MAC0438 – Programação concorrente

Daniel Macêdo Batista

IME - USP, 7 de Maio de 2013

Roteiro

Mudanças de prioridade no problema dos leitores e escritores

Alocação de recursos e escalonamento

Utilizando semáforos em grafos de precedência

Mudanças de prioridade no problema dos leitores e escritores

Prioridade no algoritmo atual

- Leitores continuam com prioridade
- Porém, na versão atual é possível controlar como o bastão é passado

  Basta mudar a ordem das comparações nos "`SIGNAL`" e os protocolos antes de ler ou escrever na base

Priorizando os escritores

- Novos leitores terão que ir para o "final da fila" se houver algum escritor esperando
- Um leitor só pode ser acordado se nenhum escritor estiver esperando

Priorizando os escritores

- Novos leitores terão que ir para o "final da fila" se houver algum escritor esperando

  Reforçar o requisito para manter o leitor esperando antes de ler a base

- `if (nw > 0 or dw > 0) {dr = dr+1; V(e); P(r);}`

Priorizando os escritores

- Um leitor só pode ser acordado se nenhum escritor estiver esperando

  Trocar a ordem das comparações após um escritor escrever na base

```
if (dw > 0) {dw = dw-1; V(w);}
elseif (dr > 0) {dr = dr-1; V(r);}
else V(e);
```

Bolando um algoritmo justo entre classes de processos

- Podemos forçar leitores e escritores a alternarem os acessos se houverem processos das duas classes esperando
- Envie um **novo** leitor para o "final da fila" se houver um escritor esperando
- Envie um **novo** escritor para o "final da fila" se houver um leitor esperando
- Acorde um escritor que estiver esperando (se houver) quando um leitor terminar de ler
- Acorde todos os leitores que estiverem esperando (se houver) quando um escritor terminar de escrever; caso contrário, acorde um escritor que estiver esperando (se houver)

Bolando um algoritmo justo entre classes de processos

- Envie um **novo** leitor para o "final da fila" se houver um escritor esperando
- Envie um **novo** escritor para o "final da fila" se houver um leitor esperando

- `if (nw > 0 or dw > 0) {dr = dr+1; V(e); P(r);}`

- `if (nr > 0 or dr > 0 or nw > 0) {dw = dw+1; V(e); P(w);}`

Bolando um algoritmo justo entre classes de processos

- Acorde um escritor que estiver esperando (se houver) quando um leitor terminar de ler
- Acorde todos os leitores que estiverem esperando (se houver) quando um escritor terminar de escrever; caso contrário, acorde um escritor que estiver esperando (se houver)
- O último algoritmo, com os SIGNAL simplificados já atendia isso :)

Bolando um algoritmo justo entre classes de processos

- Leitor após ler a base

```
if (nr == 0 and dw > 0) {dw = dw-1; V(w);}
else V(e);
```

- Escritor após escrever na base

```
if (dr > 0) {dr = dr-1; V(r);}
elseif (dw > 0) {dw = dw-1; V(w);}
else V(e);
```

Alocação de recursos e escalonamento

Alocação de recursos

- É o problema de decidir quando um processo pode ter acesso a um recurso

  Recurso = entrar na seção crítica, acessar base de dados, etc...

- As soluções até agora davam acesso a **algum** recurso que estivesse esperando se o recurso tornar-se disponível

- Não nos preocupamos com qual leitor vai ler da base, ou qual produtor dentre todos conseguirá escrever primeiro no buffer

- No máximo bolamos uma política de escalonamento entre classes de processos (leitores e escritores)

Alocação de recursos

- É possível usar semáforos para controlar qual processo dentre um conjunto vai acessar o recurso primeiro
- Útil em sistemas de tempo real, em clusters ou em grades

Solução geral

- Processos que queiram acessar um recurso devem executar um procedimento `request`

  Processos informam quantas unidades do recurso compartilhado quer acessar

  Processos passam suas identidades

- `request`

  Só vai deixar o processo prosseguir quando todas as unidades requisitadas estiverem livres

Solução geral

- Após usar os recursos, um processo os libera para outros executando um procedimento `release`
- `release`

  Libera os recursos para outros processos

Solução geral


```
request (parametros):
   <await (request poder ser atendida)
    acesse as unidades;>

release (parametros):
   <retorne as unidades;>
```

Solução geral

- Podemos implementar condições de sincronização com semáforos usando a técnica de passagem de bastão
- 1 semáforo e para controlar a execução das ações atômicas
- 1 semáforo e 1 contador para cada condição

Solução geral


```
request(parametros):
   P(e);
   if (request nao pode ser atendida) DELAY;
   acesse as unidades;
   SIGNAL;
```

```
release(parametros):
   P(e);
   retorne as unidades;
   SIGNAL;
```

Alocando a tarefa mais curta

- `DELAY` e `SIGNAL` devem ser especificados para a política de escalonamento a ser implementada
- Se a política for `Shortest-job-next`, por exemplo

  Cada processo informa um par `(tempo,id)` com o tempo que vai gastar para acessar o recurso e sua identificação

  A ideia é escalonar o que tiver menor tempo primeiro

  Há um semáforo associado com cada `id`

Alocando a tarefa mais curta

- `DELAY`

  Vai inserir o processo em uma fila (ordenar pelo menor tempo)

  Vai liberar o semáforo e e vai esperar o semáforo do processo ser liberado

- É de responsabilidade do processo liberar o acesso ao recurso chamando o `release`

Alocando a tarefa mais curta

- `SIGNAL` no `request`

  Basta liberar o semáforo e

- `SIGNAL` no `release`

  Verifica se há algum processo na fila

  Remove o primeiro processo e libera o semáforo dele

  Se não houver processo na fila só libera o semáforo e

Alocando a tarefa mais curta


```
request(tempo,id):
   P(e);
   if (!free) {
      insere(tempo,id) na fila;
      V(e);
      P(b[id]);
   }
   free=false;
   V(e);
```

Alocando a tarefa mais curta

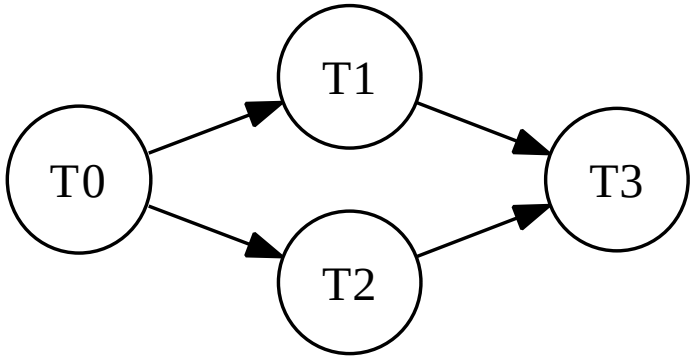

```
release():
   P(e);
   free=true;
   if (fila nao estah vazia) {
      remova primeiro par (tempo,id) da fila;
      V(b[id]);
   }
   else V(e);
```

Utilizando semáforos em grafos de precedência

Aplicações com diversas tarefas

- Programas concorrentes mais complexos costumam ser representados por grafos
- Cada tarefa (thread) é representada por um vértice
- Cada dependência entre threads é representada por um arco
  - As dependências podem ser por exemplo a espera para que um buffer esteja cheio

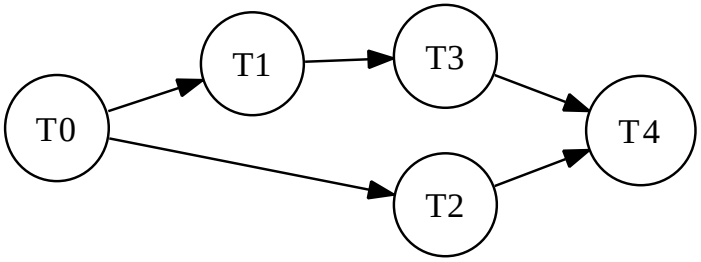
Exemplo de grafo




```
process T {
  espere os predecessores, se houver algum;
  execute a tarefa;
  sinalize os sucessores, se houver algum;
}
```

- Como implementar a espera e a sinalização com semáforos no exemplo?

Exemplo de exercício



- Para o grafo da figura, apresente uma solução de sincronização com a menor quantidade possível de semáforos **sem impor restrições não especificadas no grafo** (por exemplo, T1 e T2 devem ter condições de rodar exatamente em paralelo assim que T0 terminar)