# Ficha 2 – Sinais

# 2025

{patricio.domingues, vitor.carreira, miguel.frade, rui.ferreira, nuno.costa, gustavo.reis, carlos.machado, leonel.santos, miguel.negrao}@ipleiria.pt



# 1 Ponteiros

Um ponteiro é um tipo de dados que permite armazenar um endereço de memória. O endereço pode referir-se ao endereço de uma variável, a um bloco de memória alocado dinamicamente ou mesmo a uma função. Quando o endereço se refere a uma variável, um ponteiro permite que esta seja lida/alterada indiretamente. A passagem de parâmetros por referência é um exemplo típico da utilização de ponteiros. Para relembrar estes tópicos pode consultar os materiais de Sistemas Operativos disponibilizados no Moodle.

## 1.1 Declaração

Sintaxe:

```
tipo_dados *nome_do_ponteiro;
```

Exemplos:

```
int *iptr;   /* Declara um ponteiro para um inteiro */
float *fptr; /* Declara um ponteiro para um float */
char *cptr;  /* Declara um ponteiro para um caráter */
```

> ***NOTA***
>
> *A declaração de uma variável do tipo ponteiro apenas reserva espaço para a guardar, mas não a faz apontar para qualquer endereço de memória. É um erro utilizar um ponteiro antes de o inicializar.*

## 1.2 Operadores especiais

Existem dois operadores especiais para operações com ponteiros, `*` e `&`, com a seguinte sintaxe:

- `*<ponteiro>` - devolve o conteúdo da zona de memória definida pelo `<ponteiro>`;
- `&<variavel>` - devolve o endereço de memória da `<variavel>`.

Apesar do operador de multiplicação e o operador `*` utilizarem o mesmo símbolo, não existe nenhuma relação entre eles. Note que o operador `&` só pode ser aplicado a variáveis. Operações tais como `&10` ou `&(+2)` são ilegais.

# 1.3 Ponteiros para funções

Como foi referido no início desta secção, um ponteiro é um tipo de dados que permite armazenar um endereço de memória. Sendo uma função um conjunto de instruções guardadas em memória, um ponteiro também pode representar o endereço de uma função. Um ponteiro para uma função é útil quando, para um mesmo conjunto de dados, se pretende aplicar diferentes operações (e.g. ordenar um vetor com diferentes critérios de ordenação), ou quando a operação a invocar apenas é conhecida em tempo de execução. Nestas situações, os ponteiros para funções são utilizados para passar uma função como parâmetro de outra função.

## 1.3.1 Declaração

Sintaxe:

```
tipo_retorno (*nome_do_ponteiro)(lista de parâmetros);
```

Exemplos:

```
/* Declaração de um ponteiro para uma função que recebe dois inteiros e devolve um double */
double (*operator_func)(int, int);

/* Atribui ao ponteiro o endereço da função sum */
operator_func = sum;

/* Declara um ponteiro para uma função que recebe duas strings e devolve um inteiro e inicia-o
        com o endereço da função strcmp */
int (*strcompare_func)(char *s1, char *s2) = strcmp;
```

Quando um determinado ponteiro para função é utilizado com muita frequência, é comum utilizar a *keyword* `typedef` para definir o tipo de dados associado ao ponteiro.

Por exemplo:

```
typedef int (*STRING_CMP_T)(char*, char*);
(...)
STRING_CMP_T strcompare_func;
```

## 1.3.2 Invocação

Para invocar a função apontada pelo ponteiro utiliza-se a seguinte sintaxe: `nome_do_ponteiro(argumentos);`

Exemplos:

```
double res = operator_func(3, 4);
(...)
strcompare_func(s1, s2);
```

### 1.3.3 Aplicação prática

Imagine o cenário em que pretende construir uma função para ordenar um vetor de *strings* cujo critério de ordenação é apenas conhecido por quem utilizar a função. Neste contexto, a função de ordenação teria de receber como parâmetros: o vetor a ordenar, o número de elementos no vetor e o critério de ordenação (desconhecido à partida). A única forma de representar um critério de ordenação genérico consiste em recorrer a um ponteiro para uma função que receba duas *strings* - **s1** e **s2** - e devolve o inteiro **-1** se **s1 < s2**; **0** se **s1 == s2** e **1** se **s1 > s2**.

Um protótipo possível para a função de ordenação seria o seguinte:

```
typedef int (*STR_CMP)(const char *s1, const char *s2);

void str_asort(char *vetor[], unsigned int size, STR_CMP criteria);
```

O programa que utilizar a função `str_asort` apenas terá de indicar o critério de ordenação pretendido. Por exemplo:

```
(...)
int sort_by_length(const char *s1, const char *s2);
int sort_by_number_of_vowels(const char *s1, const char *s2);
(...)
// ordena pelo tamanho da string
str_asort(vetor, nelements, sort_by_length);

// ordena pelo número de dígitos
str_asort(vetor, nelements, sort_by_number_of_vowels);

// ordena por ordem alfabética
str_asort(vetor, nelements, strcmp);
```

### 1.3.4 Lab 1

Compile e execute o código do laboratório 1.

Depois altere o código de modo a acrescentar um critério de ordenação que permita ordenar o vetor de *strings* alfabeticamente, sem distinguir maiúsculas de minúsculas.

> ***NOTA***
>
> *Sugere-se a leitura do seguinte livro, que aborda a temática dos ponteiros na linguagem C:*
>
> - *“Understanding and Using C Pointers”, Richard Reese, O’Reilly, 2013.*

# 2 Sinais

Um processo pode ser interrompido durante a sua execução, a fim de ser notificado que determinados eventos ocorreram. A notificação é levada a cabo através da ativação de um determinado sinal, enviado por exemplo por uma sequência CTRL+C, por um comando `kill` ou pela função `kill()` de C. Os sinais mais relevantes encontram-se definidos na seguinte tabela:

| *Signal* | Valor | Descrição |
|---|---|---|
| SIGINT | 2 | Interrupção do teclado (CTRL + C) |
| SIGQUIT | 3 | Interrupção do teclado (CTRL + \) criando ficheiro core |
| SIGKILL | 9 | Termina (administrador) |
| SIGUSR1 | 10 | Definido pelo utilizador |
| SIGUSR2 | 12 | Definido pelo utilizador |
| SIGPIPE | 13 | Escrita para um *pipe* sem consumidor |
| SIGALRM | 14 | Temporizador |
| SIGTERM | 15 | Termina, sinal emitido pelo `kill` por omissão |
| SIGCHLD | 17 | Terminação do processo filho |
| SIGSTOP | 19 | Parar o processo |

> ***NOTA***
>
> *Os sinais SIGKILL e SIGSTOP não podem ser tratados. Para uma tabela mais detalhada dos sinais disponíveis em sistemas operativos Unix consulte a página de manual sobre signal da secção 7 (`man 7 signal`).*

## 2.1 Enviar um sinal através da shell - Comando `kill`

Para enviar um sinal a um processo através da *shell* (vulgo *linha de comando*), pode recorrer-se ao comando `kill`, indicando o sinal a enviar, bem como o processo alvo.

Exemplo - Enviar o sinal SIGABRT ao processo com PID 3456

```
$ kill -s SIGABRT 3456
```

> ***NOTA***
>
> *Apenas o dono de um processo ou o administrador do sistema (root) podem enviar sinais ao processo. Concretamente, um utilizador apenas pode enviar sinais aos processos que lhe pertencem, ao passo que o administrador pode enviar sinais a qualquer processo.*

## 2.2 Comportamento por omissão

Por omissão, um processo, ao receber um sinal, termina a sua execução. Dependendo do tipo de sinal, poderá ser criado um ficheiro, designado de *core*, contendo a imagem em memória do processo abortado. Esta imagem pode ser utilizada pelo depurador (*debugger*) para descobrir a causa da terminação abrupta do processo.

# 2.3 Tratamento de sinais

De forma a tirar partido da funcionalidade dos sinais, torna-se necessário associar a um determinado sinal, uma rotina de tratamento (*signal handler*), rotina essa que será executada aquando da ocorrência do sinal.

Consoante as diferentes implementações dos sistemas UNIX, os sinais podem ser fidedignos ou não fidedignos (*reliable or unreliable*). Os sistemas cujos sinais são não fidedignos são aqueles que, depois de instalada uma rotina de tratamento para um determinado sinal, essa rotina de tratamento fica desativada depois da primeira ocorrência do sinal. Caso se pretenda usar novamente a rotina de tratamento para responder ao sinal, ter-se-á de voltar a instalá-la. Contudo, essa situação pode gerar aquilo que se designa por *race-condition*, isto é, pode suceder que um sinal do mesmo tipo seja recebido ANTES que a rotina de tratamento já tenha sido reinstalada, levando a que o sinal seja perdido, ou termine o processo (comportamento por omissão).

Imaginemos que num determinado processo instalamos um *handler* para uma rotina, "**trata_sinal**", para tratar o sinal SIGINT. O processo, depois de receber um sinal do tipo SIGINT, chama a rotina "**trata_sinal**". Da segunda vez que o processo receber o sinal, o tratamento desse sinal passa a ser o considerado por omissão, isto é, conduz à terminação do processo (comportamento por omissão). Caso o processo pretenda continuar a tratar o sinal, deve associar novamente a rotina de tratamento cada vez que receber o mesmo sinal. Para resolver este problema surgiram os sistemas com sinais fidedignos em que o *handler* só é instalado uma vez para tratar o sinal de todas as vezes que este chegar.

Para tratamento dos sinais podemos utilizar a função `signal()`, em que um processo pode indicar os sinais que pretende tratar e os que quer ignorar. No entanto, o comportamento desta função depende do sistema UNIX em que está implementada. Por exemplo, em BSD esta função trata os sinais de forma fidedigna, mas em Linux já não. Assim, **se pretendermos desenvolver aplicações portáveis, devemos empregar a função** `sigaction()` que permite tratar os sinais das duas formas, fidedigna (comportamento por omissão) e não fidedigna.

# 2.4 Sigaction

```
#include <signal.h>
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

Função usada para definir a ação tomada por um processo ao receber um sinal.

- `signum` - Sinal que pretendemos tratar, exceto SIGKILL e SIGSTOP;
- `act` - Ação instalada para `signum`;
- `oldact` - Guarda a ação que anteriormente estava instalada.

## 2.4.1 Valores de retorno

**Sucesso** - Devolve 0.
**Insucesso** - Devolve -1.

### 2.4.2 Estruturas auxiliares

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int      sa_flags;
  void     (*sa_restorer)(void); /* obsoleto */
}
```

Os elementos da estrutura `sigaction` são:

| Parâmetro | Descrição |
|---|---|
| `sa_handler` | Ação para associar a signum e pode ser SIG_DFL (*default action*), SIG_IGN (ignora o sinal) ou um **ponteiro para uma função** que trata o sinal. |
| `sa_sigaction` | Ação para associar a signum caso seja passada a *flag* SA_SIGINFO. A ação corresponde a um **ponteiro para uma função** que recebe três parâmetros (sinal, estrutura com informação adicional ao sinal, contexto). Note que **não deve preencher simultaneamente** os campos `sa_handler` e `sa_sigaction`. |
| `sa_mask` | Máscara com os sinais que devem ser bloqueados durante a execução de `sa_handler`/`sa_sigaction`. Além disto, bloqueia o sinal que disparou o *handler*, a menos que seja usado SA_NODEFER. |
| `sa_flags` | Permite-nos especificar o comportamento do processo de tratamento dos sinais através de *flags* (*bitwise* OR). Seguem-se as *flags* mais importantes (para mais pormenores consultar o manual):<br>• **SA_ONESHOT** ou **SA_RESETHAND** - sinais não fidedignos;<br>• **SA_NODEFER** - não bloqueia o próprio sinal;<br>• **SA_RESTART** - proporciona um comportamento compatível com os sinais BSD, em que nas chamadas ao sistema “longas”, a chamada é reiniciada pelo próprio sistema operativo. Por omissão, as “chamadas ao sistema longas”, quando interrompidas por um sinal, não são reiniciadas pelo sistema;<br>• **SA_SIGINFO** - a ação a invocar no tratamento para o sinal é especificada pelo campo `sa_sigaction`;<br>• **0 (zero)** - sem flags. |

## 2.5 Sigemptyset

```
#include <signal.h>
int sigemptyset (sigset_t *set);
```

Esta função coloca o parâmetro `set` vazio, isto é, todos os sinais são excluídos do conjunto. Atenção, não se deve igualar o `set` a zero, não sendo nesse caso garantido o correto funcionamento da função que utilizar esse `set`.

### 2.5.1 Valores de retorno

**Sucesso** - Devolve 0.
**Insucesso** - Devolve -1.

## 2.6 Pause - Suspende o processo

```
#include <unistd.h>
int pause(void);
```

A função `pause()` suspende a execução do processo até este receber um sinal.

### 2.6.1 Valores de retorno

Devolve sempre -1.

## 2.7 Kill

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

A função `kill()` envia o sinal `sig` ao processo com PID `pid`.

### 2.7.1 Valores de retorno

**Sucesso** - Devolve 0.
**Insucesso** - Devolve -1. É atribuído um código de erro apropriado à variável de erro do sistema `errno`.

## 2.8 Alarm

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

A função `alarm()` envia o sinal SIGALRM ao próprio processo, após `seconds` segundos. Note-se que a chamada `alarm()` cancela qualquer alarme anteriormente definido e que ainda esteja pendente.

> ***NOTA***
>
> *Não é aconselhado utilizar a função `alarm()` em conjunto com a função `sleep()`, porque o `sleep()` pode, em alguns sistemas, ser implementando recorrendo ao SIGALARM.*

### 2.8.1 Valores de retorno

Devolve 0 se não foi estabelecido nenhum alarme anteriormente. Caso contrário, devolve o tempo que faltava para que o alarme anterior fosse ativado.

# 2.9 Raise

```
#include <signal.h>
int raise(int sig);
```

A função `raise()` envia o sinal `sig` ao processo corrente.

## 2.9.1 Valores de retorno

**Sucesso** - Devolve 0.
**Insucesso** - Devolve um valor diferente de 0.

# 2.10 Exemplos

## 2.10.1 Exemplo 1

Listing 1: Tratamento do sinal SIGUSR1 em espera bloqueante com resumo automático

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include "debug.h"

#define MAX 100

void trata_sinal(int signal);

void trata_sinal(int signal) {
    /* Cópia da variável global errno */
    int aux = errno;

    if (signal == SIGUSR1)
        printf("Recebi o sinal SIGUSR1 (%d)\n", signal);

    /* Restaura valor da variável global errno */
    errno = aux;
}

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

    /* Silencia warnings... */
    (void)argc;
    (void)argv;

    /* Definir a rotina de resposta a sinais */
    act.sa_handler = trata_sinal;
    /* mascara sem sinais -- nao bloqueia os sinais */
    sigemptyset(&act.sa_mask);

    act.sa_flags = SA_RESTART; /* recupera chamadas bloqueantes */

    /* Captura do sinal SIGUSR1 */
    if (sigaction(SIGUSR1, &act, NULL) < 0)
        ERROR(1, "sigaction - SIGUSR1");

    printf("O processo está pronto para receber sinais [SIGUSR1]...\n");
    printf("Introduza informação através do teclado:\n");

    n = read(0, buf, MAX - 1);  // operação bloqueante
    if (n < 0)
        ERROR(2, "Erro de entrada.");
    buf[n] = '\0';

    printf("Foi lido do teclado: %s\n", buf);
    return 0;
}
```

### 2.10.2 Exemplo 2

Listing 2: Tratamento do sinal SIGUSR1 em espera bloqueante com resumo automático utilizando o campo sa_sigaction

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include "debug.h"

#define MAX 100

void trata_sinal(int sig, siginfo_t *siginfo, void *context);

void trata_sinal(int sig, siginfo_t *siginfo, void *context) {
    (void)context;
    /* Cópia da variável global errno */
    int aux = errno;

    if (sig == SIGUSR1) {
        printf("Recebi o sinal SIGUSR1 (%d)\n", sig);
        printf("Detalhes:\n");
        printf("\tPID: %ld\n\tUID:%ld\n", (long)siginfo->si_pid,
               (long)siginfo->si_uid);
    }
    /* Restaura valor da variável global errno */
    errno = aux;
}

int main(int argc, char *argv[]) {
    /* Silencia warnings... */
    (void)argc;
    (void)argv;
    char buf[MAX];
    struct sigaction act;
    ssize_t n;

    /* Definir a rotina de resposta a sinais */
    act.sa_sigaction = trata_sinal;
    /* mascara sem sinais -- nao bloqueia os sinais */
    sigemptyset(&act.sa_mask);

    act.sa_flags = SA_SIGINFO;  /*info adicional sobre o sinal */
    act.sa_flags |= SA_RESTART; /*recupera chamadas bloqueantes*/

    /* Captura do sinal SIGUSR1 */
    if (sigaction(SIGUSR1, &act, NULL) < 0)
        ERROR(1, "sigaction - SIGUSR1");

    printf("O processo está pronto para receber sinais [SIGUSR1]...\n");
    printf("Introduza informação através do teclado:\n");

    n = read(0, buf, MAX - 1);  // operação bloqueante
    if (n < 0)
        ERROR(2, "Erro de entrada.");
    buf[n] = '\0';

    printf("Foi lido do teclado: %s\n", buf);
    return 0;
}
```

### 2.10.3 Lab 2

Compile e execute o código do projeto fornecido no Lab 2.

Depois deve lançar uma segunda *shell* e executar:

```
kill -s SIGUSR1 <PID do processo receptor>
```

De volta à primeira *shell*, e sem digitar nada, verifique a saída do programa. Repita o comando `kill` na segunda *shell* e volte a verificar a saída do programa na primeira *shell*.

O que pode concluir com a saída obtida e com a análise do código?

> ***NOTA***
>
> *Para terminar o programa, deve digitar qualquer coisa e carregar em ENTER.*

### 2.10.4 Lab 3

Altere o código no projeto do *Lab 3* de modo a eliminar a utilização da *flag* SA_RESTART.

Execute os mesmos procedimentos descritos no *Lab 2*, aplicando-os agora à nova versão do *Lab 3*.

Por fim, **comente os resultados**.

### 2.10.5 Exemplo 3

O próximo programa espera recorrendo à chamada bloqueante `pause()` até que a variável `continua` seja colocada a zero. A chamada `pause()` bloqueia o processo até que este receba um sinal (i.e. seja interrompido).

Listing 3: Tratamento de vários sinais em espera bloqueante

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

#include "debug.h"

void trata_sinal(int signal);

/* Flag para termino do ciclo principal do programa*/
int continua = 1;

/* Rotina de tratamento de sinais*/
void trata_sinal(int signal) {
    int aux;
    /* Cópia da variável global errno */
    aux = errno;

    if (signal == SIGUSR1) {
        printf("Recebi o sinal SIGUSR1 (%d)\n", signal);
    } else if (signal == SIGINT) {
        continua = 0;
        printf("Recebi o sinal SIGINT (%d)\n", signal);
    }
    /* Restaura valor da variável global errno */
    errno = aux;
}

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

    /* Silencia warnings... */
    (void)argc;
    (void)argv;

    /* Regista rotina de tratamento de sinais  */
    act.sa_handler = trata_sinal;

    /* Comportamento por omissao -- fidedigno */
    act.sa_flags = 0;

    /* Máscara sem sinais para não os bloquear */
    sigemptyset(&act.sa_mask);

    /* Captura do sinal SIGUSR1 */
    if (sigaction(SIGUSR1, &act, NULL) < 0) {
        ERROR(1, "sigaction - SIGUSR1");
    }

    /* Captura do sinal SIGINT */
    if (sigaction(SIGINT, &act, NULL) < 0) {
        ERROR(2, "sigaction - SIGINT");
    }

    /* Informa utilizador do funcionamento do programa */
    printf("O programa esta pronto a receber os signals SIGINT e SIGUSR1\n");
    printf("PID do processo: %d\n", getpid());

    /* ciclo principal */
```

```
60      while (continua) {
61          pause(); /* Espera bloqueante */
62          printf("Pause interrompido\n");
63      }
64      return 0;
65  }
```

### 2.10.6 Lab 4

Compile e execute o código do *Lab 4*.

Depois, deve lançar uma segunda *shell* e executar:

```
kill -s SIGUSR1 <PID do processo receptor>
```

Verifique agora qual a resposta do programa, na primeira *shell*. Repita o passo anterior e verifique novamente a resposta devolvida.

Por fim, na segunda *shell*, execute o seguinte:

```
kill -s SIGINT <PID do processo receptor>
```

e verifique se o processo terminou.

## 2.11 Funções *async-signal safe*

Os sinais são um mecanismo assíncrono de comunicação entre processos, pelo que podem interromper o fluxo normal de execução de um processo. Neste caso, não é recomendável incluir código na rotina de tratamento dos sinais que altere o estado de determinadas variáveis estáticas do sistema. Sendo assim, apenas se devem utilizar funções *async-signal safe*. Para obter uma lista das funções *async-signal safe* em sistemas operativos Unix, consulte a página de manual sobre *signal-safety* da secção 7 (`man 7 signal-safety`). O capítulo 8.6 do livro "*Unix Systems Programming*" também contém uma lista de funções que se podem utilizar.

Por exemplo, a função `trata_sinal()` das listagens anteriores salvaguarda a variável `errno` antes de efetuar a chamada `printf()`. A função `printf()` não é *async-signal safe*, uma vez que, nesta situação, pode alterar o valor da variável global `errno` quando é invocada. Nestes casos, o procedimento correto consiste em salvaguardar o valor da variável `errno` e restaurá-lo no final da função.

Para além da utilização de variáveis globais, como é o caso da variável `errno`, há outros motivos que determinam se uma função é ou não *async-signal safe*, como por exemplo, a utilização de *locks* ou de operações de E/S em ficheiros. Para mais informações, consulte a página de manual `signal-safety(7)`.

# 3 Exercícios

> ***NOTA***
>
> *Na resolução de cada exercício deverá utilizar o template de exercícios que inclui uma makefile bem como todas as dependências necessárias à compilação.*

# 3.1 Para a aula

1. Elabore o programa **showFirst** que leia e mostre a primeira linha de um ficheiro de texto sempre que receber um sinal de um processo filho. O processo filho deve enviar um sinal ao pai de 5 em 5 segundos. O ficheiro deve ser o único argumento de entrada (parâmetro **--file/-f** <ficheiro>). O tratamento do argumento de entrada deve feito com recurso ao **gengetopt**.

2. **(1º teste prático 2017-18)** Recorrendo à linguagem C, elabore a aplicação **exec_on_signal**. A aplicação deve estar preparada para receber o parâmetro **--execute <comando> / -e <comando>**, em que `comando` corresponde a um comando Linux sem opções adicionais. Por exemplo, é permitido especificar **--execute date**, mas não **--execute date +%Y%m%d**. Sempre que a aplicação recebe o sinal SIGUSR1, a aplicação deve mostrar uma mensagem apropriada na saída padrão e seguidamente executar o comando que foi especificado através do parâmetro **--execute/-e**. Caso a execução do comando falhe, a aplicação **exec_on_signal** deve terminar. Quando recebe o sinal SIGINT, a aplicação deve terminar, indicando através de uma mensagem para a saída padrão que recebeu o sinal SIGINT. Considere os seguintes exemplos de execução. As linhas a negrito correspondem à saída padrão produzida pela execução do commando:

**$ ./exec_on_signal -e xpto**
[PID:7851]: command to execute is ‘xpto’
[INFO] kill -s SIGUSR1 7851 (to run command)
[INFO] kill -s SIGINT 7851 (to terminate)
[INFO] SIGUSR1 received. Executing ‘xpto’
**xpto: No such file or directory**
[ERROR] Failed execution: exiting!

**$ ./exec_on_signal --execute date**
[PID:7823]: command to execute is ‘date’
[INFO] kill -s SIGUSR1 7823 (to run command)
[INFO] kill -s SIGINT 7823 (to terminate)
[INFO] SIGUSR1 received. Executing ‘date’
**Sun Nov 5 01:09:19 WET 2017**
[WAITING for signal]
[INFO] SIGUSR1 received. Executing ‘date’
**Sun Nov 5 01:09:21 WET 2017**
[WAITING for signal]
[INFO] SIGINT received. Exiting!

---

# 3.2 Exercícios extra-aula

3. Recorrendo às funções `time()`, `localtime()` e `strftime()`, elabore o programa **whatTimeIsIt** que mostre a data e hora atuais sempre que receber o sinal SIGUSR1. Para efeitos de teste, o programa deverá criar um processo filho responsável por lhe enviar o sinal (SIGUSR1) de 5 em 5 segundos. Ao receber o sinal SIGINT, o processo pai deve matar o processo filho, terminando de seguida.

4. Elabore um programa que mostre o conteúdo de um ficheiro de texto sempre que recebe o sinal SIGHUP. O processo fica em espera bloqueante até que seja terminado pela receção de um sinal não tratado. O nome do ficheiro de texto é indicado através do parâmetro de entrada –f/--file <ficheiro>. Os argumentos de entrada devem ser tratados através do **gengetopt**.

5. Elabore o programa **sonsAndSignals** que crie ***n*** filhos e que, ***t*** segundos após o início da execução do processo, lhes envie o sinal ***s***. Tenha em conta que o número de filhos (***n***), o sinal a passar (***s***) e o número de segundos (***t***) são, respetivamente, o 1º, 2º e 3º argumentos de linha de comando.