

# Programação de Sistemas

Exclusão Mútua

# Introdução

- Existem quatro classes de métodos para assegurar a exclusão mútua de tarefas:
- 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

# Espera activa

- Nos algoritmos de espera activa (“busy waiting”), a tarefa que pretende entrar interroga o valor de uma variável partilhada enquanto a RC estiver ocupada por outra tarefa.
- 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 tarefas concorrentes
  - Peterson, para  $N=2$
  - Lamport (ou algoritmo da padaria), para  $N \geq 2$

# Espera activa por Peterson

- Válido apenas para 2 tarefas
  - #define N 2 /\* número de tarefas \*/
  - 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);
  - }
  - void leave\_region(int p) {
  - flag[p]=FALSE; /\* permite entrada do outro processo \*/
  - }

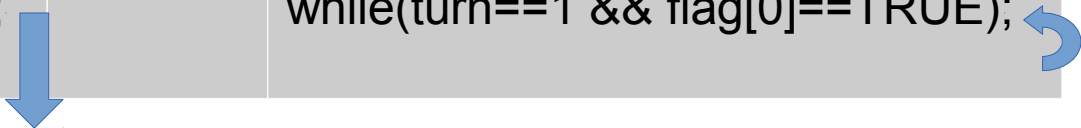
# Espera activa por Peterson

- Verificação de exclusão mútua
- Um processo, por exemplo P1, só entra na RC se
  - O outro processo não quiser ( $\text{flag}[0] == \text{FALSE}$ ) e
  - Se for a sua vez ( $\text{turn} == 1$ )
- Mesmo que ambos os processos executem as instruções  $\text{flag}[0] = \text{TRUE}$  e  $\text{flag}[1] = \text{TRUE}$ ,
  - só um dos processos entra porque  $\text{turn}$  só pode ter um valor
    - o último dos processos a executar a instrução  $\text{turn} = p$

# Espera activa por Peterson

- encadramento de instruções

Proc 0		Proc 1
flag[0]=TRUE;		
		flag[1]=TRUE;
turn=0;		
		turn=1;
while(turn==0 && flag[1]==TRUE);		while(turn==1 && flag[0]==TRUE);



- Flag [1] = TRUE antes de turn=1

- turn=0 antes

- não é suficiente para o processo 0 sair do while.

- turn=0 depois

- leva o processo 1 a não ficar preso no while.

- P0 entra na RC porque

- Turn==1

- P1 só entra quando

- Flag[0] = False

# Espera activa por Peterson

- 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 atômica.
- 
- Progresso e espera limitada
  - Uma tarefa espera, no máximo, que a outra tarefa saia da RC:
    - assim que o fizer, a tarefa à espera entra na RC.

# Espera activa por Lamport

- Válida para qualquer número de tarefas.
- Antes de entrar na RC, a tarefa recebe uma ficha numerada
  - motivo porque este algoritmo é também designado por padaria
- Entra na RC a tarefa com a ficha de número mais baixo.
- Ao contrário da realidade, várias tarefas podem receber o mesmo número de bilhete.
  - Solução: desempatar pelo número do processo (esse sim, o sistema operativo garante ser único).
- $(a,b) < (c,d)$  se
  - $a < c$  ou  $(a == c \text{ e } b < d)$
  - Semelhante ao strcmp



# Espera activa por Lamport

- Bool choosing[N]; /\*anúncio de intenção em recolher bilhete\*/
- int numb[N]; /\*número de bilhete atribuído a cada processo\*/
- void initialize() {
- int i;
- for(i=0;i<N;i++) {
- choosing[i]=FALSE;
- numb[i]=0;
- }
- }

# Espera activa por Lamport

- `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 bilhete */`
  - `/* espera enquanto alguém está na RC:`
  - `Tem bilhete, Tem preferência de acesso */`
  - `while ( numb[i]!=0 && (numb[i],i)<(numb[p],p) ) ;`
  - `}`
  - `}`
- ```
void leave_region(int p) {  
    numb[p]=0;  
}
```

# Espera activa por Lamport

- Verificação de exclusão mútua
- Pior caso quando várias tarefas recolhem o mesmo número
  - processos pode ser substituído entre o cálculo da expressão e atribuição  $\text{numb}[p] = \max(\text{numb}[0], \dots, \text{numb}[N-1]) + 1$ ).
- Processos que cheguem depois recebem números superiores.
- Se um processo  $k$  se encontrar na RC e  $p$  pretende entrar, então
  - $\text{numb}[k] \neq 0$
  - apenas na região de saída  $\text{numb}[k]$  volta a 0
  - $\text{numb}[k], k$  é menor que  $\text{numb}[p], p$
- Logo, apenas um processo pode estar na RC.

# Espera activa por Lamport

- 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.
- 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

- 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

- Sistemas multiprocessador
- Necessárias instruções especiais que permitam testar e modificar uma posição de memória de forma atômica (i.e., sem interrupções).
- Instruções de teste e modificação atômica 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

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

# Solução por hardware: TSL

- Verificação de exclusão mútua
  - lock possui dois valores, pelo que apenas uma tarefa pode executar a RC.
  - Se for FALSE, a instrução retorna FALSE
    - (a tarefa entra na RC e altera atonicamente lock para TRUE).
  - Se for TRUE,
    - a instrução retorna o mesmo valor e a tarefa mantém-se na RE.
- Progresso,
  - se as tarefas na RR não alterarem o lock.
- Espera limitada:
  - não garantida,
    - depende da forma como o sistema operativo escala as tarefas
      - (duas podem ocupar alternadamente a RC, deixando uma terceira eternamente à espera)



# Solução por hardware: Swap

- 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 tarefas.
- Cada tarefa possui uma variável local key.
  - `do {`
  - `key = TRUE;`
  - `while (key==TRUE) Swap (&lock, &key ); /* RE */`
  - 
  - `lock = FALSE; /* RS */`
  - `} while (TRUE);`

# Solução por hardware: Swap

- Verificação de exclusão mútua
  - lock possui dois valores, pelo que apenas uma tarefa pode executar a RC.
- Se for FALSE,
  - Swap coloca key a FALSE
  - teste while termina.
- Se for TRUE,
  - Swap mantém key a TRUE
  - teste while é satisfeito, mantendo-se o ciclo.
- 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
  - Compara acumulador com destino
    - Se iguais, dest <- src.
    - Se diferentes, acumulador <- destino



- O Linux disponibiliza muitas formas de sincronização (✓ a estudar neste capítulo).
- Variáveis de sincronização
  - Semáforos ✓
  - Mutexes ✓
  - Mutexes condicionais ✓
  - Spinlocks
- Mecanismos de sincronização
  - Implementação da região crítica ✓
    - por variáveis de sincronização
    - por mensagens
- Rendez-vous ✓
- Barreiras ✓
- Monitores
- Tranca de registos

# Semáforos - introdução (1)

- [Def] Semáforo: abstracção de um contador e de uma lista de descritores de processos, usado para sincronização.
  - Propostos por Dijkstra, 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 errno.
- <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 (“unnamed”), 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  $N=1$ , o semáforo é definido como binário.
- Nota: O POSIX:SEM não discrimina o tipo `sem_t`. Nesta disciplina usamos a seguinte definição
- ```
typedef struct{
```
- ```
    unsigned counter;
```
- ```
    processList *queue;
```
- ```
}sem_t;
```

Fila de descritores de processos bloqueados

# Semáforos - introdução (3)

- S.counter determina quantos processos podem entrar na zona crítica sem bloquear.
  - Se S.counter for sempre limitado a 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:
  - 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
    - Block(S); /\* insere S na fila de processos bloqueados \*/
- Signal(S), up(S), ou V(S)
  - if (S.queue!=NULL)
    - WakeUp(S); /\* retira processo da fila de bloqueados \*/
  - 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)

- `down(S)` usada por uma tarefa para esperar por acesso (concedido por outra tarefa que executa o `up(S)` ).
  - Caso o acesso não seja possível de imediato, o SO bloqueia a tarefa numa fila.
- `up(S)` usada por uma tarefa para sinalizar disponibilidade de acesso a recurso.
  - `up(S)` não é bloqueante.
- Quando o valor de inicialização é 0, os processos que escrevem e lêem 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-vous”.

# Semáforos - POSIX

| Identificados               |                                                             | Anónimos      |
|-----------------------------|-------------------------------------------------------------|---------------|
| sem_open()                  |                                                             | sem_init()    |
|                             | sem_wait()<br>sem_trywait()<br>sem_post()<br>sem_getvalue() |               |
| sem_close()<br>sem_unlink() |                                                             | sem_destroy() |

| <b>serviço</b> | <b>POSIX</b> |
|----------------|--------------|
| up             | sem_post     |
| down           | sem_wait     |

# Sem. anónimos - definição (1)

- Um semáforo anónimo é uma variável de tipo `sem_t`
  - `#include <semaphore.h>`
  - `sem_t sem;`
- Um semáforo tem de ser inicializado antes de ser usado
  - por processos filhos,
  - Por threads.
- 
- **`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 do counter.

# Sem. anónimos - definição (2)

- Um semáforo que deixe de ser útil deve ser eliminado pela função
  - `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
  - Identificador do semáforo: cadeia de caracteres na forma /name.
- Nota: os semáforo identificados são instalados em /dev/shm, com o dev identificador sem.name
- Um semáforo tem de ser aberto antes de ser acedido

# Semáforos identificados - definição

## (2)

- `sem_t *sem_open(const char *,int,...);`
  - 1º parâmetro: identificador da conexão.
  - 2º parâmetro: flags,
    - `O_CREAT` - a conexão é criada se não existir,
    - `O_EXCL` - com `O_CREAT` a função falha se a conexão existir.
- se o 2º parâmetro contiver o bit `O_CREAT` a 1,
  - devem ser indicados mais dois parâmetros de modos:
  - 3. parametro - `mode_t`,
    - determinado as permissões (`S_IRUSR`, `S_IWUSR`, `S_IRGRP` ou `S_IROTH`).
  - 4º parametro - `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` e não `O_EXCL`,
    - a função ignora os 3º e 4º parâmetros.

# Semáforos identificados - definição

## (3)

- Quando um semáforo deixa de ser necessário a um processo, ele deve ser fechado.
  - `int sem_close(sem_t *)`;
- O último processo a aceder ao semáforo deve,
  - depois de o fechar,
  - eliminá-lo igualmente pela função.
- `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)

- P(S), ou down(S) implementada nos semáforos anónimos e identificados pela função
- `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 uma tarefa a executar P(S), ele fica bloqueado.
    - O que sucede às restantes tarefas depende do conhecimento que o gestor de fios de execução tiver:
    - se o gestor residir no núcleo (LKP), as restantes tarefas não são bloqueados.
- Nota: o programador deve confirmar o modelo de implementação dos fios de execução.



# Semáforos - operações (2)

- V(S) ou up(S) implementada nos semáforos anónimos e identificados pela função
- `int sem_post(sem_t *);`
- O semáforo sem, inicializado a 1, é partilhado por todas as tarefas. RC garantida pelo seguinte código
- `do {`
- `sem_wait( &sem ); /* RE */`
- `/* RC */`
- `sem_post( &sem ); /* RS */`
- `/* RR */`
- `} while (TRUE);`

# Semáforos - operações (3)

- Verificação de exclusão mútua
  - Se o contador do semáforo tiver valor 1, nenhuma tarefa se encontra na RC. Quando uma tarefa se encontrar na RE, entra e o contador é decrementado para 0.
- Se o contador do semáforo tiver valor 0, uma tarefa que queira entrar na RC fica bloqueada até que a tarefa dentro da RC saia.
- Progresso,
  - se as tarefas na RR não alterarem o semáforo.
- Espera limitada:
  - só garantida se o armazém de processos à espera for organizada como fila.

# Semáforos - operações (4)

- Em ambos semáforos, anónimos e identificados, o processo pode observar o valor do contador pela função
- `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 o valor “retornado”:
  - para um valor negativo igual ao número de processos bloqueados no semáforo.
  - para 0 (implementação adoptada pelo Linux).

# Aplicações de semáforos (1)

- Existem 4 situações típicas de aplicação, exigindo número variável de semáforos:
- Para apropriação de recursos escassos.
  - Ex: consideremos o acesso a uma sala, de capacidade limitada, por vários visitantes.
    - `#define CAP 10 /* capacidade da sala */`
    - `sem_t room;`
    - `sem_init( &room,0,CAP );`
  - Cada visitante é uma “thread”, que acede à sala com o código
    - `sem_wait( &room );`
    - `/* estadia */`
    - `sem_post( &room );`

# Aplicações de semáforos (2)

- para sincronização da transferência de dados.
  - consideremos um único produtor e consumidor, ligados por memória tampão de 1 posição.
    - `sem_t empty, full;`
    - `sem_init( &empty, 0, 1 ); /* inicialmente há lugar */`
    - `sem_init( &full, 0, 0 );`
- Produtor
  - `sem_wait( &empty );`
  - `put(buf);`
  - `sem_post( &full );`
- Consumidor
  - `sem_wait( &full );`
  - `get( buf );`
  - `sem_post( &empty );`

# Aplicações de semáforos (3)

- Para sincronização da transferência de dados.
  - Se a memória tampão de capacidade maior que 1 for acedida por várias “threads”, as operações de acesso à memória tampão têm de ser protegidas por um semáforo binário extra (ou por um mutex ).
    - `sem_t emptyCount, fillCount, access;`
    - `sem_init( &access, 0, 1 );`
    - `sem_init( &emptyCount, 0, BUFFER_SIZE );`
    - `sem_init( &fillCount, 0, 0 );`
- Produtor
  - `sem_wait( &emptyCount );`
  - `sem_wait( &access );`
  - `put(buf);`
  - `sem_post( &access );`
  - `sem_post( &fillCount );`
- Consumidor
  - `sem_wait( &fillCount );`
  - `sem_wait( &access );`
  - `get( buf );`
  - `sem_post( &access );`
  - `sem_post( &emptyCount );`

# Aplicações de semáforos (3)

- Para sincronização da transferência de dados.
  - Se a memória tampão de capacidade maior que 1 for acedida por várias “threads”, as operações de acesso à memória tampão têm de ser protegidas por um semáforo binário extra (ou por um mutex ).
    - `sem_t emptyCount, fillCount, access;`
    - `sem_init( &emptyCount, 0, BUFFER_SIZE );`
    - `sem_init( &fillCount, 0, 0 );`
- Produtor
- `sem_wait( &emptyCount );`
- `put(buf);`
- `sem_post( &fillCount );`
- Consumidor
- `sem_wait( &fillCount );`
- `get( buf );`
- `sem_post( &emptyCount );`

# Aplicações de semáforos (4)

- [Def] Rendez-vous : barreira em que o número de tarefas envolvidas é 2.
- O rendez-vous pode ser implementado com 2 semáforos inicializados a 0.
- Inicialização
  - `sem_t rvA,rvB;`
  - `sem_init( &rvA,0,0 ); sem_init( &rvB,0,0 );`
- Solução que provoca impasse (“deadlock”) – cada thread fica à espera que a outra desbloqueie a espera.
  - `/* Thread 1 */            /* Thread 2 */`
  - `sem_wait(&rvB);        sem_wait(&rvA);`
  - `sem_post(&rvA);        sem_post(&rvB);`



# Aplicações de semáforos (4)

- Para evitar bloqueio, trocar a ordem das operações numa das tarefas
- Solução ineficiente - processador pode comutar entre as duas tarefas mais vezes que o necessário.

|                     |                   |
|---------------------|-------------------|
| - /* Thread 1 */    | /* Thread 2 */    |
| - sem_wait( &rvB ); | sem_post( &rvB ); |
| - sem_post( &rvA ); | sem_wait( &rvA ); |

- Solução eficiente

|                     |                   |
|---------------------|-------------------|
| - /* Thread 1 */    | /* Thread 2 */    |
| - sem_post( &rvA ); | sem_post( &rvB ); |
| - sem_wait( &rvB ); | sem_wait( &rvA ); |

# Sincronização por gestor (1)

- A região crítica pode ser transferida para um processo dedicado, o gestor de recursos.
  - Serialização garantida pela ordem de leitura/resposta aos pedidos.
  - Um processo que pretenda alterar dados críticos executa a seguinte sequência de passos:
    - Gerar mensagem de pedido.
    - Enviar mensagem para o gestor.
    - Ficar, bloqueado, à espera da resposta do gestor.
    - 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 \*/
  - `}`

# 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) ou mutexes sobre condições.
- 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, correspondem a semáforos binários (0-locked, 1-unlocked).
  - 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
- Nota: mutex ::= MUTual EXclusion

# Mutexes - definição (2)

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

# Mutexes - definição (3)

- Definição: localização de tipo `pthread_mutex_t`
  - `#include <pthread.h>`
  - `pthread_mutex_t mux;`
- Inicialização: Um mutex tem de ser inicializado antes de ser usado
  - Dinâmica:
- **`int pthread_mutex_init (pthread_mutex_t *, const pthread_mutexattr_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)

- Eliminação: Um mutex é eliminado pela função
- **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;`
  - `pthread_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)

- Aquisição: Aquisição de um mutex, com o respectivo trancar, é efectuada por duas funções
- **`int pthread_mutex_lock(pthread_mutex_t *)`**;
  - bloqueia até o mutex se encontrar disponível.
- **`int pthread_mutex_trylock(pthread_mutex_t *)`**;
  - 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`.
- Os mutexes devem ser trancados pelo fio de execução no mais curto intervalo de tempo possível.

# Mutexes - operações (2)

- Libertação:
  - A primitiva de destrancar um mutex é implementada pela função
- **`int pthread_mutex_unlock(pthread_mutex_t *);`**
- RC garantida pelo seguinte código
  - `pthread_mutex_t mux=PTHREAD_MUTEX_INITIALIZER;`
  - `do{`
  - `pthread_mutex_lock( &mux );     /* RE */`
  - `/* RC */`
  - `pthread_mutex_unlock( &mux ); /* RS */`
  - `/* RR */`
  - `while(TRUE);`

# Aplicações de semáforos e mutexes (1)

- A espera num semáforo pode ter de ser antecedida por uma consulta a dados protegidos por mutexes.
- O mutex tem de ser libertado (`pthread_mutex_unlock`) antes de ser executada a espera no semáforo (`sem_wait`).
- O formato do programa depende ao número de dados a consultar.
  - Um único dado: consulta executada com instrução `if-else`.
  - Vários dados: consulta repetida com instrução `while`.

# Aplicações de semáforos e mutexes

## (2)

- Um único dado
- `pthread_mutex_lock( &mux );`
- `if ( foo(contador) ) {`
- `/* espera no semáforo */`
- `alteraContador();`
- `pthread_mutex_unlock( &mux );`
- `sem_wait( &sem );`
- `}else {`
- `/* não espera no semáforo */`
- `alteraContador();`
- `pthread_mutex_unlock( &mux );`
- `}`

# Aplicações de semáforos e mutexes

## (3)

- Vários dados:
  - usar ciclo para verificar condições após o sem\_post de outra “thread”.
- `while ( 1 ) {`
- `pthread_mutex_lock( &mux );`
- `if ( foo1(cont1) && foo2(cont2) ) {`
- `/* espera no semáforo */`
- `alteraContador();`
- `pthread_mutex_unlock( &mux );`
- `sem_wait( &sem );`
- `}else`
- `break;`
- `}`
- `pthread_mutex_unlock( &mux );`

# Mutexes sobre condições - definição (1)

- Pode haver interesse em sincronizar o acesso a dados
  - com base em variáveis que satisfaçam determinadas condições
  - (e.g., sensor de pressão atinge um valor limiar-“threshold”).
- A espera activa-“busy waiting” consome muito CPU
- ```
while(1) {
```
- ```
    pthread_mutex_lock(&mux);
```
- ```
    if (var==threshold) break;
```
- ```
    pthread_mutex_unlock(&mux);
```
- ```
}
```
- O POSIX disponibiliza variáveis de condição, que actuam como uma porta:
  - Um fio de execução espera fora da porta, chamando `pthread_cond_wait()`.
  - O fio de execução à espera abre porta quando outro fio de execução chamar `pthread_cond_signal()`.

# Mutexes sobre condições - definição (2)

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

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

```
pthread_cond_t cont;
```

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

```
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_INITIALIZER;`

# Mutexes sobre condições - definição (3)

- Um mutex condicionado é eliminada pela função

**`int pthread_cond_destroy( pthread_cond_t * );`**

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



# Mutexes sobre condições— operações(1)

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

```
int pthread_cond_wait(pthread_cond_t *,  
pthread_mutex_t *);
```

- bloqueia o fio de execução até outro fio de execução assinalar possível alteração da condição.

```
int pthread_cond_timedwait(pthread_cond_t *,  
pthread_mutex_t *,const struct timespec*);
```

- a espera é temporizada por uma função.

# Mutexes sobre condições— operações (2)

- **pthread\_cond\_timedwait**
  - 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)

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

**int pthread\_cond\_signal(pthread\_cond\_t \*);**

- destranca pelo menos um fio de execução bloqueado numa variável condicionada

**int pthread\_cond\_broadcast(pthread\_cond\_t \*);**

- destranca todos os fios de execução bloqueados numa variável condicionada.
- parâmetro: endereço da localização da variável condicionada.

# Mutexes sobre condições – exemplo (1)

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

- `int main(int argc, char *argv[]) {`
- `if(argc!=2) {`
- `printf("&s numb,0<numb<2048\n",argv[0]);`
- `exit(1); }`
- `SR = atoi( argv[1] ); /*mau mau mau !!! */`
- `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; }`

# Mutexes sobre condições – exemplo (1)

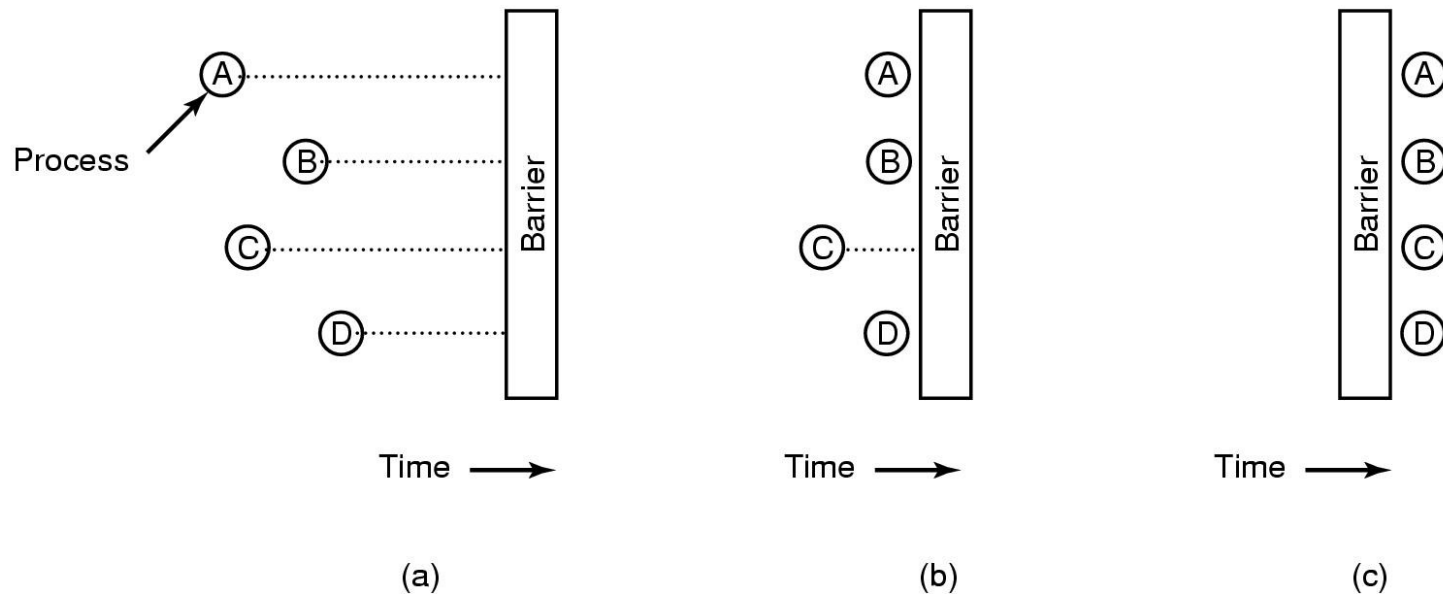
- `void *modify() {`
- `pthread_mutex_lock( &count_lock );`
- `while(1) {`
- `LFSR();`
- `pthread_mutex_lock( &cond_mut );`
- `temperature = (TH+20)*SR/7148;`
- `number++;`
- `pthread_mutex_unlock(&cond_mut);`
- `if (temperature>=TH) {`
- `pthread_cond_signal( &cvar );`
- `pthread_mutex_lock( &count_lock );}`
- `}`
- `return NULL; }`

Acorda thread bloqueada na condição

Fica à espera que temp seja restabelecida

# Barreiras (1)

- [Def] Barreira:
  - mecanismo de sincronização, em que os fios de execução ficam bloqueados até o número de threads bloqueadas atingir determinada quantidade.



# Barreiras (2)

- As barreiras são usadas quando o processamento é feito por etapas,
  - com a entrada na etapa seguinte feita após todos os fios de execução terem concluído a etapa anterior.
- Barreiras disponibilizadas no pacote pthreads,
  - com tipo de dados pthread\_barrier\_t
- Inicialização:

**pthread\_barrier\_init(pthread\_barrier\_t \*,  
pthread\_barrierattr\_t \*, unsigned int)**

- O 2º parâmetro identifica os atributos (por omissão, usar NULL).
- O 3º parâmetro determina número de threads a sincronizar.



# Barreiras (3)

- Espera

**int pthread\_barrier\_wait( pthread\_barrier\_t \*)**

- Uma das threads recebe
  - PTHREAD\_BARRIER\_SERIAL\_THREAD,
- todas as outras
  - recebem 0.
- 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 sincronizam numa barreira e imprimem o resultado final do contador.
- Nota: alteração do contador protegido por mutex.

