Notas dos slides

APRESENTAÇÃO

O presente conjunto de slides pertence à coleção produzida para a disciplina *Introdução ao Processamento Paralelo e Distribuído* ofertada aos cursos de bacharelado em Ciência da Computação e em Engenharia da Computação pelo Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas.

Os slides disponibilizados complementam as videoaulas produzidas e tratam de pontos específicos da disciplina. Embora tenham sido produzidos para ser assistidos de forma independente, a sequência informada reflete o encadeamento dos assuntos no desenvolvimento do conteúdo programático previsto para a disciplina.

computação
Universidade Federal de Pelotas

universidade federal de pelotas
CDTec
centro de desenvolvimento tecnológico

Programação Multithread

Dos modelos às estratégias de implementação

> **“ Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.**
>
> Brian Kernighan, Criador do C

Notas da videoaula

DESCRIÇÃO

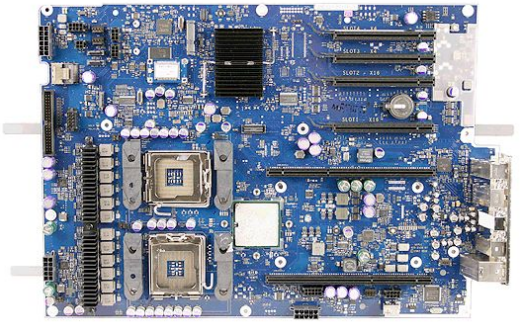
Nesta videoaula são apresentados os principais conceitos associados ao modelo de programação multithread.

OBJETIVOS

Nesta videoaula o aluno compreenderá a diferença entre as diferentes estratégias para implementação de ferramentas para programação multithread bem como as questões a serem consideradas na implementação de programas multithread.

Umas imagens: Arquiteturas Multiprocessadas

Uma arquitetura multiprocessada de 2013. Observe, nesta placa-mãe, a existência de slots para dois processadores.

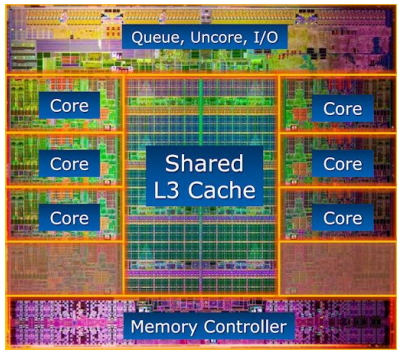


Umas imagens: Arquiteturas Multiprocessadas

O layout de um processador multicore SMP (Xeon Sandy-E 3960X).

Observe:

- Diferentes estratégias de compartilhamento de cache são utilizados, conforme o nível.
- Apenas uma saída do processador para a memória

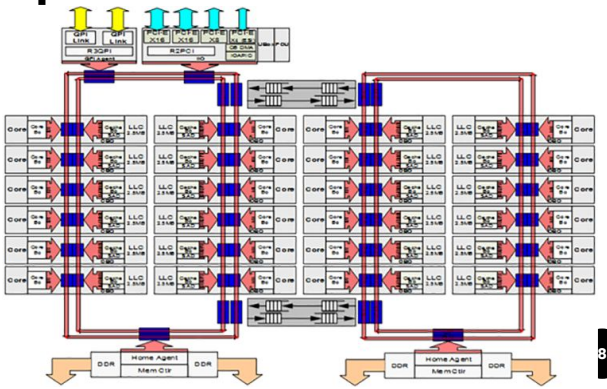


Umas imagens: Arquiteturas Multiprocessadas

O layout de um processador multicore escalável NUMA (Xeon E-2600).

Observe:

- Cada cluster de cores tem acesso direto à um módulo de memória
- Existe uma fila suportando acesso aos módulos de memória de outros clusters
- Há apenas um subsistema de I/O



Modelo de Arquitetura

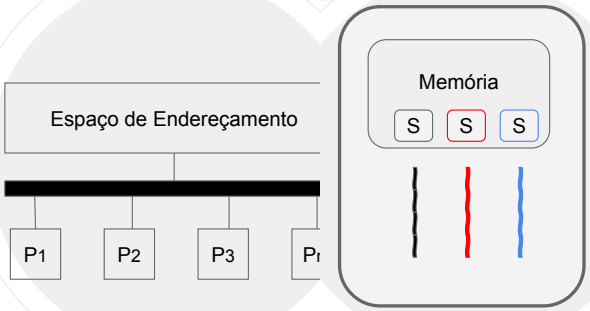
A arquitetura básica é um **multiprocessador**.

Um conjunto de *m* CPUs que compartilham acesso a um espaço de endereçamento contíguo.

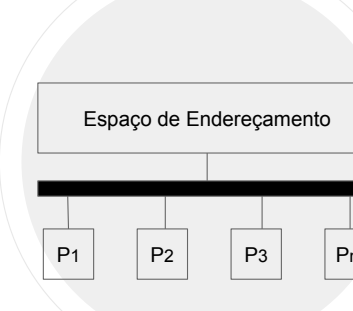
Estrutura bastante similar a de um processo multithread, onde **threads** compartilham a área de memória do processo.

A comunicação entre os threads se dá em acessos de leitura e escrita de dados na memória do processo.

Modelo de Arquitetura



O modelo de arquitetura de um multiprocessador é bastante próxima aquele apresentado por um processo multithread. Em ambos os casos, temos elementos, threads de um lado, CPUs de outro, responsáveis pela execução de instruções e um espaço de armazenamento que serve de substrato para compartilhamento de dados. **Diferença fundamental:** enquanto em um processo existe a ciência de que os threads buscam solucionar o mesmo problema, no multiprocessador supõe-se que as CPUs executem fluxos de execução não associados entre si.

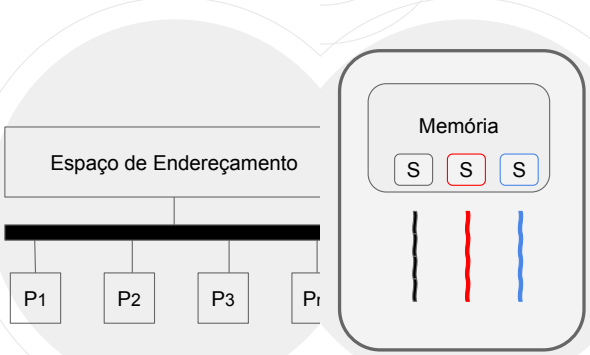


Modelo de Arquitetura

Na arquitetura:

- É garantido um mecanismo de exclusividade no acesso à memória
  - Granularidade Fina
    - Uma *palavra*
  - Semântica CREW (Concurrent Read, Exclusive Write)
  - Protocolo Snoop
- Operações atômicas
  - Prefixo *lock* às instruções
    - Garante que a palavra manipulada em uma instrução seja de uso exclusivo da CPU que a executou
      - Exemplo: `lock inc var`

Modelo de Arquitetura



No processo:

- Não existe nenhum mecanismo implícito para coordenar acesso a uma posição de memória
  - Granularidade Grossa
    - Unidade de dado manipulada no programa
- Mecanismos de sincronização
  - Coordenação da execução dos threads para acesso aos dados
  - Semânticas de acesso aos dados (Read, Write, Exclusive, Concurrent)
    - EREW
    - CREW
    - ERCW
    - CRCW
  - Estratégias
    - Exclusão mútua
    - Semáforos
    - Condições

ER: Apenas uma CPU possui o direito de leitura do dado
CR: Não há restrição para leitura do dado

EW: Apenas uma CPU possui o direito de escrita do dado
CW: Não há restrição para escrita do dado

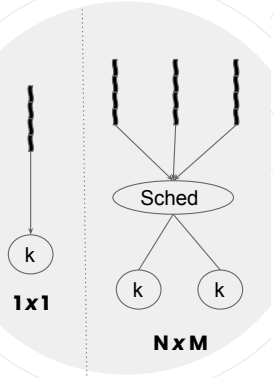
Implementação de Threads

Um para Um

Toda concorrência descrita pelo programa é mapeada, diretamente, sobre uma unidade de execução (thread kernel) gerida pelo hardware.

N para M

Distingue a descrição da concorrência da aplicação do suporte de paralelismo do hardware. Threads usuário são mapeadas, em espaço usuário, sobre unidades de execução (threads kernel) providas pelo ambiente de execução.



Existe o modelo N *x* 1.

Questões de programação

Unidade de Trabalho

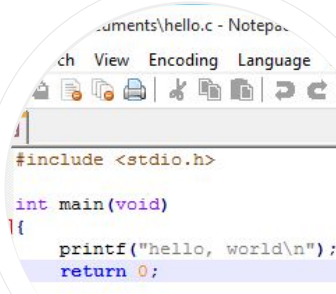
Define a granularidade da atividade concorrente. Problemas a serem considerados: lançamento, parametrização, término e retorno de resultados.

Comunicação

Transferência de dados de uma unidade de trabalho à outra. Em um ambiente com memória compartilhada, pode ser realizado pela escrita/leitura de dados em posições de memória acessíveis a ambas unidade de trabalhos envolvidas na comunicação.

Sincronização

Mecanismos que permitem coordenar a evolução das unidades de trabalho, permitindo que estas reconheçam o estado de sua execução em relação às demais.



Questões de programação

TASK

De grão mais fino, são definidas em termos de bloco de comandos. Existem em ferramentas de programação que oferece recursos de escalonamento em nível aplicativo.

Unidade de Trabalho

Em uma ferramenta de programação multithread, as unidades de trabalho representam o cálculo a ser efetuado de forma concorrente no programa. Usualmente estas unidades de trabalho são descritas na forma de **tarefas** ou de **threads**, distinguindo-se quando a granularidade. A ferramenta de programação deve oferecer mecanismos para criar e sincronizar o término destas unidades de cálculo de forma dinâmica, bem como permitir a parametrização de dados na criação e retorno de resultados. Deve também oferecer mecanismos de coordenação, para que as unidades de trabalho possam evoluir de forma coordenada.

Questões de programação

THREADS

Definidas em termos de função, são oferecidas por ferramentas de programação em que cada unidade de trabalho definida é instanciada em um novo fluxo de execução manipulado pelo SO.

TASK

Possuem escopo próprio e acessam o escopo do bloco que o envolve. A parametrização pode descrever a semântica de acesso ao escopo envolvente. Visualiza memória estática, heap e pilha da função que envolve. Não é possível o compartilhamento de dados instanciados no escopo de uma tarefa.

Comunicação

De forma geral, a comunicação se dá por leitura e escrita de dados na memória. No entanto, embora todos os endereços possam ser acessados por qualquer unidade de trabalho, quando em execução, deve ser considerado que a variável alocada naquele endereço esteja com escopo válido para todas as unidades de execução envolvidas. Também é possível parametrizar as unidades de trabalho, na sua criação, e obter resultados ao seu final.

Atenção aos dados no heap!

THREADS

Possuem escopo próprio de dados alocados na sua pilha. Acessam a memória estática e o heap. A parametrização, e o retorno de resultados, segue as regras impostas pela linguagem de programação. Embora possível, não é recomendável o compartilhamento de dados instanciados na pilha.

Questões de programação

TASK

É comum que as ferramentas de programação explorem uma estrutura fixa de programa quando oferecem tarefas como unidade de trabalho. Esta estrutura permite abstrair a identificação individual das tarefas e facilita as decisões de escalonamento aplicativo. Mecanismos de controle de avanço não são muito utilizados, devido a natureza do modelo N *x* M.

Sincronização

Permite cadenciar a evolução das unidades de trabalho, com vistas a promover a correta comunicação de dados entre estas. Os mecanismos de sincronização permitem que as unidades de trabalho tenham conhecimento de seu estado, podendo, então, acessar dados em operações de leitura e escrita. A criação e o término de unidades de trabalho consistem em operações de sincronização e embutem o transporte de informação (parâmetros de entrada ou retorno de dados). Outros mecanismos promovem execuções em regime de exclusão mútua e controle de avanço na execução.

THREADS

Usualmente, threads são identificadas individualmente, não existindo uma estrutura fixa de programa. Portanto, operações de sincronização com o término de threads devem identificar claramente qual thread está sendo sincronizado. A flexibilidade do modelo 1 *x* 1 permite que sejam explorados mecanismos de sincronização para controle do avanço de execução.

Mãos à obra

Como são as ferramentas de programação apresentam recursos de programação?

- Bibliotecas de serviços ou classes
- Esqueletos
- Extensão a linguagens
- Novas linguagens

Novas abstrações de programação!

Manipulação de UTs

PARBEGIN / PAREND

Todos as instruções no bloco PARBEGIN/PAREND são uts independentes. Todas uts estão prontas para executar no início do bloco e o bloco somente é terminado quando todas as uts terminarem.

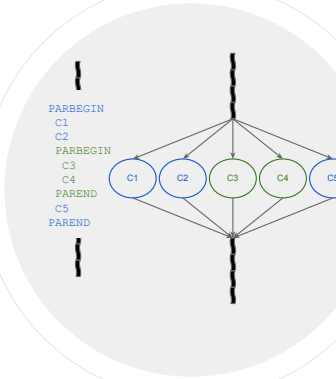
FORK / JOIN

O FORK inicia uma nova unidade de trabalho, concorrente com a sequência da unidade de trabalho atual. O JOIN bloqueia a unidade de trabalho atual, aguardando o final de outra

PARBEGIN / PAREND

Sendo um bloco de comando, possui seu escopo próprio de nomes, sendo submetido às regras gerais da linguagem de programação para acesso aos demais escopos.

É possível parametrizar blocos PARBEGIN/PAREND, identificando níveis de visibilidade aos identificadores dos escopos envolventes. Estes níveis de visibilidade determinam as semânticas de acesso aos dados.

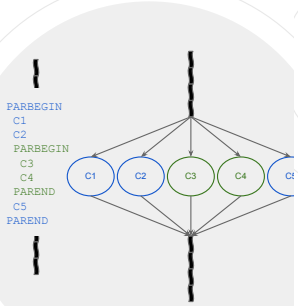


PARBEGIN / PAREND

Este modelo também é chamado COBEGIN / COEND

As uts não são identificadas individualmente e existe uma estrutura de aninhamento, com alto grau de sincronismo na criação e término das atividades. O alto grau de paralelismo promove a adoção desta abstração básica em implementações do tipo N *x* M.

O compartilhamento de dados entre as tarefas "irmãs" somente pode se dar quando o dado estiver no escopo envolvente.

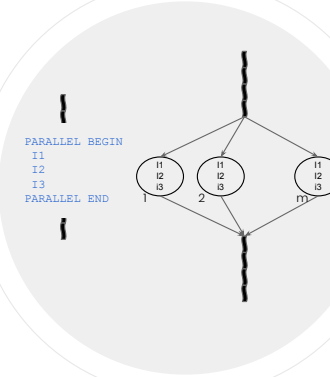


PARBEGIN / PAREND

Variações:

PARALLEL

Como variação deste modelo, existe o bloco PARALLEL. Neste caso, todo o conjunto de instruções do bloco compõem o código de uma única ut. Esta ut é instanciada tantas vezes quanto forem os recursos de processamento disponíveis.



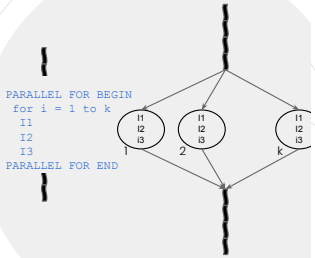
Supondo uma arquitetura com m CPUs.

PARBEGIN / PAREND

Variações:

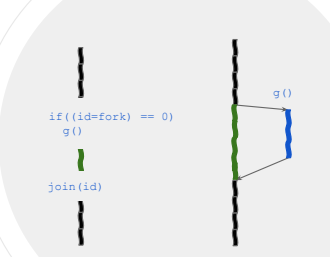
PARALLEL FOR

Trata-se de uma variação ao próprio bloco PARALLEL onde o corpo da ut é definido pelo conjunto de comandos executados no corpo de um laço, sendo instanciadas tantas uts quanto forem o número de iterações a serem executadas.



FORK / JOIN

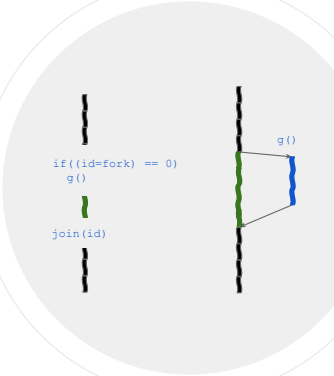
Originalmente, esta primitiva permite a criação de uma nova ut com o comando FORK, a qual seguirá sua execução de forma concorrente com a ut original. Neste modelo de execução, todo o espaço de memória é replicado e a nova ut passa a executar a partir do ponto onde o FORK foi executado. A ut original reconhece, por um retorno da primitiva FORK a identificação da ut criada para futuramente realizar um JOIN sobre ela.



FORK / JOIN

Originalmente, esta primitiva permite a criação de uma nova ut com o comando FORK, a qual seguirá sua execução de forma concorrente com a ut original. Neste modelo de execução, todo o espaço de memória é replicado e a nova ut passa a executar a partir do ponto onde o FORK foi executado. A ut original reconhece, por um retorno da primitiva FORK a identificação da ut criada para futuramente realizar um JOIN sobre ela.

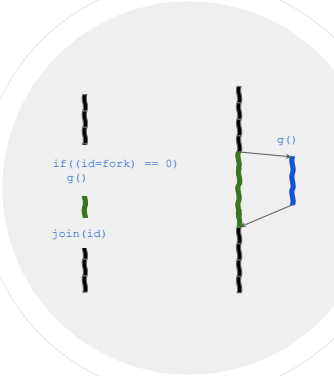
No modelo original, pela duplicação da imagem dos processos, não havia compartilhamento de espaço de endereçamento em nível de processo. A parametrização, no entanto, era possível (por cópia).



FORK / JOIN

As variantes na programação multithread prevê o compartilhamento de espaço de endereçamento do processo e, consequentemente, são aplicáveis as mesmas regras de escopo da linguagem de programação original.

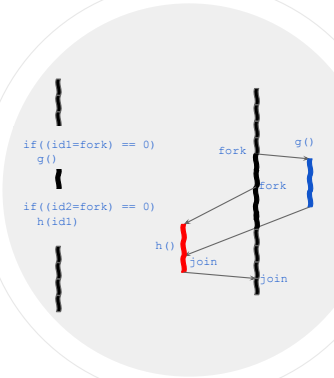
Variantes: uts identificáveis individualmente, uts anônimas.



FORK / JOIN

Variante: uts identificáveis individualmente

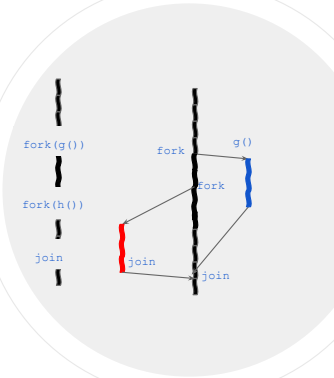
Neste caso, o identificador de uma ut é um dado, armazenado em memória. Qualquer ut que tenha acesso a este dado é capaz de realizar a operação JOIN. Consequência, a estrutura de execução não pode ser conhecida a priori.



FORK / JOIN

Variante: uts anônimas

Como no caso anterior, a primitiva FORK lança uma nova ut. No entanto, por não haver identificação individual das uts, a primitiva JOIN sincroniza a ut corrente com todas as uts que ela tenha criado, em uma estrutura de FORK / JOIN aninhado. Previsível, portanto.



FORK / JOIN

Comentários

Sinônimos: create/join, spawn/sync

Quando a estrutura de concorrência pode ser prevista, uma heurística de escalonamento pode promover alguma estratégia de execução que otimize algum índice de desempenho.

Possibilidade de uma ut ser submetida a um único ou a múltiplos joins.

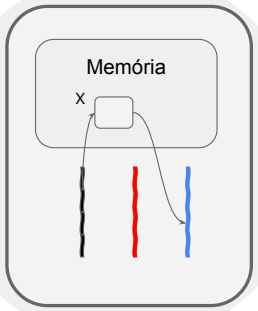
Comunicação

A troca de informação entre uts ocorre, essencialmente, com leituras e escritas em um espaço de endereçamento comum, representado pela memória do processo.

Questão a considerar: escopo e vínculo de endereço da variável.

O compartilhamento de dados não é viável no seguintes casos:

- Quando o dado é criado no escopo da própria ut
  - Tasks: o escopo é local ao bloco
  - Threads: o vínculo de endereço é efêmero, limitado à validade da pilha própria ao thread



Comunicação

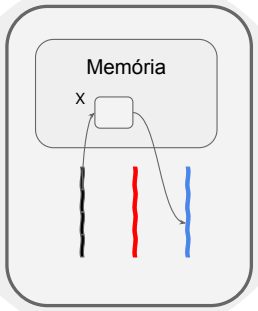
A troca de informação entre uts ocorre, essencialmente, com leituras e escritas em um espaço de endereçamento comum, representado pela memória do processo.

Questão a considerar: escopo e vínculo de endereço da variável.

O compartilhamento é viável quando o dado estiver no escopo da ut e seu vínculo de endereço for estável. Isto corresponde às seguintes situações em que o dado se encontra na área de memória:

- Estática, uma vez que seu vínculo de endereço será perdido somente quando o programa terminar;
- Dinâmica, considerando que tenha sido alocado mas ainda não desalocado.

Estes aspectos devem ser considerados levando em conta as características da linguagem de programação. Por exemplo, em uma linguagem orientada a objetos, as regras de escopo estão sujeitas às questões inerentes a este paradigma de programação.



Comunicação

PARBEGIN / PAREND

Devem ser observadas as regras de escopo aplicadas aos blocos. Existe a possibilidade de parametrizar as uts criadas limitando a semântica de acesso aos dados dos escopos envolventes. Não é possível o compartilhamento de dados criados por uts no mesmo nível.

Diferentes estratégias de comunicação de dados também devem levar em conta o tipo de ut disponibilizada pela ferramenta de programação.

É importante ressaltar que as informações aqui contidas apenas destacam características exploradas da linguagem de programação base. A programação multithread, no caso, apenas destaca a importância de considerar tais aspectos.

FORK / JOIN

O lançamento de uma nova função, como código base de uma ut, implica na instanciação de uma nova pilha, para alocação dinâmica de dados na pilha. Embora os endereços sejam visíveis, dados alocados na pilha tem seu vínculo de endereço garantido apenas enquanto esta função estiver executando. Não sendo viável o compartilhamento de seu endereço.

Comunicação

PARBEGIN / PAREND

```
int x, y, z;

PARBEGIN ERWE(x), CRCW(y)
...
PAREND
```

Exemplos

O identificador `x`, em cada ut, refere-se à variável instanciada localmente à ut, podendo existir uma semântica de atualização da variável original;

O identificador `y`, em cada ut, refere-se a variável original, instanciada no escopo envolvente, podendo existir uma semântica de atualização desta;

A ut `errado` retorna um endereço na pilha, a qual é dissolvida ao término da função envolvente;

A ut `certo` retorna um endereço alocado no heap.

FORK / JOIN

```
void* errado() {
 int ret;
 ...
 return &ret;
}
void* certo() {
 int *ret;
 ret = malloc(int)
 ...
 return ret;
}
```


Sincronização

São mecanismos que permitem controlar a execução das uts, permitindo que avancem ou contingenciando-as, em função do estado de outras uts. O objetivo destes mecanismos pode ser garantir execução em **regime de exclusão mútua** ou apenas **controlar o número** de uts que podem avançar sua execução sob uma determinada situação.

Sincronização

O regime de **exclusão mútua** prevê que apenas uma ut pode avançar sua execução para explorar o uso de um determinado recurso.

Lembrando: dado compartilhado em memória é um recurso.

A ideia dos mecanismos promovendo exclusão mútua é garantir o avanço da execução sobre o recurso quando o direito de execução for obtido.

O Problema

```
int saldo = 313
```

ut A	ut B
`...`	`...`
`saldo = saldo + 1`	`saldo = saldo - 1`
`...`	`...`

Sincronização

O regime de **exclusão mútua** prevê que apenas uma ut pode avançar sua execução para explorar o uso de um determinado recurso.

Lembrando: dado compartilhado em memória é um recurso.

A ideia dos mecanismos promovendo exclusão mútua é garantir o avanço da execução sobre o recurso quando o direito de execução for obtido.

O Problema

```
int saldo = 313
```

ut A	ut B
`...`	`...`
`saldo = saldo + 1`	`saldo = saldo - 1`
`...`	`...`

Sincronização

MUTEX

Abstração mais conhecida, e antiga, para controlar o acesso a uma seção crítica. Trata-se de uma estrutura de dados que aceita duas operações: *lock* e *unlock*. A semântica da operação lock garante que, no seu retorno a ut tem a posse do mutex. Sua liberação se dá com a execução de um unlock sobre a mesma estrutura de dados.

O uso de uma estratégia para garantir exclusão mútua visa promover execuções seguras de **seções críticas**. Uma seção crítica é um conjunto de instruções que acessam recursos compartilhados. O regime de exclusão mútua garante consistência na manipulação de ordem.

Mecanismos de exclusão mútua não garantem ordem de execução.

MONITOR

Consiste em uma abstração de mais alto nível, na qual um conjunto de operações são associada a um dado, sendo garantido que apenas uma destas operações pode estar ativa em um determinado instante de tempo.

Sincronização

MUTEX

Abstração mais conhecida, e antiga, para controlar o acesso a uma seção crítica. Trata-se de uma estrutura de dados que aceita duas operações: *lock* e *unlock*. A semântica da operação lock garante que, no seu retorno a ut tem a posse do mutex. Sua liberação se dá com a execução de um unlock sobre a mesma estrutura de dados.

A tentativa de realizar um lock sobre um mutex sob posse de outra ut ou a tentativa de executar uma operação em um monitor que esteja, naquele momento, atendendo outra demanda implica em bloqueio da ut em questão até que o recurso seja disponibilizado.

MONITOR

Consiste em uma abstração de mais alto nível, na qual um conjunto de operações são associada a um dado, sendo garantido que apenas uma destas operações pode estar ativa em um determinado instante de tempo.

Sincronização

MUTEX

```
int   saldo = 313
mutex m
```

ut A	ut B
...	...
lock(m)	lock(m)
saldo = saldo + 1	saldo = saldo - 1
unlock(m)	unlock(m)
...	...

A tentativa de realizar um lock sobre um mutex sob posse de outra ut ou a tentativa de executar uma operação em um monitor que esteja, naquele momento, atendendo outra demanda implica em bloqueio da ut em questão até que o recurso seja disponibilizado.

MONITOR

```
monitor conta {
 int saldo = 313;
 inc() : saldo = saldo + 1
 dec() : saldo = saldo - 1
}

conta c
```

ut A	ut B
...	...
c.inc()	c.dec()
...	...

Sincronização

MUTEX

```
int   saldo = 313
mutex m
```

ut A	ut B
...	...
lock(m)	lock(m)
saldo = saldo + 1	saldo = saldo - 1
unlock(m)	unlock(m)
...	...

Observe que, no uso do mutex, o próprio mutex é um dado compartilhado, estando sujeito, portanto, às mesmas regras de escopo. Também deve se notar que não existe nenhum vínculo *no código* do mutex m com o dado saldo. A associação entre o mutex m e o recurso saldo faz parte do algoritmo.

MONITOR

```
monitor conta {
 int saldo = 313;
 inc() : saldo = saldo + 1
 dec() : saldo = saldo - 1
}

conta c
```

ut A	ut B
...	...
c.inc()	c.dec()
...	...

Sincronização

MUTEX

```
int   saldo = 313
mutex m

        ut A                    ut B
...                     ...
lock(m)                 lock(m)
saldo = saldo + 1       saldo = saldo - 1
unlock(m)               unlock(m)
...                     ...
```

Monitores são uma abstração de mais alto nível. O maior esforço de programação resulta em uma maior robustez no uso dos mecanismos de exclusão mútua. Sua estrutura, conceitual, assemelha-se ao uso das classes na programação orientada a objetos.

MONITOR

```
monitor conta {
 int saldo = 313;
 inc() : saldo = saldo + 1
 dec() : saldo = saldo - 1
}

conta c

   ut A                 ut B
...                     ...
c.inc()                 c.dec()
...                     ...
```


Sincronização

Mecanismos de **controle de fluxo** permitem contingenciar o grau de paralelismo explorado em função de algum critério dado pela aplicação.

Observando: o controle de fluxo controla o número de uts avançando sobre um trecho de código controlado, mas não garante acesso em regime de exclusão mútua.

O princípio utiliza um contador de passes. Quando incrementado, disponibiliza um novo passe. Quando decrementado, significa que uma ut adquiriu um passe e tem o direito de avançar sobre o trecho de código controlado. Caso o contador esteja zerado, não é permitido o avanço e as uts que solicitarem passe permanecem bloqueadas.

O Problema

```
int   buff[TAM]
int   nbelem
mutex m

    ut Produtora                ut Consumidora

while( notEnd ) {           while( notEnd ) {
 lock(m)                     lock(m)
 insereElem(buff)            retiraElem(buff)
 ++nbelem                    --nbelem
 unlock(m)                   unlock(m)
}                           }
```


Sincronização

Proberen: P

Acessa o contador de passes, solicitando o direito de entrar em um trecho de código controlado, havendo um passe disponível, adquire um, sendo decrementado o contador de passes disponíveis. Não havendo passes disponíveis, bloqueia até que algum passe seja disponibilizado.

A solução é dada pelo uso de um semáforo. Este mecanismo possui um *contador de passes*. O número de passes neste contador informa o limite de uts que podem avançar, simultaneamente, sobre um trecho de código controlado. Além da operação de inicialização, que informa o número de passes inicialmente disponibilizados, existem duas operações: a operação P e a operação V. A operação equivale a um pedido de passe, bloqueando a ut caso um passe não esteja disponível no momento. A operação V consiste na liberação de um passe.

Verhogen: V

Ação de liberar um passe, informando a saída de uma ut de um trecho de código controlado. Um passe é adicionado ao contador de passes. Não existe um limite superior ao número de passes que podem ser disponibilizados

Do holandês: *proberen* (testar) e *verhogen* (incrementar).
Quem deu esses nomes? Edsger Dijkstra

Sincronização

O algoritmo ao lado controla o acesso a um buffer compartilhado por uts produtoras e consumidoras. O buffer tem tamanho TAM. Um semáforo é controla o acesso das uts consumidoras ao buffer de forma que somente é retirado algum dado do buffer se existir algum dado no buffer para ser retirado. O semáforo é iniciado com 0 (zero), indicando buffer vazio. A cada elemento inserido no buffer, um novo passe é entregue para os consumidores.

A Solução

```
int      buff[TAM]
int      nbelem
mutex    m
semaforo s(0)

    ut Produtora                ut Consumidora
while( notEnd ) {           while( notEnd ) {
 lock(m)                     p(s)
 insereElem(buff)            lock(m)
 ++nbelem                    retiraElem(buff)
 unlock(m)                   --nbelem
 v(s)                        unlock(m)
}                           }
```

Atividade de Acompanhamento

1) Com auxílio de um segundo semáforo, modifique o código ao lado para impedir que as uts produtoras insiram elementos acima da capacidade do buffer.
2) O mutex é necessário (sim...) justifique.
3) O mutex não é necessário para controlar acesso ao semáforo s. Justifique.

Acesse o formulário em:
https://forms.gle/usd5vJ1DjSBugXww6

A Solução

```
int      buff[TAM]
int      nbelem
mutex    m
semaforo s(0)
```

ut Produtora

```
while( notEnd ) {
 lock(m)
 insereElem(buff)
 ++nbelem
 unlock(m)
 v(s)
}
```

ut Consumidora

```
while( notEnd ) {
 p(s)
 lock(m)
 retiraElem(buff)
 --nbelem
 unlock(m)
}
```


Dicas Quentes

1. Defina a granularidade da ut em função das características de seu sistema computacional;
2. Compreenda as heurísticas de escalonamento empregadas pela ferramenta antes de desenvolver o algoritmo;
3. Evite o uso de dois ou mais recursos protegidos simultaneamente;
4. Caso necessário, elabore uma estratégia de aquisição;
5. Observe as regras de escopo e vínculo de endereço dos dados;
6. Prefira um código seguro do que utilizar uma otimização duvidosa;
7. Ao iniciar a implementação de um trecho de código concorrente, não realize nenhuma pausa até terminá-lo.

Este slide está em construção (permanente)