora in order to practice programming with signals related system calls.
- ❖ Examples will be posted on Blackboard.
 - In Fedora, access the Blackboard using the Firefox browser.
 - Download the examples and try them.
 - You may need to play with the code for deeper understanding.
- ❖ Rewrote the examples using the signal system call

CMPS 405: OPERATING SYSTEMS

Related C functions to pipes and exec
mkfifo, snprintf, r_write, system, popen, and pclose C
functions.

The unlike system calls.

This material is based on the operating system books by Silberschatz and Stallings

FIFOs (named pipes)

- ❖ A FIFO has a name within the file system and is opened in the same way as a regular file.
- ❖ Once a FIFO has been created, any process can open it, subject to the usual file permission checks.
- ❖ Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., `read()`, `write()`, and `close()`).
- ❖ Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written.
 - This fact gives FIFOs their name: first in, first out. FIFOs are also sometimes known as named pipes.
- ❖ As with pipes, when all descriptors referring to a FIFO have been closed, any outstanding data is discarded.
- ❖ To remove a FIFO use the `rm` command or the `unlink` system call.

FIFOs

```
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

- Returns 0 on success, or -1 on error
- The mode argument specifies the permissions for the new FIFO. These permissions are specified by ORing the desired combination of constants.
- #include <fcntl.h>
- Constants for file permission bits →
- In addition to the constants shown in the Table, three constants are defined to equate to masks for all three permissions for each of the categories owner, group, and other:
 - S_IRWXU (0700),
 - S_IRWXG (070), and
 - S_IRWXO (07).

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

FIFOs

❖ Creating a FIFO:

- In a C program using the function

```
mkfifo("myfifo", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

- Or,

```
#define FIFO_PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

```
//...
```

```
mkfifo("myfifo", FIFO_PERMS);
```

❖ Removing a FIFO:

- In the shell using the remove system command

- `rm myfifo`

- In a C program using the system call

- `unlike("myfifo");`

FIFOs

- ❖ Pipes and FIFOs have a very important property:
 - Writes of no more than PIPE_BUF are guaranteed to be atomic (no interleaving bytes from other writes).
 - In contrast, fprintf is not atomic, so pieces of the messages from multiple readers might be interspersed.
- ❖ FIFOs can be used to code
 - A logging server: clients write their logging char to server.
 - A request-reply protocol: multiple clients requesting from a server that replies to each separately using dedicated FIFOs.
 - A barrier: used by cooperate processes to block until all processes reach a particular point. N processes each writing a request character to a request FIFO when they reach the barrier point and only then the server reports N release characters to the release FIFO.

popen, pclose, and system functions

❖ The popen() function:

- creates a pipe, and then forks a child process that execs a shell, which in turn creates a child process to execute the string given in command.

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

```
FILE *popen(const char *command, const char *mode);
```

- Returns file stream, or NULL on error.
- The mode argument is a string that determines whether the calling process will read from the pipe (mode is r) or write to it (mode is w).

```
int pclose(FILE *stream);
```

- Returns termination status of child process, or -1 on error
 - Example:
- After the popen() call, the calling process uses the pipe to read the output of command or to send input to it.

popen, pclose, and system functions

- ❖ popen() builds the pipe, performs descriptor duplication, closes unused descriptors, and handles all of the details of fork() and exec() on our behalf.
- ❖ the calling process runs in parallel with the shell command and then calls pclose().
 - the calling process may create other child processes between the execution of popen() and pclose()
- ❖ With system() function, the execution of the shell command is encapsulated within a single function call.

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

```
int system(const char *command);
```

- The system() function creates a child process that invokes a shell to execute command.
- Example: system("ls | wc");

Additional Practice with Fedora

- ❖ Switch to Fedora in order to practice programming with related system calls.
- ❖ Examples will be posted on Blackboard.
 - In Fedora, access the Blackboard using the Firefox browser.
 - Download the examples and try them.
 - You may need to play with the code for deeper understanding.
- ❖ Write additional examples or develop small applications using these system calls.