



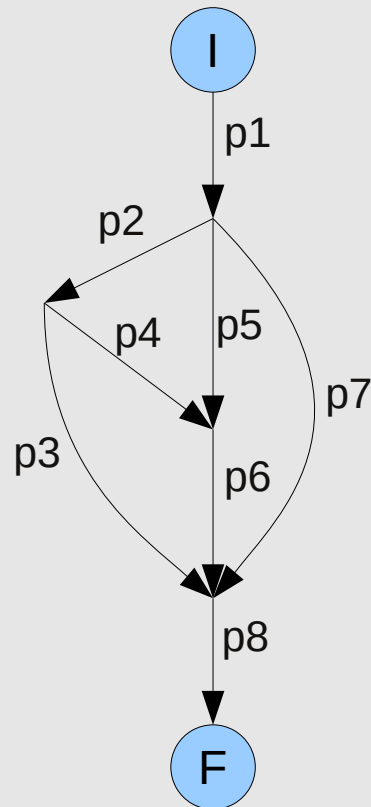
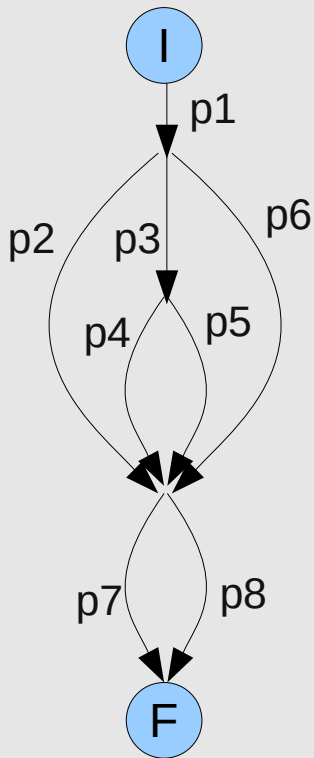
## INF 310 – Programação Concorrente e Distribuída

### Especificação de Concorrência

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

# Relação de precedência

- Especificação usando grafo dirigido
  - Grafo de fluxo de processo

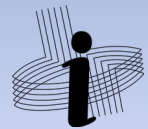
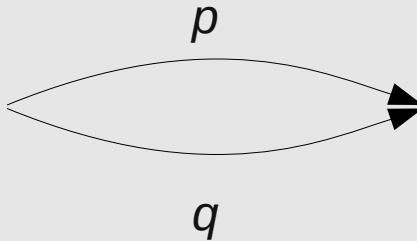


# Funções S e P

- $S(p, q)$  processos  $p$  e  $q$  em série

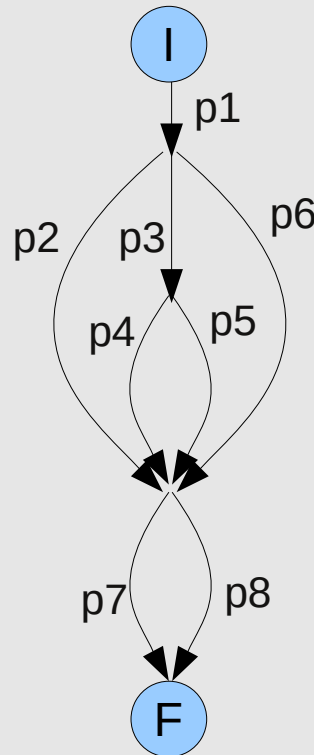


- $P(p, q)$  processos  $p$  e  $q$  em paralelo

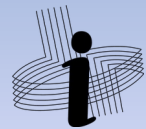


# Funções S e P

- Especificando o grafo como função S e P
  - $S(S(p1, P(P(p2, S(p3, P(p4, p5))), p6)), P(p7, p8))$

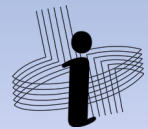
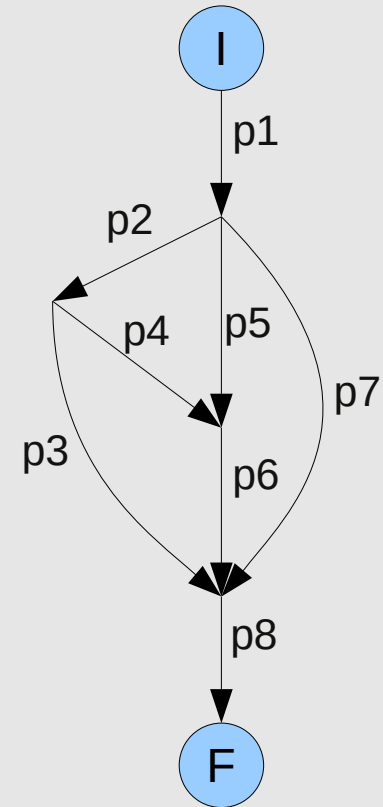


Um grafo que pode ser expresso por funções S e P é dito propriamente aninhado



# Funções S e P

- Exprese o grafo a seguir através de funções S e P
  - O grafo não é propriamente aninhado
  - Como modificá-lo para que seja propriamente aninhado?

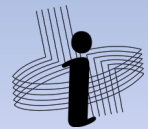


# Comandos cobegin/coend

- Proposto por Dijkstra em 1965
  - Chamados também de parbegin/parend
  - Especificam um conjunto de comandos para serem executados em paralelo
  - Sintaxe:

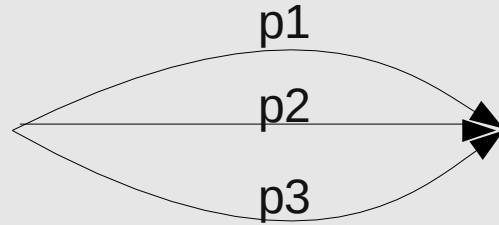
```
cobegin C1 | C2 | ... | Cn coend
```

onde cada  $C_i$  corresponde a um código autônomo



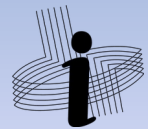
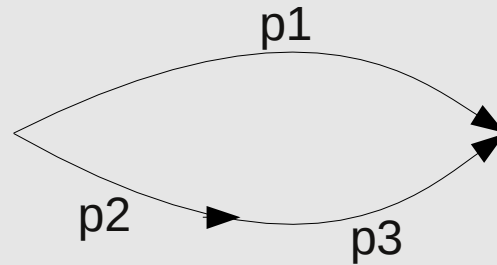
# Comandos cobegin/coend

```
cobegin p1 | p2 | p3 coend
```



; tem precedência sobre o |

```
cobegin p1 | p2; p3 coend
```

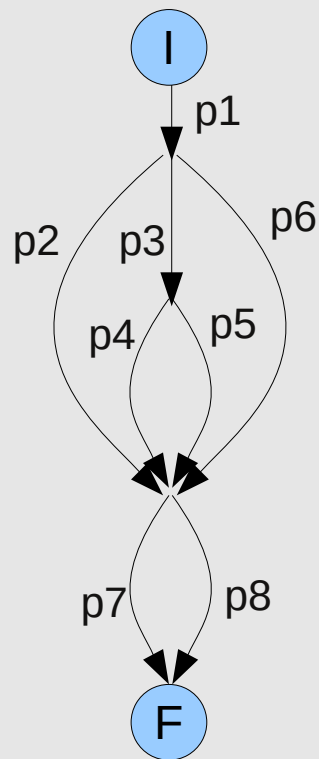


# Comandos cobegin/coend

- Código para o grafo apresentado

```
p1;  
cobegin p2  
    | p3; cobegin p4 | p5 coend  
    | p6  
coend;  
cobegin p7 | p8 coend
```

cobegin/coend conseguem  
especificar apenas grafos  
propriamente aninhados



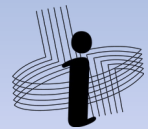


# Primitivas *fork*, *join* e *quit*

- Definidas por Conway em 1963
  - *fork*: cria um processo filho
  - *join*: usada para sincronização dos processos
    - Semântica do *join t,w*

```
t = t - 1;  
if t == 0 then goto w  
else nothing;
```

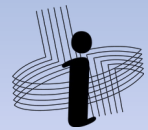
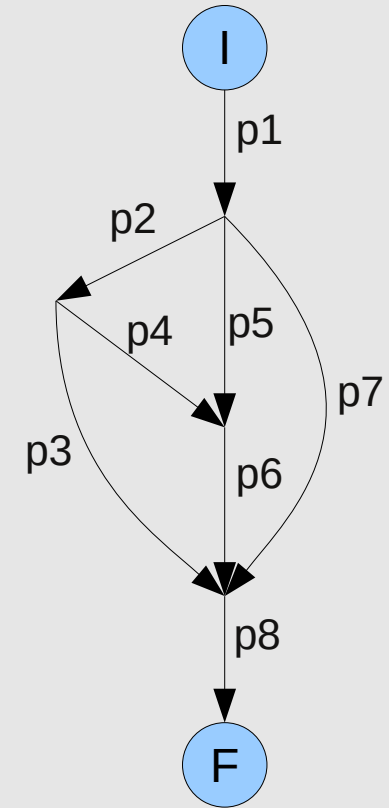
- *quit*: usada para sinalizar o término da execução de um processo



# Primitivas *fork*, *join* e *quit*

- Exemplo 1:

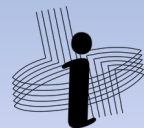
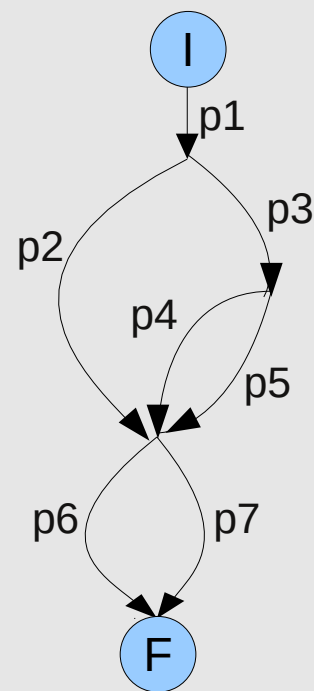
```
t1 = 2; t2 = 3;  
p1; fork a2; fork a5; fork a7; quit;  
a2: p2; fork a3; fork a4; quit;  
a3: p3; join t2, a8; quit;  
a4: p4; join t1, a6; quit;  
a5: p5; join t1, a6; quit;  
a6: p6; join t2, a8; quit;  
a7: p7; join t2, a8; quit;  
a8: p8; quit;
```



# Primitivas *fork*, *join* e *quit*

- Exemplo 2:

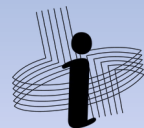
```
t1 = 3;  
p1; fork a2; fork a3; quit;  
a2: p2; join t1, a67; quit;  
a3: p3; fork a4; fork a5; quit;  
a4: p4; join t1, a67; quit;  
a5: p5; join t1, a67; quit;  
a67: fork a6; fork a7; quit;  
a6: p6; quit;  
a7: p7; quit;
```



## Primitivas *fork*, *join* e *quit*

---

- Programas concorrentes representados por grafos propriamente aninhados são geralmente mais bem estruturados
- Primitivas *fork*, *join* e *quit* tem melhor poder de representação, ao custo de poder tornar os programas mais confusos

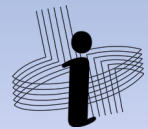


# Fork em sistemas Unix

- Cria uma cópia idêntica do processo
- Processo filho recebe cópias das variáveis e dos descritores dos arquivos do processo pai
- Retorna o identificador do processo (PID) filho para o processo pai
- Retorna 0 (zero) para o processo filho

```
#include <iostream>
#include <unistd.h>

int main() {
    if (fork() == 0)
        std::cout<<"Processo filho.\n";
    else
        std::cout<<"Processo pai.\n";
    return 0;
}
```

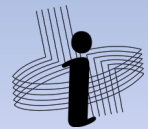
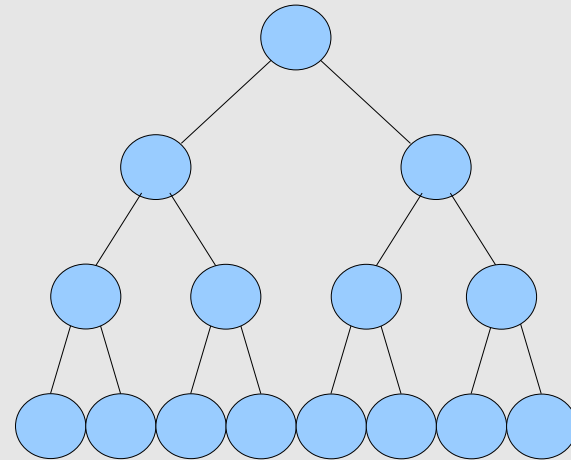


# Fork em sistemas Unix

- Quantos “Hello!” são impressos no código a seguir?

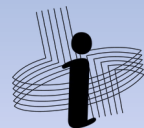
```
#include <iostream>
#include <unistd.h>

int main() {
    fork();
    fork();
    fork();
    std::cout<<“Hello!\n”;
    return 0;
}
```



# O conceito de *thread*

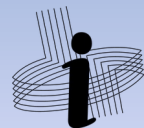
- *Threads* são sequências de instruções dentro de um processo que podem executar de forma concorrente entre si
  - Não há cópia, como acontece no *fork* de um processo
  - Compartilham o espaço de endereçamento (código, variáveis, descritores de E/S, etc)
  - Cada *thread* possui um mini-descritor com as partes específicas (contador de programa, posição de acesso nos arquivos, etc.)
  - Processo fica mais leve (mais eficiente na troca de contexto)



# Threads vs Tasks

---

- Multithreading
  - Surgiu com arquiteturas monoprocessadas
  - Permite que um processo faça mais de uma coisa ao mesmo tempo, cada *thread* realizando uma tarefa específica
  - Melhora latência geral do processo
  - Não escala bem com o número de núcleos
- Paralelismo
  - Divisão do trabalho a ser feito em partes menores (*tasks*)
  - Escala melhor com o número de núcleos
  - SO atribui *tasks* à *threads* e distribui para os núcleos de processamento
  - Melhora performance

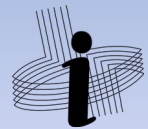




# Utilizando *threads* em C/C++

---

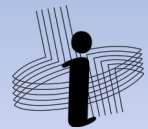
- Abordagem 1
  - POSIX Threads (pthreads): C/C++
- Abordagem 2
  - Classe thread: C++



# PThreads

---

- PThreads = POSIX (*Portable Operating System Interface*) *Threads*
- Windows não é POSIX
  - Existe uma implementação para Windows (pthreads-win32) sem todas as funções originais



# PThreads

- Primeiros passos

- Biblioteca

- ```
#include <pthread.h>
```

- Tipos básicos

- ```
pthread_t tid;          //identificador de uma thread
```

- ```
pthread_attr_t tattr;   //atributos de uma thread
```

- Definindo atributos

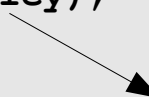
- ```
pthread_attr_init(&tattr);
```

- ```
pthread_attr_setstacksize(&tattr, size);
```

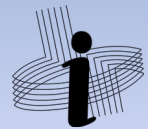
- ```
pthread_attr_setschedpolicy(&tattr, policy);
```

- ```
...
```

- ```
pthread_attr_destroy(&tattr)
```



policy = SCHED\_FIFO,  
SCHED\_RR,  
SCHED\_OTHER (default)



# PThreads

- Gerência de uma *thread*

- Criação

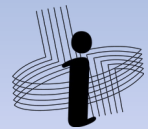
- ```
pthread_t tid;  
pthread_create(&tid, &attr, start_func, arg);
```

- Encerrando (de dentro da thread)

- ```
pthread_exit((void *)value);
```

- Esperando pelo fim de uma thread filha

- ```
void *value;  
pthread_join(tid, &value);
```



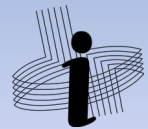
# PThreads

---

- Compilando e executando

```
gcc -o mythread mythread.c -lpthread
```

```
./mythread
```



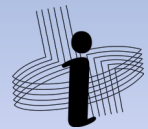
# PThreads

- Exemplo 1a

```
#include <stdio.h>
#include <pthread.h>

void *hello(void *id) {
    for (int i = 1; i <= 50; i++)
        printf("Msg %d - Thread %ld\n", i, (long)id);
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, hello, (void*)1);
    pthread_create(&t2, NULL, hello, (void*)2);
    pthread_exit(NULL); //termina thread main mas não o processo
}
```

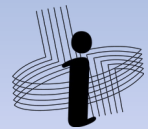


# PThreads

- Exemplo 1a (modificado para capturar retorno da *thread*)

```
...
void *hello(void *id) {
    for (int i = 1; i <= 50; i++)
        printf("Msg %d - Thread %ld\n", i, (long)id);
    long res=(long)id *10;          //cálculo qualquer realizado
    pthread_exit((void*)res);      //retorna id para pthread_join
}

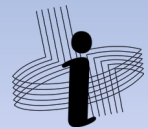
int main() {
    pthread_t t1,t2;
    pthread_create(&t1, NULL, hello, (void*)1);
    pthread_create(&t2, NULL, hello, (void*)2);
    void *a,*b;
    pthread_join(t1, &a);           //espera thread t1 terminar
    pthread_join(t2, &b);           //espera thread t2 terminar
    printf("%ld e %ld retornados\n", (long)a, (long)b);
    pthread_exit(NULL);
}
```



# PThreads

- Exemplo 1b (alocando memória para retorno da função)

```
...
void *hello(void *id) {
    for (int i = 1; i <= 50; i++)
        printf("Msg %d - Thread %ld\n", i, (long)id);
    int* r=(int*)malloc(sizeof(int)); //alocar memória para resultado
    *r=(long)id+1;
    pthread_exit(r);                //retornar ponteiro para resultado
}
int main() {
    pthread_t t1,t2;
    pthread_create(&t1, NULL, hello, (void*)1);
    pthread_create(&t2, NULL, hello, (void*)2);
    void *a,*b;
    pthread_join(t1, &a);
    pthread_join(t2, &b);
    printf("Resultados %d e %d\n", *(int*)a, *(int*)b);
    free(a); free(b);                //liberar
    pthread_exit(NULL);
}
```

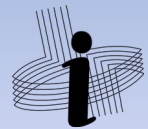




# PThreads

---

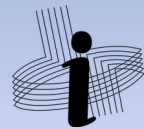
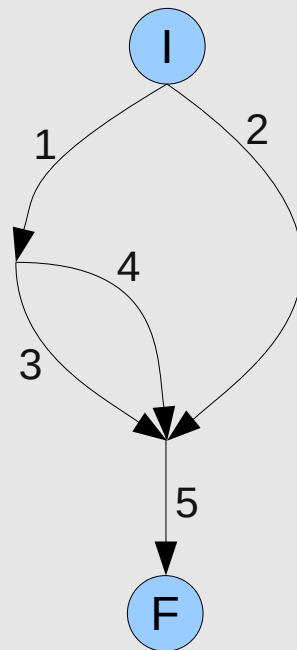
- E se uma *thread* termina antes do *join*?
  - A *thread* não é desalocada até que ocorra o *join* ou *detach*
    - *pthread\_detach* torna uma *thread* “independente” da *thread* criadora
    - Após *detach*, o *join* não pode ocorrer (*joinable == false*)



# Exercício

- Qual é o grafo que mostra a relação de precedência entre as saídas a seguir?

```
...  
void* f(void *i) {  
    printf("%ld ", (long)i);  
}  
  
int main() {  
    pthread_t t2;  
    pthread_create(&t2, NULL, f, (void*)2);  
    f((void*)1);  
    pthread_t t4;  
    pthread_create(&t4, NULL, f, (void*)4);  
    f((void*)3);  
    pthread_join(t4, NULL);  
    pthread_join(t2, NULL);  
    f((void*)5);  
    printf("\n");  
    return 0;  
}
```



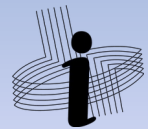
# Exercício

- Escreva um programa em C ou C++ que cria um array contendo  $n$  valores inteiros e calcula a soma deles de forma paralela utilizando *pthread*
  - Cada thread será responsável por somar uma partição (de acordo com seu identificador) e retornar o resultado para a thread principal, que somará os valores recebidos para obter a soma final.
  - Utilize uma constante global para definir o número de threads a serem utilizadas e compare os resultados ao variar esse número.
    - Para ler o número de núcleos do computador, você pode usar `get_nprocs()` da biblioteca `<sys/sysinfo.h>`
  - Você também pode definir o array como global para que as threads possam acessá-lo diretamente.
- Extra:
  - Modifique o código definindo o array como local e faça com que a thread principal o passe como parâmetro para as threads, juntamente com o id.

# Threads em C++

---

- Classe *std::thread*
  - Disponível apenas para C++ a partir da versão 11
  - Faz uso de Pthreads internamente
    - compilação utilizando diretiva `-lpthread`



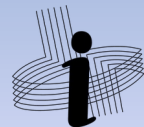
# Threads em C++

- Exemplo 2a

```
#include <iostream>
#include <thread>
using namespace std;

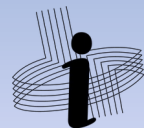
void hello(int id) {
    for (int i = 1; i < 10; ++i)
        cout<<"Thread " <<id<<" - " <<i<<endl;
}

int main() {
    thread t1(hello,1);
    thread t2(hello,2);
    t1.join();          //não existe valor de retorno da thread
    t2.join();
    cout<<"1 e 2 terminaram."<<endl;
    return 0;
}
```



# Threads em C++

- Classe *std::thread*
  - As *threads* não retornam o resultado diretamente para a função *join*
    - Passagem de parâmetro por referência é uma opção
    - Memória compartilhada pode ser mais eficiente se usada corretamente
    - Problemas
      - erro de compilação!
        - » conversão de parâmetro de “int” para “int &”
        - » uso da diretiva *std::ref()* é obrigatório
      - pode facilmente resultar em erros de sincronização
        - » a memória pode ser desalocada enquanto a *thread* ainda está executando
        - » mais de uma *thread* modificando o mesmo endereço de memória



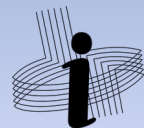
# Threads em C++

- Exemplo 2b

```
#include <iostream>
#include <thread>
using namespace std;

void hello(int &id) {
    cout<<"Hello da thread "<<id<<endl;
    id+=10;
}

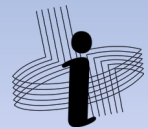
int main() {
    int id1=1;
    int id2=2;
    thread t1(hello,ref(id1)); //uso de ref() obrigatório
    thread t2(hello,ref(id2));
    t1.join();
    t2.join();
    cout<<"1 e 2 terminaram com valores "<<id1<<" e "<<id2<<endl;
    return 0;
}
```



# Threads em C++

- Exemplo 2d (erro no uso de variável compartilhada)

```
...  
void f(int &i){  
    cout<<i<<endl;  
}  
  
int main() {  
    vector<thread> threads;  
    for (int i=1; i<10; i++){  
        threads.push_back(thread(f,ref(i)));  
    }  
    cout<<"main"<<endl;  
  
    for (thread &th:threads) {  
        th.join();  
    }  
    return 0;  
}
```

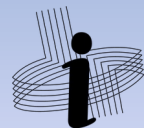




# Threads em C++

- Tipo *std::promise* e *std::future* (Exemplo 3a)

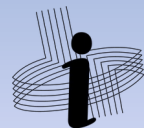
```
...
#include <future>
void hello(int id, promise<int> &&p) {
    for (int i = 1; i < 100; ++i)
        cout<<"Thread "<<id<<" - "<<i<<endl;
    p.set_value(id*10);
}
int main() {
    promise<int> prm1,prm2;
    future<int> ftr1 = prm1.get_future(); //obter o future...
    future<int> ftr2 = prm2.get_future(); //
    thread t1(hello,1,move(prm1));        //e mover promise para a thread
    thread t2(hello,2,move(prm2));
    int r1=ftr1.get();                    //espera que valor seja definido no promise
    int r2=ftr2.get();
    t1.join();
    t2.join();
    cout<<"1 e 2 terminaram com valores "<<r1<<" e "<<r2<<endl;
    return 0;
}
```



# Threads em C++

---

- Tipo *std::async*
  - Simplifica a criação de *tasks* paralelas
  - Indicado apenas quando
    - não há necessidade de sincronização com outras *threads*
    - uso de poucas *threads* que vão executar durante tempo relativamente longo
  - Possui maior *overhead* que *std::thread*
    - dificulta aproveitamento de *threads* (*thread pool*) forçando frequentes *creates* e *joins*
    - criação de novas *threads* tem custo relativamente alto



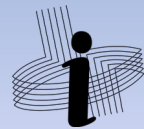
# Threads em C++

- Tipo *async* (Exemplo 3b)

```
...
#include <future>

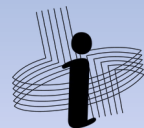
int hello(int id) {                                //função normal retornando int
    for (int i = 1; i < 100; ++i)
        cout<<"Thread "<<id<<" - "<<i<<endl;
    return(id*10);
}

int main() {
    //Future utilizado para obter retorno da task
    future<int> ftr1 = async(hello,1);
    future<int> ftr2 = async(hello,2);
    //Bloquear e esperar que o valor seja retornado pela função
    int r1=ftr1.get();
    int r2=ftr2.get();
    cout<<"1 e 2 terminaram com valores "<<r1<<" e "<<r2<<endl;
    return 0;
}
```



# Threads em C++

- Diferentes modos de passar argumentos para *threads*
  - Por valor:
    - seguro, mas pode ser custoso
  - Usando *move*: modifica ponteiros para que apenas a *thread* filha tenha acesso
    - seguro, mas para objetos complexos será necessário certificar que objetos internos também sejam movidos
  - Por referência (usando *const*): evita que a *thread* filha faça alterações
    - seguro, desde que se garanta que a *thread* mãe (ou outras *threads* com acesso) não fará modificações
  - Por referência (sem *const*):
    - são necessárias formas de garantir que outras *threads* não tenham acesso para escrita



# Threads em C++

- Medindo desempenho com *std::chrono*
  - Biblioteca que oferece relógio de alta precisão

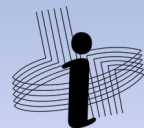
```
#include <chrono>

int main() {
    // ...
    std::chrono::time_point<std::chrono::high_resolution_clock> tp1, tp2;
    tp1=std::chrono::high_resolution_clock::now();

    //trecho de código a ser avaliado

    tp2=std::chrono::high_resolution_clock::now();
    double tempo=std::chrono::duration<double, std::ratio<1, 1000>>
                                   (tp2-tp1).count();

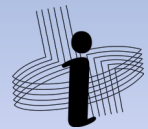
    printf("tempo: %fms\n", tt); //ratio<1, 1000> = tempo em milisegundos
    // ...
}
```



# Threads em C++

- *std::thread vs boost::thread*

|                                                       | std::thread                               | boost::thread                                              |
|-------------------------------------------------------|-------------------------------------------|------------------------------------------------------------|
| disponibilidade                                       | padrão C++11                              | fornecido por terceiro, aceita versões mais antigas do C++ |
| async                                                 | sim                                       | não                                                        |
| shared_mutex<br>(N-readers-1-writer)                  | a partir de C++17                         | sim                                                        |
| future                                                | sim                                       | sim, com o nome de boost::unique_future                    |
| Quando o código termina sem chamar join() ou detach() | chama std::terminate() e aborta aplicação | chama boost::detach()                                      |



# Utilizando *threads* em Python

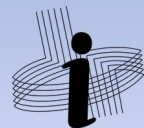
- Pacote *threading*
- Classe *Thread*
  - Argumento deve ser passado através de objeto iterável

```
t=threading.Thread(target=nome_funcao,args=(arg1,arg2,...,argN))
```

- *Thread* não inicia imediatamente após criação
- Principais funções

```
t.start() # inicia a thread t  
t.join()  # espera a thread t
```

- *join* não tem retorno



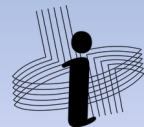
# Utilizando *threads* em Python

- Exemplo 4a

```
import threading
import time

def countdown(count):
    for i in range(count, -1, -1):
        print("Counting down", i)
        time.sleep(1)

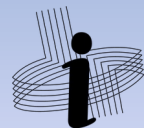
t1 = threading.Thread(target=countdown, args=(10,))
t2 = threading.Thread(target=countdown, args=(20,))
t1.start()
t2.start()
t1.join()
t2.join()
print("Contagens terminadas")
```





# Utilizando *threads* em Python

- Apesar de simples, a implementação de paralelismo em Python é ineficiente
  - GIL = *Global Interpreter Lock*
  - Uso ineficiente dos núcleos do sistema
- Biblioteca *multiprocessing* é uma alternativa
  - Baseada na criação de diversos processos
  - Perde as vantagens do conceito de *thread*
- Outras
  - asyncio
  - Cython
  - Celery
  - ...



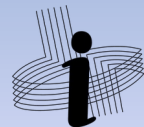
UFV

# Exemplos de concorrência

- Compartilhamento de um procedimento

```
...  
void f(int id) {  
    int i;  
    for (i = 1; i < 10; ++i)  
        cout<<id;  
}  
  
int main() {  
    thread t1(f,1);  
    thread t2(f,2);  
    t1.join();  
    t2.join();  
    cout<<endl;  
    return 0;  
}
```

São possíveis, teoricamente,  
 $20! / (10! * 10!)$  ou 184.756  
combinações diferentes.



# Exemplos de concorrência

- Compartilhamento de variável

```
...  
int s=0;  
  
void f() {  
    for (int i = 0; i < 10; ++i)  
        s=s+1;  
}  
  
int main() {  
    thread t1(f);  
    thread t2(f);  
    t1.join();  
    t2.join();  
    cout<<s<<endl;  
    return 0;  
}
```

Quais os valores possíveis de S?  
Resp: entre 2 e 20

