Lista de Exercícios 1
Implementação de Núcleo de Sistemas Operacionais

—

Pontifícia Universidade Católica de Campinas

Prof. Dr. Denis M. L. Martins

Março 2025

1. Fundamentos de Sistemas Operacionais

1.1. Cite três finalidades principais de um sistema operacional.

1.2. Qual é a finalidade das chamadas de sistema?

1.3. Descreva alguns dos desafios inerentes ao projeto de sistemas operacionais para dispositivos móveis em comparação com o projeto de sistemas operacionais para PCs tradicionais.

1.4 Os sistemas operacionais podem ser implementados com diferentes arquiteturas de kernel, como monolítico, microkernel e híbrido.

Com base nisso, analise as afirmações a seguir:

I. O kernel monolítico integra todos os serviços essenciais em um único espaço de execução, proporcionando maior desempenho, mas menor modularidade. II. Um microkernel reduz o tamanho do núcleo ao mínimo possível, delegando serviços como gerenciamento de arquivos e drivers para o espaço de usuário. III. Arquiteturas híbridas combinam características de microkernel e kernel monolítico, buscando equilibrar desempenho e segurança. IV. O uso de microkernel elimina totalmente a necessidade de chamadas de sistema.

Assinale a alternativa correta:

a) Apenas as afirmativas I, II e III estão corretas.
b) Apenas as afirmativas II e IV estão corretas.
c) Apenas as afirmativas I e IV estão corretas.
d) Todas as afirmativas estão corretas.
e) Nenhuma das afirmativas está correta.

1.5. Descreva as ações executadas por um kernel para a mudança de contexto entre processos.

1.6. Sobre conceitos introdutórios de Sistemas Operacionais, classifique cada uma das afirmações a seguir entre Verdadeiro (V) ou Falso (F). - ( ) Para tratamento de uma Chamada de Sistema, o Escalonador transfere o controle do Sistema Operacional para modo usuário, de modo a interagir mais facilmente com a aplicação final. - ( ) Em uma Troca de Contexto, o SO informa o processo em execução para salvar o seu trabalho em arquivos temporários em disco, de forma que ele possa ser retomado no futuro. - ( ) Gerenciamento de Processos é uma das principais tarefas do Sistema Operacional, em que o Escalonador seleciona que processo será o próximo a executar e o Dispatcher inicia a execução. - ( ) Chamadas de Sistema são utilizadas por aplicações de usuário para solicitar serviços do Sistema Operacional, ou seja, solicitar a execução de tarefas que são responsabilidade do Sistema Operacional. - ( ) Em um sistema monolítico, o Kernel é dividido em pequenos módulos, independentes, permitindo maior resiliência a eventuais falhas.

A alternativa que contém a sequência correta de classificação é:

a. F – F – V – V – F
b. V – V – F – F – V
c. F – F – V – V – V
d. V – F – V – V – F
e. F – V – V – F – V

2. Processos e Threads

2.1. Forneça dois exemplos de programação em que o uso de **multithreading** oferece **melhor desempenho** do que uma solução **monothread (de um único thread)**.

2.2. Explique qual será a saída na LINHA A do programa mostrado abaixo.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* processo-filho */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* processo-pai */
        wait(NULL);
        printf("PARENT: value = %d" ,value); /* LINHA A */
        return 0;
    }
}
```

2.3. Incluindo o processo-pai inicial, quantos processos são criados pelos programas abaixo?

```c
// programa1.c
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0){
        fork();
    }
    else if (pid > 0){
        fork();
        fork();
    }

    return 0;
}
```

```c
// programa2.c
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0){
```

```
        fork();
    }
    else if (pid > 0){
        fork();
        fork();
    }

    return 0;
}
```

2.4. Usando o programa abaixo, identifique os valores de pid nas linhas A, B, C e D. (Suponha que os pids reais do pai e do filho sejam 2600 e 2603, respectivamente.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid, pid1;
    /* cria um processo-filho */
    pid = fork();
    if (pid < 0) { /* um erro ocorreu */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* processo-filho */
        pid1 = getpid();
        printf("child: pid = %d", pid); /* A */
        printf("child: pid1 = %d" ,pid1); /* B */
    }
    else { /* processo-pai */
        pid1 = getpid();
        printf("parent: pid = %d", pid); /* C */
        printf("parent: pid1 = %d", pid1); /* D */
        wait(NULL);
    }
    return 0;
}
```

2.5. Considere o segmento de código a seguir:

```
    pid_t pid;
    pid = fork();
    if (pid == 0) {/*processo-filho */
        fork();
        thread_create( ... );
    }
    fork();
```

a. Quantos processos únicos são criados?
b. Quantas threads únicos são criados?

2.6. *Esboce* o código de uma aplicação multithread, usando threads Posix, para executar a multiplicação de duas matrizes quadradas `C[n][n] = A[n][n] x B[n][n]` com elementos inteiros. Assuma que o valor de `n` é uma constante em tempo de compilação ("`define`"), e você não precisa se preocupar com a alocação e inicialização das matrizes `A`, `B` ou `C`. O seu código deve definir uma função `multiplica(...)` que recebe, como parâmetros, apenas três ponteiros `int*` (sendo endereços nas matrizes `A`, `B` e `C`) e não retorna nada. A função deve se encarregar do resultado da multiplicação para um único elemento `C[i][j]`. A rotina principal

do seu programa deve iniciar a execução concorrente de n^2 Threads, uma para cada execução da função multiplica e preenchimento de um elemento `C[i][j]`. Lembre-se de fazer todos os ajustes necessários para passagem correta dos argumentos da função multiplica.

2.7. Um shell é uma aplicação de modo usuário utilizada para iniciar a execução, interativamente, de outros programas conforme solicitado pelo usuário. Para isso, o shell cria um novo processo e executa o novo programa dentro dele. Utilizando as chamadas de sistema e os padrões de chamada Linux estudados em aula, esboce um código em C para fazer a tarefa do shell de iniciar um novo programa. Você pode assumir que existe uma função de biblioteca `GetPgmInfo()` que devolve todas as informações necessárias sobre o programa solicitado pelo usuário já no formato necessário exigido pela chamada de sistema adequada.

2.8. Um processo com `PID=2600` cria um novo processo utilizando o código a seguir. O processo criado recebeu, do Sistema Operacional, o `PID=2603`. Suponha que exista uma função `getpid()` que devolve o valor do PID do processo que a chamou. Dê duas possíveis saídas impressas no terminal para a execução do programa.

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

int main(){
    pid_t pid, pid1;

    /* cria um processo-filho */
    pid = fork();
    if (pid < 0){
        /* um erro ocorreu */
        printf("Fork Failed\n");
        return 1;
    }
    else if (pid == 0) {
        /* processo-filho */
        pid1 = getpid();

        printf("child: pid = %d\n", pid);
        printf("child: pid1 = %d\n", pid1);
    }
    else {
        /* processo-pai */
        pid1 = getpid();

        printf("parent: pid = %d\n", pid);
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }
    return 0;
}
```

3. Escalonamento

3.1. Suponha que os processos da tabela a seguir cheguem para execução nos tempos indicados. Cada processo executará pelo tempo da sua duração de pico.

Processo	Tempo de chegada	Duração do pico
P1	0	8
P2	3	4
P3	10	1

(a) Desenhe um esquema do escalonamento desses processos considerando escalonamento FCFS (first come, first served). Qual é o tempo médio de espera para os três processos?
(b) Desenhe um esquema do escalonamento desses processos considerando escalonamento SRTF (shortest remaining time first, versão preemptiva de shortest job first). Qual é o tempo médio de espera para os três processos?
(c) Desenhe um esquema do escalonamento desses processos considerando escalonamento RR (round robin) com quantum de 2 unidades de tempo, em que todos os três processos têm o mesmo nível de prioridade.

3.2. Cinco tarefas em lote, A até E, chegam a um centro de computadores quase ao mesmo tempo. Elas têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a mais alta. Para cada um dos algoritmos de escalonamento a seguir, determine o tempo de retorno médio do processo. Ignore a sobrecarga de chaveamento de processo. - a. Round Robin - b. Escalonamento por prioridade - c. Primeiro a chegar, primeiro a ser servido (siga a ordem 10, 6, 2, 4, 8) - d. Tarefa mais curta primeiro

3.3. Escalonadores circulares em geral mantêm uma lista de todos os processos executáveis, com cada processo ocorrendo exatamente uma vez na lista. O que aconteceria se um processo ocorresse duas vezes? Você consegue pensar em qualquer razão para permitir isso?

3.4. Um processo, em sistemas operacionais, é uma instância de um programa em execução, que inclui o código do programa, dados e recursos necessários para sua execução. O gerenciamento de processos é fundamental para garantir que múltiplos programas possam ser executados simultaneamente, utilizando recursos como CPU e memória de forma eficiente e organizada. O escalonamento de processos é o mecanismo usado pelos sistemas operacionais para gerenciar a execução de múltiplos processos, determinando a ordem e o tempo que cada processo ocupa na CPU. Técnicas de escalonamento visam otimizar o desempenho do sistema, garantindo eficiência e justiça na alocação de recursos. Sobre processos e escalonamento de processos, analise as afirmativas abaixo, indicando, nos parênteses, V, para as verdadeiras, e F, para as falsas:

i. O módulo do sistema operacional que faz a escolha do processo que receberá tempo de CPU é chamado de escalonador, e o algoritmo que ele usa é chamado de algoritmo de escalonamento.
ii. Os possíveis estados de um processo, após criado e antes de ser encerrado, são em execução, pronto, bloqueado e em escalonamento.
iii. A ocorrência de uma interrupção é capaz de alterar a ordem da execução de processos.

A sequência correta, de cima para baixo, é:

a. V – V – F.
b. F – F – V.
c. V – F – V.
d. F – V – V.

3.5. Aplique o algoritmo de escalonamento preemptivo **Round Robin** com **quantum = 4** à lista de processos apresentada abaixo. Qual das seguintes alternativas representa corretamente os **turnaround times (TR)** dos processos **A, B e C** neste cenário específico de escalonamento? Suponha que processos que terminam antes do próximo fatia de tempo desencadeiem imediatamente uma interrupção do despachante, iniciando uma nova rodada de escalonamento.

Processo	Tempo de Chegada	Tempo de Serviço
A	0	5
B	3	7
C	3	5
D	10	2
E	12	1
F	12	5
G	13	3

a. TR(A) = 13, TR(B) = 16, TR(C) = 21
b. TR(A) = 13, TR(B) = 13, TR(C) = 14
c. TR(A) = 13, TR(B) = 14, TR(C) = 14
d. TR(A) = 13, TR(B) = 13, TR(C) = 21
e. TR(A) = 13, TR(B) = 13, TR(C) = 16
f. TR(A) = 13, TR(B) = 16, TR(C) = 27
g. TR(A) = 8, TR(B) = 14, TR(C) = 8
h. TR(A) = 4, TR(B) = 13, TR(C) = 13
i. TR(A) = 4, TR(B) = 16, TR(C) = 21

4. Sincronização

4.1. Explique o conceito de condição de corrida e como ele se relaciona com o conceito de seção crítica.

4.2. Quais são as três características de uma solução do problema da seção crítica, e o que elas significam?

4.3. Considere a solução a seguir para o problema da exclusão mútua envolvendo dois processos `P0` e `P1`. Presuma que a variável *turn* seja inicializada para 0. O código do processo P0 é apresentado a seguir.

```
/* Outro codigo */
while (turn != 0) { } /* Não fazer nada e esperar */
Critical Section /* . . . */
turn = 0;
/* Outro codigo */
```

Para o processo P1, substitua 0 por 1 no código anterior. Determine se a solução atende a todas as condições exigidas para uma solução de exclusão mútua.

4.4. Nos sistemas operacionais, qual é o problema que acontece quando processos com prioridade baixa permanecem esperando indefinidamente para serem executados, pois, processos de prioridade mais alta estão sendo constantemente escalonados?

- a. Deadlock
- b. Inversão de prioridade
- c. Fragmentação
- d. Starvation (postergação indefinida)
- e. Paginação

4.5. No código abaixo, três threads competem por recursos (identificados por letras maiúsculas **A, B, ...**), enquanto as chamadas `get()` e `release()` indicam o uso de algum **mecanismo de exclusão mútua** (locking/unlocking).

- Forneça uma execução intercalada das três threads que prossiga sem deadlocks.
- Forneça outra execução intercalada das três threads que resulte em um deadlock. Certifique-se de que **todas as três threads estejam envolvidas no deadlock**. Desenhe o **grafo de alocação de recursos** correspondente à execução proposta e destaque o ciclo que indica o deadlock.
- Modifique a ordem das chamadas `get()` para evitar qualquer possibilidade de deadlock. Você **não pode mover as requisições entre procedimentos**, mas pode **alterar a ordem das chamadas `get()`**

dentro de cada procedimento. Justifique o motivo pelo qual sua modificação previne deadlocks. *(4 pontos)*

```
/*
    Code based on Stallings et al. Operating systems: internals and design principles.
    Chapter 6. Vol. 9. New York: Pearson, 2012.
*/
void T0() {
    while(true) {
        get(A);
        get(B);
        get(D);
        // critical region: use A, B, D
        release(A);
        release(B);
        release(D);
    }
}

void T1() {
    while(true) {
        get(D);
        get(C);
        get(A);
        // critical region: use D, C, A
        release(D);
        release(C);
        release(A);
    }
}

void T2() {
    while(true) {
        get(C);
        get(B);
        get(D);
        // critical region: use B, C, D
        release(B);
        release(C);
        release(D);
    }
}
```