Processes

Signals

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Interrupts

- Interrupt
  - Interruption of the current execution due to the occurrence of an extraordinary event
- It can be caused by
  - A hardware device that sends a service request to the CPU
  - A software process that requires the execution of a particular operation
- For further information on interrupts:
  - https://www.skenz.it/listing/os/u04-processes/u04s10-interrupts.pdf

Definition

- A **signal** is
  - a software interrupt
  - i.e., 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

Definition

- Examples of common signals
  - Termination of a child
    - **SIGCHLD** sent to the parent;
      default action = ignore the signal
  - Press on the terminal Ctrl-C
    - **SIGINT** sent to the running process (in foreground);
      default action = terminate the process
  - Invalid memory access
    - **SIGTSTP** sent by the kernel to the process;
      default action = suspend the execution

Definition

- The system call sleep(t)
  - **SIGALARM** sent after t seconds;
    default action = restart the process
- Press on the terminal Ctrl-Z
  - **SIGTSTP** sent to the running process (in foreground)
    default action = suspend the execution
- Press on the terminal Ctrl-\
  - **SIGQUIT** sent to the running process (in foreground)
    default action = terminate the process and dump core

Signals sent by the exception handlers

Exception	Exception handler	Signal
Divide error	divide_error( )	SIGFPE
Debug	debug( )	SIGTRAP
Breakpoint	int3( )	SIGTRAP
Overflow	overflow( )	SIGSEGV
Bounds check	bounds( )	SIGSEGV
Invalid opcode	invalid_op( )	SIGILL
Segment not present	segment_not_present( )	SIGBUS
Stack segment fault	stack_segment( )	SIGBUS
General protection	general_protection( )	SIGSEGV
Page Fault	page_fault( )	SIGSEGV
Intel-reserved	None	None
Floating-point error	coprocessor_error( )	SIGFPE

Characteristics

- Available from the very first versions of UNIX
  - Originally managed in an unreliable way
    - They could be lost
      - Unix Version 7: a signal could be sent and never received
    - At the reception of each signal the behavior returned the default one
      - The signal handler had to be **re**loaded
    - A process could not ignore the reception of a signal

Characteristics

- 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

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**

You can display the complete list of signals using the shell command **`kill -l`**

Signal management

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

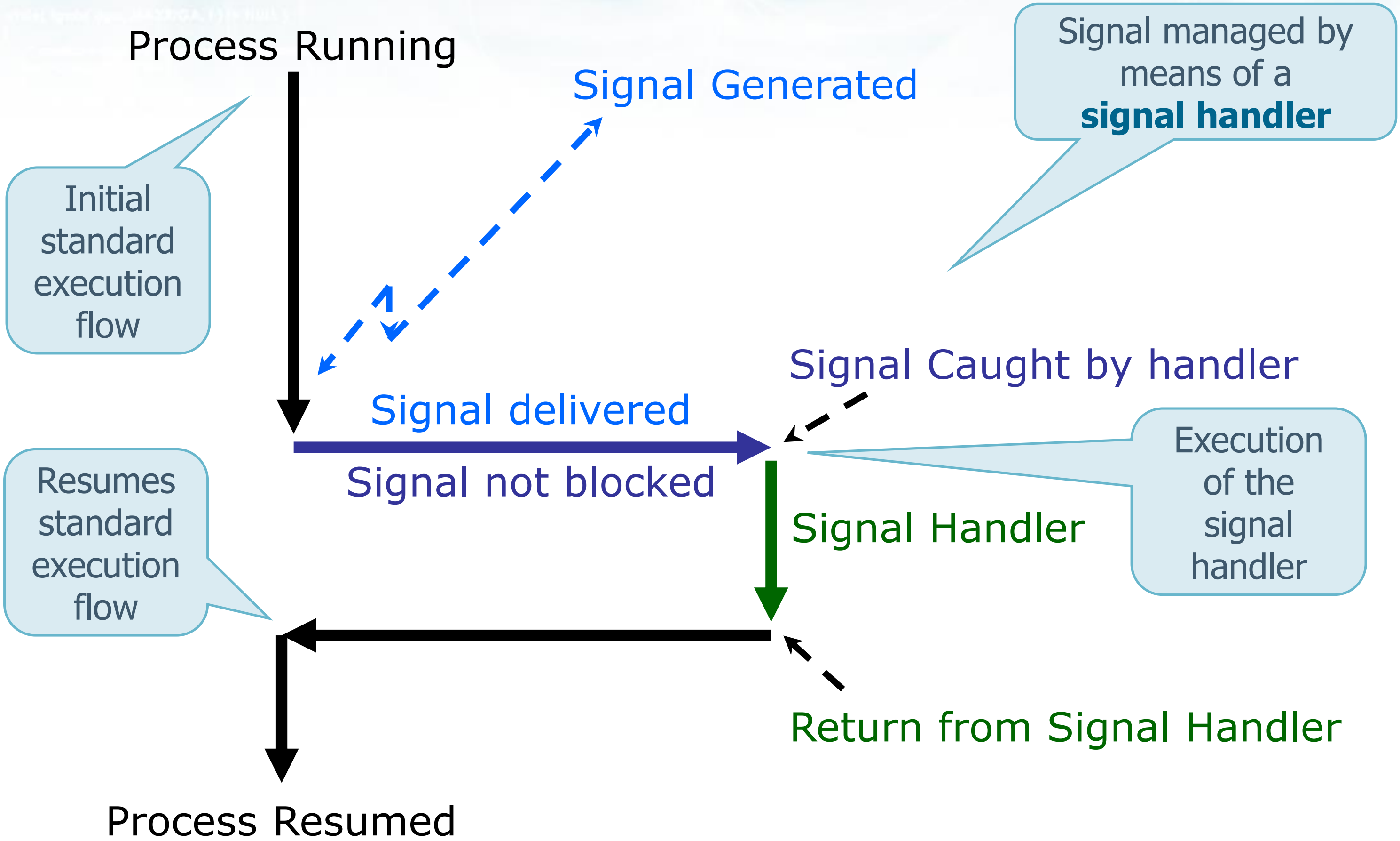
  - Signal generation
    - When the kernel or a source process causes an event that generate a signal
  - Signal delivery
    - A not yet delivered signal remains pending
    - At signal delivery a process executes the actions related to that signal
    - The lifetime of a signal is from its generation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

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**

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** specifies 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 a single argument of `int` type, which indicates the type of signal that will be handled

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

- **signal** system call allows setting three different reactions to the delivery of a signal
  - Accept the default behavior
    - signal (SIGname, **SIG_DFL**)
    - Where **SIG_DFL** is defined in `signal.h`
      - `#define SIG_DFL ((void (*)()) 0`
    - Every signal has its own default behavior, defined by the system
    - Most of the default reactions is process termination

Reaction to a signal

- Ignore signal delivery
  - signal (SIGname, **SIG_IGN**)
  - Where **SIG_IGN** is defined in `signal.h`
    - `#define SIG_DFL ((void (*)()) 1`
  - Applicable to the majority of signals
    - Ignoring a signal often leads to an indefinite behavior
  - Some signals cannot be ignored
    - `SIGKILL` and `SIGSTOP` cannot be ignored because the kernel and the superuser would not have the possibility to control all processes
    - Ignoring an illegal memory access, signaled by `SIGSEGV`, would produce an undefined process behavior

Reaction to a signal

- Catch the signal
  - signal (SIGname, **signalHandlerFunction**)
  - where
    - **SIGname** indicates the signal type
    - **signalHandlerFunction** is the user defined signal handler function
  - The signal handler
    - Can take action considered correct for the management of the signal
    - Is executed asynchronously when the signal is received
    - When it returns, the process continues with the next instruction, as it happens for interrupts

A signal handler function must be defined **for every** signal type that must be caught

Example 1

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

void manager (int sig) {
  printf ("Received signal %d\n", sig);
  // signal (SIGINT, manager);
  return;
}
int main() {
  signal (SIGINT, manager);
  while (1) {
    printf ("main: Hello!\n");
    sleep (1);
  }
}
```

Signal handler for signal SIGINT

Obsolete versions: re-instantiate the signal

Declares the signal handler

Example 2

```
...
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 3-A

Synchronous management of SIGCHLD (with wait)

```
if (fork() == 0) {
  // child
  i = 2;
  sleep (1);
  printf ("i=%d PID=%d\n", i, getpid());
  exit (i);
} else {
  // father
  sleep (5);
  pid = wait (&code);
  printf ("Wait: ret=%d code=%x\n", pid, code);
}
```

When a child dies, a SIGCHLD signal is sent to the parent

Wait: ret = 3057 code = 200

Example 3-B

Altering the behavior of **wait**

```
signal (SIGCHLD, SIG_IGN);

if (fork() == 0) {
  // child
  i = 2;
  sleep (1);
  printf (“i=%d PID=%d\n", i, getpid());
  exit (i);
} else {
  // father
  sleep (5);
  pid = wait (&code);
  printf ("Wait: ret=%d code=%x\n", pid, code);
}
```

Ignore SIGCHLD, sent by the kernel to the parent at the exit of a child

PID=3057

No wait:
Wait: ret = -1 code = 7FFFZ

The execution of a **signal(SIGCHLD, SIG_IGN)** prevents children from becoming zombies while a **signal(SIGCHLD, SIG_DFL)** is not sufficient for this purpose (even if SIGCHLD is ignored)

Example 3-C

Asynchronous management of SIGCHLD

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

kill() system call

```
#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

kill() system call

```
#include <signal.h>

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

❖ 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)

"All process" excludes a set of system processes

kill system call

```
#include <signal.h>

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

- Returned values
  - 0 on success
  - –1 on error

If sig=0 a NULL signal is sent (i.e., no signal is sent).
This is often used to check if a process exists: if the kill returns -1 the process does not exist.

raise() system call

```
#include <signal.h>

int raise (int sig);
```

- The **raise** system call allows a process to send a signal to itself
  - `raise (sig)` is equivalent to
  - `kill (getpid(), sig)`

pause() system call

```
#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

alarm() system call

```
#include <unistd.h>

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

- 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 SIGALRM is not caught or ignored, the default action is the process termination

alarm() system call

```
#include <unistd.h>

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

- If the system call is executed before the previous call has originated the corresponding signal, the count-down restarts from a new value.
  - In particular, if seconds is equal to 0 (seconds), the previous alarm is deactivated

alarm() system call

```
#include <unistd.h>

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

- Returned values
  - the number of seconds remaining until the delivery of a previously scheduled alarm
  - zero if there was no a previously scheduled alarm

alarm system call

```
#include <unistd.h>

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

- Warning
  - The signal is generated by the kernel
    - It is possible that the process get the CPU control after some time, depending on the scheduler decisions
  - There is only one time counter for each process, and system calls **sleep** and **alarm** uses the same kernel timer

Example

- Implement system call **sleep** using system calls **alarm** and **pause**

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

static void sig_alrm(int signo) {return;}

unsigned int sleep1(unsigned int nsecs)
{
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
    return (nsecs);
  alarm (nsecs);
  pause ();
  return (alarm(0));
}
```

The signal handler must be instanced before setting the alarm

After setting the alarm the system waits a signal

Returns 0, or the remaining time before the delivery if `pause` returns because another signal has been received

Example

- Implement system call **alarm** using system calls **fork**, **signal**, **kill** and **pause**

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

void myAlarm (int sig) {
  if (sig==SIGALRM)
    printf ("Alarm on ...\n");
  return;
}
```

Example

```
int main (void) {
  pid_t pid;
  (void) signal (SIGALRM, myAlarm);
  pid = fork();
  switch (pid) {
    case -1: /* error */
      printf ("fork failed");
      exit (1);
    case 0: /* child */
      sleep(5);
      kill (getppid(), SIGALRM);
      exit(0);
  }
  /* parent */
  pause ();
  exit (0);
}
```

The child waits and sends **SIGALRM**

The parent pauses, and continues only when it receives the SIGALRM sent by the child

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

Limited memory

Program with 2 signal handlers:
sigUsr1 and ...

```
...
static void sigUsr1 (int);
static void sigUsr2 (int);

static void
sigUsr1 (int signo) {
  if (signo == SIGUSR1)
    printf("Received SIGUSR1\n");
  else
    printf("Received wrong SIGNAL\n");

  fprintf (stdout, "sigUsr1 sleeping ...\n");
  sleep (5);
  fprintf (stdout, "... sigUsr1 end sleeping.\n");
  return;
}
```

Limited memory

Program with 2 signal handlers:
sigUsr1 and **sigUsr2**

```
static void
sigUsr2 (int signo) {
  if (signo == SIGUSR2)
    printf("Received SIGUSR2\n");
  else
    printf("Received wrong SIGNAL\n");

  fprintf (stdout, "sigUsr2 sleeping ...\n");
  sleep (5);
  fprintf (stdout, "... sigUsr2 end sleeping.\n");

  return;
}
```

Limited memory

```
int
main (void) {
  if (signal(SIGUSR1, sigUsr1) == SIG_ERR) {
    fprintf (stderr, "Signal Handler Error.\n");
    return (1);
  }
  if (signal(SIGUSR2, sigUsr2) == SIG_ERR) {
    fprintf (stderr, "Signal Handler Error.\n");
    return (1);
  }
  while (1) {
    fprintf (stdout, "Before pause.\n");
    pause ();
    fprintf (stdout, "After pause.\n");
  }
  return (0);
}
```

The main iterates waiting signals from shell

Limited memory

Shell commands

```
> ./pgrm &
[3] 2636
> Before pause.
> kill -USR1 2636
> Received SIGUSR1
sigUsr1 sleeping ...
... sigUsr1 end sleeping.
After pause.
Before pause.
> kill -USR2 2636
> Received SIGUSR2
sigUsr2 sleeping ...
... sigUsr2 end sleeping.
After pause.
Before pause.
```

Correctly received SIGUSR1

Correctly received SIGUSR2

Observation:
shell command **kill** sends a signal to a process with a specified PID

Limited memory

Two signals sent in sequence:
SIGUSR1 and SIGUSR2

```
> kill -USR1 2636 ; kill -USR2 2636
> Received SIGUSR2
sigUsr2 sleeping ...
... sigUsr2 end sleeping.
Received SIGUSR1
sigUsr1 sleeping ...
... sigUsr1 end sleeping.
After pause.
Before pause.
```

Both are received

The deliver order of the two signal cannot be predicted (it this case SIGUSR2 arrives first)

Limited memory

```
> kill -USR1 2636 ; kill -USR2 2636 ; kill -USR1 2636
> Received SIGUSR1
sigUsr1 sleeping ...
... sigUsr1 end sleeping.
Received SIGUSR2
sigUsr2 sleeping ...
... sigUsr2 end sleeping.
After pause.
Before pause.

> kill -9 2636
[3]+  Killed   ./pgrm
```

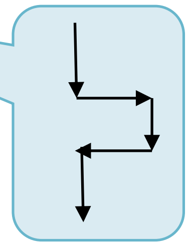
Three signals sent in sequence: two SIGUSR1 and one SIGUSR2

A SIGUSR1 is lost

-9 = SIGKILL = Kill
Kill a process

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** where it was called, i.e., the control flow was interrupted by the signal

Reentrant functions: Examples

- What happens if the signal handler performs an operation that is **not compatible** with the original execution flow?
  - Suppose a **malloc** is interrupted, and the signal handler calls another malloc
    - Function malloc manages the list of the free memory regions, which could be corrupted
  - Suppose that the execution of a function that uses a **static variable** is interrupted, but is then called by the signal handler
    - The static variable could be used to store a new value, i.e., it does not remain the same it was before the signal was delivered

Reentrant functions: Conclusions

- The "Single UNIX Specification" defines the reentrant functions, which can be interrupted without problems
  - read, write, sleep, wait, etc.
- Most of the I/O standard C functions are not reentrant
  - printf, scanf, etc.
  - They use static variables or global variables
  - They must be used carefully and being aware of possible problems

A call to printf can be interrupted and give unexpected results

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 conditions example A

- Suppose a process decides to suspend itself for a given number of seconds

See implementation of **sleep** using **alarm** and **pause**

See implementation of **alarm** using **fork**, **signal**, **kill** and **pause**

```
static void
myHandler (int signo) {
   ...
}
...
signal (SIGALARM, myHandler)
alarm (nSec);
pause ();
```

Race conditions example A

- Suppose a process decides to suspend itself for a given number of seconds
- The signal could be delivered before the execution of pause due to a contest switching and scheduling decisions (especially in high loaded systems)

```
static void
myHandler (int signo) {
   ...
}
...
signal (SIGALARM, myHandler)
alarm (nSec);
pause ();
```

Signal **SIGALRM** can be delivered before **pause**

pause blocks the process forever because the signal has been lost

Race conditions example B

- Suppose two processes P_1 and P_2 decide to synchronize by means of signals
- Unfortunately
  - If P_1 (P_2) signal is delivered before P_2 (P_1) executes **pause**
  - Process P_2 (P_1) blocks forever waiting a signal

P_1

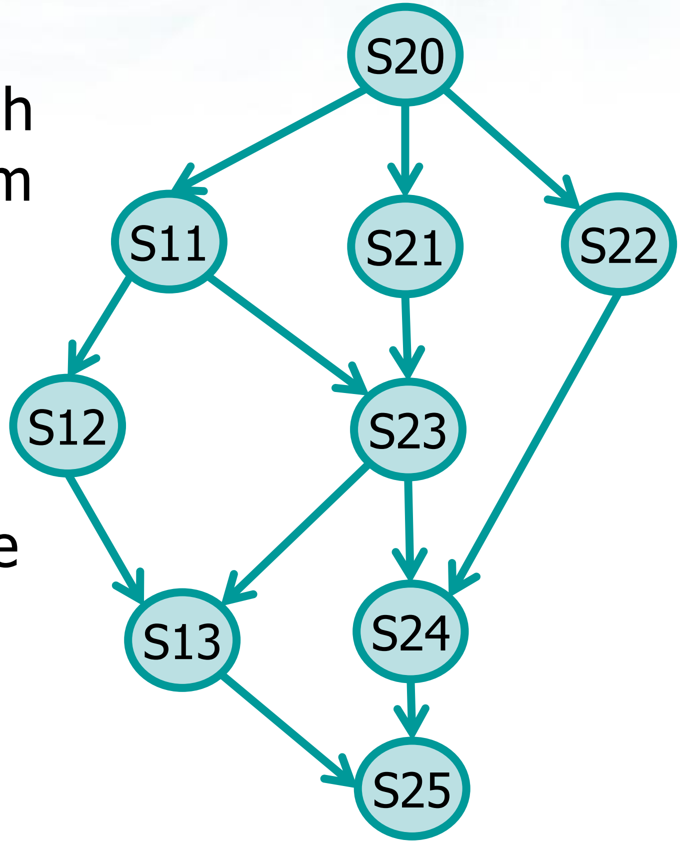
```
while (1) {
   ...
   kill (pidP2, SIG...);
   pause ();
}
```

P_2

```
while (1) {
   pause ();
   ...
   kill (pidP1, SIG...);
}
```

Exercise

- Despite their defects, signals can provide a rough synchronization mechanism
- **Ignoring the race conditions** (and using `fork`, `wait`, `signal`, `kill`, and `pause`) implement this precedence graph



Solution

Definition of the signal handler

```
static void
sigUsr ( int signo) {
  if (signo==SIGUSR1)
    printf ("SIGUSR1\n");
  else if (signo==SIGUSR2)
    printf ("SIGUSR2\n");
  else
    printf ("Signal %d\n", signo);
  return;
}
```

Solution

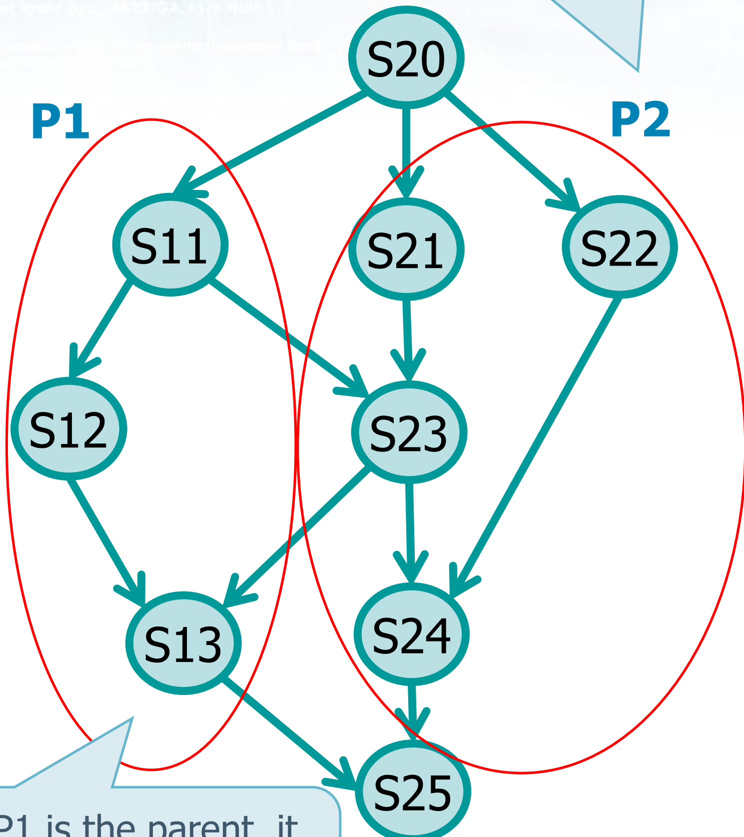
Instancing of the signal handler for signals SIGUSR1 and SIGUSR2

```
int main (void) {
  pid_t pid;

  if (signal(SIGUSR1, sigUsr) == SIG_ERR) {
    printf ("Signal Handler Error.\n");
    return (1);
  }
  if (signal(SIGUSR2, sigUsr) == SIG_ERR) {
    printf ("Signal Handler Error.\n");
    return (1);
  }
```

Solution

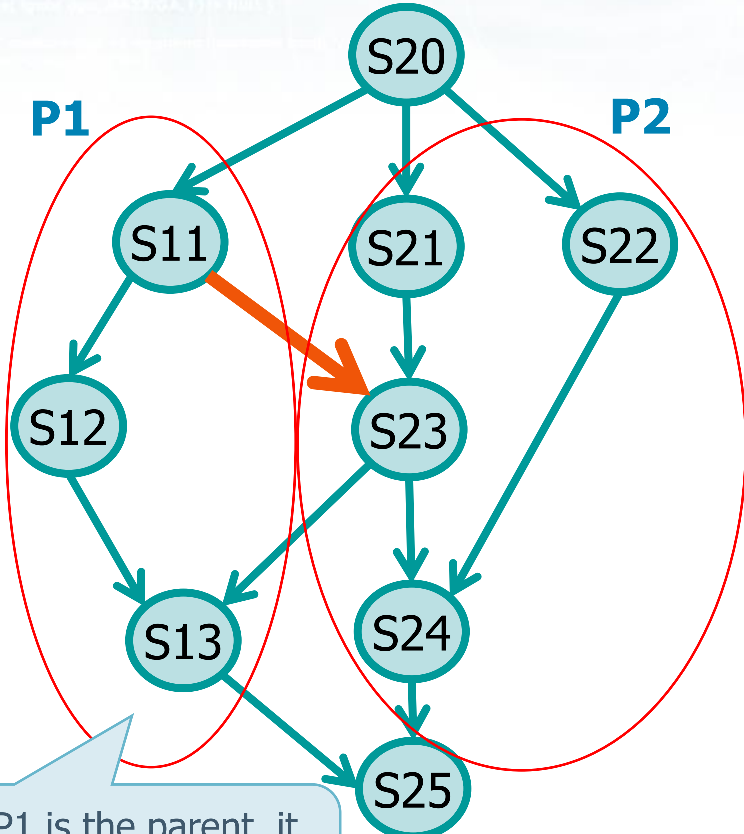
P2 is the child. It can obtain the pid of the parent with getppid()



P1 is the parent, it must store the pid of the child

```
printf ("S20\n");
pid = fork ();
if (pid > (pid_t) 0) {
   P1 (pid);
   wait ((int *) 0);
} else {
   P2 ();
   exit (0);
}
printf ("S25\n");
return (0);
}
```

Solution



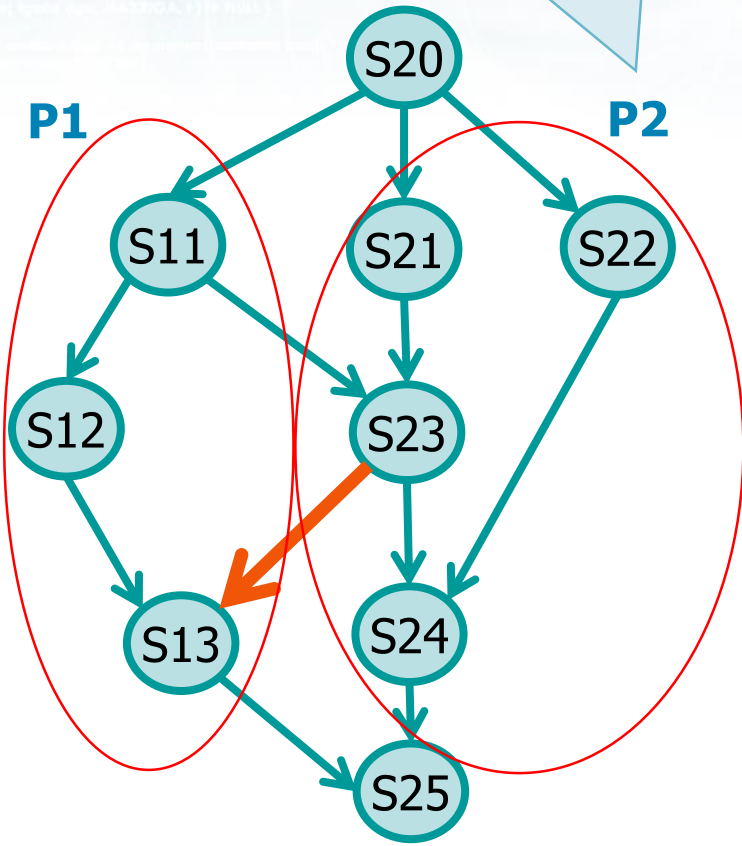
P1 is the parent, it must store the pid of the child

```
void P1 (
  pid_t cpid
) {
  printf ("S11\n");
  sleep (1);  // !?
  kill (cpid, SIGUSR1);
  printf ("S12\n");
  pause ();
  printf ("S13\n");

  return;
}
```

Solution

P2 is the child. It can obtain the pid of the parent with getppid()




```
void P2 ( ){
  if (fork () > 0) {
    printf ("S21\n");
    pause ();
    printf ("S23\n");
    kill (getppid (),
              SIGUSR2);
    wait ((int *) 0);
  } else {
    printf ("S22\n");
    exit (0);
  }
  printf ("S24\n");
  return;
}
```