MAC0438 – Programação Concorrente

Daniel Macêdo Batista

IME - USP, 24 de Maio de 2012

Monitores

- Alto nível (Semáforos $\rightarrow$ Baixo nível)
- Facilitam a sincronização entre processos

Sintaxe e semântica


```
monitor nomedomonitor {
   variaveis permanentes;
   comandos de inicializacao;
   demais procedimentos;
}
```

- Variáveis permanentes $\rightarrow$ Atributos privados de uma classe
- Comandos de inicialização $\rightarrow$ Construtor de uma classe
- Demais procedimentos $\rightarrow$ Métodos de uma classe

Sintaxe e semântica

- Única forma de modificar as variáveis é através dos procedimentos

```
call nomedomonitor.nomedoprocedimento(argumentos);
```

Exclusão mútua

- Implícita
- No máximo uma instância de qualquer procedimento pode estar ativa por vez (Ativa = um processo está executando algum comando daquele procedimento)

  Duas chamadas ao mesmo procedimento não podem estar ativas ao mesmo tempo

  Duas chamadas a diferentes procedimentos não podem estar ativas ao mesmo tempo

Condição de sincronização

- Explícita
- Utiliza **variáveis de condição**

  Usadas para atrasar um processo (O processo só pode continuar quando seu estado satisfizer alguma condição booleana)

  Usadas para acordar um processo quando a condição torna-se verdadeira

```
cond nomedavariavel;
```

Variáveis de condição

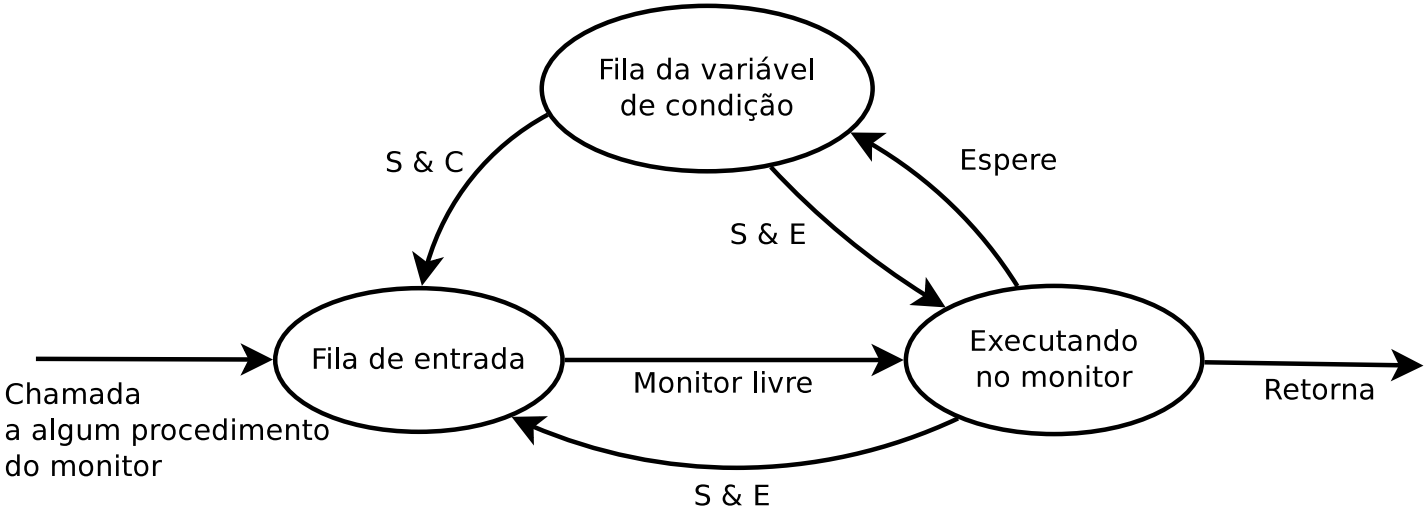
- Manipuladas dentro do monitor por diversas funções

```
empty(nomedavariavel);
```

```
wait(nomedavariavel);
```

```
signal(nomedavariavel);
```

Resumo da sincronização em monitores



- Além do Signal and Urgent Wait que leva o processo sinalizador para o início da fila de entrada

Um semáforo implementado com monitor


```
monitor Semaforo {
   int s = 0; /* Valor do semaforo (s >= 0) */
   cond pos; /* Sinaliza quando s > 0 */

   procedure Psem() {
      if (s == 0) wait(pos);
      s = s-1;
   }

   procedure Vsem() {
      s = s+1;
      signal(pos);
   }
}
```

Roteiro

Mais sobre disciplinas de sinalização

Operações adicionais com variáveis de condição

Implementando monitores por meio de semáforos

Buffers limitados

Mais sobre disciplinas de sinalização

Verificando o semáforo

- Procedimento `Psem`

```
procedure Psem() {
  if (s == 0) wait(pos);
  s = s-1;
}
```

- Se algum processo chamar o procedimento `Psem`, vai para a fila se `s==0`
- `s = s-1` apenas se o `s` for positivo
- O decremento de `s` está protegido pela definição de monitor

Verificando o semáforo

- Procedimento `Vsem`

```
procedure Vsem() {
   s = s+1;
   signal(pos);
}
```

- Se algum processo chamar `Vsem`, vai acordar outro que esteja esperando `s` ser positivo
- Se não tiver processo esperando, o `signal` não faz nada
- O processo que vai continuar a execução após `Vsem` ser chamado vai depender da disciplina de sinalização

Verificando o semáforo

- Procedimentos OK!
- Agora temos que avaliar o semáforo com as diferentes disciplinas (S&C) e (S&E)

```
 /* Processo que estah esperando na fila 'pos' */
 procedure Psem() {
   if (s == 0) wait(pos);
   s = s-1;
}
```

- S&E

  Processo que chamou o `Vsem` incrementou `s`, acordou o processo mais velho na fila 'pos' e foi para o fim da fila de entrada do monitor

Verificando o semáforo


```
/* Processo que estah esperando na fila 'pos' */
procedure Psem() {
  if (s == 0) wait(pos);
  s = s-1;
}
```

- S&E

  Processo mais velho acorda, então `s` é positivo.

  Pode decrementar `s`

  OK!

Verificando o semáforo


```
/* Processo que estah esperando na fila 'pos' */
procedure Psem() {
  if (s == 0) wait(pos);
  s = s-1;
}
```

- S&C

  Processo que chamou o `Vsem` incrementou `s` e enviou o processo mais velho na fila '`pos`' para o fim da fila de entrada do monitor (por enquanto `s` é 1)

  O processo acordado vai rodar alguma hora mas não sabemos quando

Verificando o semáforo


```
/* Processo que estah esperando na fila 'pos' */
procedure Psem() {
  if (s == 0) wait(pos);
  s = s-1;
}
```

- S&C

  `s` só pode ser decrementado se for zero

  O processo acordado foi para o fim da fila de entrada. Quando ele for escalonado para ser executado `s` pode ser decrementado?

Verificando o semáforo


```
/* Processo que estah esperando na fila 'pos' */
procedure Psem() {
  if (s == 0) wait(pos);
  s = s-1;
}
```

- S&C

  Precisa verificar o valor de s de novo antes de decrementá-lo. Como fazer isso?

Verificando o semáforo


```
/* Processo que estah esperando na fila 'pos' */
procedure Psem() {
  while (s == 0) wait(pos); /* if OK soh para S&E */
  s = s-1;
}
```

- S&C

  Agora garante que s só vai ser ser decrementado quando for positivo

  Mas um processo que chegou antes pode ser executado só depois de um que chegou depois :(Este semáforo não é FIFO! Injusto!

Bolando uma versão melhorada do semáforo

- ☐ Funcionar corretamente independente da disciplina
- ☐ Sem `while`
- ☐ Justo (Semáforo FIFO)

Revisando o problema

- A diferença entre S&C e S&E é que no primeiro, o `s` é incrementado pelo `Vsem` e pode ser visto por outros processos que estejam na fila de entrada do monitor
- Outros processos podem executar `Psem`, decrementar `s` e impedir o processo que foi acordado de executar tão cedo :(

Revisando o problema

- Mas o que o processo que chamou o `Psem` tem que fazer no final das contas?

  Acordar o processo no início da fila de '`pos`'

  Ele **não** precisa incrementar o valor de `s`, dando a chance de outros processos passarem na frente do que foi acordado

Solução

- Mudanças no `Vsem`:

  Se há processo na fila, acorda ele mas não incrementa `s`. Caso contrário incrementa

- Mudanças no `Psem`:

  Se tiver que esperar, quando for acordado não decrementa `s` porque `Psem` não incrementou. Caso contrário decrementa

Implementação melhorada de semáforo com monitor


```
monitor SemaforoMelhorado {
   int s = 0; /* Valor do semaforo (s >= 0) */
   cond pos; /* Sinaliza quando s > 0 */

   procedure Psem() {
      if (s == 0) wait(pos);
      else s = s-1;
   }

   procedure Vsem() {
      if (empty(pos)) s = s+1;
      else signal(pos);
   }
}
```

- Técnica de passagem de condição

  O sinalizador passa implicitamente, para o processo acordado, que s é positivo

Comparando variáveis de condição com semáforos

- `P` em semáforos e `wait()` em monitores $\rightarrow$ fazem um processo esperar
- `V` em semáforos e `signal()` em monitores $\rightarrow$ acordam um processo
- Diferem porque:

  1) `wait()` sempre faz o processo esperar. `P` só faz esperar se o valor do semáforo for zero

  2) `signal()` não faz nada se não houver processo esperando. `V` incrementa o valor do semáforo sempre que é chamado (Não há lembrança da execução do `signal()`)

S&C daqui pra frente quando a disciplina não for informada

- S&C permite escalonamento de processos baseado em prioridade (Não necessariamente é o processo recém despertado que vai rodar)
- S&C é o mais utilizado: Unix, Java, `pthreads`

Operações adicionais com variáveis de condição

`wait` com prioridade

- O padrão do `wait` e `signal` é fornecer uma fila FIFO
- `wait` com prioridade permite que isso seja modificado

`wait` com prioridade


```
wait(var,rank)
```

- `rank` é um inteiro. Quanto menor, mais perto do início da fila
- Em caso de empate, o processo que já está na fila tem prioridade sobre o novo
- Para evitar confusão, deve-se usar sempre o mesmo `wait` (com ou sem prioridade)

Verificando o rank do primeiro processo

- Usando `wait` com prioridade, é útil saber o `rank` do primeiro processo da fila

```
minrank(var)
```

- Se a fila estiver vazia ou se ela não utiliza prioridade, a função retorna algum valor arbitrário

Acordando todos os processos

- Algumas vezes não importa a ordem com que os processos são acordados
- Pode-se acordar todos de uma vez só

```
signal_all(var)
```

- Útil também quando o sinalizador não sabe qual processo pode continuar (porque eles precisam reavaliar suas condições de espera)

Acordando todos os processos

- Equivalente a:

```
while (!empty(var)) signal(var);
```

- Com a disciplina S&E, a `signal_all` não é bem definida.
- Como seria possível acordar todos os processos e passar a execução para todos eles se apenas um pode estar ativo no monitor por vez? (Uma das razões porque na prática não se vê muito a disciplina S&E implementada)

Implementando monitores por meio de semáforos

Motivação

- Nem todas as bibliotecas suportam monitores. Apenas semáforos
- Muitas linguagens não fornecem monitores. Apenas semáforos

O que precisa ser implementado

1. Código de entrada (chamado sempre que um processo roda o `call` em algum procedimento do monitor)
2. Código de saída (chamado sempre que um processo termina de rodar um procedimento do monitor)
3. Código que implementa `wait`, `signal` e as outras operações avançadas sobre variáveis de condição

Código de entrada e código de saída

- O monitor precisa de exclusão mútua explícita

- 1 semáforo (`m`) por monitor

  Para garantir exclusão mútua sempre que um procedimento estiver sendo executado

- Basta inicializar com 1, rodar o `P(m)` no código de entrada e `V(m)` no código de saída

wait

- `wait(cv)` libera a exclusão mútua do monitor e envia o processo para a fila da variável de condição `cv` até que ele acorde com um `signal`

- Considerando que haja um tipo `queue`, criamos uma fila FIFO `cvDelay`

- Para saber que a fila está vazia podemos ter um contador `cvN`

- `cvDelay` começa vazia (`cvN=0`)

wait

- Quando um processo executa `wait(cv)`, há um incremento em `cvN` e o descritor do processo vai para a fila `cvDelay`. Depois o processo roda `V(m)` e se bloqueia com um semáforo privado
- Ao acordar, o processo roda `P(m)`

signal

- `signal(cv)` verifica o valor de `cvN`. Se for zero, não faz nada. Caso contrário, decrementa `cvN`, remove o processo mais velho da fila e sinaliza o semáforo privado dele.

Algoritmo


```
variaveis compartilhadas:
  sem m = 1     /* 1 semaforo por monitor */
  int cvN = 0   /* 1 contador por variavel de condicao */
  queue cvDelay /* 1 fila por variavel de condicao */
  sem private[N] /* 1 posicao por processo */

entrada do monitor:
  P(m);

saida do monitor:
  V(m);
```

Algoritmo


```
wait(cv):
  cvN++;
  insere id do processo em cvDelay;
  V(m);
  P(private[id]);
  P(m);

signal(cv):
  if (cvN > 0) {
     cvN--;
     remove id de cvDelay;
     V(private[id]);
  }
```

Outras operações

- `signal_all`, `empty`, `minrank`, `wait` com prioridade
- Tentem fazer :)

Buffers limitados

Relembrando o problema

- Produtores e consumidores comunicam-se por um buffer de `n` posições
- O buffer representa uma fila de mensagens
- O produtor coloca mensagens no final da fila
- O consumidor recebe mensagens lendo o início da fila
- **A sincronização nesse caso deve evitar mensagens depositadas se a fila está cheia ou mensagens lidas se a fila está vazia**

- O buffer é o recurso compartilhado. Ele deve ser implementado como um monitor

Como implementar o buffer usando monitor?

- O monitor terá dois procedimentos:

  `armazena`: armazena uma mensagem no buffer. Se estiver cheio, espera. Também sinaliza que há mensagem no buffer

  `recupera`: retira uma mensagem do buffer. Se estiver vazio, espera. Também sinaliza que há espaço no buffer

- O monitor terá duas variáveis de condição

  Acordar processos que foram dormir porque o buffer estava cheio ou vazio

  Uma vai sinalizar que o buffer não está vazio

  Uma vai sinalizar que o buffer não está cheio

Buffer limitado implementado com monitor


```
monitor BufferLimitado {
   typeT buf[n];
   int front=0;  /* Primeira posicao com mensagem */
   int rear=0;   /* Primeira posicao vazia */
   int count=0;  /* Quantidade de posicoes com mensagens */
   /* Obs.: rear == (front+count)%n */
   cond notfull; /* Fila sinalizada quando count < n */
   cond notempty; /* Fila sinalizada quando count > 0 */
}
```

Buffer limitado implementado com monitor


```
procedure armazena (typeT data) {
   while (count == n) wait(notfull);
   buf[rear] = data; rear = (rear+1)%n; count++;
   signal(notempty);
}

procedure recupera (typeT &result) {
   while (count == 0) wait(notempty);
   result = buf[front]; front=(front+1)%n; count--;
   signal(notfull);
}
```