

# 进程

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

# 版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

# 前情提要

# 场景-1：应用触发on-demand paging

- **问：当应用调用malloc时，OS是否需要分配物理内存？**
  - 应用调用malloc后，返回的虚拟地址属于某个VMA
  - 但虚拟地址对应的页表项的valid bit可能为0
  - 当第一次访问新分配的虚拟地址时，CPU会触发page fault
- **操作系统需要做（即page-fault handler）：**
  - 找到一块空闲的物理内存页 ← **物理内存管理（页粒度）**
  - 修改页表，将该物理页映射到触发page-fault的虚地址所在虚拟页
  - 恢复应用，重复执行触发page-fault的那行代码

## 场景-2：内核运行中需要进行动态内存分配

- 内核自身用到的数据结构

- 为每个进程创建的process, VMA等数据结构
- 动态性：用时分配，用完释放，类似用户态的malloc/new
- 数据结构大小往往远小于页粒度
- 接口：`vaddr_t kmalloc(u64 size);`  
`void kfree(vaddr_t kva);`

## 场景-3：换页 (swap)

- **换出操作：物理内存不够时**
  - OS选择不常用的物理内存（不同的选择策略）
  - OS将内存中的数据写入磁盘块，并记录磁盘块与内存的关联
  - OS更新页表，将对应页表项的valid bit设置为0
- **换入操作：当换出的页被访问时，触发page fault**
  - OS判断该地址所在页被换出，找到对应的磁盘块
  - OS分配空闲的物理内存页；若没有空闲页，则再次进行换出操作
  - OS将磁盘块中的数据读入前一步分配的内存页
  - OS更新页表，将对应页表项的valid bit设置为1

# 复习：换页五问

- Q1: 如何判断缺页异常是由于换页引起的?
- Q2: 何时进行换出操作?
- Q3: 换页机制的代价与如何缓解?
- Q4: 如何选择换出的页?
- Q5: 什么是反向映射?

# 颠簸现象 (Thrashing Problem)

- **直接原因**

- 过于频繁的缺页异常（物理内存总需求过大）

- **大部分 CPU 时间都被用来处理缺页异常**

- 等待缓慢的磁盘 I/O 操作
- 仅剩小部分的时间用于执行真正有意义的工作

- **调度器造成问题加剧**

- 等待磁盘 I/O导致CPU利用率下降
- 调度器载入更多的进程以期提高CPU利用率
- 触发更多的缺页异常、进一步降低CPU利用率、导致连锁反应



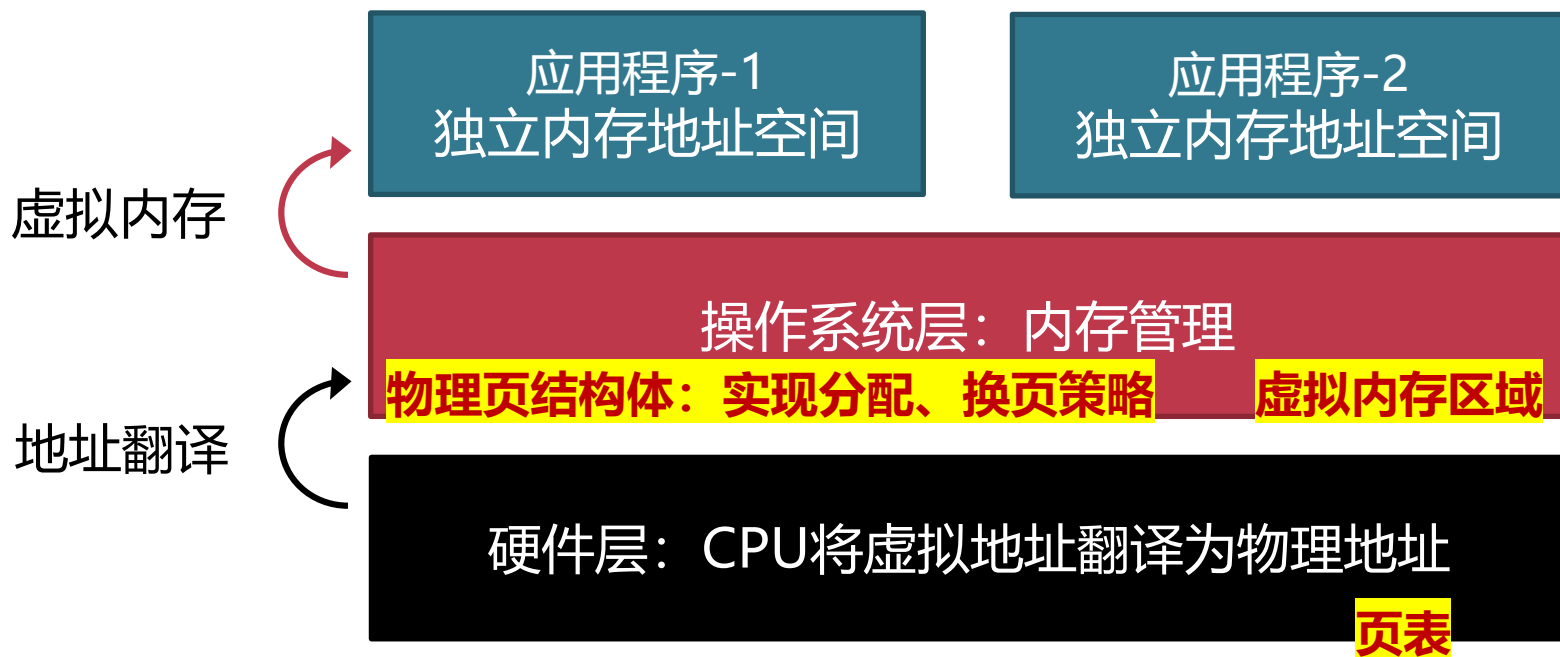
# 工作集模型（有效避免Thrashing）

- **一个进程在时间 $t$ 的工作集 $W(t, x)$ :**
  - 其在时间段  $(t - x, t)$ 内使用的内存页集合
  - 也被视为其在未来（下一个 $x$ 时间内）会访问的页集合
  - 如果希望进程能够顺利进展，则需要将该集合保持在内存中
- **工作集模型：All-or-nothing**
  - 进程工作集要么都在内存中，要么全都换出
  - 避免thrashing，提高系统整体性能表现

## 场景-4：设备需要分配DMA内存

- **DMA：设备绕过CPU直接访存**
  - 由于绕过CPU的MMU，因此直接访问物理地址
  - 通常需要大段连续的物理内存
- **操作系统必须有能力分配连续的物理页**
  - 需要用一种高效的方式来组织和管理物理页
- 之后的课程会进一步介绍

# 小结：内存管理涉及的关键数据结构



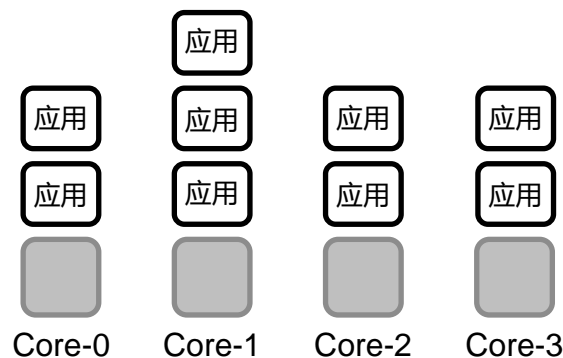


# 进程

# 回顾：分时复用有限的CPU资源

- 分时复用CPU

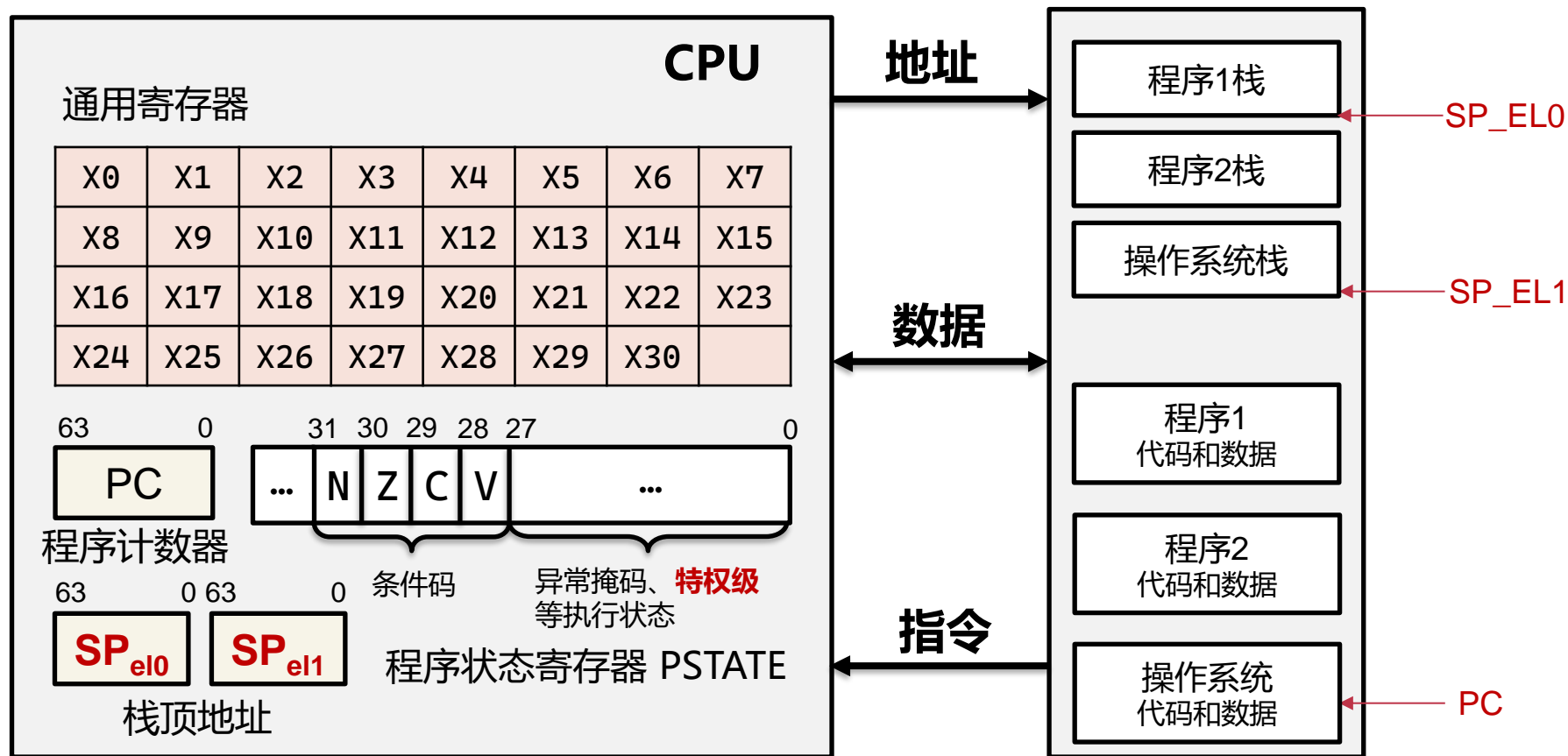
- 让多个应用程序**轮流**使用处理器核心
- 何时切换：**操作系统**决定
  - 运行时间片（例如100ms）
- 高频切换：看起来是多个应用“同时”执行



# 回顾：处理器上下文(CPU Context)

- 操作系统为每个进程维护**处理器上下文**
  - 包含恢复进程执行所需要的状态
  - 思考：进程A执行到main函数任意一条指令，切换到进程B执行，一段时间后，再切回到进程A执行
    - 为完成此过程，有哪些状态需要保存？
  - 具体包括：
    - **PC寄存器值，栈寄存器值，通用寄存器值，状态寄存器值**

# 回顾：程序员视角



## 进程的表示：PCB



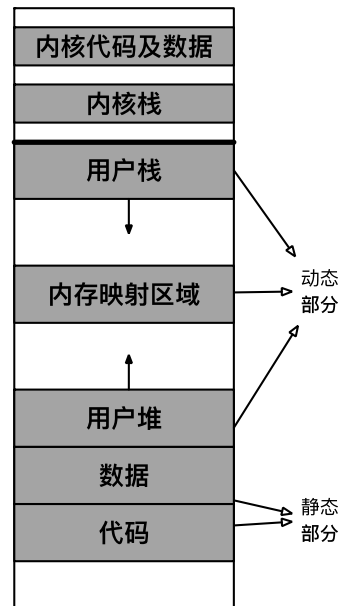
# 进程：运行中的程序

- 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



# 如何表示进程：进程控制块（PCB）

- 每个进程都对应一个元数据，称为“进程控制块” PCB
  - 进程控制块存储在内核态（为什么？）
- 想一想：进程控制块里至少应该保存哪些信息？
  - 独立的虚拟地址空间
  - 独立的处理器上下文

```
1 // 一种简单的 PCB 结构实现
2 struct process_v1 {
3     // 处理器上下文
4     struct context *ctx;
5     // 虚拟地址空间（包含页表基地址）
6     struct vmSPACE *vmSPACE;
7     // 内核栈
8     void *stack;
9 };
```

PCB结构（第一版）

# 内核栈与用户栈

- 应用需要 “又一个栈”（内核栈）？

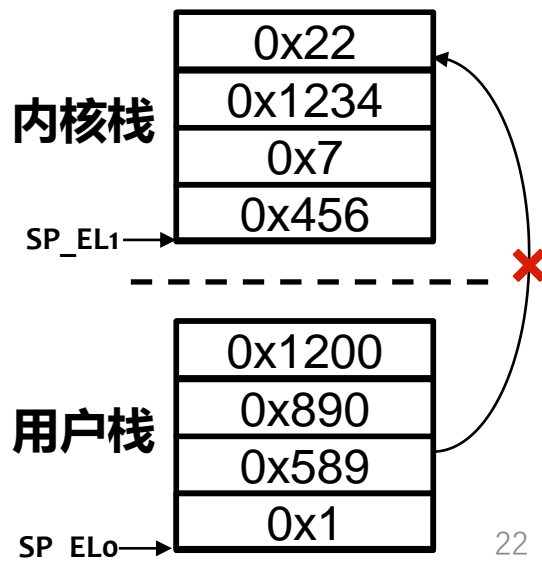
- 进程在内核中依然需要执行代码，有读写临时数据的需求
- 进程在用户态和内核态的数据应该相互隔离，增强安全性
- 能否使用原本的内核栈？

取决于是否支持内核态抢占

- AArch64实现：两个栈指针寄存器

- SP\_EL1, SP\_EL0
- CPU根据内核态与用户态使用相应的SP

\* 本PPT中假设进程使用独立内核栈



# 调度进程

- **调度队列**
  - 队列中每个元素是进程PCB
- **内核调度器从调度队列中取出一个进程PCB**
  - `schedule()`: 进程切换
  - 恢复其上下文状态
  - 通过`eret`指令恢复其执行

## 进程的创建

# 回顾：从程序员视角如何使用进程？

- 以shell为例：一个不断循环接收命令的程序
  - 收到命令后，如何执行？ -> 创建一个进程，委派给它
    - 为什么不能让shell进程自己执行？

```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```

# 如何实现进程的创建？

- 回顾：进程包含哪些内容？
  - 用户视角：代码、数据、堆栈
  - 内核视角：PCB、虚拟地址空间、上下文、内核栈
  - 创建进程就是创建及初始化以上内容过程

# 进程创建的伪代码实现

## 1. PCB相关内容初始化

## 2. 可执行文件加载

## 3. 准备运行环境

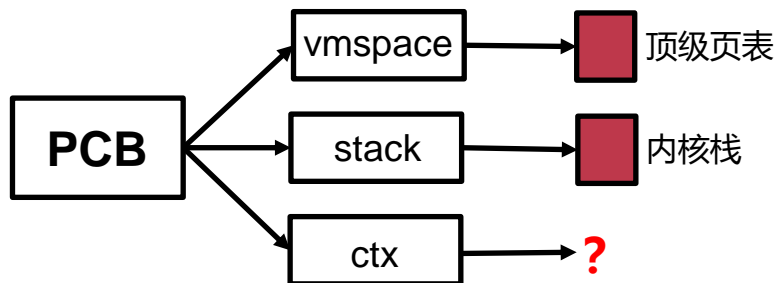
## 4. 处理器上下文初始化

```
1 int process_create(char *path, char *argv[], char  
   ↳ *envp[])  
2 {  
3     // 创建一个新的 PCB, 用于管理新进程  
4     struct process *new_proc = alloc_process();  
5     // 虚拟内存初始化: 初始化虚拟地址空间及页表基地址  
6     init_vmspace(new_proc->vmspace);  
7     new_proc->vmspace->pgtbl = alloc_page();  
8  
9     // 内核栈初始化  
10    init_kern_stack(new_proc->stack);  
11  
12    // 加载可执行文件并映射到虚拟地址空间  
13    struct file *file = load_elf_file(path);  
14    for (struct seg loadable_seg : file->segs)  
15        vmspace_map(new_proc->vmspace, loadable_seg);  
16  
17    // 准备运行环境: 创建并映射用户栈  
18    void *stack = alloc_stack(STACKSIZE);  
19    vmspace_map(new_proc->vmspace, stack);  
20  
21    // 准备运行环境: 将参数和环境变量放到栈上  
22    prepare_env(stack, argv, envp);  
23    // 上下文初始化  
24    init_process_ctx(new_proc->ctx);  
25    // 返回  
26 }
```



# 进程创建（一）：PCB相关初始化

- PCB及其包含的内容都需要创建及初始化
  - 分配PCB本身的数据结构
  - 初始化PCB：虚拟内存
    - 创建及初始化`vmSPACE`数据结构
    - 分配一个物理页，作为顶级页表
  - 内核栈：分配物理页，作为进程内核栈
  - 【思考】处理器上下文初始化？ 上下文在之后（应用运行前）才会初始化



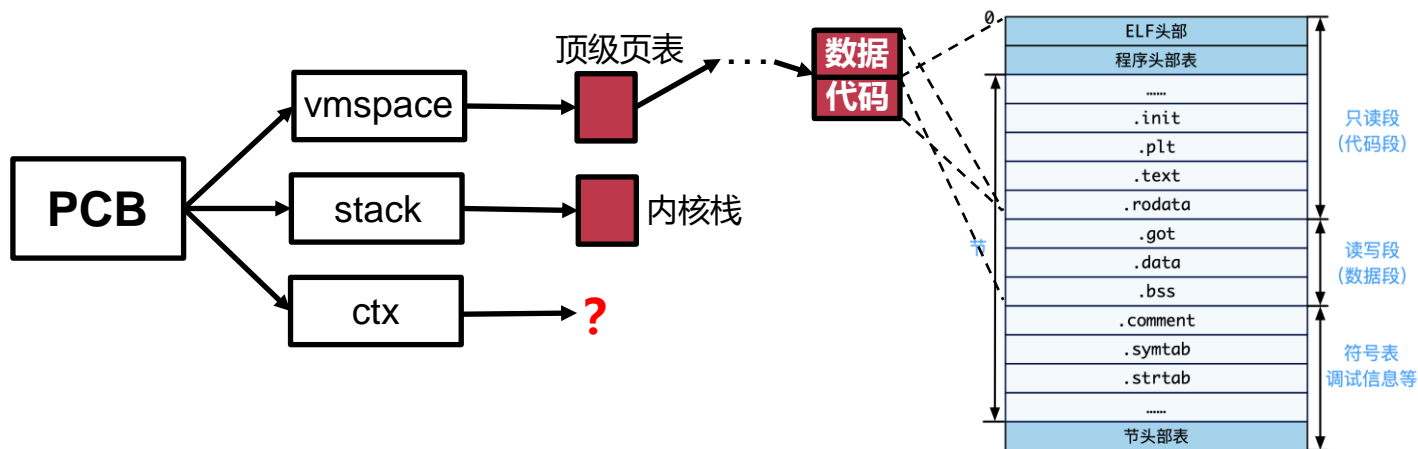
```
// 创建一个新的 PCB，用于管理新进程
struct process *new_proc = alloc_process();
// 虚拟内存初始化：初始化虚拟地址空间及页表基地址
init_vmSPACE(new_proc->vmSPACE);
new_proc->vmSPACE->pgtbl = alloc_page();

// 内核栈初始化
init_kern_stack(new_proc->stack);
```

# 进程创建（二）：可执行文件加载

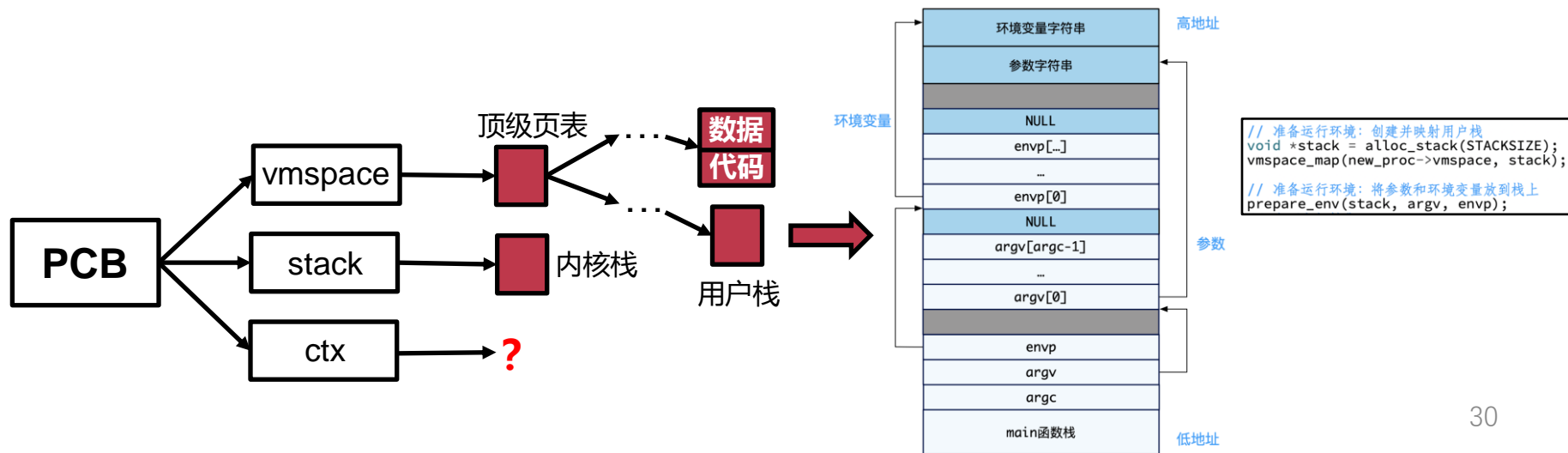
- 可执行文件通常有固定的存储格式
- 以ELF (Executable and Linkable Format) 为例
  - 从程序头部表可以获取需要的段所在位置
  - 通常只有代码段和数据段需要被加载 (loadable)
  - 加载即从ELF文件中映射到虚拟地址空间的过程

```
// 加载可执行文件并映射到虚拟地址空间
struct file *file = load_elf_file(path);
for (struct seg loadable_seg : file->segs)
    vmpace_map(new_proc->vmpace, loadable_seg);
```



# 进程创建（三）：准备运行环境

- 在返回用户态运行前，还需为进程准备运行所需的环境
  - 分配用户栈（分配物理内存并映射到虚拟地址空间）
  - 准备程序运行时的环境
    - 回顾main函数：int main(int argc, char\* argv[], char \*envp[])



# 进程创建（四）：处理器上下文初始化

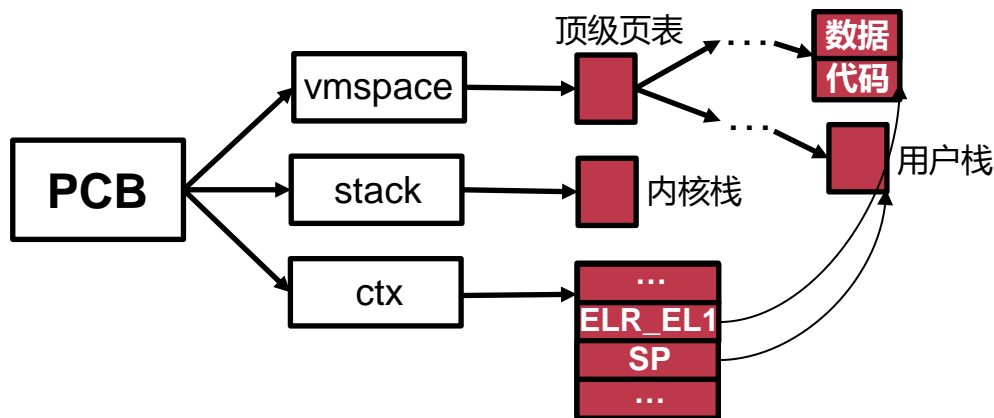
- 为什么直到最后才初始化处理器上下文？

- 其包含的内容直到前序操作完成才确定

- SP：用户栈分配后才确定地址

- PC（保存在ELR\_EL1）：加载ELF后才知道入口所在地址

- 通用寄存器初始值可直接赋为0 **问：处理器状态保存寄存器SPSR\_EL1呢？**

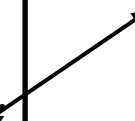


## 进程的退出与等待

# 回顾：从程序员视角如何使用进程？

- 以shell为例：一个不断循环接收命令的程序
  - 当ls执行结束后，进程也就失去了存在的意义
    - 需要一个系统调用使进程**显式**退出

```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```



```
int main (...) {  
    // ls的逻辑  
    ...  
    process_exit(0);  
}
```

# 回顾：从程序员视角如何使用进程？

- 疑问：main函数有时并不调用exit？
  - 使用return更为常见
    - 此时可由libc的代码调用process\_exit，退出进程

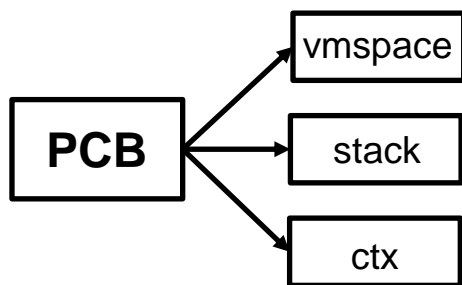
```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```

```
int main (...) {  
    // ls的逻辑  
    ...  
    return 0;  
}
```

```
...  
main(...);  
process_exit(0);  
}
```

# 进程退出的实现（第一版）

- 销毁PCB及其中保存的所有内容

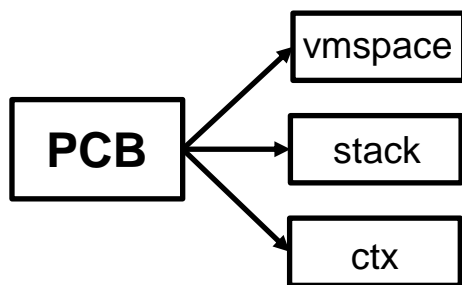


```
1 void process_exit_v1(void)
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->ctx);
5     // 销毁内核栈
6     destroy_kern_stack(curr_proc->stack);
7     // 销毁虚拟地址空间
8     destroy_vmSPACE(curr_proc->vmSPACE);
9     // 销毁 PCB
10    destroy_process(curr_proc);
11    // 告知内核选择下个需要执行的进程
12    schedule();
13 }
```



# 进程退出的实现（第一版）

- 销毁PCB及其中保存的所有内容
- 内核选择其他进程执行



```
1 void process_exit_v1(void)
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->ctx);
5     // 销毁内核栈
6     destroy_kern_stack(curr_proc->stack);
7     // 销毁虚拟地址空间
8     destroy_vmSPACE(curr_proc->vmSPACE);
9     // 销毁 PCB
10    destroy_process(curr_proc);
11    // 告知内核选择下个需要执行的进程
12    schedule();
13 }
```

# 进程退出的实现（第一版）的问题

- 问题1：内核栈应该何时销毁？

- 思路：不能在进程退出的系统调用中销毁正在使用的内核栈

- 问题2：父进程如何知道子进程是否运行结束？

- 思路：提供系统调用

```
1 void process_exit_v1(void)
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->ctx);
5     // 销毁内核栈
6     destroy_kern_stack(curr_proc->stack);
7     // 销毁虚拟地址空间
8     destroy_vmspace(curr_proc->vmSPACE);
9     // 销毁 PCB
10    destroy_process(curr_proc);
11    // 告知内核选择下个需要执行的进程
12    schedule();
13 }
```

# 回顾：从程序员视角如何使用进程？

- 以shell为例：一个不断循环接收命令的程序
  - 当ls进程执行时，如何使shell等待，直到ls退出？
    - 引入新系统调用：进程等待（process\_waitpid）

```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```

# 回顾：从程序员视角如何使用进程？

- 以shell为例：一个不断循环接收命令的程序
  - 新的问题：如何使shell知道应该等待哪个进程？
    - 方法：为每个进程引入进程标识符 (id)

```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```

## PCB结构 (第二版)

```
1 struct process_v2 {  
2     // 处理器上下文  
3     struct context *ctx;  
4     // 虚拟地址空间 (包含页表基地址)  
5     struct vmpace *vmpace;  
6     // 内核栈  
7     void *stack;  
8     // 进程标识符  
9     int pid;  
10 };
```

# 进程等待的实现（第一版）

- 假设内核使用进程列表维护所有进程
  - 仍然调用`schedule`，让其他进程执行

```
1 void process_waitpid_v1(int id)
2 {
3     while (TRUE) {
4         bool not_exit = FALSE;
5         // 扫描内核的进程列表，寻找对应进程
6         for (struct process *proc : all_processes) {
7             // 若发现该进程还在进程列表中，说明还未退出
8             if (proc->pid == id)
9                 not_exit = TRUE; break;
10        }
11        // 如果没有退出，则调度下个进程执行，否则直接返回
12        if (not_exit)
13            schedule();
14        else
15            return;
16    }
17 }
```

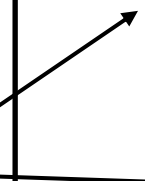
# 第一版实现的限制

- 信息太少了!

- ls进程退出时的返回值，shell进程完全获取不到
- 应该如何改进呢?

```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
        process_waitpid(id);  
    }  
    else if ...  
    ...  
}
```

```
int main (...) {  
    // ls的逻辑  
    ...  
    process_exit(1);  
}
```

Two arrows originate from the `process_waitpid(id);` line in the shell code box. One arrow points to the `process_exit(1);` line in the `main` function box, and the other points to the `process_create("/bin/ls");` line in the same box, illustrating the flow of control and the problem of not capturing the exit status.

# 改进1：为进程添加退出状态支持

- 分别修改PCB和process\_exit的实现

## PCB: 增加退出状态

```
1 struct process_v3 {  
2     // 处理器上下文  
3     struct context *ctx;  
4     // 虚拟地址空间 (包含页表基地址)  
5     struct vmSPACE *vmSPACE;  
6     // 内核栈  
7     void *stack;  
8     // 进程标识符  
9     int pid;  
10    // 退出状态  
11    int exit_status;  
12    // 标记是否退出  
13    bool is_exit;  
14 };
```

## process\_exit: 退出前设置退出状态

```
1 void process_exit_v2(int status)  
2 {  
3     // 销毁上下文结构  
4     destroy_ctx(curr_proc->ctx);  
5     // 销毁内核栈  
6     destroy_kern_stack(curr_proc->stack);  
7     // 销毁虚拟地址空间  
8     destroy_vmSPACE(curr_proc->vmSPACE);  
9     // 保存退出状态  
10    curr_proc->exit_status = status;  
11    // 标记进程为退出状态  
12    curr_proc->is_exit = TRUE;  
13    // 告知内核选择下个需要执行的进程  
14    schedule();  
15 }
```

## 改进2：修改process\_waitpid

- 如果进程已设置为退出，则记录其退出状态并回收
  - 将回收操作由exit移到了waitpid

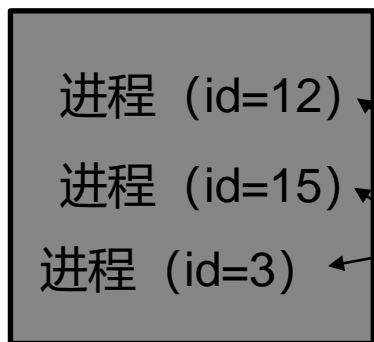
```
1 void process_waitpid_v2(int id, int *status)
2 {
3     while (TRUE) {
4         bool not_exist = TRUE;
5         // 扫描内核维护的进程列表，寻找对应进程
6         for (struct process *proc : all_processes) {
7             if (proc->pid == id) {
8                 // 标记已找到对应进程，并检查其是否已经退出
9                 not_exist = FALSE;
10                if (proc->is_exit) {
11                    // 若发现该进程已经退出，记录其退出状态
12                    *status = proc->exit_status;
13                    // 回收进程的 PCB 并返回
14                    destroy_process(proc);
15                    return;
16                } else {
17                    // 如果没有退出，则调度下个进程执行
18                    schedule();
19                }
20            }
21        }
22        // 如果列表中不存在该进程，则直接返回
23        if (not_exist)
24            return;
25    }
26 }
```



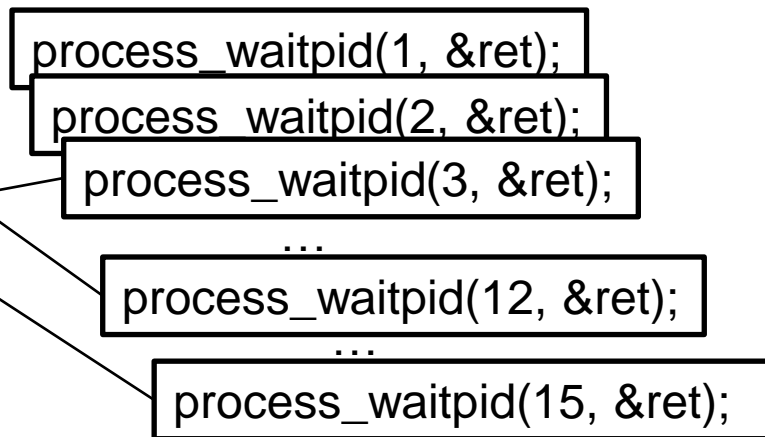
# 还有什么可以改进的地方呢?

- 考虑两个用户的场景 (小明和小红)
  - 如果小红想偷看小明进程退出的状态, 应该怎么做?
    - 暴力枚举可能的id, 对它们都调用process\_waitpid

**小明的进程 (不可见)**



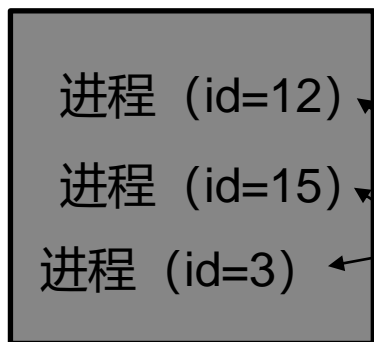
**小红的程序**



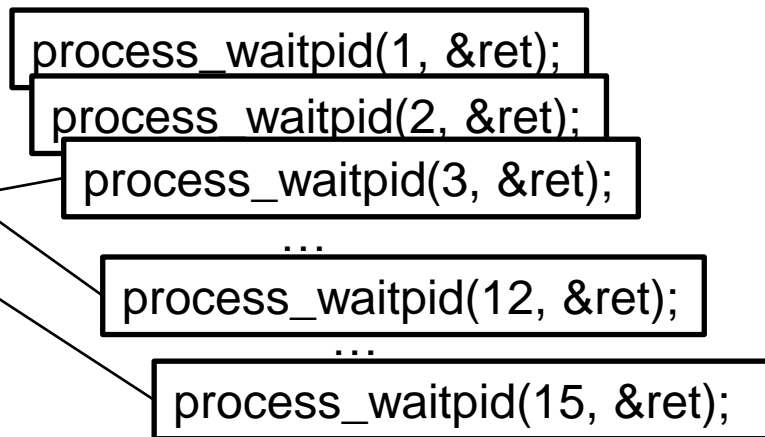
# 还有什么可以改进的地方呢？

- 考虑两个用户的场景（小明和小红）
  - 如果小红想偷看小明进程退出的状态，应该怎么做？
    - 暴力枚举可能的id，对它们都调用process\_waitpid

小明的进程（不可见）



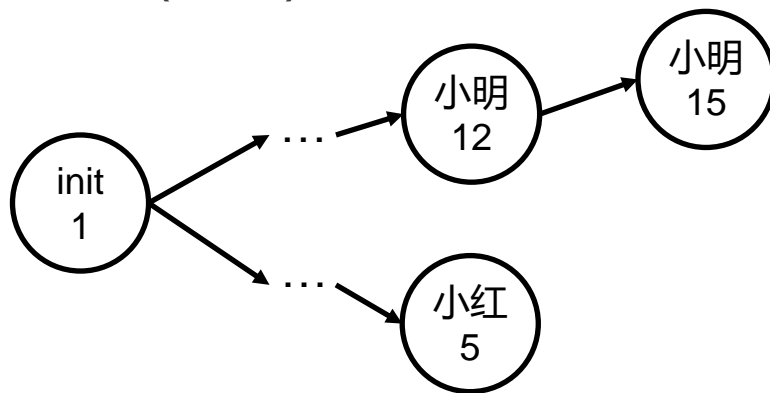
小红的程序



如何避免“偷看”？

# 改进：限制进程等待的范围

- **目标：只有创建某进程的程序才能监控它**
  - 程序属于小明，所以小红的程序不能监控
- **实现：引入父（创建者）子（被创建者）进程概念**
- **进程之间的创建关系构建了一棵进程树**
  - 第一个进程（树根）通常由内核主动创建



# 改进后的实现

- 分别修改PCB和process\_waitpid实现

## PCB: 维护子进程列表

```
1 struct process_v4 {  
2     // 处理器上下文  
3     struct context *ctx;  
4     // 虚拟地址空间 (包含页表基地址)  
5     struct vmSPACE *vmSPACE;  
6     // 内核栈  
7     void *stack;  
8     // 进程标识符  
9     int pid;  
10    // 退出状态  
11    int exit_status;  
12    // 标记是否退出  
13    bool is_exit;  
14    // 子进程列表  
15    pcb list *children;  
16 };
```

## waitpid: 扫描子进程列表而不是所有进程

```
1 void process_waitpid_v3(int id, int *status)  
2 {  
3     // 如果没有子进程, 直接返回  
4     if (!curr_proc->children)  
5         return;  
6     while (TRUE) {  
7         bool not_exist = TRUE;  
8         // 扫描子进程列表, 寻找对应进程  
9         for (struct process *proc : curr_proc->children) {  
10            if (proc->pid == id) {  
11                // 标记已找到对应进程, 并检查其是否已经退出  
12                not_exist = FALSE;  
13                if (proc->is_exit) {  
14                    // 若发现该进程已经退出, 记录其退出状态  
15                    *status = proc->exit_status;  
16                    // 回收进程的 PCB 并返回  
17                    destroy_process(proc);  
18                    return;  
19                } else {  
20                    // 如果没有退出, 则调度下个进程执行  
21                    schedule();  
22                }  
23            }  
24        }  
25        // 如果子进程列表中不存在该进程, 则立即退出  
26        if (not_exist)  
27            return;  
28    }  
29 }
```

# 一个关于process\_waitpid的疑问

- 在编程时，我们并未强制创建者调用waitpid
  - 如果shell不调用waitpid，会怎么样呢？
    - ls进程已退出，但并未销毁（有的进程活着，它已经死了）
      - 称为僵尸进程（zombie）

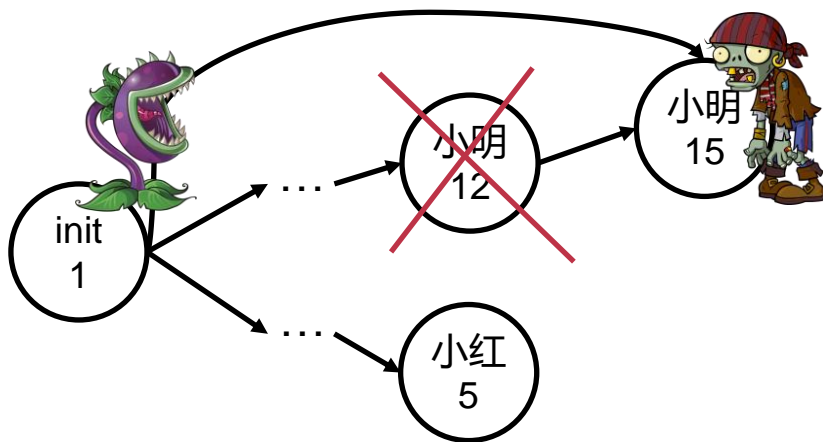
```
while (true) {  
    char *buffer = read_from_user();  
    if (strcmp(buffer, "/bin/ls") == 0) {  
        int id = process_create("/bin/ls");  
    }  
    else if ...  
    ...  
}
```

```
int main (...) {  
    // ls的逻辑  
    ...  
    process_exit(1);  
}
```



# 一个关于process\_waitpid的疑问

- 在编程时，我们并未强制创建者调用waitpid
  - 如果shell不调用waitpid，会怎么样呢？
    - 解决方案：当shell进程退出后，由init进程代管并回收



## 进程的状态

# 还有什么需求呢？睡眠

- “退出”事件太单一了，能不能等待其它的事件？
  - 常见的事件：“时间”（例如等10秒）
  - 因此可以引入**进程睡眠**
    - 不断查看时间，如果未到规定时间则继续等待

```
1 void process_sleep(int seconds)
2 {
3     // 获取当前时间作为睡眠起始时间
4     struct *date start_time = get_time();
5     while (TRUE) {
6         struct *date cur_time = get_time();
7         if (time_diff(cur_time, start_time) < seconds) {
8             // 如果时间未到，则调度下个进程执行
9             schedule();
10        } else {
11            // 时间已到，直接返回
12            return;
13        }
14    }
15 }
```



# 回顾代码：进程的执行状态

- 从之前的代码可以看出，进程处于不同的**执行状态**中
  - 状态有好几种，可以列出来，便于管理

```
1 void process_waitpid_v2(int id, int *status)
2 {
3     while (TRUE) {
4         bool not_exist = TRUE;
5         // 扫描内核维护的进程列表，寻找对应进程
6         for (struct process *proc : all_processes) {
7             if (proc->pid == id) {
8                 // 标记已找到对应进程，并检查其是否已经退出
9                 not_exist = FALSE;
10                if (proc->is_exit) {
11                    // 若发现该进程已经退出，记录其退出状态
12                    *status = proc->exit_status;
13                    // 回收进程的 PCB 并返回
14                    destroy_process(proc);
15                    return;
16                } else {
17                    // 如果没有退出，则调度下个进程执行
18                    schedule();
19                }
20            }
21        }
22        // 如果列表中不存在该进程，则直接返回
23        if (not_exist)
24            return;
25    }
26 }
```

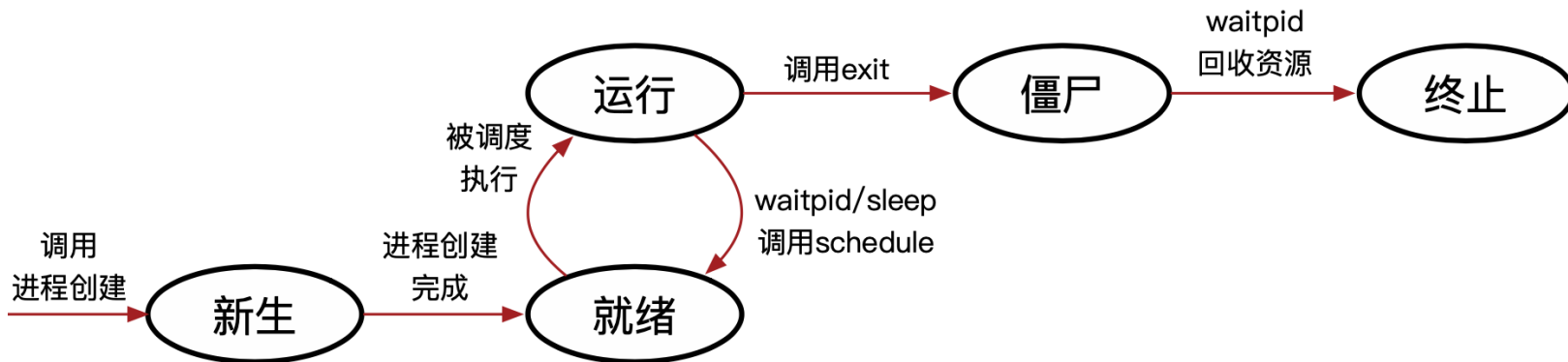
**僵尸状态**

**已销毁状态**

**即将进入不执行的状态**

# OS可以为进程设置不同的执行状态

- 新生 (new) : 刚调用process\_create
- 就绪 (ready) : 随时准备执行 (但暂时没有执行)
- 运行 (running) : 正在执行
- 僵尸 (zombie) : 退出但未回收
- 终止 (terminated) : 退出且被回收



# 进程执行状态的典型应用案例：调度

- 调度的目的：选出下一个**可以执行的进程**
  - 可以执行 = 就绪 (ready)

## PCB结构：增加执行状态

```
1 enum exec_status {NEW, READY, RUNNING,  
2                  ZOMBIE, TERMINATED};  
3  
4 struct process_v5 {  
5     // 处理器上下文  
6     struct context *ctx;  
7     // 虚拟地址空间 (包含页表基地址)  
8     struct vmSPACE *vmSPACE;  
9     // 内核栈  
10    void *stack;  
11    // 进程标识符  
12    int pid;  
13    // 退出状态  
14    int exit_status;  
15    // 子进程列表  
16    pcb_list *children;  
17    // 执行状态  
18    enum exec_status exec_status;  
19 };
```

## 调度逻辑片段：只选择状态为READY的进程

```
1 struct process* pick_next(void)  
2 {  
3     // 遍历进程列表，寻找下一个可调度 (处于就绪状态) 的进程  
4     for (struct process *proc : all_processes) {  
5         if (proc->exec_status == READY) {  
6             // 上一个正在运行的进程变为就绪  
7             curr_proc->exec_status = READY;  
8             // 选中的进程执行状态变为运行  
9             proc->exec_status = RUNNING;  
10            return proc;  
11        }  
12    }  
13 }
```

# ▶ LINUX进程创建接口

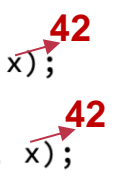
# 经典创建方法：fork()

- **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
  - 父进程和子进程会共享部分数据结构（例如文件描述符）

# 在程序中使用fork

- 创建一模一样的拷贝
  - 例子中父子进程中的x均为42

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main(void)
6 {
7     int x = 42;
8     int ret = fork();
9     if (ret == 0) {
10         // 子进程
11         printf("child: x=%d\n", x);
12     } else {
13         // 父进程
14         printf("parent: x=%d\n", x);
15     }
16     return 0;
17 }
```



# 在程序中使用fork

- 使用fork的返回值来分辨父/子进程
  - 0: 子进程
  - 非0 (子进程id) : 父进程

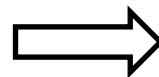
```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main(void)
6 {
7     int x = 42;
8     int ret = fork();
9     if (ret == 0) {
10         // 子进程
11         printf("child: x=%d\n", x);
12     } else {
13         // 父进程
14         printf("parent: x=%d\n", x);
15     }
16     return 0;
17 }
```

# 在程序中使用fork

- 独立执行

- Fork后父子进程顺序不确定，视调度策略而定

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main(void)
6 {
7     int x = 42;
8     int ret = fork();
9     if (ret == 0) {
10         // 子进程
11         printf("child: x=%d\n", x);
12     } else {
13         // 父进程
14         printf("parent: x=%d\n", x);
15     }
16     return 0;
17 }
```



## 两种可能结果

child: x=42  
parent: x=42

parent: x=42  
child: x=42



## 小知识点：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
- **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



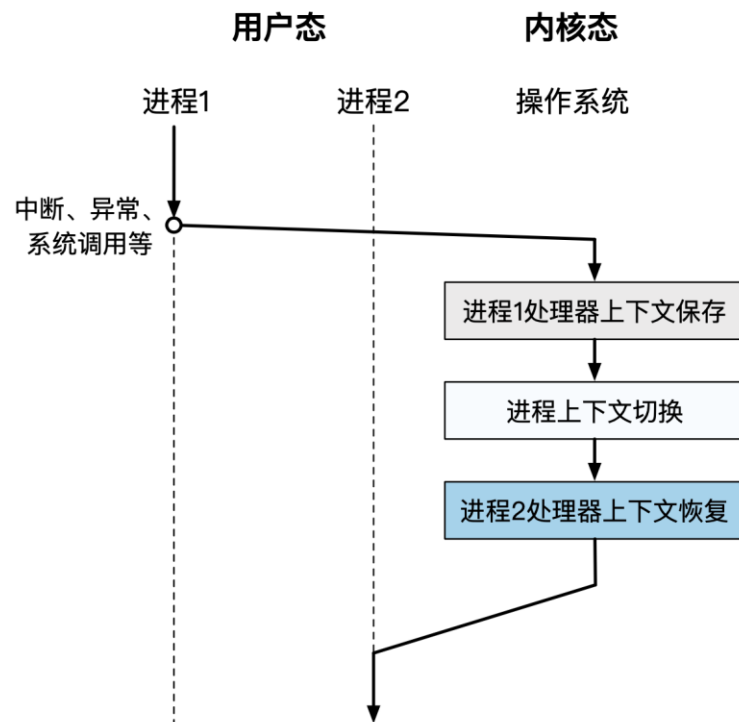
# 进程切换

基本步骤 – 处理器上下文结构 – 切换过程

# 进程切换的基本步骤

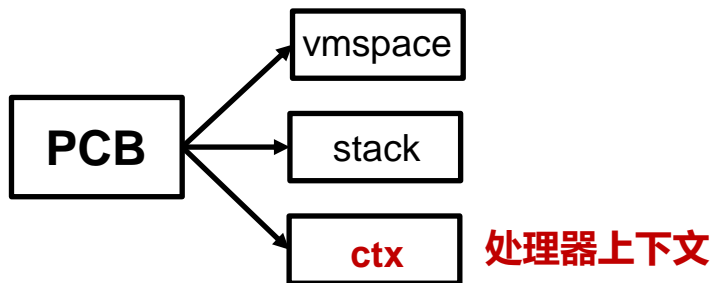
- 需要进出内核，共包含五个步骤

1. 进程1进入内核态
2. 进程1**处理器上下文**保存
3. **进程上下文**切换
4. 进程2**处理器上下文**恢复
5. 进程2返回用户态



# 处理器上下文 vs. 进程上下文

- 有两个“上下文”概念，如何辨析？
  - 处理器上下文：用于保存切换时的寄存器状态（硬件）
    - 在每个PCB中均有保存
  - 进程上下文：表示目前操作系统正以哪个进程的身份运行（软件）
    - 使用一个指向PCB的全局指针（代码中的curr\_proc）



```
1 void process_exit_v2(int status)
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->ctx);
5     // 销毁内核栈
6     destroy_kern_stack(curr_proc->stack);
7     // 销毁虚拟地址空间
8     destroy_vmSPACE(curr_proc->vmSPACE);
9     // 保存退出状态
10    curr_proc->exit_status = status;
11    // 标记进程为退出状态
12    curr_proc->is_exit = TRUE;
13    // 告知内核选择下个需要执行的进程
14    schedule();
15 }
```

进程上下文

# 小思考：操作系统如何实现curr\_proc

- 内核数据
  - 全局变量（固定位置）
- 多个CPU核心
  - 需要维护多个curr\_proc
  - 如何实现呢？

# 回顾：处理器上下文的结构

- 哪些寄存器是需要保存的？
  - 通用寄存器：所有 (X0-X30)
  - 特殊寄存器：SP\_EL0 (栈寄存器)
  - 系统寄存器：ELR\_EL1 (对应PC)，SPSR\_EL1 (对应PSTATE)

---

```
1 // 进程处理器上下文内部包含的内容
2 struct context {
3     // 通用寄存器
4     u64 x0, x1, ..., x30;
5     // 特殊寄存器
6     u64 sp_el0;
7     // 系统寄存器
8     u64 elr_el1, spsr_el1;
9 };
```

---

# 进程的切换节点

- 所有调用schedule()的地方都涉及进程切换

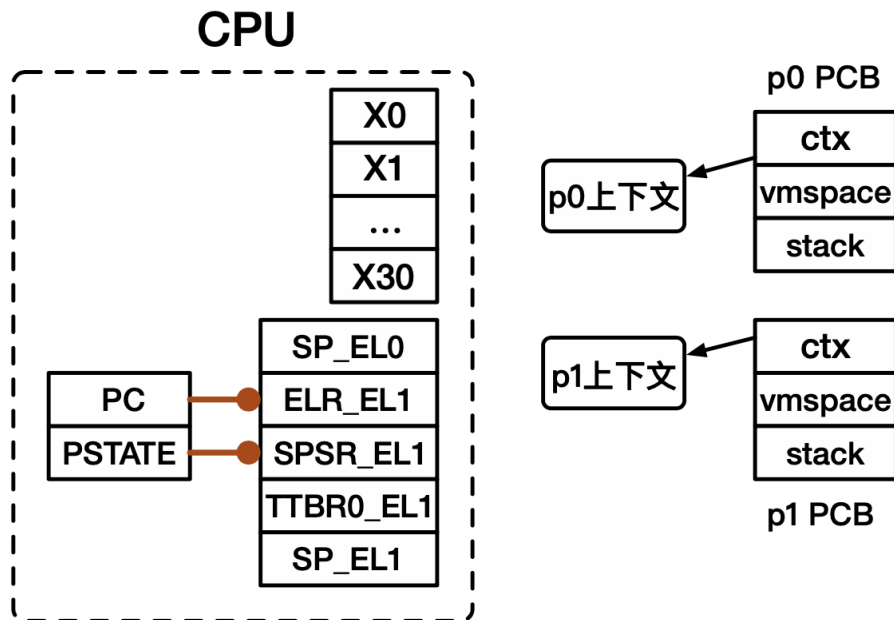
```
1 void process_exit_v1(void
2 {
3     // 销毁上下文结构
4     destroy_ctx(curr_proc->
5     // 销毁内核栈
6     destroy_kern_stack(curr
7     // 销毁虚拟地址空间
8     destroy_vmspace(curr_pr
9     // 销毁 PCB
10    destroy_process(curr_pr
11    // 告知内核选择下个需要执行
12    schedule();
13 }
```

```
1 void process_sleep(int seconds)
2 {
3     // 获取当前时间作为睡眠起始时间
4     struct *date start_time = get_
5     while (TRUE) {
6         struct *date cur_time = get_
7         if (time_diff(cur_time, star
8             // 如果时间未到, 则调度下个进程
9             schedule();
10        } else {
11            // 时间已到, 直接返回
12            return;
13        }
14    }
15 }
```

```
1 void process_waitpid_v3(int id, int *status)
2 {
3     // 如果没有子进程, 直接返回
4     if (!curr_proc->children)
5         return;
6     while (TRUE) {
7         bool not_exist = TRUE;
8         // 扫描子进程列表, 寻找对应进程
9         for (struct process *proc : curr_proc->children) {
10             if (proc->pid == id) {
11                 // 标记已找到对应进程, 并检查其是否已经退出
12                 not_exist = FALSE;
13                 if (proc->is_exit) {
14                     // 若发现该进程已经退出, 记录其退出状态
15                     *status = proc->exit_status;
16                     // 回收进程的 PCB 并返回
17                     destroy_process(proc);
18                     return;
19                 } else {
20                     // 如果没有退出, 则调度下个进程执行
21                     schedule();
22                 }
23             }
24         }
25         // 如果子进程列表中不存在该进程, 则立即退出
26         if (not_exist)
27             return;
28     }
29 }
```

# 进程切换的全过程（1）：p0进入内核态

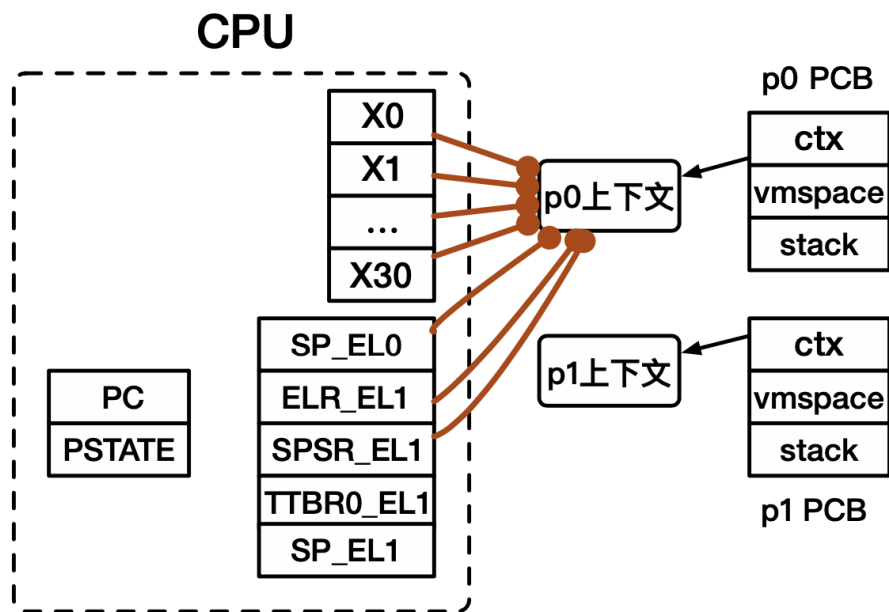
- 由硬件完成部分寄存器保存
  - PC和PSTATE分别自动保存到ELR\_EL1和SPSR\_EL1





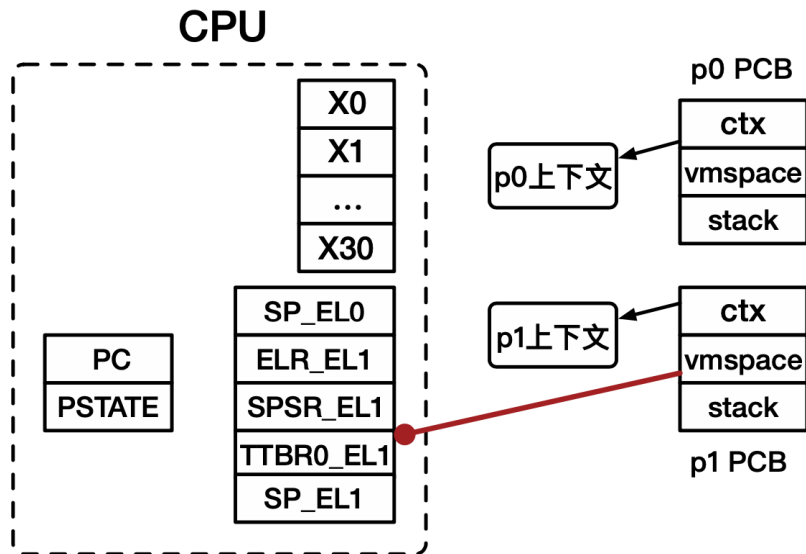
## 进程切换的全过程（2）：p0处理器上下文保存

- 将处理器中的寄存器值保存到处理器上下文对应的位置



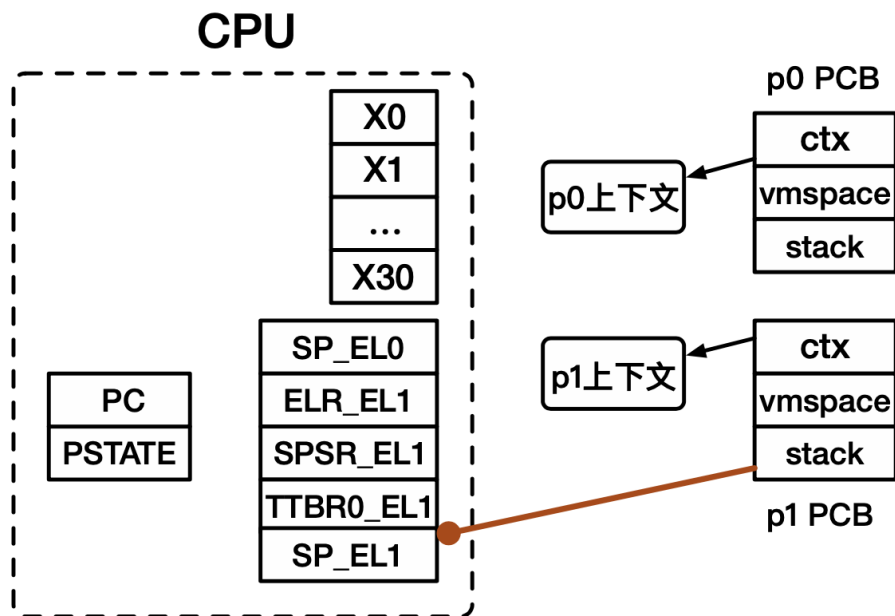
# 进程切换的全过程（3.1）：虚拟地址空间切换

- 设置页表相关寄存器（TTBR0\_EL1）
  - 使用PCB中保存的页表基地址赋值给TTBR0\_EL1



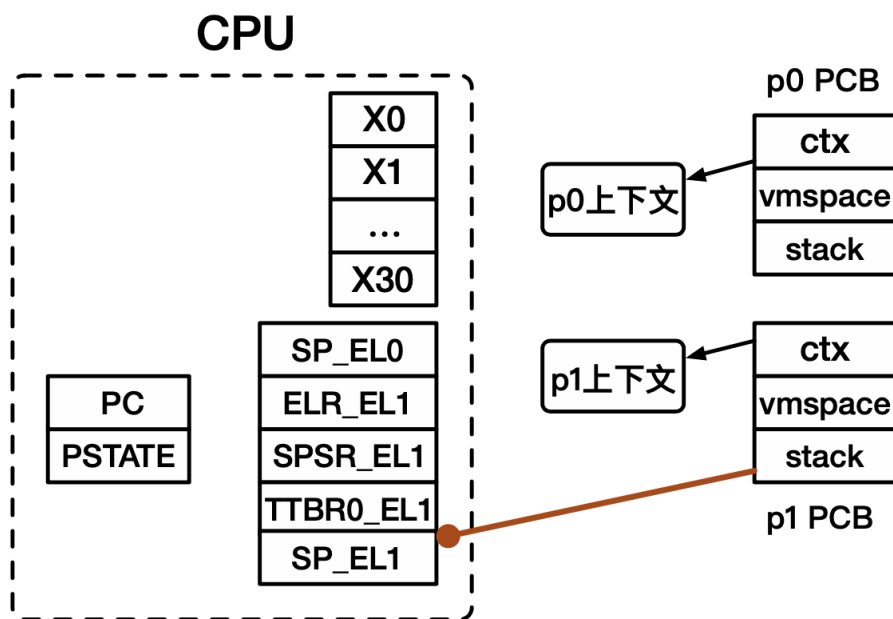
# 进程切换的全过程（3.2）：内核栈切换

- 设置内核中的栈寄存器SP\_EL1
  - 使用PCB中保存的内核栈顶地址赋值给SP\_EL1



# 进程切换的全过程 (3.3) : 进程上下文切换

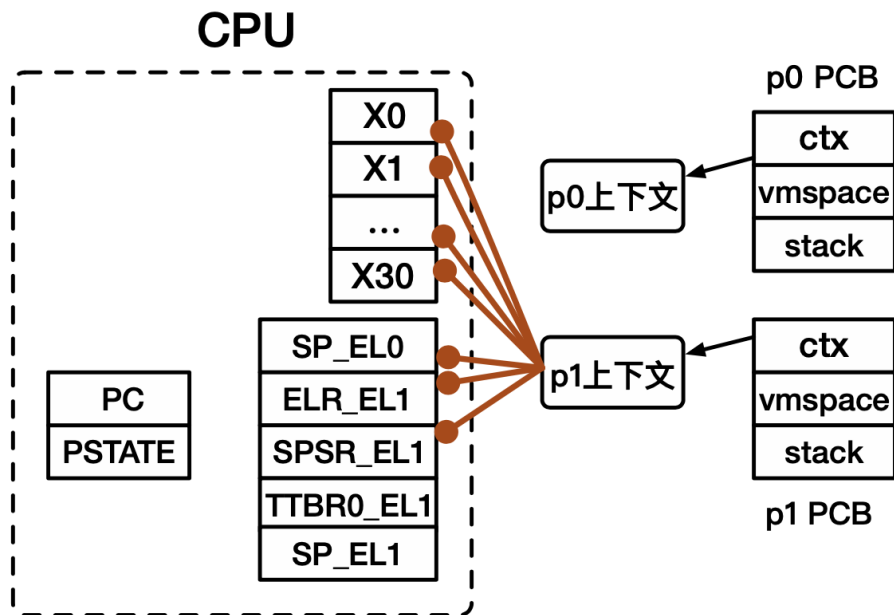
- 设置cur\_proc为之后要执行的进程 (p1)
  - 表明之后操作系统将以p1的身份运行



cur\_proc = p1

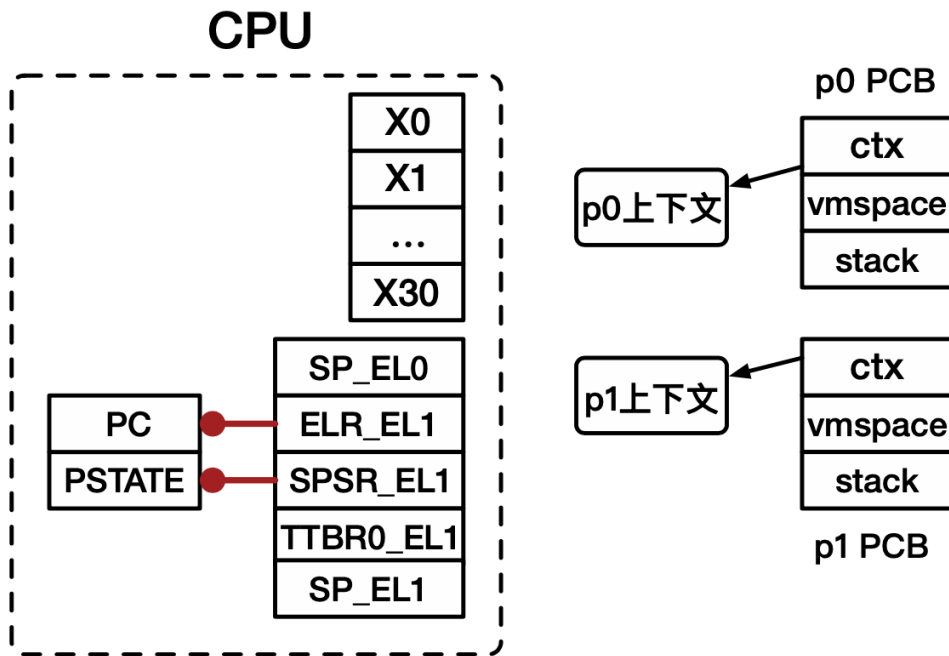
## 进程切换的全过程（4）：p1处理器上下文恢复

- 从处理器上下文中加载各寄存器的值，放入对应寄存器中



## 进程切换的全过程（5）：p1处理器上下文恢复

- 由硬件自动恢复部分寄存器
  - 将ELR\_EL1和SPSR\_EL1中的值自动保存到PC和PSTATE中



# 小结

- **PCB：表示进程的数据结构**
  - 针对不同需求，PCB的设计
- **进程生命周期**
  - 创建、等待、退出、回收
  - 进程状态
- **进程切换**
  - 处理器上下文
  - 进程上下文