Signals

Sistemas de Computadores

2020/2021

António Barros / Luís Ferreira

Dealing with the unexpected

- The main body of a program defines the *expected* sequence of instructions that a process should perform.

- However, unexpected events may occur during the execution of a process.
  - Such events may require special handling.
  - But if they are not predicable, WHERE should we code the handling?

Signals

- A *signal* is a notification of an event sent to a specific process.
  - Process receives an **integer** that maps to a specific **event**.
- Notifications are **asynchronous.**
  - A process may receive a notification at any time, during its execution.
  - If you can't predict when an event occurs, then it is not practical to handle it in the main body of the program.

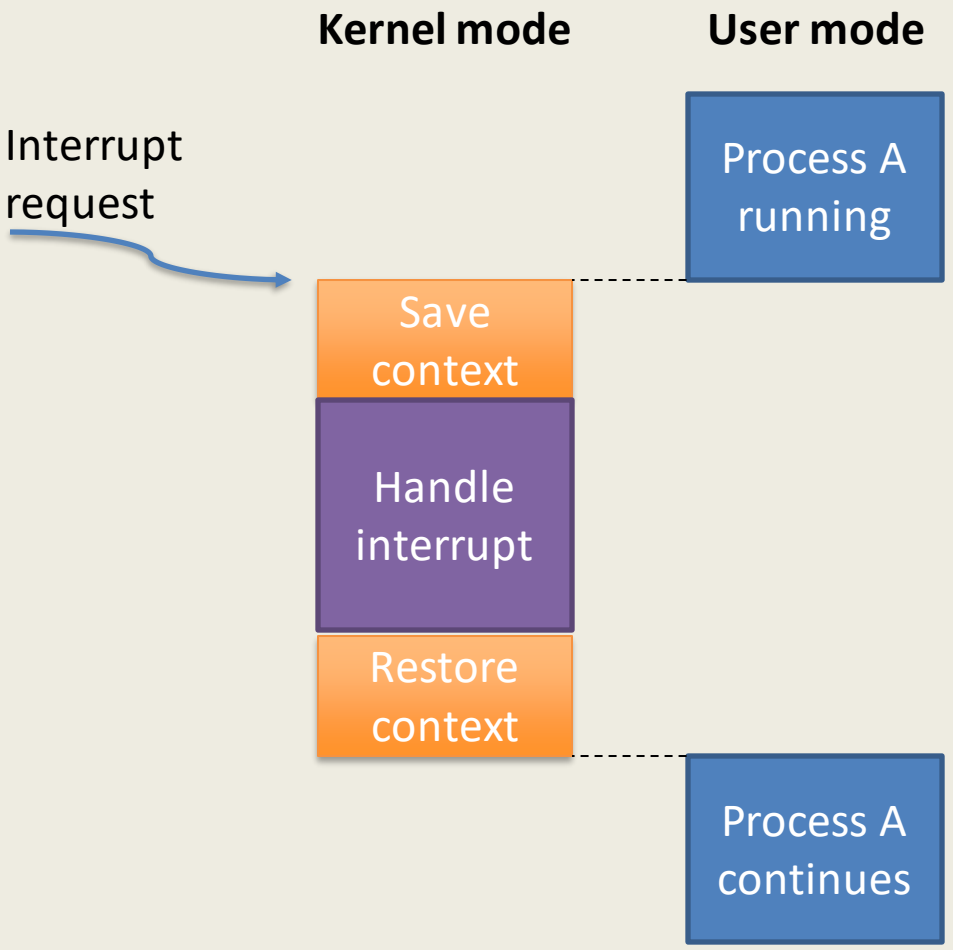
  - **The solution is to write specialized code to handle the signal and execute it only when necessary!**

Causes of signals

- Hardware interrupt
  - Exceptions such as invalid memory access or a division by zero.
  - Special key combinations (e.g. CTRL-C).
- Signals raised by the user.
- Signals raised by other processes.
- Signals raised by the process itself.

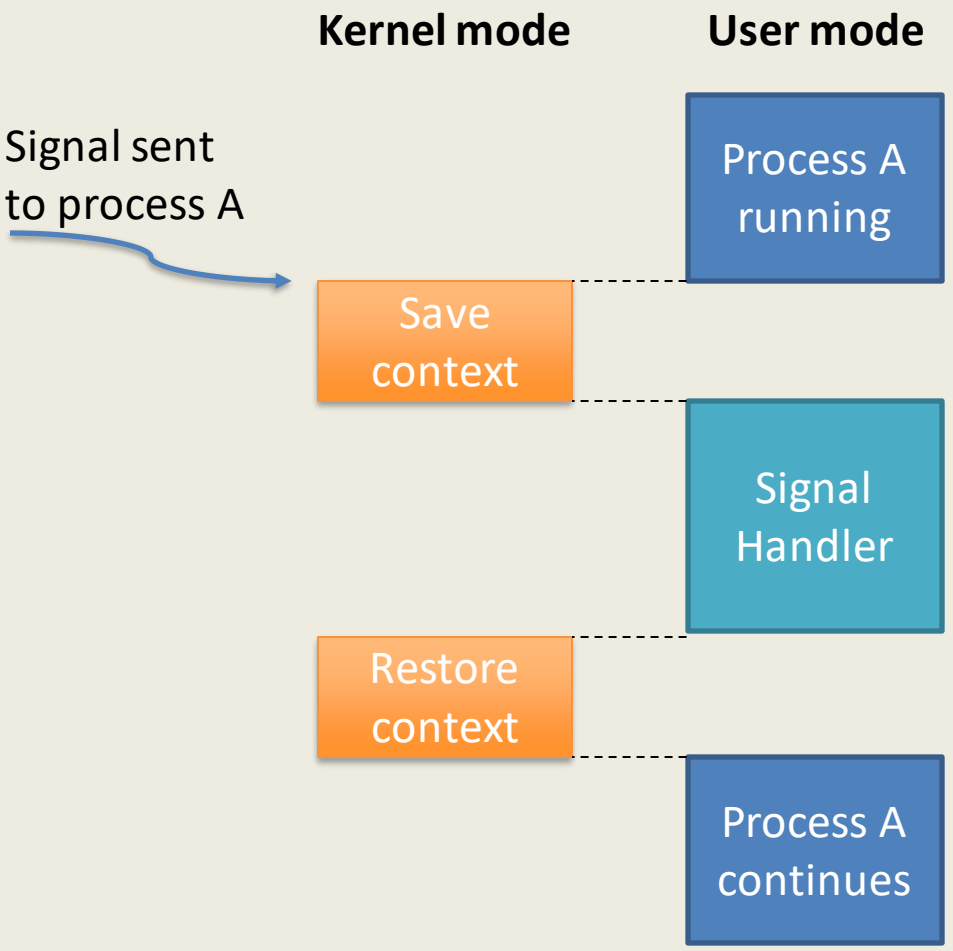
Recalling an interrupt

- CPU receives an interrupt
- Kernel takes over
  - saves current context
  - handles interrupt
  - restores saved context
- Interrupted process continues executing



Handling a signal

- Kernel receives a signal issued to process
- Kernel saves current context
- Process handles signal by calling a specific function
- Kernel restores saved context
- Signaled process continues executing (or maybe terminates)



Which events? Which signals?

- You can list the signals in your system, with command:
  - `$ kill -l`
- … and also look at the manual pages:
  - `$ man signal`

Some examples of signals

Signal	Default action	Description
SIGINT	Terminate process	Interrupt program
SIGILL	Create core image	Illegal instruction
SIGKILL	Terminate process	Kill program (***can't be ignored or handled***)
SIGSEGV	Create core image	Segmentation violation (illegal memory access)
SIGALRM	Terminate process	Real-time timer expired
SIGSTOP	Stop process	Stop (***can't be ignored or handled***)
SIGCONT	Discard signal	Continue after stop
SIGCHLD	Discard signal	Child status has changed
SIGUSR1	Terminate process	User defined signal 1
SIGUSR2	Terminate process	User defined signal 2

Dealing with an incoming signal

- Default action for specific signal
  - Terminate
    - SIGKILL, SIGINT, SIGUSR1, SIGUSR2
  - Create core image and terminate
    - SIGILL, SIGSEGV, SIGFPE, SIGABORT
  - Stop/Continue
    - SIGSTOP, SIGCONT
  - Nothing (ignore)
    - SIGCHLD, SIGIO, SIGINFO
- Programmer-defined action (if signal allows)
  - Ignore
  - Handler function

Setting the action for a signal

- The action for a signal `sig` is set using the `sigaction()` function.

```
#include <signal.h>

int sigaction(int sig,
              const struct sigaction * act,
              struct sigaction * oact);
```

- `act` sets the action details for signal `sig`, and
- `oact` (if not `NULL`) is used to store the previously set action details.

Action details

- `struct sigaction` may differ slightly across architectures but these fields must comply with POSIX.
  - `sa_handler`
    - pointer to an ANSI C handler function
  - `sa_sigaction`
    - pointer to a POSIX handler function
  - `sa_mask`
    - mask of signals to be blocked during signal handling
  - `sa_flags`
    - set of flags that modifies the behaviour of the signal

`sa_handler` field

- Used to set an ANSI C handler function
- Possible values:
  - `SIG_DFL` for the default action
  - `SIG_IGN` to ignore the signal
    - SIGKILL and SIGSTOP cannot be ignored!
  - `void (*sa_handler)(int)`
    (i.e. pointer to an ANSI C signal handler function)

Example: ignoring SIGUSR1

```
/* (...) */

int main(int argc, char *argv[]){
  struct sigaction act;

  /* Clear the act variable. */
  memset(&act, 0, sizeof(struct sigaction));

  act.sa_handler = SIG_IGN;
  sigaction(SIGUSR1, &act, NULL);

  /* SIGUSR1 will now be ignored. */
  /* (...) */
}
```

Example: capturing SIGUSR1

```
/* (...) */

void handle_USR1(int signo)
{
  write(STDOUT_FILENO, "\nCatch USR1!\n", 13);
}

int main(int argc, char *argv[]){
  struct sigaction act;

  memset(&act, 0, sizeof(struct sigaction));
  sigemptyset(&act.sa_mask); /* No signals blocked */
  act.sa_handler = handle_USR1;
  sigaction(SIGUSR1, &act, NULL);

  /* SIGUSR1 will now be captured and handled (ANSI C). */
  /* (...) */
}
```

`sa_sigaction` field

- Used to set a POSIX handler.
  - Provides more power to the signal handler.
  - Requires that the `SA_SIGINFO` option is set in the `sa_flags` field.
    - **It cannot be used simultaneously with an ANSI C handler!**

- Possible values:
  -
```
void (*sa_sigaction)(int,
                     siginfo_t *,
                     void *);
```
(i.e. pointer to a POSIX signal handler function)

Example: capturing SIGUSR1

```
void handle_USR1(int signo, siginfo_t *sinfo, void *context)
{
  /* Don't use printf: it is not safe!!!           */
  /* See man sigaction for list of safe functions. */
  printf("Signal %d sent by process %d\n",
          sinfo->si_signo, sinfo->si_pid);
}

int main(int argc, char *argv[]){
  struct sigaction act;

  memset(&act, 0, sizeof(struct sigaction));
  sigemptyset(&act.sa_mask); /* No signals blocked */
  act.sa_sigaction = handle_USR1;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR1, &act, NULL);

  /* SIGUSR1 will now be captured and handled (POSIX). */
  /* (...) */
}
```

Signal received during a handler?

- When a signal is being handled, subsequent notifications of the same signal number are automatically blocked.
  - Those notifications are stored by the OS while the handler is executed.
  - The OS makes them available to be handled afterwards.
- But what about other signals?
  - By default, they will interrupt the current handler to trigger their own handler.

`sa_mask` field

- Used to specify which signals should be blocked during the signal handling.
  - Does not allow an incoming signal to interrupt the signal handling.
- A blocked signal is not ignored!
  - The signal is delayed until it becomes unblocked (*i.e. when the handler completes*).

- Set is defined on a `sigset_t` variable.
  - Manipulated by sigset operations, check out:
    - `$ man sigsetops`

Example: setting `sa_mask`

```
int main(int argc, char *argv[]){
  struct sigaction act;

  memset(&act, 0, sizeof(struct sigaction));
  sigemptyset(&act.sa_mask);          /* No signals blocked */
  sigaddset(&act.sa_mask, SIGINT);    /* Block SIGINT */
  sigaddset(&act.sa_mask, SIGUSR2); /* Block SIGUSR2 */

  act.sa_sigaction = handle_USR1;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR1, &act, NULL);

  /* Now, when a SIGUSR1 is received, SIGINT and SIGUSR2 */
  /* are blocked.                                        */

  /* (...) */

}
```

Example: setting `sa_mask`

```
int main(int argc, char *argv[]){
  struct sigaction act;

  memset(&act, 0, sizeof(struct sigaction));
  sigfillset(&act.sa_mask);          /* All signals blocked */
  sigdelset(&act.sa_mask, SIGCHLD); /* Unblock SIGCHLD */
  sigdelset(&act.sa_mask, SIGUSR1); /* Unblock SIGUSR1 */

  act.sa_sigaction = handle_USR2;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR2, &act, NULL);

  /* Now, when a SIGUSR2 is received, all signals */
  /* except SIGCHLD and SIGUSR1 are blocked.      */


  /* (...) */
}
```

`sa_flags` field

- Used to modify the behaviour of the signal.
- Check out the manual page.
  - `$ man sigaction`

Signaling from the command line

- Using the kill command, specifying the signal and the target process.
  - `$ kill 1234`
    - Send (the default) SIGTERM to process with PID 1234.
  - `$ kill -SIGKILL 1234`
    - Send the SIGKILL to process with PID 1234.
- Check out the manual page:
  - `$ man 1 kill`

Sending a signal from within a process

- Using the `kill()` function.
- Check out the man page for more information:
  - `$ man 2 kill` (MacOS)
  - `$ man 3 kill` (Linux)

```
#include <signal.h>

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

Example: sending a signal to child

```
int main()
{
  pid_t pid;

  pid = fork();
  if(pid > 0) {
    printf("Hello, my son!\n");
    sleep(5); /* Sleep for 5 seconds... */
    kill(pid, SIGUSR1);
    printf("Goodbye, son!\n");
  }
  else {
    printf("Son is running\n");
    pause(); /* Wait for some signal... */
  }
}
```

More ways to send signals…

- `int raise(int sig);`
  - Send the `sig` signal to the current process.
- `unsigned int alarm(unsigned int seconds);`
  - Send the `SIGALRM` signal to the current process in `seconds` seconds.
- `int pause(void);`
  - Pauses the current process until a signal is received.
- Explore the manual pages.

Example: setting an alarm

```
void handle_ALARM(int signo)
{
  write(STDOUT_FILENO, "\nTime for a coffee!\n", 20);
}

int main()
{
  struct sigaction act;

  memset(&act, 0, sizeof(struct sigaction));
  sigemptyset(&act.sa_mask); /* No signals blocked */
  act.sa_handler = handle_ALARM;
  sigaction(SIGALRM, &act, NULL);
  /* SIGALRM is now set to be catched and handled. */

  alarm(10);
  /* You have 10 seconds before interrupting the */
  /* following code for a coffee. */
  do_some_very_long_task();
}
```

Delaying signals

- A section of code may require to be immune to a set of signals.
- Ignoring the signals during the execution of that section of code was the traditional approach…
  - … but the notification of the respective event was discarded.
- POSIX signals allow to block a set of signals.
  - Signals are stored and received when unblocked.
  - Thus, signals are delayed.
  - Check out the manual page for `sigpending` and `sigprocmask`.

Example: blocking signals

```
int main()
{
  sigset_t mask, pending;

  sigemptyset(&mask);
  sigaddset(&mask,SIGINT);
  sigprocmask(SIG_BLOCK, &mask, 0);
  /* CTRL-C is now blocked! */

  sleep(10);
  /* Check if SIGINT is pending... */
  sigpending(&pending);
  if(sigismember(&pending, SIGINT))
    printf("SIGINT pending\n");

  sigprocmask(SIG_UNBLOCK, &mask, 0);
  /* CTRL-C is now unblocked! */
}
```