The source code below is for a simple program named **`proc_run`** , which takes an integer argument and will run successfully on a Linux system when located in the working directory and started from a shell prompt as shown: (see the reference pages for system call details)

`-bash-3.00$  ./proc_run 5`

```
int main(int argc, char* argv[]){
        int arg;
        char nstr[10];

        printf("LEVEL: %s\n", argv[1]);
        arg = atoi(argv[1]);                 // convert arg string to int
        --arg;                               // decrement arg value
        sprintf(nstr, "%d", arg);            // convert int to arg string

        if(arg){
           switch(fork()){
              case -1:
                       exit(0);
              case 0:
                       execl("./proc_run", "proc_run", nstr, NULL);
            }
        }
        wait(NULL);
        printf("LEVEL: %s  PROC IS DONE\n", argv[1]);
}
```

A. **Write all the output** that will be generated when this program is run with the shell command shown above:

B. Although we expect the **`fork()`** calls made above to succeed, in general, what could lead to a **`fork()` call failing ?**

```
int main(int argc, char* argv[]){
        int arg;
        char nstr[10];

        printf("LEVEL: %s\n", argv[1]);
        arg = atoi(argv[1]);                  // convert arg string to int
        --arg;                                // decrement arg value
        sprintf(nstr, "%d", arg);             // convert int to arg string

        if(arg){
           switch(fork()){
              case -1:
                       exit(0);
              case 0:
                       execl("./proc_run", "proc_run", nstr, NULL);
            }
        }
        wait(NULL);
        printf("LEVEL: %s  PROC IS DONE\n", argv[1]);
}
```

A. **Write all the output** that will be generated when this program is run with the shell command shown above:

LEVEL 5
LEVEL 4
LEVEL 3
LEVEL 2
LEVEL 1
LEVEL 1 PROC IS DONE
LEVEL 2 PROC IS DONE
LEVEL 3 PROC IS DONE
LEVEL 4 PROC IS DONE
LEVEL 5 PROC IS DONE

B. Although we expect the `fork()` calls made above to succeed, in general, what could lead to a **`fork()` call failing ?**

fork() can fail when the system is out of resources (mem, swap) or a process limit is hit (too many user processes)

Exceptions are delivered to a processor under a variety of circumstances. In all cases, when the exception is delivered and the processor recognizes it, the thread that is currently running on that processor **is diverted from its code path into an exception code path** (typically changing address space from user mode into kernel mode). Exception code paths are generally activated via a **vectoring** mechanism, as we have discussed in class.

A. Exceptions are broadly **categorized** as either **synchronous or asynchronous**. **Explain the difference** between the two types of exceptions, and **provide an actual example of each type.**

B. An **running thread** tries to execute an instruction that **dereferences a NULL pointer**, which results in that thread running an **exception handler** in the kernel.

1) What **specific kind** of an exception is this event ?

2) When the **exception handler completes** and returns back to the code of the running thread**, which instruction** will the running thread **start to execute ?**

C. **Consider a** CPU **that is currently executing an** IDLE **thread in** user space **at a time when a local disk controller has just completed transferring disk blocks from a disk into memory, and has sent an** interrupt **to that CPU. Between the executions of each instruction of the IDLE thread, the CPU checks for interrupts, and when it finds this one it** forces the IDLE thread **into the kernel to run the exception handler for this event. Can the IDLE thread** lose the CPU now **while in the kernel (i.e. can a context switch happen here), or** must the IDLE thread return to user mode **after completing the exception code ?** Explain your answer**.**

A. Exceptions are broadly **categorized** as either **synchronous or asynchronous**. **Explain the difference** between the two types of exceptions, and **provide an actual example of each type.**

Sync – divide by zero – caused by instruction execution

Async – disk controller interrupt – external event

B. An **running thread** tries to execute an instruction that **dereferences a NULL pointer**, which results in that thread running an **exception handler** in the kernel.

  1) What **specific kind** of an exception is this event ?

  This is a FAULT

  2) When the **exception handler completes** and returns back to the code of the running thread**, which instruction** will the running thread **start to execute ?**

  Attempt to re-execute the offending instruction

C. Consider a **CPU** that is currently executing an **IDLE** thread in **user space** at a time when a local disk controller has just completed transferring disk blocks from a disk into memory, and has sent an **interrupt** to that CPU. Between the executions of each instruction of the IDLE thread, the CPU checks for interrupts, and when it finds this one it **forces the IDLE thread** into the kernel to run the exception handler for this event. Can the IDLE thread **lose the CPU now** while in the kernel (i.e. can a context switch happen here), or **must the IDLE thread return to user mode** after completing the exception code ? **Explain your answer**.

IDLE thread may be pre-empted, and not get back to user run state for a while, but cannot block

The following shows the original sources for a simple **masm** program to be built from **two separate source files**. It also shows the object files produced when each is assembled with **masm** using the **–o** flag.

A SIMPLE MAIN PROGRAM

```
bash-2.05$ cat main1.asm
main:   lodd arg1:
        push
        lodd arg2:
        push
        call myadd:
        stod rslt:
        halt
        .LOC 10
arg1:   25
arg2:   75
rslt:    0
```

A SIMPLE EXTERNAL FUNCTION

```
bash-2.05$ cat myadd.asm
myadd:  lodl 1
        addl 2
        addd bias:
        retn
bias:   100
```

ASSEMBLE WITH –o OPTION

```
bash-2.05$ ./masm_mrd -o < main1.asm > main1.obj
bash-2.05$ cat main1.obj
   0   U0000000000000000    arg1:
   1   1111010000000000
   2   U0000000000000000    arg2:
   3   1111010000000000
   4   U1110000000000000    myadd:
   5   U0001000000000000    rslt:
   6   1111111111000000
   10  0000000000011001
   11  0000000001001011
   12  0000000000000000
  4096 x
    rslt:                          12
    arg2:                          11
    arg1:                          10
    main:                           0
```

ALSO ASSEMBLED WITH –o OPTION

```
bash-2.05$ ./masm_mrd -o < myadd.asm > myadd.obj
bash-2.05$ cat myadd.obj
   0  1000000000000001
   1  1010000000000010
   2  U0010000000000000    bias:
   3  1111100000000000
   4  0000000001100100
  4096 x
    bias:                           4
    myadd:                          0
```

If you built a **linker program** (just as you did in assignment #6), and linked these two separate object files into an **executable binary output file** so the first executable instruction from the file **`main1.asm`** was placed **at location 0** in the executable, that output file would have **18 lines of 16 bit entries**. The **first 4** of these entries are provided below, **you must fill in the last 14**.

0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
1	1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0
2	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1
3	1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0
4	
5	
6	

Address	Executable Content		Corresponding code
0	0000000000001010	main:	lodd arg1:
1	1111010000000000		push
2	0000000000001011		lodd arg2:
3	1111010000000000		push
4	1110000000001101		call myadd:
5	0001000000001100		stod rslt:
6	1111111111000000		halt
7	1111111111111111		.LOC 10
8	1111111111111111		
9	1111111111111111		
10	0000000000011001	arg1:	25
11	0000000001001011	arg2:	75
12	0000000000000000	rslt:	0
13	1000000000000001	myadd:	lodl 1
14	1010000000000010		addl 2
15	0010000000010001		addd bias:
16	1111100000000000		retn
17	0000000001100100	bias:	100