Instuderingsfrågor och Svar

1. **Skriv ett program i Go som skapar en goroutine som skriver ut en text på skärmen**

```go
package main
import (
    "fmt"
    "time"
)
func main() {
    go func() {
        fmt.Println("Hello from goroutine")
    }()
    time.Sleep(1 * time.Second) // Vänta så att goroutinen hinner köra
}
```

2. **Skriv ett program i Go som läser ett värde från en kanal och skriver ut det på skärmen**

```go
package main
import "fmt"
func main() {
    ch := make(chan string)
    go func() {
        ch <- "Hello from channel"
    }()
    message := <-ch
    fmt.Println(message)
}
```

3. **Var är GIL i Python och hur påverkar det trådade program i Python?** GIL, eller Global Interpreter Lock, är en mekanism i CPython interpretern som begränsar exekveringen av trådar till en åt gången. Detta påverkar trådade program genom att förhindra dem från att fullt ut utnyttja flerkärniga processorer i situationer där trådarna utför CPU-bound arbete.

4. **Skriv ett program i Go som kan ge upphov till ett data race**

```go
package main
import (
    "fmt"
    "sync"
)
var counter int
func main() {
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
```

```
        go func() {
            counter++
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

5. **Implementera Peterson's algorithm i Go-liknande kod.**

```
package main
import (
    "fmt"
    "time"
)
var turn int
var flag = []bool{false, false}
func petersonsAlgorithm(threadID int) {
    other := 1 - threadID
    flag[threadID] = true
    turn = other
    for flag[other] && turn == other {
        // Vänta
    }
    // Kritisk sektion
    fmt.Printf("Thread %d is in the critical section\n", threadID)
    time.Sleep(1 * time.Second) // Simulerar arbete i den kritiska sektionen
    flag[threadID] = false
}
func main() {
    go petersonsAlgorithm(0)
    go petersonsAlgorithm(1)
    time.Sleep(3 * time.Second) // Vänta så att båda gorutinerna hinner köra
}
```

6. **Implementera en producent i Go-liknande kod som använder mutex för ömsesidig uteslutning**

```
package main
import (
    "fmt"
    "sync"
    "time"
)
var mutex sync.Mutex
var count int
func producer() {
    for i := 0; i < 5; i++ {
        mutex.Lock()
```

```
        count = count + 1
        fmt.Println("Producer produced:", count)
        mutex.Unlock()
        time.Sleep(1 * time.Second)
    }
}
func main() {
    go producer()
    time.Sleep(6 * time.Second) // Vänta så att producenten hinner producera
}
```

7. **Implementera en producent i Go-liknande kod som använder kanaler för ömsesidig uteslutning**

```
package main
import (
    "fmt"
    "time"
)
func producer(ch chan<- int) {
    for i := 0; i < 5; i++ {
        ch <- i
        fmt.Println("Produced:", i)
        time.Sleep(1 * time.Second)
    }
    close(ch)
}
func main() {
    ch := make(chan int)
    go producer(ch)
    for value := range ch {
        fmt.Println("Consumed:", value)
    }
}
```

8. **Ge ett exempel i Go där två goroutines delar en variabel och använder en mutex för att (korrekt) skydda tillgången**

```
package main
import (
    "fmt"
    "sync"
)
var wg sync.WaitGroup
var mutex sync.Mutex
var sharedVar int
func increment() {
    mutex.Lock()
    sharedVar++
    mutex.Unlock()
```

```
    wg.Done()
}
func main() {
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go increment()
    }
    wg.Wait()
    fmt.Println("Value of sharedVar:", sharedVar)
}
```

9. **Ge ett exempel på ett program i Go med kanaler som kan leda till deadlock**

```
package main
func main() {
    ch := make(chan int)
    ch <- 1 // Denna operation väntar på att någon ska läsa från kanalen,
vilket leder till deadlock eftersom det inte finns några andra goroutines.
}
```

10. **Ge ett exempel på ett program i Go utan kanaler som kan leda till deadlock**

```
package main
import (
    "sync"
)
func main() {
    var mutex1, mutex2 sync.Mutex
    go func() {
        mutex1.Lock()
        mutex2.Lock()
        mutex2.Unlock()
        mutex1.Unlock()
    }()
    mutex2.Lock()
    mutex1.Lock()
    mutex1.Unlock()
    mutex2.Unlock()
    // Deadlock kan uppstå om den första goroutinen får tag på mutex1 men
sedan preempts innan den låser mutex2,
    // och huvudgoroutinen lyckas låsa mutex2.
}
```

11. **Implementera en parallel prefix sum med goroutines i Go**

```
// Detta exempel illustrerar en enkel form och kanske inte helt
representerar en optimal parallel prefix sum implementation.
```

```
package main
import (
    "fmt"
    "sync"
)
func parallelPrefixSum(arr []int, result []int, start, end int, wg
*sync.WaitGroup) {
    sum := 0
    for i := start; i < end; i++ {
        sum += arr[i]
        result[i] = sum
    }
    wg.Done()
}
func main() {
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    result := make([]int, len(arr))
    var wg sync.WaitGroup
    parts := 2 // Antalet delar att dela upp datan i
    partSize := len(arr) / parts
    for i := 0; i < parts; i++ {
        wg.Add(1)
        start := i * partSize
        end := start + partSize
        if i == (parts - 1) {
            end = len(arr)
        }
        go parallelPrefixSum(arr, result, start, end, &wg)
    }
    wg.Wait()
    fmt.Println("Result:", result)
}
```

12. **Implementera en parallel prefix scan med goroutines i Go**

```
// Detta exempel är liknande föregående och illustrerar en grundläggande
tillvägagångssätt.
package main
import (
    "fmt"
    "sync"
)
func parallelPrefixScan(arr []int, result []int, start, end int, wg
*sync.WaitGroup) {
    if start == 0 {
        result[start] = arr[start]
        start++
    }
    for i := start; i < end; i++ {
        result[i] = result[i-1] + arr[i]
    }
    wg.Done()
```

```
}
func main() {
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    result := make([]int, len(arr))
    var wg sync.WaitGroup
    parts := 2
    partSize := len(arr) / parts
    for i := 0; i < parts; i++ {
        wg.Add(1)
        start := i * partSize
        end := start + partSize
        if i == (parts - 1) {
            end = len(arr)
        }
        go parallelPrefixScan(arr, result, start, end, &wg)
    }
    wg.Wait()
    fmt.Println("Result:", result)
}
```

13. **Implementera en parallel odd-even transposition sort med goroutines i Go**

```
// Denna implementation är förenklad och kanske inte helt effektiv.
package main
import (
    "fmt"
    "sync"
)
func oddEvenTranspositionSort(arr []int) {
    n := len(arr)
    sorted := false
    for !sorted {
        sorted = true
        for i := 1; i < n-1; i += 2 {
            if arr[i] > arr[i+1] {
                arr[i], arr[i+1] = arr[i+1], arr[i]
                sorted = false
            }
        }
        for i := 0; i < n-1; i += 2 {
            if arr[i] > arr[i+1] {
                arr[i], arr[i+1] = arr[i+1], arr[i]
                sorted = false
            }
        }
    }
}
func main() {
    arr := []int{5, 1, 4, 2, 8, 0, 2}
    oddEvenTranspositionSort(arr)
    fmt.Println("Sorted:", arr)
}
```

14. **Visa hur en for-loop kan paralleliseras i Go med goroutines**

```go
package main
import (
    "fmt"
    "sync"
)
func process(i int, wg *sync.WaitGroup) {
    fmt.Printf("Processing %d\n", i)
    wg.Done()
}
func main() {
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go process(i, &wg)
    }
    wg.Wait()
}
```

15. **Hur kan en barrier implementeras med mutex? Visa i Go-liknande kod**

```go
package main
import (
    "fmt"
    "sync"
)
type Barrier struct {
    total int
    count int
    mutex sync.Mutex
    cond  *sync.Cond
}
func NewBarrier(total int) *Barrier {
    b := &Barrier{total: total}
    b.cond = sync.NewCond(&b.mutex)
    return b
}
func (b *Barrier) Wait() {
    b.mutex.Lock()
    b.count++
    if b.count == b.total {
        b.count = 0 // Reset for next use
        b.cond.Broadcast()
    } else {
        b.cond.Wait()
    }
    b.mutex.Unlock()
```

```
}
func main() {
    var wg sync.WaitGroup
    barrier := NewBarrier(3)
    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            fmt.Printf("Goroutine %d reaching barrier\n", n)
            barrier.Wait()
            fmt.Printf("Goroutine %d passed barrier\n", n)
        }(i)
    }
    wg.Wait()
}
```

16. **Hur kan en barrier implementeras med kanaler? Visa i Go-liknande kod**

```
package main
import (
    "fmt"
    "sync"
)
func barrier(n int, wg *sync.WaitGroup, ch chan bool) {
    wg.Done()
    wg.Wait()
    ch <- true
}
func main() {
    var wg sync.WaitGroup
    ch := make(chan bool)
    wg.Add(3)
    for i := 0; i < 3; i++ {
        go func(n int) {
            fmt.Printf("Goroutine %d reaching barrier\n", n)
            barrier(3, &wg, ch)
            <-ch
            fmt.Printf("Goroutine %d passed barrier\n", n)
        }(i)
    }
    close(ch) // Ensure no deadlock
}
```

17. **Ge en concurrent/parallel algoritm för att hitta det minsta värdet i en lista i Go-liknande kod**

```
package main
import (
    "fmt"
    "sync"
```

```
)
func findMin(nums []int, ch chan int) {
    min := nums[0]
    for _, n := range nums {
        if n < min {
            min = n
        }
    }
    ch <- min
}
func main() {
    nums := []int{4, 2, 6, 3, 1, 5}
    ch := make(chan int, 2)
    go findMin(nums[:len(nums)/2], ch)
    go findMin(nums[len(nums)/2:], ch)
    min1, min2 := <-ch, <-ch
    if min1 < min2 {
        fmt.Println("Min:", min1)
    } else {
        fmt.Println("Min:", min2)
    }
}
```

18. **Vad innebär en atomär variabel i Go** Atomära variabler i Go hanteras genom paketet `sync/atomic`. En atomär operation utförs som en enda enhet utan möjlighet för andra trådar att se den i ett ofullständigt tillstånd, vilket säkerställer konsistens utan att använda tunga låsningsmekanismer.

19. **Visa hur en variabels värde kan jämföras och bytas atomärt i Go**

```
package main
import (
    "fmt"
    "sync/atomic"
)
func main() {
    var value int32 = 2
    if atomic.CompareAndSwapInt32(&value, 2, 3) {
        fmt.Println("Value was 2, now 3")
    }
    fmt.Println("Current value:", value)
}
```

20. **Förklara vad happens-before i en minnesmodell** "Happens-before" är en relation i en minnesmodell som säkerställer minneskonsistens genom att definiera en ordning på åtkomsten till minnet. Om en åtgärd A "happens-before" åtgärd B, garanteras det att ändringar av minnet som utförs av A är synliga för B.

21. **Vilka antaganden kan göras om två trådar modiferar en delad variabel (enligt Javas minnesmodell).** I Javas minnesmodell, om två trådar modiferar en delad variabel utan synkronisering,

kan ingen garanti ges om ordningen i vilken dessa modifikationer sker eller när ändringarna blir synliga för andra trådar. Detta kan leda till race conditions där programmets utfall blir oförutsägbart.

22. **Förklara begreppet multicore cpu** En multicore CPU är en processor som innehåller flera oberoende kärnor som kan läsa och exekvera programinstruktioner. Detta gör det möjligt för datorn att utföra flera processer eller trådar parallellt, vilket ökar dess totala bearbetningskapacitet.

23. **Vad är skillnanden mellan concurrency och parallelism?** Concurrency är konceptet att hantera flera uppgifter samtidigt inom ett program eller system. Parallelism är när dessa uppgifter faktiskt körs samtidigt på olika processorkärnor. Concurrency handlar om struktur, medan parallelism handlar om utförande.

24. **Vad betyder det att ett problem är embarrassingly parallel?** Ett problem är "embarrassingly parallel" om det kan delas upp i många oberoende deluppgifter som kan utföras samtidigt med liten till ingen behov för kommunikation eller beroenden mellan uppgifterna.

25. **Ge exempel på ett problem som är embarrassingly parallel** Exempel på embarrassingly parallel problem inkluderar bildbehandling där varje pixel eller bildblock kan bearbetas oberoende, eller stora simuleringar där beräkningar för varje delsystem är oberoende av andra delsystem.

26. **Förklara begreppet operativsystem** Ett operativsystem (OS) är programvara som hanterar datorns hårdvara och programresurser, och tillhandahåller gemensamma tjänster för datorprogram. Det fungerar som en mellanhand mellan användarna av en dator och datorns hårdvara.

27. **Vad betyder det att ett operativsystem kan ses som ett interface mot datorn?** Att ett operativsystem kan ses som ett interface mot datorn betyder att det tillhandahåller en användarvänlig gränssnitt för människor att interagera med datorns hårdvara, gömmer komplexiteten i hårdvaruhanteringen och tillåter körning och styrning av applikationer.

28. **Vad betyder det att ett operativsystem kan ses som ett kontrollsystem för datorn?** Det innebär att operativsystemet övervakar och styr tillgången till hårdvaruresurser som CPU, minne och I/O-enheter. Det allokerar resurser till olika processer och program, och ser till att systemet fungerar effektivt och rättvist.

29. **Vad är skillnaden på system mode och user mode?** Skillnaden mellan system mode (även kallad kernel mode) och user mode är att i system mode har koden full tillgång till alla hårdvaruresurser och kan köra privilegierade instruktioner. I user mode är koden begränsad för att skydda systemet från potentiellt skadliga eller felaktiga operationer.

30. **Ange fem tillstånd en process kan befinna sig i och förklara vad de innebär**

    - **Ny (New):** Processen har skapats men har ännu inte tilldelats de resurser som krävs för att köra.
    - **Kör (Running):** Processen utför sina instruktioner på processorn.
    - **Väntande (Waiting):** Processen väntar på någon händelse, till exempel I/O-operationer eller signaler från andra processer.
    - **Redo (Ready):** Processen är redo att köra och väntar på att tilldelas processorn.
    - **Avslutad (Terminated):** Processen har avslutats och väntar på att dess resurser ska frigöras av operativsystemet.

31. **Förklara begreppen stack, heap, data och text i relation till processer**

- **Stack:** Ett område av minnet som används för att lagra lokala variabler och kontrollinformation för varje tråd eller funktion. Stacken växer och krymper dynamiskt när funktioner anropas och återvänder.
- **Heap:** Ett område av minnet som används för dynamisk minnesallokering under programkörning. Till skillnad från stacken, hanteras minnet på heapen explicit av utvecklaren (tilldela och frigöra).
- **Data:** Segmentet innehåller globala och statiska variabler som är initialiserade av programmet.
- **Text:** Även kallad kodsegmentet, innehåller detta de faktiska maskininstruktionerna som ska utföras av processorn. Det är ofta skrivskyddat för att förhindra att programmet av misstag modifierar sin egen kod.

32. **Vad lagras i ett processkontrollblock (PCB)?** Ett Processkontrollblock (PCB) lagrar viktig information om processens tillstånd, inklusive processidentifierare (PID), programräknare, registerstatus, minnesallokeringsinformation, öppna filhanterare, säkerhetsattribut, processprioritet och pekare till schemaläggningsköer.

33. **Vad är en context switch?** En context switch är processen där operativsystemet sparar tillståndet för en körande process eller tråd så att den senare kan återupptas. Detta gör det möjligt för CPU:n att växla mellan processer/trådar och utnyttja dess kapacitet effektivt genom multitasking.

34. **Förklara vad som händer under en context switch** Under en context switch sparar operativsystemet tillståndet för den nuvarande processen/tråden till dess PCB och laddar tillståndet för den nästa processen/tråden att köras från dess PCB. Detta innefattar att byta ut registerinnehåll, programräknare och minneskartläggning, vilket gör att flera processer/trådar kan dela på samma CPU utan att störa varandra.

35. **Vad innebär schemaläggning av en process?** Schemaläggning av en process innebär att besluta vilken process eller tråd som ska tilldelas CPU:n att köras vid en given tidpunkt. Schemaläggaren använder olika algoritmer för att optimera prestanda, responsivitet och rättvisa bland processerna.

36. **När tas beslut om schemaläggning?** Beslut om schemaläggning tas vid specifika händelser som när en process blir redo att köras, när en process avslutas eller blockerar för I/O, och vid avbrott från timern som indikerar att den nuvarande processens tilldelade tid har gått ut.

37. **Vad gör dispatcher?** Dispatchern är en komponent av operativsystemets schemaläggare som ansvarar för att genomföra beslutet som tagits av schemaläggaren. Det innebär att byta kontext till den valda processen, vilket innebär att ladda dess tillstånd så att den kan börja eller återuppta körning på CPU:n.

38. **Ange tre kriterier som kan användas för att bestämma vilken schemaläggningsalgoritm som ska användas**

    - **Responsivitet:** Viktigt för interaktiva system där användaren förväntar sig snabba svar.
    - **Rättvisa:** Alla processer får rättvis tillgång till CPU-resurser.
    - **Genomströmning:** Maximera antalet processer som slutförs per tidsenhet.

39. **Vad betyder det att en process är I/O bound?** En process är I/O bound om dess utförande är begränsat av I/O-operationer, vilket betyder att den spenderar mer tid på att vänta på I/O än att utföra beräkningar. Sådana processer gynnas av snabbare I/O-system snarare än snabbare CPU.

40. **Vad betyder det att en process är CPU bound?** En process är CPU bound om dess utförande huvudsakligen begränsas av processorns hastighet. Dessa processer spenderar merparten av sin tid på att utföra beräkningar och gynnas av en snabbare CPU.

41. **Förklara hur schemaläggning sker om First Come, First Served används** Med First Come, First Served (FCFS) schemaläggningsalgoritm, hanteras processerna i den ordning de anländer till kön. Den första processen i kön tilldelas CPU:n och kör till avslut utan preemption. Detta är enkelt att implementera men kan leda till lång väntetid för processer bakom en tidskrävande process.

42. **Förklara hur schemaläggning sker om Round Robin används** Round Robin-algoritmen tilldelar varje process en liten tidsskiva (kvantum) och kör dem i tur och ordning. Om en process inte avslutas under sin tidsskiva, placeras den i slutet av kön. Detta fortsätter i en loop, vilket ger alla processer en rättvis CPU-tid och reducerar väntetiden för interaktiva processer.

43. **Förklara preemptive schemaläggning** Preemptive schemaläggning innebär att operativsystemet kan avbryta en körande process och återföra den till redo-kön för att ge plats åt en annan process. Detta används för att förbättra systemets responsivitet och för att säkerställa att högprioriterade processer får CPU-tid när de behöver det.

44. **Förklara non-preemptive schemaläggning** I non-preemptive schemaläggning, när en process väl börjat köra, fortsätter den att köra tills den blockerar (t.ex. för I/O) eller avslutas. Den nuvarande processen kan inte avbrytas av schemaläggaren till förmån för en annan process, vilket kan leda till mindre effektiv CPU-användning jämfört med preemptive schemaläggning.

45. **Vad är ett quantum?** Quantum, även kallat tidsskiva, är den tid en process tillåts köra i en preemptive multitasking miljö, som i Round Robin-schemaläggning. Efter att en process har kört för en kvantumtid, avbryts den så att nästa process i kön kan köra.

46. **Hur påverkar ett quantum schemaläggningen?** Storleken på kvantumet påverkar direkt systemets responsivitet och genomströmning. Ett kort quantum förbättrar responsiviteten genom att tillåta processer att växla ofta, men kan öka overhead för context switching. Ett långt quantum minskar overhead men kan göra systemet mindre responsivt.

47. **Vad är viktigt att tänka på när man bestämmer quantum?** Vid bestämning av quantumstorleken är det viktigt att balansera systemets responsivitet mot overhead för context switching. Quantumstorleken bör vara tillräckligt lång för att utföra meningsfullt arbete men tillräckligt kort för att hålla systemet responsivt, särskilt i en miljö med många interaktiva processer.

48. **Förklara begreppen isolation och encapsulation i relation till processer** Isolation innebär att varje process körs i sitt eget skyddade adressutrymme, vilket förhindrar att processer stör eller korrumperar varandras data. Encapsulation refererar till att en process endast kan åtkomma sina egna resurser och data, skyddade av operativsystemet, vilket främjar säkerhet och stabilitet i systemet.

49. **Vad är skillnaden på en process och en tråd?** En process är en oberoende enhet som innehåller ett program under körning, inklusive kod, data och systemresurser. En tråd är en lättviktig process som delar processens resurser men kan köras parallellt inom samma process. Trådar har sitt eget stackutrymme men delar kod, data och andra resurser.

50. **Vad används stacken till i en tråd?** Stacken i en tråd används för att lagra lokala variabler, parametrar och returadresser för funktionanrop. Varje tråd har sin egen stack som gör det möjligt att hålla koll på

dess individuella utförande historia och tillstånd.

51. **Var är skillnaden på user-level- och kernel-level-trådar?**
    - **User-level trådar:** Dessa trådar hanteras och schemaläggs helt av användarprocessen utan direkt inblandning av operativsystemets kärna. De är snabbare att skapa och växla mellan än kernel-trådar, men en blockerad tråd kan blockera hela processen eftersom kärnan inte är medveten om de individuella trådarna.
    - **Kernel-level trådar:** Dessa trådar hanteras direkt av operativsystemets kärna. Detta gör dem långsammare att skapa och växla mellan, men en blockerad tråd påverkar inte andra trådar i samma process. Dessutom kan kärnan schemalägga trådar från olika processer parallellt på multiprocessorsystem.
52. **Förklara begreppet blockerande anrop** Blockerande anrop innebär att ett anrop till en funktion eller systemkallelse pausar utförandet av den anropande tråden eller processen tills operationen är slutförd. Under denna tid kan tråden eller processen inte utföra något annat arbete, vilket kan leda till ineffektivitet om resursen den väntar på är långsam.
53. **Vad händer om en user-level tråd gör ett blockerande anrop?** Om en user-level tråd gör ett blockerande anrop, kommer hela processen att blockeras och vänta på att anropet slutförs. Detta beror på att operativsystemets kärna inte kan skilja på de enskilda user-level trådarna och ser hela processen som en enhet.
54. **Vad händer om en kernel-level tråd gör ett blockerande anrop?** Om en kernel-level tråd gör ett blockerande anrop, påverkas bara den specifika tråden. Andra trådar inom samma process eller i andra processer kan fortsätta att köras obehindrat eftersom operativsystemets kärna kan hantera och schemalägga varje kernel-level tråd individuellt.
55. **Varför vill man dela minne mellan trådar?** Att dela minne mellan trådar möjliggör effektiv kommunikation och datautbyte utan behov av dyra interprocesskommunikationsmekanismer. Det gör det också möjligt för trådar att samarbeta om att utföra uppgifter genom att manipulera gemensamma datastrukturer och resurser.
56. **Vad är skillnaden mellan att dela minne mellan trådar och processer?** Att dela minne mellan trådar innebär att trådarna inom samma process kan åtkomma och modifiera samma minnesområden direkt eftersom de delar processens adressutrymme. Delning av minne mellan processer kräver speciella interprocesskommunikationsmekanismer (som delat minne, meddelandeköer) eftersom varje process har ett eget isolerat adressutrymme.
57. **Förklara begreppet sequentially consistent i relation till minne** Sequentially consistent minne innebär att resultatet av alla minnesoperationer ser ut som om de hade utförts i en strikt sekventiell ordning, även om de i verkligheten kan ha utförts parallellt eller i olika ordningar på olika processorer. Det garanterar att alla trådar ser alla minnesoperationer i samma ordning.
58. **Vad innebär ett data race/race condition?** Ett data race eller race condition uppstår när två eller fler trådar eller processer försöker ändra eller läsa en delad resurs samtidigt utan lämplig synkronisering, vilket leder till oförutsägbart beteende eller felaktiga resultat beroende på exekveringsordningen.
59. **Vad är en kritisk sektion?** En kritisk sektion är en del av koden som åtkommer en delad resurs som inte får modifieras samtidigt av flera trådar. För att förhindra data races, måste åtkomst till kritiska

sektioner synkroniseras så att endast en tråd åt gången kan exekvera den kritiska sektionen.

60. **Ge exempel på ett data race** Ett exempel på ett data race är två trådar som samtidigt försöker uppdatera en global räknare utan synkronisering. Om båda trådarna läser räknarens värde, inkrementerar det, och sedan skriver tillbaka det nästan samtidigt, kan en av uppdateringarna skrivas över, vilket resulterar i att räknaren endast ökas med 1 istället för 2.

61. **Vad innebär ömesidig uteslutning?** Ömsesidig uteslutning är en princip som säkerställer att när en tråd utför en kritisk sektion av koden, inga andra trådar kan exekvera samma eller någon annan kritisk sektion som manipulerar samma delade resurser. Detta förhindrar data races och garanterar korrekthet i åtkomsten till delade data.

62. **Hur kan ömesidig uteslutning lösas? Ge exempel.** Ömsesidig uteslutning kan uppnås genom användning av olika synkroniseringsmekanismer, såsom:

    - **Lås (Mutex):** Ger exklusiv åtkomst till en resurs för en tråd åt gången.
    - **Semaphores:** En mer generell mekanism som kan begränsa antalet trådar som åtkommer en resurs.
    - **Monitorer:** Ett högnivåkoncept som inkapslar synkroniserade metoder eller operationer på en resurs.

    ```
    var mutex sync.Mutex
    func criticalSection() {
        mutex.Lock()
        // Kritisk sektion: modifiera delade resurser här
        mutex.Unlock()
    }
    ```

63. **Förklara hur strict alteration försöker lösa ömesidig uteslutning. Vilka problem finns?** Strict alteration är en teknik för att uppnå ömsesidig uteslutning där två processer alternerar strikt mellan att ha tillåtelse att köra sina kritiska sektioner. Detta implementeras ofta med en delad variabel som håller koll på vem som har tur att exekvera. Problemet med denna metod är att den kan leda till onödig väntan (busy waiting) och inte är skalbar till flera processer eller trådar.

64. **Förklara Peterson's algoritm för ömsesidig uteslutning** Petersons algoritm är en lösning på problemet med ömsesidig uteslutning för två processer/trådar. Den använder två flaggor (en för varje process/tråd) och en turn-variabel för att avgöra vilken process som får tillträde till sin kritiska sektion. Varje process sätter sin flagga till `true` för att indikera att den vill köra sin kritiska sektion och sätter `turn` till den andra processens ID. Processen får köra sin kritiska sektion endast om den andra processens flagga är `false` eller om det är dess tur (baserat på `turn`-variabeln). Efter att ha kört sin kritiska sektion, sätter processen sin flagga till `false`.

65. **Vad innebär progression i relation till ömesidig uteslutning?** Progression inom ömsesidig uteslutning innebär att om en process eller tråd begär tillträde till sin kritiska sektion, kommer systemet så småningom att tillåta den att fortsätta. Det garanterar att begäranden om åtkomst till kritiska sektioner inte ignoreras för alltid, vilket förhindrar låsningar (deadlocks) och hunger (starvation).

66. **Vad innebär begränsad väntan i relation till ömesidig uteslutning?** Begränsad väntan är ett krav i algoritmer för ömsesidig uteslutning som säkerställer att det finns en övre gräns på antalet gånger andra processer får tillträde till sina kritiska sektioner efter att en process har begärt tillträde till sin kritiska sektion och innan begäran beviljas. Detta förhindrar att en process väntar oändligt länge på att få köra sin kritiska sektion.

67. **Vad är en semafor?** En semafor är en synkroniseringsmekanism som används för att kontrollera åtkomst till en gemensam resurs genom flera processer i ett konkurrent system eller av flera trådar i en multiprogrammerad miljö. Semaforer kan vara binära (liknar mutex) eller kunna räkna, vilket tillåter ett visst antal trådar att åtkomma en resurs samtidigt.

68. **Vad är ett mutex lock?** Ett mutex lock (mutual exclusion lock) är en synkroniseringsmekanism som används för att säkerställa ömsesidig uteslutning när flera trådar försöker åtkomma samma resurser. Ett mutex tillåter endast en tråd att åtkomma den kritiska sektionen åt gången, vilket förhindrar race conditions.

69. **Vad är skillnaden mellan en binär och en räknande semafor?** En binär semafor, eller mutex, är en typ av semafor som endast kan ha två tillstånd: låst eller olåst. Den används för ömsesidig uteslutning. En räknande semafor tillåter ett visst antal trådar att åtkomma en resurs samtidigt, där "räknaren" anger det maximala antalet trådar som kan ha samtidig åtkomst.

70. **Visa hur en räknande semafor kan implementeras med hjälp av binära semaforer**
Implementeringen av en räknande semafor med binära semaforer (mutexes) kräver en kombination av en mutex för att skydda tillgången till räknaren och en mekanism för att blockera och väcka trådar baserat på räknarens värde. Här är en konceptuell beskrivning snarare än specifik kod:

```
Initialize:
count = N // N är antalet tillåtna trådar att åtkomma resursen samtidigt
mutex = Semaphore(1) // Binär semafor/mutex för att skydda räknaren
blockQueue = Queue() // Kö för trådar som väntar på åtkomst

wait():
mutex.acquire()
if count > 0:
    count -= 1
else:
    blockQueue.enqueue(thread)
    mutex.release()
    block() // Blockera den här tråden
mutex.release()

signal():
mutex.acquire()
if blockQueue not empty:
    thread = blockQueue.dequeue()
    wakeup(thread) // Väck upp tråden
else:
    count += 1
mutex.release()
```

Denna pseudo-kod visar grundprincipen för hur en räknande semafor kan simuleras med en binär semafor. `wait()`-metoden minskar räknaren om det finns tillgängliga "platser" och blockerar tråden om inte. `signal()`-metoden väcker en blockerad tråd om sådan finns, eller ökar räknaren om inte.

71. **Vilka risker finns det med låsning?** Låsning medför flera risker, inklusive:
    - **Deadlocks:** Kan uppstå när två eller fler processer väntar på varandra för att frigöra lås, vilket skapar en cirkulär beroendekedja där ingen kan fortsätta.
    - **Starvation:** När en eller flera trådar inte får tillgång till en resurs för en orimligt lång tid på grund av andra trådars långvariga eller upprepade åtkomst.
    - **Priority Inversion:** När en högre prioriterad tråd väntar på en resurs som hålls av en lägre prioriterad tråd, vilket leder till att systemets prestanda blir sämre än förväntat.
    - **Överdriven låsning:** Kan leda till minskad parallellitet och sämre systemprestanda.

72. **Vad är ett deadlock?** Ett deadlock uppstår när två eller flera processer/trådar väntar på varandra för att frigöra resurser eller lås, vilket skapar en situation där ingen av dem kan fortsätta. Varje process i deadlocksituationen väntar på en resurs som hålls av en annan process i samma uppsättning.

73. **Vad innebär det att en process är deadlocked?** När en process är deadlocked, är den permanent blockerad och kan inte fortsätta sin exekvering eftersom den väntar på en resurs eller ett lås som aldrig kommer att frigöras av de andra processerna/trådarna inblandade i deadlocksituationen.

74. **Vilka fyra villkor krävs för deadlock?** De fyra villkoren som måste vara uppfyllda för att en deadlock ska inträffa är:
    - **Ömsesidig uteslutning:** Minst en resurs måste vara i ett icke-delbart läge, där bara en process kan använda resursen åt gången.
    - **Håll och vänta:** Processer håller minst en resurs och väntar på att få ytterligare resurser som hålls av andra processer.
    - **Ingen fördrivning (No Preemption):** Resurser kan inte tas från processer; de måste frigöras frivilligt.
    - **Cirkulär väntan:** Det finns en uppsättning av processer {P1, P2, ..., Pn} där varje process Pi väntar på en resurs som hålls av Pi+1 (och Pn väntar på P1), vilket skapar en sluten kedja av beroenden.

75. **Förklara hold and wait** "Hold and wait" är ett av de villkor som kan leda till deadlock. Det uppstår när processer håller på en eller flera resurser samtidigt som de väntar på att få ytterligare resurser som är upptagna av andra processer. Detta villkor bidrar till potentiella deadlocks eftersom det skapar en situation där processer kan blockera varandra.

76. **Förklara no preemption** "No preemption" refererar till principen att en resurs inte kan tas bort från en process förrän processen frivilligt frigör resursen. Detta villkor är nödvändigt för ett deadlock eftersom om systemet kunde fördriva resurser, skulle det kunna bryta deadlocks genom att tvinga bort resurser från blockerade processer.

77. **Förklara circular wait** "Circular wait" uppstår när det finns en stängd kedja av processer där varje process håller på minst en resurs som nästa process i kedjan behöver. Den sista processen i denna kedja väntar på en resurs som hålls av den första processen, vilket skapar en onödigt cirkulär beroendekedja där inga processer kan fortsätta.

78. **Förklara hur deadlock kan förhindras** Deadlock kan förhindras genom att bryta minst ett av de fyra villkoren som är nödvändiga för att ett deadlock ska inträffa:
    - **Eliminera ömsesidig uteslutning:** Gör det möjligt för flera processer att dela vissa resurser, om möjligt.
    - **Undvik hold and wait:** Tilldela alla nödvändiga resurser till en process innan exekvering börjar eller kräv att processer frigör alla resurser innan de begär nya.
    - **Tillåt preemption:** Gör det möjligt att fördriva resurser från processer om de resurserna behövs för att undvika deadlock.
    - **Undvik circular wait:** Inför en ordning på hur resurser begärs för att förhindra stängda kedjor av beroenden.

79. **Hur kan mutual exclusion förhindras för att undvika deadlock?** För att förhindra mutual exclusion och därmed minska risken för deadlock, kan systemet försöka minimera antalet resurser som kräver ömsesidig uteslutning eller använda algoritmer som tillåter delning av resurser där det är möjligt utan att äventyra integritet eller prestanda.

80. **Hur kan hold and wait förhindras för att undvika deadlock?** För att förhindra hold and wait kan systemet kräva att en process begär alla de resurser den behöver samtidigt och blockerar processen tills alla resurser kan tilldelas samtidigt. Alternativt kan systemet kräva att processer frigör alla innehavda resurser innan de begär nya.

81. **Hur kan no preemption förhindras för att undvika deadlock?** För att förhindra no preemption, kan systemet tillåta att resurser fördrivs från en process under vissa omständigheter. Detta kan innebära att implementera en policy där processer med lägre prioritet tvingas ge upp resurser till förmån för processer med högre prioritet.

82. **Hur kan circular wait förhindras för att undvika deadlock?** Circular wait kan förhindras genom att införa en global ordning på alla resurser och kräva att varje process begär resurser enligt denna ordning. Genom att följa en fastställd ordning för resursallokering, kan systemet undvika de cirkulära beroendekedjor som leder till deadlock.

83. **Ge exempel på hur en semafor kan användas så att hold and wait inte gäller** Ett sätt att använda semaforer för att undvika hold and wait-tillståndet är genom att införa en schemaläggare som kontrollerar resursallokeringen. Processerna måste begära alla nödvändiga resurser på en gång från schemaläggaren. Schemaläggaren använder en semafor för att kontrollera åtkomsten till resurserna och tilldelar dem endast när alla begärda resurser kan tilldelas samtidigt. Om inte alla resurser kan tilldelas, frigörs de resurser som redan tilldelats så att ingen process håller på resurser samtidigt som den väntar på ytterligare resurser.

```
semaphore scheduler = 1

request_resources():
    wait(scheduler)
    // Begär alla nödvändiga resurser
    if all resources available:
        allocate resources
    else:
        release any allocated resources
```

```
            place process in wait queue
        signal(scheduler)

    release_resources():
        wait(scheduler)
        // Frigör alla innehavda resurser
        signal(scheduler)
        // Försök väcka processer i väntekön om resurser nu är tillgängliga
```

84. **Varför kan det vara problematiskt att förhindra no preemption** Att förhindra no preemption kan vara problematiskt eftersom det kräver att systemet kan avbryta processers arbete och ta tillbaka resurserna de använder. Detta kan vara tekniskt svårt att implementera på ett säkert sätt, särskilt för resurser som inte enkelt kan sparas och återställas (t.ex. nätverksanslutningar eller realtidsberäkningar). Dessutom kan det leda till prestandaförluster och ökad komplexitet i systemdesignen.

85. **Varför kan det vara problematiskt att förhindra hold and wait** Att helt förhindra hold and wait kan vara problematiskt eftersom det kan kräva att processer måste känna till alla resurser de kommer att behöva i förväg innan de startar, vilket inte alltid är praktiskt eller möjligt. Detta kan också leda till ineffektiv resursanvändning eftersom resurser måste allokeras i stora block, även om de inte alla används samtidigt.

86. **Förklara starvation. Varför är det ett problem?** Starvation, eller svält, uppstår när en eller flera processer inte får tillgång till de resurser de behöver för att fortsätta sitt arbete under en orimligt lång tid. Detta blir ett problem eftersom det kan leda till oacceptabla prestanda för vissa processer, speciellt i system där vissa processer kan dominera resurstillgången på bekostnad av andra. Starvation undergräver systemets rättvisa och kan orsaka att viktiga processer aldrig slutförs.

87. **Vad är en resource allocation graph och hur används den i samband med deadlock?** En resource allocation graph är en grafisk representation av vilka processer som håller på vilka resurser och vilka processer som väntar på resurser. Noder representerar processer och resurser, och kanter visar relationerna mellan dem (till exempel, en kant från en process till en resurs indikerar att processen håller resursen, och en kant från en resurs till en process indikerar att processen väntar på resursen). Denna graf kan användas för att upptäcka potentiella deadlocks genom att leta efter cykler; en cykel indikerar en uppsättning processer och resurser som är involverade i en deadlock.

88. **Hur kan man avgöra om ett system är i deadlock med hjälp av en resource allocation graph?** Ett system anses vara i deadlock om dess resource allocation graph innehåller minst en cykel. En cykel i grafen innebär att det finns en uppsättning processer som väntar på varandra i en sluten kedja, vilket är en direkt indikation på en deadlock-situation. Genom att analysera grafen för att identifiera sådana cykler kan systemadministratörer eller algoritmer upptäcka och åtgärda deadlocks.

89. **Är det ok att ignorera deadlock?** Det är generellt inte ok att ignorera deadlock i de flesta produktionssystem eftersom det kan leda till allvarliga prestandaproblem eller till och med fullständig systemstopp. Att ignorera deadlock kan vara acceptabelt i mycket särskilda fall där deadlocks är extremt sällsynta, konsekvenserna är minimala, eller systemet är utformat för att hantera sådana situationer på annat sätt (t.ex. genom omstart av tjänster).

90. **Vad är skillnaden på att förhindra och undvika deadlock?** Att förhindra deadlock innebär att designa systemet och dess algoritmer på ett sådant sätt att de fyra villkoren för deadlock aldrig kan uppfyllas. Å

andra sidan, innebär att undvika deadlock att låta systemet och processerna nå tillstånd där de potentiellt kan orsaka deadlock, men använda dynamisk resursallokering och processstyrning för att säkerställa att systemet aldrig faktiskt går in i en deadlock. Undvikande kräver ofta mer sofistikerad logik och övervakning av systemets tillstånd.

91. **Förklara Banker’s algoritm** Banker's algoritm är en resursallokerings- och deadlockundvikande algoritm som används i operativsystem. Den liknar en bankir som lånar ut pengar. Algoritmen håller koll på tillgängliga, allokerade, och maximala resurser krävda av processer för att säkerställa att systemet alltid är i ett säkert tillstånd där det kan undvika deadlocks. För varje resursbegäran kontrollerar algoritmen om att tilldela resurserna fortfarande skulle lämna systemet i ett säkert tillstånd. Om ja, tillåts begäran; om inte, väntar processen för att undvika en potentiell deadlock.

92. **Hur kan ett system återhämta sig från deadlock?** System kan återhämta sig från deadlocks genom:

    - **Processavslutning:** Avbryta en eller flera processer som är involverade i deadlocken.
    - **Resursfördrivning:** Selektivt ta bort resurser från blockerade processer och tilldela dem till andra för att bryta deadlocken. Dessa metoder kan använda olika kriterier, såsom prioritet, resursanvändning, och väntetid för att bestämma vilka processer som ska avbrytas eller vilka resurser som ska fördrivas.

93. **Förklara prefix sum** Prefix sum är en operation som för varje element i en sekvens genererar en summa av alla element upp till det elementet. Exempelvis, för sekvensen [3, 1, 4, 1, 5], är prefixsummorna [3, 4, 8, 9, 14]. Detta kan användas i flera algoritmiska applikationer, inklusive parallella algoritmer för att effektivisera beräkningar.

94. **Förklara prefix scan** Prefix scan är en generalisering av prefix sum där operationen inte nödvändigtvis är en summa utan kan vara en annan associativ operation, såsom minimum, maximum, eller multiplikation. Detta möjliggör beräkning av en sekvens där varje element representerar resultatet av operationen applicerad på alla föregående element och sig självt.

95. **Förklara odd-even transposition sort** Odd-even transposition sort är en enkel parallell sorteringsalgoritm som liknar bubbel sort. Den fungerar genom att upprepade gånger genomföra två pass över elementen: ett som jämför och eventuellt byter plats på element på udda positioner med deras nästa grannar (dvs. elementen på position 1 med 2, 3 med 4, etc.) och ett som gör detsamma för element på jämna positioner. Denna process upprepas tills listan är sorterad.

96. **Varför är det en dålig idé att skapa nya trådar för varje rekursivt anrop i en divide and conquer-algoritm?** Att skapa en ny tråd för varje rekursivt anrop i en divide and conquer-algoritm kan snabbt leda till överdriven användning av systemresurser, som trådar och stackutrymme, vilket kan minska prestanda och i värsta fall orsaka program- eller systemkrascher på grund av resursbrist.

97. **Varför kan det vara svårt att parallelisera rekursiva algoritmer?** Att parallelisera rekursiva algoritmer kan vara svårt på grund av beroenden mellan rekursiva anrop som måste lösas eller hanteras korrekt. Dessutom kan den överhängande kostnaden för att hantera trådar och synkronisering uppväga de potentiella prestandafördelarna, speciellt för små problemstorlekar eller djupt rekursiva funktioner.

98. **Vad är en barrier** En barrier är en synkroniseringsmekanism som används i parallell programmering för att stoppa alla involverade trådar eller processer vid en viss punkt tills alla har nått denna punkt. Det säkerställer att ingen tråd fortsätter förrän alla trådar har nått bariären och är redo att fortsätta tillsammans.

99. **Hur kan en semafor användas för att signalera mellan trådar?** En semafor kan användas för att signalera mellan trådar genom att en tråd väntar (kallar `wait` på semaforen) på en signal och en annan tråd skickar denna signal (kallar `signal` eller `post` på semaforen). Detta möjliggör synkronisering mellan trådar, där en tråd kan vänta på att en annan ska slutföra en uppgift innan den fortsätter.

100. **Förklara hur depth-first search (DFS) kan paralleliseras med trådar** DFS kan paralleliseras genom att tilldela olika grenar av sökträdet till olika trådar. När en tråd når en nod som har outredda barn, kan den skapa nya trådar för att utforska dessa barn parallellt. Det är viktigt att synkronisera åtkomsten till gemensamma resurser, som den delade datan eller resultatuppsamlingen, för att undvika inkonsekvenser.

101. **Förklara hur breadth-first search (BFS) kan paralleliseras med trådar** BFS kan paralleliseras genom att använda en gemensam kö som håller på noderna som ska utforskas. Trådar kan ta bort noder från kön och utforska dem samtidigt. När en tråd utforskar en nod, lägger den dess barn i kön. Synkronisering av kön är nödvändig för att förhindra race conditions. Denna metod kan sprida ut arbetsbelastningen jämnt mellan trådarna och effektivisera sökningen över en graf.

102. **Förklara hur Prim's algoritm kan paralleliseras med trådar** Prim's algoritm för att hitta ett minimalt spännande träd i en graf kan paralleliseras genom att dela upp grafen i sektioner och låta varje tråd köra algoritmen på sin sektion. Trådarna måste synkroniseras när de väljer den minsta kanten som korsar gränserna mellan sektionerna. Effektiv synkronisering och delning av information mellan trådar är avgörande för att upprätthålla algoritmens korrekthet och effektivitet.

103. **Vi kan hitta det minsta värdet i en lista på linjär tid ($O(N)$). Hur lång tid tar det att köra med P processorer? Motivera.** Om en uppgift som tar linjär tid $O(N)$ delas jämnt mellan P processorer, och varje processor arbetar på sin del av uppgiften parallellt, skulle den totala tiden teoretiskt kunna reduceras till $O(N/P)$, förutsatt att arbetsbelastningen fördelas jämnt och overhead för kommunikation och synkronisering mellan processorer är försumbar. I praktiken kan dock kommunikationskostnader och ojämn arbetsfördelning minska denna idealiska acceleration.

104. **Vad krävs för att vi skall erhålla en speedup på P med en algoritm som körs på P processorer** För att uppnå en speedup på P med P processorer krävs det: - Att arbetsbelastningen kan delas upp effektivt i P oberoende enheter av arbete. - Minimal overhead för kommunikation och synkronisering mellan processorer. - Att arbetsbelastningen är tillräckligt stor för att motivera den parallella overheaden. - Att algoritmen och problemet inte har några inneboende sekventiella beroenden som begränsar parallellisering.

105. **Är det alltid snabbare att parallelisera en algoritm och köra den på så många processorer som möjligt? Motivera.** Nej, det är inte alltid snabbare. Amdahls lag beskriver en gräns för den teoretiska maximala speedup som kan uppnås genom att parallelisera en beräkning. Överhead för kommunikation och synkronisering, samt delen av algoritmen som måste köras sekventiellt, begränsar speedup som kan uppnås. Dessutom, om antalet processorer ökar utan att arbetsbelastningen ökar i motsvarande grad, kan den extra overheaden för att hantera fler processorer minska den totala prestandavinsten.

106. **Förklara N-ary-sökning** N-ary-sökning är en generalisering av binärsökning där data delas upp i N segment istället för två vid varje steg. Vid varje iteration väljer algoritmen ett av N segment baserat på jämförelser, vilket reducerar sökområdet betydligt snabbare än binärsökning om N väljs korrekt och data är lämpligt organiserad för en sådan sökning.

107. **Förklara hur N-ary-sökning kan paralleliseras med trådar** N-ary-sökning kan paralleliseras genom att tilldela varje segment till en separat tråd för samtidig sökning. Trådarna kan utforska sina tilldelade segment oberoende av varandra, vilket minskar den totala söktiden. Effektiv parallelisering kräver dock noggrann hantering av trådsynkronisering och minimering av overhead för trådskapande och -hantering.

108. **Förklara lock free** Lock-free programmering refererar till designen av algoritmer som garanterar systemets framsteg utan att använda traditionella låsningsmekanismer som mutexar eller semaforer. Dessa algoritmer undviker lås genom att använda atomära operationer, vilket minskar risken för deadlock och kan ge bättre prestanda genom att minska kontextväxling och väntetider.

109. **Förklara wait free** Wait-free programmering är en starkare form av lock-free programmering där systemet inte bara garanterar framsteg utan också att varje tråd slutför sin operation inom ett begränsat antal steg, oavsett andra trådars beteende. Detta säkerställer att systemet är robust mot trådinterferens och ger maximal responsivitet.

110. **Vad menas med en optimistisk algoritm (med avseende på ömesidig uteslutning)?** En optimistisk algoritm antar att konflikter och resurskollisioner är sällsynta och hanterar dem när de inträffar istället för att försöka förhindra dem i förväg. Dessa algoritmer försöker vanligtvis genomföra operationer utan låsning och använder kontroller efteråt för att verifiera att ingen konflikt har uppstått. Om en konflikt upptäcks, återställs och upprepas operationen.

111. **Förklara compare and set** Compare and set (CAS) är en atomär operation som används i flertrådade program för att uppnå synkronisering utan lås. Operationen jämför värdet av en minnesplats med ett givet värde och, endast om de är lika, uppdaterar minnesplatsen med ett nytt värde. Detta sker i en enda, odelbar operation som garanterar att inga andra trådar kan ändra minnesplatsen samtidigt.

112. **Visa hur compare and set kan användas för att implementera en binär semafor**

````
```pseudo
bool locked = false

void lock() {
    while (!compare_and_set(&locked, false, true)) {
        // Vänta tills låset blir ledigt
    }
}

void unlock() {
    locked = false
}
```
Denna pseudo-kod använder en `compare_and_set` operation för att implementera en enkel låsmekanism. `lock`-funktionen loopar tills den lyckas ändra `locked` från `false` till `true`, vilket indikerar att låset har förvärvats. `unlock`-funktionen sätter sedan enkelt tillbaka `locked` till `false`.
````

113. **Förklara hand over hand locking** Hand over hand locking, även känd som lock coupling, är en teknik som används vid traversering av länkade datastrukturer, som länkade listor eller träd, där varje nod låses i tur och ordning. När en tråd rör sig från en nod till nästa, låser den nästa nod innan den frigör låset på den aktuella noden. Detta minimerar den låsta regionen till två noder åt gången och förbättrar parallell tillgång.

114. **Vad är nackdelarna med att låsa "för mycket"?** Att låsa för mycket kan leda till minskad parallellitet och systemprestanda, ökad väntetid för trådar, risk för deadlock och onödigt resursutnyttjande. Överanvändning av lås kan skapa flaskhalsar där trådar blockeras onödigt länge, vilket minskar den totala effektiviteten i programmet.

115. **Vad är nackdelarna med att låsa "för lite"?** Att låsa för lite kan leda till race conditions, inkonsistenta data och svårdebuggade problem. Utan tillräcklig synkronisering kan trådar samtidigt modifiera delade resurser på ett sätt som bryter mot programmets logik och förväntningar, vilket resulterar i felaktiga resultat eller krascher.

116. **Vad är fördelarna med en optimistisk algoritm (med avseende på låsning)?** Fördelarna med optimistiska algoritmer inkluderar högre prestanda under låg och måttlig belastning genom att undvika låskostnader, minskad risk för deadlock eftersom de inte håller kvar lås under operationer, och bättre skalbarhet i miljöer med många processorer genom att minska behovet av synkronisering.

117. **Vad är nackdelarna med en optimistisk algoritm (med avseende på låsning)?** Nackdelarna inkluderar potentialen för ökad komplexitet i algoritmimplementationen, risk för att arbete måste göras om om konflikter upptäcks, och möjlig minskning av prestanda under hög belastning på grund av omgörningar. Optimistiska algoritmer kan också kräva noggrann utformning för att effektivt hantera konflikter.

118. **Förklara hur radering i en länkad lista kan implementeras med en optimistisk algoritm** Vid radering i en länkad lista med en optimistisk algoritm, försöker man först genomföra raderingen utan att låsa hela listan. Istället kontrollerar man efteråt om operationen var korrekt med avseende på listans övriga struktur. Detta kan innebära att man temporärt låser enskilda noder snarare än hela listan. Om en inkonsekvens upptäcks, återställs operationen och försöks igen. Detta tillvägagångssätt minskar låsningens överhead men kräver noggranna kontroller för att upprätthålla dataintegriteten.

119. **Ange några problem med att skriva flertrådade program** - **Race conditions:** När flera trådar försöker ändra delade data samtidigt utan tillräcklig synkronisering. - **Deadlocks:** När två eller flera trådar väntar på varandra för att frigöra resurser, vilket skapar en oändlig väntecykel. - **Starvation:** När en eller flera trådar inte får tillräckligt med processortid eller resurser för att utföra sina uppgifter. - **Livelocks:** När trådar är upptagna med att reagera på varandra på ett sätt som hindrar dem från att göra framsteg. - **Överhead för synkronisering:** Att hantera trådsäkerhet kan lägga till komplexitet och minska prestanda.

120. **Varför kan det vara svårt att sätta samman (compose) flera flertrådade funktioner?** Att kombinera flertrådade funktioner kan vara svårt på grund av ökad risk för deadlocks, race conditions och andra synkroniseringsproblem. Funktioner som fungerar korrekt isolerat kan interagera oväntat när de kombineras, särskilt om de delar resurser eller om deras synkroniseringsmekanismer inte är kompatibla. Detta kräver noggrann design och testning för att säkerställa korrekt samverkan.

121. **Vad är en Future?** En Future är en abstraktion som representerar ett resultat som ännu inte finns tillgängligt men som kommer att bli det i framtiden. Futures används ofta i asynkron programmering för att hantera resultat från parallella eller asynkrona operationer, och de tillåter kod att fortsätta att köra medan väntar på att operationen slutförs.

122. **Förklara begreppet coroutine** Coroutines är programkomponenter som generaliserar subrutiner för att tillåta flera ingångspunkter för suspendering och återupptagande av utförande vid vissa punkter. Detta möjliggör asynkrona operationer på ett mer naturligt och effektivt sätt än traditionella callbacks, genom att tillåta sekventiell kodstruktur för asynkrona operationer.

123. **Förklara begreppet cooperative multitasking** Cooperative multitasking är en typ av multitasking där processer eller trådar frivilligt överlämnar kontrollen tillbaka till schemaläggaren. Detta skiljer sig från preemptive multitasking där schemaläggaren tvingar processen att ge ifrån sig kontrollen. I cooperative multitasking måste varje process aktivt bestämma när den ska "pausa" sin exekvering, vilket minskar behovet av komplexa synkroniseringsmekanismer men kräver noggrann design för att undvika att en process monopoliserar processortiden.

124. **Förklara begreppet asynchronous programming** Asynchronous programming är en programmeringsmodell som tillåter operationer, speciellt I/O-bound operationer, att köras utan att blockera programmets huvudflöde. Detta uppnås genom att initiera operationer som kommer att slutföras i framtiden, medan programmet fortsätter att köra annan kod. Callbacks, promises, futures och coroutines är vanliga konstruktioner som används för att hantera asynkrona operationer.

125. **Förklara begreppet callback** En callback är en funktion som skickas som ett argument till en annan funktion och som är avsedd att kallas vid ett senare tillfälle, oftast som svar på en viss händelse eller när en asynkron operation är slutförd. Callbacks används för att hantera asynkrona händelser eller för att anpassa beteendet hos generiska funktioner.

126. **Vad händer om en blockerade funktion körs av en funktion på event loop** Om en blockerande funktion körs på en event loop, kan det blockera hela event-loopen från att behandla andra händelser eller utföra andra asynkrona operationer. Detta kan leda till att programmet blir otillgängligt eller långsamt, eftersom inga andra åtgärder kan utföras medan den blockerande funktionen körs.

127. **När bör man använda asynchronous programming?** Asynkron programmering bör användas när man hanterar I/O-operationer, nätverksförfrågningar, databasåtgärder eller andra operationer som kan ta tid att slutföra och där man inte vill blockera programmet från att fortsätta med andra uppgifter. Det är särskilt användbart i GUI-applikationer, webbservrar och i situationer där man vill förbättra programmets responsivitet eller prestanda.

128. **Vad är en kanal i Go?** En kanal i Go är en typ som används för att möjliggöra säker kommunikation och synkronisering mellan goroutines. Kanaler tillåter att värden skickas från en goroutine till en annan, vilket underlättar samordning och utbyte av data i konkurrenta program.

129. **Vad gör select i Go?** `select`-satsen i Go låter en goroutine vänta på flera kommunikationsoperationer, inklusive kanalsändningar och mottagningar, att bli redo. `select` blockerar tills någon av dess fall kan utföras, varvid det fallet exekveras. Om flera fall är redo, väljs ett slumpmässigt.

130. **Förklara begreppet done-kanal** En done-kanal används i Go för att signalera till en eller flera goroutines att avsluta sitt arbete. Detta mönster används för att rent och säkert avbryta pågående operationer eller för att rensa upp resurser när ett arbete är klart eller programmet avslutas.

131. **Visa (i Go-liknande kod) hur en goroutine kan avslutas med hjälp av en done-kanal**

```go
package main

import (
    "fmt"
    "time"
)

func worker(done chan bool) {
    fmt.Println("Working...")
    time.Sleep(time.Second)
    fmt.Println("Done")

    // Signalera att arbetet är klart
    done <- true
}

func main() {
    done := make(chan bool, 1)
    go worker(done)

    // Vänta på att arbetet ska bli klart
    <-done
}
```

132. **Implementera en funktion i Go som bestämmer det största värdet som skickas på en kanal**

```go
package main

import (
    "fmt"
)

func findMax(values chan int, maxFound chan int) {
    max := <-values // Startvärde
    for v := range values {
        if v > max {
            max = v
        }
    }
    maxFound <- max
}

func main() {
```

```
    values := make(chan int)
    maxFound := make(chan int)

    go findMax(values, maxFound)

    // Skicka värden
    values <- 3
    values <- 6
    values <- 2
    values <- 8
    values <- 4
    close(values) // Stäng kanalen för att indikera att inga fler värden kommer

    // Ta emot det största värdet
    max := <-maxFound
    fmt.Println("Det största värdet är:", max)
}
```

133. **Implementera en funktion i Go som delar en kanal i flera kanaler (multiplex)**

```go
package main

import (
    "fmt"
)

// multiplex tar en inputkanal och delar upp dess innehåll i två separata
outputkanaler.
func multiplex(input chan int) (chan int, chan int) {
    out1 := make(chan int)
    out2 := make(chan int)

    go func() {
        for val := range input {
            out1 <- val
            out2 <- val
        }
        close(out1)
        close(out2)
    }()

    return out1, out2
}

func main() {
    input := make(chan int)
    out1, out2 := multiplex(input)
```

```
    // Skicka data till inputkanalen
    go func() {
        for i := 0; i < 5; i++ {
            input <- i
        }
        close(input)
    }()

    // Ta emot data från båda outputkanalerna
    for i := 0; i < 5; i++ {
        fmt.Println(<-out1, <-out2)
    }
}
```

134. **Vad är ett closure i Go?** Ett closure i Go är en funktion som kan fånga upp och använda variabler från den omgivande kontexten där den är definierad. Closures är användbara för att skapa funktioner på flygten som behåller tillstånd mellan exekveringar eller för att skapa privata dataområden.

135. **Förklara WaitGroup i Go** En `WaitGroup` i Go används för att synkronisera arbetet mellan flera goroutines. Den tillåter ett program att vänta tills en uppsättning goroutines har slutfört sitt arbete. Man anropar `Add` för att sätta antalet goroutines att vänta på, `Done` inuti varje goroutine för att signalera att den är klar, och `Wait` för att blockera tills alla goroutines har anropat `Done`.