INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA
Campus Salvador

LabSO
Gerência de Processos

AULA 4
Flávia Maristela (flavia@flaviamaristela.com)
Romildo Martins (romildo@romildo.net)

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA
Campus Salvador

Retrospectiva da aula passada...

Na aula passada...

- Processos
  - Estados do processo
  - Transições de processos
  - O que motiva a criação de um processo?
  - Como um processo pode ser executado?
  - O que motiva a finalização de um processo?
  - Como visualizar processos no Windows
  - Como visualizar processos no *Linux*
  - Hierarquia de processos

Na aula passada...

- Threads
  - Motivação
  - Conceito
  - *Multithreading*
  - Estados
  - Transições
  - Gerenciamento de *threads*
  - Operações

Na aula passada...

- Gerência de processos
  - Trata do compartilhamento de recursos entre os processos
  - Processos cooperantes precisam trocar informações
  - Como os processos se comunicam?
    - Troca de mensagens (sincronização ou *bufferização*)
    - Compartilhamento de memória
      - Informações são trocadas numa área compartilhada
      - Operação não é gerenciada pelo sistema operacional

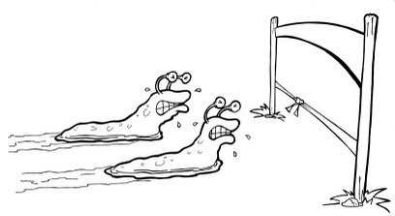
Na aula passada...

- Gerência de processos
  - Problemas associados a comunicação de processos:
    - Condição de corrida (*race condition*)
    - Produtor *vs.* Consumidor

Na aula de hoje...

- *Race condition*
- Região Crítica
- Exclusão Mútua
- Problemas Clássicos
- Escalonamento

Comunicação entre processos (-- *Race condition* --)



- Dois processos podem tentar ler ou escrever dados num espaço compartilhado, e o resultado final depende de quem está executando naquele momento.

Comunicação entre processos (-- *Race condition e* região crítica --)

- O que causa condição de corrida?
  - QUALQUER TIPO DE COMPARTILHAMENTO!!
- O trecho de código em que a memória compartilhada é acessada é chamado de região crítica.

P_1:
```
x := x + 1
y := 5 + 2
z := y + t
```

P_2:
```
x := x * 2
a := 2 * 5
c := a - 7
```

Considerando x = 2

P_1 → P_2 : x = 6
P_2 → P_1 : x = 5

Comunicação entre processos (-- *Race condition e* região crítica --)

- Como evitar condições de corrida?
  - Sincronizando os processos

ou seja

  - Proibindo que mais de um processo possa ler ou escrever numa área compartilhada ao mesmo tempo.

Comunicação entre processos (-- Exclusão Mútua --)

- Definição:
  - Mecanismo que garante que cada processo que usa uma área compartilhada terá acesso exclusivo a mesma.

Qual é o problema da exclusão mútua??

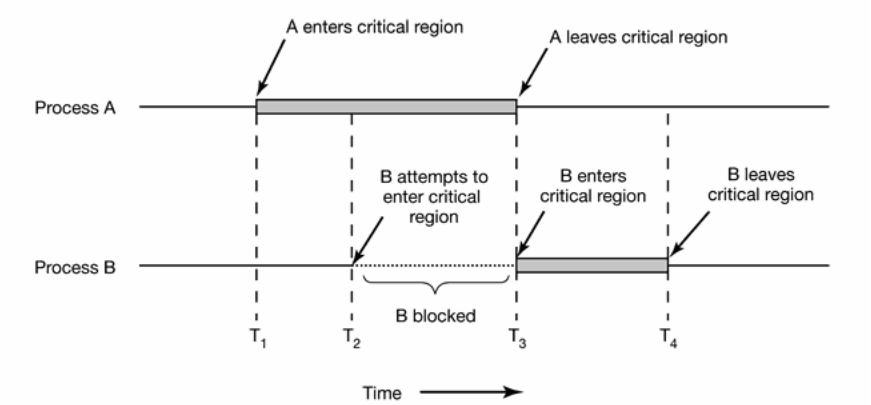
Para pensar...

- Pense no problema do PRODUTOR *vs*. CONSUMIDOR.
- O que acontece se quando o produtor estiver armazenando um item, o consumidor não puder consumir nada?

Comunicação de Processos
(-- Exclusão mútua e região crítica --)

- Dois processos não podem estar simultaneamente em suas regiões críticas
- Nada pode ser assumido com relação a velocidade dos processos ou quantidade de processadores disponível
- Nenhum processo fora de sua região crítica pode bloquear um processo que esteja na região crítica
- Nenhum processo deve esperar indefinidamente para entrar na região crítica.

Comunicação de Processos
(-- Exclusão mútua e região crítica --)



Tanenbaum, Capítulo 2

Comunicação de Processos
(-- Como implementar exclusão mútua --)

- Espera ocupada
- *Sleep* and *wakeup*
- Semáforos
- Mutex
- Monitores

Comunicação de Processos
(-- Exclusão mútua + espera ocupada --)

- Premissa da espera ocupada:
  - Enquanto um processo executa na região crítica, o outro apenas espera.
- Formas de implementar:
  - Interrupção:
    - Problema: não é ideal que processos tenham controle sobre as interrupções

Comunicação de Processos
(-- Exclusão mútua + espera ocupada --)

- Formas de implementar:
  - Alternância Obrigatória

```
while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1);  /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Comunicação de Processos
(-- Sleep e Wakeup --)

- Primitivas (chamadas de sistemas)
- *sleep()*
  - Bloqueia um processo enquanto aguarda um recurso
- *wakeup()*
  - Ativa o processo quando o recurso foi liberado

Comunicação de Processos
(-- *Sleep* e *Wakeup* --)

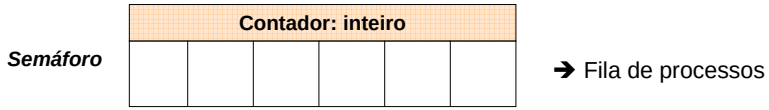
```
#define N 100       /* number of slots in the buffer */
int count = 0;      /* number of items in the buffer */

void producer (void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

Comunicação de Processos
(-- Semáforo --)

- Proposto por E. Dijkstra em 1965
- Apesar de ser um mecanismo antigo, ainda é bastante utilizado em programação concorrente.
- Na prática, é uma variável que deve ser executada de forma ***atômica****
  - A variável possui um contador e uma fila de tarefas;
- Duas primitivas podem ser executadas sobre a variável:
  - *Up()* ➔ *V()*
  - *Down()* ➔ *P()*

Comunicação de Processos (-- Semáforo --)

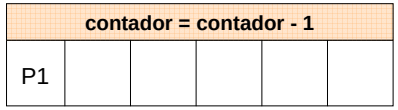
- Tipo de dado abstrato:
  - Contador: inteiro
  - Fila de processos



Comunicação de Processos (-- Semáforo --)

- *Down*()
  - Decrementa o contador
  - solicita acesso à região crítica
    - Livre: processo pode continuar sua execução;
    - Ocupada: processo solicitante é suspenso e adicionado ao final da fila do semáforo;

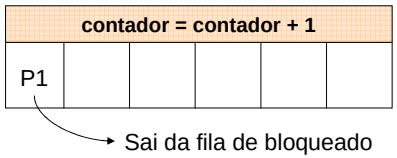
```
Down(s):
    s.counter--
    if (s.counter < 0)
    {
      s.enqueue (processo_atual)
      suspend(processo_atual)
    }
```



Comunicação de Processos (-- Semáforo --)

- *Up* ()
  - Incrementa o contador
  - Liberar a seção crítica
    - Tem processo suspenso: acordar o processo (volta a fila de pronto)
  - Chamada é não bloqueante ➔ o processo não precisa ser suspenso para executá-la.

```
Up(s):
    s.counter++
    if (s.counter ≤ 0)
    {
      s.dequeue (processo_atual)
      acorda (processo_atual)
    }
```



Comunicação de Processos (-- Semáforos --)

Como resolver o problema do Produtor vs. Consumidor usando semáforos?

Comunicação de Processos
(-- Problemas clássicos --)

- Jantar dos filósofos
- Escritores e Leitores
- Barbeiro dorminhoco

Comunicação entre processos
(-- O jantar dos filósofos --)

Comunicação entre processos
(-- O jantar dos filósofos --)

- Formulado por E. Dijkstra para caracterizar o problema da sincronização e concorrência

- Descrição
  - 5 filósofos numa mesa de jantar circular
  - 5 pratos de espaguete
  - 1 garfo entre cada par de pratos

Comunicação entre processos
(-- O jantar dos filósofos --)

- Descrição
  - Cada filósofo pode “comer” ou “pensar”
  - Cada filósofo usa dois garfos para comer
  - Cada filósofo pega um garfo por vez

Jantar dos filósofos (-- 1ª solução --)

```
#define N 5

void philosopher (int i)
{
   while (TRUE)
   {
      think();
      take_fork (i);
      take_fork ((i+1) % N);
      eat();
      put_fork (i);
      put_fork ((i+1) % N);
   }
}
```

- O que acontece se todos os filósofos pegam o garfo da esquerda simultaneamente?
  - Nenhum filósofo consegue pegar o garfo da direita
  - ***DEADLOCK***

Jantar dos filósofos (-- 2ª solução --)

```
#define N 5

void philosopher (int i)
{
   while (TRUE)
   {
      think();
      take_fork (i);
      if (fork((i+1) % N) is available)
      {
         take_fork ((i+1) % N);
         eat();
         put_fork (i);
         put_fork ((i+1) % N);
      }
      else
        put_fork (i);
   }
}
```

- O que acontece se todos os filósofos pegam o garfo da esquerda simultaneamente?
  - ***INANIÇÃO (starvation)***

Jantar dos filósofos (-- 3ª solução --)

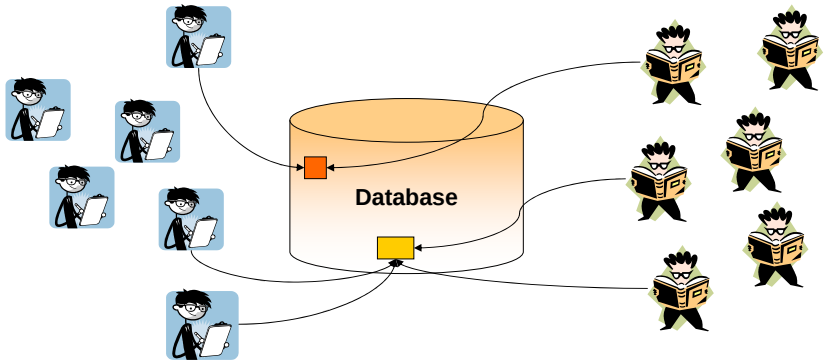
```
#define N 5

void philosopher (int i)
{
   while (TRUE)
   {
      think();
      down(mutex);
      take_fork (i);
      take_fork ((i+1) % N);
      eat();
      put_fork (i);
      put_fork ((i+1) % N);
      up(mutex);
   }
}
```

- O que acontece nesta solução?
  - Apenas um filósofo come por vez
  - Afeta o ***PARALELISMO***

Jantar dos filósofos (-- 3ª solução --)

- Atribui 3 possíveis estados aos filósofos
  - PENSANDO
  - COMENDO
  - FAMINTO
- Idéia:
  - Um filósofo no estado “faminto” só pode pegar os garfos se os seus vizinhos (esquerda e direita) não estiverem “comendo”.
- Estudar a solução para o problema dos filósofos!

Comunicação entre processos
(-- Os leitores e escritores --)



Comunicação entre processos
(-- Barbeiro dorminhoco --)

DEADLOCK

Deadlock

- Problema de programação concorrente
- "Um conjunto de n processos está em *deadlock* quando cada um dos n processos está bloqueado a espera de um evento que somente pode ser causado por cada um dos n processos."

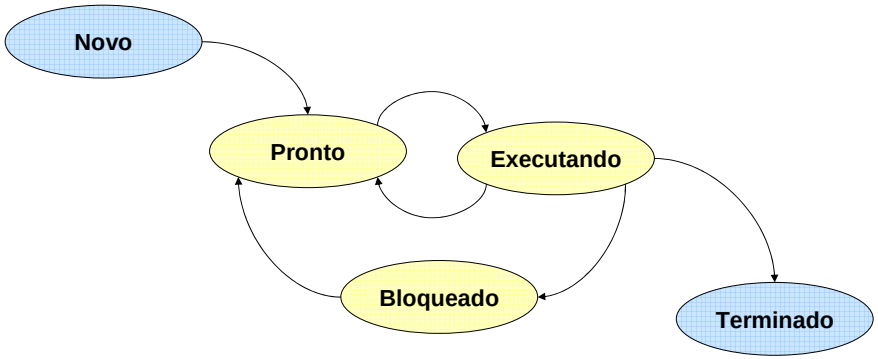
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA
Campus Salvador

Escalonamento de Processos

Porque é necessário escalonar?

- Processos precisam ser executados
- Processos concorrem a CPU
- Escalonador:
  - Componente (implementação) do sistema operacional
  - Determina a ordem de execução dos processos baseado num *algoritmo de escalonamento*
  - Lê a fila que contém os processos no estado “pronto” e os ordena para execução

O que provoca o escalonamento?



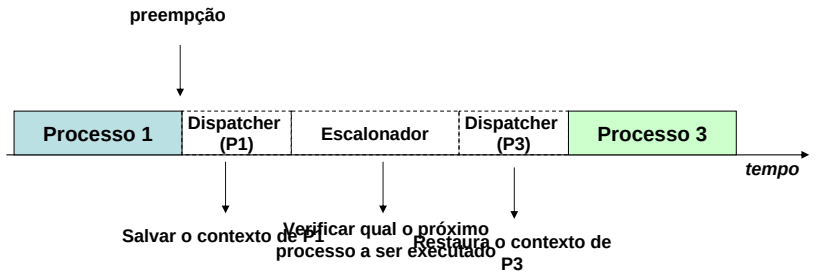
Tipos de algoritmo de escalonamento

- Preemptivo:
  - Execução de um processo dura um tempo pré-determinado
  - Quando o tempo acaba, o processo é interrompido.
- Não-preemptivo:
  - Processo fica em execução até que:
    - Termine
    - Libere a CPU VOLUNTARIAMENTE
    - Seja bloqueado por falta de recurso

O que afeta a performance de um algoritmo de escalonamento?

- Cada processo possui informações que permitem definir precisamente seu estado.
  - Tais informações definem o ***contexto*** do processo
- Troca de Contexto
  - Mecanismo que permite ao escalonador interromper uma tarefa, e executá-la posteriormente, sem corromper seu estado.
  - Separação do escalonamento
    - Escalonamento = Política + Mecanismo

Ilustração da troca de contexto



Qual o objetivo do escalonamento?

- DEPENDE do ***tipo*** de sistema operacional
  - Lote:
    - Não possui usuários aguardando ➔ pode ser preemptivo ou não
    - Não possui muita troca de contexto
    - OBJETIVOS:
      - melhorar o *throughput* (vazão)
      - melhorar o *turnaround* (tempo entre submissão e finalização)
      - manter a CPU ocupada

Qual o objetivo do escalonamento?

- Propósito Geral:
  - Possuem usuários interagindo
  - Precisam ser preemptivos
  - OBJETIVOS
    - melhorar o tempo médio de resposta
    - atender as expectativas dos usuários
- Tempo real:
  - Em geral são preemptivos
  - OBJETIVO:
    - cumprir requisitos lógicos
    - cumprir requisitos temporais

Qual o objetivo do escalonamento?

- Independente do *tipo* de sistema operacional, TODOS os algoritmos de escalonamento precisam atender a alguns critérios:
  - Justiça (fairness)
  - Aplicação da política de escalonamento
  - Equilíbrio (balance) entre as partes do sistema

Escalonamento para sistemas em lote

- FCFS (ou FIFO)
  - Primeiro processo da fila de pronto é o escolhido para executar.
  - Não-preemptivo
  - Fácil de entender
  - Fácil de programar
  - "Justo"
  - Processos de baixo custo de execução podem esperar muito tempo para ser executado

Escalonamento para sistemas em lote

- FCFS (ou FIFO)
  - Fazer o escalonamento para os seguintes processos:

Processo	Custo de execução
A	12
B	8
C	15
D	5

Escalonamento para sistemas em lote

- Menor Job Primeiro
  - O *job* de menor custo de execução executa primeiro.
  - Não-preemptivo
  - Fácil de entender
  - Fácil de programar
  - "Justo"
  - Para ser adequado requer que todos os jobs estejam disponíveis simultaneamente

Escalonamento para sistemas em lote

- Menor Job Primeiro
  - Fazer o escalonamento para os seguintes processos

Processo	Custo de execução
A	12
B	8
C	15
D	5

Escalonamento em sistemas de propósito geral

- Alternância circular (*round-robin*)
  - Processos executam dentro de uma fatia de tempo predefinida (***quantum***)
  - Preemptivo
  - Simples
  - Justo
  - Amplamente utilizado
  - Tamanho do *quantum* pode ser um problema

Escalonamento em sistemas de propósito geral

- *Round-Robin*
  - Fazer o escalonamento para os seguintes processos considerando um *quantum* = 3

Processo	Custo de execução	Prioridade
A	12	3
B	8	4
C	15	2
D	5	1

Escalonamento em sistemas de propósito geral

- Prioridade
  - Processos tem diferentes prioridade de execução
  - Preemptivo
  - Baseado nos ciclos da CPU ou *quantum*
  - Prioridade pode ser atribuída estaticamente ou dinamicamente
  - Pode ser implementado considerando filas de prioridades
  - Pode ocasionar ***starvation***

Escalonamento em sistemas de propósito geral

- Prioridade
  - Fazer o escalonamento para os seguintes processos

Processo	Custo de execução	Prioridade
A	12	3
B	8	4
C	15	2
D	5	1

Para a próxima aula

- Trazer todos os exercícios dos slides respondidos.
- Verificar as implementações de semáforo para o problema do produtor consumidor.
- Escalonamento com múltiplas filas.
- Descrever a diferença entre processos I/O- Bound e CPU-Bound.