Criacao e Identificacao de Processos em Linux

Sistemas Operacionais

16 de maio de 2025

1 Introducao

Este relatorio investiga o comportamento da criacao de processos em Linux, com foco no funcionamento das funcoes `getpid()`, `getppid()` e `fork()`. Foi desenvolvido um programa em C que cria uma hierarquia de processos (pai, filho e neto) com diferentes configuracoes de tempo de espera para observar como o sistema gerencia os identificadores de processos (PIDs) e as relacoes entre processos.

2 Analise do Programa

O programa implementa tres experimentos distintos para demonstrar diferentes aspectos do gerenciamento de processos:

- **Experimento A**: "Sem espera- Nem o filho nem o neto dormem
- **Experimento B**: "Pai espera- O processo filho dorme por 100 microssegundos
- **Experimento C**: "Neto orfaos- O processo neto dorme por 100 microssegundos

Cada experimento e executado multiplas vezes, e o programa registra o PID e PPID de cada processo em momentos especificos.

3 Analise dos Resultados

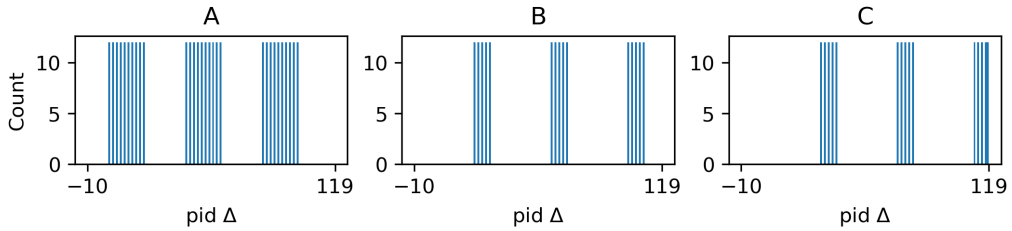

Figura 1: Histograma da diferenca entre PIDs de processos filhos e seus pais

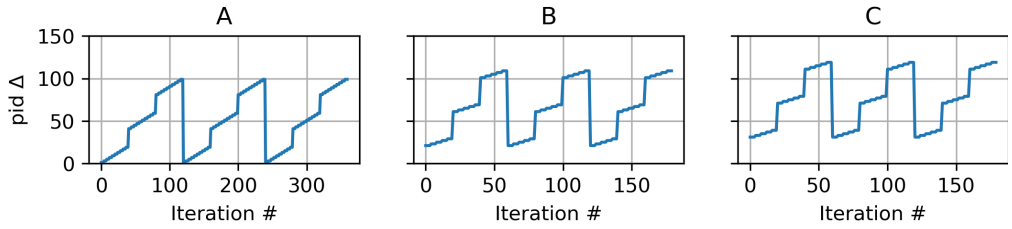

Figura 2: Variacao da diferenca de PIDs ao longo das iteracoes

A Figura 1 mostra que os PIDs sao atribuidos de forma sequencial, com diferencas pequenas entre processos pai e filho.

A Figura 2 mostra como a diferenca entre PIDs varia ao longo das iteracoes, confirmando o comportamento sequencial de atribuicao.

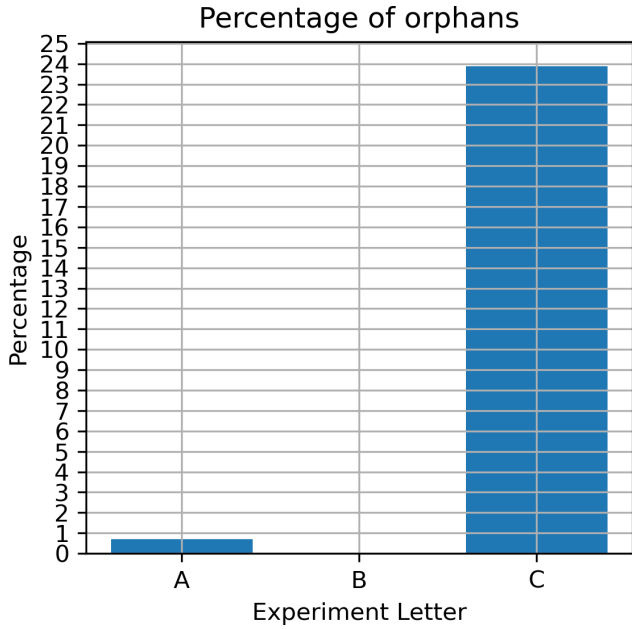

Figura 3: Porcentagem de processos orfaos por experimento

A Figura 3 demonstra que no Experimento C, ha mais processos orfaos (com PPID=1) porque o processo filho termina antes do neto acordar, fazendo com que o neto seja "adotado"pelo processo init (PID 1).

4 Respostas

4.1 Diferenca entre getpid() e getppid()

A funcao `getpid()` retorna o identificador do proprio processo, enquanto `getppid()` retorna o identificador do processo pai. O PID e um numero unico para cada processo, e o PPID estabelece a relacao hierarquica entre processos.

4.2 O que acontece com o PID do processo filho apos o fork()

Apos o `fork()`, o filho recebe um novo PID (geralmente o proximo disponivel), enquanto o pai mantem seu PID original. O PPID do filho e configurado como o PID do pai.

4.3 Como o sistema operacional identifica e organiza os processos

O Linux usa uma estrutura chamada `task_struct` para cada processo, contendo seu PID, PPID, estado e recursos. Os processos sao organizados em arvore hierarquica, onde processos orfaos sao adotados pelo init (PID 1).

4.4 Previsao do numero de mensagens apos o fork()

Nao e possivel prever o numero exato de mensagens, pois depende do escalonamento de processos e da temporização entre pai e filho. Com dois `fork()` em cascata, temos ate quatro fluxos de execucao possiveis.

4.5 Por que o mesmo programa pode ter multiplos processos com identidades distintas

Um programa pode ter multiplos processos porque o `fork()` cria uma copia do processo original com novo PID, permitindo execucao paralela e independente. Cada copia pode seguir caminhos diferentes no codigo, mesmo compartilhando o codigo original.

5 Conclusao

Os experimentos demonstram como o Linux gerencia processos, atribuindo PIDs sequencialmente e adotando processos orfaos com o init. Isso e fundamental para a integridade do sistema e para o modelo de concorrencia do Linux.

6 Apêndice: Código Fonte

A seguir apresentamos o código-fonte completo do programa utilizado nos experimentos. Este programa cria uma hierarquia de processos pai, filho e neto, com diferentes configurações de tempo de espera para cada experimento.

6.1 Definições de estruturas e funções auxiliares

6.2 Função principal de experimento

6.3 Função main

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

struct Experiment {
    size_t repetitions;
    size_t repetition_current;
    char acronym[3];
    char description[40];
    unsigned int sleep_child;
    unsigned int sleep_grandchild;
    pid_t child;
    pid_t grand_child;
};

enum FORK_RESULT {
    FORK_FAIL = -1,
    FORK_CHILD = 0,
    FORK_PARENT = 1,
};

enum FORK_RESULT check_fork(pid_t pid){
    if (pid < 0) return FORK_FAIL;
    if (pid > 0) return FORK_PARENT;
    return FORK_CHILD;
}

void print_process(struct Experiment *e, char const fc[], char id) {
    printf("%s%zu\t%s\t%d\t%d\t%c\n", e->acronym, e->repetition_current,
    ↪  fc,getpid(),getppid(), id);
    setbuf(stdout, NULL);
}

void sleep_process(struct Experiment *e, char const fc[], unsigned int duration){
    printf("%s%zu\t%s\t%d\t%d\tsleep %d\n", e->acronym, e->repetition_current, fc,
    ↪  getpid(), getppid(), duration);
    setbuf(stdout, NULL);
    usleep(duration);
    printf("%s%zu\t%s\t%d\t%d\twokeup\n", e->acronym, e->repetition_current, fc,
    ↪  getpid(), getppid());
    setbuf(stdout, NULL);
}
```

Figura 4: Definições de estruturas e funções auxiliares

```
43 void experiment(struct Experiment *e) {
44     pid_t child_pid = fork();
45 
46     e->child = child_pid;
47     char fc[2];
48 
49     switch (check_fork(child_pid)) {
50         case FORK_FAIL:
51         break;
52         case FORK_CHILD:
53             pid_t grand_child_pid = fork();
54             e->grand_child = grand_child_pid;
55             switch (check_fork(grand_child_pid)) {
56                 case FORK_FAIL:
57                 break;
58                 case FORK_CHILD:
59                     strcpy(fc, "G");
60                     print_process(e, fc,'A');
61                     sleep_process(e, fc, e->sleep_grandchild);
62                     print_process(e, fc,'B');
63 
64                 break;
65                 case FORK_PARENT:
66                     strcpy(fc, "C");
67                     print_process(e, fc,'A');
68                     sleep_process(e, fc, e->sleep_child);
69                     print_process(e, fc,'B');
70                 break;
71             }
72             printf("%s%zu\t%s\t0\t0\tE\n", e->acronym, e->repetition_current,fc);
73             setbuf(stdout, NULL);
74 
75             exit(0);
76         break;
77 
78         case FORK_PARENT:
79             usleep(100 + (e->sleep_grandchild > e->sleep_child ? e->sleep_grandchild :
                ↪  e->sleep_grandchild));
80         break;
81     }
82 }
```

Figura 5: Função experiment que implementa a hierarquia de processos

```c
int main() {
    struct Experiment overseer = {3,0,"M\0", "Main\0", 0, 0};

    struct Experiment experiments[] = {
        {10, 0, "A\0", "Experiment A - Sem espera\0", 0, 0}, // sem sono, race
        {5,  0, "B\0", "Experiment B - Pai espera\0", 100, 0}, // filho dorme
        {5,  0, "C\0", "Experiment C - Neto órfãos\0",0, 100}  // neto dorme
    };

    size_t nof_experiments = sizeof(experiments)/sizeof(struct Experiment);

    printf("%s%zu\t%s\t%d\t%d\t%c\n", overseer.acronym, overseer.repetition_current,
    ↪  "I",getpid(),getppid(), 'I');
    setbuf(stdout, NULL);

    for (size_t m = 0; m < overseer.repetitions; m++){
        for (size_t e = 0; e < nof_experiments; e++){
            for (size_t r = 0; r < experiments[e].repetitions; r++ ){
                experiments[e].repetition_current = r;
                experiment(&experiments[e]);
                wait(&experiments[e].child);
                wait(&experiments[e].grand_child);
            }
        }
    }
}
```

Figura 6: Função main que configura e executa os experimentos