SIGNALS : HIGHER LEVEL ECF

SIGNAL is a mechanism, a higher level form of ECF, that allows processes and kernel to send a small message that an event of some type has occurred in the system.

- The kernel handles lower level h/w exceptions and are not normally visible to the user process. Signals providing a mechanism to expose those exceptions to the user processes.
- For example; kernel sends the SIGFPE (or floating point error) to the process when it is trying to divide anything by zero.
- When a process is running in forground and you type 'CNTL+c' – kernel sends SIGINT signal to each process in the process group.
- Some signal correspond to higher level s/w event; like a SIGKILL may be sent by a user process to kill another process
- A process can send a signal to itself

LINUX SIGNALS

There are about 15 SIGNALS in Unix System V; 30 in LINUX of which the first 15 are same as System V.

No	Name	Default Action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
4	SIGILL	Terminate	Illegal instruction
6	SIGABRT	Terminate & dump core	Abort signal
8	SIGFPE	Terminate & dump core	Floating point error
9	SIGKILL	Terminate	Kill program
10	SIGUSR1	Terminate	User defined signal 1
17	SIGCHLD	Ignore	A Child stopped/terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop untill next SIGCONT[1]	Stop signal not from terminal
23	SIGURG	Ignore	Urgent condition on socket
30	SIGPWR	Terminate	Power failure

[1] This signal can neither be caught nor ignored

SIGNALS : Sending and Receiving

The transfer of a signal to a destination process is done through two distinct steps

- **Sending a signal:** The kernel sends a signal to a destination process – this delivery takes place for one of the two reasons;
  - i) Detection of a system event (say, divide by 0) or the termination of the child process
  - ii) A process has invoked *kill* function explicitly requesting the kernel to kill any process.

A number of mechanisms are used to send signals that relies on the notion of the *process group*. The function *pid_t getpgrp(void)* is used to get the process group (all child of a parent belong to the same process group of its parent). The function *int setpgid(pid_t pid, pid_t pgid)* may be used to change the process group of *pid* to *pgid*.

SIGNALS : Sending and Receiving

- **Receiving a signal:** A destination process receives a signal when the kernel forces it to react on the delivery.
  - i) The process can ignore the signal,
  - ii) terminate; or
  - iii) catch it by executing a user level function; i.e., the *signal handler*.

SIGNALS : Pending

A signal sent but yet to be received is known as a *pending signal*.

- At any point of time there can be only one signal pending of a particular type; so, if a signal of type k is pending and another signal of the same type comes – it would simply be discarded.
- A process can selectively blocks some signals; the blocked signal can be sent but delivered only when that signal is unblocked
- A pending signal can be received at most once
- the kernel keeps, for each process, a pending and a blocked signal bit vector
- The kernel sets bit k for a signal of type k being delivered in *pending* and clears the same bit k whenthat signal is received

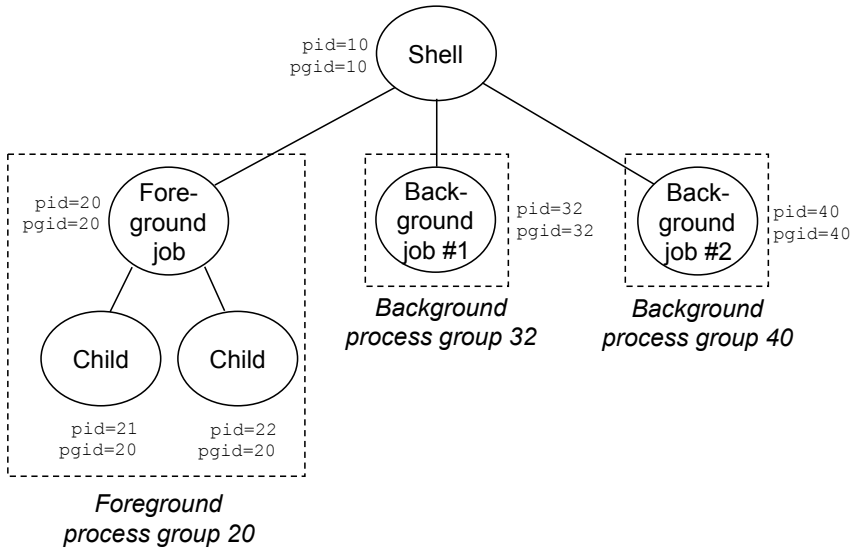
Sending Signals

The sending mechanism of a signal relies on the notion of a process group.

- Every process belongs to a single process group – which can be determined by *pid_t getpgrp(void)*.
- By default a process belongs to the process group of its parent. However, this can be changed by making a call *setpgid(pid_t pid, pid_t pgid)* which puts the process group of the process *pid* to *pgid*. [Note that if *pid = 0* then the pid of the current process is used]
- And a call *setpgid(0, 0)* from the process, say, 10000 would put process 10000 in a new process group 10000.
- A command-line /bin/kill -9 12345 sends SIGKILL to the process 12345. And /bin/kill -9 -12345 sends the SIGKILL signal to every process in the process-group 12345.

More on process group: Foreground and background processes

- The shell creates seprate progess group for each job (an abstraction of the processes that are created to evaluate a single command line; e.g. $unix > ls|sort < enter >$. Here two processes are created and connected through UNIX pipe. There can be one foreground job and 0 or more background jobs.
- A process group-id is taken from one of the parent processes for a job.



Example of a Shell with one foreground and two background jobs.

Sending SIGNALS

From the keyboard: Typing 'CNTL+c' from the keyboard results in sending signal SIGINT to every process in the foregrond process group. In the default case it terminates the foreground job.
Signal to kill : Processes send signal to other processes (including themselves) by the kill function *int kill(pid_t pid, int sig)*. Note

- If successful, the call returns 0; for error -1.
- pid = +ve sends *sig* to the process numbered pid
- pid = 0 sends *sig* to every processes in the process group of the parent process pid
- pid = -ve sends *sig* to every processes in the process group $|pid|$

Sending SIGNALs with alarm function: A process can send SIGNAL to itself by *unsigned int alarm(unsigned int secs)*

Receiving SIGNALS

When the kernel switches from a process 'p' from kernel mode to user mode (on completion of a system call or context switch etc.) it

- checks the the set of unblocked pending signals (pending & blocked) for 'p'
- if this set is empty (the usual case) the kernel passes control to the next instruction I_{curr}; however if the set is not empty
- the kernel forces a signal 'k' (usally with the lowest 'k') to the procees
- the process may in turn take appropriate action, if necessary even overriding the default action using the *signal* function.
- The default action for the signals mentioned earlier are avaiable in the table
- Also note that the default action for the signals, SIGSTOP and SIGKILL can not be changed.

```
#include <signal.h>
typedef  void (*sighandler_t) (int);
sighandler_t  signal(int signum, sighandler_t  handler);
```

Receiving SIGNALS: contd.

The signal function can change the action of the signal in a couple of ways

- if this set is empty (the usual case) the kernel passes control to the next instruction I_{curr}; however if the set is not empty
- the kernel forces a signal 'k' (usally with the lowest 'k') to the procees
- the process may in turn take appropriate action, if necessary even overriding the default action using the *signal* function.
- The default action for the signals mentioned earlier are avaiable in the table
- Also note that the default action for the signals, SIGSTOP and SIGKILL can not be changed.

```
#include <signal.h>
typedef  void (*sighandler_t) (int);
sighandler_t  signal(int signum, sighandler_t  handler);
```

Receiving SIGNALS: An example

Typing 'CNTL+c' results in termination of the forground job

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


void signal_handler(int sig)
{  printf(" Hello \n");
    exit(0);
}
int main() {  if (signal(SIGINT, signal_handler)==SIG_ERR)
 perror("error:signal");
 pause(); // wait for the handler to catch the signal
 return 0;
}
```

User defined signal: SIGUSR1

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
void signal_handler(int sig)
{    printf("Handler:  Hello\n");
}

int main() {  if (signal(SIGUSR1, signal_handler)==SIG_ERR)
 perror("error:signal");
 printf("Main\n");
 raise(SIGUSR1); // send a signal to the calling process
 return 0;
}
```

User defined signal: SIGUSR1

```
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}
void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);  kill(getppid(),SIGUSR1);
}
int main(){
  pid_t pid;
  if((pid=fork())<0){
    printf("Fork Failed\n");exit(1);
  }
  else if(pid==0){  /* Child Process */
    signal(SIGUSR1,sig_handler_child); // Register signal handler
    printf("Child: waiting for signal\n");pause();
  }
  else{  /* Parent Process */
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1); printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);  printf("Parent: waiting for response\n"); pause();
  }
  return 0;
}
// include proper .h files
```