



Laboratório de Pesquisa em Redes e Multimídia

# SVCs para Controle de Processos no Unix

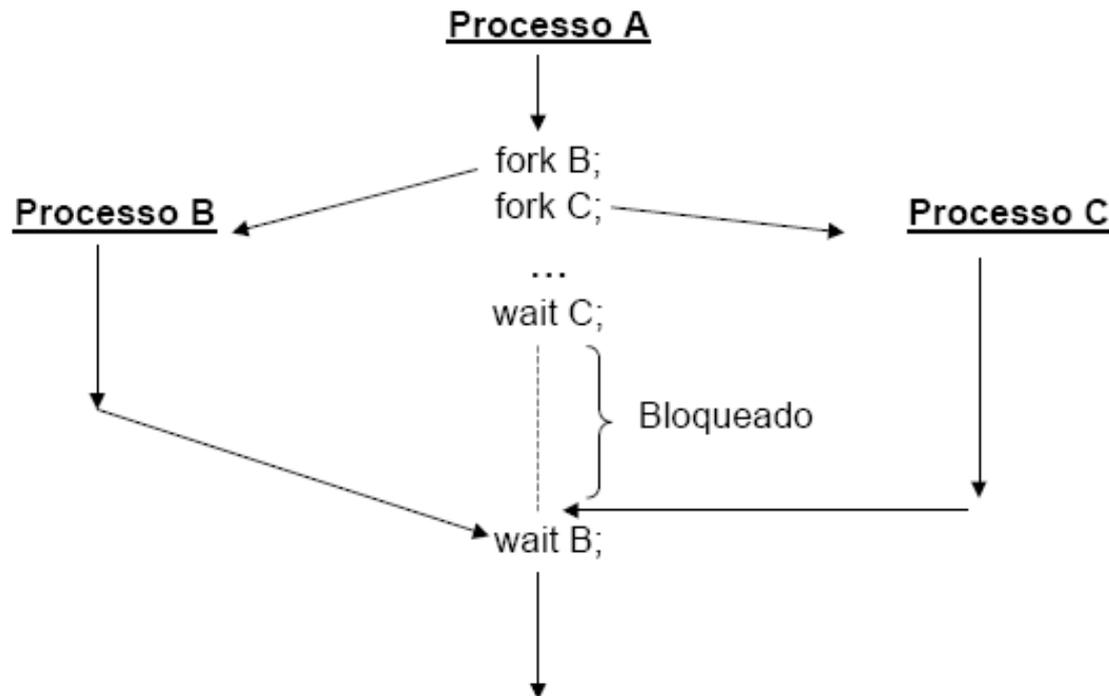


Universidade Federal do Espírito Santo  
Departamento de Informática

**Sistemas Operacionais**

# Criação de Processos

- A maioria dos sistemas operacionais usa um mecanismo de *spawn* para criar um novo processo a partir de um outro executável.



## Criação de Processos no UNIX

- No Unix, são usadas duas funções distintas relacionadas à criação e execução de programas. São elas:
  - `fork()`: cria processo filho idêntico ao pai, exceto por alguns atributos e recursos.
  - `exec()`: carrega e executa um novo programa.
- A sincronização entre processo pai e filho(s) é feita através da SVC `wait()`, que bloqueia o processo pai até que um processo filho termine.

## A SVC fork()

- No Unix, a forma de se criar um novo processo (dito processo filho) é invocando a chamada ao sistema `fork()`.

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

```
pid_t fork(void);
```

Retorna:

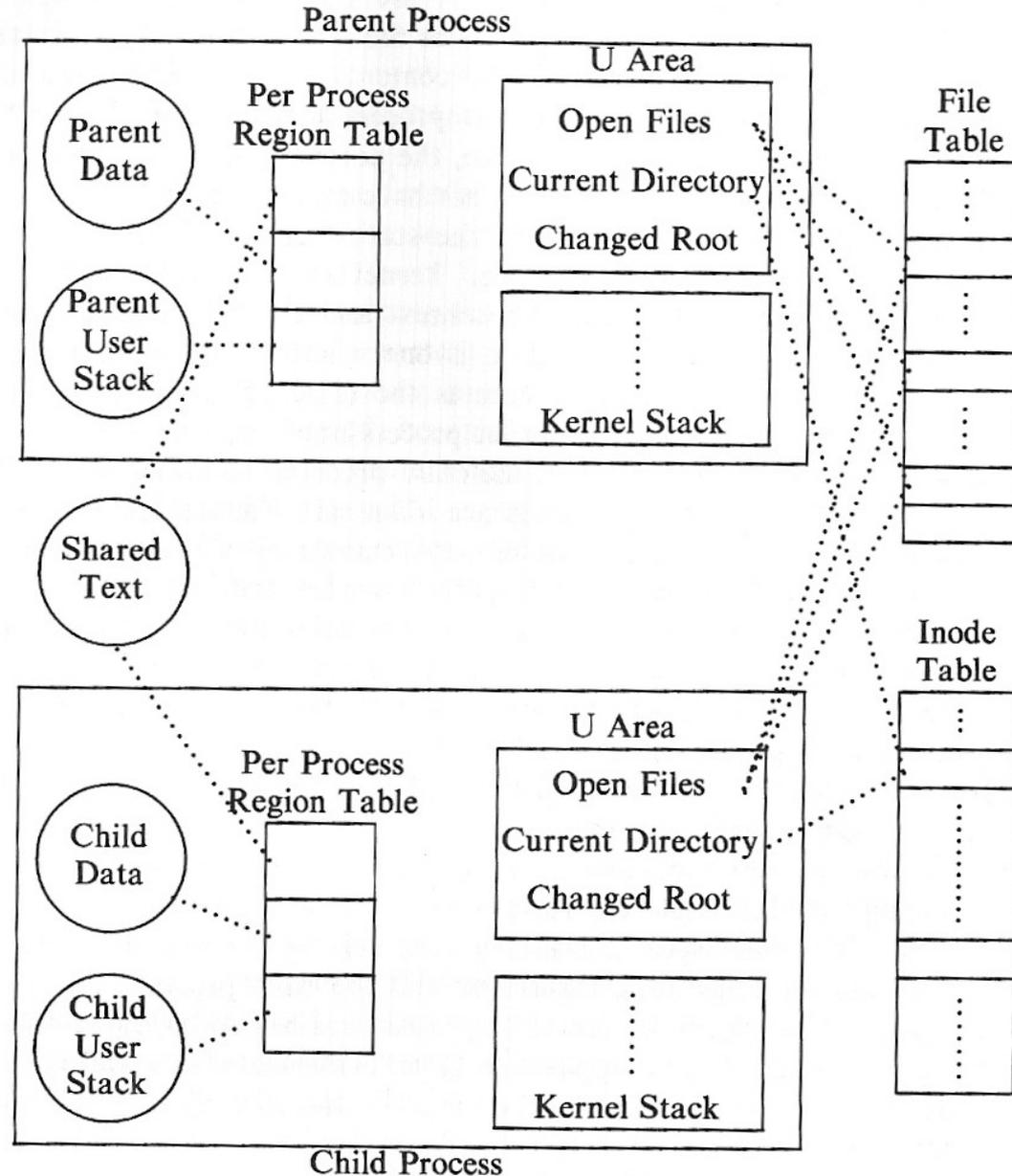
- 0 - para o processo filho
- pid do filho - para o processo pai
- 1 - se houve erro e o serviço não foi executado

- `Fork()` duplica/clona o processo que executa a chamada. O processo filho é uma cópia fiel do pai, ficando com uma cópia do segmento de dados, *heap* e *stack* (obs: o segmento de texto/código é muitas vezes partilhado por ambos).
- Processos pai e filho continuam a sua execução na instrução seguinte à chamada `fork()`.
- Em geral, não se sabe quem continua a executar imediatamente após uma chamada a `fork()`, se é o pai ou o filho. Isso depende do algoritmo de escalonamento.

## A SVC fork() (cont.)

- O processo filho tem seu próprio espaço de endereçamento, com cópia de todas as variáveis do processo pai. Essas são independentes em relação às variáveis do processo pai.
- O processo filho herda do pai alguns atributos, tais como: variáveis de ambiente, variáveis locais e globais, privilégios e prioridade de escalonamento.
- O processo filho também herda alguns recursos, tais como arquivos abertos e *devices*. Alguns atributos e recursos, tais como PID, PPDI, sinais pendentes e estatísticas do processo, não são herdados pelo processo filho.

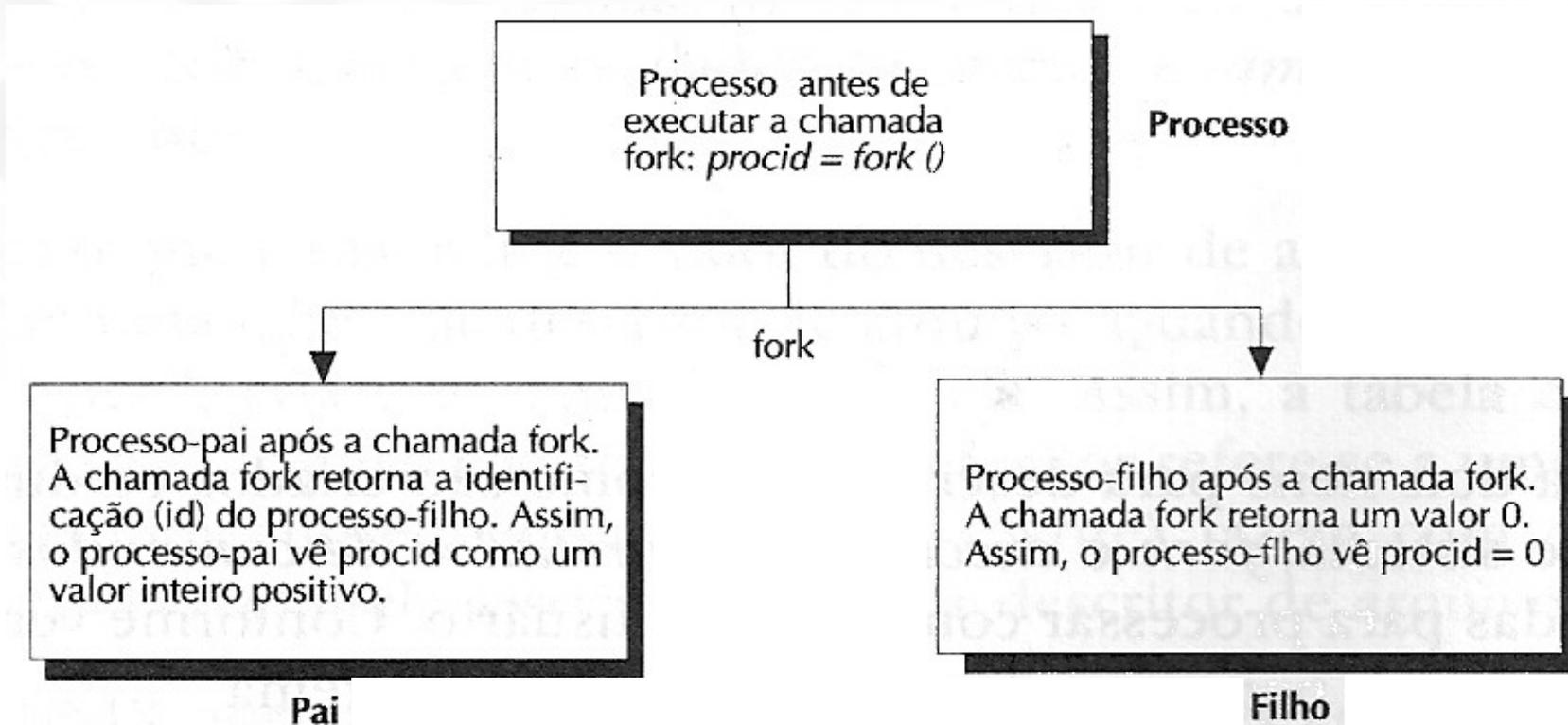
# A SVC fork() (cont.)



## A SVC fork() (cont.)

- A função `fork()` é invocada uma vez (no processo-pai) mas retorna duas vezes: uma no processo que a invocou e outra no novo processo criado, o processo-filho.
- O retorno da função `fork()`, no processo pai, é igual ao número do *pid* do processo filho recém criado (todos os processos em Unix têm um identificador, geralmente designado por *pid* – *process identifier*).
- O retorno da função `fork()` é igual a 0 (zero) no processo filho.

## A SVC fork() (cont.)



## Estrutura Geral do fork()

```
pid=fork();
if(pid < 0) {
    /* falha do fork */
}
else if (pid > 0) {
    /* código do pai */
}
else { //pid == 0
    /* código do filho */
}
```

## Copy-on-Write

- Como alternativa a significativa ineficiência do `fork()`, no Linux o `fork()` é implementado usando uma técnica chamada *copy-on-write* (COW).
- Essa técnica atrasa ou evita a cópia dos dados.
  - Ao invés de copiar o espaço de endereçamento do processo pai, ambos podem compartilhar uma única cópia somente de leitura.
  - Se uma escrita é feita, uma duplicação é realizada e cada processo recebe uma cópia.
  - Conseqüentemente, a duplicação é feita apenas quando necessário, economizando tempo e espaço.
- O único *overhead* inicial do `fork()` é a duplicação da tabela de páginas do processo pai e a criação de um novo *proc Struct* (c/ PID para o filho).

# Identificação do Processo no UNIX

- Como visto, todos os processos em Unix têm um identificador, geralmente designados por *pid* (*process identifier*). Os identificadores são números inteiros diferentes para cada processo (ou melhor, do tipo `pid_t` definido em `sys/types.h`).
- É sempre possível a um processo conhecer o seu próprio identificador e o do seu pai. Os serviços a utilizar para conhecer `pid`'s (além do serviço `fork()`) são:

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

pid_t getpid(void);           /* obtém o seu próprio pid */
pid_t getppid(void);        /* obtém o pid do pai */
```

Estas funções são sempre bem sucedidas.

## User ID e Group ID

- No Unix, cada processo tem de um proprietário, um usuário que seja considerado seu dono. Através das permissões fornecidas pelo dono, o sistema sabe quem pode e não pode executar o processo em questão.
- Para lidar com os donos, o Unix usa os números UID (*User Identifier*) e GID (*Group Identifier*). Os nomes dos usuários e dos grupos servem apenas para facilitar o uso humano do computador.
- Cada usuário precisa pertencer a um ou mais grupos. Como cada processo (e cada arquivo) pertence a um usuário, logo esse processo pertence ao grupo de seu proprietário. Assim sendo, cada processo está associado a um UID e a um GID.
- Os números UID e GID variam de 0 a 65536. Dependendo do sistema, o valor limite pode ser maior. No caso do usuário *root*, esses valores são sempre 0 (zero). Assim, para fazer com que um usuário tenha os mesmos privilégios que o *root*, é necessário que seu GID seja 0.
- Primitivas: `uid_t getuid(void) / uid_t geteuid(void)`  
`gid_t getgid(void) / gid_t getegid(void)`
- Comandos úteis: `id` (imprime user/group ID e effective ID)
- Arquivos úteis: `/etc/passwd`, `/etc/group`

## User ID e Group ID (cont.)

- Primitivas:
  - P/ user: `uid_t getuid(void) / uid_t geteuid(void)`
  - P/ group: `gid_t getgid(void) / gid_t getegid(void)`
- Comandos úteis:
  - `id`: lista os ID's do usuário e do seu grupo primário. Lista também todos os outros grupos nos quais o usuário participa.
- Arquivos úteis:
  - `/etc/passwd`
  - `/etc/group`
- Formato do arquivo `/etc/passwd`:
  - `usuário:senha:UID:GID:grupo primário do usuário:nome do usuário:diretório home:shell inicial`
- Formato do arquivo `/etc/group`:
  - `grupo:senha:GID:lista dos usuários do grupo`

## Exemplo 1 - Exibindo PID's (arquivo output\_IDs.c - exemplo3-2)

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

int main (void) {
    printf("I am process %ld\n", (long)getpid());
    printf("My parent is %ld\n", (long)getppid());

    printf("My real user ID is      %5ld\n", (long)getuid());
    printf("My effective user ID is  %5ld\n", (long)geteuid());
    printf("My real group ID is      %5ld\n", (long)getgid());
    printf("My effective group ID is %5ld\n", (long)getegid());

    return 0;
}
```

## Exemplo 2: Fork Simples (arquivo simple\_fork.c - exemplo 3.5)

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

int main(void) {
    int x;

    x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x is %d\n",
          (long)getpid(), x);
    return 0;
}
```

## Ex.3 - Diferenciando Pai e Filho

(arquivo two\_procs.c)

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int glob = 6;
int main(void) {
    int var;
    initialized data */
    pid_t pid;
    stack */
    var = 88;
    printf("before fork\n");
    if ( (pid = fork()) < 0)
        fprintf(stderr, "fork error\n");
    else if (pid == 0) {
        glob++;
        var++;
    }
    else
        sleep(60);
    that child ends first*/
    printf("pid = %d, ppid = %d, glob = %d, var = %d\n", getpid(),
    getppid(), glob, var);
    return 0;
}
```

/\* external variable in

/\* automatic variable on the

/\* \*\*\*child\*\*\* \*/

/\* modify variables \*/

/\* \*\*\*parent\*\*\*; try to guarantee

## Exemplo 4 - mypid x gettpid (arquivo myPID.c - exercise3-7)

```
#include <stdio.h>
#include <unistd.h>
/* #include <sys/type.h> */

int main(void) {
    pid_t childpid;
    pid_t mypid;

    mypid = getpid();
    childpid = fork();
    if (childpid == -1) {
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0)
        printf("I am child %ld, ID = %ld\n", (long int) getpid(),
(long int) mypid);
    else
        printf("I am parent %ld, ID = %ld\n", (long int) getpid(),
(long int) mypid);
    return 0;
}
```

## Exemplo 5 – Simple Chain (arquivo simple\_chain.c – exercise 3-10)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2){ /* check for valid number of command-line arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

## Exemplo 6 – Simple Fan (arquivo simple\_fan.c – exercise 3-10)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    /* check for valid number of command-line arguments */
    ...
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) <= 0) //only the child (or error) enters
            break;

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

## Exemplo 7 – Chain Geral (arquivo chain\_geral.c – exercise 3-10)

```
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    /* check for valid number of command-line arguments */

    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) == -1) // father and children enter
            break;

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

# O Comando ps

(retirado de *man ps*) *By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker. It displays process ID (pid=PID), terminal associated with the process (tname=TTY), cumulated CPU time in [dd-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD). Output is unsorted by default.*

## Alguns tributos:

- a Lista todos os processos
- e Mostra as variáveis associadas aos processos
- f Mostra a árvore de execução dos processos
- l Mostra mais campos
- u Mostra o nome do usuário e a hora de início
- x Mostra os processos que não estão associados a terminais
- t Mostra todos os processos do terminal

## Opções interessantes:

- \$ ps Lista os processos do usuário associados ao terminal
- \$ ps l Idem, com informações mais completas
- \$ ps a Lista também os processos não associados ao terminal
- \$ ps u Lista processos do usuário
- \$ ps U <user> ou \$ps -u <user> Lista processos do usuário <user>
- \$ ps p <PID> Lista dados do processo PID
- \$ ps r Lista apenas os processos no estado running
- \$ ps al, \$ ps ux, \$ ps au, \$ ps aux

## O Comando ps (cont.)

```
ctic-ufes@ctic-ufes:~/Documentos/Exemplos-SO$ ps
```

PID	TTY	TIME	CMD
2464	pts/0	00:00:00	bash
2885	pts/0	00:00:00	ps

```
ctic-ufes@ctic-ufes:~/Documentos/Exemplos-SO$ ps -la
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1000	2608	2592	0	80	0	-	888	wait	pts/1	00:00:00	man
0	S	1000	2618	2608	0	80	0	-	847	n_tty_	pts/1	00:00:00	pager
0	R	1000	2878	2464	0	80	0	-	626	-	pts/0	00:00:00	ps

F-flags, S-state, C-CPU utilization for scheduling (uso muito baixo é reportado como zero), NI-nice value, ADDR- process memory address, SZ-size in blocks of the process image, WCHAN-rotina do kernel onde o processo dorme (processos em execução são marcados com hífen), TIME-cumulative execution time, CMD-command name.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
spereira	6364	0.2	0.6	7892	1552	?	S	10:12	0:00	sshd: spereira@pts/3
spereira	6365	0.5	1.2	5588	3080	pts/3	Ss	10:12	0:00	-bash
spereira	6398	0.2	0.6	7892	1552	?	S	10:13	0:00	sshd: spereira@pts/4
spereira	6399	1.9	1.1	5584	3068	pts/4	Ss	10:13	0:00	-bash
spereira	6418	0.5	0.7	4432	1828	pts/4	S+	10:13	0:00	vi
spereira	6423	0.0	0.3	2560	996	pts/3	R+	10:13	0:00	ps -U spereira u

- **USER** – Nome do Utilizador
- **PID** – Número do Processo
- **%CPU** – Percentagem de utilização do CPU
- **%MEM** – Percentagem de utilização de memória
- **VSZ** – Tamanho da memória virtual
- **RSS** – Tamanho da memória residente
- **TTY** – Número do terminal, se tivermos várias sessões, temos diferentes terminais
- **STAT** – Estado actual do processo
- **START** – Hora em que o processo foi iniciado
- **TIME** – Tempo de processamento consumido
- **COMMAND** – Nome do comando

#### Estados dos Processos

D	Processo em <i>"Uninterruptible sleep"</i>
R	Processo a correr
S	Processo suspenso
T	Processo parado
X	Processo morto (não aparece na lista dos processos)
Z	Processo <i>Zombie</i> , está terminado, mas está ligado pelo processo que o iniciou

<	Corre em alta prioridade
N	Corre em baixa prioridade
L	Aloca as páginas na memória
s	Líder de sessão, garante que o processo termina quando o user faz <i>logout</i>
l	Processo em multi-thread
+	Corre em <i>foreground</i>

## O Comando ps (cont.)

## Processo Zombie

- Um processo que termina não pode deixar o sistema até que o seu pai aceite o seu código de terminação (valor retornado por `main()` ou passado a `exit()`), através da execução de uma chamada aos serviços `wait()` / `waitpid()`.
- Um processo que terminou, mas cujo pai ainda não executou um dos `wait`'s passa ao estado "zombie". Na saída do comando `ps` o estado destes processos aparece como `Z` e o seu nome identificado como `<defunct>`.
- Quando um processo passa ao estado de zombie a sua memória é liberada mas permanece no sistema alguma informação sobre ele (processo continua ocupando a tabela de processos do kernel).
- Se o processo pai terminar antes do filho, esse torna-se órfão e é adotado pelo processo `init` (`PID=1`).

## Exemplo 1 - Zombie(1) (arquivo testa\_zombie\_1.c)

```

/* rodar o programa em background */
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid ;
    printf("Eu sou o processo pai, PID = %d, e eu vou criar um filho.\n",getpid()) ;
    Printf Agora estou entrando em um loop infinito. Tchau!\n") ;
    pid = fork() ;
    if(pid == -1) /* erro */
    {
        perror("E impossivel criar um filho") ;
        exit(-1) ;
    }
    else if(pid == 0) /* filho */
    {
        printf("Eu sou o filho, PID = %d. Estou vivo mas vou dormir um pouco. Enquanto isso, use o comando ps -l
para conferir o meu PID, o meu estado (S=sleep), o PID do meu pai e o estado do meu pai (R=running). Daqui a
pouco eu acordo.\n",getpid()) ;
        sleep(60) ;
        printf("Acordei! Vou terminar agora. Confira novamente essas informações. Nãããoooooooo!!! Virei um
zumbi!!!\n") ;
        exit(0) ;
    }
    else /* pai */
    {
        for(;;) ; /* pai bloqueado em loop infinito */
    }
}

```

```
/* rodar o programa em foreground */
```

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

## Exemplo 2 - Zombie(2) (arquivo testa\_zombie\_2.c)

```
int main()
{
    int pid ;
    printf("Eu sou o processo pai, PID = %d, e eu vou criar um filho.\n",getpid()) ;
    printf("Bem, eu já coloquei mais um filho no mundo e agora vou terminar. Fui!\n") ;
    pid = fork() ;
    if(pid == -1) /* erro */
    {
        perror("E impossivel criar um filho") ;
        exit(-1) ;
    }
    else if(pid == 0) /* filho */
    {
        printf("Eu sou o filho, PID = %d. Estou vivo mas vou dormir um pouco. Use o comando ps -l para conferir
o meu estado (S=sleep) e o PID do meu pai. Notou algo diferente no PID do meu pai? Notou que eu não virei
um zumbi? Daqui a pouco eu acordo.\n",getpid()) ;
        sleep(60) ;
        printf("Acordei! Vou terminar agora. Use ps -l novamente.\n") ;
        exit(0) ;
    }
    else /* pai */
    {
        /* for(;;) ; pai bloqueado em loop infinito */
    }
}
```

## Exemplo 3 - Zombie(3) (arquivo testa\_zombie\_3.c)

```
/* rodar em background */
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid ;
    printf("Eu sou o processo pai, PID = %d, e eu vou criar um filho.\n",getpid()) ;
    printf("Bem, agora eu vou esperar pelo término da execução do meu filho. Tchau!\n") ;
    pid = fork() ;
    if(pid == -1) /* erro */
    {
        perror("E impossivel criar um filho") ;
        exit(-1) ;
    }
    else if(pid == 0) /* filho */
    {
        printf("Eu sou o filho, PID = %d. Estou vivo mas vou dormir um pouco. Use o comando ps -l para
conferir o meu estado e o do meu pai. Daqui a pouco eu acordo.\n",getpid()) ;
        sleep(60) ;
        printf("Sou eu de novo, o filho. Acordei mas vou terminar agora. Use ps -l novamente.\n") ;
        exit(0) ;
    }
    else /* pai */
    {
        wait(NULL) ; /* pai esperando pelo término do filho */
    }
}
```

## Exercício - Montar a Árvore de Processos

```
c2 = 0;
c1 = fork();           /* fork number 1 */
if (c1 == 0)
    c2 = fork();      /* fork number 2 */
fork();               /* fork number 3 */
if (c2 > 0)
    fork();           /* fork number 4 */
exit();
```