MAC0422 - Sistemas Operacionais

Daniel Macêdo Batista

IME - USP, 17 de Setembro de 2020

Roteiro

Threads

O problema da seção crítica

Semáforos

Threads

Motivação

- Muitas vezes o modelo de processos é muito rígido para resolver alguns problemas
- Por exemplo, um software que tem partes bem definidas que possam ser realizadas em paralelo mas que precisam manter comunicação entre si por meio de variáveis na memória seria difícil de ser escrito como múltiplos processos (cada processo tem seu próprio espaço **privado** de endereço!)
- Threads ajudam a resolver esse problema! Com elas, variáveis podem ser vistas e podem ser usadas para comunicação entre as threads

Threads

- As threads podem ser vistas como miniprocessos mas, diferente de processos, tem a vantagem de compartilhar o espaço de endereço
- O fato de não precisar de uma nova entrada na tabela de processos, nem criação de área de memória específica, faz a gerência das threads ser mais leve do que a gerência de processos (criar uma thread em certos SOs pode ser até 100 vezes mais rápido do que criar um processo)
- As threads podem rodar em núcleos separados, tirando proveito do paralelismo não apenas entre processos mas em vários processos (mas transformar códigos monothread em multithread nem sempre é trivial)

Threads

- Um jogo digital é um bom exemplo que mostra a vantagem de multithread. Uma thread pode cuidar do áudio do jogo, outra da renderização das imagens na tela e outra pode cuidar da comunicação via rede. Todas compartilhando a mesma área de memória. Assim, se vem uma ação de um jogador via rede, isso já pode ser exibido na tela de forma que o usuário vai acreditar que foi instantâneo.

Em C, no Linux

- `pthread.h`
- Funções `pthread_create` e `pthread_join`
- Outras funções: `pthread_exit`, `pthread_tryjoin_np`, `pthread_timedjoin_np`, ...
- Importante bloquear as threads pela lógica do código. Também há funções e técnicas para isso

Outros argumentos sobre quando usar

- A eficiência em fazer a comunicação por meio de variáveis na memória é maior do que pipes e variáveis de ambiente
- Útil quando um software precisa atender diversos clientes/usuários mantendo o status de cada um deles (o código para atender cada um é igual) – Servidores de rede por exemplo com 1 **dispatcher** e vários **workers** – Desempenho melhora significativamente

Processos vs Threads

- Processos podem ser vistos como unidades que agrupam recursos juntos
- Threads podem ser vistas como unidades que são escalonadas para execução nas CPUs
- As threads permitem que execuções múltiplas aconteçam dentro de um único processo.
- Múltiplas threads de um processo rodando em paralelo é análogo a múltiplos processos monothread de um computador rodando em paralelo (Threads compartilham espaço de endereço e outros recursos. Processos compartilham a memória física, discos, etc...)
- Mesmo que a CPU não tenha múltiplas unidades de processamento pode tirar proveito das threads (alguma hora vai ter uma atividade de E/S)
- Hierarquia entre threads pode existir mas tem que ser feita manualmente pelo programador

Usando threads

- Na prática todo programa começa monothread. A thread principal tem poder de criar outras threads.
- Cada thread independente do SO normalmente vai ser criada com uma função como parâmetro. Essa função vai ser executada para aquela thread
- Algumas funções básicas que podem ser realizadas com threads: criar, terminar, esperar uma thread específica terminar, liberar a CPU para outra thread (útil para escalonamento. Não faria sentido para processos)

Consequências da área de memória compartilhada

- As variáveis globais são compartilhadas (bom e ruim ao mesmo tempo)
- Não há controle algum no acesso a essas variáveis (desnecessário pois as threads são de um mesmo processo e por sua vez serão de um mesmo usuário)
- Mas nem tudo é compartilhado. É preciso manter **por thread**: Program counter, registradores, pilha, estado (aqui no estado é a mesma ideia dos estados de processos)
- Se são implementadas a nível de usuário, o escalonador pode ser personalizado para cada processo

O problema da seção crítica

Quais são as ações atômicas?

- Ação atômica realiza uma transformação de estado (mudança de variáveis) de forma indivisível
- Estados intermediários não podem ser vistos por outros processos
- Depende do hardware
  - Atribuições de variáveis são atômicas

Exemplo


```
int x = 0;
Thread 1: x++;
Thread 2: x++;
```

Qual o valor de x no final?

Considerações

- Tipos básicos (e.g. int) são armazenados em posições de memória que são lidas e escritas de forma atômica
- Manipulação de valores depende da cópia deles para registradores, operações nesses registradores e cópia de volta para a memória
- Cada processo tem seu próprio conjunto de registradores (mudança de contexto)
- Resultados intermediários resultantes de expressões complexas são armazenados em registradores ou na memória particular de cada processo

O problema da seção crítica

- n processos/threads repetidamente executam uma seção de código crítica e uma seção de código não crítica

```
while (true) {
   protocolo de entrada;
   secao critica;
   protocolo de saida;
   secao nao critica
}
```

- Considerando que um processo que entra na seção crítica, sairá alguma hora

Semáforos

Para que semáforos?

- Surgem para evitar estados indesejáveis na execução de códigos com múltiplas threads
- (Nas estradas)

  Mecanismos que sinalizam condições para garantir exclusão mútua de seções críticas da estrada.
- (Em programação concorrente)

  Mecanismos básicos de sinalização usados para implementar exclusão mútua e sincronização

O que é um semáforo?

- Um tipo especial de variável compartilhada proposto por Dijkstra
  - Assume valores inteiros não negativos
- Manipulado apenas por duas operações atômicas
  - `V`: usada para sinalizar a ocorrência de um evento
  - `P`: usada para atrasar um processo até que um evento ocorra

- Dijkstra era holandês
- V: holandês **v***erhogen* = incremento

  Incrementa o valor de um semáforo
- P: holandês **p***robeer te verlagen* = tentar reduzir

  Espera até o valor de um semáforo ser positivo e então decrementa o valor

Em algoritmos


```
sem s; /* Semaforo s criado. Valor inicial = zero */

sem lock = 1; /* Pode inicializar com qualquer valor
               * nao negativo */

sem forks[5] = ([5] 1); /* Vetor de semaforos criado
                         * com todos = 1 */
```

Em algoritmos

- semáforos são chamados de mutex quando o valor dos mesmos pode ser apenas 0 ou 1

Em algoritmos

- As únicas formas de manipular semáforos são as operações `P` e `V`
- Cada operação recebe um semáforo como argumento e realiza de forma atômica algo com o efeito similar a:

```
P(s):  while (s==0) skip; s = s - 1;
V(s):  s = s + 1;
```

- `V` incrementa `s` de forma atômica
- `P` decrementa `s` de forma atômica mas apenas se `s` for positivo. Se não for, espera