进程管理
Process Management

钮鑫涛
南京大学
2024春

本门课由蒋炎岩和钮鑫涛共同构建

进程

- 什么是进程? 同样地，没有一个统一的精确定义

A program in execution

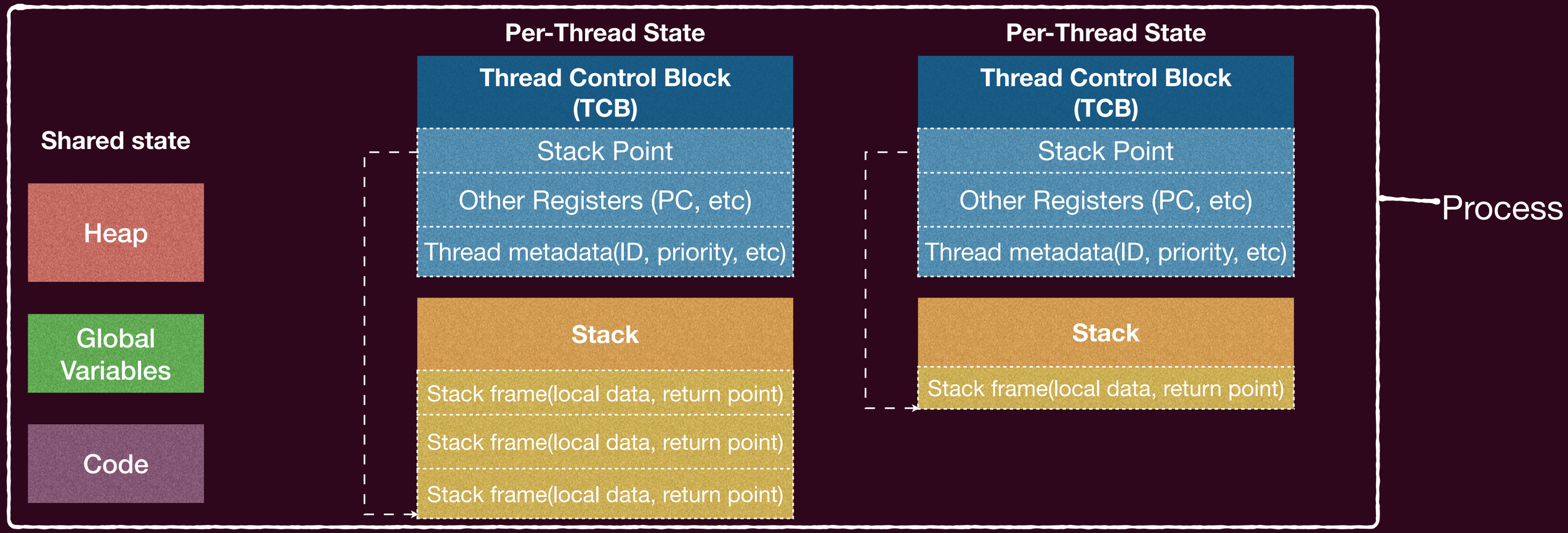
An instance of a running program

The entity that can be assigned to, and executed on, a processor

A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

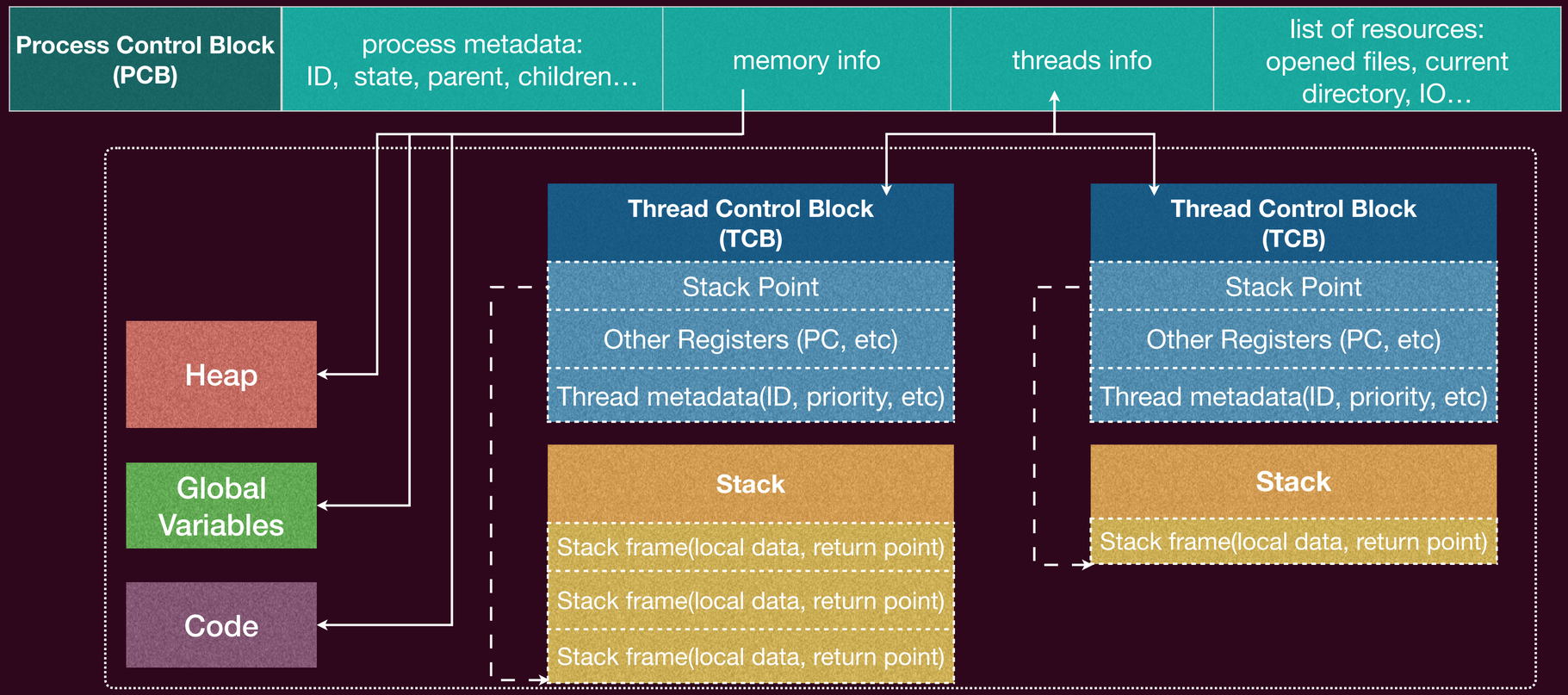
进程

- 在已经有了线程的概念之后，进程的概念其实已经很容易给出：
  - 其就是拥有独立地址空间的运行实体， 可以包含一个或多个线程



进程

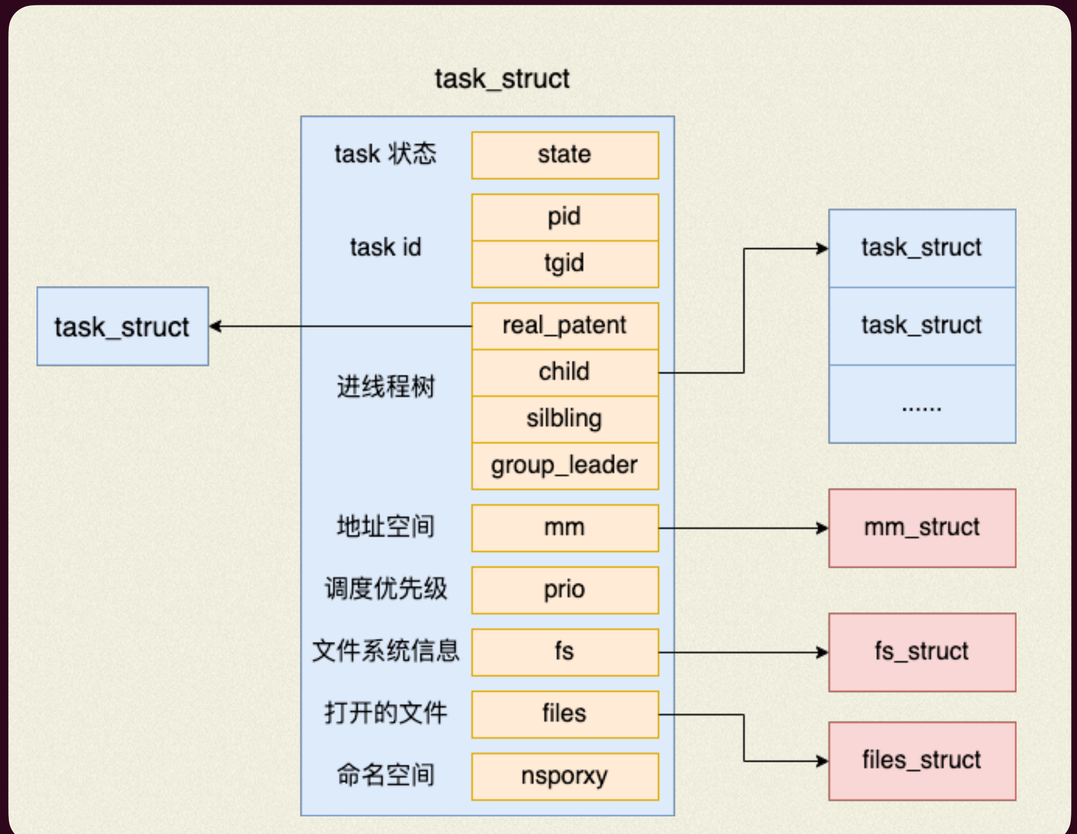
- 与线程的线程控制块（Thread Control Block, TCB）一样，进程也需要一些元信息，用来描述（抽象）进程，方便操作系统管理，即进程控制块（Process Control Block）



元信息（PCB、TCB）由内核维护

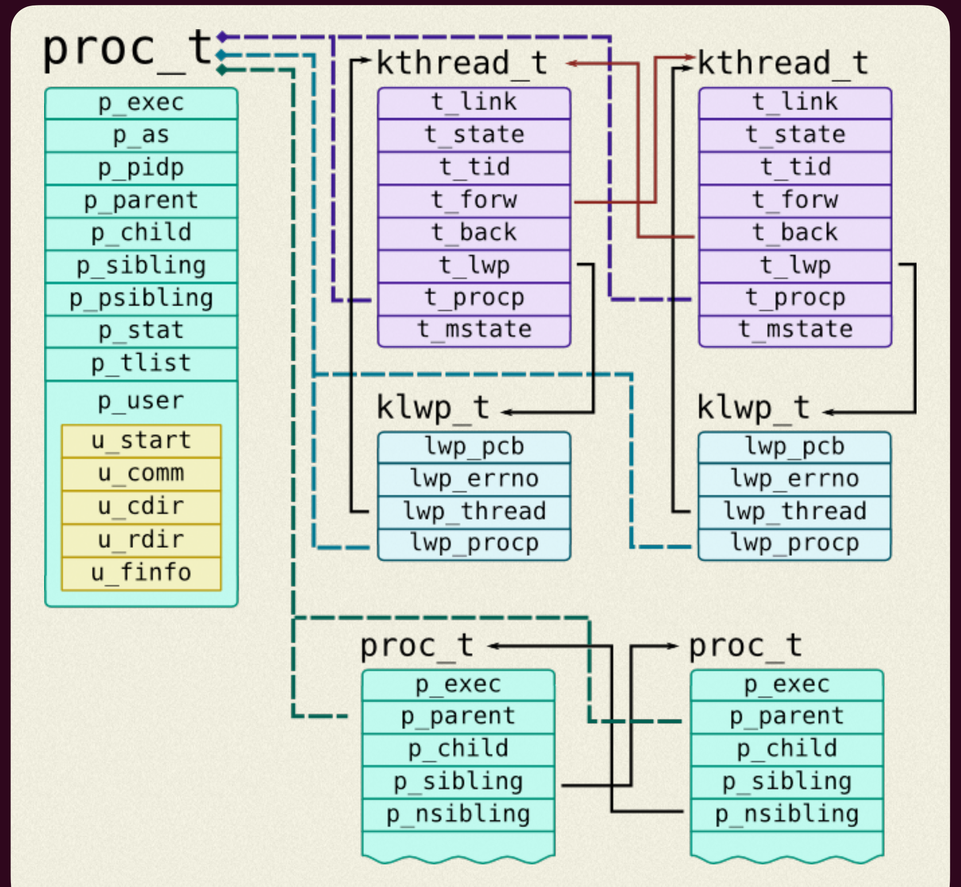
不同的系统，不同的进程实现

- Linux系统的实现中并不刻意区分进程和线程，而是将其统一存储在被称作`task_struct`的数据结构中。当两个`task_struct`共享同一个地址空间时，它们就是同一个进程的两个线程。与Linux进程有关的数据结构定义大多数都在`/include/linux/sched.h`中



不同的系统，不同的进程实现

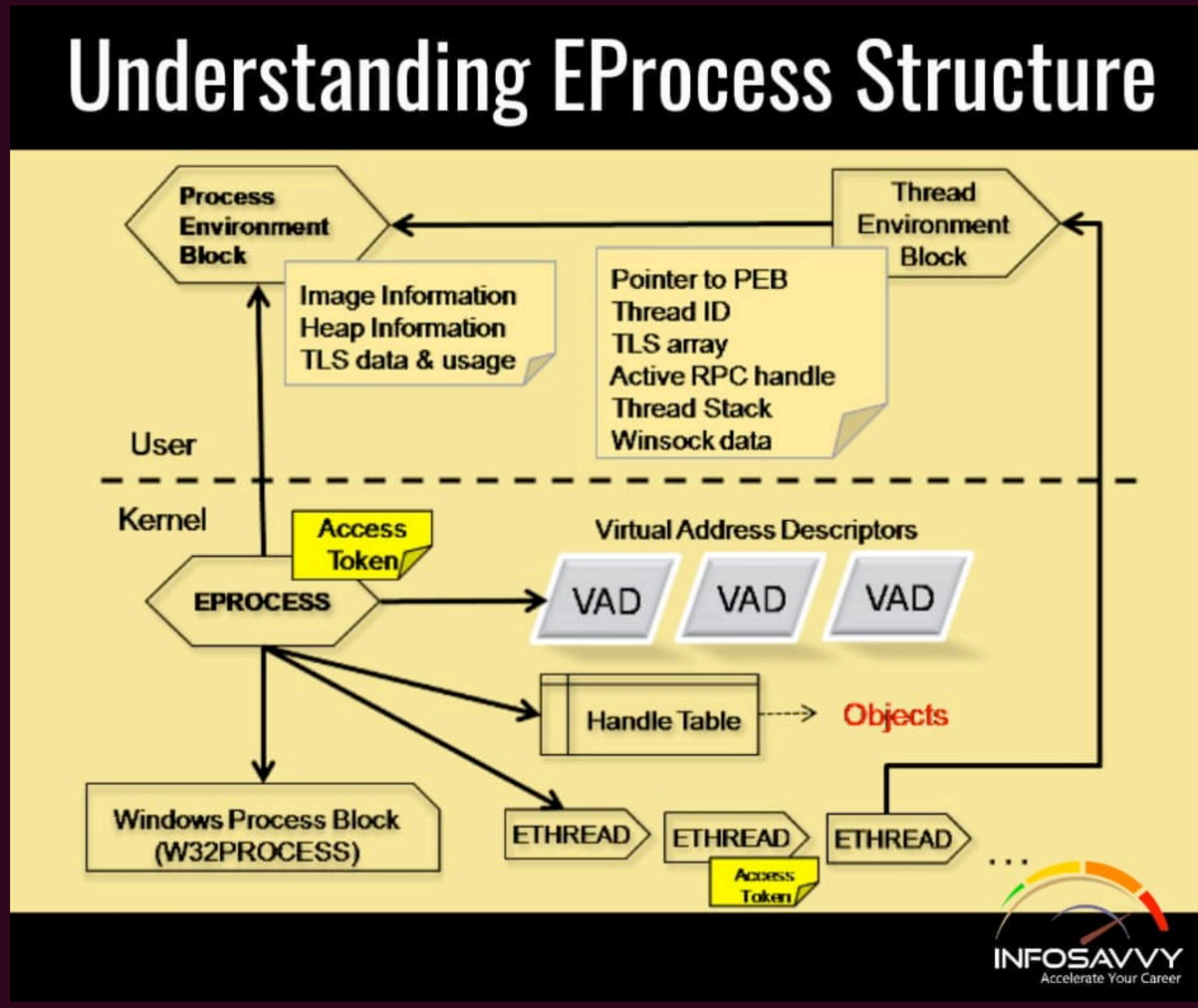
- Solaris操作系统会区分线程和进程：线程在内核态以kthread_t的形式标识，在用户态会给一个klwp_t（Light-Weight Processes）的接口可以访问一些信息，一个或多个线程构成一个进程，这些元信息之间相互引用



不同的系统，不同的进程实现

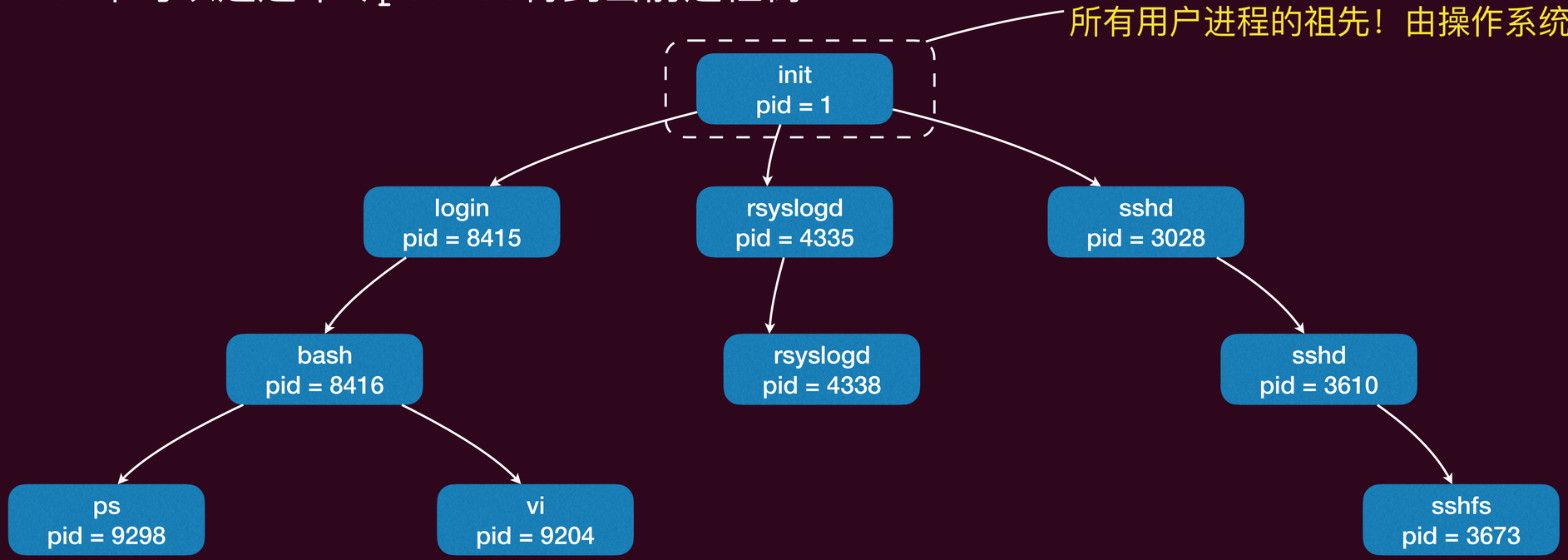
在Windows NT以后的Windows系统中，进程用EPROCESS对象表示，线程用ETHREAD对象表示。在一个EPROCESS对象中，包含了进程的资源相关信息，比如句柄表、虚拟内存、安全、调试、异常、创建信息、I/O转移统计以及进程计时等。每个EPROCESS对象都包含一个指向ETHREAD结构体的链表。

- Windows中的进程和线程除了有KPROCESS和KTHREAD这两个成员对象专门用来存储底层细节外，还提供了Process Environment Block和Thead Environment Block两个对象暴露给应用程序来访问



进程树

- 一个进程是被某个进程所创建的，一个进程也可以创建多个进程
  - 因此操作系统的进程可以构成一个进程树（Process Tree），其中父进程节点指向其所创建的子进程
  - linux下可以通过命令`pstree`得到当前进程树

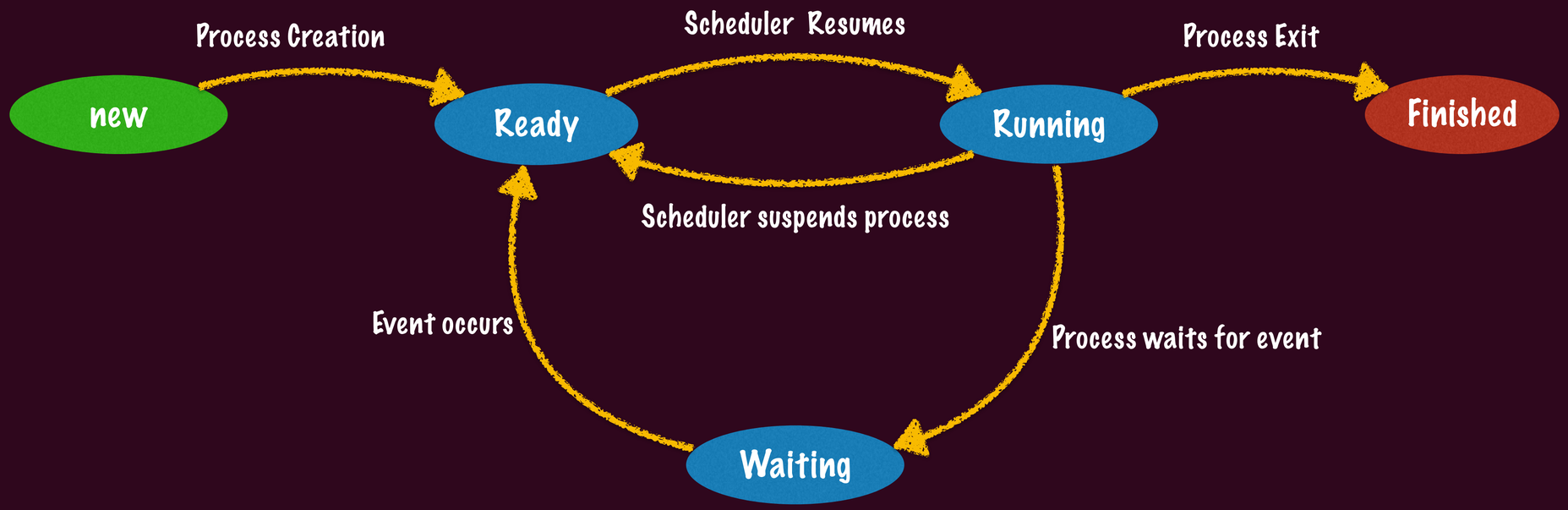


第一个进程init

- CPU reset → Firmware代码执行→加载操作系统并初始化→加载第一个进程
  - 第一个进程的创建是个漫长的过程，因为其需要的很多资源都需要从0开始准备
- init进程加载完之后，操作系统已经完成了所需要资源的管理，并“躺”在后台等待用户命令（syscall）或者中断的发生
  - 之后用户进程的创建就变得容易，第一个“蛋”/“鸡”有了，后面就是重复“鸡生蛋，蛋生鸡”的过程了
- 我们甚至可以“定制”自己的init进程

进程的生命周期

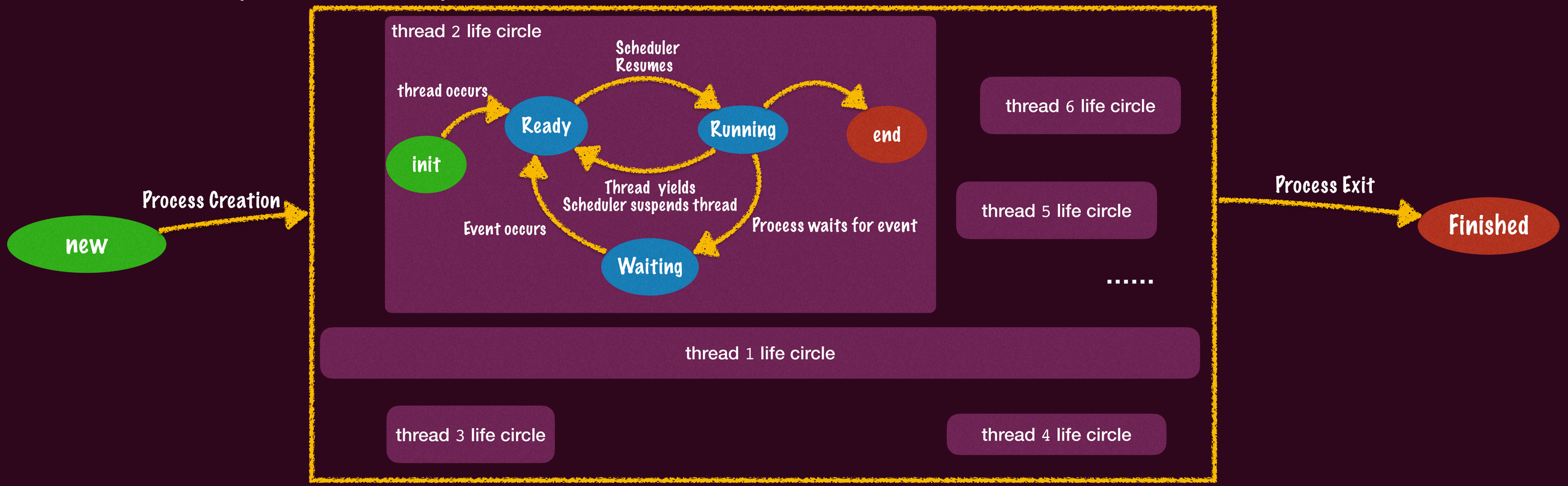
- 进程创建之后就开始了其运行，直到终止，其生命周期可以描述为如下几个状态(和线程类似)



单线程进程

进程的生命周期

- 进程创建之后就开始了其运行，直到终止，其生命周期可以描述为如下几个状态(和线程类似)



多线程进程

进程的生命周期

- 终止之后的进程的“资源”会被操作系统回收，并通知其父进程其终止状态
  - 为什么要通知?
    - PCB中父进程含有指针指向子进程PCB，如果进程终止之后连PCB也都被回收，那么该指针就会指向一个“已经”被释放的内存
    - 已终止但其PCB信息还没被回收的进程被称为“僵尸进程”（Zombie process） 
    - 父进程调用wait系统调用会得到子进程的退出通知（子进程退出前会阻塞父进程）和其退出状态，同时移除该子进程的PCB

进程的生命周期

- 但并不是每个父进程都老老实实的wait子进程的，其完全可能在子进程终止前就终止了：这时的子进程是没有父进程的！
  - 该状态下（没有父进程）的进程被称为“孤儿进程”（Orphan Process）
  - linux会让init进程重新接管这些孤儿进程，从而能够在这些孤儿进程终止时回收PCB资源

进程最重要的三类系统调用

- fork —进程的创建
- execve —进程的改变
- exit —进程的删除

fork

电影：致命魔术

fork系统调用

- 创建一个新的（子）进程
  - 通过做一份当前进程完整的复制 (内存、寄存器现场)
  - 子进程和父进程会各自独立地继续执行fork之后的指令
- 如何区分父子进程?
  - fork的返回值不同: 子进程返回 0，父进程返回子进程的process ID,
  - fork出错返回-1
    - errno 会返回错误原因 (man fork)

```
#include <unistd.h>
pid_t fork(void);
```

fork的行为

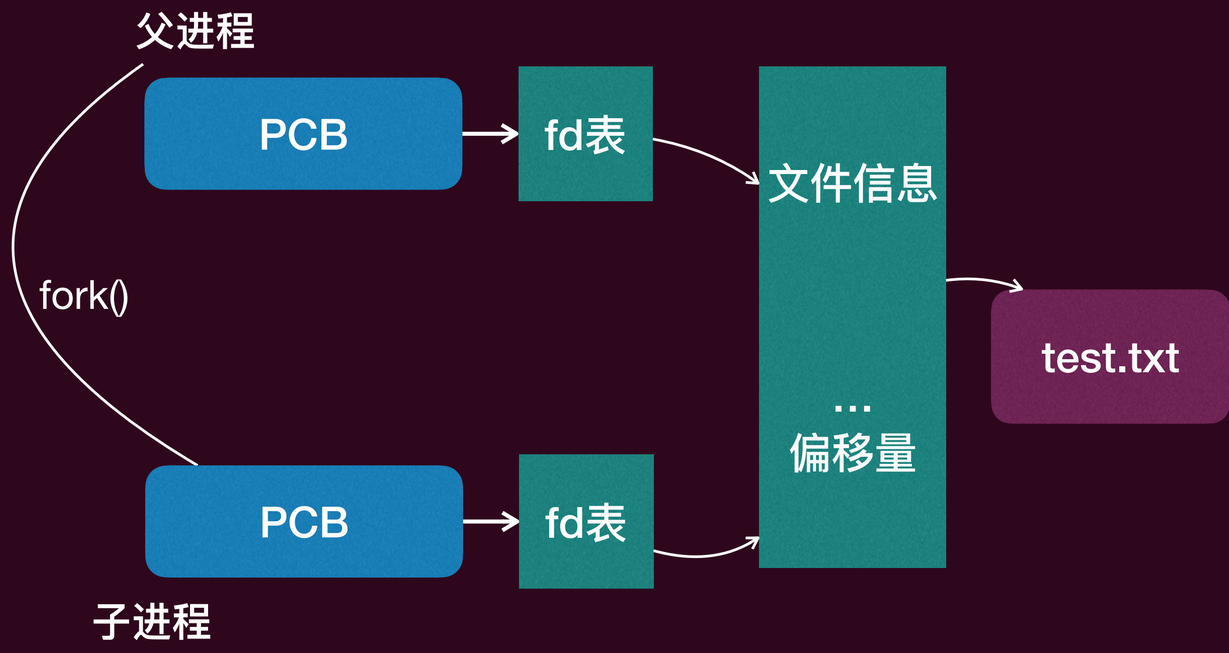
- 立即复制状态机
  - 包括**所有**信息的完整拷贝
    - 每一个字节的内存
    - PCB里的信息

```c
int x = 42;
int ret = fork();
if(ret < 0){
    //fork失败
    fprintf(stderr, "Fork failed\n");
}else if (ret == 0){
    //子进程
    printf("Child process id: %d, and the value of x is %d \n",  ret, x);
}else{
    //父进程
    printf("Parent process id: %d, and the value of x is %d \n",  ret, x);
}
```

fork的行为

- 注意，不只是地址空间的对象被复制了，PCB里的对象，如打开的文件描述符号也被一并复制！

```
char str[11] = {0};
read(fd, str, 1);
int fd = open("test.txt", O_RDWR);
if (fork() == 0) {
    ssize_t cnt = read(fd, str, 9);
    printf("Child process: %s\n", (char *)str);
} else {
    ssize_t cnt = read(fd, str, 9);
    printf("Parent process: %s\n", (char *)str);
}
close(fd);
```



文件描述符

- 文件描述符：一个指向操作系统内对象的 “指针”
  - 对象只能通过操作系统允许的方式访问
  - 从 0 开始编号 (0, 1, 2 分别是 stdin, stdout, stderr)
  - 可以通过 open 取得；close 释放； dup 复制；
  - 对于数据文件，文件描述符会 “记住” 上次访问文件的位置

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
int open(const char *pathname, int flags);
```

O_RDONLY, O_WRONLY, or O_RDWR

fork的行为

- 事实上，由于存在父子进程存在大量的共享，会造成很多不确定性，真实系统的fork的背后的实现是复杂的
  - 比如父子进程之间的process id是不同的，进程的parent process id也是不同的
  - 比如，多线程的进程进行fork时，只有一个线程被复制，就是那个调用fork的线程
    - （man 2 fork）The child process is created with a single thread—the one that called **fork**().
- Posix标准列出了调用fork时25种特殊情形的处理方法
  - 包括进程ID、文件的处理、锁、计时器、消息队列等

fork习题

- 多个fork会怎么样?
  - 总共创建了多少进程?

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

int main() {
    pid_t x = fork();
    pid_t y = fork();
    printf("%d %d\n", x, y);
}
```

如果是:

```c
while(1) fork();
```

fork炸弹💣，不要尝试，除非你想让你的电脑宕机

fork习题

- fork 加上管道？下面的命令结果是多少？

./a.out — line buffered

./a.out | cat — fully buffered

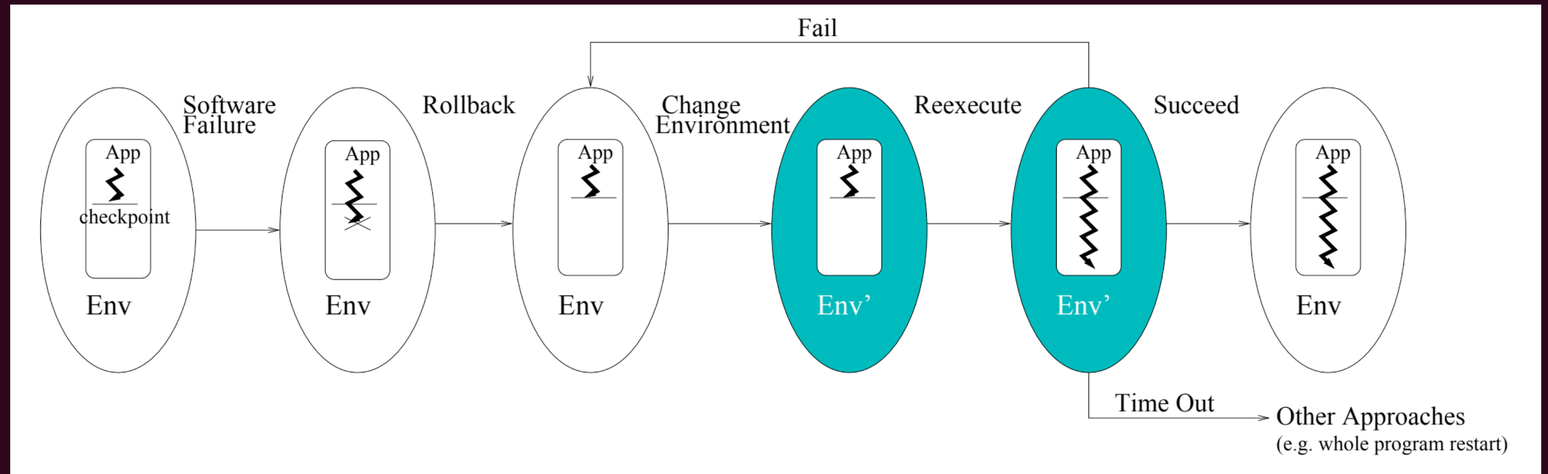
```c
#include <unistd.h>
#include <stdio.h>

int main() {
    for (int i = 0; i < 2; i++) {
        fork();
        printf("Hello\n");
    }
}
```

改为 “Hello\t” 试试

fork的小应用

- 由于fork的能力，一个简单的应用就是可以用来给进程创建“快照”
- 主进程 crash 了，启动快照重新执行
  - 有些 bug 可能调整一下环境就消失了 (比如并发)
  - Rx: Treating bugs as allergies--A safe method to survive software failures. (SOSP'05, Best Paper Award)



execve

电影：黑客帝国

execve

- 很多时候，父子进程并不是同样的逻辑，子进程有自己的逻辑，比如shell创建进程并不是为了子进程也是一个shell，而是“加载”某个可执行文件进行运行
  - execve就是这样的系统调用
    - 其参数重，pathname是要加载的可执行文件具体的路径，argv进程执行所需要的参数（main函数同款参数），envp是

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

execve的行为

一句话：将当前进程重置成一个可执行文件描述状态机的初始状态

- 加载pathname指定的可执行文件（数据段、代码段）
- 重新初始化堆和栈
- PCB中相应的memory mappings也会改变
- 将PC寄存器设置到可执行文件代码段定义的入口点，该入口点最终会调用main函数

A successful call to execve does not have a return value because the new process image overlays the calling process image

fork + execve

- 创建一个新的子进程的组合拳

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

extern char **environ; //environment variables of current process
while(1){
    type_prompt();  // display prompt on the screen
    read_command(&filename, &parameters); //read the input command
    int pid = fork();
    if (pid == -1) {
        perror("fork");
    } else if (pid == 0) { // Child
        execve(filename, parameters, environ);
    } else {
        int status; // Parent
        waitpid(pid, &status, 0);
    }
}
```

一个简易shell示意代码

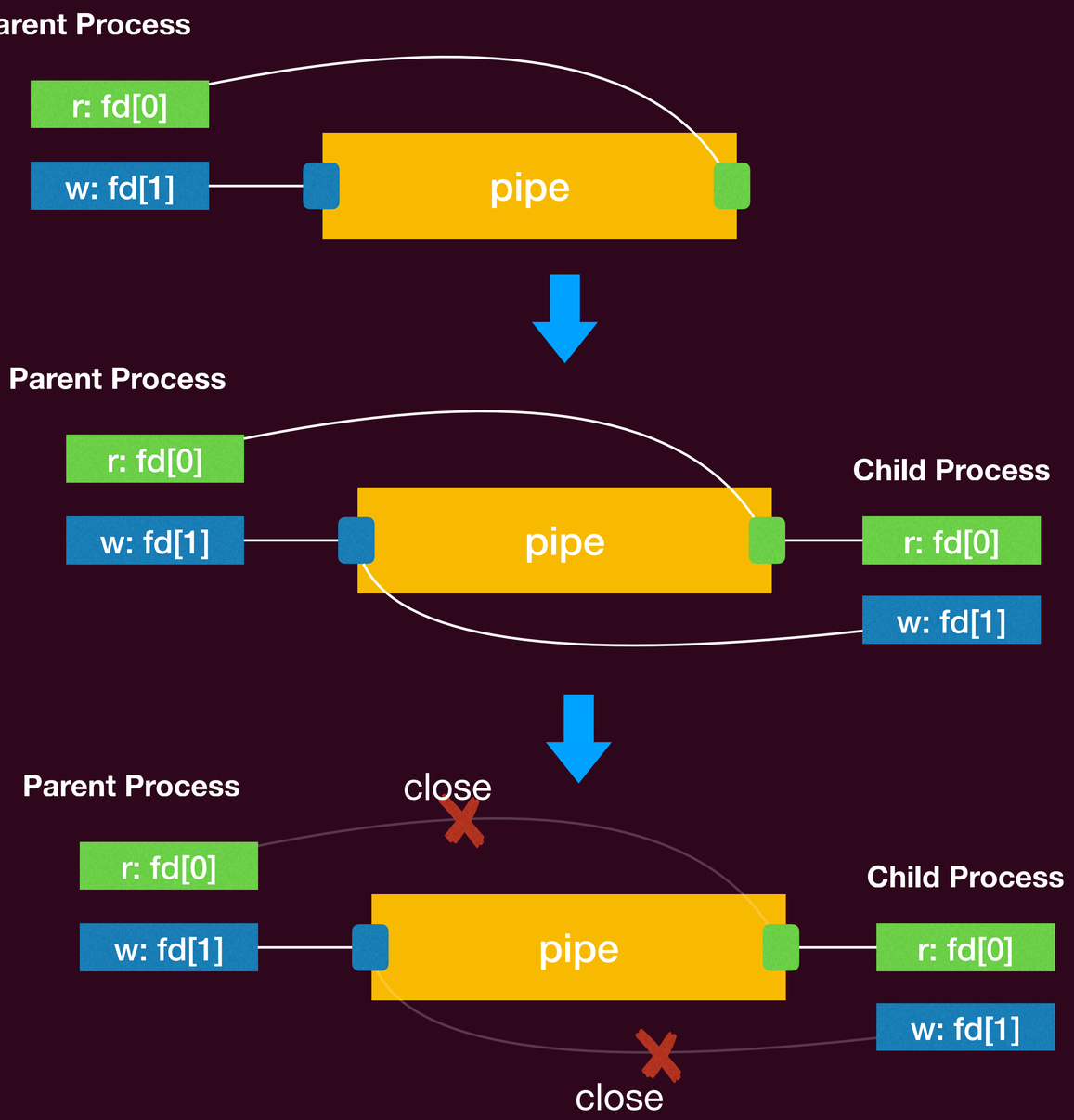
libc的execvp只需要filename和parameters

PCB中的文件描述符呢?

- execve不会改变PCB中的文件描述符，那些打开的文件描述符还会保持打开
- 这其实很方便
  - 比如shell里开启的进程打印字符都会在同样的终端(继承了stdout所指向的相同的tty)
  - 比如管道的实现

管道

- 管道其实是一个特殊的“文件”
  - 由读者/写者共享
    - 读口：支持read
    - 写口：支持write
- 匿名管道：int pipe(int pipefd[2]);
  - 返回两个文件描述符
  - 进程同时拥有读口和写口
    - fork时，父进程关闭读口，子进程关闭写口

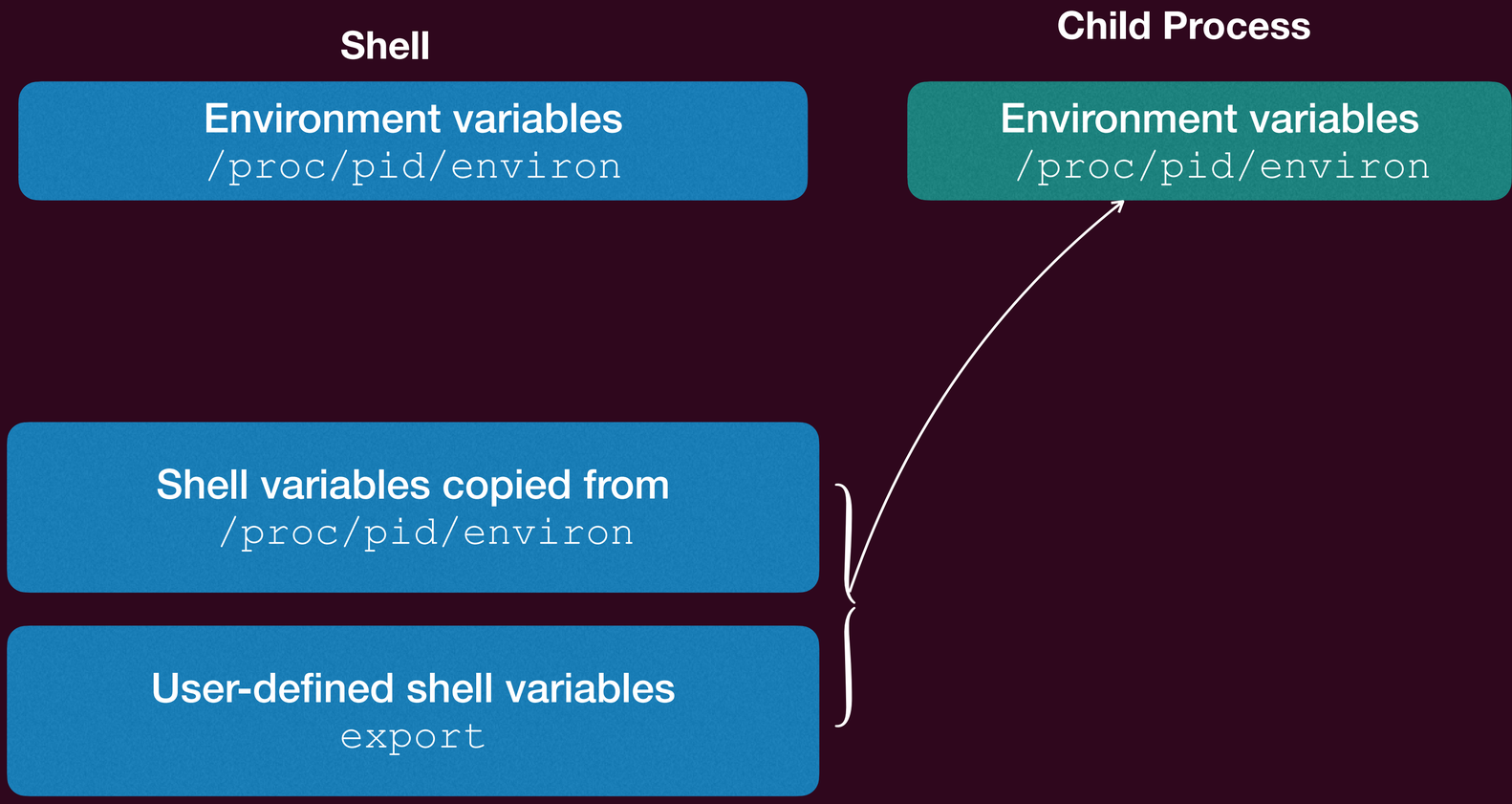


PCB中的文件描述符呢?

- 但如果是父进程打开了一个普通文件，在地址空间里有一个相应的file变量索引，但你的子进程重置了整个地址空间，因此那个文件描述符对应的file变量也没有了，此时你无法close这个文件了！
  - 这会造成资源的泄漏，当然这些资源（PCB）都会随着进程的终止而最终被回收，但在运行期间还是有一些资源的损耗
  - 可以在创建文件时增加一个选项 FD_CLOEXEC：close on exec

环境变量

- 应用程序执行的环境
  - 使用 env 命令查看
    - PATH: 可执行文件搜索路径
    - PWD: 当前路径
    - HOME: home 目录
    - LANG: 当前语言编码
    - …
  - export可以设置环境变量给子进程



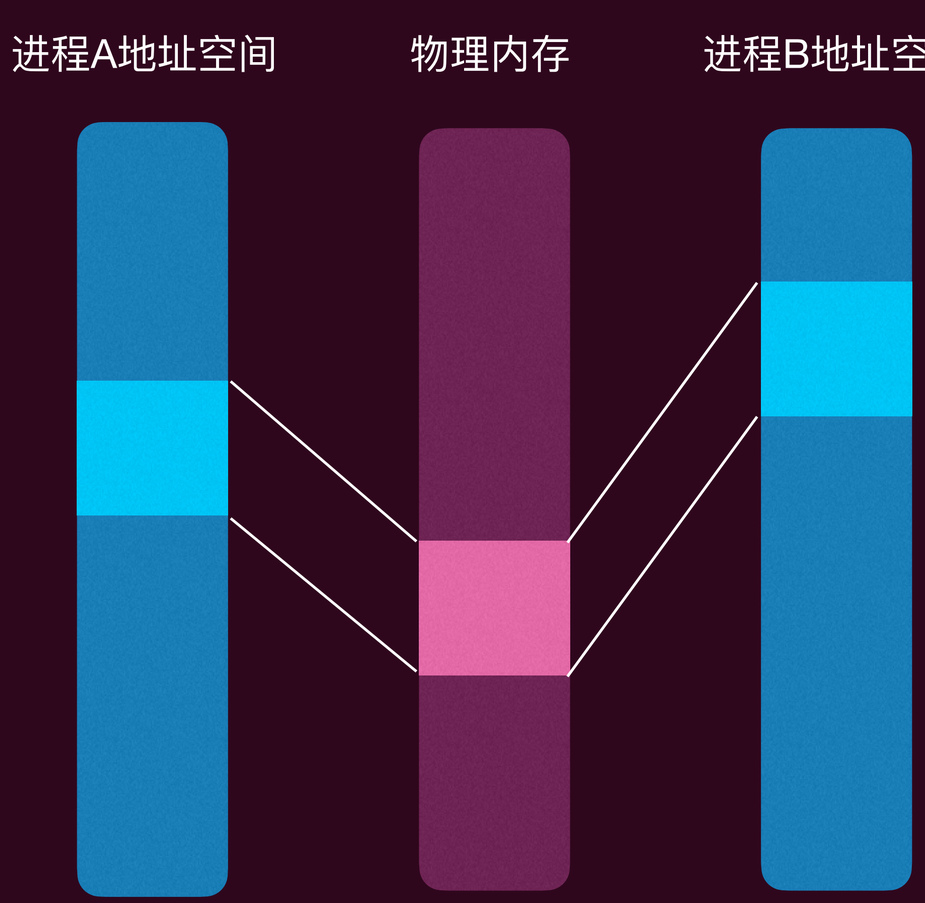
写时复制(Copy-On-Write, COW)

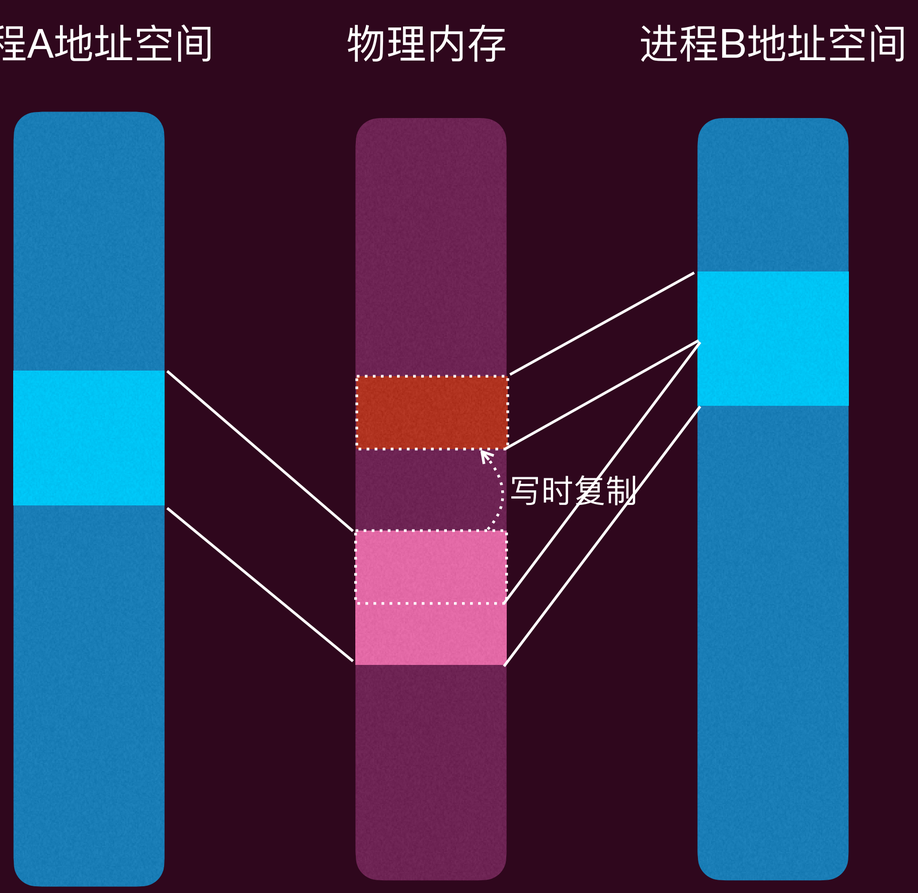
- fork后面往往跟着execve来加载子进程，那么fork的过程还必要吗?
- 事实上，早期的fork就是这么无脑的复制父进程的一切，但人们发现这个过程是低效的：
  - 有些内存是只读的(read-only)，比如代码段、共享代码库(libc)，这些没必要复制
  - 此外，立即执行execve会加载新的可执行文件，重置地址空间，因此，之前的内存拷贝完全没有意义

写时复制(Copy-On-Write, COW)

- 对于那些可以改变的内存，人们给出了一个聪明的设计：写时拷贝
  - 即两个进程共享同一份物理内存
  - 只有当一个进程尝试去写这个物理内存时才会真正在物理内存中复制一份副本用来给这个进程去写
  - 问题是，写自己的内存是一个“用户态”事件，内核又怎么知道呢?
    - 标记这个共享的内存为“只读”，一旦发生“写”操作会发生权限错误陷入内核，操作系统获知这是一个COW事件，复制内存!

写时复制(Copy-On-Write, COW)





posix_spawn接口

- posix定义了创建进程的API： posix_spawn： 结合了（fork+execve）
  - pid: 返回的进程号
  - path: 可执行文件
  - file_actions: open, close, dup
  - attrp: 信号、进程组等信息
  - argv, envp: 同 execve

```c
#include <spawn.h>

int posix_spawn(pid_t *pid, char *path,
                posix_spawn_file_actions_t *file_actions,
                posix_spawnattr_t *attrp,
                char * argv[], char * envp[]);
```

exit

电影：复仇者联盟

进程退出机制

- 除了创建进程外，进程也需要能够终止：清理其所占内存空间（包括代码区、堆、栈），这个过程需要系统调用来做
  - 因为进程这个“实体”就是操作系统抽象出来的，没有操作系统，进程就是一连串的指令流，没有终止的概念（CPU只会不断的进行取指-执行的循环）
- Linux中，进程一般有5种退出机制
  - 正常退出： 从main函数返回，调用库函数exit，调用_exit
  - 异常退出：调用abort，由信号终止

exit()

- `void exit(int status)` 是C标准库函数，也是最常用的进程退出函数:
  - 调用`atexit()`注册的函数；使得我们可以指定在程序终止时执行自己的清理动作。(`atexit()`最多可以注册32个函数，调用顺序与注册顺序相反)
  - 关闭所有打开的流(stdio)，这将导致写所有被缓冲的输出
  - 移除所有的临时文件
  - 最后调用`_exit()`函数终止进程

```c
#include <stdlib.h>
void exit(int status);
```

The `exit()` function never returns

return from `main()`

- 从main函数返回是最常见的终止方式
- main函数的返回值和调用exit()的传入参数int status是同样的语义
  - 0是函数是符合预期终止，非0是函数出现了错误终止

```c
#include <stdlib.h>
int main(int argc, char **argv) {
  /* ... */
  if (/* Something really bad happened */) {
    return EXIT_FAILURE;
  }
  /* ... */
  return EXIT_SUCCESS;
}
```

很多编译器实现main的返回是类似如下这种方式

```c
void _start(void) {
  /* ... */
  exit(main(argc, argv));
}
```

_exit()

- 所有属于这个进程的文件描述符都会被关闭
- 所有该进程的子进程都会被init进程接管
- 向该进程的父进程发送SIGCHLD信号，通知改父进程其已经终止
- 注意：_exit()只会终止当前的线程，但libc做了一层wrapper，其实真实调用exit_group()，关闭所有线程

直接调用syscall(SYS_exit, 0);可以绕过这个wrapper

```c
#include <unistd.h>
void _exit(int status);
```

异常退出

- abort()系统调用会导致系统异常终止
  - atexit注册的函数不会调用
  - io流不会关闭
  - 其行为就是产生一个SIGABRT信号发送给调用abort()的进程，然后改进程就会异常终止
    - 不会被ignore掉

```
#include <stdlib.h>
void abort(void);
```

信号（Signal）

- Linux操作系统提供了一种可以通知进程发生了某个事件的机制：信号（Signal），注：不要和并发的singal原语混淆，也不要和“信号量”（Semaphore）混淆
- 其本质上是对中断的模拟, 命令`kill -l` 可以查看有多少信号， man 7 signal查看信号的具体信息
  - 发生某个中断事件之后，用户程序也想获知并处理（而不只是在内核态由操作系统透明的处理）

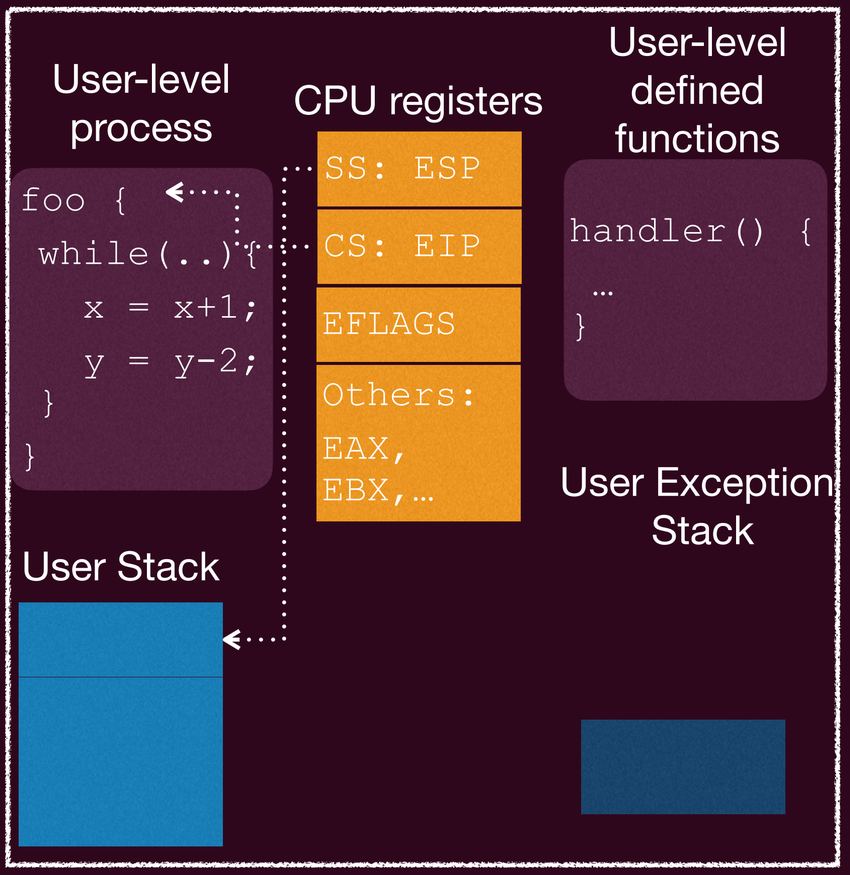
Interrupts/Exceptions

- Hardware-defined Interrupts & exceptions
- Interrupt vector for handlers (kernel)
- Interrupt stack (kernel)
- Interrupt masking (kernel)

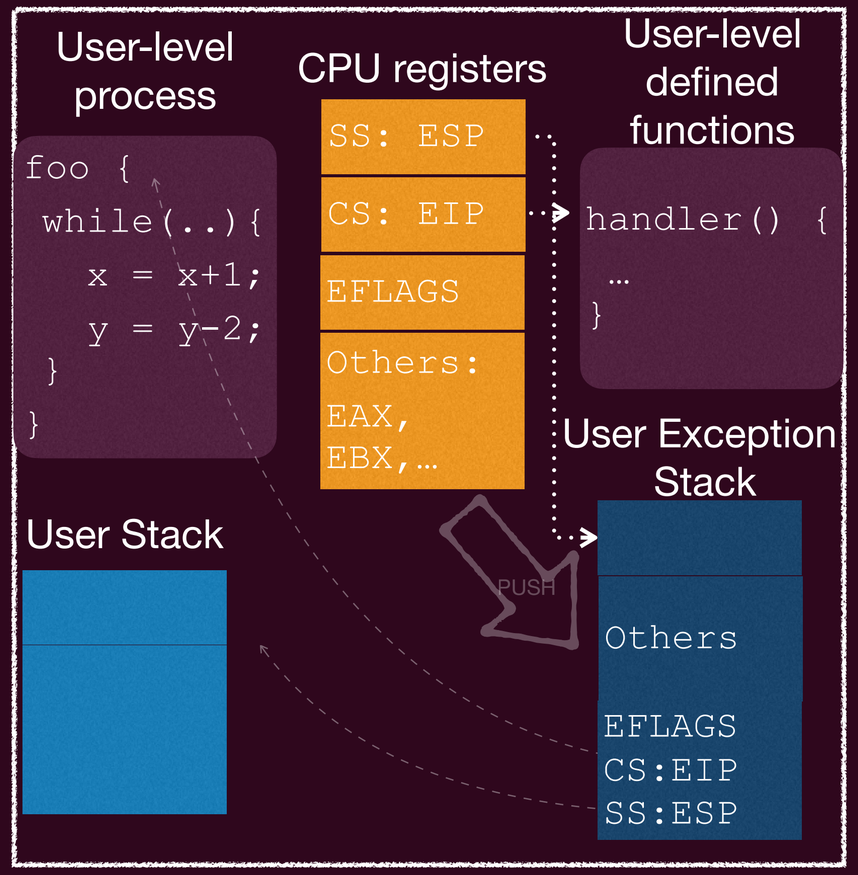
Signals

- Kernel-defined signals
- Handlers (user)
- Signal stack (user)
- Signal masking (user)

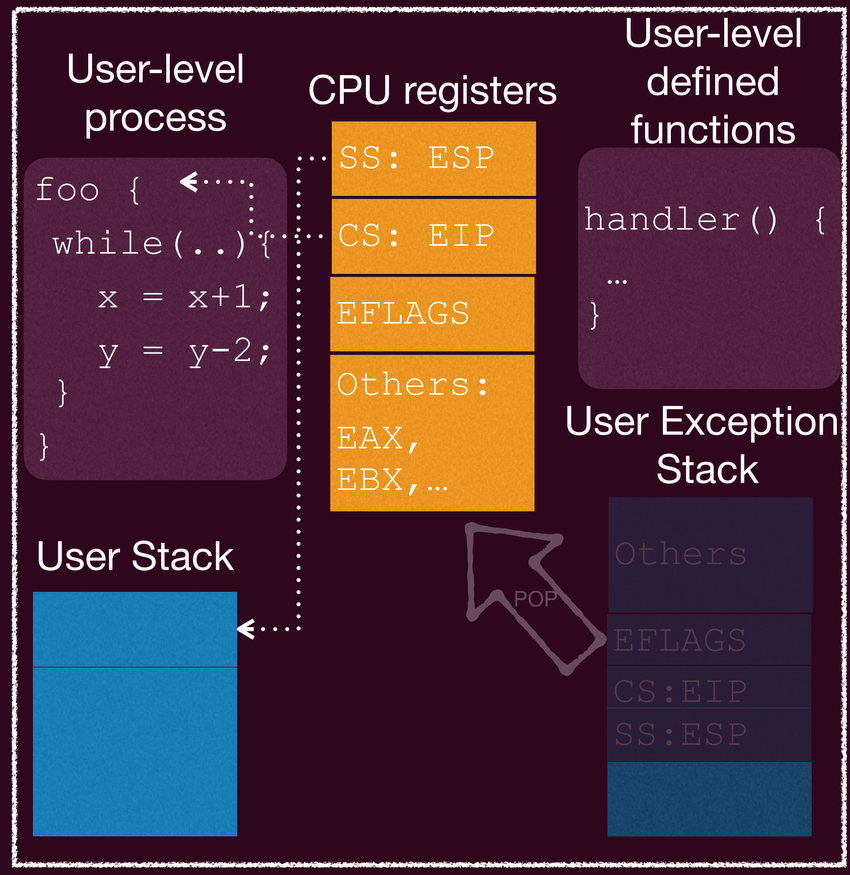
信号（Signal）



Before exception event



After received an signal



Signal handling ends and get back to previous instructions

Note: default signal handler normally directly terminates the process

信号（Signal）

- signal函数可以注册信号处理函数，只要传入相应的信号和函数名即可
  - handler设为SIG_IGN时表示忽略这个信号，比如signal(SIGCHLD,SIG_IGN) 就表示忽略掉子进程的终止信号，此时子进程结束会直接被内核完全清除（而不必先变为僵尸进程，然后再被回收）
  - handler设为SIG_DFL时表示采用linux默认的处理函数

```
#include <signal.h>
signal(SIGNAL_NUMBER, handler);
```

信号（Signal）

- 有了信号机制，可以完成很多异步的操作：
  - 比如signal(SIGCHLD， handler)，可以在handler里进行wait，而不是在main函数里wait/waitpid从而阻塞父进程
  - 有比如signal(SIGIO, handler)可以不用等待I/O完成，可以先做其他事情，如果文件描述符所指向的数据传输完成，会产生SIGIO的事件，就可以通过回调函数handler来处理
- 信号也是一种进程间通信（Inter-Process Communication, IPC）的机制之一
  - 此外还有消息传递、共享内存、管道等

总结

- 进程的概念、生命周期
- 三个重要的系统调用fork、exec、exit

阅读材料

- [OSTEP] 第3，4，5章
- [现代操作系统]第5章