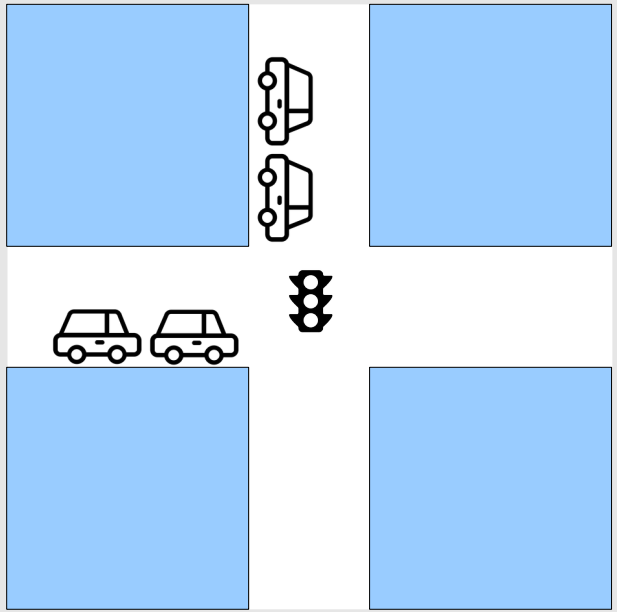
INF 310 – Programação Concorrente e Distribuída

Semáforos

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

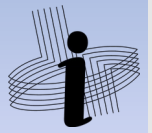
Semáforos

- Analogia com um cruzamento entre duas vias
  - Intersecção é um recurso compartilhado
  - Veículos não precisam conhecer uns aos outros
  - Sincronização realizada através de semáforos



Semáforos

- Variáveis especiais manipuladas unicamente através das operações Up e Down
  - S variável do tipo semáforo
  - Down(S): espera até S ser maior que 0 e então subtrai 1 de S
  - Up(S): incrementa S de 1
  - Alguns autores usam os nomes V e P para as operações Up e Down, respectivamente
- As operações Down(S) e Up(S) são operações atômicas, executadas no *kernel* do SO
  - Chamadas de sistema
- Um semáforo possui um valor inteiro e uma fila associada
  - A ordem em que as *threads* são desbloqueadas pode variar de acordo com implementação da biblioteca e do sistema
- É um mecanismo de nível mais baixo e mais leve que monitor

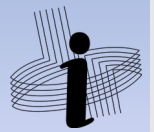


Monitor e Semáforos

- Implementação de semáforos com um monitor

```
monitor Semaf {
    S : integer;
    C : condition;

    procedure Down() {
        S = S - 1;
        if S < 0 then wait(C);     // bloqueia o processo
    }

    procedure Up() {
        S = S + 1;
        if S <= 0 then signal(C); // bloqueado na fila de C
    }

    initially S = 2;               // semáforo iniciado com 2
}
```

```
// Uso em um processo
...
Semaf.Down();
...
Semaf.Up();
```



UFV

Monitor e Semáforos

- Simulação de monitor usando semáforos

```
X : semaphore initial 1;
proc p()
    Down(X); //Entrada do monitor
    ...
    Up(X);   //Saindo do monitor

// Variável de condição C simulada com:
nC : integer initial 0; //núm processos na fila de C
sC : semaphore initial 0;

wait(C):
    nC = nC + 1;
    Up(X);
    Down(sC);

signal(C):
    if nC > 0 {
        nC = nC - 1;
        Up(sC); Down(X);
    }
```

Implementação usando fila única

Sincronizações básicas com Up e Down

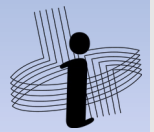
- Acesso com exclusão mútua a *n* regiões críticas

```
X: semaphore initial 1
```

```
P1:
…
Down(X);
Região Crítica;
Up(X);
…
```

```
P2:
…
Down(X);
Região Crítica;
Up(X);
…
```

```
Pn:
…
Down(X);
Região Crítica;
Up(X);
…
```



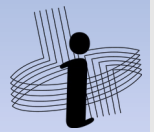
UFV

Sincronizações básicas com Up e Down

- Sincronização para comunicação

```
Y: semaphore initial 0
```

```
P1:                          P2:
…                            …
Down(Y);                     comando_B;
comando_A;                   Up(Y);
…                            …
```



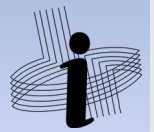
UFV

Sincronizações básicas com Up e Down

- Sincronização de barreiras

```
Y,Z: semaphore initial 0
```

```
P1:                          P2:
…                            …
Up(Z);                       Up(Y);
Down(Y);                     Down(Z);
comando_A;                   comando_B;
…                            …
```



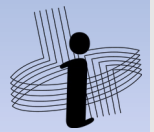
UFV

Semáforos em C/C++

- sem_t <semaphore.h>
  - Principais métodos

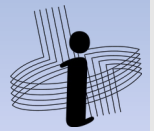
```
sem_t s;
sem_init(&s,0,v);     //inicia semáforo s com v
sem_destroy(&s);
sem_post(&s);         //Up: incrementa 1 unid.
sem_wait(&s);         //Down: decrementa 1 unid. (bloqueia se preciso)
sem_trywait(&s);
sem_timedwait(&s,&abstime)
sem_getvalue(&s,&v); //obtém o valor do contador do semáforo s
```

  - O segundo parâmetro de sem_init seria utilizado para informar se o semáforo é compartilhado entre processos após a chamada de um fork. Na prática não é utilizado



Programas clássicos

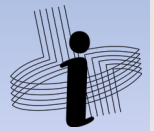
- Alocador de recursos
- Produtor-consumidor com buffer limitado
- Jantar dos filósofos
- Problema do barbeiro dorminhoco
- Problema dos leitores e escritores



UFV

Alocador de recursos

- Problema já visto em capítulos anteriores
- 5 instâncias de um mesmo recurso são compartilhadas por 2 ou mais processos
- Cada processo deve requisitar uma instância, usá-la e depois devolvê-la
- O Alocador de recursos define 2 procedimentos para as operações de requisição e devolução.

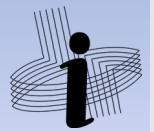


UFV

Alocador de recursos

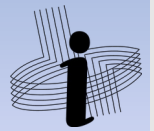
```
...
#include <semaphore.h>

int R[]={5,4,3,2,1};
int T=5;
sem_t cont, mux;

int requisita() {
    sem_wait(&cont);
    sem_wait(&mux);
    int u=R[--T];
    sem_post(&mux);
    return u;
}

void libera(int u) {
    sem_wait(&mux);
    R[T++]=u;
    sem_post(&mux);
    sem_post(&cont);
}
```

```
void *thread_function(void*) {
   ...
}

int main() {
    sem_init(&mux,0,1);
    sem_init(&cont,0,5);

    /* executa threads */

    sem_destroy(&mux);
    sem_destroy(&cont);
    return 0;
}
```



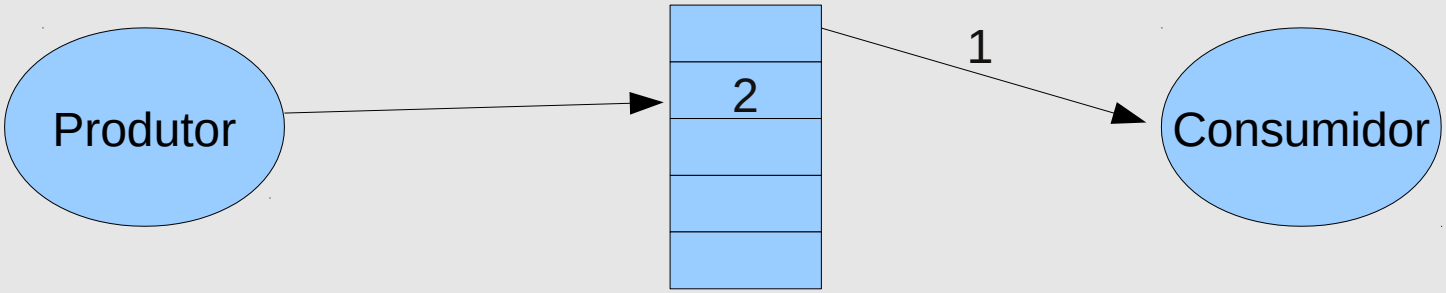
Produtor-consumidor com *buffer* limitado

- Processo 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
- Processo 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 Produtor só pode acessar o *buffer* quando existe (pelo menos uma) posição vazia
- O acesso ao *buffer* é feito com exclusão mútua



UFV

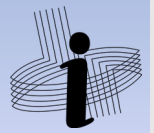
Produtor-consumidor com *buffer* limitado



Produtor-consumidor com *buffer* limitado

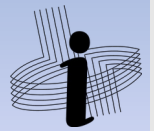
```
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
sem_t cheios;
sem_t vazios;

void *prod(void *n) {
    int msg=0, pin=0;
    for(int i=0; i<(long)n; ++i){
        /* produz msg */
        sem_wait(&vazios);
        buffer[pin]=msg;
        sem_post(&cheios);
        pin=(pin+1)%BUFFER_SIZE;
    }
    pthread_exit(NULL);
}
```

```
void *cons(void *n) {
    int msg, pout=0;
    for(int i=0; i<(long)n; ++i){
        sem_wait(&cheios);
        msg=buffer[pout];
        sem_post(&vazios);
        pout=(pout+1)%BUFFER_SIZE;
        /* consome msg */
    }
    pthread_exit(NULL);
}
int main() {
    sem_init(&cheios,0,0);
    sem_init(&vazios,0,BUFFER_SIZE);
    pthread_t p, c;
    pthread_create(&p,NULL,prod,(void*)20);
    pthread_create(&c,NULL,cons,(void*)20);
    pthread_join(p,NULL);
    pthread_join(c,NULL);
    sem_destroy(&cheios);
    sem_destroy(&vazios);
    return 0;
}
```



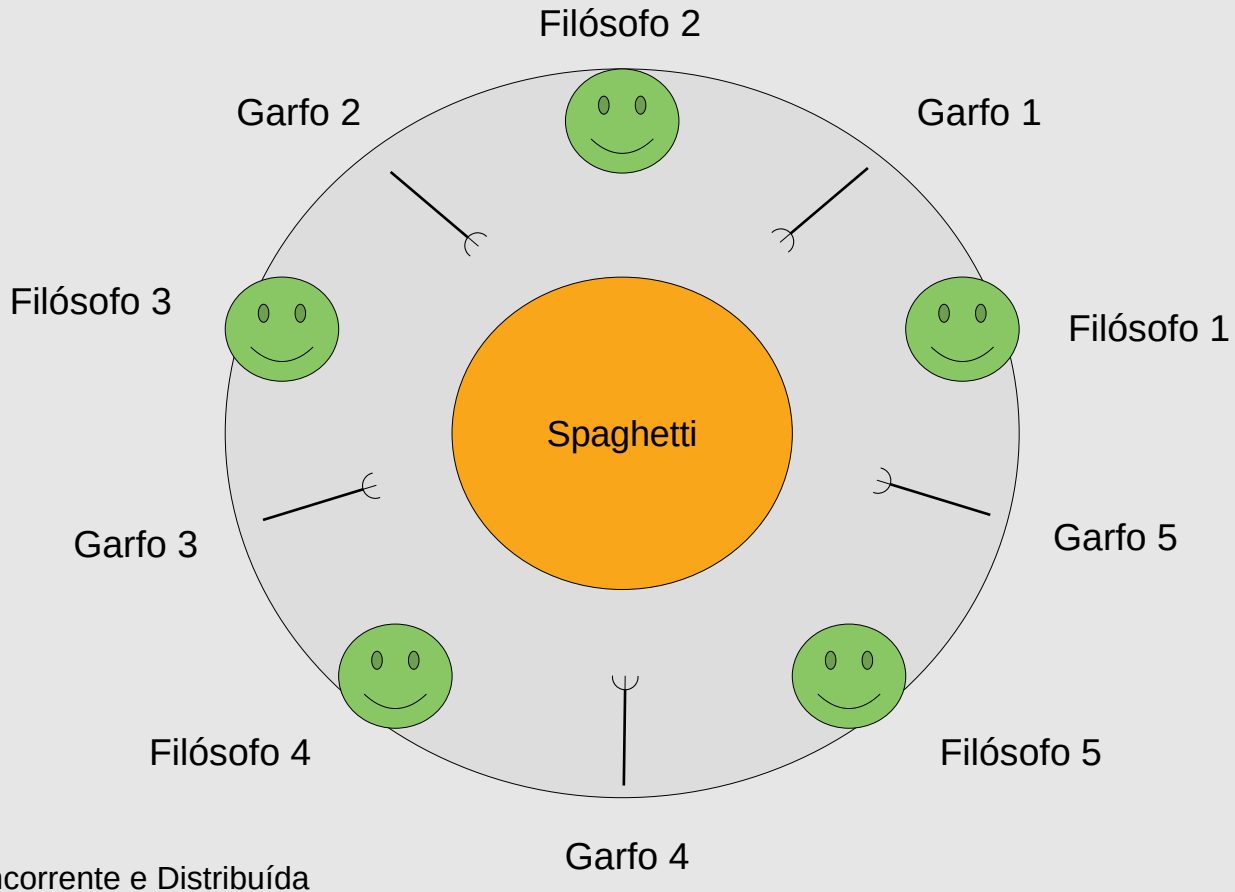
Jantar dos filósofos

- 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



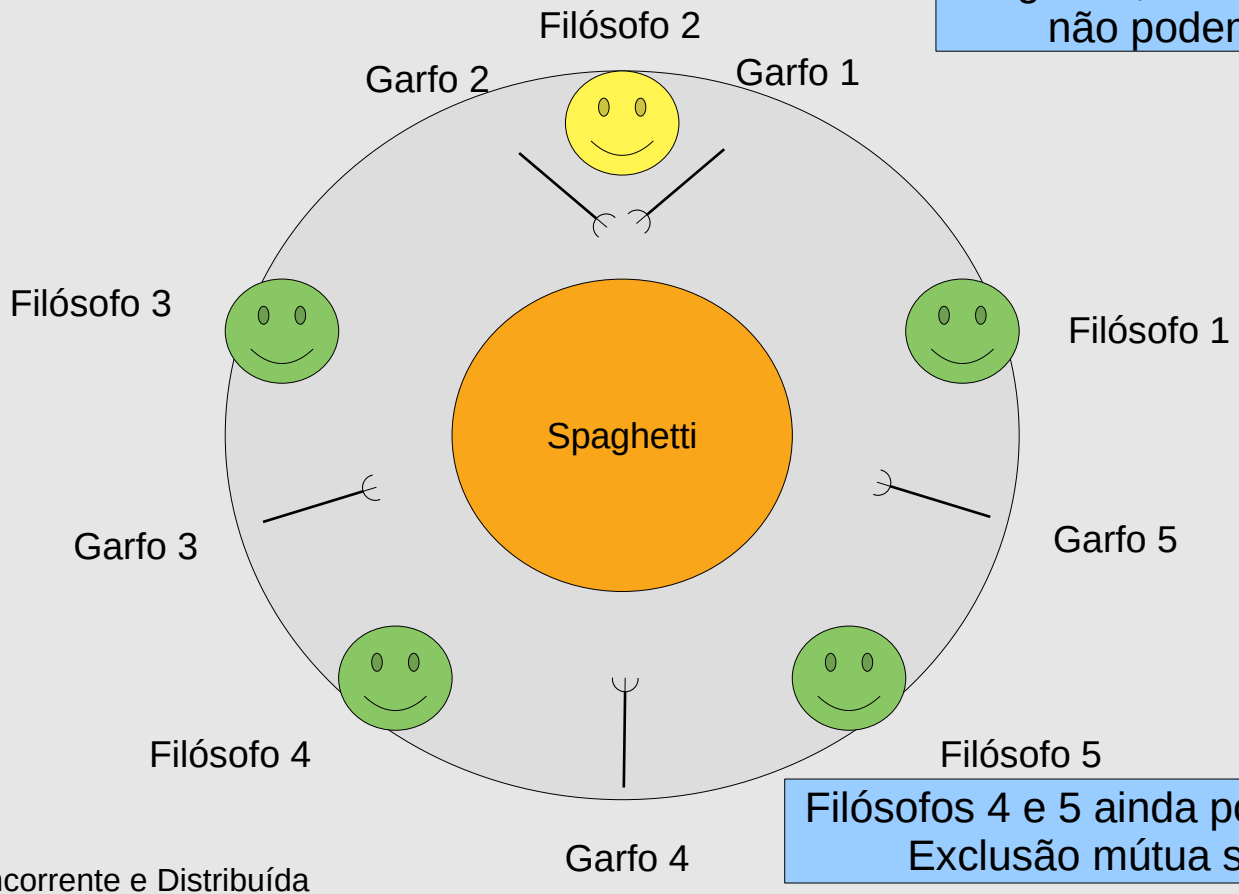
UFV

Jantar dos filósofos



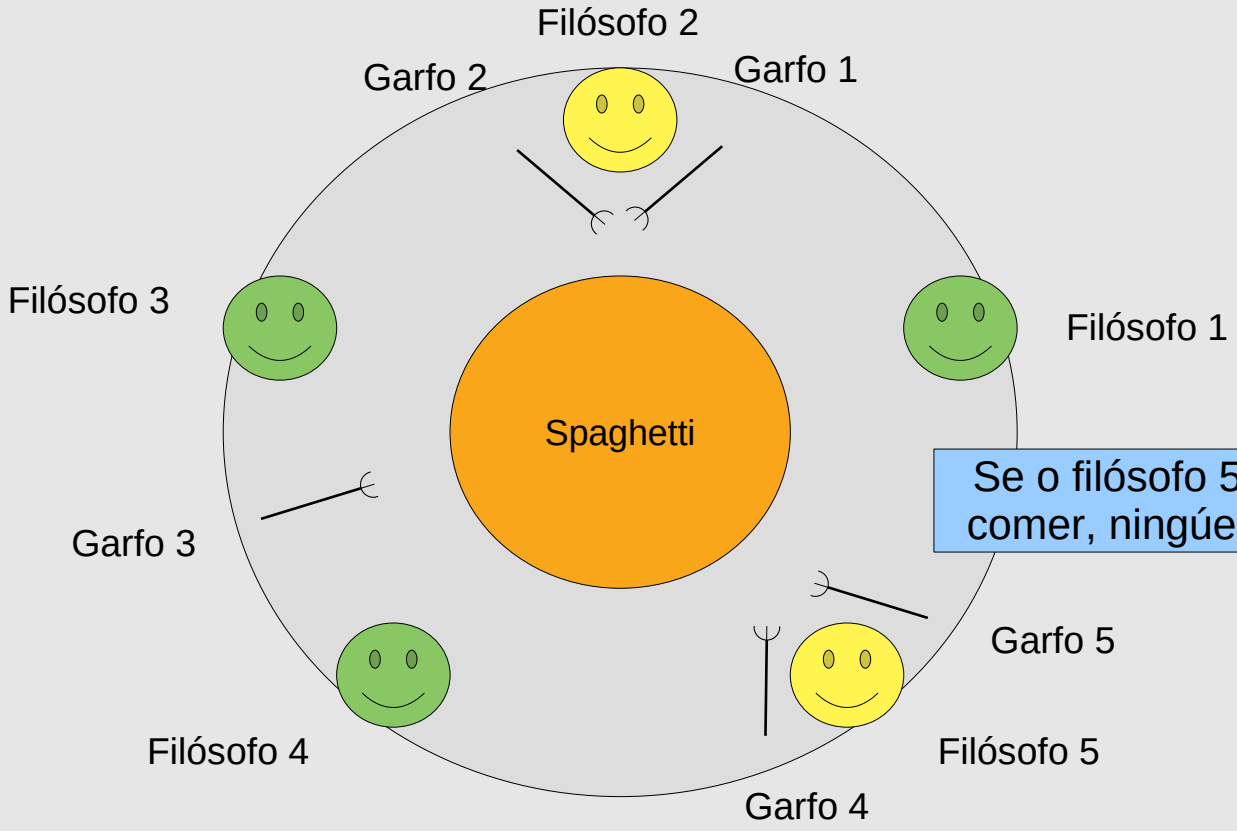
Jantar dos filósofos

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!

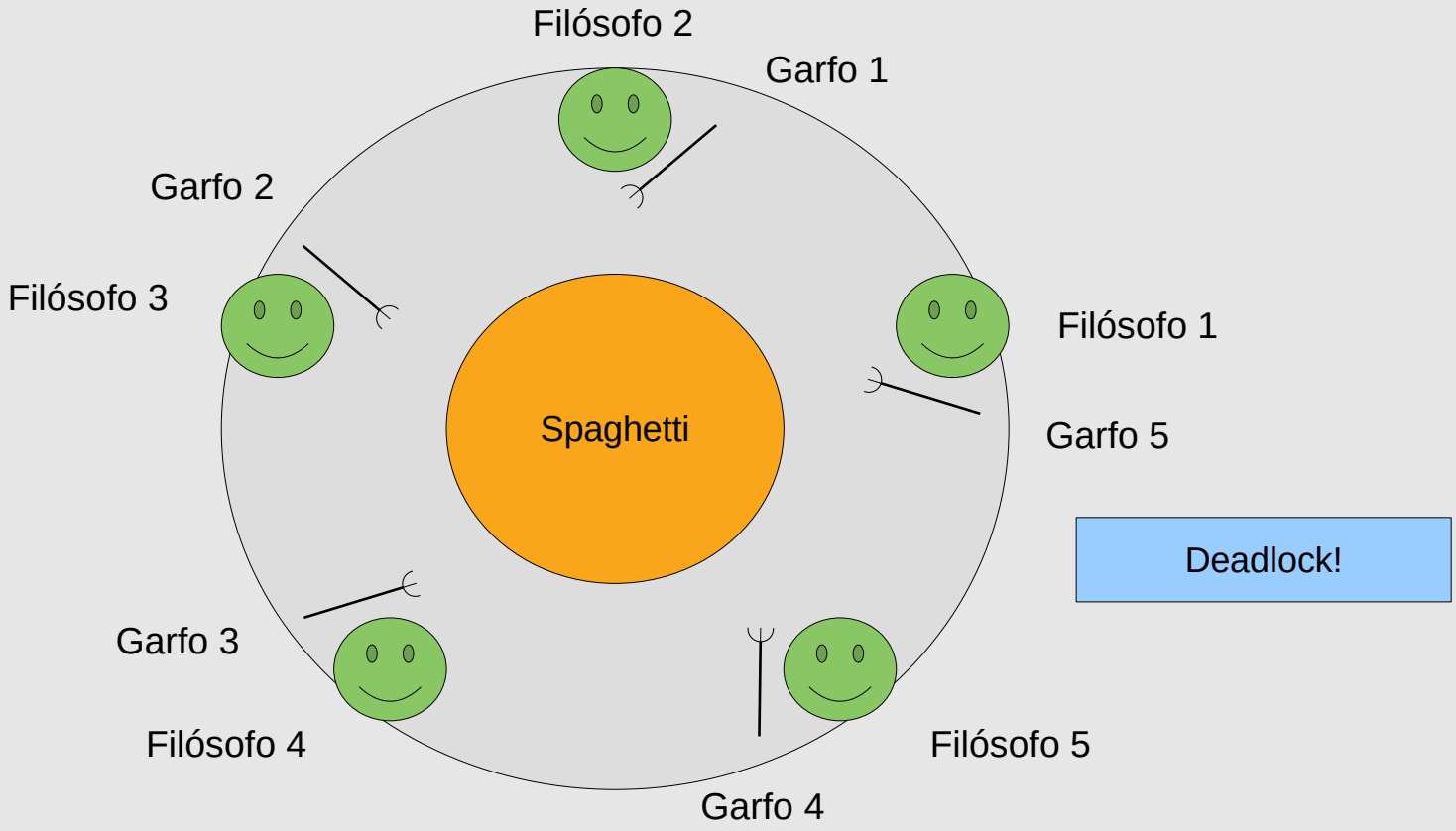
Jantar dos filósofos



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

Jantar dos filósofos

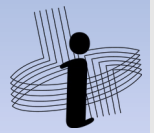
- Usando solução simétrica



Jantar dos filósofos

```
...
#define NF 5
sem_t *garfos;

void getForks(int id) {
    if(id>0) {
        sem_wait(&garfos[id-1]);
        sem_wait(&garfos[id]);
    } else {
        sem_wait(&garfos[0]);
        sem_wait(&garfos[NF-1]);
    }
}

void putForks(int id) {
    if(id>0) {
        sem_post(&garfos[id-1]);
        sem_post(&garfos[id]);
    } else {
        sem_post(&garfos[0]);
        sem_post(&garfos[NF-1]);
    }
}
```

```
void *filosofo(void *params) {
    int id=((int*)params)[0];
    int n=((int*)params)[1];
    for(int i=0; i<n; ++i) {
        getForks(id);
        /* comendo */
        putForks(id);
    }
    delete [] (int*)params;
    pthread_exit(NULL);
}
int main() {
    garfos=new sem_t[NF];
    for(int i=0; i<NF; ++i)
        sem_init(&garfos[i],0,1);
    vector<pthread_t> filosofos;
    for(int i=0; i<NF; ++i){
        pthread_t t;
        int *p = new int[2];
        p[0]=i; p[1]=10;
        pthread_create(&t,NULL,filosofo,p);
        filosofos.push_back(move(t));
    }
    ... esperar threads terminarem
    for(int i=0; i<NUM_FILOSOFOS; ++i)
        sem_destroy(&garfos[i]);
    delete [] garfos;
}
```



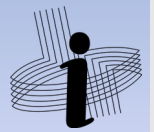
O barbeiro dorminhoco

- Uma barbearia possui N cadeiras em uma sala de espera e 1 cadeira de barbeiro
- Se não tem clientes na sala de espera, o barbeiro senta em sua cadeira e dorme.
- Quando um cliente chega
  - Se o barbeiro está dormindo, ele o acorda
  - Se o barbeiro está atendendo a um outro cliente, o cliente espera em uma das cadeiras da sala de espera
  - Se a sala de espera não tem cadeira disponível, ele vai embora

O barbeiro dorminhoco

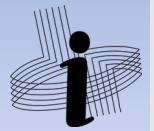
```
sem_t sCli,sFila;//iniciados com 0
sem_t mux;       //iniciado com 1

void *barbeiro(void *n) {
    for(long i=0; i<(long)n; ++i){
        sem_wait(&sCli);
        cout<<"Chama cliente"<<endl;
        sem_post(&sFila);
    }
    pthread_exit(NULL);
}
```

```
void *cliente(void *id) {
    sem_wait(&mux);
    int nEspera;
    sem_getvalue(&sCli,&nEspera);
    if(nEspera < 3) {
        sem_post(&sCli);
        sem_post(&mux);
        sem_wait(&sFila);
        cout<<(long)id<<" atendido"<<endl;
    } else {
        cout<<(long)id<<" desiste"<<endl;
        sem_post(&mux);
    }
    pthread_exit(NULL);
}

int main() {
    sem_init(&sCli,0,0);
    sem_init(&sFila,0,0);
    sem_init(&mux,0,1);
    ...
}
```



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?



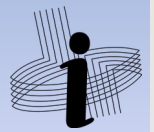
UFV

Leitores e Escritores

```
sem_t wr, mux;  //semáforos iniciados de 1
int nr=0;       //número de leitores
int v=0;        //dado compartilhado para leitura e escrita

void *leitor(void *param) {
    int id = ((int*)param)[0];
    int n = ((int*)param)[1];
    for(int i=0; i<n; ++i){
        sem_wait(&mux);
        if(nr==0) sem_wait(&wr);
        nr++;
        sem_post(&mux);

        cout<<id<<" lê "<<v<<endl;

        sem_wait(&mux);
        nr--;
        if(nr==0) sem_post(&wr);
        sem_post(&mux);
    }
    delete [] (int*) param;
    pthread_exit(NULL);
}
```

```
void *escritor(void *param) {
    int id = ((int*)param)[0];
    int n = ((int*)param)[1];
    for(int i=0; i<n; ++i){
        sem_wait(&wr);
        v++;
        cout<<id<<" escreve "<<v<<endl;
        sem_post(&wr);
    }
    delete [] (int*) param;
    pthread_exit(NULL);
}
```

Prioridade para threads leitoras



Leitores e Escritores

```
sem_t wr, rd, muxR, muxW; //iniciados de 1
int nr,nw=0;    //núm leitores e escritores
int v=0;        //dado compartilhado

void *leitor(void *param) {
    int id = ((int*)param)[0];
    int n = ((int*)param)[1];
    for(int i=0; i<n; ++i){
        sem_wait(&rd);
        sem_wait(&muxR);
        nr++;
        if(nr==1) sem_wait(&wr);
        sem_post(&muxR);
        sem_post(&rd);

        cout<<id<<" lê "<<v<<endl;

        sem_wait(&muxR);
        nr--;
        if(nr==0) sem_post(&wr);
        sem_post(&muxR);
    }
    delete [] (int*) param;
    pthread_exit(NULL);
}
```

```
void *escritor(void *param) {
    int id = ((int*)param)[0];
    int n = ((int*)param)[1];
    for(int i=0; i<n; ++i){
        sem_wait(&muxW);
        nw++;
        if(nw==1) sem_wait(&rd);
        sem_post(&muxW);

        sem_wait(&wr);
        v++;
        cout<<id<<" escreve "<<v<<endl;
        sem_post(&wr);

        sem_wait(&muxW);
        nw--;
        if(nw==0) sem_post(&rd);
        sem_post(&muxW);
    }
    delete [] (int*) param;
    pthread_e
}
```

Prioridade para threads escritoras