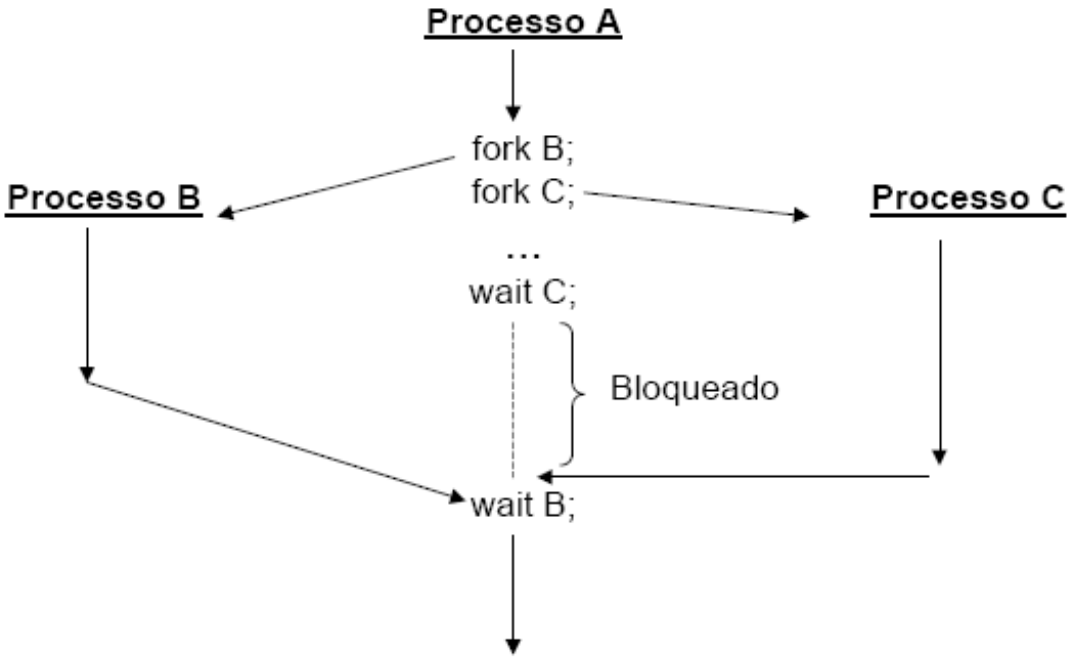
Supervisor Calls (SVCs) para Controle de Processos no Unix

Criação de Processos (1)

- 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()* (1)

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

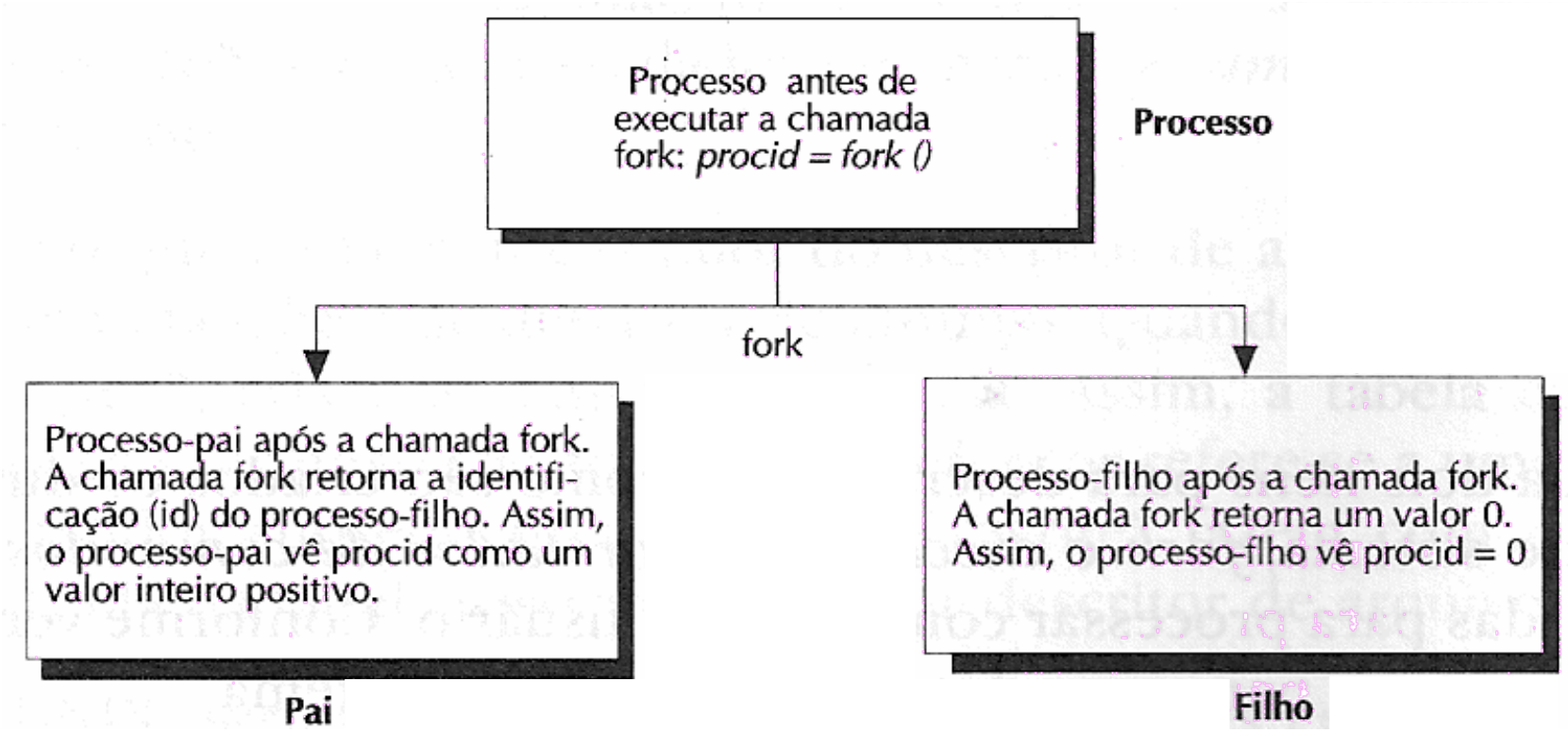
```
#include <sys/types.h>
#include <unistd.h>

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*; no entanto, 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). Depende do algoritmo de escalonamento.

A SVC *fork()* (2)

- 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 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 também herda alguns recursos, tais como arquivos abertos e *devices*. Alguns atributos e recursos, tais como PID, PPID, sinais pendentes e estatísticas do processo, não são herdados pelo processo filho.
- A função `fork()` é invocada uma vez (no processo-pai) mas retorna duas vezes, uma no processo que a invocou e outra num novo processo agora 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()* (3)


Processo antes de executar a chamada fork: procid = fork ()
Processo
fork
Processo-pai após a chamada fork. A chamada fork retorna a identificação (id) do processo-filho. Assim, o processo-pai vê procid como um valor inteiro positivo.
Pai
Processo-filho após a chamada fork. A chamada fork retorna um valor 0. Assim, o processo-filho vê procid = 0
Filho

SVCs para 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 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)`

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 */
}
```

Exemplo 1:

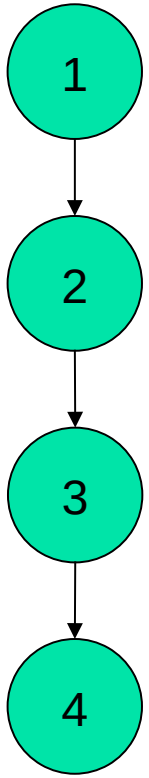
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
    pid_t childpid;
    childpid = fork();
    if (childpid == -1) {          /* error */
        perror("Failed to fork\n");
        return 1;
    }
    if (childpid == 0) {           /* child code */
        printf("I am child %d\n", getpid());
    } else {                      /* parent code */
        printf("I am parent %d\n", getpid());
    }
    return 0;
}
```

Exemplo 2:

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

int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");

    return 0;
}
```

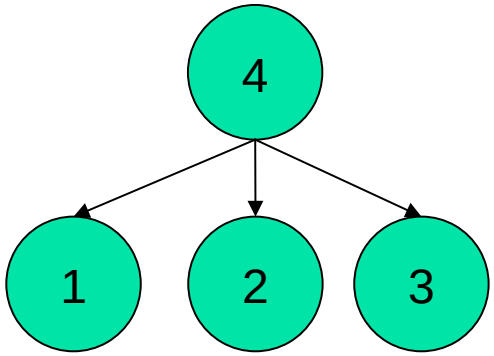
```
fork()
fork()
fork()
...
p
fork()                          // 1st fork()
p           c
fork()      fork()              // 2nd fork()
p     c     p     c
fork()fork()fork()fork()        // 3rd fork()
p  c  p  c  p  c  p  c
```

Exemplo 3:




```
#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) {
          printf("Usage: %s processes\n", argv[0]);
          return 1;
     }
     n = atoi(argv[1]);
     for (i = 1; i < n; i++) {
          childpid = fork();
          if (childpid > 0)   /* only parent enters */
               break;
     }
     fprintf(stderr, "i:%d  process ID: %d  parent ID: %d  child ID: %d\n",
          i, getpid(), getppid(), childpid);

     return 0;
}
```

Exemplo 4:




```
#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) {
          printf("Usage: %s processes\n", argv[0]);
          return 1;
     }
     n = atoi(argv[1]);

     for (i = 1; i < n; i++) {
          childpid = fork();
          if (childpid <= 0)  /* only the child (or error) enters */
               break;
     }

     fprintf(stderr, "i:%d  process ID: %d  parent ID: %d  child ID: %d\n",
          i, getpid(), getppid(), childpid);

     return 0;
}
```

A Chamada exit()

- `void exit(code)`
  - O argumento `code` é um número de 0 a 255, escolhido pela aplicação e que será passado para o processo pai na variável `status`.
- A chamada `exit()` termina o processo; portanto, `exit()` nunca retorna
  - Chama todos os *exit handlers* que foram registrados na função *atexit()*.
  - A memória alocada ao segmento físico de dados é liberada.
  - Todos os arquivos abertos são fechados.
  - É enviado um sinal para o pai do processo. Se este estiver bloqueado esperando o filho, ele é acordado.
  - Se o processo que invocou o `exit()` tiver filhos, esses serão "adotados" pelo processo `init`.
  - Faz o escalonador ser invocado.

Processo “Zombie”

- O que acontece se o processo filho termina antes do pai?
  - No Unix, o processo pai sempre tem que saber o *status* do término do processo filho. Por isso, sempre que um processo termina, o *kernel* guarda algumas informações sobre ele, de modo que essas informações estejam disponíveis para o processo pai quando ele executar `wait` ou `waitpid`.
  - Na terminologia do Unix, um processo que já terminou (já está morto) mas cujo pai ainda não executou o comando `wait` ou `waitpid` é dito um processo “*zombie*”.
- O que acontece se o processo pai termina antes do filho?
  - O processo *init* (PID 1) torna-se pai de todo e qualquer processo cujo pai termina antes do filho (isto é, o processo órfão é adotado pelo processo *init*).
  - Sempre que um filho do *init* termina, ele chama `wait/waitpid`.

A SVC *wait()* do Unix (1)

- Como processos descobrem se seu filho terminou?
  - Chamada `wait()` feita pelo pai – retorna o PID do processo filho que terminou execução
- Normalmente, pai executando `wait()` é bloqueado até filho terminar
  - Se não existirem filhos no estado zombie, esperar que um filho termine
- `waitpid()`
  - Para esperar um filho específico
  - Também pode esperar por qualquer filho
    - `waitpid(-1,...,...)` ~ `wait(...)`

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

A SVC *wait()* do Unix (2)

- Em caso de erro
  - retorna -1
  - Seta a variável global *errno*
    - ECHILD: não existem filhos para termitar (*wait*), ou pid não existe (*waitpid*)
    - EINTR: função foi interrompida por um sinal
    - EINVAL: o parâmetro *options* do *waitpid* estava inválido
- Solução para que um processo pai continue esperando pelo término de um processo filho, mesmo que o pai seja interrompido por um sinal:

```
pid_t r_wait(int *stat_loc)
{
     int retval;

     while (((retval = wait(stat_loc)) == -1) && (errno == EINTR));

     return retval;
}
```

Exemplo 5:

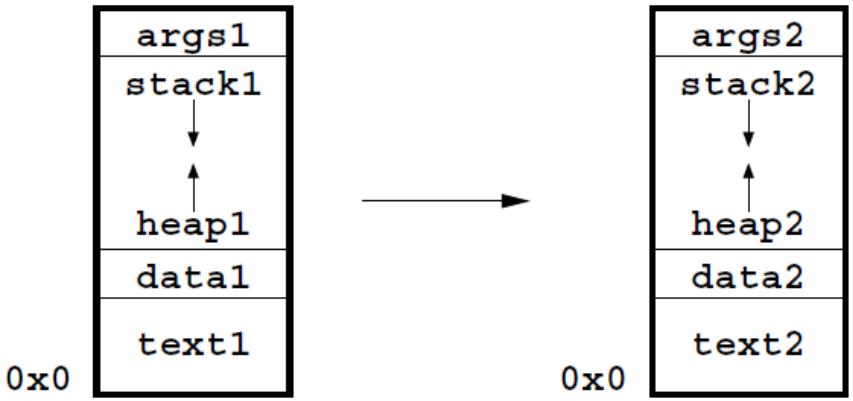
```
int main(int argc, char *argv[]){
     pid_t childpid;
     int i, n;

     if (argc != 2) {
          fprintf(stderr, "Usage: %s processes\n", argv[0]);
          return 1;
     }
     n = atoi(argv[1]);

     for (i = 1; i < n; i++) {
          childpid = fork();
          if (childpid <= 0)
               break;
     }

     while (r_wait(NULL) > 0);     /* wait for all of your children */
     fprintf(stderr, "i:%d proc ID: %d parent ID: %d  child ID: %d\n",
          i, getpid(), getppid(), childpid);

     return 0;
}
```

A SVC *exec()* do Unix (1)

- Quando um processo invoca uma das funções *exec*, ele é completamente substituído por um novo programa
  - Substitui o processo corrente (os seus segmentos text, data, heap e stack) por um novo programa carregado do disco
  - O novo programa começa a sua execução a partir da função main()
- O identificador do processo não é alterado
  - De fato, nenhum novo processo é criado
- Valor de retorno
  - **Sucesso - não retorna**
  - Erro - retorna o valor -1 e seta a variável errno com o código específico do erro
- Quando um processo executando um programa A quer executar outro programa B:
  - Primeiramente ele deve criar um novo processo usando fork()
  - Em seguida, o processo recém criado deve substituir todo o seu programa pelo programa B, chamando uma das primitivas da família *exec*

A SVC *exec()* do Unix (1)

```
#include <unistd.h>
int execve(const char* filename, char *const argv[],
           char *const envp[])
```

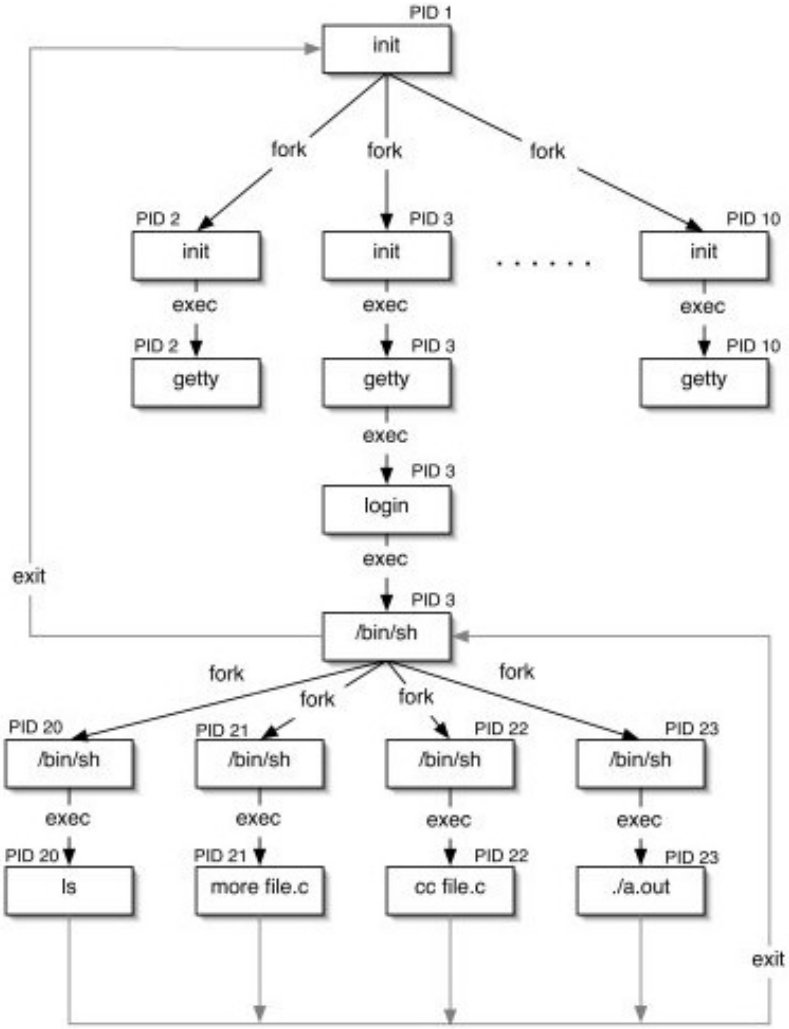


- Substitui o programa em execução pelo contido em `filename`;
- `argv` e `envp` permitem especificar os argumentos a passar à função `main()` do programa a executar.

Exemplo 6:

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

int main(int argc, char *argv[])
{
    int pid;
    /* fork a child process */
    pid = fork();
    /* check fork() return code */
    if (pid < 0) {                      /* some error occurred */
        fprintf(stderr, "Fork failed!\n" );
        exit( -1 );
    } else if (pid == 0) {              /* this is the child process */
        execl("/bin/ls", "ls", 0);      /* morph into “ls” */
    } else {
        wait(0);
        printf( "Child completed -- parent now exiting.\n" );
        exit(0);
    }
}
```

O shell do UNIX (3)



Processos *background* e *foreground* (1)

- Existem vários tipos de processos no Linux: processos interativos, processos em lote (*batch*) e *Daemons*. Processos interativos são iniciados a partir de uma sessão de terminal e por ele controlados. Quando executamos um comando do *shell*, entrando simplesmente o nome do programa seguido de <enter>, estamos rodando um processo em *foreground*.
- Um programa em *foreground* recebe diretamente sua entrada (*stdin*) do terminal que o controla e, por outro lado, toda a sua saida (*stdout* e *stderr*) vai para esse mesmo terminal. Digitando *Ctrl-Z,* suspendemos esse processo, e recebemos do *shell* a mensagem *Stopped* (talvez com mais alguns caracteres dizendo o número do *job* e a linha de comando).
- A maioria dos *shells* tem comandos para controle de *jobs*, para mudar o estado de um processo parado para *background*, listar os processos em *background*, retornar um processo de *back* para *foreground*, de modo que o possamos controlar novamente com o terminal. No *bash* o comando `jobs` mostra os *jobs* correntes, o `bg` restarta um processo suspenso em *background* e o comando `fg` o restarta em *foreground*.
- *Daemons* ou processos servidores, mais freqüentemente são iniciados na partida do sistema, rodando continuamente em *background* enquanto o sistema está no ar, e esperando até que algum outro processo solicite o seu serviço (ex: sendmail).

Processos de *background* e *foreground* (2)

- **O Comando jobs**
  - Serve para visualizar os processos que estão parados ou executando em segundo plano (*background*). Quando um processo está nessa condição, significa que a sua execução é feita pelo *kernel* sem que esteja vinculada a um terminal. Em outras palavras, um processo em segundo plano é aquele que é executado enquanto o usuário faz outra coisa no sistema.
  - Para executar um processo em *background* usa-se o “&” (ex: ls –l &). Assim, uma dica para saber se o processo está em *background* é verificar a existência do caractere & no final da linha. Se o processo estiver parado, geralmente a palavra "stopped" aparece na linha, do contrário, a palavra "running" é exibida.
- **Os comandos fg e bg**
  - O `fg` é um comando que permite a um processo em segundo plano (ou parado) passar para o primeiro plano (*foreground*), enquanto que o `bg` passa um processo do primeiro para o segundo plano. Para usar o `bg`, deve-se paralisar o processo. Isso pode ser feito pressionando-se as teclas Ctrl + Z. Em seguida, digita-se o comando da seguinte forma: `bg +número`
  - O número mencionado corresponde ao valor de ordem informado no início da linha quando o comando `jobs` é usado.
  - Quanto ao comando fg, a sintaxe é a mesma: `fg +número`

Resumo SVCs: Processos

- **fork()**: cria um novo processo que é uma cópia do processo pai. O processo criador e o processo filho continuam em paralelo, e executam a instrução seguinte à chamada de sistema.
- **wait()**: suspende a execução do processo corrente até que um filho termine. Se um filho terminou antes desta chamada de sistema(estado *zombie*), os recursos do filho são liberados e o processo não fica bloqueado, retornando imediatamente.
- **exit()**: termina o processo corrente. Os filhos, se existirem, são herdados pelo processo *init* e o processo pai é sinalizado.
- **exec()**: executa um programa, substituindo a imagem do processo corrente pela imagem de um novo processo, identificado pelo nome de um arquivo executável, passado como argumento.
- **kill()**: usada para enviar um sinal para um processo ou grupo de processos. O sinal pode indicar a morte do processo.
- **sleep()**: suspende o processo pelo tempo especificado como argumento.