Notat om tråder

Versjon 18. mars 2022

Av Stein Gjessing

1. Enkle tråder

De algoritmene og programmene vi har skrevet hittil i IN1010 har vært *sekvensielle,* det vil si at programmet har gjort én ting om gangen. Når et java-program starter, kaller kjøretid-systemet (the run-time system) main-metoden i hoved-klassen og eksekverer denne metoden ferdig før programmet terminerer når main-metoden er ferdig. Vi sier at alt som da eksekveres utføres av main-*tråden* til Java.

Nå skal vi se på hvordan programmet vårt kan gjøre flere ting samtidig eller i *parallell*. I Java gjøres det nettopp ved hjelp av tråder. Parallellitet i programmet vårt introduserer vi grovt sett av to grunner:

1. Oppgaven er så stor at det tar lang tid å utføre den. Hvis flere tråder samarbeider i ekte parallell om å løse oppgaven kan løsningen bli ferdig tidligere.
2. Oppgavens løsning er i sin natur parallell, dvs. det er naturlig å strukturere algoritmen og programmet som en samling parallelle aktiviteter.

Javas kjøretidssystem, sammen med datamaskinens operativsystem (OS), gjør at vi kan lage et program der flere deler av programmet utføres samtidig - av hver sin tråd.
Klassen Thread er den som gjør at vi kan kjøre tråder i Java. Klassen er definert i Javas API-bibliotek og inneholder en masse leamikk som sørger for at flere aktiviteter kan utføres i parallell, og i ekte parallell hvis vi har flere prosessorer eller kjerner i datamaskinen vår.

Når en ny parallell aktivitet skal starte opp må det gjøres i en metode som heter run() som er definert i Javas API i et meget enkelt interface kalt Runnable:

```
interface Runnable {
        void run( );
}
```

Klassen Thread ser grovt sett slik ut:

```
public class Thread {
        public Thread (Runnable r) { . . . }
        public void start ( ) { . . . }
}
```

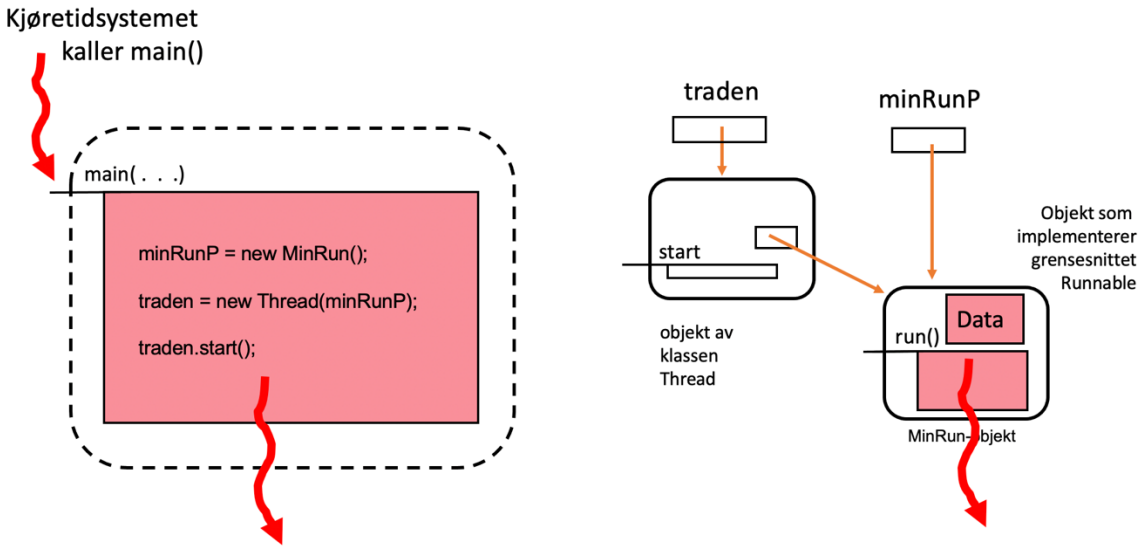
Når vi hittil bare har hatt en tråd i programmet vårt (main-tråden) så har kjøretidsystemet laget én tråd på denne måten og kalt vår main()-metode fra denne tråden. Men alt dette skjer bak kulissene.

Når vi skal programmere flere tråder og lage parallelle algoritmer må vi først og fremst finne ut hva som skal utføres i parallell. Dette er den vanskelige delen av oppgaven. Når vi starter vårt program med tråder vil kjøretidsystemet starter opp main()-metoden, og denne starter

opp den første nye tråden. Senere tråder kan startes opp av en hvilken som allerede startet tråd (inklusive main-tråden selv).

På figur 1 ser vi at kjøretidsystemet lager en tråd og kaller metoden main(). Så ser vi at main()-metoden lager et objekt av en klasse som implementerer Runnable-grensesnittet. Dette objektet ser vi nederst til høyre på tegningen. Så lager main-metoden et objekt av klassen Thread, med en referanse til det første objektet som parameter. Da får vi situasjonen til høyre på figuren. Men enda er ikke en ny tråd startet opp. Først i det main-metoden utfører traden.start() vil den nye tråden starte opp samtidig med at main-tråden fortsetter å utføre main-metoden. Da går de to aktivitetene videre i parallell, og kanskje i ekte parallell om vi har flere kjerner.

Metoden start() finnes i klassen Thread. Denne metoden sørger for at det settes av ressurser til en ny tråd og starter opp den nye tråden ved å gi kontrollen til run()-metoden. Du skal aldri kalle run()-metoden selv fra programmet ditt. Programmet ditt skal kalle start() som igjen - bak kulissene - starter run().
Den nystartede tråden terminerer når run-metoden terminerer.



Figur 1

En aktivitet som alltid utføres av en egen tråd er uttegning på skjerm av et grafisk brukergrensesnitt (Graphical User Interface, GUI). I Java kalles tråden som utfører uttegning på skjerm for GUI-tråden eller «the Event Dispatch Thread» (EDT). Grunnen til at dette gjøres som en egen tråd er at det er uavhengig av annen eksekvering av programmet og en naturlig egen aktivitet (grunn 2 over). GUI skal vi komme mer tilbake til senere i IN1010.

De fleste datamaskiner inneholder i dag flere prosessorer eller kjerner. På forelesningen snakker vi mer om dette. Når et program inneholder flere tråder er det mulig for kjøretidsystemet i samarbeid med datamaskinens OS å utføre trådene på hver sin prosessor slik at trådene eksekverer i *ekte parallell.* Grunn 1 over for å bruke tråder er nettopp basert på at vi har en datamaskin med mange kjerner.

Når tråder ikke går i virkelig parallell, for eksempel på en maskin med bare én prosessor, så skifter kjøretidsystemet på å utføre (*tids-dele*, engelsk: time slicing) de forskjellige aktive trådene, for eksempel 10 ganger i sekundet, slik at alle får en viss framdrift. Hvis et Java-program er i ferd med å utføre flere tråder på en datamaskin som har flere kjerner, vil kjøretidsystemet og OS-et samarbeide og la noen tråder gå i virkelig parallell mens andre tids-deles på prosessorene.

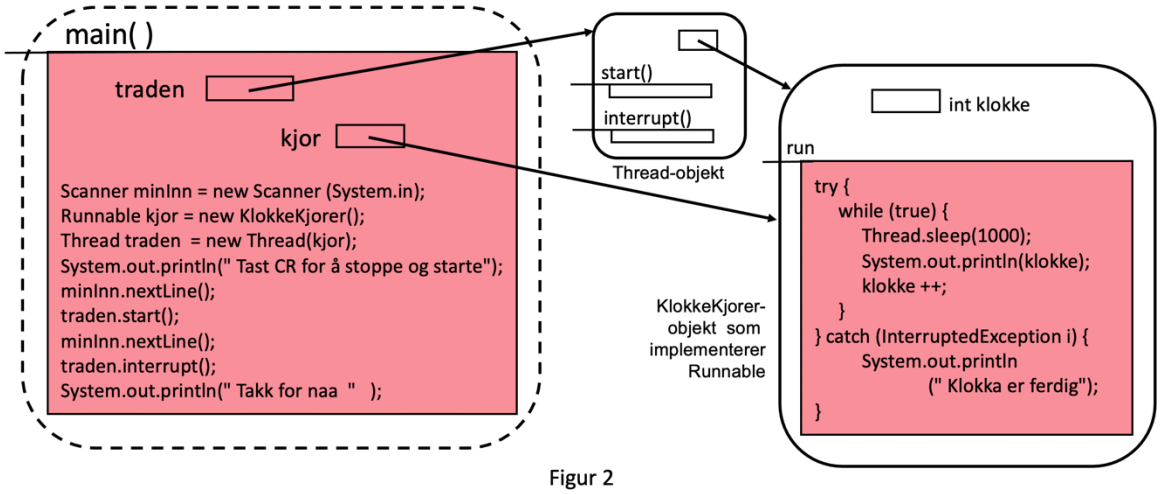
Vår første enkle eksempel på bruk av tråder i Java er en stoppeklokke som går i parallell med main-metoden (main-tråden). Hvis vi skal lage en klokke som tikker, samtidig som vi skal vente på at brukeren på terminalen skal taste noe for å stoppe den, så har vi igjen et eksempel på grunn 2 over til å bruke parallellitet i algoritmen vår. Å vikle disse to oppgaven inn i hverandre i et sekvensielt program er ikke lett. Å lage dem som to aktiviteter er både enklere å forstå, enklere å programmere og mer oversiktlig.

Vi lar klokken tikke i run-metoden mens vi lar main-metoden og main-tråden vente på at brukeren taster «ny linje» (ENTER, CR) for å stoppe klokka.

Her er først run-metoden som går i evig løkke der hvert gjennomløp av løkka venter ett sekund og så skriver ut antall sekunder. Denne run-metoden skriver vi i en klasse vi selv har definert, og som implementerer grensesnittet Runnable. Vi kaller klassen KlokkeKjorer fordi den uører (kjører) klokka:

```
class KlokkeKjorer implements Runnable {
    public void run() {
          int klokke = 0;
          try {
             while (true) {
                   Thread.sleep(1000);
                   System.out.println(klokke);
                   klokke ++;
             }
          }
          catch (InterruptedException i) {
                   System.out.println(" Klokka er ferdig");
          }
    }
}
```

Som vi ser er det en statisk metode i klassen Thread som heter sleep(). Her angis det som parameter hvor mange millisekunder den tråden som nå eksekverer skal sove.



Figur 2

I figur 2 er noe av koden som utføres av trådene farget lyserød. Stor sett er det run-metodene som er farget slik. Dette blir imidlertid veldig unøyaktig og må tas med en klype salt, fordi all kode utføres jo av en eller annen tråd. I de to første figurene er også main-metoden farget lyserød for å utheve at den uføres av main-tråden.

Klassen Thread inneholder en instansmetode kalt interrupt(). En tråd kan avbryte en annen tråd ved å kalle denne metode i den andre tråden (i det andre tråd-objektet). Vi lar brukeren starte og stoppe tråden som kjører klokka ved å taste inn en ny linje, gjerne bare en tom linje, men den må avsluttes med linjeskift (ENTER, CR). Når main-metoden leser en ny linje første gang, så starter den tråden som kjører klokka. Så legger main-metoden seg til å vente til brukeren har tastet ferdig enda en ny linje. Når main går videre etter dette kaller den metoden interrupt() i tråd-objektet for å stoppe det:

```
class TradKlokke {
   public static void main (String [] args) {
         Scanner minInn = new Scanner (System.in);
         Runnable kjor = new KlokkeKjorer();
         Thread traden  = new Thread(kjor);
         System.out.println(" Stoppeklokke  ");
         System.out.println(" Tast CR for å stoppe og starte");
         minInn.nextLine();
         traden.start();
         minInn.nextLine();
         traden.interrupt();
         System.out.println(" Takk for naa  "   );
   }
}
```

Det finnes andre og kanskje mer elegante måter for main-tråden å avslutte den tikkende klokka på. Vi kan ha en variabel som vi for eksempel kaller fortsett. Klokka (i run-metoden) sier while (fortsett) { . . . }, og når main-metoden vil stoppe den, utfører main-metoden fortsett = false;. Det kan hende at de to trådene utføres av hver sin kjerne med hver sin lokale kopi av variablen fortsett. En slik lokal kopi kan ligge i et register eller i en cashe som de to kjernene ikke deler. Når main-tråden forandrer innholdet i fortsett-variablen er det ingen garanti for at denne forandringen observeres av klokke-tråden. Det finnes imidlertid en

modifikator i Java som garanterer at en forandring av innholdet i en variabel synes for alle de andre trådene i programmet. Om vi deklarerer boolean volatile fortsett = true; så garanterer Javas kjøretidsystem at den nye verdien blir skrevet tilbake til minnet, at alle andre lokale kopier av denne variablen blir fjernet og at alle tråder må laste inn en ny og frisk kopi fra minnet når variablen skal brukes. På denne måten vil også run-metoden se at fortsett-variablen har fått verdien false, og terminere while(fortsett) { . . . }-løkka.

2. Samarbeid mellom tråder.

I klokke-eksemplet var det en veldig enkel kommunikasjon mellom de to trådene. Når tråder skal samarbeide tettere er det vanlig at de skriver og leser felles data. I IN1010 legger vi vekt på objektorientert programmering, og da er det naturlig at trådene samarbeider ved hjelp av felles objekter.

Når to tråder leser og skriver i de samme dataene samtidig skjer det lett kluss. Skal noen sette inn og ta ut penger (nesten) samtidig fra den samme kontoen kan dette ikke gjøres helt samtidig. Selv om beløpet bare lå i én minnelokasjon finnes det ingen datamaskininstruksjon for å legge til (eller trekke fra) et tall fra denne minnelokasjonen. Saldo må først hentes opp til et register i prosessoren. Der må det legges til eller trekkes fra et beløp, før resultatet legge ned igjen i minnet. Mellom disse maskininstruksjonene kan en tråd bli avbrutt når som helst og to tråder som eksekverer i ekte parallell kan aksessere minnet annen hver gang.

Hvis vi derimot hadde klart å gjør disse stegene i én ikke avbrytbar sekvens, så hadde vi vært reddet. Hvorfor det? La oss se mer detaljert på hva som kan skje hvis denne sekvensen av operasjoner blir avbrutt:

Anta at vi har en konto der en tråd skal legge til et beløp, mens en annen tråd skal trekke fra et beløp. Når to slike tråder går i tilsynelatende (eller virkelig) parallell kan den ene tråden bli avbrutt av operativsystemet eller kjøresystemet når som helst, for eksempel rett etter at den har utført en instruksjon for å hente saldoen inn i et register. Hvis nå tråden som skal legge til et beløp blir utført, så vil den også hente saldoen opp til et register. Kanskje denne tråden får fortsette en stund, så den legger til et beløp og skriver den nye saldoen tilbake til minnet. Etter en stund vil den første tråden slippe til igjen. Den har allerede saldoen i et register (og tror dette er riktig saldo), så den trekker fra et beløp og skriver den nye saldoen ned i minnet. Men da overskriver den det beløpet som lå der, og følgelig vil resultatet av å legge penger til kontoen ikke synes. Det er bare effekten av å trekke penger fra kontoen som synes.

Mekanismen som hindrer slike feil kalles *kritiske regioner*. Når vi skal operere på (skrive og lese) felles data må dette gjøres i en del av programmet som er en kritisk region. Bare én tråd kan eksekvere inne i en kritisk region (som beskytter de samme dataene) om gangen. Kommer det en eller flere andre tråder og vil inn i den kritiske regionen der en annen tråd allerede er inne, så må de nyankomne trådene vente.

Tidligere i IN1010 har vi sett mye på beholdere som kan ta vare på objekter på forskjellige måter. Disse beholderne har vært implementert som objekter, f.eks. med settInn()- og taUt()-metoder. Hittil har disse metodene vært kalt av den samme tråden, main-tråden. Når vi bruker tråder vil slike beholdere ofte hjelpe til med å kommunisere mellom tråder. F.eks. kan noe tråder (kalt produsenter) legge data inn i en buffer (ved å kalle settInn()-metoden), mens

andre tråder (kalt konsumenter) tar data ut fra bufferen (ved å kalle taUt()-metoden). Men det vi må passe på er at to tråder ikke utfører kode inne i bufferen samtidig; vi må utføre koden i settInn() og taUt() som kritiske regioner.

Kritiske regioner kan implementeres på mange måter, og de som er interessert i en (ikke objektorientert) måte å gjøre dette på kan lese om *semaforer*. Semaforer er ikke pensum i IN1010 og vil ikke bli behandlet mer.

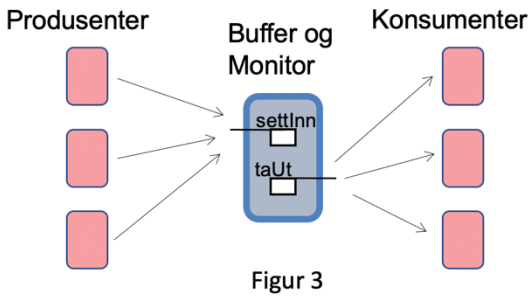
Som nevnt over, i IN1010 er felles data innkapslet i objekter, og når flere tråder kan operere på disse dataene samtidig må (public) metodene som gjør dette være kritiske regioner. Et slikt felles objekt der metodene er kritiske regioner kalles en *monitor*. Dette er en mekanisme som ble funnet på av C.A.R. Hoare i 1974.

Da Java ble laget ble monitorer introdusert i språket (selv om ordet monitor ikke er noe nøkkelord i Java). Nøkkelordet *synchronized* ble brukt foran et metodenavn for å vise at en metode skulle være en kritisk region, og alle objekter i Java har en lås som kan brukes til å låse objektet slik at bare en tråd om gangen får tilgang. Det ble også laget tre metoder i klassen Object: wait(), notify() og notifyAll() som skulle brukes til dette formålet. Uheldigvis så ble semantikken til disse mekanisme ikke bra. Hvis flere tråder venter på å slippe til for å utføre en metode (en kritisk region) i en monitor, så ble det ikke definert hvilken av disse som ble sluppet til. Dette kan føre til at en tråd tilfeldigvis blir forbigått gang på gang og ikke får slippe til.

Doug Lea laget derfor et bibliotek rundt år 2000 som er mye bedre, og det er dette vi skal bruke i IN1010. Å bruke dette biblioteket istedenfor Javas iboende mekanismer krever litt mer skrivearbeid av programmereren og det er (dessverre) litt flere sjanser for å programmere feil. På den andre siden får vi verktøy som er mye bedre å bruke når de brukes riktig.

Vi skal nå ta for oss et eksempel på samarbeidende tråder. Vi skal gjøre dette med et generelt scenario, som vi ellers i IN1010 kan gjøre mer konkret.

Et typisk eksempel på en monitor er et buffer. Et buffer har vanligvis (minst) to metoder, settInn() og taUt(), se figuren til høyre. Figur 3 viser tre produsent-tråder og tre konsument-tråder som samarbeider.



Figur 3

Produsentene lager data. Dette kan foregå på forskjellige måter. Det kan være å lese data fra en fil, lese av målinger fra sensorer, foreta beregninger, med mer. Her skal vi ikke se på et spesielt tilfelle men behandle dette helt generelt. Konsumentene mottar data fra produsentene og behandler dataene videre. Grunnen til at ikke én produsent sender data videre til én bestemt konsument er at tiden det tar å produsere og tiden det tar å konsumerer kan variere. Vi vil derfor at produsentene og konsumentene eksekverer mest mulig *asynkront.* Det får vi

til ved at data mellomlagres i et buffer. Da kan både produsenter og konsumenter jobbe på full fart, uten at en konsument må vente på en bestemt produsent.

Bufferet inneholder felles data for produsentene og konsumentene. Derfor må metodene settInn() og taUt() være kritiske regioner. Dette får vi til ved hjelp av et låse-objekt, dvs. et objekt av en klasse som implementerer grensesnittet Lock (vi skal se mer på hvilken implementasjon vi bruker senere). I det en metode starter, utfører tråden med en gang laas.lock(); der laas er en referanse til et låse-objekt.

Uansett hva som skjer inne i metoden, så må laas.unlock(); utføres før metoden terminerer. Hvis ikke vil monitoren forbli låst og ingen kommer inn i den senere. For å sikre dette bruker vi konstruksjonen try{ . . . } finally {laas.unlock();}. try-finally sørger for at finally-blokken alltid blir utført, også hvis det skjer et avbrudd inne i metoden og også etter at en return-setning er utført. Se eksempel i figur 4.

Om å vente på en bestemt tilstand i et felles objekt

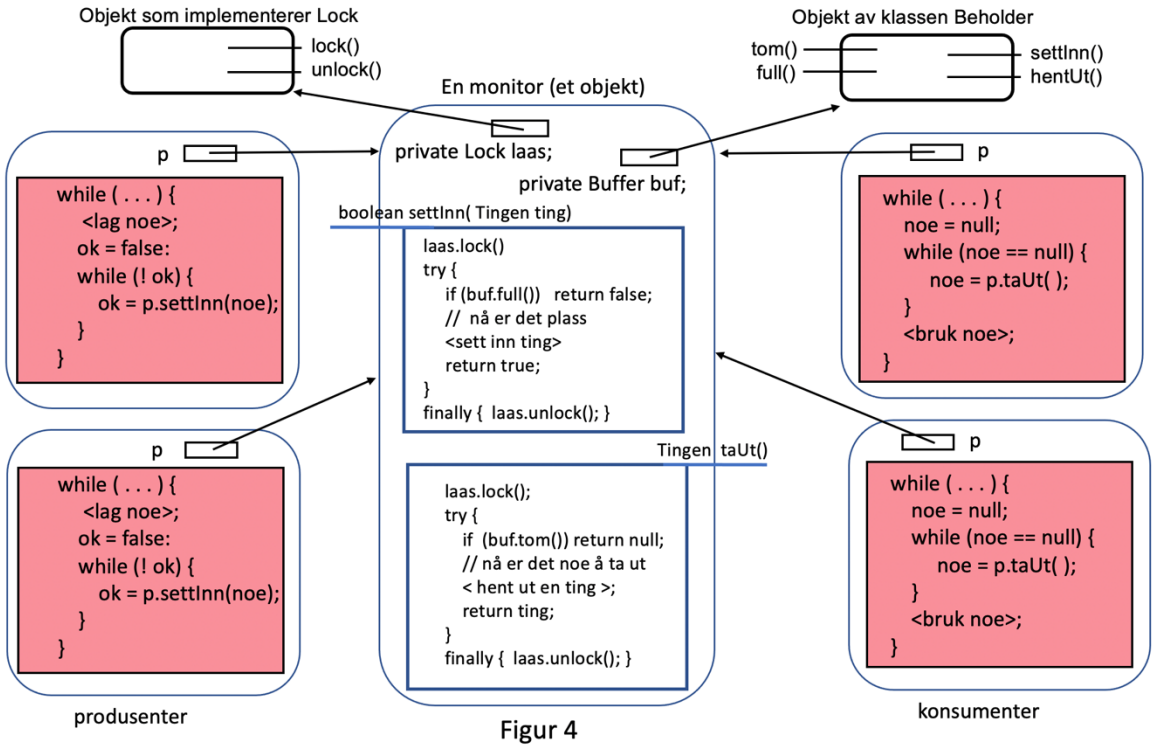
Som beskrevet over har vi hittil i IN1010 laget buffere der programmet vårt kan legge inn og ta ut data. Men hittil har dette bare vært gjort av én tråd, og om bufferen er full vil det ikke finnes noen annen tråd som kan forandre på dette. Med flere tråder der mange setter data inn i objektet og mange andre henter data ut fra objektet kan vi få en annen mulighet.

La oss repetere: Når bufferet er tomt kan en konsument ikke hente ut data fra bufferet, og når bufferet er fullt er det ikke mulig å legge noe mer inn. Men med mange produsenter og mange konsumenter vil slike tilstander (forhåpentligvis) ikke vare lenge. Når bufferet er tomt kan en konsument vent på å hente noe ut til en produsent har lagt noe inn, og når bufferet er fullt kan en produsent vente på å legge noe inn til en konsument har hentet noe ut.

Aktiv venting

En måte å løse dette på kalles *aktiv venting*. I figur 4 ser vi hva som da skjer. Når en tråd kommer inn i monitoren og finner at bufferen er tom eller full, skjønner den at her er det ikke noe å gjøre, og returnerer umiddelbart med uforrettet sak. Returverdien fra metoden sier om resultatet var vellykket eller ikke og som vi ser sjekkes dette i while-testen. Om kallet ikke var vellykket gjøres det på nytt, med håp om at nå har sitasjonen i monitoren forandret seg til det bedre for denne tråden.

Det å gå i en løkke på denne måten og stadig sjekke om resultatet nå er blitt vellykket kalles aktiv venting. Aktiv venting kan være svært fornuftig hvis ventingen er kort og tråden bare behøver å teste noen få ganger. Men ellers vil aktiv venting blokkere det felles objektet ofte og bruke prosessor-ressurser som ellers kanskje kunne vært brukt til noe annet. Og hvis flere venter aktivt på en ressurs, slik som vi har sett her, så vil det være helt tilfeldig hvem som til slutt får den.



Figur 4

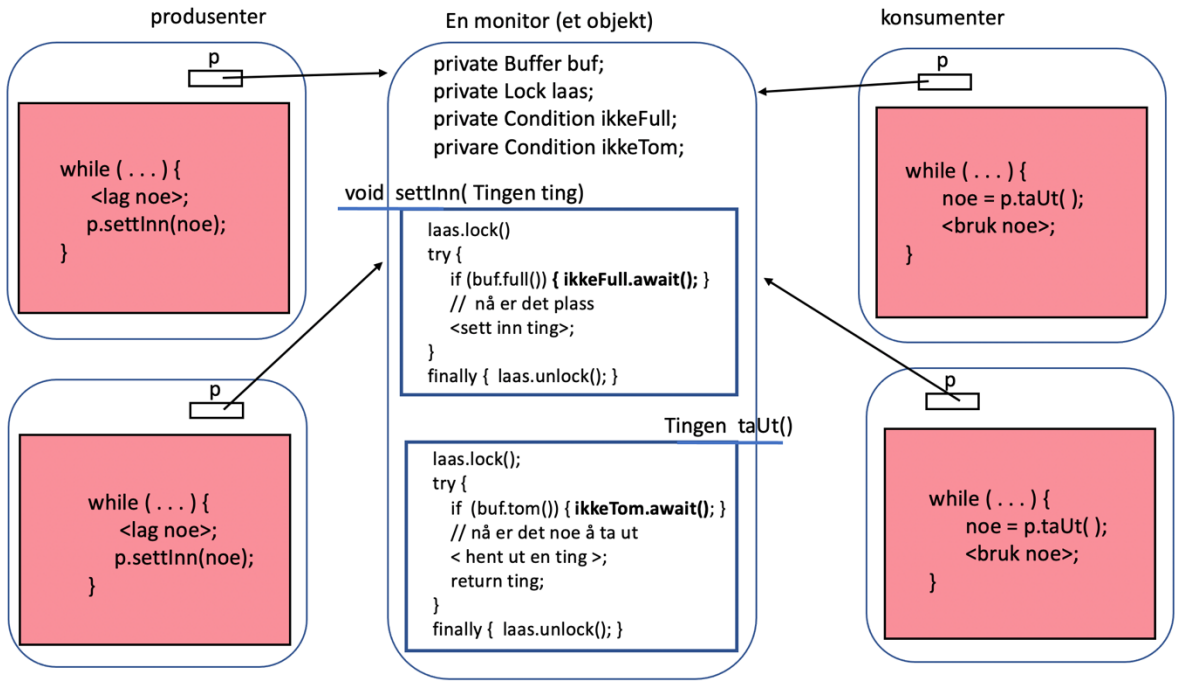
I figur 4 har vi lagt selve bufferet i et eget objekt og referer dette fra monitoren med en instansvariabel kalt buf (komposisjon / delegering). På denne måten kan vi kanskje gjenbruke en buffer-klasse vi har skrevet før. Et annet alternativ er å legge selve buffer-datastrukturen i det samme objektet som monitoren.

Passiv venting

Ved passiv venting er det et køsystem der trådene som venter blir plassert. Faktisk er det allerede et køsystem i figur 4. For å komme inn i en monitor-metode må tråden utføre laas.lock(); og hvis låsen er lukket (døren er låst) fordi det allerede er en annen tråd inne i monitoren, må tråden vente, og den venter passivt i en kø. For hvert objekt av en klasse som implementerer interfacet Lock er det en kø av tråder som venter på å slippe gjennom låsen. Når en tråd går ut av monitoren (avslutter en monitor-metode) utfører den laas.unlock(); Da må lås-objektet (objektet som laas peker på) sjekke om det ligger noen tråder i kø for å slippe inn i monitoren. Gjør det det, må en av trådene som venter, startes opp og slippes inn i monitoren. Samtidig låses låsen igjen på nytt. Denne køen av tråder som venter på å slippe inn i monitoren bør være rettferdig (FIFO – First Inn First Out).

Men egentlig er vi nå på jakt etter en måte å slippe å vente aktivt når settInn() kalles og bufferet er fullt, og når taUt() kalles og bufferet er tomt. Figur 5 viser hvordan dette kan løses. Når en tråd starter å utføre en metode, men finner at metodens hensikt ikke kan oppnås, så legger tråden seg til å vente inne i metoden, istedenfor å returnere med uforrettet sak. I det tråden starter å vente gir den også fra seg låsen, slik at en annen tråd kan slippe inn i monitoren.

Legg merke til at inne i de to metodene er det to forskjellige betingelser å vente på. Metoden settInn() må eventuelt vente på at bufferet ikke lenger er fullt (ikkeFull.await()), mens metoden taUt() må vente på at det ikke er tomt (ikkeTom.await()).



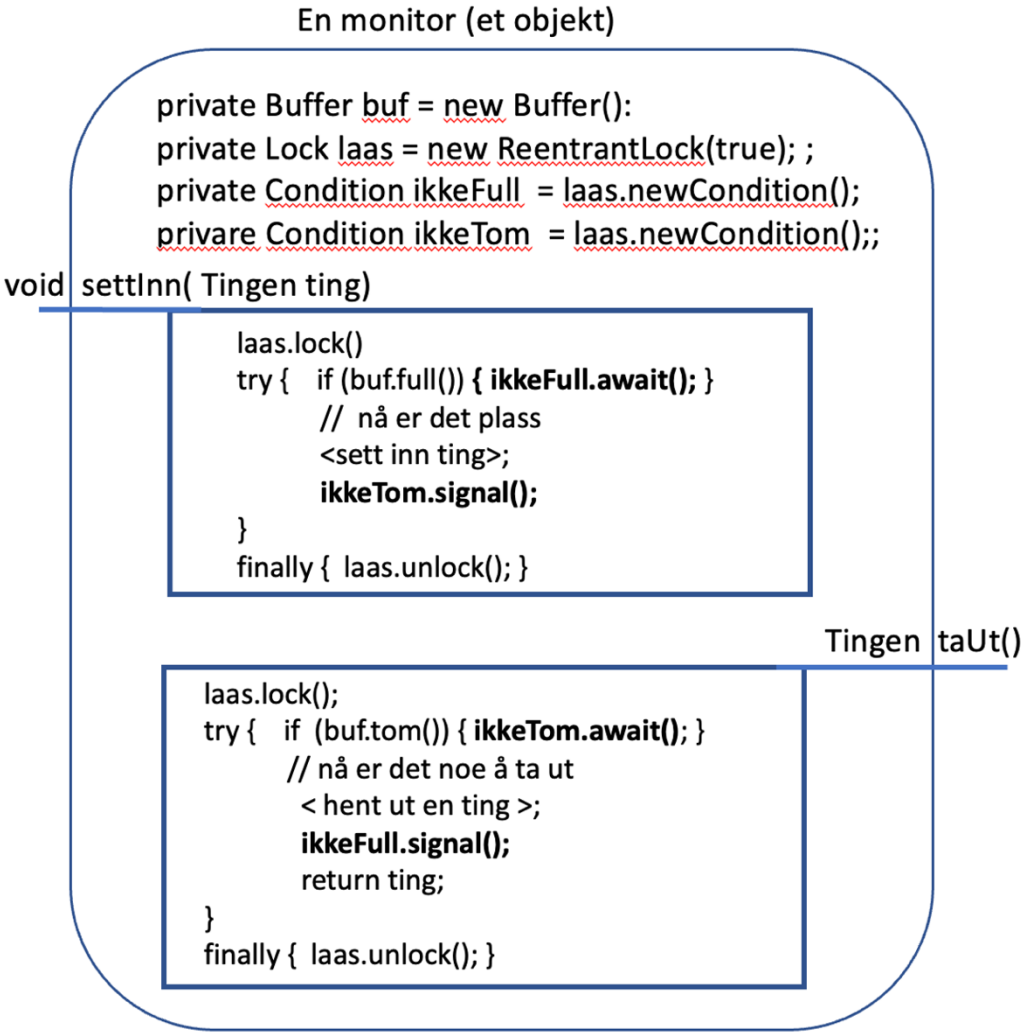
Figur 5

I figur 5 har vi følgelig tre køer av tråder som venter: En kø for de som ønsker å slippe inn i monitoren (inn i en monitor-metode, laas.lock()), og en kø for hver av de to betingelsene som det ventes på. Hver kø administreres av et vente- og kø-objekt som refereres av henholdsvis laas, ikkeFull og ikkeTom. Disse tre køene må samarbeide, derfor administreres de to køene for vente-betingelsene (engelsk: condition) av objekter som er laget av det første låse-objektet: Condition ikkeFull = laas.newCondition(); og Condition ikkeTom = laas.newCondition(); (se figur 6).

I figur 2 så vi at når en tråd sover kan den bli avbrutt. Da skjer et InterruptedException som må kunne fanges opp av programmet. Også når en tråd venter i <condition>.await(); kan det hende at tråden blir avbrutt, og dette må også kunne fanges opp. En kall på await() må derfor alltid ligge inne i en try-catch blokk, og dette har vi ikke vist på figurene her.

Signalering

De trådene som venter i køen for å komme inn i monitoren (inn i en monitor-metode) kan slippe til enten når en tråd er ferdig i monitoren (laas.unlock()) eller når en tråd må vente på en betingelse (await()). Men hvem skal starte opp de trådene som venter på en betingelse? Svaret er at det må programmereren selv sørge for at programmet gjør, og det gjøres med et kall til en av metodene signal() eller signalAll(), som begge er instansmetoder i Condition-objekter. Figur 6 viser hvordan dette kan gjøres.



Figur 6

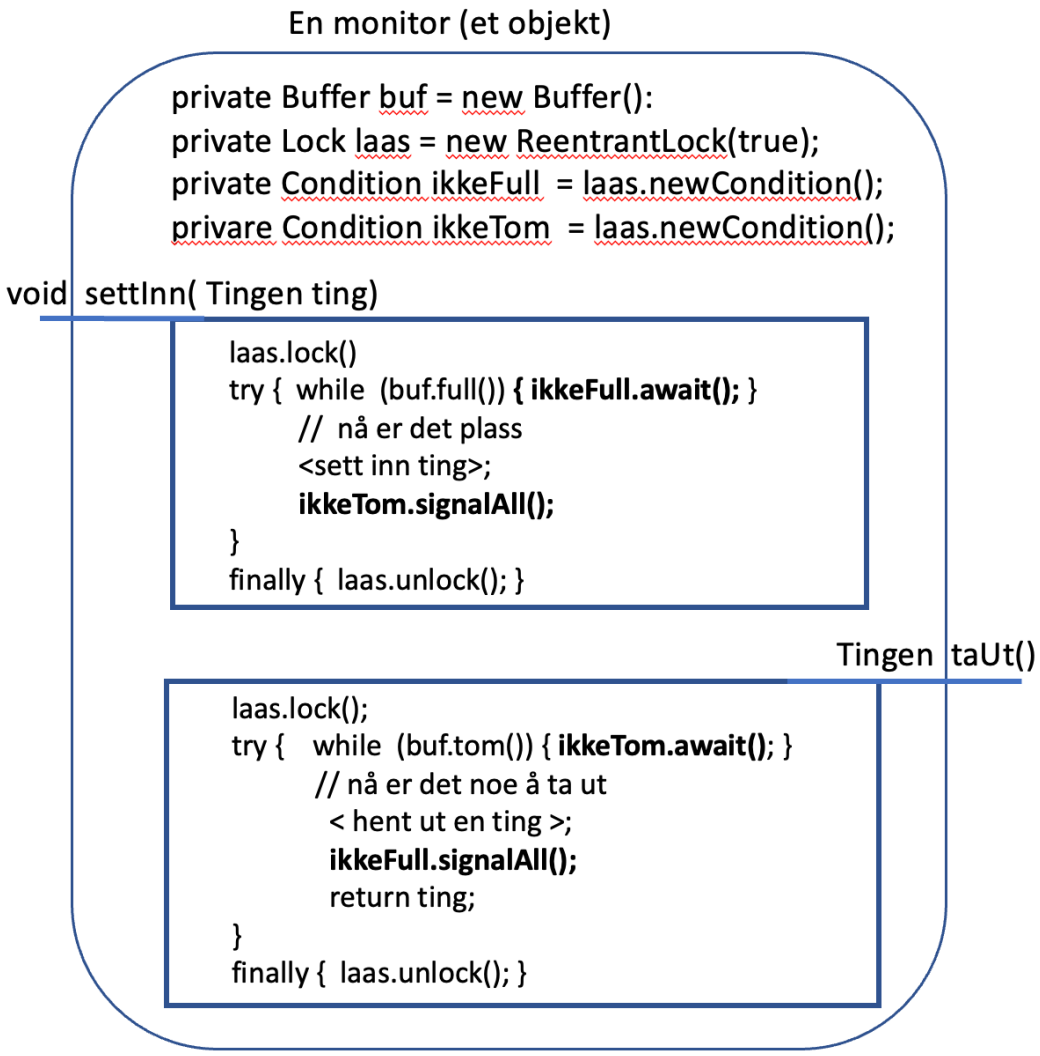
Når en tråd har tatt ut et element av bufferet i metoden taUt() så er det klart at om bufferet var fullt på forhånd, så er det ikke lenger fullt. Hvis en tråd da lå og ventet i metoden settInn(), så er det nå blitt plass til å sette inn et element. For sikkerhets skyld er det derfor alltid lurt å signalere at bufferet ikke er fullt lenger i det metoden taUt() terminerer. Slik signalering gjøres med ikkeFull.signal();. Dette vil føre til at hvis en tråd ligger og venter i ikkeFull.await(); så vil den overta monitoren og låsen etter at taUt() er terminert. Den tråden som tar over får fortsette å eksekvere videre etter ikkeFull.await();

På akkurat samme måte må settInn-metoden utføre ikkeTom.signal() når den terminerer. Da er bufferet helt sikkert ikke lenger tomt, og det er mulig for en tråd som venter i ikkeTom.await() å fortsette å eksekvere.

Vi bruker klassen Reentrantlock til å implemnterer interfacet Lock, og med parameteren «true» til konstruktøren sørger vi for at det køsystemet låsen administrerer blir rettferdig.

Om å lage robuste programmer

Hvis vi var sikre på at den tråden som ventet i await() fikk lov å eksekvere rett etter en tilsvarende signal(), så vil programmet som er vist i figur 4 være riktig. Men det er to forbedringer vi kan gjøre for å lage programmet mer robust. Dette er også programmeringsråd som er helt nødvendig å bruke i mer uoversiktlige programmer. Husk at det å programmere med tråder er svært utfordrende. Det er ikke mulig å teste alle tilfeller og svært vanskelig å forstå alle forskjellige rekkefølger trådene kan kalle metoder, vente, starte igjen osv. (engelsk: race conditions).



Figur 7

La oss se på programfragmentet if (<bufferet er fullt>) ikkeFull.await(); Når en annen tråd sier ikkeFull.signal() og metoden taUt() terminerer, så vil (antagelig) tråden som ligger og venter få starte opp igjen med en gang. Men tenk om noe annet skjer i mellomtiden. Da risikerer vi at bufferet er fylt opp igjen. For å lage robuste (og riktige) programmer legger vi alltid en await() inne i en while-løkke og re-tester betingelsen før vi går videre i programmet. Dette er illustrert i figur 7.

En annen god programmeringsregel er å være sikker på at alle som ligger og venter får en sjanse til å fortsette å eksekvere etter await(). Dette gjør programmet ved å si signalAll() istedenfor bare signal(). Når vi har lagt await() inne i en løkke er det liten skade skjedd om vi starter opp litt for mange tråder med signalAll(). Skaden er at det går litt tid mens monitoren låses opp og igjen hver gang while-løkken utføres av forskjellige tråder. På den andre siden kan vi være sikre på at alle som skal, får sjansen til å fortsette om betingelsen er oppfylt. Vi er følgelig sikrere på at monitoren er programmert riktig, og i alle fall er den mer robust. Og robusthet er spesielt viktig når vi programmerer med tråder.

3. Barrierer.

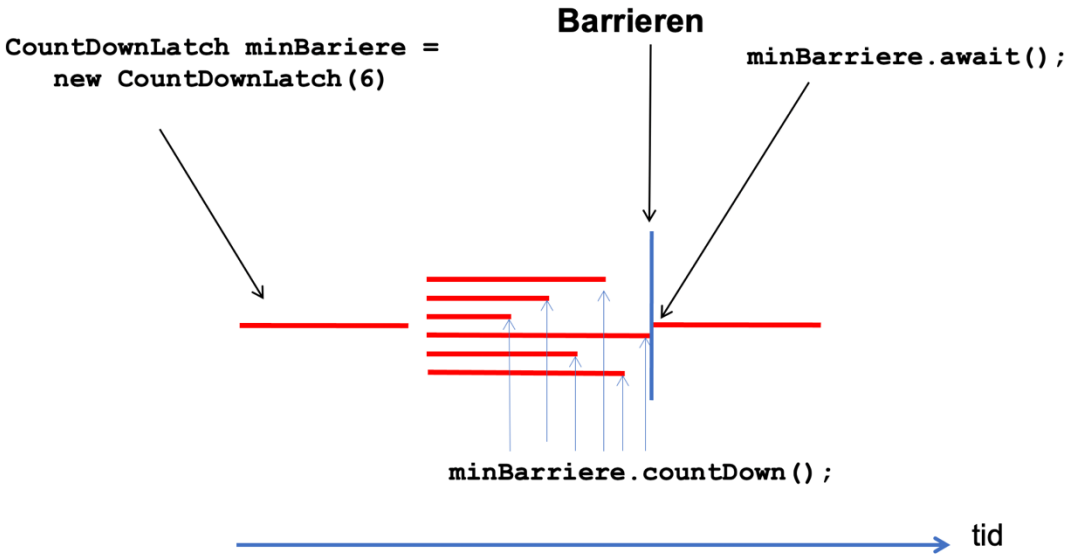
I eksemplet vårt er det ikke spesifisert hvor lenge trådene skal eksekvere. For eksempel går alle de fire trådene i figur 5 i while-løkker der tester ikke er spesifisert. Produsentene er antagelig ferdige når det ikke er mer å produsere. Konsumentene er ferdig når produsenten er ferdige og bufferet er tomt.

Ofte vil en tråd måtte vente på at andre tråder er ferdige. Hvis vi for eksempel har to tråder, og tråd 1 har en referanse kalt traden til tråd 2, kan tråd 1 utføre traden.join(); Da vil tråd 1 ligge og vente helt til tråd 2 terminerer. Hvis tråd 1 blir avbrutt mens den ligger og venter, må den fange InterruptedException.

Men ofte ønsker vi å vente på at mange tråder er ferdige. Til dette kan vi bruke en barriere, se figur 8.

En barriere er et vente- eller *synkroniseringspunkt* i programmet. Vanligvis ønsker vi at så mange tråder som mulig skal gå i parallell for å løse oppgaven vår så raskt som mulig, i alle fall så mange at vi får utnyttet prosessorkapasiteten til alle kjernene i datamaskinen. (Og kanskje ikke så mange flere heller. Det er alltid litt ekstra administrasjon med å skifte mellom tråder).

I de fleste oppgaver må vi ha et eller flere synkroniseringspunkter for å samordne oppgavene til trådene. I figur 8 tenker vi oss at vi starter opp med én tråd som starter seks andre tråder. Dette kaller vi gjerne å gafle (forgrene, grene ut, – engelsk: fork) selv om dette ikke er noe mekanisme i Java -- programmet må gå i løkke og generere og starte seks tråder. Mens disse seks trådene gjør jobbene sine ligger for eksempel den første tråden og venter. Når de seks trådene alle er ferdig kan den første tråden starte opp igjen og bruke resultatet (som kanskje ligger i en monitor).



Figur 8

En måte å få til dette på er å bruke en barriere. I Javas bibliotek finnes det flere slike, og vi skal vise bruken av en klasse som heter CountDownLatch. Når vi lager et objekt av denne klassen må vi vite hvor mange tråder vi skal vente på. I figur 8 er det 6 tråder å vente på, derfor skriver vi new CountDownLatch(6). For å lage et mer fleksibelt program burde vi nok hatt en konstant eller en variabel, for eksempel ANTALL, som angir antallet tråder, og så skrive new CountDownLatch(ANTALL).

Den første tråden lager et CountDownLatch-objekt og sender med en referanse til dette objektet til alle de seks trådene. Man sender data til tråder ved å sende dem med som parametre til konstruktøren av klassen som inneholder run-metoden (og som implementerer interfacet Runnable). Konstruktøren sørger for å lagre denne referansen i en instansvariabel. På figur 8 heter både den opprinnelige referansen og denne instansvariablen minBarriere.

Når en tråd er ferdig med jobben sin kaller den metoden countDown() i dette barriere-objektet. I mens ligger tråden som skal ha resultatet fra de seks trådene og venter ved at den har kalt metoden await() i det samme objektet. Når alle de seks trådene har utført countDown(), vil barriere-objektet vekke opp igjen tråden som venter i await().

Slutt.