**Universidade do Estado de Santa Catarina – Bacharelado em Ciência da Computação**
**Sistemas Operacionais – Prof. Rafael R. Obelheiro – Semestre: 2020.2**

# Trabalho 1 – Processos e Threads

## Sobre o trabalho

- O trabalho é individual.
- Exercícios de implementação devem ser realizados na linguagem C, e serão testados no sistema operacional Linux. As submissões estarão sujeitas a controle de plágio usando ferramentas de detecção de similaridade.
- Exercícios teóricos devem ser submetidos em formato PDF. Para os exercícios de implementação deve ser submetido o código fonte, em formato compilável (ou seja, os arquivos `*.c`, e não listagens em PDF). Submeta um único arquivo ZIP contendo todas as suas respostas (teóricas e de implementação).
- O trabalho deverá ser entregue até **SEXTA-FEIRA, 18 DE DEZEMBRO**. O Moodle aceitará submissões até 23h59min, sendo automaticamente bloqueado após a data limite. É **RESPONSABILIDADE DOS ALUNOS** garantir que o trabalho seja entregue no prazo.
- Em caso de dificuldades ou dúvidas na interpretação do trabalho, entre em contato com o professor (`rafael.obelheiro@udesc.br`).

## Exercícios

1. Considere um programa concorrente com três *threads*, `A`, `B` e `C`, mostradas no pseudocódigo abaixo. Deseja-se que a sequência de execução seja tal que o programa imprima para toda a eternidade (ou enquanto o programa executar) as mensagens "`CASO!\n`" ou "`CAOS!\n`", dependendo da sequência de escalonamento (ou seja, em cada linha da saída as duas palavras devem ser igualmente possíveis). Mostre como garantir isso usando semáforos. A solução não deve alterar a estrutura do pseudocódigo existente, apenas incluir instruções de manipulação de semáforos para garantir a sequência de execução desejada. Você pode inicializar os semáforos como variáveis regulares (`semaphore x=1`) e manipulá-los com os pares de primitivas `down()/up()` ou `sem_wait()/sem_post()`.

   **NOTA**: este exercício é teórico; trabalhe com pseudocódigo, sem se preocupar em produzir uma implementação completamente funcional.

```
A1 void A() {
A2   while (1) {
A3     printf("O");
A4   }
A5 }
```

```
B1 void B() {
B2   while (1) {
B3     printf("CA");
B4     printf("!\n");
B5   }
B6 }
```

```
C1 void C() {
C2   while (1) {
C3     printf("S");
C4   }
C5 }
```

2. Escreva um programa em C no Linux usando a biblioteca Pthreads e que atenda aos seguintes requisitos:

   R1. O programa recebe três parâmetros $m$, $n$ e $r$ na linha de comando; ou seja, o programa deve ser invocado como

   ```
   $ ./prog m n r
   ```

   R2. O programa deve alocar dinamicamente uma matriz $A$ de $m \times n$ números inteiros, onde $m$ é o número de linhas e $n$ é o número de colunas, e preenchê-la com números entre 1 e $m \times n$. Os números devem estar em posições aleatórias (isto é, dispostos de forma não sequencial), e não podem ser repetidos (sugestão: preencha a matriz sequencialmente e depois faça um rearranjo aleatório dos elementos). Por exemplo, para $m = 2$ e $n = 3$, a matriz abaixo seria válida:

   $$\begin{pmatrix} 5 & 2 & 1 \\ 6 & 3 & 4 \end{pmatrix}$$

   R3. O programa deve criar quatro *threads*, todas elas executando a mesma função (descrita mais à frente).

R4. Cada *thread* deve receber como parâmetros, pelo menos, o seu número de identificação (de 1 a 4) e um outro parâmetro que permita determinar o ponto inicial de busca. Podem ser passados quaisquer parâmetros adicionais, conforme a necessidade.

R5. O programa deve realizar $r$ rodadas do seguinte procedimento:

i. `main()` escolhe um número aleatório entre 1 e $m \times n$;

ii. as *threads* buscam (em paralelo) o número sorteado na matriz;

iii. a primeira *thread* a encontrar o número é considerada a vencedora da rodada.

R6. Cada *thread* deve buscar o número sorteado por `main()` partindo de um canto diferente da matriz ($a_{11}, a_{1n}, a_{m1}, a_{mn}$) e percorrendo todos os elementos até encontrar o número procurado. Como a matriz contém todos os números entre 1 e $m \times n$, o número sorteado sempre será encontrado.

R7. Todas as *threads* devem iniciar a busca ao mesmo tempo, depois que o número for sorteado por `main()`.

R8. A primeira *thread* que encontrar o número deve inserir o seu número de identificação em uma lista com a vencedora de cada rodada (i.e., o $k$-ésimo elemento da lista será a *thread* mais rápida da $k$-ésima rodada). Essa lista pode ser estática ou dinâmica.

R9. Depois de encontrar o número sorteado, uma *thread* deve bloquear até que outro número seja sorteado, ou, se for a última rodada, até que a última *thread* termine o seu processamento. Em outras palavras, uma *thread* só pode passar para a próxima rodada (ou finalizar) quando todas as *threads* tiverem concluído a busca. O programa principal (`main()`) deve esperar que todas as *threads* terminem a rodada $k$ antes de avançar para a rodada $k + 1$.

R10. Ao final das $r$ rodadas, o programa principal deve mostrar o número de rodadas em que cada *thread* foi a mais rápida, e declarar a(s) vencedora(s).

R11. O programa deve tratar condições de disputa no código.

O exemplo a seguir ilustra o formato esperado para a saída do programa (invocado com $m = 30$, $n = 40$ e $r = 10$) quando as *threads* 1 e 3 foram as mais rápidas em duas rodadas cada uma, e as *threads* 2 e 4 foram mais rápidas em três rodadas cada.

```
$ ./prog 30 40 10
  thread   1 =>      2 vitorias
  thread   2 =>      3 vitorias
  thread   3 =>      2 vitorias
  thread   4 =>      3 vitorias
-------------------------------
Thread(s) vencedora(s): 2 4
```

3. Um mecanismo fundamental em um sistema operacional é o **descritor de processo**, uma estrutura de dados que armazena os dados necessários para implementar a abstração de processo. A Figura 1 mostra uma versão levemente simplificada do descritor de processo do xv6, um sistema operacional didático que é uma reimplementação do Unix versão 6 para arquiteturas modernas (x86 e RISC-V). O descritor propriamente dito é representado por uma `struct proc` (linhas 16–30). Essa estrutura contém campos que representam:

- o espaço de endereçamento do processo (`mem`, `sz` e `kstack`, linhas 17–19);
- o estado de execução do processo (`state`, linha 20). Os estados possíveis são enumerados em `procstate` (linha 12);
- o identificador do processo (`pid`, linha 21);
- um ponteiro para o descritor do processo pai (`parent`, linha 22);
- registradores de CPU que formam o contexto atual de execução (`context` e `trapframe`, linhas 23 e 24);
- um ponteiro para um canal no qual o processo está bloqueado esperando dados (`chan`, linha 25);
- uma flag que indica se o processo foi encerrado (`killed`, linha 26);
- um ponteiro para uma lista de descritores dos arquivos abertos pelo processo (`ofile`, linha 27);
- um ponteiro para o descritor do diretório corrente (`cwd`, linha 28);
- o nome do processo usado para imprimir mensagens de depuração (`name`, linha 29).

Com base nos conceitos de implementação de processos e *threads*, responda às seguintes questões:

(a) Mostre a declaração em C de uma variável que armazene a tabela de processos do sistema. Você pode considerar ou não um limite máximo para o número de processos (deixe explícita sua escolha).

(b) O descritor mostrado na Figura 1 é o usado na arquitetura x86 de 32 bits. Caso você quisesse portar o xv6 para uma arquitetura diferente (MIPS ou x86-64, por exemplo), quais campos da `struct proc` provavelmente teriam que ser redefinidos? Justifique sua resposta.

(c) O xv6 não oferece suporte a *threads* de *kernel*, e cada processo têm uma única *thread*. Caso você quisesse implementar suporte a *threads* de *kernel*, quais campos da `struct proc` teriam de sair do descritor de processo e passar para um descritor de *thread*? Justifique sua resposta.

```
// registradores que o xv6 salva e restaura para suspender e depois
// retomar a execução de um processo
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

// todos os possíveis estados de um processo
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// as informações que o xv6 mantém sobre cada processo, incluindo o seu
// contexto de registradores e estado de execução
struct proc {
  char *mem;                  // início (endereço base) da memória do processo
  uint sz;                    // tamanho da memória do processo (bytes)
  char *kstack;               // base da pilha de kernel para este processo
  enum procstate state;       // estado do processo
  int pid;                    // ID do processo (PID)
  struct proc *parent;        // processo pai
  struct trapframe *tf;       // registradores de CPU salvos no tratamento de interrupções
  struct context *context;    // swtch() aqui para executar o processo
  void *chan;                 // se != 0, bloqueado em chan
  int killed;                 // se != 0, processo foi encerrado
  struct file *ofile[NOFILE]; // arquivos abertos
  struct inode *cwd;          // diretório atual
  char name[16];              // nome do processo (usado para debugging)
};

```

Figura 1: Descritor de processo do xv6 (`https://github.com/mit-pdos/xv6-public/blob/master/proc.h`)