

ICS 第0xa次小班课

2021-11-30

Virtualization of CPU #1/5

- Only a handful of CPUs, but many tens of processes. How to share the physical CPU among many jobs running seemingly at the same time?
 - **Time-sharing** to rescue.
 - **Performance:** a light overhead is desired
 - **Control:** OS should be in charge of the system, to protect the use process from doing any harm and to allocate the limited resource (e.g. memory, time, computation, etc.) “fairly” among processes.
 - **Virtual memory** to feign private address space.
- Basic idea: **Limited Direct Execution**
 - **User mode vs. kernel mode**
 - Code that runs in user mode is limited in what it can do. In contrast, in kernel mode, code can do whatever it likes. Processes must solicit help from the kernel via system calls.
 - The constrained instruction set in user mode must be supported by the hardware. Software designer (typically OS designer) implements the system call interface.
 - **Cooperative & non-cooperative scheduling (Cont.)**

Virtualization of CPU #2/5

■ Basic idea: **Limited Direct Execution**

■ **User mode vs. kernel mode**

■ **Cooperative & non-cooperative scheduling**

- In cooperative scheduling, kernel trusts the process to behave reasonably by yielding itself from time to time, typically via invoking system calls and via errors.
- In non-cooperative scheduling, kernel interrupts the execution of a process forcefully when the process runs for too long a time. This is implemented by a timer interrupt.

■ **Implementation issues:**

■ **System calls:**

■ **Save & restore context:**

- should memory and other program-invisible states be saved?
- Should registers, condition code and PC be saved?
- Where should it be saved? Say, in the user stack?

■ **Calling conventions:**

- Which system call to invoke? Where are its arguments?

Virtualization of CPU #3/5

- Implementation issues:

- Scheduling:

- **Context switch:**

- should memory and other program-invisible states be saved?
 - Should registers, condition code and PC be saved?
 - Where should it be saved? Say, in the user stack?
 - How to restore the context of the new process?

- **Which process to switch to?**

- Not covered in this course. Generally known as the scheduling policy or the scheduling algorithm.

Virtualization of CPU #4/5

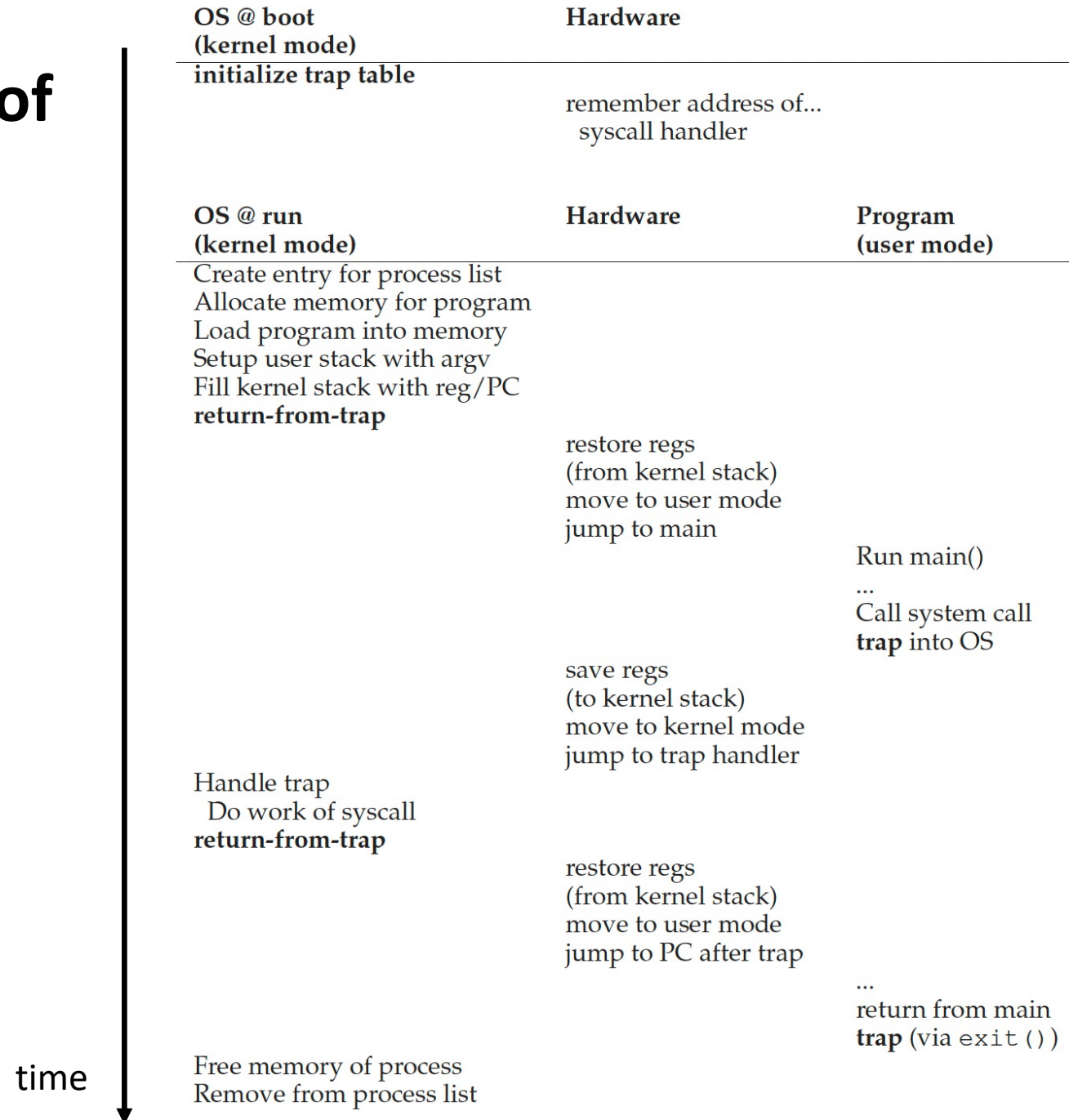


Figure 6.2: Limited Direct Execution Protocol

Virtualization of CPU #5/5

time

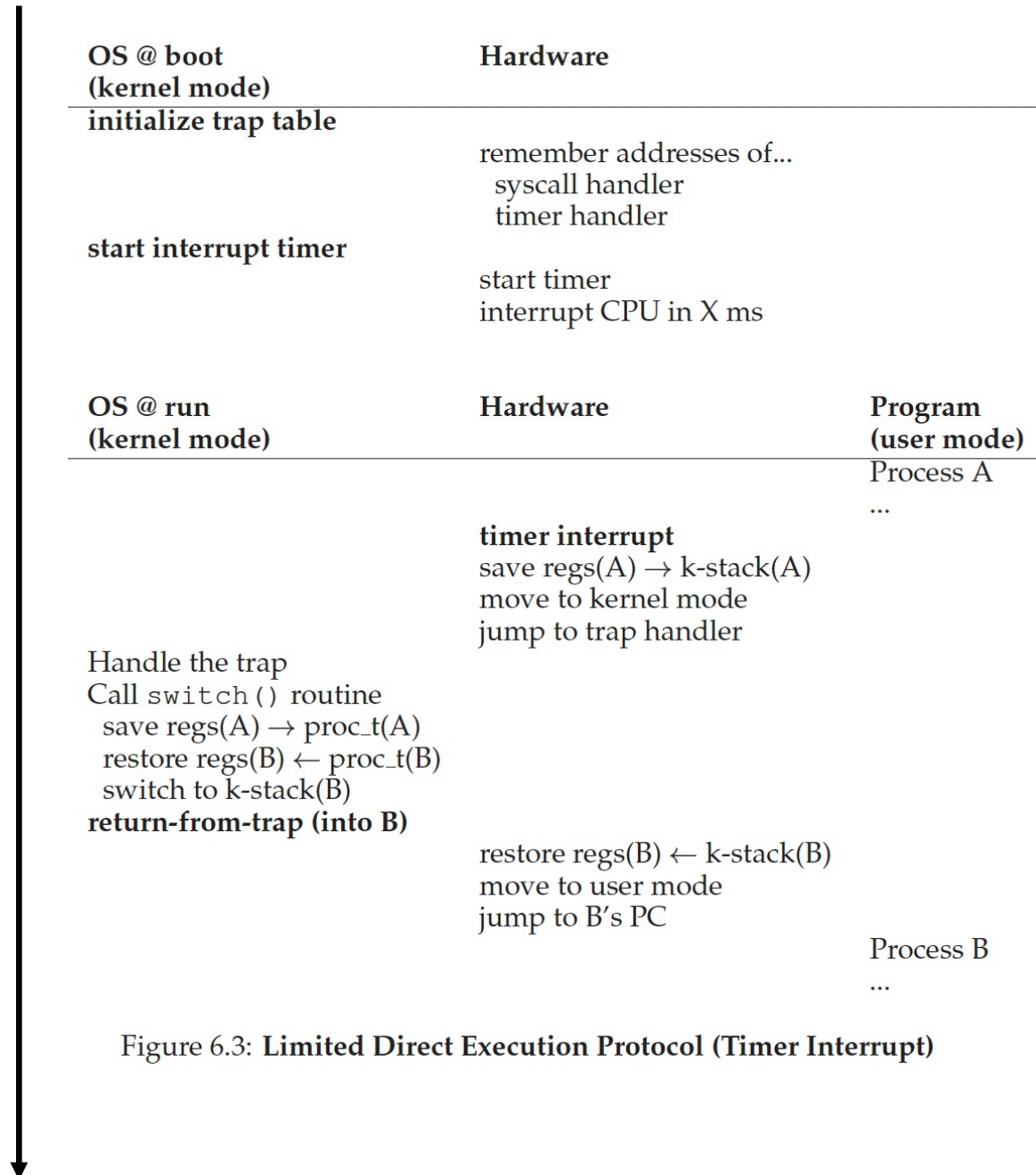
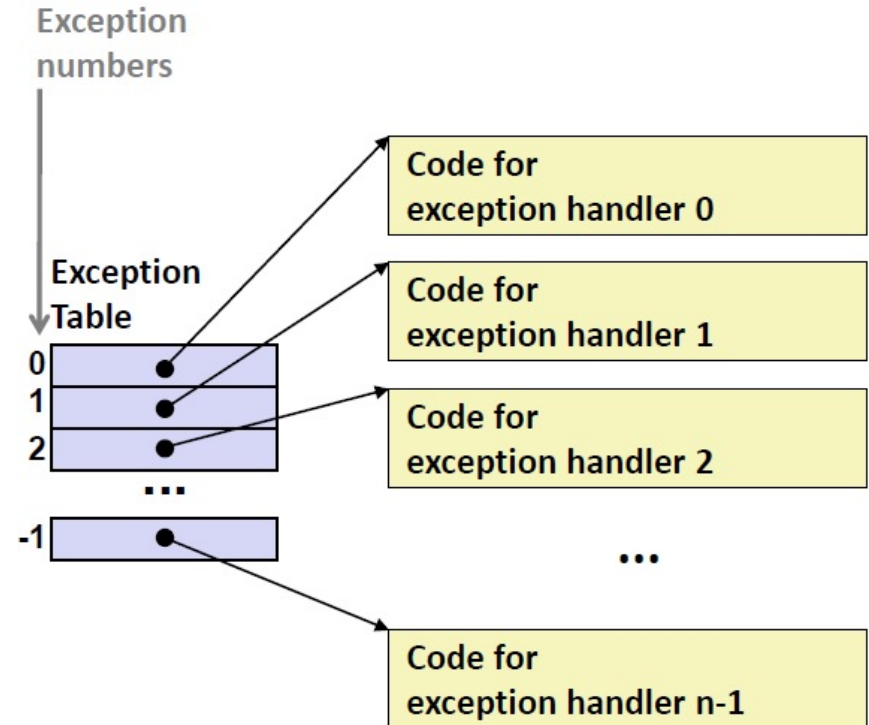


Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

Process must be aware of certain system-wide events

- This is what exactly **exceptions** do!
 - It *does not* mean something bad happens. It is just that something system-wide noteworthy happened.
 - Once an exception happens, CPU hardware saves some context and fetched the corresponding handler by using the exception number to index into the exception table, which is configured during booting. [A privileged operation.]
 - After that it is showtime for the software! If the exception is recovered, it resumes the original process.



What nippy interplay between hardware and software :)

Do you want to read the exception table?

- Oops, OS designer may not offer you such an interface.
- But when you write an OS, it is visible to you hahaha.
- But as least you can see the how the exception number corresponds to the exception from docs.

Trap (mostly system calls): Intentional, generally benign

- Your program solicits help from kernel.
 - IO requests. Access process info from Process Control Block (**PCB**). Terminate current process (since it gives the control away). Process control (fork, execve, waitpid, etc.) and so on.
- A special calling inst is required. [Raise the privilege level] In x86, this is `syscall`.
- A special return inst is required. [Restore the context (namely registers, but not only general-purpose ones). Degrade the privilege level.] In x86, this is `sysret`.
- System call handler is registered in the exception table. Once a `syscall` is triggered, system call handler (or referred to as the entry point) is invoked. It in turn calls the specific system call handlers according to the system call number pushed down the stack.
- System call table is also configured during booting.

Do you want to read the system call table?

- Oops, OS designer may not offer you such an interface.
- Why should it use system call number in lieu of the pointer to the handler?
 - Intentional **transparency** of critical kernel information to the user process.
- But as least you can see how the system call number corresponds to the system calls.

Other traps: debug, breakpoints and overflow (when INTO is executed and OF is on)

```
/* asm/unistd_64.h */

// About 350 system calls in total
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
...
#define __NR_faccessat2 439

#endif /* _ASM_X86_UNISTD_64_H */
```

```

#include <string.h>
#include <unistd.h>

void foo(const char *ptr, size_t len){
    asm("movq %rsi, %rdx; \
        movq %rdi, %rsi; \
        movq $1, %rax; \
        movq $1, %rdi; \
        syscall");
    return;
}

void bar(const char *ptr, size_t len){
    write(1, ptr, len);
    return;
}

int main(){
    const char p[] = "Hello World!\n";
    foo(p, strlen(p));
    bar(p, strlen(p));
}

```

// Linux上write的系统调用号为1，放在%rax

// 第二个参数%rdi(write的第一个参数)表明输出文件描述符，我们选择标准输出1

// 第三个参数%rsi表明缓冲区，根据foo的参数列表和x86-64传输惯例，在foo被调用时存放在%rdi中

// 第四个参数%rdx表明写的长度，根据foo的参数列表和x86-64传输惯例，在foo被调用时存放在%rsi中

Interrupts: Asynchronous

- **IO interrupts:** most often, DMA, and access to other peripheral device.
- **Timer interrupts:** scheduling, or user timer.

Faults: General protection error, might recoverable

- **Memory permission violation:** page faults, copy on write, write to a read-only area, illegal access, etc.
- **Privilege level violation.**
- **Operational exception:** Divide by 0, etc.

Aborts: Severe hardware errors or software bugs, unrecoverable

- **Double fault:** an exception occurs while the CPU is trying to call an exception handler.
- **Triple fault:** a triple fault occurs when an exception is generated when attempt to call the double fault exception handler. It results in the *processor resetting*.
- **Machine check:** a fatal hardware error

异常的种类	是同步 (Sync) 的吗?	可能的行为?		
		重复当前指令	执行下条指令	结束进程运行
中断 Interrupt			✓	
陷入 Trap	✓		✓	
故障 Fault	✓	✓		✓
终止 Abort	✓			✓

行为	中 断	陷 入	故 障	终 止
执行指令 <code>mov \$57, %eax; syscall</code>		✓		
程序执行过程中, 发现它所使用的物理内存损坏了				✓
程序执行过程中, 试图往 <code>main</code> 函数的内存中写入数据			✓	
按下键盘	✓			
磁盘读出了一块数据	✓			
用户程序执行了指令 <code>lgdt</code> , 但是这个指令只能在内核模式下执行			✓	

You should tell the difference between a program and a process!

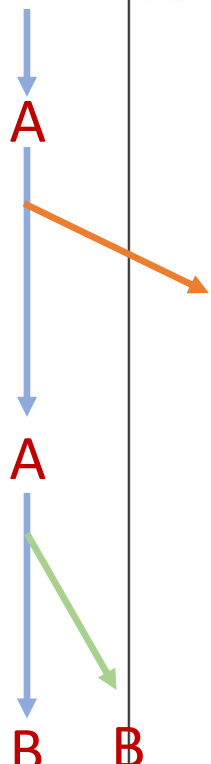
Process control

- After fork, the scheduling order of parent and its child is invisible to the user process. Your program shall not make any assumptions on it to work properly!
- After fork, parent and its child has their own address space. They no longer affect each other!
- After execve, a new program is loaded, as if it is a new process.
- On exit, the process is just terminated and becomes a zombie. That is, the memory and file resources are all destructed. It only takes up an entry in the PCB. **Why is it necessary?**

Why zombie process is necessary?

- To put it simple: For parent to know the exit status and resource usage of its children.
- One more thing to bear in mind: If the parent process never calls `wait()`, then the child process is reparented to the `init` process when the parent process dies, and `init` will `wait()` for the child.

2. 阅读下列程序



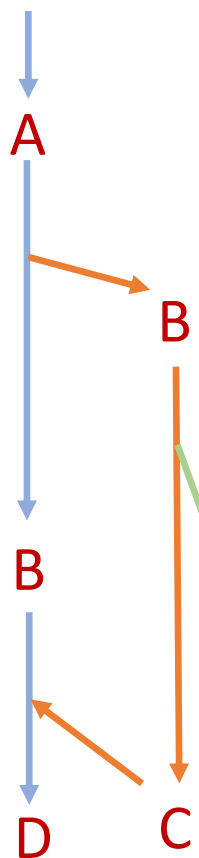
```
int main() {
    char c = 'A';
    printf("%c", c); fflush(stdout);
    if (fork() == 0) {
        c++;
        B printf("%c", c); fflush(stdout);
    } else {
        printf("%c", c); fflush(stdout);
        fork();
    }
    C c++;
    printf("%c", c); fflush(stdout);
    return 0;
}
```

假设系统调用成功，所有子进程都正常运行。判断下列哪些输出是可能的：

- (1) (☒) AABBBC (2) (☒) ABCABB (3) (☐) ABBABC
(4) (☐) AACBBC (5) (☒) ABABCB (6) (☐) ABCBAB

3. 阅读下列程序

```
int main() {
    int child_status;
    char c = 'A';
    printf("%c", c); fflush(stdout);
    c++;
    if (fork() == 0) {
        printf("%c", c); fflush(stdout);
        c++;
        fork();
    } else {
        printf("%c", c); fflush(stdout);
        c += 2;
        wait(&child_status);
    }
    printf("%c", c); fflush(stdout);
    exit(0);
}
```



假设系统调用成功，所有子进程都正常运行。判断下列哪些输出是可能的：

- (1) (☒) ABBCCD (2) (☒) ABBCDC (3) () ABBGCC
(4) () ABDBCC (5) () ABCDBC (6) () ABCDCB

5. 关于进程，以下说法正确的是：

- A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起 I/O 操作。
- B. 调用 `waitpid(-1, NULL, WNOHANG & WUNTRACED)` 会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回 0；如果有停止或终止的子进程，则返回其中一个的 ID。
- C. `execve` 函数的第三个参数 `envp` 指向一个以 `null` 结尾的指针数组，其中每一个指针指向一个形如 “`name=value`” 的环境变量字符串。
- D. 进程可以通过使用 `signal` 函数修改和信号相关联的默认行为，唯一的例外是 `SIGKILL`，它的默认行为是不能修改的。

答案：C

说明：C 正确见 P521。A 不正确 见 P510。B 中 `option` 参数应使用 `|` 运算结合 (P517)。D 中 `SIGKILL` 不是唯一的例外，例外共有两个 `SIGKILL`、`SIGSTOP` (P531)。

Case Study: 真实案例 #Live

在2018年的ICS课堂上，老师确实给同学布置了一个作业，在Linux上写出一份代码，运行它以后，输出能创建的进程的最大数目。下面真的是几位同学的答案。

PART A. Alice同学的答案是：

```
int main() {
    int pid;
    int count = 1;
    while((pid = fork()) != 0){
        // parent process
        count++;
    }
    if(pid == 0) {
        // child process
        exit(0);
    }
    printf("max = %d", count);
}
```

现场演示走起~

IPC: Help processes communicate with each other

Signals / Pipes / Socket / Message queues / Semaphore / Monitor / Shared memory / ...

What ICS cover: signal (Chapter 8), shared memory (Chapter 9), Semaphore (Chapter 12)

Signals

- A small message that notifies a process that an event of some type has occurred in the system.

- Triggered by system event or by user. E.g. From terminal,

```
> sleep 1000 &  
[1] 271318  
> kill -TERM 271318    # or %1  
[1] + 271318 terminated sleep 1000
```

- `> man 7 signal` and you can see a list of signals, triggering conditions and its default handler

■ The life of a signal (Cont.)

- Delivery (aka sending): the kernel activates the corresponding bit in the signal bitmap. At this point of time, the signal is said to be pending. [If the bit is already on, the new signal is discarded. This is what we call the *no queueing property*, so do not use signal to count!]
- You can see the ShdPnd (Pending), SigBlk (Blocked), SigIgn (Ignored), SigCgt (Caught) via the proc file system.

```
> cat /proc/<pid>/status | grep "Sig"
```

Signals

■ The life of a signal (Cont.)

- Once the process is about to return from kernel mode to user mode (e.g. after a context switch), the kernel checks the signal bitmap. If there is one unmasked, it either handles it in a default way, in a user-defined way or just ignores it.
- Upon this time, we said that the signal is received. (It is received only when it is unblocked)

■ Other minor issues:

- Signals that cannot be ignored (namely **SIGKILL** and **SIGSTOP**) cannot be blocked.
- To avoid infinite self-recurrency, the kernel blocks any pending signals of the type currently being processed by a handler. [Implicit blocking]
 - It is still possible to cause infinite loop. E.g. The code on the next slides.
- Once a process is stopped (aka suspended, e.g., after **Ctrl+Z** sends a **SIGTSTP**), no signal except for **SIGCONT** and **SIGKILL** will be received until it is continued.
- SIGCHLD is sent to the parent once its state is changed [*Not only when it is terminated*], namely when it exits, interrupted or resumes after being interrupted. So you must deal with SIGCHLD correctly in your sigchld_handler in tshlab.
 - Hint: `waitpid(-1, &old_status, WNOTRACE | WCONTINUED | WNOHANG)`

```

#include <signal.h>
#include <unistd.h>

void sig1_handler(int signum) {
    kill(getpid(), SIGUSR2);
    sleep(1);
    write(1, "hello\n", 7);
}

void sig2_handler(int signum) {
    kill(getpid(), SIGUSR1);
    sleep(1);
    write(1, "world\n", 7);
}

int main() {
    signal(SIGUSR1, sig1_handler);
    signal(SIGUSR2, sig2_handler);
    kill(getpid(), SIGUSR1);
    while (1);
}

```

Infinite loops

```

# One possible output
> ./a.out
world
hello
world
hello
world
hello
world
hello
world
hello
...

```

In-class Exercise

- First let us walk through **Exercise 8.23**.
- Write a program that forks two children, `chld1` and `chld2`. Use only signals to print the pid of `chld1` in `chld2` and vice versa.
 - *Do not* use the function `sigaction`.
- **Hint:** at most 2 signals suffices. To this end, you may use `SIGUSR1` and `SIGUSR2`.
- **Crux:** How to use signals to count a number? I.e. to send the pid of `chld2` to `chld1`?

Buffer & Cache

缓冲 & 缓存

- Cache is a smaller and fast memory component in the computer which is inserted between the CPU and the main memory. It is hardware-implemented.
 - Why cache works is two-fold:
 - Commonly-seen Locality
 - Memory Hierarchy [Speed, cost and storage]
 - Sets, lines, ways, direct mappings, set associativity.
- Buffers are associated with a specific block device [mostly memory], and cover caching of filesystem metadata as well as tracking in-flight pages. It is software-implemented. It aggregates reads and writes of only several bytes to a few of several pages and thus saves the time.
- TLB? Its name is most misleading. It is a cache, not a buffer, de facto.
- If Memory Hierarchy never exists, cache will perish. But buffer still prevails.

Block Device vs Character Device

- Most important features of block device:
 - Random access
 - Data can be fetched and processed in blocks
 - E.g. Memory, Disk
- .. of character device:
 - The driver communicates by sending and receiving single characters
 - E.g. Keyboard

Buffers

- Input buffer & output buffer:
 - What does scanf and printf do when it reads from keyboard or print to the screen?
- The kernel generally does little or no buffering on character devices.
- The kernel does a certain amount of buffering when reading from files in filesystems. [全缓冲]

Buffers

- line-buffered: Flushed when the buffer is full or a '\xa' ('\n') is encountered.
 - E.g. Keyboard, screen.
- Buffered: Flushed when the buffer is full.
 - E.g. Files on disks
- No buffer:
 - E.g. Stderr
- Apart from these, when will a buffer be flushed?
 - fflush()
 - When the file is closed.
 - On exit() [*Not _exit()*] Examples is on the next slides

What is the difference?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello ICS!");
    exit(0);
}
```

```
> ./a.out
>
```

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello ICS!");
    _exit(0);
}
```

```
> ./a.out
Hello ICS!>
```

Can you figure out why `printf` is not async-signal-safe while `sprintf` is even recurrent in the glibc?

- `printf` has a global locking scheme...

- Suppose that the main program is in the middle of a call to a *stdio* function such as [`printf\(3\)`](#) where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls [`printf\(3\)`](#), then the second call to [`printf\(3\)`](#) will operate on inconsistent data, with unpredictable results.

From > man signal-safety

- `sprintf` uses only local variables and status.

[Not specified in the standard though.]

1. 假设缓冲区足够大, 且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)	(2)	(3)
<pre>int main() { printf("a"); fork(); printf("b"); fork(); printf("c"); return 0; }</pre>	<pre>int main() { write(1, "a", 1); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>	<pre>int main() { printf("a"); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>

abc

abc

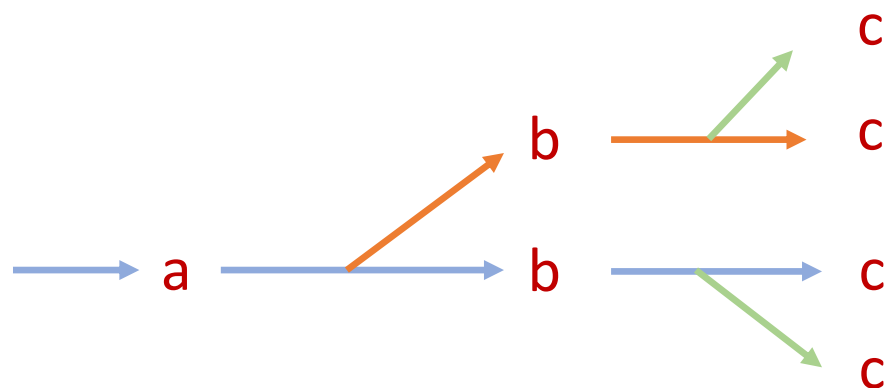
abc

abc

abcbcabcbcabcb

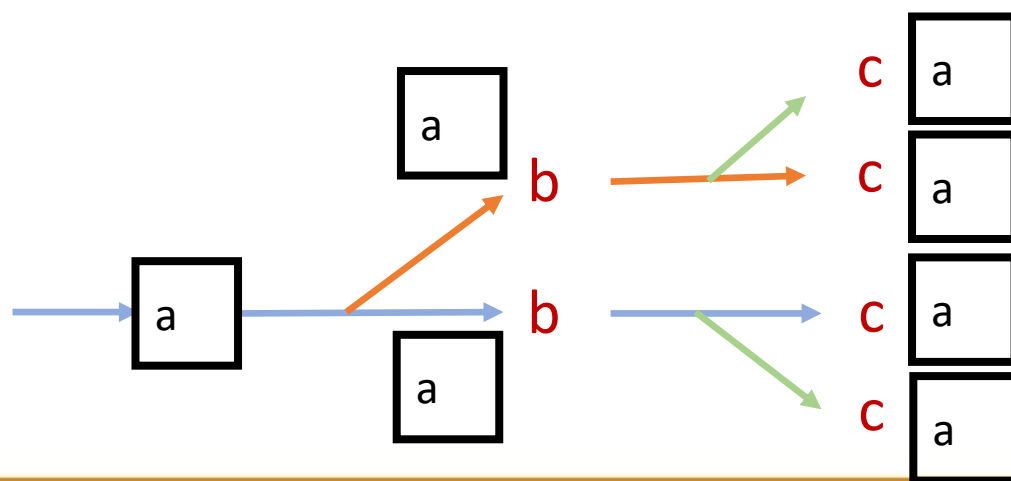
1. 假设缓冲区足够大, 且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)	(2)	(3)
<pre>int main() { printf("a"); fork(); printf("b"); fork(); printf("c"); return 0; }</pre>	<pre>int main() { write(1, "a", 1); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>	<pre>int main() { printf("a"); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>



1. 假设缓冲区足够大, 且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

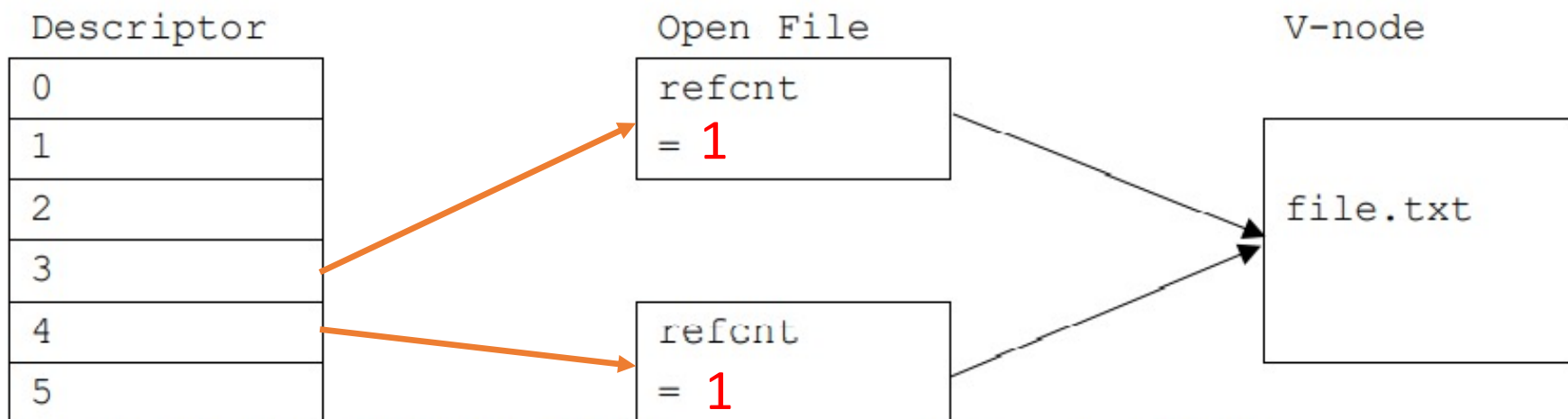
(1)	(2)	(3)
<pre>int main() { printf("a"); fork(); printf("b"); fork(); printf("c"); return 0; }</pre>	<pre>int main() { write(1, "a", 1); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>	<pre>int main() { printf("a"); fork(); write(1, "b", 1); fork(); write(1, "c", 1); return 0; }</pre>



2. 假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。

```
int main() {  
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);  
    int fd2 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);  
    printf("%d %d\n", fd1, fd2);  
    write(fd1, "123", 3);  
    write(fd2, "45", 2);  
    close(fd1); close(fd2);  
    return 0;  
}
```

(1) 程序关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。

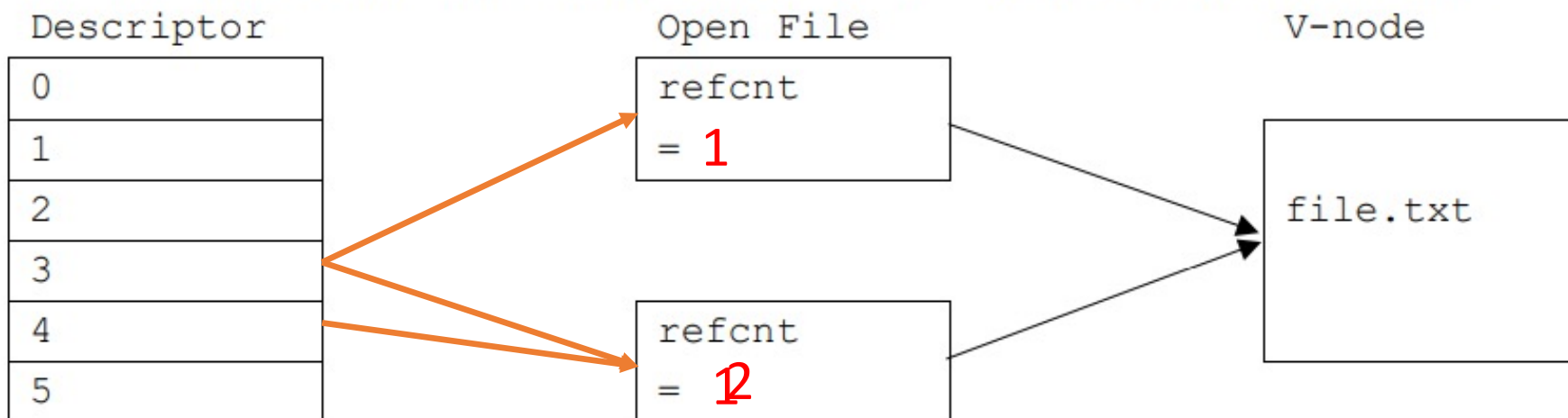


(2) 程序结束时，标准输出上的内容是 3 4，file.txt 中的内容是 453。

3. 假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。

```
int main() {  
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);  
    int fd2 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);  
    dup2(fd2, fd1);  
    printf("%d %d\n", fd1, fd2);  
    write(fd1, "123", 3);  
    write(fd2, "45", 2);  
    close(fd1); close(fd2); return 0;  
}
```

(1) 程序关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。

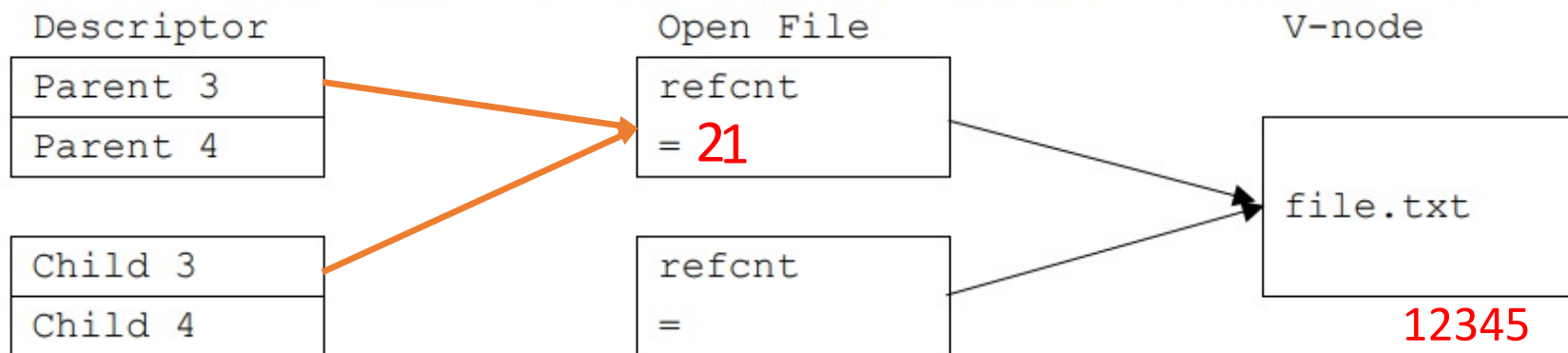


(2) 程序结束时，标准输出上的内容是 3 4，file.txt 中的内容是 12345。

4. 假设磁盘上有空文件 file.txt。程序运行过程中的所有系统调用均成功。缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。

```
int main() {
    pid_t pid; int child_status;
    int fd1 = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if ((pid = fork()) > 0) {                // Parent
        printf("P:%d ", fd1);
        write(fd1, "123", 3);
        waitpid(pid, &child_status, 0);
    } else {                                // Child
        printf("C:%d ", fd1);
        write(fd1, "45", 2);
    }
    close(fd1); return 0;
}
```

(1) 子进程关闭 fd1 前，画出 LINUX 三级表结构。填写 Open File 表中的 refcnt。



(2) 程序结束时，标准输出上的内容是 C:3 P:3，file.txt 中的内容是 或45123

往年题 1

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int c = 1;
void handler1(int sig)
{
    c++;
    printf("%d", c);
}

```

这段代码有几种可能的输出？

Hint: 官答的结果是错的。

```

int main() {
    signal(SIGUSR1, handler1);
    sigset_t s;
    sigemptyset(&s);
    sigaddset(&s, SIGUSR1);
    sigprocmask(SIG_BLOCK, &s, 0);

    int pid = fork() ? fork() : fork() ;

    if (pid == 0) {
        kill(getppid(), SIGUSR1);
        printf("S");
        sigprocmask(SIG_UNBLOCK, &s, 0);
        exit(0);
    } else {
        while (waitpid(-1, NULL, 0) != -1);
        sigprocmask(SIG_UNBLOCK, &s, 0);
        printf("P");
    }
    return 0;
}

```

往年题 2

Part A. (1 分) 此时，X 处语句和 Y 处语句都是 `count++;`，Z 处语句是空语句。Alice 测试该代码，发现有时 `file.txt` 中没有任何输出！请解释原因。（提示：考虑 28 行语句 `fork` 以后，下一次被调度的进程，并从这个角度回答本题。不需要给出解决方案）

Part B. (6 分) Bob 根据 Alice 的反馈，在某两行之间加了若干代码，修复了 Part A 的问题。当 X 处代码和 Y 处代码都是 `count++;`、Z 处为空时，Bob 期望 `file.txt` 中的输出是：

+

可 Alice 测评 Bob 的程序的时候，却发现有时 Bob 的程序在 `file.txt` 中的输出是：

而与此同时，终端上出现了如下的输出：

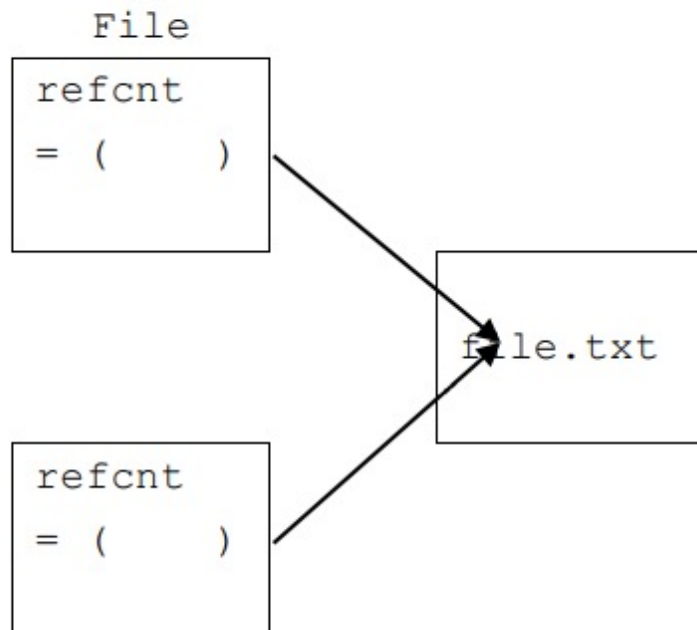
+

Bob 找不到自己的代码的 BUG，只好向 Alice 求助。Alice 帮他做了如下分析：

-																				
+	+																			
	-	-																		
		+	+	+																
			-	-	-															
						+	+	+	+											
							-	-	-	-										
										+	+	+	+	+						
											-	-	-	-	-					
															+	+	+	+	+	+
																-	-	-	-	-
+	-	+	-	-	+	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-

父进程 Parent	() 0
	() 1
	() 2
	() 3

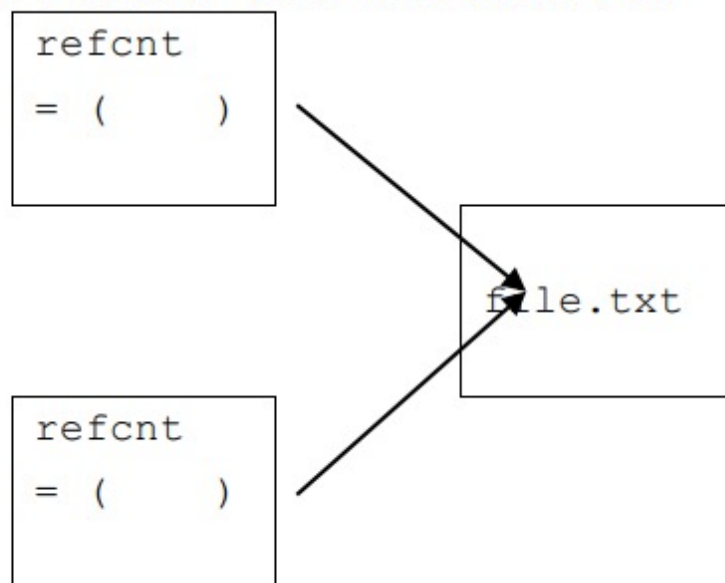
子进程 Child	() 0
	() 1
	() 2
	() 3



分析 2. 当程序第一次在 **file.txt** 中输出+的瞬间，仿照上题要求完成下表：

父进程 Parent	() 0
	() 1
	() 2
	() 3

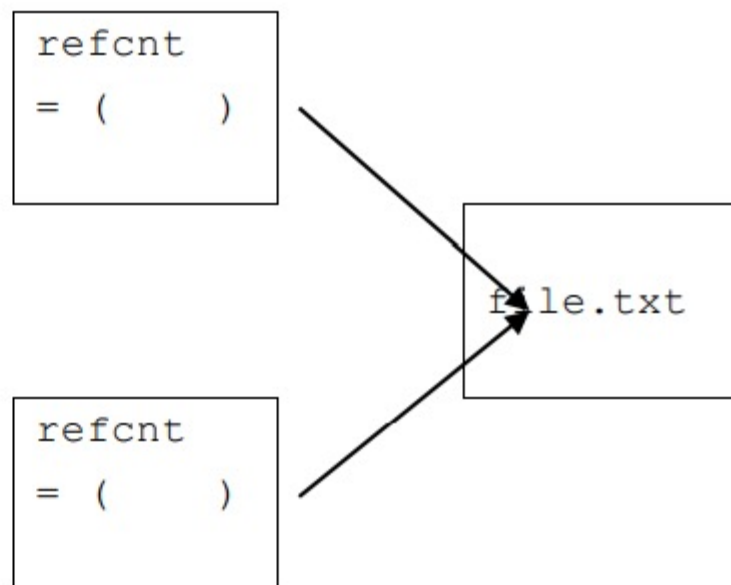
子进程 Child	() 0
	() 1
	() 2
	() 3



分析 3. 如果要产生 Bob 预期的输出，三级表的关系应当是什么？仿照上题要求完成下表：

父进程 Parent	() 0
	() 1
	() 2
	() 3

子进程 Child	() 0
	() 1
	() 2
	() 3



Part C. (2 分) Bob 很高兴,他知道 Part B 的代码是怎么错的了!不过 Alice 仍然想考考 Bob。对于 Part B 的错误代码,如果终端上输出的是+++,那么 file.txt 中的内容是什么?请在下框中写出答案。

++++++-+++----- (6+1-3+2-3+6-)

Part D. (1 分) Bob 修复了 Part B 的问题,使得代码能够产生预期的输出。现在, Bob 又希望自己的代码最终输出的是+---+++-----+++++-----,为此,他对 X、Y、Z 处做了如下的修改。X、Y 处语句已做如下填写,请帮助 Bob 补上 Z 处语句。

X 处填写为: `count += 2;`

Y 处填写为: `count += 2;`

Z 处填写为:

`count = 2;`

任何把 `count` 初始化为 2 的语句都行

演示：进程实现的素数筛

Any questions?