Kernel Synchronization
Sistemas Operacionais

Rafael de Moraes S. Fernandes

rafaelmsf@cos.ufrj.br

Motivação

- Um simples exemplo de retirada de dinheiro

```
int total = pegue_total_da_conta(); //Saldo disponível
int pedido = pegue_total_pedido(); //Quanto cliente pediu
if(total < pedido) erro("Saldo insuficiente");
//Se tem fundos
total -=pedido;
atualiza_conta(total);
fornece_dinheiro(pedido);
```

Motivação (cont.)

- Aparentemente, código perfeito, mas...
- Kernel é preemptivo -> Escalonador pode retirar o processo em execução **a qualquer** momento e colocar outro.
- Então qual o resultado do código anterior?
- Não se pode afirmar, pois:
- Cliente quer retirar R$100,00 e Banco quer descontar R$10,00. Saldo:R$105,00.

Motivação(cont)

- Caso cliente retire o dinheiro primeiro que o banco, teremos:

Tempo	Cliente	Banco
1	total=R$105;	
2	pedido=R$100;	
3	if -> OK;	
4	total=R$5,00;	
5		total=R$105,00
6		pedido=R$10,00
7		if->OK
8		total=R$95,00
9	atualiza(total);	
10		atualiza(total);

Motivação(cont)

- Resultado da execução anterior:

  DINHEIRO “GRÁTIS”

Introdução

- O código do Kernel precisa de proteção -> Recursos compartilhados necessitam de proteção.
- Duas “escritas” na mesma posição de memória compartilhada ao mesmo tempo -> Sobrescreve
- A partir do Kernel 2.0 -> Multiprocessamento Simétrico -> Pode-se executar 2 ou mais processos mesmo tempo.

Regiões Criticas e Condições de Corrida

- Regiões Críticas -> Partes do código que acessam e manipulam recursos compartilhados;
- Prevenir acesso concorrente -> Código região crítica executado Atomicamente – Instruções indivisíveis.
- Condições de Corrida -> Duas Threads (Processos) executarem simultaneamente disputando mesmo recurso.
- Sincronização -> Assegurar que Condições de Corrida e que concorrencias fora da Região Crítica não corram.

Locking

- Garante que apenas uma Thread manipule os dados compartilhados no mesmo momento.
- Funciona como uma Fechadura.
- Após uma Thread entrar numa “Sala” (dados compartilhados), a fechadura é trancada.
- Só é reaberta quando a Thread acaba a execução na região crítica.

Locking

- Lock previne a concorrência:

Thread 1	Thread2
1- Tentando entrar na sala	Tentando entrar na sala
2- Sucesso: “Trava” obtida	Falha: Esperando....
3- Executando	Esperando
4- “Destravando”	Esperando
5-	Sucesso: “Trava” obtida
6-	Executando
7-	“Destravando”

Locking

- Em Linux existem algumas maneiras para dar um “LOCK” em uma Thread.
- Uma maneira é fazer a Thread fora da região Crítica ficar dando LOOPs;
- Outra maneira é colocar o Processo em SLEEP até que a Região Crítica seja liberada.

Locking

- Mas, como o processo de locking é código, o que garante que não haja condição de corrida neste código?
- O código do Locking é desenvolvido com instruções atômicas (indivisíveis).
- Este desenvolvimento é específico da arquitetura.

Afinal, o que causa Concorrência?

- User-space -> Programas são escalonados preemptivamente.
- É possível que um processo seja retirado no meio da execução na Região Crítica.
- Outro processo pode acessar a mesma Região Crítica do processo anterior -> Condição de Corrida.

Afinal, o que causa Concorrência?

- Pseudo-Concorrência -> Ocorre quando dois processos não executam ao mesmo tempo, mas compartilham os mesmos recursos.
- Verdadeira Concorrência -> Ocorre em Multiprocessamento Simétrico, onde dois processos podem executar ao mesmo tempo,compartilhando o mesmo recurso.

Afinal, o que causa Concorrência?

- O Kernel também causa concorrência, são elas:

1. Interrupções -> Pode ocorrer assincronamente, interrompendo o processo em execução
2. Preempção do Kernel -> Uma tarefa no Kernel pode retirar outra da execução

Afinal, o que causa Concorrência?

3. *Sleeping e Synchronization with user-space* -> Uma tarefa pode ir para Sleep e invocar o escalonador, resultando na execução de um novo processo.
4. Multiprocessamento Simétrico -> Dois ou mais processadores podem executar o código do Kernel ao mesmo tempo.

Como saber o que necessita de Proteção

- Pensando em simples perguntas podemos responder:

1. O dado é Global? Pode uma outra Thread acessar o dado que a thread atual está acessando?
2. O dado é compartilhado entre contexto de processos e contexto de interrupção? É compartilhado entre 2 tratamentos de interrupção?

Como saber o que necessita de Proteção

3. Se um processo é retirado do processador enquanto está acessando um dado compartilhado, o novo processo acessa este dado compartilhado?
4. Se um processo entrar no modo SLEEP, que estado ele deixa os dados compartilhados?
5. O que previne o dado de ser liberado (unlocked) for a da execução do processo atual?

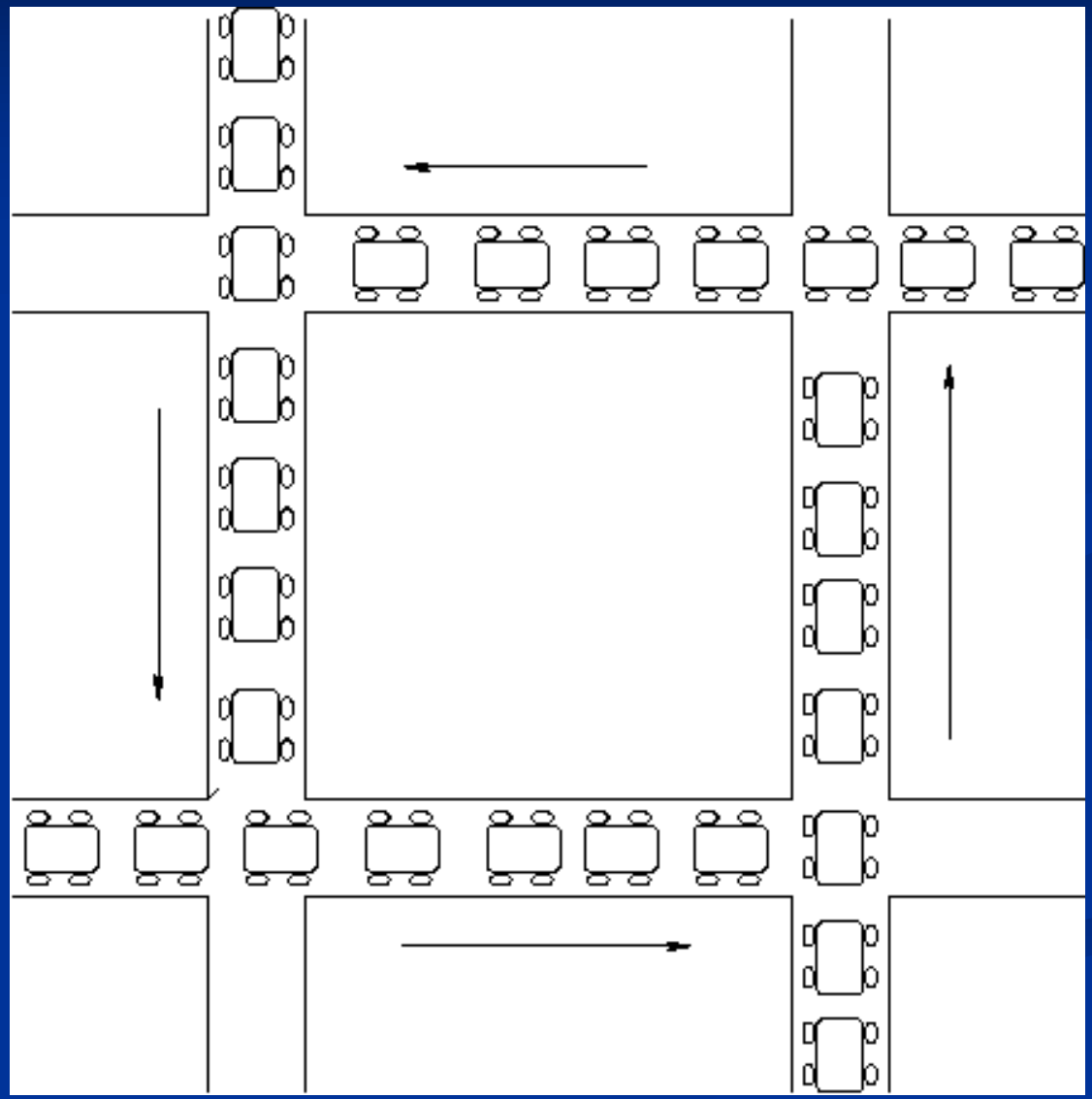
Como saber o que necessita de Proteção

6. O que acontece se este processo for chamado por outro processador?
7. O que será feito sobre isso?

Deadlocks

- Condição em que uma ou mais Threads necessitam de um ou mais recursos e todos os recursos estão alocados.
- Todas as Threads ficam esperando o recurso ser liberado pelas outras Threads, ficando em uma espera interminável.
- Portanto nenhuma das Threads pode prosseguir, causando um Deadlock.

Deadlock



Deadlock

Thread1	Thread 2
Pegou recurso1	Pegou recurso2
Tentando obter recurso2	Tentando obter recurso1
Esperando recurso2	Esperando recurso1

- Este exemplo é chamado “deadly embrace” ou “ABBA deadlock”

Deadlock

- Prevenção de Deadlock é muito importante
- Algumas regras de prevenção:

1. A ordenação dos locks é muito importante. Locks no mesmo código devem ser obtidos sempre na mesma ordem.

Deadlock

- Exemplo:
- Se os recursos forem, na ordem: Abacate, Laranja e Uva.
- Se deixarmos uma Thread obter os recursos fora desta ordem podemos ter um Deadlock:

Thread1	Thread 2
Pegou Abacate	Pegou Uva
Pegou Laranja	Tentando obter Abacate
Tentando obter Uva	Esperando Abacate
Esperando Uva	

Deadlock

- Agora se os recursos somente puderem ser obtidos na ordem, não teremos Deadlock.

- A ordem de destravar (unlock) não é importante, mas é uma prática comum destravar na ordem inversa.

Deadlocks

2. Pergunte se este código sempre termina (prevenção de Starvation).
3. Não permita adquirir a mesma chave (lock) duas vezes.
4. Faça o código do locking simples. Complexidade neste código pode gerar Deadlocks.

Escalabilidade e Disputa

- Se um lock está em uso e outra Thread está tentando adquirir o lock -> Temos uma condição de Disputa;
- Caso haja mais de uma Thread tentando adquirir o lock temos uma Grande Disputa;
- A Grande Disputa pode ocorrer porque o lock é bastante requerido ou porque demora muito tempo para ter o unlock

Escalabilidade e Disputa

- Lock diminui a vazão do sistema.
- Grande Disputa pode se tornar um funil no sistema

Escalabilidade e Disputa

- Escalabilidade -> Medida de quão bom o sistema pode ser expandido;