Processos concorrentes

Sistemas Operacionais
Gerência de processos

Agenda

* Revisão
* Introdução a concorrência
* Mecanismos de controle de concorrência
* Problemas clássicos de concorrência

Revisão

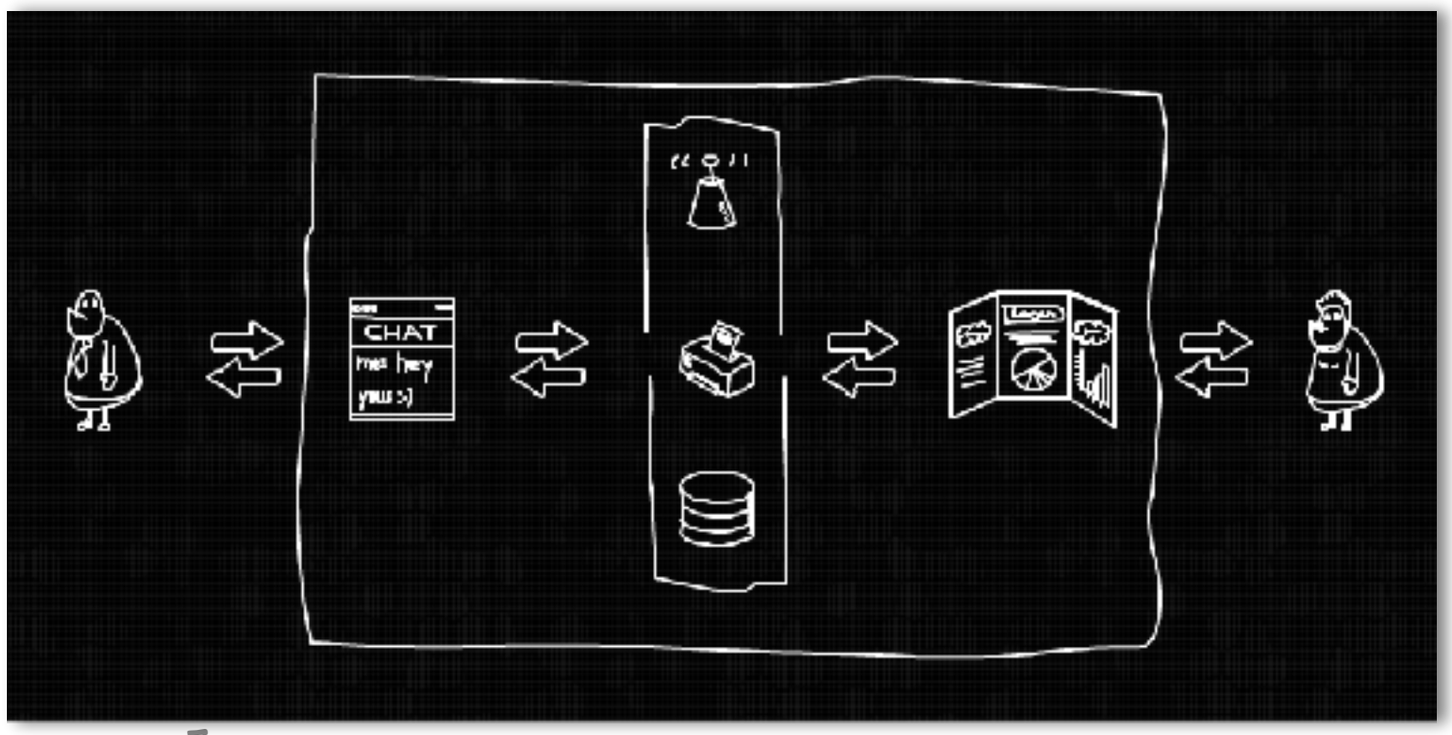
* SO Multiprogramado permite diversos processos "paralelos"
* Multiprocessado permite diversos processos paralelos e simultâneos
* Os processos usam recursos do sistema
  * hardware : impressora, interface de rede, disco
  * software : variáveis (espaço em memória)

Agenda

* Revisão
* Introdução a concorrência
* Mecanismos de controle de concorrência
* Problemas clássicos de concorrência

Introdução concorrência

* Concorrência (condições de disputa ou race conditions)
  * Vários processos
  * Compartilhamento de recursos

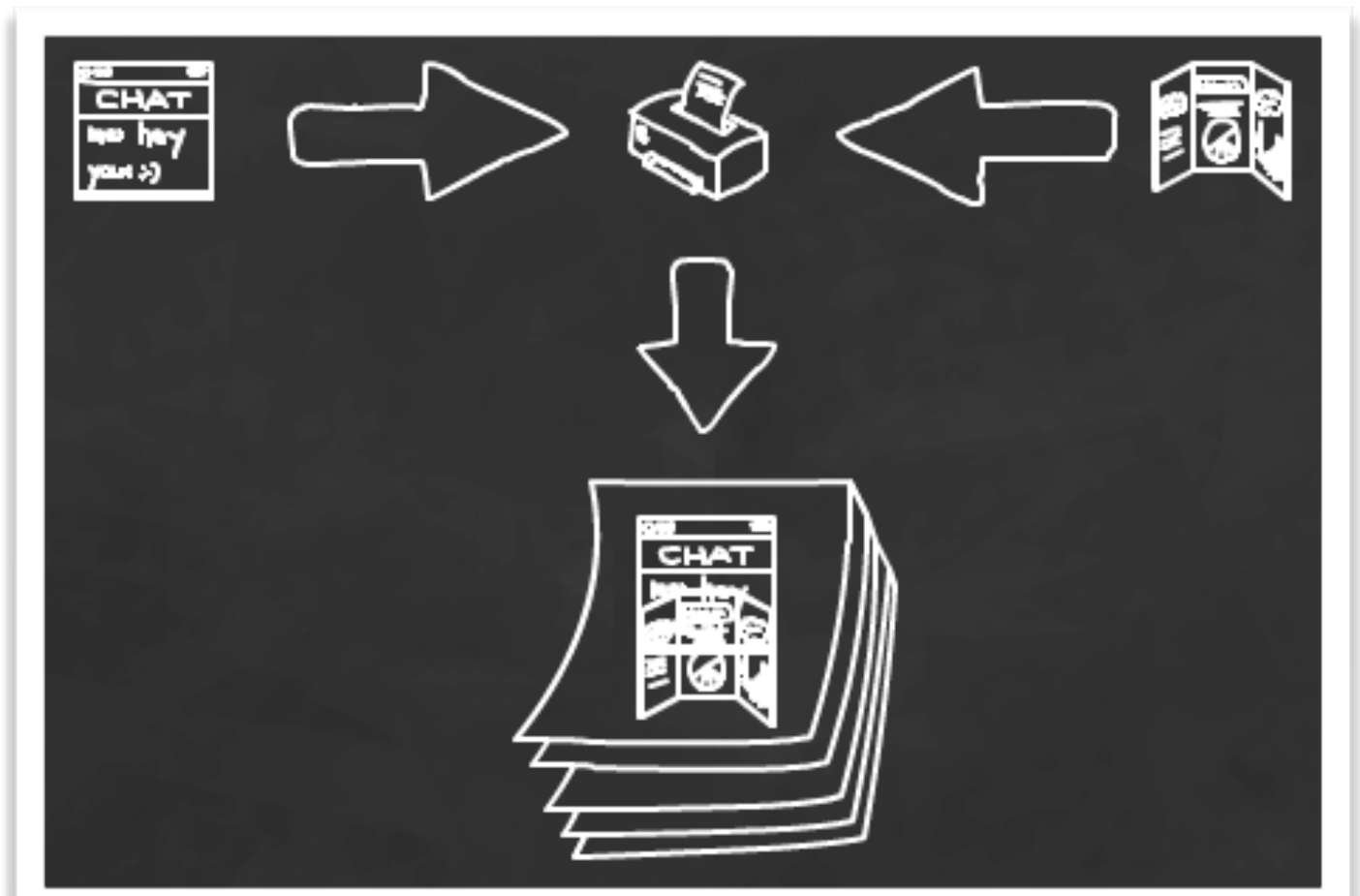


Concerrência

* 3 aspectos importantes na concorrência
  * Como um processo envia e recebe informação para/de outro processo
  * Como garantir que processos não invadam espaços uns dos outros
  * Qual a dependência entre processos (sequência adequada de execução)

Introdução inconsistência

* Acesso concorrente aos recursos compartilhados pode resultar em inconsistência

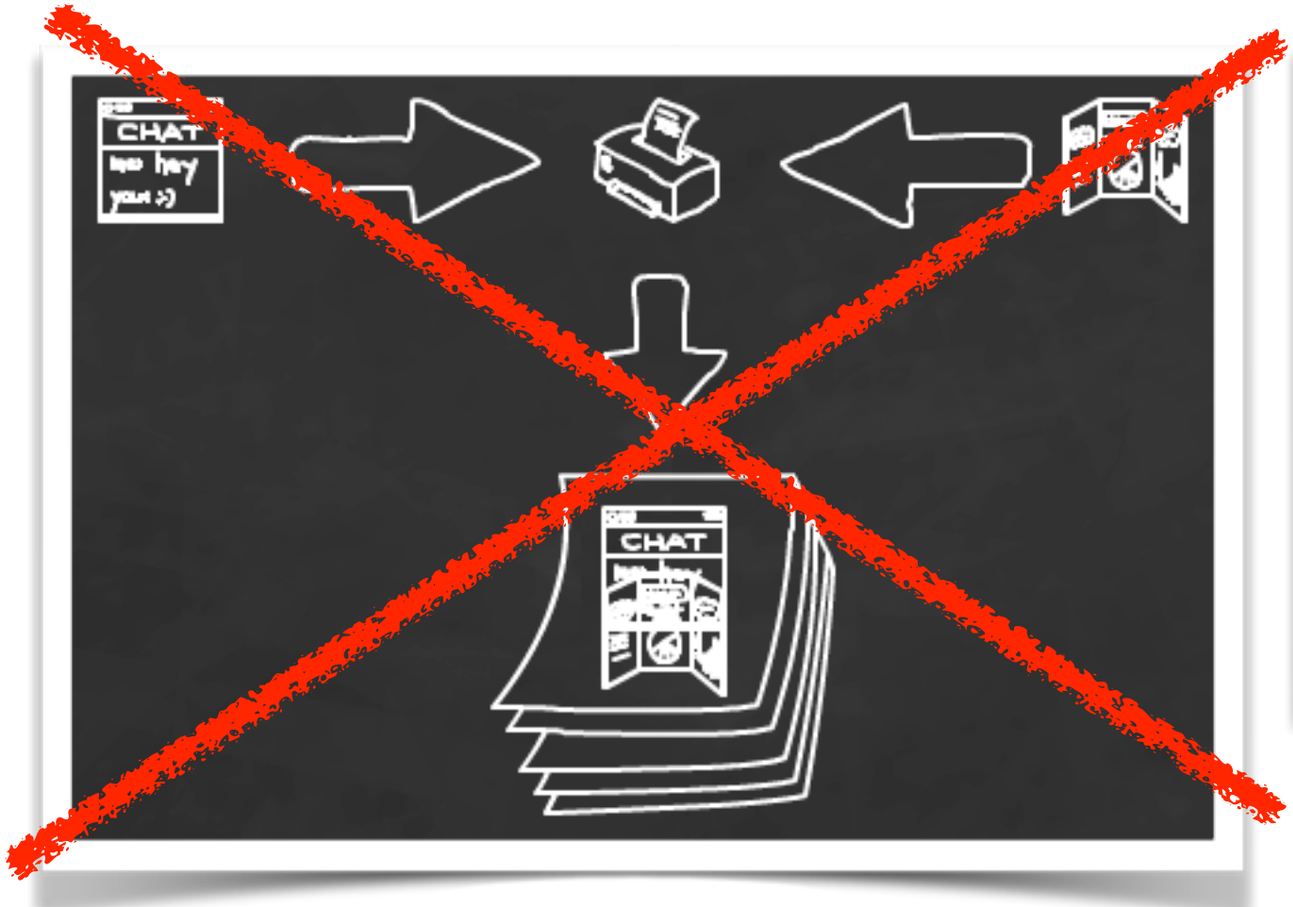


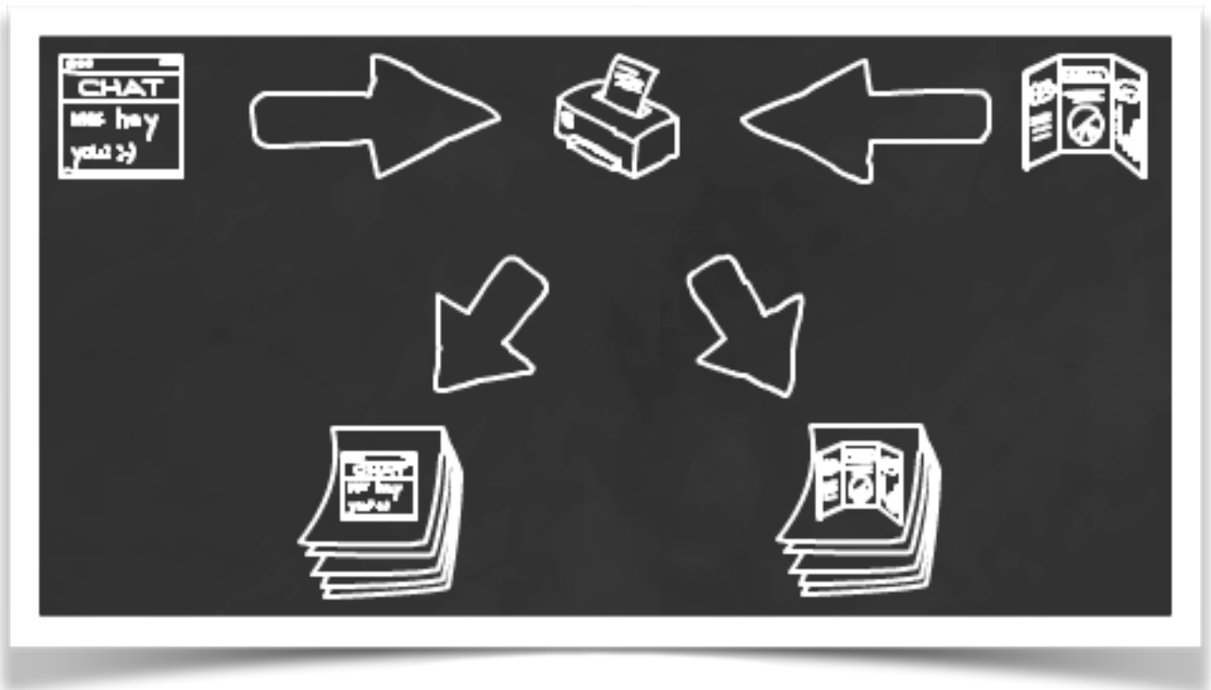
Exemplos de inconsistência

* Implicam em perdas financeiras
  * Transações bancárias
  * Bolsas de valores
* Implicam em perdas humanas
  * Atendimento em UTI
  * Sistema de controle aéreo, ferroviário, rodoviário
* Outros
  * Repositório de código (CVS, SVN, Git...)

Introdução mecanismos

* Manutenção da consistência dos recursos (dados) requer mecanismos para garantir a execução ordenada dos processos concorrentes

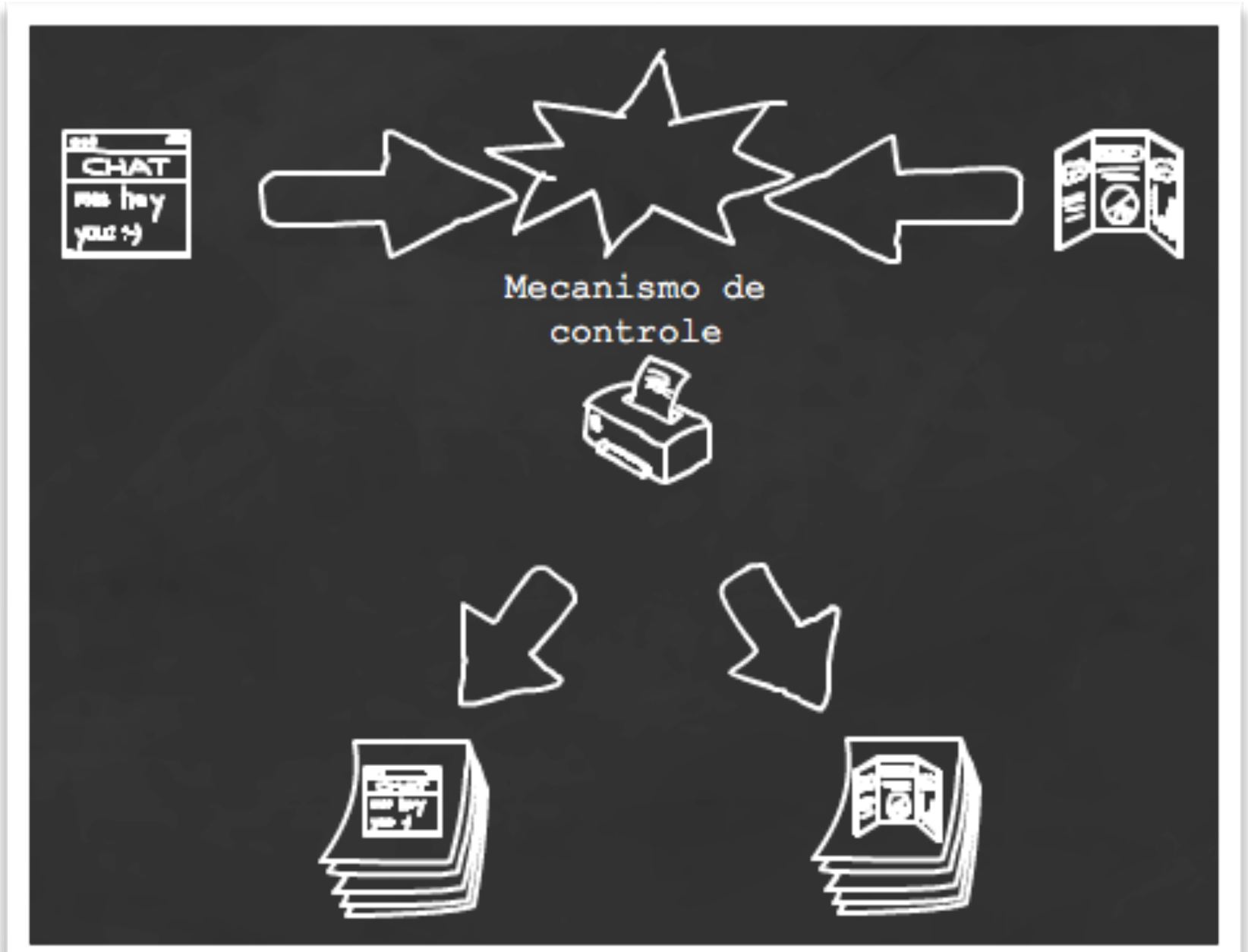




Mecanismo de controle

* Concorrência
  * Vários processos acessando um determinado recurso
* Mecanismo simplificado
  * Bloqueio do acesso ao recurso, permitindo apenas um processo por vez

Mecanismo de controle



Onde fazer o bloqueio

* Onde?
  * A região/seção/bloco do software que realiza acesso ao recurso compartilhado
* Nomenclatura
  * região/seção crítica (critical region/section)

Condições de uma boa solução

1. Dois processos não podem estar simultaneamente em regiões críticas
2. Nenhuma restrição deve ser feita com relação à CPU
3. Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas
4. Processos não podem esperar para sempre para acessarem regiões críticas

Agenda

* Revisão
* Introdução a concorrência
* Mecanismos de controle de concorrência
* Problemas clássicos de concorrência

Mecanismos de controle de concerrência

* Alguns mecanismos de controle
  * Espera ocupada
  * Primitivas sleep/wakeup
  * Semáforos
  * Monitores
  * Passagem de mensagem
  * Barreira

Espera ocupada

* Espera Ocupada (Busy Waiting)
  * Caracterizado por uma constante checagem de algum valor (variável)
* Algumas soluções para exclusão mútua com espera ocupada:
  * Desabilitar interrupções, Variáveis de travamento (Lock), Estrita alternância (Strict Alternation), Solução de Peterson, e Instrução TSL

Espera ocupada
Desabilitar interrupções

- Processo desabilita todas as suas interrupções ao entrar na região crítica e habilita essas interrupções ao sair da região crítica
    - Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos
        - Viola condição 2
- Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado
    - Viola condição 4

Condições de uma boa solução

1. Dois processos não podem estar simultaneamente em regiões críticas
2. Nenhuma restrição deve ser feita com relação à CPU
3. Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas
4. Processos não podem esperar para sempre para acessarem regiões críticas

Espera ocupada
Variáveis de travamento

```
while (true) {
  while (lock != 0); // loop
  lock = 1;
  criticial_region();
  lock = 0;
  non_critical_region();
}
```

Processo A

```
while (true) {
  while (lock != 0); // loop
  lock = 1;
  criticial_region();
  lock = 0;
  non_critical_region();
}
```

Processo B

Espera ocupada
Variáveis de travamento

- Algoritmo
  - O processo que deseja utilizar uma região crítica atribuí um valor a uma variável chamada lock
    - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica
    - Se a variável está com valor 1 (um) significa que existe um processo na região crítica;
- Problema
  - Condições 1 e 4

Condições de uma boa solução

1. Dois processos não podem estar simultaneamente em regiões críticas
2. Nenhuma restrição deve ser feita com relação à CPU
3. Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas
4. Processos não podem esperar para sempre para acessarem regiões críticas

Espera ocupada
Estrita alternância

```
while (true) { // processo 0, lock 0
  while (lock != 0); // loop
  criticial_region();
  lock = 1; // libera para o processo 1
  non_critical_region();
}
```

Processo 0

```
while (true) { // processo 1, lock 1
  while (lock != 1); // loop
  criticial_region();
  lock = 0; // libera para o processo 0
  non_critical_region();
}
```

Processo 1

Espera ocupada
Estrita alternância

* Fragmentos de programa controlam o acesso às regiões críticas
* Variável **turn**, inicialmente em **0**, estabelece qual processo pode entrar na região crítica
* Problema
  * Viola a condição 3. processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas

Espera ocupada
Solução de Peterson

```
#define FALSE 0
#define TRUE  1
#define N     2 // numero de processos

int turn;       // controla de quem eh a vez
int interested[N];

void enter_region(int process) {
  int other;

  other = 1 - process;         // numero do processo oposto
  interested[process] = TRUE; // mostra que voce esta interessado
  turn = process;              // altera o valor do processo da vez
  while (turn==process) && interested[other]==TRUE); // entrada da RC
}

void leave_region(int process) { // quem esta saindo
  interested[process] = FALSE;
}
```

Espera ocupada
Solução de Peterson

* Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região
* Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica
* Ambas as soluções possuem fragmentos de programas que controlam a entrada e a saída da região crítica

Espera ocupada
Test and Set Lock - TSL

```
enter_region:
   TSL REGISTER, LOCK      | Copia lock para reg. e lock=1
   CMP REGISTER, #0        | lock valia zero?
   JNE enter_region        | Se sim, entra na região crítica,
                           | Se não, continua no laço
   RET                     | Retorna para o processo chamador

leave_region
    MOVE LOCK, #0          | lock=0
    RET                    | Retorna para o processo chamador
```

Espera ocupada
Test and Set Lock - TSL

* Utiliza registradores do hardware;
* TSL RX, LOCK
  * Lê o conteúdo de **lock** em **RX**, e armazena um valor diferente de zero (0) em **lock**
  * Esta operação é indivisível;
* Lock é acessa em todos os processos
  * Se **lock == 0**, então região crítica "**liberada**".
  * Se **lock <> 0**, então região crítica "**ocupada**".

Espera ocupada

- Problema
  - Todas as soluções apresentadas utilizam espera ocupada, ocupam tempo de CPU
  - Processos ficam em estado de espera (looping) até que possam utilizar a região crítica
  - Efeitos inesperadas
    - Processos saem ou entram na região crítica em conjunto
    - Problema de inversão de prioridade

Primitivas sleep/wakeup

```
#define N 100   // numero de lugares no buffer
int count = 0;  // numero de itens no buffer

void process_writer(void) {
  int item;

  while (TRUE) { // loop infinito
    item = create_item();     // produz um item
    if (count == N) sleep(); // se o buffer estiver cheio, durma
    write_item(item);         // ponha o item no buffer
    count = count + 1;        // incremente contador de itens no buffer
    if (count == 1) wakeup(process_reader);
                              // se buffer nao eh vazio, acorde o processo leitor
  }
}

void process_reader(void) {
  int item;

  while (TRUE) { // loop infinito
    if (count == 0) sleep();  // se buffer vazio, durma
    item = get_item();        // retire o item do buffer
    count = count - 1;        // decremente contador de itens do buffer
    if (count == N - 1) wakeup(producer);
                              // o buffer nao esta cheio, acorde o processo escritor
    read_item(item);
  }
}
```

Primitivas sleep/wakeup

* Chamada de sistema que bloqueia e libera o processo
  * Sleep -> bloqueia
  * Wakeup -> libera
* Pode conter 2 parâmetros
  * identificador do processo
  * [endereço de memória] : endereço para equiparar os wakeups a seus respectivos sleeps

Primitivas sleep/wakeup

* Problemas desta solução: Acesso à variável **count** é irrestrita
* Exemplo:
  * O buffer está vazio e o **Leitor** acabou de checar a variável **count** com valor 0;
* O escalonador decide que o **Escritor** será executado;
  * Insere um item no buffer e incrementa a variável **count** com valor 1
  * Envia um sinal de **wakeup** para o **Leitor**
* Escalonador alterna para **Leitor**, que dorme indefinidamente

Semáforos

```
#define N 100 // numero de lugareres no buffer

typedef int semaphore; // semaforos sao um tipo especial de int
semaphore mutex = 1;   // controla o acesso a regiao critica
semaphore empty = N;   // conta os lugares vazios no buffer
semaphore full = 0;    // conta os lugares preenchidos no buffer

void process_writer(void) {
  int item;
  while (true) {               // loop infinito
    item = create_item(item); // cria o item

    down(&empty);              // decrementa contador de lugares vazios
    down(&mutex);              // entra na regiao critica

    write_item(item);          // escreve no buffer

    up(&mutex);                // sai da regiao critica
    up(&full);                 // incrementa contador de lugares preenchidos
  }
}
```

```
void process_reader(void) {
  int item;
  while (true) {          // loop infinito
    down(&full);          // decrementa contador de lugares preenchidos
    down(&mutex);         // entra na regiao critica

    item = get_item();    // retira um item do buffer

    up(&mutex);           // sai da regiao critica
    up(&empty);           // incrementa o contador de lugares vazios

    read_item(item);      // realiza a "leitura" do item
  }
}
```

Semáforos

- Idealizados por E. W. Dijkstra (1965)
- Variável inteira que armazena o número de sinais wakeups enviados
  - 0 (zero) quando não há sinal armazenado
  - Um valor positivo referente ao número de sinais armazenados
- Duas primitivas de chamadas de sistema: down (sleep) e up (wake);
  - Originalmente P (down) e V (up) em holandês

Semáforo - Down

* Down
  * verifica se o valor do semáforo é maior do que 0
  * Se for, o semáforo é decrementado
  * Se o valor for 0, o processo é colocado para dormir sem completar sua operação de down

Semáforo - Up

* Up
  * incrementa o valor do semáforo, fazendo com que algum processo que esteja dormindo possa terminar de executar sua operação down

Semáforo MUTEX

* Mutex acrônimo para Mutual exclusion
* Também chamado de semáforo binário
* Permite 2 valores, 0 e 1
  * Normalmente utilizado para controle de entrada em região crítica

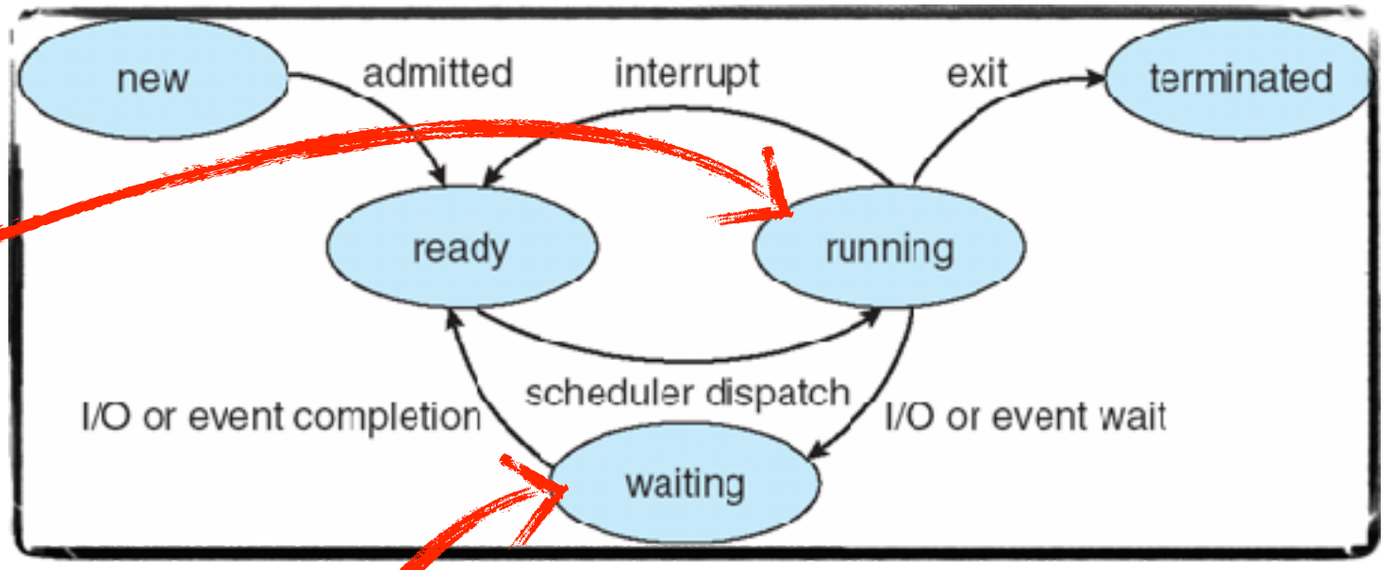
```
down( &mutex )    // solicita entrada em regiao critica

regiao_critica() // acessa recurso compartilhado

up( &mutex )      // sai da regiao critica
```

Espera ocupada vs Mutex

```c
while (true) {
  while (lock != 0); // loop
  lock = 1;
  criticial_region();
  lock = 0;
  non_critical_region();
}
```




```c
down( &mutex )    // solicita entrada em regiao critica

regiao_critica() // acessa recurso compartilhado

up( &mutex )      // sai da regiao critica
```

Problema

* Por que o código abaixo é problemático? Demonstre num passo a passo do código!

```
#define N 100 // numero de lugareres no buffer

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void process_writer(void) {
  int item;
  while (true) {
    item = create_item(item);
    down(&mutex);
    down(&empty);
    write_item(item);
    up(&mutex);
    up(&full);
  }
}
```

```
void process_reader(void) {
  int item;
  while (true) {
    down(&full);
    down(&mutex);
    item = get_item();
    up(&mutex);
    up(&empty);
    read_item(item);
  }
}
```

Monitores

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitores

* Idealizado por Hoare (1974) e Brinch Hansen (1975)
* Monitor - características
  * Primitiva (unidade básica de sincronização) de alto nível para sincronizar processos
  * Conjunto de todos os procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote

Monitores

- Características
  - Somente um processo pode estar ativo dentro do monitor em um mesmo instante
  - Outros processos ficam bloqueados até que possam estar ativos no monitor
  - Controle depende da linguagem de programação

Monitores
Estrutura exemplo

```
monitor example {
   variáveis
   estruturas de dados

   procedimentos A(){
     ...
   }
   função B() {
     ...
   }
}
```

Monitores

* Variáveis : indicam uma condição de corrida
* Operações Básicas : WAIT e SIGNAL
  * wait (var) : bloqueia o processo
  * signal (var) : "acorda" o processo que executou um wait (var) e foi bloqueado

Monitores

* Processo chama a uma rotina do monitor que executa as seguintes tarefas :
  * Testa se um outro processo está ativo dentro do monitor
  * Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor
  * Caso contrário, o processo novo executa a rotina no monitor

Monitores
Concorrência interna

* Como evitar dois processos ativos no monitor ao mesmo tempo?

1. Hoare
   * Coloca o processo mais recente para rodar, suspendendo o outro!!! (sinalizar e esperar)
2. B. Hansen
   * Um processo quando "executa" um SIGNAL deve deixar o monitor imediatamente
   * SIGNAL deve ser o último comando

Semáforos e monitores

* Limitações de semáforos e monitores:
  * Ambos são boas soluções somente para CPUs com memória compartilhada
  * Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas
  * Dependem de uma linguagem de programação

Passagem de mensagem

```c
#define N 100  // numero de lugares no buffer

void produtor(void) {
  int item;
  mensagem msg;                        // buffer de uma mensagem

  while (true) {
    item = prouz_item();               // gera alguma coisa para colocar na mensagem
    receive(consumidor, &msg);         // espera que uma mensagem vazi chegue
    constroi_mensagem(&msg, item); // monta a mensagem para enviar
    send(consumidor, &msg);            // envia mensagem para consumidor
  }
}

void consumidor(void) {
  int item, i;
  mensagem msg;

  for (i = 0; i < N; i++) send(produtor, &msg);
                                  // inicializa enviando todas as mensagens vazias

  while (true) {
    receive(produtor, &msg);    // espera receber mensagem do produtor
    item = extraia_item(&msg); // extrai o item da mensagem
    send(produtor, &msg);       // envia a mensagem vazia como resposta
    usa_item(item);             // faz algo com o item
  }
}
```

Passagem de mensagem

* Características
  * Comandos de enviar (send) e receber (receive)
* Várias formas de implementação que dependem das necessidades de projeto
  * ver comunicação entre processos

Agenda

* Revisão
* Introdução a concorrência
* Mecanismos de controle de concorrência
* Problemas clássicos de concorrência

Problemas clássicos

* 3 problemas
* Jantar dos filósofos
* Leitores e escritores
* Barbeiro sonolento

Jantar dos filósofos (Dining philosophers problem)

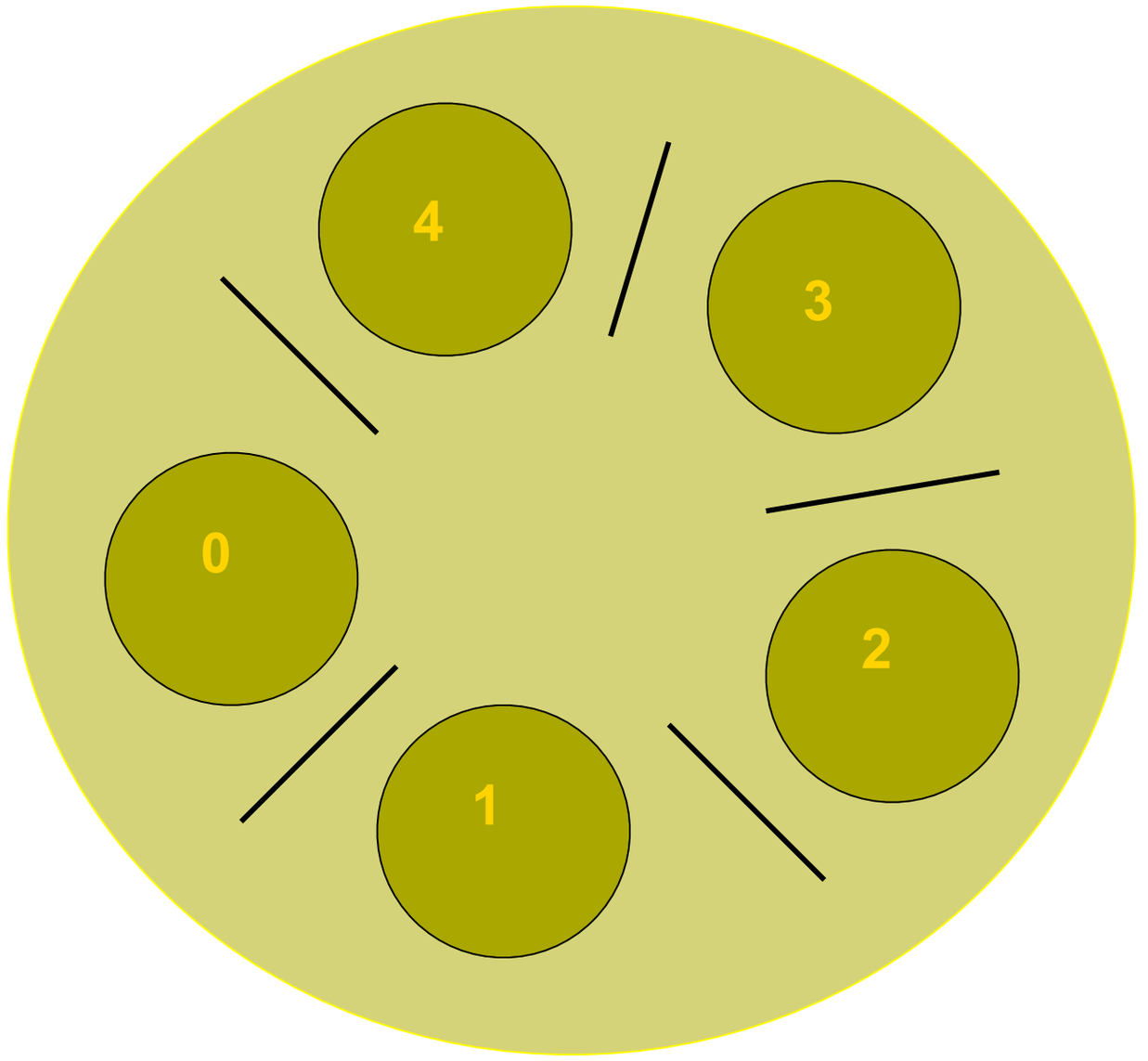
* Dijkstra, 1965
* Definição do problema
  * Cinco filósofos desejam comer espaguete
  * Cada filósofo precisa utilizar dois garfos para comer
  * Cada filósofo tem 1 garfo e um prato de espaguete
  * Os filósofos comem e pensam
  * Filósofos comem por um determinado período de tempo

Jantar dos filósofos

* Deve ser evitado

  1. Um ou mais processos impedidos de continuar sua execução (deadlock ou impasse)

  2. Processos executam mas não progridem (starvation ou inanição)



Jantar dos filósofos

* Solução óbvia

```
#define NUM     5           // numero de filosofos
#define DIREITA (id+1)%NUM // numero do vizinho a direita de id

void filosofo(int id) {
  while (true) {
    pensar();                  // o filosofo esta pensando
    pegar_garfo( id );         // pega o garfo esquerdo
    pegar_garfo( DIREITA );    // pega o garfo direito
    comer();                   // filosofo come
    largar_garfo( id );        // devolve o garfo esquerdo
    largar_garfo( DIREITA ); // devolve o gardo direito
  }
}
```

Jantar dos filósofos

* Problemas da solução óbvia
  * Impasse
    * Todos os 5 filósofos "simultaneamente" pegam o seu garfo
    * Espera eterna pelo garfo "direito"
  * Inanição
    * Se pegar o seu garfo e verificar o garfo esquerdo
    * Indisponível, devolve seu garfo a mesa e espera por algum tempo
    * Todos podem pegar seu garfo, devolvê-lo e esperar

Jantar dos filósofos

* Solução com mutex

```
#define NUM     5           // numero de filosofos
#define DIREITA (id+1)%NUM // numero do vizinho a direita de id

semaphore mutex = 1;

void filosofo(int id) {
  while (true) {
    pensar();                    // o filosofo esta pensando
    down( &mutex );      // solicita entrada na regiao critica
     pegar_garfo( id );          // pega o garfo esquerdo
     pegar_garfo( DIREITA );  // pega o garfo direito
     comer();                    // filosofo come
     largar_garfo( id );         // devolve o garfo esquerdo
     largar_garfo( DIREITA ); // devolve o gardo direito
    up( &mutex );        // solicita saida da regiao critica
  }
}
```

Jantar dos filósofos

* Solução com mutex
* Do ponto de vista teórico, solução adequada
* Do ponto de vista prático, problema de desempenho

Jantar dos filósofos

* Outra solução, baseado em semáforos

```
#define NUM     5               // numero de filosofos

#define EQUERDA (id+NUM-1)%NUM // numero do vizinho a esquerda de id
#define DIREITA (id+1)%NUM     // numero do vizinho a direita de id

#define PENSANDO 0
#define FAMINTO  1
#define COMENDO  2

int estado[NUM];     // arranjo para controlar estado de cada filosofo
semaphore mutex = 1; // mutex para regiao critica
semaphore s[NUM];    // arranjo com um semaforo para cada filosofo

void filosofo(int id) {
  while (true) {
    pensar();               // o filosofo esta pensando
    pegar_garfos( id );     // pega 2 garfos ou bloqueia
    comer();                // filosofo come
    largar_garfos( id );    // devolve os 2 garfos a mesa
  }
}
```

Jantar dos filósofos

* Outra solução, bas

```
#define NUM     5                   // numero de f

#define EQUERDA (id+NUM-1)%NUM // numero do v
#define DIREITA (id+1)%NUM     // numero do v

#define PENSANDO 0
#define FAMINTO  1
#define COMENDO  2

int estado[NUM];     // arranjo para controla
semaphore mutex = 1; // mutex para regiao critica
semaphore s[NUM];    // arranjo com um semaforo para cada filosofo

void filosofo(int id) {
  while (true) {
    pensar();                 // o filosofo esta pensa
    pegar_garfos( id );       // pega 2 garfos ou bloq
    comer();                  // filosofo come
    largar_garfos( id );      // devolve os 2 garfos a
  }
}
```

```
// procedimento ordinario
void pegar_garfos(int id) {
  down( &mutex );         // entra na regiao critica
   estado[id] = FAMINTO; // registra que filosofo id esta faminto
   testar( id );         // tenta pegar 2 garfos
  up( &mutex );          // sai da regiao critica
  down( &s[id] );        // bloqueia se os garfos nao forem pegos
}
```

```
// procedimento ordinario
void largar_garfos(int id) {
  down( &mutex );         // entra na regiao critica
   estado[id] = PENSANDO; // filosofo terminou de comer
   testar(ESQUERDA);      // verifica se o ESQUERDO pode comer agora
   testar(DIREITA);       // verifica se o DIREITO pode comer agora
  up( &mutex );          // sai da regiao critica
}
```

```
// procedimento ordinario
void testar(int id) {
  // verifica se o filosofo esta faminto e seus vizinhos
  // nao estao comendo
  if ( estado[id]       == FAMINTO &&
       estado[ESQUERDA] != COMENDO &&
       estado[DIREITA]  != COMENDO) {
    estado[id] = COMENDO; // modifica o estado do filosofo
    up( &s[id] );         // sinaliza que o filosofo pode
                          //   iniciar a comer
  }
}
```

Leitores e escritores (Readers-writers problem)

* Courtois et al, 1971
* Definição do problema
  * compartilhamento de base de dados
  * 2 tipo de processos: leitores e escritores
  * leitores acessam "simultaneamente" a base
  * escritores têm acesso exclusivo a base

Leitores e escritores

```c
void writer(void) {
  int item;

  while (true) {
    item = create_data(); // cria dados
    down(&db);            // obtem acesso exclusivo a db
    write_data_base(item);    // escreve dados
    up(&db);              // libera acesso exclusivo a db
  }
}
```

```c
void reader(void) {
  int item;

  while (true) {
    down(&mutex);     // obtem acesso exlusivo a rc
    rc = rc + 1;            // um leitor a mais agora
    if (rc == 1) down(&db); // se este for o 1o bloqueia db
    up(&mutex);       // libera acesso exclusivo a rc
    item = read_data_base();
    down(&mutex);     // obtem acesso exclusivo a rc
    rc = rc -1;             // um leitor a menos agora
    if (rc == 0) up(&db);  // se for o ultimo, libera db
    up(&mutex);       // libera acesso exclusivo a rc
    use_data(item);   // realiza a leiruta
  }
}
```

Barbeiro sonolento (Sleeping barber problem)

* Dijkstra, 1965

* Definição do problema
  * Variáveis são : barbeiro, 1 cadeira do barbeiro, e n cadeiras de clientes
  * Quando não há clientes, o barbeiro dorme
  * Quando tem cliente, o barbeiro deve trabalhar
  * Se tem diversos clientes, devem usar as cadeiras disponíveis
  * Caso as cadeiras estejam todas ocupadas, cliente vai embora ou aguarda do lado de fora da barbearia

Barbeiro sonolento

```
void barber(void) {
  while (true) {
    down(&customers);       // vai dormir se o numero de clientes == 0
    down(&mutex);           // obtem acesso exclusivo a 'waiting'
    waiting = waiting - 1;   // descresce o contador de clientes em espera
    up(&barbers);            // um barbeiro esta aguardando o proximo cliente
    up(&mutex);             // libera acesso a 'waiting'
    cut_hair();             // corta o cabelo fora da regiao critica
  }
}
```

```
void customer(void) {
  down(&mutex);             // obtem acesso exclusivo a 'waiting'
  if (waiting < CHAIRS) { // existe cadeira vazias
    waiting = waiting + 1;  // incrementa contador de clientes a espera
    up(&customers);          // acorda barbeiro se necessario
    up(&mutex);              // libera acesso a 'waiting'
    down(&barbers);          // vai dormir se nao tem barbeiro livre
    get_haircut();           // sentado cortando o cabelo
  } else {                  // barbearia esta cheia
    up(&mutex);             // libera acesso a 'waiting'
  }
}
```

Resumo

* Jantar dos filósofos
  * Processos que competem por acesso exclusivo a um número limitado de recurso
* Leitores e escritores
  * Processos de acesso exclusivo e processos de acesso simultâneo
* Barbeiro sonolento
  * Situações de controle com várias filas

Bibliografia

Processos concorrentes
Gerência de processos
Sistemas operacionais

Bibliografia

* **The Java Tutorials: concurrency**. Disponível em http://docs.oracle.com/javase/tutorial/essential/concurrency/ (acessado em 18/01/2013)
* SILBERSCHATZ, G.; GAGNE, G. Sistemas Operacionais com Java. Campus, 7a Ed, 2007.
* R. S. Oliveira, A. S. Carissimi, S. S. Toscani. Sistemas Operacionais. Ed Bookman (Série livros didáticos de informática da UFRGS), 4a Ed.

Processos concorrentes

Sistemas Operacionais
Gerência de processos