U1761249 Adam Birch

Operating Systems Assignment

Contents

Question 1

a. User mode doesn't have the ability for executing code to directly access hardware or reference memory. Kernel mode allows executing code to have unrestricted access to the hardware within the machine. Kernel mode is responsible for the most critical executions within the computer, while User mode is responsible for most executions within the use of a computer. Kernel mode processes that crash are more likely to stop the entire machine from working, while User mode crashes may stop a program or operation.

b. The three common events that lead to a process switch are: Clock interrupt, I/O interrupt, and Memory fault. Clock interrupt will cause a process switch when the time taken for the process to run has lapsed – its time slice expired. An I/O interrupt will occur when a process requires an I/O event and is taking up space waiting for it. Memory faults occur when a process requests memory from secondary memory. As another process is required to retrieve this data, the process that requested the data is swapped.

c. Threads are smaller than processes. Theoretically, a process can be made of threads while a thread cannot be made of processes. Threads within the same process share the same address space, allowing threads to read and write to the same data locations – while the same is possible for processes using IPC, it is difficult and resource intensive. Multi-threading allows for faster processing of a process, while multi-tasking allows the running of multiple processes – however these tasks do not run concurrently and thus are not faster because of multi-tasking.

d. Semaphores are a way of sending signals between processes, preventing the processes from trying to run at the same time. General semaphores allow a numerical value to be assigned and altered, while a Binary semaphore only uses 0 and 1. Both types of semaphores can be accessed by using the wait() and signal() operations regardless of type.

e. Deadlocks occur when a process or processes are waiting for at least one more to finish using a resource, while that one also waits for a process to finish using a resource. Livelock prevents a thread from continuing because it is too busy to make progress. This can often occur if a thread is used to respond to another thread. Starvation occurs when a process is unable to use the required resources because they are being used by another process for a long period of time. Deadlock is the most common issue of the three, and the easiest to resolve. Starvation and Livelock are harder to deal with as they occur without being completely stuck but have a possibility to resolve themselves if the dependencies resolve themselves.

f. Dynamic Partitioning allows logical processors, memory, and adapters to be dynamically added and removed from logical partitions. This is good for when the computer requires different quantities of storage space during runtime as it allows many different processes to have the right amount of storage allocated. An issue caused by this is that as more processes start and end the memory locations become more and more fragmented due to the allocated memory being given an address based on what was available at the time. This is avoided by using four methods of partitioning methods: Best fit, Worst fit, First fit, and Last fit. These allocate memory in different ways to reduce the amount of wasted space within the memory. Each of these methods are used during memory allocation to attempt to fill in holes in the memory usage to make more efficient use of the resources available.

g. Multi-Level Page tables can be thought of as pages of pages. The pages point to addresses within other pages rather than to a memory address. Another way to describe it is a hierarchical file structure within a computer, where a folder can hold folders of files that are semi-related. One issue with this is that a program can be slowed down by several times its usual running speed. Inverted Page Tables use one entry per physical page within a linear array, where each physical page is mapped to a virtual page. Due to this, each entry is mapped to a virtual page rather than a physical page. This differs from a regular page table as the address that is stored doesn't exist. A regular page table can record many entries within one table. Inverted page tables can only store one. This means that to record data for many processes it would require many page tables.
h. Starvation can be caused by simplistic scheduling algorithms if a process begins to run but the process requires data that is being used by another process which blocks its ability to run. If a priority system is in place it can cause starvation as, if a high priority process never releases a resource, a lower resource may be unable to process due to a lack of access to the resource. One common approach to prevent this is to allocate each process a set time to process, after which it is paused, and another process runs. This prevents long periods of a process from being able to run as it periodically releases the resources held by processes and allows another process the opportunity to run.
i. Direct Memory Access (DMA) allows data to pass from an attached device straight to the memory on a computer's motherboard. This is possible due to the configuration of some bus architectures. This allows the processor to perform other tasks while a task (e.g. I/O operations) to take place. This frees up the CPU to perform other more critical tasks.
j. Indexed sequential files are faster than sequential due to the index. Data is stored with an index generated by the data itself. If this process is known, the sought data can be run through this to find the index that the data is likely being held in.

Question 2

a. This is an unsafe state as P1 and P4 can't be allocated more resources for R5 to be able to execute. P1 can release all of the R5 processes that it has to be able to execute.
The following sequence of executions assumes that P1 has released the R5 resources that it has.

AVAILABLE: 8 3 6 6 4

P4: already allocated 1 1 0 0 1
P4: allocated 3 3 4 4 3 – claimed = 4 4 4 4 4 and executed.
P3: already allocated 4 1 0 2 0
P3: allocated 0 2 3 0 0 – claimed = 4 3 3 2 0 and executed.
P2: already allocated 0 1 1 1 0
P2: allocated 2 1 2 2 0 – claimed = 2 2 3 3 0 and executed.
P1: already allocated 2 0 2 1 2
P1: allocated 7 5 3 4 3 – claimed = 9 5 5 5 5 and executed.

b. This is not a safe state and will result in a Deadlock as there are no possible executions with the current quantity of available resources. All process would need to release its resources before any process is able to run.

Question 3

a. PTE = 512 Frame Number = (512 / 2) = 256 Address = 0011011010101010

16-bit address means there are 2^{16} addresses in the virtual address space.

2^{16} / page size = 2^{9}
65536 / page size = 512
65536 / 512 = page size
Page size = 128 = 2^{7} bits

The most significant 9 bits will denote the page being referenced.
The remaining 7 bits denote the offset of the data required.

Page number	Offset
001101101	0101010

Page Entry in question is (64+32+8+4+1) = Entry 109.
The Frame number is half of the Entry: 109/2 = 54.5 (Floored = 54)
The offset remains as 0101010 (32 + 8 + 2) = 42

Physical Address = 000110110 0101010

b. Segment Size = 2048 bytes (2 KB) Segment Table = 22 + 4096 + Segment Number
Logical Address = 0011011010101010

If (offset < Segment Limit) Physical Address = offset + Base Address.
Offset = 11 bits (2^11 ==2048)
Therefore, Segment number = 5 bits

Segment Number	Offset
00110	11010101010

The segment in question is 6.
The segment table is 22 + 4096 + 6 = 4124 = 0001 000000011100
The offset is (1024 + 512 + 128 + 32 + 8 + 2) = 1706
Physical Address = 0001011010101010

Question 4

32-bit Physical Address 48-bit Virtual Address

a. Page Size = 8KB

   48-bit Virtual Address = 2^{48} addresses
   2^{48} / (8 * 2^{10})
   2.8147498e+14 / 8192 = 34359738368 = 2^35

   Therefore, there are 35 Pages in the table.

b.

   Main Memory access = 100ns Disk memory = 2ms Fault Rate = 0.005%
   0.005% of the time the requested page is in Secondary Storage.

   Hit time = 100ns Page Fault Rate = 0.005% Miss Penalty = 2ms

   T = Hit time + Miss Rate * Miss Penalty

   T = 100 + 0.005 * 2000000 = 10100 ns

c. DMAT + 25% = 125 ns

   T = HT + MR * MP

   T – HR / MP = MR

   125 – 100 / 2000000 = 0.0000125%

d. AMAT = Hit time + Miss Rate * Miss Penalty = 30 + 0.05 * 100 = 35ns

Question 5

Process	**Burst Time**
P1	3
P2	3
P3	4
P4	2
P5	7

T.T = Turn around Time = Completion Time – Arrival Time

W.T = Waiting Time = T.T – Burst Time

a. FIFO

P1	P2	P3	P4	P5
0 - 3	3 - 6	6-10	10-12	12-19

Process	Burst Time	Completion Time	Turn Around Time	Waiting Time
P1	3	3	3	0
P2	3	6	6	3
P3	4	10	10	6
P4	2	12	12	10
P5	7	19	19	12

Average Wait time = (3+6+10+12+19) / 5 = 10
Average Turnaround time = (0+3+6+10+12) / 5 = 6.2

b. RR

P1	P2	P3	P4	P5	P1	P2	P3	P5	P5	P5
0-2	2-4	4-6	6-8	8-10	10-11	11-12	12-14	14-16	16-18	18-19

Process	Burst Time	Completion Time	Turn Around Time	Waiting Time
P1	3	11	11	8
P2	3	12	10	7
P3	4	14	10	6
P4	2	8	2	0
P5	7	19	11	4

Average Wait time = (8+7+6+4) / 5 = 5
Average Turnaround time = (11+10+10+2+11) / 5 = 8.8

c. SJF

P4	P1	P2	P3	P5
0-2	2-5	5-8	8-12	12-19

Process	Burst Time	Completion Time	Turn Around Time	Waiting Time
P1	3	5	5	2
P2	3	8	8	5
P3	4	12	12	8
P4	2	2	2	0
P5	7	19	19	12

Average Wait time = (5+8+12+2+19) / 5 = 9.2
Average Turnaround time = (2+5+8+0+12) / 5 = 5.4

Question 6

a) Total Seek Length = 256
Average Seek length = 28.4

b) Total Seek Length = 194
Average Seek length = 21.5

c) Total Seek Length = 153
Average Seek length = 17*

SSTF			SCAN			C-SCAN		
Next track Accessed	Interval	Number of tracks Accessed	Next track Accessed	Interval	Number of tracks Accessed	Next track Accessed	Interval	Number of tracks Accessed
50	0	0	50	0	0	50	0	0
52	2	2	27	23	23	27	23	23
76	24	26	22	5	28	22	5	28
120	44	60	11	11	39	11	11	39
140	20	70	52	41	80	166	0	39
166	26	96	76	24	104	140	26	65
27	139	235	120	44	148	120	20	85
22	5	240	140	20	168	76	44	129
11	11	256	166	26	194	52	24	153

Question 7

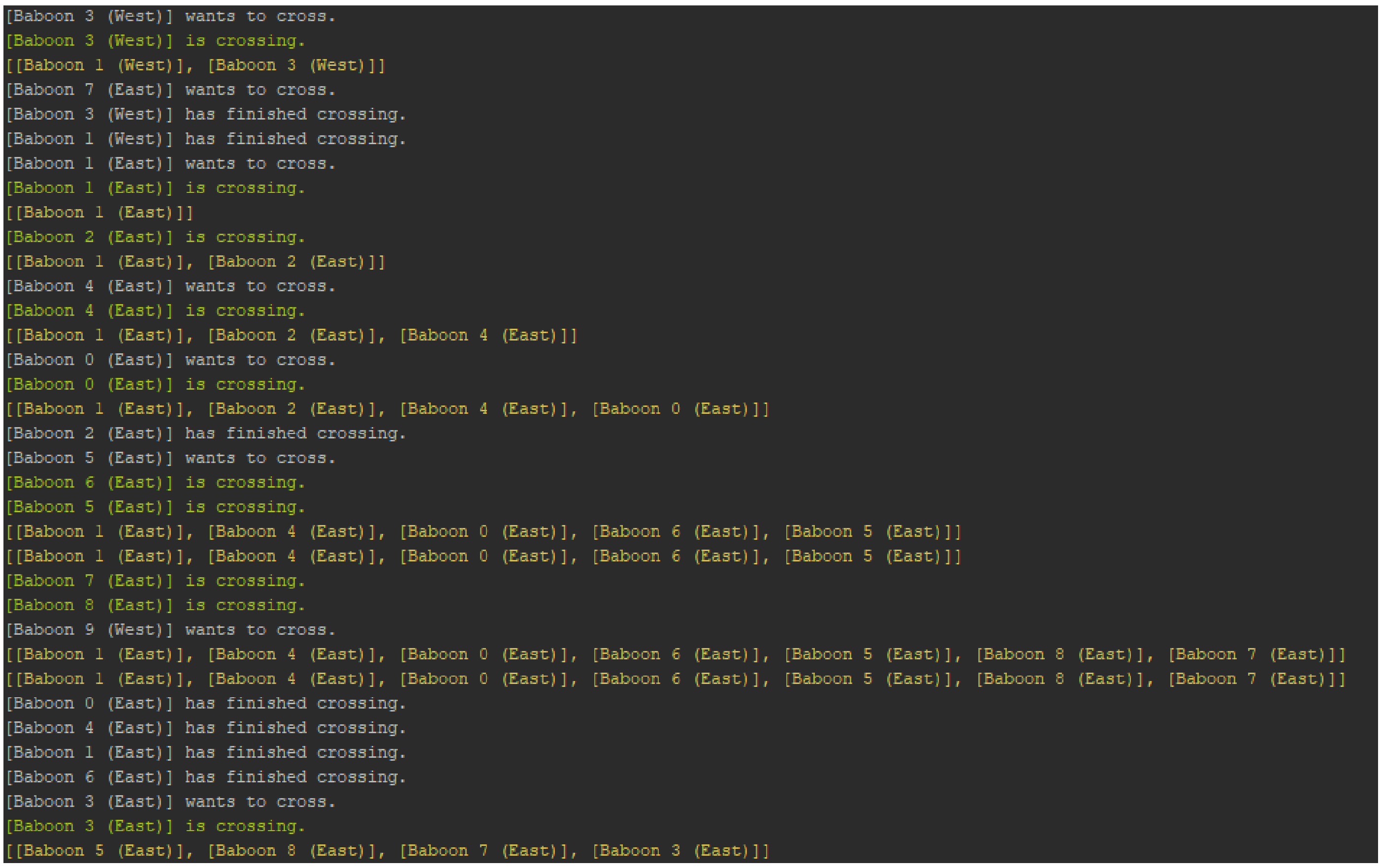
a. 512 / 250 = 2.04
   2 records can fit per block with 12 bytes wasted.
   1000 blocks are required with 12000 bytes wasted.
b. 8KB = 8192B
   8129 / 250 = 32.5
   32 records can fit per block with 129 bytes wasted.
   63 blocks (62.5) are required with 8127 bytes wasted (+125) = 8252 total waste.
c. 8KB = 8192B
   8129 / 250 = 32.5
   32 records can fit per block with 129 bytes wasted.
   Every 2 blocks, another record can be stored spanning the two with 8 bytes wasted between them.
   Every 5th record spans a block.
   47 blocks (46.88) are required with 188 bytes wasted (+ 125) = 313.

Question 8

a.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import java.util.concurrent.Semaphore;


class Baboon implements Runnable
{
    private static final int BaboonCount = 10;
    private static final Semaphore[] rope = new Semaphore[BaboonCount];
    private static Random rand = new Random();
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_GREEN = "\u001B[32m";
    public static final String ANSI_YELLOW = "\u001B[33m";
    private static ArrayList crossing = new ArrayList();
    private static ArrayList onEast = new ArrayList();
    private static ArrayList onWest = new ArrayList();

    private int id;
    private boolean side = false; //    True - on East  False - on West

    public Baboon(int id) {
        this.id = id;
        rope[id] = new Semaphore(1);
        if (id % 2 == 0){
            this.side = true;
            onEast.add(this);
        } // Baboon on East.
        else{
            onWest.add(this);
        }

    }

    private void nap() {          //Make Baboon thread sleep for a while (e.g. to
                                   simulate thinking or eating for a short period)
        try {
            Thread.sleep(rand.nextInt(10) * 1000);
        } catch (InterruptedException e) {
            System.err.println("One baboon thread died :(");
            System.exit(1);
        }
    }

    private void eatBanana() {
        this.nap();
    }

    private void takeRope(){
        try {
            crossing.add(this);
            rope[id].acquire();

        } catch (InterruptedException e) {
            System.out.println(" A baboon thread died. ");
        }
    }

    private synchronized void cross() {

        System.out.println(this + " wants to cross.");

        while(true){
           if(crossing.isEmpty()){
               takeRope();
               break;
           }
           else {
               if (side){
                   if (Collections.disjoint(crossing, onWest)){ // if no westward baboons
                      takeRope();
                      break;
                   }
               }
               if (!side){
                   if (Collections.disjoint(crossing, onEast)){ // if no eastward baboons
                       takeRope();
                       break;
                   }
               }
               this.nap();
           }
        }

        System.out.println(ANSI_GREEN + this + " is crossing." + ANSI_RESET);
        System.out.println(ANSI_YELLOW + crossing + ANSI_RESET);
        this.nap();          //Baboon is crossing for a while

        rope[id].release();
        crossing.remove(this);

        System.out.println(this + " has finished crossing.");
        this.swapSides();
    }

    private synchronized void swapSides() {
        this.side = !this.side;
        if (this.side){onWest.remove(this); onEast.add(this);}
        if(!this.side){onEast.remove(this); onWest.add(this);}
    }

    public String toString() {
        String stringSide = "(West)";
        if(this.side){stringSide = "(East)";}
        return "[Baboon " + id + " " + stringSide + "]";
    }

    public void run() {
        System.out.println(this + " approaches the rope.");
        for (int i = 0; i < 5 ; i++) {
            eatBanana();
            cross();
        }
    }

    public static void main(String [] args) {
        Thread baboons[] = new Thread[BaboonCount];
        System.out.println("Start");

        for (int i = 0; i < BaboonCount; i++) {
            baboons[i] = new Thread(new Baboon(i));
        }


        for (int i = 0; i < BaboonCount; i++) {
            baboons[i].start();
        }

        try {
            for (int i = 0; i < BaboonCount; i++)
                baboons[i].join();
        }
        catch (InterruptedException e) {
            System.err.println("One baboon thread died :(");
            System.exit(1);
        }
        System.out.println("End");
    }
}
```

Here is an example of the output:



```
[Baboon 3 (West)] wants to cross.
[Baboon 3 (West)] is crossing.
[[Baboon 1 (West)], [Baboon 3 (West)]]
[Baboon 7 (East)] wants to cross.
[Baboon 3 (West)] has finished crossing.
[Baboon 1 (West)] has finished crossing.
[Baboon 1 (East)] wants to cross.
[Baboon 1 (East)] is crossing.
[[Baboon 1 (East)]]
[Baboon 2 (East)] is crossing.
[[Baboon 1 (East)], [Baboon 2 (East)]]
[Baboon 4 (East)] wants to cross.
[Baboon 4 (East)] is crossing.
[[Baboon 1 (East)], [Baboon 2 (East)], [Baboon 4 (East)]]
[Baboon 0 (East)] wants to cross.
[Baboon 0 (East)] is crossing.
[[Baboon 1 (East)], [Baboon 2 (East)], [Baboon 4 (East)], [Baboon 0 (East)]]
[Baboon 2 (East)] has finished crossing.
[Baboon 5 (East)] wants to cross.
[Baboon 6 (East)] is crossing.
[Baboon 5 (East)] is crossing.
[[Baboon 1 (East)], [Baboon 4 (East)], [Baboon 0 (East)], [Baboon 6 (East)], [Baboon 5 (East)]]
[[Baboon 1 (East)], [Baboon 4 (East)], [Baboon 0 (East)], [Baboon 6 (East)], [Baboon 5 (East)]]
[Baboon 7 (East)] is crossing.
[Baboon 8 (East)] is crossing.
[Baboon 9 (West)] wants to cross.
[[Baboon 1 (East)], [Baboon 4 (East)], [Baboon 0 (East)], [Baboon 6 (East)], [Baboon 5 (East)], [Baboon 8 (East)], [Baboon 7 (East)]]
[[Baboon 1 (East)], [Baboon 4 (East)], [Baboon 0 (East)], [Baboon 6 (East)], [Baboon 5 (East)], [Baboon 8 (East)], [Baboon 7 (East)]]
[Baboon 0 (East)] has finished crossing.
[Baboon 4 (East)] has finished crossing.
[Baboon 1 (East)] has finished crossing.
[Baboon 6 (East)] has finished crossing.
[Baboon 3 (East)] wants to cross.
[Baboon 3 (East)] is crossing.
[[Baboon 5 (East)], [Baboon 8 (East)], [Baboon 7 (East)], [Baboon 3 (East)]]
```

The green lines show baboons successfully beginning a crossing.

The yellow lines show the current list of crossing baboons.

I added a tag to the name of the baboon to show what side they START on. When they dismount this changes.

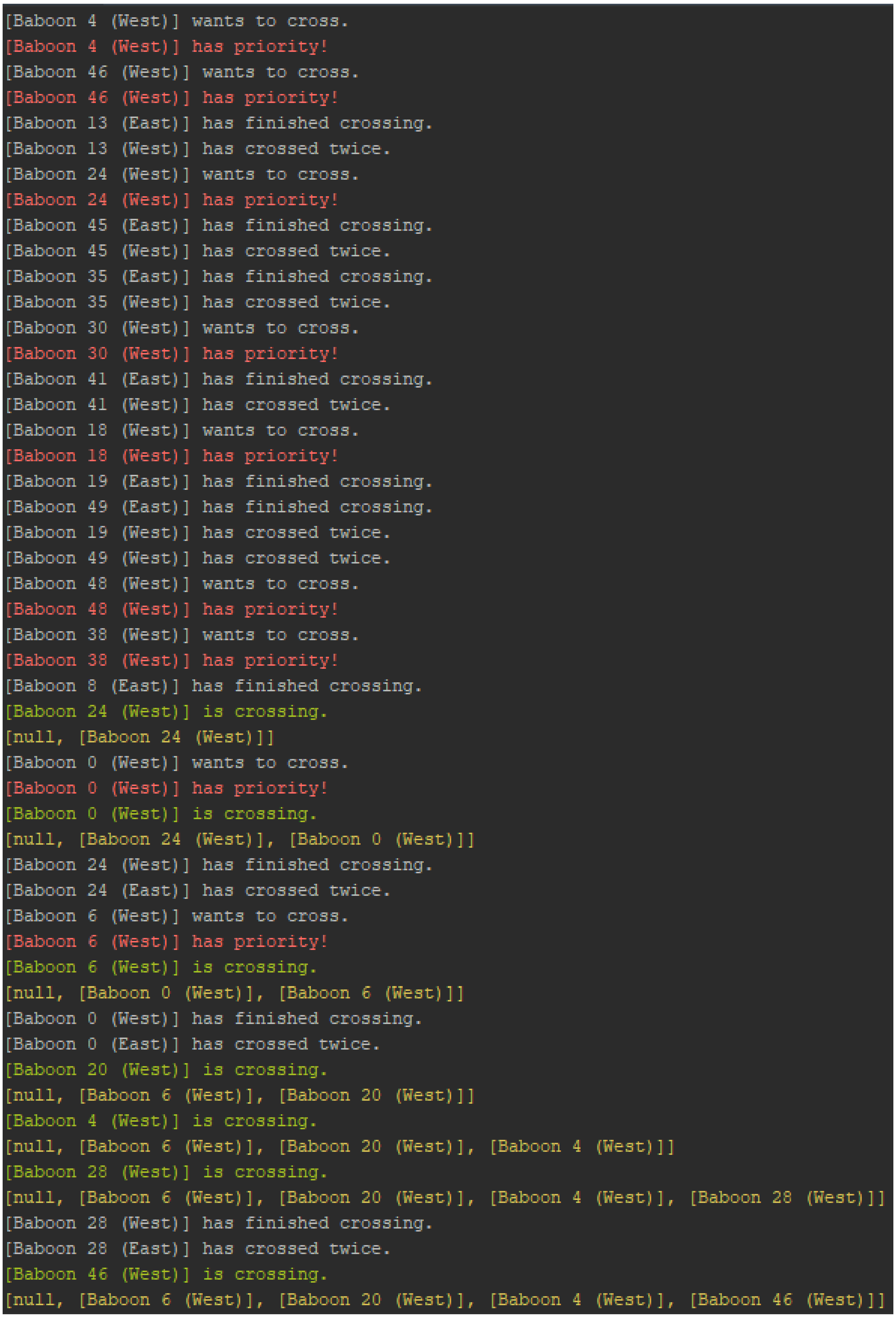
b.

```
 import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import java.util.concurrent.Semaphore;


class Baboon implements Runnable
{
    private static final int BaboonCount = 50;
    private static int killCount = 0;
    private static final Semaphore[] rope = new Semaphore[BaboonCount];
    private static Random rand = new Random();
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_GREEN = "\u001B[32m";
    public static final String ANSI_YELLOW = "\u001B[33m";
    public static final String ANSI_RED = "\u001B[31m";
    private static ArrayList crossing = new ArrayList();
    private static ArrayList onEast = new ArrayList();
    private static ArrayList onWest = new ArrayList();
    private static ArrayList priority = new ArrayList();

    private int id;
    private int age;
    private boolean side = false; //    True - on East  False - on West

    public Baboon(int id) {
        this.id = id;
        this.age = 0;
        rope[id] = new Semaphore(1);
        if (id % 2 == 0){
            this.side = true;
            onEast.add(this);
        } // Baboon on East.
        else{
            onWest.add(this);
        }

    }

    private void nap() {          //Make Baboon thread sleep for a while (e.g. to simulate
thinking or eating for a short period)
        try {
            Thread.sleep(rand.nextInt(10) * 1000);
        } catch (InterruptedException e) {
            System.err.println("One baboon thread died :(");
            System.exit(1);
        }
    }

    private void eatBanana() {
        this.nap();
    }

    private void takeRope(){
        try {
            crossing.add(this);
            rope[id].acquire();

        } catch (InterruptedException e) {
            System.out.println(" A baboon thread died. ");
        }
    }

    private synchronized void cross() {

        System.out.println(this + " wants to cross.");

        while(true){
            if (priority.isEmpty() || priority.contains(this)) {
                if(crossing.isEmpty()){
                    takeRope();
                    break;
                }
                else {
                    try {
                        if (side){
                            crossing.removeAll(Collections.singleton(null));
                            if (Collections.disjoint(crossing, onWest)){ // if no westward
baboons

                               takeRope();
                               break;
                            }
                        }
                        if (!side){
                            if (Collections.disjoint(crossing, onEast)){ // if no eastward
baboons

                                takeRope();
                                break;
                            }
                        }
                    } catch (Exception e) {

                    }
                    this.nap();
                }
            }
            this.age++;
            if (this.age == 5){
                priority.add(this);
                System.out.println(ANSI_RED + this + " has priority!" + ANSI_RESET);}
        }

        System.out.println(ANSI_GREEN + this + " is crossing." + ANSI_RESET);
        try {
            System.out.println(ANSI_YELLOW + crossing + ANSI_RESET);
        } catch (Exception e) {

        }
        this.age = 0;
        if (priority.contains(this)){priority.remove(this);}
        this.nap();          //Baboon is crossing for a while

        rope[id].release();
        crossing.remove(this);

        System.out.println(this + " has finished crossing.");
        this.swapSides();
    }

    private synchronized void swapSides() {
        this.side = !this.side;
        if (this.side){onWest.remove(this); onEast.add(this);}
        if(!this.side){onEast.remove(this); onWest.add(this);}

        if (this.id % 2 == 0 && this.side){
            kill();
        }
        else if (!(this.id % 2 == 0) && !this.side){
            kill();
        }
    }

    private void kill() {
        System.out.println(this + " has crossed twice. ");
        killCount++;
        if (killCount == BaboonCount){
            System.out.println(ANSI_YELLOW + " All baboons successfully crossed twice" +
ANSI_RESET);
            System.exit(0);
        }
        while (true) {
            try {
                this.wait(1000000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public String toString() {
        String stringSide = "(West)";
        if(this.side){stringSide = "(East)";}
        return "[Baboon " + id + " " + stringSide + "]";
    }

    public void run() {
        System.out.println(this + " approaches the rope.");
        for (int i = 0; i < 5 ; i++) {
            eatBanana();
            cross();
        }
    }

    public static void main(String [] args) {

        Thread baboons[] = new Thread[BaboonCount];
        System.out.println("Start");

        for (int i = 0; i < BaboonCount; i++) {
            baboons[i] = new Thread(new Baboon(i));
        }


        for (int i = 0; i < BaboonCount; i++) {
            baboons[i].start();
        }

        try {
            for (int i = 0; i < BaboonCount; i++)
                baboons[i].join();
        }
        catch (InterruptedException e) {
            System.err.println("One baboon thread died :(");
            System.exit(1);
        }
        System.out.println("End");
    }
}
```

*NOTE – sometimes the program times out before all the baboons have crossed (I am not sure why) and sometimes a priority is not needed.

Here is an example of the use of priority.

In the example, many west baboons are waiting for baboon 8 to finish crossing.



```
[Baboon 4 (West)] wants to cross.
[Baboon 4 (West)] has priority!
[Baboon 46 (West)] wants to cross.
[Baboon 46 (West)] has priority!
[Baboon 13 (East)] has finished crossing.
[Baboon 13 (West)] has crossed twice.
[Baboon 24 (West)] wants to cross.
[Baboon 24 (West)] has priority!
[Baboon 45 (East)] has finished crossing.
[Baboon 45 (West)] has crossed twice.
[Baboon 35 (East)] has finished crossing.
[Baboon 35 (West)] has crossed twice.
[Baboon 30 (West)] wants to cross.
[Baboon 30 (West)] has priority!
[Baboon 41 (East)] has finished crossing.
[Baboon 41 (West)] has crossed twice.
[Baboon 18 (West)] wants to cross.
[Baboon 18 (West)] has priority!
[Baboon 19 (East)] has finished crossing.
[Baboon 49 (East)] has finished crossing.
[Baboon 19 (West)] has crossed twice.
[Baboon 49 (West)] has crossed twice.
[Baboon 48 (West)] wants to cross.
[Baboon 48 (West)] has priority!
[Baboon 38 (West)] wants to cross.
[Baboon 38 (West)] has priority!
[Baboon 8 (East)] has finished crossing.
[Baboon 24 (West)] is crossing.
[null, [Baboon 24 (West)]]
[Baboon 0 (West)] wants to cross.
[Baboon 0 (West)] has priority!
[Baboon 0 (West)] is crossing.
[null, [Baboon 24 (West)], [Baboon 0 (West)]]
[Baboon 24 (West)] has finished crossing.
[Baboon 24 (East)] has crossed twice.
[Baboon 6 (West)] wants to cross.
[Baboon 6 (West)] has priority!
[Baboon 6 (West)] is crossing.
[null, [Baboon 0 (West)], [Baboon 6 (West)]]
[Baboon 0 (West)] has finished crossing.
[Baboon 0 (East)] has crossed twice.
[Baboon 20 (West)] is crossing.
[null, [Baboon 6 (West)], [Baboon 20 (West)]]
[Baboon 4 (West)] is crossing.
[null, [Baboon 6 (West)], [Baboon 20 (West)], [Baboon 4 (West)]]
[Baboon 28 (West)] is crossing.
[null, [Baboon 6 (West)], [Baboon 20 (West)], [Baboon 4 (West)], [Baboon 28 (West)]]
[Baboon 28 (West)] has finished crossing.
[Baboon 28 (East)] has crossed twice.
[Baboon 46 (West)] is crossing.
[null, [Baboon 6 (West)], [Baboon 20 (West)], [Baboon 4 (West)], [Baboon 46 (West)]]
```