Processos

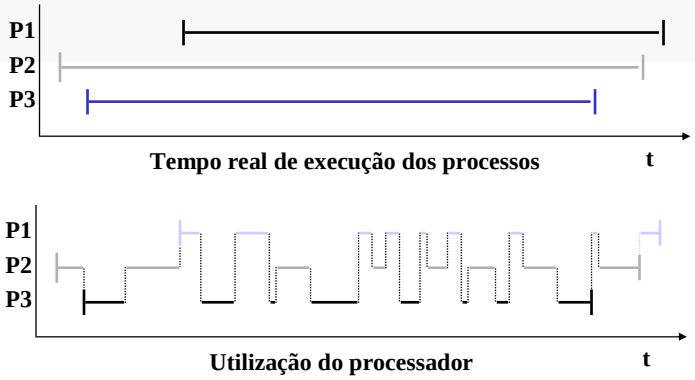
Sistemas Operativos

2008 / 2009

Multiprogramação

- Execução, em paralelo, de múltiplos programas na mesma máquina
- Cada instância de um programa em execução denomina-se um **processo**
- Considerando um grau de tempo fino, o paralelismo não é real
- **Pseudoparalelismo** ou **pseudoconcorrência** – implementação de sistemas multiprogramados sobre um computador com um único processador

Pseudoconcorrência

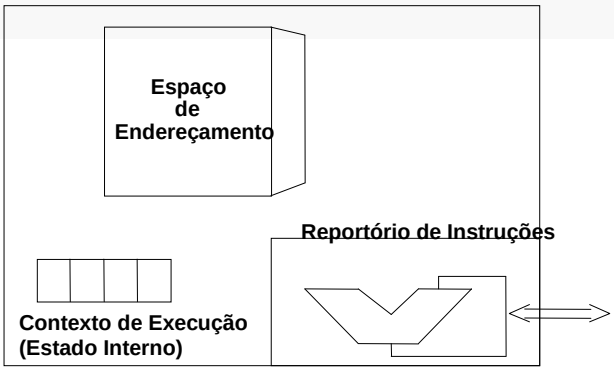


- Analogia com a Física
  - As leis da física quântica determinam a evolução de um sistema mas na maioria dos fenómenos as leis físicas macroscópicas representam adequadamente o seu comportamento
- Analogia com a vida de um estudante do IST?

Processos vs. Programas

- Programa = Fich. executável (sem actividade)
- Um processo é um objecto do sistema operativo que suporta a execução dos programas
- Um processo pode, durante a sua vida, executar diversos programas
- Um programa ou partes de um programa podem ser partilhados por diversos processos (ex.: biblioteca partilhadas as DLL no Windows)

Processo Como Uma Máquina Virtual



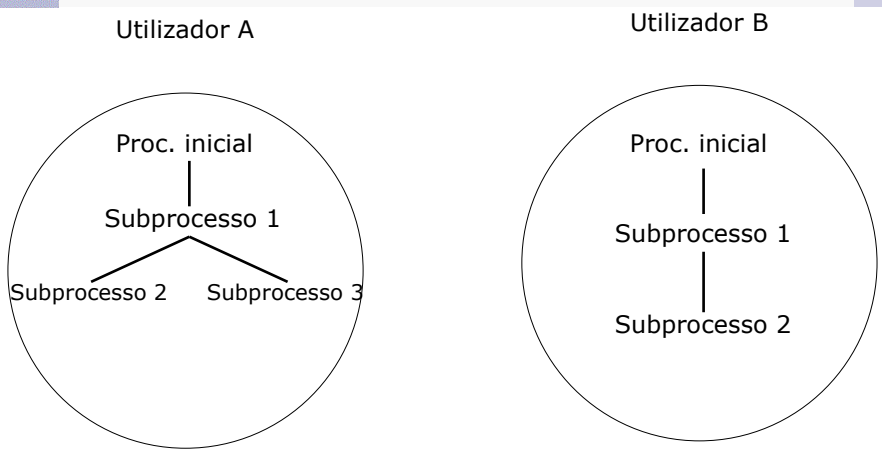
- Elementos principais da máquina virtual que o SO disponibiliza aos processos

Departamento de Engenharia Informática

Processo Como Uma Máquina Virtual

- Tal como um processador um processo tem:
  - **Espaço de endereçamento (virtual):**
    - Conjunto de posições de memória acessíveis
    - Código, dados, e pilha
    - Dimensão variável
  - **Reportório de instruções:**
    - As instruções do processador executáveis em modo utilizador
    - As funções do sistema operativo
  - **Contexto de execução (estado interno):**
    - Valor dos registos do processador
    - Toda a informação necessária para retomar a execução do processo
    - Memorizado quando o processo é retirado de execução

Hierarquia de Processos



- Certas informações são herdadas

Modelo: Objecto "Processo"

- Propriedades
  - Identificador
  - Programa
  - Espaço de Endereçamento
  - Prioridade
  - Processo pai
  - Canais de Entrada Saída, Ficheiros,
  - Quotas de utilização de recursos
  - Contexto de Segurança
- Operações – Funções sistema que actuam sobre os processos
  - Criar
  - Eliminar
  - Esperar pela terminação de subprocesso

Criação de um processo

```
IdProcesso = CriarProc (Código, Prioridade,... )
```

A função tem frequentemente diversos parâmetros: a prioridade, canais de entrada/saída, ...

Quando a criação tem sucesso o sistema atribui um identificador interno (PID) ao processo que é retornado na função

Na criação de um processo tem de ficar definido qual é o programa que o processo vai executar. Normalmente é especificado um ficheiro contendo um programa executável.

Eliminação de processos

- Eliminação do processo quando o seu programa termina, libertando todos os recursos e estruturas de dados detidas pelo processo

  `Sair ([Estado])`

- Eliminação de outro processo

  `EliminarProc ( IdProcesso )`

O processo cujo identificador é passado como parâmetro é eliminado. O núcleo do SO valida se o processo que invoca esta função tem privilégios para a poder executar

Terminação do Processo Filho

- Em numerosas situações o processo pai pode querer bloquear-se esperando a terminação de um processo filho

```
Estado = EsperarTerminacao (Idprocesso)
```

O processo pai pode esperar por um processo específico ou genericamente por qualquer processo

Departamento de Engenharia Informática

Unix – Processos

(Sob o ponto de vista do utilizador)

Processos em Unix

- Identificação de um processo
  - um inteiro designado por PID
  - Alguns identificadores estão pré atribuídos: processo 0 é o *swapper* (gestão de memória) e o processo 1 *init* é o de inicialização do sistema
- Os processos relacionam-se de forma hierárquica
  - O processo herda todo o ambiente do processo pai
  - O processo sabe quem é o processo de que descende designado por processo pai.
  - Quando o processo pai termina os subprocessos continuam a executar-se, são adoptados pelo processo de inicialização (pid = 1)
- Os processos têm prioridades variáveis.
  - Veremos as regras de escalonamento mais adiante.

Departamento de Engenharia Informática

Processos em Unix

- Espaço de endereçamento em modo Utilizador
  - organiza-se em três zonas que no Unix original se designavam por segmentos:
    - texto - código do programa
    - dados - espaço de dados do programa
    - pilha (stack)
- Espaço de endereçamento em modo Núcleo
  - No interior do núcleo existe uma zona de dados para cada processo que contem o seu contexto
  - Uma pilha para execução do processo em modo núcleo.

Processos em Unix

- Cada processo também tem associado um contexto de execução acessível em modo utilizador e que contem diversas variáveis úteis para os programas utilitários ou para as aplicações.
- Exemplo:

```
HOME=/usr/jam
SHELL=/bin/csh
USER=jamarques
PATH=/usr/jam/bin/:/usr/local/bin:/bin
```

- Este contexto é herdado do processo pai e pode ser modificado livremente porque reside no espaço utilizador.
- Nos programas em C é acessível através do parâmetro do main ou de uma variável externa:

```
main (arc, arv, envp)
extern char **environ
```


Criação de um Processo

```
id = fork()
```

A função não tem parâmetros, em particular o ficheiro a executar. A imagem do novo processo é uma cópia da do criador.

O contexto do processo pai é copiado para o filho

A função retorna o PID do processo.

Este parâmetro assume valores diferentes consoante o processo em que se efectua o retorno:

- ao processo pai é devolvido o "pid" do filho
- ao processo filho é devolvido 0
- -1 em caso de erro

Retorno de uma função com valores diferentes → não existente na programação sequencial

Exemplo de fork

```
main() {
    int pid;

    pid = fork();
    if (pid == 0) {
        /* código do processo filho */
    } else {
        /* código do processo pai */
    }

    /* instruções seguintes */
}
```


Terminação do Processo

- Termina o processo, liberta todos os recursos detidos pelo processo, ex.: os ficheiros abertos
- Assinala ao processo pai a terminação

```
void exit (int status)
```

Status é um parâmetro que permite passar ao processo pai o estado em que o processo terminou.

Normalmente um valor negativo indica um erro

Terminação do Processo

- Em Unix existe uma função para o processo pai se sincronizar com a terminação de um processo filho
- Bloqueia o processo pai até que um dos filhos termine

```
int wait (int *status)
```

Retorna o pid do processo terminado. O processo pai pode ter vários filhos sendo desbloqueado quando um terminar

Devolve o estado de terminação do processo filho que foi atribuído no parâmetro da função exit

Exemplo de Sincronização entre o Processo Pai e o Processo Filho

```
main () {
    int pid, estado;

    pid = fork ();
    if (pid == 0) {
        /* algoritmo do processo filho  */
        exit(0);
    } else {
        /* o processo pai bloqueia-se à espera da
           terminação do processo filho  */
        pid = wait (&estado);
    }
}
```

Execução de um Programa

- O **`fork`** apenas permite lançar processo com o mesmo código → problemas?
- A função **`exec`** permite substituir a imagem do processo onde é invocada pela contida num ficheiro executável.
- Não há retorno numa chamada com sucesso.
- Parâmetros: valores que são passados para os parâmetros de entrada na função main do código a executar.
- Os ficheiros mantêm-se abertos.

Departamento de Engenharia Informática

Execução de um Programa

```
int execl(char* ficheiro, arg0, argl,…, argn,0)

int execv(char* ficheiro, *argv [])
```

Caminho de acesso ao ficheiro executável

Argumentos para o novo programa. Podem ser passado como apontadores individuais ou como um array de apontadores. Estas parâmetros são passados para a função main do novo programa e acessíveis através do argv

Exemplo de Exec

```
main ()
{
    int pid;

    pid = fork ();
    if (pid == 0) {
        execl ("/bin/who", "who", 0);
        /* controlo deveria ser transferido para o novo
             programa  */
        printf ("Erro no execl\n");
        exit (-1);
    } else {
        /* algoritmo do proc. pai */
    }
}
```

Por convenção o arg0 é o nome do programa

Shell

- O shell constitui um bom exemplo da utilização de fork e exec (*esqueleto muito simplificado*)

```
while (TRUE){
    prompt();
    read_command (command, params);

    pid = fork ();
    if (pid < 0) {
        printf (“Unable to fork”):
         continue;
    }
    if (pid !=0) {
       wait(&status)
    } else{
       execv (command, params):
    }
}
```

Autenticação

- Um processo tem associados dois identificadores que são atribuídos quando o utilizador efectua o login (se autentica) perante o sistema:
  - o número de utilizador UID - *user identification*
  - o número de grupo GID - *group identification*
- Os UID e GID são obtidos do ficheiro **`/etc/passwd`** no momento do *login*
- O UID e o GID são herdados pelos processos filhos
- *superuser* é um UID especial – zero. Normalmente está associado ao utilizador root (privilegiado).

Protecção no Acesso aos Recursos

- A protecção dos recursos em Unix é uma versão simplificada do modelo de Listas de Controlo de Acesso (ACL)
- Para um recurso (ficheiro, socket, etc.) a protecção é definida em três categorias:
  - Dono (*owner*): utilizador que normalmente criou o recurso
  - Grupo (*group*): conjunto de utilizadores com afinidades de trabalho que justificam direitos semelhantes
  - Restantes utilizadores (*world*)

SetUID

- Mecanismo de Set UID (SUID) – permite alterar dinamicamente o utilizador
- Duas variantes: bit de setuid, ou função sistema setuid

Bit SetUID

- No ficheiro executável pode existir uma indicação especial que na execução do exec provoca a alteração do uid
- O processo assume a identidade do dono do ficheiro durante a execução do programa.
- Exemplo: comando **`passwd`**
- Operação crítica para a segurança

Funções Sistema de identificação

- *Real UID e GID* – UID e GID originais do processo
- *Effective UID e GID* – usado para verificar permissões de acesso e que pode ter sido modificado pelo setuid

`getpid()` - devolve a identificação do processo
`getuid(), getgid()`
devolvem a identificação real do utilizador
`geteuid(), getegid()`
devolvem a identificação efectiva do utilizador
`setuid(uid), setgid(gid)`
altera a identificação efectiva do utilizador para uid e gid
só pode ser invocada por processos com privilégio de superutilizador

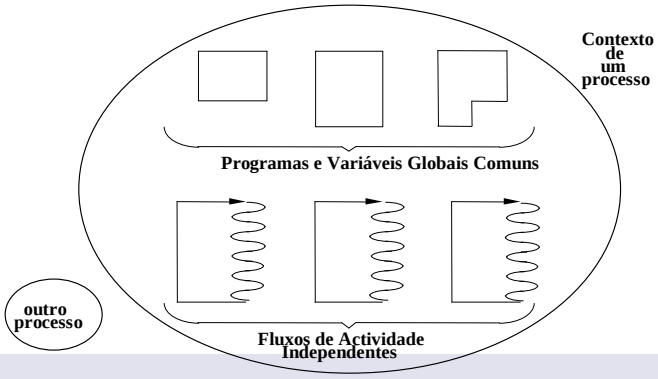
Departamento de Engenharia Informática

Tarefas (Threads)

Múltiplos fluxos de execução no mesmo processo

Tarefas

- Mecanismo simples para criar fluxos de execução independentes, partilhando um contexto comum



Tarefas vs. Processos

- Porque não usar processos?
  - Processos obrigam ao isolamento (espaços de endereçamentos disjuntos) → dificuldade em partilhar dados (mas não impossível… exemplos?)
  - Eficiência na criação e comutação

Tarefas: Exemplos de Utilização

- Servidor (e.g., web)
- Aplicação cliente de correio electrónico
- Quais as tarefas em cada caso?

Modelos Multitarefa no Modelo Computacional

- Operações sobre as Tarefas

```
IdTarefa = CriarTarefa(procedimento);
```

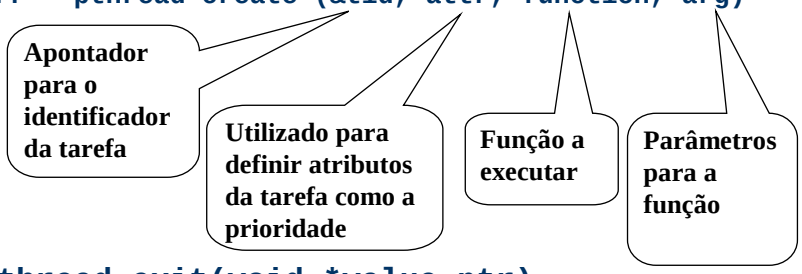
A tarefa começa a executar o procedimento dado como parâmetro e que faz parte do programa previamente carregado em memória

```
EliminarTarefa (IdTarefa);
EsperaTarefa (IdTarefa)
```

Bloqueia a tarefa à espera da terminação de outra tarefa ou da tarefa referenciada no parâmetro Idtarefa

Interface POSIX

```
err = pthread-create (&tid, attr, function, arg)
```




```
pthread_exit(void *value_ptr)
```

```
int pthread_join(pthread_t thread, void **value_ptr)
```

- se a tarefa alvo não terminou a tarefa bloqueia-se

Exemplo sequencial

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];

void *soma_coluna (void *ptr) {
  int indice = 0, soma=0;
  int *b = (int *) ptr;
  while (indice < TAMANHO - 1)
    /* soma col.*/
    soma += b[indice++];

  /* soma->ult.col.*/
  b[indice]=soma;
  return NULL;
}
```

```
int main (void) {
  int i,j;

  for (i=0; i<N; i++){
    for (j=0; j< TAMANHO - 1; j++)
      buffer[i] [j] =rand()%10;
  }
  for (i=0; i< N; i++)
    soma_coluna((void *) buffer[i]);

  for(i=0; i< N; i++) {
    printf ("soma da linha %d: %3d \n",
      i, buffer [i][TAMANHO-1]);
   }
   exit(0);
}
```

Exemplo com threads

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 5
#define TAMANHO 10

int buffer [N] [TAMANHO];

void *soma_coluna (void *ptr) {
  int indice = 0, soma=0;
  int *b = (int *) ptr;
  while (indice < TAMANHO - 1)
    /* soma col.*/
    soma += b[indice++];

  /* soma->ult.col.*/
  b[indice]=soma;
  return NULL;
}
```

```
int main (void) {
  int i,j;
  pthread_t tid[N];

  for (i=0; i<N; i++){
    for (j=0; j< TAMANHO - 1; j++)
      buffer[i] [j] =rand()%10;
  }

  for (i=0; i< N; i++){
    if (pthread_create (&tid[i], 0,soma_coluna,
                        (void *) buffer[i])!= 0) {
      printf("Erro na criação da tarefa\n");
      exit(1);
    } else {
      printf ("Criada a tarefa %d\n", tid[i]);
    }
  }
  for (i=0; i<N; i++){
    pthread_join (tid[i], NULL);
  }
  printf ("Terminaram todas as threads\n");

  for(i=0; i< N; i++) {
    printf ("soma da linha %d: %3d \n",
            i, buffer [i][TAMANHO-1]);
  }
  exit(0);
}
```


Tarefas em Posix e Windows

Tarefas	Criar	Sincronizar com a Terminação	Transferir Controlo	Transferir/ Adormecer	Terminar
POSIX	pthread_create	pthread_join	pthread_yield	sleep	pthread_exit
Windows	CreateThread	WaitForSingleObject	SwitchToThread	Sleep	ExitThread

Programação num ambiente multitarefa

- As tarefas partilham o mesmo espaço de endereçamento e portanto têm acesso às mesmas variáveis globais.
- A modificação e teste das variáveis globais tem de ser efectuada com precauções especiais para evitar erros de sincronização.
- Veremos no cap. 4 a forma de resolver estes problema com objectos de sincronização.

Alternativas de Implementação

- Tarefas-núcleo
- Tarefas-utilizador (pseudotarefas)
  - Projecto da cadeira

Pseudotarefas (Tarefas-Utilizador)

- As tarefas implementadas numa biblioteca de funções no espaço de endereçamento do utilizador.
- Ideia proveniente das linguagens de programação.
- Núcleo apenas "vê" um processo.
- Processo guarda lista de tarefas, respectivo contexto

Departamento de Engenharia Informática

Pseudotarefas (Tarefas-Utilizador)

- A comutação entre tarefas explícita → função **`thread-yield`**
  - Pode ser contornado usando interrupções ("preempção")
- Problema: e se uma tarefa faz chamada bloqueante?
- Solução?

Tarefas-Núcleo (ou Tarefas Reais)

- Implementadas no núcleo do SO
  - Mais comuns
- Lista de tarefas e respectivo contexto são mantidos pelo núcleo

Comparação Tarefas Utilizador e Núcleo

- Capacidade de utilização em diferentes SOs?
- Velocidade de criação e comutação? (vs. processos?)
- Tirar partido de execução paralela em multiprocessadores?
- Aproveitamento do CPU quando uma tarefa bloqueia (ex: ler do disco)?

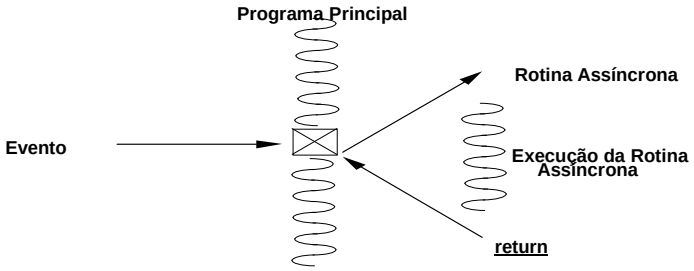
Eventos

Rotinas Assíncronas para Tratamento de acontecimentos assíncronos e excepções

Rotinas Assíncronas

- Certos acontecimentos devem ser tratados pelas aplicações, embora não seja possível prever a sua ocorrência
  - Ex: Ctrl-C
  - Ex: Acção desencadeada por um timeout
- Como tratá-los na programação sequencial?
- Poder-se-ia lançar uma tarefa por acontecimento. Desvantagem?
- Alternativa: Rotinas assíncronas associadas aos acontecimentos (**eventos**)

Modelo de Eventos



- Semelhante a outro conceito...

Rotinas Assíncronas

```
RotinaAssincrona (Evento,Procedimento)
```

Tem de existir uma tabela com os eventos que o sistema pode tratar

Identificação do procedimento a executar assincronamente quando se manifesta o evento.

Signals – Acontecimentos Assíncronos em Unix

Signal	Causa
SIGALRM	O relógio expirou
SIGFPE	Divisão por zero
SIGINT	O utilizador carregou na tecla para interromper (normalmente o CNTL-C
SIGQUIT	O utilizador quer terminar o processo e provoca
SIGKILL	Signal para terminar o processo não pode ser tratado
SIGPIPE	O processo escreveu para um pipe que não tem receptores
SIGSEGV	Acesso a uma posição de memória inválida
SIGTERM	O utilizador pretende terminar ordeiramente o proc
SIGUSR1	Definido pelo utilizador
SIGUSR1	Definido pelo utilizador

Excepção

Interacção com o terminal

Desencadeado por interrupção HW

Explicitamente desencadeado por outro processo

- Definidos em **`signal.h`**

Departamento de Engenharia Informática

SIGHUP	Terminal desligado.
SIGINT	Interrupção do terminal (normalmente gerado pelas teclas 'Ctrl+C').
SIGQUIT	Abortar a execução.
SIGILL	Instrução ilegal.
SIGTRAP	Utilizado pelo depurador de erros (*debugger*).
SIGFPE	Erro de operação de vírgula flutuante (*overflow*, divisão por zero).
SIGKILL	Termina incondicionalmente o processo.
SIGBUS	Erro do *bus* (acesso à memória com alinhamento não permitido).
SIGSEGV	Violação de endereçamento.
SIGSYS	Erro num argumento de uma chamada ao sistema.
SIGALRM	Temporização para implementação de alarme.
SIGTERM	Terminação ordenada do processo.
SIGUSR1	Definido pelo utilizador.
SIGUSR2	Definido pelo utilizador.

Interacção com o terminal

Desencadeado por interrupção HW

Excepção

Explicitamente desencadeado por outro processo

Tratamento dos Signals

- Por omissão – termina o processo.
- Ignorado – Alguns signals como o SIGKILL não podem ser ignorados. Porquê?
- Associado a uma rotina de tratamento (handler) através da chamada à função sistema **`signal`**

Chamada Sistema "Signal"

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

A função retorna um ponteiro para função anteriormente associada ao signal

Identificador do signal para o qual se pretende definir um handler

Ponteiro para a função ou macro especificando:
- **SIG_DFL – acção por omissão**
- **SIG_IGN – ignorar o signal**

Parâmetro para a função de tratamento

Exemplo do tratamento de um Signal

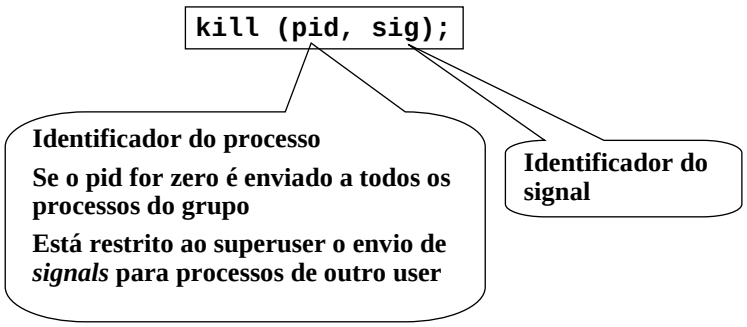
```
#include <stdio.h>
#include <signal.h>

apanhaCTRLC () {
   char ch;
   printf (“Quer de facto terminar a execucao?\n”);
   ch = getchar();
   if (ch == ‘s’) exit(0);
   else {
      printf (“Entao vamos continuar\n”);
      signal (SIGINT, apanhaCTRLC);
   }
}

main () {
   signal (SIGINT, apanhaCTRLC);
   printf("Associou uma rotina ao signal SIGINT\n");
   for (;;)
     sleep (10);
}
```


Chamada Sistema Kill

- Envia um signal ao processo
- Nome enganador. Porquê?



Outras funções associadas aos signals

- **`unsigned alarm (unsigned int segundos);`**
  - o *signal* SIGALRM é enviado para o processo depois de decorrerem o número de segundos especificados. Se o argumento for zero, o envio é cancelado.
- **`pause();`**
  - aguarda a chegada de um *signal*
- **`unsigned sleep (unsigned int segundos);`**
  - A função faz um alarm e bloqueia-se à espera do signal

Função System

```
#include <signal.h>

int system(char *argv[]) {
   int pid, status;
   void (*del) (), (*quit) (); /* variaveis (ponteiros para f.) */

   del = signal (SIGINT, SIG_IGN);
   quit = signal (SIGQUIT, SIG_IGN);
   switch (pid = fork ()) {
      case 0 : signal (SIGINT, del);
               signal (SIGQUIT, quit);
               execl ("/bin/sh", "sh", "-c", argv, 0);
               exit (-1);
      case -1: /* ... */
      default: while (wait(&status) != pid);
   }
   signal (SIGINT, del);
   signal (SIGQUIT, quit);
}
```