SSC0640 - Sistemas Operacionais I

Trabalho 2
Problema do Produtor e Consumidor

Grupo: gso03
Higor Tessari - nºUSP: 10345251
Lucas Tavares dos Santos - nºUSP: 10295180
Renata Oliveira Brito - nºUSP: 10373663

Link dos códigos no gitHub

https://github.com/lucast98/Sistemas-Operacionais-gso3

Link da explicação em vídeo

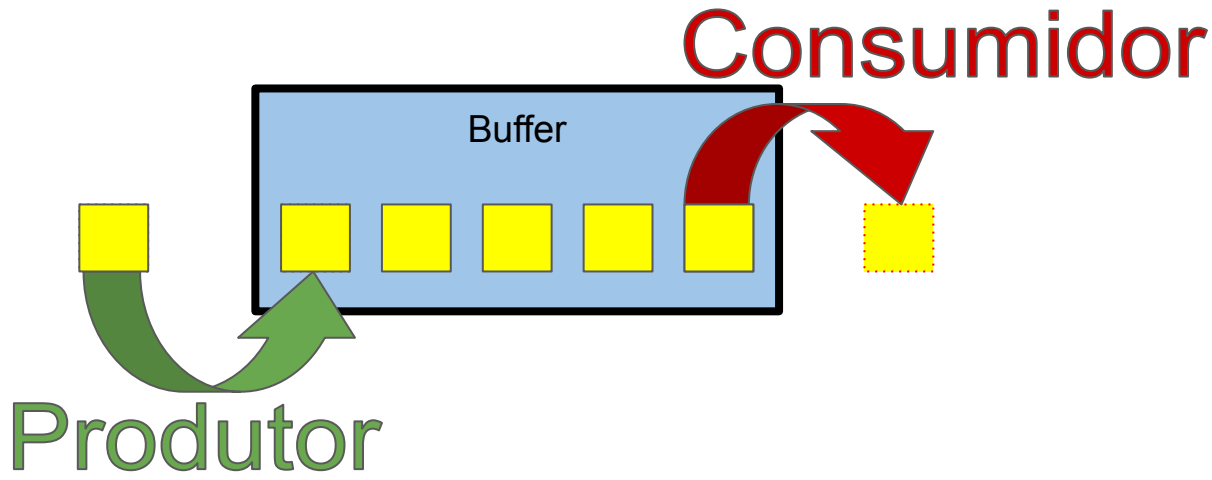
https://www.youtube.com/watch?v=JszMkAb0vwk

Problema do Produtor e Consumidor

Este problema consiste em um conjunto de processos que compartilham um mesmo buffer.

A informação é introduzida por processos produtores

E retiradas por processos consumidores.

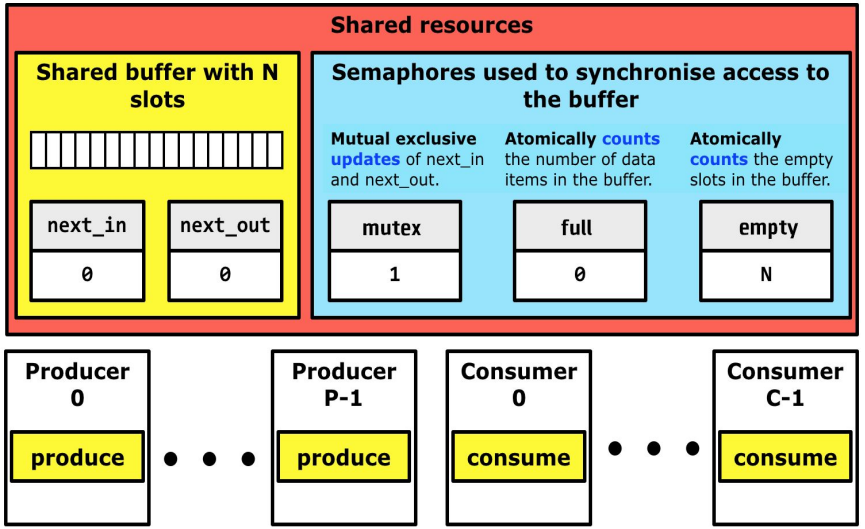


Problema do Produtor e Consumidor

Este problema também tem por objetivo trabalhar a sincronia de processos que utilizam o mesmo recurso.

Também foram tratados assuntos como espera ociosa.

Escalonamento de processos e uso de primitivas up e down para controle do tempo de uso da CPU.



Problema do Produtor e Consumidor

Na máquina virtual dentro da pasta “trab2”, está o arquivo “prodcons.c” correspondente a nossa solução para este problema, nela foram utilizados as seguintes bibliotecas:

<stdlib.h> que contém funções de alocação de memória e controle de processos.

<pthread.h> esta biblioteca permite gerar um novo fluxo de processo simultâneos como as threads.

<unistd.h> é a biblioteca que fornece acesso à api do sistema operacional, com essa podemos fazer o uso da função write e close por exemplo.

<queue.h> esta biblioteca serve para o uso de uma fila, em que o produtor possa produzir um item e o consumidor possa consumir o item, juntamente com suas devidas manipulações.

```
gso03@tau01-vm3:~/trab2$ cat prodcons.c
/**
 * Higor Tessari - 10345251
 * Lucas Tavares dos Santos - 10295180
 * Renata Oliveira Brito - 10373663
 */

#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include "queue.h"
```

prodcons.c

Fazemos a construção do problema na main do código prodcons.c, onde declaramos o tamanho limitado do buffer, o número dos produtores, a quantidade a ser produzida por cada produtor e o número de consumidores.

```
gso03@tau01-vm3:~/trab2$ cat prodcons.c
/**
 * Higor Tessari - 10345251
 * Lucas Tavares dos Santos - 10295180
 * Renata Oliveira Brito - 10373663
 */

#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include "queue.h"

const int tam_buffer = 10; /** Tamanho do buffer da fila */
const int produtores = 10; /** Numero de produtores */
const int qtd_prod = 20; /** Quantidade a ser produzida por cada produ
tor */
const int consumidores = 5; /** Numero de consumidores */

/** Função que representa as operações dos produtores */
void *produtor(void *arg){
    for (int i = 0; i < qtd_prod; ++ i){
        int *value = malloc(sizeof(*value)); //aloca um espaço de memo
ria para o valor a ser inserido na fila
        *value = i; //associa o numero da interacao ao valor a ser ins
erido na fila
        enqueue(arg, value); //insere o valor na fila
        if(i % 2 == 1) //condicao para trocar entre produtor e consumi
dor
            sleep(1);
    }
}
```

prodcons.c

Criamos um função que representa as operações dos produtores, passando como parâmetro um ponteiro genérico que a função enqueue da biblioteca “queue.h” possa verificar que é uma fila, possibilitando que eles produzem do 0 até a quantidade máxima, alocando um espaço de memória para cada valor a ser inserido na fila. O número da iteração será o valor a ser inserido na fila, por fim chamamos a função enqueue, para inserirmos na fila, juntamente no buffer.

```
gso03@tau01-vm3:~/trab2$ cat prodcons.c
/**
 * Higor Tessari - 10345251
 * Lucas Tavares dos Santos - 10295180
 * Renata Oliveira Brito - 10373663
 */

#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include "queue.h"

const int tam_buffer = 10; /** Tamanho do buffer da fila */
const int produtores = 10; /** Numero de produtores */
const int qtd_prod = 20; /** Quantidade a ser produzida por cada produ
tor */
const int consumidores = 5; /** Numero de consumidores */

/** Função que representa as operações dos produtores */
void *produtor(void *arg){
    for (int i = 0; i < qtd_prod; ++ i){
        int *value = malloc(sizeof(*value)); //aloca um espaço de memo
ria para o valor a ser inserido na fila
        *value = i; //associa o numero da interacao ao valor a ser ins
erido na fila
        enqueue(arg, value); //insere o valor na fila
        if(i % 2 == 1) //condicao para trocar entre produtor e consumi
dor
            sleep(1);
    }
}
```

prodcons.c

Criamos um função que representa as operações dos consumidores, passando como parâmetro um ponteiro genérico para que a função dequeue possa verificar que é uma fila, na nossa iteração manipulamos de tal forma que podemos dividir “igualmente” para cada consumidor, consumindo o elemento e retornando o valor que foi retirado do buffer, liberando o espaço na memória.

```
            sleep(1);
    }
}

/** Função que representa as operações dos consumidores */
void *consumidor(void *arg){
    for (int i = 0; i < qtd_prod * produtores / consumidores; ++ i){
        int *value = dequeue(arg); //remove o elemento e retorna o val
or que foi retirado
        free(value); //libera o valor da memoria
        if (i % 2 == 1) //condicao para trocar entre produtor e consum
idor
            sleep(1);
    }
}

int main(){
    int i;
    void *buffer[tam_buffer]; //buffer limitado
    queue_t fila = { buffer, sizeof(buffer) / sizeof(buffer[0]), 0, 0,
 0, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, PTHREAD_COND_
INITIALIZER }; //condicoes de inicio da fila

    pthread_t produtores[produtores]; //thread dos produtores
    for (i = 0; i < sizeof(produtores) / sizeof(produtores[0]); ++ i){
        if(pthread_create(&produtores[i], NULL, produtor, &fila) != 0)
 //cria os produtores
            return -1; //erro
    }
    pthread_t consumidores[consumidores]; //thread dos consumidores
    for (i = 0; i < sizeof(consumidores) / sizeof(consumidores[0]); ++
 i){
```

prodcons.c

Na main criaremos as threads para eliminar a espera ociosa dos processos, para isso, iniciaremos o buffer, a fila e as threads dos produtores e dos consumidores, e utilizamos a chamada de sistema **pthread_join** para que possa haver essa sincronização das threads.

```
int main(){
    int i;
    void *buffer[tam_buffer]; //buffer limitado
    queue_t fila = { buffer, sizeof(buffer) / sizeof(buffer[0]), 0, 0,
 0, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, PTHREAD_COND_
INITIALIZER }; //condicoes de inicio da fila

    pthread_t produtores[produtores]; //thread dos produtores
    for (i = 0; i < sizeof(produtores) / sizeof(produtores[0]); ++ i){
        if(pthread_create(&produtores[i], NULL, produtor, &fila) != 0)
 //cria os produtores
            return -1; //erro
    }
    pthread_t consumidores[consumidores]; //thread dos consumidores
    for (i = 0; i < sizeof(consumidores) / sizeof(consumidores[0]); ++
 i){
        if(pthread_create(&consumidores[i], NULL, consumidor, &fila) !
= 0) //cria os consumidores
            return -1; //erro
    }
    for (i = 0; i < sizeof(produtores) / sizeof(produtores[0]); ++ i){
        pthread_join(produtores[i], NULL); //permite que uma thread es
pere outra terminar sua execução
    }
    for (i = 0; i < sizeof(consumidores) / sizeof(consumidores[0]); ++
 i){
        pthread_join(consumidores[i], NULL); //permite que uma thread
espere outra terminar sua execução
    }
}
```

queue.h

<stdio.h> que contém várias funções de entrada e saída, como exemplo as funções de printf do nosso programa.

<sys/syscall.h> é um cabeçalho que contém as chamadas de sistemas.

```
gso03@tau01-vm3:~/trab2$ cat queue.h
/**
 * Higor Tessari - 10345251
 * Lucas Tavares dos Santos - 10295180
 * Renata Oliveira Brito - 10373663
 */

#include <pthread.h>
#include <sys/syscall.h>
#include <stdio.h>

typedef struct fila{
    void **buffer; /** buffer limitado */
    const int capacidade; /** tamanho maximo da fila */
    int tam; /** quantidade de elementos na fila */
    int next_in; /** variavel que representa a posicao dos elementos i
nseridos na fila */
    int next_out; /** variavel que representa a posicao dos elementos
removidos da fila */
    pthread_mutex_t mutex; /** controla o acesso a regiao critica */
    pthread_cond_t empty;  /** conta os lugares vazios no buffer */
    pthread_cond_t full; /** conta os lugares preenchidos no buffer */
} queue_t;

/** Função para inserir elementos na fila */
void enqueue(queue_t *fila, void *value){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == fila->capacidade) //se tiver lotada
        pthread_cond_wait(&(fila->full), &(fila->mutex)); //coloca o p
rocesso em modo de espera
    printf("Produtor com ID %u produziu %d \n", t, *(int *)value);
    fila->buffer[fila->next_in] = value; //insere o valor no buffer
    ++fila->tam; //incrementa quantidade de elementos na fila
```

queue.h

Na construção da fila, optamos por colocar dentro da estrutura o buffer, o tamanho máximo da fila, a quantidade de elementos, uma variável que representa a posição de onde será inserido o próximo na fila, outra que representa a posição de remoção de elementos e os semáforos, utilizando a biblioteca pthread, criamos os semáforos mutex, empty e full.

```
gso03@tau01-vm3:~/trab2$ cat queue.h
/**
 * Higor Tessari - 10345251
 * Lucas Tavares dos Santos - 10295180
 * Renata Oliveira Brito - 10373663
 */

#include <pthread.h>
#include <sys/syscall.h>
#include <stdio.h>

typedef struct fila{
    void **buffer; /** buffer limitado */
    const int capacidade; /** tamanho maximo da fila */
    int tam; /** quantidade de elementos na fila */
    int next_in; /** variavel que representa a posicao dos elementos i
nseridos na fila */
    int next_out; /** variavel que representa a posicao dos elementos
removidos da fila */
    pthread_mutex_t mutex; /** controla o acesso a regiao critica */
    pthread_cond_t empty;  /** conta os lugares vazios no buffer */
    pthread_cond_t full; /** conta os lugares preenchidos no buffer */
} queue_t;

/** Função para inserir elementos na fila */
void enqueue(queue_t *fila, void *value){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == fila->capacidade) //se tiver lotada
        pthread_cond_wait(&(fila->full), &(fila->mutex)); //coloca o p
rocesso em modo de espera
    printf("Produtor com ID %u produziu %d \n", t, *(int *)value);
    fila->buffer[fila->next_in] = value; //insere o valor no buffer
    ++fila->tam; //incrementa quantidade de elementos na fila
```

queue.h

Para que os produtores possam inserir os elementos da fila, criamos a função enqueue, onde será esperado como parâmetro a fila e o valor que deseja inserir, inicialmente chamamos o semáforo binário (mutex) para controlar a região crítica, bloqueando-a. Enquanto a fila está totalmente preenchida, ou seja, os produtores produziram a quantidade máxima possível, o processo deles entram em modo de espera para que os consumidores, consumam os elementos da fila.

```
/** Função para inserir elementos na fila */
void enqueue(queue_t *fila, void *value){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == fila->capacidade) //se tiver lotada
        pthread_cond_wait(&(fila->full), &(fila->mutex)); //coloca o p
rocesso em modo de espera
    printf("Produtor com ID %u produziu %d \n", t, *(int *)value);
    fila->buffer[fila->next_in] = value; //insere o valor no buffer
    ++fila->tam; //incrementa quantidade de elementos na fila
    ++fila->next_in; //incrementa a posicao da fila
    fila->next_in %= fila->capacidade; //faz voltar para o inicio se c
hegou ao fim
    pthread_mutex_unlock(&(fila->mutex)); //desbloqueia o acesso a reg
iao critica
    pthread_cond_broadcast(&(fila->empty)); //desbloqueia todas as thr
eads bloqueadas na variavel de condição 'empty'
}
```

queue.h

Porém se a fila não está cheia o processo dos produtores não é bloqueado e eles inserem elementos no buffer que está na fila, atualizando a quantidade de elementos na fila e a posição da fila, caso chegue na última posição a ser inserida, a variável que representa a posição dos elementos inseridos na fila volta para o início da fila, quando o produtores terminam de produzir, a região crítica é desbloqueada.

```
/** Função para inserir elementos na fila */
void enqueue(queue_t *fila, void *value){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == fila->capacidade) //se tiver lotada
        pthread_cond_wait(&(fila->full), &(fila->mutex)); //coloca o p
rocesso em modo de espera
    printf("Produtor com ID %u produziu %d \n", t, *(int *)value);
    fila->buffer[fila->next_in] = value; //insere o valor no buffer
    ++fila->tam; //incrementa quantidade de elementos na fila
    ++fila->next_in; //incrementa a posicao da fila
    fila->next_in %= fila->capacidade; //faz voltar para o inicio se c
hegou ao fim
    pthread_mutex_unlock(&(fila->mutex)); //desbloqueia o acesso a reg
iao critica
    pthread_cond_broadcast(&(fila->empty)); //desbloqueia todas as thr
eads bloqueadas na variavel de condição 'empty'
}
```

queue.h

Para que os consumidores pudessem remover os elementos da fila, criamos a função dequeue, onde será esperado como parâmetro somente fila. Além disso, como há a preocupação sobre como lidar com a região crítica, foi utilizado também um semáforo binário. Enquanto a fila estiver vazia, o processo dos consumidores estará em modo de espera.

```
/** Função para remover elementos da fila */
void* dequeue(queue_t *fila){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == 0) //se tiver vazia
        pthread_cond_wait(&(fila->empty), &(fila->mutex)); //coloca o
processo em modo de espera
    void *value = fila->buffer[fila->next_out]; //remove o valor do bu
ffer
    printf("Consumidor com ID %u consumiu %d \n", t, *(int *)value);
    --fila->tam; //decrementa quantidade de elementos na fila
    ++fila->next_out; //incrementa a posicao da fila
    fila->next_out %= fila->capacidade; //faz voltar para o inicio se
chegou ao fim
    pthread_mutex_unlock(&(fila->mutex)); //desbloqueia o acesso a reg
iao critica
    pthread_cond_broadcast(&(fila->full)); //desbloqueia todas as thre
ads bloqueadas na variavel de condição 'empty'
    return value;
}
```

queue.h

No entanto, se ela possuir ao menos um elemento, o consumidor consumirá, atualizando a quantidade de elementos e a posição da fila. Caso chegue na última posição a ser consumida, a variável representante da posição dos elementos removidos da fila volta para o início da fila. Finalizando a ação dos consumidores, desbloqueamos a região crítica com o semáforo binário.

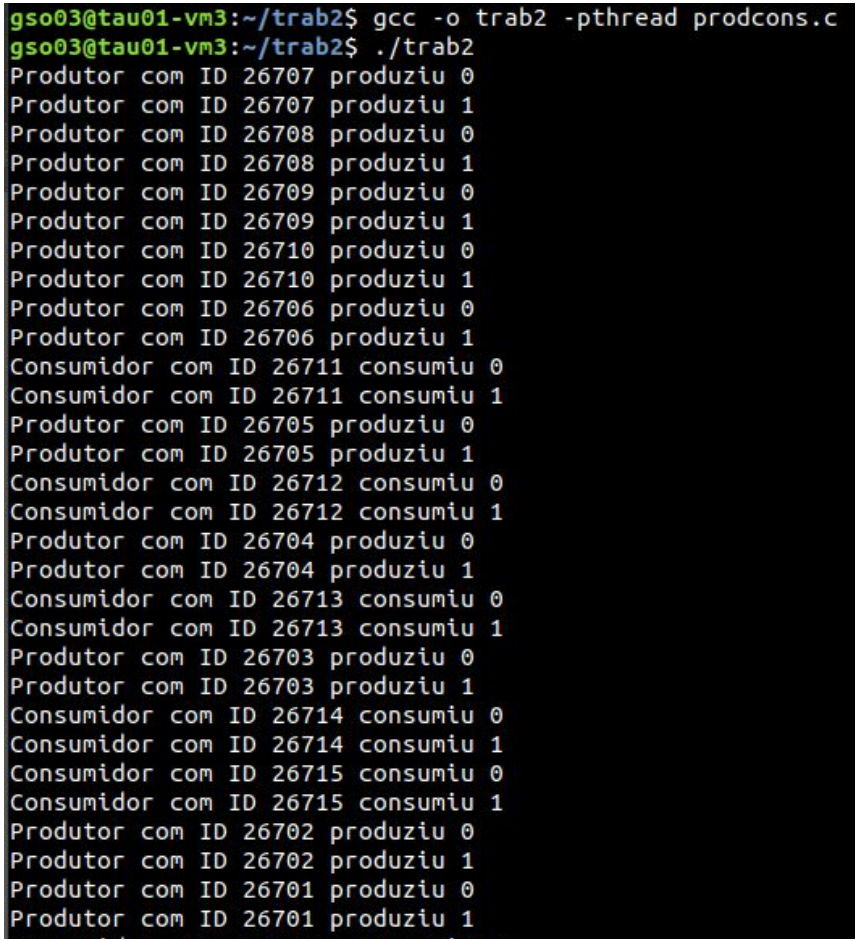
```
/** Função para remover elementos da fila */
void* dequeue(queue_t *fila){
    int t = syscall(SYS_gettid);
    pthread_mutex_lock(&(fila->mutex)); //bloqueia o acesso a regiao c
ritica
    while (fila->tam == 0) //se tiver vazia
        pthread_cond_wait(&(fila->empty), &(fila->mutex)); //coloca o
processo em modo de espera
    void *value = fila->buffer[fila->next_out]; //remove o valor do bu
ffer
    printf("Consumidor com ID %u consumiu %d \n", t, *(int *)value);
    --fila->tam; //decrementa quantidade de elementos na fila
    ++fila->next_out; //incrementa a posicao da fila
    fila->next_out %= fila->capacidade; //faz voltar para o inicio se
chegou ao fim
    pthread_mutex_unlock(&(fila->mutex)); //desbloqueia o acesso a reg
iao critica
    pthread_cond_broadcast(&(fila->full)); //desbloqueia todas as thre
ads bloqueadas na variavel de condição 'empty'
    return value;
}
```

Interação e resultados:

Após a implementação dos programas no diretório onde se encontram os mesmos, segue os seguintes passos para a execução:

- Criar os binários com “gcc -o nome_exec -pthread nome_arq.c”
- Executá-los com “./nome_exec”

Executando o algoritmo temos a resposta de encaminhamento de processos, e remoção de processos em um buffer de tamanho 10.



Conclusão

Foi possível notar que os princípios de sincronização de processos foram atendidos, pois os produtores inserem no buffer até serem postos em modo de espera, e só aí o consumidor começa a agir, mantendo esse ciclo.

Todas as atualizações do buffer foram feitas em uma região crítica, ou seja, a exclusão mútua foi respeitada. Para isso, o semáforo binário foi indispensável.

Além disso, a implementação mostra o quão cuidadoso é preciso ser quando usa semáforos. Um erro sutil e tudo para completamente (como por exemplo trocar a iteração da função dos consumidores).

Bibliografia

www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

www.researchgate.net/figure/Producer-Consumer-implemented-with-PThreads-ANSI-C_fig1_220856559

www.it.uu.se/education/course/homepage/os/vt18/module-4/bounded-buffer/