5 进程

5.1 背景简介

操作系统最初的原型是一种**批处理系统**（batch）：在最开始我们把正在执行的程序称为作业，操作员将所有的作业放在一起，由批处理系统进行读取并交给计算机执行，当一个作业执行完成以后，批处理系统会自动地取出下一个作业。在批处理系统中，存在的最严重的问题是任务执行的过程，会经常需要等待IO操作，这会导致CPU经常性地空闲。

为了充分提高CPU的利用率，一种解决方案是引入**多道程序设计**，在内存当中划分多个区域，每个区域存储一个作业的指令和数据，当其中一个作业在等待IO操作时，另外一个作业可以使用CPU。

除了希望充分利用CPU资源，程序员还希望自己的对计算机的操作能够得到迅速的响应，这种需求导致了**分时系统**的诞生。在分时系统中，操作系统会给每个作业分配一些时间来使用CPU，这样CPU就不会沉浸在一个大型的作业中，期间的用户操作也会很快地得到响应。

由于同时执行多个作业，所以计算机需要提供一种优雅的方式来在各个任务之间来分配CPU，并且同一时间也会有多个作业的数据和指令在内存当中驻留。传统的基于硬件的CPU和内存保护机制显得十分复杂，并且程序员编写程序的时候需要花费很多心智负担来考虑计算机硬件的限制。因此，操作系统在计算机硬件和用户之间引入一个中间层——**进程**。在用户的视角中，会把运行中的程序看成一个进程，每个进程拥有自己独立的CPU和内存资源，并且可以管理各种其他所有资源，当然这种看起来的独占都是**虚拟**的；操作系统的作用就要管理真实、复杂和丑陋的底层硬件，为用户视角当中简单且美好的进程来提供支持。

5.2 进程的概念

通过对操作系统发展史的简单回顾，我们就可以给进程下一个完善的定义了。

从用户的角度来看，进程是一个程序的动态执行过程。程序是静态文件，是一系列二进制指令和数据的集合，程序通常存储在磁盘当中。进程则是动态的，当程序被触发以后（比如用户启动程序或者被其他进程启动程序），启动者的权限和属性以及程序的指令和数据会被加载到内存当中，并且占用CPU和其他系统资源动态地执行指令和读写数据。进程的状态会在动态地在创建、调度、运行和消亡之间转换。

从操作系统的角度来看，进程是各种计算机资源分配的基本单位。操作系统需要提供支持让每个进程以为自己能够独占CPU和内存等资源——即所谓的**虚拟CPU**和**虚拟内存**。每个进程需要占用CPU资源以执行程序指令；而除了需要占用CPU资源以外，进程还需要占据存储资源来保存状态。进程需要保存的内容包括数据段、代码段、堆以及其他内存空间，进程也需要占用资源管理打开的文件、挂起的信号、内核内部数据、处理器状态、存在内存映射的内存空间以及执行线程，而执行线程的信息则包含程序计数器、栈和寄存器状态。

5.3 虚拟CPU和虚拟内存

5.3.1 虚拟CPU

利用进程机制，所有的现代操作系统都支持在同一个时间来完成多个任务。尽管某个时刻，真实的CPU只能运行一个进程，但是从进程自己的角度来看，它会认为自己在**独享**CPU（即**虚拟CPU**），而从用户的角度来看多个进程在一段时间内是同时执行的，即**并发**执行。在实际的实现中，操作系统会使用**调度器**来分配CPU资源。调度器会根据策略和优先级来给各个进程一定的时间来占用CPU，进程占用CPU时间的基本单位称为**时间片**，当进程不应该使用CPU资源时，调度器会抢占CPU的控制权，然后快速地切换CPU的使用进程。这种切换在用户的视角中对程序执行毫无影响，可以认为是透明的。由于切换进程消耗的时间和每个进程实际执行的时间片是在非常小的，以至于用户无法分辨，所以在用户看起来，多个进程是在同时运行的。

5.3.2 调度器和优先级

Linux内核存在一个专门用来调度进程的内核线程，称为**调度器(或者是调度程序)**。调度器的基本工作是从一组处于可执行状态的进程中选择一个来执行。和很多其他操作系统一样，Linux提供了**抢占式**的多任务模式。调度器会决定某个进程在什么时候停止运行，并且可以让另一个进程进入执行，这个动作即所谓的**抢占**。

传统的Unix操作系统采用的**时间片轮转法**。这种方法的大致流程是这样的：调度器它把所有的可运行的非实时的进程组织在一起，这个数据结构被称为**就绪队列**。通常调度器取一个固定时间作为**调度周期（又称为调度延迟）**，当一个调度周期开始的时候，调度器会为就绪队列当中的每个进程分配时间片，每个进程能够获取的**时间片长度**和进程的**优先级**有关。正在执行的进程会在执行的过程逐渐消耗它的时间片，当时间片耗尽时，如果该进程依然处于运行状态，那么它就会被就绪队列中的下一个进程抢占。如果整个调度周期执行完成，调度器就会抢占该进程，并且根据优先级分配新的时间片。

在进程执行过程中，会出现等待IO操作的情况。此时，进程就会从就绪队列当中移除，并且将其放入**等待队列**，并且将自己的状态调整为**等待状态**。进程运行终止时，进程也会从就绪队列中移除。

当进程被创建或者被从等待状态唤醒时，调度器会根据优先级分配时间片，再将其插入到就绪队列当中。

Linux所使用的调度算法之一是**完全公平调度**算法，简称CFS，虽然在概念上和时间片轮转法有很多区别，但是具体表现基本一致。

5.3.3 虚拟内存

在进程本身的视角中，它除了会认为CPU是独占的以外，它还会以为自己是内存空间的独占者，这种从进程视角看到的内存空间被称为**虚拟内存空间**。当操作系统中有多个进程同时运行时，为了避免真实的**物理内存**访问在不同进程之间发生冲突，操作系统需要提供一种机制在虚拟内存和真实的物理内存之间建立**映射**。

因此，进程之间具有并行性、互不干扰等特点。也就是说，进程之间是分离的任务，拥有各自的权利和责任。每个进程都运行在各自独立的虚拟地址空间，因此，即使一个进程发生了异常，它也不会影响到系统的其他进程。

5.3.4 进程地址空间

Linux为每个进程维持了一个单独的虚拟地址空间，也就是进程地址空间。

在内核态空间里面，与进程相关的数据结构是每个进程都不同的，物理存储器和内核代码数据是所有进程共享的。

5.3.5 内核态和用户态

CPU或者说计算机硬件的运行状态可以分为内核态和用户态。在计算机处于**内核态**的情况下，它可以执行**所有**的指令；在计算机处于用户态的情况下，它只能执行部分的指令。所有涉及到内存管理、外部设备访问（比如读写网卡、读写文件等等）以及需要访问内核态地址空间的指令都需要在内核态下完成。计算机可以通过中断或者是异常的方式从用户态陷入到内核态。系统调用的底层实现当中，就使用了主动触发中断即异常的指令。

5.4 进程管理

5.4.1 内核管理进程信息

内核把所有的进程维护成一个名为任务队列的双向循环链表。链表的节点类型名为`task_struct`，它声明在`/include/linux/sched.h`中，`task_struct`可以被称为进程控制块。它里面存储了进程的各种静态信息，包括打开的文件、进程的地址空间、挂起的信号、进程的状态等等。下面是某个版本的`task_struct`结构体的声明，在后面的课程中会不断地涉及这些内容。

```
//某个版本内核的完整task_struct
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
```

```
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info; //用来分配内存的数据结构
#endif
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state; //进程状态

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack; //内核栈地址
	atomic_t			usage; //用于多线程，描述有多少个线程共享一个task_struct
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;

#ifdef CONFIG_SMP //对称多处理器架构的架构特定字段 目前的PC一般都是SMP
	struct llist_node		wake_entry;
	int				on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/* Current CPU: */
	unsigned int			cpu;
#endif
	unsigned int			wakee_flips;
	unsigned long			wakee_flip_decay_ts;
	struct task_struct		*last_wakee;

	/*
	 * recent_used_cpu is initially set as the last CPU used by a task
	 * that wakes affine another task. Waker/wakee relationships can
	 * push tasks around a CPU where each wakeup moves to the next one.
	 * Tracking a recently used CPU allows a quick search for a recently
	 * used CPU that may be idle.
	 */
	int				recent_used_cpu;
	int				wake_cpu;
#endif
	int				on_rq; //是否在就绪队列

	int				prio; //动态优先级
	int				static_prio; //静态优先级 可以通过nice调整
	int				normal_prio; //普通优化级 由prio&normal_prio决定
	unsigned int			rt_priority; //实时优先级

	const struct sched_class	*sched_class; //调度类
	struct sched_entity		se; // CFS调度实体
	struct sched_rt_entity		rt; //实时调度实体
#ifdef CONFIG_CGROUP_SCHED
	struct task_group		*sched_task_group; //用于Cgroup机制，可以为不同的控制组分
配资源
#endif
	struct sched_dl_entity		dl; //EDF调度实体
```

```
#ifdef CONFIG_PREEMPT_NOTIFIERS //可抢占内核通知
    /* List of struct preempt_notifier: */
    struct hlist_head       preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE //追踪块设备IO
    unsigned int            btrace_seq;
#endif

    unsigned int            policy; //调度策略
    int             nr_cpus_allowed;
    cpumask_t           cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU //可抢占内核的同步机制
    int             rcu_read_lock_nesting;
    union rcu_special       rcu_read_unlock_special;
    struct list_head        rcu_node_entry;
    struct rcu_node         *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */

#ifdef CONFIG_TASKS_RCU
    unsigned long           rcu_tasks_nvcsw;
    u8              rcu_tasks_holdout;
    u8              rcu_tasks_idx;
    int             rcu_tasks_idle_cpu;
    struct list_head        rcu_tasks_holdout_list;
#endif /* #ifdef CONFIG_TASKS_RCU */

    struct sched_info       sched_info;

    struct list_head        tasks; //任务队列的队首
#ifdef CONFIG_SMP
    struct plist_node       pushable_tasks;
    struct rb_node          pushable_dl_tasks;
#endif

    struct mm_struct        *mm; //内存分配情况
    struct mm_struct        *active_mm; //区分用户进程和内核线程

    /* Per-thread vma caching: */
    struct vmacache         vmacache;

#ifdef SPLIT_RSS_COUNTING
    struct task_rss_stat        rss_stat;
#endif
    int             exit_state; //退出相关的状态 可以用wait获取
    int             exit_code;
    int             exit_signal;
    /* The signal sent when the parent dies: */
    int             pdeath_signal;
    /* JOBCTL_*, siglock protected: */
    unsigned long           jobctl;

    /* Used for emulating ABI behavior of previous Linux versions: */
    unsigned int            personality;
```

```
    /* Scheduler bits, serialized by scheduler locks: */
    unsigned            sched_reset_on_fork:1;
    unsigned            sched_contributes_to_load:1;
    unsigned            sched_migrated:1;
    unsigned            sched_remote_wakeup:1;
#ifdef CONFIG_PSI
    unsigned            sched_psi_wake_requeue:1;
#endif

    /* Force alignment to the next boundary: */
    unsigned            :0;

    /* Unserialized, strictly 'current' */

    /* Bit to tell LSMs we're in execve(): */
    unsigned            in_execve:1;
    unsigned            in_iowait:1;
#ifndef TIF_RESTORE_SIGMASK
    unsigned            restore_sigmask:1;
#endif
#ifdef CONFIG_MEMCG
    unsigned            in_user_fault:1;
#endif
#ifdef CONFIG_COMPAT_BRK
    unsigned            brk_randomized:1;
#endif
#ifdef CONFIG_CGROUPS
    /* disallow userland-initiated cgroup migration */
    unsigned            no_cgroup_migration:1;
#endif
#ifdef CONFIG_BLK_CGROUP
    /* to be used once the psi infrastructure lands upstream. */
    unsigned            use_memdelay:1;
#endif

    unsigned long           atomic_flags; /* Flags requiring atomic access. */

    struct restart_block        restart_block;

    pid_t               pid; //pid
    pid_t               tgid; //线程组id，用于多线程的时候获取pid

#ifdef CONFIG_STACKPROTECTOR
    /* Canary value for the -fstack-protector GCC feature: */
    unsigned long           stack_canary;
#endif
    /*
     * Pointers to the (original) parent process, youngest child, younger
sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->real_parent->pid)
     */

    /* Real parent process: */
    struct task_struct __rcu    *real_parent;//真实的父进程
```

```
    /* Recipient of SIGCHLD, wait4() reports: */
    struct task_struct __rcu    *parent; //当前父进程

    /*
     * Children/sibling form the list of natural children:
     */
    struct list_head        children; //子进程
    struct list_head        sibling; //兄弟进程
    struct task_struct      *group_leader; //组长的进程描述符

    /*
     * 'ptraced' is the list of tasks this task is using ptrace() on.
     *
     * This includes both natural children and PTRACE_ATTACH targets.
     * 'ptrace_entry' is this task's link on the p->parent->ptraced list.
     */
    struct list_head        ptraced; //用于描述追踪状态 比如gdb、ptrace等工具
    struct list_head        ptrace_entry;

    /* PID/PID hash table linkage. */
    struct pid          *thread_pid;
    struct hlist_node       pid_links[PIDTYPE_MAX];
    struct list_head        thread_group;
    struct list_head        thread_node;

    struct completion       *vfork_done;

    /* CLONE_CHILD_SETTID: */
    int __user          *set_child_tid;

    /* CLONE_CHILD_CLEARTID: */
    int __user          *clear_child_tid;

    u64             utime;
    u64             stime;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
    u64             utimescaled;
    u64             stimescaled;
#endif
    u64             gtime;
    struct prev_cputime     prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
    struct vtime            vtime;
#endif

#ifdef CONFIG_NO_HZ_FULL
    atomic_t            tick_dep_mask;
#endif
    /* Context switch counts: */
    unsigned long           nvcsw;
    unsigned long           nivcsw;

    /* Monotonic time in nsecs: */
    u64             start_time;
```

```
	/* Boot based time in nsecs: */
	u64				real_start_time;

	/* MM fault and swap info: this can arguably be seen as either mm-specific or
thread-specific: */
	unsigned long			min_flt;
	unsigned long			maj_flt;

#ifdef CONFIG_POSIX_TIMERS
	struct task_cputime		cputime_expires;
	struct list_head		cpu_timers[3];
#endif

	/* Process credentials: */

	/* Tracer's credentials at attach: */
	const struct cred __rcu		*ptracer_cred;

	/* Objective and real subjective task credentials (COW): */
	const struct cred __rcu		*real_cred;

	/* Effective (overridable) subjective task credentials (COW): */
	const struct cred __rcu		*cred;

	/*
	 * executable name, excluding path.
	 *
	 * - normally initialized setup_new_exec()
	 * - access it with [gs]et_task_comm()
	 * - lock it with task_lock()
	 */
	char				comm[TASK_COMM_LEN];

	struct nameidata		*nameidata;

#ifdef CONFIG_SYSVIPC
	struct sysv_sem			sysvsem;
	struct sysv_shm			sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
	unsigned long			last_switch_count;
	unsigned long			last_switch_time;
#endif
	/* Filesystem information: */
	struct fs_struct		*fs; //文件系统的信息

	/* Open file information: */
	struct files_struct		*files; //打开文件表

	/* Namespaces: */
	struct nsproxy			*nsproxy;

	/* Signal handlers: */
	struct signal_struct		*signal;
	struct sighand_struct		*sighand;
	sigset_t			blocked;
```

```
    sigset_t            real_blocked;
    /* Restored if set_restore_sigmask() was used: */
    sigset_t            saved_sigmask;
    struct sigpending       pending;
    unsigned long           sas_ss_sp;
    size_t              sas_ss_size;
    unsigned int            sas_ss_flags;

    struct callback_head        *task_works;

    struct audit_context        *audit_context;
#ifdef CONFIG_AUDITSYSCALL
    kuid_t              loginuid;
    unsigned int            sessionid;
#endif
    struct seccomp          seccomp;

    /* Thread group tracking: */
    u32             parent_exec_id;
    u32             self_exec_id;

    /* Protection against (de-)allocation: mm, files, fs, tty, keyrings,
mems_allowed, mempolicy: */
    spinlock_t          alloc_lock;

    /* Protection of the PI data structures: */
    raw_spinlock_t          pi_lock;

    struct wake_q_node      wake_q;

#ifdef CONFIG_RT_MUTEXES
    /* PI waiters blocked on a rt_mutex held by this task: */
    struct rb_root_cached       pi_waiters;
    /* Updated under owner's pi_lock and rq lock */
    struct task_struct      *pi_top_task;
    /* Deadlock detection and priority inheritance handling: */
    struct rt_mutex_waiter      *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
    /* Mutex deadlock detection: */
    struct mutex_waiter     *blocked_on;
#endif

#ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int            irq_events;
    unsigned long           hardirq_enable_ip;
    unsigned long           hardirq_disable_ip;
    unsigned int            hardirq_enable_event;
    unsigned int            hardirq_disable_event;
    int             hardirqs_enabled;
    int             hardirq_context;
    unsigned long           softirq_disable_ip;
    unsigned long           softirq_enable_ip;
    unsigned int            softirq_disable_event;
    unsigned int            softirq_enable_event;
```

```
	int				softirqs_enabled;
	int				softirq_context;
#endif

#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH			48UL
	u64				curr_chain_key;
	int				lockdep_depth;
	unsigned int			lockdep_recursion;
	struct held_lock		held_locks[MAX_LOCK_DEPTH];
#endif

#ifdef CONFIG_UBSAN
	unsigned int			in_ubsan;
#endif

	/* Journalling filesystem info: */
	void				*journal_info;

	/* Stacked block device info: */
	struct bio_list			*bio_list;

#ifdef CONFIG_BLOCK
	/* Stack plugging: */
	struct blk_plug			*plug;
#endif

	/* VM state: */
	struct reclaim_state		*reclaim_state;

	struct backing_dev_info		*backing_dev_info;

	struct io_context		*io_context;

	/* Ptrace state: */
	unsigned long			ptrace_message;
	kernel_siginfo_t		*last_siginfo;

	struct task_io_accounting	ioac;
#ifdef CONFIG_PSI
	/* Pressure stall state */
	unsigned int			psi_flags;
#endif
#ifdef CONFIG_TASK_XACCT
	/* Accumulated RSS usage: */
	u64				acct_rss_mem1;
	/* Accumulated virtual memory usage: */
	u64				acct_vm_mem1;
	/* stime + utime since last update: */
	u64				acct_timexpd;
#endif
#ifdef CONFIG_CPUSETS
	/* Protected by ->alloc_lock: */
	nodemask_t			mems_allowed;
	/* Seqence number to catch updates: */
	seqcount_t			mems_allowed_seq;
```

```
	int				cpuset_mem_spread_rotor;
	int				cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
	/* Control Group info protected by css_set_lock: */
	struct css_set __rcu		*cgroups;
	/* cg_list protected by css_set_lock and tsk->alloc_lock: */
	struct list_head		cg_list;
#endif
#ifdef CONFIG_X86_CPU_RESCTRL
	u32				closid;
	u32				rmid;
#endif
#ifdef CONFIG_FUTEX
	struct robust_list_head __user	*robust_list;
#ifdef CONFIG_COMPAT
	struct compat_robust_list_head __user *compat_robust_list;
#endif
	struct list_head		pi_state_list;
	struct futex_pi_state		*pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
	struct perf_event_context	*perf_event_ctxp[perf_nr_task_contexts];
	struct mutex			perf_event_mutex;
	struct list_head		perf_event_list;
#endif
#ifdef CONFIG_DEBUG_PREEMPT
	unsigned long			preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
	/* Protected by alloc_lock: */
	struct mempolicy		*mempolicy;
	short				il_prev;
	short				pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
	int				numa_scan_seq;
	unsigned int			numa_scan_period;
	unsigned int			numa_scan_period_max;
	int				numa_preferred_nid;
	unsigned long			numa_migrate_retry;
	/* Migration stamp: */
	u64				node_stamp;
	u64				last_task_numa_placement;
	u64				last_sum_exec_runtime;
	struct callback_head		numa_work;

	/*
	 * This pointer is only modified for current in syscall and
	 * pagefault context (and for tasks being destroyed), so it can be read
	 * from any of the following contexts:
	 *  - RCU read-side critical section
	 *  - current->numa_group from everywhere
	 *  - task's runqueue locked, task not running
	 */
	struct numa_group __rcu		*numa_group;
```

```
    /*
     * numa_faults is an array split into four regions:
     * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
     * in this precise order.
     *
     * faults_memory: Exponential decaying average of faults on a per-node
     * basis. Scheduling placement decisions are made based on these
     * counts. The values remain static for the duration of a PTE scan.
     * faults_cpu: Track the nodes the process was running on when a NUMA
     * hinting fault was incurred.
     * faults_memory_buffer and faults_cpu_buffer: Record faults per node
     * during the current scan window. When the scan completes, the counts
     * in faults_memory and faults_cpu decay and these values are copied.
     */
    unsigned long           *numa_faults;
    unsigned long           total_numa_faults;

    /*
     * numa_faults_locality tracks if faults recorded during the last
     * scan window were remote/local or failed to migrate. The task scan
     * period is adapted based on the locality of the faults with different
     * weights depending on whether they were shared or private faults
     */
    unsigned long           numa_faults_locality[3];

    unsigned long           numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */

#ifdef CONFIG_RSEQ
    struct rseq __user *rseq;
    u32 rseq_len;
    u32 rseq_sig;
    /*
     * RmW on rseq_event_mask must be performed atomically
     * with respect to preemption.
     */
    unsigned long rseq_event_mask;
#endif

    struct tlbflush_unmap_batch tlb_ubc;

    struct rcu_head         rcu;

    /* Cache last used pipe for splice(): */
    struct pipe_inode_info      *splice_pipe;

    struct page_frag        task_frag;

#ifdef CONFIG_TASK_DELAY_ACCT
    struct task_delay_info      *delays;
#endif

#ifdef CONFIG_FAULT_INJECTION
    int             make_it_fail;
    unsigned int            fail_nth;
```

```
#endif
	/*
	 * When (nr_dirtied >= nr_dirtied_pause), it's time to call
	 * balance_dirty_pages() for a dirty throttling pause:
	 */
	int				nr_dirtied;
	int				nr_dirtied_pause;
	/* Start of a write-and-pause period: */
	unsigned long			dirty_paused_when;

#ifdef CONFIG_LATENCYTOP
	int				latency_record_count;
	struct latency_record		latency_record[LT_SAVECOUNT];
#endif
	/*
	 * Time slack values; these are used to round up poll() and
	 * select() etc timeout values. These are in nanoseconds.
	 */
	u64				timer_slack_ns;
	u64				default_timer_slack_ns;

#ifdef CONFIG_KASAN
	unsigned int			kasan_depth;
#endif

#ifdef CONFIG_FUNCTION_GRAPH_TRACER
	/* Index of current stored address in ret_stack: */
	int				curr_ret_stack;
	int				curr_ret_depth;

	/* Stack of return addresses for return function tracing: */
	struct ftrace_ret_stack		*ret_stack;

	/* Timestamp for last schedule: */
	unsigned long long		ftrace_timestamp;

	/*
	 * Number of functions that haven't been traced
	 * because of depth overrun:
	 */
	atomic_t			trace_overrun;

	/* Pause tracing: */
	atomic_t			tracing_graph_pause;
#endif

#ifdef CONFIG_TRACING
	/* State flags for use by tracers: */
	unsigned long			trace;

	/* Bitmask and counter of trace recursion: */
	unsigned long			trace_recursion;
#endif /* CONFIG_TRACING */

#ifdef CONFIG_KCOV
	/* Coverage collection mode enabled for this task (0 if disabled): */
```

```
	unsigned int			kcov_mode;

	/* Size of the kcov_area: */
	unsigned int			kcov_size;

	/* Buffer for coverage collection: */
	void				*kcov_area;

	/* KCOV descriptor wired with this task or NULL: */
	struct kcov			*kcov;
#endif

#ifdef CONFIG_MEMCG
	struct mem_cgroup		*memcg_in_oom;
	gfp_t				memcg_oom_gfp_mask;
	int				memcg_oom_order;

	/* Number of pages to reclaim on returning to userland: */
	unsigned int			memcg_nr_pages_over_high;

	/* Used by memcontrol for targeted memcg charge: */
	struct mem_cgroup		*active_memcg;
#endif

#ifdef CONFIG_BLK_CGROUP
	struct request_queue		*throttle_queue;
#endif

#ifdef CONFIG_UPROBES
	struct uprobe_task		*utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
	unsigned int			sequential_io;
	unsigned int			sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
	unsigned long			task_state_change;
#endif
	int				pagefault_disabled;
#ifdef CONFIG_MMU
	struct task_struct		*oom_reaper_list;
#endif
#ifdef CONFIG_VMAP_STACK
	struct vm_struct		*stack_vm_area;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/* A live task holds one reference: */
	atomic_t			stack_refcount;
#endif
#ifdef CONFIG_LIVEPATCH
	int patch_state;
#endif
#ifdef CONFIG_SECURITY
	/* Used by LSM modules for access restriction: */
	void				*security;
#endif
```

```c
#ifdef CONFIG_GCC_PLUGIN_STACKLEAK
    unsigned long           lowest_stack;
    unsigned long           prev_lowest_stack;
#endif

    /*
     * New fields for task_struct should be added above here, so that
     * they are included in the randomized portion of task_struct.
     */
    randomized_struct_fields_end

    /* CPU-specific state of this task: */
    struct thread_struct        thread;

    /*
     * WARNING: on x86, 'thread_struct' contains a variable-sized
     * structure.  It *MUST* be at the end of 'task_struct'.
     *
     * Do not put anything below here!
     */
};
```

5.4.2 进程标识符PID

为了方便普通用户定位每个进程，操作系统为每个进程分配了一个唯一的正整数标识符，称为**进程ID（PID）**。在Linux中，进程之间存在着亲缘关系，如果一个进程在执行过程中启动了另外一个进程，那么启动者就是**父进程**，被启动者就是**子进程**。从`task_struct`声明可知，进程的信息中包含它的进程ID和父进程ID，实际上PID和PCB之间存在着一一对应的关系。使用ps命令可以查看进程相关信息。

```
$ps -l
# 这里会出现两个进程，一个进程是shell，而另一个进程ps。通过PPID可知，ps的父进程就是bash
```

在Linux启动时，如果所有的硬件已经配置好的情况下，进程0会被bootloader程序启动起来，它会配置实时时钟，启动**init进程（进程1）**和页面守护进程（进程2）。init就是所谓的“盘古”进程（在新版本中被systemd取代），它会启动shell进程。在多用户的情况下，init会开启运行/etc/rc中配置的脚本进程，然后这个进程再从`/etc/ttys`中读取数据。`/etc/ttys`中列出了所有的终端，终端可以用于让用户从某种渠道登陆操作系统。

使用系统调用`getpid`和`getppid`可以获取当前运行进程的进程ID和父进程ID。

```c
//getpid.c
#include <func.h>
int main(){
    printf("getpid = %d, getppid = %d\n", getpid(), getppid());
}
//运行以后可以通过ps命令查到其父进程就是bash
```

5.4.3 进程的用户ID和组ID

进程在运行过程中，存在一个用户身份的属性，以便于控制进程的权限。在默认情况下，程序进程拥有启动用户的身份。例如，假设当前登录用户为student，他运行了任意一个程序（无论是不是他创建的），则程序在运行过程中就具有student的身份，该进程的用户ID和组ID分别为student和student所属的组，其ID和组ID就被称为进程的真实用户ID和真实组ID。真实用户ID和真实组ID可以通过函数`getuid()`和`getgid()`获得。

```c
//getuid.c
#include <func.h>
int main(){
  uid_t uid;
  gid_t gid;
  uid = getuid();
  gid = getgid();
  printf("uid = %d, gid = %d\n",uid,gid);
}
//使用不同的用户启动进程会得到不同的结果
```

与真实ID对应，进程还具有**有效用户ID**和**有效组ID**的属性，内核对进程的访问权限检查时，它检查的是进程的有效用户ID和有效组ID，而不是真实用户ID和真实组ID。默认情况下，用户的有效用户ID和真实ID是相同的，有效组ID和真实组ID是相同的。有效用户ID和有效组ID通过函数`geteuid()`和`getegid()`获得。

```c
//geteuid.c
#include <func.h>
int main(){
  uid_t euid;
  gid_t egid;
  uid = geteuid();
  gid = getegid();
  printf("euid = %d, egid = %d\n",euid,egid);
}
//默认情况下uid和euid一致
```

passwd命令的设计原理

Linux操作系统的密码存储在文件`/etc/shadow`当中，这个文件的拥有者是root，所在组是shadow。所以如果需要查看文件内容，需要将当前用户加入shadow组(可以修改/etc/group文件)。而只有root才拥有文件的写权限。当拥有读权限的时候，可以使用cat命令或者vim查看shadow文件时，发现密码都是采用密文存储。

```
$cat /etc/passwd
#将自己的密码密文记录下来
$passwd username
#修改密码
$cat /etc/passwd
#发现密码密文被修改了
```

所上述操作的结果可以知道，如果一个用户想要使用passwd命令修改自己的密码，显然他需要通过一定的方式获取shadow文件的写权限。passwd进程在执行的时候，它会将自己的有效用户ID调整为root，从而拥有了对文件的写权限。

参看passwd的设计，我们自己也可以实现让另一个用户在运行进程时，修改自己的有效用户ID或者有效组ID。

```
//changeFile.c
#include <func.h>
int main(int argc, char *argv[]){
  ARGS_CHECK(argc,2);
  printf("euid = %d, egid = %d\n",geteuid(),getegid());
  int fd = open(argv[1],O_RDWR);
  ERROR(fd,-1,"open");
  write(fd,"hello",5);
}
```

如果将上述程序直接编译，切换用户以后将无法执行，但是如果给可执行程序加上s权限以后，程序就能运行了。具体的运行原理就是进程在运行过程中修改了它的有效用户ID。

```
$./changeFile file1
#使用其他用户无法打开文件
$chmod u+s changeFile
#给程序加上s权限以后，文件就可以打开了，此时的有效用户ID已经被修改了
```

使用ls -l命令可以检查passwd命令和sudo命令的权限，会发现它们都拥有s权限。

```
$which passwd
#检查passwd命令所在的位置
$ls -l /etc/passwd
$ls -l /etc/sudo
```

sudo的实现原理

sudo给用户添加权限的方法自然时在运行时修改有效用户ID。那么sudo命令是怎么控制哪些普通用户可以使用sudo的呢?

sudo命令在执行的时候会检查/etc/sudoers文件，只有文件存在的用户才能使用sudo命令。除此以外sudoers可以给用户配置某些命令的sudo权限。具体的配置规则可以查询手册。

5.4.4 文件特殊权限

文件的权限是由12位来控制的，其中最低的9位自然是我们所熟知的普通权限。而在最高3位中，最高位和次高位就是上面所述的s权限。

- 对于一个二进制程序，如果最高位为1，且用户的执行权限也为1时，那么该程序运行时将修改自己的有效用户ID为文件拥有者。最高位权限又称为SUID权限。
- 对于一个二进制程序，如果次高位为1，且组的执行执行也为1时，那么该程序运行时将修改自己的有效组ID为文件用户组ID。次高位权限又称为SGID权限。
- 对于一个目录文件，如果它设置了SGID权限后，如果当前用户拥有此目录的r和x的权限，则可以进入该目录。若用户拥有目录的w权限，则该用户新建的文件会和此目录的组一致。

```
$mkdir dir1
$chmod g+s dir1
$chmod o+w dir1
#再切换用户，cd到dir1中创建文件，会发现文件的组ID和目录组ID一致
```

- 对于一个目录文件，如果它的权限第3高位为1，也就是所谓的粘滞位（sticky bit或SBIT）为1时，并且用户拥有对目录的w和x权限的时候，用户可以往其中添加文件，但是这个文件只有自己和root用户才能删除。

```
$mkdir dir2
$chmod o+w dir2
$chmod o+t dir2
#再切换用户，cd到dir2中创建文件，然后切换回另一个用户
$rm file
#虽然当前用户拥有对文件的写权限，但是rm命令执行会失效
#/tmp目录就拥有t权限
```

除了可以使用文字设定法，也可以采用数字设定法来设置特殊权限。

5.5 进程的状态

在进程从创建到消亡的过程中，进程会存在很多种状态。其中最基本的三种的状态是执行态、就绪态、等待态

- 执行态：该进程正在运行，即进程正在占用CPU。
- 就绪态：进程已经具备执行的一切条件，正在等待分配CPU的处理时间片。
- 等待态：进程不能使用CPU，通常由于等待IO操作、信号量或者其他操作。

除了最基本的上面3种状态以外，使用ps命令还可以观察到一些更细致划分的状态

```
$ps -elf
#找到第二列，也就是列首为S的一列
#R 运行中
#S 睡眠状态，可以被唤醒
#D 不可唤醒的睡眠状态，通常是在执行IO操作
#T 停止状态，可能是被暂停或者是被跟踪
#Z 僵尸状态，进程已经终止，但是无法回收资源
```

5.7 进程相关的命令

命令名	详细信息
ps	查看系统当中的进程
top	动态显示系统当中的进程
nice	用于shell脚本中，指定程序的优先级
renice	改变正在运行进程的优先级
kill	发送信号（可以给后台进程发送）
crontab	控制cron后台进程
bg	将暂停的进程放到后台

5.7.1 ps命令

ps命令的选项很多很复杂，具体的细节可以使用man命令查看手册，不过工作中使用最多的是两种：`ps -elf`和`ps aux`

UNIX风格：ps -elf

```
$ps -elf
#第一列是F 表示进程标识，通常用4来表示root权限，1来表示只有拷贝没有执行
#第二列是S 表示运行状态
#随后三列是UID/PID/PPID 表示有效用户ID、进程ID和父进程ID
#随后是C 表示CPU占用百分比
#随后是PRI/NI 表示优先级和nice值，用来分配时间片
#随后是ADDR/SZ/WCHAN ADDR表示进程在内存的哪个部分（- 表示用户态），SZ表示占用了多少物理内存页（包括数据段、代码段和栈）WCHAN 表示睡眠进程正在执行的内核函数的名字
#随后是TTY 描述登录的终端，远程终端是pts/编号
#随后是TIME 表示占用CPU的总时间
#随后是CMD 表示触发进程的命令是什么
```

BSD风格：ps aux

```
$ps aux
#首列是USER 表示有效用户
#随后是PID 表示进程ID
#随后是%CPU 表示CPU资源百分比
#随后是%MEM 表示占用物理内存百分比
#随后是VSZ 表示占用的虚拟内存量
#随后是RSS 表示占用的固定内存量（内存驻留集即未交换的内存的大小）
#随后是TTY 表示运行终端 本机登录进程是tty1～6 网络连接是pts/n
#随后是STAT 表示进程状态
#随后是START 表示启动时间
#随后是TIME 表示CPU占用时间
#随后是COMMAND 表示进程触发命令
```

使用`free`命令也可以查看系统的内存占用信息：

```
$free
```

5.7.2 top命令

```
top - 16:24:25 up 284 days, 4:59, 1 user, load average: 0.10, 0.05, 0.01
Tasks: 115 total,  1 running, 114 sleeping,  0 stopped,  0 zombie
Cpu(s): 0.1%us, 0.0%sy, 0.0%ni, 99.8%id, 0.0%wa, 0.0%hi, 0.1%si, 0.0%st
Mem:  4074364k total, 3733628k used,  340736k free,  296520k buffers
Swap: 2104504k total,  40272k used, 2064232k free,  931680k cached


PID USER    PR NI VIRT RES SHR S %CPU %MEM  TIME+ COMMAND
11836 root   15  0 2324 1028 800 R 0.3 0.0  0:00.02 top
27225 root   25  0 1494m 696m 11m S 0.3 17.5  2304:03 java
1 root   18  0 2072 620 532 S 0.0 0.0  7:04.48 init
```

第一行分别显示：

```
16:24:25当前时间、
up 284 days, 4:59系统启动时间、
1 user当前系统登录用户数目、
load average: 0.10, 0.05, 0.01平均负载（1分钟,10分钟,15分钟）。
```

平均负载（load average）,一般对于单个cpu来说，负载在0～1.00之间是正常的，超过1.00须引起注意。在多核cpu中，系统平均负载不应该高于cpu核心的总数。

第二行分别显示：

```
115 total进程总数
1 running运行进程数
114 sleeping休眠进程数
0 stopped终止进程数
0 zombie僵死进程数
```

第三行：

```
0.1%us     %us用户空间占用cpu百分比；
0.0%sy     %sy内核空间占用cpu百分比；
0.0%ni     %ni用户进程空间内改变过优先级的进程占用cpu百分比；
99.8%id     %id空闲cpu百分比，反映一个系统cpu的闲忙程度。越大越空闲；
0.0%wa     %wa等待输入输出（I/O）的cpu百分比；
0.0%hi     %hi指的是cpu处理硬件中断的时间；
0.1%si     %si值的是cpu处理软件中断的时间；
0.0%st     %st用于有虚拟cpu的情况，用来指示被虚拟机偷掉的cpu时间。
```

第四行（Mem）：

```
4074364k total     total总的物理内存；
3733628k used used使用物理内存大小；
340736k free free空闲物理内存；
296520k buffers     buffers用于内核缓存的内存大小
```

第五行（Swap）：

```
2104504k total     total总的交换空间大小；
40272k used  used已经使用交换空间大小；
2064232k free free空间交换空间大小；
931680k cached     cached缓冲的交换空间大小
```

buffers与cached区别：buffers指的是块设备的读写缓冲区，cached指的是文件系统本身的页面缓存。他们都是Linux系统底层的机制，为了加速对磁盘的访问。

然后下面就是和ps相仿的各进程情况列表了

第六行：

```
PID 进程号
USER 运行用户
PR优先级，PR(Priority)优先级
NI   任务nice值
VIRT  进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES
RES 物理内存用量
SHR 共享内存用量
S 该进程的状态。其中S代表休眠状态；D代表不可中断的休眠状态；R代表运行状态；Z代表僵死状态；T代表停止或跟踪状态
%CPU 该进程自最近一次刷新以来所占用的CPU时间和总时间的百分比
%MEM 该进程占用的物理内存占总内存的百分比
TIME+ 累计cpu占用时间
COMMAND 该进程的命令名称，如果一行显示不下
```

5.7.3 优先级和nice值

Linux的优先级总共的范围有140，对于ubuntu操作系统而言，其范围是-40到99，优先级的数值越低，表示其优先级越高。

Linux中拥有两种类型的调度策略，分别是**实时调度策略**和**普通调度策略。**

普通调度策略又称为OTHER策略，其调度规则即CFS算法。普通调度策略**不会一定**保证某个进程会在规定时间执行。普通调度策略的优先级是从60到99范围之间。在ubuntu系统中，当一个普通进程创建时，其默认优先级是80。普通调度策略优先级的调整是依赖于nice值的。nice值可以用来调整优先级，其范围为-20～19。其中正数表示降低权限，负数表示提升权限。root用户可以任意地修改进程的nice值，其他用户只能提升自己的进程的nice值。使用nice命令和renice命令可以用来调整nice值。

```
$nice -n 10 ./a.out #注意之后不能使用renice调整它的优先级
$renice -5 pid #执行失败
$renice 5 pid #执行成功，普通用户只能提升nice
```

实时调度策略是针对于实时进程，这些实时进程对于时间延迟非常敏感（想象下如果航天飞机的指令出现延迟造成的灾难性后果），所以普通调度策略不足以满足实时性需求。Linux的实时调度策略有两种，分别是RR和FIFO。其中FIFO以按照先进先出的方式运行进程，除非主动退出，它不会被同级或着更低优先级的进程抢占，只能被更高优先级的进程抢占；RR在FIFO的基础上增加时间片管理，相同优先级的进程会分配相同的时间片，而低优先级的进程无法抢占高优先级的进程，即使高优先级的进程时间片耗尽。只有系统调用`sched_getscheduler`和`sched_setscheduler`才能修改调度策略，使用`nice`系统调用（以及基于它的同名命令）或者`setpriority`

系统调用这种修改优先级数值的方法无法改变调度策略。

```
$sudo renice -21 pid
#使用renice最多只能降低20
```

5.7.4 kill命令和任务控制

`kill`命令可以用来给指定的进程发送信号。

通常用户经常会从终端启动shell再启动进程，当进程正在运行时，它可以接受一些键盘发送的信号：比如`ctrl+c`表示终止信号，`ctrl+z`表示暂停信号。这种可以直接接受键盘信号的状态被称为**前台**，否则称为后台。当进程处于后台的时候，只能通过`kill`命令发送信号给它。

```
$kill -9 pid
# 以异常方式终止进程
$kill -l
# 显示所有信号
```

使用shell启动进程的时候如果在末尾加上`&`符号可以用来直接运行后台进程。

```
$vim test &
#它会输出一个整形数字，表示它的任务编号
```

使用`ctrl+z`可以暂停当前运行的前台进程，并将其放入后台。它也会输出一个任务编号到屏幕上。

使用`jobs`命令可以查看和管理所有的后台任务，使用`fg`命令可以将后台进程拿到前台来。使用`bg`命令可以将后台暂停的程序运行起来。

5.7.5 crontab设置计划任务

`crontab`可以实现定期执行任务

使用`crontab -e`然后选择合适的文本编辑器（这种情况任务计划放置在`/var/spool/cron/crontab`下面）或者直接用root权限打开`/etc/crontab`文件即可。

每项任务拥有六个字段，分别表示分钟（0～60）、小时（0～23）、日期（1～31）、月份（1～12）、周（0～7 0和7都代表周日）和要执行的命令。如果是修改文件，还需要在执行命令的前面添加用户名。

```
#比如希望给一个女生每年5月20日写入一个信息
* * 20 5 * echo "I love you">>toMay
```

特殊字符	含义
*	任意时刻
,	A,B A时刻或者B时刻
-	A-B A时刻到B时刻
*/n	A-B/n 从A时刻开始，到B时刻为止，每隔n个单位 */n 从0到结束，每隔n个单位

5.8 使用系统调用创建进程

5.8.1 system函数

```
 #include <stdlib.h>

int system(const char *command);
```

`system`函数可以创建一个新进程，新进程使用shell脚本执行传入的命令command。

```c
//system.c
#include <func.h>
int main(){
  system("sleep 20");
  return 0;
}
```

如果在程序执行过程使用`ps`命令查看所有进程，我们会发现创建了3个进程，并且3个进程之间存在父子亲缘关系。由于创建进程的时间消耗是很大的，对于性能要求比较苛刻的任务来说，这种使用`system`的方式往往是不能被接受的。

除了可以执行shell指令，`system`函数还可以嵌入其他编程语言所编写的程序，比如`python`：

```python
#文件名为hello.py
print("hello")
```

```c
//systemPy.c
int main(){
  //需要在操作系统上装好python3解释器
  system("python3 hello.py");
  return 0;
}
```

5.8.2 fork函数

`fork`用于拷贝当前进程以创建一个新进程。

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

`fork`执行以后创建的新进程和当前进程拥有着几乎一致的用户态地址空间。新进程和原进程之间存在一些小小的差异：

- fork系统调用的返回结果不一样，子进程返回值为0，父进程返回孩子的PID
- 父子进程之间的PPID也不一样，其中子进程的PPID的进程为它的父进程

```c
//fork.c
#include <func.h>
int main(){
    pid_t pid = fork();
  if(pid == 0){
        printf("I am child, pid = %d\n", pid);//操作系统不保证进程的执行先后顺序，不过通常进程创建大约在数百毫秒量级
  }
  else{
        printf("I am parent, pid = %d\n", pid);
    //sleep(1); 如果不添加sleep，将会出现一些显示异常
  }
}
```

fork的实现原理

首先，我们需要引入中断的概念。所谓**中断**，就是利用硬件发送信息给CPU，然后CPU通知操作系统处理中断事宜。从这个角度上来看中断的处理是**异步**的，即当前进程的执行指令和中断发生顺序是未知的。不同类型的中断可以使用唯一的中断号进行区分，操作系统为不同类型的中断提供了对应的中断处理程序。

异常是一个类似于中断的概念，它是进程主动发送给CPU的信息，所以也被称为**软件中断**。在x86体系结构当中，**系统调用**就是利用了软件中断（这个软件中断名为**陷入**）实现的。除了信息的来源不同以外，操作系统处理中断和异常的流程是一样的。

中断处理程序的设计目标有两个：就是运行时间短，并且完成任务多。而显然这两个设计目标之间存在矛盾，一种解决方案是将中断处理分成两个部分：**上半部**和**下半部**。上半部要完成一些时间短并且不能延迟的工作，比如调整硬件、或者是不能被抢占的指令。以网卡为例子：上半部阶段，网卡会给内核发送信号，内核会执行中断处理程序，然后将网卡的内容拷贝到系统内存。下半部会要完成一些可以稍后执行的指令，比如网卡中断的下半部中，内核才会处理数据包的解析和其他处理操作。

现在我们回到`fork`系统调用，既然是系统调用，它自然可以分为上半部和下半部，上半部当中，它会"拷贝"一份进程控制块，自然也就拷贝了一份地址空间（包括进程的正在执行的状态和指令（PC）），然后修改子进程的`task_struct`的内容，将PPID和PID进行调整。随后，它将子进程放入就绪度列等待调度，并将子进程`fork`的（将要返回的）返回值修改为**0**，父进程的返回值设置为**子进程的PID**。上半部过程中，`fork`的执行过程是**不能被抢占**的，所以能保证一定能执行完成。随后`fork`要执行它的下半部，那就是将其返回值返回给两个进程，随后修改PC指针，让各个进程继续执行后续的命令。

fork的资源

通过`fork`创建的子进程，它从父进程继承了进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组ID、当前工作目录、根目录、资源限制、控制终端，而子进程所独有的只有它的进程ID、资源使用和计时器等。

```
//forkStack.c
#include <func.h>
int main(){
    pid_t pid = fork();
  int i = 0;
  if(pid == 0){
        puts("child");
    printf("child i = %d,&i = %p\n",i, &i);
    ++i;
    printf("child i = %d,&i = %p\n",i, &i);
  }
  else{
        puts("parent");
    printf("parent i = %d,&i = %p\n",i,&i);
    sleep(1);
    printf("parent i = %d,&i = %p\n",i,&i);//子进程会拷贝父进程的内容，但是修改的内容会互相独立
  }
}
//父子进程中的变量包括其地址是一致
```

```
//forkStack.c
#include <func.h>
int main(){
```

```c
    pid_t pid = fork();
  char *p = (char *)malloc(6);
  strcpy(p,"hello")
  if(pid == 0){
        puts("child");
    printf("child p = %s,p = %p\n",p,p);
    strcpy(p,"world")
    printf("child p = %s,p = %p\n",p,p);
  }
  else{
        puts("parent");
    printf("parent p = %s,p = %p\n",p,p);
    sleep(1);
    printf("parent p = %s,p = %p\n",p,p);//和栈空间的表现差不多
  }
}
```

```c
//forkGlobal.c
#include <func.h>
int i = 0;
int main(){
    pid_t pid = fork();
  if(pid == 0){
        puts("child");
    printf("child i = %d,&i = %p\n",i, &i);
    ++i;
    printf("child i = %d,&i = %p\n",i, &i);
  }
  else{
        puts("parent");
    printf("parent i = %d,&i = %p\n",i,&i);
    sleep(1);
    printf("parent i = %d,&i = %p\n",i,&i);//数据段也是类似的
  }
}
```

fork的写时复制

当执行了`fork`了以后，父子进程地址空间的内容是完全一致，所以完全可以共享同一片物理内存，也就是父子进程的同一个虚拟地址会对应同一个物理内存字节。通常来说，内存的分配单位是页，我们可以为每一个内存页维持一个引用计数。代码段的部分因为只读，所以完全可以多个进程同时共享。而对于地址空间的其他部分，当进程对某个内存页进行写入操作的时候，我们再真正执行被修改的虚拟内存页分配物理内存并拷贝数据，这就是所谓的**写时复制**。在执行拷贝以后，同样的虚拟地址就无法对应同样物理内存字节了。

fork对打开文件的影响

内核态地址空间拷贝和用户态会有所区别。`fork`产生的子进程会拷贝一份文件描述符数组，但是通过文件描述符所指向的文件对象是共享的。这种拷贝方式类似于`dup`系统调用，所以父子进程对同一个文件对象会共享读写位置。

```c
//forkFile
#include <func.h>
int main()
```

```
{
    int fd = open("file", O_RDWR);//特别注意，文件打开要在fork之前。
    ERROR_CHECK(fd,-1,"open");
    pid_t pid = fork();
    if(pid == 0){
        printf("I am child process\n");
        //lseek(fd,5,SEEK_SET);
        write(fd,"hello",5);
    }
    else{
        printf("I am parent process\n");
        sleep(1);
        char buf[6] = {0};
        read(fd,buf,5);
        printf("I am parent process, buf = %s\n", buf);
    }
}
```

父子进程共享偏移量在文件是标准输出的时候非常有效，这样可以在一个终端界面上显示父子进程的输出。如果尝试对一个文件使用多进程读写，由于共享偏移量的原因，速度并不会更快。此外，文件读写的瓶颈是磁盘的读写效率，所以即便父子进程设法不共享偏移量，对文件读写的速度提升也不会有效果。

5.8.3 exec函数族

`exec*` 是一系列的系统调用。它们通常适用于在 `fork` 之后，将子进程的指令部分进行替换修改。当进程执行到 `exec*` 系统调用的时候，它会将传入的指令来取代进程本身的代码段、数据段、栈和堆，然后将PC指针重置为新的代码段的入口。 `exec*` 当中包括多个不同的函数，这些函数之间只是在传入参数上面有少许的区别。

```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*, (char *)0, char *const
envp[] */);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

在这些函数参数当中，path表示可执行文件路径，file表示可执行文件名字（file只能在PATH环境变量指定的目录下查找文件）。函数名当中的l表示列表list的含义，它要求传入可变数量的参数，并且每个参数对应一个命令行参数，最后以0结尾。函数名当中的v表示向量vector的含义，它要求传入一个指针数组，数组中的每个元素指向同一个字符串的不同位置，表示不同的命令行参数。函数名当中的e表示环境（比如PATH）的含义，它要求传入表示环境变量的指针数组。

```
//被执行的程序代码 名为add.c
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int i1 = atoi(argv[1]);
    int i2 = atoi(argv[2]);
    printf("%d + %d = %d\n",i1,i2,i1+i2);
    return 0;
}
```

```
//调用exec的程序代码
#include <func.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        printf("I am child\n");
        //execl("./add","add","3","4",(char *)0);
        char *const argv[] = {"add","3","4",NULL};
        execv("./add",argv);
        printf("you can not see me!\n");//这句话并不会打印
    }
    else{
        printf("I am parent\n");
        printf("you can see me!\n");
        sleep(1);
    }
    return 0;
}
```

实际上，我们之前所使用的`system`函数以及从bash或者是其他shell启动进程的本质就是`fork+exec`。

5.9 进程控制

5.9.1 孤儿进程

如果父进程先于子进程退出，则子进程成为**孤儿进程**，此时将自动被PID为1的进程（即init）收养。当一个孤儿进程退出以后，它的资源清理会交给它的父进程（此时为init）来处理。

```
//orphan.c
#include <func.h>
int main()
{
    pid_t pid =fork();
    if(pid == 0){
        printf("I am child\n");
        while(1);
    }else{
        printf("I am parent\n");
        return 0;//在main函数中执行return语句是退出进程
    }
}
//随后可以使用ps -elf|grep orphan 查看子进程的父进程ID就是1
//也可以在代码中使用getppid()
```

5.9.2 僵尸进程

如果子进程先退出，系统不会自动清理掉子进程的环境，而必须由父进程调用`wait`或`waitpid`函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为**僵尸进程(defunct)**，在系统中如果存在的僵尸（zombie）进程过多，将会影响系统的性能，所以必须对僵尸进程进行处理。

```
//zombie.c
#include <func.h>
```

```
int main()
{
    pid_t pid =fork();
    if(pid == 0){
        printf("I am child\n");
        return 0;
    }else{
        printf("I am parent\n");
        while(1);
    }
}
//随后使用ps -elf|grep zombie 可以看到一个<defunct>的僵尸标记
```

当一个进程执行结束时，它会向它的父进程发送一个SIGCHLD信号，从而父进程可以根据子进程的终止情况进行处理。在父进程处理之前，内核必须要在进程队列当中维持已经终止的子进程的PCB。如果僵尸进程过多，将会占据过多的内核态空间。并且僵尸进程的状态无法转换成其他任何进程状态。

5.9.3 wait和waitpid

`wait`和`waitpid`系统调用都会阻塞父进程，等待一个已经退出的子进程，并进行清理工作；`wait`随机地等待一个已经退出的子进程，并返回该子进程的PID；`waitpid`等待指定PID的子进程；如果为-1表示等待所有子进程。

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

```
#include <func.h>
int main()
{
    pid_t pid =fork();
    if(pid == 0){
        printf("I am child,pid = %d, ppid = %d\n",getpid(),getppid());
        return 0;
    }else{
        printf("I am parent\n");
        pid_t cpid;
        cpid = wait(NULL);
        printf("cpid = %d\n", cpid);
        return 0;
    }
}
```

`stat_loc`参数是一个整型指针。如果不关心进程的退出状态，那么该参数可以是一个空指针；否则`wait`函数会将进程终止的状态存入参数所指向的内存区域。这个整型的内存区域中由两部分组成，其中一些位用来表示退出状态（当正常退出时），而另外一些位用来指示发生异常时的信号编号，有4个宏可以用来检查状态的情况。

宏	说明
WIFEXITED(status)	子进程正常退出的时候返回真，此时可以使用WEXITSTATUS(status)，获取子进程的返回情况

宏	说明
WIFSIGNALED(status)	子进程异常退出的时候返回真，此时可以使用WTERMSIG(status)获取信号编号，可以使用WCOREDUMP(status)获取是否产生core文件
WIFSTOPPED(status)	子进程暂停的时候返回真，此时可以使用WSTOPSIG(status)获取信号编号
WIFCONTINUED(status)	只适用于waitpid，子进程暂停后恢复时返回真

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    int status;
    if(pid == 0){
        printf("child, pid = %d, ppid = %d\n", getpid(),getppid());
        char *p = NULL;
        *p = 'a';
        return 123;
    }
    else{
        printf("parent, pid = %d, ppid = %d\n", getpid(),getppid());
        //wait(&status);
        waitpid(pid,&status,0);//第三个参数为0，和直接使用wait没有说明区别
        if(WIFEXITED(status)){
            printf("child exit code = %d\n", WEXITSTATUS(status));
        }
        else if(WIFSIGNALED(status)){
            printf("child crash, signal = %d\n",WTERMSIG(status));
        }
    }
    return 0;
}
```

默认情况下，`wait`和`waitpid`都会使进程处于阻塞状态，也就是执行系统调用时，进程会中止运行。如果给`waitpid`的options参数设置一个名为WNOHANG的宏，则系统调用会变成非阻塞模式：当执行这个系统调用时，进程会立刻检查是否有子进程发送子进程终止信号，如果没有则系统调用立即返回。

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    int status = 0;
    if(pid == 0){
        printf("child, pid = %d, ppid = %d\n", getpid(),getppid());
        sleep(5);
        return 123;
    }
    else{
        printf("parent, pid = %d, ppid = %d\n", getpid(),getppid());
        int ret = waitpid(pid,&status,WNOHANG);
        if(ret > 0){
```

```
            if(WIFEXITED(status)){
                printf("child exit code = %d\n", WEXITSTATUS(status));
            }
            else if(WIFSIGNALED(status)){
                printf("child crash, signal = %d\n",WTERMSIG(status));
            }
        }
        printf("ret = %d\n",ret);
    }
    return 0;
}
```

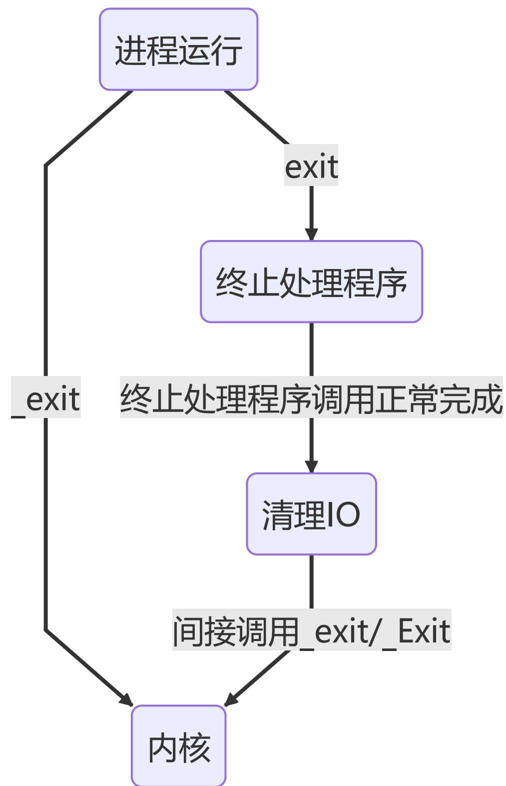
`waitpid`除了可以等待指定子进程以外，它还可以修改pid参数来支持更多种模式的等待方式。

5.10 进程的终止

在进程阶段，进程总共有5种终止方式，其中3种是正常终止，还有2种是异常终止：

终止方式	终止情况
在main函数调用return	正常
调用exit函数	正常
调用_Exit函数或者_exit函数	正常
调用abort函数	异常
接受到能引起进程终止的信号	异常

`exit`函数、`_Exit`函数和`_exit`函数可以立刻终止进程，无论当前进程正在执行什么函数。注意，和一般的函数不同，调用这3个退出函数是没有返回返回值这个过程的。当调用`_exit`和`_Exit`的时候，进程会直接终止返回内核，而`exit`它的实现会多一些额外步骤，它会首先执行终止处理程序（使用`atexit`函数可以注册终止处理程序），然后清理标准IO（就是把所有打开的流执行一次`fclose`），最后再终止进程回到内核。




```c
#include <stdlib.h>
void exit(int status);//可以看出exit函数是由ISO C规定的
#include <unistd.h>
void _exit(int status;//_exit是一个Linux系统调用
#include <stdlib.h>
void _Exit(int status);//_Exit是ISO C规定的库函数
```

```c
//return和exit的区别
#include <func.h>
int  func(){
    printf("func\n");
    //return 3;
    exit(3);
}
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        puts("child");
        func();
        return 0;
    }
    else{
        puts("parent");
        int status;
        wait(&status);
        if(WIFEXITED(status)){
            printf("child exit code = %d\n",WEXITSTATUS(status));
        }
```

```
    }
    return 0;
}
```

如果在程序没有注意缓冲区，并且又使用了`_exit`或者是`_Exit`的话，很容易出现缓冲区内容丢失的情况。

```
//_exit.c
#include <func.h>
int  func(){
    printf("func");//这里不要添加换行符
    //exit(3);
    _exit(3);//这种情况func不会打印
}
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        puts("child");
        func();
        return 0;
    }
    else{
        puts("parent");
        int status;
        wait(&status);
        if(WIFEXITED(status)){
            printf("child exit code = %d\n",WEXITSTATUS(status));
        }
    }
    return 0;
}
```

当进程处于前台的时候，按下`ctrl+c`或者是`ctrl+\`可以给整个进程组发送键盘中断信号SIGINT和SIGQUIT。

使用`abort`可以主动调用异常终止。

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    int status;
    if(pid == 0){
        printf("child, pid = %d, ppid = %d\n", getpid(),getppid());
        abort();
    }
    else{
        printf("parent, pid = %d, ppid = %d\n", getpid(),getppid());
        waitpid(pid,&status,0);
        if(WIFEXITED(status)){
            printf("child exit code = %d\n", WEXITSTATUS(status));
        }
        else if(WIFSIGNALED(status)){
```

```
            printf("child crash, signal = %d\n",WTERMSIG(status));
        }
    }
    return 0;
}
//使用kill -l 可以得知6号信号就是SIGABRT
```

5.11 守护进程

大致上来说，所谓**守护进程(daemon)**，就是在默默运行在后台的进程，也称作"后台**服务**进程"，通常守护进程的命名会以d结尾。为了更准确地把握守护进程的概念，需要先了解一些其他的概念。

5.11.1 终端

终端是登录到Linux操作系统所需要的入口设备。终端可以是本地的，也可以是远程的。当操作系统启动的时候，init进程会创建子进程并使用`exec`来执行getty程序，从而打开终端设备或者等待远程登录，然后再使用`exec`调用login程序验证用户名和密码。

5.11.2 进程组

每个进程除了有一个进程ID以外，还属于一个**进程组**。前文曾经提到过，使用键盘中断给前台进程发送信号的时候，是会给一个进程组的所有进程来发送信号的。进程组是一个或者多个进程构成的集合。不同的进程组拥有不同的进程组ID，每个进程组有一个组长进程，组长的PID就是进程组ID。进程组组长可以创建一个进程组、创建组中进程，但是只要进程组当中存在至少一个进程（这个进程可以不是组长），该进程组就存在。

```
#include <unistd.h>
pid_t getpgrp(void);//获取进程组ID
pid_t getpgid(pid_t pid);//获取PID为pid的进程的进程组ID，如果pid为0，则获取本进程所属进程组ID
```

当使用shell运行程序创建进程的时候，被创建进程是shell的子进程，并且这个进程将会创建一个进程组，再使用`fork`派生的进程都属于这个进程组。

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        printf("child, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        exit(0);
    }
    else{
        printf("parent, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        wait(NULL);
        exit(0);
    }
}
```

进程`fork`产生一个子进程以后，子进程默认和父进程属于同一个进程组。`setpgid`系统调用可以用来修改进程或者是`exec`之前的子进程的进程组ID。

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);//将pid进程的进程组ID设置为pgid
//如果pid为0，使用调用者的进程ID
//如果pgid为0，则进程组ID和pid一致
```

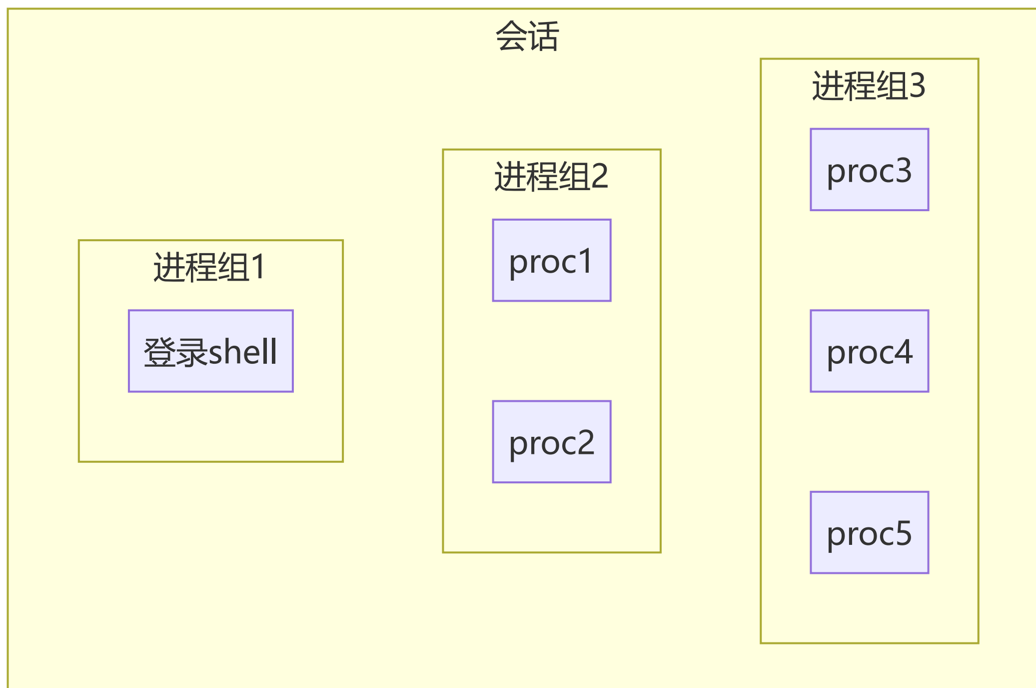
```
#include <func.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        printf("child, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        setpgid(0,0);
        printf("child, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        while(1);
        exit(0);
    }
    else{
        printf("parent, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        while(1);
        wait(NULL);
        exit(0);
    }
}
```

如果使用`setpgid`修改进程组，那么再次使用`ctrl+c`触发键盘中断信号的时候，将只会终止父进程。

5.11.3 会话

会话是一个或者多个进程组的集合。创建新会话的进程被称为新会话的**会话首进程**，会话首进程的PID就是会话ID。shell当中的管道可以用来构造如下所示的会话。

```
$proc1|proc2 &
$proc3|proc4|proc5
```



会话拥有一些特征：

- 一个会话可以有一个**控制终端。**
- 和控制终端建立连接的会话首进程被称为**控制进程。** （通常登录时会自动连接，或者使用`open`打开文件`/dev/tty`）
- 一个会话存在最多一个**前台进程组**和**多个后台进程组**，如果会话和控制终端相连，则必定存在一个前台进程组。
- 从终端输入的中断，会将信号发送到前台进程组所有进程
- 终端断开连接，挂断信号会发送给控制进程

对于目前不是进程组组长的进程，可以使用系统调用`setsid`可以新建一个会话。使用`getsid`可以获取会话ID。

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
pid_t getsid(pid_t pid);
```

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        printf("child, pid = %d, ppid = %d, pgid = %d, sid = %d\n", getpid(),
getppid(), getpgid(0), getsid(0));
        setpgid(0,0);
        while(1);
        exit(0);
    }
    else{
        printf("parent, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0), getsid(0));
```

```
        while(1);
        wait(NULL);
        exit(0);
    }
}
//获取会话ID的例子
```

```
#include <func.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0){
        printf("child, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        //setpgid(0,0);
        //printf("child, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        int ret = setsid();//注意不能是进程组组长
        ERROR_CHECK(ret,-1,"setsid");
        while(1);
        exit(0);
    }
    else{
        printf("parent, pid = %d, ppid = %d, pgid = %d\n", getpid(), getppid(),
getpgid(0));
        while(1);
        wait(NULL);
        exit(0);
    }
}
//创建新会话以后，即使原来的shell被关闭了，子进程依然不受任何影响。
```

5.11.4 守护进程的创建流程

- 父进程创建子进程，然后让父进程终止。
- 在子进程当中创建新会话。
- 修改当前工作目录为根目录，因为正在使用的目录是不能卸载的。
- 重设文件权限掩码为0，避免创建文件的权限受限。
- 关闭不需要的文件描述符，比如0、1、2。

```
//daemon.c
#include <func.h>

void Daemon()
{
  const int MAXFD=64;
  int i=0;
  if(fork()!=0){
    exit(0);
  } //父进程退出
  setsid();   //成为新进程组组长和新会话领导，脱离控制终端
  chdir("/");  //设置工作目录为根目录
```

```
  umask(0);   //重设文件访问权限掩码
  for(;i<MAXFD;i++){
    close(i);//尽可能关闭所有从父进程继承来的文件
  }
}

int main()
{
  Daemon(); //成为守护进程
  while(1){
    sleep(1);
  }
  return 0;
}
```

5.11.5 守护进程和日志

使用守护进程经常可以用记录日志。操作系统的日志文件存储在`/var/log/messages`中。

```
#include <syslog.h>
void syslog(int priority, const char *format, ...);
```

```
#include <func.h>
int main()
 {
   int i = 0;
   if(fork() > 0)
      exit(0);
   setsid();
   chdir("/");
   umask(0);
   for(; i < 64; i++)
   {
      close(i);
   }
   i = 0;
   while(i < 10)
   {
      printf("%d\n",i);
      time_t ttime;
      time(&ttime);
      struct tm *pTm = gmtime(&ttime);
      syslog(LOG_INFO,"%d %04d:%02d:%02d", i, (1900 + pTm->tm_year), (1 + pTm-
>tm_mon), (pTm->tm_mday));
      i++;
      sleep(2);
   }
 }
```