CSCE 313 QUIZ 2 Spring 2021

Name:.........Chaoyang Zhu................................ UIN……930001840...
Total Points: ---------/100

1. [5 pts] While implementing the process state diagram, what is the problem of having only 1 queue for all blocked processes waiting for all events? What is the solution to this problem? Describe with an example.

   The efficiency of dealing with different types of events is low. It does not make sense to make a file request from disk and URL request wait in the same queue. We can solve this problem by separating different kinds of events into different queues.

2. [25 pts] Assume the following processes A, B, C are loaded in memory of a system that uses both multiprogramming(MP) and timesharing (TS) techniques. These processes have 15, 7, and 13 instructions respectively. Also assume that the dispatcher lives at address 100 in memory and spans 4 instructions (i.e., 100-103). The time quantum is long enough for exactly **5 instructions.**

   The following table shows only instruction addresses in the memory with I/O requests labelled, along with the duration of these I/O operations in terms of CPU instructions. Although I/O operations do not take CPU instructions, the duration means that the I/O operations will finish by the time the corresponding number of CPU instructions execute.

   Please draw one possible trace of these 3 processes running together in the CPU. Use page 14-15 of Lecure 3 for reference. You may skip the first invocation of the dispatcher to decide the first process to run in the CPU.

Process A	Process B	Process C
5000	8000	12000
5001	8001	12001(I/O, takes 3 ins.)
5002	8002	12002
5003	8003 (I/O, takes **7 ins.**)	12003
5004 (I/O, takes **6 ins.**)	8004	12004
5005 (I/O, takes **4 ins.**)	8005	12005
5006	8006	12006(I/O, takes 1 ins.)
5007		12007
5008(I/O, takes **5 ins.**)		12008(I/O, takes 2 ins.)
5009		12009
5010		12010
5011		12011
5012		12012

5013 5014		

My answer:
1 5000
2 5001
3 5002
4 5003
5 5004
---------- Timeout and I/O Request (6 ins)
6 100
7 101
8 102
9 103
10 8000
11 8001
12 8002
13 8003
----------- I/O Request (7 ins)
14 100
15 101
16 102
17 103
18 12000
19 12001
----------- I/O Request (3 ins)
20 100
21 101
22 102
23 103
24 5005
----------- I/O Request (4 ins)
25 100
26 101
27 102
28 103
29 8004
30 8005
31 8006
----

32 100
33 101
34 102
35 103
36 12002
37 12003
38 12004
39 12005
40 12006
----------- I/O Request (1 ins)
41 100
42 101
43 102
44 103
45 5006
46 5007
47 5008
----------- I/O Request (5 ins)
48 100
49 101
50 102
51 103
52 12007
53 12008
----------- I/O Request (2 ins)
54 100
55 101
56 102
57 103
58 5009
59 5010
60 5011
61 5012
62 5013
----------- Timeout
63 100
64 101
65 102
66 103
67 12009
68 12010

69 12011
70 12012
----------- I/O Request
71 100
72 101
73 102
74 103
75 5014
----

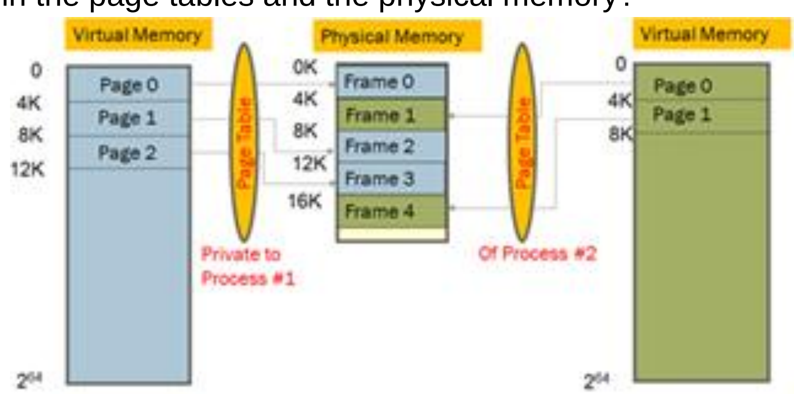
3. [5 pts] What is the difference between the "New" state and the "Ready to Run" state in the process state diagram?

   The “New” means the PCB for the process was created, but the executable image has not been loaded into the memory. While “Ready to Run” means that the executable image has been loaded and the program can be executed.

4. [5 pts] Is a transition from the "Blocked" state to directly to the "Exit" state possible in the process state diagram? How?

   Yes, it is possible. For example, parent process killed the child while it was waiting.

5. [10 pts] Assume that the following physical memory is full with already allocated 5 pages as shown below (i.e., it is 20KB in capacity). Describe what happens if process 2 wants to allocate and use another page. What changes in the page tables and the physical memory?

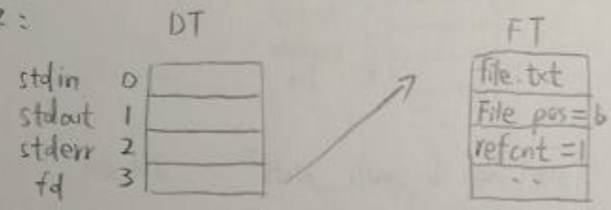


   When process 2 wants to allocate and use another page, the Page Fault happens. The fault-handler allocates the required page into the physical memory. The Least Recently Used frame will be evicted to the disk. Meanwhile, the corresponding page table will be updated.

6. [25 pts] Draw the Descriptor Table (DT) and File Table (FT) for each possible process just after executing the lines 2, 5 and 10 and explain what you see in the standard output and **file.txt** along the way. FT entries should contain the `cursor` and the `reference count`

```
1.int fd=open("file.txt", O_CREAT|O_RDWR);
2.write (fd, "Hello world", 6);
3.if (!fork()){
4.  dup2 (fd, 1); //redirect stdout
5.  cout << "Mars, I am here!!" << endl;
6.}else{
7.  wait(0);
8.  lseek (fd, 0, SEEK_SET);
9.  cout << "Mars is great" << endl;
10. write (fd, "Hola  ", 5); //draw tables
11. close (fd);
12.}
```

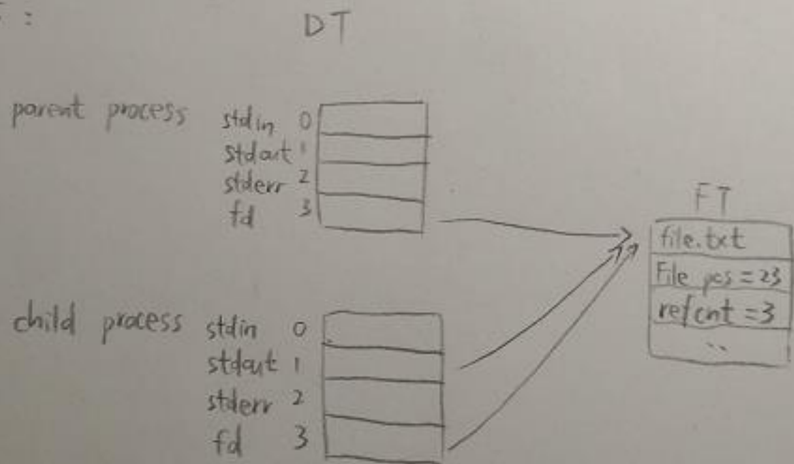
Line 2 :



file.txt : "Hello "

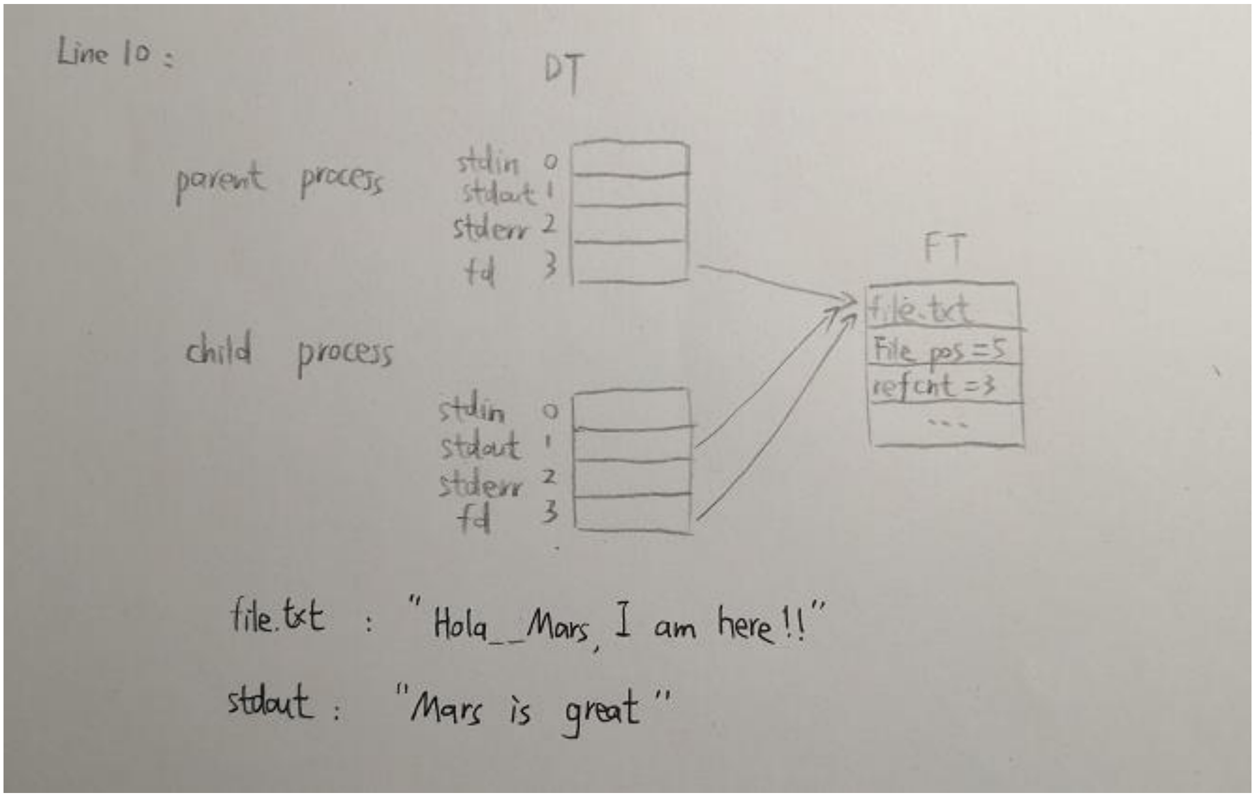
stdout : nothing

Line 5 :



file.txt : "Hello Mars, I am here !!"

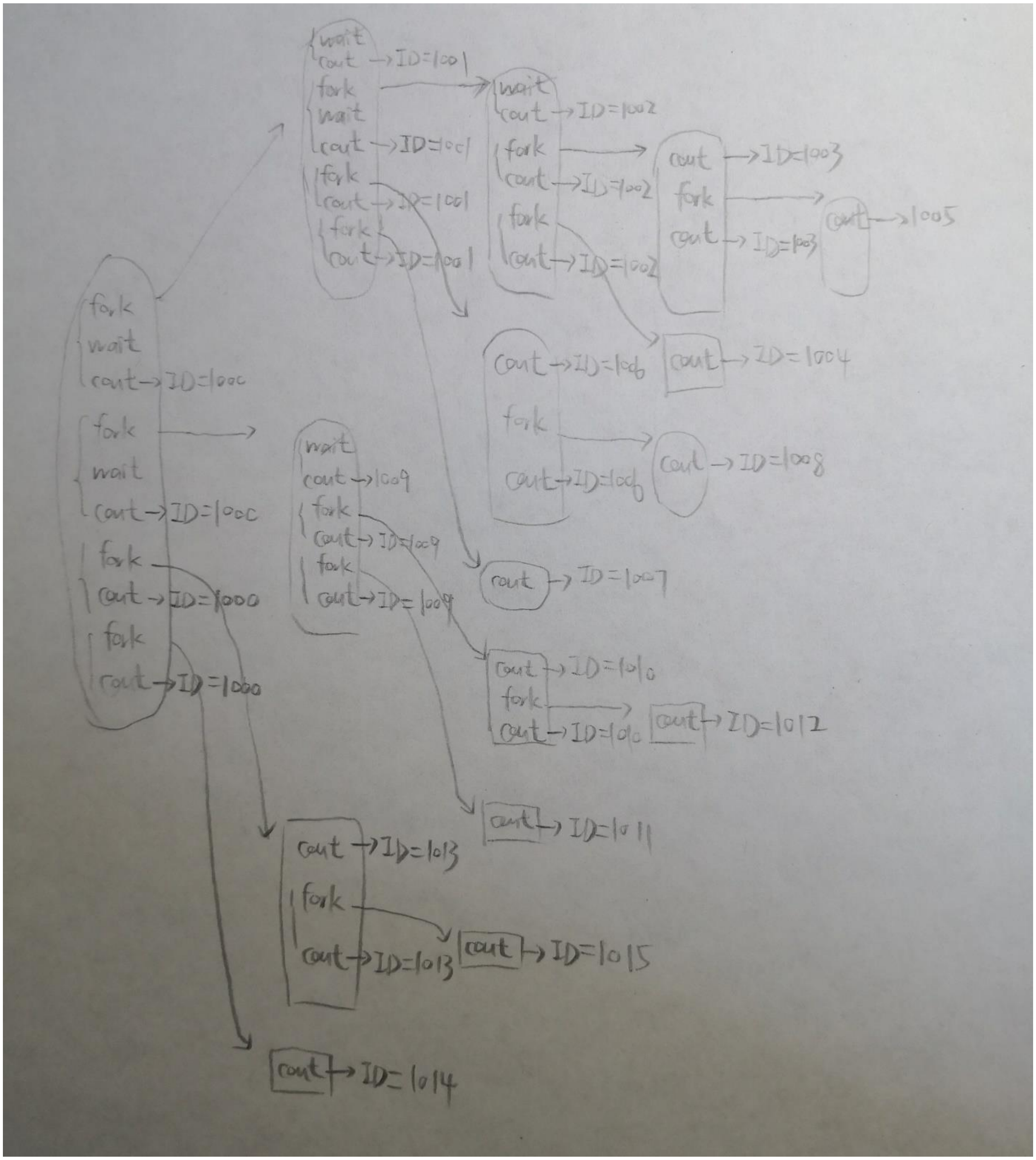
in child process, stdout points to file.txt



7. [25 pts] Please explain the output of the following program using a process tree diagram as shown in Lecture 4 page 23. Assuming the main process's ID is 1000, explain the order of process creation (in fact, the number itself should show what order the processes are created). Also explain what you see in the standard output. Are different outputs possible for this program? Why or why not?

```
for (int i=0; i<4; i++){
    int cid = fork ();
    if (i < 2)
        wait (0);
    cout << "ID=" << getpid () << endl;
}
```

If there is a wait between two forks, all the child processes of the former fork will print out their pid numbers before the latter fork.
If there is no wait between two forks, the order of pid numbers being printed out in different layers of fork is undetermined. Since the system scheduler algorithm is different.
Different outputs are possible for this program in different systems because of the different scheduler algorithm.


fork
wait
cout → ID=1000
fork
wait
cout → ID=1000
fork
cout → ID=1000
fork
cout → ID=1000
wait
cout → ID=1001
fork
wait
cout → ID=1001
fork
cout → ID=1001
fork
cout → ID=1001
wait
cout → ID=1002
fork
cout → ID=1002
fork
cout → ID=1002
cout → ID=1003
fork
cout → ID=1003
cout → 1005
cout → ID=1004
cout → ID=1006
fork
cout → ID=1006
cout → ID=1008
wait
cout → 1009
fork
cout → ID=1009
fork
cout → ID=1009
cout → ID=1007
cout → ID=1010
fork
cout → ID=1010
cout → ID=1012
cout → ID=1011
cout → ID=1013
fork
cout → ID=1013
cout → ID=1015
cout → ID=1014