

Sistemas Operacionais – Atividade 1.2

Manual de Funções de Processos em C

David Osvaldo Caldas Pereira¹, Tales Lima de Oliveira¹

¹Instituto 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 é demonstrar a criação e o controle de processos utilizando funções da linguagem C, como: **getpid**, **getppid**, **fork**, **execv**, **execve**, **wait** e **waitpid**.

Todos os códigos produzidos para este projeto estão incluídos junto com este manual, permitindo a execução dos exemplos. Além disso, é possível acessar o repositório no GitHub, onde o código está disponível: [Repositório do Projeto](#).

2. Funções de Processos em C

2.1. Getpid e Getppid

Essas funções retornam o identificador de processo (PID) do processo atual e o PID do processo pai, respectivamente. O uso dessas informações é fundamental para o rastreamento e controle de processos específicos dentro do sistema operacional.

getpid() retorna o PID do *processo chamador*.

getppid() retorna o PID do *processo pai*.

Exemplo das funções **getpid()** e **getppid()** abaixo:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(void) {
5      printf("Processo atual ID (PID): %d\n", getpid());
6      printf("Processo Pai ID (PPID): %d\n", getppid());
7      return 0;
8  }
```

2.2. Fork

Esta função permite a criação de um novo processo filho, que é uma cópia exata do processo pai no momento da chamada. O processo filho herda o mesmo espaço de memória, variáveis e contexto, o que possibilita operações paralelas.

No *processo pai*, **fork()** retorna o PID (Process ID) do *filho*.

No *processo filho*, retorna 0.

Se ocorrer um erro, retorna -1.

Exemplo da função **fork()** abaixo:

```
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
19     //Processo Pai
20     else {
21         printf("Pai com PID: %d ", getpid());
22         printf("e Filho com PID: %d\n", pid);
23     }
24
25     return 0;
26 }
```

2.3. Wait e Waitpid

Estas funções 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.

A função **wait()** faz com que o processo **pai** aguarde a finalização de um processo *filho*.

A função **waitpid()** é uma versão mais flexível, permitindo especificar qual processo *filho* aguardar.

Retorna o PID do *filho* que terminou ou -1 em caso de erro.

Exemplo das funções **wait()** e **waitpid()** abaixo:

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

```

3  #include <sys/wait.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         sleep(2); // Simula Trabalho
16         return 0; // Fim do Filho
17     }
18
19     //Processo Pai
20     int status;
21     pid_t waited_pid = wait(&status);
22     printf("Filho com PID %d terminou.\n", waited_pid);
23
24     return 0;
25 }

```

2.4. Execv e Execve

Essas funções substituem o programa em execução em um processo por outro programa, permitindo a execução de novos comandos ou aplicações. Isso é essencial quando um processo precisa alterar seu comportamento para executar tarefas diferentes ou lançar outras aplicações.

A função **execv()** substitui o processo atual por um novo processo. Recebe como argumentos o caminho do executável e um vetor de argumentos.

A função **execve()** é semelhante, mas aceita um vetor de **variáveis de ambiente**.

Em caso de sucesso, não retornam. Em caso de erro, retornam -1.

Exemplo das funções **execv()** e **execve()** abaixo:

```

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 executara 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 a compilação e execução.

- Para **compilar** os programas, utilize:
 - make all
- Para **executar** os programas, utilize:
 - make run
 - ou
 - ./build/bin/NOME-DO-PROGRAMA