The Design and Implementation of Microkernel

Team members: 刘源，章鑫磊，李庭弘

Tutor: 华保健

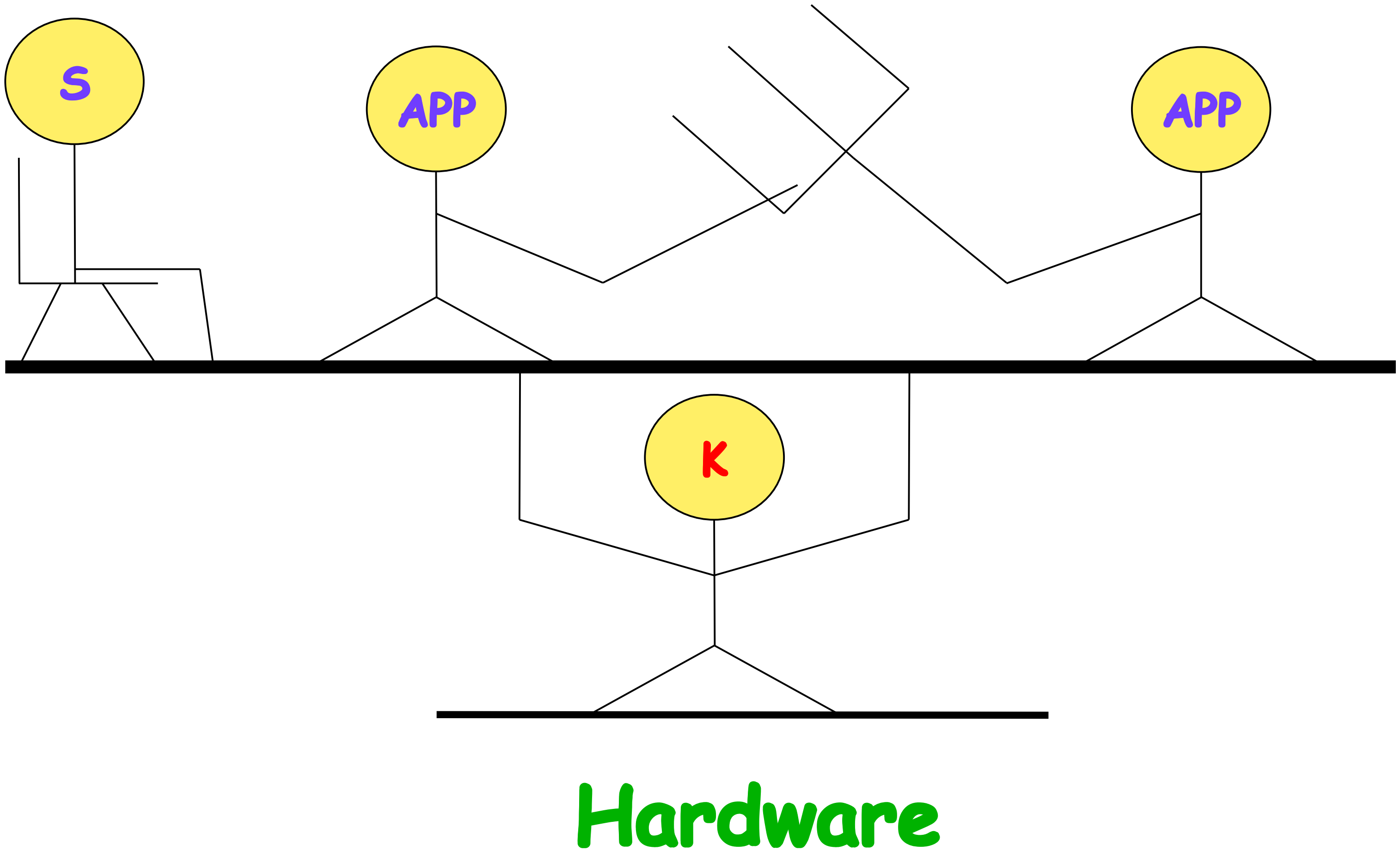
Speaker: 刘源

Mar 26 2009

Prepared by 刘源

中国科学技术大学

- APP, Server, Kernel, HW

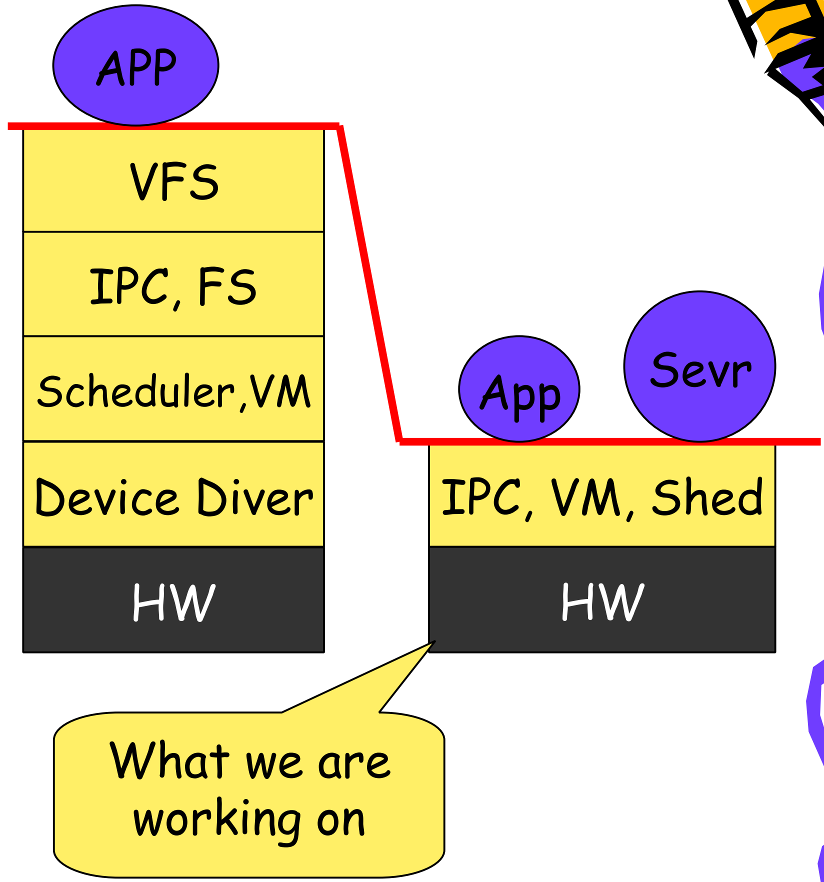


OS organization

- Monolithic Kernel
  - Kernel is a big program
    - Good: easy for sub-systems to cooperate
    - Bad: complex, no isolation within kernel sub-systems

  Very successful and efficient as a whole
- Microkernel
  - Kernel provides only a small set core functionality
    - Good: simple/efficient kernel, sub-system isolated, better modularity
    - Bad: lots of IPC may be slow

  Lots of good individual ideas, but overall plan fails expectations because of not-so-good performance
- Two approach are just different reactions to increased complexity of Unix



What is an OS

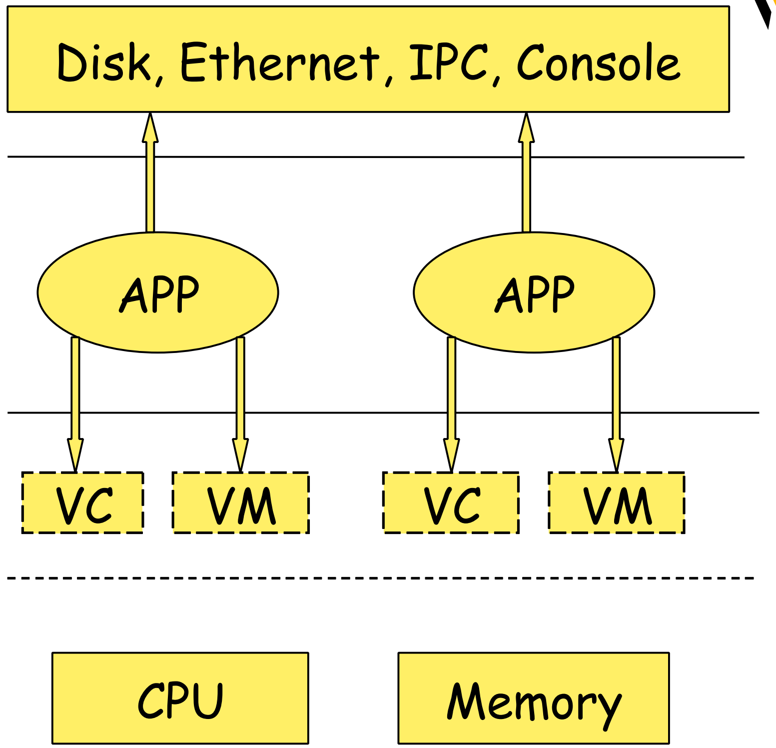
- Provides **abstractions** for APP
  - Manages and hides details of hardware
  - Access hardware through low level interfaces unavailable to APP
- Provides **isolation/protection** for APP
- Make hardware useful to APP

Ways to structure an OS

- The traditional Unix approach
  - **Virtualize** some resources: CPU, MEM.
    - each APP has a CPU and MEM.

    Forge the abstraction for APP: **Process**
  - **Abstract** others: disk, network, IPC
    - Sharable abstraction over HW. E.g. file system.

    Provide a clean and consistent interface / abstraction for APP: **File**
    - Socket: not very well integrated into File abstraction



Virtualization of CPU

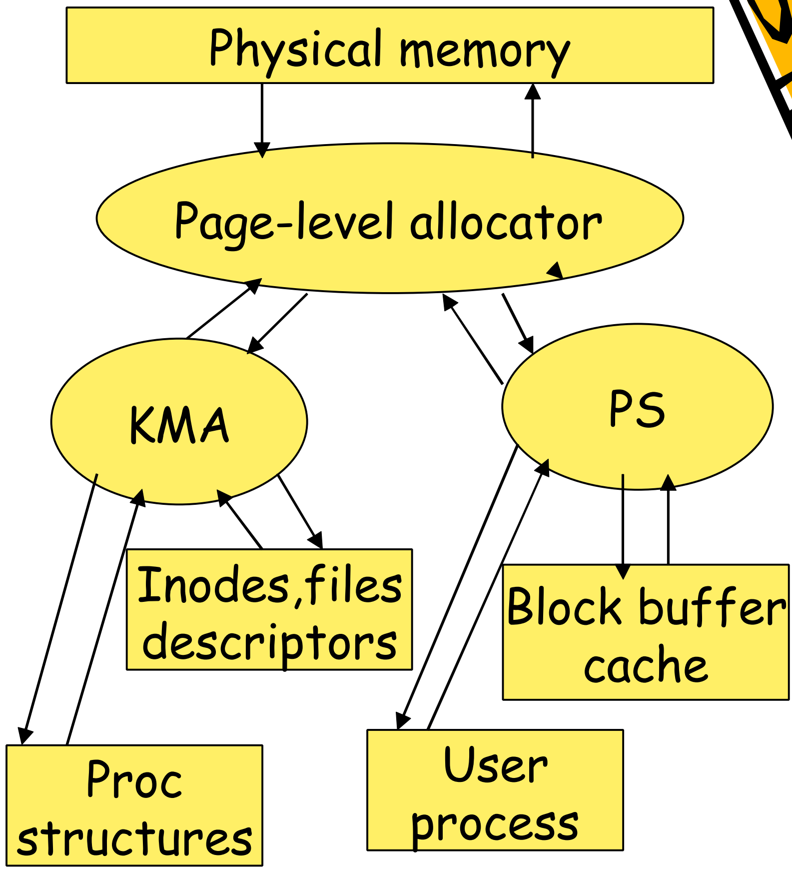
- Goal
  - A dedicated CPU for each process
    - Via clock interrupt: Run processes in turn and give each process 'time slice'.
  - **Transparent** CPU multiplexing
    - Via hardware registers or per process structure: OS saves state, then restores
- Point:
  - APP need not worry about CPU virtualization (but may notice CPU is actually multiplexed in some system calls. E.g. read())

Virtualization of MEM

- Traditional Unix's goal:
  - A complete memory system for each APP (Manage both physical and virtual memory)
    - Via paging or segmentation hardware: give a fundamental abstraction of process: **Address Space**
    - Via system calls: Traditional Unix just export very limited virtual memory control to APP (APP recognize virtual memory rather than physical memory)
- Point
  - Addresses start at 0
  - Run programs larger than physical memory
  - Relieve APP's burden of managing physical memory

Memory Management

- Traditional Unix approach:
  - Page-level Allocator
    - Usually maintains a linked list of free physical pages
  - Paging System
    - Allocate pages to hold process's address space
  - Kernel Memory Allocator
    - Provide chunks of memory of variable sizes, for kernel sub-system 's dynamic allocation requests.

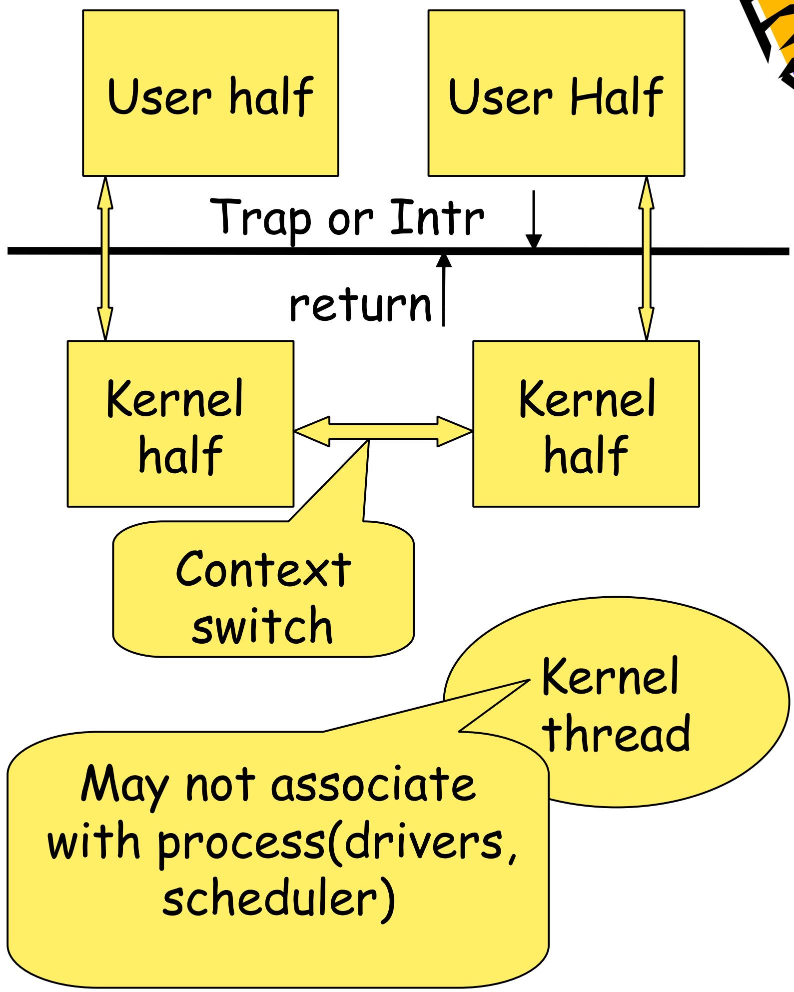


Memory Management

- Our approach
  - Rudimental memory management in kernel
    - Only page-level allocator
    - Kernel objects are allocated statically (For simplicity)
  - Export more virtual memory control to user-level APP
    - Processes can manipulate address space of themselves or children
    - Via IPC, processes even can pass on page mappings
  - User-level page fault handler
    - Make it possible for user-level fork() and Copy-On-Write
    - Allow APP to define their own semantics of fork()
- Point
  - Make it possible for user-level servers/libraries to manipulate address space and get involved in process creation
  - Keep kernel simpler thus more likely to be correct

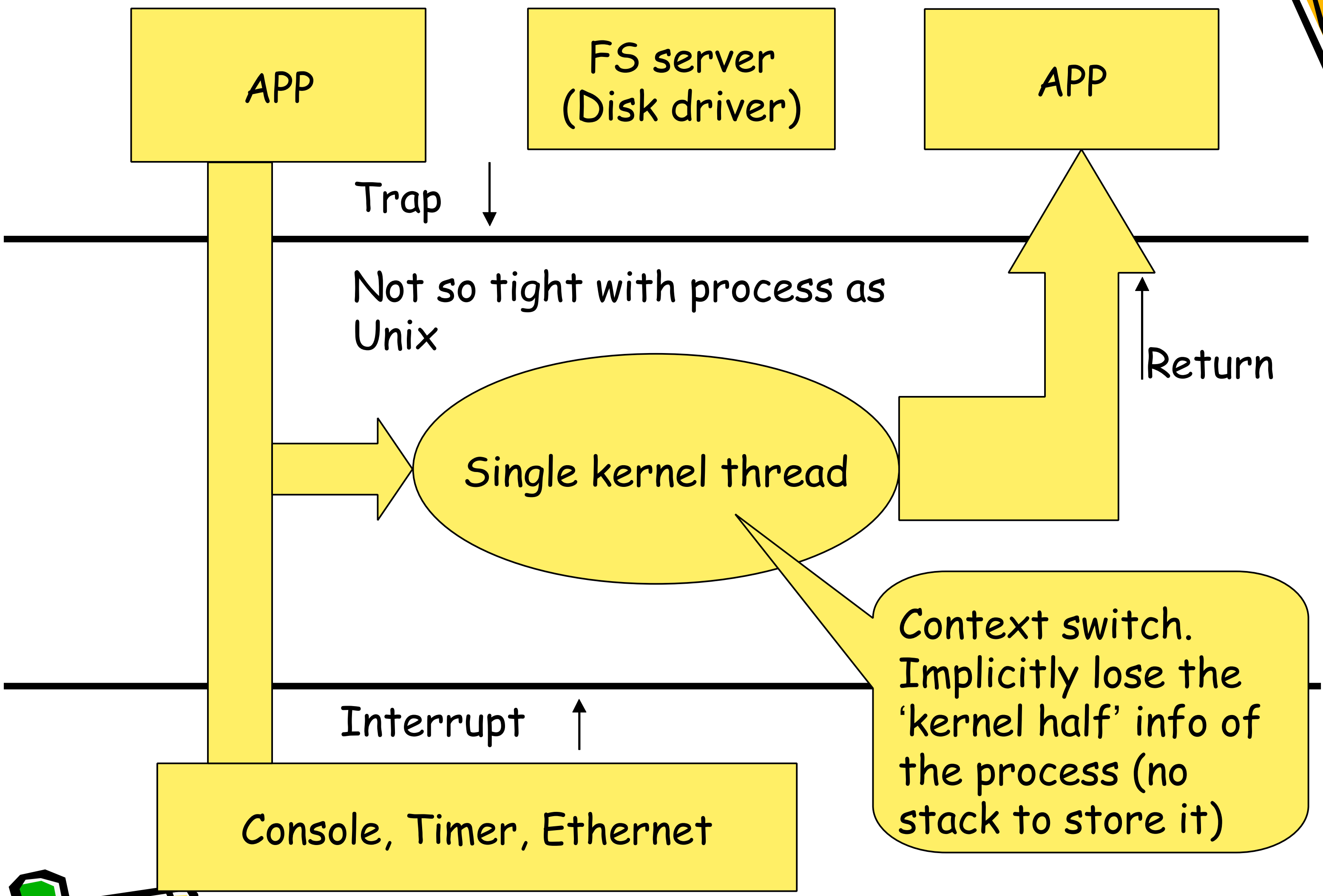
Programming model

- Traditional Unix approach:
  - App's perspective of process:
    - address space + single execution of unit (known as thread)
  - Kernel's perspective of process:
    - user half + kernel half (associate with a kernel thread)
    - Context switch happens at kernel half (no immediate effect)
  - Process model: per-process kernel stack
  - Kernel is multi-threaded
  - User/Kernel transitions can been actually seen as cross-boarder control transfer within process



Programming model

- Our kernel approach:
  - Interrupt model: Treat system calls and exceptions as interrupts, using a single per-processor stack.
    - Memory-efficient, simplicity but complicate the issue in multi-threaded kernel with interrupts enabled
  - To make it simpler (poorer performance) in our kernel
    - Interrupts are *always disabled* in kernel (so wider than unix approach)
    - Single threaded kernel, thus result in:
      - *No* kernel data synchronization (So no 'wakeup/sleep' primitives)
      - Kernel *never* sleeps (If it sleeps, no alternative thread!)
    - Asynchronous interface(so non-blocking) in kernel level but simulate traditional synchronous (blocking) interface in user-level.

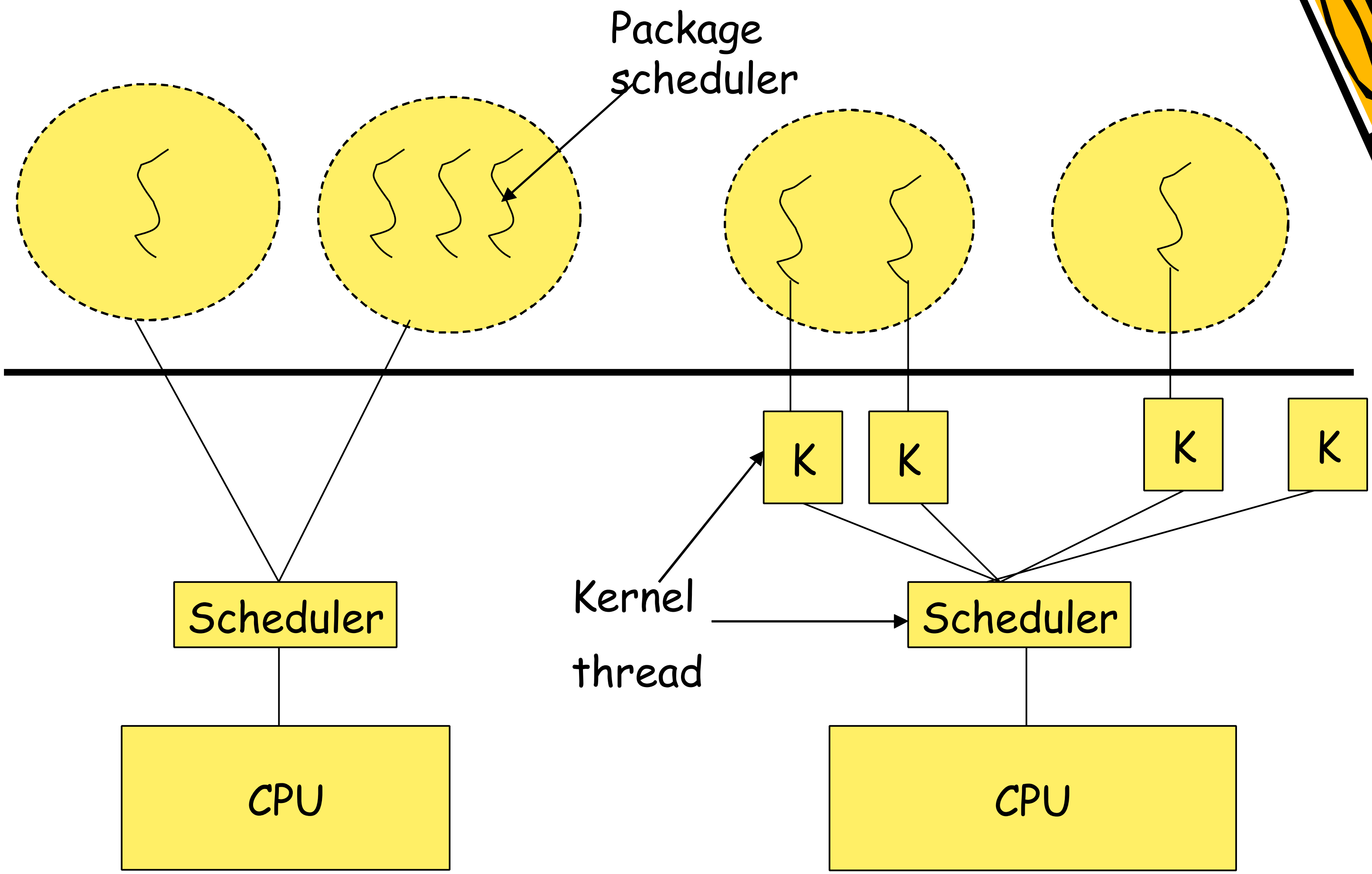

APP
FS server
(Disk driver)
APP
Trap
Not so tight with process as Unix
Single kernel thread
Return
Context switch. Implicitly lose the 'kernel half' info of the process (no stack to store it)
Interrupt
Console, Timer, Ethernet

Scheduler

- Traditional Unix approach
  - Priority scheduling based on a time slice
    - Kernel contains a number of priority-marked run queues
    - Kernel round-robins among processes on the highest-priority run queue
    - Priorities recomputed dynamically: (1) niceness (2) estimated CPU usage
- Our approach
  - Round-robin based on a time slice
    - Need much improvement, now leave it for simplicity

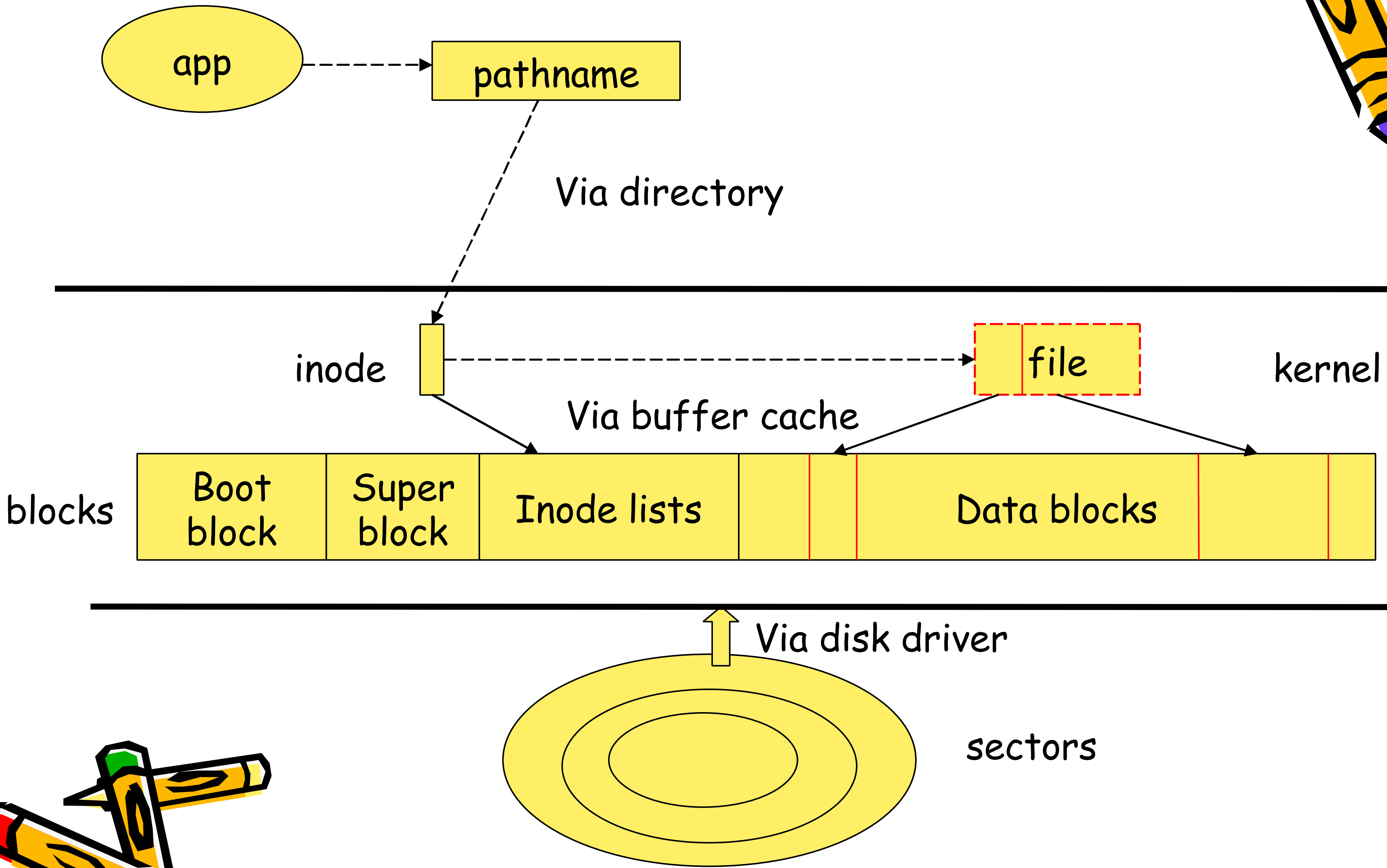
Multi-threads in a process

- Our approach:
  - User level threads package: kernel has *no* knowledge of it (So kernel has no corresponding data structures for it)
    - Good: faster! (No system call overhead)
    - Bad: If one thread blocks, the sibling threads block too!
- Alternative approach:
  - Kernel-supported user threads.
    - Kernel *must* be multi-threaded. (User-threads base on ‘em!)
    - Good: Each schedule-entity in process *can* be scheduled by kernel independently.
    - Bad:
      - More system calls involving creating threads, so more overhead
      - More memory spent to store per-thread information in kernel.


Package scheduler
K
K
K
K
Scheduler
Kernel thread
Scheduler
CPU
CPU

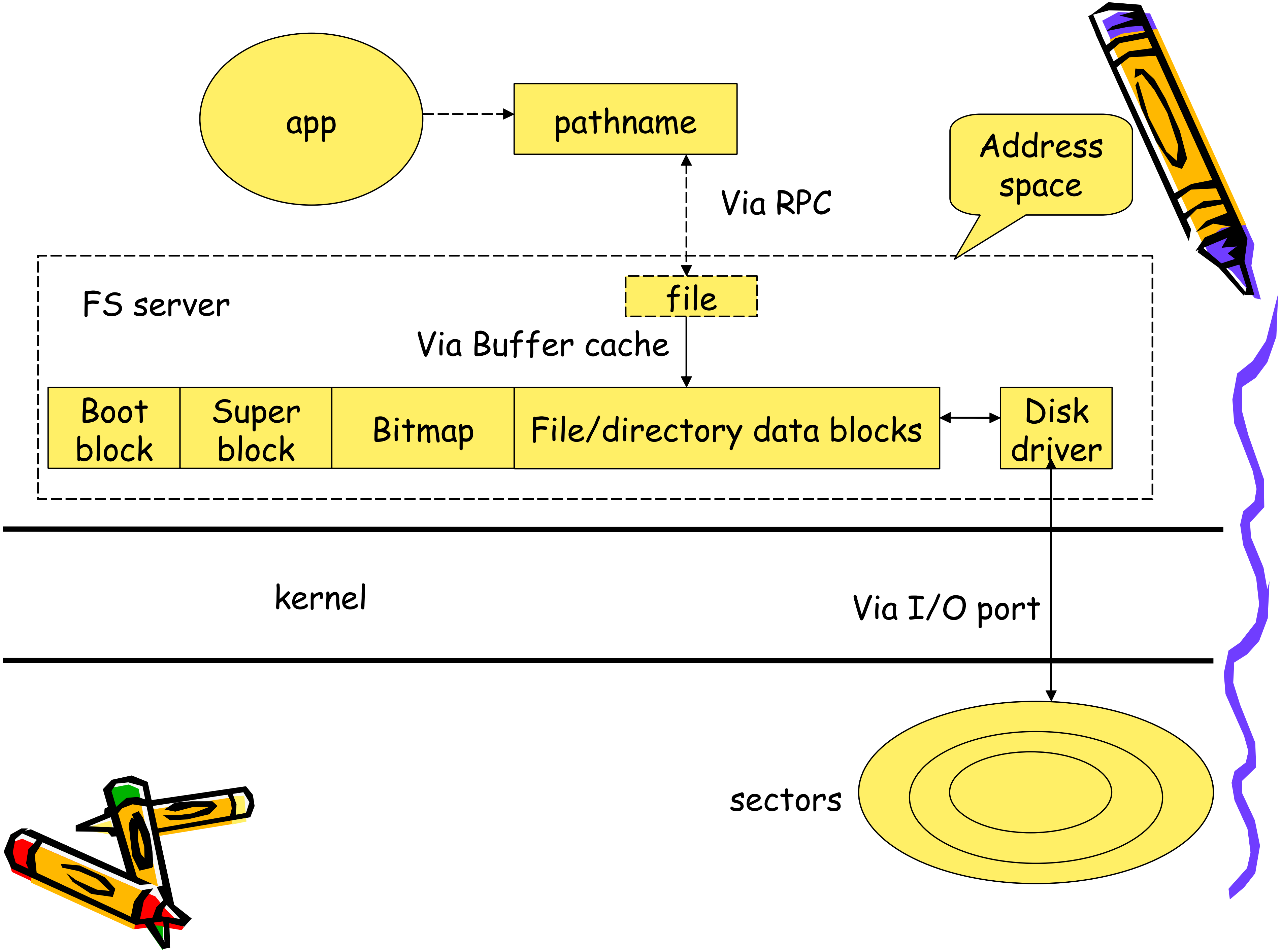
Ways to structure FS

- Traditional Unix approach
  - **Metadata**: Linking multiple disk sectors into files and identifying these files (give structure to raw disk)
    - Inodes: Disk layout of file data, a layer of **indirection** to data (This also makes hard link possible)
    - Directories: Itself a special file. A layer of **indirection** to inode (1)Mapping pathname (textual name of file) to inode. (2)Give tree-hierarchy to file system
      - Hard link: Just another pathname in the directory mapping to the already-mapped inode
      - Soft link: Just a file whose data is the pathname to some inode
    - Lists of free blocks, lists of free inodes, etc. (Super block)
  - **Data**: Stored into Blocks, accessed by Block numbers, indexed by Inode, interfaced by File
    - Block: low-level container of data, May be multiple times of size of sector (physical container of data)
    - File: Logical Container of data, stream of bytes then other formats can be structured on top of it.
      - For APP, data appears at 0 and get logical offset
    - Buffer cache: I/O in terms of memory speed rather than of disk (This also makes delayed write possible)


app
pathname
Via directory
inode
file
kernel
Via buffer cache
blocks
Boot block
Super block
Inode lists
Data blocks
Via disk driver
sectors

File System

- Our approach
  - Single user FS
    - No notion of file ownership and permissions
  - Do NOT support hard link
    - So NO notion of Inode
    - Then store all of the file metadata with in the directory entry
  - Instead of link-list, Use bitmap to track free disk blocks
    - Try to allocate contiguously
  - User-level file system server
    - Client/Server file system access
      - Via Remote Procedure Call: cross-address-space call service as if ordinary C function call
      - Via I/O port, disk driver as part of FS server
  - Implement "buffer cache" in FS server's address space
    - So need IPC to send page mappings that contain the disk block

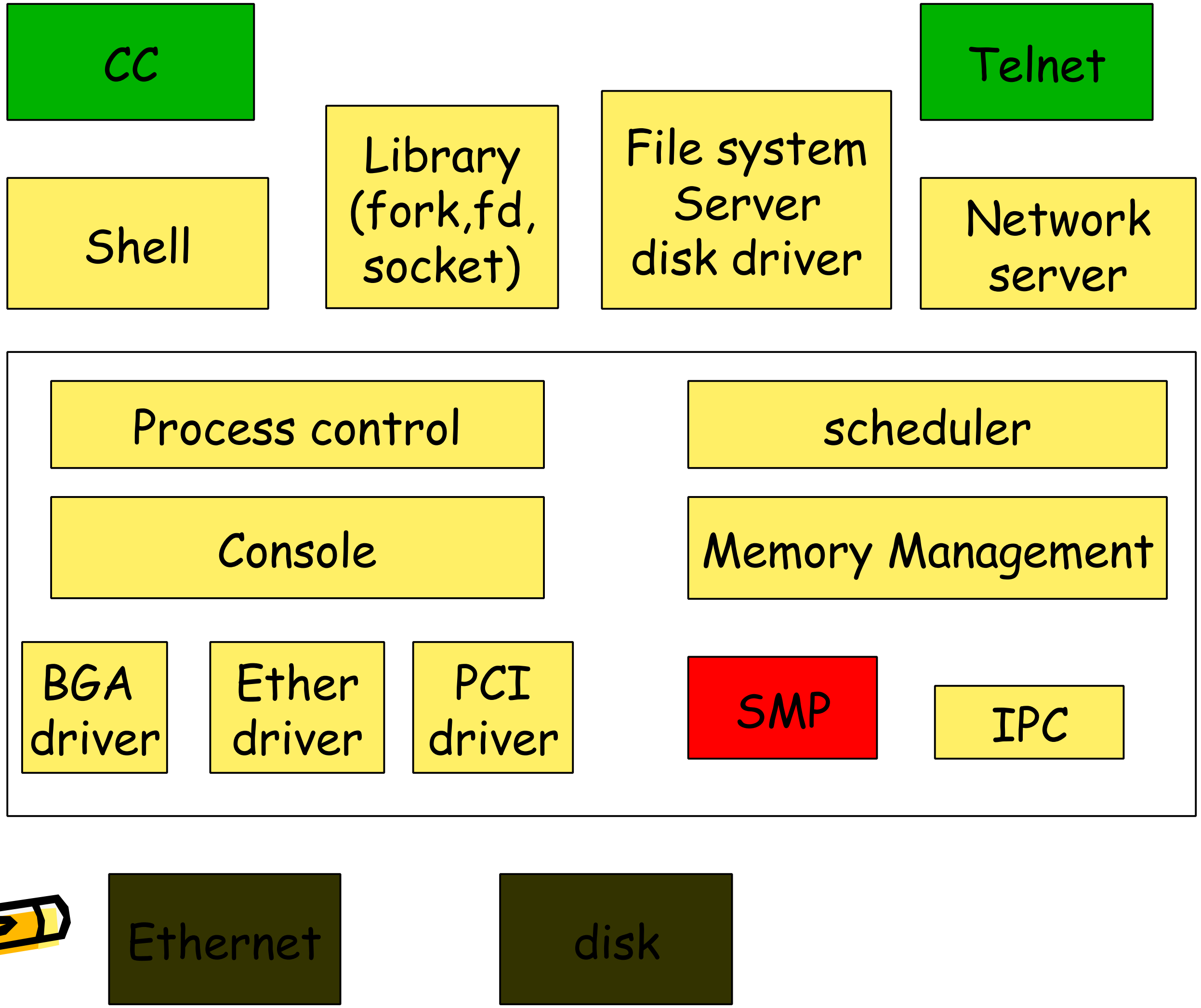

app
pathname
Via RPC
Address space
FS server
file
Via Buffer cache
Boot block
Super block
Bitmap
File/directory data blocks
Disk driver
kernel
Via I/O port
sectors

The beauty of Unix abstractions

- APP get a more powerful/reliable virtual machine from process abstraction
- APP get a bigger memory from address space abstraction
- APP get a clean and consistent interface to communicate with outside world from file abstraction
- APP get a reliable/friendly disk from file system abstraction
- Go give a look at Multics and Plan9, find more ;-)

Now and Future

- What we have done
  - Support GB2312 encoding
  - Full TCP/IP support
  - Shell support redirection, pipe, very few utilities with limited functionality
- We are currently working on SMP support in kernel
- Wish list
  - Add a home-made C compiler to self-support
  - Add some interesting APP like telnet


CC
Telnet
Shell
Library (fork,fd, socket)
File system Server disk driver
Network server
Process control
scheduler
Console
Memory Management
BGA driver
Ether driver
PCI driver
SMP
IPC
Ethernet
disk

Pic: Shell

```
init: bss seems okay
init: args: 'init' 'initarg1' 'initarg2'
init: running sh
init: starting sh
$ cat lorem
                                    少年中国说
                                    梁启超

制出将来之少年中国者，则中国少年之责任也。故今日之责任，不在他人，而全在我少年。


少年智则国智，少年富则国富，少年强则国强，少年独立则国独立，少年自由则国自由，少
年进步则国进步，少年胜于欧洲则国胜于欧洲，少年雄于地球则国雄于地球。

美哉我少年中国，与天不老；壮哉我中国少年，与国无疆！
$ cat out

$ cat lorem >out
$ cat out | num
    1                                     少年中国说
    2                                     梁启超
    3
    4 制出将来之少年中国者，则中国少年之责任也。故今日之责任，不在他人，而全在我
少年。
    5
    6 少年智则国智，少年富则国富，少年强则国强，少年独立则国独立，少年自由则国自
由，少年进步则国进步，少年胜于欧洲则国胜于欧洲，少年雄于地球则国雄于地球。
    7
    8 美哉我少年中国，与天不老；壮哉我中国少年，与国无疆！
$
```

Pic: Echo server via TCP

```
Physical memory: 66556K available, base = 640K, extended = 65532K
enabled interrupts: 1 2 4
     Setup timer interrupts via 8259A
enabled interrupts: 0 1 2 4
     unmasked timer interrupt
  -> reg_base[0] = f0000000
  -> reg_size[0] = 00001000
  -> reg_base[1] = 0000c040
  -> reg_size[1] = 00000040
  -> reg_base[2] = f0020000
  -> reg_size[2] = 00020000
enabled interrupts: 0 1 2 4 11
FS is running
FS can do I/O
Device 1 presence: 1
superblock is good
read_bitmap is good
ns: 52:54:00:12:34:56 bound to static 527(10)
NS: TCP/IP initialized.
opened socket
trying to bind
bound
Client connected: 0.0.0.0
Get:Hiya kid
```

```
File  Edit  View  Terminal  Tabs  Help
yeti@sse:~$ telnet localhost 4242
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hiya kid
Hiya kid
```

Thank you