Curso Tecnologia em ADS
Sistemas Operacionais

# **Laboratório :** Criação de Processos no Linux

Professor: Eraldo Silveira e Silva

## 1 Objetivo do Experimento

Este experimento tem por objetivo:

- Explorar a criação de processos com a chamada de sistema `fork()`;
- Compreender como pai e filho se relacionam e como variáveis, arquivos e status são tratados;
- Explorar comandos no Linux para visualizar e analisar processos (`ps`, `pstree`, `htop`, `lsof`).

## Instruções Gerais

Antes de iniciar os exercícios, siga estas orientações:

1. Registre no terminal o seu nome com:

```
1 echo "Exercícios realizados pelo aluno NOME_COMPLETO"
2
```

2. Antes de executar cada exercício, registre também:

```
3 echo "Exercicio X"
4
```

   (substitua X pelo número correspondente). Isso facilitará a identificação no histórico de comandos.

3. Ao final do laboratório, gere o histórico dos comandos digitados com:

```
5 history > meus_comandos.txt
6
```

4. Envie o arquivo pelo sistema acadêmico SIGAA, na tarefa aberta para este fim.

—

## Exercício 1 – Criação de Processos com `fork()`

**Conceito:** Um **processo** é um programa em execução, incluindo código, dados e recursos associados. Todo sistema operacional moderno fornece uma **API para criação de processos**.

No mundo **UNIX/Linux**, a criação de processos é realizada pela dupla de chamadas `fork()` e `exec()`:

- `fork()` duplica o processo atual, criando um novo processo (o filho);

- `exec()` pode ser usada pelo filho para carregar um novo programa em sua memória.

Neste exercício vamos observar em detalhes apenas a chamada `fork()`. Um processo ao invocar o `fork()` solicita ao SO a sua duplicação (criação de um filho). O código do processo criador (pai) permanece exatamente o mesmo, porém a área de dados é duplicada (e copiada). O `fork()` possui, portanto, um duplo retorno:

- o PID do filho é retornado para o processo pai;
- o valor 0 é retornado para o processo filho.

**Prática:**

1. Crie o arquivo `fork1.c`:

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

int main() {
    printf("Antes do fork, PID=%d\n", getpid());
    pid_t pid = fork();

    if (pid == 0) {
        printf("Filho: PID=%d, PPID=%d\n", getpid(), getppid());
        sleep(10);
    } else {
        printf("Pai: PID=%d, Filho=%d\n", getpid(), pid);
        sleep(10);
    }
    return 0;
}

```

2. Compile e execute:

```
gcc fork1.c -o fork1
./fork1 &

```

3. Em outro terminal, liste os processos criados:

```
ps -o pid,ppid,stat,comm | grep fork1

```

4. Observe que aparecem dois processos: o pai e o filho, ambos executando o mesmo programa.

—

## Exercício 2 – Múltiplos forks e visualização no `pstree`

**Conceito:** Um processo pode chamar o `fork()` mais de uma vez, criando vários filhos. Dessa forma, forma-se uma **árvore de processos**. O comando `pstree` permite visualizar essa hierarquia de forma gráfica no terminal.

**Prática:**

1. Crie o arquivo `fork2.c`:

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

int main() {
    fork();
    fork();
    printf("Processo: PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(20); // mantém os processos vivos
    return 0;
}

```

2. Compile e execute:

```
gcc fork2.c -o fork2
./fork2 &

```

3. Visualize a hierarquia de processos:

```
pstree -p | grep fork2

```

4. Observe: - Foram criados até 4 processos `fork2`. - Eles aparecem organizados como filhos do mesmo processo pai (seu shell). - Essa estrutura corresponde à árvore resultante das chamadas múltiplas a `fork()`.

**Nota**

Neste programa não houve teste do valor de retorno do `fork()`. Assim, tanto o processo pai quanto o filho continuam executando o código subsequente. Na linha 7 ocorre a primeira duplicação. Em seguida, tanto o pai quanto o filho chegam na linha 8 e cada um executa novamente o `fork()`, gerando mais um processo. É por isso que aparecem até 4 instâncias de `fork2` na árvore.

## Exercício 3 – Áreas de Dados Não Compartilhadas

**Conceito:** Após o `fork()`, pai e filho têm cópias independentes da memória. Alterações feitas por um não afetam o outro.

**Prática:**

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

int x = 10;

int main() {
    printf("Antes do fork: x=%d (PID=%d)\n", x, getpid());
    pid_t pid = fork();

    if (pid == 0) {
        x = 20;
        printf("Filho: x=%d (PID=%d)\n", x, getpid());
    } else {
        sleep(1);
        printf("Pai: x=%d (PID=%d)\n", x, getpid());
    }
    return 0;
}
```

**Observação:** O filho imprime 20, o pai continua com 10. Se fosse memória compartilhada, o pai também veria 20.

**Nota**

Na chamada `fork()`, o sistema operacional cria um novo processo com uma cópia da área de dados do processo pai. Isso significa que pai e filho partem do mesmo estado inicial das variáveis, mas cada um tem a sua própria cópia. A modificação feita pelo filho (`x = 20`) não afeta o valor da variável no pai (`x = 10`), pois os espaços de memória são distintos. Esse mecanismo é chamado de **copy-on-write** em implementações modernas, ou seja, a cópia física das páginas de memória só ocorre quando uma delas é modificada.

—

## Exercício 4 – Arquivos Abertos e Descritores Compartilhados

**Conceito:** Durante o `fork()`, a tabela de descritores de arquivos do pai é copiada para o filho. Cada entrada dessa tabela aponta para uma estrutura global de arquivo mantida pelo kernel.

Portanto:

- pai e filho inicialmente compartilham o mesmo deslocamento (offset) e flags do arquivo;
- alterações no deslocamento feitas por um afetam o outro;
- cada processo, no entanto, pode alterar sua própria tabela de descritores (fechar, duplicar ou redirecionar), sem impactar os descritores já existentes no outro processo.

Diferentemente da memória (que é copiada e isolada), os descritores são herdados de forma que pai e filho referenciam as mesmas estruturas globais de arquivo no kernel.

—

**Prática (parte 1 – escrita concorrente):**

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

int main() {
    int fd = open("saida.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);
    if (fd < 0) { perror("open"); exit(1); }

    pid_t pid = fork();
    if (pid == 0) {
        write(fd, "Mensagem do filho\n", 19);
    } else {
        write(fd, "Mensagem do pai\n", 17);
    }

    close(fd);
    return 0;
}
```

```
gcc fork4a.c -o fork4a
./fork4a
cat saida.txt
```

**Observação:** Pai e filho escrevem no mesmo arquivo, mas a ordem pode variar a cada execução.

—

**Prática (parte 2 – redirecionando saída do filho):**

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

int main() {
    int fd = open("saida.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);
    if (fd < 0) { perror("open"); exit(1); }

    pid_t pid = fork();
    if (pid == 0) {
        // Fecha stdout (descritor 1) e abre novo arquivo no lugar
        close(STDOUT_FILENO);
        open("filho.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);

        printf("Filho redirecionou sua saída\n");
    } else {
        printf("Pai continua escrevendo no terminal\n");
    }

    return 0;
}
```

```
gcc fork4b.c -o fork4b
./fork4b
cat filho.txt
```

**Observação:** - O pai mantém a saída padrão no terminal. - O filho redireciona sua saída padrão (`stdout`) para o arquivo `filho.txt`. - Mostra-se que, apesar de herdarem os descritores, cada processo pode manipulá-los de forma independente.

**Nota: Descritores de arquivos**

Cada processo no UNIX/Linux herda três descritores de arquivos abertos logo na criação:

- **0** – entrada padrão (`stdin`);
- **1** – saída padrão (`stdout`);
- **2** – saída de erro padrão (`stderr`).

No exemplo, fechamos explicitamente o descritor `1` (stdout) no filho, e em seguida o `open()` devolve o próximo descritor livre, que é justamente o 1. Isso faz com que a saída padrão do filho seja redirecionada para o arquivo `filho.txt`, sem afetar o pai.
Essa técnica é equivalente ao uso de `dup2()`, mas deixa claro o funcionamento interno da tabela de descritores.

## Exercício 5 – Conferindo os Descritores com `/proc` e `lsof`

**Conceito:** Podemos inspecionar descritores abertos de processos no Linux.
**Prática:**

1. Compile e rode:

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

int main() {
```

```
    int fd = open("saida.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);
    if (fd < 0) { perror("open"); exit(1); }

    pid_t pid = fork();
    if (pid == 0) {
        dprintf(fd, "Filho escreveu (PID=%d)\n", getpid());
        pause();
    } else {
        dprintf(fd, "Pai escreveu (PID=%d)\n", getpid());
        pause();
    }

    close(fd);
    return 0;
}

```

2. Descubra os PIDs:

```
ps -o pid,ppid,comm | grep fork_fdinfo

```

3. Veja os descritores:

```
ls -l /proc/<PID>/fd/
cat /proc/<PID>/fdinfo/3
lsof -p <PID>

```

**Observação:** Pai e filho compartilham o mesmo inode e offset.

—

## Exercício 6 – Retorno de Status com `wait()`

**Conceito:** Quando um processo filho termina, o sistema operacional mantém suas informações em estado **zumbi** até que o pai colete o seu status. A chamada `wait()` permite ao processo pai:

- bloquear até que um de seus filhos termine;
- recuperar o **código de saída** do filho;
- liberar a entrada do filho na tabela de processos.

Se o pai nunca chama `wait()`, o filho permanece como zumbi até que o pai termine (momento em que o `init/systemd` coleta).

**Prática:**

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

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

    if (pid == 0) {
        printf("Filho (PID=%d) finalizando com código 42\n", getpid());
        return 42; // filho termina com código de saída 42
    } else {
        int status;
```

```c
        wait(&status); // pai espera pelo filho
        if (WIFEXITED(status)) {
            printf("Pai (PID=%d): filho terminou com status=%d\n",
                   getpid(), WEXITSTATUS(status));
        }
    }
    return 0;
}
```

```
gcc fork_status.c -o fork_status
./fork_status
```

**Observação:** O pai imprime o código de saída retornado pelo filho (42).

**Nota: Funcionamento do `wait()`**

- O `wait()` coleta informações de um filho terminado e libera sua entrada na tabela de processos.
- O valor retornado em `status` é codificado:
  - `WIFEXITED(status)` indica se o filho terminou normalmente.
  - `WEXITSTATUS(status)` devolve o código passado no `return` ou `exit()`.
  - `WIFSIGNALED(status)` indica se o filho terminou por um sinal.
- Enquanto o pai não chama `wait()`, o filho permanece como **zumbi**.
- Quando o pai chama `wait()`, o zumbi é removido da tabela de processos.

## Exercício 7 – Processo Zumbi

**Conceito:** Um processo **zumbi** é aquele cujo código já terminou, mas cujo processo pai ainda não chamou `wait()` para coletar o status de saída. O processo zumbi:

- não consome CPU nem memória significativa;
- permanece apenas como uma entrada na tabela de processos;
- desaparece assim que o pai chama `wait()` ou termina (nesse caso o `init/systemd` faz a coleta).

**Prática:**

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Filho terminou (PID=%d)\n", getpid());
        return 0; // termina imediatamente
    } else {
        printf("Pai dormindo sem chamar wait()...\n");
        sleep(30); // pai continua vivo
    }
    return 0;
}
```

```
gcc fork_zumbi.c -o fork_zumbi
./fork_zumbi &
ps -o pid,ppid,stat,comm | grep fork_zumbi
```

**Observação:** - O filho aparece com estado **Z** (zumbi). - O pai aparece em estado **S** (sleeping). - No `htop`, o processo zumbi é destacado com a marcação “zombie”.

**Nota: Ligação com o `wait()`**

No Exercício 6 vimos que o `wait()` coleta o status do filho e remove sua entrada da tabela de processos. Aqui, como o pai não chama `wait()`, o filho permanece como zumbi durante todo o tempo em que o pai está vivo.

## Exercício 8 – Processo Órfão

**Conceito:** Um órfão ocorre quando o pai termina antes do filho. O filho passa a ter PPID=1.
**Prática:**

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

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

    if (pid == 0) {
        printf("Filho iniciado, PPID=%d\n", getppid());
        sleep(20);
        printf("Filho continua, novo PPID=%d\n", getppid());
    } else {
        exit(0);
    }
    return 0;
}
```

**Observação:** Após o pai encerrar, o PPID do filho se torna 1 (init/systemd).
—

## Exercício 9 – Comparação Zumbi vs Órfão

| | **Zumbi** | **Órfão** |
|---|---|---|
| Quando acontece | Filho termina, pai não chama wait() | Pai termina antes do filho |
| Estado | Z | Normal (R/S/etc.) |
| Pai associado | Pai ainda existe | Novo pai é init/systemd |
| Como desaparece | Pai chama wait() ou termina | Filho termina normalmente |
| Impacto | Entrada inútil na tabela de processos | Comportamento esperado |

## Exercício 10 – Substituindo o Filho com `exec()`

**Conceito:** O `fork()` cria um novo processo, que inicialmente é uma cópia do pai. Muitas vezes, o objetivo do processo filho não é executar o mesmo código do pai, mas sim carregar um programa totalmente diferente. Isso é feito com a família de chamadas `exec()`, que substitui a imagem do processo filho por outro programa.

Características importantes:

- O PID do processo filho não muda.
- Código, dados e pilha são substituídos pelo novo programa.
- O pai continua executando normalmente.

—

**Prática:**

1. Crie um programa simples que será chamado pelo filho:

```c
#include <stdio.h>

int main() {
    printf("Programa externo executado com exec()!\n");
    return 0;
}

```

Compile:

```
gcc hello.c -o hello

```

2. Agora crie o arquivo `fork_exec.c`:

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Filho vai executar outro programa...\n");
        execl("./hello", "hello", NULL);
        perror("exec falhou");
    } else {
        printf("Pai continua vivo, PID=%d\n", getpid());
        sleep(30); // mantém o pai ativo
    }
    return 0;
}

```

3. Compile e execute:

```
gcc fork_exec.c -o fork_exec
./fork_exec &

```

4. Liste os processos:

```
ps -o pid,ppid,comm | grep -E "fork_exec|hello"

```

—

**Observação:** - O processo filho, que antes aparecia como `fork_exec`, agora surge como `hello`, pois foi substituído. - O PID permanece o mesmo, mas a “imagem” mudou. - Se o pai não chamar `wait()`, o filho (após terminar o programa chamado com `exec`) poderá permanecer como zumbi até o pai terminar.

**Nota: Lista de argumentos no `exec()`**

Na chamada `execl("./hello", "hello", NULL)`:

- O primeiro parâmetro é o caminho para o programa a ser executado.
- O segundo parâmetro (`"hello"`) corresponde ao `argv[0]` do novo processo.
- Poderiam vir outros argumentos em sequência (`argv[1]`, `argv[2]`, ...).
- O `NULL` final é obrigatório para indicar o fim da lista de argumentos.

## Exercício 11 – Capturando o Valor de Retorno de um Programa Executado com `exec()`

**Conceito:** Quando um processo filho é substituído por outro programa através de `exec()`, o pai continua podendo coletar seu código de saída normalmente com `wait()`. Ou seja:

- O `exec()` muda a imagem do processo filho, mas não muda seu PID.
- O pai ainda enxerga esse PID como seu filho.
- O valor retornado pelo programa executado é repassado ao pai.

—

**Prática:**

1. Crie o programa que será chamado pelo filho, `hello_ret.c`:

```c
#include <stdio.h>

int main() {
    printf("Programa externo: vou terminar com código 7\n");
    return 7;
}

```

Compile:

```
gcc hello_ret.c -o hello_ret

```

2. Agora crie o arquivo `fork_exec_wait.c`:

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Filho vai executar outro programa...\n");
        execl("./hello_ret", "hello_ret", NULL);
        perror("exec falhou");
        return 1; // só roda se exec falhar
    } else {
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Pai: filho terminou com código %d\n",
```

```
                    WEXITSTATUS(status));
        } else {
            printf("Pai: filho terminou de forma anormal\n");
        }
    }
    return 0;
}

```

3. Compile e execute:

```
gcc fork_exec_wait.c -o fork_exec_wait
./fork_exec_wait

```

—

**Observação:** - O programa `hello_ret` termina com código 7. - O pai captura esse valor com `WEXITSTATUS(status)`. - Isso mostra que mesmo após um `exec()`, o mecanismo de coleta via `wait()` continua funcionando normalmente.