

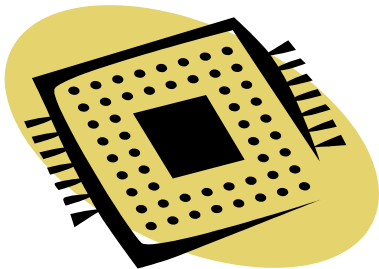
# Sistemas Operacionais II

## Processos



# Noções Básicas

- **Processo:** abstração usada pelo Linux para representar um programa em execução
  - Objeto através do qual podem ser gerenciados:
    - Uso de memória
    - Tempo do processador
    - Recursos de entrada e saída



# Componentes de um Processo

- Mapa de espaços de endereço do processo
  - Conjunto de páginas de memória que o *kernel* marcou para serem usadas pelo processo
    - Uma página de memória em um PC tem tipicamente 4 KB
  - Estado atual do processo (espera, parado, em execução, etc.)
  - Prioridade de execução do processo

# Componentes de um Processo

- Informações sobre os recursos que o processo usou
- Máscara de sinalização do processo (um registro de quais sinais são bloqueados)
- Proprietário do processo

# PID: número de identificação do processo

- O Linux não fornece uma chamada de sistema para criar um novo processo
- Um processo existente tem que se clonar para criar um processo novo, o clone então troca o programa que está executando
- Quando um processo se clona, o original é chamado de pai e a cópia de filho

# PID: número de identificação do processo

- O PID é um número de 16 bits designado sequencialmente pelo Linux conforme novos processos são criados
- Todo processo tem um pai, exceto o processo especial **init**
  - Os processos no Linux são arrançados como em uma árvore, com o **init** sendo a raiz.



A terminal window showing a list of processes. The columns are labeled 'PID' and 'USER'. The first three rows are visible, showing PIDs 1, 2, and 3, all with the user 'root'.

PID	USER
1	root
2	root
3	root

# PID

- O tipo `pid_t` é definido em `<sys/types.h>` para referenciar PIDs
- Um programa pode obter seu PID executando a chamada de sistema **`getpid()`**
- Um programa pode obter o PID de seu pai executando a chamada de sistema **`getppid()`**

```
int main ()
{
    printf ("O id do processo é %d\n", (int) getpid ());
    printf ("O id do pai do processo é %d\n", (int) getppid ());
    return 0;
}
```

*print-pid.c*

# Gerenciamento de processo

- Responsável principalmente pela alocação de processadores aos processos.
- Também entrega sinais, carrega módulos de núcleo e recebe interrupções.



# Terminal de Controle

- A maioria dos processo está associada a um terminal de controle (tty)
  - Determina terminal de entrada/saída/erro padrão
  - Ao iniciar um comando no shell seu terminal se torna o terminal de controle do processo



# Ciclo de Vida de um Processo

- Um processo faz uma cópia de si mesmo para criar um novo processo, usando a chamada de sistema **fork**
- **fork** retorna o PID do filho recém-criado para o pai e 0 para o filho
- O processo filho assume outro papel
- Quando o sistema é inicializado, o *kernel* cria e instala vários processos, dentre eles o **init**, que é responsável pela maioria dos scripts de inicialização
- Quando um processo é completado, **init** chama uma rotina **\_exit** para notificar o *kernel* de que ele está pronto para expirar
- O processo pai reconhece e faz uma chamada **wait** para reconhecer a expiração do processo filho
- Filhos órfãos passam a ser filhos de **init**, que se encarrega de fazer a chamada **wait**

# Processos no Linux

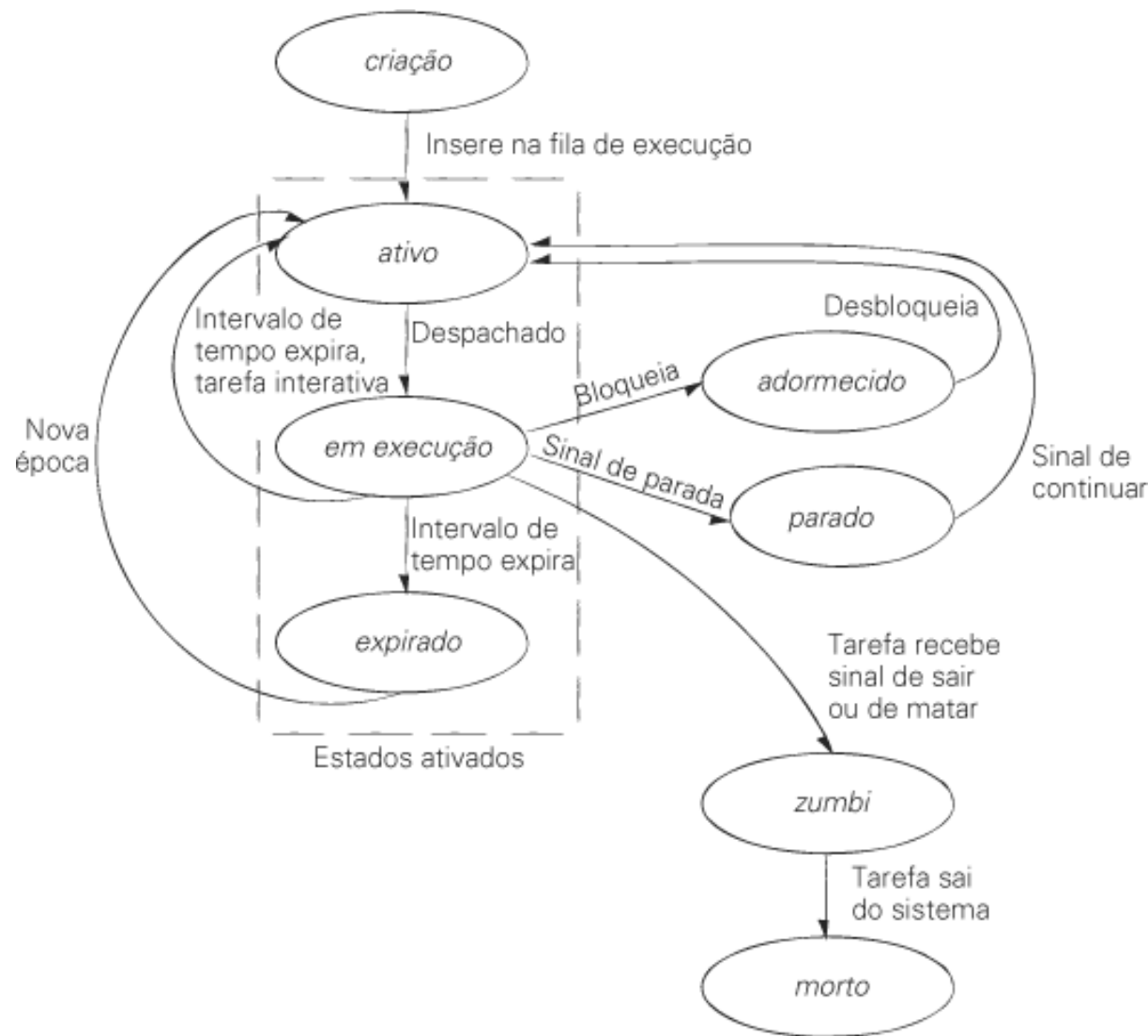
```
pid = fork( );                /* se o fork tiver êxito, o processo pai obterá pid > 0*/  
if (pid < 0) {                /* o fork falhou (por exemplo, a memória ou alguma tabela está cheia) */  
    handle_error ( );  
} else if (pid > 0) {  
    /* código do pai segue aqui./ */  
  
} else {  
    /* código do filho segue aqui./ */  
  
}
```

■ **Figura 10.3** Criação de processo no Linux.

# Estados de Processos

- Há basicamente quatro estados de execução de um processo:
  - **Executável:** O processo pode ser executado
    - Apenas esperando tempo de CPU para processar seus dados
  - **Dormente:** O processo está aguardando algum recurso
    - Aguardando uma entrada de teclado ou de rede, um dado de disco, etc.
  - **Zumbi:** O processo está tentando se destruir
    - Terminou sua execução mas ainda não teve seus dados coletados
  - **Parado:** O processo é suspenso (não há permissão para ser executado)
    - Proibido administrativamente de executar (c/ um STOP ou TSTP e são reiniciados com CONT)

# Organização de processos e threads



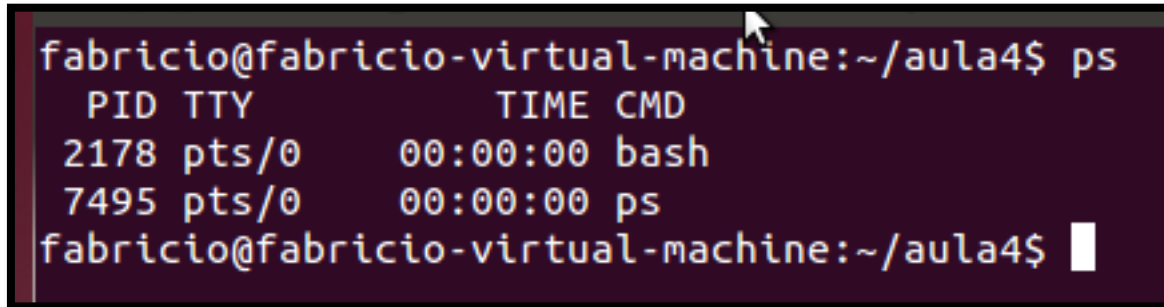
**Figura 20.2** Diagrama de transição de estado de tarefa.

# ps: Monitorando Processos

- **ps** é a principal ferramenta para monitorar processos, ela fornece:
  - PID
  - UID
  - Prioridade
  - Terminal de controle
  - Informação de quanta memória foi consumida
  - Estado atual
- **ps** tem muitas implementações diferentes no UNIX
  - O Linux dá suporte a maioria delas
  - Também suporta opções nos estilos:
    - Opções UNIX: podem ser agrupadas e devem ser precedidas por um hífen
    - Opções BSD: podem ser agrupadas e não devem usar hífen
    - Opções longas GNU: são precedidas por dois hífen
- Exemplo:
  - Utilize **ps aux** para obter uma visão geral dos processos em execução
    - Padrão BSD

# ps: Monitorando Processos

- Por padrão **ps** mostra apenas processos controlados pelo terminal (ou janela de terminal) em que foi invocado.

A terminal window with a dark purple background and light-colored text. The prompt is 'fabricio@fabricio-virtual-machine:~/aula4\$'. The command 'ps' has been entered, and the output is displayed in a table format. The table has four columns: PID, TTY, TIME, and CMD. There are two rows of data: one for PID 2178 (TTY pts/0, TIME 00:00:00, CMD bash) and one for PID 7495 (TTY pts/0, TIME 00:00:00, CMD ps). The prompt 'fabricio@fabricio-virtual-machine:~/aula4\$' is shown again at the bottom with a cursor.

```
fabricio@fabricio-virtual-machine:~/aula4$ ps
  PID TTY          TIME CMD
 2178 pts/0    00:00:00 bash
 7495 pts/0    00:00:00 ps
fabricio@fabricio-virtual-machine:~/aula4$
```

# ps: Monitorando Processos

- `ps -e -o pid,ppid,command`
  - Note que o ID do pai de **ps** é o ID do **bash**
  - E o de **bash**, por sua vez, é o do terminal

```
fabricao@fabricao-virtual-machine: ~/aula4
2088 1 /usr/lib/unity-lens-files/unity-files-daemon
2089 1 /usr/lib/unity-lens-applications/unity-applications-daemon
2090 1 /usr/lib/unity-lens-music/unity-music-daemon
2114 1 /usr/bin/python /usr/bin/zeitgeist-daemon
2138 1 /usr/lib/unity-lens-music/unity-musicstore-daemon
2154 2114 /bin/cat
2156 1 zeitgeist-datahub
2233 1817 /usr/bin/python /usr/share/system-config-printer/applet.py
2247 1817 update-notifier
2264 1 /usr/bin/python /usr/lib/system-service/system-service-d
2323 1817 /usr/lib/deja-dup/deja-dup/deja-dup-monitor
7367 1 kdeinit4: kdeinit4 Running...
7370 7367 kdeinit4: klauncher [kdeinit] --fd=8
7372 1 kdeinit4: kded4 [kdeinit]
7381 1 /usr/bin/knotify4
7438 2 [kworker/0:0]
7465 1 /usr/lib/gvfs/gvfsd-metadata
7493 2 [kworker/0:2]
7514 2 [kworker/0:1]
7525 1 gnome-terminal
7530 7525 gnome-pty-helper
7531 7525 bash
7590 7531 ps -e -o pid,ppid,command
fabricao@fabricao-virtual-machine:~/aula4$
```

```
7514 2 [kworker/0:1]
7525 1 gnome-terminal
7530 7525 gnome-pty-helper
7531 7525 bash
7590 7531 ps -e -o pid,ppid,command
fabricao@fabricao-virtual-machine:~/aula4$
```



# ps aux

Red Hat

```
root      2668  0.0  1.2  9484 3256 ?        Ss   16:53   0:00 sendmail: accepti
smmsp     2676  0.0  1.0  6620 2588 ?        Ss   16:53   0:00 sendmail: Queue r
root      2687  0.0  0.2  3012  532 ?        Ss   16:53   0:00 gpm -m /dev/input
root      2697  0.0  0.4  5424 1152 ?        Ss   16:53   0:00 crond
xfs       2720  0.0  0.6  4528 1612 ?        Ss   16:53   0:00 xfs -droppriv -da
root      2730  0.0  0.2  1724  640 ?        SNs  16:53   0:00 anacron -s
root      2739  0.0  0.2  1924  744 ?        Ss   16:53   0:00 /usr/sbin/atd
dbus      2758  0.0  0.5 14004 1324 ?        Ssl  16:53   0:00 dbus-daemon-1 --s
root      2771  0.0  0.2  5440  540 ?        Ss   16:53   0:00 rhnsd --interval
root      2780  0.0  0.4  3588 1040 ?        Ss   16:53   0:00 cups-config-daemo
root      2791  0.1  1.6  6880 4140 ?        Ss   16:53   0:03 hald
root      2803  0.0  0.1  2408  404 tty3     Ss+  16:53   0:00 /sbin/mingetty tt
root      2807  0.0  0.1  3224  404 tty4     Ss+  16:53   0:00 /sbin/mingetty tt
root      2811  0.0  0.1  2824  404 tty5     Ss+  16:53   0:00 /sbin/mingetty tt
root      2812  0.0  0.1  2424  404 tty6     Ss+  16:53   0:00 /sbin/mingetty tt
root      2813  0.0  0.9 11348 2352 ?        Ss   16:53   0:00 /usr/bin/gdm-bina
root      3256  0.0  1.1 11984 2936 ?        S    16:53   0:00 /usr/bin/gdm-bina
root      3263  0.2  3.3 10712 8444 ?        S    16:53   0:06 /usr/X11R6/bin/X
gdm       3400  0.0  4.2 20892 10740 ?        Ss   16:53   0:01 /usr/bin/gdmgreet
root      3517  0.0  0.5  4628 1284 ?        Ss   17:09   0:00 login -- root
root      3603  0.0  0.5  6024 1380 tty2     Ss+  17:12   0:00 -bash
root      4121  0.1  0.5  2948 1488 ?        Ss   17:26   0:00 login -- fbreve
fbreve    4206  0.3  0.5  5520 1372 tty1     Ss   17:27   0:00 -bash
fbreve    4250  0.0  0.2  3204  744 tty1     R+   17:27   0:00 ps aux
[fbreve@localhost ~]$ _
```

# ps aux

Ubuntu

```
fabricio@fabricio-virtual-machine: ~  
fabricio 1989 0.0 0.2 63660 2956 ? S 22:36 0:00 /usr/lib/gvfs/g  
fabricio 1990 1.0 1.8 322856 18820 ? SL 22:36 0:00 /usr/lib/unity/  
fabricio 2002 0.0 0.6 336860 7104 ? SL 22:36 0:00 /usr/lib/indica  
fabricio 2003 0.0 0.5 261092 6072 ? SL 22:36 0:00 /usr/lib/indica  
fabricio 2004 0.0 0.7 233504 7724 ? SL 22:36 0:00 /usr/lib/indica  
fabricio 2006 0.0 0.4 152404 4952 ? SL 22:36 0:00 /usr/lib/indica  
fabricio 2013 0.1 0.6 242504 6716 ? SL 22:36 0:00 /usr/lib/indica  
fabricio 2028 0.0 0.2 49264 2672 ? S 22:36 0:00 /usr/lib/geoclu  
fabricio 2033 0.0 0.3 77244 3924 ? SL 22:37 0:00 telepathy-indic  
fabricio 2035 0.0 0.3 60396 3516 ? S 22:37 0:00 /usr/lib/telepa  
fabricio 2061 0.1 0.9 188420 9452 ? S 22:37 0:00 /usr/lib/gnome-  
fabricio 2065 3.9 5.9 553192 60500 ? SL 22:37 0:00 /usr/bin/unity-  
fabricio 2088 0.2 0.4 124244 4872 ? SL 22:37 0:00 /usr/lib/unity-  
fabricio 2089 1.1 0.9 155908 9660 ? SL 22:37 0:00 /usr/lib/unity-  
fabricio 2090 0.3 0.4 124572 4940 ? SL 22:37 0:00 /usr/lib/unity-  
fabricio 2114 1.4 1.8 141716 19208 ? SL 22:37 0:00 /usr/bin/python  
fabricio 2138 0.1 0.3 102268 3140 ? SL 22:37 0:00 /usr/lib/unity-  
fabricio 2154 0.0 0.0 11252 580 ? S 22:37 0:00 /bin/cat  
fabricio 2156 0.1 0.5 160968 5456 ? SL 22:37 0:00 zeitgeist-datah  
fabricio 2170 2.0 1.6 249692 16988 ? SL 22:37 0:00 gnome-terminal  
fabricio 2175 0.0 0.0 14656 824 ? S 22:37 0:00 gnome-pty-helpe  
fabricio 2178 1.5 0.4 29460 4328 pts/0 Ss 22:37 0:00 bash  
fabricio 2232 0.0 0.1 22232 1268 pts/0 R+ 22:37 0:00 ps aux  
fabricio@fabricio-virtual-machine:~$
```

Campo	Conteúdo
USER	Nome do usuário do proprietário do processo
PID	ID (identificador) do processo
%CPU	Porcentagem dos recursos de CPU que este processo está usando
%MEM	Porcentagem da memória real que este processo está usando
VSZ	Tamanho virtual do processo
RSS	Resident Set Size (número de páginas na memória)
TTY	ID (identificador) de terminal de controle
STAT	Estado de processo atual <div>             R = Executável                      D = Espera no disco (ou a curto prazo)              S = Dormente (&lt;20 seg)            T = Rastreado ou interrompido              Z = Zumbi           </div> Flags adicionais: W = Processo paginado em disco < = O processo tem uma prioridade maior do que a normal N = O processo tem uma prioridade menor do que a normal L = Algumas páginas são bloqueadas no núcleo (kernel)
START	Horário em que o processo foi iniciado
TIME	Tempo de CPU que o processo consumiu
COMMAND	Nome do comando e argumentos <sup>a</sup>

a. Os argumentos podem ser truncados; acrescente o argumento **ww** para evitar isto. Os programas são capazes de modificar esta informação, de modo que não é necessariamente uma representação acurada da verdadeira linha de comando.

# **ps**: Monitorando Processos

- **ps** recorta os comandos para caberem em uma linha
  - Para evitar esse truncamento use o argumento **w**
- Use o argumento **lax** para obter mais informações técnicas
  - Mais rápido, não traduz cada UID em nome
  - Inclui PPID (PID do pai), valor nice (NI) e nome ou função do kernel na qual o processo está dormente (WCHAN)

# **top:** Monitoramento ainda melhor de processos

- O comando **ps** dá apenas um instantâneo de seu sistema
- O comando **top** fornece um sumário atualizado regularmente
  - Como padrão a tela é atualizada a cada 3 segundos
  - Processos mais ativos aparecem no alto
  - **top** consome recursos, portanto deve ser usado apenas para fins de diagnóstico

# top

Red Hat

```
top - 19:57:32 up 3:05, 3 users, load average: 0.20, 0.16, 0.09
Tasks: 87 total, 1 running, 86 sleeping, 0 stopped, 0 zombie
Cpu(s): 33.2% us, 18.6% sy, 0.0% ni, 48.2% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 254784k total, 239672k used, 15112k free, 10968k buffers
Swap: 524280k total, 160k used, 524120k free, 112792k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
11816	fbreve	16	0	70136	22m	14m	S	67.9	9.2	0:00.99	firefox-bin
3263	root	16	0	35540	16m	5724	S	23.0	6.4	0:34.59	X
11603	fbreve	15	0	21240	9928	7580	S	5.0	3.9	0:01.17	wnck-applet
11574	fbreve	15	0	14876	7360	6180	S	3.0	2.9	0:01.66	metacity
2427	root	16	0	1916	484	404	S	2.0	0.2	0:02.91	irqbalance
11578	fbreve	15	0	23888	12m	8560	S	2.0	5.1	0:03.64	gnome-panel
11550	fbreve	15	0	4748	2152	1712	S	1.0	0.8	0:00.58	xscreensaver
11588	fbreve	25	10	35916	23m	11m	S	1.0	9.5	0:18.04	rhn-applet-gui
11605	fbreve	15	0	22560	10m	8020	S	1.0	4.2	0:00.82	mixer_applet2
11701	fbreve	16	0	2124	940	752	R	1.0	0.4	0:01.03	top
1	root	16	0	3124	560	480	S	0.0	0.2	0:01.28	init
2	root	RT	0	0	0	0	S	0.0	0.0	0:00.62	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.06	ksoftirqd/0
4	root	RT	0	0	0	0	S	0.0	0.0	0:00.26	migration/1
5	root	34	19	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd/1
6	root	5	-10	0	0	0	S	0.0	0.0	0:00.20	events/0
7	root	5	-10	0	0	0	S	0.0	0.0	0:00.21	events/1
8	root	5	-10	0	0	0	S	0.0	0.0	0:00.02	khelper



# top

Ubuntu

```
fabricio@fabricio-virtual-machine: ~  
top - 22:45:15 up 9 min, 1 user, load average: 0.47, 0.47, 0.28  
Tasks: 141 total, 1 running, 140 sleeping, 0 stopped, 0 zombie  
Cpu(s): 1.3%us, 0.7%sy, 0.0%ni, 84.4%id, 13.6%wa, 0.0%hi, 0.0%si, 0.0%st  
Mem: 1019072k total, 790896k used, 228176k free, 80080k buffers  
Swap: 1046524k total, 27024k used, 1019500k free, 204672k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1229	root	20	0	191m	39m	6652	S	1.7	4.0	0:03.41	Xorg
1854	fabricio	20	0	27388	3032	864	S	0.3	0.3	0:01.28	dbus-daemon
1891	fabricio	20	0	224m	12m	8968	S	0.3	1.3	0:00.38	metacity
1929	fabricio	20	0	289m	23m	9952	S	0.3	2.3	0:01.42	vmtoolsd
2170	fabricio	20	0	244m	15m	9740	S	0.3	1.5	0:00.65	gnome-terminal
1	root	20	0	24184	1856	1240	S	0.0	0.2	0:01.34	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.04	ksoftirqd/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.27	kworker/0:0
5	root	20	0	0	0	0	S	0.0	0.0	0:00.25	kworker/u:0
6	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
7	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	cpuset
8	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	khelper
9	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	netns
10	root	20	0	0	0	0	S	0.0	0.0	0:00.00	sync_supers
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	bdi-default
12	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kintegrityd

# top em Linux

```
top - 01:10:38 up 248 days,  9:10,  2 users,  load average: 0.13, 0.07, 0.02
Tasks:  44 total,   1 running,  43 sleeping,   0 stopped,   0 zombie
Cpu(s):  5.2% user,   8.6% system,   0.0% nice,  86.2% idle
Mem:      61744k total,   60284k used,    1460k free,    5984k buffers
Swap:    160640k total,   1240k used,  159400k free,   34828k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21011	fabricio	14	0	1128	1128	896	R	13.8	1.8	0:17.46	top
1	root	8	0	216	216	196	S	0.0	0.3	2:23.89	init
2	root	9	0	0	0	0	S	0.0	0.0	0:00.11	keventd
3	root	19	19	0	0	0	S	0.0	0.0	0:03.89	ksoftirqd_CPU0
4	root	9	0	0	0	0	S	0.0	0.0	7:14.57	kswapd
5	root	9	0	0	0	0	S	0.0	0.0	0:04.03	bdflush
6	root	9	0	0	0	0	S	0.0	0.0	0:03.31	kupdated
10	root	-1	-20	0	0	0	S	0.0	0.0	0:00.00	mdrecoveryd
11	root	9	0	0	0	0	S	0.0	0.0	3:16.71	kjournald
40	root	9	0	0	0	0	S	0.0	0.0	0:12.62	kjournald
41	root	9	0	0	0	0	S	0.0	0.0	0:01.11	kjournald
63	root	9	0	244	224	220	S	0.0	0.4	19:35.51	syslogd
66	root	9	0	52	4	4	S	0.0	0.0	0:00.05	klogd
409	root	9	0	0	0	0	S	0.0	0.0	0:00.00	khubd
431	bin	9	0	104	4	4	S	0.0	0.0	0:00.89	rpc.portmap
435	root	9	0	96	4	4	S	0.0	0.0	0:00.09	rpc.statd
437	root	9	0	0	0	0	S	0.0	0.0	6:44.83	rpciod



# top em Sun OS

```
last pid: 4234; load averages: 0.11, 0.07, 0.04 00:38:20
69 processes: 66 sleeping, 1 zombie, 1 stopped, 1 on cpu
CPU states: 97.5% idle, 1.6% user, 0.9% kernel, 0.0% iowait, 0.0% swap
Memory: 2048M real, 1840M free, 33M swap in use, 2015M swap free
```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	CPU	COMMAND
4234	fabricio	1	50	0	2608K	2184K	cpu11	0:12	1.72%	top
549	apfreire	1	58	0	2320K	1120K	sleep	0:15	0.01%	lmgrd
214	0	5	58	0	5248K	3168K	sleep	60:07	0.00%	automountd
513	0	12	58	0	2744K	2032K	sleep	19:47	0.00%	mibiisa
334	0	1	58	0	3600K	824K	sleep	18:23	0.00%	sshd
246	0	11	58	0	9896K	8648K	sleep	5:40	0.00%	nsd
170	0	3	58	0	2528K	1640K	sleep	1:22	0.00%	nis_cachemgr
228	0	16	58	0	3800K	1952K	sleep	0:39	0.00%	syslogd
510	0	1	58	0	2248K	1632K	sleep	0:35	0.00%	snmpd
1	0	1	58	0	856K	168K	sleep	0:32	0.00%	init
168	0	7	30	0	3648K	1288K	sleep	0:10	0.00%	keyserv
165	0	1	58	0	2464K	1560K	sleep	0:06	0.00%	rpcbind
26908	pvgrf	1	58	0	6672K	2632K	sleep	0:05	0.00%	sshd
194	0	1	58	0	2448K	720K	sleep	0:05	0.00%	inetd
227	0	1	48	0	2512K	1760K	sleep	0:03	0.00%	cron

# Chamadas de sistema para gerenciamento de processo no Linux

Chamada de sistema	Descrição
<code>pid = fork( )</code>	Cria um processo filho idêntico ao pai
<code>pid = waitpid(pid, &amp;statloc, opts)</code>	Espera o processo filho terminar
<code>s = execve(name, argv, envp)</code>	Substitui a imagem da memória de um processo
<code>exit(status)</code>	Termina a execução de um processo e retorna o status
<code>s = sigaction(sig, &amp;act, &amp;oldact)</code>	Define a ação a ser tomada nos sinais
<code>s = sigreturn(&amp;context)</code>	Retorna de um sinal
<code>s = sigprocmask(how, &amp;set, &amp;old)</code>	Examina ou modifica a máscara do sinal
<code>s = sigpending(set)</code>	Obtém o conjunto de sinais bloqueados
<code>s = sigsuspend(sigmask)</code>	Substitui a máscara de sinal e suspende o processo
<code>s = kill(pid, sig)</code>	Envia um sinal para um processo
<code>residual = alarm(seconds)</code>	Ajusta o relógio do alarme
<code>s = pause( )</code>	Suspende o chamador até o próximo sinal

**Tabela 10.3** Algumas chamadas ao sistema relacionadas com processos. O código de retorno `s` é `-1` quando ocorre um erro, `pid` é o ID do processo e `residual` é o tempo restante no alarme anterior. Os parâmetros são aqueles sugeridos pelos próprios nomes.

# Criando um Processo



- Duas formas mais comuns:
  - **system**
  - **fork e exec**

# system

- Faz parte da biblioteca padrão do C
- Fácil de usar
- Cria um subprocesso rodando o *shell* padrão e passa o comando ao novo *shell* para execução

```
#include <stdlib.h>

int main ()
{
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

*system.c*

# system

- Retorna o código de saída do comando no *shell*
  - Se o próprio *shell* não puder ser executado, retorna 127
  - Se ocorrer outro erro, retorna -1
- Problemas ao usar o *shell* para invocar o comando:
  - Não dá para confiar na disponibilidade de qualquer *shell* ou versão específica
    - Diferentes versões do GNU/Linux usam diferentes versões do *bash*
      - Resultados podem ser diferentes
  - É preferível usar *fork* e *exec*

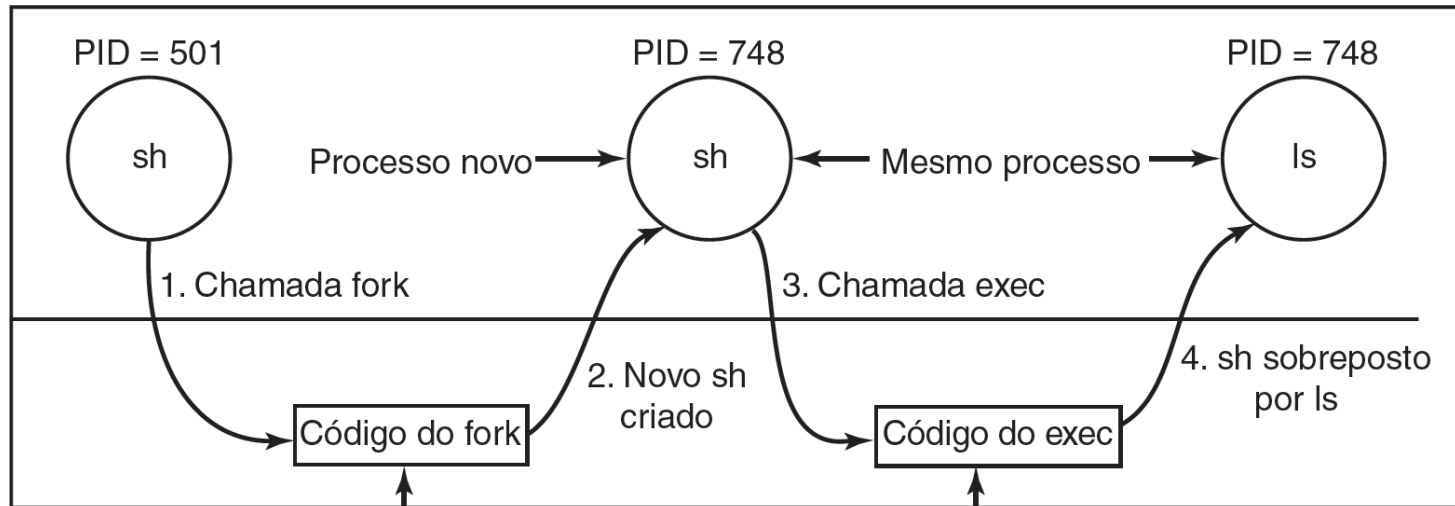
# Usando *fork* e *exec*

- O Linux não contém uma única função que cria um novo processo e executa um novo programa em um único passo
- Em vez disso, existem duas funções:
  - **fork**: cria um processo filho que é uma cópia exata do processo pai
  - **exec**: família de processos que faz um processo deixar de ser a instância de um programa e passar a ser uma instância de outro programa
- Para criar um novo processo, utiliza-se **fork** para fazer uma cópia do processo atual e **exec** para transformar um dos processos em uma instância de outro programa

# Chamando *fork*

- Quando um programa chama *fork*, uma cópia do processo, chamado *processo filho*, é criado.
  - Ambos executam o mesmo programa a partir do ponto onde *fork* foi chamado
    - Então como eles se diferenciam?
      - O processo filho é um novo processo e portanto tem um novo PID, diferente do PID de seu pai
      - Chamando *getpid* os processos podem saber quem é quem
      - A função *fork* também fornece valores de retorno diferentes para os processos pai e filho
        - » PID do filho é retornado para o processo pai
        - » Zero é retornado para o processo filho

# Implementação de um Exec



Aloca entrada na tabela de processo do filho  
Preenche entrada do filho com dados do pai  
Aloca pilha e área de usuário para o filho  
Preenche a área de usuário do filho com dados do pai  
Aloca PID para o filho  
Ajusta filho para compartilhar código do pai  
Copia tabelas de páginas para dados e pilha  
Ajusta compartilhamento de arquivos abertos  
Copia registradores do pai para o filho

Encontra o programa executável  
Verifica a permissão de execução  
Lê e verifica o cabeçalho  
Copia argumentos e variáveis de ambiente para o núcleo  
Libera o antigo espaço de endereçamento  
Aloca novo espaço de endereçamento  
Copia argumentos e variáveis de ambiente para a pilha  
Reinicializa os sinais  
Inicializa os registradores

**Figura 10.5** Passos na execução do comando `ls` digitado no shell.



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
    pid_t child_pid;

    printf ("o id do processo do programa principal é %d\n", (int) getpid ());

    child_pid = fork ();
    if (child_pid != 0) {
        printf ("este é o processo pai, com o id %d\n", (int) getpid ());
        printf ("o id do processo filho é %d\n", (int) child_pid);
    }
    else
        printf ("este é o processo filho, com o id %d\n", (int) getpid ());

    return 0;
}
```

# A família *exec*

- A função *exec* substitui o programa em execução em um processo por outro programa.
- Quando um programa chama *exec*, aquele processo imediatamente deixa de executar o programa e passa a executar um novo programa desde o início, desde que *exec* não encontre um erro.

# A família *exec*

- Dentro da família *exec*, há funções com pequenas variações em suas capacidades e forma de chamada
  - Funções que contém a letra *p* em seus nomes (*execvp* e *execlp*)
    - aceitam um nome de programa e procuram por um programa com este nome no caminho de execução atual; funções sem *p* precisam do caminho completo
  - Funções que contém a letra *v* em seus nomes (*execv*, *execvp*, e *execve*)
    - aceitam uma lista de argumentos para o novo programa como um *array* terminado em NULL de ponteiros para *strings*
  - Funções que contém a letra *l* em seus nomes (*execl*, *execlp*, e *execle*)
    - aceitam a lista de argumentos com o mecanismo *vararg* da linguagem C
  - Funções que contém a letra *e* em seus nomes (*execve* e *execle*)
    - aceitam um argumento adicional, um *array* de variáveis de ambiente. O argumento deve ser um *array* terminado em NULL de ponteiros para *strings* de caracteres. Cada *string* deve ter a forma “VARIÁVEL=valor”

# A família *exec*

- Por substituir o programa que o chama, *exec* nunca retorna a menos que ocorra um erro
- A lista de argumentos passada ao programa é análoga à lista de argumentos de um programa executado pelo *shell*
  - Disponíveis através de *argc* e *argv*
  - Lembre-se de passar *argv[0]* como o nome do programa, *argv[1]* como o primeiro argumento e assim por diante quando usar a função *exec*

# *fork e exec juntos*

- Um padrão comum para executar um subprograma é
  1. Bifurcar o processo com *fork*
  2. Executar o subprograma com *exec*
- O programa chamador
  - Continua sua execução no processo pai
  - É substituído pelo subprograma no processo filho

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

*/\* Gera um processo filho executando um novo programa. PROGRAM é o nome do programa a ser executado; ele será buscado no path. ARG\_LIST é uma lista terminada em NULL de strings de caracteres a serem passados como a lista de argumentos do programa. Retorna o id do processo gerado. \*/*

```
int spawn (char* program, char** arg_list)
{
    pid_t child_pid;

    /* Duplicar este processo. */
    child_pid = fork ();
    if (child_pid != 0)
        /* Este é o processo pai. */
        return child_pid;
    else {
        /* Agora execute PROGRAM, buscando-o no path. */
        execvp (program, arg_list);
        /* A função execvp só retorna se um erro ocorrer. */
        fprintf (stderr, "um erro ocorreu em execvp\n");
        abort ();
    }
}
```

```
int main ()
{
    /* A lista e argumentos para ser passada ao comando "ls". */
    char* arg_list[] = {
        "ls",      /* argv[0], o nome do programa. */
        "-l",
        "/",
        NULL       /* A lista de argumentos deve ser terminada com
        NULL. */
    };

    /* Gera um processo filho executando o comando "ls". Ignore o
    id do processo filho retornado. */
    spawn ("ls", arg_list);

    printf ("o programa principal terminou\n");

    return 0;
}
```

*fork-exec.c*

# Um shell Linux simples

```
while (TRUE) {                                /* repete para sempre */
    type_prompt( );                           /* mostra o prompt na tela */
    read_command(command, params);            /* lê a linha de entrada do teclado */

    pid = fork( );                            /* cria um processo filho */
    if (pid < 0) {
        printf("Unable to fork0);           /* condição de erro */
        continue;                          /* repete o laço */
    }

    if (pid != 0) {
        waitpid(-1, &status, 0);            /* pai espera o filho */
    } else {
        execve(command, params, 0);         /* filho traz o trabalho */
    }
}
```

**Figura 10.4** Um shell altamente simplificado.

# Escalonamento

- Determina quanto tempo de CPU um determinado processo recebe
- O kernel usa um algoritmo dinâmico para calcular prioridades, levando em conta:
  - A quantidade de tempo de CPU que o processo consumiu recentemente
  - O tempo que ele ficou aguardando para ser executado



# Escalonamento de Processos

- O escalonamento de processos pai e filho são independentes
  - Não há garantias de:
    - Qual executará primeiro
    - Quanto tempo executará antes de ser interrompido pelo Linux para que outro processo execute
  - O Linux só promete que cada processo eventualmente será executado

# Prioridade

- O escalonador normalmente faz um bom trabalho no gerenciamento da CPU (cada vez mais rápida), tornando a configuração manual de prioridades desnecessária na maioria dos casos
- O gargalo normalmente é no sistema de E/S (Ex.: discos rígidos), onde o valor do *nice* não tem nenhuma influência

# Nice e Renice

- A “gentileza” de um processo é uma dica numérica para o kernel em relação a como o processo deve ser tratado em relação aos outros processos lutando por recursos da CPU
- O intervalo de valores de nice vai de -20 a +19
  - Um valor “nice” (gentil) alto significa baixa prioridade
  - Um valor “nice” baixo significa alta prioridade
- O proprietário de um processo pode aumentar o “nice”, mas não diminuí-lo
- Root pode configurar “nice” da maneira que quiser
  - Pode até colocar um valor tão baixo que outros processos não poderão executar

# Nice e Renice

- O valor de *nice* pode ser configurado no momento da criação do processo com o comando **nice**:
  - `nice -n 5 -/bin/tarefademorada`
- Pode ser configurado com **renice**:
  - `renice -5 8829`
- Atenção: alguns *shell* (não *bash*) incluem um comando **nice** com sintaxe diferente do sistema

# Nice e Renice

- Para mudar a *gentileza* de um processo dentro de um programa, utilize a função **nice**
  - O argumento é um valor de incremento, que será adicionado ao valor de *gentileza* do processo que o chama.
    - Valor positivo aumenta a *gentileza* e, portanto, reduz prioridade do processo
  - Somente um processo com privilégio *root* pode executar um processo com um valor de *gentileza* negativo ou reduzir o valor de *gentileza* de um processo sendo executado
    - Somente um processo executando como *root* pode passar um valor negativo para a função **nice**

# Sinais

- Podem ser enviados entre processos como meio de comunicação
- Podem ser enviados pelo driver de terminal para extinguir, interromper ou suspender processos quando teclas especiais foram pressionadas (Ctrl+C, Ctrl+Z)
- Podem ser enviados pelo administrador (via *kill*)
- Podem ser enviados pelo kernel quando um processo comete uma infração (ex.: divisão por zero)

# Sinais

- Quando um sinal é recebido:
  - Se houver uma rotina de manipulação pra esse sinal em particular ela será usada
  - Caso contrário o *kernel* toma uma atitude padrão em nome do processo
- Os programas podem ignorar ou bloquear a chegada de sinais
  - Sinais ignorados são descartados
  - Sinais bloqueados são jogados em uma fila

# Sinais

Sinais que todo administrador de sistemas deve conhecer.

#	Nome	Descrição	Padrão	Pode capturar?	Pode bloquear?	core dump?
1	HUP	Suspender	Encerrar	Sim	Sim	Não
2	INT	Interromper	Encerrar	Sim	Sim	Não
3	QUIT	Abandonar	Encerrar	Sim	Sim	Sim
9	KILL	Destruir	Encerrar	Não	Não	Não
<sup>a</sup>	BUS	Erro de barramento	Encerrar	Sim	Sim	Sim
11	SEGV	Falha de segmentação	Encerrar	Sim	Sim	Sim
15	TERM	Encerramento por parte do software	Encerrar	Sim	Sim	Não
<sup>a</sup>	STOP	Parar	Parar	Não	Não	Não
<sup>a</sup>	TSTP	Parada de teclado	Parar	Sim	Sim	Não
<sup>a</sup>	CONT	Continuar após parada	Ignorar	Sim	Não	Não
<sup>a</sup>	WHINCH	Modificado por janela	Ignorar	Sim	Sim	Não
<sup>a</sup>	USR1	Definido pelo usuário	Encerrar	Sim	Sim	Não
<sup>a</sup>	USR2	Definido pelo usuário	Encerrar	Sim	Sim	Não

a. Varia conforme a arquitetura de hardware; veja **man 7 signal**.



# Sinais no LINUX

Sinal	Efeito
SIGABRT	Enviado para abortar um processo e forçar o despejo de memória (core dump)
SIGALRM	O relógio do alarme disparou
SIGFPE	Ocorreu um erro de ponto flutuante (por exemplo, divisão por 0)
SIGHUP	A linha telefônica que o processo estava usando caiu
SIGILL	O usuário pressionou a tecla DEL para interromper o processo
SIGQUIT	O usuário pressionou uma tecla solicitando o despejo de memória
SIGKILL	Enviado para matar um processo (não pode ser capturado ou ignorado)
SIGPIPE	O processo escreveu em um pipe que não tem leitores
SIGSEGV	O processo referenciou um endereço de memória inválido
SIGTERM	Usado para requisitar que um processo termine elegantemente
SIGUSR1	Disponível para propósitos definidos pela aplicação
SIGUSR2	Disponível para propósitos definidos pela aplicação

# Sinais

- KILL e STOP: não podem ser capturados, bloqueados ou ignorados
  - KILL destrói o processo
  - STOP suspende o processo até que um sinal CONT seja recebido
    - CONT pode ser capturado ou ignorado, mas não bloqueado
- TSPS é um STOP mais “soft” (solicitação de parada), é gerado quando pressionamos Ctrl+Z
  - Programas que recebem esse sinal normalmente limpam seus estados e enviam um STOP para eles mesmos
  - TSPS pode ser ignorado para impedir que um programa seja encerrado pelo teclado

# Sinais

- KILL: não pode ser bloqueado e encerra um processo em nível de S.O. Um processo jamais “recebe” esse sinal
- INT: enviado pelo driver de terminal quando digitamos Ctrl+C, programas simples podem sair (caso capturem o sinal) ou permitir que sejam eliminados (caso não capturem o sinal)
- TERM: solicitação para terminar completamente a execução, o processo deve limpar seu estado e sair

# Sinais

- HUP: (tem duas interpretações)
  - solicitação de reinicialização (utilizada por vários *daemons*)
  - Gerados pelo driver de terminal numa tentativa de limpar os processos agregados a um terminal (sessão concluída / conexão perdida)
- QUIT: similar a TERM, porém seu padrão gera um despejo de memória caso não seja capturado
  - Poucos programas canibalizam esse sinal e o interpretam de outra forma

# Sinais

- Sinais são assíncronos
  - Quando um processo recebe o sinal, ele processa o sinal imediatamente
    - Sem terminar a função atual ou mesmo a linha atual de código
- Em programas, geralmente referimos ao sinais utilizando seus nomes, definidos em `/usr/include/bits/signum.h`
  - ou `/usr/include/x86_64-linux-gnu/bits/signum.h`
  - Inclua `<signal.h>` para utilizá-los

# Sinais

- Para cada tipo de sinal, existe uma ação padrão, que determina o que acontece se o programa não especificar outro comportamento
- O sistema Linux envia sinais para processos em respostas a condições específicas:
  - SIGBUS (Erro de barramento)
  - SIGSEGV (Violação de segmentação)
  - SIGFPE (Exceção de ponto flutuante)
- Por padrão, estes sinais encerram o processo e gravam um despejo de memória

# Sinais

- A função **sigaction** pode ser usada para determinar o que um programa deve fazer ao receber um determinado sinal
  - Parâmetros
    - Número do sinal
    - Ponteiro para estrutura **sigaction** com ação desejada para o número de sinal
    - Ponteiro para estrutura **sigaction** que armazenará a ação anterior para o número de sinal
      - A que estava configurada antes de ser substituída
      - Pode ser usada para consulta, deixando o primeiro ponteiro em NULL

# Sinais

- O campo mais importante na estrutura **sigaction** é **sa\_handler**
  - Ele pode ter 3 valores:
    - SIG\_DFL
      - Trate o sinal com a ação padrão
    - SIG\_IGN
      - Ignore o sinal
    - Ponteiro para uma função que manipulará o sinal
      - A função deve aceitar um parâmetro (o número do sinal) e retornar **void**.
      - Nessa função, evite utilizar operações de entrada/saída e a maioria das chamadas de funções de biblioteca ou sistema



# Sinais

- O manipulador de sinal deve realizar o mínimo trabalho necessário para responder ao sinal e retornar o controle para o programa principal (ou encerrá-lo)
  - Na maioria dos casos ele simplesmente grava o fato de que o sinal ocorreu
    - O programa principal checa periodicamente se algum sinal ocorreu e reage de acordo
  - Pode acontecer do manipulador de sinal ser interrompido pela chegada de um novo sinal

# Sinais

- Até mesmo atribuir um valor para uma variável global em um manipulador de sinal pode ser perigoso
  - A atribuição pode se transformar em duas ou mais instruções de máquina
    - Outro sinal pode ocorrer enquanto a atribuição é feita e deixar a variável em um estado corrompido
  - Se usar uma variável global, use **sig\_atomic\_t**, que garante que atribuições sejam feitas com uma única instrução
    - No Linux **sig\_atomic\_t** é um **int** normal, pois atribuições para inteiros do tamanho de **int** ou menores, ou para ponteiros, são sempre atômicas
      - De qualquer forma, use **sig\_atomic\_t** para garantir portabilidade com sistemas UNIX

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
```

```
sig_atomic_t sigusr1_count = 0;
```

```
void handler (int signal_number)
{
    ++sigusr1_count;
}
```

```
int main ()
{
    struct sigaction sa;
    memset (&sa, 0, sizeof (sa));
    sa.sa_handler = &handler;
    sigaction (SIGUSR1, &sa, NULL);
```

```
/* Faça algo demorado aqui. */
/* ... */
```

```
printf ("SIGUSR1 foi chamado %d vezes\n", sigusr1_count);
return 0;
}
```

# Encerrando um Processo

- Normalmente, um processo termina em uma de duas formas:
  - O programa em execução chama a função **exit**
  - A função **main** retorna
- O código de saída de um programa é enviado para seu pai
  - O código de saída é o argumento da função **exit** ou o valor retornado por **main**

# Encerrando um Processo

- Um processo também pode terminar de forma anormal, em resposta a um sinal
  - SIGBUS, SIGSEGV e SIGFPE sinalizam erros que fazem o programa terminar
  - SIGINT é o sinal enviado quando o usuário digita Ctrl+C no terminal
  - SIGTERM é enviado pelo comando **kill** por padrão
    - A ação padrão para SIGINT e SIGTERM é encerrar o programa
  - SIGABRT finaliza o processo e gera um despejo de memória
    - Quando usamos a função **abort**, o programa envia um sinal SIGABRT para si mesmo
  - SIGKILL encerra o processo imediatamente e não pode ser bloqueado ou manipulado pelo programa

# Kill

- KILL pode enviar qualquer sinal, porém tipicamente é usado para encerrar um processo, por padrão o sinal enviado é TERM
  - Kill pode ser usado por usuários em seus próprios processos ou pelo root em qualquer processo
  - Sintaxe:
    - `kill [-sinal] pid`
      - sinal: número ou nome do sinal a ser enviado
      - pid: número de identificação do processo (use -1 para todos os processos exceto init)
  - Kill sem especificar o sinal não garante que o processo será terminado, pois TERM pode ser ignorado, bloqueado ou capturado. O sinal 9 não pode ser capturado e “garante” que o processo será eliminado.

# Kill

- Para enviar um sinal de um programa, use a função **kill**
  - O primeiro parâmetro é o ID do processo alvo
  - O segundo parâmetro é o número de sinal
    - Use SIGTERM para simular o comportamento padrão do comando **kill**
  - Exemplo: `child_pid` contém PID do processo filho
    - `kill(child_pid, SIGTERM)`
      - Inclua `<sys/types.h>` e `<signal.h>` para usar a função **kill**

# Exercício 1

- a) Faça um programa que manipule o sinal **USR1**,
- b) No manipulador, incremente um contador (variável global inicializada em zero).
- c) No programa principal, inclua um laço *for* com 30.000 iterações.
- d) Em cada iteração do laço, use *usleep*(1000).
- e) Imprima o valor do contador ao final das iterações.
- f) Compile o programa criado.
- g) Execute o programa em segundo plano.
- h) Envie o sinal **USR1** para o processo do programa algumas vezes utilizando o comando **kill**.
- i) O que acontece? Por que?

**Atenção:** Coloque a resposta dos exercício 1, 3 e 4 no Moodle. Escreva diretamente no campo apropriado ou anexe arquivo texto (.txt) ou PDF (.pdf).

```
fabricio@fabricio-virtual-machine:~/so2/aula4$ ./sigusr1 &
[1] 11399
fabricio@fabricio-virtual-machine:~/so2/aula4$ kill -s SIGUSR1 11399
fabricio@fabricio-virtual-machine:~/so2/aula4$ kill -s SIGUSR1 11399
fabricio@fabricio-virtual-machine:~/so2/aula4$ kill -s SIGUSR1 11399
fabricio@fabricio-virtual-machine:~/so2/aula4$ kill -s SIGUSR1 11399
fabricio@fabricio-virtual-machine:~/so2/aula4$
fabricio@fabricio-virtual-machine:~/so2/aula4$ SIGUSR1 foi chamado 4 vezes
```



# Encerrando um Processo

- Use o código de saída 0 para indicar que o programa foi encerrado corretamente
- Use códigos de 1 a 127 para indicar erros
- Códigos acima de 128 indicam terminos por sinal (128 + código do sinal)
- Na maioria dos shells, verifique código de saída do último programa executado com:
  - **echo \$?**

# Esperando um processo terminar

- Nos exemplos com *fork* e *exec* você deve ter notado que a saída do *ls* frequentemente aparece depois que o programa principal foi concluído
  - Isto acontece porque o processo filho é escalonado independentemente do processo pai
  - Como o Linux é multitarefa, ambos parecem executar simultaneamente
  - Não é possível prever se *ls* terá chance de executar antes que o programa principal termine

# Esperando um processo terminar

- Em alguns situações é desejável que o processo pai espere que um ou mais processos filhos tenham terminado
  - Isto pode ser feito com a família de chamadas de sistema **wait**
    - Permite esperar que um processo termine
    - Permite obter informações sobre o término dos filhos

# Chamadas de Sistema *wait*

- **wait**
  - A mais simples. Bloqueia o processo chamador até que um de seus filhos termine (ou um erro ocorra)
  - Retorna um código de status através de um argumento ponteiro, do qual podem ser extraídas informações de como um processo filho terminou
  - A macro WEXITSTATUS extrai o código de saída do processo filho
  - A macro WIFEXITED determina, pelo código de saída do processo filho, se ele terminou normalmente (com a função **exit** ou retornando de **main**) ou se morreu de um sinal não manipulado
    - No último caso a macro WTERMSIG extrai de seu código de saída, o número do sinal pelo qual ele morreu

# Exercício 2

- Modifique *fork-exec.c* (salve com outro nome), para esperar o encerramento do processo filho

```
int main ()
{
    int child_status;

    /* A lista e argumentos para ser passada ao comando "ls". */
    char* arg_list[] = {
        "ls",      /* argv[0], o nome do programa. */
        "-",
        "/",
        NULL      /* A lista de argumentos deve ser terminada com NULL. */
    };

    /* Gera um processo filho executando o comando "ls". Ignore o
       id do processo filho retornado. */
    spawn ("ls", arg_list);

    /* Espere o processo filho completar. */
    wait (&child_status);
    if (WIFEXITED (child_status))
        printf ("O processo filho encerrou normalmente, com código de saída %d\n",
                WEXITSTATUS (child_status));

    else
        printf("O processo filho terminou anormalmente");

    return 0;
}
```

```
fabricio@fabricio-virtual-machine:~/so2/aula4$ ./ex2
total 92
drwxr-xr-x  2 root root 4096 Mar 17 12:41 bin
drwxr-xr-x  3 root root 4096 Mar 25 19:48 boot
drwxr-xr-x  2 root root 4096 Mai  2 2012 cdrom
drwxr-xr-x 15 root root 4260 Mar 25 19:41 dev
drwxr-xr-x 132 root root 12288 Mar 25 19:48 etc
drwxr-xr-x  3 root root 4096 Mai  2 2012 home
lrwxrwxrwx  1 root root   33 Mar 25 19:48 initrd.img -> /boot/initrd.img-3.2.0-39-generic
lrwxrwxrwx  1 root root   33 Mar 17 12:43 initrd.img.old -> /boot/initrd.img-3.2.0-38-generic
drwxr-xr-x 22 root root 4096 Mar 17 12:39 lib
drwxr-xr-x  2 root root 4096 Mar 17 12:31 lib64
drwx----- 2 root root 16384 Mai  2 2012 lost+found
drwxr-xr-x  3 root root 4096 Mar 17 01:05 media
drwxr-xr-x  3 root root 4096 Mar 17 11:33 mnt
drwxr-xr-x  2 root root 4096 Abr 25 2012 opt
dr-xr-xr-x 166 root root    0 Mar 25 19:41 proc
drwx----- 3 root root 4096 Mai  2 2012 root
drwxr-xr-x 22 root root  880 Mar 25 19:48 run
drwxr-xr-x  2 root root 4096 Mar 17 12:41/sbin
drwxr-xr-x  2 root root 4096 Mar  5 2012 sellinux
drwxr-xr-x  2 root root 4096 Abr 25 2012 srv
drwxr-xr-x 13 root root    0 Mar 25 19:41 sys
drwxrwxrwt 12 root root 4096 Mar 25 23:33 tmp
drwxr-xr-x 10 root root 4096 Abr 25 2012 usr
drwxr-xr-x 13 root root 4096 Mar 17 12:55 var
lrwxrwxrwx  1 root root   29 Mar 25 19:48 vmlinuz -> boot/vmlinuz-3.2.0-39-generic
lrwxrwxrwx  1 root root   29 Mar 17 12:43 vmlinuz.old -> boot/vmlinuz-3.2.0-38-generic
0 processo filho encerrou normalmente, com código de saída 0
fabricio@fabricio-virtual-machine:~/so2/aula4$
```

Obs: Teoricamente é necessário incluir o cabeçalho `<sys/wait.h>`, apesar do programa funcionar sem ele em alguns casos.

# Chamadas de Sistema *wait*

- Mais algumas chamadas da família **wait**:
  - **waitpid**
    - Espera por um processo filho específico em vez de qualquer processo.
      - -1 para esperar qualquer processo filho, equivalente ao **wait**
  - **waitid**
    - Permite maior controle sobre quais mudanças de estado do filho devem ser esperadas
  - **wait3**
    - Retorna estatísticas de uso de CPU sobre o processo filho.
  - **wait4**
    - Permite especificar opções adicionais sobre o processo a ser esperado.

Obs: **wait3** e **wait4** estão obsoletas e suas funcionalidades podem ser obtida por **waitpid** ou **waitid**

# Processos Zumbis

- Se um processo filho termina enquanto seu pai está chamando uma função **wait**
  - Processo filho desaparece e seu status de término é passado ao pai através da função **wait**
- Se um processo filho termina e o pai não está chamando **wait**
  - Processo filho não desaparece
    - Informações de término (normal ou anormal, código de saída, etc.) seriam perdidas
  - Ao terminar, o processo filho se torna um processo zumbi

# Processos Zumbis

- Um processo zumbi é um processo que terminou mas ainda não foi eliminado
  - É responsabilidade do processo pai limpar seu filho zumbi
    - A função **wait** faz isso
  - Suponha que um programa use **fork**, depois faz alguns cálculos e então chame **wait**
    - Se o processo filho não terminou, o processo pai bloqueia em **wait** e espera
    - Se o processo filho termina antes, ele vira zumbi
      - Quando o pai chamar **wait**, seu status de término será extraído, o processo filho será excluído e a função **wait** retornará imediatamente



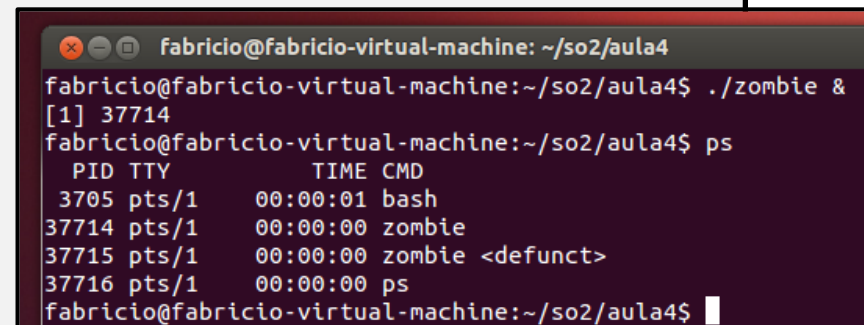
# Processos Zumbis

- E se o pai não limpar o filho?
  - Ele ficará no sistema como um processo zumbi

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
    pid_t child_pid;

    /* Cria um processo filho. */
    child_pid = fork ();
    if (child_pid > 0) {
        /* Este é o processo pai. Dormir por um minuto. */
        sleep (60);
    }
    else {
        /* Este é o processo filho. Sair imediatamente. */
        exit (0);
    }
    return 0;
}
```



A terminal window titled 'fabricio@fabricio-virtual-machine: ~/so2/aula4' showing the execution of the 'zombie.c' program. The user runs './zombie &' which returns '[1] 37714'. Then, the user runs 'ps' which displays a list of processes. The output shows the parent process (PID 3705) as 'bash', the child process (PID 37714) as 'zombie', and a defunct process (PID 37715) as 'zombie <defunct>'. The parent process (PID 37716) is 'ps'.

PID	TTY	TIME	CMD
3705	pts/1	00:00:01	bash
37714	pts/1	00:00:00	zombie
37715	pts/1	00:00:00	zombie <defunct>
37716	pts/1	00:00:00	ps

zombie.c

# Exercício 3

- a) Compile *zombie.c* para um executável chamado **make-zombie**
- b) Execute **make-zombie** e enquanto ele ainda estiver executando, liste os processos no sistema invocando o seguinte comando:
  - ps -e -o pid,ppid,stat,cmd
- c) O que acontece quando o processo pai termina, sem nunca ter chamado **wait**? Por que?

**Atenção:** Coloque a resposta dos exercício 1, 3 e 4 no Moodle. Escreva diretamente no campo apropriado ou anexe arquivo texto (.txt) ou PDF (.pdf).

# Processos Zumbis

```
9782  2  S   [Kworker/0:1]
9819  7531 S   ./make-zombie
9820  9819 Z   [make-zombie] <defunct>
9821  7531 R+  ps -e -o pid,ppid,stat,cmd
fabricao@fabricao-virtual-machine:~/aula4$
```

- Note o status Z de zumbi
- Quando um programa termina, seus filhos são herdados por **init**, cujo PID é sempre 1
  - **init** é o primeiro processo quando Linux dá boot
  - **init** automaticamente limpa qualquer processo filho zumbi que ele herda

# Limpando filhos assincronamente

- Se você vai continuar executando tarefas no processo pai após um **fork**
  - Não chamará **wait** imediatamente, pois isso bloquearia o processo pai
  - Por outro lado, demorar a chamar **wait** poderia deixar processos filhos zumbis consumindo recursos do sistema inutilmente
- Como resolver essa situação?

# Limpendo filhos assincronamente

- Uma solução: utilizar **waitpid**, **wait3** ou **wait4** periodicamente
  - Tem um parâmetro de *flags* que pode ser ajustado para WNOHANG
    - Modo não bloqueante
      - Se há um filho terminado, ele faz a limpeza; se não há ele simplesmente retorna
      - O valor de retorno é o ID do filho se houve limpeza ou zero se não houve

# Limpando filhos assincronamente

- Uma solução mais elegante: notificar o processo pai quando um filho termina
  - O Linux já faz isso pra você usando sinais
    - Quando um processo filho termina, o Linux envia SIGCHLD ao pai, cuja ação padrão é não fazer nada
    - Criar um manipulador para SIGCHLD é a solução
      - Lembre-se de armazenar o status de término do filho se ele for necessário

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
sig_atomic_t child_exit_status;
```

```
void clean_up_child_process (int signal_number)
{
    /* Limpa o processo filho. */
    int status;
    wait (&status);
    /* Guarda o código de saída em uma variável global. */
    child_exit_status = status;
}
```

```
int main ()
{
    /* Manipula SIGCHLD chamando clean_up_child_process. */
    struct sigaction sigchld_action;
    memset (&sigchld_action, 0, sizeof (sigchld_action));
    sigchld_action.sa_handler = &clean_up_child_process;
    sigaction (SIGCHLD, &sigchld_action, NULL);

    /* Agora faça coisas, incluindo criar um processo filho com fork. */
    /* ... */

    return 0;
}
```

*sigchld.c*

# Exercício 4

- Modifique *sigchld.c* para criar processos filhos que viverão por um tempo razoavelmente longo
- Em um segundo terminal, use **ps -e -o pid,ppid,stat,cmd** para verificar a atividade dos filhos enquanto o pai ainda está em execução.
- Algum fica como zumbi? Por que?

```
pid_t child_pid;
int i=0;
int t=-1;
while (t!=0)
{
    printf("Digite quantos segundos o filho deve viver (ou 0 para encerrar): ");
    while(t==-1) scanf("%i",&t);
    if (t==0) return(0);
    i++;
    printf("Criando %dº processo filho, viverá %d segundos\n",i,t);
    child_pid = fork ();
    if (child_pid == 0)
    {
        sleep(t);
        return(0);
    }
    t=-1;
}
```

*Sugestão de código*

**Atenção:** Coloque a resposta dos exercício 1, 3 e 4 no Moodle. Escreva diretamente no campo apropriado ou anexe arquivo texto (.txt) ou PDF (.pdf).



# Referências Bibliográficas

1. [NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.; \*Manual Completo do Linux: Guia do Administrador\*. São Paulo: Pearson Prentice Hall, 2007. Cap. 4](#)
2. [DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; \*Sistemas Operacionais: terceira edição\*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20](#)
3. [MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; \*Advanced Linux Programming\*. New Riders Publishing: 2001. Cap. 3](#)
4. [TANENBAUM, Andrew S.; \*Sistemas Operacionais Modernos\*. 3ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 10](#)

