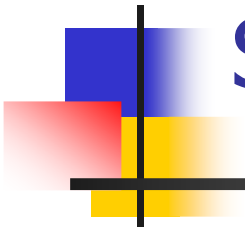
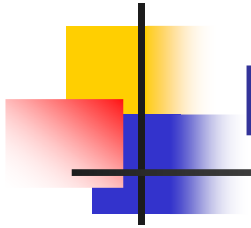


Sistemas Operacionais

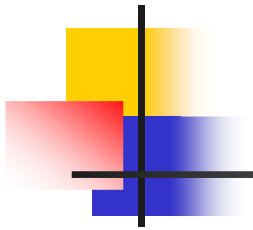
Unidade Três

Prof. Flávio Márcio de Moraes e Silva



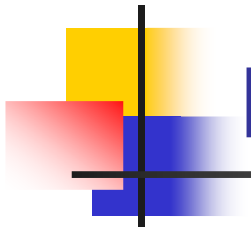
Programação Concorrente

- Programa Seqüencial: executado por apenas um processo.
- Programa Concorrente: executado simultaneamente por diversos processos que cooperam entre si, isto é, trocam informações. (Programação Paralela)
- O verbo "concorrer" em português possui vários sentidos como "cooperar", "disputa ou competição" e "existir simultaneamente". Todos estes sentidos são aplicáveis para a programação concorrente.
  - João providenciou as bebidas concorrentemente Maria fez a comida, para o jantar.
  - João concorreu com Maria nos jogos de matemática.
  - João e Maria viveram em épocas concorrentes.



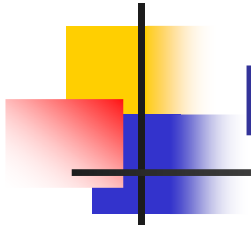
Motivação

- Aumento de desempenho:
  - Permite a exploração do paralelismo real disponível em máquinas multiprocessadoras
  - Sobreposição de operações de E/S com processamento
- Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínseco



Desvantagens

- Programação concorrente mais complexa que a seqüencial.
- Problemas com: interação entre processos, velocidade dos processos e situações de escalonamento dos processos.
- Erros difíceis de serem encontrados e reproduzidos.

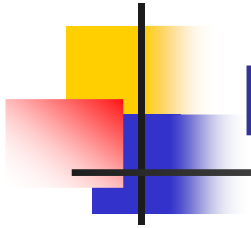


Exemplo – Processo Impressor

- Seqüencial:
- Arquivo -> Processo -> Impressora
- Problema: enquanto o processo esta lendo arquivo do HD a impressora esta parada e vice-versa.

- Solução concorrente:
- Arquivo->Processo Leitor->Buffer
- Buffer->Processo Impressor->Impressora

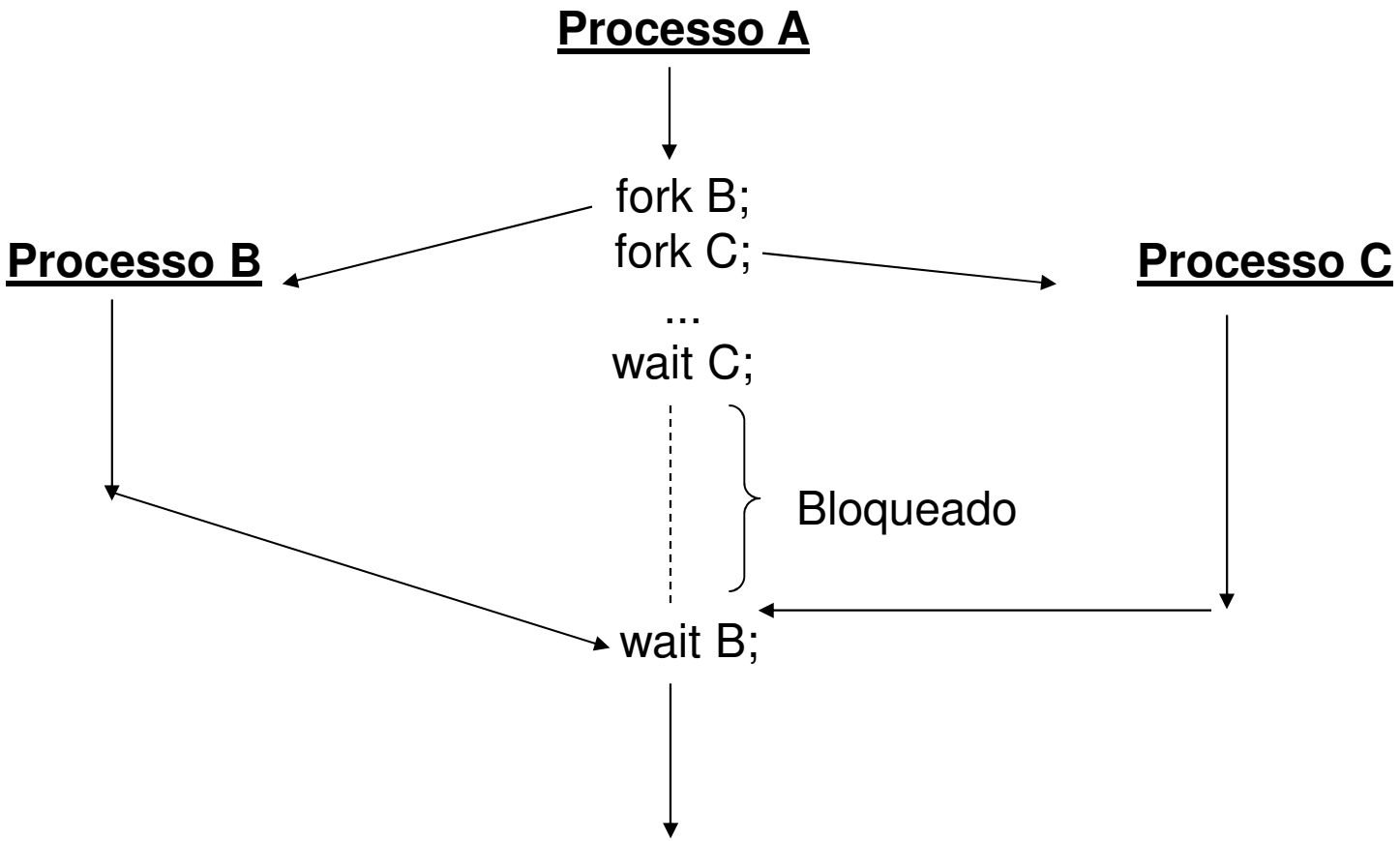
- OBS: Buffer região compartilhada entre processos



Especificação do Paralelismo

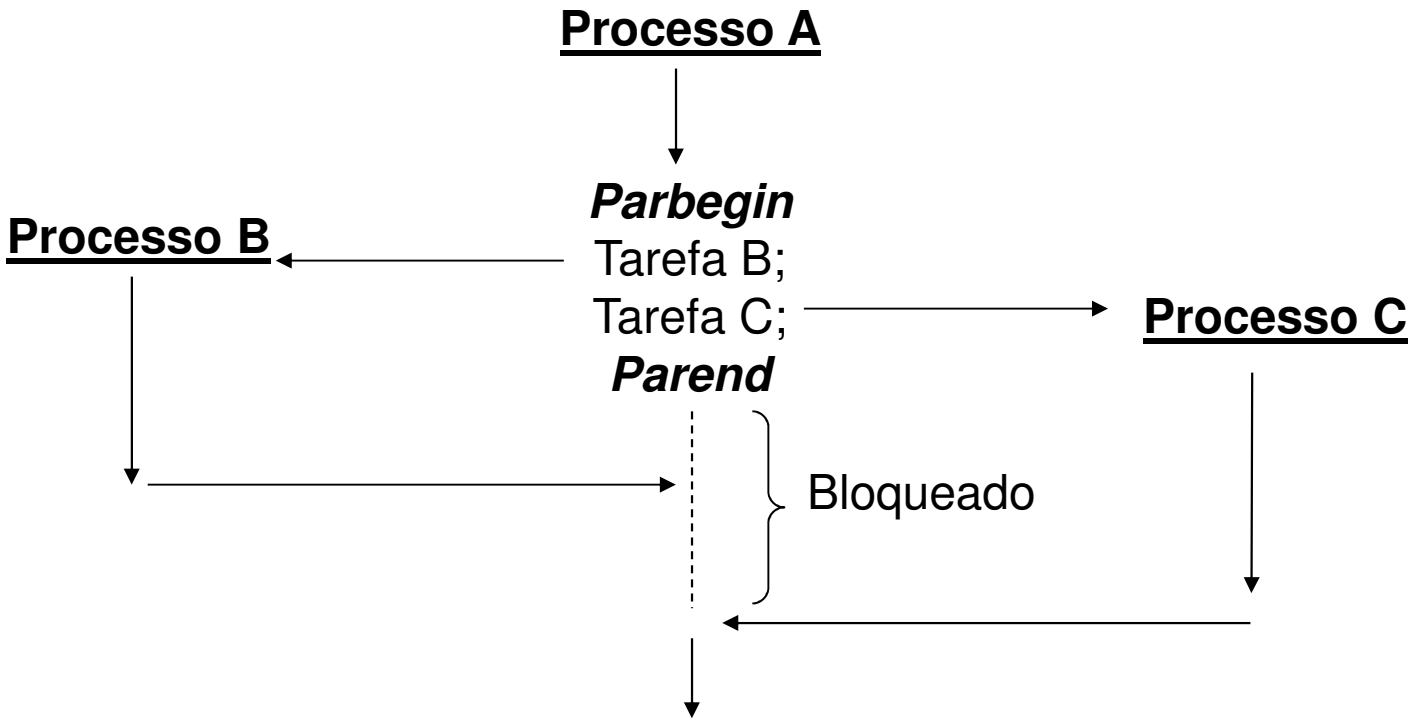
- Quantos processos participarão?
- Quem fará o que?
- Dependência entre tarefas (pode-se ilustrar com grafos)
- Notação para especificar paralelismo:
  - Fork\Wait
  - ParBegin\ParEnd

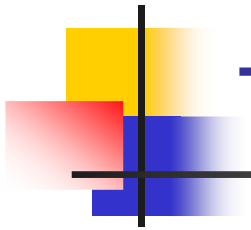
Fork/wait


Processo A
fork B;
fork C;
...
wait C;
Processo B
Processo C
Bloqueado
wait B;

Parbegin/parend

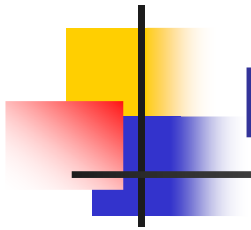
- Comandos empregados para definir uma sequência de comandos a serem executados concorrentemente
- A primitiva *parend* funciona como um ponto de sincronização (barreira)





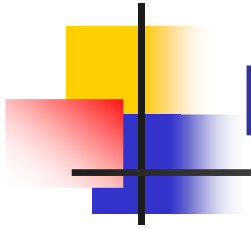
Troca de informação entre processos

- Existem três formas básicas de processos trocarem informações:
  - Memória compartilhada
  - Mensagens
  - Arquivos (O emprego de arquivos para comunicação entre processos requer o uso de técnicas de Banco de Dados e não será abordado nesta disciplina. Abordado na disciplina BDII.)



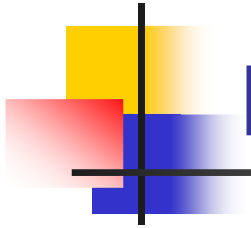
Regiões Críticas

- Região crítica
  - Parte do código em que há acesso à memória compartilhada
- Condição de disputa\corrida: vários processos manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os processos são feitos. Ex: dois processos editando um mesmo arquivo texto.
- O que fazer para evitar condições de disputa\corrida?
  - **Exclusão mútua:** assegurar que processos sejam impedidos de usar uma variável ou arquivo compartilhado que já estiver em uso por outro processo.



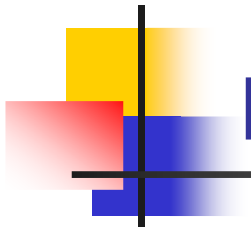
Exemplo Anterior – Seção Crítica

- Processo leitor – completa a leitura do arquivo do HD e salva o nome do mesmo na fila a ser lida pelo processo impressor.
- Processo impressor – lê o nome do próximo arquivo a ser imprimido.
- Seção crítica – fila de nomes de arquivos



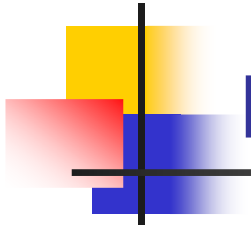
Regiões Críticas

- Uma boa solução deve satisfazer:
  1. Nunca dois processos podem estar simultaneamente em suas regiões críticas
  2. Nenhum processo executado fora da sua região crítica pode ser bloqueado por outros processos
  3. Nenhum processo deve esperar eternamente para entrar em sua região crítica
  4. Nada pode ser afirmado sobre o número e a velocidade dos processos ou sobre o número de CPUs
  5. Se um processo P deseja entrar na região crítica e nenhum outro processo esta usando a mesma, o processo P não pode ser impedido de entrar



Exclusão Mútua - Mecanismos básicos

- **Desabilitando interrupções**
  - O processo desabilita todas as interrupções logo após entrar em sua região crítica e reabilita-as antes de sair
- Problema dessa abordagem
  - Processos de usuário podem desabilitar interrupções e nunca reabilitá-las.
  - Rompimento grave na segurança do SO. Somente o núcleo do SO deve ter a permissão de desabilitar alguma interrupção.



Exclusão Mútua - Mecanismos básicos

- **Variáveis de impedimento**
  - Exemplo:
    - Se uma variável *lock* estiver marcada como 0 o processo entra na região crítica e a altera para 1
    - Se a variável *lock* estiver marcada como 1 o processo espera
- Essa idéia não funciona (pode ocorrer disputa)
  - Antes de alterar a variável um o processo a lê como 0 e é escalonado pela CPU. Assim outro processo pode ler a variável como 0, alterá-la para 1 e entrar na região crítica
  - Exemplo no próximo slide

Exemplo anterior

- Processo A(){
- ...
- If(Lock == 0){
  - Lock = 1;
  - ... \\ Região crítica
  - Lock = 0;
  - }
- Else Aguardar();
- ...
- }

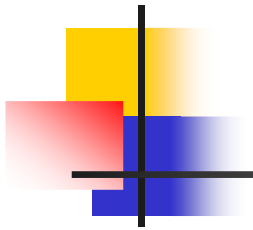
- Processo B(){
- ...
- If(Lock == 0){
  - Lock = 1;
  - ... \\ Região crítica
  - Lock = 0;
  - }
- Else Aguardar();
- ...
- }

Exclusão Mútua - Mecanismos básicos

- **Alternância Obrigatória**
  - Baseada na seqüência lógica antecessor/sucessor
  - Alterna o acesso à região crítica entre dois processos
- Desvantagem
  - Teste continuo do valor da variável compartilhada (usa CPU)
  - Se um processo falha o outro jamais entra na região crítica

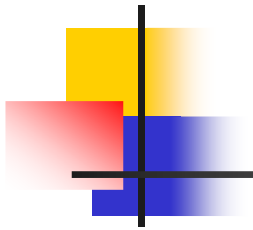
```
Processo A
while (TRUE){
    while (turn==0); // lock()
    região_critica
    turn=1; // unlock()
}
```

```
Processo B
while (TRUE){
    while (turn==1); // lock()
    região_critica
    turn=0; // unlock()
}
```



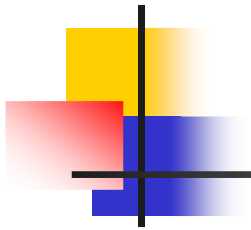
Mecanismos para exclusão mútua

- Solução Algorítmica:
  - Combinação de variáveis do tipo *lock* e alternância (Dekker 1965, Peterson 1981)
- Primitivas:
  - Mutex
  - Semáforos
  - Monitor



Mutex (variável lock, acesso atômico)

- Variável compartilhada para controle de acesso a seção crítica
- CPU são projetadas levando-se em conta a possibilidade do uso de múltiplos processos
- Inclusão de duas instruções *assembly* para leitura e escrita de posições de memória de forma atômica.
  - CAS: *Compare and Store*
    - Copia o valor de uma posição de memória para um registrador interno e escreve nela o valor 1
  - TSL: *Test and Set Lock*
    - Lê o valor de uma posição de memória e coloca nela um valor não zero

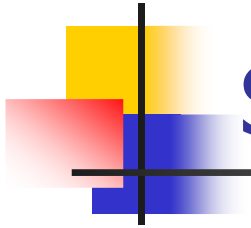


Primitivas *lock* e *unlock*

- O emprego de mutex necessita duas primitivas

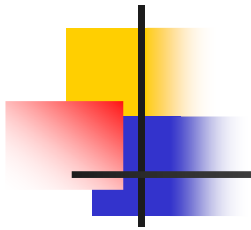
```
enter_region:   tsl register,flag
                cmp register,0
                jnz   enter_region
                ret

leave_region:   mov flag,0
                ret
```

```
                lock(flag);
Seção crítica
                unlock(flag);
```



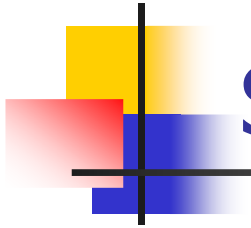
Semáforos

- Utiliza uma variável inteira (o semáforo) para controlar o acesso ao recurso compartilhado
  - O valor 1 indica que nenhum sinal de acordar está pendente
  - Um valor negativo N indica o número de sinais pendentes
  - Semáforo é iniciado com 1
- Primitivas do semáforo
  - Down (P – Proberen - testar) – Chamada quando o processo deseja acessar o recurso compartilhado
  - Up (V – Verhogen - incrementar)– Chamada quando o processo deseja liberar o recurso compartilhado
- As operações Down e Up são executadas de forma atômica sobre o semáforo
  - Uso de mutex



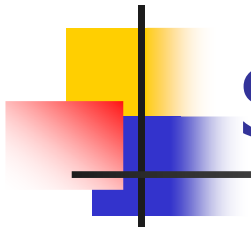
Semáforos

- Primitivas *lock* e *unlock* são necessariamente feitas por um mesmo processo
  - Acesso a seção crítica
- Primitivas Down (*P)* e Up (*V)* podem ser realizadas por processos diferentes
  - Gerência de recursos



Semáforos

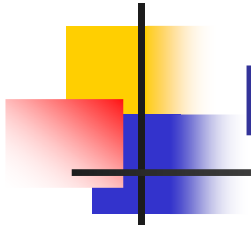
- Problema do produtor-consumidor
  - Necessita 3 semáforos
    - 1 para garantir a exclusão mútua
      - mutex – região crítica: Usados para implementar exclusão mútua.
    - 1 para bloquear os produtores o quando a fila estiver cheia (sem slots disponíveis)
      - full – buffer cheio
    - 1 para bloquear os consumidores o quando a fila estiver vazia (sem itens na fila)
      - empty – buffer vazio



Semáforos

```
define N 100
semaphore mutex=1;
semaphore empty=0;
semaphore full=N;

void producer(void) {
   int item;
   while(1) {
      item=produz_item();
      down(&full);
      down(&mutex);
      insere_item(item);
      up(&mutex);
      up(&empty);
   }
}
```

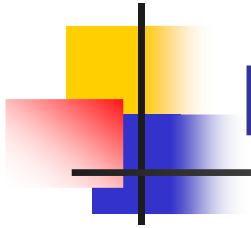
```
void consumer(void) {
   int item;
   while(1) {
      down(&empty);
      down(&mutex);
      item=remove_item();
      up(&mutex);
      up(&full);
      consome_item(item);
   }
}
```



Exercícios programação concorrente

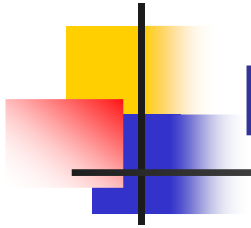
- Para cada exercício a seguir responda:
  - Quantos e quais processos participarão?
  - Quem fará o que?
  - Qual a dependência entre tarefas?
  - Quais são as regiões críticas?
  - Qual a regra de acesso para cada região crítica?
  - Quantos semáforos serão necessários?
  - Qual o objetivo de cada semáforo?

- Mostre um esboço de algoritmo para cada processo.



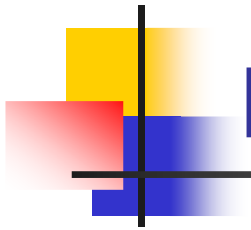
Exercício 1

- Uma ponte, que separa duas cidades A e B, somente permite tráfego em um sentido de A para B ou de B para A. Se a ponte estiver vazia pode ser utilizada por carros de A ou de B. Se um carro de A acessou a ponte, ela é trancada para os carros de B e todos os carros de A que desejarem podem também acessar. Quando o último carro que sai de A para B deixar a ponte, ela deve ser liberada. Implemente o problema da ponte utilizando semáforos.



Exercício 2

- Suponha um ambiente em que processos compartilham impressoras. Existem dois tipos de impressora: tipo A e Tipo B. Existem 3 classes de processos: classe PA que somente utilizam impressoras do tipo A, classe PB que somente utilizam impressoras do tipo B e classe PAB que utilizam impressoras de qualquer tipo. Esses processos, do tipo PAB, têm prioridade sobre os demais processos. Implemente utilizando semáforos.



Exercício 3

- Implemente o problema dos fumantes utilizando monitor. Três fumantes e um agente sentados em uma mesa. Cada fumante possui dois dos três ingredientes para se fazer um cigarro: fósforo, fumo e palha. O agente possui os três e aleatoriamente sorteia um dos ingredientes. O fumante contemplado faz o seu cigarro, fuma e libera o agente para fazer novo sorteio.