Programação Distribuída e Concorrente

Universidade de Vila Velha

Comprometida com a excelência no ensino, pesquisa e inovação.

✉ wanderson.santana@uvv.br

Av. Comissário José Dantas de Melo, 21 – Boa Vista – Vila Velha/ES

www.uvv.br | (27) 3421-2001

2 Abordagem prática para Programação Concorrente

Resumo

Esta unidade apresenta uma introdução teórica e prática aos limites do paralelismo computacional por meio da Lei de Amdahl, destacando o equilíbrio entre otimização da parte sequencial e aumento do número de processadores. Discutimos os conceitos fundamentais de speedup, fração sequencial e paralelizável, bem como a derivação matemática da fórmula que estabelece o ganho máximo de desempenho em sistemas paralelos. Simulações em Python utilizando processamento concorrente para a verificação de números primos, evidenciando empiricamente o fenômeno de saturação do paralelismo previsto teoricamente. Os resultados experimentais mostram que o aumento do número de núcleos produz ganhos iniciais significativos, mas tende a apresentar retornos decrescentes devido à sobrecarga de coordenação e à presença de trechos sequenciais. Também discutiremos as implicações práticas da Lei de Amdahl no desenvolvimento de software, ressaltando que melhorias algorítmicas e a redução da parte sequencial frequentemente são mais eficazes do que a simples adição de hardware. Complementarmente, são introduzidos os conceitos de concorrência, multithreading e threads em Python, com exemplos didáticos que ilustram execução simultânea, gerenciamento de processos e compartilhamento de recursos.

"Mesmo quando multiplicamos nossas possibilidades, permanecemos limitados por aquilo que não podemos transformar."
Paradoxo da Liberdade Condicionada

Lei de Amdahl

Como encontrar o equilíbrio entre paralelizar um programa sequencial (aumentando o número de processadores) e otimizar a eficiência da própria parte sequencial? Por exemplo: o que é melhor — ter quatro processadores executando 40% de um programa em paralelo, ou dois processadores operando sobre 80% do código paralelizável?

Esse tipo de *trade-off*, comum na prática de programação, pode ser estrategicamente analisado por meio da Lei de Amdahl [1].

Embora o paralelismo proporcione ganhos expressivos em desempenho, ele não representa uma solução mágica capaz de acelerar indefinidamente qualquer programa. É fundamental que desenvolvedores compreendam os limites teóricos do paralelismo. A Lei de Amdahl nos ajuda exatamente nesse ponto [2, 3].

Conceitos fundamentais

A Lei de Amdahl fornece uma fórmula matemática que estima o ganho de desempenho ao se adicionar processadores a uma tarefa parcialmente paralelizável. Antes de abordar a equação, é importante definir alguns termos:

- A lei trata da **aceleração (speedup)** na latência da execução de tarefas paralelizáveis. Embora a concorrência não seja tratada diretamente, os resultados podem ser generalizados para programas concorrentes [2].
- A **velocidade** de um programa é o tempo total necessário para sua execução completa.
- **Speedup** (aceleração) mede o ganho ao se usar múltiplos processadores. É definido como:

$$S = \frac{T(1)}{T(N)}$$

onde $T(1)$ é o tempo de execução com um único processador e $T(N)$ é o tempo com N processadores.

Derivação da fórmula da Lei de Amdahl

Assuma que uma fração B do programa é estritamente sequencial, e que a parte paralelizável pode ser distribuída entre N processadores. O tempo total de execução com paralelismo é:

$$T(N) = B \cdot T(1) + \frac{(1 - B) \cdot T(1)}{N}$$

Logo, o ganho de speedup é dado por:

$$S = \frac{T(1)}{T(N)} = \frac{1}{B + \frac{1-B}{N}}$$

Exemplo 1: Ganho com 4 processadores

Problema

2.1 Se 40% de um programa pode ser paralelizado ($B = 0.6$) e utilizamos 4 processadores ($N = 4$), o ganho será:

$$S = \frac{1}{0.6 + \frac{0.4}{4}} = \frac{1}{0.6 + 0.1} = \frac{1}{0.7} \approx 1.43$$

Exemplo 2: Aumentando a paralelização

Agora, suponha que o programa seja reestruturado para permitir que 80% seja executado em paralelo ($B = 0.2$), mesmo com apenas 2 processadores ($N = 2$):

$$S = \frac{1}{0.2 + \frac{0.8}{2}} = \frac{1}{0.2 + 0.4} = \frac{1}{0.6} \approx 1.67$$

Este resultado mostra que otimizar a parte paralelizável pode ser mais eficaz do que simplesmente aumentar o número de processadores [2].

Exemplo 3: Limite teórico com muitos processadores

Se $B = 0.1$ (isto é, 90% do programa é paralelizável), mesmo com um número infinito de processadores, o ganho máximo é:

$$\lim_{N \to \infty} S = \frac{1}{B} = \frac{1}{0.1} = 10$$

Ou seja, o ganho está limitado à fração sequencial do programa [1].

Exemplo 4: Casos reais (aplicação em renderização)

Considere um sistema de renderização gráfica 3D. Suponha que 95% da tarefa de renderização de um quadro pode ser paralelizada. Se usamos 8 processadores:

$$B = 0.05, \quad N = 8$$
$$S = \frac{1}{0.05 + \frac{0.95}{8}} \approx \frac{1}{0.05 + 0.11875} = \frac{1}{0.16875} \approx 5.93$$

Neste caso, mesmo com 95% do trabalho paralelizável, o ganho máximo com 8 processadores é cerca de 6 vezes. Para obter ganhos maiores, é necessário reduzir ainda mais a parte sequencial.

Crescimento do número de processadores

A fórmula também mostra que, à medida que N aumenta, o ganho incremental diminui:

$$\frac{1}{B + \frac{1-B}{N}} < \frac{1}{B + \frac{1-B}{N+1}} \Rightarrow S_N < S_{N+1}$$

Contudo, a melhoria em performance tende a um limite $\frac{1}{B}$:

$$\begin{cases} S \leq \frac{1}{B} \\ \lim_{N \to \infty} S = \frac{1}{B} \end{cases}$$

Implicações práticas

A Lei de Amdahl fornece um limite superior teórico para a aceleração que podemos obter com paralelismo. Mesmo que tenhamos hardware com muitos núcleos, o ganho de desempenho será limitado pela fração sequencial do programa [3, 4].

Na prática, isso significa que é mais vantajoso trabalhar na **melhoria do algoritmo** ou na **redução da parte sequencial**, antes de aumentar indefinidamente o número de processadores.

Limitações da Lei de Amdahl

A Lei de Amdahl assume que:

- O tamanho do problema permanece fixo.
- A sobrecarga de comunicação entre processadores é desprezível.
- Não há gargalos de memória ou I/O.

Modelos como a Lei de Gustafson complementam essas análises ao considerar aumento no tamanho do problema junto com o número de processadores [3].

Simulação em Python

Nesta seção, veremos os resultados da Lei de Amdahl por meio de um programa Python. Ainda considerando a tarefa de determinar se um número inteiro é um número primo,

conforme discutido na Unidade I, veremos como a aceleração real é alcançada por meio da concorrência.

Para relembrar, o código apresenta a função que verifica números primos. Na sequência, implementamos uma função que recebe um número inteiro de processadores que utilizaremos para resolver com concorrência o problema.

Código: `exemplo_01.py`

```python
from math import sqrt
import concurrent.futures
import multiprocessing
from timeit import default_timer as timer


def eh_primo(x):
    """
    Retorna o proprio numero se for primo, None caso contrario.
    """
    if x < 2:
        return None
    if x == 2:
        return x
    if x % 2 == 0:
        return None

    limit = int(sqrt(x)) + 1
    for i in range(3, limit, 2):
        if x % i == 0:
            return None
    return x


def encontrar_primos_no_intervalo(inicio, quantidade, n_workers):
    """
    Encontra numeros primos em um intervalo usando ProcessPoolExecutor
    Retorna: (lista de primos, tempo de coleta de resultados, tempo total)
    """
    intervalo = range(inicio, inicio + quantidade)
```

```
31
32      start_total = timer()
33      primos = []
34
35      with concurrent.futures.ProcessPoolExecutor(max_workers=n_workers) as executor:
36          # Submete todas as tarefas
37          futures = [executor.submit(eh_primo, num) for num in intervalo]
38
39          start_coleta = timer()
40
41          # Processa resultados conforme vao ficando prontos
42          for future in concurrent.futures.as_completed(futures):
43              resultado = future.result()
44              if resultado is not None:
45                  primos.append(resultado)
46
47          tempo_coleta = timer() - start_coleta
48
49      tempo_total = timer() - start_total
50
51      return primos, tempo_coleta, tempo_total
52
53
54  def main():
55      # Configuracoes do teste
56      _inicio = 10**13
57      _quantidade = 1000
58      _max_workers = multiprocessing.cpu_count()
59
60      print(f"Testando intervalo: {_inicio:,} ate {_inicio + _quantidade:,}")
61      print(f"Numero maximo de workers disponiveis: {_max_workers}\n")
62
63      for n_workers in range(1, _max_workers + 1):
64          print(f"\nTestando com {n_workers} processo(s)")
65
66          primos, t_coleta, t_total = encontrar_primos_no_intervalo(
67              _inicio, _quantidade, n_workers
68          )
69
```

```
        print(f"Primos encontrados: {len(primos)}")
        if primos:
            print(f"  Menor primo: {min(primos):,}")
            print(f"  Maior primo: {max(primos):,}")

        print(f"  Tempo apenas coleta de resultados : {t_coleta:8.4f} s")
        print(f"  Tempo total (submissao + execucao + coleta): {t_total:8.4f} s")
        print("-" * 30)


if __name__ == "__main__":
    main()
```

Eu disponho de oito núcleos no meu computador. Assim, o resultado obtido é o seguinte:

```
Numero de Processadores: 1.
Duracao Intermediaria: 2.0738 seconds.
Duracao Total: 2.0885 seconds.

Numero de Processadores: 2.
Duracao Intermediaria: 1.0942 seconds.
Duracao Total: 1.1028 seconds.

Numero de Processadores: 3.
Duracao Intermediaria: 0.7915 seconds.
Duracao Total: 0.8013 seconds.

Numero de Processadores: 4.
Duracao Intermediaria: 0.6724 seconds.
Duracao Total: 0.6826 seconds.

Numero de Processadores: 5.
Duracao Intermediaria: 0.6921 seconds.
Duracao Total: 0.7024 seconds.
```

```
Numero de Processadores: 6.
Duracao Intermediaria: 0.6774 seconds.
Duracao Total: 0.6871 seconds.

Numero de Processadores: 7.
Duracao Intermediaria: 2.0061 seconds.
Duracao Total: 2.0200 seconds.

Numero de Processadores: 8.
Duracao Intermediaria: 0.7921 seconds.
Duracao Total: 0.8041 seconds.
```

Algumas observações:

- Primeiro, em cada iteração, a subseção da tarefa era quase tão longa quanto todo programa. Em outras palavras, a computação concorrente correspondeu à maior parte do programa durante cada iteração.
- Em segundo lugar, e possivelmente mais interessante, podemos ver que, embora consideráveis melhorias foram obtidas após aumentar o número de processadores de 1 para 2 (2.0885 segundos para 1.1028 segundos), nota-se que interações com outras quantidades de núcleos tiveram resultados da mesma ordem que obtivemos ao utilizarmos somente 1 núcleo: é o caso quando 7 processadores foram ativados.
- Vamos recorrer a uma curva de aceleração para entendermos melhor esse fenômeno. Uma curva aceleração é simplesmente um gráfico com o eixo x mostrando o número de processadores, em comparação com o eixo y, mostrando a aceleração alcançada. Num cenário perfeito, onde $S = N$ (ou seja, a aceleração alcançada é igual ao número de processadores usado), a curva de aceleração seria uma linha reta de 45 graus. No entanto, a Lei de Amdahl mostra que a curva de aceleração produzida por qualquer programa permanecerá abaixo dessa linha e começará a se estabilizar à medida que a eficiência for reduzida. No exemplo aqui apresentado, isso parece ocorrer a partir do oitavo núcleo ser ativado!!!

Explicação Didática do Código em Python

O código apresentado tem como objetivo ilustrar, de forma prática, os efeitos do paralelismo no desempenho computacional, tema central da Lei de Amdahl. Ele utiliza a linguagem Python e a biblioteca `concurrent.futures`, que permite a execução concorrente de tarefas utilizando múltiplos processos. A tarefa computacional escolhida — verificar a primalidade de números muito grandes — é especialmente custosa, o que a torna ideal para evidenciar os ganhos (e limites) do paralelismo.

Um dos elementos centrais no código é a construção `with ProcessPoolExecutor(...) as executor`. Esta estrutura garante que os processos paralelos serão corretamente criados, gerenciados e finalizados. Ao utilizar a palavra-chave `with`, o Python entra automaticamente em um contexto no qual os recursos alocados (neste caso, os processos do pool) são liberados de forma segura e eficiente ao final da execução. Assim, evita-se a necessidade de encerrar manualmente os processos e garante-se que o código seja mais robusto.

Para distribuir as tarefas entre os múltiplos processos, é utilizada a função `executor.submit(funcao, argumento)`, que envia uma chamada de função para ser executada de maneira assíncrona por um dos trabalhadores do pool. Essa função retorna um objeto chamado *futuro* (do inglês *future*), que representa uma promessa de que o resultado estará disponível em algum momento. Esses objetos são posteriormente verificados por meio do construtor `concurrent.futures.as_completed()`, que retorna os resultados conforme eles forem sendo concluídos.

Outro aspecto importante do código é a medição dos tempos de execução. Utiliza-se a função `timer()` da biblioteca `timeit` para medir tanto o tempo total de execução quanto o tempo intermediário, que corresponde apenas à etapa de coleta e verificação dos resultados. Com esses dados, é possível observar com precisão o quanto a execução foi acelerada à medida que se aumenta o número de processadores.

Este experimento prático está diretamente relacionado à Lei de Amdahl, que estabelece um limite teórico para o ganho de desempenho em programas paralelizáveis. A lei afirma que, mesmo com um número muito grande de processadores, a aceleração total de um programa será limitada pela fração do código que não pode ser paralelizada. Isso significa

que, a partir de certo ponto, adicionar mais núcleos não resultará em reduções significativas no tempo de execução — e, em alguns casos, pode até haver aumento devido ao custo de coordenação entre processos.

Com isso, o código permite ao estudante observar concretamente como o tempo de execução diminui inicialmente com o aumento do número de núcleos, mas eventualmente se estabiliza, demonstrando o fenômeno de *saturação do paralelismo*, conforme previsto teoricamente. Além disso, a visualização gráfica dos resultados, por meio de um gráfico relacionando o número de processadores ao tempo total de execução, reforça essa compreensão de maneira intuitiva e visual.

Em suma, este experimento computacional alia teoria e prática, permitindo que o estudante compreenda não apenas o funcionamento de técnicas de paralelismo em Python, mas também os princípios fundamentais que regem os limites de aceleração em sistemas paralelos.

Código: `graphic_01.py`

```python
import matplotlib.pyplot as plt

processadores = [1, 2, 3, 4, 5, 6, 7, 8]  # ajuste conforme seu CPU
tempos = [12.2, 6.8, 5.0, 4.3, 4.1, 4.0, 3.9, 3.9]  # exemplo hipotetico

plt.plot(processadores, tempos, marker='o')
plt.xlabel('Numero de Processadores')
plt.ylabel('Tempo Total (s)')
plt.title('Aceleracao vs Paralelismo (Lei de Amdahl)')
plt.grid(True)
plt.show()
```

Threads

Embora esse estilo de programação possa levar a desvantagens de uso e problemas que precisam ser resolvidos, aplicativos modernos com o mecanismo de *multithreading* estão sendo amplamente usados. Praticamente todos os sistemas operacionais existentes

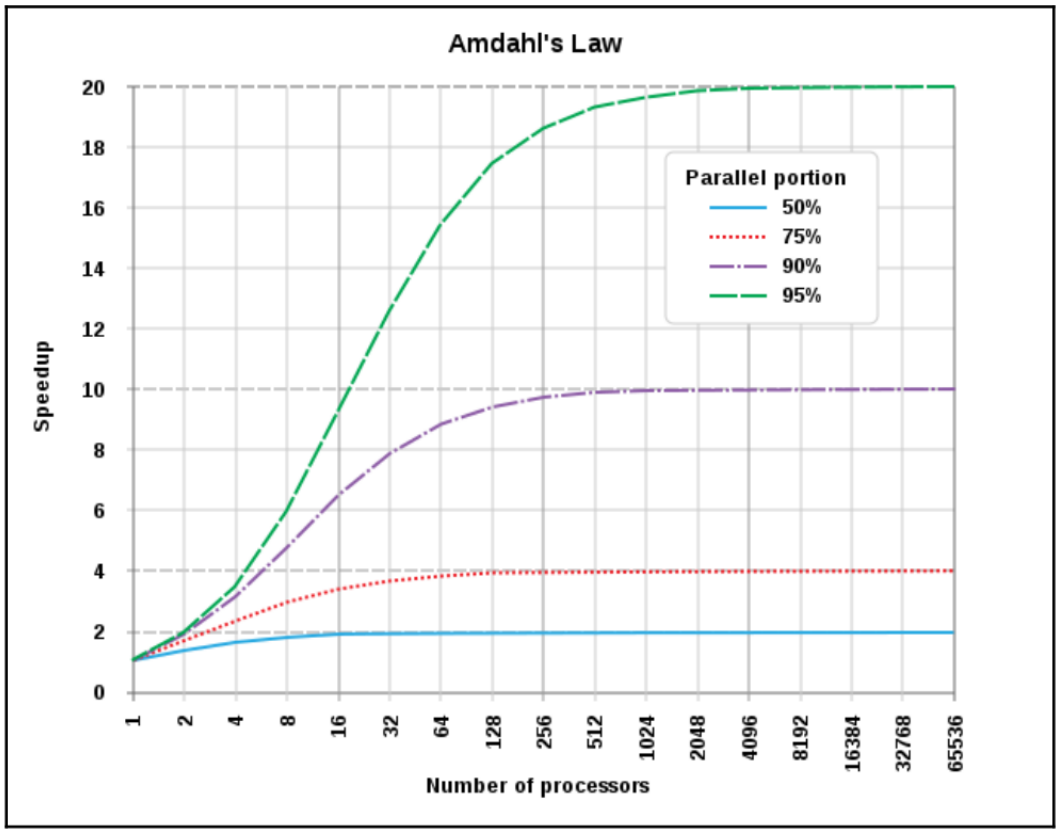

Figura 2.1: Curvas de aceleração com diferentes porções paralelas.

suportam `multithreading` e, em quase todas as linguagens de programação, há mecanismos que você pode usar para implementar aplicativos simultâneos por meio do uso de `threads`. Portanto, a programação `multithread` é definitivamente uma boa escolha para obter aplicativos simultâneos.

No entanto, não é a única opção disponível — há várias outras alternativas, algumas das quais, entre outras, têm melhor desempenho na definição de `thread`.

No campo da ciência da computação, um `thread` de execução é a menor unidade de comandos de programação (código) que um planejador (**scheduler** - geralmente como parte de um sistema operacional) pode processar e gerenciar. Dependendo do sistema operacional, a implementação de `threads` e processos varia, mas um `thread` é tipicamente um elemento (um componente) de um processo.

Mais de um `thread` pode ser implementado dentro do mesmo processo, geralmente executando simultaneamente e acessando/compartilhando os mesmos recursos, como memória; processos separados não fazem isso. `Threads` no mesmo processo compartilham

suas últimas instruções (seus trechos de códigos) e o contexto (valores que suas variáveis referenciam em qualquer momento).

A principal diferença entre os dois conceitos é que um `thread` é tipicamente um componente de um processo. Portanto, um processo pode incluir vários `threads`, que podem estar executando simultaneaente. `Threads` também geralmente permitem recursos compartilhados, como memória e dados, enquanto é bastante raro que processos façam isso. Em resumo, um `thread` é um componente independente de computação que é semelhante a um processo, mas os `threads` dentro de um processo podem compartilhar o espaço de endereço e, portanto, os dados desse processo.

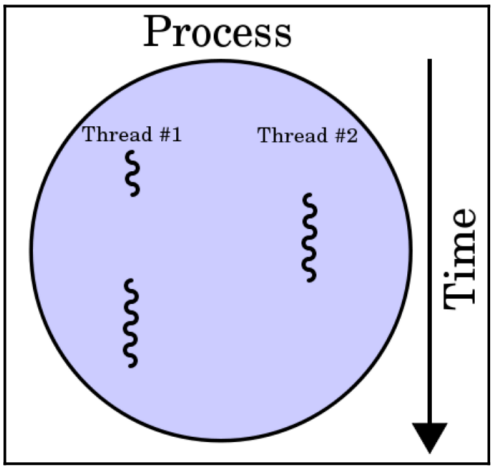

Figura 2.2: Idealização de um processo com duas `Theads` de execução rodando num processador.

> **NOTA**
>
> Os `threads` foram supostamente usados pela primeira vez para um número variável de tarfas na multiprogramação do OS/360, que é um sistema de processamento em lote descontinuado que foi desenvolvido pela IBM em 1967. Na época, os `threads` eram chamados de **tarefas** pelos desenvolvedores, enquanto o termo `thread` se tornou popular mais tarde e fois atribuído a Victor A. Vyssotsky, um matemático e cientísta da computação que foi diretor fundador do Cambridge Research Lab da Digital.

Na ciência da computação, o *single-threading* é semelhante ao processamento sequencial

tradicional, executando um único comando a qualquer momento. Por outro lado, o *multithreading* implementa mais de um `thread` para existir e executar em um único processo, simultaneamente. Ao permitir que vários `threads` acesses recursos/contextos compartilhados e sejam executados independentemente, essa técnica de programação pode ajudar os aplicativos a ganhar velocidade na execução de tarefas independentes.

O *multithreading* pode ser alcançado principalmente de duas maneiras. Em sistemas de processador único, o *multithreading* é tipicamente implementado via divisão de tempo, uma técnica que permite à CPU alternar entre diferentes softwares em execução em diferentes `threads`. No fatiamento de tempo, a CPU alterna sua execução tão rapidamente e com tanta frequência que os usuários geralmente percebem que o software está sendo executado em paralelo (por exemplo, quando você abre dois softwares ao mesmo tempo e um computador de processador único).

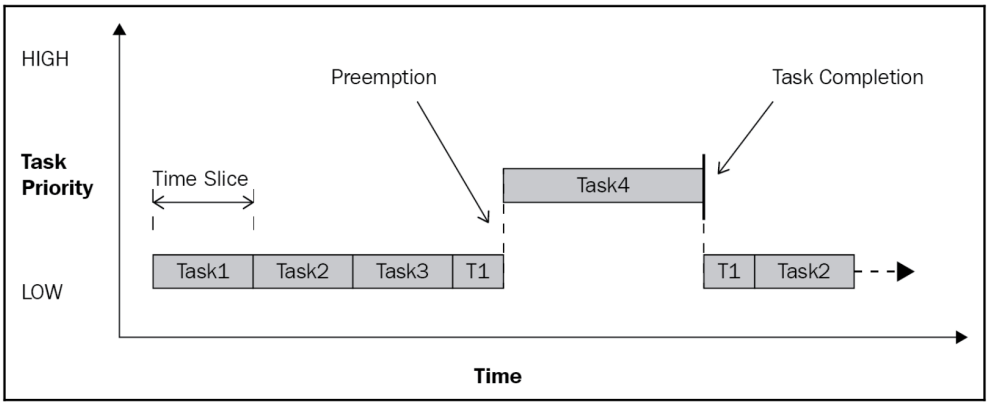

Figura 2.3: Exemplo de divisão de tempo (**time slicing**), conhecida como **round-robin scheduling**.

Ao contrário dos sistemas de processador único, os sistemas com múltiplos processadores ou núcleos podem implementar facilmente *multithreading*, executando cada `thread` em um processador ou núcleo separado, simultaneamente. Além disso, a divisão de tempo é uma opção, pois esses sistemas multiprocesso ou **multicore** podem ter apenas um processdor/núcleo para alternar entre tarefas - embora isso geralmente não seja uma boa prática.

Os aplicativos *multithreaded* têm uma série de vantagens, em comparação com os aplicativos sequenciais tradicionais. Listamos algumas dessas vantagens a seguir:

- **Tempo de execução mais rápido:** umas das principais vantagens da concorrência por meio do *multithreading* é a aceleração obtida. `Threads` separados no mesmo programa podem ser executados simultaneamente ou em paralelo, se forem suficientemente independentes um do outro.

- **Responsividade:** Um programa *single-threaded* só pode processar uma parte da entrada por vez; portanto, se o `thread` de execução principal bloquear em uma tarefa de longa execução (ou seja, uma parte de entrada que requer computação e processamento pesado), o programa inteiro não será capaz de continuar com outra entrada e, portanto, ele parecerá congelado.

  Ao usar `threads` separados para executar a computação e permanecer em execução para receber diferentes entradas do usuário simultaneamente, um programa *multithread* pode fornecer melhor responsividade.

- **Eficiência no consumo de recursos:** como mencionamos anteriormente, vários `threads` dentro do mesmo processo podem compartilhar e acessar os mesmos recursos. Consequentemente, programas *multithread* podem atender e processar muitas solicitações de clientes para dados simultaneamente, usando significativamente menos recursos do que seria necessário ao usar programas *single-threaded* ou multiprocesso. Isso também leva a uma comunicação mais rápida entre `threads`.

Dito isso, programas *multithread* também têm suas desvantagens, como segue:

- **Travamentos:** mesmo que um processo possa conter vários `threads`, uma única operação ilegal dentro de um `thread` pode afetar negativamente o processamento de todos os outros `threads` no processo e pode travar o programa inteiro como resultado.

- **Sincronização:** embora compartilhar os mesmos recursos possa ser uma vantagem sobre a programação sequencial tradicional ou programas de multiprocessamento, uma consideração curiosa também é necessária para os recursos compartilhados. Normalmente, os `threads` devem ser coordenados de forma deliberada e sistemática, para que os dados compartilhados sejam computados e manipulados corretamente. Problemas não intuitivos que podem ser causados por coordenação des-

cuidada de `threads` incluiem `deadlocks`[1], `livelocks`[2] e condições de corrida, que discutiremos mais à frente.

Um exemplo em Python

Para ilustrar o conceito de executar vários `threads` no mesmo processador, vamos dar uma olhada no exemplo a seguir.

Código: `my_tread.py`

```
1  import threading
2  import time
3
4  class MyThread(threading.Thread):
5      def __init__(self, name, delay):
6          threading.Thread.__init__(self)
7          self.name = name
8          self.delay = delay
9
10     def thread_count_down(self, name, delay):
11         counter = 5
12         while counter:
13             time.sleep(delay)
14             print('Thread %s counting down: %i...' % (name, counter))
15             counter -= 1
16
17     def run(self):
18         print('Iniciando thread %s.' % self.name)
19         self.thread_count_down(self.name, self.delay)
20         print('Finalizando thread %s.' % self.name)
```

No exemplo acima, utilizamos o módulo `threading` do `Python` como base da classe `MyThread`. Cada objeto desta classe tem um nome e um parâmetro de atraso. A função `run()`, que

[1] Um estado de deadlock ocorre quando dois ou mais processos estão esperando indefinidamente por um evento que só pode ocorrer por um dos processos em espera.

[2] Livelock é um caso especial de inanição de recursos; a definição geral afirma apenas que um processo específico não está progredindo.

é chamada assim que uma nova `thread` é inicializada, imprime uma mensagem de início e, por sua vez, chama a função `thread_count_down()`. Esta função conta regressivamente do número 5 ao número 0, enquando *"dorme"* entre as iterações por um número de segundos, especificado pelo parâmetro de atraso.

O objetivo deste exemplo é mostrar a natureza simultânea de executar mais de um `thread` no mesmo programa (ou processo) iniciando mais de um objeto da classe `MyThread` ao mesmo tempo.

Agora, vejamos como rodar o nosso objeto.

Código: `exemplo_02.py`

```
1  from my_thread import MyThread
2
3  thread1 = MyThread('A', 0.5)
4  thread2 = MyThread('B', 0.5)
5  thread1.start()
6  thread2.start()
7
8  thread1.join()
9  thread2.join()
10
11 print('Finalizado.')
```

Aqui, estamos inicializando dois `threads` juntos, cada um dos quais tem 0.5 segundos como parãmetro de atraso. Executando esse *script*, obtemos a seguinte saída:

```
Iniciando thread A.
Iniciando thread B.
Thread B counting down: 5...
Thread A counting down: 5...
Thread B counting down: 4...
Thread A counting down: 4...
Thread B counting down: 3...
Thread A counting down: 3...
Thread B counting down: 2...
```

```
Thread A counting down: 2...
Thread B counting down: 1...
Finalizando thread B.
Thread A counting down: 1...
Finalizando thread A.
Finalizado.
```

Como esperávamos, a saída nos diz que as duas contagens regressivas para os `threads` foram executadas simultaneamente. Em vez de terminar a contagem regressiva do primeiro `thread` e então iniciar a contagem regressiva do segundo `thread`, o programa executou as duas contagens regressivas ao mesmo tempo. Sem incluir alguma sobrecarga e declarações diversas, essa técnica de `threading` permite uma melhoria quase dupla na velocidade do programa.

No entanto, há uma observação importante: após a primeira contagem regressiva para o número 5, podemos ver que a contagem regressiva do `thread` B realmente ficou à frente do `thread` A na execução, embora saibamos que o `thread` A tenha sido inicializado primeiro. Essa mudança realmente permitiu que o `thread` B terminasse antes do `thread` A.

Esse fenômeno é um resultado direto da simultaneidade, via *multithreading*; como os dois `threads` foram inicalizados quase simultaneamente, era bem provável que um `thread` ficasse à frente do outro na execução.

Visão geral sobre o módulo `threading`

Há muitas escolhas quando se trata de implementar programas **multithread** em `Python`. Uma das maneiras mais comuns de trabalhar com `threads` em Python é por meio do módulo `threading`.

O módulo `threading` foi projetado para ser amigável para aqueles que vêm do paradigma de desenvolvimento de software orientado a objetos, tratando cada `thread` que é criado como um objeto. Ele suporta uma série de métodos, como:

- `threading.activeCount()`: função que retorna o número de objetos `thread` ativos

no programa

- `threading.current_thread()`: retorna o número de objetos `thread` no controle `thread` atual do chamador
- `threading.enumerate()`: retorna uma lista de todos os objetos `thread` ativos no programa

Seguindo o paradigma de desenvolvimento de software orientado a objetos, o módulo `threading` também fornece uma classe `Thread` que suporta a implementação orientada a objetos de `threads`. Os seguintes métodos são suportados nesta classe:

- `run()`: este método é executado quando um novo `thread` é inicializado e disparado para o processador;
- `start()`: este método inicia o objeto `thread` de chamada inicializado, invocando o método `run()`;
- `join()`: este método aguarda o término do objeto `thread` antes de continuar a executar o restante do programa;
- `isAlive()`: este método retorna um valor booleano, indicando se o objeto `thread` está em execução no momento;
- `name()`: este método retorna o nome do objeto `thread`;
- `setName()`: este método define o nome do objeto `thread`.

A partir das definições listadas acima, podemos perceber que cada `thread` parece ser composta principalmente de três elementos: **contador de programa**, **registradores** e **pilha**.

Recursos compartilhados com outras `threads` do mesmo processo incluem essencialmente dados e recursos do sistema operacional. Semelhante ao que acontece com os processos, até mesmo as `threads` têm seu próprio estado de execução e podem sincronizar entre si. Os estados de execução de uma `thread` são geralmente chamados de **pronto**, **em execução** e **bloqueado**.

Uma aplicação típica de uma `thread` é certamente a paralelização de um software de aplicativo, especialmente, para aproveitar os processadores multi-core modernos, onde cada núcleo pode executar uma única `thread`. Como já verificamos aqui, a vantagem das `threads` sobre o uso de processos está no desempenho, pois a troca de contexto entre processos acaba sendo muito mais pesada do que o contexto de troca entre `threads` que pertencem ao mesmo processo.

A programação *multithread* prefere um método de comunicação entre `threads` usando o espaço de informações compartilhadas. Essa escolha requer que o principal problema a ser abordado pela programação com threads esteja relacionado ao gerenciamento desse espaço.

Praticando com o módulo `threading`

Como definir um `thread`

A maneira mais simples de usar um `thread` é instanciá-lo com uma função de destino e, em seguida, chamar o método `start()` para deixá-lo começar seu trabalho. O módulo `Python threading` tem o método `Thread()` que é usado para executar processos e funções em um `thread` diferente:

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={})
```

Vamos interpretar cada um dos argumentos do método acima:

- **`group`**: é o valor de `group` que deve ser `None`; esse argumento será explorado no futuro;
- **`target`**: função que deve ser executada quando você inicia uma atividade de `thread`;
- **`name`**: nome do `thread`; por padrão, um nome exclusivo do formato `Thread-N` é

atribuído a ele;

- `args`: tupla de argumentos que devem ser passados para um alvo;
- `kwargs`: dicionário de argumentos de palavras-chaves que devem ser usados para a função alvo.

É útil gerar um `thread` e passar argumentos para ele que dizem qual o trabalho a fazer. O exemplo a seguir passa um número, que é o número do `thread`, e então imprime o resultado.

Vamos ver como definir um `thread` com o módulo `threading`, para isso, algumas linhas de código são necessárias:

Código: `exemplo_03.py`

```
1  import threading
2
3  def function(i):
4      print ("Funcao chamada pelo thread %i\n" % i)
5      return
6
7  threads = []
8  for i in range(5):
9      t = threading.Thread(target=function , args=(i,))
10     threads.append(t)
11     t.start()
12     t.join()
```

A saída do código precedente é a seguinte:

```
Funcao chamada pelo thread 0

Funcao chamada pelo thread 1

Funcao chamada pelo thread 2

Funcao chamada pelo thread 3
```

```
Funcao chamada pelo thread 4
```

Devemos salientar que a saída pode ser obtida de uma maneira diferente; na verdade, várias `threads` podem imprimir o resultado de volta para `stdout` ao mesmo tempo, então a ordem de saída não pode ser predeterminada.

Interpretando o resultado

Para importar o módulo `threading`, simplesmente usamos o comando `Python`:

```
1 import threading
```

No programa, instanciamos um `thread`, usando o objeto `Thread` com uma função de destino chamada `function`. Além disso, passamos um argumento para a função que será incluído na mensagem de saída:

```
1     t = threading.Thread(target=function , args=(i,))
```

O `thread` não começa a ser executado até que o método `start()` seja chamado, e que `join()` faça o `thread` de chamada esperar até que o `thread` tenha concluído a execução:

```
1     t.start()
2     t.join()
```

Como definir um `currentThread`

Usar argumentos para identificar ou nomear o `thread` é trabalhoso e desnecessário. Cada instância de `Thread` tem um nome com um valor padrão que pode ser alterado conforme o `thread` é criado. Nomear `threads` é útil em processos de servidor com vários `threads` que lidam com operações diferentes.

Para determinar qual `thread` está em execução, criamos três funções de destino e importamos o módulo `time` para introduzir uma execução suspensa de cinco segundos.

Código: `exemplo_04.py`

```
import threading
import time

def primeira_funcao():
    print(threading.current_thread().name + str(' Esta Iniciando \n'))

    time.sleep(5)
    print (threading.current_thread().name + str(' Esta Finalizando \n'))
    return

def segunda_funcao():
    print(threading.current_thread().name + str(' Esta Iniciando \n'))
    time.sleep(5)
    print(threading.current_thread().name + str(' Esta Finalizando \n'))
    return

def terceira_funcao():
    print(threading.current_thread().name + str(' Esta iniciando \n'))
    time.sleep(5)
    print (threading.current_thread().name + str(' Esta Finalizando \n'))
    return

if __name__ == "__main__":
    t1 = threading.Thread(name='primeira_funcao', target=primeira_funcao)
    t2 = threading.Thread(name='segunda_funcao', target=segunda_funcao)
    t3 = threading.Thread(name='terceira_funcao',target=terceira_funcao)
    t1.start()
    t2.start()
    t3.start()
```

A saída disso deve ser a seguinte:

```
primeira_funcao Esta Iniciando

segunda_funcao Esta Iniciando
```

```
terceira_funcao Esta iniciando

primeira_funcao Esta Finalizando

terceira_funcao Esta Finalizando

segunda_funcao Esta Finalizando
```

Como utilizar um `thread` em numa subclasse

Para implementar um novo `thread` usando o módulo `threading`, precisamos fazer o seguinte:

- Definir uma nova subclasse da classe `Thread`.
- Substituir o método `__init__(self [,args])` para adicionar argumentos adicionais.
- Substituir o método `run(self [,args])` para implementar o que o `thread` deve fazer quando for iniciado.

Depois de criar a nova subclasse `Thread`, você pode criar uma instância dela e então iniciar um novo `thread` invocando o método `start()`, que, por sua vez, chamará o método `run()`. Para implementar um `thread` em uma subclasse, definimos a classe `myThread`. Observe que ela terá dois métodos que devem ser substituídos pelos argumentos do `thread`.

Código: `exemplo_05.py`

```
1 import threading
2 import time
3
4 exitFlag = 0
5
6 class myThread (threading.Thread):
7     def __init__(self, threadID, name, counter):
```

```
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter

    def print_time(self, threadName, delay, counter):
        while counter:
            if exitFlag:
                threadName.exit()
            time.sleep(delay)
            print ("%s: %s" % (threadName, time.ctime(time.time())))
            counter -= 1

    def run(self):
        print("Iniciando " + self.name)
        self.print_time(self.name, self.counter, 5)
        print ("Finalizando " + self.name)


# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()
print("Saindo do Thread Principal!")
```

Cuja saída é:

```
Iniciando Thread-1
Iniciando Thread-2
Saindo do Thread Principal!
Thread-1: Fri Feb 21 17:14:31 2025
Thread-2: Fri Feb 21 17:14:32 2025
Thread-1: Fri Feb 21 17:14:32 2025
Thread-1: Fri Feb 21 17:14:33 2025
```

```
Thread-2: Fri Feb 21 17:14:34 2025
Thread-1: Fri Feb 21 17:14:34 2025
Thread-1: Fri Feb 21 17:14:35 2025
Finalizando Thread-1
Thread-2: Fri Feb 21 17:14:36 2025
Thread-2: Fri Feb 21 17:14:38 2025
Thread-2: Fri Feb 21 17:14:40 2025
Finalizando Thread-2
```

Lista de Exercícios

2.1 Se um algoritmo tiver um único trecho que ocupa 20% do tempo total de execução e que pode ser pararelizado, ao adicionarmos mais 4 processadores ao sistema, qual seria o aumento de velocidade?

2.2 Considerando que um programa tenha 40% do seu código podendo ser aprimorado para rodar em paralelo, e que um novo processador consiga oferecer 2.3 mais velocidade que o anterior, qual é a aceleração geral do sistema?

2.3 Considere um algoritmo que possui um trecho de código que pode ser melhorado, a fim de que sua execução rode em paralelo. Caso essa implementação venha a ser executada, a aceleração geral do sistema será 3.3 vezes mais rápida. Que fração original do código pode ser melhorada?

2.4 Qual é o principal benefício da programação multithread?

a. Redução do número de erros no código.

b. Capacidade de implementar aplicativos simultâneos.

c. Menor complexidade na construção de aplicativos.

d. Evitar o uso de recursos de processamento.

2.5 Qual é a relação dos sistemas operacionais modernos com o multithreading?

a. Apenas alguns sistemas operacionais suportam multithreading.

b. A maioria dos sistemas operacionais modernos não suporta multithreading.

c. Todos os sistemas operacionais existentes suportam multithreading.

d. O multithreading é uma funcionalidade obsoleta nos sistemas operacionais modernos.

2.6 Das opções a seguir, qual é a desvantagem sobre o uso de multithreading?

a. A programação multithread tem uso desafiador e problemas que precisam ser resolvidos.

b. Os sistemas operacionais não suportam multithreading.

c. Multithreading nunca apresenta problemas em aplicativos modernos.

d. Não há desvantagens mencionadas no texto.

2.7 Qual é uma vantagem de sistemas com múltiplos processadores ou núcleos em relação ao multithreading?

a. Eles não precisam dividir tempo para alternar entre tarefas.

b. Eles não suportam multithreading.

c. Eles sempre alternam entre tarefas usando apenas um processador ou núcleo.

d. Eles podem executar cada thread em um processador ou núcleo separado, simultaneamente.

2.8 O que pode acontecer com um programa single-threaded quando o thread de execução principal bloqueia em uma tarefa de longa execução?

a. O programa continua processando outras entradas normalmente.

b. O programa inteiro pode parecer congelado e não continuará com outras entradas.

c. O programa executa todas as tarefas sequenciais sem bloqueios.

d. O programa reinicia automaticamente o processo principal.

2.9 Qual é a vantagem do uso de threads separados em um programa multithreaded para computação?

a. Elas permitem que o programa continue processando diferentes entradas simultaneamente, melhorando a responsividade.

b. Elas consomem mais recursos do sistema do que um programa single-threaded.

c. Elas fazem o programa ser mais suscetível a travamentos.

d. Elas garantem que a comunicação entre threads seja mais lenta.

2.10 O que significa eficiência no consumo de recursos em programas multithreaded?

a. Programas multithreaded sempre consomem mais recursos do que programas single-threaded.

b. Programas multithreaded não utilizam recursos compartilhados.

c. Programas multithreaded consomem os mesmos recursos que programas de multiprocessamento.

d. Programas multithreaded podem processar muitas solicitações simultaneamente, usando menos recursos em comparação com programas single-threaded ou multiprocessos.

2.11 Quais são os problemas que podem surgir da coordenação descuidada de threads, conforme mencionado no texto?

a. Aumento na velocidade do processamento.

b. Deadlocks, livelocks e condições de corrida.

c. Melhor comunicação entre os threads.

d. Redução da carga de processamento do sistema.

O Módulo queue

O módulo `queue` em `Python` fornece uma implementação simples de estrutura de dados em fila. Nativo no `Python`, basta importá-lo com a declaração `import queue`. O módulo `queue` do `Python` oferece três tipos de classes: `Queue`, `LifoQueue` e `PriorityQueue`:

- `Queue`: É o primeiro tipo de fila. Ele segue a propriedade `FIFO` onde o primeiro elemento inserido é o primeiro a ser extraído.
- `LifoQueue`: Esta é a fila que segue a propriedade `LIFO` (*Last-In-First-Out*) onde o último elemento inserido é o primeiro a ser extraído.
- `PriorityQueue`: Não segue nenhuma das propriedades acima, `FIFO` ou `LIFO`. Esta fila segue o conceito de **ordem de prioridade**. Se os elementos da fila têm prioridades diferentes, então o elemento com alta prioridade é extraído primeiro.

Cada fila na classe `queue.Queue` pode conter uma quantidade específica de elementos e pode ter os seguintes métodos com sua API de alto nível:

- ✓ `get()`: este método retorna o próximo elemento do objeto da fila de chamada e o remove do objeto da fila;
- ✓ `put()`: este método adiciona um novo elemento ao objeto da fila de chamada;
- ✓ `qsize()`: este método retorna o número de elementos atuais no objeto da fila de chamada (ou seja, seu tamanho);
- ✓ `empty()`: este método retorna um Booleano, indicando se o objeto da fila de chamada está vazio;
- ✓ `full()`: este método retorna um Booleano, indicando se o objeto da fila de chamada está cheio.

Por exemplo, essa seria a sintaxe básica para cada tipo de fila:

```
1 import queue
2
3 fila_fifo = queue.Queue()
```

```
fila_lifo = queue.LifoQueue()
fila_prioridade = queue.PriorityQueue()
```

Com a biblioteca `queue`, pode-se adicionar elementos na fila utilizando o método `put` e remover utilizando o método `get`. Exemplo:

```
import queue

fila = queue.Queue()
fila.put("Primeiro da fila")
fila.put("Segundo da fila")

print(fila.get())
```

E a saída será:

```
Primeiro da fila
```

FIFO, LIFO, Priority Queue

Como já discutimos, quando utilizamos filas, temos três abordagens principais: `FIFO`, `LIFO` e `Priority Queue`. Veja a diferença entre elas:

```
# Exemplo de FIFO
fila_FIFO = queue.Queue()
fila_FIFO.put("Primeiro da fila")
fila_FIFO.put("Segundo da fila")
print(fila_FIFO.get())
# Saida sera: Primeiro da fila

# Exemplo de LIFO
fila_LIFO = queue.LifoQueue()
fila_LIFO.put("Primeiro da fila")
fila_LIFO.put("Segundo da fila")
print(fila_LIFO.get())
# Saida sera: Segundo da fila

```

```
15 # Exemplo de PriorityQueue
16 fila_prioridade = queue.PriorityQueue()
17 fila_prioridade.put((2, "Primeiro da fila"))
18 fila_prioridade.put((1, "Segundo da fila"))
19 print(fila_prioridade.get())
20 # Saida sera: (1, 'Segundo da fila')
```

Enfileiramento em Programação Concorrente

O conceito de uma fila é ainda mais prevalente no subcampo da programação simultânea, especialmente quando precisamos implementar um número fixo de *threads* em nosso programa para interagir com um número de variável de recursos compartilhados.

Nos exemplos anteriores, aprendemos a atribuir uma tarefa específica a uma nova *thread*. Isso significa que o número de tarefas que precisam ser processadas determinará o número de *threads* que nosso programa deve gerar.

Às vezes, é indesejável ter tantas *threads* e fazer com que cada *thread* execute apenas uma tarefa. Poderia ser mais benéfico ter um número fixo de *threads* (comumente conhecido como `pool` de *threads*[3]) que trabalharia nas tarefas de forma cooperativa. É aqui que entra o conceito de fila.

Podemos projetar uma estrutura na qual o `pool` de *threads* não manterá nenhuma informação sobre as tarefas que cada uma deve executar, em vez disso, as tarefas são armazenadas em uma fila (em outras palavras, uma fila de tarefas) e os itens na fila serão alientados para membros individuais do `pool` de *threads*. Se a fila de tarefas ainda contiver elementos a serem processados, então o próximo elemento na fila será enviado para a *thread* que acabou de ficar disponível.

O diagrama a seguir, exibido na Figura (2.4), ilustra esta dinâmica proposta.

Vamos considerar um exemplo rápido em `Python`, para ilustra essa discussão. Consideremos o problema de determinar todos os fatores positivos de um elemento em uma lista dada de inteiros positivos. O código a seguir possui toda a implementação que precisamos

[3]O termo `pool` de *threads* pode ser traduzido livremente como sendo um **conjunto de** *threads*.

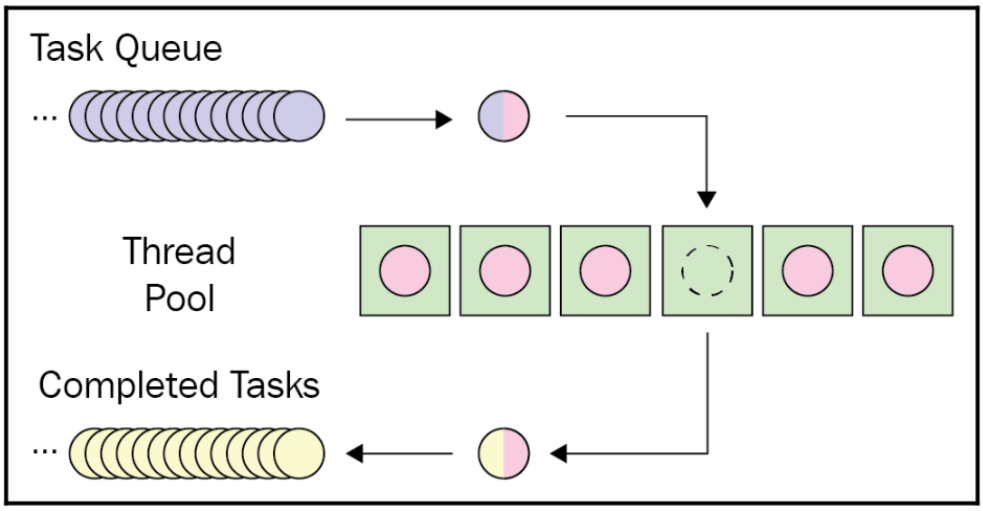

Figura 2.4: Enfileiramento em *threading*.

neste exemplo e que vamos discutir com mais detalhes os pontos que desejamos destacar.

```
#   Problema de fatoracao com threads

import queue
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):
        print("Iniciando thread %s." % self.name)
        processando_fila()
        print("Encerrando thread %s." % self.name)

def processando_fila():
    while True:
        try:
            x = my_queue.get(block=False)
        except queue.Empty:
            return
        else:
```

```
            fatoracao(x)
        time.sleep(1)

def fatoracao(x):
    resultado = 'Fatores positivos de %i sao: ' % x
    for i in range(1, x + 1):
        if x % i == 0:
            resultado += str(i) + ' '
    resultado += '\n' + '_' * 30
    print(resultado)

# Carregando o manipulador de filas
my_queue = queue.Queue()

# Determinando valores a serem fatorados
input_ = [1, 10, 4, 3]

# Inserindo os valores numa fila
for x in input_:
    my_queue.put(x)

#print(my_queue.qsize())

#for j in input_:
#    fatoracao(j)


# Inicializando 3 threads
thread1 = MyThread('A')
thread2 = MyThread('B')
thread3 = MyThread('C')

thread1.start()
thread2.start()
thread3.start()

# Finalizando os 3 threads
thread1.join()
thread2.join()
```

```
63 thread3.join()
64
65 print('\n', 'Concluido')
```

Atente, para começar, a nossa função principal `fatoracao`. Esta função recebe um argumento, x, e então itera por todos os números positivos entre 1 e ele mesmo, para verificar se um número é um fator de x. Finalmente, ela imprime uma mensagem formatada que contém todas as informações que ela acumula através do *loop*.

Na classe `MyThread`, quando ua nova instância é inicializada, a função `processando_fila()` será chamada. Esta função tentará primeiro obter o próximo elemento do objeto `queue` que a variável `my_queue` contém de uma maneira não bloqueante, chamando o método `get(block=False)`. Se ocorrer uma exceção `queue.Empty` (que indica que a fila não contém nenhum valor no momento), então encerramos a execução da função. Caso contrário, simplesmente passamos o elemento que acabamos de obter para a função `fatoracao()`.

A variável `my_queue` é definida em nossa função principal com um objeto `Queue` do módulo `queue` que contém os elmentos na lista `input_`.

Por fim, no restante do programa, simplesmente iniciamos e executamos três *threads* (o nosso `pool` de *threads*), separadas, até que todas elas terminem suas respectivas execuções.

Ao rodar o programa obtemos uma saída semelhante a apresentada a seguir.

```
Iniciando thread A.
Fatores positivos de 1 sao: 1

------------------------------
Iniciando thread B.
Fatores positivos de 10 sao: 1 2 5 10

------------------------------
Iniciando thread C.
Fatores positivos de 4 sao: 1 2 4

------------------------------
Fatores positivos de 3 sao: 1 3
```

```
------------------------------
Encerrando thread B.
Encerrando thread C.
Encerrando thread A.

 Concluido
```

Fila de Prioridade Multithread

Os elementos em uma fila são processados na ordem em que foram adicionados à fila; em outras palavras, o primeiro elemento que é adicionado sai da fila primeiro (FIFO). Embora essa estrutura de dados abstrata simule a vida real em muitas situações, dependendo do aplicativo e seus propósitos, às vezes, precisamos redefinir/alterar a ordem dos elementos dinamicamente. É aqui que o conceito de enfileiramento de prioridades se torna útil.

Na estrutura de dados **pririty queue** cada um dos elementos em uma fila de prioridade, como o nome sugere, tem uma prioridade associada a ele; em outras palavras, quando um elemento é adicionado a uma fila de prioridade, sua prioridade precisa ser especificada. Ao contrário das filas regulares, o princípio de desenfileiramento de uma fila de prioridade depende da prioridade dos elementos: os elementos com prioridade mais altas são processados antes daqueles com prioridades mais baixas.

O conceito de uma fila de prioridade é usado em uma variedade de aplicações diferentes – a saber: gerenciamento de largura de banda, algoritmo de Dijkstra, algoritmo de busca *best-first* e assim por diante. Cada uma dessas aplicações normalmente usa um sistema/função de pontuação definido para determinar a prioridade de seus elementos. Por exemplo, no gerenciamento de largura de banda, o tráfego priorizada como *streaming* em tempo real é processadocom o menor atraso e a menor probabilidade de ser rejeitado. Em algoritmos de melhor busca que são usados para encontrar o caminho mais curto entre dois nós de um grafo, uma fila de prioridade é implementada para manter o controle de rotas inexploradas; as rotas com comprimento de caminho estimados mais curtos recebem prioridades mais altas nas filas.

Sincronização de Threads com Semáforos

Inventado por E. Dijkstra e usado pela primeira vez no sistema operacional, um semáforo é um tipo de dado abstrato gerenciado pelo sistema operacional, usado para sincronizar o acesso por múltiplos `threads` a recursos e dados compartilhados. Essencialmente, um semáforo é constituído de uma variável interna que identifica o número de acessos simultâneos a um recurso ao qual está associado.

Além disso, no módulo de threading, a operação de um semáforo é baseada nas duas funções `acquire()` e `release()`, conforme explicado:

- Sempre que um `thread` quiser acessar um recurso que esteja associado a um semáforo, ele deve invocar a operação `acquire()`, que diminui a variável interna do semáforo e permite acesso ao recurso se o valor dessa variável parecer não negativo. Se o valor for negativo, o `thread` seria suspenso e a liberação do recurso por outro `thread` será colocada em espera.
- Sempre que um `thread` terminar de usar os dados ou o recurso compartilhado, ele deve liberar o recurso por meio da operação `release()`. Dessa forma, a variável interna do semáforo é incrementada, e o primeiro `thread` em espera na fila do semáforo terá acesso ao recurso compartilhado.

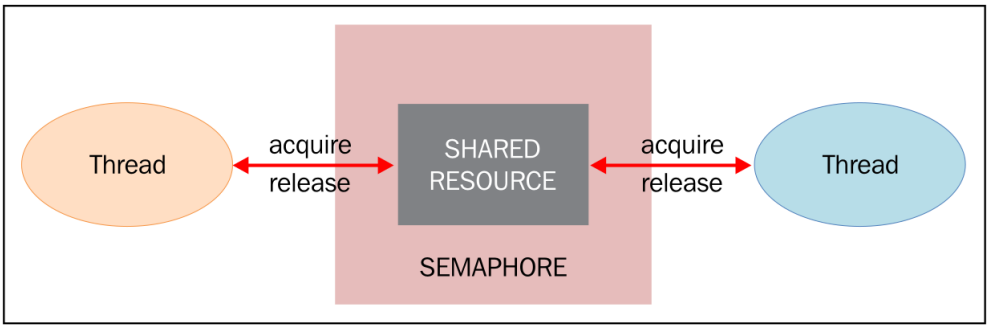

Figura 2.5: Sincronização de `threads` com semáforo.

Embora à primeira vista o mecanismo dos semáforos não apresente problemas óbvios, ele funciona corretamente somente se as operações de espera e sinal forem executadas em blocos atômicos. Se não, ou se uma das duas operações for interrompida, isso pode gerar situações desagradáveis.

Suponha que duas `threads` sejam executadas simultaneamente, a operação espera em um semáforo, cuja variável interna tem o valor 1. Suponha também que após a primeira `thread` ter o semáforo decrementado de 1 para 0, o controle vai para a segunda `thread`, que decrementa a luz de 0 para -1 e espera como o valor negativo da variável interna. Neste ponto, com o controle que retorna para a primeira `thread`, o semáforo tem um valor negativo e, portanto, a primeira `thread` também espera.

Portanto, apesar do semáforo ter acesso a uma `thread`, o fato de que a operação de espera não foi executada em termos atômicos levou a uma solução de paralisação.

O próximo código descreve o problema, onde temos dois `threads`, `produtor()` e `consumidor()` que compartilham um recurso comum, que é o item. A tarefa do `produtor()` é gerar o item enquanto a tarefa do `thread consumidor()` é usar o item produzido.

Se o item ainda não produziu o `thread consumidor()`, ele tem que esperar. Assim que o item é produzido, o `thread produdor()` notifica o `consumidor` que o recurso deve ser usado.

O processo de sincronização feito via semáforos é mostrado no código a seguir:

```
1  #   Sincronizacao de threads via semaforo
2  #   cons_prod_01.py
3
4  import threading
5  import time
6  import random
7
8  #   O argumento opcinal do metodo passa um valor inicial
9  #   para o contador interno da funcao
10 #   Seu valor padrao eh 1
11 #   Caso algum valor negativo seja passado, ValueError eh acionado
12 #   https://docs.python.org/3/library/threading.html#semaphore-example
13 semaphore = threading.Semaphore(0)
14
15 def consumidor():
16     print ("Consmidor esta aguardando")
17     #   Adquirindo um semaforo
18     semaphore.acquire()
19     #   O consumidor tem acesso ao recurso compartilhado
```

```
    print ("Consumidor notificando: consumido item numero %s " % item)


def produtor():
    global item
    time.sleep(10)
    # Criando um item aleatorio
    item = random.randint(0,1000)
    print ("Produtor notificando: produzido item numero %s " % item)

    #   Libera um semaforo, incrementando o contador interno em um.
    #   Quando eh zero na entrada e outro thread esta esperando
    #   que ele fique maior que zero novamente e ative esse thread.
    semaphore.release()


#   Parte principal
if __name__ == '__main__':
    for i in range (0,5) :
        t1 = threading.Thread(target=produtor)
        t2 = threading.Thread(target=consumidor)
        t1.start()
        t2.start()
        t1.join()
        t2.join()
        print ("Programa encerrado")
```

Inicializando um **semáforo** em 0, obtemos um evento cujo único propósito é sincronizar a computação de dois ou mais `threads`. Aqui, um `thread` deve necessariamente fazer uso de dados ou recursos comuns simultaneamente:

```
semaphore = threading.Semaphore(0)
```

O `thread produtor()` cria o `item` (através de uma variável `global`) e, depois disso, libera o recurso chamando:

```
semaphore.release()
```

O método `release()` do semáforo incrementa o contador e então notifica o outro `thread`.

Similarmente, o método `consumidor()` adquire os dados por:

```
semaphore.acquire()
```

Se o contador do semáforo for igual a 0, ele bloqueia o método `acquire()` da condição até que seja notificado por um `thread` diferente. Se o contador do semáforo for maior que 0, ele diminui o valor.

Finalmente, os dados adquiridos são impressos na saída padrão:

```
print ("Consumidor notificando: consumido item numero %s " % item)
```

Sincronização de Threads com Condições

Uma condição identifica uma mudança de estado no aplicativo. Este é um mecanismo de sincronização onde um `thread` espera por uma condição específica e outro `thread` notifica que esta condição ocorreu. Uma vez que a condição ocorre, o `thread` adquire o bloqueio para obter acesso exclusivo ao recurso compartilhado.

Uma boa maneira de ilustrar esse mecanismo é olhar novamente para um problema **produtor/consumidor**. A classe **produtor** grava em um `buffer` enquanto ele não estiver cheio, e a classe **consumidor** pega os dados do `buffer` (eliminando-os deste último), enquanto o `buffer` estiver cheio. A classe **produtor** notificará o **consumidor** de que o `buffer` não está vazio, enquanto o **consumidor** reportará ao **produtor** que o `buffer` não está cheio.

Para mostrar o mecanismo de condição, usaremos novamente o modelo **produtor consumidor**:

```
#   cons_prod_02.py
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumidor(Thread):
    def __init__(self):
```

```
10          Thread.__init__(self)
11
12      def consume(self):
13          global condition
14          global items
15
16          condition.acquire()
17          if len(items) == 0:
18              condition.wait()
19              print("Consumidor notificando: nenhum item a consumir")
20          items.pop()
21          print("Consumidor notificando: consumido 1 item")
22          print("Consumidor notificando: itens a consumir sao " + str(len(items)))
23          condition.notify()
24          condition.release()
25
26      def run(self):
27          for i in range(0, 20):
28              time.sleep(10)
29              self.consume()
30
31  class produtor(Thread):
32      def __init__(self):
33          Thread.__init__(self)
34
35      def produce(self):
36          global condition
37          global items
38          condition.acquire()
39          if len(items) == 10:
40              condition.wait()
41              print("Produtor notificando: itens produzidos sao " + str(len(items)))
42              print("Produtor notificando: producao interrompida!!")
43          items.append(1)
44          print("Produtor notificando: total itens produzidos " + str(len(items)))
45          condition.notify()
46          condition.release()
47
48      def run(self):
```

```
        for i in range(0,20):
            time.sleep(5)
            self.produce()

if __name__ == "__main__":
    producer = produtor()
    consumer = consumidor()
    producer.start()
    consumer.start()
    producer.join()
    consumer.join()
```

A classe `consumidor` adquire o recurso compartilhado que é modelado através da lista `items[]`:

```
    condition.acquire()
```

Se o comprimento da lista for igual a 0, o `consumidor` é colocado em um estado de espera:

```
    if len(items) == 0:
        condition.wait()
```

Caso contrário, ele faz uma operação `pop` da lista de `itens`:

```
    items.pop()
```

Então, o estado do `consumidor` é notificado ao `produtor` e o recurso compartilhado é liberado:

```
    condition.notify()
    condition.release()
```

A classe `produtor` adquire o recurso compartilhado e então verifica se a lista está completamente cheia (em nosso exemplo, colocamos o número máximo de itens, 10, que podem ser contidos na lista de `itens`). Se a lista estiver cheia, então o `produtor` é colocado no estado de espera até que a lista seja consumida:

```
    condition.acquire()
    if len(items) == 10:
```

```
        condition.wait()
```

Se a lista não estiver cheia, um único item é adicionado. O estado é notificado e o recurso é liberado:

```
    condition.notify()
    condition.release()
```

Sincronização de Threads com um Evento

Eventos são objetos usados para comunicação entre `threads`. Uma `thread` espera por um sinal enquanto outra `thread` o emite. Basicamente, um objeto de evento gerencia um sinalizador interno que pode ser definido com `True` com o método `set()` e redefinido como `False` com o método `clear()`.

O método `wait()` bloqueia até que o sinalizador seja `True`.

Para entender a sincronização de `threads` por meio do objeto de evento, vamos dar uma olhada novamente no problema **produtor consumidor**:

```
#   cons_prod_03.py
import time
from threading import Thread, Event
import random

items = []
event = Event()
class consumidor(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
```

```
18              item = self.items.pop()
19              print ('Consumidor notificando: %d retirado da lista por %s' %(item,
                    self.name))
20
21  class produtor(Thread):
22      def __init__(self, integers, event):
23          Thread.__init__(self)
24          self.items = items
25          self.event = event
26
27      def run(self):
28          global item
29          for i in range(100):
30              time.sleep(2)
31              item = random.randint(0, 256)
32              self.items.append(item)
33              print ('Produtor notificando: item N %d adicionado a lista por %s' % (
                    item, self.name))
34              print ('Produtor notificando: evento definido por %s' % self.name)
35              self.event.set()
36              print ('Produtor notificando: evento apagado por %s \n' % self.name)
37              self.event.clear()
38
39  if __name__ == '__main__':
40      t1 = produtor(items, event)
41      t2 = consumidor(items, event)
42      t1.start()
43      t2.start()
44      t1.join()
45      t2.join()
```

Analisem a saída desse código e observem o que obtemos quando executamos o programa. O `thread` t_1 anexa um valor à lista e então define o evento para notificar o consumidor. A chamada do consumidor para `wait()` para de bloquear e o inteiro é recuperado da lista.

A classe produtora é inicializada com a lista de itens e a função `Event()`. Ao contrário do exemplo com objetos de condição, a lista de itens não é `global`, mas é passada como um

parâmetro:

```
class consumer(Thread):
    def __init__(self, items, event):
        Thread._init__(self)
        self.items = items
        self.event = event
```

No método `run` para cada item que é criado, a classe produtora o anexa à lista de itens e então notifica o evento. Há duas etapas que precisamos seguir para isso e a primeira etapa é a seguinte:

```
self.event.set()
```

A segunda etapa é:

```
self.event.clear()
```

A classe consumidora é inicializada com a lista de itens e a função `Event()`. No método `run`, o consumidor espera um novo item para consumir. Quando o item chega, ele é retirado da lista de itens:

```
    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
            item = self.items.pop()
            print ('Consumidor notificando : %d retirado da lista por %s' % (item,
                self.name))
```

Todas as operações entre as classes produtora e consumidora podem ser facilmente retomadas com a ajuda do seguinte esquema:

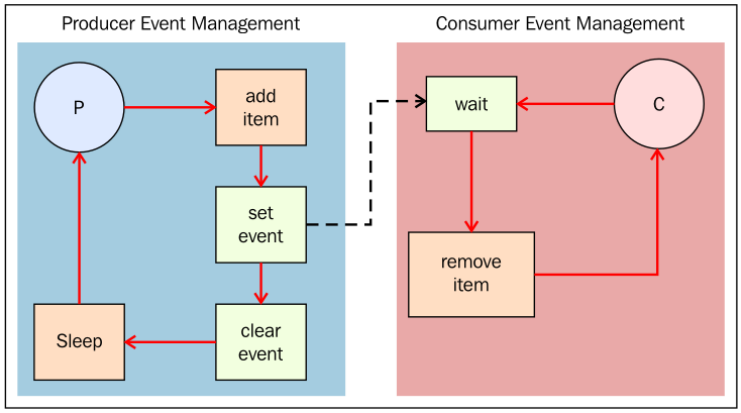

Figura 2.6: `Theads` sincronizadas a partir de eventos.

Referências Bibliográficas

[1] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, Springer, v. 30, p. 483–485, 1967.

[2] ANDREWS, G. R. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Boston, MA: Addison-Wesley, 2000. ISBN 978-0201302013.

[3] HENNESSY, J. L.; PATTERSON, D. A. *Arquitetura de computadores: uma abordagem quantitativa*. 6. ed. Rio de Janeiro: Elsevier, 2019. Tradução da obra original em inglês: Computer Architecture: A Quantitative Approach.

[4] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. 10th. ed. Hoboken, NJ: Wiley, 2018. ISBN 978-1119456339.