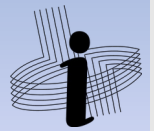
INF 310 – Programação Concorrente e Distribuída

Monitores

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Monitores

- Um monitor é um mecanismo de alto nível para a sincronização de *threads*
  - Similar a uma estrutura abstrata de dados
- Possui primitivas de sincronização explícitas
- O acesso ao Monitor é feito com exclusão mútua

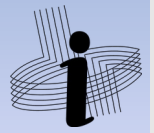


UFV

Monitores

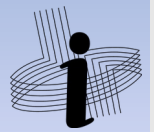
- Estrutura de um monitor

```
monitor <nome> {
    //dados compartilhados pelas threads
    x : condition;

    //procedimentos usados pelas threads
    procedure P(arg1: T1, ..., argN: TN)
    {   ...
        wait(x);
        ...
    }
    procedure Q(arg1: T1, ..., argN: TN)
    {   ...
        signal(x);
        ...
    }
    ...
    //inicialização dos dados do monitor
    initially { ... }
}
```



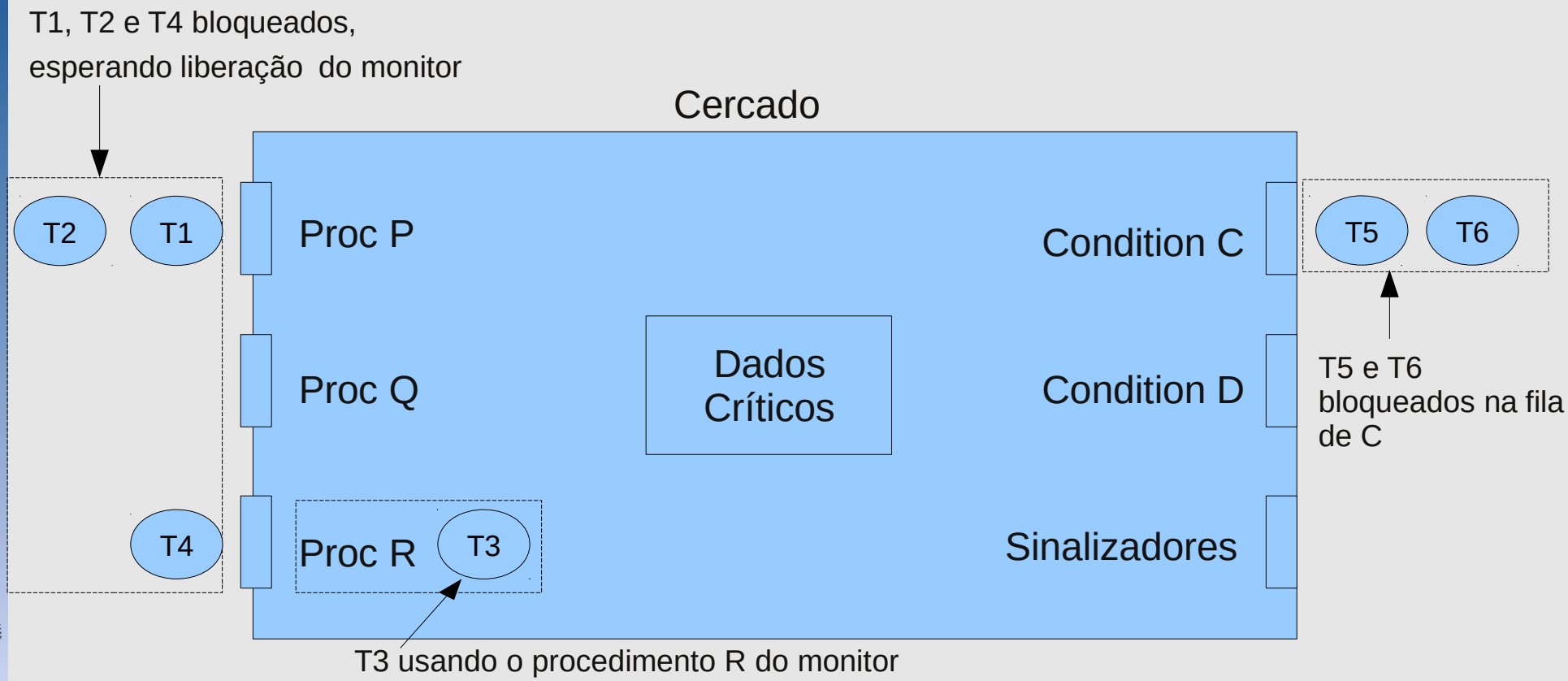
Condition, *wait* e *signal*

- Procedimentos são reentrantes
  - Uso de primitivas de sincronização é essencial
- Variável *condition*
  - Deve ser definida e utilizada apenas pelo monitor através de seus procedimentos
  - Manipulada pelas operações *wait* e *signal*
  - Cada variável do tipo *condition* pode possuir uma fila interna
    - Em C/C++ a escolha do processo desbloqueado é não-determinística
- Operação *wait*
  - Bloqueia a *thread* que a chama
  - A *thread* vai para o grupo de *threads* esperando pela sinalização naquela *condition*
- Operação *signal*
  - Retira um processo da fila e o executa (caso exista)
    - A implementação de C/C++ não garante que ele será executado imediatamente



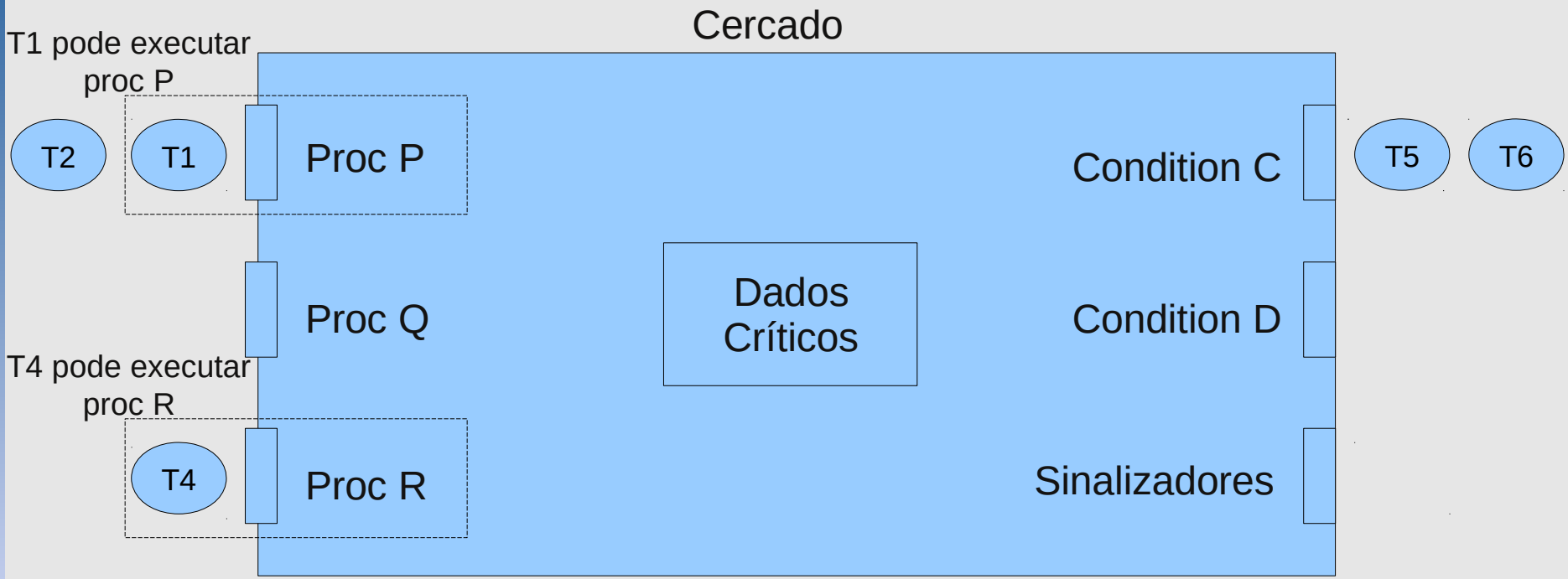
UFV

Analogia com um cercado



Analogia com um cercado

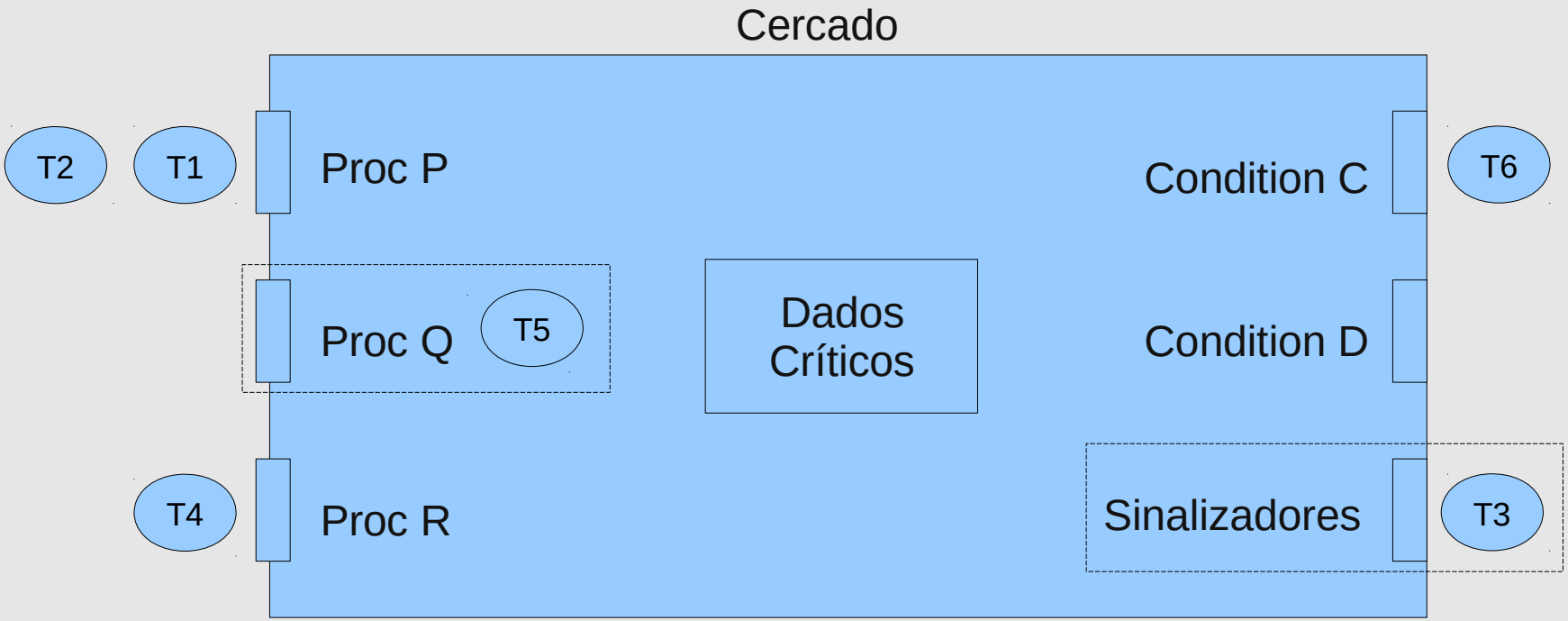
- T3 termina sua execução



E se T3 fizesse *signal*(C)?

Analogia com um cercado

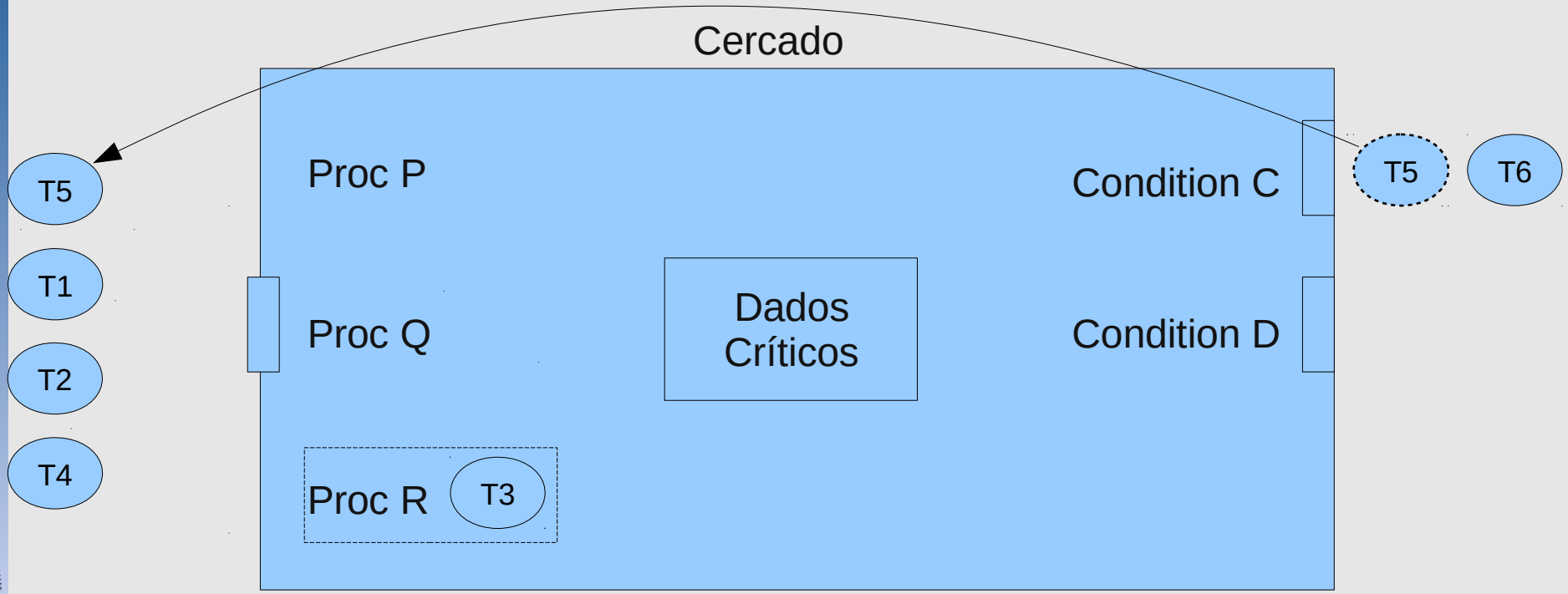
- T3 suspende com *signal* (C)



» T5 passa a usar o Monitor (supondo que T5 tenha chamado a operação *wait(C)* quando estava executando o procedimento Q).

Outras implementações

- Implementação em C/C++ (entrada única)

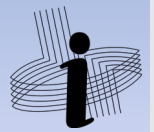


» Todas as *threads* disputam a entrada através de um *mutex*

» O sinal não libera a *thread* T5 imediatamente. Ao invés disso, T5 volta a disputar a entrada para continuar o procedimento interrompido

Outras implementações

- Simplificações
  - Uso do *signal* como o último comando de um procedimento
    - *signal* equivaleria a um *return*
    - Neste caso o processo não voltaria para a fila de entrada do monitor (caso a fila de sinalizadores exista)
- Restrições a serem satisfeitas
  - Exclusão mútua entre os procedimentos
  - A operação *wait* deve bloquear incondicionalmente o processo que a executa
  - A operação *signal* deve ser sem memória (sinal pode ser perdido)
- Restrição não satisfeita em C/C++
  - Uma operação *signal* em fila não vazia faz o processo que a executa perder a CPU
    - Na implementação C/C++ o processo que sinaliza continua no monitor



UFV

Implementação em C++

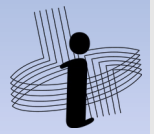
- Uma condição deve estar associada a um *mutex*
- *Mutex* também utilizado para garantir exclusão mútua entre procedimentos
- pthread_cond_t <pthread.h>

```
pthread_cond_t C;
pthread_cond_init(&C,&attr);
pthread_cond_destroy(&C);
pthread_cond_wait(&C,&mux); //mux é um pthread_mutex_t
pthread_cond_timedwait(&C,&mux,&abstime);
pthread_cond_signal(&C);
pthread_cond_broadcast(&C);
```

- std::condition_variable_any <condition_variable>

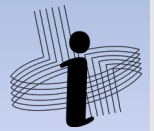
```
std::condition_variable_any C;
C.wait(mux);
C.wait_for(mux,rtime);
C.wait_until(mux,atime);
C.notify_one();
C.notify_all();
```

  - *std::condition_variable* requer uma trava do tipo *std::unique_lock*



Problemas Clássicos

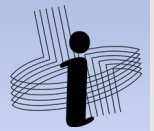
- Alocador de recursos
- Produtor-consumidor com *buffer* limitado
- Caixa Postais tipo *single slot*
- Jantar dos filósofos
- Problema dos leitores e escritores



UFV

Alocador de recursos

- Problema já visto anteriormente
- R instâncias de um mesmo recurso são compartilhadas por T *threads* (T > R)
- Cada *thread* deve requisitar uma instância, usá-la e depois devolvê-la
- O alocador de recursos define 2 procedimentos para as operações de requisição e devolução



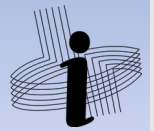
UFV

Alocador de recursos

```
#include<mutex>
#include<condition_variable>

class MonitorRecurso {
private:
    int T;
    int *R;
    mutex mux;
    condition_variable_any C;

public:
    MonitorRecurso(int numR){
        T=numR;
        R=new int[numR];
        for(int i=0;i<numR;++i)
            R[i]=numR-i;
    }

    ~MonitorRecurso(){
        delete [] R;
    }

    void requisita(int &U) {
        mux.lock();
        while(T==0) C.wait(mux);
        U=R[--T];
        printR();
        mux.unlock();
    }

    void libera(int &U) {
        mux.lock();
        R[T++]=U;
        C.notify_one();
        printR();
        mux.unlock();
    }
};

...
/* Uso do monitor */
MonitorRecurso recurso(5);
recurso.requisita(u);
...
recurso.libera(u);
```

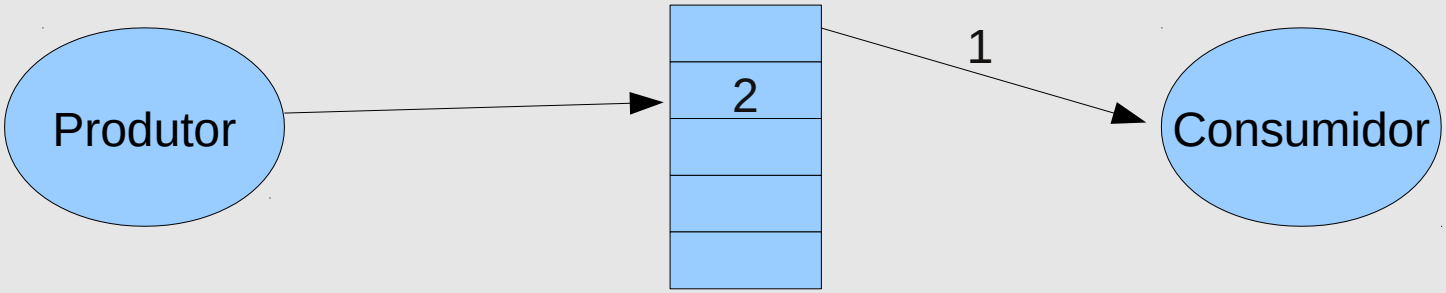
Produtor-consumidor com *buffer* limitado

- Produtor produz um item e tenta colocá-lo no *buffer*. Após colocar um item no *buffer*, volta a produzir outro item
  - Produtor só pode acessar o *buffer* quando existe (pelo menos uma) posição vazia
- Consumidor tenta retirar um item do *buffer*. Após retirar um item do *buffer*, ele o consome volta a tentar pegar outro item no *buffer*
  - Consumidor só pode acessar o *buffer* quando existe (pelo menos uma) posição cheia
- O acesso ao *buffer* é feito com exclusão mútua



UFV

Produtor-consumidor com *buffer* limitado



Produtor-consumidor com *buffer* limitado

```
class MonitorProdCons {
private:
    int bSize,pin,pout,cont;
    int *buffer;
    mutex mux;
    condition_variable naoCheio,naoVazio;

public:
    MonitorProdCons(int bufferSize){
        bSize=bufferSize;
        buffer = new int[bSize];
        pin=pout=cont=0;
    }

    ~MonitorProdCons(){
        delete [] buffer;
    }

    void produzir(int msg) {
        unique_lock<mutex> lck(mux);
        naoCheio.wait(lck,[this](){
                    return cont<bSize; });
        buffer[pin]=msg;
        pin=(pin+1)%bSize;
        ++cont;
        printBuffer();
        naoVazio.notify_one();
    }

    void consumir(int &msg) {
        unique_lock<mutex> lck(mux);
        naoVazio.wait(lck,[this](){
                    return cont>0; });
        msg=buffer[pout];
        pout=(pout+1)%bSize;
        --cont;
        printBuffer();
        naoCheio.notify_one();
    }
};
```

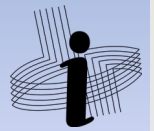
O loop do wait pode ser substituído por uma função lambda que retorna true quando a thread pode ser desbloqueada

Função lambda em C++

- Estrutura de uma função lambda

```
[]() throw() -> tipo {código}
```

  - [ ] define quais objetos ou variáveis estarão acessíveis dentro da função lambda
  - ( ) passagem de parâmetros para a função lambda (opcional)
  - throw() lançamento de exceção (opcional)
  - -> tipo de retorno da função lamba (opcional)
  - { } corpo da função lambda



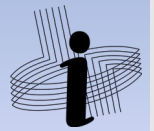
UFV

Caixas postais tipo *single slot*

- Conjunto de caixas postais cada uma com um único *slot*
  - Não há fila de mensagens
- Mensagens

```
send(cp,m)    //envia mensagem m para caixa postal cp
receive(cp,m) //retira mensagem m da caixa postal cp
```

  - ambas as mensagens são síncronas

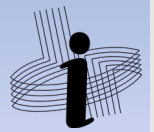


UFV

Caixas postais tipo *single slot*

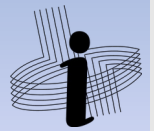
```
class MonitorCaixaPostal {
private:
    int *caixa;
    bool *temMsg;
    condition_variable *cheio, *vazio;
    mutex mux;
public:
    MonitorCaixaPostal(int n){
        caixa=new int[n];
        cheio=new condition_variable[n];
        vazio=new condition_variable[n];
        temMsg=new bool[n];
        for(int i=0;i<n;++i)
            temMsg[i]=false;
    }

    ~MonitorCaixaPostal(){
        delete [] caixa;
        delete [] cheio;
        delete [] vazio;
        delete [] temMsg;
    }
```

```
    void enviar(int pos, int msg) {
        unique_lock<mutex> lck(mux);
        vazio[pos].wait(lck,
                  [this,&pos]()->bool
                  {return !temMsg[pos];});
        caixa[pos]=msg;
        temMsg[pos]=true;
        printBuffer();
        cheio[pos].notify_one();
    }

    void receber(int pos, int &msg) {
        unique_lock<mutex> lck(mux);
        cheio[pos].wait(lck,
                  [this,&pos]()->bool
                  {return temMsg[pos];});
        msg=caixa[pos];
        temMsg[pos]=false;
        printBuffer();
        vazio[pos].notify_one();
    }
};
```



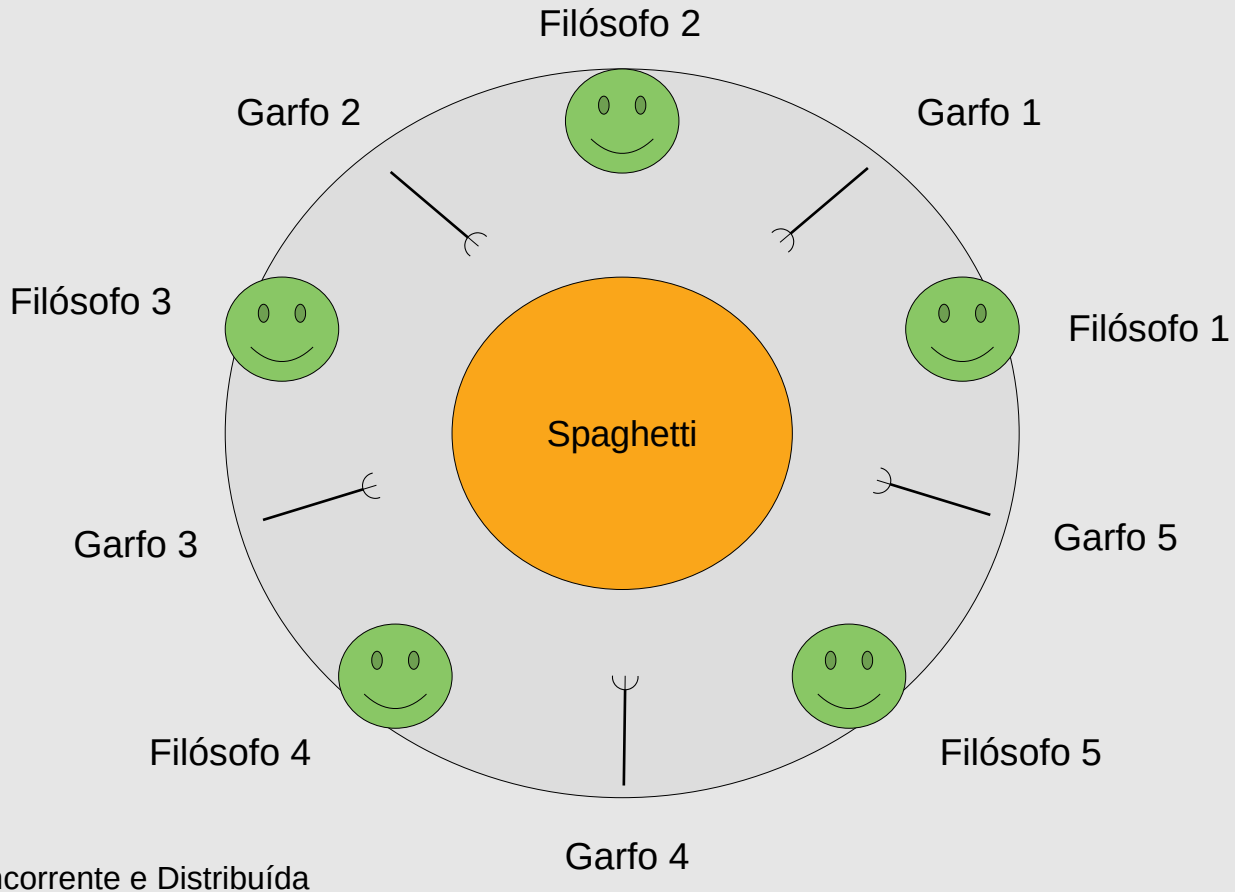
Jantar dos filósofos

- 5 filósofos passam a vida sentados em torno de uma mesa, pensando e comendo.
- Para comer, cada filósofo precisa de 2 garfos
- Cada garfo é compartilhado por 2 filósofos
- Se um filósofo consegue pegar os 2 garfos, ele pode comer e impede seus 2 vizinhos de comer
- Até N/2 filósofos podem comer simultaneamente



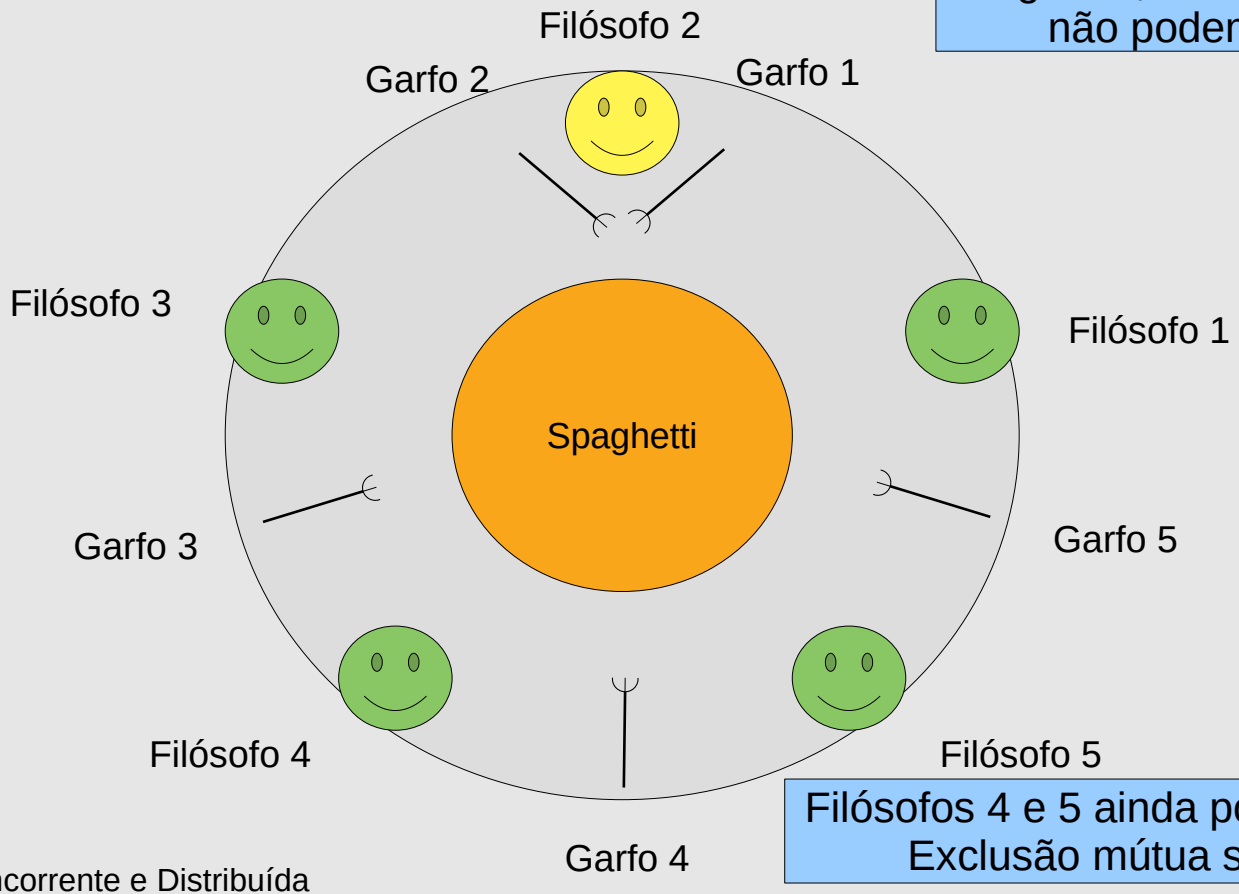
UFV

Jantar dos filósofos



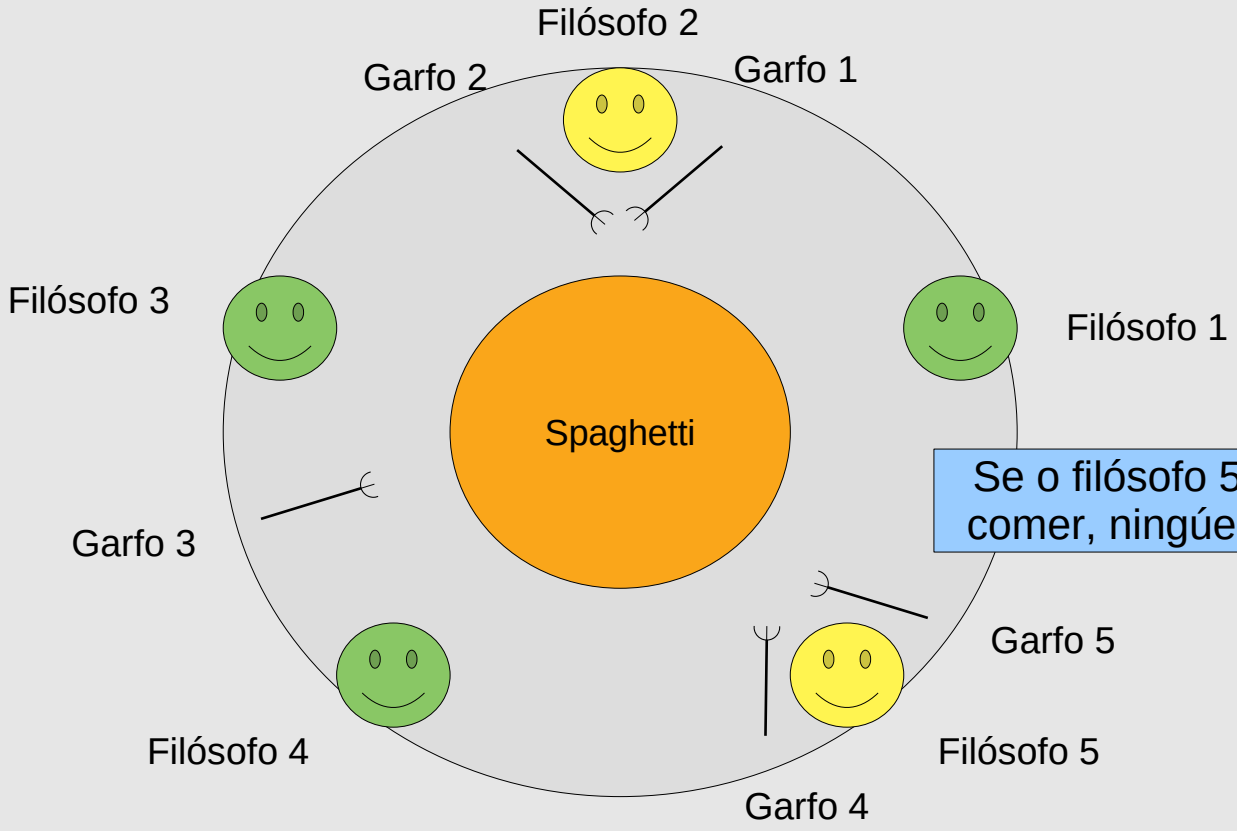
Jantar dos filósofos

Quando o filósofo 2 pega os garfos, os filósofos 1 e 3 não podem comer



Filósofos 4 e 5 ainda podem comer: Exclusão mútua seletiva!

Jantar dos filósofos

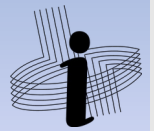


Se o filósofo 5 também começa a comer, ningúem mais pode comer

Jantar dos filósofos

- Solução 1: pegando um garfo de cada vez

```
const int N_FILOSOFOS=5;
MonitorFilosofos garfos(N_FILOSOFOS);

void filosofo(int id, int n) {
    for (int i=0; i<n; ++i) {
        cout<<(id+1)<<" pensando"<<endl;
        garfos.pega(id);   //pega o primeiro garfo
        garfos.pega(id);   //pega o segundo garfo
        cout<<(id+1)<<" comendo pela "<<(i+1)<<"ª vez"<<endl;
        garfos.libera(id); //libera os 2 garfos
    }
}
int main() {
    vector<thread> filosofos;
    for(int i=0; i<N_FILOSOFOS; ++i){
        filosofos.push_back(thread(filosofo,i,3));
    }
    for(thread &f : filosofos)
        f.join();
    return 0;
}
```



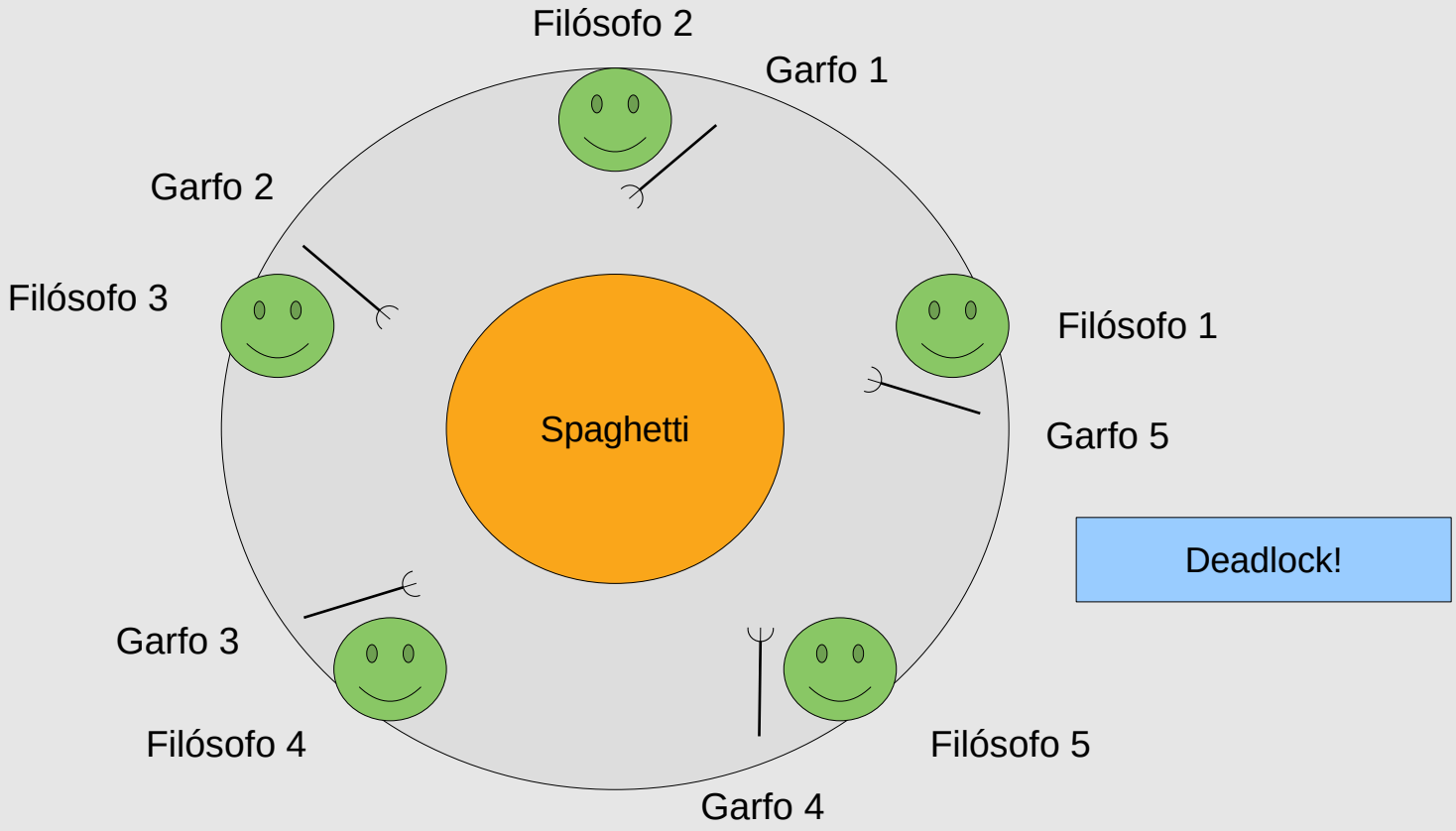
Jantar dos filósofos

```
class MonitorFilosofos {
private:
    bool *gLivre;            //garfo livre
    bool *prim;              //primeiro garfo?
    condition_variable *ok; //uma por filosofo
    mutex mux;

public:
    MonitorFilosofos(int n){ /* ... */ }
    ~MonitorFilosofos(){ /*...*/ }

    void pega(int i) {
        unique_lock<mutex> lck(mux);
        if(prim[i]){
            int j=(i>0 ? i-1 : 4);
            ok[i].wait(lck,[this,&j]()->bool{ return gLivre[j]; });
            gLivre[j]=false;
            prim[i]=false;
        } else {
            ok[i].wait(lck,[this,&i]()->bool{ return gLivre[i]; });
            gLivre[i]=false;
        }
    }

    void libera(int i) {
        unique_lock<mutex> lck(mux);
        int j=(i>0 ? i-1 : 4);
        int k=(i+1)%5;
        gLivre[j]=true;
        gLivre[i]=true;
        prim[i]=true;
        ok[j].notify_one();
        ok[k].notify_one();
    }
};
```

Solução simétrica. Pode haver *deadlock*!

Jantar dos filósofos

- Usando solução simétrica



Jantar dos filósofos

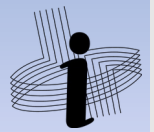
```
void pega(int i) {
    unique_lock<mutex> lck(mux);

    /* Solução assimétrica. Pega primeiro o garfo com índice menor */
    if(prim[i]){
        int j=i>0 ? i-1 : 0;          //menor valor entre esq/dir
        ok[i].wait(lck,[this,&j]()->bool{ return gLivre[j]; });
        gLivre[j]=false;
        prim[i]=false;
    } else {
        int j=i>0 ? i : N_FILOSOFOS-1;    //maior valor entre esq/dir
        ok[i].wait(lck,[this,&j]()->bool{ return gLivre[j]; });
        gLivre[j]=false;
    }
}
```

Solução assimétrica. Sem *deadlock.*

Jantar dos filósofos

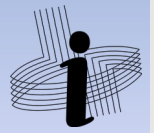
- Solução 2: pegando os dois garfos de uma vez

```
...
MonitorFilosofos garfos(N_FILOSOFOS);
...

void filosofo(int id, int n) {
    for (int i=0; i<n; ++i) {
        cout<<(id+1)<<" pensando"<<endl;
        garfos.pega(id);            //pega os dois garfos
        cout<<(id+1)<<" comendo pela "<<(i+1)<<"ª vez"<<endl;
        garfos.libera(id);          //libera os 2 garfos
    }
}
```



Jantar dos filósofos

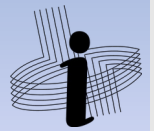
```
int *gLivre;     //num garfos livres para filosofo i (2 inicialmente)
...
void pega(int i) {
    unique_lock<mutex> lck(mux);

    /* Pega os dois garfos na mesma requisição */
    int j=(numFilosofos+i-1)%numFilosofos;  //filosofo da esquerda
    int k=(i+1)%numFilosofos;               //filosofo da direita
    while(gLivre[i]<2) ok[i].wait(lck);
    gLivre[j]-=1;
    gLivre[k]-=1;
}

void libera(int i) {
    unique_lock<mutex> lck(mux);
    int j=(numFilosofos+i-1)%numFilosofos;  //filosofo da esquerda
    int k=(i+1)%numFilosofos;               //filosofo da direita
    gLivre[j]+=1;
    gLivre[k]+=1;
    ok[j].notify_one();
    ok[k].notify_one();
}
```



UFV

Leitores e escritores

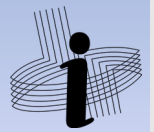
- Processos leitores e escritores compartilham uma base de dados
- Um processo escritor deve acessar a base de dados com exclusão mútua (atualização)
- Processos leitores podem fazer acesso concorrente a base de dados
- Que tipo de processo deve ter prioridade para acessar a base de dados?



UFV

Leitores e escritores

```
...
MonitorReadersAndWriters rw; //monitor
int data=0;                  //dados compartilhados

/* threads escritoras */
void writer(int n) {
    for(int i=0; i<n; i++) {
        rw.startWrite();
        data+=1;
        rw.endWrite();
    }
}

/* threads leitoras */
void reader(int n){
    for(int i=0; i<n; i++){
        rw.startRead();
        cout<<data<<endl;
        rw.endRead();
    }
}
```



Leitores e escritores

```
class MonitorReadersAndWriters {
private:
    int nReader=0;
    bool activeWriter=false;
    int numEsperaW=0;
    condition_variable okToRead, okToWrite;
    mutex mux;
public:
    void startRead() {
        unique_lock<mutex> lck(mux);
        while(activeWriter||numEsperaW>0) {
            okToRead.wait(lck);
        }
        nReader++;
    }

    void endRead() {
        unique_lock<mutex> lck(mux);
        nReader--;
        if(nReader==0) {
            okToWrite.notify_one();
        }
    }

    void startWrite() {
        unique_lock<mutex> lck(mux);
        while(nReader>0||activeWriter) {
            numEsperaW++;
            okToWrite.wait(lck);
            numEsperaW--;
        }
        activeWriter=true;
    }

    void endWrite() {
        unique_lock<mutex> lck(mux);
        activeWriter=false;
        if(numEsperaW>0)
            okToWrite.notify_one();
        else
            okToRead.notify_all();
    }
};
```

De quem é a prioridade?
Leitores ou escritores?

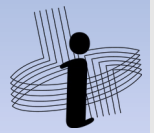
Read-Write locks

- A biblioteca Pthreads implementa trava do tipo leitor-escritor
- pthread_rwlock_t <pthread.h>

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock,&attr);
pthread_rwlock_destroy(&rwlock);
pthread_rwlock_rdlock(&rwlock);        //trava para leitura
pthread_rwlock_tryrdlock(&rwlock);
pthread_rwlock_timedrdlock(&rwlock,&abstime)
pthread_rwlock_wrlock(&rwlock);        //trava para escrita
pthread_rwlock_trywrlock(&rwlock);
pthread_rwlock_timedwrlock(&rwlock,&abstime)
pthread_rwlock_unlock(&rwlock);        //destrava
```

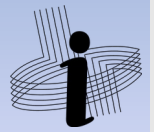
  - Criação e inicialização podem ser feitas ao mesmo tempo com

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```



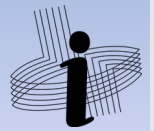
Read-Write locks

```
...
pthread_rwlock_t rwlock;
int data=0;                    //dados compartilhados

void writer(int n) {
    for(int i=0; i<n; i++) {
        pthread_rwlock_wrlock(&rwlock);
        data+=1;
        pthread_rwlock_unlock(&rwlock);
    }
}
void reader(int n){
    for(int i=0; i<n; i++){
        pthread_rwlock_rdlock(&rwlock);
        cout<<data<<endl;
        pthread_rwlock_unlock(&rwlock);
    }
}
int main() {
   pthread_rwlockattr_t attr;     //definindo atributos com preferência pra escritor
   pthread_rwlockattr_setkind_np(&attr,PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP);
   pthread_rwlock_init(&rwlock,&attr);
   ...
   pthread_rwlock_destroy(&rwlock);
}
```



Read-Write locks

- Atributos para definir prioridades
  - PTHREAD_RWLOCK_PREFER_READER_NP
    - política padrão
    - permite que vários leitores consigam a trava mesmo que existam escritores aguardando
  - PTHREAD_RWLOCK_PREFER_WRITER_NP
    - pode resultar em *deadlock* e, por isso, não deve ser utilizada
  - PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP
    - se existe um escritor esperando, novos leitores não têm acesso
    - evita *starvation* dos escritores



UFV

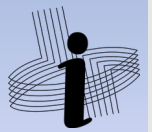
Implementando barreiras

- Muitas implementações de Pthreads não oferecem barreiras
  - Para garantir portabilidade, podemos implementar nossa própria
- Possibilidades
  - block/wakeup : necessidade das *threads* se conhecerem
  - mutex : uso de espera ocupada

```
pthread_mutex_lock(&muxbarrier);
counter++;                          //variável compartilhada iniciada com 0
pthread_mutex_unlock(&muxbarrier);
while(counter<numbarrier);          //espera até valor desejado
```

    - O que acontece se a barreira for reutilizada (ex.: em um *loop*)?

  - Uso de variável de condição pode ser uma forma melhor



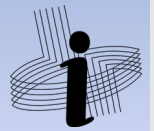
UFV

Implementando barreiras

- Uso de variáveis do tipo condição

```
int counter=0;
pthread_mutex_t muxbarrier=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void barrier(int nThreads) {
    pthread_mutex_lock(&muxbarrier);
    counter++;
    if (counter < nThreads) {
        while(pthread_cond_wait(&cond,&muxbarrier)!=0);
    } else {
        counter=0;
        pthread_cond_broadcast(&cond);
    }
    pthread_mutex_unlock(&muxbarrier);
}
```

  - É possível que a thread retorne do *wait* mesmo que não aconteça um *signal* ou *broadcast*
    - nesse caso pthread_cond_wait retorna 0 e deve bloquear novamente
    - o *wait* de *std::condition_variable* não possui valor de retorno, exigindo maior cuidado



Thread pool

- Criar *n* worker threads (ex: n = thread::hardware_concurrency())
- Cada thread implementa um loop que retira uma *task* da fila e executa
  - Cada entrada da fila pode ser um *struct* ou *pair* contendo o *task* e seus parâmetros
  - O *loop* espera até a fila não estar vazia ou não existir mais nada para ser feito

```
typedef void (*functiontype)(int); //função void que recebe um param int
queue<pair<functiontype,int>> tasks;
...
void thread_loop() {
    while(true){
        mux.lock();
        condition.wait(mux, []{ return !tasks.empty() || terminate_pool };
        if(terminate_pool) break;
        pair<functiontype,int> t;
        t=tasks.front();          //lê par <função,param> da fila
        tasks.pop();
        mux.unlock();
        t.first(t.second);        //executa função com parâmetro passado
    }
}
```