Escalonamento

Eduardo Ferreira dos Santos

Ciência da Computação
Centro Universitário de Brasília – UniCEUB

Abril, 2017

Sumário

1. Multiprogramação
2. Escalonamento
3. Concorrência
   - Memória compartilhada
   - Troca de mensagens
   - Prevenção de deadlock

1 Multiprogramação

2 Escalonamento

3 Concorrência
- Memória compartilhada
- Troca de mensagens
- Prevenção de deadlock

Multiprogramação

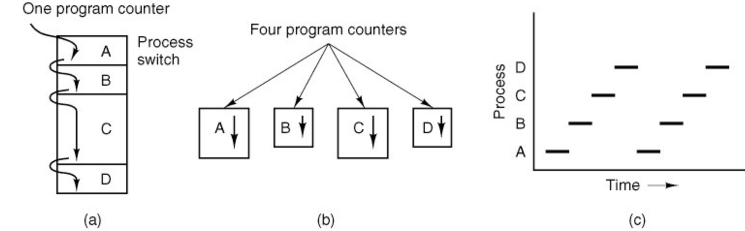


Figura 1.1: (a) Multiprogramação para quatro programas (b) Modelo conceitual de quatro processos sequenciais independentes (c) Somente um programa está ativo a cada momento [Tanenbaum and Machado Filho, 1995]

UniCEUB
Centro Universitário de Brasília

Estados dos processos

Durante o ciclo de vida de um processo ele passa por diferentes estados. Em sistemas Unix [Guarezi and Silva, 2010] são:

run
: Está sendo executado no processador;

ready ou executável
: Dispõe de todos os recursos que precisa e está pronto para ser executado;

sleep ou dormente
: Bloqueado à espera de algum recurso, e só pode ser desbloqueado se receber um sinal de outro processo;

zumbi
: Caso cada vez mais raro, onde um processo é criado por um programa, que por sua vez é finalizado antes de receber o resultado do processo;

parado
: Recebeu ordem do administrador para interromper a execução. Será reiniciado se receber um sinal de continuação (CONT).

UniCEUB
Centro Universitário de Brasília

Estados dos processos (Gráfico)

Estado de Execução

Estado de Término

Estado de Espera

Estado de Pronto

Estado de Criação

Figura 1.2: Estados dos processos [Chagas, 2016]

Concorrência

- Paradigma: controlar/restringir o acesso ao recurso em determinado espaço de tempo;
- O controle de acesso aos recursos é realizado através de eventos;
- Eventos inesperados pode causar um desvio inesperado no fluxo de execução.
- Definição [Chagas, 2016]:
  1. O programa perde o uso do processador;
  2. O programa retorna para continuar o processamento;
  3. O estado do programa deve ser idêntico ao do momento em que foi interrompido.
- O programa continua a execução exatemente na instrução seguinte.

Troca de Contexto

process P_0 operating system process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

interrupt or system call

executing

save state into PCB_1

idle

reload state from PCB_0

executing

Figura 1.3: Troca de Contexto [FARINES and MELO, 2000]

Programação concorrente

- Na programação concorrente existe mais de uma tarefa sendo executada ao mesmo tempo. Ex.: Fatorial
- No caso de múltiplas tarefas é necessário haver comunicação entre elas.

Memória compartilhada As tarefas compartilham área de memória;

Troca de mensagens Sinais trocados entre processos.

1 Multiprogramação

2 Escalonamento

3 Concorrência
- Memória compartilhada
- Troca de mensagens
- Prevenção de deadlock

Conceitos

- Objetivo: maximizar a utilização de CPU em programação concorrente;
- O escalonador é o mecanismo que seleciona um dos processos disponíveis na fila de pronto para ir à execução;
- Troca de contexto entre os processos;
- Organização da fila de prioridades.

Escalonamento

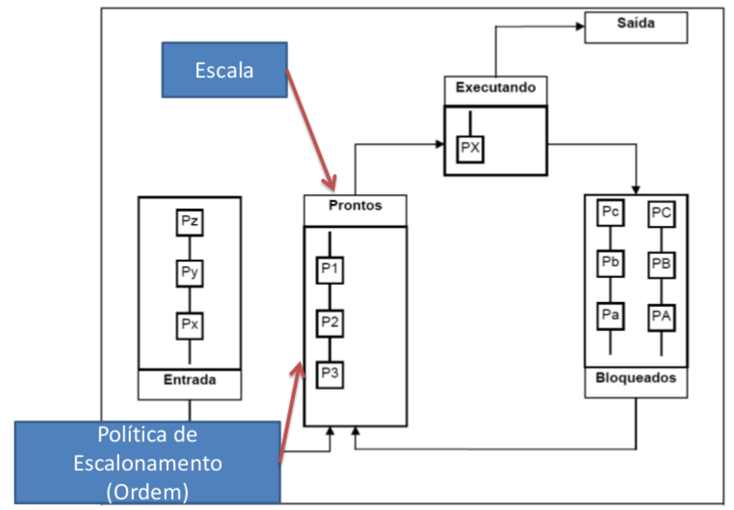


Figura 2.1: Descrição do escalonamento [Chagas, 2016]

Troca de contexto

process P_0 | operating system | process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

interrupt or system call

executing

save state into PCB_1

idle

reload state from PCB_0

executing

Figura 2.2: Organização da fila de pronto [Galvin et al., 2013]

Escalonadores

Short-term scheduler Seleciona os processos que vão para a CPU
- Algumas vezes o único escalonador disponível;
- Bastante utilizado, ou seja, precisa ser rápido.

Long-term scheduler Seleciona os processos que devem ir à fila de pronto
- Não é tão utilizado (menos frequente);
- Controla o grau de multiprogramação.

- Os processos podem ser descritos como:

  I/O-bound Gasta mais tempo realizando operações de I/O do que computações;

  CPU-bound Mais tempo realizando computações.

1 Multiprogramação

2 Escalonamento

3 Concorrência
- Memória compartilhada
- Troca de mensagens
- Prevenção de deadlock

Conceitos

- Na programação concorrente existe mais de uma tarefa sendo executado ao mesmo tempo. Ex.: Fatorial
- No caso de múltiplas tarefas é necessário haver comunicação entre elas.

Memória compartilhada As tarefas compartilham área de memória;

Troca de mensagens Sinais trocados entre processos.

Comunicação entre processos

- Gerência de recursos de memória compartilhada: condição de corrida
  - Exclusão mútua;
  - Semáforo;
  - Monitor.
- Comunicação por troca de mensagens: deadlocks
  - Leitura assíncrona;
  - Método *rendezvous*.

1. Multiprogramação
2. Escalonamento
3. Concorrência
   - Memória compartilhada
   - Troca de mensagens
   - Prevenção de deadlock

Condição de corrida

Situações onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende de quem roda quando.

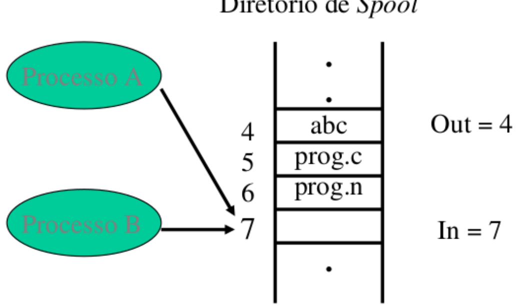


Figura 3.1: Exemplo da condição de corrida

Solução para condição de corrida

- Uma boa solução para a condição de corrida requer quatro condições [Favacho, 2009]:
  1. Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes;
  2. Nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito do número de processadores disponíveis no sistema;
  3. Nenhum processo que esteja executando fora de sua região crítica pode bloquear a execução de outro processo;
  4. Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica.

Exclusão mútua [Chagas, 2016]

- Solução: impedir que mais de um processo acesse o dado ao mesmo tempo.
- Deve ser executada somente quando um dos processos estiver acessando o recurso compartilhado;
- A parte do código onde o acesso ao recurso é feito é chamada de região crítica.

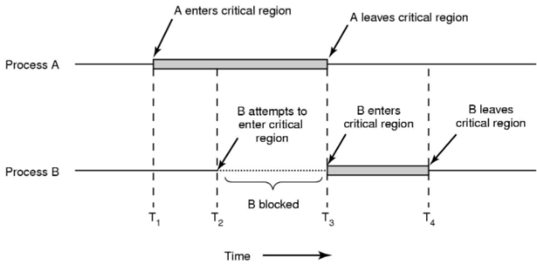


Figura 3.2: Região crítica

Soluções para exclusão mutua I

Inibição das interrupções Inibir as interrupções de cada processo logo após o ingresso na região crítica, habilitando-as novamente após deixá-las.

- Desabilitar interrupções deve ser uma atribuição do kernel;
- Interferir no kernel pode não ser uma boa ideia.

Variáveis de travamento (locks) Utilização de variável única compartilhada (*lock*) que pode assumir 0 ou 1.

- Se dois processos chegam ao mesmo tempo?
- Condição de corrida.

Soluções para exclusão mutua II

Chaveamento obrigatório Utiliza a variável inteira `turn` [Tanenbaum and Machado Filho, 1995].

- A variável `turn` indica a vez de quem é de entrar na região crítica;
- Se um dos processos for mais lento que o outro requer a solução estritamente alternada;
- Espera ocupada: teste contínuo do valor esperando por uma mudança.

Listing 1: a

```
while (TRUE) {
  while (turn!=0)    /* çlao */
  critical_region();
  turn = 1;
  non_critical_region();
}
```

Listing 2: b

```
while (TRUE) {
  while (turn!=1)    /* çlao */
  critical_region();
  turn = 0;
5 non_critical_region();
}
```

Problema do produtor-consumidor

- Dois processos compartilham um *buffer* de tamanho fixo;
- Um põe a informação dentro do *buffer*: produtor;
- Outro retira a informação do *buffer*: consumidor;
- Problema: produtor quer colocar um item no *buffer*, mas já está cheio;
- Solução: colocar o produtor para dormir (*sleep*) e só acordar quando o consumidor remover um ou mais itens;
- Grande possibilidade de gerar condição de corrida: perda do envio de sinal para acordar (*wakeup*) quando o processo ainda não está dormindo.

Semáforos [Favacho, 2009]

- Baseado em um tipo de variável que possui dois estados: UP e DOWN.
  1. O semáforo fica associado a um recurso compartilhado;
  2. Se o valor da variável semáforo for diferente de zero, nenhum processo está utilizando o recurso; caso contrário,o processo fica impedido do acesso;
  3. Sempre que deseja entrar em sua região crítica, o processo executa uma instrução DOWN;
  4. Se o semáforo for maior que 0, este é decrementado de 1, e o processo que solicitou a operação pode executar sua região crítica;
  5. Entretanto, se uma instrução DOWN é executada em um semáforo cujo valor seja igual a 0, o processo que solicitou a operação ficará no estado de espera;

Semáforos (cont.) [Favacho, 2009]

6. Além disso, o processo que está acessando o recurso, ao sair de sua região crítica, executa uma instrução UP, incrementando o semáforo de 1 e liberando o acesso ao recurso;
7. A verificação do valor do semáforo, a modificação do seu valor e, eventualmente a colocação do processo para dormir são operações atômicas;
8. Operações atômicas são únicas e indivisíveis;
9. Os semáforos aplicados ao problema da exclusão mútua são chamados de mutex (*mutual exclusion*) ou binários, por apenas assumirem os valores 0 e 1.

Implementação [Favacho, 2009]

```
#define N 100
typedef int semaphore
semaphore mutex = 1; /* controla ãregio ícrtica */
semaphore empty = N; /* controla çõposies vazias */
semaphore full = 0; /* controla çõposies ocupadas */
```

```
void producer(void ) {
  while (TRUE) {
    item = produce_item ();
    down (&empty);
    down (& mutex );
    insert_item(item ); /* R_critica
        */
    up(&mutex );
    up(& full );
  }
}
```

```
void consumer(void ) {
  while (TRUE) {
    down (& full );
    down (& mutex );
    item = remove_item (); /* R_cr í
        tica */
    up(&mutex );
    up(&empty );
    consume_item(item);
  }
}
```

1. Multiprogramação

2. Escalonamento

3. Concorrência
   - Memória compartilhada
   - Troca de mensagens
   - Prevenção de deadlock

Problema do deadlock

- Em ambiente de multiprogramação diversos processos podem competir por um número finito de recursos;
- O processo não pode mudar de estado enquanto estiver aguardando algum outro recurso;
- Exemplo de deadlock [Favacho, 2009]:
  - Um processo A bloqueia o registro R1;
  - Um processo B bloqueia o registro R2;
  - O processo A entra em espera pois precisa utilizar o registro R2 e;
  - O processo B entra em espera pois precisa utilizar o registro R1.
- Os processos ficam bloqueados para sempre: deadlock.

Definição

Um conjunto de processos estará em situação de deadlock se todos os processos pertencentes ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer. [Favacho, 2009]

- O número de processos, bem como, o número e tipo dos recursos não são importantes;
- Isso é válido para qualquer tipo de recurso, tanto para hardware como para software.

Condições para Deadlock

- Há quatro condições para ocorrência de *deadlock* [Favacho, 2009]:

  Exclusão mútua Apenas um processo de cada vez pode utilizar o recurso;

  Prende e espera Um processo bloqueia os recursos que precisa e aguarda pelos que estão sendo utilizados pelos outros processos;

  Não preempção Um recurso pode ser liberado apenas voluntariamente pelo processo após o mesmo ter completado sua tarefa;

  Espera circular Cada um dos processos espera um recurso que está sendo usado por um outro processo em uma fila.

Grafo

- Representação:
  - Processos (círculos);
  - Recursos (retângulos);
  - Instâncias dos recursos (pontos).
- Uma aresta orientada é denominada:
  - P1 → R1;
  - Arco.

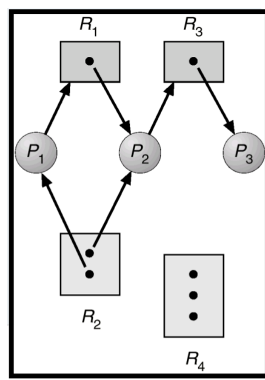


Figura 3.3: Grafo de alocação de recursos [Favacho, 2009]

Grafo

- P1 está bloqueando R2 e aguardando R1;
- P2 está bloqueando R1 e R2 e aguardando R3;
- P3 está bloqueando R3.

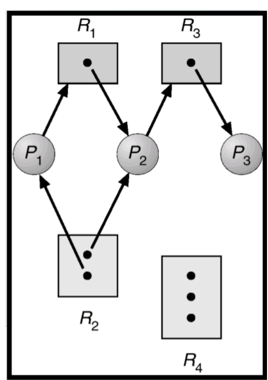


Figura 3.4: Grafo de alocação de recursos [Favacho, 2009]

Soluções

- Como lidar com *deadlock*?
  - Usar um protocolo de prevenção para garantir que não está acontecendo;
  - Permitir que entre em *deadlock*, detectá-lo e recuperá-lo;
  - Ignorar o problema, fingindo que nunca acontecerá.
- A maioria dos sistemas operacionais prefere fingir que nada está acontecendo [Favacho, 2009];
- O que tem menor impacto? Lidar com o *deadlock* ou impedir recursos simultâneos?

1 Multiprogramação

2 Escalonamento

3 Concorrência
- Memória compartilhada
- Troca de mensagens
- Prevenção de deadlock

Exclusão mútua

- Se não houver alocação exclusiva, nunca ocorrerá *deadlock*;
- Recursos compartilháveis, como arquivos abertos, não necessitam de acesso por exclusão mútua. Assim, nunca ocorrerá um *deadlock*;
- Não é possível prevenir *deadlock* negando a exclusão mútua sempre. Alguns recursos precisam da garantia.

Manter e esperar

- Impedir que alguém que tenha algum recurso solicite outro já previne o *deadlock*;
- Fazer que um processo inicie com todos os recursos que vai precisar: possível solução;
- Só permite solicitar um recurso quando não estiver consumindo nenhum outro.
    - Nesse caso pode acontecer *starvation* se o recurso for popular.

Preempção

- Se solicitar um recurso e não estiver disponível, libera todos os outros que tinha solicitado;
- Somente se reinicia o processo quando puder reaver seus recursos antigos;
- Só se aplica a recursos cujo estado pode ser salvo.

Espera circular

- Impor ordenação a todos os tipos de recursos;
- O grafo de alocação de recursos nunca conterá ciclos.

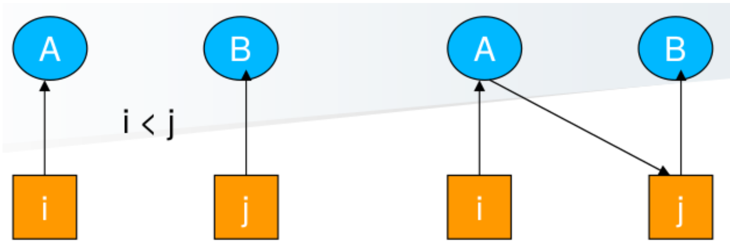


Figura 3.5: Prevenção de deadlock [Favacho, 2009]

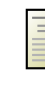
OBRIGADO!!!
PERGUNTAS???

Chagas, F. (2016).
Notas de aula do prof. fernando chagas.

FARINES, J. M. and MELO, R. (2000).
Sistemas de Tempo Real, volume 1.
IME-USP.

Favacho, A. (2009).
Notas de aula da Profa. Aletéia Favacho.

Galvin, P. B., Gagne, G., and Silberschatz, A. (2013).
Operating system concepts.
John Wiley & Sons, Inc.



Guarezi, D. J. and Silva, E. B. (2010).
Processos em windows e unix.
Disponível em:

http://www.inf.ufsc.br/∼magro/PROCESSOS%20EM%20WINDOWS%2

Acessado em 28/01/2011.

Tanenbaum, A. S. and Machado Filho, N. (1995).

Sistemas operacionais modernos, volume 3.
Prentice-Hall.