Signals

Prof. Euiseong Seo
TA – Hyeonsu Lee(hyeonsu.lee@csl.skku.edu)
TA – Seungmin Oh(ohsm1008@gmail.com)
Computer System Laboratory
Sungkyunkwan University
http://csi.skku.edu
Multitasking

- **Programmer’s model of multitasking**

  - **fork()** spawns new process
    - Called once, returns twice
  
  - **exit()** terminates own process
    - Called once, never returns
    - Puts it into “zombie” status

  - **wait() and waitpid()** wait for and reap terminated children

  - **execve()** runs new program in existing process
    - Called once, never returns
Signal

Definition

- A signal is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID’s.
  - The only information in a signal is its ID and the fact that it arrived.
## Signal list

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from <em>abort</em>(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Term</td>
<td>Broken pipe: write to pipe with no readers</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Term</td>
<td>Timer signal from <em>alarm</em>(2)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,10,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>31,12,17</td>
<td>Term</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>20,17,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,18,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,19,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>18,20,24</td>
<td>Stop</td>
<td>Stop typed at terminal</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>21,21,26</td>
<td>Stop</td>
<td>Terminal input for background process</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22,22,27</td>
<td>Stop</td>
<td>Terminal output for background process</td>
</tr>
</tbody>
</table>

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.
# Signal list

<table>
<thead>
<tr>
<th>번호</th>
<th>시그널</th>
<th>기본처리</th>
<th>발생조건</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>종료</td>
<td>터미널과 연결이 끊어졌을때</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>종료</td>
<td>인터럽트로 ctrl + c 입력시</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>코어 덮프</td>
<td>ctrl + ⌋ 입력시</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>코어 덮프</td>
<td>잘못된 명령 사용</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>코어 덮프</td>
<td>trace, breakpoint에서 TRAP 발생</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>코어 덮프</td>
<td>abort (비정상종료) 함수에 의해 발생</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>종료</td>
<td>강제 종료시</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>코어 덮프</td>
<td>버스 오류시</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>코어 덮프</td>
<td>세그먼테이션 폴트 시</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>코어 덮프</td>
<td>system call 잘못했을때</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>코어 덮프</td>
<td>파일 처리 잘못했을때</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>코어 덮프</td>
<td>알람에 의해 발생함.</td>
</tr>
<tr>
<td>16</td>
<td>SIGUSR1</td>
<td>종료</td>
<td>사용자 정의 시그널1</td>
</tr>
<tr>
<td>17</td>
<td>SIGUSR2</td>
<td>종료</td>
<td>사용자 정의 시그널2</td>
</tr>
<tr>
<td>18</td>
<td>SIGCHLD</td>
<td>무시</td>
<td>자식 프로세스(child process) 상태 변할때</td>
</tr>
<tr>
<td>23</td>
<td>SIGSTOP</td>
<td>중지</td>
<td>이 시그널을 받으면 SIGCONT시그널을 받을때 까지 프로세스 중지</td>
</tr>
<tr>
<td>24</td>
<td>SIGTSTP</td>
<td>중지</td>
<td>ctrl + z 입력시</td>
</tr>
<tr>
<td>25</td>
<td>SIGCONT</td>
<td>무시</td>
<td>중지된 프로세스 실행시</td>
</tr>
<tr>
<td>28</td>
<td>SIGVTALRM</td>
<td>종료</td>
<td>가상 타이머 종료시.</td>
</tr>
</tbody>
</table>

http://blockdmask.tistory.com
Signal Concepts (1)

- Sending a signal
  - Kernel *sends* (delivers) a signal to a destination process by updating some state in the context of the destination process.
  - Kernel sends a signal for one of the following reasons:
    - Generated internally:
      » Divide-by-zero (*SIGFPE*)
      » Termination of a child process (*SIGCHLD*), ...
    - Generated externally:
      » *kill* system call by another process to request signal to the destination process.
Signal Concepts (2)

- Receiving a signal
  - A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Explicitly ignore the signal
    - Execute the default action
    - Catch the signal by invoking signal-handler function
      » Akin to a hardware exception handler being called in response to an asynchronous interrupt.
Signal Concepts (3)

- **Default actions**
  - Abort
    - The process is destroyed
  - Dump
    - The process is destroyed & core dump
  - Ignore
    - The signal is ignored
  - Stop
    - The process is stopped
  - Continue
    - If the process is stopped, it is put into running state
### Signal Concepts Example

- Or you can see it from ‘man 7 signal’

#### Standard signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from abort(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Term</td>
<td>Broken pipe: write to pipe with no readers</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Term</td>
<td>Timer signal from alarm(2)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>31,17</td>
<td>Term</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>20,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>18,24</td>
<td>Stop</td>
<td>Stop typed at terminal</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>21,26</td>
<td>Stop</td>
<td>Terminal input for background process</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22,27</td>
<td>Stop</td>
<td>Terminal output for background process</td>
</tr>
</tbody>
</table>

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.
Signal Concepts (4)

- **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) (One more... SIGSTOP)*
  
  - A pending signal is received at most once.
    - Kernel uses a bit vector for indicating pending signals.
Process Groups

- Every process belongs to exactly one process group.

<table>
<thead>
<tr>
<th>Process Group</th>
<th>PID</th>
<th>PGID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreground job</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>Background job #1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Background job #2</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Shell</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

- `getpgrp()` – Return process group of current process
- `setpgid()` – Change process group of a process
Sending Signals (1)

- **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.
Sending Signals (2)

- `int kill(pid_t pid, int sig)`
  - Can be used to send any signal to any process group or process.
    - `pid > 0`, signal `sig` is sent to `pid`.
    - `pid == 0`, `sig` is sent to every process in the process group of the current process.
    - `pid == -1`, `sig` is sent to every process except for process 1.
    - `pid < -1`, `sig` is sent to every process in the process group `-pid`.
    - `sig == 0`, no signal is sent, but error checking is performed.
int main(void) {
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            while(1); /* Child infinite loop */
        }
    }
    /* 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);
    }
    return 0;
}
Installing Signal Handlers

- **sighandler_t signal (int sig, sighandler_t handler)**
  - `typedef void (*sighandler_t)(int);`
  - The signal function modifies the default action associated with the receipt of signal `sig`.

- **Different values for handler:**
  - SIG_IGN: ignore signals of type sig.
  - SIG_DFL: revert to the default action.
  - Otherwise, handler is the address of a signal handler.
    - Called when process receives signal of type `sig`.
    - Referred to as "installing" the signal handler.
    - Executing handler is called "catching" or "handling" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.
Example

```c
void int_handler(int sig)
{
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(22);
}

int main(void)
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            while(1); /* Child infinite loop */
        }
    }

    /* 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);
    }

    return 0;
}
```
Handling Signals (1)

- Things to remember
  - Pending signals are not queued.
    - For each signal type, just have single bit indicating whether or not signal is pending.
    - Even if multiple processes have sent this signal.
  - A newly arrived signal is blocked while the handler of the signal is running.
  - Sometimes system calls such as `read()` are not restarted automatically after they are interrupted by the delivery of a signal.
    - They return prematurely to the calling application with an error condition. *(errno == EINTR)*
Example #2

- Deal with non-queueing signals

```c
#define N (10)

pid_t pid[N];
int ccount = 0;

void handler (int sig) {
    pid_t id = wait(NULL);
    ccount--;
    printf("Received signal %d from pid %d\n", sig, id);
}

int main(void) {
    int i;
    ccount = N;
    signal(SIGCHLD, handler);
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            exit(0); /* child */
        }
    }

    while (ccount > 0)
        sleep (5);

    return 0;
}
```
Example #2

```c
#define N (10)

pid_t pid[N];
int ccount = 0;

void handler (int sig) {
    pid_t id;
    while((id=waitpid(-1, NULL, WNOHANG))>0) {
        ccount--;
        printf("Received signal %d from pid %d\n", sig, id);
    }
}

int main(void) {
    int i;
    ccount = N;
    signal (SIGCHLD, handler);
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            exit(0); /* child */
        }
    }

    while (ccount > 0)
        sleep (5);

    return 0;
}
```
Example #2

- Deal with non-queueing signals

```c
pid_t waitpid(pid_t pid, int *status, int options);
```

<table>
<thead>
<tr>
<th>pid</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>meaning wait for any child process.</td>
</tr>
<tr>
<td>0</td>
<td>meaning wait for any child process whose process group ID is equal to that of the calling process.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>meaning wait for the child whose process ID is equal to the value of pid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNOHANG</td>
<td>return immediately if no child has exited.</td>
</tr>
<tr>
<td>0</td>
<td>meaning wait for any child process.</td>
</tr>
</tbody>
</table>

Return Value

-> on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.
Exercise #1

- React to externally generated events

- Make zombie process
  - When the process get ctrl+c signal from keyboard, it just prints “BEEP” to the monitor 5 times with 1-second interval (use sleep)
  - Print “I’m Alive!” to the monitor after 5-times beep
Exercise #2

- React to internally generated events

- Make alarm for every 1 second
  - Print “BEEP” for each second
  - Tip: alarm(int t) send SIGALRM after t seconds