**DAT103 – Oblig 3**

**Oppg. 1**

**1.**

Operativsystem er noe de fleste datamaskiner har, dette for å koble sammen alle de ulike komponentene og få de til å jobbe sammen på en effektiv og organisert måte. Operativsystemet holder kontroll på og setter i gang lesing av data, endring av data og sending av data. Den fungerer som en communicator mellom de ulike delene, får de til å kommunisere og interprette og outputte det vi er interesserte i. I sammenheng med operativsystem kommer ofte «kernel» fram, dette er kjernen i systemet, og det mest grunneleggende i et operativsystem.

De mest kjente til dags dato er MacOS, Windows og Linux, der Linux ofte gir mest «frihet».

**2.**

Man kan prate om en prosess som en forandring, noe som går fra en ting til å bli en annen og kanskje ny ting. Det en systematisk handling der data blir prossesert og brukt for å gi ny data. Prosesser brukes ofte fordi man ønsker et resultat, og en prosess kan sees på en måte å komme seg fra nuet til en annen ønskelig framtid.

I datamaskiner er det mange prosesser, alle med sine hensikter. Det kan være en prosess for å hente data fra bruker og displaye det til bruker, eller omvendt, hente data fra datamaskinen og sende det ut av dataen (minnebrikke, cloud storage etc.). Prosesser finnes overalt rundt oss, og de kommer i ulike former og farger.

**3.**

En prosess kan inneholde en eller flere tråder, som oftest flere tråder. Prosessen er på en måte placeholderen som holder på de ulike trådene, og sette de i gang på et bestemt og planlagt punkt. Trådene kan kjøres hver for seg eller parallelt. Det er raskere å lage en ny tråd inni en prosess, enn å opprette en prosess. Ting skjer mer effektivt mellom tråder enn mellom prosesser, og kommunikasjon går raskere. Trådene er flyten av utførelsen inni en prosess, og deler opp de ulike tinga som skal skje for mer effektivitet i prosessen, mens prosessen er samlingen av alt dette. Trådene deler samme minne, mens ulike prosesser trenger ikke å gjøre dette.

4.

Hver fork() gir 2 child prosesser, unntatt den første, den har 1 parent prosess og 1 child prosess så totalt lager 4 forks 8 prosesser, der 7 er child prosesser.

**Oppg. 2**

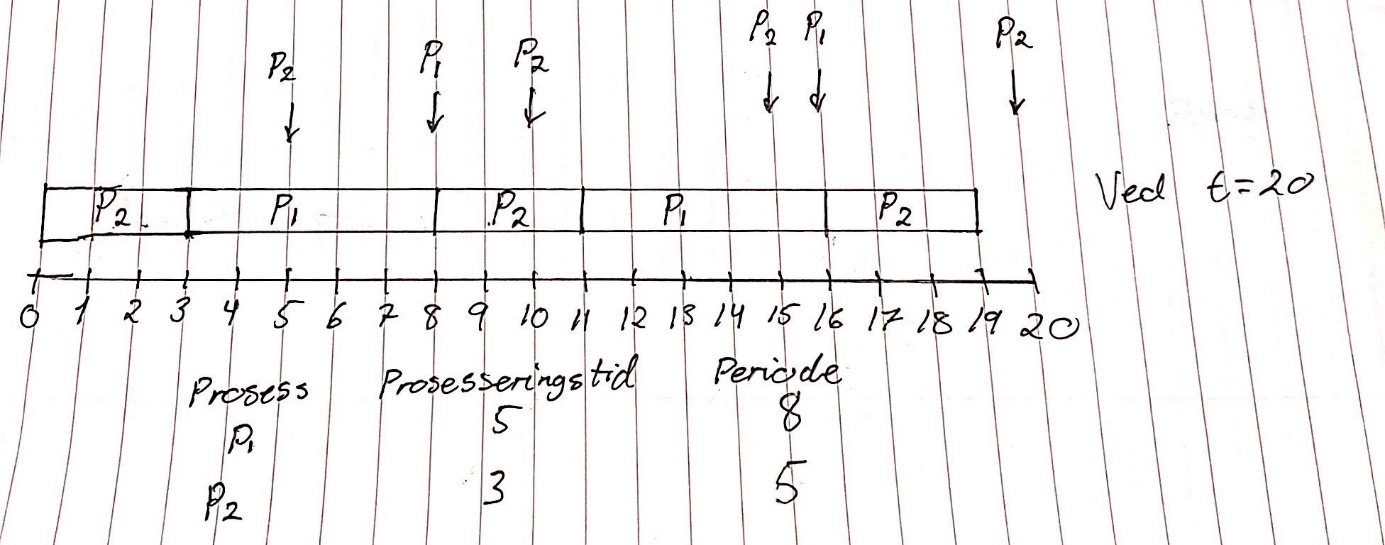
|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P1 | P3 |

1.

0 1 5 13 22

P0 starter med en gang siden det ikke er noen andre som venter. Så starter P1 og avbryter P0 fordi den har mindre utbruddstid, da har P0 8 ms igjen. Da går P1 i 4 ms. Så fortsetter P1 fordi den tiden den har igjen er mindre enn utbruddstiden til P2. P0 venter da 4 ms, P1 venter i 0 ms og P2 venter i 11 ms. Den gjennomsnittlige ventetiden blir da 5 ms.

2.



Ved t = 20 vil begge prosessene oppfylle alle tidsfristene.

**Oppg. 3**

**1.**

**a)**

En kritisk region er når 2 eller flere prosesser har tilgang til og brukes den samme kodesekvensen, eller når prosessene bruker de samme variablene eller ressursene for å oppnå ønsket resultat. Ordet kritisk blir brukt siden disse dataene trengs å synkroniseres for å passe på at det en viss konsistens og orden i prosessen. Det kan gis uventet errors og feil dersom to eller flere tråder prøver å jobbe på eller endre de samme variablene eller verdiene.

**b)**

Semaforer er noe som blir brukt til synkronisering og utelukkelse av de ulike prosessene i et operativsystem. Det brukes for å beskytte den kristiske region, som nevnt ovenfor. Semaforer kan bli sett på som beskyttende variabler som unngår venting.

**c)**

Kritisk region i Java.

import java.util.concurrent.Semaphore;

public class KritiskRegion {

static int *kritiskRegion* = 0;

public static void main(String[] args) throws InterruptedException {

Semaphore semaphore = new Semaphore(1);

Runnable runnable = new Runnable() {

public void run() {

try {

semaphore.acquire();

*kritiskRegion*++;

} catch (InterruptedException e) {

e.printStackTrace();

}

semaphore.release();

}

};

Thread thread1 = new Thread(runnable);

Thread thread2 = new Thread(runnable);

thread1.start();

thread2.start();

thread1.join();

thread2.join();

System.***out***.println(*kritiskRegion*);

}

}

**2.**

Løst på samme måte som boka:

public class Oppg2bok {

static BookSemaphore *sem1*;

static BookSemaphore *sem2*;

static int *counter* = 0;

public Oppg2bok() {

*sem1* = new BookSemaphore(1);

*sem2* = new BookSemaphore(1);

}

public static class Read implements Runnable {

public void run() {

try {

*sem1*.acquire();

*counter*++;

if (*counter* == 1) *sem2*.acquire();

*sem1*.release();

System.***out***.println("Reading " + Thread.*currentThread*().getName());

Thread.*sleep*(1000);

System.***out***.println("Done reading " + Thread.*currentThread*().getName());

*sem1*.acquire();

*counter*--;

if (*counter* == 0) *sem2*.release();

*sem1*.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static class Write implements Runnable{

public void run() {

try {

*sem2*.acquire();

System.***out***.println("Writing " + Thread.*currentThread*().getName());

Thread.*sleep*(1000);

System.***out***.println("Done writing " + Thread.*currentThread*().getName());

*sem2*.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main (String[] args) throws Exception {

Read read = new Read();

Write write = new Write();

Oppg2bok bok = new Oppg2bok();

new Thread(read, "thread1").start();

new Thread(write, "thread2").start();

new Thread(read, "thread3").start();

new Thread(read, "thread4").start();

new Thread(write, "thread5").start();

}

}

og har her definert vår egen BookSemaphore class:

public class BookSemaphore {

int value;

public BookSemaphore(int v) {

value = v;

}

public synchronized void acquire() throws InterruptedException {

if (value > 0) value--;

else {

this.wait();

value--;

}

}

public synchronized void release() throws InterruptedException {

this.value++;

if(value > 0) this.notify();

}

}

Løst med innebygd Semaphore class i Java:

import java.util.concurrent.Semaphore;

public class Oppg2java {

static Semaphore *sem1*;

static Semaphore *sem2*;

static int *counter* = 0;

public Oppg2java() {

*sem1* = new Semaphore(1);

*sem2* = new Semaphore(1);

}

public static class Read implements Runnable {

public void run() {

try {

*sem1*.acquire();

*counter*++;

if (*counter* == 1) *sem2*.acquire();

*sem1*.release();

System.***out***.println("reading " + Thread.*currentThread*().getName());

Thread.*sleep*(1000);

System.***out***.println("done reading " + Thread.*currentThread*().getName());

*sem1*.acquire();

*counter*--;

if (*counter* == 0) *sem2*.release();

*sem1*.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static class Write implements Runnable{

public void run() {

try {

*sem2*.acquire();

System.***out***.println("writing " + Thread.*currentThread*().getName());

Thread.*sleep*(1000);

System.***out***.println("done writing " + Thread.*currentThread*().getName());

*sem2*.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main (String[] args) throws Exception {

Read read = new Read();

Write write = new Write();

Oppg2java java = new Oppg2java();

new Thread(read, "thread1").start();

new Thread(write, "thread2").start();

new Thread(read, "thread3").start();

new Thread(read, "thread4").start();

new Thread(write, "thread5").start();

}

}

**3.**

Løst på samme måte som boka:

import java.util.concurrent.Semaphore;

import java.util.Random;

public class Oppg3bok {

static int *numPhilosophers* = 5;

static Fork[] *forks* = new Fork[*numPhilosophers*];

static Philosopher[] *philosophers* = new Philosopher[*numPhilosophers*];

public static class Philosopher extends Thread {

static int *num*;

static Fork *leftFork*;

static Fork *rightFork*;

Philosopher(int number, Fork left, Fork right){

*num* = number;

*leftFork* = left;

*rightFork* = right;

}

public void run(){

while (true){

if (*num* % 2 == 0){

if (*leftFork*.*free*()){

*leftFork*.*pickUp*();

System.***out***.println("Philosopher " + *num* + " picked up the left chopstick");

*rightFork*.*pickUp*();

System.***out***.println("Philosopher " + *num* + " picked up the right chopstick");

eat();

*leftFork*.*putDown*();

System.***out***.println("Philosopher " + *num* + " put down the left chopstick");

*rightFork*.*putDown*();

System.***out***.println("Philosopher " + *num* + " put down the right chopstick");

}

}

else {

if (*leftFork*.*free*()){

*leftFork*.*pickUp*();

System.***out***.println("Philosopher " + *num* + " picked up the left chopstick");

if (*rightFork*.*free*()){

*rightFork*.*pickUp*();

System.***out***.println("Philosopher " + *num* + " picked up the right chopstick");

eat();

*leftFork*.*putDown*();

System.***out***.println("Philosopher " + *num* + " put down the left chopstick");

*rightFork*.*putDown*();

System.***out***.println("Philosopher " + *num* + " put down the right chopstick");

}

else {

*leftFork*.*putDown*();

System.***out***.println("Philosopher " + *num* + " put down the left chopstick");

}

}

}

}

}

public void eat(){

try {

Random random = new Random();

int eating = random.nextInt((2000-1000)+1)+1000;

System.***out***.println("Philosopher " + *num* + " eats for " + eating);

Thread.*sleep*(eating);

}

catch (Exception e){

e.printStackTrace();

}

}

}

public static class Fork {

static Semaphore *sem* = new Semaphore(1);

static void putDown(){

*sem*.release();

}

static void pickUp(){

try {

*sem*.tryAcquire();

}

catch (Exception e){

e.printStackTrace();

}

}

static boolean free(){

if (*sem*.availablePermits() > 0) System.***out***.println(*sem*.availablePermits());

return *sem*.availablePermits() > 0;

}

}

public static void main(String[] args){

for (int i = 0; i < *numPhilosophers*; i++){

*forks*[i] = new Fork();

}

for (int i = 0; i < *numPhilosophers*; i++) {

*philosophers*[i] = new Philosopher(i, *forks*[i], *forks*[(i + 1) % *numPhilosophers*]);

*philosophers*[i].start();

}

while (true) {

try {

Thread.*sleep*(1000);

boolean deadlock = true;

for (Fork f : *forks*) {

if (f.*free*()) {

deadlock = false;

break;

}

}

if (deadlock) {

Thread.*sleep*(1000);

System.***out***.println("Deadlock has occured");

break;

}

} catch (Exception e) {

e.printStackTrace(System.***out***);

}

}

System.*exit*(0);

}

}

Løst med innebygd Semaphore class i java:

import java.util.concurrent.Semaphore;

import java.util.concurrent.ThreadLocalRandom;

public class Oppg3java {

static Philosopher *philosophers*[] = new Philosopher[5];

static Fork *forks*[] = new Fork[5];

private static class Fork {

Semaphore sem = new Semaphore(1);

void signal() {

sem.release();

}

boolean free() {

return sem.availablePermits() > 0;

}

void waitSem() {

try {

sem.acquire();

} catch (Exception e) {

e.printStackTrace(System.***out***);

}

}

}

private static class Philosopher extends Thread {

public int num;

public Fork leftFork;

public Fork rightFork;

Philosopher(int number, Fork left, Fork right) {

num = number;

leftFork = left;

rightFork = right;

}

public void run() {

System.***out***.println("Philosopher " + (num +1));

while (true) {

if (num % 2 == 0) {

rightFork.waitSem();

System.***out***.println("Philosopher " + (num +1) + " grabs right fork.");

leftFork.waitSem();

System.***out***.println("Philosopher " + (num +1) + " grabs left fork.");

eat();

rightFork.signal();

System.***out***.println("Philosopher " + (num +1) + " releases right fork.");

leftFork.signal();

System.***out***.println("Philosopher " + (num +1) + " releases left fork.");

} else {

leftFork.waitSem();

System.***out***.println("Philosopher " + (num +1) + " grabs left fork");

rightFork.waitSem();

System.***out***.println("Philosopher " + (num +1) + " grabs right fork");

eat();

leftFork.signal();

System.***out***.println("Philosopher " + (num +1) + " releases left fork");

rightFork.signal();

System.***out***.println("Philosopher " + (num +1) + " releases right fork");

}

}

}

public void eat() {

try {

int sleep = ThreadLocalRandom.*current*().nextInt(0, 100);

System.***out***.println("Philosopher " + (num +1) + " eats " + sleep);

Thread.*sleep*(sleep);

} catch (Exception e) {

e.printStackTrace(System.***out***);

}

}

}

public static void main(String argv[]) {

for (int i = 0; i < 5; i++) {

*forks*[i] = new Fork();

}

for (int i = 0; i < 5; i++) {

*philosophers*[i] = new Philosopher(i, *forks*[i], *forks*[(i + 1) % 5]);

*philosophers*[i].start();

}

while (true) {

try {

Thread.*sleep*(1000);

boolean deadlock = true;

for (Fork f : *forks*) {

if (f.free()) {

deadlock = false;

break;

}

}

if (deadlock) {

Thread.*sleep*(1000);

System.***out***.println("Deadlock has occured");

break;

}

} catch (Exception e) {

e.printStackTrace(System.***out***);

}

}

System.*exit*(0);

}

}