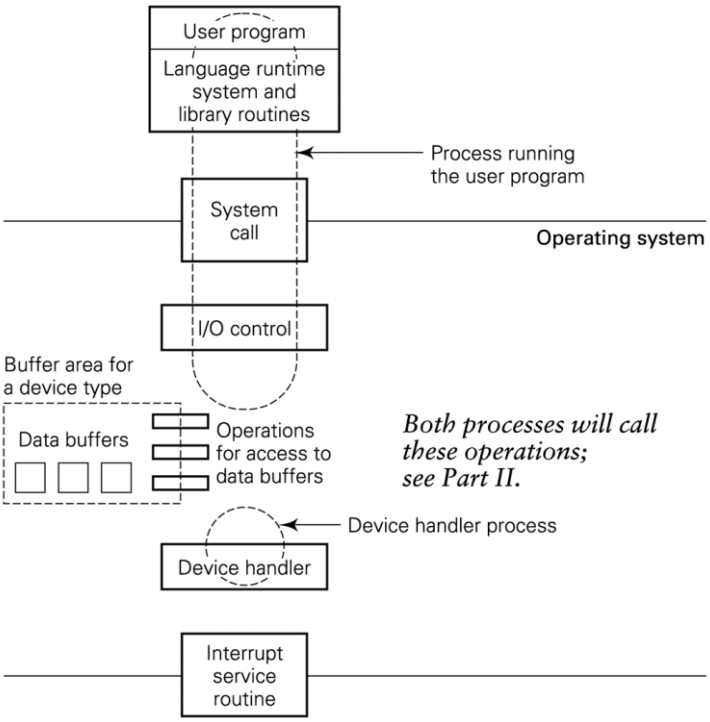
Vraag 1 – Synchrone versus asynchrone I/O, interrupts

Beschouw onderstaande figuur, dat een programma in uitvoering (een proces) voorstelt dat gegevens nodig heeft van een file op schijf, en daarom een synchrone leesoperatie (*synchronous read*) uitvoert.



Waarom moet er dan een *System call* gebeuren?

Wat moet er dan gebeuren met dit proces (in welke modus en toestand komt dit proces) bij die *System call*?

- modus (*mode*)

- toestand (*state*)

Speelt DMA een rol in deze context?

Wat is de rol van de *hardware cache* bij DMA? Is die *cache* zichtbaar in de figuur?

Wanneer is er bij deze leesoperatie dan sprake van een *interrupt*? Door wie of wat wordt die *interrupt* gegenereerd?

Wat is het effect van die *interrupt* in de context van die leesoperatie?

Waarom wordt de *Interrupt service routine* zelf niet als een proces voorgesteld in de figuur? Hoe wordt de code van die routine dan uitgevoerd?

Wat echter als de file asynchroon wordt gelezen (*asynchronous read*), zoals in onderstaand Java programma:[1]

```
import ...;

public class AsynchronousReadExample {

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

        Path path = Paths.get("c:\\java\\bacon\\abc.txt");

        try (AsynchronousFileChannel afc = AsynchronousFileChannel.open(path, READ)) {

            int fileSize = (int) afc.size();
            int loopCount = 0;
            ByteBuffer dataBuffer = ByteBuffer.allocate(fileSize);
            Future<Integer> result = afc.read(dataBuffer, 0);

            while (!result.isDone()) {
                  loopCount++;
            }

            int readBytes = result.get();

            System.out.format("%s loops performed during read %n", loopCount);
            System.out.format("%s bytes read from %s%n", readBytes, path);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Uitvoering van dit programma levert bijvoorbeeld volgende output:

```
2775 loops performed during read
2080 bytes read from c:\java\bacon\abc.txt
```

Wat gebeurt er tijdens de uitvoering van dit programma met het proces (in welke toestand komt dit proces) als de gegevens nog niet gebufferd zijn op het moment dat de file gelezen wordt?

Wat is nu (bij asynchroon lezen) het effect van die eerder vermelde *interrupt*?

[1] Om plaats te besparen zijn de `import` statements niet volledig weergegeven.

Vraag 2 – Concurrency en multithreading

Beschouw het volgende Java programma, waarin 26 draden (*threads*) a, b, ... z worden gecreëerd,[2] waarbij iedere draad verantwoordelijk is voor één specifieke letter van het alfabet ("a", "b", ... "z"):

```
public class PrintThread6 extends Thread {

    String message;
    public PrintThread6 (String m) {
        message = m;
    }

    public void run() {
        for (int i = 0; i<10000; i++) {
            try {
                Thread.sleep(1);
            }
            catch(InterruptedException e) {
                System.out.println("Thread interrupted in sleep");
            }
            System.out.print(message);
        }
    }

    public static void main (String args[ ]) {

        Thread a = new PrintThread6("a");
        Thread b = new PrintThread6("b");
        ....
        Thread z = new PrintThread6("z");

        a.setPriority(3);
        b.setPriority(7);
        System.out.println("Priority of main: " + Thread.currentThread().getPriority()
               + ", thread a: " + a.getPriority() + ", thread b: " + b.getPriority());

        a.start();  b.start();   ...   z.start();

        int active = Thread.activeCount();

        System.out.print("%%%");

        try {
            a.join();  b.join();  ...   z.join();
        }
        catch (InterruptedException ie) {
            System.out.println("Main interrupted whilst joining threads");
        }
        System.out.println("%%%");
        System.out.println("\nActive threads: " + active);
    }
}
```

Uitvoering van dit programma levert volgende output[3] (enkel begin en einde zijn weergegeven):

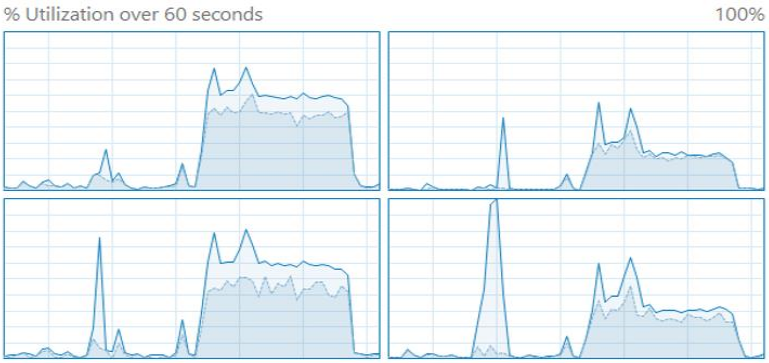
```
c:\Java\Bacon\PrintThread>java PrintThread6
Priority of main: 5, thread a: 3, thread b: 7
cabbaed%%%fhgeadbjonirfmlkpaghkcdxebxdeazxgypchkvcuhwtulrfmicnoqsjsnonqrulimfvtcwhkypgzxadabeg
zxkypwchtvfmilrunoqsjnoqruvotfimlcwphgzekwbyxdalcbmzecihgptvfourqsjnvftpgcheialdbzmxywkaldyeim
zaxbchpatgqroufvjsnthzxcbowkimldeycxhzstnvjufranqgpqvtshxzcikwoycedlombeywlkidchixzstvptqarfuj
agnqtfparuxzhivsklwdcyebomcdwlbcyoeivksarhuxzpzfnjgtqxhupayeosrcvkbilwmbdikvlwsrcaoaeypuxhqgtj
ntzfjqgaeoxhpuylvkwcrsyibdmupxhrcwvlkajzfvotngqetvfoalkjwzcrxphudmysiburxhspczjwjafvtlweqkogkn
oeqwlatjvfzphcsymdxbruidmryschpzsvfaenokgtjqlwafvqszeonymrhypcduxibnrmoezsqvajtkgfvlwaqsezqogk
tjucdpmyrhnixbutjkogqzawlvsefkojbtumpdcixnhyrpmuytjhnxcbidokfsvebdalwezqgaldqwbzwevskfoicumprt
xhnhyjpmutrxicofksvezwebagelqdgawbwvzicwxrsfokyjhntpmujkyrxcwosfvizbagedlqvsofkjuwmcxyrpnthjkf
ovsqrpyxsuwmcadclegbizawmusyxpzrqvofkjhtnagilecdxywsbumafovqrzpcelignthkjafmuswybdxgilecrpzqov
awysumfjkhtnzrpeclgixdberptznhfkjmauybsqovwpreydxigclaumjkfhntzrpwvoqsbafnhkjmulcgixdeynabovwp
irztsfqagclujmwkhovnyexbdbawmjulcgqfsrtzipdxenyhvkoraqfgclumjwovkyhnexdspibztdaexnhykvowjmlucf
gqrtzsbripaqgfculmjwovkhynexdstzakvowjmulcfgqprbiaovtkzsxdenhytarpqgfclumjwyhnedxszkoibvbahwyj

.....

mtrexswfdopghjzknluqcbvbcqulnkzjhgpodfwsexrmtmviyrxewsfdopghjzlknbucqcklzjhgopdfsewxriyvtqmbun
virxwesfdpoghjzlkcnuqmbytrgopdfsweixvrtymqunckljzhvxiewsdfgobptqymrtpogfdswiexvhzjkcnurybqmmyr
unckjzhvxeisfwdgpobqttqopgdwfsiexvhjzkcnyubmrbomuynckzjvhxesifwdgqptrjzkcnyumotrpqdgwfisvexbhw
prtomuyncjkzhvxsegifbdqigesxzhkjcnyutmorpwqdbfsfdqwrpomutynjckhzxeigbdigexhzkcjntyumoprswbqffq
swrpomuytnjckzxhbedigqgidehxzkcjntyurmopwfsbfwoprmuytncjkzxhedigqsbpsqgidehzxkjcytnbwumrofpqed
sgiepformuwntycjkxzhsgibqddqigshzxkjctynwurmofebpdpfeormuwntycjkxzsihbqgdgqhiszxjkcytnuwmroefb
```

[2] Niet alle 26 draden worden expliciet getoond in de programmacode (enkel a, b, ... z worden getoond). Bemerk dat a een lagere prioriteit (3) krijgt dan normaal (5), en b een hogere (7).

[3] Op een eenvoudige laptop met vier logische processoren.

```
pxihqgdpfeormwuntycksbjzxzjskcytnuwmroefpgdqdibhgpferoumwntkycsjxzskytnwmuorefpghidqzcbxjzdihg
fperoumwntyksxjqcbzbqjxsktynmuworephgfcizbzicfghpreowmuyntksqxjbjxqsktnyuwmoeprhgfizcjczigfhrp
eomwuyntksqxbhfgizcjxqkstnyuwmropebormwuyntskqxjczigfghizcjkxsqntyuwmroepbferomwuytnqsxkjczihb
gphbjkxsqntyuwmroefpcgizbzigcpfeomrwutynxqshjkzjshqxnyturwomefpcgkibpmowrutynxhqsjzigkcebfbckg
izjsqhxnyutrwofmpbpmowtuynxhqsjzigckbutwpompkcgizjqhxnymobwuspuswonmxhjqzikcbgpgckizxmqnsowubc
uonsxmqizkgbugkzmiqcsxobnunosxqcmikgzbbzgkimcxqosnukunsqoxcmizgbsgzimxcokqbunbnuqkocxmizsgbnug
zsixmockqbqkcosmiznugbgnziosmcqkgkmqcsoinbzknziscgmbqbmqgsczikbqmkzncimincczkbqmqkzcnbizinckmb
qbqmkcinzikcbmqckizczikmqbmqcbizmzibqcmcqzbiqicbmcmibqbqcimcmbiqbqimcbcqiciqbcbiqcqbicibqcqbqb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb%%%

Active threads: 27
```

Tijdens deze uitvoering kon onderstaand CPU gebruik worden gemeten over een periode van 60 seconden. De eerste piek rond 20 seconden is voor de compilatie van het Java programma (`javac PrintThread6.java`), terwijl de eigenlijke uitvoering (`java PrintThread6`) een tiental seconden later van start gaat.



Gaat het in beide gevallen (bij de voorafgaande compilatie, en tijdens de uitvoering) om een concurrent systeem? Inherent concurrent of potentieel concurrent?

Waarom vallen de vier pieken van de voorafgaande compilatie niet perfect samen in de tijd, terwijl de eigenlijke uitvoering van het programma op de vier logische processoren wel perfect lijkt samen te vallen?

Waarom verloopt de voorafgaande compilatie van het programma hoofdzakelijk in *user mode* (lichter gekleurd), terwijl de eigenlijke uitvoering van PrintThread6 haast volledig in *kernel mode* gebeurt (het donkere gedeelte)?

Welke elementen komen bij de uitvoering van PrintThread6 in de stack? Welke in de heap?

Over hoeveel stacks gaat het bij de uitvoering van PrintThread6? Hoeveel heaps?

Wanneer wordt bij de uitvoering van PrintThread6 een nieuw stackframe gecreëerd? Wanneer wordt dat stackframe terug verwijderd?

Constatatie: ondanks de hogere prioriteit van thread b, wordt op het einde nog een hele reeks b's geproduceerd. Mogelijke verklaring(en) en/of opmerkingen?

Vraag 3 – Bounded buffer, semaforen, monitor, deadlock

Beschouw volgende implementatie van een begrensde circulaire buffer (*bounded buffer*), die gebruikt wordt door meerdere producenten (*producers*) en consumenten (*consumers*).[4]

```
private static class BoundedBuffer {

    static final int BUFFER_SIZE = 5
    private Object[] buffer;
    private final Semaphore guard, spaces, items;
    private int in, out;

    private BoundedBuffer() {
        guard = new Semaphore(1, true);
        spaces = new Semaphore(BUFFER_SIZE, true);
        items = new Semaphore(0, true);
        buffer = new Object[BUFFER_SIZE];
        in = 0;  out = 0;
    }

    public void insert(Object item) throws InterruptedException {
        guard.acquire();
        spaces.acquire();
        buffer[in] = item;
        System.out.println("item " + item + " inserted in slot " + in);
        in = (in + 1) % BUFFER_SIZE;
        guard.release();
        items.release();
    }

    public Object remove() throws InterruptedException {
        guard.acquire();
        items.acquire();
        Object item = buffer[out];
        System.out.println ("     item " + item + " removed from slot " + out);
        out = (out + 1) % BUFFER_SIZE;
        guard.release();
        spaces.release();
        return item;
    }
}
```

Waarvoor worden de semaforen `guard`, `spaces`, `items` gebruikt in dit voorbeeld? Over welke soorten semaforen gaat het?

- `guard`

- `spaces`

- `items`

[4] De Java code toont enkel de klasse BoundedBuffer; de programmacode van Producers en Consumers wordt hier niet getoond.

Is er hier een risico op *deadlock*? Gebruik de vier voorwaarden voor *deadlock* om aan te tonen of *deadlock* al dan niet kan voorkomen.

Kan het risico op *deadlock* (of de afwezigheid daarvan) ook met een *object allocation graph* (die normalerwijze gebruikt wordt voor *deadlock detection*) worden aangetoond? (Beschouw hiervoor het geval met één producent en één consument.)

Eventueel: hoe moet of kan die deadlock worden aangepakt in dit geval?

Beschouw de alternatieve implementatie van de buffer met behulp van de monitorconstructie:

```
private static class BoundedBuffer {

    static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out, count;

    private BoundedBuffer() {
        buffer = new Object[BUFFER_SIZE];
        in = 0;  out = 0;  count = 0;
    }

    public synchronized void insert(Object item) throws InterruptedException {

        while(count == BUFFER_SIZE) {  wait(); }
        buffer[in] = item;
        System.out.println("item " + item + " inserted in slot " + in);
        ++count;
        in = (in + 1) % BUFFER_SIZE;
        notifyAll();
    }

    public synchronized Object remove() throws InterruptedException {

        while (count == 0) {  wait(); }
        Object item = buffer[out];
        System.out.println ("     item " + item + " removed from slot " + out);
        --count;
        out = (out + 1) % BUFFER_SIZE;
        notifyAll();
        return item;
    }
}
```

Is er hier een risico op deadlock?

Waarom `notifyAll()` en niet gewoon `notify()`?

Vraag 4 – Memory-mapped file

In volgend Java programma wordt een file op twee verschillende manieren random verwerkt. De eerste test werkt met de klassieke lees (*read*) en schrijf (*write*) operaties. De tweede test werkt met een *memory-mapped* file. In beide tests wordt herhaaldelijk een willekeurige positie gekozen, en als het element op die positie (een integer) in de file nog op nul staat, dan wordt die waarde overschreven met het nummer van de positie. Het aantal pogingen is gelijk aan de grootte van de file (hoe groter de file hoe meer pogingen).

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class FileIO {
    private static int numOfInts = 1000000;
    private abstract static class Tester {
        private String name;
        public Tester(String name) {
            this.name = name;
        }
        public long runTest() {
            System.out.print(name + ": ");
            try {
                long startTime = System.currentTimeMillis();
                test();
                long endTime = System.currentTimeMillis();
                return (endTime - startTime);
            }
            catch (IOException e) { throw new RuntimeException(e); }
        }
        public abstract void test() throws IOException;
    }

    private static Tester[] tests = {
        new Tester("Processing a file randomly") {
            public void test() throws IOException {
                RandomAccessFile raf = new RandomAccessFile(new File("temp1.tmp"), "rw");
                raf.seek(numOfInts * 4);
                raf.writeInt(0);
                for (int i = 0; i < numOfInts; i++) {
                    int number = (int) (Math.random() * numOfInts);
                    raf.seek(number * 4);
                    if (raf.readInt() == 0) {
                        raf.seek(number * 4);
                        raf.writeInt(number);
                    }
                }
                raf.close();
            }
        },
        new Tester("Processing a memory-mapped file randomly") {
            public void test() throws IOException {
                RandomAccessFile raf = new RandomAccessFile(new File("temp2.tmp"), "rw");
                raf.setLength(numOfInts * 4);
                FileChannel fc = raf.getChannel();
                IntBuffer ib = fc.map(FileChannel.MapMode.READ_WRITE, 0,
                                      fc.size()).asIntBuffer();
                for(int i = 1; i < numOfInts; i++) {
                    int number = (int) (Math.random() * numOfInts);
                    if (ib.get(number) == 0) {
                        ib.put(number, number);
                    }
                }
                fc.close();
            }
        }
    };
    public static void main(String[] args) {
        File oldCopy = new File("temp1.tmp");
        if(oldCopy.exists()) oldCopy.delete();
        oldCopy = new File("temp2.tmp");
        if(oldCopy.exists()) oldCopy.delete();
        if(args.length > 0) {
            numOfInts = Integer.parseInt(args[0]);
        }
        for(int i = 0; i < tests.length; i++) {
            System.out.println(tests[i].runTest());
        }
    }
}
```

Wat is de betekenis van een *memory-mapped* file?

Kan dergelijke *memory-mapped* file gebruikt worden voor IPC? Ook voor IPC tussen gedistribueerde processen? Waarom (niet)?

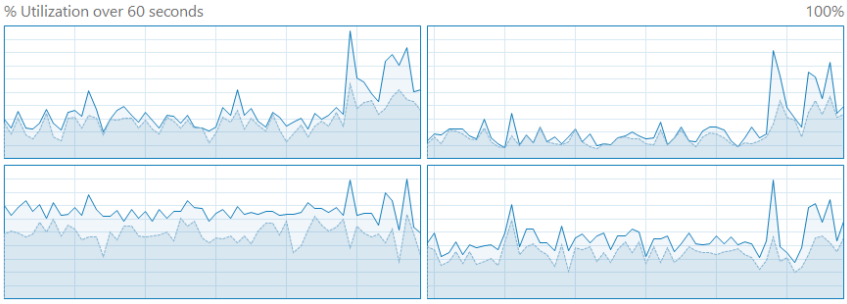
Herhaaldelijke uitvoering van het programma voor toenemende grootte van de file (100, 1000, 10000, ... integers) levert volgend resultaat voor beide tests, in milliseconden:

```
c:\Java\Bacon\FileIO\Test>java  FileIO 100
Processing a file randomly: 11
Processing a memory-mapped file randomly: 59

c:\Java\Bacon\FileIO\Test>java  FileIO 1000
Processing a file randomly: 28
Processing a memory-mapped file randomly: 5

c:\Java\Bacon\FileIO\Test>java  FileIO 10000
Processing a file randomly: 207
Processing a memory-mapped file randomly: 6

c:\Java\Bacon\FileIO\Test>java  FileIO 100000
Processing a file randomly: 1870
Processing a memory-mapped file randomly: 14

c:\Java\Bacon\FileIO\Test>java  FileIO 1000000
Processing a file randomly: 20689
Processing a memory-mapped file randomly: 116

c:\Java\Bacon\FileIO\Test>java  FileIO 10000000
Processing a file randomly: 223431
Processing a memory-mapped file randomly: 1393

c:\Java\Bacon\FileIO\Test>java  FileIO 100000000
Processing a file randomly: 2313795
Processing a memory-mapped file randomly: 16723
```

Terwijl er voor kleine files geen wezenlijk verschil is tussen de resultaten van beide tests,[5] begint het verschil vanaf een grootte van 10000 duidelijk op te lopen. Uiteindelijk is de eerste test ("Processing a file randomly") ongeveer 150 keer trager dan de tweede test met de *memory-mapped* file. Hoe kan dit verschil worden verklaard?

[5] Aangezien er geen garantie is dat een operating systeem echt tot op de milliseconde meet, kunnen kleine gemeten waardes niet gebruikt worden om met zekerheid te vergelijken.

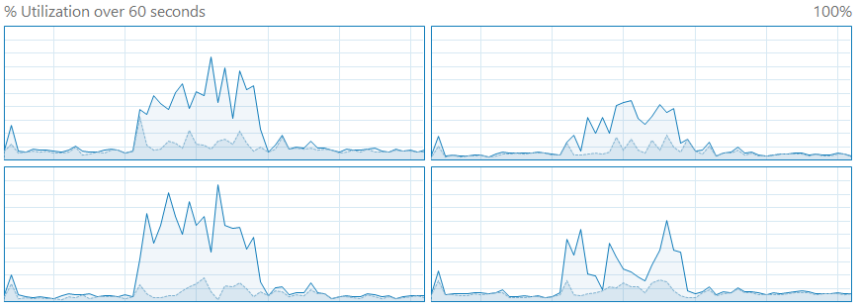
Tijdens de minutenlange klassieke verwerking ("Processing a file randomly") van de file met 100000000 elementen, kan het CPU gebruik worden gevolgd met behulp van een task manager tool:[6]



Opvallend: het CPU gebruik schommelt tijdens de eerste test voortdurend rond 50%. Waarom geen 100%?

Waarom verloopt de uitvoering van deze eerste test haast uitsluitend in *kernel mode* (het donkere gedeelte)?

Wanneer de tweede test (met de *memory-mapped* file) apart wordt gemeten, krijgen we de ruim 16 seconden activiteit centraal in beeld:



Waarom verloopt de uitvoering dit keer hoofdzakelijk in user mode (het lichter gekleurde gedeelte)?

[6] Bijvoorbeeld Task Manager (Windows) of Activity Monitor (macOS).

Wanneer nog een stap verder wordt gegaan, en de test wordt uitgevoerd met een file van 500000000 integers, verschijnt na ongeveer 220 minuten volgend resultaat:

```
c:\Java\Bacon\FileIO\Test>java  FileIO 500000000
Processing a file randomly: 13140021
Processing a memory-mapped file randomly: Exception in thread "main"
java.lang.RuntimeException: java.io.IOException: Map failed
        ...
Caused by: java.io.IOException: Map failed
        ...
Caused by: java.lang.OutOfMemoryError: Map failed
        ...
```

Wat is er nu aan de hand? Waarom lukt de eerste test wel, maar de tweede niet?

Hoe kan worden verklaard dat tijdens beide tests alle vier de logische processoren gelijktijdig bezig zijn? Is hier sprake van *multithreading*?

Dit deel van de pagina niet gebruiken.