Process Groups

- Every process belongs to exactly one process group.

- getpgid() – Return process group of current process
- setpgid() – Change process group of a process
Sending Signals

- **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.

![Diagram](image)

Signals from Keyboard

The most common way of sending signals to processes is using the keyboard:

- **Ctrl-C**: Causes the system to send an INT signal (SIGINT) to the running process.
- **Ctrl-Z**: causes the system to send a TSTOP signal (SIGTSTP) to the running process.
- **Ctrl-\**: causes the system to send a ABRT signal (SIGABRT) to the running process.
Signals from Command-Line

- The `kill` command has the following format:

  \[ \text{kill } [\text{options}] \text{ pid} \]

  - \(\text{--}\text{-}l\) lists all the signals you can send
  - \(\text{--}\text{-}\text{signal}\) is a signal number
  - the default is to send a \text{T}ERM signal to the process.

- The `fg` command will resume execution of the process (that was suspended with Ctrl-Z), by sending it a \text{C}ONT signal.

  ```
  $ \text{kill 10231} \quad // \text{SIGTERM : default signal}
  $ \text{kill -9 10231} \quad // \text{SIGKILL}
  ```

Signals from a Process

- `int kill(pid_t pid, int sig)`
  - Can be used to send any signal to any process group or process.
    - \(\text{pid} > 0\), signal \(\text{sig}\) is sent to \(\text{pid}\).
    - \(\text{pid} = 0\), \(\text{sig}\) is sent to every process in the process group of the current process.
    - \(\text{pid} = -1\), \(\text{sig}\) is sent to every process except for process 1.
    - \(\text{pid} < -1\), \(\text{sig}\) is sent to every process in the process group \(-\text{pid}\).
    - \(\text{sig} = 0\), no signal is sent, but error checking is performed.

- `raise(signo)` causes the specified signal to be sent to the process that executes the call to `raise`.
void fork12(int N) {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++) {
        pid[i] = fork();
        if (pid[i] == 0) {
            if (i==2) signal(SIGINT, SIG_IGN);
            while(1); /* Child infinite loop */
        } else
            if (pid[i]>0) printf("Child process %d is created.
", pid[i]);
    }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d..\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}

Catching the Signal

User Mode

Normal program flow

Signal handler

return code on the stack

Kernel Mode

do_signal ()

handle_signal ()

setup_frame ()

system_call ()

sys_sigreturn ()

restore_sigcontext ()
**Actions on Signal**

- `no_signal()`
  - Ignoring the signal
    - No → Executing the default action
      - No → Executing the signal handler
    - Yes →

**Non-Catchable Signals**

- Most signals may be caught by the process, but there are a few signals that the process cannot catch, and cause the process to terminate.
  - For example: **KILL** and **STOP**.
- If you install no signal handlers of your own the runtime environment sets up a set of default signal handlers.
  - For example:
    - The default signal handler for the **TERM** signal calls the `exit()`.
    - The default handler for the **ABRT** is to dump the process's memory image into a file, and then exit.
Default Actions

- **Abort** – terminate the process after generating a dump
- **Exit** – terminate the process without generating a dump
- **Ignore** – the signal is ignored
- **Stop** – suspends the process
- **Continue** – resumes the process, if suspended

Signal Semantics

- A signal is **pending** if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Signals are not queued!
- A process can **block** the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
  - There is one signal that can not be blocked by the process. (SIGKILL)
- A pending signal is received at most once.
  - Kernel uses a bit vector for indicating pending signals.
Implementation

- Kernel maintains **pending** and **blocked** bit vectors in the context of each process.
  - **pending** – represents the set of pending signals
    » Kernel sets bit k in **pending** whenever a signal of type k is delivered.
    » Kernel clears bit k in **pending** whenever a signal of type k is received.
  - **blocked** – represents the set of blocked signals
    » Can be set and cleared by the application using the `sigprocmask` function.

Receiving Signals

- **Handling signals**
  - Suppose kernel is returning from exception handler and is ready to pass control to process p.

  - Kernel computes \( \text{pnb} = \text{pending} \& \sim \text{blocked} \)
    – The set of pending nonblocked signals for process p
  - if (pnb != 0) {
    – Choose least nonzero bit k in pnb and force process p to receive signal k.
    – The receipt of the signal triggers some action by p.
    – Repeat for all nonzero k in pnb.
  }
  - Pass control to next instruction in the logical flow for p.
Overlapping Signals

- SIGY interrupts SIGX
  - ex: phone then door
  - When you press CTRL-C then CTRL-\, the program first jumps to inthandler, then to quithandler, then back to inthandler, then back to main loop.
- SIGX interrupts SIGX
  - ex: two people coming to your door
  - Three ways this can be handled:
    1. Recursively call the same handler
    2. Ignore the second signal, like a phone without call waiting
    3. Block the second signal until done handling the first
- Original systems used method 1, though method 3 is safest.
- Interrupted System Calls
  - receiving a signal while waiting for input

Example from Last Lecture

```c
main(int ac, char *av[])
{
    void    inthandler(int);
    void    quithandler(int);
    char    input[100];

    signal( SIGINT,  inthandler );    //set trap
    signal( SIGQUIT, quithandler );   //set trap

    do {
        printf("\nType a message\n");
        if ( gets(input) == NULL )
            perror("Saw EOF ");
        else
            printf("You typed: %s\n", input);
    }
    while( strcmp( input , "quit" ) != 0 );
}
```
Example from Last Lecture (cont.)

```c
void inthandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(2);
    printf(" Leaving inthandler \n");
}

void quithandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(3);
    printf(" Leaving quithandler \n");
}
```

Ignore other Interrupts inside Handler?

```c
void quithandler(int s)
{
    printf(" Received signal %d .. waiting\n", s );
    sleep(3);
    printf(" Leaving quithandler \n");
}

void inthandler(int s)
{
    signal(SIGQUIT, SIG_IGN);
    printf(" Received signal %d .. waiting\n", s );
    sleep(2);
    printf(" Leaving inthandler \n");
    signal( SIGQUIT, quithandler );
}
```
sigaction() Function

- The `sigaction()` function allows the calling process to examine and/or specify the action to be associated with a specific signal.

```c
int sigaction(int sig,
              struct sigaction *new_act,
              struct sigaction *old_act);
```

sigaction() Function (cont.)

- This function is “newer” than `signal`, and provides considerably more flexibility.
- Like `signal`, the first argument is a signal number (or name).
- The second argument is a pointer to a structure containing the new characteristics for the signal; the third argument points to a structure which will receive the old characteristics of the signal. Either or both of these pointers may be `NULL`, allowing any combination of setting or querying the action associated with a signal.
**sigaction Structure**

- **struct sigaction** has the following members:
  - **sa_handler** – Set to SIG_DFL, SIG_IGN, or pointer to handler function (compare this with the second argument to signal).
  - **sa_mask** – A set of additional signals to be blocked during execution of the function identified by sa_handler.
  - **sa_flags** – Special flags that affect the signal behavior.
  - **sa_sigaction** (used only for POSIX real-time signals).

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

**sa_flags**

- **SA_NOCLDSTOP**: If signum is SIGCHLD, do not receive notification when child processes stop.
- **SA_NOCLDWAIT**: If signum is SIGCHLD, do not transform children into zombies when they terminate.
- **SA_RESETHAND**: Restore the signal action to the default state once the signal handler has been called.
- **SA_ONSTACK**: Call the signal handler on an alternate signal stack provided by sigaltstack(2).
- **SA_RESTART**: Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.
- **SA_NODEFER**: Do not prevent the signal from being received from within its own signal handler.
- **SA_SIGINFO**: The signal handler takes 3 arguments, not one. In this case, sa_sigaction should be set instead of sa_handler.
sigaction() (cont.)

- A new signal mask is calculated and installed only for the duration of the signal-catching function, which includes the signal being delivered.
- Once an action is installed for a specific signal, it remains installed until another action is explicitly requested.

```c
main()
{
    struct sigaction newhandler;
    sigset_t         blocked;
    void             inthandler();
    char             x[INPUTLEN];

    newhandler.sa_handler = inthandler;
    newhandler.sa_flags = SA_RESETHAND | SA_RESTART;

    sigemptyset(&blocked);
    sigaddset(&blocked, SIGQUIT);
    newhandler.sa_mask = blocked;

    if ( sigaction(SIGINT, &newhandler, NULL) == -1 )
        perror("sigaction");
    else
        while( 1 ){
            fgets(x, INPUTLEN, stdin);
            printf("input: %s", x);
        }
}
```
Masking Signals - Avoid Race Conditions

- The occurrence of a second signal while the signal handler function executes.
  - The second signal can be of different type than the one being handled, or even of the same type.
- The system also contains some features that will allow us to block signals from being processed.
  - A global context which affects all signal handlers, or a per-signal type context.

Masking Signals (cont.)

- Each process maintains a signal mask which controls which signals are immediately delivered to the process and which have delivery deferred.
- If a signal is in the signal mask, it is said to be blocked.
- The signal mask for a process is initially empty, which means any signal can be delivered.
- Some signals (in particular, SIGKILL and SIGSTOP) cannot be deferred. Including them in a signal mask is not an error, but will not be effective.
- Blocking a signal is a temporary measure; don’t confuse it with ignoring (with SIG_IGN) a signal.
**sigprocmask() Function**

- The system call allows to specify a set of signals to block, and returns the list of signals that were previously blocked.

```c
sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

1. int how:
   - Add (SIG_BLOCK)
   - Delete (SIG_UNBLOCK)
   - Set (SIG_SETMASK).
2. const sigset_t *set:
   - The set of signals.
3. sigset_t *oldset:
   - If this parameter is not NULL, then it'll contain the previous mask.

---

**Suspending Masked Signals**

`sigsuspend(newmask)`

- This function blocks the calling process until one of the signals not masked in `newmask` is delivered to the process.
- When invoked, it sets the process signal mask to `newmask` and blocks.
- When an unmasked signal arrives, its handler is invoked.
- Then `sigsuspend` returns, always with a value of -1 and `errno` = EINTR.
Manipulating Signal Sets

The `sa_mask` component of the `sigaction` structure contains a set of signals. This set is modified using the following functions (or macros):

- `sigemptyset (sigset_t *set);` -- init to no signals
- `sigfillset (sigset_t *set);` -- init to all signals
- `sigaddset (sigset_t *set, int signo);` -- add signal
- `sigdelset (sigset_t *set, int signo);` -- remove signal
- `sigismember (sigset_t *set, int signo);` -- check signal

Real-time Signals

- POSIX.4 adds some additional signal facilities. The key features are:
  - The real-time signals are in addition to the existing signals, and are in the range `SIGRTMIN` to `SIGRTMAX`.
  - Real-time signals are queued, not just registered (as is done for non real-time signals).
  - The source of a real-time signal (`kill, sigqueue, asynchronous I/O completion, timer expiration, etc.`) is indicated when the signal is delivered.
  - A data value can be delivered with the signal.
Summary

• Signals
  - Generating & Catching Signals
  - Overlapping Signals
  - Preventing Race Conditions
  - Masking Signals

HW 2 out today, due Oct 14th, Tuesday!

Acknowledgments

• Advanced Programming in the Unix Environment by R. Stevens
• The C Programming Language by B. Kernighan and D. Ritchie
• Understanding Unix/Linux Programming by B. Molay
• Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), and J. Kim (KAIST).