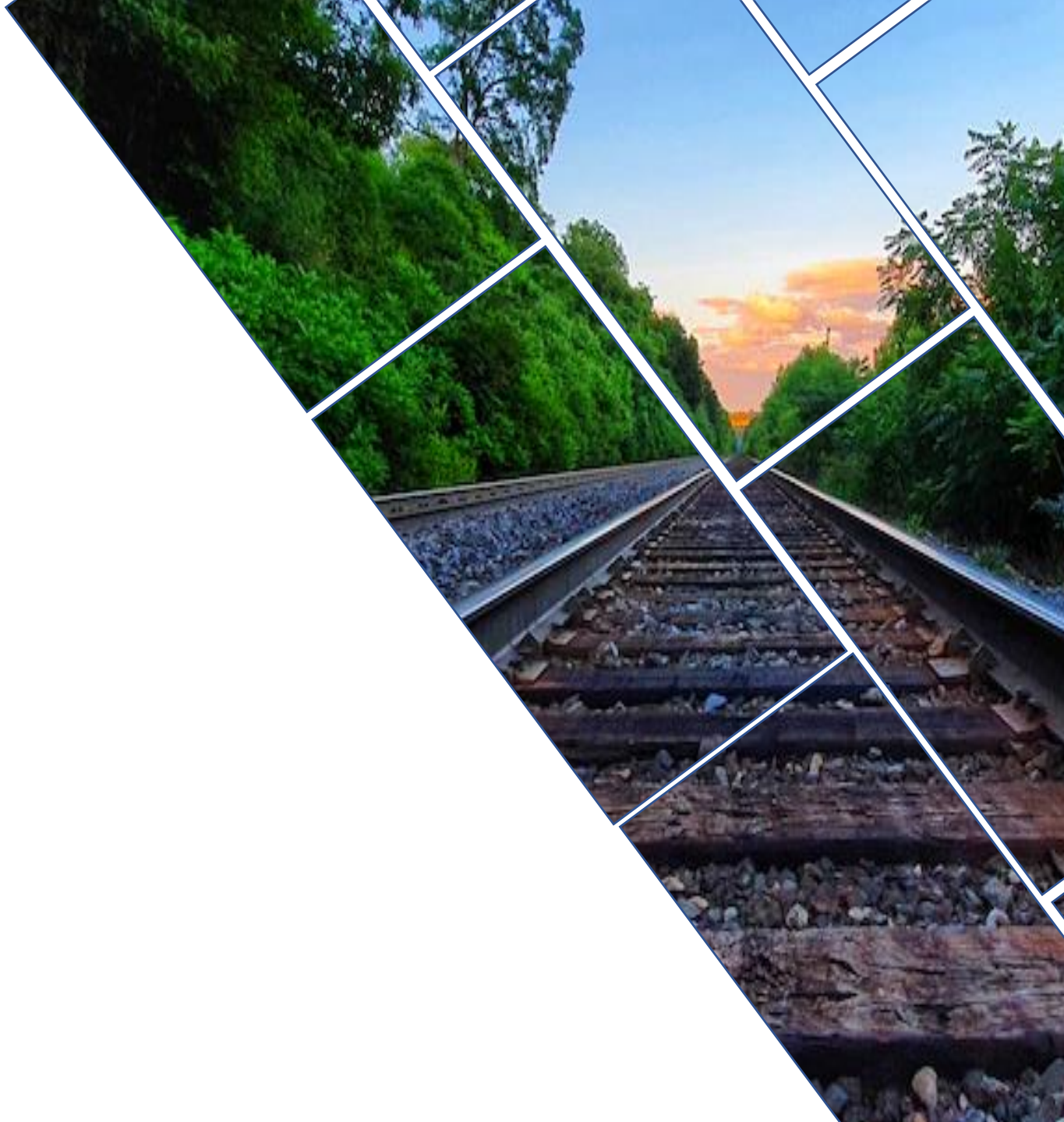
Performance em Sistemas Ciberfísicos

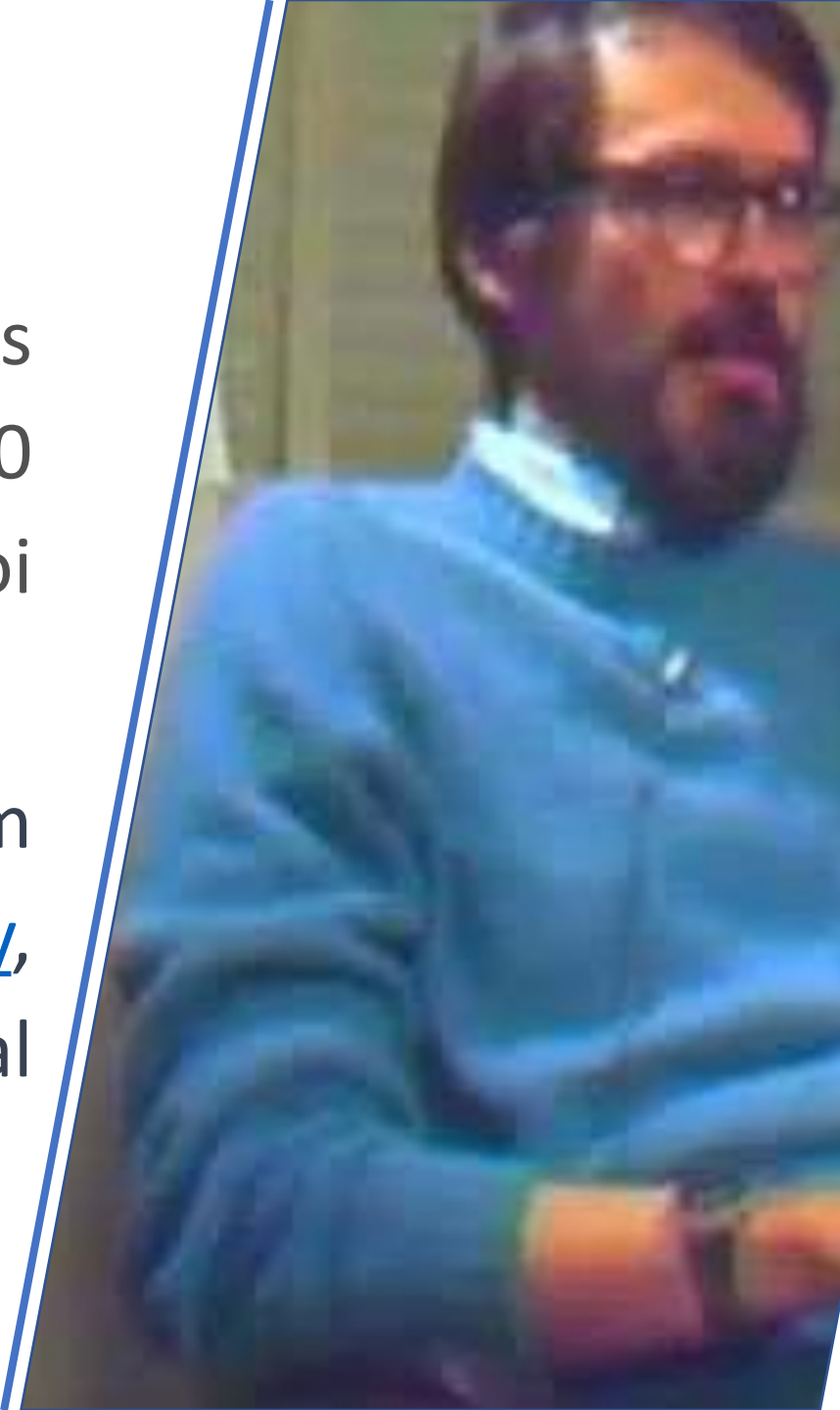
Aula 9 – Threads



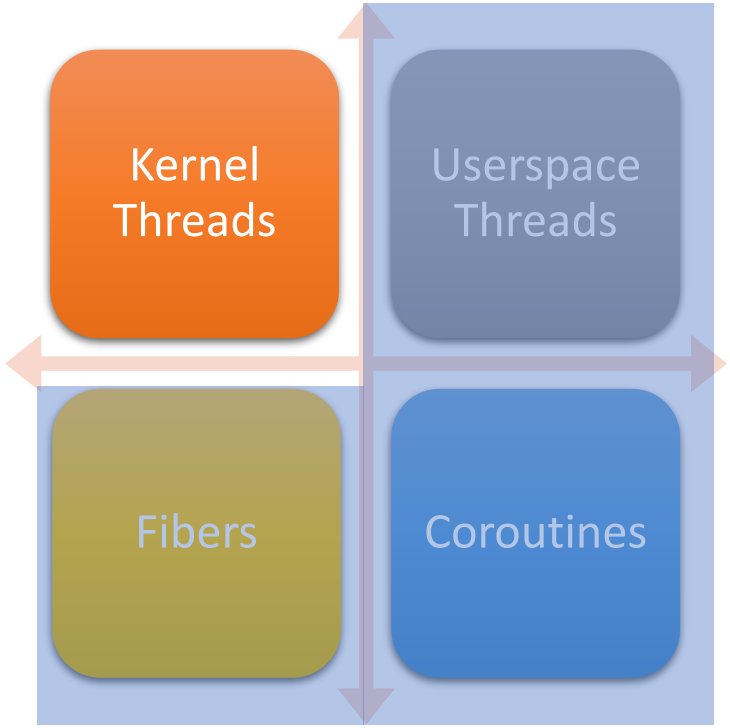
Frank Coelho de Alcantara – 2023 -1

Um Pouco de História

- O termo *Thread* pode ser traçado até os anos 1960 até os Sistemas IBM/360 inicialmente este artefato de código foi chamado apenas de *Tasks*.
- Creditamos o termo *thread* um pesquisador chamado Victor A. Vyssotsky, em conjunto com Robert Morris, no final da década de 1960.

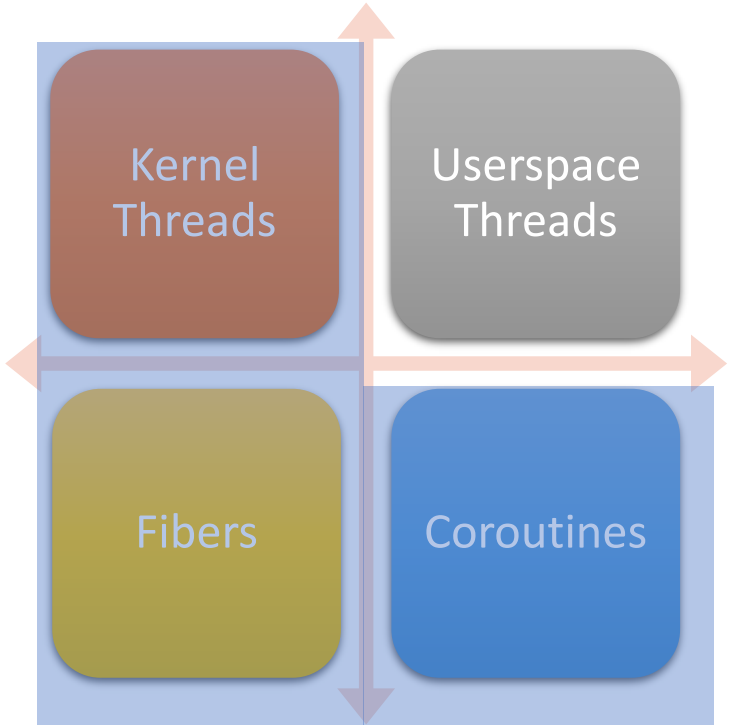


Quatro Conceitos



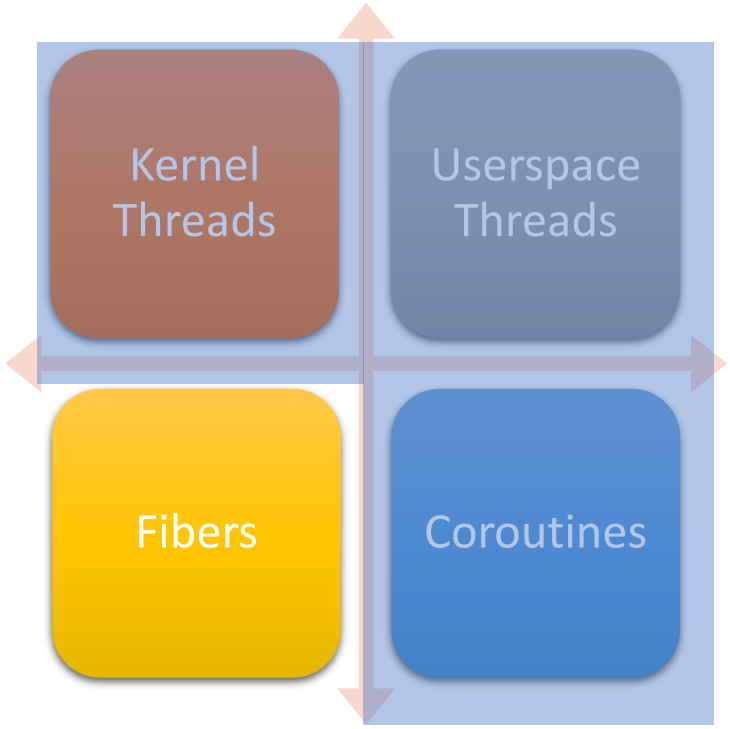
- **Kernel Threads**: são gerenciados diretamente pelo kernel do sistema operacional. Cada *kernel thread* é uma entidade independente, com seu próprio contexto de execução. O kernel é responsável por escalonar e agendar a execução desses *threads* diretamente no nível do sistema operacional. Os kernel threads têm acesso direto aos recursos do sistema operacional, como dispositivos de E/S e memória.

Quatro Conceitos



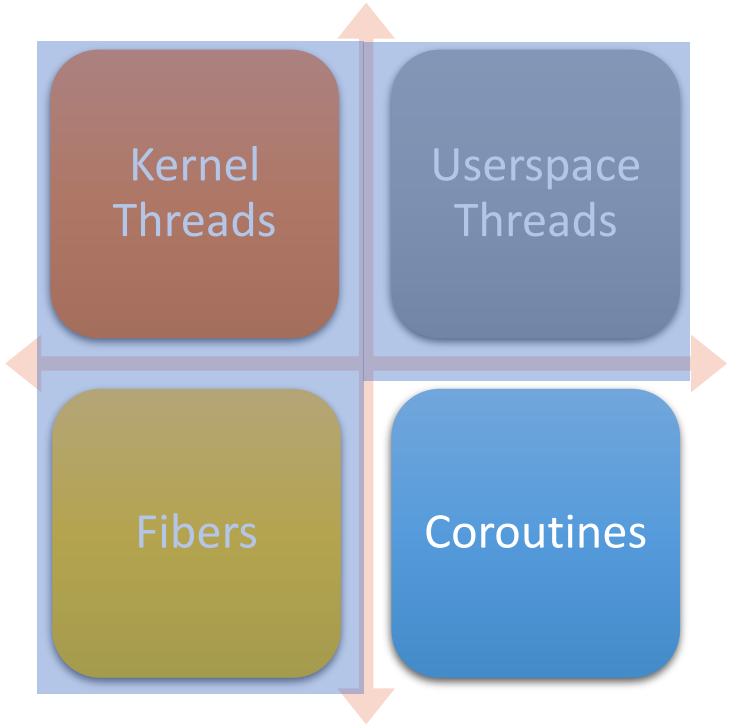
- **User-Space Threads**: ou threads em espaço de usuário são threads gerenciados em nível de usuário, sem intervenção direta do kernel do sistema operacional. Nesse modelo, o escalonamento e o agendamento das threads são implementados por uma biblioteca ou por um ambiente de tempo de execução em nível de usuário, em vez de depender do kernel. Essas threads são mais leves e têm menor sobrecarga se comparados com kernel threads.

Quatro Conceitos



- ***Fibers***: unidades de execução cooperativas, também conhecidas como *stackful coroutines*. Diferente dos threads tradicionais, as *fibers* não são escalonadas pelo sistema operacional. Em vez disso, elas são controladas pelo programa em si. As *fibers* têm um contexto de execução mais simples em comparação com as *threads*, pois não requerem uma pilha de execução separada.

Quatro Conceitos



- **Coroutines**: são unidades de execução cooperativas semelhantes às *fibers*, porém com um modelo de programação mais abstrato. Diferentemente das *threads* e *fibers*, que geralmente seguem um modelo de concorrência preemptiva, as *coroutines* seguem um modelo de concorrência cooperativa. As coroutines cooperam entre si para decidir quando ceder o controle de execução, em vez de serem interrompidas pelo sistema operacional.

Threads

- Um *thread*, *fluxo de execução*, é a menor unidade de processamento que pode ser gerenciada pelo sistema operacional (SO).
- Trata-se de uma sequência de instruções definidas para execução em série em um núcleo de processamento diferente do núcleo de processamento do processo principal. *Threads* de um mesmo processo compartilham o mesmo espaço de memória e recursos do sistema.

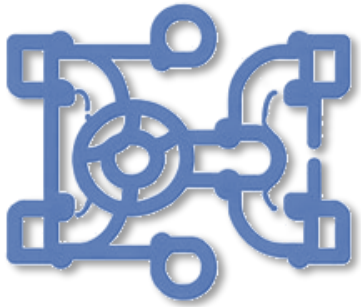
Compartilhamento de Recursos entre Threads

Espaço de memória: Todos os threads de um processo compartilham o mesmo espaço de endereço de memória. Todos têm acesso as variáveis globais no heap. Facilita a comunicação e cria problemas.

Recursos do sistema: descritores de arquivos, que podem representar arquivos abertos, soquetes de rede, pipes, etc.

Informações do processo: ID do processo, sinais e tratadores de sinais, e contadores de uso de recursos.

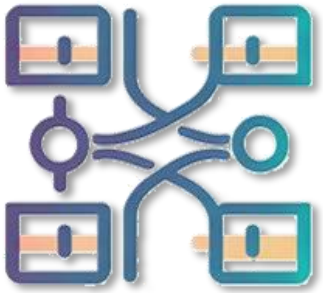
Conceitos



Pipes

Soquetes

Sinais



Contadores de Recursos

Compartilhamento de Recursos

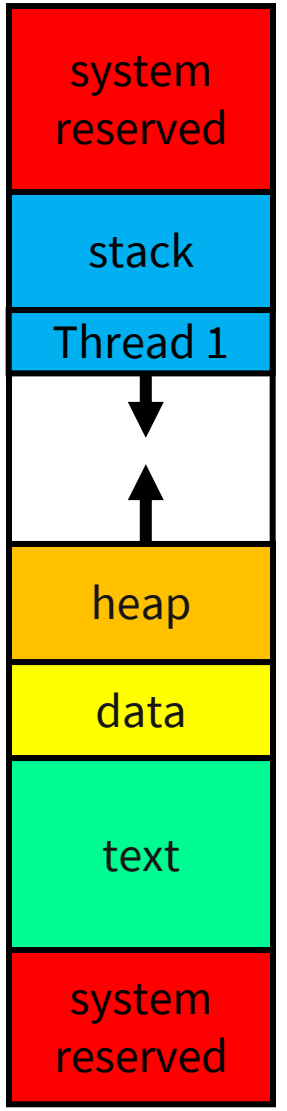
- ***Handles* para objetos do sistema operacional**: *threads* compartilham *handles* para objetos como semáforos, eventos, mutexes e variáveis de condição, que podem ser usados para sincronização entre threads.
- *handle* é um termo usado para representar uma referência a um recurso disponível. Esses recursos podem incluir artefatos diversos como: arquivos, soquetes de rede, conexões de banco de dados, objetos e *threads* e processos.

Threads e Processos

- Um processo é uma instância de um programa em execução e tem seu próprio espaço de memória alocado, um *thread* é um artefato de código deste processo que pode ser executado separadamente. Todos os *threads* de um mesmo processo compartilham o espaço de memória do processo.
- Um *thread* pode escrever em uma variável ou estrutura de dados na memória e outros, do mesmo processo podem ler essa informação, sem a necessidade de mecanismos de comunicação entre processos.

Voltando ao *Stack*

- Cada *thread* em um processo tem seu próprio *stack*, que é usada para armazenar variáveis locais e informações de controle de chamadas de função. Cada uma destes contém a sua própria lista de *stack frames.*
- Cada *thread* contém seu próprio *conjuntos de* registros de instruções, mas todas as *threads* compartilham o restante do espaço de memória do processo (*heap*, *data* e *text*)



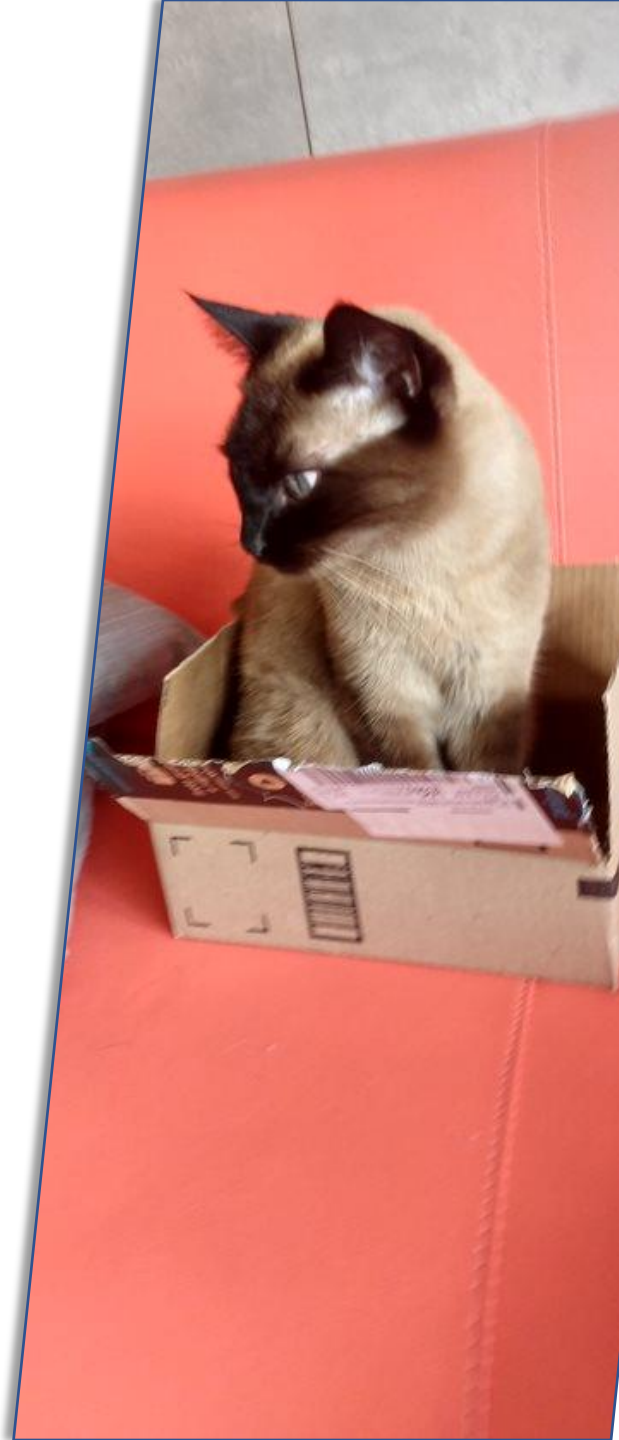
Comparando com os Processos

- A criação e a destruição de um processo requerem uma quantidade significativa de trabalho do sistema operacional, incluindo a alocação de endereçamento, a cópia de variáveis de ambiente, a inicialização de recursos.
- **Os *threads* são mais leves**, o que significa que consomem menos recursos do sistema operacional. Permite que o sistema crie e gerencie um número maior de *threads* do que seria capaz de fazer com processos. A criação e a destruição de threads são operações mais rápidas.

Comparando com os Processos – Mais Leves

- **Compartilhamento de Recursos**: Os *threads* de um mesmo processo compartilham um espaço de endereçamento e os mesmos recursos. Não há necessidade de alocar e inicializar um novo espaço de endereçamento para cada *thread*.
- **Mudança de Contexto mais Rápida**: mudanças de contexto entre *threads* são rápidas por não requerem uma mudança no espaço de endereçamento. A mudança de um processo para outro requer que o SO altere o mapa de memória.

Threads e o Sistema Operacional



Criação de Threads Pelo Sistema Operacional

Aloca a estrutura de controle do *thread* (TCB - *Thread Control Block*), que contém informações sobre o estado e o contexto do thread.

Aloca um novo *stack* para o *thread*.

Define o ponto de entrada do *thread* (a função que o *thread* executará).

Adiciona o *thread* ao Agendador de Tarefas (escalonador), que decidirá quando e onde o *thread* será executado.

Agendamento de Execução - Escalonamento

- O agendamento de *threads* é o processo pelo qual o sistema operacional decide qual *thread* deve ser executado em um dado momento. Existem políticas de agendamento, incluindo *round-robin*, *FCFS*, por prioridade e outros.
- O Agendador de Tarefas pode ser preemptivo (onde o sistema operacional pode interromper um thread para dar tempo de CPU a outro thread) ou não preemptivo (onde um thread continua a ser executado até que se bloqueie ou ceda voluntariamente o controle).

Estratégias de Agendamento - 1

- ***First-Come, First-Served (FCFS)***: o primeiro que solicita uma CPU é o primeiro a obtê-la. Pode levar ao efeito comboio, onde um processo longo pode fazer com que outros processos tenham que esperar muito tempo.
- ***Shortest Job Next (SJN) / Shortest Job First (SJF)***: o algoritmo seleciona o processo com o menor tempo de execução. Minimiza o tempo de espera médio. Pode ser difícil de implementar porque o tempo de execução de um processo raramente é conhecido.

Estratégias de Agendamento - 2

- ***Round Robin (RR)***: Este algoritmo atribui a cada processo um quantum de tempo igual. Quando o quantum de um processo acaba, ele é enviado de volta à fila.
- **Prioridade**: Este algoritmo atribui a cada processo uma prioridade e sempre escolhe o processo com a maior prioridade para executar a seguir. A questão da inanição (*starvation*) pode surgir se alguns processos sempre tiverem uma prioridade mais baixa do que os outros.

Estratégias de Agendamento - 3

- **Multinível Feedback Queue (MFQ)**: usa várias filas com diferentes prioridades e a ideia de envelhecimento para prevenir a inanição.
- Os processos são inicialmente colocados na fila de maior prioridade, mas se usarem muito tempo de CPU, são rebaixados para uma fila de menor prioridade.
- Se esperarem muito tempo sem execução (envelhecerem), são promovidos a uma fila de maior prioridade.

Sincronização

- Como os *threads* de um processo compartilham o mesmo espaço de memória, eles precisam coordenar o acesso a recursos compartilhados para evitar condições de corrida.
- O sistema operacional fornece várias primitivas de sincronização para isso, incluindo mutexes, semáforos e variáveis de condição.
- A sincronização pode implicar que um determinado *thread* tenha que esperar antes de realizar seu processamento.

Threads Esperando

- Quando um *thread* está esperando, ou hibernando, por uma mudança de estado em um mutex, semáforo, ou variável condicional, ele entra em um estado de bloqueio.
- Um thread em um estado de espera não está ativamente verificando a condição de espera.
- Em hibernação o *thread* será notificado pelo sistema operacional quando a condição for atendida. Isso é mais eficiente que ficar constantemente verificando a condição.

Estados dos Threads Fora de Execução

Bloqueio: não é executado. Não consome tempo de CPU e não avança em sua execução.

Desescalonado: O SO remove o *thread* do conjunto de *threads* prontos para executar. Liberando CPUs.

Espera: permanece em um estado de espera até que a condição que ele está esperando seja atendida.

Despertar: Quando a condição é atendida o *thread* é colocado de volta no conjunto de *threads* prontos.

Rescalonamento: quando escolhido o *thread* retoma sua execução do ponto em que parou.

Threads também são Destruídos

- Quando um thread termina, o sistema operacional precisa limpar seus recursos.
- Desalocar a estrutura de controle do thread e a stack, e possivelmente ajustar o escalonador.
- Se um processo não limpar adequadamente seus threads terminados, pode ocorrer um vazamento de recursos, onde os recursos não são recuperados e acabam sendo desperdiçados.

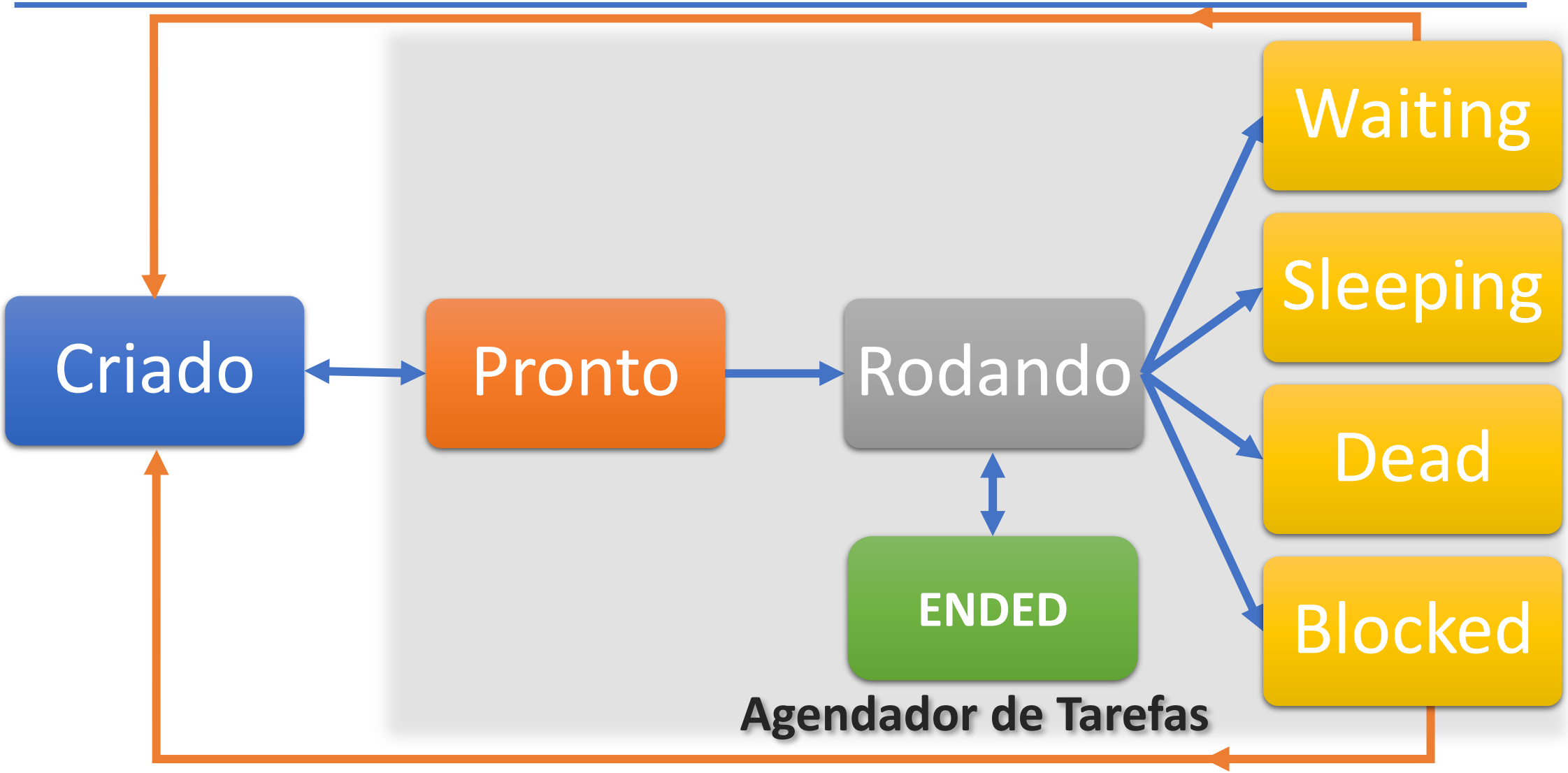
Estratégia de Desalocação de um Thread

1. **Finalização do Thread**: O *thread* indica ao sistema operacional que concluiu sua execução. Isso ocorre com: o thread retornando de sua função principal; chamando explicitamente uma função de término ou sendo cancelado por outro thread.
2. **Desalocação da Estrutura de Controle do Thread**: A estrutura chamada de ***Thread Control Block*** (TCB), mantém informações sobre o *thread*, como seu estado, prioridade, contexto de execução.

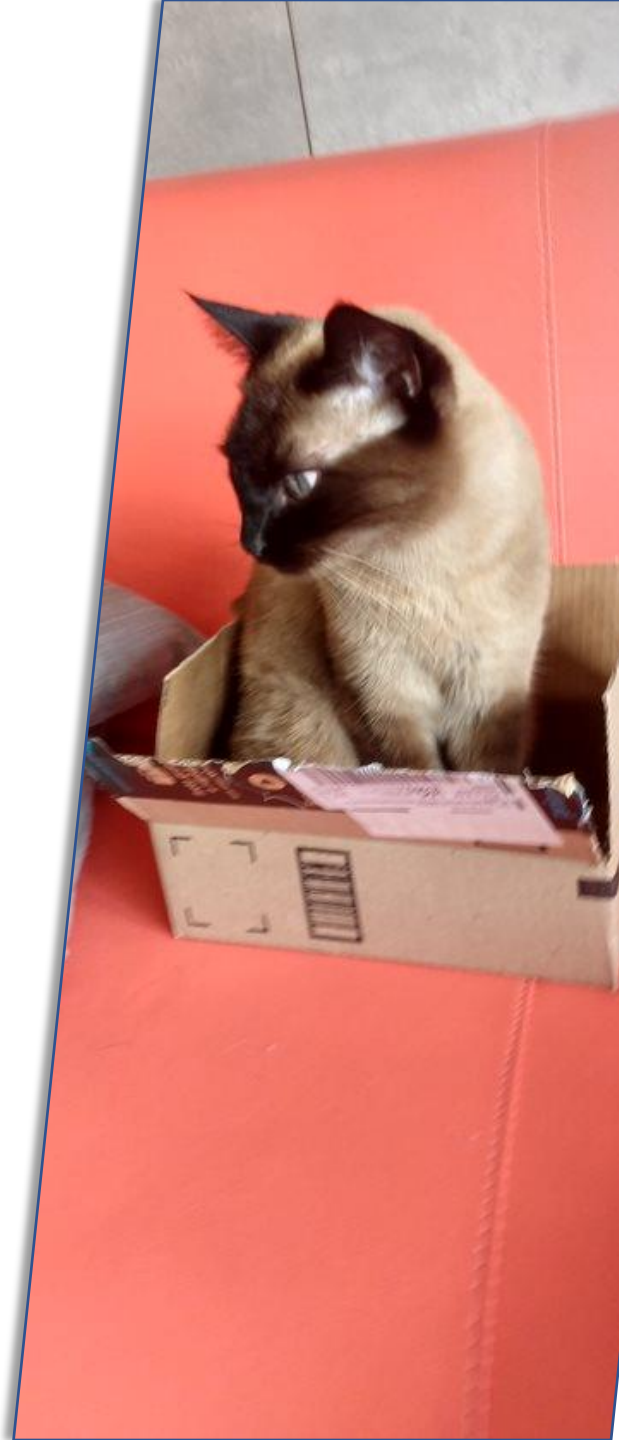
Estratégia de Desalocação de um Thread

3. **Desalocação da Stack do Thread**: Cada thread tem sua própria pilha de memória, usada para armazenar variáveis locais e pontos de retorno. Quando o *thread* termina, a memória alocada para a pilha do **thread** é liberada.
4. **Atualização do Agendador de Tarefas**: atualizado para refletir o fato de que o thread terminou. Isso pode envolver a remoção do thread de quaisquer filas de escalonamento e a possibilidade de escalonar outros *threads* que estavam esperando por recursos que o *thread* terminado estava usando.

Ciclo de Vida em um Diagrama



Threads em Python



Threads em Python

- O módulo *threading* em Python fornece uma maneira de criar *threads* e controlar sua execução.
- **O Problema do Global Interpreter Lock (GIL)**. O GIL é um mecanismo que permite que apenas um *thread* execute código Python por vez em cada processo, mesmo em sistemas com múltiplos núcleos de CPU. Isso limita a eficácia dos *threads* para tarefas que exigem muita capacidade computacional.

O Global Interpreter Lock

- O GIL existe devido à forma como o CPython gerencia a memória. Este gerenciador de memória que não é *thread-safe*. O GIL é necessário para prevenir que os *threads* Python acessem simultaneamente o mesmo objeto Python, o que poderia levar a problemas como condições de corrida. Isso significa que, os threads Python podem ser úteis para tarefas de IO-bound, como leitura de disco, e impressão em tela, eles são pouco úteis para tarefas que são CPU-bound.

Criando Threads em Python

- Usamos a classe *Thread* do módulo *threading*. Para criar um novo thread, você inicializa uma instância de *Thread* e fornece uma função alvo, que será executada pelo thread.

```
# Cria um novo thread que executa a função print_hello
thread = threading.Thread(target=print_hello, args=("Alice",))
```

- Você pode passar argumentos para a função alvo por meio do parâmetro args ou kwargs na inicialização do thread.

```
# Cria um novo thread que executa a função print_greeting
thread = threading.Thread(target=print_greeting, kwargs={"greeting": "Good morning", "name": "Bob"})
```

Inicializando e Finalizando *Threads* em Python

- **thread.start():** o método start() é o que realmente instrui o sistema operacional a executar o *thread* de forma independente. Após o *thread* ser iniciado com start(), o interpretador Python passa a executar a função alvo desse thread em paralelo com o restante do programa.
- **thread.join()**: quando você chama join() em um thread, o fluxo de execução do programa principal é bloqueado até que o thread especificado termine. É uma forma de garantir que o programa principal não prossiga até que as operações em outro thread sejam concluídas.

Um Exemplo Simples

Vamos ver o código de um programa que abre dois *threads*. Em um deles imprimimos um algarismo entre **0 e 9** e no outro uma letra entre **a e j**.

```python
import threading
import time

def print_numbers():
    for i in range(10):
        time.sleep(0.1) #atrasar o thread
        print(i)

def print_letters():
    for letter in 'abcdefghij':
        time.sleep(0.15) #atrasar o thread
        print(letter)

# Criando threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Iniciando threads
t1.start()
t2.start()

# Esperando os threads terminarem
t1.join()
t2.join()
```

Um Exemplo Simples

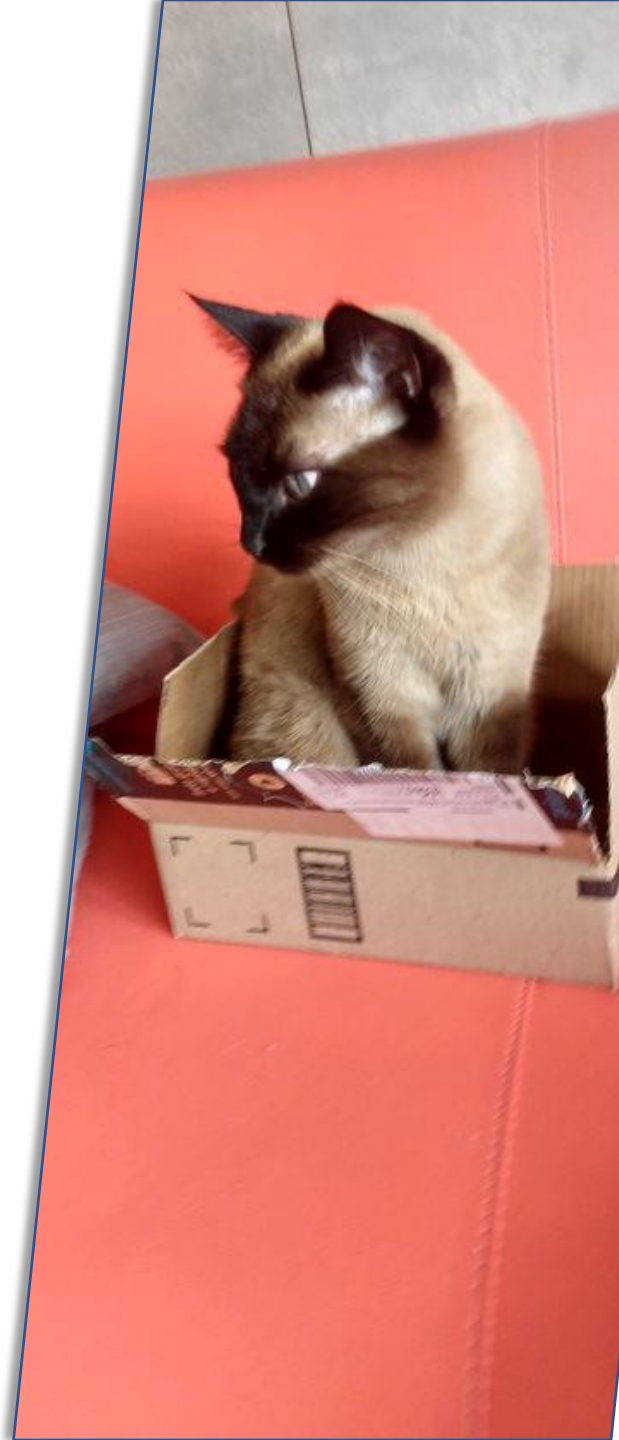
```
1   import threading
2   import time
3
4   def print_numbers():
5       for i in range(10):
6           time.sleep(0.1) #atrasar o thread
7           print(i)
8
9   def print_letters():
10      for letter in 'abcdefghij':
11          time.sleep(0.15) #atrasar o thread
12          print(letter)
13
```

Um Exemplo Simples

```
13
14  # Criando threads
15  t1 = threading.Thread(target=print_numbers)
16  t2 = threading.Thread(target=print_letters)
17
18  # Iniciando threads
19  t1.start()
20  t2.start()
21
22  # Esperando os threads terminarem
23  t1.join()
24  t2.join()
```

Link para o Código

Exercício Prático



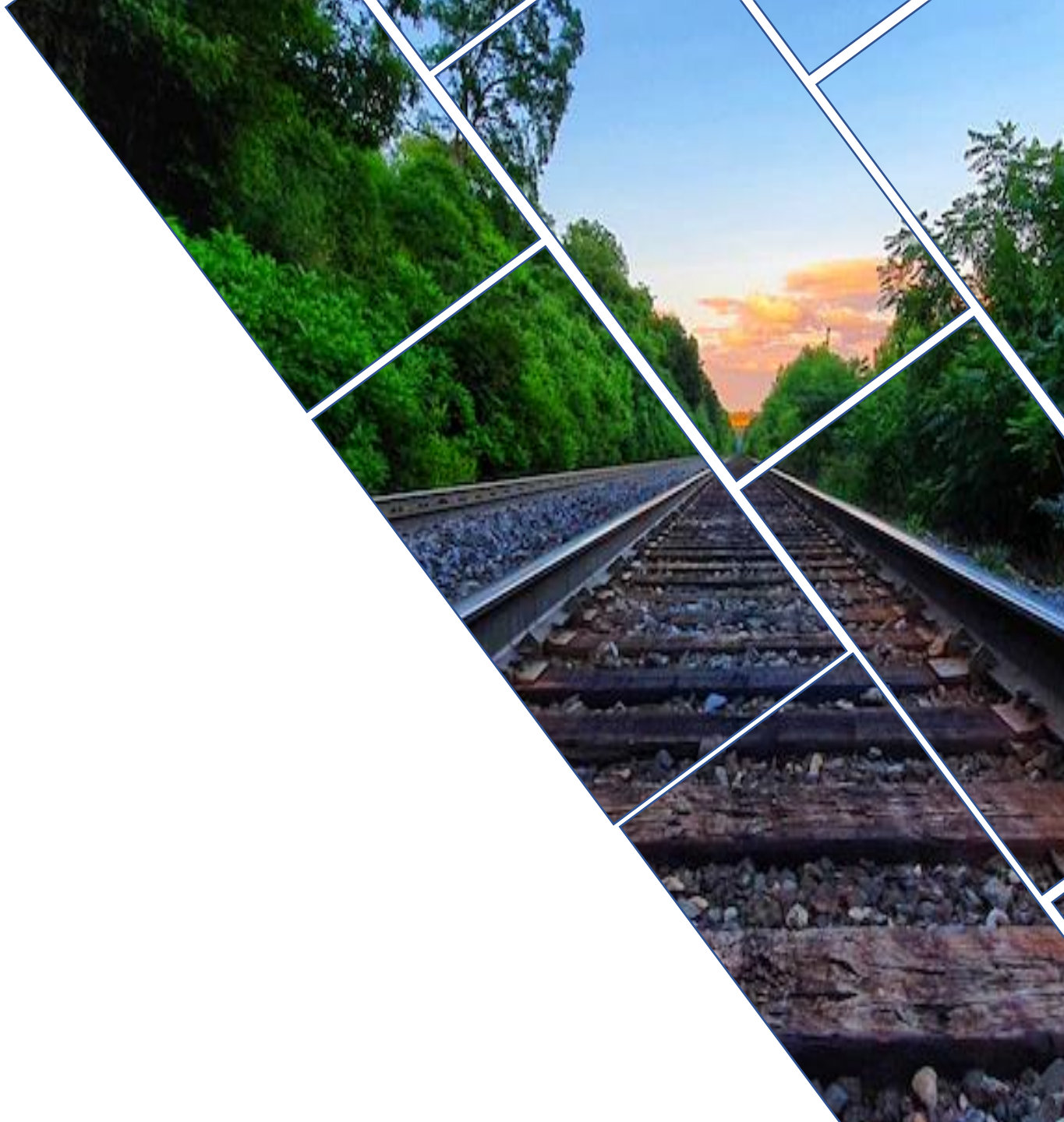
Web Scraping com Python

- Vocês foram contratados por uma empresa de *web scraping* que precisa baixar muitos dados de websites. Eles querem que vocês acelerem o processo usando *threads* em Python.
- Para esta tarefa, vocês devem criar um *script* que baixe o conteúdo de várias páginas da **Wikipedia** ao mesmo tempo, cada uma em um *thread* separado. O conteúdo de cada página deve ser escrito em um arquivo com o nome da página.

Web Scraping com Python - Regras

- Vocês devem usar as bibliotecas *threading* e *request* do Python.
- Cada página da Wikipedia deve ser baixada em um *thread* separado.
- Vocês devem provar que os *threads* estão ocorrendo em concorrência. Para isso, insira declarações de impressão (*print*) em seus *threads* para mostrar quando cada *download* começa e termina. As páginas da Wikipedia a serem baixadas são: [*'Web scraping', 'Data science', 'Python (programming language)', 'Thread (computing)', 'Concurrency (computer science)'*]
- O *script* deve ser capaz de ser executado no Google Colab.

Obrigado!



Frank Coelho de Alcantara – 2023 -1

Sinais - 1

- Os sinais são uma forma de comunicação entre processos em um sistema operacional. São usados para notificar um processo ou um *thread* específico sobre um evento particular.
- Por exemplo:
  - **SIGTERM**: usado para indicar que um processo precisa ser encerrado;
  - **SIGFPE**: usado para indicar que um processo está tentando realizar uma operação ilegal, como dividir por zero;
  - **SIGALRM**: que um temporizador definido pelo processo expirou.

Sinais - 2

- SIGINT: Este sinal é enviado quando o usuário pressiona Ctrl+C no teclado. O propósito deste sinal é interromper a execução de um processo.
- SIGSEGV: Este sinal é enviado quando um programa tenta acessar um local de memória que não está disponível para ele. Este sinal geralmente resulta em um erro de "segmentation fault" e o encerramento do processo.
- SIGKILL: Este sinal é usado para forçar a terminação de um processo. É uma forma definitiva de encerrar um processo e não permite que o processo faça limpezas antes de ser encerrado.

voltar

Contadores de Recursos

- Os contadores de uso de recursos são usados pelo sistema operacional para monitorar o uso de recursos do sistema por um processo.
- Cada thread contribui para os contadores de uso de recursos do processo como um todo. Por exemplo, se um processo tem vários threads e cada thread usa uma quantidade de tempo de CPU, o contador de uso de recursos do processo irá refletir o tempo de CPU usado por todos os threads. Eventualmente podemos rastrear os *threads* de forma individual.

voltar

Handles

- Quando um programa solicita um recurso do sistema operacional (como abrir um arquivo ou criar um novo processo), o sistema operacional retorna um *handle* para esse recurso.
- Quando você cria um novo processo ou thread, o sistema operacional retorna um handle de processo ou handle de thread. Os *Handles* podem ser compartilhados entre *threads.*

voltar

Inter Process Communication (IPC)

Permitem a troca de dados entre processos, quando processos precisam cooperar para realizar tarefas, sincronizar atividades.

Pipes e Sockets: transferência de dados entre processos, localmente ou através de uma rede.

Filas de Mensagens: permitem que os processos enviem e recebam blocos de dados uns dos outros.

Memória Compartilhada: Um bloco de memória que pode ser acessado por processos diferentes.

Pipes

- Forma de Comunicação que permite a transferência de dados entre dois ou mais processos. Funcionam no modelo de produtor-consumidor, onde um processo escreve no pipe (produtor) e outro processo lê esses dados (consumidor).
- **Pipes Anônimos**: entre processos relacionados, como um processo pai e seus processos filhos. Unidirecional.
- **Pipes Nomeados (ou FIFOs)**: processos não relacionados. Usados para comunicação bidirecional. Identificados por um nome de arquivo no sistema de arquivos.

voltar

Soquetes

- Pontos de contato em comunicação que permitem a interação entre processos em uma rede e podem suportar diversos protocolos, como TCP/IP e UDP.
- **Comunicação Bidirecional**: permitem a comunicação bidirecional.
- **Comunicação entre Processos (IPC)**: soquetes também podem ser usados para IPC no mesmo sistema.
- **Comunicação entre Máquinas**: em uma rede, troca de dados entre diferentes sistemas e plataformas.

voltar

Contexto Computacional

- **Contexto:** se refere ao estado da máquina em um thread ou processo em um determinado momento.
- **Registros de CPU:** inclui o ponteiro de instrução; registros gerais e outros registros especiais.
- **Stack de Thread**: usado para armazenar variáveis locais, informações de chamada de função, etc.
- **Estado de Recursos do Sistema**: quaisquer arquivos abertos, soquetes de rede, que o thread, ou processo.

Contexto Computacional – Memória do Kernel

- **No Windows**: é chamada de CONTEXT. Ela é definida pela API do Windows e contém os valores dos registros de CPU, os ponteiros de pilha e frame, a flag de controle e outros componentes do estado de execução.
- **No Linux**: uma estrutura chamada *task_struct* que contém informações sobre o estado do processo ou *thread*, incluindo os valores dos registros de CPU, a pilha do *kernel*, informações sobre a memória virtual e muito mais.

voltar

Condição de Corrida e Semáforos

- **Condição de Corrida**: ocorre quando dois ou mais threads manipulam uma variável compartilhada simultaneamente. O resultado depende da ordem de execução dos *threads*.
- **Semáforos**: artefato de sincronização que pode ser usado para controlar o acesso a recursos compartilhados. Um semáforo, um valor inteiro que pode ser incrementado ou decrementado. Se o *thread* tenta decrementar o semáforo e o valor do semáforo é zero, o *thread* é bloqueado até que outro *thread* incremente o semáforo.

voltar

Mutexes e Variáveis Condicionais

- **Mutexes - exclusão mútua**: usados para sincronização que pode garantir que apenas um *thread* acesse um recurso compartilhado de cada vez. Um *thread* deve travar o mutex antes de acessar o recurso e destrava o mutex quando terminar. Se o mutex está travado o thread é bloqueado.
- **Variáveis Condicionais**: usada para sinalizar alterações de estados que outros *threads* podem estar esperando. As variáveis condicionais são geralmente usadas com mutexes para evitar condições de corrida.

voltar

Thread Control Block (TCB) - 1

- **Identificador de Thread (Thread ID)**: Um valor único que identifica o thread dentro do sistema operacional.
- **Estado do Thread**: O estado atual do thread, que pode ser novo, pronto, executando, esperando ou terminado.
- **Contexto do Processador**: Informações sobre o estado atual do processador para este thread. Isso inclui o valor do contador de programa, os valores dos registradores do processador e a pilha de chamadas.

Thread Control Block (TCB) - 2

- **Prioridade do Thread**: *threads* com maior prioridade são normalmente escalonados para executar antes de threads com prioridade menor.
- **Informações de Escalonamento**: Informações usadas pelo algoritmo de agendamento do SO. Por exemplo, o tempo que o thread gastou e a última vez que foi executado.
- **Recursos de Sistema Alocados**: lista de recursos como arquivos abertos ou conexões de rede, usados no *thread*.

Thread Control Block (TCB) - 3

- **Informações de Sincronização e Comunicação**: Informações usadas para sincronizar a execução deste thread com outros threads. Isso pode incluir mutexes, semáforos, e variáveis condicionais que o thread está usando.
- **Ponteiro de Pilha**: Um ponteiro para a pilha de memória do thread. A pilha é usada para armazenar variáveis locais e retornar endereços para chamadas de função.

Tarefas CPU-Bond e I/O-Bond

- **Tarefas CPU-bound:** aquelas em que a velocidade de execução é determinada pela velocidade da CPU. Exemplos incluem tarefas de processamento de imagem, de aprendizado de máquina, física de simulação, cálculos matemáticos complexos e mais.
- **Tarefas I/O-bound:** aquelas em que a velocidade de execução é determinada principalmente pela velocidade de I/O, como leitura/escrita de disco, download/upload de rede, interação do usuário, etc.

voltar

Args

- **args**: é uma tupla de argumentos posicionais. Quando você usa **args**, você está passando argumentos para a função na ordem em que eles são definidos.

```
def my_function(a, b, c):
    print(a, b, c)


args = (1, 2, 3)
my_function(*args)  # Imprime: 1 2 3
```

Kwargs

- **kwargs**: é um dicionário de argumentos de palavras-chave. Quando você usa kwargs, está passando argumentos para a função como pares de chave-valor, o que permite especificar o nome do argumento ao passá-lo. Por exemplo:

```
def my_function(a, b, c):
    print(a, b, c)

kwargs = {"a": 1, "b": 2, "c": 3}
my_function(**kwargs)  # Imprime: 1 2 3
```

Args ou Kwargs

- Com **args**, os argumentos são passados na ordem correta, a ordem em que foram definidos na função. Enquanto com **kwargs**, os argumentos são passados pelo nome.
- Os **kwargs** são úteis quando você tem muitos argumentos e quer tornar o código mais legível, ou quando você quer passar argumentos em uma ordem diferente da que eles são definidos na função.

Agendador de Tarefas Preemptivo

- O agendador pode interromper a execução de uma tarefa a qualquer momento, sem a necessidade de cooperação explícita da tarefa. Essa interrupção é baseada em prioridades atribuídas às tarefas ou em determinadas condições de eventos, como o término de uma tarefa de alta prioridade.
- Quando uma tarefa é interrompida, o contexto da tarefa atual, incluindo o estado do processador e os registros, é salvo para que possa ser retomado posteriormente.

Agendador de Tarefas Preemptivo

- A abordagem preemptiva do agendamento de tarefas permite que tarefas de alta prioridade sejam executadas rapidamente, mesmo que tarefas de baixa prioridade estejam em execução.
- Ajuda a melhorar a responsividade do sistema, em situações em que tarefas críticas precisem ser executadas em tempo real ou quando há uma necessidade de resposta rápida a eventos externos.

voltar