Gerência de Processos

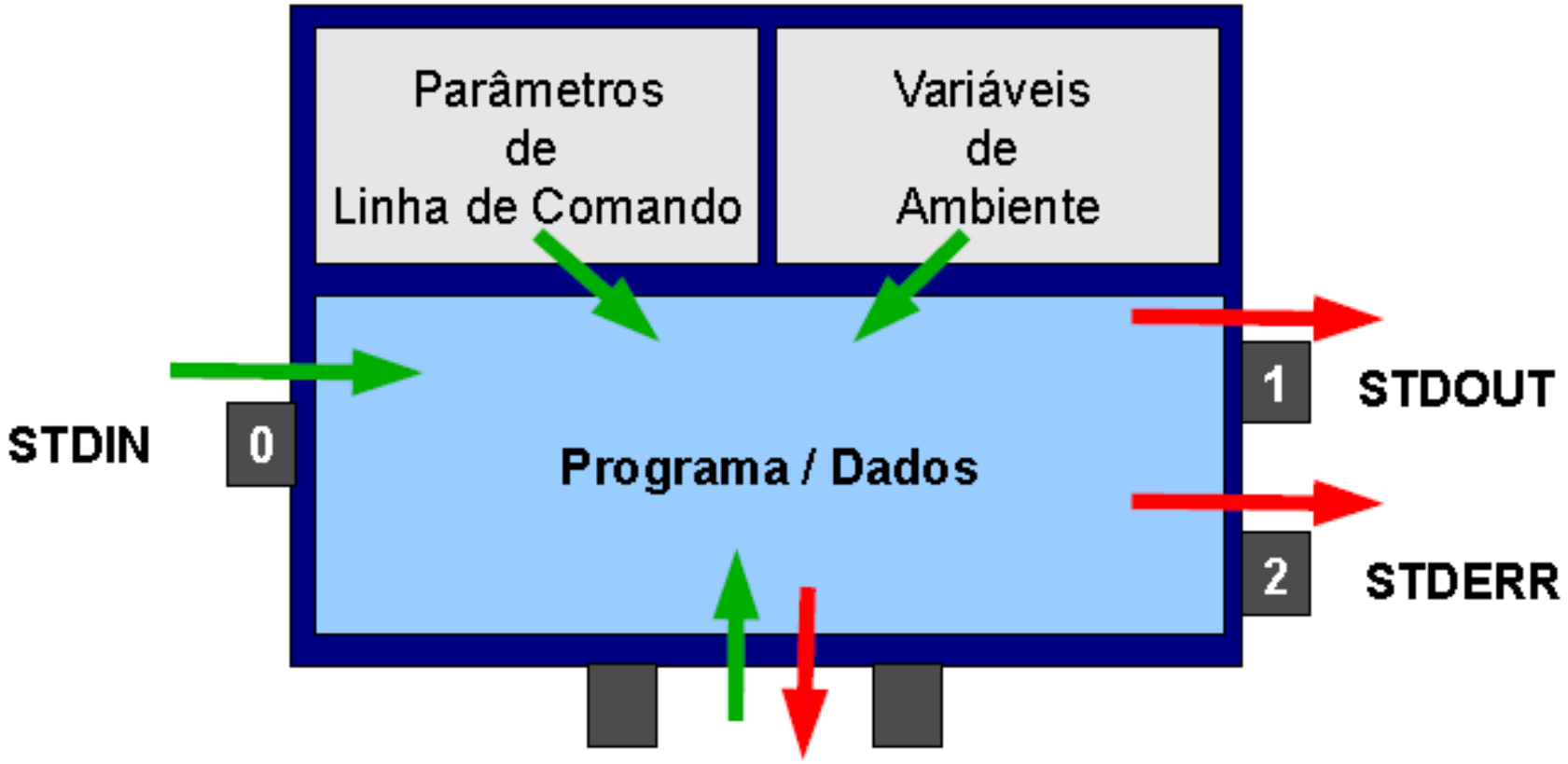
Laboratório de Sistemas Operacionais

Prof. MSc. João Tavares

Roteiro

- Processos em C
- Argumentos de Linha de Comando
- Disposição em Memória de um Programa C
- Identificadores e Credenciais de Processos
- Criação de Processos
- Funções fork, exit, wait e exec

Processo



Outros Canais de Comunicação (descritores)

Processo em C

- Um processo criado a partir de um programa em C começa executando pela função

  int **main**() { … }

- Antes do main(), uma função especial de inicialização é chamada

  - Obtém parâmetros do kernel e realiza configurações

- O processo pode terminar de forma

  - **normal** → permite que ações de limpeza sejam executadas antes da terminação

  - **anormal** → não executa ações de limpeza cadastradas pelo programa

Processo em C

- Causas para a **Terminação Normal**
  - Chamada a exit() em qualquer ponto do programa
  - Retorno de main()
    * "return *n*" na função main() equivale a "exit(*n*)"
  - Retorno da função principal (*entrypoint*) da última thread
  - Chamada a pthread_exit() na última thread
  - Chamada a _exit() ou _Exit() → limpeza reduzida

- A função atexit() permite registrar tratadores (callbacks) de terminação

Processo em C

- Causas para a **Terminação Anormal** → usualmente tem causas assíncronas
  - Recepção de sinal gerado pelo SO ou outro processo
  - Chamada a abort()
    * De fato, envia um sinal para o próprio processo
  - Resposta da última thread a uma requisição de cancelamento
- Terminação anormal pode ter efeitos colaterais
  - Os recursos alocados ao processo são liberados pelo SO, porém podem ficar inconsistentes pois as ações de limpeza não são executadas

Argumentos de linha de comando

- São recebidos como parâmetros da função main()
  Ex.: *echoarg.c*

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        int i;
        for (i = 0; i < argc; i++)
                printf("argv[%d]: %s\n", i, argv[i]);
        exit(0);
}
```

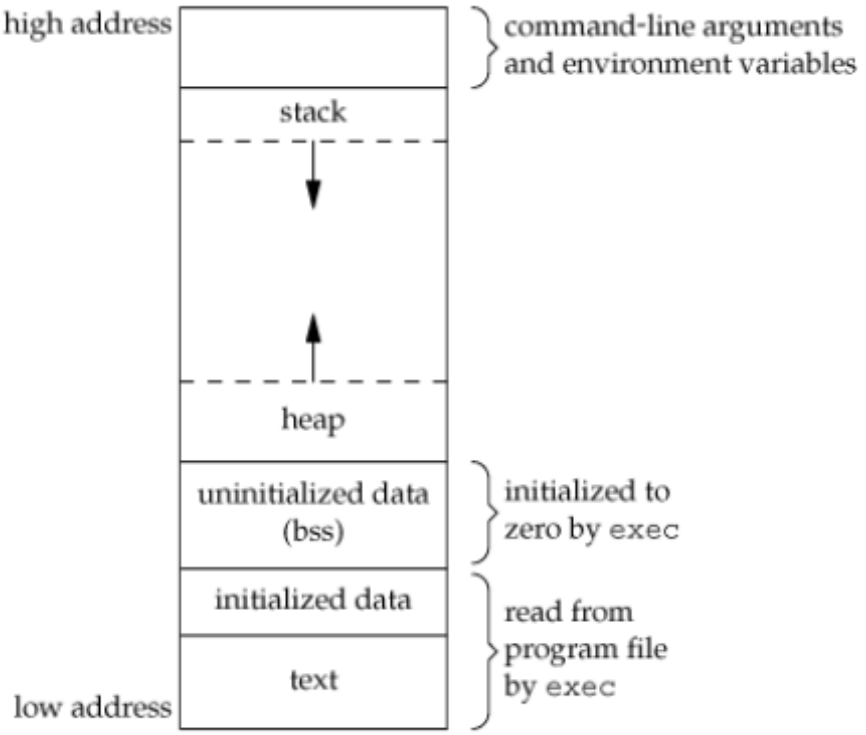
Saída:

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

Disposição em memória

- Espaço segmentado

- Considera uso e características da informação
  - leitura / escrita
  - código / dados
    - * inicializados ou não
    - * dinâmicos ou estáticos
    - * locais ou globais



Disposição em memória

- Segmento de texto
  - Instruções de máquina que o programa executa
  - Usualmente compartilhado e somente-leitura

- Segmento de dados inicializados
  - Variáveis globais inicializadas

- Segmento de dados não inicializado
  - bss (*block started by symbol*)
  - Variáveis globais não inicializadas

Disposição em memória

- Pilha (*stack*)
  - armazena variáveis automáticas e informações salvas na chamada de funções.

- Heap
  - armazenamento de variáveis alocadas dinamicamente.

- **Nota:** Formato de Armazenamento de Programas
  - Somente são salvos no disco o segmento de texto e os dados de inicialização.
  - Pilha, Heap e dados não inicializados não ocupam espaço no arquivo executável (são apenas anotados).

Formato de armazenamento

- Nem todas as estruturas que compõem a imagem do processo em memória são armazenadas no arquivo do programa
  - São salvos no disco:
    * segmento de texto
    * dados de inicialização (inclui símbolos para o *bss*)

  - Não ocupam espaço (são apenas anotados)
    * Pilha
    * Heap
    * Dados não inicializados

Formato de armazenamento

- O comando size no Unix informa o tamanho dos segmentos de texto, dados e bss em bytes

Ex.:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex filename
  79606    1536     916   82058   1408a /usr/bin/cc
 619234   21120   18260  658614   a0cb6 /bin/sh
```

Identificadores de Processos

- Todo processo tem um identificador único no sistema chamado Process ID ou PID
  - número inteiro não negativo
  - valor único entre todos os processos atualmente em execução

- PIDs são reutilizáveis
  - quando o processo termina, seu PID é candidato a reuso
  - atraso no reuso de PID é imposto por vários Unix como estratégia para aumentar a segurança e a consistência no sistema

Identificadores de Processos

- PID 0 → normalmente é o escalonador (process swapper)

- PID 1 → processo init
  - Chamado pelo kernel ao final do bootstrap do kernel para completar a inicialização do sistema
  - Processo normal de usuário, porém nunca morre

- PID 2 → kthreadd (Linux Kernel 2.6.x)
  - Gerencia as threads utilizadas pelos serviços implementados dentro do kernel

Gestão de processos

- Listagem de processos ativos
  ps aux
- Listagem de processos com uso de CPU
  top
- “Matar” processos
  kill –SIGTERM PID ou kill –9 PID

Identificadores de Processos: API

- Funções definidas no header unistd.h

  - pid_t **getpid**(void);
  Retorna o PID do processo chamador

  - pid_t **getppid**(void);
  Retorna o PID do pai (PPID) do processo chamador

Credenciais de Processo

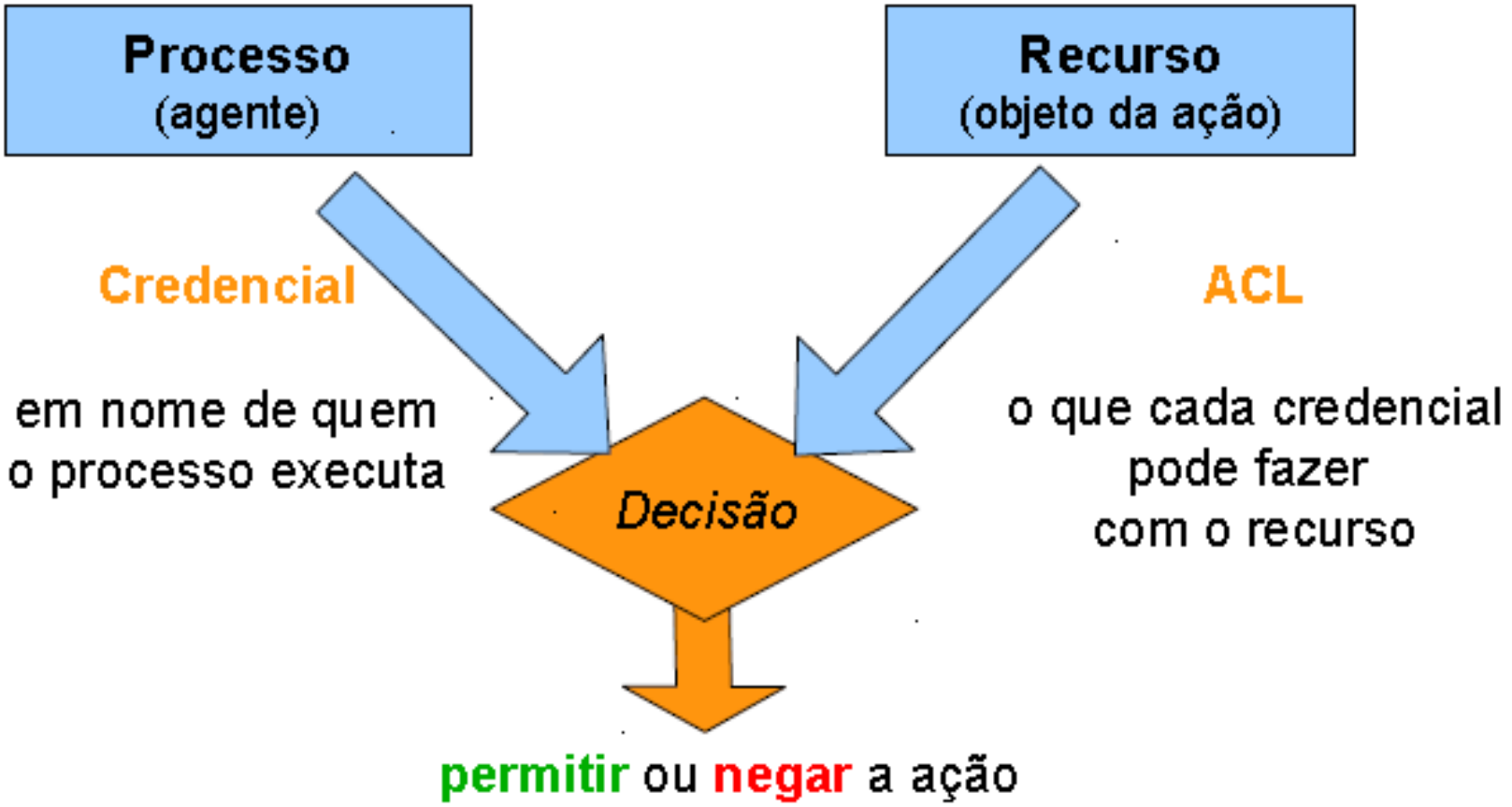
- Influenciam o que um processo pode fazer ou não no sistema

- Composta de informação de usuário e grupo

- Dois tipos principais
  - Credenciais Reais → quem é o dono do processo
  - Credenciais Efetivas → utilizadas para determinar as permissões do processo a recursos compartilhados

- O Linux pode usar informações adicionais: filesystem ID, grupos suplementares, e *capabilities*
  - *man credentials*
  - *man capabilities*

Credenciais de Processo



Identificadores de Processos: API

- Operações de consulta → podem ser realizadas por qualquer processo

  - uid_t **getuid**(void);
  Retorna o real user ID do processo chamador

  - uid_t **geteuid**(void);
  Retorna o effective user ID do processo chamador

  - gid_t **getgid**(void);
  Retorna o real group ID do processo chamador

  - gid_t **getegid**(void);
  Retorna o effective group ID do processo chamador

Identificadores de Processos: API

- Operações de modificação → apenas processos privilegiados podem modificar suas credenciais

  - int **setuid** (uid_t id)
  Configura o ID do usuário do processo chamador

  - int **seteuid** (uid_t id)
  Configura o ID efetivo do usuário do processo chamador

  - int **setgid** (gid_t id)
  Configura o ID do grupo do processo chamador

  - int **setegid** (gid_t id)
  Configura o ID efetivo do grupo do processo chamador

Criação de Processos

- Para criar processo no Unix, duplica-se um processo existente

- Após a duplicação, os processos pai e filho são praticamente iguais
  - códigos, dados e pilha são cópia do pai
  - continuam a executar o mesmo código
  - pid, ppid e tempos de execução diferem

- Processo filho pode substituir o seu código por outro programa executável se desejar
  - Substituição não cria outro processo!

Criação de Processos

- Quando o filho termina, o pai é avisado de seu encerramento para que possa realizar alguma ação

- Em geral, pai e filho executam concorrentemente após a duplicação
  - Processo pai pode, se desejar, solicitar ao SO para permanecer suspenso até que o filho termine
  * Ex.: shell quando está executando programas em primeiro plano

Criação de Processos: API

- Principais chamadas de sistema:

  fork() → duplica um processo

  exit() → termina um processo

  wait() → espera pelo término de um filho

  exec() → substitui código, dados e pilhas de um processo

A função fork()

- Cria um novo processo

  pid_t **fork**(void)

- Chamada uma vez, retorna duas vezes
  - no filho → retorna 0 (zero),
  - no pai → retorna o PID do filho (-1 em caso de erro)

- Gera a hierarquia de processos
  - Não existe chamada para um pai saber o PID de seus filhos
  - Processos podem ter vários filhos; cada processo só tem um pai
  * init é o antepassado raiz de todos processos

Exemplo usando fork()

- Ex.: processo.*c*

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

int glob = 6;                                   /* variavel no seg. dados inicializado */

int main(){
        int var=88;                             /* variavel automatica na pilha */
        pid_t pid;
        printf("antes do fork\n");
        if ((pid=fork()) < 0) {
                perror("erro no fork!");        /* trat. de erro */
        }
        else if (pid == 0) {
                glob++; var++;                  /* filho modifica variaveis */
        }
        else sleep(2);                          /* pai dorme 2 segundos */

        printf("pid=%d, glob=%d, var=%d\n",getpid(), glob, var);
        exit(0);
}
```

Exemplo usando fork()

- Saída:

```
$ ./fork1
antes do fork
pid = 3776, glob = 7, var = 89
pid = 3775, glob = 6, var = 88
```

- Pai e filho executam a instrução que segue o fork;
  - ordem de execução depende do SO
  - filho é uma cópia do pai (dados, heap e pilha)

- Otimizações da implementação do fork()
  - pai e filho compartilham segmento de texto
  - uso de copy-on-write (COW) para segmento/páginas de dados

A função exit()

- Termina um processo
  void **exit**(int status)

- Executa disparo da execução das ações de limpeza
  - Programadas com atexit()

- Fecha arquivos abertos pelo processo, desaloca código, dados e pilha e termina

- Quando termina envia um sinal SIGCHLD e espera que o pai aceite
  - permanece em estado chamado zumbi até que o pai trate a notificação de terminação usando wait()
  - Se pai morre, filhos orfãos são adotados pelo processo init

Exemplo usando exit()

- Ex.: processoorfao.*c*

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

int main(){
        int pid;
        printf("Antes do fork\n");
        pid = fork();                   /* Criar processo filho */
        printf("Depois do fork\n");

        switch(pid) {
        case -1:
                perror("Erro no fork\n");
                exit(-1);
                break;
        case 0:
                printf("Sou o filho, estou entrando em loop... Meu PID eh: i \n", getpid());
                while(1){
                        sleep(100);     /* Filho: nunca termina */
                }
                break;
        default:
                printf("Sou o pai e vou finalizar com erro 42\n");
                exit(42);               /* Pai: Termina com status qualquer */
        }
        return 0;                       /* nunca será executado */
}
```

Exemplo usando exit()

- Ex.: processozumbi.c

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

int main(){
        int pid;
        printf("Antes do fork\n");
        pid = fork();                                   /* Criar processo filho */
        printf("Depois do fork\n");

        switch(pid) {
        case -1:
                perror("Erro no fork\n");
                exit(-1);
                break;
        case 0:
                printf("Sou o filho, e vou morrer...\n");       /* Filho: Termina com status qualquer */
                exit(43);
                break;
        default:
                printf("Sou o pai e vou entrar em loop...\n");
                while(1){
                        sleep(100);                     /* Pai: nunca termina */
                }
        }
        return 0;
}
```

A função wait ()

- Espera por um processo filho

  pid_t **wait**(int *status)

- O processo chamador é suspenso até que um de seus filhos termine
  - A função retorna o pid do filho que termina
    * wait() prioriza processos zumbis, se existem
    * Se não tem filhos, retorna imediatamente -1

- status combina diversas informações
  - Necessário usar macros para extrair de *status* a informação particular de interesse
    * Ex.: WEXITSTATUS, WIFEXITED, WIFSIGNALED, WTERMSIG,...)

Exemplo usando wait()

- Ex.: processowait.*c*

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

int main(){
        pid_t pid;
        int status;
        printf("Pai: Meu PID = %d \n",getpid());

        switch(fork()) {
        case -1:
                perror("Erro no fork\n");
                exit(-1);
                break;
        case 0:
                printf("Filho: Meu PID = %d, PPID = %d \n",getpid(), getppid());        /* Filho: id proprio e id do pai */
                exit(13);
                break;
        default:
                printf("Pai: PID = %d, PPID = %d \n", getpid(), getppid());
                pid = wait(&status);                                                    /* Espera filho terminar.*/
                printf("Pai: Filho (PID = %d) terminou, Status = %d \n",pid,WEXITSTATUS(status));
        }
        printf("PID %d terminando... \n", getpid());
        return 0;
}
```

As funções exec()

- Substitui o código do executável do processo chamador por um novo programa
  - exec substitui segmentos de texto, dados, heap e pilha por novo programa lido do disco;
  - caso não encontre retorna -1

- Novo programa começa a executar a partir de seu main()

- PID não muda → não cria outro processo!

As funções exec()

- Existem 6 funções C diferentes no header unistd.h para acessar a chamada de sistema exec

- Escolha leva em consideração a facilidade:
  - da passagem de parâmetros
  - de localização do novo programa executável
  - de modificação das variáveis de ambiente

As funções exec()

- exec**v*** → recebem argumentos a serem passados ao novo programa em um vetor (array)
  - Último elemento do array deve ser o ponteiro nulo!

```
int execv(const char *pathname, char *const argv []);

int execvp(const char *filename, char *const argv []);

int execve(const char *pathname, char *const argv [],
   char *const envp []);
```

  - **p** → procura programa no path
  - **e** → substitui variáveis de ambiente

As funções exec()

- exec**l*** → recebem a lista expandida de argumentos que serão passados ao novo programa
  - Último elemento do lista deve ser o ponteiro nulo!

```
int execl (const char *pathname, const char *arg0, ... /*
  NULL */ );

int execlp(const char *filename, const char *arg0, ... /* NULL
  */ );

int execle(const char *pathname, const char *arg0, ... /*
  NULL */, char *const envp[]);
```

  - **p** → procura programa no path
  - **e** → substitui variáveis de ambiente

Exemplo usando execl()

- Ex.: processoexec*.c*

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

int main(){
        pid_t p = fork();                                       /* cria novo processo */
        if (p < 0) {                                            /* ocorrencia de erro */
                fprintf(stderr, "Criação de novo processo falhou! \n");
                exit(-1);                                       /* terminacao de erro */
        } else if (p == 0) {                                    /* processo filho */
                printf("Iniciando a execução do filho... \n");
                execlp("/bin/ls", "ls", NULL);                  /* atribui novo programa ao filho */
                printf("Filho com parada de execução forçada! \n");
                exit(-1);
        } else {                                                /* processo pai */
                printf("Pai aguardando o filho terminar... \n");
                wait(NULL);                                     /* pai vai aguardar até o filho completar */
                printf("Filho completou a execução! \n");
                exit(0);                                        /* terminacao de sucesso */
        }
}
```

Leituras complementares

- STEVENS, W.R. Advanced Programming in the UNIX Environment. 2nd. Ed., Addison Wesley, 2005.

- Man pages das funções abordadas

- Página info da libc, em especial as seções
  - Program Basics
  - Processes

- Livro: Advanced Linux Programming
  Disponível para download em:
  http://www.advancedlinuxprogramming.com/alp-folder

Referências Bibliográficas

- Material originalmente elaborado por Prof. Cristiano Costa. Material autorizado e cedido pelo autor. Revisado e atualizado por Prof. Luciano Cavalheiro e posteriormente pelo Prof. João Tavares.