T1 - Processos e Threads

Vinicius Gasparini

SOP - BCC - UDESC - 2020.2

1 Exercício 1

Com o objetivo de garantir que apenas as saídas "`CAOS!\n`" e "`CASO!\n`" sejam reproduzidas e que ambas possuam a mesma probabilidade de ser executadas, fez-se necessário o uso de semáforos como segue descrito no Algoritmo 1.

```
1  semaphore s1 = 0, s2 = 1, s3 = 0, s4 = 0;
2
3  void A() {
4      while(1) {
5          down(&s1);
6          printf("O");
7          up(&s1);
8          up(&s3);
9      }
10 }
11
12 void B() {
13     while(1) {
14         printf("CA");
15         up(&s1);
16         down(&s2);
17         down(&s3);
18         down(&s1);
19         printf("!\n");
20     }
21 }
22
23 void C() {
24     while(1) {
25         down(&s1);
26         printf("S");
27         up(&s1);
28         up(&s2);
29     }
30 }
```

Algoritmo 1: Extração da escala.

Para solução do problema proposto, utilizou-se três semáforos de sincronização (isto é, inicializado em 0). As funções `A` e `C` iniciam esperando sinal do semáforo 1 (`s1`) para garantir que "`CA`" seja a primeira saída escrita. Após

escrito "CA", a função B sinaliza o s1, liberando A e C. Essa concorrência faz com que a saída seguinte seja unicamente influenciada pelo escalonador. Uma vez dentro de alguma das funções, é executado o printf, seja qual for, e então é sinalizado o semáforo s1. De volta a função B, tanto s2 quanto s3 entram em espera. Isso faz com que a outra função que ainda não realizou seu printf retome sua execução e sinalize s1 e s2 ou s3, a depender de quem estiver executando no momento. Isso libera a função B para impressão de "!\n" determinando o fim do ciclo. Importante destacar o sinal down(&s1) antes do printf("!\n"). Ele se faz necessário para reiniciar o estado do semáforo s1 para que no próximo ciclo todos estejam zerados.

É possível testar o funcionamento deste semáforo por meio do código ex1.c anexo, onde foram implementados *asserts* para garantir ciclicidade.

2 Exercício 2

A implementação deste exercício faz uso de *mutexes* e *barriers*. Os *mutexes* são utilizados em 2 momentos.

1. Para garantir exclusão mútua de uma região crítica. Isto é, garantir que apenas uma *thread* se declare vencedora.

2. Para sincronizar o fim da busca de todas as *threads*.

Já a *barrier* foi utilizada para garantir que todas as matrizes foram instanciadas, preenchidas e sorteadas antes do inicio de cada busca.

Para controlar o placar da "corrida" foi utilizado um vetor estático de tamanho MAX_THREADS. Optou-se por essa solução pois como o dado histórico não é utilizado, a complexidade do cálculo final das execuções cai de $\mathcal{O}(R)$ para $\mathcal{O}(N)$, sendo R a quantidade de *rounds* enquanto N seria MAX_THREADS.

O programa é executado pelos comandos bash conforme Algoritmo 2.

```
1    $ make all|
2    $ ./ex2 M N R \{0|1|2\}
```

Algoritmo 2: Execução do exercício 2.

Os parâmetros M N R conforme especificado se referem, respectivamente, a quantidade de linhas, colunas e rodadas. Já o último parâmetro é opcional e varia entre 0, 1 e 2. Sendo este o nível de *debbug*. Sendo 0 sem nenhum tipo de informação, 1 algumas informações de inicialização e encerramento de *thread*. Por fim, o nível 2 oferece informações sobre cada iteração da busca de todas as *threads*.

3 Exercício 3

a) Conforme já estudado, a estrutura da tabela de processos de um sistema Unix pode na verdade ser abstraída para uma árvore por conta das suas

propriedades intrínsecas de hierarquia. Isso pode nos levar a imaginar que uma árvore poderia ser utilizada para representar todos os processos. Porém este raciocínio não leva em consideração o desempenho de tal estrutura. Para navegar entre os processos seria necessário algoritmos de caminho em árvore para alternar entre execuções. Para mitigar esse problema, podemos implementar a tabela de processos como um vetor estático de `struct proc`. Optou-se por utilizar um vetor estático para prevenir problemas como *fork bombs*, mesmo sabido que essa solução demande conhecimento prévio da disponibilidade de memória da máquina. Por tanto, faz-se uso da constante `TABLE_SIZE` que é definida pela equação conforme descrito no trecho de código abaixo.

```
\\TABLE_SIZE = ( MAX_MEMORY * PROCESS_LIMIT_PERCENTAGE / 100 )
\\              / sizeof(struct proc)

\\Sendo MAX_MEMORY a capacidade em bytes da mem ria do
    sistema
\\PROCESS_LIMIT_PERCENTAGE o limite em porcentagem (0 - 100)
    de uso da memoria para tabela de processos

#define TABLE_SIZE 1024
struct proc process_table[TABLE_SIZE]
```

b) Para o caso de adequar o descritor para as arquitetura x86-64 e MIPS deverão modificados os registradores de contexto. Isto ocorre pois na arquitetura em cada arquitetura são utilizados registradores diferentes para diferentes propósitos. Quanto a estrutura `struct proc`, essa pode se manter inalterada. O compilador lidará com os limites de tipo em particular para cada hardware.

c) Para representar *threads* o atributo `char *mem`, isto é, o endereço de memória do inicio do processo agora deverá ser compartilhado no descritor de *thread* pois é uma premissa básica o compartilhamento de memória entre *threads* instanciadas pelo mesmo processo. Para implementar *kernel threads* faz-se necessário modificar as operações de `fork` e `init` para previr o usuário de criar-las/encerrar-las, além disso, o endereçamento de memória para processos de *kernel* (e por consequência, o endereçamento de memória de suas *threads*) deverá ser compartilhado.