Programação Concorrente

Semáforos

Prof. Eduardo Alchieri

Semáforos

- **Semáforos**
  - Também baseados em um tipo de variável (semáforo) que pode sofrer duas operações básicas: DOWN e UP (generalização das primitivas sleep e wakeup)
    - O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes
    - Sempre que deseja acessar o recurso compartilhado, um processos executa uma instrução DOWN
      - Se o semáforo for maior que 0, este é decrementado de 1, e o processo que solicitou a operação pode executar sua região crítica
      - Entretanto, se uma instrução DOWN é executada em um semáforo cujo valor seja igual a 0, o processo que solicitou a operação ficará no estado de espera

Semáforos

- **Semáforos**
  - O processo que está acessando o recurso, ao deixar de acessar o recurso, executa uma instrução UP, incrementando o semáforo de 1 e liberando o acesso ao recurso
  - A verificação do valor do semáforo, a modificação do seu valor e, eventualmente a colocação do processo para dormir são operações atômicas (são únicas e indivisíveis)
  - Os semáforos aplicados ao problema da exclusão mútua são chamados de ***mutex*** (*mutual exclusion*) ou binários, por apenas assumirem os valores 0 e 1

Semáforos

- Problema do produtor e consumidor usando semáforos

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE é a constante 1 */
        item = produce_item( );                 /* gera algo para pôr no buffer */
        down(&empty);                           /* decresce o contador empty */
        down(&mutex);                           /* entra na região crítica */
        insert_item(item);                      /* põe novo item no buffer */
        up(&mutex);                             /* sai da região crítica */
        up(&full);                              /* incrementa o contador de lugares preenchidos */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* laço infinito */
        down(&full);                            /* decresce o contador full */
        down(&mutex);                           /* entra na região crítica */
        item = remove_item( );                  /* pega item do buffer */
        up(&mutex);                             /* sai da região crítica */
        up(&empty);                             /* incrementa o contador de lugares vazios */
        consume_item(item);                     /* faz algo com o item */
    }
}
```

Semáforos

- **Exercícios**
  - Problema da escolha das impressoras
  - Problema da fórmula 1
  - Problema dos pombos
  - Problema dos fumantes
  - Jantar dos filósofos
  - O problema do barbeiro sonolento

Semáforos

- Semáforos possuem um poder computacional equivalente a locks (mutex) e variáveis condição
  - É possível implementar semáforos utilizando locks e variáveis condição
  - É possível implementar locks e variáveis condição utilizando semáforos

Semáforos

- Implementar um semáforo usando locks e variáveis condição

```
typedef struct {
  int value;    /* Valor atual do semáforo */
  int n_wait;   /* Número de threads esperando */
  mutex_t lock;
  cond_t cond;
} sem_t;
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_init

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value) {
  sem->value = value;
  sem->n_wait = 0;
  mutex_init(&sem->lock, NULL);
  cond_init(&sem->cond, NULL);
  return 0;
}
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_wait

```
int sem_wait(sem_t * sem) {
  mutex_lock(&sem->lock);
  if (sem->value > 0)
    sem->value--;
  else {
    sem->n_wait++;
    cond_wait(&sem->cond, &sem->lock);
  }
  mutex_unlock(&sem->lock);
  return 0;
}
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_post

```
int sem_post(sem_t * sem) {
  mutex_lock(&sem->lock);
  if (sem->n_wait) {
    sem->n_wait--;
    cond_signal(&sem->cond);
  } else
    sem->value++;
  mutex_unlock(&sem->lock);
  return 0;
}
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_trywait

```
int sem_trywait(sem_t * sem) {
  int r;
  mutex_lock(&sem->lock);
  if (sem->value > 0) {
    sem->value--;
    r = 0;
  } else
    r = EAGAIN;
  mutex_unlock(&sem->lock);
  return r;
}
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_getvalue

```
int sem_getvalue(sem_t *sem, int *sval) {
  mutex_lock(&sem->lock);
  *sval = sem->value;
  mutex_unlock(&sem->lock);
  return 0;
}
```

Semáforos

- Implementar um semáforo usando locks e variáveis condição

sem_destroy

```
int sem_destroy(sem_t *sem) {
  if (sem->n_wait)
     return EBUSY;
  mutex_destroy(&sem->lock);
  cond_destroy(&sem->cond);
  return 0;
}
```

Semáforos

- Implementar um lock (mutex) usando semáforos

```
typedef struct {
  sem_t sem;
} mutex_t;
```

mutex_init e mutex_destroy

```
int mutex_init(mutex_t *lock, mutex_attr* attr) {
  return sem_init(&lock->sem, 0, 1);
}

int mutex_destroy(mutex_t *lock) {
  return sem_destroy(&lock->sem);
}
```

Semáforos

- Implementar um lock (mutex) usando semáforos

mutex_lock e mutex_unlock

```
int mutex_lock(mutex_t *lock) {
  return sem_wait(&lock->sem);
}

int mutex_unlock(mutex_t *lock) {
  return sem_post(&lock->sem);
}
```

Semáforos

- Implementar variáveis de condição usando semáforos e locks

```
typedef struct {
  mutex_t lock;
  sem_t sem;
  int n_wait;
} cond_t;
```

cond_init

```
int cond_init(cond_t *cond) {
  mutex_init(&cond->lock, NULL);
  sem_init(&cond->sem, 0, 0);
  n_wait = 0;
  return 0;
}
```

Semáforos

- Implementar variáveis de condição usando semáforos e locks

cond_wait

```
int cond_wait(cond_t *cond,
              mutex_t *mutex_externo) {
  mutex_lock(&cond->lock);
  cond->n_wait++;
  mutex_unlock(&cond->lock);
  mutex_unlock(mutex_externo);
  sem_wait(&cond->sem);
  mutex_lock(mutex_externo);
  return 0;
}
```

Semáforos

- Implementar variáveis de condição usando semáforos e locks

cond_signal

```
int cond_signal(cond_t *cond) {
  mutex_lock(&cond->lock);
  if (cond->n_wait > 0) {
    cond->n_wait--;
    sem_post(&cond->sem);
  }
  mutex_unlock(&cond->lock);
  return 0;
}
```

Semáforos

- Implementar variáveis de condição usando semáforos e locks

cond_broadcast

```
int cond_broadcast(cond_t *cond) {
  mutex_lock(&cond->lock);
  while (cond->n_wait > 0) {
    cond->n_wait--;
    sem_post(&cond->sem);
  }
  mutex_unlock(&cond->lock);
  return 0;
}
```

Semáforos

- Se são equivalentes, como optar?
  - Locks e variáveis condição
    - Separação clara entre sincronização e exclusão mútua
    - Mais fácil de expressar condições complexas para bloqueio
  - Semáforos
    - Representação mais compacta para contadores