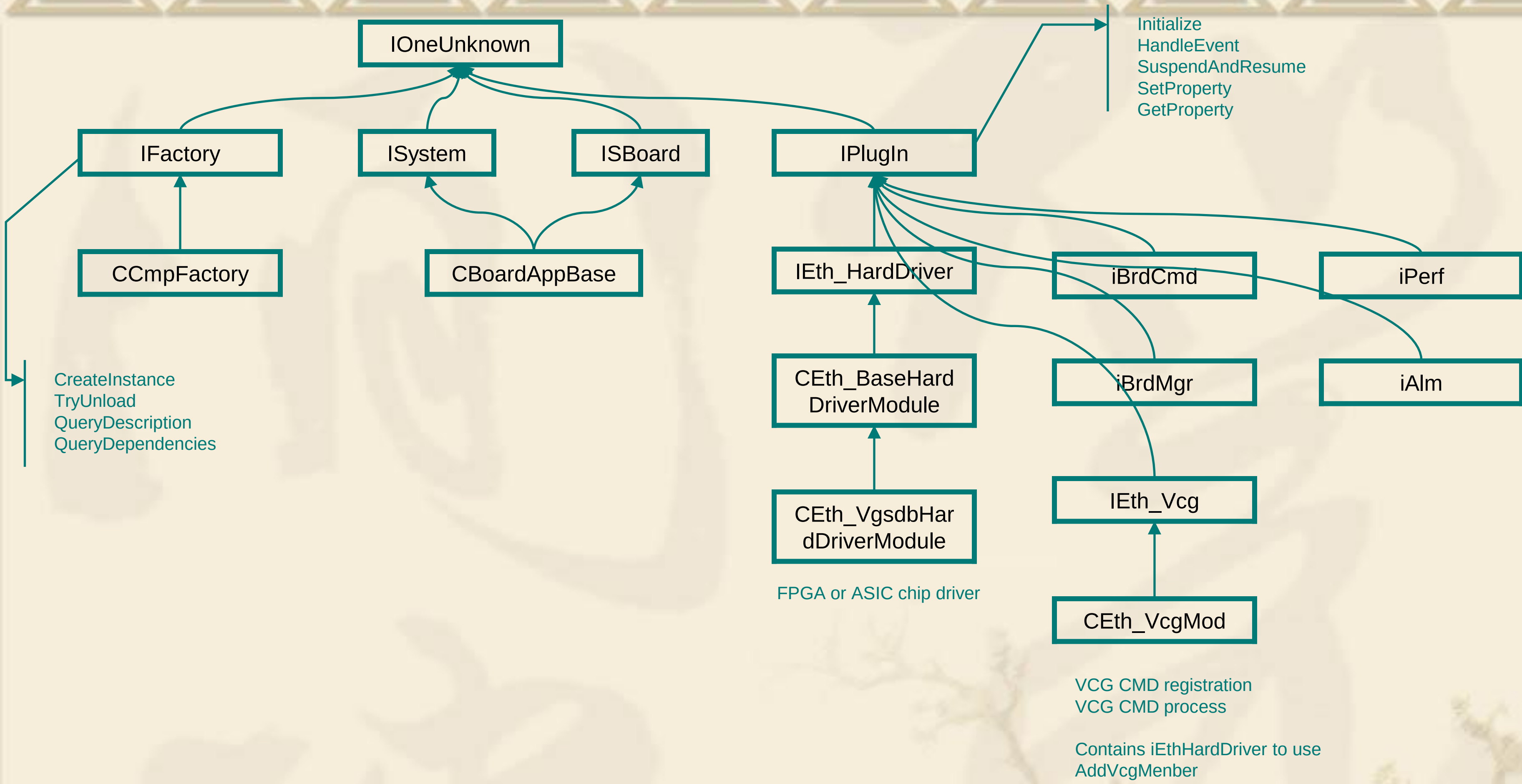
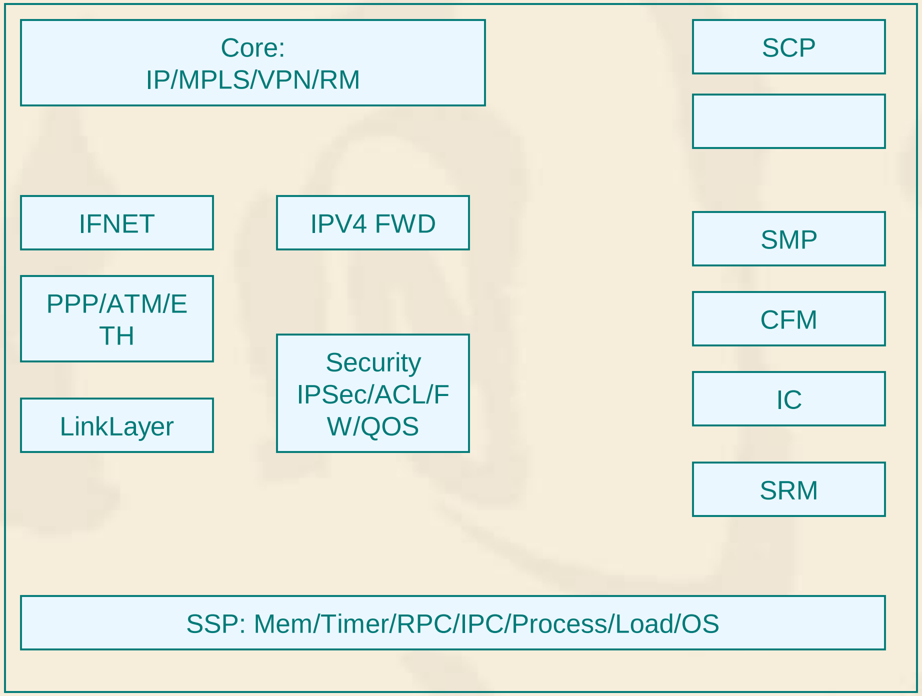
1. A COM application (Harbour Networks)
2. An Network OS (Huawei VRP)
3. C++ advanced features (OOP, C++11)
4. SOCKET&TCP illustration (NetBSD)
5. OS concepts

Harold Mei


IOneUnknown
Initialize
HandleEvent
SuspendAndResume
SetProperty
GetProperty
IFactory
ISystem
ISBoard
IPlugIn
CCmpFactory
CBoardAppBase
IEth_HardDriver
iBrdCmd
iPerf
CreateInstance
TryUnload
QueryDescription
QueryDependencies
CEth_BaseHard DriverModule
iBrdMgr
iAlm
CEth_VgsdbHar dDriverModule
IEth_Vcg
FPGA or ASIC chip driver
CEth_VcgMod
VCG CMD registration
VCG CMD process
Contains iEthHardDriver to use
AddVcgMenber


Core:
IP/MPLS/VPN/RM
SCP
IFNET
IPV4 FWD
SMP
PPP/ATM/ETH
CFM
Security IPSec/ACL/FW/QOS
IC
LinkLayer
SRM
SSP: Mem/Timer/RPC/IPC/Process/Load/OS

1. Encapsulation:

public, private, protected;
All are inherited, but the accessibilities for those three are different.

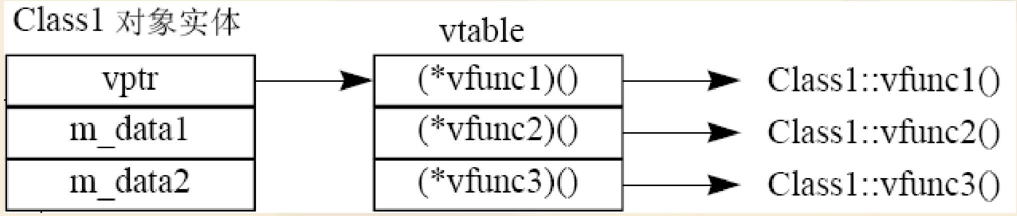
2. Inheritance:

Overload: same function name, different parameters. Could be class members or independent functions.
Overwrite: virtual member functions, **polymorphism & RTTI**.
Override: for non-virtual member functions.

Virtual member function: can be called by a base class object pointer, if the pointer refers to objects of a class which the function is implemented.
Non-virtual member function: can not be called by a base class object.

3. polymorphism

A base pointer refers to a descendent, hence when similar logic process (generalization) need to call a virtual function from the base class, it will call the **nearest** overwritten virtual function from the exact object class.



4. static member

Even though static member is defined in class, it is independent of any objects, and it should be initialized outside the constructor.

5. Constructor & Destructor

Copy constructor:
Assignment constructor: avoid default constructors.
Operator xx():
Implicitly and explicitly called in programs.

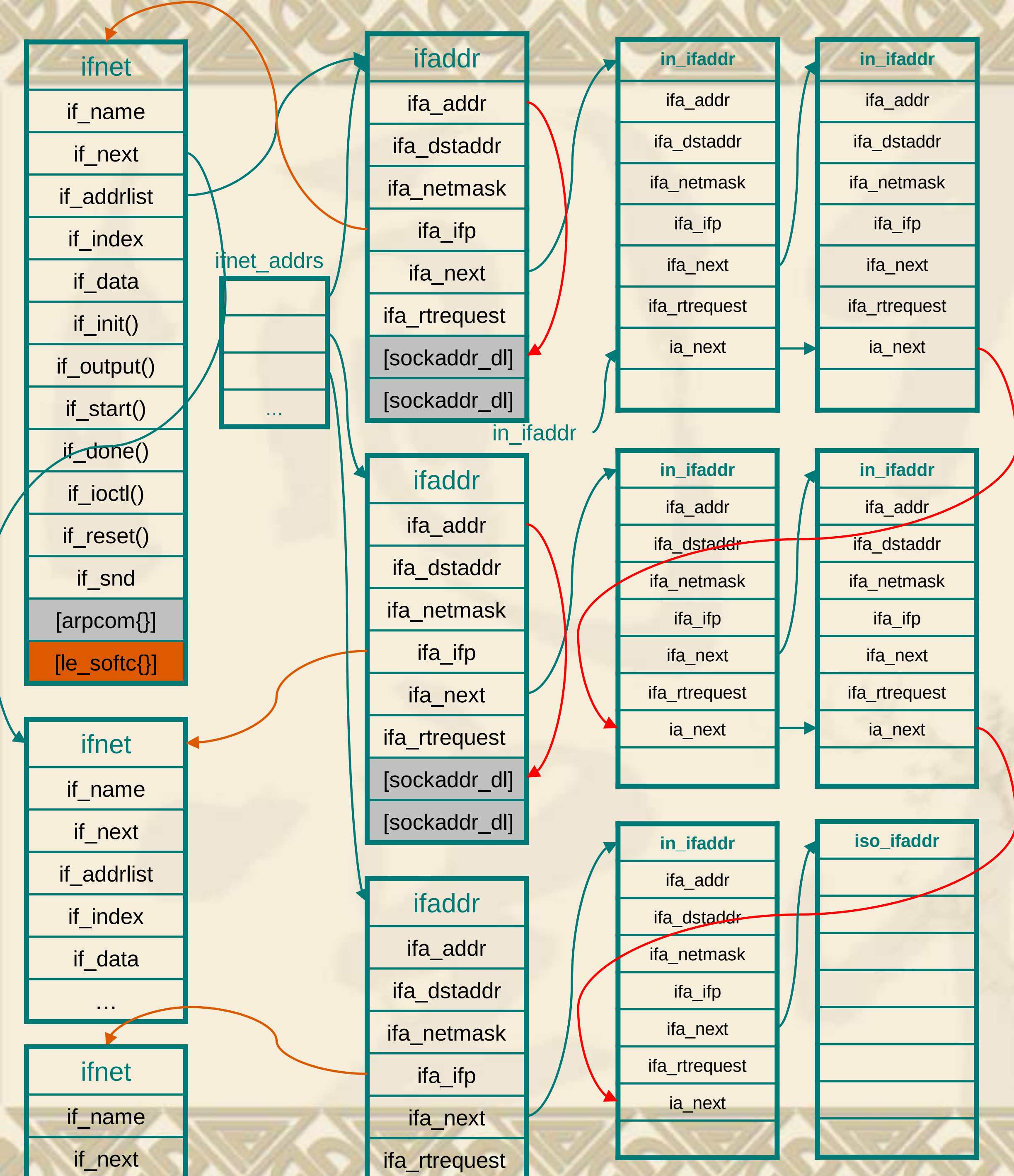
6. Exception handling

7. Template

*1. Memory type

Static memory; stack; heap.
volatile;

1、泛化(继承)
2、实现
3.1单向关联
3.2双向关联
3.3聚合关系
3.4组合关系
4、依赖




```
main(framep)
void *framep;
{
cpu_startup();
…
for (pdev = pdevinit; pdev->pdev_attach !=
NULL; pdev++)
    (*pdev->pdev_attach)(pdev->pdev_count);
//if_attach(); ether_attach();
…
ifinit();
domaininit();
scheduler();
}
```

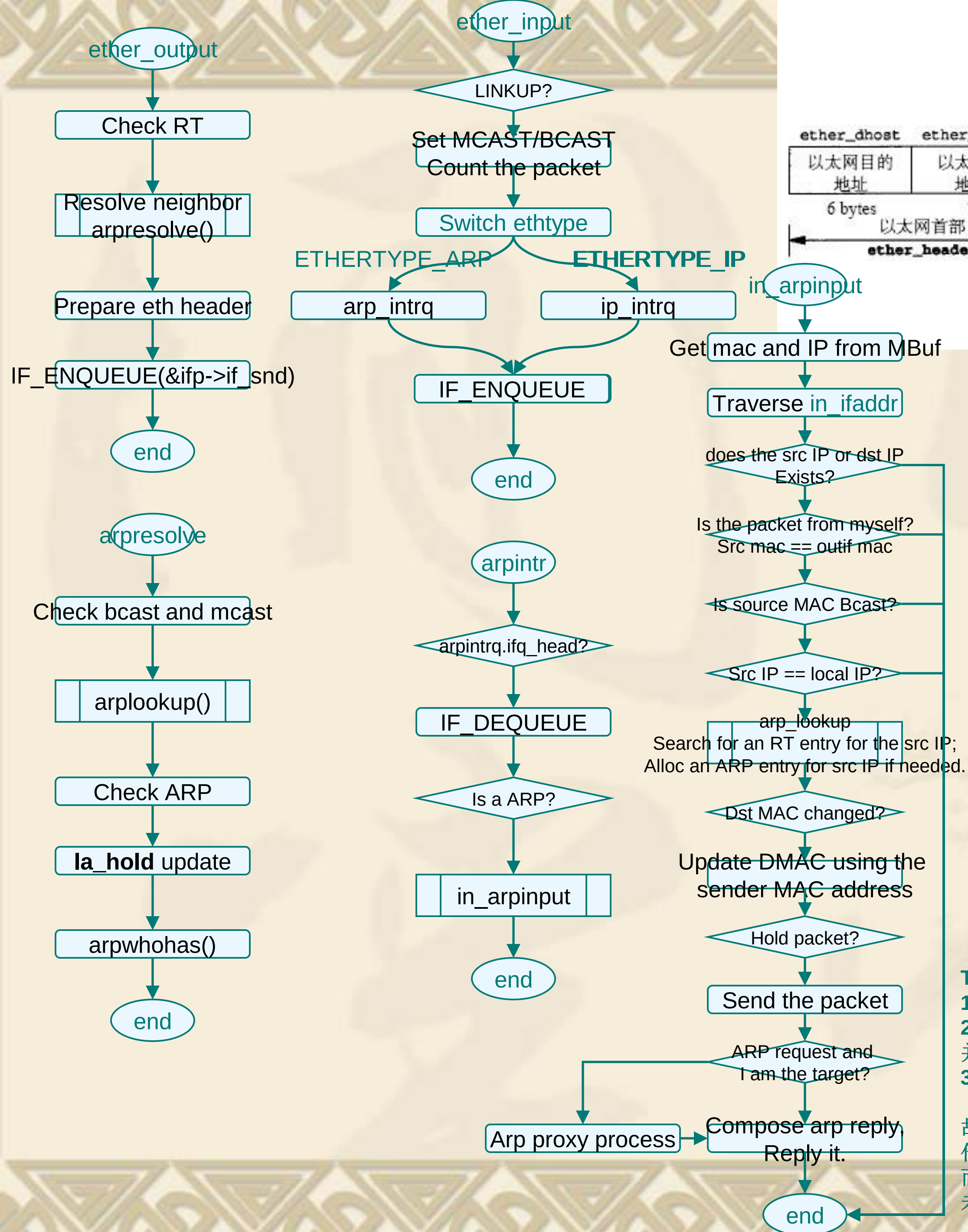
IOCTL命令:

0, SIOCGIFCONF: 获得接口配置
1, SIOCGIFFLAGS:获得接口标志
2, SIOCGIFMETRIC: 获得接口度量
3, SIOCSIFFLAGS: 设置接口标志(LINK status conf)
4, SIOCSIFMETRIC: 设置接口度量
5, SIOCG(S)IFADDR
6, SIOCG(S)IFNETMASK
7, SIOCG(S)IFDSTADDR
8, SIOCG(S)IFBRDADDR
9, SIOCDIFADDR
10, SIOCAIFADDR

sockaddr_dl
sdl_len
sdl_family
sdl_index
sdl_type
sdl_nlen
sdl_alen
sdl_slen
sdl_data[12]

sockaddr
len
family
data (14 bytes)

sockaddr_in
len
family
port(2 bytes)
addr (4 bytes)
zero (8 bytes)



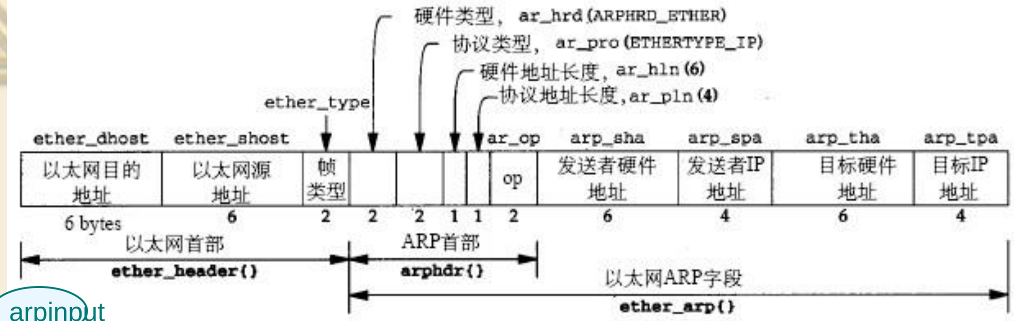
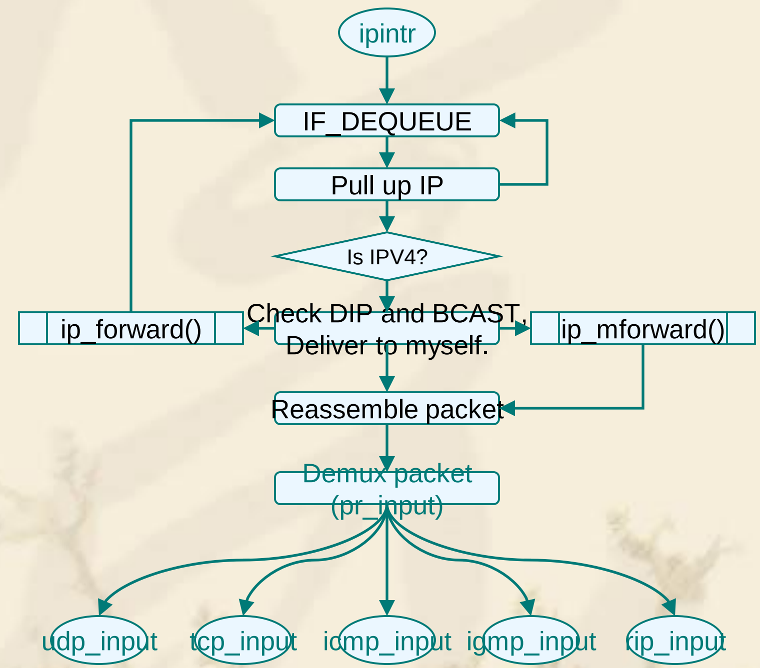


图21-7　在以太网上使用时ARP请求或回答的格式



TCP分段与IP分片区别：

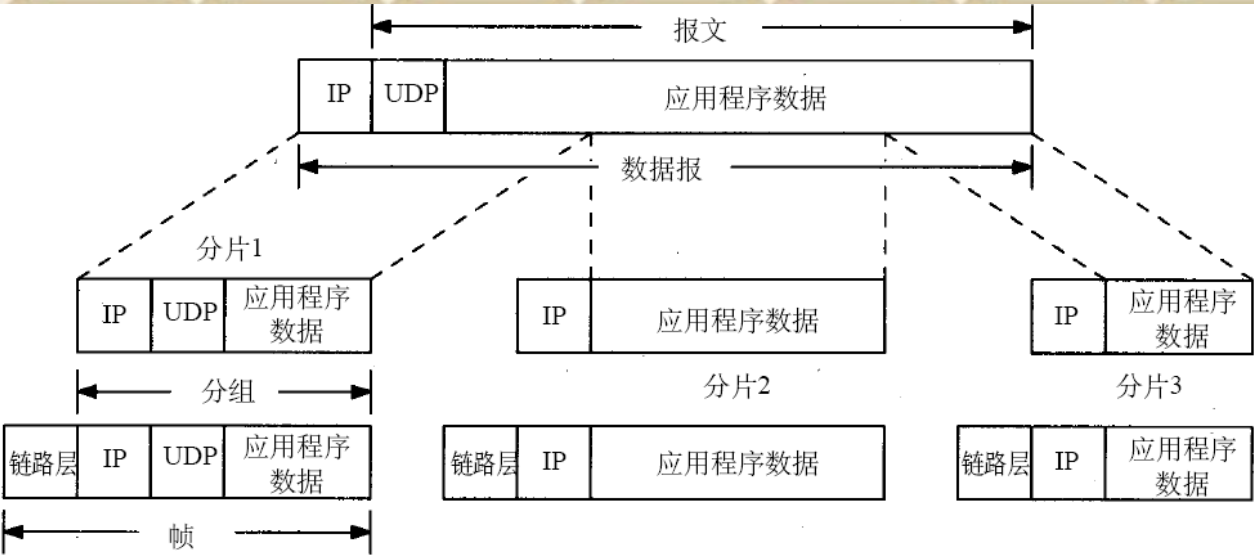
1.IP分片产生的原因是网络层的MTU；TCP分段产生原因是MSS.

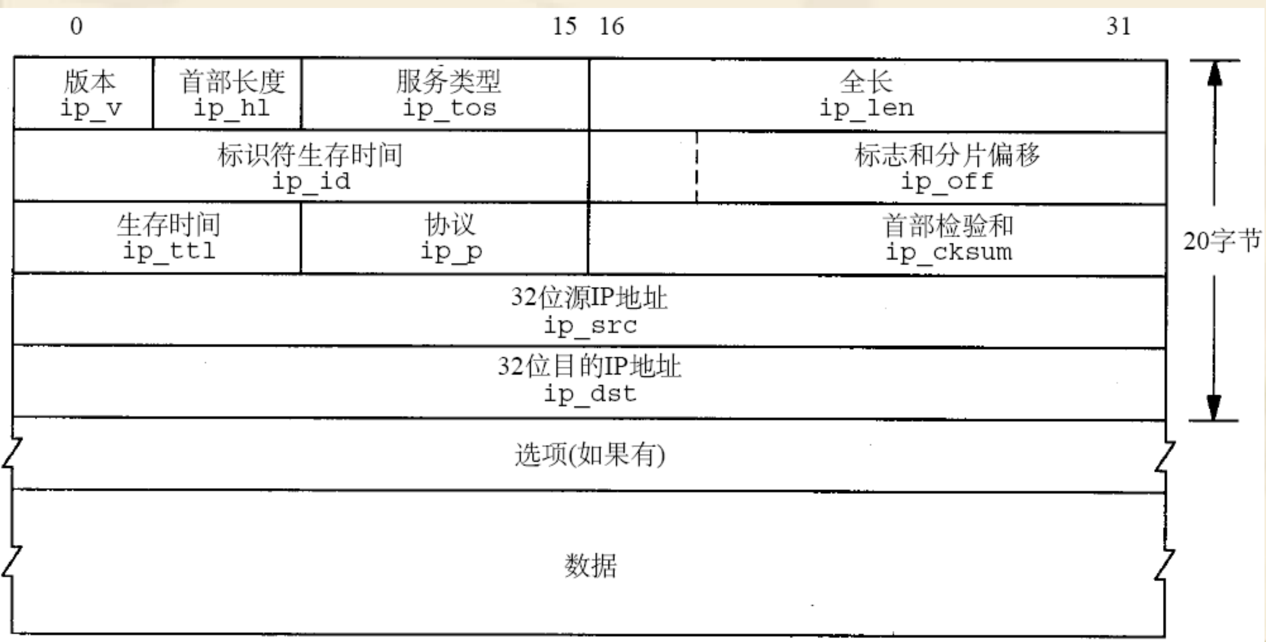
2.IP分片由网络层完成，也在网络层进行重组；TCP分段是在传输层完成，并在传输层进行重组. //透明性

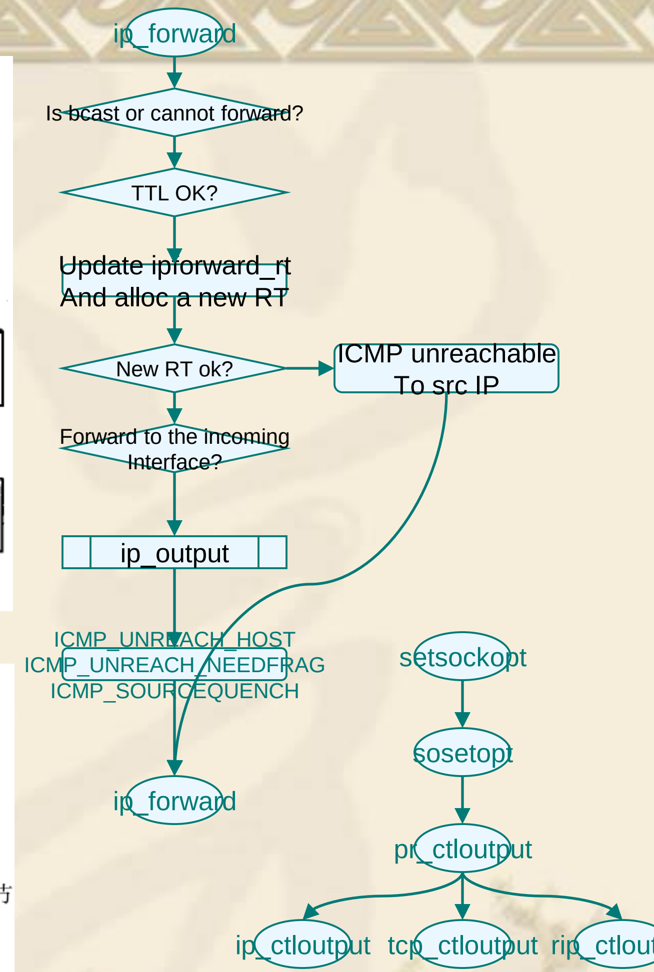
3.对于以太网，MSS为1460字节，而MTU往往会大于MSS.

故采用TCP协议进行数据传输，是不会造成IP分片的。若数据过大，只会在传输层进行数据分段，到了IP层就不用分片。

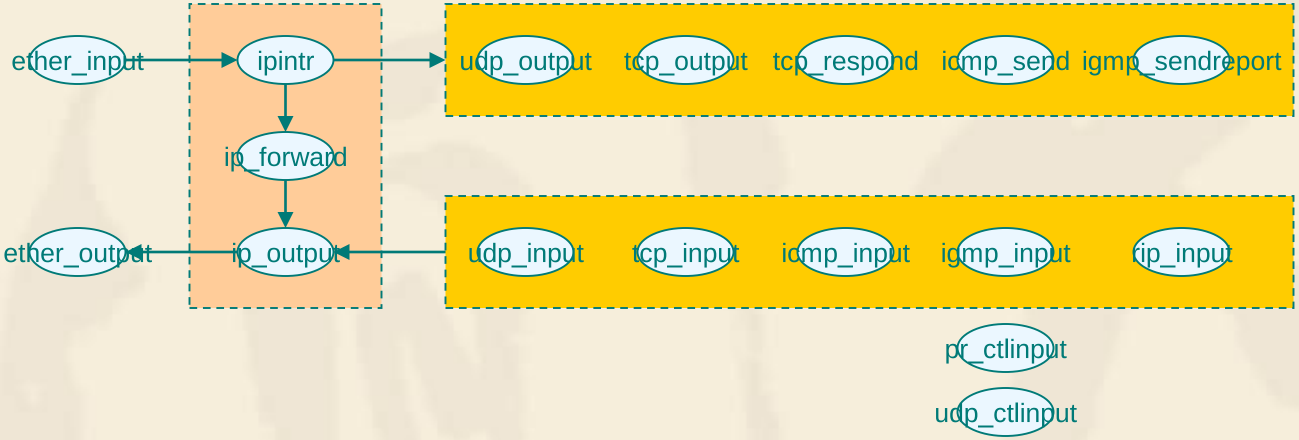
而我们常提到的IP分片是由于UDP传输协议造成的，因为UDP传输协议并未限定传输数据报的大小。

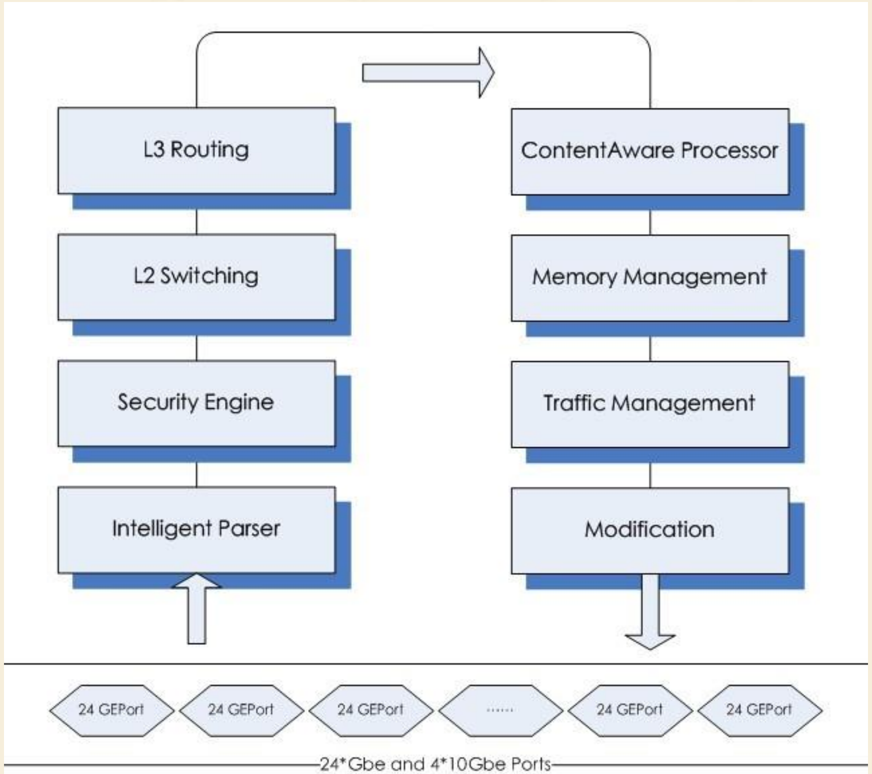


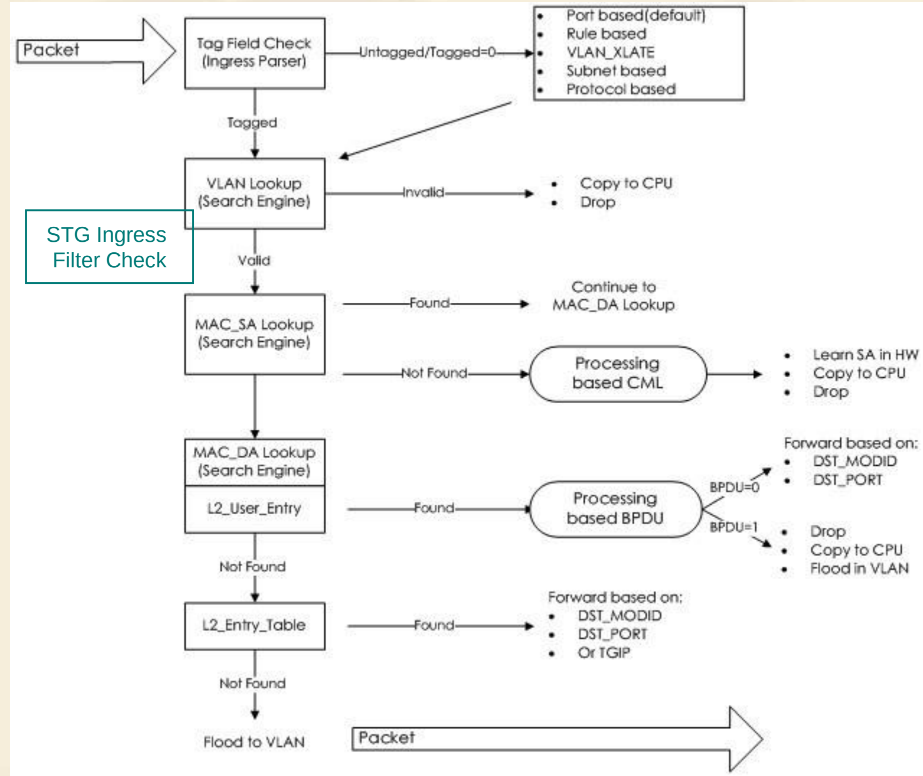


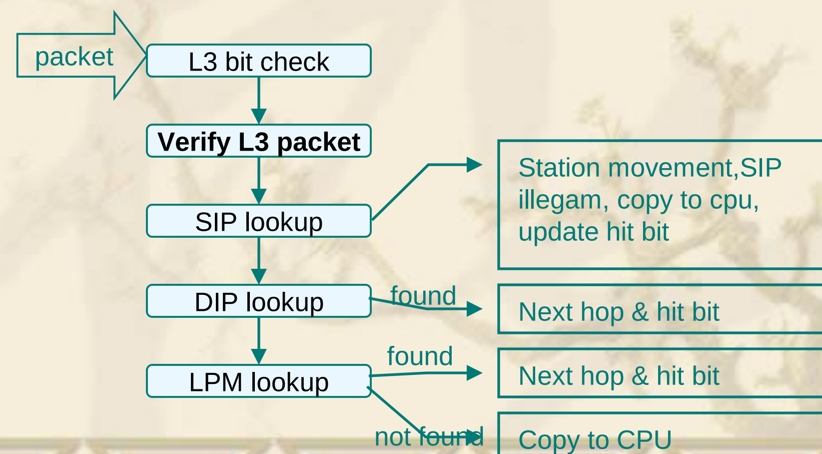


SYSCTL命令:
0, IPCTL_FORWARDING 转发功能使能
1, IPCTL_SENDREDIRECTS ICMP重定向使能
2, IPCTL_DEFTTL 设置默认TTL
3, IPCTL_DEFMTU 设置IP的MTU


ether_input
ipintr
udp_output
tcp_output
tcp_respond
icmp_send
igmp_sendreport
ip_forward
ether_output
ip_output
udp_input
tcp_input
icmp_input
igmp_input
rip_input
pr_ctlinput
udp_ctlinput

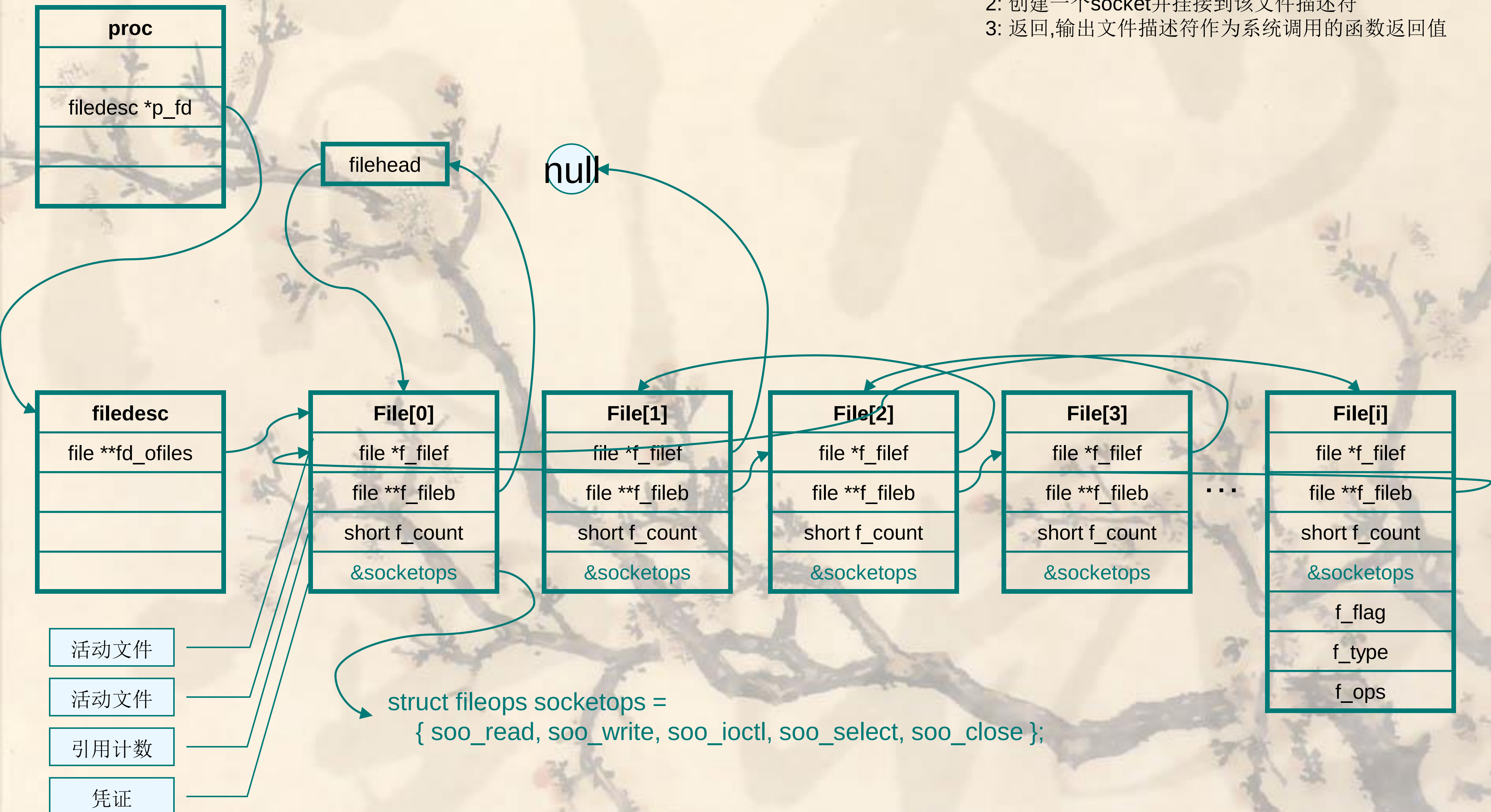

L3 Routing
L2 Switching
Security Engine
Intelligent Parser
ContentAware Processor
Memory Management
Traffic Management
Modification
24 GEPort
24 GEPort
24 GEPort
......
24 GEPort
24 GEPort
24*Gbe and 4*10Gbe Ports


Packet
Tag Field Check (Ingress Parser)
Untagged/Tagged=0
Port based(default)
Rule based
VLAN_XLATE
Subnet based
Protocol based
Tagged
VLAN Lookup (Search Engine)
Invalid
Copy to CPU
Drop
STG Ingress Filter Check
Valid
Found
Continue to MAC_DA Lookup
MAC_SA Lookup (Search Engine)
Not Found
Processing based CML
Learn SA in HW
Copy to CPU
Drop
MAC_DA Lookup (Search Engine)
L2_User_Entry
Found
Processing based BPDU
BPDU=0
Forward based on:
DST_MODID
DST_PORT
BPDU=1
Drop
Copy to CPU
Flood in VLAN
Not Found
L2_Entry_Table
Found
Forward based on:
DST_MODID
DST_PORT
Or TGIP
Not Found
Flood to VLAN
Packet


packet
L3 bit check
Verify L3 packet
SIP lookup
Station movement,SIP illegam, copy to cpu, update hit bit
DIP lookup
found
Next hop & hit bit
LPM lookup
found
Next hop & hit bit
not found
Copy to CPU

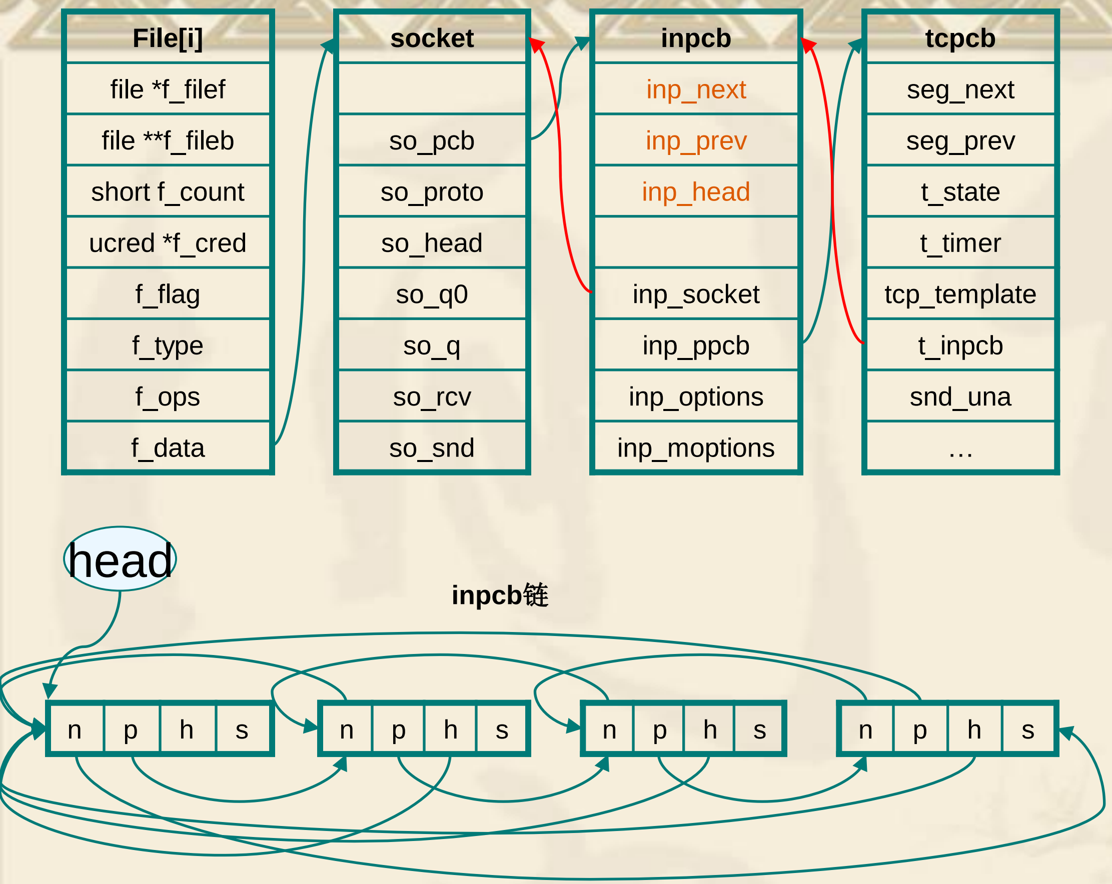
socket(p,uap,retval)

1: 创建一个文件描述符并挂接到进程的活动文件列表
2: 创建一个socket并挂接到该文件描述符
3: 返回,输出文件描述符作为系统调用的函数返回值



falloc(p, &fp, &fd)

socket: socreate



参数:

domain: 所使用协议族

PF_INET	TCP/IP协议;
PF_OSI,PF_ISO	OSI
PF_LOCAL,PF_UNIX	本地IPC
PF_ROUTE	路由表
n/a	链路层

type: socket的类型(协议的通信语义)

SOCK_STREAM	(TCP)
SOCK_DGRAM(UDP)	
SOCK_RAW	(ICMP, IGMP, raw IP)

protocol:

0	IP协议
IPPROTO_UDPUDP	
IPPROTO_TCP TCP	
IPPROTO_RAW	raw IP
IPPROTO_ICMP	ICMP
IPPROTO_IGMP	IGMP

过程:

1, 基于domain, type以及protocol查找protosw;
2, 分配并初始化新的socket;重点是挂接protosw到socket

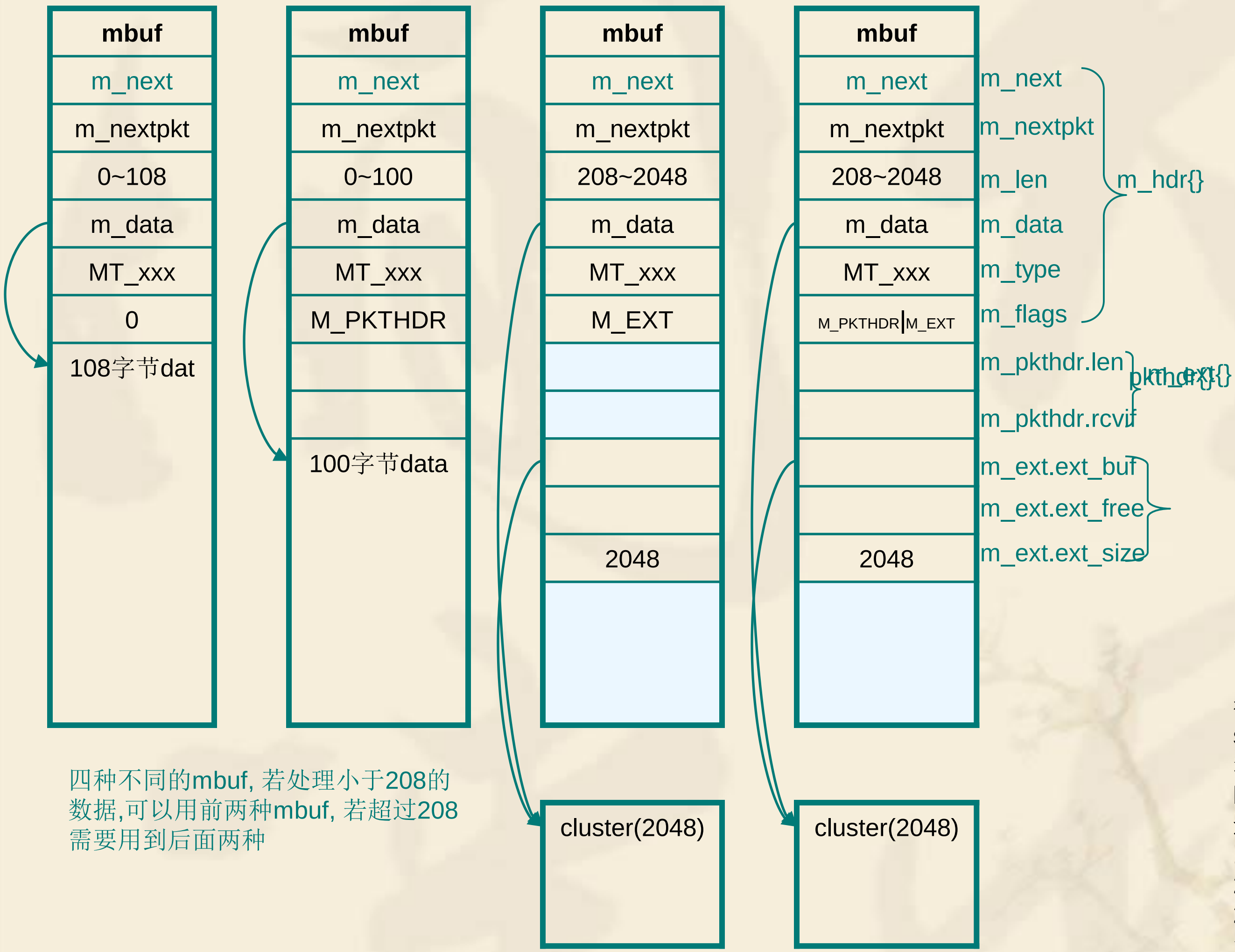
3, 使用具体协议的usrreq进行PRU_ATTACH请求*
3.1: in_pcballoc进行inpcb的创建和插入
3.2: soreserve进行mbuf的大小支配(此时并未实际分配内存**)**

4, 返回,输出参数为新创建的socket

socreate(uap-domain, &so, uap->type, uap->protocol)

bind(p,uap,retval)

参数:
s: socket的文件描述符
name: 传输地址或者主机名 (sockaddr)
namelen:
过程:
1, getsock()获取文件file指针;
2, sockargs()将传入参数复制到内核中的一个新分配的mbuf中 (m_flags=0); mbuf存储的sockaddr格式的数据.
3, sobind进行usrreq请求PRU_BIND.
3.1 in_pcbbind(inp, nam):
说明: 为inpcb设定地址和端口;
A, 接口必须有地址; 插口还未绑定;
B, socket选项是否允许本地地址复用;
C, 是否为请求连接或者接受连接
D, 是否允许通配. 匹配若B,C都真则. wild=INPLOOKUP_WILDCARD;
E, 若为组播地址??
F, 若为单播, 查找一个地址相同的接口
G, 端口不可为保留端口
H, 根据端口和地址查找可用pcb,
I, 判断该pcb是否允许复用端口和地址
J, 若未指明绑定端口号,则从pcb链中查找一个可用的端口号分配给用户使用.

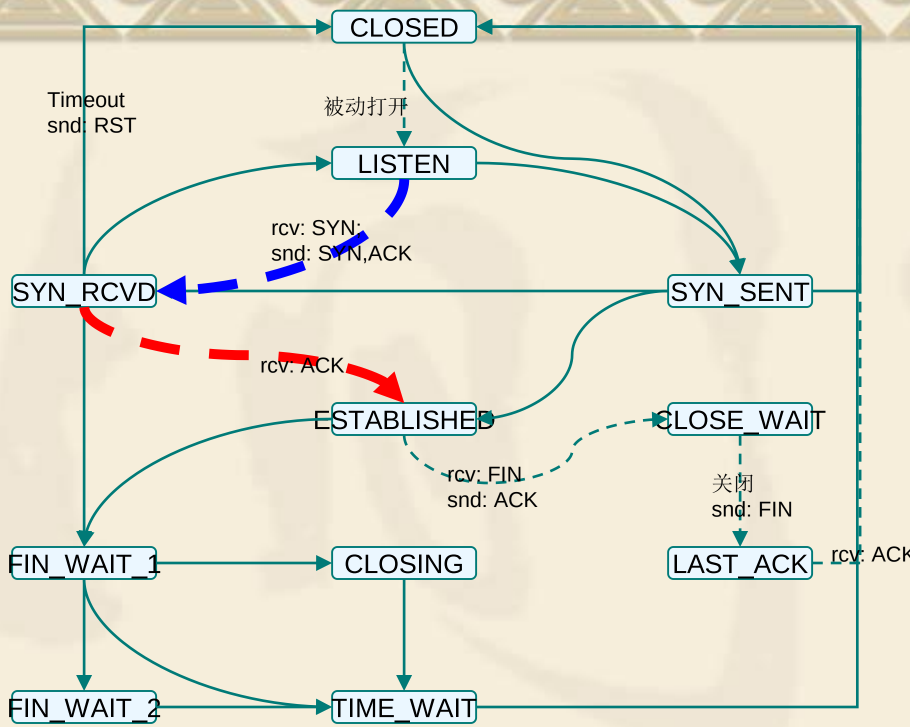


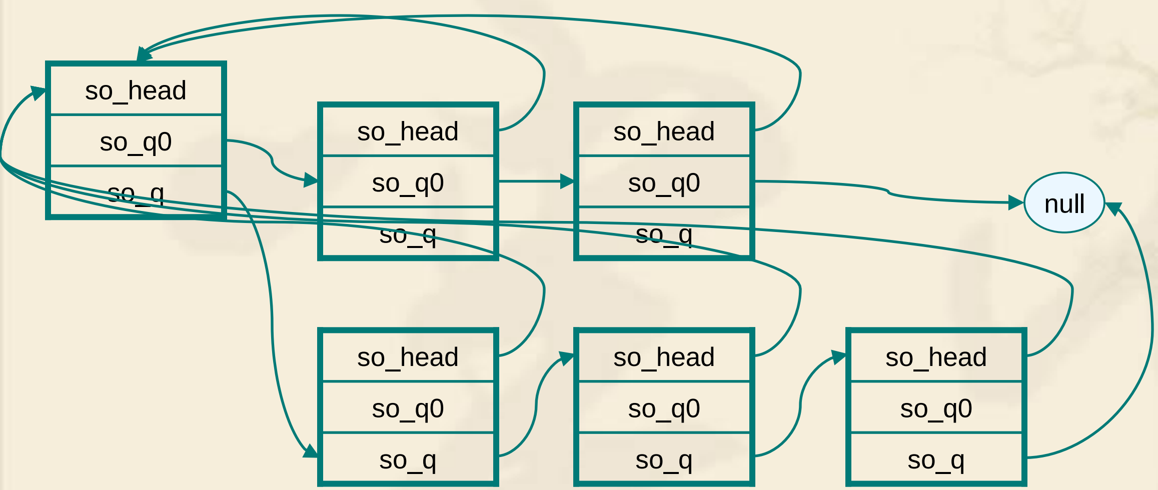
四种不同的mbuf, 若处理小于208的数据,可以用前两种mbuf, 若超过208需要用到后面两种

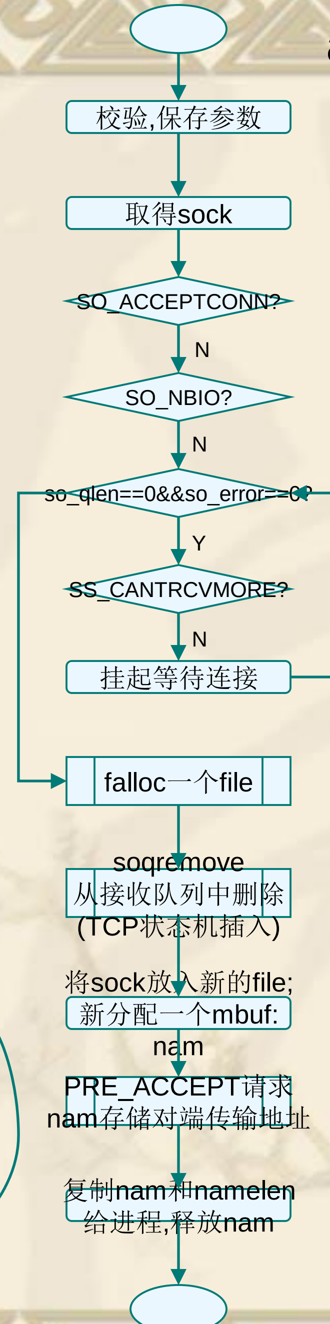
listen(p,uap,retval)

参数:
s: socket的文件描述符,指明哪个socket被设为被动模式;
backlog: 指明socket使用队列的长度
过程:
1, getsock()获取文件file指针;
2, 进行usrreq请求PRU_LISTEN.
2.1 in_pcbbind(inp, nam):
说明: 见bind
UDP不支持listen
与udp不同处:
A, inpcb转化成tcpcb
B, tcpcb设置为监听状态.
3, 设置sock模式为SO_ACCEPTCONN, 设置sock的队列最大长度为backlog.

accept(p,uap,retval)







参数:

s: socket的文件描述符,指明哪个socket被设为被动模式;

name: 缓冲指针, 用来保存外部主机传输地址

anamelen: 缓冲指针长度指针

过程:

1, getsock()获取文件file指针;

2, 进行usrreq请求PRU_LISTEN.

2.1 in_pcbbind(inp, nam):

说明: 见bind

UDP不支持listen,accept

与udp不同处:

A, inpcb转化成tcpcb

B, tcpcb设置为监听状态.

3, 设置sock模式为SO_ACCEPTCONN, 设置sock的队列最大长度为backlog.

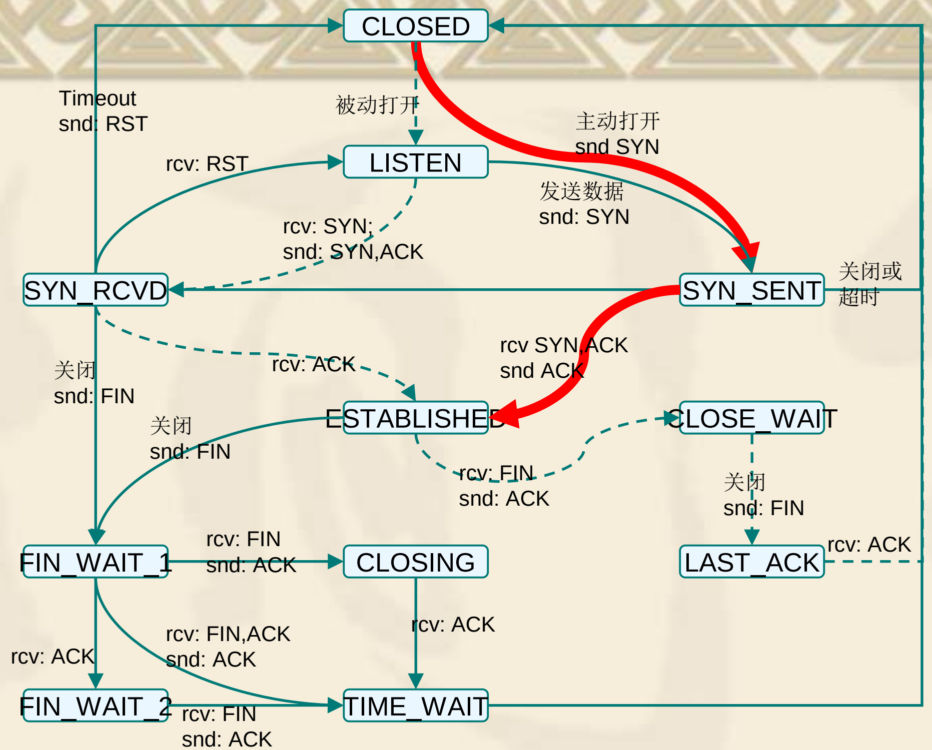
状态变迁说明:

1. 当服务器状态由LISTEN到SYN_RCVD变迁的时候, 新进来的连接将会使tcp_input产生一个新的sock, 并将该sock插入到半连接队列.

2. 只有当服务器状态由SYN_RCVD变迁到ESTABLISHED的时候进入tcp_input的SYN_RCVD分支, soisconnected()函数被调用, 挂起的accept才被唤醒并将sock插入到连接队列

PRE_ACCEPT:

直接通过mbuf返回对端传输地址



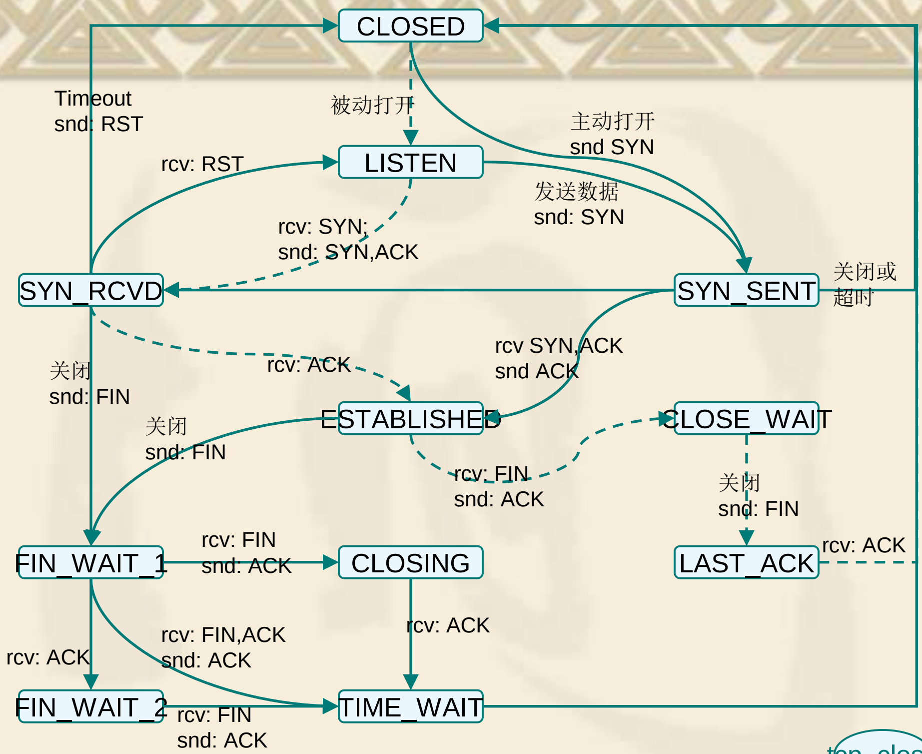
shutdown(p,uap,retval)

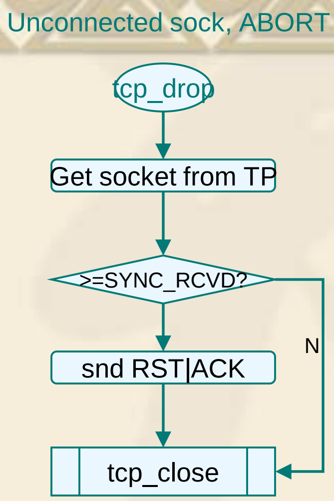
参数:
s: socket的文件描述符;
how: 连接关闭的方式
FREAD关闭连接的读通道,FWRITE关闭连接的写通道,FREAD|FWRITE关闭连接的读写通道.
过程:
1, getsock()获取文件file指针;
2, usrreq-SHUTDOWN
FREAD: sock设置为CANTRCVMORE
唤醒进程;
对于注册了dispose的协议,调用dispose释放资源
FWRITE: sock设置为CANTSENDMORE
唤醒进程;

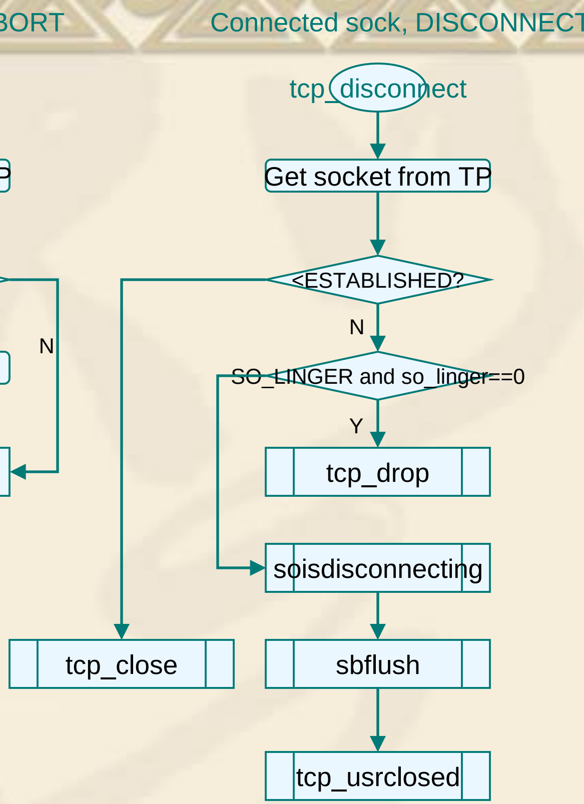
connect(p,uap,retval)

参数:
s: socket的文件描述符;
name: 指向远端地址
namelen: name的长度
connect既可用在TCP也可用在UDP;用于主动初始化一条连接,为客户端函数.
过程:
1, getsock()获取文件file指针;
2, sock状态不能为NBIO和CONNECTING;
3, sockargs()将传入参数复制到内核中的一个新分配的mbuf中 (m_flags=0);
4, soconnect进行usrreq请求PRU_LISTEN.
UDP: in_pcbconnect
A, 校验和转换外部地址
B, 路由判断,首次发送需要建立一条路由并删除老的路由,再次发送可以直接使用已有路由.
C, 选择路由出接口对应的本地地址()
D, in_pcblookup验证插口对的唯一性
E, in_pcbbind隐式绑定本地端口
F, 分配远端地址和端口

TCP:
1, 若为bind则调用in_pcbbind进行本地端口绑定
2, 同UDP, 使用in_pcbconnect对sock指定本地IP和外部端口号
3, tcp_template为连接的IP和TCP首部创建一个模板
4, 计算窗口缩放因子
5, sock设为connecting状态, tcp_connattempt++
6, TCP状态机进入SYN_SENT状态,
7, 初始化TCP报文序号,
8, tcp_output发送TCP报文, 此时由于处于SYN_SENT, 发送SYN报文到服务器
9, tsleep等待连接建立, 并取消sock的connecting状态.
10, 从服务器端收到对应的ACK, 发送ACK并进入ESTABLISHED状态并抹掉sock的CONNECTING状态.(此时CONNECTED被TCP状态机置位)







close(p,uap,retval)

参数:

s: socket的文件描述符;

过程:

0, 对于未连接的sock, usrreq-ABORT 释放q,q0

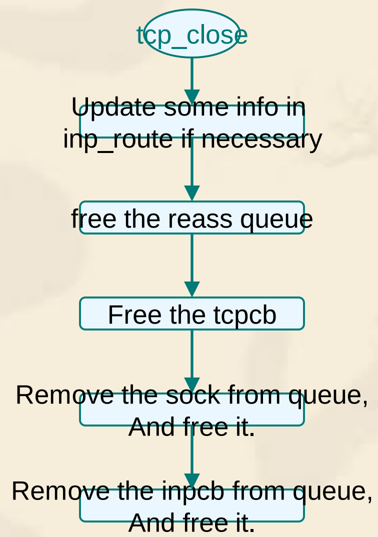
1, 对于已连接的sock, usrreq-DISCONNECT

2, 如果是connected状态则挂起等待

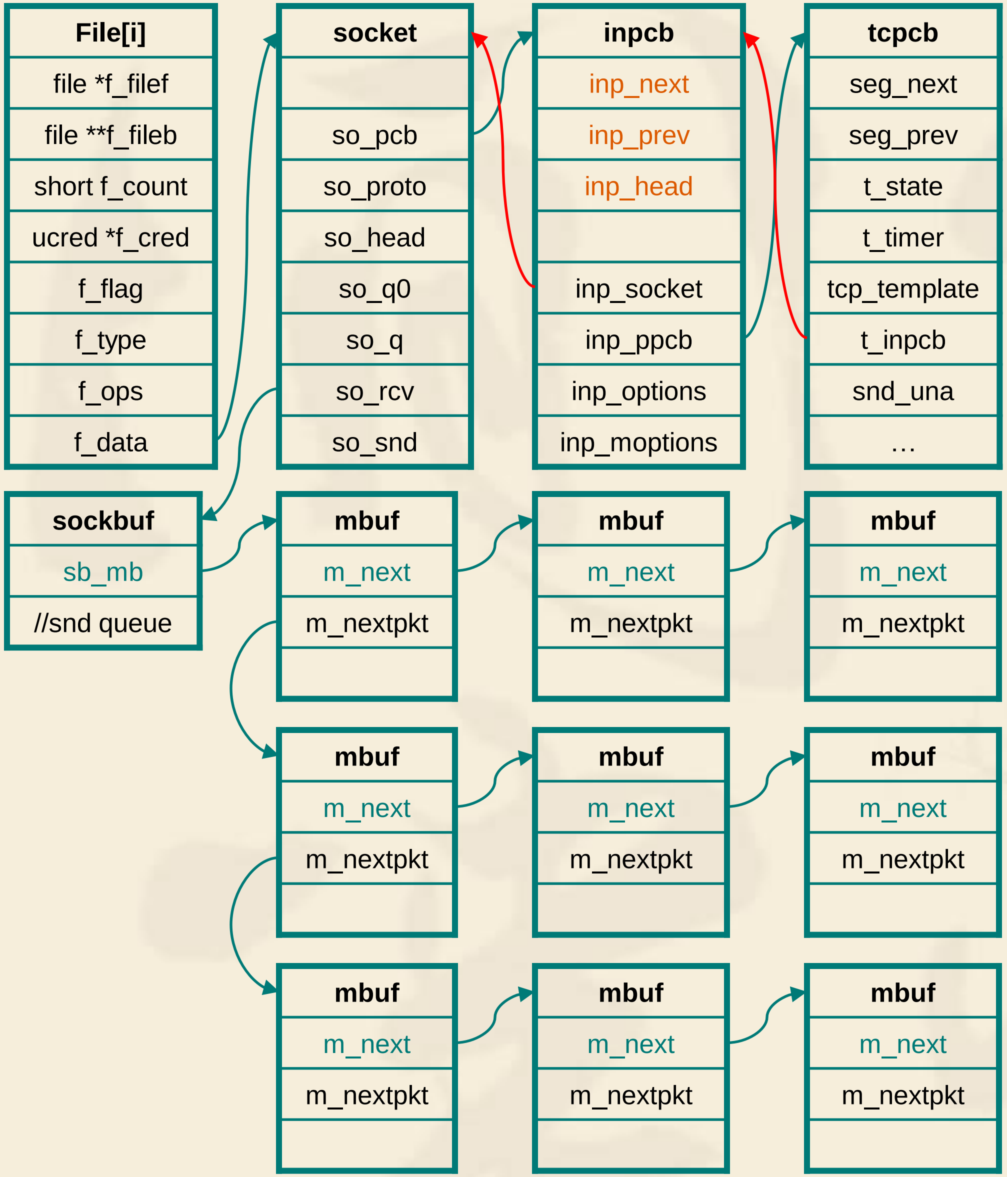
3, usrreq-DETACH,

4, 释放sock

5, 释放file



sock I/O



read
Read data to recv queue;

recv
Read data to recv queue, with options

write
Write data to snd queue

send
Write data to snd queue, with options

select
Waiting for I/O events

```
#define TH_FIN    0x01 //finalize
#define TH_SYN    0x02 //sync, init a connection
#define TH_RST    0x04 //reset
#define TH_PUSH   0x08 //push to app
#define TH_ACK    0x10 //ACK on
#define TH_URG    0x20 //urgent pointer on
```

#define TCP_MSS 512 //Maximum Segment Size, to avoid fragmentation

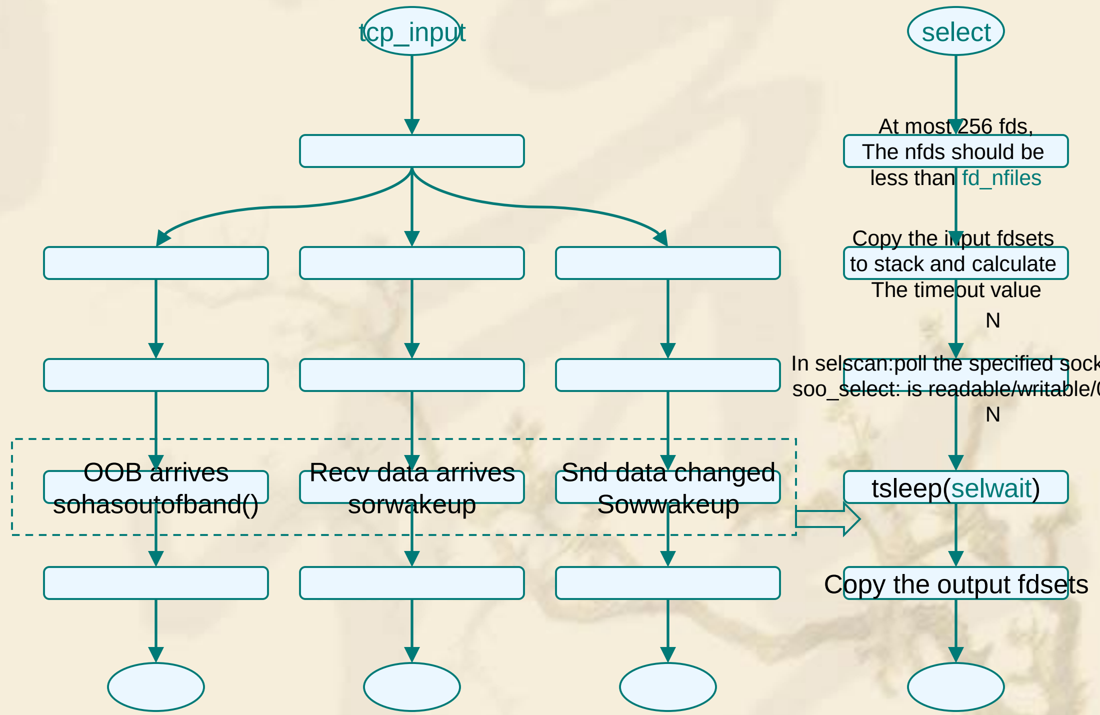
#define TCPTV_MSL (30*PR_SLOWHZ)
//max seg lifetime.
Usage: 1, When TCP is in TIME_WAIT state, if receives a RST, drop the connection. If it receives another FIN, the timer should be reset and wait for another 2MSL.
2, Quiet time. When TCP reset, no connection can be established in MSL.

select

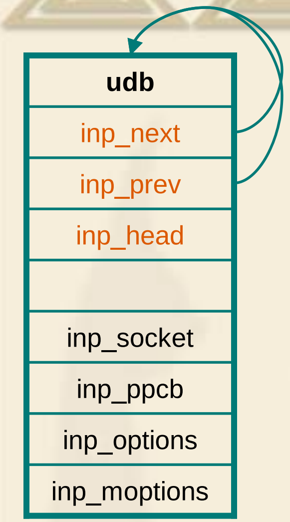
select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

Params:
nfds: max handle
fd_set: three types of monitored fd;
timeout:

Compare with poll/epoll



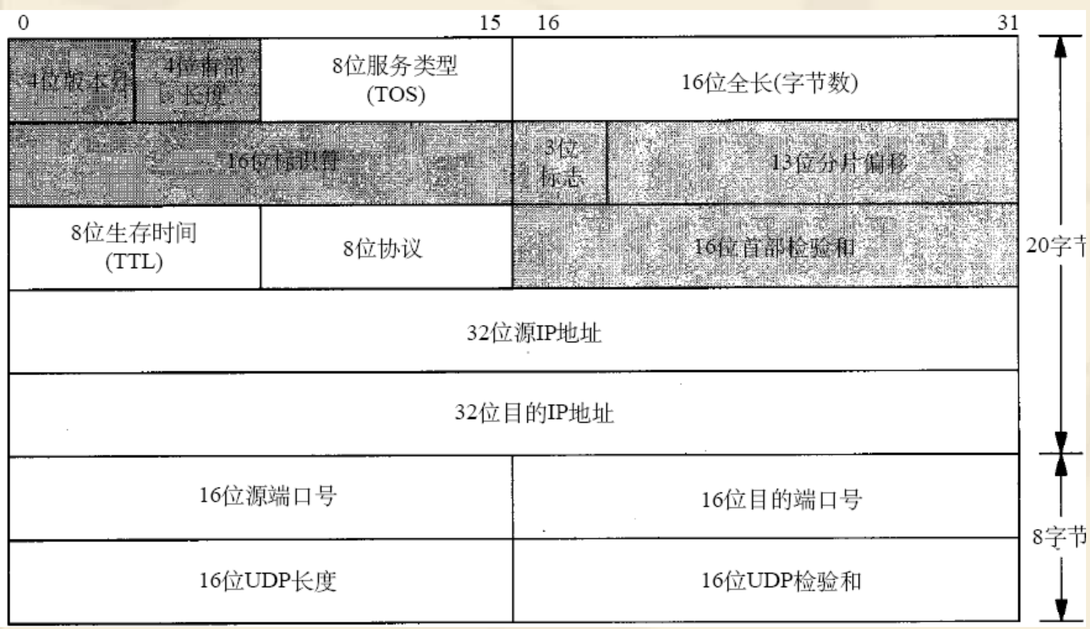
UDP

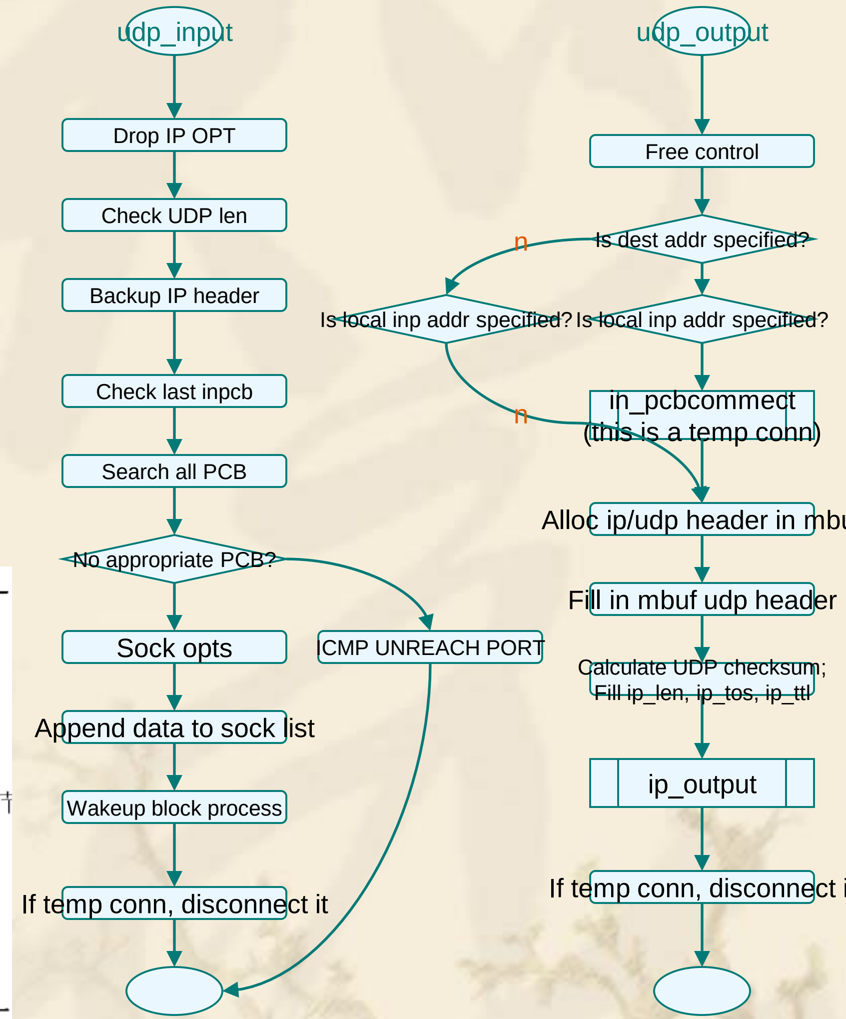


udp_init()

SYSCTL命令:
0, IPCTL_FORWARDING 转发功能使能
1, IPCTL_SENDREDIRECTS ICMP重定向使能
2, IPCTL_DEFTTL 设置默认TTL
3, IPCTL_DEFMTU 设置IP的MTU

4, UDPCTL_CHECKSUM 使能UDP检验和





TCP

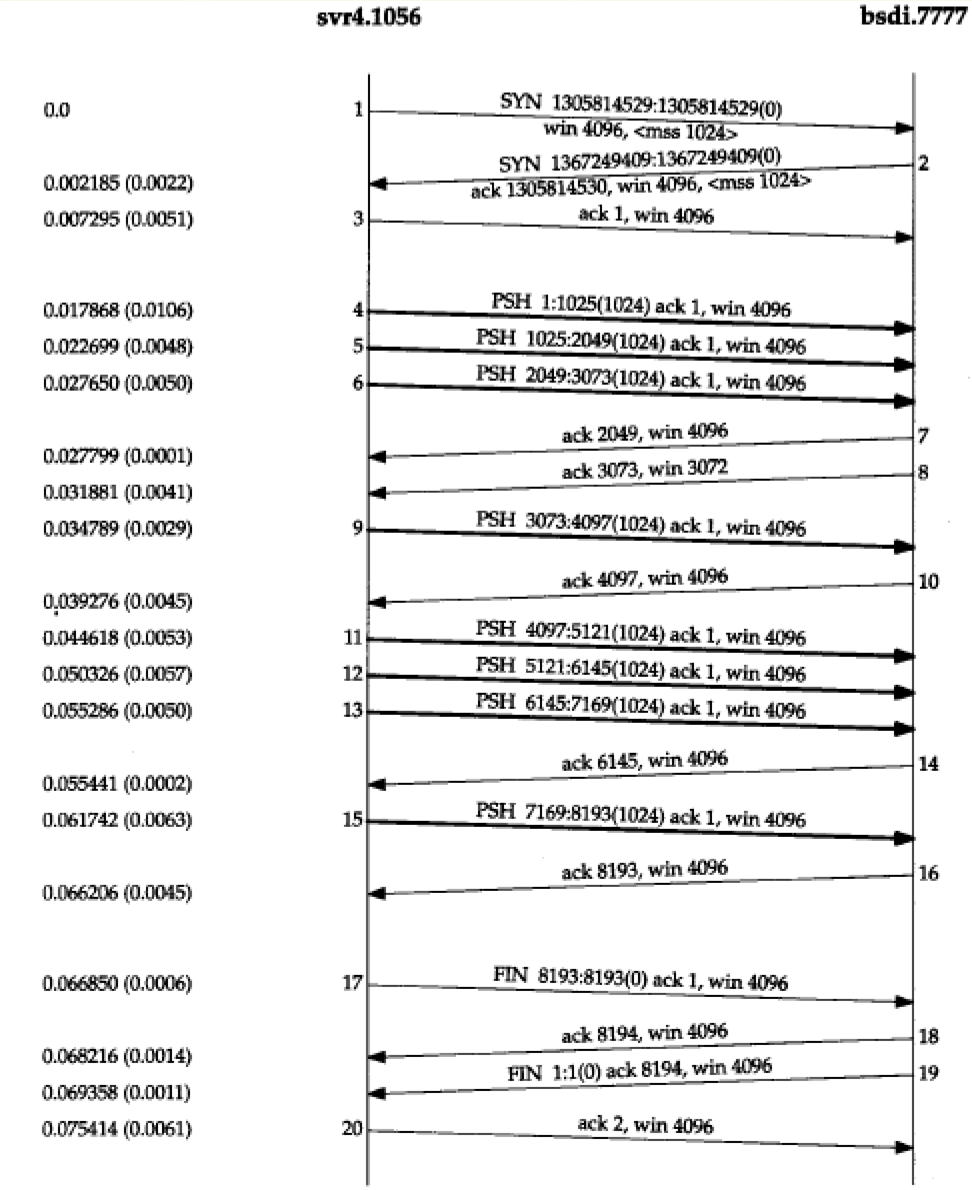


图20-1 从svr4 传输8192个字节到bsdi

name	Description
snd_una	The first unacknowledged no.
snd_nxt	The next transmitted no.
snd_wnd	The offered window size
snd_max	The maximum sent no. (snd_nxt<snd_max, retransmitting)
rcv_wnd	Recv window to inform the sender
rcv_nxt	The next recv no.
rcv_adv	the first no. outside window

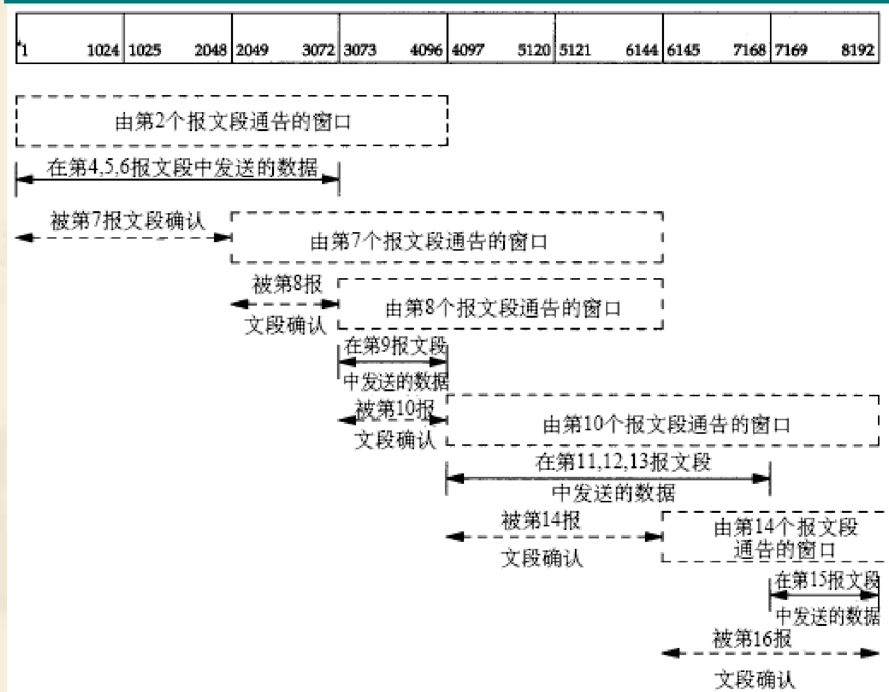
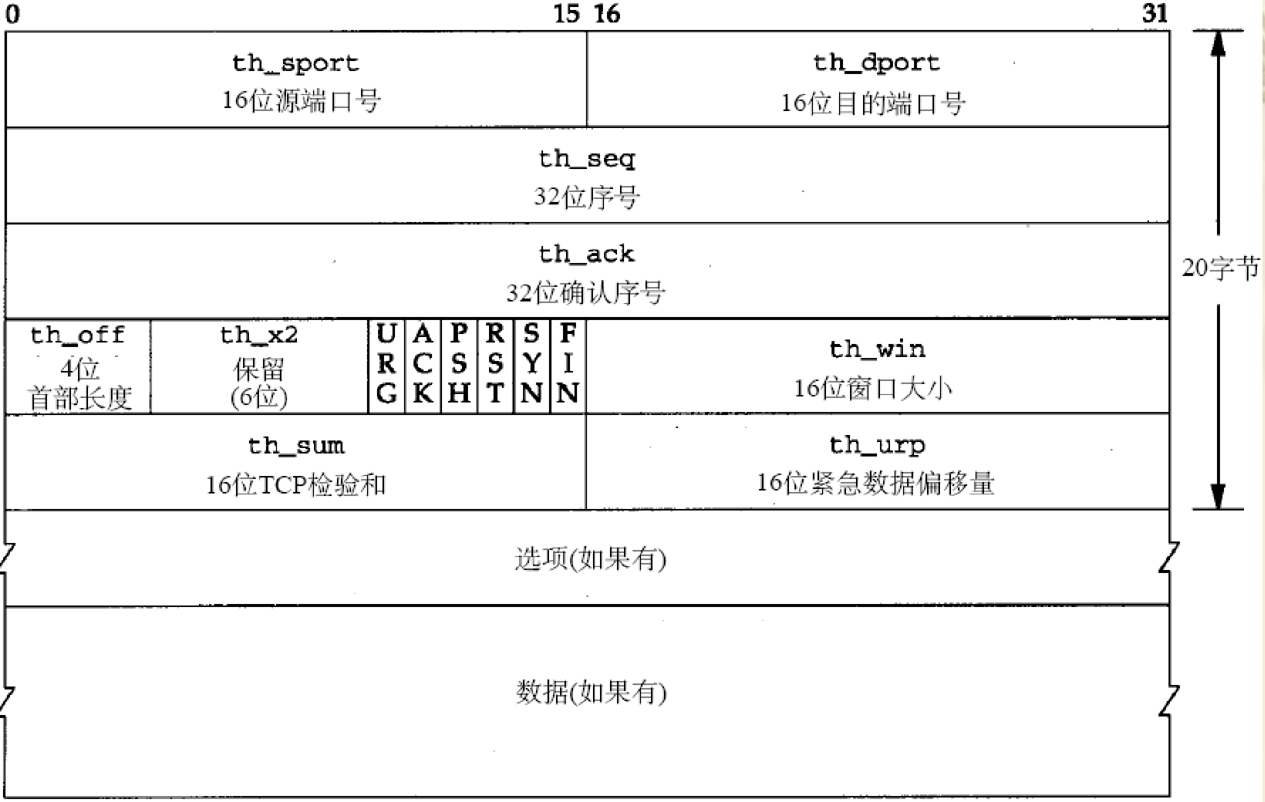
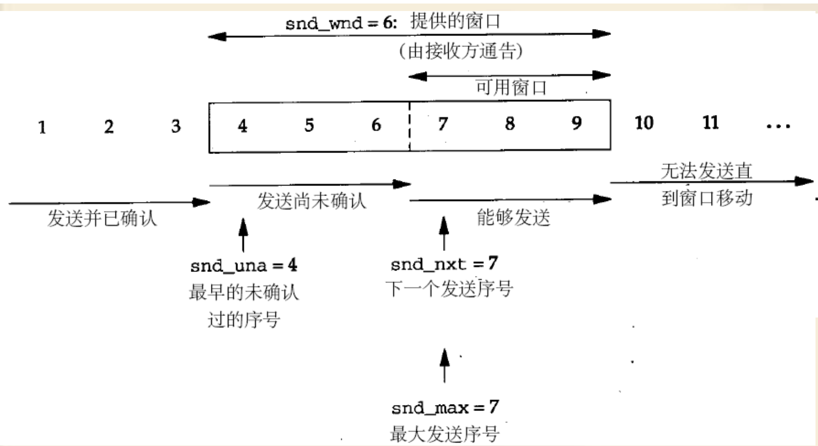


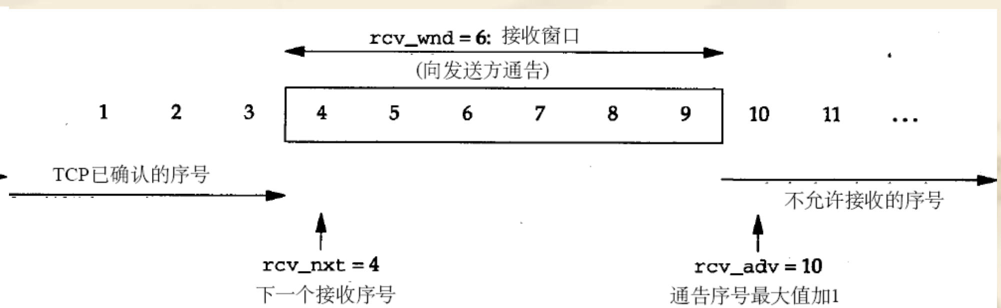
图20-6 图20-1的滑动窗口协议

TCP



timer	description
connection	SYN, 75s
retransmission	**Dynamic, retransmit data**
Delayed ACK	捎带确认 ACK doesn't need to be confirmed immediately. (fast timer)
persist	**Win=0, send 1 byte, dynamic** **Test the availability of the peer's window size.**
keepalive	SO_KEEPALIVE, alive test
FIN_WAIT_2	10m+75s, close called Avoid the losing FIN from peer.
TIME_WAIT	2 MSL



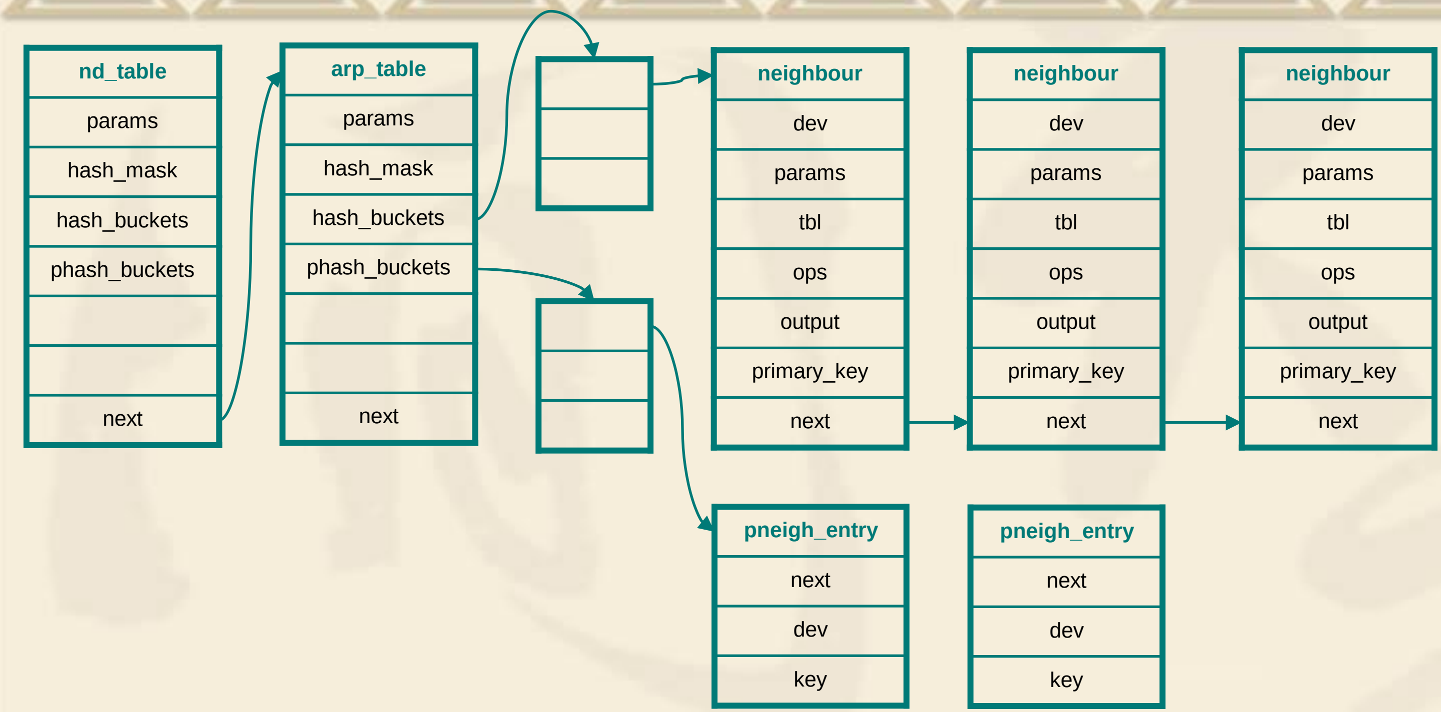


RTT: round trip time
srtt: average RTT
sdrtt: average deviation RTT

慢启动和拥塞避免算法
指数退避 exponential backoff

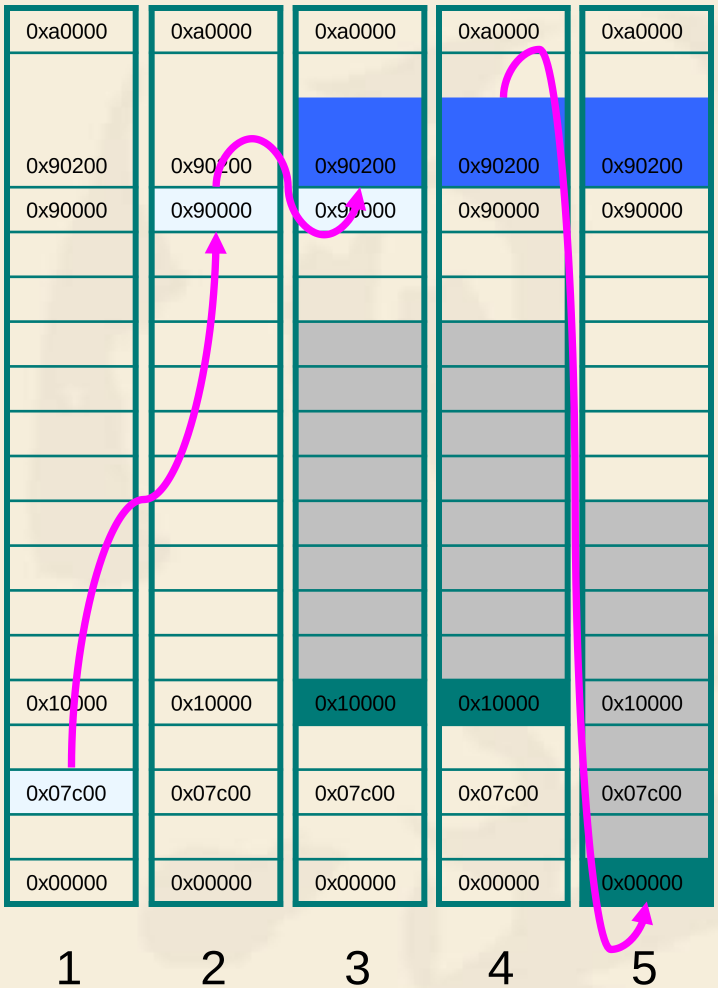
neigh_table

Neighbor system(Linux)



struct neigh_table
struct neigh_statistics
struct neighbour
struct hhcache
struct pneigh_entry
struct neigh_parms

Boot



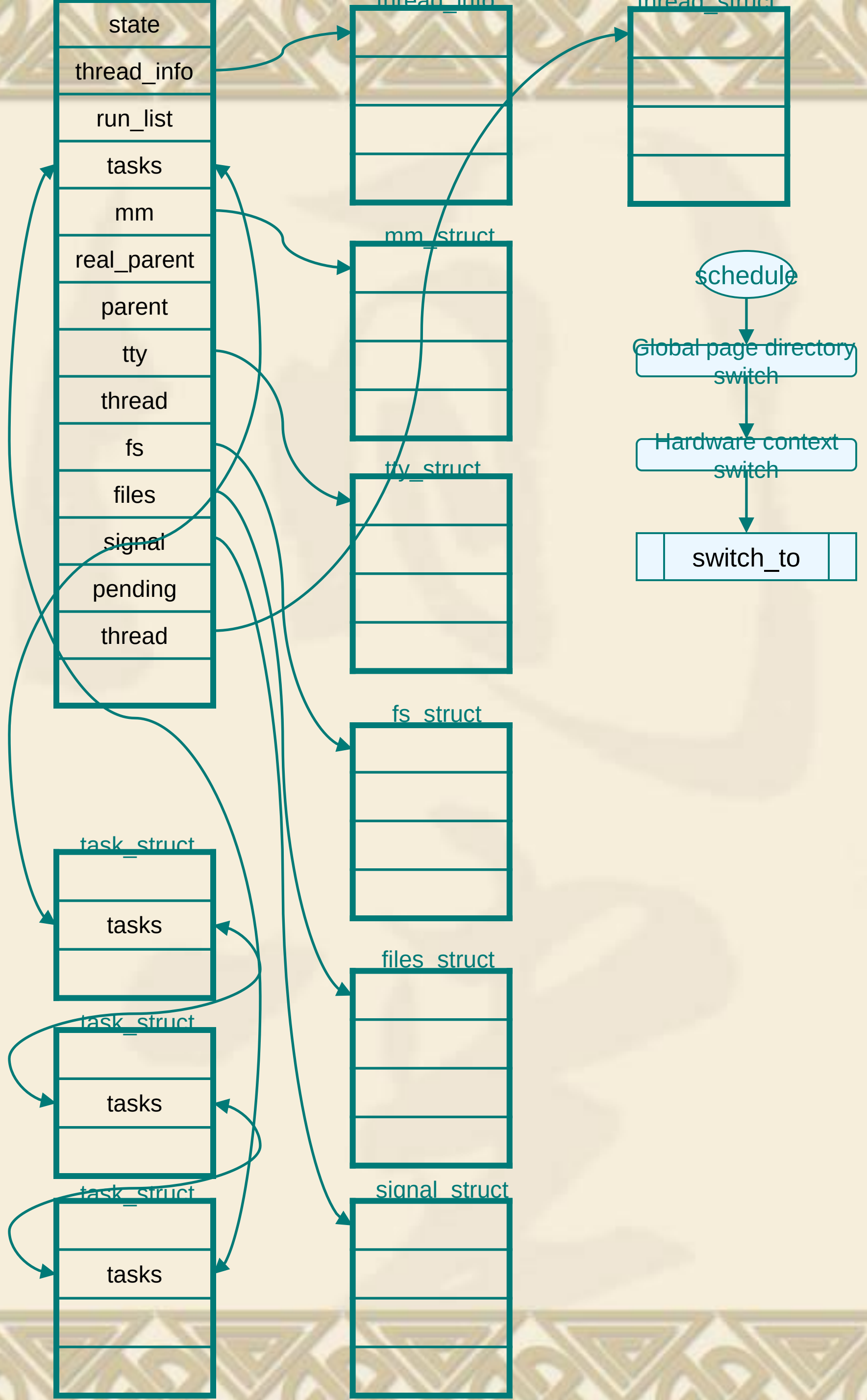

```
SETUPLEN = 4
BOOTSEG  = 0x07c0
INITSEG  = 0x9000
SETUPSEG = 0x9020
SYSSEG   = 0x1000
ENDSEG   = SYSSEG + SYSSIZE

! ROOT_DEV:      0x000 - same type of
!               0x301 - first partit
ROOT_DEV = 0x306

entry start
start:
        mov     ax,#BOOTSEG
        mov     ds,ax
        mov     ax,#INITSEG
        mov     es,ax
        mov     cx,#256
        sub     si,si
        sub     di,di
        rep
        movw
        jmpi    go,INITSEG
go:     mov     ax,cs
        mov     ds,ax
        mov     es,ax
! put stack at 0x9ff00.
        mov     ss,ax
        mov     sp,#0xFF00

! load the setup-sectors directly af
! Note that 'es' is already set up.

load_setup:
        mov     dx,#0x0000
        mov     cx,#0x0002
        mov     bx,#0x0200
        mov     ax,#0x0200+SETUPLEN
        int     0x13
        jnc     ok_load_setup
        mov     dx,#0x0000
        mov     ax,#0x0000
        int     0x13
        j       load_setup
```



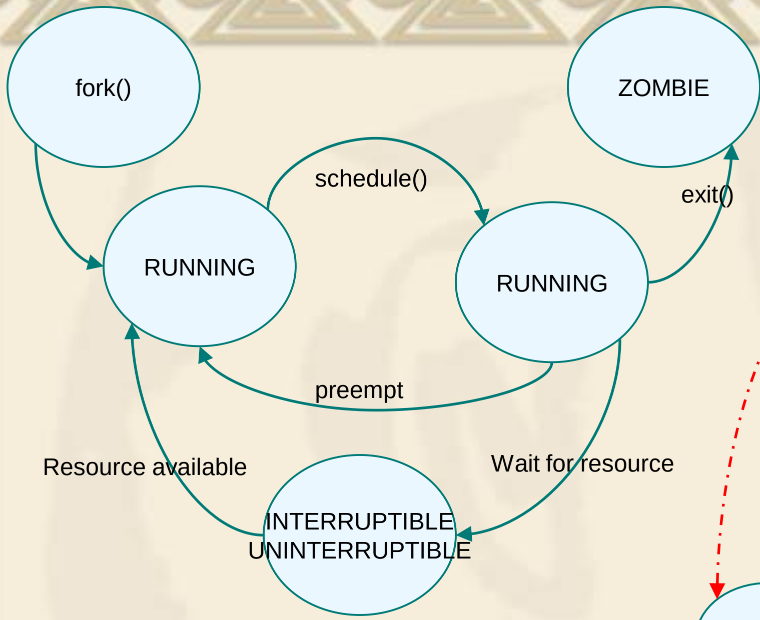
进程描述符task_struct
状态、基本信息、内存描述符指针、tty、目录、文件、信号 等
pids[PIDTYPE_MAX]: **四个散列表**
run_list
状态：TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, TASK_TRACED, EXIT_ZOMBIE, EXIT_DEAD
设置指定进程状态 set_task_state
设置当前进程状态 set_current_state
进程标志符 最大为32767 设置这个最大值：/proc/sys/kernel/pid_max
Getpid & getppid
Alloc_thread_info free_thread_info;
current_thread_info: 由esp获取当前thread_info结构
四个散列表 PIDTYPE_PID, PIDTYPE_TGID, PIDTYPE_PGID, PIDTYPE_SID
冲突双向链表

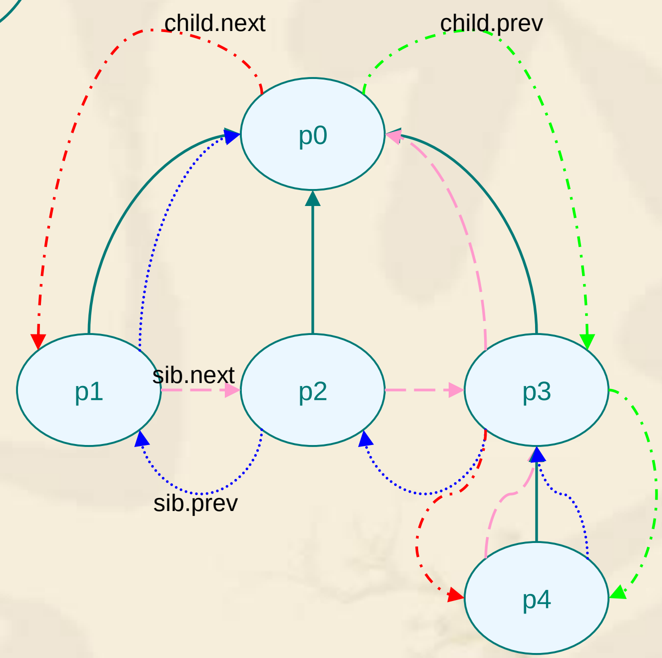
EFLAGS(program status and control)程序的状态及控制

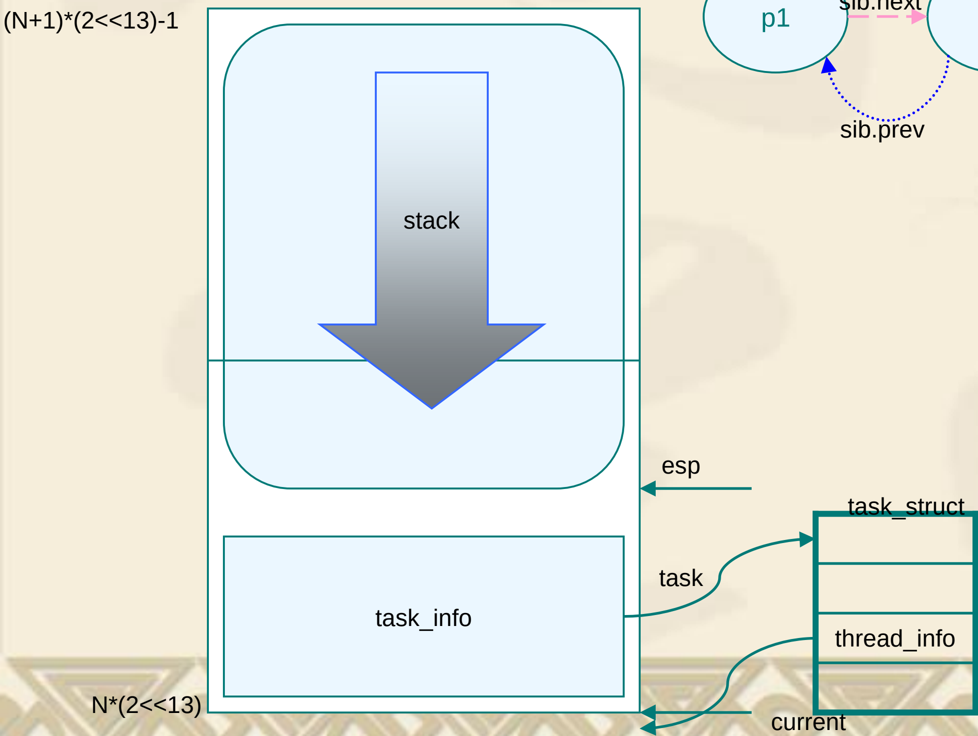
等待队列头的创建宏DECLARE_WAIT_QUEUE_HEAD(name)
初始化动态分配的等待队列的头变量init_waitqueue_head()
初始化wait_queue_t结构：init_waitqueue_entry(q,p)
非互斥进程唤醒：default_wake_function
DEFINE_WAIT声明一个wait_queue_t；autoremove_wake_function初始化这个项；init_waitqueue_func_entry定义唤醒函数
add_wait_queue把一个非互斥进程插入到队列头；
add_wait_queue_exclusive把一个互斥进程插入到队列尾
waitqueue_active判断等待队列是否空；remove_wait_queue从等待队列删除一个进程
重要函数 sleep_on(), interruptible_sleep_on(), sleep_on_timeout(), interruptible_sleep_on_timeout()
schedule, schedule_timeout, wait_event, wait_event_interruptible, prepare_to_wait, prepare_to_wait_interruptible

Kernel mode stack: is combined with current thread info, so the change of the stack, ie, the change of esp, means the change of process.

Process & Thread







Process
do_fork():
create a process by coping current process, share the same process context with its parent.
Return twice with different value.

do_exec(): read and load bin, alloc new context and excute it.

Thread
In linux, it is the same as process;
In Windows, it is called 'lightweight process' and supported differently with process.

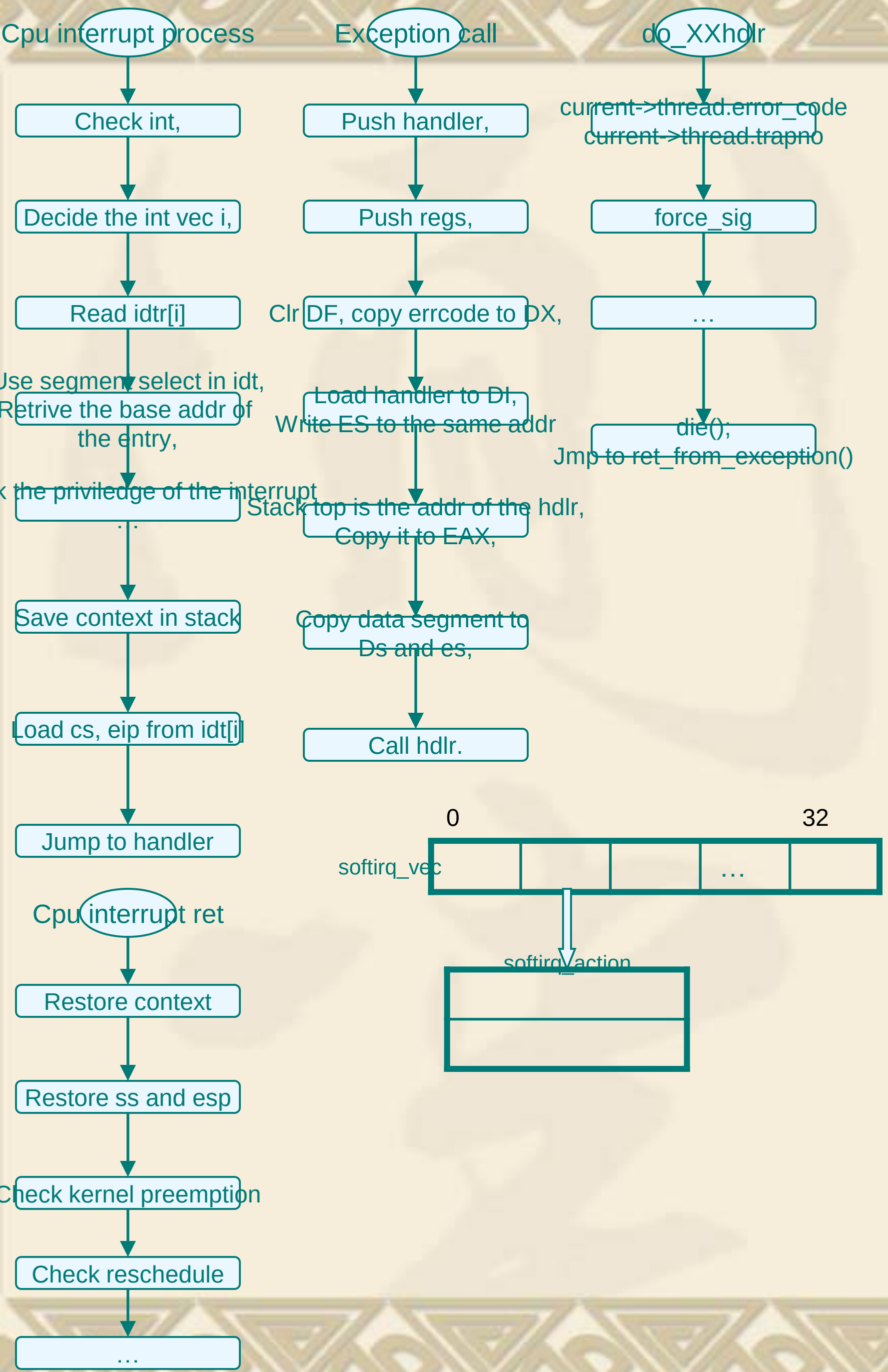
Kernel threads: background thread, only run in kernel mode, mm is NULL.; kernel_thread().
0, 1, keventd; kapmd; kswapd; pdflush; kblockd; ksoftirqd;

exit_group() & **do_exit**(): exit() & pthread_exit().

Schedule
Cooperative multitasking & Preemptive multitasking
Runable task queue
Priority Array: 140

priority inversion

Exception & Interrupt



Exception:

Generated by CPU itself and only occurs after an instruction finished.

1. Processor detecetd – fault, trap, abort;
2. Programmed – software interrupt, int.

Interrupt:

Externally generated by other devices.

1. Maskable interrupt: a masked interrupt will be ignored by CPU.
2. Nonmaskable interrupt: cannot be masked.
3. 3 types of interrupt: I/O, clock, inter-CPU,

Shared IRQ: how to differentiate different IRQs coming from one IRQ line. All ISRs will be excuted, but only one will finish its function.

Dynamic IRQ: the same IRQ line visited by different devices in different times.

Steps of I/O:

1, save IRQ and regs in stack;
2, answer to PIC;
3, ISR;
4, return from intr.

Inter-Processor Interrupt: (call function, reschedule, invalidata TLB)

1, save IRQ and regs in stack;
2, answer to PIC;
3, ISR;
4, return from intr.

SoftIRQ/tasklet

SoftIRQ: 可重入**ksoftirqd**
Tasklet: 不可重入,特殊类型的软中断.

工作者线程 **worker thread**

可延迟中断

Programmable Interrupt Controller

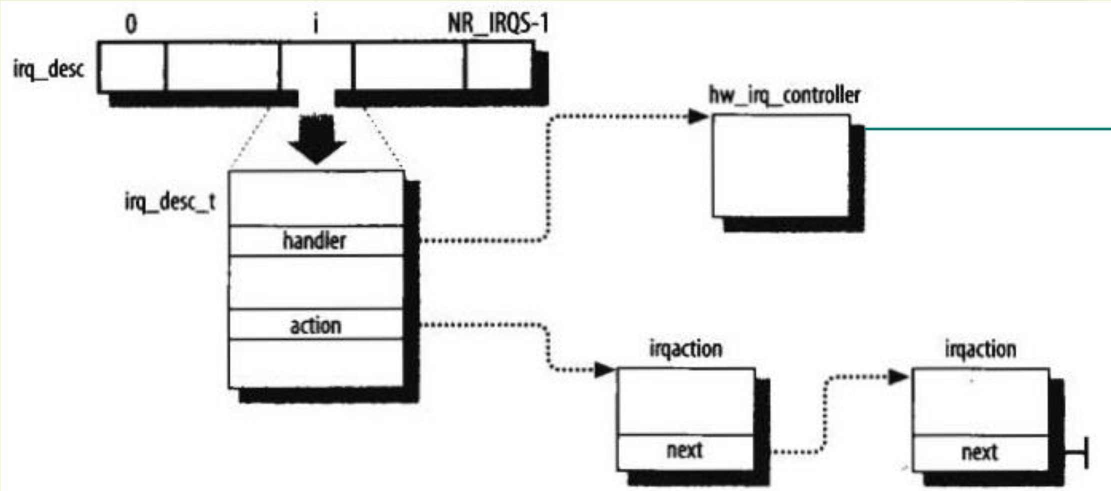
1,Convert IRQ to interrupt vector, store in PIC IO port;2,Send interrupt to CPU INTR;3,Wait and confirm, clear INTR.

IDT&idtr

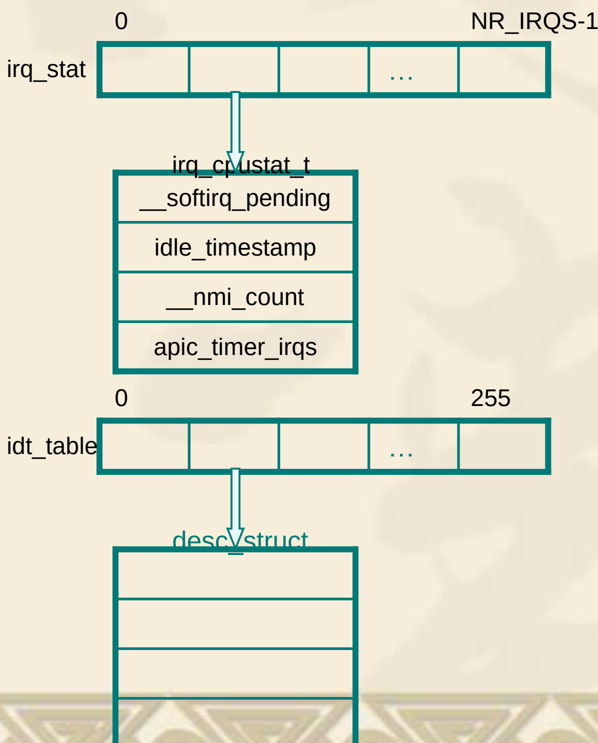
lidtr, 3 types of IDT defined by Intel:(task gate, interrupt gate, trap gate)
Kernel woops?

Page fault

The only exception that can happen in kernel mode.




```
struct hw_interrupt_type i8259A_irq_type = {
    .typename       = "XT-PIC",
    .startup        = startup_8259A_irq,
    .shutdown       = shutdown_8259A_irq,
    .enable         = enable_8259A_irq,
    .disable        = disable_8259A_irq,
    .ack            = mask_and_ack_8259A,
    .end            = end_8259A_irq,
    .set_affinity = NULL
};
```



Kernel stack:

1. Current process kernel stack used for all;
2. Exception stack, per process;
3. Hard interrupt stack. hardirq_stack[], per CPU;
4. Soft interrupt stack. softirq_stack[],per CPU;

1,所有hdlr响应,不允许产生相同的中断事件
2,中断处理,软中断和tasklet既不能被抢占也不能被堵塞
3,hdlr不能执行可延迟函数或系统调用服务的内核控制路径中断??
4,软中断和tasklet不能在一个CPU上交错执行
5,同一个tasklet不能同时在几个CPU上执行

kirqd:adjuct the CPU allocation of IRQs

Regenerate a lost IRQ caused by IRQ_DISABLED

进程同步/通信

Kernel Sync

A:SYSTEM V IPC:

1. Semaphore 信号量

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);

2.Share Memory 共享内存

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);

3.Message Queues 消息队列

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

B:PIPE
C:FIFO
D:POSIX MSGQ
E:SIGNAL

线程同步

1. semophore 信号量

sem_wait(&bin_sem);
sem_post(&bin_sem);
sem_destroy(&bin_sem);

2. Mutex 互斥

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

3. 读写锁

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

4. 条件变量

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t   *cond,   pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t   *cond,   pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);

Event
Critical section

自旋锁: 保护不同的CPU并发访问

1, ISR; 2, forbid deferable function, 3, set positive preempt counter.

?只有当内核在执行异常处理程序而且抢占没有被禁止，才可能抢占内核?

中断返回的时候会判断是返回到用户模式还是内核模式.在非抢占系统中只有返回用户模式才会判断TIF_NEED_RESCHED以及重新调度,返回内核态的时候不会(用户态抢占).而抢占式内核则不论中断返回用户态还是内核态都会判断并调度(内核态抢占)

临界区 关中断,禁止抢占,

进入内核态的途径:

竞争条件的产生: 交叉内核控制路径; 多CPU

1,中断hdlr和tasklet不必可重入;2,仅被软中断和tasklet访问的每CPU变量不需要同步;3,仅被一种tasklet访问的数据结构不需要同步.

内核同步：

主要是防止多核处理器同时访问修改某段代码，或者在对设备驱动程序进行临界区保护。主要有一下几种方式：

1,每cpu变量,禁止抢占的情况下访问
2,原子操作,atomic_xxx(); xxx_bit()
3,内存屏障, 流水处理技术导致的指令交错.(优化屏障:避免编译器优化导致的指令重排序.(volatile))
4,自旋锁, (读-写spinlock,对读,写操作分别加锁)(顺序锁,写具有较高悠闲级)
内核路径自旋&进程挂起
5,信号量, (mutex是一种特殊的sem)(读写信号量,分别对读写设置)
6,顺序锁
7,本地中断禁止
8,本地软中断禁止
9,读-拷贝-更新, 读和拷贝同时进行,当读锁全部释放才进行更新.

Memory

Logical Address:
Linear Address: virtual address, 4G
Physical Address:
程序员可见Logical Address ->（**分段单元**）Linear Address ->（**分页单元**）Physical Address
Memory arbiter: DMA和CPU，CPU和CPU之间并发访问；由硬件访问，对程序员不可见
Real mode: 操作系统自举
Protected mode:
段选择符: 16 bit。Segment Selector(16 bit) + Offset(32 bit)。组成：
index, 指定放在GDT或者LDT中的相应**段描述符**入口。段描述符地址 = gdtr或ldtr + index * 8
TI, GDT还是LDT
RPL 特权级

段寄存器: 存放**段选择符**
CS, 当前特权级(Current Priviledge Level)使用2 bit，表明CPU当前特权。在Linux中，0为内核态，3为用户态。
SS, DS;
ES, FS, GS //**都是附加段寄存器**
SI, DI //源/目变址寄存器

段描述符Segment Descriptor, 8 byte (64 bit),
组成：描述段的特征（段线性地址，粒度，最后一个内存单元偏移，段类型(系统段或者普通代码/数据段)，类型和存取权限，
段描述符特权DPL，规定访问该段的特权级别
CPL, 当前进程的特权级别
Segment-Present标志，D/B标志(段偏移量32或者16位？)，AVL(Linux未用)）
GDT (Global Descriptor Table) 全局描述符表 gdtr控制寄存器, 最大8191 个(13 bit)
LDT (Local Descriptor Table) 局部描述符表 在不同的进程中创建, ldtr控制寄存器,
描述符种类：**代码段描述符，数据段描述符，任务状态段描述符TSSD**(该段用于保存处理器寄存器的内容，GDT中)，局部描述符表描述符LDTD(GDT 中)。
快速访问段描述符 缓存**段描述符**

分段单元：逻辑地址转换到线性地址 1: gdtr或ldtr + index * 8 寻址段描述符； 2: 段描述符base字段+逻辑地址偏移
分段：不同进程分配不同的线性地址
分页：同一线性地址映射到不同的物理地址
用户代码段，用户数据段，内核代码段，内核数据段
宏__USER_CS, __USER_DS, __KERNEL_CS, __KERNEL_DS

Memory

Linux GDT
数组cpu_gdt_table 存放GDT
数组cpu_gdt_descr 存放GDT地址极其大小

任务状态段TSS: init_tss. TSS为246字节长
切换CPU时恢复现场；执行I/O时检查许可位图

缺省局部描述符表段 LDT
局部线程存储段 TLS 允许多线程应用使用最多3个局部于线程的数据段
set_thread_area(), get_thread_area()

高级电源管理段3个
PnP功能段5个
双重错误段 处理异常时引发另外一个异常，产生双重错误

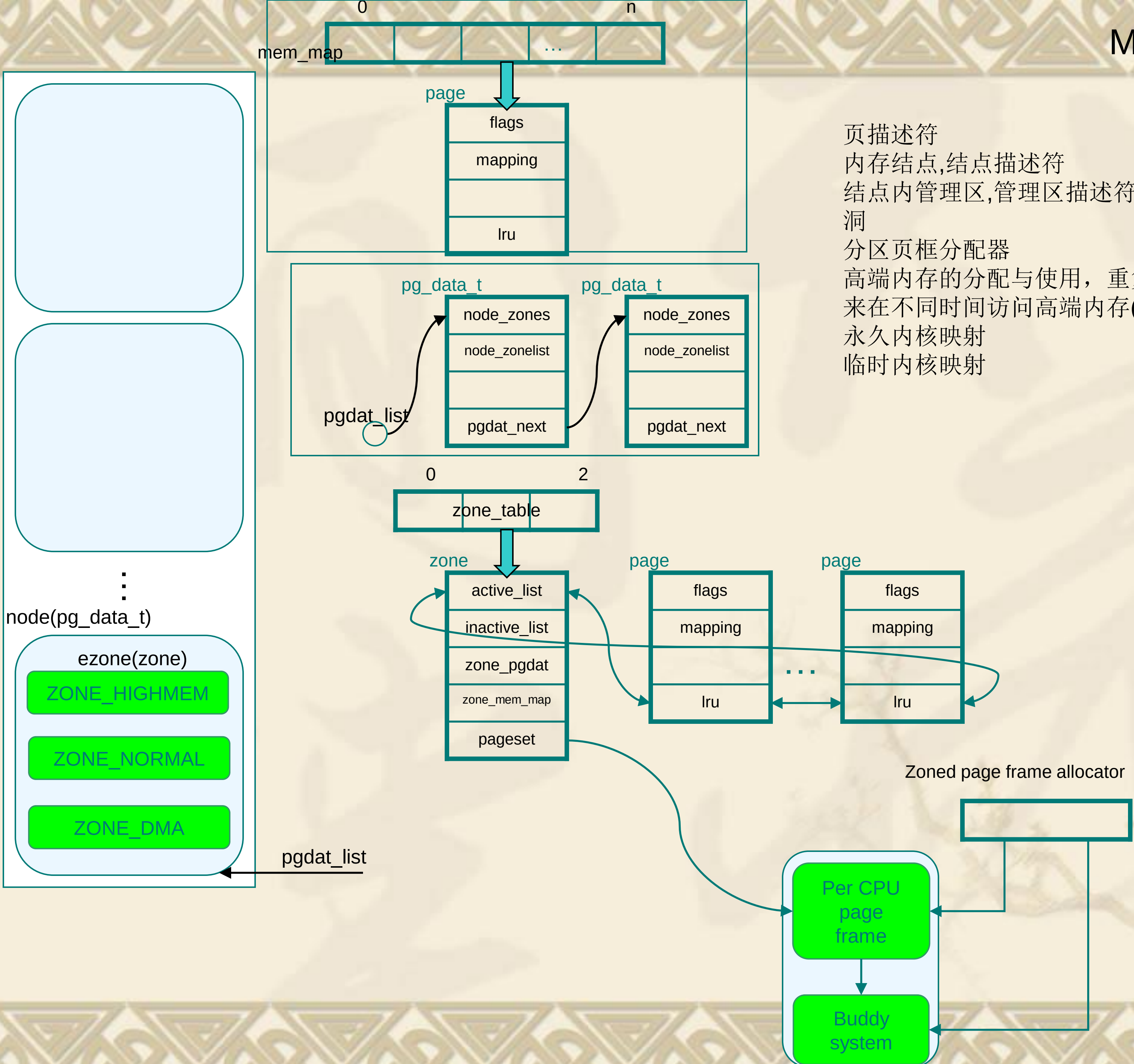
Linux LDT
缺省LDT: default_ldt. 5个表项; modify_ldt()创建自己的局部描述符表

调用门 在调用预定义函数的时候改变CPU特权
分页单元: page frame, page table
缺页异常 访问类型与线性地址的访问权限不一致
控制寄存器cr0: PG标志为0时，线性地址被解释成物理地址
32位线性地址：Directory(10 bit) + Table(10 bit) + Offset(12 bit)

页目录 page directory
组成 Present, Accessed, Dirty, R/W, User/Supervisor, PCD/PWT, Page size, Global
TLB: Translation Lookaside Buffer, Associative Memory, 俗称 快表

页表 page table
控制寄存器cr3: 正在使用页目录的物理地址
扩展分页
线性地址没有Table部分，页框大小为4M, page size置位，20位物理地址只有高10位有意义。Cr4的PSE标志使得两种分页共存。
User/Supervisor=0时，只有内核态才能寻址。为1时，总能寻址
物理地址扩展分页(PAE和PSE(Linux未使用))
启用: cr4的PAE标志置位; page directory中的page size使用大尺寸页面.此时页面大小为2M.

Memory



页描述符
内存结点,结点描述符
结点内管理区,管理区描述符
洞
分区页框分配器
高端内存的分配与使用，重复使用最后128M线性地址来在不同时间访问高端内存(与高端内存映射)
永久内核映射
临时内核映射