IV. Sistemas operacionais: Processo e threads

IV.1 Conceitos de processo

- Um sistema operacional executa diversos tipos de programas
  - Sistema *batch* - *jobs*
  - Sistemas de tempo compartilhado - programas de usuários ou tarefas
- Os livros texto usam os termos *job* e processos quase como equivalentes.
- Processo - um programa em execução; a execução de um processo deve ocorrer de maneira sequencial.
- Um processo inclui:
  - contador de programa
  - pilha
  - seção de dados
  - texto do programa

Estados de um processo

- Quando um processo executa, ele troca de estados:
  - **novo**: O processo está sendo criado.
  - **rodando**: As instruções estão sendo executadas.
  - **espera**: O processo está esperando por algum evento.
  - **pronto**: O processo está esperando um processador para voltar rodar.
  - **terminado**: O processo terminou a sua execução.

Diagrama de estados de um processo.

Bloco de Controle de um Processo (PCB)

Estrutura de dados com as informações associadas a um processo:

- Estado do processo
- Contador de programa
- Registradores da CPU
- Informação de escalonamento da CPU
- Informação para o gerenciamento da memória
- Informação de contabilidade
- Informação do *status* das E/S

PCB - Process Control Block.

Chaveamento de CPU do processo P_0 para o processo P_1.

Escalonamento de processos

Filas de Processos

- Fila de *jobs* - conjunto de todos os processos no sistema
- *Fila de prontos* - conjunto de processos residentes na memória que estão prontos para executar e estão esperando um CPU
- *Filas de dispositivos* - conjunto de processos esperando por um dispositivo de E/S
- Migração dos processos entre as várias filas.

Fila de processos prontos e filas de E/S.

Escalonamento de CPU.

Escalonadores

- Escalonador de longo prazo (ou escalonador de job) - seleciona qual processo deve ser trazido para a fila de prontos.
- Escalonador de curto prazo (ou escalonador de CPU) - seleciona qual processo deve ser executado a seguir e aloca a CPU.

Adição de um escalonador de prazo médio.

- Escalonador de curto prazo é acionado freqüentemente (milisegundos) => deve ser rápido

- Escalonador de longo prazo é menos acionado (segundos, minutos) => pode ser lento
- O escalonador de longo prazo controla o grau de multiprogramação
- Os processos podem ser caracterizados como:
    - *processos limitados por E/S* - gastam a maior parte do tempo fazendo entradas e saídas, usa a CPU em diversos intervalos de tempo curtos.
    - *processos limitados pela CPU* - passa a maior parte do tempo processando; poucos intervalos longos de tempo com a CPU.

Operações em processos

Comutação (troca) de contexto

- Quando a CPU troca de processo, o sistema deve salvar o estado do processo anterior e carregar o estado salvado do novo processo.
- A comutação de contexto representa um tempo de processamento extra; o sistema não está realizando trabalho útil numa comutação de contexto.
- O tempo gasto depende do suporte do *hardware*.

Criação de processo

- O processo pai cria processos filhos, que, por sua vez, criam outros processos, formando uma árvore de processos.
- Compartilhamento de recursos:
    - Os processos pai e filhos compartilham todos os recursos.
    - Os processos filhos compartilham um sub-conjunto dos recursos.
    - Os processos pai e filho não compartilham recursos.
- Execução:
    - O pai e o filho executam concorrentemente
    - O pai espera a execução do filho terminar.
- Espaço de endereços:
    - O filho tem uma duplicata do pai
    - O filho tem um outro programa carregado nele
- Exemplos do UNIX:
    - a chamada sistema *fork* cria um novo processo
    - a chamada sistema *execve* usada após um *fork* substitui o processo no espaço de memória por um novo programa.

Término de Processo

- O processo executa a sua última instrução e pede ao sistema operacional para sair (**exit**)
    - Passa dados do filho para o pai (via **wait**)
    - Os recursos do processo são desalocados pelo sistema operacional.
- O pai pode terminar a execução dos processos filhos (**abort**).
    - O filho ultrapassou a quantidade de recursos alocada.

- A tarefa dada ao filho não é mais necessária.
- O processo pai terminou:
  - O sistema operacional não permite que o filho continue se o pai terminou.
  - Término em cascata.

Exemplo de programa C em UNIX com criação de processo pai que espera processo filho terminar.

```
/**
 * newproc.c
 *
 * This program creates a separate child process.
 * The child process performs the "ls" command to
 * output a directory listing.
 *
 * @author Greg Gagne, Peter Galvin, Avi Silberschatz
 * @version 1.0 - January 1, 2000
 *
 * Copyright 2000 by Greg Gagne, Peter Galvin, Avi Silberschatz
 * Applied Operating Systems Concepts - John Wiley and Sons, Inc.
 */

#include

main(int argc, char *argv[])
{
int pid;

        /* fork another process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);

                printf("Child Complete\n");
                exit(0);
        }
}
```

Cooperação entre processos

- Processos independentes não podem afetar ou ser afetados pela execução dos outros processos.

- Processos *cooperativos* podem ser afetados ou afetar um ao outro,
- Vantagens de processos cooperativos:
    - compartilhamento de informações
    - aceleração do processamento
    - modularidade
    - conveniência

Problema do produtor-consumidor

- paradigma para processos cooperativos, o *produtor* produz informação que será consumida pelo processo *consumidor*.
    - *buffer ilimitado*, não existe limitação prática sobre o tamanho do *buffer*
    - *buffer limitado*, assume que existe uma limitação para o tamanho do *buffer*.

Threads

- Um *thread* (ou processo leve) é uma unidade básica de utilização de CPU; ele consiste de:
    - contador de programa
    - conjunto de registradores
    - espaço de pilha
- Um *thread* compartilha com os *threads*-pares:
    - a seção de código
    - a seção de dados
    - os recursos do sistema operacional

  Coletivamente chamados de tarefa.

- Um processo tradicional (ou de peso) é igualmente uma tarefa com um único *thread*.
- Numa tarefa *multithread*, enquanto um *thread* servidor está bloqueado e esperando, um segundo *thread* na mesma tarefa pode está rodando.
    - A cooperação de vários *threads* num mesmo *job* aumenta o *throughput* e melhora o desempenho.
    - Aplicações que requerem o compartilhamento de um *buffer* (p.ex., produtor-consumidor) se beneficiam da utilização de *threads*.
- *Threads* fornecem um mecanismo que permite aos processos seqüenciais que fazem chamadas de sistema bloqueadoras conseguirem também paralelismo.
- Os *threads* podem ter suporte do kernel (Mach e OS/2)
- *Threads* no nível de usuário; suporte acima do nível de kernel, através de uma biblioteca de rotinas do nível do usuário (projeto Andrew da CMU)
- *Approach* híbrido (solaris 2)

Comunicação entre processos (ICP)

Duas maneiras de implementar a comunicação entre processos:

1. Usando mensagens: o Processo A envia uma mensagem para o processo B através do sistema operacional.
2. Usando memória compartilhada: os processos A e B compartilham um espaço de memória, o processo A escreve nele e o processo lê dele.

- Mecanismo para os processos se comunicarem e para sincronizar as suas ações.
- Sistema de mensagens - os processos se comunicam entre eles sem o uso de variáveis compartilhadas.
- O serviço de IPC fornece duas operações:
    - **send**(mensagem) - envia mensagem de tamanho fixo ou variável.
    - **receive**(mensagem) - recebe mensagem.
- Se *P* e *Q* querem se comunicar, eles precisam:
    1. estabelecer um enlace de comunicação entre eles
    2. trocar mensagens com **send**/**receive**
- Implementação do enlace de comunicação:
    - físico (p.ex., memória compartilhada, barramento de *hardware*)
    - lógico (p.ex., propriedades lógicas)

Questões de Implementação

- Quantos enlaces estabelecer?
- Pode um enlace ser associado a mais de dois processos?
- Quantos enlaces podem existir entre cada par de processos ccoperativos?
- Qual é a capacidade de um enlace?
- O tamanho de uma mensagem deve ser fixo ou variável?
- Um enlace é unidirecional ou bidirecional?

Comunicação direta

- Os processos precisam se endereçar explicitamente:
    - **send**(*P*, *mensagem*) - envia uma mensagem ao processo P
    - **receive**(*Q*, *mensagem*) - recebe uma mensagem do processo Q
- Propriedades do enlace de comunicação:
    - Os enlaces são estabelecidos automaticamente.
    - Um enlace é associado a exatamente um par de processos comunicantes.
    - Entre cada par existe exatamente um enlace.
    - O enlace pode ser unidirecional, mas geralmente é bidirecional.

Comunicação Indireta

- As mensagens são direcionadas e recebidas em caixas postais (também chamadas de portas).
    - Cada caixa postal tem um identificador, id, único.

    - Os processos podem se comunicar apenas se eles compartilharem uma caixa postal comum.
- Propriedades do enlace d comunicação:
    - O enlace só é estabelecido se houver uma caixa postal comum compartilhada.
    - Um enlace pode estar associado com muitos processos.
    - Cada par de processos pode compartilhar diversos enlaces de comunicação.
    - Os enlaces podem ser uni-direcionais ou bi-direcionais.
- Operações:
    1. criar uma nova caixa postal
    2. enviar ou receber mensagens através da caixa postal
    3. destruir uma caixa postal
- Compartilhamento de caixa postal:
    - P_1, P_2 e P_3 compartilham a caixa postal **A**.
    - P_1 envia; P_2 e P_3 esperam receber.
    - Quem recebe?
- Soluções:
    - Permitir que um enlace seja associado a no máximo dois processos.
    - Permitir apenas um processo de cada vez na espera de recepção de uma caixa postal.
    - Permitir ao sistema escolher arbitrariamente o receptor. Quem enviou recebe um aviso de quem foi que recebeu.

Bufferização

- Fila de mensagens ligadas a um enlace; implementadas por uma das três opções:
    1. Capacidade zero - 0 mensagens
       O emissor deve esperar pelo receptor (*rendezvous*).
    2. Capacidade limitada - comprimento finito de *n* mensagens
       O emissor deve esperar se o enlace estiver cheio.
    3. Capacidade ilimitada - comprimento infinito
       O emissor nunca espera.

Condições de exceção - Recuperação de erros

- O processo termina
- Mensagens são perdidas
- As mensagens estão embaralhadas

Benefícios de threads

- Rapidez de resposta
- Compartilhamento de recursos
- Economia
- Utilização de arquiteturas multiprocessadores

Processos com um *thread* ou *multithreads*

Threads de usuários e de *kernel*

Threads de usuário

- A gerência dos *threads* é feita por uma biblioteca de *threads* no nível usuário.
- Exemplos:
  - POSIX *Pthreads*
  - MACH *C-threads*
  - Solaris *threads*

Threads de *kernel*

- Têm suporte do *kernel*
- Exemplos:
  - Windows 95/98/NT/XP
  - Solaris
  - UNIX da Digital

Modelos de implementação de *Multithreads*

Muitos-para-um

- Muitos *threads* do nível usuário são mapeados num único *thread* no kernel.
- Usado em sistemas que não têm suporte para *threads* no *kernel*.

Muitos *threads* usuários usando um único *thread* de kernel.

Um-para-um

- Cada *thread* usuário é mapeado num *thread* de *kernel*
- Exemplos:
  - Windows 95/98/NT/XP
  - OS/2

Cada *thread* usuário usa um *thread* de *kernel*.

Muitos-para-muitos

Diversos *threads* usuários usam diversos *threads* de kernel.

Threads em Java

- Os *threads* de Java podem ser criados:
  - Extendendo a classe **Thread**
  - Implementando a interface **Runnable**

Extendendo a classe *Thread*

```
Class Worker1 extends Thread {
    public void run() {
        System.out.println("Eu sou um Worker Thread.");
    }
}

public class First {
    public static void main(String args[]) {
        Worker runner=new Worker1();
        runner.start();
        System.out.println("Eu sou o thread principal.");
    }
}
```

Implementando a interface *Runnable*

```
class Worker2 implements Runnable {
    public void run() {
        System.out.println("Eu sou outro Worker Thread.");
    }
}

public class Second {
    public static void main(String args[]) {
        Runnable runner=new Worker2();
        Thread thrd=new Thread(runner);    // necessário
        thrd.start();                      // Worker2 não tem start()
        System.out.println("Eu sou o segundo thread principal.");
    }
}
```

Gerência de *threads* no Java

suspend()
suspende a execução do *thread* que está sendo executado.

sleep()
põe o *thread* para *dormir* por um tempo especificado.

resume()
continua a execução de um *thread* suspenso.

stop()

termina a execução de um *thread*.

Diagrama de estados do *Threads* no Java.

Problema Produtor Consumido

```
public class Server {
    public Server() {
        MessageQueue mailBox = new MessageQueue();
        Producer producerThread = new Producer(mailBox);
        Consumer consumerThread = new Consumer(mailBox);

        producerThread.start();
        consumerThread.start();
    }
    public static void main(String args[])   {
        Server server = new Server();
    }
}

class Producer extends Thread {
    public Producer(MessageQueue m) {
        mbox = m;
    }
    public void run() {
        while (true) {
          // produce an item & enter it into the buffer
           Date message = new Date();
           mbox.send(message);
        }
    }
    private  MessageQueue mbox;
}

class Consumer extends Thread {
    public Consumer(MessageQueue m) {
        mbox = m;
    }
    public void run() {
        while (true) {
            Date message = (Date)mbox.receive();
            if (message != null)
                // consume the message
        }
    }
    private  MessageQueue mbox;
}
```