Name / ID (please PRINT) **Seq#:____** **Seat #:_____**

CS 3733.001 -- Operating system

Fall 2017 -- Midterm I -- Oct 5, 2017

You have 75 min. Good Luck!

- This is a **closed book/note** examination. But *You can use C reference card(s) given to you.*.
- This exam has 5 questions in 9 pages. Please read each question carefully and answer all the questions, which have 100 points in total. Feel free to ask questions if you have any doubts about questions.
- **Partial credit will be given, so do not leave questions blank**.

You can get **2pt bonus** credit if you complete the **boldfaced two columns** of the grading table below. Please do this after answering the questions in the exam. Thanks!

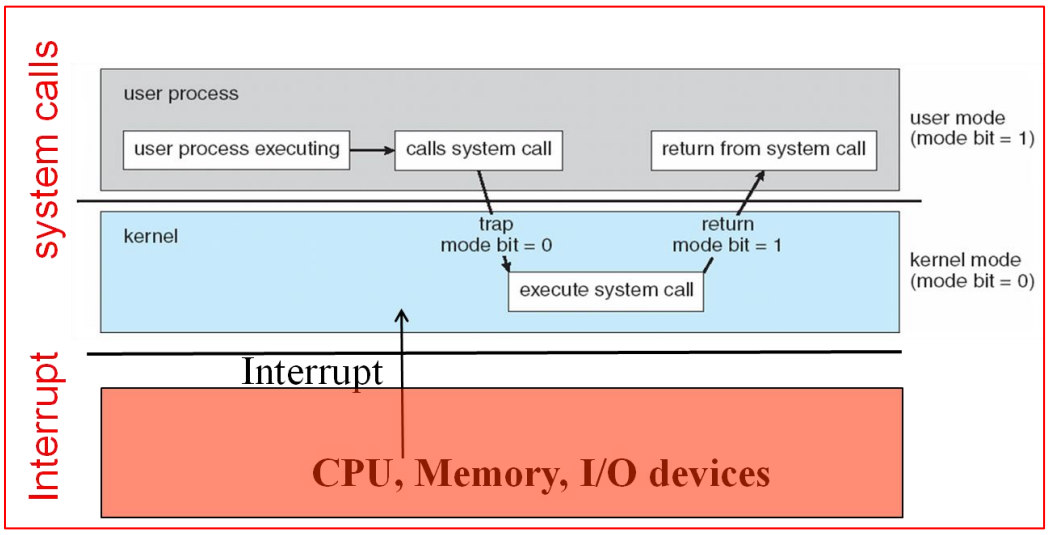
Question	Topic	Possible Points	**Difficulty level of this question 1: Easiest 5: Most difficulty**	**Student Expects**	Student Received
1	Review questions, short explanations	20			
2	Program and Process, static keyword, cmd line	20			
3	CPU Scheduling	20			
4	Unix I/O and file operations, fork/exec	20			
5	IPC: pipes -fifo	20			
	Bonus	2			
	Bonus for pipe quiz	4			
Total		100+6			

1. [**20 points]** Answer the following questions with short explanations. You can also draw figures to better explain your reasoning if needed.

   a. [2pt] What are the **goals** of Operating System? Briefly explain/discuss.

   Convenience: Make the computer convenient to use for general user and programmers..
   Efficiency: Manage system resources in an efficient manner

   b. [5pt] What are the **two key mechanisms** to interact with the Operating System kernel (1pt)? And explain <u>how they work</u> (each 2pt)?



System Calls	Interrupts
• User application calls a user-level library routine (gettimeofday(), read(), exec(), etc.) • Invokes system call through stub, which specifies the system call number. From unistd.h: #define __NR_getpid 172 __SYSCALL(__NR_getpid, sys_getpid) • This generally causes a **software interrupt**, trapping to kernel • Kernel looks up **system call number** in syscall table, calls appropriate function • Function executes and returns to interrupt handler, which returns the result to the user space process	1. The interrupt is issued 2. Processor finishes execution of current instruction 3. Processor signals acknowledgement of interrupt 4. Processor pushes PSW(*Program Status Word)* and PC to control stack 5. Processor loads new PC value through the interrupt vector 6. ISR saves remainder of the process state information 7. ISR executes 8. ISR restores process state information Old PSW and PC values are restored from the control stack

[if needed, here is extra space for question 1.b]

c. **[2pt]** What are the important information items stored in a generic Process Control Block (PCB) structure. List at least 4 items (each 0.5 pt).

Registers: in addition to general registers
- Program Counter (PC): contains the memory address of the next instruction to be fetched.
- Stack Pointer (SP): points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited.
- Program Status Word (PSW): contains the condition code bits and various other control bits

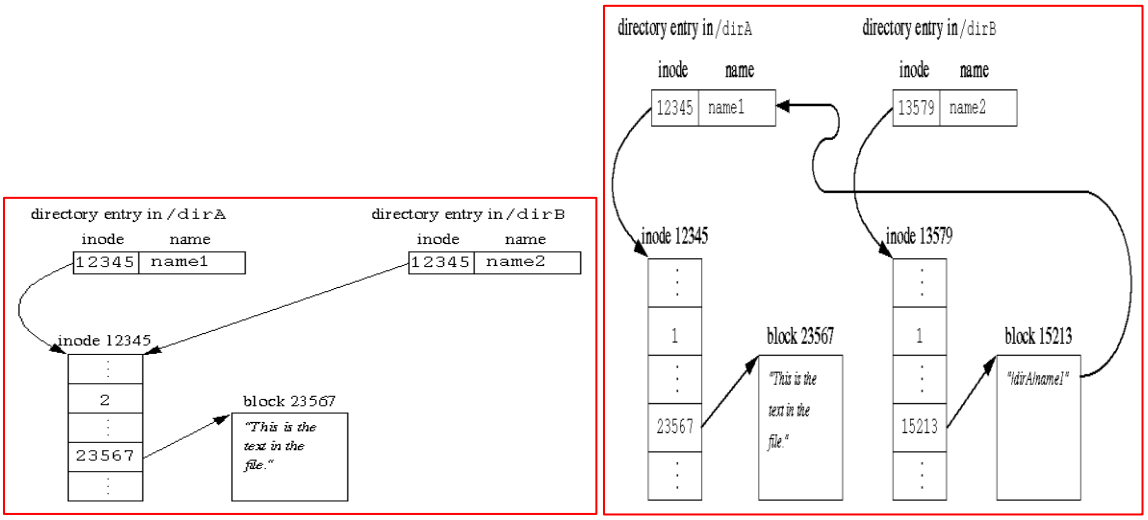
CPU scheduling information
Memory-management information
Accounting information
I/O status information
Thread synchronization and communication resource: semaphores and sockets

d. **[2pt]** Suppose you are asked to choose a CPU scheduling algorithm for a computing system where most of the programs are **interactive** applications. If you are asked to choose only one scheduling algorithm, which one would you choose (1pt)? And why (1pt)?

Round robin with relatively short quantum so that we can quickly serve each Interactive systems
Response/wait time: respond quickly to users' requests
Proportionality: meet users' expectations

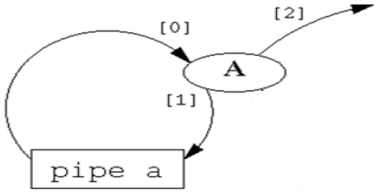
e. **[4pt]** There are two types of links in Unix/Linux: **Hard** and **Symbolic/Soft** links. Using diagrams explain the difference between them.

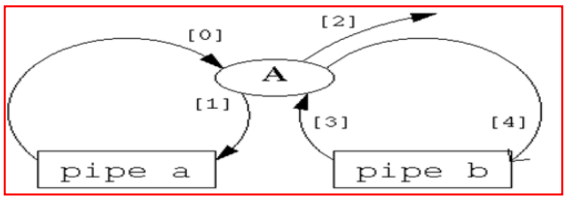
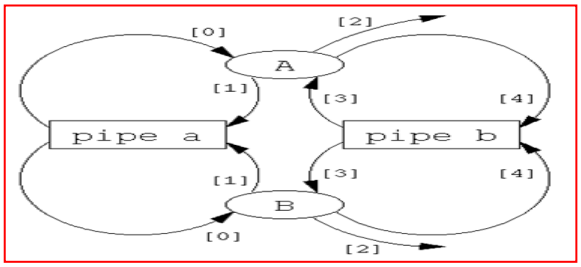
Hard link: A hard link just creates another file (a new entry in directory) with a link to the same underlying inode.
Symbolic/Soft link: link to another filename in the file system



f. [**5pt**] Suppose the following code sections are executed without any error. Draw the diagrams showing the relationship between the file descriptors, pipes and processes (first one is given).

Code section	Diagram after the execution of the code section

`int fda[2], fdb[2];`
`pipe(fda);`
`dup2(fda[0], STDIN_FILENO);`
`dup2(fda[1], STDOUT_FILENO);`
`close(fda[0]);`
`close(fda[1]);`
	

`pipe(fdb);` `// 2pt`	
`fork();` `// 3pt`	

2. [**20 points**] Program and Process, command-line arguments, `static` keyword , ...

a. [**10 points**] You are asked to implement a function `char *next_label();` whose consecutive calls will return labels like `"Fig. 1"`, `"Fig. 2"`, and so on. One possible solution is given below. However, it is not safe (e.g., when the second label is generated the first one is overwritten). Also it uses a global variable, which might be changed in other parts of the program. Re-implement this function by avoiding the use of a global variable and making it thread safe.

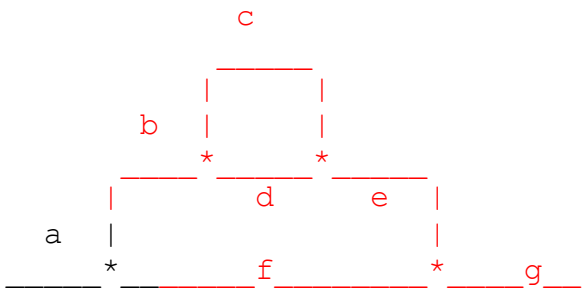
```
int count=0;
char *next_label()
{
   static char b[10];
   count++;
   sprintf(b,"Fig. %d", count);
   return b;
}
```

```
char *next_label()
{
   static int count=0;
   char *b;
   b = malloc(10);
   if (!b) exit();
   count++;
   sprintf(b,"Fig. %d", count);
   return b;
}
```

b. [**10 pt**] Show the relationship among the process created in the following program and give at least **two** possible outputs except `a, b, c, d, e, f, g.`

```
void main()
{
   printf("a\n");
   if (fork() == 0) {
      printf("b\n");
      if (fork() == 0) {
        printf("c\n");
        exit(0);
      }
      printf("d\n");
      wait(NULL);
      printf("e\n");
      exit(0);
   }
   printf("f\n");
   wait(NULL);
   printf("g\n");
   exit(0);
}
```

Complete the relationship diagram (6pt)



Two possible outputs (each 2pt)

a, f, b, c, d, e, g
a, b, f, c, d, e, g
a, b, c, f, d, e, g
a, b, c, d, f, e, g
a, b, c, d, e, f, g
c d can appear in different order

3. [20 Points] CPU Scheduling

a. [**12 points**] Consider two processes as in Assignment 2/Quiz 4, where each process has two CPU bursts with one I/O burst in between on a single core CPU. Suppose P1 and P2 have the following life-cycles:

P1 has x1=**6**, y1=**1**, z1=**2** units for the first CPU burst, I/O burst, second CPU burst, respectively.

P2 has x2=**8**, y2=**7**, z2=**3** units for the first CPU burst, I/O burst, second CPU burst, respectively.

Both arrives at the same time (in case of ties, pick P1) and there is no other processes in the system.

For each of the scheduling algorithms below, create process Gantt charts as you did for the Quiz 4. Fill each box with the state of the corresponding process. Use **R** for **R**unning, **w** for **W**aiting, and **r** for **r**eady. Calculate the waiting times and CPU utilization (as a fraction) for each process and fill in the table below.

SJF	P1: rrrrrrRRRRRRRRwwwwwwwwRRR P2: RRRRRRwrrrrrrrrRR	6	7	24	19/24	0.7917
PSJF	P1: rrrrrrRrrRRRRRRRwwwwwwwwRRR P2: RRRRRRwRR	8	0	26	19/26	0.7308

Gantt Charts for SJF (Shortest Job First, non-preemptive) [4pt]

a) SJF

	5				10				15				20				25				30							
P1																												
P2																												

Gantt Charts for PSJF (Preemptive SJF) [4pt]

b) PSJF

	5				10				15				20				25				30							
P1																												
P2																												

Waiting time and CPU utilizations [4pt]

Algorithm	Waiting times in ready queue			Finish time		Longest Schedule length	CPU utilization
	Process 1	Process 2	average	Process 1	Process 2		
b) SJF							
c) PSJF							

b. [**8 points**] Suppose we have a system using **multilevel queuing**. Specifically there are two queues and each queue has its own scheduling algorithm: QueueA uses RR with quantum 3 while QueueB uses RR with quantum 2. CPU simply gets processes form these two queues in a weighted round robin manner with 2:1 ratio (i.e. it gets **two** processes from QueueA then gets **one** process from QueueB, and then gets **two** processes from QueueA then gets **one** process from QueueB, and so on), But when it gets a process from QueueA, it applies RR scheduling with q=3. When it gets a process from QueueB, it applies RR wit q=2 scheduling.

Draw the Gantt charts (**5pt**) and compute waiting times (**3pt**) for the following four processes: P1, P2, P3, P4 on a single core CPU. Assume these processes arrived at the same time and in that order. Each process has a single CPU burst time of 5 units. There is no other processes or IO bursts.

	ratio		5						10						15						20			
QueueA RR q=3	2	P1	R	R	R						R	R												
		P2				R	R	R					R	R										
QueueB RR q=2	1	P3							R	R							R	R				R		
		P4													R	R			R	R			R	

Compute Waiting times in ready queue				
P1	P2	P3	P4	average
5	7	14	15	41/4= 10.25

4. [**20 points**] Unix I/O and file operations

a. [**10 points**] Suppose the following code is executed correctly without generating any errors, and parent's PID is 7 while child's PID is 8.

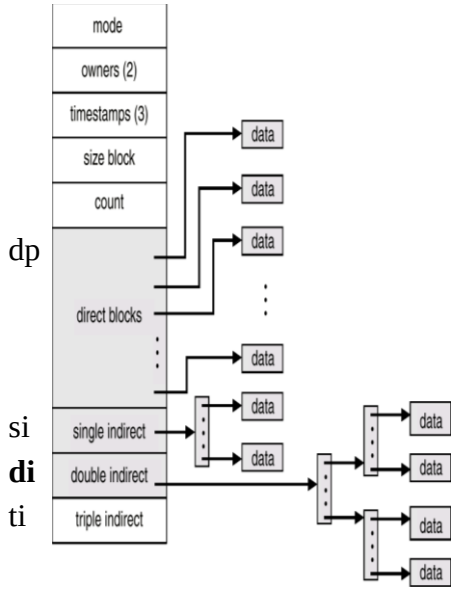
```
main(){
    fprintf(stdout, "%d: a ",                    getpid());
    fprintf(stderr, "%d: a has been written \n", getpid());
    fprintf(stdout, "%d: b \n",                  getpid());
    fprintf(stderr, "%d: b has been written \n", getpid());
    fprintf(stdout, "%d: c ",                    getpid());
    fork( );
    fprintf(stdout, "%d: all done! \n",          getpid());
    return 0;
}
```

Give a possible output for the above program.

```
7: a has been written
7: a 7: b
7: b has been written
7: c 7: all done
7: c 8: all done

Last two lines might change
```

b. [**10 points**] Consider the given INODE structure and assume that block size is 8K bytes and pointers are 4 bytes. So each block has **`num_ptr = 8*1024/4`** pointers! You are asked to implement **`char *get_blk_n_di(struct inode *myinode, int n);`** which returns the address of the n[th] double indirect (di) data block (if it exists); otherwise, it returns NULL.
For the first di data block (if exists), `n` is `0`. When n is 0, `return myinode->di[0][0];`
For the last di data block (if exists), `n` is `num_ptr*num_ptr-1`. In this case,
`return myinode->di[num_ptr-1][num_ptr-1];`
Note that direct or any indirect level might be partially filled with blocks! So if there is no more block or level the corresponding pointer will contain NULL.



dp

si

di

ti

```
char *get_blk_n_di(struct inode *myinode, int n)
   int i, j, k;
   int count=0;
   int num_ptr =   8 * 1024 / 4;
   if (n<0 || n > num_ptr*num_ptr-1) return NULL;

   if (myinode->di){
     for(i=0; i < num_blk; i++)
       if (myinode->di[i]) {
         for(j=0; j < num_blk; j++)
            if (myinode->di[i][j]) {
              if (n==0) return myinode->di[i][j];
              n--;
            } else return NULL;
         } else return NULL;
   }
   return NULL;
}                                  // A BETTER WAY
i = n / num_blk;
j = n % num_blk;
if (myinode->di && myinode->di[i] && myinode->di[i][j])
   return myinode->di[i][j];
else return NULL;
```

5. [20 points] IPC and pipes -fifo

Write a program (say prog.c) that forks and runs a sub-shell as a child process, and simply counts the number of output characters that the shell printed on the standard output. When the sub-shell terminates, the parent simply reports the number of output characters produced by the sub-shell.

For parent to do its job, it needs to get everything the child shell prints on the standard output. Hope you see how a pipe will be useful here:

child [1:STDOUT_FILENO] **--> pipe -->** [0: STDIN_FILENO] **parent.**

Since now the parent can get everything the child writes on the standard output, the parent can count the number of characters and write them into its own standard output.

You are asked to complete the following program so you can create the necessary pipe, child process and connect them as explained in the above scenario. You can ignore most of the error checking to make your solution clear, but you need to close all unnecessary file or pipe descriptors and check what read-write etc return. Also read/write one char at a time to make counting job easy!

```
/* your simple implementation of prog.c */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
int main (int argc, char *argv[]) {
   int mypipe[2];
   int childpid, numread, numwrite, count=0;
   char buf;

   /* 4pt - create pipe and child process */

        pipe(mypipe);
        childpid = fork();




   if(childpid == 0){ /* child */
      /* 4pt - child sets up the pipe */

      dup2(mypipe[1], STDOUT_FILENO);
      close(mypipe[0]);
      close(mypipe[1]);



     execl("/bin/bash","shell",NULL);
     perror("cannot start shell");
     return 1;
   }

   /* continue in the next page */
```

```
/* continue problem 5.b here */

        /* 4pt - parent sets up the pipe */

        dup2(mypipe[0], STDIN_FILENO);
        close(mypipe[0]);
        close(mypipe[1]);



        /* 8pt - parent reads from the pipe as long as there is data */
        /* counts the characters and puts them on standard output    */


        while(1){
           numread = read(STDIN_FILENO ,&buf, 1);
           if (numread <= 0) break;          // what if EINTER ?

           count++;

           numwrite = write(STDOUT_FILENO, &buf, 1);
           if (numwrite<=0) break;           // what if EINTER ?


         } // end of while


         // no need for  close in that case  ....
         fprintf(stderr,Shell printed %d characters \n", count);
         waitpid(NULL);
        }
```