Rüzgar Ayan 21801984

2020-2021 SPRING CS342 HOMEWORK 4 Report

Question 1.

Since the page size is $256 = 2^8$ bytes, the last 8 bits of the addresses are the offset. First, convert the given binary addresses into hexadecimal.

Binary	Hexadecimal
0010 1011 1100 0101	0x2b c5
0001 0011 0001 1100	0x13 1c
1110 0100 1010 0011	0xe4 a3
0010 1011 0001 0111	0x2b 17

Then,

Virtual Address	Virtual Page Number	Offset	Physical Page Number	Physical Address
0x2b c5	0x2b	0xc5	0xfc	**0xfcc5**
0x13 1c	0x13	0x1c	0xa3	**0xa31c**
0xe4 a3	0xe4	0xa3	0xe5	**0xe5a3**
0x2b 17	0x2b	0x17	0xfc	**0xfc17**

Question 2.

Every fourth level table covers: $2^9 * 2^{12} = 2^{21}$ = 2 MB

So, we will need 16 fourth level tables. All of these fit into the first third level table. Then we also need one second level table and the one and only first level table. In total, we have 16 + 1 + 1 + 1 = 19 tables. Each of them have 2^9 entries and each of these entries are 8 bytes. Then, the total RAM required is

$$19(2^9 * 8\ bytes) = 19(4\ KB) = 76\ KB.$$

Question 3.

I tried to draw a binary decision tree in each step for whether the parent process or the child process executes. That got a little bit messy but I was able to detect **10** of the possible results:

1. 40, 60, 30, 10, 80, 45
2. 40, 60, 10, 30, 80, 45
3. 40, 60, 10, 80, 30, 45
4. 60, 40, 30, 10, 80, 45
5. 60, 40, 10, 30, 80, 45
6. 60, 40, 10, 80, 30, 45
7. 40, 30, 60, 10, 80, 45
8. 60, 10, 40, 30, 80, 45
9. 60, 10, 40, 80, 30, 45
10. 60, 10, 80, 40, 30, 45

Question 4.

```
//Shared variables
semaphore mutex = 1;
semaphore table_empty = 0;
semaphore p_and_m = 0;
semaphore p_and_t = 0;
semaphore t_and_m = 0;
```

Non-smoking agent

```
while (1)
{
        wait(mutex);
        rand = #Random integer in [0,2];
        if (rand == 0)
                signal(p_and_m);
        else if (rand == 1)
                signal(p_and_t);
        else if (rand == 2)
                signal(t_and_m);
        signal(mutex);
        wait(table_empty);
}
```

Smoker with infinitely many Tobacco (t) -- Other smokers have similar functions, only the waited semaphore in the first line is different.

```
while (1)
{
        wait(p_and_m); //Wait for paper and match
        wait(mutex);

        //Make the cigarette

        signal(table_empty);
        signal(mutex);

        //Smoke the cigarette for a while
        sleep(100);
}
```

Question 5.

We haven't seen these replacement algorithms in the lectures yet.

Question 6.

Page size is 64 bytes => There are 6 offset bits.

a) (0, 50) => 1024+50 = 1074 = 0000010000 110010

=> Page number = 0000010000 = 16 => Frame number = 26 = 0000011010

=> **Physical address = 0000011010 110010 = 1714**

b) (1,0) => 4196+0 = 4196 = 0001000001 100100

=> Page number = 0001000001 = 65 => Frame number = 75 = 0001001011

=> **Physical address = 0001001011 100100 = 4836**

c) (1,100) => 4196+100 = 4296 = 0001000011 001000

=> Page number = 0001000011 = 67 => Frame number = 77 = 0001001101

=> **Physical address = 0001001101 001000 = 4936**

d) (1,700) => Segment 1 has length 512, but 700 > 512. So this is not a valid address.

e) (2,10) => 128+10 = 138 = 0000000010 001010

=> Page number = 0000000010 = 2 => Frame number = 12 = 0000001100

=> **Physical address = 0000001100 001010 = 778**

f) (3,200) => 2048+200 = 2248 = 0000100011 001000

=> Page number = 0000100011 = 35 => Frame number = 45 = 0000101101

=> **Physical address = 0000101101 001000 = 2888**

Question 7.

Initially the NEED and AVAILABLE tables are computed as follows:

NEED

	A	B	C
P1	4	2	1
P2	2	1	1
P3	4	4	2
P4	2	2	0
P5	2	1	2

AVAILABLE

A	B	C
2	1	0

Wee see that AVAILABLE is not enough to fullfill the NEED of any the processes. So, we are in a deadlock and this is not a safe state.

Question 8.

The AVAILABLE tables is initially computed as follows:

AVAILABLE

A	B	C
3	3	0

We can first grant the request (2,3,0) of P4. After it finishes and releases the resources, the table becomes,

A	B	C
4	3	1

Next, we can grant the request (0,0,1) of P1. After it finishes and releases the resources, the table becomes,

A	B	C
5	3	1

After this point, we cannot grant the requests of any remaining processes (P2, P3, P5). So, these processes cannot finish and we are in a deadlock situation.

Question 9.

To keep the information that how many processes are waiting on a semaphore, I let the value of the semaphore be negative. If the value of the semaphore is –n, it means that there are n processes in that semaphores waiting queue. So when we signal, if the value of the semaphore were negative, we wake up a process waiting on the condition variable. Implementation is as follows:

```
monitor SemaphoreSimulation {
        condition waitSemaphore;

        func wait(semaphore S)
        {
                S = S - 1;
                if (S < 0)
                        wait(waitSemaphore);
        }

        func signal(semaphore S)
        {
                S = S - 1;
                if (S <= 0)
                        signal(waitSemaphore);
        }

}
```