1. [15pts] The following is a part of codes which handle signals sent to the shell. The code does **not** execute normally and you must fix the bug.

```
  struct job_t {            /* The job struct
*/
       pid_t pid;            /* job PID
    */
       int jid;             /* job ID [1, 2, ...]
    */
       int state;           /* UNDEF, BG, FG, or ST
    */
       char cmdline[MAXLINE];  /* command line
    */
};
struct job_t jobs[MAXJOBS]; /* The job list */

/* Helper functions  */
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t
pid);
pid_t fgpid(struct job_t
*jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t
pid);
struct job_t *getjobjid(struct job_t *jobs, int
jid);
...

void sigchld_handler(int sig)
{
  pid_t child_pid;
  int child_jid;
  int status;

  /* '-1' option means to detect any terminated or stopped jobs*/
  while ((child_pid = waitpid(-1, &status, WNOHANG)) > 0) {

    if (WIFSTOPPED(status)) {
      /* This code block will be executed, when a child process receives SIGTSTP signal. */
       struct job_t *j = (a)_______________________________ /*get job using process id */
       (b)_______________________________   /*change state  */
       printf("sigchld_handler: Job stopped by signal\n");
    }
    else if (WIFSIGNALED(status)) {
       /*delete job process*/
```

```
        ...
        printf(“sigchld_handler: Job terminated by signal\n”);
     }
     else if (WIFEXITED(status)) {
         /*delete job process*/
        ...

        printf(“sigchld_handler: Job deleted\n”);
     }
  }
  return;
}
void sigint_handler(int sig)
{
  ...
  pid_t pid=fgpid(jobs);  /*find foreground job */
  (c)________________________________   /*send signal */
  printf(“sigint_handler: Job killed\n”);
  return;
}

void sigtstp_handler(int sig)
{
  /* This handler will be called when SIGTSTP is sent to the shell.*/
  pid_t pid=fgpid(jobs);  /*find foreground job */
  (d)________________________________   /*send signal */
  printf(“sigtstp_handler: Job stopped\n”);
  return;
}
```

Assume all the codes not shown are properly implemented.

Q1-A [5pts]. **Fill in the codes for blank line (a) ~(d).** (For (a), you can use one of these helper functions.)

Q1-B [5pts]. Give a combination of shell commands which does not work for this shell and explain the case. One simple case is enough to receive a full credit. Also explain how to fix the code to handle this case properly.

The shell will be receiving any combination of the following commands
1) Execute a foreground job
2) Execute a background job
3) Receive and handle signals that are sent from the kernel

4) Receive and handle signals that are signaled from child process

5) Built-in Commands : quit, jobs

2. **[10pts]** The following code is an incomplete code fragment for a tiny shell.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];         /* argv for execve() */
    int bg;                      /* should the job run in bg or fg? */
    pid_t pid;                   /* process id */
    sigset_t mask;        /* signal mask */

    bg = parseline(cmdline, argv);
    ...

    if (!builtin_cmd(argv)) {

        /* mask initialization for signal blocking(SIGINT,SIGCHLD,SIGTSTP)
         ...
       sigprocmask(SIG_BLOCK,&mask,NULL);
        pid = fork();

        if (pid == 0) {
           sigprocmask(SIG_UNBLOCK,&mask,NULL);

            /* Now load and run the program in the new job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }
        addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
        sigprocmask(SIG_UNBLOCK,&mask,NULL);

        if (!bg) waitfg(pid);
       else printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    return;
}
```

Q2-A [5pts] Above function does **not execute correctly.** Explain why this code does not work.

Q2-B [5pts] Which function(s) should be added to the code above? (Specify the position of function)

3. **[20pts]** MallocLab. *Explicit free list* is one of free block management mechanisms that can be used when implementing a memory allocator. Answer the following questions to implement explicit free lists.

Q3-A [5pts] Design a free block to be managed by your memory allocator. Your free block design must track the size of the block, must contain pointers to be inserted into a doubly linked list, and must be capable of coalescing with adjacent free blocks.

Provide an illustration of your free block including sizes of each element and where which part of the block the pointers (`ptr`: payload pointer, `bp`: block pointer) would point to. Write the code for a C `struct` with functions or series of macro functions (`#define`) to access elements of the block.

Your definitions will need to provide the following functionality in the form of functions or macro functions:

Get and Set the following elements:

- Size of the block (If you use more than one 'size' element, provide as many as you use)
- The free/allocated bit of the block
- Pointer to the next block
- Pointer to the previous block

Any other functions you think necessary.

Your function arguments must distinguish the payload pointer as `ptr` and block pointer as `bp`. And if necessary include functions to convert the `ptr<->bp`. **Hint**: Look at Q.B and Q.C to define any functions that may be required in those questions.

Q3-B [5pts]. Implement a next-fit free block search function using the functions you defined in Q.A. You may add global pointer variable(s) for your next-fit implementation. **You must explain your code with comments for points.**

```
void * firstFreeBP; //Pointer that keeps the first free block of the list




/**
 * Return the address of the next fit free block bp satisfying requestSize,
 * It should return NULL if there is no free block.
 * You need to also design how you represent first/last blocks of the
 * explicit free list. (Add as comment, and as condition in your code)
 */

void *findNextFitFreeBlock(size_t requestSize){














}
```

4. [10pts] Suppose virtual address bits = 48, physical address bits = 40, and page size = 8KB

Q5-A [3pts] What information should be in the page table entry? (how many bits would be necessary for each field?)

Q5-B [3pts] How large is a single-level page table, assuming each table entry size is 8B.

Q5-C [4pts] Suppose the value of the page table pointer is 0x1000, and the virtual address for a mov instruction is “0x10002000”. What is the address of the corresponding page table entry for the virtual address?

5. [10pts] Virtual memory, interrupt, signal

Q7-A [5pts] Fork() creates a new process with the same memory content as the parent process. Explain how virtual memory support with page tables, can reduce the overhead of creating a new process. Also, discuss what happens if the child process updates part of the memory, if the overhead reduction technique is used.

Q7-B [5pts] A certain page fault causes a segmentation fault, while another one may not incur any problem with the program execution. Explain why page faults can result in such completely different outcomes. In addition, describe how interrupts and signals are used to kill the process for a segmentation fault.

6. 5pts] For the following 4 cases, explain possible errors or risks. Both p1 and p2 have references to variable x (and y if y is defined).

```
int x;        
p1() {}       p1() {}
```

```
int x;        int x;
p1() {}       p2() {}
```

```
int x;        double x;
int y;        p2() {}
p1() {}
```

```
int x=7;      double x;
int y=5;      p2() {}
p1() {}
```

7. [5pts] The following assembly code is generated from the C file, and it is not yet linked to the final program.

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```
0000000000000000 <main>:
  0:  48 83 ec 08           sub    $0x8,%rsp
  4:  be 02 00 00 00         mov    $0x2,%esi
  9:  bf 00 00 00 00         mov    $0x0,%edi      # %edi = &array
                   a: R_X86_64_32 array        # Relocation entry

  e:  e8 00 00 00 00         callq  13 <main+0x13> # sum()
                   f: R_X86_64_PC32 sum-0x4      # Relocation entry
 13:  48 83 c4 08           add    $0x8,%rsp
 17:  c3                   retq
```

- [2pts] For the first relocation entry (bf 00 00 00 00), what information is necessary during the linking process?

- [3pts] Suppose the starting address of main is 0x4003b0 and the starting address of sum is 0x4003e8 in the final relocated binary. How should the second relocation entry be completed? (e8 00 00 00 00)

1. [10pts] The following two C code fragments show two different ways of supporting 2D data structures.

   A. [5pts] Complete the following assembly codes for the two access functions. Discuss the pros and cons of the two approaches.

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};

int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3

int *univ[UCOUNT] = {mit, cmu, ucb};

int get_univ_digit (size_t index,
size_t digit){

  return univ[index][digit];

}
```

```
1)________________________
addl  %rax, %rsi
movl   2)__________, %eax
```

```
salq    $2, %rsi
addq    3)__________, %rsi
4)__________________________
```

   B. [5pts] Complete the following assembly codes

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
.L11:                          # loop:
  movslq  16(%rdi), %rax      #   i = M[r+16]
    1) ____________________     #
    2) ____________________     #
   testq   %rdi, %rdi          #   Test r
   jne     .L11                #   if !=0 goto loop
```