题号：V1

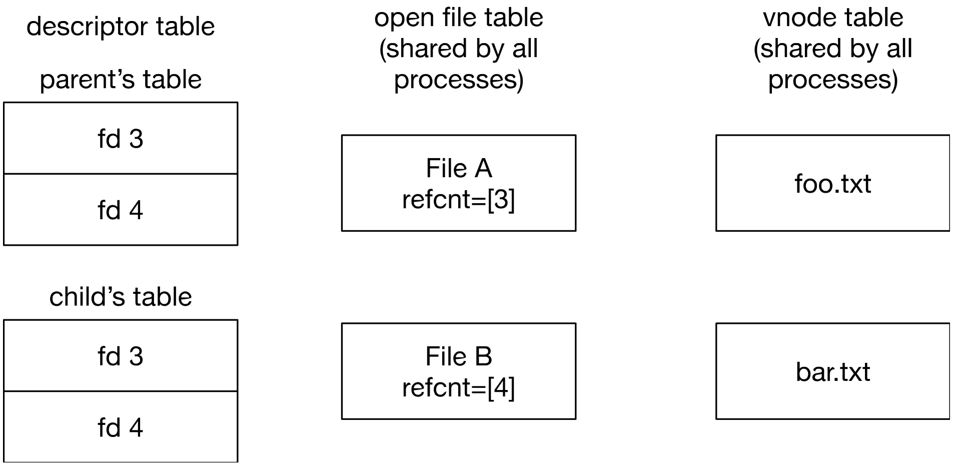
IO1 (16 points)

```
#include <csapp.h>

int main(void) {
        int fd1, fd2;
        int pid;
        char c[4] = "ics";
        fd1 = Open("foo.txt", O_RDWR, 0);
        fd2 = Open("bar.txt", O_RDWR, 0);
        Write(fd2, c, 2);
        Dup2(1, fd2);

        if (Fork() == 0) {
                Write(fd2, c, 1);
                Read(fd1, c, 1);
                Write(fd1, c, 2);
                Write(fd2, c, 2);
                Dup2(fd1, fd2);
        }
        else
                Wait(NULL);

        Write(fd1, c, 2);
        Write(fd2, c, 2);

        return 0;
}
```

NOTE: Initially, `foo.txt` contains "`online-exam`"; `bar.txt` contains **"`easy`"**; No error occurs in the execution. **NOTE**: suppose that read and write operations are atomic.

1. Please write down the output on **screen**. (2′)
2. Right before the child process exit, please fill in the blanks in the **figure and table** below, like **Figure 10.12** in your text book. (NOTE: fd 0, 1, 2 are ignored, you need to complete **4 blanks**) (4′)

descriptor table	**open file table**	**vnode table**
fd=3	File A	foo.txt
fd=4	**[1]**	**[2]**



3. Please write down the **contents** of `foo.txt` and `bar.txt`. (6′)

4. If we change **line 13** from `Write(fd2, c, 1)` to `printf(“%c”, c[0]),` write down the output on screen. (2′) (**Hint**: `printf` has a buffer) If we want the same output as before, how to modify the code? (NOTE: you can't modify `printf`) (2′)

题号：V2

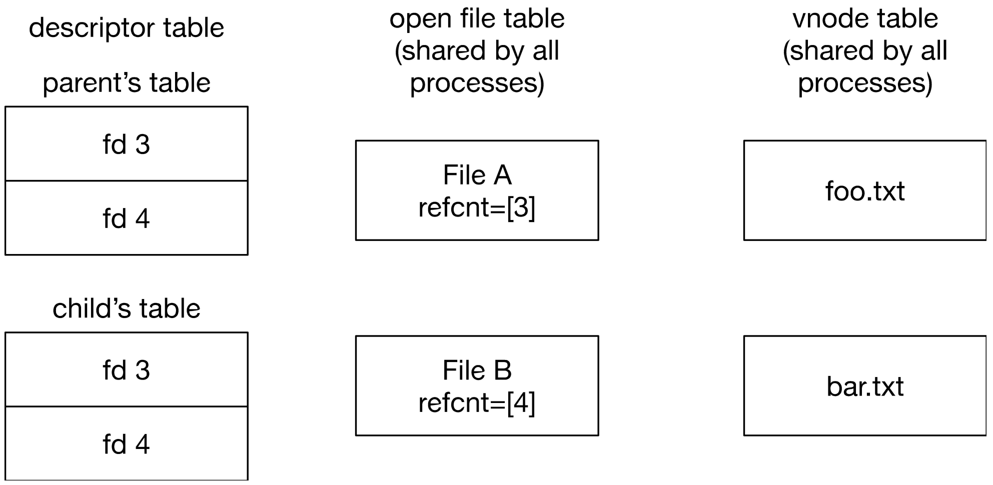
IO2 (16 points)

```
#include <csapp.h>

int main(void) {
        int fd1, fd2;
        int pid;
        char c[4] = "NOP";
        fd1 = Open("foo.txt", O_RDWR, 0);
        fd2 = Open("bar.txt", O_RDWR, 0);
        Write(fd2, c, 2);
        Dup2(1, fd2);

        if (Fork() == 0) {
                Write(fd2, c, 1);
                Read(fd1, c, 1);
                Write(fd1, c, 2);
                Write(fd2, c, 2);
                Dup2(fd1, fd2);
        }
        else
                Wait(NULL);

        Write(fd1, c, 2);
        Write(fd2, c, 2);

        return 0;
}
```

NOTE: Initially, **foo.txt** contains "**ICS2020**"; **bar.txt** contains **"SJTU"**; No error occurs in the execution. **NOTE**: suppose that read and write operations are atomic.

1. Please write down the output on **screen**. (2′)
2. Right before the child process exit, please fill in the blanks in the **figure and table** below, like **Figure 10.12** in your text book. (NOTE: fd 0, 1, 2 are ignored, you need to complete **4 blanks**) (4′)

descriptor table	open file table	vnode table
fd=3	File A	foo.txt
fd=4	**[1]**	**[2]**



3. Please write down the **contents** of **foo.txt** and **`bar.txt`**. (6′)

4. If we change **line 13** from **`Write(fd2, c, 1)`** to **`printf("%c", c[0]),`** write down the output on screen. (2′) (**Hint**: **`printf`** has a buffer) If we want the same output as before, how to modify the code? (NOTE: you can't modify **`printf`**) (2′)

题号：W1

Schedule (16 points)

We have following jobs in the workload. No I/O issues are involved.

Job	Arrival Time	Length of Run-time
A	0ms	8ms
B	4ms	4ms
C	6ms	6ms
D	10ms	4ms

- The **RR** time-slice is 2**ms**.
- Suppose when a job arrives, it is added to the tail of a work queue. The **RR** policy selects the next job of the current job in the queue.

1. Please calculate the average turnaround time and average response time for various scheduling policies. (8*1′ = 8′)

Job	**Average Turnaround Time**	**Average Response Time**
FIFO	**[1]**	**[2]**
SJF	**[3]**	**[4]**
STCF	**[5]**	**[6]**
RR	**[7]**	**[8]**

2. We decide to use **MLFQ** scheduling policy now. The workload remains the same.

- There are **2 priority queues**. The highest one has time-slice of **1 ms**, the lowest one has time-slice of **2ms**.
- **RR** is used in each queue.
- The CPU scheduling is carried out only at **completion** of processes or time-slices.

Following table shows the execution of CPU. No I/O issues are involved. Please fill in the blanks (8*1′=8′).

Time	0	1	2	3	4	5	6	7	8	9	10
CPU	A	A	A	A	**[1]**	**[2]**	**[3]**	B	B	C	**[4]**
Time	11	12	13	14	15	16	17	18	19	20	21
CPU	**[5]**	D	D	A	A	B	**[6]**	C	D	A	C

Average turnaround time: __**[7]**__ ms

(HINT: **DON'T** forget the last executing time-slice when calculating turnaround time).

Average response time: __**[8]**__ms

题号：W2

Process (16 points)

Assume we have the following codes. We have following jobs in the workload. No I/O issues.

```
// headers are omitted


int num = 3;

int main(void) {

    for (int i = 0; i < 2; ++i) {

        pid_t pid = fork();

        if (pid == 0) {

            num++;

            printf(“child with i: %d, num: %d\n”, i, num);

        } else {

            waitpid(pid, NULL, 0);

            printf(“parent with i: %d, num: %d\n”, i, num);

        }

        num <<= 1;

    }



    return 0;

}
```

1. How many times will **`fork()`** be called in this program? How many times will **`printf()`** be called in this program(2*2′ = 4′)?

2. If **`waitpid()`** in line 11 is removed, will the times that **`fork()`** is called change (2′)?

3. Please give a possible output of the program. You can use **`[<P/C>,<i>,<num>]`** **`(e.g., [C,0,4])`** to represent each printed line for simplicity. And you can write your answer in one line and neglect **`'\n'`** (6′).

4. Please explain constrains that **`waitpid()`** puts on the output of the program(4′).

题号：X1

Signal1 (18 points)

```
#include <csapp.h>
#define MAX 2
#define SEC 4
int cnt = 0;
void handler(int n) {
        if (cnt < MAX) {
                cnt++;
                Alarm(2);
                Fork();
                printf("%d\n", cnt);
        }
}

int main(void) {
        Signal(SIGALRM, handler);
        Alarm(2);
        for (int i = 0; i < SEC; i++) {
                Sleep(1);
                printf("S%d\n", i+1);
        }
        return 0;
}
```

One of the TAs wrote a program as shown above. NOTE:

- Assume all system calls are successful.
- A child created via `fork` inherits a copy of its parent's set of currently blocked signals.
- A child created via `fork` initially has an **empty** pending signal set.
- `alarm` before `fork` will **only** send signal to parent process later.
- We assume that the execution time of `sleep(1)` is strictly 1 second, the same for `alarm`.

1. Please write one possible output on the screen. (Note: you don't have to fill out all the blanks) (3')

Line1	Line2	Line3	Line4	Line5	Line6	Line7	Line8
S1							
Line9	**Line10**	**Line11**	**Line12**	**Line13**	**Line14**	**Line15**	**Line16**

2. Suppose **MAX** is **4** and **SEC** is **5** in this question.
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)
3. Suppose **MAX** is **4** and **SEC** is **10** in this question.
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)
4. Suppose we exchange the codes in **line 8** and **line 9** in this question (Note: **MAX** is **2** and **SEC** is **4**).
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)
   c) Is there any race modifying the global variable cnt between the parent process and its child process(es), why? (3')

题号：X2

Signal2 (18 points)

```
#include <csapp.h>
#define MAX 2
#define SEC 4
int cnt = MAX;
void handler(int n) {
        if (cnt > 0) {
                cnt--;
                Alarm(2);
                Fork();
                printf("%d\n", cnt);
        }
}

int main(void) {
        Signal(SIGALRM, handler);
        Alarm(2);
        for (int i = 0; i < SEC; i++) {
                Sleep(1);
                printf("S%d\n", i+1);
        }
        return 0;
}
```

One of the TAs wrote a program as shown above. NOTE:

- Assume all system calls are successful.
- A child created via **fork** inherits a copy of its parent's set of currently blocked signals.
- A child created via **fork** initially has an **empty** pending signal set.
- **alarm** before **fork** will **only** send signal to parent process later.
- We assume that the execution time of **sleep(1)** is strictly 1 second, the same for **alarm**.

1. Please write one possible output on the screen. (Note: you don't have to fill out all the blanks) (3')

Line1	Line2	Line3	Line4	Line5	Line6	Line7	Line8
S1							
Line9	**Line10**	**Line11**	**Line12**	**Line13**	**Line14**	**Line15**	**Line16**

2. Suppose **MAX** is **4** and **SEC** is **6** in this question.
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)

3. Suppose **MAX** is **5** and **SEC** is **9** in this question.
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)

4. Suppose we exchange the codes in **line 8** and **line 9** in this question (Note: **MAX** is **2** and **SEC** is **4**).
   a) How many lines of output are on the screen when this program ends? (2')
   b) How many processes are there before the end of the program? (2’)
   c) Is there any race modifying the global variable cnt between the parent process and its child process(es), why? (3')

题号：Y1

Network1 (24 points)

Your task is to design a simple system to get your lab grades. This system consists of one server and several clients. The hostname of the server is **`ipads.se.sjtu.edu.cn`**, and the port number is **`8080`**.

Suppose you want to query your grades of one of the labs. Firstly, you need to send a request to get the lab lists that you have handed in. Then you will know which lab's grade you can query about. Finally, you need to send a request again to get the lab's grade.

The detailed communication protocol between the server and the clients is defined as follows:

1. A client sends the **`LABHISTORY`** request to get the submission history of student with id "**`518xxxxxxxx`**". The password of this student must be provided, and is simply transferred in plaintext:

   `"LABHISTORY" '\n' "518xxxxxxxx" '\n' "myPassword" '\n'`

2. The server receives the **`LABHISTORY`** request and returns the following messages containing the submission history of this student:

   `"lab1" '\n' "lab2" '\n' "lab3" '\n' "lab4" '\n' ... "lab10" '\n' "END LABHISTORY" '\n'`

   If the password is incorrect or the student id does not exist, the server simply returns error message:

   `"INCORRECT ACCOUNT\n"`

3. If the server returns error message, or the name of the lab provided by the user does not appear in the lab history list, then client prints corresponding error message:

   `"Incorrect account!\n"`, or

   `"Invalid labname!\n"`

4. If the name of the lab is found in the lab history list, the client will send another **`GRADE`** request to get the grade:

   `"GRADE" '\n' "lab3" '\n' "518xxxxxxxx" '\n' "myPassword" '\n'`

5. The server receives the **`GRADE`** request and returns the corresponding grade:

   `"95\n"`

   We do not consider incorrect account in this step, since it's already checked.

Examples:

1] Command: ./getGrade lab3 51800001111 myPassword

Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’ “myPassword” ‘\n’

Server => Client, sends response: “lab1” ‘\n’ “lab2” ‘\n’ “lab3” ‘\n’ “lab4” ‘\n’ “END LABHISTORY” ‘\n’

Client => Server, sends request: “GRADE” ‘\n’ “lab3” ‘\n’ “51800001111” ‘\n’ “myPassword” ‘\n’

Server => Client, sends response: “95\n”

Client prints “95\n”

2] Command: ./getGrade lab6 51800001111 myPassword

Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’ “myPassword” ‘\n’

Server => Client, sends response: “lab1” ‘\n’ “lab2” ‘\n’ “lab3” ‘\n’ “lab4” ‘\n’ “END LABHISTORY” ‘\n’

Client prints “Invalid labname!\n”

3] Command: ./getGrade lab3 51800001111 wrongPass

Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’ “wrongPass” ‘\n’

Server => Client, sends response: “INCORRECT ACCOUNT\n”

Client prints “Incorrect account!\n”

Please complete the following client-side code. **Hint**: you can use `sprintf` to construct the requests. Also you can use `sscanf` to parse the messages from the server.

```
#include "csapp.h"


/* the program is executed by ./queryInfo <labname> <studentid> <password> */
int main(int argc, char **argv) {
    char *labname = argv[1];
    char *stu_id   = argv[2];
    char *pass       = argv[3];
    char send_buf[1024], recv_buf[1024];
    memset(send_buf, 0, 1024);
    memset(recv_buf, 0, 1024);

    /* connect to <hostname>:<port> and send out the request */
    [1] // Write your code here (5')


    /* read the labhistory information from server */
    /* check the lab's existence */
    [2] // Write your code here (9')


    /* send GRADE request */
    [3] // Write your code here (5')
    /* finally get the grade information
        and print to the screen */
    [4] // Write your code here (5’)


    return 0;
}
```

题号：Y2

Network (24 points)

Your task is to design a simple system to get your lab grades. This system consists of one server and several clients. The hostname of the server is **`ipads.se.sjtu.edu.cn`**, and the port number is **`8888`**.

Suppose you want to query your grades of one of the labs. Firstly, you need to send a request to get the lab lists that you have handed in. Then you will know which lab's grade you can query about. Finally, you need to send a request again to get the lab's grade.

The detailed communication protocol between the server and the clients is defined as follows:

1. A client sends the **`LABHISTORY`** request to get the submission history of student with id "**`518xxxxxxxx`**". The password of this student must be provided, and is simply transferred in plaintext:

   `"LABHISTORY" '\n' "518xxxxxxxx" '\n' "myPassword" '\n'`

2. The server receives the **`LABHISTORY`** request and returns the following messages containing the submission history of this student:

   `"LABHISTORY BEGIN" '\n' "lab1" '\n' "lab2" '\n' "lab3" '\n' "lab4" '\n' ... "lab10" '\n' "LABHISTORY END" '\n'`

   If the password is incorrect or the student id does not exist, the server simply returns error message:

   `"INCORRECT ACCOUNT\n"`

3. If the server returns error message, or the name of the lab provided by the user does not appear in the lab history list, then client prints corresponding error message:

   `"Incorrect account!\n"`, or

   `"Invalid labname!\n"`

4. If the name of the lab is found in the lab history list, the client will send another **`GRADE`** request to get the grade:

   `"GRADE" '\n' "518xxxxxxxx" '\n' "myPassword" '\n' "lab3" '\n'`

5. The server receives the **`GRADE`** request and returns the corresponding grade:

   `"95\n"`

   We do not consider incorrect account in this step, since it's already checked.

Examples:

```
1] Command: ./getGrade 51800001111 myPassword lab3

   Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’
   “myPassword” ‘\n’

   Server => Client, sends response: “LABHISTORY BEGIN” ‘\n’ “lab1” ‘\n’
   “lab2” ‘\n’ “lab3” ‘\n’ “lab4” ‘\n’ “LABHISTORY END” ‘\n’

   Client => Server, sends request: “GRADE” ‘\n’ “51800001111” ‘\n’
   “myPassword” ‘\n’ “lab3” ‘\n’

   Server => Client, sends response: “95\n”

   Client prints “95\n”

2] Command: ./getGrade 51800001111 myPassword lab6

   Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’
   “myPassword” ‘\n’

   Server => Client, sends response: “LABHISTORY BEGIN” ‘\n’ “lab1” ‘\n’
   “lab2” ‘\n’ “lab3” ‘\n’ “lab4” ‘\n’ “LABHISTORY END” ‘\n’

   Client prints “Invalid labname!\n”

3] Command: ./getGrade 51800001111 wrongPass lab3

   Client => Server, sends request: “LABHISTORY” ‘\n’ “51800001111” ‘\n’
   “wrongPass” ‘\n’

   Server => Client, sends response: “INCORRECT ACCOUNT\n”

   Client prints “Incorrect account!\n”
```

Please complete the following client-side code. **Hint**: you can use `sprintf` to construct the requests. Also you can use `sscanf` to parse the messages from the server.

```
#include "csapp.h"



/* the program is executed by ./queryInfo <labname> <studentid> <password> */

int main(int argc, char **argv) {

    char *stu_id   = argv[1];

    char *pass     = argv[2];

    char *labname = argv[3];

    char send_buf[1024], recv_buf[1024];

    memset(send_buf, 0, 1024);

    memset(recv_buf, 0, 1024);


    /* connect to <hostname>:<port> and send out the request */

    [1] // Write your code here (5')


    /* read the labhistory information from server */

    /* check the lab's existence */

    [2] // Write your code here (9')


    /* send GRADE request */

    [3] // Write your code here (5')
    /* finally get the grade information
        and print to the screen */
    [4] // Write your code here (5’)


    return 0;
}
```

题号：Z1

Lock1 (26 points)

Sam modifies the spin lock by adding one line “`IDLE(...);`” in the while-loop.

- `IDLE(n)` will consume *C*n* CPU cycles where *C* is a user-defined **constant**.
- `TAS` will **atomically test-and-set** the content of the target address
- When there are **multiple** threads **TAS** the same address concurrently, it will take **much longger** to finish this operation

The original spin lock	A modified spin lock

`typedef struct __lock_t {`
`    int flag;`
`} lock_t; // init to 0`
`void lock(lock_t *lock) {`
`    while (TAS(&lock->flag, 1))`
`        ;`
`}`
`void unlock(lock_t *lock) {`
`    lock->flag = 0;`
`}`
	`typedef struct __lock_t {`
	`    int flag;`
	`} lock_t; // init to 0`
	`void lock(lock_t *lock) {`
	`   while (TAS(&lock->flag, 1))`
	`       IDLE(...);`
	`}`
	`void unlock(lock_t *lock) {`
	`    lock->flag = 0;`
	`}`

1. Does this lock work **correctly**? Why or why not? (3’)

2. Can this lock guarantee the acquire **fairness**? Explain your answer through an example. (3’)

3. What’s the **advantage** after adding the “`IDLE(...);`” line? (3')

4. What’s the **disadvantage** after adding the “`IDLE(...);`” line? (3')
   HINT: Consider what if an inappropriate constant *C* is chosen.

5. Since **IDLE** still consume CPU time, can we use **park** instead? The modified version of spin lock is shown below. Will this lock work correctly? Explain your answer through an example. (**FAA** will **atomically fetch-and-add** the target address with the given value) (6’)

```
typedef struct __lock_t { int flag; int wait; } lock_t;
void lock(lock_t *lock) {
    while (TAS(&lock->flag, 1)) {
        FAA(&lock->wait, 1);
        park();
    }
}
void unlock(lock_t *lock) {
    lock->flag = 0;
    if (lock->wait) {
        FAA(&lock->wait, -1);
        unpark();
    }
}
```

6. Rather than **IDLE** for a certain time, can you come up with some better idea? Write down you own version of spin lock. (6’)

   (HINT: Since executing **TAS** concurrently will cause significant overhead, you can use **while** to wait on some condition before executing **TAS** again.)

题号：Z2

Lock2 (26 points)

Sam modifies the ticket lock by adding one line “`IDLE(...);`” in the while-loop.

- `IDLE(n)` will consume ***C*n*** CPU cycles where ***C*** is a user-defined **constant**.
- **`FAA`** will **atomically fetch-and-add** the content of the target address
- When there are **multiple** threads access the same address concurrently, it will take **much longger** to finish the memory operation

The original ticket lock

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;
void lock(lock_t *lock) {
    int myturn =
        FAA(&lock->ticket);
    while (lock->turn != myturn)
        ;
}
void unlock(lock_t *lock) {
    lock->turn ++;
}
```

A modified ticket lock

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;
void lock(lock_t *lock) {
   int myturn =
        FAA(&lock->ticket);
   while (lock->turn != myturn)
        IDLE(...);
}
void unlock(lock_t *lock) {
   lock->turn ++;
}
```

1. Does this lock work **correctly**? Why or why not? (3’)
2. Can this lock guarantee the acquire **fairness**? Why? (3’)
3. What’s the **advantage** after adding the “`IDLE(...);`” line? (3')
4. What’s the **disadvantage** after adding the “`IDLE(...);`” line? (3')
   HINT: Consider what if an inappropriate constant ***C*** is chosen.

5. Since **IDLE** still consume CPU time, can we use **park** instead? The modified version of spin lock is shown below. Will this lock work correctly? Explain your answer through an example. (**FAA** will **atomically fetch-and-add** the target address with the given value) (6’)

```
typedef struct __lock_t { int ticket; int turn; } lock_t;
void lock(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn) {
        park();
    }
}
void unlock(lock_t *lock) {
    lock->turn ++;
    if (lock->ticket > lock->turn) {
        unpark();
    }
}
```

6. Rather than asking all threads to **IDLE** for a same number of cycles, can you come up with some better idea? Write down you own version of ticket lock. (6’)