阅码场

YOMOCODE

Linux进 程 、 线 程 和 调 度(1)

麦当劳喜欢您来，喜欢您再来

第 一 次 课 大 纲

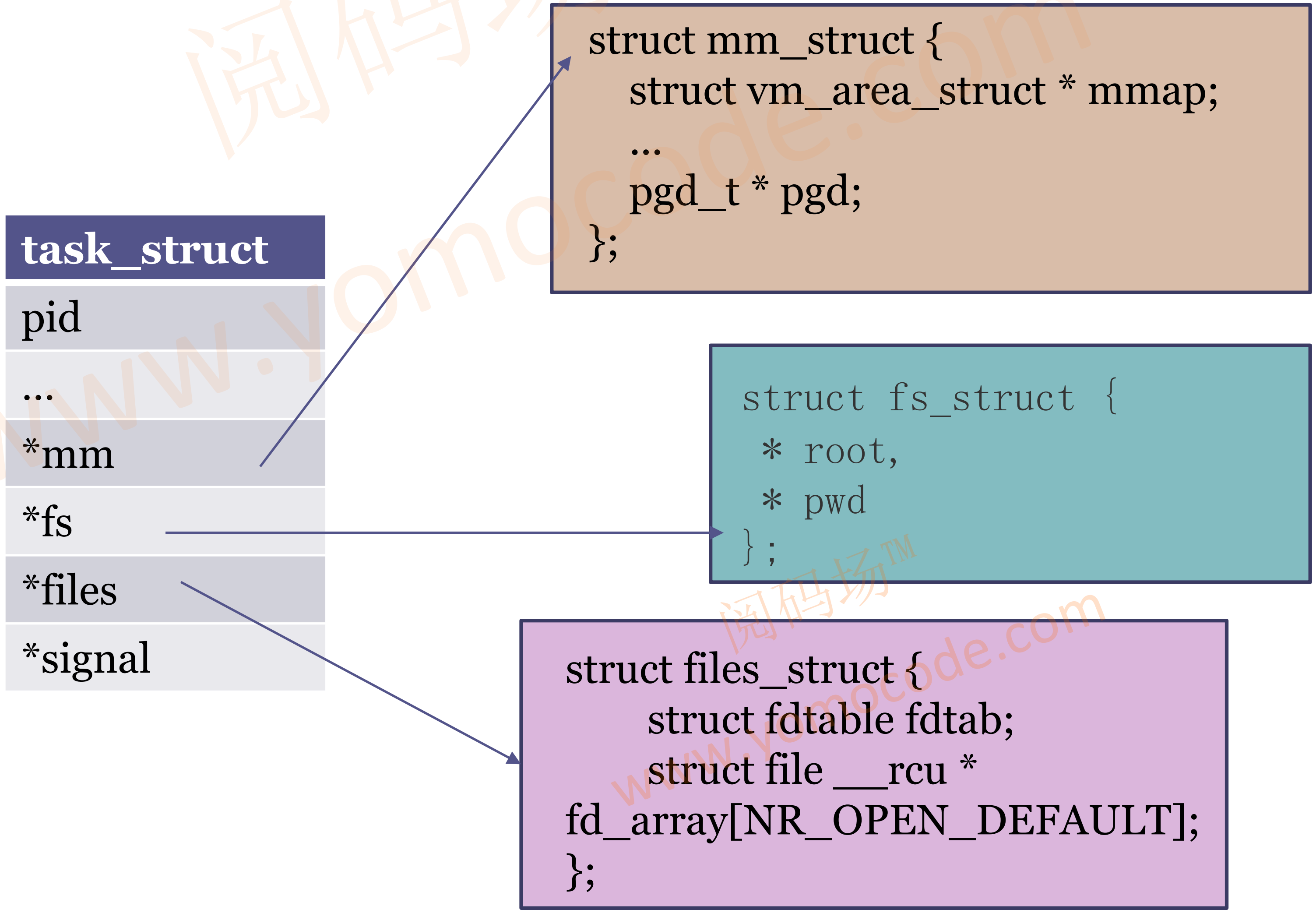
* Linux进程生命周期(就绪、运行、睡眠、停止、僵死)
- 僵尸是个什么鬼?
- 僵尸可以被杀死的一个假象

* 停止状态与作业控制，cpulimit
* 内存泄漏的真实含义
* task_struct以及task_struct之间的关系
* 初见fork和僵尸

练习题

* fork的例子
* life-period例子，观察僵尸
* 用cpulimit控制CPU利用率

进程控制块PCB



```
task_struct
pid
...
*mm
*fs
*files
*signal
```

```
struct mm_struct {
    struct vm_area_struct * mmap;
    ...
    pgd_t * pgd;
};
```

```
struct fs_struct {
 * root,
 * pwd
};
```

```
struct files_struct {
        struct fdtable fdtab;
        struct file __rcu *
fd_array[NR_OPEN_DEFAULT];
};
```

pid

pid的数量是有限的

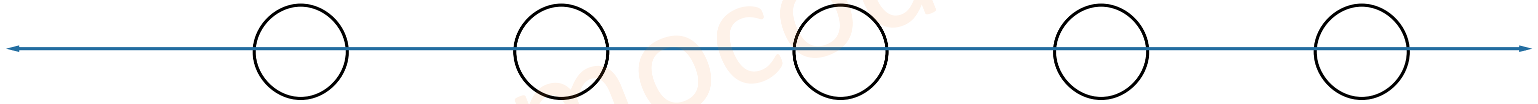
```
$ cat /proc/sys/kernel/pid_max
32768
```

Fork炸弹

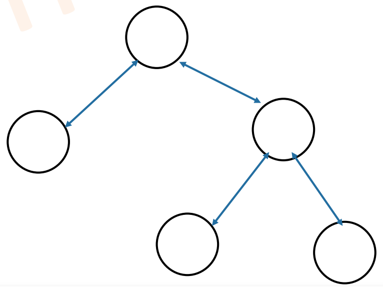
```
:(){ :|:& };:
```

task_struct被管理

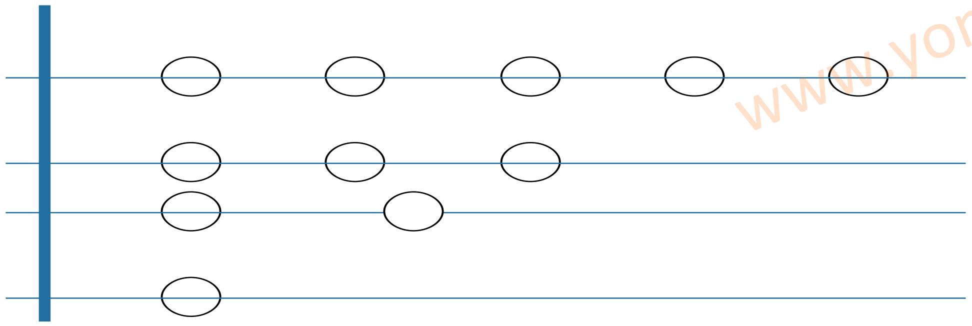
形成链表



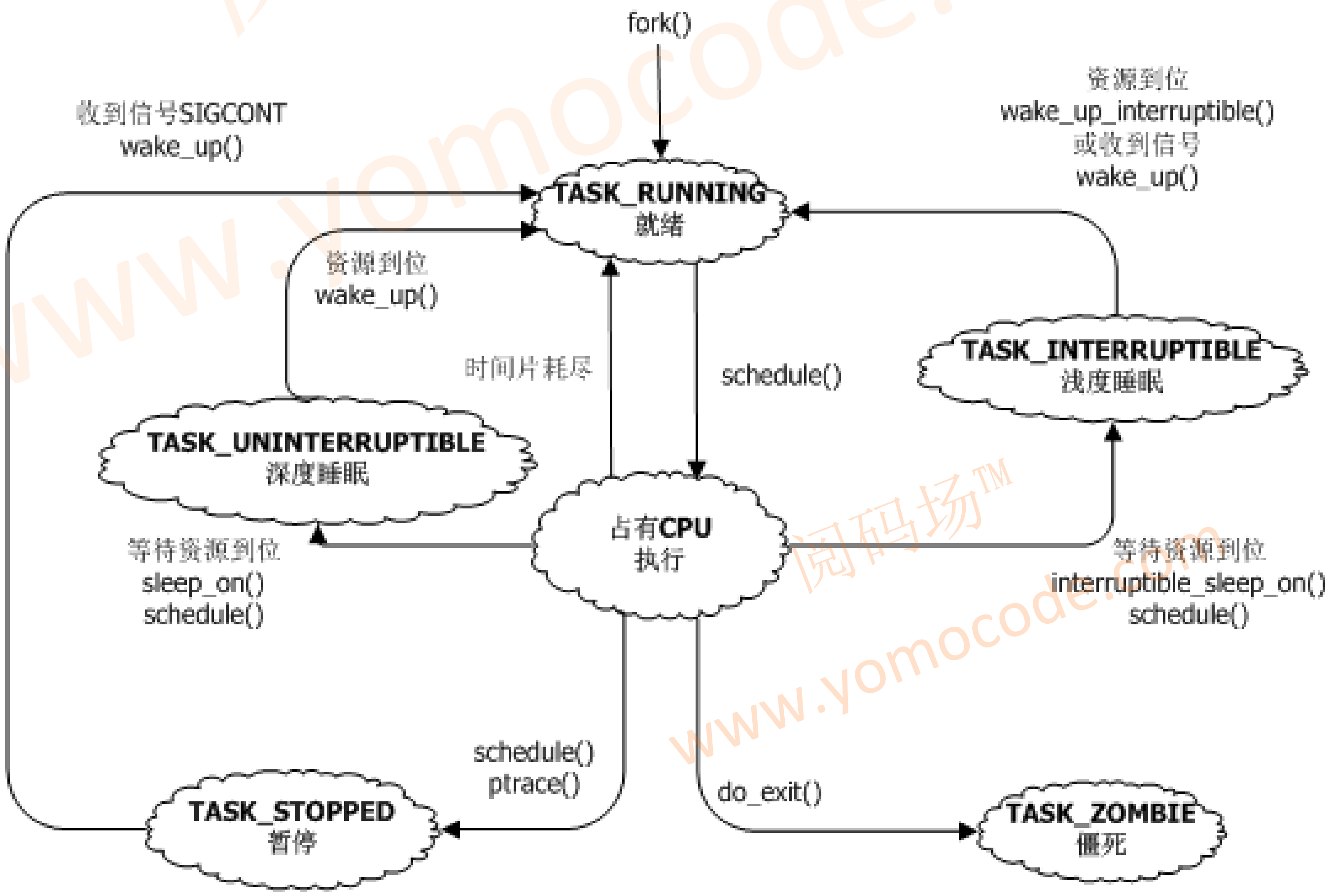
形成树



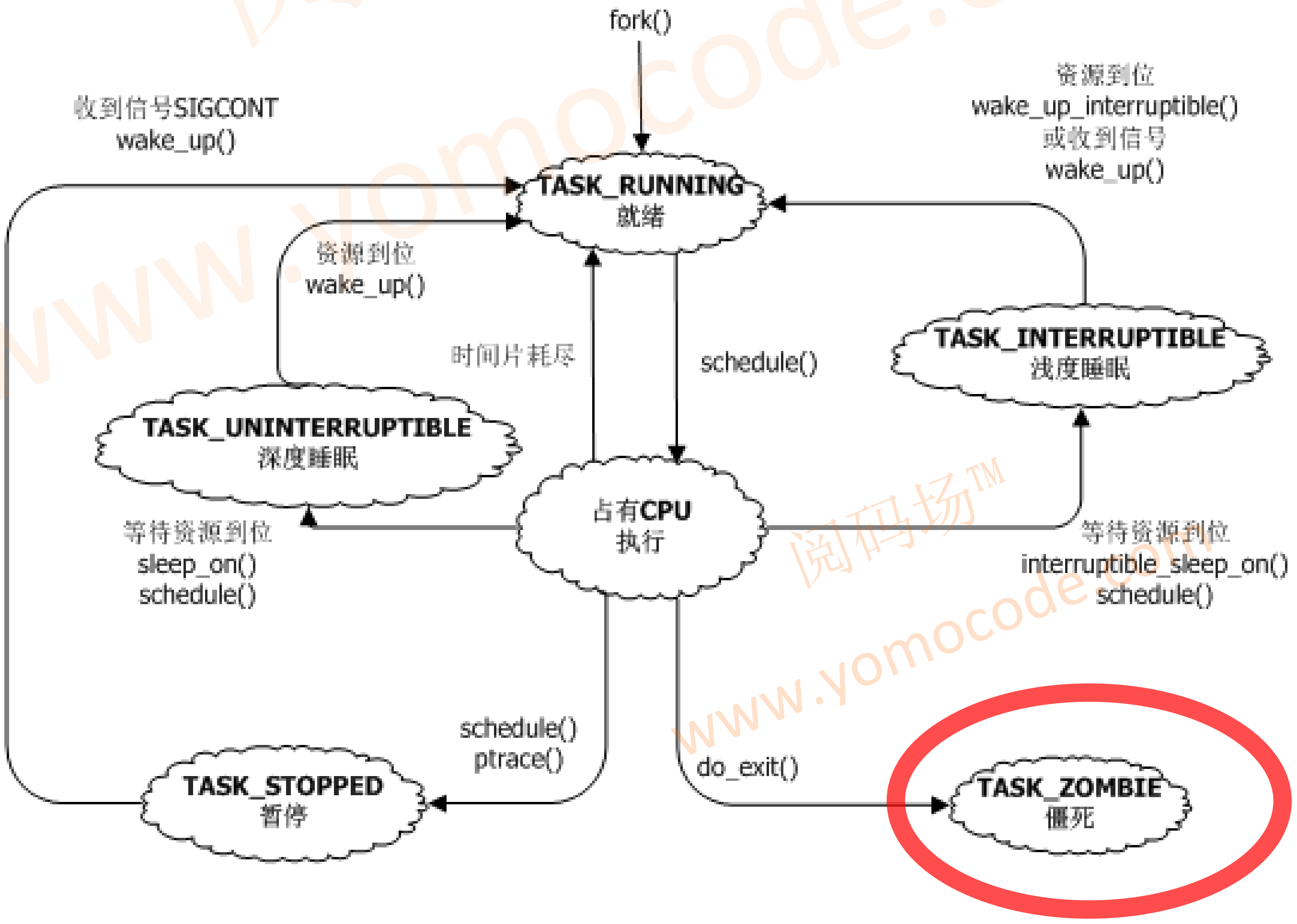
形成哈希：pid-> task_struct



进程生命周期

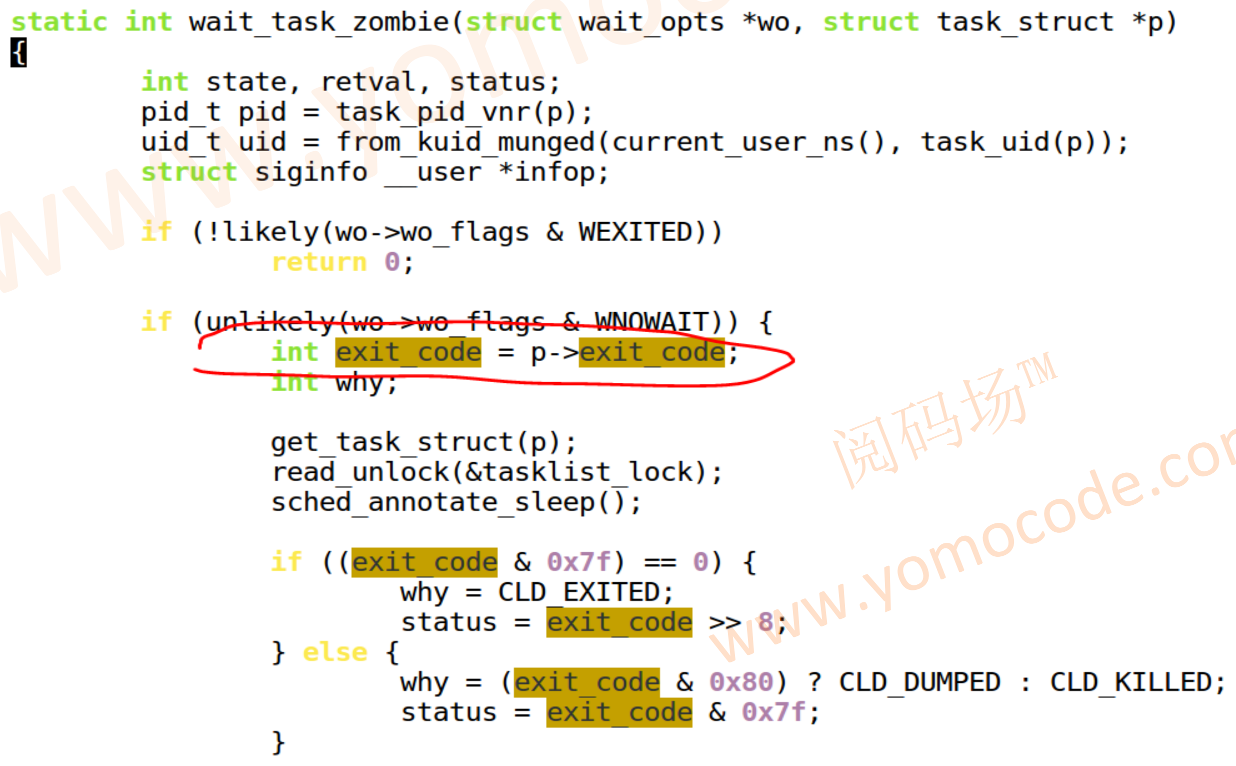

fork()
收到信号SIGCONT
wake_up()
资源到位
wake_up_interruptible()
或收到信号
wake_up()
TASK_RUNNING
就绪
资源到位
wake_up()
时间片耗尽
schedule()
TASK_INTERRUPTIBLE
浅度睡眠
TASK_UNINTERRUPTIBLE
深度睡眠
占有CPU
执行
等待资源到位
sleep_on()
schedule()
等待资源到位
interruptible_sleep_on()
schedule()
schedule()
ptrace()
do_exit()
TASK_STOPPED
暂停
TASK_ZOMBIE
僵死

进程生命周期


fork()
收到信号SIGCONT
wake_up()
资源到位
wake_up_interruptible()
或收到信号
wake_up()
TASK_RUNNING
就绪
资源到位
wake_up()
时间片耗尽
schedule()
TASK_INTERRUPTIBLE
浅度睡眠
TASK_UNINTERRUPTIBLE
深度睡眠
占有CPU
执行
等待资源到位
sleep_on()
schedule()
等待资源到位
interruptible_sleep_on()
schedule()
schedule()
ptrace()
do_exit()
TASK_STOPPED
暂停
TASK_ZOMBIE
僵死

僵尸是什么

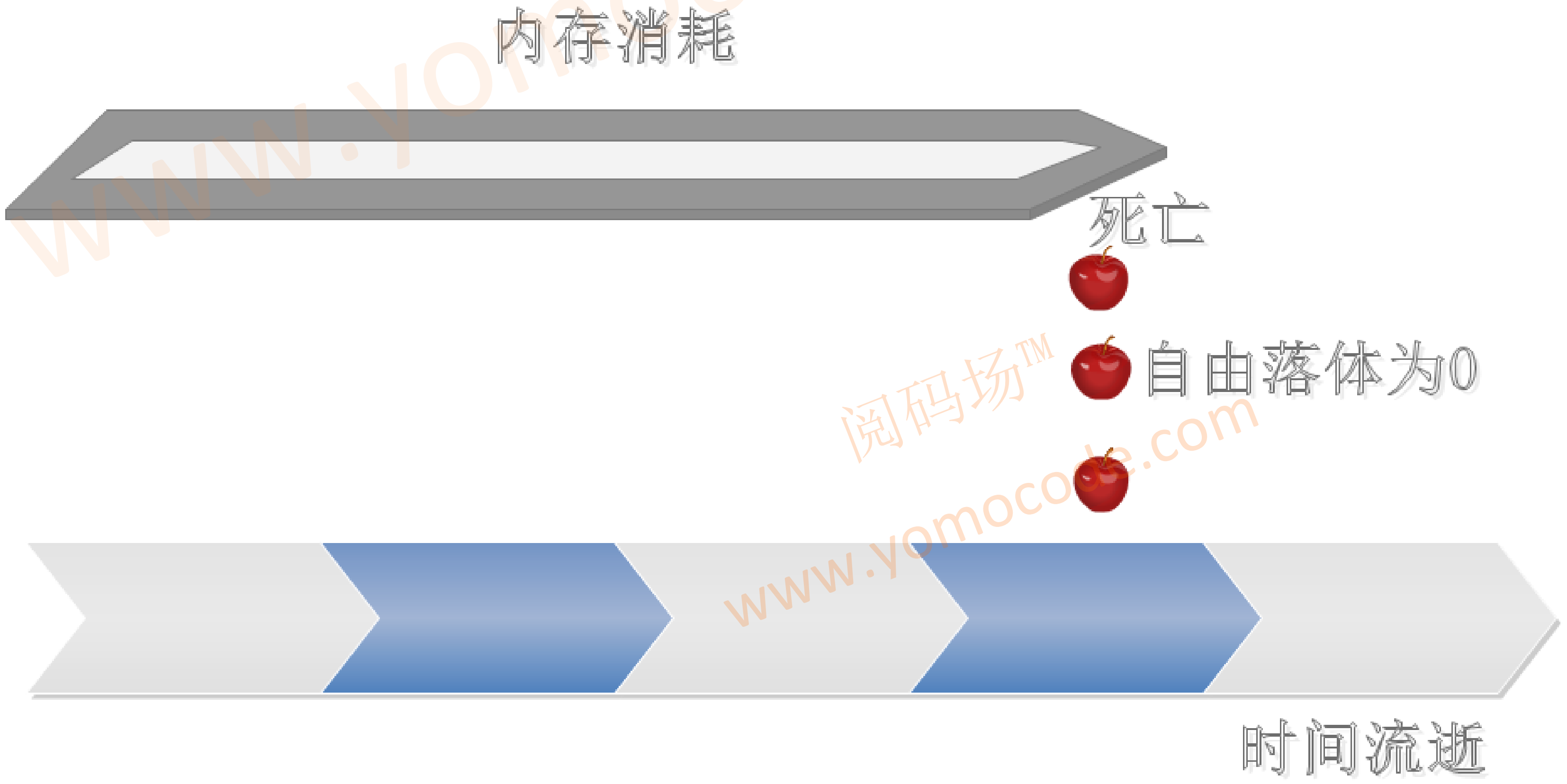
资源已经释放：无内存泄漏等
task_struct还在：父进程可以查到子进程死因



```
static int wait_task_zombie(struct wait_opts *wo, struct task_struct *p)
{
        int state, retval, status;
        pid_t pid = task_pid_vnr(p);
        uid_t uid = from_kuid_munged(current_user_ns(), task_uid(p));
        struct siginfo __user *infop;

        if (!likely(wo->wo_flags & WEXITED))
                return 0;

        if (unlikely(wo->wo_flags & WNOWAIT)) {
                int exit_code = p->exit_code;
                int why;

                get_task_struct(p);
                read_unlock(&tasklist_lock);
                sched_annotate_sleep();

                if ((exit_code & 0x7f) == 0) {
                        why = CLD_EXITED;
                        status = exit_code >> 8;
                } else {
                        why = (exit_code & 0x80) ? CLD_DUMPED : CLD_KILLED;
                        status = exit_code & 0x7f;
                }
```

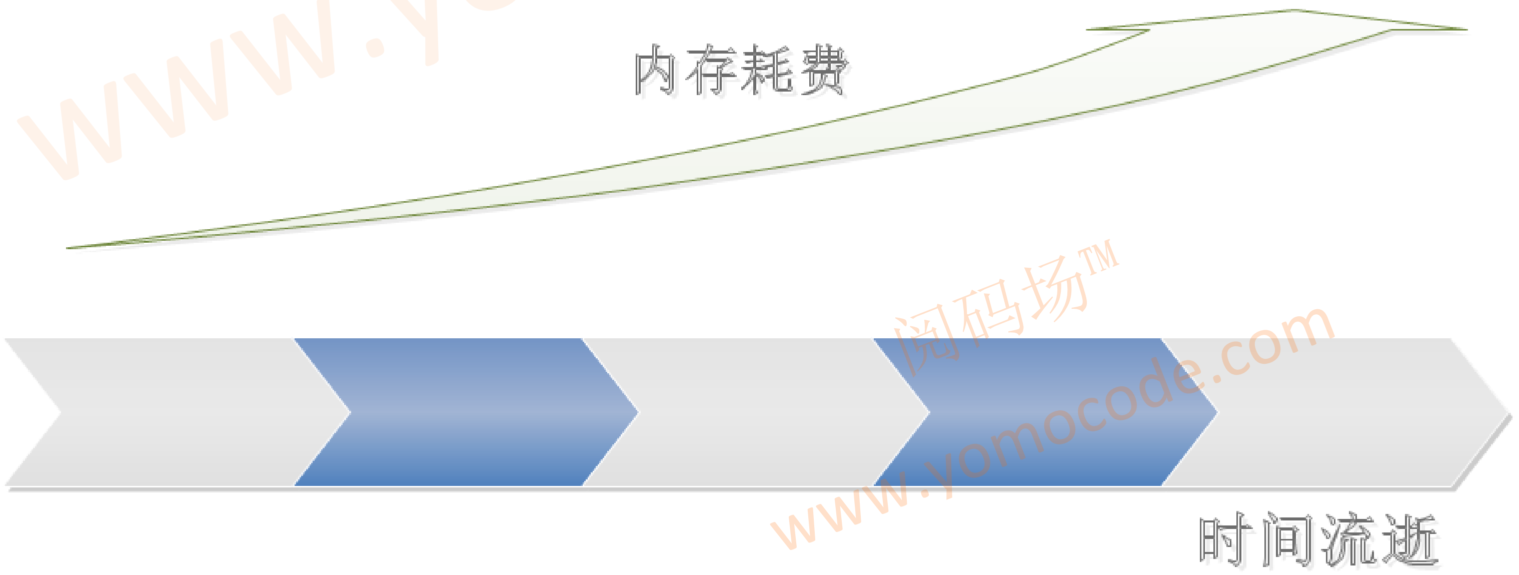
内存泄漏到底是什么？

不是：进程死了，内存没释放

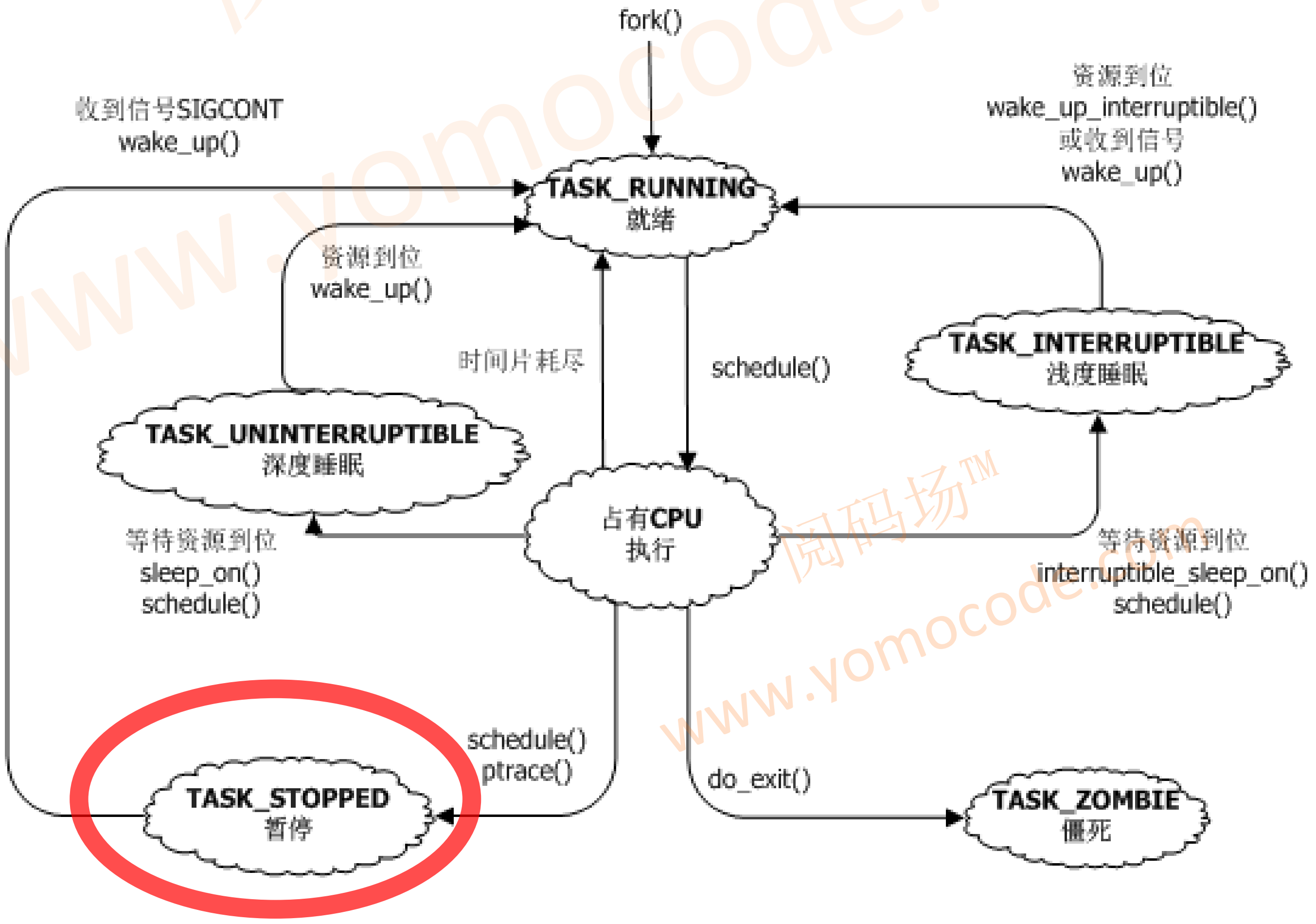


内存泄漏到底是什么(cont.)?

而是：进程活着，运行越久，耗费内存越多



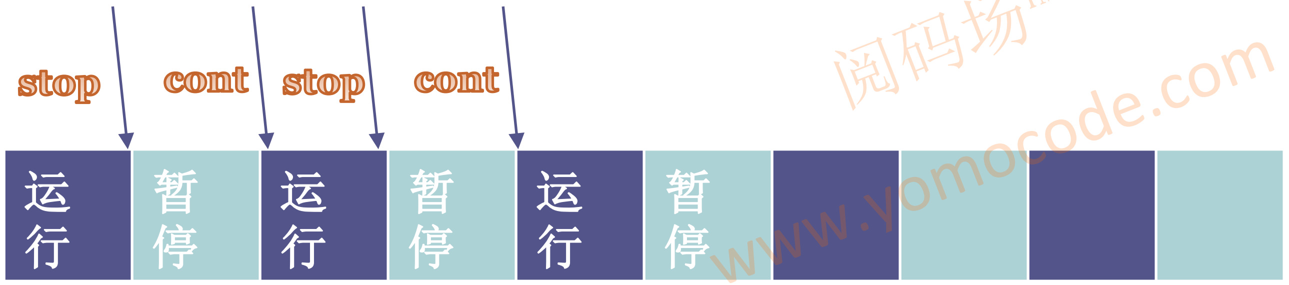
进程生命周期


fork()
收到信号SIGCONT
wake_up()
资源到位
wake_up_interruptible()
或收到信号
wake_up()
TASK_RUNNING
就绪
资源到位
wake_up()
时间片耗尽
schedule()
TASK_INTERRUPTIBLE
浅度睡眠
TASK_UNINTERRUPTIBLE
深度睡眠
占有CPU
执行
等待资源到位
sleep_on()
schedule()
等待资源到位
interruptible_sleep_on()
schedule()
schedule()
ptrace()
do_exit()
TASK_STOPPED
暂停
TASK_ZOMBIE
僵死

作业控制

ctrl+ z, fg/bg
cpulimit

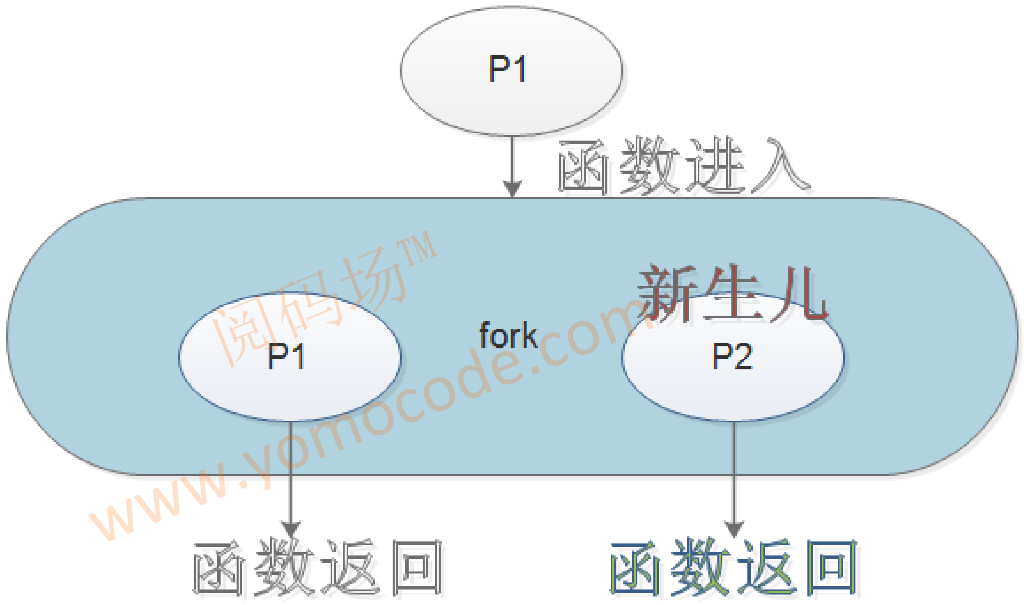
cpulimit -l 20 -p 10111
限制pid 为10111程序 的 cpu使用率不超过 10%



fork

打印几个hello?

```
main()
{
        fork();
        printf("hello\n");
        fork();
        printf("hello\n");
        while(1);
}
```

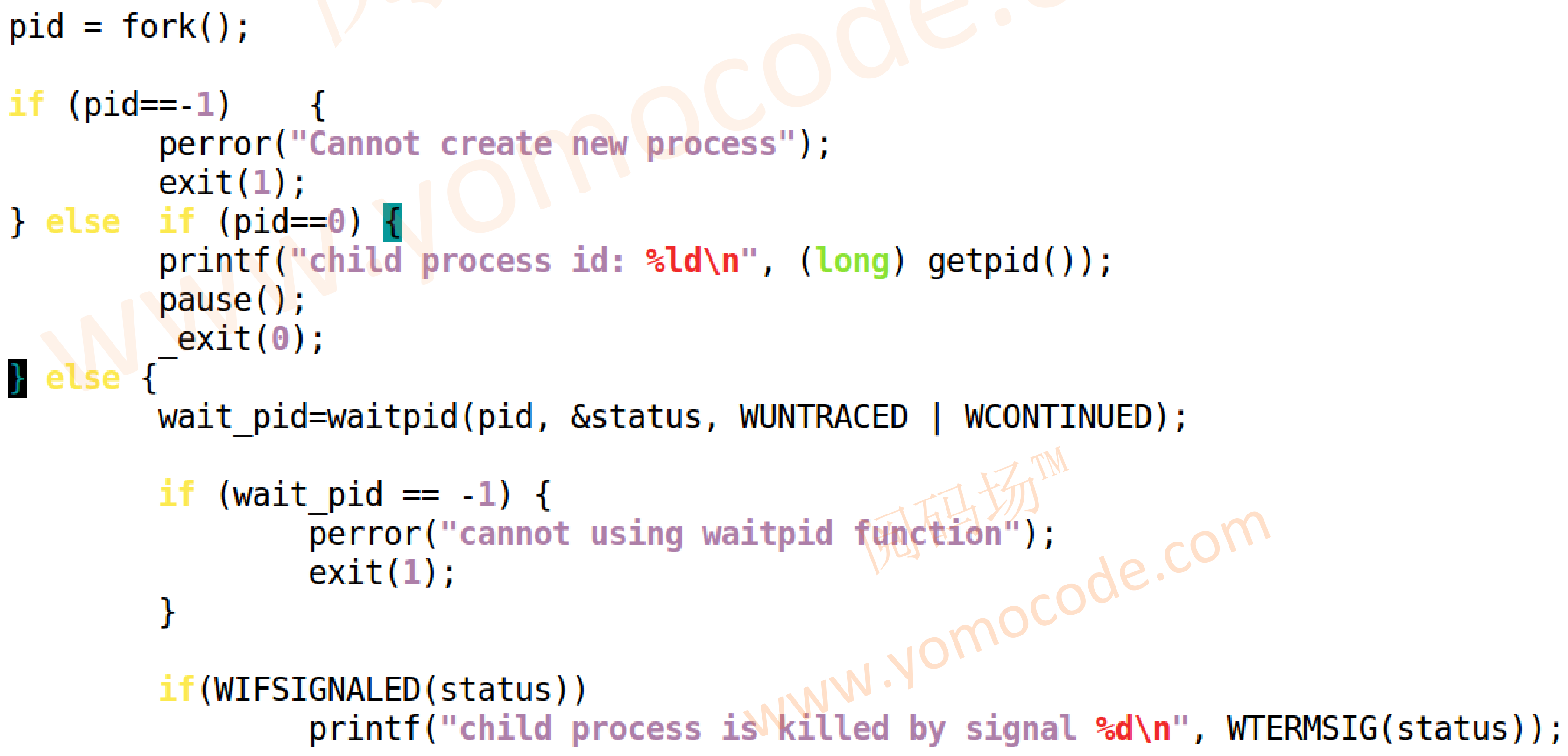


fork(cont.)

怎么打印?

```
int main(void)
{
        pid_t pid,wait_pid;
        int status;

        pid = fork();

        if (pid==-1)    {
                perror("Cannot create new process");
                exit(1);
        } else  if (pid==0) {
                printf("a\n");
        } else {
                printf("b\n");
        }

        printf("c\n");
        while(1);
}
```

子死父清场



```
pid = fork();

if (pid==-1)    {
        perror("Cannot create new process");
        exit(1);
} else  if (pid==0) {
        printf("child process id: %ld\n", (long) getpid());
        pause();
        _exit(0);
} else {
        wait_pid=waitpid(pid, &status, WUNTRACED | WCONTINUED);

        if (wait_pid == -1) {
                perror("cannot using waitpid function");
                exit(1);
        }

        if(WIFSIGNALED(status))
                printf("child process is killed by signal %d\n", WTERMSIG(status));
```

僵尸可以被杀死的一个假象

- **主线程通过pthread_exit退出**

```
int main(void)
{
        pthread_t tid1,tid2;
        …
        ret=pthread_create(&tid1,NULL,thread_fun,&info1);
        …
        ret=pthread_create(&tid2,NULL,thread_fun,&info2);
        …
        pthread_exit(0);

        return 0;
}
```

参考源码

https://github.com/srclib/RageAgainstTheCage/blob/master/RageAgainstTheCage/rageagainstthecage.c

谢谢!