Threads e Sincronização no Linux

MCTA026-13 - Sistemas Operacionais

Emilio Francesquini e Fernando Teubl Ferreira
e.francesquini@ufabc.edu.br / fernando.teubl@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Sistemas Operacionais na UFABC**.
- Este material foi preparado com base naquele elaborado e gentilmente cedido pelo Prof. Gustavo Souza Pavani (UFABC)
- As figuras e tabelas foram retiradas do livro [TB]: *Andrew S. Tanenbaum, Herbert Bos.* **Modern Operating Systems**

Threads e sincronização

Objetivos

UFABC

- Vamos aprender a fazer as operações básicas utilizando threads e sincronização
- Vamos também estudar dois problemas clássicos de sincronização

Problemas clássicos de sincronização

UFABC

- São frequentemente usados para testar novas propostas de esquemas de sincronização
  - Problema do buffer de tamanho limitado
  - Problema dos leitores e escritores
  - Problema do jantar dos filósofos
- Apresentaremos soluções baseadas em semáforos

Semáforos

Semáforos

UFABC

- Semáforos são uma ferramenta de sincronização um pouco mais sofisticada que mutexes
- Pode-se enxergar um semáforo como uma variável inteira
- Só pode ser acessado através de 2 operações
  - `wait()` e `signal()` (originalmente chamadas de `P()` e `V()`)
- Definição em pseudo-código de `wait()`

```
wait(S) {
    while (S <= 0); //laço de espera ocupada
    S = S - 1;
}
```

- Definição em pseudo-código de `signal()`

```
signal(S) {
    S = S + 1;
}
```

Usando Semáforos

- Semáforos podem ser usados para resolver uma variedade de problemas de sincronização

Poblema de delimitação de regiões-críticas.

Semáforo `synch` inicializado com 1

```
wait(synch);
//Região crítica
signal(synch);
```

UFABC

Ordenação de execução

Considere que os processos P_1 e P_2 (concorrentes) precisem que o segmento de código S_1 execute antes de S_2. Semáforo `synch` inicializado com 0

```
P1:
    S_1;
    signal(synch);

P2:
    wait(synch);
    S_2;
```

Tipos de semáforos

UFABC

- Semáforo de contagem - O valor inteiro do semáforo pode variar de maneira irrestrita
- Semáforo binário - O valor inteiro só pode assumir os valores 0 e 1
  - É o mesmo que um mutex!
- É possível construir um tipo com o outro. Consegue explicar como?

Exemplo de semáforo de contagem

UFABC

- Permitir que no máximo 2 processos estejam executando em uma região crítica
- Crie um semáforo S inicializado com 2

```
wait(S);
//Região crítica
signal(S);
```

Problemas clássicos de sincronização

Problema do buffer de tamanho limitado

UFABC

- n espaços no buffer com capacidade para um item cada
- Semáforo `mutex` inicializado com o valor 1
- Semáforo `full` inicializado com o valor 0
- Semáforo `empty` inicializado com o valor n

Estrutura do produtor

UFABC

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

Estrutura do consumidor

UFABC

```
do {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
} while (true);
```

- Note a simetria entre os dois

Problema dos leitores e escritores

UFABC

- Um conjunto de dados é compartilhado entre um número de processos concorrentes
  - Leitores - Apenas leem os dados, **não alteram** os dados
  - Escritores - Podem tanto ler quanto escrever

Problema dos múltiplos leitores e escritores

Escritores devem ter acesso exclusivo aos dados (um por vez).

- Dependendo de como o problema for abordado, leitores e escritores podem ter prioridades de execução bem diferentes
- Dados compartilhados:
  - Semáforo `rw_mutex` inicializado com 1
  - Semáforo `mutex` inicializado com 1
  - Inteiro `read_count` inicializado com 0

Estrutura de um escritor

UFABC

```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```

Estrutura de um leitor

UFABC

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

Qual é a deficiência do código acima?

Variações do problema dos leitores e escritores

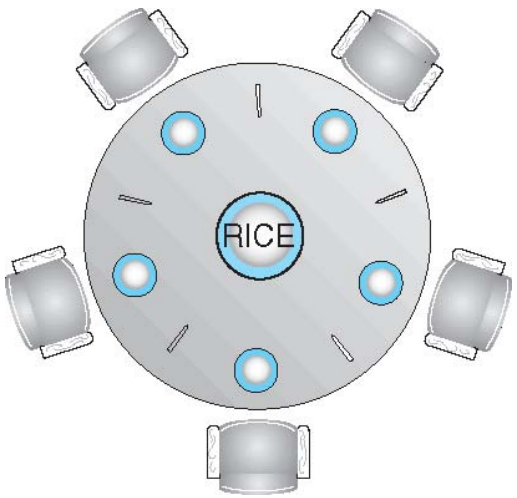
UFABC

- **Primeira** variação - nenhum leitor fica esperando a menos que um escritor esteja utilizando os dados compartilhados
- **Segunda** variação - Uma vez que um escritor pedir para acessar a região crítica, ele ganha acesso assim que possível
- Ambos os casos podem sofrer de inanição (*starvation*) o que leva a diversas variações adicionais
- Em vários sistemas operacionais são oferecidas APIs para este problema específico

O problema do jantar dos filósofos

UFABC

- Os filósofos passam as suas vidas pensando e comendo
- Eles nao interagem com os seus vizinhos
- De vez em quando eles tentam pegar os ”pauzinhos”(JP: 箸 *hashi*, EN: *chopsticks*) para comer
  - Eles pegam um de cada vez
- Para conseguirem comer eles precisam ter ambos em seu poder. Quando acabam de comer, devolvem os pauzinhos à mesa



Uma possível solução: Estrutura do filósofo `i`

UFABC

- Dados compartilhados:
  - Tigela de arroz (conjunto de dados)
  - Semáforos `chopstick[5]` inicializado em 1

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
    // eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
    // think
} while (TRUE);
```

Problemas com a solução

UFABC

- Apesar de garantir que nenhum vizinho pode comer simultâneamente, pode haver um impasse (*deadlock*)
  - Imagine que todos os 5 filósofos tentem comer ao mesmo tempo e todos peguem o pauzinho à esquerda
- Algumas possíveis soluções
  - Permitir que no máximo 4 filósofos comam ao mesmo tempo
  - Permitir que os filósofos peguem os pauzinhos apenas se ambos estiverem disponíveis (é preciso que seja feito em uma seção crítica. Por que?)
  - Utilizar uma solução assimétrica - um filósofo de índice ímpar pega primeiramente o pauzinho à direita e depois à esquerda enquanto um filósofo de índice par pega primeiro o pauzinho à esquerda e depois à direita

Roteiro

UFABC

- Exercício 1 Criação de threads.
  1. Compile com a opção `-lpthread` o programa `ex1_1.c`. Execute-o várias vezes e comente sobre as saídas. Explique o funcionamento do programa.
  2. Compile com a opção `-lpthread` o programa `ex1_2.c`. Execute-o várias vezes e comente sobre as saídas. Qual a solução para o problema apresentado?
- Exercício 2 Implementando a exclusão mútua.
  1. Compile com a opção `-lpthread` o programa `ex2.c`. Execute-o várias vezes e comente sobre as saídas.

UFABC

- **Exercício 3** Operações atômicas
  Como as instruções atômicas variam de processador para processador, compiladores como o GCC oferecem uma série de funções *built-in* que quando compiladas são transformadas em uma instrução específica para cada processador. As instruções *test and set* e *compare and swap* podem ser executadas chamando-se as *built-ins* `__sync_lock_test_and_set` e `__sync_val_compare_and_swap` respectivamente[1].
  1. Altere o código `ex2.c`, mais especificamente as funções `enter_region()` e `leave_region()` para que se baseiem na operação TAS.
  2. Faça o mesmo, mas desta vez utilizando a operação CAS.

[1] Veja a documentação em https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

UFABC

- Exercício 4 Impasse.
  1. Compile com a opção `-lpthread` e execute o programa `ex4_1.c`. O que aconteceu na saída? Justifique.
  2. Compile com a opção `-lpthread` e execute o programa `ex4_2.c`, que é baseado na figura 2.38 do livro texto [TB]. O problema do programa anterior foi resolvido? Justifique.