POSIX Signal Handling

A signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts.

Signals can be sent

- By one process to another process (or to itself)
- By the kernel to a process

Every signal is associated with an action.

For example: The SIGCHLD signal is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.

We set the disposition of a signal by calling the sigaction function and we have three choices for the disposition:

1. Catching a signal. We can provide a function called a signal handler that is called whenever a specific signal occurs. The two signals SIGKILL and SIGSTOP cannot be caught. Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore:

   ```
   void handler(int signo);
   ```

   For most signals, we can call sigaction and specify the signal handler to catch it. A few signals, SIGIO, SIGPOLL, and SIGURG, all require additional actions on the part of the process to catch the signal.

2. Ignoring a signal. We can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored.

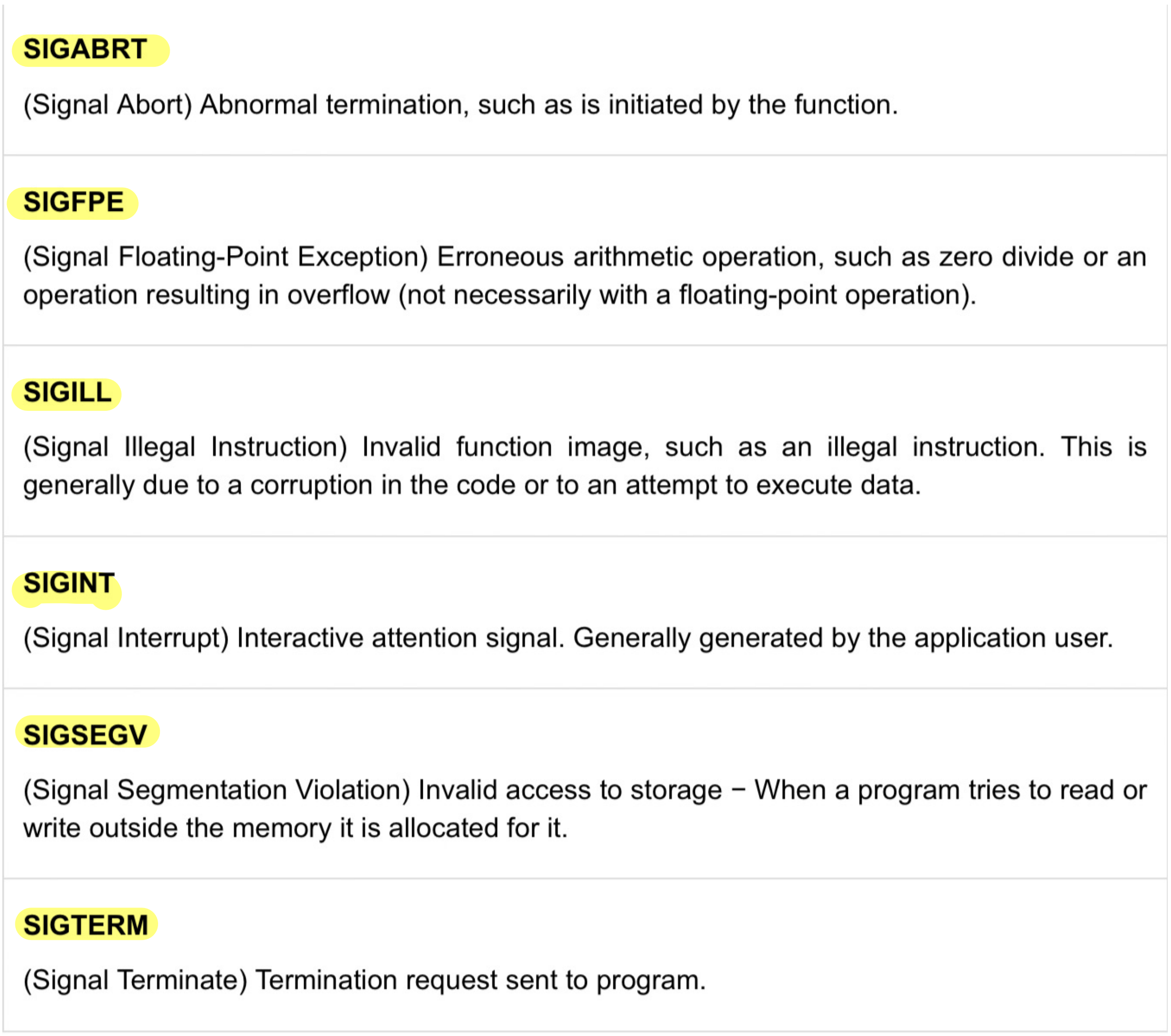
3. Setting the default disposition for a signal. This can be done by setting its disposition to SIG_DFL. The default is normally to terminate a process on receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: SIGCHLD and SIGURG (sent on the arrival of out-of-band data) are two that we will encounter in this text.

signal Function

The C library function void (*signal(int sig, void (*func)(int)))(int) sets a function to handle signal i.e. a signal handler with signal number sig.

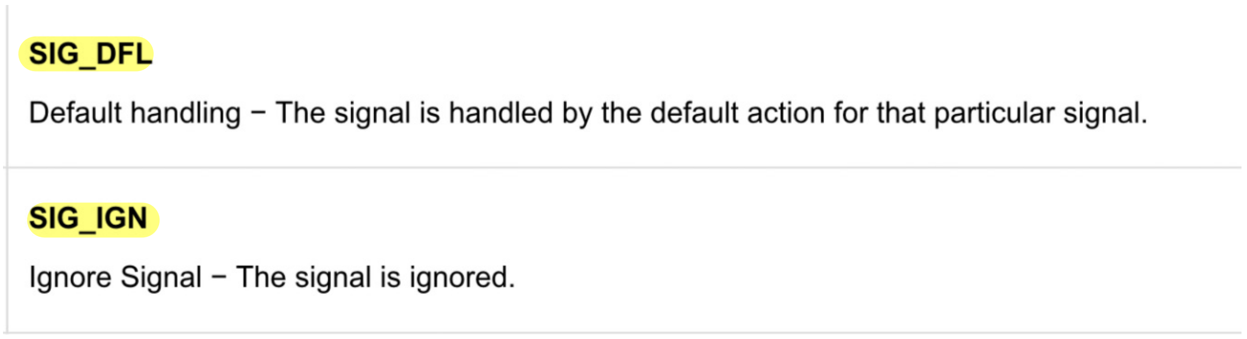
```
void (*signal(int sig, void (*func)(int)))(int)
```

sig – This is the signal number to which a handling function is set. The following are few important standard signal numbers



SIGABRT	(Signal Abort) Abnormal termination, such as is initiated by the function.
SIGFPE	(Signal Floating-Point Exception) Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow (not necessarily with a floating-point operation).
SIGILL	(Signal Illegal Instruction) Invalid function image, such as an illegal instruction. This is generally due to a corruption in the code or to an attempt to execute data.
SIGINT	(Signal Interrupt) Interactive attention signal. Generally generated by the application user.
SIGSEGV	(Signal Segmentation Violation) Invalid access to storage – When a program tries to read or write outside the memory it is allocated for it.
SIGTERM	(Signal Terminate) Termination request sent to program.

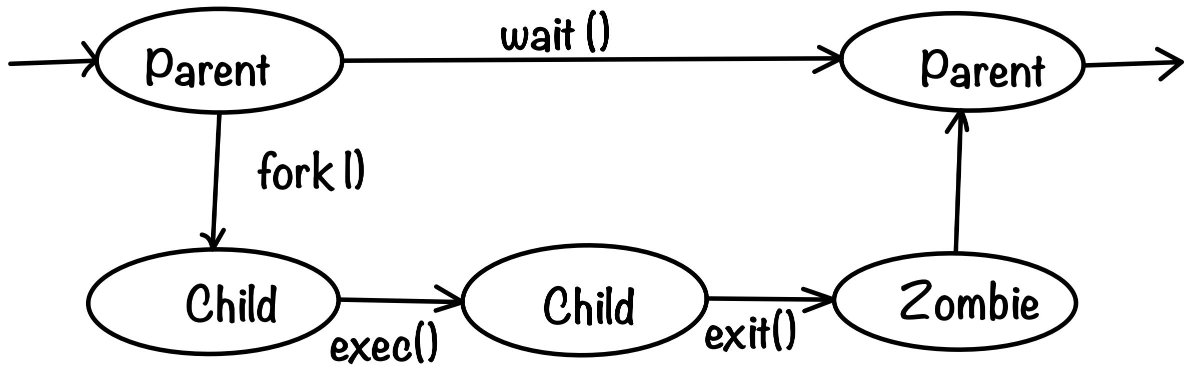
func – This is a pointer to a function. This can be a function defined by the programmer or one of the following predefined functions



SIG_DFL	Default handling – The signal is handled by the default action for that particular signal.
SIG_IGN	Ignore Signal – The signal is ignored.

Zombie process

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status.



ps -eaf   Command to check for the process table

Why zombie process are dangerous?

There is one process table per system. The size of the process table is finite. If too many zombie processes are generated, then the process table will be full. That is, the system will not be able to generate any new process, then the system will come to a standstill. Hence, we need to prevent the creation of zombie processes.

How to prevent zombie process from creating?

```c
// A C program to demonstrate ignoring
// SIGCHLD signal to prevent Zombie processes
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();
    if (pid == 0)
        for (i=0; i<20; i++)
            printf("I am Child\n");
    else
    {
        signal(SIGCHLD,SIG_IGN);
        printf("I am Parent\n");
        while(1);
    }
}
```

Handling Interrupts

If user presses ctrl-c to terminate the process because of SIGINT signal sent and its default handler to terminate the process.

```c
// CPP program to illustrate
// User-defined Signal Handler
#include<stdio.h>
#include<signal.h>

// Handler for SIGINT, caused by
// Ctrl-C at keyboard
void handle_sigint(int sig)
{
    printf("Caught signal %d\n", sig);
}

int main()
{
    signal(SIGINT, handle_sigint);
    while (1) ;
    return 0;
}
```

Handling Interrupted System Calls

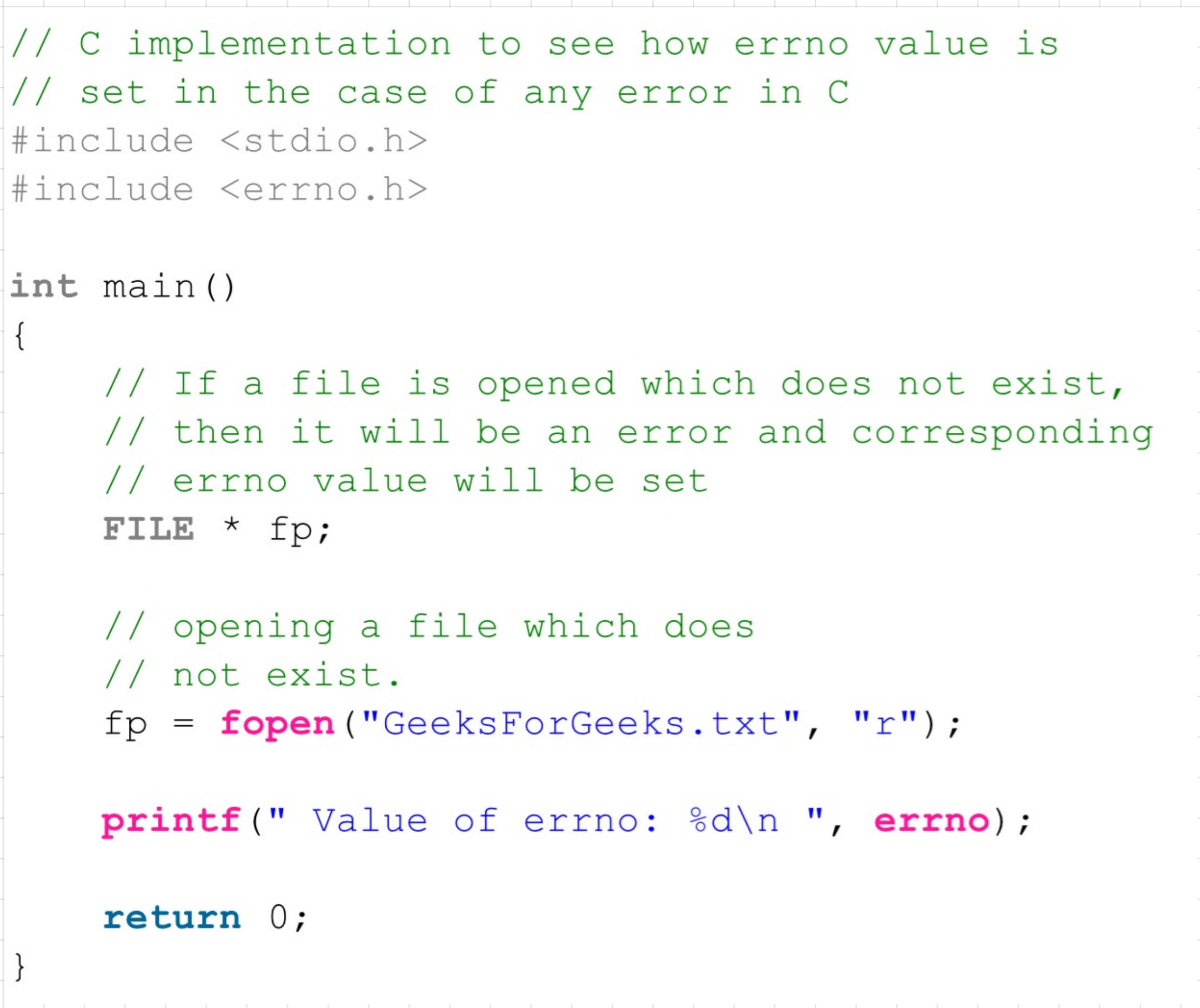
The term "slow system call" is used to describe any system call that can block forever, such as accept. That is, the system call need never return. Most networking functions fall into this category. Examples are:

- accept: there is no guarantee that a server's call to accept will ever return, if there are no clients that will connect to the server.
- read: the server's call to read in server will never return if the client never sends a line for the server to echo.

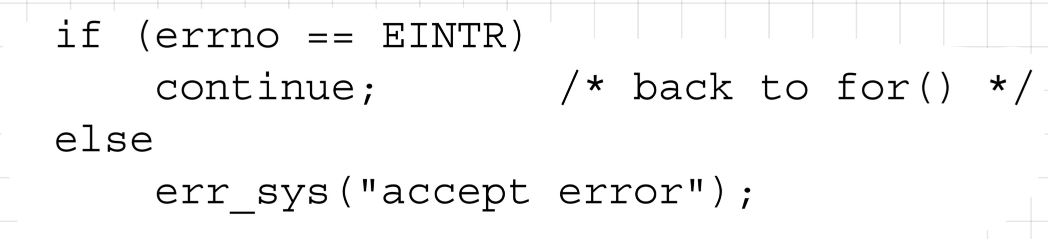
In such cases EINTR error is rised.

To compare error generated by the function we need errno.h header

errno value	Error
1	/* Operation not permitted */
2	/* No such file or directory */
3	/* No such process */
4	/* Interrupted system call */
5	/* I/O error */
6	/* No such device or address */
7	/* Argument list too long */
8	/* Exec format error */
9	/* Bad file number */
10	/* No child processes */
11	/* Try again */
12	/* Out of memory */
13	/* Permission denied */



```c
// C implementation to see how errno value is
// set in the case of any error in C
#include <stdio.h>
#include <errno.h>

int main()
{
    // If a file is opened which does not exist,
    // then it will be an error and corresponding
    // errno value will be set
    FILE * fp;

    // opening a file which does
    // not exist.
    fp = fopen("GeeksForGeeks.txt", "r");

    printf(" Value of errno: %d\n ", errno);

    return 0;
}
```



```c
if (errno == EINTR)
    continue;       /* back to for() */
else
    err_sys("accept error");
```

strerror(): returns a pointer to the textual representation of the current errno value.

wait and waitpid Functions

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 or –1 on error

wait and waitpid both return two values:

1. the return value of the function is the process ID of the terminated child
2. the termination status of the child (an integer)

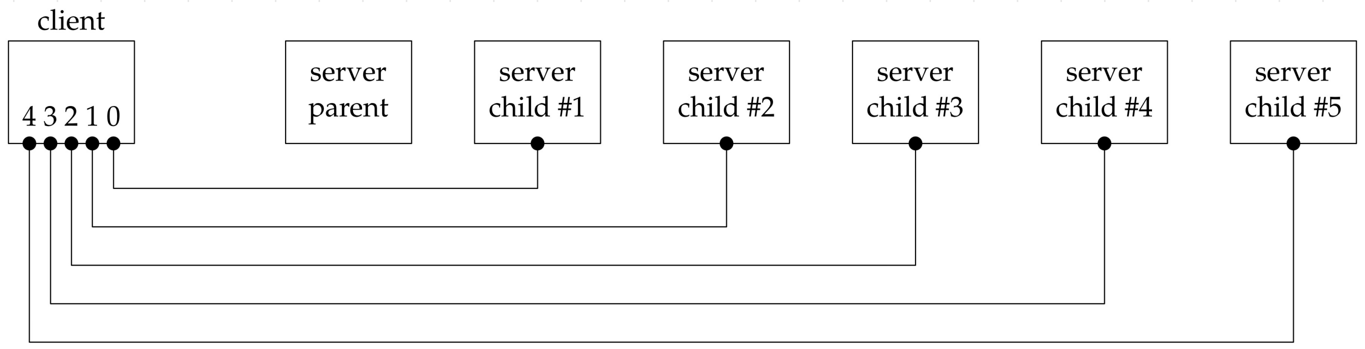
Using wait to precast zombie process:

```
// A C program to demonstrate handling of
// SIGCHLD signal to prevent Zombie processes.
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

void func(int signum)
{
    wait(NULL);
}

int main()
{
    int i;
    int pid = fork();
    if (pid == 0)
        for (i=0; i<20; i++)
            printf("I am Child\n");
    else
    {
        signal(SIGCHLD, func);
        printf("I am Parent\n");
        while(1);
    }
}
```

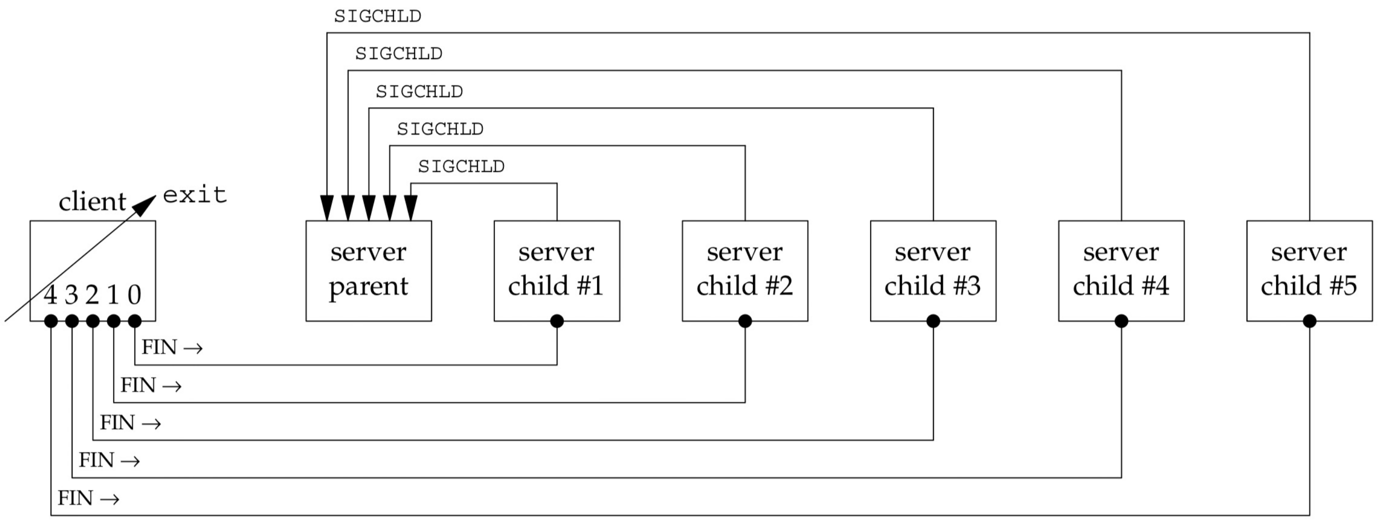
Makes the parent to collect the exist status of child.

waitpid gives us more control over which process to wait for and whether or not to block. First, the pid argument lets us specify the process ID that we want to wait for. A value of −1 says to wait for the first of our children to terminate.

How wait is different to waitpid function?



client
4 3 2 1 0
server parent
server child #1
server child #2
server child #3
server child #4
server child #5



SIGCHLD
SIGCHLD
SIGCHLD
SIGCHLD
SIGCHLD
client
exit
4 3 2 1 0
server parent
server child #1
server child #2
server child #3
server child #4
server child #5
FIN →
FIN →
FIN →
FIN →
FIN →