**1.intro to OS**

**What is operation system?**

OS is a kind of **software** which helps to interact with hardware, it consists of **bootloader, kernel and system programs**. In order to illustrate how it collaborate with other part of computer, we may abstract the computer structure as an onion model, then the **kernel part is the core** of OS, and explain why OS can make a difference.

**What does operation system consist of?**

We may illustrate further by detailed explain each part of OS.

**Bootloade**r is at the first sector of hard-disk, which helps load OS into memory. (as we mention in the first part, OS is more like a program)

**Kernel** is the core of OS, which mainly help with process management and memory management, which decided how to use CPU and how to use memory. It will also involve something like security and network.

**System programs** are what makes use of OS (kernel), which helps user to interact with OS, so OS can interact with hardware and hardware can help to get job done for people. E.g. system libraries, OS configuration, system program (vim, Xcode), shells (bash, windows GUI), admin tools, user application.

**What is system call? What is kernel mode and user mode?**

**(1)** system call is a mechanism used by an application to request a service from the operating system’s kernel. When this happens, means, user-mode applications request services or resources from the OS, which operates in kernel mode. When the more upper application wants to make use of OS, it should obey some standard, such as POSIX (Linux) and WinAPI standards.

**(2)** user mode and kernel mode

Because we don’t want user directly operate on hardware, so we separate the right. For kernel it can interact with hardware, so it is called kernel mode, else is user mode, which can do system call in order to use hardware.

**2. process management**

**how interrupt happens?**

Timer can make it happen, then ISR can help to find the next address for CPU.

Some **trigger** (for example, timer) pushes the IRQ to interrupt, then PC will save the recent value in **stack**, and ask interrupt vector table to find **ISR**, which is a small bit of code can help to read/ write/ tend to device. (it is actually a kind of interrupt handler, which can help to switch to new process) after this handler exists, the pc will continue from the previous address.

**what happens when want to do switch context?**

For example, when timer interrupt process A for process B

(1) push the recent PC value into A’s stack

(2) ISR calls portSAVECONTEXT, so CPU register values and CPU status Word Register also into A’s stack( we also change A’s TCB’s stack pointer’s value when store)

(3) find the location(SPH/SPL) for B’s stack, and change the current pxCurrentTCB to be that location, call portRESTORE\_CONTEXT

(4) B’s register value load to registers

(5) ISR exit, B’s PC load to PC

Then it will continue execute B like it never leaved before

**what is TCB?**

TCB is task control block, which contains all the status of a process. For example, process id, stack pointer, open files, pending signals, CPU usage and so on. Each process has its own TCB, and all the TCB are stored in a process table.

# 2023.11.22

**3. scheduling**

**Why we need scheduling?**

There are many different computer task types and process types, some are CPU-bound and some are I/O bound. Also, there are different requirements from human beings (different types of multitaskers). Some may want us to react quick, some may strictly control the execute time , some may just require the reasonable allocation of resources. According to this requirement, we cannot randomly deal with the problem that how to use CPU on each process, so we need a reasonable schedule to make fully use of computer resources.

**Some types of multitaskers**

**Batch** Processing (single process runs to completion)

**Co-operative** Multitasking (processes **voluntarily** give up CPU time)

**Pre-emptive** Multitasking (processes can be **forcefully** suspended)

**Real-Time** Multitasking (processes have fixed deadlines)​

**How can we arrange the process?**

**// simple and roughly explanation about scheduling policy**

**Fixed Priority Policy**: Tasks are queued and run based on assigned priority.

**FCFS**: Jobs are stored in a queue and run in the order they arrive.

**Shortest Job First**: Jobs are ordered by the total CPU time used.

**Voluntary Scheduling**: Processes invoke the scheduler using a "yield" function​

**Some scheduling policies**

**Fixed priority policy (suitable for all four types of multitaskers)**

each process is assigned a priority by the programmer (0 is the highest), and they will be executed in this turn. For batch processing and co-operative, we will pick the one in the queue with highest priority; for pre-emptive and real-time, we will suspend the current task as long as there is high priority task becomes ready.

**First come first served policy (suitable for batch processing)**

Arriving job stored in a queue, after one is finished, next in the queue takes its turn and run. If every job is bounded, then it is starvation free.

**Shortest job first (suitable for batch processing)**

Processes are ordered by total CPU time used. Task that takes less time will be served first. It can reduce average waiting time, however, it may cause starvation, if one task is very large and always wait for others to be executed first.

**Voluntary scheduling (suitable for co-operative multitasking)**

Use yield function to invoke the scheduler, cause the current process to suspend and another process started up

**Shortest remaining time (suitable for pre-emptive multitasking)**

Process is ordered by remaining CPU time. (which is different from batching processing is that, we don’t have to wait for one process ends to start another)

**Round-robin with timer (suitable for pre-emptive multitasking)**

Like first come first server, tasks are put in the queue. But instead of waiting for it to finished, we set a fixed time for it, after this fixed time, scheduler will be invoked and next task in the queue is selected to run.

In the same system, there can be different queues, each with different policy and priority. And they will be executed by the priority.

We can use nice to adjust the priority of our process in the system. Normal user can change the priority at the same level or down by 19, while superuser can improve priority by 20 or reduce priority by 19.

**Scheduling in Linux**

**Real-Time FIFO**

Basic like first in first out policy, but can be pre-empted by higher priority RT-FIFO process.

If there is real-time FIFO in the Linux system, they always run first in their priority order.

**Real-Time Round-Robin**

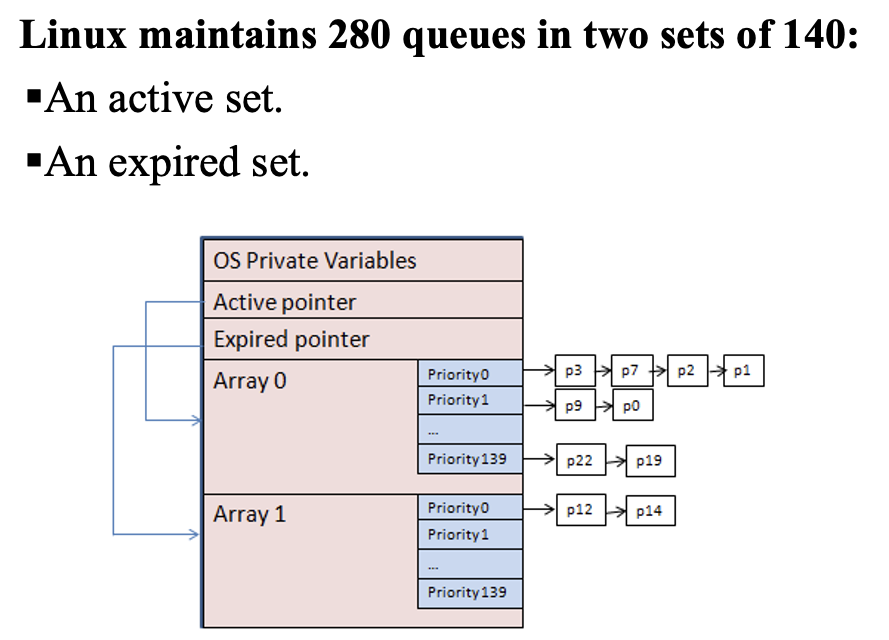
Ordered by the priority, then it just can be pre-empted after a time slice

(time tick is 1 ms, we can also say the scheduler is called at a rate of 1000Hz)

**Non-real time processes**

While the process with priority 0 to 99 belong to the first 2 policies, processes priority level from 100 to 139 are non-real time processes. (which can be seen at the below picture)

// attention: Linux is “soft real-time” scheduling, which means it try hard to meet the DDL, but doesn’t guarantee success. (unlike RMS and EDF. Because scheduling policy itself is not enough for real-time system, you must also consider the worst time of tasks’ execution and hardware support, which is more predictable when using)



**Time quantums and process priorities(for non-real time processes)**

(for RR, for example, priority level 100-800 ms, priority 139-5ms)

Priority = base + f(nice) + g(cpu usage estimate) ( if a process has already consumed a lot of CPU, the priority may be downgraded->for system performance and fairness)

In this case, we will boost the priority to the process which has been blocked or is interactive process (Linux may give -5 boost for them).

**4. Inter-process communication**

**Race condition**

More than one process want to access the same storage (memory location, global variables, hardware registers (not CPU, but more like configuration register),files)

**Mutual exclusion(mutex)**

Two processes cannot read or write shared resources at the same time

**Critical section**

When a process is multiplate the storage out of its own memory space (reading/updating global variables)

**How to implement a mutual exclusion**

**Disabling interrupts**

We don’t use timer to interrupt when use this implementation. This is working because (1) when one process enters its critical section, it will not switch to other process before it leaves the critical section. (2) If other process want to run, we need timer to switch process, but now we cannot use them, so it ensure there is only one process in its critical section.

However, as other process may be blocked because of that, it may be inefficient. Because it may (1) cause the entire system to grind to halt. (2) when there are more than one processors (2) will not happen, as the other process can choose to use another CPU instead of being blocked. (only suitable for single-processor and single core systems)

**Lock variables**

Use a lock variable to indicate if there is some process is in critical section, if other process sees there is, then it doesn’t run until it see the lock has been released

Which doesn’t work because on lock itself, two processes can enter race section.

**Test and set lock**

（Which tries to solve the problem appears in lock variable）

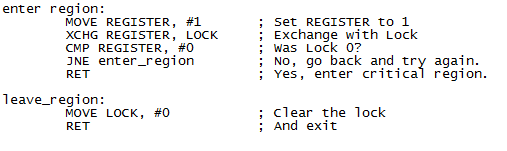
As we cannot solve this problem directly by software, we have to use some traits of hardware to solve this problem. And we use the instruction provided by microprocessors, named TSL reg, lock.

This instruction is atomic that CPU locks the address and data buses! (which means other process cannot even read from this address, even if there can be other CPU). Then the value will be in reg, and we set to 0 or 1 to present it has been used or released. After change the data, CPU unlocks the address and data buses.

This will not block switching process, just ensure there will be one process can have the right to update the value in this address. However, it may also cause waste of CPU resources, because if one process has not released the lock, the other process which is running and need to check the lock will continuously checking if the lock has been released. (the essence of it is to make some condition when a process want to use some global variables)

We may also use other instructions such as XCHG instruction. (which used In intel machines) （it store a value in itself and exchange this value and lock. It actually has the same effect as TSL instruction）

Major issue of this approach is the inefficiency of busy-waiting, which will consume CPU resources as the process acquire the lock, load it into register and compare.



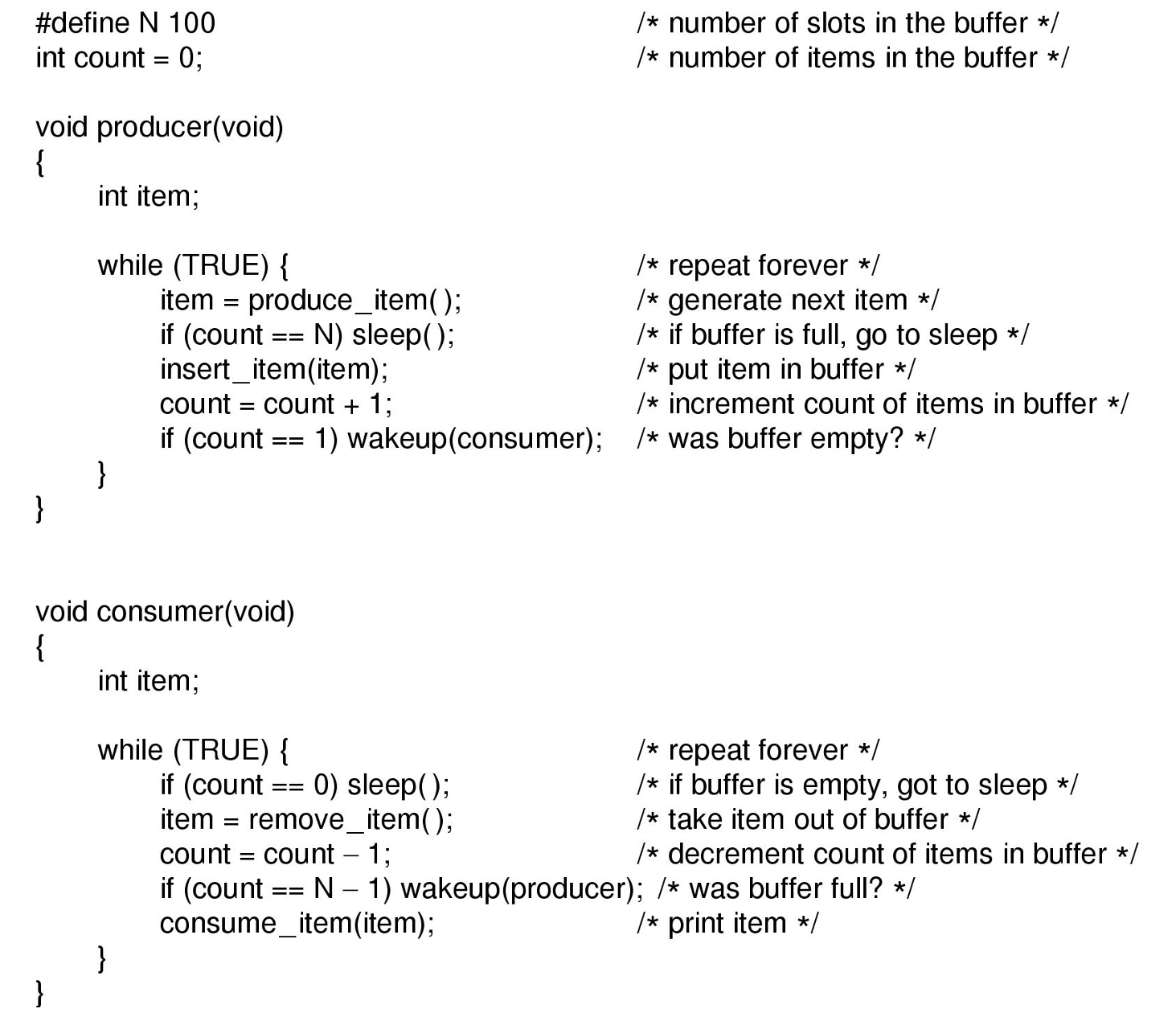
（the reason why busy waiting will happen is because JNE enter\_region, so it will constantly checking）

Another problem is called dead-lock, when processes has set precondition for each other in a bad manner, or the processes has priority contradict to the usage of global variable.

**Sleep/wakeup**

When a process find another process is in critical section (using the variable it wants to use), it sleeps, until another process release the lock and wake up this process.

Take producer/ consumer problem as example, this may happen, when the stack is empty, and before customer sleep, it pre-empted, so when producer give the wake signal, it is not in the state of sleeping, thus, the signal is missing. So the customer will never wake up, therefore, there is a deadlock. （I think when there is n things, then producer is pre-empted, so count is reduce to n-1 and consumer wake producer, but producer is not sleep yet, so producer will sleep forever, until all the items being used up, customer sleep as well）



**Semaphores**

Is a special lock variable to count the number of wake-ups for future use. Each lock variable has their own semaphore. When a process wants to use this variable, it use DOWN operation, if the semaphore is larger than 0 which means there is resources, so the process can use it. Else, the process will be blocked into queue. When a process release the resource, it will call UP operation, when the semaphore is equal to 0 or less than 0 mean there is process waiting for running, so we pick one and run.

DOWN == TAKE == PEND == P

UP == POST == GIVE == V

Reference code：

// 伪代码定义一个简单的信号量结构

struct Semaphore {

int value; // 信号量的值

Queue waitingQueue; // 等待队列，存放因DOWN操作阻塞的线程

};

// DOWN操作（也叫P操作）

void DOWN(Semaphore \*sem) {

sem->value--;

if (sem->value < 0) {

// 如果信号量的值小于0，表示没有可用资源

// 将当前线程加入等待队列并阻塞

enqueue(currentThread, sem->waitingQueue);

block(currentThread);

}

}

// UP操作（也叫V操作）

void UP(Semaphore \*sem) {

sem->value++;

if (sem->value <= 0) {

// 如果信号量的值小于或等于0，表示有线程在等待资源

// 从等待队列中唤醒一个线程

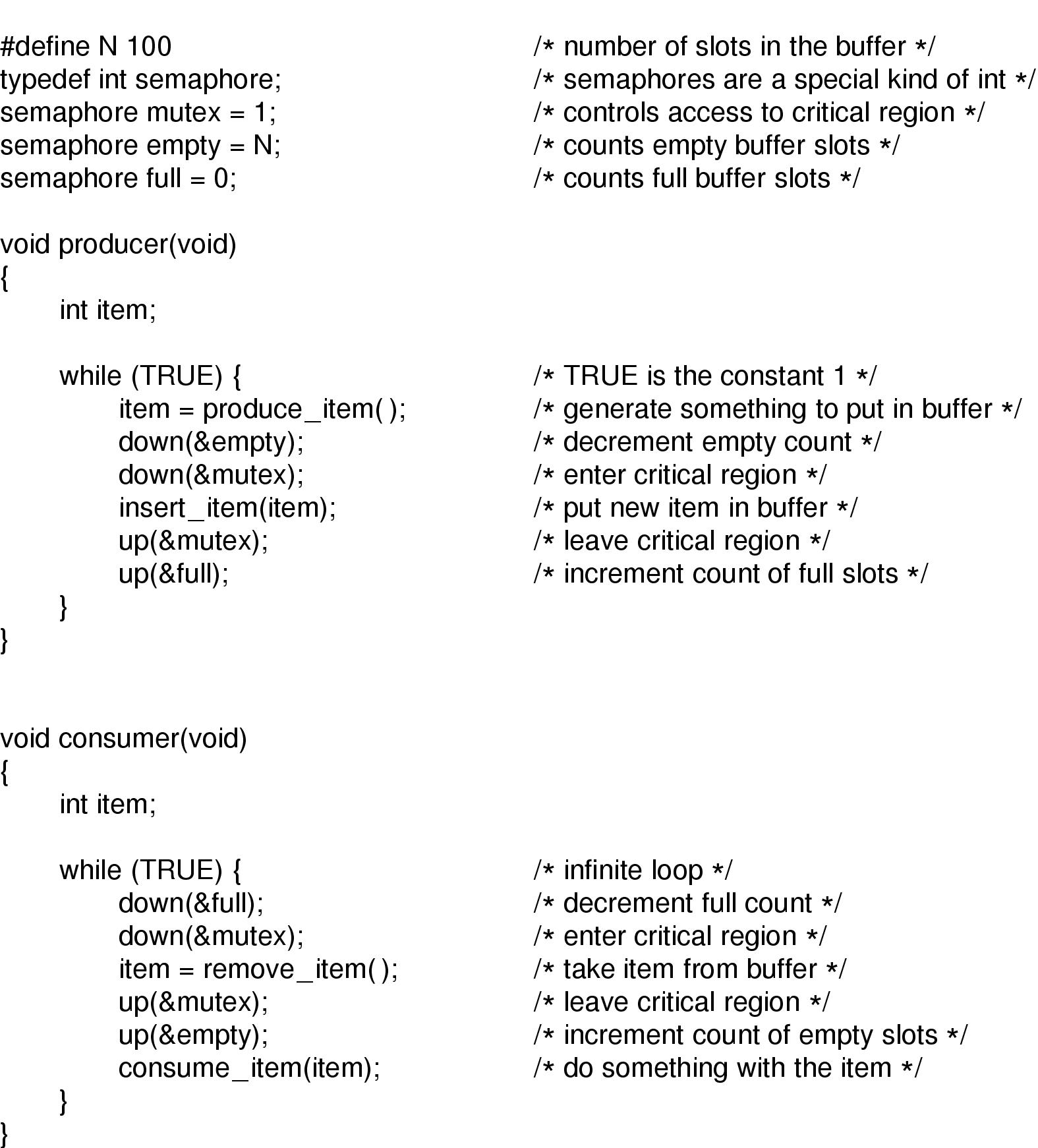
Thread thread = dequeue(sem->waitingQueue);

unblock(thread);

}

}

So the producer and consumer question will become like



**Realize semaphores by TSL/XCHG**

0 = unlocked, 1 = locked

Mutex lock加锁

mutex\_lock:

TSL REGISTER, MUTEX ; 将MUTEX的值加载到REGISTER并设置MUTEX为1

CMP REGISTER, #0 ; 比较REGISTER与0

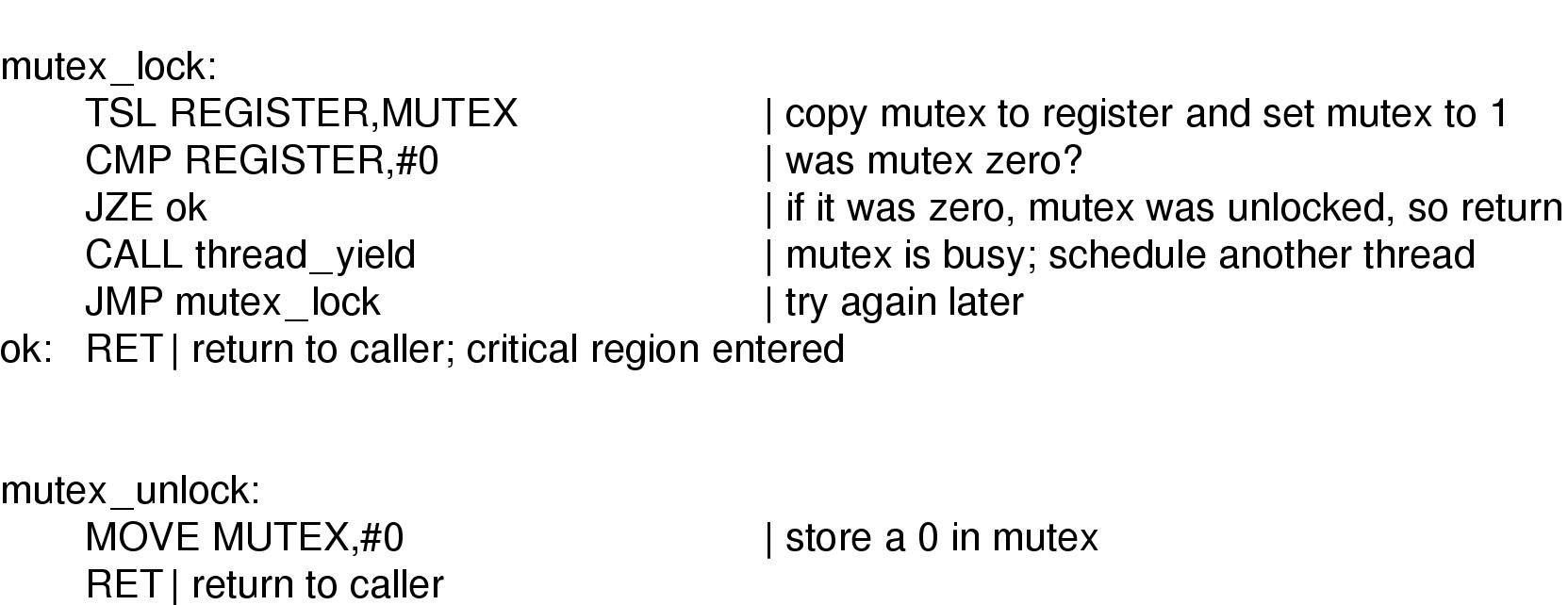
JZE ok ; 如果REGISTER为0，跳转到ok（锁未被占用）

CALL thread\_yield ; 让出CPU给其他线程

JMP mutex\_lock ; 重新尝试获取锁

ok:

RET ; 返回到调用者，进入临界区



**3 conditions make deadlock happen**

(1) Mutual exclusion (2) hold and wait: a process hold one resource and ask for another (3) circular wait: 2 process wants each others’ holden resource( 2 processes must be blocked on each other)

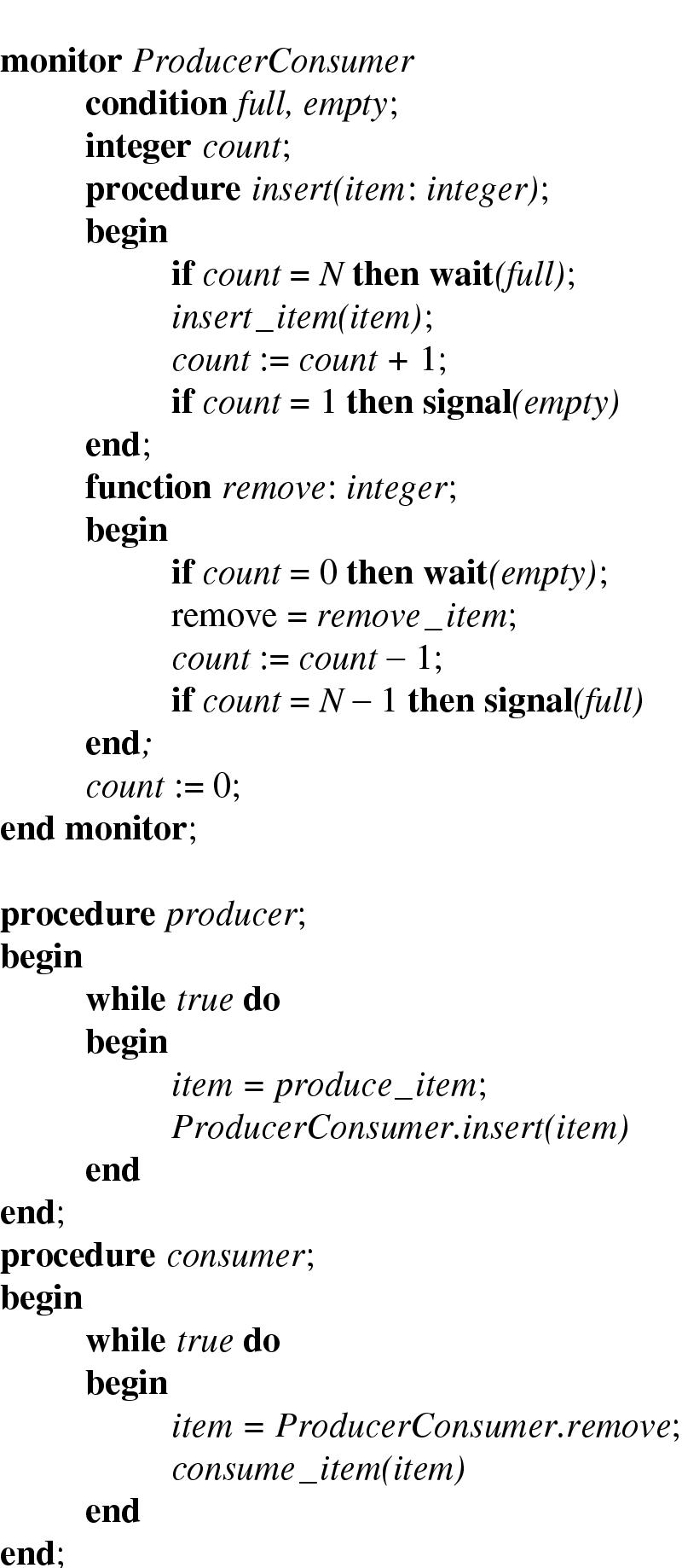
**Monitor**

Monitor is a what makes the method, resources together, and each time, there is only one process can use monitor to change something. While the reason dead lock happen roots from the behavior for each process hold the resources each other needs and unwilling to give away, the monitor can bound the resources together to avoid the situation like this. (for example, like the printer, there will no situation like one process has the printer and another process has the file and thus causes a dead lock)

**More about producer/consumer model**

So instead of just use mutex to save this problem, consumer/ producer problem actually involves something deeper, which is, it not only involves two process deals with the same variable at the same time, it also uses one process to inform another process what to do!

How can we do that?



Why the even the signal may loss, but this still works?

Because the signal is in the condition variable within the monitor, so when give the signal, another process must enter the monitor to see the change of the condition variable, that is, condition variable helps to save the signal.

**5. memory management**

**Basic knowledge**

(1) it happens between memory(instant) and operation system

(2) In single operation time, the amount of data the CPU can handle is more than one byte, which is the smallest storage unit in memory, so in what order should CPU fetch data?

-big endian: higher order bytes at lower address, lower order byte at lower address

(3)logical address and physical address

(4) as all the program data, process data are in memory, conflicting addresses, access violations can happen, so we need operation system to manage memory to ensure everything works normal.

**Mapping between logical memory and physical address**

(1) base register(start address for program)

(2) limit register: length of the memory segment

Physical address = base address + logical address

Segmentation faults: when there is memory access out of the process’s own partition.

Partitioning issue: fragmentation

Internal fragmentation: Partition is larger than needed

external fragmentation: many small chunks are not sufficient to fulfill requests

**For every process:**

Os allocate space for instruction text segments

Os tells the global variables as program’s variable initialized data/BSS segment

Os create stack to store local variables stack

Os create heap to create dynamic variables heap

Their address from small to large is:

Text

Initialized data segment

Uninitialized BSS segment(0)

Heap

Stack

Argument & environment variables

Kernel

**Take control over the location of the data**

Bit map: use 0 to indicate the free memory

Free/allocated list: use double-linked list to record units, with start address + length, P for allocated and H for free

Memory has been already divided into fixed sized chunks, called “allocation units”

# Units = size of requested memory/ memory per unit

Allocation policy: First fit/ best fit(useless hole)/ worst fit

Buddy allocation( quick fit) efficient: binary splitting: 1) cut the block into half when allocate 2)when two buddy blocks are free and are beside each other, coalesce

In detail: avail[i] means free block of size 2\*\*i. when avail[i]==null, no space

It will start from 0, but also able to start from non-zero

**6. File system**

**Hierarchical view**

**Directory management: map id to filename**

Filename = name+ extension+ type (in header, includes basic types and application-specific types)

File directories: in tree structure

Information of entry (each of them has their advantages and disadvantages): all descriptive information/ symbolic name and pointer (to descriptor)

Store entry: array(fixed-size)/ linked list(variable-size)

Store directory: if need expand, may use B+ tree (For example in windows 2000)

(For example, dos directory is a special file consisting of directory entries)

basic operation:

assume：

* + descriptors are in a dedicated area (a specific area)
  + directory entries have name and pointer only（the second way to indicate entry）

create\rename\delete\copy\change protection

entry == name + pointer to descriptor in directory

access：find entry 🡪 find descriptor🡪 find data block

create：find free descriptor and enter attributes –> find free slot in directory, enter name+pointer

**Basic file system: open/close file**

Process：get file descriptor 🡪 manage open file table(OFT)

File descriptor includes: owner id, file type, protection information (the right to write/read or execute), mapping to physical disk blocks, time of creation, last use, last modification, reference counter

Mapping to physical disks: where does the data block store

Reference counter: the times of the file to be opened

Filesystem is also at the kernel of os

How to get OFT

Search directory for given file

Verify access right

Allocate OFT entry (find the table, and add the entry for the later information about the file)

Allocate read/write buffer (buffer is in memory, we can load the information in disk into buffer)

Fill in OFT entry (fill the information about the file into the allocated entry)

Initialization (current position)

Information from descriptor (file length, disk location)

Pointers to allocated buffers (where in the memory)

Return OFT index (the location of file’s information in memory)

What is OFT:

OFT is for the currently opened file not for those is not opened

If the file is just open and not has its own buffer: create buffer for this file and find the data in disk of this file

If the file is open and already exist some data in buffer: find the location in memory if the data in memory or find the location in disk if not in buffer

If currently writes into a file: when save, make the information newly added or updated into the disk (or when the file close or buffer is full)

// close command

Flush modified buffers to disk

Release buffer

Update file descriptor

Free OFT entry

More details

Pointer is in buffer/ file open or close’s information is known by file system (file system create OFT)

When OFT is full, file system will expand its size

The OFT index can also treat as the file pointer in OFT, this pointer points to the entry in OFT, this entry tells the location of buffer and file related information. However, for a file, it may have more than one pointer in buffer.

**Device organization methods: map file data to disk blocks**

How the disk organized?

Boot (at the start, master boot record, MBR): help to boot the os, also includes the information about the disk partition

Partition: each partition has its own file system program ,basic unit is sector/block. We have 4 real partition for PCs in disk, but we can have more if we use logical partition.

In each partition:（for example, in DOS system）

MS DOS Boot (contains code for bootstrapping the os)

FAT (track the usage of blocks on the disk)

FAT duplicate(optional)

Root directory (which will be load to memory and become the directory)

information in entry (fat directory entry)

Filename + extension (8+3 bytes)

Attributes (1 byte: readable, hidden, system) and length

directory flag (distinguish directories from normal files), archive, volume label (disk volume name is dir entry)], time+date created, last access (read/write) date, time+date last write (creation is write)

Pointer to first block

Data blocks

When we use logical partition, that is the a collection of logical blocks(which doesn’t have the same size as the physical sector/block)

However, we may have the block the same size of sector in some simple schemes, in system like MSDOS, the logical block can be even larger and called cluster.

Why we want the logical block not as the same size as the physical block? Because sometimes our file may be much larger than one block, and logical block enable us to deal the I/O consecutive, which makes it faster. Also, sometimes the file can also be smaller than the sector, so logical block can avoid internal fragment

How to allocate disk?

Continuous organization (simple implementation, fast sequential access, but inset/delete is difficult/ external fragment)

Linked organization (simple insert/delete, no external fragmentation, but sequential access less efficient, direct access not possible, poor reliability (when one part of the disk break, all others cannot use))

How we know where is the data we need in the disk?

except file content, we also store file description in a data structure in disk, to find the block quickly

Typical data structure: list/ trees/ arrays

And FAT is one of these data structures

FAT (file allocation table) file system

0. as we mentioned before, in file descriptor, there will be information called “mapping to physical disk block”, this is usually represented as a pointer to the starting disk block or sector where the file’s actual data begins. (In FAT, this is the index to the starting block of the file in the FAT)

1.MSDOS use File allocation table

2. FAT use linked list to show how the data is organized on the disk.

3. FAT is in RAM (stored in disk but duplicated in RAM)

4. things stored in FAT: block number (of next block) 、EOF code (NULL pointer)、FREE code（the block is unused）, BAD block(block is unusable)

5. 1entry is one block

How to delete file/directory?

1) set first letter in filename to 0xE5 (destroy the first byte of original filename)

2) free data blocks, according to FAT, set the related entries to FREE

How FAT16 works?

(1) each entry in the file allocation table is 16 bits, which means they can represent the number up to 2^16-1(65535)

So it can at most refer to 65536 blocks. If sector size = 512byte, then the overall size it can cover is: 2^16\*512 byte = 2^6\*512 kb = 2^15kb = 2^5 mb = 32MB

If disk is larger than 32MB, what to do?

logical block size = multiple of sectors, which is called cluster size

In FAT16, maximum cluster size = 32K, if we use the number to denote cluster, then the overall size can be: 32K\*2^16 = 32\*2^6 M = 2G

We can use this for FAT16 to denote more space, but in this way, it may cause large internal fragmentation

Other FAT:

VFAT: add long filenames (255 chars, introduced in Win95), it manages 2 kinds of names, old SW uses short name

FAT32: increase each entry size to 28bits, so it can denote more sectors (if we assume the size of each sector is 515b, 2^28\*512b = 2^7G = 128G) or more clusters (if we assume the size of each cluster is 32KB, 2^28\*32KB = 8TB) on the disk.

Which is much larger, however, each OS system set some limitation, Win98 127G limit, Win2K 32G limit.

Also, compared to the settled size of cluster in FAT16, the FAT32 allows different size of clusters, which leads to less internal fragmentation

It also removes the root directory limitation in FAT16, and the maximum file size is 2^32-1

Disk fragmentation

Logical contiguous blocks can be far apart on disk (not the same concept as memory fragmentation)

Solution:

FAT(MSDOS): run defragmentation to move all blocks to be contiguous. One big free space chunk after defragmentation, which may take a long time to run.

Unix S5FS: also has disk fragmentation (can be even worse than DOS with smaller logical block size)

Other file system (physical organization methods): indexed organization

Index table:

1) sequential list of records

2) Unlike FAT store in the first few parts in the disk, the index table is stored in the descriptor (keep index list in descriptor, or in unix file system, for example, s5fs may be called, inodes)

3) easy to insert/ delete and read. File size limited by number of index entries

Multiple-level index hierarchy: primary index points to secondary indices (for example, s5fs)

Incremental indexing: fixed number of entries at the top-level index, only when this fixed number is insufficient, allocate additional index level (for example, unix, 3-level expansion)

Structure of inode

1) inode is a file object

2) it contains Meta-data about file except filename

Meta-data includes Table of Content(TOC): gives mapping of file data to disk blocks

3) reference count