GSI018 – Sistemas Operacionais

Sincronização e Comunicação de/entre Processos

Universidade Federal de Uberlândia
Faculdade de Computação
Prof. Dr. rer. nat. Daniel D. Abdala

Na Aula Anterior...

- Ciclos de CPU e E/S;
- Conceitos de Preempção;
- Algoritmos de Escalonamento;
- FCFS – First Come First Served;
- SJF – Shortest Job First;
- Round-Robin;
- Escalonamento por Prioridade;
- Filas Multinível.

Nesta Aula

- Comunicação entre Processos
  - Pipes;
  - Memória compartilhada;
  - Sockets;
- Seção Crítica;
- Suporte em Hardware para Sincronismo;
- Semáforos;
- Mutexes;
- Barreiras;

Comunicação entre Processos

- Enviar e receber dados e mensagens entre processos;
- Compartilhar dados;
- Os mecanismos dependem do sistema operacional;
- Principais formas de Comunicação:
  - Pipes;
  - Memória Compartilhada;
  - Sockets;


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>

int main()
{
    pid_t child_pid;
    char s[10];
    int c;

    child_pid = fork();

    if ( child_pid != 0 ) {
        printf( "este é o processo atual cujo id eh: %d\n", (int) getpid() );
        for (c = 1; c < 100000; c++)
        {
            if (c % 5000 == 0)
                printf("Pai - %d\n", c);
        }

    } else {
        printf( "este eh o processo rescem criado, cujo id eh: %d\n", (int)
        getpid() );
        for (c = 1; c < 100000; c++)
        {
            printf("Filho - %d\n", c);
        }
    }
    exit( 0 );
}
```


Pipes

- Fluxo de dados de um processo para outro;
- Método bidirecional de comunicação;
- Diferente do pipe do terminal;
- Projetado para ser utilizado entre processos bifurcados;

```
>> ls -al | more
```

PIPE: Chamada do Sistema

```
CHAMADA DO SISTEMA: pipe();
BIBLIOTECA: #inclide <unistd.h>

  PROTÓTIPO: int pipe( int fd[2] );
    RETORNO: 0 on success
             -1 on error: errno = EMFILE (não há descritores livres)
                                  EMFILE (tabela de arq. do sistema cheia)
                                  EFAULT (fd não é válido)

  NOTAS: fd[0] leitura, fd[1] escrita
```

7

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
            strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
exit(EXIT_SUCCESS);
}
```

8

Memória Compartilhada

- Porção de Memória compartilhada entre dois ou mais processos;

```
#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

9

Memória Compartilhada

- **shmget** – shared memory get
  - Tal como malloc; no entanto a memória pode ser compartilhada;
  - Shmflg são flags (9 bits) tal como no sistema de arquivos;
- **shmat** – shared memory attach
  - Associa a memória compartilhada ao espaço de endereçamento de um processo;
- **shmdt** – shared memory detach
  - Remove a memória compartilhada do espaço de endereçamento de um processo;
- **shmctl** – shared memory control functions

10

Sockets

- Funciona como pipes mas entre processos em computadores distintos;
- Usado para criar aplicações cliente/servidor;

11

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    int sockfd;
    int len;
    struct sockaddr_un address;
    int result;
    char ch = 'A';

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "server_socket");
    len = sizeof(address);

    result = connect(sockfd, (struct sockaddr *)&address, len);
    if(result == -1) {
        perror("oops: client1");
        exit(1);
    }

    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

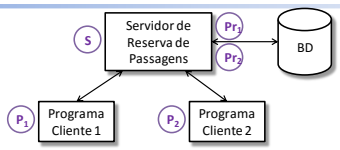
12

Sincronização de Processos

- Devido a multiprogramação e comunicação remota, situações inusitadas podem ocorrer:
  - Recursos compartilhados (memória, arquivos, etc) podem assumir um estado instável e ficar inconsistentes;
- Três principais tópicos de interesse
  - Mecanismos para troca de informações entre processos;
  - Mecanismos para garantir que recursos compartilhados por mais de um processo não sejam utilizados de maneira conflitante;
  - Mecanismos para sincronizar atividades entre processos.
- **Exclusão Mútua** → Forma de garantir que um recurso não seja utilizado por outro processo uma vez que ele tenha começado a ser utilizado por um processo;

13

O Problema da Condição de Corrida



1. P_1, P_2 e S são programas rodando em computadores distintos;
2. P_1 envia uma consulta para S sobre uma passagem de GRU→ROM dia 30/10/2016;
3. Logo a seguir P_2 envia a mesma consulta a S;
4. Ambos P_1 e P_2 requerem o mesmo assento A3;

14

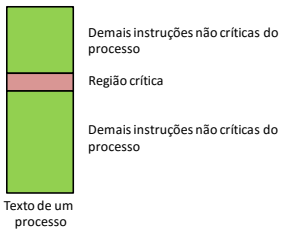
O Problema da Condição de Corrida

5. Considere que o servidor S lança um novo processo Pr_i para tratar cada requisição enviada para cada cliente P_i;
6. Ambos Pr_1 e Pr_2 querem reservar o assento A3;
7. Digamos que Pr_1 receba a CPU de S primeiro. Ele registrará o assento para o cliente de P_1;
8. No meio da execução Pr_2 ganha a CPU. A reserva não foi finalizada por Pr_1. Logo ele reserva o cliente para P_2;
9. Há um chaveamento de volta para Pr_1 e então ele finaliza a reserva, sobrescrevendo a reserva que acabou de ser feita por Pr_2.

15

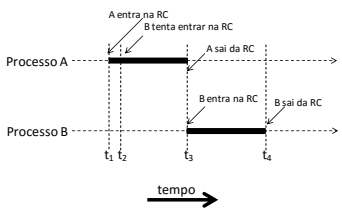
O Conceito de Região Crítica

- Região Crítica (RC) → parte do programa que acessa recursos que podem levar a condições de disputa;



16

Comportamento de Regiões Críticas



17

Condições Para Regiões Críticas

① Dois Processos nunca podem estar simultaneamente em suas regiões críticas;
② Nada pode ser afirmado sobre a velocidade ou o número de CPUs;
③ Nenhum processo executando fora de sua região crítica pode bloquear outros processos;
④ Nenhum processo deve esperar indefinidamente para entrar em sua região crítica;

18

Desabilitando Interrupções

- Quando um processo entra na região crítica a primeira coisa que ele faz é desabilitar todas as interrupções do sistema;
- Problemas:
  - Usuários devem possuir a abilidade de desabilitar interrupções de todo o sistema;
  - Em sistemas com múltiplos cores, o que ocorre numa CPU não afeta as demais;

19

Variáveis do Tipo Trava (Locks)

- Variável booleana compartilhada associada a um recurso compartilhado;
  - Se lock == 0 → recurso não utilizado;
    - Altera para 1
    - Faz o serviço
    - Retorna para 0
  - Se lock == 1 → fica testando até que lock seja 0
- O que acontece caso o processo chaveie entre testar se lock == 0 e setar lock para 1?

20

A Instrução TSL

- TSL → Test and Lock
  - `TSL RX,LOCK`
- Operação atômica, ou seja resolve o problema dos dois passos do lock;

```
enter_region:
    TSL REGISTER,LOCK     | copia lock para o registrador e põe lock em 1
    CMP REGISTER,#0       | lock valia zero?
    JNE enter_region      | se fosse diferente de zero, lock estaria ligado, portanto,
    RET                   | retorna a quem chamou; entrou na região crítica

leave_region:
    MOVE LOCK,#0          | coloque 0 em lock
    RET                   | retorna a quem chamou
```

21

Problema: Produtor vs Consumidor

- Dois processos estão em comunicação;
- Um produz dados e os coloca em um buffer para que o segundo os leia e utilize;
- O problema ocorre quando o produtor quer colocar um novo dado no buffer mas este já se encontra cheio;
- Similarmente, outro problema ocorre quando o consumidor quer ler um dado do buffer e ele encontra-se vazio;

22

Semáforos

- Criado por Dijkstra em 1965 para resolver o problema do produtor-consumidor;
- Uma variável inteira;
- Duas rotinas:
  - Down (sleep) – `sleep(Semaforo)`
    - testa se o valor de semaforo é maior que zero, se sim decrementa-o em uma unidade;
    - Se o valor for zero, processo é bloqueado sem retornar da função sleep;
  - UP (wakeup) – `wakeup(Semaforo)`
    - Incrementa o valor do semáforo
    - Se o valor atual é zero, testa se há processos dormindo nele, e em caso afirmativo escolhe um deles para acordar. Neste caso o valor do semáforo continua zero

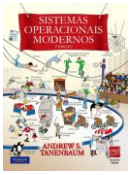
23

Mutexes

- Versão simplificada de semáforos;
- Mutex – abreviação de mutual exclusion;
- Dois estados:
  - Desimpedido
  - Impedido
- Duas rotinas:
  - `mutex_lock(m1)`
  - `mutex_unlock(m1)`

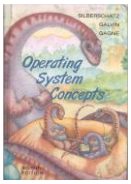
24

Bibliografia - Básica



- 3ª Edição
- Páginas 70-87

Bibliografia - Básica



- 7ª edição
- Páginas 96-102

Bibliografia - Adicional

- Capítulos 13, 14 e 15