# Homework 6

1. *FORK AND EXECVE.* Read the C program and answer the questions below. NOTE: `/bin/echo` is an executable file that will print its arguments on the screen.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

char *ch;

int main()
{
        ch = malloc(1);
        *ch = 'A';

        if (fork() == 0) {
                *ch = 'B';
                printf("%c\n", *ch);

                if (fork() == 0) {
                        printf("C\n");
                }
                else {
                        exit(0);
                }
        }
        else {
                while (waitpid(-1, NULL, WUNTRACED) > 0);
                char *my_argv[] = {"/bin/echo", ch, 0};
                execve(my_argv[0], my_argv, 0);
        }

        free(ch);
        return 0;
}
```

   (a) What is the possible output of this program? Is the output deterministic? Please explain why.

   The output of this program is not deterministic. It can be `"B\nC\nA\n"` or `"B\nA\nC\n"`.

   (b) Is there any memory leakage or double free issue for the variable `ch` in each process? Please explain why.

   There is no double free issue since the memory spaces of these three processes are isolated.

   The grandchild process frees the allocated memory before exits while the child process does not. The parent process calls `execve` and afterwards its memory mappings are not preserved (Modern implementations of `malloc()` use anonymous memory mappings, see the `man` page of `mmap(2)` and `execve(2)`) so no memory leakage for him.

2. *BLOCKING AND UNBLOCKING SIGNALS.* Linux provides an explicit mechanism for blocking signals. Read the C program and answer the questions below.

```c
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>

void sig_han(int sig)
{
     printf("signal␣handled\n");
}

int main()
{
     sigset_t set;
     int i;
15
16      signal(SIGKILL, sig_han);
17      signal(SIGINT, sig_han);
18      sigemptyset(&set);
19      sigaddset(&set, SIGINT);
20      sigprocmask(SIG_BLOCK, &set, NULL);
21
22      for (i = 0; i < 3; i++) {
23              printf("send␣signal\n");
24              kill(getpid(), SIGINT);
25      }
26
27      sigprocmask(SIG_UNBLOCK, &set, NULL);
28      return 0;
29 }
```

(a) When run, the call at line 16 fails. Why? And what is the return value of this call and what value would `errno` be set to? (You can run the program or refer to the `man` page `signal(2)`).

"SIGKILL can be neither caught nor ignored." (Section 8.5, Figure 8.26). `signal(SIGKILL, sig_han)` returns `SIG_ERR` and `errno` is set to 22 (`EINVAL`).

(b) What is the output of this program? Please explain your answer.

The output is `"send signal\nsend signal\nsend signal\nsignal handled\n"`.

The pending signal is not delivered to the process when the signal is blocked. So the control flow of the program will not change to the signal handler until the signal is unblocked. The signal will not be delivered multiple times because pending signals are not queued. "the existence of a pending signal merely indicates that at least one signal has arrived" (Section 8.5.5, Correct Signal Handling).

3. (Optional) *SIGNAL HANDLER INSIDE*. For a user-defined signal handler (suppose a x86-64 machine runnnig Linux),

(a) does it run in user mode or kernel mode (that is, in non-privileged mode or privileged mode)?

User mode (non-privileged) mode.

(b) what stack does it use, does it share the same stack with your normal functions? Use `GDB` to stop inside a signal handler and check the stack address.

Signal handlers use different stacks. Also, handlers for different signals run at different stacks.

User can assign an alternative signal handling stack by `sigaltstack` (See the `man` page `sigaltstack(2)` for details).

(c) where does the signal handler function return to, what is the next instruction after the signal handler function `retq`? Use `GDB` to stop inside a signal handler and step instruction-by-instruction to check where is the handler function returns to (You may find `GDB`'s `si`, `ni`, `disassemble` commands useful).

The signal handler returns to a piece of code named `__restore_rt` which do a system call (`rt_sigreturn`).