

System V IPC

NCTU CIS

OS Lab

Linux Kernel Trace Seminar

2002.5.20

konami@os.nctu.edu.tw

Outline

- I Introduction
- I Message Queues
- I Semaphore
- I Shared Memory

Introduction

- | All the System V IPC services can be used only for communication between processes on the same machine
- | System V IPC features enable communication among more than two processes.
- | System V IPC enables communication among processes that have no common heritage.
- | In the linux 2.4 kernel, the System V IPC subsystem was mostly rewritten.

Message Queues

- | For processes to send messages asynchronously to each other.
- | New messages are always added to the end of a queue, but can be removed anywhere in the queue



msg_msg structure

Describes a messages and holds the contents

```
struct msg_msg {  
    struct list_head m_list;  
    long m_type;  
    int m_ts;          /* message text size */  
    struct msg_msgseg *next;  
    /* the actual message follows immediately */  
};
```

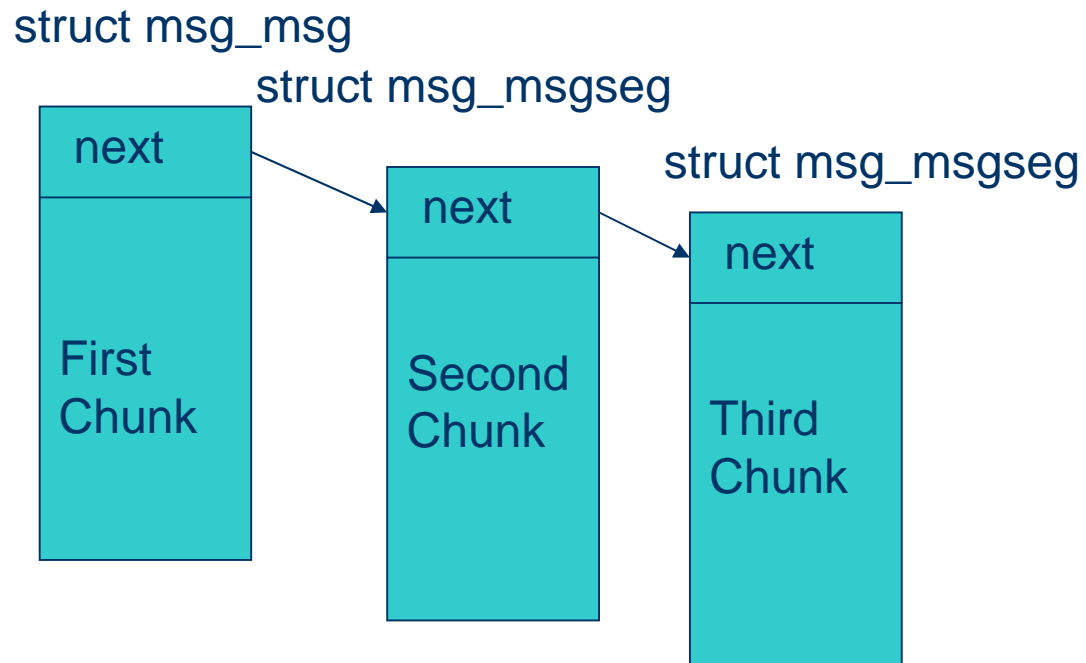
Maintains a list of the linked messages in a single queue

msg_msgsegs structure

- I If a message is too large, it will be broken into more chunks linked within msg_msgseg

```
struct msg_msgseg {  
    struct msg_msgseg *next;  
    /* the next part of the message follows immediately */  
};
```

Structure of large messages



msg_queue structure

I Represents a single message queue

```
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time_t q_stime;          /* last msgsnd time */
    time_t q_rtime;         /* last msgrcv time */
    time_t q_ctime;         /* last change time */
    unsigned long q_cbytes; /* current number of * bytes on queue */
    unsigned long q_qnum;   /* number of messages in* queue */
    unsigned long q_qbytes; /* max number of bytes* on queue */
    pid_t q_lspid;          /* pid of last msgsnd */
    pid_t q_lrpid;         /* last receive pid */
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

Default:
MSGMNB = 16384

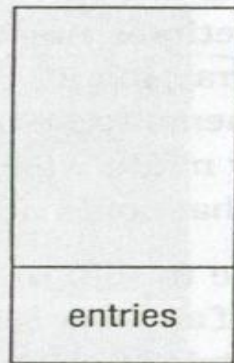
ipc_ids structure

I Where the kernel keeps its message queue

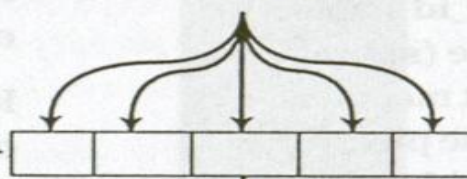
```
struct ipc_ids {  
    int size;  
    int in_use;  
    int max_id;  
    unsigned short seq;  
    unsigned short seq_max;  
    struct semaphore sem;  
    spinlock_t ary;  
    struct ipc_id *entries;  
};
```

Message queue data structures

struct ipc_ids



struct ipc_id



q_perm

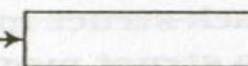


struct msg_queue

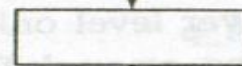
struct msg_msg



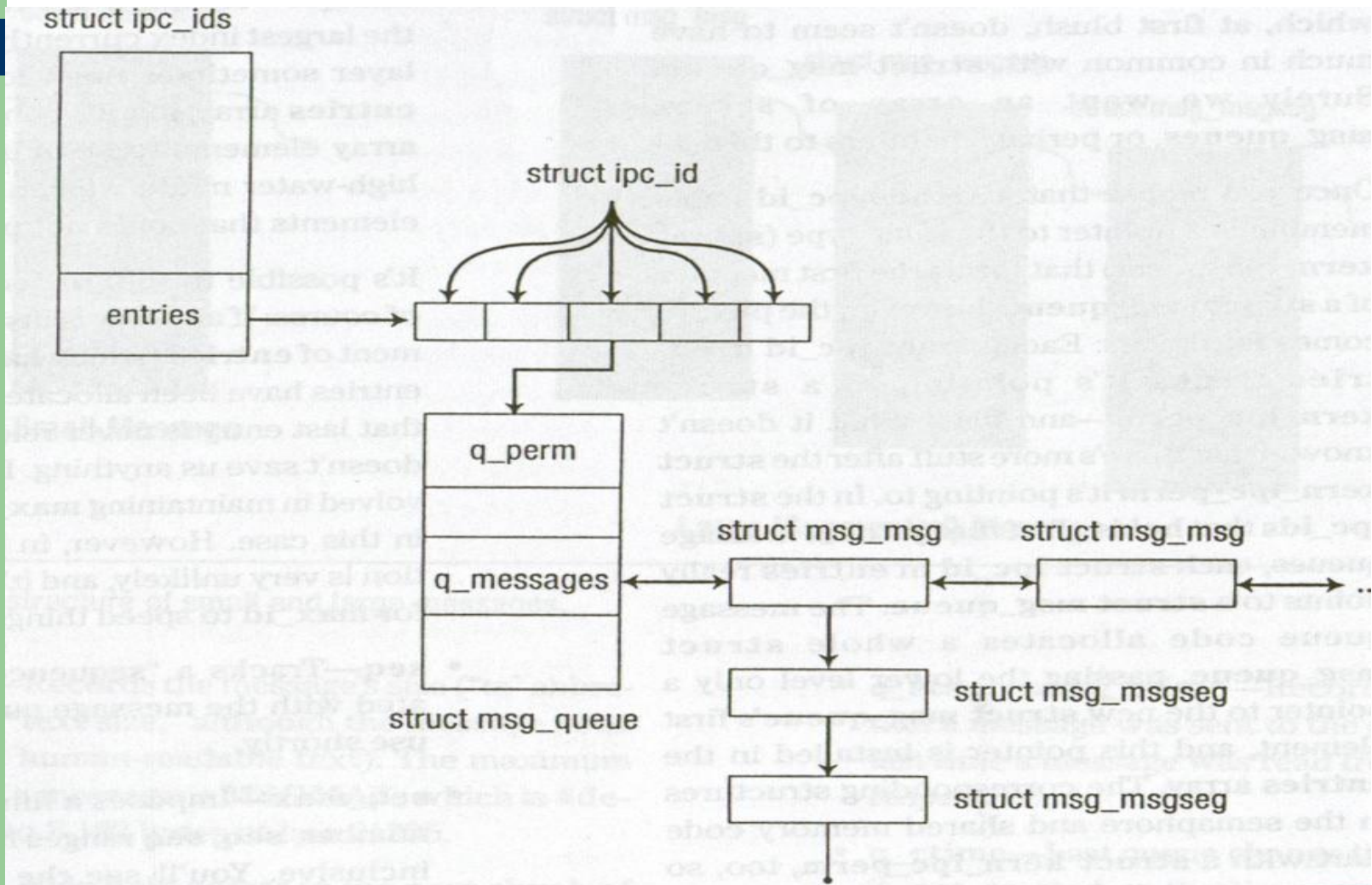
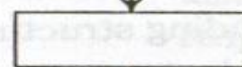
struct msg_msg



struct msg_msgseg



struct msg_msgseg



Message queue related system call

- I msgget
 - Return an id either for the existing queue with the key, or for a new message queue with the key
- I msgsnd
 - Sends a message to a message queue
- I msgrcv
 - Receives a message from a message queue
- I msgctl
 - Perform a set of administrative operations on a message queue

msg_init

n Invoked by ipc_init

```
void __init msg_init(void)
{
    ipc_init_ids(&msg_ids, msg_ctlmni);
#ifdef CONFIG_PROC_FS
    create_proc_read_entry("sysvipc/msg", 0, 0, sysvipc_msg_read_proc, NULL);
#endif
}
```

ipc_init_ids

```
void __init ipc_init_ids(struct ipc_ids *ids, int size)
{
    int i;
    sema_init(&ids->sem, 1);
    if (size > IPCMNI)
        size = IPCMNI;
    ids->size = size;
    ids->in_use = 0;
    ids->max_id = -1;
    ids->seq = 0;
    {
        int seq_limit = INT_MAX / SEQ_MULTIPLIER;
        if (seq_limit > USHRT_MAX)
            ids->seq_max = USHRT_MAX;
        else
            ids->seq_max = seq_limit;
    }
}
.....
```

ipc_init_ids(cont.)

```
....
ids->entries = ipc_alloc(sizeof(struct ipc_id) * size);
if (ids->entries == NULL) {
    printk(KERN_ERR "ipc_init_ids() failed,
    "ipc service disabled.\n");
    ids->size = 0;
}
ids->ary = SPIN_LOCK_UNLOCKED;
for (i = 0; i < size; i++)
ids->entries[i].p = NULL;
}
```



```
void *ipc_alloc(int size)
{
    void *out;
    if (size > PAGE_SIZE)
        out = vmalloc(size);
    else
        out = kmalloc(size, GFP_KERNEL);
    return out;
}
```

sys_msgget

```
|  
{  
    int id, ret = -EPERM;  
    struct msg_queue *msq;  
    down(&msg_ids.sem);  
    if (key == IPC_PRIVATE)  
        ret = newque(key, msgflg);  
    else if ((id = ipc_findkey(&msg_ids, key)) == -1) { /* key not used */  
        if (!(msgflg & IPC_CREAT))  
            ret = -ENOENT;  
        else  
            ret = newque(key, msgflg);  
    } else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {  
        ret = -EEXIST;  
    }  
}
```

#define IPC_PRIVATE ((__kernel_key_t) 0)

.....

sys_msgget(cont.)

```
.....
}else{
    msq = msg_lock(id);
    if (msq == NULL)
        BUG();
    if (ipcperms(&msq->q_perm, msgflg))
        ret = -EACCES;
    else
        ret = msg_buildid(id, msq->q_perm.seq);
    msg_unlock(id);
}
up(&msg_ids.sem);
return ret;
}
```

ipc_findkey

I *Locates the message queue with the given key*


```
int ipc_findkey(struct ipc_ids *ids, key_t key)
{
    int id;
    struct kern_ipc_perm *p;
    for (id = 0; id <= ids->max_id; id++) {
        p = ids->entries[id].p;
        if (p == NULL)
            continue;
        if (key == p->key)
            return id;
    }
    return -1;
}
```

sys_msgsnd

```
asmlinkage long sys_msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
    struct msg_queue *msq;
    struct msg_msg *msg;
    long mtype;
    int err;

    .....

    msg = load_msg(msgp->mtext, msgsz);
    if (IS_ERR(msg))
        return PTR_ERR(msg);
    msg->m_type = mtype;
    msg->m_ts = msgsz;
    .....
}
```



Read the message
content, Discuss Later!!

sys_msgsnd(cont.)

```
.....
    msq = msg_lock(msqid);
    err = -EINVAL;
    if (msq == NULL)
        goto out_free;
retry:
    err = -EIDRM;
    if (msg_checkid(msq, msqid))
        goto out_unlock_free;
    err = -EACCES;
    if (ipcperms(&msq->q_perm, S_IWUGO))
        goto out_unlock_free;
.....
```

sys_msgsnd(cont.)

Check whether the maximum allowed size is exceeded

```
.....  
if (msgsz + msq->q_cbytes > msq->q_qbytes || 1 + msq->q_qnum > msq->q_qbytes) {  
    if (msgflg & IPC_NOWAIT) {  
        err = -EAGAIN;  
        goto out_unlock_free;  
    }  
    ss_add(msq, &s);  
    msg_unlock(msqid);  
    schedule();  
    current->state = TASK_RUNNING;  
    msq = msg_lock(msqid);  
    err = -EIDRM;  
    if (msq == NULL)  
        goto out_free;  
    ss_del(&s);  
.....
```

Add the calling process to the queue of sleeping processes

sys_msgsnd(cont.)

```
.....
if (signal_pending(current)) {
    err = -EINTR;
    goto out_unlock_free;
}
goto retry;
}
if (!pipelined_send(msq, msg)) {
/* noone is waiting for this message, enqueue it */
    list_add_tail(&msg->m_list, &msq->q_messages);
    msq->q_cbytes += msgsz;
    msq->q_qnum++;
    atomic_add(msgsz, &msg_bytes);
    atomic_inc(&msg_hdrs);
}
.....
out_unlock_free:
    msg_unlock(msqid);
out_free:
    if (msg != NULL)
        ree_msg(msg);
    return err;
}
```

load_msg

```
static struct msg_msg *load_msg(void *src, int len)
{
.....
    alen = len;
    if (alen > DATALEN_MSG)
        alen = DATALEN_MSG;
    msg = (struct msg_msg *) kmalloc(sizeof(*msg) + alen, GFP_KERNEL);
....
    msg->next = NULL;
    if (copy_from_user(msg + 1, src, alen)) {
        err = -EFAULT;
        goto out_err;
    }
.....
```

Message Data length

Control and message content will be allocated continuously

Copy message content from user space right after the msg structure

load_msg(cont.)

```
.....  
len -= alen;  
src = ((char *) src) + alen;  
pseg = &msg->next;  
while (len > 0) {  
    struct msg_msgseg *seg;  
    alen = len;  
    if (alen > DATALEN_SEG)  
        alen = DATALEN_SEG;  
    seg = (struct msg_msgseg *)kmalloc(sizeof(*seg) + alen, GFP_KERNEL);  
    .....  
    if (copy_from_user(seg + 1, src, alen)) {  
        err = -EFAULT;  
        goto out_err;  
    }  
    return msg;  
    .....  
}
```

Need more chunk!!

Allocate msg_msgseg for the rest message content

sys_msgrcv

```
asmlinkage long sys_msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp,  
int msgflg)
```

```
{  
    struct msg_queue *msq;  
    struct msg_receiver msr_d;  
    struct list_head *tmp;  
    struct msg_msg *msg, *found_msg;  
    int err;  
    int mode;  
    if (msqid < 0 || (long) msgsz < 0)  
        return -EINVAL;  
    mode = convert_mode(&msgtyp, msgflg);  
    msq = msg_lock(msqid);  
    if (msq == NULL)  
        return -EINVAL;
```

.....

```
static int testmsg(struct msg_msg *msg, long type,  
int mode){  
    switch (mode) {  
    case SEARCH_ANY:  
        return 1;  
    case SEARCH_LESSEQUAL:  
        if (msg->m_type <= type)  
            return 1;  
        break;  
    case SEARCH_EQUAL:  
        if (msg->m_type == type)  
            return 1;  
        break;  
    case SEARCH_NOTEQUAL:  
        if (msg->m_type != type)  
            return 1;  
        break;  
    }  
    return 0;  
}
```

sys_msgrcv(cont.)

```
.....
retry:
err = -EACCES;
if (ipcperms(&msq->q_perm, S_IRUGO))
    goto out_unlock;
tmp = msq->q_messages.next;
found_msg = NULL;
while (tmp != &msq->q_messages) {
    msg = list_entry(tmp, struct msg_msg, m_list);
    if (testmsg(msg, msgtyp, mode)) {
        found_msg = msg;
        if (mode == SEARCH_LESSEQUAL && msg->m_type != 1) {
            found_msg = msg;
            msgtyp = msg->m_type - 1;
        } else {
            found_msg = msg;
            break;
        }
    }
    tmp = tmp->next;
}
.....
```

For next time
finding a **smaller**
one (if exist).

sys_msgrcv(cont.)

```
.....  
if (found_msg) {  
    msg = found_msg;  
    if ((msgsz < msg->m_ts) && !(msgflg & MSG_NOERROR)) {  
        err = -E2BIG;  
        goto out_unlock;  
    }  
    list_del(&msg->m_list);  
    msq->q_qnum--;  
    msq->q_rtime = CURRENT_TIME;  
    msq->q_lrp_id = current->pid;  
    msq->q_cbytes -= msg->m_ts;  
    atomic_sub(msg->m_ts, &msg_bytes);  
    atomic_dec(&msg_hdrs);  
    ss_wakeup(&msq->q_senders, 0);  
    msg_unlock(msqid);  
    .....
```

Receiver buffer too small

Receiving operation starts!

Wake up senders

!!

sys_msgrcv(cont.)

```
if (found_msg) {
```

```
.....
```

```
out_success:
```

```
    msgsz = (msgsz > msg->m_ts) ? msg->m_ts : msgsz;
```

```
    if (put_user(msg->m_type, &msgp->mtype) ||  
        store_msg(msgp->mtext, msg, msgsz)) {
```

```
        msgsz = -EFAULT;  
    }
```

```
free_msg(msg);
```

```
return msgsz;
```

```
} else {
```

```
.....
```

Number of bytes to deliver

!!!

More chance?? NO!

sys_msgrcv(cont.)

```
if (found_msg) {
    .....
} else {
    struct msg_queue *t;
    if (msgflg & IPC_NOWAIT) {
        err = -ENOMSG;
        goto out_unlock;
    }
    list_add_tail(&msr_d.r_list, &msq->q_receivers);
    msr_d.r_tsk = current;
    msr_d.r_msgtype = msgtyp;
    msr_d.r_mode = mode;
    if (msgflg & MSG_NOERROR)
        msr_d.r_maxsize = INT_MAX;
    else
        msr_d.r_maxsize = msgsz;
    msr_d.r_msg = ERR_PTR(-EAGAIN);
    current->state = TASK_INTERRUPTIBLE;
    .....
```

Sleep for a while

sys_msgrcv(cont.)

```
.....  
schedule();  
current->state = TASK_RUNNING;  
msg = (struct msg_msg *) msr_d.r_msg;  
if (!IS_ERR(msg))  
    goto out_success;  
t = msg_lock(msqid);  
if (t == NULL)  
    msqid = -1;  
msg = (struct msg_msg *) msr_d.r_msg;  
if (!IS_ERR(msg)) {  
    if (msqid != -1)  
        msg_unlock(msqid);  
    goto out_success;  
}  
err = PTR_ERR(msg);  
.....
```

sys_msgrcv(cont.)

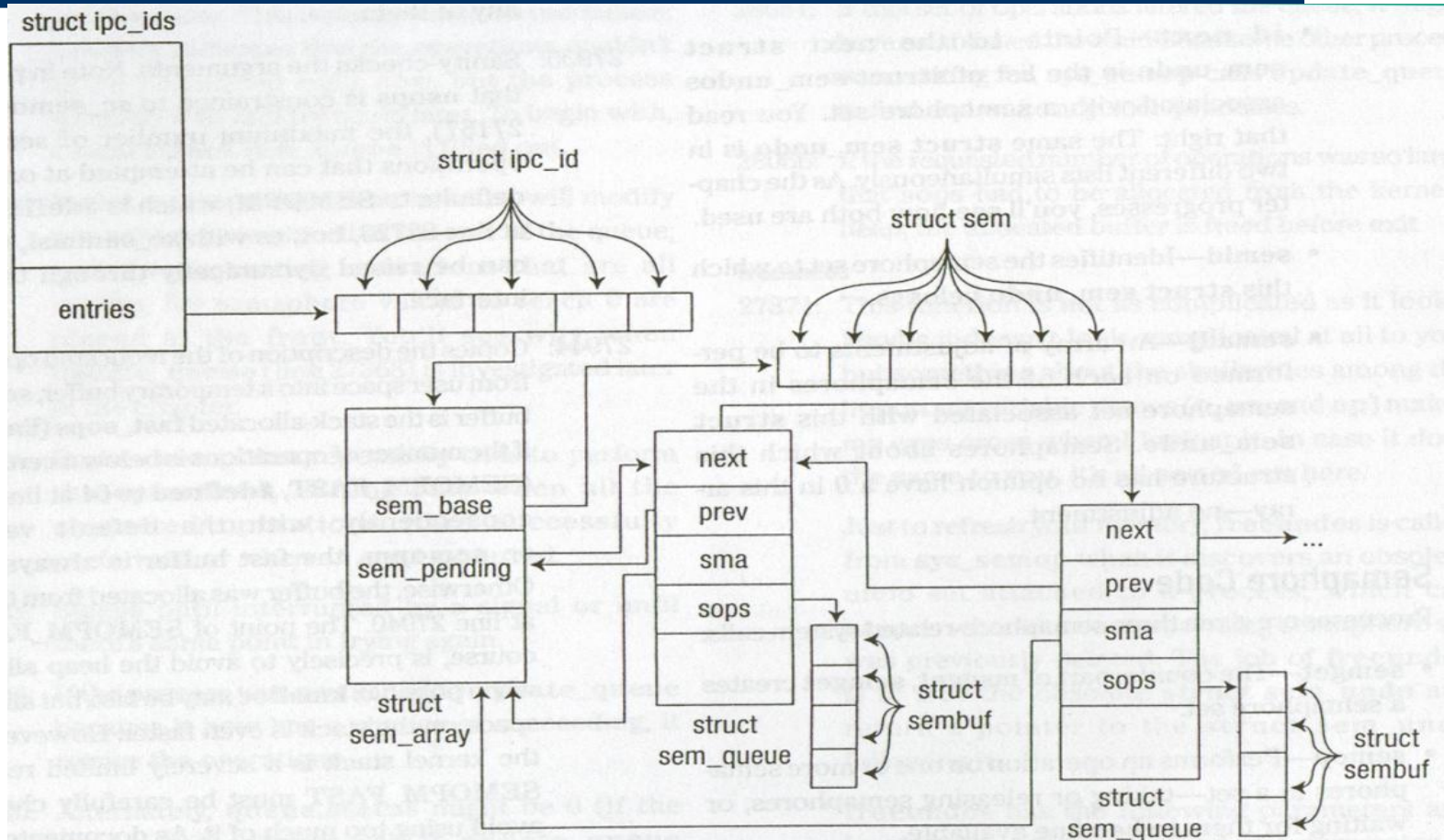
.....

```
if (err == -EAGAIN) {
    if (msqid == -1)
        BUG();
    list_del(&msr_d.r_list);
    if (signal_pending(current))
        err = -EINTR;
    else
        goto retry;
}
out_unlock:
if (msqid != -1)
    msg_unlock(msqid);
return err;
}
```

Semaphore

- | Example: Limited number of Keys to the door
- | Resources
 - Binary
 - Counted
- | Mutual exclusion(mutex)
- | Lock file

Semaphore data structure



sem & sem_array structure

```
struct sem {  
    int semval;           /* current value */  
    int sempid;         /* pid of last operation */  
};
```

If Positive or 0:
semval + 1 = keys
If Negative:
Num of process want to
access + 1

```
struct sem_array {  
    struct kern_ipc_perm sem_perm; /* permissions */  
    time_t sem_otime;             /* last semop time */  
    time_t sem_ctime;            /* last change time */  
    struct sem *sem_base;        /* ptr to first semaphore in array */  
    struct sem_queue *sem_pending; /* pending operations to be processed */  
    struct sem_queue **sem_pending_last; /* last pending operation */  
    struct sem_undo *undo;        /* undo requests on this array */  
    unsigned long sem_nsems;     /* no. of semaphores in array */  
};
```

sem_queue structure

```
struct sem_queue {
    struct sem_queue *next;    /* next entry in the queue */
    struct sem_queue **prev;  /* previous entry in the queue, *(q->prev) == q */
    struct task_struct *sleeper; /* this process */
    struct sem_undo *undo;    /* undo structure */
    int pid;                  /* process id of requesting process */
    int status;               /* completion status of operation */
    struct sem_array *sma;    /* semaphore array for operations */
    int id;                   /* internal sem id */
    struct sembuf *sops;     /* array of pending operations */
    int nsops;                /* number of operations*/
    int alter;                /* operation will alter semaphore */
};
```

sembuf structure

```
struct sembuf {
    unsigned short sem_num;    /* semaphore index in array */
    short sem_op;             /* semaphore operation */
    short sem_flg;            /* operation flags */
};

struct sem_undo {
    struct sem_undo *proc_next; /* next entry on this process */
    struct sem_undo *id_next;   /* next entry on this semaphore set */
    int semid;                  /* semaphore set identifier */
    short *semadj;              /* array of adjustments, one per semaphore */
};
```

3 semaphore system calls

I semget

- The counterpart of msgget, semget creates a semaphore set. (Nearly all the same in code)

I semop

- Performs an operation on one or more semaphores in a set – taking or releasing semaphores, or waiting for them to become available

I Semctl

- Like msgctl, semctl performs administrative operations on a semaphore set

sys_semop

```
asmlinkage long sys_semop(int semid, struct sembuf *tsops, unsigned nsops)
{
    int error = -EINVAL;
    struct sem_array *sma;
    struct sembuf fast_sops[SEMOPM_FAST];
    struct sembuf *sops = fast_sops, *sop;
    struct sem_undo *un;
    int undos = 0, decrease = 0,
    struct sem_queue queue;
    if (nsops < 1 || semid < 0)
        return -EINVAL;
    if (nsops > sc_semopm)
        return -E2BIG;
    if (nsops > SEMOPM_FAST) {
        sops = kmalloc(sizeof(*sops) * nsops, GFP_KERNEL);
        if (sops == NULL)
            return -ENOMEM;
    }
    .....
```

#define SEMOPM 32 /* <= 1 000 max num of
* ops per semop call */

#define SEMOPM_FAST 64

sys_semop(cont.)

```
.....  
if (copy_from_user(sops, tsops, nsops * sizeof(*tsops))) {  
    error = -EFAULT;  
    goto out_free;  
}  
sma = sem_lock(semid);  
error = -EINVAL;
```

Copy to a fast temp buffer
"sops"

```
.....  
for (sop = sops; sop < sops + nsops; sop++) {  
    if (sop->sem_num >= sma->sem_nsems)  
        goto out_unlock_free;  
    if (sop->sem_flg & SEM_UNDO)  
        undos++;  
    if (sop->sem_op < 0)   
        decrease = 1;  
    if (sop->sem_op > 0)  
        alter = 1;  
}  
alter |= decrease;
```

procure

vacate

```
.....
```

sys_semop(cont.)

```
.....  
if (ipcperms(&sma->sem_perm,alter ? S_IWUGO : S_IRUGO))  
    goto out_unlock_free;  
if (undos) {  
    un = current->semundo;  
    while (un != NULL) {  
        if (un->semid == semid)  
            break;  
        if (un->semid == -1)  
            un = freeundos(sma, un);  
        else  
            un = un->proc_next;  
    }  
    if (!un) {  
        error = alloc_undo(sma, &un, semid, alter);  
        if (error)  
            goto out_free;  
    }  
} else  
un = NULL;  
.....
```

sys_semop(cont.)

.....

0: success
<0: fail

```
error = try_atomic_semop(sma, sops, nsops, un, current->pid, 0);
if (error <= 0)
    goto update;
queue.sma = sma;
queue.sops = sops;
queue.nsops = nsops;
queue.undo = un;
queue.pid = current->pid;
queue.alter = decrease;
queue.id = semid;
if (alter)
    append_to_queue(sma, &queue);
else
    prepend_to_queue(sma, &queue);
current->semsleeping = &queue;
```

.....

We need to sleep on this operation, so we put the current task into the **pending queue** and go to sleep.

sys_semop(cont.)

```
.....
for (;;) {
struct sem_array *tmp;
queue.status = -EINTR;
queue.sleeper = current;
current->state = TASK_INTERRUPTIBLE;
sem_unlock(semid);
schedule();
.....
if (queue.status == 1) {
    error = try_atomic_semop(sma, sops, nsops, un,current->pid, 0);
    if (error <= 0)
        break;
    } else {
    error = queue.status;
    if (queue.prev)
        break;
    urrent->semsleeping = NULL;
    goto out_unlock_free;
    }
}
```

Awakened by update_queue

sys_semop(cont.)

.....

```
    current->semsleeping = NULL;
    remove_from_queue(sma, &queue);
update:
    if (alter)
        update_queue(sma);
out_unlock_free:
    sem_unlock(semid);
out_free:
    if (sops != fast_sops)
        kfree(sops);
    return error;
}
```

Remove the calling process
from the queue

try_atomic_semop

Determine whether a sequence of semaphore operations would succeed all at once. Return 0 if yes, 1 if need to sleep, else return error code.

```
static int try_atomic_semop(struct sem_array *sma, struct sembuf *sops, int nsops, struct
    sem_undo *un,int pid, int do_undo)
{
    int result, sem_op;
    struct sembuf *sop;
    struct sem *curr;
    for (sop = sops; sop < sops + nsops; sop++) {
        curr = sma->sem_base + sop->sem_num;
        sem_op = sop->sem_op;
        if (!sem_op && curr->semval)
            goto would_block;
        curr->sempid = (curr->sempid << 16) | pid;
        curr->semval += sem_op;
        if (sop->sem_flg & SEM_UNDO)
            un->semadj[sop->sem_num] -= sem_op;
        if (curr->semval < 0)
            goto would_block;
        if (curr->semval > SEMVMX)
            goto out_of_range;
    }
}
```

.....

try_atomic_semop (cont.)

```
.....  
  
if (do_undo) {  
    sop--;  
    result = 0;  
    goto undo;  
}  
sma->sem_otime = CURRENT_TIME;  
return 0;  
  
out_of_range:  
    result = -ERANGE;  
    goto undo;  
  
would_block:  
    if (sop->sem_flg & IPC_NOWAIT)  
        result = -EAGAIN;  
    else  
        result = 1;  
.....
```

try_atomic_semop (cont.)

.....

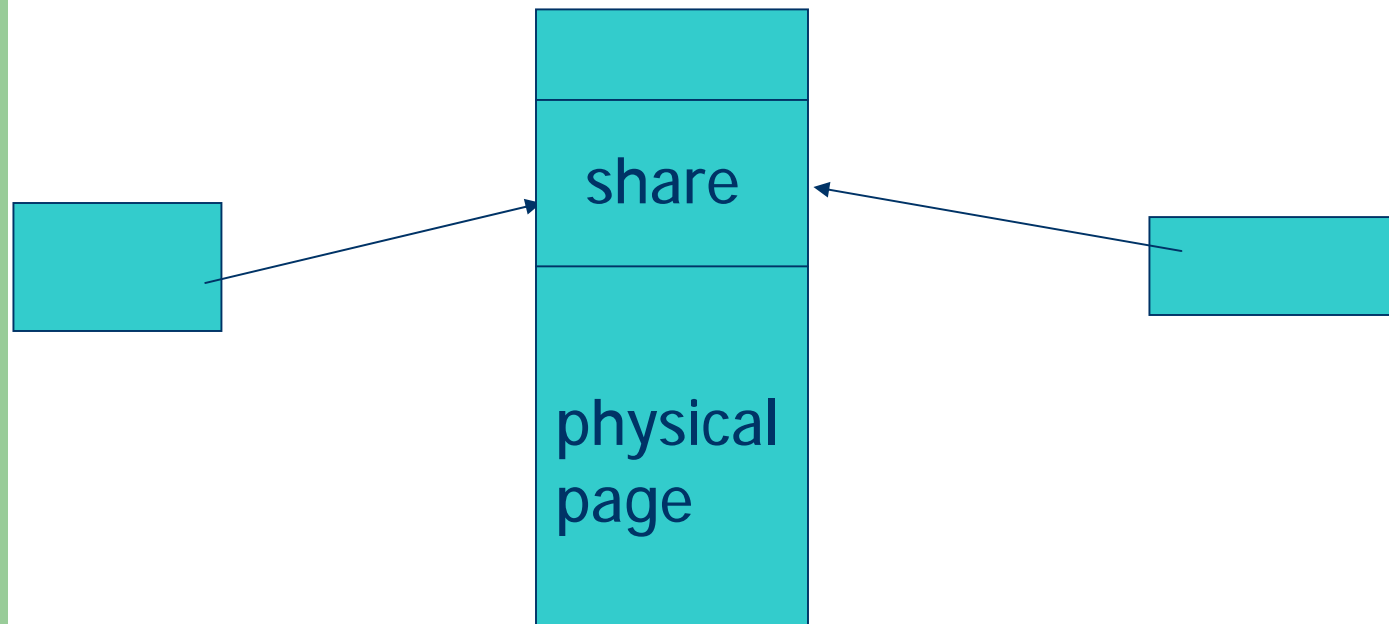
undo:

```
while (sop >= sops) {
    curr = sma->sem_base + sop->sem_num;
    curr->semval -= sop->sem_op;
    curr->sempid >>= 16;
    if (sop->sem_flg & SEM_UNDO)
        un->semadj[sop->sem_num] += sop->sem_op;
    sop--;
```

```
    }
return result;
}
```

Shared Memory

- | Fastest, easiest of the IPC services.
- | Mutual exclusion problem



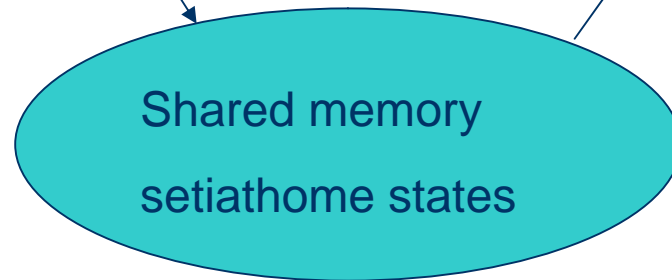
A share memory example: seti@home

Pure statistic computation

setiathome

Display in X-window

xsetiathome



shmid_kernel structure

```
struct shmid_kernel {      /* private to the kernel */  
  
    struct kern_ipc_perm shm_perm;  
    struct file *shm_file;  
    int id;  
    unsigned long shm_nattch;  
    unsigned long shm_segsz;  
    time_t shm_atim;  
    time_t shm_dtim;  
    time_t shm_ctim;  
    pid_t shm_cprid;  
    pid_t shm_lprid;  
  
};
```

The shared memory implementation is based on the concept of mmaping a file in memory

Size in bytes

Share memory system calls

- I `shmget`
 - Returns a unique id for a shared memory region
- I `shmat`
 - Attaches the calling process to a shared memory region
- I `shmdt`
 - Detaches the calling process from the shared memory region
- I `shmctl`
 - Performs administrative operations on a shared memory region

sys_shmat

```
asmlinkage long sys_shmat(int shmid, char *shmaddr, int shmflg, ulong * raddr)
{
    struct shmid_kernel *shp;
    unsigned long addr;
    struct file *file;
    int err;
    unsigned long flags;
    unsigned long prot;
    .....
    if ((addr = (ulong) shmaddr)) {
        if (addr & (SHMLBA - 1)) {
            if (shmflg & SHM_RND)
                addr &= ~(SHMLBA - 1); /* round down */
            else
                return -EINVAL;
        }
        flags = MAP_SHARED | MAP_FIXED;
    } else
        flags = MAP_SHARED;
    .....
```

Do_mmap will not be allowed to choose another address

sys_shmat (cont.)

```
if (shmflg & SHM_RDONLY) {  
    prot = PROT_READ;  
    o_flags = O_RDONLY;  
    acc_mode = S_IRUGO;  
} else {  
    prot = PROT_READ | PROT_WRITE;  
    o_flags = O_RDWR;  
    acc_mode = S_IRUGO | S_IWUGO;  
}  
shp = shm_lock(shmid);  
if (shp == NULL)  
    return -EINVAL;  
if (ipcperms(&shp->shm_perm, acc_mode)) {  
    shm_unlock(shmid);  
    return -EACCES;  
}  
file = shp->shm_file;  
shp->shm_nattch++;  
shm_unlock(shmid);  
.....
```

Read Only Permission

Attempt to map the fake file representing
the shared memory region

sys_shmat (cont.)

```
.....
down(&current->mm->mmap_sem);
user_addr = (void *) do_mmap(file, addr, file->f_dentry->d_inode->i_size, prot, flags, 0);
up(&current->mm->mmap_sem);
down(&shm_ids.sem);
if (!(shp = shm_lock(shmid)))
BUG();
shp->shm_nattch--;
if (shp->shm_nattch == 0 && shp->shm_flags & SHM_DEST)
shm_destroy(shp);
shm_unlock(shmid);
up(&shm_ids.sem);
*raddr = (unsigned long) user_addr;
err = 0;
if (IS_ERR(user_addr))
err = PTR_ERR(user_addr);
return err;
}
```

A bug? No.....

Use `do_mmap` to map the fake file representing the shared memory region into the calling process's memory space

No more attachment

sys_shmdt

```
asmlinkage long sys_shmdt(char *shmaddr)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *shmd, *shmdnext;

    down(&mm->mmap_sem);
    for (shmd = mm->mmap; shmd; shmd = shmdnext) {
        shmdnext = shmd->vm_next;
        if (shmd->vm_ops == &shm_vm_ops &&
            shmd->vm_start - (shmd->vm_pgoff << PAGE_SHIFT) == (ulong) shmaddr)
            do_munmap(mm, shmd->vm_start,
                    shmd->vm_end - shmd->vm_start);
    }
    up(&mm->mmap_sem);
    return 0;
}
```

Iterating over all the VMAs representing the process's memory

Calls `unmap_fixup()`, which calls `shm_close()`

shm_close

```
static void shm_close(struct vm_area_struct *shmd)
{
    struct file *file = shmd->vm_file;
    int id = file->f_dentry->d_inode->i_ino;
    struct shmid_kernel *shp;

    down(&shm_ids.sem);
    /* remove from the list of attaches of the shm segment*/
    if (!(shp = shm_lock(id)))
        BUG();
    shp->shm_lprid = current->pid;
    shp->shm_dtim = CURRENT_TIME;
    shp->shm_nattch--;
    if (shp->shm_nattch == 0 && shp->shm_flags & SHM_DEST)
        shm_destroy(shp);

    shm_unlock(id);
    up(&shm_ids.sem);
}
```

Reference

- I Linux Core Kernel Commentary
2nd Edition
- I Understanding the LINUX KERNEL
O'reilly
- I Cross-Reference Linux
 - <http://lxr.linux.no>