- x

Skrivet blandats av Chat GPT och oss själva, rätta gärna till om ni ser något fel

1. Förklara begreppet multicore cpu

En multicore CPU (Central Processing Unit) är en typ av processor som innehåller flera kärnor eller processingenheter på ett enda chip. Varje kärna är i princip en separat processor som kan arbeta oberoende av de andra kärnorna på samma chip.

2. Vad är skillnaden mellan concurrency och parallelism?

Concurrency innebär att man startar flera processer och kör dem. De behöver inte nödvändigtvis köra samtidigt men kan. En processor kan växla mellan att utföra olika uppgifter vilket ger en illusion av att de körs samtidigt. Parallelism innebär att två saker verkligen körs vid samma tidpunkt.

3. Skriv ett program i Java som skapar en tråd som skriver ut en text på skärmen

```
public class Test extends Thread {
   public void run() {
      System.out.println("Hello World!");
   }

   public static void main(String[] args) {
      Test thread = new Test();
      thread.start();
   }
}
```

4. Skriv ett program i Python som skapar en tråd som skriver ut en text på skärmen

```
from threading import Thread

def printer():
   print("Hello World!")

thread =Thread(target=printer)
thread.start()
thread.join()
```

5. Skriv ett program i Python som skapar en process som skriver ut en text på skärmen

```
from multiprocessing import Process

def printer():
   print("Hello World!")

if __name__ == "__main__":
   processor = Process(target=printer)
   processor.start()
   processor.join()
```

6. Förklara Runnable och Callable i Java
Runnable är ett interface som har en inbyggd funktion *run()*. run startar en ny tråd som körs utan att returnera något. Callable är också ett interface som har en funktion *call()*. Denna funktion startar en tråd som körs och returnerar ett värde av den specificerade typen.

7. Vad betyder det att ett problem är embarrassingly parallel?
Att ett problem är embarrassingly parallel betyder att det kan delas upp i mindre delproblem som kan lösas samtidigt och är oberoende av varandra, utan att behöva kommunicera med varandra under beräkningsprocessen. Detta är en idealisk situation för parallellisering och gör det möjligt att utnyttja många processorer eller trådar för att lösa problemet mycket snabbare än om det skulle lösas sekventiellt. Detta gäller särskilt för problem med stora mängder data eller beräkningar, där parallellisering kan ge en mycket stor prestandaförbättring.

8. Ge exempel på ett problem som är embarrassingly parallel
Exempel 1) Ett exempel på embarrassingly parallel är att beräkna pi med hjälp av monte carlo simulering där man genererar massa slumpmässiga punkter i en kvadrat och räkna hur stor andel av dessa som hamnar inom en cirkel som får plats i denna kvadraten. Denna process kan köras av flera trådar samtidigt och sedan räkna ut pi med hjälp av resultaten.

Exempel 2) Ett exempel på ett embarrassingly parallel problem är att söka igenom en stor mängd data för att hitta en viss förekomst eller ett mönster. Till exempel kan man leta efter en viss sträng i en stor mängd textdata eller söka igenom en stor samling av bilder för att hitta en specifik bild.

Exempel 3) En annan exempel på ett embarrassingly parallel problem är att utföra simuleringar eller beräkningar på stora mängder data. Till exempel kan man simulera vädermönster eller ekonomiska scenarier på stora datamängder, där varje datapunkt kan beräknas oberoende av de andra.

9. Vad är GIL i Python och hur påverkar det trådade program i Python?
GIL (Global Interpreter Lock) låser Python-interpretern så att bara en tråd kan exekvera åt gången, oavsett hur många trådar som körs på en multiprocessor eller flerkärnig dator. Detta innebär att trådar i Python inte kan utnyttja flera kärnor samtidigt, vilket kan påverka prestandan negativt i vissa scenarier. Speciellt när man vill köra trådbaserade program.

10. Förklara begreppet operativsystem
Ett operativsystem (OS) är en programvara som är ansvarig för att hantera och övervaka hårdvaru resurserna och tillhandahålla tjänster och funktioner som används av andra program på en dator eller annan enhet. Operativsystemet är gränssnittet mellan datorns hårdvara och användarens programvara.
Funktionerna som operativsystemet tillhandahåller inkluderar bland annat hantering av filer och kataloger, hantering av minne och resurser, upprättande och avslutning av processer och trådar, hantering av in- och utmatning från enheter som skärmar, tangentbord och mus, samt säkerhet och skydd av systemresurser och data.

11. 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 operativsystemet fungerar som en mellanhand mellan användarens programvara och datorns hårdvara. Operativsystemet tillhandahåller ett gränssnitt som används av program för att interagera med hårdvaran. Användaren kan använda operativsystemets gränssnitt för att kontrollera datorn och köra program.

12. Vad betyder det att ett operativsystem kan ses som ett kontrollsystem för datorn?
Att ett operativsystem kan ses som ett kontrollsystem för datorn betyder att operativsystemet har en övergripande kontroll över datorn och dess komponenter. Operativsystemet övervakar och reglerar tillgången till datorns resurser, såsom processortid, minne och lagring. Operativsystemet är också ansvarigt för att tillhandahålla ett gränssnitt för att hantera datorns inställningar, konfigurationer och användarnas åtkomst. Genom att fungera som ett kontrollsystem kan operativsystemet hantera datorns resurser effektivt och säkerställa att de används på ett lämpligt sätt.

13. Vad är skillnaden på system mode och user mode?
System mode och user mode är två olika tillstånd som en processor kan vara i. User mode är ett begränsat tillstånd där program körs med begränsade privilegier och begränsad tillgång till systemresurser, medan system mode är ett högre privilegierat tillstånd där operativsystemet och dess drivrutiner körs och har fullständig kontroll över systemresurserna. I system mode kan man göra saker som användarprogram inte kan göra, såsom att direkt läsa och skriva till systemresurser och använda hårdvaran på ett direkt sätt.

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

1. New: När en ny process skapas, befinner den sig i tillståndet "New". I det här tillståndet har processen ännu inte fått några resurser eller minne tilldelade till den.

2. Ready: När processen är redo att exekveras, men väntar fortfarande på att CPU-tid ska tilldelas, befinner den sig i tillståndet "Ready". Processen är redo att köra men måste vänta på sin tur att få CPU-tid.

3. Running: När processen exekverar instruktioner befinner den sig i tillståndet "Running". I det här tillståndet tilldelas processen CPU-tid och exekverar sin kod.

4. Blocked: När processen väntar på en resurs som inte är tillgänglig, såsom till exempel att läsa in data från en hårddisk eller att vänta på en annan process, befinner den sig i tillståndet "Blocked". Processen kan inte fortsätta förrän den nödvändiga resursen blir tillgänglig.

5. Terminated: När processen är färdig med att exekvera och avslutas befinner den sig i tillståndet "Terminated". Processen frigör all tilldelad minnes- och CPU-resurs till systemet.

6. Suspended: När en process har blivit pausad av systemet eller användaren och inte längre exekverar sin kod, befinner den sig i tillståndet "Suspended". I det här

tillståndet kan processen antingen väckas upp och fortsätta exekvera eller återgå till "Terminated" tillståndet.

7. Zombie: När en process har avslutats men dess information fortfarande finns kvar i systemet, befinner den sig i tillståndet "Zombie". Processen är inte längre aktiv men dess information lagras tills det att dess parent process hämtar informationen om dess avslut.

8. Orphan: När en process vars parent process avslutas, befinner den sig i tillståndet "Orphan". Processen kan fortfarande fortsätta exekvera men utan en parent process och det är nu operativsystemets ansvar att hantera den processen.

15. Förklara begreppen stack, heap, data och text i relation till processer
Stack är en del av minnet som används för att lagra variabler och funktionella anrop. När en funktion kallas, skapas ett nytt "stack frame" som innehåller alla variabler och funktionella anrop för den funktionen. När funktionen avslutas, tas stackframen bort och processorn återgår till den tidigare stackframen.

Heap är en del av minnet som används för att dynamiskt tilldela minne under exekvering. Objekt som skapas under exekvering som inte har en fördefinierad storlek kan tilldelas minne från heapen. Det är upp till programvaran att hantera minnet som allokeras från heapen, inklusive att frigöra minne när det inte längre behövs.

Data används för att lagra variabler med en fördefinierad storlek, såsom globala variabler eller variabler som är definierade i en funktion men inte lagras på stacken.

Text används för att lagra programkoden som instruktioner som ska exekveras. Detta minnesområde innehåller instruktionerna som definieras av programkod, såsom instruktioner för att läsa in en fil eller skriva ut en sträng.

16. Vad lagras i ett processkontrollblock (PCB)?
PCB används av operativsystemet för att lagra information om en process. Informationen som lagras i ett PCB inkluderar processens identifierare (PID), programräknare (PC), statusregister, stackpekare (SP), minnesallokering och övriga resurser. PCB gör det möjligt för operativsystemet att hantera processen effektivt, inklusive hantering av dess minnesanvändning och resursanvändning samt tilldela CPU-tid och andra resurser till processen.

17. Vad är en context switch?
När en processor går från att jobba på en process till att jobba på en annan process för att sedan återgå till den första. För att göra detta måste processorn spara alla tillstånd från den första processen för att sedan återgå till den.
18. Förklara vad som händer under en context switch
En context switch (kontextbyte) är en operation i en dator som innebär att datorns proce,ssortid tillfälligt avbryts från en process eller tråd och överförs till en annan. Vid en context switch sparas den nuvarande processens kontext (till exempel registerinnehåll och instruktionspekare) i minnet och laddar kontextet för nästa process.

När en context switch sker, byter datorn från en process/tråd till en annan. Detta kan ske av olika skäl, till exempel när en process har slutfört sitt arbete eller när en annan process har högre prioritet att utföra. Under en context switch, pausas den aktuella processen och dess nuvarande tillstånd sparas i minnet, medan nästa process laddas in och dess tillstånd återställs så att den kan fortsätta från där den senast avslutade.

Context switchen är en viktig mekanism för multitasking i operativsystem, eftersom den gör det möjligt för flera processer/trådar att dela på en enda processor och använda den effektivt, genom att ge intryck av att flera program körs samtidigt.

19. Vad innebär schemaläggning av en process?
Schemaläggning av en process innebär att bestämma när och i vilken ordning uppgifterna i processen ska utföras. Detta kan inkludera att bestämma vilka resurser som behövs, tidsramar för varje uppgift och hur lång tid varje uppgift tar att slutföra. Syftet med schemaläggning är att optimera processen för att öka effektiviteten och minska tiden det tar att slutföra uppgifterna.

20. När tas beslut om schemaläggning?
Beslut om schemaläggning (scheduling) tas kontinuerligt av operativsystemet för att avgöra vilken process/tråd som ska utföra arbete härnäst. Beslutet tas av en scheduler-komponent som ingår i operativsystemet och som ansvarar för att fördela datorns processortid mellan de olika processer/trådar som kör på systemet.

Valet av process att köra kan ske av flera olika skäl, till exempel prioritet, deadline, I/O-aktiviteter, eller helt enkelt baserat på den process som har väntat längst på att få tillgång till processortiden. Beslut om schemaläggning kan tas när en process/tåd blir tillgänglig för körning, när en process/tåd blockerar på en I/O-operation eller när en process/tåd avbryts på grund av en tidskvantum som har gått ut.

21. Vad gör dispatcher?
En dispatcher är en del av operativsystemet som är ansvarig för att tilldela och hantera systemresurser till de olika processerna som körs på datorn. Dispatchern avgör vilken process som ska ha tillgång till CPU:en och andra resurser vid en given tidpunkt och ser till att processen får tillräckligt med tid för att utföra sina uppgifter. Dispatchern övervakar också processernas status och kan vidta åtgärder om en process av någon anledning inte svarar eller inte beter sig som den ska.

22. Ange tre kriterier som kan användas för att bestämma vilken schemaläggningsalgoritm som skall användas

1. Responstid: Om processernas responstid är en viktig faktor kan man välja en algoritm som ger korta väntetider för interaktiva processer. Exempelvis, Round Robin-algoritmen.

2. Prioritet: Om vissa processer är mer kritiska än andra, kan man använda en prioriteringsalgoritm för att ge högre prioritet till dessa processer. Exempelvis, Priority Scheduling-algoritmen.

3. Rättvis fördelning: Om det är viktigt att fördela systemresurser rättvist mellan alla processer, kan man använda en algoritm som ger varje process en lika stor andel av CPU-tiden. Exempelvis, Fair Share Scheduling-algoritmen.

23. Vad betyder det att en process är I/O bound?
Detta betyder att en process är begränsad av input/output till exempel disk, nätverk eller användarinputs såsom tangentbord.
24. Vad betyder det att en process är CPU bound?
Detta betyder att en process är begränsad av CPU hastigheten eller interna resurser
25. Förklara hur schemaläggning sker om First Come, First Served används
Detta betyder att när processer vill ha CPU tid hamnar de i en kö där den den första som anländer får processortid först så alla processer måste vänta på att det är deras tid i kön. Detta förknippas ofta med en typ av non-*preemptive schemaläggningsmetod.*
26. Förklara hur schemaläggning sker om Round Robin används
Round Robin är en schemaläggning som försöker ge varje process lika mycket CPU tid. Detta gör den genom att ge en förbestämd tid som varje process får köra, sedan får denna processen pausas och läggas sist i kön och vänta på sin tur igen. Detta förknippas ofta med en typ av *preemptive schemaläggningsmetod*
27. Förklara preemptive schemaläggning
Preemptive schemaläggning är en typ av schemaläggning där operativsystemet kan avbryta en process mitt i dess exekvering och ge CPU-tiden till en annan process. Detta kan ske om en högre prioriterad process anländer eller om den nuvarande processen tar för lång tid att slutföra sin uppgift. Preemptive schemaläggning ger en mer rättvis fördelning av CPU-tiden mellan processer och minskar risken för att en enda process monopoliserar resurserna. Det kan dock leda till ökad overhead och minskad effektivitet om processen avbryts för ofta.
28. Förklara non-preemptive schemaläggning
Non-preemptive schemaläggning är en typ av schemaläggning där en process tilldelas CPU-tiden och får använda den tills den slutför sin uppgift eller avbryts av operativsystemet. Andra processer väntar på sin tur tills CPU-tiden blir tillgänglig. Non-preemptive schemaläggning ger enkel och effektiv schemaläggning, eftersom det inte kräver någon övervakning och avbrott av processer. Det kan dock leda till att en process monopoliserar CPU-tiden och orsakar väntetid för andra processer, vilket kan påverka systemets prestanda negativt.
29. Vad är ett quantum?
Ett quantum i sammanhanget med schemaläggning är den maximala tidsperioden som en process kan tilldelas av CPU-tiden innan den eventuellt avbryts av operativsystemet och ersätts av en annan process.
30. Hur påverkar ett quantum schemaläggningen?
Quantum påverkar schemaläggningen genom att bestämma hur länge en process kan få använda CPU-tiden innan den avbryts. En kortare quantum ger mer rättvis fördelning av CPU-tiden mellan processer men kan öka overheadkostnaderna. En längre quantum kan minska overheadkostnaderna men ge vissa processer en orimligt lång väntetid och minska responstiden.
31. Vad är viktigt att tänka på när man bestämmer quantum?
När man bestämmer quantum är det viktigt att ta hänsyn till balansen mellan övergripande prestanda och responstid för enskilda processer. Om quantum är för långt kan det leda till att processer tar för lång tid att slutföra och göra systemet trögt och om quantum är för kort kan

det öka antalet kontextbyten och göra systemet ineffektivt. Valet av quantum kan också påverkas av systemets användningsområde och typ av processer som körs, t.ex. kan interaktiva applikationer kräva kortare quantum för att ge en snabb respons.

32. Förklara begreppen isolation och encapsulation i relation till processer
Isolation betyder att begränsa tillgången till resurser eller data från en process till en annan. Detta uppnås genom att separera processerna i olika skyddade minnesområden och begränsa deras kommunikation med varandra. På så sätt kan en process inte påverka eller störa en annan process, vilket bidrar till en säkrare och mer pålitlig systemdesign.

Encapsulation handlar om att skydda en process eller en del av en process från andra processer. Detta uppnås genom att isolera data och funktioner inom en process eller ett objekt och göra dem otillgängliga för andra processer eller objekt. Genom att begränsa tillgången till interna funktioner och data kan man säkerställa att de inte ändras av misstag och att processen fortsätter att fungera som den ska.

33. Vad är skillnaden på en process och en tråd?
En process är en självständig enhet som innehåller en uppsättning resurser och skapas när ett program eller en applikation körs i datorn. En tråd är en del av en process och delar samma resurser som andra trådar inom samma process. Trådar kan utföra uppgifter parallellt inom samma process, vilket ökar prestandan, medan processer inte kan det.
34. Vad används stacken till i en tråd?
Stacken används i en tråd för att lagra lokala variabler, funktioners argument och återgångsadresser. När en funktion kallas, skapas en ny stackram för att lagra dess parametrar och lokala variabler. När funktionen återgår tas dessa stackramar bort från stacken. Stacken är också viktig för att hålla reda på var tråden befinner sig i dess körning, vilket möjliggör funktioner som rekursion och hantering av undantag.
35. Var är skillnaden på user-level- och kernel-level-trådar?
User-level-trådar hanteras av applikationsprogrammen utan inblandning från operativsystemet och är relativt lätta att skapa och hantera. Kernel-level-trådar hanteras av operativsystemets kärna och är mer resurskrävande men kan utnyttja flera processorkärnor samtidigt för högre prestanda. User-level-trådar är begränsade till en enda processorkärna medan kernel-level-trådar kräver privilegierad tillgång till systemet.
36. Förklara begreppet blockerande anrop
Ett blockerande anrop är en typ av anrop i programmering där programmet stannar upp och väntar på att anropet ska slutföras innan det fortsätter med andra uppgifter. Detta kan orsaka problem när programmet utför flera uppgifter samtidigt eller arbetar med nätverksanslutningar. Ett alternativ till blockerande anrop är icke-blockerande anrop, där programmet fortsätter att köra samtidigt som anropet utförs i bakgrunden.
37. Vad händer om en user-level tråd gör ett blockerande anrop?
Om en user-level tråd gör ett blockerande anrop kommer tråden att blockeras och inte kunna utföra någon annan uppgift tills det blockerande anropet har avslutats. Eftersom user-level trådar hanteras av användarprogram och inte operativsystemet, kan andra trådar i samma process fortsätta att köra medan den blockerade tråden väntar på svar från det blockerande anropet. Detta kan leda till att andra trådar utnyttjas bättre, men det kan också leda till prestandaproblem om många trådar blir blockerade samtidigt och programmet inte hanterar det på ett effektivt sätt.
38. Vad händer om en kernel-level tråd gör ett blockerande anrop?

Om en kernel-level tråd gör ett blockerande anrop kommer hela processen att blockeras tills det blockerande anropet har avslutats. Kernel-level trådar hanteras av operativsystemet och kan inte köras parallellt med andra trådar i samma process, vilket kan leda till sämre prestanda om det finns många blockerande anrop i en process. Men å andra sidan kan kernel-level trådar kommunicera direkt med hårdvaran och har tillgång till systemresurser som inte är tillgängliga för user-level trådar.

39. Varför vill man dela minne mellan trådar?

Man vill dela minne mellan trådar för att undvika att duplicera data och för att möjliggöra kommunikation och samarbete mellan trådar. Genom att dela minne kan trådar dela information och samarbeta på ett effektivt sätt utan att behöva kommunicera genom andra metoder som kan vara mer kostsamma i tid och resurser, såsom meddelandeköer eller låskonstruktioner. Delat minne kan också minska minnesanvändningen genom att undvika att duplicera data i flera trådar, vilket kan förbättra prestanda och minska belastningen på systemet.

40. Vad är skillnaden mellan att dela minne mellan trådar och processer?

Skillnaden mellan att dela minne mellan trådar och processer är att trådar delar samma minnesutrymme inom en process, medan processer har separata minnesutrymmen som inte delas. Det innebär att trådar inom samma process kan dela data direkt genom att använda gemensamma variabler och datastrukturer, medan processer måste använda andra metoder för att kommunicera och dela data, såsom delade minnessegment eller interprocesskommunikation. Eftersom trådar inom samma process delar samma minnesutrymme kan de också kommunicera och samarbeta mer effektivt utan överheaden för att överföra data mellan processer, men det innebär också att det finns en större risk för datakonflikter och fel om trådarna inte synkroniseras och hanteras på rätt sätt.

41. Förklara begreppet sequentially consistent i relation till minne

Sequentially consistent är en modell för att beskriva hur minnesoperationer i ett parallellt system ska ordnas. En minnesmodell som är sequentially consistent garanterar att alla minnesoperationer utförs i en viss ordning som liknar den som skulle ha inträffat om operationerna hade utförts seriellt. Detta innebär att om en minnesoperation utförs av en tråd och en annan tråd senare läser samma minnesplats, så kommer den senare läsningen att reflektera resultatet av den tidigare operationen. Med andra ord garanterar sequentially consistent att resultatet av en sekvens av minnesoperationer i en parallell miljö uppträder på ett förutsägbart och logiskt sätt, vilket gör det enklare att programmera parallella system och undvika felaktigt beteende.

42. Vad innebär ett data race/race condition?

Ett data race (race condition) är en bugg i datorprogrammering när flera trådar/processer försöker åtkomma och ändra samma del av minnet samtidigt utan synkronisering. Det kan leda till felaktiga resultat. Synkroniseringsmekanismer, såsom lås eller atomära operationer, används för att undvika data races.

43. Vad är en kritisk sektion?

En kritisk sektion är en del av programkoden som innehåller gemensamma resurser som delas mellan flera trådar eller processer och där det krävs synkronisering för att undvika att flera trådar samtidigt manipulerar eller läser samma resurs.

44. Ge exempel på ett data race

Om två trådar samtidigt adderar till samma variabel kan detta ske. Eftersom det inte är synkade kan variabeln bli större än förväntat. Att addera till en variabel innebär att

processen måste läsa och skriva till variabeln vilket gör att båda trådarna kanske läser variabeln samtidigt och adderar till en “gammal” variabel. Detta leder till oväntade resultat.

,

45. Skriv ett program i Java som kan ge upphov till ett data race

```
public class TestPara extends Thread {
   static int race = 0;
   public void run() {
      for(int i = 0; i < 100; i++){
         race += 1;
      }
   }

   public static void main(String[] args) {
      TestPara thread1 = new TestPara();
      TestPara thread2 = new TestPara();
      thread1.start();
      thread2.start();
      try {
         thread1.join();
         thread2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(race);
   }
}
```

46. Vad innebär ömesidig uteslutning?

Ömsesidig uteslutning är en teknik för att undvika konflikter som uppstår när flera trådar eller processer försöker åtkomma eller manipulera samma resurs samtidigt. Tekniken innebär att en tråd som har åtkomst till den gemensamma resursen får blockera andra trådar från att manipulera den tills den första tråden släpper resursen. Detta garanterar att bara en tråd åt gången kan åtkomma den gemensamma resursen, vilket undviker datakonflikter och felaktigt beteende.

47. Hur kan ömesidig uteslutning lösas? ge exempel.

Ömsesidig uteslutning kan lösas med hjälp av synkroniseringsmekanismer som lås, semaforer och monitorer. Dessa mekanismer används för att säkerställa att endast en tråd åt gången kan åtkomma en kritisk sektion i koden. Ett exempel på hur ömsesidig uteslutning kan lösas är att använda ett binärt lås som tillåter bara en tråd att åtkomma en resurs åt gången. När en tråd har låst resursen genom att sätta låset, måste alla andra trådar vänta tills låset släpps innan de kan manipulera resursen. På så sätt säkerställer man att bara en tråd åt gången kan manipulera resursen.

48. Förklara hur strict alteration försöker lösa ömesidig uteslutning. Vilka problem finns?
Strict alteration är en metod som försöker lösa ömsesidig uteslutning genom att varje part tar initiativ till förändring i tur och ordning, där en part gör en förändring och sedan väntar på att den andra parten ska svara med en liknande förändring. Detta syftar till att skapa en balanserad interaktion mellan parterna och undvika en situation där en part alltid tar initiativ till förändringar och den andra parten alltid reagerar.

Ett problem med strict alteration är att det kan kräva mycket tid och tålamod för att uppnå en balanserad interaktion, vilket kan leda till frustration och brist på engagemang från båda parterna. Dessutom är det inte alltid lätt att avgöra vad som utgör en liknande förändring, vilket kan leda till missförstånd och konflikt. Det kan också finnas situationer där en part inte vill eller kan göra en förändring, vilket kan leda till att strict alteration inte fungerar som en lösning på ömsesidig uteslutning.

49. Förklara Peterson's algoritm för ömsesidig uteslutning
Petersons algoritm är en klassisk algoritm för ömsesidig uteslutning (mutual exclusion) i datorprogrammering. Algoritmen fungerar för två processer/trådar och använder två flags och en turvariabel (turn), vilket möjliggör för en process att vänta tills den andra processen har slutfört sitt arbete innan den tar över resursen.

Algoritmen fungerar enligt följande:

1. Varje process sätter sin bollplanka till sann (true) för att indikera att den vill ta resursen.
2. Processerna växlar på att kontrollera turvariabeln och väntar tills det blir sin tur att utföra arbete.
3. När det blir en processes tur, sätter den turvariabeln till sin egen processid och kontrollerar om den andra processen vill ta resursen genom att kolla dess flagga. Om den andra processen vill ta resursen, väntar den här processen tills den andra processen har slutfört sitt arbete.
4. Om ingen annan process försöker ta resursen, kan processen utföra sitt arbete och när den är klar, sätter den sin flagga till false och ger tur till den andra processen.

Genom att använda flags och turvariabeln kan Petersons algoritm uppnå ömsesidig uteslutning så att bara en process åt gången har tillgång till den delade resursen.

50. Vad innebär progression i relation till ömsesidig uteslutning?
Progression i relation till ömsesidig uteslutning innebär att två eller flera parter rör sig mot olika riktningar eller utvecklar olika åsikter eller handlingar som gör det svårt eller omöjligt att samarbeta eller kommunicera effektivt. Det kan leda till att samarbetet avbryts eller att parterna väljer att inte interagera med varandra längre, vilket är en form av ömsesidig uteslutning.
51. Vad innebär begränsad väntan i relation till ömsesidig uteslutning?
Begränsad väntan i relation till ömsesidig uteslutning innebär att en part ger den andra parten en bestämd tidsperiod att svara på ett erbjudande eller förslag. Om den andra parten inte svarar inom den angivna tidsramen, så tolkas det som ett negativt svar eller brist på intresse. Det kan sedan leda till ömsesidig uteslutning, där parterna väljer att inte interagera med varandra längre eller att samarbetet avbryts.

52. Implementera Peterson's algorithm i Java- eller Python-liknande kod.

```
interested = [False, False]
turn = 0

def enter(process):
    global turn, interested
    interested[process] = True
    other = 1 - process
    turn = other

    while interested[other] and turn == process:
        continue

def leave(process):
    global interested
    interested[process] = False
```

53. Vad är en semafor?

Inom datorer och programmering refererar semafor vanligtvis till en teknik som används för att hantera samtidig åtkomst till delade resurser, såsom minneslokaler, filer eller databaser. En semafor är en mekanism som används för att koordinera och synkronisera tillgång till dessa resurser för att undvika konflikter eller datorkrascher. Det fungerar genom att låsa en resurs när en process behöver den och sedan frigöra den när processen är klar med den. Detta säkerställer att endast en process kan åtkomma en resurs åt gången, vilket minskar risken för konflikter och förhindrar att data förloras.

54. Vad är ett mutex lock?

Ett mutex lock (förkortning av mutual exclusion lock) är en typ av semafor som används för att undvika att två eller flera processer eller trådar samtidigt försöker komma åt eller modifiera en delad resurs, såsom en minneslokal eller en fil.

En mutex lock fungerar genom att tillfälligt blockera åtkomst till en resurs när en process eller tråd försöker låsa upp den, och sedan låser upp den när den är klar med att använda den. När en process eller tråd har låst en resurs med mutex lock kan ingen annan process eller tråd komma åt resursen tills den har frigjorts. Detta säkerställer att endast en process eller tråd kan modifiera resursen åt gången, vilket minskar risken för datakonflikter och fel.

55. Vad är skillnaden mellan en binär och en räknande semafor?

En binär semafor kan endast anta två tillstånd: låst eller upplåst. Det innebär att en binär semafor endast kan användas för att kontrollera åtkomst till en enda resurs. När semaforen är låst kan ingen annan process eller tråd komma åt resursen tills den har frigjorts.

En räknande semafor, å andra sidan, kan hantera flera åtkomster till en resurs. Semaforen innehåller en räknare som ökas när en process eller tråd använder resursen och minskas när processen eller tråden frigör resursen. Om räknaren är större än noll, kan flera processer eller trådar komma åt resursen samtidigt. En räknande semafor kan användas för att kontrollera åtkomst till en resurs som har ett begränsat antal instanser eller som tillåter flera parallella åtkomster.

Således kan man säga att skillnaden mellan en binär och en räknande semafor är att en binär semafor bara kan ha två tillstånd medan en räknande semafor kan ha flera tillstånd beroende på räknaren.

56. Implementera en producent i Java- eller Python-liknande kod som använder semaforer för ömesidig uteslutning

```
import threading
import time

MAX_BUFFER_SIZE = 10

mutex = threading.Semaphore(1)
empty = threading.Semaphore(MAX_BUFFER_SIZE)
full = threading.Semaphore(0)

buffer = []

def producer():
    global buffer
    for i in range(10):
        new_item = time.time()
        empty.acquire()
        mutex.acquire()
        buffer.append(new_item)
        print(f"Producer produced {new_item}. Buffer size is {len(buffer)}.")
        mutex.release()
        full.release()

producer_thread = threading.Thread(target=producer)
producer_thread.start()

producer_thread.join()
```

57. Visa hur en räknande semafor kan implementeras med hjälp av binära semaforer

Vi börjar med skapa en integer som räknande variabel och en binär semafor. Varje gång man ökar eller minskar räknare måste man låsa variabeln. Man måste också checka att räknaren inte gå under 0.

58. Ge ett exempel i Java där två trådar delar en variabel och använder en semafor för att (korrekt) skydda tillgången

```
public class TestThread implements Thread{
   static int deladVaribel = 0;
   Semaphore mutexLock = new Semaphore(1);

   public void run() {
      mutexLock.aquire();
      deladVaribel += 1;
      mutexLock.release();

   }

   public static void main(String[] args) {
      TestPara thread1 = new TestPara();
      TestPara thread2 = new TestPara();
      thread1.start();
      thread2.start();
      try {
         thread1.join();
         thread2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(deladVariabel);
   }
}
```

59. Vilka risker finns det med låsning?

Det finns flera risker med låsning, särskilt om det används felaktigt eller överanvänds:

1. Deadlocks: En deadlock uppstår när två eller fler processer eller trådar blir blockerade på varandra, eftersom varje process eller tråd väntar på att den andra ska släppa en resurs som den behöver för att fortsätta. Detta kan leda till att hela systemet fryser eller kraschar.

2. Långa väntetider: Om en process eller tråd väntar på att en låst resurs ska frigöras, kan det orsaka onödiga fördröjningar och långa väntetider, vilket kan påverka systemets prestanda och effektivitet.

3. Resursbrist: Om en process eller tråd behåller en låsning på en resurs för länge kan det orsaka resursbrist för andra processer eller trådar som behöver samma resurs, vilket kan orsaka onödiga fördröjningar och långa väntetider.

Därför är det viktigt att använda låsning med försiktighet och förstå dess begränsningar och risker för att undvika potentiella problem och störningar i systemet.

60. Vad är ett deadlock?
Deadlock är en situation där två eller flera processer (eller trådar) blir blockerade på varandra, eftersom varje process eller tråd väntar på att den andra ska släppa en resurs som den behöver för att fortsätta. Ingen av processerna kan avslutas utan att släppa resursen, men eftersom de väntar på varandra, är de inte heller kapabla att göra det. Detta leder till att hela systemet fryser eller kraschar.

Ett exempel på ett dödläge kan vara två processer som behöver tillgång till två olika resurser, men varje process har redan tagit en resurs och väntar på att få tillgång till den andra resursen. Båda processerna är nu blockerade och kan inte fortsätta, och systemet är i ett dödläge.

För att undvika deadlocks, behöver man använda tekniker som förebygger eller hanterar dödlägen, såsom att använda sig av en hierarki av resurslåsning, att undvika överlappande låsning, eller att använda en timeout mekanism för att hantera låsning.

62. Ge ett exempel på ett program i Java eller Python som kan leda till deadlock

```
import threading
import time

lock1 = threading.Lock()
lock2 = threading.Lock()

def function1():
    lock1.acquire()
    time.sleep(100)
    lock2.acquire()
    print("Function 1 acquired both locks.")
    lock2.release()
    lock1.release()

def function2():
    lock2.acquire()
    time.sleep(100)
    lock1.acquire()
    print("Function 2 acquired both locks.")
    lock1.release()
    lock2.release()

t1 = threading.Thread(target=function1)
t2 = threading.Thread(target=function2)

t1.start()
t2.start()
```

t1.join()
t2.join()

63. Vilka fyra villkor krävs för deadlock?

1. Mutual Exclusion: Minst en resurs måste vara i exklusivt bruk av en process, vilket innebär att andra processer inte kan använda resursen samtidigt.

2. Hold and Wait: Processen håller minst en resurs och väntar på att få tillgång till ytterligare en resurs som är upptagen av en annan process.

3. No Preemption: Resurser kan inte tas ifrån en process som redan har tilldelats dem, utan den måste frivilligt frigöra resursen efter att ha använt den.

4. Circular Wait: Det måste finnas en cyklisk beroende av resurser, där varje process i cykeln väntar på en resurs som hålls av nästa process i cykeln.

64. Förklara hold and wait

"Hold and Wait" är en av de fyra villkor som krävs för att en deadlock ska uppstå. Det innebär att en process håller minst en resurs och samtidigt väntar på att få tillgång till ytterligare en resurs som är upptagen av en annan process.

65. Förklara no preemption

"No Preemption" är en av de fyra villkor som krävs för att en deadlock ska uppstå. Det innebär att en process som har tilldelats en resurs och kan inte tvingas att frigöra den förrän den själv är klar med att använda den. Detta betyder att om en process redan har fått tillgång till en resurs, så kan inte den resursen tas ifrån den och tilldelas till en annan process utan processens frivilliga samtycke.

66. Förklara circular wait

"Circular Wait" är en av de fyra villkor som krävs för att en deadlock ska uppstå. Det innebär att det finns en kedja av processer som håller resurser som behövs av nästa process i kedjan. Med andra ord, processen 1 håller en resurs som processen 2 behöver, processen 2 håller en resurs som processen 3 behöver, och så vidare, tills den sista processen i kedjan håller en resurs som behövs av den första processen.

67. Förklara hur deadlock kan förhindras

Deadlock kan förhindras genom att undvika/bryta en av de fyra villkoren som krävs för att en deadlock ska uppstå. De fyra villkoren är:
Mutual exclusion
No preemption
Hold and wait
Circular wait

68. Hur kan mutual exclusion förhindras för att undvika deadlock?

Ett sätt att undvika mutual exclusion är att tillåta flera processer att dela på en resurs. Till exempel kan flera processer dela på en fil genom att ha läs- eller skrivrättigheter till olika delar av filen.

69. Hur kan hold and wait förhindras för att undvika deadlock?

Ett sätt att undvika hold and wait är att tvinga processer att släppa alla resurser som de har innan de försöker få tillgång till nya resurser. Detta kan uppnås genom att kräva att

processerna släpper alla sina resurser när de försöker få tillgång till nya, eller genom att kräva att processerna ska frivilligt släppa resurserna när de inte längre behöver dem.

70. Hur kan no preemption förhindras för att undvika deadlock?

Ett sätt att undvika no preemption är att tillåta att en process tar resurser från en annan process när det behövs. Detta kan uppnås genom att processer tvingas släppa resurserna de har innan de kan begära nya, eller genom att processer frivilligt släpper resurserna när de inte längre behöver dem.

71. Hur kan circular wait förhindras för att undvika deadlock?

Ett sätt att undvika circular wait är att kräva att processerna ska begära resurser i en viss ordning. Detta kan bryta kedjan av resurser och minska risken för att dödläge uppstår.

72. Ge exempel på hur en semafor kan användas så att hold and wait inte gäller

För att undvika hold and wait-problemet kan en semafor användas på ett lämpligt sätt. En semafor kan användas på ett sätt som kallas "no-hold-and-wait". Detta innebär att en process som inte kan få tillgång till en resurs omedelbart inte kommer att hålla några resurser och därmed inte blockerar andra processer.

73. Varför kan det vara problematiskt att förhindra no preemption

Det kan vara problematiskt att förhindra no preemption i vissa situationer av flera skäl:

Resursallokering: Om en aktivitet inte kan avbrytas, kan det vara svårt att omallokera resurser som den använder sig av, vilket kan hindra andra aktiviteter från att få tillgång till dessa resurser.

Prioritering: Ibland är det nödvändigt att prioritera vissa aktiviteter över andra. Om en aktivitet inte kan avbrytas kan det bli svårt att ge företräde åt viktigare aktiviteter och processer.

Säkerhet: Ibland kan det vara nödvändigt att avbryta en aktivitet eller process av säkerhetsskäl, t.ex. om den innebär en risk för skada eller olycka. Om en aktivitet inte kan avbrytas kan detta utgöra en fara för människor eller egendom.

Effektivitet: I vissa fall kan det vara mer effektivt att tillåta avbrytning av en aktivitet eller process för att optimera användningen av resurser och tid.

74. Varför kan det vara problematiskt att förhindra hold and wait

"Hold and wait" betyder att en process håller en resurs medan den väntar på att få tillgång till en annan resurs som behövs för att slutföra uppgiften. Det kan vara problematiskt att förhindra hold and wait eftersom det kan leda till resursbrist, säkerhetsproblem, ineffektivitet och ökad komplexitet.

75. Förklara starvation. Varför är det ett problem?

Starvation är ett tillstånd där en process inte kan fortsätta att arbeta eller få tillgång till de resurser som den behöver på grund av brist på resurser. Det kan leda till förseningar, störningar och förlust av data eller korrupta resultat. För att undvika starvation är det viktigt att tilldela tillräckliga resurser, prioritera processer och aktiviteter och frigöra resurser efter användning.

76. Vad är en resource allocation graph och hur används den i samband med deadlock?
RAG (resource allocation graph) är ett verktyg som används för att visualisera resursanvändning och för att upptäcka potentiella deadlocks. Genom att använda RAG kan man se till att ingen process håller en resurs och väntar på en annan resurs samtidigt som en annan process håller den andra resursen som den första processen behöver.

77. Hur kan man avgöra om ett system är i deadlock med hjälp av en resource allocation graph?
För att avgöra om systemet är i deadlock med hjälp av resursallokeringsgrafen måste man identifiera om det finns en cirkel i grafen som innehåller endast allokerade resurser och inga tillgängliga resurser. Om en sådan cirkel finns i grafen indikerar detta att processerna i systemet inte kan fortsätta att arbeta eftersom de väntar på resurser som hålls av andra processer i cirkeln.

78. Är det ok att ignorera deadlock?
No, it is not okay to ignore deadlocks. Deadlocks can cause significant problems in a computer system, including causing programs to freeze or crash, reducing system performance, and potentially leading to data loss or corruption.

79. Vad är skillnaden på att förhindra och undvika deadlock?
Att förhindra deadlock innebär att man tar bort något av de fyra nödvändiga villkoren för att en deadlock ska kunna uppstå. Det kan till exempel göras genom att använda tekniker som banker's algorithm eller resource ordering, där man ser till att processer aldrig blockeras på resurser som inte är tillgängliga. Genom att eliminera möjligheten för en deadlock att uppstå kan man säkerställa systemets korrekta funktion.

Att undvika deadlock innebär att man tillåter att en deadlock kan uppstå, men försöker undvika att det faktiskt händer genom att dynamiskt hantera systemets resursallokering. Ett exempel på en teknik som används för att undvika deadlock är resursallokering med banker's algorithm, där man alltid försöker säkerställa att processer har tillräckligt med resurser tillgängliga för att undvika att de blockeras. Genom att hantera systemets resursallokering på ett smart sätt kan man undvika att en deadlock faktiskt uppstår.

80. Förklara Banker’s algoritm
Banker's algorithm fungerar genom att förutsäga om en begärd resurs kommer att orsaka en deadlock i systemet. Algoritmen använder sig av två typer av data för att göra denna prognos: det maximala antalet resurser som är tillgängliga för varje typ av resurs och det nuvarande antalet resurser som används av varje process.

För att använda Banker's algorithm för att bestämma om en begärd resurs kan tilldelas eller inte, kontrollerar algoritmen om det finns tillräckligt med lediga resurser för att uppfylla processens begäran. Om det finns tillräckligt med lediga resurser för att uppfylla begäran tilldelas resursen till processen, annars måste processen vänta tills resurserna blir tillgängliga.

En av de stora fördelarna med Banker's algorithm är att den kan användas för att förhindra deadlocks i realtid. Algoritmen är också ganska enkel att implementera och kan användas i en mängd olika system, inklusive operativsystem, nätverk och databassystem. En nackdel med Banker's algorithm är dock att det kan vara svårt att bestämma det maximala antalet resurser som krävs för varje process, vilket kan göra algoritmen mindre effektiv i vissa situationer.

81. Hur kan ett system återhämta sig från deadlock?
Ett system kan återhämta sig från en deadlock på flera sätt, beroende på hur det är utformat. Här är några möjliga sätt:

1. Processen som orsakade deadlocked kan avbrytas eller avslutas, vilket frigör resurserna och tillåter de andra processerna att fortsätta.

2. En eller flera resurser kan tas ifrån en process i deadlocked och tilldelas en annan process, vilket kan bryta deadlocked och tillåta de andra processerna att fortsätta.

3. Alla processer som är inblandade i deadlocked kan avbrytas eller återställas till ett tidigare tillstånd, vilket frigör resurserna och tillåter de andra processerna att fortsätta.

82. Förklara prefix sum
Prefix sum är en teknik för att beräkna en sekvens av numeriska värden där varje element i sekvensen är summan av alla element före det i sekvensen, inklusive det själva elementet.

För att beräkna prefixsumman för en sekvens kan man använda en algoritm som går igenom varje element i sekvensen och för varje element addera det till summan av de föregående elementen. Genom att spara summan i en ny sekvens kan man sedan använda den nya sekvensen för att beräkna prefixsumman för en annan sekvens genom att enkelt hämta summan för det element som behövs.

En fördel med prefix sum är att det kan användas för att parallellisera beräkningar eftersom det är enkelt att dela upp sekvensen i mindre delar och beräkna prefixsumman för varje del parallellt. Detta kan leda till ökad prestanda och minskad beräkningstid.

83. Implementera en parallel prefix sum med trådar i Java eller Python

84. Förklara prefix scan
Prefix scan är en algoritm som används för att beräkna en sekventiell summa över en mängd element. Den grundläggande idén bakom prefix scan är att för varje element i en ingående mängd, summera alla element som kom före det, inklusive det själv. Resultatet av varje beräkning lagras i en utgående mängd, vilket skapar en prefixsumma för den ursprungliga mängden.

För att genomföra en prefix scan på en mängd element börjar man med att beräkna en lokalt prefixsumma för varje processor i parallell. Därefter kombineras dessa lokala prefixsummor för att beräkna en global prefixsumma för hela mängden. Detta kan upprepas för en sekventiell summa över flera iterationer.

Prefix scan används ofta inom parallella algoritmer, till exempel parallella sorteringsalgoritmer eller parallella trädoperationer. Det kan också användas inom grafik, signalbehandling och annan datorteknik.

85. Implementera en parallel prefix scan med trådar i Java eller Python

```
import threading

def sum(list, index):
    start = len(list) // 4 * index
    end = len(list) // 4 * (index + 1)
    for i in range(start + 1, end, 1):
        list[i] = list[i] + list[i - 1]
    endsOfRanges[index] = list[end - 1]

def sumWithAdd(list, index):
    start = len(list) // 4 * index
    end = len(list) // 4 * (index + 1)
    for i in range(start, end, 1):
        list[i] += endsOfRangesExclusive[index]

list = [4,5,2,6,8,4,-2,9]
threads = []
endsOfRanges = [None] * 4
for i in range(4):
    newThread = threading.Thread(target=sum(list, i))
    threads.append(newThread)
    newThread.start()

for i in threads:
    i.join()

endsOfRangesExclusive = [0, None, None, None]

for i in range(3):
   endsOfRangesExclusive[i+1] = endsOfRanges[i] + endsOfRangesExclusive[i]

for i in range(4):
    newThread = threading.Thread(target=sumWithAdd(list, i))
    threads[i] = newThread
    newThread.start()

for i in threads:
    i.join()
```

```
print(list)
```

86. Förklara odd-even transposition sort

Odd-even transposition sort är en parallell sorteringsalgoritm som kan användas för att sortera data i en lista eller en array. Algoritmen är en form av jämförelsebaserad sortering och bygger på jämförelser mellan element i listan.

Algoritmen består av två faser, en jämförelsefas och en utbytesfas. Under jämförelsefasen jämförs och utbyts element parvis med jämnt eller udda index, beroende på vilken fas som genomförs. I utbytesfasen fortsätter samma process, men nu jämförs och utbyts element med udda eller jämnt index. Dessa faser upprepas tills listan är helt sorterad.

Odd-even transposition sort är en paralleliserad version av bubble sort, där varje process arbetar med en del av datan parallellt. Genom att använda parallellism kan sorteringen utföras mycket snabbare än om den skulle utföras sekventiellt.

87. Implementera en parallel odd-even transposition sort med trådar i Java eller Python

```
import threading

class ParallelSorter:
    def __init__(self, lst):
        self.lst = lst
        self.n = len(lst)  # antal element i listan
        self.num_threads = threading.active_count()  # antal trådar som körs just nu
        self.exchange_occurred = True
        self.offset = 0

    def _sort(self, thread_id: int) -> None:
        while self.exchange_occurred:
            local_exchange_occurred = False
            start_index, end_index = self.n // self.num_threads * thread_id, (self.n //
self.num_threads * (thread_id + 1)) + 1
            if thread_id == self.num_threads - 1:
                end_index = self.n

            for i in range(start_index + 1 + self.offset, end_index, 2):
                if self.lst[i - 1] > self.lst[i]:
                    self.lst[i - 1], self.lst[i] = self.lst[i], self.lst[i - 1]
                    local_exchange_occurred = True

            threading.Barrier(self.num_threads).wait()
            self.exchange_occurred = False

            threading.Barrier(self.num_threads).wait()
```

```
            if local_exchange_occurred:
                self.exchange_occurred = True

            threading.Barrier(self.num_threads).wait()
            if thread_id == 0:
                self.offset = (self.offset + 1) % 2

    def sort(self) -> None:
        threads = []
        for i in range(self.num_threads):
            threads.append(threading.Thread(target=self._sort, args=(i,)))
            threads[-1].start()

        for thread in threads:
            thread.join()
```

88. Varför är det en dålig idé att skapa nya trådar för varje rekursivt anrop i en divide and conqueralgoritm?
Att skapa nya trådar för varje rekursivt anrop i en divide and conquer-algoritm kan leda till en stor mängd trådar som konkurrerar om systemresurser och kan orsaka överbelastning eller trådsvält. Varje tråd kräver också en viss mängd overhead, vilket kan påverka prestandan och öka sannolikheten för trådfel. Dessutom kan det vara svårt att hantera kommunikationen mellan de olika trådarna och att samordna deras resultat på ett effektivt sätt.

89. Varför kan det vara svårt att parallelisera rekursiva algoritmer?
Beroenden mellan delproblem: Rekursiva algoritmer består ofta av att dela upp problemet i mindre delproblem som sedan löses rekursivt. Dessa delproblem kan ha beroenden mellan varandra, vilket kan göra det svårt att parallellisera lösningen.

Kostnaden för att skapa nya trådar: Om en rekursiv algoritm skapar en ny tråd för varje rekursivt anrop kan kostnaden för att skapa och synkronisera trådar bli hög. Detta kan leda till att algoritmen blir långsammare när den körs parallellt.

Svårt att dela upp problemet: Det kan vara svårt att dela upp ett rekursivt problem i mindre delproblem som kan lösa parallellt. Ofta är det inte uppenbart hur man ska göra detta utan att introducera onödig överlappning eller kommunikation mellan trådar.

90. Visa hur en for-loop kan paralleliseras i Java eller Python med trådar
lol
91. Vad är en barrier
En barrier är en synkroniseringsmekanism som används i parallella beräkningar för att säkerställa att alla trådar i en grupp av trådar har nått en viss punkt i koden innan de fortsätter vidare. Barriärer används för att undvika problem som kan uppstå när trådar kör förbi varandra och förlorar synkroniseringen, vilket kan leda till felaktigt resultat. När en tråd når en barrier väntar den där tills alla andra trådar har nått samma punkt, och därefter går de tillsammans vidare. På så sätt säkerställer man att alla trådar har uppdaterat sina värden eller tillstånd på rätt sätt och att ingen tråd kommer att köra på felaktiga eller föråldrade värden.

92. Hur kan en semafor användas för att signalera mellan trådar?
En semafor kan användas för att signalera mellan trådar genom att en tråd väntar på att semaforen ska släppas och en annan tråd signalerar till semaforen när det är dags att fortsätta. Semaforen fungerar som en räknare som håller koll på antalet tillgängliga resurser. När en tråd vill använda en resurs minskar den semaforen med 1. Om semaforen redan är 0 så kommer tråden att blockeras tills semaforen blir större än 0 igen. När tråden är klar med resursen ökar den semaforen med 1. Om det finns andra trådar som väntar på resursen kommer en av dem att få tillgång till den. På så sätt kan semaforer användas för att synkronisera trådar och signalera när en resurs är tillgänglig.

93. Hur kan en barrier impementeras med semaforer. Visa i Java- eller Python-liknande kod

```
from threading import Semaphore

class Barrier:
    def __init__(self, n):
        self.n = n  # Antal trådar som behöver vänta på barriären
        self.count = 0  # Antal trådar som har nått barriären
        self.mutex = Semaphore(1)  # Mutual exclusion semaphore
        self.barrier_sem = Semaphore(0)  # Semafor för att vänta på barriären

    def wait(self):
        # Kritisk sektion
        self.mutex.acquire()
        self.count += 1
        if self.count == self.n:
            self.barrier_sem.release()  # Signalera att alla trådar har nått barriären
        self.mutex.release()

        self.barrier_sem.acquire()  # Vänta på signalen att alla trådar har nått barriären
        self.barrier_sem.release()  # Släpp igenom tråden

    def reset(self):
        self.count = 0
```

94. Förklara hur depth-first search kan paralleliseras med trådar
Depth-first search kan paralleliseras med trådar genom att varje tråd söker efter en del av grafen i djupet parallellt med andra trådar. Varje tråd väljer en del av grafen och utför sökningen som vanligt, men när den når en nod som har flera outgående kanter skapar den nya trådar som kan utforska de olika grenarna parallellt. När alla trådar har avslutat sökningen sammanfogas resultaten till en sammanhängande lösning. Genom att använda trådar kan sökningen utföras parallellt och därmed minska tiden för att hitta en lösning.

95. Förklara hur breadth-first search kan paralleliseras med trådar
Breadth-first search kan paralleliseras med trådar genom att dela upp grafen i lager och utforska varje lager parallellt. Varje tråd börjar med att utforska en nod från det första lagret och sedan går vidare till nästa lager, där varje tråd utforskar en del av noderna parallellt. När en nod besöks av en tråd kan det skapas nya trådar för att utforska dess obesökta grannar parallellt. När alla trådar har avslutat sökningen sammanfogas resultaten till en

sammanhängande lösning. Genom att använda trådar kan sökningen utföras parallellt och därmed minska tiden för att hitta en lösning.

96. Förklara hur Prim's algoritm kan paralleliseras med trådar

Prim's algoritm kan paralleliseras med trådar genom att dela upp mängden av oändligt många noder i mindre delmängder och utföra sökningen parallellt på varje delmängd. Varje tråd börjar med att välja en nod från den aktuella delmängden och sedan välja en närliggande nod som har den minsta kostnaden att ansluta till den aktuella delmängden. När varje tråd har valt den billigaste anslutande noden kan det skapas nya trådar för att utforska dess obesökta grannar parallellt. När alla trådar har avslutat sökningen sammanfogas resultaten till en sammanhängande lösning. Genom att använda trådar kan Prim's algoritm utföras parallellt och därmed minska tiden för att hitta en minsta spännande träd.

97. Ge en parallell algoritm för att hitta den minsta värdet i en lista i Java eller Python-liknande kod

```
import threading

def parallel_min(lst, num_threads):
    min_values = [None] * num_threads

    def find_min(tid):
        start = tid * (len(lst) // num_threads)
        end = start + (len(lst) // num_threads)

        min_val = min(lst[start:end])

        min_values[tid] = min_val

    threads = []
    for i in range(num_threads):
        thread = threading.Thread(target=find_min, args=(i,))
        threads.append(thread)

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    return min(min_values)
```

98. Vi kan hitta det minsta värdet i en lista på linjär tid ($O(N)$). Hur lång tid tar det att köra med P procecssorer? Motivera.

Tiden för att hitta det minsta värdet i en lista med P processorer blir O(N/P), eftersom varje processor kan söka igenom en del av listan parallellt. Eftersom P processorer delar upp listan i P delar, söker varje processor igenom en del av storleken N/P, vilket ger en total tidskomplexitet på O(N/P).

99. Vad krävs för att vi skall erhålla en speedup på P med en algorim som körs på P procecssorer

lol

100. Är det alltid snabbare att parallelisera en algoritm och köra den på så många processorer som möjligt? Motivera.

Nej, det är inte alltid snabbare att parallelisera en algoritm och köra den på så många processorer som möjligt. Det beror på flera faktorer, inklusive:

1. Algoritmens egenskaper: Vissa algoritmer är lättare att parallelisera än andra. Om en algoritm har mycket databeroenden eller om dess beräkningar är svåra att dela upp i mindre delar, kan det vara svårt att hitta en effektiv parallelisering.
2. Storlek på problemet: Om problemet är tillräckligt litet, kan det hända att overheaden för att skicka data och koordinera arbetet mellan processorer är större än själva beräkningstiden. I sådana fall kan en seriell algoritm faktiskt vara snabbare än en parallell.
3. Tillgängliga resurser: Om det finns begränsningar i antalet tillgängliga processorer eller annan hårdvara kan det vara svårt att dra nytta av parallelliseringen på ett effektivt sätt.
4. Kommunikationskostnader: Om parallelliseringslösningen kräver att processorer behöver skicka mycket data till varandra eller att det finns mycket synkronisering mellan processorer, kan kommunikationskostnaderna göra det mindre effektivt än en seriell lösning.
5. Programmering och underhållskostnader: Parallelisering kan kräva mer komplicerad kod och mer resurskrävande underhåll än en seriell lösning.

Därför måste man göra en noggrann bedömning av de ovannämnda faktorerna innan man beslutar om en parallelliseringslösning, och ibland kan en seriell lösning vara den mest effektiva.

101. Förklara N-ary-sökning.

N-ary sökning fungerar som en Binär sökning men istället för att dela upp i 2 stycken barn så delas sökningen upp i N-stycken barn. Den används för att hitta ett visst index till ett värde i en sorterad lista.

102. Förklara hur N-ary-sökning kan paralleliseras med trådar

N-ary kan paralleliseras genom att dela upp varje tråd på en del av listan. Därefter kollar varje nod om det sökta värdet är större eller mindre än det kollade värdet och hittar en brytpunkt där det sökta värdet ligger emellan 2 stycken trådar, eller så hittar den det sökta värdet. Därefter kollar den i värdena emellan dessa trådar och delar upp den i en mindre lista för att fortsätta samma process tills dom hittar det sökta värdet.

103. Förklara lock free

Lock-free-tekniken bygger på algoritmer och strukturer som gör det möjligt för flera processer att arbeta med samma data samtidigt, utan att orsaka konflikter eller deadlocks. Istället för att använda lås, använder lock-free teknik andra mekanismer, som "atomiska operationer" (där en operation på en variabel antingen utförs helt eller inte alls, och inte kan avbrytas mitt i processen).

104. Förklara wait free
Wait-free är när varje enskild tråd eller process som använder algoritmen garanteras att avsluta sitt arbete inom en begränsad tidsperiod, oavsett vad andra trådar eller processer gör eller hur de agerar. Med andra ord behöver inte någon tråd eller process vänta på en annan tråd eller process för att slutföra sitt arbete.

105. Vad menas med en optimistisk algoritm (med avseende på ömesidig uteslutning)?
Optimistisk algoritm är när varje tråd eller process som använder algoritmen antar att det inte finns några konflikter eller överlappningar i dataåtkomsten och börjar utföra sin kritiska sektion direkt utan att först försöka låsa resurserna. Om två eller flera trådar eller processer försöker utföra sin kritiska sektion samtidigt och det uppstår en konflikt, så kommer algoritmen att upptäcka detta och försöka lösa konflikten på något sätt. Wait-free och Lock-free är sådana algoritmer.

106. Förklara compare and set
Compare and set är ett sätt att få flera processer att använda sig av samma kritiska region utan att behöva låsa. Detta görs genom att processerna jämför (compare) ett värde för att se om alla processer har samma värde. Om det inte har samma värde så avbryts operationen. Om värdet är densamma så sätter det den gemansamma variabeln till det nya värdet (set). Detta garanterar att operationen inte störs av andra trådar och/eller processer och garanterar att alla jobbar med samma värde.

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

108. Förklara hand over hand locking
Hand over hand locking är ett sätt att förhindra att 2 trådar försöker komma åt en resurs samtidigt. Genom att låsa den föregående resursen i en linkad lista och låsa den resurs man försöker komma åt, så kan inte en tråd komma åt resursen förrens den föregående resursen är upplåst. Man kan jämföra tekniken med att klättra upp för en stege, där du tar en hand i taget för att klättra uppåt.

109. Vad är nackdelarna med att låsa “för mycket”?
Algoritmen tar längre tid och blir inte särskilt effektiv. Risken för deadlocks ökar också när du låser mycket.

110. Vad är nackdelarna med att låsa “för lite”?
Nackdelen med att låsa för lite är att risken för att flera trådar kommer åt samma resurs samtidigt ökar, vilket gör att trådarna kan få ouppdaterade operationer. Risken för deadlocks ökar också när flera trådar försöker komma åt samma resurs, och det kan leda till dålig effektivitet då trådar kan behöva vänta på sin tur.

111. Vad är fördelarna med en optimistisk algoritm (med avseende på låsning)?
Some advantages of optimistic locking include preventing users and applications from editing stale data, notifying users of any locking violation immediately when updating an object, not requiring you to lock up database resources, and preventing database deadlock.

It can also reduce locking overhead and improve performance and scalability by avoiding the need to acquire and release locks on the database.

112. Vad är nackdelarna med en optimistisk algoritm (med avseende på låsning)?

- det kräver underhåll av de olika tillstånden och versioner
- Man måste implementera felhantering och parallellisering själv på grund av dess komplexa design
- Dom är inte särskilt skalbara så det är svårt att anpassa algoritmen till ett problem med förbestämda strikta parametrar och krav.

113. Förklara hur radering i en länkad lista kan implementeras med en optimistisk algoritm

114. Vad innebär volatile i Java?

Nyckelordet volatile innebär att variabeln alltid sparas till primärminnet och inte i cache. Detta gör att alla trådar alltid får den senaste versionen av ett värde. Detta skapar även extra overhead.

115. Förklara vad happens before i Javas minnesmodell

"Happens-before" är en relation mellan händelser som beskriver ordningen av utförandet av operationer i Java. Det är en del av Java Memory Model (JMM), som är en modell för hur Java-program interagerar med datorns minne och hur flera trådar kan samarbeta.

Här är några exempel på situationer där "happens-before" relationen gäller i Java:

- Om en tråd skriver till en variabel, och sedan en annan tråd läser från samma variabel, måste skrivningen "happen-before" läsningen för att garantiera korrekt ordning.
- Om en tråd startar en annan tråd, garanterar "happens-before" att alla synliga ändringar som gjorts av den ursprungliga tråden är synliga för den nystartade tråden.
- Om en tråd frigör ett lås och sedan en annan tråd tar låset, garanterar "happens-before" att alla synliga ändringar som gjorts innan låset frigjordes är synliga för den andra tråden när den tar låset.

116. Vilka antaganden kan göras om två trådar modiferar en delad variabel (enligt Javas minnesmodell).

Enligt Javas minnesmodell finns det inga garantier om när eller i vilken ordning två trådar som modifierar en delad variabel kommer att se de uppdateringar som utförs av den andra tråden. Detta är känt som en "race condition" och kan leda till odefinierade beteenden och felaktiga resultat.

Mer specifikt kan följande antaganden göras om två trådar modifierar en delad variabel:

1. Om båda trådarna försöker att läsa och skriva till samma minnesplats samtidigt kan det leda till en "race condition". Detta beror på hur minnesåtkomsten sker på en specifik arkitektur.

2. Även om båda trådarna har läst och skrivit till samma delade variabel, så kan de ha gjort det i olika ordning eller på olika tidpunkter. Detta kan påverka hur variabeln används i programmet.
3. Om en tråd skriver till den delade variabeln och en annan tråd läser från den, är det inte garanterat att den andra tråden ser de senaste uppdateringarna.

För att undvika dessa problem är det viktigt att använda lämpliga synkroniseringsmekanismer som exempelvis lås eller semaforer för att se till att alla trådar har en konsistent och korrekt syn på den delade variabeln.

117. Ange några problem med att skriva flertrådade program
Race conditions
Deadlocks
Extra overhead

118. Varför kan det vara svårt att sätta samman (compose) flera flertrådade funktioner?
Att sätta samman flera flertrådade funktioner kan vara svårt eftersom det kan uppstå komplexa problem relaterade till synkronisering och trådsäkerhet.

F Flertrådade funktioner innebär att flera trådar av kod kan köras samtidigt, vilket ökar möjligheten till biverkningar eller "race conditions" där två eller flera trådar försöker åtkomma eller ändra samma resurs samtidigt. Detta kan leda till felaktiga resultat eller kraschar.

När man sätter samman flera flertrådade funktioner måste man ta hänsyn till hur trådar kan samverka och vilka resurser som delas mellan dem. Man måste se till att synkronisering används på rätt sätt för att undvika problem som "race conditions" eller att en tråd försöker åtkomma en resurs som redan har tagits bort av en annan tråd.

119. Vad är en Future?
En future i Java är ett objekt som representerar resultatet av en asynkron operation som utförs i bakgrunden. Det används för att göra kod mer parallell och effektiv genom att låta en tråd fortsätta köra medan en annan tråd hanterar operationen i bakgrunden. Futures är användbara för att hantera långsamma eller resurskrävande operationer och är vanliga i Java-baserade ramverk och bibliotek.

120. Förklara begreppet coroutine
En coroutine är en typ av lätta trådar eller kooperativ multitrådning som är en del av en tråd som körs i en kontext som möjliggör temporär pausning och återupptagning av dess exekvering. De används ofta i sammanhang där man har flera “väntade” tillstånd och värden i sitt system som till exempel databasanrop och nätverksanrop. De är användbara där parallellism inte är så viktigt men det fortfarande är viktigt att programmet är responsivt och kan användas effektivt.

121. Förklara begreppet cooperative multitasking
Cooperative multitasking är en form av multitasking som innebär att flera uppgifter eller processer delar på samma tråd eller CPU och samarbetar om att exekvera sina respektive

delar av koden. I denna typ av multitasking är det upp till de olika uppgifterna att frivilligt ge upp kontrollen över CPU-tiden och låta andra uppgifter exekvera.

I motsats till preemptive multitasking, där operativsystemet aktivt byter mellan olika processer eller trådar baserat på en fördefinierad tidskvant eller prioritet, är cooperative multitasking mer flexibel och tillåter mer kontroll över hur CPU-tiden fördelas mellan olika uppgifter. Detta beror på att samarbetet mellan uppgifterna är mer direkt och kontrollerat av själva programmet istället för av operativsystemet.

122. Vad gör yield i Python eller Java (Thread.yield)
I Python används nyckelordet "yield" i en funktion för att skapa en generator. En generator är en typ av iterator som kan skapa och ge tillbaka flera värden över tiden, i stället för att generera dem alla på en gång. Exempel:

```
def tal_sekvens(n):
    i = 0
    while i < n:
        yield i
        i += 1

sekvens = tal_sekvens(5)
print(next(sekvens)) # Output: 0
print(next(sekvens)) # Output: 1
print(next(sekvens)) # Output: 2
print(next(sekvens)) # Output: 3
print(next(sekvens)) # Output: 4
```

123. Förklara begreppet asynchronous programming
Asynkron programmering är en programmeringsmetodik där programmet kan utföra flera uppgifter samtidigt och parallellt utan att blockera varandra. Det används ofta för att öka prestandan eller för att hantera flera källor av data samtidigt. Det använder sig av callbacks, promises eller futures för att hantera resultatet av asynkrona uppgifter.

124. Vad är en event loop
En event loop är en mekanism som används för att hantera asynkrona händelser och I/O-operationer i ett program. Den sätter upp en loop som kontinuerligt kontrollerar om det finns nya händelser som väntar på att behandlas, och tillåter att flera händelser hanteras samtidigt utan att blockera applikationens huvudtråd. Detta gör att applikationen kan vara mycket effektiv och snabb i att svara på händelser och förbättra prestandan i applikationen.

125. Förklara hur en event loop kan användas för asynchronous programming
En event loop kan användas för asynkron programmering på flera sätt. Ett exempel är att använda den i webbutveckling där man vill hantera många anslutningar samtidigt. När en användare gör en förfrågan, registreras en callback-funktion för den förfrågan. Sedan kan event loopen vänta på svar från olika databaser eller andra servrar medan den samtidigt hanterar andra förfrågningar från andra användare. När svar från en server har kommit

tillbaka, anropas den tillhörande callback-funktionen för att skicka svaret tillbaka till användaren.

126. Skriv ett Python-program som schemalägger en funktion att köra på event loop om 30 tidsenheter

127. Förklara begreppet callback
I programmering refererar callback till en metod eller funktion som skickas som en parameter till en annan metod, och som sedan kallas när en viss händelse inträffar eller en viss operation är klar.

Callback används ofta för att hantera asynkrona operationer eller händelser i program. Istället för att vänta på att en operation ska slutföras, kan en callback-funktion skickas som parameter till operationen. När operationen är klar kommer callback-funktionen att kallas, vilket tillåter programmet att fortsätta med andra uppgifter medan operationen utförs i bakgrunden.

128. Vad händer om en blockerade funktion körs av en funktion på event loop
Om en blockerande funktion körs på en event loop, kommer event loop att vänta på att denna funktion ska slutföras innan den fortsätter att köra andra funktioner. Detta kan leda till att andra funktioner på event loop blockeras och att hela programmet kan bli långsamt eller slutar svara.
129. Definiera och anropa en asynkron funktion/coroutin i Python
130. Vad gör await i Python?
await i Python används i samband med asynkron programmering för att pausa exekveringen av en funktion tills resultatet av en asynkron operation har blivit klart. Det gör att funktionen kan fortsätta exekvera utan att blockera hela programmet från att fortsätta köra andra uppgifter samtidigt.

131. Blockerar await i Python?
await kan blockera den aktuella koroutinen, men det blockerar inte hela programmet eftersom andra korutiner fortfarande kan exekveras samtidigt.
132. När bör man använda asynchronous programming?
Asynkron programmering är användbart i situationer där programmet behöver utföra flera uppgifter samtidigt eller parallellt utan att blockera varandra. Här är några exempel på när man kan överväga att använda asynkron programmering:

Nätverkskommunikation: när man behöver kommunicera med en extern webbtjänst eller databas kan asynkron programmering användas för att skicka begäranden och fortsätta arbeta medan man väntar på svar.

Användargränssnitt: i GUI-programmering kan asynkron programmering användas för att hålla användargränssnittet interaktivt medan bakgrundsuppgifter körs.

IO-operationer: vid filhantering eller inläsning från externa enheter som hårddiskar, kan asynkron programmering användas för att fortsätta exekvera andra uppgifter medan filerna läses in.

Beräkningar: i situationer där tunga beräkningar ska utföras, kan man använda asynkron programmering för att hålla programmet interaktivt medan beräkningarna körs i bakgrunden.

133. Hur bör man hantera om en asynkron funktion/corutin i Python behöver göra en CPUkrävande operation?
Om en asynkron funktion eller coroutine i Python behöver utföra en CPU-krävande operation, kan detta orsaka en blockerande effekt och fördröja andra asynkrona operationer som är i kö. För att hantera detta kan man använda en teknik som kallas för "yield from" eller "awaitable", som tillåter att CPU-krävande operationer hanteras på ett annat sätt.

134. Vad händer om en asynkron funktion/corutin i Python anropar en blockerande funktion? Motivera ditt svar.

Om en asynkron funktion eller coroutine i Python anropar en blockerande funktion, kan detta leda till att hela programmet blockeras. Blockerande funktioner är funktioner som väntar på att något ska hända, exempelvis när man läser eller skriver data från en fil, eller när man väntar på en nätverksförfrågan att returnera data.

När en blockerande funktion anropas i en asynkron funktion eller coroutine, kommer hela den asynkrona funktionen att blockeras och inte fortsätta förrän den blockerande funktionen har slutfört sin operation. Detta kan leda till att andra asynkrona funktioner som är i kö också blockeras, vilket kan orsaka en kaskadeffekt och fördröja hela programmet.

135. Vad är en CompletableFuture i Java?
En CompletableFuture i Java är en klass som används för asynkron programmering och hantering av promises/futures. En CompletableFuture representerar en potentiell framtida värde eller undantag, som kan behandlas och manipuleras på olika sätt för att skapa en pipeline av asynkrona operationer.

Förutom att hantera asynkrona operationer, kan CompletableFuture också användas för att hantera undantag och fel. En CompletableFuture kan ha en hanterare som reagerar på olika undantag som kan uppstå under exekveringen av en operation, och kan också ha fallback-metoder som körs om en operation misslyckas.

136. Definiera och anropa en metod i Java asynkront
```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MyClass {
  public static void main(String[] args) {
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Runnable task = new Runnable() {
      public void run() {
        myMethod();
      }
    };
    executor.submit(task);
```

```
    // Annan kod som körs parallellt med myMethod()
  }
  public static void myMethod() {
    // Kod som ska köras asynkront
    System.out.println("Hello from myMethod!");
  }
}
```

Parallel Programming Summary

1. **Multicore CPU**
    - **Multiple processor cores on a single chip**
    - **Enables parallelism and increased performance**
2. **Concurrency vs Parallelism**
    - **Concurrency: tasks overlap in execution but not necessarily executed simultaneously**
    - **Parallelism: tasks executed simultaneously on multiple cores or processors**
3. **Threads in Java and Python**
    - **Lightweight, share memory with parent process**
    - **Java: extend Thread class or implement Runnable interface**
    - **Python: use threading module**
4. **Processes in Python**
    - **Independent units of execution, separate memory space**
    - **Use multiprocessing module**
5. **Runnable and Callable in Java**
    - **Runnable: void run() method, no return value**
    - **Callable: V call() method, can return a value**
6. **Embarrassingly Parallel Problems**
    - **Little to no dependency or communication between tasks**
    - **Easy to parallelize, scalable performance**
7. **Global Interpreter Lock (GIL) in Python**
    - **Mutex that prevents multiple native threads from executing Python bytecodes concurrently**
    - **Impacts multi-threaded Python programs**
8. **Operating Systems**
    - **System software managing hardware and software resources**
    - **Interface and control system for computer**
9. **Process States**
    - **New, Ready, Running, Waiting, Terminated**
    - **Describes the lifecycle of a process**
10. **Stack, Heap, Data, Text in relation to processes**
    - **Stack: local variables and function call information**

- **Heap: dynamic memory allocation**
- **Data: global and static variables**
- **Text: executable code**

11. Mutual Exclusion

- **Ensuring only one thread can access a shared resource at a time**
- **Solutions: locks, semaphores, algorithms (e.g., Peterson's algorithm)**

12. Deadlock

- **Situation where processes are stuck waiting for resources held by others**
- **Four conditions: mutual exclusion, hold and wait, no preemption, circular wait**

13. Asynchronous Programming

- **Non-blocking execution, allowing multiple tasks to run concurrently without waiting for each other**
- **Event loop, coroutines, and async/await in Python**