System and Device Programming

UNIX Signals

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Definition

- A **signal** is
  - A software interrupt
  - An asynchronous notification sent, by the kernel or by another process, to a process to notify it of an event that occurred
- Signals
  - Allow notify asynchronous events
    - Such as the occurrence of particular events (e.g., error conditions, memory access violations, calculation errors, illegal instructions, etc.)
  - Can be used as a limited form of inter-process communication

Characteristics

- Available from the very first versions of UNIX
  - Originally managed in an unreliable way
- Standardized by the POSIX standard, they are now stable and relatively reliable
- Each signal has a name
  - Names start with **SIG...**
  - The file **signal.h** defines signal names
    - Unix FreeBSD, Mac OS X and Linux support 31 signals
    - Solaris supports 38 signals

Main signals

The shell command **kill –l** displays a complete list of signals

Name	Description
SIGABRT	Process abort, generated by system call abort
SIGALRM	Alarm clock, generated by system call alarm
SIGFPE	Floating-Point exception
SIGILL	Illegal instruction
SIGKILL	Kill (non maskable)
SIGPIPE	Write on a pipe with no reader
SIGSEGV	Invalid memory segment access
SIGCHLD	Child process stopped or exited
SIGUSR1 **SIGUSR2**	**User-defined signal ½** **default action = terminate the process** **Available for use in user applications**

Signal management

- Signal management goes through three phases:
  signal generation, signal delivery, reaction to a signal

  - Signal generation
    - When the kernel or a source process causes an event that generate a signal
  - Signal delivery
    - A not yet delivered signal remains pending
    - At signal delivery a process executes the actions related to that signal
    - The lifetime of a signal is from its generation to its delivery

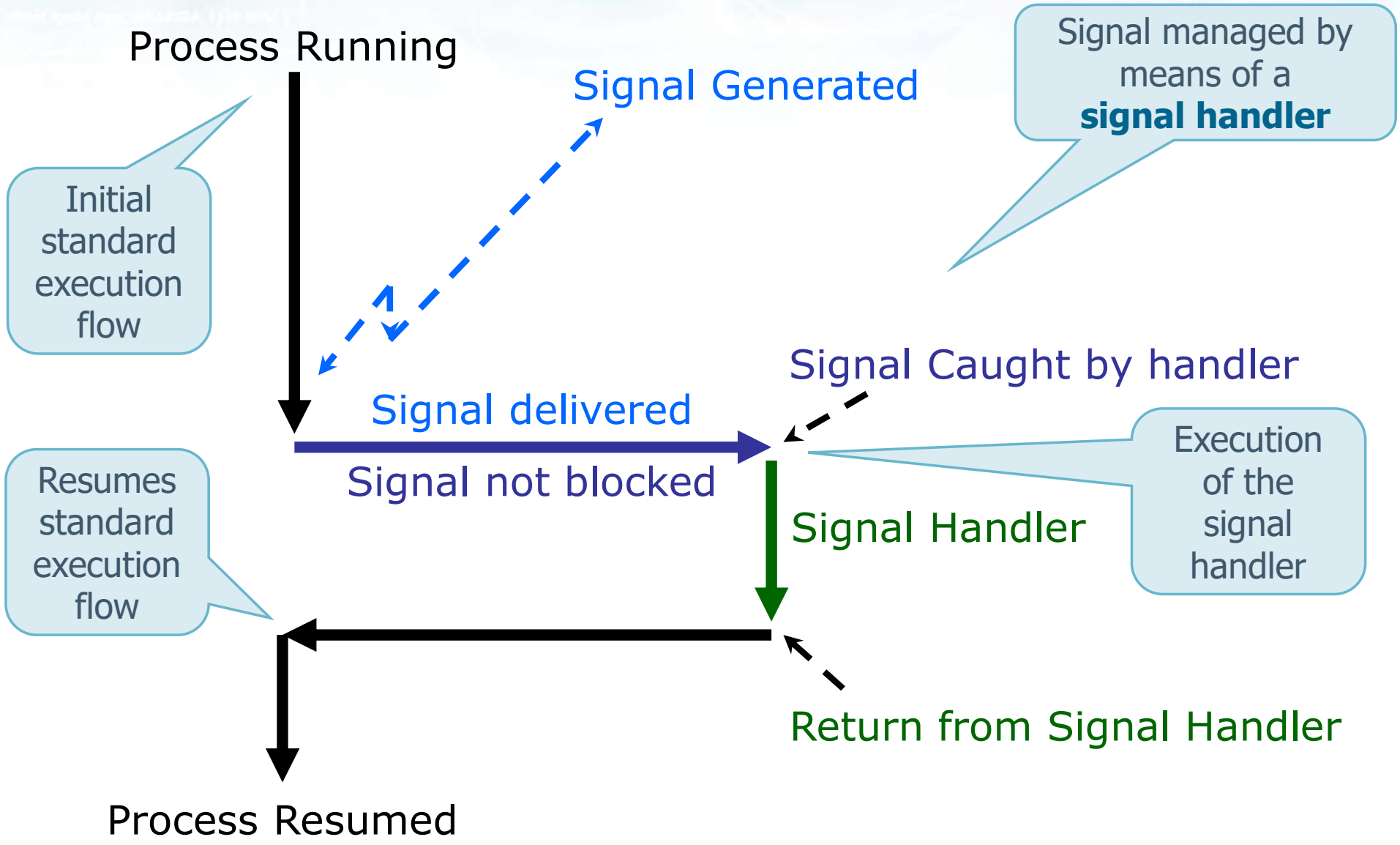
There is no signal queue; the kernel sets a flag in the process table

Signal management

Reaction to a signal

- To properly react to the asynchronous arrival of a given type of signal, a process must inform the kernel about the action that it will perform when it will receive a signal of that type
- A process may
  - Accept the default behavior (be terminated)
  - Declare to the kernel that it wants to ignore the signals of that type
  - Declare to the kernel that it wants to catch and manage the signals of that type by means of a signal handler function (similarly to the interrupt management)

Signal management



Signal management

- Signal management can be carried out with the following system calls
  - **signal**
    - Instantiates a signal handler
  - **kill** (and **raise**)
    - Sends a signal
  - **pause**
    - Suspends a process, waiting the arrive of a signal
  - **alarm**
    - Sends a SIGALARM signal, after a preset time
  - **sleep**
    - Suspends the process for a specified amount of time (waits for signal SIGALRM)

The terms **signal** and **kill** are relatively inappropriate.
signal does not send a signal!!

The signal system call

```
#include <signal.h>

void (*signal (int sig,
               void (*func)(int)))(int);
```

Received parameter of the signal handler

Returned parameter of the signal handler

- Allow to instantiate a signal handler
  - Specifies the signal to be managed (**sig**)
  - The function use to manage it (**func**), i.e., the **signal handler**

The signal system call

```
#include <signal.h>

void (*signal (int sig,
               void (*func)(int)))(int);
```

- Arguments
  - **sig** indicates the type of signal to be caught
    - SIGALRM, SIGUSR1, etc.
  - **func** specifices the address (i.e., pointer) to the function that will be executed when a signal of that type is received by the process
    - This function has an argument of `int` type, which indicates the type of signal that will be handled

The signal system call

```
#include <signal.h>

void (*signal (int sig,
               void (*func)(int)))(int);
```

- Returned values
  - on success, the previous value of the signal handler, i.e., the pointer to the previous signal handler function
    - Returns a void *
  - **SIG_ERR** on error, **errno** is set to indicate the cause
    - #define SIG_ERR ((void (*)()) -1

Reaction to a signal

- The **signal** system call allows setting three different reactions to the delivery of a signal
  - Accept the default behavior
    - signal (SIGname, **SIG_DFL**)
  - Ignore signal delivery
    - signal (SIGname, **SIG_IGN**)
  - Catch the signal
    - signal (SIGname, **signalHandlerFunction**)

Example

Setting a program to deal with 2 signals

```
...
void manager (int sig) {
  if (sig==SIGUSR1)
    printf ("Received SIGUSR1\n");
  else if (sig==SIGUSR2)
    printf ("Received SIGUSR2\n");
  else printf ("Received %d\n", sig);
  return;
}
...
int main () {
  ...
  signal (SIGUSR1, manager);
  signal (SIGUSR2, manager);
  ...
}
```

Same signal handler for more than one signal type

Both signal types must be declared

Example

Asynchronous manipulation of SIGCHLD (with no wait)

```
static void sigChld (int signo) {
  if (signo == SIGCHLD)
    printf ("Received SIGCHLD\n");
  return;
}
...
signal(SIGCHLD, sigChld);
if (fork() == 0) {
  // child
  ...
  exit (i);
} else {
  // father
  ...
}
```

There is no pid = wait (&code);

System call kill

```
#include <signal.h>

int kill (pid_t pid, int sig);
```

- Send signal (**sig**) to a process or to a group of processes (**pid**)
- To send a signal to a process, you must have the rights
  - A **user** process can send signals only to processes having the same UID
  - The **superuser** can send signal to any process

System call kill

- Arguments

If pid is	Send sig
>0	To process with PID equal to `pid`
==0	To all processes with GID equal to its GID (if it has the rights)
<0	To all processes with GID equal to the absolute value of `pid` (if it has the rights)
==–1	To all processes (if it has the rights)

- Return value
  - The value 0, if successful
  - The value –1, in case of error

```
int kill (pid_t pid, int sig);
```

Esercise

- Write a program using UNIX signals to synchronize two processes P_1 and P_2
  - P_1 displays a message on standard output and send a message to wake-up P_2
  - P_2 wakes-up, displays a message on standard output, and send a message to wake-up P_1
  - This is an example of the expected output
    - Child Woke-up; Father Woke-up; Child Woke-up; Father Woke-up, etc.
- Use sleep intructions, to decrease the probability to have race-conditions

Solution

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

static void signalHandler ();
static void father (pid_t);
static void child (pid_t);

static void signalHandler (void) {
  return;
}
```

Signal Handler

Solution

```
int main (void) {
  pid_t pid;
  if (signal(SIGUSR1, signalHandler) == SIG_ERR) {
    fprintf (stderr, "Signal Handler Error.\n");
    return (1);
  }
  pid = fork ();
  if (pid==0) {
    // Child Running
    child (getppid ());
  } else {
    // Father Running
    father (pid);
  }
  return (0);
}
```

Signal handler instantiation

Run process P_1 and P_2

Solution

```
static void father (pid_t pid) {
  fprintf (stdout,
    "Father Running: pid=%d, childPid=%d\n",
    getpid (), pid);

  while (1) {
    sleep (2);
    kill (pid, SIGUSR1);
    pause ();
    fprintf (stdout, "Father Woke-up\n");
  }
  return;
}
```

Solution

```
static void child (pid_t pid) {
  fprintf (stdout,
    "Child Running: pid=%d, fatherPid=%d\n",
    getpid (), pid);
  while (1) {
    pause ();
    fprintf (stdout, "Child Woke-up\n");
    sleep (2);
    kill (pid, SIGUSR1);
  }
  return;
}
```

System call raise

```
#include <signal.h>

int raise (int sig);
```

- The **raise** system call allows a process to send a signal to itself
  - The system call
    - raise (sig)

    is equivalent to
    - Kill (getpid(), sig);

System call pause

```
#include <unistd.h>

int pause (void);
```

- Suspends the calling process until a signal is received
- Returns after the completion of the signal handler
  - In this case the function returns -1

System call alarm

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

Return value:
The number of
seconds remaining or 0

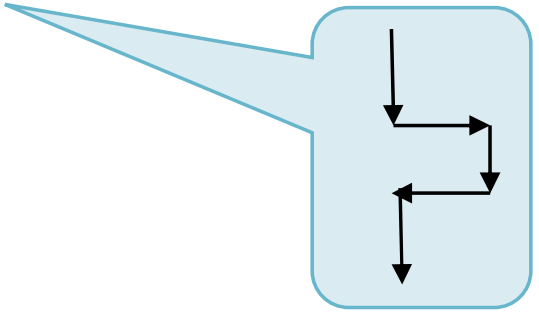
- Activate a timer (i.e., a count-down)
  - The **seconds** parameter specifies the count-down value (in seconds)
    - At the end of the countdown the signal **SIGALRM** is generated
  - If the system call is executed before the previous call has originated the corresponding signal, the count-down restarts from a new value

Signal limitations

- Signals do not convey any information
- The **memory** of the "pending" signals is **limited**
  - Max one signal pending (sent but not delivered) per type
    - Forthcoming signals of the same type are lost
  - Signals can be ignored
- Signals require functions that must be **reentrant**
- Produce **race conditions**
- Some limitations are avoided in POSIX.4

Reentrant functions

- A signal has the following behavior
  - The interruption of the current execution flow
  - The execution of the signal handler
  - The return to the standard execution flow at the end of the signal handler



- Consequently
  - The kernel **knows** where a signal handler returns, but the signal handler **does not know**
  - The signal handler must operate in a **compatible way** with the original execution flow

Race conditions

- **Race condition**
  - The result of more concurrent processes working on common data depends on the execution order of the processes instructions
- Concurrent programming is subject to race conditions
  - Using signals increases the probability of race conditions
- Race condition should be avoided at all costs