SO: Sincronização

Sistemas Operacionais
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

Mapa da Disciplina

- Fim da Seção de Processos
  - SOs geralmente são lecionados em três grandes temas
    - Processos
    - Memória
    - Arquivos
- Silberschatz Ed9
  - Sincronização: Capítulo 5
  - Deadlocks: Capítulo 7
- Tanenbaum (Sistemas Operacionais Modernos):
  - Fim do Capítulo 2
  - Parte do Capítulo 6 (Impasses)
- http://pages.cs.wisc.edu/~remzi/OSTEP/
  - Parte “Azul”: 25 até 34
- Depois disto vamos para segunda parte, memória

Lembrando de PThreads

- **int pthread_create(pthread_t *_thread_, const pthread_attr_t *_attr_, void *(*_start_routine_) (void *), void *_arg_);**
  - Cria nova Thread
  - Inicia a execução da Thread
  - Ponteiro para a Thread
  - Atríbutos
  - Ponteiro para função
  - Argumentos da Função

- **int pthread_join(pthread_t _thread_, void **_retval_);**
  - Espera a thread finalizar
  - Copia o valor de retorno para retval

Qual vai ser a saída do código seguinte

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo1.c

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo2.c

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo3.c

Respostas

1. Não Sei

Respostas

1. Não Sei

2. Não Sei

Respostas

1. Não Sei
2. Não Sei
3. Pior ainda, nem sei o sinal

Olhando no Nível de Instruções

`count++` em assembler:

```
a) MOV R1, $counter
b) INC R1
c) MOV $counter, R1
```

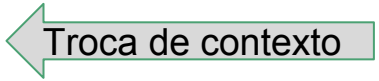
`count--` em assembler:

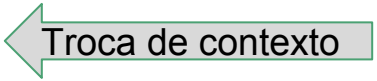
```
x) MOV R2, $counter
y) DEC R2
z) MOV $counter, R2
```

- Cada instrução é independente
- Interrupções podem ocorrer entre quaisquer duas instruções
  - Logo, trocas de contexto também podem ocorrer
- As sequências [a,b,c] e [x,y,z] podem ocorrer intercaladas

Falta de sincronização

```
a) MOV R1, $counter        R1 = 5
b) INC R1                  R1 = 6
x) MOV R2, $counter        R2 = 5
y) DEC R2                  R2 = 4
c) MOV $counter, R1        counter = R1 = 6
z) MOV $counter, R2        counter = R2 = 4
```





Condições de corrida

Dados e estruturas são acessados de forma concorrente

Resultado depende da ordem de execução dos processos

- Resultado é *indeterminado*

Necessidade de sincronização

Problema da seção crítica

Contexto

- Vários processos utilizam uma estrutura de dados compartilhada
- Cada processo tem um segmento de código onde a estrutura é acessada
- Processos executam a uma velocidade não nula (fazem progresso)
- O escalonamento e velocidade de execução são indeterminados

Garantir que apenas um processo acessa a estrutura por vez

Problema da seção crítica

Requisitos da solução:

- Exclusão mútua
  - Apenas um processo na seção crítica por vez
- Progresso garantido
  - Se nenhum processo está na seção crítica, qualquer processo que tente fazê-lo não pode ser detido indefinidamente
  - [Outra Forma de Escrever] Nenhum processo fora de sua região crítica pode bloquear outros
- Espera limitada
  - Se um processo deseja entrar na seção crítica, existe um limite no número de outros processos que entram antes dele

Controle de acesso à seção crítica

Considere dois processos, Pi e Pj

Processos podem compartilhar variáveis
para conseguir o controle de acesso

```
do {
    enter section
    // critical section
    leave section
    // remainder section
} while (1);
```

Como garantir as 3 condições?

1. Pensando em interrupções
2. Usando conceitos conhecidos while/for/if apenas

Para ajudar: Duas Threads i, j

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica
- Meio extrema, mas ok, funciona.
  - Pelo menos para a exclusão mútua
- Perdemos tudo que aprendemos de escalonamento
  - Além de outros problemas como processos fazendo tarefas de SO
- Vamos pensar em algo melhor

Tentativa 1

```
int turn; // variável de controle, compartilhada, inicializada para i ou j

do {
    // id da thread na variável i
    while(turn != i);
    // critical section.
    turn = j;
    // remainder section
} while (1);
```

Tentativa 2

```
int queue[2] = {0, 0}; // variável de controle, compartilhada

do {
    queue[i] = 1;
    while(queue[j]);
    // critical section
    queue[i] = 0;
    // remainder section
} while (1);
```

Tentativa 3: Solução de Peterson

```
int queue[2] = {0, 0}; // variável de controle, compartilhada
int turn = i;
do {
    queue[i] = 1;
    turn = j;
    while(queue[j] && turn == j);
    // critical section
    queue[i] = 0;
    // remainder section
} while (1);
```

Solução de Peterson

- Funciona em sistemas de um processador apenas
- Hardwares modernos com vários cores não garantem sequência de operações
- Caches criam a oportunidade de discórdia
  - Cada CPU observa um valor diferente
- Instruções atômicas de hardware para resolver isto
  - e.g., TSL e CTXT
- Mais complicada com n processos
  - Ainda mais se n é dinâmico


SCREW IT
I GIVE UP

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

```
int lock = 0;
do {
    while(test_and_set(&lock));
    // critical section
    lock = 0;
    // remainder section
} while (1);
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
int lock = 0; // compartilhada
do {
    boolean key = true;
    while(key) swap(&lock, &key);
    // critical section
    lock = 0;
    // remainder section
} while (1);
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
int lock = 0; // compartilhada
do {
    boolean key = true;
    while(key) swap(&lock, &key);
    // critical section
    lock = 0;
    // remainder section
} while (1);
```


SCREW IT
I GIVE UP

Exclusão mútua justa - cada processo passa a vez para o próximo na fila

```
int lock = 0; int n; //n tem o número de processos
do {
    waiting[i] = 1;
    while(waiting[i] && test_and_set(&lock));
    // critical section
    waiting[i] = 0;
    j = (i + 1) % n;
    while((j != i) && !waiting[j])
        j = (i + 1) % n;
    if (j == i) { lock = 0; }
    else { waiting[j] = 0; }
    // remainder section
 } while (1);
```

Ainda temos problemas?

Ainda temos problemas?

Todos os algoritmos com base em busy wait.

Sistema Operacional Ajuda

- Primitivas de block/wakeup (wait/notify em Java)
- Processo que chama block passa a esperar
- Processo que chama wakeup acorda um outro processo esperando (block)
  - Se existir, se não a vida continua

Semáforos

Primitivas de alto nível oferecidas pelo sistema operacional ou linguagem

Conceitualmente, semáforo é uma variável inteira acessível por duas operações atômicas

Funciona com qualquer número de processos. Conceito com busy wait abaixo

```
wait(int *s) {
   while(*s <= 0);
   (*s)--;
}
// acquire, down, lock
```

```
signal(int *s) {
   (*s)++;
}

// release, up, unlock
```

Implementação de semáforos

```
typedef struct {
    int value;
    struct process *list;
} sem_t;

void wait(sem_t *s) {
   s->value--;
   if(s->value < 0) {
      add(me, s->list);
      block();
   }
}
```

```
void signal(sem *s) {
   s->value++;
   if(s->value <= 0) {
      p = remove(s->list);
      wakeup(p);
   }
}
```


LIES
LIES EVERYWHERE
memegenerator.net

Ainda Precisamos de Busy Wait

- wait
- signal
- Locks com base em espera ocupada
  - Bem codificadas, poucas instruções

Mutex

- Semanticamente: Semáforo de tamanho 1
- Apenas o processo/thread que adquire pode liberar
- Não precisa ser implementado com semáforos

Exclusão mútua com semáforos

```
do {
    wait(&mutex);
    // critical section
    signal(&mutex);
    // remainder section
} while (1);
```

Sincronização com semáforos

```
Processo Pi, task
inicializada com 0

// do something
signal(&task);
```

```
Processo Pj depende da
tarefa realizada por Pi

wait(&task);
// do something else
```

Controle de dispositivos com semáforos

```
Computador com N dispositivos que não podem ser compartilhados;
int dispositivo inicializado com N;
```

```
wait(&dispositivo);
// obtém dispositivo
// usa dispositivo
// libera dispositivo
signal(&dispositivo);
```

Deadlocks

```
// P1
wait(S);
wait(Q);
…
signal(S);
signal(Q);
```

```
// P2
wait(Q);
wait(S);
…
signal(Q);
signal(S);
```

```
P1: wait(S);
P2: wait(Q);
P1: wait(Q);
P2: wait(S);
// pwned
```