

42 | IPC（下）：不同项目组之间抢资源，如何协调？

2019-07-03 刘超

趣谈Linux操作系统

[进入课程 >](#)




讲述：刘超

时长 15:55 大小 14.59M



IPC 这块的内容比较多，为了让你能够更好地理解，我分成了三节来讲。前面我们解析完了共享内存的内核机制后，今天我们来看最后一部分，信号量的内核机制。

首先，我们需要创建一个信号量，调用的是系统调用 `semget`。代码如下：

 复制代码

```
1 SYSCALL_DEFINE3(semget, key_t, key, int, nsems, int, se
2 {
3     struct ipc_namespace *ns;
4     static const struct ipc_ops sem_ops = {
5         .getnew = newary,
6         .associate = sem_security,
7         .more_checks = sem_more_checks,
8     };
9     struct ipc_params sem_params;
10    ns = current->nsproxy->ipc_ns;
11    sem_params.key = key;
12    sem_params.flg = semflg;
13    sem_params.u.nsems = nsems;
14    return ipcget(ns, &sem_ids(ns), &sem_ops, &sem_
15 }
```


我们解析过了共享内存，再看信号量，就顺畅很多了。这里同样调用了抽象的 `ipcget`，参数分别为信号量对应的 `sem_ids`、对应的操作 `sem_ops` 以及对应的参数 `sem_params`。

`ipcget` 的代码我们已经解析过了。如果 `key` 设置为 `IPC_PRIVATE` 则永远创建新的；如果不是的话，就会调用

ipcget_public。

在 ipcget_public 中，我们会按照 key，去查找 struct kern_ipc_perm。如果没有找到，那就看看是否设置了 IPC_CREAT。如果设置了，就创建一个新的。如果找到了，就将对应的 id 返回。

我们这里重点看，如何按照参数 sem_ops，创建新的信号量会调用 newary。

 复制代码

```
1 static int newary(struct ipc_namespace *ns, struct ipc_
2 {
3     int retval;
4     struct sem_array *sma;
5     key_t key = params->key;
6     int nsems = params->u.nsems;
7     int semflg = params->flg;
8     int i;
9     .....
10    sma = sem_alloc(nsems);
11    .....
12    sma->sem_perm.mode = (semflg & S_IRWXUGO);
13    sma->sem_perm.key = key;
14    sma->sem_perm.security = NULL;
15    .....
16    for (i = 0; i < nsems; i++) {
17        INIT_LIST_HEAD(&sma->sems[i].pending_al
18        INIT_LIST_HEAD(&sma->sems[i].pending_cc
19        spin_lock_init(&sma->sems[i].lock);
```


```

20         }
21         sma->complex_count = 0;
22         sma->use_global_lock = USE_GLOBAL_LOCK_HYSTERES
23         INIT_LIST_HEAD(&sma->pending_alter);
24         INIT_LIST_HEAD(&sma->pending_const);
25         INIT_LIST_HEAD(&sma->list_id);
26         sma->sem_nsems = nsems;
27         sma->sem_ctime = get_seconds();
28         retval = ipc_addid(&sem_ids(ns), &sma->sem_perm
29         .....
30         ns->used_sems += nsems;
31         .....
32         return sma->sem_perm.id;
33     }

```

newary 函数的第一步，通过 `kvmalloc` 在直接映射区分配一个 `struct sem_array` 结构。这个结构是用来描述信号量的，这个结构最开始就是上面说的 `struct kern_ipc_perm` 结构。接下来就是填充这个 `struct sem_array` 结构，例如 `key`、权限等。

`struct sem_array` 里有多个信号量，放在 `struct sem` `sems[]` 数组里面，在 `struct sem` 里面有当前的信号量的数值 `semval`。

 复制代码

```
1 struct sem {
```

```

2      int      semval;          /* current value */
3      /*
4      * PID of the process that last modified the se
5      * Linux, specifically these are:
6      * - semop
7      * - semctl, via SETVAL and SETALL.
8      * - at task exit when performing undo adjustr
9      */
10     int      sempid;
11     spinlock_t lock; /* spinlock for fine-gr
12     struct list_head pending_alter; /* pending sing
13     struct list_head pending_const; /* pending sing
14     time_t    sem_otime; /* candidate for sem_ot
15 } ____cacheline_aligned_in_smp;

```

struct sem_array 和 struct sem 各有一个链表 struct list_head pending_alter，分别表示对于整个信号量数组的修改和对于某个信号量的修改。

newary 函数的第二步，就是初始化这些链表。


newary 函数的第三步，通过 ipc_addid 将新创建的 struct sem_array 结构，挂到 sem_ids 里面的基数树上，并返回相应的 id。

信号量创建的过程到此结束，接下来我们来看，如何通过 semctl 对信号量数组进行初始化。

```
1 SYSCALL_DEFINE4(semctl, int, semid, int, semnum, int, c
2 {
3     int version;
4     struct ipc_namespace *ns;
5     void __user *p = (void __user *)arg;
6     ns = current->nsproxy->ipc_ns;
7     switch (cmd) {
8     case IPC_INFO:
9     case SEM_INFO:
10    case IPC_STAT:
11    case SEM_STAT:
12        return semctl_nolock(ns, semid, cmd, ve
13    case GETALL:
14    case GETVAL:
15    case GETPID:
16    case GETNCNT:
17    case GETZCNT:
18    case SETALL:
19        return semctl_main(ns, semid, semnum, c
20    case SETVAL:
21        return semctl_setval(ns, semid, semnum,
22    case IPC_RMID:
23    case IPC_SET:
24        return semctl_down(ns, semid, cmd, vers
25    default:
26        return -EINVAL;
27    }
28 }
```

这里我们重点看，SETALL 操作调用的 semctl_main 函数，以及 SETVAL 操作调用的 semctl_setval 函数。

对于 SETALL 操作来讲，传进来的参数为 union semun 里面的 unsigned short *array，会设置整个信号量集合。

 复制代码

```
1 static int semctl_main(struct ipc_namespace *ns, int se
2             int cmd, void __user *p)
3 {
4     struct sem_array *sma;
5     struct sem *curr;
6     int err, nsems;
7     ushort fast_sem_io[SEMMSL_FAST];
8     ushort *sem_io = fast_sem_io;
9     DEFINE_WAKE_Q(wake_q);
10    sma = sem_obtain_object_check(ns, semid);
11    nsems = sma->sem_nsems;
12    .....
13    switch (cmd) {
14    .....
15    case SETALL:
16    {
17        int i;
18        struct sem_undo *un;
19        .....
20        if (copy_from_user(sem_io, p, nsems*siz
21        .....
22        }
23        .....
24        for (i = 0; i < nsems; i++) {
25            sma->sems[i].semval = sem_io[i]
```

```

26             sma->sems[i].sempid = task_tgic
27         }
28     .....
29         sma->sem_ctime = get_seconds();
30         /* maybe some queued-up processes were
31         do_smart_update(sma, NULL, 0, 0, &wake_
32         err = 0;
33         goto out_unlock;
34     }
35 }
36 .....
37     wake_up_q(&wake_q);
38 .....
39 }

```

在 `semctl_main` 函数中，先是通过 `sem_obtain_object_check`，根据信号量集合的 `id` 在基数树里面找到 `struct sem_array` 对象，发现如果是 `SETALL` 操作，就将用户的参数中的 `unsigned short *array` 通过 `copy_from_user` 拷贝到内核里面的 `sem_io` 数组，然后是一个循环，对于信号量集合里面的每一个信号量，设置 `semval`，以及修改这个信号量值的 `pid`。

对于 `SETVAL` 操作来讲，传进来的参数 `union semun` 里面的 `int val`，仅仅会设置某个信号量。


```


1 static int semctl_setval(struct ipc_namespace *ns, int
2                          unsigned long arg)
3 {
4     struct sem_undo *un;
5     struct sem_array *sma;
6     struct sem *curr;
7     int err, val;
8     DEFINE_WAKE_Q(wake_q);
9     .....
10    sma = sem_obtain_object_check(ns, semid);
11    .....
12    curr = &sma->sems[semnum];
13    .....
14    curr->semval = val;
15    curr->sempid = task_tgid_vnr(current);
16    sma->sem_ctime = get_seconds();
17    /* maybe some queued-up processes were waiting
18    do_smart_update(sma, NULL, 0, 0, &wake_q);
19    .....
20    wake_up_q(&wake_q);
21    return 0;
22 }

```



在 `semctl_setval` 函数中，我们先是通过 `sem_obtain_object_check`，根据信号量集合的 id 在基数树里面找到 `struct sem_array` 对象，对于 SETVAL 操作，直接根据参数中的 `val` 设置 `semval`，以及修改这个信号量值的 `pid`。

至此，信号量数组初始化完毕。接下来我们来看 P 操作和 V 操作。无论是 P 操作，还是 V 操作都是调用 semop 系统调用。

 复制代码

```
1 SYSCALL_DEFINE3(semop, int, semid, struct sembuf __user
2             unsigned, nsops)
3 {
4     return sys_semtimedop(semid, tsops, nsops, NULL
5 }
6
7 SYSCALL_DEFINE4(semtimedop, int, semid, struct sembuf _
8             unsigned, nsops, const struct timespec
9 {
10     int error = -EINVAL;
11     struct sem_array *sma;
12     struct sembuf fast_sops[SEMOPM_FAST];
13     struct sembuf *sops = fast_sops, *sop;
14     struct sem_undo *un;
15     int max, locknum;
16     bool undos = false, alter = false, dupsop = fal
17     struct sem_queue queue;
18     unsigned long dup = 0, jiffies_left = 0;
19     struct ipc_namespace *ns;
20
21     ns = current->nsproxy->ipc_ns;
22     .....
23     if (copy_from_user(sops, tsops, nsops * sizeof(
24             error = -EFAULT;
25             goto out_free;
26     }
27
28     if (timeout) {
```

```

29         struct timespec _timeout;
30         if (copy_from_user(&_timeout, timeout,
31                             })
32             jiffies_left = timespec_to_jiffies(&_ti
33     }
34     .....
35     /* On success, find_alloc_undo takes the rcu_re
36     un = find_alloc_undo(ns, semid);
37     .....
38     sma = sem_obtain_object_check(ns, semid);
39     .....
40     queue.sops = sops;
41     queue.nsops = nsops;
42     queue.undo = un;
43     queue.pid = task_tgid_vnr(current);
44     queue.alter = alter;
45     queue.dupsop = dupsop;
46
47     error = perform_atomic_semop(sma, &queue);
48     if (error == 0) { /* non-blocking successful pa
49         DEFINE_WAKE_Q(wake_q);
50     .....
51         do_smart_update(sma, sops, nsops, 1, &w
52     .....
53         wake_up_q(&wake_q);
54         goto out_free;
55     }
56     /*
57     * We need to sleep on this operation, so we pu
58     * task into the pending queue and go to sleep.
59     */
60     if (nsops == 1) {
61         struct sem *curr;
62         curr = &sma->sems[sops->sem_num];
63     .....

```

```

64             list_add_tail(&queue.list,
65                             &curr->
66 .....
67         } else {
68 .....
69             list_add_tail(&queue.list, &sma->pendir
70 .....
71         }
72
73     do {
74         queue.status = -EINTR;
75         queue.sleeper = current;
76
77         __set_current_state(TASK_INTERRUPTIBLE)
78         if (timeout)
79             jiffies_left = schedule_timeout
80         else
81             schedule();
82 .....
83         /*
84          * If an interrupt occurred we have to
85          */
86         if (timeout && jiffies_left == 0)
87             error = -EAGAIN;
88     } while (error == -EINTR && !signal_pending(cur
89 .....
90 }


```



semop 会调用 semtimedop, 这是一个非常复杂的函数。

`semtimedop` 做的第一件事情，就是将用户的参数，例如，对于信号量的操作 `struct sembuf`，拷贝到内核里面来。另外，如果是 P 操作，很可能让进程进入等待状态，是否要为此等待状态设置一个超时，`timeout` 也是一个参数，会把它变成时钟的滴答数目。

`semtimedop` 做的第二件事情，是通过 `sem_obtain_object_check`，根据信号量集合的 `id`，获得 `struct sem_array`，然后，创建一个 `struct sem_queue` 表示当前的信号量操作。为什么叫 `queue` 呢？因为这个操作可能马上就能完成，也可能因为无法获取信号量不能完成，不能完成的话就只好排列到队列上，等待信号量满足条件的时候。`semtimedop` 会调用 `perform_atomic_semop` 在实施信号量操作。

 复制代码

```
1 static int perform_atomic_semop(struct sem_array *sma,
2 {
3     int result, sem_op, nsops;
4     struct sembuf *sop;
5     struct sem *curr;
6     struct sembuf *sops;
7     struct sem_undo *un;
8
9     sops = q->sops;
10    nsops = q->nsops;
11    un = q->undo;
12
```

```

13         for (sop = sops; sop < sops + nsops; sop++) {
14             curr = &sma->sems[sop->sem_num];
15             sem_op = sop->sem_op;
16             result = curr->semval;
17         .....
18             result += sem_op;
19             if (result < 0)
20                 goto would_block;
21         .....
22             if (sop->sem_flg & SEM_UNDO) {
23                 int undo = un->semadj[sop->sem_
24         .....
25                 }
26     }
27
28     for (sop = sops; sop < sops + nsops; sop++) {
29         curr = &sma->sems[sop->sem_num];
30         sem_op = sop->sem_op;
31         result = curr->semval;
32
33         if (sop->sem_flg & SEM_UNDO) {
34             int undo = un->semadj[sop->sem_
35             un->semadj[sop->sem_num] = undc
36         }
37         curr->semval += sem_op;
38         curr->sempid = q->pid;
39     }
40     return 0;
41 would_block:
42     q->blocking = sop;
43     return sop->sem_flg & IPC_NOWAIT ? -EAGAIN : 1;
44 }

```

在 `perform_atomic_semop` 函数中，对于所有信号量操作都进行两次循环。在第一次循环中，如果发现计算出的 `result` 小于 0，则说明必须等待，于是跳到 `would_block` 中，设置 `q->blocking = sop` 表示这个 queue 是 block 在这个操作上，然后如果需要等待，则返回 1。如果第一次循环中发现无需等待，则第二个循环实施所有的信号量操作，将信号量的值设置为新的值，并且返回 0。

接下来，我们回到 `semtimedop`，来看它干的第三件事情，就是如果需要等待，应该怎么办？

如果需要等待，则要区分刚才的对于信号量的操作，是对一个信号量的，还是对于整个信号量集合的。如果是对于一个信号量的，那我们就将 queue 挂到这个信号量的 `pending_alter` 中；如果是对于整个信号量集合的，那我们就将 queue 挂到整个信号量集合的 `pending_alter` 中。


接下来的 `do-while` 循环，就是要开始等待了。如果等待没有时间限制，则调用 `schedule` 让出 CPU；如果等待有时间限制，则调用 `schedule_timeout` 让出 CPU，过一段时间还回来。当回来的时候，判断是否等待超时，如果没有等待超时则进入下一轮循环，再次等待，如果超时则退出循环，返回错误。在让出 CPU 的时候，设置进程的状态为 `TASK_INTERRUPTIBLE`，并且循环的结束会通过

signal_pending 查看是否收到过信号，这说明这个等待信号量的进程是可以被信号中断的，也即一个等待信号量的进程是可以通过 kill 杀掉的。

我们再来看，semtimedop 要做的第四件事情，如果不需要等待，应该怎么办？

如果不需要等待，就说明对于信号量的操作完成了，也改变了信号量的值。接下来，就是一个标准流程。我们通过 DEFINE_WAKE_Q(wake_q) 声明一个 wake_q，调用 do_smart_update，看这次对于信号量的值的改变，可以影响并可以激活等待队列中的哪些 struct sem_queue，然后把它们都放在 wake_q 里面，调用 wake_up_q 唤醒这些进程。其实，所有的对于信号量的值的修改都会涉及这三个操作，如果你回过头去仔细看 SETALL 和 SETVAL 操作，在设置完毕信号量之后，也是这三个操作。

我们来看 do_smart_update 是如何实现的。
do_smart_update 会调用 update_queue。

 复制代码

```
1 static int update_queue(struct sem_array *sma, int semr
2 {
3     struct sem_queue *q, *tmp;
4     struct list_head *pending_list;
```




```

5         int semop_completed = 0;
6
7         if (semnum == -1)
8             pending_list = &sma->pending_alter;
9         else
10            pending_list = &sma->sems[semnum].pendi
11
12 again:
13     list_for_each_entry_safe(q, tmp, pending_list,
14                             int error, restart;
15     .....
16
17         error = perform_atomic_semop(sma, q);
18
19         /* Does q->sleeper still need to sleep?
20         if (error > 0)
21             continue;
22
23         unlink_queue(sma, q);
24     .....
25         wake_up_sem_queue_prepare(q, error, wak
26     .....
27     }
28     return semop_completed;
29 }
30 static inline void wake_up_sem_queue_prepare(struct sem
31                                             struct wak
32 {
33     wake_q_add(wake_q, q->sleeper);
34     .....
35 }

```

update_queue 会依次循环整个信号量集合的等待队列 pending_alter，或者某个信号量的等待队列。试图在信号量的值变了的情况下，再次尝试 perform_atomic_semop 进行信号量操作。如果不成功，则尝试队列中的下一个；如果尝试成功，则调用 unlink_queue 从队列上取下来，然后调用 wake_up_sem_queue_prepare，将 q->sleeper 加到 wake_q 上去。q->sleeper 是一个 task_struct，是等待在这个信号量操作上的进程。

接下来，wake_up_q 就依次唤醒 wake_q 上的所有 task_struct，调用的是我们在进程调度那一节学过的 wake_up_process 方法。

 复制代码

```
1 void wake_up_q(struct wake_q_head *head)
2 {
3     struct wake_q_node *node = head->first;
4
5     while (node != WAKE_Q_TAIL) {
6         struct task_struct *task;
7
8         task = container_of(node, struct task_s
9
10         node = node->next;
11         task->wake_q.next = NULL;
12
13         wake_up_process(task);
14         put_task_struct(task);
15     }
```

```
16 }
```

```
17
```




至此，对于信号量的主流操作都解析完毕了。

其实还有一点需要强调一下，信号量是一个整个 Linux 可见的全局资源，而不像咱们在线程同步那一节讲过的都是某个进程独占的资源，好处是可以跨进程通信，坏处就是如果一个进程通过 P 操作拿到了一个信号量，但是不幸异常退出了，如果没有来得及归还这个信号量，可能所有其他的进程都阻塞了。


那怎么办呢？Linux 有一种机制叫 SEM_UNDO，也即每一个 semop 操作都会保存一个反向 struct sem_undo 操作，当因为某个进程异常退出的时候，这个进程做的所有的操作都会回退，从而保证其他进程可以正常工作。

如果你回头看，我们写的程序里面的 semaphore_p 函数和 semaphore_v 函数，都把 sem_flg 设置为 SEM_UNDO，就是这个作用。


等待队列上的每一个 struct sem_queue，都有一个 struct sem_undo，以此来表示这次操作的反向操作。

 复制代码

```
1 struct sem_queue {
2     struct list_head    list;    /* queue of pe
3     struct task_struct  *sleeper; /* this proce
4     struct sem_undo     *undo;    /* undo struct
5     int                 pid;      /* process id
6     int                 status;   /* completion
7     struct sembuf       *sops;   /* array of pe
8     struct sembuf       *blocking; /* the opera
9     int                 nsops;    /* number of c
10    bool                 alter;    /* does *sops
11    bool                 dupsop;   /* sops on mor
12 };
```



在进程的 `task_struct` 里面对于信号量有一个成员 `struct sysv_sem`，里面是一个 `struct sem_undo_list`，将这个进程所有的 `semop` 所带来的 `undo` 操作都串起来。

 复制代码

```
1 struct task_struct {
2     .....
3     struct sysv_sem          sysvsem;
4     .....
5 }
6
7 struct sysv_sem {
8     struct sem_undo_list *undo_list;
9 };
10
```

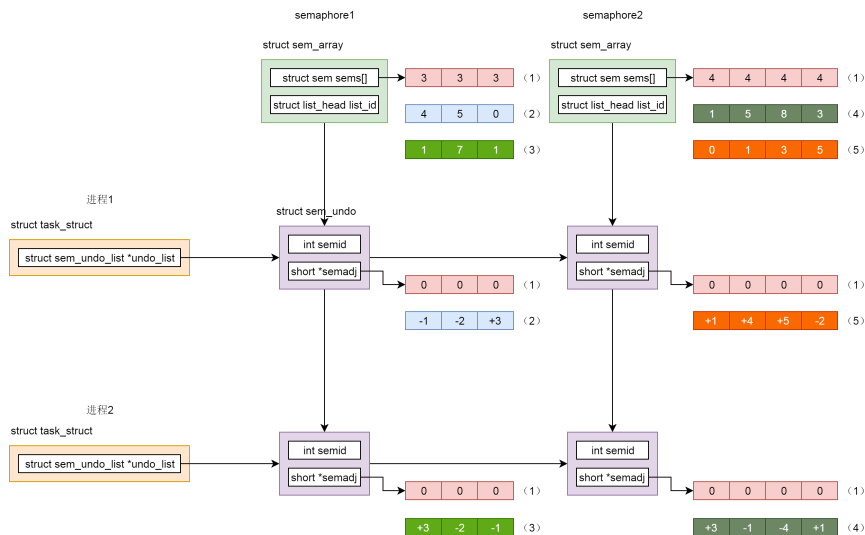
```

11 struct sem_undo {
12     struct list_head    list_proc;    /* per-
13                                         * all
14                                         * rcu
15     struct rcu_head      rcu;         /* rcu
16     struct sem_undo_list *ulp;        /* back
17     struct list_head     list_id;     /* per
18                                         * all
19     int                   semid;       /* sema
20     short                 *semadj;     /* arra
21                                         /* one
22 };
23
24 struct sem_undo_list {
25     atomic_t              refcnt;
26     spinlock_t            lock;
27     struct list_head      list_proc;
28 };

```

为了让你更清楚地理解 struct sem_undo 的原理，我们这里举一个例子。

假设我们创建了两个信号量集合。一个叫 semaphore1，它包含三个信号量，初始化值为 3，另一个叫 semaphore2，它包含 4 个信号量，初始化值都为 4。初始化时候的信号量以及 undo 结构里面的值如图中 (1) 标号所示。



首先，我们来看进程 1。我们调用 `semop`，将 semaphore1 的三个信号量的值，分别加 1、加 2 和减 3，从而信号量的值变为 4,5,0。于是在 semaphore1 和进程 1 链表交汇的 undo 结构里面，填写 -1,-2,+3，是 `semop` 操作的反向操作，如图中 (2) 标号所示。

然后，我们来看进程 2。我们调用 `semop`，将 semaphore1 的三个信号量的值，分别减 3、加 2 和加 1，从而信号量的值变为 1、7、1。于是在 semaphore1 和进程 2 链表交汇的 undo 结构里面，填写 +3、-2、-1，是 `semop` 操作的反向操作，如图中 (3) 标号所示。

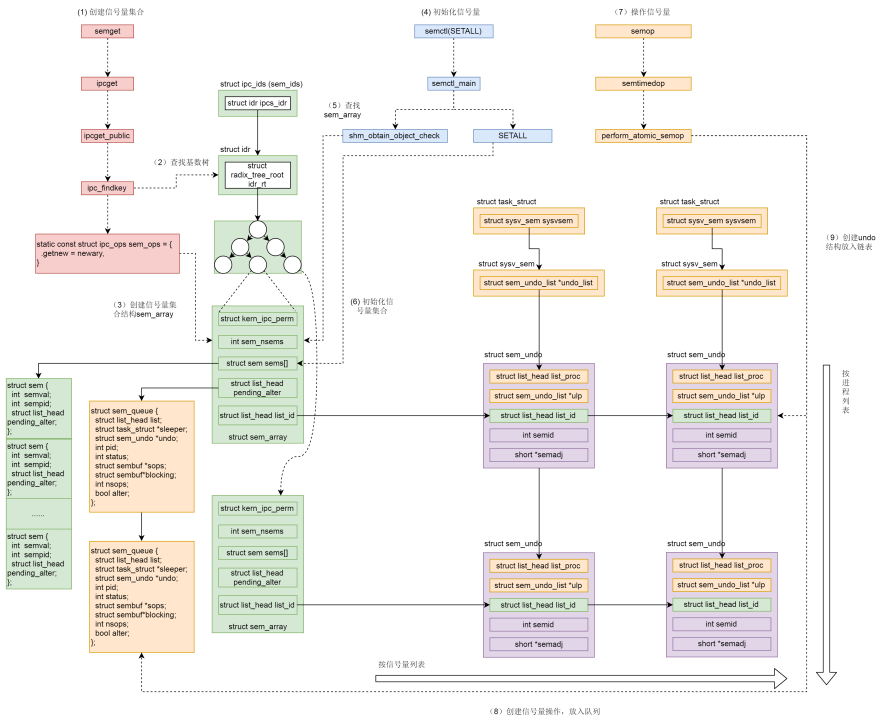
然后，我们接着看进程 2。我们调用 semop，将 semaphore2 的四个信号量的值，分别减 3、加 1、加 4 和减 1，从而信号量的值变为 1、5、8、3。于是，在 semaphore2 和进程 2 链表交汇的 undo 结构里面，填写 +3、-1、-4、+1，是 semop 操作的反向操作，如图中 (4) 标号所示。

然后，我们再来看进程 1。我们调用 semop，将 semaphore2 的四个信号量的值，分别减 1、减 4、减 5 和加 2，从而信号量的值变为 0、1、3、5。于是在 semaphore2 和进程 1 链表交汇的 undo 结构里面，填写 +1、+4、+5、-2，是 semop 操作的反向操作，如图中 (5) 标号所示。

从这个例子可以看出，无论哪个进程异常退出，只要将 undo 结构里面的值加回当前信号量的值，就能够得到正确的信号量的值，不会因为一个进程退出，导致信号量的值处于不一致的状态。

总结时刻

信号量的机制也很复杂，我们对下面这个图总结一下。



1. 调用 `semget` 创建信号量集合。
2. `ipc_findkey` 会在基数树中，根据 `key` 查找信号量集合 `sem_array` 对象。如果已经被创建，就会被查询出来。例如 `producer` 被创建过，在 `consumer` 中就会查询出来。
3. 如果信号量集合没有被创建过，则调用 `sem_ops` 的 `newary` 方法，创建一个信号量集合对象 `sem_array`。例如，在 `producer` 中就会新建。
4. 调用 `semctl(SETALL)` 初始化信号量。
5. `shm_obtain_object_check` 先从基数树里面找到 `sem_array` 对象。

6. 根据用户指定的信号量数组，初始化信号量集合，也即初始化 `sem_array` 对象的 `struct sem sems[]` 成员。
7. 调用 `semop` 操作信号量。
8. 创建信号量操作结构 `sem_queue`，放入队列。
9. 创建 `undo` 结构，放入链表。

课堂练习

现在，我们的共享内存、信号量、消息队列都讲完了，你是不是觉得，它们的 API 非常相似。为了方便记忆，你可以自己整理一个表格，列一下这三种进程间通信机制、行为创建 `xxxget`、使用、控制 `xxxctl`、对应的 API 和系统调用。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 41 | IPC（中）：不同项目组之间抢资源，如何协调？

下一篇 43 预习 | Socket通信之网络协议基本原理

精选留言 (5)

写留言



王之刚

2019-07-06

请问一下老师，在应用程序开发中，像信号量 共享内存 这些内核资源怎么样防止泄漏呢？比如有进程a和b用共享

内存共享数据，共享内存资源由教程a申请和维护，但由于异常情况导致教程异常退出导致共享内存资源没有释放，导致了申请的共享内存没有释放。这种情况一般怎...
展开 ∨



安排

2019-07-04

schedule_timeout调用完后，会让出cpu，过一段时间还会回来。这个过一段时间是多长时间啊？是说超时之后返回来吗，还是被其它信号打断睡眠之后回来？

展开 ∨



莫名

2019-07-04

老师，有没有打算讲一下POSIX IPC呢？

展开 ∨



免费的人

2019-07-03

消息队列的内核实现好像没讲过？

展开 ∨



Sharry

2019-07-03

终于把共享内存和信号量集合的知识串联在一起了, 其中的操作的确有些复杂

共享内存若想实现进程之间的同步读写, 则需要配合信号量共同使用...

展开 ∨

