# Programação Concorrente

## Comunicação Entre Processos - Dormir e Acordar

Professor: **Jeremias Moreira Gomes**
E-mail: jeremiasmg@gmail.com



# Introdução



# Introdução

- Até agora foram apresentadas soluções que utilizam a **espera ocupada** para implementar a exclusão mútua.
- No entanto, a exclusão mútua comumente é implementada utilizando o bloqueio de processos, que esperam a permissão de entrada na seção crítica.



# Bloqueio de Processos



# Bloqueio de Processos

- Por que utilizar o bloqueio de processos?
  - **Evitar desperdício de CPU**
  - Evitar o problema da prioridade invertida (1/2)
    - Se o algoritmo de escalonamento utilizado é um algoritmo de **escalonamento com prioridades estáticas**, e se os processo que acessam a seção crítica possuem classes de prioridades diferentes, pode-se chegar a seguinte situação:



# Bloqueio de Processos

- Problema da prioridade invertida (2/2)
    - ... pode-se chegar a seguinte situação:
        - Processo P1 (prioridade baixa) entra na seção crítica
        - Processo P2 (prioridade alta) entra no *loop* de teste de acesso a seção crítica
        - Como a execução do *loop* não causa o bloqueio de P2 ele só sai pelo quantum, mas nesse caso um processo com a a mesma prioridade que a de P2 ou maior será escolhido
        - P1 não é executado nunca mais

# Bloqueio de Processos

- Geralmente são utilizadas primitivas para implementar o bloqueio de processos à espera pela seção crítica.
- Essas primitivas são implementadas em pares (**Dormir e Acordar**):
  - Uma primitiva para bloquear o processo quando a seção crítica já estiver ocupada.
  - Outra primitiva para desbloquear os processos à espera da permissão de acesso à seção crítica.



# Problema do Produtor-Consumidor

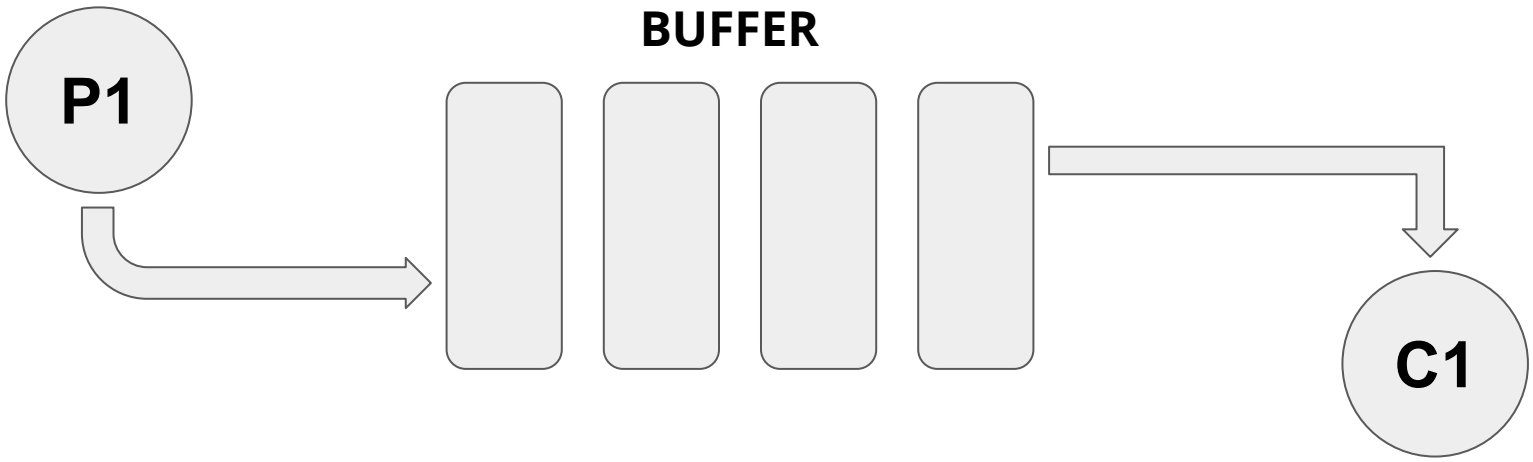


# Problema do Produtor-Consumidor

- Problema clássico utilizado para demonstrar os problemas e o uso de primitivas envolvendo o bloqueio de processos.
- Também conhecido com o problema do **buffer limitado**.

# Problema do Produtor-Consumidor

- Dois processos P1 e C1 compartilham um buffer de tamanho fixo.
- O processo P1 escreve dados no buffer.
- O processo C1 retira dados do buffer.





# Problema do Produtor-Consumidor

- O que fazer quando o produtor quer colocar um item no buffer mas ele já está cheio?
  - Ir dormir.
  - Ser acordado quando um ou mais itens forem consumidos.
- O que fazer se o consumidor for consumir um item do buffer mas ele estiver vazio?
  - Ir dormir.
  - Ser acordado quando um item for colocado no buffer.



# Problema do Produtor-Consumidor

- Aparentemente é uma abordagem simples, mas esse problema leva a diversas condições de corrida já apresentadas anteriormente, por lidarem com acessos compartilhados do recurso buffer.



# Problema do Produtor-Consumidor

- Por ser de tamanho fixo, o **buffer** precisa de uma **variável (`cont`)** que **controle** o **número itens (`N`)** nele.
  - Para que o produtor não escreva quando o buffer estiver cheio.
  - Para que o consumidor não retire dados quando o buffer estiver vazio.
- **O número de itens no buffer é uma variável compartilhada**.
  - O acesso a ela pode levar a condições de corrida.



# Problema do Produtor-Consumidor

- **Produtor:**
  - Testa se **`cont`** é igual a **`N`**.
    - Se for, vai dormir.
    - Se não for, incrementa **`cont`** e acorda o consumidor, se necessário.
- **Consumidor:**
  - Testa se **`cont`** é igual a **`0`**.
    - Se for, vai dormir.
    - Se não for, decrementa **`cont`** e acorda o produtor, se necessário.



# Problema do Produtor-Consumidor

```
#define N 100
int cont = 0;

void produtor()
{
    int item;
    while (TRUE) {
        item = produz_item( );
        if (cont == N)
            dormir( );
        insere_item(item);
        cont = cont + 1;
        if (cont == 1)
            acordar(consumidor);
    }
}
```

```
void consumidor()
{
    int item;
    while (TRUE) {
        if (cont == 0)
            dormir( );
        item = remove_item( );
        cont = cont - 1;
        if (cont == N - 1)
            acordar(producer);
        consome_item(item);
    }
}
```



# Problema do Produtor-Consumidor

- **Condição de Corrida do código anterior:**

  a. O **`buffer`** encontra-se vazio.
  b. Consumidor lê **`cont`** e percebe que está em zero.
  c. Escalonador troca consumidor pelo produtor.
  d. Produtor insere um item (01) no **`buffer`**.
  e. Produtor acorda consumidor (que já está acordado).
  f. Escalonador troca de produtor para consumidor.
  g. Consumidor que lêu zero (b), vai dormir.
  h. Produtor produz até encher o **`buffer`** e também vai dormir.



# Problema do Produtor-Consumidor

- A essência do problema apresentado no código anterior, é a **perda do sinal de acordar**.
  - O sinal é enviado para um processo que já está acordado.
- Uma solução com dois processos, seria criar uma região para armazenar um bit com o sinal de acordar.
  - Antes de ir dormir, o processo checa esse bit, e se ele for verdadeiro, o processo não dorme.
  - Só funciona de maneira simples para dois processos.



# Semáforos

# Semáforos

- Semáforos foram definidos por Dijkstra em 1965.
  - Sugeriu uma variável inteira para contar o número de sinais de acordar.
- Variável utilizada para controlar o acesso a recursos compartilhados:
  - Sincronizar o uso de recursos em grande quantidade
  - Garantir exclusão mútua
  - Semáforo == 0: Recurso está sendo utilizado
  - Semáforo > 0: Recurso está livre



# Semáforos

- Um semáforo é uma variável inteira que **admite somente valores não-negativos**.
- **Duas operações indivisíveis** podem ser executadas sobre um semáforo:
  - P(sem) ou Down(sem)
  - V(sem) ou Up(sem)



# Semáforos

- Down (sem) ou P (sem): Decrementa o valor do semáforo se este for maior que 0. Se o valor do semáforo for 0, o processo que executou a operação é bloqueado.

```
Down(sem):
    if (sem > 0)
        sem = sem - 1;
    else
        bloqueia(sem);
```

# Semáforos

- Up (sem) ou V (sem): Incrementa o valor do semáforo e acorda os processos que estiverem bloqueados no semáforo.

```
Up(sem):
    if (processos_espera())
        acorda_processo();
    Sem = sem + 1;
```

# Semáforos

- A verificação do valor do semáforo, a modificação e a operação de dormir devem e são feitas por **ação atômica** indivisível.
    - Quando um semáforo começa suas operações, ele sempre termina.
- Atomicidade é uma propriedade essencial para resolver problemas de sincronização e evitar condições de corrida em programação concorrente.



# Semáforos

- Durante a operação de acordar, se mais de um processo estiver dormindo, um destes é escolhido (aleatoriamente) para acordar e continuar sua execução.
- Além disso, e reforçando, nenhum processo é bloqueado durante a operação de acordar.



# Semáforos

- Delimitando uam seção crítica com semáforos:
  - Admita que o valor inicial do semáforo sem é 1.

```
Down(sem);
/* Seção Crítica */
Up(sem);
```



# Semáforos

- **Semáforos Binários**: só podem assumir os valores 0 ou 1.
- **Semáforos Generalizados**: podem assumir também valores negativos.
- **Semáforos com Busy Waiting**: Os semáforos também podem ser implementados de maneira mais simples.

```
P(sem):
    while (sem <= 0) {}
    sem = sem - 1;

V(sem):
    sem = sem + 1;
```

# Problema do Produtor-Consumidor - Semáforos

```
semaphore mutex = 1; // Seção crítica
semaphore vazio = N;
semaphore cheio = 0;

void produtor()
{
    int item;
    while (TRUE) {
        produz_item(&item);
        Up(&vazio);
        Up(&mutex);
        insere_item(&item);
        Down(&mutex);
        Down(&cheio);
    }
}
```

```
/* continuação  */


void consumidor()
{
    int item;
    while (TRUE) {
        Up(&cheio);
        Up(&mutex);
        remove_item(&item);
        Down(&mutex);
        Down(&vazio);
        consome_item(item);
    }
}
```

# Semáforos

- Três semáforos são utilizados nesta solução, com dois propósitos distintos.
  - Mutex - **Garantir a Exclusão Mútua**
    - Apenas um processo irá ler ou escrever no **`buffer`**.
  - Vazio e cheio - **Garantir a Sincronização**
    - Determinadas sequências não podem ocorrer.
      - Produtor para de produzir quando o buffer estiver cheio.
      - Consumidor para de consumir quando o buffer estiver vazio.



# Semáforos

- Para que a utilização de semáforos funcione corretamente, Up e Down precisam ser atômicos.
  - Normalmente, são implementados por Chamadas de Sistema
  - SO desabilita temporariamente todas as interrupções, para testar o semáforo.
  - Com múltiplas CPUs, TSL é utilizado para um processo checar o semáforo por vez.
    - Diferente da espera ocupada que pode ficar por muito tempo esperando.



# Mutexes



# Mutex

- Versão simplificada de um semáforo.
    - Não realiza contagem
- Servem para gerenciar a exclusão mútua ou trechos de código compartilhado.
- São fáceis e eficientes de implementar.
    - Presentes em bibliotecas de *threads* (espaço de usuário).



# Mutex

- É uma **variável compartilhada**.
  - Possui dois estados: travado xor destravado.
  - Necessita apenas de um bit para representação (usa um int).
- É concedido a um processo (*thread*) de cada vez.
- Operações:
  - acquire(1), lock(1), mutex_lock(1): Obtém a propriedade.
  - release(1), unlock(1) ou mutex_unlock(1): Libera a propriedade.



# Mutex

- Diferente do TSL, utilizando espera ocupada, quando o mutex_lock falha em adquirir o *lock* (a trava), a *thread* chama uma função **`thread_yield()`** para abrir mão da CPU para outra *thread*.
  - Dessa forma, não há espera ocupada.
  - Por ser uma chamada para o escalonador, a função é **`thread_yield()`** extremamente rápida.
    - Não exigem chamadas para o modo núcleo.



# Mutex em PThreads

- PThreads possuem uma série de funções para sincronização de *threads*.
- De maneira básica há uma variável **`mutex`** que pode ser travada ou destravada.
  - Se está destavada, uma *thread* pode adentrar a região crítica.
  - Se está travada, a *thread* que a chamou é bloqueada até ser desbloqueada.
    - Se há múltiplas *threads* bloqueadas em um mesmo mutex, uma é destravada por vez.



# Mutex em PThreads

- Travas e destravas não são obrigatórias de existirem em pares.
  - Liberdade e consciência do programador.
- Principais funções:

| Chamada | Descrição |
|---|---|
| `Pthread_mutex_init` | Cria um mutex |
| `Pthread_mutex_destroy` | Destrói um mutex existente |
| `Pthread_mutex_lock` | Obtém um *lock* ou é bloqueado |
| `Pthread_mutex_trylock` | Obté um *lock* ou falha |
| `Pthread_mutex_unlock` | Libera um *lock* |

# Mutex

- Além disso, cabe perguntar: se é possível lidarmos com diferentes processos (sem compartilhamento de espaço de endereços), **como realizar o compartilhamento de variáveis compartilhadas**?
  a. Armazenar no núcleo e responder por chamadas de sistema.
  b. Mecanismo de compartilhamento de informações (SO provê).
     - Torna obscuro a diferença entre processos e *threads*.
  c. Arquivos compartilhados.



# Variáveis de Condição



# Variáveis de Condição

- Mecanismo de sincronização além dos mutexes em PThreads.
- Mutexes são bons para permitir ou bloquear o acesso a uma região crítica.
- Variáveis de Condição permitem realizar bloqueios baseados no atendimento de condições.
- Mutexes e Variáveis de Condições costumam serem utilizados em conjunto.



# Variáveis de Condição

- Variáveis de condição permitem que um processo se bloqueie esperando uma ação de outro processo.
- Existem duas operações possíveis sobre as variáveis de condição:
  - **`wait(var)`**: bloqueia o processo em var
  - **`signal(var)`**: acorda os processos bloqueados em var



# Variáveis de Condição em PThreads

- Principais funções:

| Chamada | 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 |



# Variáveis de Condição

- Funcionamento das variáveis de condições no pacote PThreads:
  - Cada variável de condição possui um *lock* associado.
  - O processo obtém o *lock*(mutex_lock).
  - O processo libera o *lock* e espera na variável de condição(cond_wait).
  - Quando a condição desejada ocorre, o processo sinaliza(cond_signal).
  - O processo em wait é acordado e quando ele volta a executar está de posse do *lock*.



# Variáveis de Condição

- Variáveis de condição e mutexes são utilizados em conjunto.
- O ciclo comum é:
  - *Thread* trava um mutex.
  - *Thread* espera em uma variável condicional quando não tem o que precisa.
  - Quando que é necessário para essa *thread* está disponível, outra *thread* sinaliza que ela pode continuar.



# Variáveis de Condição

- Variáveis de condição não possuem memória.
  - Sinais enviados podem ser perdidos.
  - Fica a par do programador realizar a correta implementação e gerenciamento de sinais.



# Problema do Produtor-Consumidor - Variáveis de Condição (1/2)

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 10000

pthread_mutex_t xmutex;
pthread_cond_t condc, condp;
int buffer = 0;

void *produtor(void *empty);
void *consumidor(void *empty);
```

```c
/* continuação  */

int main()
{
    pthread_t pro, con;
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&pro, NULL, produtor, NULL);
    pthread_create(&con, NULL, consumidor, NULL);
    pthread_join(pro, NULL);
    pthread_join(con, NULL);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```



# Problema do Produtor-Consumidor - Variáveis de Condição (1/2)

```c
void *produtor(void *empty)
{
    for (int i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex);
        if (buffer != 0) {
            pthread_cond_wait(&condp, &mutex);
        }
        buffer = i;
        printf("Conteúdo produzido.\n");
        pthread_cond_signal(&condc);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

```c
void *consumidor(void *empty)
{
    for (int i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex);
        if (buffer == 0) {
            pthread_cond_wait(&condc, &mutex);
        }
        buffer = 0;
        printf("Conteúdo consumido.\n");
        pthread_cond_signal(&condp);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```



# Monitores

# Monitores

- Até agora, ao programador é dado um conjunto de primitivas e ele deve garantir que estas primitivas são utilizadas corretamente.
  - Exemplo: Todo Up deve ter um Down associado a ele.
  - Em programação concorrente, é conhecida como programação de baixo nível.
- Em 1974, Hoare propôs um mecanismo de alto nível para a sincronização de processos, chamado de monitor.



# Monitores

- Monitor: conjunto de procedimentos e dados agrupados por módulos.

  Cada módulo é um monitor. Os processos so podem chamar os

  procedimentos, nunca acessando diretamente os dados.

```
monitor nome
/* declaração de variáveis */
procedure p0;
...
end p0;
end monitor;
```



# Monitores

- Não pode haver dois processos ativos dentro do monitor, simultaneamente.
- Um monitor normalmente é uma construção de linguagem de programação.
- Assim, o compilador desta linguagem, ao reconhecer que é uma chamada a monitor e não uma chamada a um procedimento comum, deve garantir que o processo só executa este procedimento se não houver mais ninguém executando.



# Monitores

- Quem se preocupa em garantir a exclusão mútua é o compilador e não o programador.
- A única **tarefa do programador é identificar as seções críticas** e colocá-las dentro do monitor.
- Para implementar a exclusão mútua dentro de um monitor, o compilador usa geralmente estruturas de baixo nível, como por exemplo, semáforos.



# Monitores

- Problema: o conceito de monitor deve estar contido na linguagem de programação e a maioria das linguagens não suporta esse conceito.



# Problema do Produtor-Consumidor - Monitores

```
monitor produtor-consumidor
condicao cheio, vazio;
int cont = 0;

procedure insere()
{
    if (cont == N) wait(cheio);
    insere_item();
    cont++;
    if (cont == 1)
signal(vazio);
}
```

```
/* continuação  */

procedure remove()
{
    if (cont == 0) wait(vazio);
    remove_item();
    cont--;
    if (cont == N - 1) signal(cheio);
}
end monitor;
```



# Problema do Produtor-Consumidor - Monitores

```
produtor()
{
    while (TRUE) {
        produz_item(&item);
        produtor-consumidor.insere();
    }
}
```

```
consumidor ()
{
    while (TRUE) {
        produtor-consumidor.remove();
        consome_item(item);
    }
}
```



# Monitores

- Java é uma linguagem que dá suporte a monitores.
  - Por ser orientada a objetos, permite que *threads* de usuários e rotinas sejam agrupadas em classes.
  - Palavra-chave **syncronized** garante apenas uma *thread* por vez em uma determinada região (não usada só para regiões críticas)
- Apesar de solucionar o problema da exclusão mútua, essa solução não pode ser aplicada (assim como as outras apresentadas) em sistemas com memória distribuída.