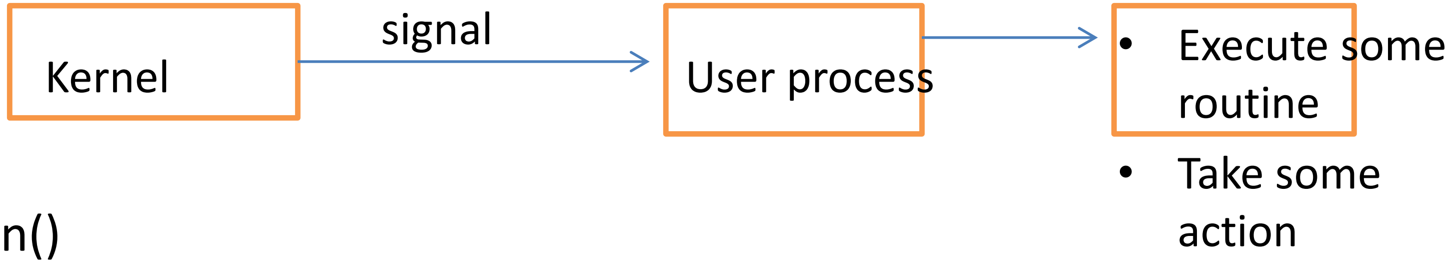
Signal

Signals

- Interprocess communication primitive



Kernel

signal

User process

- Execute some routine
- Take some action

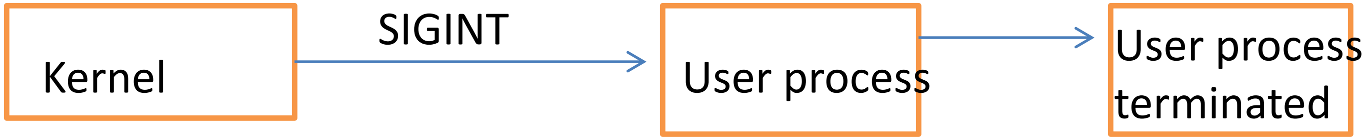
```
Main()
{
        for(;;);
}
```

How to terminate this infinite loop?

Press Ctrl-C

Exactly what happened?

- The process is running
- You pressed Ctrl-C.
- Kernel sends a signal SIGINT to the process (process group)
- Process stopped working
- Kernel executes a routine to terminate the process



Kernel

SIGINT

User process

User process terminated

Signal is like a software interrupt

- Each signal has an interrupt number
- With each signal, a routine is associated to perform some task

Signals

- SIGINT
  - The SIGINT signal is sent to a process by its controlling terminal when a user wishes to interrupt the process. This is typically initiated by pressing Control-C
- SIGKILL
  - The SIGKILL signal is sent to a process to cause it to terminate immediately (**kill**). This signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal.
- SIGQUIT
  - The SIGQUIT signal is sent to a process by its controlling terminal when the user requests that the process **quit** and perform a core dump.
- SIGFPE
  - The SIGFPE signal is sent to a process when it executes an erroneous arithmetic operation, such as division by zero (the FPE stands for **floating point error**)
- SIGSEGV
  - The SIGSEGV signal is sent to a process when it makes an invalid virtual memory reference, or segmentation fault, i.e. when it performs a ***seg*****mentation *v*iolation**
- SIGCHLD
  - The SIGCHLD signal is sent to a process when a **child process** terminates, is interrupted, or resumes after being interrupted.

Other signals

- SIGSEGV
  - Segmentation fault-core dumped
- SIGFPE
  - Division by zero
- SIGTSTP and SIGCONT

Signal Handling

- Each signal has a default code attached
  - Activated whenever the signal is sent
- Is it possibly to replace this default code?
  - Signal handling

  Signal(Signal name, function name)

  Signal.h

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

void abc();
int main()
{
        signal(SIGINT,abc);
        for(;;);

}

void abc()
{
        printf(“You have pressed Ctrl-C\n");
}
```

- Ctrl-C terminates user process
- Doesn’t terminate shell
  - Shell is also a process!
- Ignore a signal!
- Signal(SIGINT,SIG_IGN)

```
int main()
{
		signal(SIGINT,SIG_IGN);
		for(;;);


}
```

SIGQUIT

- **Press Ctrl-**
- **Terminates a process and dump the core**

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

void abc(int);
int main()
{
                signal(SIGINT,abc);
                signal(SIGQUIT,abc);
                for(;;);

}

void abc(int signo)
{
                printf(“You have killed the process with signal ID=%d”,signo\n");
}
```

SIGCLD

- A process sends SIGCLD to its parent after termination
- When a user process X terminates
  - Send this signal to it’s parent (shell)
  - Shell removes the process X from the Process Table
- Not? Then Zombie!
  - Role of wait()

SIGCLD

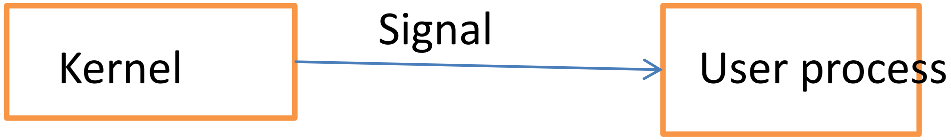
```
int main()
{
        pid=fork();
        if(pid==0)
                sleep(1);
        else
        {
                signal(SIGCLD, abc);
                sleep(10);
                printf(“Parent exiting”);
        }
}
Void abc()
{
        printf(“child died”);
}
```

Other signals

- SIGSEGV
  - Segmentation fault-core dumped
- SIGFPE
  - Division by zero
- SIGTSTP (CRL-Z) and SIGCONT

Sending signal

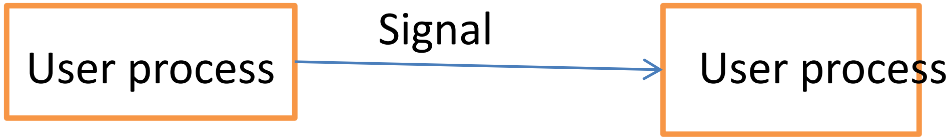
So far, kernel process sends signal to user process


Kernel

Signal

User process

How user process can send signal to another user process?


User process

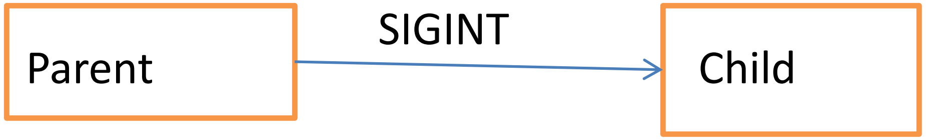
Signal

User process

Kill(process ID, signal ID)

```
int main()
{
        pid=fork();
        if(pid==0)
        {
                signal(SIGINT,abc);
                sleep(2)

        }
        else
        {
                sleep(1);
                kill(pid,SIGINT)
                sleep(10);
                printf(“Parent exiting”);
        }
}
void abc()
{
        printf(“Signal received by child ”);
}
```



Parent

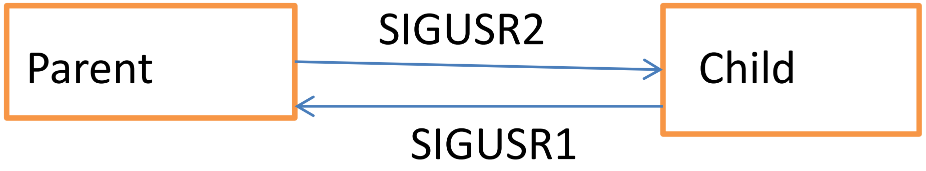
SIGINT

Child

Open signals

```
int main()
{
        pid=fork();
        if(pid==0)
        {
                signal(SIGUSR2, abc);
                sleep(1);
                printf("Hello parent!");
                kill(getppid(),SIGUSR1);
                sleep(4);
        }
        else
        {
                signal(SIGUSR1,def);
                sleep(5);
        }
}
void abc()
{       sleep(2);
        printf("Bye  Parent ");
}
```

- SIGUSR1 and SIGUSR2
- Are not mapped to any event



```
Void def()
{
        printf("Hello child");
        kill(pid,SIGUSR2);
}
```

Process group

Every process is member of a unique process group, identified by its **process group ID**. (When the **child process** is created, it becomes a member of the process group of its **parent**.)

By convention, the **process group ID** of a process group equals the **process ID** of the **first member** of the process group, called the **process group leader**.

A process finds the ID of its process group using the system call **getpgrp()**, or, equivalently, **getpgid(0)**.

One finds the process group ID of process p using **getpgid(p)**.

One may use the command **ps -j** to see PPID (parent process ID), PID (process ID), PGID (process group ID) of processes.

Creation of group

A process pid is put into the process group pgid by

setpgid(pid, pgid);

If pgid == pid or pgid == 0 then this creates a **new process group** with process **group leader pid**.

Otherwise, this puts pid into the already existing process group pgid.

A **zero pid** refers to the **current process**. The call setpgrp() is equivalent to setpgid(0,0).

Restrictions on setpgid()

The calling process must be pid itself, or its parent,

Typical sequence

```
p = fork();
if (p == (pid_t) -1) {
        /* ERROR */
} else if (p == 0) {    /* CHILD */
        setpgid(0, pgid);
        ...
} else {                /* PARENT */
        setpgid(p, pgid);
        ...
}
```