Sistemas Operacionais – Atividade 1.2
Manual de Funções de Processos em C

David Osvaldo Caldas Pereira[1], **Tales Lima de Oliveira**[1]

[1]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 Victor de Araujo 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 b] [LinuxManual c].

- 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 b].

Abaixo, um exemplo do uso das funções `getpid()` e `getppid()`:

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

int main(void){
    pid_t process_id = getpid();
    pid_t p_process_id = getppid();

    printf("ID do processo atual (PID): %d\n", process_id);
    printf("ID do processo pai (PPID): %d\n", p_process_id);
    return 0;
}
```

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 a] [LinuxManual b].

- 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()`:

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

int main(void){
    pid_t pid = fork();

    if (pid < 0){
        perror("Fork falhou");
        return 1;
    }

    //Processo Filho
    if (pid == 0){
        printf("Filho com PID: %d e ", getpid());
        printf("Pai com PID: %d\n", getppid());
    }
    //Processo Pai
    else{
        printf("Pai com PID: %d ", getpid());
        printf("e Filho com PID: %d\n", pid);
    }

    return 0;
}
```

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].

- 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 qualquer processo filho no mesmo grupo de processos, e recebe o argumento `*status`.

A função **`waitpid()`** é mais flexivel e permiti especificar qual *processo filho* aguardar, e recebe os argumentos: `pid`, `*status`, `options` [LinuxManual d] [TutorialsPoint].

- **`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()`:

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

int main(void){
    pid_t pid = fork();

    if (pid < 0){
        printf("Fork falhou");
        return 1;
    }

    //Processo Filho
    if (pid == 0){
        sleep(2); // Simula Trabalho
        exit(0); // Termina o filho para nao haver duplicatas
    }

    //Processo Pai
    int status;
    pid_t waited_pid = wait(&status);
    printf("Filho com PID %d terminou.\n", waited_pid);

    return 0;
}
```

Abaixo, exemplo da função `waitpid()`:

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

int main(void){
    pid_t pid = fork();

    if (pid < 0){
        printf("Fork falhou");
        return 1;
    }

    //Processo Filho
    if (pid == 0){
        sleep(2); // Simula Trabalho
        exit(0); // Termina o filho para nao haver duplicatas
    }

    //Processo Pai
    int status;
    // Espera pelo processo filho específico
    pid_t waited_pid = waitpid(pid, &status, 0);

    if (waited_pid < 0){
        printf("Erro ao esperar pelo filho.\n");
        return 1;
    }

    if (!WIFEXITED(status)){
        printf("Filho com PID %d terminou anormal.\n", waited_pid);
        return 1;
    }

    printf("Filho com PID %d terminou com status %d.\n", waited_pid,
    ↪  WEXITSTATUS(status));
    return 0;
}
```

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 a].

- 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()`:

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

int main(void) {
    // Comando para listar arquivos (ls -l)
    char *args[] = {"/bin/ls", "-l", NULL};

    // Substitui o processo pelo comando 'ls -l'
    execv(args[0], args);

    // Em caso de sucesso o programa não executará essa linha
    // Mas em caso de erro...
    printf("execv falhou");
    return 1;
}
```

3. Compilação e Execução do Código

Este projeto utiliza um `Makefile` para simplificar o processo de compilação, execução e limpeza 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 utilizados como exemplo, utilize o comando:
  - `make run`
- Para a **limpeza** dos arquivos binarios, utilize o comando:
  - `make clean`

Referências

[IncludeHelp a] IncludeHelp. C - fork() function: Explained with examples. https://www.includehelp.com/c-programs/c-fork-function-linux-example.aspx [Online; accessed November-2024].

[IncludeHelp b] IncludeHelp. getpid() and getppid() functions in c linux. https://www.includehelp.com/c/getpid-and-getppid-functions-in-c-linux.aspx [Online; accessed November-2024].

[LinuxManual a] LinuxManual. exec(3) — linux manual page. arroz. https://man7.org/linux/man-pages/man3/exec.3.html [Online; accessed November-2024].

[LinuxManual b] LinuxManual. fork(2) — linux manual page. https://man7.org/linux/man-pages/man2/fork.2.html [Online; accessed November-2024].

[LinuxManual c] LinuxManual. getpid(2) — linux manual page. https://man7.org/linux/man-pages/man2/getpid.2.html [Online; accessed November-2024].

[LinuxManual d] LinuxManual. waitpid(3p) — linux manual page. https://man7.org/linux/man-pages/man3/waitpid.3p.html [Online; accessed November-2024].

[TutorialsPoint] TutorialsPoint. waitpid() - unix, linux system call. https://www.tutorialspoint.com/unix_system_calls/waitpid.htm [Online; accessed November-2024].