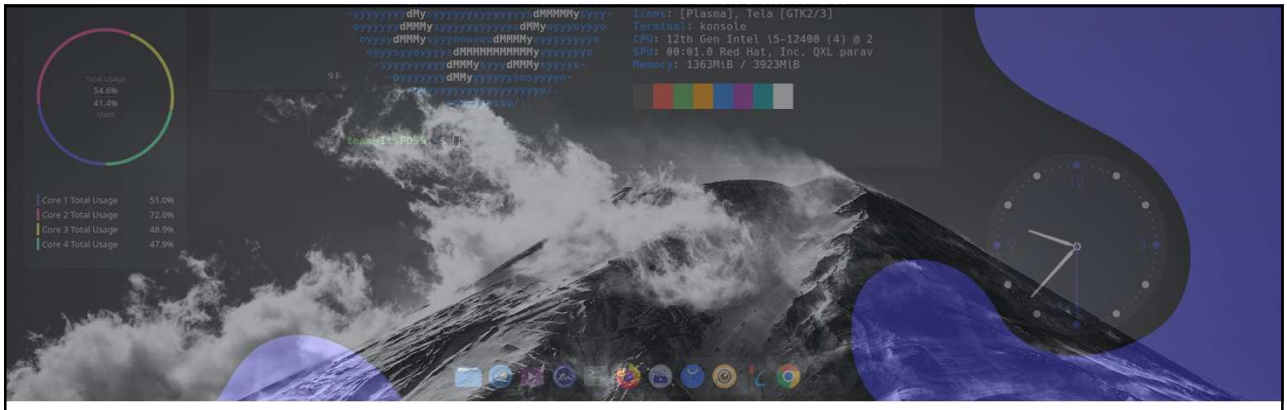


SISTEMAS OPERATIVOS

Processos e tarefas

António Godinho

1

PROCESSOS E PROGRAMAS

O termo "processo" foi introduzido nos anos 60, pelos autores do SO Multics.

Algumas definições:

- Programa em execução;
- Recurso lógico do sistema operativo que suporta a execução de um programa;
- Fluxo de actividade autónomo que executa as acções definidas num programa.

Do ponto de vista conceptual, um processo pode ser considerado como uma máquina virtual que executa as diversas etapas de actividade definidas pelo algoritmo explicitado num programa.

2

2

PROCESSOS E PROGRAMAS

- Durante o seu tempo de vida, um processo evolui por diferentes fases, desde a sua **criação**, **execução** continuada, até que **termina** a sua atividade quando a função para que tiver sido criado tenha sido realizada. Durante a fase de execução, o processo poderá ficar **suspenso** ou **bloqueado**, eventualmente à espera que ocorra algum evento externo.

- Um **programa**, enquanto entidade armazenada num ficheiro em disco (normalmente referido por ficheiro executável), não desenvolve qualquer tipo de atividade. É uma **entidade passiva**.

3

3

PROCESSOS E PROGRAMAS

- O **mesmo programa** pode ser executado por **diferentes processos**.
- Um **processo** pode, ao longo do seu tempo de vida, executar uma sequência de **vários programas**.
- Um **programa** ou secções de um programa podem ser **partilhados** por **vários processos** (Ex. DLL do Windows)
- Ao receber um pedido de execução de um programa, o **SO cria um processo** e **aloca os recursos** necessários para a sua execução, nomeadamente o **espaço em memória** para armazenamento dos dados e do código do programa, conferindo-lhe, ainda, a possibilidade de utilização da CPU.

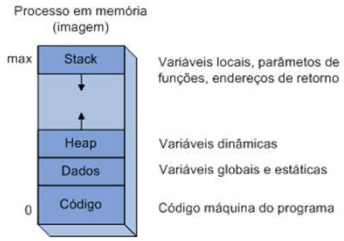
4

4

PROCESSO ENQUANTO MÁQUINA VIRTUAL

Máquina virtual constituída por:

1. Espaço de endereçamento próprio (isolado dos restantes processos): Posições de memória organizadas em várias secções (designado por imagem na terminologia Unix):
   - **Secção de código**: armazena o código do programa a ser executado pelo processador (text na terminologia Unix).
   - **Secção de dados**: armazena as variáveis globais e as variáveis estáticas do programa.
   - **Secção de heap**: armazena as variáveis criadas dinamicamente durante a execução do programa.



5

5

PROCESSO ENQUANTO MÁQUINA VIRTUAL

▪Máquina virtual constituída por:

- Secção de pilha (stack): armazena dados temporários como variáveis locais e endereços de retorno de chamadas a sub-rotinas. A pilha cresce quando o programa invoca uma sub-rotina e decresce quando a sub-rotina termina e o programa regressa ao ponto de invocação.

2. Conjunto das instruções do processador executáveis em modo utilizador e funções do SO.

3. Contexto de execução: estado do processo, valores dos registos do processador, informação sobre recursos E/S alocados, informação relacionada com a gestão de memória, etc.

A máquina virtual é o suporte da execução do programa. Estende a máquina física porque disponibiliza as funções do sistema, mas restringe-a, confinando o espaço de endereçamento e impedindo a execução de determinadas operações restritas ao SO.

6

6

DESCRITORES DE PROCESSOS

✓Para suporte da atividade de gestão de processos, o SO mantém uma estrutura de dados com informações relativas a cada processo (PCB Process Control Block), onde armazena:

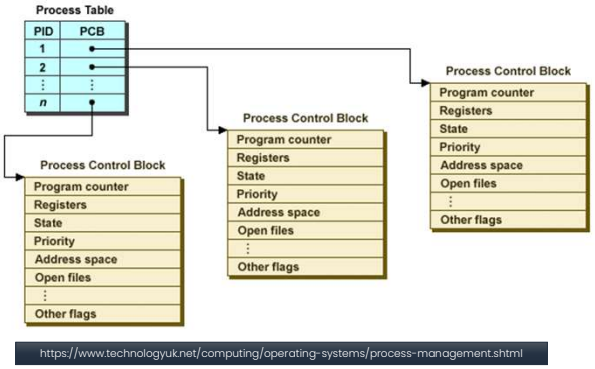
- Identificador do processo (PID - Process Identification Number);
- Estado do processo;
- Program Counter (registo com o endereço da próxima instrução a executar) e restantes registos do processador (acumuladores, registos de índices, ponteiro de pilha, registos de flags), normalmente referido por contexto de execução;
- Prioridade de escalonamento;
- Informação sobre o espaço de endereçamento em memória;
- Informação sobre recursos alocados (ex. ficheiros abertos, dispositivos E/S);
- Credenciais do processo (ex. utilizador, permissões, etc.) permitindo saber quais os recursos a que o processo pode aceder e o correspondente modo de acesso (contexto de segurança);
- Processo pai (processo que criou este processo);
- Lista de processos filho (processos criados por este processo);

7

7

DESCRITORES DE PROCESSOS

- Esta estrutura de dados é armazenada no espaço de endereçamento do kernel do SO (não acessível aos processos dos utilizadores).
- Operações sobre processos realizadas pelo SO: Criar, destruir, suspender, reactivar, alterar prioridade, bloquear, desbloquear, despacho.



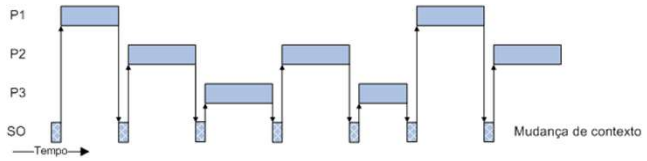
https://www.technologyuk.net/computing/operating-systems/process-management.shtml

8

8

GESTOR DE PROCESSOS DO SO

Suporte à multiprogramação: execução de múltiplos processos em "paralelo" (pseudo-paralelismo ou pseudo-concorrência).



Responsável por:

- Criar e eliminar processos;
- Despacho: alocação do processador a um processo, garantindo a salvaguarda dos contextos de execução de cada processo (mudança de contexto);
- Escalonar a execução dos processos, de acordo com algoritmos (de escalonamento) que garantam o completamento, a não privação e que, simultaneamente, atendam a diferentes prioridades associadas aos processos;
- Alterar o estado dos processos;
- Garantir o isolamento adequado dos espaços de endereçamento de cada processo.

9

9

10

PROCESSOS EM LINUX

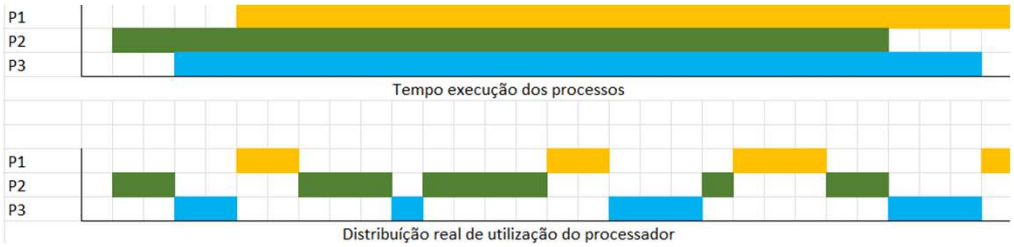
10

PSEUDO-PARALELISMO

Consiste em ter várias actividades (programas/processos) a decorrer ao mesmo tempo do ponto de vista macroscópico

Aplica-se a sistemas multiprogramados (tempo partilhado ou não)

Decorrer do tempo visto pelos processos (P1, P2 e P3 representados com cores diferentes)

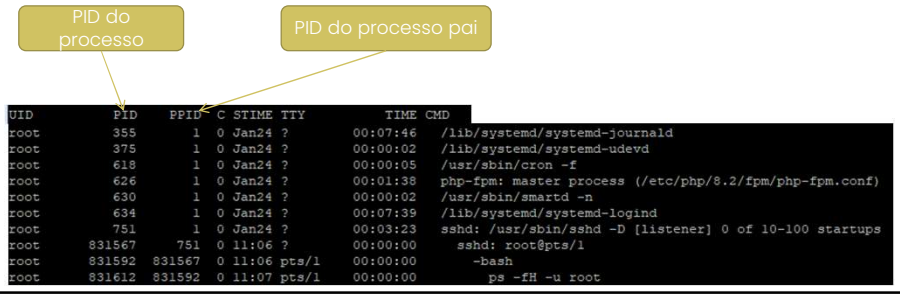


11

11

PROCESSOS EM LINUX

Criação hierarquizada de processos: um processo (pai) cria um novo processo (filho). O processo filho herda o contexto de execução do processo pai no momento da criação (duplicação da imagem do processo, à excepção da secção de código que é partilhada pelos dois processos). Nota: Na realidade, a secção de dados não é de imediato duplicada, apenas se e quando o processo filho alterar esta zona da memória, é feita a necessária duplicação (modelo copy-on-write).



12

12

PROCESSOS EM LINUX

Processos em execução: Durante a inicialização, o processo init (PID=1) lança os restantes processos do SO (atualmente o proceso systemd).

```
root      564246       1  0 Feb05 ?        00:00:23 /usr/sbin/apache2 -k start
www-data  811254  564246  0 00:00 ?        00:00:10  \_ /usr/sbin/apache2 -k start
www-data  811257  564246  0 00:00 ?        00:00:08  \_ /usr/sbin/apache2 -k start
www-data  811262  564246  0 00:00 ?        00:00:07  \_ /usr/sbin/apache2 -k start
www-data  812448  564246  0 00:44 ?        00:00:07  \_ /usr/sbin/apache2 -k start
www-data  821275  564246  0 05:48 ?        00:00:07  \_ /usr/sbin/apache2 -k start
www-data  821295  564246  0 05:49 ?        00:00:06  \_ /usr/sbin/apache2 -k start
www-data  821303  564246  0 05:49 ?        00:00:05  \_ /usr/sbin/apache2 -k start
www-data  828092  564246  0 09:06 ?        00:00:05  \_ /usr/sbin/apache2 -k start
www-data  828107  564246  0 09:06 ?        00:00:05  \_ /usr/sbin/apache2 -k start
www-data  829424  564246  0 09:49 ?        00:00:02  \_ /usr/sbin/apache2 -k start
root      831571       1  0 11:06 ?        00:00:00 /lib/systemd/systemd --user
root      831572  831571  0 11:06 ?        00:00:00  \_ (sd-pam)
```

13

13

CRIAÇÃO DE PROCESSOS EM LINUX

função de sistema fork()

- O SO cria um novo processo e duplica a imagem do processo pai (processo que invoca função de sistema fork()), à excepção da secção de código que é partilhada pelos processos. O processo filho herda o contexto de execução do processo pai.
- A função devolve:
  - No processo pai, o PID do processo filho ou o valor -1 em caso de erro;
  - No processo filho, o valor 0;

```c
#include <stdio.h>
main () {
        int id;
        if (id=fork()) { // Processo pai
                wait(); // Aguarda fim do processo filho
                printf("PP:O meu PID=%d\n", getpid());
                printf("PP:Id do processo filho=%d\n",id);
        } else { // Processo filho
                printf("PF: Id no processo filho=%d\n",id);
                printf("PF:0 meu PID=%d\n", getpid());
                printf("PF:0 PID do meu pai=%d\n",getppid());
        }
}
```

```
root@gandalf:~# ./pids
PF: Id no processo filho=0
PF:0 meu PID=832021
PF:0 PID do meu pai=832020
PP:O meu PID=832020
PP:Id do processo filho=832021
root@gandalf:~#
```

getpid() – devolve o PID do processo

getppid() – devolve o PID do processo pai

14

14

ISOLAMENTO DO ESPAÇO DE ENDEREÇAMENTO

O processo filho herda uma cópia integral da secção de dados do processo pai. Depois de criado o processo filho, os espaços de endereçamento estão isolados.

```c
#include <stdio.h>
main () {
        int i=10;
        if (fork()) { // Processo pai
                printf("PP:Valor inicial i=%d\n",i);
                i=i+1;
                printf("PP:Valor actualizado_i=%d\n",i);
        } else { // Processo filho
                printf("PF: Valor inicial_i=%d\n",i); i=i+2;
                printf("PF: Valor actualizado_i=%d\n",i);
        }
        printf("Codigo comum i=%d\n",i);
}
```

```
root@gandalf:~# ./pids2
PP:Valor inicial i=10
PP:Valor actualizado_i=11
Codigo comum i=11
PF: Valor inicial_i=10
PF: Valor actualizado_i=12
Codigo comum i=12
root@gandalf:~#
```

15

15

HERANÇA DE DESCRITORES DE FICHEIROS ABERTOS

O processo filho herda os descritores de ficheiros abertos.

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
        int fd;
        char ch;
        fd=open("test.file", O_RDONLY);
        if (fork()) { // Processo pai
                printf("PP-PID: %d\n",getpid());
                wait(NULL);
        } else { // Processo filho
                printf("PF-PID: %d\n",getpid());
                read (fd, &ch,1);
                printf("PF: Leitura do ficheiro: %c\n",ch);
                printf("PF: Encerra ficheiro\n");
                close (fd);
        }
        if (read (fd, &ch, 1) < 0) {
                printf("PID DO processo que falha: %d\n",getpid());
                perror("READ falhou\n");
                exit(1);
        }
        printf("Leitura do ficheiro: %c - PID: %d \n",ch, getpid());
        return 0;
}
```

```
root@gandalf:~/SO# ./pids3
PP-PID: 1136820
PF-PID: 1136821
PF: Leitura do ficheiro: A
PF: Encerra ficheiro
PID DO processo que falha: 1136821
READ falhou
: Bad file descriptor
Leitura do ficheiro: B - PID: 1136820
root@gandalf:~/SO#
```

16

16

CARREGAMENTO DE UMA NOVA IMAGEM

Através da função execl(), uma nova imagem é criada para receber o novo programa substituindo a actual imagem do processo que invocou a função. Em consequência, o código existente depois da chamada da função execl() nunca será executado

```
#include <stdio.h>
#include <unistd.h>
main() {
        printf("Proc. Pai\n");
        if (fork()==0) { // Processo filho
                printf("Carregamento de uma nova imagem\n");
                execl("/bin/ls","ls", NULL);
                printf("Este codigo nunca sera executado\n");
        } else { // Processo pai
                wait();
                printf("Terminou o processo filho\n");
        }
}
```

execl() – substitui a imagem do processo por uma nova imagem correspondente ao programa a ser carregado (ex. ls)

```
root@gandalf:~/SO# ./execl
Proc. Pai
Carregamento de uma nova imagem
execl  pids  pids2  pids2.c  pids3  pids3.c  pids4.c  pids.c  test.file
Terminou o processo filho
root@gandalf:~/SO#
```

17

17

CARREGAMENTO DE UMA NOVA IMAGEM

Através da função execl(), uma nova imagem é criada para receber o novo programa substituindo a actual imagem do processo que invocou a função. Em consequência, o código existente depois da chamada da função execl() nunca será executado

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
main(int argc, char *argv[]){
        int fd, status, pid;
        if (fork()) { // Processo pai
                pid=wait(&status);
                printf("Proc. filho %d terminou com %d\n",pid, status>>8);
        } else { // Processo filho
                if ((fd=open(argv[1],O_RDONLY)) !=-1) {
                        close(fd);
                        exit(0);
                }
        else exit(1);
        }
}
```

Só interessam os 8 bits + significativos

```
root@gandalf:~/SO# ./pids5 test.file
Proc. filho 835185 terminou com 0
root@gandalf:~/SO# ./pids5 naoexiste
Proc. filho 835189 terminou com 1
root@gandalf:~/SO#
```


18

18

PROCESSO "ZOMBIE"

O processo pai não se "preocupa" com a terminação do processo filho. Este termina, ficando no estado "zombie". Não voltará a ser executado. O SO mantém-no na lista de processos para, eventualmente mais tarde, o processo pai poder executar o wait().

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
main(int argc, char *argv[]) {
        int id;
        if (id=fork()) { // Processo pai
                printf("PP:Nao faz wait() do proc. filho\n");
                printf("PP:Em ciclo infinito!!!\n");
                while (1);
        }
        else { // Processo filho
                printf("PF: Vou terminar, mas fico em estado 'zombie'\n");
        }
}
```

```
root@gandalf:~/SO#
root@gandalf:~/SO# nano -w pids6.c
root@gandalf:~/SO# ./pids6
PP:Nao faz wait() do proc. filho
PP:Em ciclo infinito!!!
PF: Vou terminar, mas fico em estado 'zombie'
```

```
root         751       1  0 Jan24 ?        00:03:24   sshd: /usr/sbin/sshd -D [listener] 1 of 10-100 startups
root      831567     751  0 11:06 ?        00:00:01     sshd: root@pts/1
root      831592  831567  0 11:06 pts/1    00:00:00       -bash
root      835303  831592 99 12:48 pts/1    00:01:01         ./pids6
root      835304  835303  0 12:48 pts/1    00:00:00           [pids6] <defunct>
root      835290     751  0 12:48 ?        00:00:00     sshd: [accepted]
root      835318     751  0 12:49 ?        00:00:00     sshd: root@pts/2
root      835328  835318  0 12:49 pts/2    00:00:00       -bash
root      835338  835328  0 12:49 pts/2    00:00:00         ps -fH -u root
```

19

19

PROCESSO ÓRFÃO

O processo pai não aguarda pela terminação do processo filho. O processo pai termina e o processo filho fica "órfão", ou melhor, é adoptado pelo processo Init (PID=1).

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
main(int argc, char *argv[]){
        int id;
        if (id=fork()) { // Processo pai
                printf("PP: Nao faz wait() do proc. filho\n");
                printf("PP: Vou terminar!\n");
        } else { // Processo filho
                printf("PF: Vou aguardar pela terminacao do proc. Pai\n");
                while(getppid() !=1);
                printf("PF: Fui adoptado por PID Pai=%d\n", getppid()); while (1);
        }
}
```

```
root@gandalf:~/SO# ./pids7
PP: Nao faz wait() do proc. filho
PP: Vou terminar!
PF: Vou aguardar pela terminacao do proc. Pai
PF: Fui adoptado por PID Pai=1
root@gandalf:~/SO#
```

```
root      831571       1  0 11:06 ?        00:00:00   /lib/systemd/systemd --user
root      831572  831571  0 11:06 ?        00:00:00     (sd-pam)
root      835471       1 99 12:53 pts/1    00:00:38   ./pids7
```

20

20

21

TAREFAS (THREADS)

21

TAREFAS (THREADS)

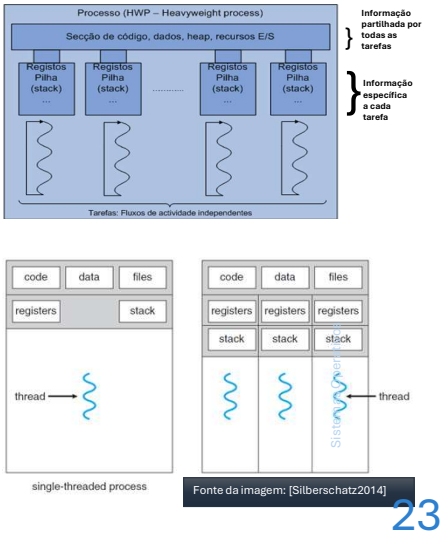
- Fluxos de execução independentes e concorrentes que executam o mesmo programa no mesmo espaço de endereçamento do processo.
- Tarefa (thread), também designado por LWP- lightweight process, em oposição a um processo (HWP -heavyweight process).
- As tarefas não podem existir sem estarem associadas a um processo (processo multitarefa). Existem apenas no contexto de um processo. Partilham a secção de código, secção de dados, secção de heap, recursos E/S, mas cada tarefa tem o seu próprio conjunto de registos, uma secção de pilha (stack) e outros dados específicos a cada tarefa (tratamento de sinais e exceções). Quando um processo termina, todas as tarefas associadas são eliminadas.
- A função main() do processo é uma das tarefas do processo.
- A criação de uma tarefa e a mudança de contexto entre tarefas exigem um menor esforço computacional, comparativamente às mesmas atividades para os processos.
- Por partilharem a mesma secção de dados e heap, a comunicação entre tarefas do mesmo processo não necessita de recorrer ao kernel do SO. No entanto, esta facilidade exige uma maior complexidade na programação dos mecanismos de comunicação e sincronização entre tarefas.

22

22

TAREFAS (THREADS) – DEFINIÇÃO

- **A** thread **é uma unidade básica de utilização do CPU**
- **Cada** thread **tem de forma exclusiva**
  - thread id, program counter**, valores dos registos do CPU e pilha**
- **As várias** threads **de um processo partilham**
  - **secção de dados, secção de texto (código) e outros recursos do SO como ficheiros e sinais**
- **Se um processo tem várias threads** (multithreading**) pode efetuar várias tarefas simultaneamente, como por exemplo**
  - **Browser com uma** thread **a mostrar texto e imagens e outra a fazer download de dados**
  - **Processador de texto com uma** thread **para mostrar imagem, outra para interagir com utilizador e ainda outra para verificar se há erros gramaticais**



Fonte da imagem: [Silberschatz2014]

23

BENEFÍCIOS

- Porquê criar uma thread e não um novo processo?
  - razões de desempenho (criar processo demora + tempo e é + "pesado") e eficiência (processo necessita de + recursos)
- Vantagens relacionadas
  - Responsividade: o programa continua a correr mesmo que alguma das tarefas estejam bloqueadas/lentas, permitindo que se mantenha interacção com o utilizador
  - Partilha de recursos: as técnicas de comunicação entre processos (memória partilhada, troca de mensagens) têm de ser criadas explicitamente, mas as threads partilham as variáveis globais, facilitando a comunicação
  - Economia: as threads são mais "leves", sendo mais económico criar novas threads e mudar de contexto entre threads
  - Escalabilidade: em sistemas multiprocessador, várias threads podem estar a correr simultaneamente

24

TAREFAS (THREADS)- MODELO MULTITAREFA

Processo monotarefa

```
for(k = 0; k < n; k++)
     a[k] = b[k] * c[k] + d[k]  * e[k];
```

Processo multitarefa (vantajoso apenas em sistemas multiprocessador)

```
Cria_Tarefa (fn, 0, n/2);
Cria_Tarefa (fn, n/2, n);
fn(j,m) {
     for(k = j; k < m; k++)
          a[k] = b[k] * c[k] + d[k] * e[k];
}
```

Processo monotarefa

```
Servidor_web()
  enquanto
    aguarda_pedido_cliente(pedido)
    obtem_URL_disco(pedido)
    controi_resposta()
    envia_resposta ()
  fim_enquanto
```

Processo multitarefa

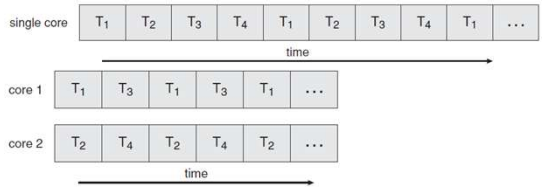
```
Servidor_web()
  enquanto
    aguarda_pedido_cliente(pedido)
    Cria_Tarefa(Trata(pedido))
  fim_enquanto

Trata (pedido)
  obtm_URL_disco(pedido)
  controi_resposta()
  envia_resposta()
```

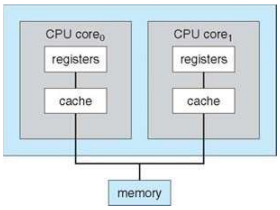
25

25

PROGRAMAÇÃO MULTICORE

- Sistema multicore: sistemas com vários CPUs (cores) num único chip
- Nestes sistemas as threads podem correr em paralelo…





Fonte da imagem: [Silberschatz2014]

- …o que traz imensos desafios para os programadores
  - Como dividir as tarefas?
  - Como balancear o trabalho?
  - Como distribuir/dividir os dados?
  - Como sincronizar as tarefas?
  - Como fazer o teste e debugging?

26

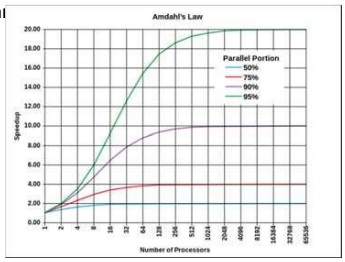
26

LEI DE AMDAHL

- Como a maior parte dos programas têm componentes sequenciais e componentes paralelizáveis, então:

O ganho em performance de um programa que use múltiplos processadores é limitado pela fração do programa que é sequencial

- Exemplo:
  - A **Lei de Amdahl**, **numérica e graficamente**:



$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S: fração sequencial
N: número de processadores

Fonte da imagem: [Wikipedia]

  - Programa demora 20 horas a correr num computador com apenas um core. Tarefa sequencial demora uma hora (5% sequencial).
  - Por mais cores que tenhamos, precisamos de pelo menos 1 hora para terminar o trabalho. A melhoria nunca será maior do que 20x!

27

27

TIPOS DE THREADS

- Threads de nível utilizador
  - geridas por bibliotecas de nível utilizador
  - Principais exemplos: POSIX Pthreads, Win32 threads, Java threads
- Threads do kernel
  - geridas pelo sistema operativo
  - Praticamente todos os sistemas atuais têm destas threads (Windows, Linux, Mac OS X, etc.)
- Os modelos multithreading relacionam as threads nível utilizador com as threads do kernel
  - muitas-para-uma (many-to-one)
  - uma-para-uma (one-to-one)
  - muitas-para-muitas (many-to-many)
    - variante dois-níveis (two-level)

28

28

MODELO MULTITAREFA: MANY-TO-ONE

- Tarefas ao nível do utilizador (user-level threads): Tarefas implementadas através de uma biblioteca de funções no espaço de endereçamento do utilizador. A gestão de tarefas (criação, eliminação, escalonamento, despacho e comutação entre tarefas) é realizada por funções que executam no espaço do utilizador.

- Para o SO, um processo multitarefa neste modelo é um único processo ao qual está associado um único contexto de execução. O SO desconhece que existem várias tarefas implementadas no processo. O tempo de processador alocado a este processo é, internamente, distribuído pelas várias tarefas.

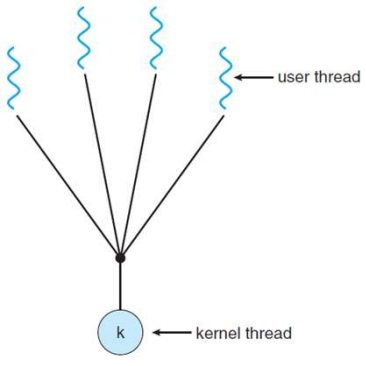
- O programador tem de, explicitamente, proceder à comutação de tarefas (multitarefa cooperativa). Por não recorrer aos serviços do SO para implementar o ambiente multitarefa, é maior a portabilidade das aplicações. Por outro lado, o programador tem total controlo no escalonamento e despacho das tarefas, permitindo a adequação destas atividades a necessidades específicas das aplicações. Além disso, o esforço computacional necessário à comutação de tarefas é bastante menor, comparativamente à mesma atividade no âmbito do SO.

29

29

MODELO MULTITAREFA: MANY-TO-ONE

- Gestão feita por biblioteca em espaço utilizador
  - pouco comum atualmente
- Vantagens
  - eficiência na gestão das threads
    - por ser feita em user space (espaço utilizador)
- Desvantagens
  - processo pode bloquear se uma thread faz uma chamada de sistema bloqueante
  - multiplas threads não podem correr em paralelo num sistema multiprocessador/multicore



Fonte da imagem: [Silberschatz2014]

30

30

MODELO MULTITAREFA: ONE-TO-ONE

- Tarefas ao nível do kernel (kernel-level threads): A cada tarefa do processo multitarefa é associado um contexto de execução distinto no âmbito do SO. Sempre que um processo invoca uma chamada ao sistema para a criação de uma tarefa, é criada uma tarefa ao nível do kernel. Cada tarefa tem um identificador próprio, tem um contexto de execução próprio e está sujeito ao mecanismo de escalonamento e despacho do SO.

- Comparativamente à criação de múltiplos processos monotarefa (HWP), este mecanismo não é tão dispendioso do ponto de vista computacional. Recorde-se que as tarefas de um processo multitarefa partilham vários segmentos (código, dados, heap).

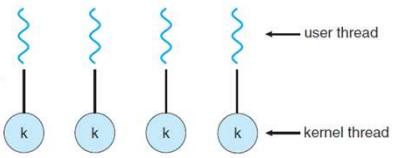
- Do ponto de vista de desempenho, neste modelo é possível fazer o despacho das várias tarefas por múltiplos processadores, sendo estas executadas realmente em paralelo. Por outro lado, se uma das tarefas bloquear à espera de uma operação de E/S, as restantes não serão afectadas. Cada tarefa é reconhecida pelo SO, podendo alterar-se a prioridade de cada uma delas.

31

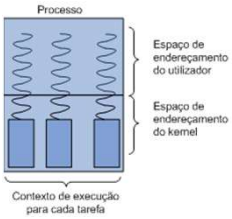
31

MODELO MULTITAREFA: ONE-TO-ONE

- Vantagens
  - uma thread pode correr quando outra está bloqueada
  - várias threads correm em paralelo num sistema multiprocessador/multicor
- Desvantagens
  - o overhead de criar threads no kernel pode afetar o desempenho da aplicação
    - por isso normalmente restringe-se o número de threads permitidas no sistema
- Por exemplo, o Windows e o Linux implementam este modelo



Fonte da imagem: [Silberschatz2014]



- O programador não tem de se preocupar, explicitamente, com o escalonamento e despacho por serem da responsabilidade do SO. Poderá, no entanto, alterar a prioridade de cada tarefa de modo a optimizar o tempo global de resposta do processo.

- O facto de se recorrer às chamadas do SO condicionam, de alguma forma, o desempenho.

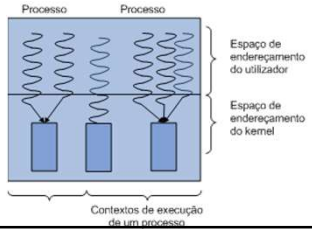
- A designação one-to-one (um para um) resulta do facto de cada tarefa no espaço do utilizador corresponder a uma tarefa no âmbito do kernel do SO.

32

32

MODELO MULTITAREFA: MANY-TO-MANY

- Modelo híbrido entre tarefas ao nível do utilizador e tarefas ao nível do kernel.
- Um modelo multitarefa one-to-one requer que o SO mantenha uma estrutura de dados para cada tarefa do processo, o que pode constituir-se problemático à medida que o número de tarefas aumenta. Neste modelo híbrido, o processo multitarefa solicita ao SO um número específico de tarefas ao nível do kernel.
- De modo a minimizar o tempo gasto na criação e destruição de tarefas, o SO mantém um conjunto destas estruturas (thread pooling). Depois das tarefas terem terminado, estas estruturas não são destruídas, ficando disponíveis aquando da criação de outras tarefas do mesmo ou de outro processo.



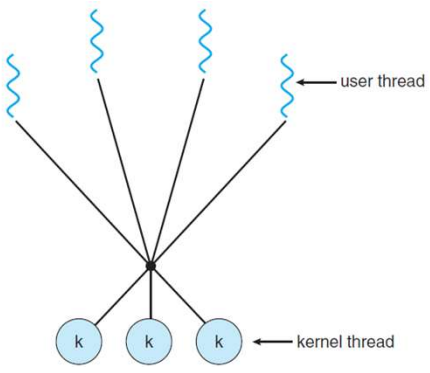
33

33

MODELO MANY-TO-MANY

- Multiplexam-se threads de nível utilizador num número inferior ou igual de threads do kernel
- Maior flexibilidade
  - junta vantagens do modelo many-to-many...
    - programador pode criar as threads que quiser

  ... com vantagens do modelo one-to-one
    - uma thread pode correr quando outra está bloqueada
    - múltiplas threads podem correr em paralelo num sistema multiprocessador



Fonte da imagem: [Silberschatz2014]

34

34

MULTITAREFA EM LINUX

Utilização da função de chamada ao sistema clone():

- Modelo one-to-one (kernel-level threads);
- Não é aconselhável a sua utilização por não existir garantia de portabilidade

```c
#define THREAD_STACK_SIZE 16384
#define _GNU_SOURCE
#include <sched.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int i,fd;
int mythread(){
  i++;
  close(fd);
  printf("Thread i=%d\n",i);
}

int main(){
  void **thread_stack;
  char ch;
  i=10;
  fd=open("test.file",O_RDONLY);
  thread_stack=(void **) malloc(THREAD_STACK_SIZE);
  printf("Process i=%d\n",i);
  clone(mythread,thread_stack+THREAD_STACK_SIZE/sizeof(void **),CLONE_VM|CLONE_FILES,NULL);
  sleep(2);
  printf("Process i=%d\n",i);
  if (read(fd,&ch,1)<1){
    perror("Erro de leitura");     exit(1);   }
  printf("Process Ficheiro: %c\n",ch);
}
```

Tarefa

Cria pilha para a tarefa

Partilha do espaço de endereçamento e de ficheiros abertos

Partilha do espaço onde reside a variável i e o descritor de ficheiro fd

```
root@gandalf:~/SO$ ./tarefas
Process i=10
Thread i=11
Process i=11
Erro de leitura: Bad file descriptor
root@gandalf:~/SO#
```

35

35

MULTITAREFA EM LINUX

Utilização da API pthreads, conforme IEEE Std 1003, Portable Operating System Interface (POSIX):

```c
// gcc -lpthread tarefas2.c -o tarefas2

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#define _GNU_SOURCE
#include <sched.h>
#include <pthread.h>

void *print_ch(void *ptr){
  char *ch;
  int i;
  ch=(char *) ptr;
  while (1) {
        printf("%c",*ch);
  }
}

int main(){
  pthread_t thread1, thread2;
  char *ch1="0";char *ch2="1";char *chp="x";
  int iret1, iret2;
  iret1=pthread_create(&thread1,NULL,print_ch, (void*) ch1);
  iret2=pthread_create(&thread2,NULL,print_ch,(void*) ch2);
  while(1) printf("%c",*chp);
  exit(0);
}
```

Tarefa

```
root@gandalf:~/SO# Tarefas
0000000000000000000 ....
1111111111111111 ....
Xxxxxxxxxxxxxxxxxxx ....
root@gandalf:~/SO#
```

36

36

MULTITAREFA EM LINUX

Utilização da API pthreads, conforme IEEE Std 1003, Portable Operating System Interface (POSIX):

```c
// Este programa deve ser compilado com:  gcc -pthread tarefas3.c -o tarefas3
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <pthread.h>

int i;
void *mythread1(){
  while (1) {
        printf("Thread 1: i=%d\n",i);
        sleep(1);
  }
}
void *mythread2(){
  while (1) {
        i++;
        sleep(3);
  }
}

int main(){
  pthread_t thread1, thread2;
  int iret1, iret2;
  i=0;
  iret1=pthread_create(&thread1,NULL,mythread1,NULL);
  iret2=pthread_create(&thread2,NULL,mythread2,NULL);
  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);
}
```

Tarefas

Aguarda pelo fim das tarefas

```
root@gandalf:~/SO# ./tarefas3
Thread 1: i=0
Thread 1: i=1
Thread 1: i=1
Thread 1: i=2
Thread 1: i=2
Thread 1: i=2
Thread 1: i=3
Thread 1: i=3
Thread 1: i=3
Thread 1: i=4
Thread 1: i=4
Thread 1: i=4
Thread 1: i=5
^C
root@gandalf:~/SO#
```

37

37

MULTITAREFA EM LINUX

Utilização da API pthreads, conforme IEEE Std 1003, Portable Operating System Interface (POSIX):

```c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <pthread.h>

int i, fd;
void *mythread(){
  printf("Thread i=%d\n",i);
  i++;
  close(fd);
}

int main(){
  pthread_t thread;
  int iret; char ch;
  i=0; fd=open("test.file", O_RDONLY);
  iret=pthread_create(&thread,NULL,mythread,NULL);
  pthread_join(thread,NULL);
  printf("Process i=%d\n",i);
  if (read(fd,&ch,1)<1){
        perror("Process Erro na leitura do ficheiro");
        exit(1); }
  printf("Process Ficheiro: %c\n",ch);
}
```

Tarefas

Aguarda pelo fim das tarefas

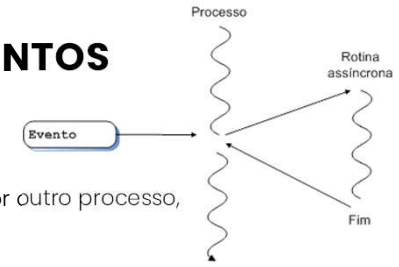
Partilha do espaço onde reside a variável i e o descritor de ficheiro fd

```
root@gandalf:~/SO# ./tarefas4
Thread i=0
Process i=1
Process Erro na leitura do ficheiro: Bad file descriptor
root@gandalf:~/SO#
```

38

38

TRATAMENTO DE EVENTOS: ACONTECIMENTOS ASSÍNCRONOS E EXCEPÇÕES



- Acontecimentos assíncronos: provocados pelo utilizador, assinalados por outro processo, etc.
- Excepções: divisão por zero, violação do espaço de endereçamento, execução de uma instrução ilegal, etc.
- O tratamento destas ocorrências torna-se difícil num ambiente de programação estritamente sequencial. O programa deve sistemática e periodicamente verificar da existência destes eventos, o que é penalizante para o desempenho. Num ambiente multitarefa, uma das tarefas poderia tomar esta responsabilidade, mas, mesmo esta solução estaria comprometida para um número elevado de ocorrências.
- Alguns sistemas oferecem um mecanismo que permite associar uma rotina a um evento. Sempre que o evento é detectado, o controlo de execução é transferido para a rotina. No fim da execução desta rotina, o controlo de execução é transferido para a instrução seguinte onde ocorreu a interrupção. Do ponto de vista funcional, este mecanismo é semelhante às interrupções por hardware.

39

39

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

- Em Linux, é possível enviar sinais aos processos de modo a assinalar acontecimentos assíncronos ou excepções. Os sinais constituem um mecanismo de grande importância na utilização do Linux, permitindo:
  - assinalar um processo sobre excepções com origem na máquina física (ex. divisão por zero, erro no bus de sistema, erro em dispositivos de I/O, etc.) ou na máquina virtual (ex. violação do espaço de endereçamento, erro na stack, instrução ilegal, etc.);
  - assinalar assincronamente um processo sobre a ocorrência de eventos externos.
- O envio de sinais a processos pode ser efectuado pelo utilizador a partir da shell (ex. comando kill ou <Ctrl> <c>) ou a partir de outros processos de forma explícita (função signal()/kill()) ou na sequência de outros acontecimentos (ex. terminação de um processo filho).
- Cada sinal é definido por um inteiro positivo único. Os primeiros 31 sinais (1..31) são os sinais standard. Uma nova classe de sinais (32..63),designados por sinais de tempo real, foi introduzida pelo standard POSIX.

40

40

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

Os sinais constituem um dos mecanismos fundamentais na comunicação e sincronização entre processos. No entanto, apenas permitem assinalar a ocorrência de um tipo de evento, não sendo possível a transmissão de informação adicional.

Alguns exemplos:

Designação	Nº	Descrição
SIGINT	2	Pedido de interrupção (equivalente ao <Ctrl> <c>)
SIGQUIT	3	Pedido de terminação
SIGILL	4	Instrução ilegal
SIGBUS	7	Erro no bus
SIGFPE	8	Erro na unidade aritmética e lógica (Floating point exception)
SIGKILL	9	Terminação do processo (Interrupção - não pode ser ignorado ou tratado de outra forma)
SIGUSR1, SIGUSR2	10,12	Definido pelo utilizador
SIGSEGV	11	Violação do espaço de endereçamento
SIGSTKFLT	16	Erro na stack
SIGCHLD	17	Processo filho terminou
SIGSTOP	19	Fim de execução (não pode ser ignorado ou tratado de outra forma)
SIGALRM	14	Alarme do relógio
SIGCONT	18	Continuar execução, se suspenso
SIGTSTP	20	Suspender processo

41

41

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

- Um processo pode definir três tipos de tratamento para os sinais:
  - Para cada um dos sinais definidos no sistema, existe um tratamento predefinido que na maioria dos casos resulta na terminação do processo. Este é o tratamento por omissão e é representado pela constante SIG_DFL.
  - O processo pode solicitar que os sinais sejam simplesmente ignorados (SIG_IGN), não resultando qualquer acção. Alguns sinais não podem ser ignorados, de modo a ser sempre possível a terminação de um processo (ex. SIGKILL, SIGSTOP).
  - O processo associa uma rotina de tratamento do sinal que é executada de forma assíncrona na sequência da recepção de um sinal.
- Exemplo: O processo nada define sobre o tratamento de sinais, logo é considerado o tratamento por omissão (terminação do processo).

```
#include <stdio.h>

main(){
  while (1);
}
```

Execução em segundo plano, de modo a ter acesso à shell

```
root@gandalf:~/SO# ./sinais1 &
[1] 885505
root@gandalf:~/SO# ps s
  UID      PID          PENDING          BLOCKED          IGNORED          CAUGHT STAT TTY        TIME COMMAND
    0      814 0000000000000000 0000000000000000 0000000000000006 0000000000000000 Ss+  tty1       0:00 /sbin/agetty -o -p -- \u --noclear - linux
    0   883466 0000000000000000 0000000000010000 0000000000384004 000000004b813efb Ss   pts/2      0:00 -bash
    0   885505 0000000000000000 0000000000000000 0000000000000000 0000000000000000 R    pts/2      0:01 ./sinais1
    0   885506 0000000000000000 0000000000000000 0000000000000000 000000007391fef9 R+   pts/2      0:00 ps s
root@gandalf:~/SO#
```

Nenhum sinal ignorado

Nenhum sinal tratado

42

42

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

Exemplo: Através da função alarm(3), o processo recebe um sinal SIGALRM passados 3 segundos.

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

main(){
  printf("Daqui a 3 segundos vou morrer ...\n");

  alarm(3);

  while (1) printf(".");
}
```

Envia um sinal SIGALARM daqui a 3 segundos

```
.....
.....
.....Alarm clock
root@gandalf:~/SO#
```

43

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

▪Exemplo: Ignorar todos os sinais (1..31). Notar que o SIGKILL(9) e SIGSTOP(19) não podem ser ignorados.

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

main(){
  int i;
  for (i=1;i<=31;i++)
      signal(i, SIG_IGN);
  while (1) ;
}
```

Ignorar todos os sinais

Execução em segundo plano, de modo a ter acesso à shell

```
root@gandalf:~/SO# ./sinais3 &
[2] 884964
root@gandalf:~/SO# ps s
  UID     PID         PENDING         BLOCKED         IGNORED         CAUGHT STAT TTY        TIME COMMAND
    0     814 0000000000000000 0000000000000000 0000000000000006 0000000000000000 Ss+  tty1       0:00 /sbin/agetty -o -p -- \u --noclear - linux
    0  883466 0000000000000000 0000000000010000 0000000000384004 000000004b813efb Ss   pts/2      0:00 -bash
    0  884413 0000000000000000 0000000000000000 0000000000000000 0000000000000000 R    pts/2     19:40 ./tarefas5
    0  884964 0000000000000000 0000000000000000 000000007ffbfeff 0000000000000000 R    pts/2      0:03 ./sinais3
    0  884965 0000000000000000 0000000000000000 0000000020000000 000000007391fef9 R+   pts/2      0:00 ps s
root@gandalf:~/SO#
```

Todos os sinais ignorados, à excepção de SIGKILL (9) e SIGSTOP (19)

Nenhum sinal tratado

44

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

- Exemplo: Tratar todos os sinais (1..31). Notar que o SIGKILL(9) e SIGSTOP(19) não podem ser tratados.

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


void trataSinais(int sinal){
  printf("[trataSinais]: recebi o sinal %d\n",sinal);
}
int main(){
  int i;
  for (i=1;i<=31;i++)
        signal(i, trataSinais);
  while (1) ;
}
```

Tratar todos os sinais. Associar a função trataSinais

Execução em segundo plano, de modo a ter acesso à shell

Todos os sinais tratados à excepção de SIGKILL (9) e SIGSTOP (19)

```
root@gandalf:~/SO# ./sinais4 &
[3] 885101
root@gandalf:~/SO# ps s
  UID     PID       PENDING         BLOCKED         IGNORED         CAUGHT STAT TTY        TIME COMMAND
    0     814 0000000000000000 0000000000000000 0000000000000006 0000000000000000 Ss+  tty1       0:00 /sbin/agetty -o -p -- \u --noclear - linux
    0  883466 0000000000000000 0000000000010000 0000000000384004 000000004b813efb Ss   pts/2      0:00 -bash
    0  884964 0000000000000000 0000000000000000 000000007ffbfeff 0000000000000000 R    pts/2      5:10 ./sinais3
    0  885101 0000000000000000 0000000000000000 0000000000000000 000000007ffbfeff R    pts/2      0:02 ./sinais4
    0  885102 0000000000000000 0000000000000000 0000000000000000 00000000731fef9 R+   pts/2      0:00 ps s
root@gandalf:~/SO#
```

Nenhum sinal ignorado

```
root@gandalf:~/SO# kill 885101
[trataSinais]: recebi o sinal 15
```

45

45

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

- Exemplo: Utilização de sinais na sincronização entre dois processos (pai e filho).

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int fid;
void ptrataUSR1(int sinal){
  printf("PAI: Recebi o sinal %d\n",sinal);
  sleep(1);kill(fid,SIGUSR1);
}
void ftrataUSR1(int sinal){
  printf("FILHO: Recebi o sinal %d\n",sinal);
  sleep(1);kill(getppid(),SIGUSR1);
}
main(){
  fid=fork();
  if (fid!=0) { /* Processo pai */
        signal(SIGUSR1,ptrataUSR1);
        while (1);
  }
  else { /* Processo filho */
       signal(SIGUSR1,ftrataUSR1);
       sleep(1); kill(getppid(),SIGUSR1);
       while (1);
  }
}
```

Envia sinal SIGUSR1 ao processo filho

Envia sinal SIGUSR1 ao processo pai

Sinal SIGUSR1 tratado pela função ptrataUSR1

Sinal SIGUSR1 tratado pela função ftrataUSR1

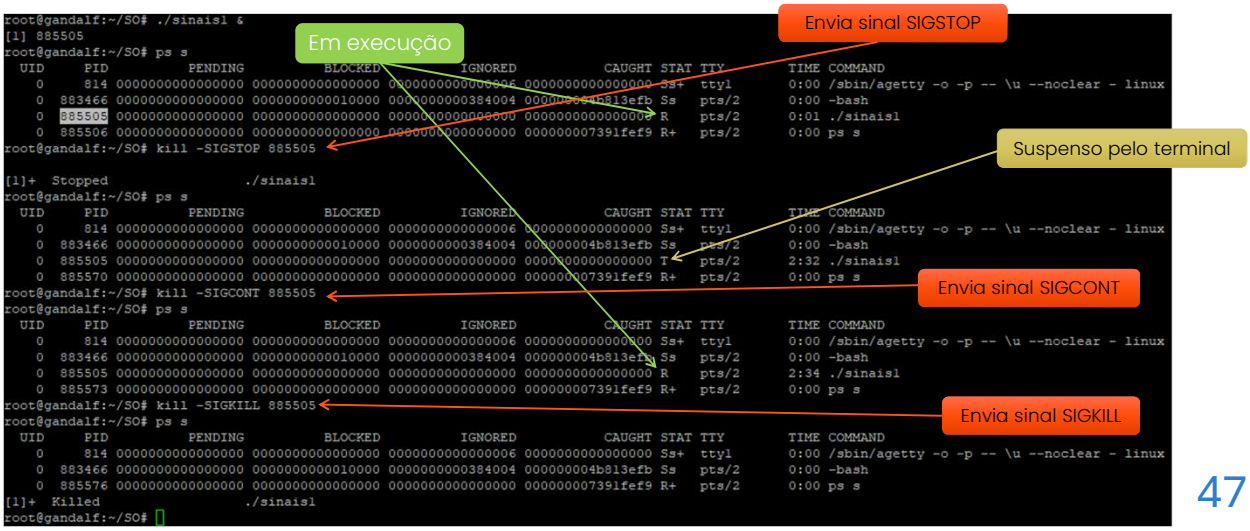
```
root@gandalf:~/SO# ./sinais5
PAI: Recebi o sinal 10
FILHO: Recebi o sinal 10
PAI: Recebi o sinal 10
FILHO: Recebi o sinal 10
PAI: Recebi o sinal 10
FILHO: Recebi o sinal 10
PAI: Recebi o sinal 10
```

46

46

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

- Exemplo: Suspender e reactivar processos.



47

47

TRATAMENTO DE SINAIS (SIGNALS) EM LINUX

Exemplo: Tratar sinal SIGCHLD

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

int status;
void filhoterminou(int sinal) {
  wait( & status); /*impede zombie */
  printf("\nProcesso filho terminou, status= %d\n", status >> 8);
}
main() {
  signal(SIGCHLD, filhoterminou);
  if (fork()) {
    /* Processo pai */
    /* Não precisa de ficar bloqueado à espera */
    while (1); /* para simular actividade normal do pai */
  } else {
    /* Processo filho */
    sleep(10); /* para simular o tempo de actividade do filho */
    return 5;
  }
}
```

```
root@gandalf:~/SO# ./sinais6 &
[1] 886769
root@gandalf:~/SO# ps -fH
UID          PID    PPID  C STIME TTY          TIME CMD
root      883466  883457  0 15:17 pts/2    00:00:00 -bash
root      886769  883466 99 17:17 pts/2    00:00:06   ./sinais6
root      886770  886769  0 17:17 pts/2    00:00:00     ./sinais6
root      886774  883466  0 17:17 pts/2    00:00:00   ps -fH
root@gandalf:~/SO#
Processo filho terminou, status= 5

root@gandalf:~/SO# ps -fH
UID          PID    PPID  C STIME TTY          TIME CMD
root      883466  883457  0 15:17 pts/2    00:00:00 -bash
root      886769  883466 99 17:17 pts/2    00:00:17   ./sinais6
root      886782  883466  0 17:18 pts/2    00:00:00   ps -fH
root@gandalf:~/SO#
```

48

48