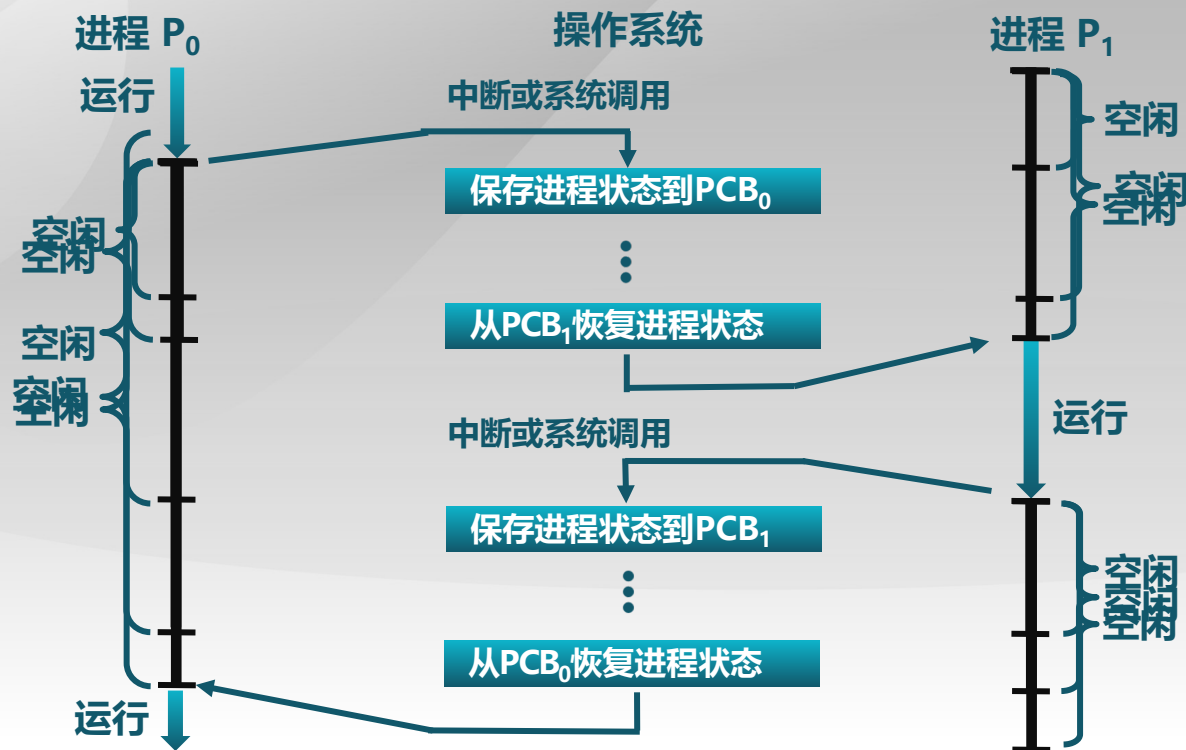


# 进程切换

- 进程切换(上下文切换)
  - ▣ 暂停当前运行进程，从运行状态变成其他状态
  - ▣ 调度另一个进程从就绪状态变成运行状态
- 进程切换的要求
  - ▣ 切换前，保存进程上下文
  - ▣ 切换后，恢复进程上下文
  - ▣ 快速切换
- 进程生命周期的信息
  - ▣ 寄存器 (PC, SP, ...)
  - ▣ CPU状态
  - ▣ 内存地址空间

# 上下文切换图示



# 进程控制块PCB:内核的进程状态记录

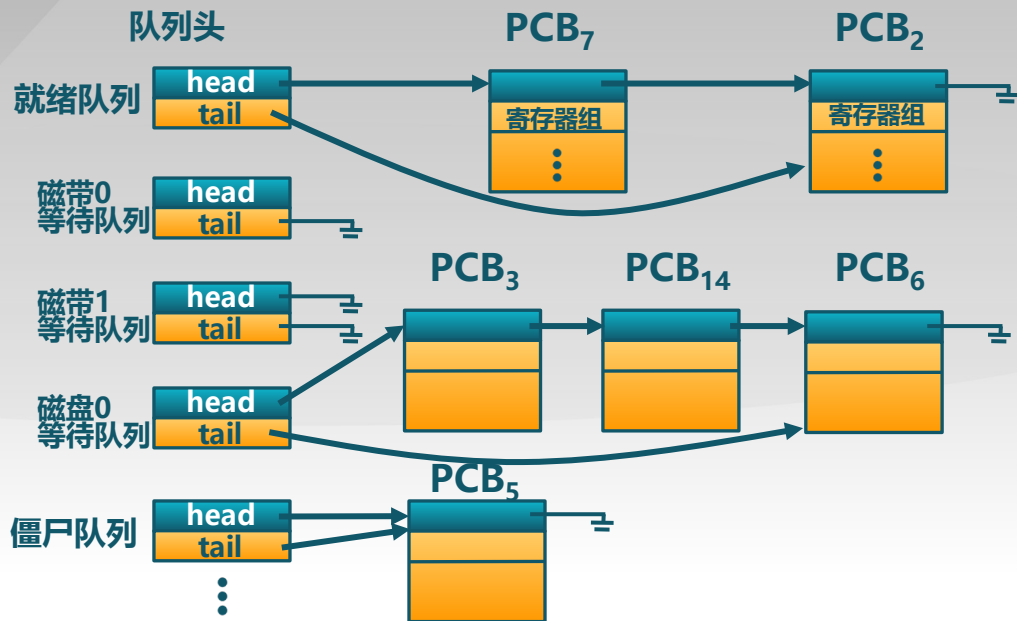
- 内核为每个进程维护了对应的进程控制块 (PCB)
- 内核将相同状态的进程的PCB放在同一队列

## ▶ 就绪队列

## ▶ I/O等待队列

- ▶ 每个设备一个队列

## ▶ 僵尸队列



# ucore的进程控制块结构proc\_struct

char name[PROC\_NAME\_LEN + 1]

int pid

int runs

struct proc\_struct \*parent

uintptr\_t kstack

struct mm\_struct \*mm

list\_entry\_t hash\_link

char name[PROC\_NAME\_LEN + 1]

struct proc\_struct

kern-ucore/process/proc.h

uint32\_t flags

uintptr\_t cr3

enum proc\_state state

volatile bool need\_resched

struct trapframe \*tf

struct context context

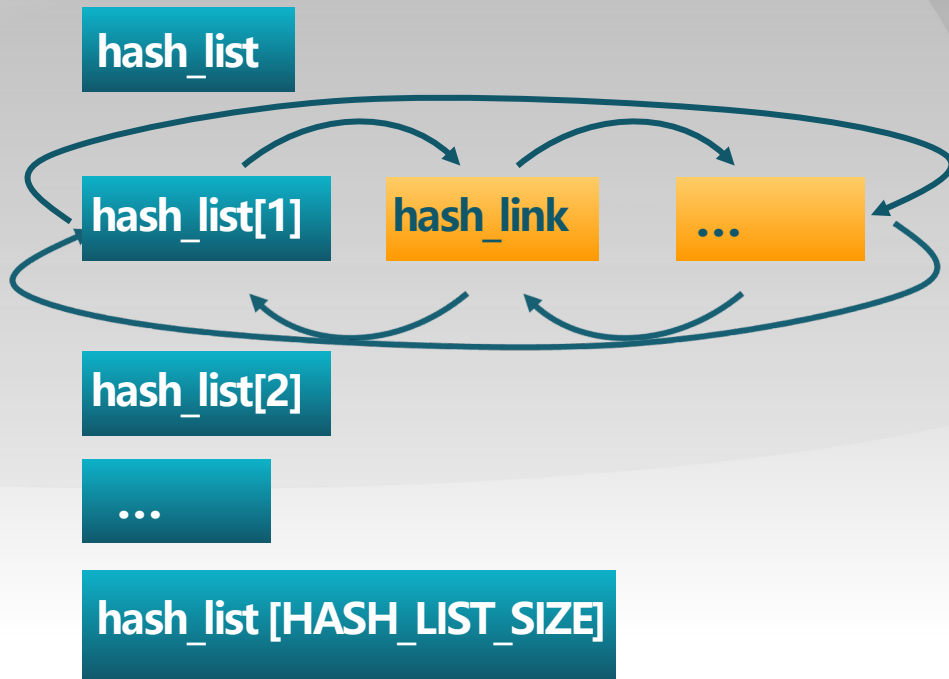
# ucore的内存地址空间结构mm\_struct

/kern-ucore/mm/vmm.h

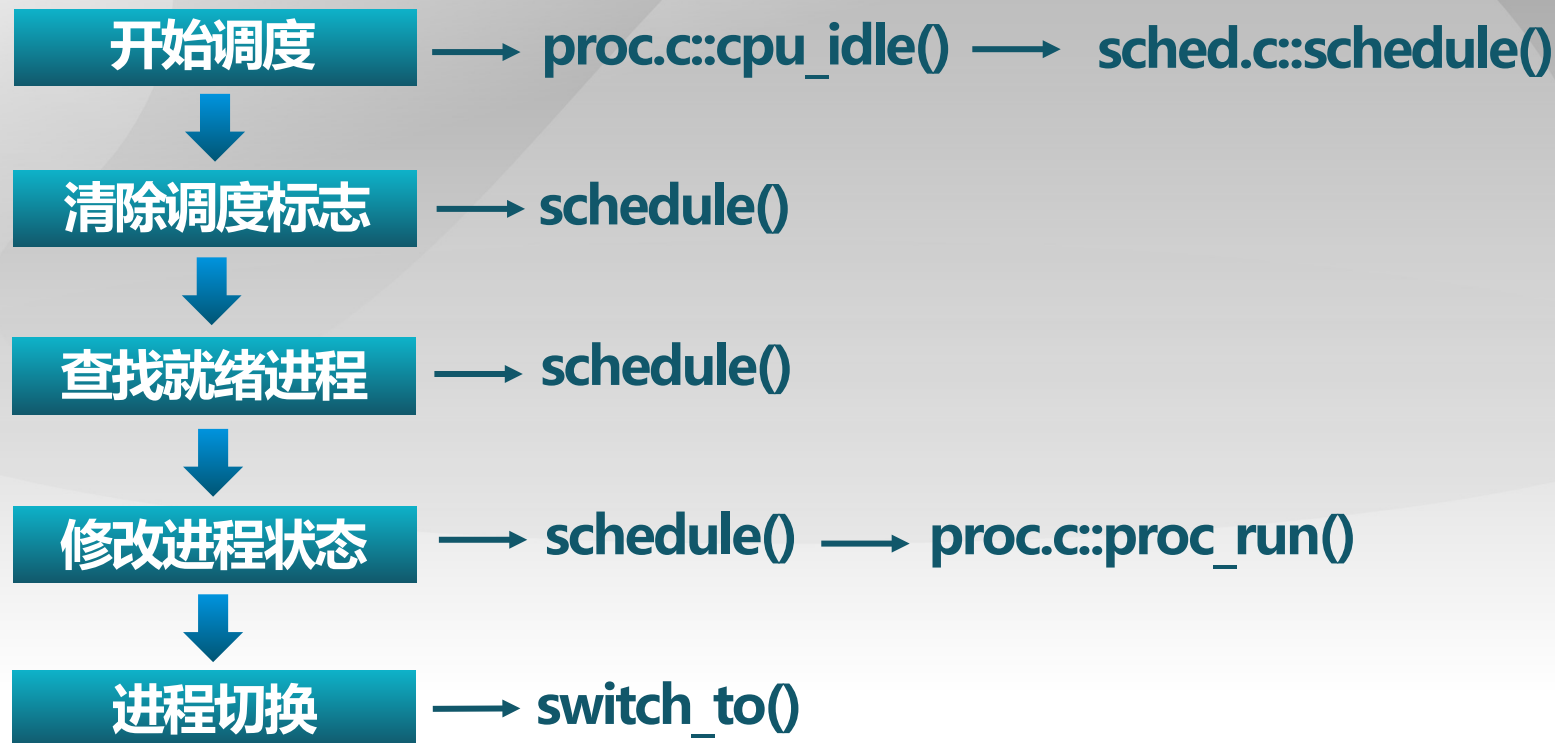
```
struct mm_struct {  
    // linear list link which sorted by start addr of vma  
    list_entry_t mmap_list;  
    // current accessed vma, used for speed purpose  
    struct vma_struct *mmap_cache;  
    pde_t *pgdir; // the PDT of these vma =cr3=boot_cr3  
    int map_count; // the count of these vma  
    void *sm_priv; // the private data for swap manager  
};
```

# ucore+ 的进程队列

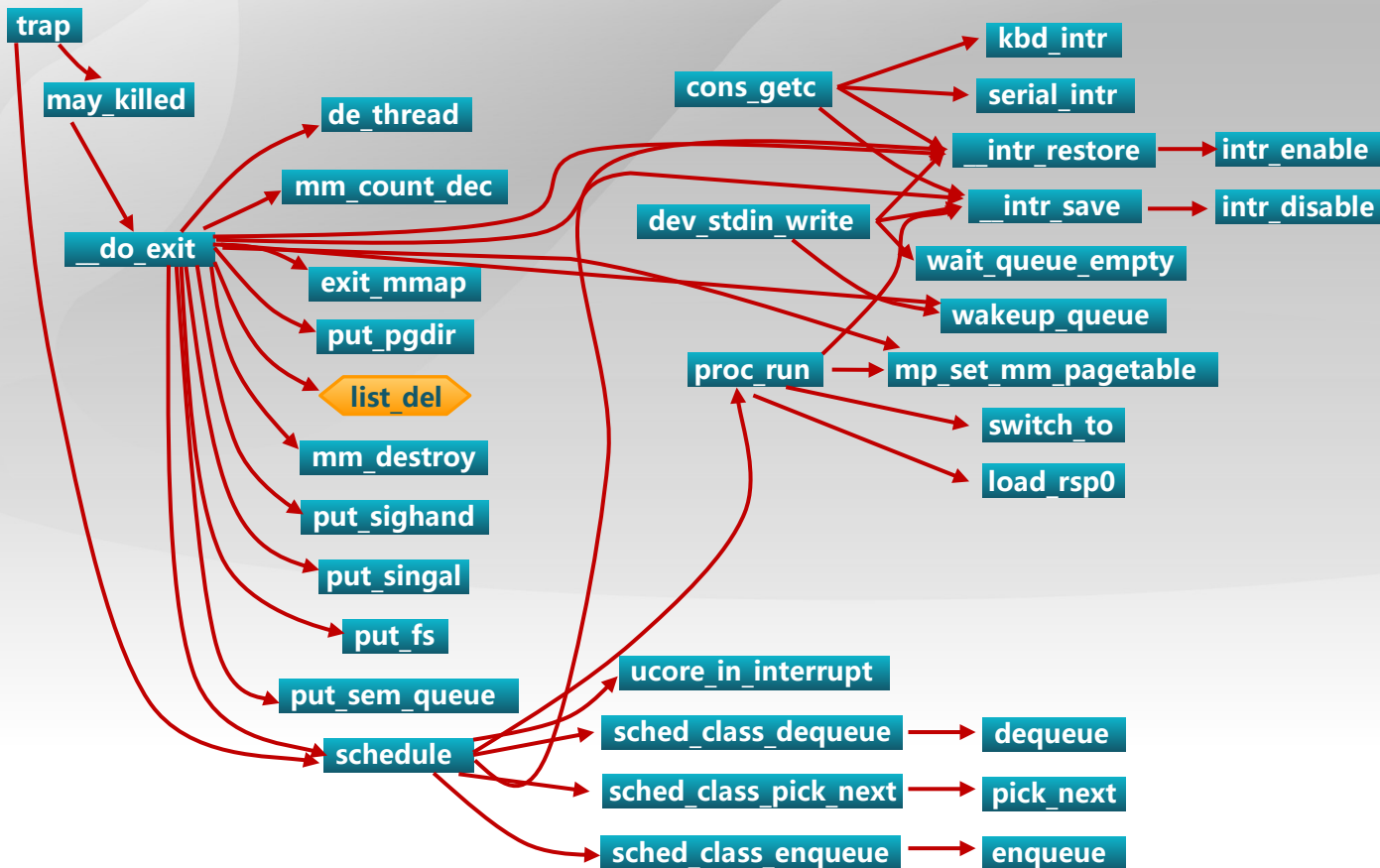
/ucore/src/kern-ucore/process/proc.h



# ucore + 进程/线程切换流程



# ucore+ 的进程切换





# switch\_to的实现

kern-ucore/arch/i386/process/switch.S

```
.text
.globl switch_to
switch_to:      # switch_to(from, to)
    # save from's registers
    movl 4(%esp), %eax    # eax points to from
    popl 0(%eax)         # save eip !popl
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)
    # restore to's registers
    movl 4(%esp), %eax    # not 8(%esp): popped return address already
                        # eax now points to to
    movl 28(%eax), %ebp
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp
    pushl 0(%eax)        # push eip
    ret
```



# 操作系统

Operating System

# 创建新进程

- Windows进程创建API: `CreateProcess(filename)`
- Unix进程创建系统调用: `fork/exec`
  - ▣ `fork()`把一个进程复制成二个进程
    - ▣ parent (old PID), child (new PID)
  - ▣ `exec()`用新程序来重写当前进程
    - ▣ PID没有改变
  - ▣ 创建时关闭所有在子进程里的文件描述符  
`CreateProcess(filename, CLOSE_FD)`
  - ▣ 创建时改变子进程的环境  
`CreateProcess(filename, CLOSE_FD, new_envp)`
  - ▣ 等等

# 创建新进程

- 用fork和exec创建进程的示例

```
int pid = fork();           // 创建子进程
if(pid == 0) {              // 子进程在这里继续
    // Do anything (unmap memory, close net connections...)
    exec( "program" , argc, argv0, argv1, ...);
}
```

- fork() 创建一个继承的子进程

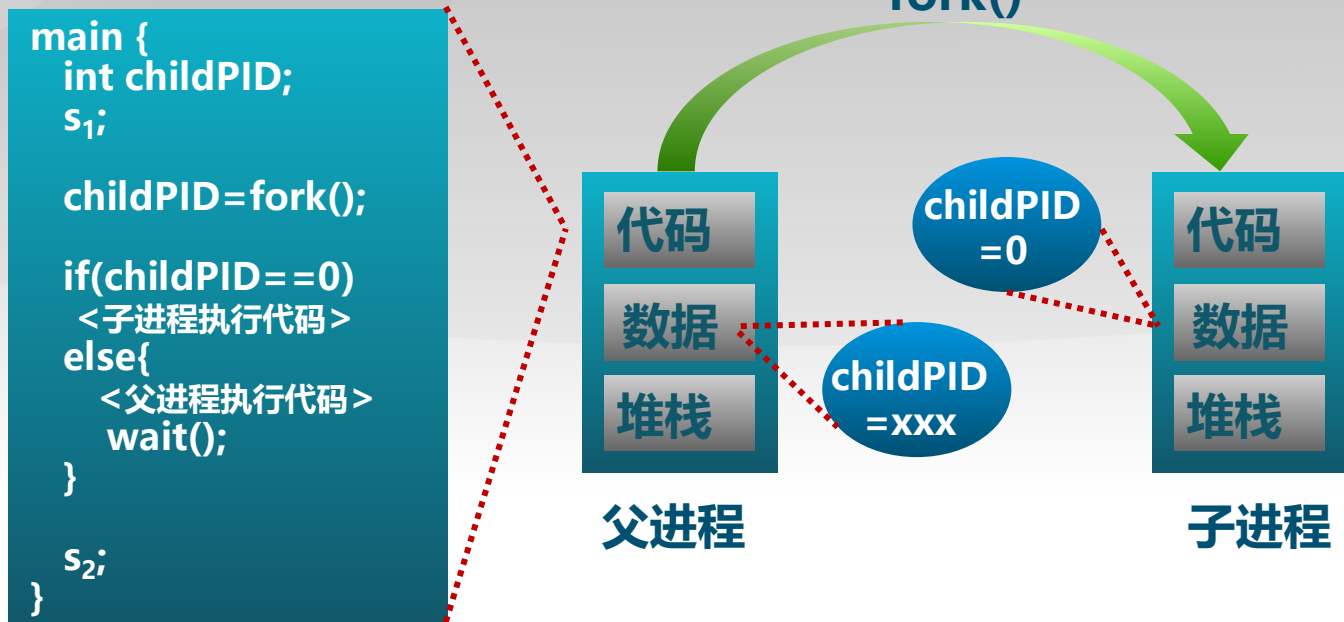
- 复制父进程的所有变量和内存
- 复制父进程的所有CPU寄存器(有一个寄存器例外)

- fork()的返回值

- 子进程的fork()返回0
- 父进程的fork()返回子进程标识符
- fork() 返回值可方便后续使用，子进程可使用getpid()获取PID

# fork()的地址空间复制

- fork()执行过程对于子进程而言，是在调用时间对父进程地址空间的一次复制
  - 对于父进程fork() 返回child PID, 对于子进程返回值为0



# 程序加载和执行

系统调用exec( )加载新程序取代当前运行进程

exec()示例代码

main()

...

int pid = fork();

// 创建子进程

if (pid == 0) {

// 子进程在这里继续

exec\_status = exec( "calc" , argc, argv0, argv1, ...);

printf( "Why would I execute?" );

} else {

// 父进程在这里继续

printf( "Whose your daddy?" );

...

child\_status = wait(pid);

}

if (pid < 0) { /\* error occurred \*/

# 在shell中调用fork()后加载计算器的图示

```
int pid = fork();  
if(pid == 0) {  
    exec( "/bin/calc" );  
} else {  
    wait(pid);  
}
```

用户态

```
int calc_main(){  
    int q = 7;  
    do_init();  
    ln = get_input();  
    exec_in(ln);  
}
```

内核态

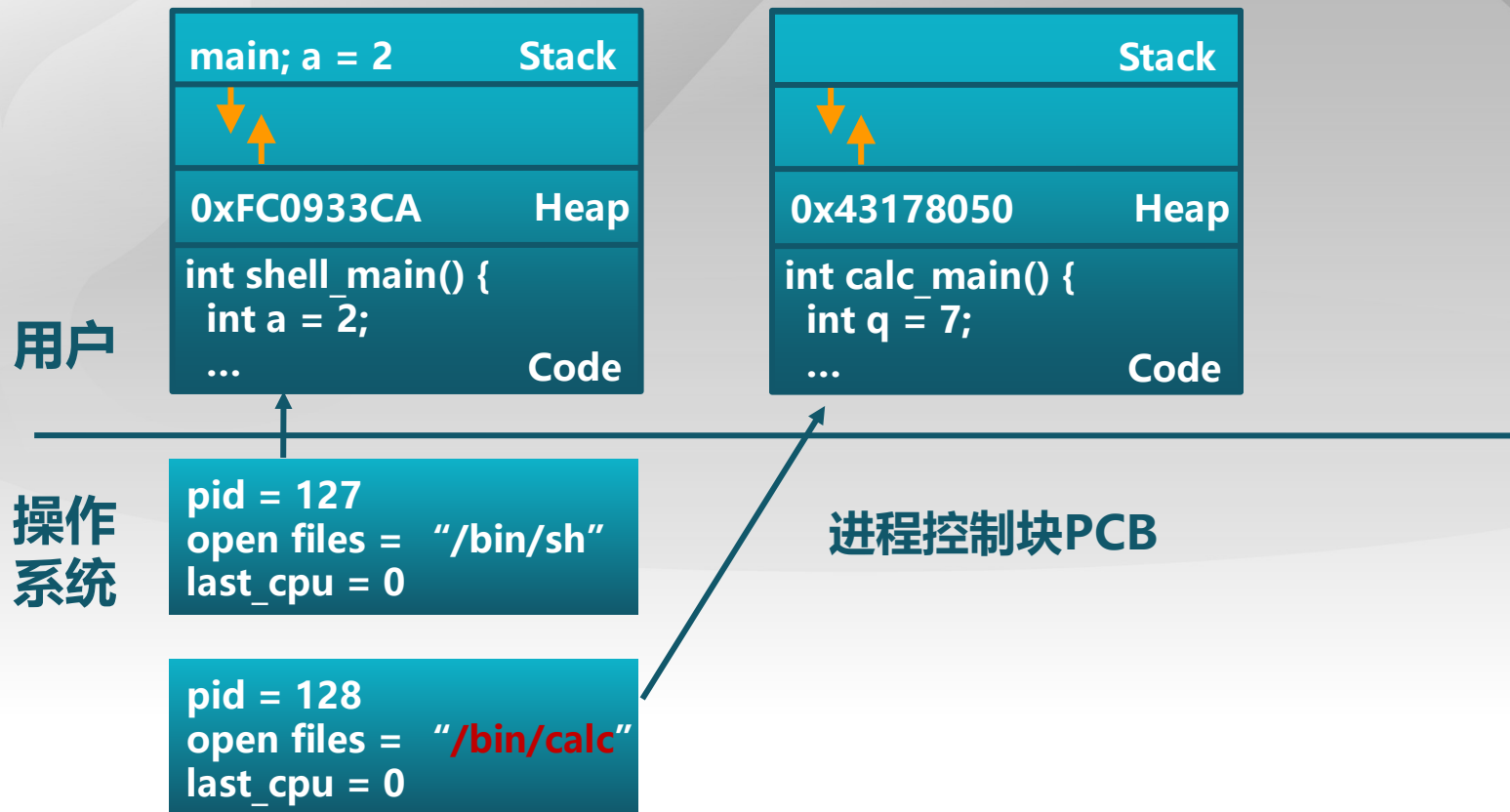
```
pid = 127  
open files = "/bin/sh"  
last_cpu = 0
```

```
pid = 128  
open files = "/bin/calc"  
last_cpu = 0
```

进程控制块PCB



# 在shell中调用fork()后加载计算器的图示



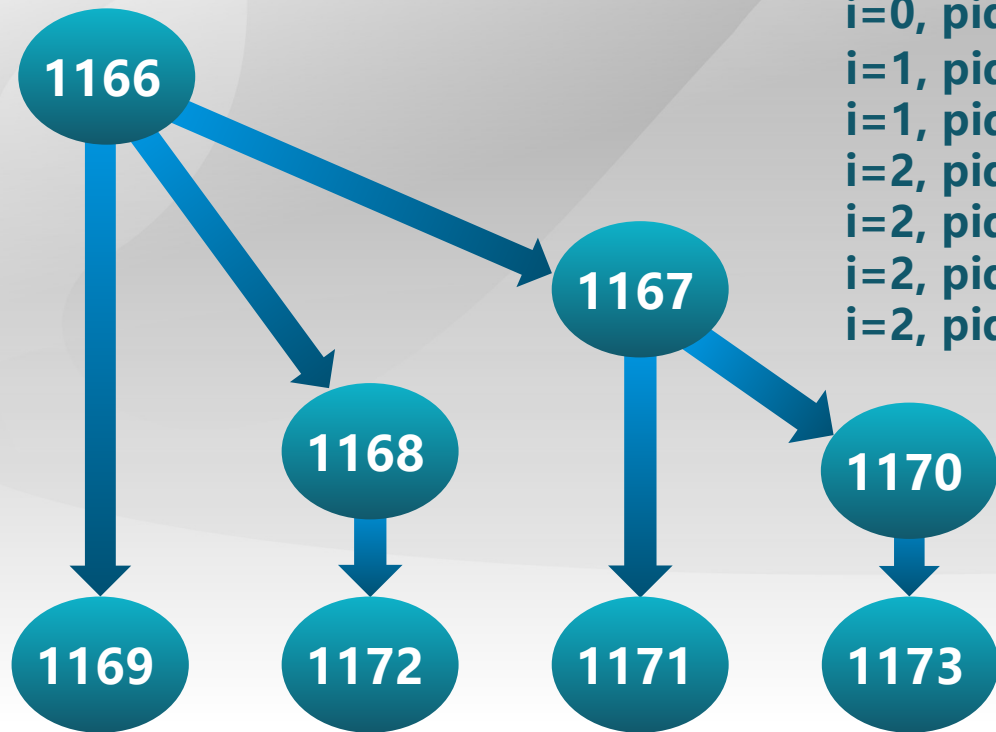


# fork()使用示例

```
int main()
{
    pid_t pid;
    int i;

    for (i=0; i<LOOP; i++)
    {
        /* fork another process */
        pid = fork();
        if (pid < 0) { /*error occurred */
            fprintf(stderr, "Fork Failed" );
            exit(-1);
        }
        else if (pid == 0) { /* child process */
            fprintf(stdout, "i=%d, pid=%d, parent pid=%d\n" ,i,
                getpid() ,getppid());
        }
    }
    wait(NULL);
    exit(0);
}
```

# fork()使用示例?



i=0, pid=1167, parent pid=1166

i=1, pid=1168, parent pid=1166

i=1, pid=1170, parent pid=1167

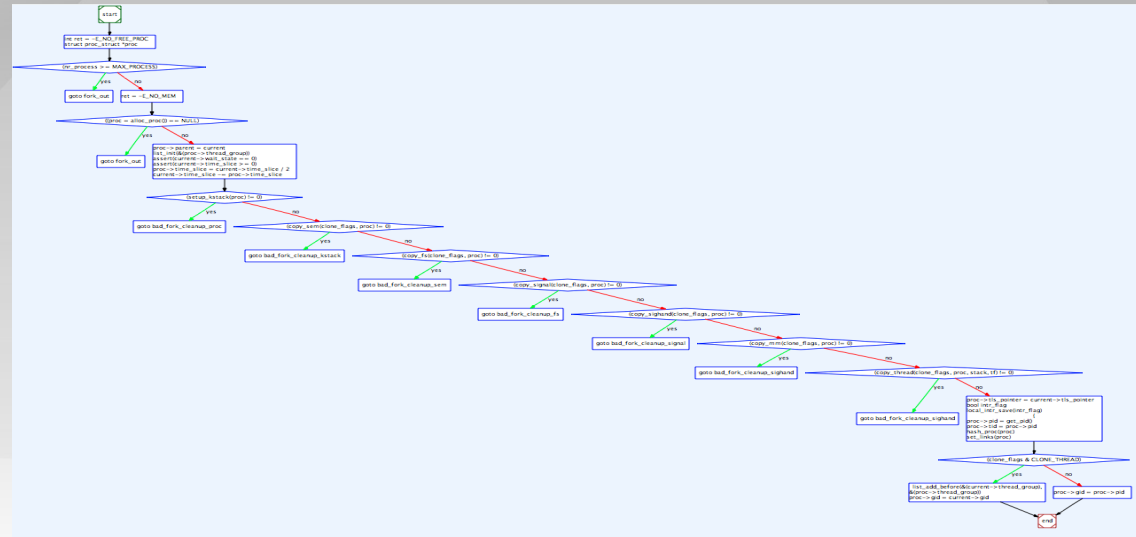
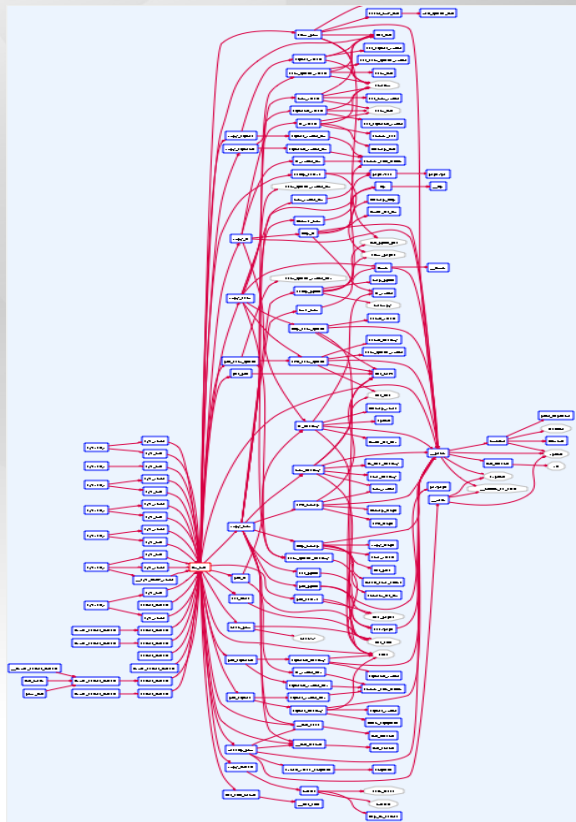
i=2, pid=1169, parent pid=1166

i=2, pid=1172, parent pid=1168

i=2, pid=1171, parent pid=1167

i=2, pid=1173, parent pid=1170

## ucore中fork()的实现



# 空闲进程的创建

\kern-ucore/process/proc.c

**idleproc**



分配idleproc需要的资源



初始化idleproc进程控制块

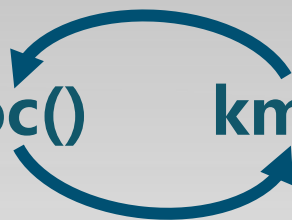


完成idleproc的初始化

proc\_init()

alloc\_proc()

kmalloc()



alloc\_proc()

proc\_init()

# 创建第一个内核线程

\kern-ucore/process/proc.c

**initproc**

初始化trapframe

初始化initproc

初始化内核堆栈

内存共享

把initproc放到就绪队列

唤醒 initproc

proc\_init()

kernel\_thread()tf

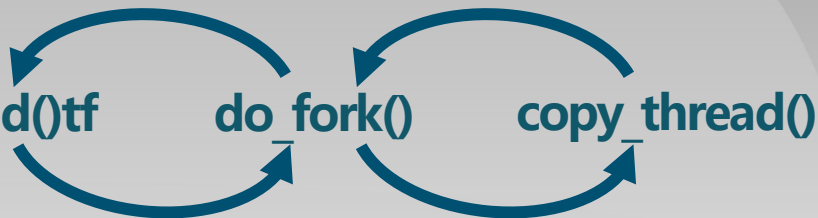
alloc\_proc()

setup\_stack()

copy\_stack()

do fork()

copy\_thread()



# Fork()的开销?

- fork()的实现开销
  - ▣ 对子进程分配内存
  - ▣ 复制父进程的内存和CPU寄存器到子进程里
  - ▣ **开销昂贵!!**
- 在99%的情况里，我们在调用fork()之后调用exec()
  - ▣ 在fork()操作中内存复制是没有作用的
  - ▣ 子进程将可能关闭打开的文件和连接
  - ▣ 为什么不能结合它们在一个调用中?
- vfork()
  - ▣ 创建进程时，不再创建一个同样的内存映像
  - ▣ 一些时候称为轻量级fork()
  - ▣ 子进程应该几乎立即调用exec()
  - ▣ 现在使用 Copy on Write (COW) 技术



# 操作系统

Operating Systems

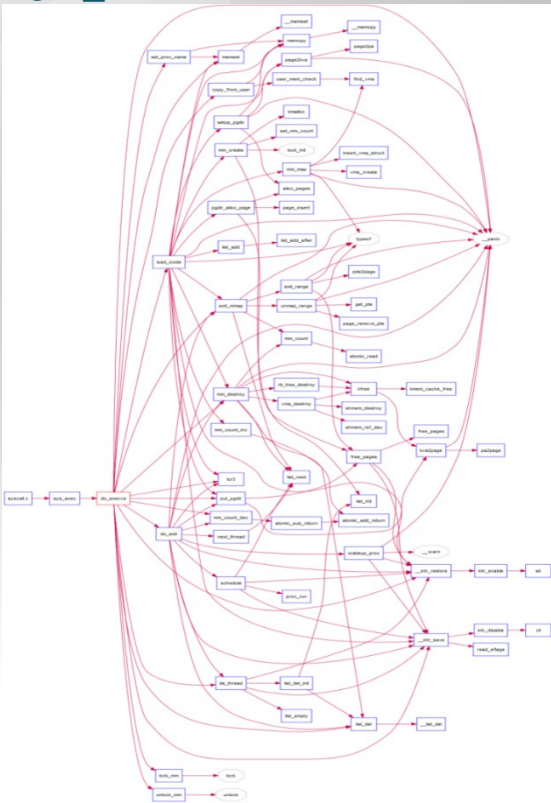
# 程序加载和执行系统调用exec()

- 允许进程“加载”一个完全不同的程序，并从main开始执行(即\_start)
- 允许进程加载时指定启动参数(argc, argv)
- exec调用成功时
  - ▣ 它是相同的进程...
  - ▣ 但是运行了不同的程序
- 代码段、堆栈和堆(heap)等完全重写

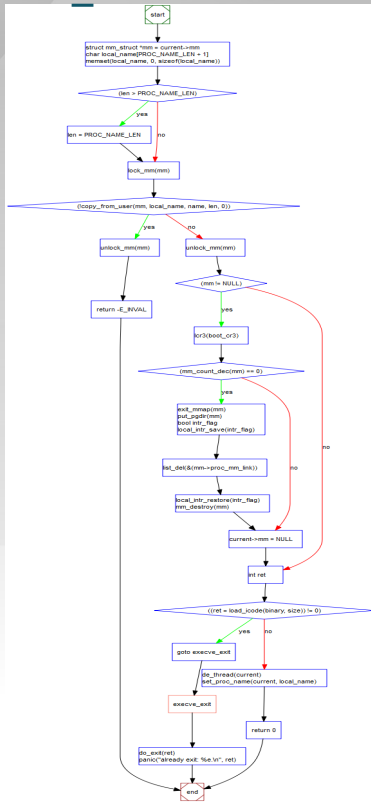


## ucore中的exec()实现

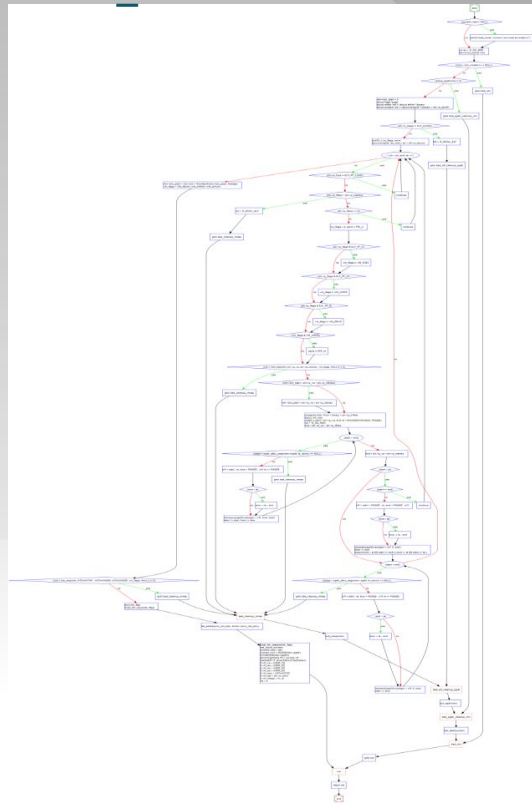
## sys\_exec



do exerce

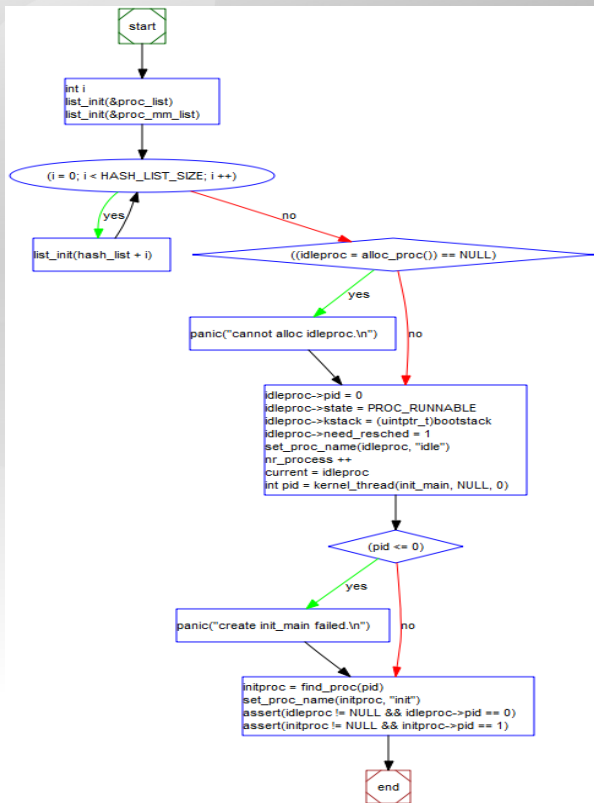


# load icode

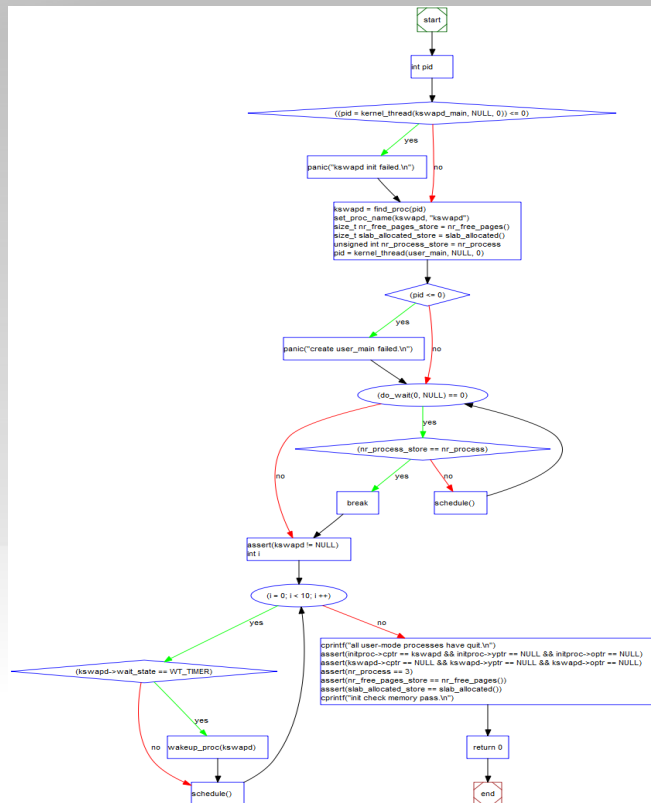


# ucore中第一个进程

## proc\_init



## init\_main





# 操作系统

Operating Systems

# 父进程等待子进程

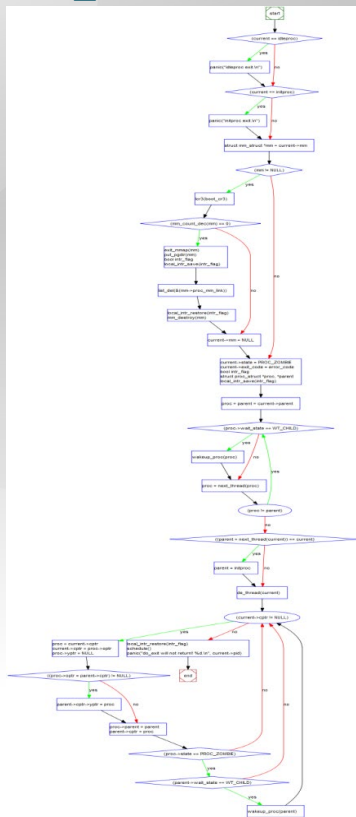
- **wait()系统调用用于父进程等待子进程的结束**
  - ▣ 子进程结束时通过exit()向父进程返回一个值
  - ▣ 父进程通过wait()接受并处理返回值
- **wait()系统调用的功能**
  - ▣ 有子进程存活时，父进程进入等待状态，等待子进程的返回结果  
当某子进程调用exit()时,唤醒父进程，将exit()返回值作为父进程中wait的返回值
  - ▣ 有僵尸子进程等待时，wait()立即返回其中一个值
  - ▣ 无子进程存活时，wait()立刻返回

# 进程的有序终止 exit()

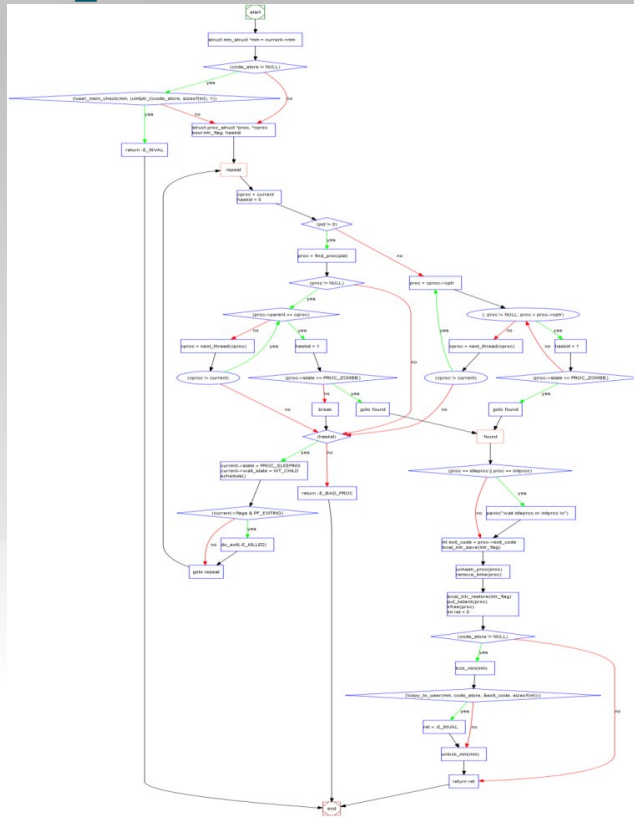
- 进程结束执行时调用**exit()**，完成进程资源回收
- exit()系统调用的功能
  - ▣ 将调用参数作为进程的“结果”
  - ▣ 关闭所有打开的文件等占用资源
  - ▣ 释放内存
  - ▣ 释放大部分进程相关的内核数据结构
  - ▣ 检查是否父进程是存活着的
    - ▣ 如存活，保留结果的值直到父进程需要它，进入僵尸(zombie/defunct)状态
    - ▣ 如果没有，它释放所有的数据结构，进程结果
  - ▣ 清理所有等待的僵尸进程
- 进程终止是最终的垃圾收集(资源回收)

## ucore中的exit /wait

## do\_exit



## do wait



# 其他进程控制系统调用

- 优先级控制
  - ▣ nice()指定进程的初始优先级
  - ▣ Unix系统中进程优先级会随执行时间而衰减
- 进程调试支持
  - ▣ ptrace()允许一个进程控制另一个进程的执行
  - ▣ 设置断点和查看寄存器等
- 定时
  - ▣ sleep()可以让进程在定时器的等待队列中等待指定

# 进程控制 v.s. 进程状态

