

INF1010 - Våren 2013
Tråder del 1 –
Parallele programmer og felles data

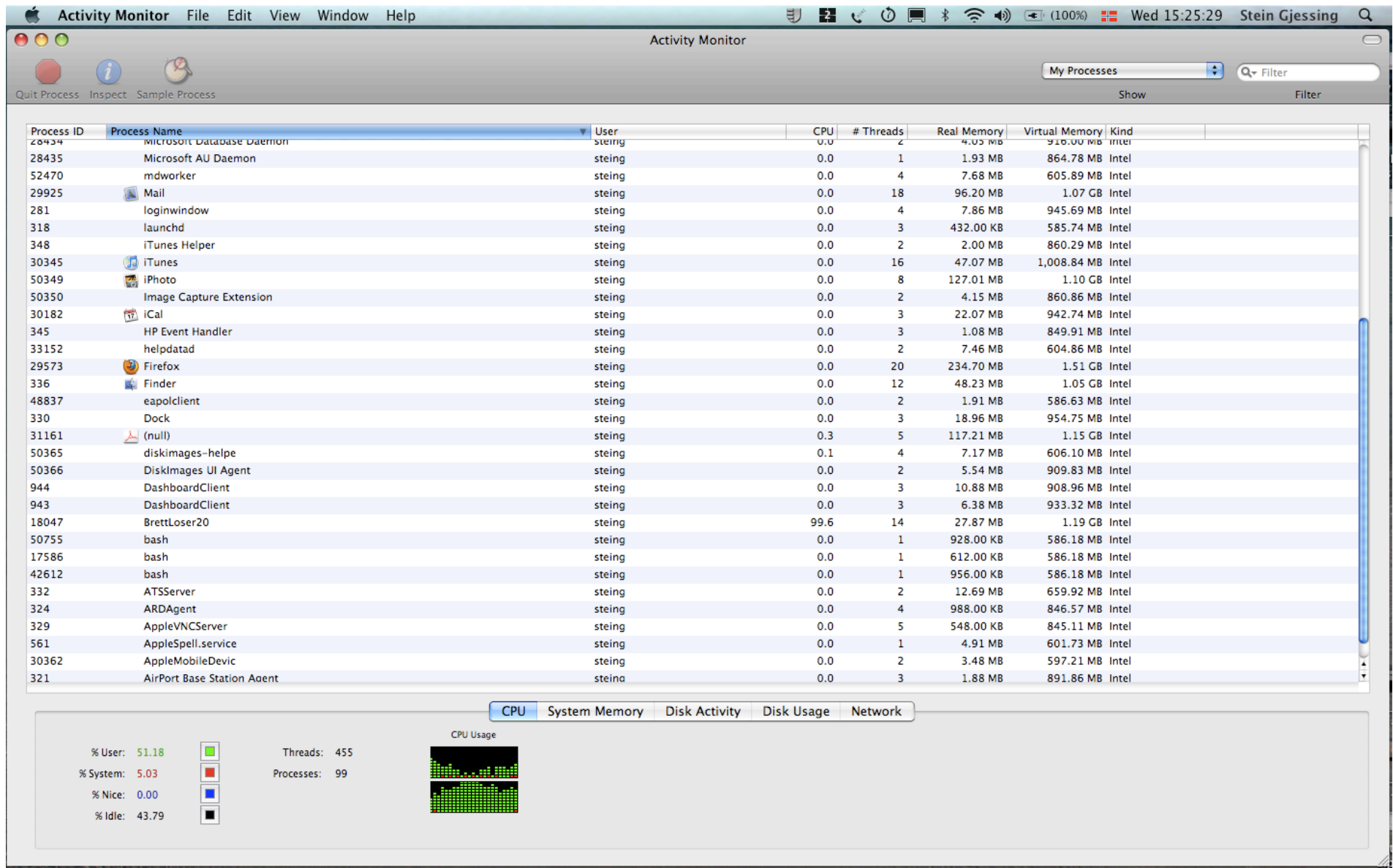
Stein Gjessing,
Institutt for informatikk,
Universitetet i Oslo



UNIVERSITETET
I OSLO



Institutt for informatikk



Oversikt

- Hva er parallelle programmer?
- Hvorfor parallelle programmer ?
- Hvordan kan dette skje på en CPU/prosessor ?
- Hvordan kan dette skje med flere CPU-er/prosessorer ?
- På engelsk: Parallel vs Concurrent computing
- Noen begreper:
 - Programmer, prosesser og tråder
 - Avbrytbare (pre-emptive) eller *ikke* avbrytbare (non pre-emptive) prosesser og tråder.
- Tråder i Java
- Hvordan nytter vi tråder
- Oppdaterings problemet:
 - Samtidig oppdatering av data
 - Løsningen: "Monitorer" = Kritiske regioner med synkroniserte metoder.



Hva er parallelle programmer - I

- På en datamaskin kan flere programmer kjøre samtidig:
- **Virkelig samtidig:** To programmer får på samme tidspunkt utført hver sin instruksjon (må da ha flere CPU-er/prosessorer/kjerner)
 - Engelsk: Parallel computing
- **Tilsynelatende samtidig:** Maskinen (CPU-en/prosessoren) skifter så raskt mellom programmene at du ikke merker det (flere ganger i sekundet). Maskinen utfører da noen få millioner instruksjoner for hvert program før den skynder seg til neste program.
 - Engelsk: Concurrent computing: Programmet/programmene kjører ekte eller tilsynelatende samtidig.
- På norsk har vi vanligvis ikke dette skille mellom ”parallel” og ”concurrent” – vi kaller alt parallell (parallelle programmer) eller samtidig
- Uansett om programmene går virkelig eller tilsynelatende samtidig, må de behandles som om de går virkelig samtidig (fordi man ikke kan forutsi når maskinen skifter fra et program til et annet)



Hva er parallelle programmer – I I

- Administrasjonen av hvilke program som kjører, og hvordan de deler tiden og kjernene mellom seg, gjøres av **operativsystemet** (Unix, Windows, Mac-OS). Operativsystemet er et meget stort program og er det første programmet som startes i maskinen. Det har alle tillatelser, men skrur mange av disse tillatelsene av for vanlige programmer (eks: skrive direkte til disken eller til alle steder i primærlageret/RAM).
- På de fleste operativsystem kan *flere* brukere være pålogget og kjøre sine programmer samtidig og *flere* uavhengige programmer kan kjøre samtidig
 - Datamaskinen kan skifte rask mellom alle de ulike programmene som er i gang i datamaskinen
 - Eks: Du kjører både et Java-program, skriver et brev i Word og får en melding
 - Unntak: en del svært enkle mobiltelefoner ol.



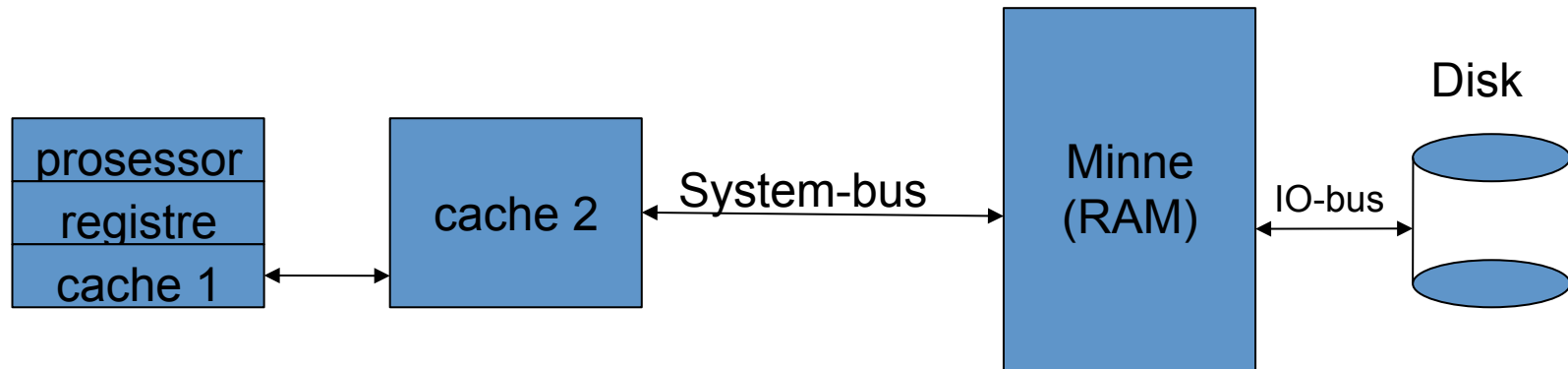
Hvorfor parallelle programmer ?

- Maskinen har mye større kapasitet enn det vi vanligvis trenger; går for det meste på tomgang.
 - Min PC kjører vanligvis mer enn 99% av tiden tomgangsprogrammet 'the System Idle Process'
 - Bortsett fra siste helg hvor 99,8% gikk med til å kjøre en simulering av en transportprotokoll i Internett
- En bruker trenger å kjøre flere programmer samtidig (se film/Skype samtidig som du jobber....)
- Ofte *må* mange ulike brukere jobbe på *samme data* samtidig (eks. et bestillingssystem med flere selgere på samme data: en kino, en flyavgang,... Dette skjer på "servere" (tjenermaskiner)
- "Dual-core", "N-core" – prosessorer: Ekte parallellitet

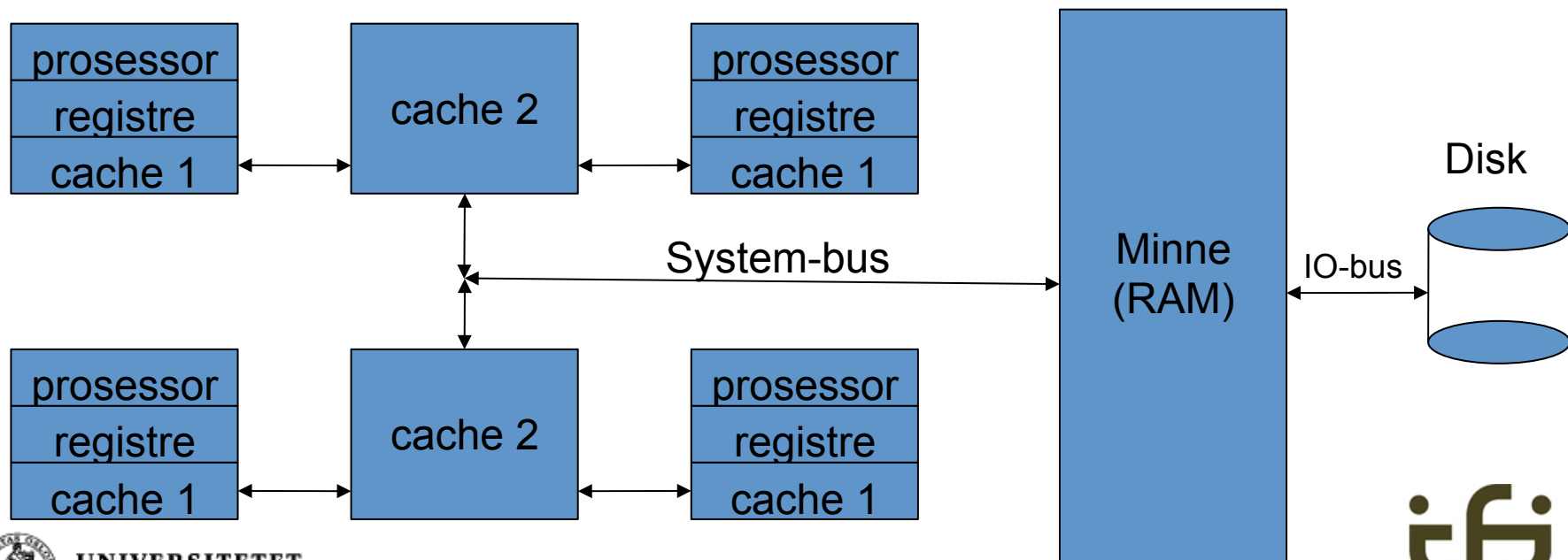


Datamaskinarkitektur (Computer Architecture)

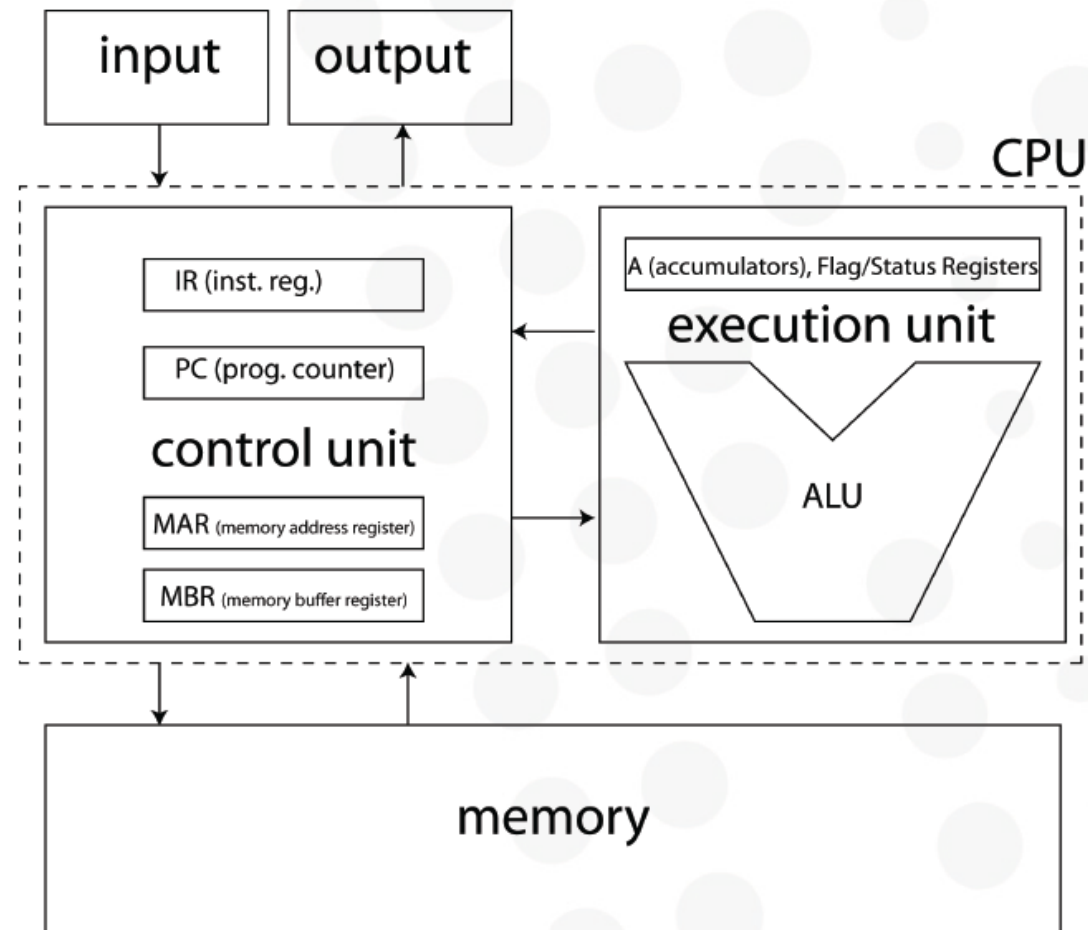
Før:



Nå, f.eks:



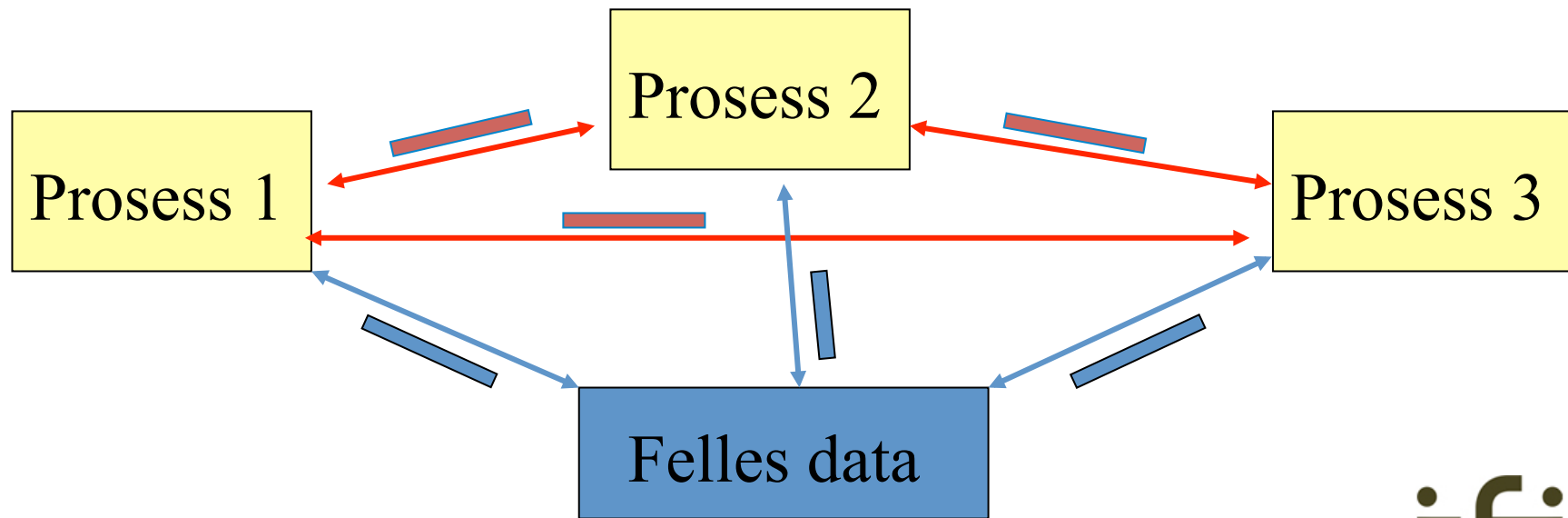
Von Neumann Architecture Block Diagram



Parallellprogrammering

Parallellprogrammering vil si å løse en oppgave ved hjelp av programmer (eller programbiter) som skal/kan utføres samtidig.

Samarbeidende prosesser sender meldinger til hverandre (røde piler)* **eller** leser og skriver i felles primærlager (blå piler) (men vanligvis ikke begge deler).



Operativsystemets oppgave (Anbefalt kurs: INF3151)

- Operativsystemet velger *hver gang* det kjører, blant de mange programmene som er klare til å kjøre, hvilke som skal kjøres nå.
- ”Hver gang” betyr:
 - Det kommer noe data utenfra (tastatur, disk, nettverk, . . .)
 - Den innebygde klokka tikker (50 ganger i sekundet)
 - En prosess er (midlertidig) ferdig

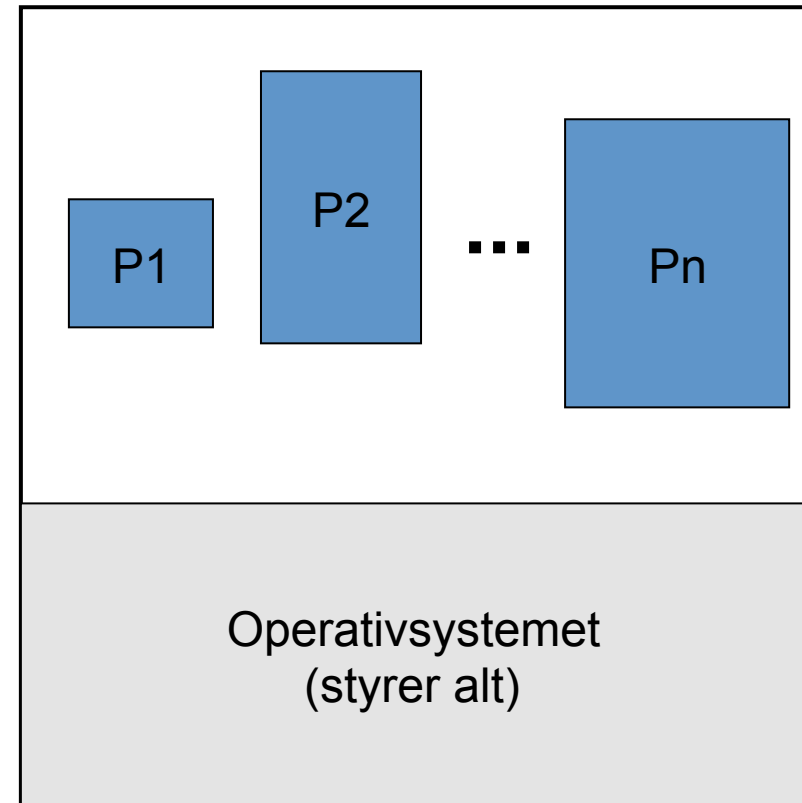
Da ’ vekkes’ operativsystemet opp og overtar kontrollen fra (avbryter) det programmet / de programmene som kjørte. Må huske hva de avbrutte programmene var i ferd med å gjøre.

- Så bestemmer operativsystemet ’ rettferdig’ hvilke program som nå skal overta og kjøre litt:
 - De med høyest prioritet må slippe til oftest og mest
 - Ingen må vente ’ alt for lenge’
 - Dersom intet program ønsker å kjøre, har operativsystemet en egen tomgangsprosess som kan kjøre (går bare rundt i en tom løkke).



Programmer, prosesser og tråder

- Operativsystemet administrerer
 - Prosesser
 - (og et antall tråder i hver prosess)
- Prosesser
 - En prosess er utføringen av et program
 - Er isolert fra hverandre, kan i utgangspunktet bare snakke til operativsystemet
 - Kan sende meldinger til andre prosesser via operativsystemet
 - Eier hver sin del av hukommelsen
 - Eier hver sine filer,...
- Et program
 - Startes som én prosess (kan så evt. starte andre prosesser)
- En tråd
 - Er parallelle eksekveringer **inne i én prosess**
 - Alle tråder i en prosess deler prosessens del av hukommelsen (ser de samme variable og programkode)
 - Tråder er som ”små”-prosesser inne i en vanlig ”stor” prosess
 - Tråder kan også gå i ekte parallell



Hvorfor fant man på tråder ?

- Vi har prosesser – hvorfor ikke bare bruke dem?
 - Det går greit, men litt tregt
 - Å skifte fra at en prosess kjører til at en annen kjører tar omlag 20 000 instruksjoner
- Prosesser ble funnet på omlag 1960, tråder minst 20 år seinere.
- Tråder er som små prosesser inne i én prosess, og det er langt raskere å skifte fra en tråd til en annen tråd (kalles ofte lettvektsprosesser)
- Prosesser kommuniserer vanligvis via operativsystemet
- Tråder kommuniserer via felles data
- Ellers har tråder og prosesser omlag samme muligheter og problemer når man lager programmer
- Parallellprogrammering
 - = bruke **flere** prosesser og/eller **flere** tråder for å løse en programmeringsoppgave
 - Programmer med tråder er mye vanskeligere å skrive og teste/feilsøke enn bare én tråd i én prosess (som vi grovt sett har gjort til nå)



Hva bruker vi tråder til?

- Alle slags tunge beregninger:
- Vi har et program som skal betjene flere brukere (f.eks. web-sider som Facebook, Google, flybestillinger, . . .).
- Tolking og visning av video (animasjon) , tolking av tale,
- Simuleringsmodeller: Etterape prosesser i virkeligheten for å finne ut statistikk/utfallet for f.eks. køer på et motorvei, variasjoner i biologiske systemer (simulere 100 000 laks fra fødsel til død) , datanettverk, hva skjer med et olje eller gassreservoar når oljen/gassen pumpes ut, krigføring (slag med tanks og fly),.... (men kvasiparallellitet er også vanlig (og enklere))
- Med flere prosessorer kan programmer med tråder gå raskere ("Dual/Quad/8/16/ -core"-prosessorer).
 - Også mulig med grafiske prosessorer
- Om å programmere med tråder og prosesser:
 - Prøv å finne det som naturlig er parallelt og uavhengig i oppgaven
 - Lag tråder eller prosesser som kommuniserer minst mulig med hverandre.
 - Mer om dette i del II om tråder.



```

public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrørtter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
        antBestilt = ant;
        for (int i = 0; i < antKokker; i++) {
            Kokk k = new Kokk(this, "Kokk nr. " + i);
            k.start();
        }
        for (int i = 0; i < antServitører; i++) {
            Servitor s = new Servitor(this, "S" + i);
            s.start();
        }
    }

    public static void main(String[] args) {
        new Restaurant(Integer.parseInt(args[0]));
    }

    synchronized boolean kokkFerdig() {
        return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
        return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
        // Kokketråden blir eier av låsen.
        while (antLaget - antServert > 2) {
            /* så lenge det er minst 2 tallerkner
             * som ikke er servert, skal kokken vente. */
            try {
                wait(); /* Kokketråden gir fra seg
                        * låsen og sover til den
                        * blir vekket */
            } catch (InterruptedException e) {}
            // Kokketråden blir igjen eier av låsen
        }
        boolean ferdig = kokkFerdig();
        if (!ferdig) {
            antLaget++;
            System.out.println(k.getName() + " laget nr: " + antLaget);
        }
        notify(); /* Si ifra til servitøren. */
        return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
        // Servitørtråden blir eier av låsen.
        while (antLaget == antServert && !servitørFerdig()) {
            /* så lenge kokken ikke har plassert
             * en ny tallerken. Derved skal
             * servitøren vente. */
            try {
                wait(); /* Servitørtråden gir fra seg
                        * låsen og sover til den
                        * blir vekket */
            } catch (InterruptedException e) {}
            // Servitørtråden blir igjen eier av låsen.
        }
        boolean ferdig = servitørFerdig();
        if (!ferdig) {
            antServert++;
            System.out.println(s.getName() + " serverer nr: " + antServert);
        }
        notify(); /* si ifra til kokken */
        return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (!rest.putTallerken(this)) {
            // levert tallerken
            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servitor(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (!rest.getTallerken(this)) {
            // hentet tallerken
            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Servitøren er ferdig
    }
}

```

Tankemodell for tråder

En tråd i programmet

```

public class Restaurant {
    int antBestilt;
    int antLaget = 0, antServert = 0; // tallerkenrørtter
    int antKokker = 5, antServitører = 50;

    Restaurant(int ant) {
        antBestilt = ant;
        for (int i = 0; i < antKokker; i++) {
            Kokk k = new Kokk(this, "Kokk nr. " + i);
            k.start();
        }
        for (int i = 0; i < antServitører; i++) {
            Servitor s = new Servitor(this, "Servitør nr. " + i);
            s.start();
        }
    }

    public static void main(String[] args) {
        new Restaurant(Integer.parseInt(args[0]));
    }

    synchronized boolean kokkFerdig() {
        return antLaget == antBestilt;
    }

    synchronized boolean servitørFerdig() {
        return antServert == antBestilt;
    }

    synchronized boolean putTallerken(Kokk k) {
        // Kokketråden blir eier av låsen.
        while (antLaget - antServert > 2) {
            /* så lenge det er minst 2 tallerkner
             * som ikke er servert, skal kokken vente. */
            try {
                wait(); /* Kokketråden gir fra seg
                        * låsen og sover til den
                        * blir vekket */
            } catch (InterruptedException e) {}
            // Kokketråden blir igjen eier av låsen
        }
        boolean ferdig = kokkFerdig();
        if (!ferdig) {
            antLaget++;
            System.out.println(k.getName() + " laget nr: " + antLaget);
        }
        notify(); /* Si ifra til servitøren. */
        return !ferdig;
    }

    synchronized boolean getTallerken(Servitor s) {
        // Servitørtråden blir eier av låsen.
        while (antLaget == antServert && !servitørFerdig()) {
            /* så lenge kokken ikke har plassert
             * en ny tallerken. Derved skal
             * servitøren vente. */
            try {
                wait(); /* Servitørtråden gir fra seg
                        * låsen og sover til den
                        * blir vekket */
            } catch (InterruptedException e) {}
            // Servitørtråden blir igjen eier av låsen.
        }
        boolean ferdig = servitørFerdig();
        if (!ferdig) {
            antServert++;
            System.out.println(s.getName() + " serverer nr: " + antServert);
        }
        notify(); /* si ifra til kokken */
        return !ferdig;
    }
}

class Kokk extends Thread {
    Restaurant rest;

    Kokk(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (!rest.putTallerken(this)) {
            // levert tallerken
            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Kokken er ferdig
    }
}

class Servitor extends Thread {
    Restaurant rest;

    Servitor(Restaurant rest, String navn) {
        super(navn); // Denne tråden heter nå <navn>
        this.rest = rest;
    }

    public void run() {
        while (!rest.getTallerken(this)) {
            // hentet tallerken
            try {
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }
        // Servitøren er ferdig
    }
}

```

To tråder i programmet

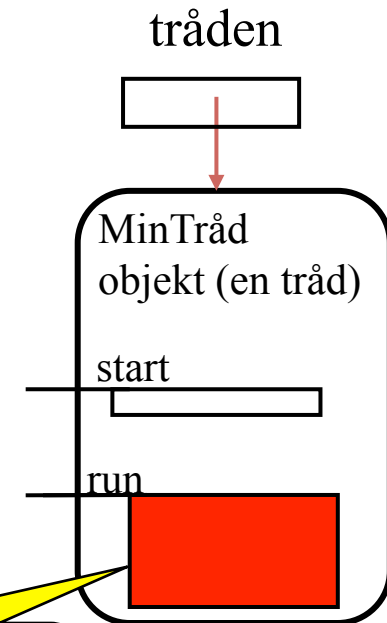
Tråder i Java

- Er innebygget i språket
- I ethvert program er det minst én tråd (den som starter og kjører i **main**)
- GUI kjører en egen tråd (Event Dispatch Thread, EDT) når knapper ol. aktiveres
- Fra en tråd kan vi starte flere andre, nye tråder
- En tråd starter enten som:
 - Et objekt av en subklasse av **class Thread** som inneholder en metode **run()** som du skriver i subklassen. Slik gjør du:
 - Lag først objektet av subklassen (new...)
 - Kall så metoden **start()** i objektet (ikke laget av deg , men arvet fra Thread). **start()** sørger for at **run()**, som du selv har skrevet, vil bli kalt.
 - Thread implementerer grensesnittet **Runnable** (se under)
 - Et objekt av en klasse som også implementerer grensesnittet **Runnable**
 - **run()** er eneste metode i grensesnittet
 - Gi dette objektet som parameter til klassen Thread, og kall **start()** på dette Thread-objektet
 - Litt mer komplisert.
 - Litt mer fleksibelt (kan da også være en del av et klassehierarki)



Tråder i Java:

```
class MinTråd extends Thread {  
    public void run( ) {  
        while (<mer å gjøre>) {  
            <gjør noe>;  
            try {sleep(<et tall, dvs. en stund>;)}  
            catch (InterruptedException e) { ... }  
        } // end while  
    } // end run  
} //end class MinTråd
```



run inneholder vanligvis en løkke som utføres til oppgaven er ferdig.

En tråd lages og startes opp slik:

```
MinTråd tråden;  
tråden = new MinTråd( );  
tråden.start( );
```



Her går den nye og den gamle tråden (dette programmet), videre hver for seg

start() er en metode i Thread som må kalles opp for å få startet tråden.
start-metoden vil igjen kalle metoden run (som vi selv programmerer).



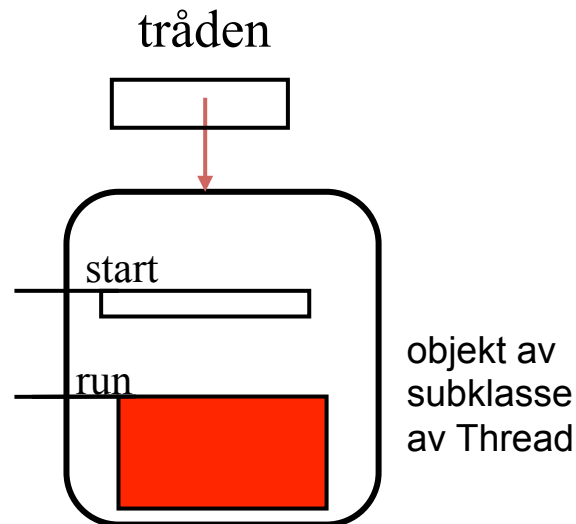
Det er det vi trenger å vite om tråder

- Nå er resten bare opp til fantasien

(og å passe på at de samarbeider riktig, mer om det senere)



```
MinTråd tråden;  
tråden = new MinTråd( );  
tråden.start( );
```



Sove og våkne

- En tråd kan sove et antall milli- (og nano) sekunder; metode i Thread:
 - static void [sleep](#) (long millis)
"Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds."
 - Andre tråder slipper da til (får kjøre)
 - Hvis noen avbryter tråden mens den sover skjer et unntak (ikke mer om avbrudd av tråder i INF1010):

```
try { sleep(1000); } // sover ett sekund  
catch (InterruptedException e) { behandle uventet avbrudd }
```



En Stoppeklokke,
subklasse av **Thread**,
inneholder :

public void run()

Den som lager objektet ,
kaller start() som igjen
kaller run() (bak
kulissene).

```
import java.awt.*;    import java.awt.event.*; import java.io.*;

class Klokke {
    public static void main(String[] args) throws IOException {
        System.out.println("Trykk [ENTER] for å starte / stoppe");
        Scanner tastatur= new Scanner (System.in);
        tastatur.next();

        // Her lages stoppeklokke-objektet:
        Stoppeklokke stoppeklokke = new Stoppeklokke();
        // og her settes den nye tråden i gang.
        stoppeklokke.start();

        tastatur.next();
        stoppeklokke.avslutt();
    }
}

class Stoppeklokke extends Thread {
    private volatile boolean stopp = false;
    // blir kalt opp av superklassens start-metode.
    public void run() {
        int tid = 0;
        while (!stopp) {
            System.out.println(tid++);
            try {
                Thread.sleep(1 * 1000); // ett sekund
            } catch (InterruptedException e)
            { System.out.println("ERROR"); System.exit(1); }
        }
    }

    public void avslutt() {
        stopp = true;
    }
}
```

Avbrytbare eller ikke-avbrytbare prosesser og tråder.

- Hvis en **prosess** (eller en tråd) prøver å gjøre en så lang beregning at andre prosesser (tråder) ikke slipper til, må den kanskje avbrytes.
- Alle skikkelige operativsystemer (Windows, MAC OS og Unix/linux) greier å avbryte både prosesser og tråder – f.eks. vha. klokka som sender avbruddssignal 50 ganger per sek.
- Gamle/enkle operativsystemer som MS-DOS og Win95 greide ikke det
 - Og kanskje fremdeles ikke noen enkle mobiltelefoner/lesebrett i dag ?
- Java-definisjonen sier at det er opp til hver implementasjon av kjøresystemet 'java' om tråder skal være avbrytbare eller ikke.
- Selv om vi har ”mange-core” prosessorer vil en tråd ikke gjøre fornuftig arbeid hele tide. Vi bør ha 2 – 8 ganger så mange tråder som prosessorer (”core-er”), avhengig av hvor mye hver tråd trenger å vente (på andre tråder eller I/O)



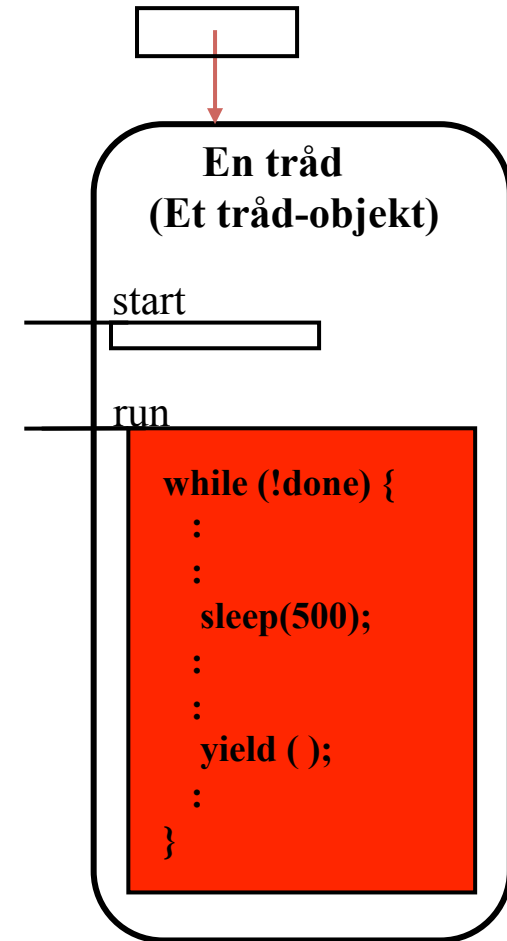
Tråder kan avbryte seg selv

Vanligvis vil operativsystemet dele prosessoren mellom alle aktive tråder (tidsdeling - “time-slicing”), og altså kaste en tråd ut av prosessoren etter en tid (på engelsk: pre-emption).

run-metoden kan eksplisitt la andre tråder slippe til ved å si `yield()`;

Noen ganger det kan være ønskelig å la andre aktiviteter komme foran.

`yield()`; slipper andre tråder til, og lar tråden som utfører `yield` vente midlertidig (men denne tråden blir igjen startet opp når det på ny er den sin tur).



Sove og vike



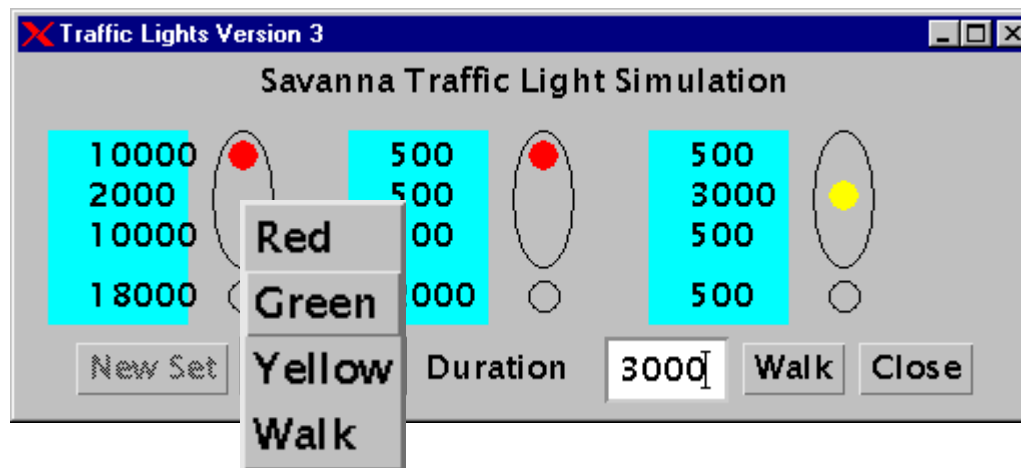
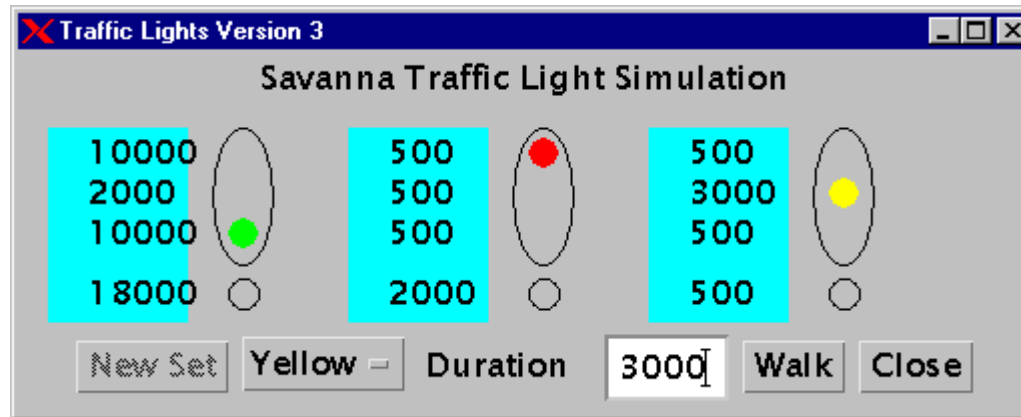
UNIVERSITETET
I OSLO



Institutt for informatikk

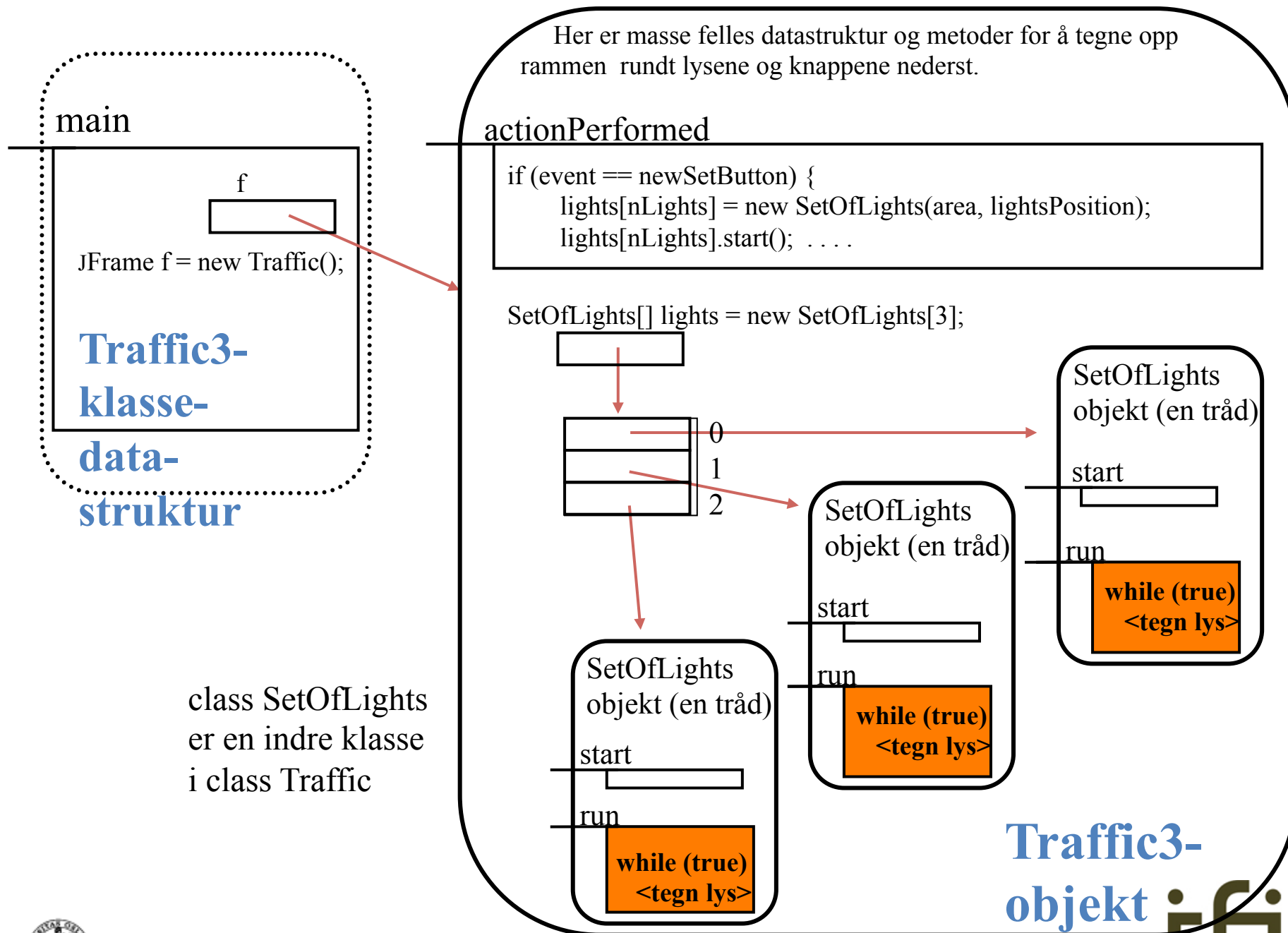
Tre trafikklys styrt av tre tråder

Laget av Judy Bishop i læreboka Java Gently



Men inni
er det
ingen ting





```
import java.awt.*;
import java.awt.event.*;

public class Traffic3 extends JFrame
    implements ActionListener, ItemListener {

    private JPanel area;
    private int lightsPosition = 105;
    private static final int lightsWidth = 150;
    private SetOfLights[] lights = new SetOfLights[3];
    private int nLights = 0, setWanted = 0;
    private JTextField duration;
    private JComboBox colours;
    private int light;
    private JButton newSetButton;
    private JButton walkButton;
    private JButton closeButton;
```



**Alle data er
skjult for
omverdenen**

```
public Traffic3() {
    super("Traffic Lights version 3 ");
    getContentPane().add("North",new Label
        ("Savanna Traffic Light Simulation",Label.CENTER));
    area = new JPanel();
    area.addMouseListener(new MouseEvtHandler());
    getContentPane().add("Center",area);
    JPanel buttons = new JPanel();
    newSetButton = new JButton("New Set");
    newSetButton.addActionListener(this);
    buttons.add(newSetButton);
    colours = new JComboBox ();
    colours.addItem("Red");
    colours.addItem("Green");
    colours.addItem("Yellow");
    colours.addItem("Walk");
    colours.setEditable(false);
    colours.addItemListener(this);
    light = 0;    buttons.add(colours);
    buttons.add(new JLabel("Duration"));
    duration = new JTextField("", 4);
    duration.addActionListener(this);
    buttons.add(duration);
    walkButton = new JButton("Walk");
    walkButton.addActionListener(this);
    buttons.add(walkButton);
    closeButton = new JButton("Close");
    closeButton.addActionListener(this);
    buttons.add(closeButton);
    getContentPane().add("South",buttons);
}
```

**Et eget
objekt
som tar
seg av
museklikk**


```
public void itemStateChanged(ItemEvent e) {
    String s = (String) e.getItem();
    if (s.equals("Red")) {light = 0;} else
    if (s.equals("Green")) {light = 1;} else
    if (s.equals("Yellow")) {light = 2;} else
    if (s.equals("Walk")) {light = 3;}
}
```

Sett den
fargen
som skal
forandres

Lyttemetoden ligger i
hoved-objektet:

```
class MouseEvtHandler extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        int n = e.getX() / lightsWidth;
        if (n < nLights)
            setWanted = n;
    }
}
```

En egen klasse
som tar seg av
museklikk

```
public static void main(String[] args) {
    JFrame f = new Traffic3();
    f.setSize(450, 210);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter () {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
} // slutt main
```

En annen måte å
sørge for at programmet
terminerer normalt

Nytt
lys

Max
tre
lys

```
public void actionPerformed(ActionEvent e) {
    Object event = e.getSource();
    if (event == newSetButton) {
        lights[nLights] = new SetOfLights(area,  

lightsPosition);
        lights[nLights].start();
        lightsPosition += lightsWidth;
        nLights++;
        if (nLights == 3)
            newSetButton.setEnabled(false);
    } else if (event == walkButton) {
        lights[setWanted].walk = true;
    } else if (event == duration) {
        lights[setWanted].time[light]=  

        Integer.parseInt(duration.getText());
    } else if (event == closeButton) {
        for (int i = 0; i < nLights; i++)
            lights[i].stop();
        setVisible(false);
        dispose();
        System.exit(0);
    }
} // slutt actionPerformed
```

Trådene

```
class SetOfLights extends Thread {  
    private int red = 0;  
    private JPanel area;  
    private int x;  
    private int light;  
    boolean walk = false;  
    boolean walkOn = false;  
    int time [] = {500, 500, 500, 500};  
}
```

↑ Tråden(e)s
tilstand

Konstruere en tråd:

```
public SetOfLights(JPanel c, int x) {  
    area = c;  
    this.x = x;  
}
```



UNIVERSITETET
I OSLO

Selve tråden(e)

```
public void run() {  
    while (true) {  
        for (int light = 0; light < 3; light++) {  
            if (light == red & walk) {  
                walkOn = false;  
                for (int i = 0; i < 11; i++) {  
                    draw(light);  
                    try { sleep(time[3]); }  
                    catch (InterruptedException e) { }  
                    walkOn = !walkOn;  
                }  
                walk = false;  
            } else {  
                draw(light);  
                try { sleep(time[light]); }  
                catch (InterruptedException e) { }  
            }  
        }  
    } // slutt while (true)  
} // slutt run
```

```
void draw(int light) {
    Graphics g = area.getGraphics();
    g.setColor(Color.black);
    g.drawOval(x-8, 10, 30, 68);
    g.setColor(Color.cyan);
    g.fillRect(x-90,10,70,100);
    g.setColor(Color.black);
    g.drawString(""+time[0], x-70, 28);
    g.drawString(""+time[2], x-70, 48);
    g.drawString(""+time[1], x-70, 68);
    g.drawString(""+time[3], x-70, 98);
```

```
switch (light) {
```

```
case 0:
```

```
    g.setColor(Color.red);
    g.fillOval(x, 15, 15, 15);
    g.setColor(Color.lightGray);
    g.fillOval(x, 35, 15, 15);
    g.fillOval(x, 55, 15, 15);
    break;
```

```
case 1:
```

```
    g.setColor(Color.green);
    g.fillOval(x, 55, 15, 15);
    g.setColor(Color.lightGray);
    g.fillOval(x, 15, 15, 15);
    g.fillOval(x, 35, 15, 15);
    break;
```

Grønn

Gul

**Her tegner
tråden ut
sitt lys**

Rød

**variablen
light
styrer
ut-
tegningen**

```
case 2:
```

```
    g.setColor(Color.yellow);
    g.fillOval(x, 35, 15, 15);
    g.setColor(Color.lightGray);
    g.fillOval(x, 15, 15, 15);
    g.fillOval(x, 55, 15, 15);
    break;
```

```
} // slutt case
```

```
if (light == red & walk) {
    if (walkOn)
        g.setColor(Color.green);
    else
        g.setColor(Color.white);
    g.fillOval(x+1, 85, 14, 14);
} else {
    g.setColor(Color.black);
    g.drawOval(x, 85, 15, 15);
}
```

```
} // slutt draw
```

```
} // slutt tråd-klassen SetOfLights
```

```
} // slutt class Traffic3
```



NYTT OG VIKTIG: Oppdateringsproblemet:

Om og å passe på at tråder samarbeider riktig

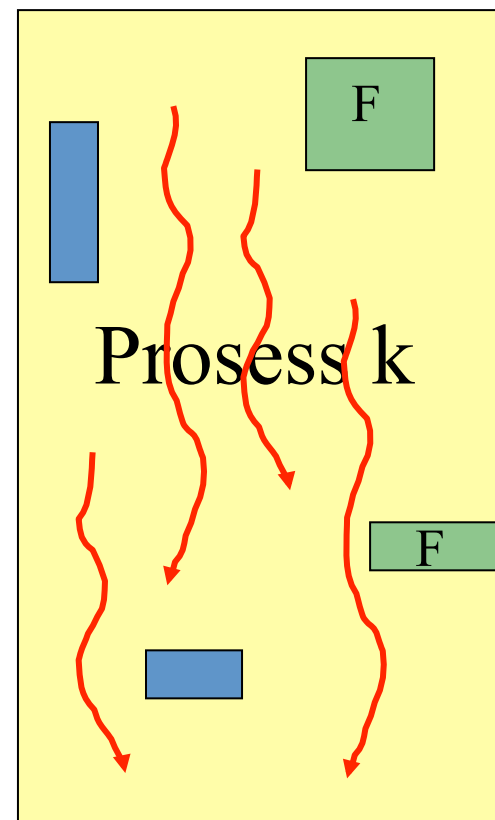
- **Samtidig oppdatering av felles data**
- Løsningen: Kritiske regioner med synkroniserte metoder.
- Først: Mer intro og aller enkleste eksempel
- Så et større program (Kokk og servitør)
- Ekstrastoff:
 - Et program som feiler: Flysalg.java
 - Deretter et program som er riktig: Flysalg2.java



Felles data

Felles data (grønne felt) må vanligvis bare aksesseres (lese og skrives i) av en tråd om gangen. Hvis ikke blir det kluss i dataene.

På figuren vår er det to områder vi har problemer med (dvs. at to eller flere tråder kan risikere å manipulere data i disse områdene samtidig). Disse to områdene er markert med F (for Felles). Et slikt område kalles en monitor.

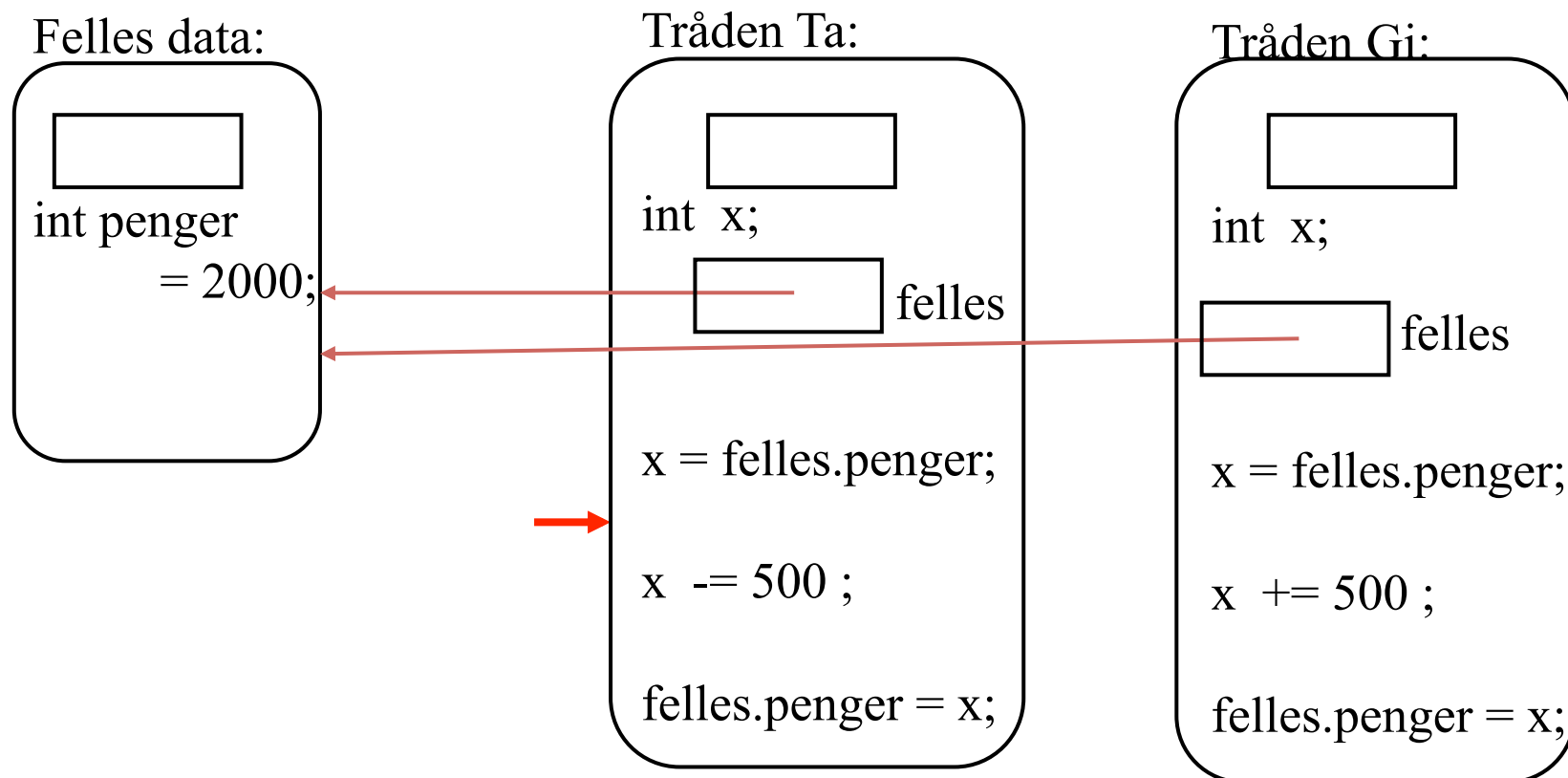


De andre to områdene inneholder data som vi vet at bare en tråd om gangen bruker. Slikt resonement er imidlertid farlige og ofte feilaktig, derfor er det ofte best å beskytte alle delte data som om de kan bli oppdatert samtidig.

Eneste unntak er data som bare leses av alle (“immutable”).



Enkelt eksempel på at to tråder kan ødelegge felles data.



La oss se hva som skjer hvis tråden Ta først utføres litt, og stopper opp ved pilen. Deretter overtar tråden Gi, og hele denne tråden utføres ferdig. Til slutt utføres resten av tråden Ta.



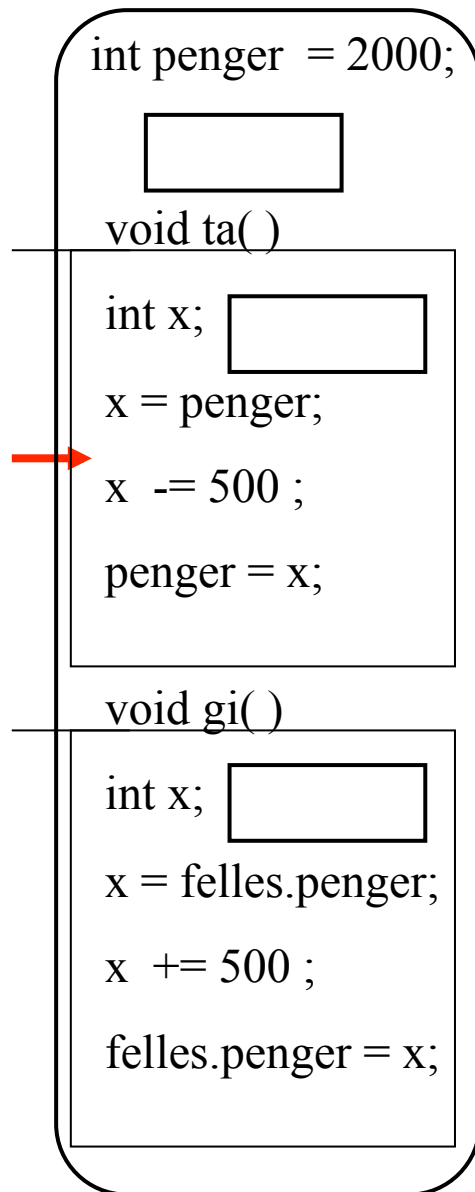
Hva med
ekte
parallellitet ?



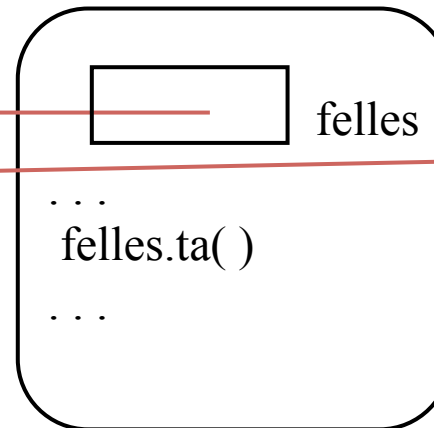
UNIVERSITETET
I OSLO

Samme – med metoder

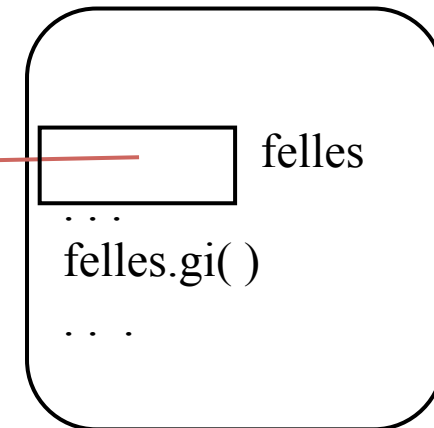
Felles data:



Tråden Ta:



Tråden Gi:



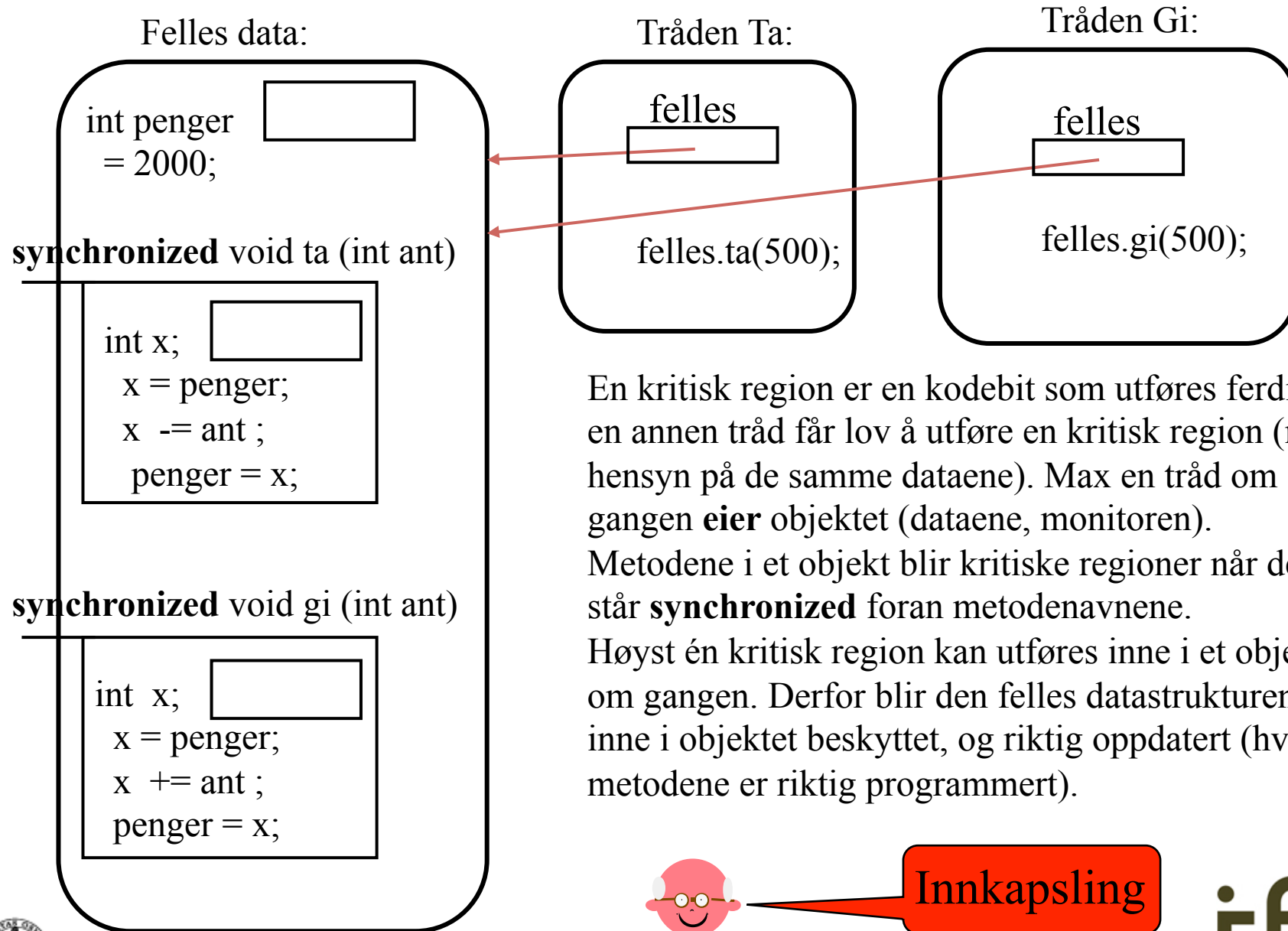
Trådene utfører metodene ta og gi som om koden inne i metodene var en del av trådenes kode.

På samme måte som på forrige side kan vi se hva som skjer hvis tråden Ta først utfører metoden ta litt, og så stopper opp ved pilen. Deretter overtar tråden Gi, og hele denne tråden utføres ferdig (og utfører hele metoden gi).

Til slutt utføres resten av tråden Ta (metoden ta).



Vi ordner dette med kritiske regioner / synkroniserte metoder.



En kritisk region er en kodebit som utføres ferdig før en annen tråd får lov å utføre en kritisk region (med hensyn på de samme dataene). Max en tråd om gangen **eier** objektet (dataene, monitoren).

Metodene i et objekt blir kritiske regioner når det står **synchronized** foran metodenavnene.

Høyst én kritisk region kan utføres inne i et objekt om gangen. Derfor blir den felles datastrukturen inne i objektet beskyttet, og riktig oppdatert (hvis metodene er riktig programmert).



Innkapsling



UNIVERSITETET
I OSLO

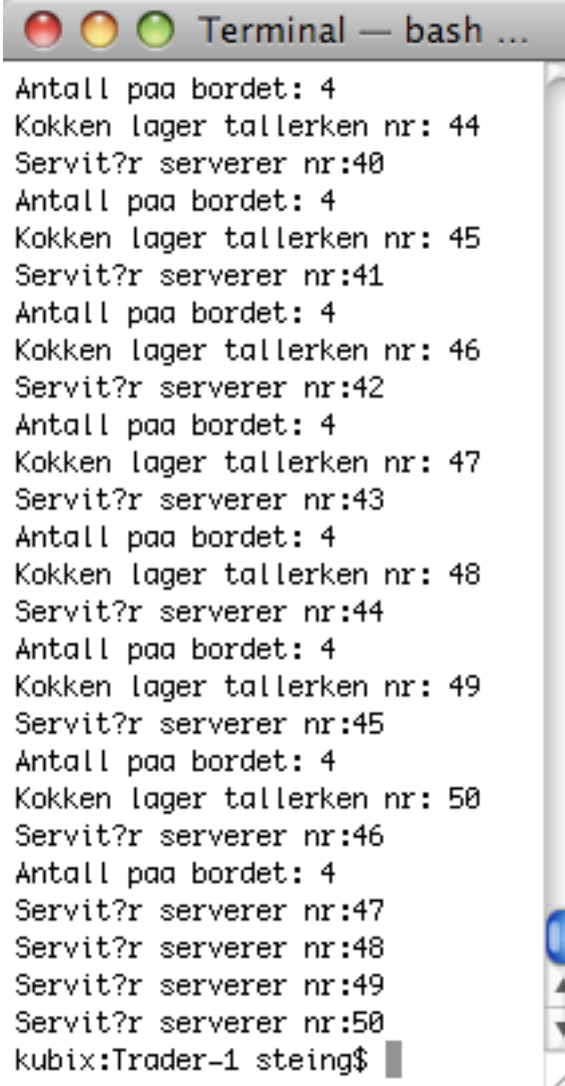
Mer om synkroniserte metoder / kritiske regioner / monitorer

- Alle tråder har felles adresserom
- Hvis flere tråder forsøker samtidig å
 - først lese en variabel 'a'
 - så oppdatere (endre) 'a' basert på den verdien den leste, kan det gå galt (jfr. eksemplet side 30-32)
 - FORDI:
 - En tråd X kan først lese verdien av 'a', og så bli avbrutt.
 - Så kan andre tråder Y, Z komme inn og endre 'a'
 - Når X igjen får kjøre, vil den oppdatere 'a' ut fra 'a' s gamle verdi, og ikke det den nå er
- Vi må beskytte slik lesing og etterfølgende skriving av samme data
- Setter vi **synchronized** foran metodene, vil vi sikre at:
 - Høyst en tråd er inne i noen av de synkroniserte metodene i dette objektet samtidig. Dette er monitorens (objektets) **eier**.
 - En tråd får slippe til, de andre trådene må vente
 - Når den ene er ferdig, slipper de som venter til (en etter en)
 - Alle data i objektet blir skrevet skikkelig ned i variablene i primærlageret (RAM)



Et større litt større eksempel – kokk og servitør

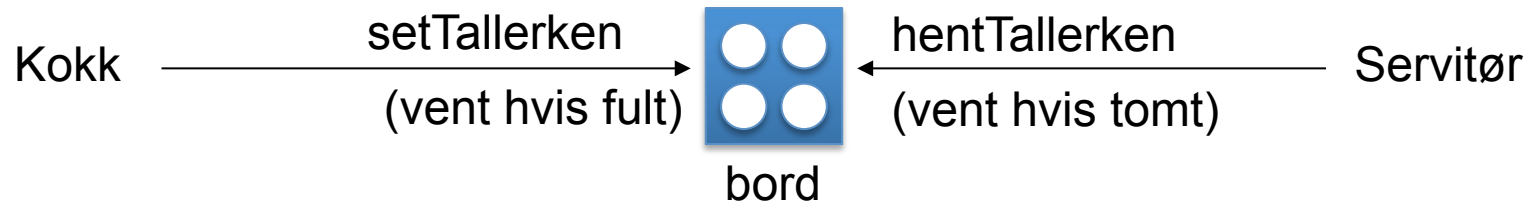
- Kokken lager mat og setter en og en tallerken på et bord
- Servitøren tar en og en tallerken fra bordet og serverer
- Kokken må ikke sette mer enn BORD_KAPASITET tallerkener på bordet (maten blir kald)
- Servitøren kan selvsagt ikke servere mat som ikke er laget (bordet er tomt)
- Her: en kokk og en servitør – Oppgave: lag flere av hver.



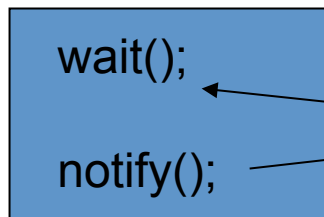
```
Terminal — bash ...
Antall paa bordet: 4
Kokken lager tallerken nr: 44
Servit?r serverer nr:40
Antall paa bordet: 4
Kokken lager tallerken nr: 45
Servit?r serverer nr:41
Antall paa bordet: 4
Kokken lager tallerken nr: 46
Servit?r serverer nr:42
Antall paa bordet: 4
Kokken lager tallerken nr: 47
Servit?r serverer nr:43
Antall paa bordet: 4
Kokken lager tallerken nr: 48
Servit?r serverer nr:44
Antall paa bordet: 4
Kokken lager tallerken nr: 49
Servit?r serverer nr:45
Antall paa bordet: 4
Kokken lager tallerken nr: 50
Servit?r serverer nr:46
Antall paa bordet: 4
Servit?r serverer nr:47
Servit?r serverer nr:48
Servit?r serverer nr:49
Servit?r serverer nr:50
kubix:Trader-1 steing$
```

wait(); notify();

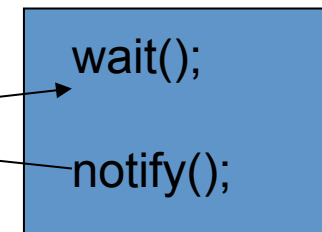
- Kokken må vente når det allerede er fire tallerkener på bordet
- Servitøren må vente når det ikke er laget noe mat (ingen tallerkener på bordet)
- Kokken må starte opp kelneren igjen når han har satt tallerken nr. 1 på bordet (eller alltid når han har satt en tallerken på bordet ?)
- Servitøren må starte opp kokken igjen når han tar tallerken nr. 4 fra bordet (eller alltid når han tar en tallerken fra bordet ?)



Kokk:



Servitør:



Tror du at programmet nedenfor er feil ?

Hvis ja: Påvis en kjøring som gir en feilsituasjon

Tror du dette programmet er riktig ?

Hvis ja: Begrunn hvorfor det er riktig ?

```
public class RestaurantS {  
    RestaurantS(String[] args) {  
        int antall = Integer.parseInt(args[0]); // antall porsjoner  
        FellesBord bord = new FellesBord();  
        Kokk kokk = new Kokk(bord, antall);  
        kokk.start();  
        Servitor servitor = new Servitor(bord, antall);  
        servitor.start();  
    }  
  
    public static void main(String[] args) {  
        new RestaurantS(args);  
    }  
}
```

Dette er en versjon av tilsvarende program i “Rett på Java”,
men dette programmet er bedre (fordi bordet er skilt ut som en monitor)

```

class FellesBord {    // en monitor
    private int antallPaBordet = 0;    // invarianten gjelder
    private final int BORD_KAPASITET = 4;
    /* Invariant: 0 <= antallPaBordet <= BORD_KAPASITET */

    synchronized void settTallerken() {
        while (antallPaBordet >= BORD_KAPASITET) {
            /* Så lenge det allerede er BORD_KAPASITET tallerkner
               på bordet er det ikke lov å sette på flere. */
            try { wait();
            } catch (InterruptedException e) {}
        } // Nå er antallPaBordet < BORD_KAPASITET
        antallPaBordet++; // bevarer invarianten
        System.out.println("Antall på bordet: " + antallPaBordet);
        notify(); // Si fra til den som henter tallerkener
    }

    synchronized void hentTallerken() {
        while (antallPaBordet == 0) {
            /* Så lenge det ikke er noen tallerkener på
               bordet er det ikke lov å ta en */
            try { wait();
            } catch (InterruptedException e) {}
        } // Nå er antallPaBordet > 0
        antallPaBordet--; // bevarer invarianten
        notify(); // si fra til den som setter på tallerkener
    }
}

```



```

class Kokk extends Thread {
    private FellesBord bord;
    private final int ANTALL;
    private int laget = 0;
    Kokk(FellesBord bord, int ant) {
        this.bord = bord;  ANTALL = ant;
    }
    public void run() {
        while(ANTALL != laget) {
            laget ++;
            System.out.println("Kokken lager tallerken nr: " + laget);
            bord.settTallerken();  // lag og lever tallerken
            try { sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }  // Kokken er ferdig
    }
}

class Servitor extends Thread {
    private FellesBord bord;
    private final int ANTALL;
    private int servert = 0;
    Servitor(FellesBord bord, int ant) {
        this.bord = bord;  ANTALL = ant;
    }
    public void run() {
        while (ANTALL != servert) {
            bord.hentTallerken(); /* hent tallerken og server */
            servert++;
            System.out.println("Servitør serverer nr:" + servert);
            try { sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}
        }  // Servitøren er ferdig
    }
}

```



```

class Flysalg {
    static int antSeter = 7;
    int antSolgt = 0;

    public static void main(String[] args) {
        System.out.println("Skal selge::" + antSeter
                           + " flyseter med 2 selgere");
        Flysalg fly= new Flysalg();
        Selger s1 = new Selger ("Stein ", fly);
        s1.start();
        Selger s2 = new Selger("Michael", fly);
        s2.start();
    }

    int antIgjen () {
        return antSeter;
    }

    void selg(int ant ) {
        antSeter -= ant;
        antSolgt +=ant;
        System.out.println(" Salg: ant igjen:"
                           + antSeter +", ant solgt: " + antSolgt);
    }
}

```

Ekstrastoff:
Eksempel som feiler:

```

class Selger extends Thread {
    Flysalg fly;
    String navn;

    Selger(String navn, Flysalg fly) {
        this.navn = navn;
        this.fly = fly;
    }
    public void run() {
        while (fly.antIgjen() > 0 ) {
            int ønske = (int) (Math.random() * fly.antIgjen() + 1);
            System.out.println( navn+ " vil selge:" + ønske);
            try {
                // vent litt
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}

            fly.selg(ønske);

            try {
                sleep((long) 2000); // vent 2 sek
            } catch (InterruptedException e) {}
        }
        // Selgere ferdig
    }
}

```

```

Skal selge::7 flyseter med 2 selgere
Stein  vil selge:2
Michael vil selge:4
Salg: ant igjen:3, ant solgt: 4
Salg: ant igjen:1, ant solgt: 6
Stein  vil selge:1
Michael vil selge:1
Salg: ant igjen: 0, ant solgt: 7
Salg: ant igjen: -1, ant solgt: 8
Press any key to continue . . .

```


Bedre:

```
class Flysalg2 {
    static int antSeter = 7 ;
    int antSolgt=0;

    public static void main(String[] args) {
        System.out.println("Skal selge::" + antSeter
                           + " flyseter med 2 selgere");
        Flysalg2 fly= new Flysalg2();
        Selger s1 = new Selger ("Stein ", fly);
        s1.start();
        Selger s2 = new Selger("Michael", fly);
        s2.start();
    }

    int antIgjen () {
        return antSeter;
    }

    synchronized boolean prøvSalg(int ant) {
        if (antSeter >= ant ) {
            antSeter -= ant;
            antSolgt += ant;
            System.out.println(" Salg: ant igjen:" + antSeter +
                               ", ant solgt: " + antSolgt);

            return true;
        } else {
            return false;
        }
    }
}
```

```

class Selger extends Thread {
    Flysalg fly;
    String navn;

    Selger(String navn, Flysalg fly) {
        this.navn = navn;
        this.fly = fly;
    }
    public void run() {
        while (fly.antIgjen() > 0 ) {
            int ønske = (int) (Math.random() * fly.antIgjen() +1);
            System.out.println( navn+ " vil selge:" + ønske);
            try {
                // vent litt
                sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {}

            if (fly.prøvSalg(ønske))
                System.out.println("Salget OK for:" + navn);
            else
                System.out.println("Allerede solgt for " + navn);

            try {
                sleep((long) 2000); // vent 2 sek
            } catch (InterruptedException e) {}
        }
        // Selgere ferdig
    }
}

```

```

Skal selge::7 flyseter med 2 selgere
Stein   vil selge:6
Michael vil selge:5
  Salg:5, ant igjen:2, ant solgt: 5
Salget OK for:Michael
Allerede solgt for Stein
Michael vil selge:2
Stein   vil selge:2
  Salg:2, ant igjen:0, ant solgt: 7
Salget OK for:Stein
Allerede solgt for Michael
Press any key to continue . . .

```