O que acontece quando uma nova tarefa é submetida a um ExecutorService no qual não há *threads* disponíveis?

Escolha uma opção:

A. O executor adiciona a tarefa a uma fila interna (*queue*) e completa quando uma *thread* estiver disponível.
B. O executor para uma tarefa existente e inicia a recém-submetida.
C. O executor descarta a tarefa sem completá-la.
D. O executor lança uma exceção quando a tarefa é submetida.
E. A *thread* que submete a tarefa espera na chamada de submissão até que uma *thread* esteja disponível antes de continuar.

Análise da Questão

Vamos falar sobre como funciona o **`ExecutorService`** no Java e o que acontece quando você submete uma nova tarefa a ele e não há threads disponíveis para executá-la imediatamente.

`ExecutorService` é uma interface que fornece uma maneira de gerenciar a execução de tarefas assíncronas. Isso significa que você pode submeter tarefas para execução e elas serão gerenciadas por um pool de threads. O pool de threads é um conjunto de threads reutilizáveis que podem ser usadas para executar várias tarefas em paralelo.

Quando você submete uma nova tarefa ao **`ExecutorService`**, e todas as threads do pool já estão ocupadas, a tarefa não pode ser executada imediatamente. Vamos discutirar as diferentes alternativa da questão acima:

A. **O executor adiciona a tarefa a uma fila interna e a completa quando houver uma thread disponível.**
   - **Correto!** Essa é a resposta certa para o comportamento padrão do `ExecutorService`. Quando não há threads disponíveis, o executor coloca a nova tarefa em uma fila interna. Assim que uma thread fica livre, ela pega a próxima tarefa na fila e a executa. Dessa forma, nenhuma tarefa é perdida, e todas serão executadas assim que possível.
B. **O executor interrompe uma tarefa existente e começa a executar a nova.**
   - **Incorreto.** O `ExecutorService` não interrompe tarefas em andamento para dar prioridade a novas tarefas. Uma vez que uma tarefa é iniciada, ela continua até ser concluída ou até ser interrompida por outro motivo, como uma exceção ou um pedido explícito de cancelamento.
C. **O executor descarta a tarefa sem completá-la.**

- **Incorreto.** Em situações normais, o `ExecutorService` não descarta tarefas submeter sem executá-las. Ele garante que todas as tarefas submetidas sejam eventualmente executadas, a menos que o executor seja explicitamente configurado para descartar tarefas em situações específicas, o que não é o comportamento padrão.

D. **O executor lança uma exceção quando a tarefa é submetida.**
   - **Incorreto.** O `ExecutorService` não lança uma exceção simplesmente porque todas as threads estão ocupadas. Ele vai colocar a tarefa na fila e executá-la assim que possível. Exceções podem ser lançadas se houver outros problemas, como se o executor já tiver sido encerrado.

E. **O thread que está submetendo a tarefa espera na chamada de submissão até que uma thread esteja disponível antes de continuar.**
   - **Incorreto.** O thread que está submetendo a tarefa não espera pela disponibilidade de uma thread. Ele continua sua execução normalmente após submeter a tarefa. A gestão da execução das tarefas é responsabilidade do `ExecutorService`.
   - Portanto, quando você submete uma nova tarefa a um `ExecutorService` e não há threads disponíveis, a tarefa é **adicionada a uma fila interna e executada quando uma thread fica disponível**.

Portanto a resposta correta é a **alternativa A.**

Mas afinal, vamos entender melhor...

O que é o `ExecutorService`?

O `ExecutorService` é uma interface na biblioteca `java.util.concurrent` que fornece um framework para gerenciar um pool de threads, facilitando a execução assíncrona de tarefas. Ele ajuda a lidar com a complexidade de criar e gerenciar threads manualmente, permitindo que você foque mais na lógica do seu programa.

Para que serve?

- **Gerenciamento de Threads**: Ele gerencia a criação, execução e término das threads.
- **Reuso de Threads**: Um pool de threads permite reusar threads existentes em vez de criar novas, o que economiza recursos do sistema.
- **Execução Assíncrona**: Permite executar tarefas em paralelo sem bloquear o thread principal.

Exemplo de Utilização:

Imagine que você está criando um servidor web que precisa lidar com múltiplas requisições de clientes ao mesmo tempo. Usar um `ExecutorService` pode te ajudar a gerenciar essas requisições eficientemente, alocando threads do pool conforme necessário. Segue um exemplo bem simples de sua utilização:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
for (int i = 0; i < 50; i++) {
    executorService.submit(() -> {
        // Código para lidar com a requisição do cliente
        System.out.println("Requisição processada por: " + Thread.curren-
tThread().getName());
    });
}
executorService.shutdown();
```

O código acima cria um `ExecutorService` utilizando `Executors.newFixedThreadPool(10)`, que cria um pool de threads fixo com 10 threads. Em seguida, um loop é executado 50 vezes, onde cada iteração submete uma tarefa anônima para o `ExecutorService` através do método `submit()`. Cada tarefa simplesmente imprime uma mensagem indicando que uma requisição foi processada, junto com o nome da thread que está executando essa tarefa. Após submeter todas as tarefas, o método `shutdown()` é chamado no `ExecutorService` para sinalizar que nenhuma tarefa adicional será submetida, mas as tarefas existentes serão concluídas.

Quando usar:

- **Tarefas Independentes**: Quando você tem várias tarefas que podem ser executadas de forma independente e em paralelo.
- **Desempenho e Escalabilidade**: Quando você precisa melhorar o desempenho e a escalabilidade do seu aplicativo, especialmente em sistemas IO-bound (entrada/saída intensiva) ou CPU-bound (processamento intensivo).

Quando não usar:

- **Tarefas que Compartilham Estado**: Quando suas tarefas compartilham estado mutável que não é adequadamente sincronizado, o que pode levar a problemas de concorrência.
- **Tarefas Simples**: Para tarefas muito simples ou para uma aplicação de pequena escala, o *overhead* de gerenciar um `ExecutorService` pode não valer a pena.

Em relação a resposta correta, vamos verificar na prática que o ExecutorService realmente implementa uma queue para lidar com várias tasks quando o pool de threads está tomado. Usaremos o código abaixo como exemplo:

```java
public class ExecutorServiceExample {
    public static void main(String[] args) {
        // Criando um ExecutorService com um pool de 3 threads
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        // Submetendo 12 tarefas para o pool de threads
        for (int i = 0; i < 12; i++) {
            executorService.submit(() -> {
                System.out.println("Executando tarefa " + Thread.currentThread().getName());
                try {
                    Thread.sleep(2000); // Simulando uma tarefa que leva 2 segundos
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Verificação do estado da fila
        checkQueueState((ThreadPoolExecutor) executorService);

        shutdownAndAwaitTermination(executorService);
    }

    static void checkQueueState(ThreadPoolExecutor threadPoolExecutor) {
        Runnable queueCheckTask = () -> {
            try {
                while (!threadPoolExecutor.isTerminated()) {
                    System.out.println("Fila está vazia? " + threadPoolExecutor.getQueue().isEmpty());
                    System.out.println("Tarefas ativas: " + threadPoolExecutor.getActiveCount());
                    System.out.println("Tarefas concluídas: " + threadPoolExecutor.getCompletedTaskCount());
                    System.out.println("Tarefas na fila: " + threadPoolExecutor.getQueue().size());
                    Thread.sleep(1000); // Espera de 1 segundo antes de verificar novamente
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("Verificação da fila interrompida.");
            }
        };

        Thread queueCheckThread = new Thread(queueCheckTask);
        queueCheckThread.start();
    }

    static void shutdownAndAwaitTermination(ExecutorService pool) {
        pool.shutdown(); // Disable new tasks from being submitted
        try {
            // Wait a while for existing tasks to terminate
            if (!pool.awaitTermination(15, TimeUnit.SECONDS)) {
                pool.shutdownNow(); // Cancel currently executing tasks
                // Wait a while for tasks to respond to being cancelled
                if (!pool.awaitTermination(15, TimeUnit.SECONDS)) {
                    System.err.println("Pool did not terminate");
                }
            }
        } catch (InterruptedException ie) {
            // (Re-)Cancel if current thread also interrupted
            pool.shutdownNow();
            // Preserve interrupt status
            Thread.currentThread().interrupt();
        }
    }
}
```

O resultado no console será:

```
Executando tarefa pool-1-thread-2
Executando tarefa pool-1-thread-1
Executando tarefa pool-1-thread-3
Fila está vazia? false
```

```
Tarefas ativas: 3
Tarefas concluídas: 0
Tarefas na fila: 9
Fila está vazia? false
Tarefas ativas: 3
Tarefas concluídas: 0
Tarefas na fila: 9
Executando tarefa pool-1-thread-2
Executando tarefa pool-1-thread-1
Executando tarefa pool-1-thread-3
Fila está vazia? false
Tarefas ativas: 3
Tarefas concluídas: 3
Tarefas na fila: 6
Fila está vazia? false
Tarefas ativas: 3
Tarefas concluídas: 3
Tarefas na fila: 6
Executando tarefa pool-1-thread-2
Executando tarefa pool-1-thread-3
Executando tarefa pool-1-thread-1
Fila está vazia? false
Tarefas ativas: 3
Tarefas concluídas: 6
Tarefas na fila: 3
Fila está vazia? false
Tarefas ativas: 3
Tarefas concluídas: 6
Tarefas na fila: 3
Executando tarefa pool-1-thread-2
Executando tarefa pool-1-thread-3
Executando tarefa pool-1-thread-1
Fila está vazia? true
Tarefas ativas: 3
Tarefas concluídas: 9
Tarefas na fila: 0
Fila está vazia? true
Tarefas ativas: 3
Tarefas concluídas: 9
Tarefas na fila: 0
```

Podemos perceber que o ExecutorService em Java utiliza um gerenciador de filas para lidar com as tarefas que são submetidas para execução no pool de threads. O gerenciador de filas é responsável por armazenar as tarefas que foram submetidas ao ExecutorService e ainda não foram atribuídas a uma thread disponível para execução.

Explicação do Código

Vamos agora esmiuçar o seu código linha por linha.

1. **Criação do `ExecutorService`**:

```
ExecutorService executorService = Executors.newFixedThreadPool(3);
```

Aqui você cria um pool de threads fixo com 3 threads. Isso significa que, no máximo, 3 tarefas podem ser executadas simultaneamente.

2. **Submissão das Tarefas**:

```
for (int i = 0; i < 12; i++) {
    executorService.submit(() -> {
        System.out.println("Executando tarefa " +
        Thread.currentThread().getName());
        try {
            Thread.sleep(2000); // Simulando uma tarefa que leva 2 se
                                // gundos
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

Aqui você está submetendo 12 tarefas ao pool. Como o pool tem apenas 3 threads, as tarefas excedentes serão enfileiradas.

3. **Verificação do Estado da Fila**:

```
checkQueueState((ThreadPoolExecutor) executorService);
```

Este método verifica periodicamente o estado da fila de tarefas no pool.

4. **Método `checkQueueState`**:

```
static void checkQueueState(ThreadPoolExecutor threadPoolExecutor) {
    Runnable queueCheckTask = () -> {
        try {
            while (!threadPoolExecutor.isTerminated()) {
                System.out.println("Fila está vazia? " + threadPoolExe-
cutor.getQueue().isEmpty());
                System.out.println("Tarefas ativas: "
                + threadPoolExecutor.getActiveCount());
                System.out.println("Tarefas concluídas: "
                + threadPoolExecutor.getCompletedTaskCount());
                System.out.println("Tarefas na fila: "
                + threadPoolExecutor.getQueue().size());
                Thread.sleep(1000); // Espera de 1 segundo antes de ve
                                    // rificar novamente
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
```

```
            System.out.println("Verificação da fila interrompida.");
        }
    };

    Thread queueCheckThread = new Thread(queueCheckTask);
    queueCheckThread.start();
}
```

Este método cria uma nova thread (`queueCheckThread`) que periodicamente verifica e imprime o estado da fila de tarefas, número de tarefas ativas, concluídas e enfileiradas.

5. **Método `shutdownAndAwaitTermination`:**

```
static void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
        // Wait a while for existing tasks to terminate
        if (!pool.awaitTermination(15, TimeUnit.SECONDS)) {
            pool.shutdownNow(); // Cancel currently executing tasks
            // Wait a while for tasks to respond to being cancelled
            if (!pool.awaitTermination(15, TimeUnit.SECONDS)) {
                System.err.println("Pool did not terminate");
            }
        }
    } catch (InterruptedException ie) {
        // (Re-)Cancel if current thread also interrupted
        pool.shutdownNow();
        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}
```

O método `shutdownAndAwaitTermination` é usado para encerrar um `ExecutorService` em duas fases: primeiro, ele chama `shutdown` para rejeitar novas tarefas; depois, se necessário, chama `shutdownNow` para cancelar qualquer tarefa remanescente. Vamos detalhar melhor como ele funciona:

Função `shutdownAndAwaitTermination`:

- Essa função recebe um `ExecutorService` chamado `pool` e faz o fechamento ordenado desse pool de threads, garantindo que todas as tarefas que estavam em execução ou que foram submetidas antes do fechamento sejam completadas ou interrompidas apropriadamente.

`pool.shutdown()`:

- Esse método impede que novas tarefas sejam submetidas ao `pool`, mas as tarefas já submetidas continuarão a ser executadas.

Primeiro `try` bloco:

`pool.awaitTermination(15, TimeUnit.SECONDS)`

- Esse método faz a thread atual aguardar até 15 segundos para que todas as tarefas do `pool` sejam concluídas. Retorna `true` se todas as tarefas terminarem dentro desse tempo; caso contrário, retorna `false`.

Primeiro `if`:

- Se as tarefas não terminarem dentro de 15 segundos (`!pool.awaitTermination(15, TimeUnit.SECONDS)`), executa o método `pool.shutdownNow()`, que tenta cancelar todas as tarefas em execução e retorna uma lista de tarefas que estavam aguardando para serem executadas.

Segundo `if`:

- Depois de chamar `shutdownNow`, o código aguarda mais 15 segundos (`pool.awaitTermination(15, TimeUnit.SECONDS)`) para que as tarefas respondam ao cancelamento.
- Se, após esse tempo, as tarefas ainda não tiverem terminado, imprime uma mensagem de erro: `"Pool did not terminate"`.

Bloco `catch` (foco principal):

`catch (InterruptedException ie)`:

- Esse bloco é acionado se a thread atual for interrompida enquanto está esperando (`awaitTermination`). A `InterruptedException` é lançada nesse caso.

`pool.shutdownNow()`:

- Tenta cancelar todas as tarefas imediatamente, assim como antes.

`Thread.currentThread().interrupt()`:

- Reinterrompe a thread atual. Isso é importante porque a interrupção foi tratada (pegada pelo `catch`), mas o status de interrupção da thread foi limpo quando a exceção foi lançada. Reinterromper a thread restaura o status de interrupção, permitindo que o código acima na pilha de chamadas saiba que a thread foi interrompida. Você pode verificar isso com esse código, se você não restaurar o status da interrupção usando `Thread.currentThread().interrupt();`, o bloco `if` nunca será executado:

```
public class ThreadInterrupted {
```

```
public static void main(String[] args) {

      try {
          // código que pode lançar InterruptedException
          Thread.sleep(1000);
          throw new InterruptedException("Exceção de Interrupcao");
      } catch (InterruptedException e) {
          // restabelece o status de interrupção
          Thread.currentThread().interrupt();
          // lida com a interrupção (se necessário)
          if(Thread.currentThread().isInterrupted()) {
                  System.out.println("Thread foi interrompida");
          }
      }
          }
}
```

O pacote `java.util.concurrent.Executors` em Java oferece diferentes tipos de pools de threads para atender a diversas necessidades de programação concorrente. Neste exemplo trabalhamos com `FixedThreadPool` que mantém um número fixo de threads no pool. Você especifica o número de threads no momento da criação do pool. Mas existem outras estratégias:

1. **SingleThreadExecutor:**

   - Um `SingleThreadExecutor` mantém uma única thread em seu pool. Todas as tarefas submetidas são executadas sequencialmente nessa única thread.
   - É útil em cenários onde você deseja garantir a ordem de execução das tarefas ou quando é importante evitar a concorrência entre tarefas.
   - Exemplo de criação:

     ```
     ExecutorService executorService = Executors.newSingleThreadExecutor();
     ```

2. **CachedThreadPool:**

   - Um `CachedThreadPool` é um pool que cria threads conforme necessário e as reutiliza se estiverem disponíveis, caso contrário, cria uma nova thread.
   - É adequado para situações onde o número de tarefas ou a carga de trabalho podem variar substancialmente, permitindo uma alocação dinâmica de recursos.
   - Exemplo de criação:

     ```
     ExecutorService executorService = Executors.newCachedThreadPool();
     ```

3. **ScheduledThreadPool:**

   - Um `ScheduledThreadPool` é um tipo de `ThreadPoolExecutor` que pode executar tarefas em momentos específicos ou periodicamente.
   - Permite agendar a execução de tarefas com atraso inicial e/ou intervalos fixos.

- Exemplo de criação com 5 threads no pool:

```
ScheduledExecutorService executorService = Executors.newSchedule-
dThreadPool(5);
```

4. **WorkStealingPool:**

- Um `WorkStealingPool` é uma implementação especializada de `ForkJoinPool` que usa a técnica de roubo de trabalho (work-stealing) para balancear automaticamente a carga entre threads.
- É eficaz para algoritmos de divisão e conquista (como processamento paralelo de arrays) onde as sub-tarefas podem ser divididas e distribuídas entre as threads do pool de forma eficiente.
- Exemplo de criação com o número de threads sendo o número de processadores disponíveis no sistema:

```
ExecutorService executorService = Executors.newWorkStealingPool();
```