4.1 | (5pts) Consider the situation that there are two processes waiting in the ready queue, P1 and P2. The current process is P0. P0 calls fork() and creates a child process P0C and P0 continue to run without giving up the CPU. What does the ready queue look like at this time?

Ready Queue

P1 P2 P0C

Ex. 4.2 |(5pts) Continuing the scenario in Exercise 4.1, P0 calls waitpid() so it gets blocked. Now P1 runs. What are the processes in the ready queue? Is P0 in the ready queue? If not, where is it?

Ready Queue

P2 P0C

No, the P0 is right in the Wait Queue

Wait Queue

P0

Ex. 4.3 | (5pts) The yield() system calls also let the current process give up the CPU as waitpid(). Discuss differences between the two.

Yield() is to let the current process give up the CPU. But Yield() does not wait for anything except available CPU. It will make the current process come to the back of the ready queue. For waitpid() is to block the current process until the process which the current process is waiting for finishes its work. Before finishing, the current processor is in the wait queue. For example, if the process A calls waitpid(1). Before the No1's processor finishes its work A will be blocked in the wait queue. After finishing, A will come to the ready queue. The blocked process will come to the ready queue again from the wait queue.

Ex. 4.4 | (15pts) Under a Unix/Linux environment, type the following program and save it as *MyThreadExampleOneV.java*. Compile the program and run it multiple times. Observe the outputs and explain the outputs. See the Ex.4.4.png.

```
root@user1-VirtualBox:/home/user1/cis657/hw4# java MyThreadExampleOneV
Thread 1 haswritten 1 to Global.Variable
Thread 2 haswritten 2 to Global.Variable
Thread 3 haswritten 3 to Global.Variable
Global Variable=3
Thread 4 haswritten 4 to Global.Variable
root@user1-VirtualBox:/home/user1/cis657/hw4# java MyThreadExampleOneV
Thread 1 haswritten 1 to Global.Variable
Thread 2 haswritten 2 to Global.Variable
Thread 3 haswritten 3 to Global.Variable
Global Variable=3
Thread 4 haswritten 4 to Global.Variable
root@user1-VirtualBox:/home/user1/cis657/hw4# java MyThreadExampleOneV
Thread 1 haswritten 1 to Global.Variable
Thread 2 haswritten 2 to Global.Variable
Thread 3 haswritten 3 to Global.Variable
Global Variable=3
Thread 4 haswritten 4 to Global.Variable
root@user1-VirtualBox:/home/user1/cis657/hw4#
```

I tried to run the code many times on my own virtual machine using ubuntu16.01. There is no difference between each running output. However, the 5 threads should execute in a random order, due to no waitpid() or yield() function in the code. So the multiple threads should race to execute certain program instructions ,producing a random result. And the print Global variable may execute randomly in the main thread rather than just printing after thread 3. I think the output should like the following, which is my windows environment's result

```
C:\WINDOWS\system32\cmd.exe

F:\CIS657\hw4>java MyThreadExampleOneV
Global Variable=-100
Thread 2 haswritten 2 to Global.Variable
Thread 1 haswritten 1 to Global.Variable
Thread 3 haswritten 3 to Global.Variable
Thread 4 haswritten 4 to Global.Variable

F:\CIS657\hw4>java MyThreadExampleOneV
Thread 1 haswritten 1 to Global.Variable
Thread 4 haswritten 4 to Global.Variable
Global Variable=1
Thread 2 haswritten 2 to Global.Variable
Thread 3 haswritten 3 to Global.Variable

F:\CIS657\hw4>java MyThreadExampleOneV
Thread 1 haswritten 1 to Global.Variable
Thread 3 haswritten 3 to Global.Variable
Thread 2 haswritten 2 to Global.Variable
Global Variable=4
Thread 4 haswritten 4 to Global.Variable

F:\CIS657\hw4>java MyThreadExampleOneV
Global Variable=-100
Thread 2 haswritten 2 to Global.Variable
Thread 4 haswritten 4 to Global.Variable
Thread 1 haswritten 1 to Global.Variable
Thread 3 haswritten 3 to Global.Variable

F:\CIS657\hw4>
```

Ex. 4.5 | (20pts) Use the Banker's algorithm to find a safe state/safe sequence of execution for the following case. P0-P4 are five processes. A-D are four different type of resources.

	HAS	MAX	FREE

	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2
P1	1	0	0	0	1	7	5	0
P2	1	3	5	4	2	3	5	6
P3	0	6	3	2	0	6	5	2
P4	0	0	1	4	0	6	5	6

A	B	C	D
1	5	2	0

So the needed resource for those 4 processors is like below.

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Using the Banker's algorithm,
P0 can meet the resource with current free resource.
So P0 is done.
And the free resource will be:

A	B	C	D
1	5	3	2

The free resource can't meet the P1's needed resource, but can meet the P2's resource.
After P2 is done.

A	B	C	D
2	8	8	6

The free resource can meet the requirement of P3.
After P3 is done.

A	B	C	D
2	14	11	8

The free resource can meet the requirement of P4.
After P4 is done.

A	B	C	D
2	14	12	12

The free resource can meet the requirement of P1.
After P1 is done.

A	B	C	D
3	14	12	12

So the P0->P2->P3->P4->P1 is the result.

Ex. 4.6| (30pts) Understanding Fork(), Exec*(), WaitPid(), and Exit() system calls.

- Download exec-example0.c, exec-example1.c, exec-example2.c, to the linux machine.
- Compile each program.
    - To compile a C program, you can use the command: gcc -o <executable file name> <source file name>.
    - For example, to compile exec-example0.c, you can use: gcc -o exec-example0 exec-example0.c
    - This will generate a binary executable file: exec-example0
- Run exec-example0; What is the output? Explain the output.

The following is the output.

```
hw4> ./exec-exemple0
.   exec-example0.c  exec-example2.c  exec-exemple1  MyThreadExampleOneV.java
..  exec-example1.c  exec-exemple0    exec-exemple2
```

After running exec-exemple0 file, we will output all the files in the same folder. The ls -a instruction will let the system output all the files in this directory.

- Run exec-example1; What is the output? Explain how the output was generated by the program.

The following is the output.

```
hw4> ./exec-exemple1
########## I am THE parent!!!!!!!!!!!!!!
hw4> .   exec-example0.c  exec-example2.c  exec-exemple1  MyThreadExampleOneV.ja
va
..  exec-example1.c  exec-exemple0    exec-exemple2
./exe./exec-exemple2
```

When running exec-exemple1 file, the parent firstly creates a child process using fork(). Due to parent pid not equals to 0, it will output the ######## I am THE parent!!!!!!! regardless of the child process. Then the process pid is 0, then output the ls -a instruction's output, which is the output of all the files in this folder.

- Run exec-example2; What is the output. How is this output different from the output of exec-example1? Why are they different?

The following is the output:

```
./exe./exec-exemple2
.   exec-example0.c  exec-example2.c  exec-exemple1  MyThreadExampleOneV.java
..  exec-example1.c  exec-exemple0    exec-exemple2
$$$$$$ I am THE parent!!!!!
```

When running exec-exemple2 file, the parent firstly creates a child process using fork(). Due to the parent pid not equal to 0, it will execute waitpid(pid, &status, 0);, waiting for the child process to finish its work, then will output the $$$$$$ I am THE parent!!!!!..

- What to submit: Answers to 3, 4, and 5. (10pts each)

Ex. 4.7 | (20pts) Write pseudocode to solve the dining philosopher problem. More importantly, explain whether your code will cause deadlock or not, and why. (Code: 10pts; explanation: 10pts)

In the dining philosopher problem, the shared five forks are resources. Suppose the initial value of fork is 1, which is stored in a semaphore array. And numbers everyone from 0 to 4; five forks and their corresponding signal numbers 0~4, and the forks with the same number are located at the left hand of the philosopher . The thought is to take the forks on the left of each one first, and then take the forks that it has, and eat after taking two forks; after the meal, two forks are released at a time.
The dining philosopher problem pseudocode

```
semaphore fork[5];
void main(){
    fork[5]={1,1,1,1,1};
    parbegin(Philosopher(i)(i=0...4));
}

void Philosopher(i){
    do{
        thinking;
        wait(fork[i]);
        wait(fork[i+1]mod5);
        eating;
        signal(fork[i]);
```

```
        signal)fork[i+1]mod5);
      }while(1);
}
```

However, if five philosophers succeed in taking the forks on the left at the same time, they will fail when taking the forks on the right later. The five philosophers all waited for the forks on the right side and could not eat, and they were unable to release the forks on the left that had been taken, and led to a deadlock.

One modified version of the pseudocode. Up to four forks are allowed to be picked up at the same time, so as to ensure a successful two forks

```
semaphore fork[5];
semaphore take;
void main(){
    fork[5]={1,1,1,1,1};
    take=4;
    parbegin(Philosopher(i)(i=0...4));
}


void Philosopher(i)
{
    do{
        thinking;
        wait(take); //wait for fork
        wait(fork[i]); //required the left hand forks
        wait(fork[(i+1)mod5]); //required the right hand fork
        eating;
        signal(fork[(i+1)mod5]); //release the right hand fork
        signal(fork[i]); //release the left hand fork
        signal(take);
    }while(1);
}
```

Only when the forks on the left and right sides of the philosopher are available, he is allowed to pick up two forks at the same time.

```
semaphore fork[5];
void main(){
    fork[5]={1, 1, 1, 1, 1};
    parbegin(Philosopher(i)(i=0...4));
}

void Philosopher(i)
{
    do {
        thinking;
        Swait(fork[(i+1)]mod5,fork[i]);
        eating;
        signal(fork[(i+1)]mod5,fork[i]);
```

```
    }while(1);
}
```

Only the odd-numbered philosophers take their left fork first, and then their right fork after success; while even-numbered philosophers take their right fork first, and then take their left fork after success. This will ensure that a philosopher can get two forks

```
semaphore fork[5];
void main(){
   fork[5]={1, 1, 1, 1, 1};
   parbegin(Philosopher(i)(i=0...4));
}

void Philosopher(i)
{
   do{
      thinking;
      if(imod2 == 0) //even-numbered first right then left
      {
         wait(fork[i+1]mod5) ;
         wait(fork[i]) ;
         eating;
         signal(fork[i+1]mod5) ;
         signal(fork[i]) ;
      }
      else //odd-numbered first left then right
      {
         wait(fork[i]) ;
         wait(fork[i+1]mod 5) ;
         eating;
         signal(fork[i]) ;
         signal(fork[i+1]mod 5) ;
      }
   }while(1);
}
```