Presentation notes for Inter Process Communication

Pipes

1. *Structure pipe_inode_info (/usr/src/linux-2.4/include/linux/pipe_fs_i.h)*

```
5  struct pipe_inode_info {
6         wait_queue_head_t wait;  // Pipe/FIFO wait queue
7         char *base;  // address of buffer
8         unsigned int len;  // # bytes written to buffer yet to be read
9         unsigned int start; // offset of next byte to be read
10        unsigned int readers; // # readers
11        unsigned int writers; // # writers
12        unsigned int waiting_readers; // # reading processes in queue
13        unsigned int waiting_writers; // # writing processes in queue
14        unsigned int r_counter;
15        unsigned int w_counter;
16 };
```

2. *Mounting of pipefs special file system (/usr/src/linux-2.4/fs/pipe.c)*

```
630 static int __init init_pipe_fs(void)
631 {
632         int err = register_filesystem(&pipe_fs_type);
633         if (!err) {
634                 pipe_mnt = kern_mount(&pipe_fs_type);
635                 err = PTR_ERR(pipe_mnt);
636                 if (IS_ERR(pipe_mnt))
637                         unregister_filesystem(&pipe_fs_type);
638                 else
639                         err = 0;
640         }
641         return err;
642 }
```

3. Creation of a pipe

sys_pipe () function (/usr/src/linux-2.4/arch/i386/kernel/sys_i386.c)

```
29 asmlinkage int sys_pipe(unsigned long * fildes)
30 {
31         int fd[2];
32         int error;
33
34         error = do_pipe(fd);
35         if (!error) {
36                 if (copy_to_user(fildes, fd, 2*sizeof(int)))
37                         error = -EFAULT;
38         }
39         return error;
40 }
```

do_pipe () function (/usr/src/linux-2.4/fs/pipe.c)

```
505 int do_pipe(int *fd)
506 {
507         struct qstr this;
508         char name[32];
509         struct dentry *dentry;
510         struct inode * inode;
511         struct file *f1, *f2;
512         int error;
513         int i,j;
514
515         error = -ENFILE;
516         f1 = get_empty_filp();
517         if (!f1)
518                 goto no_files;
519
520         f2 = get_empty_filp();
521         if (!f2)
522                 goto close_f1;
523
524         inode = get_pipe_inode();
.
.
.
.
585 }
```

get_pipe_inode () function (/usr/src/linux-2.4/fs/pipe.c)

```
473 static struct inode * get_pipe_inode(void)
474 {
475         struct inode *inode = new_inode(pipe_mnt->mnt_sb);
476
477         if (!inode)
478                 goto fail_inode;
479
480         if(!pipe_new(inode))
481                 goto fail_iput;
482         PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 1;
.
.
.
.
503 }
```

pipe_new() function (/usr/src/linux-2.4/fs/pipe.c)

```
439 struct inode* pipe_new(struct inode* inode)
440 {
441         unsigned long page;
442
443         page = __get_free_page(GFP_USER);
444         if (!page)
445                 return NULL;
446
447         inode->i_pipe = kmalloc(sizeof(struct pipe_inode_info),
GFP_KERNEL);
448         if (!inode->i_pipe)
449                 goto fail_page;
450
451     init_waitqueue_head(PIPE_WAIT(*inode));
452     PIPE_BASE(*inode) = (char*) page;
453     PIPE_START(*inode) = PIPE_LEN(*inode) = 0;
454     PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 0;
455     PIPE_WAITING_READERS(*inode) = PIPE_WAITING_WRITERS(*inode)=0;
456         PIPE_RCOUNTER(*inode) = PIPE_WCOUNTER(*inode) = 1;
.
.
.
.
462 }
```

4. Reading from a pipe

pipe_read () function (/usr/src/linux-2.4/fs/pipe.c)

```
39 static ssize_t
40 pipe_read(struct file *filp, char *buf, size_t count, loff_t *ppos)
41 {
42         struct inode *inode = filp->f_dentry->d_inode;
43         ssize_t size, read, ret;
44
45         /* Seeks are not allowed on pipes.  */
46         ret = -ESPIPE;
47         read = 0;
48         if (ppos != &filp->f_pos)
49                 goto out_nolock;
50
51         /* Always return 0 on null read.  */
52         ret = 0;
53         if (count == 0)
54                 goto out_nolock;
55
56         /* Get the pipe semaphore */
57         ret = -ERESTARTSYS;
58         if (down_interruptible(PIPE_SEM(*inode)))
59                 goto out_nolock;
60
61         if (PIPE_EMPTY(*inode)) {
62 do_more_read:
63                 ret = 0;
64                 if (!PIPE_WRITERS(*inode))
65                         goto out;
66
67                 ret = -EAGAIN;
68                 if (filp->f_flags & O_NONBLOCK)
69                         goto out;
70
71                 for (;;) {
72                         PIPE_WAITING_READERS(*inode)++;
73                         pipe_wait(inode);
74                         PIPE_WAITING_READERS(*inode)--;
.
.
.
.
84         }
85
86 out:        /* Read what data is available.  */

87         ret = -EFAULT;
88         while (count > 0 && (size = PIPE_LEN(*inode))) {
89                 char *pipebuf = PIPE_BASE(*inode) +
PIPE_START(*inode);
90                 ssize_t chars = PIPE_MAX_RCHUNK(*inode);
91
92                 if (chars > count)
93                         chars = count;
94                 if (chars > size)
```

```
95                          chars = size;
96
97                  if (copy_to_user(buf, pipebuf, chars))
98                          goto out;
99
100                 read += chars;
101                 PIPE_START(*inode) += chars;
102                 PIPE_START(*inode) &= (PIPE_SIZE - 1);
103                 PIPE_LEN(*inode) -= chars;
104                 count -= chars;
105                 buf += chars;
106         }
.
.
.
.

112         if (count && PIPE_WAITING_WRITERS(*inode) && !(filp-
>f_flags & O_NONBLOCK)) {
113                 /*
114                  * We know that we are going to sleep: signal
115                  * writers synchronously that there is more
116                  * room.
117                  */
118                 wake_up_interruptible_sync(PIPE_WAIT(*inode));
119                 if (!PIPE_EMPTY(*inode))
120                         BUG();
121                 goto do_more_read;
122         }
123         /* Signal writers asynchronously that there is more room.
*/
124         wake_up_interruptible(PIPE_WAIT(*inode));
125
126         ret = read;
127 out:
128         up(PIPE_SEM(*inode));
129 out_nolock:
130         if (read)
131                 ret = read;
132
133         UPDATE_ATIME(inode);
134         return ret;
135 }
```

5. *pipe_wait() function (/usr/src/linux-2.4/fs/pipe.c)*

```
27 void pipe_wait(struct inode * inode)
28 {
29          DECLARE_WAITQUEUE(wait, current);
30          current->state = TASK_INTERRUPTIBLE;
31          add_wait_queue(PIPE_WAIT(*inode), &wait);
32          up(PIPE_SEM(*inode));
33          schedule();
34          remove_wait_queue(PIPE_WAIT(*inode), &wait);
35          current->state = TASK_RUNNING;
36          down(PIPE_SEM(*inode));
37 }
```

6. Writing to a pipe

pipe_write () function (/usr/src/linux-2.4/fs/pipe.c)

```
137 static ssize_t
138 pipe_write(struct file *filp, const char *buf, size_t count, loff_t
*ppos)
139 {
140         struct inode *inode = filp->f_dentry->d_inode;
.
.
.
.

153         /* Acquire the semaphore.  */
154         ret = -ERESTARTSYS;
155         if (down_interruptible(PIPE_SEM(*inode)))
156                 goto out_nolock;
157
158         /* No readers yields SIGPIPE.  */
159         if (!PIPE_READERS(*inode))
160                 goto sigpipe;
161
162         /* If count <= PIPE_BUF, we have to make it atomic.  */
163         free = (count <= PIPE_BUF ? count : 1);
164
165         /* Wait, or check for, available space.  */
166         if (filp->f_flags & O_NONBLOCK) {
167                 ret = -EAGAIN;
168                 if (PIPE_FREE(*inode) < free)
169                         goto out;
170         } else {
171                 while (PIPE_FREE(*inode) < free) {
172                         PIPE_WAITING_WRITERS(*inode)++;
173                         pipe_wait(inode);
174                         PIPE_WAITING_WRITERS(*inode)--;
175                         ret = -ERESTARTSYS;
176                         if (signal_pending(current))
177                                 goto out;
178
179                         if (!PIPE_READERS(*inode))
```

```
180                                     goto sigpipe;
181                             }
182                     }
183
184                     /* Copy into available space.  */
185                     ret = -EFAULT;
186                     while (count > 0) {
187                             int space;
188                             char *pipebuf = PIPE_BASE(*inode) +
PIPE_END(*inode);
189                             ssize_t chars = PIPE_MAX_WCHUNK(*inode);
190
191                             if ((space = PIPE_FREE(*inode)) != 0) {
192                                     if (chars > count)
193                                             chars = count;
194                                     if (chars > space)
195                                             chars = space;
196
197                                     if (copy_from_user(pipebuf, buf, chars))
198                                             goto out;
199
200                                     written += chars;
201                                     PIPE_LEN(*inode) += chars;
202                                     count -= chars;
203                                     buf += chars;
204                                     space = PIPE_FREE(*inode);
205                                     continue;
206                             }
207
208                             ret = written;
209                             if (filp->f_flags & O_NONBLOCK)
210                                     break;
.
.
.
.
230                     /* Signal readers asynchronously that there is more data.
*/
231                     wake_up_interruptible(PIPE_WAIT(*inode));
232
233                     inode->i_ctime = inode->i_mtime = CURRENT_TIME;
234                     mark_inode_dirty(inode);
235
236 out:
237                     up(PIPE_SEM(*inode));
238 out_nolock:
239                     if (written)
240                             ret = written;
241                     return ret;
242
243 sigpipe:
244                     if (written)
245                             goto out;
246                     up(PIPE_SEM(*inode));
247                     send_sig(SIGPIPE, current, 0);
248                     return -EPIPE;
249 }
```

FIFOs

1. Opening a FIFO

fifo_open() function (/usr/src/linux-2.4/fs/fifo.c)

```
31 static int fifo_open(struct inode *inode, struct file *filp)
32 {
.
.
.
.
37         if (down_interruptible(PIPE_SEM(*inode)))ret = -ENOMEM;
42                 if(!pipe_new(inode))
.
.
.
.
47         switch (filp->f_mode) {
48         case 1:
49         /*
50          *  O_RDONLY
51          *  POSIX.1 says that O_NONBLOCK means return with the FIFO
52          *  opened, even when there is no process writing the FIFO.
53          */
.
.
.
.
55                 PIPE_RCOUNTER(*inode)++;
56                 if (PIPE_READERS(*inode)++ == 0)
57                         wake_up_partner(inode);
58
59                 if (!PIPE_WRITERS(*inode)) {
60                         if ((filp->f_flags & O_NONBLOCK)) {
61                                 /* suppress POLLHUP until we have
62                                  * seen a writer */
63                                 filp->f_version =
PIPE_WCOUNTER(*inode);
64                         } else
65                         {
66                                 wait_for_partner(inode,
&PIPE_WCOUNTER(*inode));
67                                 if(signal_pending(current))
68                                         goto err_rd;
69                         }
70                 }
71                 break;
72
73         case 2:
74         /*
75          *  O_WRONLY
76          *  POSIX.1 says that O_NONBLOCK means return -1 with
77          *  errno=ENXIO when there is no process reading the FIFO.
78          */
79                 ret = -ENXIO;
```

```
80                  if ((filp->f_flags & O_NONBLOCK)
&& !PIPE_READERS(*inode))
81                          goto err;
82
83                  filp->f_op = &write_fifo_fops;
84                  PIPE_WCOUNTER(*inode)++;
85                  if (!PIPE_WRITERS(*inode)++)
86                          wake_up_partner(inode);
87
88                  if (!PIPE_READERS(*inode)) {
89                          wait_for_partner(inode,
&PIPE_RCOUNTER(*inode));
90                          if (signal_pending(current))
91                                  goto err_wr;
92                  }
93                  break;
94
95          case 3:
96          /*
97           *  O_RDWR
98           *  POSIX.1 leaves this case "undefined" when O_NONBLOCK is
set.
99           *  This implementation will NEVER block on a O_RDWR open,
since
100          *  the process can at least talk to itself.
101          */
102                 filp->f_op = &rdwr_fifo_fops;
103
104                 PIPE_READERS(*inode)++;
105                 PIPE_WRITERS(*inode)++;
106                 PIPE_RCOUNTER(*inode)++;
107                 PIPE_WCOUNTER(*inode)++;
108                 if (PIPE_READERS(*inode)==1||(*inode) == 1)
109                         wake_up_partner(inode);
110                 break;
111


.
.
.
.
139                 kfree(info);
140
.
.
.
.
```

System V IPC

1. Structure ipc_ids (/usr/src/linux-2.4/ipc/util.h)

```
15 struct ipc_ids {
16       int size;   //current max # or resources
17       int in_use; // # of allocated resources
18       int max_id;  // max slot index in use
19       unsigned short seq; // slot usage sequence # for next alloc.
20       unsigned short seq_max; // max seq
21       struct semaphore sem;   // semaphore structure protecting race
22         spinlock_t ary;   // spinlock protecting resources
23         struct ipc_id* entries;  // array of IPC descriptors
24 };
```

2. Structure kern_ipc_perm (include/linux/ipd.h)

```
57 struct kern_ipc_perm
58 {
59         key_t           key;
60         uid_t           uid;
61         gid_t           gid;
62         uid_t           cuid;
63         gid_t           cgid;
64         mode_t          mode;
65         unsigned long   seq;  // used to compute IPC id
66 };
```

Semaphores

1. Structure sem_array (include/linux/sem.h)

```
88 struct sem_array {
89     struct kern_ipc_perm    sem_perm;
90     time_t                  sem_otime; // timestamp of last semop()
91     time_t                  sem_ctime; // time of last change
92     struct sem              *sem_base;  // ptr to first sem struct
93     struct sem_queue        *sem_pending; // pending operations
94     struct sem_queue        **sem_pending_last;
95     struct sem_undo         *undo;  // undo struct
96     unsigned long           sem_nsems; // # of semaphores
97 };
```

2. Structure sem (include/linux/sem.h)

```
82 struct sem {
83         int     semval;
84         int     sempid;
85 };
```

3. Structure sem_queue (include/linux/sem.h)

```
100 struct sem_queue {
101       struct sem_queue *      next;
102       struct sem_queue **     prev;
103       struct task_struct*     sleeper;
104       struct sem_undo *       undo;
105       int                     pid;
106       int                     status;
107       struct sem_array *      sma; // ptr of semaphore descriptor
108       int                     id;  // slot index
109       struct sembuf *         sops; // array of pending operations
110       int                     nsops; // # pending operations
111       int                     alter; // flag for setting sem value
112 };
```

4. Structure sem_undo (include/linux/sem.h)

```
117 struct sem_undo {
118         struct sem_undo *       proc_next;
119         struct sem_undo *       id_next;
120         int                     semid;
121         short *                 semadj;
122 };
```

5. Structure semun (include/linux/sem.h)

```
45 union semun {
46         int val;
47         struct sem_arrary *buf;
48         unsigned short *array;
49
50 };
```

5. Structure sem buf (include/linux/sem.h)

```
38 struct sembuf {
39         unsigned short  sem_num;
40         short           sem_op;
41         short           sem_flg;  // flag SEM_UNDO, NO_WAIT etc
42 };
```

7. semget() (/usr/src/linux-2.4/ipc/sem.c)

```
152 asmlinkage long sys_semget (key_t key, int nsems, int semflg)
153 {
154         int id, err = -EINVAL;
155         struct sem_array *sma;
156
157         if (nsems < 0 || nsems > sc_semmsl)
158                 return -EINVAL;
159         down(&sem_ids.sem);
160
161         if (key == IPC_PRIVATE) {
162                 err = newary(key, nsems, semflg);
163         } else if ((id = ipc_findkey(&sem_ids, key)) == -1) {  /*
key not used */
164                 if (!(semflg & IPC_CREAT))
165                         err = -ENOENT;
166                 else
167                         err = newary(key, nsems, semflg);
168         } else if (semflg & IPC_CREAT && semflg & IPC_EXCL) {
169                 err = -EEXIST;
170         } else {
171                 sma = sem_lock(id);
172                 if(sma==NULL)
173                         BUG();
174                 if (nsems > sma->sem_nsems)
175                         err = -EINVAL;
176                 else if (ipcperms(&sma->sem_perm, semflg))
177                         err = -EACCES;
178                 else
179                         err = sem_buildid(id, sma->sem_perm.seq);
180                 sem_unlock(id);
181         }
182
183         up(&sem_ids.sem);
184         return err;
185 }
```

8. semctl()(/usr/src/linux-2.4/ipc/sem.c)

```
755 asmlinkage long sys_semctl (int semid, int semnum, int cmd, union
semun arg)
756 {
757         int err = -EINVAL;
758         int version;
759
760         if (semid < 0)
761                 return -EINVAL;
762
763         version = ipc_parse_version(&cmd);
764
765         switch(cmd) {
766         case IPC_INFO:
767         case SEM_INFO:
768         case SEM_STAT:
769                 err = semctl_nolock(semid,semnum,cmd,version,arg);
770                 return err;
771         case GETALL:
772         case GETVAL:
773         case GETPID:
774         case GETNCNT:
775         case GETZCNT:
776         case IPC_STAT:
777         case SETVAL:
778         case SETALL:
779                 err = semctl_main(semid,semnum,cmd,version,arg);
780                 return err;
781         case IPC_RMID:
782         case IPC_SET:
783                 down(&sem_ids.sem);
784                 err = semctl_down(semid,semnum,cmd,version,arg);
785                 up(&sem_ids.sem);
786                 return err;
787         default:
788                 return -EINVAL;
789         }
790 }
791
```

9. semop()(/usr/src/linux-2.4/ipc/sem.c)

```
840 asmlinkage long sys_semop (int semid, struct sembuf *tsops,
unsigned nsops)
841 {
842         int error = -EINVAL;
843         struct sem_array *sma;
844         struct sembuf fast_sops[SEMOPM_FAST];
845         struct sembuf* sops = fast_sops, *sop;
846         struct sem_undo *un;
847         int undos = 0, decrease = 0, alter = 0;
848         struct sem_queue queue;
849
850         if (nsops < 1 || semid < 0)
851                 return -EINVAL;
852         if (nsops > sc_semopm)
853                 return -E2BIG;
854         if(nsops > SEMOPM_FAST) {
855                 sops = kmalloc(sizeof(*sops)*nsops,GFP_KERNEL);
856                 if(sops==NULL)
857                         return -ENOMEM;
858         }
859         if (copy_from_user (sops, tsops, nsops * sizeof(*tsops))) {
860                 error=-EFAULT;
861                 goto out_free;
862         }
863         sma = sem_lock(semid);
864         error=-EINVAL;
865         if(sma==NULL)
866                 goto out_free;
867         error = -EIDRM;
868         if (sem_checkid(sma,semid))
869                 goto out_unlock_free;
870         error = -EFBIG;
871         for (sop = sops; sop < sops + nsops; sop++) {
872                 if (sop->sem_num >= sma->sem_nsems)
873
.
.
.
.                       goto out_unlock_free;
887                 /* Make sure we have an undo structure
888                  * for this process and this semaphore set.
889                  */
890                 un=current->semundo;
.
.
.
.
906
907         error = try_atomic_semop (sma, sops, nsops, un, current-
>pid, 0);
908         if (error <= 0)
909                 goto update;
```

```
924         if (alter)
923                 append_to_queue(sma ,&queue);
924         else
925                 prepend_to_queue(sma ,&queue);
926         current->semsleeping = &queue;
927
928         for (;;) {
929                 struct sem_array* tmp;
930                 queue.status = -EINTR;
931                 queue.sleeper = current;
932                 current->state = TASK_INTERRUPTIBLE;
933
935                 schedule();
.
.
.
.
967         current->semsleeping = NULL;
968         remove_from_queue(sma,&queue);
969 update:
970         if (alter)
971                 update_queue (sma);
972 out_unlock_free:
973         sem_unlock(semid);
974 out_free:
975         if(sops != fast_sops)
976                 kfree(sops);
977         return error;
978 }
```

Message Queues

1. Structure msg_queue (include/linux/msg.h)

```
15 struct msg_queue {
16         struct ipc_perm msg_perm;
17         struct msg *msg_first;
18         struct msg *msg_last;
19         __kernel_time_t msg_stime;
20         __kernel_time_t msg_rtime;
21         __kernel_time_t msg_ctime;
22         unsigned long  msg_lcbytes;
23         unsigned long  msg_lqbytes;
24         unsigned short msg_cbytes;
25         unsigned short msg_qnum;
26         unsigned short msg_qbytes;
27         __kernel_ipc_pid_t msg_lspid;
28         __kernel_ipc_pid_t msg_lrpid;
29 };
```

2. Structure msg_msg (include/linux/msg.h)

```
35 struct msgbuf {
36         long mtype;         /* type of message */
37         char mtext[1];      /* message text */
38 };
```

Shared Memory

1. Structure shmid_kernel (include/linux/shm.h)

```
21 struct shmid_kernel {
22         struct ipc_perm         shm_perm;
23          struct file*            shm_file
23         int                     shm_segsz;
24         __kernel_time_t         shm_atime;
25         __kernel_time_t         shm_dtime;
26         __kernel_time_t         shm_ctime;
27         __kernel_ipc_pid_t      shm_cpid;
28         __kernel_ipc_pid_t      shm_lpid;
29         unsigned short          shm_nattch;
30         unsigned short          shm_unused;
31 };
```