Instituto de Informática - UFRGS

Sistemas Operacionais

Introdução à
Programação Concorrente

Aula 09

Números primos... (1ª tentativa)

- Problema:
  - Fazer um programa para determinar os números primos existentes entre 1 e 10^{10}. Supor um processador com 8 cores e solução baseada em threads (cada *thread* executa a função *primos*)

```
int counter = 0; //Global compartilhada entre threads

int incCounter(void) {
   return counter++;
}

void primos (void) {
int i=0;
int limit = power(10,10);
   while (i < limit) {
      i = incCounter();
      if (isPrime(i))
          print(i);
   }
}
```

Problemas?

Instituto de Informática - UFRGS
A. Carissimi -4-abr-12

Valores inconsistentes para *counter*...

Thread A	supondo *counter=10*	Thread B
`MOV R,COUNTER (R=10)`		
		`MOV R,COUNTER (R=10)`
		`INC R (R=11)`
		`MOV COUNTER,R (R=11)`
		`(cálculo do primo)`
		`MOV R,COUNTER (R=11)`
		`INC R (R=12)`
		`MOV COUNTER,R (R=12)`
		`(cálculo do primo)`
		`MOV R,COUNTER (R=12)`
		`INC R (R=13)`
		`MOV COUNTER,R (R=13)`
`INC R (R=11)`		
`MOV COUNTER,R (R=11)`		

Incrementar *counter* exige duas operações em um objeto compartilhado: ler e escrever!

Instituto de Informática - UFRGS
A. Carissimi -4-abr-12

Introdução

- Existe concorrência sempre que houver mais de uma unidade de execução (*processo* ou *thread*) no sistema
  - Concorrem (disputam) pelo recurso CPU (pelo menos)
- Execução concorrente pode ser de forma:
  - Independente
  - Colaborativa: duas ou mais unidades de execução interagem para realizar uma tarefa → necessitam comunicar ou sincronizar com seus pares
    - Uma unidade de execução é capaz de afetar ou ser afetada por outra
- Programação concorrente assíncrona
  - Atividades podem ser interrompidas sem aviso prévio e de forma não determinística

Instituto de Informática - UFRGS
A. Carissimi -4-abr-12

Condição de corrida (*race condition*)

- Situação onde o resultado da computação feita por processos* cooperantes é dependente da ordem em que esses são executados
  - Ordem é afetada, entre outros fatores, pelo escalonamento
  - Programas concorrentes devem ser imunes a ordem de execução
- Ocorre quando processos compartilham recursos em escrita ou escrita/leitura. Exemplos:
  - Relação produtor-consumidor (acesso a buffer)
  - A variável *counter* (programa dos primos)
  - Etc
- Necessário coordenar a execução dos processos

*válido para threads também!

Sincronização de processos

- Exclusão mútua
  - Composição de ações de forma a evitar entrelaçamentos indesejáveis que levem a resultados inconsistentes
  - Conceito de seção crítica
    - Conjunto de ações de um processo* que não podem ser intercaladas com as ações (sobre o mesmo conjunto) de outro processo
- Condicional
  - "atrasar" a execução de um processo* até que uma condição seja satisfeita

*ou threads!

O problema da seção crítica....

- Seção crítica
  - Porção de código que não pode ser executado por dois ou mais processos simultaneamente sob pena de inconsistência de dados
    - Acesso (em modificação) a recursos compartilhados
- Necessário garantir acesso exclusivo de um processo à seção crítica
  - Se um processo está executando instruções da seção crítica, outro processo é impedido de entrar até que a primeira saia
    - É o que se denomina de exclusão mútua
  - Resulta em uma serialização no acesso
  - Necessário prever primitivas para exclusão mútua
    - *enter_EM*
    - *exit_EM*

Números primos... (2ª tentativa)

- Versão 2: Proteger *counter* dos acessos concorrentes

```
int counter = 0; //Global compartilhada entre threads

int incCounter(void) {
   <entrada na exclusão mútua>
   counter++;
   <saída da exclusão mútua>
   return counter
}

void primos (void) {
int i;
int limit = power(10,10);
   while (i < limit) {
      i = incCounter();
      if (isPrime(i))
          print(i);
   }
}
```

Propriedades para primitivas de exclusão mútua

(1) Exclusão mútua
- Garantir que apenas um processo esteja executando dentro da seção crítica.

(2) Progressão
- Um processo fora da seção crítica não pode impedir outro processo de entrar

(3) Espera limitada
- Um processo não deve ser indefinidamente impedido de entrar na seção crítica

(4) Não deve fazer suposições sobre a velocidade relativa da execução dos processos, nem de número de processadores

Processo ou *thread*

Como implementar primitivas de exclusão mútua?

- Sem suporte de hardware, i.é, em software puro
  - Algoritmo de Dekker
  - Algoritmo de Peterson
  - Algoritmo de Lamport
- Com suporte de hardware
  - Baseado em habilitação/desabilitação de interrupções
  - Instruções atômicas (*test-and-set* e *swap*)
- Procedural
  - Baseado em suporte por compiladores

Problemas com soluções em software

- Software puro: algoritmos de Dekker, Peterson e Lamport
  - Vantagem: independente de hardware e sistema operacional
  - Problemas:
    - Complexidade
    - Problema de desempenho
    - Espera ativa (busy wait)
    - Falta de clareza o que é lógica de controle de acesso a seção crítica e o que é da lógica do programa

Solução

Contar com o auxílio de mecanismos de hardware (projeto do processador)

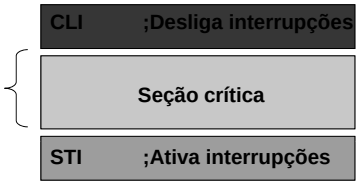
Mecanismos de hardware

- Habilitação e desabilitação de Interrupções
- Instruções específicas a serem usadas como base para se implementar primitivas de exclusão mútua:
  - Test and set
  - Compare and Store
  - Swap

Unidades de execução: *thread* ou processos (discurso válido para ambos!)

Desabilitação de interrupções

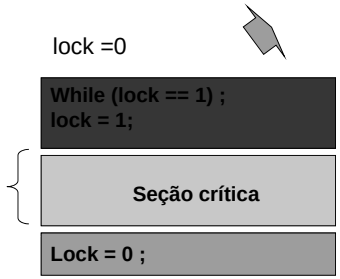
- Só há chaveamento de unidades de execução com a ocorrência de interrupções de tempo ou de eventos externos (E/S, traps, etc)
- Problemas:
  - Poder demais para um usuário !!
  - Não funciona em máquinas multiprocessadoras (SMP) pois apenas a CPU que realiza a instrução é afetada
    - Violação da regra "não fazer suposições sobre processadores..."



Variável do tipo *lock*

- Criação de uma variável especial, compartilhada, que armazena dois estados:
  - Zero: livre
  - 1: ocupado
- Desvantagem:
  - *Race conditions*

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



Instruções especiais

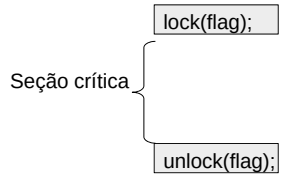
- Processadores são projetados considerando uso em ambientes de programação concorrente
- Instruções *assembly* para leitura e escrita em posições de memória de forma atômica (indivisível).
  - SWAP: swap (a, b)
    - Executa a permutação de valores
  - *Compare and Store:* cas r1, mem
    - Copia o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

Uso de TLS para implementar variáveis *lock*

- Resolve o problema de condição de corrida por hardware
  - Duas primitivas (*lock e unlock*) e uma variável para indicar uso da seção crítica
    - Zero significa que a região está livre e diferente de zero ocupada
- Construção denominada de *mutex*

```
entrada_seçao (lock):   tst register,flag
                        cmp register,0
                        jnz  entrada_seção
                        ret

saida_seção (unlock):   mov flag,0
                        ret
```



- Desvantagens:
  - Inversão de prioridades
  - Espera ativa (*busy waiting*)

Problema e solução para a espera ativa

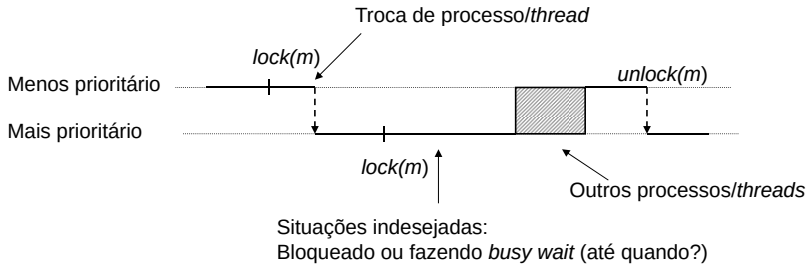
- Espera ativa ou *busy wait* (*spin lock*)
  - Sem sentido, pois para alguém liberar o recurso deve executar, mas para executar tem que usar a CPU
    - Argumento válido para monoprocessadores (*monocore*)
- Solução é bloquear o processo (*thread*)
  - Baseado em duas novas primitivas mais elaboradas
    - *sleep*: Bloqueia um processo a espera de uma sinalização
    - *wakeup*: Sinaliza um processo

Mas, espera ativa nem sempre é um problema...

- Em multiprocessadores (multicore) o *busy wait* pode ser interessante
  - Há outro "processador" para liberar o recurso
  - O custo de chaveamento ***pode*** não compensar (relação com o tamanho da seção crítica)
- Porém viola a regra sobre "fazer suposições sobre processadores"
- Possibilidade: primitiva *trylock*
  - Faz *busy wait* por um período de tempo, se durante esse período o *mutex* não tiver sido liberado bloqueia (*sleep*)
  - O programador escolhe entre *trylock* e *lock*
  - Problema é dimensionar o período de tempo
    - Compromisso entre o tamanho da seção crítica, tempo de chaveamento de contexto e tempo estimado de espera

Problema: Inversão de prioridades

- Uma *thread* de mais baixa prioridade impedir 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 um mutex (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 o mutex 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

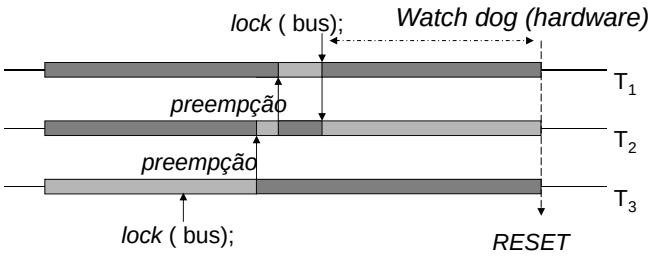
O caso Missão *Mars pathfinder:* robô *sojourner*

O problema:

A sonda, depois de algum tempo da missão iniciou, de forma não determinística, a fazer *resets* e se auto-inicializar. A cada *reset*, os dados coletados e não transmitidos eram perdidos.

O problema do robô *sojourner*

- Thread 1: controle do barramento (alta prioridade)
  - Transferência de dados de controle em um barramento de informações
- Thread 2: transmissão de dados (média prioridade)
- Thread 3: coleta de dados meteorológicos (baixa prioridade)
  - Coleta dados meteorológicos e os publica no barramento de informações



Semáforos

- Proposto por Dijkstra (1965)
- Usar uma variável inteira para contabilizar o número de *wakeups* para uso futuro
- Semáforo é um tipo abstrato de dados:
  - Um valor inteiro
  - Fila de processo
- Duas primitivas:
  - P (*Proberen*, testar) → similar ao *sleep*
  - V (*Verhogen,* incrementar)→ similar ao *wakeup*

Primitiva 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 nesse 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 {
            Bloqueia processo (sleep);
            Insere processo em s.fila;
       }
```

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

IMPORTANTE: atentar para as seções críticas na implementação de P e V

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

Monitores

- Problema com primitivas do tipo semáforo e *mutexes* é *deadlock*
  - Má utilização das primitivas levando a um intertravamento do processos
- Monitor surge como solução
  - Conjunto de variáveis, funções e estruturas de dados agrupadas em um único módulo funcional (filosofia de OO → não se acessa variáveis diretamente)
  - Garante que apenas um processo está dentro do monitor em um determinado instante de tempo
  - Solução integrada a um compilador

```
monitor  example
         integer i;
         condition c;

procedure producer(x);
....
end

procedure consumer(x)
....
end
end monitor;
```


Variável de condição

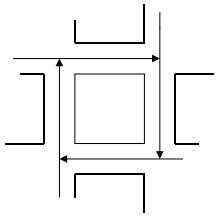
- Forma de bloquear um processo dentro do monitor
- Duas operações:
  - *wait*: bloqueia um processo dentro do monitor e permite a entrada de outro
  - signal: forma de acordar o processo que estava bloqueado no *wait*
    - Importante: apenas um deles pode prosseguir no monitor!!
- Variáveis de condição não são contadores, ou seja, não acumulam a informação de quantos sinais foram enviados (perda!!)
  - Similar as operações wakeup e sleep exceto pelo fato que o monitor garante por construção que não ocorrerá a condição de corrida

Comparação entre as primitivas

- Mutexes (*lock* e *unlock*) são necessariamente feitos por um mesmo processo
  - Acesso a seção crítica
- Primitivas *P* e *V* podem ser realizadas por processos diferentes
  - Gerência de recursos
  - Sincronização
  - Controle de acesso a seção crítica (semáforo binário)
- Monitores
  - Construção em nível de linguagem e dependente de compilação

Deadlock (uma palavrinha final...)

- Situação na qual um, ou mais processos, fica impedido de prosseguir sua execução devido ao fato de cada um estar aguardando acesso a recursos já alocados por outro processo



DEADLOCK é assunto de Sistemas Operacionais II !!!

Condições para ocorrência de *deadlocks*

- Quatro condições necessárias para que *deadlock* ocorra
  - Exclusão mútua:
    - Um recurso só pode estar alocado a um processo em um dado instante
  - Segura/espera:
    - Processos não liberam recursos previamente alocados enquanto esperam pela alocação de um novo recurso
  - Recurso não-preemptível:
    - Um recurso não pode ser "arrancado a força" de um processo
  - Espera circular:
    - Existência de um ciclo de 2 ou mais processos, onde cada processo no ciclo possui um recurso solicitado pelo próximo processo no ciclo.

Estratégias para tratamento de *deadlocks*

- Ignorar
- Deteção e recuperação
  - Monitoração dos recursos liberados e alocados
  - Eliminação de processos
- Impedir ocorrência cuidando na alocação de recursos
  - Algoritmo do banqueiro
- Prevenção (por construção)
  - Evitar a ocorrência de pelo menos uma das quatro condições necessárias

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)