Relatório: Criação e Identificação de Processos

Sistemas Operacionais I

12 de maio de 2025

Sumário

1 Introdução

Este relatório apresenta uma análise detalhada sobre a criação e identificação de processos em sistemas Linux através da implementação de programas em linguagem C. Exploramos as funções `getpid()`, `getppid()` e `fork()` para entender o comportamento dos processos, sua hierarquia e como o sistema operacional os gerencia.

2 Programas Desenvolvidos

2.1 Programa 1: Identificação de Processo (pid_info.c)

O primeiro programa criado mostra informações básicas sobre o identificador do processo atual (PID) e o identificador do processo pai (PPID):

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

int main() {
    pid_t pid = getpid();
    pid_t ppid = getppid();

    printf("Informações do Processo:\n");
    printf("PID (Process ID): %d\n", pid);
    printf("PPID (Parent Process ID): %d\n", ppid);

    return 0;
}
```

2.2 Programa 2: Criação de Processos (fork_example.c)

O segundo programa utiliza a função `fork()` para criar um processo filho, demonstrando a bifurcação de execução:

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

int main() {
    pid_t pid;

    // Fork and create a child process
    pid = fork();

    // Check if fork() was successful
    if (pid < 0) {
        // Fork failed
        fprintf(stderr, "Erro ao criar processo filho\n");
        return 1;
    }
    else if (pid == 0) {
        // This code runs in the child process (pid = 0 in the child)
        printf("=== PROCESSO FILHO ===\n");
```

```
        printf("PID do filho: %d\n", getpid());
        printf("PPID do filho (PID do pai): %d\n", getppid());
        printf("O valor de pid para o filho é: %d\n", pid);

        // Adding a small delay to allow the parent process to be observed
        sleep(5);
        printf("Processo filho finalizando...\n");
    }
    else {
        // This code runs in the parent process (pid = PID of the child)
        printf("=== PROCESSO PAI ===\n");
        printf("PID do pai: %d\n", getpid());
        printf("PPID do pai: %d\n", getppid());
        printf("PID do filho criado: %d\n", pid);

        // Wait for the child to finish
        int status;
        waitpid(pid, &status, 0);
        printf("Processo pai finalizando após o término do filho...\n");
    }

    return 0;
}
```

3 Resultados e Observações

3.1 Execução Múltipla do pid_info

Após executar o programa `pid_info` múltiplas vezes, observamos que:

1. O PID é diferente em cada execução, geralmente incrementando sequencialmente (37525, 37526, 37527, etc.).
2. O PPID permaneceu constante (37522) entre execuções, indicando que todas as execuções foram iniciadas pelo mesmo processo pai (o shell).
3. A taxa de incremento do PID pode variar dependendo da atividade do sistema.

3.2 Comportamento com fork()

Na execução do `fork_example`, observamos:

1. O processo pai recebeu um PID único (37530) e reportou o PPID (37522) do shell que o executou.
2. O processo filho recebeu seu próprio PID único (37531).
3. O PPID do processo filho é igual ao PID do processo pai, confirmando a relação pai-filho.
4. O valor retornado por `fork()` é diferente no pai (PID do filho) e no filho (sempre zero).

3.3 Observação com ps

Usando o comando `ps` durante a execução do programa com fork, pudemos visualizar:

1. Os dois processos (pai e filho) aparecendo na lista de processos ativos.
2. A hierarquia entre eles, com o PPID do filho apontando para o PID do pai.
3. Ambos processos executando o mesmo programa (`fork_example`), mas com PIDs distintos.

4 Experimento de Oscilação de PIDs

Para entender melhor como o sistema operacional atribui PIDs em um ambiente com alta atividade, realizamos um experimento adicional criando processos em background e matando-os aleatoriamente enquanto monitorávamos as atribuições de PID.

4.1 Configuração do Experimento

Desenvolvemos os seguintes programas e scripts:

1. **infinite_loop.c**: Um programa C que executa um loop infinito até ser morto, coletando estatísticas de execução.
2. **spawn_processes.sh**: Script que cria múltiplos processos de loop infinito em background.
3. **kill_random.sh**: Script que mata aleatoriamente processos criados anteriormente.
4. **pid_chaos_demo.sh**: Script principal que orquestra a criação e destruição de processos enquanto observa o comportamento dos PIDs.

4.2 Resultados Observados

Execução	Contexto	PID	PPID	Incremento
1	Estado inicial	38568	38560	-
2	Após criar 7 processos	38593	38560	+25
3	Após matar 2 processos	38604	38560	+11
4	Após criar 11 processos	38639	38560	+35
5	Após matar 3 processos	38651	38560	+12
6-11	Durante ciclos rápidos	38662-38727	38560	Variados
12	Após limpeza final	38732	38560	+5

4.3 Conclusões do Experimento

1. **Incremento não linear**: Os PIDs não aumentam por um valor fixo entre execuções.
2. **Influência da atividade do sistema**: A taxa de incremento dos PIDs é diretamente proporcional à atividade de criação e destruição de processos no sistema.
3. **Não reutilização imediata**: O Linux não reutiliza imediatamente os PIDs de processos recém-terminados por questões de segurança.
4. **Comportamento do contador**: O Linux utiliza um contador monotônico crescente para atribuir PIDs, que só retorna ao início após atingir o valor máximo (definido em `/proc/sys/kernel/pid_max`).

5 Perguntas e Respostas

5.1 Qual é a diferença entre getpid() e getppid()? O que eles representam?

- **getpid()**: Retorna o identificador único (PID) do processo que está executando a chamada. Este identificador é atribuído pelo sistema operacional quando o processo é criado.
- **getppid()**: Retorna o identificador do processo pai (PPID), ou seja, o PID do processo que criou o processo atual.

Esses identificadores são essenciais para o sistema operacional controlar, monitorar e gerenciar os processos em execução. O PID é único para cada processo ativo no sistema, enquanto o PPID estabelece a relação hierárquica entre processos.

5.2 O que acontece com o PID do processo filho após o fork()?

Após um `fork()`, o sistema operacional cria um novo processo (filho) que é praticamente uma cópia do processo original (pai). O processo filho recebe um novo PID único, diferente do PID do pai. O sistema operacional geralmente atribui PIDs sequencialmente, então o PID do filho é frequentemente o próximo número disponível na sequência.

Nos testes realizados, observamos que o processo filho obteve um PID único (exemplo: 37531), enquanto seu PPID era o PID do processo pai (37530), estabelecendo claramente a relação de parentesco entre os processos.

5.3 Como o sistema operacional identifica e organiza os processos em execução?

O sistema operacional Linux organiza os processos em uma estrutura hierárquica, conhecida como "árvore de processos". Cada processo, exceto o processo inicial (init/systemd, PID 1), possui um processo pai. Esta organização é mantida através dos PIDs e PPIDs.

O kernel mantém uma tabela de processos (`task_struct` no Linux) que contém informações sobre cada processo, incluindo:

- PID único
- PPID (processo pai)
- Estado do processo (executando, dormindo, zombi, etc.)
- Recursos alocados (memória, arquivos abertos)
- Informações de programação (prioridade, tempo de CPU)
- Contexto de execução

Esta estrutura permite que o sistema operacional controle o ciclo de vida dos processos, gerencie recursos, programe a execução e estabeleça relações entre processos. Comandos como `ps`, `top` e `htop` utilizam essas informações para exibir detalhes sobre os processos em execução.

Para atribuir PIDs, o Linux utiliza um mecanismo de contador sequencial que:

1. Mantém o próximo PID disponível em uma variável global
2. Incrementa esta variável para cada novo processo criado
3. Verifica se o novo valor já está em uso (caso tenha dado uma volta completa)
4. Quando o contador atinge o valor máximo (`pid_max`), retorna ao valor mínimo e busca PIDs disponíveis

Este mecanismo explica por que observamos que os PIDs incrementam de forma variável durante nosso experimento, dependendo da atividade geral do sistema.

5.4 É possível prever quantas vezes o programa imprimirá mensagens após o fork()? Justifique.

Sim, é possível prever. Após um único `fork()` bem-sucedido, o programa imprimirá mensagens exatamente duas vezes:

1. Uma vez no processo pai
2. Uma vez no processo filho

Isso ocorre porque o `fork()` cria uma cópia quase idêntica do processo, incluindo o contador de programa (PC). Após o `fork()`, ambos os processos continuam executando o código a partir do mesmo ponto, mas em espaços de memória separados e com valores de retorno diferentes da função `fork()`:

- No processo pai: retorna o PID do filho
- No processo filho: retorna 0

Com base nisso, os dois processos seguem caminhos de execução distintos no código usando a verificação condicional `if (pid == 0)`, resultando em duas sequências de mensagens.

Se múltiplos `fork()` forem encadeados sem verificações adequadas, o número de processos (e mensagens) cresceria exponencialmente (2^n para n chamadas de fork).

5.5 Por que o mesmo programa pode ter múltiplos processos com identidades distintas?

Um programa pode ter múltiplos processos com identidades distintas porque, no contexto de sistemas operacionais, existe uma diferença fundamental entre um programa e um processo:

- **Programa**: É um arquivo executável que contém código e instruções armazenados em disco.
- **Processo**: É uma instância em execução de um programa, com seu próprio espaço de endereçamento, contador de programa, registradores e recursos do sistema.

Quando um programa é executado, o sistema operacional cria um processo para ele. Quando usamos `fork()`, criamos um novo processo que executa o mesmo código, mas tem sua própria identidade (PID) e cópia independente do espaço de memória. Isso permite que o mesmo código seja executado em contextos diferentes, com dados diferentes.

Esta capacidade é fundamental para a concorrência em sistemas operacionais e permite:

1. Servidores web que atendem múltiplas requisições simultaneamente
2. Shells que executam comandos em background
3. Aplicações multitarefa que executam operações paralelas

Os testes realizados demonstraram claramente que mesmo sendo o mesmo código (programa), cada execução ou instância após um `fork()` recebe uma identidade única (PID) do sistema operacional, permitindo que múltiplas cópias do mesmo programa sejam executadas e gerenciadas independentemente.

Nosso experimento de caos de PIDs mostrou ainda que, mesmo quando executamos repetidamente o mesmo programa (`pid_info`), cada instância recebe um novo PID que depende da atividade geral do sistema. Por exemplo, após criar 7 processos em background, o próximo PID atribuído saltou 25 unidades, mostrando como a atribuição de PIDs é influenciada dinamicamente pela atividade do sistema.

6 Conclusão

Através dos experimentos realizados, pudemos compreender na prática como o sistema operacional Linux gerencia processos. As funções `getpid()`, `getppid()` e `fork()` permitem visualizar e manipular a estrutura hierárquica de processos, demonstrando como o sistema operacional mantém a organização mesmo quando múltiplos processos executam o mesmo código.

Nossa investigação adicional sobre a oscilação de PIDs revelou o comportamento dinâmico e não-determinístico da atribuição de identificadores de processo. Demonstramos que:

1. Os PIDs são atribuídos de forma sequencial, mas com incrementos variáveis
2. A atividade geral do sistema influencia diretamente a taxa de incremento dos PIDs
3. O sistema operacional não reutiliza imediatamente PIDs de processos terminados
4. Mesmo em um ambiente controlado, a atribuição de PIDs é influenciada por fatores externos

A criação deliberada de processos em background e sua terminação aleatória nos permitiu observar de forma controlada o comportamento não-determinístico da atribuição de PIDs, confirmando que o valor do próximo PID é sensível à atividade do sistema como um todo, não apenas à atividade do usuário atual.

Esta capacidade de criar e gerenciar múltiplos processos, cada um com sua identidade única, é um dos fundamentos dos sistemas operacionais modernos, permitindo multitarefa, paralelismo e utilização eficiente dos recursos do sistema.