Revisão

Introdução

Modelos de programação

Máquinas/arquiteturas paralelas

- Pipelining, super-escalares e VLIW
- SIMD e máquinas vetoriais
- Multicomputadores e multiprocessadores
- Exemplos

Hoje

Sincronização

- Spin locks

- Barreiras

- Sincronização baseada em escalonador

- Problemas tradicionais em sincronização

Sincronização

Relembrando: sincro = regras para execução // $\rightarrow$ eliminar conds de corrida indesejadas

Maior parte: exclusão mútua e sincro por condição.

Propriedades:

- Safety. Ex: exclusão mútua garantida

- Liveness. Exs: deadlock, livelock e starvation não ocorrem

Safety sozinha = correção parcial

Safety + terminação = correção total

Exemplo: Bounded Buffer

```
buffer: array [1..N] of data

procedure insert (x : data)
   insert, waiting if full

procedure remove : data
   remove, waiting if empty
```

Excl mútua: buffer accessado sequencial/

Sincro por condição: buffer cheio/vazio

Locks e Barreiras

- Locks são usados para exclusão mútua
- Locks podem ser usados para sincro de alto níivel (ex. monitores)
- Barreiras usadas para separar fases de algs (ex. SOR)

Spinning ou Blocking?

Deve-se usar spinning quando:

- Tempo de bloqueio maior que tempo esperado de spinning

- Recursos não seriam usados para outras tarefas

- Blocking impossíível ou inapropriado

Spin Locks

Fundamental que todo o spinning seja feito em memória local!

- Reads e writes

- Instruções atômicas
  - test_and_set
  - Ticket lock
  - Queuing locks baseados em arrays
  - Queuing locks baseados em listas (MCS)

Locks com Reads e Writes

Sincronização de 2 processos (Dekker)

```
procedure acquire
    c[i] := true
    while c[otherguy]
        if turn = otherguy
            c[i] := false
            repeat until turn = i
            c[i] := true

procedure release
    turn := otherguy
    c[i] := false
```

Locks com Reads e Writes

Sincronização de 2 processos (Peterson)

```
procedure acquire
    c[i] := true
    turn := otherguy
    repeat until (not c[otherguy]) or
        (turn = i)

procedure release
    c[i] := false
```

Locks com Reads e Writes

Sincronização de N processos (Lamport)

```
procedure acquire
    loop
        b[i] := true
        x := i          // most recent process
        if y <> 0       // to begin protocol
            b[i] := false
            repeat until y = 0
            continue
        y := i
        if x <> i       // contention: recover
            b[i] := false
            for j <> i in pid
                repeat until not b[j]
            if y <> i
                repeat until y = 0
                continue
        return

procedure release
    y := 0
    b[i] := false
```

Lock Test_and_set

```
procedure acquire (L : ^lock)
    repeat until test_and_set (L) = false
procedure release (L : ^lock)
    L^ := false
```

Problemas:

- Starvation possíivel
- Muita contenção por memória e rede

Lock MCS

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil
        I->locked := true
        predecessor->next := I
        repeat while I->locked

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil
        if compare_and_swap (L, I, nil)
            return
        repeat while I->next = nil
    I->next->locked := false
```

Barreiras

- Centralizada
- Combining tree
- Dissemination
- Tournament

Barreira Centralizada

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    // each processor toggles its own sense
    local_sense := not local_sense
    if fetch_and_decrement (&count) = 1
        count := P
        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense = local_sense
```

Dissemination Barrier

```
type flags = record
    myflags : array [0..1][0..LogP-1] of Boolean
    partflags : array [0..1][0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags

// on proc i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is
//         false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//     allnodes[i].partflags[r][k] points to
//         allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for i : integer := 0 to LogP-1
        localflags^.partflags[parity][i]^ := sense
        repeat until
            localflags^.myflags[parity][i] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Sincronização Bloqueante

Spinning gasta ciclos de processador

Se o processador tiver outras coisas para fazer (ex. devido a multiprogramação), talvez seja melhor bloquear

Técnicas de sincronização bloqueante

- Semáforos

- Monitores

- Regiões críticas condicionais

Semáforos

```
P(sem)
    sem.val := sem.val - 1
    if sem.val < 0
        bloqueia

V(sem)
    sem.val := sem.val + 1
    if sem.val <= 0
       desbloqueia
```

Exemplo: Bounded Buffer

```
shared buf : array [1..N] of data
shared front, rear : integer := 1
shared empty, full : sem := N, 0

procedure deposit (m : data)
    P (empty)
    buf[rear] := m
    rear := rear mod N + 1
    V (full)

procedure fetch : data
    P (full)
    m : data := buf[front]
    front := front mod N + 1
    V (empty)
    return m
```

Implementação de Semáforos

```
procedure P (var S : sem)
        acquire (L)
        N := N - 1
        if N < 0
            put self on Q
            release (L)
            yield
        else
            release (L)

procedure V (var S : sem)
        acquire (L)
        N := N + 1
        if N <= 0
            move from Q to RL
        release (L)
```

Monitores

Problemas com semáforos

- Muito baixo níivel

- Chamadas ficam espalhadas pelo código

Solução: Monitores

Monitores

Componentes

- Operacões

- Estado interno

- Filas de condição

Apenas uma operação ativa a cada momento

Quando monitor ocupado, processo que chama bloqueado

Qualquer operação pode esperar por/sinalizar condição

Fila de condição != semáforo: se não há processos na fila $\rightarrow$ signal = no-op, enqto V != no-op

Exemplo: Bounded Buffers

```
monitor BB
    export deposit, fetch
    buf : array [1..N] of data
    front, rear : integer := 1
    slots : integer := 0
    emptied, filled : condition

    procedure deposit (m : data)
        if slots = N
            wait (emptied)
        buf[rear] := m
        rear := rear mod N + 1
        slots := slots + 1
        signal (filled)

    procedure fetch : data
        if slots = 0
            wait (filled)
        m : data := buf[front]
        front := front mod N + 1
        slots := slots - 1
        signal (emptied)
        return m
```

Semântica de Monitores

O que acontece qdo um processo P sinaliza uma condição pela qual Q está esperando?

- P espera ate que Q termine ou espere em outra condição (Hoare)

- Q espera ate que P termine ou espere em outra condição (Mesa)

Problemas com semântica Hoare: Monitores aninhados e excesso de trocas de contexto

Problema com semântica Mesa: Q precisa re-testar condição qdo for executar

Monitores x Semáforos

Semaforos	Variaveis de Condicao
Podem ser usados em qq local do programa, mas nao em monitor	Somente em monitores
Wait() nem sempre bloqueia o caller (pe, qdo cont > 0)	Wait() sempre bloqueia o caller
Signal() libera thread bloqueada ou incrementa contador	Signal() libera thread bloqueada ou e perdido (no-op)
Se Signal() libera thread bloqueada, ambos continuam (caller e thread)	Se Signal() libera thread o caller yields o monitor (H) ou continua (M) Somente um continua

Regiões Críticas Condicionais

```
var v: shared T;
```

```
region v when B do S;
```

Var compartilhada v, expr booleana B e seqüência de comandos S (seção críitica)

Regiões que se referem a mesma var se excluem

Qdo processo entra na região, B avaliada. Se B == true, S executada. Se B == false, exclusão liberada e processo atrasado até que (B == true e nenhum outro processo em região associada a var)

Exemplo: Bounded Buffers

```
var buffer: shared record
    pool: array [0..n-1] of data;
    count, in, out: integer;
    end;

region buffer when count < n do
    pool[in] = nextp;
    in = in+1 mod n;
    count = count + 1;
end;

region buffer when count > 0 do
    nextc = pool[out];
    out = out+1 mod n;
    count = count - 1;
end;
```

Sincronização – Hardware

Suporte de hardware para sincronização

- test_and_set, fetch_and_phi, compare_and_swap
- load_linked e store_conditional
- Hardware para enfileirar processos (QOLB)
- Barramento de sincronização
- Barreiras em hardware
- Redes de interconexão com combining

Barreiras em Hardware

Redes com Combining

Problemas Tradicionais de Sincro

- Bounded buffer
- Leitores e escritores: leitores concorrentes, escritores sequenciais
- Escalonadores: processos pedem servico e são escalonados de acordo com políítica pré-estabelecida
- Dining philosophers

Leitores e Escritores

```
Writer:
    P(wrt);
        ...
        writing is performed
        ...
    V(wrt);


Readers:
    P(mutex);
        readcount = readcount + 1;
        if readcount = 1 then P(wrt);
    V(mutex);
        ...
        reading is performed
        ...
    P(mutex);
        readcount = readcount - 1;
        if readcount = 0 then V(wrt);
    V(mutex);
```

Dining Philosophers

```
Philosopher i:
    repeat
        P(chopstick[i]);
        P(chopstick[i+1 mod 5]);
            ...
            eat
            ...
        V(chopstick[i]);
        V(chopstick[i+1 mod 5]);
            ...
            think
            ...
```

Problema?

Próxima Aula

Escrever monitores com semáforos e vice-versa. Explique.

Escrever uma solução livre de deadlock e starvation para o problema dos dining philosophers. Explique.