Processos

Prova vai até comunicação interprocessos, escalonamento não cai.

Processo é um programa em execução.

Programa Carregado Código (estático) : está na memória principal

em -> na -> + -> Registradores (CS)

Execução Memória Dados (dinâmico) - Podem estar nos:

1. registradores,

2. memória principal (Reg DS),

3. arquivos: - arquivos abertos no programa, - posição corrente

(do mais perto, para o mais longe do processador)

+

Pilha (registradores: SG, SP, BP)

+

Informações de estado (estado de execução)

Tudo isso caracteriza a execução do programa, o PROCESSO.

Quantos processos consigo executar de cada vez? Se quero executar mais processos, do que a máquina permite? Opções:

- Chaveamento de processos ao longo do tempo (pausar 1 e iniciar o outro, cada um executa um pouco);

- Fila de processos;

- Descarta processos excedentes;

Escalonamento:

D…………………….\_\_\_\_\_\_\_\_\_\_………………………………..

C……………...\_\_\_\_……………...\_\_\_\_………………………….

B………………………………………….\_\_\_\_…………………...

A…….. \_\_\_…………………………………….. \_\_\_…………..….

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Tempo

Chamada para criação de processos:

1. Início do sistema;

2. Execução de chamada ao sistema de criação de processos

fork (UNIX), CreateProcess (Windows), SYS$CREPRC (VAX/VMS);

3. Solicitação do usuário para criar um novo processo;

4. Início de um job em lote.

Tipos de processos:

- Processos interativos: interagem com usuários: primeiro plano (foreground)

- Processos de segundo plano (background): serviços do sistema (daemons)

No UNIX processos são criados através da chamada FORK.

Visão conceitual de FORK:

Memória

|  |
| --- |
|  |
| f=fork()  //processo 21 Pai  f==84 |
|  |
| f=fork()  //processo 84 Filho  f==0 |
|  |

1. Processo pai é copiado para outra região na memória

2. Execução prossegue (No Pai E no filho) a partir da instrução seguinte ao fork()

Exemplo Fork do Moodle:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 24 | pai: n=7 | 19 | filho: n=7 | 19 | filho: n=7 |
| 19 | filho: n=7 | 24 | pai: n=7 | 21 | filho: n=10 |
| 21 | filho: n=10 | 21 | filho: n=10 | 24 | pai: n=7 |
| 27 | pai: n=10 | 27 | pai: n=7 | 27 | pai: n=7 |

Precedência: 14 -> 16 ->17 ->23 ->24 --------------->26 ->27 (pai)

-> 17 ->15 ->20 ->21 ->22 ^ (filho)

Término do processo:

- saída normal (voluntária)

- saída por erro (voluntária)

- programa detecta um erro

- erro fatal (involuntário)

- programa faz algo ilegal

- cancelamento por outro processo (involuntário)

O término de um processo pode causar o término dos processos que ele criou

- não ocorre nem em UNIX nem em Windows

Hierarquia:

- Processos procriam por várias gerações

- um processo pai cria processos lhos, que por sua vez também criam seus lhos, ad nauseam

- Leva à formação de hierarquias de processos

- Chamadas grupos de processos no UNIX

- sinalizações de eventos se propagam através do grupo, e cada processo decide o que fazer com o sinal (ignorar, tratar ou ser morto)

- todos os processos UNIX descendem de init systemd em várias distribuições Linux

- Windows não possui hierarquias de processos todos os processos são criados iguais

Estados:

- em execução: processo que está usando a CPU

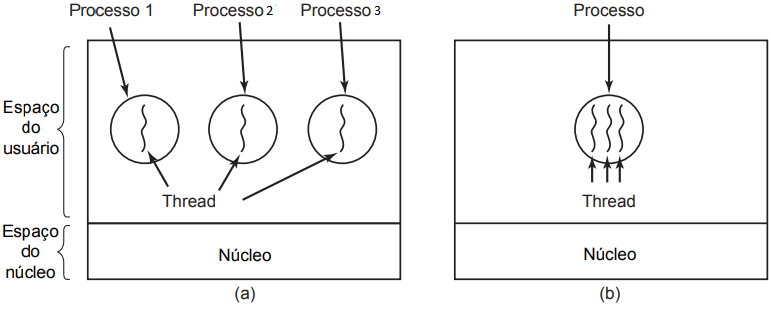
- pronto: processo temporariamente parado enquanto outro processo executa

- fila de prontos (aptos)

- bloqueado: esperando por um evento externo

Threads:

Vários processos podem ter apenas 1 fluxo de execução, 1 thread. Ou 1 processo pode ter várias threads.



Uma pilha para cada thread.

Diferença de estado de processo e thread: dependendo da implementação, o bloqueio de uma das threads de um processo pode bloquear todas as demais.

Funções Essenciais Pthread

- Criação: pthread\_create()

- Encerramento: pthread\_exit()

- Sincronização(espera pelo término): pthread\_join()

Casting: - Conversão de/para outro tipo de ponteiro (ex. char \*)

- Conversão de/para long -> sizeof(long) == sizeof(void\*)

Exercícios

-1. Compilar e executar o programa simples.c, disponível nos exemplos:

$ gcc-Wal -o simples simples.c -pthread

0. Modificar o programa simples.c para criar, além das threads que executam PrintHello(), NUM\_THREADS+3 THREADS que imprimem o quadrado do número recebido como parâmetro. (Portanto, no total o programa deve criar 2\*NUM\_THREADS+3THREADS)

Inanição de Processos: algum processo vai ser prejudicado.

IPC: Condição de disputa: Isso tem a ver com código concorrente que manipula o estado compartilhado: Chamamos de Região Crítica: A solução para resolver isso é: Exclusão mútua. Algumas soluções: Mecanismo (algoritmos, técnicas): com espera ocupada, e sem espera ocupada. A melhor opção é usar chamadas bloqueantes.

Sleep e Wakeup //Sem espera ocupada.

Problema do produtor-consumidor

Produtor -> Coloca no buffer o que produz

|  |  |
| --- | --- |
| buf | fer |

-> Consumidor

Estado compartilhado => Buffer

produtor () {

while (true) {

item=produz\_item();

insere(item); //no buffer. RC-região crítica

}

}

consumidor (){

while(true){

item=retira\_item(); //região crítica

consome(item);

}

}

Exclusão mútua entre as regiões críticas.

1 item=produce\_item();

2 if (count == N) sleep();

3 insert\_item(item);

4 count=count+1;

5 if (count==1) wakeup(consumer);

}

}

void consumer(void)

{

int item;

while(TRUE){

6 if (count == 0) sleep();

7 item=remove\_item();

8 count=count− 1;

9 if(count==N− 1) wakeup(producer);

10 consume\_item(item);

execução: 6, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 7, 8, 9, 10, 3, 4, 5, 1, 6, 7, 8, 9, 10, 6, 2, 3, 7

2a if (count==N) //verifica uma condição

2b sleep (); //age (bloqueia) de acordo com a condição

//essas condições são dois passos separados.

6a if (count==0)

6b sleep ();

execução: 1, 6a, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 6b

Outra primitiva: Semáforos

Conta quantos wakeups o processo sofreu.

down(S): decrementa S; se S<0, bloqueia

up(S): incrementa S; se S≤0, acorda um processo que está esperando por S

Tipos de Semáforos:

Semáforo binário: Serve para garantir exclusão mútua

SB=1

down(SB)

RC

up(SB)

Isso é o que torna uma variável binária.

Sincronização: Sinalização de condição.

Quem espera: down()

Quem sinaliza: up()

Semáforo full e empty, são contadores.

Problemas de IPC:

- Exclusão mútua (regiões críticas que manipulam estado compartilhado)

- Sincronização (sequência de execução, sinalizar condições)

Para resolver Exclusão mútua, usamos semáforos binários: sem=1

down(sem)

RC

up (sem)

Essa é a lógica/Protocolo de Acesso.

Para sincronização, usamos Semáforos Contadores: Quem espera a condição, vai ter que fazer um down, e quem sinaliza, faz um up.

Espera (cond)

down(sem)

Sinaliza (cond)

up(sem)

Monitores

Por construção

Variáveis de Condição

wait()

signal ()

Solução com Sleep e Wakeup:

if (count==N) //verifica a condição

sleep (); // age com base na condição

Com monitor:

if count=N then //verifica condição

wait (full); //age com base na condição

Monitores em Java

do {

this.wait();

} while(!condicao1);

condicao1=true;

this.notifyall();

do {

this.wait();

} while (!condicao2);

Semáforo Binário

semaphore SB=1;

down(SB);

RC

up(SB);

Fazer a mesma coisa, mas com mutex:

MUTEX:

pthread\_mutex\_t SB=PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_lock(&SB);

RC

phtread\_mutex\_unlock(&SB);

pthread\_cond\_t cond y=PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mtx=PTHREAD\_MUTEX\_INITIALIZER;

void x() {

faz algo();

pthread\_mutex\_lock(&mutex);

pthread\_cond\_signal(&cond y);

phtread\_mutex\_unlock(&mutex);

faz\_outra\_coisa();

}

void y() {

faz\_uma\_coisa();

pthread\_mutex\_lock(&mtx);

pthread\_cond\_wait(&cond y, &mtx);

pthread\_mutex\_unlock(&mtx);

faz\_outra\_coisa();

}