INTERRUPT HANDLING AND INTERPROCESS COMMUNICATIONS

* Partially Adopted “Interrupts in Linux, Prof. Kaustubh R. Joshi”

2016 Operating Systems Design
Euiseong Seo (euiseong@skku.edu)
Why Interrupts

- Devices require a prompt response from the CPU when various events occur, even when the CPU is busy running a program
- Need a mechanism for a device to gain CPU’s attention
Kernel as a Multithreaded Server

- Process
- Process
- Process
- Kernel

Data structures
In common address space

Syscall

Interrupt

I/O device

Timer

I/O device

Timer
Interrupt Handling

- Forcibly change normal flow of control
- Similar to context switch (but lighter weight)
  - Hardware saves some context on stack; Includes interrupted instruction if restart needed
  - Enters kernel at a specific point; kernel then figures out which interrupt handler should run
  - Execution resumes with special ‘iret’ instruction
- Many different types of interrupts
Types of Interrupts

- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed

- Synchronous (also called exceptions)
  - Processor-detected exceptions
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - Programmed exceptions
    - Requests for kernel intervention (software intr/sysealls)
Interrupt Handling

- Do as little as possible in the interrupt handler
- Defer non-critical actions till later
- Three types of actions
  - Critical: Top-half (interrupts disabled – briefly!)
    - Example: acknowledge interrupt
  - Non-critical: Top-half (interrupts enabled)
    - Example: read key scan code, add to buffer
  - Non-critical deferrable: Bottom half, do it “later” (interrupts enabled)
    - Example: copy keyboard buffer to terminal handler process
    - Soairqs, tasklets
- No process context available
Deferrable Work

- We don't want to do too much in regular interrupt handlers
  - Interrupts are masked
  - We don't want the kernel stack to grow too much
- Instead, interrupt handlers schedule work to be performed later
- Three deferred work mechanisms: softirqs, tasklets, and work queues
- Tasklets are built on top of softirqs
- For all of these, requests are queued
Softirqs

- Statically allocated: specified at kernel compile
- Limited number:
  - Priority 0: High-priority tasklets
  - Priority 1: Timer interrupts
  - Priority 2: Network transmission
  - Priority 3: Network reception
  - Priority 4: SCSI disks
  - Priority 5: Regular tasklets
Running Softirqs

- Run at various points by the kernel
- Most important: after handling IRQs and after timer interrupts
- Softirq routines can be executed simultaneously on multiple CPUs:
  - Code must be re-entrant
  - Code must do its own locking as needed
Rescheduling Softirqs

- A softirq routine can reschedule itself
- This could starve user-level processes
- Softirq scheduler only runs a limited number of requests at a time
- The rest are executed by a kernel thread, ksoftirqd, which competes with user processes for CPU time
Tasklets

- Similar to soairqs
- Created and destroyed dynamically
- Individual tasklets are locked during execution; no problem about re-entrancy, and no need for locking by the code
- Only one instance of tasklet can run, even with multiple CPUs
- Preferred mechanism for most deferred activity
Work Queues

- Always run by kernel threads
- Softirqs and tasklets run in an interrupt context; work queues have a process context
- Because they have a process context, they can sleep
- However, they’re kernel-only; there is no user mode associated with it
Interprocess Communications
Named Pipes

- Anonymous pipes can be used only between related processes
- Processes not from the same ancestor sometimes need to communicate with each other
- FIFO
  - A type of files (S_ISFIFO macro tests against st_mode)
  - Usually called named pipes
- Multiple readers/writers are allowed
Uses for FIFOs

- Used by shell commands to pass data from one shell pipeline to another without creating temporary files.
- Used as rendezvous points in client-server applications to pass data between clients and servers.
Duplicate Output Streams with FIFOs

Interprocess Communication

Chapter 15

Interleaved. As with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO.

There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

We discuss each of these uses with an example.

Example — Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files). But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice.

Figure 15.20 shows this arrangement.

With a FIFO and the UNIX program tee (1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

Example — Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known...
Client-Server Communication using FIFOs

**Figure 15.23**
Client–server communication using FIFOs

- **Server**
  - Receives requests from clients.
  - Sends replies to clients.

- **Well-known FIFO**
  - Read-only by servers.
  - Read-write access by clients.

- **Client-specific FIFOs**
  - Clients write requests.
  - Server reads requests.
  - Server writes replies.
  - Clients read replies.

- **Request Flow**
  - Client sends request to well-known FIFO.
  - Server reads request and sends reply.
  - Client reads reply.

- **Server Handling**
  - Server must catch SIGPIPE, since a client might terminate before reading the response.
  - Server-specific FIFOs may be read by multiple clients.

- **Exercise 15.10**
  - Server can open its well-known FIFO for read–write access.

---

**15.6 XSI IPC**

The three types of IPC that we call XSI IPC—message queues, semaphores, and shared memory—have many similarities. In this section, we cover these similar features; in the following sections, we look at the specific functions for each of the three IPC types.

- **Identifier and Keys**
  - Each IPC structure (message queue, semaphore, or shared memory) is referred to by a non-negative integer identifier.
  - To send a message to or fetch a message from a message queue, all we need know is the identifier for the queue.
  - Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a client sends a message, it needs to know the identifier of the message queue.

---

**AT&T's Columbus UNIX**

The AT&T version of the UNIX System called "Columbus UNIX" was a 1970s development that included these IPC features. They were later added to System V and have often been criticized for inventing their own namespace instead of using the file system.
Creating a FIFO

Prototype

```c
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```
Accessing a FIFO

- Like a normal file access, a process opens a FIFO for read, write or read/write accesses
- `open()` for read-only blocks until some other process opens FIFO for writing
- `open()` for write-only blocks until some other process opens FIFO for reading
- `open()` for read-only with O_NONBLOCK returns immediately
- `open()` for write-only with O_NONBLOCK returns -1 if no process has FIFO open for reading
Close and Delete a FIFO

- Close a FIFO with `close()` when you no longer use it
- Delete a FIFO with `unlink()`
System V IPC (XSI IPC)

- Message queues, semaphores and shared memory
- Message queues
  - A linked list of messages stored within kernel
- Semaphore
  - A counter for sharing data object for multiple processes
- Shared memory
  - A memory region that can be shared by multiple processes
Key and Identifier

- All IPC objects are stored in kernel memory
- They are referred to by IPC object identifier

```
euiseong@accept:~/Temp$ ipcs

----- Shared Memory Segments ------
key   shmid   owner   perms   bytes   nattch   status

----- Semaphore Arrays --------
key   semid   owner   perms   nsems

----- Message Queues --------
key   msgid   owner   perms   used-bytes  messages
0x000004d2 32768 euiseong 666 2304     8
```

- ID is a unique large number assigned by kernel when you create an IPC structure with a key
- Server and client rendezvous with key, not with ID
- Kernel will translate a given key to corresponding ID
Message Queue

- Data structures

```c
struct msqid_ds {
    struct ipc_perm msg_perm;
    msgqnum_t msg_qnum;  // # of messages on queue
    msglen_t msg_qbytes; // max # of bytes on queue
    pid_t msg_lspid;     // pid of last msgsnd()
    pid_t msg_lrpid;     // pid of last msgrcv()
    time_t msg_stime;   // last-msgsnd() time
    time_t msg_rtime;   // last-msgrcv() time
    time_t msg_ctime;   // last-change time
};

struct ipc_perm {
    uid_t uid;     // owner's EUID
    gid_t gid;     // owner's EGID
    uid_t cuid;    // creator's EUID
    gid_t cgid;    // creator's EGID
    mode_t mode;   // access mode
};
```
Creating a Message Queue

- **Prototype**

  ```c
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>

  int msgget(key_t key, int msgflg);
  ```

- `msgget()` returns a non-negative queue ID
- `msgflg` can be a combination of flags
  - `IPC_CREAT`
  - `IPC_EXCL`
Controlling a Message Queue

- **Prototype**
  ```c
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>

  int msgctl(int msqid, int cmd, struct msqid_ds *buf);
  ```

- **cmd** can be one of the followings
  - IPC_STAT
  - IPC_SET
  - IPC_RMID
Sending and Receiving via Queue

**Prototype**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

**Data structure**

```c
struct msgbuf {
    long mtype; /* message type, must be > 0 */
    char mtext[1024]; /* message data */
};
```
Sending and Receiving via Queue

msgtyp can be one of followings
- 0: first message in queue
- >0: first message in queue of type msgtyp unless MSG_EXCEPT in msgflg
- <0: first message in queue with lowest type less than or equal to absolute value of msgtyp
IPC-Related Commands

- **ipcs**
  - Provide information on IPC facilities

- **ipcrm**
  - Remove a message queue, semaphore set or shared memory
  - Users can select an IPC facility to remove with its key or ID
Example - Producer

```c
struct msgbuf {
    long msgtype;
    char mtext[256];
    int seq;
};

int main() {
    time_t curtime;
    key_t key_id;
    int i;
    struct msgbuf sendbuf;

    key_id = msgget(2014, IPC_CREAT|0666);
    if (key_id == -1)
        {perror("msgget error : ");
            exit(0);} 
    memset(sendbuf.mtext, 0x0, 256);
    sendbuf.seq = 0;
    for(i = 0; i<100; i++) {
        sendbuf.seq = i;
        if (i % 2 == 0)
            sendbuf.msgtype = 2;
        else
            sendbuf.msgtype = 1;
        time(&curtime);
        strcpy(sendbuf.mtext, ctime(&curtime));
        if (msgsnd(key_id, (void * )&sendbuf, sizeof(struct msgbuf), IPC_NOWAIT)< 0)
            {perror("msgsnd error : ");
            exit(0);}
        sleep(1);
    }
} 
```
Example - Consumer

```c
struct msgbuf {
    long msgtype;
    char mtext[256];
    int seq;
};

int main(int argc, char **argv) {
    key_t key_id;
    struct msgbuf recvbuf;
    int msgtype;

    if (argc >= 2 &&
        strcmp(argv[1], "2") == 0)
        msgtype = 2;
    else
        msgtype = 1;

    key_id = msgget(2014, IPC_CREAT|0666);
    if (key_id < 0)
        { perror("msgget error : ");
            exit(0);}

    while (1) {
        if (msgrecv(1024, (void *)&recvbuf,
                    sizeof(struct msgbuf), msgtype, 0) == -1)
            { perror("msgrecv"); exit(0);}
        printf("%d\t%s\n", recvbuf.seq, recvbuf.mtext);
    }
    exit(0);
}
```
Semaphore

- Procedure to obtain a shared resource
  1. Test semaphore that controls resource
  2. If value of semaphore is positive, process can use resource by decrementing semaphore by 1
  3. If value of semaphore is 0, sleep and wait till it becomes 1

- Process increases semaphore value by 1 after it is done with shared resource

- Initial value of semaphore determines how many processes can concurrently access shared resource

- Semaphore operation is atomic (non-interruptible)
Creating Semaphores

- **Prototype**

  ```c
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/sem.h>

  int semget(key_t key, int nsems, int semflg);
  ```

- Similar to message queue operations
- This creates or obtains an array of semaphores
- Number of semaphores is nsems
Controlling Semaphores

- **Prototype**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

- You can do followings with `semctl()`
  - Read a semaphore member value
  - Remove a semaphore array
  - Read permission or owner information
  - And so on…
Semaphore Operations

- **Prototype**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);

struct sembuf {
    unsigned short sem_num; // member index
    short sem_op; // negative, 0 or positive
    short sem_flg; // IPC_NOWAIT, SEM_UNDO
};
```

- `semop()` blocks when `sem_op` is negative, `sem_num`-th semaphore is smaller than `sem_op` and `sem_flg` is not `IPC_NOWAIT`

- Otherwise `sem_op` will be added to `sem_num`-th semaphore and return
**Shared Memory**

- Multiple process can map the same contents in their address space by using `mmap()` to the same file.
- IPC shared memory provide shared memory area that can be mapped into processes’ address space.
- IPC shared memory does not use an actual file as its medium.
Creating a Shared Memory Region

- **Prototype**
  
  ```c
  #include <sys/shm.h>

  int shmget(key_t key, size_t size, int shmflg);
  ```

- size-byte memory space will be created in kernel
Control, Mapping and Unmapping of SHM

Prototype

```c
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

- **shmat()** maps shmid into shmaddr
  - For portability, let shmaddr be 0
    - Kernel will choose appropriate address
  - Returns actual mapped address, which will be used by shmdt() to unmap
Example - Producer

```c
int main()
{
    time_t curtime;
    key_t key_id;
    int semid, shmid;
    union semun newcmd;
    int i;
    void *shmaddr;
    char strbuffer[256];
    struct sembuf oparr;
    semid = semget(2015, 1, IPC_CREAT|0666);
    newcmd.val = 1;
    semctl(semid, 0, SETVAL, newcmd);
    shmid=shmget(2016, 256,IPC_CREAT|0666);
    shmaddr=shmat(shmid, 0, 0);
    memset(shmaddr, 0x0, 256);
    oparr.sem_num = 0;
    oparr.sem_flg = 0;
    for(i = 0;i<100;i++) {
        time(&curtime);
        sprintf(strbuffer,"%d:%s",i,ctime(&curtime));
        oparr.sem_op = -1;
        semop(semid, &oparr, 1);
        strcpy(shmaddr, strbuffer);
        oparr.sem_op = 1;
        semop(semid, &oparr, 1);
        sleep(1);
    }
}
```
```c
int main()
{
    time_t curtime;
    key_t key_id;
    int semid, shmid;
    union semun newcmd;
    int i;
    void *shmaddr;
    char strbuffer[256];
    struct sembuf oparr;

    semid = semget(2015, 1, IPC_CREAT|0666);
    newcmd.val = 1;
    semctl(semid, 0, SETVAL, newcmd);

    shmid = shmget(2016, 256, IPC_CREAT|0666);
    shmaddr=shmat(shmid, 0, 0);
    while(1) {
        oparr.sem_num = 0;
        oparr.sem_op = -1;
        oparr.sem_flg = 0;
        semop(semid, &oparr, 1);
        if(*((int *)shmaddr == 0) {
            oparr.sem_op = 1;
            semop(semid, &oparr, 1);
        } else {
            printf("%s", (char *)shmaddr);
            memset(shmaddr, 0x0, 256);
            oparr.sem_op = 1;
            semop(semid, &oparr, 1);
        }
    }
}
```