Estudo

Termos –

* A **concorrência** é obtida multiplexando as CPUs entre os processos/threads;
* **Overhead** é o tempo de custo adicional

Gerência de Memória do E/S:

* **Bufferização** é armazenar dados enquanto transferidos;
* **Caching** é armazenar partes dos dados mais rápido para aumentar desempenho;
* **Spooling** é sobreposição da saída de um processo com entrada de outros processos.

Processo:

* **Swapping** é mover toda ou uma parte de um processo da memória para o disco
* **UCP** é unidade de controle de processador
* **BCP** é bloco de controle de processo

Escalonamento de Processos

* **Throughput** é utilizar o máximo de processos executados;
* **Turnaround** é o tempo total para executar um processo;
* **Waiting time** é o tempo de espera na fila de prontos;
* **Response time** é o tempo entre requisição e primeira resposta.

Aula 1 (**Sistemas** **Operacionais**) –

* É um programa que age como intermediário entre o usuário e o hardware de um computador;
* O objetivo é executar os programas do usuário usando o hardware de maneira mais eficiente;
* É um alocador de recursos, gerencia e decide entre requisições conflitantes para uso justo dos recursos;
* É um programa de controle, controla a execução dos programas para prevenir erros;
* É ele quem gerencia os arquivos (criando, mapeando, entre outros).

Aula 3 (**Processos**) –

* É um programa individual em execução;
* Pode ser chamado de **job** ou **task**;
* Um processo **sempre** faz parte de uma fila;
* **Contexto** de um processo são todas as informações necessárias para que o S.O. possa restaurar a execução do processo a partir do ponto interrompido;
* O tempo de troca de contexto é puro **Overhead**;
* Deve-se **chavear** um processo (trocar a execução para outro) quando há falta de memória, Trap (erro), interrupção do relógio (expirou) ou interrupção de E/S.

Aula 4 (**Estrutura de Controle dos Processos**) –

* O **Scheduler** (Escalonador) é o responsável pelo controle do processador e divide o tempo da UCP entre os processos do sistema

**Escalonador**:

* Existem 3: curto, médio e longo prazo
* **Curto prazo** = ready -> running: Invocado muito frequentemente, portanto, precisa ser muito rápido;
* **Médio prazo** = swapping: Utiliza o swap out e copia os dados para o disco e utiliza o swap in para copiar de volta para a memória e retomar do ponto de onde parou. Intimamente ligado à gerência de memória;
* **Longo prazo** = new ready: baixa frequência de invocação, selecionando quais processos devem ser levados para a fila de prontos.

Tipos de Processos

* **CPU Bound\*:** Muito uso (podendo monopolizar) de CPU e pouca operação de E/S.
* **I/O Bound:** Orientado a I/O e devolve o controle da CPU.

\*Bound significa vinculado

Aula 5 (**Escalonamento de Processos**) –

Objetivos

* Maximizar a taxa de utilização da UCP
* Maximizar a vazão (throughput) do sistema (número de processos utilizados)
* Minimizar o tempo de execução (turnaround)
* Minimizar o tempo de espera (waiting time)
* Minimizar o tempo de resposta (response time)

Políticas

* **Preemptivas\***: Não permite a monopolização da UCP, podendo perder o processo de posse quando um mais prioritário está pronto, ou no fim da fatia de tempo.
* **Não**-**preemptivas**: Só perde a posse da UCP caso termine ou bloqueia a si mesmo (ex.: uma operação de E/S)

\*Preemptiva significa algo que pode ser antecipado

Exemplos de Algoritmos

**First-Come First-Served:**

* Processos que se tornam aptos para execução são inseridos no final da fila de prontos;
* Baixa complexidade e não preemptivo;
* Processos pequenos podem esperar muito tempo atrás dos longos (**convoy** **effect**);
* Favorece processos **CPU**-**bound**;
* Problemático para sistemas de tempo compartilhado;

Ex.: P1 (24s), P2(3s) e P3(3s), average waiting time = 17s.

**Shortest Job First:**

* Privilegia os processos pequenos para o tempo médio ser menor;
* Ótimo algoritmo;
* Seleciona-se o que possui menor tempo de CPU burst\*;
* Existe o não preemptivo e o preemptivo se chega um novo processo com tempo menor de CPU burst;
* A maior dificuldade é conhecer o tamanho da próxima requisição de CPU.

\*CPU burst é o tempo que um processo utiliza a CPU para executar as instruções

**Escalonamento por Prioridade**

* Número associado a cada processo, referindo a sua prioridade;
* CPU alocada ao processo de maior valor de prioridade (menor valor);
* Pode ocorrer **starvation** (morrer de fome) e nunca executar;
* A solução pode ser o **aging** (prioridade aumenta com o passar do tempo);
* Prioridades podem ser definidas interna ou externamente;
* Interna usa alguma medida para computar o valor, como limite de tempo, num. de arquivos abertos, razão entre average etc;
* Externa é definida por algum critério externo como o tipo do processo, custo, departamento responsável etc.

**Round-Robin**

* Cada processo recebe uma pequena fatia de tempo de CPU (10ms a 100ms);
* Após o término do tempo é interrompido e colocado no final da fila de prontos (preempção baseado no tempo);
* Costuma apresentar um tempo de **turnaround** médio maior que o SJF;
* Grande tende a First In First Out e pequeno tende a gerar muito overhead devido às trocas de contexto.

**Multinível**

* Divide os processos em diferentes grupos com diferentes requisitos de tempo de resposta
* Cada grupo é associado a uma fila de prioridade conforme importância
* Cada fila recebe uma quantidade de tempo de CPU diferente
* Entra na fila Q0, que é servida segundo a estratégia RR. Quando ele ganha a CPU ele recebe 8 ms. Se não terminar em 8 ms, é movido para a fila Q1. Em Q1 é novamente servido RR e recebe 16 ms adicionais. Se ainda não completar, ele é interrompido e movido para a fila Q2. Em Q2, FCFS.

Aula 6 (**Chamadas** **de** **Sistema**) –

* No Unix, há duas funções: fork() e exec();
* Fork cria um processo filho quase idêntico ao pai;
* Exec carrega e executa um novo programa;
* A sincronização é feita através de wait() que bloqueia o pai até que o filho termine.

**Fork()**

* Herda do pai alguns atributos como variáveis e prioridade de escalonamento;
* Retorna o PID (ID do processo) no pai e retorna 0 no filho;

**Exit()**

* Termina o processo;
* Envia um sinal para o pai do processo, se estiver bloqueado esperando o filho, é acordado;
* Se o processo que executou esta chamada tem filhos, serão “adotados” pelo init;
* Invoca o escalonador.

**Wait()**

* Os porocessos sabem que seu filho terminou quando o wait() retorna o PID do processo filho.
* Normalmente pai é bloqueado(wait()) até o filho terminar (exceto zombie)
* Waitpid() é para esperar um filho específico

**Processo “Zombie”**

* Sempre que um processo acaba, é executado o wait no pai
* Se um processo terminou e o pai ainda não usou wait, é um processo zumbi
* Se um pai acaba antes de um filho, são adotados pelo init

**Resumo**

* **fork():** cria um novo processo que é uma cópia do processo pai. O processo criador e o processo filho continuam em paralelo, e executam a instrução seguinte à chamada de sistema;
* **wait():** suspende a execução do processo corrente até que um filho termine. Se um filho terminou antes desta chamada de sistema (estado zombie), os recursos do filho são liberados e o processo não fica bloqueado, retornando imediatamente;
* **exit():** termina o processo corrente. Os filhos, se existirem, são herdados pelo processo init e o processo pai é sinalizado;
* **exec():** executa um programa, substituindo a imagem do processo corrente pela imagem de um novo processo, identificado pelo nome de um arquivo executável, passado como argumento;
* **kill():** usada para enviar um sinal para um processo ou grupo de processos. O sinal pode indicar a morte do processo;
* **sleep():** suspende o processo pelo tempo especificado como argumento.

Aula 7 (**Threads**) –

* Thread significa “fluxo” ou “fio”, é um fluxo de execução dentro de um processo;
* Permite que uma aplicação execute mais de um trecho de código simultaneamente;
* Continua respondendo mesmo se uma parte está bloqueada;
* Mutithreading é a capacidade do S.O suportar múltiplas threads concorrentes em um processo;
* A criação e comunicação entre threads é mais rápida que entre processos;

Aula 8 (**Sincronização de Threads e Processos**)

* Condições de corrida são situações em que dois ou mais processos ou threads acessam dados compartilhados e o resultado depende da ordem em que são executados;

**Possíveis Soluções**

* **Inibição de Interrupções (DI/EI)**: faz o processo desativar interrupções antes de entrar na Região Crítica. Problema: Não funciona com vários processadores e não é aconselhável;
* **Busy Wait:** quando quiser entrar na Região Crítica verifica se entrada é permitida, se não for, espera até ser liberado. Problema: desperdício de tempo de CPU (Variável de bloqueio não funciona, 2 processos podem pensar que está liberado simultaneamente);
* **Dekker:** funciona se o número de processos é menor ou igual ao número de CPUs, ou seja, pouco usada;
* **Lamport:** funciona, solução para vários processos do algoritmo de Dekker e é utilizado para compartilhamento de um recurso em uma seção crítica;
* **Peterson:** funciona, marcando sua intenção de entrar, já indica que a vez é do outro;
* **Semáforos:** funciona, mas a operação é lenta.

Aula 11 (**Comunicação entre Processos**) –

* Processos são independentes, mas interagem com outros na execução de tarefas (ex.: dividir tarefas para maior velocidade);
* Inter-Process Communication é um conjunto de mecanismos de trocas de informação de múltiplas threads de um ou mais processos com sincronização;
* Responsável por isso é o Sistema Operacional;
* Comunicação via memória compartilhada (mais rápido, mas pode ter problemas de sincronização);
* Comunicação via núcleo (pode ser feito com várias CPUs, porém, mais complexo e demorado);
* Pipes (tubos) constituem o mecanismo original de comunicação entre processos (I/O);
* É anônimo, temporário (tempo de execução do processo) e limitado;
* Para ter uma comunicação entre processos precisa de dois Pipes;
* who | sort | lpr = output do who é input do sort, output do sort é input do lpr

(Processo who escreve no pipe1, processo sort lê do pipe1 e grava no pipe2 e processo lpr lê do pipe2)

* Constituem um canal de comunicação entre processos pai-filho
* São definidos antes da criação dos filhos, ligam apenas com antepassado comum
* Um pipe criado em um único processo é quase sem utilidade. Normalmente, depois do pipe, o processo chama fork(), criando um canal e comunicação entre pai e filho.
* Quando um processo faz um fork() depois de criado o pipe, o processo filho recebe os mesmos descritores de leitura e escrita do pai.
* Processo que pretende ler de um pipe vazio fica bloqueado até que um processo escreva os dados.
* Filas (FIFOs – First In First Out), também designados de “tubos nomeados” (“named pipes”), permitem a comunicação entre processos não relacionados.
* Referenciadas por um id dentro do sistema de arquivos
* É bloqueado caso tente abrir uma fila em modo leitura sem em uma fila em modo de acesso de escrita e vice-versa.