Programação de Sistemas

Métodos de Exclusão Mútua

Introdução

- Existem três classes de métodos para assegurar a exclusão mútua de unidades de execução
  - Algoritmos de espera activa
    - Peterson
    - Lamport
  - Por hardware, com instruções especiais do processador
  - Por serviços do sistema operativo
    - Semáforos
    - Mutexes e Spinlocks
    - Barreiras
    - Mensagens
  - Por mecanismos de linguagens de programação
    - Monitores (Nota: não cobertos nesta disciplina)

Espera activa

- Nos algoritmos de espera activa ("busy waiting"), a unidade de execução a querer entrar interroga o valor de uma variável partilhada enquanto a RC estiver ocupada por outra unidade de execução.

  Vantagens:

  – Fáceis de implementar em qualquer máquina

  Inconvenientes:

  – São da competência do programador
  – A ocupação do CPU por processos à espera é um desperdício de recurso! Seria muito melhor bloquear os processos à espera.
- Algoritmos dividem-se de acordo com o número de unidades de execução concorrentes
  – Peterson, para N=2
  – Lamport (ou algoritmo da padaria), para N>2

Espera activa por Peterson (1)

- Válido apenas para 2 unidades de execução

```
#define N 2  /* número de unidades de execução */
typedef enum {FALSE,TRUE} Bool;

/* Variáveis partilhadas */
int turn;                    /* vez de quem entra na RC (0 ou 1) */
Bool flag[N];                /* flag[i]=TRUE: i está pronto a entrar na RC,
                                inicialmente flag[0]=flag[1]=FALSE; */
void enter_region(int p){
  int other=1-p;    /* número do outro processo */
  flag[p]=TRUE;     /* afirma que está interessado */
  turn=p;
  while(turn==p && flag[other]==TRUE);}
```

Espera activa por Peterson (2)

```
void leave_region(int p) {
  flag[p]=FALSE; /* permite entrada do outro processo */
}
```

I. Verificação de exclusão mútua

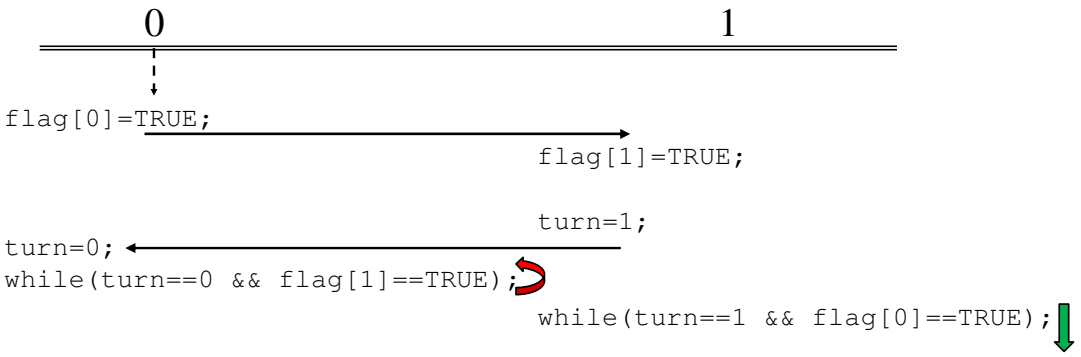
Um processo, por exemplo P1, só entra na RC se

- O outro processo não quiser (`flag[0]==FALSE`) e
- Se for a sua vez (`turn==1`)

Mesmo que ambos os processos executem as intruções `flag[0]=TRUE` e `flag[1]=TRUE`, só um dos processos entra porque `turn` só pode ter um valor (o último dos processos a executar a instrução `turn=p`).

Espera activa por Peterson (3)

Vejamos a hipótese de entrefolhagem que potencia corrida



O facto de flag [1] ter sido colocado a TRUE antes da atribuição `turn=1` faz com que

- A atribuição posterior `turn=0` não é suficiente para o processo 0 sair do while.
- A atribuição posterior `turn=0` leva o processo 1 a não ficar preso no while.

Espera activa por Peterson (4)

Nota: a instrução `flag[p]=TRUE` é transcrita por

```
MOV    %EAX,1      ; TRUE representado por 1
LEA    %EBX, flag
MOV    %ECX,p
MOV    [%EBX+%ECX],%EAX
```

a região crítica é formada apenas pela última instrução `MOV[%EBX+%ECX],%EAX` que é atómica.

II. Progresso e espera limitada

Uma unidade de execução espera, no máximo, que a outra unidade saia da RC: assim que o fizer, essa unidade entra na RC.

Espera activa por Lamport (1)

- Válida para qualquer número de unidades de execução.
- Antes de entrar na RC, a unidade de execução recebe uma ficha numerada (motivo porque este algoritmo é também designado por padaria)
- Entra na RC a unidade de execução com a ficha de número mais baixo.
- Ao contrário da realidade, várias unidades de execução podem receber o mesmo número de bilhete. Solução: desempatar pelo número do processo (esse sim, o sistema operativo garante ser único).

  Nota: `(a,b)<(c,d)` se `a<c` ou `(a==c` e `b<d)`

Espera activa por Lamport (2)

```
#define N …   /* número de unidades de execução (qualquer>2) */

/* Variáveis partilhadas */
Bool choosing[N];        /* anúncio de intenção em recolher bilhete */
int numb[N];             /* número de bilhete atribuído aos processos */

void initialize() {
  int i;
  for(i=0;i<N;i++) {
     choosign[i]=FALSE;
     numb[i]=0;}
}
```


Espera activa por Lamport (3)

```
void enter_region(int p) {
  int i;
  choosing[p]=TRUE;              /* anuncia que vai recolher bilhete */
  numb[p]=max(numb[0],…,numb[N-1])+1;    /* recolhe bilhete */
  choosing[p]=FALSE;
  for(i=0;i<N;++i) {
    while (choosing[i]);       /* espera outros recolham o bilhete */
        /* espera enquanto alguem está na RC:
             – Tem bilhete,
             – Tem preferência de acesso */
    while (numb [i]!=0 &&
              (numb [i],i)<(numb[p],p]) ) ;
  }
}
```

Espera activa por Lamport (4)

```
void leave_region(int p) {
    /* despeja o bilhete (no papelão, para reciclagem ;-) */
  numb[p]=0;}
```

I. Verificação de exclusão mútua

- Pior caso quando várias unidades de execução recolhem o mesmo número (lembrar que o processo pode ser substituído entre o cálculo da expressão e atribuição `numb[p]=max(numb[0],…,numb[N-1])+1`). No entanto, os processos que cheguem depois recebem números superiores.
- Se um processo k se encontrar na RC e p pretende entrar, então
  - numb[k]!=0
    (lembrar que apenas na região de saída numb[k] volta a 0)
  - (numb[k],k) é menor que (numb[p],p)

Logo, apenas um processo pode estar na RC.

Espera activa por Lamport (5)

II. Progresso

- Quando um processo sai da RC, na próxima vez que pretender entrar recebe um bilhete de número superior a todos os processo à espera.

III. Espera limitada

- Um processo à espera apenas tem que esperar que os restantes processos de número de bilhete inferior utilizem a RC. Sendo o número de processos limitado, a espera é limitada (se nenhum processo bloquear indefinidamente na RC).

Para além da ocupação do CPU enquanto está à espera de entrada na RC, o algoritmo da padaria tem o inconveniente de exigir um número sempre crescente de bilhetes e os inteiros são limitados. Uma possível solução é adicionar o tempo.

Soluções por hardware (1)

A. Sistemas uniprocessador
   - Uma vez que os processos são alternados por interrupções, basta inibir as interrupções nas RC.
   - No Pentium, a inibição de interrupções feita pela instrução `CLI` (CLear Interrupt flag)
   - No Pentium, a autorização de interrupções feita pela instrução `STI` (SeT Interrupt flag)

   Inconvenientes:
   - Dão poder ao programador para interferir com o sistema operativo.
   - Impraticável para computadores com vários processadores.
   - Se o processo bloquear dentro da RC, todo sistema fica bloqueado.

Soluções por hardware (2)

B. Sistemas multiprocessador
   - Necessárias instruções especiais que permirem testar e modificar uma posição de memória de forma atómica (i.e., sem interrupções).
   - Instruções de teste e modificação atómica de memória:
     - Test and Set
     - Swap

   Vantagens:
   - Rápido

   Inconvenientes:
   - Não garantem a espera limitada.
   - Exige ocupação do CPU por processos à espera.

Solução por hardware: TSL (1)

Test and Set: instrução descrita por

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv; }
```

- Uma variável booleana `lock`, inicializada a `FALSE`, é partilhada por todas as unidades de execução.

```
do {
    while ( TestAndSet(&lock) ) ; /* RE */
    /* RC */
    lock = FALSE; /* RS */
    /* RR */
} while (TRUE);
```


Solução por hardware: TSL (2)

I. Verificação de exclusão mútua
   - `lock` possui dois valores, pelo que apenas uma unidade de execução pode executar a RC.
     - Se for `FALSE`, a instrução retorna `FALSE` (a unidade de execução entra na RC e altera atomicamente `lock` para `TRUE`).
     - Se for `TRUE`, a instrução retorna o mesmo valor e a unidade de execução mantém-se na RE.

II. Progresso, se as unidades de execução na RR não alterarem o `lock`.

III. Espera limitada: **não garantida**, por depender da forma como o sistema operativo escala as unidades de execução (duas unidades podem ocupar alternadamente a RC, deixando uma terceira eternamente à espera).

Solução por hardware: Swap (1)

Swap: instrução descrita por

```
void Swap (boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp; }
```

- Uma variável booleana `lock`, inicializada a `FALSE`, é partilhada por todas as unidades de execução.
- Cada unidade de execução possui uma variável local `key`.

```
do {
  key = TRUE;
  while (key==TRUE) Swap (&lock, &key ); /* RE */
  /* RC */
  lock = FALSE; /* RS */
  /* RR */
} while (TRUE);
```


Solução por hardware: Swap (2)

I. Verificação de exclusão mútua
   - `lock` possui dois valores, pelo que apenas uma unidade de execução pode executar a RC.
     - Se for `FALSE`, Swap coloca `key` a `FALSE` e o teste while termina.
     - Se for `TRUE`, Swap mantém `key` a `TRUE` e o teste while é satisfeito, mantendo-se o ciclo.

I.II. Progresso e Espera limitada no Swap justificadas na mesma forma que para Test and Set.

- A Intel definiu no 486 a instrução `CMPXCHG dest,src` que atomicamente
  1. Compara acumulador com destino
  2. Se iguais, dest ← src. Se diferentes, acumulador ← destino

Semáforos - introdução (1)

[Def] Semáforo: abstracção de um contador e de uma lista de descritores de processos, usado para sincronização.

- Propostos por Dijsktra, em 1965
- Não requerem espera activa
- Especificados pelo POSIX:SEM. Os SO modernos (todas as versões do Unix, Windows NT/XP/Vista) disponibilizam semáforos

- As funções do POSIX:SEM indicam resultado no valor de retorno
  - Em caso de sucesso, o valor de retorno é 0.
  - Em caso de falha, o valor de retorno é -1. O código da causa de erro é afixado na variável de ambiente `errono`.
  - <semaphore.h> lista as causas de erro, que podem ser impressas no stderr pela função `perror(const char*)`.

Semáforos – introdução (2)

- O POSIX:SEM diferencia dois tipos de semáforos:
  - Anónimos (“unamed”), residentes em memória partilhada pelos processos. Neste caso, o semáforo é uma localização.
  - Identificados (“named”) por nome IPC. Neste caso, o semáforo é uma conexão entre sistemas distintos.
- Os semáforos podem igualmente ser classificados pelo número máximo de processos N que partilham o recurso. Se N=1, diz que o semáforo é binário.

Nota: O POSIX:SEM não descrimina o tipo sem_t. Nesta disciplina usamos a seguinte definição

```
typedef struct{
   unsigned counter;
   processList *queue;} sem_t;
```

Semáforos - introdução (3)

- `S.counter` determina quantos processos podem entrar na zona crítica sem bloquear.
  Se `S.counter` for sempre limitado aos valores 0 e 1, a exclusão mútua é assegurada.
- O comprimento da lista `S.queue` determina o número de processos bloqueados à espera de entrada na zona crítica.

Nota 1: no Linux, o editor de ligações deve indicar o arquivo realtime (librt.a), através da directiva `-lrt`.

Nota 2: as funções de gestão dos semáforos são listadas no `<semaphore.h>`.

Semáforos - introdução (4)

- Primitivas sobre um semáforo S:
  - Wait(S), down(S), ou P(S)

```
if (S.counter>0) S.counter--;
else {
   /* insere S na fila */
   Block(S); }
```

  - Signal(S), up(S), ou V(S)

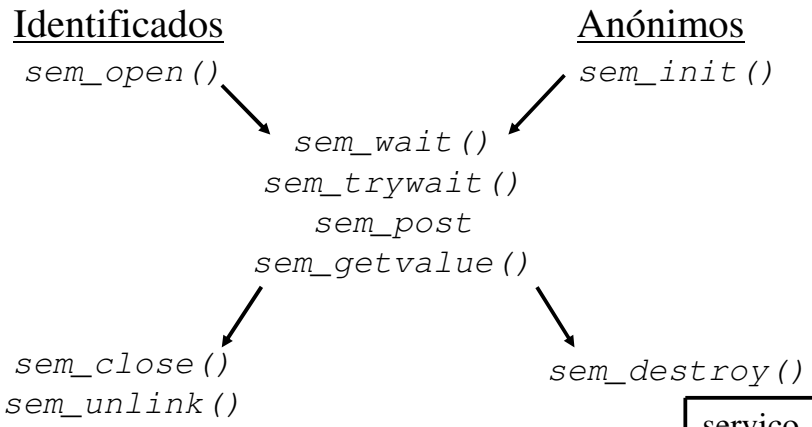
```
if (S.queue!=NULL) WakeUp(S);
else S.counter++;
```

- As primitivas podem ser implementadas por inibição de interrupções ou por instruções test-and-set.

Nota: P(S) e V(S) provêm das palavras holandesas *prolaag*-tentar decrementar e *verhoog*-incrementar.

Semáforos - introdução (5)

Funções disponíveis aos dois tipos de semáforos:



serviço	POSIX
up	sem_post
down	sem_wait

Semáforos - introdução (6)

- Na maioria dos casos, o programador usa o valor 0 ou 1 par inicialização de um semáforo.
  `S.counter=1;`
- Quando o valor de inicialização é 0, os processos que escrevem e lêm o semáforo sincronizam-se totalmente:
  - O primeiro processo a efectuar a operação Wait/Signal fica bloqueado até o outro processo efectuar a operação complementar.
  - Sincronização de processos por semáforo de contador vazio é denominada “rendez-vouz”.

Semáforos anónimos - definição (1)

- Um semáforo anónimo é uma localização de tipo `sem_t`

```
#include <semaphore.h>
sem_t sem;
```

- Um semáforo tem de ser inicializado antes de ser usado

POSIX:SEM `int sem_init(sem_t *,int,unsigned);`

  - 1º parâmetro: endereço da localização do semáforo.
  - 2º parâmetro: valor não negativo (0 indica que apenas pode ser usado pelos fios de execução do processo que inicializa o semáforo, positivo pode ser usado por qualquer processo com acesso à localização).
  - 3º parâmetro: valor de inicialização.

Nota: O Linux não suporta semáforos partilhados por processos, logo o 2º parâmetro é sempre 0.

Semáforos anónimos - definição (2)

- Um semáforo que deixe de ser útil deve ser eliminado pela função

POSIX:SEM `int sem_destroy(sem_t *);`

```
sem_t semaforo;

if (sem_init(&semaforo,0,1)==-1)
   perror("Falha na inicializacao");

if (sem_destroy(&semaforo)==-1)
   perror("Falha na eliminacao");
```

Semáforos identificados-definição (1)

- Um semáforo identificado é uma conexão que permite sincronizar processos sem memória partilhada, possuindo
  - Identificador: cadeia de caracteres na forma `/name`.
    Nota: os semáforo identificados são instalados em `/dev/shm`, com o identificador sem.name
  - ID utilizador,
  - ID grupo, e
  - permissões
- Um semáforo tem de ser aberto

POSIX:SEM `sem_t *sem_open(const char *,int,…);`

  - 1º parâmetro: identificador da conexão.
  - 2º parâmetro: bandeiras, `O_CREAT`-a conexão é criada se não existir, `O_EXCL`-com `O_CREAT` a função falha se a conexão existir.

Semáforos identificados-definição (2)

- Se o 2º parâmetro contiver o bit `O_CREAT` a 1, devem ser indicados mais dois parâmetros de modos:
  3. `mode_t`, determinado as pemissões (`S_IRUSR`, `S_IWUSR`, `S_IRGRP` ou `S_IROTH`).
  4. unsigned, especificando o valor inicial do semáforo.
- Se o 2º parâmetro contiver os bits `O_CREAT` e `O_EXCL` a 1, a função devolve erro se o semáforo já existir.
- Se o semáforo já existe e 2º parâmetro contiver o bit `O_CREAT` mas não `O_EXCL`, a função ignora os 3º e 4º parâmetros.

Semáforos identificados-definição (2)

- Quando um semáforo deixa de ser necessário a um processo, ele deve ser fechado.

  POSIX:SEM `int sem_close(sem_t *);`

- O último processo que o semáforo deve eliminá-lo pela função.

  POSIX:SEM `int sem_unlink(const char *);`

  – Se houver um processo que mantenha o semáforo aberto, o efeito de `sem_unlink` é suspenso até o último processo fechar o semáforo.

Semáforos - operações (1)

- A primitiva P(S) é implementada, nos semáforos anónimos e identificados, pela função
  POSIX:SEM `int sem_wait(sem_t *);`

  – Se o contador do semáforo estiver a zero
    - Se for um processo a executar P(S), ele fica bloqueado.
    - Se for um fio de execução a executar P(S), ele fica bloqueado. O que sucede aos restantes fios de execução depende do conhecimento que o gestor de fios de execução tiver: se o gestor residir no núcleo (LKP), os restantes fios de execução não são bloqueados.
      Nota: o programador deve confirmar o modelo de implementação dos fios de execução.

Semáforos - operações (2)

- A primitiva V(S) é implementada , nos semáforos anónimos e identificados, pela função
  POSIX:SEM `int sem_post(sem_t *);`

- O semáforo `sem`, inicializado a `1`, é partilhado por todas as unidades de execução. RC garantida pelo seguinte código:

```
do {
    sem_wait( sem ); /* RE */
    /* RC */
    sem_post( sem ); /* RS */
    /* RR */
} while (TRUE);
```


Semáforos - operações (3)

I. Verificação de exclusão mútua
   - Se o contador do semáforo tiver valor 1, nenhuma unidade de execução se encontra na RC. Quando uma unidade de execução se encontrar na RE, entra e o contador é decrementado para 0.
   - Se o contador do semáforo tiver valor 0, uma unidade de execução que queira entrar na RC fica bloqueada até que a unidade de execução dentro da RC saia.

II. Progresso, se as unidades de execução na RR não alterarem o semáforo.

II. Espera limitada: **só garantida** se a o armazém de processos à espera for organizada como fila.

Semáforos - operações (4)

- Em ambos semáforos, anónimos e identificados, o processo pode alterar o valor do contador pela função

  POSIX:SEM
  ```
  int sem_getvalue(
                  sem_t *, int *);
  ```

  – A localização onde é colocado o valor do contador é indicada pelo 2º parâmetro.
  – Se na altura da chamada o semáforo se encontrar fechado, o POSIX admite duas implementações para a forma de alterar a localização indicada pelo 2º parâmetro:
    - para um valor negativo igual ao número de processos bloqueados no semáforo.
    - para 0 (implementação adoptada pelo Linux).

Balanço dos semáforos

Vantagens dos semáforos:

– Programador não se preocupa com implementação das operações P(S) e V(S)
– Na especificação POSIX, podem ser sincronizar processos com e sem memória partilhada.

Inconvenientes dos semáforos:

– Obriga programador a inserir explicitamente instruções `sem_wait` e `sem_post`.
  Solução: usar monitores disponíveis em linguagens de programação (ex: Ada,CHILL).
– Má programação (ex:: não executar `sem_post` na RS) leva a resultados inesperados-por exemplo, bloqueio.

Mutexes - definição (1)

[Def] Mutex: variável que pode ter apenas dois valores, trancado ("locked") e destrancado ("unlocked") e de uma lista de descritores de fios de execução.

- Introduzidos no Kernel 2.6.16
- Mais leves (no x86, o struct semaphore ocupa 28B e o struct mutex ocupa 16B) e mais rápidos.
- Um mutex trancado pertence apenas a um único fio de execução, que é o único a poder destrancar o mutex.
- Os fios de execução que pretendam trancar um mutex que já se encontra trancado são guardados numa lista associada ao mutex (ordem de inserção depende da política de escalonamento dos fios de execução).
- Especificados pelo POSIX:THR.

Nota: mutex ::= MUTual EXclusion

Mutexes - definição (2)

- Um mutex é usado com a seguinte sequência de etapas:
  1. Criação e inicialização de um mutex.
  2. Vários fios de execução, na Região de Entrada, tentam trancar o mutex.
     Só um deles consegue, passando a ser o dono.
  3. O dono do mutex executa a Região Crítica.
  4. O dono do mutex entra na Região de Saída, destrancando o mutex, e passa para a Região Restante.
  5. Outro mutex na Região de Entrada tranca novamente o mutex, e executa os passos 3-4.
  6. …
  7. Finalmente, o mutex é eliminado.

Mutexes - definição (3)

A. Definição: localização de tipo `pthread_mutex_t`

```
#include <pthread.h>
pthread_mutex_t mux;
```

B. Inicialização: Um mutex tem de ser inicializado antes de ser usado

– Dinâmica:

POSIX:THR `int pthread_mutex_init ( pthread_mutex_t *, const pthread_mutextattr_t);`

- 1º parâmetro: endereço da localização do mutex.
- 2º parâmetro: atributos (por omissão, usar `NULL`)
- Em caso de falha, `pthread_mutex_init` retorna -1

– Se a localização for estática a inicialização pode ser feita atribuindo o valor `mux=PTHREAD_MUTEX_INITIALIZER;`
A inicialização estática tem como vantagens (1) ser mais eficiente (2) garante ser feita antes do arranque do fio de execução.

Mutexes - definição (4)

C. Eliminação: Um mutex é eliminado pela função

POSIX:THR `int pthread_mutex_destroy( pthread_mutex_t *);`

- Em caso de sucesso retorna 0.
- As etapas 1 e 7 da vida de um mutex executadas pelo seguinte código

```
int error;
thread_mutex_t mux;

if (error=pthread_mutex_init(&mux,NULL))
   fprintf(stderr,"Falha por %s\n",strerror(error));
/* ... */
if (error=pthread_mutex_destroy(&mux))
   fprintf(stderr,"Falha por %s\n",strerror(error));
```

Mutexes - operações (1)

D. Aquisição: Aquisição de um mutex, com o respectivo trancar, é efectuada por duas funções

POSIX:THR
```
int pthread_mutex_lock(
      pthread_mutex_t *);
int pthread_mutex_trylock(
      pthread_mutex_t *);
```

- `pthread_mutex_lock` bloqueia até o mutex se encontrar disponível.
- `pthread_mutex_trylock` retorna imediatamente.
- Ambas as funções retornam 0, em caso de sucesso. A falha no trancar do mutex pela função `pthread_mutex_trylock` é indicado pelo valor `EBUSY` na variável `error`.

Nota: os mutexes devem ser trancados pelo fio de execução no mais curto intervalo de tempo possível.

Mutexes - operações (2)

E. Libertação: A primitiva de destrancar um mutex é implementada pela função

POSIX:THR
```
int pthread_mutex_unlock(
      pthread_mutex_t *);
```

- RC garantida pelo seguinte código

```
thread_mutex_t mux=PTHREAD_MUTEX_INITIALZER;
do{
   pthread_mutex_lock( &mux ); /* RE */
      /* RC */
   pthread_mutex_unlock( &mux ); /* RS */
   /* RR */
while(TRUE);
```

Mutexes sobre condições - definição (1)

Curiosidade, para avaliação avançada apenas

- Pode haver interesse sincronizar o acesso a dados com base em variáveis que satisfaçam determinadas condições (ex: um sensor de pressão, ou de temperatura, atingir um valor limiar-"threshold")
- A espera activa-"busy waiting" consome muito CPU

```
while(1) {
   pthread_mutex_lock(&mux);
   if (var==threshlold) break;
   pthread_mutex_unlock(&mux); }
```

- POSIX disponibiliza o mecanismo de associação de mutexes a variáveis de condições.

Mutexes sobre condições - definição (2)

Curiosidade, para avaliação avançada apenas

- Um mutex condicionado é uma localização de tipo `pthread_cont_t`

```
#include <pthread.h>
pthread_cont_t cont;
```

- O mutex condicionado tem de ser inicializado antes de ser usado

POSIX:THR

```
int pthread_cond_init(
   pthread_cond_t *,
   const pthread_condattr_t *);
```

  - 1º parâmetro: endereço da localização do mutex condicionada.
  - 2º parâmetro: atributos (por omissão, usar `NULL`)
- A inicialização estática pode ser feita atribuindo um valor `cont=PTHREAD_COND_INITIALZER;`

Mutexes sobre condições - definição (3)

Curiosidade, para avaliação avançada apenas

- Um mutex condicionado é eliminada pela função

  POSIX:THR
  ```
  int pthread_cond_destroy(
        pthread_cond_t *);
  ```

- Um mutex condicionado é usado na seguinte sequência de etapas:
  1. Criar e inicializar o mutex condicionado.
  2. Se a condição não for satisfeita, o fio de execução bloqueia sobre a condição.
  3. Outro fio de execução altera variáveis envolvidas na condição e assinala os fios de execução bloqueados sobre variáveis condicionadas (individualmente ou por difusão).
  4. Fio de execução assinalado volta ao ponto 2.
  5. Eliminar o mutex condicionado.

Mutexes sobre condições – operações (1)

Curiosidade, para avaliação avançada apenas

- O bloqueio de um fio de execução sobre um mutex condicionado é feito por uma das funções:

  POSIX:THR
  ```
  int pthread_cond_wait(
        pthread_cont_t *,
        pthread_mutex_t *);
  int pthread_cond_timedwait(
        pthread_cont_t *,
        pthread_mutex_t *,
        const struct timespec*);
  ```

  - `pthread_cond_wait` bloqueia até outro fio de execução assinalar possível alteração da condição.
  - Na função `pthread_cond_timedwait` a espera é temporizada por uma função.

Mutexes sobre condições – operações (2)

Curiosidade, para avaliação avançada apenas

- 1º parâmetro: endereço da localização do mutex condicionado.
- 2º parâmetro: mutex que tranca a região crítica (é na RC que as variáveis de condição podem ser alteradas).
- 3º parâmetro: função de temporização da espera

Nota: as funções `pthread_cond_wait()` e `pthread_cond_timedwait()` só devem ser chamadas após ter sido executado `pthread_lock()`

Mutexes sobre condições – operações (3)

Curiosidade, para avaliação avançada apenas

- Outro fio de execução assinala alteração sobre variável condicionada pelas funções

POSIX:THR

```
int pthread_cond_signal(
      pthread_cont_t *);
int pthread_cond_broadcast(
      pthread_cont_t *);
```

- parâmetro: endereço da localização da variável condicionada.
- `pthread_cond_signal` destranca pelo menos um fio de execução bloqueado numa variável condicionada, `pthread_cond_broadcast` destranca todos os fios de execução bloqueados numa variável condicionada.

Mutexes sobre condições – exemplo (1)

Curiosidade, para avaliação avançada apenas

- Considere-se um simulador de um termómetro, que deve detectar valores superiores ao limiar TH=40ºC.
  - No exemplo, as temperaturas são geradas por um registo de deslocamento realimentado LFSR, definido pelo polinómio irredutível $x^{11}+x^2+1$.
  - O utilizado deve ser o valor de inicialização, IV, entre 0 e 2048.
  - Quando forem recolhidas IV amostras, o simulador termina.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#define TH 40

void *modify();
void *test();
```

```
int temperature,number,top;
unsigned int SR;

pthread_t mid,tid;

pthread_mutex_t cond_mut=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t
    count_lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cvar=PTHREAD_COND_INITIALIZER;
```


Mutexes sobre condições – exemplo (2)

Curiosidade, para avaliação avançada apenas

```
int main(int argc, char *argv[]) {
 if(argc!=2) {
   printf("&s numb,0<numb<2048\n",argv[0]);
   exit(1); }
 SR = atoi(argv[1]);
 if(SR<1 || SR>=2048) {
   printf("%s numb,0<numb<2048\n",argv[0]);
   exit(1); }
 temperature = 0;
 number = 0;
 top = SR;

 pthread_create(&mid,NULL,modify,NULL);
 pthread_create(&tid,NULL,test,NULL);
 pthread_join(tid,NULL);
 return 0; }
```

```
void LFSR() {
 /*gerador x^11+x^2+1*/
  SR = ( ( (
           ( SR>>10 )
         ^ ( SR>>1 )
        )
      & 0x01 )
      << 10)
      | (SR>>1);
}
```

Mutexes sobre condições – exemplo (3)

Curiosidade, para avaliação avançada apenas

```
void *modify() {
  pthread_mutex_lock(&count_lock);
  while(1) {
     LFSR();
     pthread_mutex_lock(&cond_mut);
     temperature = (TH+20)*SR/2048;
     number++;
     pthread_mutex_unlock(&cond_mut);
     if (temperature>=TH) {
        pthread_cond_signal(&cvar);
        pthread_mutex_lock(&count_lock); }
  }
  return NULL; }
```

Acorda thread bloqueada na condição

Fica à espera que temp seja restabelecida

Mutexes sobre condições – exemplo (4)

Curiosidade, para avaliação avançada apenas

```
void *test() {

  while(1) {
     pthread_mutex_lock(&cond_mut);
     while (temperature<TH) pthread_cond_wait(&cvar,&cond_mut);
     printf("Atingida temperatura %d na %d-sima vez\n",
            temperature, number);
     temperature = TH-2;
     pthread_mutex_unlock(&cond_mut);
     pthread_mutex_unlock(&count_lock);
     if (number>=top) {
        pthread_kill(mid,SIGKILL);
        break; }
  }
  return NULL; }
```

Thread bloqueada enquanto a condição for satisfeita

Mutexes sobre condições – exemplo (5)

Curiosidade, para avaliação avançada apenas

```
[rgc@asterix Termometro]$ CV 23
Atingida temperatura 46 na 5-sima vez
Atingida temperatura 53 na 6-sima vez
Atingida temperatura 56 na 7-sima vez
Atingida temperatura 58 na 8-sima vez
Atingida temperatura 59 na 9-sima vez
Atingida temperatura 59 na 10-sima vez
Atingida temperatura 49 na 21-sima vez
Atingida temperatura 54 na 22-sima vez
Atingida temperatura 57 na 23-sima vez
Killed
[rgc@asterix Termometro]$
```


Spinlocks (1)

Curiosidade, para avaliação avançada apenas

- O **Spinlock** é um tipo especial de mutex
  - se a variável estiver trancada, outro fio de execução não bloqueia e fica em ciclo ("spin"-movimento giratório) a tentar trancar a variável (abordagem designada por *espera activa*-"busy waiting").
  - O que se perde em tempo de CPU, ganha-se na salvaguada de contexto num mutex normal.
  - Normalmente usado no Kernel (fios de execução muito rápidos) ou em sistemas multiprocessador.

- Disponibilizado no pacote pthread

Spinlocks (2)

Curiosidade, para avaliação avançada apenas

- Tipo de dados `pthread_spinlock_t`
- Funções de gestão de spinlocks:
  A. Inicialização:
     `pthread_spin_int(pthread_spinlock_t *, int)`
     O 2º parâmetro deve ter um dos valores:
     A. `PTHREAD_PROCESS_SHARED`
     B. `PTHREAD_PROCESS_PRIVATE`
  B. Aquisição/Libertação região crítica:
     `pthread_spin_lock(pthread_spinlock_t *)`
     `pthread_spin_unlock(pthread_spinlock_t *)`
  C. Destruição:
     `pthread_spin_destroy(pthread_spinlock_t *)`

Spinlocks (3)

Curiosidade, para avaliação avançada apenas

- No 80386, ou posterior, o spinlock é implementado por

```
spin_lock:
MOV     EAX, 1
XCHG    EAX, [lock]
loop:
TEST    EAX,EAX
JNZ     loop
RET

spin_unlock:
MOV     EAX, 0
XCHG    EAX, [lock]
RET
```

Barreiras (1)

[Def] Barreira: mecanismo de sincronização de um número fixo de barreiras. Os fios de execução ficam bloqueados até atingir a quantidade indicada.

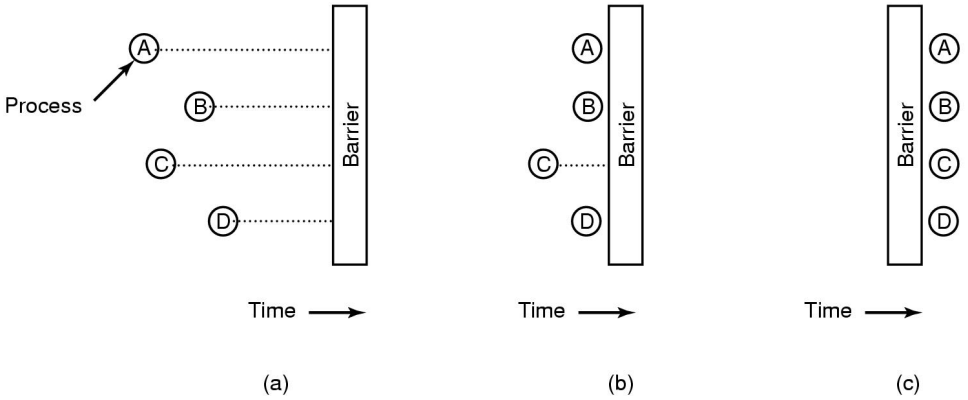


Figura 2-30, Modern Operating Systems

Barreiras (2)

- As barreiras usadas quando o processamento é feito por etapas, com a entrada na etapa seguinte feita após todos os fios de execução terem concluido a etapa anterior.
- Barreiras disponibilizadas no pacote *pthreads*.
- Tipo de dados `pthread_barrier_t`
- Funções de gestão de barreiras:

A. Inicialização:

```
pthread_barrier_init(
    pthread_barrier_t *,
    pthread_barrierattr_t *,
    unsigned int)
```

O 2º parâmetro identifica os atributes (por omissão, usar NULL).

O 3º parâmetro determina os fios de execução a sincronizar.

Barreiras (3)

B. Espera

```
int pthread_barrier_wait(
     pthread_barrier_t *)
```

Uma das threads recebe `PTHREAD_BARRIER_SERIAL_THREAD`, todas as outras recebem 0.

C. Destruição:

```
pthread_barrier_destroy(
    pthread_barrier_t *)
```

Exemplo: Lançar vários fios de execução, cada um incrementa um contador com a sua ordem. Os fios de execução sincrionizam numa barreira e imprimem o resultado final do contador.

Nota: alteração do contador a proteger por um mutex.

Barreiras (4)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct{
   int pos;
} arguments;

pthread_barrier_t barr;
pthread_mutex_t key;
unsigned counter, /* contador global */
         numbT;   /* numero de threads */
```

Barreiras (5)

```
void * entry_point(void *arg) {
   int rc;

   /* incrementa contador */
   pthread_mutex_lock( &key );
   counter += ((arguments *)arg)->pos;
   printf( "Inicio thread %lx com counter=%d\n",
      pthread_self(),counter );
   pthread_mutex_unlock( &key );

   /* ponto de sincronizacao */
   rc = pthread_barrier_wait( &barr );
   if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
      fprintf( stderr,"Impossivel esperar na barreira\n");
      exit(-1); }

   printf( "Thread %lx termina com contador=%d\n",
      pthread_self(),counter );
   pthread_exit( NULL ); }
```


Barreiras (6)

```
int main(int argc, char *argv[]) {
   pthread_t *thr;
   arguments *arg;
   int i;

   if( argc!=2 ) {
      fprintf( stderr," %s number\n",argv[0] );
      exit( -1 ); }
   numbT = atoi( argv[1] );
   thr = ( pthread_t* )malloc( numbT*sizeof(pthread_t) );

   /* Inicializacao da barreira */
   if( pthread_barrier_init( &barr,NULL,numbT ) ) {
      fprintf( stderr,"Impossivel criar barreira\n" );
      return -1; }

   /* Inicializacao de variaveis */
   pthread_mutex_init( &key,NULL );
   counter = 0;
```

Barreiras (7)

```
   /* Lanca threads */
   for( i=0;i<numbT;i++ ) {
     arg = (arguments *)malloc(sizeof(arguments));
     arg->pos = i;
     if( pthread_create( &(thr[i]),NULL,entry_point,(void *) arg) ) {
        fprintf( stderr,"Erro no lancamento de thread %d\n",i );
        exit( 20 ); }
   }

   /* Espera pela conclusao das threads */
   for( i = 0;i<numbT;++i ) {
      if( pthread_join(thr[i],NULL) ) {
         fprintf( stderr,"Nao pude juntar thread %d\n",i );
         exit( 21 ); }
   }

   pthread_barrier_destroy( &barr );
}
```


Barreiras (8)

```
[rgc@asterix Barrier]$ barrier 6
Inicio thread b7faeb90 com counter=0
Inicio thread b75adb90 com counter=1
Inicio thread b6bacb90 com counter=3
Inicio thread b61abb90 com counter=6
Inicio thread b57aab90 com counter=10
Inicio thread b4da9b90 com counter=15
Thread b4da9b90 termina com contador=15
Thread b75adb90 termina com contador=15
Thread b57aab90 termina com contador=15
Thread b61abb90 termina com contador=15
Thread b6bacb90 termina com contador=15
Thread b7faeb90 termina com contador=15
[rgc@asterix Barrier]$
```

Sincronização por gestor (1)

- A região crítica pode ser transferida para um processo dedicado, o gestor de recursos.
- Um processo que pretenda alterar dados críticos executa a seguinte sequência de passos:
  1. Gerar mensagem de pedido.
  2. Enviar mensagem para o gestor.
  3. Ficar, bloqueado, à espera da resposta do gestor.
  4. Acção dependente do resultado

Sincronização por gestor (2)

- O gestor possui o seguinte programa tipo:

```
/* Inicialização de variáveis */
while(1) {
    /* Ler, bloqueado, um pedido */
    /* Processar pedido */
    /* Preparar resposta */
    /* Enviar, ao processo que pediu alterações, a indicação do
    resultado do pedido */
}
```

Fechadura de registos (1)

Curiosidade, para avaliação avançada apenas

- Um ficheiro pode ser acedido em simultâneo por processos distintos (desde que possuam permissões).
- A sincronização é implementada, no Linux, por fechadora de uma secção do ficheiro ("Record locking") usando a função

POSIX:SEM

```
#include <unistd.h>
int fcntl(int, int, struck flock *);
```

  - 1º parâmetro: descritor do ficheiro.
  - 2º parâmetro – comando de manipulação.
  - 3º parâmetro: localização de uma estrutura onde são armazenados dados sobre a secção a trancar.

Fechadura de registos (2)

Curiosidade, para avaliação avançada apenas

- Comandos disponibilizados
  - `F_SETLK`: tranca fechadura, sem bloquear se não ficar trancado
  - `F_SETLKW`: tranca a secção do ficheiro (chamada bloqueante)
  - `F_GETLK`: identifica processo que trancou a secção do ficheiro.
- Secção do ficheiro definida na estrutura

```
struct flock {
   short l_type;    /* Tipo de tranca */
   short l_whence; /* Posição de início */
   off_t l_start;   /* Distância do início */
   off_t l_len;     /* Número de Bytes a trancar */
   pid_t l_pid;     /* PID do processo que tranca (apenas F_GETLK ) */
     ...
};
```

Fechadura de registos (3)

Curiosidade, para avaliação avançada apenas

- O campo `l_type` identifica o tipo de fechadura
  - `F_RDLCK`: leitura (vários processos podem partilhar a fechadura de leitura)
  - `F_WRLCK`: escrita (apenas um processo pode possuir a fechadura de escrita)
  - `F_UNLCK`: destranca a secção.
- O resultado dos comandos `F_SETLK` e depende das fechaduras aplicadas sobre a secção e o valor de `l_type`

Fechaduras aplicadas	l_type=F_RDLCK	l_type=F_WRLCK
Nenhuma	OK	OK
Uma, ou mais, RDLCK	OK	Recusado
Uma WRLCK	Recusado	Recusado

Fechadura de registos (4)

Curiosidade, para avaliação avançada apenas

[Exemplo] Programa lançado duas vezes, recebe do terminal dois caracteres: se forem letras minúsculas substituem os dois primeiros Bytes do mesmo ficheiro pela ordem indicada na linha de lançamento.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

void ERR( char *msg ) {
   printf( "%s\n",msg );
   _exit( 1 ); }
```

Fechadura de registos (5)

Curiosidade, para avaliação avançada apenas

```
void lockByte( int fd,off_t offset,char me ) {
   struct flock lock;

   lock.l_type = F_WRLCK; lock.l_start = offset;
   lock.l_whence = SEEK_SET; lock.l_len = 1;

   if( fcntl(fd,F_SETLK,&lock)<0 ) {
       printf( "%c(%d): Byte %d esta' trancado, vou ficar 'a espera\n",
               me,getpid(),offset );
       fcntl( fd,F_SETLKW,&lock ); }
   printf( "%c(%d): obtive tranca no Byte %d\n", me,getpid(),offset );}

void unlockByte( int fd,off_t offset,char me ) {
   struct flock lock;

   lock.l_type = F_UNLCK; lock.l_start = offset;
   lock.l_whence = SEEK_SET;  lock.l_len = 1;
   if( fcntl(fd,F_SETLK,&lock)<0 ) ERR( "Erro fcntl" );
   printf( "%c(%d): libertei tranca no Byte %d\n",me,getpid(),offset );
}
```


Fechadura de registos (6)

Curiosidade, para avaliação avançada apenas

```
main( int argc,char *argv[] ) {
   int fd, newIn;
   int i, out;
   off_t pos;
   char ch, extra, me;
   char msg[]=": Letra minuscula => ";

   /* Recolhe parametros */
   if( argc!=4 ) ERR( "testaAcesso fich {0|1} {A|B}" );
   newIn = dup( STDIN_FILENO ); close( STDIN_FILENO );
   pos = (off_t)atoi( argv[2] );
   if( pos<0 || pos>1 ) ERR( "testaAcesso fich {0|1} {A|B}");
   me = argv[3][0];

   printf( "Teste de tranca em ficheiro\n" );
   if( (fd=open(argv[1],O_RDWR)<0) ) ERR( "Erro abertura ficheiro" );
```

Fechadura de registos (7)

Curiosidade, para avaliação avançada apenas

```
 for( i=1;i<=2;i++ ) {

      lockByte( fd,pos,me );

      /* obtem caractere */
      write( STDOUT_FILENO,&me,1 );
      write( STDOUT_FILENO,&msg,strlen(msg) );
      read( newIn,&ch,1 );
      do read( newIn,&extra,1 ); while( extra!='\n' );

      if( ch>='a' && ch<='z' ) { /* substitui, se for letra minuscula
*/
         lseek( fd,pos,SEEK_SET );
         out = write( fd,&ch,1 ); }
      unlockByte( fd,pos,me );

      /* identifica nova posicao */
      pos = (pos+1)%2; }

   close( fd );
  _exit( 0 );}
```


Fechadura de registos (8)

```
[root@asterix AcessoFich]# cat >tempLock (1)
abcdefgh
[root@asterix AcessoFich]# cat tempLock (5)
wbcdefgh
[root@asterix AcessoFich]# cat tempLock (7)
wzcdefgh
[root@asterix AcessoFich]# cat tempLock (10)
wqcdefgh
[root@asterix AcessoFich]#
```

Curiosidade, para avaliação avançada apenas

```
[rgc@asterix tmp]$ testaAcesso /home/ec-ps/public_html/Exemplos/Sincronizacao/AcessoFich/tempLock 0 A (2)
Teste de tranca em ficheiro
A(17037): obtive tranca no Byte 0
A: Letra minuscula => w (4)
A(17037): libertei tranca no Byte 0
A(17037): Byte 1 esta' trancado, vou ficar 'a espera
A: Letra minuscula => q (9)
A(17037): libertei tranca no Byte 1
[rgc@asterix tmp]$
```

```
[amg@asterix ec-ps]$ testaAcesso /home/ec-ps/public_html/Exemplos/Sincronizacao/AcessoFich/tempLock 1 B (3)
Teste de tranca em ficheiro
B(17079): obtive tranca no Byte 1
B: Letra minuscula => z (6)
B(17079): libertei tranca no Byte 1
B(17079): obtive tranca no Byte 0
B: Letra minuscula => 4 (8)
B(17079): libertei tranca no Byte 0
[amg@asterix ec-ps]$
```

Corridas a nível de *design*

- Os métodos de exclusão mútua podem não evitar corridas, devido a problemas de *design*.
- Exemplo: seja a variável `count` alterada de forma distinta em diversas regiões críticas

```
int count=10;
thread1( ) {
   lock(mux);
   count++;
   unlock(mux);}
```

```
thread2( ) {
   lock(mux);
   count--;
   unlock(mux);}
```

```
thread3( ) {
   lock(mux);
   printf("%d",count);
   unlock(mux); }
```

O resultado depende da ordem de execução:

10: thread3-> thread1-> thread2

9: thread2-> thread3-> thread1

11: thread1-> thread3-> thread2

- A detecção de corridas é um problema NP-hard.