O que é um sistema operacional?
O sistema operacional é uma camada de software que fica entre o hardware e o usuário, servindo principalmente como um gerenciador de recursos para seus usuários e aplicações.

Quais as funcionalidades básicas de um sistema operacional?
É possível listar 3 funcionalidades básicas, alocação de recursos, isolamento de processos, e comunicação entre os dispositivos de hardware e outros processos.

O que diferencia o sistema operacional do kernel?
O sistema operacional é composto por um kernel, que é o seu cérebro. Basicamente, o kernel trata de questões como tratamento de exceções, gerenciamento de memória, comunicação entre componentes do hardware e processos, enquanto o sistema operacional é algo mais genérico, que apesar de incluir as funcionalidades do kernel, possui também uma interface e um conjunto de utilitários do sistema que permite que o usuário possa se comunicar com o dispositivo, algo como uma abstração.

Ao executar um código qualquer, qual o comportamento do seu sistema operacional?
Para executar um programa, o sistema operacional copia as instruções e os dados da imagem executável criada pelo compilador para a memória física, as instruções ficam num espaço chamao code, e os dados(ou variáveis globais) em uma BSS, também chamada de data. Em seguida, ele reserva uma área, chamada stack, ou pilha de execução, utilizada para manter o estado das variáveis locais durante a execução do processo. Uma outra região, chamada heap, também é criada, esta por sua vez, armazena quaisquer estruturas de dados alocadas dinamicamente, direto no endereço de memória.

Qual a diferença entre a heap e a stack?
A heap é uma área que armazena as informações dinamicamente, mas ao mesmo tempo, não possui uma forma automática de desalocação, pode possuir um tamanho mais variável e seu crescimento não é linear. Já a stack, armazena informações de variáveis locais, e chamadas de funções, sua alocação e desalocação é feita de forma automática pelo compilador, possui um tamanho fixo e seu crescimento se dá de forma linear.
Heap é usada para armazenar objetos e estrutura de dados
Stack é usada para armazenar variáveis locais, parâmetros de funções e seus endereços de retorno.

Por que o processo é considerado uma estrutura de dados?
O que faz o sistema operacional ser uma estrutura de dados, é o fato de precisar organizar suas informações, como em um documento de identificação, de forma que o kernel possa gerenciar melhor a execução de todos os processos de forma uniforme, evitando assim, o desperdício de ciclos de clock. Esse documento de identificação é o PCB, que armazena as informações em uma estrutura de dados.

Como o kernel do sistema operacional impede que um processo prejudique outros processos ou o próprio sistema operacional?
Há diversos caminhos que devem ser bloqueados para impedir que um processo prejudique o kernel ou qualquer outro processo em execução. Um deles, é impedir que o processo

altere seu nível de privilégio, por meio das system calls esse problema é solucionado, afinal, um processo poderá pedir ao kernel que execute uma função que não é permitida para ele. Da mesma forma, um processo também não pode alterar sua área de acesso à memória, assim como não podem desabilitar as interrupções, que são a chave para a retomada do kernel em casos de obstrução do processador.
Há também um mecanismo base & bound, que define as bordas da memória do processo, onde ele não pode ultrapassar nenhuma dessas barreiras, daí sempre que o processador executa, ele verifica se o endereço está entre o base e o bound, se não tiver, lança uma exceção que provavelmente irá encerrar o processo. Obviamente, o kernel não possui base nem bound

Heap e a stack são estruturas de dados dos processos?

Não!
No contexto dos processos, heap e stack são áreas da memória física, designados para armazenar informações importantes que são utilizadas durante a execução dos processos.

O que seria uma interrupção, e o que a diferencia das exceções?

Uma interrupção é gerada periodicamente pelo kernel para verificar se algum processo está causando algum problema em alguma das CPUs, mas também pode ocorrer uma interrupção induzida pelo hardware/usuário, como por exemplo digitar uma tecla, ou algum dado que chegou do HD que foi requisitado há alguns milissegundos. As exceções, são um tipo de interrupção, mas geradas pelo próprio processo em execução, como uma divisão por 0 por exemplo.

O que acontece se um processo tentar acessar memória restrita ou alterar seu nível de privilégio?

Qualquer operação que tente acessar uma memória fora do escopo do processo deve ser feita por meio de uma system call, sendo assim, caso o processo tente fazer alguma operação desse tipo, seria lançada uma exceção, que é tratada em um manipulador de exceções do sistema operacional. Geralmente, se ocorrer uma violação de privilégio, o kernel simplesmente interrompe o programa considerado culpado.

O que faz o unix fork()?

Cria um novo processo identico ao processo pai em outro espaço disponível na memória. Sendo assim, todas as áreas são idênticas, mas não iguais. É como se eles fossem dois processos distintos, com uma única diferença. O pai recebe do fork() o pid do filho, e o filho por sua vez, recebe um 0, que serve para distinguir o processo filho do pai.

O que faz o unix exec()?

O exec, só requer dois parâmetros; O primeiro é o nome do programa a ser executado e uma matriz de argumentos que serão passado para ele, isso significa que ele irá reescrever o processo inteiro com a nova imagem que receber do novo arquivo que for executar.

Por que o fork() seguido de um exec() ainda não foi substituído até os dias de hoje?(Manel pediu pra responder em casa)

Um dos principais motivos para o fork() seguido de um exec() não ter sido substituído ainda, é a simplicidade na ideia de criar um novo processo. No windows, por exemplo, existe um createProcess, mas que requer 10 parâmetros. Uma outra razão para isso, é que após o fork(), o processo filho pode fazer diversas operações antes de se tornar um novo processo. Também tem o fato de que, mesmo que o filho mude sua estrutura totalmente, o pid permanece igual, e seu pai continuará tendo acesso ao mesmo.

O que é, e para que serve o pipe na comunicação entre processos?

O pipe é como se fosse uma corredor que liga um cômodo a outro, mas com uma simples e pequena diferença, de que cada pipe é unidirecional, só um processo pode escrever, e esse pipe liga apenas 2 processos. Geralmente, o pipe liga um pai e um filho, mas é possível implementar um pipe com processos distintos, apesar de ser uma abordagem mais complexa.

Quais os 3 possíveis estados de um processo, e o que pode fazer com que eles mudem de estado?

Running: rodando instruções no processador
Ready: pronto para ser executado, mas o escalonador do SO ainda não decidiu o executar
Blocked: em algum momento, esse processo executou alguma operação que o tirou do processador(por exemplo, uma solicitação de I/O.

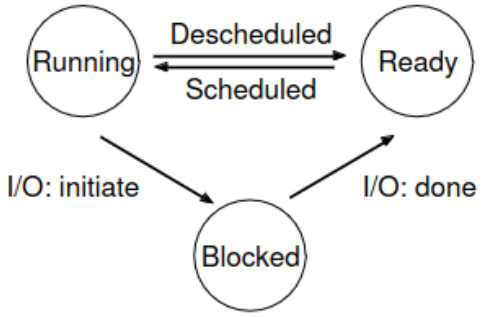

Figure 4.2: **Process: State Transitions**

Um processo Running, pode ser desescalonado, quando precisar fazer algum tipo de chamada ao sistema, isso é uma requisição de I/O, acesso à memória secundária e etc. Processos Running podem ir direto para ready ou para blocked, dependendo do que o fez sair desse estado, como o caso em que o escalonador, por algum motivo, o removeu de lá.
Um processo Ready só pode ir para Running, mas quando ele irá, cabe ao escalonador e à estratégia adotada por ele para escalonar um processo.
Um processo Blocked, só pode ir para Ready. A condição para que ele mude de estado, é que ele conclua a operação que está fazendo. Na maioria dos casos, como o acesso a um dado na memória secundária, ele sai por meio de interrupções, caso ele esteja bloqueado por um semáforo, a saída não gerará uma interrupção.

Qual a necessidade de utilizar um escalonador no sistema operacional para gerenciar qual processo é o próximo a ser executado no processador?

Para responder essa pergunta, já imaginou o que aconteceria se deixássemos que os processos decidissem qual sua prioridade e o quão rápido eles deveriam executar em relação aos outros?
Primeiro que isso significa permitir que ele tivesse acesso à informações de outro processos, e talvez comportamentos inesperados ocorressem. O fato aqui, é que precisamos de alguém que gerencie esses processos. A melhor forma de fazer isso, é dando a responsabilidade para alguém que tenha acesso ao tempo de chegada dos processos, e as métricas utilizadas para definir quando um processo Ready, pode entrar na CPU e executar suas instruções de máquina.

O que são disciplinas no contexto dos escalonadores?

Sabemos que os escalonadores são os responsáveis por escalonar e desescalonar processos do processador, sendo assim, há algumas métricas que quando inseridas em um escalonador, influenciam na ordem da execução dos processos. Duas métrcas contrastantes são a eficiência(quantidade de processos atendidos) e justiça(tempo de espera). Essas ideias requerem soluções, e essas soluções são chamadas de disciplinas, ou políticas de escalonamento.

O que aconteceria se os escalonadores não assumissem uma postura preemptiva?

Algum processo poderia simplesmente tomar conta do processador e só sair de lá quando o dispositivo fosse reiniciado. O escalonador precisa ser preemptivo para garantir que vários processos rodem ao mesmo tempo em um único processador por meio da troca de contexto.

A clássica: Explique como o algoritmo Round-Robin(RR) e o RR com prioridade funcionam impondo suas diferenças.

Como não sabemos o tempo que um processo leva para concluir, e não temos noção do quão injusto seria interferir na ordem dos processos, a solução mais trivial possível é definir um tempo máximo para que um processo fique no processador até que seja removido forçadamente pelo escalonador. Para o round-robin, esse tempo é chamado de **quantum**, que deve ser um múltiplo do tempo entre uma interrupção e outra.

Qual o principal problema do RR utilizar o quantum para definir o tempo que um processo pode ficar no processador? Há alguma forma de melhorar isso?

Sempre que um processo é desescalonado, há um custo, denominado custo de troca de contexto. Isso significa que no momento do PC sair do processo atual, ele ainda precisará copiar as informações que esse processo guarda(como o valor de cada registrador) e a última instrução exeutada para o próprio processo. O mesmo acontece no caso contrário, então se um processo já estava em execução e foi desescalonado, precisará de um tempo para então continuar executando. A troca de contexto ainda possui um custo adicional, que é o fato de que o processo possivelmente estaria utilizando cache e buffers.
Se houvesse alguma forma melhor de decidir quando o processo deve sair do CPU sem saber o tempo que ele levaria para executar, o round-robin não utilizaria o quantum para decidir quando um processo deve sair do CPU.

Sobre as métricas utilizadas para medir o desempenho de um escalonador em relação ao RR.

O RR é muito bom para a métrica tempo de resposta, que é definida pelo momento em que o processo chegou na fila menos o tempo em que foi escalonado. No entanto, quando nos referimos ao turnaround, o tempo total que ele levou desde que começou a executar na CPU até quando concluiu seu trabalho, o RR consegue ser um dos piores escalonadores. Ainda bem que no computador, se precisássemos tomar uma decisão binária(um, ou outro), nosso objetivo seria conseguir atender todos os processos, e não que eles sejam atendidos da forma mais rápida.

Quando o STFC e o SJF é melhor que o RR?

Como é implementado o RR com filas de prioridade? Descreva o algoritmo

O que é uma thread?

Threads é uma sequência de instruções única que representa uma tarefa que pode ser escalonada. As threads são criadas no contexto de um processo, que executam em uma CPU concorrentemente, e que compartilham recursos entre outras threads em um mesmo processo.

Quais as vantagens de utilizar uma thread?

O uso de threads é muito vantajoso, pois:
Permite que tarefas sejam executadas concorrentemente
Permite que tarefas sejam executadas em segundo plano, sem que uma precise esperar que a outra seja concluída para executar
Permite rodar threads distintos em processadores distintos e em paralelo, deixando a execução de um processo ainda mais rápida.
Enquanto uma thread está agurdando por I/O, o CPU pode progredir uma thread diferente.

Como o sistema operacional implementa a ilusão de que há um número infinito de processadores para as threads?

Acontece que quando uma thread está no CPU, o escalonador pode removê-la de lá e colocar em Ready, dando a ilusão para o usuário de que a thread só demorou um pouco para executar, quando na verdade ela chegou a parar de executar por alguns ciclos. Justamente por isso, é possível criar muitas threads, que não vêem que podem estar sendo pausadas de tempo em tempos.

Porque nunca devemos tentar prever o tempo de execução de uma thread?

Assim como em qualquer processo, as threads podem sofrer interferencias externas e realizar determinadas operações que não permitem prever em quanto tempo ela será concluída. Por exemplo, se uma thread estiver esperando por outra thread que estiver sendo depurada pelo usuário, ela só vai parar de esperar, quando o usuário decidir.

Qual o ciclo de vida de uma thread?
Toda thread começa no estado Init e só vai para Ready, quando todas as estruturas de dados relacionados à ela são carregadas completamente. Quando o escalonador decidir que uma thread pode ser escalonada para uma CPU, ele irá carregar os dados dos registradores do TCB para o processador e mudar seu estado para Running. Em seguida, há 3 possíveis caminhos; O primeiro é que a thread passou muito tempo executando, e o escalonador decide parar por um tempo e executar outra thread. Isso significa que ele irá colocá-lo na lista de Ready e pegar outro processo dessa lista para executar. O segundo caminho, é o caso em que a thread precisou fazer uma system call, ou alguma exceção foi lançada. Isso fará com que a thread mude seu estado para blocked(waiting) esperando pelo retorno da chamada efetuada. O terceiro cenário é o qual o processo que estava Running conseguiu executar toda a sua lista de tarefas, ou obteve o erro que o fez ser encerrado precocemente.

Para que serve o TCB da thread?
Assim como o PCB dos processos, o TCB(Thread Control Block) é utilizado como um documento de identificação da thread. Quando o processador realiza a troca de contexto entre threads, a necessidade é a mesma que nos processadores. Uma estrutura para guardar o estado atual da thread é de suma importância para a retomada da mesma posteriormente.

O que diferencia uma thread de um processo?
Um processo é único, e não se comunica tão fácilmente com outros processos, apesar de permitir concorrência. As threads não só também permitem concorrência, e paralelismo como se comunicam com as outras threads a partir das áreas de data e heap do processo pai. A única área que não permanece compartilhada entre threads são as stacks locais e o TCB, que armazenam informações das variáveis locais, pilha de funções e informações sobre a execução.

Como uma thread pode gerar um HeisenBug(bugs que mudam constantemente de comportamento)?
Durante a execução de uma thread, há o risco de ocorrer alguma interferência que vem por meio de fatores externos, como por exemplo outras threads que acessam dados críticos em regiões críticas. Race conditions são o principal causador de HeinsenBugs.

Lista antiga disponibilizada pelos monitores do 24.1:

O que são interrupções e exceções?
Interrupções são eventos gerados por eventos externos, como I/O, chegada de dados do disco ou chegada de pacotes de rede, que fazem com que o CPU pare de executar o processo atual, tirando seu controle, a fim de realizar alguma outra operação. As exceções, também tiram o controle do processo da CPU, mas o responsável por esse feito é o próprio processo, por meio de uma system call.

Explique o que é um processo? Descreva as estruturas de dados que compõem um e as responsabilidades de cada uma.

O processo é uma estrutura de dados responsável por organizar as informações de um conjunto de instruções. Essas instruções ficam armazenadas em uma área estática, geralmente chamada de code. Mas para que ela funcione, são necessários outros 3 componentes para o processo, são eles a DATA(ou bss) que armazena as informações estáticas do programa, que não sofrem alteração durante a execução do mesmo. Há também a HEAP, que guarda informações dinamicamente, e que consegue ser mais poderosa(apesar de mais lenta) que a área da STACK, que assim como a HEAP cresce durante a execução do processo, mas que armazena a informação de uma forma um pouco distinta. Na HEAP geralmente ficam as váriáveis alocadas com malloc() ou estruturas de dados. Na STACK, informações sobre a pilha de retorno e execução das funções e coisas do tipo.

Por que no código abaixo não ocorrem condições de corrida.

```python
import os
import sys
import threading

def do_sum(path):
    _sum = 0
    try:
        with open(path, 'rb', buffering=0) as f:
            byte = f.read(1)
            while byte:
                _sum += int.from_bytes(byte, byteorder='big', signed=False)
                byte = f.read(1)
    except Exception as e:
        print(f"Error processing {path}: {e}")
    finally:
        print(f"{path} : {_sum}")

def process_file(path):
    _sum = do_sum(path)

if __name__ == "__main__":
    paths = sys.argv[1:]
    threads = []

    for path in paths:
        t = threading.Thread(target=process_file, args=(path,))
        t.start()

```

Como e com quais finalidades o SO utiliza os níveis de processamento (modos de execução)?

Primeiramente, não há nenhuma operação sobre variáveis globais por meio das threads. Um outro possível problema, seria se elas também realizassem uma operação de escrita no arquivo de texto. Sendo assim, como cada thread possui sua stack, não há como haver interferência de nenhuma delas com esse tipo de operação.

Diga quais são e explique os estados e as transições entre processos.

a. **Desenhe um diagrama de estados/transições, contendo os 3 principais estados pelos quais um processo pode passar.**

Minha pomba que vou desenhar, já respondi essa lá em cima.

b. Explique os eventos que podem causar essas transições e como o sistema operacional ganha o controle da CPU para realizá-las.

O SO consegue ganhar o controle da CPU por meio de interrupções, sejam elas programadas ou ocasionadas por algum dispositivo de hardware, sendo assim, ele consegue realizar uma transição de estado atráves desse mecanismo. Quanto as possíveis transições:
Ready -> Running: Ocorre quando um processo que estava rodando na CPU perde o controle, e um novo processo pode ser escalonado. Neste caso, um algoritmo combinado à uma estrutura de dados, decide qual será o próximo a executar.
Running -> Ready: Caso esse processo que conseguiu o controle do CPU demore muito tempo executando, ele pode vir a perder o controle da mesma e voltar para o estado Ready.
Running -> Blocked: Há também a possibilidade do processo precisar realizar alguma operação no modo kernel, ou aguardar alguma resposta de algum dispositivo de hardware, sendo assim, ele irá para a lista de blockeds para aguardar a sua resposta.
Running -> Finished: Caso o processo consiga concluir sua última instrução, ele mudará seu estado para finished(independentemente de sucesso, ou não) e o próximo processo a ser escalonado assume a CPU.
Blocked -> Ready: Caso a operação de interação com o hardware tenha sido concluída, ou a exceção lançada ter sido tratada, o processo pode voltar à lista de processos Ready para a retomada de seu trabalho.

Como e por que ocorrem condições de corrida e como evitar?

As condições de corrida só ocorrem, porque o sistema operacional evoluiu para executar múltiplas tarefas visando aproveitar cada ciclo de clock sem desperdício de recursos. No entanto, ficar trocando de contexto entre processos/threads, ou permitir que várias CPU's executem processos/threads distintos ao mesmo tempo, pode fazer com que eles acessem o mesmo local de memória ao mesmo tempo para operações que deveriam ser executadas em momentos distintos, como a escrita em um endereço de memória. Essas situações são chamadas de race conditions, e a melhor maneira de evitar isso, é utilizando uma política de acesso a regiões como estas, isso pode ser por meio de mutexes, semáforos, spin locks e etc.

Como o SO pode executar uma operação down de forma atômica e por que ser atômica?

A operação down precisa ser atômica, pois caso dois processos tentem realizar essa operação ao mesmo tempo, algum deles, ou ambos podem não entrar na lista de blockeds quando deveriam estar lá. Como o semáforo não pode ser lido, uma operação de down apenas reduz em 1 o valor no semáforo. Geralmente, a implementação dos semáforos internamente possui uma operação atômica como down_and_check. Se o valor do semáforo após a operação for negativo, o processo entrará na lista de bloqueados.

Uma vez que um processo do usuário está de posse da CPU, em que condições o sistema operacional pode voltar a executar?

O kernel DEVE executar sempre que uma interrupção for gerada, sendo assim, qualquer tipo de operação advinda do hardware gera uma interrupção que faz com que o kernel assuma a CPU e trate a interrupção para que, só então, se for o caso, o processo retome o controle do CPU. Outra coisa que pode acontecer, é uma exceção lançada pelo próprio

processo, que fará com que o kernel trate a exceção e decida o que fazer com esse processo, como por exemplo, matar.

Suponha um programa que executa as threads D, T1, T2 e T3. D é a Thread despachante, ela é responsável pela recepção de serviços e colocação do pedido num dos três buffers apropriados para aquele serviço. T1, T2 e T3 são responsáveis pela realização de serviços distintos. Cada uma delas tem um buffer associado. Escreva um pseudocódigo usando semáforos de D, T1, T2 e T3 considerando que cada buffer tem capacidade para armazenar no máximo N pedidos. Modifique sua solução para o caso em que todas as threads compartilham um mesmo buffer.

```
mutexBuffer1 = Semaphore(1)
itemsBuffer1 = Semaphore(N)

mutexBuffer2 = Semaphore(1)
itemsBuffer2 = Semaphore(N)

mutexBuffer3 = Semaphore(1)
itemsBuffer3 = Semaphore(N)
```

Thread D:

```
task = getTask()
if task.tipo == T1 then
  mutexBuffer1.wait()
    buffer1.putJob(task)
  mutexBuffer1.signal()
  itemsBuffer1.signal()
else if task.tipo == T2
then
  mutexBuffer2.wait()
    buffer2.putJob(task)
  mutexBuffer2.signal()
  itemsBuffer2.signal()
else then
  mutexBuffer3.wait()
    buffer3.putJob(task)
  mutexBuffer3.signal()
  itemsBuffer3.signal()
```

Thread T1:

```
itemsBuffer1.wait()
mutexBuffer1.wait()
  task = buffer1.getJob()
mutexBuffer1.signal()
process(task)
```

Thread T2:

```
itemsBuffer2.wait()
mutexBuffer2.wait()
  task = buffer2.getJob()
mutexBuffer2.signal()
process(task)
```

Thread T3:

```
itemsBuffer3.wait()
mutexBuffer3.wait()
  task = buffer3.getJob()
mutexBuffer3.signal()
process(task)
```

```
mutexBuffer = Semaphore(1)
itemsBuffer = Semaphore(N)
```

Thread D:

```
task = getTask()
if task.tipo == T1 then
  mutexBuffer.wait()
    buffer.putJob(task)
  mutexBuffer.signal()
  itemsBuffer.signal()
elif task.tipo == T2 then
  mutexBuffer.wait()
    buffer.putJob(task)
  mutexBuffer.signal()
  itemsBuffer.signal()
else then
  mutexBuffer.wait()
    buffer.putJob(task)
  mutexBuffer.signal()
  itemsBuffer.signal()
```

Thread T1:

```
itemsBuffer.wait()
mutexBuffer.wait()
  task = buffer.getJob()
mutexBuffer.signal()
process(task)
```

Thread T2:

```
itemsBuffer.wait()
mutexBuffer.wait()
  task = buffer.getJob()
mutexBuffer.signal()
process(task)
```

Thread T3:

```
itemsBuffer.wait()
mutexBuffer.wait()
  task = buffer.getJob()
mutexBuffer.signal()
process(task)
```

Explique como um escalonador de processos Round-Robin é implementado. Descreva o algoritmo a ser usado, explicitando as estruturas de dados usadas, bem como em que instantes cada parte do algoritmo é executada.

O RR sem filas de prioridade pode ser implementado com uma fila convencional e uma lista para armazenar os processos bloqueados. Na primeira iteração, o escalonador Round Robin irá pegar o elemento da cabeça da fila e escaloná-lo para o processador. Lá, suas instruções começarão a ser executadas. Caso o processo lance uma exceção ele irá para a lista de blocked, caso o quantum do processo acabe(o quantum é o tempo máximo que um processo pode permanecer executando instruções) ele entrará no fim da lista de Ready, e caso o kernel receba uma interrupção, o processo precisará parar de executar para que ela seja processada. Ao receber o que estava esperando, seja um dado do disco, ou uma resposta do usuário, uma interrupção (ou exceção no caso dos semáforos) é gerada, e o processo retorna para a lista de Ready onde será novamente escalonado posteriormente.

Considere duas threads, A e B, que precisam executar dois métodos dependentes entre si, garanta que a chamada do método bar_2() só ocorra após foo(), assim como, executar bar() apenas após foo_2(). Crie um pseudocódigo que lide com essa sincronização.

```
s1 = Semaphore(0);
s2 = Semaphore(0);
```

```
Thread A
    fn foo();
    s1.signal();
    s2.wait();
    fn bar();
```

```
Thread B
    fn foo_2();
    s2.signal();
    s1.wait();
    fn bar_2();
```

Usando semáforos, escreva o pseudocódigo de um tipo abstrato de estrutura de dados, cujas instâncias podem ser compartilhadas por várias threads,que tem como estado um vetor de 10 posições para armazenar inteiros e que implementa as seguintes operações:

a. **void put (int dado): esse método tenta armazenar o parâmetro dado na estrutura de dados que armazena o estado da instância em qualquer posição livre do vetor; se não há espaço para armazenar mais um inteiro, a chamada bloqueia;**
b. **int get (dado); esse método retorna algum inteiro armazenado na estrutura de dados; se não há nenhum dado armazenado, a chamada bloqueia;**
c. **int sumOdd(); esse método retorna a soma dos inteiros armazenados nas posições ímpares do vetor, ou zero se não há nenhum inteiro armazenado nessas posições; e**
d. **int sumEven(); esse método retorna a soma dos inteiros armazenados nas posições pares do vetor, ou zero se não há nenhum inteiro armazenado nessas posições.**

```
items = Semaphore(10);
mutex = Semaphore(1);
lista = [];

def put(int valor):
  mutex.wait();
  items.wait();
  lista.append(valor);
  mutex.signal();

def get(dado):
  mutex.wait();
    retorno = lista.find(dado);
  mutex.signal();
  return retorno
```

```
def sumOdd():
  soma = 0
  mutex.wait();
for i in range(0, len(lista), 2):
    soma += lista[i]
  mutex.signal();
return soma

def sumEven():
  soma = 0
  mutex.wait();
    if len(lista) > 1:
      for i in range(1, len(lista), 2):
        soma += lista[i]
  mutex.signal();
return soma
```

Dado o código abaixo, onde além dos semáforos vazio, cheio, e mutex, é também compartilhado um buffer com um número finito de posições, onde as informações produzidas são armazenadas e de onde as informações a serem consumidas são retiradas:

```
semaphore vazio = X;
semaphore cheio = Y;
mutex = Z;
produtor() {
        while (VERDADE) {
                produz_item();
                down(&vazio);
                down(&mutex);
                adiciona_item();
                up(&mutex);
                up(&cheio);
        }
}
consumidor() {
        while(VERDADE) {
                down(&cheio);
                down(&mutex);
                remove_item();
                up(&mutex);
                up(&vazio);
                imprime_item();
        }
}
```

Indique os valores de X (vazio), Y (cheio) e Z (mutex) para atender às seguintes especificações do problema do produtor consumidor:

a. **Existem 10 posições no buffer, que inicialmente então ocupadas, um produtor e um consumidor.**
   X = 0, Y = 10, Z = 1
b. **Existem 20 posições no buffer, que inicialmente estão vagas, um produtor e um consumidor.**
   X = 20, Y = 0, Z = 1

c. **Existem 15 posições no buffer, das quais, inicialmente, 10 posições estão ocupadas e 5 estão livres, dois produtores e três consumidores.**
   X = 5, Y = 10, Z = 1