

# ICS 第 '\10' 次小班课

2021-11-16

1. 下图为一个典型的编译过程。将正确的过程填上，并补充缺失的拓展名。

A. 汇编器 as

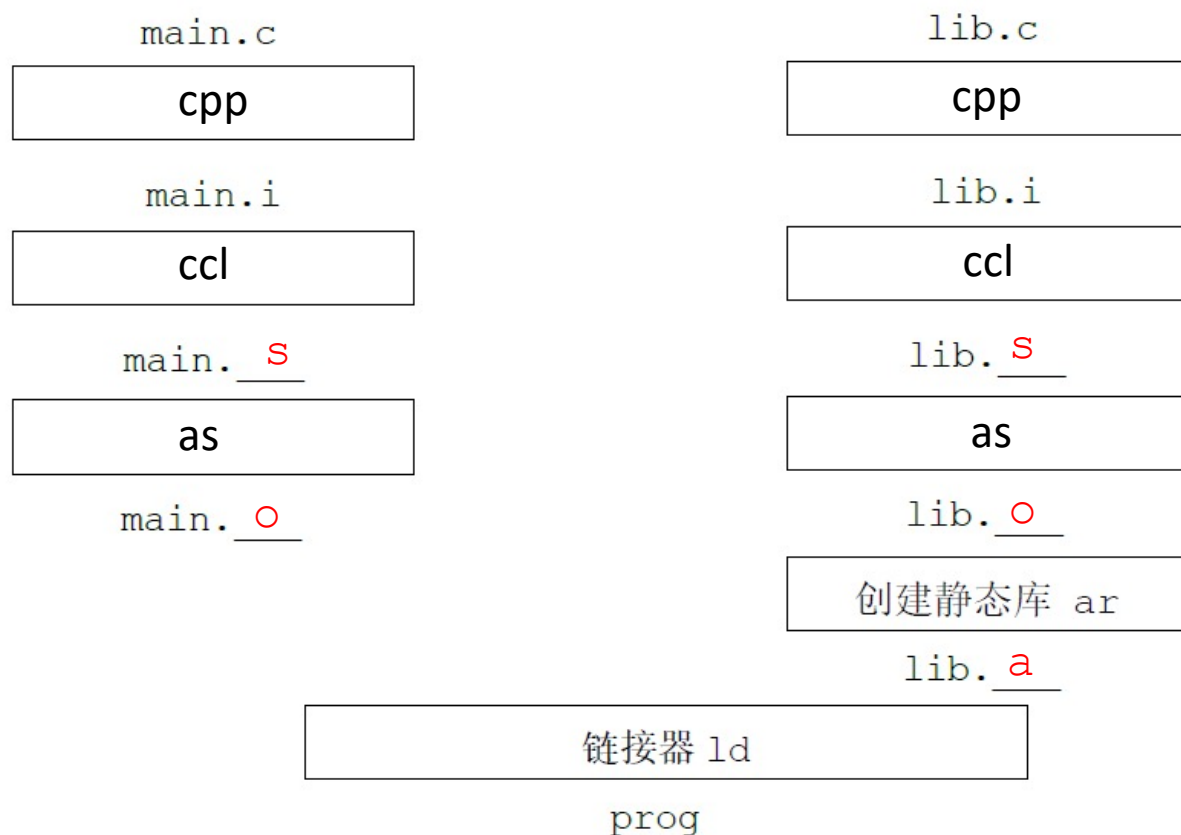
B. 预处理器 cpp

C. 编译器 ccl

D. \*.a

E. \*.s

F. \*.o



```

#include <stdio.h>
int x = 1;
int foo(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}
int bar(int n);
void op(void) {
    x = x + 1;
}
int main() {
    for (int i = 0; i < 3; i++) {
        int a1 = foo(0);
        int a2 = bar(0);
        op();
        printf("%d %d ", a1, a2);
    }
    return 0;
}

```

Annotations for main.c:

- `x` Global strong (points to `x = 1;`)
- `ans` Local (points to `static int ans = 0;`)
- `bar` Global External (.UNDEF) (points to `int bar(int n);`)

main.c

```

int x;
int bar(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}

```

Annotations for count.c:

- `x` Global weak (points to `int x;`)
- `bar` Global strong (points to `int bar(int n) {`)
- `ans` Local (points to `static int ans = 0;`)

count.c

```

> gcc -o a.out main.c count.c
> ./a.out
1 1 3 3 6 6
>

```

```

#include <stdio.h>
static int x = 1;
int foo(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}
int bar(int n);
void op(void) {
    x = x + 1;
}
int main() {
    for (int i = 0; i < 3; i++) {
        int a1 = foo(0);
        int a2 = bar(0);
        op();
        printf("%d %d ", a1, a2);
    }
    return 0;
}

```

main.c

x Local ----

ans.xxxx Local ----

bar Global External (.UNDEF)

```

static int x;
int bar(int n) {
    static int ans = 0;
    ans = ans + x;
    return n + ans;
}

```

count.c

x Local ----

bar Global strong

ans.xxxx Local ----

```

> gcc -o a.out main.c count.c
> ./a.out
1 1 3 2 6 3
>

```

# Bonus 1

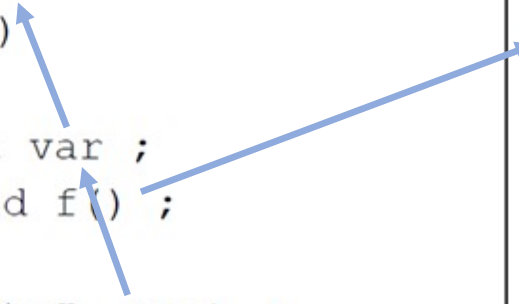
C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为（ ）

- A. 100                      B. 200  
C. 201                      D. 链接错误

```
// f1.c
#include <stdio.h>
static int var = 100;
int main(void)
{
    extern int var ;
    extern void f() ;
    f() ;
    printf("%d\n", var) ;
    return 0;
}
```

```
//f2.c
int var = 200;

void f()
{
    var++;
}
```



## Bonus 2

8. C 源文件 m1.c 和 m2.c 的代码分别如下所示，编译链接生成可执行文件后执行，结果最可能为：

```
// m1.c
#include <stdio.h>

int a1 ;
int a2 = 2 ;
extern int a4 ;

void hello()
{
    printf("%p ", &a1);
    printf("%p ", &a2);
    printf("%p\n", &a4);
}
```

```
//m2.c
int a4 = 10 ;

int main()
{
    extern void hello() ;

    hello() ;
    return 0 ;
}
```

如果再把红框中m1.c m2.c的顺序反过来呢？（Hint: 蓝框是答案）

\$ gcc -o a.out **m2.c m1.c** ; ./a.out

**m1.c m2.c**

A. 0x1083018, 0x108301c

C. 0x1083024, 0x1083028

B. 0x1083028, 0x1083024

**D. 0x108301c, 0x1083018**

ELF header

Section header table

Segment header table (*Rarely use this term, program header table is more usual*)

.rel

*Why do segment not exist in a relocatable object file?*

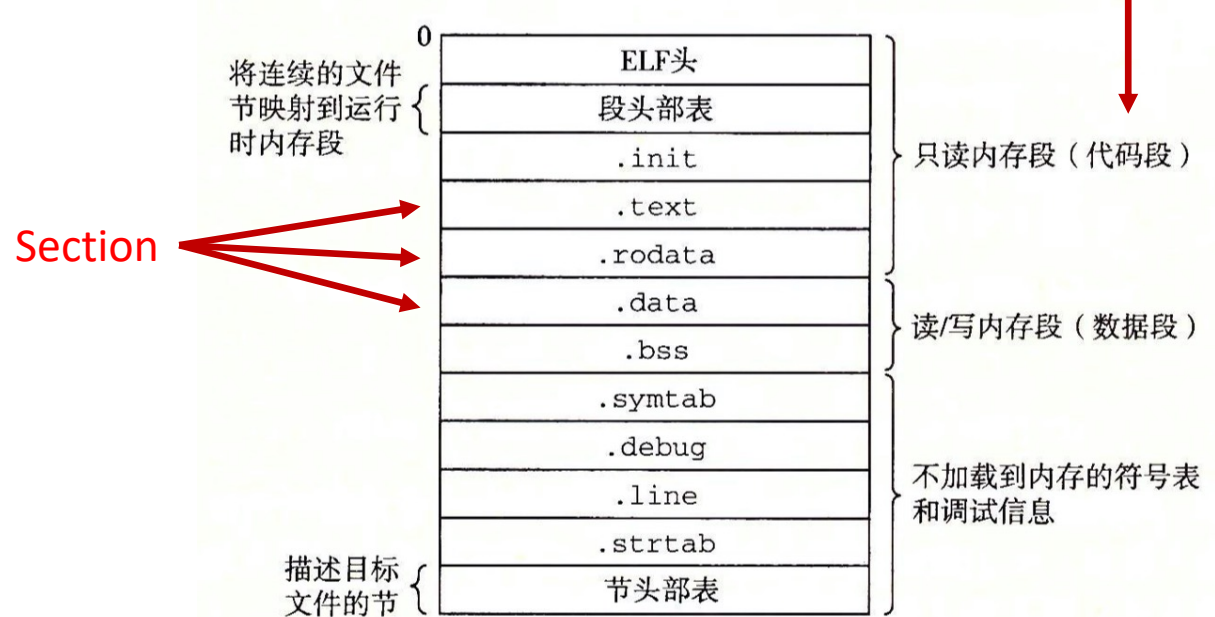
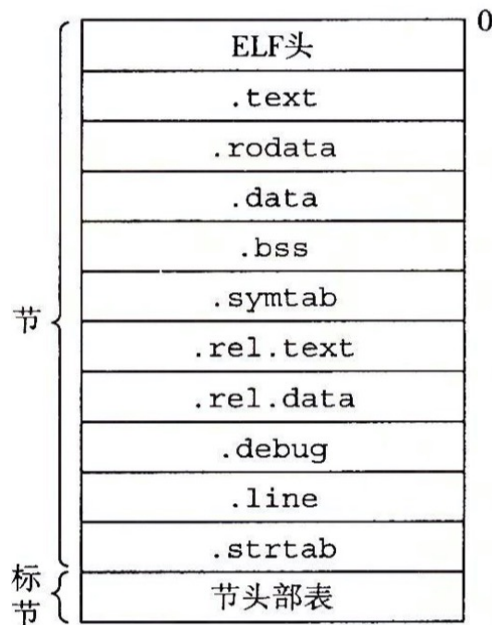
.init

.interp (*Likely to present in modern .o files*)

.rel (*removed if fully linked*)

## Relocatable Object Files

## Executable Object Files

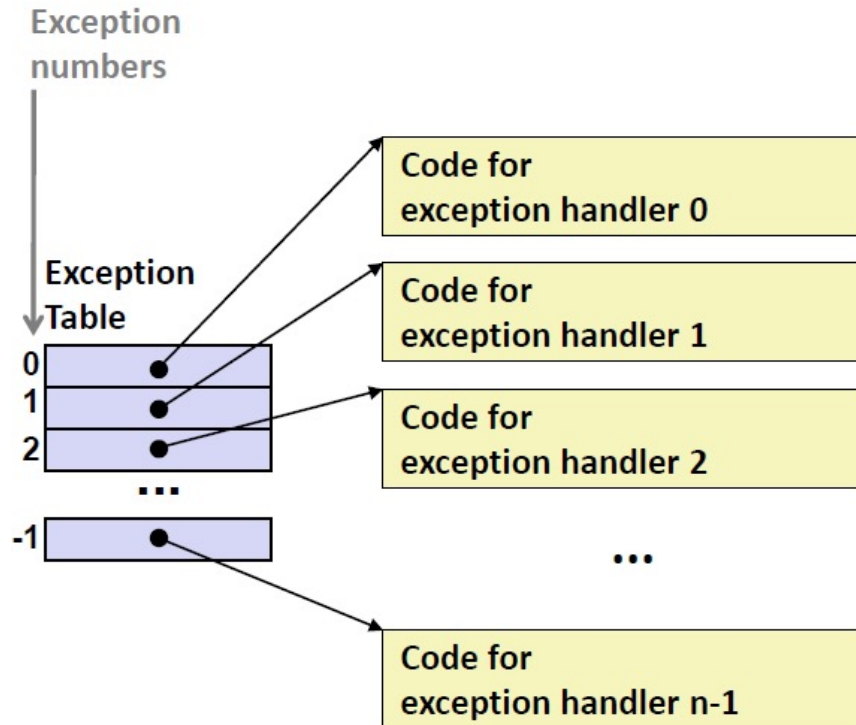


| 异常的种类           | 是同步 (Sync) 的吗? | 可能的行为? |        |        |
|-----------------|----------------|--------|--------|--------|
|                 |                | 重复当前指令 | 执行下条指令 | 结束进程运行 |
| 中断<br>Interrupt |                |        | ✓      |        |
| 陷入<br>Trap      | ✓              |        |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

| 行为  | 中<br>断 | 陷<br>入 | 故<br>障 | 终<br>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 |        |        | ✓      |        |



# Handle exceptions



- 软件和硬件的合作
- 异常表和异常号
- 软硬件的配合：简单地说，
  - 【硬件】处理器检测到事件并确定响应的异常号，通过异常表的相应条目来转到处理程序
  - 【软件】执行处理程序，完成后执行“从中断返回”指令

**Additional(Not in Exams):** What about traps, since they are specified by a system call number? How do these two numbers (exception No. & syscall No.) reconcile?

# Process & Program

- 异常提供了进程这一概念实现的条件，并提供了事件驱动的操作系统的基本机制。

-> 事件即异常 (be aware of the diversification on the term, *exception*)

- 进程具有独立的逻辑控制流以及私有的地址空间。操作系统选择策略调度各个进程。操作系统通过给每个进程一个可执行的时间片，并在时间片用完或者其它系统事件发生时切换它们的上下文，来使不同的进程得以安全地并发执行，同时向用户制造出它们在同时执行的假象。
- 进程总是执行中的程序的一个实体。它们唯一的联系在于相应的可执行文件提供了进程逻辑流和地址空间中的初始代码和数据。因此我们并不非常关心进程与源程序的关系。
- 软硬件必须配合实现异常处理的机制。为了保护系统和硬件资源，以及限制用户行为，硬件CPU提供特权级给操作系统使用。用户代码总是要运行在用户态下。处理异常的操作系统代码一般需要运行在内核态下(如果需要使用到用户态不能提供的服务，那么必须在内核态下。)
- 操作系统通过系统调用管理并控制用户对于系统资源有意识的使用。

# Control the Process

## ■ 进程的创建与使用

-> 一系列系统函数或者系统调用接口。后者例如`fork()`, `waitpid()`, `execve()`, 它们封装了更底层的操作系统提供给用户的系统调用。

## ■ 进程间的同步

-> IPC机制 (Inter-process communication), 典型的Linux IPC机制就包括我们下周讲到的信号(signal), 后面的第十二章还将提到基于原语(primitives)操作的信号量(semaphore)的PV操作。其他有用的IPC机制还包括管道(pipes)、锁(lock)、管程(moniter) 等等。这些概念除了signal和信号量 ics 中不作要求, 信号量是第十二章的事。

# Good Habits

- 检查调用的返回值，并在出错时捕获错误并将它反馈给用户。

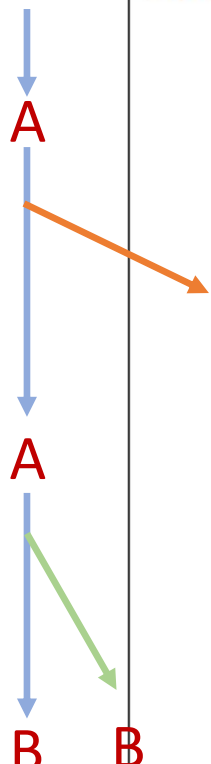
-> 这些函数的设计者有意地将出错设置为返回负值(通常是-1)，其余情况要么返回零，要么返回正数。尽管有少许例外，应该熟记常用函数的返回值。例如 `fork()` `getpid()` `getppid()` `waitpid()` 等等。

- 合适的注释

-> 今年的tshlab (also Shelllab, where tsh stands for tiny shell) 我会适当关注大家注释的有效性(适当即可，不必死抠)。

- 调用某些功能丰富的函数时，使用系统文件定义好的宏显示地指定工作方式。除非你十分确定或者该函数功能众所周知，否则不要使用默认的工作方式调用函数(像`waitpid`这样的太过熟悉就没有必要啦，这种情况 ics 课里暂时遇不到)。

## 2. 阅读下列程序



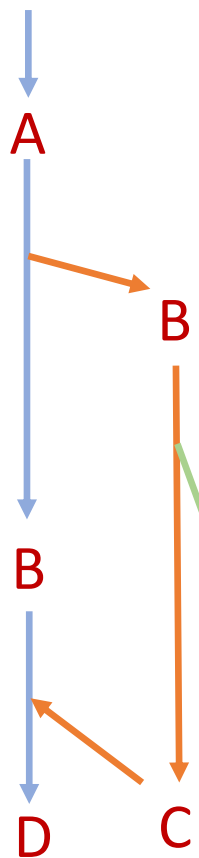
```
int main() {
    char c = 'A';
    printf("%c", c); fflush(stdout);
    if (fork() == 0) {
        c++;
        B printf("%c", c); fflush(stdout);
    } else {
        printf("%c", c); fflush(stdout);
        fork();
    }
    c++;
    B C printf("%c", c); fflush(stdout);
    return 0;
}
```

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)。

**Any questions?**