Operating System

Dr. GuoJun LIU

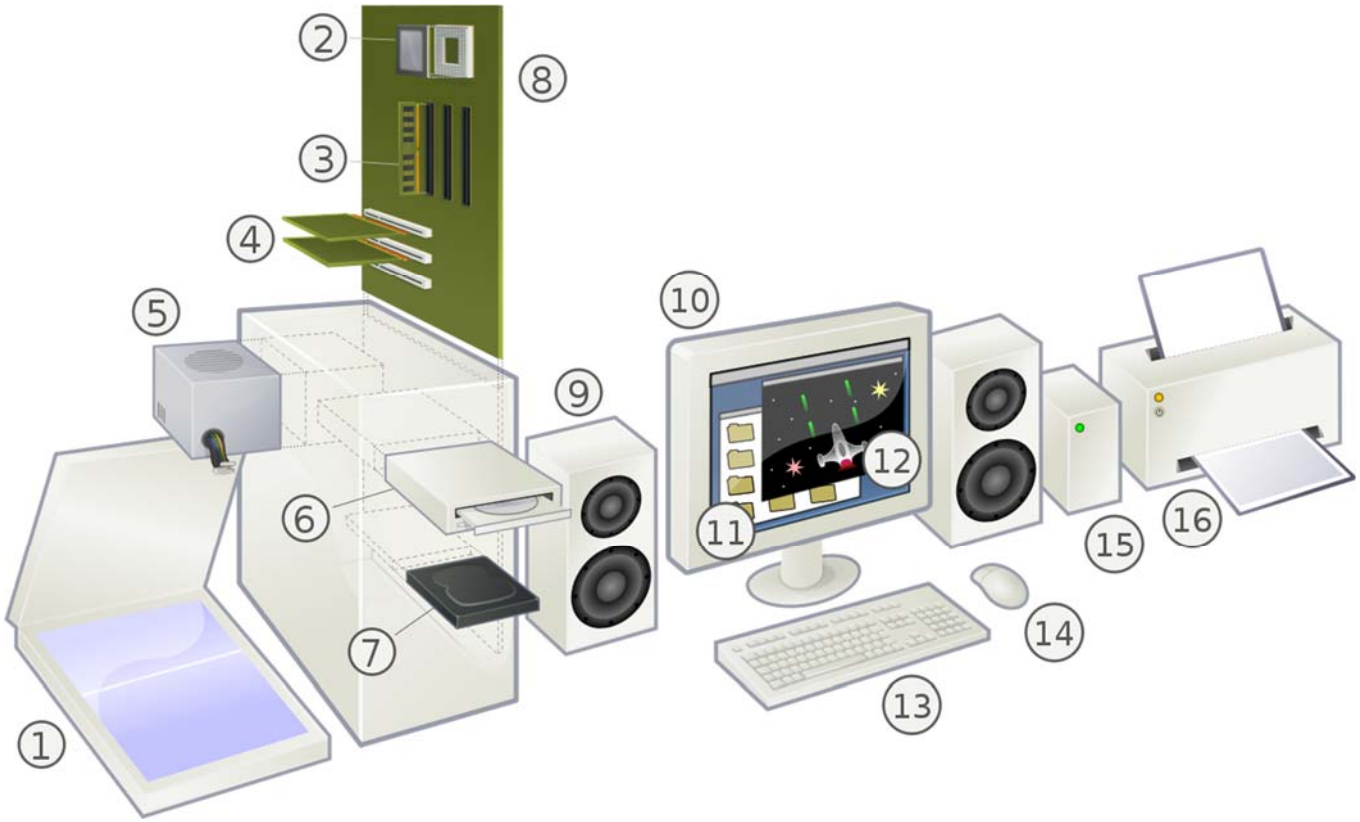
Harbin Institute of Technology

http://guojun.hit.edu.cn/os/

My Summary of OS

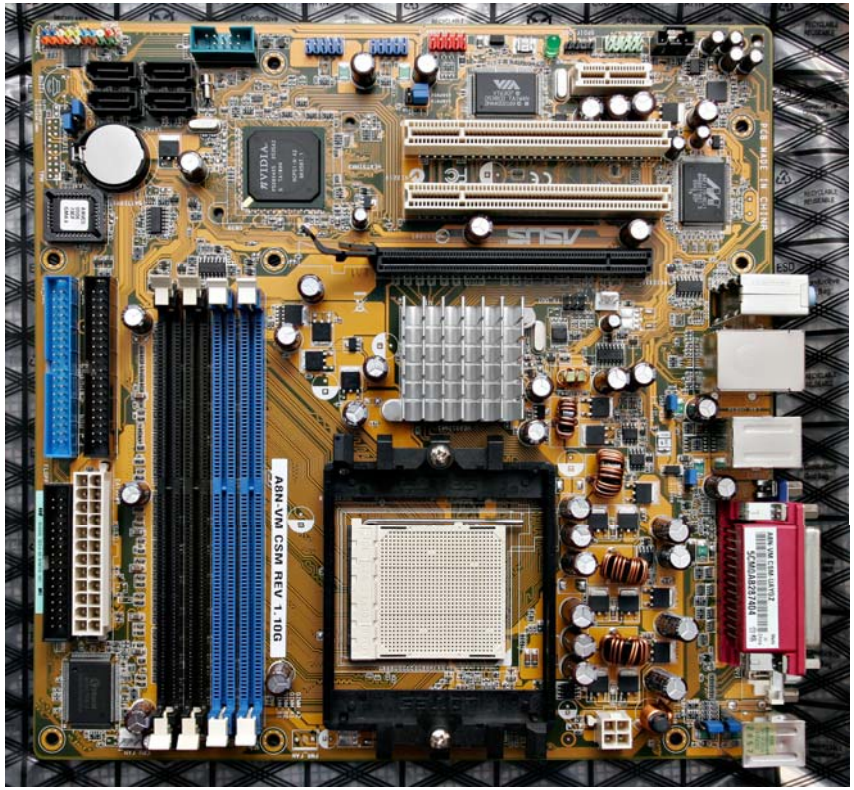
... ...

A modern PC and peripherals



An exploded view of a modern personal computer and peripherals

1. Scanner
2. CPU
3. Memory (RAM)
4. Expansion cards (graphics cards, etc.)
5. Power supply
6. Optical disc drive
7. Storage (Hard disk)
8. Motherboard
9. Speakers
10. Monitor
11. System software
12. Application software
13. Keyboard
14. Mouse
15. External hard disk
16. Printer



Motherboard

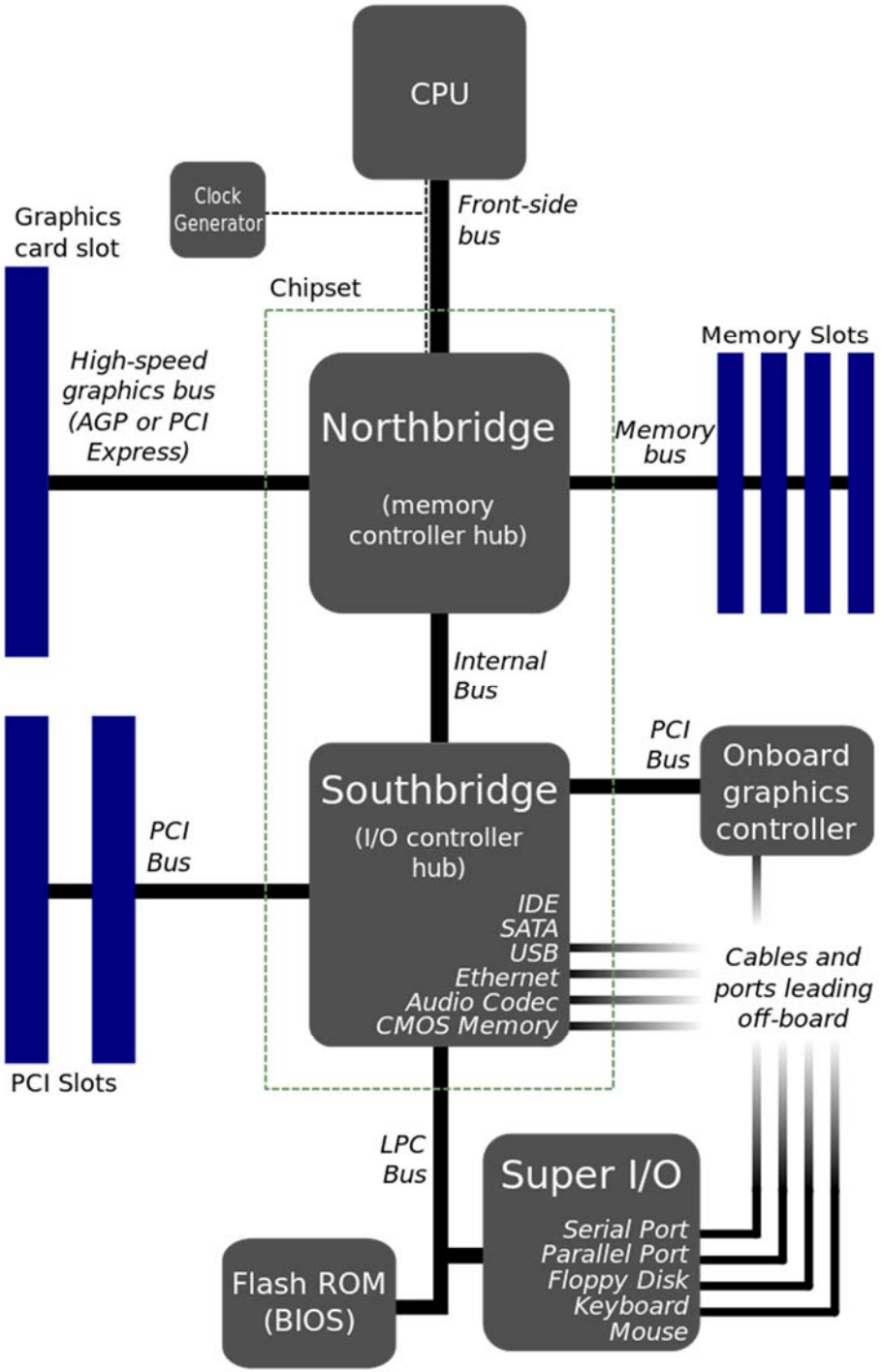
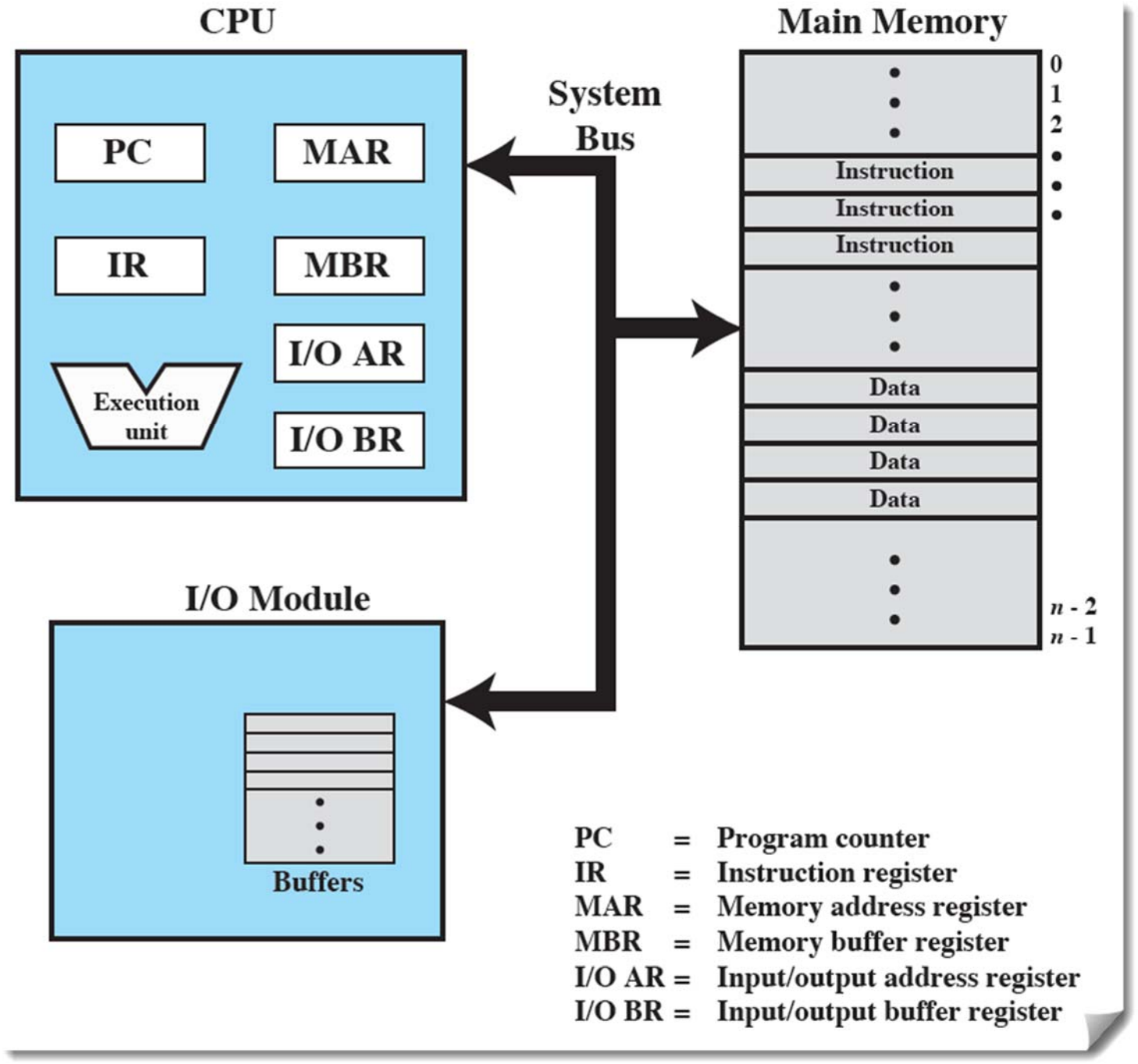


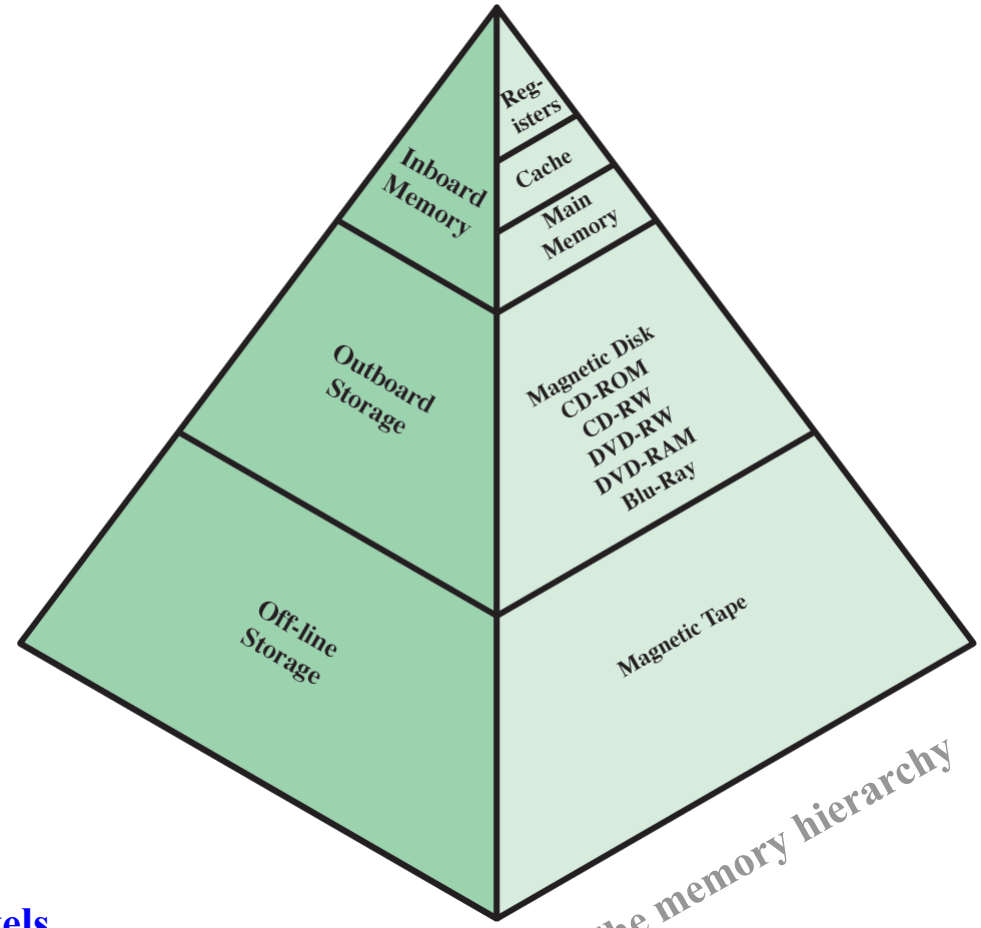
Photo from wikipedia.org



Computer Components: Top-Level View

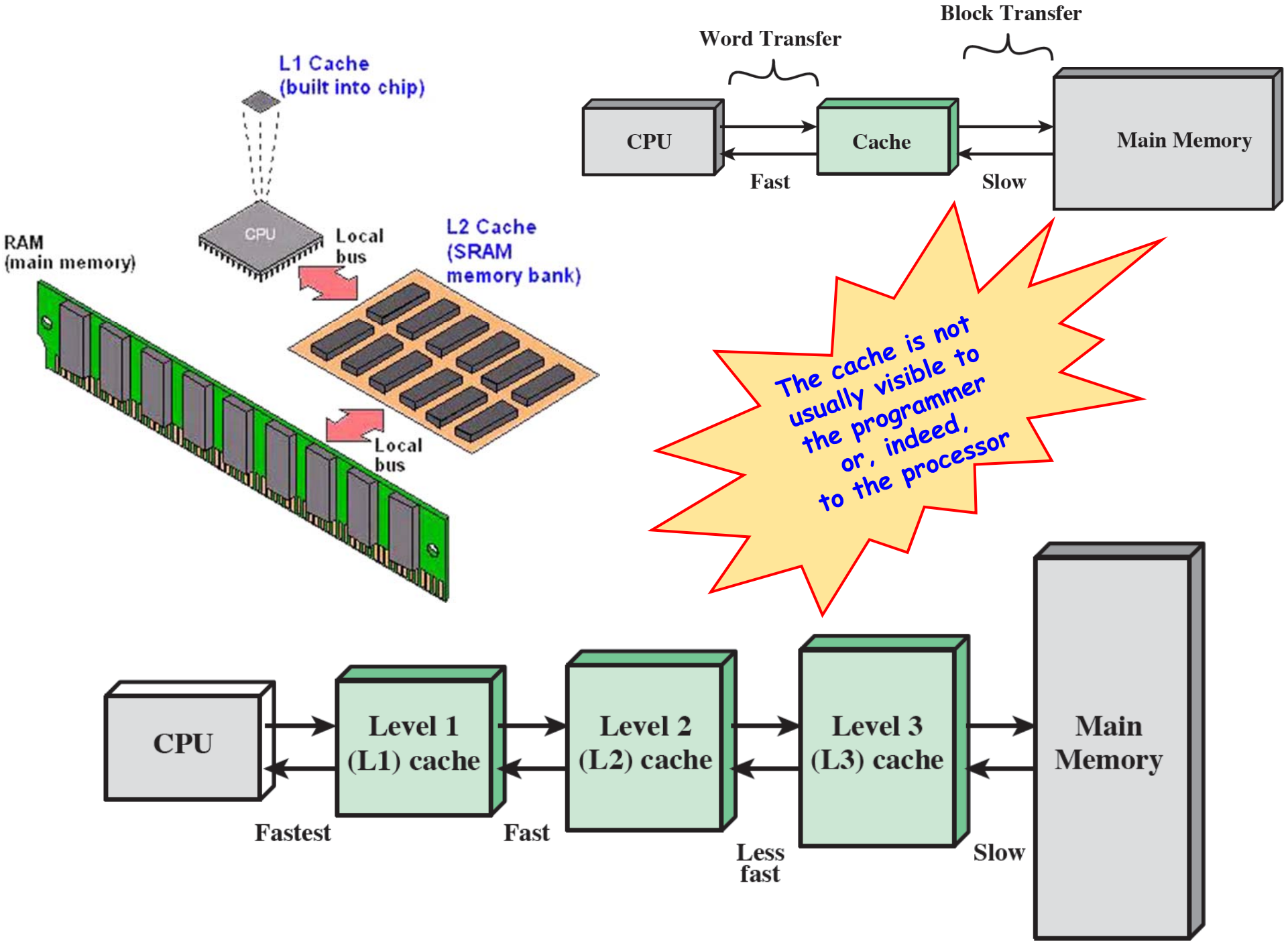
The Memory Hierarchy

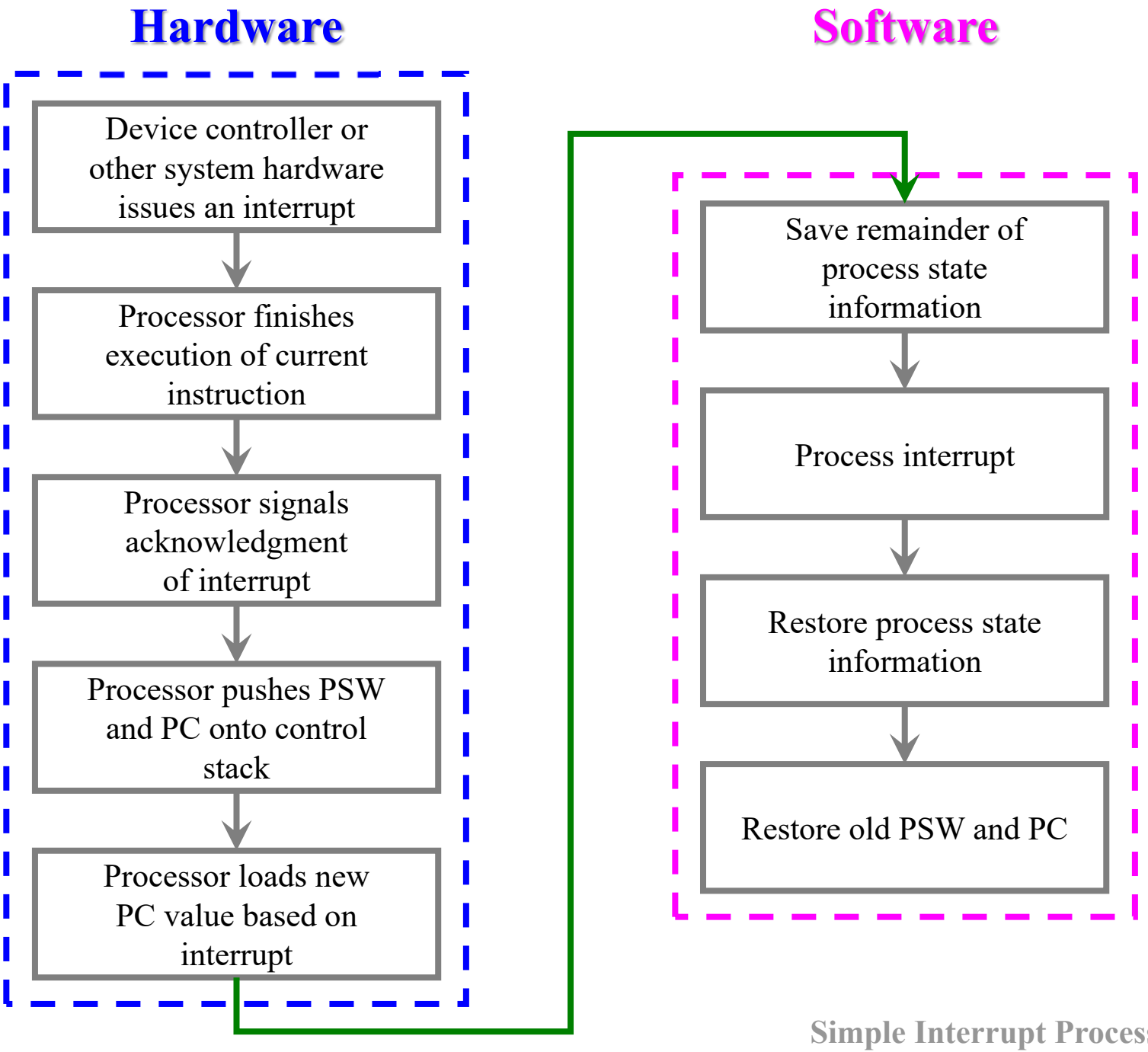
- **Going down the hierarchy:**
  a. decreasing cost per bit
  b. increasing capacity
  c. increasing access time
  d. decreasing frequency of access to the memory by the processor



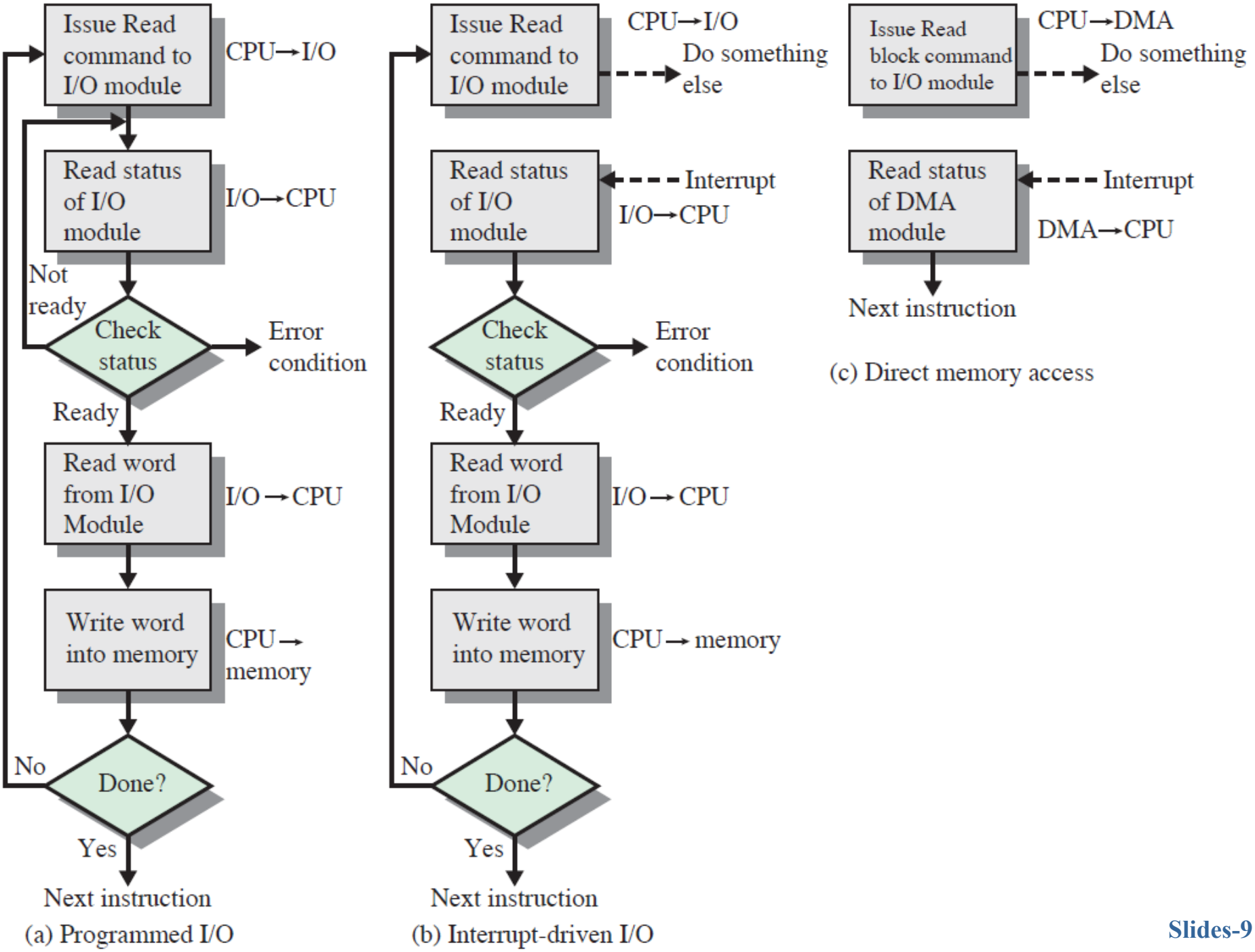
The memory hierarchy

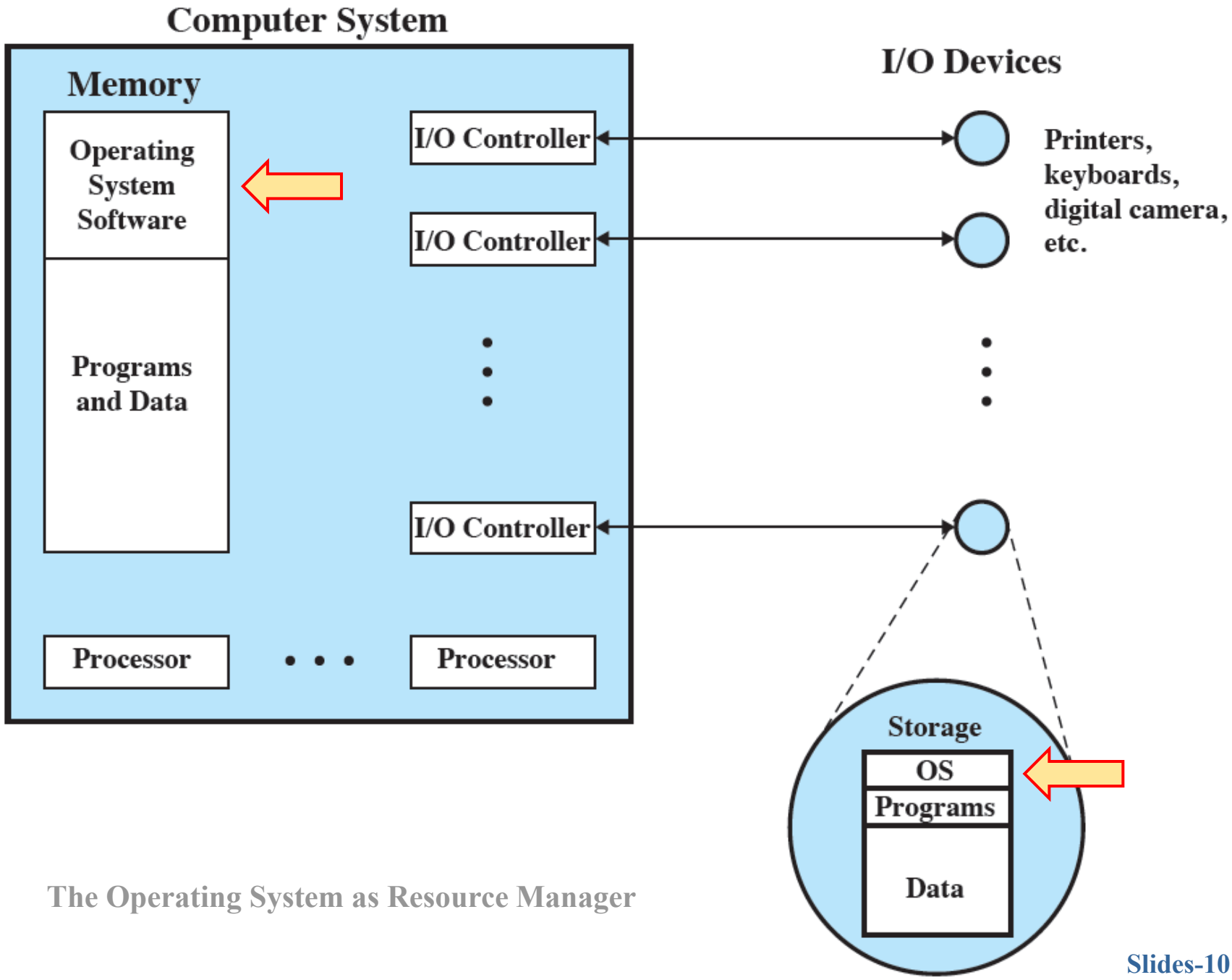
Decreasing frequency of access at lower levels


L1 Cache
(built into chip)
CPU
Local
bus
L2 Cache
(SRAM
memory bank)
RAM
(main memory)
Local
bus
Word Transfer
Block Transfer
CPU
Cache
Main Memory
Fast
Slow
The cache is not
usually visible to
the programmer
or, indeed,
to the processor
CPU
Level 1
(L1) cache
Level 2
(L2) cache
Level 3
(L3) cache
Main
Memory
Fastest
Fast
Less
fast
Slow



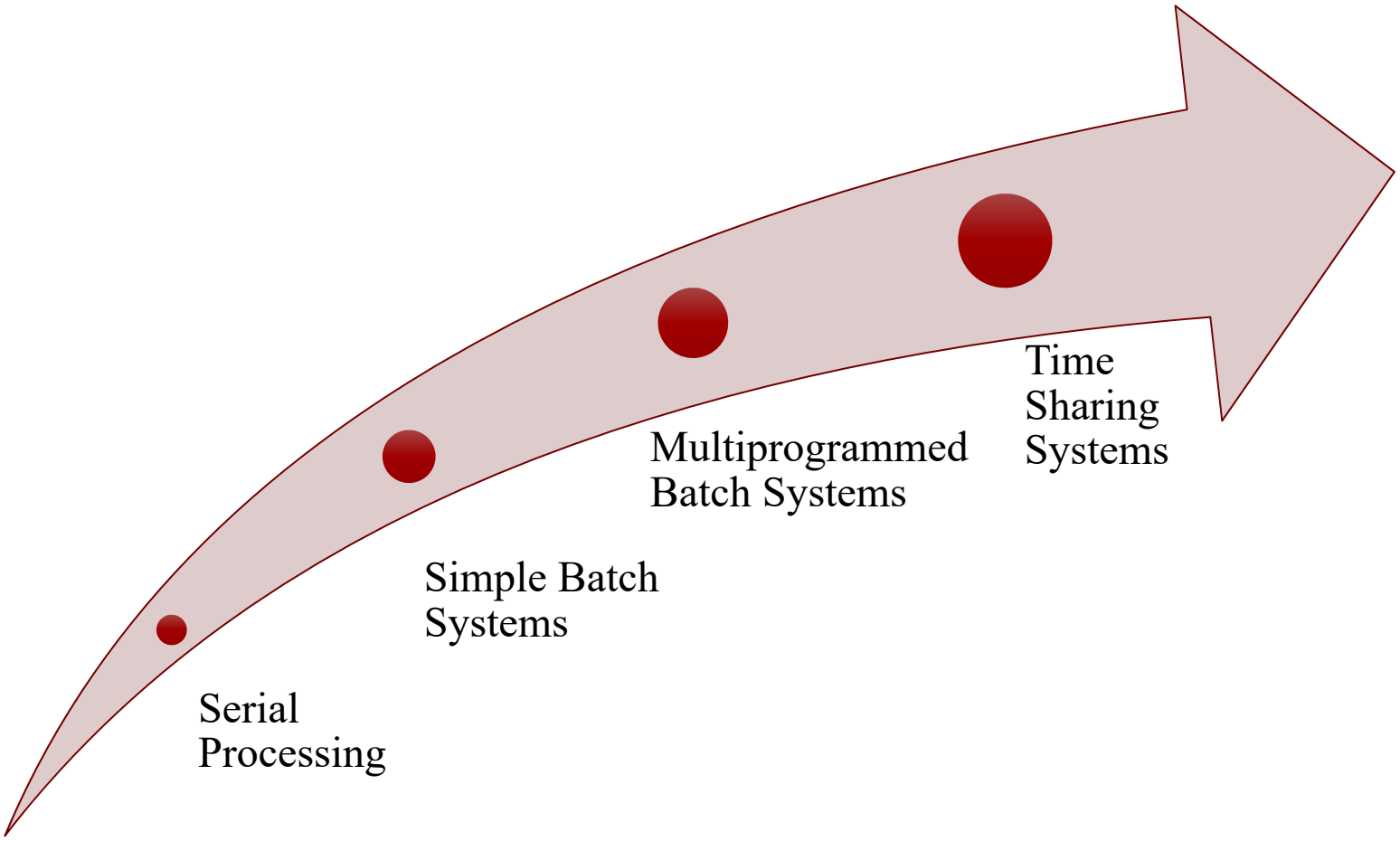
Simple Interrupt Processing


Issue Read command to I/O module
CPU→I/O
Read status of I/O module
I/O→CPU
Not ready
Check status
Error condition
Ready
Read word from I/O Module
I/O→CPU
Write word into memory
CPU→ memory
No
Done?
Yes
Next instruction
(a) Programmed I/O
Issue Read command to I/O module
CPU→I/O
Do something else
Read status of I/O module
Interrupt
I/O→CPU
Check status
Error condition
Ready
Read word from I/O Module
I/O→CPU
Write word into memory
CPU→ memory
No
Done?
Yes
Next instruction
(b) Interrupt-driven I/O
Issue Read block command to I/O module
CPU→DMA
Do something else
Read status of DMA module
Interrupt
DMA→CPU
Next instruction
(c) Direct memory access

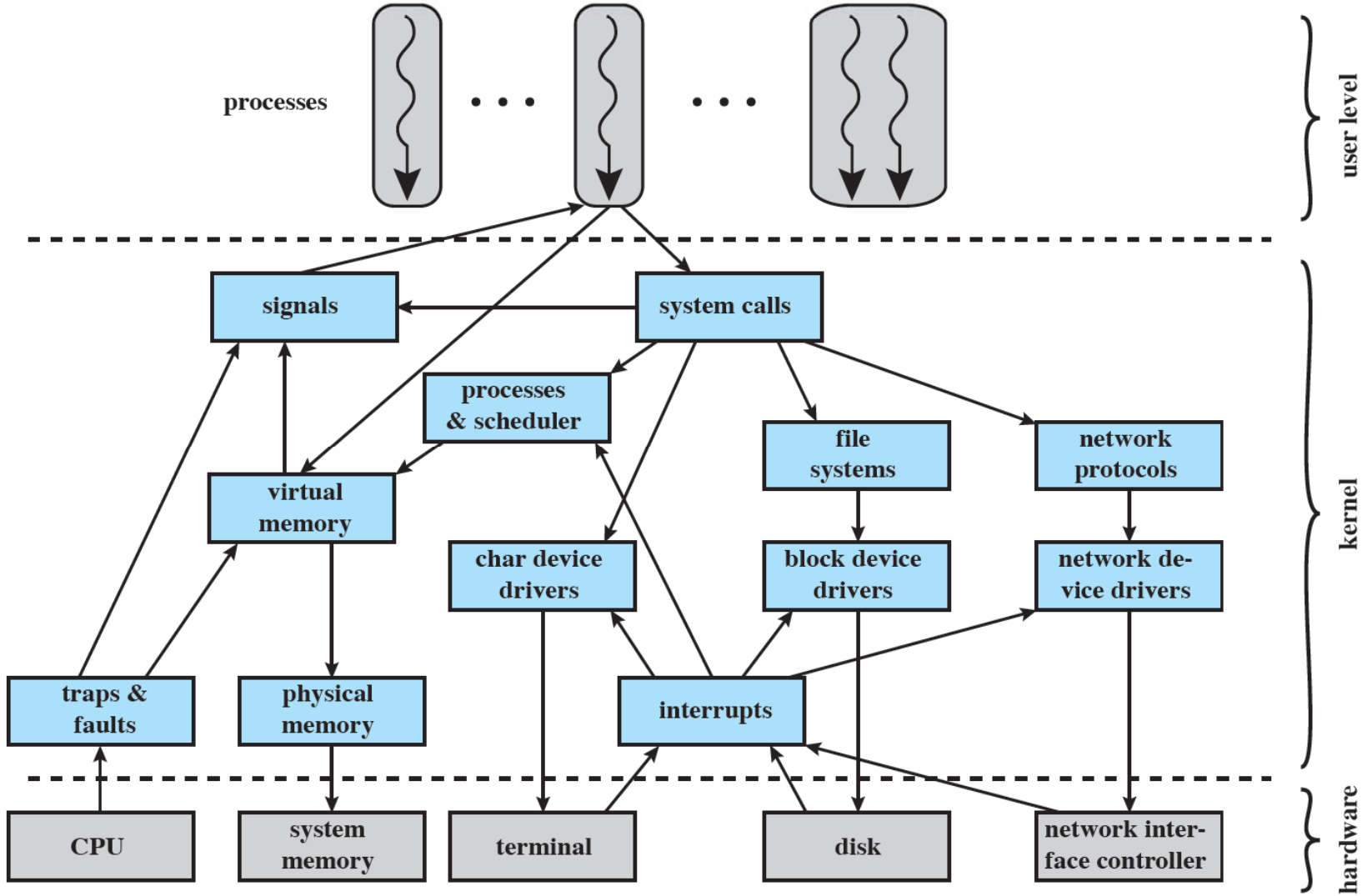


The Operating System as Resource Manager

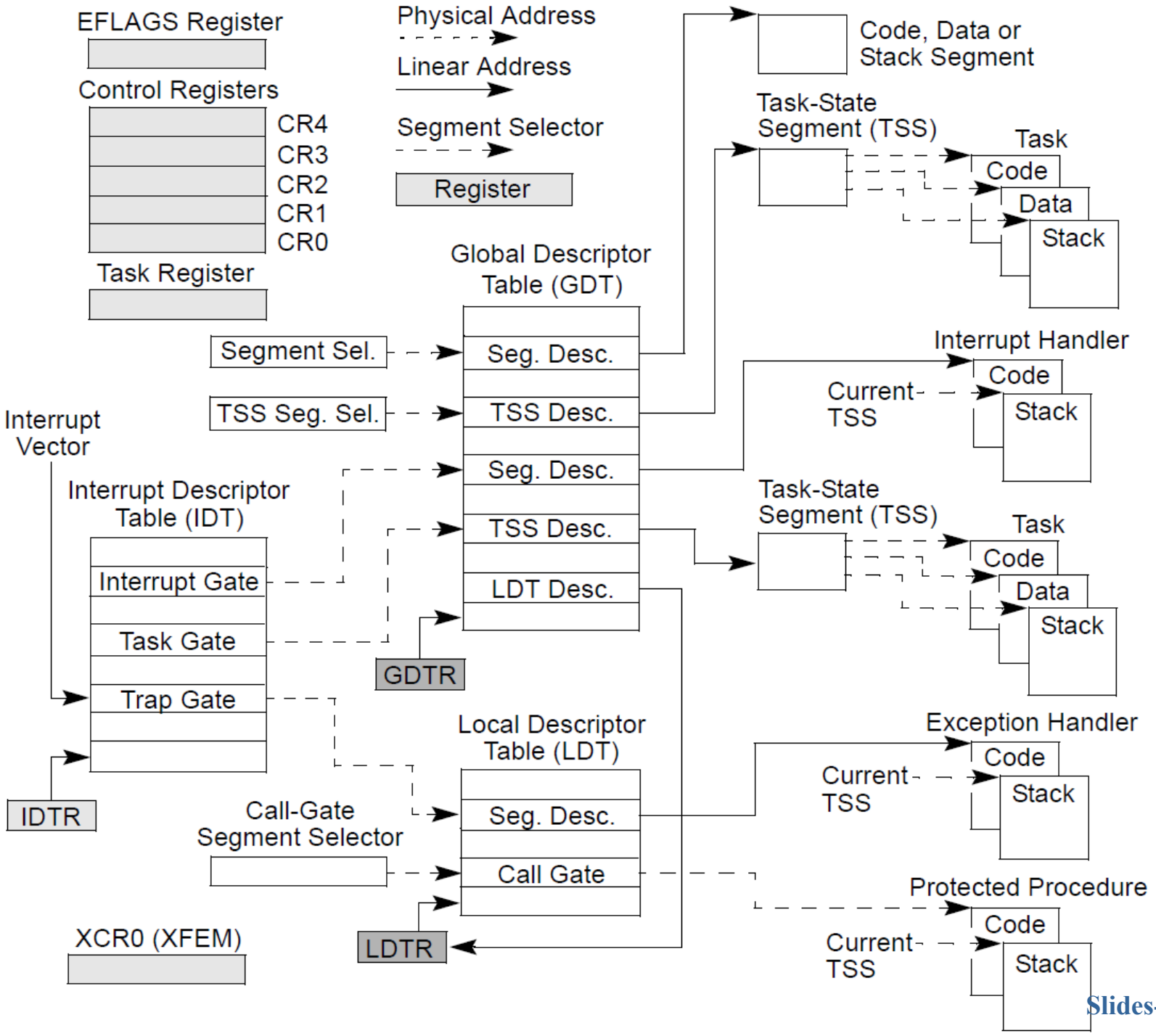
Evolution of Operating Systems



Linux Kernel Components



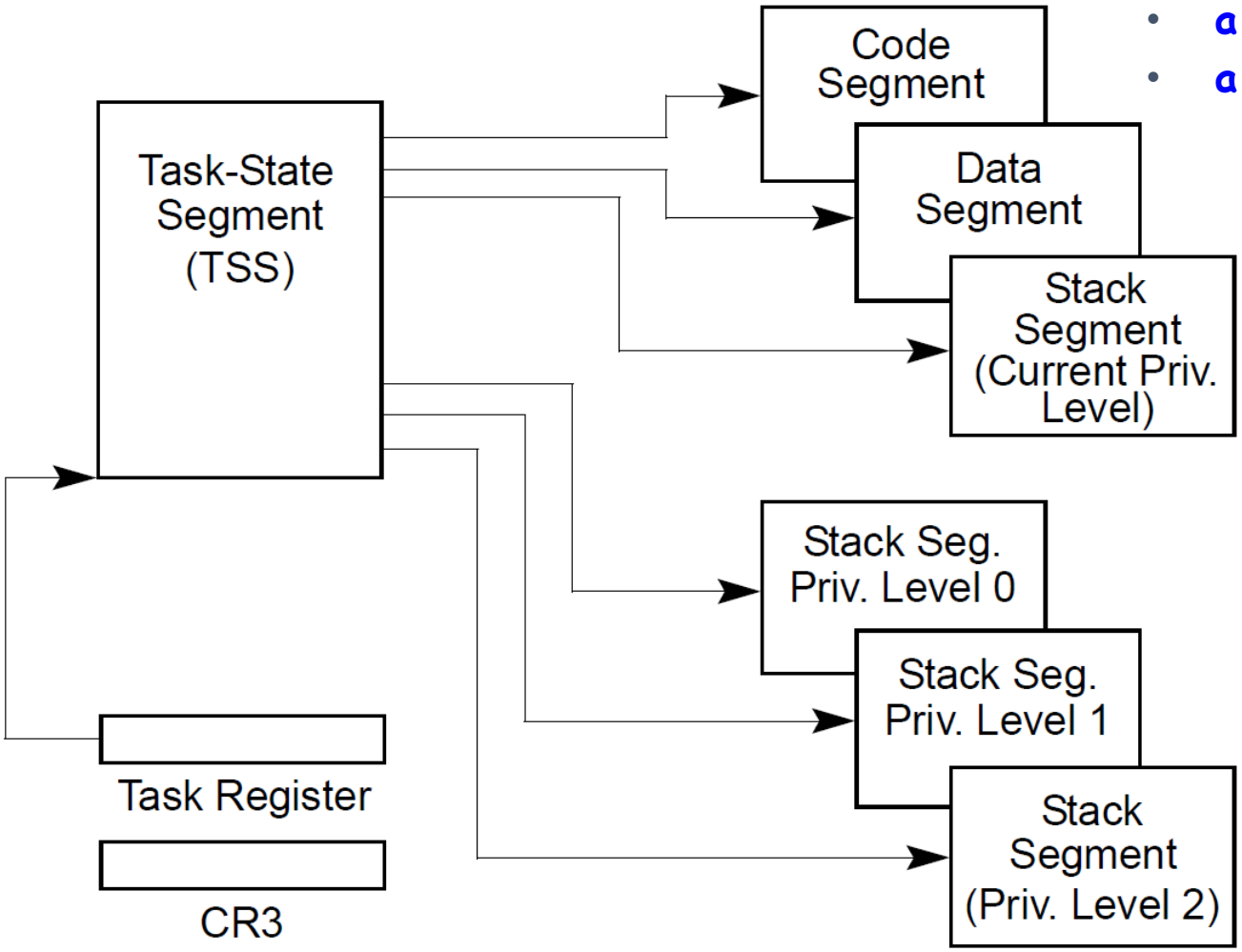
IA-32 System-Level Registers and Data Structures

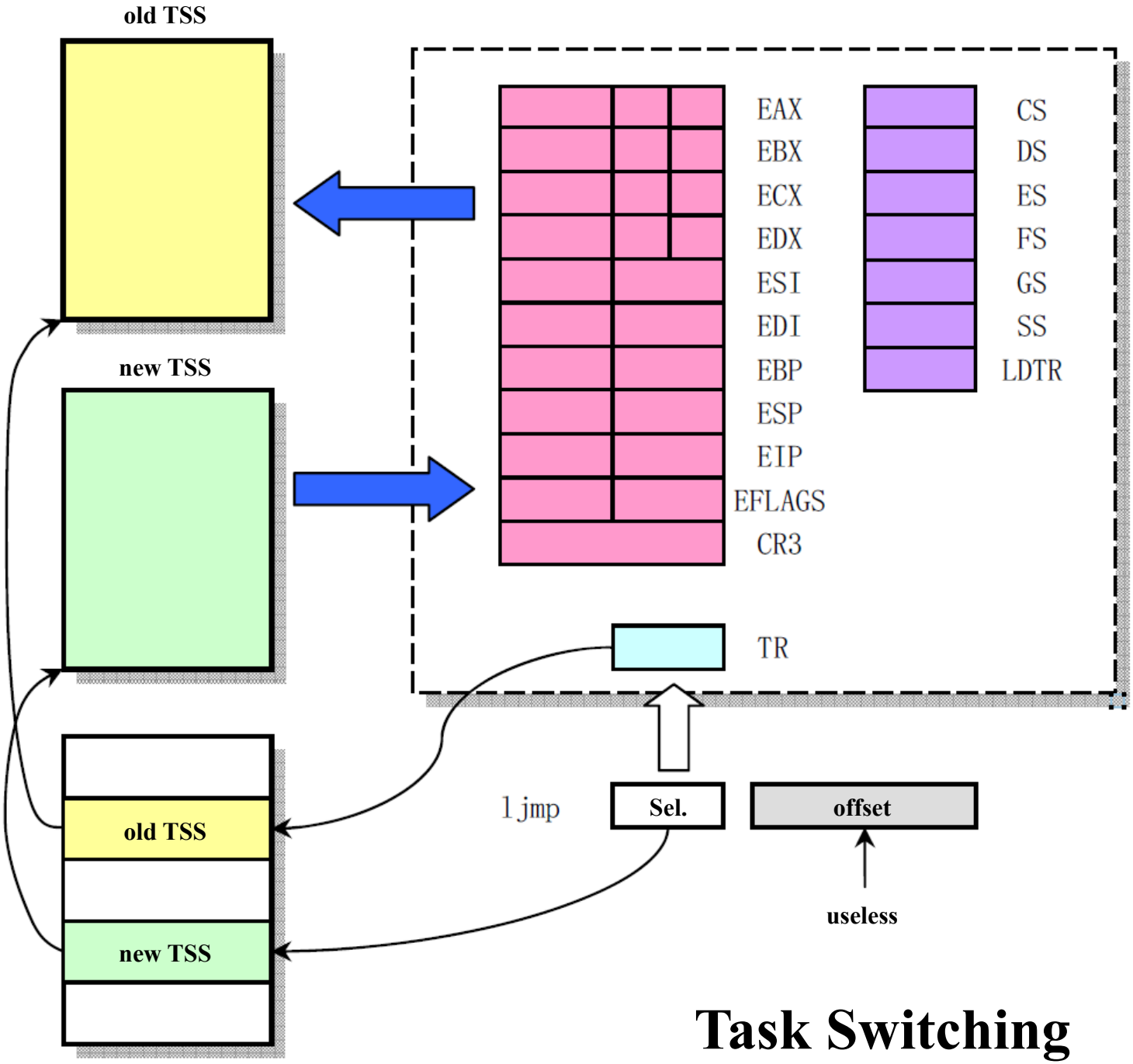


Structure of a Task

A task is made up of two parts:

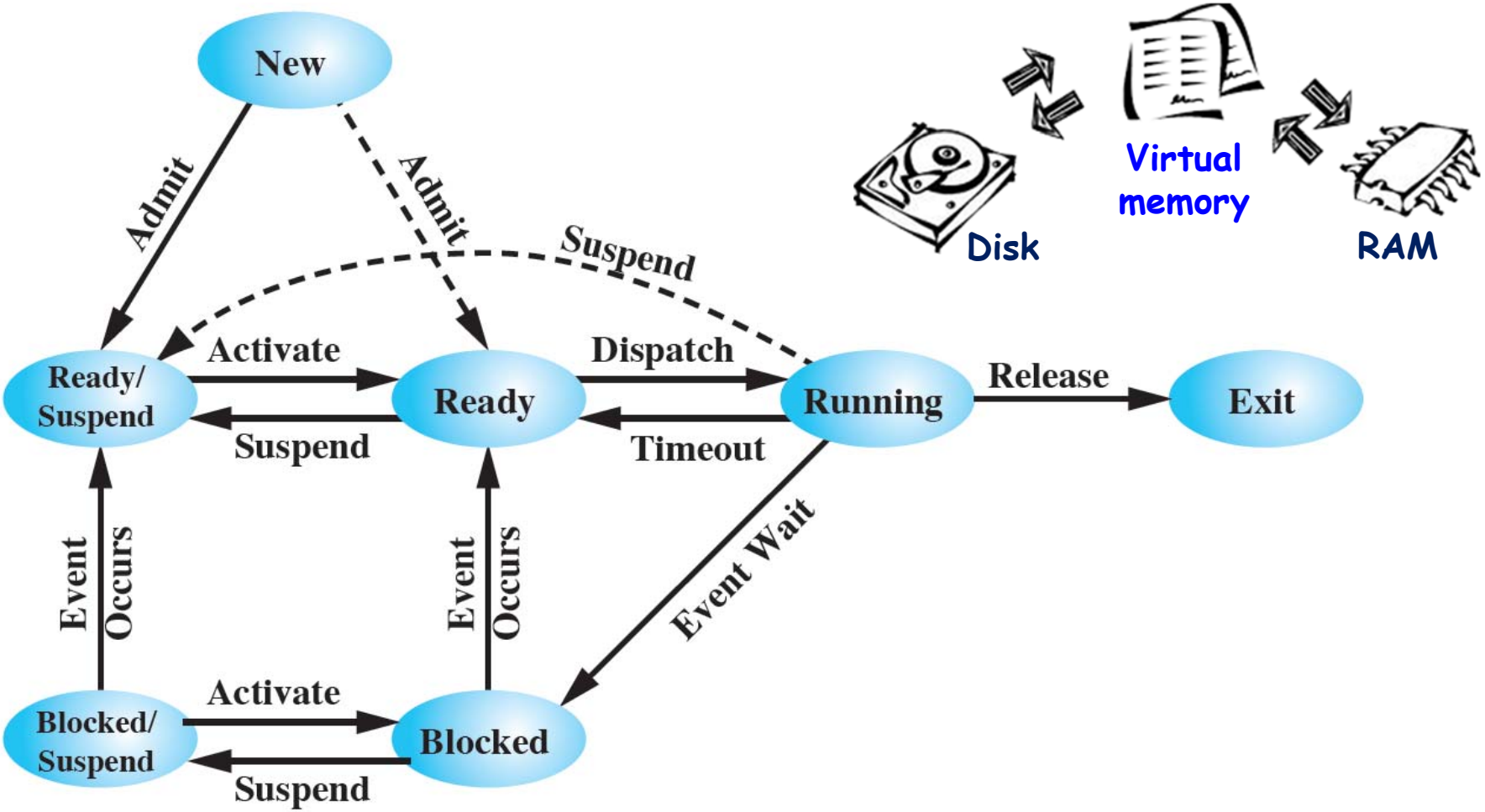
- a task execution space
- a task-state segment (TSS)



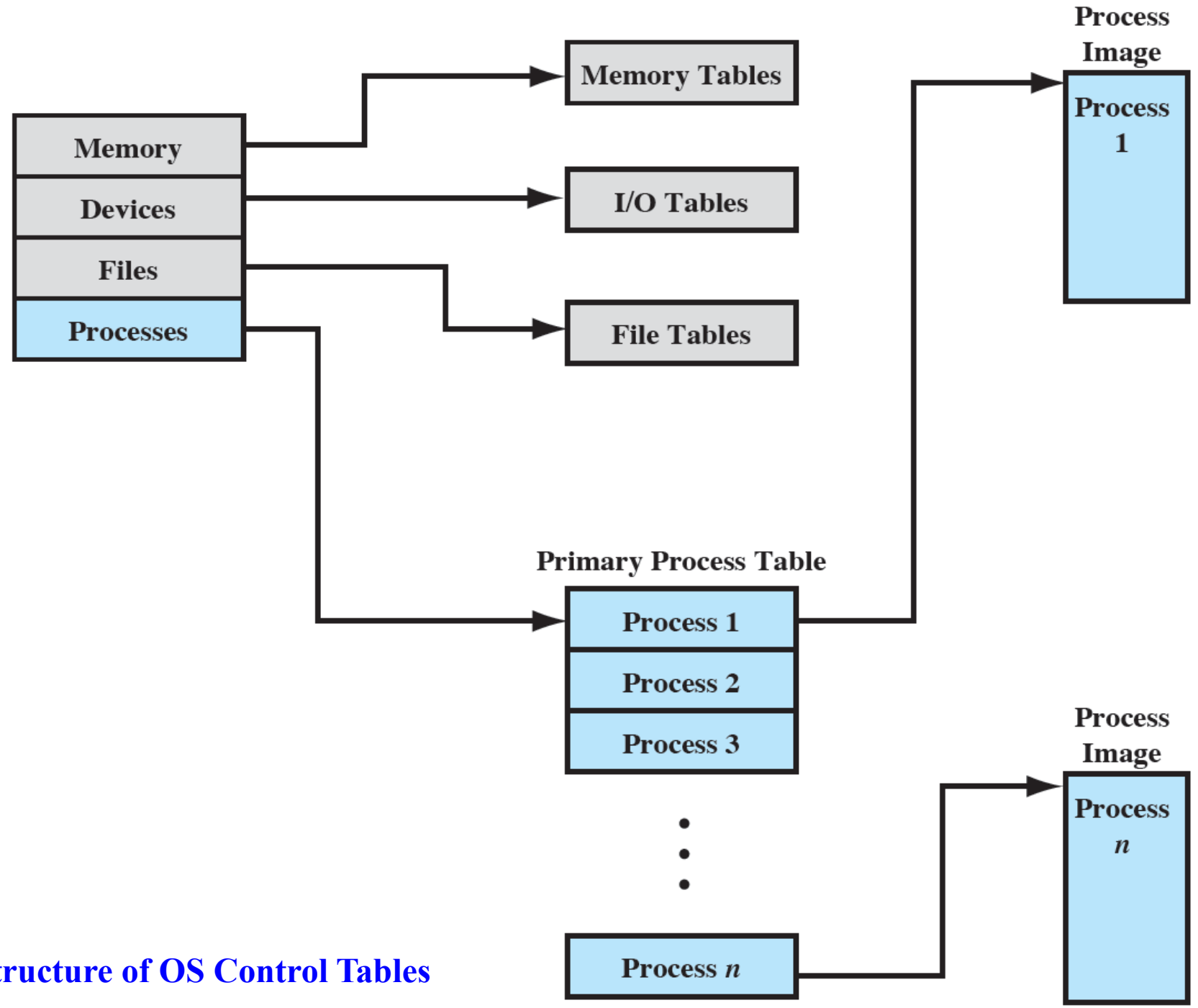


Task Switching

Two Suspend States

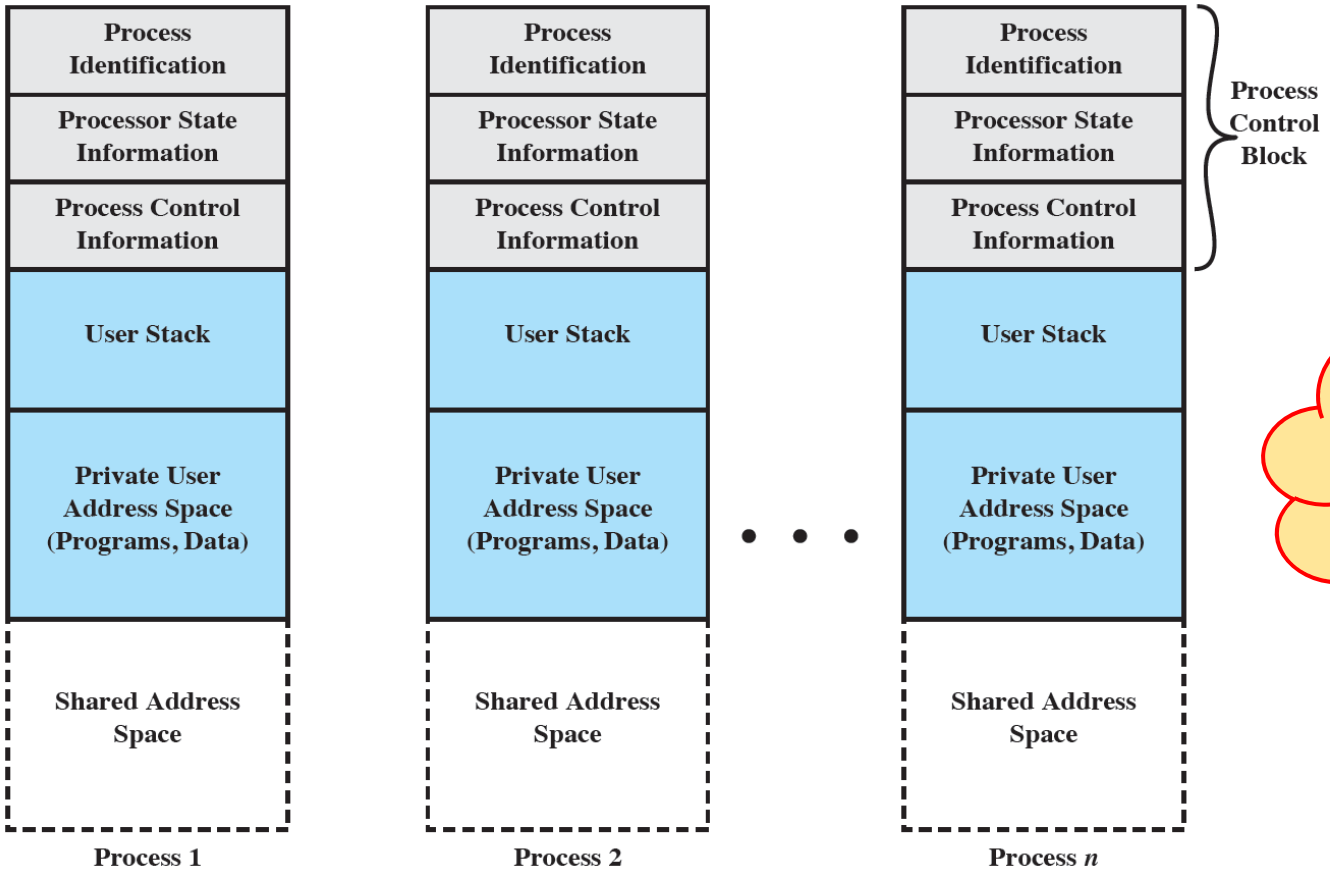


(b) With Two Suspend States



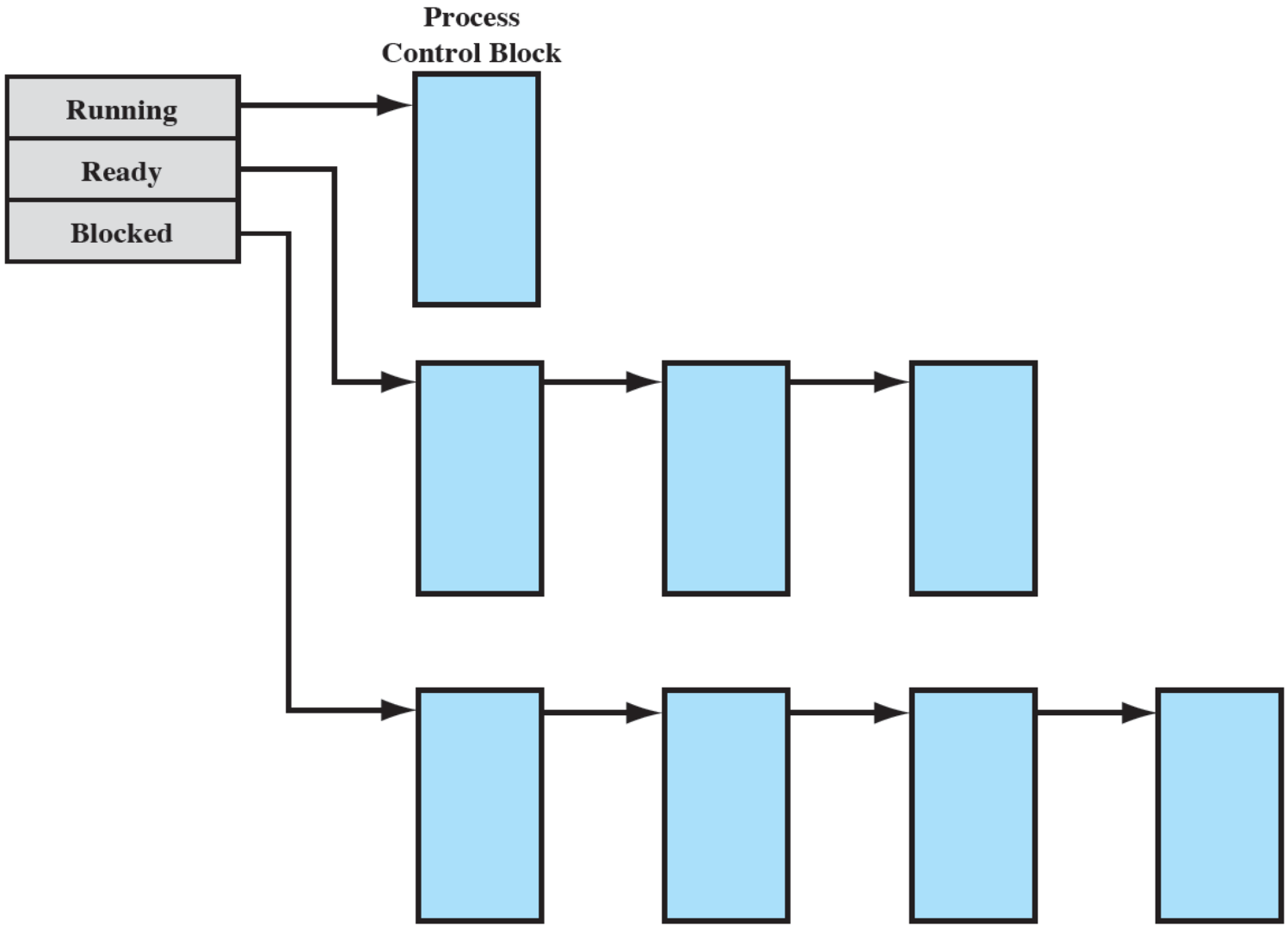
General Structure of OS Control Tables

Structure of Process Images

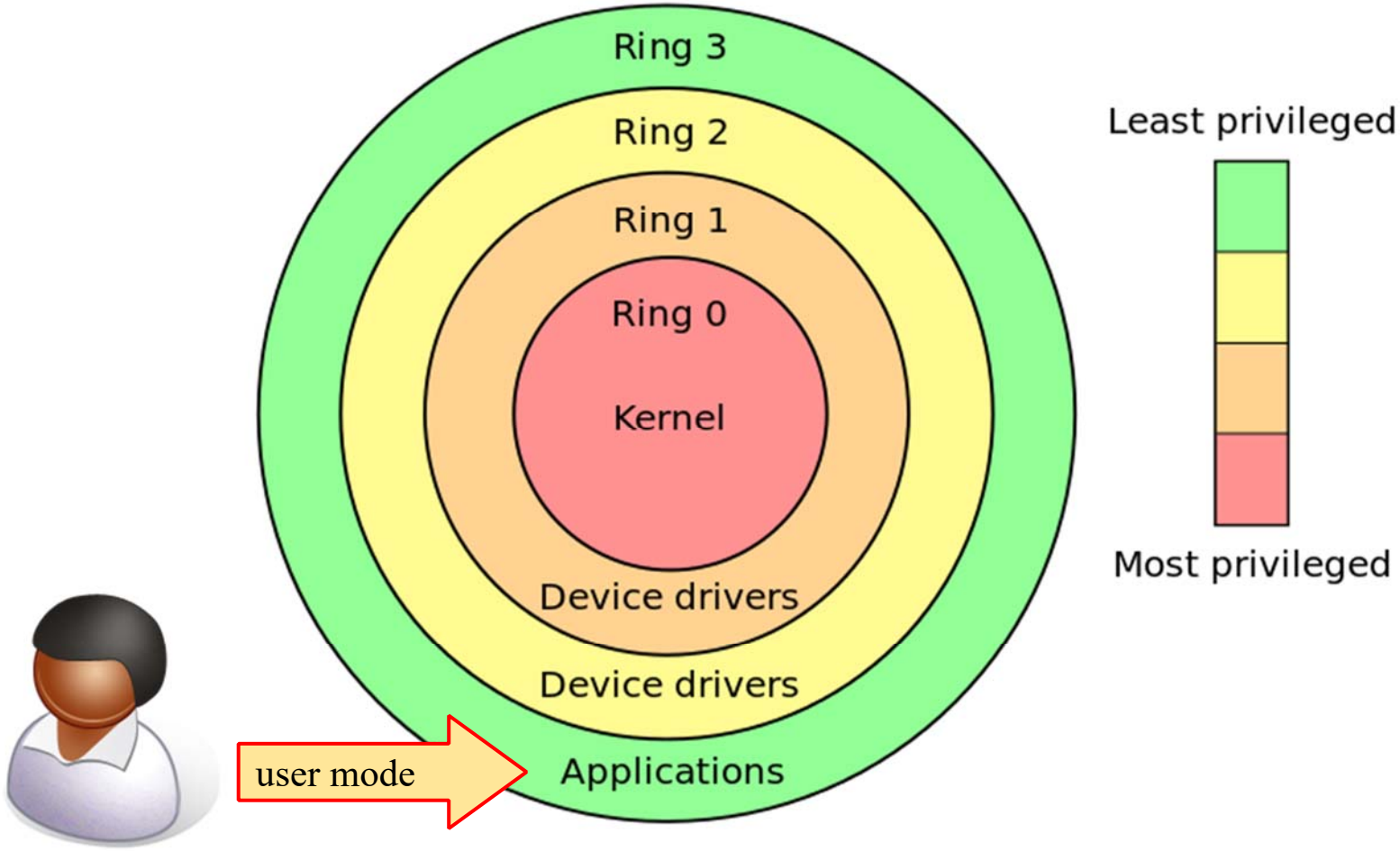


User Processes in Virtual Memory

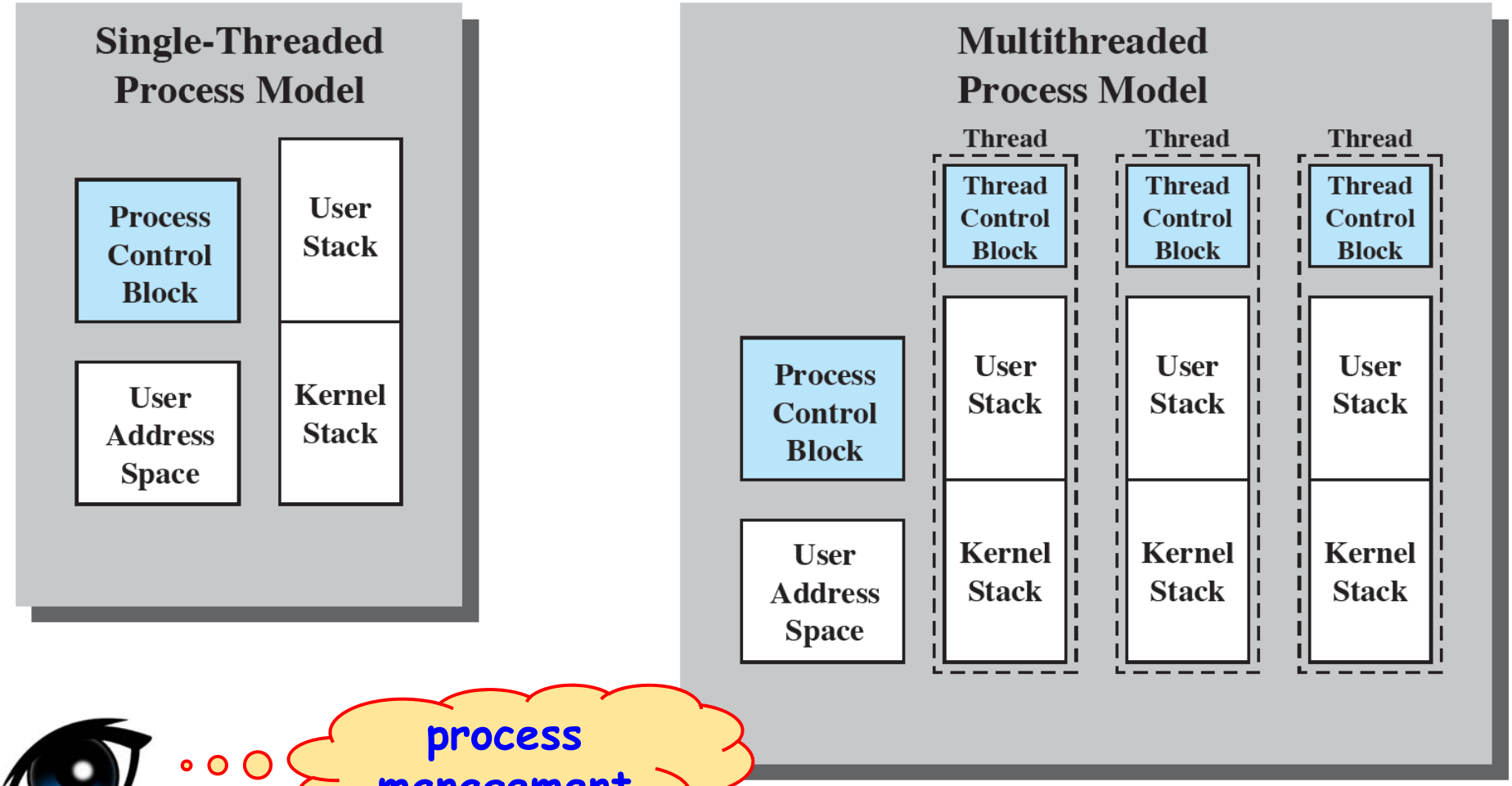
Process List Structures



Modes of Execution

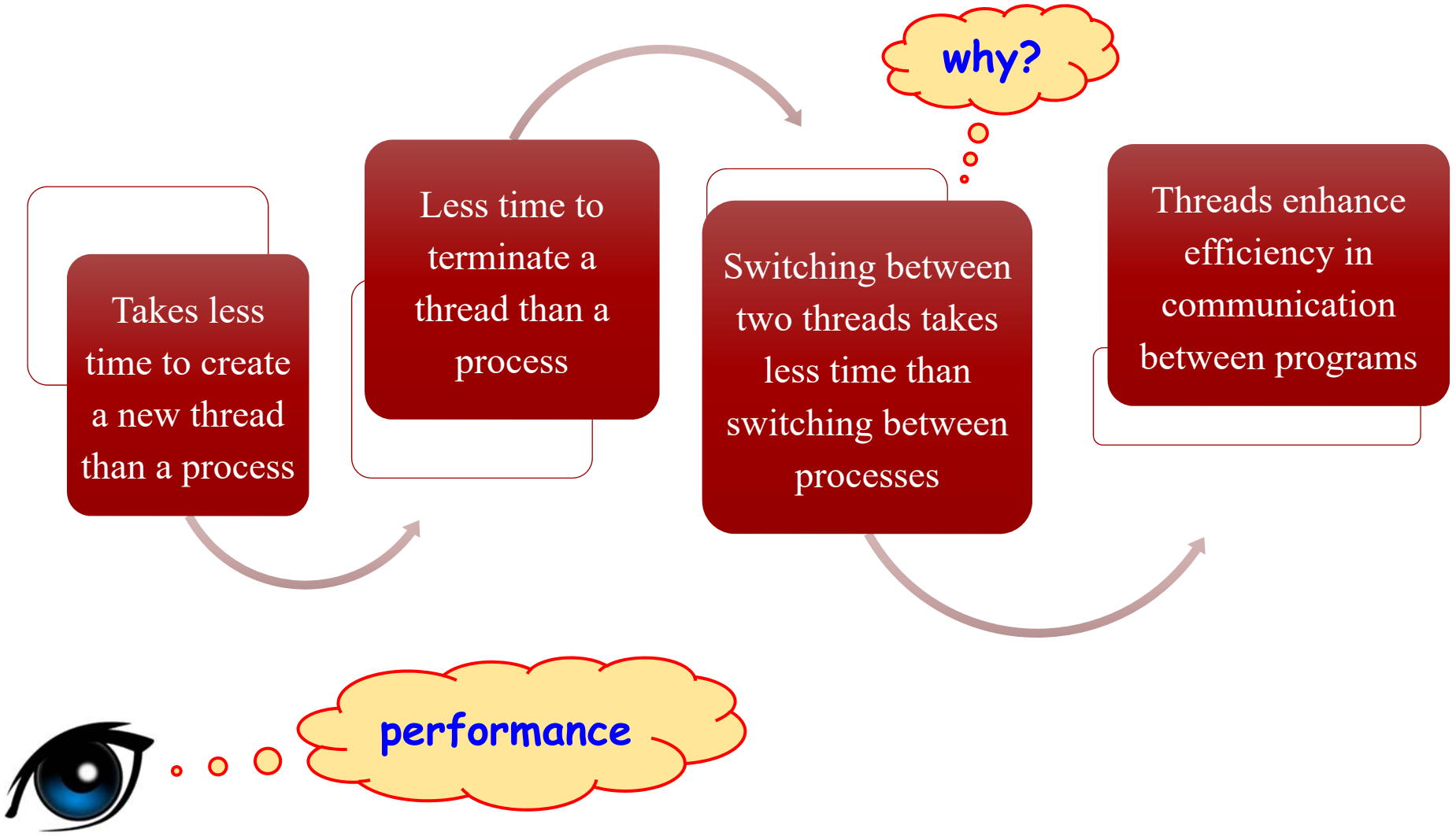


Threads vs. Processes

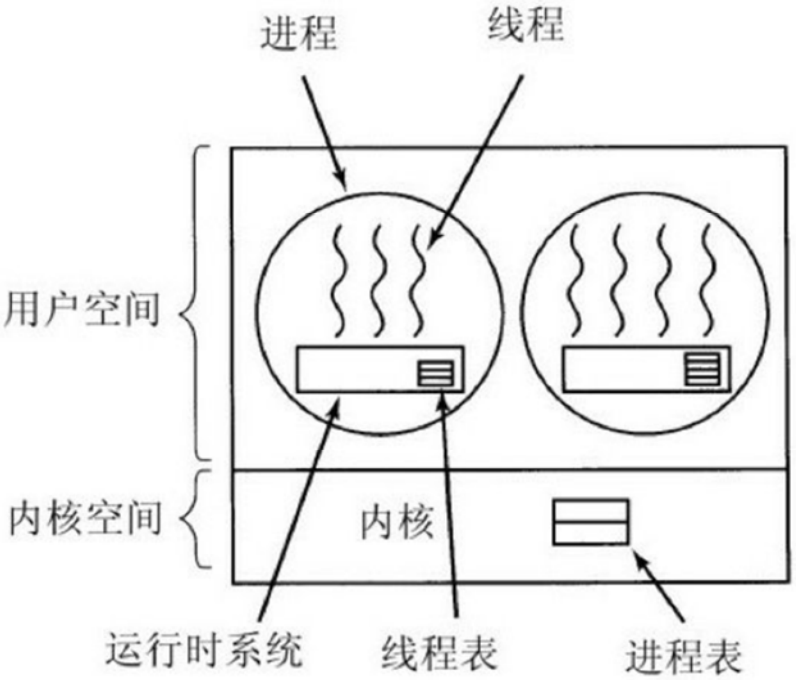


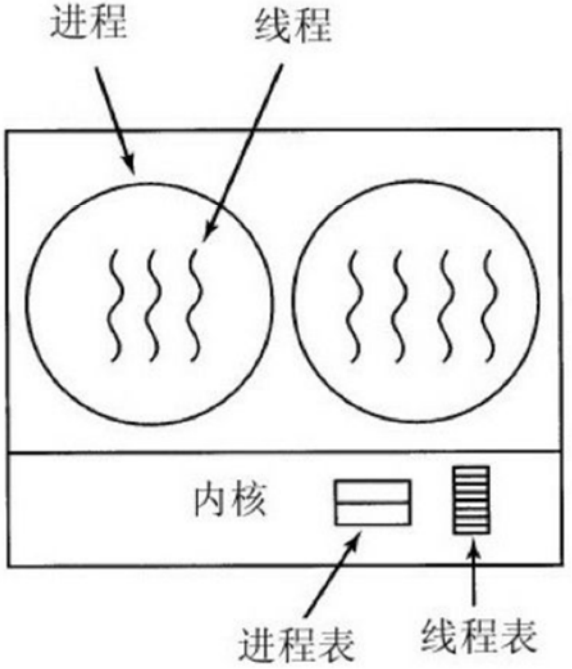
process management

Key Benefits of Threads

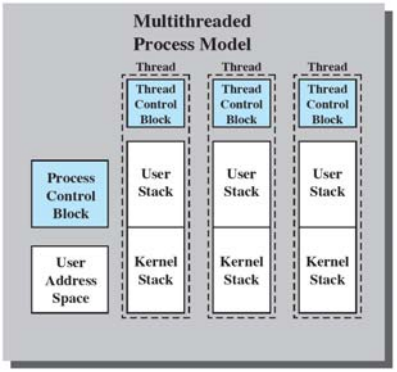


ULT vs. KLT





线程表位置不同



A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

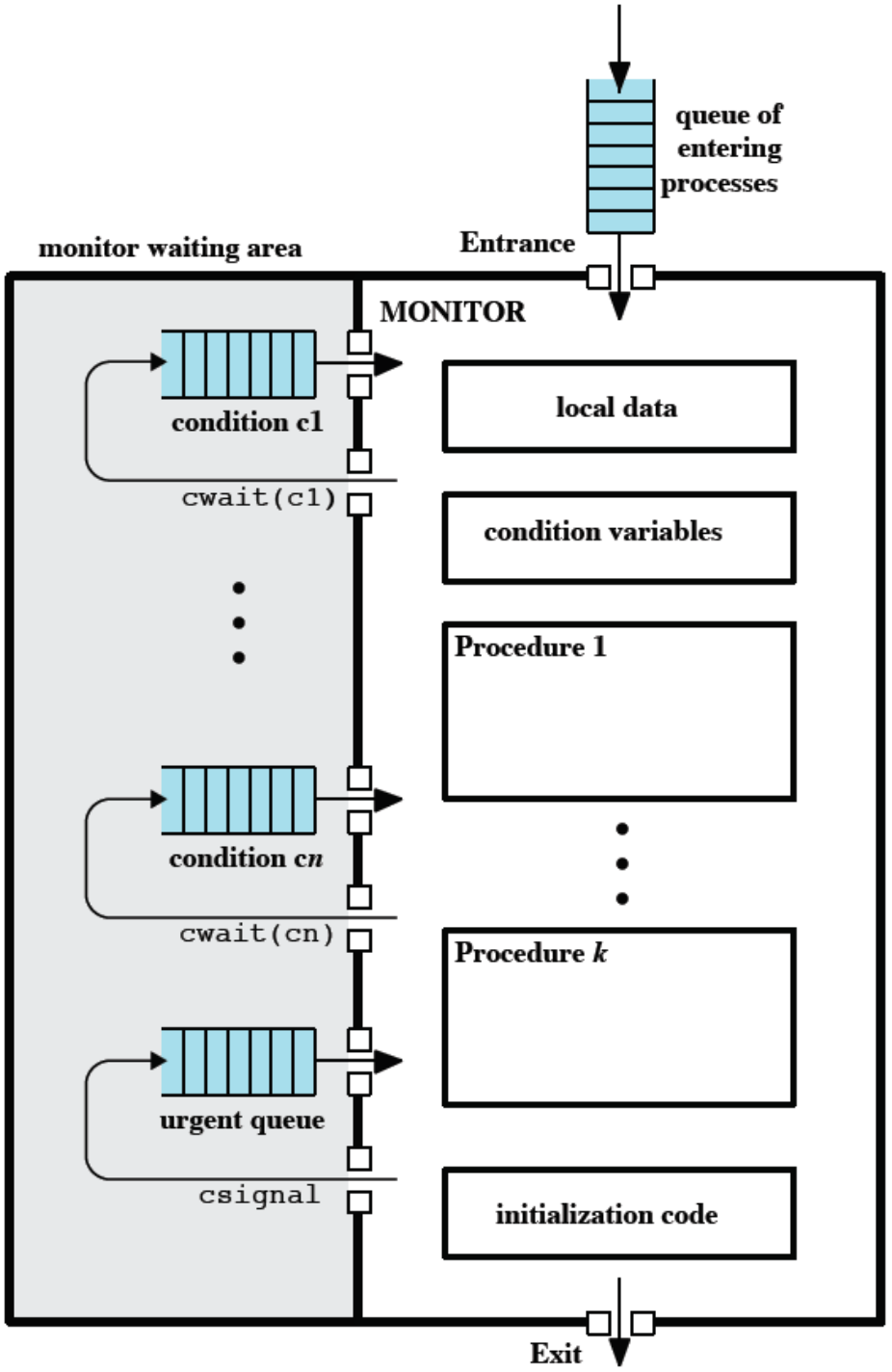
Which one produces a serious flaw ?

信号量总结

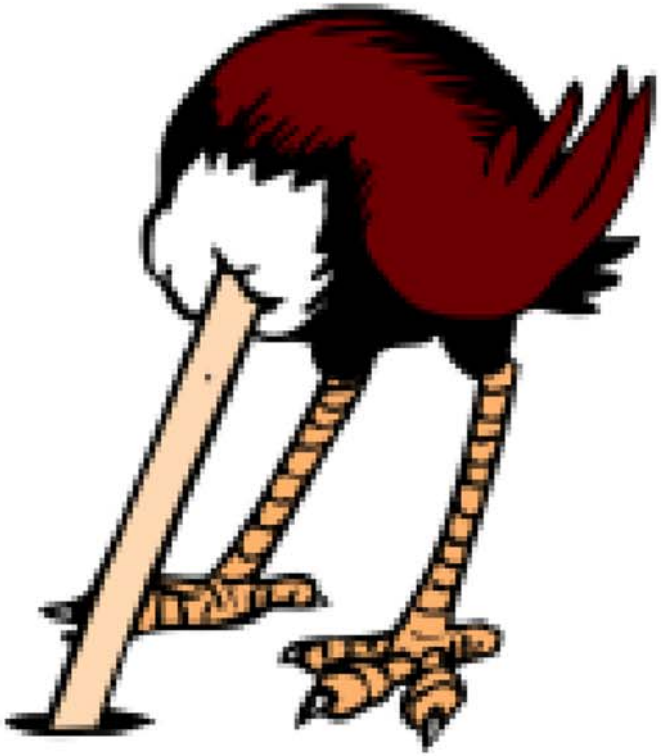
- 信号量总是成对出现的
- 互斥信号量在一个进程内，同步信号量分散在不同的进程
- 同步信号量 **semWait** 操作在互斥信号量之前，**semSignal** 顺序不重要
- 不要把同步信号量加在互斥信号量之间，否则容易死锁

- 要求会写伪代码

Structure of a Monitor



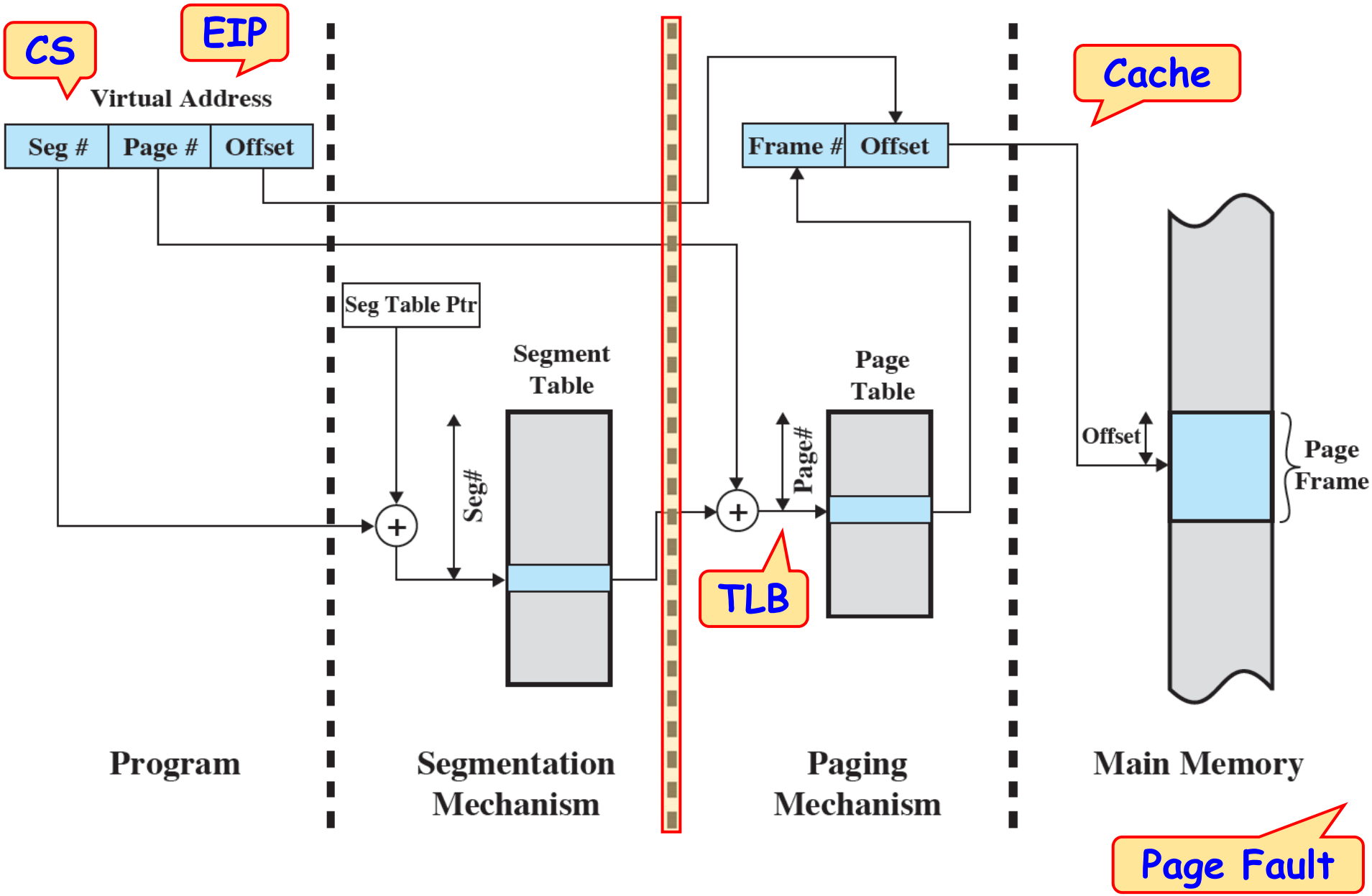
Deadlock



(a) 遇到危险的鸵鸟

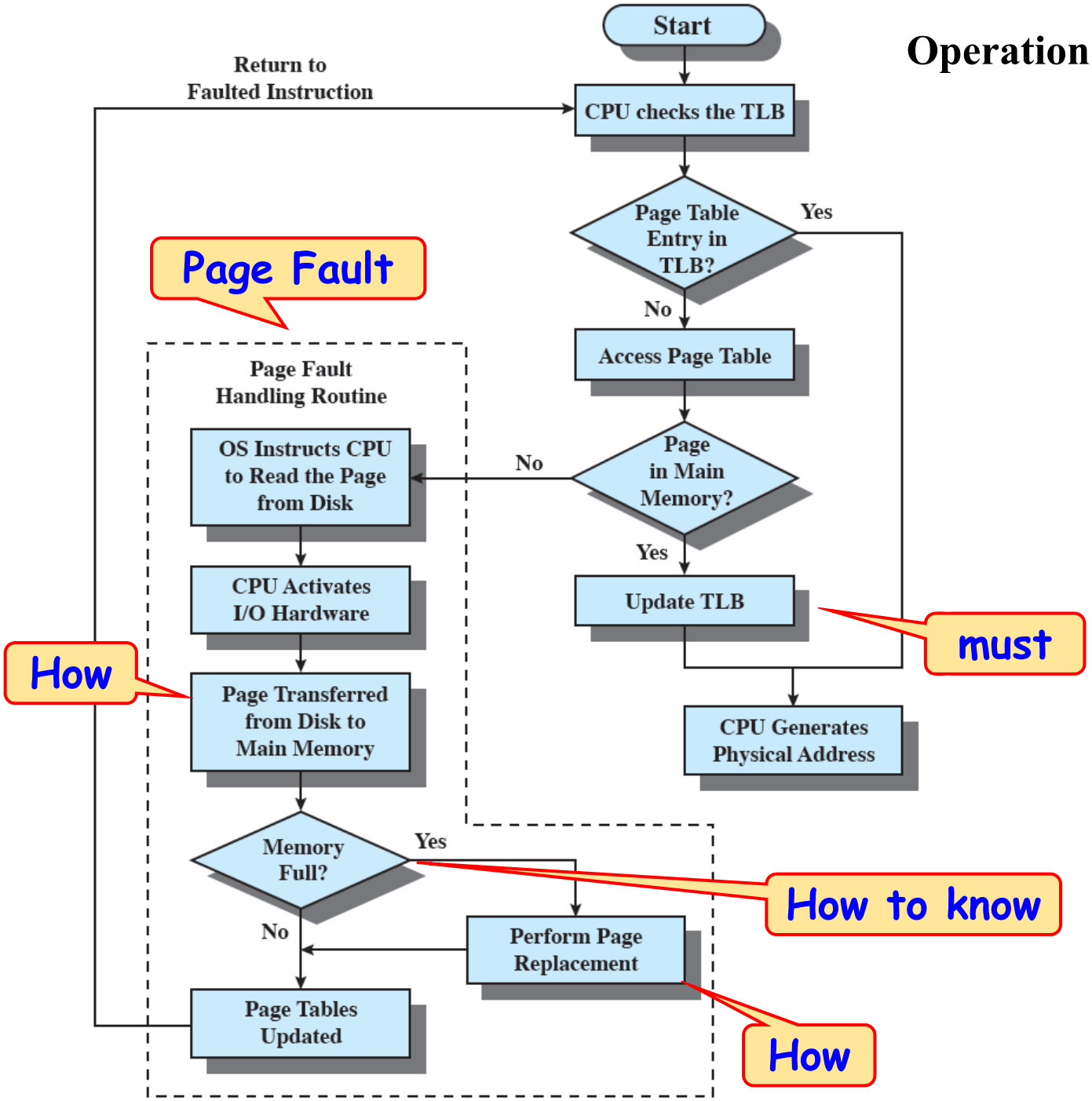
(b) 假装什么都没看见

图 6.1　鸵鸟算法

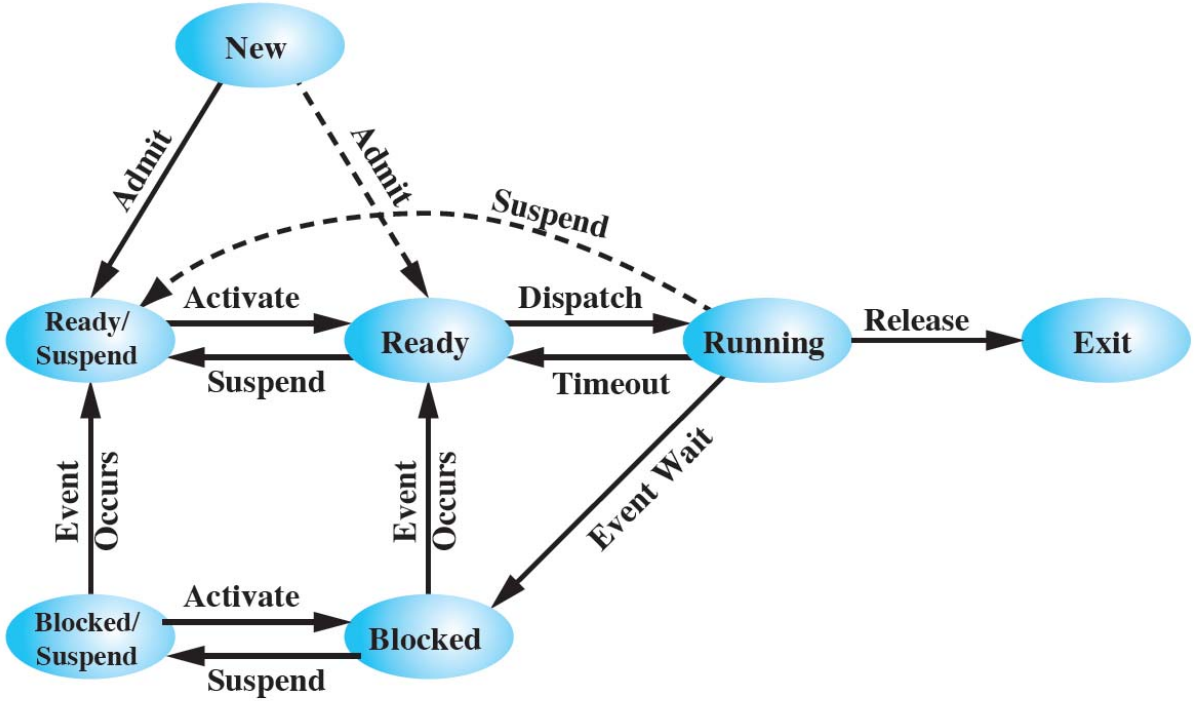


Address Translation in a Segmentation/Paging System

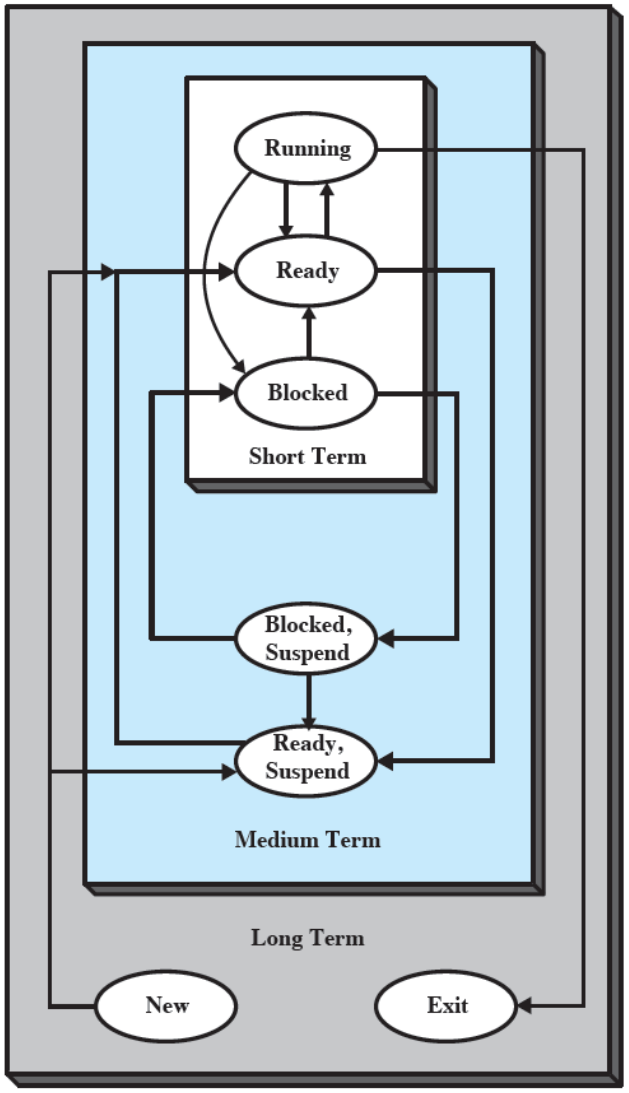
Operation of Paging and TLB



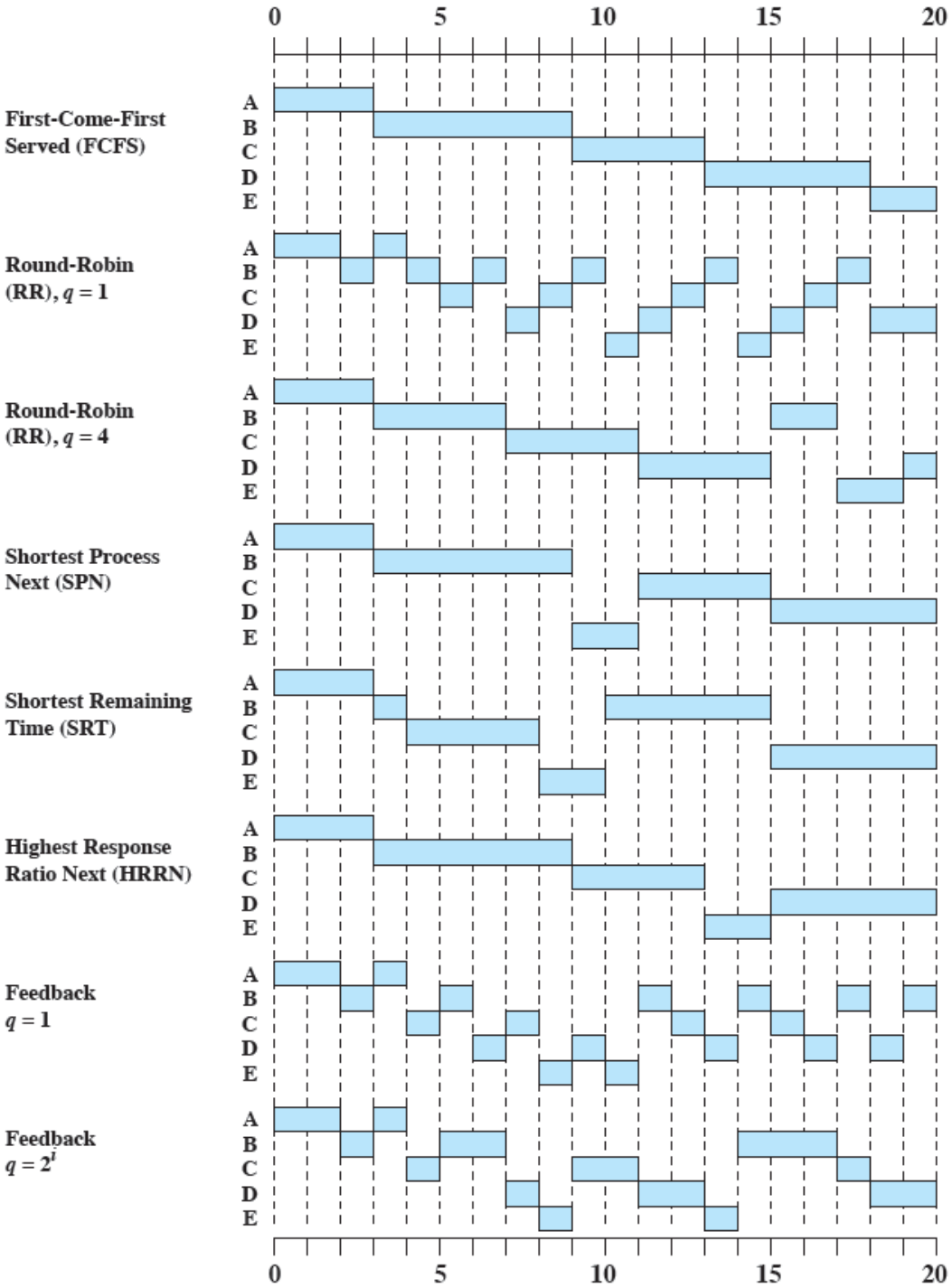
Nesting of Scheduling Functions



(b) With Two Suspend States



A Comparison of Scheduling Policies

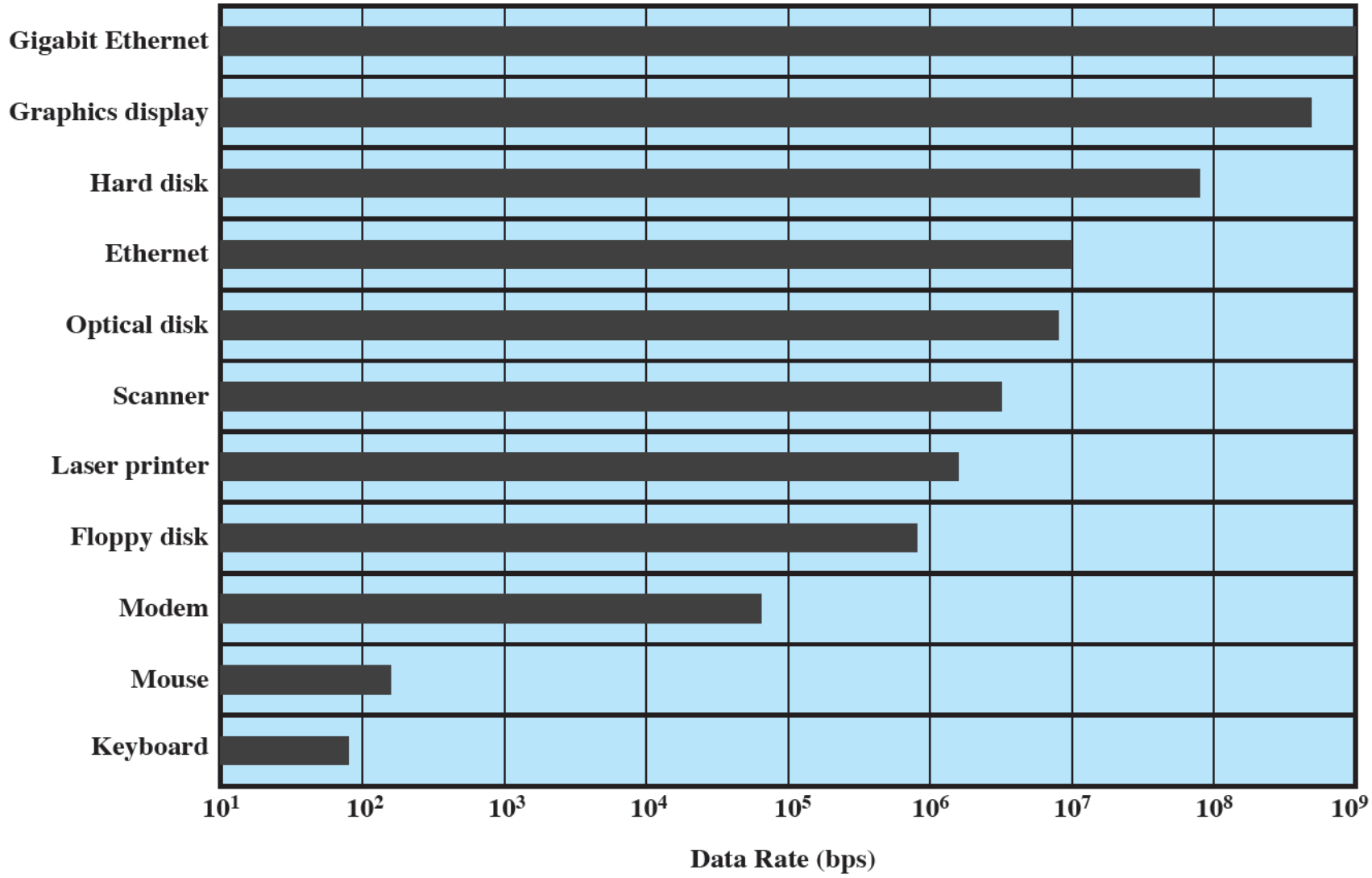


Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

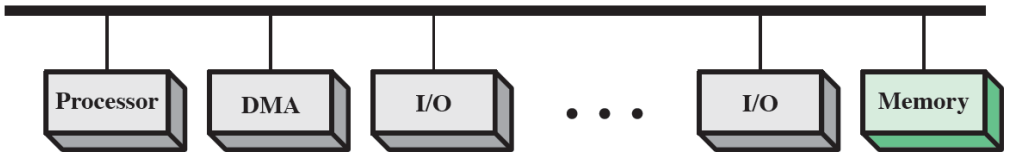
A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (Tr)	3	7	9	12	12	8.60
Tr/Ts	1.00	1.17	2.25	2.40	6.00	2.56
RR q = 1						
Finish Time	4	18	17	20	15	
Turnaround Time (Tr)	4	16	13	14	7	10.80
Tr/Ts	1.33	2.67	3.25	2.80	3.50	2.71
RR q = 4						
Finish Time	3	17	11	20	19	
Turnaround Time (Tr)	3	15	7	14	11	10.00
Tr/Ts	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (Tr)	3	7	11	14	3	7.60
Tr/Ts	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (Tr)	3	13	4	14	2	7.20
Tr/Ts	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (Tr)	3	7	9	14	7	8.00
Tr/Ts	1.00	1.17	2.25	2.80	3.5	2.14
FB q = 1						
Finish Time	4	20	16	19	11	
Turnaround Time (Tr)	4	18	12	13	3	10.00
Tr/Ts	1.33	3.00	3.00	2.60	1.5	2.29

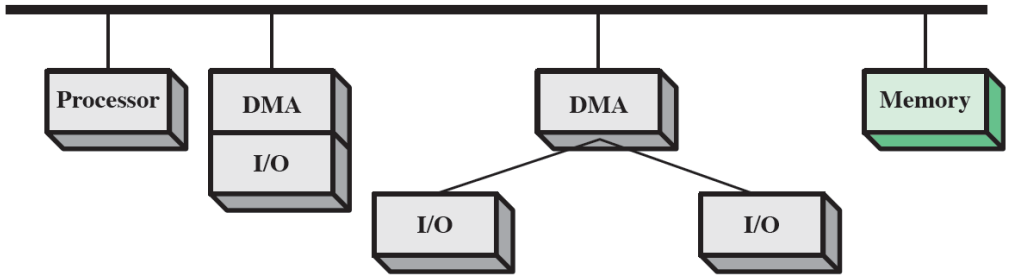
Typical I/O Device Data Rates



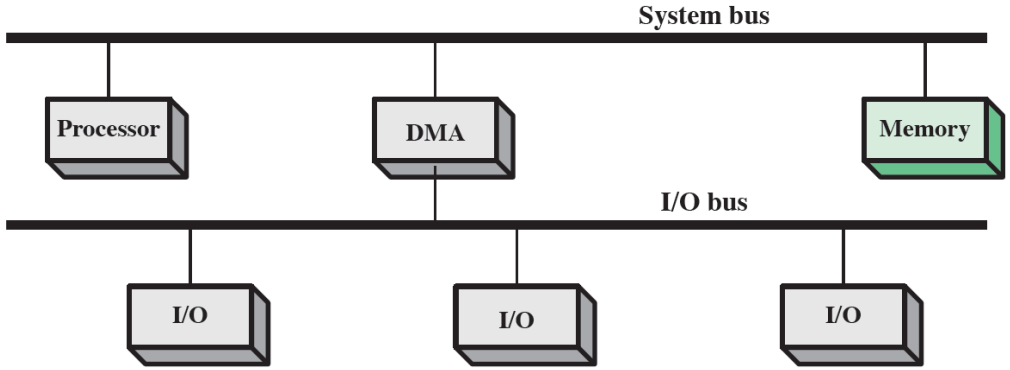
Alternative DMA Configurations



(a) Single-bus, detached DMA

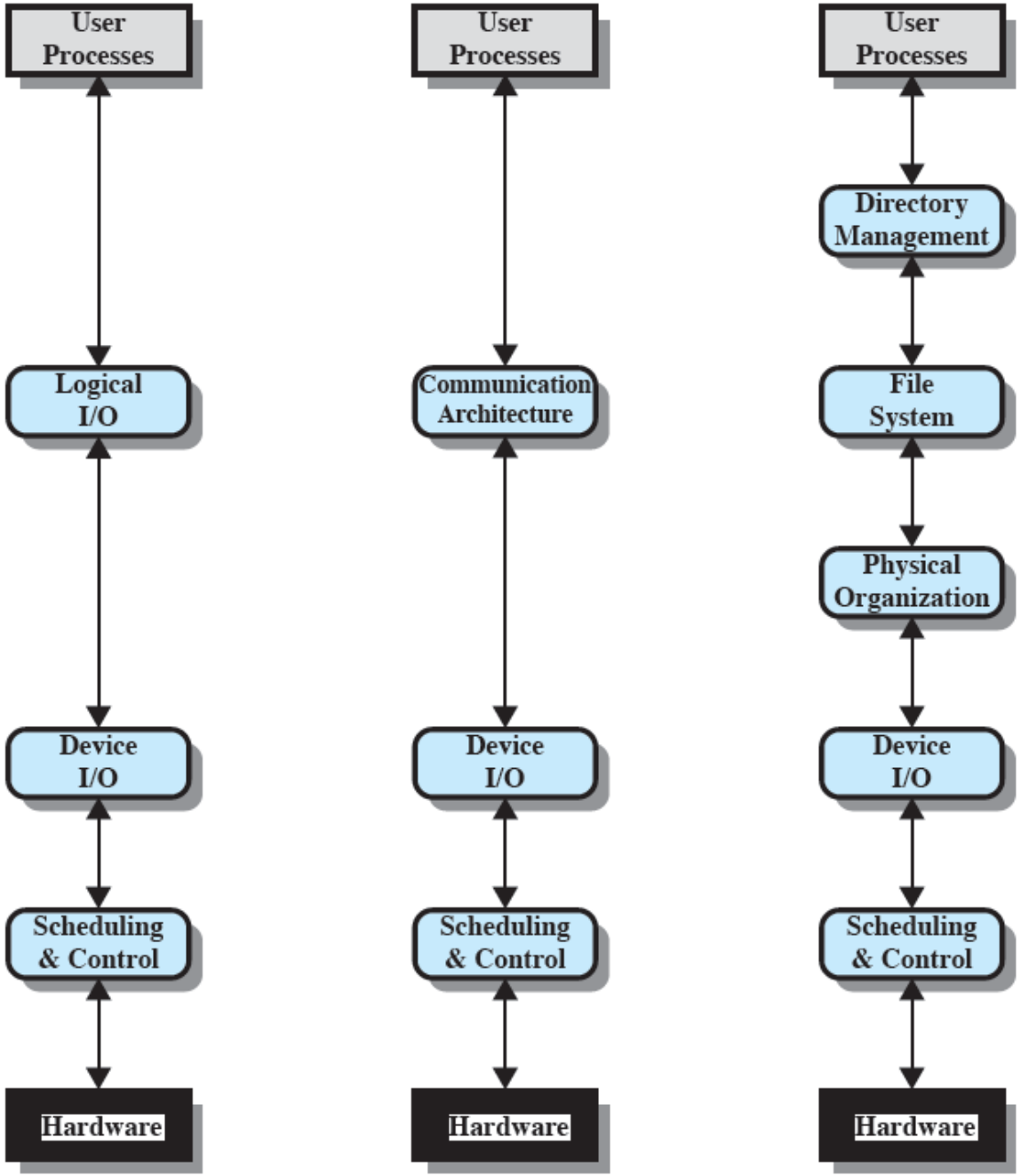


(b) Single-bus, Integrated DMA-I/O



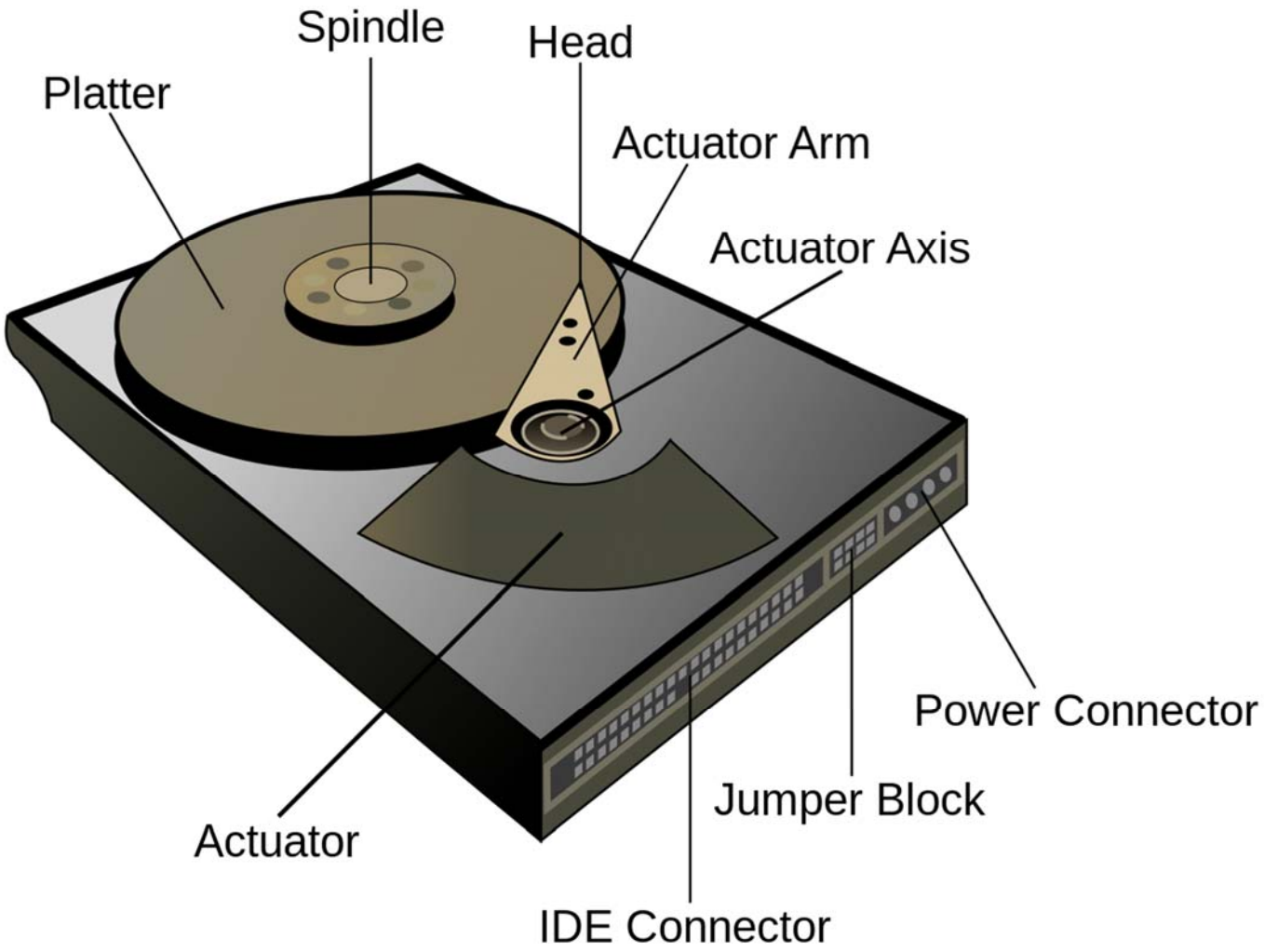
(c) I/O bus

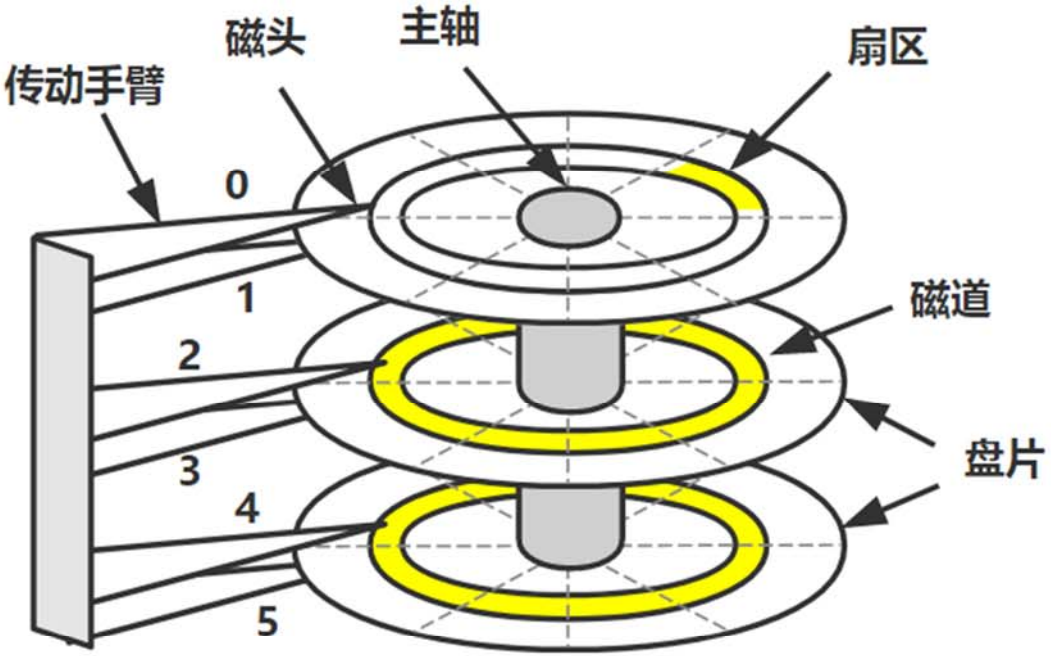
A Model of I/O Organization

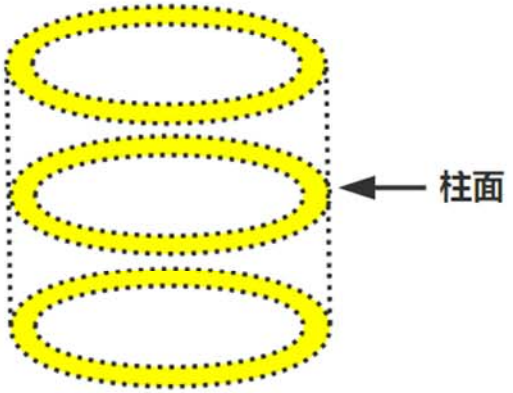


(a) Local peripheral device (b) Communications port (c) File system

Diagram labeling the major components of HDD







9.3.3.4 扇区号与柱面号、当前磁道扇区号和当前磁头号的对应关系

假定硬盘的每磁道扇区数是 track_secs，硬盘磁头总数是 dev_heads，磁道总数是 tracks。对于指定的硬盘顺序扇区号是 sector，对应的当前柱面号是 cyl，在当前磁道上的扇区号是 sec，当前磁头号是 head。那么若想从指定顺序扇区号 sector 换算成对应的当前柱面号、当前磁道上扇区号以及当前磁头号，则可以使用以下步骤：

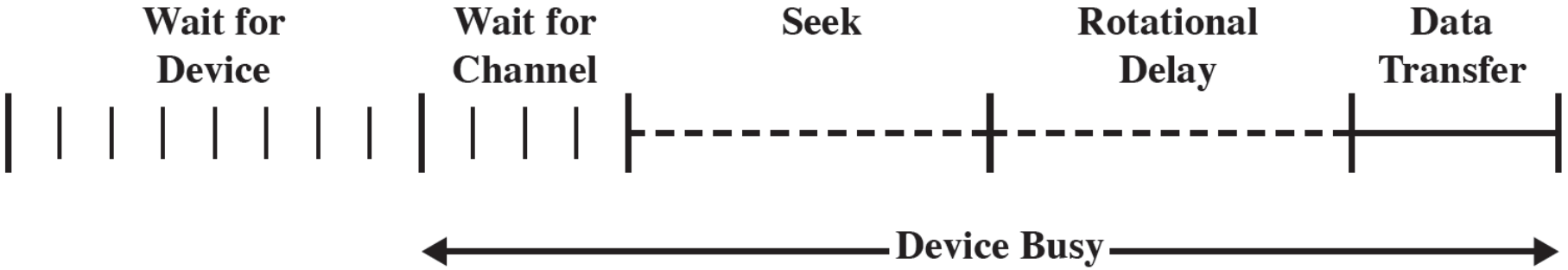
- sector / track_secs = 整数是 tracks，余数是 sec；
- tracks / dev_heads = 整数是 cyl，余数是 head；
- 在当前磁道上扇区号从 1 算起，于是需要把 sec 增 1。

若想从指定的当前 cyl、sec 和 head 换算成从硬盘开始算起的顺序扇区号，则过程正好与上述相反。换算公式和上面给出的完全一样，即：

```
sector = (cyl * dev_heads + head) * track_secs + sec -1
```

Disk Performance Parameters

- **The actual details of disk I/O operation depend on**
  - computer system
  - operating system
  - nature of the I/O channel and disk controller hardware



Timing of a Disk I/O Transfer

T = transfer time,
b = number of bytes to be transferred,
N = number of bytes on a track, and
r = rotation speed, in revolutions per second.

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

举例：一个典型的磁盘

- 平均寻道时间 **4 ms**
- 转速 **7500 rpm**
- 每个磁道有 **500** 个扇区、每个扇区有 **512** 个字节
- 假设读取一个包含**2500** 个扇区，大小**1.28M**的文件
- 估计传送需要的总时间

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

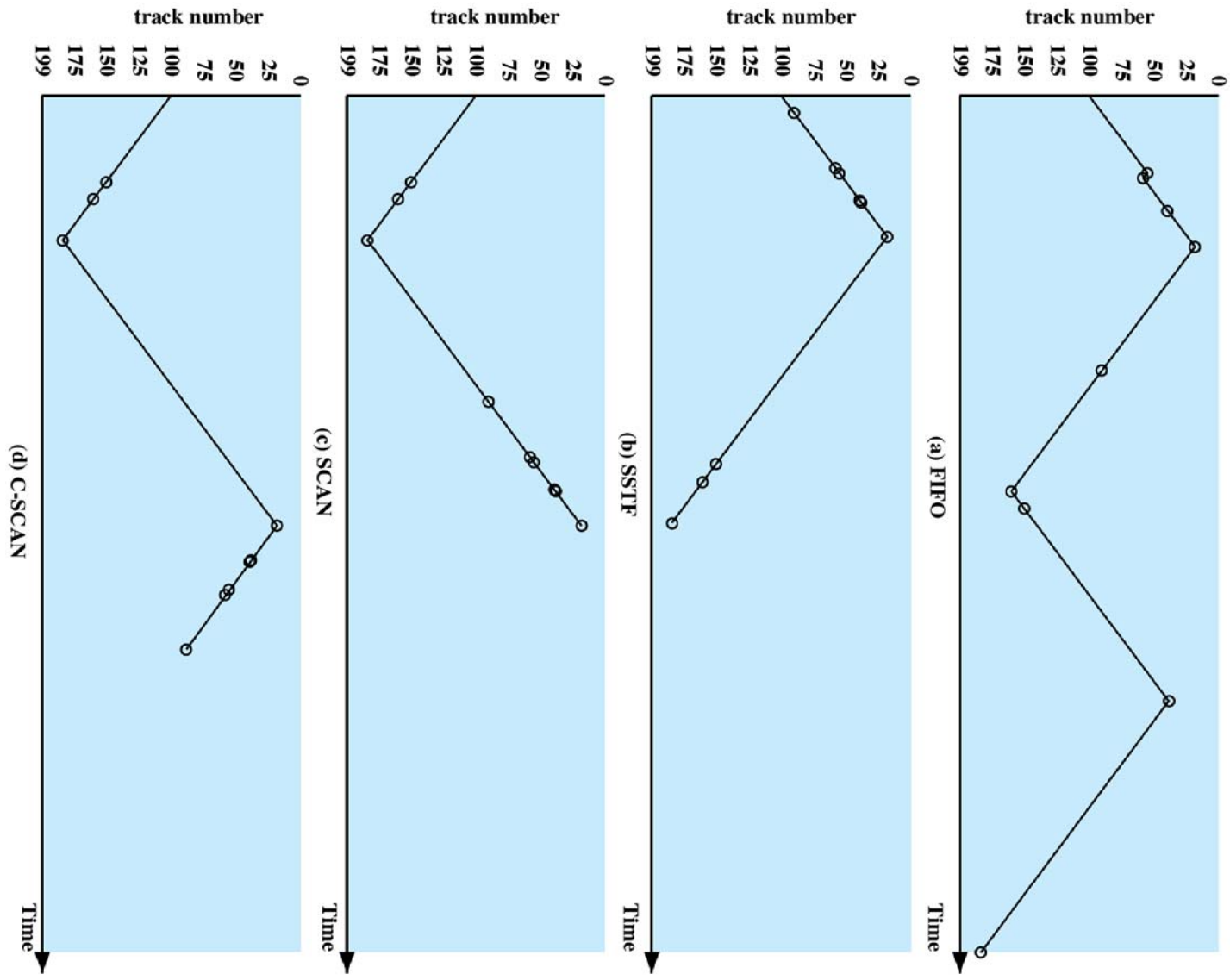
Average seek	4 ms
Rotational delay	4 ms
Read 500 sectors	8 ms
	16 ms

Total time = 16 + (4 × 12) = 64 ms = 0.064 seconds

Average seek	4 ms
Rotational delay	4 ms
Read 1 sector	0.016 ms
	8.016 ms

Total time = 2,500 × 8.016 = 20,040 ms = 20.04 seconds

Comparison of Disk Scheduling Algorithms



Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	**45**	**90**	**10**	**150**	**50**	**150**	**50**
58	**3**	**58**	**32**	**160**	**10**	**160**	**10**
39	**19**	**55**	**3**	**184**	**24**	**184**	**24**
18	**21**	**39**	**16**	**90**	**94**	**18**	**166**
90	**72**	**38**	**1**	**58**	**32**	**38**	**20**
160	**70**	**18**	**20**	**55**	**3**	**39**	**1**
150	**10**	**150**	**132**	**39**	**16**	**55**	**16**
38	**112**	**160**	**10**	**38**	**1**	**58**	**3**
184	**146**	**184**	**24**	**18**	**20**	**90**	**32**
Average seek length	**55.3**	Average seek length	**27.5**	Average seek length	**27.8**	Average seek length	**35.8**

硬盘分区表

表 9-10 硬盘主引导扇区 MBR 的结构

偏移位置	名称	长度（字节）	说明
0x000	MBR 代码	446	引导程序代码和数据。
0x1BE	分区表项 1	16	第 1 个分区表项，共 16 字节。
0x1CE	分区表项 2	16	第 2 个分区表项，共 16 字节。
0x1DE	分区表项 3	16	第 3 个分区表项，共 16 字节。
0x1EE	分区表项 4	16	第 4 个分区表项，共 16 字节。
0x1FE	引导标志	2	有效引导扇区的标志，值分别是 0x55, 0xAA。

主引导扇区

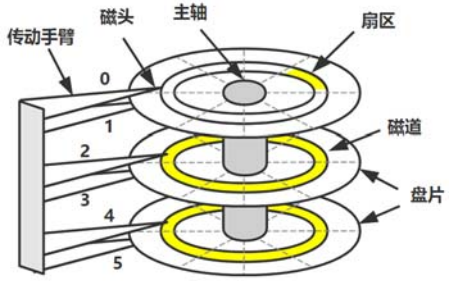
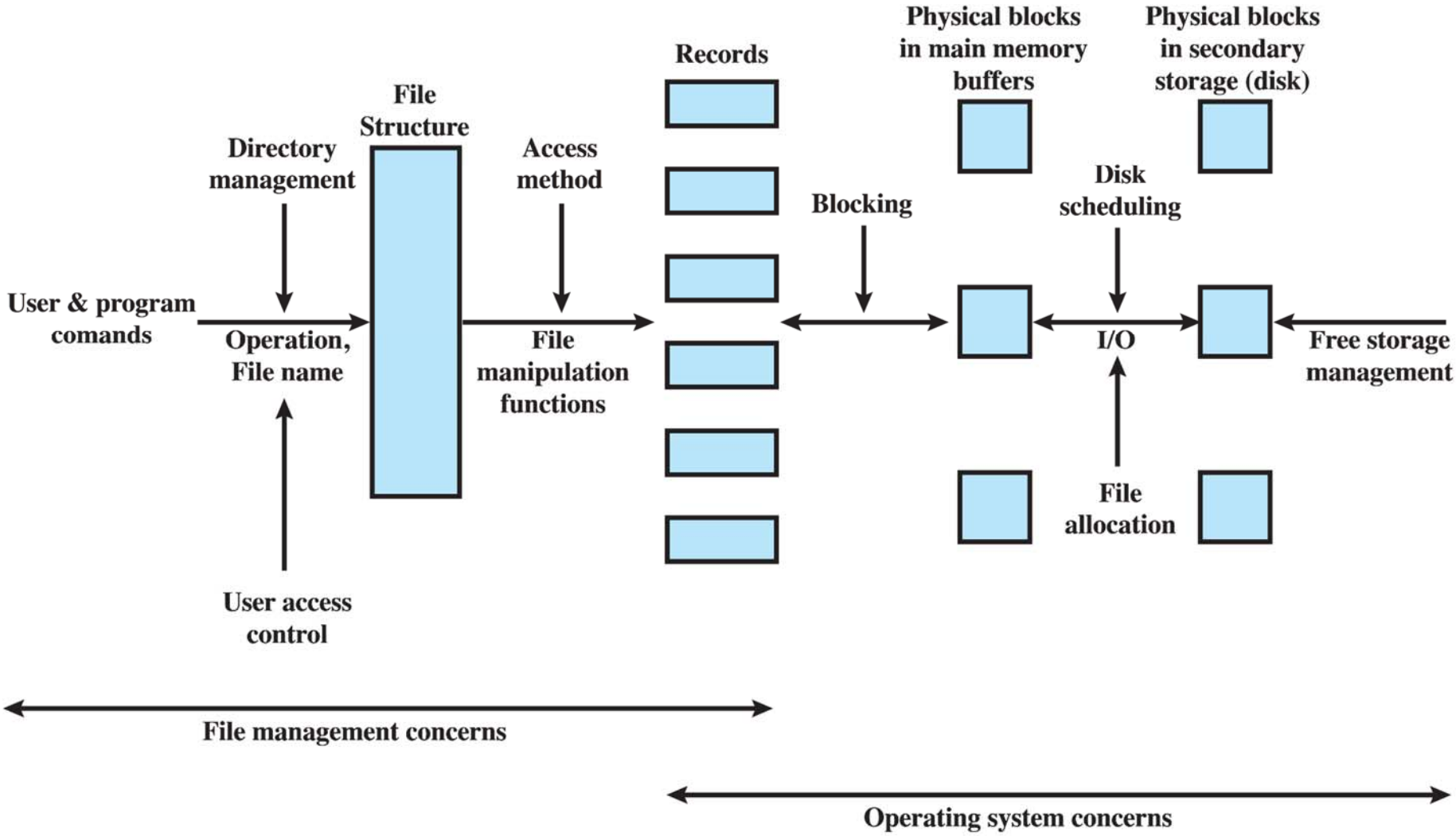


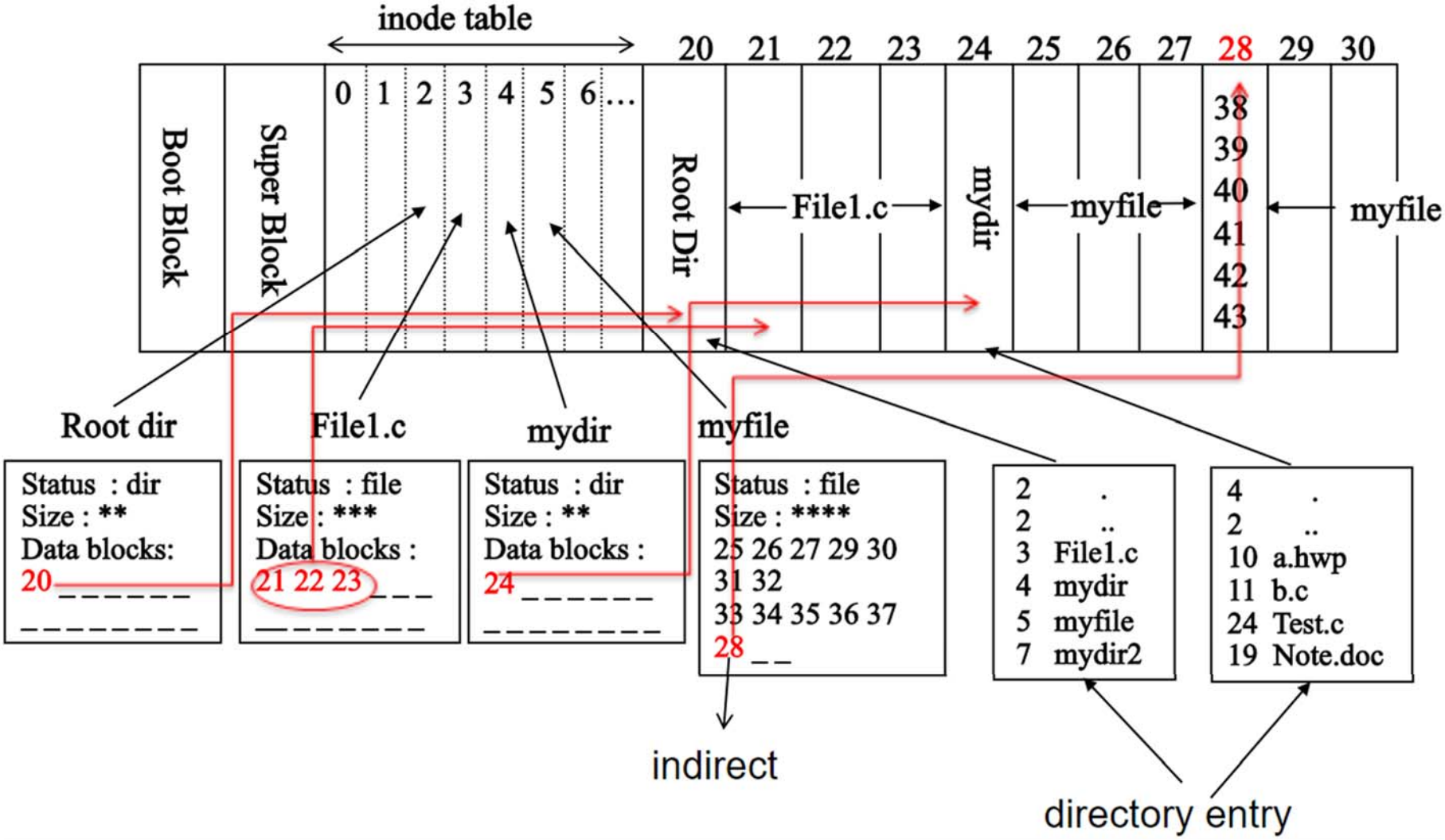
表 9 - 11 硬盘分区表项结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。 0x00-不从该分区引导操作系统；0x80-从该分区引导操作系统。
0x01	head	字节	分区起始处的磁头号。磁头号范围为 0--255。
0x02	sector	字节	分区起始的当前柱面中扇区号(位 0-5) 和柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始处的柱面号低 8 位。柱面号范围为 0--1023。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux …
0x05	end_head	字节	分区结束处的磁头号。磁头号范围为 0--255。
0x06	end_sector	字节	分区结束的当前柱面中扇区号(位 0-5) 和柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	分区结束的柱面号低 8 位。柱面号范围为 0--1023。
0x08-0x0b	start_sect	长字	分区起始的物理扇区号。它以整个硬盘的扇区号顺序从 0 计起。
0x0c-0x0f	nr_sects	长字	分区占用的扇区数。

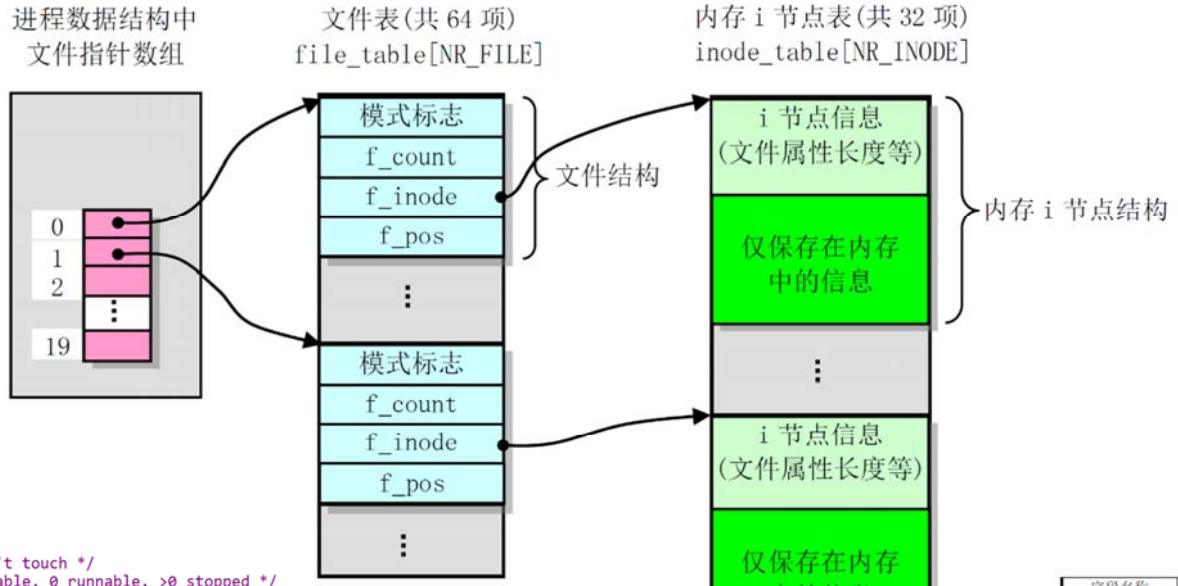
Elements of File Management



Demo


inode table
0 1 2 3 4 5 6 …
20 21 22 23 24 25 26 27 28 29 30
Boot Block
Super Block
Root Dir
File1.c
mydir
myfile
myfile
38
39
40
41
42
43
Root dir
File1.c
mydir
myfile
Status : dir
Size : **
Data blocks:
20
Status : file
Size : ***
Data blocks :
21 22 23
Status : dir
Size : **
Data blocks :
24
Status : file
Size : ****
25 26 27 29 30
31 32
33 34 35 36 37
28
2 .
2 ..
3 File1.c
4 mydir
5 myfile
7 mydir2
4 .
2 ..
10 a.hwp
11 b.c
24 Test.c
19 Note.doc
indirect
directory entry

进程打开文件使用的内核数据结构




```
93: struct m_inode {
94:     unsigned short i_mode;
95:     unsigned short i_uid;
96:     unsigned long i_size;
97:     unsigned long i_mtime;
98:     unsigned char i_gid;
99:     unsigned char i_nlinks;
100:    unsigned short i_zone[9];
101: /* these are in memory also */
102:    struct task_struct * i_wait;
103:    unsigned long i_atime;
104:    unsigned long i_ctime;
105:    unsigned short i_dev;
106:    unsigned short i_num;
107:    unsigned short i_count;
108:    unsigned char i_lock;
109:    unsigned char i_dirt;
110:    unsigned char i_pipe;
111:    unsigned char i_mount;
112:    unsigned char i_seek;
113:    unsigned char i_update;
114: } « end m_inode » ;
```

```
80: struct task_struct {
81: /* these are hardcoded - don't touch */
82:     long state; /* -1 unrunnable, 0 runnable, >0 stopped */
83:     long counter;
84:     long priority;
85:     long signal;
86:     struct sigaction sigaction[32];
87:     long blocked;   /* bitmap of masked signals */
88: /* various fields */
89:     int exit_code;
90:     unsigned long start_code,end_code,end_data,brk,start_stack;
91:     long pid,father,pgrp,session,leader;
92:     unsigned short uid,euid,suid;
93:     unsigned short gid,egid,sgid;
94:     long alarm;
95:     long utime,stime,cutime,cstime,start_time;
96:     unsigned short used_math;
97: /* file system info */
98:     int tty;        /* -1 if no tty, so it must be signed */
99:     unsigned short umask;
100:    struct m_inode * pwd;
101:    struct m_inode * root;
102:    struct m_inode * executable;
103:    unsigned long close_on_exec;
104:    struct file * filp[NR_OPEN];
105: /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
106:    struct desc_struct ldt[3];
107: /* tss for this task */
108:    struct tss_struct tss;
109: } « end task_struct » ;
```

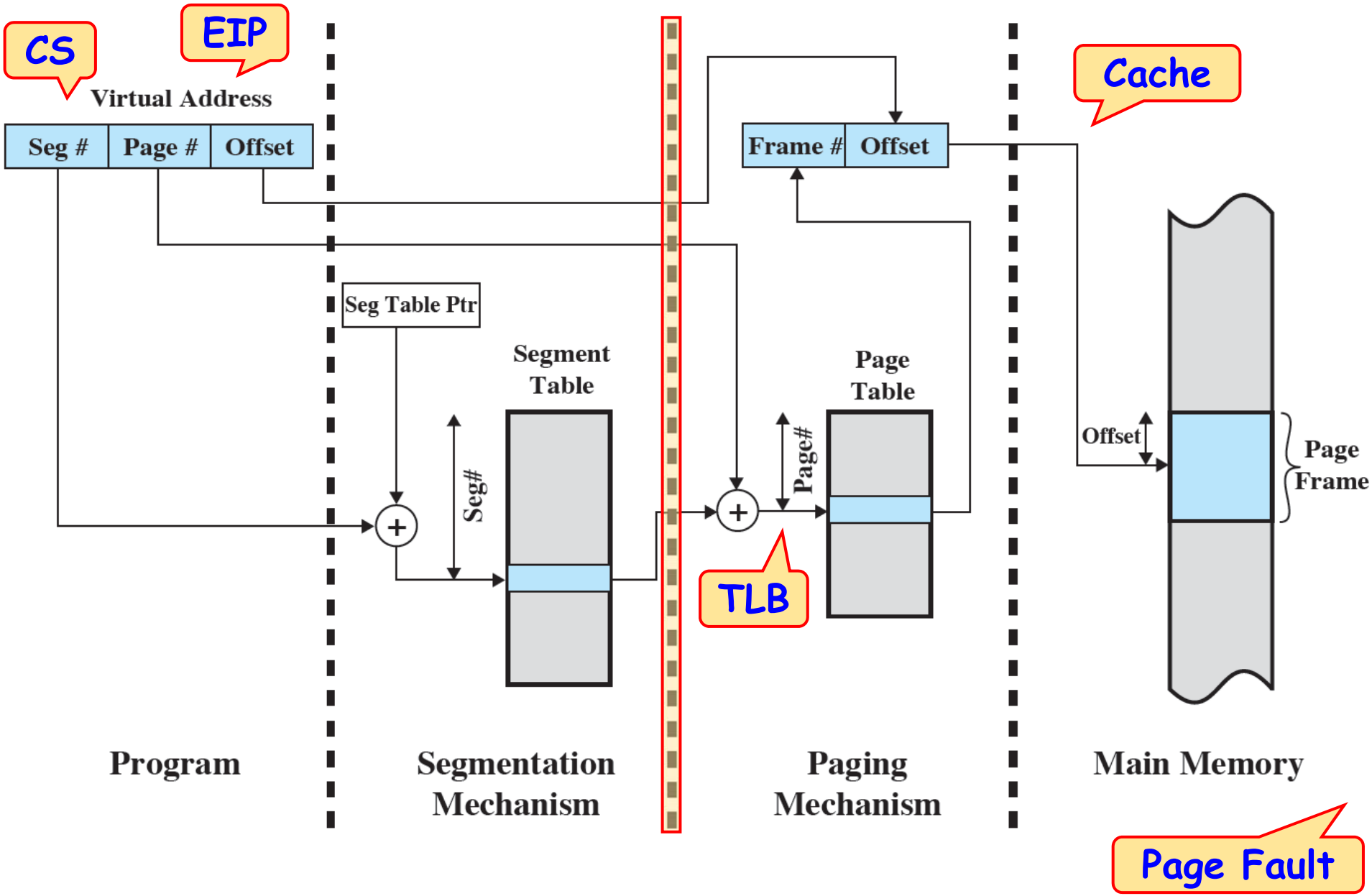
```
157: struct dir_entry {
158:     unsigned short inode;
159:     char name[NAME_LEN];
160: };
161:
162: extern struct m_inode inode_table[NR_INODE];
163: extern struct file file_table[NR_FILE];
```

```
116: struct file {
117:     unsigned short f_mode;
118:     unsigned short f_flags;
119:     unsigned short f_count;
120:     struct m_inode * f_inode;
121:     off_t f_pos;
122: };
123:
```

字段名称	数据类型	说明
i_mode	short	文件的类型和属性（rwx 位）
i_uid	short	文件宿主的用户 id
i_size	long	文件长度（字节）
i_mtime	long	修改时间（从 1970.1.1:0 时算起，秒）
i_gid	char	文件宿主的组 id
i_nlinks	char	链接数（有多少个文件目录项指向该 i 节点）
i_zone[9]	short	文件所占用的盘上逻辑块号数组。其中： zone[0]-zone[6]是直接块号； zone[7]是一次间接块号； zone[8]是二次（双重）间接块号。 注：zone 是区的意思，可译成区块或逻辑块。 对于设备特殊文件名的 i 节点，其 zone[0]中存放的是该文件名所指设备的设备号。
i_wait	task_struct *	等待该 i 节点的进程。
i_atime	long	最后访问时间。
i_ctime	long	i 节点自身被修改时间。
i_dev	short	i 节点所在的设备号。
i_num	short	i 节点号。
i_count	short	i 节点被引用的次数，0 表示空闲。
i_lock	char	i 节点被锁定标志。
i_dirt	char	i 节点已被修改（脏）标志。
i_pipe	char	i 节点用作管道标志。
i_mount	char	i 节点安装了其他文件系统标志。
i_seek	char	搜索标志（lseek 操作时）。
i_update	char	i 节点已更新标志。

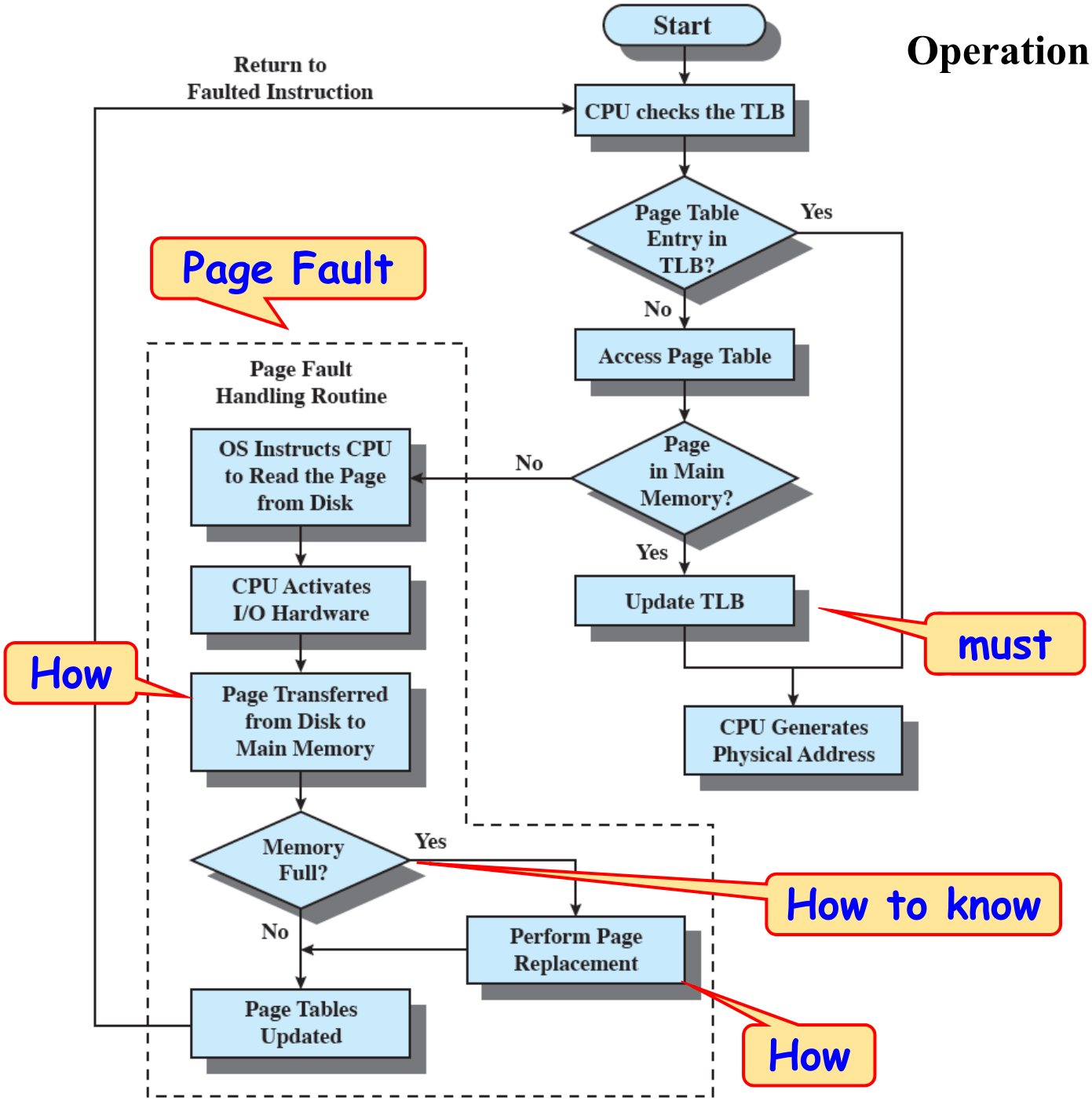
在盘上和内存中的字段，共 32 字节

仅在内存中使用的字段

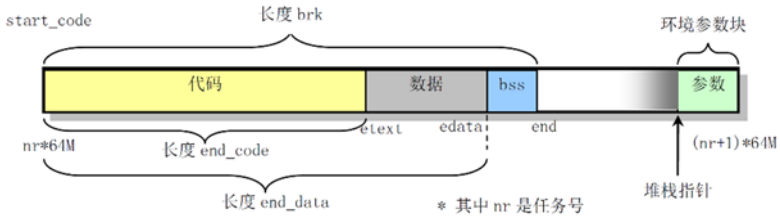


Address Translation in a Segmentation/Paging System

Operation of Paging and TLB



Page fault




```
366: void do_no_page(unsigned long error_code,unsigned long address)
367: {
368:     int nr[4];
369:     unsigned long tmp;
370:     unsigned long page;
371:     int block,i;
372:
373:     address &= 0xfffff000;
374:     tmp = address - current->start_code;
375:     if (!current->executable || tmp >= current->end_data) {
376:         get_empty_page(address);
377:         return;
378:     }
379:     if (share_page(tmp))
380:         return;
381:     if (!(page = get_free_page()))
382:         oom();
383: /* remember that 1 block is used for header */
384:     block = 1 + tmp/BLOCK_SIZE;
385:     for (i=0 ; i<4 ; block++,i++)
386:         nr[i] = bmap(current->executable,block);
387:     bread_page(page,current->executable->i_dev,nr);
388:     i = tmp + 4096 - current->end_data;
389:     tmp = page + 4096;
390:     while (i-- > 0) {
391:         tmp--;
392:         *(char *)tmp = 0;
393:     }
394:     if (put_page(page,addres
395:         return;
396:     free_page(page);
397:     oom();
398: } « end do_no_page »
```

```
page_fault:
        xchgl %eax,(%esp)
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%edx
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
        movl %cr2,%edx
        pushl %edx
        pushl %eax
        testl $1,%eax
        jne 1f
        call do_no_page
        jmp 2f
1:      call do_wp_page
2:      addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

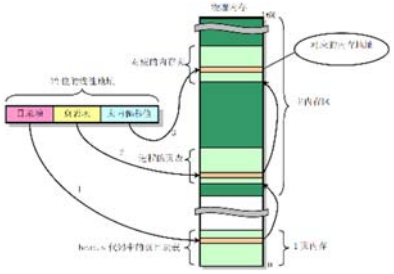
```
140: int bmap(struct m_inode * inode,int block)
141: {
142:     return _bmap(inode,block,0);
143: }
```

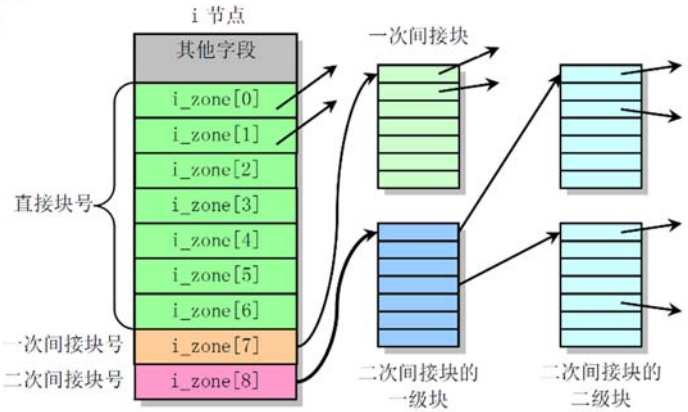
```
72: static int _bmap(struct m_inode * inode,int block,int create)
73: {
74:     struct buffer_head * bh;
75:     int i;
76:
77:     if (block<0)
78:         panic("_bmap: block<0");
79:     if (block >= 7+512+512*512)
80:         panic("_bmap: block>big");
81:     if (block<7) {
82:         if (create && !inode->i_zone[block])
83:             if ((inode->i_zone[block]=new_block(inode->i_dev))) {
84:                 inode->i_ctime=CURRENT_TIME;
85:                 inode->i_dirt=1;
86:             }
87:         return inode->i_zone[block];
88:     }
89:     block -= 7;
```

```
286: #define COPYBLK(from,to) \
287: __asm__("cld\n\t" \
288:     "rep\n\t" \
289:     "movsl\n\t" \
290:     ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
291:     )
292:
293: /*
294:  * bread_page reads four buffers into memory at the desired address. It's
295:  * a function of its own, as there is some speed to be got by reading them
296:  * all at the same time, not waiting for one to be read, and then another
297:  * etc.
298:  */
299: void bread_page(unsigned long address,int dev,int b[4])
300: {
301:     struct buffer_head * bh[4];
302:     int i;
303:
304:     for (i=0 ; i<4 ; i++)
305:         if (b[i]) {
306:             if ((bh[i] = getblk(dev,b[i])))
307:                 if (!bh[i]->b_uptodate)
308:                     ll_rw_block(READ,bh[i]);
309:         } else
310:             bh[i] = NULL;
311:     for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
312:         if (bh[i]) {
313:             wait_on_buffer(bh[i]);
314:             if (bh[i]->b_uptodate)
315:                 COPYBLK((unsigned long) bh[i]->b_data,address);
316:             brelse(bh[i]);
317:         }
318: } « end bread_page »
```

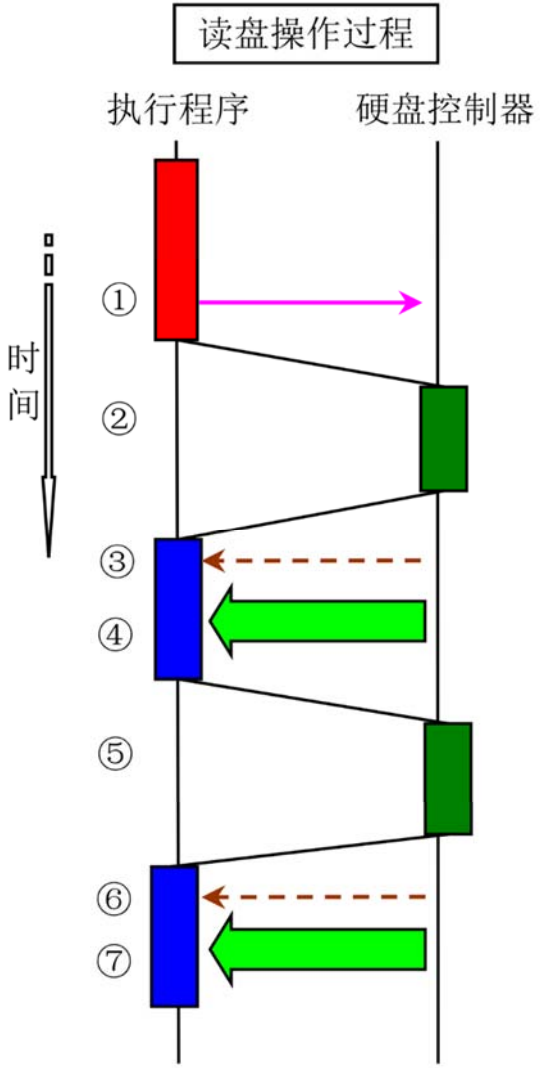
```
39: static inline void wait_on_buffer(struct buffer_head * bh)
40: {
41:     cli();
42:     while (bh->b_lock)
43:         sleep_on(&bh->b_wait);
44:     sti();
45: }
```

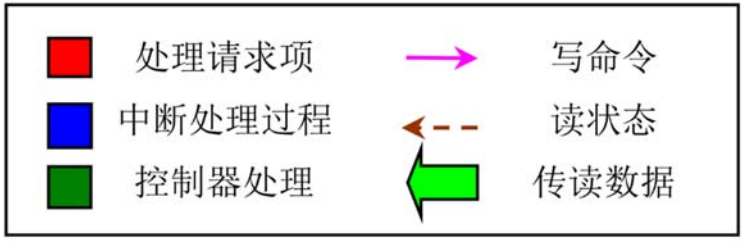
```
151: void sleep_on(struct task_struct **p)
152: {
153:     struct task_struct *tmp;
154:
155:     if (!p)
156:         return;
157:     if (current == &(init_task.task))
158:         panic("task[0] trying to sleep");
159:     tmp = *p;
160:     *p = current;
161:     current->state = TASK_UNINTERRUPTIBLE;
162:     schedule();
163:     if (tmp)
164:         tmp->state=0;
165: }
```





读硬盘数据操作的时序关系





① 函数 do_hd_request()中用 hd_out()发送读命令，然后退出返回调用程序执行，同时等待控制器发中断；

② 控制器接到读命令后就执行从硬盘读数据到控制器缓冲区中的操作。操作完后就发中断；

③ 在中断处理程序中执行 read_intr()。读取控制器状态；

④ 并从控制器缓冲区读取 1 个扇区数据到请求项缓冲块中；

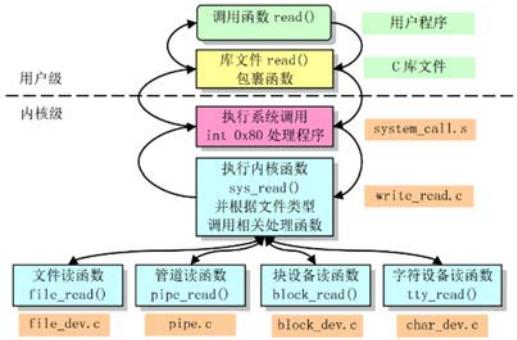
⑤ 控制器随后再执行从硬盘读下一扇区数据到控制器缓冲区中的操作。操作完后就发中断；

⑥ 在中断处理程序中执行 read_intr()。读取控制器状态；

⑦ 并从控制器缓冲区读取 1 扇区数据到请求项缓冲块中，若数据已全部读出则处理本请求项结束事宜，并执行其他硬盘请求操作。

```
74: fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
75: sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
76: sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
77: sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
78: sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
79: sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
80: sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
81: sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
82: sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
83: sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
84: sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
85: sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
86: sys_setreuid,sys_setregid };
```

```
55: int sys_read(unsigned int fd,char * buf,int count)
56: {
57:     struct file * file;
58:     struct m_inode * inode;
59:
60:     if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
61:         return -EINVAL;
62:     if (!count)
63:         return 0;
64:     verify_area(buf,count);
65:     inode = file->f_inode;
66:     if (inode->i_pipe)
67:         return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
68:     if (S_ISCHR(inode->i_mode))
69:         return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
70:     if (S_ISBLK(inode->i_mode))
71:         return block_read(inode->i_zone[0],&file->f_pos,buf,count);
72:     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73:         if (count+file->f_pos > inode->i_size)
74:             count = inode->i_size - file->f_pos;
75:         if (count<=0)
76:             return 0;
77:         return file_read(inode,file,buf,count);
78:     }
79:     printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
80:     return -EINVAL;
81: } « end sys_read »
```




```
17: int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18: {
19:     int left,chars,nr;
20:     struct buffer_head * bh;
21:
22:     if ((left=count)<=0)
23:         return 0;
24:     while (left) {
25:         if ((nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE))) {
26:             if (!(bh=bread(inode->i_dev,nr)))
27:                 break;
28:         } else
29:             bh = NULL;
30:         nr = filp->f_pos % BLOCK_SIZE;
31:         chars = MIN( BLOCK_SIZE-nr , left );
32:         filp->f_pos += chars;
33:         left -= chars;
34:         if (bh) {
35:             char * p = nr + bh->b_data;
36:             while (chars-->0)
37:                 put_fs_byte(*(p++),buf++);
38:             brelse(bh);
39:         } else {
40:             while (chars-->0)
41:                 put_fs_byte(0,buf++);
42:         }
43:     } « end while left »
44:     inode->i_atime = CURRENT_TIME;
45:     return (count-left)?(count-left):-ERROR;
46: } « end file_read »
```

```
11: /* devices are as follows: (same as minix, so we can use the minix
12:  * file system. These are major numbers.)
13:  *
14:  * 0 - unused (nodev)
15:  * 1 - /dev/mem
16:  * 2 - /dev/fd
17:  * 3 - /dev/hd
18:  * 4 - /dev/ttyx
19:  * 5 - /dev/tty
20:  * 6 - /dev/lp
21:  * 7 - unnamed pipes
```

do_hd_request ( )

```
78: #elif (MAJOR_NR == 3)
79: /* harddisk */
80: #define DEVICE_NAME "harddisk"
81: #define DEVICE_INTR do_hd
82: #define DEVICE_REQUEST do_hd_request
83: #define DEVICE_NR(device) (MINOR(device)/5)
84: #define DEVICE_ON(device)
85: #define DEVICE_OFF(device)
86:
87: #elif
88: /* unknown blk device */
89: #error "unknown blk device"
90:
91: #endif
92:
93: #define CURRENT (blk_dev[MAJOR_NR].current_request)
94: #define CURRENT_DEV DEVICE_NR(CURRENT->dev)
```

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```


```
145: void ll_rw_block(int rw, struct buffer_head * bh)
146: {
147:     unsigned int major;
148:
149:     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
150:     !(blk_dev[major].request_fn)) {
151:         printk("Trying to read nonexistent block-device\n\r");
152:         return;
153:     }
154:     make_request(major,rw,bh);
155: }
```

```
88: static void make_request(int major,int rw, struct buffer_head * bh)
89: {
90:     struct request * req;
91:     int rw_ahead;
92:
93: /* WRITEA/READA is special case - it is not really needed, so if the */
94: /* buffer is locked, we just forget about it, else it's a normal read */
95:     if ((rw_ahead = (rw == READA || rw == WRITEA))) {
96:         if (bh->b_lock)
97:             return;
98:         if (rw == READA)
99:             rw = READ;
100:        else
101:            rw = WRITE;
102:    }
103:    if (rw!=READ && rw!=WRITE)
104:        panic("Bad block dev command, must be R/W/RA/WA");
105:    lock_buffer(bh);
106:    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
107:        unlock_buffer(bh);
108:        return;
109:    }
110: repeat:
111: /* we don't allow the write-requests to fill up the queue completely:
112:  * we want some room for reads: they take precedence. The last third
113:  * of the requests are only for reads.
114:  */
115:    if (rw == READ)
116:        req = request+NR_REQUEST;
117:    else
118:        req = request+((NR_REQUEST*2)/3);
119: /* find an empty request */
120:    while (--req >= request)
121:        if (req->dev<0)
122:            break;
123: /* if none found, sleep on new requests: check for rw_ahead */
124:    if (req < request) {
125:        if (rw_ahead) {
126:            unlock_buffer(bh);
127:            return;
128:        }
129:        sleep_on(&wait_for_request);
130:        goto repeat;
131:    }
132: /* fill up the request-info, and add it to the queue */
133:    req->dev = bh->b_dev;
134:    req->cmd = rw;
135:    req->errors=0;
136:    req->sector = bh->b_blocknr<<1;
137:    req->nr_sectors = 2;
138:    req->buffer = bh->b_data;
139:    req->waiting = NULL;
140:    req->bh = bh;
141:    req->next = NULL;
142:    add_request(major+blk_dev,req);
143: } « end make_request »
```

```
270: struct buffer_head * bread(int dev,int block)
271: {
272:     struct buffer_head * bh;
273:
274:     if (!(bh=getblk(dev,block)))
275:         panic("bread: getblk returned NULL\n");
276:     if (bh->b_uptodate)
277:         return bh;
278:     ll_rw_block(READ,bh);
279:     wait_on_buffer(bh);
280:     if (bh->b_uptodate)
281:         return bh;
282:     brelse(bh);
283:     return NULL;
284: }
```

```
39: static inline void wait_on_buffer(struct buffer_head * bh)
40: {
41:     cli();
42:     while (bh->b_lock)
43:         sleep_on(&bh->b_wait);
44:     sti();
45: }
46:
```

sleep_on ( ) -> Schedule( )

```
145: void ll_rw_block(int rw, struct buffer_head * bh)
146: {
147:     unsigned int major;
148:
149:     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
150:     !(blk_dev[major].request_fn)) {
151:         printk("Trying to read nonexistent block-device\n\r");
152:         return;
153:     }
154:     make_request(major,rw,bh);
155: }
```

```
54: static void add_request(struct blk_dev_struct * dev, struct request * req)
55: {
56:     struct request * tmp;
57:
58:     req->next = NULL;
59:     cli();
70:     if (req->bh)
71:         req->bh->b_dirt = 0;
72:     if (!(tmp = dev->current_request)) {
73:         dev->current_request = req;
74:         sti();
75:         (dev->request_fn)();
76:         return;
77:     }
78:     for ( ; tmp->next ; tmp=tmp->next)
79:         if ((IN_ORDER(tmp,req) ||
80:             !IN_ORDER(tmp,tmp->next)) &&
81:             IN_ORDER(req,tmp->next))
82:             break;
83:     req->next=tmp->next;
84:     tmp->next=req;
85:     sti();
} « end add_request »
```

电梯算法

一路返回到
wait_on_buffer

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

    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
    !(blk_dev[major].request_fn)) {
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    make_request(major,rw,bh);
}
```

do_hd_request ( )

```c
void do_hd_request(void)
{
    int i,r = 0;
    unsigned int block,dev;
    unsigned int sec,head,cyl;
    unsigned int nsect;

    INIT_REQUEST;
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
        end_request(0);
        goto repeat;
    }
    block += hd[dev].start_sect;
    dev /= 5;
    __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
        "r" (hd_info[dev].sect));
    __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
        "r" (hd_info[dev].head));
    sec++;
    nsect = CURRENT->nr_sectors;
    if (reset) {
        reset = 0;
        recalibrate = 1;
        reset_hd(CURRENT_DEV);
        return;
    }
    if (recalibrate) {
        recalibrate = 0;
        hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
            WIN_RESTORE,&recal_intr);
        return;
    }
    if (CURRENT->cmd == WRITE) {
        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
        for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
            /* nothing */ ;
        if (!r) {
            bad_rw_intr();
            goto repeat;
        }
        port_write(HD_DATA,CURRENT->buffer,256);
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    } else
        panic("unknown hd-command");
} « end do_hd_request »
```

read_intr ( )

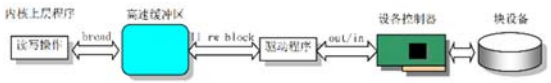
硬盘信息相关，
端口读写

```c
#define outb(value,port) \
__asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))

#define inb(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
_v; \
})

#define outb_p(value,port) \
__asm__ ("outb %%al,%%dx\n" \
        "\tjmp 1f\n" \
        "1:\tjmp 1f\n" \
        "1:"::"a" (value),"d" (port))

#define inb_p(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al\n" \
    "\tjmp 1f\n" \
    "1:\tjmp 1f\n" \
    "1:":"=a" (_v):"d" (port)); \
_v; \
})
```

```c
struct hd_i_struct {
    int head,sect,cyl,wpcom,lzone,ctl;
    };
#ifdef HD_TYPE
struct hd_i_struct hd_info[] = { HD_TYPE };
#define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))
#else
struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
static int NR_HD = 0;
#endif

static struct hd_struct {
    long start_sect;
    long nr_sects;
} hd[5*MAX_HD]={{0,0},};

#define port_read(port,buf,nr) \
__asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr))
```

硬盘信息相关，
端口读写

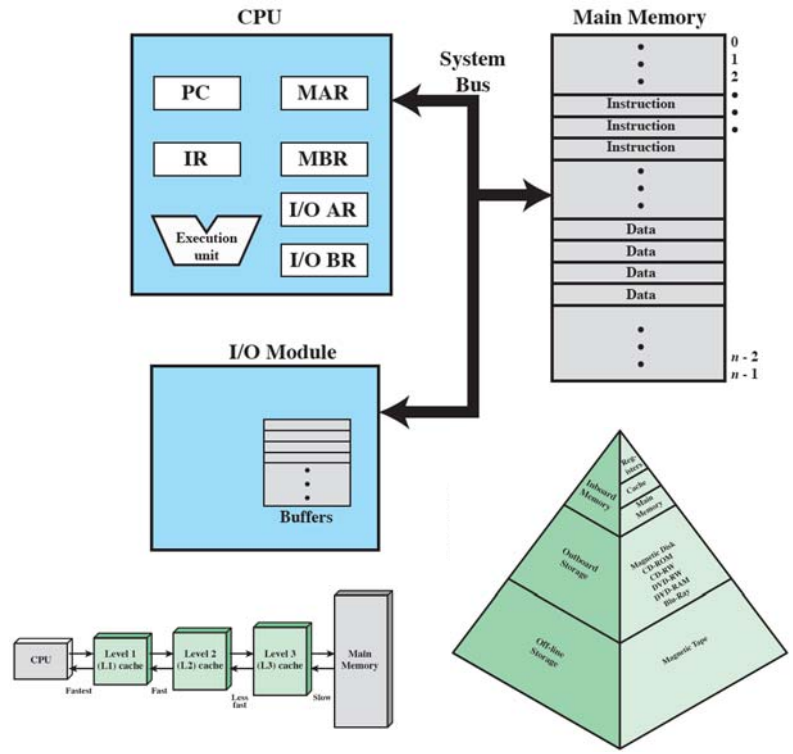
```c
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
        unsigned int head,unsigned int cyl,unsigned int cmd,
        void (*intr_addr)(void))
{
    register int port asm("dx");

    if (drive>1 || head>15)
        panic("Trying to write bad sector");
    if (!controller_ready())
        panic("HD controller not ready");
    do_hd = intr_addr;
    outb_p(hd_info[drive].ctl,HD_CMD);
    port=HD_DATA;
    outb_p(hd_info[drive].wpcom>>2,++port);
    outb_p(nsect,++port);
    outb_p(sect,++port);
    outb_p(cyl,++port);
    outb_p(cyl>>8,++port);
    outb_p(0xA0|(drive<<4)|head,++port);
    outb(cmd,++port);
} « end hd_out »
```

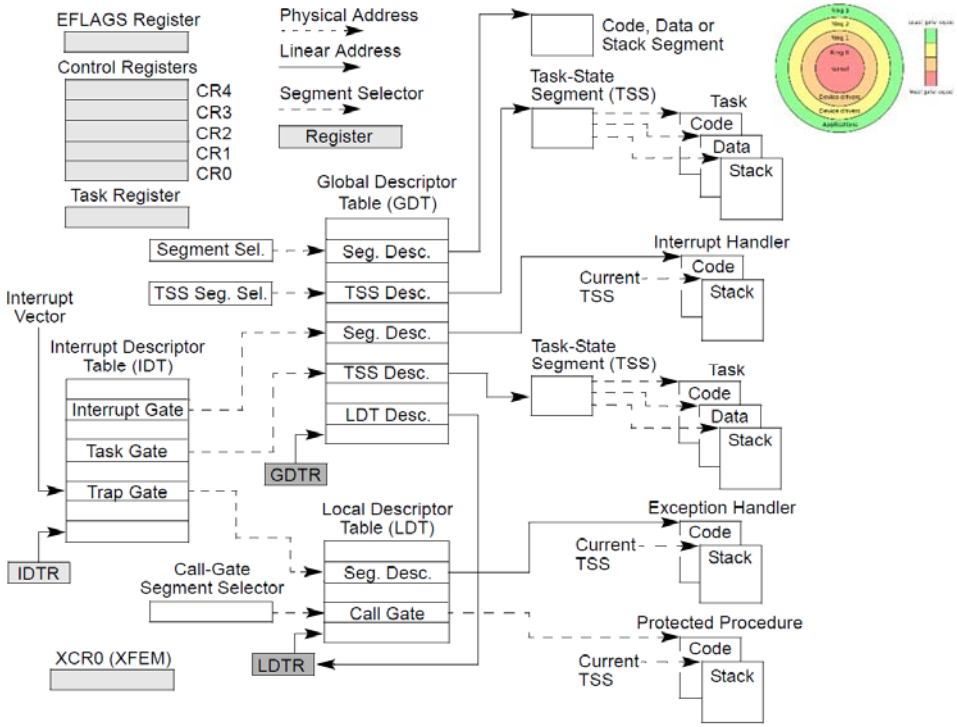


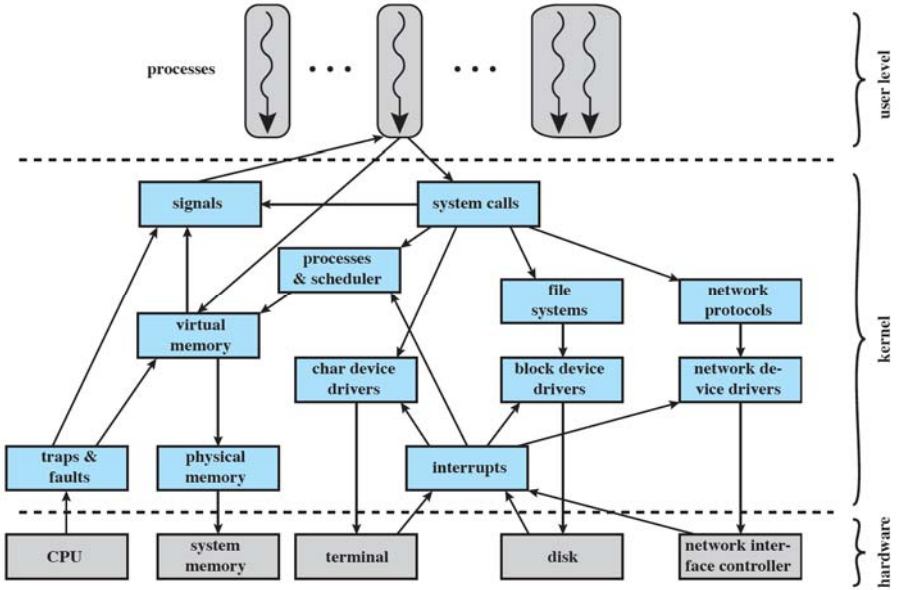

```c
static inline void end_request(int uptodate)
{
    DEVICE_OFF(CURRENT->dev);
    if (CURRENT->bh) {
        CURRENT->bh->b_uptodate = uptodate;
        unlock_buffer(CURRENT->bh);
    }
    if (!uptodate) {
        printk(DEVICE_NAME " I/O error\n\r");
        printk("dev %04x, block %d\n\r",CURRENT->dev,
            CURRENT->bh->b_blocknr);
    }
    wake_up(&CURRENT->waiting);
    wake_up(&wait_for_request);
    CURRENT->dev = -1;
    CURRENT = CURRENT->next;
}
```

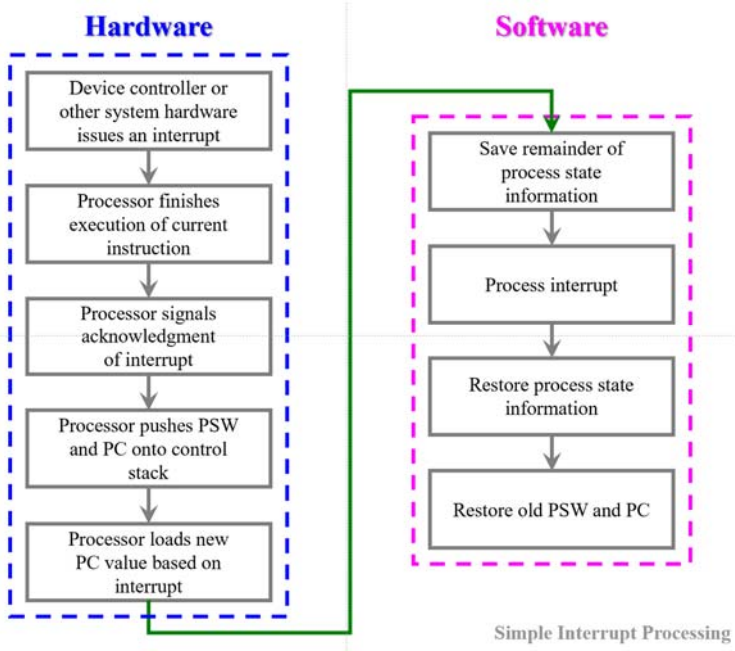
wake_up

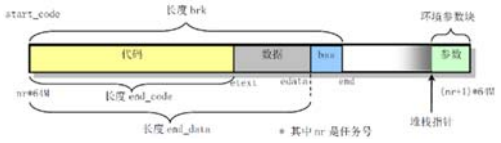
```c
static void read_intr(void)
{
    if (win_result()) {
        bad_rw_intr();
        do_hd_request();
        return;
    }
    port_read(HD_DATA,CURRENT->buffer,256);
    CURRENT->errors = 0;
    CURRENT->buffer += 512;
    CURRENT->sector++;
    if (--CURRENT->nr_sectors) {
        do_hd = &read_intr;
        return;
    }
    end_request(1);
    do_hd_request();
}
```

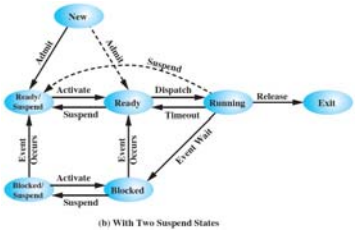

CPU
PC
MAR
IR
MBR
I/O AR
I/O BR
Execution unit
System Bus
Main Memory
Instruction
Instruction
Instruction
Data
Data
Data
Data
I/O Module
Buffers
CPU
Level 1 (L1) cache
Level 2 (L2) cache
Level 3 (L3) cache
Main Memory
Fastest
Fast
Less fast
Slow

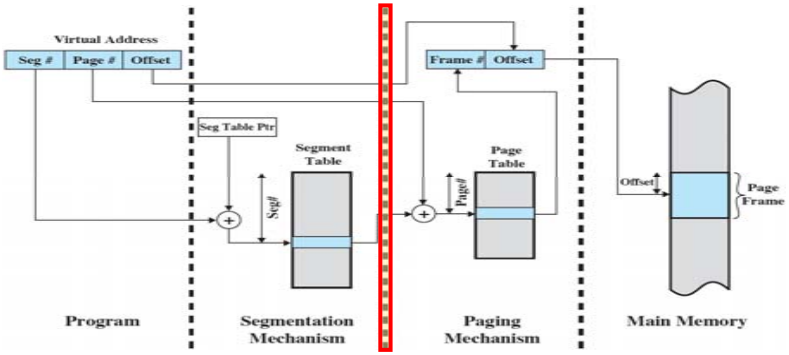

EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Physical Address
Linear Address
Segment Selector
Register
Global Descriptor Table (GDT)
Segment Sel.
TSS Seg. Sel.
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
IDTR
GDTR
Seg. Desc.
TSS Desc.
Seg. Desc.
TSS Desc.
LDT Desc.
Local Descriptor Table (LDT)
Call-Gate Segment Selector
Seg. Desc.
Call Gate
LDTR
XCR0 (XFEM)
Code, Data or Stack Segment
Task-State Segment (TSS)
Task
Code
Data
Stack
Interrupt Handler
Code
Stack
Current TSS
Task-State Segment (TSS)
Task
Code
Data
Stack
Exception Handler
Code
Stack
Current TSS
Protected Procedure
Code
Stack
Current TSS

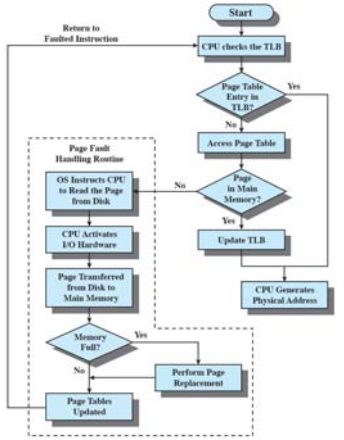

processes
user level
signals
system calls
processes & scheduler
file systems
network protocols
virtual memory
char device drivers
block device drivers
network de-vice drivers
traps & faults
physical memory
interrupts
kernel
CPU
system memory
terminal
disk
network inter-face controller
hardware

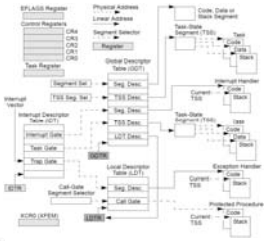

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

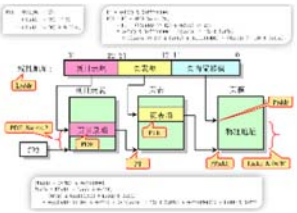

start_code
长度 brk
环境参数块
代码
数据
bss
参数
end
长度 end_code
长度 end_data
堆栈指针
* 其中 nr 是任务号

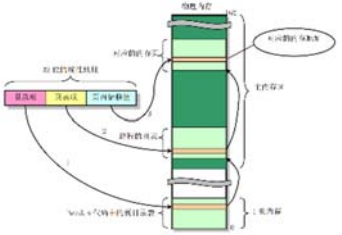

New
Admit
Admit
Suspend
Activate
Ready
Dispatch
Running
Release
Exit
Ready/ Suspend
Suspend
Timeout
Event Occurs
Event Occurs
Event Wait
Activate
Blocked/ Suspend
Blocked
Suspend
(b) With Two Suspend States

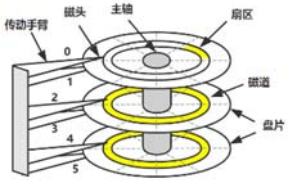

Virtual Address
Seg #
Page #
Offset
Seg Table Ptr
Segment Table
Page Table
Frame #
Offset
Offset
Page Frame
Program
Segmentation Mechanism
Paging Mechanism
Main Memory

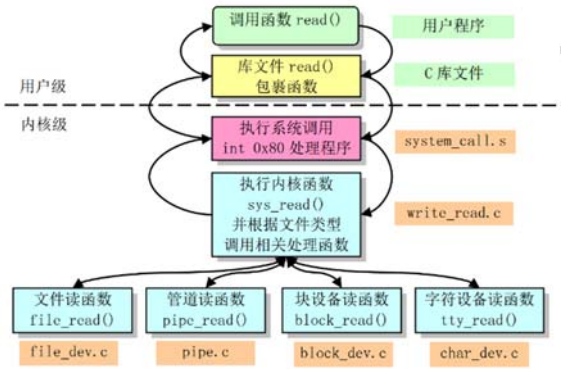

Start
CPU checks the TLB
Page Table Entry in TLB?
Yes
No
Access Page Table
Page Fault Handling Routine
OS Instructs CPU to Read the Page from Disk
No
Page in Main Memory?
Yes
Update TLB
CPU Activates I/O Hardware
CPU Generates Physical Address
Page Transferred from Disk to Main Memory
Memory Full?
Yes
No
Perform Page Replacement
Page Tables Updated
Return to Faulted Instruction

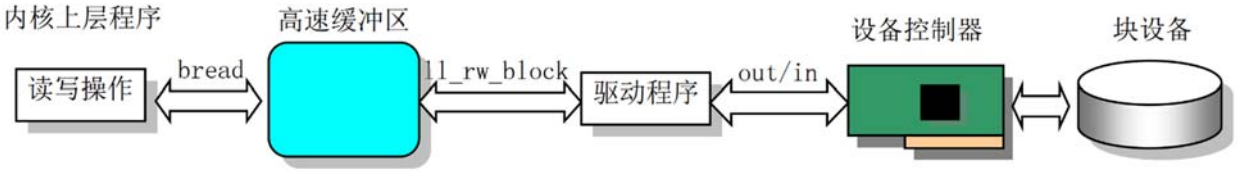


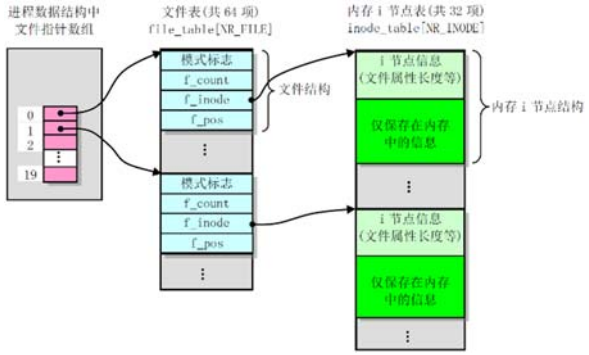


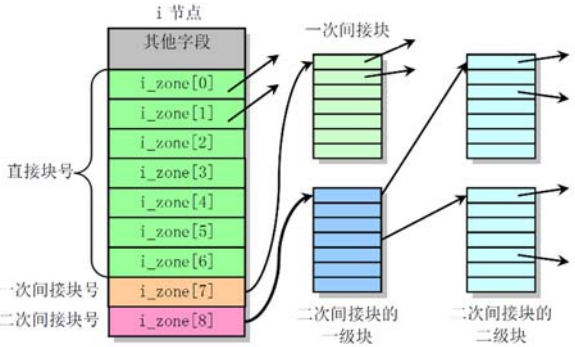


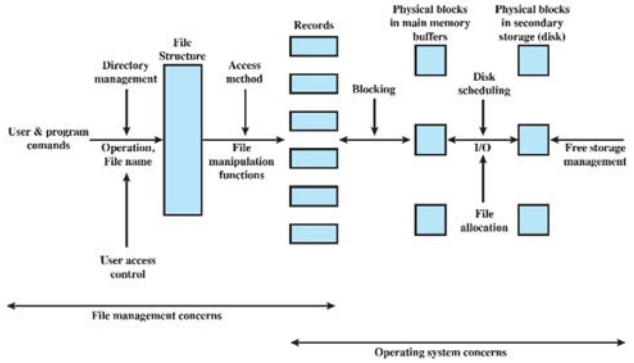

传动手臂
磁头
主轴
扇区
磁道
盘片


用户级
内核级
调用函数 read()
库文件 read()
包裹函数
执行系统调用
int 0x80 处理程序
执行内核函数
sys_read()
并根据文件类型
调用相关处理函数
用户程序
C 库文件
system_call.s
write_read.c
文件读函数
file_read()
管道读函数
pipe_read()
块设备读函数
block_read()
字符设备读函数
tty_read()
file_dev.c
pipe.c
block_dev.c
char_dev.c


内核上层程序
高速缓冲区
设备控制器
块设备
读写操作
bread
ll_rw_block
驱动程序
out/in


进程数据结构中
文件指针数组
文件表(共 64 项)
file_table[NR_FILE]
内存 i 节点表(共 32 项)
inode_table[NR_INODE]
模式标志
f_count
f_inode
f_pos
文件结构
i 节点信息
(文件属性长度等)
仅保存在内存中的信息
内存 i 节点结构


i 节点
其他字段
i_zone[0]
i_zone[1]
i_zone[2]
i_zone[3]
i_zone[4]
i_zone[5]
i_zone[6]
i_zone[7]
i_zone[8]
直接块号
一次间接块号
二次间接块号
一次间接块
二次间接块的一级块
二次间接块的二级块


Directory management
File Structure
Access method
Records
Physical blocks in main memory buffers
Physical blocks in secondary storage (disk)
User & program comands
Operation, File name
File manipulation functions
Blocking
Disk scheduling
I/O
Free storage management
File allocation
User access control
File management concerns
Operating system concerns

漫谈 OS

- 虚拟
  - 虚拟内存、虚拟文件系统
- 并行
  - 流水、进程、线程
- 硬件加速
  - Cache，TLB，缓冲，磁盘缓冲
- 抽象
  - 层次模型
- 主奴机制 -- operating
  - 软硬结合
  - 模式
- 并发、共享
  - 加锁，信号量
- 中断
- 栈
  - 各种巧妙用法
- 内存映射
  - Map、变换
- 调度
  - Trade-off
- 视角
  - Hardware、OS、User