Introduction (1)

Operating system control I/O devices for three reasons:

- The hardware interface to such devices is crude, requiring complex software packets for their control called *device drivers*
- device drivers are *shared resources*, witch need to be protected and allocated in a fair and safe way
- A uniform, flexible interface should be provided, - A high-level interface - allowing users to write programs without knowing the machine configuration.

Introduction (2)

Discussed in this chapter is: the selection of a set of machine-independent high-level I/O primitives and data structures required to relate these primitives to specific devices

How are these primitives selected ?

- By generating a list of desirable properties
- Delivering a set of high-level primitives, and give their meaning w.r.t. certain abstract (classes of) devices, terminal, disks, etc.
- Build software mapping the abstract devices to particular instances of that device.

11.1 Properties of the I/O interface

- Should processes block while performing I/O operations ?
  or should they continue executing and be notified when the operation completes ?

  Asynchronous ops: useful for controlling overlap - i.e., more parallelism - of computation and I/O ops
  Synchronous ops:
  - delay input ops until data arrives and output ops until data has been consumed
  - their advantage is that users can depend on data immediately ofter an input op., and change data immediately after an output op.
- Which format have data, and what is the size of transfer ?
  - Single-byte transfer (teletype terminals, e.g. the console)
  - block transfer (a block of many bytes)

11.2 Abstract operations

- *getc(), putc(,ch)*: deal with single-character transfer
  - *getc()* reads next character from keyboard
  - *putc(,ch)* display one character on terminal
- *read, wrtie*: deal with transfer to/from contiguous blocks of memory
  - *read* read a specialised number of characters
  - *write* displays several characters within one call
- *control* allows control of the device (driver): e.q., whether the system echoes each character as it is typed in the keyboard
- *seek* applies only to randomly accessible memory and the searches for a particular position
- *open, close* inform device (drivers) that data transfer will begin or has ended (applies to disk and file access)
- *init* initialize the device and device driver at system start up

11.3 Binding abstract operations to real devices

The system maps these high-level I/O operations to specific device drivers

- it hides the details of the hardware and the device drivers.
- E.g., that keyboard and display are independent
- it makes programs independent of the particular hardware configuration
- The high-level calls of these operations constitute the environment which the system presents to running programs - i.e.
  the programs only perceive the peripheral devices through these abstract calls
- The system also maps abstract names s.a. console to real devices
- Coded into the system is a description of each abstract devices:
  - e.g., the device driver routines which it uses, the address of the real device to which it corresponds
- When a new device is added to the system, or, e.g., the device addresses are modified, the system must be altered and recompiled
- However, since programs do not contain direct calls to these devices and their drivers, and no device addresses, programs don't need to be recompiled, as long as the abstract device descriptors do not change

11.4 Binding I/O calls to device drivers at run-time

- Routines like *read* in the compiled code should map abstract device descriptors, s.s., console, to device driver routines and real device addresses
- In Xinu, each abstract device is assigned an integer device descriptor, 0,... 8 at system configuration.
  E.g., console has the same device descriptor in all Xinu systems.
- After system configuration, device descriptors are bound into the system when it is compiled, and these are placed in a library.
  No recompilation needed unless the system changes (e.g, a new device is added)
- At run-time (after compilation) the program calls high-level routines s.a. *read(descriptor,,)*, and *putc(descriptor,ch)* having the device descriptor as argument.
- Device descriptor: an index into the device switch table which contains an entry for each value of the device descriptor
- Device switch table: *devtab[0...8]* maps 0...8 to a structure of *devsw* containing the device drivers for that abstract device, and device addresses.

11.4.1 device switch table

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

#define NULLPTR  (char *)0
/* device table declarations */
struct  devsw    {           /* deive table entry */
        int      dvnum;
        int      (*dvinit)();
        int      (*dvopen)();
        int      (*dvclose)();
        int      (*dvread)();
        int      (*dvwrite)();
        int      (*dvseek)();
        int      (*dvgetc)();
        int      (*dvputc)();
        int      (*dvcntl)();
        int      dvsr;
        int      dvivec;
        int      dvovec;
        int      (*dviint)();
        int      (*dvoint)();
        int      (*dvioblk)();
        char     *dvioblk;
        int      dvminor;
        };
```

```
extern  struct   devsw devtab[];  /* One entry per device */

/* Device name definitions */

#define CONSOLE  0           /* type tty       */
#define OTHER    1           /* type tty       */
#define RINGOIN  2           /* type dlc       */
#define RINGOOUT 3           /* type dlc       */
#define DISK0    4           /* type dsk       */
#define FILE1    5           /* type df        */
#define FILE2    6           /* type df        */
#define FILE3    7           /* type df        */
#define FILE4    8           /* type df        */

/* Control block sizes */

#define Ntty   2
#define Ndlc   2
#define Ndsk   1
#define Ndf    4

#define NDEVS  9
```

```
/* Declarations of I/O routines referenced */

extern  int    ttyinit();
extern  int    ionull();
extern  int    ttyread();
extern  int    ttywrite();
extern  int    ioerr();
extern  int    ttyctl();
extern  int    ttygetc();
extern  int    ttyputc();
extern  int    ttyiin();
extern  int    ttyoin();
extern  int    dlcinit();
extern  int    dlcread();
extern  int    dlcwrite();
extern  int    dlccntl();
extern  int    dlcputc();
extern  int    dlciin();
extern  int    dlcoin();
extern  int    dsinit();
extern  int    dsopen();
extern  int    dsread();
extern  int    dswrite();
extern  int    dsseek();
```

```
extern  int     dscntl();
extern  int     dsinter();
extern  int     lfinit();
extern  int     lfclose();
extern  int     lfread();
extern  int     lfwrite();
extern  int     lfseek();
extern  int     lfgetc();
extern  int     lfputc();

/* Configuration and Size Cnstants */

#define MEMMARK     /* define if memory marking used */
#define NNETS   1   /* number of Xinu ring networks  */

#define NPPROC   10 /*  (remove if there are zero)   */
#define NSEM     50 /* total number of semaphores    */
#define RTCLOCK     /* system has a real-time clock  */
                    /* label printed at startup      */
#define VERSION "6.1b (05/22/84)"
```

Each entry corresponds to a single device, containing:

- *dvnem* the corresponding entry into the interrupt dispatch table *intmap*
- addresses of the device driver routines for that device: *dvgetc, devputc, dvread, dvwrite, dvcontrl, dvseek, dvinit*
  These routines hold addresses of driver routines corresponding to high-level operations.
- device address and other informations, since more than one device can use the same device driver.
- device switch table also contains:
  - hardware device addresses *dvcsr*
  - interrupt vector addresses *dvivec*, *dvovec*
  - the interrupt routines for :
    input *dviint*
    output *dvoint*
  - buffer pointer *dvioblk*
  - an integer *dvminor* distinguishes among multiple copies of a device

11.5 Implementation of high-level I/O operations

There is a procedure call in the object code of a program for each of the abstract operations *getc, put, read* etc. - e.g.

read(descrp, buff, count)

descrp is the device descriptor: index into device switch table *devtab*
buff is the address of buffer into which the data read should be written
count is the number of charters to read

```
/* read.c - read */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------
 * read - read one or more bytes from a device
 *------------------------------------------------
 */
read(descrp, buff, count)
int  descrp, count;
char *buff;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvread)(devptr,buff,count) );
}
```

```
/* control.c - control */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------
 * control - control a device (e.g., set the mode)
 *------------------------------------------------------
 */
control(descrp, func, addr, addr2)
int descrp, func;
char *addr,*addr2;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvcntl)(devptr, func, addr, addr2) );
}
```

```
/* getc.c - getc */

#include <conf.h>
#include <kernel.h>
#include <io.h>


/*------------------------------------------------
 * getc - get one character from a device
 *------------------------------------------------
 */
getc(descrp)
int descrp;
{
    struct  devsw   *devptr;

    if (isbaddev(descrp) )
            reurn(SYSERR);
    devptr = &devtab[descrp];
    return( (*devptr->dvgetc)(devptr) );
}
```

```
/* init.c - init */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------
 * init - initialize a device
 *------------------------------------------------
 */
init(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvinit)(devptr) );
}
```

```
/* putc.c - putc */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------
 * init - write a single character to a device
 *------------------------------------------------
 */
putc(descrp, ch)
int descrp;
char ch;
{
        struct  devsw   *devptr;

        if (isbaddev   (descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvputc)(devptr,ch) );
}
```

```
/* seek.c seek */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------
 * seek -- position a device (very common case of control
 *------------------------------------------------------
 */
seek(descrp, pos)
int descrp;
long pos;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvseek)(devptr,pos) );
}
```

```
/* write.c - write */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------
 * write - write 1 or more bytes to a device
 *------------------------------------------------------
 */
write(descrp, buff, count)
        int descrp, count;
        char *buff;
{
        struct devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvwrite)(devptr,buff,count) );
}
```

11.6 Opening and closing devices

Disk devices require programs to start them up before a transfer operation, and to stop them before after the transfer completed.

- *open*
- *close*

```
/* colse.c - close */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------
 * close - close a device
 *------------------------------------------------
 */
close(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvclose)(devptr));
}
```

```
/* open.c - open */

#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------
 * open - open a connection to a device/file
 * (parms 2 & 3 are optinal)
 *------------------------------------------------
 */
open(descrp, nam, mode)
int     decrp;
char    *nam;
char    *mode;
{
        struct   devsw   *devptr;

        if ( isbaddev(descrp))
                 return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvopen)(devptr, nam, mode) );
}
```

11.7 Null and error entires in devtab

Some entries in the device table are not meaningful for a particular device. E.g., *open* and *close* do not apply to consoles. Calling there operations would result in a n error for console

How should these entries be filled in ?
By using:

- *ioerr* - returns SYSERR when called - This signifies an illegal operation
- *ionull* - returns OK - Signifies an unnecessary but, otherwise, innocuous operations.

```
/* ioerr.c - ioerr */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 * ioerr - return an error (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
ioerr()
{
        return(SYSERR);
}

/* ionull.c - ionull */

/*------------------------------------------------------------------------
 * ionull - do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
ionull()
{
        return(OK);
}
```

11.8 Initialization of the I/O system

Devtab is initialized at system configuration time, so it is completely filled in by the time the system is complied - See Comer pg. 154,155.

11.9 Interrupt vector intialization

Interrupt vector and the interrupt dispatch table *intmap* are initialied at run-time, using the information in *devtab*, by calling *init(0)*, ...*init(8)* - See chapter 13.