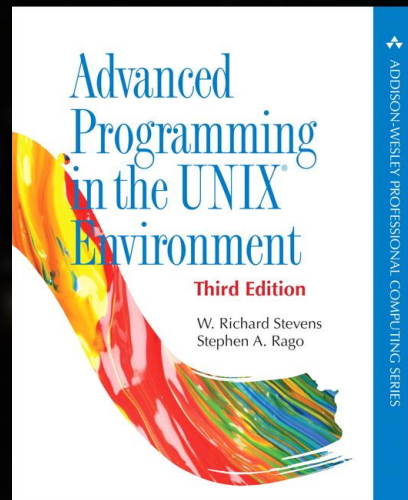


# KILL BILL

VOLUME 1



Kill Bill (2003) - Quentin Tarantino



# Chapter 10: Signal

## Chapter 15: Inter-Process Communication

---

# Multiprocessing

aka multiprocessing

---

Single Processor Multiprocessor

---

# Inter-Process Communication

Parent ↔ Child

Any Process ↔ Any Other Process

---

Single Processor Multiprocessor



---

# Parent → Child

Passing Tasks

Passing Information

---

After `fork()`, any change to the variables are local to the parent and child processes.

After `fork()`, there is no conversation/communication until ...

Parent

Parent → Child

13

```
int child_pid = fork();
if(child_pid == -1){
    perror("impossible to have a child!");
    exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
    if(child_pid > 0)
        printf("I am the parent, pid=%d\n", getpid());
    else{//(child_pid == 0)
        printf("I am the child, pid=%d\n", getpid());
        printf("My parent is pid=%d\n", getppid());
    }
}

//Assign parent tasks here
int child_exit;
wait(&child_exit);
```

Child

Parent → Child

0

```
int child_pid = fork();
if(child_pid == -1){
    perror("impossible to have a child!");
    exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
    if(child_pid > 0)
        printf("I am the parent, pid=%d\n", getpid());
    else{//(child_pid == 0)
        printf("I am the child, pid=%d\n", getpid());
        printf("My parent is pid=%d\n", getppid());
        exit(0);
    }
}
```

---

# Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

---

```
#include <sys/wait.h>
pid_t wait(int *statloc);
```

Return Child's PID if OK, or -1 on error

---

## Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

---

```
#include <sys/wait.h>
pid_t wait(0);
```

 Parent does not care about how the child terminates!

Return Child's PID if OK, or `-1` on error



---

# Wait for Child Process be over

`int *statloc → status`



Higher Order Byte  
[0x00, 0xFF]

Lower Order Byte  
[0x00, 0xFF]

Macro	Description
<code>WIFEXITED (status)</code>	<p>True if status was returned for a child that terminated normally. In this case, we can execute</p> <p style="text-align: center;"><code>WEXITSTATUS (status)</code></p> <p>to fetch the low-order 8 bits of the argument that the child passed to <code>exit</code>, <code>_exit</code>, or <code>_Exit</code>.</p>
<code>WIFSIGNALED (status)</code>	<p>True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute</p> <p style="text-align: center;"><code>WTERMSIG (status)</code></p> <p>to fetch the signal number that caused the termination.</p> <p>Additionally, some implementations (but not the Single UNIX Specification) define the macro</p> <p style="text-align: center;"><code>WCOREDUMP (status)</code></p> <p>that returns true if a core file of the terminated process was generated.</p>
<code>WIFSTOPPED (status)</code>	<p>True if status was returned for a child that is currently stopped. In this case, we can execute</p> <p style="text-align: center;"><code>WSTOPSIG (status)</code></p> <p>to fetch the signal number that caused the child to stop.</p>
<code>WIFCONTINUED (status)</code>	<p>True if status was returned for a child that has been continued after a job control stop (XSI option; <code>waitpid</code> only).</p>

**Figure 8.4** Macros to examine the termination status returned by `wait` and `waitpid`

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    int a = 0;
    int b = 0;
    a = atoi(argv[1]);
    b = atoi(argv[2]);

    printf("I am a lonely process, pid=%d\n", getpid());
    int child_pid = fork();
    if(child_pid == -1){
        perror("impossible to have a child!\n");
        exit(1);
    }
    if(child_pid >= 0){ // (child_pid != -1)
        if(child_pid > 0)
            printf("I am the parent, pid=%d\n", getpid());
        else{// (child_pid == 0)
            printf("I am the child, pid=%d\n", getpid());
            //Assign child's tasks here
            printf("child: %d + %d = %d\n", a, b, a - b);
            exit(0);
        }
    }
    //Assign parent tasks here
    printf("parent: %d + %d = %d\n", a, b, a + b);

    int child_exit;
    wait(&child_exit);

    if (WIFEXITED(child_exit))
        printf("normal termination, exit status = %d\n", WEXITSTATUS(child_exit));
    else if (WIFSIGNALED(child_exit))
        printf("abnormal termination, signal number = %d\n", WTERMSIG(child_exit));
}

```

hfani@alpha:~\$ ./child\_exit\_status 3 5

I am a lonely process, pid=1911307

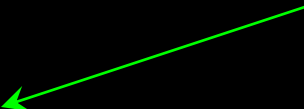
I am the parent, pid=1911307

parent: 3 + 5 = 8

I am the child, pid=1911308

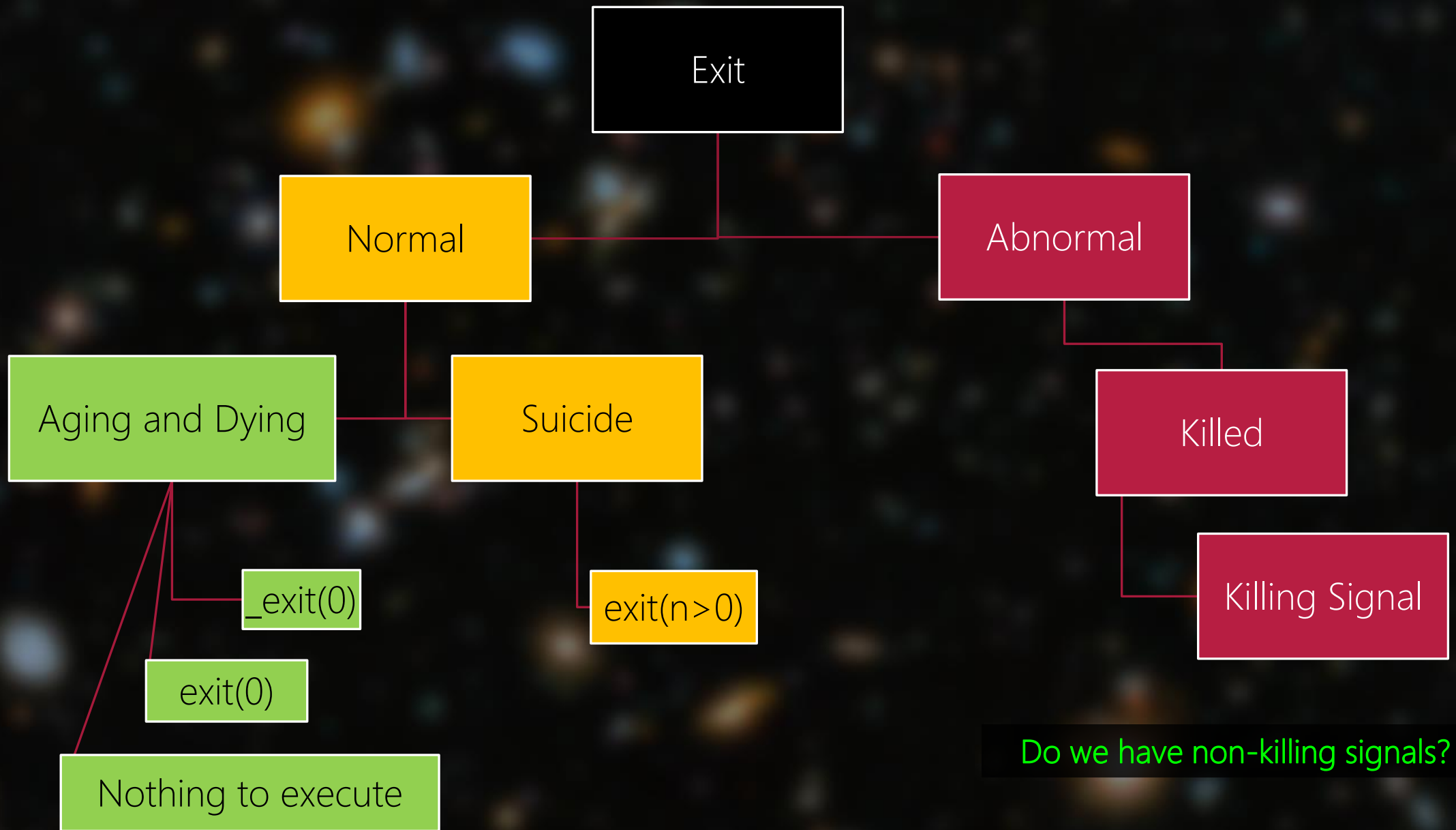
child: 3 + 5 = -2

normal termination, exit status = 0



Macro	Description
<code>WIFEXITED (status)</code>	<p>True if <code>status</code> was returned for a child that terminated normally. In this case, we can execute</p> <p style="text-align: center;"><code>WEXITSTATUS (status)</code></p> <p>to fetch the low-order 8 bits of the argument that the child passed to <code>exit</code>, <code>_exit</code>, or <code>_Exit</code>.</p>
<code>WIFSIGNALED (status)</code>	<p>True if <code>status</code> was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute</p> <p style="text-align: center;"><code>WTERMSIG (status)</code></p> <p>to fetch the signal number that caused the termination.</p> <p>Additionally, some implementations (but not the Single UNIX Specification) define the macro</p> <p style="text-align: center;"><code>WCOREDUMP (status)</code></p> <p>that returns true if a core file of the terminated process was generated.</p>
<code>WIFSTOPPED (status)</code>	<p>True if <code>status</code> was returned for a child that is currently stopped. In this case, we can execute</p> <p style="text-align: center;"><code>WSTOPSIG (status)</code></p> <p>to fetch the signal number that caused the child to stop.</p>
<code>WIFCONTINUED (status)</code>	<p>True if <code>status</code> was returned for a child that has been continued after a job control stop (XSI option; <code>waitpid</code> only).</p>

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`



Do we have non-killing signals? Yes!



---

# Signaling

Like Electric Shock (IRQ) from Devices to Processor (hardware), **Signals are Process Shock to Another Process (software)**  
Software Interrupts

---

Kernel Process → Other Processes  
Parent → Child  
Ancestor Process → Grandchildren

Name	Description	ISO C	SUS	FreeBSD	Linux	Mac OS X	Solaris	Default action
				8.0	3.2.0	10.6.8	10	
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers	•		•	•	•	•	terminate
SIGPOLL	pollable event (poll)				•		•	terminate
SIGPROF	profiling time alarm (setitimer)			•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXRES	resource control exceeded						•	ignore

Figure 10.1 UNIX System signals

---

# Terminal/Shell → Process

SIGINT (interrupt signal), SIGTSTP (terminal stop)

When user hits `Ctrl+C` or `Ctrl+Z` keys, an IRQ (device manger, file manager) becomes a Signal (process manager)

---

SIGTSTP (non-killing): To stop a runaway process

SIGINT (killing): To terminate a runaway process

---

Processor → Kernel → Process

SIGILL (illegal Instruction)

SIGFPE (floating point exception, e.g., division by 0)

SIGSEGV (invalid memory reference)

---

An IRQ (device manger, file manager) becomes a Signal (process manager)

In general, any hardware can generate an error that becomes a signal to a process with the help of kernel!

*All Killing!*



---

Child → Kernel → Parent

SIGCHLD

We've already seen this behind the scene for `wait()` by the parent

Non-killing!

---



---

Parent	→ Child
Ancestor	→ Grandchildren
Powerful Process	→ Other Processes

SIGXXX: Killing, non-Killing (warning)

---

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

0 if OK, -1 on error (signal# invalid, pid invalid, or not have permission to send the signal to any receiving process)



---

Do we have a non-killing signal? Yes!

- Those whose default behavior is ignore!
  - Those whose default behavior is stopping!
-

Name	Description	ISO C	SUS	FreeBSD	Linux	Mac OS X	Solaris	Default action
				8.0	3.2.0	10.6.8	10	
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers	•		•	•	•	•	terminate
SIGPOLL	pollable event (poll)				•		•	terminate
SIGPROF	profiling time alarm (setitimer)			•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXRES	resource control exceeded						•	ignore

Figure 10.1 UNIX System signals

---

## +core

Dumping the memory image of the process before termination  
Into a file in the current working directory of the process  
Historically, the name of the file was "core"

---

This file can be used with most debuggers to examine the state of the process at the time it terminated.

Not part of POSIX!

Name	Description	ISO C	SUS	FreeBSD	Linux	Mac OS X	Solaris	Default action
				8.0	3.2.0	10.6.8	10	
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers	•		•	•	•	•	terminate
SIGPOLL	pollable event (poll)				•		•	terminate
SIGPROF	profiling time alarm (setitimer)			•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXRES	resource control exceeded						•	ignore

Figure 10.1 UNIX System signals



---

default behavior!  
terminate/ignore

Does this mean we can change the (default) behavior of a signal? Yes!

---



---

# Signal Handling

aka disposition of a signal or action associated w/ a signal

---

The receiver process should do what?

---

# Signal Handling

aka disposition of a signal or action associated w/ a signal

---

The receiver process should do what?

- 1) Ignore it! Do nothing. Sometimes it is not at your hand though (SIGKILL, SIGSTOP, SIGFPE, ...)
- 2) Let the default action happen (every signal has already assigned to a default action)
- 3) Catch the signal and handle it properly (define a function as signal handler)





Kill Bill (2003) - Quentin Tarantino



The background of the slide is a deep space image showing numerous galaxies in various colors (yellow, orange, blue, white) against a black background. A thin, solid blue horizontal line spans the width of the slide, positioned just above the main title.

# Catch the Signal

By the receiver process

A thin, solid blue horizontal line spans the width of the slide, positioned just below the subtitle.



---

# Catch the Signal

By the receiver process

Reminder from C program

We can have pointer to a function (like a pointer to a variable)

You can call the function by its address rather than by its name!

```
return_type func_name(type0 arg0, type1 arg1, ...) → return_type (*func_ptr)(type0, type1, ...)
```

```
hfani@alpha:~$ vi func_pointer.c
```

```
#include <stdio.h>
void say_hi(int times){
    for(int i = 0; i < times; ++i)
        printf("Hi!\n");
}
int main() {

    printf("saying hi by calling the function's name:\n");
    say_hi(3);
}
```

```
hfani@alpha:~$ cc func_pointer.c -o func_pointer
```

```
hfani@alpha:~$ ./func_pointer
```

```
saying hi by calling the function's name:
```

```
Hi!
```

```
Hi!
```

```
Hi!
```

```
hfani@alpha:~$ vi func_pointer.c
```

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

```
void say_hi(int times){
```

```
    for(int i = 0; i < times; ++i)
```

```
        printf("Hi!\n");
```

```
}
```

```
int main() {
```

```
    printf("saying hi by calling the function's name:\n");
```

```
    say_hi(3);
```

```
    printf("saying hi by calling the function's address:\n");
```

```
    void (*func_ptr)(int); /* function pointer declaration */
```

func\_ptr can keep address of ANY function that has

- 1) Single input argument of type int
- 2) output of type void

```
hfani@alpha:~$ vi func_pointer.c
```

```
#include <stdio.h>
void say_hi(int times){
    for(int i = 0; i < times; ++i)
        printf("Hi!\n");
}
int main() {

    printf("saying hi by calling the function's name:\n");
    say_hi(3);

    printf("saying hi by calling the function's address:\n");
    void (*func_ptr)(int); /* function pointer declaration */
    func_ptr = say_hi; /* pointer assignment */
```



Pointer assignment happens just by assigning the name of the function to the pointer!

```
hfani@alpha:~$ vi func_pointer.c
```

```
#include <stdio.h>
void say_hi(int times){
    for(int i = 0; i < times; ++i)
        printf("Hi!\n");
}
int main() {

    printf("saying hi by calling the function's name:\n");
    say_hi(3);

    printf("saying hi by calling the function's address:\n");
    void (*func_ptr)(int); /* function pointer declaration */
    func_ptr = say_hi; /* pointer assignment */
    func_ptr(3); /* function call */

    return 0;
}
```

Call a function by its name is exactly the same as its pointer (address)

```
saying hi by calling the function's address:
```

```
Hi!
Hi!
Hi!
```



---

# Catch the Signal

the receiver process subscribe a function to the kernel for one or more signals

---

```
#include <signal.h>
```

```
typedef void func(int);  
func_ptr *signal(int, func_ptr *);
```

Returns previous signal catcher if OK, SIG\_ERR on error



---

# Example I

- 1) Parent send a signal to the child (hurry up)
  - 2) Child catches the signal and decides to ignore or do sth about it
-

```

int main(int argc, char *argv[]){
    printf("I am a lonely process, pid=%d\n", getpid());
    int child_pid = fork();
    if(child_pid == -1){
        perror("impossible to have a child!\n");
        exit(1);
    }
    if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
            printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)

```

## Child Tasks

```

        printf("I sleep for 10 second and will be back with you to check on your progress, child.\n");
        sleep(10);
        kill(child_pid, SIGUSR1);

        printf("I sleep for another 10 second. Hopefully you're done by then.\n");
        sleep(10);
        kill(child_pid, SIGUSR1);

        printf("What's up child.\n");
        sleep(10);
        kill(child_pid, SIGUSR1);

        int child_exit;
        wait(&child_exit);
        if (WIFEXITED(child_exit))
            printf("normal termination, exit status = %d\n", WEXITSTATUS(child_exit));
        else if (WIFSIGNALED(child_exit))
            printf("abnormal termination, signal number = %d\n", WTERMSIG(child_exit));

        exit(0);
    }
}

```

Parent Tasks  
 Here, parent does nothing  
 Just does follow ups on the child  
 Every 10 seconds

```
int main(int argc, char *argv[]){
    printf("I am a lonely process, pid=%d\n", getpid());
    int child_pid = fork();
    if(child_pid == -1){
        perror("impossible to have a child!\n");
        exit(1);
    }
    if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
            printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
            exit(0);
        }
    }
}
```

## Child Tasks

In order to catch the signals of the parent, the child must:

- 1) Define a function to catch the signal
- 2) Subscribe the function to the kernel
- 3) Does her other tasks till the parent sends the signal

```

int signal_counter = 2;
void child_signal_handler(int signal){
    printf("I received a signal from my parent %d\n", getpid());
    printf("The signal is %d\n", signal);
    if(signal_counter <= 0){
        printf("Sorry ma! I couldn't finish it. It's not my fault. Kernel does not give me the processor. I quit!\n");
        exit(1);
    }
    else{
        printf("I still have %d chances. Let's finish it soon.\n", signal_counter);
        --signal_counter;
    }
}

```

In order to catch the signals of the parent, the child must:

- 1) Define a function to catch the signal → this function can handle any signal (why?)

```

int main(int argc, char *argv[]){
    printf("I am a lonely process, pid=%d\n", getpid());
    int child_pid = fork();
    if(child_pid == -1){
        perror("impossible to have a child!\n");
        exit(1);
    }
    if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
            printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
            printf("I am the child, pid=%d\n", getpid());
            //subscribe a signal handler to the kernel
            void *res = signal(SIGUSR1, child_signal_handler);
            if(res == SIG_ERR)
                printf("subscription for SIGUSR1 was not successful!\n");

            while(1){} //busy waiting
            exit(0);
        }
    }
}

```

In order to catch the signals of the parent, the child must:

- 1) Define a function to catch the signal
- 2) Subscribe the function to the kernel → the child only subscribe the function for SIGUSR1 signal



```

int main(int argc, char *argv[]){
    printf("I am a lonely process, pid=%d\n", getpid());
    int child_pid = fork();
    if(child_pid == -1){
        perror("impossible to have a child!\n");
        exit(1);
    }
    if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
            printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
            printf("I am the child, pid=%d\n", getpid());
            //subscribe a signal handler to the kernel
            void *res = signal(SIGUSR1, child_signal_handler);
            if(res == SIG_ERR)
                printf("subscription for SIGUSR1 was not successful!\n");

            while(1){} //busy waiting
            exit(0);
        }
    }
}

```

In order to catch the signals of the parent, the child must:

- 1) Define a function to catch the signal
- 2) Subscribe the function to the kernel
- 3) Does her other tasks till the parent sends the signal → busy waiting, not doing anything :D

```
hfani@alpha:~$ cc parent_child_signal.c -o parent_child_signal
hfani@alpha:~$ ./parent_child_signal
I am a lonely process, pid=1886308
I am the parent, pid=1886308
I sleep for 10 second and will be back with you to check on your progress, child.
I am the child, pid=1886309
I sleep for another 10 second. Hopefully you're done by then.
I received a signal from my parrent 1886308
The signal is 10
I still have 2 chances. Let's finish it soon.
What's up child.
I received a signal from my parrent 1886308
The signal is 10
I still have 1 chances. Let's finish it soon.
I received a signal from my parrent 1886308
The signal is 10
Sorry ma! I couldn't finish it. It's not my fault. Kernel does not give me the processor. I quit!
normal termination, exit status = 1
hfani@alpha:~$
```

Build and run the program to feel the conversation between the parent and her child.

# Example I

- 1) Parent send a signal to the child (hurry up)
- 2) Child catches the signal and decides to ignore or do sth about it

If the child finishes sooner, what happens to the signals of the parent?

# Example I

- 1) Parent send a signal to the child (harry up)
- 2) Child catches the signal and decides to ignore or do sth about it

What if the child wants to subscribe the same signal handler for another signal, e.g., SIGUSR2?

A cosmic background image featuring a dense field of galaxies in various colors (yellow, orange, blue, and white) against a dark space. Two horizontal blue lines are positioned above and below the central text.

# SIGKILL, SIGSTOP

- 1) Cannot be Ignored
- 2) Cannot be caught





A cosmic background image featuring a dense field of galaxies in various colors (yellow, orange, blue, and white) against a dark space. Two horizontal blue lines are positioned above and below the central text.

## Example II

Child ignores all the parent's signals

```
void child_signal_handler(int signal){  
    printf("I received a signal from my parrent %d\n", getppid());  
    printf("The signal is %d\n", signal);  
    printf("I don't care about your signal, ma! I simply ignore it");  
}
```

The signal handler does nothing, just printing out messages.

```
printf("I sleep for 10 second and will be back with you to check on your progress, child.\n");
sleep(10);
kill(child_pid, SIGUSR1);

printf("I sleep for another 10 second. Hopefully you're done by then.\n");
sleep(10);
kill(child_pid, SIGUSR1);

printf("What's up child.\n");
sleep(10);
kill(child_pid, SIGUSR1);

printf("Seems you don't care. I'll kill you!");
kill(child_pid, SIGINT);

int child_exit;
wait(&child_exit);
if (WIFEXITED(child_exit))
    printf("normal termination, exit status = %d\n", WEXITSTATUS(child_exit));
else if (WIFSIGNALED(child_exit))
    printf("abnormal termination, signal number = %d\n", WTERMSIG(child_exit));

exit(0);
```

The parent send a signal that cannot be ignored or caught.

```
hfani@alpha:~$ vi parent_child_ignore.c
```

```
hfani@alpha:~$ ./parent_child_ignore
```

```
I am a lonely process, pid=1978948
```

```
I am the parent, pid=1978948
```

```
I sleep for 10 second and will be back with you to check on your progress, child.
```

```
I am the child, pid=1978949
```

```
I sleep for another 10 second. Hopefully you're done by then.
```

```
I received a signal from my parrent 1978948
```

```
The signal is 10
```

```
I don't care about your signal, ma! I simply ignore it
```

```
What's up child.
```

```
I received a signal from my parrent 1978948
```

```
The signal is 10
```

```
I don't care about your signal, ma! I simply ignore it
```

```
Seems you don't care. I'll kill you
```

```
I received a signal from my parrent 1978948
```

```
!abnormal termination, signal number = 2
```

# What if the child catches & ignores SIGINT?

What if a process catches & ignores CTRL+C or CTRL+Z?


```
#include <signal.h>
typedef void func(int);
void terminal_sig_handler(int signo){}
void main() {
    void *res = signal(SIGINT, terminal_sig_handler);
    res = signal(SIGTSTP, terminal_sig_handler);
    while(1) {}
}
```

The background of the slide is a deep space image showing numerous galaxies in various colors (blue, orange, white) against a black sky. A solid blue horizontal line spans the width of the slide, positioned above the text.

# Signals

Software Shocks: Urgent Communications

I'll send you a signal, if you don't do anything about it, I'll kill you!

A solid blue horizontal line spans the width of the slide, positioned below the text.



---

# Inter-Process Communication (IPC)

Normal Communication

Can you do this for me? Yes, here is it. Anything else?

---

Single Processor ~~Multiprocessor~~