NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 10

- Implementar atomicidade
- Semântica das Variáveis de Condição
- Semáforos
  - exemplos sincronização, produtor-consumidor, leitores-escritores

- OSTEP: cap. 28.6-28.14, cap. 30, 31.1 – 31.5

2

Como implementar a atomicidade?

- Porquê mutex, VC e outros mecanismos em vez de um simples ciclo em espera?

```
flag=0  // início

lock( f ) {
      while ( flag == 1 )
            ;
      flag = 1          // não funciona

unlock( f ) {
      flag = 0
}
```

- Usando instruções especiais, exemplo, test-and-set.
  - *spin lock*:

```
flag=0  // início

lock( f ) {
      while ( test-and-set( f, 1 ) == 1 )
            ;

unlock( f ) {
      flag = 0
}
```


Crítica à Espera Ativa

- Porquê mutex, VC e outros mecanismos em vez de um ciclo em espera?
- Enquanto um recurso estiver ocupado
  - o CPU fica mais ocupado desnecessariamente → mau desempenho/*performance*
  - Não garante *fairness* ou ausência de *starvation*
- Alternativa:
  - dar a vez aos outros ou bloquear, mantendo um fila de espera
  - ir executando o que está a usar o recurso e outros threads/processos
  - ir desbloqueando da fila

E no SO?

- Como podem algumas operações ser indivisíveis?
- Num monoprocessador é simples: quando o código do SO executa não executa mais nada
  - mesmo as interrupções podem ser desligadas se necessário
- num multiprocessador, se podem existir chamadas concorrentes ao SO, há que garantir a exclusão mútua nas regiões críticas
  - p.ex.: por algoritmos e instruções específicas do CPU (Test-and-Set)
  - exemplo com *yield* (soluções melhores bloqueiam o kernel-thread):

```
flag=0  // início

lock( f ) {
      while ( test-and-set( f, 1 ) == 1 )
            yield()
}

unlock( f ) {
      flag = 0
}
```


Soluções com bloqueio

- Os mutex e VC representam os recursos e condições pretendidas
  - Só usar o recurso se obtiver o mutex ou a condição respetiva
- O lock de mutex ou wait numa VC, podem bloquear o thread!
  - cada objeto mutex e VC tem associada um fila de espera (ver OSTEP 28.13, 28.14)
  - o processo/thread pode passar a BLOCKED para aguardar e SO retira-lhe o CPU
- Quando o recurso é libertado, há que desbloquear um dos threads/processos dessa fila, se os houver
  - Pode ter politicas de ordenação para garantir justiça e evitar starvation
  - Este tipo de controlo é, normalmente, da responsabilidade do SO
- Outro mecanismo, desta classe de soluções: Semáforo de Sincronização

Implementação de Var. de Condição

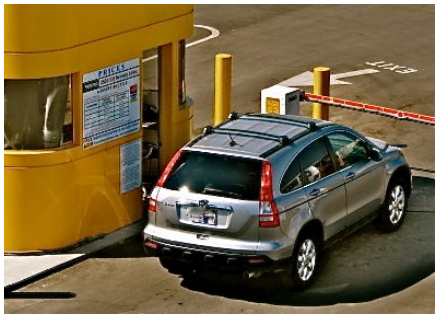
- Semântica Hoare:
  - o *signal* desbloqueia um só *thread*
  - obriga a garantir a passagem para o *thread* sinalizado sem que outro lhe passe à frente
- Semântica Mesa:
  - o *signal* desbloqueia um *thread*/processo pondo-o READY
  - este vai ter de competir com os outros para voltar a obter o *mutex*
  - pode levar a que mais de um *thread* vá testar a condição (equivale a mais do que um desbloqueado)
- Mesa mais fácil de implementar e seguida pelos Pthreads e outros
  - usada num *while* para voltar a testar e **confirmar** a condição

Cuidados

- Um sistema concorrente, sem estes mecanismos ou usando mal estes mecanismos
  - pode produzir resultados errados
  - corromper as estruturas de dados
  - introduzir injustiças entre processos ou threads (**fairness**)
  - impedir alguns de executar (**starvation**)
  - levar a bloqueios do sistema (**deadlock**)

Recursos com capacidade > 1

- Não queremos exclusão mútua num sistema com capacidade para atender mais *threads* ou processos:



Semáforos

- Os mutex são um caso particular dos semáforos antes propostos por Dijkstra
- Semáforo: tipo abstrato com um contador >= 0 e operações de incremento e decremento:
  - originalmente P() e V(), hoje chamadas por wait() e signal()/post()
- O decremento de um semáforo a zero bloqueia o processo ou *thread*, até que alguém o incremente

Implementação de Semáforos

- Cada Semáforo (S) é representado por:
  - uma variável inteira (*S.valor*)
  - uma fila de processos que esperam pelo semáforo *(S.fila)*
- É, tipicamente, implementado pelo SO que garante a atomicidade das operações e que bloqueia os processos em espera:
  - se um P(*S*) deve bloquear, o processo passa a Bloqueado e fica na fila *S.fila*
  - um V(*S*), pode permitir que um outro processo seja Desbloqueado e saia da fila *S.fila*

Implementação de P

- Possível implementação, pelo SO, da chamada P(S) (no OSTEP têm outra):

```
P(S)  /* operação indivisível */
{       if (S.valor > 0)
                S.valor = S.valor-1;
        else {
                acrescentaProcesso(S.fila);
                bloqueiaProcesso();  // passa a BLOCKED
                /* chama o escalonador */
        }
}
```

Implementação de V

- Possível implementação, pelo SO, da chamada V(S) (no OSTEP têm outra):

```
V(S)  /* operação indivisível */
{       if ( vazia(S.fila) )
                S.valor = S.valor+1;
        else {
                p=retiraProcesso(S.fila);
                desbloqueiaProcesso(p); //passa a READY
        }
}
```


Resumindo

- Semáforos não estão limitados a exclusão mútua
  - se valor inicial do semáforo = 1, temos exclusão mútua
  - se valor inicial = 2, temos no máximo 2 threads na respectiva RC, etc
- O semáforo tem memória. É um contador.
- Permite notificação/sincronização (semelhante a variáveis condição)
- As implementações existentes funcionam entre threads e/ou entre processos

1- Partilha de dados em exclusão mútua

- Semáforo de exclusão mútua (valor inicial 1) ou um mutex:

init(exmut, 1)

. . .

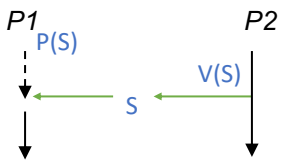
P(exmut); ← *pede entrada (decrementa exmut)*

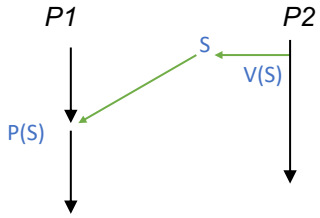
< região crítica >

V(exmut); ← *indica que acabou (incrementa exmut)*

2A- sincronização simples

- Um thread/processo aguarda por outro
  - exemplo: aguarda que uma var. partilhada seja alterada
- Solução com semáforo:
  - init(S, 0) (0 = não houve notificação)
  - Notificar: V(S) (o semáforo é incrementado)
  - Testar/aguardar notificação: P(S) (se ainda não houve notificação fica a aguardar)
  - o valor de S é o número de notificações por receber





2B- sincronização mútua

- **Rendezvous**: dois threads/processos aguardam um pelo outro
  - exemplo: ambos devem fazer algo ao "mesmo tempo"
- Para *n threads* → barreira
- Solução com semáforos:
  - Um semáforo para cada um notificar o outro, com valor inicial 0
  - Notificar: V(S) (o semáforo é incrementado)
  - Testar/aguardar notificação: P(S) (se ainda não houve notificação fica a aguardar)

Semáforo e mutex

- O *mutex* é como um semáforo binário (que só toma valores 0 e 1):
  - como um testemunho que pode ser detido por um *thread* de cada vez
- mutex equivale a um semáforo sempre iniciado a 1, mas...
  - Não funciona como contador. Só possui dois estados.
  - Normalmente só o thread que obteve o lock pode fazer unlock
  - *Unlock* de um *mutex* livre não faz nada ou pode dar erro

Semáforos Unix (Posix)

- Semáforos funcionam para threads e para processos!

```
#include <semaphore.h>
```

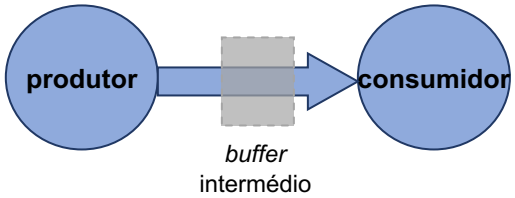
- Tipo: **`sem_t`**
- Operações:
- semáforos com nome: **`sem_open, sem_close`**
- semáforos em memória partilhada: **`sem_init`**
- incremento e decremento:

`sem_wait, sem_post`

  - *existe outra API mais antiga nos IPC Sys V (semget, semctl, semop)*

3 - Produtor(es) – Consumidor(es)

- Modelo comunicação por uma fila FIFO (limitada).
  - Caso de comunicação entre threads ou processos
- Consideremos Thread Produtor: produz dados que põe na fila
- Thread Consumidor: retira do buffer os dados anteriores



Solução com VC e mutex

```
por( T e) {
  lock(m);
  while (bufffull())
     wait(space, m);
  bufput( e );
  signal(elem);
  unlock(m);
}
```

```
T tirar() {
  lock(m);
  while (bufempty())
     wait(elem, m);
  T e = bufget();
  signal(space);
  unlock(m);
  return e;
}
```


Solução com semáforos

- os valores dos semáforos podem representar contadores (por exemplo, recursos disponíveis)
- No caso de podutores – consumidores:
  - *semáforo "vazias"* – indicando as posições vazias no buffer

    init(vazias, capacidade)
  - *semáforo "ocupadas"* – indicando as posições ocupadas

    init(ocupadas, 0)
  - buffput e bufget continuam a necessitar de exclusão mútua (só um de cada vez)

    init(mx, 1)

Solução com semáforos Posix

- Início (entre threads):

```
sem_init(&ex, 0, 1)
sem_init(&vazias, 0, capacidade)
sem_init(&ocupadas, 0, 0)
```

```
void por(int e){
   sem_wait(&vazias);
   sem_wait(&ex);
   bufPut( e );
   sem_post(&ex);
   sem_post(&ocupadas);
}
```

```
int tirar(){
   sem_wait(&ocupadas);
   sem_wait(&ex);
   int e = bufGet();
   sem_post(&ex);
   sem_post(&vazias);
   return e;
}
```

A ordem dos wait e post é relevante!
Exemplo: o que acontece se trocarem os dois primeiros wait?

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

23

4- Leitores - Escritores

- Exemplo: threads/processos concorrentes acedem a um recurso partilhado para ler e/ou escrever
  - Leitores: só consultam a informação
  - Escritores: atualizam/alteram a informação
  - **Não é necessário sempre exclusão mútua!**
- Que restrições de sincronização?
  - Não se pode escrever em concorrência com qualquer outro processo, seja Leitor ou Escritor
  - Podemos ter qualquer número de Leitores concorrentes entre si (mas só Leitores)
- *(cada thread/processo pode desempenhar ambos os papéis em alturas distintas do seu programa)*

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

24

Ler:

Pede-Leitura(); (1)

/* ler */

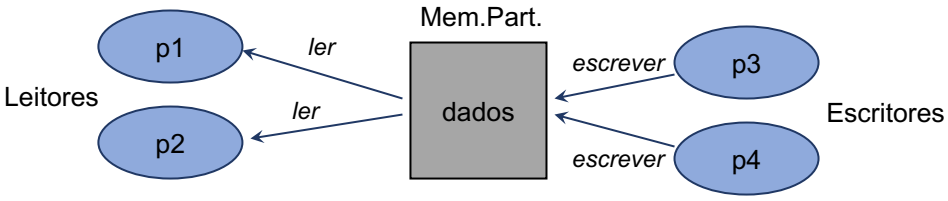
Termina-Leitura(); (2)

Escrever:

Pede-Escrita(); (3)

/* escrever */

Termina-Escrita(); (4)



Como programar as acções 1, 2, 3 e 4?

Esquema de uma solução

- Cada escritor necessita de acesso exclusivo → pede exclusão mútua
  - Nenhum processo, leitor ou escritor, pode entrar na sua região crítica
- O **primeiro** leitor obtém acesso para leitura → pede exclusão mútua
  - Não deixa "entrar" escritores
- Os restantes leitores verificam só se já existe algum leitor → não pedem exclusão mútua
- O **último** leitor liberta o acesso de leitura → liberta a exclusão mútua
  - O último leitor deixa que "entrem" escritores

Usando semáforos (incompleto)

Pede-Leitura:

```
NL=NL+1;
if (NL==1) wait(wrlock);   (1)

/* ler */
```

Pede-Escrita:

```
wait(wrlock);   (3)

/* escrever */
```

Termina-Leitura:

```
NL=NL-1;
if(NL==0) post(wrlock);   (2)
```

Termina-Escrita:

```
post(wrlock);   (4)
```

NL – contador do núm. de leitores
wrlock– lock aos escritores

Solução usando semáforos

Pede-Leitura:

```
wait(rdlock);
NL=NL+1;
if (NL==1) wait(wrlock);   (1)
post(rdlock);

/* ler */
```

Pede-Escrita:

```
wait(wrlock);   (3)

/* escrever */
```

Termina-Leitura:

```
wait(rdlock);
NL=NL-1;
if(NL==0) post(wrlock);   (2)
post(rdlock);
```

Termina-Escrita:

```
post(wrlock);   (4)
```

NL – contador do núm. de leitores
wrlock– lock aos escritores
rdlock – exclusão entre leitores (o teste e alteração da variável NL é uma região crítica)

Usando Semáforos

```
void readerLock(){//(1)
    sem_wait(rdlock);
    NL = NL+1;
    if ( NL==1 )
       sem_wait( wrlock );
    sem_post(rdlock);
}
 void readerUnLock(){//(2)
    sem_wait(rdlock);
    NL = NL-1;
    if ( NL==0 )
       sem_post( wrlock );
    sem_post(rdlock);
}
```

```
void writerLock(){//(3)
    sem_wait( wrlock );
}

void writerUnLock(){//(4)
    sem_post( wrlock );
}
```

NL – contador do núm. de leitores
wrlock– lock aos escritores
rdlock – exclusão entre leitores (o teste e alteração da variável NL é uma região crítica)

Tentando usar Pthreads-mutex

```
void readerLock(){//(1)
    pthread_mutex_lock(rdlock);
    NL = NL+1;
    if ( NL==1 )
       pthread_mutex_lock(wrlock);
    pthread_mutex_unlock(rdlock);
}
 void readerUnLock(){//(2)
    pthread_mutex_lock(rdlock);
    NL = NL-1;
    if ( NL==0 )
       pthread_mutex_unlock(wrlock);
    pthread_mutex_unlock(rdlock);
}
```

```
int writerLock(){//(3)
    pthread_mutex_lock(wrlock);
}


int writerUnLock(){//(4)
    pthread_mutex_unlock(wrlock);
}
```

A norma não prevê que unlock dum mutex de que não fez lock funcione! (ver pthread rwlock)

NL – contador do núm. de leitores
wrlock– lock aos escritores
rdlock – exclusão entre leitores (o teste e alteração da variável NL é uma região crítica)

Problemas destas soluções

- Enquanto existirem leitores a ler, outros leitores entram sempre
  - os leitores passam à frente dos escritores → **injusto**
  - os escritores podem não ter hipótese de escrever → ***starvation***
- solução?
  - quando chega um escritor, este impede que mais leitores entrem
  - exercício para casa!