作业5

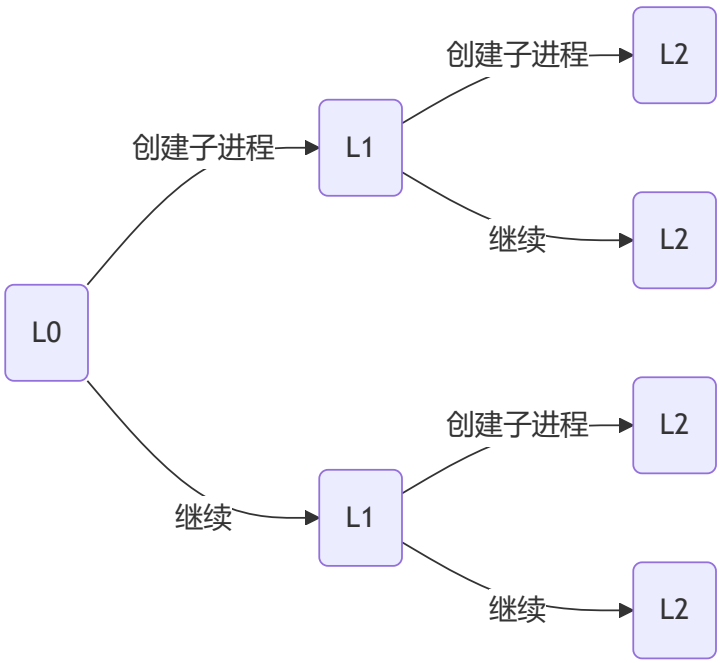
250109 Update：更新作业5习题解答

1. fork()

阅读代码，回答以下问题

```
#include <cstdlib>
#include <unistd.h>
#include <cstdio>
int counter = 0;
int main(){
    for (int i = 0;i < 2; i++) {
        fork();
        counter++;
        printf("counter = %d\n", counter);
    }
    // 注意：这里没有 counter++;
    printf("counter = %d\n", counter);
    return 0;
}
```

画出父子进程关系：



在 `fork()` 后会输出一次 `counter` 的值，退出for循环后再输出一次。首先创建L1进程时共输出2次，创建L2进程时共输出4次，L2进程们退出for循环在输出，共4次，因此总共10行。

因为先有L0创建L1进程后，L1进程再创建L2进程，所以第一行必定是1，而由于L2进程都会退出循环后再输出，最后一行必定是2。

根据上述分析，由于OS对进程的调度，在创建L1和L2时顺序可能会被打乱，故所有的可能如下：

```
112222 2222
121222 2222
122122 2222
112212 2222
112221 2222
```

1. 10行
2. 1, 2
3. 5种可能

2. 文件读写

假设文件 `file1.txt` 中有一个字符串 `aabbccdd` 。

下列C文件分别被编译成 `./program1` 和 `./program2` ：

```c
/* Program 1 */
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char const *argv[]) {
    int pid, fd_x, fd_y, fd_z;
    char buf[8];

    fd_x = open("file1.txt", O_RDWR);
    fd_y = open("file1.txt", O_RDWR);
    fd_z = open("file1.txt", O_RDWR);

    read(fd_x, buf, 2);
    read(fd_y, buf + 2, 4);

// ----------------

    if ((pid = fork()) == 0) {
        dup2(fd_x, STDOUT_FILENO);
        dup2(fd_y, STDIN_FILENO);
        execl("program2", "program2", NULL);
    }

    wait(NULL);

// ----------------

    read(fd_y, buf + 6, 2);
    write(fd_z, buf + 6, 2);
    write(fd_x, buf + 4, 2);
    write(fd_x, buf + 2, 2);

    close(fd_x);
    close(fd_y);
    close(fd_z);

    return 0;
}


/* Program 2 */
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char const *argv[]) {
    char buf2[2];
    read(STDIN_FILENO, buf2, 2);
    write(STDOUT_FILENO, buf2, 2);
}
```

对所调用的一些方法的解释

- `open(filename, O_RDWR)`：打开一个已有文件 `filename`，对其进行读写操作，起始位置为 `0`；
- `execl(exename,...)`：执行 `exename` 程序，与我们讲的 `execve()` 功能类似。

当执行 `./program1` 后，文件 `file1.txt` 的内容是什么？（2分）

从以下几点解释一下为什么是这个结果（8分，一点2分，言之有理即可，**建议不超过256个字**）：

- `./program2` 做了什么?
- `./program1` 在 `fork()` 之前做了什么，相应的 `buf` 的内容如何变化?
- `./program1` 调用 `fork()` 了以后，子进程在干什么?
- 子进程返回后，`./program1` 又做了什么，相应的 `buf` 的内容如何变化?

> 参考课件《10-IO处理》P.26~32，同时建议完成 P.33~35 的练习题。

对于同一文件描述符 `fd` 来说，每次成功的 `read()` 和 `write()`，都会更新当前文件的访问位置（见课件《10-IO处理》P.8,9 的提示），**此次的偏移量即当前成功读写的字节数**：

`read()`，`write()` 更新文件访问位置

这里节选了关于文件访问位置更新的部分内容，详细可通过 Linux manual pages 查阅：

- read(2) — Linux manual page：
  On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and `read()` returns zero.
- write(2) — Linux manual page：
  For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was open(2)ed with `O_APPEND`, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

以 ^ 表示当前文件的访问位置，举例来说：

```
int fd_x = open("file1.txt", O_RDWR);
/* fd_x: aabbccdd
         ^
 */

read(fd_x, buf, 2);
/* fd_x: aabbccdd
           ^
 */

write(fd_x, "gg", 2);
/* fd_x: aaggccdd
             ^
 */
```

阅读源代码，可知 `./program2` 从自己的标准输入流stdin中读入2个字节保存到 `buf[]` 后，再将 `buf[]` 写入自己的标准输出流stdout中。

按照第二小题的提示，我们可以分为三个部分来分析：

不妨假定初始时 `buf` 为"空"（每一个元素为 `'\0'` ），即 `buf = ""` 。

在 `fork()` 之前：

1. 从 `fd_x` 中读入2个字节，即 `"aa"` ，保存到 `buf + 0` 上： `buf = "aa"` ；
2. 从 `fd_y` 中读入4个字节，即 `"aabb"` ，保存到 `buf + 2` 上： `buf = "aaaabb"` 。

此时各文件描述符的访问位置如下：

```
/* fd_x: aabbccdd
           ^
 * fd_y: aabbccdd
             ^
 * fd_z: aabbccdd
         ^
 */
```

调用 `fork()` 了以后，子进程通过 `dup2()` 将它的stdin重定向至 `fd_y` 、stdout重定向至 `fd_x` ，然后执行 `./program2` （为了方便区分，下面以 `buf2` 表示 `program2.c` 中的 `buf[]` ）：

1. 先从 `fd_y` 中读入两个字节 `"cc"` ，保存到 `buf2 + 0` 上： `buf2 = "cc"` ；
2. 将 `buf2` 保存的两个字节 `"cc"` 写入 `fd_x` 中： `file1.txt` 的内容为 `aaccccdd`

父进程等待子进程结束后再继续，继续前各文件描述符的访问位置与内容如下：

```
/* fd_x: aaccccdd
            ^
 * fd_y: aaccccdd
              ^
 * fd_z: aaccccdd
        ^
 */
```

子进程返回后：

1. 从 `fd_y` 中读入两个字节 `"dd"` ，保存到 `buf + 6` 上： `buf = "aaaabbdd"`
2. 将 `buf + 6` 起始的两个字节 `"dd"` 写入 `fd_z` 中： `file1.txt` 的内容为 `ddccccdd`
3. 将 `buf + 4` 起始的两个字节 `"bb"` 写入 `fd_x` 中： `file1.txt` 的内容为 `ddccbbdd`
4. 将 `buf + 2` 起始的两个字节 `"aa"` 写入 `fd_x` 中： `file1.txt` 的内容为 `ddccbbaa`

在 `close()` 之前，不妨看一下各文件描述符的访问位置：

```
/* fd_x: ddccbbaa
               ^
 * fd_y: ddccbbaa
               ^
 * fd_z: ddccbbaa
          ^
 */
```

最终 `file1.txt` 的内容为 `ddccbbaa` 。

3. 信号量

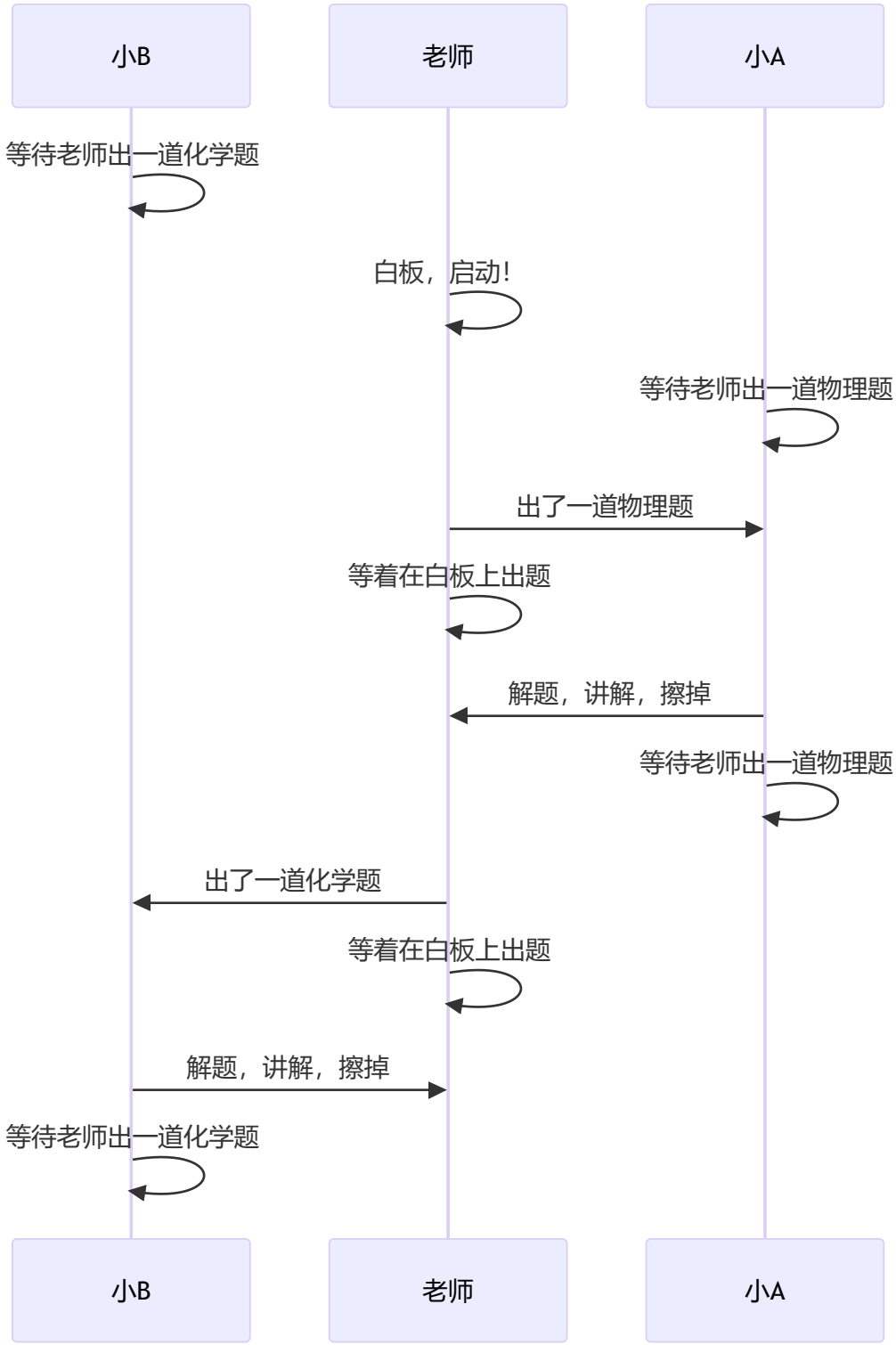
老师的办公室有一个空白板。老师会在白板为空时往白板上写一道物理题或一道化学题。如果是一道物理题，喜爱物理的小A会解答出这道题并把题目擦掉，如果是一道化学题，喜爱化学的小B会解答出这道题并把题目擦掉，请使用信号量和 `P` 、 `V` 原语实现老师、小A、小B三者的同步。

```
sem_t board;      // 白板是否为空
sem_t physics;    // 白板上是否为物理题
sem_t chemistry;  // 白板上是否为化学题

void init() {
    Sem_init(&board, 0, ___(A) ___);
    Sem_init(&physics, 0, ___(B) ___);
    Sem_init(&chemistry, 0, ___(C) ___);
}
void teacher() {
    while (1) {
        Course c = (rand() & 1) ? PHYSICS : CHEMISTRY;
        ____(D) ____;
        在白板上写题目;
        if (c == PHYSICS) {
            ____(E) ____;
        } else {  // c == CHEMISTRY
            ____(F) ____;
        }
    }
}
void studentA() {
    while (1) {
        P(___(G) ___);
        解答物理题，将其擦掉;
        V(___(H) ___);
    }
}
void studentB() {
    while (1) {
        P(___(I) ___);
        解答化学题，将其擦掉;
        V(___(J) ___);
    }
}
```

P，V 原语的定义见课件《11-线程与线程同步基础》P.31，这里不加赘述。

不妨分析刚开始时白板为空，老师先出一题物理题让小A作答，再出一题化学题让小B作答的过程：



获得这样的序列图后，结合 `P`，`V` 原语可知：

- "自环"表示当前线程正在监听某个信号量直到其大于 `0`，对应于 `P()`；
- "单向传递"表示线程设置某个信号量以通知其他线程工作，对应于 `V()`。

同样也能明确信号量的设置：

- 老师监听 `board`，设置 `physics` 和 `chemistry`；小A监听 `physics`，小B监听 `chemistry`，他们作答完毕后设置 `board`；
- 初始时小A和小B都在监听，直到老师出题，因此 `physics` 和 `chemistry` 初值为 `0`；
- 白板一开始就是空的，这样老师就能开始出题；同时为了保证老师能顺利等待 `board` 被释放，`P(&board)` 结束后应进入等待（即 `board` 的值为 `0`），所以 `board` 初值只能为 `1`。

综上，答案为：

空格	值
A	1
B	0
C	0
D	P(&board)
E	V(&physics)
F	V(&chemistry)
G	&physics
H	&board
I	&chemistry
J	&board

> 注： __空格(G)__ 至 __空格(J)__ 中漏写 & 、信号量漏写字母 不扣分。

4. 线程与子进程（7分，全填对满分）

阅读程序写结果，**要求列出所有可能输出**，不考虑进程/线程创建失败的情况，并假设 printf 的输出不会被其他 printf 打断。换行请用 \n 表示。

> 为了方便大家看得清楚，参考解答中"人为地"在 \n 前后加入空格，实际输出是不包含空格的；同时，漏写末尾的 \n 不扣分。

1. a=1 \n

```c
// (1)
#include <pthread.h>
#include <stdio.h>
int a = 0;
void* test(void* ptr) { a++; return NULL; }
int main() {
    pthread_t pid;
    pthread_create(&pid, NULL, test, NULL);
    pthread_join(pid, NULL);
    printf("a=%d\n", a);
    return 0;
}
```

   只有一个线程使用 a ，不会发生竞争。

2. a=1 \n a=1 \n

```c
// (2)
#include <unistd.h>
#include <stdio.h>
int a = 0;
void test() { a++; }
int main() {
    int pid = fork();
    test();
    printf("a=%d\n", a);
    return 0;
}
```

父进程与子进程各有一个变量 a 的拷贝，且都执行了 a++ ，都会输出 a=1 。

3. a=1 \n 或 a=2 \n

```c
// (3)
#include <pthread.h>
#include <stdio.h>
int a = 0;
void* test(void* ptr) { a++; return NULL; }
int main() {
    pthread_t pid1, pid2;
    pthread_create(&pid1,NULL,test, NULL);
    pthread_create(&pid2,NULL,test, NULL);
    pthread_join(pid1, NULL);
    pthread_join(pid2, NULL);
    printf("a=%d\n", a);
    return 0;
}
```

具体结果取决于 pid1 和 pid2 之间对于共享变量 a 的竞争 (races) 是否发生：

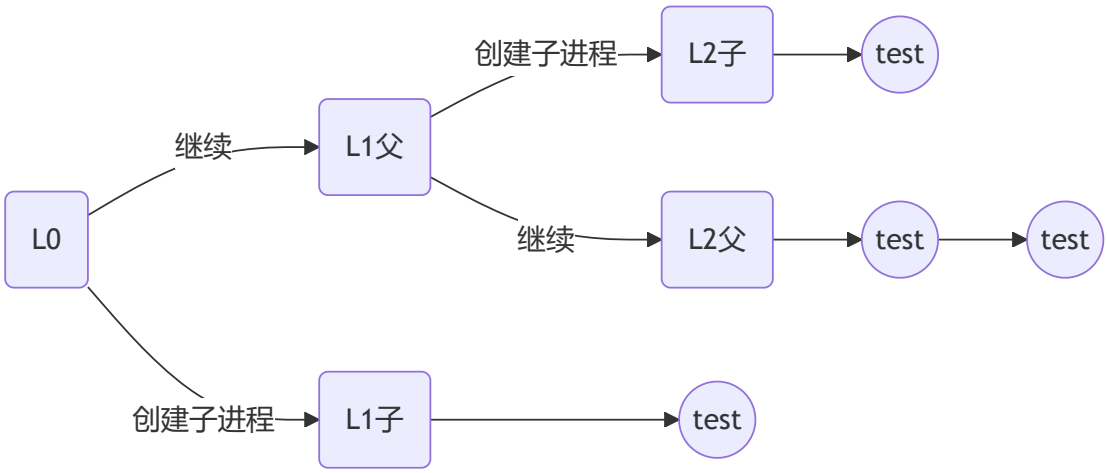
- 如果发生，只有一个线程抢占到 a ， a=1 ；
- 如果没发生，两个线程先后使用 a ， a=2 。

4. a=1 \n a=1 \n a=2 \n 或 a=1 \n a=2 \n a=1 \n 或 a=2 \n a=1 \n a=1 \n

```c
// (4)
#include <unistd.h>
#include <stdio.h>
int a = 0;
void test() { a++; }
int main() {
    int pid = fork();
    if (pid != 0) pid = fork();
    test();
    if (pid != 0) test();
    printf("a=%d\n", a);
    return 0;
}
```

记住：子进程 pid == 0 ，父进程的 pid 为子进程的进程号

画出父子进程关系：



按照三个进程可能的先后顺序写出输出即可。

5. 信号处理

考虑如下程序：

```
void handler (int sig) {
    printf("D");
    exit(4);
}
int main() {
    int pid, status;
    signal(SIGINT, handler);
    printf("A");
// ----------------
    pid = fork();
    printf("B");
// ----------------
    if (pid == 0) {
        printf("C");
    } else {
        kill(pid, SIGINT);
        waitpid(pid, &status, 0);
        printf("%d", WEXITSTATUS(status));
    }
    printf("E");
    exit(7);
}
```

以下哪些是可能的输出结果（多选，根据标答完全匹配给分）：_____

A. `ABCBE7E` B. `ABD7E` C. `ABBCE4E` D. `ABCDB4E` E. `ABBD4E`

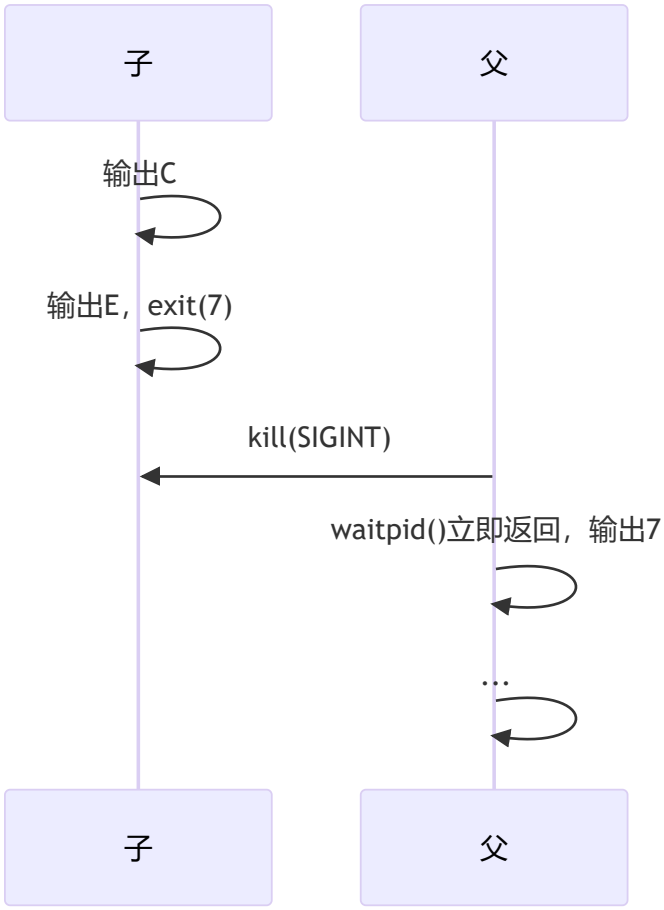
面对多选题，不妨采取排除法；与第一题类似，我们分成几个阶段来分析：

在 `fork()` 之前：

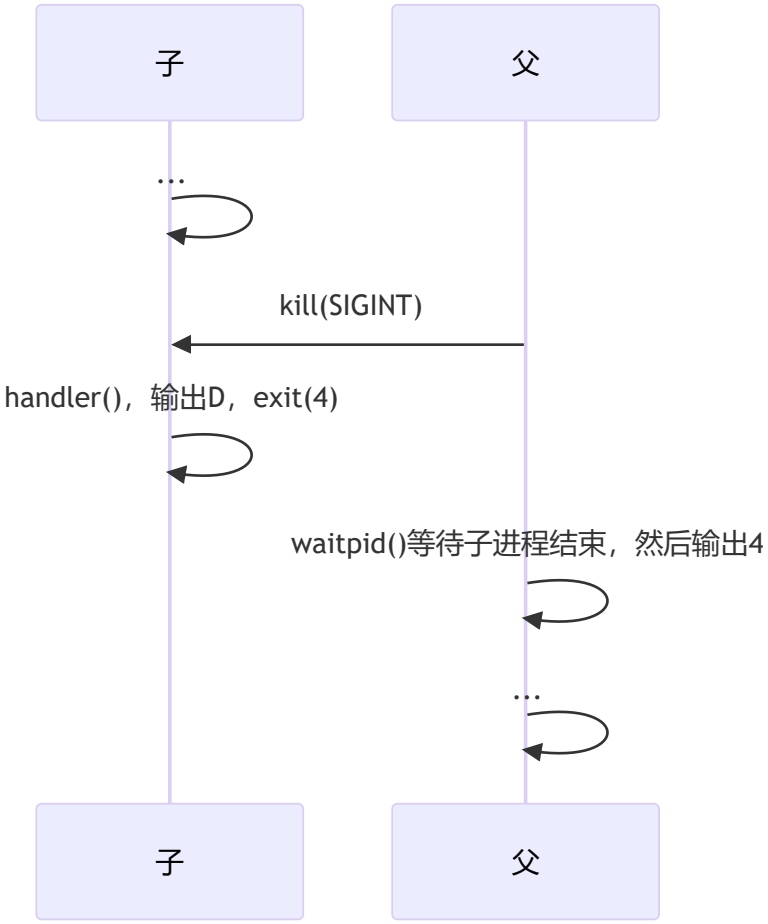
1. 通过 `signal()` 修改接收到信号 `SIGINT` 后，触发 `handler`
2. 在 `fork()` 之前只有一个进程，因此首先必定会输出一个 `A` ： 选项A,B,C,D,E符合

考虑到父进程与子进程的调度可能存在差异，分两种情况讨论：

- 子进程在 `exit(7)` 后接收到父进程的 `SIGINT`



- 子进程在 `exit(7)` 之前接收到父进程的 `SIGINT`



- 由上述分析可知，输出结果中 `4` 和 `D` 要么同时存在要么同时不存在，不可能只出现其中之一，选项C是不合理的；
- 父进程与子进程都会输出一个 `B` ，除非子进程被提前kill了（同上， `4` 和 `D` 同时存在），而选项B的输出只有一个 `B` ，有 `D` 却没有 `4` ，不合理；
- 如果子进程是在输出 `B` 后才被kill，对应的父进程也应该执行完 `printf("B");` 操作才会执行 `kill()` ，因此2个 `B` 必须出现在 `4` 和 `D` 之前，在操作系统顺序响应中断请求的情况下，选项D是不合理。

简单描述A,E对应的情况（可以画出序列图来辅助思考）来验证是否合理：

- 选项A：子进程在结束后， `SIGINT` 才发出去
- 选项E：子进程在输出 `C` 之前就接收到 `SIGINT`

选项 A,E 对应的情况是合理的。