作者：代兴　　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

线程的含义

线程定义：1 个进程包含 0 或多个线程，进程内的线程共享文件、内存等资源，多线程模式可以提高并发性能。
线程状态：主要包括运行、就绪、阻塞、终止。有些编程语言、操作系统对线程状态做细分。
线程同步：使用锁、条件变量等实现多线程协作，主要操作为挂起线程、唤醒线程、等待线程完成。
线程栈：线程的私有内存空间，每个线程分配一个线程栈，包括用户栈、内核栈。
线程调度：CPU 切换不同的线程运行，实现负载均衡，包括主动调度、被动调度。

linux 源码中的线程定义

线程主要的属性：
线程 ID。整数。
线程状态。整数。
线程栈。分为用户栈、内核栈。
线程组。
调度器。分为 cfs、rt、dl、idle、stop 等。
内存管理。进程维度，多线程共享。
文件管理。进程维度，多线程共享。
信号管理。进程维度，多线程共享。

include/linux/sched.h
task_struct 表示任务，可以认为对应线程。这里截取部分属性。

```
struct task_struct {
    struct thread_info      thread_info;

    // ====== 任务状态 ======
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long           state;

    // ====== 内核栈 ======
    void                *stack;

    // ====== CPU 调度 ======
    /* Current CPU: */
    unsigned int            cpu;
    int             on_rq;
    int             prio;
    int             static_prio;
    int             normal_prio;
    unsigned int            rt_priority;
    const struct sched_class    *sched_class;
```

```
    struct sched_entity     se;
    struct sched_rt_entity      rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group       *sched_task_group;
#endif
    struct sched_dl_entity      dl;

    // ====== 内存 ======
    struct mm_struct        *mm;
    struct mm_struct        *active_mm;
    /* Per-thread vma caching: */
    struct vmacache         vmacache;

    // ====== 任务之间的关系 ======
    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;

    // ====== 文件 ======
    /* Filesystem information: */
    struct fs_struct        *fs;
    /* Open file information: */
    struct files_struct     *files;

    // ====== 信号 ======
    /* Signal handlers: */
    struct signal_struct        *signal;
    struct sighand_struct __rcu     *sighand;
    sigset_t            blocked;
    sigset_t            real_blocked;
    /* Restored if set_restore_sigmask() was used: */
    sigset_t            saved_sigmask;
    struct sigpending       pending;

    // ====== 线程状态 ======
    /* CPU-specific state of this task: */
    struct thread_struct        thread;
}
```

arch/x86/include/asm/switch_to.h

任务调度，找到新的任务 next，把当前的任务和新的任务做切换，CPU 就能运行新的任务。

```
#define switch_to(prev, next, last)                 \
do {                                    \
```

```
	prepare_switch_to(next);					\
									\
	((last) = __switch_to_asm((prev), (next)));		\
} while (0)
```

arch/x86/entry/entry_64.S
任务上下文切换，使用汇编代码，切换寄存器，切换函数栈。

```
/*
 * %rdi: prev task
 * %rsi: next task
 */
SYM_CODE_START(__switch_to_asm)
	UNWIND_HINT_FUNC
	/*
	 * Save callee-saved registers
	 * This must match the order in inactive_task_frame
	 */
	pushq	%rbp
	pushq	%rbx
	pushq	%r12
	pushq	%r13
	pushq	%r14
	pushq	%r15

	/* switch stack */
	movq	%rsp, TASK_threadsp(%rdi)
	movq	TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_STACKPROTECTOR
	movq	TASK_stack_canary(%rsi), %rbx
	movq	%rbx, PER_CPU_VAR(fixed_percpu_data) + stack_canary_offset
#endif

#ifdef CONFIG_RETPOLINE
	/*
	 * When switching from a shallower to a deeper call stack
	 * the RSB may either underflow or use entries populated
	 * with userspace addresses. On CPUs where those concerns
	 * exist, overwrite the RSB with entries which capture
	 * speculative execution to prevent attack.
	 */
	FILL_RETURN_BUFFER %r12, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
#endif

	/* restore callee-saved registers */
	popq	%r15
	popq	%r14
	popq	%r13
	popq	%r12
```

```
    popq    %rbx
    popq    %rbp

    jmp __switch_to
SYM_CODE_END(__switch_to_asm)
```

线程同步：多个线程交替输出序号

编写代码： thread.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdint.h>
#include <syscall.h>
#include <sched.h>
#include <stdbool.h>

#define cpuid sched_getcpu()    // CPU ID
#define pid syscall(SYS_getpid) // 进程 ID
#define tid syscall(SYS_gettid) // 线程 ID

pthread_mutex_t mutex; // 锁
pthread_cond_t cond1;  // 条件变量
pthread_cond_t cond2;  // 条件变量
pthread_cond_t cond3;  // 条件变量

volatile int curr_seq = 0;  // 当前序号
volatile int max_seq = 8;   // 最大序号
volatile bool go_on = true; // 是否继续

// 打印
void print_seq()
{
    printf(" thread = %d  cpu = %d  seq = %2d \n", tid, cpuid, curr_seq);
    ++curr_seq;             // 序号加 1
    if (curr_seq > max_seq) // 超过限制
    {
        go_on = false; // 不再继续
        printf(" thread = %d  set  go_on = false \n", tid);
    }
    sleep(1); // 休眠线程
}

void thread_created()
{
```

```
    int tmp = 1;
    printf(" thread = %d  stack_addr = %p \n", tid, &tmp);
}

// 挂起线程
int cond_wait(pthread_cond_t *cond)
{
    pthread_mutex_lock(&mutex);
    int ret = pthread_cond_wait(cond, &mutex);
    pthread_mutex_unlock(&mutex);
    return ret;
}

// 唤醒线程
int cond_notify(pthread_cond_t *cond)
{
    pthread_mutex_lock(&mutex);
    int ret = pthread_cond_broadcast(cond);
    pthread_mutex_unlock(&mutex);
    return ret;
}

// 线程 1 的函数
void *thread_func1(void *param)
{
    thread_created();
    cond_wait(&cond3); // 挂起线程
    while (1)
    {
        if (!go_on) // 不再运行
        {
            cond_notify(&cond1); // 唤醒线程
            break;
        }
        print_seq();         // 打印
        cond_notify(&cond1); // 唤醒线程
        cond_wait(&cond3);   // 挂起线程
    }
    printf(" thread = %d  exit \n", tid);
    return NULL;
}

// 线程 2 的函数
void *thread_func2(void *param)
{
    thread_created();
    cond_wait(&cond1); // 挂起线程
    while (1)
    {
```

```
        if (!go_on) // 不再运行
        {
            cond_notify(&cond2); // 唤醒线程
            break;
        }
        print_seq();         // 打印
        cond_notify(&cond2); // 唤醒线程
        cond_wait(&cond1);   // 挂起线程
    }
    printf(" thread = %d  exit \n", tid);
    return NULL;
}

// 线程 3 的函数
void *thread_func3(void *param)
{
    thread_created();
    cond_wait(&cond2); // 挂起线程
    while (1)
    {
        if (!go_on) // 不再运行
        {
            cond_notify(&cond3); // 唤醒线程
            break;
        }
        print_seq();         // 打印
        cond_notify(&cond3); // 唤醒线程
        cond_wait(&cond2);   // 挂起线程
    }
    printf(" thread = %d  exit \n", tid);
    return NULL;
}

int main()
{
    // 进程
    printf(" process = %d \n", pid);
    // 主线程
    printf(" main thread = %d \n", tid);

    // 初始化，锁、条件变量
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond1, NULL);
    pthread_cond_init(&cond2, NULL);
    pthread_cond_init(&cond3, NULL);

    // 创建线程
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func1, NULL);
```

```
    pthread_t t2;
    pthread_create(&t2, NULL, thread_func2, NULL);
    pthread_t t3;
    pthread_create(&t3, NULL, thread_func3, NULL);
    sleep(1);
    printf(" threads  created \n");

    printf(" input char to continue . \n");
    char ch;
    scanf("%c", &ch);

    // 唤醒线程
    cond_notify(&cond3);

    // 主线程等待子线程结束
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    // 销毁，锁、条件变量
    pthread_cond_destroy(&cond1);
    pthread_cond_destroy(&cond2);
    pthread_cond_destroy(&cond3);
    pthread_mutex_destroy(&mutex);

    printf(" threads  exit \n");
    return 0;
}
```

编译代码：

```
gcc thread.c -lpthread -std=gnu99 -o thread
```

运行代码：

```
[root@local thread]# ./thread
 process = 111723
 main thread = 111723
 thread = 111724  stack_addr = 0x7f68b0c5eeec
 thread = 111725  stack_addr = 0x7f68b045deec
 thread = 111726  stack_addr = 0x7f68afc5ceec
 threads  created
 input char to continue .
m
 thread = 111724  cpu = 1  seq =  0
 thread = 111725  cpu = 2  seq =  1
 thread = 111726  cpu = 0  seq =  2
 thread = 111724  cpu = 1  seq =  3
 thread = 111725  cpu = 0  seq =  4
 thread = 111726  cpu = 2  seq =  5
 thread = 111724  cpu = 1  seq =  6
```

```
thread = 111725  cpu = 0  seq =  7
thread = 111726  cpu = 2  seq =  8
thread = 111726  set  go_on = false
thread = 111724  exit
thread = 111725  exit
thread = 111726  exit
threads  exit
```

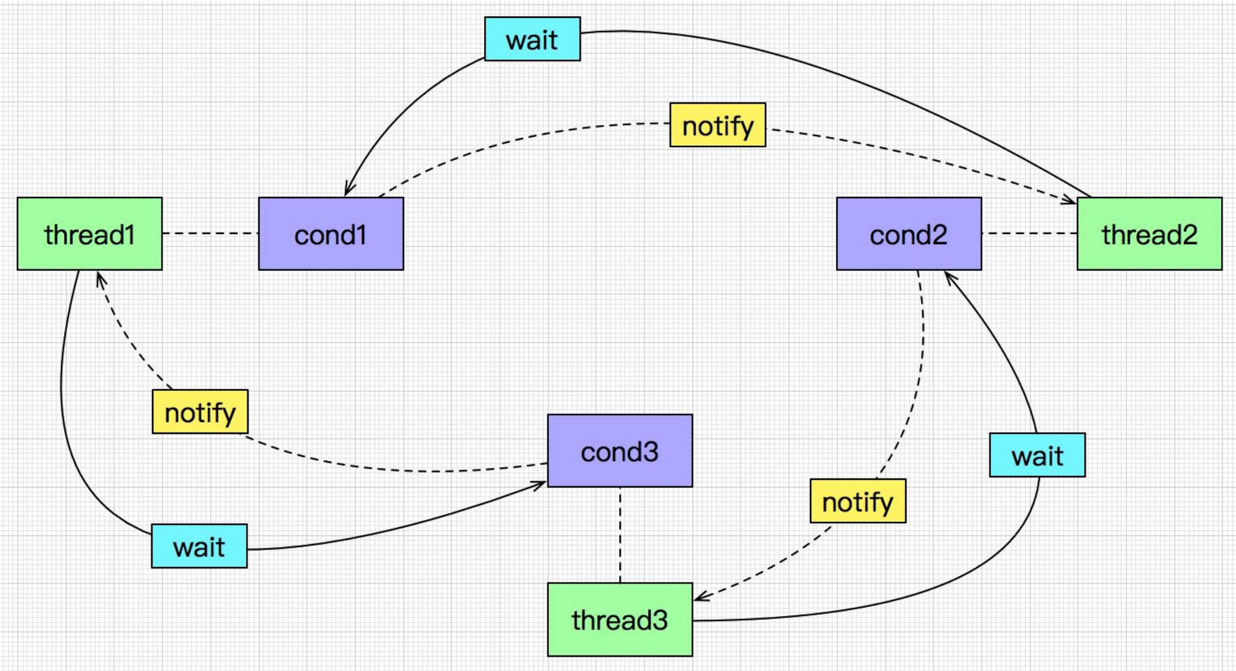
分析结果：
进程 ID 为 111723，主线程 ID 为 111723，两者相同。
创建 3 个线程，线程 ID 分别为 111724、111725、111726，线程栈的地址差值大约为 8MB。

使用输入字符暂停主线程，方便查看线程状态。
使用 top 命令查看进程的线程列表 `top -H -p 111723` ，一共 4 个线程。

```
top - 22:56:16 up 18 days,  5:02,  1 user,  load average: 0.00, 0.03, 0.05
Threads:   4 total,   0 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.8 us,  1.1 sy,  0.0 ni, 98.0 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem :  2851708 total,   740896 free,   480268 used,  1630544 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2053912 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
111723 root      20   0   31104    652    524 S  0.0  0.0   0:00.00 thread
111724 root      20   0   31104    652    524 S  0.0  0.0   0:00.00 thread
111725 root      20   0   31104    652    524 S  0.0  0.0   0:00.00 thread
111726 root      20   0   31104    652    524 S  0.0  0.0   0:00.00 thread
```

线程同步使用锁和条件变量，使用方法 `pthread_cond_wait(cond, &mutex)` 、 `pthread_cond_broadcast(cond)` 。
3 个线程，3 个条件变量，每个线程对应一个条件变量。
3 个线程，组成环状结构，后一个线程依赖前一个线程的条件变量。
前一个线程输出序号之后，使用条件变量唤醒后一个线程，让后一个线程运行起来。
输出序号的方法 print_seq()没有加锁，因为某个时刻只有 1 个线程调用 print_seq()。
输出的序号 seq 依次递增，从 0 到 8。
线程按照 111724、111725、111726 的顺序，多次循环输出序号。



用 C 分析 CPU 密集型和 IO 密集型

编写代码： cpu_busy.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdint.h>
#include <syscall.h>
#include <sched.h>

#define tid syscall(SYS_gettid) // 线程 ID

void *thread_func(void *param)
{
    int taskid = tid;
    printf(" thread = %d  created\n", taskid);
    int64_t num = 0;

    // 循环
    while (1)
    {
        // 复杂的计算
        int64_t tmp = taskid * 55 - 99;
        tmp = (tmp << 3) + 77;
        tmp = tmp / (taskid - 66);
        num += tmp;
    }
```

```
    printf(" thread = %d  output = %lld \n", taskid, num);
    return NULL;
}

int main()
{
    printf(" main thread = %d \n", tid);

    // 创建线程
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, NULL);
    pthread_t t2;
    pthread_create(&t2, NULL, thread_func, NULL);
    sleep(1);
    printf(" threads are created \n");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf(" after threads exit \n");
    return 0;
}
```

编写代码： io_busy.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdint.h>
#include <syscall.h>
#include <sched.h>
#include <fcntl.h>
#include <string.h>

#define tid syscall(SYS_gettid) // 线程 ID

int buf_size = 50;  // 缓存大小
int file_off1 = 0;  // 文件偏移
int file_off2 = 50; // 文件偏移

void *thread_func(void *param)
{
    int taskid = tid;
    char *path = "./io_file.txt";
    int fd = open(path, O_RDWR | O_CREAT, 0666);
    printf(" thread = %d  file = %s  fd = %d \n", taskid, path, fd);

    // 文件偏移
    int *off_ptr = (int *)param;
    int off = *off_ptr;
```

```
    // 循环
    uint64_t num = 0;
    char buf[buf_size];
    while (1)
    {
        lseek(fd, off, SEEK_SET); // 文件偏移
        read(fd, buf, buf_size);  // 读文件

        memset(buf, ' ', buf_size);
        sprintf(buf, "\n thread = %d  loop = %llu \n\n", taskid, num);

        lseek(fd, off, SEEK_SET); // 文件偏移
        write(fd, buf, buf_size); // 写文件
        fsync(fd);                // 刷文件

        lseek(fd, 5000 + off, SEEK_SET); // 文件偏移
        write(fd, buf, buf_size);        // 写文件
        fsync(fd);                       // 刷文件

        ++num;
    }
    return NULL;
}

int main()
{
    printf(" main thread = %d \n", tid);

    // 创建线程
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, &file_off1);
    pthread_t t2;
    pthread_create(&t2, NULL, thread_func, &file_off2);
    sleep(1);
    printf(" threads are created \n");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf(" after threads exit \n");
    return 0;
}
```

编译代码：

```
gcc cpu_busy.c -lpthread -std=gnu99 -o cpu_busy
gcc io_busy.c -lpthread -std=gnu99 -o io_busy
```

运行代码：

```
[root@local thread]# ./cpu_busy
```

```
 main thread = 114703
 thread = 114704  created
 thread = 114705  created
 threads are created

[root@local thread]# ./io_busy
 main thread = 127740
 thread = 127741  file = ./io_file.txt  fd = 3
 thread = 127742  file = ./io_file.txt  fd = 4
 threads are created
```

分析结果：
运行 cpu_busy，查看线程状态，使用命令 `top -H -p 114703` 。

```
top - 00:16:03 up 18 days,  6:22,  1 user,  load average: 1.59, 0.69, 0.28
Threads:   3 total,   2 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 67.8 us,  1.4 sy,  0.0 ni, 30.7 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem :  2851708 total,   698084 free,   503060 used,  1650564 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2031088 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
114704 root      20   0   22900    396    312 R 91.7  0.0   1:57.48 cpu_busy
114705 root      20   0   22900    396    312 R 91.7  0.0   1:57.43 cpu_busy
114703 root      20   0   22900    396    312 S  0.0  0.0   0:00.00 cpu_busy
```

运行 io_busy，查看线程状态，使用命令 `top -H -p 127740` 。备注，本例使用固体硬盘。

```
top - 01:58:52 up 18 days,  8:04,  1 user,  load average: 1.48, 0.77, 0.52
Threads:   3 total,   2 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  9.2 us, 24.3 sy,  0.0 ni, 39.4 id, 23.7 wa,  0.0 hi,  3.4 si,  0.0 st
KiB Mem :  2851708 total,   663856 free,   530264 used,  1657588 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2003884 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
127742 root      20   0   22900    392    312 R 44.3  0.0   0:25.54 io_busy
127741 root      20   0   22900    392    312 R 43.3  0.0   0:25.25 io_busy
127740 root      20   0   22900    392    312 S  0.0  0.0   0:00.00 io_busy
```

cpu_busy 表示 CPU 密集型，CPU 时间占比为 `%Cpu(s): 67.8 us,  1.4 sy,  0.0 ni, 30.7 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st` 。
io_busy 表示 IO 密集型，CPU 时间占比为 `%Cpu(s):  9.2 us, 24.3 sy,  0.0 ni, 39.4 id, 23.7 wa,  0.0 hi,  3.4 si,  0.0 st` 。
重点关注指标 us、sy、wa。
用户态时间占比 us，cpu_busy 高，io_busy 低。
系统态时间占比 sy，cpu_busy 低，io_busy 高。
IO 等待时间占比 wa，cpu_busy 低，io_busy 高。

CPU 密集型程序，CPU 一直运行程序指令，线程阻塞少，上下文切换少，CPU 使用率高。
IO 密集型程序，线程执行 IO 操作、等待 IO 完成，线程阻塞多，CPU 使用率比 CPU 密集型程序低。本例读写固体硬盘，如果读写机械硬盘、读写网络，IO 等待时间占比会更高，效果更明显。
某些程序同时具有 CPU 密集型和 IO 密集型的特征，比如数据库程序，读写磁盘文件属于 IO 密集型，执行复杂的查

询条件属于 CPU 密集型。