Instituto de Informática - UFRGS

Sistemas Operacionais

Sincronização de processos

Aula 07

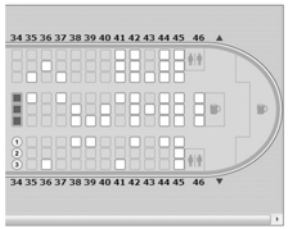
Introdução

- Em sistemas multiprogramados há interação entre processos para
  - comunicação, que pode ser feita via compartilhamento de memória/arquivos
  - evitar inconsistências de acesso compartilhado
  - executar de ações de forma ordenada e determinística
- Concorrência
  - Processos (*threads*) podem afetar a execução ou serem afetadas por outro(a)
  - Programação concorrente assíncrona
    - Atividades podem ser interrompidas sem aviso prévio e de forma não determinística
- Dois problemas abordados:
  - Compartilhamento de dados (sincronização para acesso a dados)
  - Coordenação de atividades (sincronização de controle)

Site de companhia área

- Sistema *web* para reserva de assentos em voo
  - Vários clientes remotos, via browser, acessam de forma concorrente o trecho de código abaixo
  - Uma *thread* para cada requisição de usuário (servidor web *multithreaded*)

```
int reservaAssento() {
   int n;
   consultaAssentoLivre(&n);
   alocaAssentoLivre(n);
   return(n);
}
```



Considerar duas situações:
1. execução paralela (hardware *multicore*)
2. execução concorrente

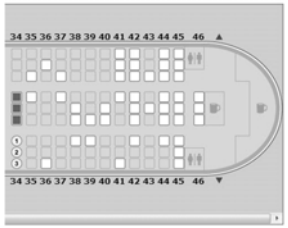
FUNCIONA??

Condição de corrida (*race condition*)

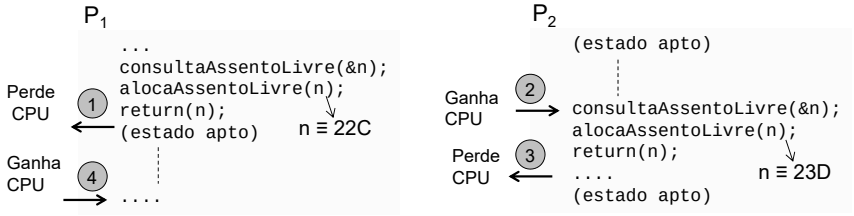
- Situação onde o resultado final depende da ordem em que processos (*threads*) são executados
  - Ocorre quando há acesso compartilhado em leitura-escrita ou escrita
- A ordem de execução é afetada pelo escalonamento

Exemplo:
reserva de assentos em avião

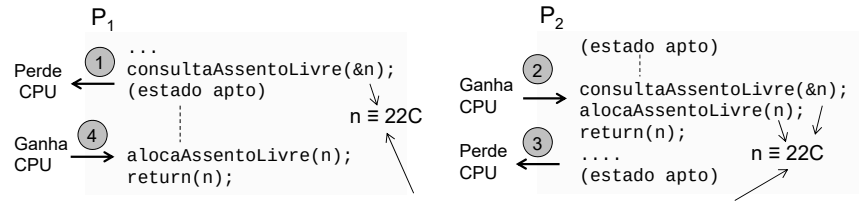
```
int reservaAssento() {
   int n;
   consultaAssentoLivre(&n);
   alocaAssentoLivre(n);
   return(n);
}
```



Exemplo: reserva assentos em avião



Mas é se acontecesse isso...

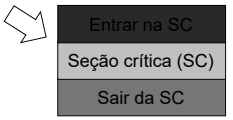


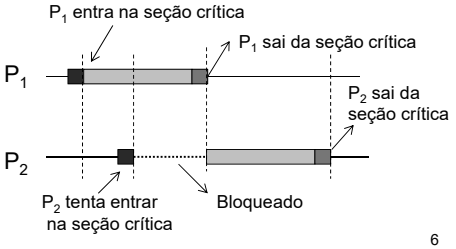
Resulta em dois passageiros no mesmo assento.

Seção crítica e exclusão mútua

- Seção crítica
  - Trecho de código que não pode ser executado por dois ou mais processos simultaneamente (real ou aparente) sob pena de inconsistência de dados
  - Gera condição de corrida em acessos em escrita a recursos compartilhados
- Exclusão mútua
  - Consiste em garantir que se um processo está usando um recurso compartilhado os outros processos estão impedidos de fazê-lo

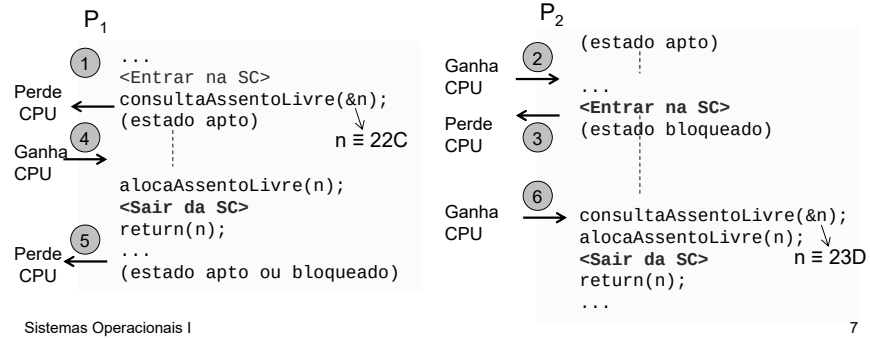
Como garantir?
Criando mecanismos para exclusão mútua.





O problema do assento de avião revisitado...

```
int reservaAssento() {
   int n;
   <Entrar na SC>
   consultaAssentoLivre(&n);
   alocaAssentoLivre(n);
   <Sair da SC>
   return(n);
}
```



Propriedades para mecanismos de exclusão mútua

- Exclusão mútua
  - Dois ou mais processos não podem estar simultaneamente na seção crítica
- Progressão
  - Um processo fora da seção crítica não pode impedir outro de entrar nela
- Espera limitada
  - Um processo não pode ficar esperando eternamente para entrar na seção crítica
- Suposições
  - Nenhuma suposição pode ser feita sobre a velocidade relativa de execução dos processos, nem de número de processadores

Processo ou *thread*

Como implementar primitivas de exclusão mútua?

- Com suporte de hardware
  - Baseado em habilitação/desabilitação de interrupções
  - Instruções atômicas (*test-and-set* e *swap*)
- Sem suporte de hardware, i.é, em software puro
  - Variáveis do tipo trava (*lock*)

Contexto de INF01142

  - Algoritmo de Dekker
  - Algoritmo de Peterson
  - Algoritmo de Lamport
- Procedural
  - Baseado em suporte por compiladores (monitores)

Contexto de INF01151

Desabilitação de interrupções

- Mecanismo em hardware (instruções do processador)
- Só há troca de contexto com a ocorrência de interrupções de hardware
  - Conclusão de E/S, violação de proteção, temporização

CLI;	Desliga interrupções
Seção crítica	
STI;	Ativa interrupções

- Problemas:
  - Poder demais para um usuário !!
  - Risco de perda de eventos
  - Não funciona em multiprocessadores (só CPU que realiza CLI é afetada)
  - Não funciona em *multicore* (o outro *core* ainda pode executar o mesmo código)

Variáveis do tipo trava (*lock*)

Também chamada de MUTEX
(*MUTual Exclusion*)

- Mecanismo em software
- Uso de uma variável compartilhada que armazena dois estados
  - Livre e ocupado

while (flag == 1); flag = 1;
Seção crítica
flag = 0

Solução:
Fazer com que o ***while*** e a **atribuição** sejam feitas de forma indivisível com auxílio do hardware.

- Problema:
  - Viola a propriedade de "exclusão mútua"
    - Condição de corrida na primitiva de entrada

Instruções especiais

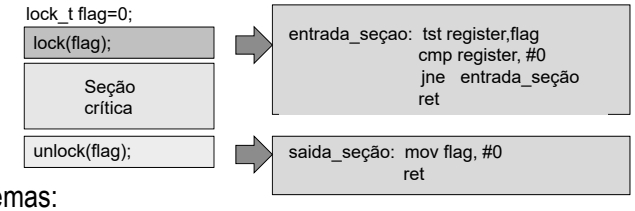
- Mecanismo em hardware
  - Processadores são projetados para uso em ambientes de programação concorrente
- Instruções *assembly* para leitura e escrita em posições de memória de forma atômica (indivisível).
  - SWAP: swap (a, b)
    - Executa a permutação de valores
  - *Compare and Store:* cas r1, mem
    - Copia a posição de memória para um reg. interno e escreve o valor 1 nela
  - *Test and Set Lock:* tst registrador, mem
    - Lê o valor de uma posição de memória e coloca nela um valor não zero

Intel x86 possuem a instrução *assembly* XCHG

Três versões: xchg *reg, reg*
xchg *reg, mem*
xchg *mem, reg*

Uso de TSL para implementar variáveis do tipo trava (*lock)*

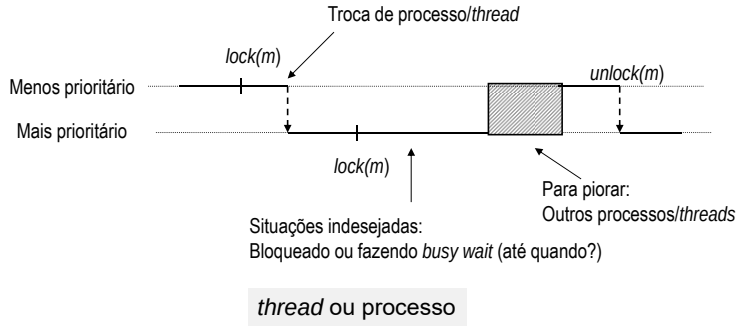
- Solução baseada em instruções especiais
  - Variável para indicar uso da seção crítica (livre:0; ocupado ≠0)
  - Duas primitivas: *lock*(flag) e *unlock* (flag)



- Problemas:
  - Inversão de prioridade
  - Desperdício de tempo de processamento (*spinlock* ou trava giratória)
  - Travamento, se não houver interrupção de tempo (*livelock*)

Inversão de prioridades

- Uma *thread* de mais baixa prioridade impede a progressão de uma *thread* de mais alta prioridade



Soluções para inversão de prioridade

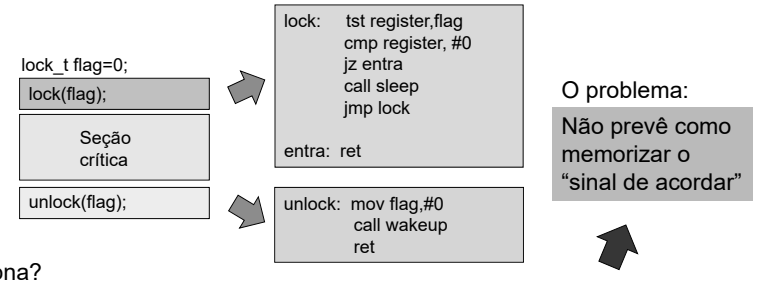
- Teto para prioridade (*priority ceiling*)
  - Aumento da prioridade de uma *thread* para um valor pré-determinado sempre que ela adquirir uma trava (*lock*)
  - Teto deve ser maior que a prioridade da maior *thread*
  - Na liberação (*unlock*) a *thread* retorna a sua prioridade original
- Herança (*priority inheritance*)
  - A prioridade da *thread* que detém a trava é elevada ao nível da *thread* mais prioritária que solicita o recurso
  - Elevação da prioridade só ocorre se houver risco de conflito
  - Na liberação (*unlock*) a *thread* retorna a sua prioridade original

O problema da espera ativa

- Espera ativa ou *busy wait*
  - Ficar testar testando continuamente se a variável de entrada (*flag*) foi liberada
  - Desperdício de tempo de processamento
- Solução:
  - Bloquear o processo (*thread)* até que alguém libere a seção crítica
- Oferecido por primitivas mais elaboradas que fazem
  - *sleep*: bloqueia um processo a espera de um recurso*
    - Executado no *lock* quando o recurso não estão disponível
  - *wakeup*: desbloqueia o processo que esperava por um recurso
    - Executado no *unlock*, na liberação do recurso, quando há alguém a espera

* Recurso (caso particular) = seção crítica

Uma nova versão para *lock* e *unlock*



O problema:

Não prevê como memorizar o "sinal de acordar"

Funciona?

Condição de corrida

Situação:
- P_1 executa o *lock*, entra na seção crítica e perde o processador antes de liberar.
- P_2 executa o *lock* e perde o processador depois do JZ e antes do CALL
  -Estava "indo dormir", mas ainda não dormiu (não está na fila dos "dorminhocos")
- P_1 retoma a execução, faz o *unlock*, mas não libera o P_2 porque ele ainda não está na fila.
- P_2 retoma execução, vai dormir (CALL), e perde o sinal de acordar

Semáforos

- Proposto por Dijkstra (1965)
  - Mecanismo para contar o número de sinais de acordar para uso futuro
- Semáforo é um tipo abstrato de dados:
  - Um valor inteiro: contabiliza o número de sinais de acordar
  - Fila: processos que esperam sinal de acordar associado a este semáforo
- Duas primitivas:
  - P (*Proberen*, testar) → similar ao *sleep*
  - V (*Verhogen,* incrementar)→ similar ao *wakeup*
- Disponibilizadas para usuários (programas aplicativos)
  - Introduzidos a primeira vez em Algol 68 com os nomes *down* e *up*.

Primitivas P(s) e V(s)

- Primitiva P: decrementa o valor inteiro e testa se menor que zero
  - Sim: processo é posto para dormir (*sleep*) sem continuar a operação P
  - Não: segue adiante
- Primitiva V: incrementa o valor inteiro e verifica se tem processos esperando no semáforo
  - Sim: um é escolhido para continuar a operação P que iniciou previamente
  - Não: segue adiante

```
P(s):  s.valor = s.valor – 1;
       se s.valor < 0 {
            Insere processo P em s.fila
            Bloqueia processo P (sleep);
       }
```

```
V(s): s.valor = s.valor + 1
      se s.valor <=0 {
          Retira um processo P´ de s.fila;
          Acorda processo P´ (wakeup);
      }
```

IMPORTANTE: atentar para as seções críticas na implementação de P(s) e V(s). As ações são feitas de forma indivisível para evitar condições de corrida

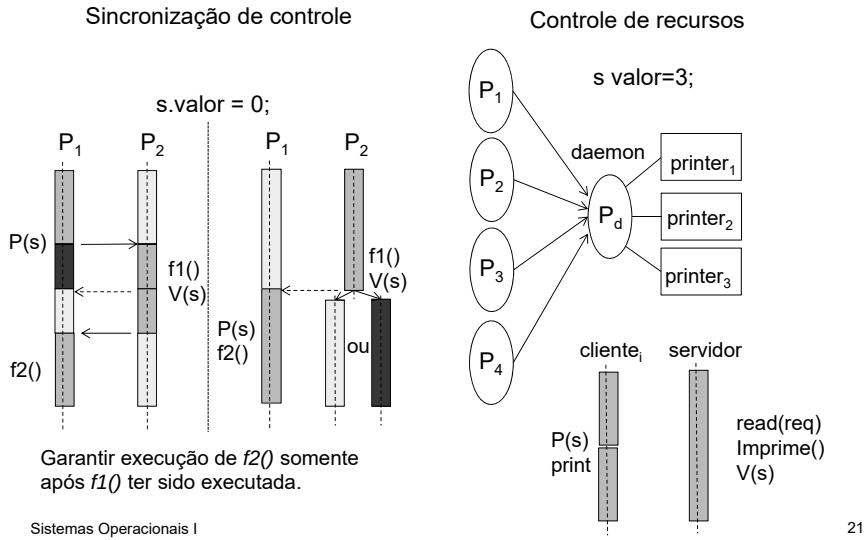
Tipos e empregos de semáforos

- Dependendo dos valores assumidos por *s.valor*
  - Semáforos binários: *s.valor = 1*
  - Semáforos contadores: *s.valor = n*
- Emprego:
  - Exclusão mútua: semáforos binários
  - Controle de recursos: semáforos contadores
  - Sincronização entre processos
    - Garantir que um processo pode prosseguir após uma condição se tornar verdadeira

É isto que diferencia de travas (*locks*)

Muito mais sobre semáforos (e outras primitivas) em INF01151 – Sistemas Operacionais II

Exemplo de emprego de semáforos



Na prática o que se tem é...

- Seções críticas são pequenas
  - Apenas a ação de ler-modificar a variável compartilhada
- Sistemas com um processador
  - Emprega a técnica de habilitar e desabilitar interrupções para ações do núcleo
    - Supõe que o programador do núcleo tem todo cuidado e é responsável
- Sistemas com mais de um processador/*core*
  - Emprega a técnica de *mutex* com suporte a hardware (instruções indivisíveis e travamento de acesso concorrente a posições de memória)
  - Deixa ocorrer a espera ativa
    - Aposta é que no outro processador/core será escalonado o processo ou a thread que vai liberar o acesso a seção crítica
- Usuários empregam semáforos para sincronização (acesso a dados e/ou controle)
  - Semáforos são implementados pelo núcleo e disponibilizados aos usuários através de chamadas de função (encapsulam chamadas de sistema)

Leituras complementares

- A. Tanenbaum. *Sistemas Operacionais Modernos* (3ª edição), Pearson Brasil, 2010.
  - Capítulo 2: seções 2.3.1 a 2.3.7
- A. Silberchatz, P. Galvin; *Sistemas Operacionais*. (7ª edição). Campus, 2008.
  - Capítulo 6 (seções 6.1, 6.2, 6.4, 6.5 e 6.7)
- R. Oliveira, A. Carissimi, S. Toscani; *Sistemas Operacionais*. Editora Bookman 4ª edição, 2010
  - Capítulo 3 (seções 3.1 a 3.7 e 3.10)