Trabalho 02 - Comunicação Sincronismos, Deadlock, Threads

SSC0640- Sistemas Operacionais I

Prof. Assoc. Júlio Cezar Estrella

André Baconcelo Prado Furlanetti - Nº USP: 10748305
Diego da Silva Parra - Nº USP: 10716550
Mateus Fernandes Doimo - Nº USP: 10691971

Acesso ao GitHub

https://github.com/andrebpradof/sistemas-operacionais

Acesso ao YouTube

Produtor e Consumidor - Trabalho 2 de Sistemas Operacionais - #GRUPO-8

Comunicação entre processos

Precisamos compreender as seguintes questões:

- como um processo passa informações para outro;
- como garantir que dois ou mais processos não se atrapalhem;
- qual o sequenciamento adequado quando temos dependências entre os processos.

Comunicação entre processos

Quando há compartilhamento de recursos entre os processos (em grande parte dos sistemas operacionais) precisamos nos atentar às *race conditions* (falhas no acesso simultâneo a um mesmo recurso).

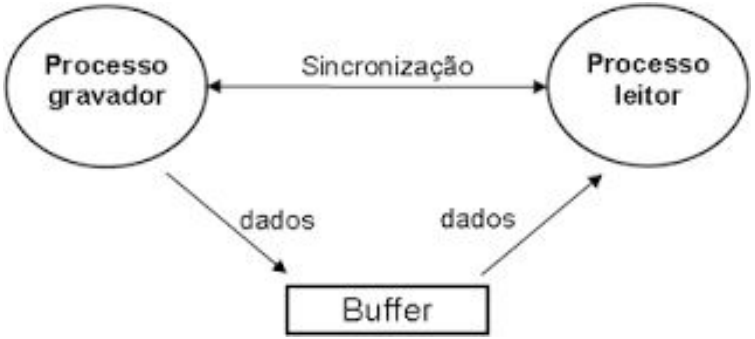
Sincronia de processos

Permite o gerenciamento do acesso concorrente aos recursos do SO.

O acesso é controlado por parte dos processos, de maneira que um recurso não é modificado em simultâneo e que os processos não fiquem em espera.

Sincronia de processos

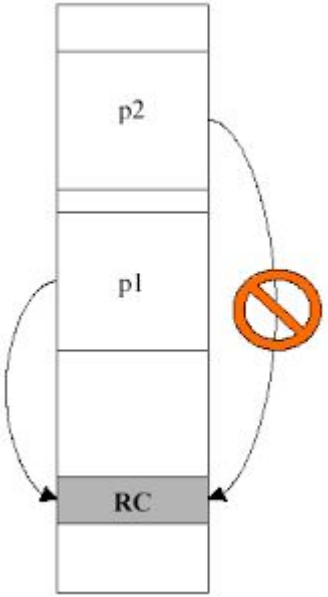
Exemplo: processos de um computador compartilham variáveis globais, instruções de E/S, bancos de dados, etc.



Comunicação entre processos

Região crítica

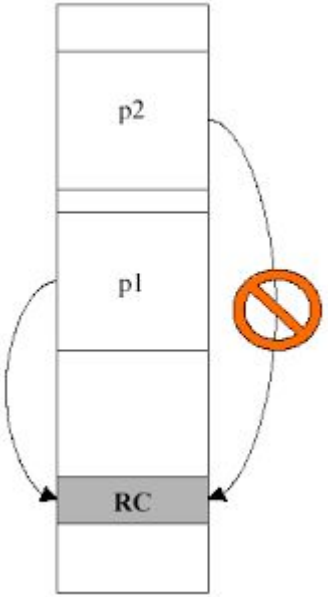
Área de um recurso compartilhado que exige um uso sequencial por parte dos processos, ou seja, não pode ser acessada por mais de um processo ao mesmo tempo.



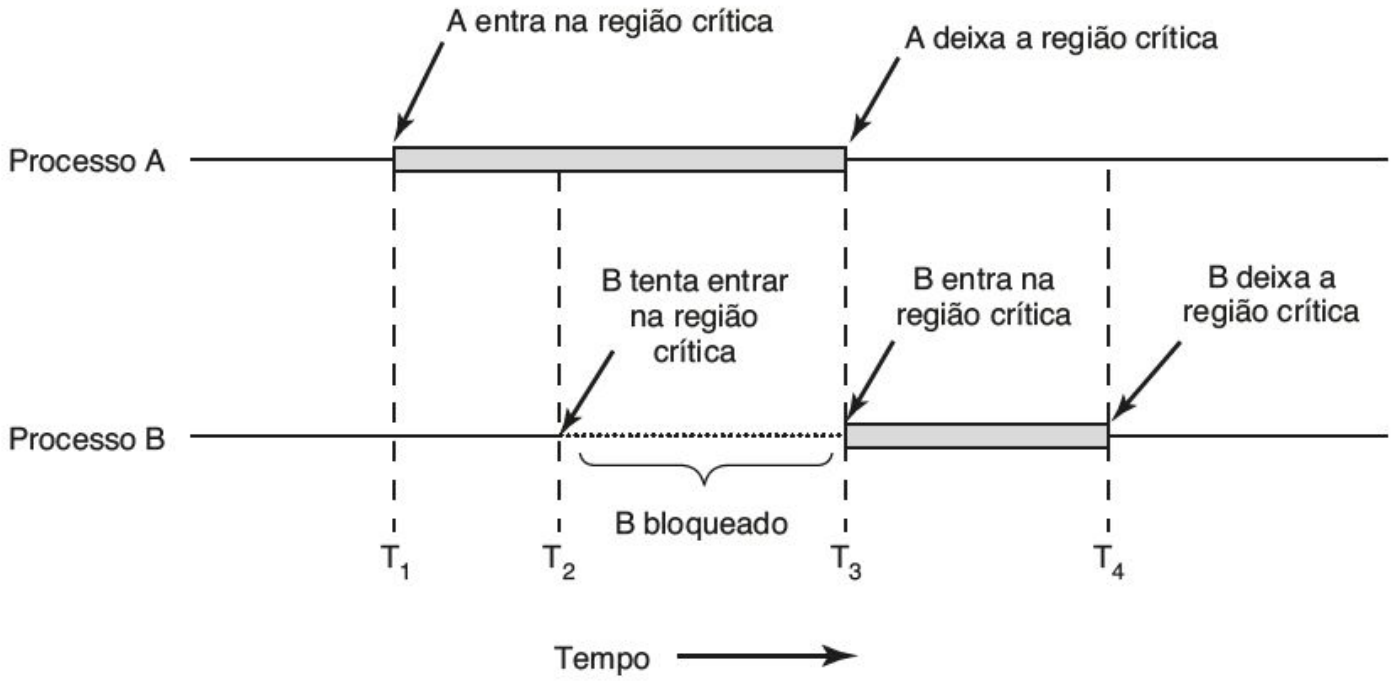
Comunicação entre processos

Região crítica

Solução: exclusão mútua (implementação de métodos que impedem as falhas de acesso citadas) com o uso de semáforos, monitores, *flags*, etc.



Regiões críticas

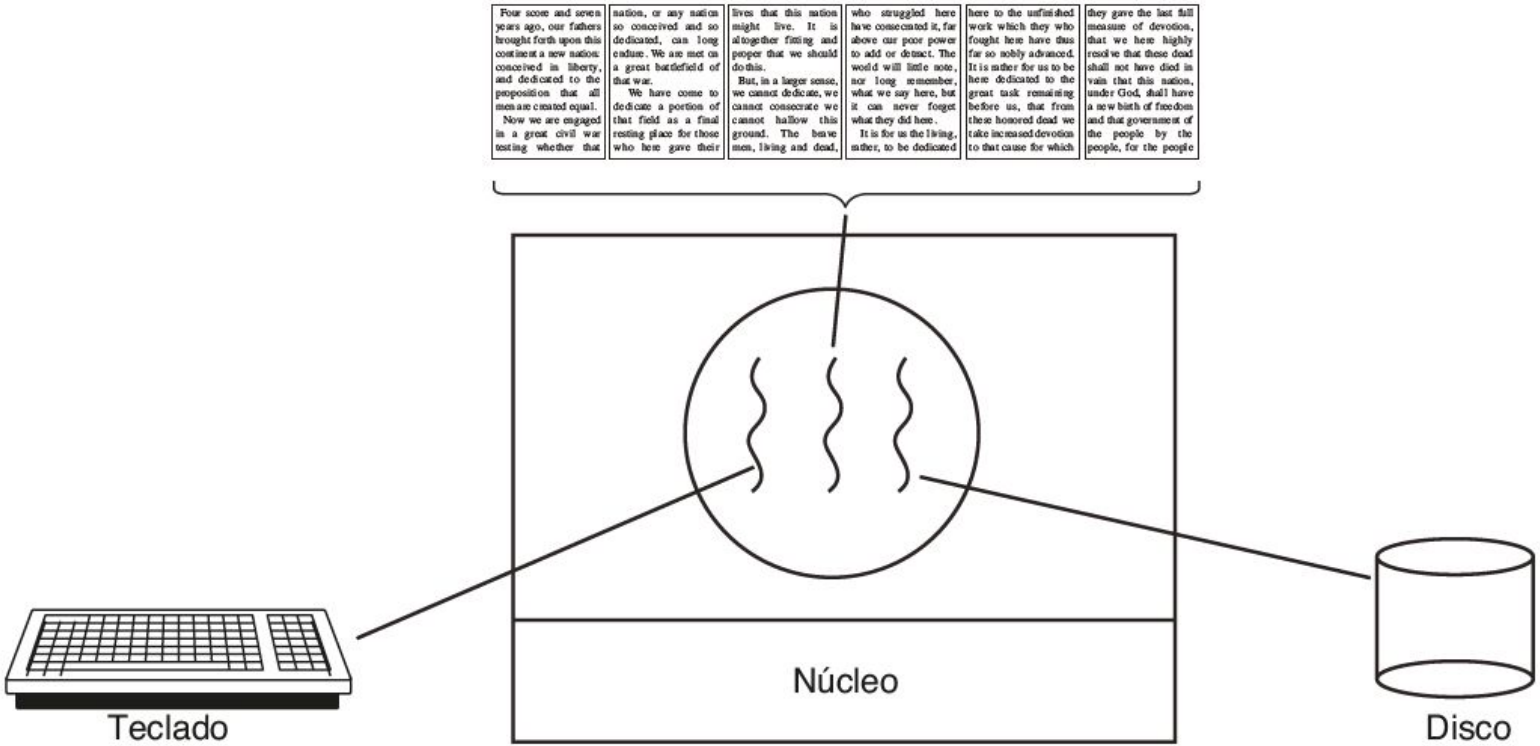


Threads

Consiste na divisão da execução do processo.

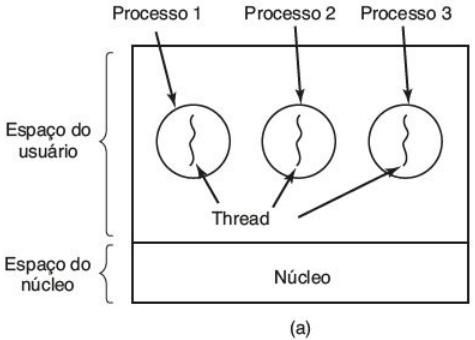
Cada thread realiza uma parte diferente do processo ao mesmo tempo.

Com essa divisão o programa é executado mais rapidamente.

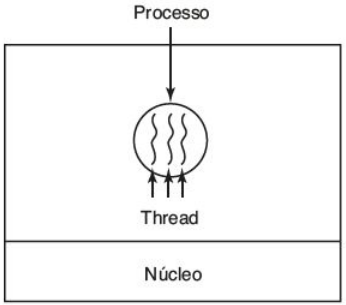


Andrew S. Tanenbaum, Herbert Bos - Sistemas Operacionais Modernos

Threads



(a)



Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Em comparação, vemos um único processo com três threads de controle.

Embora em ambos os casos tenhamos três threads, a primeira figura indica que cada um deles opera em um espaço de endereçamento diferente, enquanto na segunda figura todos os três compartilham o mesmo espaço de endereçamento.

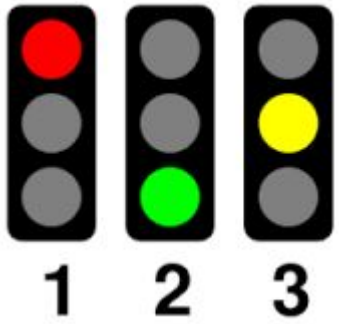
Semáforos

Variáveis especiais protegidas que gerenciam o controle de acesso aos recursos compartilhados.

Indicam quantas *threads* podem acessar o dado recurso.

Operações:

- Inicialização: quantidade de processos permitidos.
- *Wait*: decrementa o valor até zero (dormir).
- *Signal*: utilizado para despertar o processo.



Semáforos

As operações em um semáforo são atômicas, ou seja, apenas um processo pode executar as operações ao mesmo tempo e com o mesmo semáforo.

O segundo processo deve esperar o primeiro terminar sua operação, ou seja, entra em modo *sleep* (dormir) até poder ser acordado pela primitiva *wakeup*.

Mutexes

Um *mutex* é uma variável compartilhada que pode estar em um de dois estados: destravado ou travado.

Em consequência, apenas 1 bit é necessário para representá-lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando travado.

Mutexes em pthreads

Pthreads proporcionam uma série de funções que podem ser usadas para sincronizar *threads*. O mecanismo básico usa uma variável *mutex*, que pode ser travada ou destravada, para guardar cada região crítica.

Chamada de thread	Descrição
Pthread_mutex_init	Cria um mutex
Pthread_mutex_destroy	Destrói um mutex existente
Pthread_mutex_lock	Obtém uma trava ou é bloqueado
Pthread_mutex_trylock	Obtém uma trava ou falha
Pthread_mutex_unlock	Libera uma trava

Chamada de thread	Descrição
Pthread_cond_init	Cria uma variável de condição
Pthread_cond_destroy	Destrói uma variável de condição
Pthread_cond_wait	É bloqueado esperando por um sinal
Pthread_cond_signal	Sinaliza para outro thread e o desperta
Pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

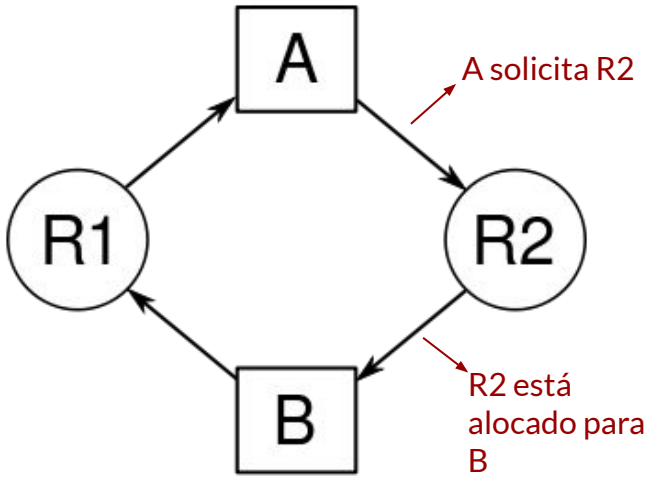
Deadlock

Quando dois ou mais processos ficam bloqueados de continuar suas execuções, ou seja, ficam bloqueados esperando uns pelos outros. Normalmente ocorre com recursos como dispositivos, arquivos, memória, etc.

Deadlock

Os processos A e B, cada um com um recurso alocado R1 e R2, se encontram.

Cada um solicita o recurso que está alocado ao outro processo (Espera circular).



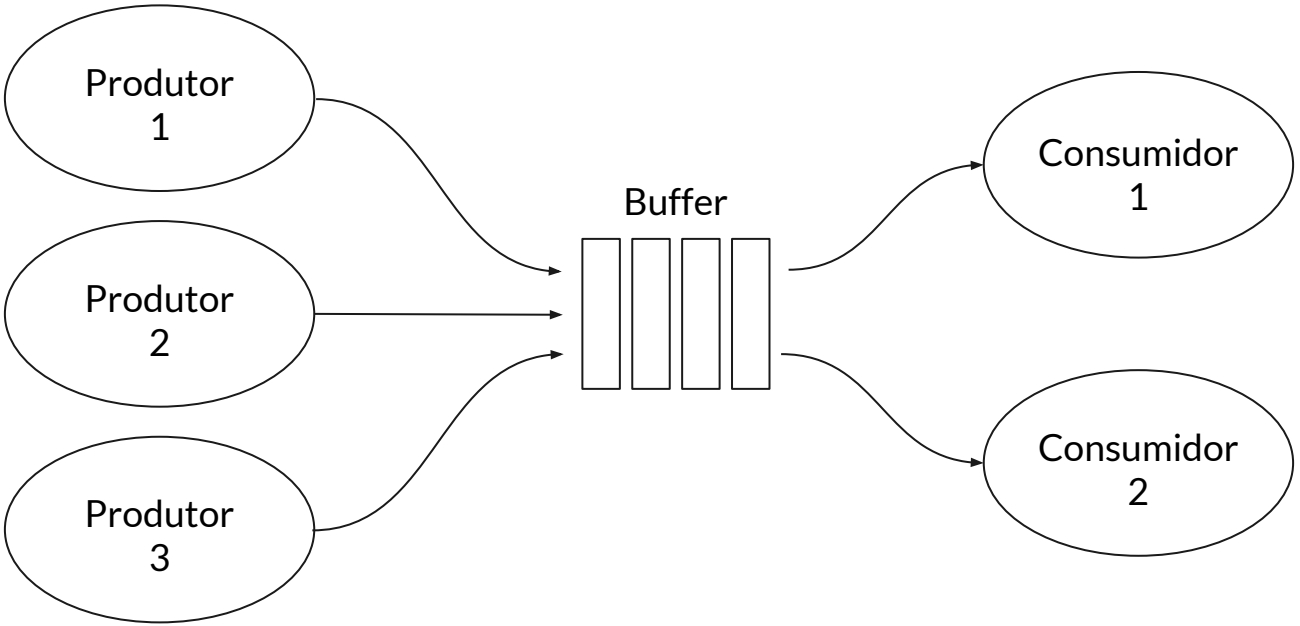
Problema do produtor-consumidor

Também conhecido como problema do ***buffer* limitado**.

Dois processos compartilham um mesmo *buffer* de tamanho fixo. O produtor insere informações no *buffer* e o consumidor as retira dele.

O problema surge quando o produtor quer inserir informações no *buffer* mas este se encontra cheio.

Problema do produtor-consumidor



Problema do produtor-consumidor

Solução: colocar o produtor para dormir (*sleep*) para ser desperto (*wakeup*) quando o consumidor tiver removido um ou mais itens.

Analogamente, se o consumidor quiser remover itens do *buffer*, estando este vazio, ele é posto para dormir (*sleep*) até que o produtor insira algo no *buffer*.

Problema do produtor-consumidor

Count: variável com a quantidade de itens no buffer.

O produtor precisa consultá-la antes de cada inserção, verificando se ela é menor do que o tamanho do *buffer*, caso contrário ele é posto para dormir (*sleep*).

No caso do consumidor, ele precisa verificar que *count* é maior do que zero, caso contrário ele também é posto para dormir (*sleep*).

Solução

Buffer e declaração das filas

```c
#define TAM_BUFFER 8    // Tamanho do buffer
#define NUM_INFO 20     // Numero de informacoes para processar
#define SLEEPING 0      // Valor 0 para status dormindo
#define AWAKE 1         // Valor 1 para status acordado
// struct da fila de consumo e producao
```

You, 34 minutes ago | 1 author (You)

```c
typedef struct fila{
    int posicao;        // Posicao para consumo ou producao
    struct fila *prox;  // Proxima posicao para consumo ou producao
} Fila;

// struct do head da fila
```

You, 34 minutes ago | 1 author (You)

```c
typedef struct head{
    Fila *inicio;   // Ponteiro para inicio da fila
    Fila *fim;      // Ponteiro para fim da fila
} Head;

Head trafego_producao;  // Fila de producao
Head trafego_consumo;   // Fila de consumo

int buffer[TAM_BUFFER];     // Buffer com o numero TAM_BUFFER de posicoes

int n_livre_buffer = TAM_BUFFER;    // Numero de info livres no buffer
int n_ocup_buffer = 0;              // Numero de info ocupadas no buffer
```

Controle da sessão crítica

```c
pthread_mutex_t mutex_estado_prod_cons = PTHREAD_MUTEX_INITIALIZER; // Mutex que verifica o estado (dormindo ou acordado)
                                                                   // consumidor e do produtor
pthread_mutex_t mutex_buffer = PTHREAD_MUTEX_INITIALIZER; // Mutex que controla o acesso ao buffer
pthread_cond_t cond_produtor = PTHREAD_COND_INITIALIZER; // Thread do produtor
pthread_cond_t cond_consumidor = PTHREAD_COND_INITIALIZER; // Thread do consumidor

int estado_consumo = AWAKE; // Seta o estado do consumidor para acordado
int estado_producao = AWAKE; // Seta o estado do produtor para acordado

int total_info = NUM_INFO; // Seta total de informações que seram transmitidas pelo buffer com o valor de NUM_INFO
```

Fila

```c
// Função que add um elemnto na fila recebida
// Retorna 1 para sucesso ou -1 para erro
int push(Head *head, int posicao){
    Fila *nova = (Fila *)malloc(sizeof(Fila)); // Aloca um novo elemento da fila
    if (nova == NULL){ // Caso ocorra erro
        printf(">> Erro de memoria!");
        return (-1);
    }
    nova->prox = NULL;
    nova->posicao = posicao;
    if (head->fim == NULL){ // Se o fim for null significa que a fila esta vazia
        head->inicio = nova; // Add o novo elemento no inicio
    }
    else{
        head->fim->prox = nova; // Se não, aponta o prox do penultimo elemnto para o novo
    }
    head->fim = nova; // Coloca o novo elemento no fim
    return (1);
}
```

Fila

```c
// Retorna a posicao quardada no primeiro elemento da fila e o libera
int pop(Head *head){
    int posicaoBuffer;
    Fila *pt;
    if (head->fim != NULL){
        pt = head->inicio;
        head->inicio = head->inicio->prox; //Passa próximo elemento para inicio
        if (head->inicio == NULL) //Se inicio for nulo, fila acabou
            head->fim = NULL;
        posicaoBuffer = pt->posicao;
        free(pt);
        return (posicaoBuffer); //Retorna primeiro elemento
    }
    return -1;
}
```

Produtor

```c
void *produtor(){
    //Informacao
    int info[NUM_INFO] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
    for (int i = 0; i < NUM_INFO; i++){
        pthread_mutex_lock(&mutex_estado_prod_cons); // Bloquear os status para dormir
        //Buffer vazio -> Produtor dorme
        if (n_livre_buffer == 0 || n_ocup_buffer == 8){
            printf("\nProdutor--> Dormir\n");
            estado_producao = SLEEPING; // Dormir
            pthread_cond_wait(&cond_produtor, &mutex_estado_prod_cons); // Aguarda o sinal para voltar a produzir
            estado_producao = AWAKE; // Acorddado
            printf("\nProdutor--> Acordar\n");
        }
        pthread_mutex_unlock(&mutex_estado_prod_cons); // Libera os status para dormir
        pthread_mutex_lock(&mutex_buffer); // Bloquear  buffer
        produz_info(info[i]); // Produz informacao
        if (estado_consumo == SLEEPING){
            printf("\nProdutor--> Acordar consumidor\n");
            sleep(rand() % 2);
            pthread_cond_signal(&cond_consumidor); // Envia sinal para o consumidor acordar
        }
        pthread_mutex_unlock(&mutex_estado_prod_cons); // Libera os status para dormir
        pthread_mutex_unlock(&mutex_buffer); // Libera buffer
        total_info--;
        sleep(rand() % 5); // Tempo aleatorio para pausar o produtor
    }
}
```

Produtor

```
int produz_info(char info){
    int indice = pop(&trafego_producao); // Pega a posição que ira produzir
    if (indice == -1){ // Caso a fila esteja vazia
        indice = n_ocup_buffer; // Utiliza a quantidade ocupada do buffer como indice
    }
    n_livre_buffer--;
    n_ocup_buffer++;
    if (push(&trafego_consumo, indice) == -1){ // Add o indice onde foi produzido na fila de trafego do consumidor
        printf(">> Erro de memoria!");
        exit(0);
    }
    buffer[indice] = info; // Add a producao no buffer
    printf("\n\nProdutor--> Produzi na posicao: %d\tValor:  %d\n",indice, buffer[indice]);
    printf("            Pid: %d\t\t        Tid: %u\n",getpid(),(unsigned int)pthread_self());
    return 0;
}
```

Consumidor

```c
void *consumidor(){
    while (1){
        pthread_mutex_lock(&mutex_estado_prod_cons); // Bloqueia os status para dormir
        if (n_ocup_buffer == 0){
            // Finalizou o numero de informacoes e o consumo no buffer
            if (total_info == 0 && n_livre_buffer == 8){
                printf("\nConsumidor--> Acabou o consumo\n\n\n");
                break;
            }
            printf("\nConsumidor--> Dormir\n");
            estado_consumo = SLEEPING;
            pthread_cond_wait(&cond_consumidor, &mutex_estado_prod_cons); // Aguarda o sinal para voltar a consumir
            estado_consumo = AWAKE;
            printf("\nConsumidor--> Acordar\n");
        }
        pthread_mutex_unlock(&mutex_estado_prod_cons); // Liberar os status para dormir
        pthread_mutex_lock(&mutex_buffer); // Liberar o buffer
        // Consome a informacao da fila
        if(consome_info() == -1){
            exit(0);
        }
        pthread_mutex_lock(&mutex_estado_prod_cons);
        if(estado_producao == SLEEPING){
            printf("\nConsumidor--> Acordar o produtor\n");
            sleep(rand() % 4);
            pthread_cond_signal(&cond_produtor); // Envia sinal para o produtor acordar
        }
        pthread_mutex_unlock(&mutex_estado_prod_cons);
        pthread_mutex_unlock(&mutex_buffer);
        sleep(rand() % 10); // Tempo aleatorio para pausar o consumidor
    }
}
```

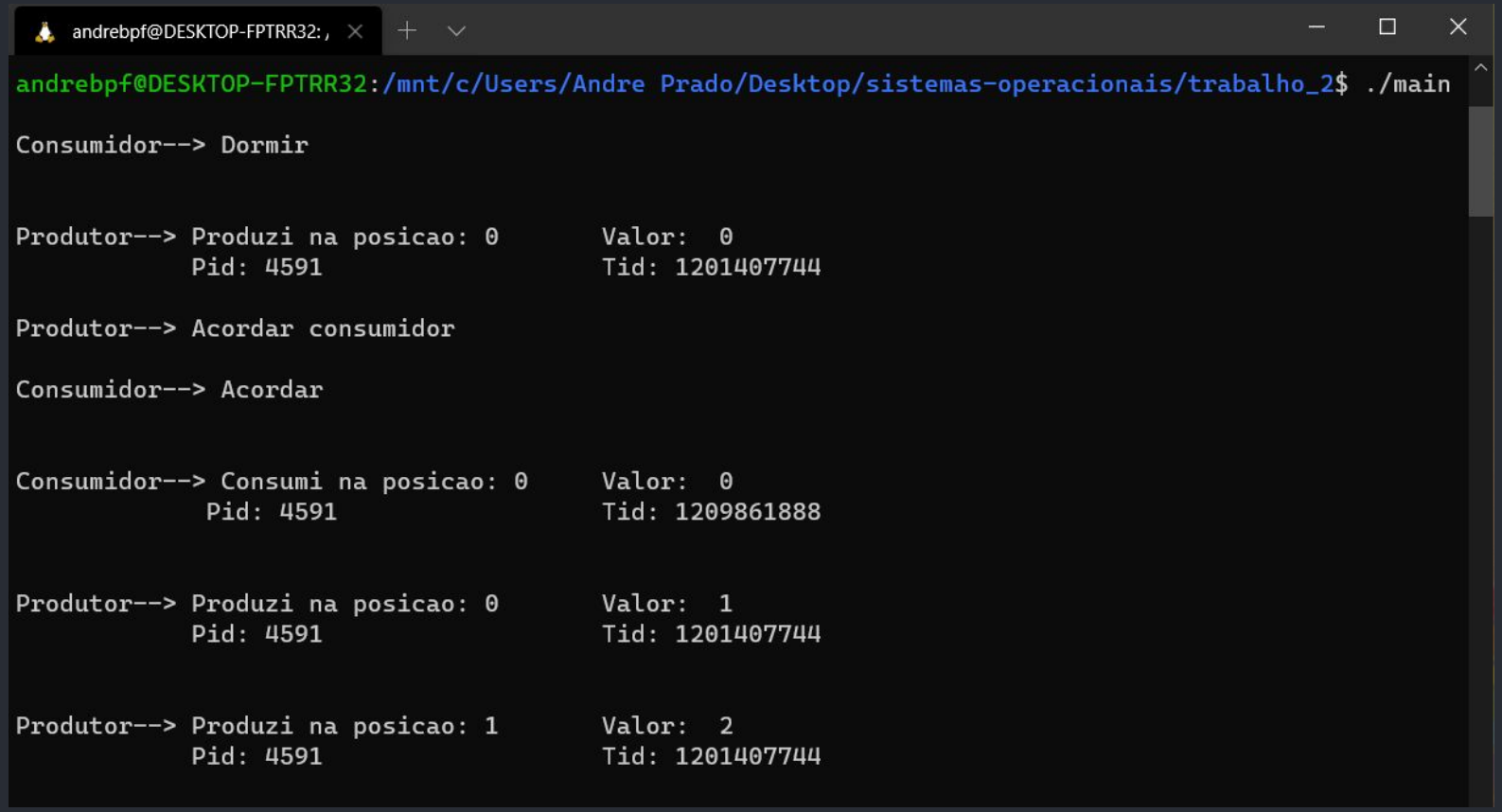
Consumidor

```c
// Acessa o buffer e consome na posicao indicada pela fila de trafego_consumo
int consome_info(void){
    int indice = pop(&trafego_consumo); // Pega o primeiro indice da fila
    if (indice != -1){ // Caso não haja erro
        // Printa o que foi consumido
        printf("\n\nConsumidor--> Consumi na posicao: %d\tValor:  %d\n",indice, buffer[indice]);
        printf("             Pid: %d\t\t        Tid: %u\n",getpid(),(unsigned int)pthread_self());
        buffer[indice] = -1;
        n_livre_buffer++; // Aumenta o buffer liver
        n_ocup_buffer--; // Diminui o buffer ocupado
        // Add na fila de trafego_producao o indice do buffer que foi consumido
        if (push(&trafego_producao, indice) == -1){ // Caso ocorra erro
            printf(">> Erro de memoria!");
            return -1;
        }
    }else{
        return -1;
    }
    return 0;
}
```

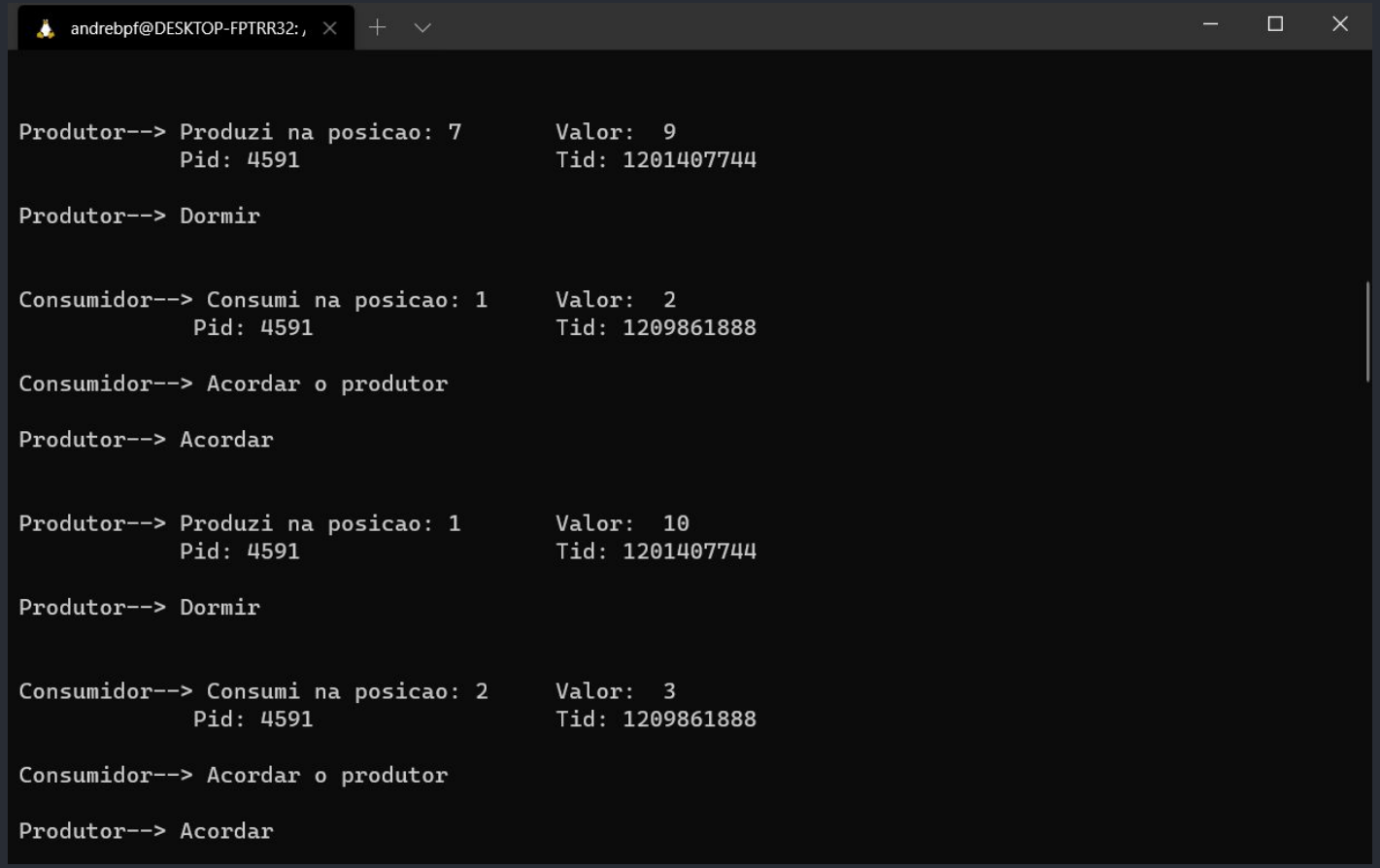
Threads e main

```c
int main(int argc, char *argv[]){
    // Declaracao das threads
    pthread_t thread_consumidor;
    pthread_t thread_produtor;
    // Threads para os consumidores
     if (pthread_create(&thread_consumidor, NULL, consumidor, (void*)0) != 0){
        printf("Erro ao criar a thread do consumidor");
        exit(1);
    }
    // Threads para os produtores
    if (pthread_create(&thread_produtor, NULL, produtor,  (void*)0) != 0){
        printf("Erro ao criar a thread do produtor");
        exit(1);
    }
    // Aguarda finalizar as threads
    if (pthread_join(thread_consumidor, NULL) != 0){
        printf("Erro ao finalizar a thread do consumidor");
        exit(1);
    }
    if (pthread_join(thread_produtor, NULL) != 0){
        printf("Erro ao finalizar a thread do produtor");
        exit(1);
    }
    return 0;
}
```

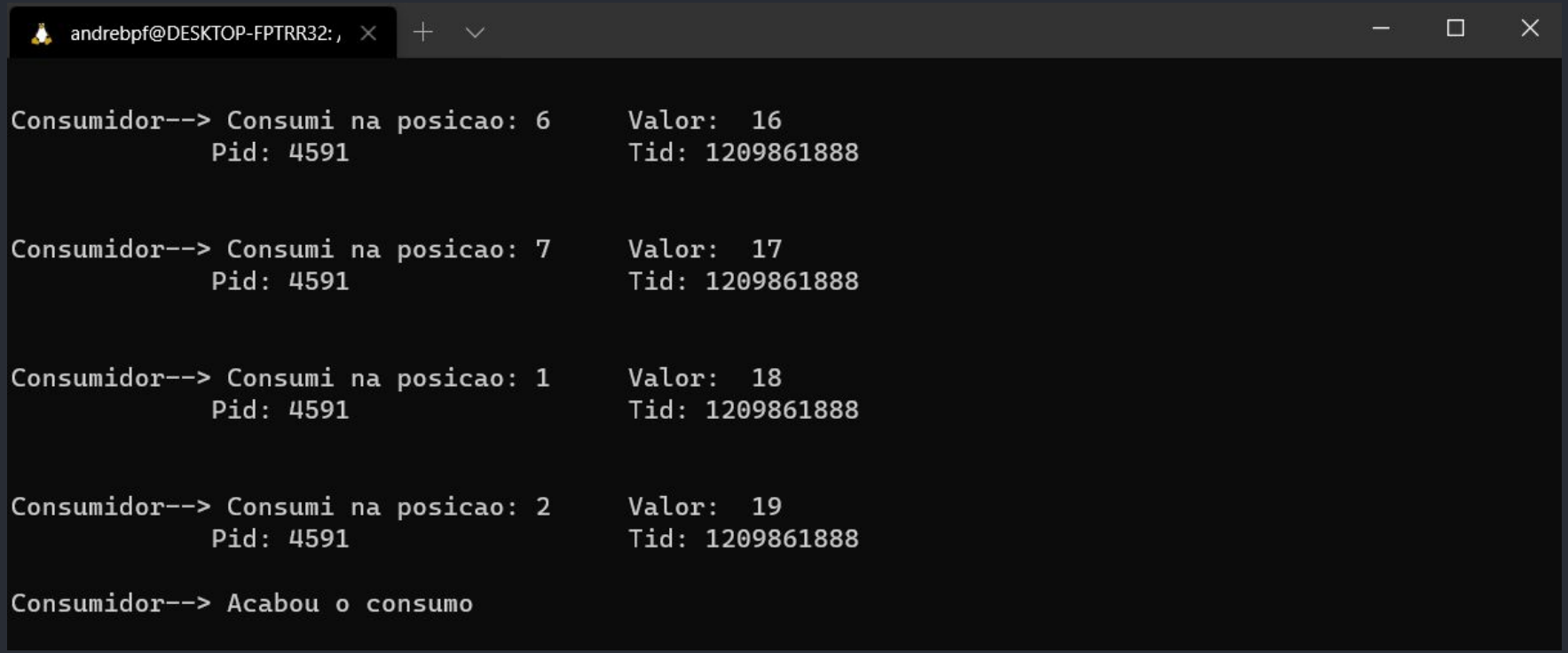
Execução


andrebpf@DESKTOP-FPTRR32:
andrebpf@DESKTOP-FPTRR32:/mnt/c/Users/Andre Prado/Desktop/sistemas-operacionais/trabalho_2$ ./main
Consumidor--> Dormir
Produtor--> Produzi na posicao: 0 Valor: 0
Pid: 4591 Tid: 1201407744
Produtor--> Acordar consumidor
Consumidor--> Acordar
Consumidor--> Consumi na posicao: 0 Valor: 0
Pid: 4591 Tid: 1209861888
Produtor--> Produzi na posicao: 0 Valor: 1
Pid: 4591 Tid: 1201407744
Produtor--> Produzi na posicao: 1 Valor: 2
Pid: 4591 Tid: 1201407744

Execução


andrebpf@DESKTOP-FPTRR32: ,
Produtor--> Produzi na posicao: 7 Valor: 9
Pid: 4591 Tid: 1201407744
Produtor--> Dormir
Consumidor--> Consumi na posicao: 1 Valor: 2
Pid: 4591 Tid: 1209861888
Consumidor--> Acordar o produtor
Produtor--> Acordar
Produtor--> Produzi na posicao: 1 Valor: 10
Pid: 4591 Tid: 1201407744
Produtor--> Dormir
Consumidor--> Consumi na posicao: 2 Valor: 3
Pid: 4591 Tid: 1209861888
Consumidor--> Acordar o produtor
Produtor--> Acordar

Execução


andrebpf@DESKTOP-FPTRR32: ,
Consumidor--> Consumi na posicao: 6 Valor: 16
Pid: 4591 Tid: 1209861888
Consumidor--> Consumi na posicao: 7 Valor: 17
Pid: 4591 Tid: 1209861888
Consumidor--> Consumi na posicao: 1 Valor: 18
Pid: 4591 Tid: 1209861888
Consumidor--> Consumi na posicao: 2 Valor: 19
Pid: 4591 Tid: 1209861888
Consumidor--> Acabou o consumo

Obrigado!