

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 $N>2$

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 */
 - }

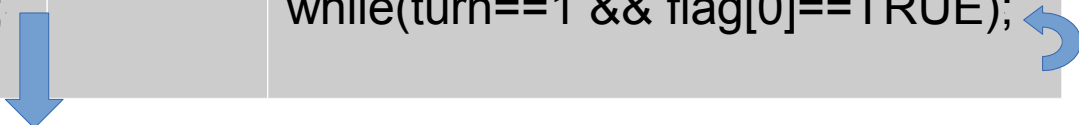
Espera activa por Peterson

- 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 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 atomicamente 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 */`
 - `/* RC */`
 - `lock = FALSE; /* RS */`
 - `/* RR */`
 - `} 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