中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

进程与线程I

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

再回来看Hello World

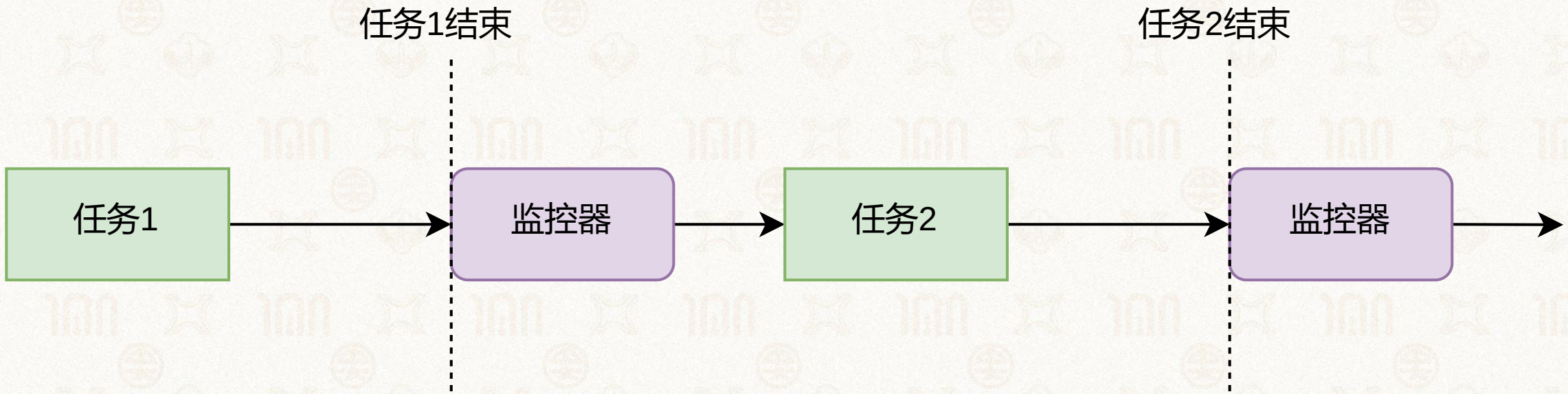
```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

```
bash$ gcc hello.c -o hello

# 运行一个hello world程序
bash$./hello
Hello World!

# 同时启动两个hello world程序
bash$ ./hello & ./hello
[1] 144
Hello World!
Hello World!
[1]+ Done        ./hello
```

运行多个hello时，操作系统怎么抽象与管理？

进程的诞生：从单任务到多任务

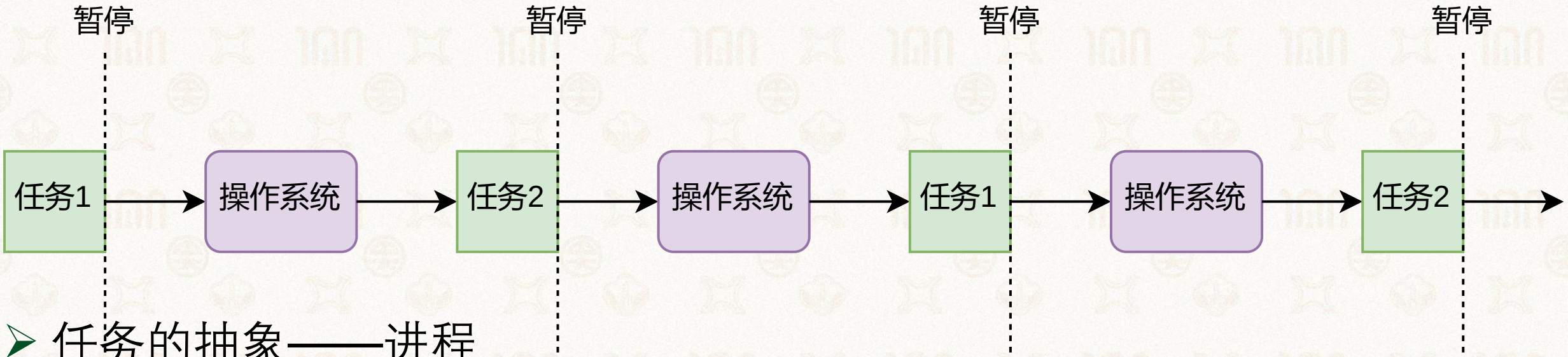
- 早期的计算机一次只能执行一个任务



- 计算机的发展趋势
  - 计算机程序种类越来越多（文本编辑、科学计算、web服务……）
  - 外部设备种类越来越多（硬盘、显示器、网络），造成程序等待

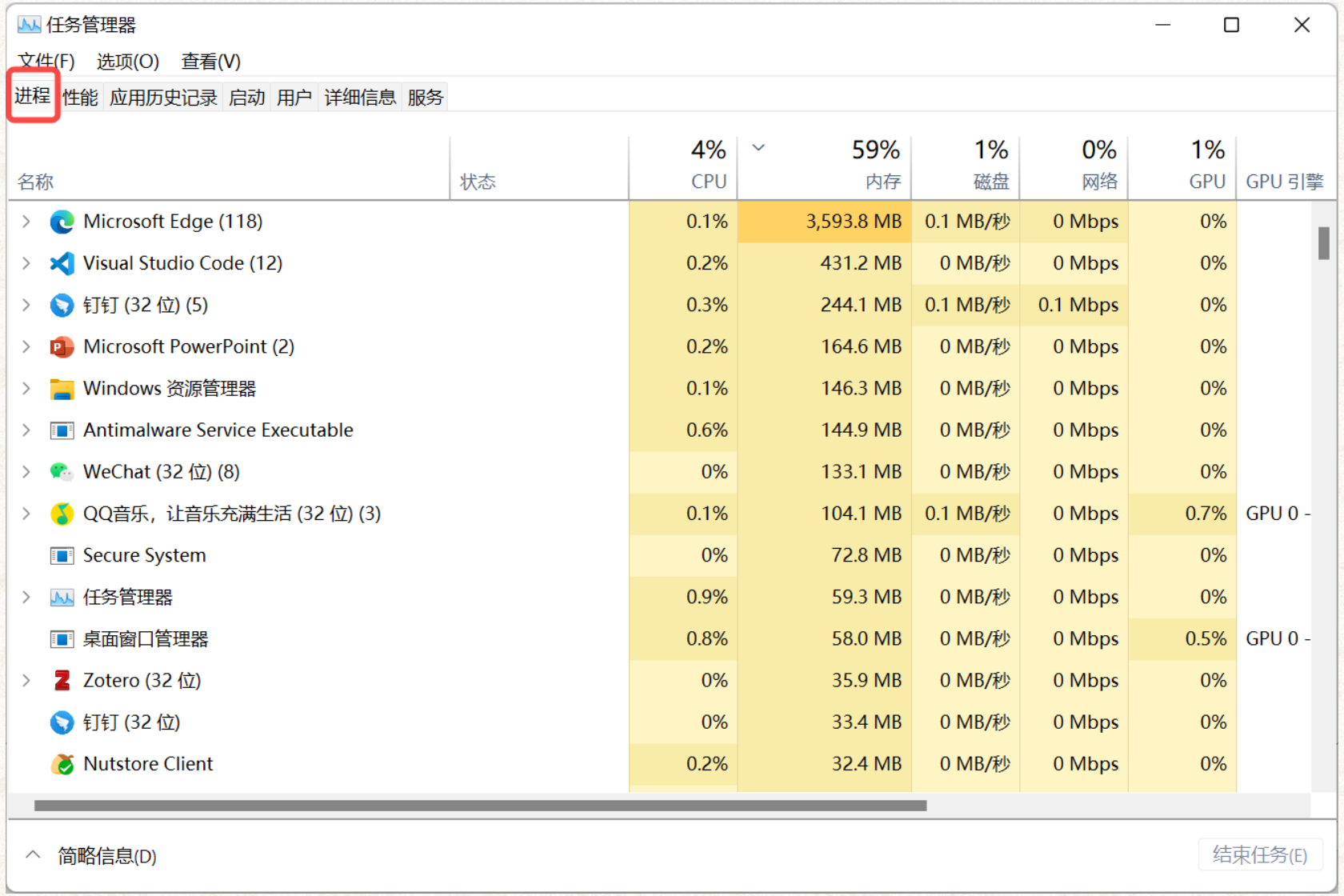
进程的诞生：从单任务到多任务

- 思路：提出分时（time-sharing）操作系统
  - 多任务并行：
    - 当一个任务需要等待时，切换到其他任务



- 任务的抽象——进程
  - 进程的执行状态不断更新
  - 不断切换处理器上运行的进程（上下文切换）
  - 操作系统需要对进程进行调度

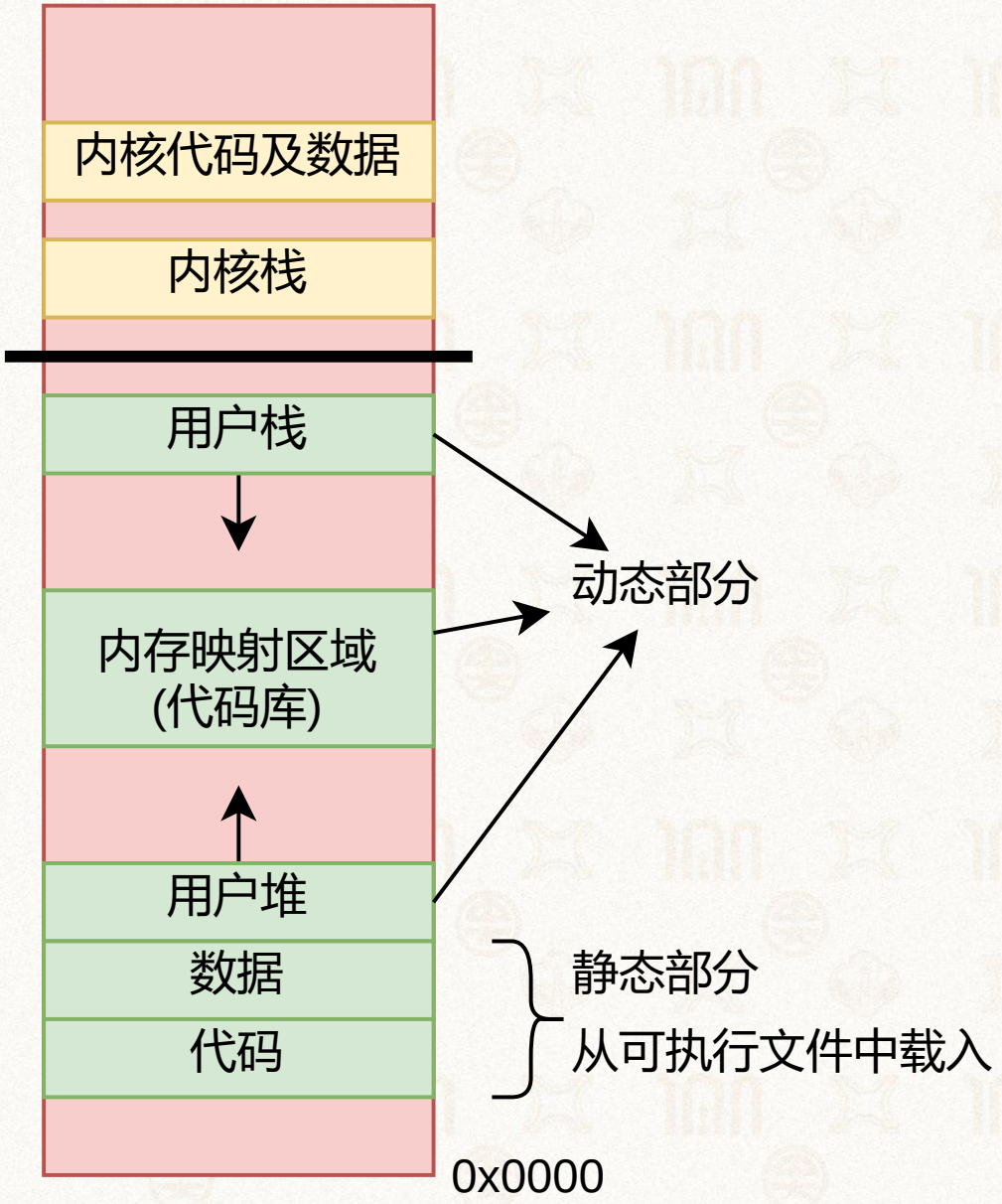
进程：运行中的程序



Windows系统中按 Ctrl+Alt+Delete 可以调出任务管理器

进程：运行中的程序

- 进程是计算机程序运行时的抽象
  - 静态部分：程序运行需要的代码和数据
  - 动态部分：程序运行期间的状态（程序计数器、堆、栈……）
- 进程具有独立的虚拟地址空间
  - 每个进程都具有“独占全部内存”的假象
  - 内核中同样包含内核栈和内核代码、数据
- 示例：查看某进程的内存空间布局
  - 命令：cat /proc/PID/maps
  - 查看所有进程信息：ps -aux

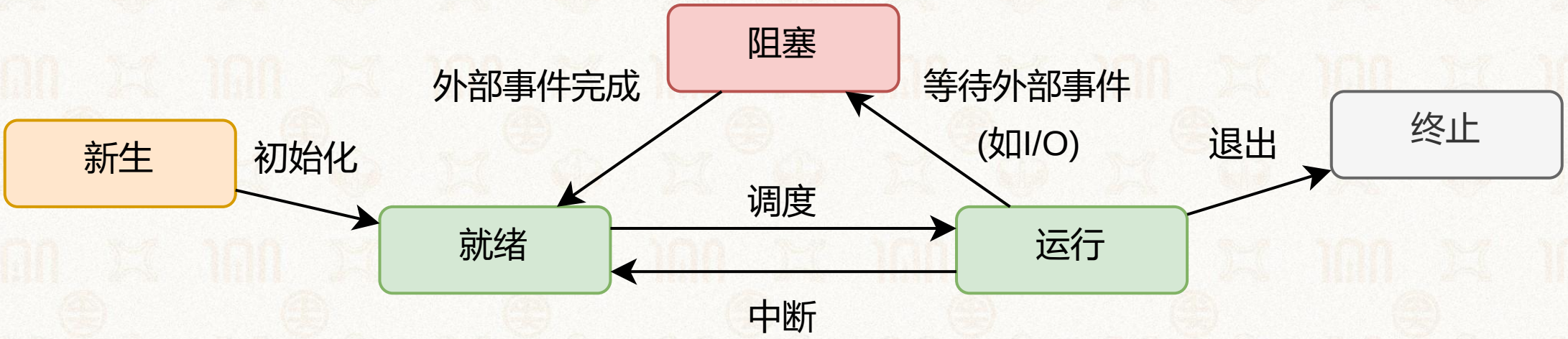


大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

进程的状态

- 进程至少应当拥有以下五种状态:
  - 新生状态（new）：进程刚被创建
  - 就绪状态（ready）：进程可以运行，但没有被调度
  - 运行状态（running）：进程正在处理器上运行
  - 终止状态（terminated）：进程完成了执行
  - 阻塞状态（blocked）：进程进入等待状态，短时间不再运行
- 进程会不断进行状态切换
  - 被调度器调度，开始执行： 准备->运行

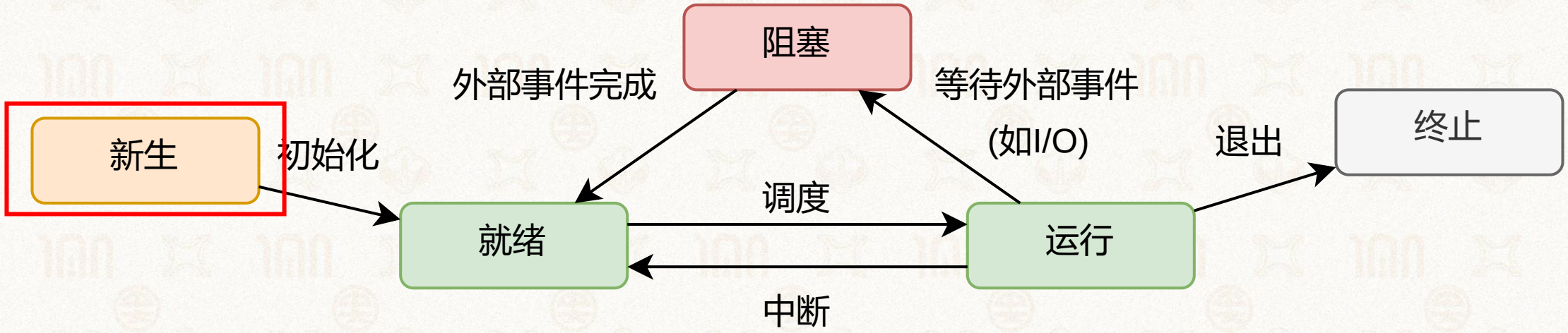


进程的状态

- 新生：
  - 执行./hello-name
  - 正在创建新进程运行该程序

```
yxsu@Dell-T6401:~/os/process$ gcc -o hello-name hello-name.c
yxsu@Dell-T6401:~/os/process$ ls
hello-name  hello-name.c
yxsu@Dell-T6401:~/os/process$ ./hello-name
```

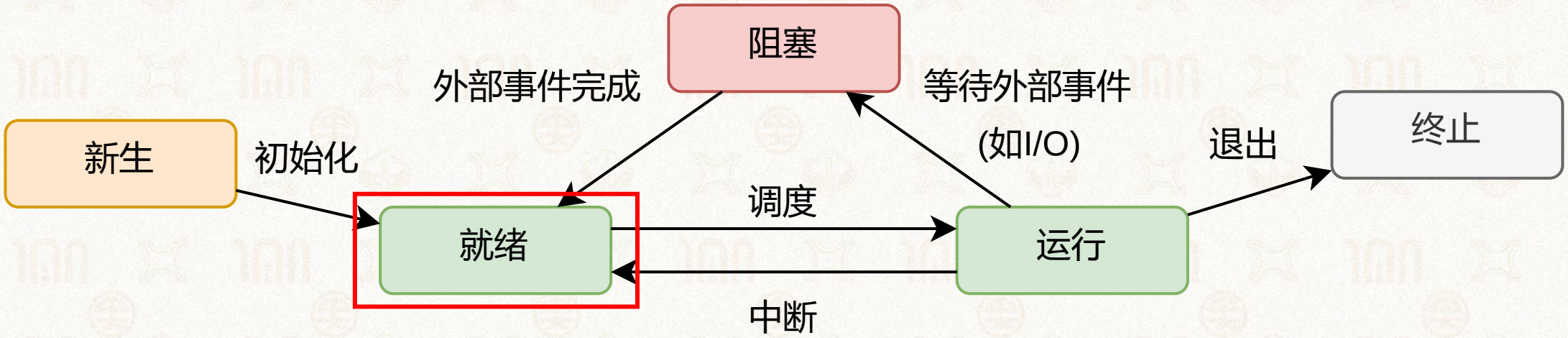
```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



进程的状态

- 就绪：
  - 对相关数据结构进行初始化
  - 交给调度器(目前还抽象，不知所云)

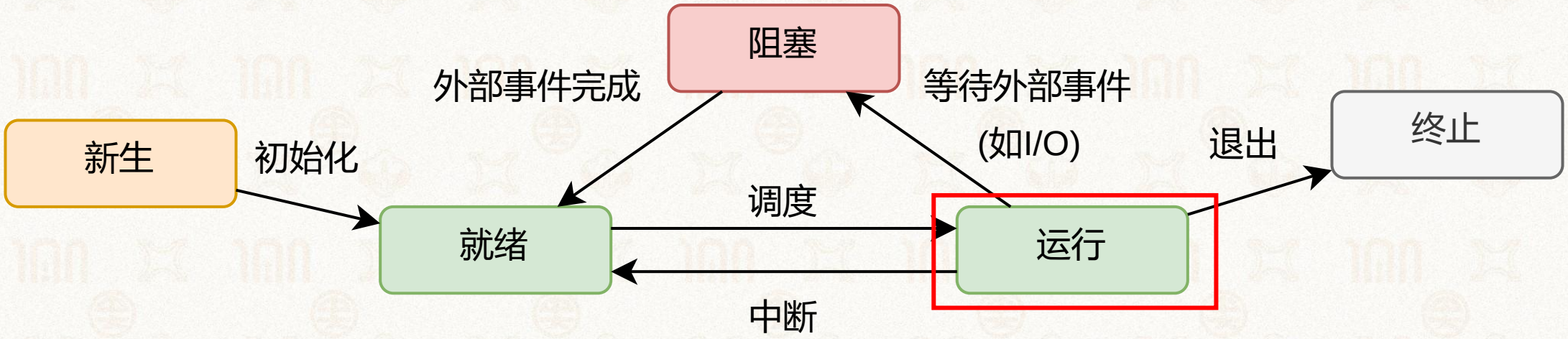
```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



进程的状态

- 运行：
  - 从main函数开始执行

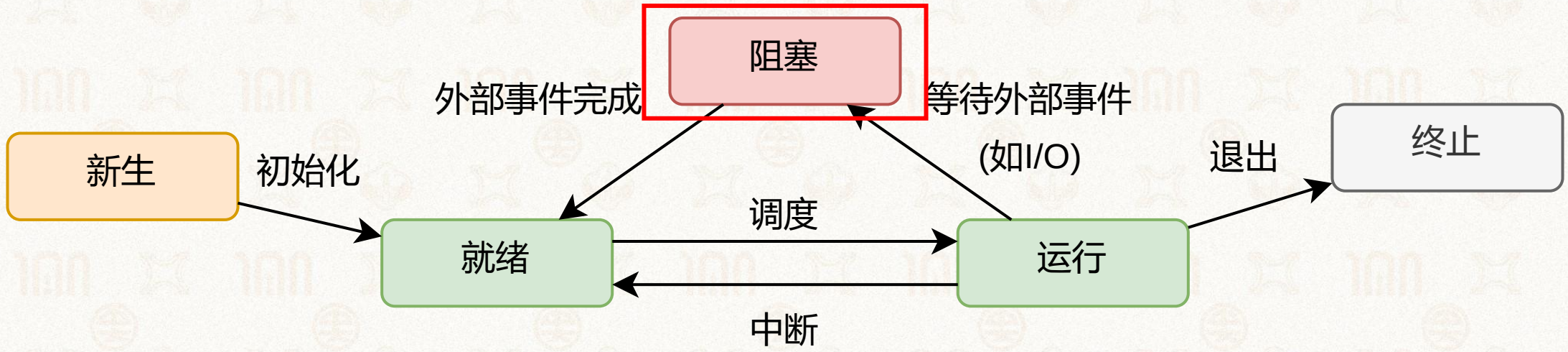
```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



进程的状态

- 阻塞：
  - 需要接受用户输入
  - 此时进程不处于运行状态
  - 不在运行队列中

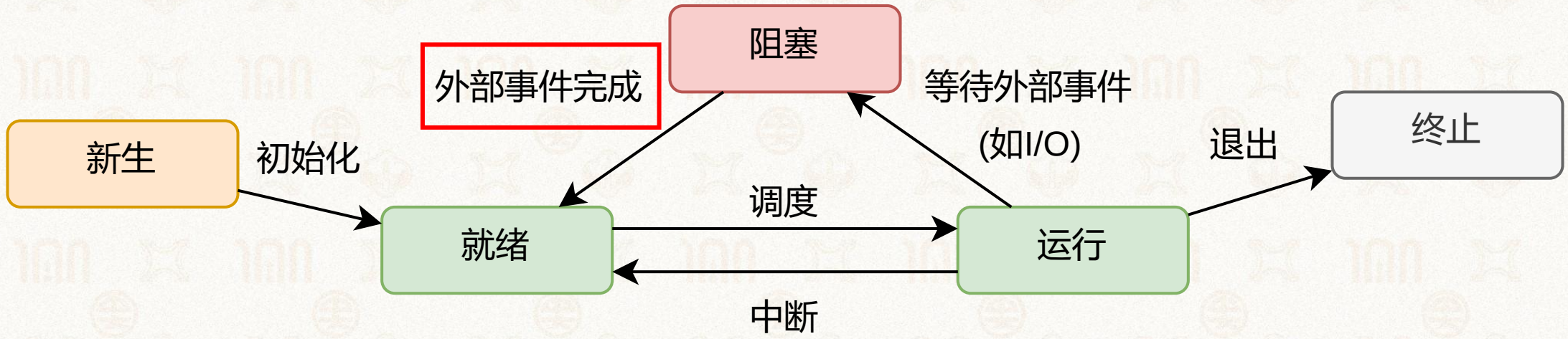
```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



进程的状态

- 就绪、运行状态：
  - 用户完成输入并回车

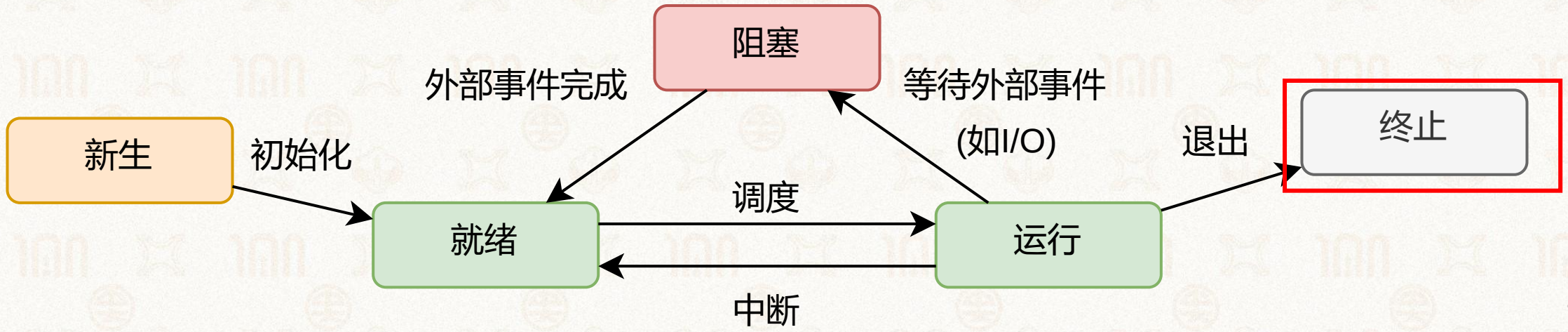
```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



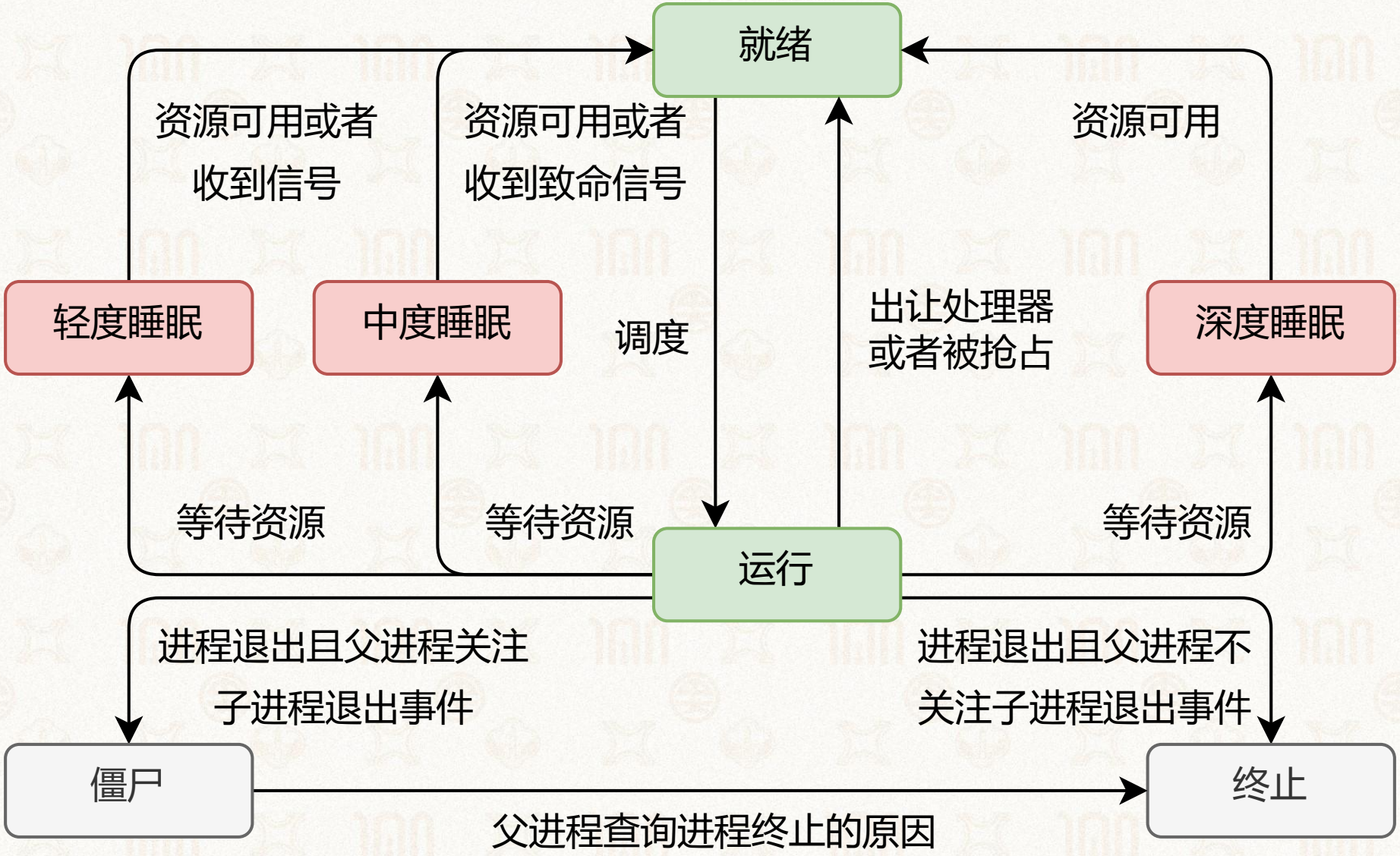
进程的状态

- 终止状态：
  - 进程执行完毕，回到内核中
  - 内核回收进程相关资源

```c
#include <stdio.h>
#define LEN 10

int main(int argc, char* argv[]) {
  char name[LEN] = {0};
  fgets(name, LEN, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```



Linux系统中进程的状态



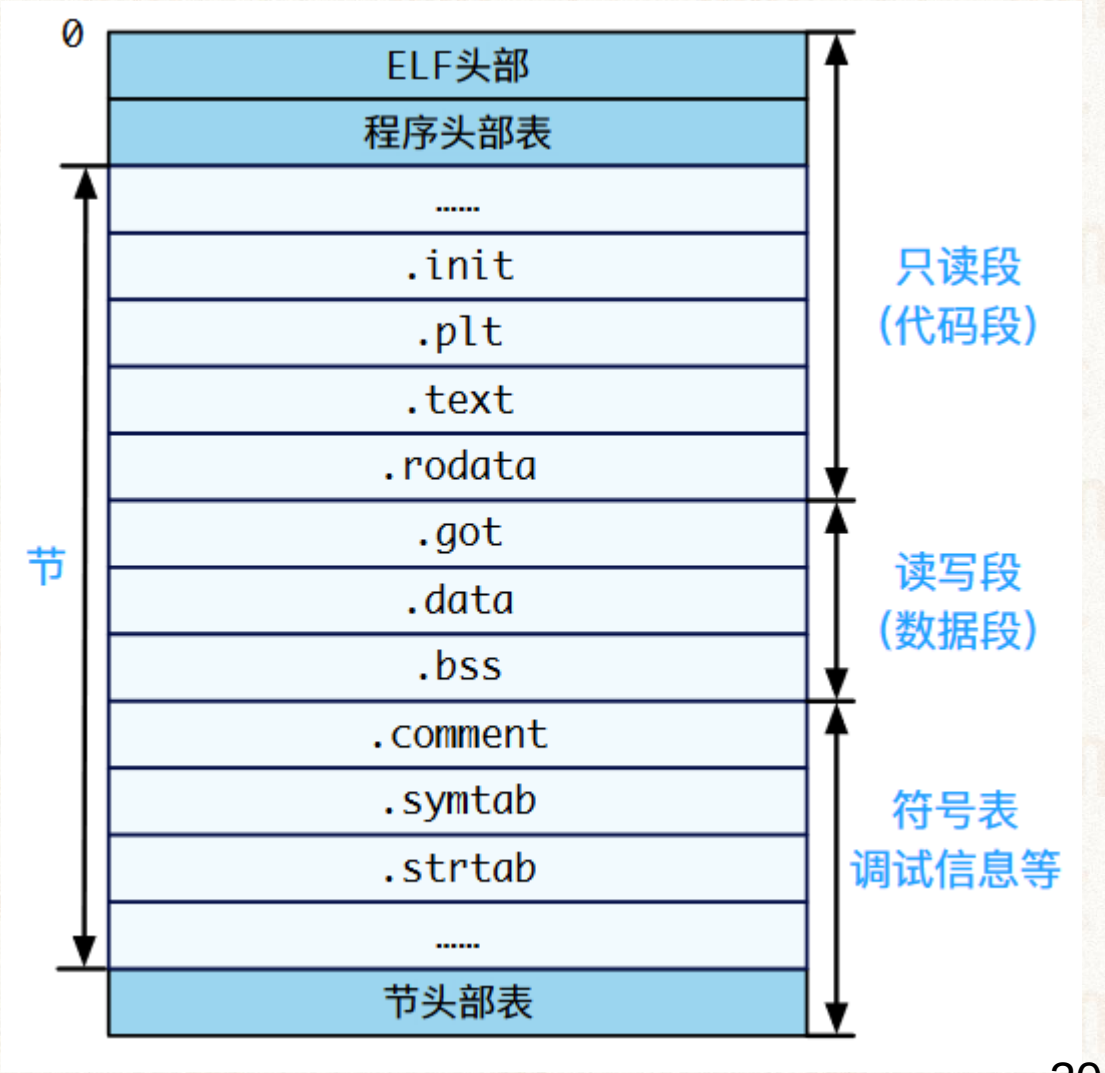
https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L83

大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

应用程序的原始形态：可执行文件的 ELF 格式

- ELF可执行可链接文件
- 常见于：
  - Linux/Android系统的可执行文件
  - 共享库(.so / .a)
  - 目标文件(.o)
- 组成
  - ELF头部(ELF header)
  - 多个程序段(program section)
    - 每个程序段都是一个连续的二进制块
    - （硬件或软件）加载器将它们作为代码或数据加载到指定地址的内存中并开始执行

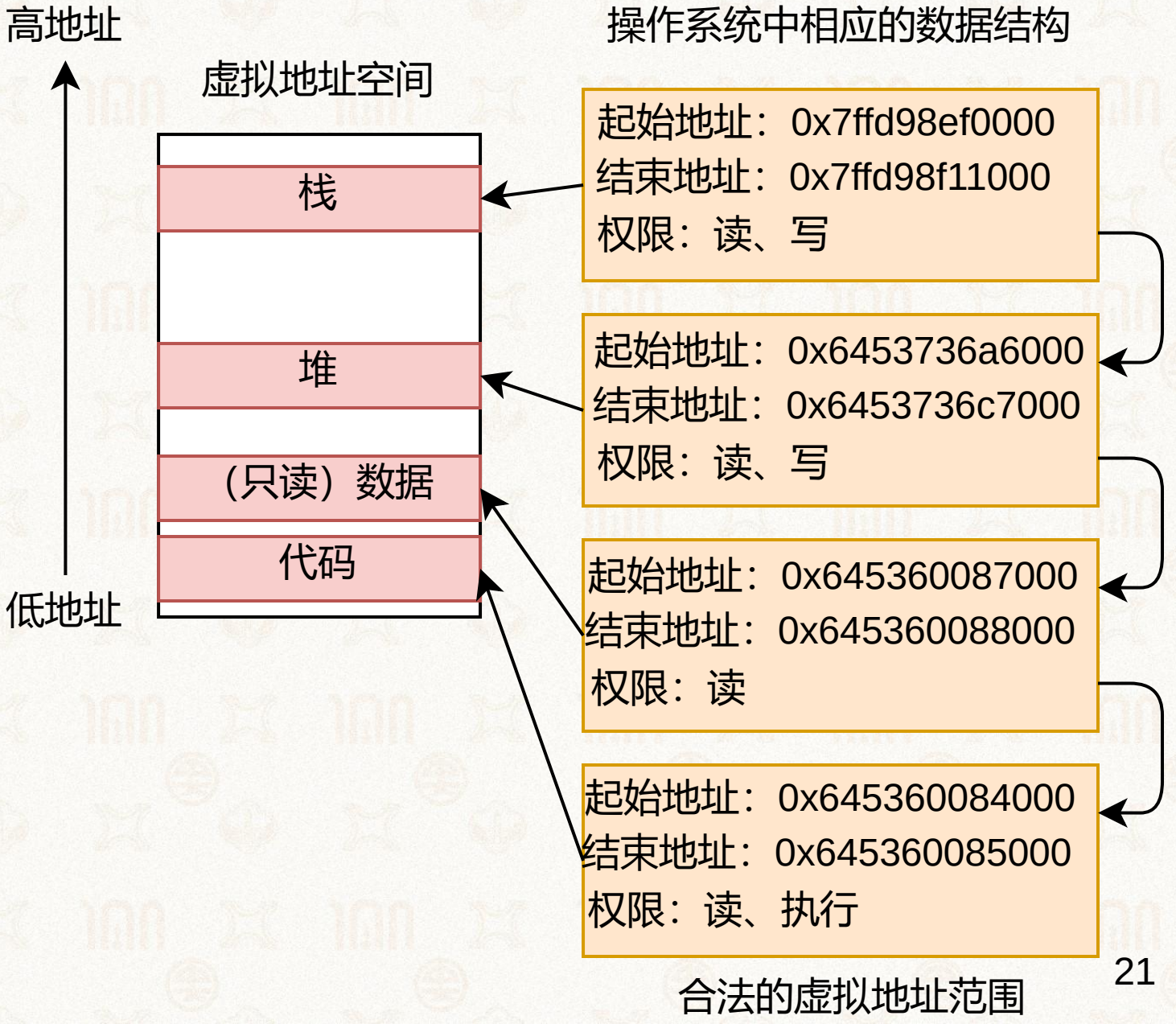


进程的相关数据结构：Process Control Block

- 进程创建后，将ELF格式的内容映射到内存中

- 需要管理映射的内容

```
struct process_v1 {
    // 上下文
    struct context *ctx;
    // 虚拟地址空间
    //（包含页表基地址）
    struct vmspace *vmspace;
    // 内核栈
    void *stack;
};
```

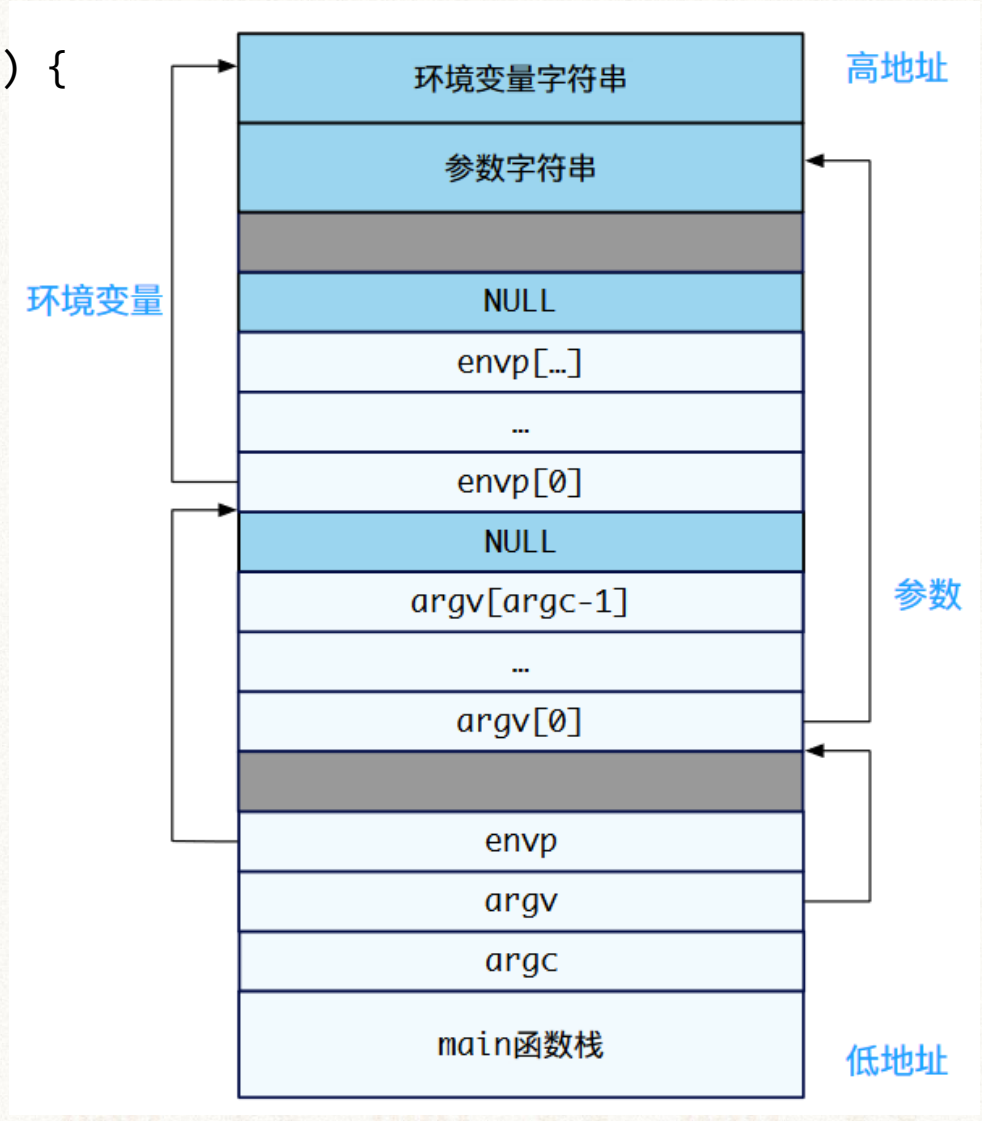


创建进程

```
int process_create(char *path, char *argv[], char *envp[]) {
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 虚拟内存初始化：初始化虚拟地址空间及页表基地址
    init_vmspace(new_proc->vmspace);
    new_proc->vmspace->pgdir = alloc_new_page();
    // 内核栈初始化
    init_kern_stack(new_proc->stack);
    // 加载可执行文件
    struct file *file = load_elf_file(path);
    for loadable_seg in file.segs
        vmspace_map(new_proc->vmspace, loadable_seg);
    // 准备运行环境：创建并映射用户栈
    void *stack = alloc_stack(STACKSIZE);
    vmspace_map(cur_proc->vmspace, stack);
    // 准备运行环境：将参数和环境变量放到栈上
    prepare_env(stack, argv, envp);
    // 上下文初始化
    init_process_ctx(new_proc->ctx);
    // 返回
    ...
}
```

创建进程

```
int process_create(char *path, char *argv[], char *envp[]) {
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 虚拟内存初始化：初始化虚拟地址空间及页表基地址
    init_vmspace(new_proc->vmspace);
    new_proc->vmspace->pgdir = alloc_new_page();
    // 内核栈初始化
    init_kern_stack(new_proc->stack);
    // 加载可执行文件
    struct file *file = load_elf_file(path);
    for loadable_seg in file.segs
        vmspace_map(new_proc->vmspace, loadable_seg);
    // 准备运行环境：创建并映射用户栈
    void *stack = alloc_stack(STACKSIZE);
    vmspace_map(cur_proc->vmspace, stack);
    // 准备运行环境：将参数和环境变量放到栈上
    prepare_env(stack, argv, envp);
    // 上下文初始化
    init_process_ctx(new_proc->ctx);
    // 返回
    ...
}
```



进程的相关数据结构：Process Control Block

- 存放进程相关的各种信息
  - 进程的标识符、内存、打开的文件……
  - 进程在切换时的状态（上下文context）
- 在Linux中进程叫task

简版：

```
struct process_v2 {
        // 上下文
        struct context *ctx;
        // 虚拟地址空间（包含页表基地址）
        struct vmspace *vmspace;
        // 内核栈
        void *stack;
        // 进程标识符
        int pid;
    };
```

Linux 版

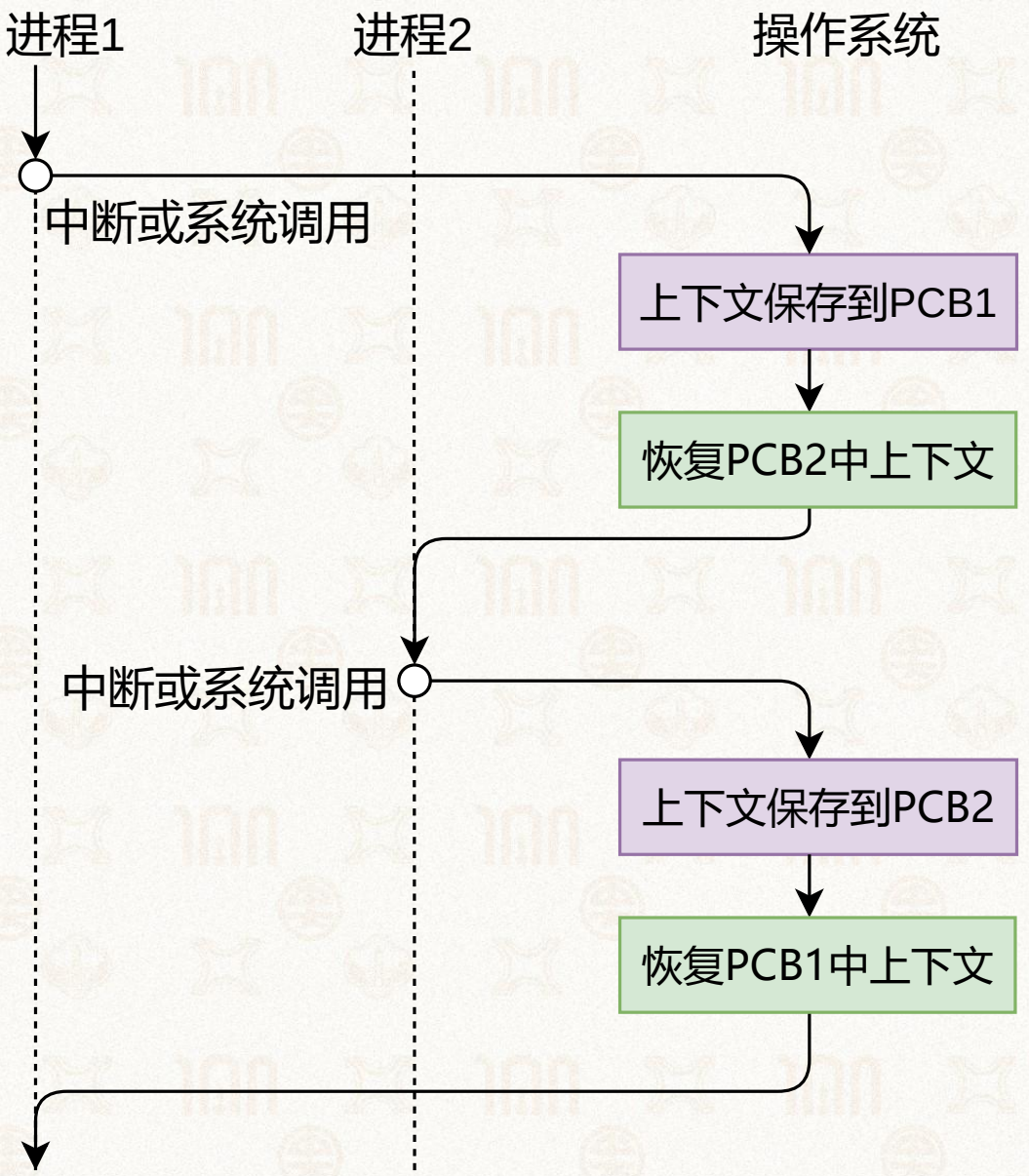
```
struct task_struct {
    unsigned int            __state;
    void                    *stack;
    struct list_head        tasks;
    struct mm_struct        *mm;
    struct mm_struct        *active_mm;
    pid_t                   pid;
    pid_t                   tgid;
    /* Real parent process: */
    struct task_struct __rcu    *real_parent;
    /* Recipient of SIGCHLD, wait4() reports: */
    struct task_struct __rcu    *parent;
    struct list_head        children;
    struct list_head        sibling;
    struct task_struct      *group_leader;
    u64             utime;
    u64             stime;
    /* Filesystem information: */
    struct fs_struct        *fs;
    /* Open file information: */
    struct files_struct     *files;
    /* CPU-specific state of this task: */
    struct thread_struct        thread;
};
```

https://elixir.bootlin.com/linux/v5.16.16/source/include/linux/sched.h#L723

进程的上下文切换

- 进程通过中断或系统调用进入内核

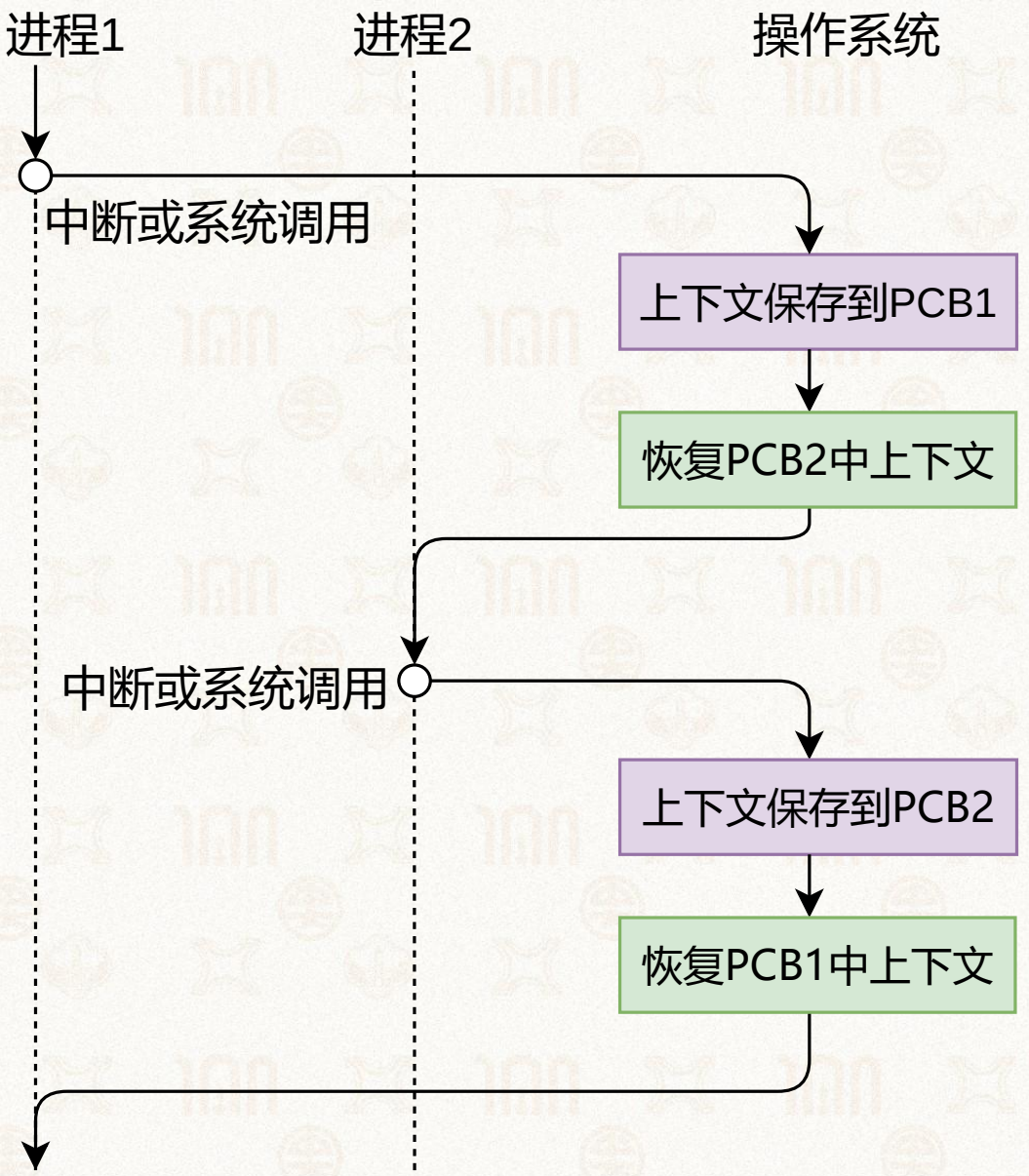
- 上下文保存在对应的PCB中
  - 被调度时，从PCB中取出上下文并恢复



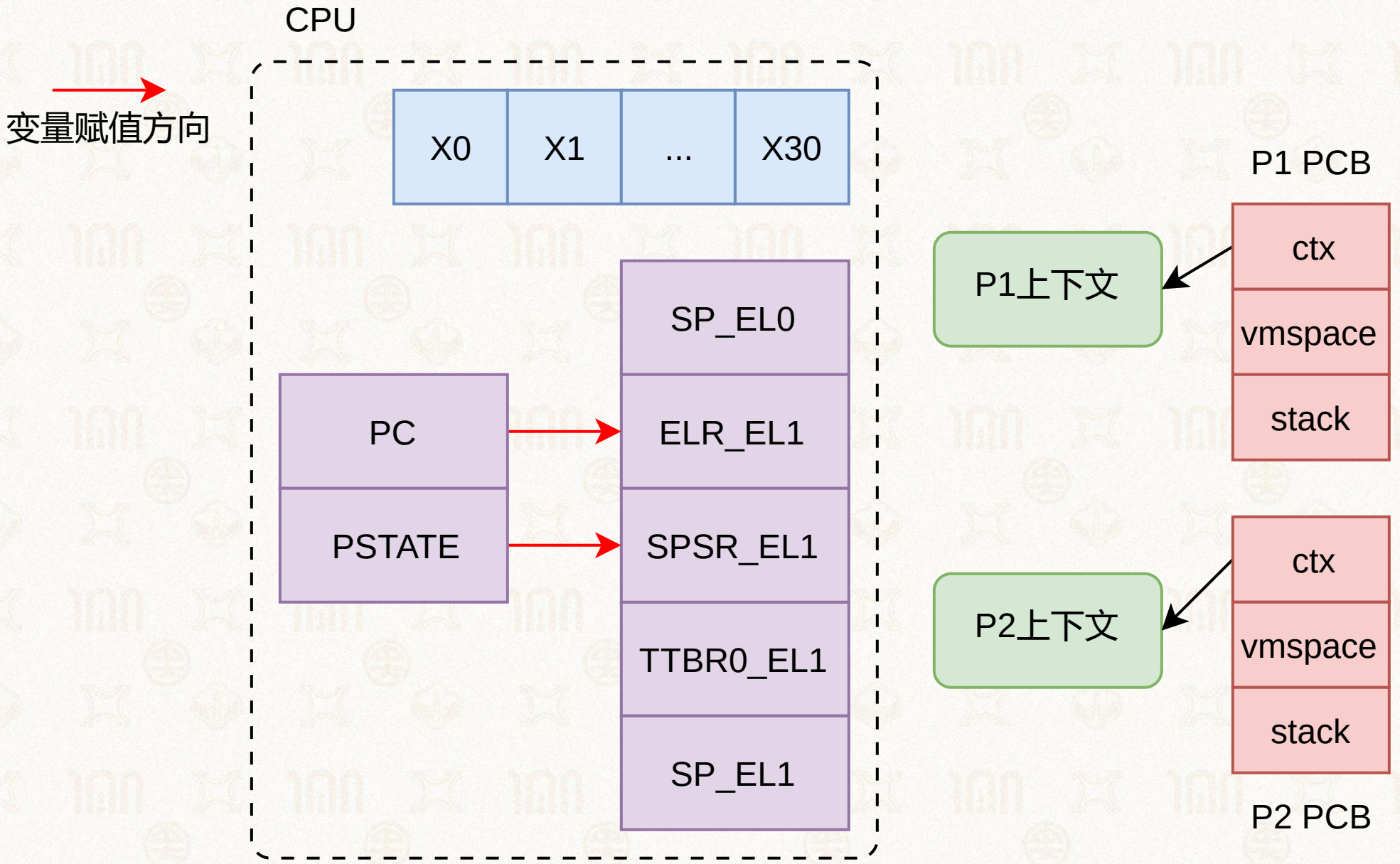
进程的上下文切换

- 进程通过中断或系统调用进入内核

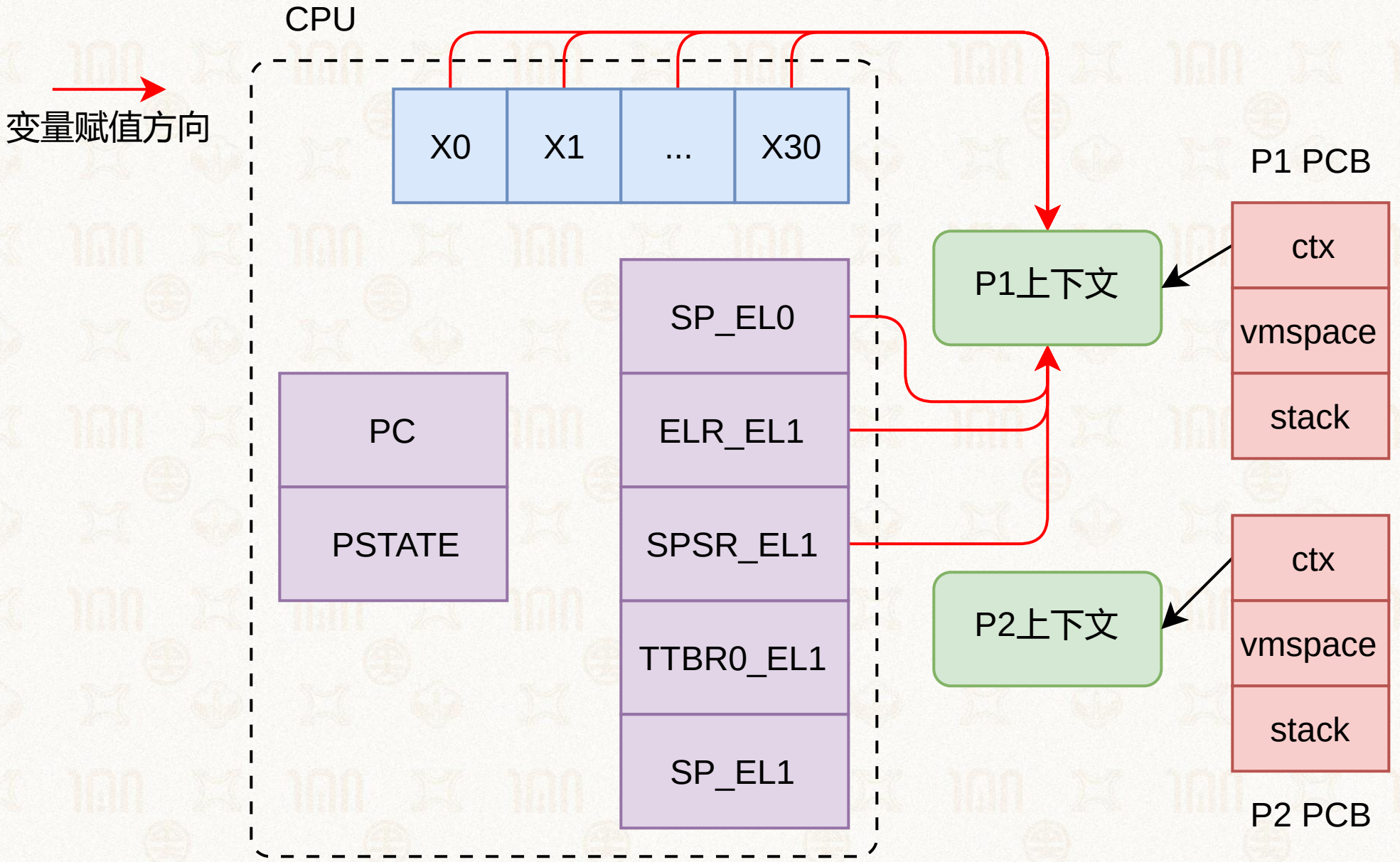
```
// 进程处理器上下文内部包含的内容
struct context {
    // 通用寄存器
    u64 x0, x1, ..., x30;
    // 特殊寄存器
    u64 sp_el0;
    // 系统寄存器
    u64 elr_el1, spsr_el1;
};
```



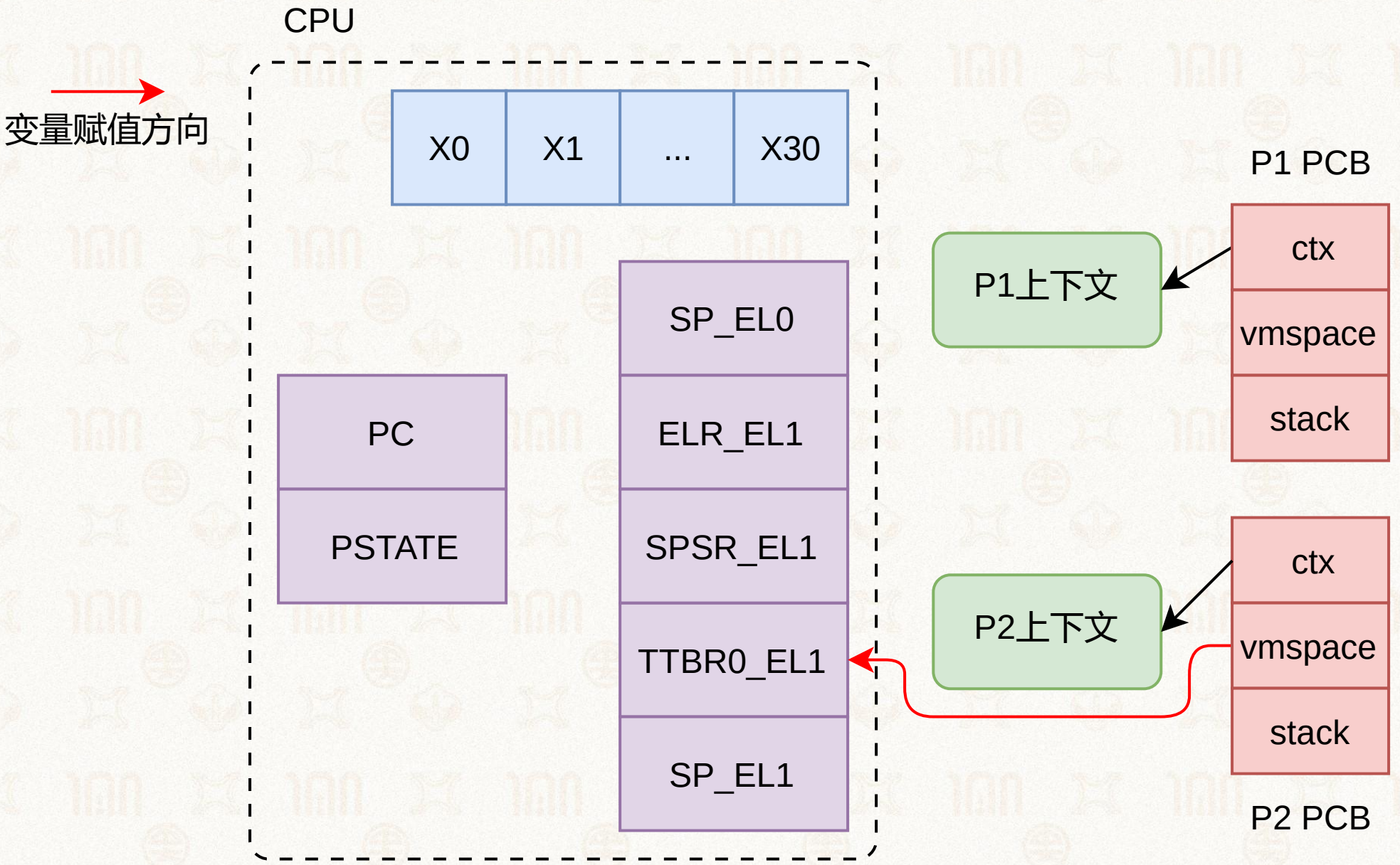
上下文切换：P1进入内核态



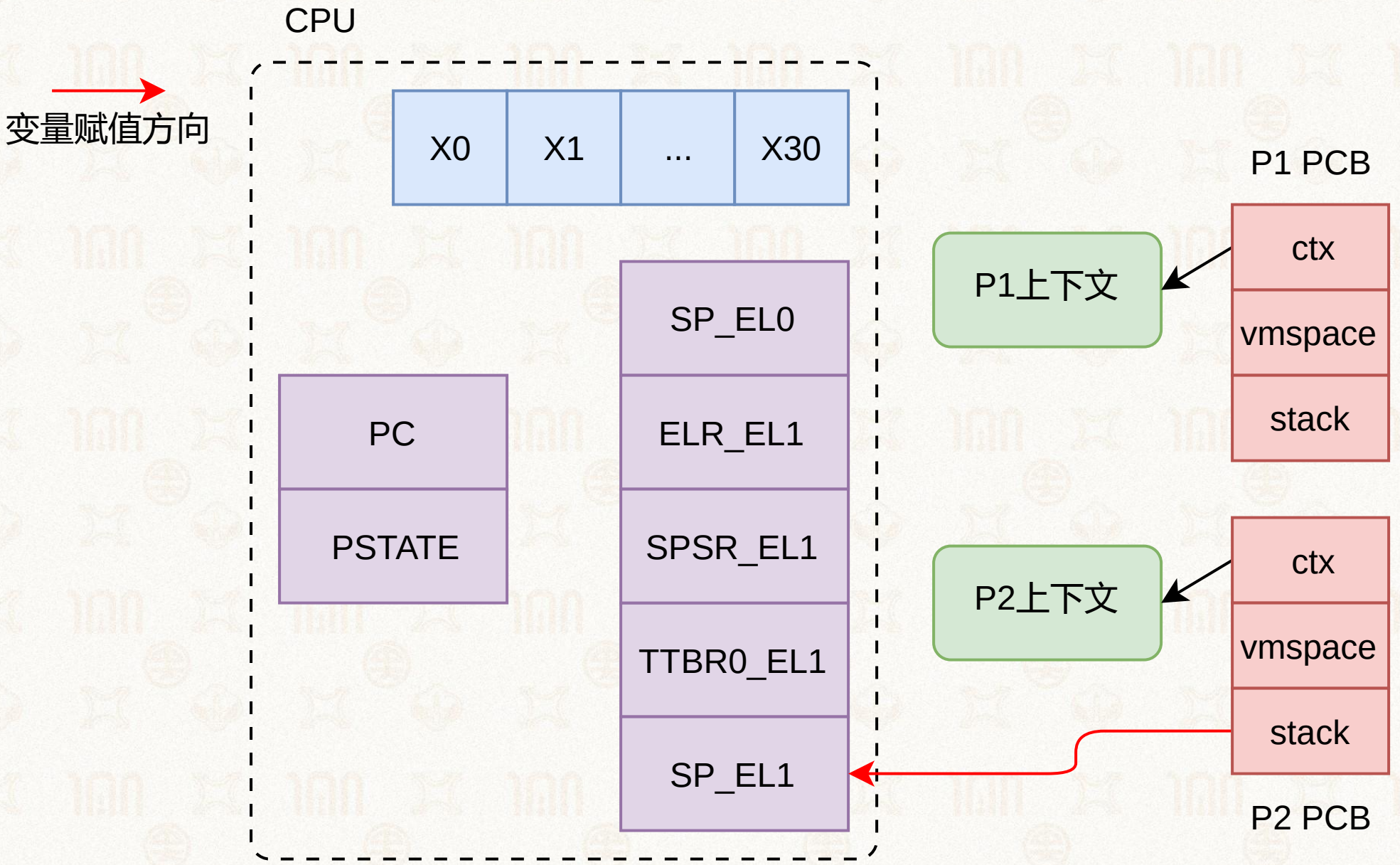
上下文切换：P1上下文保存



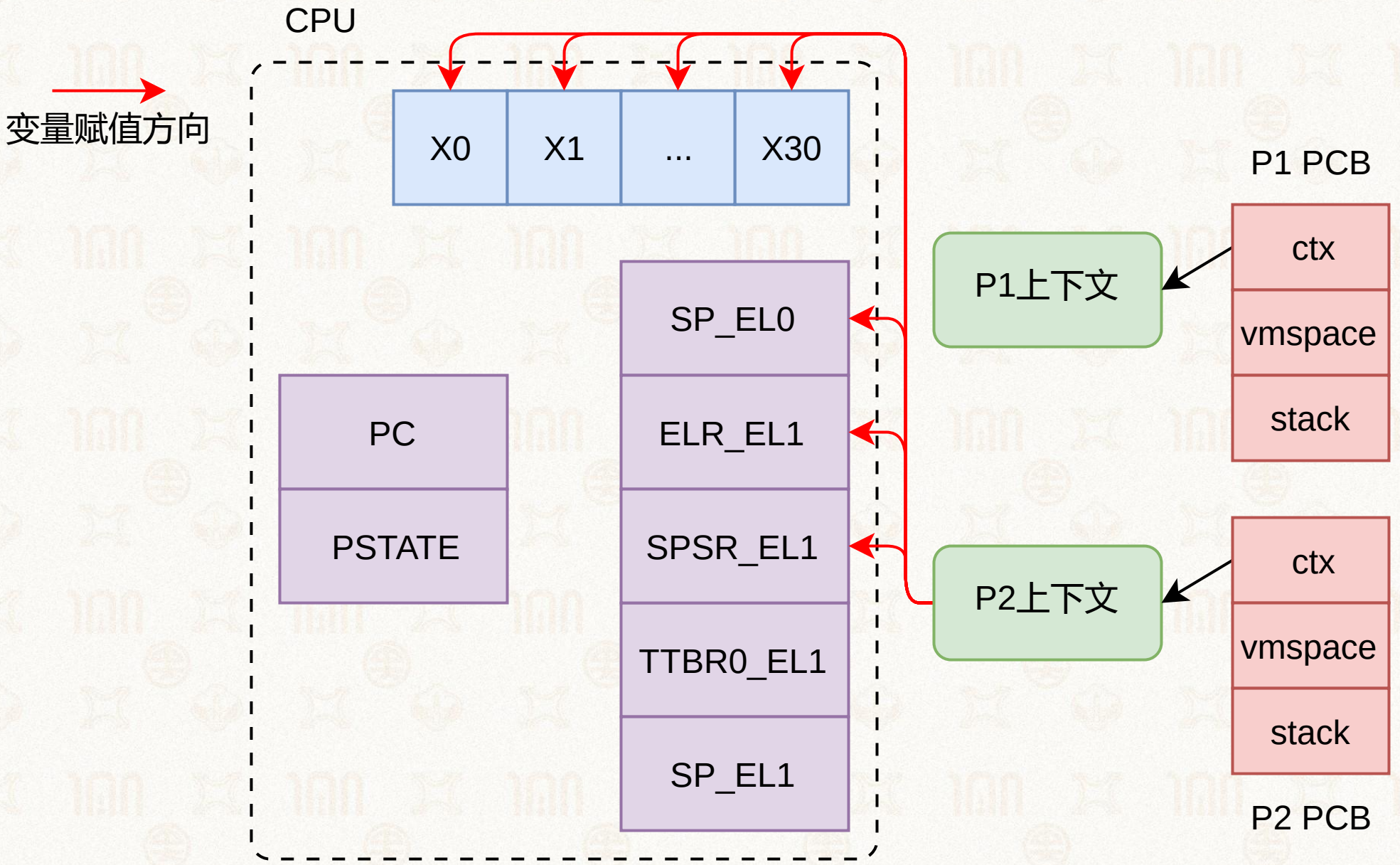
上下文切换：虚拟地址空间切换



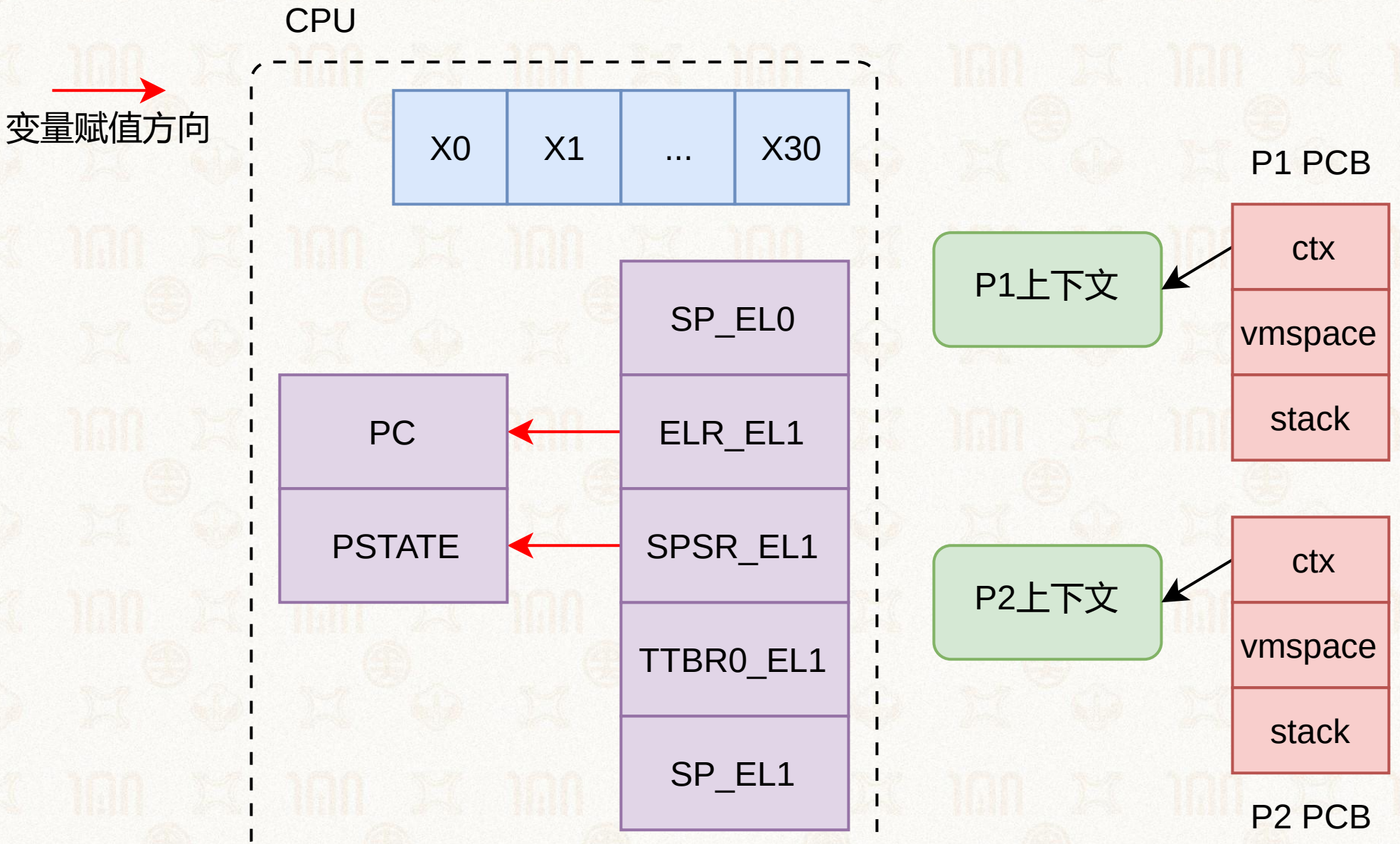
上下文切换：内核栈切换


CPU
变量赋值方向
X0
X1
...
X30
SP_EL0
ELR_EL1
SPSR_EL1
TTBR0_EL1
SP_EL1
PC
PSTATE
P1 PCB
P1上下文
ctx
vmspace
stack
P2上下文
ctx
vmspace
stack
P2 PCB

上下文切换：P2上下文恢复



上下文切换：P2返回用户态



大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

进程的基本操作接口

- 进程创建：fork (spawn, vfork, clone)
- 进程执行：exec
- 进程间同步：wait
- 进程退出：exit/abort

进程创建：fork()

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

pid_t fork(void);
```

还记得它么：

```c
int main () {
    while (1) {
        fork();
    }
}
```

- 语义：为调用进程创建一个一模一样的新进程
  - 调用进程为父进程，新进程为子进程
  - 接口简单，无需任何参数

- fork后的两个进程均为独立进程
  - 拥有不同的进程id
  - 可以并行执行，互不干扰（除非使用特定的接口）
  - 父进程和子进程会共享部分数据结构（内存、文件等）

进程创建：fork()

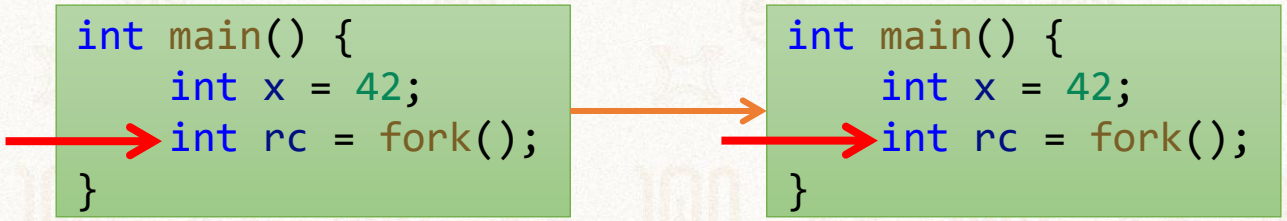
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
    int x = 42;
    int rc = fork();
    if (rc < 0) {
        // fork 失败
        fprintf(stderr, "Fork failed\n");
    } else if (rc == 0) {
        // 子进程
        printf("Child process: rc is: %d; The value of x is: %d\n", rc, x);
    } else {
        // 父进程
        printf("Parent process: rc is %d; The value of x is: %d\n", rc, x);
    }
}
```

```c
int main() {
    int x = 42;
    int rc = fork();
}
```

进程创建：fork()

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

int main(int argc, char* argv[]) {
    int x = 42;
    int rc = fork();
    if (rc < 0) {
        // fork 失败
        fprintf(stderr, "Fork failed\n");
    } else if (rc == 0) {
        // 子进程
        printf("Child process: rc is: %d; The value of x is: %d\n", rc, x);
    } else {
        // 父进程
        printf("Parent process: rc is %d; The value of x is: %d\n", rc, x);
    }
}
```

首先复制一份一模一样的程序：代码、数据相同



调用fork的叫父进程　　被创建的叫子进程

唯一的差异在fork的返回值：

父进程的fork返回子进程的PID值

子进程的fork返回0

fork()的示例

以下代码Hello会出现？次，每次a的值为？

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

void main() {
    int a = 0;
    int rc = fork();
    a++;
    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello pid: %d !\n", rc);
    printf("a is % d\n", a);
}
```

fork()的示例

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

char str[11] = {0};

int main(int argc, char* argv[]) {
    int fd = open("test.txt", O_RDWR);
    if (fork() == 0) {
        ssize_t cnt = read(fd, str, 10);
        printf("Child process: %s\n", str);
    } else {
        ssize_t cnt = read(fd, str, 10);
        printf("Parent process: %s\n", str);
    }
    close(fd);
    return 0;
}
```

- test.txt文件中的内容:
  abcdefghijklmnopqrst

- 猜一猜程序运行的结果:

Child process: abcdefghijklmnopqrst
Parent process: abcdefghijklmnopqrst

Parent process: abcdefghijklmnopqrst
Child process: abcdefghijklmnopqrst

Child process: abcdefghij
Parent process: klmnopqrst

Parent process: abcdefghij
Child process: klmnopqrst

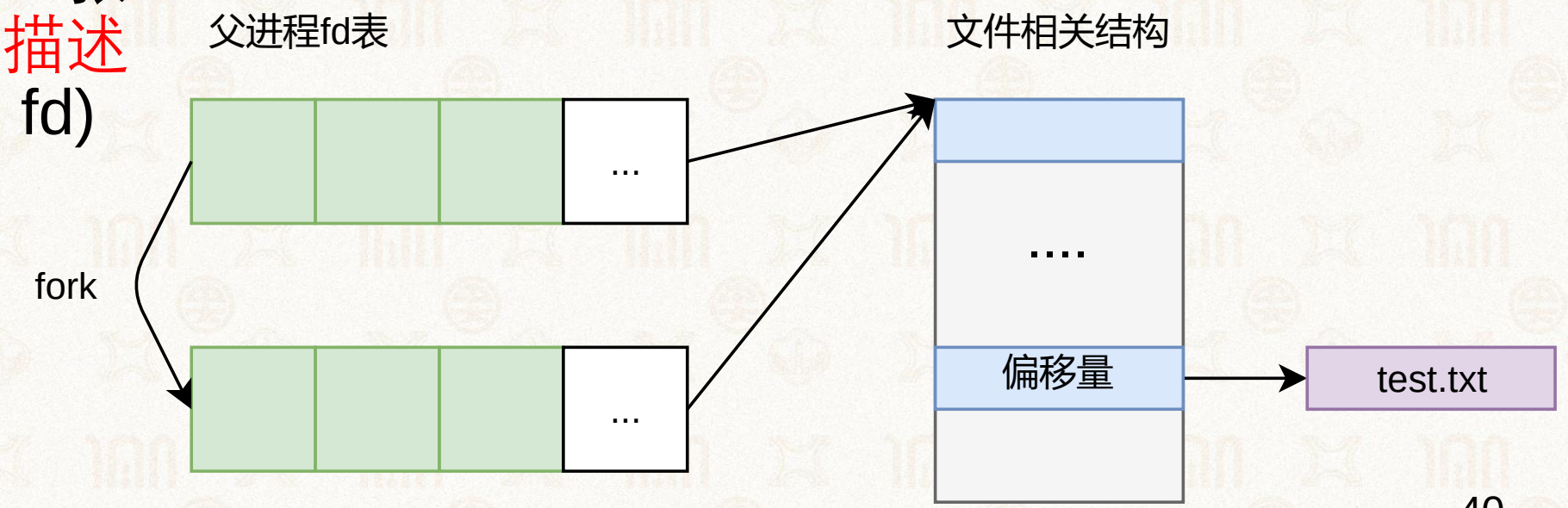
fork()的示例

- 说好的“一模一样”呢？

- 原因：两个进程共享了同一个指向文件的结构体

- 每个进程都会维护一张已打开文件的文件描述符(File Descriptor, fd)表

```
struct task_struct {
    unsigned int                __state;
    void                        *stack;
    struct list_head            tasks;
    struct mm_struct            *mm;
    /* Filesystem information: */
    struct fs_struct            *fs;
    /* Open file information: */
    struct files_struct         *files;
};
```



https://elixir.bootlin.com/linux/v5.16.16/source/include/linux/sched.h#L1068

fork() 参考实现

```
int fork(void)
{
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 虚拟内存初始化：初始化页表基地址
    new_proc->vmspace->pgdir = alloc_new_page();
    // 虚拟内存初始化：将当前进程（父进程）PCB 中页表完整拷贝一份
    copy_vmspace(new_proc->vmspace, cur_proc->vmspace);
    // 上下文初始化：将父进程 PCB 中的上下文完整拷贝一份
    copy_context(new_proc->ctx, old_proc->ctx);
    // 内核栈初始化
    copy_stack(new_proc->stack, old_proc->stack);
    // 返回
    ...
}
```

Windows的进程创建: CreateProcess

- 从头创建进程
- 指定要运行的二进制程序
- 需要配置多个运行参数
- 比fork的逻辑直观，但实操更复杂

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

https://docs.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

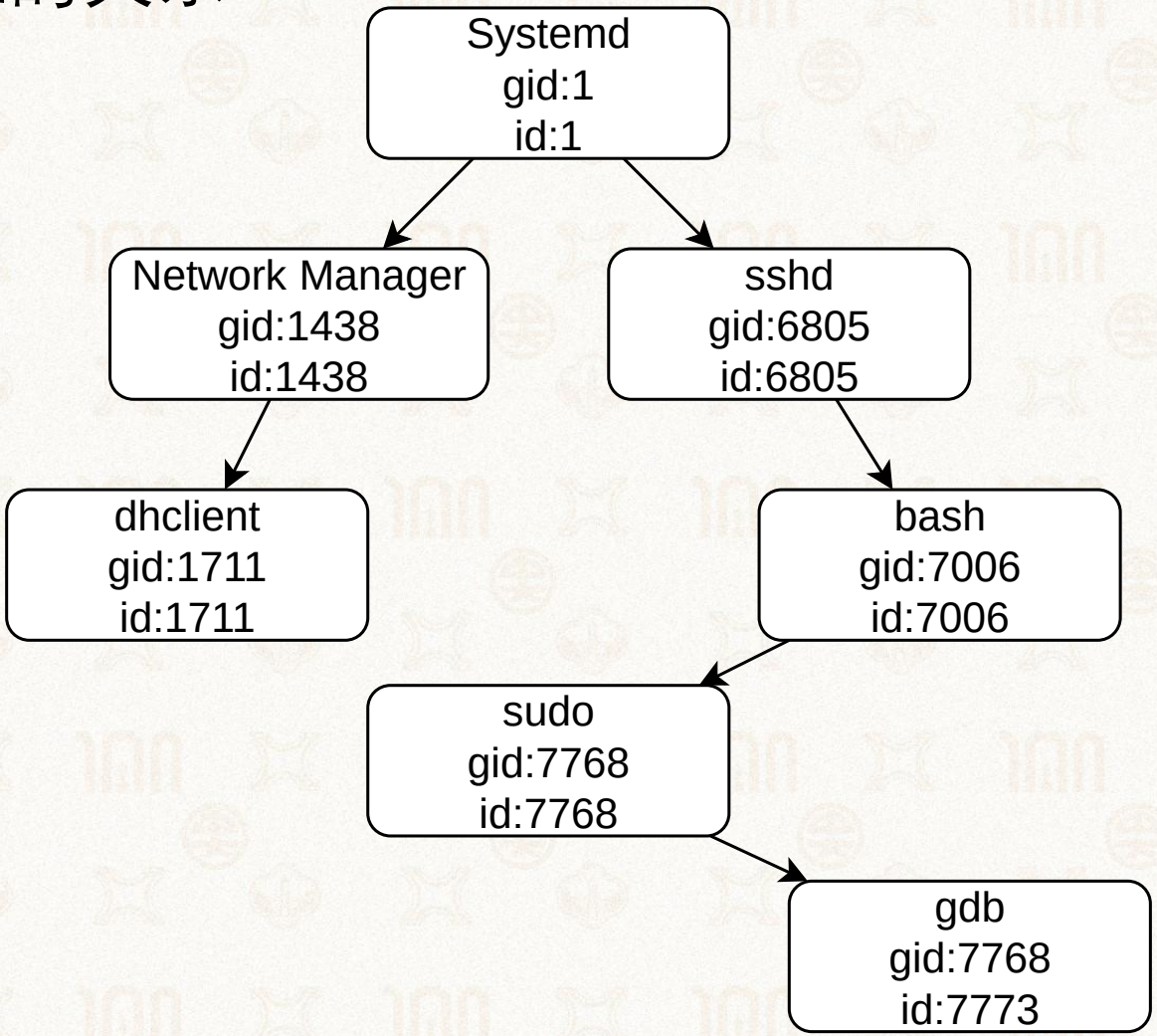
Windows的进程创建: CreateProcess

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void _tmain( int argc, TCHAR *argv[] ) {
    STARTUPINFO si; PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) ); si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    ) { printf( "CreateProcess failed (%d).\n", GetLastError() ); return;}
    WaitForSingleObject( pi.hProcess, INFINITE ); // Wait until child process exits.
    CloseHandle( pi.hProcess ); // Close process and thread handles.
    CloseHandle( pi.hThread );
}
```

https://docs.microsoft.com/zh-cn/windows/win32/procthread/creating-processes

进程树与进程组

- fork为进程之间建立了父进程和子进程的关系
  - 进程之间建立了树型结构
  - Linux可使用pstree命令查看



- 多个进程可以属于同一个进程组
  - 子进程默认与父进程属于同一个进程组
  - 可以向同一进程组中的所有进程发送信号
  - 主要用于shell程序中
  - Linux可使用ps -exjf 命令查看

进程树与进程组

- pstree 以树状结构表示进程树的关系

```
yxsu@lg:~$ pstree
systemd─┬─2*[agetty]
        ├─containerd───10*[{containerd}]
        ├─cron
        ├─dbus-daemon
        ├─dockerd───11*[{dockerd}]
        ├─init-systemd(Ub─┬─SessionLeader───Relay(1496)───fsnotifier-wsl
        │                 ├─SessionLeader───Relay(34363)───ion.clangd.main───7*[{ion.clangd.main}]
        │                 ├─SessionLeader───Relay(35886)───sh───fork_demo───fork_demo
        │                 ├─SessionLeader───Relay(40323)───bash───pstree
        │                 ├─init───{init}
        │                 ├─2*[login───bash]
        │                 └─{init-systemd(Ub}
        ├─networkd-dispat
        ├─packagekitd───2*[{packagekitd}]
        ├─polkitd───2*[{polkitd}]
        ├─rsyslogd───3*[{rsyslogd}]
        ├─snapd───13*[{snapd}]
        ├─5*[snapfuse───2*[{snapfuse}]]
        ├─snapfuse───5*[{snapfuse}]
        ├─2*[snapfuse───4*[{snapfuse}]]
        ├─subiquity-serve───python3.10─┬─python3
        │                              └─{python3.10}
        ├─2*[systemd───(sd-pam)]
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-resolve
        ├─systemd-udevd
        └─unattended-upgr───{unattended-upgr}
```

进程树与进程组

- ps -exjf 查看进程组

fork_demo运行的结果：

Parent process: rc is 35888; The value of x is: 42
Child process: rc is: 0; The value of x is: 42

```
yxsu@lg:~$ ps -exjf
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
  40318   40323   40323   40323 pts/5     41276 Ss    1000   0:00 -bash HOSTTYPE=x86_64 LANG=C.UTF-8 PATH=/usr/local/sbin
  40323   41276   41276   40323 pts/5     41276 R+    1000   0:00  \_ ps -exjf SHELL=/bin/bash WSL2_GUI_APPS_ENABLED=1
  35885   35886   35886   35886 pts/4     35886 Ss+   1000   0:00 /bin/sh -c cd /mnt/c/Users/suyux/Documents/GitHub/sse202_internal
  35886   35887   35886   35886 pts/4     35886 S+    1000   0:00  \_ /mnt/c/Users/suyux/Documents/GitHub/sse202_internal/fork_demo
  35887   35888   35886   35886 pts/4     35886 S+    1000   0:00      \_ /mnt/c/Users/suyux/Documents/GitHub/sse202_internal/fork_demo
  34360   34363   34363   34363 pts/2     34363 Ssl+  1000   0:04 /tmp/clangd728fd33ac8de77780dd7b6e2be737d6b8369db07/clangd --clion
   1513    1845    1845    1513 pts/1      1845 S+    1000   0:00 -bash HOME=/home/yxsu SHELL=/bin/bash USER=yxsu LOGNAME=yxsu
   1484    1496    1496    1496 pts/0      1496 Ss+   1000   0:00 ./fsnotifier-wsl HOSTTYPE=x86_64 LANG=C.UTF-8 PATH=/usr/local/sbin
      1    1713    1713    1713 ?           -1 Ss    1000   0:00 /lib/systemd/systemd --user LANG=C.UTF-8 PATH=/usr/local/sbin:/usr/local/bin
   1713    1724    1713    1713 ?           -1 S     1000   0:00  \_ (sd-pam)
```

进程的执行：exec

- 为进程指定可执行文件和参数

可执行文件的位置　　运行参数　　环境变量

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[],  char *const envp[]);
```

- 在fork之后调用
  - exec在载入可执行文件后会重置地址空间

```
yxsu@Dell-T6401:~/os/process$ strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffe7c7e50f0 /* 26 vars */) = 0
brk(NULL)                                       = 0x560913b5d000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff39be4a50) = -1 EINVAL (无效的参数)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (没有那个文件或目录)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
```

exec示例

```c
/* myecho.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    for (int j = 0; j < argc; j++) {
        printf("argv[%d]: %s\n", j, argv[j]);
    }
    exit(EXIT_SUCCESS);
}
```

先编译：

```
gcc execve_demo.c -o execve_demo
gcc myecho.c -o myecho
```

再运行：

```
yxsu@Dell-T6401:~/os/process$ ./execve_demo ./myecho
argv[0]: ./myecho
argv[1]: hello
argv[2]: world
```

```c
/* execve_demo.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    char *newargv[] = {
        NULL, "hello", "world", NULL };
    char *newenviron[] = { NULL };
    if (argc != 2) {
        exit(EXIT_FAILURE);
    }
    newargv[0] = argv[1];

    execve(argv[1], newargv, newenviron);
    /* execve() returns only on error */
    perror("execve");
    exit(EXIT_FAILURE);
}
```

fork的优缺点分析

- fork的优点
  - 接口非常简洁
  - 将进程“创建”和“执行”（exec）解耦，提高了灵活度
  - 刻画了进程之间的内在关系（进程树、进程组）

- fork的缺点
  - 完全拷贝过于粗暴（不如clone）
  - 性能差、可扩展性差（不如vfork和spawn）
  - 不可组合性 (例如：fork() + pthread())

fork的替代接口

- vfork：类似于fork，但让父子进程共享同一地址空间
  - 优点：连映射都不需要拷贝，性能更好
  - 缺点：
    - 只能用在”fork + exec”的场景中
    - 共享地址空间存在安全问题

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int tmp = 3;

int main() {
    pid_t res = vfork();
    if (res < 0) {
        printf("vfork failed");
        exit(-1);
    } else if (res == 0) {
        tmp = 10;
        printf("Child process: res = %d\n", tmp);
    } else {
        printf("Parent process: res = %d\n", tmp);
    }
    return 0;
}
```

fork的替代接口：vfork参考实现

```
int vfork(void)
{
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 虚拟内存初始化：直接使用父进程的页表
    new_proc->vmspace->pgdir = cur_proc->vmspace->pgdir;

    // 上下文初始化：将父进程 PCB 中的上下文完整拷贝一份
    copy_context(new_proc->ctx, old_proc->ctx);
    // 阻塞父进程，直到子进程退出或调用 exec
    block_process(cur_proc);
    // 返回
    ...
}
```

fork() 多出来的部分

```
// 虚拟内存初始化：将当前进程（父进程）PCB 中页表完整拷贝一份
    copy_vmspace(new_proc->vmspace, cur_proc->vmspace);
```

fork的替代接口

- posix_spawn: 相当于fork + exec
  - 优点：可扩展性、性能较好
  - 缺点：不如fork灵活

```c
#include <spawn.h>

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

```c
pid_t child_pid;
int ret;

ret = posix_spawn(&child_pid, "/home/yxsu/process/a.out", NULL, NULL, NULL, NULL);
if (ret != EOK) {
    printf("posix_spawn() failed: %s\n", strerror(ret));
    return EXIT_FAILURE;
}
printf("Child pid: %d\n\n", child_pid);
```

fork的替代接口：posix_spawn参考实现

```
int posix_spawn(pid_t *pid, const char *path,
                ...,
                const posix_spawnattr_t *attrp,
                char *const argv[], 5 char *const envp[]) {
    // 先执行 vfork 创建一个新进程
    int ret = vfork();
    if (ret == 0) {
        // 子进程：在 exec 之前，根据参数对其进行配置
        prepare_exec(attrp, ...);
        // 执行 exec
        exec(path, argv, envp);
    } else {
        // 父进程：将子进程的 pid 设置到传入的参数中
        *pid = ret;
        return 0;
    }
}
```

fork的替代接口

- clone: fork的“进阶版”，可以选择性地不拷贝内存
  - 优点：高度可控，可依照需求调整
  - 缺点：接口比fork复杂，选择性拷贝容易出错

```
#include <sched.h>

int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
          /* pid_t *parent_tid, void *tls, pid_t *child_tid */);
```

点开以下链接，滑到最底可看示例代码：

https://www.man7.org/linux/man-pages/man2/clone.2.html

fork的替代接口： clone参考实现

```
int clone(..., int flags, ...) {
    // 创建一个新的 PCB，用于管理新进程
    struct process *new_proc = alloc_process();
    // 如果设置了 CLONE_VM 则直接使用父进程的页表，否则拷贝一份
    if (flags & CLONE_VM) {
        new_proc->vmspace->pgdir = cur_proc->vmspace->pgdir;
    } else {
        new_proc->vmspace->pgdir = alloc_new_page();
        copy_vmspace(new_proc->vmspace, cur_proc->vmspace);
    }
    // 上下文初始化：将父进程 PCB 中的上下文完整拷贝一份
    copy_context(new_proc->ctx, old_proc->ctx);
    // 如果设置了 CLONE_VFORK 则阻塞父进程
    if (flags & CLONE_VFORK) {
        block_process(cur_proc);
    }
    // 返回
    ...
}
```

waitpid: 进程间监控及同步


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

int main(int argc, char* argv[]) {
    int rc = fork();
    if(rc < 0) {
        fprintf(stderr, "Fork failed\n"); // fork 失败
    } else if (rc == 0) { // 子进程
        printf("Child process: existing\n");
    } else { // 父进程
        int status = 0;
        if(waitpid(rc, &status, 0) < 0) {
            fprintf(stderr, "Parent process: waitpid failed\n"); exit(-1);
        }
        if(WIFEXITED(status)) {
            printf("Parent process: my child has exited\n");
        } else {
            fprintf(stderr, "Parent processes: waitpid returns for unknown reasons\n");
        }
    }
}
```

- 保证按一定的顺序执行

waitpid: 实现

```
void process_waitpid_v3(int id) {
    // 如果没有子进程，直接返回
    if (!cur_proc->children)
        return;
    while (TRUE) {
        not_exit = FALSE;
        // 扫描内核的进程列表，寻找对应进程
        for proc in all_processes {
            not_exist = FALSE; // 若发现该进程还在进程列表中，说明还未退出
            if (proc->is_exit) {
                *status = proc->exit_status; // 若发现该进程已经退出，记录其退出状态
                // 回收进程的 PCB 并返回
                destroy_process(proc);
                return;
            } else {
                wait_in_kernel(); // 如果没有退出，则陷入等待，否则直接返回
            }
        }
        // 如果列表中不存在该进程，则直接返回
        if (not_exit)
            return;
    }
}
```

大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫