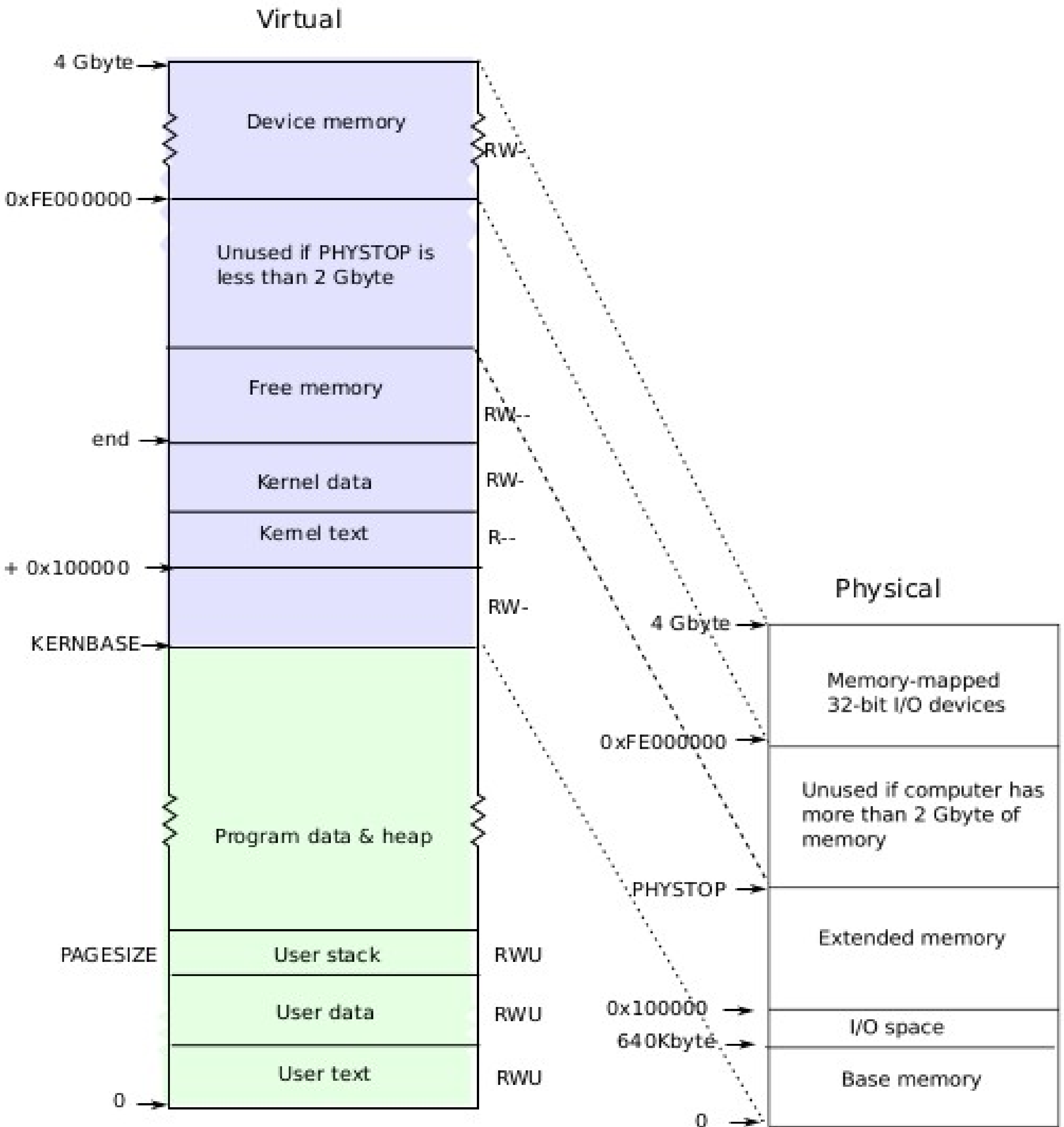
Processes in xv6 code

Process Table

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- **One single global array of processes**
- **Protected by**

  `ptable.lock`

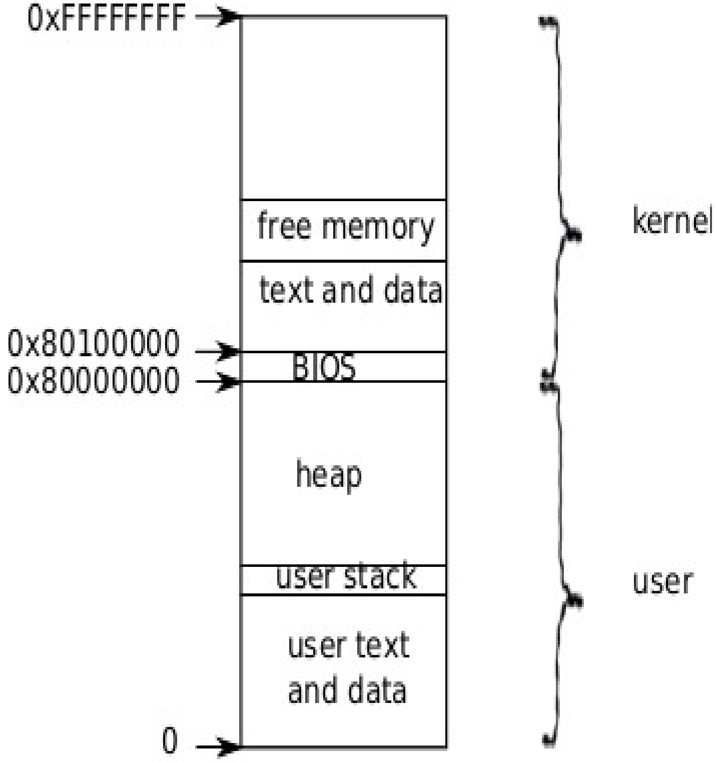


Layout of process's VA space

xv6 schema!

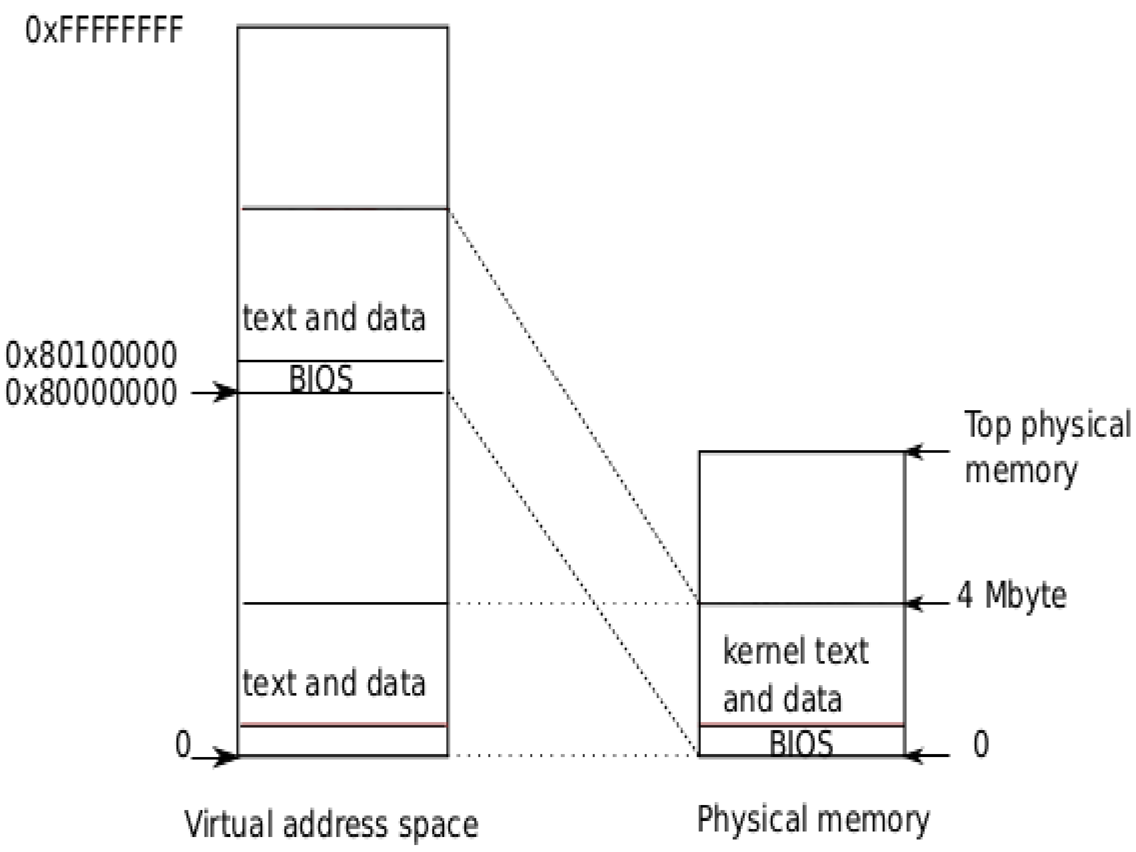
different from Linux

Logical layout of memory for a process



- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**
- **Each processe's address space maps kernel's text, data also --> so that system calls run with these mappings**
- **Kernel code can directly access user memory now**

Kernel mappings in user address space actual location of kernel

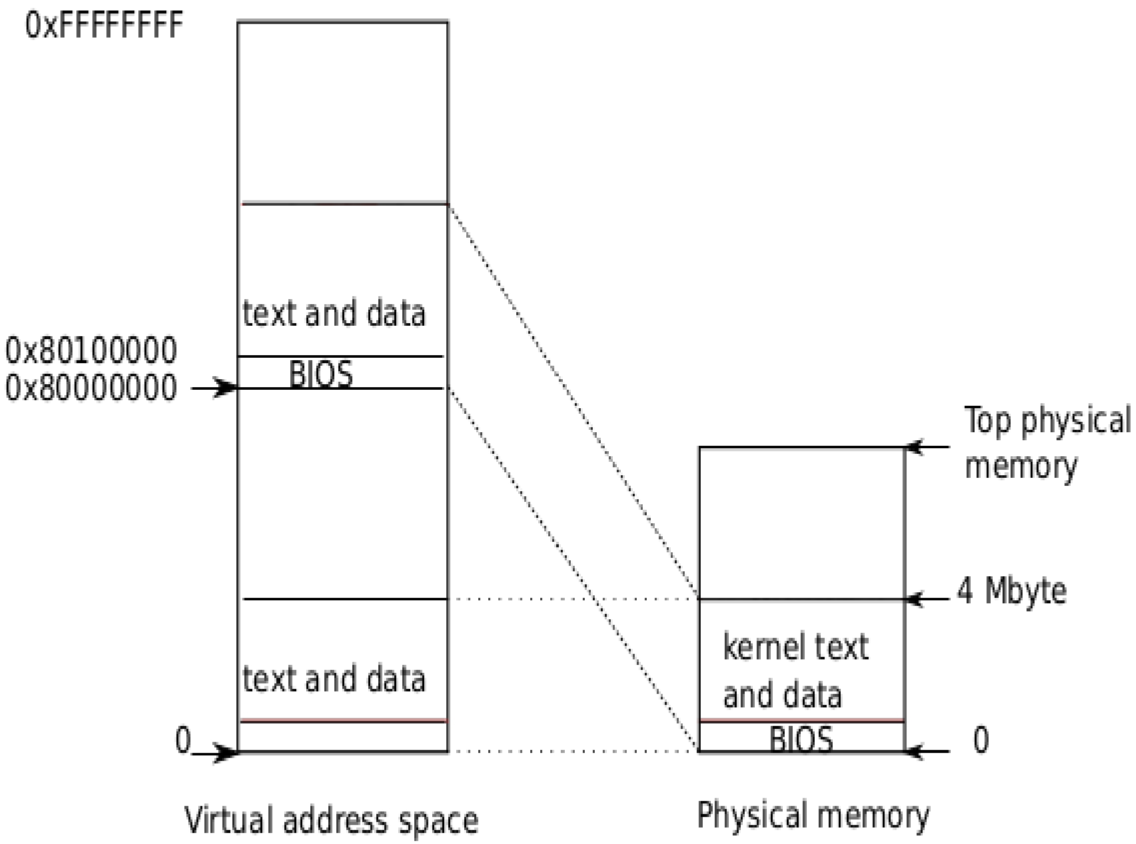


- **Kernel is loaded at 0x100000 physical address**
- **PA 0 to 0x100000 is BIOS and devices**
- **Process's page table will map**

  VA 0x80000000 to PA 0x00000 and

  VA 0x8010000 to 0x100000

Kernel mappings in user address space
actual location of kernel



- **Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory**
- **0x80000000 is called KERNBASE in xv6**

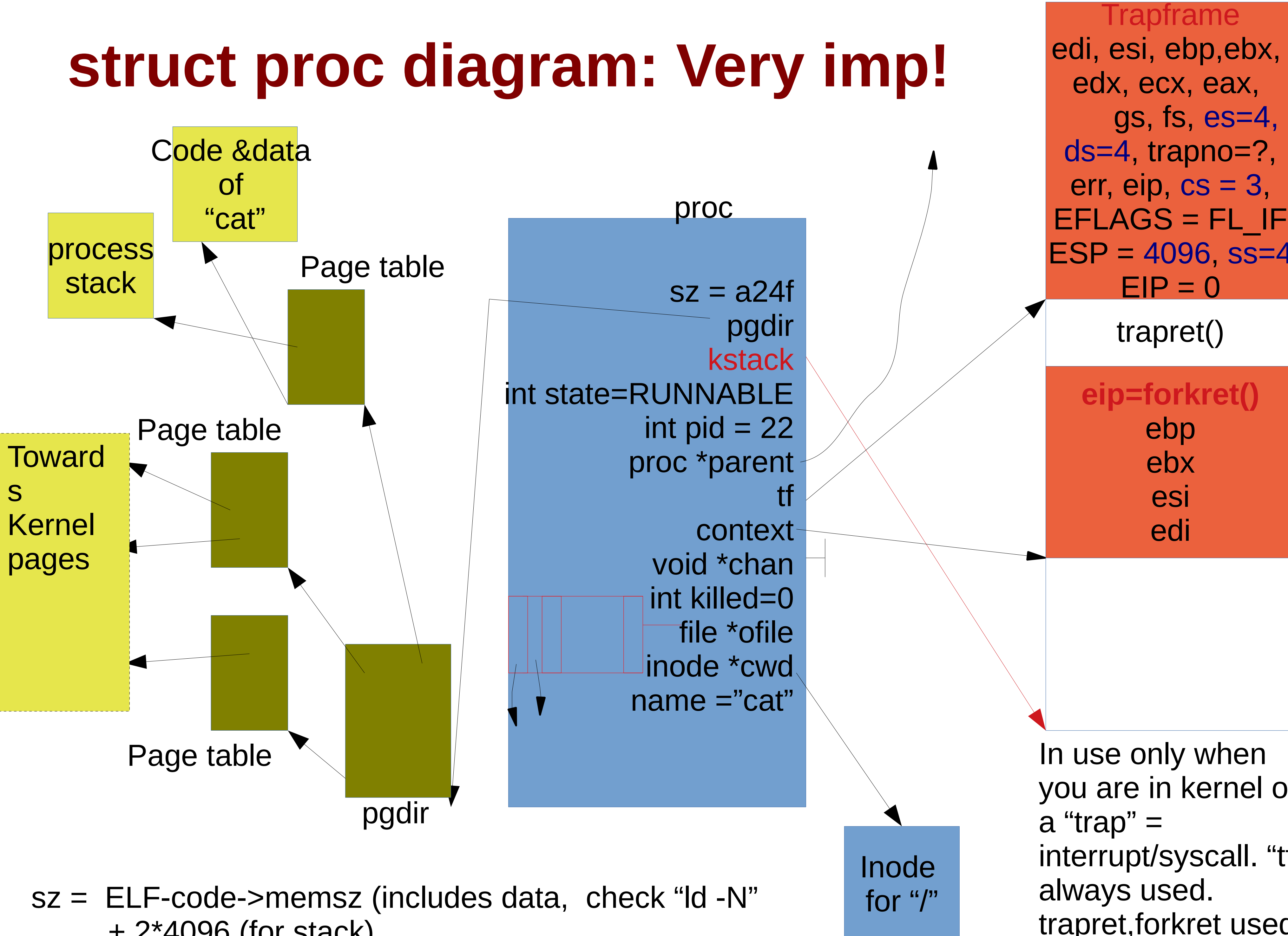
Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Struct proc

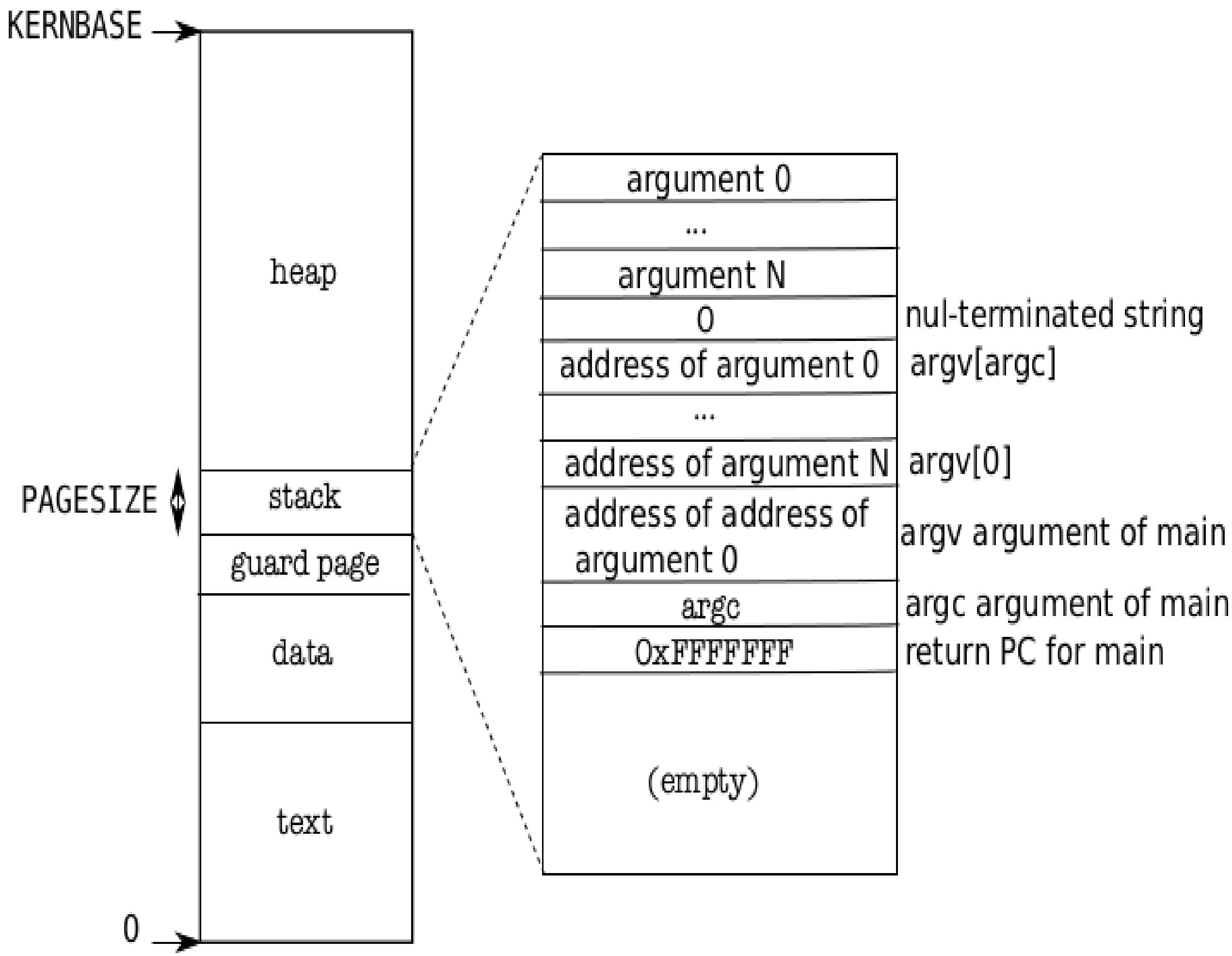
```
// Per-process state
struct proc {
  uint sz;                    // Size of process memory (bytes)
  pde_t* pgdir;               // Page table
  char *kstack;               // Bottom of kernel stack for this process
  enum procstate state;       // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  struct trapframe *tf;       // Trap frame for current syscall
  struct context *context;    // swtch() here to run process. Process’s context
  void *chan;                 // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files, used by open(), read(),...
  struct inode *cwd;          // Current directory, changed with “chdir()”
  char name[16];              // Process name (for debugging)
};
```

struct proc diagram: Very imp!



In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)



Memory Layout of a user process

Memory Layout of a user process

After exec()

Note the argc, argv on stack

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

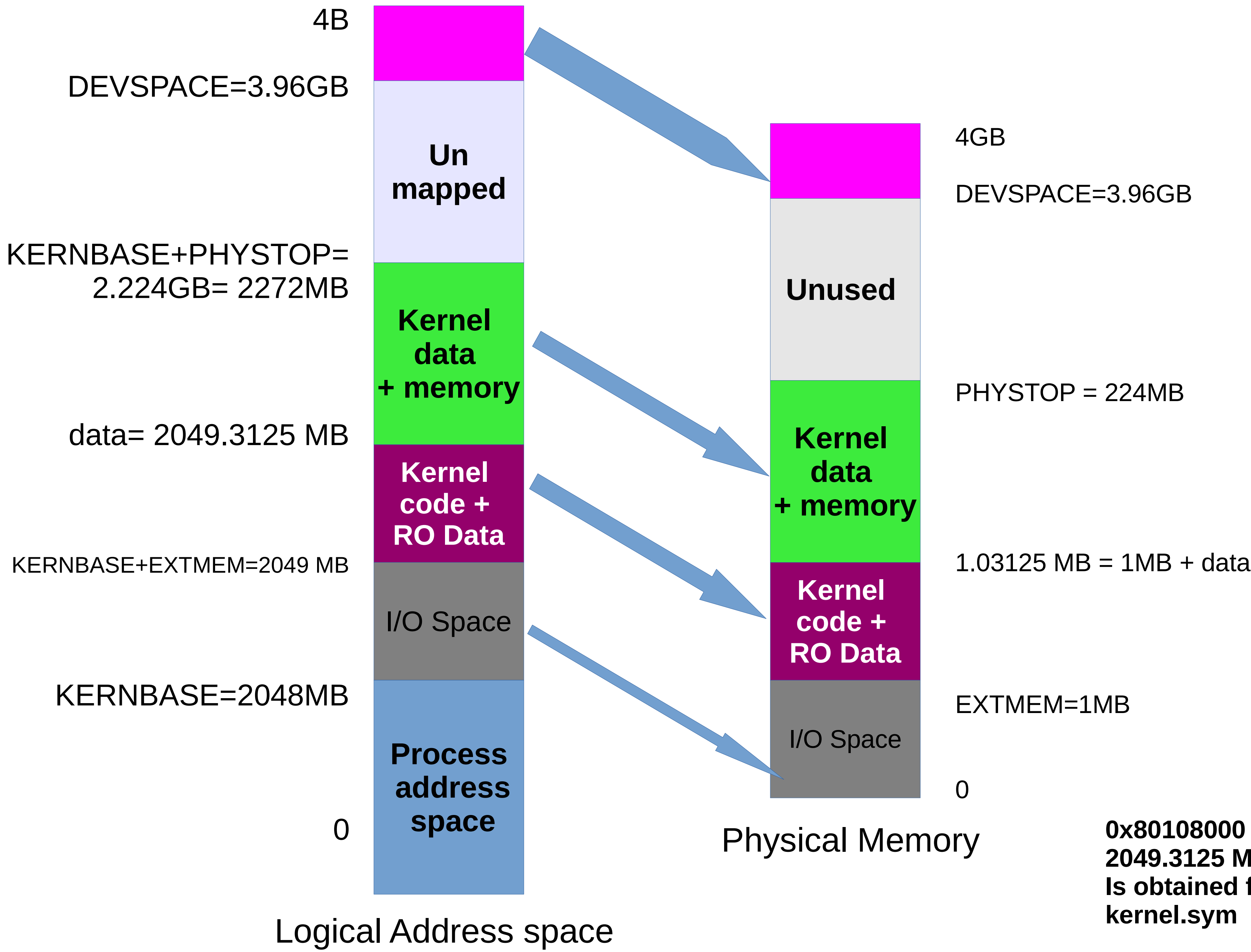
Handling Traps

Some basic steps

- Xv6.img is created by "make"
  - Contains bootsector, kernel code, kernel data
- QEMU boots using xv6.img
  - First it runs bootloader
  - Bootloader loads kernel of xv6 (from xv6.img)
  - Kernel starts running

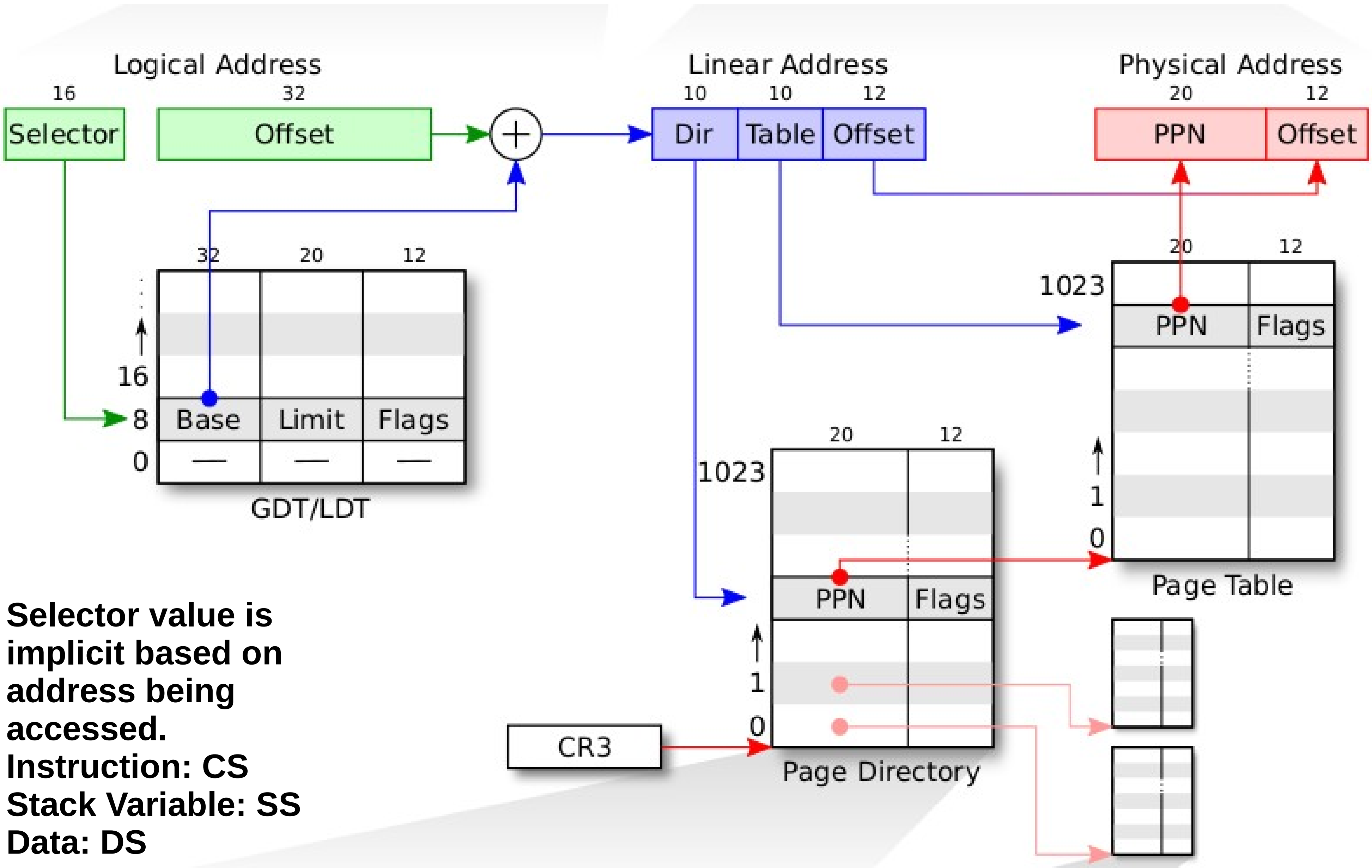
- Kernel running..
  - Kernel calls main() of kernel (NOT a C application!) & Initializes:
    - memory management data structures
    - process data structures
    - file handling data structures
    - Multi-processors
    - Multi-processor data structures
    - Interrupt Descriptor Table
    - ...
  - Then creates init() ...
  - Init() fork-execs shell

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings

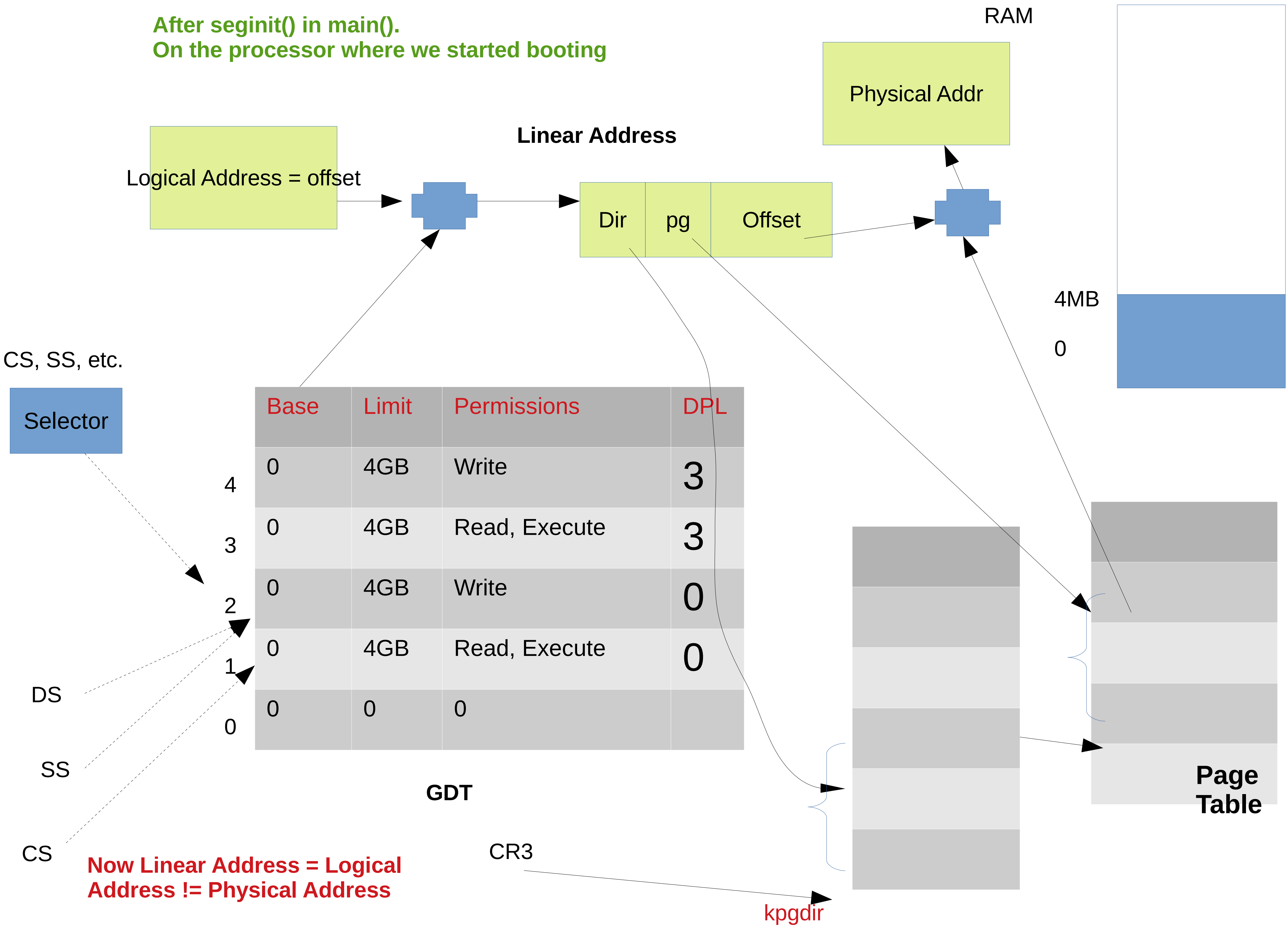


0x80108000 =data = 2049.3125 MB Is obtained from kernel.sym

Segmentation + Paging



Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.



After seginit() in main().
On the processor where we started booting
RAM
Physical Addr
Linear Address
Logical Address = offset
Dir
pg
Offset
4MB
0
CS, SS, etc.
Selector
Base
Limit
Permissions
DPL
4
0
4GB
Write
3
3
0
4GB
Read, Execute
3
2
0
4GB
Write
0
1
0
4GB
Read, Execute
0
0
0
0
0
DS
SS
CS
GDT
CR3
kpgdir
Page Table
Now Linear Address = Logical Address != Physical Address

Handling traps

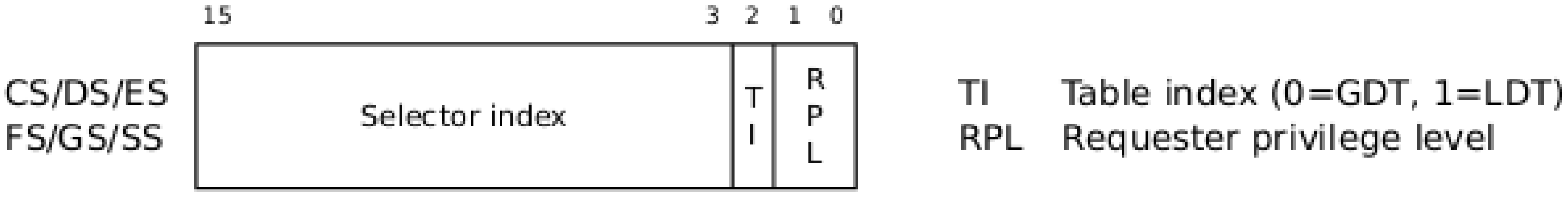
- Transition from user mode to kernel mode
  - On a system call
  - On a hardware interrupt
  - User program doing illegal work (exception)
- Actions needed, particularly w.r.t. to hardware interrupts
  - Change to kernel mode & switch to kernel stack
  - Kernel to work with devices, if needed
  - Kernel to understand interface of device

Handling traps

- **Actions needed on a trap**
  - **Save the processor’s registers (context) for future use**
  - **Set up the system to run kernel code (kernel context) on kernel stack**
  - **Start kernel in appropriate place (sys call, intr handler, etc)**
  - **Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)**

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.



Privilege level

- Changes automatically on
  - "int" instruction
  - hardware interrupt
  - exeception
- Changes back on
  - iret
- "int" 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt'
- Xv6 uses "int 64" for actual system calls

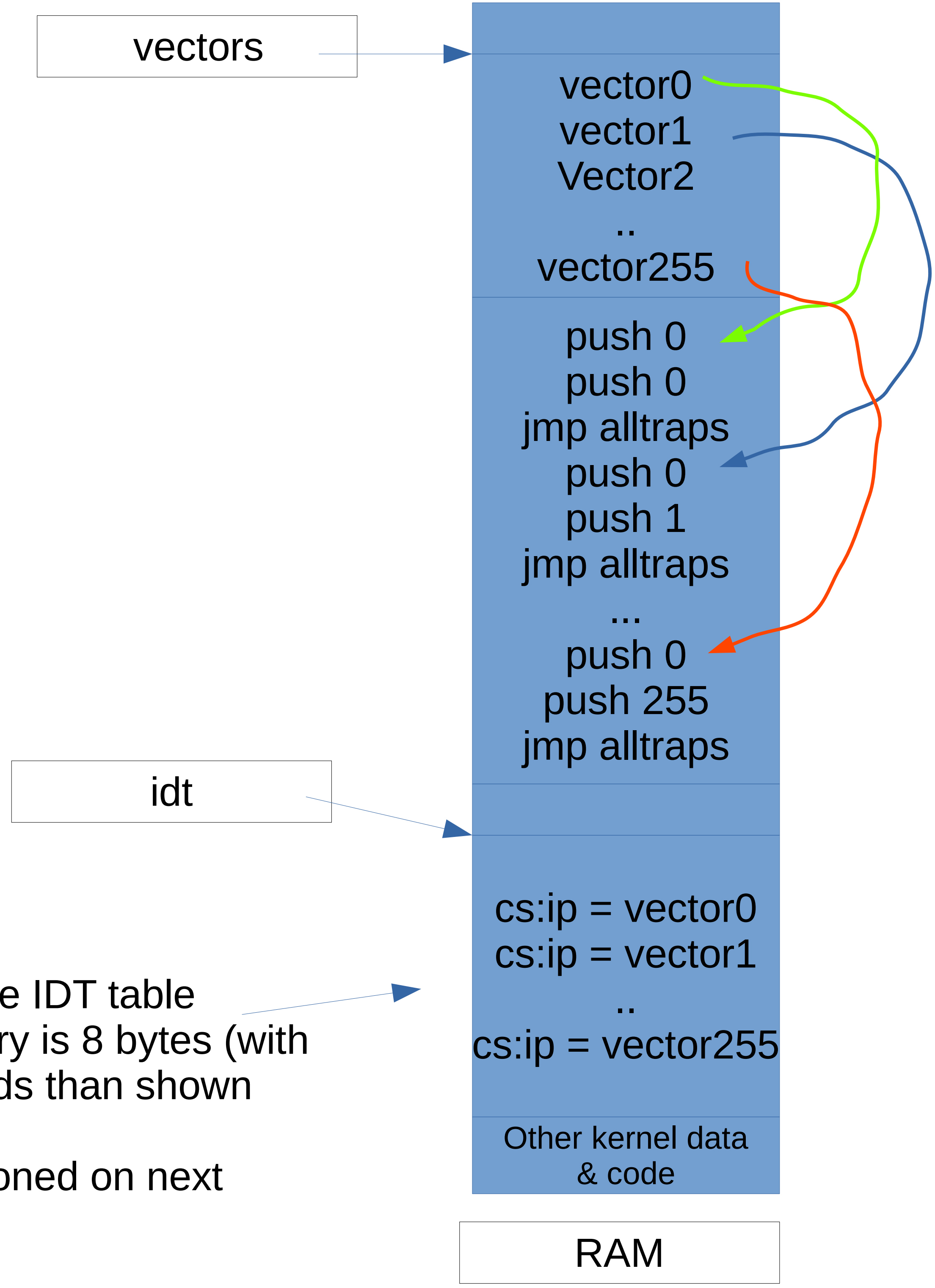
Interrupt Descriptor Table (IDT)

- IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Mapping
  - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
  - Xv6 maps the 32 hardware interrupts to the range 32-63
  - and uses interrupt 64 as the system call interrupt

IDT setup
done by tvinit() function

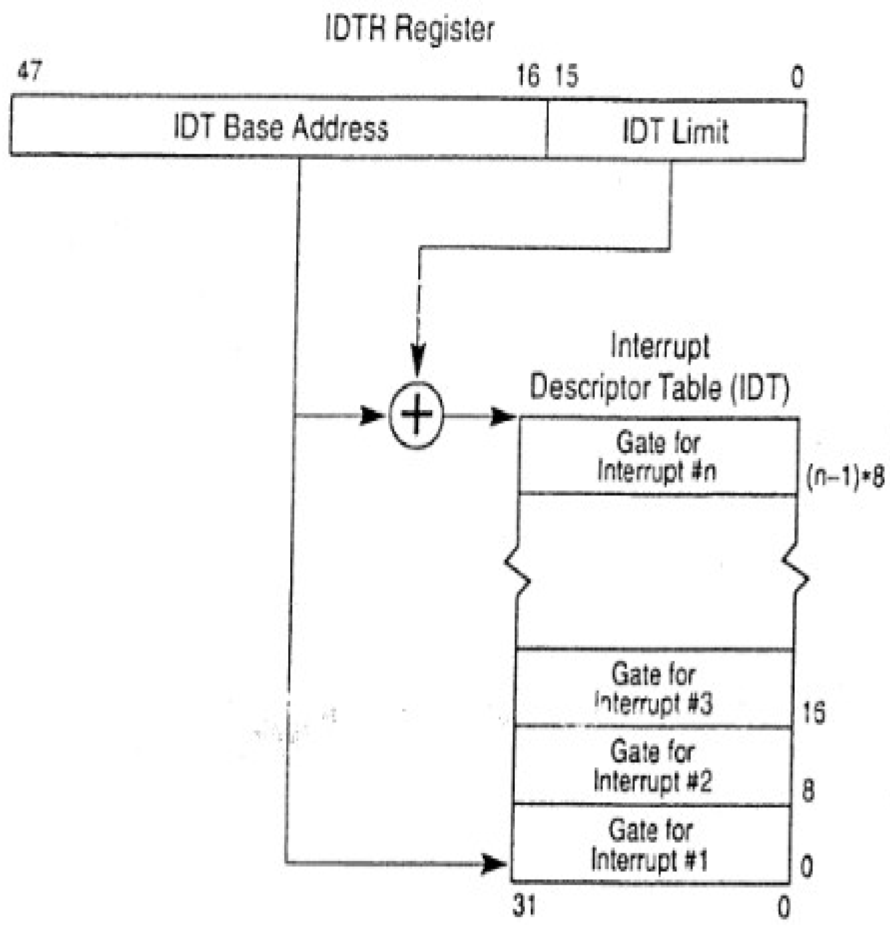
The array of “vectors”
And the code of
“push, ..jmp”
Is part of kernel image
(xv6.img)

The tvinit() is called during
kernel initialization



This is the IDT table
Each entry is 8 bytes (with
more fields than shown
here)
as mentioned on next
slides

IDTR and IDT



IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;// low 16 bits of offset in
segment
  uint cs : 16; // code segment selector
  uint args : 5; // # args, 0 for interrupt/trap gates
  uint rsv1 : 3; // reserved(should be zero I guess)
  uint type : 4; // type(STS_{IG32,TG32})
  uint s : 1; // must be 0 (system)
  uint dpl : 2; //descriptor(new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;// high bits of offset in segment
};
```

Setting IDT entries

```
void
tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER);
  /* value 1 in second argument --> don't disable
interrupts
         * DPL_USER means that processes can raise
this interrupt. */
    initlock(&tickslock, "time");
}
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d)              \
{                                                       \
  (gate).off_15_0 = (uint)(off) & 0xffff;              \
  (gate).cs = (sel);                                    \
  (gate).args = 0;                                      \
  (gate).rsv1 = 0;                                      \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;         \
  (gate).s = 0;                                         \
  (gate).dpl = (d);                                     \
  (gate).p = 1;                                         \
  (gate).off_31_16 = (uint)(off) >> 16;                \
}
```

Setting IDT entries

Vectors.S

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
 jmp alltraps
```

trapasm.S

```
#include "mmu.h"
# vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  Pushal
  ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n’th descriptor from the IDT, where n is the argument of int. (IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
  - Temporarily save %esp and %ss in CPU-internal registers, but only if the target segment selector’s PL < CPL.
    - Switching from user mode to kernel mode. Hence save user code’s SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set
  - Remember vectors.S
- So jump to 64th entry in vector’s

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- **Now stack contains**
- **ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi**
  - **This is the struct trapframe !**
  - **So the kernel stack now contains the trapframe**
  - **Trapframe is a part of kernel stcak**

trap()

```
void
trap(struct trapframe *tf)
{
 if(tf->trapno == T_SYSCALL){
  if(myproc()->killed)
   exit();
  myproc()->tf = tf;
  syscall();
  if(myproc()->killed)
   exit();
  return;
 }
 switch(tf->trapno){
 .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler

- Timer
  - wakeup(&ticks)
- IDE: disk interrupt
  - Ideintr()
- KBD
  - Kbdintr()
- COM1
  - Uatrintr()
- If Timer
  - Call yield() -- calls sched()
- If process was killed (how is that done?
  - Call exit()!

when trap() returns

- #Back in alltraps

```
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

Scheduler

Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
  - **The process is chosen out of “READY” state processes**
  - **Saving of context of “earlier” process and loading of context of “next” process needs to happen**
- **Questions**
  - **What are the different scenarios in which a scheduler called ?**
  - **What are the intricacies of “passing control”**
  - **What is “context” ?**

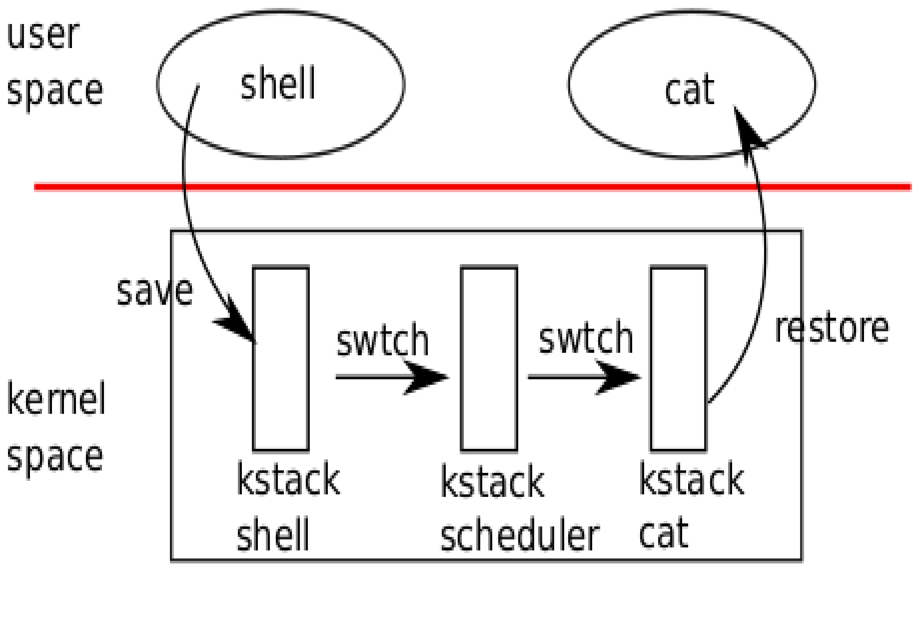
Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing

  F() { i++; j++;}
- P2 was doing

  G() { x--; y++; }
- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!



- User stack of process -> kernel stack of process
  - Switch to kernel stack
  - The normal sequence on any interrupt !
- Kernel stack of process -> kernel stack of scheduler
  - Why?
- Kernel stack of scheduler -> kernel stack of new process . Why?
- Kernel stack of new process -> user stack of new process

scheduler()

- Enable interrupts
- Find a RUNNABLE process. Simple round-robin!
- c->proc = p
- switchuvm(p) : Save TSS and make CR3 to point to new process pagedir
- p->state = RUNNING
- swtch(&(c->scheduler), p->context)

swtch

```
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

scheduler()

- swtch(&(c->scheduler), p->context)
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
  - The call does NOT return!
  - The new process P2 given by ‘p’ starts running !
  - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

  ret

  - where? in the case of first process – returns to forkret() because stack was setup like that !
  - in case of other processes, return where?
    - Return address given on kernel stack. But what's that?
    - The EIP in p->context
    - When was EIP set in p->context ?

scheduler()

- **Called from?**
  - **mpmain()**
  - **No where else!**
- **sched() is another scheduler function !**
  - **Who calls sched() ?**
    - **exit() - a process exiting calls sched ()**
    - **yield() - a process interrupted by timer calls yield()**
    - **sleep() - a process going to wait calls sleep()**

sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
/*A*/  mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
  - Note the arguments' order
  - p->context first, mycpu()->scheduler second
- swtch() is a function call
  - pushes address of */*A*/* on stack of current process p
  - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
  - when was mycpu()->scheduler set? Ans: during scheduler()!

sched() and schduler()

```
sched() {
...
  swtch(&p->context, mycpu()->scheduler); /* X */

}
```

```
scheduler(void) {
...
      swtch(&(c->scheduler), p->context);  / * Y */
}
```

- scheduler() saves context in c->scheduler, sched() saves context in p->context
- after swtch() call in sched(), the control jumps to Y in scheduler
  - Switch from process stack to scheduler's stack
- after swtch() call in scheduler(), the control jumps to X in sched()
  - Switch from scheduler's stack to new process's stack
- Set of co-operating functions

sched() and scheduler() as co-routines

- In sched()

```
swtch(&p->context, mycpu()->scheduler);
```

- In scheduler()

```
swtch(&(c->scheduler), p->context);
```

- These two keep switching between processes
- These two functions work together to achieve scheduling
- Using asynchronous jumps
- Hence they are co-routines

To summarize

- On a timer interrupt during P1
  - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
  - trap()->yield()
  - yield()->sched()
  - sched() -> swtch(&p->context, c->scheduler()
  - Stack changes to scheduler's kernel stack.
  - Switches to location "Y" in scheduler().
-

- Now the loop in scheduler()
  - calls switchkvm()
  - Then continues to find next process (P2) to run
  - Then calls switchuvm(p): changing the page table to the P2's page tables
  - then calls swtch(&c->scheduler, p2's->context)
  - Stack changes to P2's kernel stack.
  - P2 runs the last instruction it was was in ! Where was it?
    - mycpu()->intena = intena; in sched()
    - Then returns to the one who called sched() i.e. exit/sleep, etc
    - Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code

File Systems

Abhijit A M
abhijit.comp@coep.ac.in

What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-sysetm
- Design aspects of OS to implement the file system
  - On disk data structure
  - In memory kernel data structures

What is a file?

- A (dumb!) sequence of bytes (typically on a permanent storage:seconary, tertiary) , with
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- Types: Text files, binary files (one classification)
  - Text: All bytes are human readable
  - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc. (another classification)
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application (not kernel)

File types and kernel

- For example, MP4 file
  - *vlc* will do a open(...) on the file, and call read(...), interpret the contents of the file as movie and show movie
  - Kernel will simply provide open(...) read(...), write(...) to access file data
  - Meaning of the file contents is known to VLC and not to kernel!

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be *interpreted (by an application)* to be
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records , by database application, etc
  - A complexly organized, collection of records and bytes
    - E.g. a “ODT” or “DOCX” file
- What’s the role of OS in above mentioned file type, and organization?
  - Mostly NO role on Unixes, Linuxes!
  - They are handled by applications !
  - Types handled by OS: normal file, directory, block device file, character device file, FIFO file (named pipe), etc.
  - Also types handled by OS: executable file, non-executable file

File attributes

- Run

  $ ls -l

  on Linux

  To see file listing with different attributes

- Different OSes and file-systems provide different sets of file attributes
  - Some attributes are common to most, while some are different
  - E.g. name, size, owner can be found on most systems
  - “Executable” permission may not be found on all systems

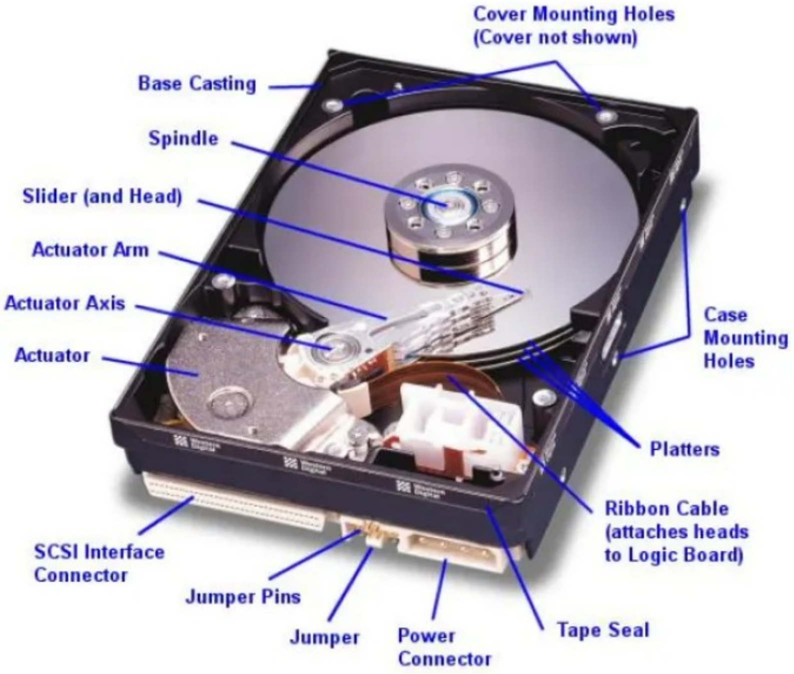
Access methods

- OS system calls may provide two types of access to files
  - Sequential Access
    - read next
    - write next
    - reset
    - no read after last write
      (rewrite)
  - Linux provides sequential access using open(), read(), write(), ...

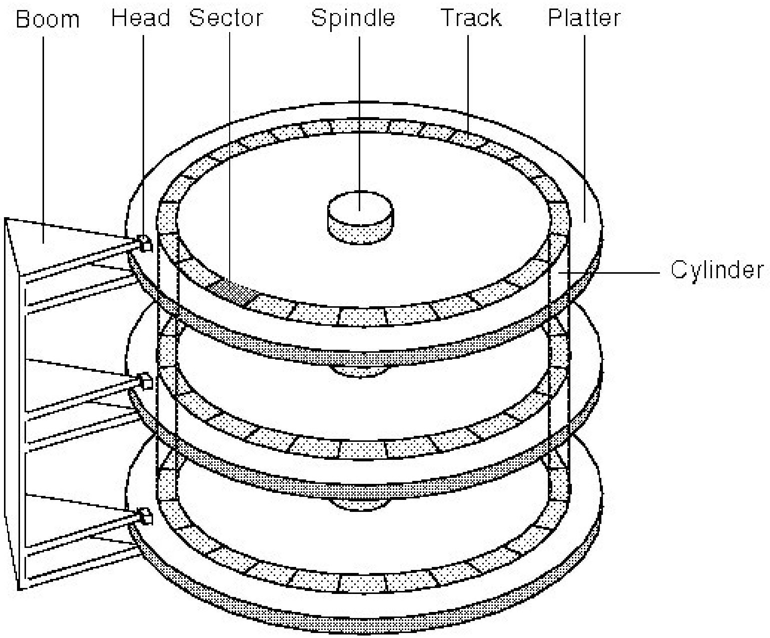
- Direct Access
  - read n
  - write n
  - position to n
    read next
    write next
  - rewrite n

  n = relative block number
- pread(), pwrite() on Linux
  - ssize_t pread(int fd, void *buf, size_t count, off_t offset);
  - ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);

Disk

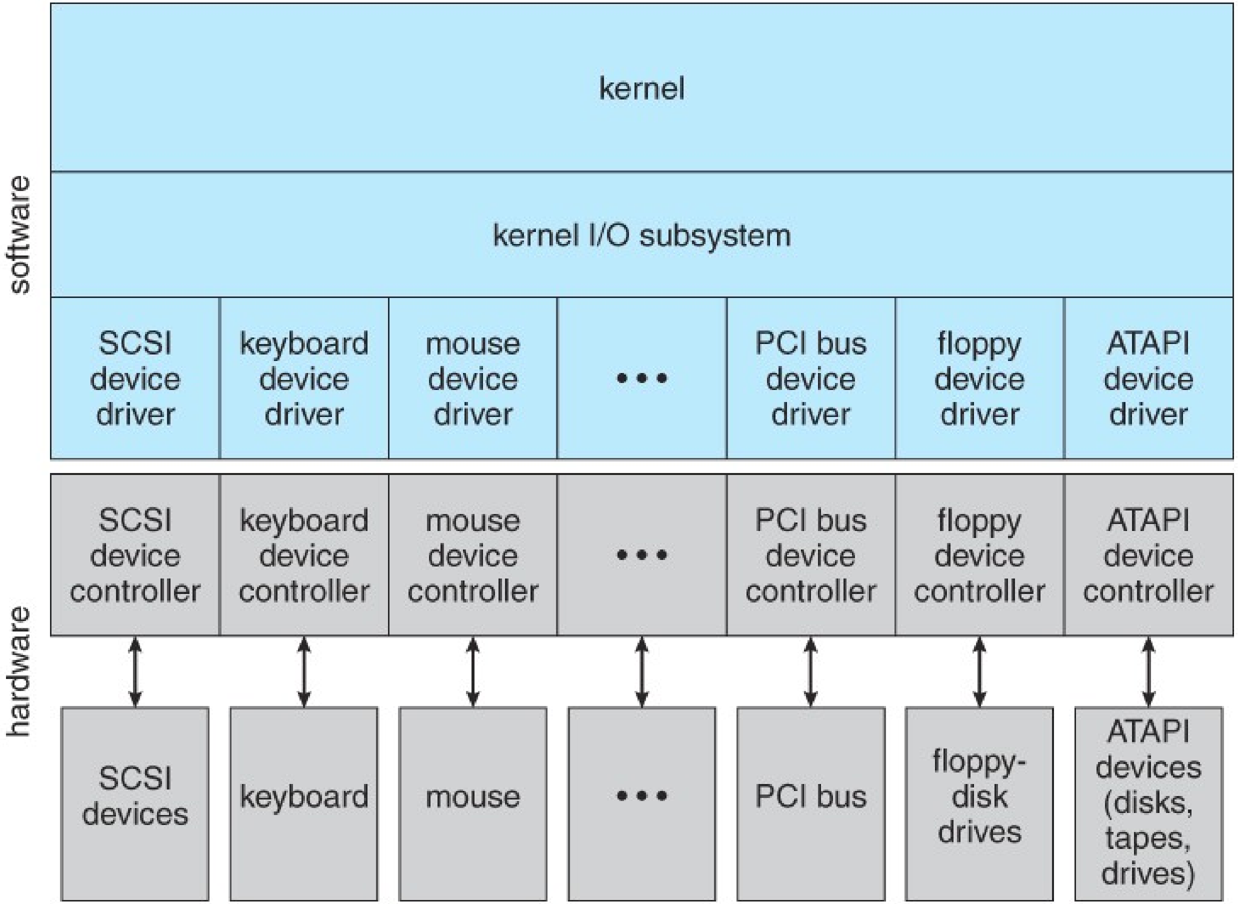


Cover Mounting Holes
(Cover not shown)
Base Casting
Spindle
Slider (and Head)
Actuator Arm
Actuator Axis
Actuator
Case
Mounting
Holes
Platters
Ribbon Cable
(attaches heads
to Logic Board)
SCSI Interface
Connector
Jumper Pins
Jumper
Power
Connector
Tape Seal

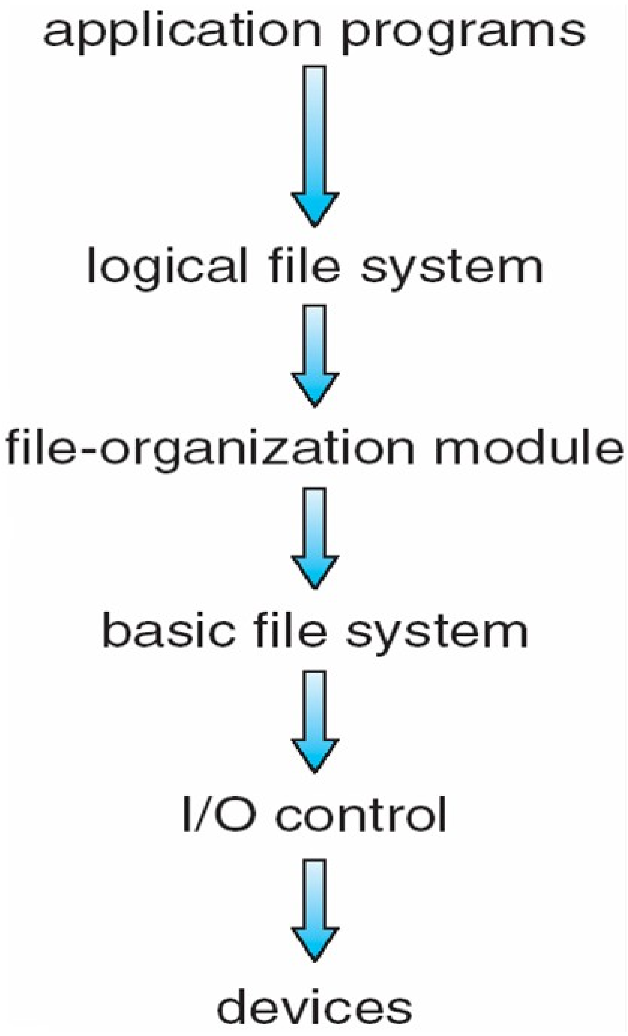


Boom
Head
Sector
Spindle
Track
Platter
Cylinder

Device Driver



File system implementation: layering



Application programs

```
int main() {
	char buf[128]; int count;
	fd = open(...):
	read(fd, buf, count);
}
-----
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
	file *fp = currproc->fdarray[fd];
	file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
	offset = fp->current-offset;
	translate offset into blockno;
	basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
	os_buffer *bp;
	sectorno = calculation on blockno;
	disk_driver_read(sectono, bp );
	move-process-to-wait-queue;
	copy-data-to-user-buffer(bp, buf);

}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
	issue instructions to disk controller
(often assembly code)
	to read sectorno  into specific
location;
}
```

XV6 does it slightely differently, but following the layering principle!

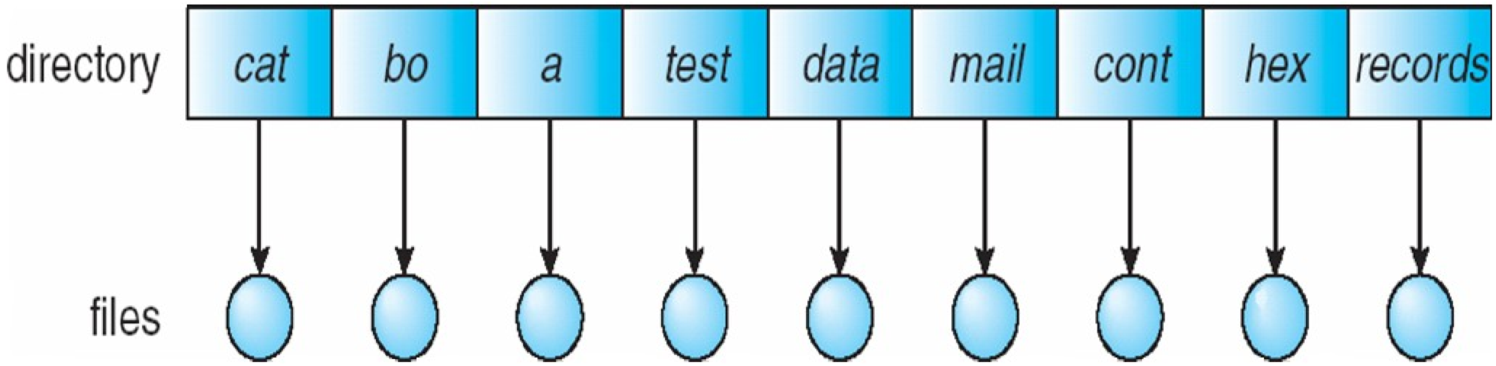
OS’s job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offerred by the device driver

Formatting

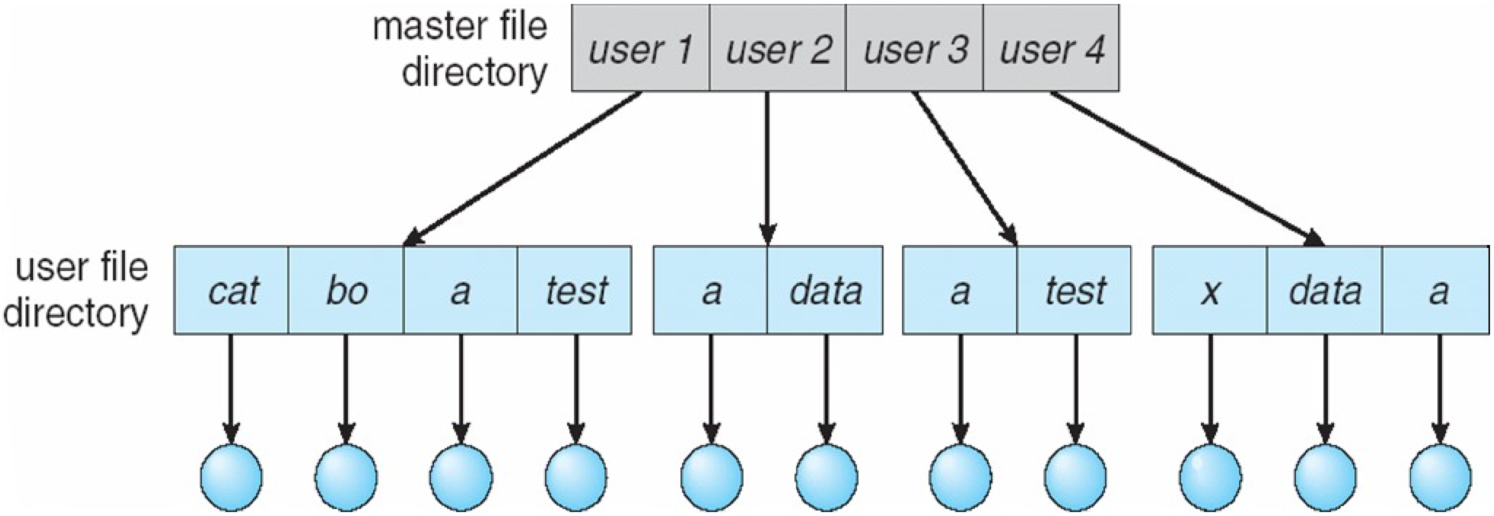
- Physical hard disk divided into partitions
  - Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
  - Like an un-initialized array, or sectors/blocks
- Formatting
  - Creating an initialized data structure on the partition, so that it can start storing the acyclic graph tree structure on it
  - Different formats depending on different implementations of the directory tree structure: ext4, NTFS, vfat, VxFS, ReiserFS, WafleFS, etc.
- Formatting happens on “a physical partition” or “a logical volume made available by volume manager”

Different types of "layouts"
Single level directory



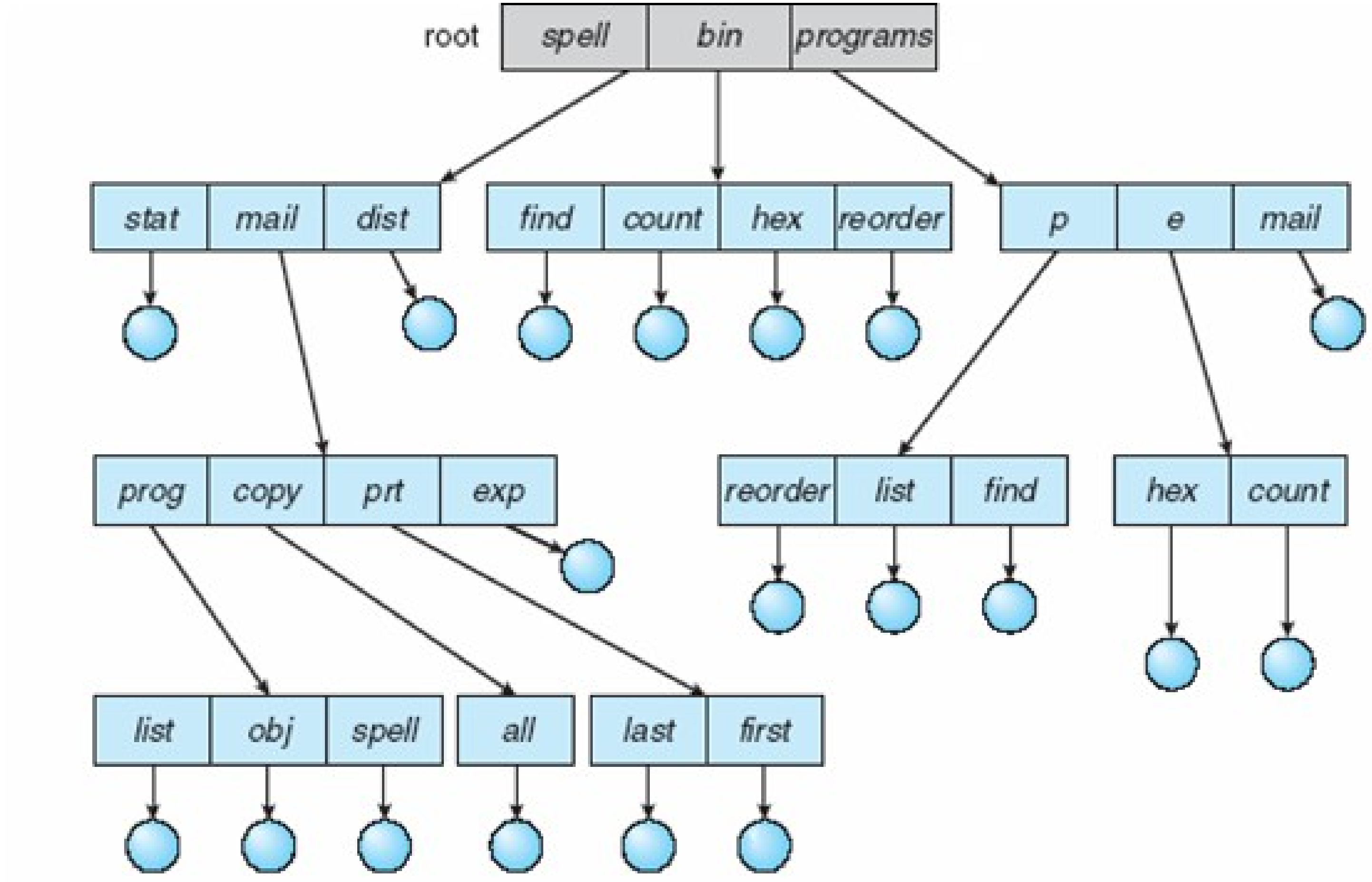
Naming problem
Grouping problem
Example: RT-11, from 1970s
https://en.wikipedia.org/wiki/RT-11

Different types of “layouts” Two level directory

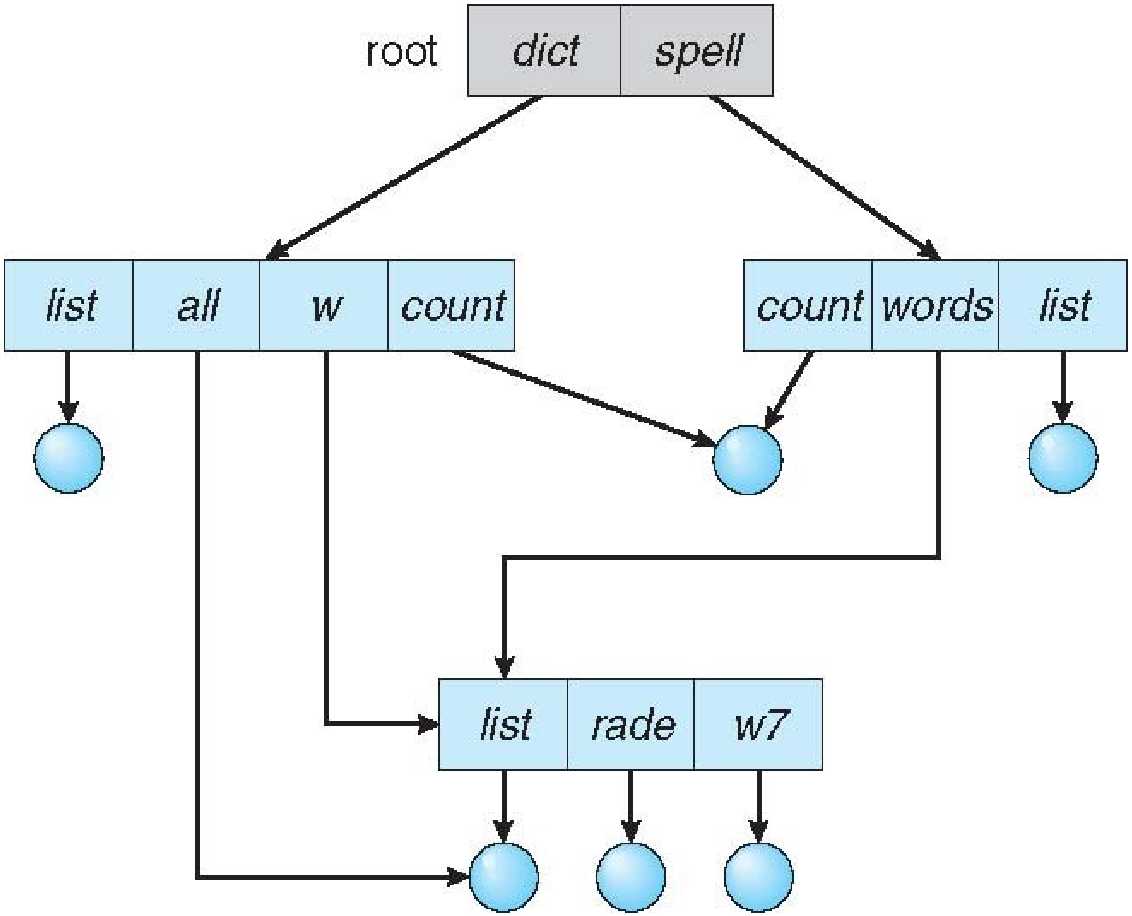


Path name
Can have the same file name for different user
Efficient searching
No grouping capability

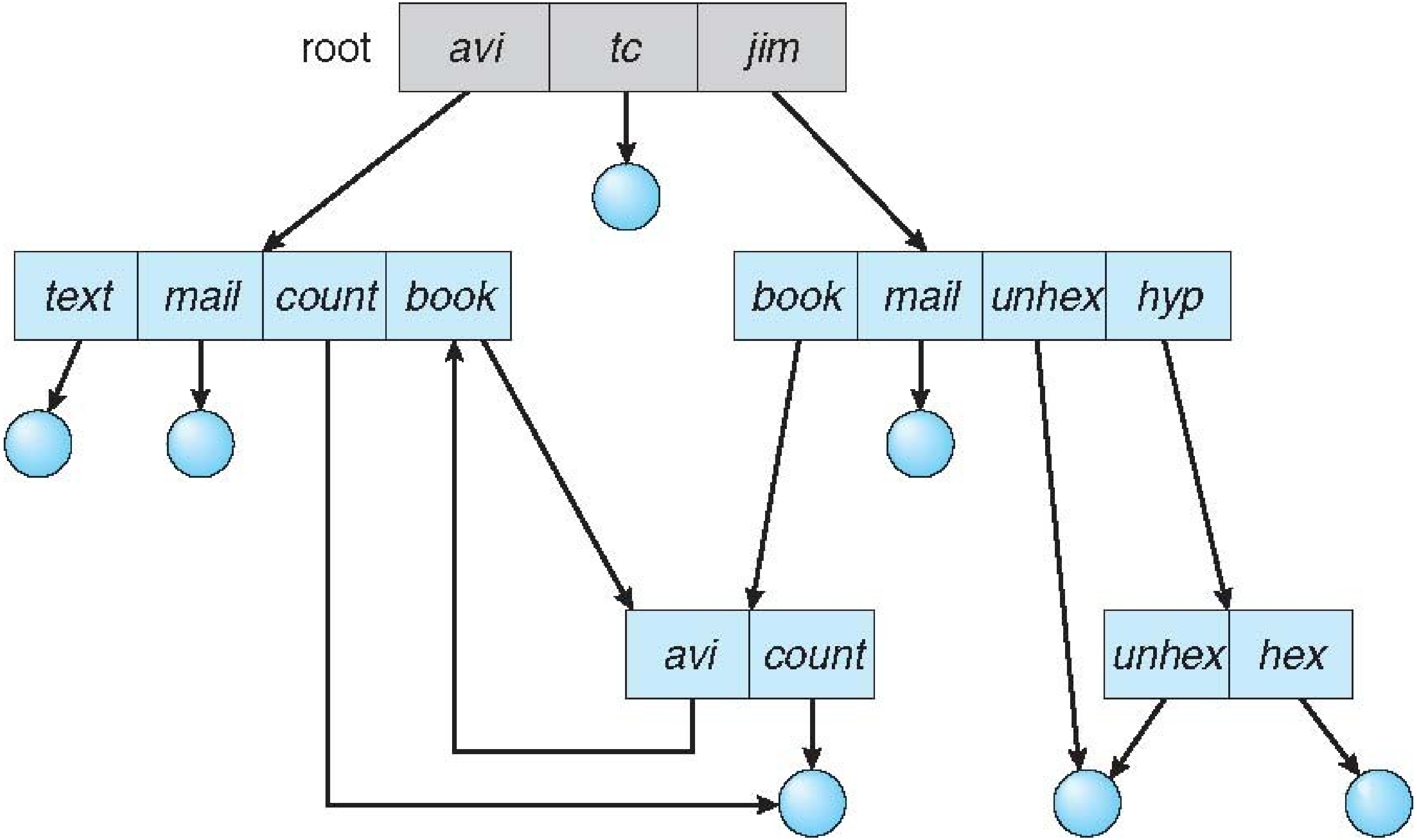
Tree Structured directories



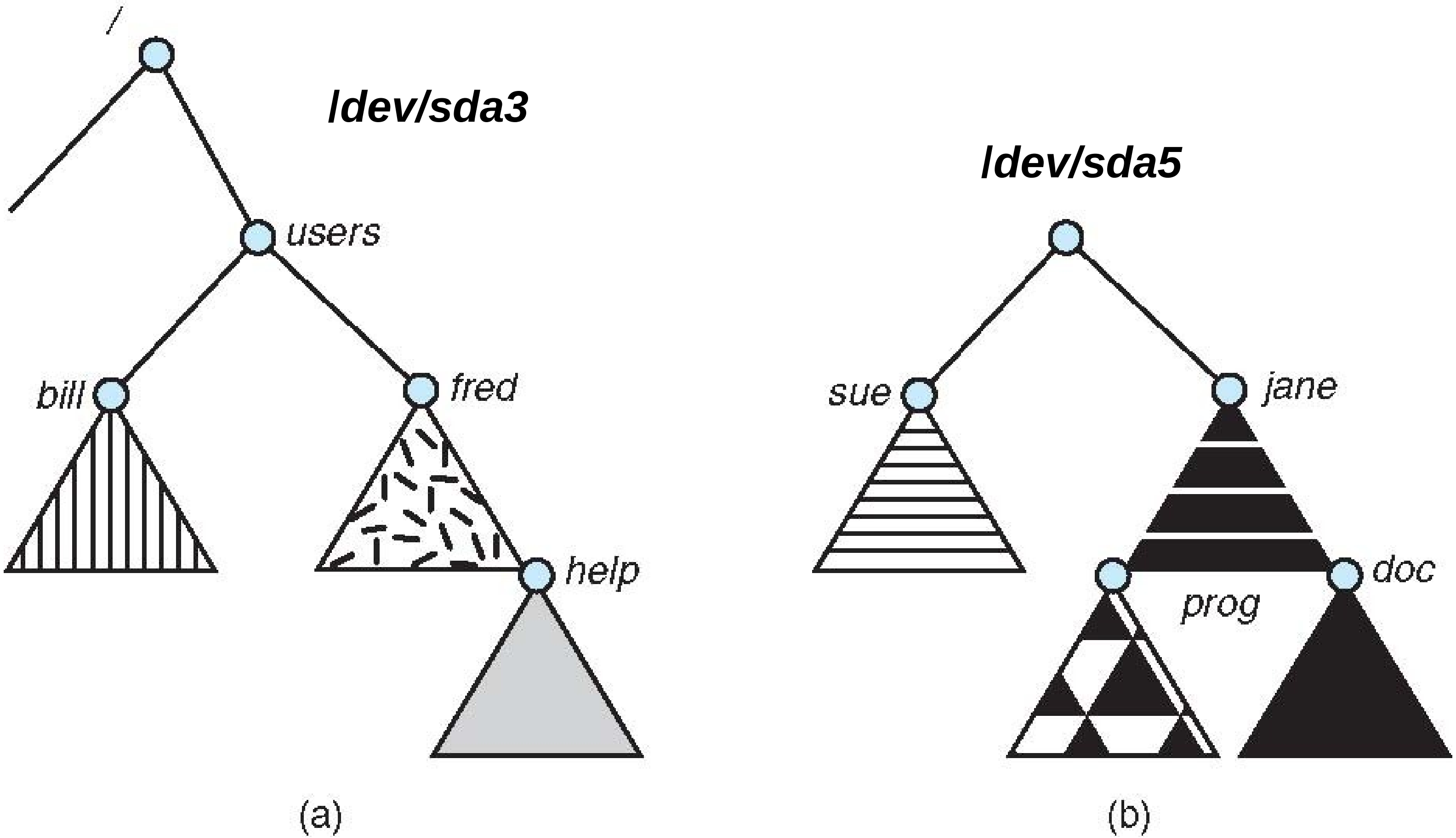
Acyclic Graph Directories



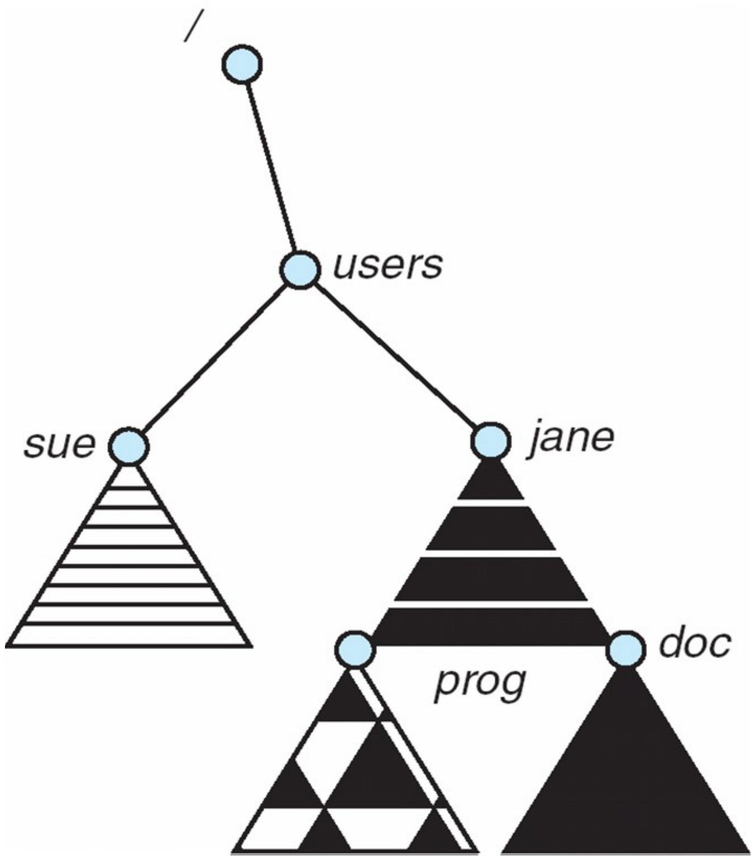
General Graph directory



Mounting of a file system: before



Mounting of a file system: after



$sudo mount */dev/sda5 /users*

Remote mounting: NFS

- Network file system
- $ sudo mount 10.2.1.2:/x/y /a/b
  - The /x/y partition on 10.2.1.2 will be made available under the folde /a/b on this computer
-

File sharing semantics

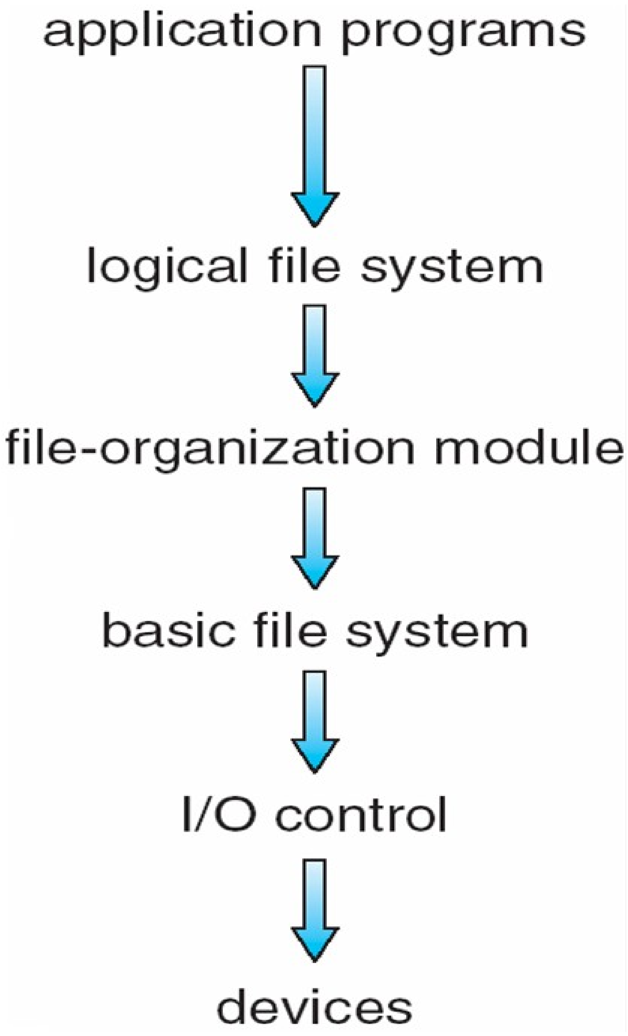
- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
  - Writes only visible to sessions starting after the file is closed

Implementing file systems

File system on disk

- What we know
  - Disk I/O in terms of sectors (512 bytes)
  - File system: implementation of acyclic graph using the linear sequence of sectors
    - Store a acyclic graph into array of “blocks”/“sectors”
  - Device driver available: gives sector/block wise access to the disk

File system implementation: layering



File system: Layering

- Device drivers manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- Basic file system given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- Logical file system manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
  - Directory management
  - Protection

File system implementation: Different problems to be solved

- **What to do at boot time, how to locate kernel ?**
- **How to store directories and files on the partition ?**
  - **Complex problem. Hiearchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)**
- **How to manage list of free sectors/blocks?**
- **How to store the summary information about the complete file system : #files, #free-blocks, ...**
- **How to mount a file system , how to unmount?**

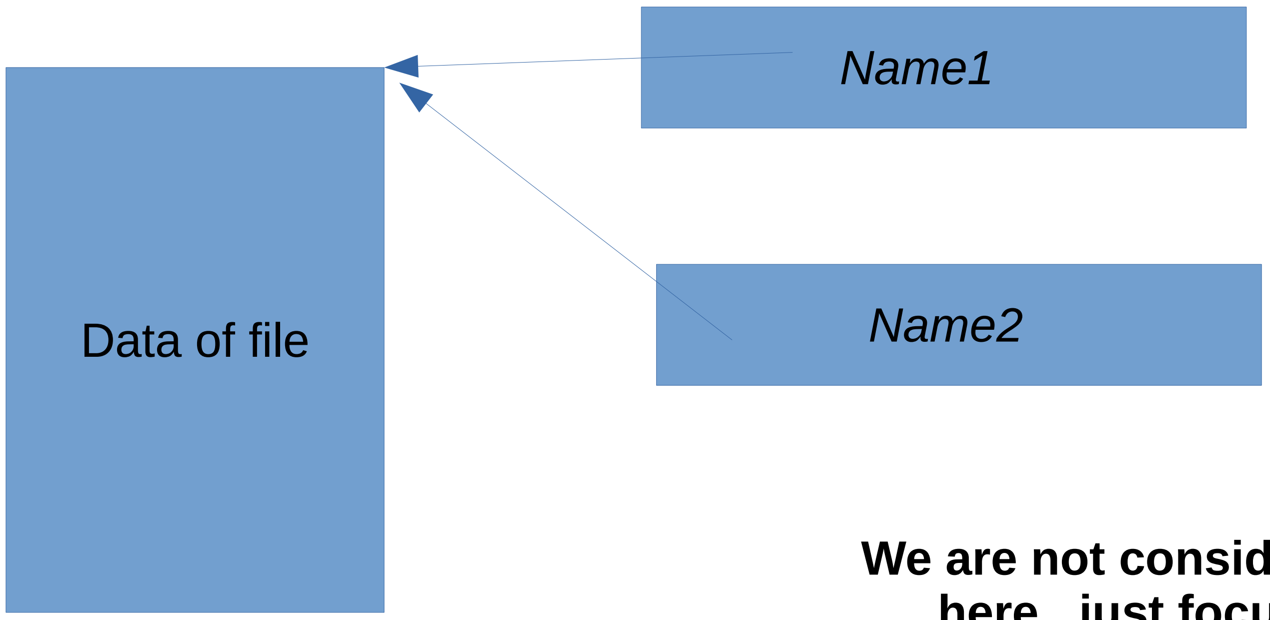
File system implementation: Different problems to be solved

- About storing a file, how to store
  - Data
  - Attributes
  - Name
  - Link count

The hard link problem

- Need to separate name from data !
  - */x/y   and /a/b* should be same file. How?
  - Both names should refer to same data !
  - Data is separated separately from name, and the name gives a “reference” to data
- What about attributes ?
  - They go with data! (not with name!)
- So solution was: indirection !

The hard link problem



We are not considering other problems here , just focussing on hard link problem

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Name is stored separately

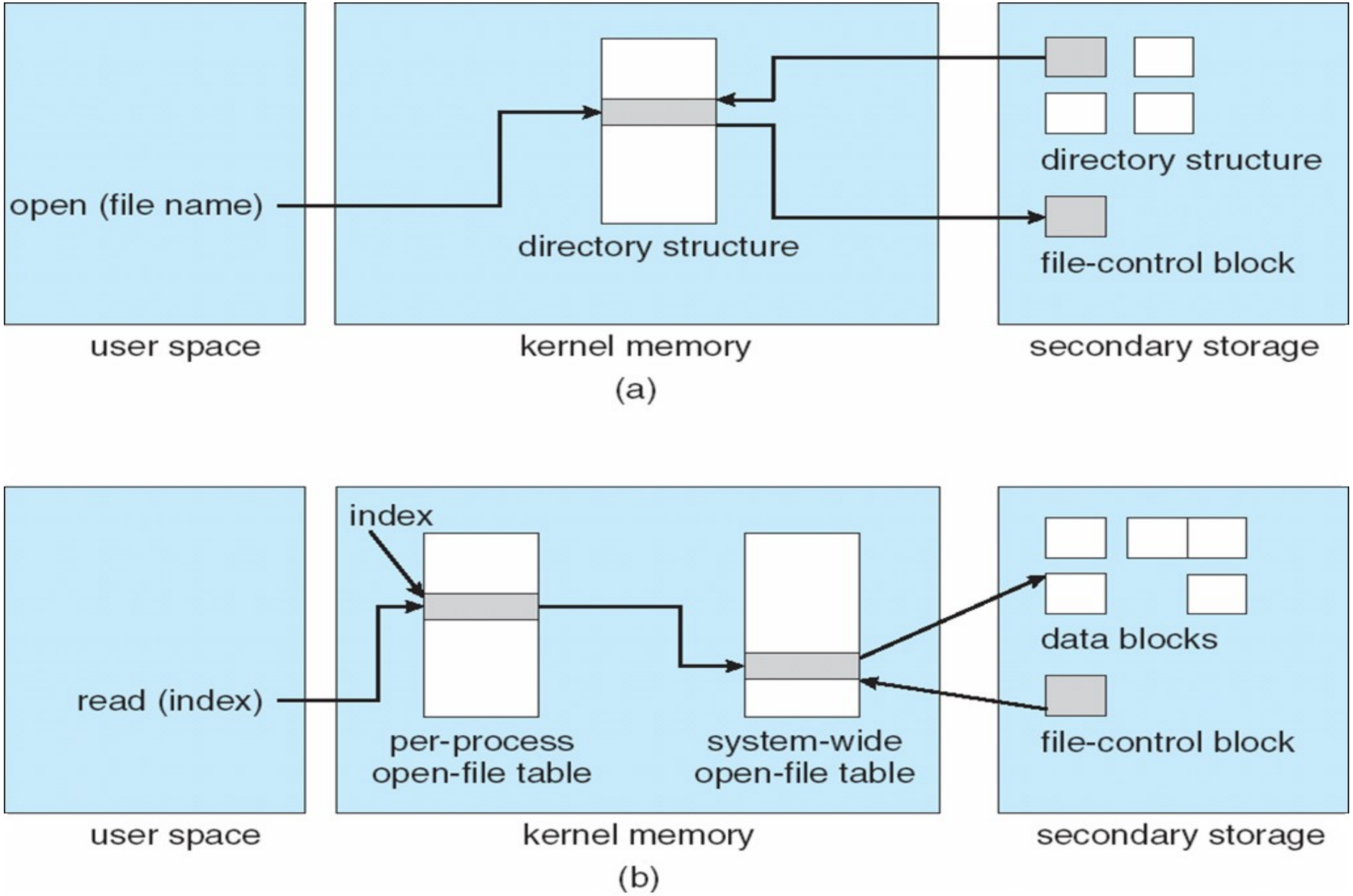
Where?

IN data block of directory

In memory data structures

- Mount table
  - storing file system mounts, mount points, file system types
- See next slide for “file” related data structures
- Buffers
  - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

At boot time

- Root partition
  - Contains the file system hosting OS
  - “mounted” at boot time – contains “/”
    - Normally can’t be unmounted!
- Check all other partitions
  - Specified in /*etc*/fstab on Linux
  - Check if the data structure on them is consistent
    - Consistent != perfect/accurate/complete

Directory Implementation

- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

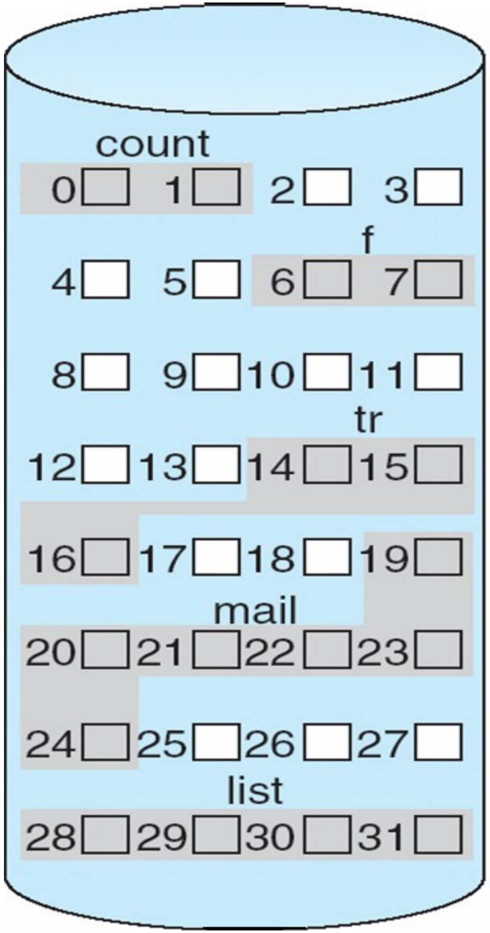
Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space



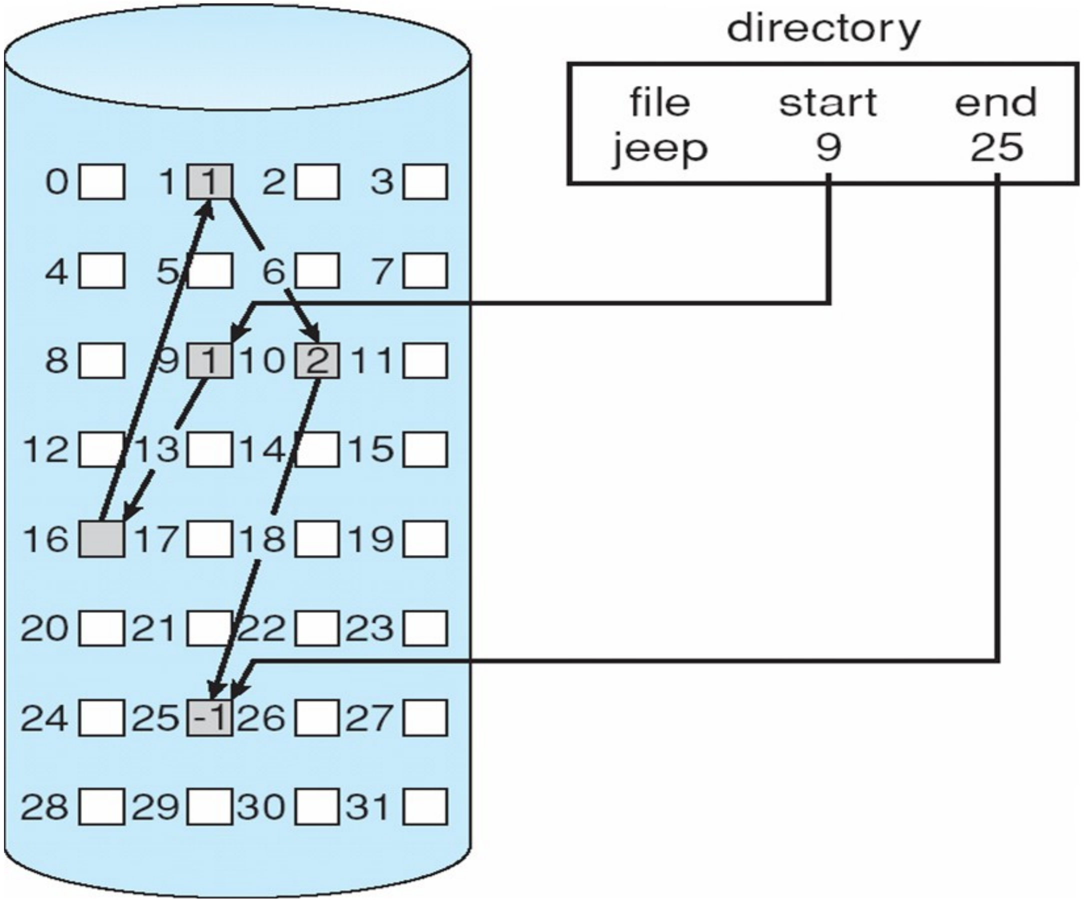
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

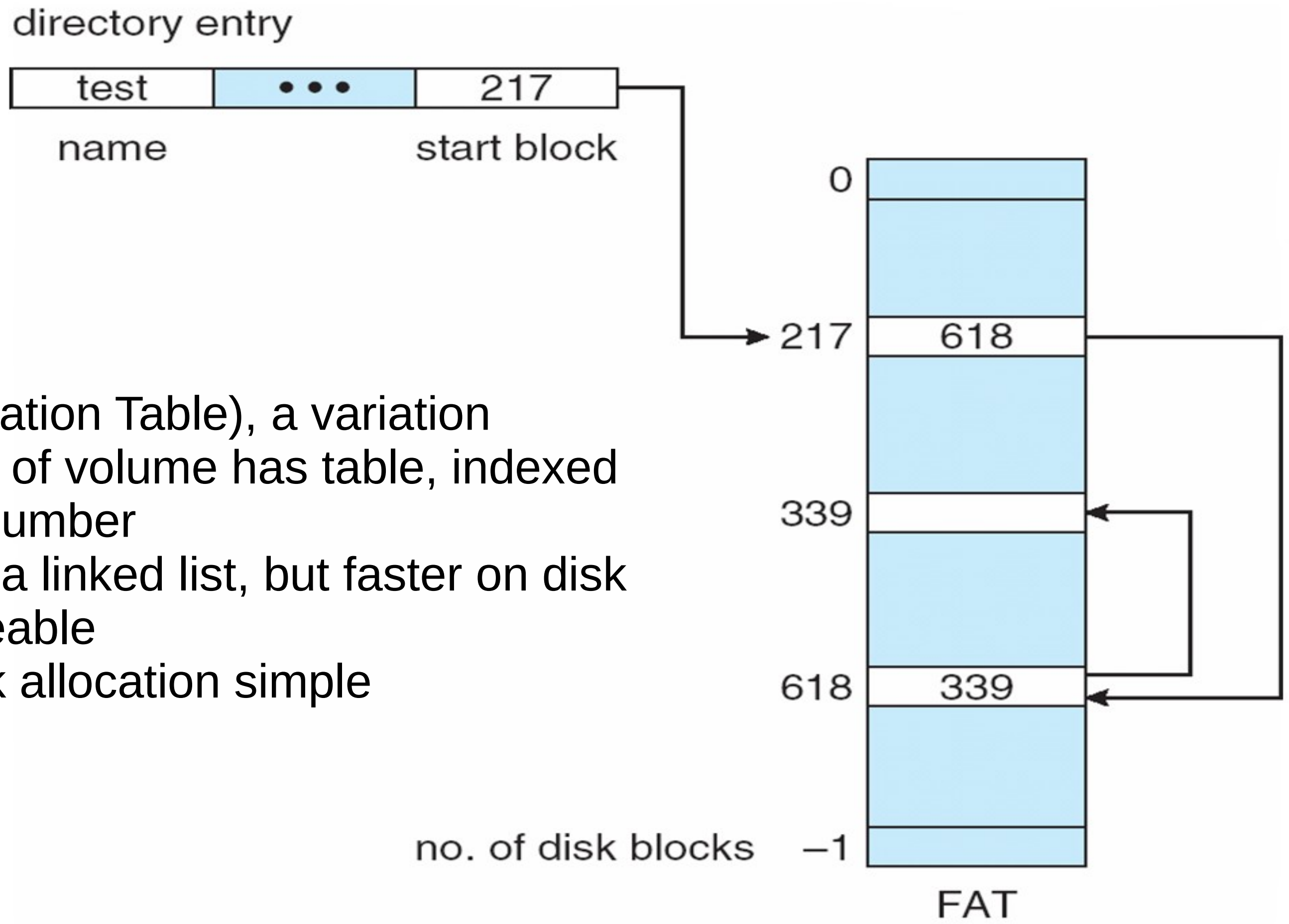


Linked allocation of blocks to a file

- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation

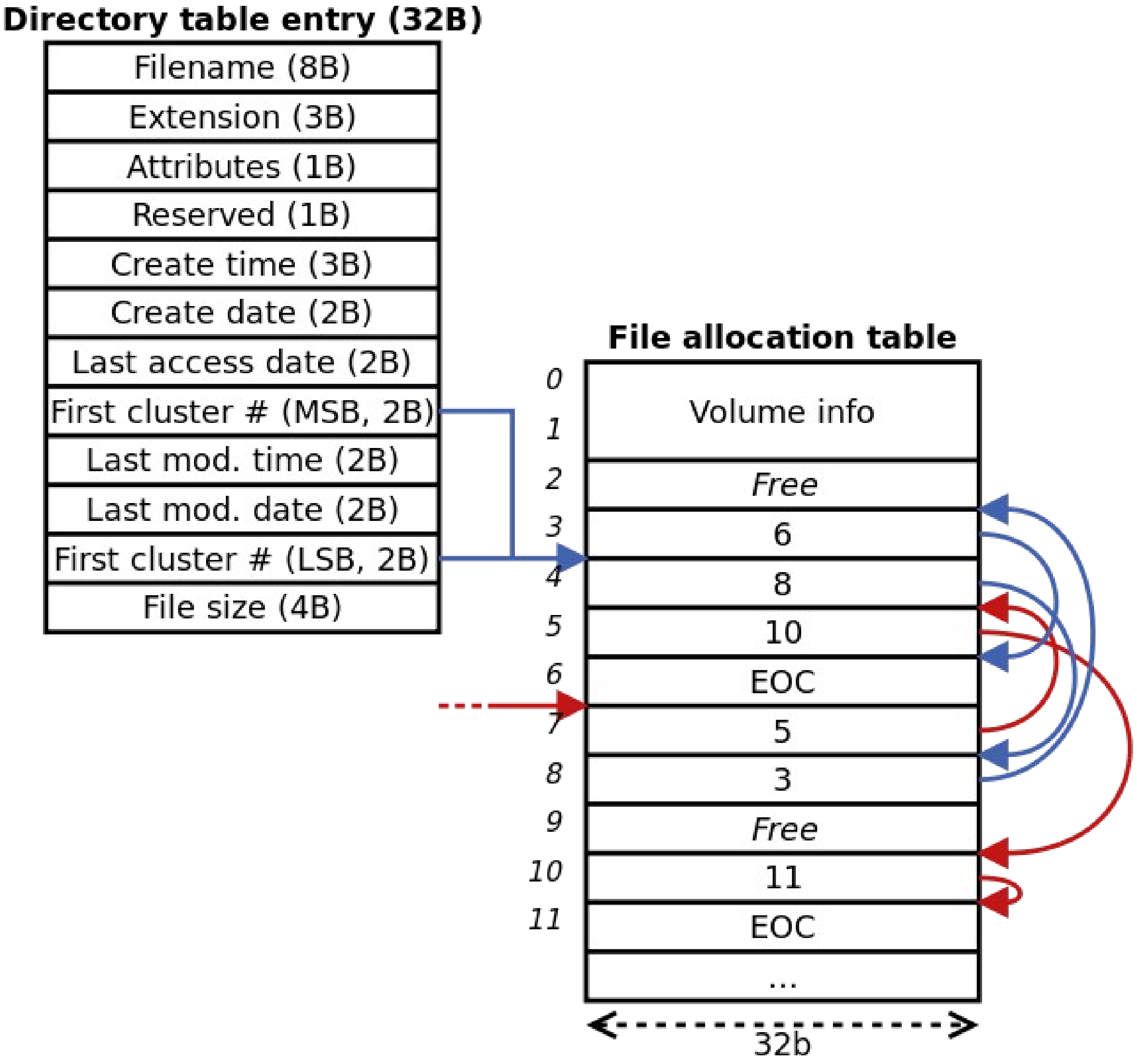
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table



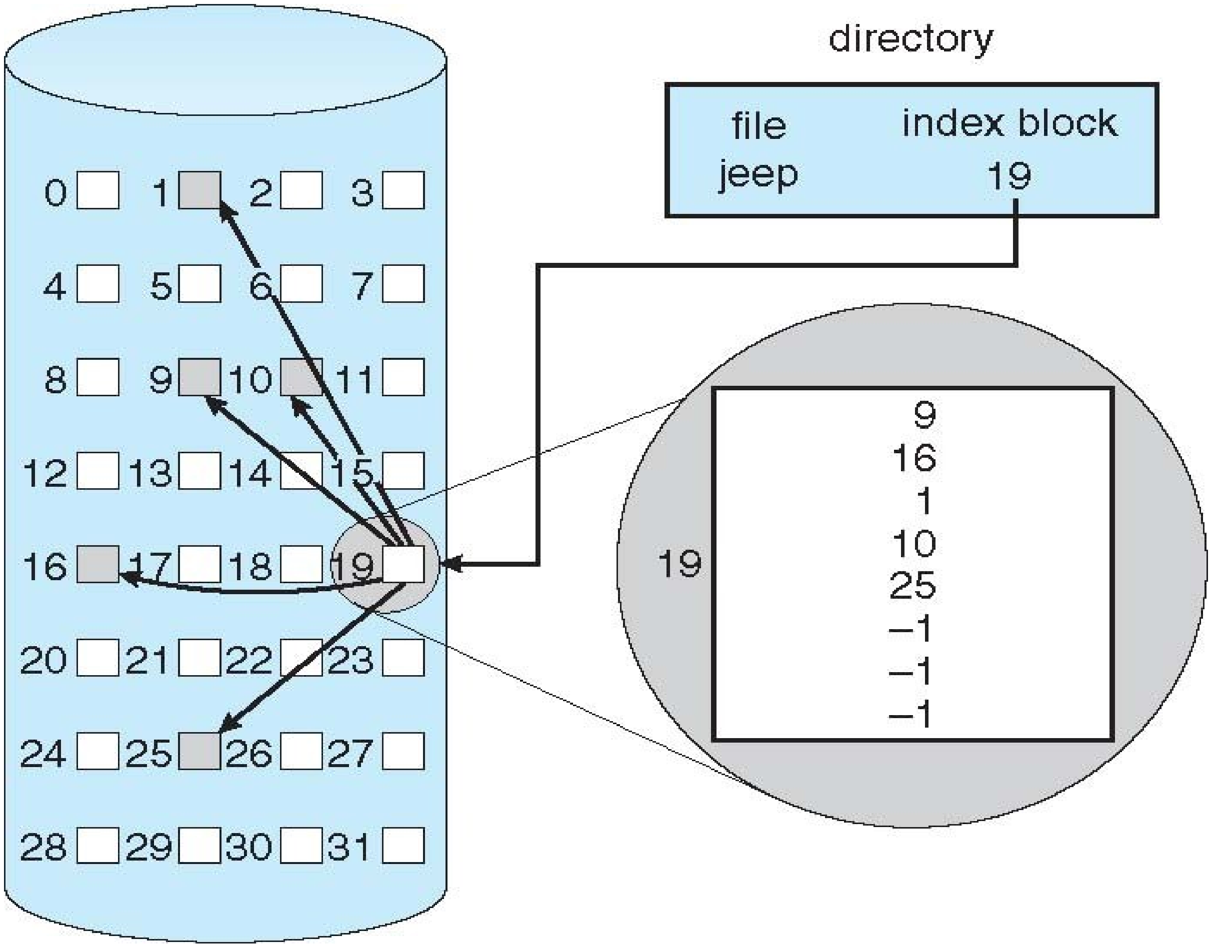
- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table



Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

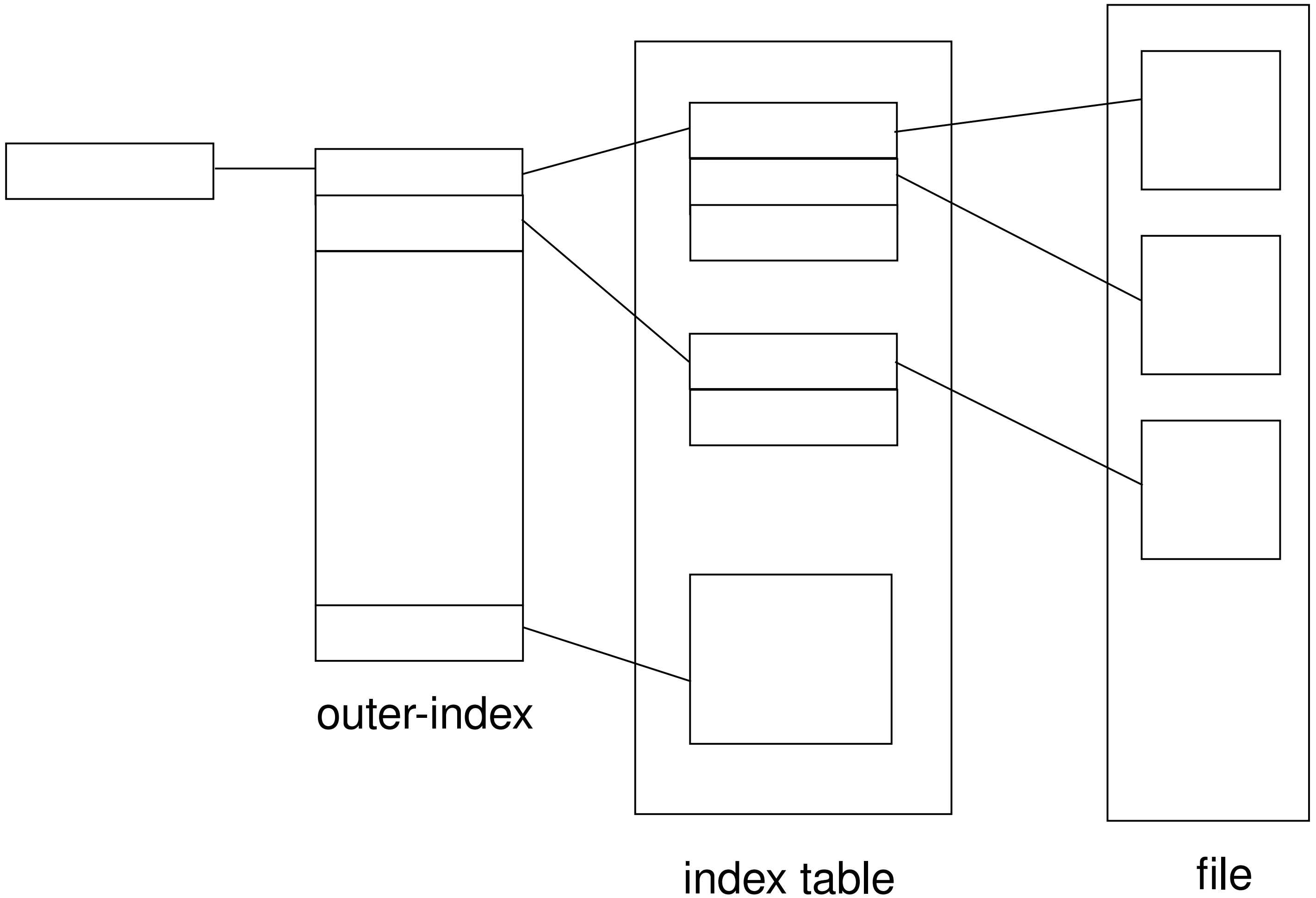
Indexed allocation



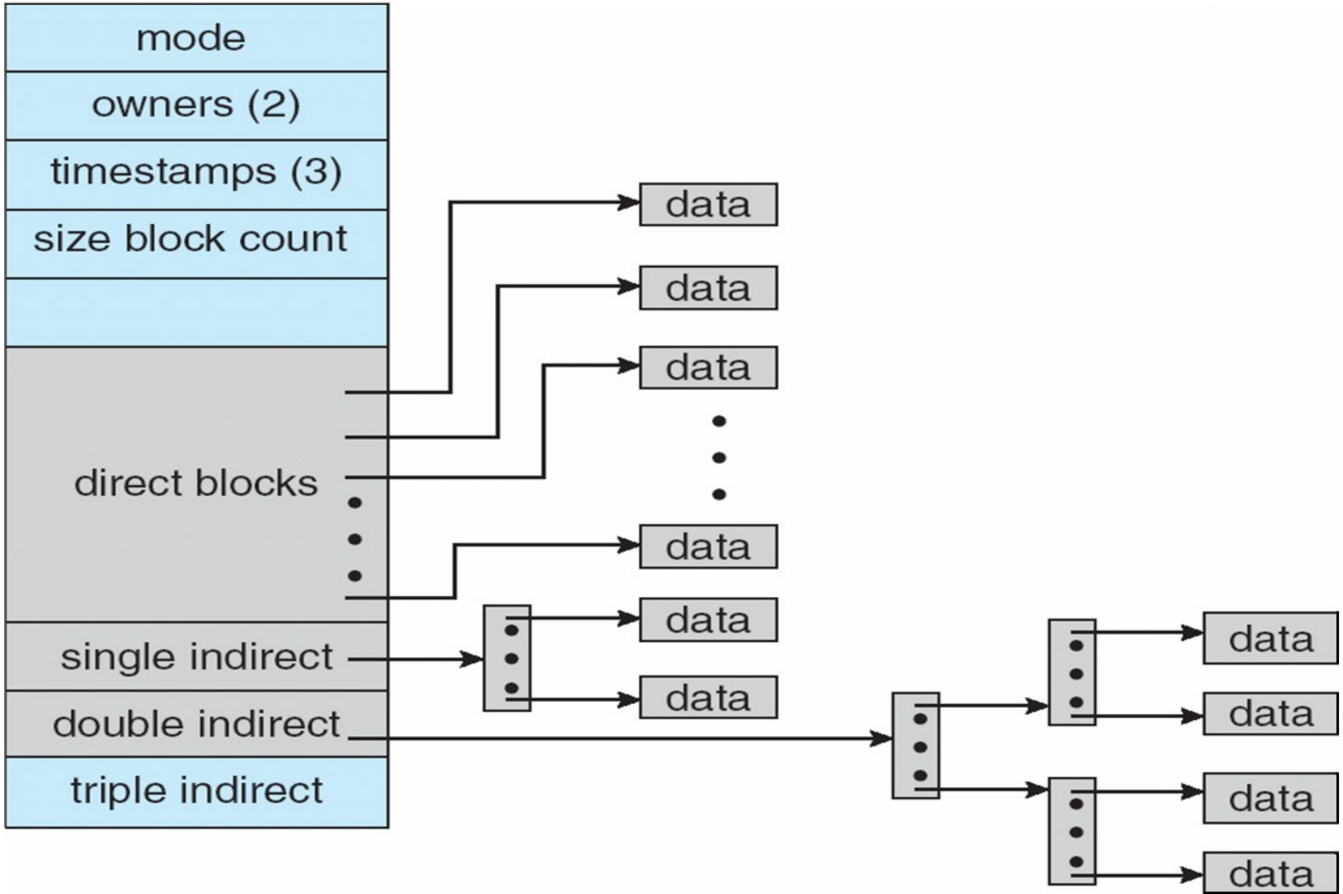
Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing



Unix UFS: combined scheme for block allocation



Free Space Management

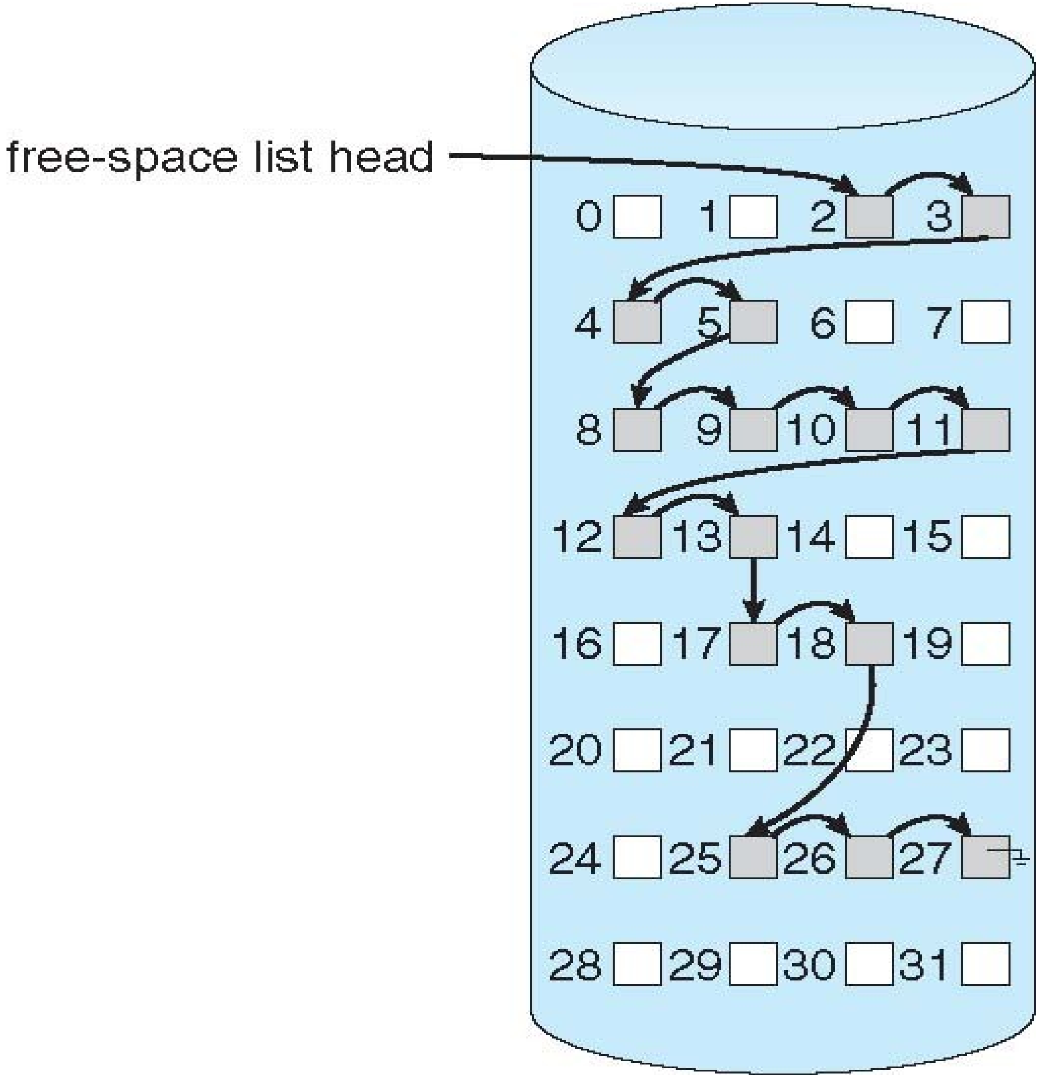
- **File system maintains free-space list to track available blocks/clusters**
  - **Bit vector or bit map (n blocks)**
  - **Or Linked list**

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

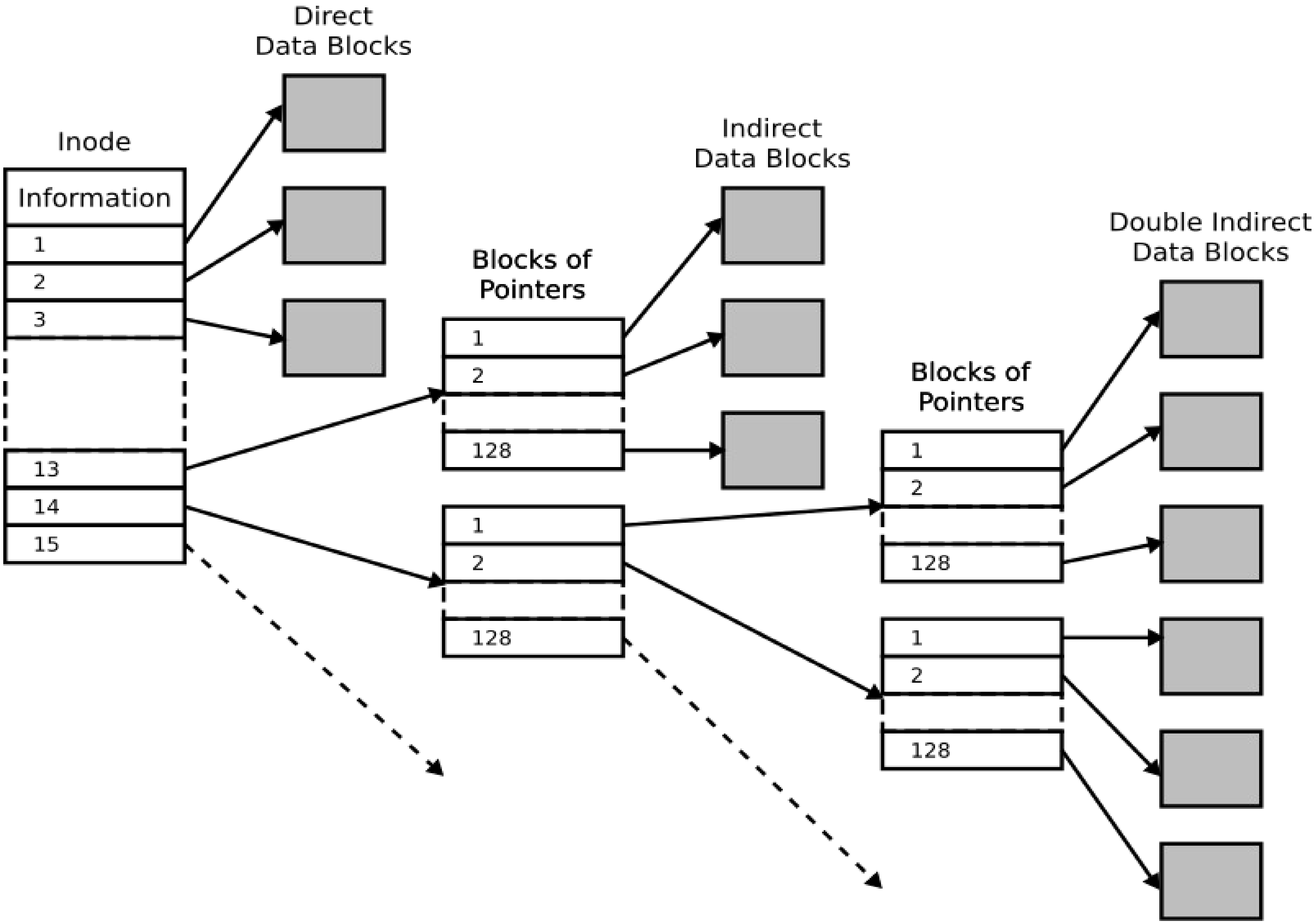
    001111001111110001100000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)



Ext2 FS layout

```
struct ext2_inode {
    __le16  i_mode;      /* File mode */
    __le16  i_uid;       /* Low 16 bits of Owner Uid */
    __le32  i_size;      /* Size in bytes */
    __le32  i_atime;     /* Access time */
    __le32  i_ctime;     /* Creation time */
    __le32  i_mtime;     /* Modification time */
    __le32  i_dtime;     /* Deletion Time */
    __le16  i_gid;       /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;    /* Blocks count */
    __le32  i_flags;     /* File flags */
```



Inode in ext2

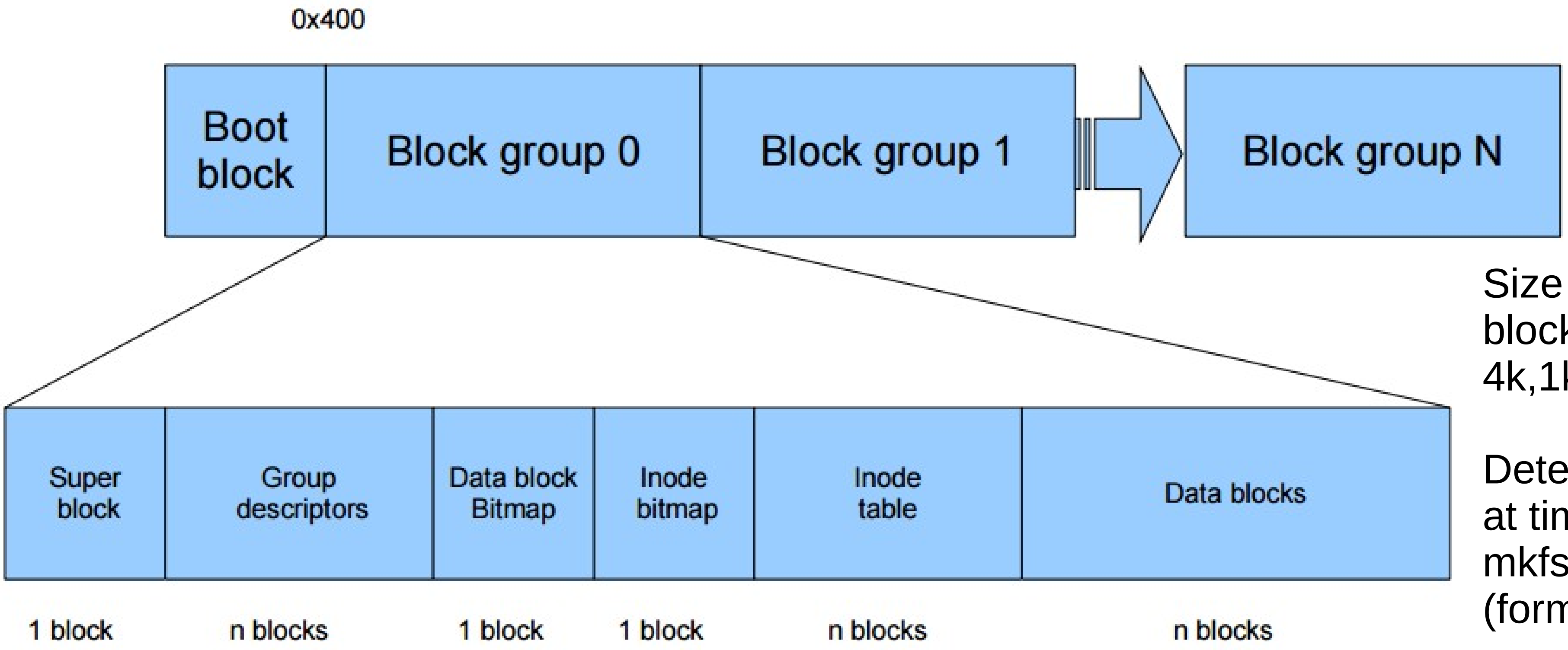
```
struct ext2_inode {
  ...
   union {
      struct {
         __le32  l_i_reserved1;
      } linux1;
      struct {
         __le32  h_i_translator;
      } hurd1;
      struct {
         __le32  m_i_reserved1;
      } masix1;
   } osd1;            /* OS dependent 1 */
   __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
   __le32  i_generation;   /* File version (for NFS) */
   __le32  i_file_acl; /* File ACL */
   __le32  i_dir_acl;  /* Directory ACL */
   __le32  i_faddr;    /* Fragment address */
```

```
struct ext2_inode {
  ...
   union {
       struct {
           __u8    l_i_frag;   /* Fragment number */            __u8    l_i_fsize;  /* Fragment size */
           __u16   i_pad1;            __le16  l_i_uid_high;   /* these 2 fields    */
           __le16  l_i_gid_high;   /* were reserved2[0] */
           __u32   l_i_reserved2;
       } linux2;
       struct {
           __u8    h_i_frag;   /* Fragment number */            __u8    h_i_fsize;  /* Fragment size */
           __le16  h_i_mode_high;            __le16  h_i_uid_high;
           __le16  h_i_gid_high;
           __le32  h_i_author;
       } hurd2;
       struct {
           __u8    m_i_frag;   /* Fragment number */            __u8    m_i_fsize;  /* Fragment size */
           __u16   m_pad1;            __u32   m_i_reserved2[2];
       } masix2;
   } osd2;             /* OS dependent 2 */
```

Ext2 FS Layout: Entries in directory’s data blocks

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

Ext2 FS Layout



Size of block 4k,1k, etc

Determined at time of mkfs (formatting)

Calculations done by “mkfs” like this

- Block size = 4KB (specified to mkfs)
- Number of total blocks = size of partition / 4KB
  - How to get size of partition ?
- 4KB = 4 * 1024 * 8 = 32768 bits
- Data Block Bitmap, Inode Bitmap are always one block
- So
  - size of a group is 32,768 Blocks
  - #groups = #blocks-in-partition / 32,768

```
struct ext2_super_block {
    __le32  s_inodes_count;      /* Inodes count */
    __le32  s_blocks_count;      /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
   __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16   s_inode_size;      /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

Traversal / path-name resolution

```
//resolving /a/b

s = read_superblock(); // struct
g = read_bg_descriptors(); // array
inode getinode(int n) {
 calculate the block number for n'th inode
 (using info from superblock, bg descriptors, block-size etc)
 read that block
 extract inode from block
 return inode
}
ino = 2
i = getinode(ino) ; //root
while (path not complete) {
    if (i is directory and path not complete)
  x = get-pathname-component(path); // give "a" from "/a/b", then "b" , etc
  read-data blocks of i'th inode
  search for x in the data-blocks
  if found
   ino = inode for found entry
  else
   return not-found
    else
        return not-found
}
```

Let’s see a program to read superblock of an ext2 file system.

Synchronization

My formulation

- OS = data structurs + synchronization
- Synchronization problems make writing OS code challenging
- Demand exceptional coding skills

Race problem

```
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {
        c++;
        c1++;
    }
}
void *thread2(void *arg) {
    while(run == 1) {
        c++;
        c2++;
    }
}
```

```
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    //fprintf(stdout, "Ending main\n");
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Race problem

- **On earlier slide**
  - **Value of c should be equal to c1 + c2, but it is not!**
  - **Why?**
- **There is a “race” between thread1 and thread2 for updating the variable c**
- **thread1 and thread2 may get scheduled in any order and *interrupted* any point in time**
- **The changes to c are not** atomic!
  - **What does that mean?**

Race problem

- C++, when converted to assembly code, could be

    mov c, r1

    add r1, 1

    mov r1, c

- Now following sequence of instructions is possible among thread1 and thread2

    thread1: mov c, r1

    thread2: mov c, r1

    thread1: add r1, 1

    thread1: mov r1, c

    thread2: add r1, 1

    thread2: mov r1, c

- What will be value in c, if initially c was, say 5?
    - It will be 6, when it is expected to be 7. Other variations also possible.

Races: reasons

- Interruptible kernel
  - If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
  - This introduces concurrency
- Multiprocessor systems
  - On SMP systems: memory is shared, kernel and process code run on all processors
  - Same variable can be updated parallely (not concurrently)
- What about non-interruptible kernel on multiprocessor systems?
- What about non-interruptible kernel on uniprocessor systems?

Critical Section problem

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

Critical Section Problem

- Consider system of n processes {p0, p1, … pn-1}
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

Expected solution characteristics

- **1. Mutual Exclusion**
  - **If process Pi is executing in its critical section, then no other processes can be executing in their critical sections**
- **2. Progress**
  - **If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely**
- **3. Bounded Waiting**
  - **A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted**
    - **Assume that each process executes at a nonzero speed**
    - **No assumption concerning relative speed of the n processes**

suggested solution - 1

```
int flag = 1;
void *thread1(void *arg) {
    while(run == 1) {
        while(flag == 0)
        ;
        flag = 0;
        c++;
        flag = 1;
        c1++;
    }
}
```

- What's wrong here?
- Assumes that

  while(flag ==) ; flag = 0

  will be atomic

suggested solution - 2

```
int flag = 0;
void *thread1(void *arg) {
    while(run == 1) {
        if(flag)
            c++;
        else
            continue;
        c1++;
        flag  = 0;
    }
}
```

```
void *thread2(void *arg) {
    while(run == 1) {
        if(!flag)
            c++;
        else
            continue;
        c2++;
        flag = 1;
    }
}
```

Peterson's solution

- **Two process solution**
- **Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted**
- **The two processes share two variables:**

  int turn;

  Boolean flag[2]
- **The variable turn indicates whose turn it is to enter the critical section**
- **The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!**

Peterson's solution

```
do {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] && turn == j)
        ;
  critical section
  flag[i] = FALSE;
  remainder section
} while (TRUE);
```

- Provable that
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Basically two operations (read/write) done atomically in hardware

Solution using test-and-set

lock = false; //global

```
do {
   while ( TestAndSet (&lock ))
          ;   // do nothing
    //    critical section
   lock = FALSE;
   //      remainder section
 } while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean
     *target)
   {
        boolean rv = *target;
        *target = TRUE;
        return rv:
   }
```

Solution using swap

```
lock = false; //global

do {
   key = true
    while ( key == true))
      swap(&lock, &key)
     //    critical section
    lock = FALSE;
    //      remainder section
  } while (TRUE);
```

Spinlock

- **A lock implemented to do 'busy-wait'**
- **Using instructions like T&S or Swap**
- **As shown on earlier slides**

```
spinlock(int *lock){
    While(test-and-set(lock))
        ;
}
spinunlock(lock *lock) {
    *lock = false;
}
```

Bounded wait M.E. with T&S

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

sleep-locks

- Spin locks result in busy-wait
- CPU cycles wasted by waiting processes/threads
- Solution – threads keep waiting for the lock to be available
  - Move thread to wait queue
  - The thread holding the lock will wake up one of them

Sleep locks/mutexes

```
//ignore syntactical issues
typedef struct mutex {
    int islocked;
    int spinlock;
    waitqueue q;
    }mutex;
wait(mutex *m) {
    spinlock(m->spinlock);
    while(m->islocked)
        Block(m, m->spinlock)
    lk->islocked = 1;
    spinunlock(m->spinlock);
}
```

```
Block(mutex *m, spinlock *sl) {
    currprocess->state = WAITING
    move current process to  m->q
    spinunlock(sl);
    Sched();
    spinlock(sl);
}
release(mutex *m) {
    spinlock(m->spinlock);
    m->islocked = 0;
    Some process in m->queue
    =RUNNABLE;
    spinunlock(m->spinlock);
    }
```

Some thumb-rules of spinlocks

- **Never block a process holding a spinlock !**
- **Typical code:**

```
while(condition)
    { Spin-unlock()
    Schedule()
    Spin-lock()
}
```

- **Hold a spin lock for only a short duration of time**
  - **Spinlocks are preferable on multiprocessor systems**
  - **Cost of context switch is a concern in case of sleep-wait locks**
  - **Short = < 2 context switches**

Locks in xv6 code

struct spinlock

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

spinlocks in xv6 code

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  struct buf head;
} bcache;
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock sl;
```

```
static struct spinlock idelock;
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
struct log {
  struct spinlock lock;
...}
struct pipe {
  struct spinlock lock;
...}
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
struct spinlock tickslock;
```

Spinlock in xv6

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;
   // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters) that locked the lock.
};
```

```
void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
 // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
//extra debugging code
}
void release(struct spinlock *lk)
{  //extra debugging code
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );
  popcli();
}
```

spinlocks

```
Void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");
    ......
void pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

- Pushcli() - disable interrupts on that processor
- One after another many acquire() can be called on different spinlocks
  - Keep a count of them in mycpu()->ncli

spinlocks

```
void
release(struct spinlock *lk)
{
...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
    popcli();
}
.
Void popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

- Popcli()
  - Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

spinlocks

- **Always disable interrupts while acquiring spinlock**
  - **Suppose iderw held the idelock and then got interrupted to run ideintr.**
  - **Ideintr would try to lock idelock, see it was held, and wait for it to be released.**
  - **In this situation, idelock will never be released**
  - **Deadlock**
- **General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled**
- **Xv6 rule: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.**

sleeplocks

- Sleeplocks don’t spin. They move a process to a wait-queue if the lock can’t be acquired
- XV6 approach to “wait-queues”
  - Any memory address serves as a “wait channel”
  - The sleep() and wakeup() functions just use that address as a ‘condition’
  - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
    - costly, but simple

sleep()

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
....
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
 p->state = SLEEPING;
  sched();
  // Reacquire original lock.
   if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
```

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

Calls to sleep() : examples of "chan" (output from cscope)

0 console.c
consoleread 251
sleep(&input.r, &cons.lock);

2 ide.c iderw
169 sleep(b, &idelock);

3 log.c begin_op
131 sleep(&log, &log.lock);

6 pipe.c piperead
111 sleep(&p->nread, &p->lock);

7 proc.c wait
317 sleep(curproc, &ptable.lock);

8 sleeplock.c
acquiresleep 28
sleep(lk, &lk->lk);

9 sysproc.c
sys_sleep 74
sleep(&ticks, &tickslock);

Wakeup()

```
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
static void wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING &&
p->chan == chan)
      p->state = RUNNABLE;
}
```

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

sleeplock

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock sl; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

Sleeplock acquire and release

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    /* Abhijit: interrupts are not disabled in sleep !*/
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Where are sleeplocks used?

- struct buf
  - waiting for I/O on this buffer
- struct inode
  - waiting for I/o to this inode
- Just two !

Sleeplocks issues

- sleep-locks support yielding the processor during their critical sections.
- This property poses a design challenge:
  - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
  - and thread T2 wishes to acquire L1,
  - we have to ensure that T1 can execute
  - while T2 is waiting so that T1 can release L1.
  - T2 can’t use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.
- To avoid this deadlock, the sleep-lock acquire routine (called acquiresleep) yields the processor while waiting, and does not disable interrupts.

Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.

More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated

- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
        while S <= 0
             ; // no-op
          S--;
    }
signal (S) {
      S++;
   }
```

--> Note this is Signal() on a semaphore, different froms signal system call

Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

    Also known as **mutex locks**

- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
wait (mutex);
        // Critical Section
    signal (mutex);
// remainder section
} while (TRUE)
```

Different uses of semaphores

For mutual exclusion

```
/*During inialization*/
semaphore sem;
initsem (&sem, 1);

/* On each use*/
P (&sem);
Use resource;
V (&sem);
```

Event-wait

```
/* During initialization */
semaphore event;
initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */
P (&event); /* Blocks if event has not occurred */
/* Event has occurred */
V (&event); /* So that another thread may wake up */
/* Continue processing */

/* Code executed by another thread when event occurs */
V (&event); /* Wake up one thread */
```

Control countable resources

```
/* During initialization */
semaphore counter;
initsem (&counter, resourceCount);
/* Code executed to use the resource */
P (&counter); /* Blocks until resource is available */
Use resource; /* Guaranteed to be available now */
V (&counter); /* Release the resource */
```

Semaphore implementation

```
Wait(sem *s) {
    while(s <=0)
        block(); // could be ";"
    s--;
}
signal(sem *s) {
    s++;
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the sematics of the semaphore.

Semaphore implementation? - 1

```
struct semaphore {
    int val;
    spinlock sl;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}
```

```
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

- suppose 2 processes trying wait.

val = 1;

Th1: spinlock Th2: spinlock-waits

Th1: while -> false, val-- => 0; spinulock;

Th2: spinlock success; while() -> true, loops;

Th1: is done with critical section, it calls signal. it calls spinlock() -> wait.

Who is holding spinlock-> Th2. Itis waiting for val > 0. Who can set value > 0 , ans: Th1, and Th1 is waiting for spinlock which is held by The2.

circular wait. Deadlock.

None of them will proceed.

Semaphore implementation? - 2

```
struct semaphore {
    int val;
    spinlock sl;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

Problem: race in spinlock of whille loop and signal's spinlock. Bounded wait not guaranteed.

Spinlocks are not good for a long wait.

Semaphore implementation? - 3, idea

```
struct semaphore {
    int val;
    spinlock sl;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block() {
    put this current process on wait-q;
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        Block();
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

Semaphore implementation? - 3a

```
struct semaphore {
    int val;
    spinlock sl;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}
```

problem is that block() will be called without holding the spinlock and the access to the list is not protected.

Note that - so far we have ignored changes to signal()

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

Semaphore implementation? - 3b

```
struct semaphore {
    int val;
    spinlock sl;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}
```

Problem: after a blocked process comes out of the block, it does not hold the spinlock and it's goinng to change the s->sl;

Semaphore implementation? - 3c

```
struct semaphore {
    int val;
    spinlock sl;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl)); // A
    while(s->val <=0) {
        block(s);
        spinlock(&(s->sl)); // B
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}
```

Question: there is race between A and B. Can we guarantee bounded wait ?

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore in Linux

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};

static noinline void __sched
__down(struct semaphore *sem)
{
    __down_common(sem,
TASK_UNINTERRUPTIBLE,
MAX_SCHEDULE_TIMEOUT);
}
```

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock,
flags);
}
```

Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem-
>wait_list);
    waiter.task = task;
    waiter.up = false;

    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
    ....
}
```

Drawbacks of semaphores

- **Need to be implemented using lower level primitives like spinlocks**
- **Context-switch is involved in blocking and signaling – time consuming**
- **Can not be used for a short critical section**

Deadlocks

Deadlock

- **two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes**
- **Let S and Q be two semaphores initialized to 1**

P0	**P1**
wait (S);	**wait (Q);**
wait (Q);	**wait (S);**
. .	
. .	
. .	
signal (S);	**signal (Q);**
signal (Q);	**signal (S);**

Example of deadlock

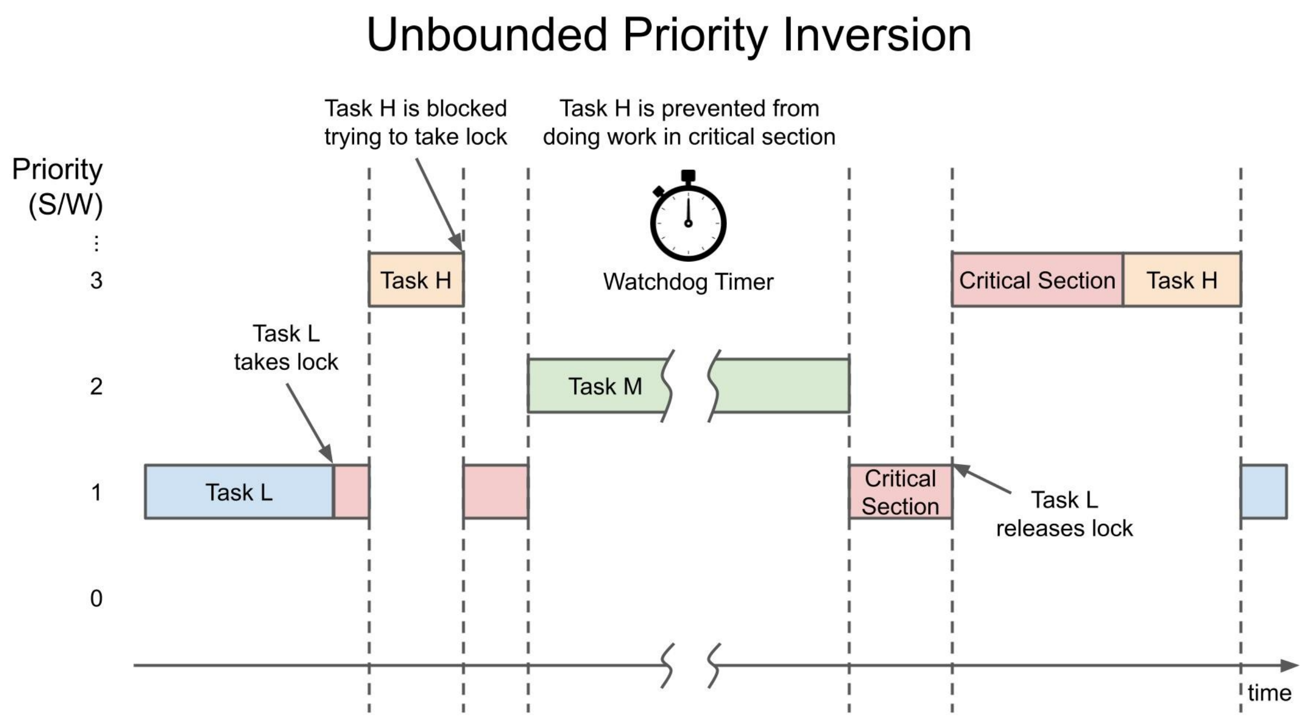
- Let’s see the pthreads program : deadlock.c
- Same programe as on earlier slide, but with pthread_mutex_lock();

Non-deadlock, but similar situations

- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Non-deadlock, but similar situations

- **Priority Inversion**
  - **Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task**
  - **Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest**



Livelock

- Similar to deadlock, but processes keep doing ‘useless work’
- E.g. two people meet in a corridor opposite each other
  - Both move to left at same time
  - Then both move to right at same time
  - Keep Repeating!
- No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing

Livelock example

```
#include <stdio.h>
#include <pthread.h>
struct person {
   int otherid;
   int otherHungry;
   int myid;
};
int main() {
   pthread_t th1, th2;
   struct person one, two;
   one.otherid = 2; one.myid = 1;
   two.otherid = 1; two.myid = 2;
   one.otherHungry = two.otherHungry = 1;
   pthread_create(&th1, NULL, eat, &one);
   pthread_create(&th2, NULL, eat, &two);
   printf("Main: Waiting for threads to get over\n");
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   return 0;
}
```

```
/* thread two runs in this function */
int spoonWith = 1;
void *eat(void *param)
{
  int eaten = 0;
  struct person person= *(struct person *)param;
  while (!eaten) {
      if(spoonWith == person.myid)
         printf("%d going to eat\n", person.myid);
      else
         continue;
      if(person.otherHungry) {
         printf("You eat %d\n", person.otherid);
         spoonWith = person.otherid;
         continue;
      }
      printf("%d is eating\n", person.myid);
      break;
  }
}
```

More on deadlocks

- Under which conditions they can occur?
- How can deadlocks be avoided/prevented?
- How can a system recover if there is a deadlock ?

System model for understanding deadlocks

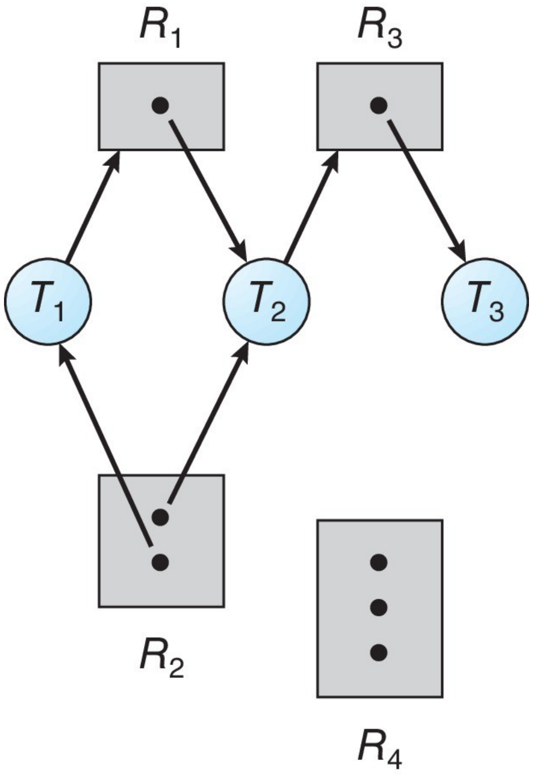
- System consists of resources
- Resource types R1, R2, . . ., Rm
  - CPU cycles, memory space, I/O devices
  - Resource: Most typically a lock, synchronization primitive
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock characterisation

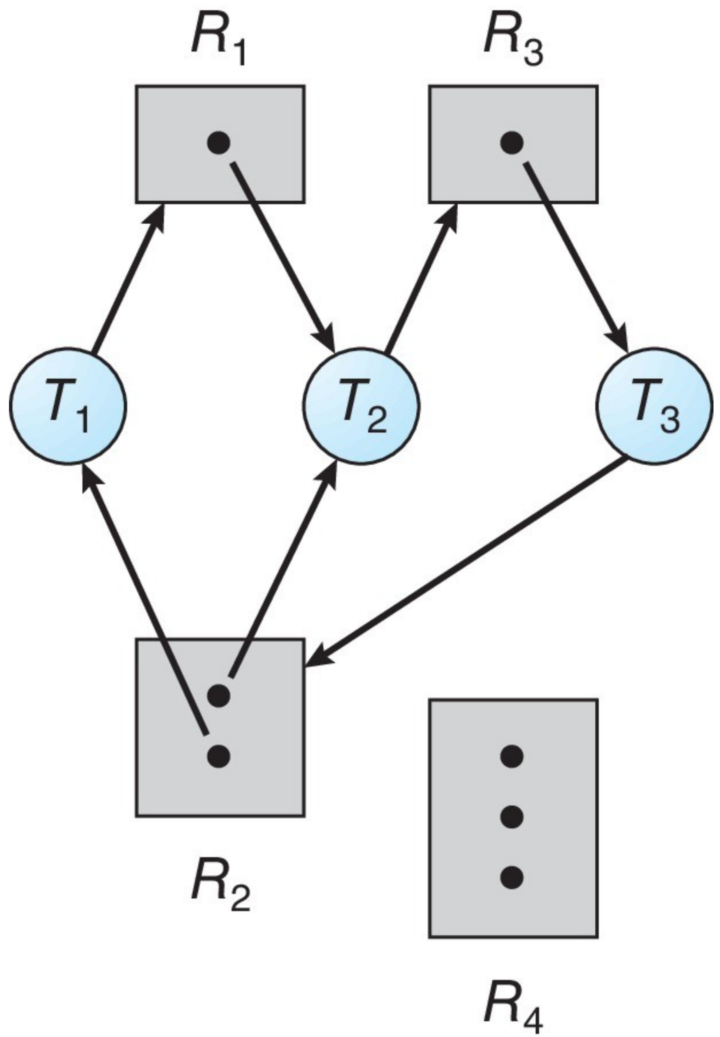
- Deadlock is possible only if ALL of these conditions are TRUE at the same time
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource Allocation Graph Example

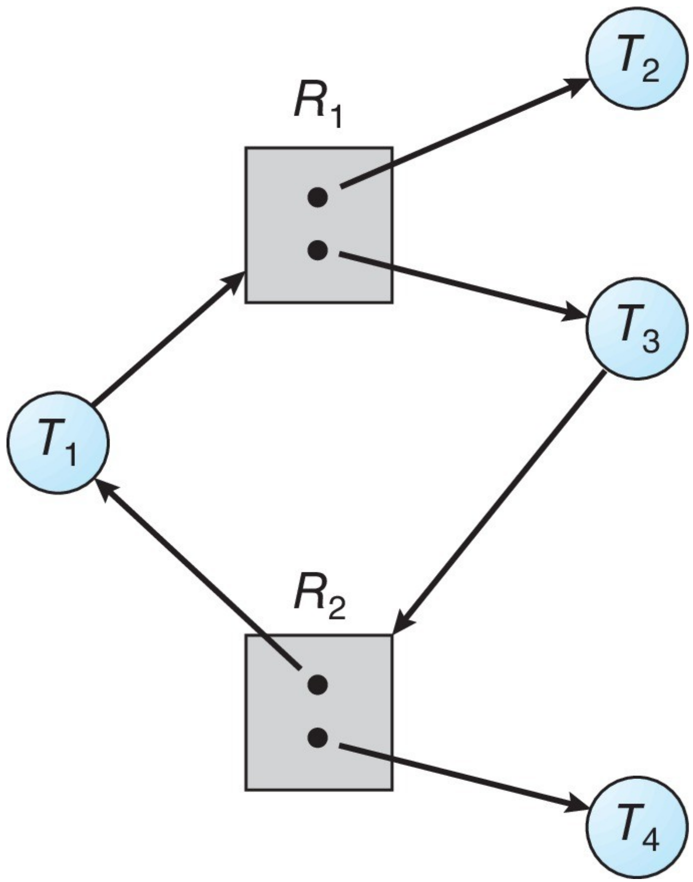
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Basic Facts

- If graph contains no cycles -> no deadlock
- If graph contains a cycle :
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- **Ensure that the system will never enter a deadlock state:**

  1) Deadlock prevention

  2) Deadlock avoidance

  3) Allow the system to enter a deadlock state and then recover

  4) Ignore the problem and pretend that deadlocks never occur in the system.

(1) Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

(1) Deadlock Prevention (Cont.)

- **No Preemption:**
  - **If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released**
  - **Preempted resources are added to the list of resources for which the process is waiting**
  - **Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting**
- **Circular Wait:**
  - **Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration**

(1) Deadlock prevention: Circular Wait

- **Invalidating the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number.**
- **Resources must be acquired in order.**
- **If:**

  first_mutex is mapped to order 1

  second_mutex is mapped to order 5

  code for thread_two could not be written like on RHS

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

(1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
  - **Decide an ordering among all ‘locks’**
  - **Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!**
  - **Poses coding challenges!**
  - **A key differentiating factor in kernels**
  - **Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!**

(1) Prevention in Xv6: Lock Ordering

- **lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.**

(2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
  - **Processes declare resources the want, BEFORE-hand**
  - **Resources are always allocated by an ALLOCATOR algorithm**
    - **It can predict if a deadlock can happen**

(2) Deadlock avoidance

- Please see: concept of safe states, unsafe states, Banker's algorithm

(3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.
- How to recover?
  - Abort all processes or abort one by one?
  - Which processes to abort?
    - Priority ?
    - Time spent since forked()?
    - Resources used?
    - Resources needed?
    - Interactive or not?
    - How many need to be terminated?

“Condition”
Synchronization Tool

What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

Proc *next

Proc *prev

Spinlock *lock

}

Different variables of this type can be used as different ‘conditions

Code for condition variables

```
//Spinlock s is held before calling wait
void wait (condition *c, spinlock_t *s)
(
   spin_lock (&c->listLock);
   add self to the linked list;
   spin_unlock (&c->listLock);
   spin_unlock (s); /* release
   spinlock before blocking */
    swtch(); /* perform context switch */
   /* When we return from swtch, the
event has occurred */
   spin_lock (s); /* acquire the spin
lock again */
   return;
)
```

```
void do_signal (condition *c)
/*Wakeup one thread waiting on the  condition*/
{
   spin_lock (&c->listLock);
   remove one thread from linked list, if it is nonempty;
   spin_unlock (&c->listLock);
   if a thread was removed from the list, make it
     runnable;
   return;
}
void do broadcast (condition *c)
/*Wakeup al lthreads waiting on the condition*/
{
  spin_lock (&c->listLock);
  while (linked list is nonempty) {
    remove a thread from linked list;
    make it runnable;
  }
  spin_unlock (&c->listLock);
}
```

Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
  //something
  condition c;
}semaphore;
```

- Now write code for semaphore P() and V()

Classical Synchronization Problems

Bounded-Buffer Problem

- Producer and consumer processes
  - N buffers, each can hold one item
- Producer produces ‘items’ to be consumed by consumer , in the bounded buffer
- Consumer should wait if there are no items
- Producer should wait if the ‘bounded buffer’ is full

Bounded-Buffer Problem: solution with semaphores

- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Bounded-buffer problem

The structure of the producer process

```
do  {
       //   produce an item in nextp
      wait (empty);
      wait (mutex);
      //  add the item to the  buffer
       signal (mutex);
       signal (full);
} while (TRUE);
```

The structure of the Consumer process

```
do  {
      wait (full);
      wait (mutex);
      //  remove an item from
        //  buffer to nextc
       signal (mutex);
       signal (empty);
      // consume item in nextc
} while (TRUE);
```

Bounded buffer problem

- **Example : pipe()**
- **Let’s see code of pipe in xv6 – a solution using sleeplocks**

Readers-Writers problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers problem

The structure of a writer process

```
do {
    wait (wrt) ;
//    writing is performed
    signal (wrt) ;
} while (TRUE);
```

The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)
       // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

Readers-Writers Problem Variations

- **First variation – no reader kept waiting unless writer has permission to use shared object**
- **Second variation – once writer is ready, it performs write asap**
- **Both may have starvation leading to even more variations**
- **Problem is solved on some systems by kernel providing reader-writer locks**

Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared
  - Upgrade() -> from shared to excl

Code for reader-writer locks

```
struct rwlock {
   int nActive; /* num of active
readers, or-1 if a writer is
active */
   int nPendi ngReads;
   int nPendingWrites;
   spinlock_t sl;
   condition canRead;
   condition canWrite;
);
```

```
void lockShared {struct rwlock *r)
(
  spin_lock {&r->sl);
  r->nPendingReads++;
  if (r->nPendingWrites > O)
    wait (&r->canRead, &r->sl ); /*don'tstarve
      writers */
  while {r->nActive < 0) /* someone has
      exclusive lock */
      wait (&r->canRead, &r->sl);
  r->nActive++;
  r->nPendingReads--;
  spin_unlock (&r->sl);
)
```

Code for reader-writer locks

```
void unlockShared (struct rwlock
*r)
{
  spin_lock (&r->sl);
  r->nActive--;
  if (r->nActive == O) {
    spin_unlock (&r->sl);
    do signal (&r->canWrite);
  } else
    spin_unlock (&r->M);
)
```

```
void lockExclusive (struct rwlock
*r)
(
  spin_lock (&r->sl);
  r->nPendingWrltes++;
  while (r->nActive)
    wait (&r->canWrite, &r->sl);
  r->nPendingWrites--;
  r->nActive = -1;
  spin_unlock (&r->sl);
}
```

Code for reader-writer locks

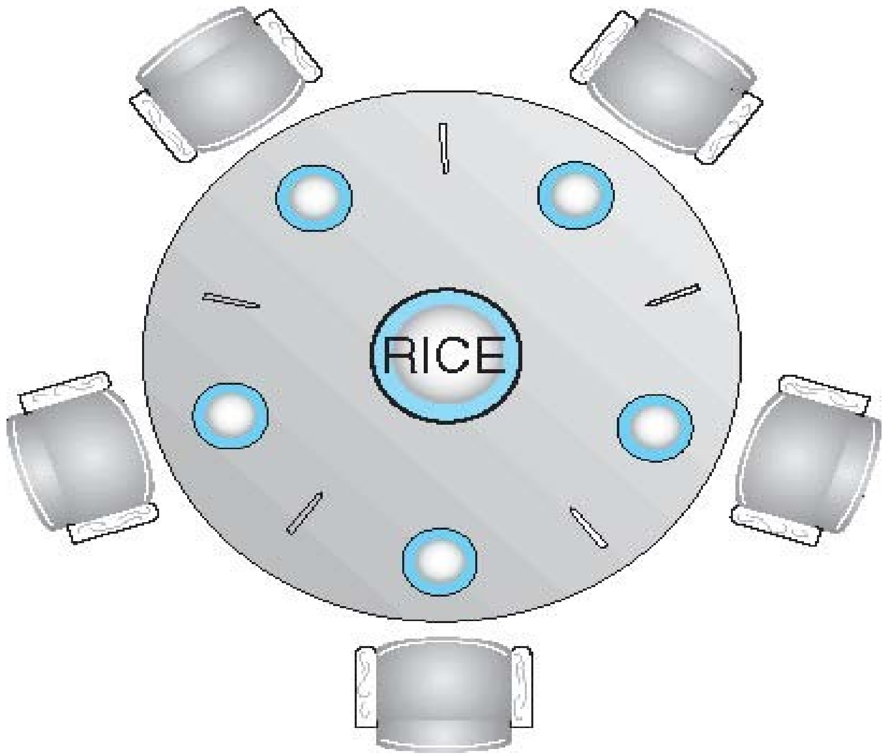
```
void unlockExclusive (struct rwlock *r){
   boolean t wakeReaders;
   spin_lock (&r->sl);
   r->nActive = O;
   wakeReaders = (r->nPendingReads != 0);
   spin_unlock (&r->sl);
   if (wakeReaders)
        do broadcast (&r->canRead); /* wake
allreaders */
  else
      do_signal (&r->canWrite);
       /*wakeasinglewri  r */
}
```

Try writing code for downgrade and upgrade

Try writing a reader-writer lock using semaphores!

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining philosophers: One solution

The structure of Philosopher i:

```
do  {
     wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );
    //  eat
    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
    //  think
} while (TRUE);
```

What is the problem with this algorithm?

Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section
- Use an asymmetric solution
  - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

Other solutions to dining philosopher’s problem

- Using higher level synchronization primitives like ‘monitors’
-

Practical Problems

Lost Wakeup problem

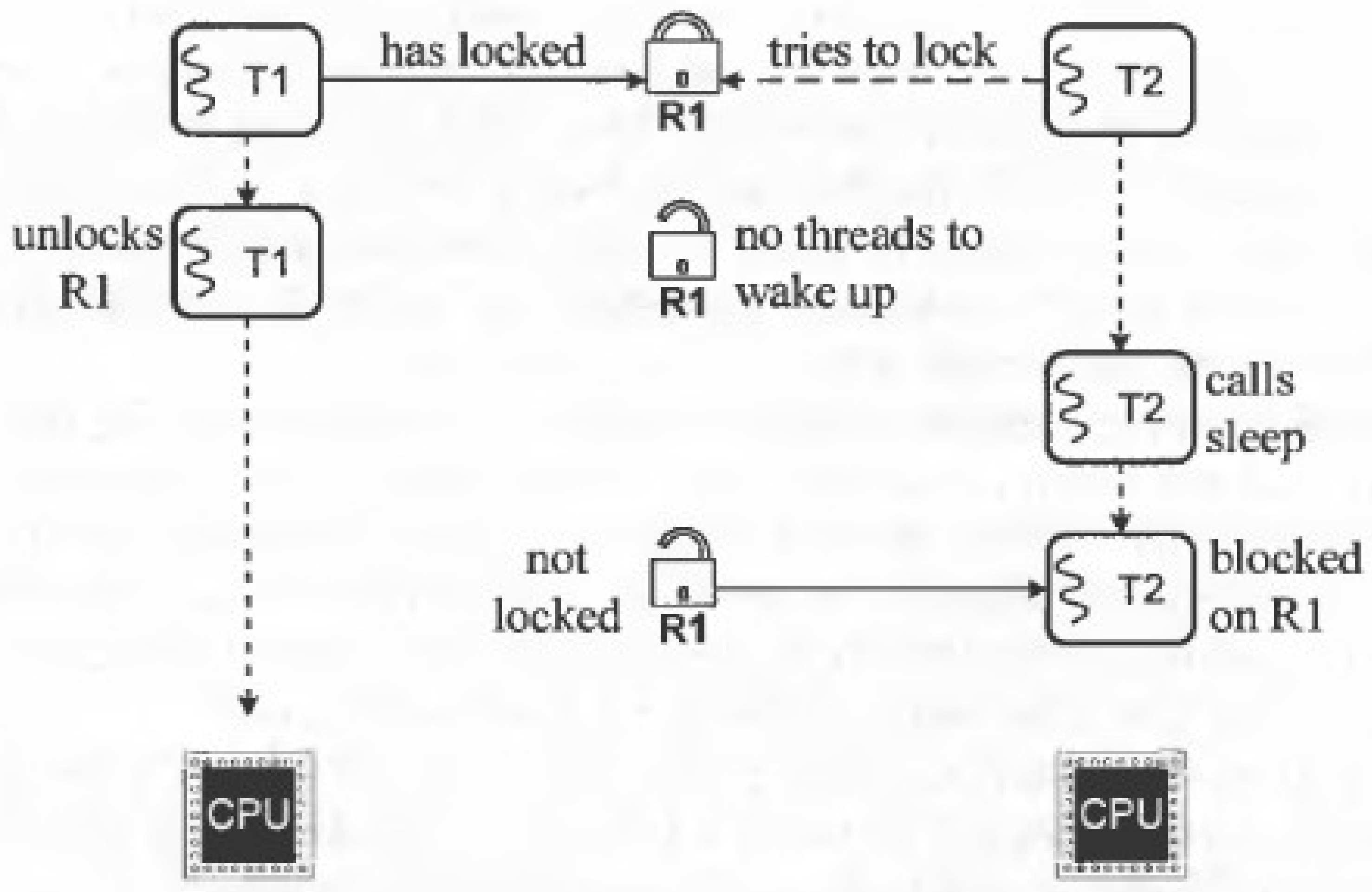


Figure 7-6. The lost wakeup problem.

Requires some mechanism to combine the test for the resource and the call to sleep () into a single atomic operation.

Lost Wakeup problem

- The sleep/wakeup mechanism does not function correctly on a multiprocessor.
- Consider a potential race:
  - Thread T1 has locked a resource R1.
  - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
  - T2 calls sleep() to wait for the resource.
  - Between the time T2 finds the resource locked and the time it calls s]eep (), T1 frees the resource and proceeds to wake up all threads blocked on it.
  - Since T2 has not yet been put on the sleep queue, it will miss the wakeup.
  - The end result is that the resource is not locked, but T2 is blocked waiting for it to be unlocked.
  - If no one else tries to access the resource, T2 could block indefinitely.
  - This is known as the lost wakeup problem,
- Requires some mechanism to combine the test for the resource and the call to sleep () into a single atomic operation.

Thundering herd problem

- **Thundering Herd problem**
  - **On a multiprocessor, if several threads were locked the resource**
  - **Waking them all may cause them to be simultaneously scheduled on different processors**
  - **and they would all fight for the same resource again.**
- **Starvation**
  - **Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.**
  - **In this interval, an unrelated thread may grab the resource causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.**
  - **This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it.**

Case Studies

Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
  - Atomic integers
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- **Atomic variables**

  atomic_t is the type for atomic integer

- **Consider the variables**

  atomic_t counter;

  int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Synchronization issues in xv6 kernel

Difference approaches

- Pros and Cons of locks
  - Locks ensure serialization
  - Locks consume time !
- Solution – 1
  - One big kernel lock
  - Too enefficient
- Solution – 2
  - One lock per variable
  - Often un-necessary, many data structures get manipulated in once place, one lock for all of them may work
- Problem: ptable.lock for the entire array and every element within
  - Alternatively: one lock for array, one lock per array entry

Three types of code

- System calls code
  - Can it be interruptible?
  - If yes, when?
- Interrupt handler code
  - Disable interrupts during interrupt handling or not?
  - Deadlock with iderw ! - already seen
- Process's user code
  - Ignore. Not concerned with it now.

Interrupts enabling/disablilng in xv6

- Holding every spinlock disables interrupts!
- System call code or Interrupt handler code won’t be interrupted if
  - The code path followed took at least once spinlock !
  - Interrupts disabled only on that processor!
- Acquire calls pushcli() before xchg()
- Release calls popcli() after xchg()

Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- This often improves performance
- Compiler may reorder 4 after 6 --> Troble!
- Solution: Memory barrier
  - __sync_synchronize(), provided by GCC
  - Do not reorder across this line
  - Done only on acquire and release()

- Consider this

```
1)l = malloc(sizeof *l);
2)l->data = data;
3)acquire(&listlock);
4)l->next = list;
5)list = l;
6)release(&listlock);
```

Lost Wakeup?

- Do we have this problem in xv6?
- Let's analyze again!
  - The race in acquiresleep()'s call to sleep() and releasesleep()
- T1 holding lock, T2 willing to acquire lock
  - Both running on different processor
  - Or both running on same processor
  - What happens in both scenarios?
- Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?
- See page 69 in xv6 book revision-11.

Code of sleep()

```
if(lk != &ptable.lock){
   acquire(&ptable.lock);
   release(lk);
}
```

- Why this check?
- Deadlock otherwise!
- Check: wait() calls with ptable.lock held!

Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){
acquire(&ptable.lock);
release(lk);
}
```

with

```
release(lk);
acquire(&ptable.lock);
```

Doing this would break sleep. How?

`

bget() problem

- bget() panics if no free buffers!
- Quite bad
- Should sleep !
- But that will introduce many deadlock problems. Which ones ?

iget() and ilock()

- iget() does no hold lock on inode
- Ilock() does
- Why this separation?
  - Performance? If you want only “read” the inode, then why lock it?
- What if iget() returned the inode locked?

Interesting cases in namex()

```
  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip
  }
```

--> only after obtaining next from dirlookup() and iget() is the lock released on ip;

-> lock on next obtained only after releasing the lock on ip. Deadlock possible if next was “.”

Xv6
Interesting case of holding and releasing ptable.lock in scheduling

One process acquires, another releases!

Giving up CPU

- A process that wants to give up the CPU
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- Yield follows this convention, as do sleep and exit
- Lock held by one process P1, will be released another process P2 that starts running after sched()
  - remember P2 returns either in yield() or sleep()
  - In both, the first thing done is releasing ptable.lock

Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
  - Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in yield taking ptable.lock prevents this

Homework

- Read the version-11 textbook of xv6
- Solve the exercises!