Praktiske øvingsoppgaver

Her finnes tekstene til de fire praksisøvingene (P1-P4). Hver av disse øvingene er omtalt ut fra følgende fem tema: Overordnet formål, detaljert beskrivelse, nyttige tips, tilgjengelig delløsning og eksakt oppgave.

Disse øvingene dekker henholdsvis lærebokas kapitler 5 (P1), 5 (P2), 9 (P3) og 8 & 12 (P4). Øvingene P1 og P4 er frivillige, mens øvingene P2 og P3 er obligatoriske.

Lærebokas kapitler 6 & 7 og 10 & 11 dekkes av de frivillige teoriøvingene – se disse.

P1 – Gjensidig utelukkelse – Kap. 5

Overordnet formål

Her skal du studere et tilfelle der gjensidig utelukkelse (mutual exclusion) er nødvendig for å få korrekt oppførsel i et enkelt program. Du må selv identifisere koden som har behov for gjensidig utelukkelse, den såkalte kritiske koden. Du må kunne forklare hvorfor koden er kritisk og vise hvordan manglende gjensidig utelukkelse kan føre til feil. Oppgaven omfatter problemstillinger tilsvarende lærebokas kapittel 5.

Beskrivelse av oppgaven

Du skal studere et enkelt program bestående av en bank og en rekke kurerer. Kurerene henter penger hos butikker og setter pengene inn i banken. Banken tar i mot penger i nattsafen, og når det er for mye penger i nattsafen overføres alt som ligger i nattsafen til hvelvet. I løpet av en kjøring sender banken ut en rekke kurerer, som så alle gjør en rekke innskudd i banken. Ved slutten av kjøringen viser det seg imidlertid at det noen ganger mangler penger i banken i forhold til hvor mye kurerene hevder at de har satt inn. Andre ganger er det til og med for mye penger i banken. Din oppgave er å finne ut hvorfor og løse problemet.

Tilgjengelig delløsning

Her er koden utdelt, og du skal analysere den. Du vil måtte gjøre minimalt med endringer for å få koden til å fungere korrekt.

Nyttige tips

Det kan være greit å lese informasjonen om tråder og synkronisering som er gitt i seksjon 3.2. Du bør også lese første del av kapittel 5 i boka. Gjensidig utelukkelse implementeres i Java ved hjelp av nøkkelordet synchronized.

Eksakt oppgave

- Analyser koden og finn ut hvorfor den ikke fungerer. Rett opp feilen(e).
- Sammenlign kjøretiden til den rettede løsningen og den opprinnelige koden. Forklar forskjellen.
- Vis eksempler på situasjoner der den feilaktige koden kan føre til for mye penger i banken og for lite penger i banken.
- Lever inn rettet kode og ovennevnte eksempler.

Utdelt kode

```
public class Bank {
   private int nightSafeAmount;
   private int vaultAmount;
   private Courier[] couriers;
   private long startTime;
   private int nofActiveCouriers;

   /**
    * Creates a new bank, with a number of couriers, and
    * starts all the couriers.
    */
   public Bank(int nofCouriers, int nofVisits) {
      startTime = System.currentTimeMillis();
      couriers = new Courier[nofCouriers];
      nofActiveCouriers = nofCouriers;
      for(int i = 0; i < nofCouriers; i++) {
         couriers[i] = new Courier(nofVisits, this);
         couriers[i].start();
      }
   }

   /**
    * Deposit a given amount of money. The money is put in the
    * night safe. If there is too much money in the night safe,
    * all the money is transferred to the vault.
    */
   public void depositMoney(int amount) {
      nightSafeAmount += amount;
      if(nightSafeAmount > 200) {
          // Transfer the money in the night safe to the vault:
         vaultAmount += nightSafeAmount;
         nightSafeAmount = 0;
      }
   }

   /**
    * Called by a courier when he is done with his deposits.
    */
   public void courierDone() {
      nofActiveCouriers--;
      if(nofActiveCouriers == 0) {
         System.out.println("All couriers are done.");
          // Check if the money deposited equals the money in the bank:
         int moneyInBank = nightSafeAmount + vaultAmount;
         int moneyDeposited = calculateMoneyDeposited();
         System.out.println("Money in the bank: "+moneyInBank);
         System.out.println("Total money deposited: "+moneyDeposited);
         System.out.println("Discrepancy: "+(moneyDeposited-moneyInBank));
         System.out.println("Elapsed time: "+(System.currentTimeMillis()-startTime)+
            " milliseconds.");
      }
   }
```

```
    /**
     * Sum up all the reported deposits to find out how much
     * the couriers claim that they have deposited.
     */
    public int calculateMoneyDeposited() {
        int result = 0;
        for(int i = 0; i < couriers.length; i++)
            result += couriers[i].getAmountDeposited();
        return result;
    }

    /**
     * Reads the number of couriers and the number of visits to the
     * bank for each courier from the command line parameters, and
     * starts the simulation.
     */
    public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println("Usage: java Bank <number of couriers> <number of visits>");
            System.exit(0);
        }
        int nofCouriers = new Integer(args[0]).intValue();
        int nofVisits = new Integer(args[1]).intValue();
        System.out.println("Starting simulation of "+nofCouriers+" couriers with "+nofVisits+
            " deposits each.");
        Bank b = new Bank(nofCouriers, nofVisits);
      }
}

public class Courier extends Thread {
    private Bank bank;
    private int nofVisits;
    private int moneyDeposited;

    public Courier(int nofVisits, Bank bank) {
        this.nofVisits = nofVisits;
        this.bank = bank;
    }

    /**
     * Visit the bank nofVisits times, and each time
     * deposit between 50 and 149 dollars.
     * Keep records of how much we have deposited, as the
     * bank has been known to keep faulty records.
     */
    public void run () {
        for(int i = 0; i < nofVisits; i++) {
            int sum = 50+(int)(Math.random()*100);
            bank.depositMoney(sum);
            moneyDeposited += sum;
        }
        bank.courierDone();
    }

    public int getAmountDeposited() {
        return moneyDeposited;
    }
}
```

P2 – Synkronisering av tråder – Kap. 5

Overordnet formål

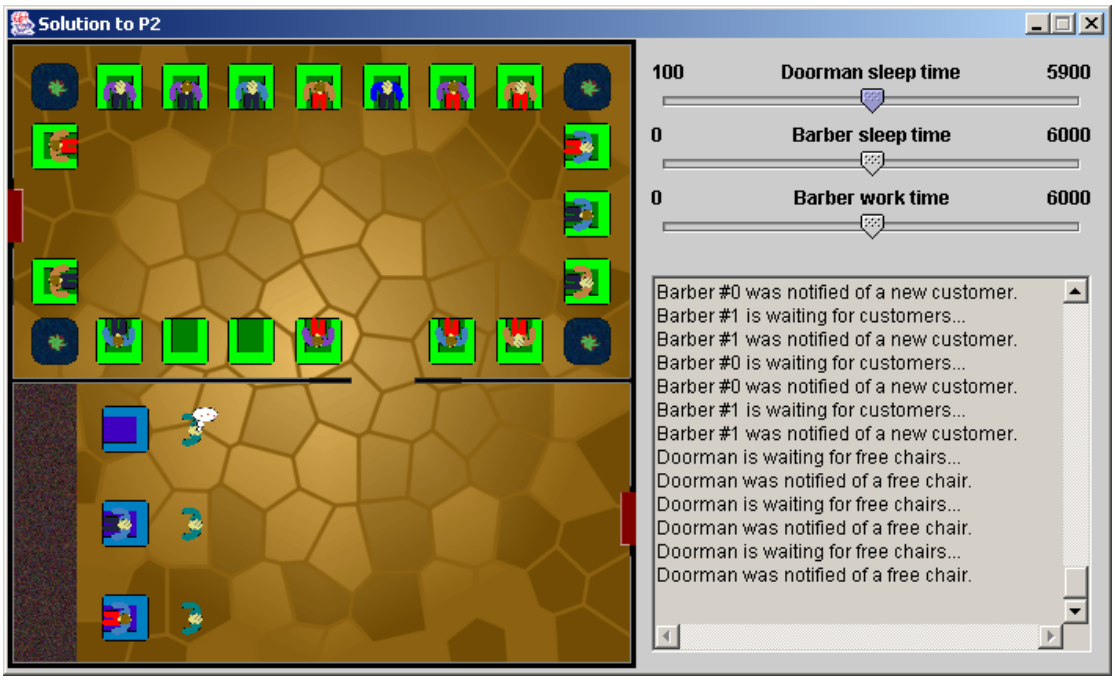
Her skal du implementere en variant av frisørsalongproblemet beskrevet i boka. Du skal bruke en produsent/konsument-modell. Produsent og konsument synkroniseres ved hjelp av metodene wait() og notify(), og gjensidig utelukkelse implementeres via Javas monitorkonsept; dvs. synchronized funksjoner. Oppgaven omfatter problemstillinger tilsvarende lærebokas kapittel 5.

Beskrivelse av oppgaven

Produsenten din, en innkaster ved en frisørsalong, utgjøres av en tråd som periodisk prøver å føye en ny person til en buffer med stoler i frisørsalongen. Med periodisk menes at produsenten sover en viss tid mellom hver gang han prøver å slippe inn en ny person.
Produsenten og konsumenten deler en sirkulær buffer med stoler. Produsenten plasserer en person i neste tilgjengelige stol, mens konsumenten alltid tar den personen som har ventet lengst.
Konsumentene dine er frisører, og frisørsalongen kan ha opp til tre frisører arbeidende på en gang. Hver frisør utgjør sin egen tråd. En frisering tar en viss tid. Mellom hver frisering dagdrømmer frisøren i tilfeldig lang tid innen gitte rammer, før han er klar til å klippe en ny kunde.

Tilgjengelig delløsning

Du vil få utdelt en GUI som gjør det lettere å visualisere hvordan synkroniseringen fungerer. Et skjermbilde fra GUI'en er vist under. Stolene øverst på bildet visualiserer den sirkulære bufferen. Nederst ses de tre frisørene. Tankeboblen betyr at frisørene dagdrømmer. Hvis frisøren ikke har tankeboble og ikke har en kunde i stolen, venter han på at det skal komme kunder. Til høyre er det et tekstområde der dere kan skrive ut ytterligere informasjon. Det er også tre slidere som stiller på hvor lenge de forskjellige aktørene sover og hvor lang tid en klipp tar. Innkasteren er ikke synlig men gjør seg gjeldende ved at det stadig kommer nye kunder.



Det er syv metoder i GUI'en som dere kan kalle på passende tidspunkter for å holde GUI'en oppdatert. Disse metodene er definert i grensesnittet Gui, og kan brukes av klassen Doorman, CustomerQueue og Barber til å oppdatere GUI'en etter hvert som ting skjer.
Les kommentarene i den utdelte koden for å se nøyaktig hva disse metodene gjør. I tillegg bør dere lese metoden startSimulation() i klassen BarbershopGui, som starter opp simuleringen. For at programmet skal kompilere og kjøre er det skissert et rammeverk til klassene klassene Doorman, Barber og CustomerQueue. Det er disse klassene dere må utvide for å få til en ferdig løsning.
Det er tre variabler i klassen Globals som angir hvor lenge de forskjellige aktørene skal sove og jobbe. Når sliderene i GUI'en skyves på, vil disse variablene endres. Referer derfor alltid til disse variablene når dere skal

avgjøre hvor lenge en produsent eller konsument skal sove. Soving kan gjøres med metodekallet Thread.sleep(). For å gi en litt mer spennende kjøring kan dere legge inn tilfeldige variasjoner i hvor lang tid aktørene sover, ved hjelp av metoden Math.random().

Nyttige tips

Her vil du måtte bruke den informasjonen om tråder og synkronisering som er gitt i seksjon 3.2.
Du bør ikke fysisk flytte kunder fra stol til stol etter som de rykker framover i køen (dette ville heller ikke være normalt i en vanlig frisørsalong). Bruk heller bufferpekere til å holde styr på begynnelsen og slutten av køen.

Når du skal bestemme hvor lenge innkasteren eller frisøren sover har du behov for å generere tilfeldige tall. En enkel måte å generere et tilfeldig tall fra min til max på er:
int r = min+(int)(Math.random()*(max-min+1));

Rammeverket til øvinga er ferdig utdelt. Dere må lage logikken. Klassene som må gjøres ferdige er: Doorman, CustomerQueue og Barber.

Eksakt oppgave

Følgende forventes som resultat av denne øvingen:

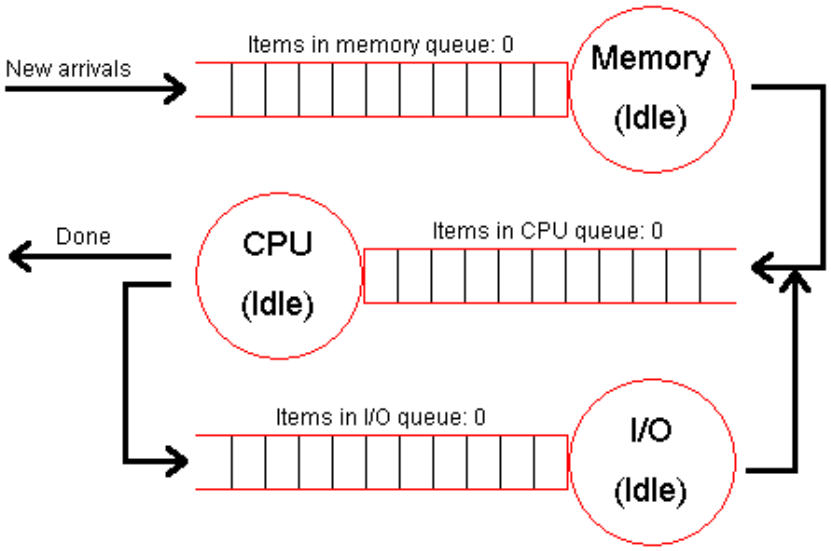
- Innlevering av kildekode til programmet
- En tekstfil med kommentarer som kort forklarer hvilke java-mekanismer du har brukt til å implementere produsent/konsument-modellen.
- Demonstrasjon av programmet for stipendiat / undervisningsassistent / studentassistent.

P3 – Tidsstyring av prosesser – Kap. 9

Overordnet formål

Her skal dere implementere en form for tidsstyring av prosesser ved å simulere bruk av Round-Robin algoritmen i et forenklet datamaskinsystem. Oppgaven omfatter problemstillinger tilsvarende lærebokas kapittel 9.

Betrakt en enkel datamaskin med et enkelt CPU, et lite lager samt en I/O enhet. Datamaskinen behandler prosesser som utfører batchjobber med statisk minnebehov. Det finnes bare en type lager (minne), altså ingen form for virtuelt minne. Datamaskinen er modellert som et køsystem som vist i figuren under, der prosesser ankommer med jevne mellomrom og forsvinner ut av systemet når de er ferdig med jobben sin.



Det er tre køer i systemet:

- En kø for prosesser som venter på å få tildelt minne.
- En kø for prosesser som venter på prosessortid.
- En kø for prosesser som venter på å få utføre I/O.

Utnyttelsen av CPU-kraften styres med Round Robin-algoritmen. De andre køene opereres etter FIFO-prinsippet. Jobbene og deres behov genereres tilfeldig.

Detaljert beskrivelse

Programmet som skal lages, må først etterspørre en del parametere fra brukeren for å kontrollere simuleringen. Dette gjelder:

- Tilgjengelig lagerstørrelse (for eksempel 2048 KB)
- Tidskvant for Round Robin (for eksempel 500 ms)
- Gjennomsnittlig I/O operasjontid (for eksempel 225 ms)

- Total simuleringstid (for eksempel 250.000 ms)
- Gjennomsnittlig tid mellom ankomst for nye jobber (5.000 ms)

For hver ny jobb som skal kjøres, må systemet videre generere tilfeldige parameterverdier for de tre karakteristikaene:

- Prosessens lagerbehov (for eksempel 100 KB). Lagerbehov skal variere mellom 100 KB og 25% av tilgjengelig lagerstørrelse.
- Prosessens eksekveringstid (for eksempel 5.000 ms). Eksekveringstid skal variere mellom 100 ms og 10.000 ms.

- Prosessens gjennomsnittlige eksekveringstid mellom I/O-behov (for eksempel 250 ms). Gjennomsnittlig tid mellom I/O-behov skal variere mellom 1% og 25% av prosessens eksekveringstid.

Den tilfeldige tallgeneratoren må også benyttes for følgende tre andre forhold: Tid fram til neste I/O-behov hver enkelt gang (variert rundt tilsvarende gjennomsnittlige tid), tid for neste I/O-behov hver enkelt gang (variert rundt tilsvarende gjennomsnittlige tid), samt tid til neste jobb-ankomst hver enkelt gang (variert rundt tilsvarende gjennomsnittlige tid).

En nyankommet jobb plasseres i lagerkøen. Her venter prosesser på tilstrekkelig lagerplass. Køen administreres ut fra FIFO prinsippet. En gitt prosess trenger ikke å vente på at dens lagerbehov kan dekkes som en enkelt minneblokk, bare på at dens lagerbehov kan dekkes totalt sett. Virtuelt lager eller swapping brukes ikke. Med andre ord trenger lageret kun å holde rede på hvor mye minne som er ledig til enhver tid.

Når en prosess får tildelt lagerplass, flyttes den direkte over i CPU-køen. Selve minnetildelingen tar ingen tid, så minne-enheten vil til enhver tid vises som ”Idle”, men minnekøen kan være lang. I CPU-køen venter prosesser på CPU-kraft. Denne køen administreres altså ut fra Round Robin prinsippet.

Når en prosess får tildelt CPU-en, beholder den kontrollen inntil ett av tre forhold skjer: Når tidskvantet spesifisert av RR-algoritmen er oppbrukt – hvorpå prosessen flyttes tilbake i CPU-køen, når et I/O-behov oppstår – hvorpå prosessen flyttes over til en egen I/O-kø, eller når prosessen er ferdig – hvorpå prosessens lagerplass tilbakeleveres og den forlater systemet.

En prosess som står i den egne I/O-køen, flyttes derfra og over til CPU-køen igjen etter tilhørende I/O-avslutning.

Når simuleringen er avsluttet – dvs. når simuleringstiden er ute selv om det fortsatt kan være prosesser som ikke er ferdige, skal programmet rapportere følgende:

- Antall ferdige prosesser
- Antall opprettede prosesser
- Antall prosessskifter (som skyldes oppbrukt tidskvant)
- Antall utførte I/O-operasjoner
- Gjennomsnittlig gjennomstrømning (ferdige prosesser per sekund)

- Total tid CPU har brukt på å prosessere
- Total tid CPU har vært ledig
- Prosentvis hvor mye tid CPU har brukt på å prosessere (utilisasjon)
- Prosentvis hvor lenge CPU har vært ledig
- Størst forekommende samt gjennomsnittlig lengde på alle køer
- Hvor mange ganger en ferdige prosess gjennomsnittlig har blitt plassert i hver enkelt kø

- Gjennomsnittlig tid tilbrakt i systemet per ferdige prosess
- Gjennomsnittlig tid ventende på tilstrekkelig lagerplass per ferdige prosess
- Gjennomsnittlig tid ventende på CPU-kraft per ferdige prosess
- Gjennomsnittlig tid brukt i CPU per ferdige prosess
- Gjennomsnittlig tid ventende på I/O-kapasitet per ferdige prosess
- Gjennomsnittlig tid brukt i I/O per ferdige prosess

Slik rapportering krever at ulike verdier tas vare på underveis i simuleringen. Resultatene avhenger mye av hvilke parametere som brukes i simuleringen, og dette åpner opp for ulik eksperimentering for å forbedre systemet.

Nyttige tips

Denne oppgaven er et eksempel på såkalt diskret hendelsessimulering. To hovedkomponenter vil være en klokke og en hendelseskø. Klokken vil holde oversikt over forløpt tid i simuleringen, mens hendelseskøen vil holde styr på framtidige hendelser i simuleringen. Eksempler på hendelser i denne sammenhengen er: Ankomst av en ny jobb, utløp av et tidskvant, oppstart av en I/O-operasjon, avslutning av en I/O-operasjon og terminering av en eksisterende jobb. Framtidige hendelser er vanligvis sortert på tid i hendelseskøen.

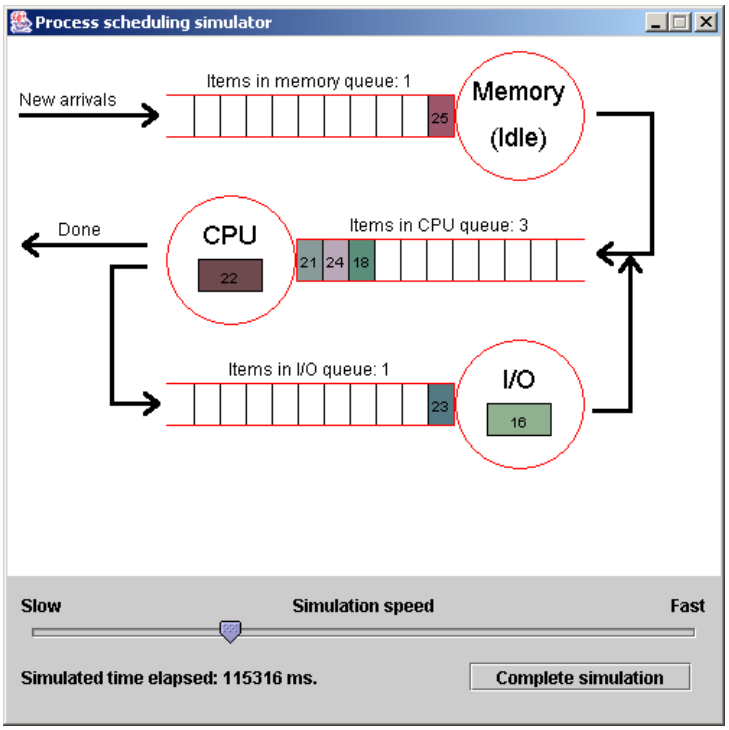
Hovedløkken i simuleringen tar hele tiden første hendelse ut av hendelseskøen, oppdaterer tiden tilsvarende den nye hendelsen og behandler hendelsen i seg selv. Således hopper tiden hele tiden framover i diskrete steg. Hvis flere hendelser skal skje på samme tid, vil de håndteres i rekkefølge uten at tiden forandrer seg. Behandlingen av en hendelse kan medføre at nye hendelser skapes og legges inn i hendelseskøen på riktig sted.

Som en liten pekepinn er kjøreutskriften fra et korrekt fungerende program vist under:

```
Please input system parameters:
Memory size (KB): 2048
Maximum uninterrupted cpu time for a process (ms): 500
Average I/O operation time (ms): 225
Simulation length (ms): 250000
Average time between process arrivals (ms): 5000
Simulating.....done.

Simulation statistics:

Number of completed processes:                                  39
Number of created processes:                                    48
Number of (forced) process switches:                            207
Number of processed I/O operations:                             753
Average throughput (processes per second):                      0.156

Total CPU time spent processing:                                199213 ms
Fraction of CPU time spent processing:                          79.6852%
Total CPU time spent waiting:                                   50787 ms
Fraction of CPU time spent waiting:                             20.3148%

Largest occuring memory queue length:                           2
Average memory queue length:                                    0.13544
Largest occuring cpu queue length:                              6
Average cpu queue length:                                       1.753956
Largest occuring I/O queue length:                              6
Average I/O queue length:                                       1.121616
Average # of times a process has been placed in memory queue: 1
Average # of times a process has been placed in cpu queue:    20.333334
Average # of times a process has been placed in I/O queue:    14.256411

Average time spent in system per process:                       22772 ms
Average time spent waiting for memory per process:              657 ms
Average time spent waiting for cpu per process:                 10104 ms
Average time spent processing per process:                      4465 ms
Average time spent waiting for I/O per process:                 4329 ms
Average time spent in I/O per process:                          3215 ms
```

Tilgjengelig delløsning

For å gjøre arbeidsomfanget mindre, vil vi her ferdigstille utvalgte moduler av det som etterspørres. Vi har laget en GUI som viser køsystemmodellen og hvor i systemet prosessene befinner seg. Et skjermbilde fra GUI'en er vist under.



GUI'en har et par metoder som må kalles på passende tidspunkter for at simulasjonens oppførsel skal vises i GUI'en. Disse metodene er deklarert i grensesnittet Gui, som deles ut. GUI'en er implementert av de utdelte klassene SimulationGui, PicturePanel, Resource og Constants. Når det gjelder selve simuleringen er klassene Event, EventQueue og Memory ferdig implementerte, mens oppstartsklassen Simulator og klassen Process er delvis implementert. Dere må ferdigstille resten av systemet, inkludert en klasse som simulerer CPU'en og en som simulerer I/O.

For å gjøre koden mest mulig ryddig anbefaler vi å samle flest mulig av variablene som fører statistikk med kjøringen på ett sted, gjerne i en separat klasse.

Kode for, utfyllende beskrivelse av og kommentar til de utleverte modulene vil bli gjort tilgjengelig dels via hjemmesiden, dels som papirkopier og dels i øvingstimene.

Eksakt oppgave

Følgende forventes som resultat av denne øvingen:

- Innlevering av kildekode til programmet – inkludert en kommentarseksjon som klart angir hvordan du vil eksperimentere med Round-Robin algoritmen for å forbedre systemet.
- Demonstrasjon av programmet for stipendiat / undervisningsassistent / studentassistent.
- Innlevering av utskrift fra kjøringen – inkluderende en resultatseksjon som detaljert angir effektene av Round-Robin eksperimenteringen. Prøv å identifisere nøkkelparametere som har stor innvirkning på systemets totale ytelse.

P4 – Virtuell filhåndtering – Kap. 8 & 12

Overordnet formål

Her skal dere implementere en mekanisme for virtuell filhåndtering; dvs. en måte å skape et virtuelt adresserom til å aksessere en fil med. En virtuell fil består av blokker som ligger lagret på disk og caches i minne. Denne blokkdelingen er imidlertid usynlig for applikasjoner, som kun ser et sammenhengende adresserom. Dette er analogt med virtuelt minne der minnet er organisert i sider, men prosesser ser et sammenhengende adresseområde. Filhåndteringsmekanismen skal således bruke et sett med sider og en tilhørende sideombyttingsalgoritme i forbindelse med filaksessering. Oppgaven omfatter problemstillinger tilsvarende lærebokas kapitler 8 og 12.

Sett fra brukerens synspunkt vil en fil bestå av en sekvens med bytes, adressert fra 0 og oppover. Enhver lese- / skriveaksess angir første byte og antall byte involvert. Hvis tilsvarende sider ikke er tilgjengelige, genereres en sidefeil som så håndteres for å bringe inn de nødvendige sidene.

Sett fra systemets side består en fil av et fast antall sider som angis når filen skapes. Som sideombyttingsalgoritmer skal FIFO, LRU og CLOCK brukes.

Detaljert beskrivelse

Den virtuelle filhåndtereren trenger bare å administrere en åpen fil om gangen. Funksjonaliteten som skal tilbys applikasjoner er spesifisert av følgende java-grensesnitt:

```
public interface VirtualFileSystem
{
    /**
     * Initializes the virtual file system.
     * @param pageSize  The size of pages (file blocks).
     * @param nofFrames The number of page frames in the system.
     */
    public void init(int pageSize, int nofFrames);

    /**
     * Creates a new file with the given name and size.
     * @param filename   A unique name identifying the file.
     * @param nofPages   The size of the file, in pages.
     */
    public void create(String filename, int nofPages);

    /**
     * Opens a file created by the create() method.
     * @param filename   The name of the file to be opened.
     */
    public void open(String filename);

    /**
     * Reads a sequence of bytes from the open file.
     * @param address   The position of the first byte to be read, relative
     *                  to the first byte in the file.
     * @param nofBytes  The size number of bytes to be read.
     * @param data      The buffer in which to place the read bytes.
     */
    public void read(int address, int nofBytes, byte[] data);
```

```
    /**
     * Writes a sequence of bytes to the open file.
     * @param address   The position of the first byte to be written, relative
     *                  to the first byte in the file.
     * @param data      A buffer of bytes to be written to the file.
     */
    public void write(int address, byte[] data);

    /**
     * Closes the file opened by the open() method.
     */
    public void close();

    /**
     * Specifies which page replacement algorithm to be used.
     *
     * @param chooser   An object, implementing the FrameChooser interface,
     *                  specifying the page replacement algorithm.
     */
    public void setFrameChooser(FrameChooser chooser);

    /**
     * Prints the contents of the frame table in a compact but readable form
     * to System.out. It suffices to list the indexes of the pages currently
     * in the table, and the corresponding clock bits.
     */
    public void printFrameTable();
}
```

Metoden setFrameChooser gir inn et objekt som implementerer grensesnittet FrameChooser vist under:

```
/**
 * An interface specifying methods for selecting which frame in
 * a frame table should be chosen when the need to replace a
 * page arises.
 */
public interface FrameChooser
{
    /**
     * Selects a replaceable frame from a full frame table.
     *
     * @param frameTable  The frame table.
     * @return            The frame whose page can be replaced.
     */
    public Frame chooseFrame(FrameTable frameTable);

    /**
     * Returns the number of page faults processed.
     *
     * @return  The number of times the chooseFrame() method has been invoked.
     */
    public int getNofPageFaultsProcessed();
```

```
    /**
     * Returns a string describing the algorithm used by this FrameChooser.
     *
     * @return  The name of the page replacement algorithm used.
     */
    public String getAlgorithm();
}
```

Implementasjoner av dette grensesnittet, samt klassen FrameTable, må dere lage selv. Dere må også lage en implementasjon av grensesnittet VirtualFileSystem. For å ta seg av selve skrivingen til og fra disk kan dere benytte dere av den utdelte klassen DiskManager.

Nyttige tips

Konkrete tips til hva de forskjellige metodene i implementasjonen av VirtualFileSystem må gjøre er listet opp under:

- init(): Denne metoden tar inn to parametere som spesifiserer hvor mange siderammer filsystemet har og hvor stor (antall bytes) hver side er. Sidestørrelsen må være lik blokkstørrelsen brukt i DiskManager. Konstruktoren bør opprette en rammetabell med siderammer, og en standard FrameChooser bør opprettes i tilfelle applikasjonen ikke spesifiserer en algoritme ved et senere kall til setFrameChooser().

- create(): Denne metoden kan benytte seg av create-metoden i klassen DiskManager.

- open(): Dere må opprette en sidetabell for filen. Størrelsen på denne avhenger av antall sider i filen. Initielt opptar filen ingen siderammer. Tilsvarende sider hentes etterhvert inn i siderammer når det oppstår sidefeil i.f.m. aksessering av filen. Hvis alle siderammene er opptatte når det oppstår en sidefeil må FrameChooser-objektet brukes til å velge en side som kan erstattes. Siden som erstattes må skrives ut til disk hvis den er endret.

- read(): Adresseområdet som skal leses må oversettes til sideaksessoperasjoner. For hver side som skal aksesseres må det sjekkes om siden ligger i rammetabellen. Hvis den ikke gjør det, oppstår det en sidefeil som må behandles før leseoperasjonen kan fortsette. Sideaksessoperasjoner kan utføres ved hjelp av metoder i DiskManager.

- write(): Skriving til fil fungerer veldig likt som lesing fra fil, og kan også generere sidefeil. Her bør dere utnytte de muligheter til gjenbruk som finnes. Modifiserte sider i rammetabellen skal ikke skrives ut til disk før filen lukkes, eller sidene blir erstattet.

- close(): Når en fil lukkes må sider i rammetabellen som er endret skrives til disk. Alle siderammene frigjøres, og sidetabellen slettes.

Hver sideramme skal altså inneholde et bit som angir om tilhørende side er endret eller ikke siden den ble innlest. Dette vil avgjøre om den må skrives tilbake til disk når den blir ombyttet med en annen side. Hver sideramme bør også inneholde et bit (eller noe lignende) som angir om tilhørende side er aksessert ”siden sist”.

Dere trenger åpenbart klasser for sider og siderammer, og implementasjoner av FrameChooser-grensesnittet som implementerer de forskjellige sideerstattingsalgoritmene. Det kan også være nyttig å lage egne klasser med tilhørende metoder for sidetabellen og rammetabellen.

Hvis dere forsøker å kompilere den utdelte koden, vil dere få en rekke ”cannot resolve symbol”-feilmeldinger. Disse indikerer noen klasser som må implementeres.

Tilgjengelig delløsning

Deler av løsningen er ferdig utdelt. Dette omfatter grensesnittene som skal implementeres, en klasse som tester ut filsystemet, og DiskManager-klassen. Kode for, utfyllende beskrivelse av og kommentar til disse modulene vil igjen bli gjort tilgjengelig dels via hjemmesiden, dels som papirkopier og dels i øvingstimene.

Eksakt oppgave

Følgende forventes som resultat av denne øvingen:

- Innlevering av kildekode til programmet
- Overordnet beskrivelse av hvordan programmet er bygd opp og fungerer
- Demonstrasjon av programmet for stipendiat / undervisningsassistent / studentassistent
- Innlevering av utskrift fra kjøringen, med diskusjon av de forskjellige sideombyttingsalgoritmenes fordeler og ulemper.