1. SOPE – Programação em UNIX (introdução)

Programa-> conjunto de instruções e informação para atingir um objectivo. Quando em execução, designa-se processo.

Processo-> instância de um programa de computador, controlado pelo sistema operativo. Engloba um espaço de endereçamento, variáveis, recursos necessários (mutex).

Threads-> forma de um processo se dividir em várias tarefas (podem ser executadas em paralelo).

Processos vs Threads

- criar um processo é caro em termos de tempo/memória/sincronização;

- as threads podem ser criadas sem ter de clonar o processo inteiro;

- as threads são criadas no espaço do usuário;

- tempo gasto para troca de threads é menor.

Código Fonte-> conjunto de palavras ou símbolos escritos de forma estruturada contendo instruções numa linguagem de programação.

Estrutura (código fonte)

- Partes especificas

- Partes gerais

    - Declarações

    - Bibliotecas

        - Estáticas

        - Dinâmicas ou partilhadas

Bibliotecas dinâmicas

- Vantagens: partilha de espaço de memória, actualização automática de aplicações, ficheiro executável tem tamanho mínimo;

- Desvantagens: mobilidade, desempenho.

‘Make’-> programa de computador para facilitar a compilação de código fonte, interpretador de programas feito num ficheiro de texto (‘makefile’) com linguagem própria (blocos). É constituído pelo produto final, dependências e instruções.

2. SOPE – Princípios de sistemas operativos

Um computador é formado por hardware com objectivos como manipular dados (semi) automaticamente, facilitar a vida dos utilizadores e executar programas.

Sistema Operativo-> programa de actua como intermediário entre utilizadores e o hardware, gestor de recursos (tempo, espaço).

3.SOPE – Processos, threads e eventos

Multi-tarefa-> execução, em paralelo, de programas no mesmo computador, cada instância em execução denomina-se processo.

Um processo tem associado:
um programa, uma zona de memória e uma entrada na tabela de processos.

Processos vs Programas

- programa é um ficheiro executável;

- um processo é um objecto do sistema operativo que suporta a execução dos programas;

- um processo pode executar vários programas;

- um programa ou partes dele podem ser partilhadas por diversos processos.

O SO deverá ser capaz de criar, suspender e reiniciar a execução de processos assim como suportar a comunicação entre processos.

Os processos “competem por recursos” para executar as suas tarefas e cabe ao SO fazer o escalonamento dos processos.

Estados (processo)

- Execução: foi-lhe atribuído CPU;

- Bloqueado: o processo está logicamente impedido de prosseguir, porque lhe falta um recurso;

- Pronto a executar: aguarda escalonamento.

Multiprogramação (benefícios), é possível num sistema multi-programado, existir apenas um processador, sendo os vários processados simultaneamente.

- Maximiza o tempo de utilização de CPU;

- Utilização do CPU = 1 - p^n (n – nº de processos, p – fracção de tempo em espera por I/O)

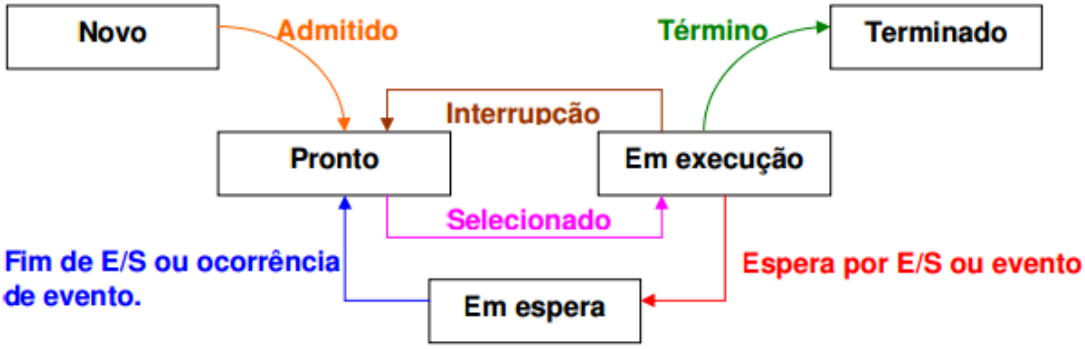

Diagrama de estados de um processo

Os processos são identificados por um inteiro *pid* e relacionam-se de forma hierárquica: o processo filho herda o ambiente do pai e sabe o *pid* do pai. Se o processo pai terminar, os subprocessos são adoptados pelo *pid* = 1.

Criação de um processo

Id = fork;

pai = *pid* do filho

filho = 0

-1 -> erro

```
int pid;
pid = fork();
if(pid == 0) {/* código do filho */}
else {/* código do pai */}
```

Terminação

- termina o processo e liberta todos os recursos detidos pelo processo;

- assinala ao processo pai a terminação (`void exit(status)`, status – permite passar ao pai o estado em que o processo terminou (< 0 = erro)).

Em UNIX existe uma função para sincronizar o processo pai com a terminação do processo filho que bloqueia o processo pai até que o filho termine (`int wait(int *status)`, status – estado de terminação do filho, retorna o *pid* do processo terminado).

Execução de um programa

- o *fork* apenas permite lançar processos com o mesmo código;

- a função *exec* permite substituir a imagem do processo onde é invocada;

- não há retorno numa chamada com sucesso.

Exec

- `int execl (char *fich, arg0, arg1, …, argN, 0);`

- `int execv (char *fich, *argv[]), fich` – caminho do ficheiro, argv – argumentos.

Threads-> múltiplos fluxos de execução do mesmo processo.

Threads vs Processos

- Processos obrigam ao isolamento (espaços de endereçamento disjuntos, dificuldade em partilhar);

- Threads garantem eficiência na criação e comutação.

Criação de threads

- `err = pthread_create(&tid, attr, func, arg)`, tid – id tarefa, attr – atributos, func – função a executar, arg – parâmetros para a função;

- `pthread_exit(void *value_ptr);`

- `pthread_join(pthread_t thread, void **value_ptr)` -> se a tarefa alvo não terminou bloqueia-se.

Programação em ambiente multitarefa

- threads partilham o mesmo endereçamento (acesso às mesmas variáveis globais);

- modificação das variáveis globais deve ser feito com precaução (solução: sincronização).

Implementação:

Threads utilizador:

- as tarefas são implementadas numa biblioteca no espaço de endereçamento do utilizador;

- núcleo apenas vê o processo;

- processo guarda lista de tarefas;

- comutação entre tarefas é explícita (pode ser contornado usando interrupções).

Threads núcleo:

- implementação no núcleo do SO (mais comuns, lista de tarefas é gerido pelo núcleo).

Certos acontecimentos devem ser tratados, embora não seja possível prever a sua ocorrência (ctrl-c, time-out, div-0).

Evento-> rotinas para tratamento de acontecimentos assíncronos e excepções.

Sinais-> acontecimentos assíncronos em UNIX.

Tratamento de sinais

- omissão – termina o processo;

- ignorado – alguns sinais não podem ser ignorados (ex: SIGKILL);

4. SOPE – Escalonamento e sincronização

Politica de escalonamento

- Conveniência: redução dos tempos de resposta, sendo justo para os processos;

- Eficiência: débito e máxima utilização do CPU.

Critérios de escalonamento:

- IO-bound ou CPU-bound;

- interactivo ou não;

- urgência de resposta (tempo real);

- comportamento recente;

- necessidade de periféricos especiais;

- por prioridades.

Primitivas de despacho

- bloqueia(evento): coloca processo na fila de processos parados à espera do evento e invoca próximo passo;

- liberta(evento) ou liberta(processo, evento): se o processo não está à espera de mais eventos, coloca-o na lista de espera;

- próximo_processo(): selecciona um dos processos existentes na lista de processos prontos a executar e executa a comutação.

Escalonamento de processos

O escalonamento de processos pode ocorrer de duas maneiras:

- Se, após lhe ter sido atribuído o CPU, nunca mais lhe for retirado (escalonamento cooperativo). Sensível às variações de carga;

- Se o CPU lhe for retirado (escalonamento com desafectação forçada). Sistema responde melhor, comutação de contexto é cara.

Os processos prontos são seriados numa fila (ready list), lista ligada para process control block. Quando um processo é escalonado, é retirado da ready list e posto a executar. O processo pode perder o CPU se aparecer um com maior prioridade, pedido de I/O (bloqueado) ou o quantum expira (pronto). Deve-se escalonar um processo:

- quando um processo passa de a-executar a bloqueado;

- quando um processo passa a pronto;

- quando se termina uma operação de I/O;

- quando um processo termina.

Algoritmos de escalonamento

Objectivo: equidade, balanceando e forçar que as regras impostas pelo algoritmo são respeitadas.

FCFS (First come, first served):

- ready list é uma lista FIFO;

- o processo é colocado no fim da fila e seleccionado o da frente;

- método cooperativo;

- nada apropriado para ambiente interactivos;

- tempo de espera com grandes flutuações dependendo da ordem de chegada e das características dos processos;

- simplicidade de implementação.

SJF (shortest job first):

- escalonar processo mais curto primeiro;

- possibilidades:

  - desafectação forçada: interrompe o processo em execução se aparecer um mais curto;

  - cooperativo: aguarda terminação do processo em execução mesmo que apareça um mais curto.

- não se consegue calcular ‘a priori’ o tempo de execução dos processos;

- apenas se podem fazer estimativas.

Preemptive Priority Scheduling:

- associa uma prioridade a cada processo;

- a ready queue é uma fila seriada por prioridades;

- escalona sempre o processo de maior prioridade;

- se um processo de maior aparecer, faz a troca de processos;

- Problema: starvation. Solução: envelhecimento – aumenta a prioridade dos processos pouco a pouco para que sejam executados eventualemente;

RR (round-robin):

- dá a cada processo um intervalo de tempo de CPU e volta a colocá-lo no fim da fila;

- quando um processo esgota o seu quantum retira-o do CPU e volta a colocá-lo no fim da fila;

- cada um dos processos terá 1/n do tempo disponível de CPU;

- se o quantum for muito grande, RR tende a comportar-se como FCFS;

- se o quantum for muito pequeno, então o overhead de mudanças do contexto degrada os níveis de utilização do CPU;

- tempo de resposta melhor que o SJF.

Avaliação dos algoritmos

	Modelo determinístico	Teoria da fila de espera	Simulação
Explicação	Definição da carga tipo: ordem de chegada dos processos, tempos de execução, etc.	Definição de um modelo matemático do sistema e avaliar segundo a teoria de filas de espera	Escrever/adoptar/ usar um programa que modele o sistema e analisar o desempenho do algoritmo
Vantagens	Simples		
Desvantagens	Ajuste dos	Muitas	Tempo de

	resultados depende directamente dos dados de entrada	simplificações para que o modelo seja tratável	execução longo

Sincronização de processos

- A possibilidade de execução “simultânea” leva ao acesso em concorrência a recursos partilhados;

- O acesso concorrente pode ser feito a zonas de endereçamento partilhadas;

- O acesso concorrente pode resultar na incoerência dos dados/resultados.

Seccção crítica-> em programação concorrente sempre que se testam ou se modificam estruturas de dados deve-se fazê-lo dentro de uma secção crítica.

Propriedades (secção crítica):

- exclusão mútua;

- progresso (liveness):

    - ausência de deadlocks;

    - ausência de míngua (starvation).

Condições (secção crítica):

- só um processo pode estar dentro da secção crítica;

- não se deve assumir valores quanto a velocidade de execução ou #CPU’s disponíveis;

- processos fora da secção crítica não deverão bloquear outros processos;

- nenhum processo deverá esperar indefinidamente para entrar na região crítica.

- um processo pode ser retirado de uma secção critica.

Mutex/Exclusão mútua-> técnica usada para evitar que dois processos ou threads tenham acesso simultâneo a um recurso partilhado (secção crítica).

Espera activa-> técnica que verifica que um processo verifica uma condição repetidamente até que ela seja verdadeira.

Os mutex não são suficientemente expressivos para resolver alguns problemas de sincronização.

Mutex é considerado um semáforo binário, ou seja, pode assumir o valor de 1 ou 0.

5. SOPE – Deadlocks

Deadlock-> cenário em que 2 ou mais processos ficam bloqueados indefinidamente.

O estudo do problema de deadlock tem origem em SO's, no contexto da gestão de recursos de um computador.

Recurso-> "objecto" usado por um processo, podendo haver mais do que um recurso do mesmo tipo.

Propriedades (recursos)

- tipo de acesso: certos recursos têm que ser acedidos em exclusão mútua;

- preemptibilidade: recursos podem ser ou não preemptíveis, isto é, podem ou não ser removidos dum processo sem problemas de maior.

Os recursos são geridos por gestores, os quais podem ser ou não componentes do SO.

Caso o recurso pedido por um processo não estiver disponível, o processo não o pode usar. Neste caso, o processo:

- ou bloqueia, à espera que o recurso fique disponível;

- ou prossegue, mas não terá acesso ao recurso pedido.

Um conjunto de processos está mutuamente bloqueado (deadlocked), se cada processo nesse conjunto está á espera dum evento que apenas outro processo nesse conjunto pode causar (normalmente, o evento pelo qual um processo espera é a libertação dum recurso na posso de outro processo nesse conjunto).

Condições necessárias para deadlock

- Bloqueio: à espera que o recurso fique disponível;

- Espera com retenção: processos não libertam os recursos na sua posse quando bloqueia;

- Não-preempção: recursos na posse dum processo não lhe podem ser retirados;

- Espera-circular: tem que haver uma cadeia de 2 ou mais processos, cada um dos quais à espera dum recurso na posso do processo seguinte.´

Grafos de alocação de recursos

- Processos: círculos;

- Recursos: quadrados;

- Um arco dum recurso para um processo: está em posse de;

- Um arco dum processo para um recurso: bloqueado à espera.

Algoritmos de resolução de deadlock

- Ignorar o problema: algoritmo da avestruz (Tanenbaum)

    - ignorar o problema;

    - pode ser aceitável se a probabilidade de deadlock for muito baixa e o custo das outras soluções for muito elevado;

    Unix tenta evitar situações de deadlock, mas não as elimina completamente.

- Detectar e recuperar

    - detectar o deadlock: o algoritmo testa se há alguma ordem de terminação dos processos;

    - recuperar da situação:

        - preempção de recursos: depende do recurso em causa (não é sempre aplicável e pode requerer intervenção humana);

    - rollback de processos:

        - fazer o checkpoint do estado dos processos, periodicamente;

        - em caso de deadlock, identificar um processo que se rolled-back quebrará o deadlock;

    - terminar um processo (de preferência um que possa ser reexecutado de inicio sem problemas de maior).

-Deadlock avoidance (Solução dinâmica):

    - Antes de satisfazer um pedido, averiguar se essa satisfação conduz a um estado não-seguro;

    - Faz uso de 4 estruturas de dados:

        - Vector dos recursos existentes: valor de cada elemento é o número de recursos desse tipo;

        - Vector de recursos disponíveis: depois de atribuídos, recursos restantes que sobraram;

- Matriz dos recursos atribuídos: cada linha representa um processo e cada coluna os recursos atribuídos de cada tipo;

- Matriz dos recursos a atribuir: recursos que ainda faltam atribuir a cada processo para ele executar.

-Prevenir deadlocks (Solução estática)