COPPE UFRJ

GTA / UFRJ

Comunicação entre processos (2)

Pedro Cruz

EEL770 – Sistemas Operacionais

Lembretes

GTA / UFRJ

- Proposta de trabalho
  - Enviar para cruz@gta.ufrj.br
    - Prazo: dia 02 de abril

Aula passada

GTA / UFRJ

- Comunicação entre processos é útil
  - Troca de dados
  - Consistência
  - Sincronização
- Problemas e soluções parecidos
  - Processos
  - *Threads*
  - Dispositivos em sistemas distribuídos

Aula passada

GTA / UFRJ

- Condição de corrida
- Região crítica
- Exclusão mútua
- Espera ocupada
  - Desabilitar interrupções
  - Variáveis do tipo trava
  - Variáveis do tipo "vez"
  - Solução de Peterson
  - Instrução Testar e Configurar Trava
- Dormir e acordar
- Semáforo

Aula passada

GTA / UFRJ

- Implementação de comunicação
  - Área de memória compartilhada
    - Problema de leitura e escrita não-atômica
  - Instruções especiais de processador
    - Necessita de suporte do *hardware*
  - Semáforos
    - Implementados como chamadas de sistema

Semáforos

GTA / UFRJ

- Variável "compartilhada" diz quantos recursos estão disponíveis
  - Operação de verificar variável, alterar variável e dormir são atômicas
    - Implementadas como chamadas de sistema
      - *Up*
      - *Down*

Up e *Down*

GTA / UFRJ

- *Up*
  - Verifica variável
    - Se possível, incrementa e segue para execução
      - Envia sinal de *wake*
    - Se não, dorme e aguarda sinal
- Down
  - Verifica variável
    - Se possível, decrementa e segue para execução
      - Envia sinal de *wake*
    - Se não, dorme e aguarda sinal

Ocorrem de forma atômica!

Semáforo

GTA / UFRJ

- Exclusão mútua
- Sincronismo

Produtores-consumidores com semáforos


```
mutex = 1          // acesso à região crítica
empty = N          // contador de espaços vazios
full = 0           // contador de espaços preenchidos
```

- Produtor

```
down(&empty);
down(&mutex);
insert(item);
up(&mutex);
up(&full);
```

- Consumidor

```
down(&full);
down(&mutex);
item = remove();
up(&mutex);
up(&empty);
```

Espera ocupada em *threads*

GTA / UFRJ

- Threads em espaço de núcleo
  - Acordadas pelo escalonador
- Threads em espaço de usuário
  - Acordadas pelo processo do usuário

Como ocorre a espera ocupada em ambos os casos?

Mutex

GTA / UFRJ

- Forma simples de garantir exclusão mútua
  - Se pode acessar a região crítica, acessa
  - Se não pode acessar a região crítica, dorme

Mutexes

GTA / UFRJ

```
mutex_lock:
   MOVE Reg, #1          | copia 1 para Reg
   TSL Reg, MUTEX        | troca MUTEX com Reg
   CMP Reg, #0           | compara Reg com 0
   JZE ok                | se for 0, vai para ok
   CALL thread_yield     | mutex ocupado, libere CPU
   JMP mutex_lock        | tente outra vez
ok: RET
mutex_unlock:
   MOVE MUTEX, #0        | copia 0 em MUTEX
   RET                   | Retorna
```

Problemas com semáforos

GTA / UFRJ

- Erros são comuns
  - Ordem das chamadas de “reserva” e “contagem” é crítica
    - Se é feita de outra maneira, pode haver um **impasse**

Produtores-consumidores com semáforos - problema


```
mutex = 1          // acesso à região crítica
empty = N          // contador de espaços vazios
full = 0           // contador de espaços preenchidos
```

- Produtor

```
down(&mutex);
down(&empty);
insert(item);
up(&mutex);
up(&full);
```

- Consumidor

```
down(&full);
down(&mutex);
item = remove();
up(&mutex);
up(&empty);
```

Monitores

GTA / UFRJ

- Abstração "melhor" para os semáforos
  - Recurso é protegido pelo monitor
  - Processos entram numa fila
  - Monitor organiza quais processos devem acessar o recurso
    - Apenas um por vez
  - Monitor oferece interface de programação

Monitor

GTA / UFRJ

- Chamadas
  - `wait(c)`
    - Bloqueia o processo enquanto a condição c for verdadeira
      - Outro processo pode usar o recurso protegido
  - `signal(c)`
    - Acorda o processo aguardando a condição c ser falsa
- Filas para utilização
- Filas para sinalização
- Deve ter suporte da linguagem

Monitor - exemplo

GTA / UFRJ

```
Monitor ProduceConsume {
  insert(item){
    if(count == N) wait(full);  // desculpa =/
    insert_item(item);
    count++;
    if (count == 1) signal(empty);}
  item remove(){
    if (count == 0) wait(empty);
    item = remove_item();
    count--;
    if (count == (N-1)) signal(full);}} //perdão =/
```

Problemas de todos os mecanismos vistos

- Resolvem:
  - Exclusão mútua
  - Sincronismo
- Funcionam
  - Para uma única CPU
  - Para várias CPUs com memória compartilhada

- Não resolvem:
  - Compartilhamento direto de dados
- Não funcionam:
  - Em sistemas distribuídos sem compartilhamento de memória

Troca de mensagens

GTA / UFRJ

- Chamadas de sistema
  - `send(destination, &message);`
  - `receive(source, &message);`
    - Processos devem:
      - Saber como identificar destino e/ou fonte
      - Ter um formato de mensagem estabelecido
    - Mensagens podem ser:
      - Bloqueantes
      - Não-bloqueantes

Troca de mensagem

GTA / UFRJ

- Envios podem gerar reconhecimento
  - *ACK*
- Falta de reconhecimento pode gerar reenvio
  - Indica perda de mensagem
- Reenvio pode gerar recebimento duplicado
  - Pode gerar problemas na aplicação
- Solução é numerar as mensagens e rejeitar mensagens duplicadas

Paradigmas de troca de mensagem

- Um para um
  - Endereçamento direto
  - Caixa postal
- Um para muitos
  - *Broadcast*
- Muitos para um
  - Cliente-servidor
- Muitos para muitos
  - Quadro negro

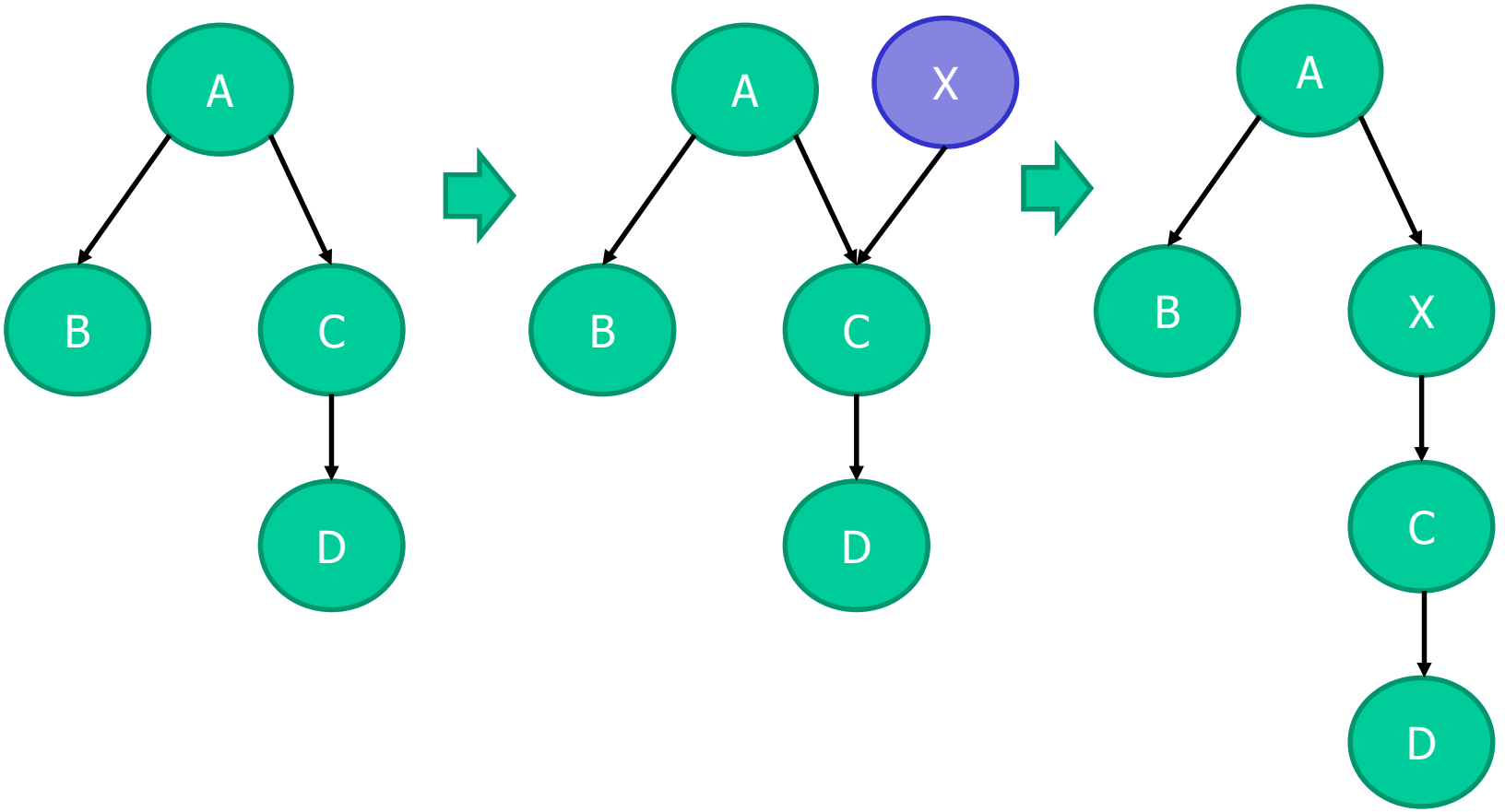
Barreira

- Processos concorrentes impõe uma barreira
  - Processos que alcançam a barreira são bloqueados
  - Processos são liberados quando todos alcançam a barreira

Leitura-cópia-atualização

GTA / UFRJ

- Não permitir a leitura de dados durante alteração
  - Leitor acessa dados anteriores até alteração completa



COPPE UFRJ

GTA / UFRJ

Comunicação entre processos (2)

Pedro Cruz

EEL770 – Sistemas Operacionais