

SYSTÈME D'EXPLOITATION I

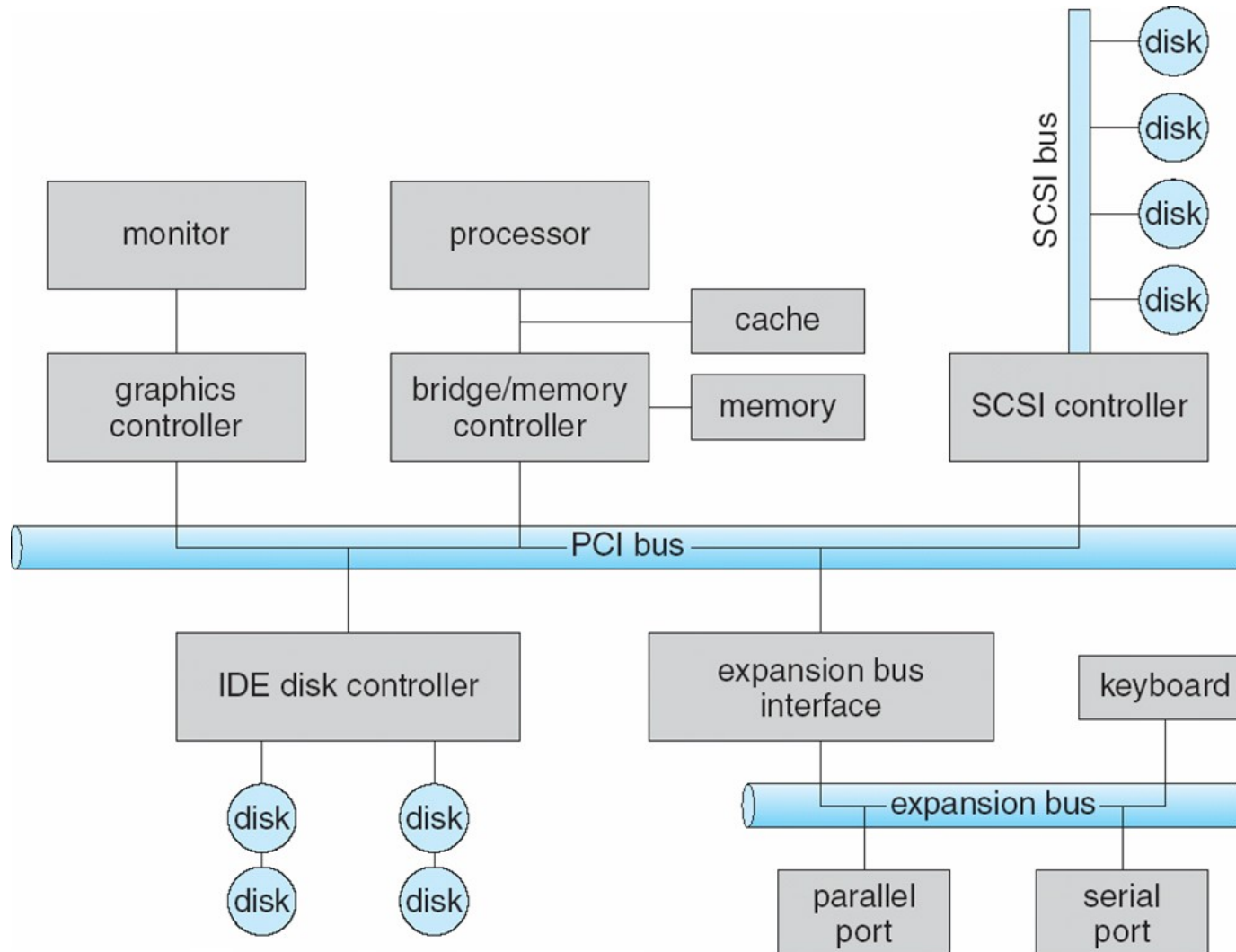
SIF-1015

Contenu du cours 4

- Entrées/Sorties
 - Concepts
 - Appels système
 - Implémentation générale des E/S
 - Implémentation détaillée des E/S
 - LECTURES:
 - Chapitre 3 (Matthew)
 - Chapitre 7 (Card)
 - Chapitre 11 (CSAPP, Stalling)
 - Annexe B (Mitchell)

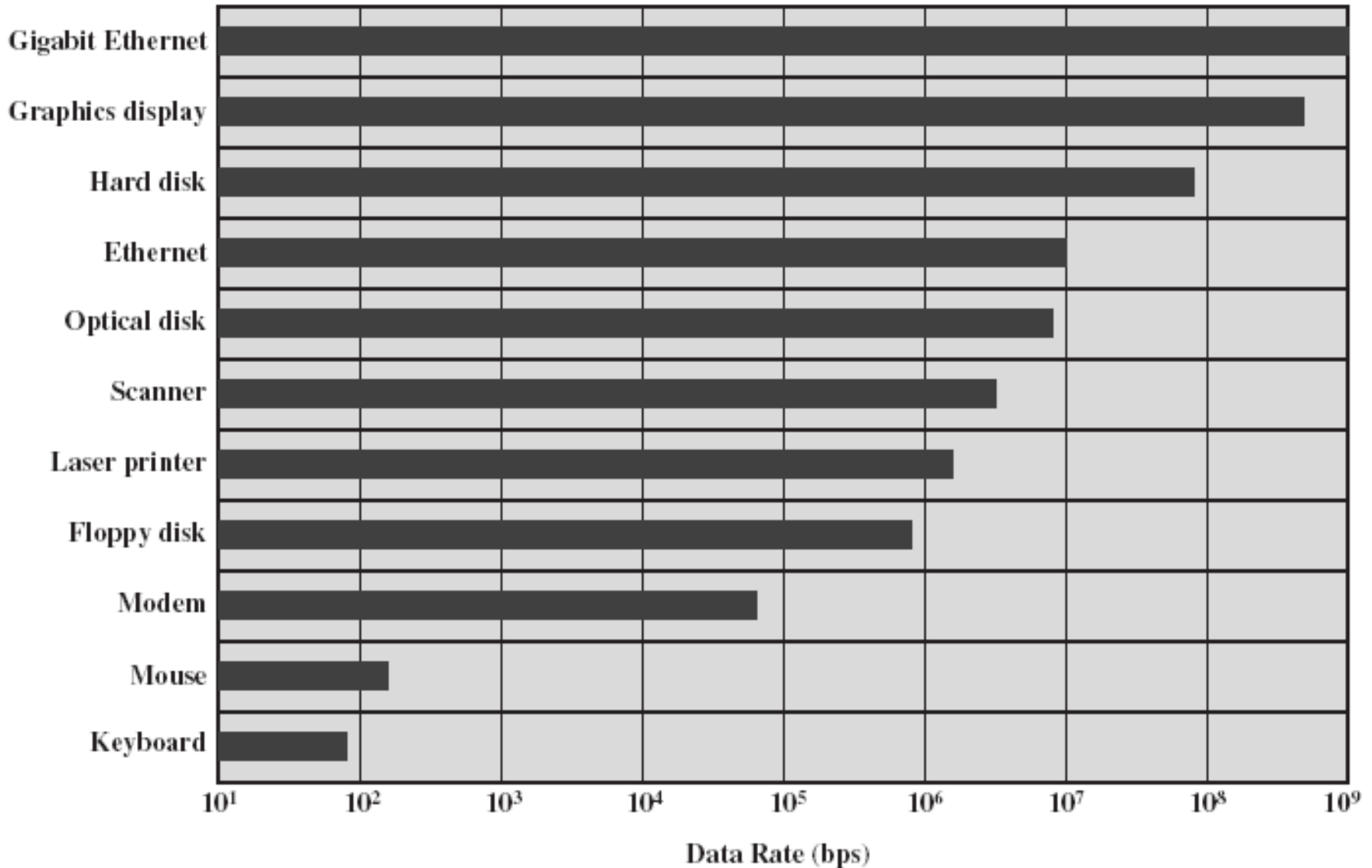
Concepts

- Structure générale d'un PC typique



Concepts

- Dispositifs d'E/S



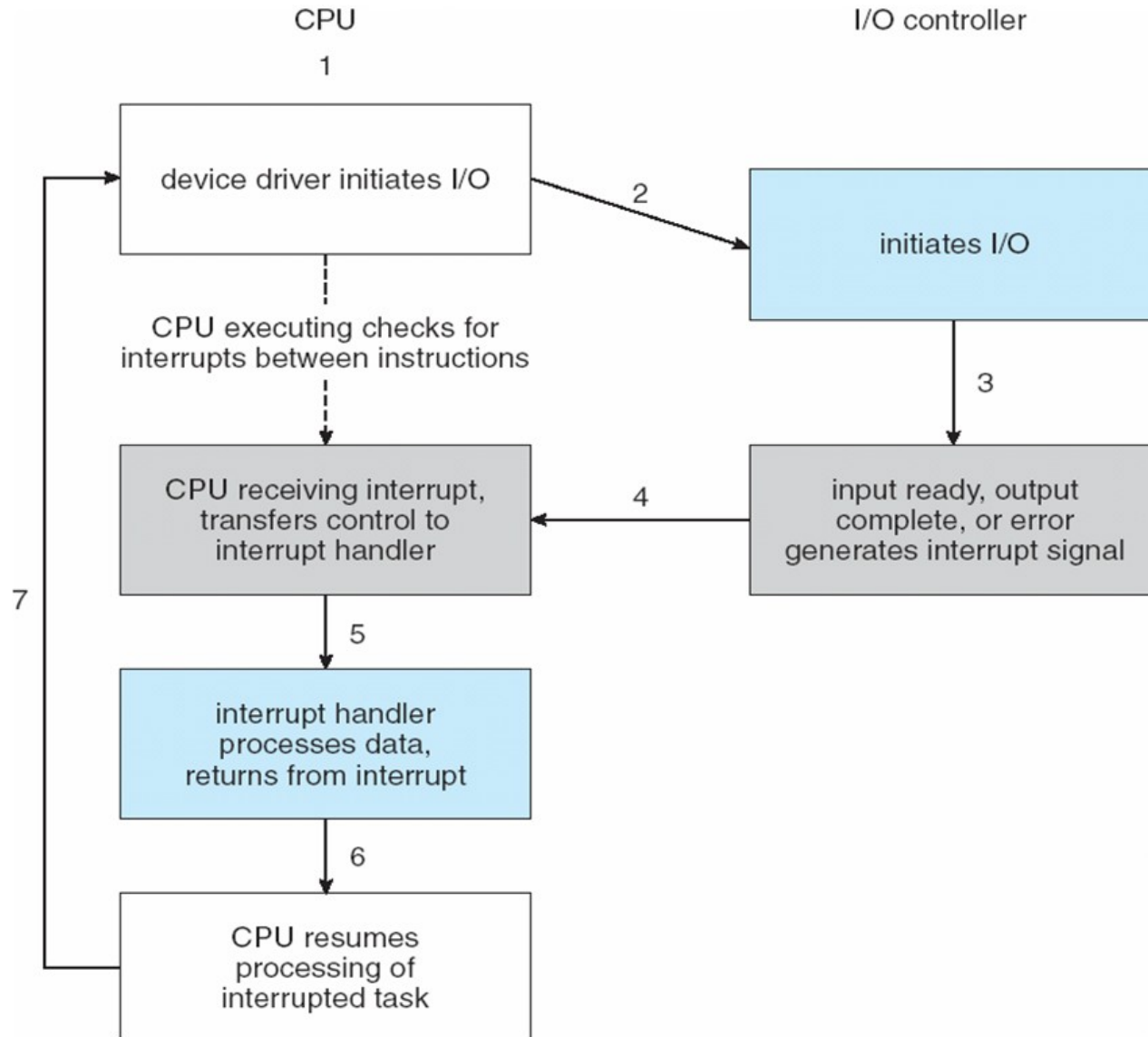
Concepts

- Architecture matériel (Ports d'E/S typiques)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

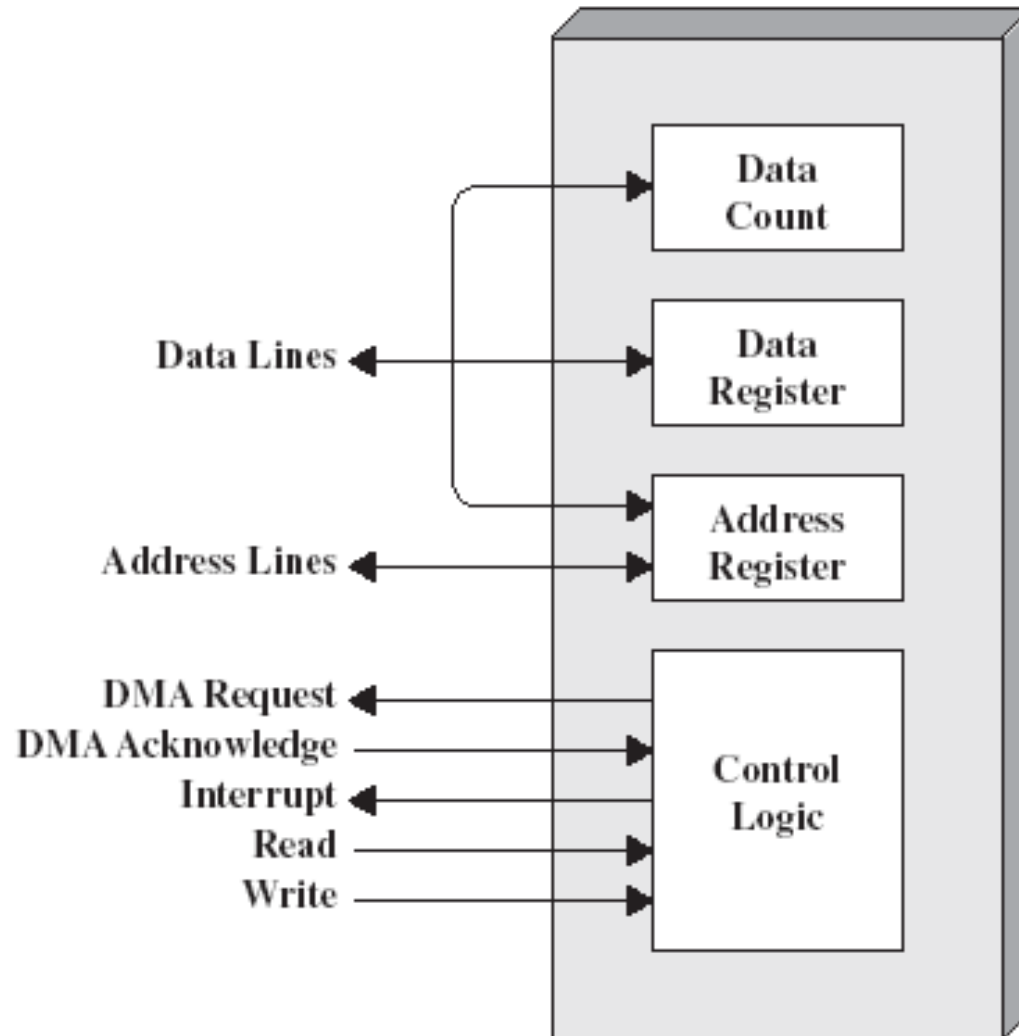
Concepts

- Architecture matériel (Interrogation des dispositifs)



Concepts

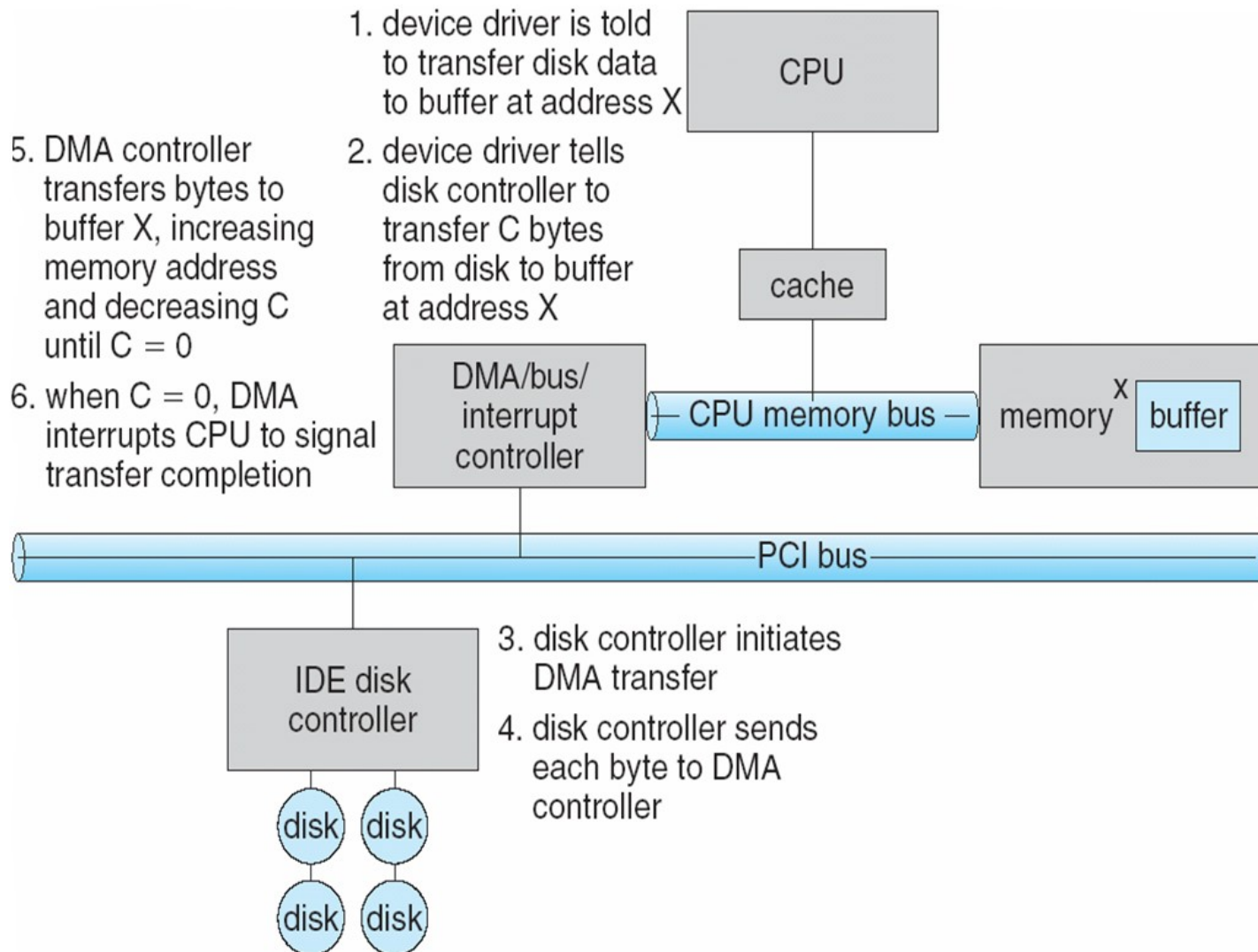
- Architecture matériel
 - Contrôleur de périphériques (ex: DMAC)



Concepts

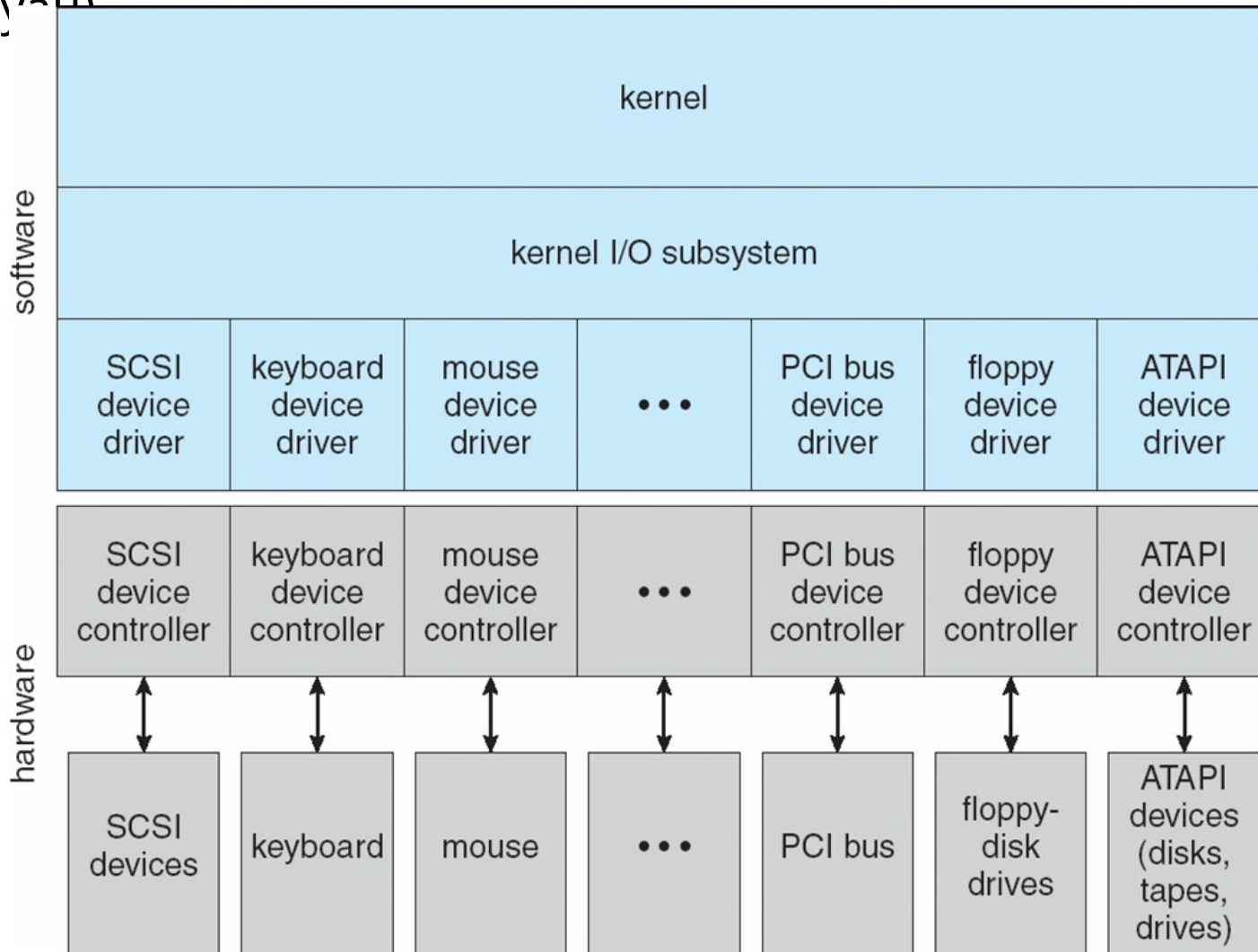
- Architecture matériel

- Transfert de données en mode DMA (6 étapes)



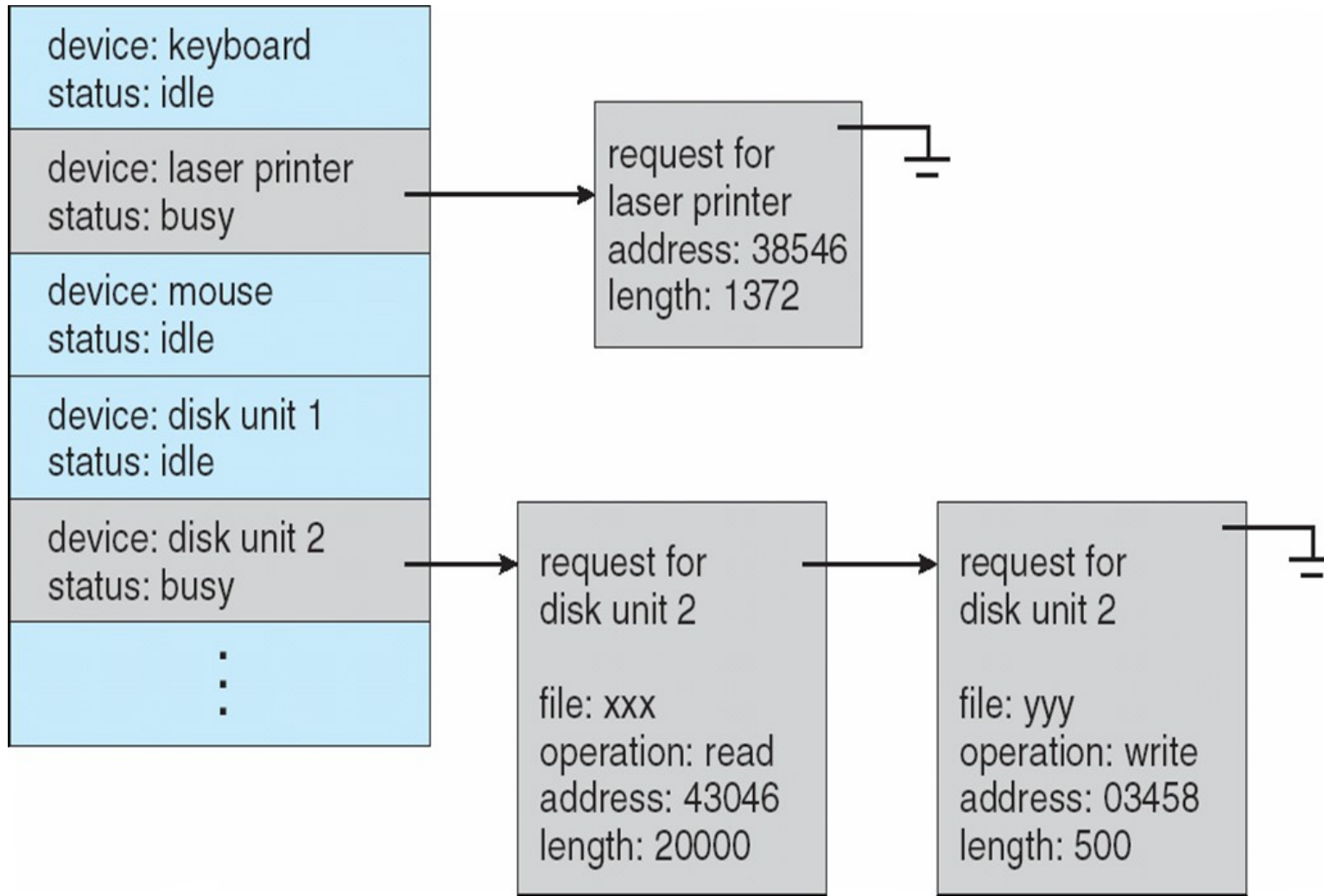
Concepts

- Architecture matériel
 - Modèle d'organisation de E/S (Structure des E/S au niveau du noyau)



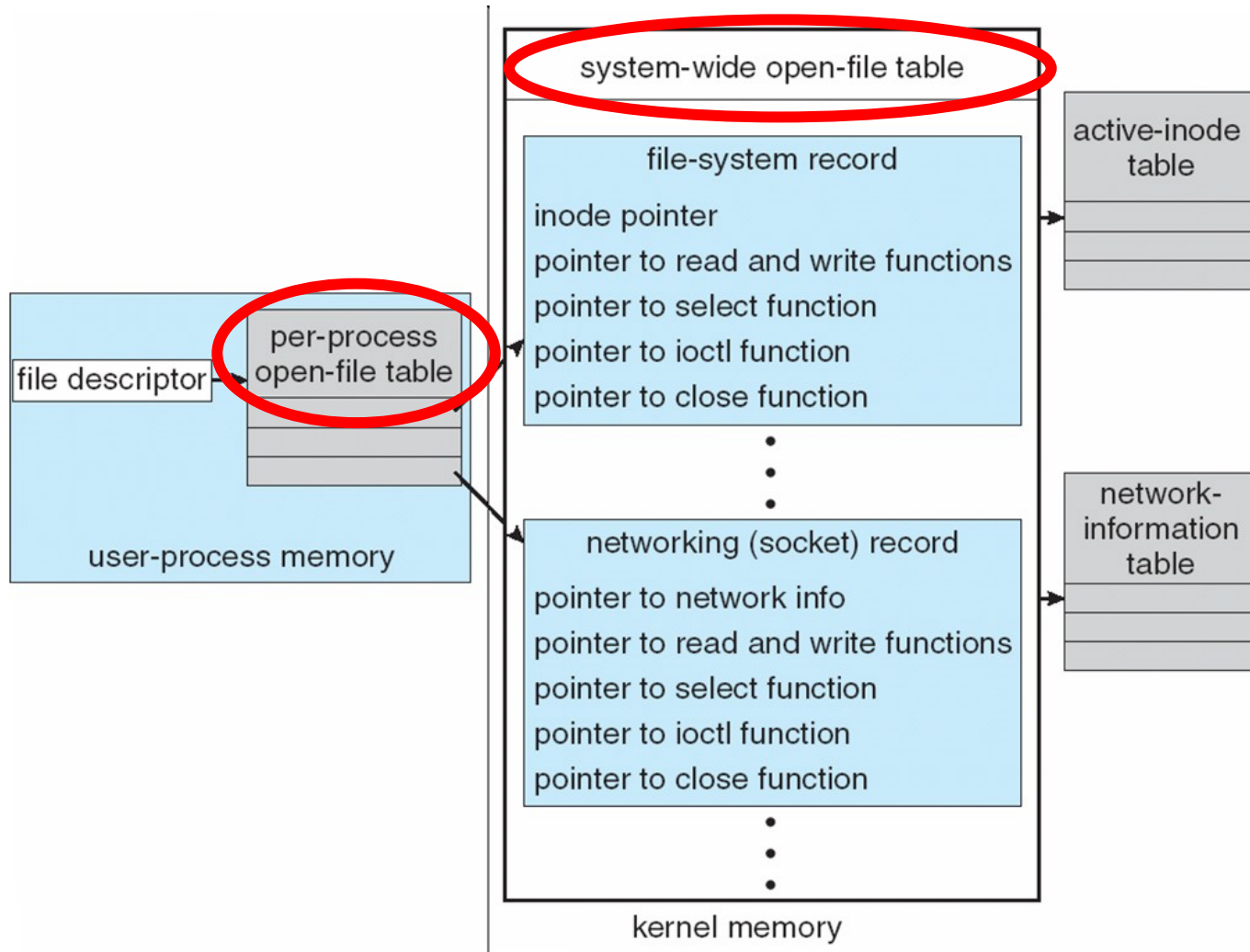
Concepts

- Architecture matériel
 - Modèle d'organisation de E/S (Table d'état des dispositifs au niveau du noyau)



Concepts

- Architecture matériel
 - Modèle d'organisation de E/S (Structures des E/S du noyau UNIX)



Concepts

- Architecture matériel
 - Modèle d'organisation des E/S
 - Les requêtes d'E/S impliquant soient un flot d'octets ou d'enregistrements (blocs) suivent un mode de communication basé sur un modèle en couches:
 - E/S logique: Ce module gère un ensemble de fonctions d'E/S génériques permettant ainsi d'interfacer les dispositifs d'E/S physiques en termes d'identificateurs de dispositifs d'E/S, et de fonctions génériques (***open()***, ***close()***, ***read()***, ***write()***).
 - E/S matérielle: À ce niveau, le OS achemine des instructions d'E/S au contrôleur d'E/S qui lui achemine les données souvent contenues dans des tampons en RAM vers le dispositif d'E/S approprié.
 - Séquençage et contrôle: Ce niveau permet de gérer la mise en file d'attente, le séquençage et le contrôle (état des E/S) des opérations d'E/S.

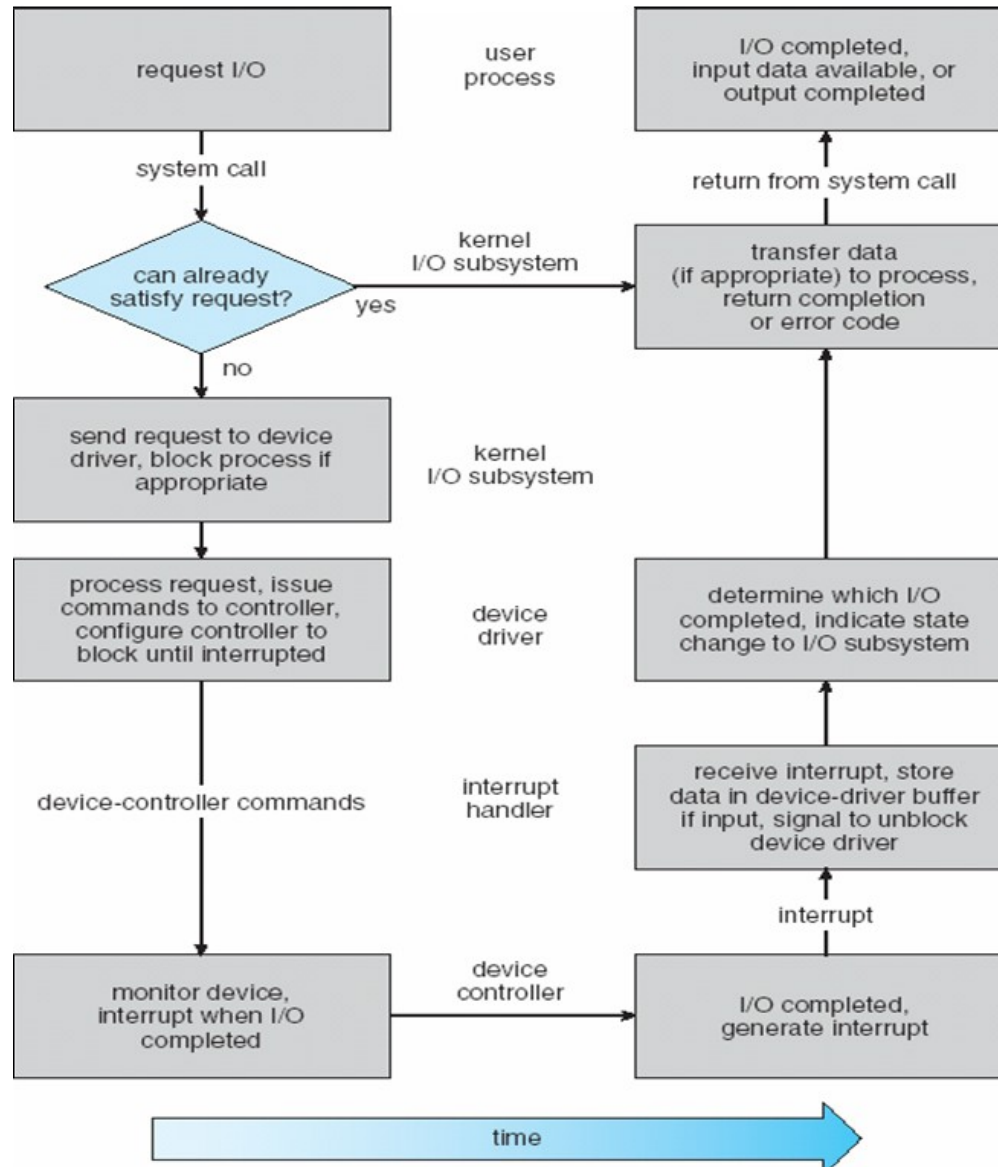
Concepts

- Architecture matériel (Modèle d'organisation des E/S:
Caractéristiques des dispositifs d'E/S)

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Concepts

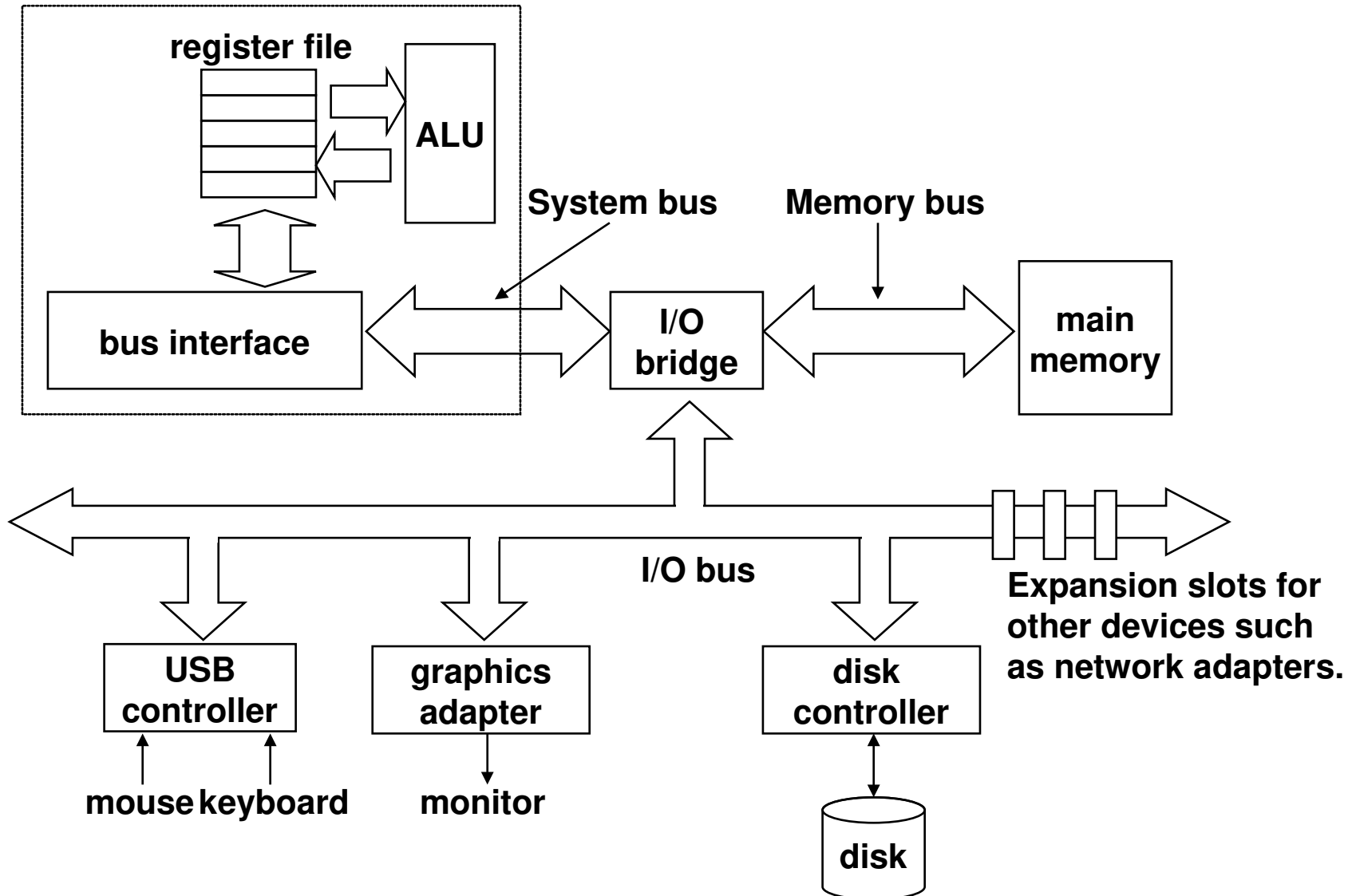
- Architecture matériel (Cycle de vie d'une requête d'E/S)



Concepts

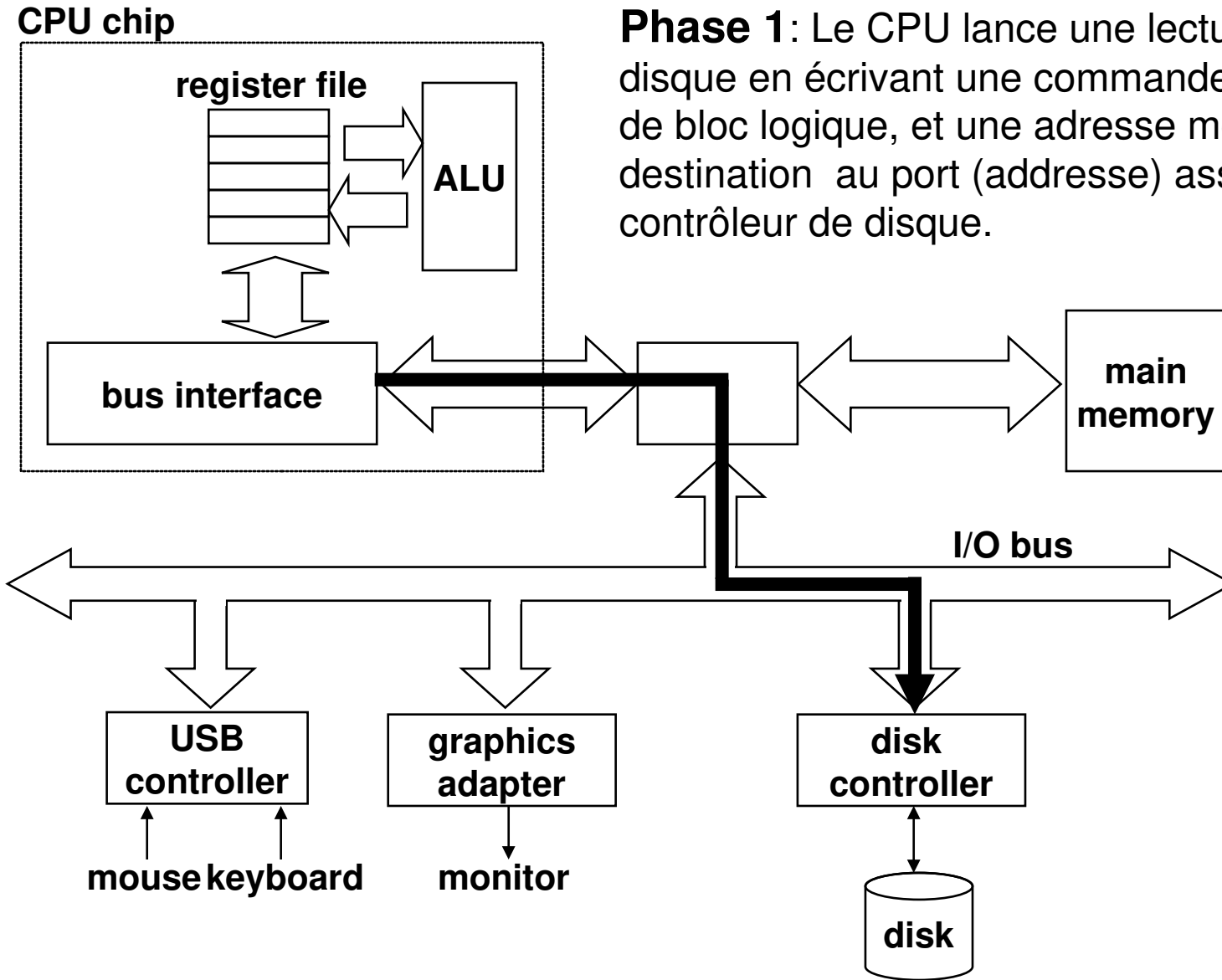
- Architecture matériel (Système typique)

Circuit CPU



Concepts

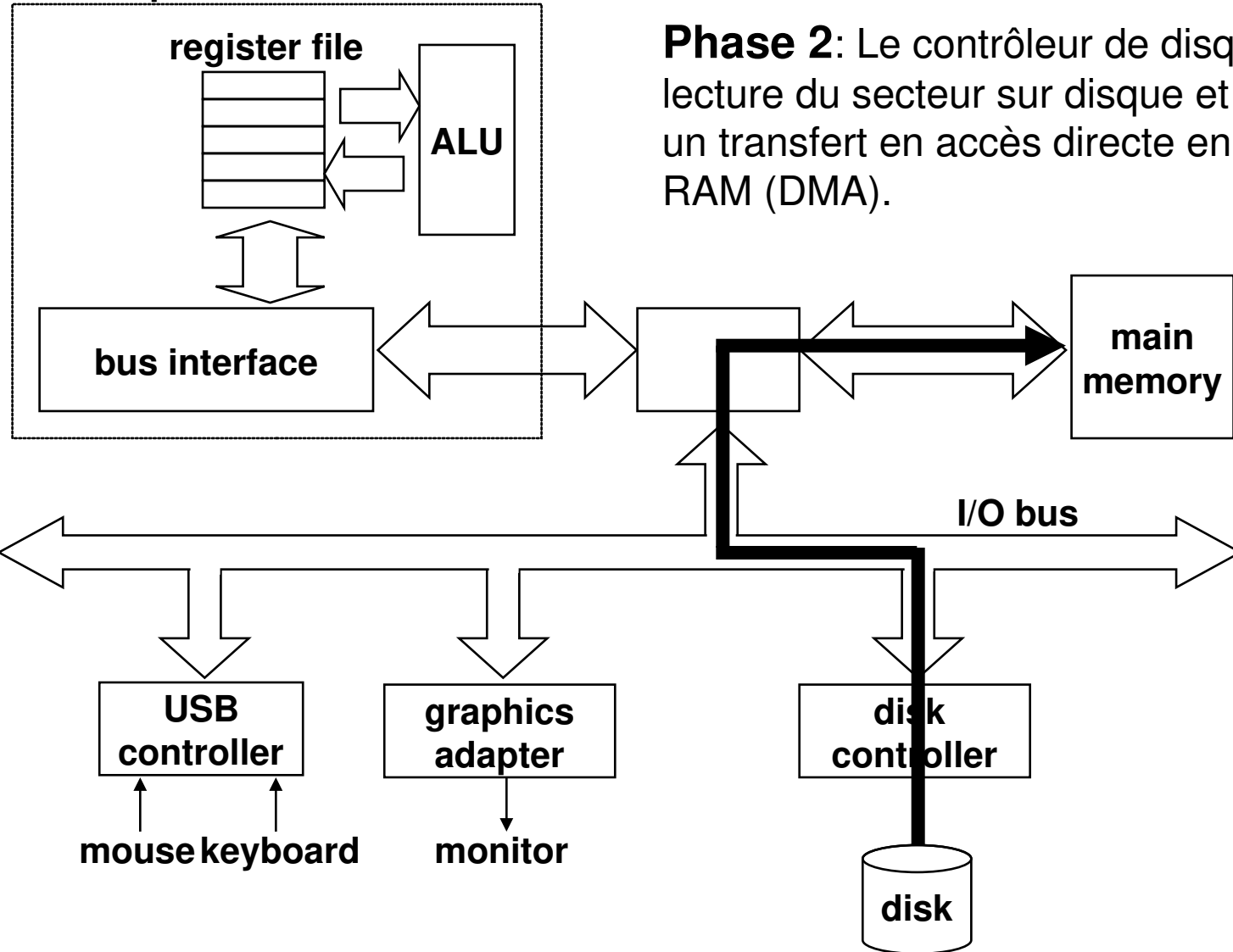
- Architecture matériel (Lecture d'un secteur sur disque)



Concepts

- Architecture matériel (Lecture d'un secteur sur disque)

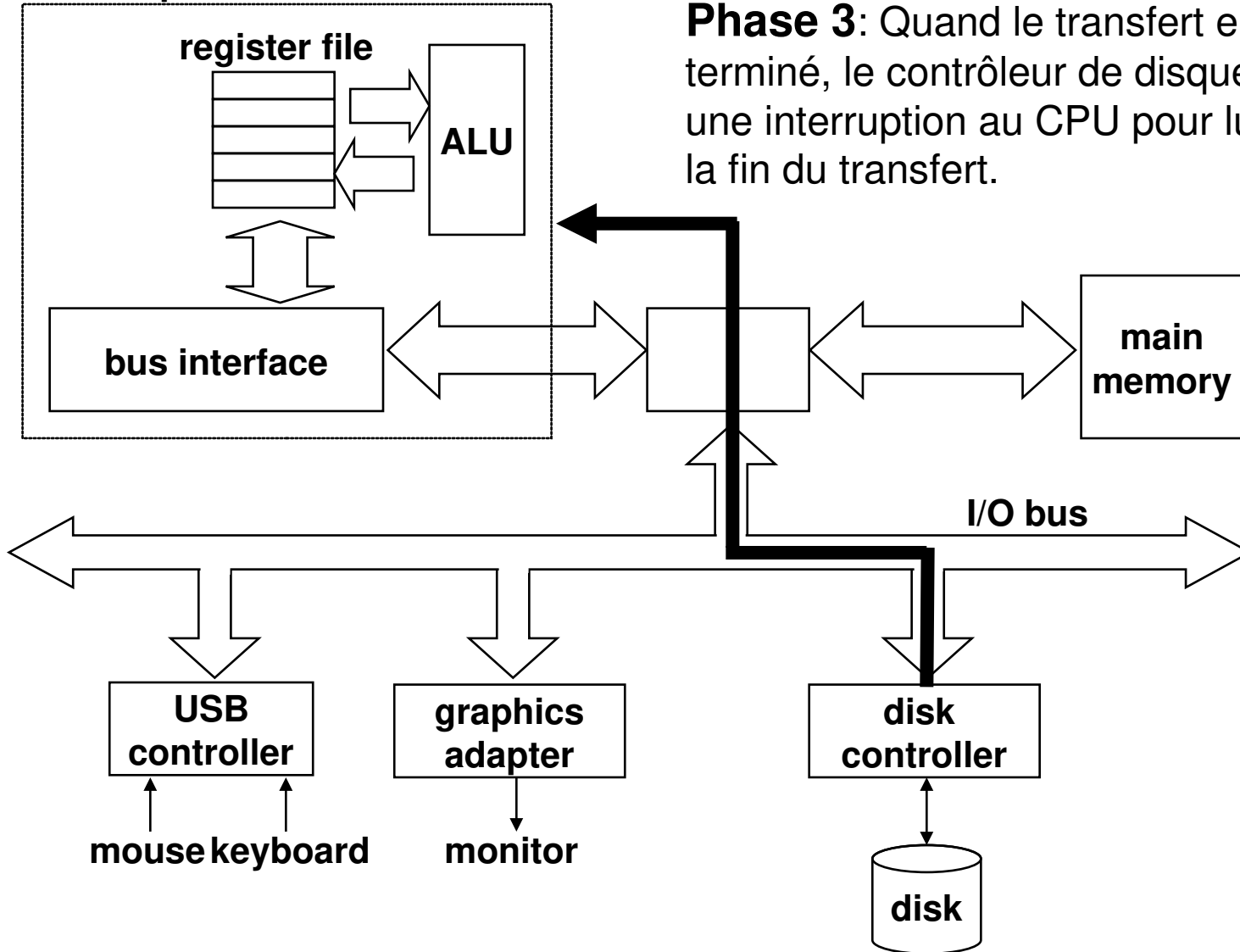
CPU chip



Concepts

- Architecture matériel (Lecture d'un secteur sur disque)

CPU chip



Concepts

- Concepts de fichiers (fichier UNIX)
 - Un fichier UNIX est une séquence de n octets:
 $B_0, B_1, \dots, B_k, \dots, B_{n-1}$
 - Tout les dispositifs d'I/O sont représentés comme des fichiers:
 - /dev/sda2 (partition de disque /usr)
 - /dev/tty2 (terminal)
 - Même le noyau est représenté comme un fichier:
 - /dev/kmem (image mémoire du noyau)
 - /proc (structure de données du noyau)
 - Le noyau identifie chaque fichier par un descripteur numérique (nombre entier):
 - 0: standard input
 - 1: standard output
 - 2: standard error

Concepts

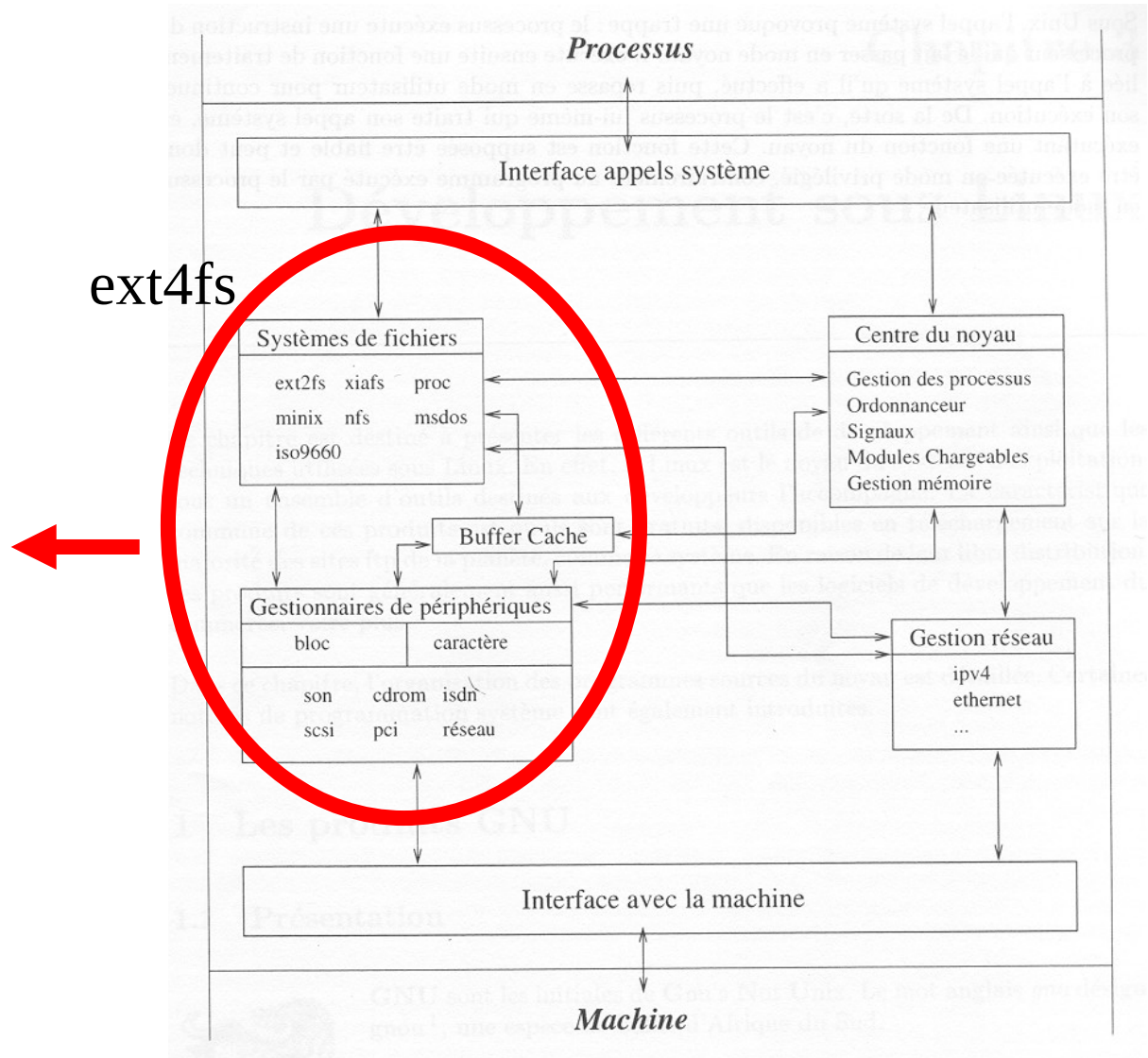
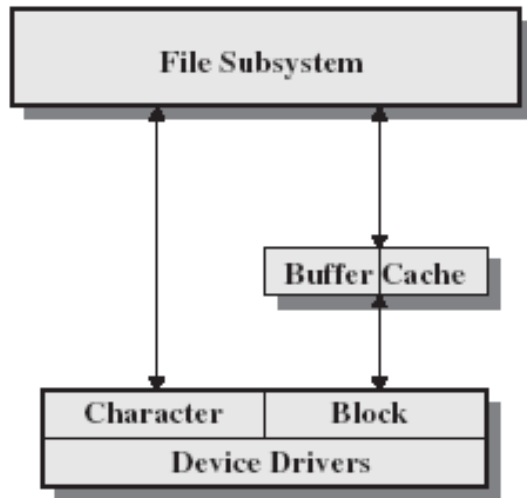
- Concepts de fichiers (Type de fichiers UNIX)
 - Fichier régulier
 - Fichier texte ou binaire.
 - Unix ne fait pas de différence!
 - Fichier répertoire
 - Fichier qui contient les noms et les localisations d'autres fichiers
 - Fichier spécial de type caractère
 - Pour certains types de dispositifs en mode flot (ex: terminaux)
 - Fichier spécial bloc
 - Typiquement pour les transferts aux disques
 - FIFO (pipes nommées)
 - Type de fichier utilisé pour la communication interprocessus (IPC)
 - Socket
 - Type de fichier utilisé pour la communication réseau

Concepts

- Concepts de fichiers
 - L'accès aux périphériques est effectué via des fichiers spéciaux (**device file**). Un fichier spécial est similaire au fichier régulier (ex: fichiers sources **.c**)
 - Chaque fichier spécial correspond à un pilote de périphérique (**driver**) dont le code est intégré au noyau
 - Lorsqu'un fichier spécial est ouvert par un processus, ses requêtes d'E/S sont transmises au pilote du périphérique correspondant
 - Lorsqu'un processus utilise les appels système **read()** et **write()**, le pilote effectue les E/S physiques sur le périphérique en dialoguant avec son contrôleur
 - Types de fichiers spéciaux:
 - **Mode bloc**: Correspondent à des périphériques structurés en blocs (disques, disquettes, CD-ROM), accessibles en fournissant un numéro de bloc à lire ou à écrire. Les E/S sont effectuées par les fonctions du **buffer cache**
 - **Mode caractère**: Correspondent à des périphériques sur lesquels on peut lire et écrire les données octet par octet de façon séquentielle (imprimantes, terminaux)

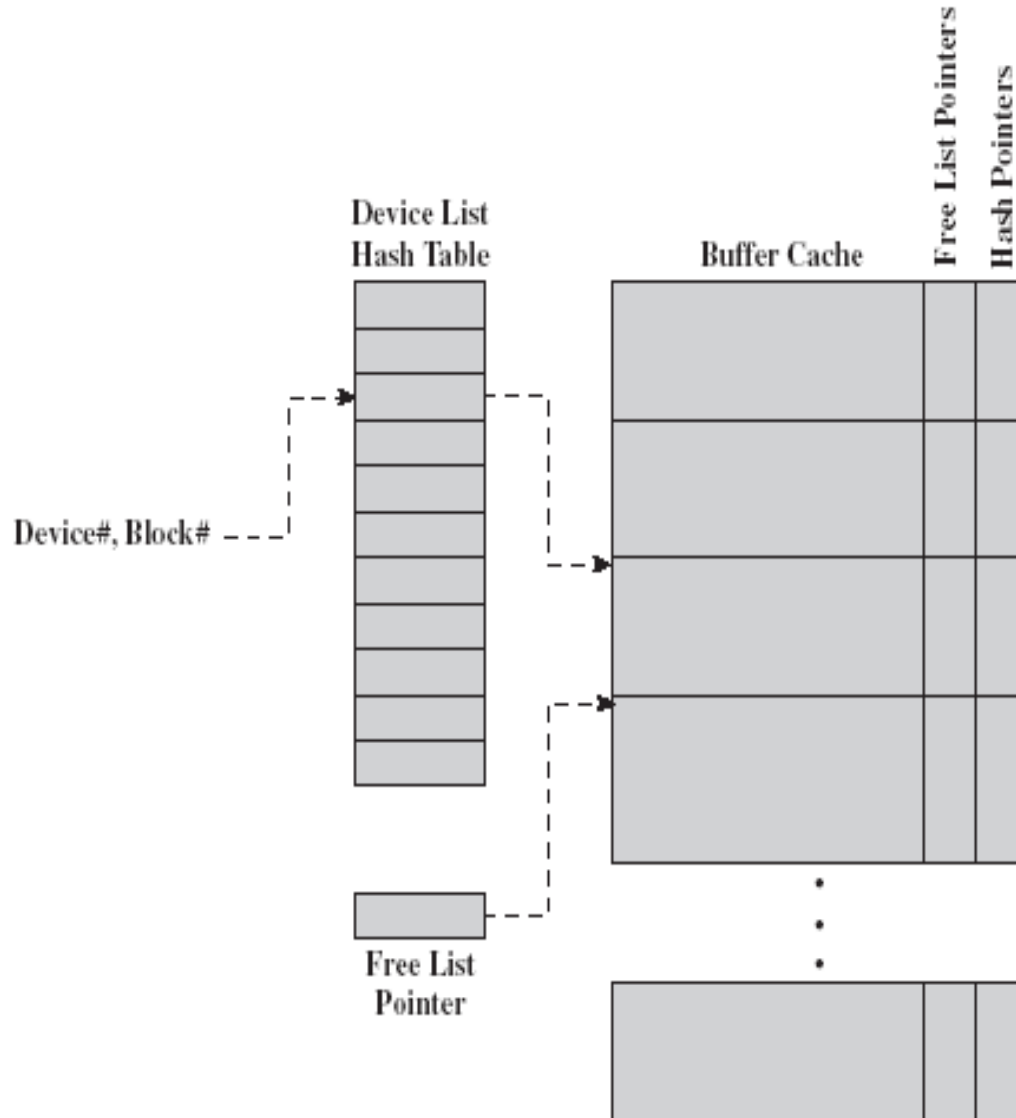
Concepts

- Concepts de fichiers




Concepts

- Concepts de fichiers (Organisation du BUFFER CACHE)



Concepts

- Concepts de fichiers
 - Chaque fichier spécial est caractérisé par trois attributs:
 - Type: bloc ou caractère
 - Numéro majeur: Identificateur du pilote de périphérique
 - Numéro mineur: Identificateur du périphérique spécifique à l'intérieur d'un groupe de périphériques géré par le même pilote et qui partage donc le même numéro majeur
 - Les fichiers spéciaux sont situés dans le répertoire **/dev** pouvant contenir:



nom	type	major	minor	description
/dev/fd0	block	2	0	lecteur de disquettes
/dev/hda	block	3	0	premier lecteur IDE
/dev/hda2	block	3	2	2e partition primaire du 1er disque IDE
/dev/hdb	block	3	64	2e disque IDE
/dev/hdb3	block	3	67	2e partition primaire du 2e disque IDE
/dev/tty0	char	3	0	terminal
/dev/console	char	5	1	console
/dev/tp1	char	6	1	imprimante sur port parallèle
/dev/ttySO	char	4	64	1er port série
/dev/rtc	char	10	135	horloge temps-réel
/dev/null	char	1	3	périphérique null (trou noir)

Appels système

- Création d'un fichier spécial: ***mknod()***

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/sysmacro.h>

#include <fcntl.h>

#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);

pathname: spécifie le nom du fichier à créer

mode: indique les permissions et le type (***S_IFCHR***, ***S_IFBLK***) du fichier à créer

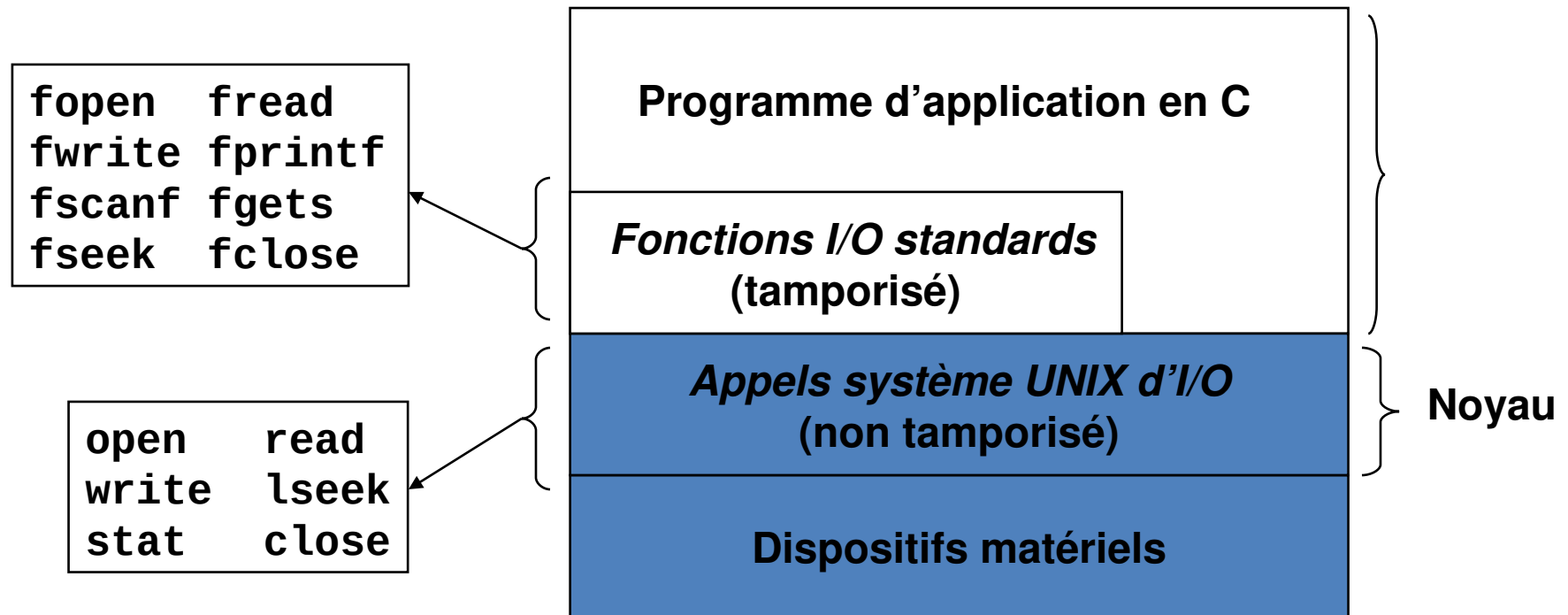
dev: identificateur (numéros mineur et majeur) du périphérique correspondant au fichier spécial

Appels système

- E/S sur des périphériques (Approche UNIX/LINUX)
 - Les E/S sur périphériques sont similaires à celles faites sur des fichiers réguliers
 - L'ouverture du périphérique est effectuée par la primitive ***open()*** en spécifiant le nom du fichier spécial. Le noyau retourne un descripteur d'E/S correspondant au périphérique. Le processus appelant peut alors accéder au périphérique par des appels système ***read()***, ***write()***, ***seek()***
 - La primitive ***close()*** permet de fermer le périphérique

Appels système

- E/S sur des périphériques



I/O Standard vs I/O UNIX

- Quand doit-on utiliser les fonctions standards d'I/O?
 - Autant que possible!
- Pourquoi alors utiliser les fonctions d'I/O de UNIX?
 - Pour obtenir les informations d'un fichier (ex: dimension, date de modification)
 - Utilisation de la fonction de bas niveau ***stat()***
 - Pour la programmation réseau
 - Il survient parfois de mauvaises interactions entre les sockets et les fonctions de la librairie standard
 - Pour de meilleures performances

Informations retournées par *stat()*

```
/*
 * file info returned by the stat function
 */
struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* protection and file type */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last change */
};
```

I/O fichier UNIX: *open()*

- Pour utiliser un fichier il faut au préalable l'ouvrir par la fonction *open()*.

```
int fd;    /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- *open()* retourne un entier (descripteur de fichier)
 - *fd* < 0 indique qu'une erreur d'ouverture est survenue
- Descripteurs de fichiers prédéfinis:
 - 0: *stdin*
 - 1: *stdout*
 - 2: *stderr*

I/O fichier UNIX: *read()*

- *read()* permet à un programme de retirer de l'information d'un fichier à partir de la position courante dans le fichier.

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* open the file */
...
/* read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- *read()* retourne le nombre d'octets lus à partir du fichier *fd*.
 - *nbytes* < 0 indique qu'une erreur est survenue.
 - Selon la disponibilité des données (*nbytes* < *sizeof(buf)*) est possible et ce sans erreur!
- Les *nbytes* octets sont insérés en mémoire à l'adresse *buf*

I/O fichier UNIX: *read()*

```
#include <unistd.h>

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```


I/O fichier UNIX: *write()*

- ***write()*** permet à un programme de modifier le contenu d'un fichier à partir de la position courante dans le fichier.

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* open the file */
...
/* write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

- ***write()*** retourne le nombre d'octets écrit à partir du tampon ***buf*** au fichier ***fd***.
 - ***nbytes*** < 0 indique qu'une erreur est survenue.
 - Dans certain cas, (***nbytes*** < ***sizeof(buf)***) est possible et ce sans erreur!

I/O fichier UNIX: *write()*

```
#include <unistd.h>

int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",46);
    exit(0);
}
```

I/O fichier UNIX: *open()* , *read()* et *write()* (version caractère)

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char c;
    int in, out;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);

    exit(0);
}
```

I/O fichier UNIX: *open()* , *read()* et *write()* (version bloc)

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in, block, sizeof(block))) > 0)
        write(out, block, nread);

    exit(0);
}
```

I/O fichier UNIX: Problématique des Short Counts

- **Possibilités d'occurrences des Short counts:**
 - Rencontre du EOF lors de la lecture d'un fichier.
 - Lecture d'une ligne au terminal.
 - E/S sur un socket réseau ou un tube anonyme UNIX (pipe).
- **Occurrences impossibles de Short counts :**
 - Lecture sur disque sauf lors de la lecture du EOF.
 - Ecriture sur un fichier sur disque.
- **Comment gérer les short counts dans un programme?**
 - Utilisez les fonctions RIO (Robust I/O) dont le code source est dans le fichier csapp.c.

Les fonctions RIO

- Les fonctions RIO permettent des E/S efficaces et robustes dans des applications comme la programmation réseau qui sont sujettes aux short counts.
- RIO fournit deux types de fonctions
 - Non-tamposées permettant les E/S de données binaires
 - ***rio_readn()*** et ***rio_writen()***
 - Tamposées permettant la lecture de données binaires ou de lignes de texte
 - ***rio_readlineb()*** et ***rio_readnb()***
 - Ces fonctions sont **thread-safe** et peuvent donc être entrelacées arbitrairement sur le même descripteur.
- Téléchargement
`csapp.cs.cmu.edu/public/ics/code/src/csapp.c`
`csapp.cs.cmu.edu/public/ics/code/include/csapp.h`

E/S non-temposi  e (RIO)

- M  me interface que les fonctions Unix ***read()*** et ***write()***
- Sp  cialement utile pour transf  rer des donn  es par sockets

```
#include "csapp.h"
```

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n);  
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Return: # d'octets transf  r  s SI OK, 0 SI EOF (*rio_readn* seulement), -1 SI erreur

- ***rio_readn()*** retourne un short count seulement si elle rencontre un EOF.
- ***rio_writen()*** ne retourne jamais de short count.
- Les appels    ***rio_readn()*** et ***rio_writen()*** peuvent   tre entrelac  es sur le m  me descripteur.

Implémentation de rio_readn()

```
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by sig
                                handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);        /* return >= 0 */
}
```


Lecture tamposiées (RIO)

- Lecture efficace de lignes de texte ou de données binaires à partir d'un fichier partiellement stocké dans un tampon en **DAM**

```
#include "csapp.h"
```

```
void rio_readinitb(rio_t *rp, int fd);
```

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
```

```
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Retourne: # d'octets lus SI OK, 0 sur un EOF, -1 SI erreur

- ***rio_readlineb()*** lit une ligne de texte d'au plus maxlen octets à partir du fichier ***fd*** et emmagasine cette ligne dans le tampon ***usrbuf***.
 - Utile pour lire sur un socket.
- ***rio_readnb()*** lecture de n octets à partir du fichier ***fd***.
- Les appels à ***rio_readlineb()*** et ***rio_readnb()*** peuvent être entrelacés sur le même descripteur.
 - DANGER: Ne pas entrelacer avec des appels à ***rio_readn()***

Lecture tamposiées (RIO)

- Structure *rio_t*

```
└ #define RIO_BUFSIZE 8192
└ typedef struct {
    int rio_fd;           /* descriptor for this internal buf */
    int rio_cnt;          /* unread bytes in internal buf */
    char *rio_bufptr;     /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

- Fonction d'initialisation de la structure *rio_t* (*rio_readinitb()*)

```
└ void rio_readinitb(rio_t *rp, int fd)
    {
        rp->rio_fd = fd;
        rp->rio_cnt = 0;
        rp->rio_bufptr = rp->rio_buf;
    }
```

Lecture tamposiées (RIO)

- Fonction *rio_readlineb()*

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
{
    int n, rc;
    char c, *bufp = usrbuf;

    for (n = 1; n < maxlen; n++) {
        if ((rc = rio_read(rp, &c, 1)) == 1) {
            *bufp++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return 0; /* EOF, no data read */
            else
                break;    /* EOF, some data was read */
        } else
            return -1;    /* error */
    }
    *bufp = 0;
    return n;
}
```

Lecture tamposiées (RIO)

- Fonction *rio_readnb()*

```
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = rio_read(rp, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by sig handler return */
                nread = 0;      /* call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

Lecture tamposiées (RIO)

- Fonction *rio_read()*

```
static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
{
    int cnt;

    while (rp->rio_cnt <= 0) { /* refill if buf is empty */
        rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
                           sizeof(rp->rio_buf));
        if (rp->rio_cnt < 0) {
            if (errno != EINTR) /* interrupted by sig handler return */
                return -1;
        }
        else if (rp->rio_cnt == 0) /* EOF */
            return 0;
        else
            rp->rio_bufptr = rp->rio_buf; /* reset buffer ptr */
    }

    /* Copy min(n, rp->rio_cnt) bytes from internal buf to user buf */
    cnt = n;
    if (rp->rio_cnt < n)
        cnt = rp->rio_cnt;
    memcpy(usrbuf, rp->rio_bufptr, cnt);
    rp->rio_bufptr += cnt;
    rp->rio_cnt -= cnt;
    return cnt;
}
```

Exemple d'utilisation des fonctions RIO

- Copier des lignes de texte du standard input (clavier) vers le standard output (écran).

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
```

Exemple d'utilisation de *stat()*

```
/* statcheck.c - Querying and manipulating a file's meta data */  
#include "csapp.h"
```

```
int main (int argc, char **argv)  
{  
    struct stat stat;  
    char *type, *readok;  
  
    Stat(argv[1], &stat);  
    if (S_ISREG(stat.st_mode)) /* file type*/  
        type = "regular";  
    else if (S_ISDIR(stat.st_mode))  
        type = "directory";  
    else  
        type = "other";  
    if ((stat.st_mode & S_IRUSR)) /* OK to read?*/  
        readok = "yes";  
    else  
        readok = "no";  
  
    printf("type: %s, read: %s\n", type, readok);  
    exit(0);  
}
```

```
bass> ./statcheck statcheck.c  
type: regular, read: yes  
bass> chmod 000 statcheck.c  
bass> ./statcheck statcheck.c  
type: regular, read: no
```

Fonctions d'E/S Standard

- La librairie standard C (`libc.a`) contient une collection de fonctions d'E/S standards de haut niveau.
- Exemples de ces fonctions:
 - Ouverture et fermeture (***fopen()*** et ***fclose()***)
 - Lecture et écriture binaire (octets) (***fread()*** et ***fwrite()***)
 - Lecture et écriture de lignes de texte (***fgets()*** et ***fputs()***)
 - Lecture et écriture formatée (***fscanf()*** et ***fprintf()***)

Flot (stream) d'E/S Standard

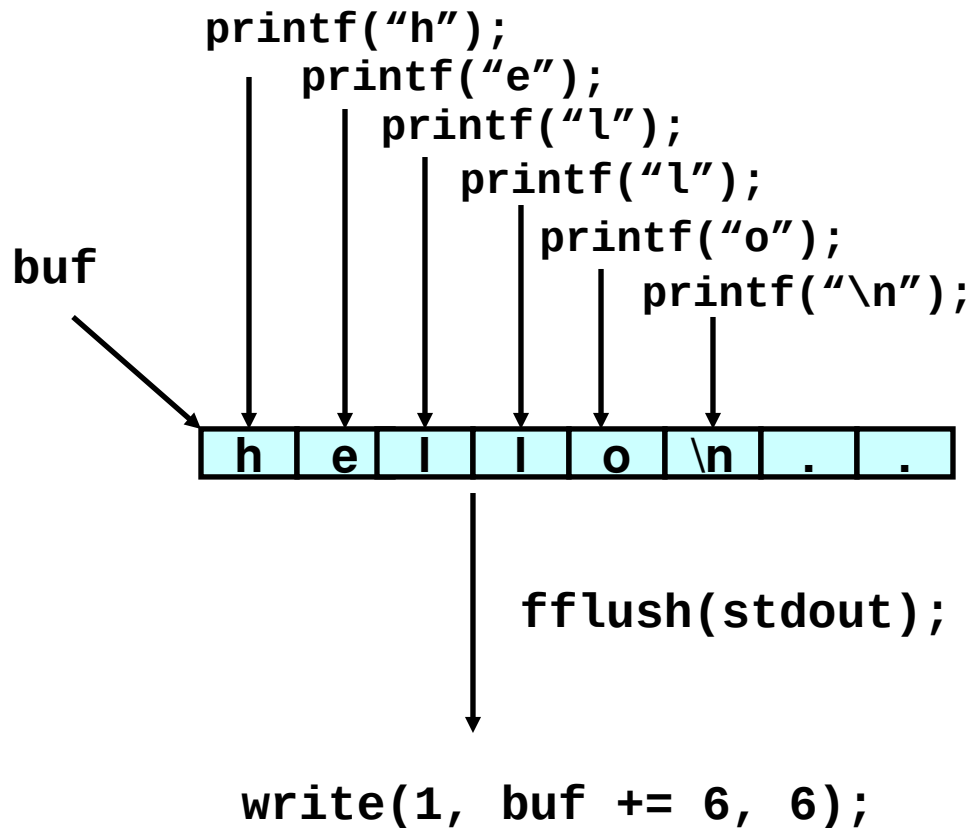
- Les E/S Standard utilisent des fichiers ouverts comme des *streams*
 - Abstraction correspondant à un descripteur de fichier et un tampon en mémoire.
- Chaque processus démarré correspondant à un programme C est associé à trois ***streams*** (défini dans stdio.h)
 - stdin (standard input)
 - stdout (standard output)
 - stderr (standard error)

```
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

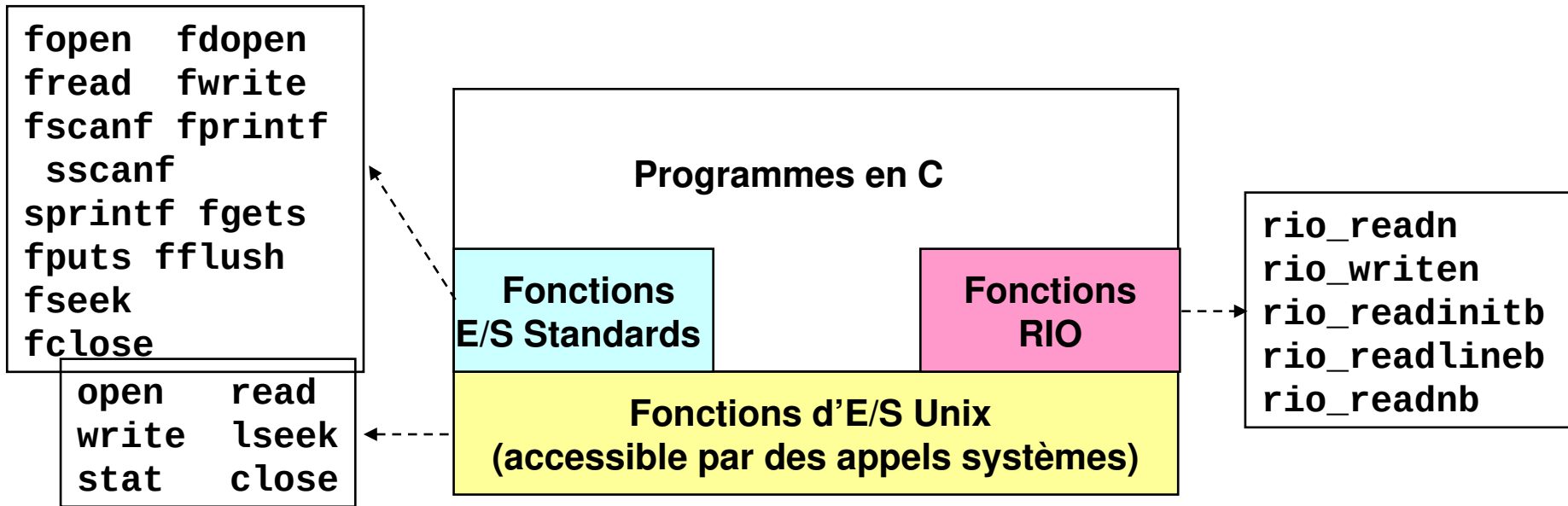
Tamponnement des E/S Standard

- Les fonctions d'E/S Standard utilisent la tamponnement



E/S Unix vs. E/S Standard vs. RIO

- Les E/S Standard et RIO sont implémentées utilisant les appels systèmes d'E/S UNIX.



- Lesquelles doit-t-ont utiliser dans nos programmes?

+/- des E/S Unix

- +:
 - E/S Unix sont les plus générales et occasionnent le moins d'overhead.
 - Les autres bibliothèques d'E/S sont implémentées par dessus les fonctions d'E/S UNIX.
 - E/S Unix fournit des fonctions permettant d'accéder aux metadata des fichiers (stat()).
- -:
 - La gestion des short counts est problématique.
 - La lecture efficace de lignes de texte requiert de la tamponisation et est aussi problématique.
 - Ces problématiques sont corrigées avec les fonctions d'E/S standards et la bibliothèque RIO.

+/- des E/S Standards

- **+:**
 - La tamponsation augmente l'efficacité des E/S en diminuant le nombre d'appels systèmes ***read()*** et ***write()***.
 - Les shorts counts sont traités automatiquement,
- **-:**
 - Ne permettent pas l'accès aux métadonnées des fichiers.
 - Ces fonctions d'E/S ne sont pas appropriées pour des E/S sur des sockets réseaux.

+/- des E/S Standards

- Restrictions sur les streams:
 - **Restriction 1:** Une fonction d'input ne peut suivre une fonction d'output sans intercaler un appel *fflush()*, *fseek()*, *fsetpos()*, ou *rewind()*.
 - Ces 3 fonctions utilisent *lseek()* pour changer la position dans un fichier.
 - **Restriction 2:** Une fonction d'output ne peut suivre une fonction d'input sans intercaler un appel à *fseek()*, *fsetpos()*, ou *rewind()*.
- Restriction sur les sockets:
 - Il est impossible de faire un *lseek()* sur un socket

+/- des E/S Standards

- **Correction de la restriction 1:**
 - Utiliser une fonction ***fflush()*** pour vider le stream après chaque output.
- **Correction de la restriction 2:**
 - Ouvrir 2 streams sur le même descripteur, un en lecture et l'autre en écriture:

```
FILE *fpin, *fpout;  
  
fpin = fdopen(sockfd, "r");  
fpout = fdopen(sockfd, "w");
```

- Cependant, il faut fermer le même descripteur deux fois:

```
fclose(fpin);  
fclose(fpout);
```

- Occasionne une course fatale (race) pour des programmes utilisant des threads concurrents.

Choix des fonctions d'E/S

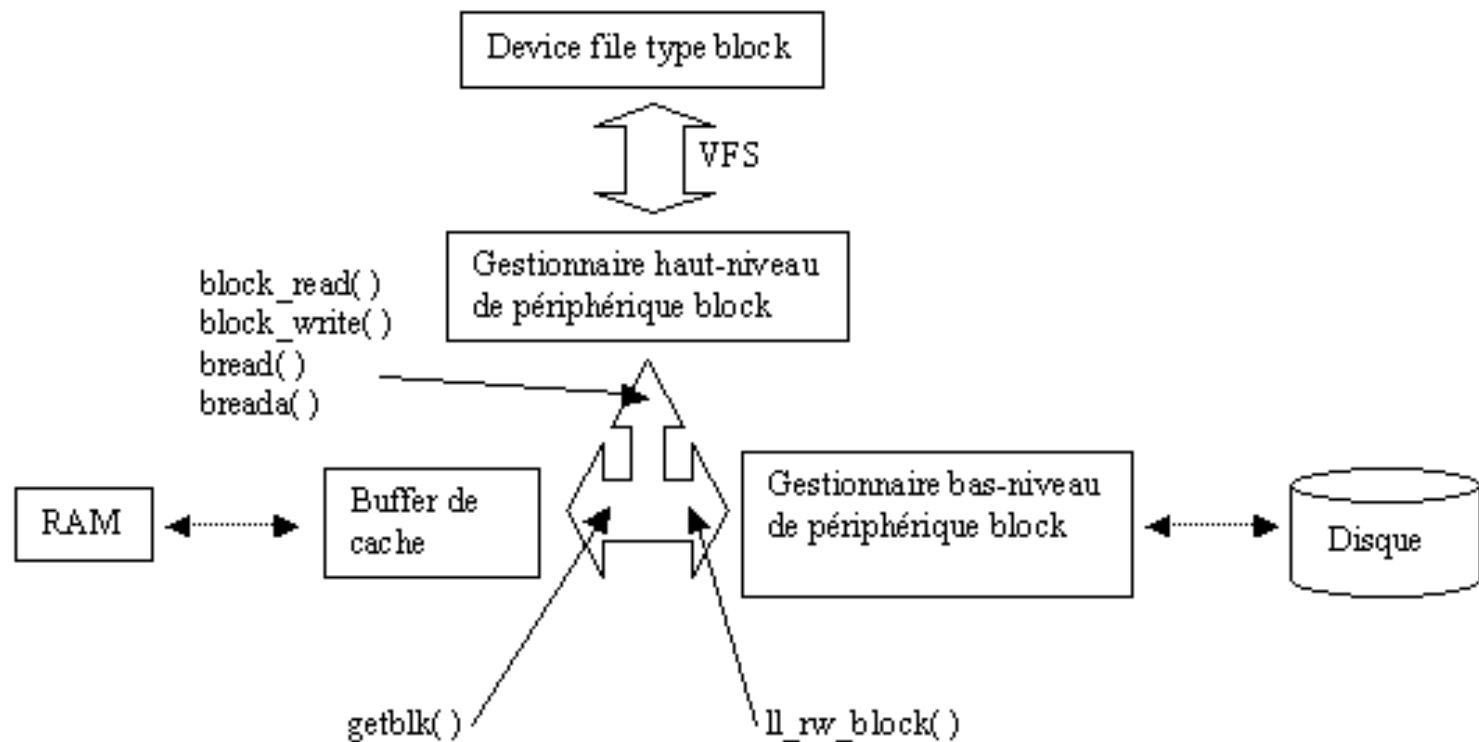
- Règle Générale: Utiliser le plus possible les fonctions d'E/S Standards.
- Quand spécifiquement utiliser les E/S standard ?
 - Avec les E/S disque ou au terminal (console).
- Quand utiliser les E/S UNIX
 - Pour accéder aux métadonnées d'un fichier.
 - Pour des besoins strictes de performance.
- Quand utiliser les E/S RIO?
 - Pour les E/S sockets ou les pipes.

Implémentation générale des E/S

- Périphériques supportés par LINUX
 - LINUX maintient la liste des pilotes de périphérique supportés, en utilisant deux tableaux: ***blkdevs*** et ***chrdevs***, qui contiennent respectivement, les descripteurs des périphériques en mode bloc et ceux en mode caractère
 - La structure ***device_struct*** (définie dans ***fs/devices.c***) représente le format de chaque entrée de ces tableaux
 - ***const char * name;*** // nom du périphérique
 - ***struct file_operations * fops;*** // opérations sur fichiers associées au périphérique

Implémentation générale des E/S

- E/S sur disque (périphérique bloc)



Implémentation générale des E/S

- E/S sur disque (périphérique bloc)
 - Un pilote de périphérique bloc est divisé en deux parties: un pilote haut niveau qui s'interface avec le système de fichiers, et un pilote de bas niveau qui traite le périphérique matériel
 - Un processus qui effectue un appel système ***read()*** ou ***write()*** sur un fichier de périphérique bloc verra ses appels acheminés au gestionnaire de fichiers
 - Le gestionnaire de fichiers appelle une procédure dans un gestionnaire de périphérique haut niveau
 - Cette procédure exécute toutes les actions liées à la requête d'I/O spécifique au périphérique matériel

Implémentation générale des E/S

- E/S sur disque (périphérique bloc)
 - LINUX offre deux fonctions générales, ***block_read()*** et ***block_write()*** permet d'exécuter les requêtes d'I/O
 - Ces fonctions traduisent le requête d'accès d'un fichier de périphérique d'I/O en requête pour certains blocs sur le périphérique matériel correspondant
 - Les blocs requis peuvent être déjà dans la RAM, donc ***block_read()*** et ***block_write()*** appellent ***getblk()*** pour vérifier si le buffer cache contient déjà le bloc et qu'il n'a pas été modifié depuis le dernier accès

Implémentation générale des E/S

- E/S sur disque (périphérique bloc)
 - Si le bloc n'est pas dans le buffer cache, la fonction ***getblk()*** doit traiter sa requête à partir du périphérique bloc en faisant appel à la fonction ***ll_rw_block()***
 - Cette fonction active un pilote de bas niveau qui gère le contrôleur de périphérique pour exécuter l'opération demandée sur le périphérique bloc
 - Lorsqu'une opération d'I/O s'effectue sur un bloc spécifique, l'accès direct est exécuté par ***bread()*** ou ***breada()*** qui à son tour appelle les fonctions ***getblk()*** et ***ll_rw_block()***

Implémentation générale des E/S

- E/S sur disque (périphérique bloc)
 - Les E/S en mode bloc sont satisfaites par le **buffer cache**: les blocs sont chargés dans des tampons mémoire en appelant les fonctions **getblk()**, **bread()** et **breada()**. Ces fonctions font appel à un module d'E/S pour lire ou écrire des blocs sur disque
 - Le rôle de ce module d'E/S consiste à maintenir la liste des requêtes d'E/S. Quand un ou plusieurs blocs doivent être lus ou écrits sur un disque, la requête d'E/S est ajoutée dans la liste associée au périphérique physique. Cette liste est explorée par le pilote de périphérique qui effectue les E/S une à la fois.

Implémentation générale des E/S

- E/S sur disque
 - Pour optimiser les temps de déplacement des têtes de lecture/écriture sur disque, la liste des requêtes est maintenue triée sur le numéro de secteur physique. De cette façon, les requêtes sur des secteurs proches sont effectuées en minimisant le déplacement des têtes
 - La structure ***request*** (définie dans ***linux/blkdev.h***) permet de spécifier le type des éléments de la liste des requêtes. Les champs principaux de cette structure:
kdev_t rq_dev; // identificateur du périphérique
int cmd; // commande: READ ou WRITE
unsigned long sector; // secteur du début
unsigned long nr_sector; // nombre de secteurs à lire ou écrire
char *buffer; // adresse des données à lire ou écrire

Implémentation générale des E/S

- E/S sur disque
 - La structure ***request*** (définie dans ***linux/blkdev.h***) permet de spécifier le type des éléments de la liste des requêtes

Descripteur de requête *request*

<i>int</i>	<i>rq_status</i>	statut de la requête
<i>kdev_t</i>	<i>rq_dev</i>	identificateur de périphérique
<i>int</i>	<i>cmd</i>	opération demandée (READ ou WRITE)
<i>int</i>	<i>errors</i>	code de réussite ou d'échec
<i>unsigned long</i>	<i>sector</i>	1 ^{er} numéro de secteur correspondant au 1 ^{er} block de la requête
<i>unsigned long</i>	<i>nr_sector</i>	nombre de secteurs de la requête à transférer
<i>unsigned long</i>	<i>current_nr_sector</i>	nombre de secteurs dans le block courant
<i>char *</i>	<i>buffer</i>	espace mémoire pour le transfert de données
<i>struct semaphore *</i>	<i>sem</i>	sémaphore de requêtes
<i>struct buffer_head *</i>	<i>bh</i>	descripteur du premier buffer
<i>struct buffer_head *</i>	<i>bhtail</i>	descripteur du dernier buffer
<i>struct request *</i>	<i>next</i>	pointeur vers la prochaine requête dans la liste

rq_status:

RQ_ACTIVE à être exécutée ou en cours d'exécution par le pilote bas-niveau du périphérique
RQ_INACTIVE descripteur de requête présentement non utilisé

Implémentation détaillée des E/S

- E/S sur disque
 - Les fonctions de gestion des listes de requêtes d'E/S sur disque sont implémentées dans le fichier ***drivers/block/ll_rw_blk.c***. Plusieurs fonctions sont définies:
 - ***get_request()***: cette fonction recherche un descripteur de requête libre dans la liste globale. Le descripteur retourné est initialisé et mis à l'état actif
 - ***add_request()***: permet d'ajouter une requête d'E/S à la liste correspondant à un périphérique, en maintenant cette liste triée
 - ***make_request()***: permet la création d'une requête d'E/S et l'insère dans la liste de périphériques appropriée (***add_request()***)
 - ***ll_rw_block()***: cette fonction est appelée par le ***buffer cache***, ***les systèmes de fichiers*** et le ***module d'E/S de blocs*** pour effectuer une E/S. Crée une requête d'E/S en appelant ***make_request()***. Le buffer correspondant est verrouillé tant que l'E/S n'est pas terminée

Implémentation détaillée des E/S

- E/S sur périphériques en mode bloc
 - Ces périphériques (disques durs) ont un temps d'accès assez élevé. Pour améliorer les performances, plusieurs blocs adjacents sont transférés à la fois. Le noyau fournit les services suivants pour supporter les périphériques de type bloc:
 - Offre une interface uniforme par le système de fichier virtuel VFS
 - Implémente le read-ahead (lecture anticipée) des données sur disque
 - Fournit une cache disque pour les données
 - Opérations d'E/S tampon
 - Données gardées dans un tampon
 - Chaque tampon est associé à un bloc spécifique identifié par un numéro de périphérique et un numéro de bloc
 - Opérations d'E/S asynchrones. Le noyau peut faire autre chose quand l'opération d'E/S est lancée

Implémentation détaillée des E/S

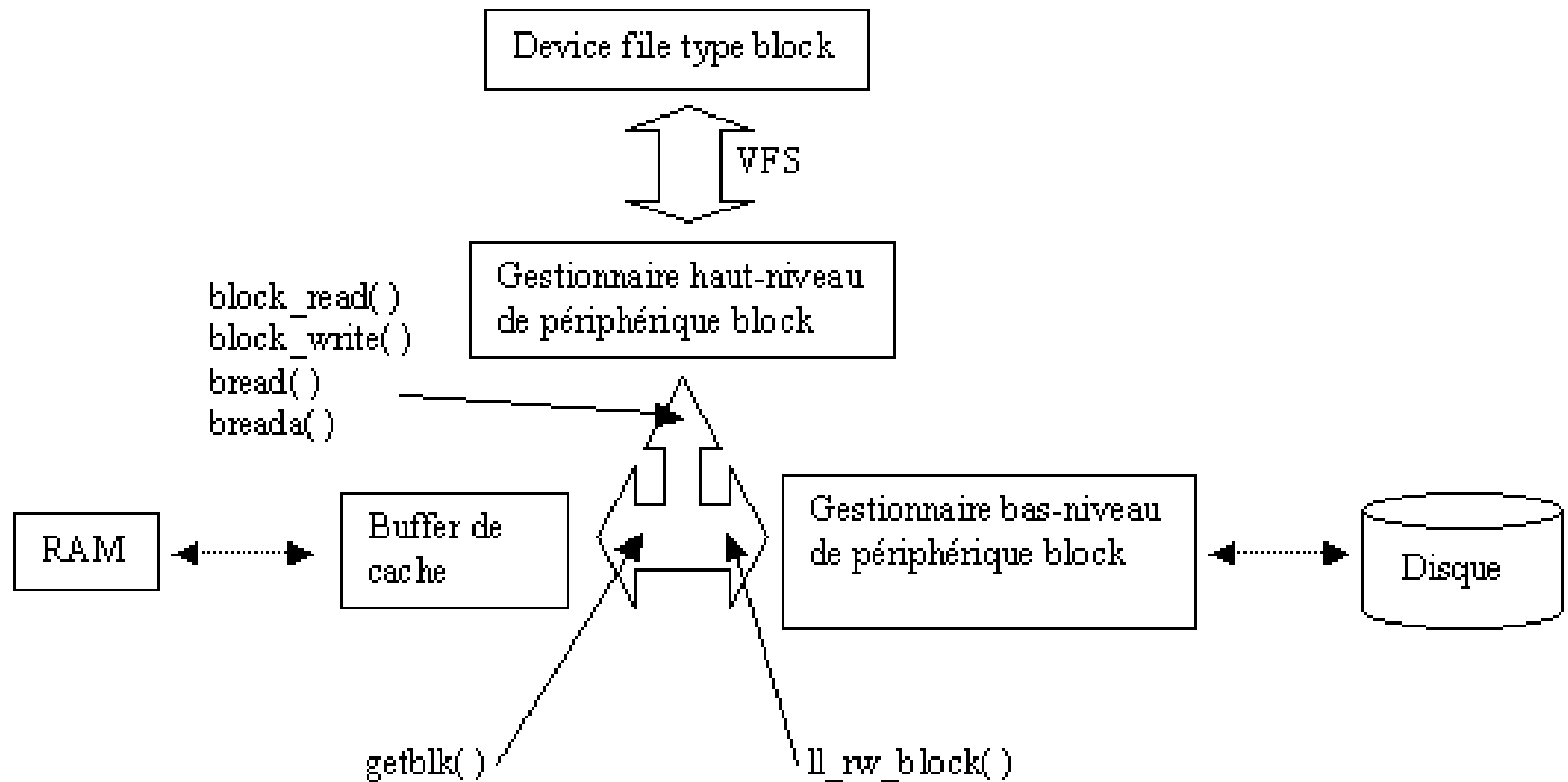
- E/S sur périphériques en mode bloc
 - Secteurs, blocs et tampons
 - Chaque transfert de données d'un périphérique bloc agit sur un groupe d'octets adjacents appelé secteur (ex: 512 octets). Le secteur est l'unité de base d'un transfert de données
 - Un bloc est un groupe d'octets adjacents pouvant être transférés lors d'une opération d'E/S. Sous LINUX un bloc à une puissance de 2 d'octets et un multiple de la taille d'un secteur. Sur PC, les tailles de bloc permises sont 512, 1024, 2048 et 4096 octets
 - Chaque bloc possède son propre tampon dans la mémoire RAM

Implémentation détaillée des E/S

- E/S sur périphériques en mode bloc
 - Les fonctions de lecture et d'écriture de données sur des périphériques accessibles en mode bloc sont implémentées dans le fichier ***fs/block_dev.c***. Ces fonctions utilisent les primitives du ***buffer cache*** pour accéder au tampon mémoire associés aux périphériques
 - Les fonctions ***block_write()***, ***block_read()*** et ***block_sync()*** implémentent les opérations sur fichiers ***write()***, ***read()*** et ***fsync()*** associées aux périphériques en mode bloc
 - La fonction ***block_write()*** implémente l'écriture de données sur un périphérique. Elle copie les données à écrire depuis l'adresse spécifiée par le processus appelant dans les tampons mémoire (buffer) créés, puis elle écrit ces tampons par la fonction ***ll_rw_block()***. Elle utilise aussi un mécanisme de lecture anticipée pour charger en mémoire les blocs de données suivants (disques à accès séquentiel, amélioration des performances des disques).
 - La fonction ***block_read()*** implémente la lecture de données depuis un périphérique, en utilisant la lecture anticipée permettant la lecture de plusieurs blocs. La fonction ***ll_rw_block()*** est ensuite appelée pour effectuer la lecture des blocs

Implémentation détaillée des E/S

- E/S sur disque



Implémentation détaillée des E/S

- E/S sur disque (fonctions)
 - ***int block_read(int dev, unsigned long *pos, char *buf, int count)***
 - ***dev***: descripteur d'un périphérique bloc
 - ***pos***: adresse de la position du premier octet à lire sur le périphérique
 - ***buf***: adresse d'une zone de mémoire où seront stockées les données lues par le périphérique
 - ***count***: # d'octets à transférer (lecture)
 - Cette fonction retourne le nombre d'octets de données effectivement lues

Implémentation détaillée des E/S

- E/S sur disque (fonctions)
 - ***block_read()***

```
int block_read(int dev, unsigned long * pos, char * buf, int count)
{
    int block = *pos / BLOCK_SIZE;
    int offset = *pos % BLOCK_SIZE;
    int chars;
    int read = 0;
    struct buffer_head * bh;
    register char * p;

    while (count > 0) {
        bh = bread(dev, block);
        if (!bh)
            return read ? read : -EIO;
        chars = (count < BLOCK_SIZE) ? count : BLOCK_SIZE;
        p = offset + bh->b_data;
        offset = 0;
        block++;
        *pos += chars;
        read += chars;
        count -= chars;
        while (chars-- > 0)
            put_fs_byte(*(p++), buf++);
        bh->b_dirt = 1;
        brelse(bh);
    }
    return read;
}
```

Implémentation détaillée des E/S

- E/S sur disque (fonctions)
 - ***int block_write(int dev, unsigned long *pos, char *buf, int count)***
 - ***dev***: descripteur d'un périphérique bloc
 - ***pos***: adresse de la position du premier octet où écrire sur le périphérique
 - ***buf***: adresse d'une zone de mémoire où seront extraites les données à écrire sur le périphérique
 - ***count***: # d'octets à transférer (écriture)
 - Cette fonction retourne le nombre d'octets de données effectivement écrits

Implémentation détaillée des E/S

- E/S sur disque (fonctions)
 - ***block_write()***

```
int block_write(int dev, long * pos, char * buf, int count)
{
    int block = *pos / BLOCK_SIZE;
    int offset = *pos % BLOCK_SIZE;
    int chars;
    int written = 0;
    struct buffer_head * bh;
    register char * p;

    while (count > 0) {
        bh = bread(dev, block);
        if (!bh)
            return written ? written : -EIO;
        chars = (count < BLOCK_SIZE) ? count : BLOCK_SIZE;
        p = offset + bh->b_data;
        offset = 0;
        block++;
        *pos += chars;
        written += chars;
        count -= chars;
        while (chars-- > 0)
            *(p++) = get_fs_byte(buf++);
        bh->b_dirty = 1;
        brelse(bh);
    }
    return written;
}
```

Implémentation détaillée des E/S

- E/S sur disque (fonctions)
 - ***void ll_rw_block(int rw, struct buffer_head *bh)***
 - ***rw***: type d'opération READ ou WRITE
 - ***bh***: adresse d'un descripteur de tampon ***bh*** spécifiant le bloc précédemment créé

```
void ll_rw_block(int rw, struct buffer_head * bh)
{
    blk_fn blk_addr;
    unsigned int major;

    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||!(blk_addr=rd_blk[major]))
        panic("Trying to read nonexistent block-device");
    blk_addr(rw, bh);
}
```

Implémentation détaillée des E/S

- E/S sur disque (Récapitulation)
 - Un processus ouvre un fichier spécial correspondant à un disque dur
 - Ce processus appelle les fonctions de haut niveau ***read()*** ou ***write()*** sur le fichier spécial
 - Le gestionnaire de haut-niveau fournit les fonctions correspondantes d'un périphérique bloc (***block_read()***, ***block_write()***, ***bread()***, ***breada()***)
 - Le gestionnaire de bas-niveau se charge de l'interaction avec le disque