

Echtzeitbetriebssysteme

IO-System, Filesysteme

Abspeichern, Sichern auf Hintergrundspeicher von:

- **Programmen (BS-Kern, Dienstprogramme und Applikationen)**
- **Konfigurationsinformationen**
- **Daten (z.B. HTML-Seiten)**

Embedded Systems: Kaum Festplatten, sondern

- **RAM-Filesysteme (beim Starten aus persistentem Speicher geladen, beim Beenden auf persistenten Speicher geschrieben)**
- **EEPROM**
- **Flash („soldered“, CompactFlash, ...)**



Echtzeitbetriebssysteme

IO-System, Filesysteme

Organisationsstruktur (Filesystem, z.B. FAT, VFAT, NTFS, Linux Ext2, Ext3, TFFS) soll:

- schnellen Zugriff ermöglichen
- wenig Overhead (Speicher) für Verwaltungsinfo benötigen

Meist wird Cache verwendet

Problem:

- Daten temporär inkonsistent
- Kein zeitlicher Determinismus bei Zugriff

Sync-Mode:

- Keine Verwendung von Cache
- Explizit synchronisieren



Echtzeitbetriebssysteme

IO-System, Synchronisation

Zugriffe auf Geräte (z.B. Platte) benötigen Zeit

Zugriffszeit: Zeit zwischen Auftrag und Erfüllung des Auftrags

Erforderlich:

- **Mechanismen, um Prozess mit Gerätezugriff zu synchronisieren**

Prinzipiell:

- **Synchroner Zugriff (implizites Warten)**
- **Asynchroner Zugriff (explizites Warten)**



Echtzeitbetriebssysteme

IO-System, Synchronisation

Implizites Warten (synchroner Zugriff, *synchronous IO*)

Auf ein Gerät zugreifende Task wird solange “schlafen” gelegt (*pending*)

- bis das Gerät antwortet
- eine Fehlersituation (z.B. das Gerät antwortet innerhalb einer definierten Zeitspanne nicht) eingetreten ist
- Die Task ein Signal erhält



Echtzeitbetriebssysteme

Synchroner Zugriff auf Geräte:

Blockierende POSIX-Systemcalls und Signale

OS9: unterbricht I/O nicht wg. Signalempfang

Ein Beispiel...

```
do {
    ret = read ( fd, buf, bufsize );
    if ( ret <= 0 && errno != EINTR ){
        perror ( "read" );          /* Fehlerbehandlung */
        return ( ret );
    }
} while ( errno == EINTR );      /* Wiederaufsetzen bei
                                  Signalen */
```



Echtzeitbetriebssysteme

IO-System, Synchronisation

Explizites Warten (asynchroner Zugriff, *asynchronous IO*)

**Zugriff auf Gerät („Zugriffsanforderung, Zugriffsauftrag“)
kehrt sofort zurück**

- **Aufrufende Task kann weiterarbeiten**
- **Mitteilung der Auftragsausführung durch das BS an die Task per Abfrage (Polling), Ereignis (Signal) oder Callback-Funktion**
- **Task kann dann das Ergebnis abholen**



Echtzeitbetriebssysteme

IO-System, Synchronisation

Explizites Warten (asynchroner Zugriff, *asynchronous IO*)

Probleme:

- **Mehrere Aufträge können gleichzeitig vorliegen; mehrere Dienste müssen sich auf denselben Auftrag beziehen können.**
- **Ggf. wird eine Dienstkette „Auftrag-Status ermitteln“-“Ergebnis holen“ nicht bis zum Ende einer Applikation beendet**

Zustandsverwaltung im Betriebssystem erforderlich



Echtzeitbetriebssysteme

IO-System, Synchronisation

Explizites Warten (asynchroner Zugriff, *asynchronous IO*)

3 Varianten:

1. Asynchroner Zugriff, Win32 und POSIX
2. Asynchroner Zugriff mit Threads und synchronen Diensten
3. Asynchroner Zugriff mit nichtblockierenden Diensten



Echtzeitbetriebssysteme

IO-System asynchroner Zugriff mit nicht blockierenden Diensten (1)

- Leseauftrag
- Warten auf Ergebnis
- Ergebnis holen
 - Polling
 - Signal/Warten(Schlafen)
 - Signal/Callback

```
char *OverlappedAccess()  
{  
    char buf[100];  
    DWORD dwBytesRead;  
    DWORD dwCurBytesRead;  
    OVERLAPPED ol;  
  
    ol.Offset = 10;           //read Filestartoffset: 10th byte from start  
    ol.OffsetHigh = 0;  
    ol.hEvent = NULL;  
  
    HANDLE hFile = CreateFile("overlap.test", GENERIC_READ,  
        FILE_SHARE_WRITE, 0, OPEN_EXISTING,  
        FILE_FLAG_OVERLAPPED, 0);  
  
    ReadFile(hFile, buf, 100, &dwBytesRead, &ol);  
    // assumed ReadFile returned FALSE and ERROR_IO_PENDING  
    // perform other tasks in this thread  
    / ...  
  
    // Synchronise with file I/O  
    WaitForSingleObject(hFile, INFINITE);  
  
    GetOverlappedResult(hFile, &ol, &dwCurBytesRead, TRUE);  
    CloseHandle(hFile);  
  
    return(result_buffer);  
}
```



Echtzeitbetriebssysteme

IO-System

Asynchroner Zugriff mit nicht blockierenden Diensten, POSIX 1003.1b (1)

- Erhöhung der Performance (Audio, Video, Netzwerk, Datenbanken...)
- Besseres Antwortverhalten

Zentrale Struktur zur Spezifikation eines IO-Zugriffs

```
struct aiocb {
    int aio_fildes;          /* File descriptor          */
    int aio_lio_opcode;     /* Auszuführende Operation */
    int aio_reqprio;       /* Priorität der Anfrage   */
    volatile void *aio_buf; /* Adresse des Buffers     */
    size_t aio_nbytes;     /* Länge des Transfers     */
    struct sigevent aio_sigevent; /* Signal (Nummer und Wert)*/
    ...}

```



Echtzeitbetriebssysteme

IO-System

Asynchroner Zugriff mit nicht blockierenden Diensten, POSIX 1003.1b (1)

```
int aio_read(struct aiocb *aiocbp);
```

Absetzen eines Leseauftrags

```
int aio_write(struct aiocb *aiocbp)
```

Absetzen eines Schreibauftrags

```
int aio_fsync(int operation, struct aiocb *aiocbp);
```

Synchronisiert alle ausstehenden IO-Operationen im zu aiocbp gehörenden Filedeskriptor (z.B. Cache im Treiber mit Platte in Übereinstimmung bringen)

```
int aio_suspend(const struct aiocb *const list[],  
int nent, const struct timespec *timeout)
```

Suspendiert den aufrufenden Thread, bis mindestens eine Operation aus list beendet wird, ein Signal gesendet wird oder timeout abgelaufen ist.



Echtzeitbetriebssysteme

IO-System

Asynchroner Zugriff mit nicht blockierenden Diensten, POSIX 1003.1b (1)

```
int lio_listio(int mode, struct aiocb *const list[], int nent,  
              struct sigevent *sig)
```

startet eine Menge von IO Anforderungen gleichzeitig

```
int aio_cancel(int fd, struct aiocb *aiocbp);
```

löscht die IO-Anforderung aiocbp

```
ssize_t aio_return(struct aiocb *aiocbp)
```

Liefert den Ergebnisstatus der beendeten IO-Operation

```
int aio_error(const struct aiocb *aiocbp);
```

Abfrage des aktuellen Zustands von IO-Anforderung aiocbp



Echtzeitbetriebssysteme

IO-System

Asynchroner Zugriff mit nicht blockierenden Diensten, POSIX 1003.1b (1)

Beispiel

```
if ((fd = open (exFile, O_CREAT | O_TRUNC | O_RDWR, 0666)) == ERROR {
    printf ("aioExample: cannot open %s errno 0x%x\n", exFile, errno);
    return (ERROR);
}

/* initialize read and write aiocbs */
aiocb_read.aio_fildes = fd;
aiocb_read.aio_buf = buffer;
aiocb_read.aio_offset = 0; // offset to buffer start
aiocb_read.aio_nbytes = BUFFER_SIZE;
aiocb_read.aio_reqprio = 0;
aiocb_write.aio_fildes = fd;
aiocb_write.aio_buf = test_string;
aiocb_write.aio_nbytes = strlen (test_string);
aiocb_write.aio_offset = 0; // offset to buffer start
aiocb_write.aio_reqprio = 0;
```



Echtzeitbetriebssysteme

IO-System

Asynchroner Zugriff mit nicht blockierenden Diensten, POSIX 1003.1b (1)

Beispiel

```
/* initiate the read */
if (aio_read (&aiocb_read) == -1)
    printf ("aioExample: aio_read failed\n");

/* write to pipe - the read should be able to complete */
if (aio_write (&aiocb_write) == -1)
    printf ("aioExample: aio_write failed\n");

/* wait til both read and write are complete */
while ((aio_error (&aiocb_read) == EINPROGRESS) ||
        (aio_error (&aiocb_write) == EINPROGRESS))
    taskDelay (1);

/* print out what was read */
printf ("aioExample: message = %s\n", buffer);
```



Echtzeitbetriebssysteme

IO-System Asynchroner Zugriff mit nicht blockierenden Diensten OS9 (1)

- OS9 sendet Signal READ_DATA_SIG wenn Lesedaten auf Pfad <pathid> bereit

```
time = 100;                // max. 100 TICS Timeout warten
_os_sigmask(1);           // Hier bis sleep() Signalempfang sperren ?!
_os_ss_sendsig(pathid, READ_DATA_SIG); // Auftrag Signal schicken!
.....
_os_sleep(&time, &sig);    // Hier: Signalempfang automatisch entsperret
                        //          Warten auf signal oder max time
_os_ss_relea(pathid);     // Auftrag immer loeschen
if(sig == READ_DATA_SIG) { // Daten da
    _os_gs_ready (pathid, &size); // wieviel ?? (size aufpassen!)
    myerr= _os_read(pathid,buffer,&size);
}
```



Echtzeitbetriebssysteme

IO-System

asynchroner Zugriff

mit Threads und

blockierendem Aufruf

(2)

- Thread erzeugen
- Dort blockierend zugreifen
- Thread-Ende abwarten

Was kann man in diesem

Beispiel

besser machen???

```
static char buf[100]; // global or as argument to pthread_create
```

```
void *AsyncThread( void *fp )
```

```
{  
    fread( buf, sizeof(buf), 1, (FILE *)fp );  
    pthread_exit( NULL );  
}
```

```
main( int argc, char **argv )
```

```
{  
    pthread_t MyThread;  
    FILE *fp;  
  
    fp = fopen( "overlap.test", "r" );  
    if( fp==NULL ) {  
        perror( "overlap.test" );  
        return;  
    }  
    if( pthread_create( &MyThread, NULL, AsyncThread, fp )!= 0 ) {  
        fprintf(stderr,"creation of thread failed\n");  
        exit( -1 );  
    }  
  
    // perform other tasks  
    // ...  
  
    // Synchronise with file I/O  
    pthread_join( AsyncThread, NULL );  
}
```



Echtzeitbetriebssysteme

IO-System

Nicht blockierender Aufruf (*non blocking mode*) (3)

Verwendet beim gleichzeitigen Warten auf mehrere Kanäle

Lesen nach vorherigem Warten auf Statusänderung am Kanal
(`select` bzw. `poll`)

Gerät im *non blocking mode* öffnen

```
int fd;

fd = open( "/dev/modem", O_RDWR | O_NONBLOCK );
if( fd < 0 ) {
    perror( "/dev/modem" );
    return( -1 );
}
```

- Lesende/schreibende Zugriffe blockieren dann nicht (read liefert 0, wenn keine zu lesenden Daten vorhanden)
- Synchronisation mit `FD_SETs` und `select`



Echtzeitbetriebssysteme

IO-System (3)

Nicht blockierender Aufruf

**Nicht blockierender Lesezugriff
wie blockierender, nur**

- **Liefert Ergebnis sofort, wenn vorhanden**
- **Liefert andernfalls Information „nichts zu lesen vorhanden“**
 - `retval == 0` timeout
 - **Hier: `retval == 1` data da**
 - `retval < 0` error in `errno`

```
int main(void)
{
    fd_set rfd;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input.
    FD_ZERO(&rfd);
    FD_SET(0, &rfd);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfd, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfd) will be true. */
    else
        printf("No data within five seconds.\n");

    exit(0);
}
```



Echtzeitbetriebssysteme

IO-System OS9 (3) Nicht blockierende Leseaufrufe

- Vor Leseaufruf teste ob <size> Daten lesebereit, dann maximal <size> Daten nicht-blockierend lesen

```
myerr = _os_gs_ready (pathid, &size);

if(myerr != EOS_NOTRDY) // Daten da
{
    if(size > 1000) size = 1000; /* max. 1000Byte wg. Buffergroesse*/

    myerr=_os_read(pathid,buffer,&size);
}
```



Echtzeitbetriebssysteme

Bibliotheken (*Libraries*)

Systemcalls meist durch Libraries gekapselt

Interface zu Applikationen und zu Dienstprogrammen

Typen von Libraries:

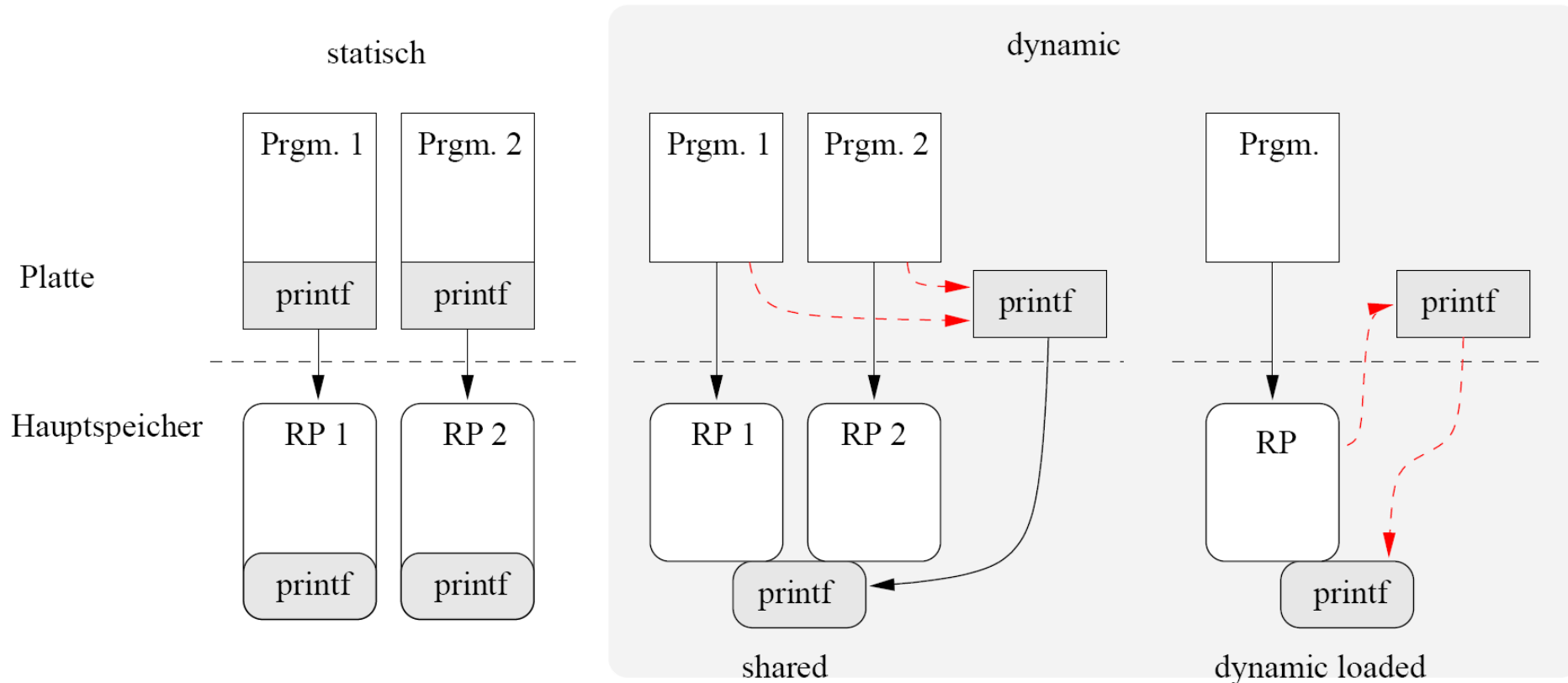
- **Statische (*static libraries*)**
 - zum Programm beim Linken hinzugebunden
- **Dynamische (*shared libraries*)**
 - Binden erfolgt zur Laufzeit, entweder
 - Beim Programmstart
 - Beim Aufruf der Libraryfunktion



Echtzeitbetriebssysteme

Bibliotheken (*Libraries*)

- **Statische Libs, Dynamische Libs, Laden zur Laufzeit**



Echtzeitbetriebssysteme

Bibliotheken (*Libraries*)

Shared vs. Static

Vorteile:

- kleinere Programme (Library-Code ist kein Programmbestandteil)
- Programme teilen sich “shared libs”, (Code-Segment gemeinsam, Data-Segment je Programm); spart Hauptspeicherplatz, weil derselbe, mehrfach genutzte Code nur einmal im Speicher ist

Nachteile:

- Versionsverwaltung: Zur Programmausführung sind exakt die zur Compilierung verwendeten Versionen der Shared Libs erforderlich. Manchmal (bei sich dynamisch ändernden Systemen) problematisch



Echtzeitbetriebssysteme

Bibliotheken (*Libraries*)

Dynamisches Laden durch die Applikation zur Laufzeit

- In Spezialfällen (welche kennen wir???)
- Verbesserte Modularität: „Attached Procedures“ in Datenbanken, (Mathematik-) Werkzeuge zur Anbindung effizienter Funktionen oder spezieller Interfaces



Echtzeitbetriebssysteme

Bibliotheken (*Libraries*)

Dynamisches Laden durch die Applikation zur Laufzeit

```
int main(int argc, char **argv){
void *handle;
int (*function)(int);
char *error;

handle = dlopen ("myshlib.so", RTLD_LAZY);
if( (error=dlerror()) ) {
fprintf(stderr,"%s\n", error );
return( -1 );
}
function = dlsym(handle, "DoubleMinusOne");
printf ("%d\n", function(5) );
dlclose(handle);
}
```

