Spinlocks e semáforos

Este projeto faz parte do PPOS v2.

O objetivo deste projeto é implementar **spinlocks** e **semáforos clássicos** em nosso sistema:

- *Spinlock*: controle de exclusão mútua baseado em espera ocupada e instruções atômicas do processador;
- *Semáforo clássico*: controle de exclusão mútua eficiente e justo, conforme a definição de Dijkstra.

Spinlocks

O *spinlock* é um controle de exclusão mútua baseado em espera ocupada. Para implementá-lo, sugere-se usar operações atômicas sobre números inteiros. As operações de travar e destravar devem ser implementadas:

```
void spin_lock(int *lock)

void spin_unlock(int *lock)
```

O *spinlock* é representado por um inteiro `lock`, cujo valor inicial em 0 (zero) indica que a trava está livre.

Semáforos

Cada semáforo é representado por uma estrutura `semaphore_t` que deve ser definida no arquivo `kernel/semaphore.c`. Essa estrutura contém um ID, um contador, uma fila e um inteiro para o controle de concorrência no acesso ao semáforo (*spinlock*).

As funções a seguir estão declaradas em `semaphore.h` e devem ser implementadas em `semaphore.c`:

Cria um semáforo

```
int sem_create(int value);
```

Cria um semáforo com o valor inicial `value` e uma fila vazia. A chamada retorna o UID do semáforo ou -1 em caso de erro.

Requisita um semáforo

```
int sem_down(int sem_id)
```

Realiza a operação *Down* no semáforo com ID `sem_id`. Esta chamada **pode ser bloqueante**: caso o contador do semáforo seja negativo, a tarefa corrente é suspensa, inserida no final da fila do semáforo e a execução volta ao *dispatcher*; caso contrário, a tarefa continua a executar sem ser suspensa.

Se a tarefa for suspensa, ela será acordada mais tarde, quando uma outra tarefa liberar o semáforo (através da operação `sem_up`) ou caso o semáforo seja destruído (operação `sem_destroy`).

A chamada retorna 0 em caso de sucesso ou -1 em caso de erro (semáforo não existe ou foi destruído).

Libera um semáforo

```
int sem_up(int sem_id)
```

Realiza a operação *Up* no semáforo com UID `sem_id`. Esta chamada **não é bloqueante** (a tarefa que a executa não perde o processador). Se houverem tarefas aguardando na fila do semáforo, a primeira da fila deve ser acordada e retornar à fila de tarefas prontas.

A chamada retorna 0 em caso de sucesso ou -1 em caso de erro (semáforo não existe ou foi destruído).

Destrói um semáforo

```
int sem_destroy(int sem_id)
```

Destrói o semáforo com UID `sem_id`, liberando esse UID e acordando todas as tarefas que aguardavam por ele.

A chamada retorna 0 em caso de sucesso ou -1 em caso de erro.

As tarefas que estavam suspensas aguardando o semáforo que foi destruído devem ser acordadas e retornar da operação *Down* correspondente com um **código de erro** (valor de retorno -1).

UIDs dos semáforos

Cada semáforo é identificado pela aplicação por um UID (*Unique IDentifier*) inteiro positivo. Use um mapa genérico para gerenciar esses UIDs.

Condições de disputa

Caso duas tarefas tentem acessar o mesmo semáforo simultaneamente, podem ocorrer **condições de disputa** nas variáveis internas dos semáforos. Por isso, as funções que implementam os semáforos devem ser protegidas usando um mecanismo de **exclusão mútua**.

Obviamente não podemos usar semáforos para resolver esse problema  ... Por isso, deve ser usado um mecanismo mais primitivo, como o *spinlock*.

Arquivos

Os seguintes arquivos são relevantes para este projeto:

- `kernel/semaphore.c`: implementação de *spinlocks* e semáforos
- `test/pingpong-semaphore.c`: programa de teste simples
- `test/pingpong-semaphore-stress.c`: programa de teste exaustivo
- `test/pingpong-semaphore*.txt`: saídas esperadas dos testes

Os semáforos deverão ser totalmente implementados em seu código; sua implementação não deverá usar funções de semáforo de bibliotecas externas ou do sistema operacional

subjacente.

Outras informações

- Duração estimada: 4 horas.
- Dependências:
  - Tarefas cooperativas
  - Despachante de tarefas
  - Preempção por Tempo
  - Tarefas que esperam
  - Tarefas que dormem
  - Mapa genérico