

# Sistemas Operacionais – Atividade 1.2

## Manual de Funções de Processos em C

David Osvaldo Caldas Pereira<sup>1</sup>, Tales Lima de Oliveira<sup>1</sup>

<sup>1</sup>Instituto Federal de Brasília (IFB)  
Taguatinga – DF – Brasil

{david.pereira3, tales.oliveira}@estudante.ifb.edu.br

### 1. Introdução

Este manual faz parte de um projeto desenvolvido para a disciplina de Sistemas Operacionais (2024/2), ministrada pelo professor João Oliveira.

O objetivo deste projeto é apresentar de forma prática a criação e o controle de processos na linguagem C, utilizando as funções: `getpid`, `getppid`, `fork`, `wait`, `waitpid`, `execv` e `execve`. Essas funções facilitam o desenvolvimento de programas que exigem controle detalhado sobre a execução de subprocessos.

Todos os códigos gerados para este projeto acompanham este manual, facilitando e permitindo a execução dos exemplos descritos. Para acesso completo ao código-fonte e contribuições, é possível acessar o repositório no GitHub: [Repositório do Projeto](#).

### 2. Funções de Processos em C

#### 2.1. Funções `getpid()` e `getppid()`

As funções `getpid()` e `getppid()` pertencem à biblioteca `unistd.h` e são utilizadas para obter os identificadores de processo (PID). Essas informações são especialmente úteis em programas que gerenciam processos ou que precisam manter uma relação de hierarquia entre eles [IncludeHelp-PID] [LinuxManual-PID].

- A função **`getpid()`** retorna o o identificador do processo atual.
- A função **`getppid()`** retorna o o identificador do processo pai.

**Nota:** O programador pode utilizar um tipo de dado específico para identificadores de processos, `pid_t`, disponível na biblioteca `sys/types.h` [IncludeHelp-PID].

Abaixo, um exemplo que demonstra o uso das funções `getpid()` e `getppid()`:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4
5  int main(void) {
6      pid_t process_id = getpid();
7      pid_t p_process_id = getppid();
8
9      printf("ID do processo atual (PID): %d\n", process_id);
10     printf("ID do processo pai (PPID): %d\n", p_process_id);
11     return 0;
12 }
```

## 2.2. Função `fork()`

A função `fork()` pertence à biblioteca `unistd.h` e é usada para criar um novo processo filho que é uma cópia exata do processo pai no momento da chamada. Esse processo filho herda o mesmo espaço de memória, variáveis e contexto, possibilitando a execução de operações paralelas entre o processo pai e o processo filho [IncludeHelp-Fork] [LinuxManual-Fork].

- No *processo pai*, a função **`fork()`** retorna o PID do *processo filho*.
- No *processo filho*, a função **`fork()`** retorna o valor 0.
- Em caso de erro na criação do processo, a função retorna -1.

Abaixo, um exemplo que demonstra o uso da função `fork()`:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main(void) {
6      pid_t pid = fork();
7
8      if (pid < 0) {
9          perror("Fork falhou");
10         return 1;
11     }
12
13     // Processo Filho
14     if (pid == 0) {
15         printf("Filho com PID: %d e ", getpid());
16         printf("Pai com PID: %d\n", getppid());
17     }
18     // Processo Pai
19     else {
20         printf("Pai com PID: %d ", getpid());
21         printf("e Filho com PID: %d\n", pid);
22     }
23
24     return 0;
25 }
```

## 2.3. Funções `wait()` e `waitpid()`

As funções `wait()` e `waitpid()` permitem que um processo pai aguarde o término de um processo filho, proporcionando sincronização entre processos. Isso é crucial para evitar condições de corrida e garantir que o processo pai obtenha o resultado final do processo filho antes de continuar sua execução [TutorialsPoint-Wait].

- Essas funções retornam o PID do *filho* que terminou.
- Em caso de erro, retornam -1.

A função **wait()** faz com que o *processo pai* aguarde a finalização de um processo *filho*, e recebe o argumento *\*status*. A função **waitpid()** permiti especificar qual *processo filho* aguardar, e recebe os argumentos: *pid*, *\*status*, *options* [LinuxManual-Wait ] [TutorialsPoint-Wait ].

- **pid\_t pid**: O ID do processo que você deseja esperar. Pode ser:
  - *Um valor positivo*: espera pelo processo com o PID especificado.
  - *Zero*: espera por qualquer processo filho no mesmo grupo de processos.
  - *-1*: espera por qualquer processo filho.
  - *Menor que -1*: espera por qualquer processo filho cujo grupo de processos é igual ao valor absoluto de *pid*.
- **int \*status**: Um ponteiro para uma variável onde o status de término do processo será armazenado. Este status pode ser analisado usando macros como *WIFEXITED*, *WEXITSTATUS*, *WIFSIGNALED*, entre outras.
- **int options**: Um conjunto de opções que modifica o comportamento da função. Pode ser 0 ou uma combinação das seguintes opções:
  - *WNOHANG*: Retorna imediatamente se nenhum filho terminou.
  - *WUNTRACED*: Retorna se um filho parou, mas não foi rastreado.
  - *WCONTINUED*: Retorna se um filho que estava parado foi continuado.

Abaixo, exemplo da função `wait()`:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      pid_t pid = fork();
8
9      if (pid < 0) {
10         printf("Fork falhou");
11         return 1;
12     }
13
14     // Processo Filho
15     if (pid == 0) {
16         sleep(2); // Simula Trabalho
17         exit(0); // Termina o filho para nao haver duplicatas
18     }
19
20     // Processo Pai
21     int status;
22     pid_t waited_pid = wait(&status);
23     printf("Filho com PID %d terminou.\n", waited_pid);
24
25     return 0;
26 }
```

Abaixo, exemplo da função `waitpid()`:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      pid_t pid = fork();
8
9      if (pid < 0) {
10         printf("Fork falhou");
11         return 1;
12     }
13
14     // Processo Filho
15     if (pid == 0) {
16         sleep(2); // Simula Trabalho
17         exit(0); // Termina o filho para nao haver duplicatas
18     }
19
20     // Processo Pai
21     int status;
22     // Espera pelo processo filho específico
23     pid_t waited_pid = waitpid(pid, &status, 0);
24
25     if (waited_pid < 0) {
26         printf("Erro ao esperar pelo filho.\n");
27         return 1;
28     }
29
30     if (!WIFEXITED(status)) {
31         printf("Filho com PID %d terminou anormal.\n", waited_pid);
32         return 1;
33     }
34
35     printf("Filho com PID %d terminou com status %d.\n", waited_pid,
36           ↪ WEXITSTATUS(status));
37     return 0;
38 }
```

## 2.4. Funções `execv()` e `execve()`

As funções `execv()` e `execve()` substituem o programa em execução em um processo por outro programa, permitindo a execução de novos comandos ou aplicações. Essas funções são essenciais quando um processo precisa alterar seu comportamento para executar tarefas diferentes ou lançar outras aplicações [LinuxManual-Exec].

- Em caso de sucesso, essas funções não retornam ao processo chamador, pois ele é substituído pelo novo processo.
- Em caso de erro, retornam -1.

A função **execv ()** substitui o processo atual por um novo programa. Ela recebe dois argumentos:

- **path**: o caminho para o executável que será executado.
- **argv**: um vetor de argumentos, onde o primeiro elemento é o próprio nome do programa e o último é `NULL`.

A função **execve ()** é semelhante a `execv ()`, mas permite especificar um vetor adicional de **variáveis de ambiente**:

- **path**: o caminho para o executável.
- **argv**: um vetor de argumentos.
- **envp**: um vetor de variáveis de ambiente no formato `KEY=VALUE`, onde o último elemento é `NULL`.

Abaixo, um exemplo da função `execv ()`:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(void) {
5      // Comando para listar arquivos (ls -l)
6      char *args[] = {"/bin/ls", "-l", NULL};
7
8      // Substitui o processo pelo comando 'ls -l'
9      execv(args[0], args);
10
11     // Em caso de sucesso o programa não executará essa linha
12     // Mas em caso de erro...
13     printf("execv falhou");
14     return 1;
15 }
```

### 3. Compilação e Execução do Código

Este projeto utiliza um `Makefile` para simplificar o processo de compilação e execução dos programas. Instruções detalhadas podem ser encontradas no arquivo **README.md** do projeto.

- Para **compilar** todos os programas, utilize o comando:
  - `make all`
- Para **executar** os programas, use um dos seguintes comandos:
  - `make run-main`
  - ou
  - `make run-exemplos`

## **Referências**

- [IncludeHelp-Fork ] IncludeHelp-Fork. C - fork() function: Explained with examples. [Online; accessed November-2024].
- [IncludeHelp-PID ] IncludeHelp-PID. getpid() and getppid() functions in c linux. [Online; accessed November-2024].
- [LinuxManual-Exec ] LinuxManual-Exec. exec(3) — linux manual page. [Online; accessed November-2024].
- [LinuxManual-Fork ] LinuxManual-Fork. fork(2) — linux manual page. [Online; accessed November-2024].
- [LinuxManual-PID ] LinuxManual-PID. getpid(2) — linux manual page. [Online; accessed November-2024].
- [LinuxManual-Wait ] LinuxManual-Wait. waitpid(3p) — linux manual page. [Online; accessed November-2024].
- [TutorialsPoint-Wait ] TutorialsPoint-Wait. waitpid() - unix, linux system call. [Online; accessed November-2024].