

Processos, Threads e Comunicação

EMB5632 - Sistemas Operacionais

Professor: Lucas Leandro Nesi (lucas.nesi@ufsc.br)

Semestre 2024/01

Universidade Federal de Santa Catarina – Departamento de Engenharias da Mobilidade

Sistemas Operacionais

- Camada de abstração entre o *hardware* e o *software*
 - Particularidades de cada dispositivo são abstraídas (HD/SSD Seagate/Western Digital, Memória Kingston/Corsair, Processador Intel/AMD ou de gerações diferentes).
- Gerenciamento dos recursos
 - Decisão de que aplicação vai utilizar (ou quanto) a CPU/memória/rede. . .
 - Mesmo em um processador single core é possível executar n programas

Abstração da execução

- Informações comuns entre processos (identificador)
- Interface comum para manipulação (enviar sinal)

Pseudo paralelismo e Multiprogramação

- Vários programas estão carregados na memória
- Divisão temporal da CPU entre processos/threads

Isolamento

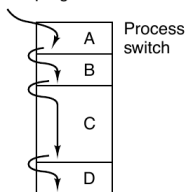
- Cada processo tem sua região de memória
- Os processos não precisam saber da existência dos outros
 - Não é uma necessidade, mas dependendo o sistema é importante projetar todos os processos conjuntamente

Um processo é um programa em execução

- Cada processo tem um contador de programa, registradores, memória
- Na visão de um processo a CPU é exclusivamente dele, o processo não sabe que outros processos estão disputando os recursos
 - A memória também é individual e virtual, o processo não tem acesso direto à memória física

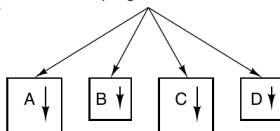
Execução de múltiplos processos em uma única CPU

One program counter

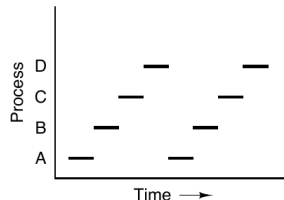


(a)

Four program counters



(b)



(c)

Eventos de criação de processos

- Na inicialização do sistema (init/systemd/SCM)
- Outro processo executou uma chamada de sistema para criação
- Solicitação do usuário
- batch jobs: fila/agendamento de tarefas

POSIX fork() – duplica o processo

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main(){
5     int var = 1;
6     pid_t PID = fork();
7     if(PID == 0){
8         var = 2;
9     }
10    printf("Valor de PID: %ld - Valor de var: %ld \n", PID, var);
11    sleep(300);
12    return 0;
13 }
```

Razoes para término do process

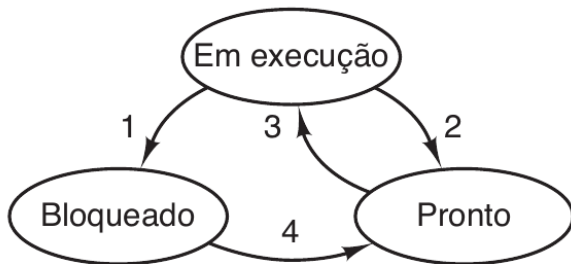
- Saída voluntaria (sucesso ou erro)
- Erro fatal
- Morto por outro processo

Hierarquias de processos

- Relação entre processos pai e filhos

Linux: organizados em uma árvore de processos

PIDΔ	TIME+	CPU%	MEM%	Command
1	0:02.93	0.0	0.1	/sbin/init
484	0:00.63	0.0	0.1	└─ /lib/systemd/systemd-journald
507	0:00.26	0.0	0.0	└─ /lib/systemd/systemd-udevd
701	0:00.08	0.0	0.0	└─ /lib/systemd/systemd-timesyncd
704	0:00.00	0.0	0.0	└─ /lib/systemd/systemd-timesyncd
728	0:00.44	0.0	0.1	└─ /usr/libexec/accounts-daemon
778	0:00.19	0.0	0.1	└─ /usr/libexec/accounts-daemon
882	0:00.00	0.0	0.1	└─ /usr/libexec/accounts-daemon
730	0:00.00	0.0	0.0	└─ /usr/sbin/anacron -d -q -s



1. O processo é bloqueado aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

Implementação de processos

Tabela de processos

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

Exercício 1

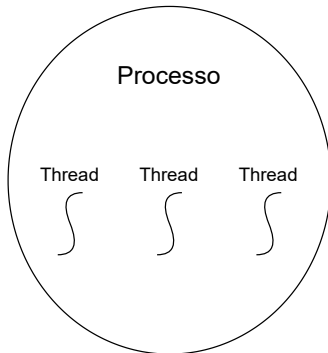
1. Crie seu próprio terminal: seu terminal aceita uma string como entrada. Se for a string "exit" termine o programa. Para qualquer outra string, considere que é um programa que o usuário deseja executar. Verifique se o programa existe (fopen retorna nulo se o arquivo não existe), caso não exista, imprima um erro e peça ao usuário um novo comando. Caso exista, você deve executar este programa. Para realizar isso com a interface posix, é necessário abrir um novo processo com `fork()`, e mudar somente o conteúdo do processo filho com `execle`. `execle` recebe o caminho do executável, argumentos, e ambiente (utilize o `envp` do seu `main`). Faça o seu terminal (processo original) esperar o processo filho com `waitpid(pid, &status, 0)`; sendo `pid` o retorno do `fork`, e `status` um novo inteiro. Após a execução do comando, peça um novo comando ao usuário.
 - Utilize o seguinte `main`: `int main(int argc, char *argv[], char *envp[])`

Sistemas complexos com várias atividades

- E se meu processo precisa realizar diversas operações simultaneamente?
- Diferentes processos tem memórias independentes, e se for interessante múltiplos fluxos de execução com o mesmo código/memória?
- Mudar o contexto de processos pode ser lento (chaveamento, carregar instruções, alterar memória), existe uma maneira mais eficiente?

Thread: Fluxo de execução de um processo

- Um processo tem uma ou mais threads
- Também conhecido como um processo "leve"
- Possui sua pilha, contexto de execução
- Compartilha a memória com as outras threads do processo



Threads – Informações compartilhadas

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

Nível de usuário

- Um processo utiliza uma biblioteca de threads
- O SO não sabe nada sobre as threads
- Leve, mas se uma thread bloquear, o processo é bloqueado, impossibilitando a execução de outras threads

Nível do SO

- O sistema operacional que gerencia as threads
- Se uma thread é bloqueada, outras threads do mesmo processo ainda podem executar

- `pthread_create` : cria uma thread
 - `struct pthread_t`
 - Atributos de pthread, NULL para padrão
 - Função `void * (*) (void *)`
 - Ponteiro para parâmetros da função encapsulados numa estrutura
- `pthread_join` : espera uma thread
 - `struct pthread_t`
 - Ponteiro para colocar retorno (NULL é válido)
- Compilar com `-lpthread`

Exemplo 1 - Olá com pthreads

```
1 void* func(void *arg) {
2     printf("Olá da thread %d\n", *(int*)arg);
3 }
4
5 int main() {
6     pthread_t threads[2];
7
8     int arg1 = 1, arg2 = 2;
9
10    pthread_create(&threads[0], NULL, &func, &
11    arg1);
12
13    pthread_create(&threads[1], NULL, &func, &
14    arg2);
15
16    pthread_join(threads[0], NULL);
17    pthread_join(threads[1], NULL);
18 }
```

Exemplo 1 - Olá com pthreads

```
1 void* func(void *arg) {  
2     printf("Olá da thread %d\n", *(int*)arg);  
3 }  
4  
5 int main() {  
6     pthread_t threads[2];  
7  
8     int arg1 = 1, arg2 = 2;  
9  
10    pthread_create(&threads[0], NULL, &func, &  
11    arg1);  
12    pthread_create(&threads[1], NULL, &func, &  
13    arg2);  
14  
15    pthread_join(threads[0], NULL);  
16    pthread_join(threads[1], NULL);  
17 }
```

Exemplo 1 - Olá com pthreads

```
1 void* func(void *arg) {
2     printf("Olá da thread %d\n", *(int*)arg);
3 }
4
5 int main() {
6     pthread_t threads[2];
7
8     int arg1 = 1, arg2 = 2;
9
10    pthread_create(&threads[0], NULL, &func, &
11    arg1);
12
13    pthread_create(&threads[1], NULL, &func, &
14    arg2);
15
16    pthread_join(threads[0], NULL);
17    pthread_join(threads[1], NULL);
18 }
```

Exemplo 1 - Olá com pthreads

```
1 void* func(void *arg) {
2     printf("Olá da thread %d\n", *(int*)arg);
3 }
4
5 int main() {
6     pthread_t threads[2];
7
8     int arg1 = 1, arg2 = 2;
9
10    pthread_create(&threads[0], NULL, &func, &
11    arg1);
12
13    pthread_create(&threads[1], NULL, &func, &
14    arg2);
15
16    pthread_join(threads[0], NULL);
17    pthread_join(threads[1], NULL);
18 }
```

Exemplo 1 - Olá com pthreads

```
1 void* func(void *arg) {
2     printf("Olá da thread %d\n", *(int*)arg);
3 }
4
5 int main() {
6     pthread_t threads[2];
7
8     int arg1 = 1, arg2 = 2;
9
10    pthread_create(&threads[0], NULL, &func, &
11    arg1);
12    pthread_create(&threads[1], NULL, &func, &
13    arg2);
14
15    pthread_join(threads[0], NULL);
16    pthread_join(threads[1], NULL);
17 }
```

Exemplo 2 - Operação matemática

- Verificar arquivo `mul.c` e `mul_pt.c` no moodle.

Exercício 2 - Calculando PI

Considere o código abaixo que calcula PI com a série de Leibniz¹ usando 5 bilhões de termos. Utilize pthreads para acelerar sua execução:

- **Dica:** Mantenha a função `calculo_pi`, apenas passe como parâmetro um novo intervalo para o `for`. Mantenha a variável `sum` local na função e salve o resultado na struct de parâmetros. Realize os `joins` e some o retorno de todas as funções.

¹https://pt.wikipedia.org/wiki/F%C3%B3rmula_de_Leibniz_para_%CF%80

Exercício 2 - Calculando PI

```
1 double calculo_pi(long int n){
2     double sum = 0;
3     for(long int i=0; i<n; i++){
4         double sinal = i%2==0 ? 1.0 : -1.0;
5         sum += sinal/(2.0 * i + 1.0);
6     }
7     return(sum*4);
8 }
9
10 int main(){
11     double pi = calculo_pi(5000000000L);
12     printf("Valor de PI: %.20lf\n", pi);
13 }
```

Comunicações entre processos

Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

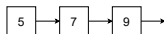
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11

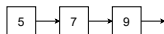
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

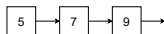
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

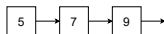
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

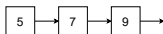
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

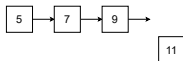
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

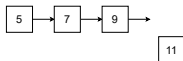
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 

- Processo B insere 12 - struct elemento* atual:

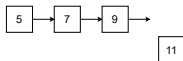
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →

- Processo B insere 12 - struct elemento* atual:  →

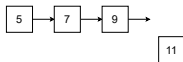
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →

- Processo B insere 12 - struct elemento* atual:  →

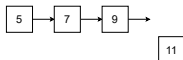
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →

- Processo B insere 12 - struct elemento* atual:  →

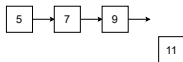
Condição de corrida – Estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →

- Processo B insere 12 - struct elemento* atual:  →

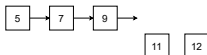
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 
- Processo B insere 12 - struct elemento* atual: 

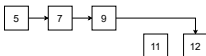
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →
- Processo B insere 12 - struct elemento* atual:  →

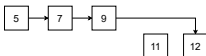
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 
- Processo B insere 12 - struct elemento* atual: 

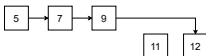
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual:  →
- Processo B insere 12 - struct elemento* atual:  →

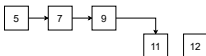
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 
- Processo B insere 12 - struct elemento* atual: 

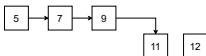
Condição de corrida – Estrutura de dados compartilhada



Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 void inserir(struct elemento* raiz, void* dado){  
2     struct elemento* atual = raiz;  
3     while(atual->next!=NULL) atual = atual->next;  
4     struct elemento* novo_elemento = malloc(...);  
5     novo->dado = dado;  
6     atual->next = novo_elemento;  
7 }
```

- Estado da Lista:



- Processo A insere 11 - struct elemento* atual: 
- Processo B insere 12 - struct elemento* atual: 

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 0

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 0
- Processo A: v[i]: 40

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 0
- Processo A: v[i]: 40

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 40
- Processo A: v[i]: 40

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 40
- Processo A: v[i]: 50

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 40
- Processo A: v[i]: 50

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 40
- Processo A: v[i]: 50
- Processo B: v[i]: 60

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 40
- Processo A: v[i]: 50
- Processo B: v[i]: 60

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 60
- Processo A: v[i]: 50
- Processo B: v[i]: 60

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 60
- Processo A: v[i]: 50

Condição de corrida – Encontrar o maior número

Encontrar o maior elemento de um vetor

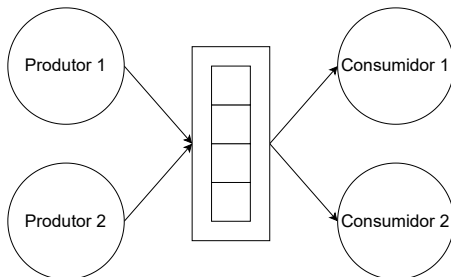
- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

- Maior atual: 60 → 50
- Processo A: v[i]: 50

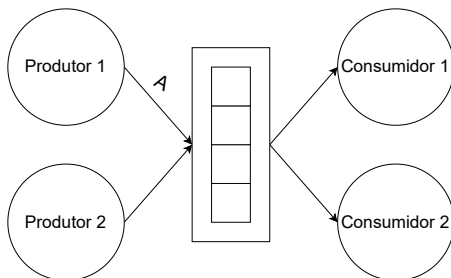
Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



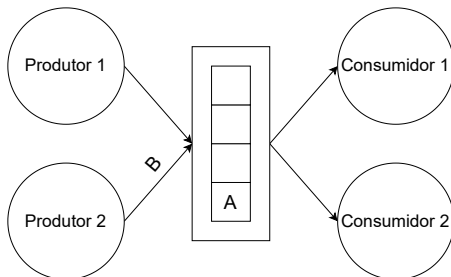
Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



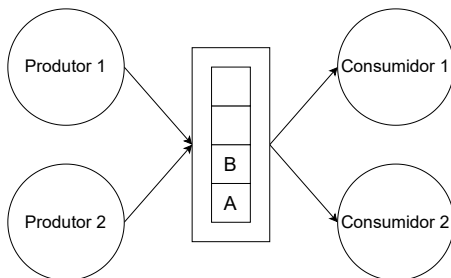
Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



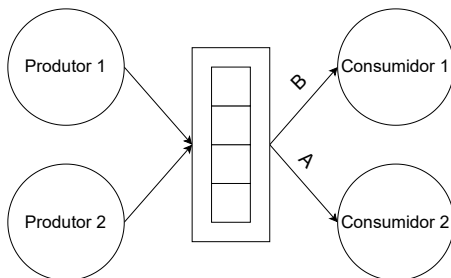
Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



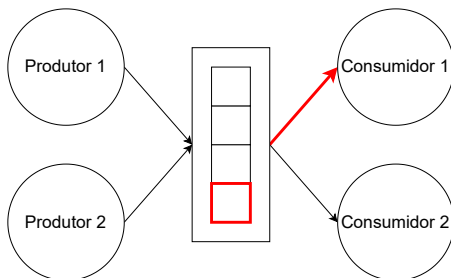
Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



Condição de corrida – Problema Produtor - Consumidor

Existe processos que produzem dados, e processo que consomem



Qual o problema comum?

- Memória/recurso compartilhado sendo acessado sem controle
- O resultado é imprevisível e depende de quem acessa o dado primeiro (**corrida**)

Como podemos solucionar este problema?

- Área de código que acessa o recurso é chamada de **região crítica**
- O acesso a região crítica deve ter uma exclusão mútua entre processos

Propriedades desejáveis

- Apenas um processo acessa a região por vez
- Estratégia deve funcionar independente de hardware, escalonamento
- Um processo não pode esperar para sempre
- Um processo não pode ficar bloqueado se a região crítica está livre

Os processos devem se comunicar para se coordenarem

- Comunicação com variáveis compartilhadas
- Comunicação com troca de mensagens

Memória compartilhada

- Processos: com auxílio do sistema operacional
- Thread: a memória já é compartilhada

Funcionamento

- A variável descreve o estado do compartilhamento de um recurso
- Funções atômicas são utilizadas sobre as variáveis

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     while(trava == 1) {  
3         // Espera ocupada  
4     }  
5     trava = 1;  
6 }
```

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     while(trava == 1) {  
3         // Espera ocupada  
4     }  
5     trava = 1;  
6 }
```

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     while(trava == 1) {  
3         // Espera ocupada  
4     }  
5     trava = 1;  
6 }
```

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     while(trava == 1) {  
3         // Espera ocupada  
4     }  
5     trava = 1;  
6 }
```

- Instruções atômicas especiais (XCHG / TSL)
 - XCHG: troca o conteúdo atômicamente de um registrador e um local de memória
 - TSL: cópia o conteúdo de um local de memória para um registrador e armazena 1

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     local = 1  
3     while(local == 1){  
4         XCHG trava local  
5     }  
6 }
```

Espera ocupada

Uso de uma variável para informar que a região crítica está travada

Variável de trava

- Se 1 a região já está sendo executada
- Se 0 a região está livre

```
1 void entrar_regiao() {  
2     local = 1  
3     while(local == 1) {  
4         XCHG trava local  
5     }  
6 }
```

Desvantagens:

- Gasto de ciclos da CPU
- Possível deadlock entre processos

Se um processo sabe que deve esperar outro, ele pode simplesmente bloquear

- Ajuda do sistema operacional
- O processo não é escalonado, não consumindo tempo de CPU

Operação dormir

- Bloqueia o processo, colocando na fila de espera

Operação acordar

- Libera um processo alvo ou alguém da fila de espera

Estratégia para garantir Exclusão Mútua

- Utiliza uma variável de trava (0 ou 1)

Duas operações atômicas: trancar e destrancar

- trancar: tenta trancar a variável
 - Se não estiver trancada: tranque e continue
 - Se estiver trancada: bloqueie, sendo adicionado numa lista de espera
- destrancar: destranca a variável que foi bloqueada pelo processo
 - Se existe um processo bloqueado, desbloqueie

Mutex – Exemplo estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 mutex regioao_critica_lista;
2 void inserir_lista(struct elemento* raiz, void*
   novo_dado) {
3     trancar(regiao_critica_lista);
4     struct elemento* atual = raiz;
5     while(atual->proximo!=NULL) atual = atual->
       proximo;
6     struct elemento* novo_elemento = malloc(sizeof
       (struct elemento));
7     novo->dado = novo_dado;
8     atual->proximo = novo_elemento;
9     destrancar(regiao_critica_lista);
10 }
```

Mutex – Exemplo estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 mutex regioao_critica_lista;
2 void inserir_lista(struct elemento* raiz, void*
   novo_dado) {
3     trancar(regiao_critica_lista);
4     struct elemento* atual = raiz;
5     while(atual->proximo!=NULL) atual = atual->
       proximo;
6     struct elemento* novo_elemento = malloc(sizeof
       (struct elemento));
7     novo->dado = novo_dado;
8     atual->proximo = novo_elemento;
9     destrancar(regiao_critica_lista);
10 }
```

Mutex – Exemplo estrutura de dados compartilhada

Uma lista é compartilhada por dois processos

- Processo A e Processo B

```
1 mutex regioao_critica_lista;  
2 void inserir_lista(struct elemento* raiz, void*  
   novo_dado) {  
3     trancar(regiao_critica_lista);  
4     struct elemento* atual = raiz;  
5     while(atual->proximo!=NULL) atual = atual->  
       proximo;  
6     struct elemento* novo_elemento = malloc(sizeof  
       (struct elemento));  
7     novo->dado = novo_dado;  
8     atual->proximo = novo_elemento;  
9     destrancar(regiao_critica_lista);  
10 }
```

- `pthread_mutex_t` : estrutura para guardar o mutex
- `mutex_lock (&pthread_mutex_t)` : trancar
- `mutex_unlock (&pthread_mutex_t)` : destrancar

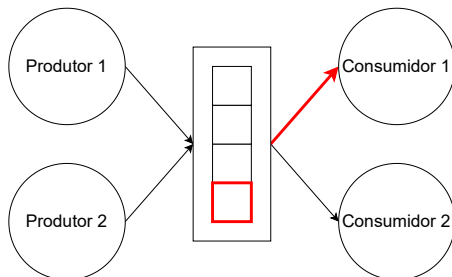
Considere o exercício 2

- Utilize uma variável global para armazenar a soma da serie
 - Execute o código várias vezes, o que acontece?
- Defina a região crítica
- Proteja a região critica com mutexes

Exemplo – Consumidor - Produtor

Problema Consumidor - Produtor

Existe processos que produzem dados, e processo que consomem



Estratégia para garantir exclusão mútua com uma variável condicional

- Utiliza uma variável contadora
- Diferentes implementações podem definir o comportamento das seguintes funções:

Funções UP e DOWN

- DOWN: Tenta decrementar a variável
 - Se maior que 0, continua
 - Se igual (ou menor que) 0, bloqueie, sendo adicionado numa lista de espera
 - Decrementa a variável contadora
- UP: Incrementa a variável contadora
 - Se um processo estiver bloqueado, desbloqueie alguém da lista de espera

Atenção: Algumas implementações permitem contador negativo 36/50

A ordem de bloqueio/desbloqueio pode ser diferente

Problema Consumidor - Produtor

Existem processos que produzem dados, e processo que consomem

```
1 void* lista;  
2 produtor() {  
3     dado <- produzir()  
4     inserir(lista, dado)  
5 }  
6 consumidor() {  
7     dado <- remover(lista, dado)  
8     consumir(dado)  
9 }
```


Problema Consumidor - Produtor

Existem processos que produzem dados, e processo que consomem

```
1 void* lista;  
2 produtor() {  
3     dado <- produzir()  
4     inserir(lista, dado)  
5 }  
6 consumidor() {  
7     dado <- remover(lista, dado)  
8     consumir(dado)  
9 }
```

Semáforo – Exemplo Consumidor - Produtor

Problema Consumidor - Produtor

Existe processos que produzem dados, e processo que consomem

```
1 void* lista;
2 semaforo producao = N;
3 semaforo consumo = 0;
4 produtor() {
5     dado <- produzir()
6     down(producao)
7     inserir(lista, dado)
8     up(consumo)
9 }
10 consumidor() {
11     down(consumo)
12     dado <- remover(lista, dado)
13     up(producao)
14     consumir(dado)
15 }
```

Semáforo – Exemplo Consumidor - Produtor

Problema Consumidor - Produtor

Existem processos que produzem dados, e processo que consomem

```
1 void* lista;
2 semaforo producao = N;
3 semaforo consumo = 0;
4 produtor() {
5     dado <- produzir()
6     down(producao)
7     inserir(lista, dado)
8     up(consumo)
9 }
10 consumidor() {
11     down(consumo)
12     dado <- remover(lista, dado)
13     up(producao)
14     consumir(dado)
15 }
```

Semáforo – Exemplo Consumidor - Produtor

Problema Consumidor - Produtor

Existem processos que produzem dados, e processo que consomem

```
1 void* lista;
2 semaforo producao = N;
3 semaforo consumo = 0;
4 produtor() {
5     dado <- produzir()
6     down(producao)
7     inserir(lista, dado)
8     up(consumo)
9 }
10 consumidor() {
11     down(consumo)
12     dado <- remover(lista, dado)
13     up(producao)
14     consumir(dado)
15 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```


Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 0
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 1
- Fila de espera Consumo: Thread 1

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 1
- Fila de espera Consumo:

```
1 produtor () {  
2   dado <- produzir()  
3   down(producao)  
4   inserir(lista, dado)  
5   up(consumo)  
6 }  
7 consumidor () {  
8   down(consumo)  
9   dado <- remover(lista, dado)  
10  up(producao)  
11  consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 1
- Fila de espera Consumo:

```
1 produtor () {  
2   dado <- produzir()  
3   down(producao)  
4   inserir(lista, dado)  
5   up(consumo)  
6 }  
7 consumidor () {  
8   down(consumo)  
9   dado <- remover(lista, dado)  
10  up(producao)  
11  consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor () {  
2   dado <- produzir()  
3   down(producao)  
4   inserir(lista, dado)  
5   up(consumo)  
6 }  
7 consumidor () {  
8   down(consumo)  
9   dado <- remover(lista, dado)  
10  up(producao)  
11  consumir(dado)  
12 }
```


Semáforo – Exemplo Consumidor - Produtor

- Producao = 1
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor() {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor() {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor() {  
2   dado <- produzir()  
3   down(producao)  
4   inserir(lista, dado)  
5   up(consumo)  
6 }  
7 consumidor() {  
8   down(consumo)  
9   dado <- remover(lista, dado)  
10  up(producao)  
11  consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor() {  
2   dado <- produzir()  
3   down(producao)  
4   inserir(lista, dado)  
5   up(consumo)  
6 }  
7 consumidor() {  
8   down(consumo)  
9   dado <- remover(lista, dado)  
10  up(producao)  
11  consumir(dado)  
12 }
```

Semáforo – Exemplo Consumidor - Produtor

- Producao = 2
- Consumo = 0
- Fila de espera Consumo:

```
1 produtor () {  
2     dado <- produzir()  
3     down(producao)  
4     inserir(lista, dado)  
5     up(consumo)  
6 }  
7 consumidor () {  
8     down(consumo)  
9     dado <- remover(lista, dado)  
10    up(producao)  
11    consumir(dado)  
12 }
```

Construção de uma linguagem de programação

- Classe, módulo, ou pacote que todas suas rotinas são protegidas
- Apenas um processo pode estar executando uma rotina do monitor
- Utilização de variáveis de condição

Exemplo: Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

Exemplo: Encontrar o maior número

Encontrar o maior elemento de um vetor

- Processo A e Processo B
 - Variável maior compartilhada

```
1 for(int i = 0; i < size; i++){  
2     if(v[i] > maior){  
3         maior = v[i];  
4     }  
5 }
```

Mensagens

- Qualquer tipo de dado
- Não necessita de memória compartilhada
- Facilmente abstraída para recursos distribuídos

Funcionamento

- Algum método que defina a origem, a mensagem, e o destinatário
- A transferência é transparente ao processo

Comunicação unidirecional entre processos

- Um processo escreve dados, enquanto o outro processo consome
- Os processos devem saber previamente da utilização da pipe
- Buffer especial do sistema operacional
- Exemplo: shell: `cat arquivo | gzip`

Named pipes

- Arquivo especial nomeado para servir de buffer da pipe
- Ambos os processos acessam a pipe de mesmo nome

Comunicação bidirecional

- Ambos os processos podem enviar ou receber mensagens
- Métodos:
 - enviar(destino, mensagem)
 - receber(origem, mensagem)
- As operações devem ser as opostas
 - A mesma ordem de enviar e receber na origem, deve ter seu par de enviar e receber no destino

Especificação MPI (Message Passing Interface)

- Diversas implementações (bibliotecas)
- Permite chamadas assíncronas

Exemplo: Encontrar o maior número

Encontrar o maior elemento de um vetor

Processo A

```
1 for(int i = 0; i <
    size/2; i++){
2     if(v[i] > maiorA){
3         maiorA = v[i];
4     }
5 }
6 receber(B, maiorB);
7 maior = max(maiorA,
    maiorB);
```

Processo B

```
1 for(int i = size; i <
    size; i++){
2     if(v[i] > maiorB){
3         maiorB = v[i];
4     }
5 }
6 enviar(A, maiorB);
7 ;
```

Exemplo: Encontrar o maior número

Encontrar o maior elemento de um vetor

Processo A

```
1 for(int i = 0; i <
    size/2; i++){
2     if(v[i] > maiorA){
3         maiorA = v[i];
4     }
5 }
6 receber(B, maiorB);
7 maior = max(maiorA,
    maiorB);
```

Processo B

```
1 for(int i = size; i <
    size; i++){
2     if(v[i] > maiorB){
3         maiorB = v[i];
4     }
5 }
6 enviar(A, maiorB);
7 ;
```

Exemplo: Encontrar o maior número

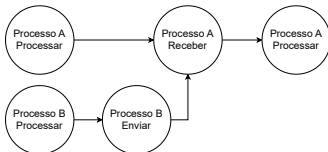
Encontrar o maior elemento de um vetor

Processo A

```
1 for(int i = 0; i <
    size/2; i++){
2     if(v[i] > maiorA){
3         maiorA = v[i];
4     }
5 }
6 receber(B, maiorB);
7 maior = max(maiorA,
    maiorB);
```

Processo B

```
1 for(int i = size; i <
    size; i++){
2     if(v[i] > maiorB){
3         maiorB = v[i];
4     }
5 }
6 enviar(A, maiorB);
7 ;
```



Conceitos

- Condição de corrida
- Região crítica
- Exclusão mútua

Estratégias de comunicação

- Espera ocupada
- Mutex
- Semáforo
- Pipes
- Troca de mensagens: enviar, receber

Exercício 4 - Lista Thread Safe

Implemente as seguintes funções de uma lista (double)

- Inicializar com tamanho n
- Inserir no início
- Inserir no fim
- Remove no início (e retorne)
- Remove no fim (e retorne)

Proteja as regiões críticas da sua lista

- Utilizando mutex posix
- Utilize sua lista com múltiplas threads
- Verifique utilizando `-fsanitize=thread`

Escreva um pequeno relatório

- Apresentado a implementação e decisões
- Como foi testado (quantidade de threads / `fsanitize`)

Exercício 5 - Produtor / Consumidor

Considere as seguintes funções de produzir e consumir:

```
1 #include <unistd.h>
2 int elem_id=0;
3
4 void produzir(int thread_id, lista l){
5     printf("[%d]Produzindo %d...", thread_id,
6         elem_id);
7     usleep(333000);
8     l.insere_fim(elem_id++);
9     printf("[%d]Produzido", thread_id);
10 }
11 void consumir(int thread_id, lista l){
12     printf("[%d]Consumindo...", thread_id);
13     v = l.removeApresentado c_inicio();
14     usleep(500000);
15     printf("[%d]Consumido %d", thread_id, v);
16 }
```


Exercício 5 - Produtor / Consumidor

- Utilize semáforos POSIX ² (`sem_post` / `sem_wait`) para proteger a utilização da lista (produzir somente se a lista tem espaço, consumir somente se a fila tem elementos)
- Implemente um programa que inicialize a lista com no máximo *i* elementos, *n* produtores e *m* consumidores, e produza no máximo *x* itens.

Escreva um pequeno relatório

- Apresentado a implementação e decisões
- Mostrando a saída com fila de tamanho 4, 2 produtores, 2 consumidores, 20 itens, e tempo de execução.
- Mostrando a saída com fila de tamanho 4, 2 produtores, 3 consumidores, 20 itens, e tempo de execução.

²https://linux.die.net/man/7/sem_overview

Bibliografia Básica

- Tanenbaum, A. S., Sistemas Operacionais Modernos, 3a. edição, Prentice-Hall do Brasil, 2010. (Capítulo 2).
- Silberschatz, A., Galvin, P. B., Gagne, Greg, Operating Systems Concepts, 9th ed, Elsevier, 2013. (Capítulos 3 e 6).
- Stallings, W., Operating Systems: Internals and Design Principles – Third Edition, Prentice Hall, 1998. (Capítulo 5).

Bibliografia Complementar

- Oliveira, RS de, Carissimi, A. da S., Toscani, SS, Sistemas operacionais, 4ª edição, Bookman, 2010.
- Thomas W. Doeppner. Operating Systems in Depth. John Wiley & Sons, Inc. 2011.