Systems Software/Programming

Inter-Process Communication

Inter-Process Communications using Signals

What is a signal?

- Program must sometimes deal with unexpected or unpredictable events, such as :
  - a floating point error
  - a power failure
  - an alarm clock “ring”
  - the death of a child process
  - a termination request from a user (i.e., Control+C)
  - a suspend request from a user (i.e., Control+Z)
- These kind of events are sometimes called interrupts, as they must interrupt the regular flow of a program in order to be processed.
- When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal.

Therefore a signal is:

- a software interrupt delivered to a process by the OS because:
  - it did something (segmentation fault, FPE)
  - the user did something (pressed Ctrl+C)
  - another process wants to tell it something (SIGUSR?)
- one way a process can communicate with other processes
- Signal Types:
  - Some signals are asynchronous, they may be raised at any time (user pressing Ctrl+C)
  - Some signals are directly related to hardware/process code (illegal instruction, arithmetic exception, such as attempt to divide by 0) - synchronous signals
- Signals functions are defined in signal.h and signal numbers are defined in signal.h or signum.h (signum.h is used inside signal.h if used)

Actions when Signal is received

- The default handler usually performs one of the following actions:
  - Abort: terminate the process and generate a core file (dump)
  - Quit/Exit: terminate the process without generating a core image file (quit)
  - Ignore: ignore and discard the signal (ignore)
  - Stop: suspend the process (suspend)
  - Continue: resume the process (resume)

List of Signals (Linux)

Macro	Signal Code	Default Action	Description
SIGHUP	1	Quit	Hang up - sent to a process when its controlling terminal has disconnected
SIGINT	2	Quit	Interrupt – Ctrl+C pressed by user, terminate the process after saving the work, Signal can be trapped
SIGQUIT	3	Dump	Quit
SIGILL	4	Dump	Illegal instruction (bad opcode)
SIGTRAP	5	Dump	Trace trap (used by debuggers)
SIGABRT	6	Dump	Abort process – Ctrl+\ pressed by user, terminate immediately
SIGBUS	7	Dump	bus error (bad format address or unaligned memory access)
SIGFPE	8	Dump	Floating Point (Arithmetic) execution bad argument
SIGKILL	9	Quit	shell command Kill –SIGKILL or kill -9 or system call kill() to send SIGKILL signal by another process (unblockable)

List of Signals (BSD)

Macro	Signal Code	Default Action	Description
SIGUSR1	10	Quit	user signal 1
SIGSEGV	11	Dump	segmentation violation (out-of-range address)
SIGUSR2	12	Quit	user signal 2
SIGPIPE	13	Quit	write on a pipe or other socket with no one to read it.
SIGALRM	14	Quit	alarm clock timeout
SIGTERM	15	Quit	software termination signal(default signal sent by kill)
SIGSTKFLT	16	Quit	Stack fault
SIGCHLD	17	Ignore	child status changed
SIGCONT	18	Ignore	continued after suspension
SIGSTOP	19	Quit	Suspend process by signal (unblockable)
SIGTSTP	20	Quit	Keyboard store, Stopped by user (Ctrl+Z pressed which suspend/pause the process)

List of Signals (BSD)

Macro	Signal Code	Default Action	Description
SIGTTIN	21	Quit	Background read from tty
SIGTTOU	22	Quit	Background write from tty output
SIGURG	23	Ignore	Urgent condition on socket
SIGXCPU	24	Dump	CPU time limit exceeded
SIGXFSZ	25	Dump	file size limit exceeded
SIGVTALRM	26	Quit	virtual timer expired
SIGPROF	27	Quit	profiling timer expired
SIGWINCH	28	Ignore	window size change
SIGIO	29	Ignore	I/O is now possible
SIGINFO	29	Ignore	Status request from keyboard
SIGPWR	30	Quit	Power failure restart
SIGSYS	31	Dump	bad argument to system call or non-existence system call

Sending signals from keyboard

- Process suspended using Ctrl+Z (SIGTSTP) can be brought back to life using fg command which sends (SIGCONT) signal to resume
- For background process cant use Ctrl+C, Ctrl+Z etc hence kill command is used (how one can run process in background ?)

kill [options] pid

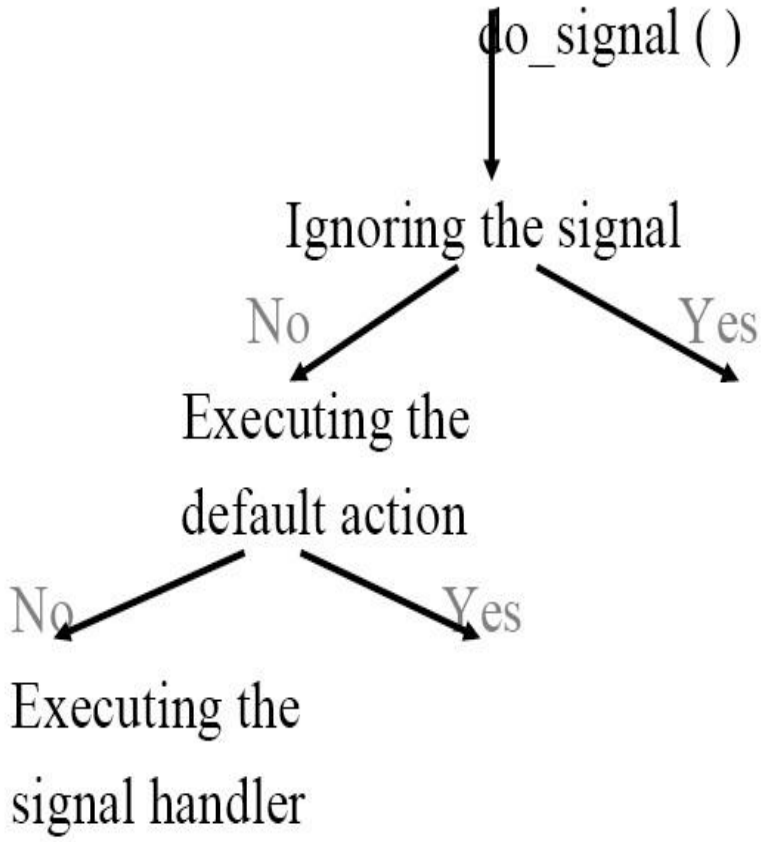
--l lists all signals you can send

--signal number

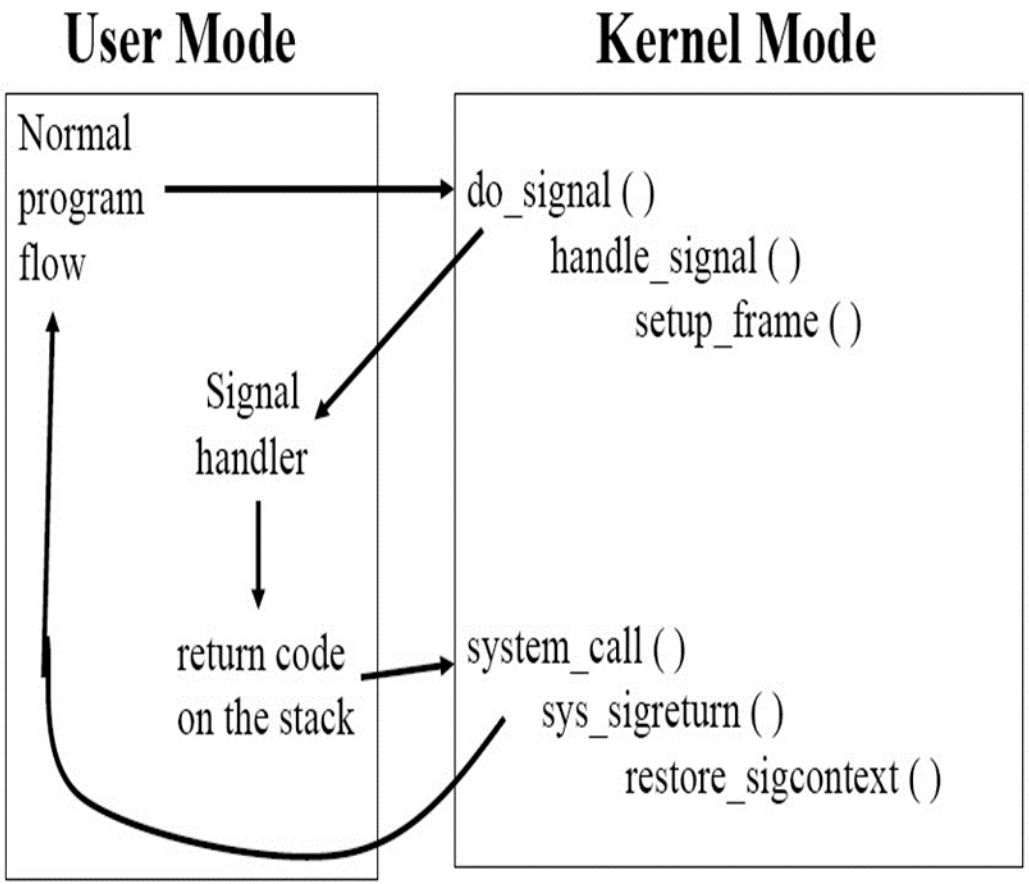
kill command can send SIGKILL signal upon pressing Ctrl+C to the foreground process which cannot be ignored

Signal Handling

- Most signals can be ignored except SIGKILL and SIGSTOP
- Application can trap the signal (e.g. SIGINT when Ctrl+C pressed) and handle signal disposition via signal handler
- Signal handler can perform default action (usually process termination) or customized actions such as blocking the signal



Catching Signal



Requesting an Alarm Signal SIGALRM using alarm() System Call

System Call : unsigned int alarm(unsigned int count)

- alarm() instructs the kernel to send the SIGALRM signal to the calling process after counting seconds.
  - If an alarm had already been scheduled, than an alarm is overwritten.
  - If count is 0, any pending alarm requests are cancelled.
- The default handler for this signal displays the message “Alarm clock” and terminates the process.
- alarm() returns the number of seconds that remain until the alarm signal is sent

System Call alarm() example

- Set Alarm for 3 second with default action handler which will display a default message “Alarm clock” and exit the program Signals\alarm_test.c

Handling Signals (Overriding Default Action)

System Call: void(*signal(int sigCode, void (*func)(int))) (int)

- signal() allows a process to specify the action that it will take when a particular signal is received.
  - sigCode specifies the signal number (as per the table shown in earlier slides) that is to be reprogrammed
  - func may be one of several values:
    - SIG_IGN, which indicates that the specified signal should be ignored and discarded (i.e. No Action to be taken)
    - SIG_DFL, which indicates that the kernel's default handler should be used.
    - an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

Handling Signals

- The valid signal numbers are stored in /usr/include/signal.h or /usr/include/bits/signum.h
  - The signals SIGKILL and SIGSTP may not be reprogrammed.
- signal() returns the previous func value associated with sigCode if successful; otherwise, it returns a value of -1
- signal() system call can be used to override the default action
- A child process inherits the signal settings from its parent during a fork()

System Call pause()

System Call: int pause(void)

- pause() suspends the calling process and returns when a calling process receives a signal.
- It is most often used to wait efficiently for a signal.
- pause() doesn’t return anything useful.

Difference between sleep(int) and pause:

pause() suspends process for indefinite time whereas sleep(int secs) suspends processes for specified number of seconds passed as parameter secs.

Catch SIGALRM of alarm() system call using signal() system call

- Write your own handler to handle alarm signal SIGALRM
- Override default action using signal() system call
- Signals\alarm_override.c

Protecting Critical Code from Ctrl-C attack

- Overriding may be used to protect critical pieces of code against Control-C attacks and other such signals.
- it can be restored after the critical code has executed.
- Here’s the source code of a program that protects itself against SIGINT signals Signals\ctrl_c_override.c

Process Groups

- When you execute a program in terminal window (shell) that creates several children, a single Control-C from the keyboard will normally terminate the foreground parent process and all its children processes and then return to the shell.
- In order to support this kind of behavior, UNIX introduced a few new concepts.
  - In addition to having a unique process ID number, every process is also a member of a process group. Each process has process id (pid) and process group id (pgid).
    - Several processes can be members of the same process group (pgid).
    - When a process forks, the child inherits its process group from its parent.
    - A process may change its process group to a new value by using setpgid()
    - When a process runs exec???() system call, its process group remains the same.

Process Groups and Control Terminals (i.e. Which processes have control of the terminal windows)

- Every process can have an associated control terminal, which is typically the terminal where the process was started.
  - When a process forks, the child inherits its control terminal from its parent.
  - When a process execs, its control terminal stays the same.
- Every terminal can be associated with a single control process.
  - When a metacharacter such as a Control-C is detected, the terminal sends the appropriate signal to all of the processes in the process group of its control process.
- If a process attempts to read from its control terminal and is not a member of the same process group as the terminal's control process,
  - the process is sent a SIGTTIN signal, which normally suspends the process.

Process Groups and Control Terminals (i.e. Which processes have control of the terminal windows)

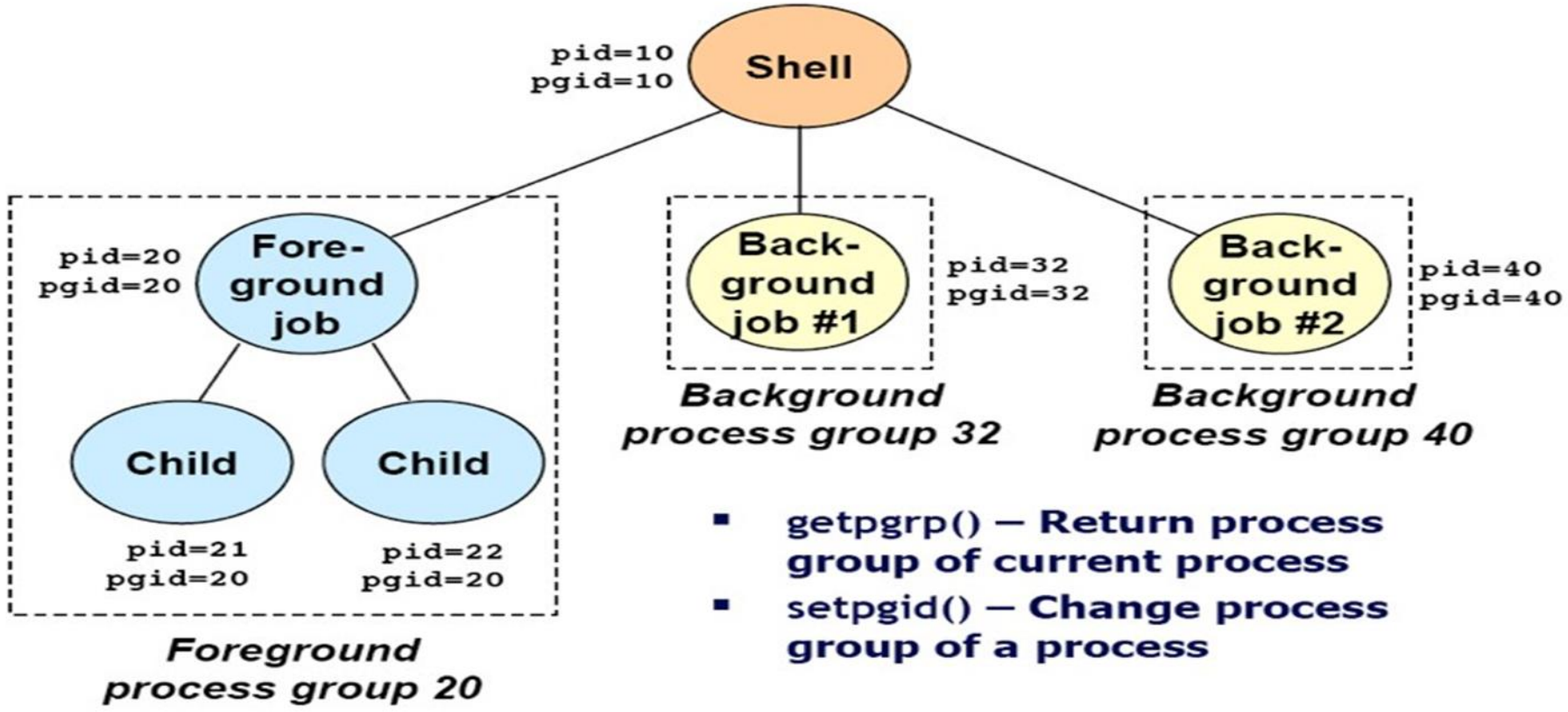
- When an interactive shell begins (i.e. terminal window), it is the control process of a terminal and has that terminal as its control terminal.
- When a shell executes a foreground process:
  - the child shell places itself in a different process group before exec'ing the command and takes control of the terminal by executing "int tcsetpgrp(int fd, pid_t pgrp)" system call
  - Any signals generated from the terminal thus go to the foreground command rather than to the original parent shell.
  - When the foreground command terminates, the original parent shell takes back control of the terminal again using tcsetpgrp system call

Process Groups and Control Terminals Signals\process_group_background.c

- When a shell executes a background process:
  - the child shell places itself in a different process group before executing, but does not take control of the terminal.
  - Any signals generated from the terminal continue to go to the shell.
  - If the background process tries to read from its control terminal, it is suspended by a SIGTTIN signal.

Process Groups and Control Terminals

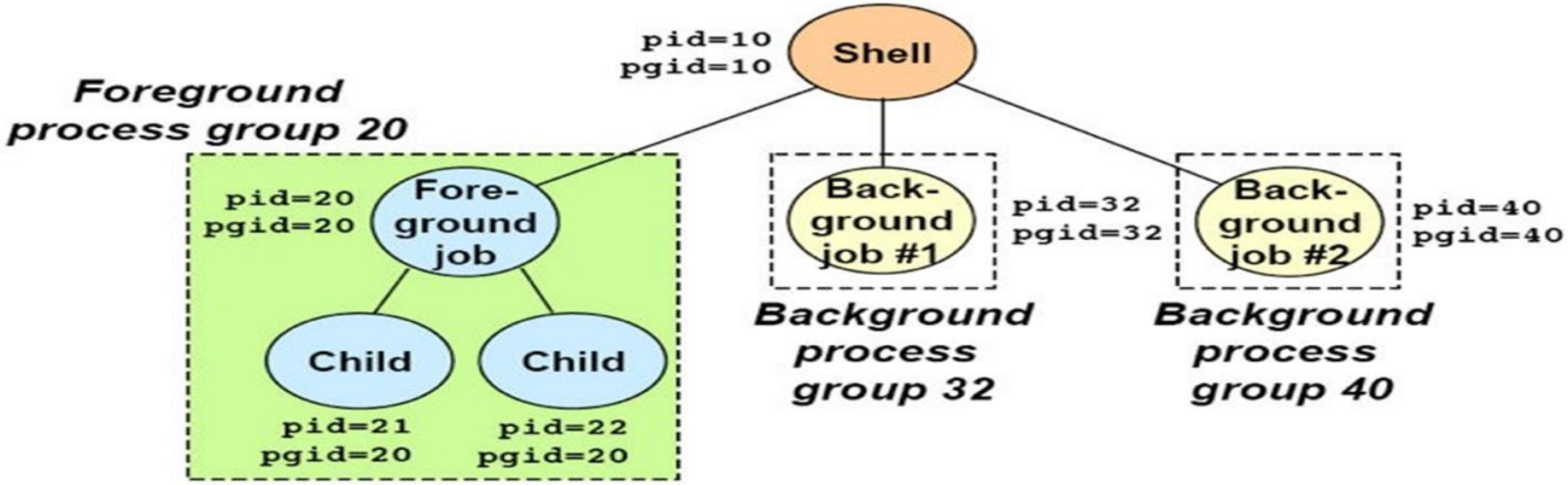
- **Every process belongs to exactly one process group.**



- **getpgrp() – Return process group of current process**
- **setpgid() – Change process group of a process**

Process Groups and Control Terminals

- **Sending signals from the keyboard**
  - Typing **ctrl-c** (**ctrl-z**) sends a **SIGINT** (**SIGTSTP**) to every job in the foreground process group.
    - SIGINT: default action is to terminate each process.
    - SIGTSTP: default action is to stop (suspend) each process.



System Call setpgid() and getpgid()

System Call : pid_t setpgid(pid_t pid, pid_t pgrpId)

- setpgid() sets the process-group ID of the process with PID pid to pgrpId.
  - if pid is zero, the caller's process group ID is set to pgrpId.
  - In order for setpgid() to succeed and set the process group ID, at least one of the following conditions must be met:
    - The caller and the specified process must have the same owner.
    - The caller must be owned by a super-user.
- If setpgid() fails, it returns a value of -1.

System Call: pid_t getpgid(pid_t pid)

- getpgid()" returns the process group ID of the process with PID pid.
  - If pid is zero, the process group ID of the caller is returned.

Parent and Child in Same Process Groups, both gets the Signal

- example illustrates the fact that a terminal distributes signals to all of the processes in its control process' process group.
- Since the child inherited its process group from its parent, both the parent and child caught the SIGINT signal.

Signals\process_group_same.c

$ ./process_group_same.out parent pid 19354 and parent group 19354 waits

child pid 19355 and child group 19354 waits

^C

process 19355 got a SIGINT

process 19354 got a SIGINT

$

Parent and Child in Different Process Groups, Only one gets the Signal

- example illustrates the fact that a terminal distributes signals to all of the processes in its control process' process group.
- Since the child is in different process group SIGINT signal was caught only by parent and not child. We had to explicitly kill child since it was orphan.

Signals\process_group_diff.c

```
$ ./process_group_diff.out
parent pid 19460 and parent group 19460 waits
child pid 19461 and child group 19461 waits
^C
process 19460 got a SIGINT
$ ps 19461
PID TTY      STAT   TIME COMMAND
19461 pts/1    S      0:00 ./process_group_diff.out
$ kill 19461
$ ps 19461
PID TTY      STAT   TIME COMMAND
$
```

How parent can get additional info about child terminated or stopped by signal?

childpid = wait(&wstatus); → returns the exit status from child which can further be inspected using these macros

WIFEXITED(wstatus) → returns true if child terminated normally

WEXITSTATUS(wstatus) → returns exit status (least significant 8 bits)

WIFSIGNALED(wstatus) → returns true if child process was terminated by a signal

WTERMSIG(wstatus) → returns the number of signal

WCOREDUMP(wstatus) → returns true if child produced a core dump

WIFSTOPPED(wstatus) → returns true if child was stopped by a signal

WSTOPSIG(wstatus) → returns the signal number which caused child to stop

WIFCONTINUED(wstatus) → returns true if child was resumed with SIGCONT signal

Sending Signal from another Process: System Call kill()

- A process may send a signal to another process by using the kill() system call
- kill() is a misnomer, since many of the signals that it can send to do not terminate a process
- Its call kill() because it was originally designed to terminate a process which is still one of its function

System Call kill()

System Call: int kill(pid_t pid, int sigCode)

- kill() sends the signal with value sigCode to the process with PID pid.
- kill() succeeds and the signal is sent as long as at least one of the following conditions is satisfied:
  - The sending process and the receiving process have the same owner.
  - The sending process is owned by a super-user.
- There are a few variations on the way that “kill()” works:
  - If pid is positive, then *signal* is sent to the process with the ID specified by *pid*
  - If the pid is negative, but not -1, then *signal* is sent to every process in the process group whose ID is *-pid*.
  - If pid is zero, then *signal* is sent to every process in the same process group as of the calling process.
  - If pid is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.
  - If pid is -1 and the sender is not owned by a super-user, the signal is sent to all of the processes owned by the same owner as that of the sender, excluding the sending process.

Detecting death of a Child Process

- When a child terminates, the OS sends parent of the terminated child a SIGCHLD signal.
- A parent process often installs a handler to deal with this signal; this handler typically executes a wait() system call to accept the child's termination code and let the child dezombify.
- the parent can choose to ignore SIGCHLD signals.
- The program works by performing the following steps:
  - The parent process installs a SIGCHLD handler that is executed when its child process terminates.
  - The parent process forks a child process to execute the command.
  - The parent process goes on and do other activities.
  - If the child terminates before its parent finishes, the parent's SIGCHLD handler is executed for that child.
  - Note: If parent has multiple children for each child SIGCHLD handler will be executed upon termination the child.

Master-Slave Implementation with SIGCHLD

- In earlier parent and child processes which implemented master-slave relationship, parent was not doing anything except waiting for child process, hence child was not zambified
- But if parent process is also performing some tasks and has to make sure that child is not zambified, it should override SIGCHLD signal and execute wait system call to dezambify child
- Signals\process_parent_sigchld.c
- Signals\process_child_sigchld.c

Detecting death of a Child Process

Signals\child_death_detection.c

```
#include <stdio.h>
#include <signal.h>
int delay;
void childHandler();
int main( int argc, char* argv[] )
{
   int pid;
   signal( SIGCHLD, childHandler );  /* Install death-of-child handler */
   pid = fork();  /* Duplicate */
   if ( pid == 0 )  /* Child */
   {
        execvp( argv[2], &argv[2] );   /* Execute command */
        perror("limit");   /* Should never execute */
   }
else   /* Parent */
 {
  sscanf( argv[1], "%d", &delay );  /* Read delay from command-line */
  sleep(delay);   /* Sleep for the specified number of seconds */
  printf("Child %d exceeded limit and is being killed \n", pid );
  kill( pid, SIGINT );   /* Kill the child */
 }
}
```

Detecting death of a Child Process

```
void childHandler()  /* Executed if the child dies before the parent */
{
  int  childPid, childStatus;
  childPid = wait(&childStatus);   /* Accept child's termination code */
  printf("Child %d terminated within %d seconds \n", childPid, delay);
  exit(/* EXIT SUCCESS */  0);
}
```

```
$ limit  5  ls   ---> run the program; command finishes OK.
a.out        alarm          critical        handler        limit
alarm.c      critical.c       handler.c     limit.c
Child 4030  terminated within 5 seconds.

$ limit  4  sleep  100  ---> run it again;  command takes too long
Child 4032 exceeded limit and is being killed
```

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

- The SIGSTOP and SIGCONT signals suspend and resume a process, respectively.
- They are used by the UNIX shells that support job control to implement built-in commands like stop, fg, and bg.
- Program in next slide performs these steps:
  - The main program creates two children; they both enter an infinite loop and display a message every second.
  - The main program waits for three seconds and then suspend the first child.
  - After another three seconds, the parent restarts the first child, waits a little while longer, and then terminated both children.

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

Signals\sigstop_sigcont_example.c

```
#include <signal.h>
#include <stdio.h>
int main(void)
{ int pid1;
  int pid2;
  pid1 = fork();
  if (pid1== 0) /* First child */
   {
     while(1)  /* Infinite loop */
      {
        printf("pid1 is alive \n");
        sleep(1);
      }
   }
  pid2 = fork();  /* Second child */
  if ( pid2 == 0 )
    {
      while(1)   /* Infinite loop */
       {
         printf("pid2 is alive \n");
         sleep(1);
       }
    }
  sleep(3);
  kill( pid1, SIGSTOP );   /* Suspend first child */
  sleep(3);
  kill( pid1, SIGCONT );  /* Resume first child */
  sleep(3);
  kill( pid1, SIGINT );     /* Kill first child */
  kill( pid2, SIGINT );     /* Kill second child */
}
```

Suspending and Resuming Process using SIGSTOP and SIGCONT signals

- $ pulse ---> run the program.
- pid1 is alive ---> both run in first three seconds.
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid2 is alive ---> just the second child runs now since first child is suspended
- pid2 is alive
- pid2 is alive
- pid1 is alive ---> the first child is resumed.
- pid2 is alive
- pid1 is alive
- pid2 is alive
- pid1 is alive
- pid2 is alive

SIGUSR1 and SIGUSR2 → User Defined Signals

- Developer can use SIGUSR1 and/or SIGUSR2 to create own signals
- Steps for implementing SIGUSR1/2:
  - Process designated as recipient of SIGUSR1/2:
    - First create a signal handler function → void my_signal_handler(int signum)
      - signum will be passed as SIGUSR1 or SIGUSR2 depending on which signal was sent
    - Register signal handler function for each signal so that it can be called upon receipt of that signal → signal(SIGUSR1, my_signal_handler)
  - Process designated for sending signals SIGUSR1/2:
    - Uses kill() system call to generate/send signal to a specific process using pid → kill(pid1, SIGUSR1)
    - As we studied earlier kill() system call is misnomer and you can use it to send any signal to a process, in this case it is sending SIGUSR1 signal to pid1

Synchronize the common activity between all children

- At a particular time interval parent and child all need to do certain activity
- Parent can set alarm for a specified time and upon receipt of the SIGALRM it sends SIGUSR1 to the group
- Signals\siguser_sync_example.c

Synchronization between two processes

- Inter Process Communication between parent and child using SIGUSR1 and SIGUSR2
- This example shows
- Register signal handler for SIGUSR1 and SIGUSR2 for parent process which inherited by child process because they belong to the same process group
- Child first sends SIGUSR1 to parent
- Parent upon receiving SIGUSR1 sends SIGUSR1 to child
- Child upon receiving SIGUSR1 sends SIGUSR2 to parent
- Parent upon receiving SIGUSR2 sends SIGUSR2 to child
- Child upon receiving SIGUSR2 terminates itself by sending SIGINT signal to itself
- Parent when detect child has died it termininates
- Signals\siguser_test.c

Ignore Other Signals inside Signal Handler

Signals\ignore_signls_inside_signalhandler.c

```
int main(int ac, char *av[])
{
        void inthandler(int);
        void quithandler(int);
        char input[100];
        signal( SIGINT, inthandler ); // set trap
        signal( SIGQUIT, quithandler ); // set trap
        int i=1;
        while (i<5) {
                sleep(1);
                printf("main:%d\n",i++);
        }
}
```

```
void inthandler(int s)
{
        printf(" Received SIGINT signal %d .. waiting\n", s );
        printf(" Leaving inthandler \n");
}

void quithandler(int s)
{
        signal(SIGINT, SIG_IGN);
        printf(" Received SIGQUIT signal %d .. waiting\n", s );
        sleep(5);
        printf(" Leaving quithandler \n");
        signal( SIGINT, inthandler );
}
```

System Call: sigaction()

System Call : int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)

- It is more advance system call then signal()
- It uses struct sigaction for additional controls

struct sigaction {

void (*sa_handler)(int); →SIG_DFL or SIG_IGN or pointer to signal handler

void(*sa_sigaction)(int, siginfo_t *, void *); → Used only of SA_SIGINFO value in sa_flags, passes additional information

sigset_t sa_mask; → set of additional signals to be blocked during execution of sa_handler function

int sa_flags; → special flags affecting signal behavior e.g. SA_SIGINFO value in this flag indicates handler is sa_sigaction rather than sa_handler

void (*sa_restore)(void); → used for POSIX real time signal only not used by application

}

Manipulating Signal Sets

sa_mask of sigaction structure has set of signal to block.

Set can be modified using these functions:

sigemptyset(sigset_t *set) →init to no signals

sigfillset(sigset_t *set) → init to all signals

sigaddset(sigset_t *set, int signum) → add signal to set

sigdelset(sigset_t *set, int signum) → remove signal from set

sigismember(sigset_t *set, int signum) → check if signal is member of set

System Call : sigprocmask()

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

oldset: oldset is set with currently block set of signals before making any changes (this will be used to restore the block state before the call to sigprocmask)

set : set of signals that will follow action of “how”

how:

SIG_BLOCK: add set of signals in “set” to current blocked signal set

SIG_UNBLOCK: remove set of signals in “set” from current blocked signal set

SIG_SETMASK: set of blocked signal is set to “set”

Block Other Signal inside Signal Handler using segprocmask()

```
void inthandler(int s)
{
sigset_t sigset, sigoldset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGQUIT);
sigprocmask(SIG_SETMASK, &sigset,
&sigoldset);
printf(" Received signal %d .. waiting\n", s );
....
printf(" Leaving SIGINT Handler \n");
sigprocmask(SIG_SETMASK, &sigoldset, NULL);
}
```

```
int main(int ac, char *av[])
{
        void inthandler(int);

        signal( SIGINT, inthandler ); // set trap

        int i=1;
        while (i<5) {
                sleep(1);
                printf("main:%d\n",i++);
        }
}
```

sigaction() example

```
main()
{
struct sigaction newhandler; /* new settings */
sigset_t blocked; /* set of blocked sigs */
void inthandler(); /* the handler */
newhandler.sa_handler = inthandler; /* handler function */
sigfillset(&blocked); /* mask all signals */
newhandler.sa_mask = blocked; /* store blockmask */
int i;
for (i=1; i<31;i++)
        if (i!=9 && i!=17) /* catch all except these signals */
                if ( sigaction(i, &newhandler, NULL) == -1 )
                        printf("error with signal %d\n", i);
while(1){}
}
```