#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXPAROLA 30
#define MAXRIGA 80
int main(int argc, char *argv[])
{
int freq[MAXPAROLA] ; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
char riga[MAXRIGA] ;
int i, inizio, lunghezza ;
FILE * f ;
for(i=0; i<MAXPAROLA; i++)
freq[i]=0 ;
if(argc != 2)
{
fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
exit(1);
}
f = fopen(argv[1], "rt") ;
if(f==NULL)
{
fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
exit(1);
}
while(fgets(riga, MAXRIGA, f) != NULL)

Processes

Signals

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

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
  - can be used as a limited form of inter-process communication
  - allow notify asynchronous events such as the error conditions illustrated for exceptions

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
  - can be used as a limited form of inter-process communication
  - allow notify asynchronous events such as the error conditions illustrated for exceptions

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

Terminal signals

- Typing some key combinations at the controlling terminal of a process causes the system to send it a signal:
- **Ctrl-C** sends an SIGINT signal
  - by default, this causes the process to terminate.
- **Ctrl-Z** sends a terminal stop SIGTSTP signal
  - by default, this causes the process to suspend execution.
- **Ctrl-\** sends a SIGQUIT signal;
  - by default, this causes the process to terminate and dump core.

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 1/2**

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

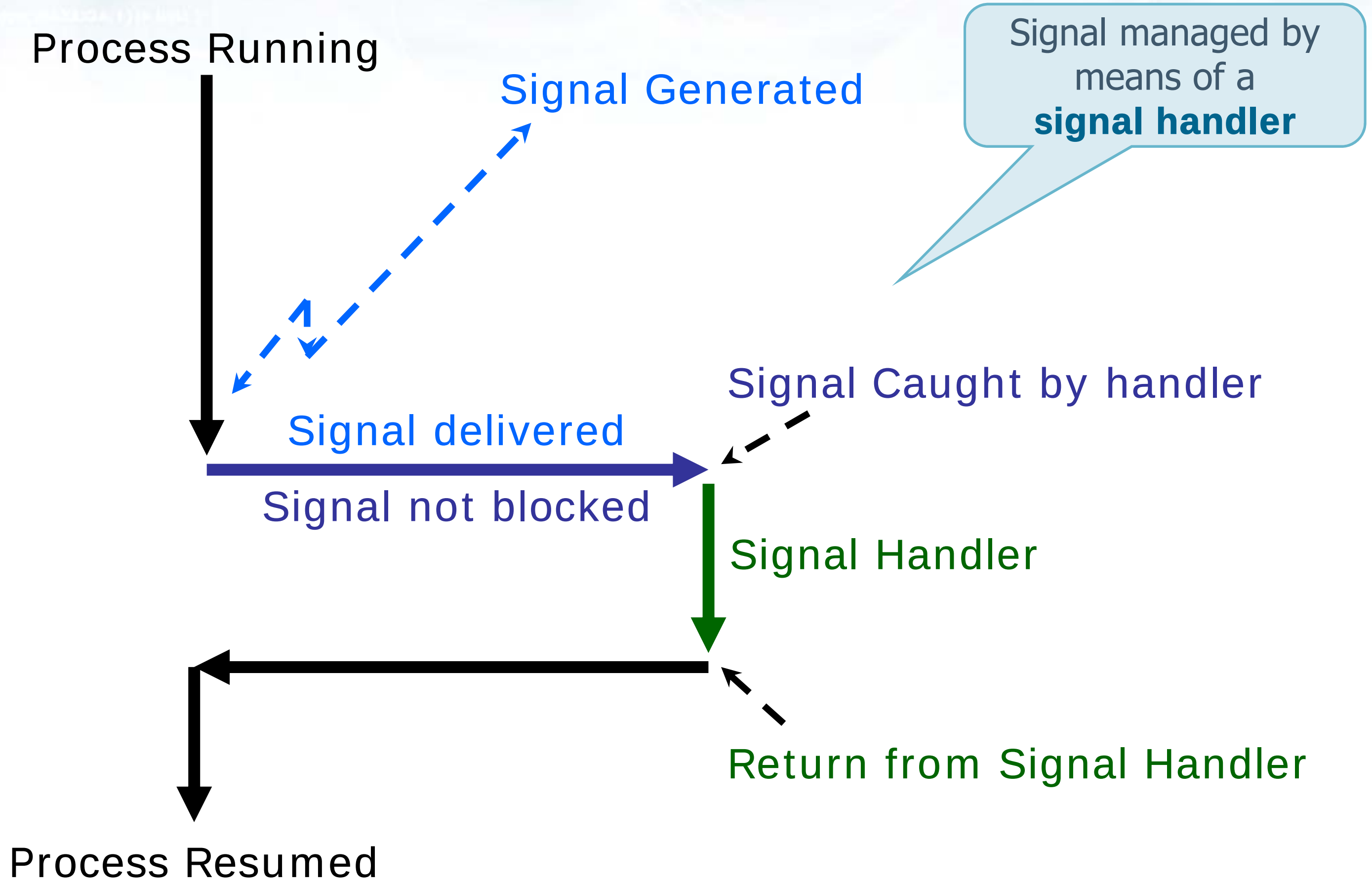
- Signal generation
  - When the kernel or a process causes an event that generate a signal
- Signal delivery
  - A not yet delivered signal remains pending
  - At signal delivery a process executes the actions related to that signal
  - The lifetime of a signal is from its generation to its delivery

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

- Precondition to properly handle a received signal for a process is to **declare** to the kernel if a signal of a given type will be ignored or caught.
- This is done using the system call **signal**
  - Which **instantiates** a signal handler

signal does not send a signal !!

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 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);
```

- Return value
  - the previous value of the signal handler, i.e., the previous signal handler function
  - **SIG_ERR** in case of error, **`errno`** is set to indicate the cause

Signal generation

- The kernel generates signals
  - SIGCHLD, SIGFPE, etc.
- A process can (ask the kernel to) generate a signal by means of the system call
  - **kill** (and **raise**)

kill is misleading, does not kill a process, just send to it a signal

  - **alarm**
    - Ask the kernel to receive a SIGALRM after a specified amount of time

Waiting for a signal

- A process can wait for a signal by means of the system call
  - **pause,** and any other blocking system call
    - Suspend the process until any signal is received
  - **sleep**
    - Suspend the process for a specified amount of time (waits for signal SIGALRM)

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**
  - 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
    - Is executed when the signal is delivered,
    - 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 caugth

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

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

Standard behavior of **wait**

```
// signal (SIGCHLD, SIG_IGN);

for (i=0; i<3; i++) {
  if (fork() == 0) {
    // child
    sleep (1);
    printf (“i=%d PID=%d\n", i, getpid());
    exit (i);
  }
}
sleep (5);
for (i=0; i<3; i++) {
  ret = wait (&code);
  printf ("Wait: ret=%d code=%x\n", ret, code);
}
```

```
i=2 PID=3057
i=1 PID=3056
i=0 PID=3055
```

```
Wait: ret = 3055 code = 0
Wait: ret = 3056 code = 100
Wait: ret = 3057 code = 200
```

Example 3

Altering the behavior of **wait**

```
signal (SIGCHLD, SIG_IGN);

for (i=0; i<3; i++) {
  if (fork() == 0) {
    // child
    sleep (1);
    printf (“i=%d PID=%d\n", i, getpid());
    exit (i);
  }
}
sleep (5);
for (i=0; i<3; i++) {
  ret = wait (&code);
  printf ("Wait: ret=%d code=%x\n", ret, code);
}
```

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

No wait:

Wait: ret = -1 code = 7FFF

Wait: ret = -1 code = 7FFF

Wait: ret = -1 code = 7FFF

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**)
  - 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
<0	To all processes with GID equal to the absolute value of **pid**
==–1	To all processes

"All process" excludes a set of system processes

kill system call

```
#include <signal.h>

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

- Return value
  - 0 on success
  - –1 on error

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 delivered
- Returns after the completion of the signal handler
  - returns -1, and errno is set to EINTR

alarm system call

```
#include <unistd.h>

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

- Ask the kernel so send a **`SIGALRM`** to the calling process in **`seconds`** seconds
  - The default action for SIGALRM is process termination
  - A call to **`alarm(seconds)`**before the expiration of a previous **`alarm`** reschedules the request to the kernel
  - **`alarm(0)`** cancels any pending alarm

alarm system call

```
#include <unistd.h>

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

- Return value
  - the number of seconds remaining until the delivery of a previously scheduled alarm
  - zero if there was no 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
  - 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));
}
```

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) {
  printf (“Alarm\n”);
}
```

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 per type
    - Forthcoming signals of the same type are lost
  - Signals can be ignored
- Signal interrupt functions that must be **reentrant**
- Produce **race conditions**
- Some limitations are avoided in POSIX.4

Limited memory

```
...
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

```
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

```
> 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

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 of them are SIGUSR1

A SIGUSR1 is lost

-9 = SIGKILL = Kill
Kill a process

Reentrant functions

- 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

- 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 are not reentrant
  - printf, scanf, etc.
  - They use static variables or global variables, thus must be used carefully and being aware of possible problems

Race conditions

- **Race condition**
  - The result of more concurrent processes working on common data depends on the execution order of the processes instructions
- Using signals increases the probability of race conditions.

Race conditions example

- 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.

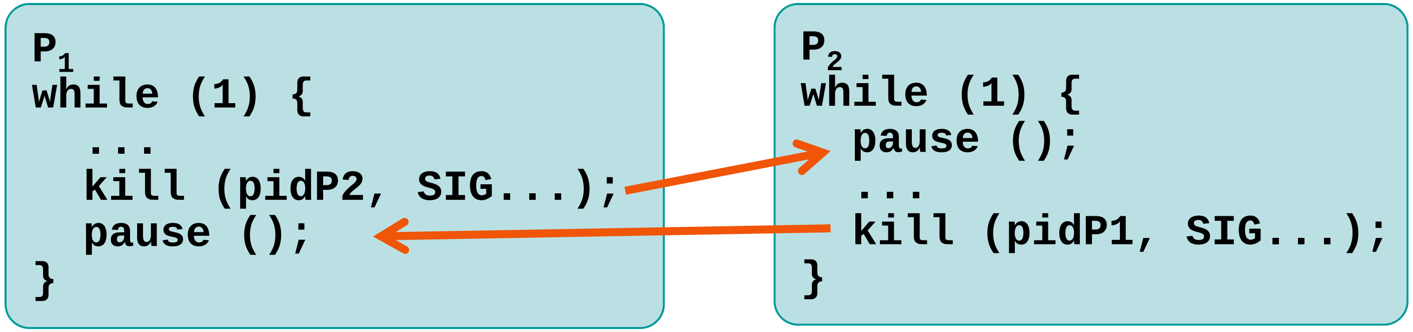
```
static void
myHandler (int signo) {
  ...
}
...
signal (SIGALARM, myHandler)
alarm (nSec);
pause ();
```

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

pause blocks the process forever because the signal has been lost

Race conditions example

- Suppose two processes P_1 and P_2 decide to synchronize by means of signals
  - 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

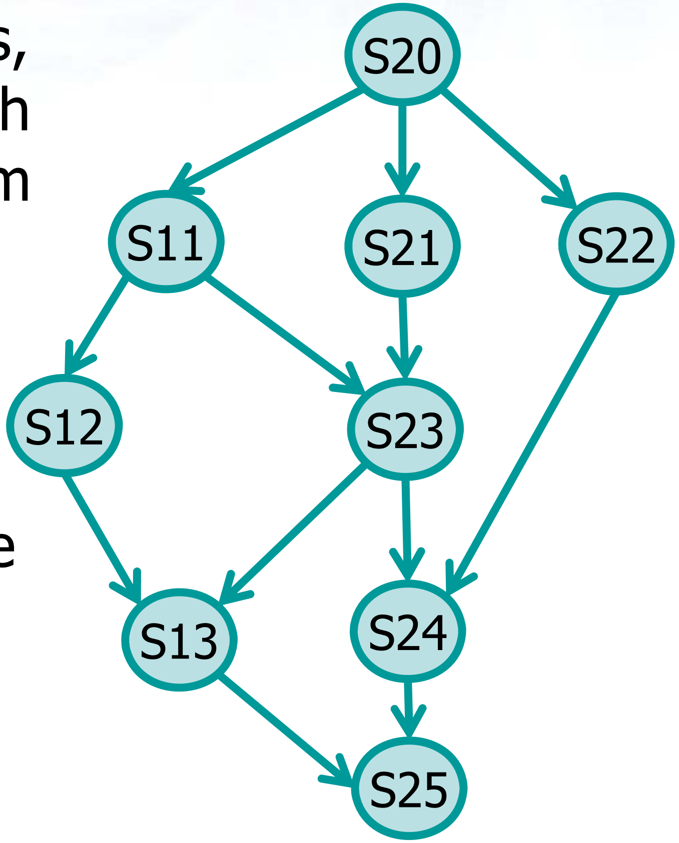


```
P1
while (1) {
  ...
  kill (pidP2, SIG...);
  pause ();
}
```

```
P2
while (1) {
  pause ();
  ...
  kill (pidP1, SIG...);
}
```

Exercise

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



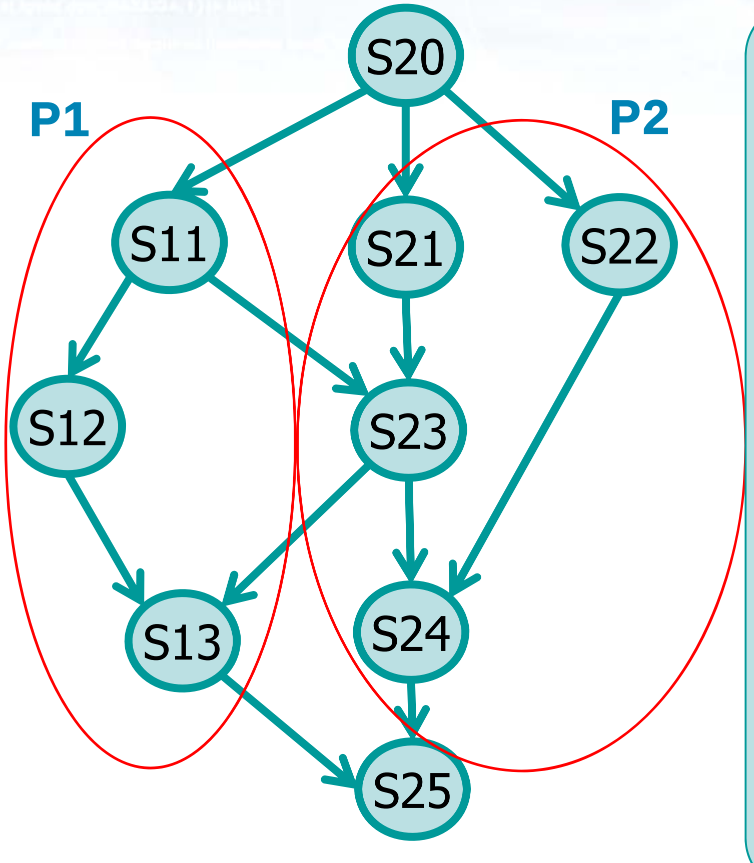
Solution

```
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

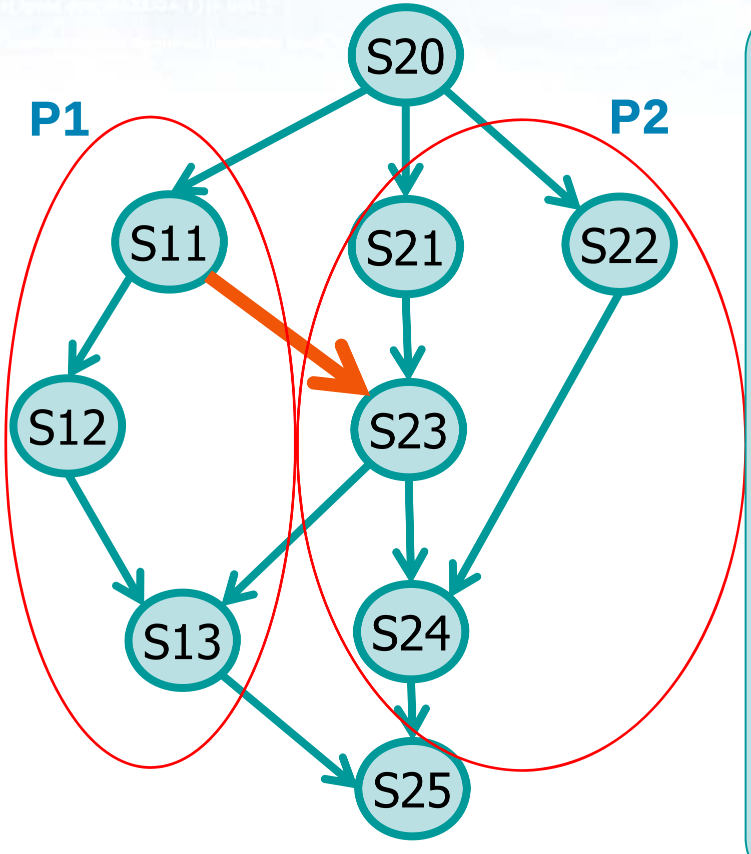
```
int main (void) {
  pid_t pid;

  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



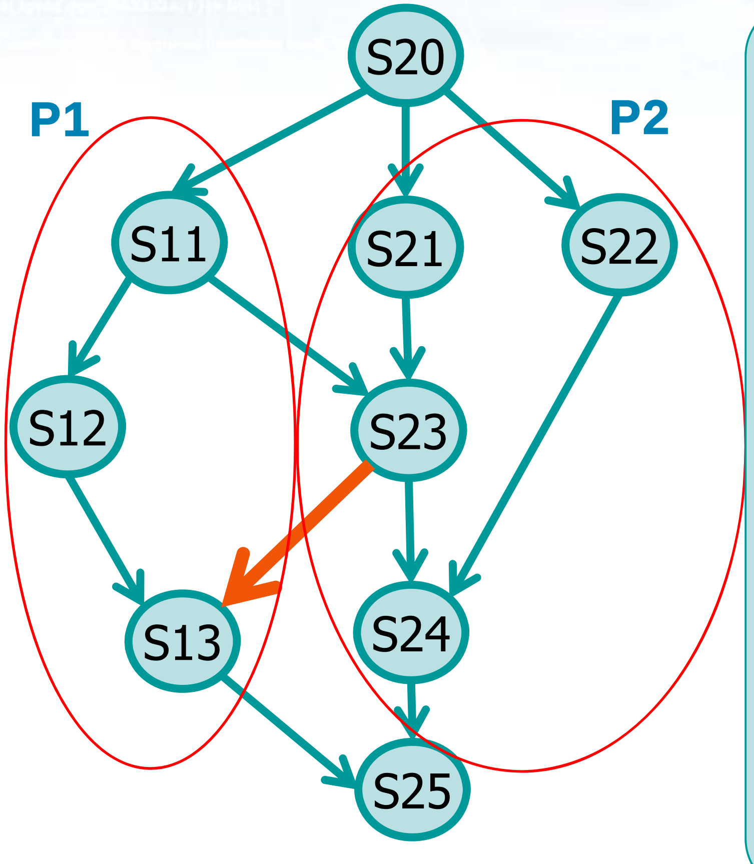
```
  printf ("S20\n");
  pid = fork ();
  if (pid > (pid_t) 0) {
    P1 (pid);
    wait ((int *) 0);
  } else {
    P2 ();
    exit (0);
  }
  printf ("S25\n");
  return (0);
}
```

Solution



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

  return;
}
```

Solution



```
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;
}
```