1.a What happens if the return value of execlp() is not checked and you give an invalid command, say, lssss, as input to the shell?

Ans. If the return value of execlp() is not checked and we give an invalid command, the child process is not terminated and the next process(command we enter) gets spawned with its parent set as the non-terminated child process.

Example:

Upon entering 'ls', a valid command (*17467* is the parent process_id) *[17467]$ ls*

The child process gets terminated as can be seen below (there is no other process running other than the parent process)

sslab03 32 $ ps -ef | grep 17467

sdorbala 17467 16865 0 14:41 pts/0   00:00:00 ./a.out

sdorbala 17470 17450 0 14:42 pts/1   00:00:00 grep 17467

Whereas, if we enter 'lsss', an invalid command (*17467* is the parent process_id) *[17467]$ lss*

The child process (*17471*) is not terminated and now there are two processes running (child and parent)

sslab03 34 $ ps -ef | grep 17467

sdorbala 17467 16865 0 14:41 pts/0   00:00:00 ./a.out

sdorbala 17471 17467 0 14:42 pts/0   00:00:00 ./a.out

sdorbala 17478 17450 0 14:42 pts/1   00:00:00 grep 17467

1.b What happens if the parent process does not perform waitpid() and immediately returns to the beginning of the while-loop?

Ans. If the parent process doesn't perform waitpid(), then the child that got terminated turns into a zombie implying that the resources allotted to the child processes are not released and are stored in the kernel with a minimal set of information.

These zombies are stored as slots in kernel process table and once the slots are all occupied, no further processes could be created. Also, the next processes are all created with the child process set as the new parent process.

Example:

sdorbala 19463 19299 0 19:14 pts/2   00:00:00 ./a.out

sdorbala 19466 19463 0 19:14 pts/2   00:00:00 [ls] <defunct>

sdorbala 19470 19463 0 19:15 pts/2   00:00:00 [ls] <defunct>

<*defunct*> implies a zombie process.

19463 - parent process; *19466* - child process that acts as a parent for the next operation;

1.c If the concurrent server were a file server that receives client requests from processes on the same host/OS (or over a network), why is performing waitpid() as a blocking call from within the parent process not a valid approach? Describe an asynchronous method for performing waitpid() so child processes are prevented from becoming zombies and their exit status can be checked.

Ans. Performing *waitpid()* is not a valid approach as it blocks the parent process from moving forward until the child process gets terminated. To prevent this blocking call, asynchronous method using *SIGCHLD* handler (calls *waitpid()* on *pid*: -1) can be used.

For executing *SIGCHLD*, we need to register a *handler()* that when called executes *waitpid()* with *WNOHANG* option parameter. This optional parameter is enabled to prevent the parent process from blocking while getting the status of a child. And, as there could be more than one child processes that might have got terminated, the *while* loop ensures that all such terminated processes are reaped one after the other.

```
void handler(int signal)
{
        While(waitpid(pid_t(-1),0,WNOHANG)>0){}
}
```

Also, there could be other child processes that are not yet terminated during that particular execution of *SIGCHLD*, but such processes leave behind a pending *SIGCHLD* that will result in the handler being called again.

POSIX recommended way of registering this handler is through the *sigaction* function. The flags *SA_RESTART* is enabled to prevent unnecessary operation system interrupts and *SA_NOCLDSTOP* is enabled to prevent *SIGCHLD* from being raised when the child process gets stopped or continued as opposed to terminating.

```
struct sigaction sa;
sa.sa_handler = &handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
if (sigaction(SIGCHLD, &sa, 0) == -1) {
  perror(0);
  exit(1);
}
```

1.d Ignoring the functional simplifications of the shell code, point out at least two programming bugs that should be fixed to yield more reliable server code.

Ans. **A) Missing fork() error check:** There is no error check when the **fork()** call fails. Fork() function return *0* for a child process, *child_pid* for the parent process and *-1* for errors.

```
k=fork();
if(k==0 {  // child process code }
else if(k==-1 { // fork fails, exit(-1) }
else{     // parent process code }
```

B) Missing waitpid() error check: *waitpid()* returns the *processID* of the child when the status is available, else it returns -1 (with error set to one of the values *ECHILD, EINT, EVAL*, etc). This could help in debugging process when something gets wrong at *waitpid().*