

Virtual Memory

Introduction

- Virtual memory != Virtual address

Virtual address is address issued by CPU's execution unit,
later converted by MMU to physical address

Virtual memory is a memory management technique
employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
        f(a);
}
```

All parts of array a[] not accessed

Function f() may not be called

Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used

Error code, unusual routines, large data structures are rarely used

- So, entire program code, data not needed at same time

- So, consider ability to execute partially-loaded program

One Program no longer constrained by limits of physical memory

One Program and collection of programs could be larger than physical memory

.

What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory

Only part of the program needs to be in memory for execution

Logical address space can therefore be much larger than physical address space

Allows address spaces to be shared by several processes

Allows for more efficient process creation

More programs running concurrently

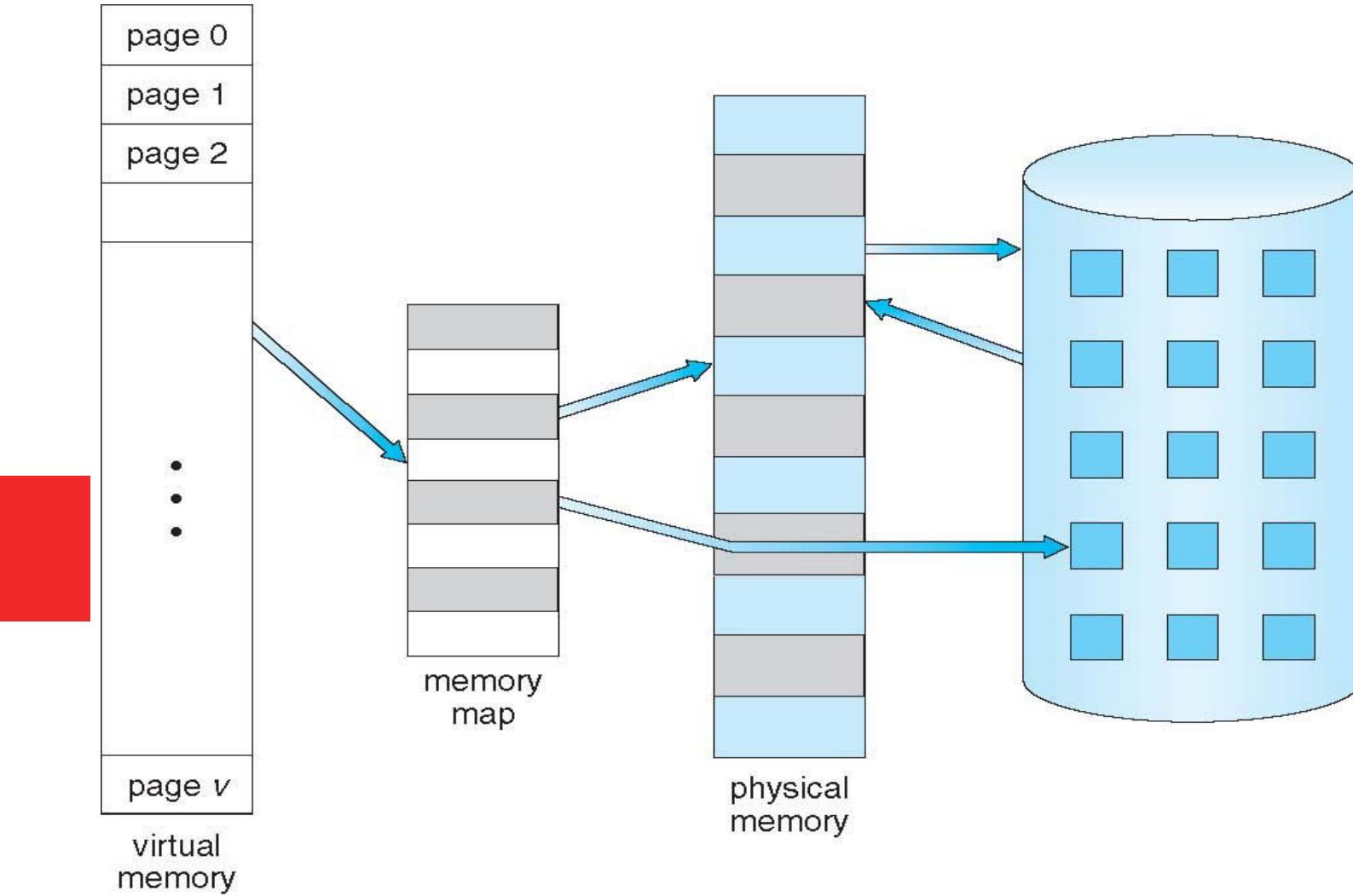
Less I/O needed to load or swap processes

- Virtual memory can be implemented via:

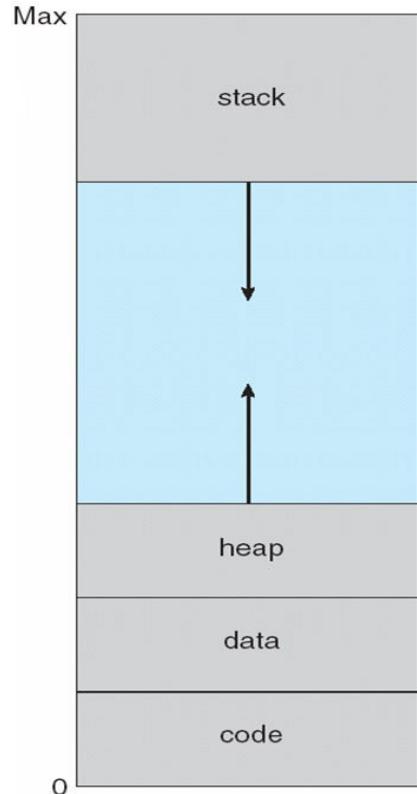
Demand paging

Demand segmentation

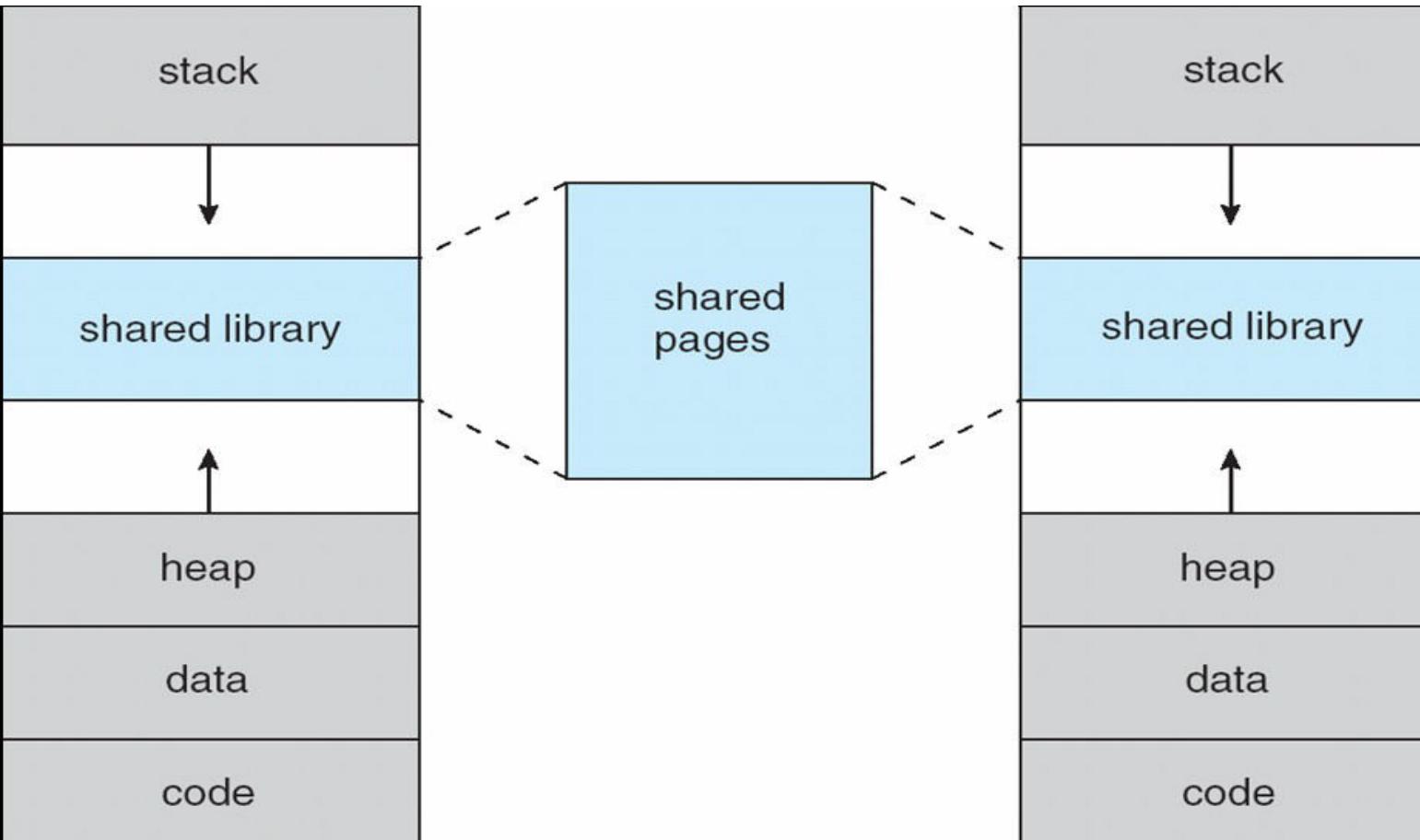
Virtual Memory
Larger
Than
Physical
Memory



Virtual Address space



Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc



Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)



Demand Paging

Demand Paging

- Load a “page” to memory when it’s needed (on demand)

Less I/O needed, no unnecessary I/O

Less memory needed

Faster response

More users

-

Demand Paging

- Options:

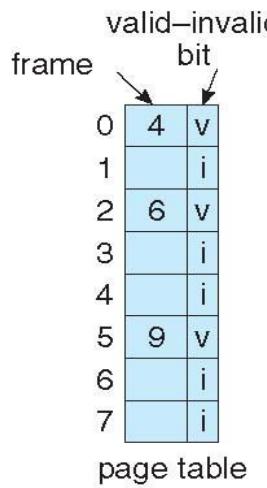
- Load entire process at load time : achieves little

- Load some pages at load time: good

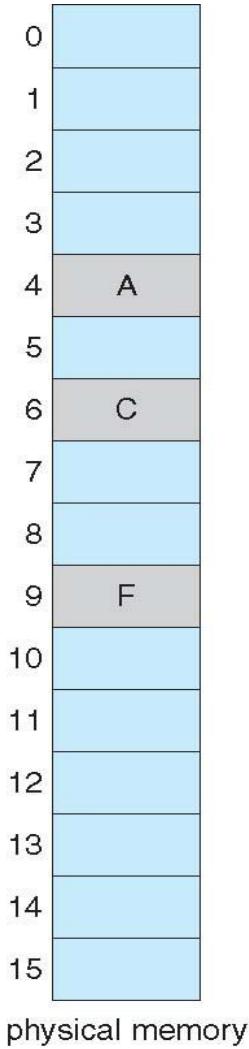
- Load no pages at load time: pure demand paging

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

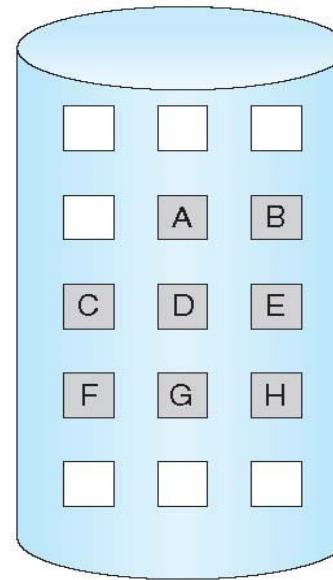
logical memory



page table



physical memory



**Page
Table
With
Some pages
Not
In memory**



Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

Change of stack to kernel stack

Saving the context of process

Switching to kernel code

-

On a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

If it's Invalid reference -> abort the process (segfault)

Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated!)

3) Swap page into frame via scheduled disk/IO operation

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

6) Restart the instruction that caused the page fault

Additional problems

- Extreme case – start process with *no* pages in memory
OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

And for every other process pages on first access

Pure demand paging

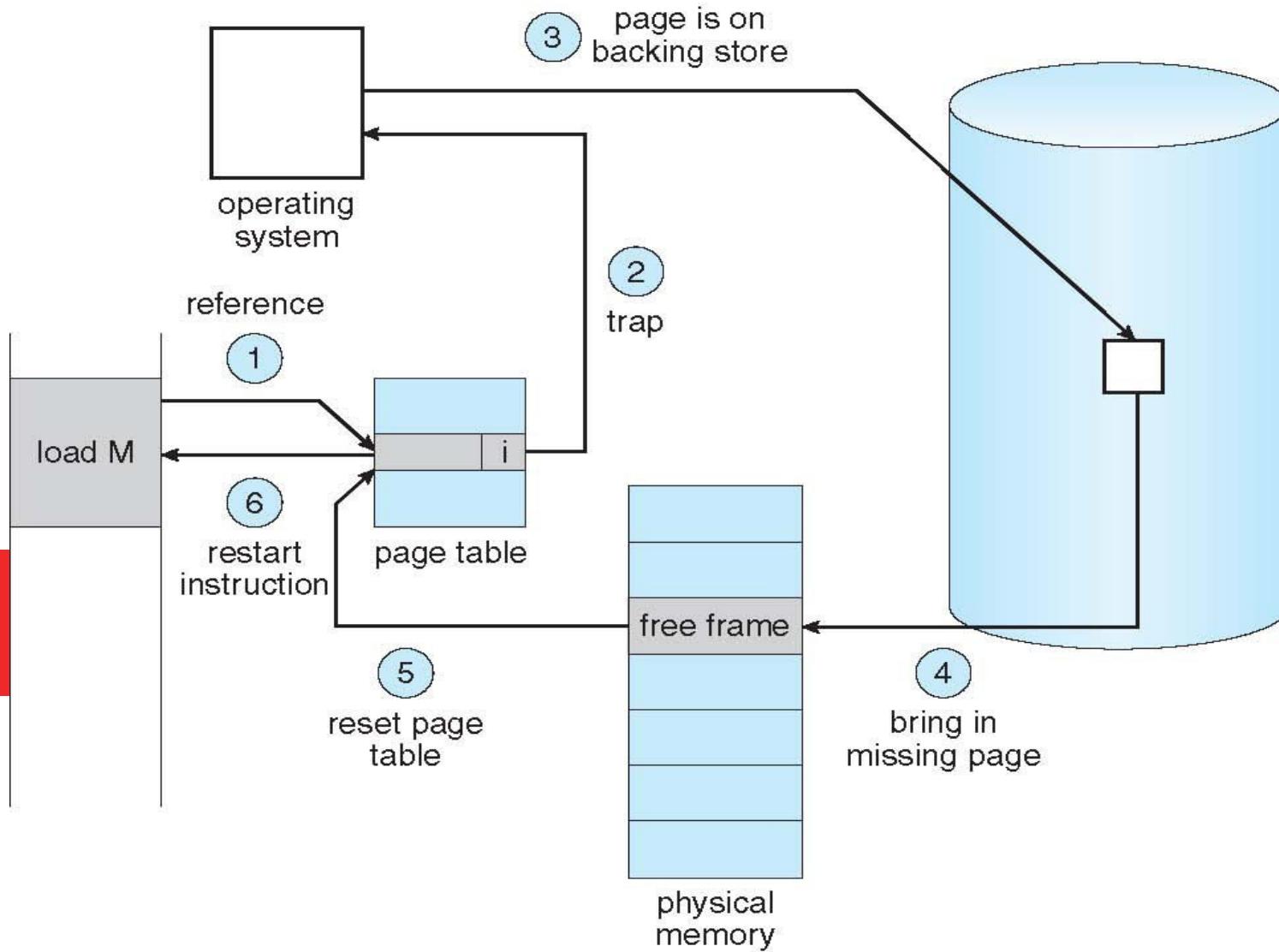
- Actually, a given instruction could access multiple pages -> multiple page faults
Pain decreased because of **locality of reference**

- Hardware support needed for demand paging
Page table with valid / invalid bit

Secondary memory (swap device with **swap space**)

Instruction restart

Handling A Page Fault



Page fault handling

1)Trap to the operating system

2)Default trap handling():

Save the process registers and process state

Determine that the interrupt was a page fault. Run page fault handler.

3)Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.

4)Find a free frame. Issue a read from the disk to a free frame:

Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

6)(as said on last slide) While waiting, allocate the CPU to some other process

7)(as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)

8)Default interrupt handling():

Save the registers and process state for the other user

Determine that the interrupt was from the disk

9)Disk interrupt handler():

Figure out that the interrupt was for our waiting process

Make the process runnable

10)Wait for the CPU to be allocated to this process again

Kernel restores the page table of the process, marks entry as "v"

Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate $0 \leq p \leq 1$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective (memory) Access Time (EAT)

$$EAT = (1 - p) * \text{memory access time} +$$

$p * (\text{page fault overhead} // \text{Kernel code execution time}$

$+ \text{swap page out} // \text{time to write an occupied frame to disk}$

$+ \text{swap page in} // \text{time to read data from disk into free frame}$

$+ \text{restart overhead}) // \text{time to reset process context, restart it}$

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

$$EAT = (1 - p) \times 200 + p \text{ (8 milliseconds)}$$

$$= (1 - p \times 200 + p \times 8,000,000)$$

$$= 200 + p \times 7,999,800$$

If one access out of 1,000 causes a page fault, then

$$EAT = 8.2 \text{ microseconds.}$$

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

$$220 > 200 + 7,999,800 \times p$$

$$20 > 7,999,800 \times p$$

$$p < .0000025$$

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

The problem with fork() and exec(). Consider the case of a shell

```
scanf("%s", cmd);

if(strcmp(cmd, "exit") == 0)

    return 0;

pid = fork(); // A->B

if(pid == 0) {

    ret = execl(cmd, cmd, NULL);

    if(ret == -1) {

        perror("execution failed");

        exit(errno);

    }

} else {

    wait(0);

}
```

- During fork()
- Pages of parent were duplicated
- Equal amount of page frames were allocated
- Page table for child differed from parent
- (as it has another set of frames)
- In exec()
- The page frames of child were taken away and new frames were allocated
- Child's page table was rebuilt!
- Waste of time during fork() if the exec() was to be called immediately

An optimization: Copy on write

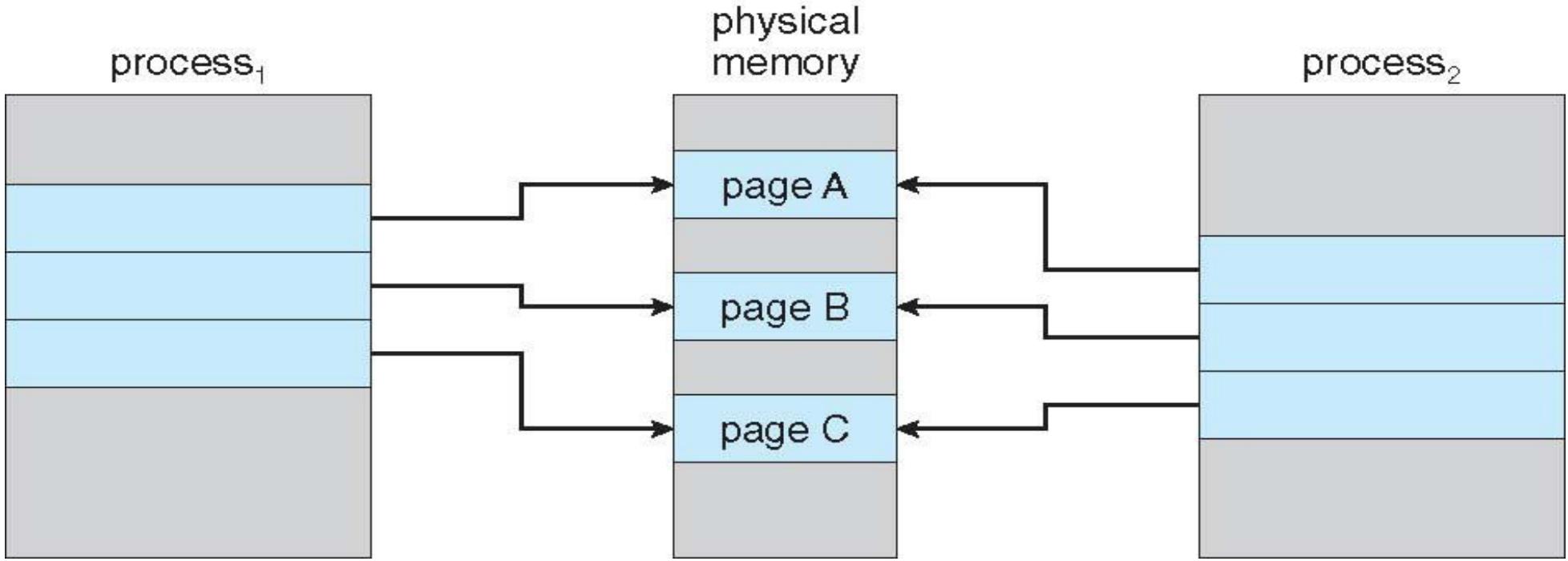
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory

If either process modifies a shared page, only then is the page copied

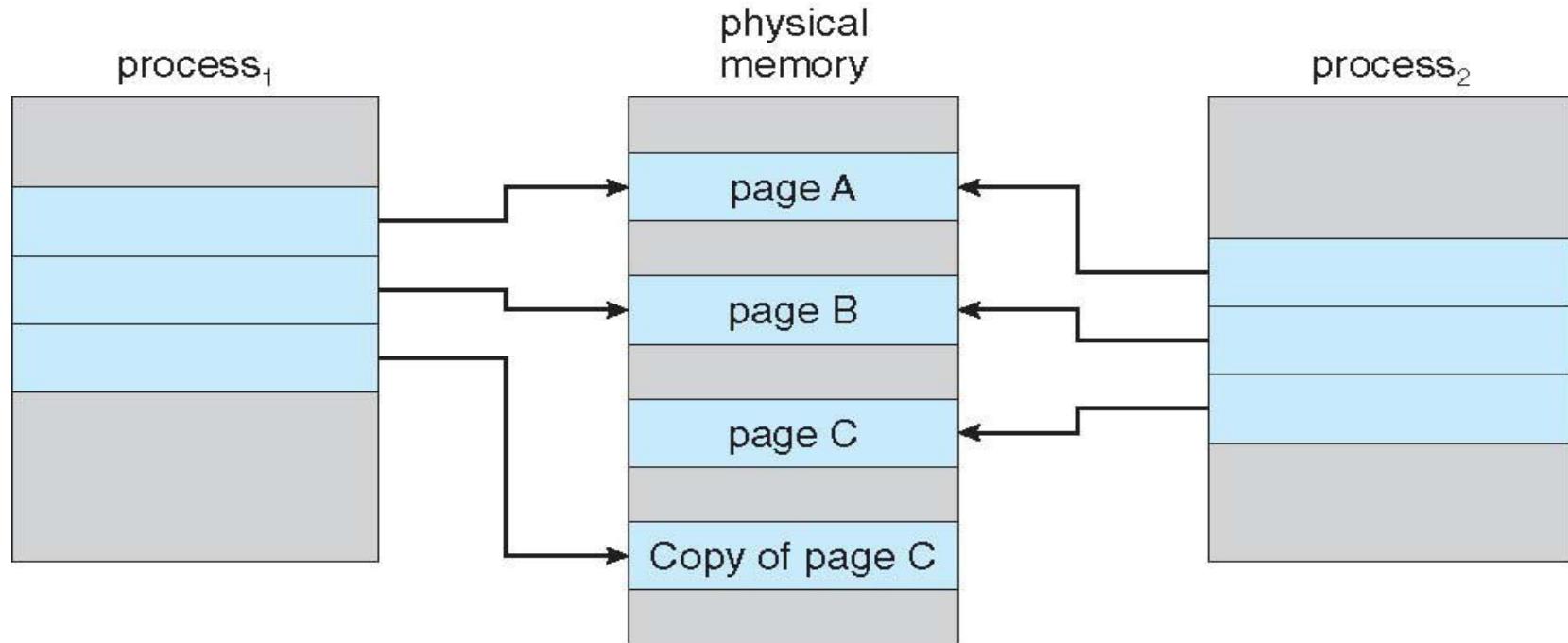
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent

Designed to have child call exec()

Very efficient



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store

For stack, heap pages: on swap partition

For code, shared library? : swap partition or the actual executable file on the file-system?

If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system

- Is the page table itself pagable?

If no, good

If Yes, then there can be page faults in accessing the page tables themselves! More complicated!

- Is the kernel code pageable?

If no, good

If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.



Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created
- Frames are used for allocating memory to a process
But may also be used for managing kernel's own data structures also
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”

More multiprogramming -> overallocation of frames

Also in demand from the kernel, I/O buffers, etc

How much to allocate to each process? How many processes to allow?

- Page replacement – find some page(frame) in memory, but not really in use, page it out

Questions : terminate process? Page out process? replace the page?

For performance, need an algorithm which will result in minimum number of page faults

- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

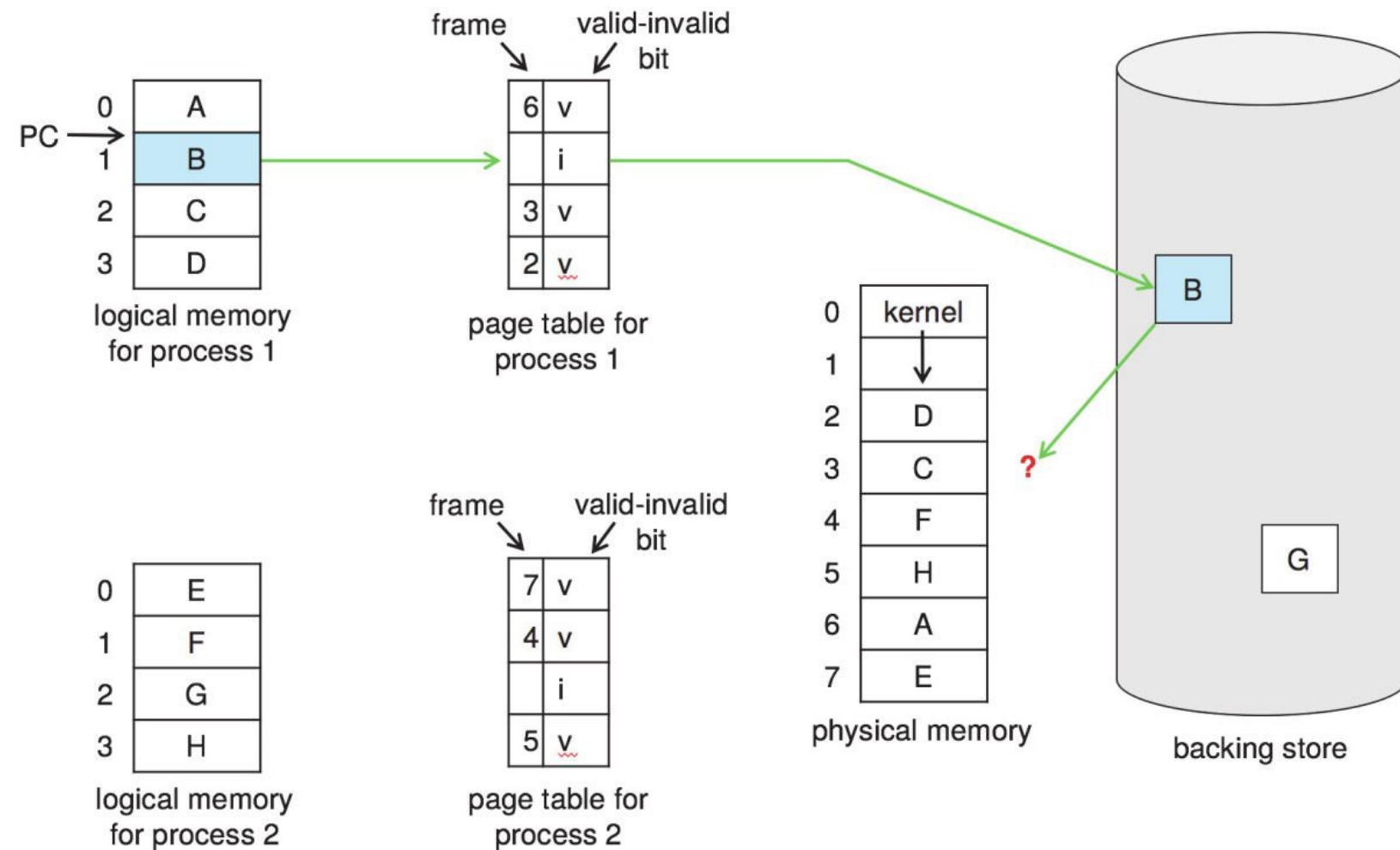


Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance

Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

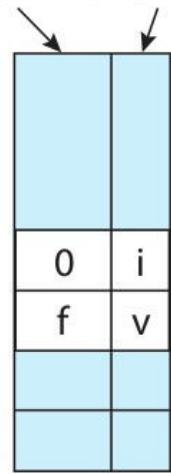
Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

- 1)Find the location of the desired page on disk
- 2)Find a free frame:
- 3) - If there is a free frame, use it
- 4) - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
- 5)Bring the desired page into the free frame; update the page table of process and global frame table/list
- 6)Continue the process by restarting the instruction that caused the trap
 - Note now potentially 2 page transfers for page fault – increasing EAT

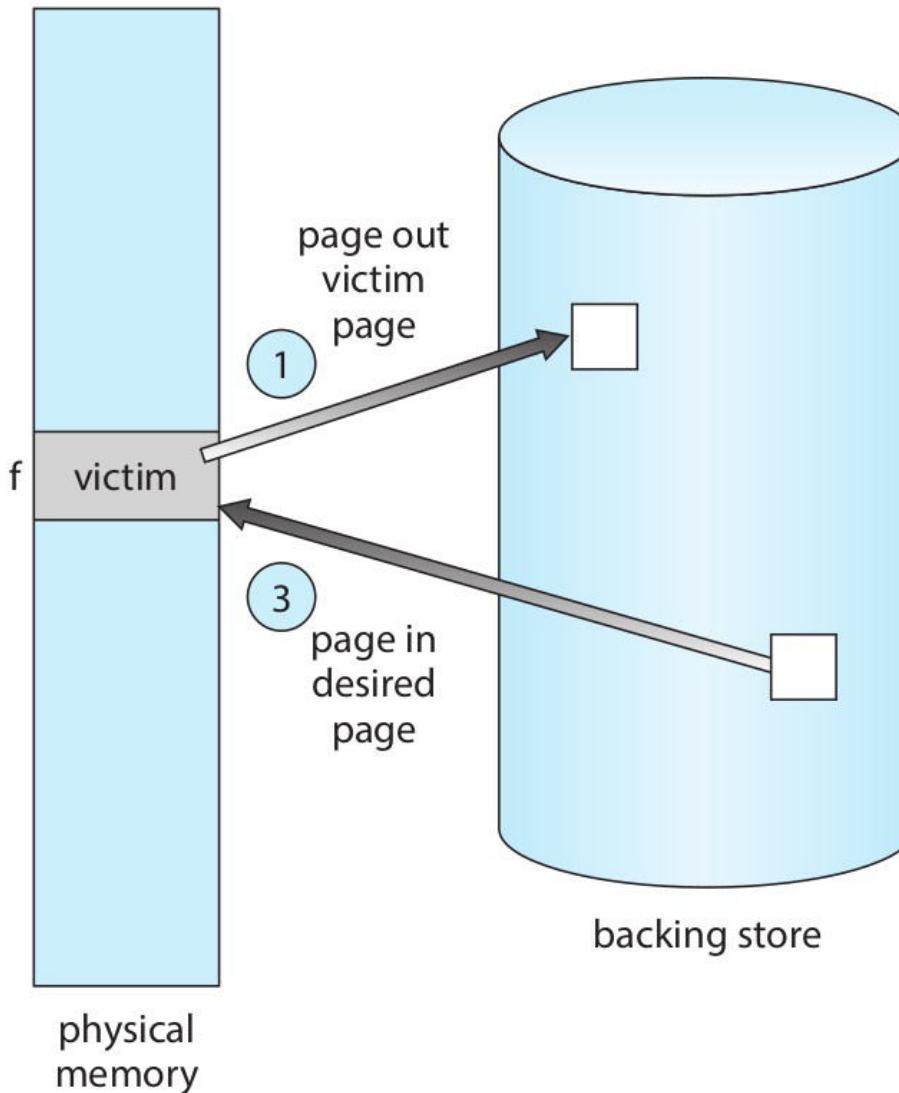
Page Replacement

frame valid-invalid bit



2 change to invalid

4 reset page table for new page

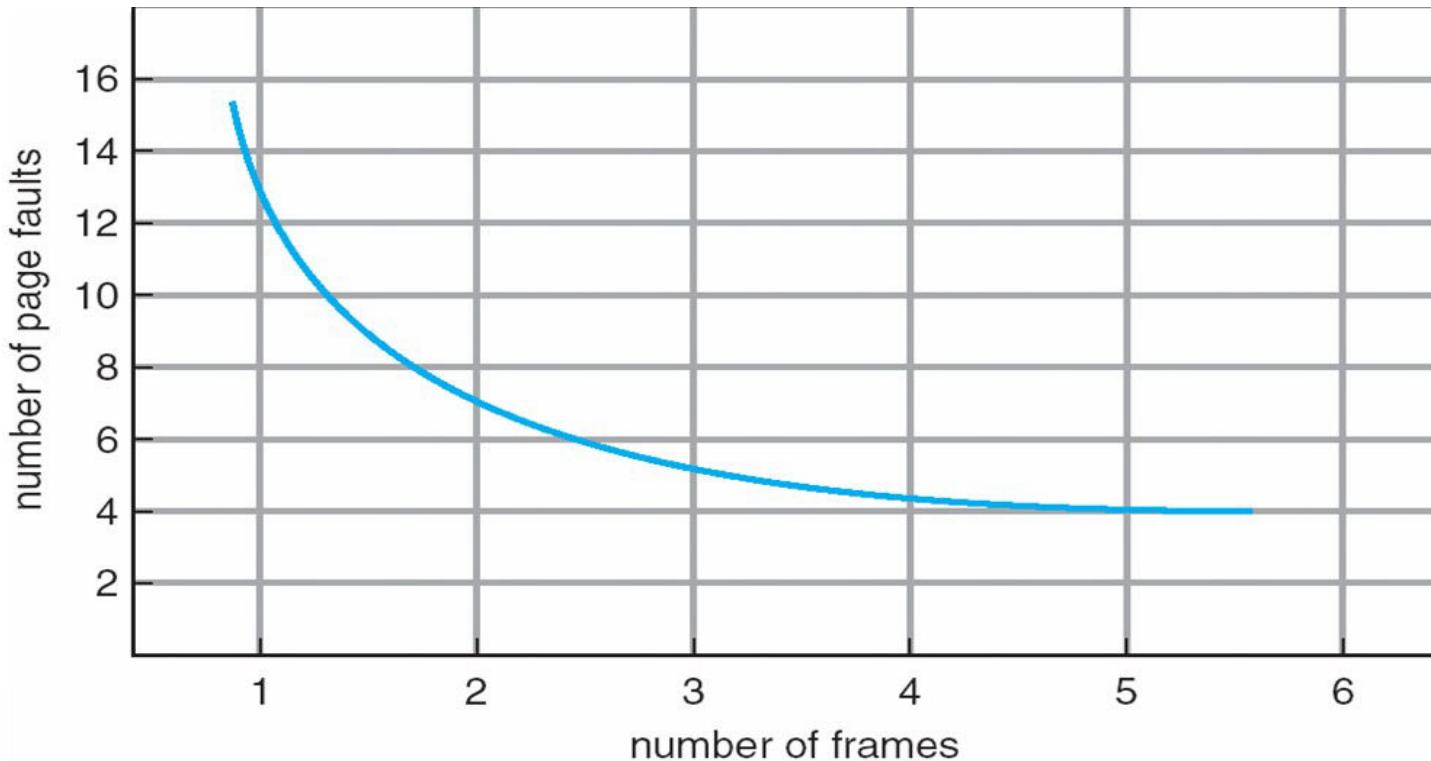


Two problems to solve

- Frame-allocation algorithm determines
 - How many frames to give each process
 - Which frames to replace
- Page-replacement algorithm
 - Want lowest page-fault rate on both first access and re-access

Evaluating algorithm: Reference string

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- String is just page numbers, not full addresses
- Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



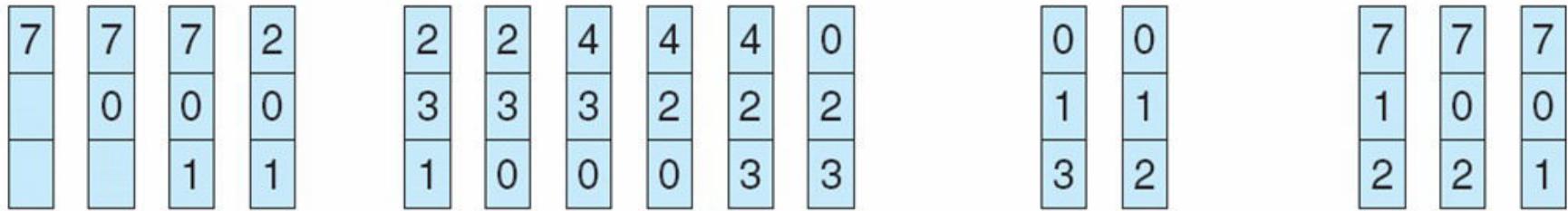
An
Expectation

**More page
Frames
Means less
faults**

FIFO Algorithm

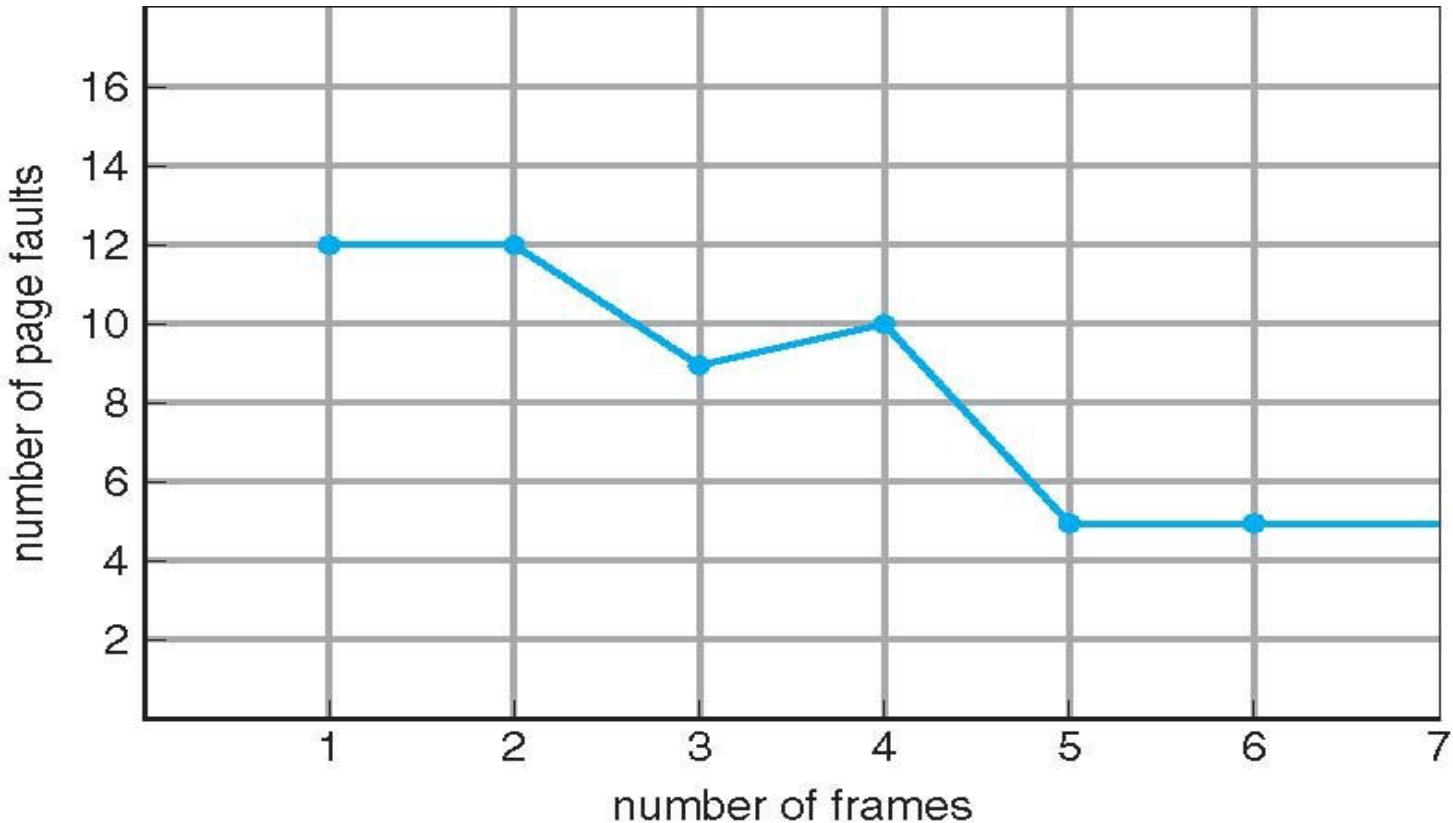
reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

FIFO Algorithm: Balady's anamoly



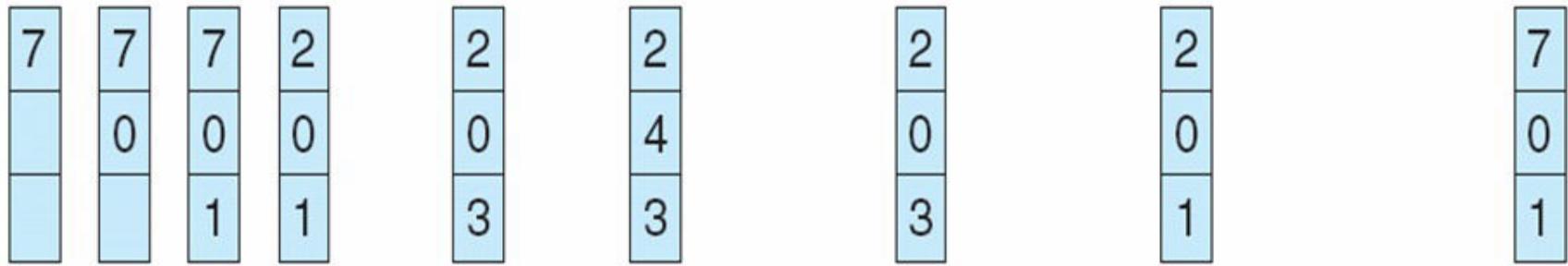
Optimal Algorithm

- Replace page that will not be used for longest period of time
- 9 is optimal #replacements for the example on the next slide
- How do you know this?
- Can't read the future
- Used for measuring how well your algorithm performs

Optimal page replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

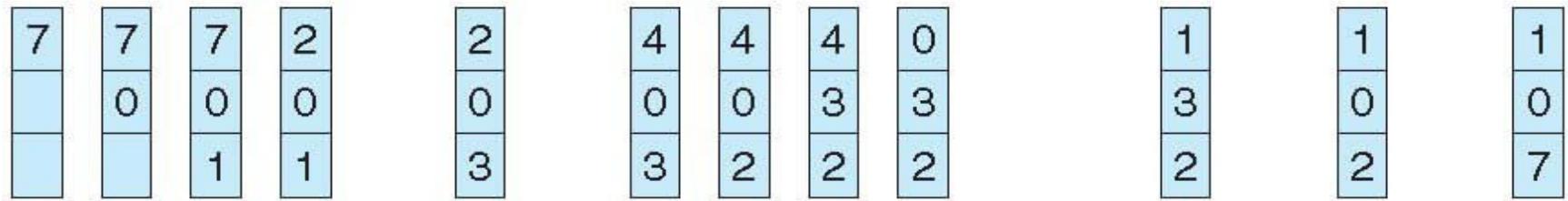


page frames

Least Recently Used: an approximation of optimal

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Use past knowledge rather than future

Replace page that has not been used in the most amount of time

Associate time of last use with each page

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

LRU: Counter implementation

- **Counter implementation**
- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
- When a page needs to be changed, look at the counters to find smallest value
- Search through table needed

LRU: Stack implementation

- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
- requires 6 pointers to be changed and
- each update more expensive
- But no need of a search for replacement

reference string

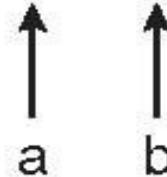
4 7 0 7 1 0 1 2 1 2 7 1 2

2
1
0
7
4

stack
before
a

7
2
1
0
4

stack
after
b



**Use Of A
Stack to
Record The**

Most

Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n + 1$ frames
- Do not suffer from Balady's anomaly
- For example: Optimal, LRU

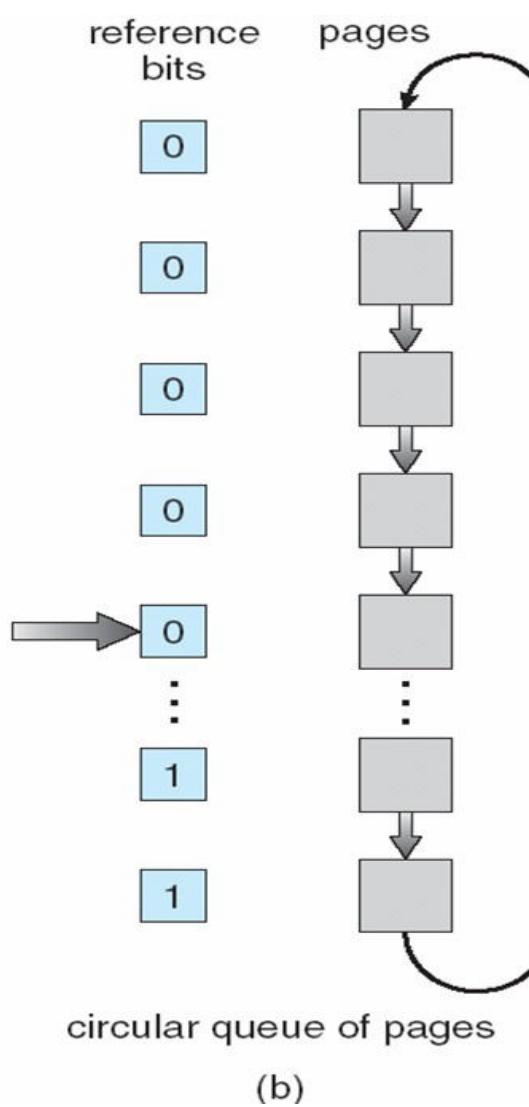
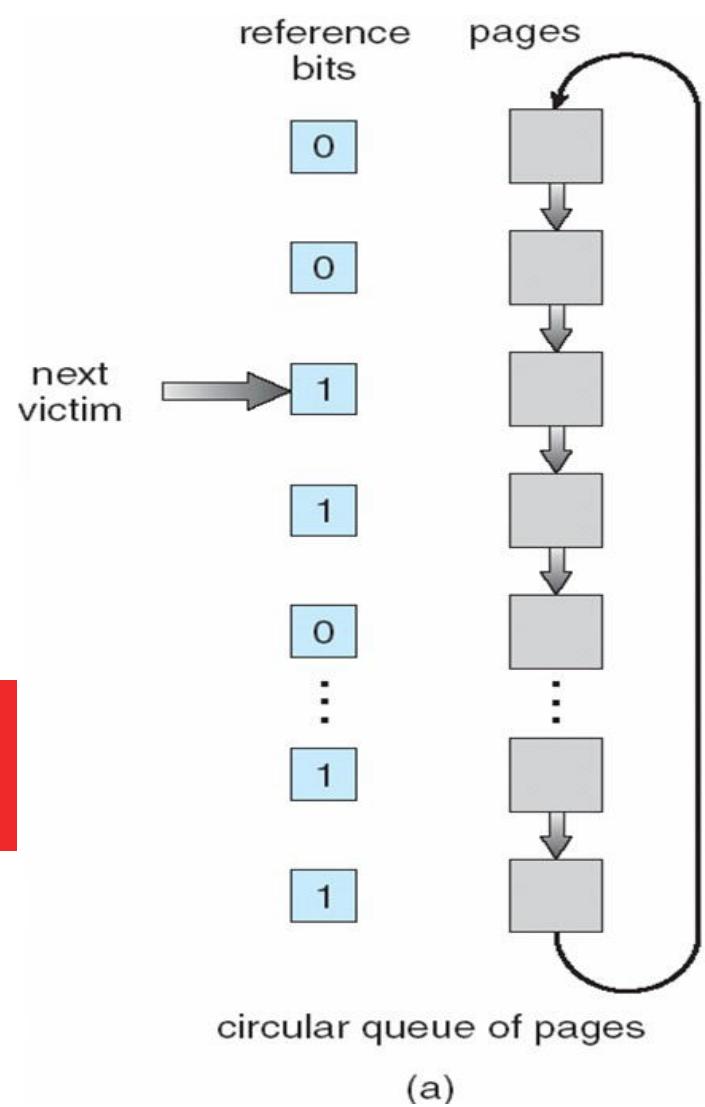
LRU: Approximation algorithms

- LRU needs special hardware and still slow
- **Reference bit**
- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace any with reference bit = 0 (if one exists)
- We do not know the order, however

LRU: Approximation algorithms

- **Second-chance algorithm**
- FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- **An implementation of second-chance: Clock replacement**
 - If page to be replaced has
 - Reference bit = 0 -> replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

Second- Chance (clock) Page- Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- Not common
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page buffering algorithms

- Keep a pool of free frames, always
- Then frame available when needed, not found at fault time
- Read page into free frame and select victim to evict and add to free pool
- When convenient, evict victim
- Possibly, keep list of modified pages
- When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
- If referenced again before reused, no need to load contents again from disk
- Generally useful to reduce penalty if wrong victim frame selected

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
- Fault + page not in memory
- Minor fault
- Fault, but page is in memory
- For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run
 - \$ ps -eo min_flt,maj_flt,cmd

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
- OS keeps copy of page in memory as I/O buffer
- Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
 - Raw disk mode
 - Bypasses buffering, locking, etc

Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process

Dynamic as degree of multiprogramming, process sizes change

$$m = 64$$

s_i = size of process p_i

$$s_1 = 10$$

$S = \sum s_i$

$$s_2 = 127$$

m = total number of frames

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

a_i = allocation for p_i = $\frac{s_i}{S} \times m$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
- select for replacement one of its frames
- select for replacement a frame from a process with lower priority number

Global Vs Local allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- But then process execution time can vary greatly
- But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
- More consistent per-process performance
- But possibly underutilized memory



Virtual Memory – Remaining topics

Agenda

- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

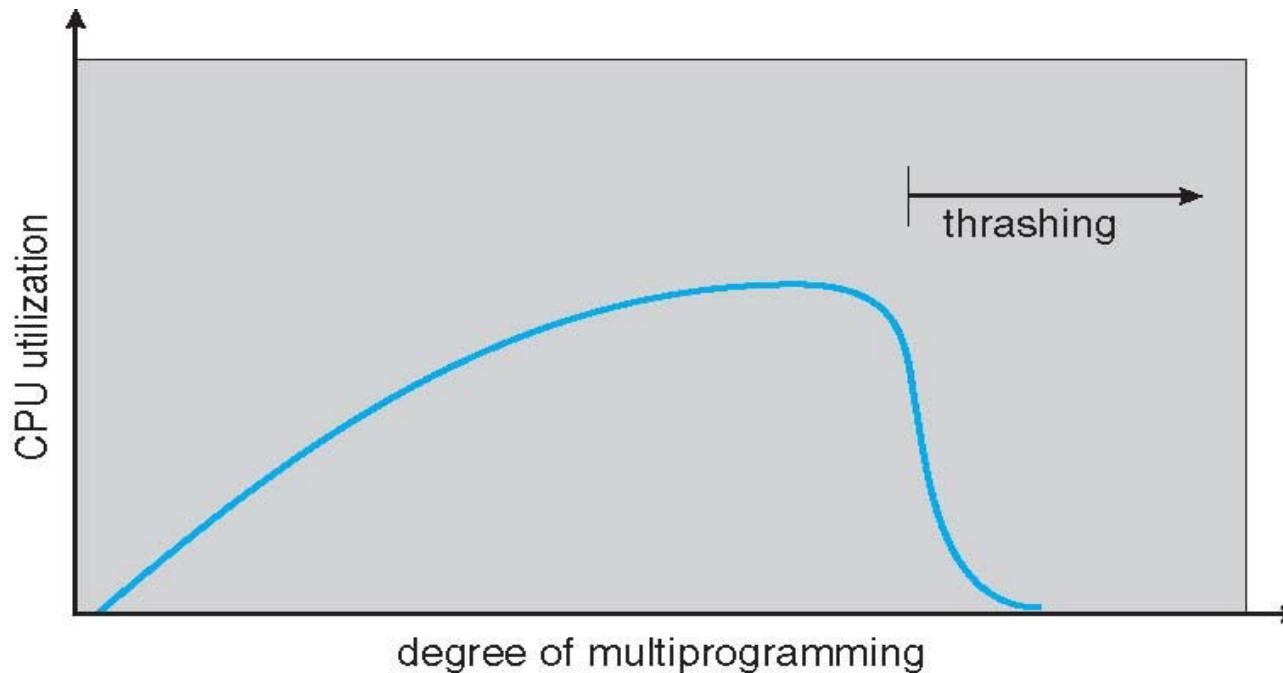


Thrashing

Thrashing

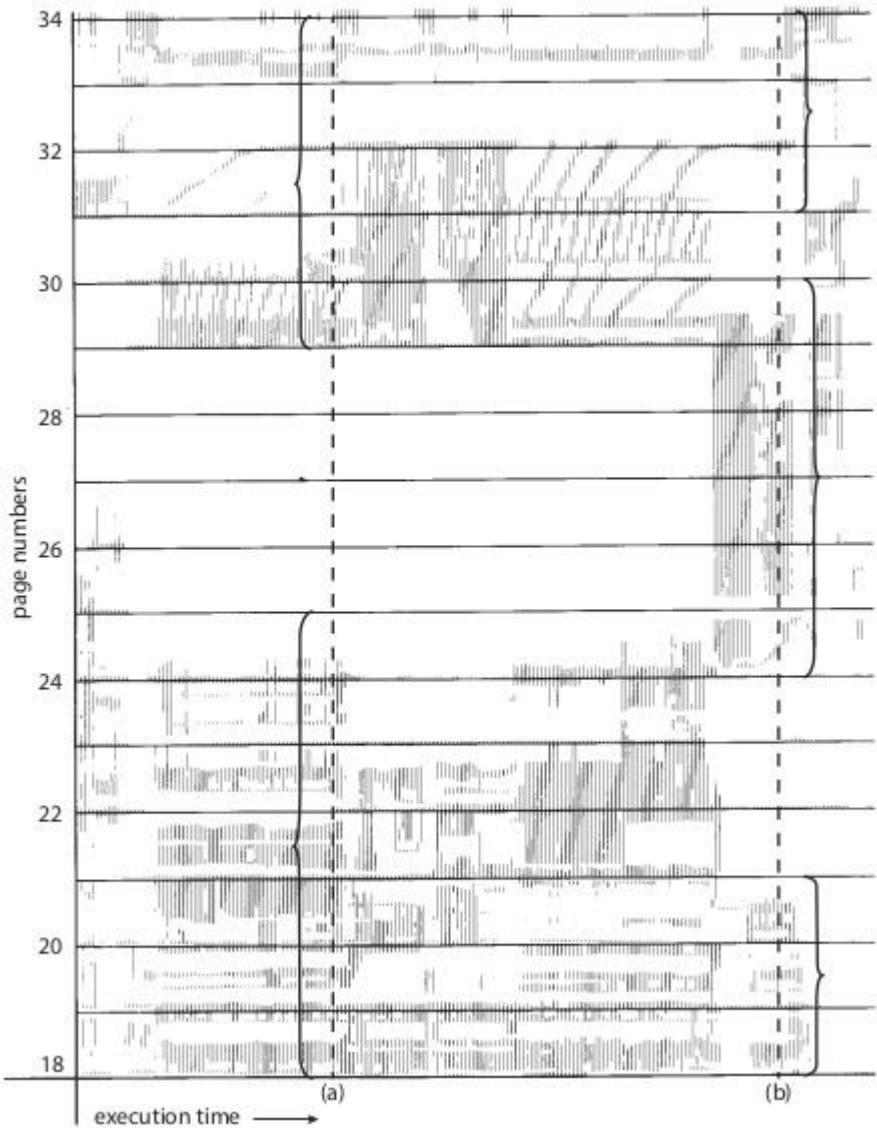
- If a process does not have “enough” pages, the page-fault rate is very high
- Page fault to get page
- Replace existing frame
- But quickly need replaced frame back
- This leads to:
 - Low CPU utilization
 - Operating system thinking that it needs to increase the degree of multiprogramming
 - Another process added to the system
 - Thrashing : a process is busy swapping pages in and out

Thrashing



Demand paging and thrashing

- Why does demand paging work?
- Locality model
- Process migrates from one locality to another
- Localities may overlap
- Why does thrashing occur?
- size of locality > total memory size
- Limit effects by using local or priority page replacement



Locality In A Memory- Reference Pattern

Working set model

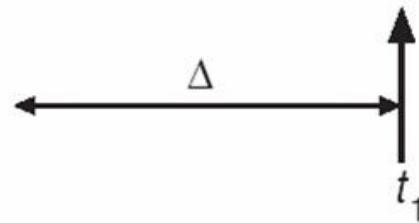
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
- Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
 - total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set model

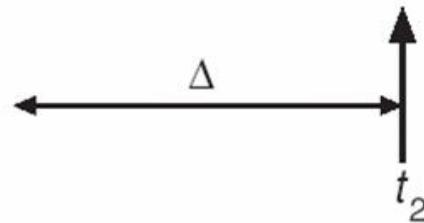
- $D = \sum WSS_i \equiv$ total demand frames
- Approximation of locality
- if $D > m$ (total available frames) \Rightarrow Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$

Timer interrupts after every 5000 time units

Keep in memory 2 bits for each page

Whenever a timer interrupt occurs, copy (to memory) and sets the values of all reference bits to 0

If one of the bits in memory = 1 \Rightarrow page in working set

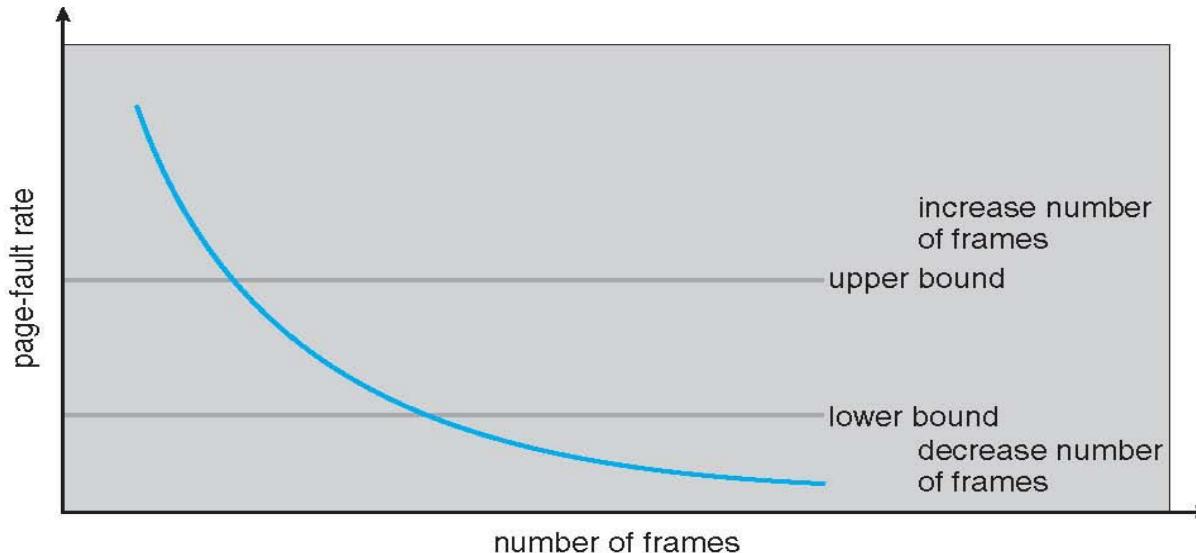
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

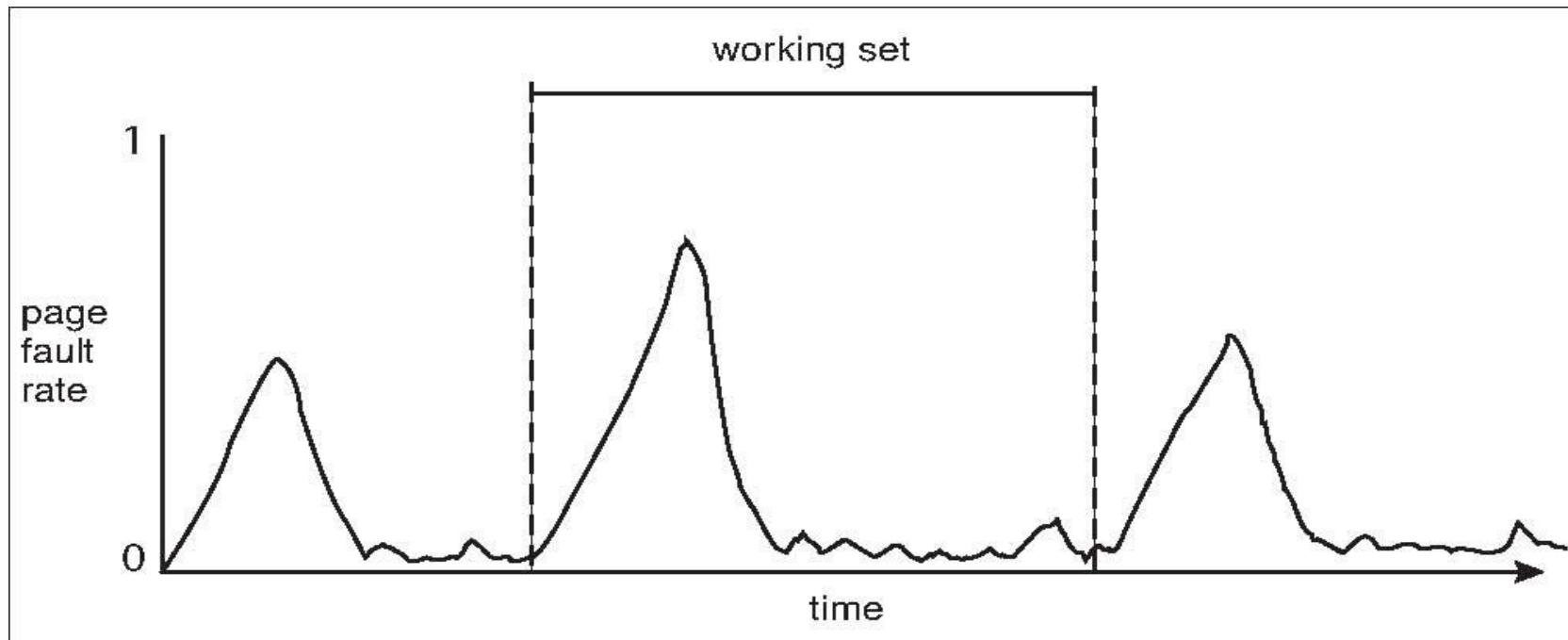
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

If actual rate too low, process loses frame

If actual rate too high, process gains frame



Working Sets and Page Fault Rates



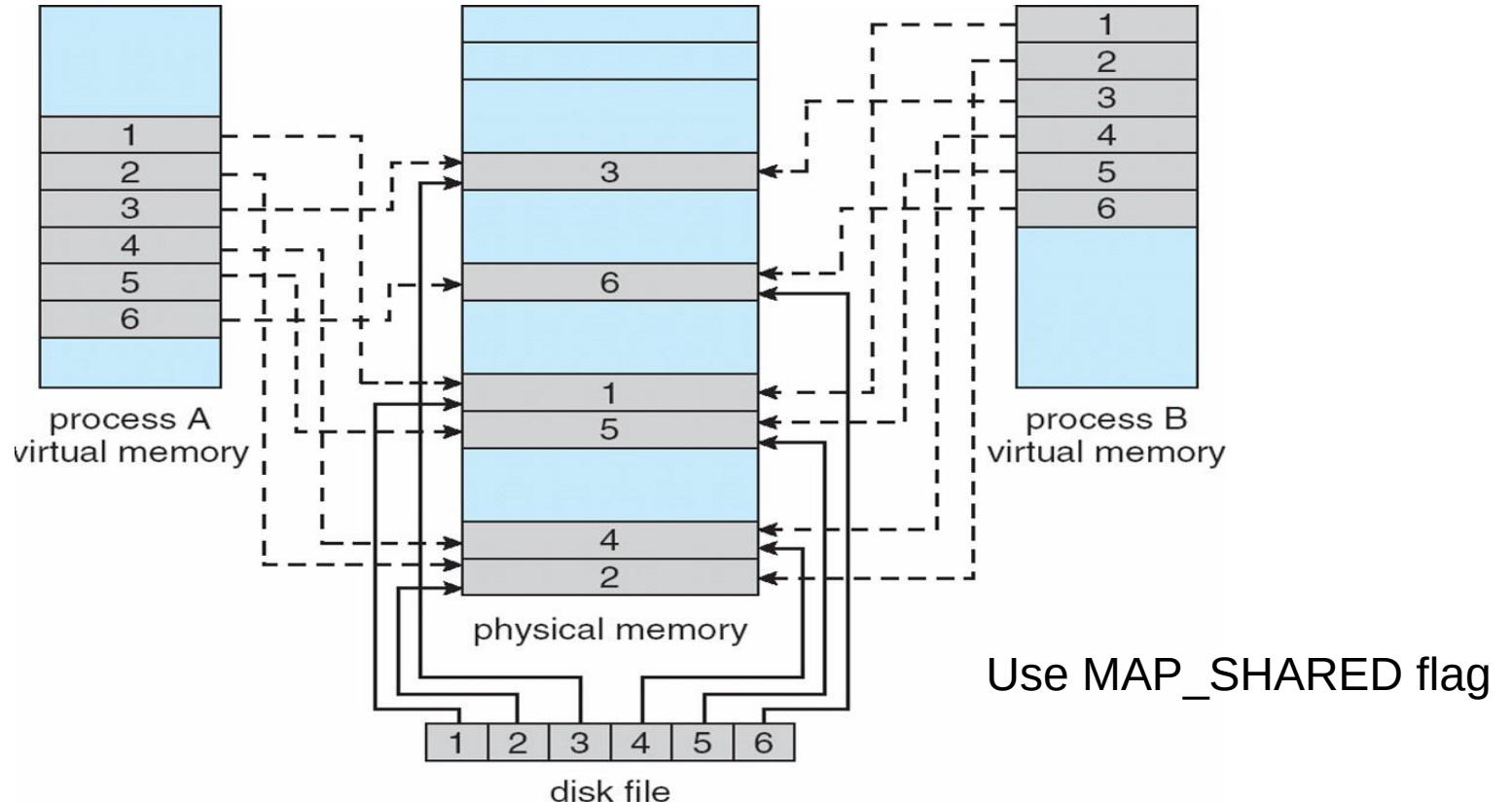


Memory Mapped Files

Memory-Mapped Files

- First, let's see a demo of using mmap()

Memory-Mapped Files



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging

A page-sized portion of the file is read from the file system into a physical page
Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?

Periodically and / or at file close() time
For example, when the pager scans for dirty pages

Memory-Mapped Files

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool

Kernel requests memory for structures of varying sizes

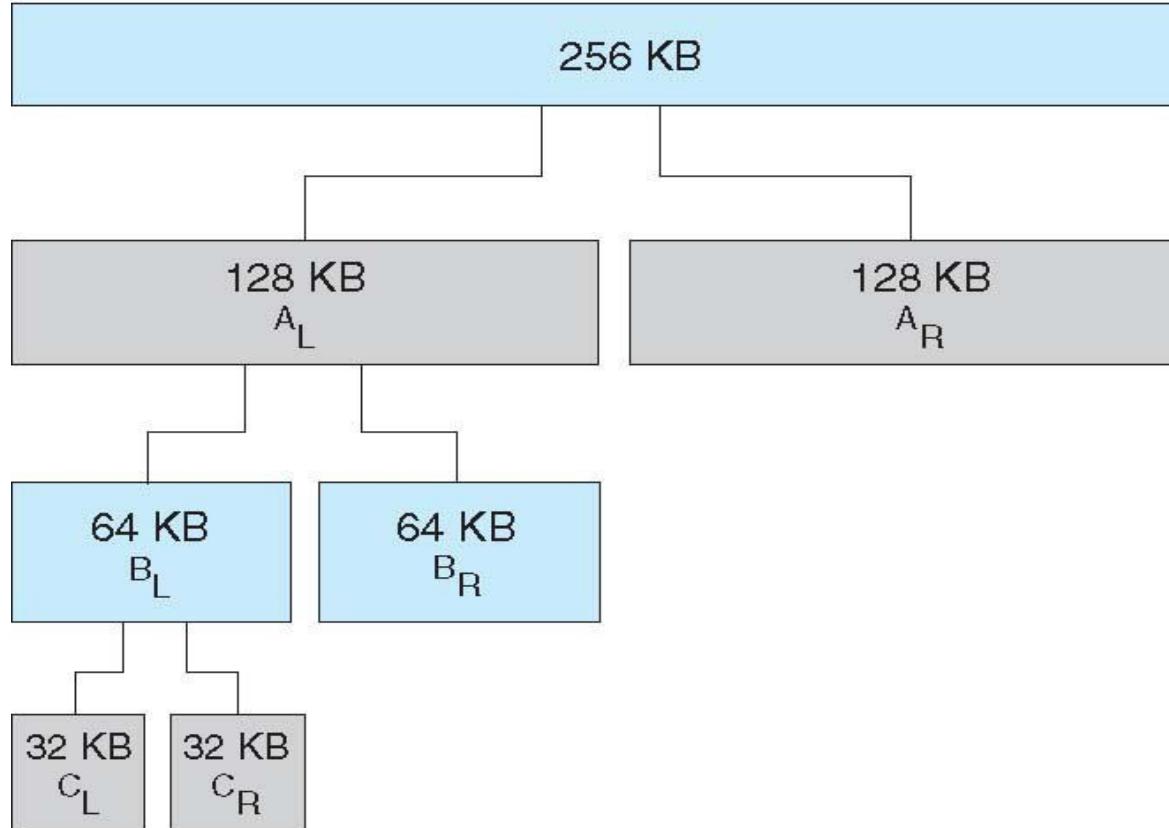
Some kernel memory needs to be contiguous

I.e. for device I/O

-

Buddy Allocator

physically contiguous pages



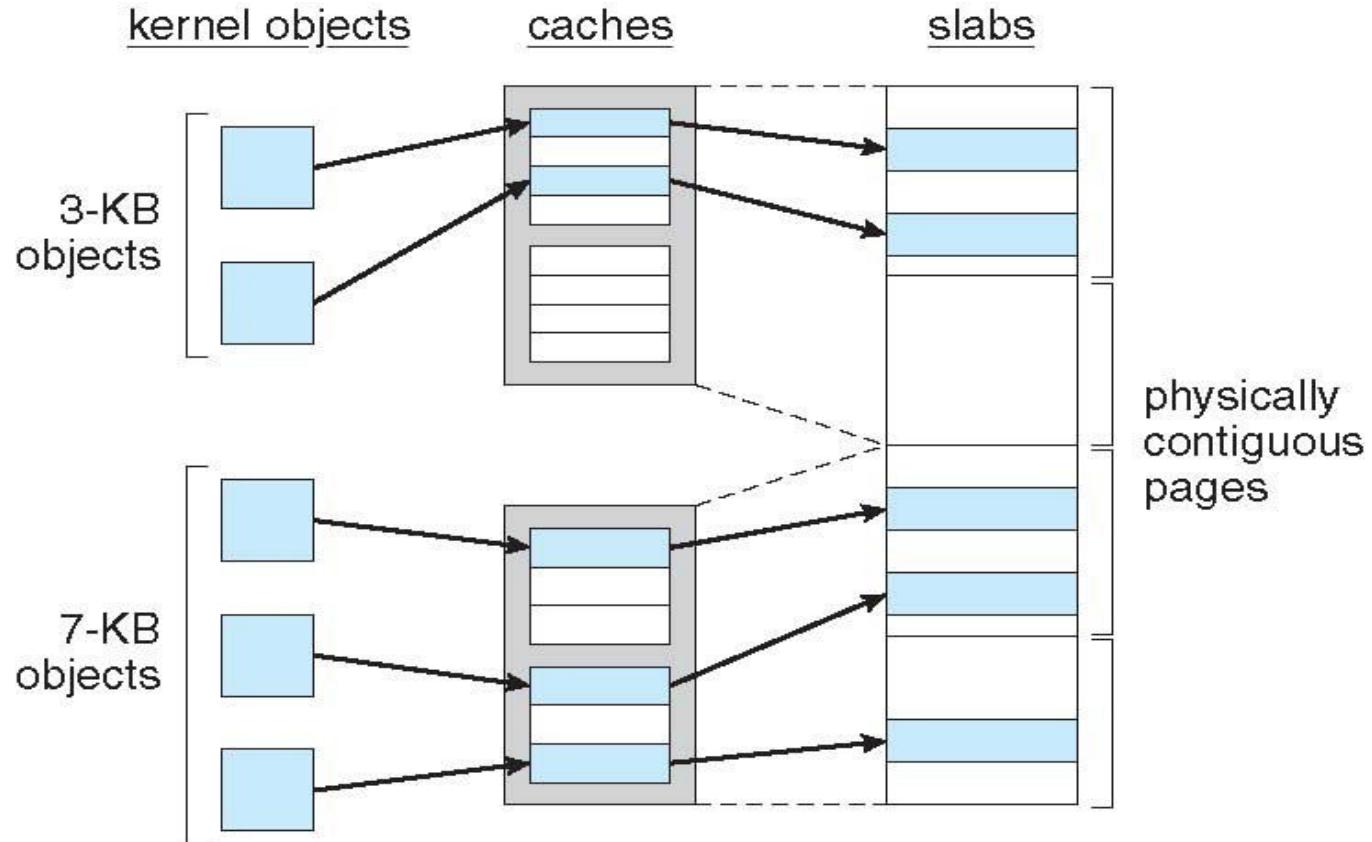
Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - Continue until appropriate sized chunk available

Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
 - Split into AL and Ar of 128KB each
 - One further divided into BL and BR of 64KB
 - One further into CL and CR of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
 - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
 - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction



Other considerations

Other Considerations --

Prepaging

- To reduce the large number of page faults that occurs at process startup
 - Prepage all or some of the pages a process will need, before they are referenced
 - But if prepaged pages are unused, I/O and memory was wasted
 - Assume s pages are prepaged and α of the pages is used
 - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
- α near zero --> prepaging loses

Page Size

- Sometimes OS designers have a choice
Especially if running on custom-built CPU
- Page size selection must take into consideration:
Fragmentation

Page table size

Resolution

I/O overhead

Number of page faults

Locality

TLB size and effectiveness

- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
Otherwise there is a high degree of page faults
- Increase the Page Size
This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure

- `Int [128, 128] data;`

- Each row is stored in one page

- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

$$128 \times 128 = 16,384 \text{ page faults}$$

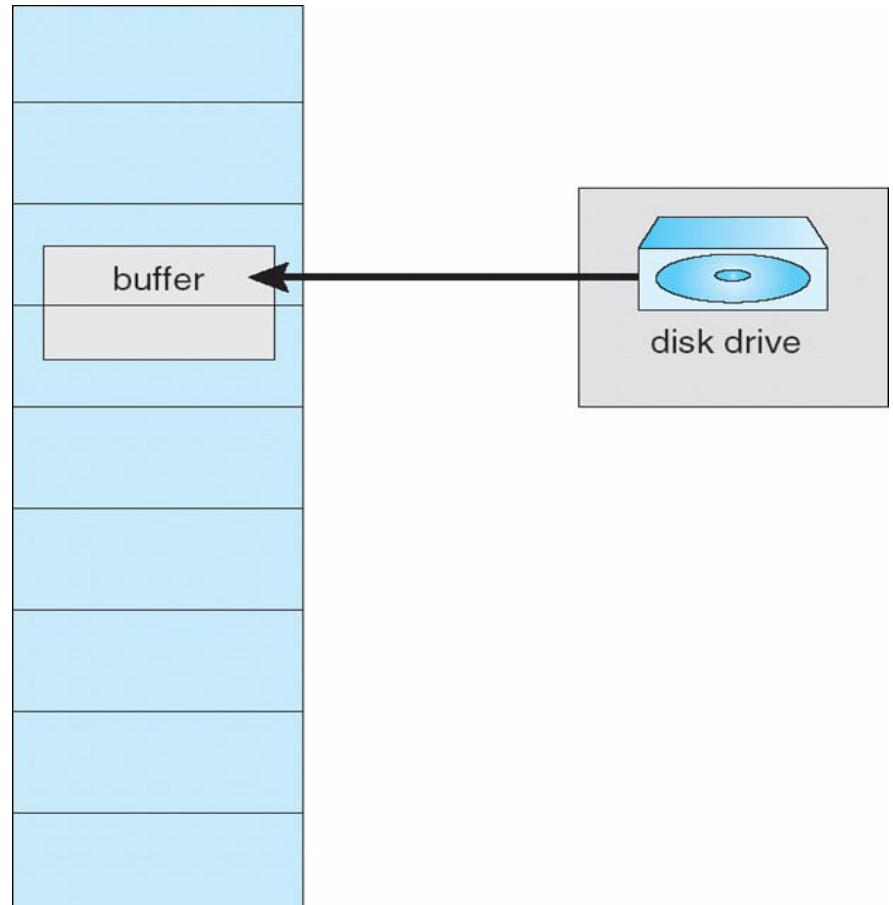
- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

$$128 \text{ page faults}$$

I/O Interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



Threads, Signals

Abhijit A M

abhijit.comp@coep.ac.in

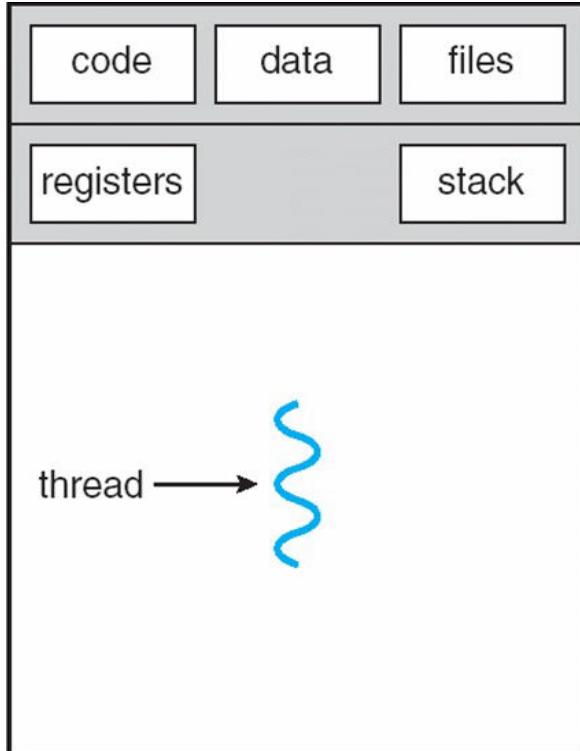
Threads

- **thread — a fundamental unit of CPU utilization**
- **A separate control flow within a program**
- **set of instructions that execute “concurrently” with other parts of the code**
- **Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time**
- **Threads run within application**
- **An application can be divided into multiple parts**
- **Each part may be written to execute as a threads**
- **Let's see an example**
-

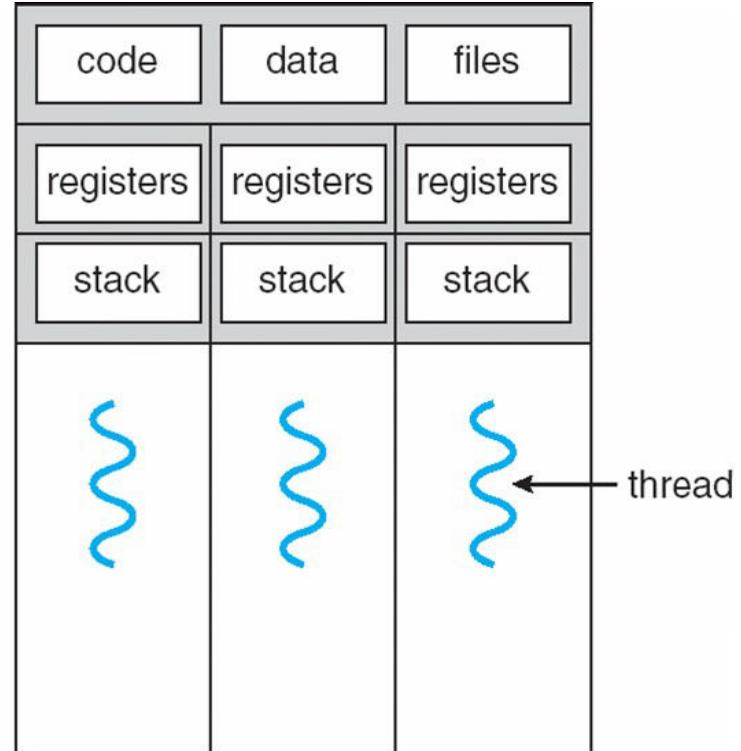
Threads

- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single vs Multithreaded process

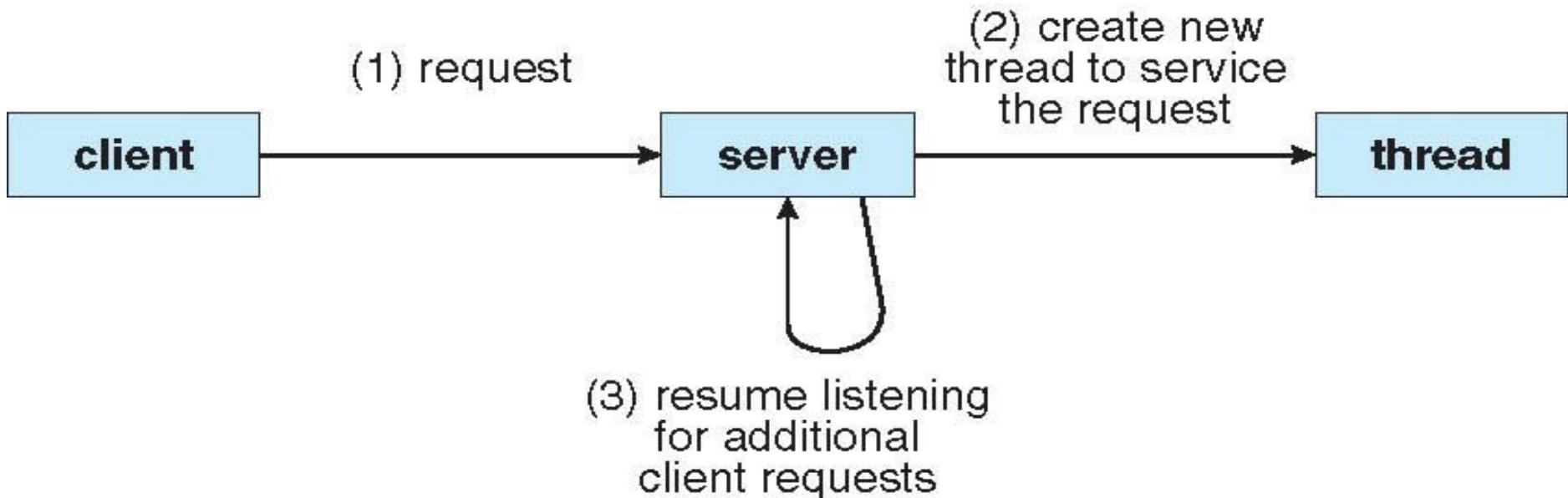


single-threaded process



multithreaded process

A multithreaded server

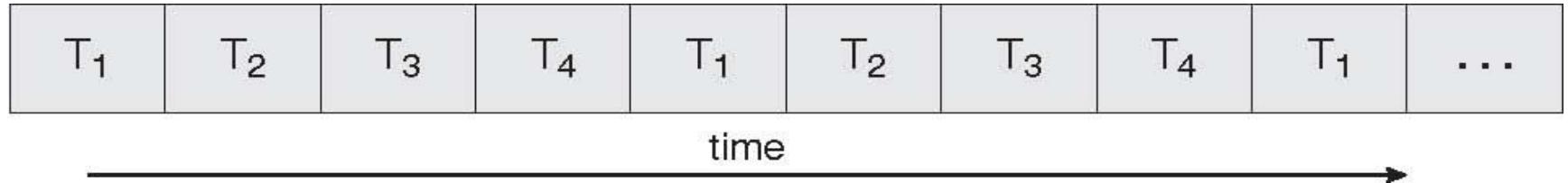


Benefits of threads

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**

Single vs Multicore systems

single core



Single core : Concurrency possible

Multicore : parallel execution possible

core 1



core 2



Multicore programming

- Multicore systems putting pressure on programmers, challenges include:
- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging
-

User vs Kernel Threads

- **User Threads:** Thread management done by user-level threads library
- Three primary thread libraries:
 - **POSIX Pthreads**
 - **Win32 threads**
 - **Java threads**
- **Kernel Threads:** Supported by the Kernel
 - Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

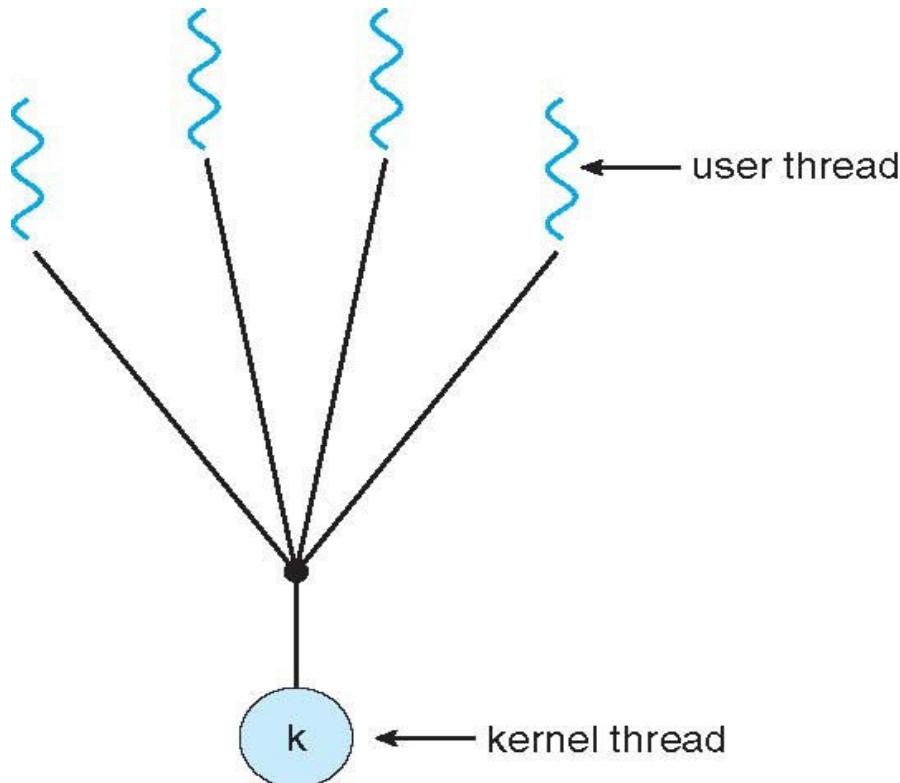
User threads vs Kernel Threads

- User threads
- User level library provides a “typedef” called threads
- The scheduling of threads needs to be implemented in the user level library
- Need some type of timer handling functionality at user level execution of CPU
- OS needs to provide system calls for this
- Kernel does not know that there are threads!
- Kernel Threads
- Kernel implements concept of threads
- Still, there may be a user level library, that maps kernel concept of threads to “user concept” since applications link with user level libraries
- Kernel does scheduling!

Multithreading models

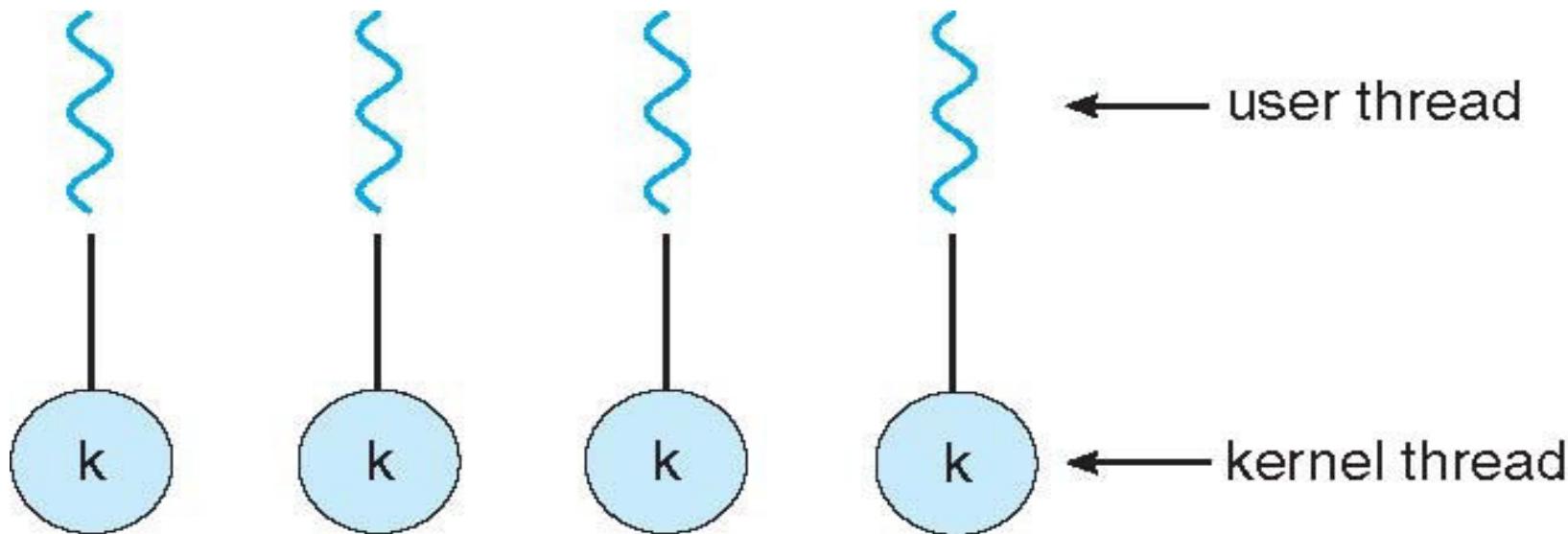
- How to map user threads to kernel threads?
- Many-to-One
- One-to-One
- Many-to-Many
- What if there are no kernel threads?
- Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
- Is One-One possible?

Many-One Model



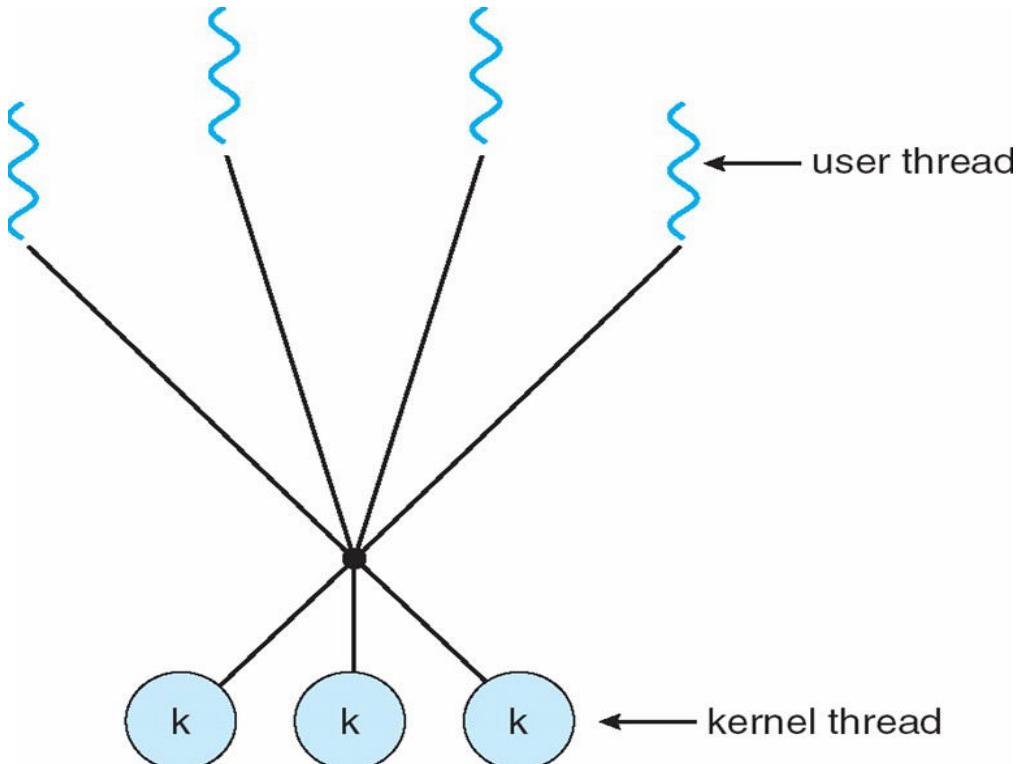
- **Many user-level threads mapped to single kernel thread**
- **Examples:**
- **Solaris Green Threads**
- **GNU Portable Threads**

One-One Model



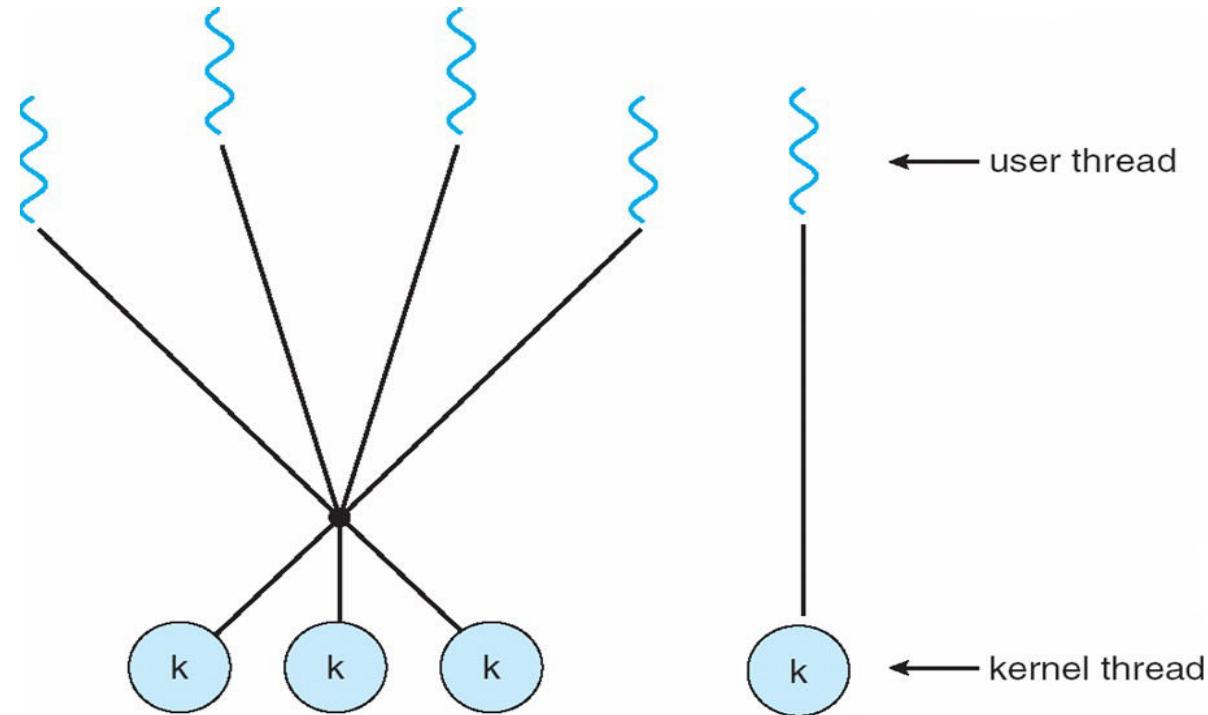
- Each user-level thread maps to kernel thread
- Examples
- Windows NT/XP/2000
- Linux
- Solaris 9 and later

Many-Many Model



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

Thread Libraries

Thread libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
- Library entirely in user space
- Kernel-level library supported by the OS

pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

Demo of pthreads code

**Demonstration on Linux – see the code,
compile and execute it.**

Other libraries

- Windows threading API
- `CreateThread(...)`
- `WaitForSingleObject(...)`
- `CloseHandle(...)`
- Java Threads
- The `Threads` class
- The `Runnable` Interface

Issues with threads

- Semantics of fork() and exec() system calls
- Does fork() duplicate only the calling thread or all threads?
- Thread cancellation of target thread
- Terminating a thread before it has finished
- Two general approaches:
 - Asynchronous cancellation terminates the target thread immediately.
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

More on threads

Thread pools

- Some kernels/libraries can provide system calls to : Create a number of threads in a pool where they await work, assign work/function to a waiting thread
- Advantages:
- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
-

Thread Local Storage (TLS)

- Thread-specific data, Thread Local Storage (TLS)
- Not local, but global kind of data for all functions of a thread, more like “static” data
- Create Facility needed for data private to thread
- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- gcc compiler provides the storage class keyword **thread** for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];  
  
int f0 {  
    a(); b(); c();  
}  
  
int g0 {  
    x(); y();  
}  
  
int main() {  
    th_create(...,f,...);  
    th_create(...,g,...);  
}  
  
//arr is visible to all of them!  
//need data for only f,a,b,c  
//need data for only g,x,y
```

Scheduler activations for threads

Library

```
--  
  
th_setup(int n) {  
    max_threads = n;  
    curr_threads = 0;  
}  
  
th_create(..., fn,...) {  
    if(curr_threads < max_threads)  
        create kernel thread;  
    schedule fn on one of the kernel threads;  
}
```

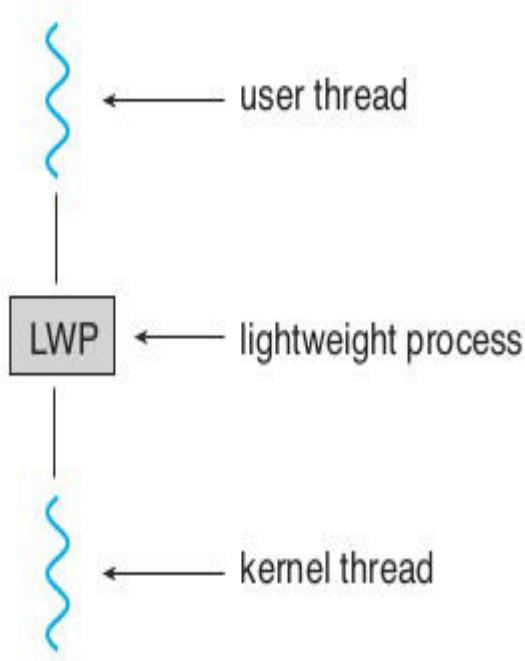
application

```
---  
  
f0 {  
    scanf();  
}  
  
g0 {  
    recv();  
}  
  
h0 {...}; i0 {...}  
  
main()  
{  
    th_setup(2);  
    th_create(...,f,...);  
    th_create(...,g,...);  
    th_create(...,h,...);  
    th_create(...,i,...);  
}
```

Scheduler activations for threads

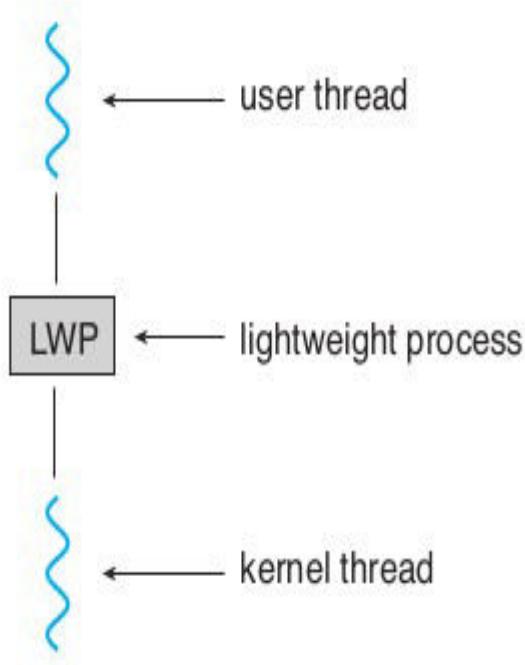
- **Scheduler Activations**
- **Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application**
- **Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library**
- **This communication allows an application to maintain the correct number kernel threads**

Issues with threads



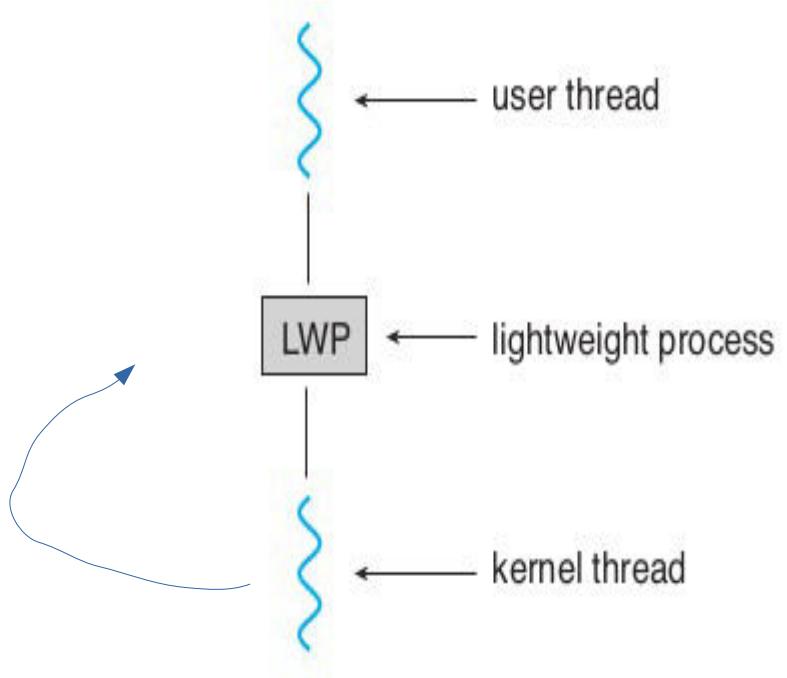
- **Scheduler Activations: LWP approach**
- **An intermediate data structure LWP**
- **appears to be a virtual processor on which the application can schedule a user thread to run.**
- **Each LWP attached to a kernel thread**
- **Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed**
-

Issues with threads



- **Scheduler Activations: LWP approach**
- **Kernel needs to inform application about events like: a thread is about to block, or wait is over**
- **This will help application relinquish the LWP or request a new LWP**

Issues with threads



- **The actual upcalls**

Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- Clone() system call to create a thread
-

Linux threads

- **clone() takes options to determine sharing on process create**
- **struct task_struct points to process data structures (shared or unique depending on clone options)**
- **fork() is a wrapper on top of clone()**

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

Issues in implementing threads project

- Understand the clone() system call completely
- Try out various possible ways of calling it
- Passing different options
- Passing a user-land buffer as stack
- How to save and restore context?
- C: setjmp, longjmp
- Setcontext, getcontext(), makecontext(), swapcontext() functions
- Sigaction is more powerful than signal
- Learn SIGALRM handling for timer and scheduler, timer_create() & timer_stop() system calls
- Customized data structure to store threads, and manage thread-lists for scheduling

Signals

Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal handling
- Synchronous and asynchronous
- A signal handler (a function) is used to process signals
- Signal is generated by particular event
- Signal is delivered to a process
- Then, signal is “handled” by the handler
-

Signals

- More about signals
- Different signals are typically identified as different numbers
- Operating systems provide system calls like `kill()` and `signal()` to enable processes to deliver and receive signals
- `Signal()` - is used by a process to specify a “signal handler”
 - a code that should run on receiving a signal
- `Kill()` is used by a process to send another process a signal
- There are restrictions on which process can send which signal to other processes

Demo

- Let's see a demo of signals with respect to processes
- Let's see signal.h
- /usr/include/signal.h
- /usr/include/asm-generic/signal.h
- /usr/include/linux/signal.h
- /usr/include/sys/signal.h
- /usr/include/x86_64-linux-gnu/asm/signal.h
- /usr/include/x86_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

Signal handling by OS

```
Process 12323 {  
    signal(19, abcd);  
}
```

OS: sys_signal {

Note down that process 12323
wants to handle signal number
19 with function abcd

```
}
```

```
Process P1 {  
    kill (12323, 19) ;  
}
```

OS: sys_kill {

Note down in PCB of process 12323 that signal
number 19 is pending for you.

```
}
```

When process 12323 is scheduled, at that time
the OS will check for pending signals, and
invoke the appropriate signal handler for a
pending signal.

Threads and Signals

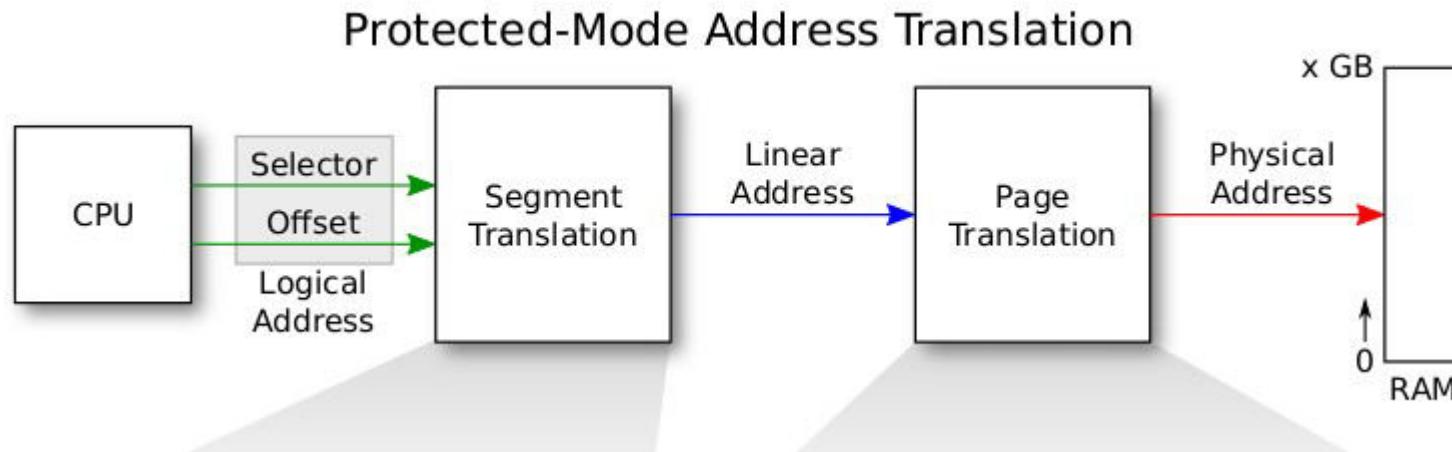
- **Signal handling Options:**
- **Deliver the signal to the thread to which the signal applies**
- **Deliver the signal to every thread in the process**
- **Deliver the signal to certain threads in the process**
- **Assign a specific thread to receive all signals for the process**

Some numbers and their ‘meaning’

- These numbers occur very frequently in discussion
- **0x 80000000 = 2 GB = KERNBASE**
- **0x 100000 = 1 MB = EXTMEM**
- **0x 80100000 = 2GB + 1MB = KERNLINK**
- **0x E000000 = 224 MB = PHYSTOP**
- **0x FE000000 = 3.96 GB = 4064 MB = DEVSPACE**
- **4096 – 4064 = 32 MB left on top**
-

X86 Memory Management

X86 address : protected mode address translation



X86 paging

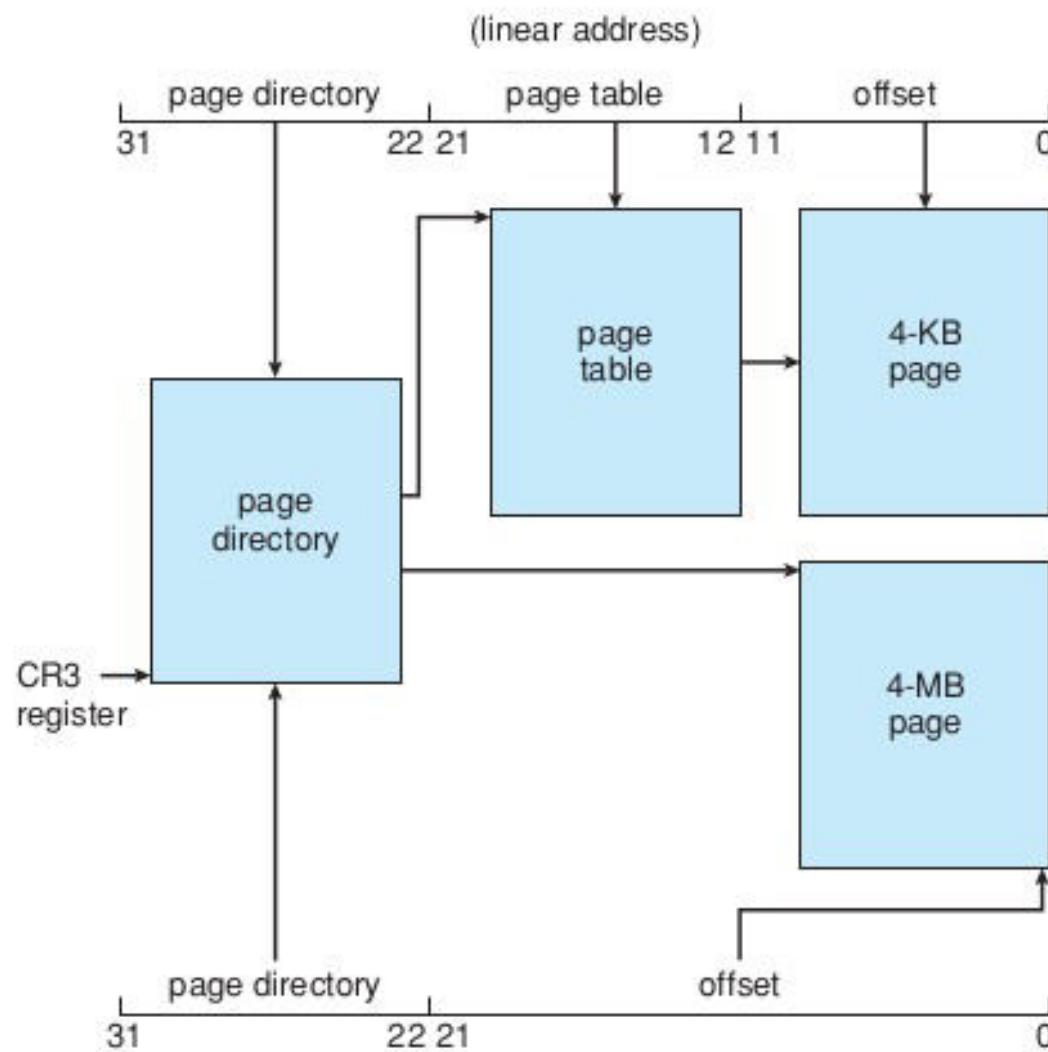
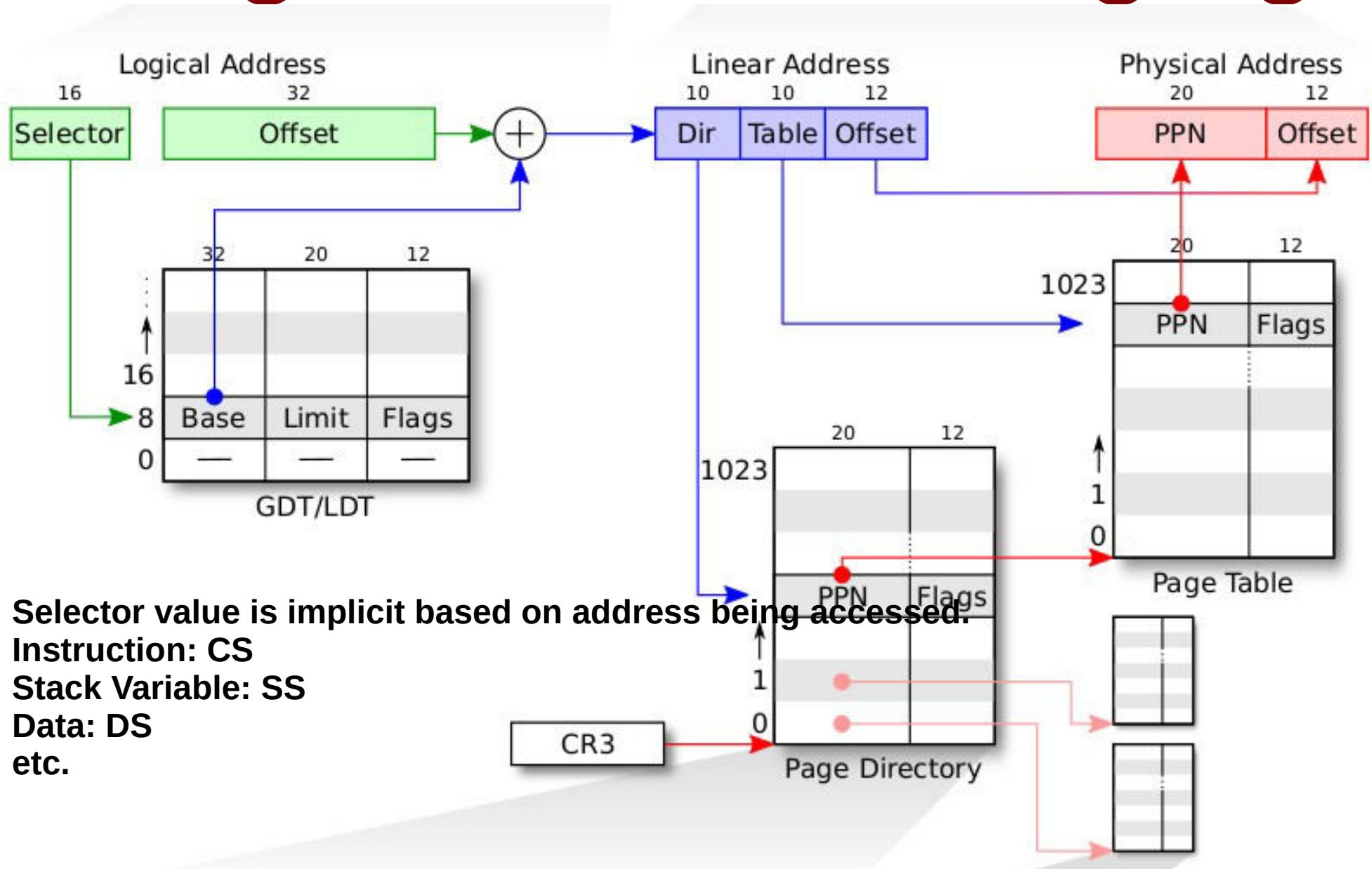


Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging



GDT Entry

31	16	15	0
Base 0:15 _____		Limit 0:15 _____	
63	56	55 52	51 48 47 40 39 32
Base 24:31 _____	Flags	Limit 16:19 _____	Access Byte Base 16:23 _____

Page Directory Entry (PDE) Page Table Entry (PTE)

31

Page table physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G S	P S	0	A	C D	W T	U	W	P			

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

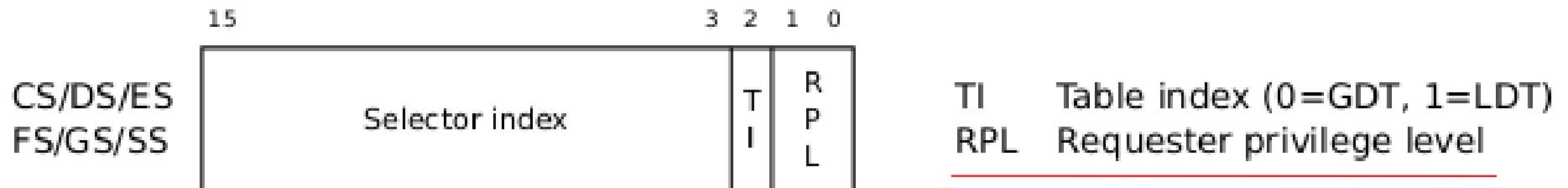
AVL Available for system use

31

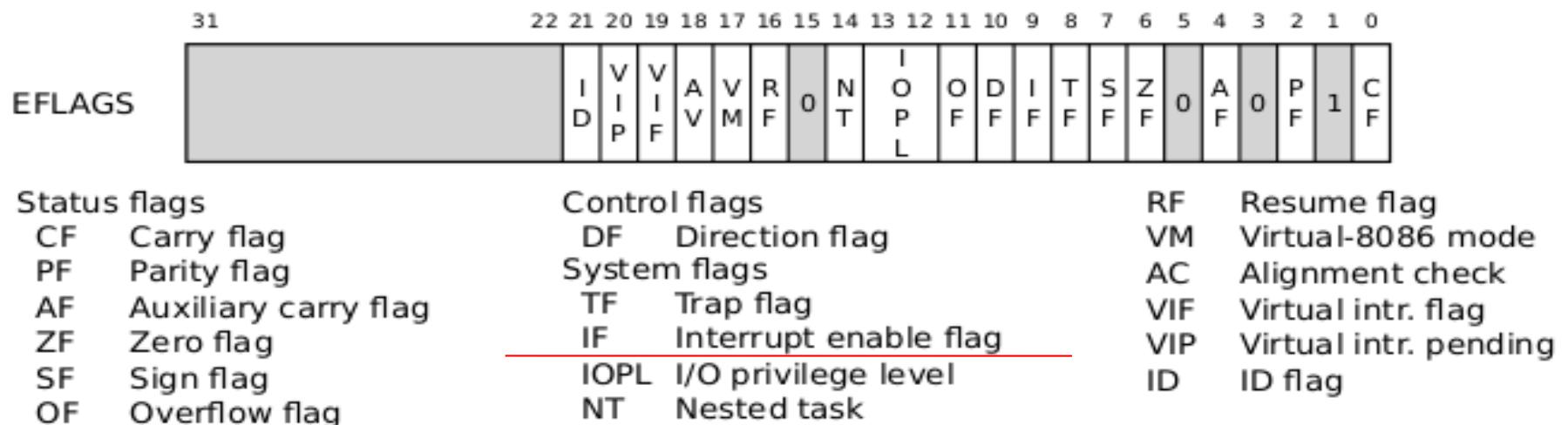
Physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G T	P A	D A	A D	C D	W T	U	W	P			

PTE

Segment selector



EFLAGS register



CR0

CR0	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0		
	P C N G D W	A M W P	N E T E M P P E T S M P E		
PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not

WP: Write protection on/off

PE: Protection Enabled --> protected mode.

CR2

CR2	31	0
		Page fault virtual address

CR3

CR3

	31	12 11	5 4 3 2	0
	Page-Directory-Table Base Address		P C D	P W T

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4

31	11	10	9	8	7	6	5	4	3	2	1	0
	O S X M	O S F X	P C G E	P G C E	M A S E	P A S E	P S E	D T S D	T P V I	P V M E	V	

VME Virtual-8086 mode extensions
PVI Protected-mode virtual interrupts
TSD Time stamp disable
DE Debugging extensions
PSE Page size extensions
PAE Physical-address extension

MCE Machine check enable
PGE Page-global enable
PCE Performance counter enable
OSFXSR OS FXSAVE/FXRSTOR support
OSXMM- OS unmasked exception support
EXCPT

mmu.h : paging related macros

```
#define PTXSHIFT          12      // offset of PTX in a linear address

#define PDXSHIFT           22      // offset of PDX in a linear address

#define PDX(va)    (((uint)(va) >> PDXSHIFT) & 0x3FF) // page directory index

#define PTX(va)    (((uint)(va) >> PTXSHIFT) & 0x3FF) // page table index

// construct virtual address from indexes and offset

#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// +-----10-----+-----10-----+-----12-----+
// | Page Directory | Page Table | Offset within Page |
// |       Index     |       Index   |                         |
// +-----+-----+-----+
// \--- PDX(va) --/ \--- PTX(va) --/
```

mmu.h : paging related macros

```
// Page directory and page table constants.

#define NPENTRIES          1024    // # directory entries per page directory

#define NPTENTRIES          1024    // # PTEs per page table

#define PGSIZE              4096    // bytes mapped by a page

#define PGROUNDUP(sz)      (((sz)+PGSIZE-1) & ~(PGSIZE-1))

#define PGROUNDDOWN(a)     (((a)) & ~ (PGSIZE-1))
```

mmu.h : paging related macros

```
// Page table/directory entry flags.
```

```
#define PTE_P          0x001 // Present
```

```
#define PTE_W          0x002 // Writeable
```

```
#define PTE_U          0x004 // User
```

```
#define PTE_PS         0x080 // Page Size
```

```
// Address in page table or page directory entry
```

```
#define PTE_ADDR(pte) ((uint)(pte) & ~0xFFFF) // get all but last  
12 bits
```

```
#define PTE_FLAGS(pte) ((uint)(pte) & 0xFFFF) // get last 12 bits
```

mmu.h : Segmentation related macros

// various segment selectors.

#define SEG_KCODE 1 // kernel code

#define SEG_KDATA 2 // kernel data+stack

#define SEG_UCODE 3 // user code

#define SEG_UDATA 4 // user data+stack

#define SEG_TSS 5 // this process's task state

mmu.h : Segmentation related macros

// various segment selectors.

#define SEG_KCODE 1 // kernel code

#define SEG_KDATA 2 // kernel data+stack

#define SEG_UCODE 3 // user code

#define SEG_UDATA 4 // user data+stack

#define SEG_TSS 5 // this process's task state

#define NSEGS 6

mmu.h : Segmentation related macros

```
struct segdesc { // 64 bit in size

    uint lim_15_0 : 16; // Low bits of segment limit

    uint base_15_0 : 16; // Low bits of segment base address

    uint base_23_16 : 8; // Middle bits of segment base address

    uint type : 4;      // Segment type (see STS_constants)

    uint s : 1;         // 0 = system, 1 = application

    uint dpl : 2;       // Descriptor Privilege Level

    uint p : 1;         // Present

    uint lim_19_16 : 4; // High bits of segment limit

    uint avl : 1;       // Unused (available for software use)

    uint rsv1 : 1;      // Reserved

    uint db : 1;        // 0 = 16-bit segment, 1 = 32-bit segment

    uint g : 1;          // Granularity: limit scaled by 4K when set

    uint base_31_24 : 8; // High bits of segment base address

};
```

mmu.h : Segmentation related code

// Application segment type bits

```
#define STA_X    0x8 // Executable segment
```

```
#define STA_W    0x2 // Writeable (non-executable segments)
```

```
#define STA_R    0x2 // Readable (executable segments)
```

// System segment type bits

```
#define STS_T32A  0x9 // Available 32-bit TSS
```

```
#define STS_IG32   0xE // 32-bit Interrupt Gate
```

```
#define STS_TG32   0xF // 32-bit Trap Gate
```

Code from bootasm.S bootmain.c is over!

Kernel is loaded.

Now kernel is going to prepare itself

main() in main.c

- Initializes “free list” of page frames
- In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
- Just like the first processor
- Creates the first process!
- Initializes LAPIC on each processor, IOAPIC
- Disables PIC
- “Console” hardware (the standard I/O)
- Serial Port
- Interrupt Descriptor Table
- Buffer Cache
- Files Table
- Hard Disk (IDE)

main() in main.c

```
int main(void) {
    kinit1(end,
P2V(4*1024*1024)); // phys
page allocator

    kvmalloc();    // kernel page
table

    void kinit1(void *vstart, void
*vend) {
        initlock(&kmem.lock,
"kmem");

        kmem.use_lock = 0;

        freerange(vstart, vend);

    }
}
```

main() in main.c

void

```
freerange(void *vstart, void  
*vend)
```

```
{
```

```
    char *p;
```

```
    p =  
(char*)PGROUNDUP((uint)vstar  
t);
```

```
    for(; p + PGSIZE <=  
(char*)vend; p += PGSIZE)
```

```
        kfree(p);
```

```
}
```

```
kfree(char *v) {  
    struct run *r;  
    if((uint)v % PGSIZE || v <  
end || V2P(v) >= PHYSTOP)  
        panic("kfree");  
    // Fill with junk to catch  
dangling refs.  
    memset(v, 1, PGSIZE);  
    if(kmem.use_lock)  
        acquire(&kmem.lock);  
    r = (struct run*)v;  
    r->next = kmem.freelist;  
    kmem.freelist = r;  
    if(kmem.use_lock)  
        release(&kmem.lock); }
```

lock
uselock
run *freelist

run *next

Allocated frame

run *next

Allocated frame

Allocated frame

run *next

run *next

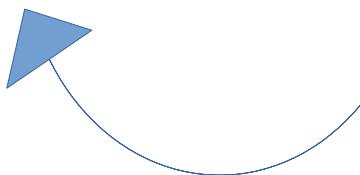
Allocated frame

RAM –
divided
into frames

Free List in XV6 Obtained after main() -> kinit1()

Pages obtained Between
end = 801154a8 = 2049 MB to **P2V(4MB) = 2052 MB**
Remember
Right now Logical = Physical address.

Actually like
this in memory



lock
uselock
run *freelist

kmem

Seen independently

run *

run *

run *

Back to main()

int

main(void) {

kinit1(end,
P2V(4*1024*1024)); //
phys page allocator

kvmalloc(); //
kernel page table

**// Allocate one page table
for the machine for the
kernel address**

**// space for scheduler
processes.**

void

kvmalloc(void)

{

kpgdir = setupkvm();

switchkvm();

}

Back to main()

```
int  
  
main(void) {  
  
    kinit1(end,  
P2V(4*1024*1024)); //  
phys page allocator  
  
    kvmalloc(); //  
kernel page table
```

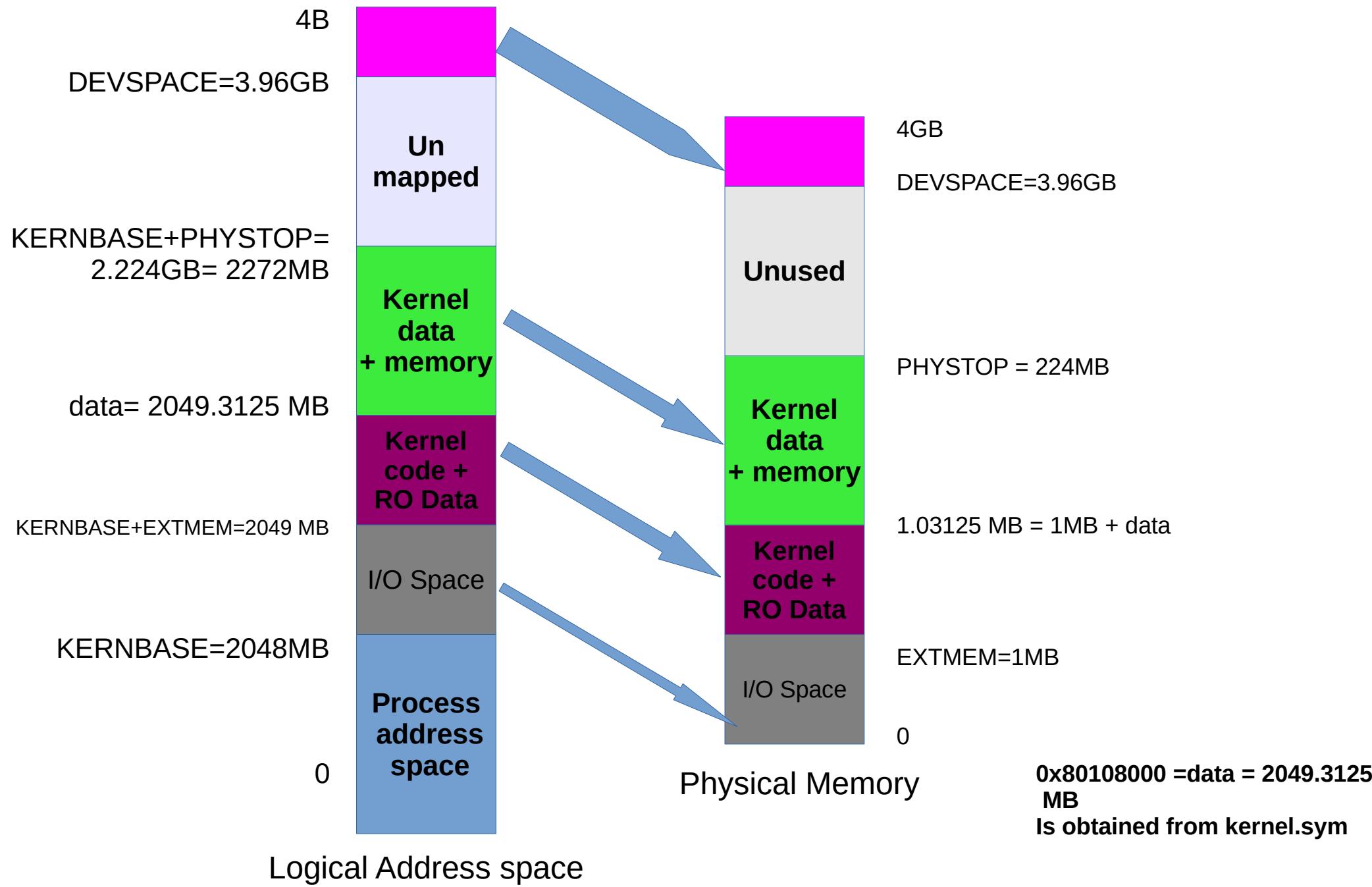
```
// Allocate one page table  
for the machine for the  
kernel address  
  
// space for scheduler  
processes.  
  
void  
  
kvmalloc(void)  
{  
  
    kpgdir = setupkvm(); //  
global var kpgdir  
  
    switchkvm();  
}
```

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;
    if((pgdir = (pde_t*)kalloc())
== 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) >
(void*)DEVSPACE)
        panic("PHYSTOP too
high");
```

```
for(k = kmap; k <
&kmap[NELEM(kmap)];
k++)
    if(mappages(pgdir, k-
>virt, k->phys_end - k-
>phys_start,
            (uint)k-
>phys_start, k->perm) <
0) {
        freevm(pgdir);
        return 0;
    }
    return pgdir;
}
```

```
static struct kmap {  
    void *virt;  
    uint phys_start;  
    uint phys_end;  
    int perm;  
} kmap[] = {  
    { (void*)KERNBASE, 0,           EXTMEM,  PTE_W}, // I/O space  
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata  
    { (void*)data,   V2P(data),   PHYSTOP,  PTE_W}, // kern data+memory  
    { (void*)DEVSPACE, DEVSPACE, 0,       PTE_W}, // more devices  
};
```

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page ta



Remidner: PDE and PTE entries

31

Page table physical page number	A V L	G S	P 0	A	C D	W T	U	W P	12 11 10 9 8 7 6 5 4 3 2 1 0
---------------------------------	-------------	--------	--------	---	--------	--------	---	--------	------------------------------

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

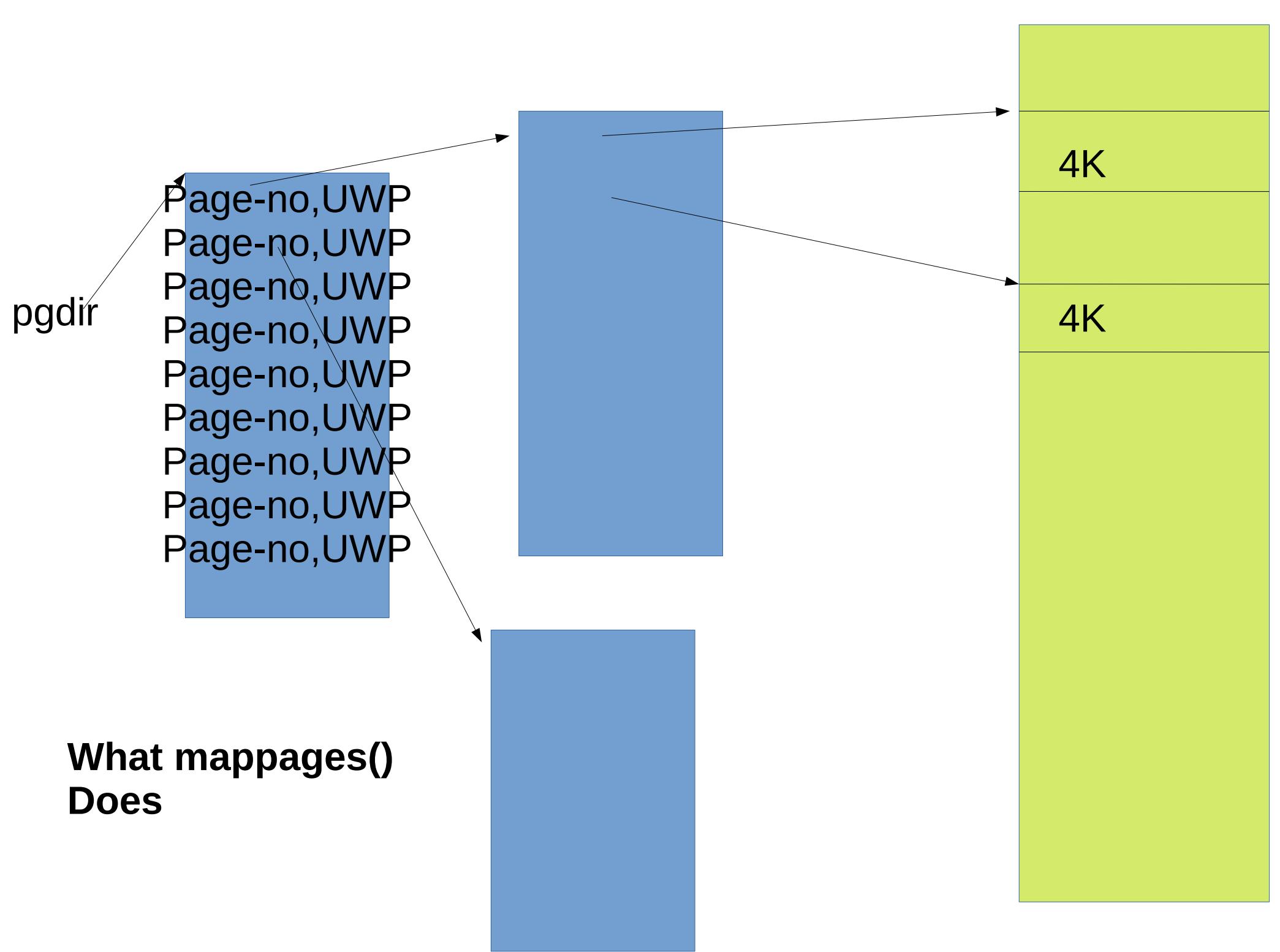
31

Physical page number	A V L	G A T	P D	A	C D	W T	U	W P	12 11 10 9 8 7 6 5 4 3 2 1 0
----------------------	-------------	-------------	--------	---	--------	--------	---	--------	------------------------------

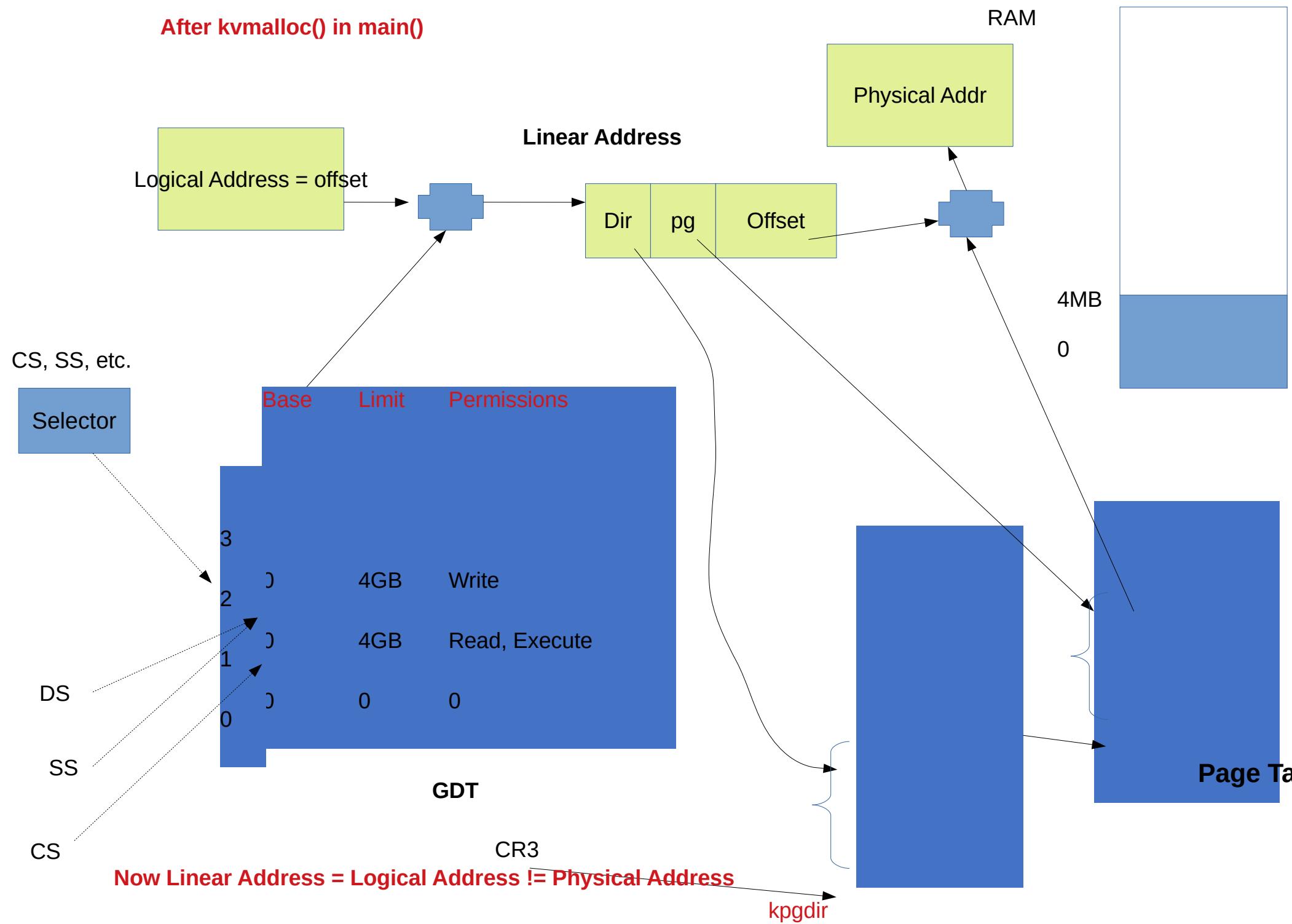
PTE

Before mappages()





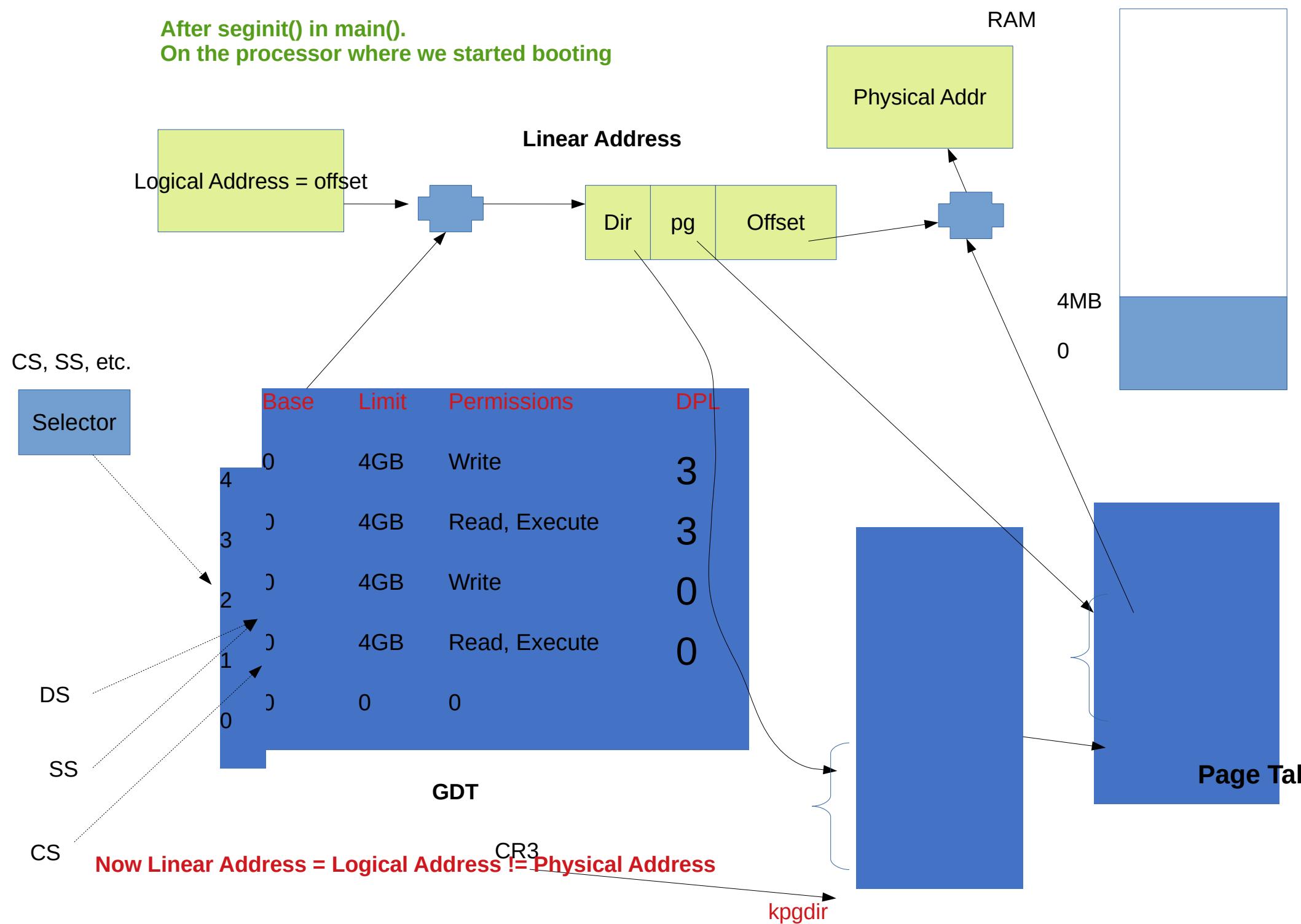
After kvmalloc() in main()



main() -> seginit()

- **Re-initialize GDT**
- **Once and forever now**
- **Just set 4 entries**
- **All spanning 4 GB**
- **Differing only in permissions and privilege level**

After seginit() in main().
On the processor where we started booting



After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

Memory Management

lock
uselock
run *freelist

run *next

Allocated frame

run *next

Allocated frame

Allocated frame

run *next

run *next

Allocated frame

RAM –
divided
into frames

Actually like
this in memory

Free List in XV6

lock
uselock
run *freelist

kmem

Seen independent

run *

run *

run *

exec()

- **sys_exec()**

- exec(path, argv)**

- **exec(pARTH, argv)**

- ip = namei(path))**

- readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf)**

- for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){**

- if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))**

- if((sz = allocuvm(pgdир, sz, ph.vaddr + ph.memsz)) == 0)**

- if(loaduvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)**

- }**

exec()

- exec(pARTH, argv)

// Allocate two pages at the next page boundary.

// Make the first inaccessible. Use the second as the user stack.

sz = PGROUNDUP(sz);

if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)

// Push argument strings, prepare rest of stack in ustack.

for(argc = 0; argv[argc]; argc++) {

 sp = (sp - (strlen(argv[argc]) + 1)) & ~3;

 if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)

Creation of first process by kernel

Why first process needs ‘special’ treatment?

- Normally process is created using fork()
- and typically followed by a call to exec()
- Fork will use the PCB of existing process to create a new process
- as a clone
- The first process has nothing to copy from!
- So it’s PCB needs to “built” by kernel code

Why first process needs ‘special’ treatment?

- XV6 approach
- Create the process as if it was created by “fork”
- Ensure that the process starts in a call to “exec”
- Let “Exec” do the rest of the JOB as expected
- In this case exec() will call
- exec(“/init”, NULL);
- See the code of init.c
- opens console() device for I/O; dups 0 on 1 and 2!
- Same device file for I/O
- forks a process and execs (“sh”) on it.
- Itself keeps waiting for zombie processes

Why first process needs ‘special’ treatment?

- What needs to be done ?
- Build struct proc by hand
- How data structures (proc, stack, etc) are hand-crafted so that when kernel returns, the process starts in code of init

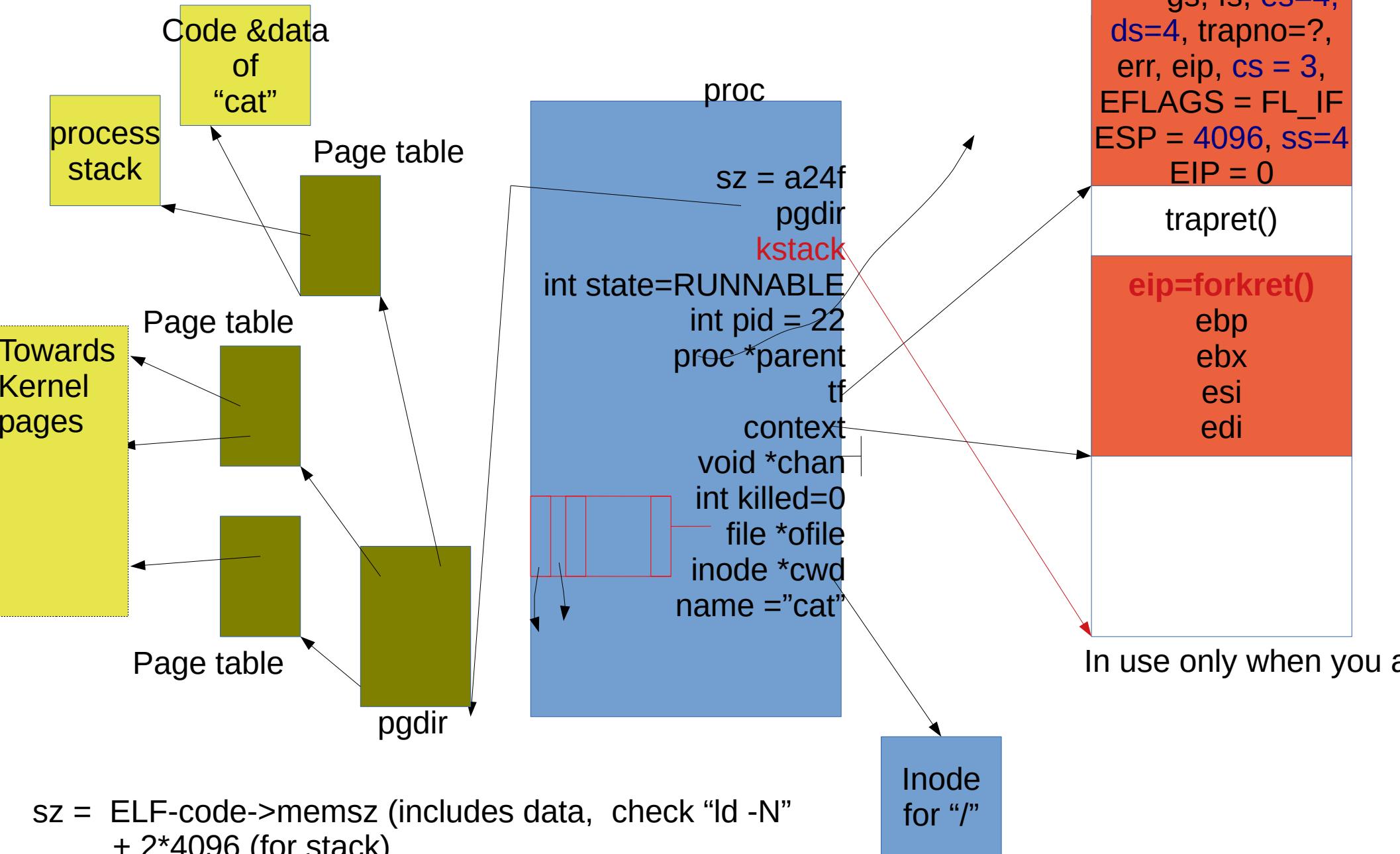
Imp Concepts

- **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**

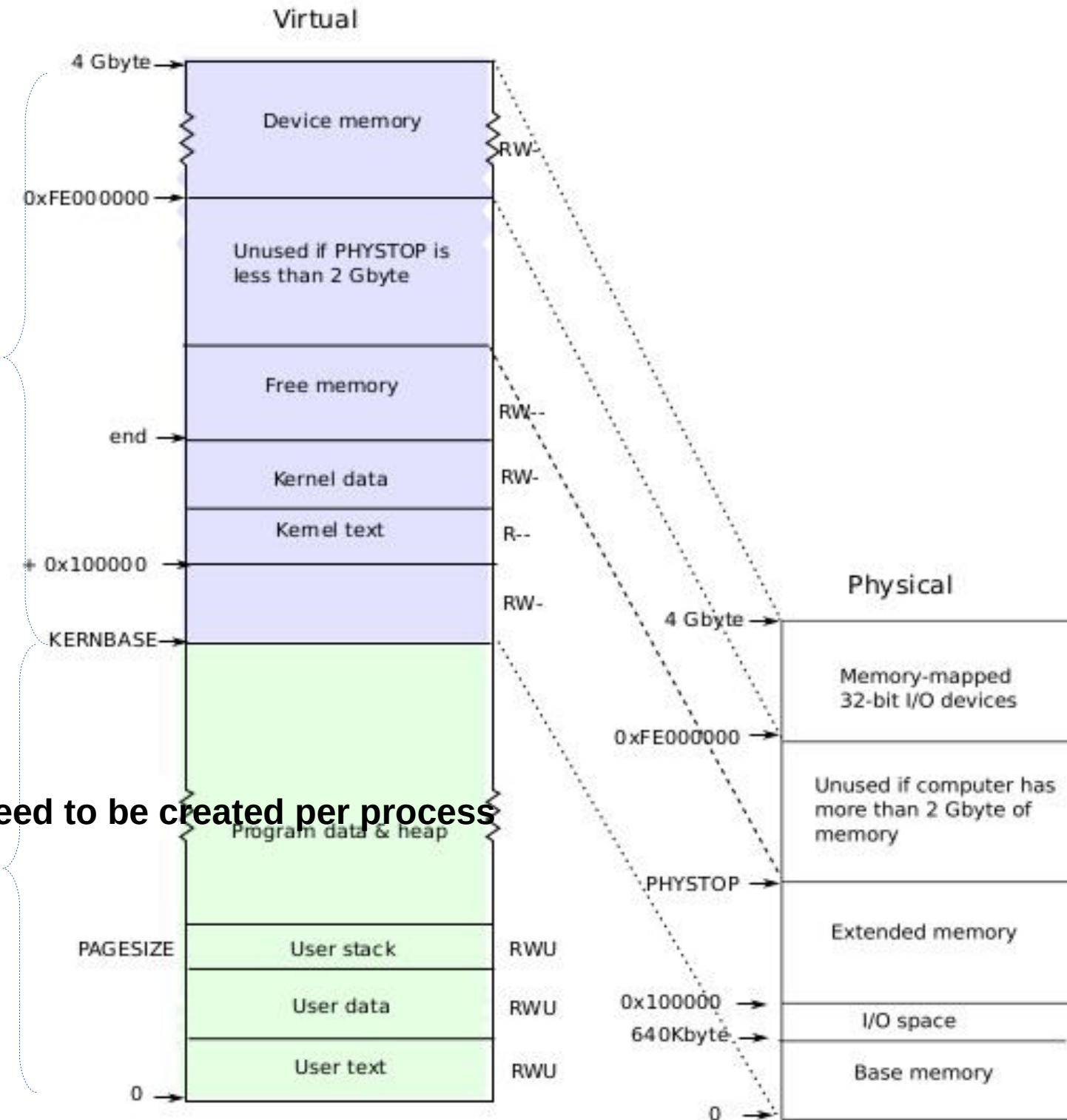
```
struct proc {  
    uint sz;                                // Size of process memory (bytes)  
    pde_t* pgdir;                            // Page table  
    char *kstack;                            // Bottom of kernel stack for this process  
    enum procstate state;      // Process state  
    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  
    void *chan;                             // If non-zero, sleeping on chan  
    int killed;                            // If non-zero, have been killed  
    struct file *ofile[NOFILE]; // Open files  
    struct inode *cwd;                 // Current directory  
    char name[16];                           // Process name (debugging)  
};
```

Imp Concepts

struct proc diagram: Very imp!

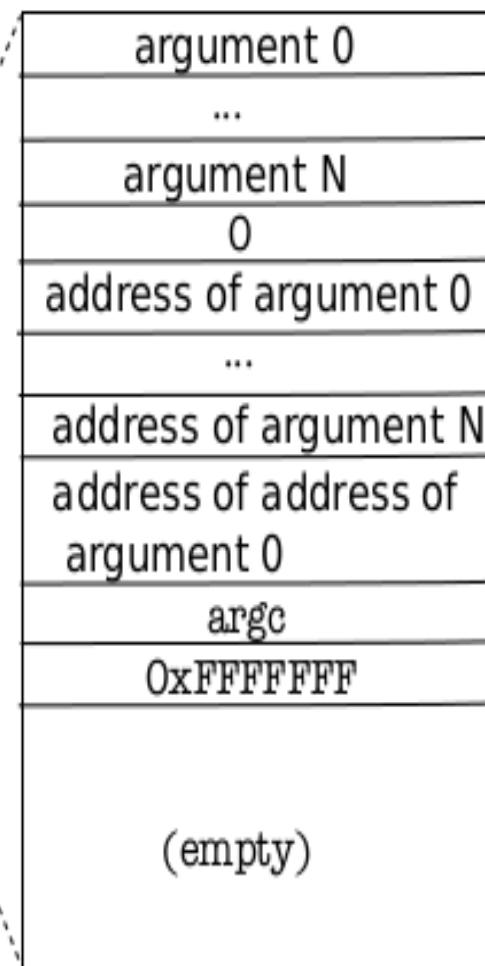
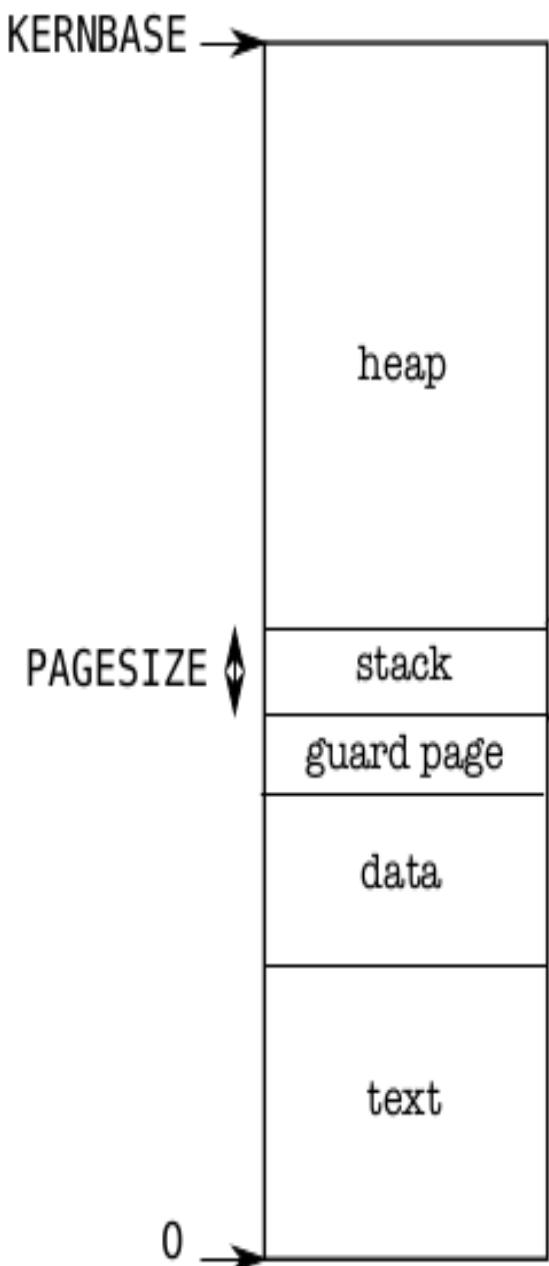


setupkvm()
does this mapping



These mappings need to be created per process

Memory Layout of a user process After exec()



Note the argc, argv on stack

stack is just one page

size of text and data is derived

nul-terminated string

argv[argc]

argv[0]

argv argument of main

argc argument of main

return PC for main

main() -> userinit()

Creating first process by hand

- **Code of the first process**
- **initcode.S and init.c**
- **init.c is compiled into “/init” file**
- **During make !**
- **Trick:**
- **Use initcode.S to “exec(“/init”)”**
- **And let exec() do rest of the job**
- **But before you do exec()**
- **Process must exist as if it was forked() and running**

main()->userinit()

Creating first process by hand

```
void  
userinit(void)  
{  
    struct proc *p;  
    extern char _binary_initcode_start[], _binary_initcode_size[];  
  
    // Abhijit: obtain proc 'p', with stack initialized  
    // and trapframe created and eip set to 'forkret'  
    p = allocproc();  
    // let's see what allocproc() does
```

First process creation

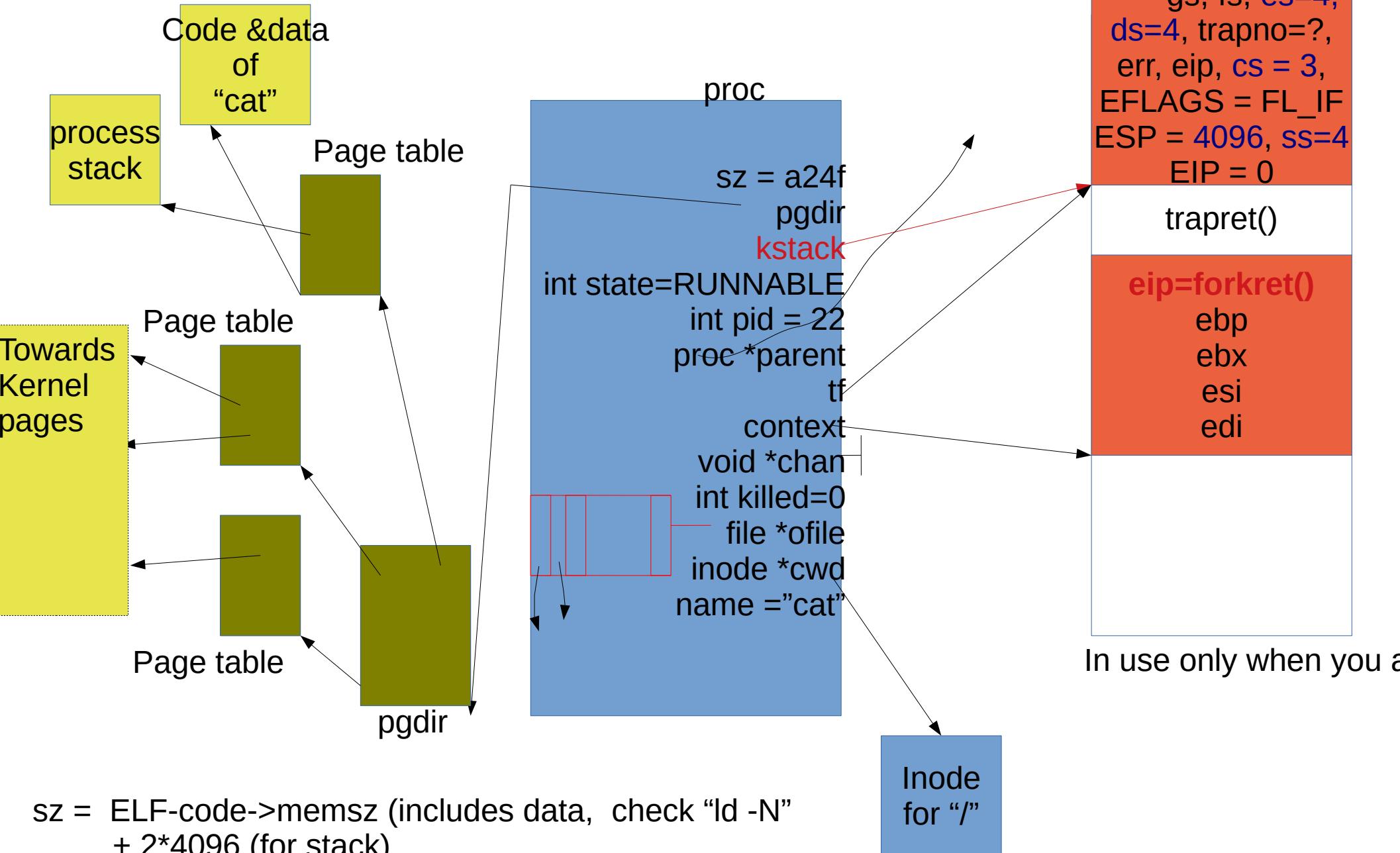
Let's revisit struct proc

```
// 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, waiting 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[NFILE]; // Open files, used by open(), read(),...
    struct inode *cwd; // Current directory, changed with "chdir()"
    char name[16];    // Process name (for debugging)
};
```

struct proc diagram



allocproc()

```
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC];
p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;
}
```

found:

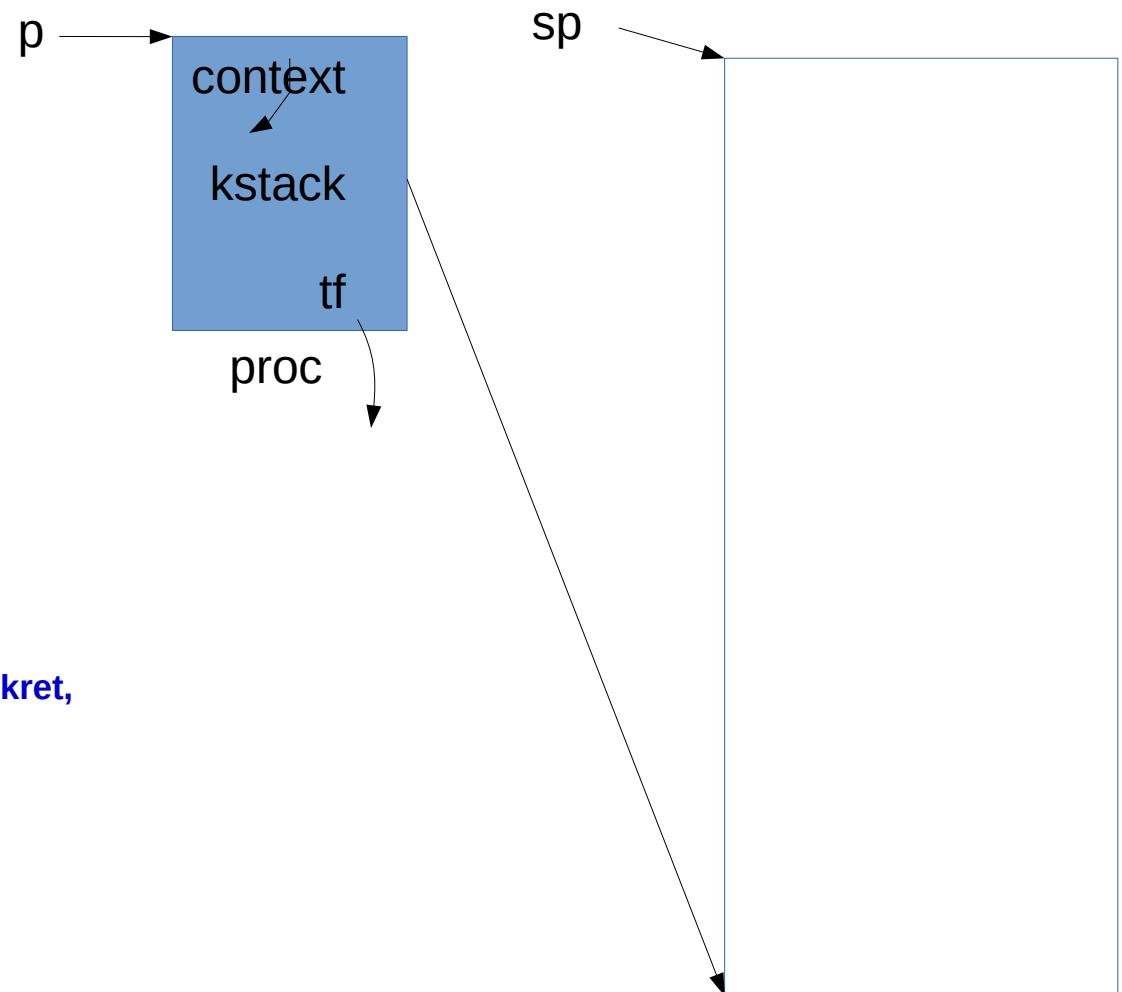
p->state = EMBRYO;

p->pid = nextpid++;

release(&ptable.lock);

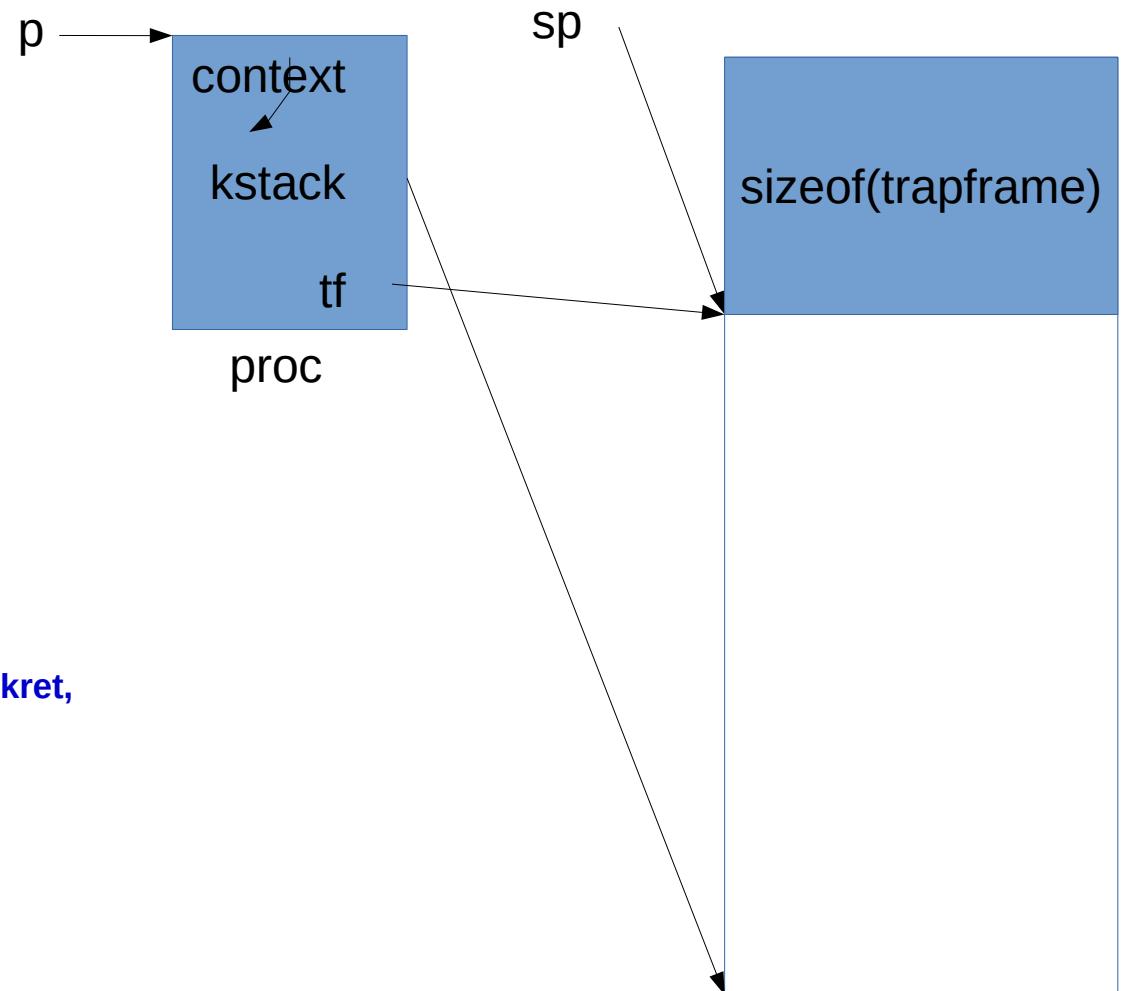
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



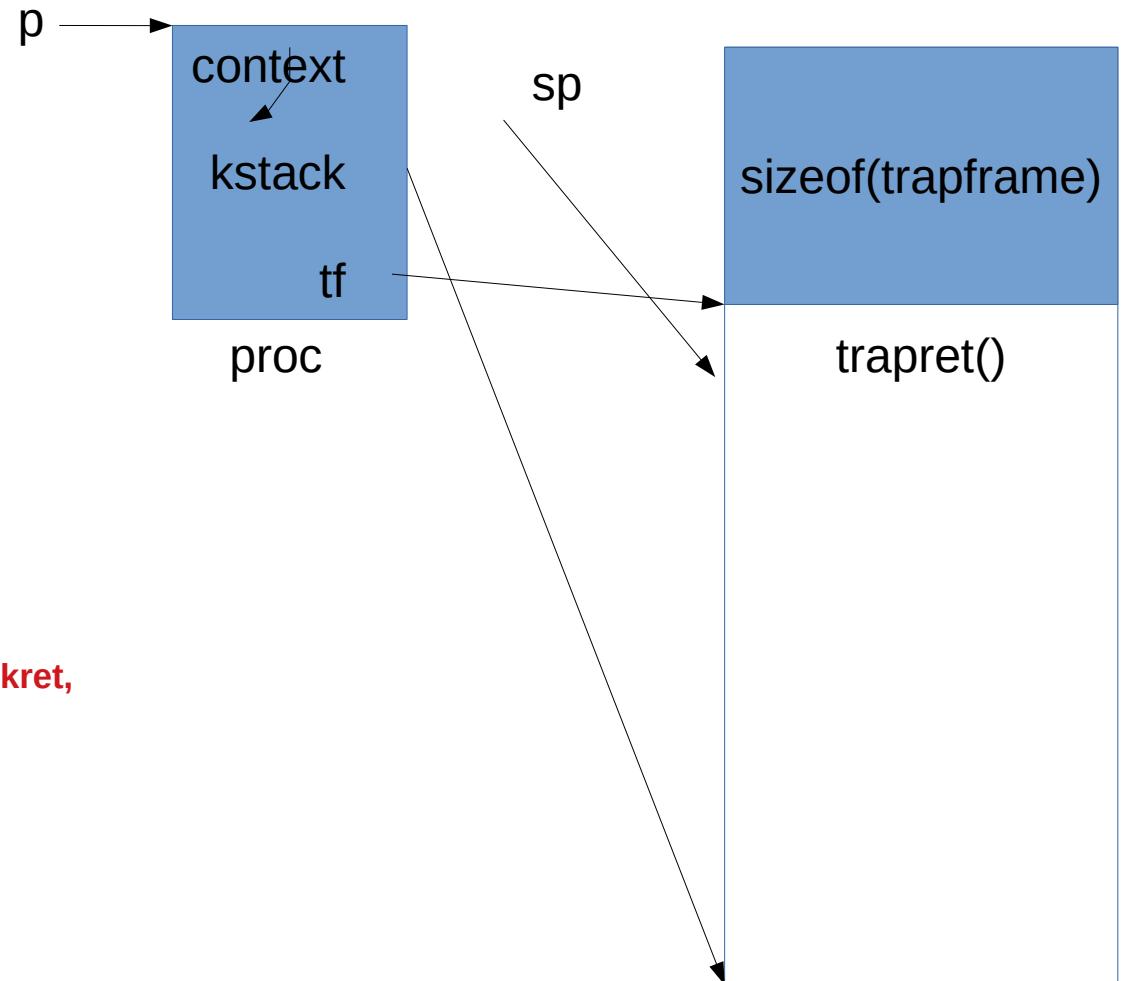
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
  
// Abhijit KSTACKSIZE = PGSIZE  
  
// Leave room for trap frame.  
  
sp -= sizeof *p->tf;  
  
p->tf = (struct trapframe*)sp;  
  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
  
sp -= 4;  
  
*(uint*)sp = (uint)trapret;  
  
sp -= sizeof *p->context;  
  
p->context = (struct context*)sp;  
  
memset(p->context, 0, sizeof *p->context);  
  
p->context->eip = (uint)forkret;
```



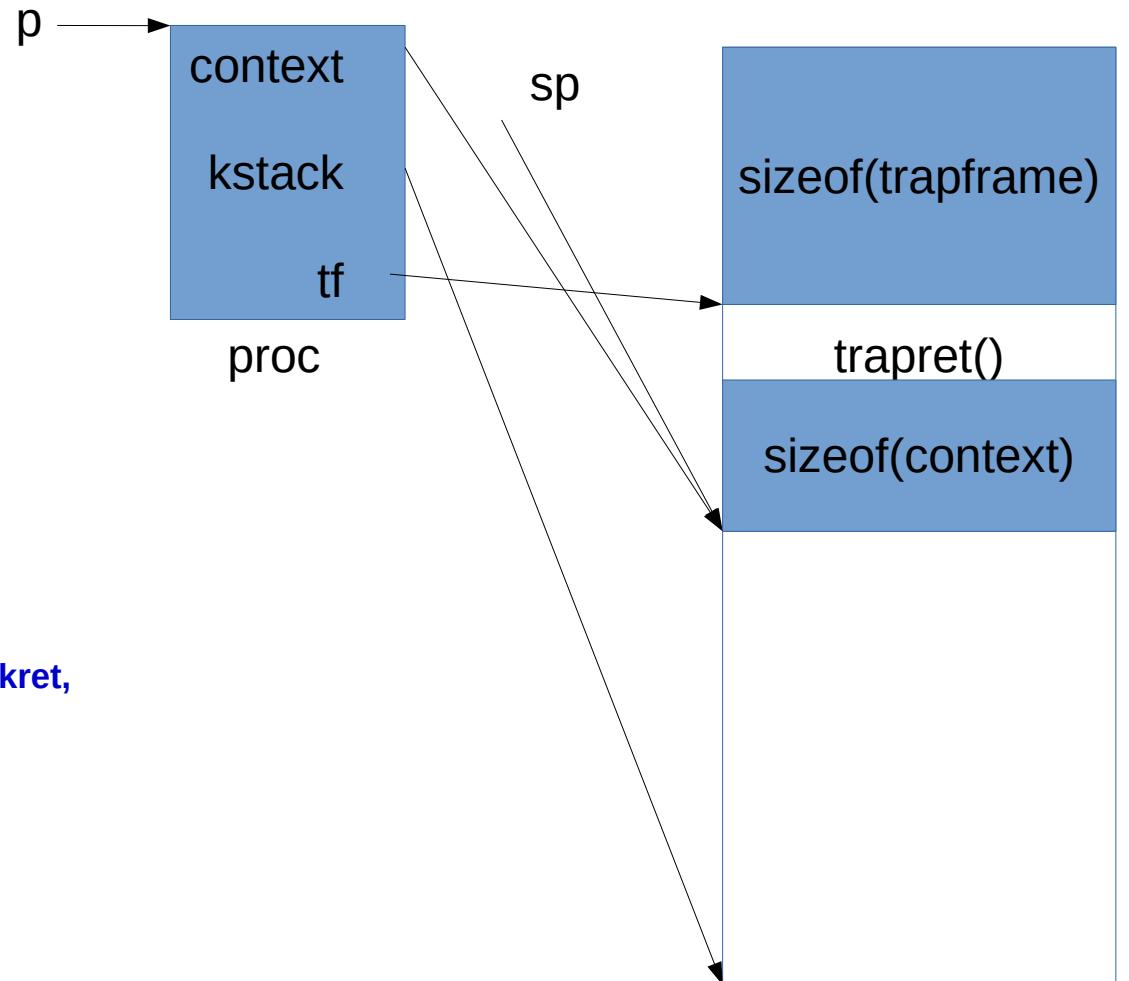
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
  
// Leave room for trap frame.  
  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
  
sp -= sizeof *p->context;  
  
p->context = (struct context*)sp;  
  
memset(p->context, 0, sizeof *p->context);  
  
p->context->eip = (uint)forkret;
```



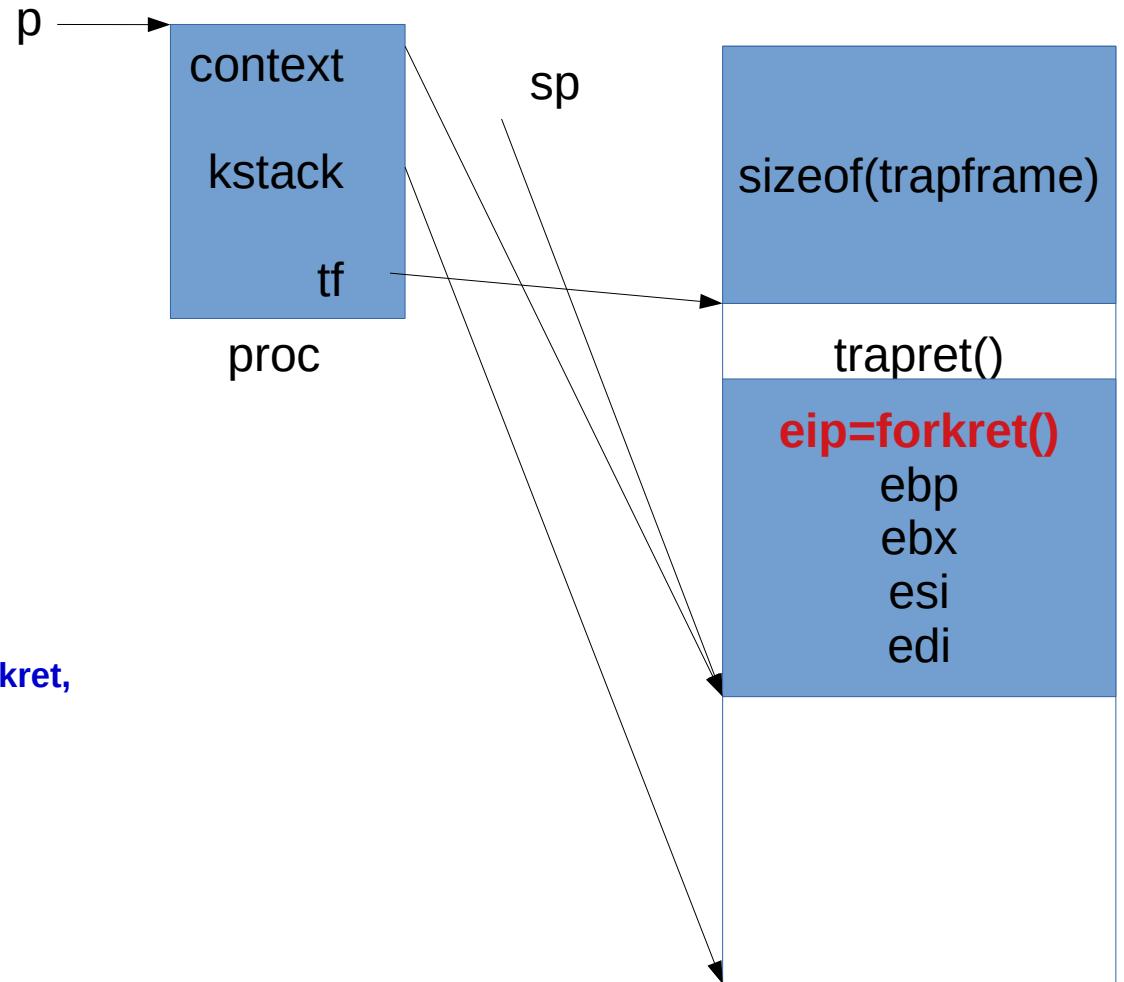
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



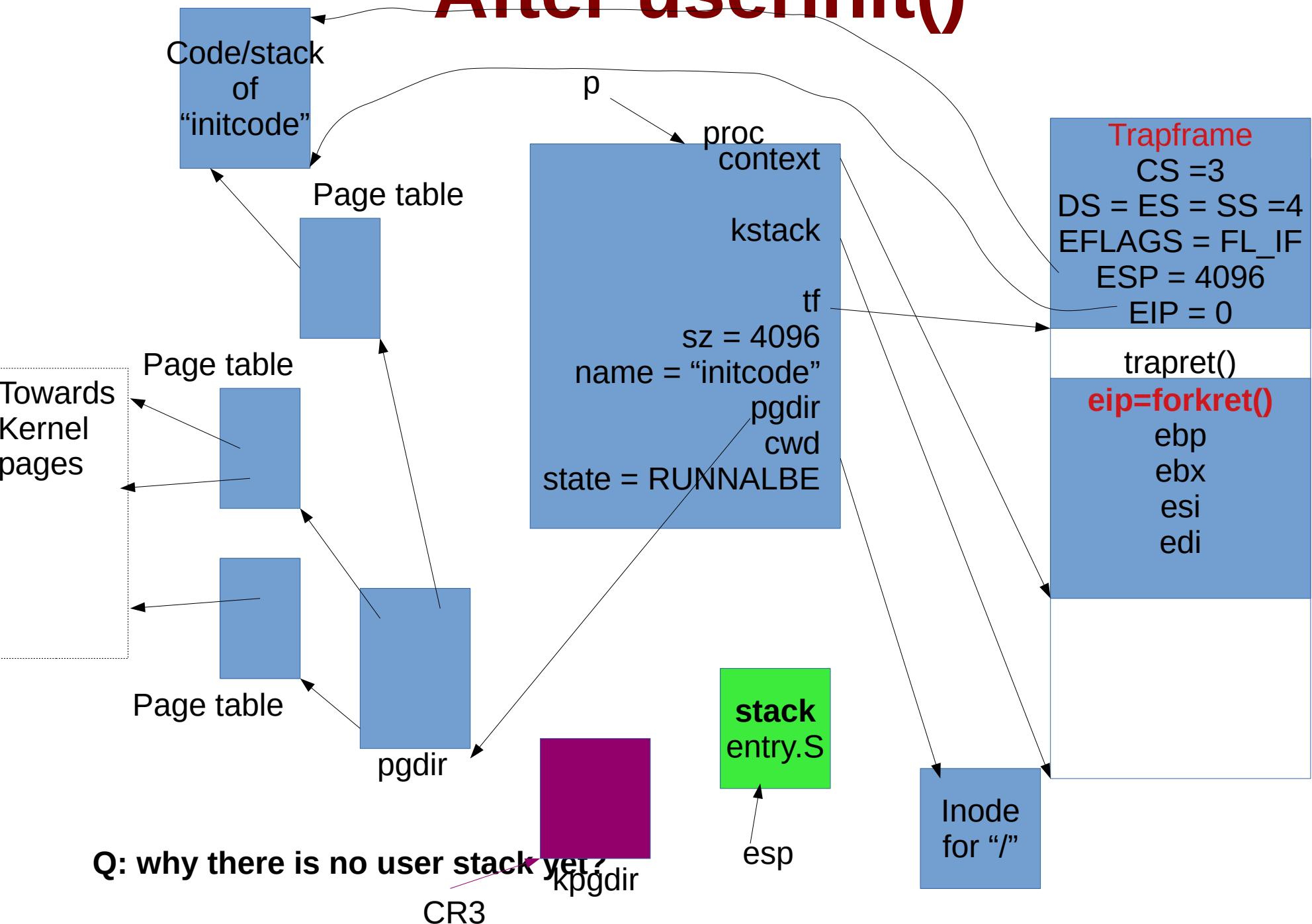
Next in userinit()

```
initproc = p;
if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of
memory");

inituvm(p->pgdir,
        _binary_initcode_start,
        (int)_binary_initcode_size);
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) |
DPL_USER;
p->tf->ds = (SEG_UDATA << 3) |
DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;

p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0; // beginning of
initcode.S
safestrcpy(p->name, "initcode",
sizeof(p->name));
p->cwd = namei("/");
acquire(&phtable.lock);
p->state = RUNNABLE;
release(&phtable.lock);
```

After userinit()



main() -> mpmain()

```
static void  
mpmain(void)  
{  
    cprintf("cpu%d: starting %d\n",  
    cpuid(), cpuid());  
    idtinit(); // load idt register  
    xchg(&(mycpu()->started), 1); //  
    tell startothers() we're up  
    scheduler(); // start running  
    processes  
}
```

- Load IDT register
- Copy from idt[] array into IDTR
- Call scheduler()
- One process has already been made runnable
- Let's enter scheduler now

Before reading scheduler(): Note

- The esp is still pointing to the stack which was allocated in entry.S !
- this is the kernel only stack
- Not the per process kernel stack.
- CR3 points to kpgdir
- Struct cpu[] has been setup up already
- apicid – in mpinit()
- segdesc gdt – in seginit()
- started – in mpmain()
- Fields in cpu[] not yet set
- context * scheduler --> will be setup in sched()
- taskstate ts --> large structure, only parts used in switchuvvm()
- ncli, intena --> used while locking
- proc *proc -> set during scheduler()

scheduler()

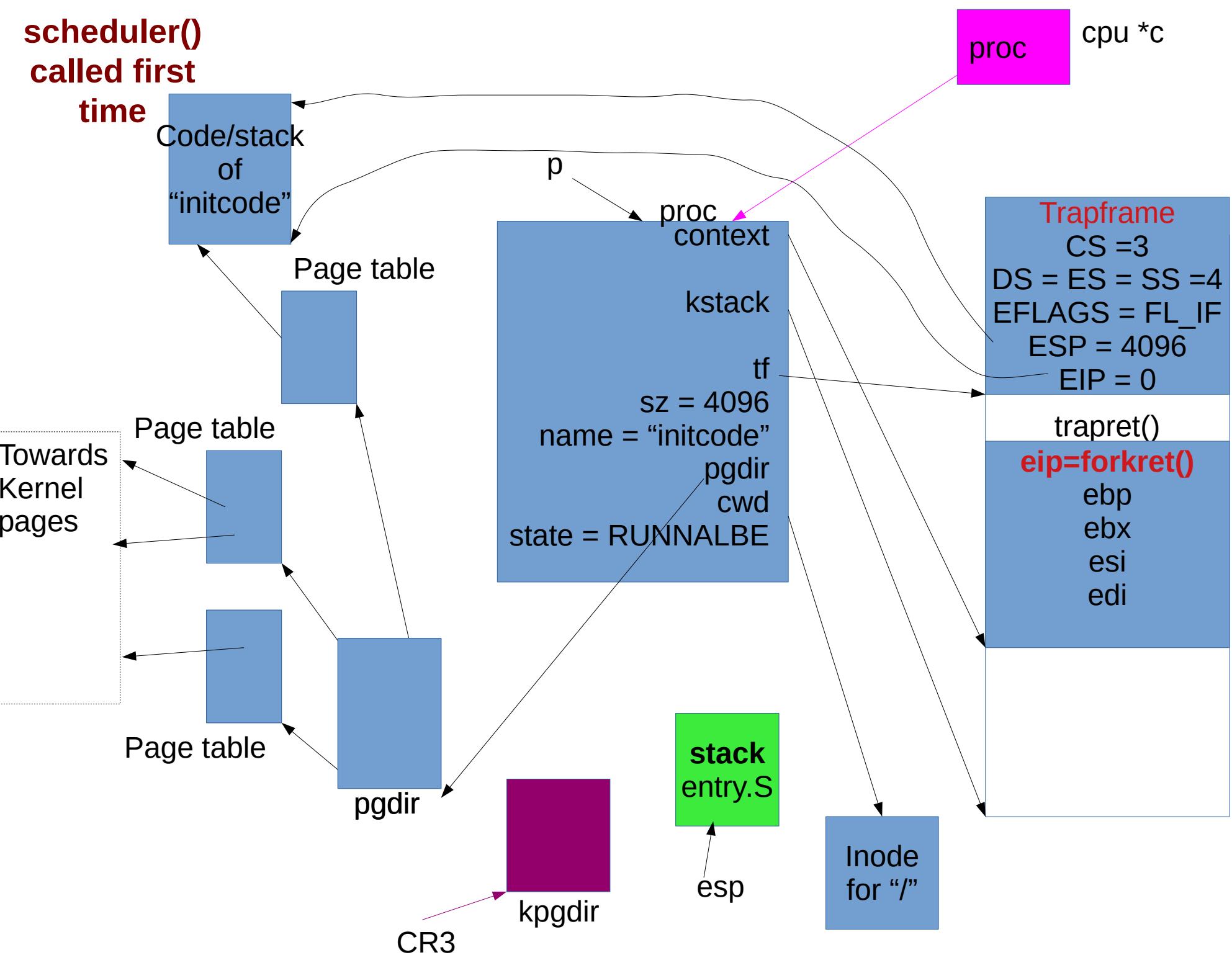
```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.

            c->proc = p;
        }
    }
}
```

scheduler()
called first

time

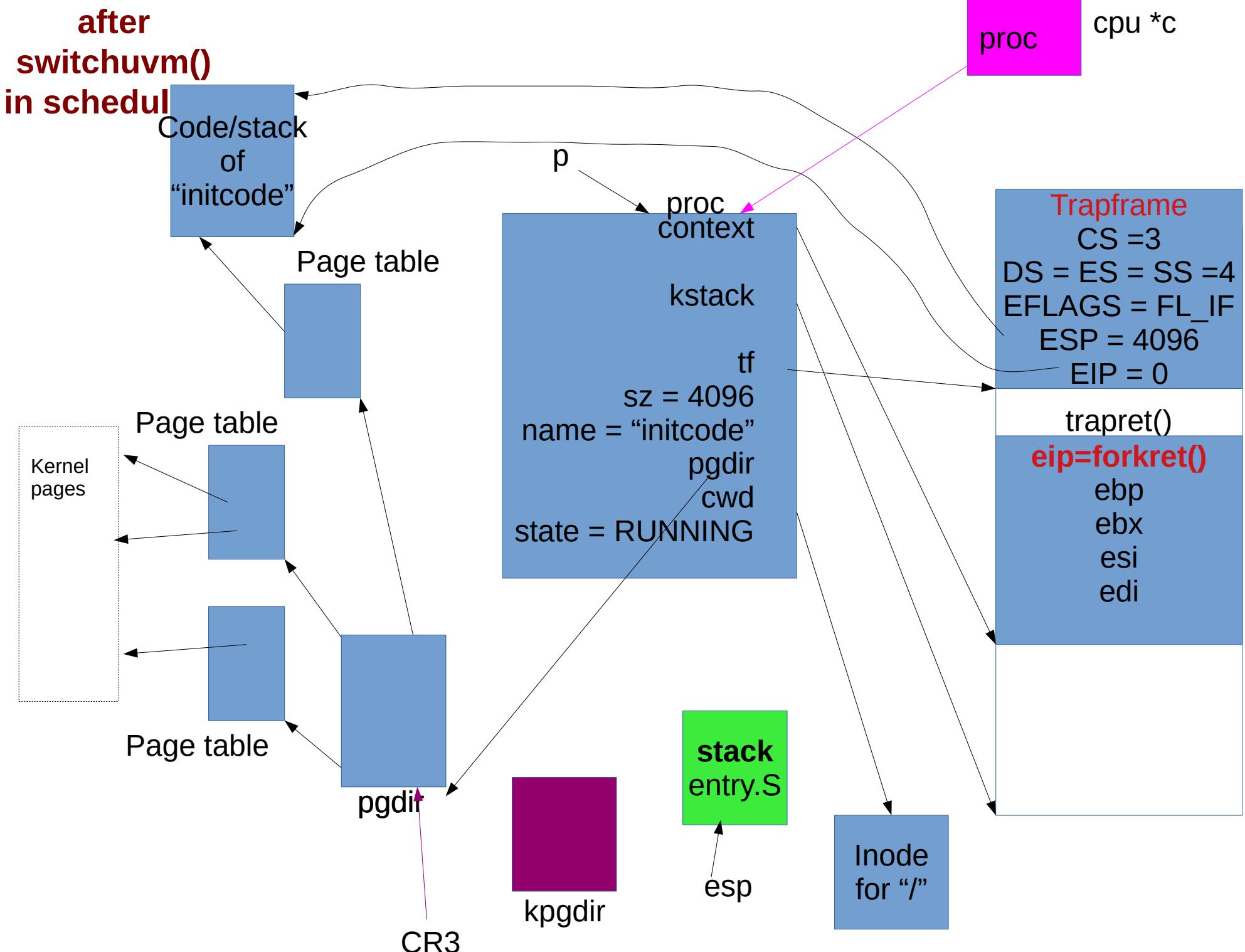


scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;

    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.

    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
```



scheduler()

```
acquire(&ptable.lock);

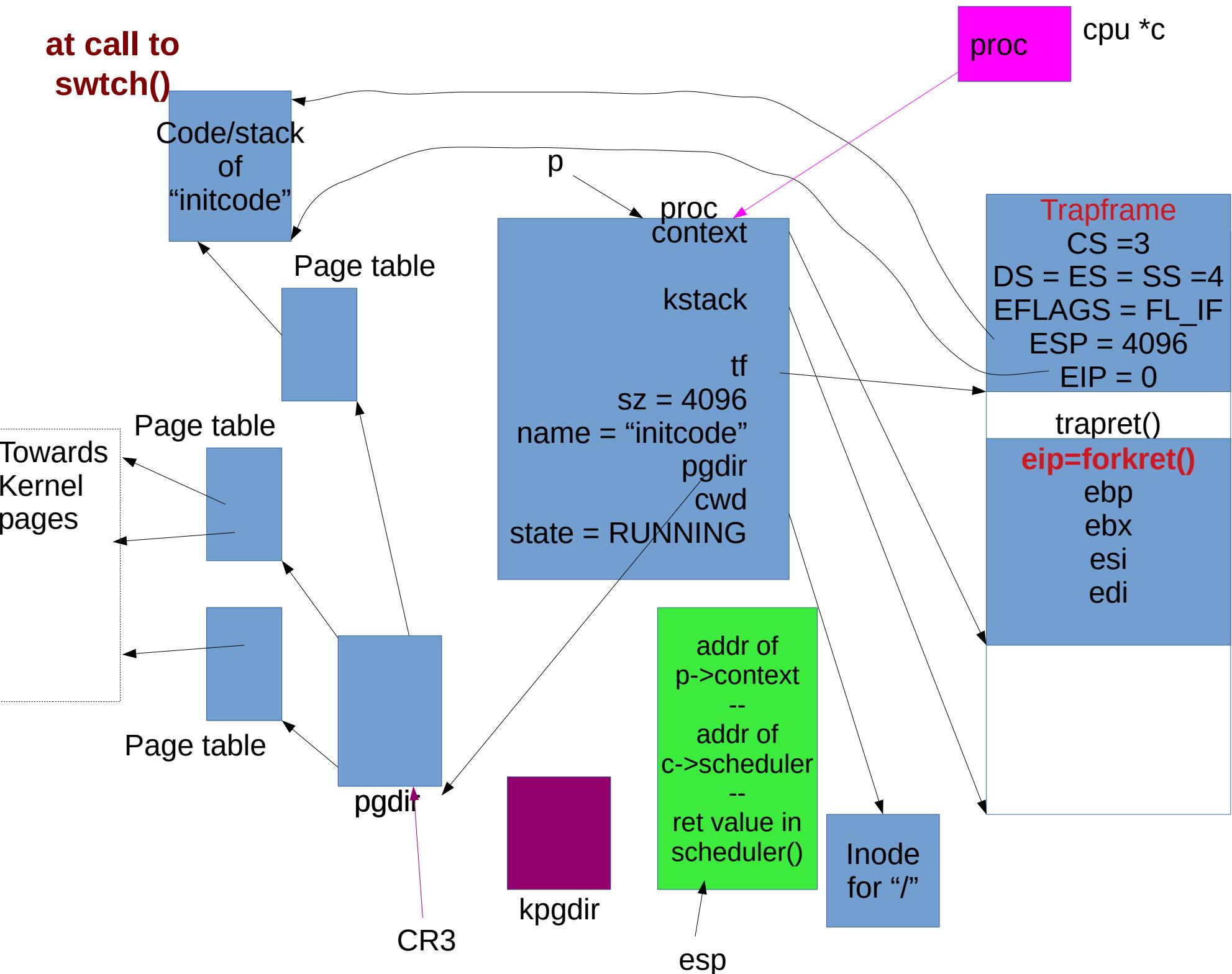
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;

    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.

    c->proc = p;
    switchuvm(p);
    p->state = RUNNING
    swtch(&(c->scheduler), p->context);

};
```

at call to
swtch()



swtch

swtch:

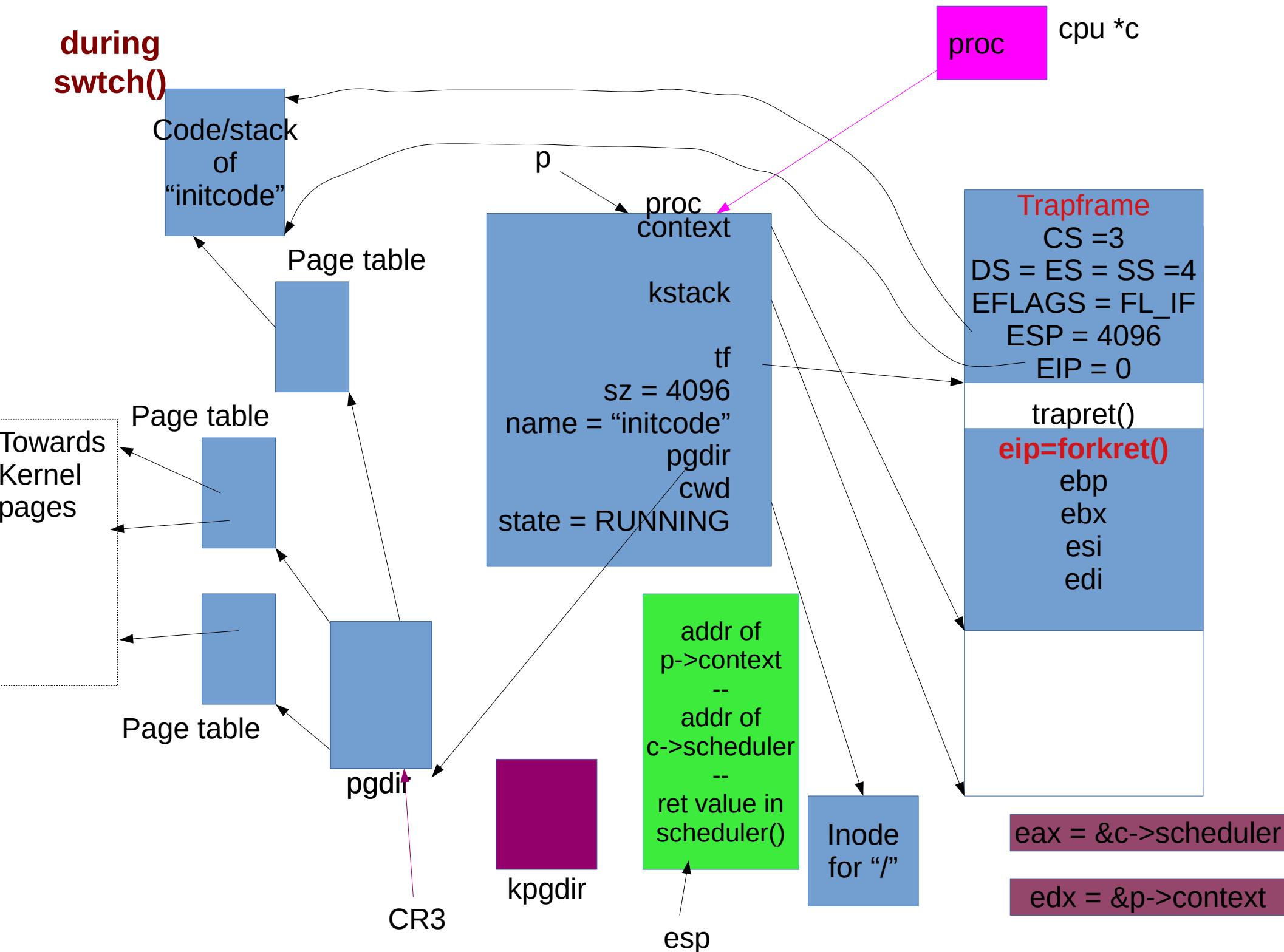
#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

**during
swtch()**



swtch

swtch:

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

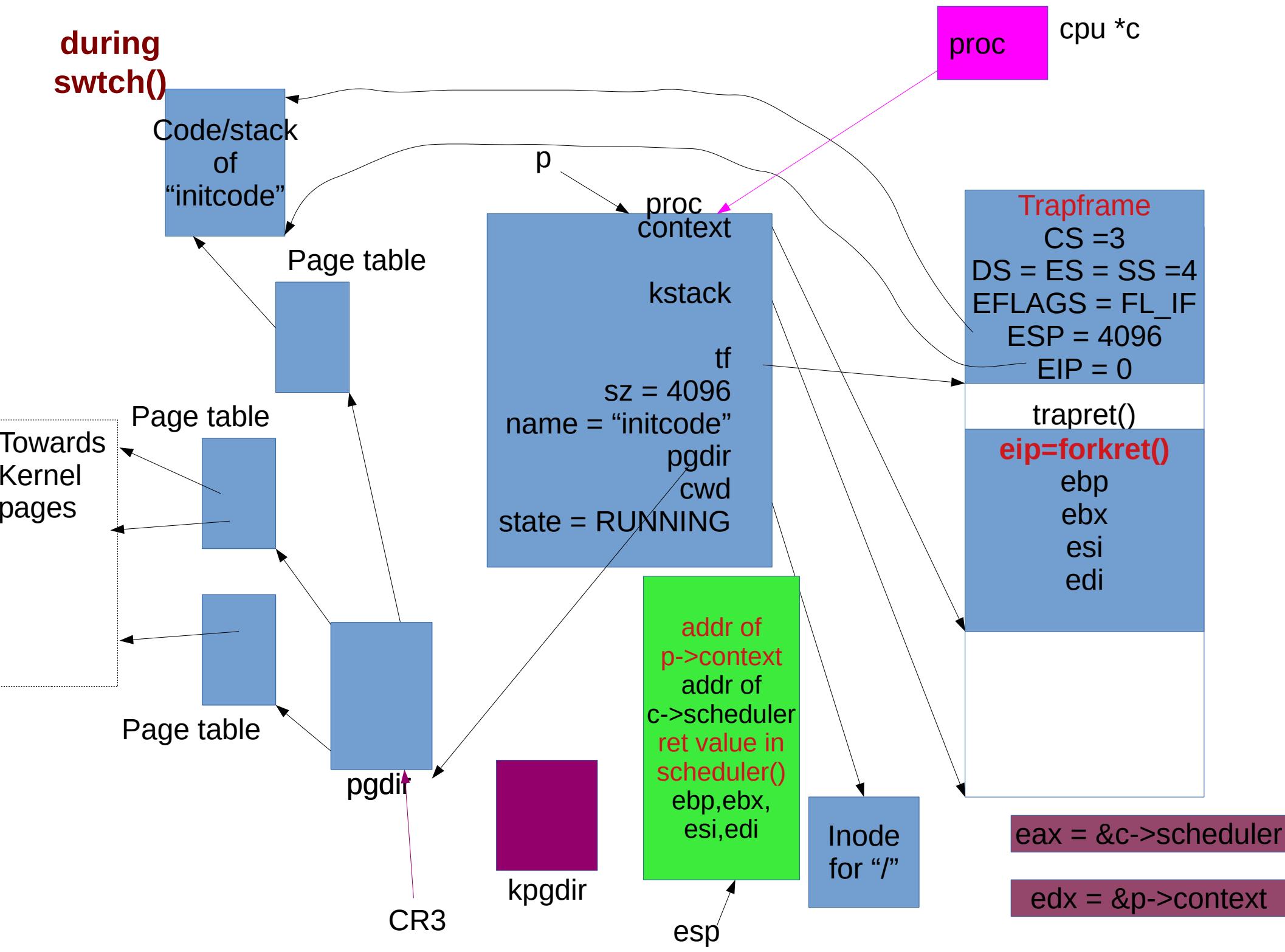
pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

**during
swtch()**



swtch

swtch:

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

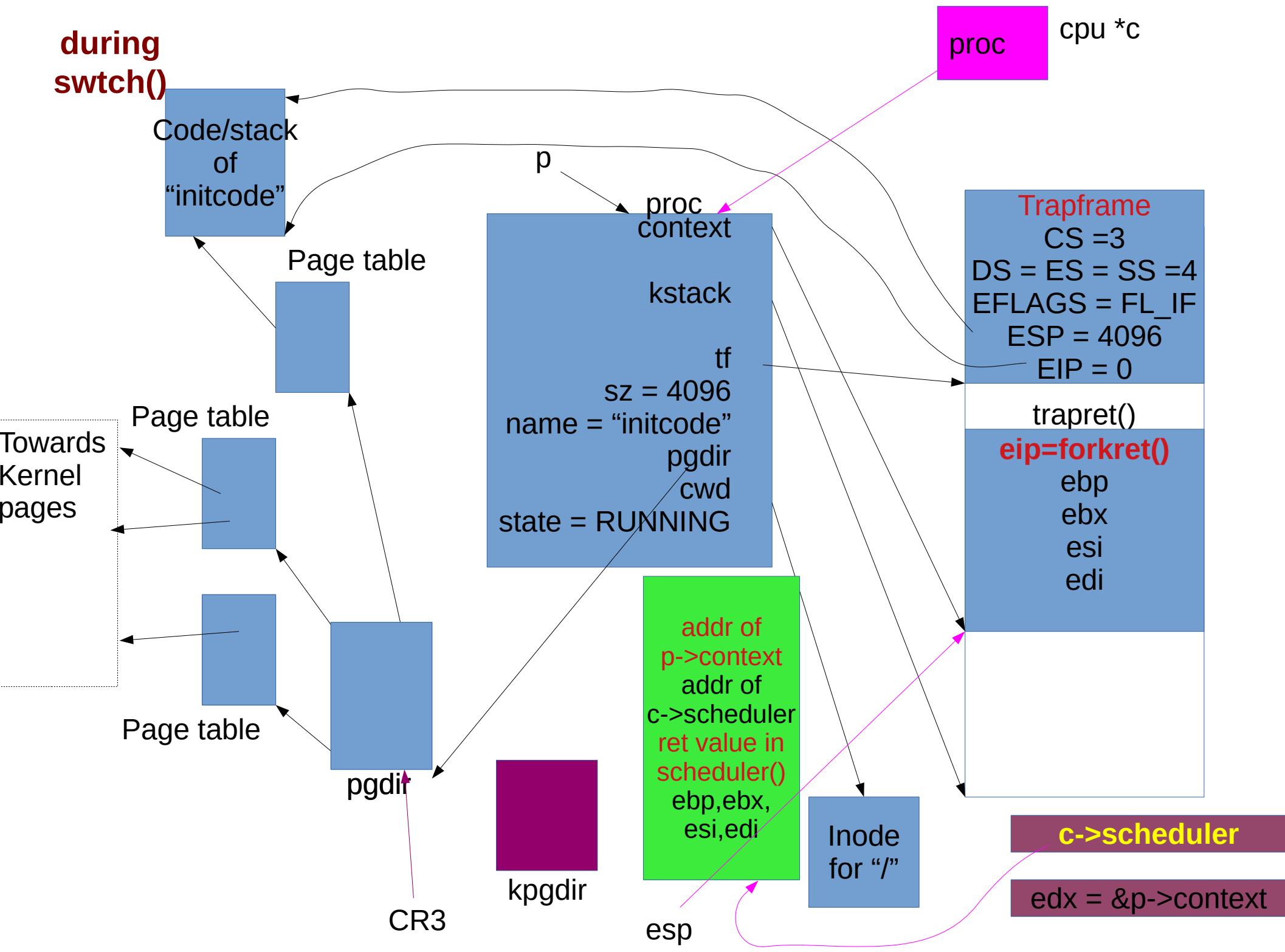
pushl %edi # Abhijit: esp = esp + 16

Switch stacks

movl %esp, (%eax) # Abhijit: *old = updated old stack

movl %edx, %esp # Abhijit: esp = new

**during
swtch()**



swtch

swtch:

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

Switch stacks

movl %esp, (%eax) # Abhijit: *old = updated old stack

movl %edx, %esp # Abhijit: esp = new

Load new callee-saved registers

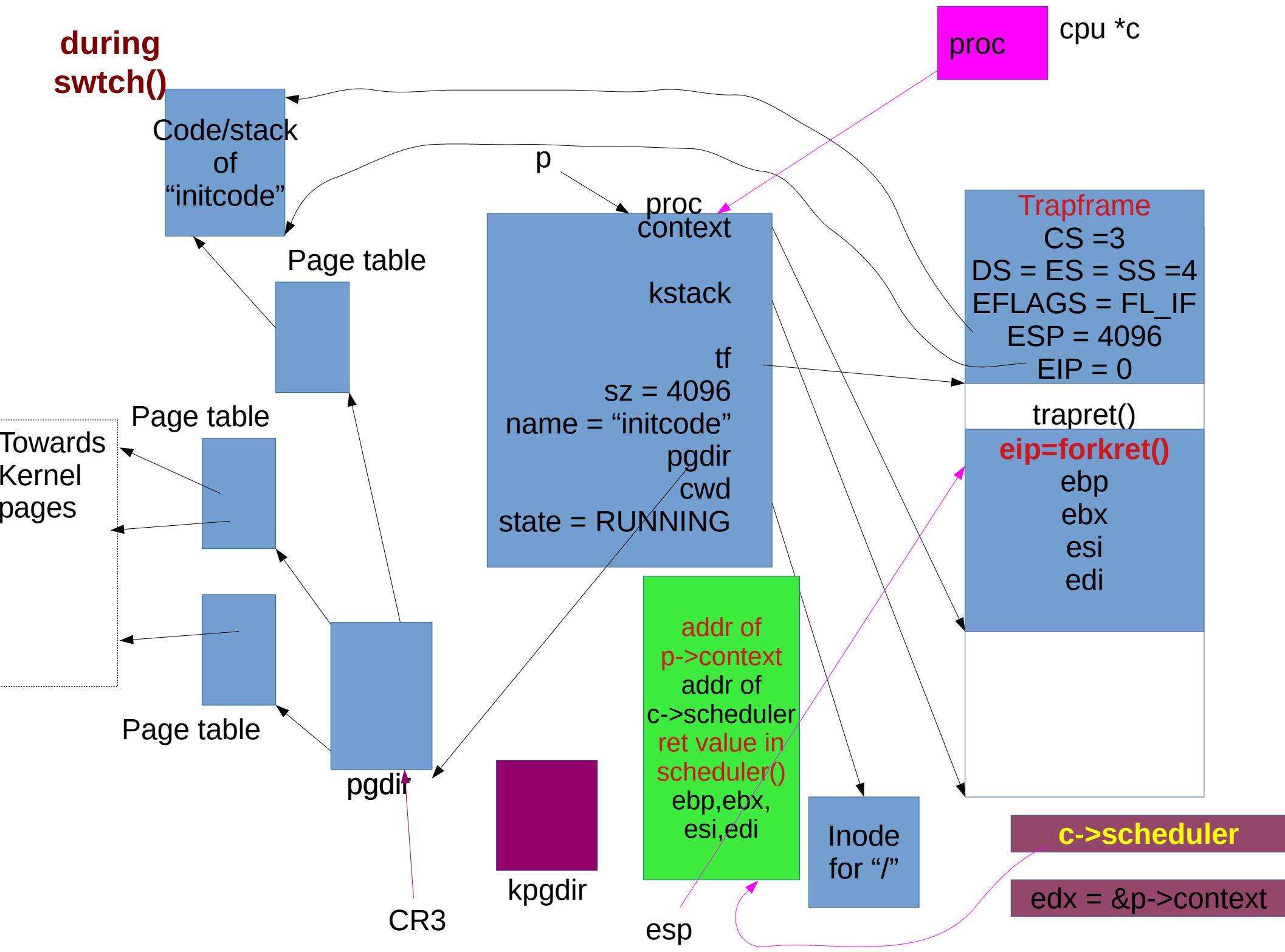
popl %edi

popl %esi

popl %ebx

popl %ebp # Abhijit: newesp = newesp - 16, context restored

**during
swtch()**



swtch:

Swtch

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

Switch stacks

movl %esp, (%eax) # Abhijit: *old = updated old stack

movl %edx, %esp # Abhijit: esp = new

Load new callee-saved registers

popl %edi

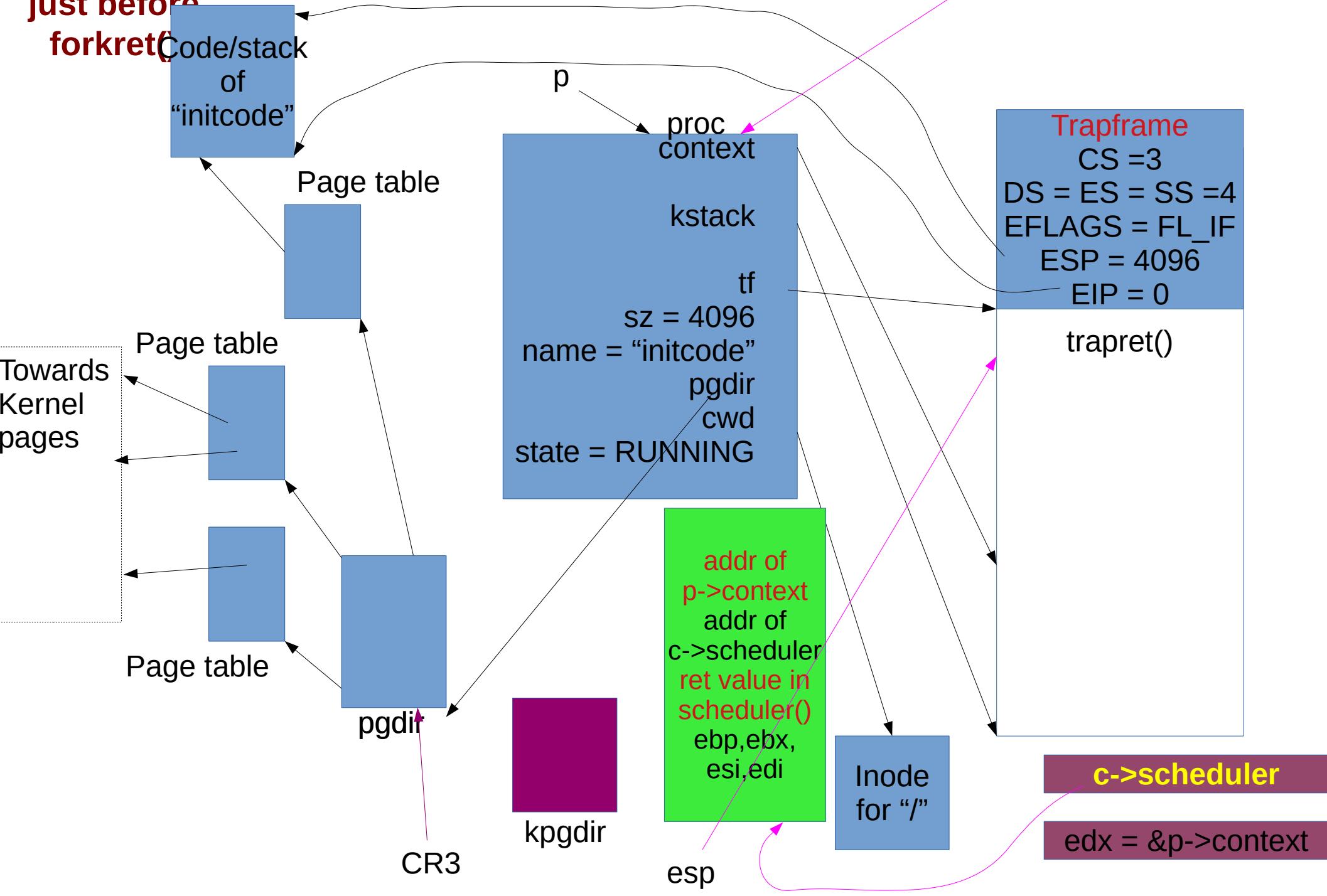
popl %esi

popl %ebx

popl %ebp # Abhijit: newesp = newesp - 16, context restored

ret # Abhijit: will pop from esp now -> function where to return

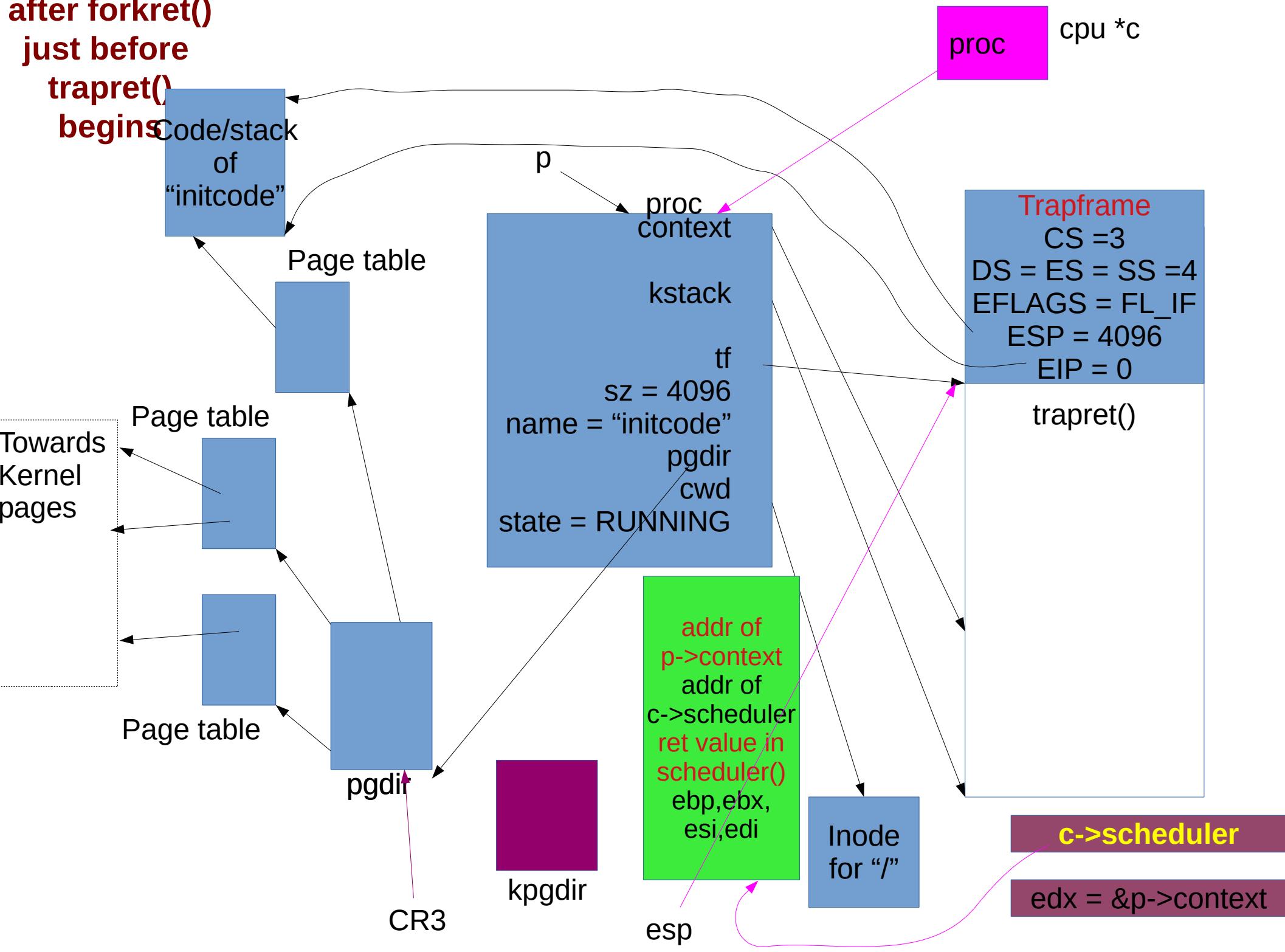
after "ret"
from swtch()
just before
`forkret()`



After swtch()

- Process is running in forkret()
- c->cscheduler has saved the old kernel stack
- with the context of p, return value in scheduler, ebp, ebx, esi, edi on stack
- remember {edi, esi, ebx, ebp, ret-value } = context
- The c->scheduler is pointing to old context
- CR3 is pointing to process pgdir

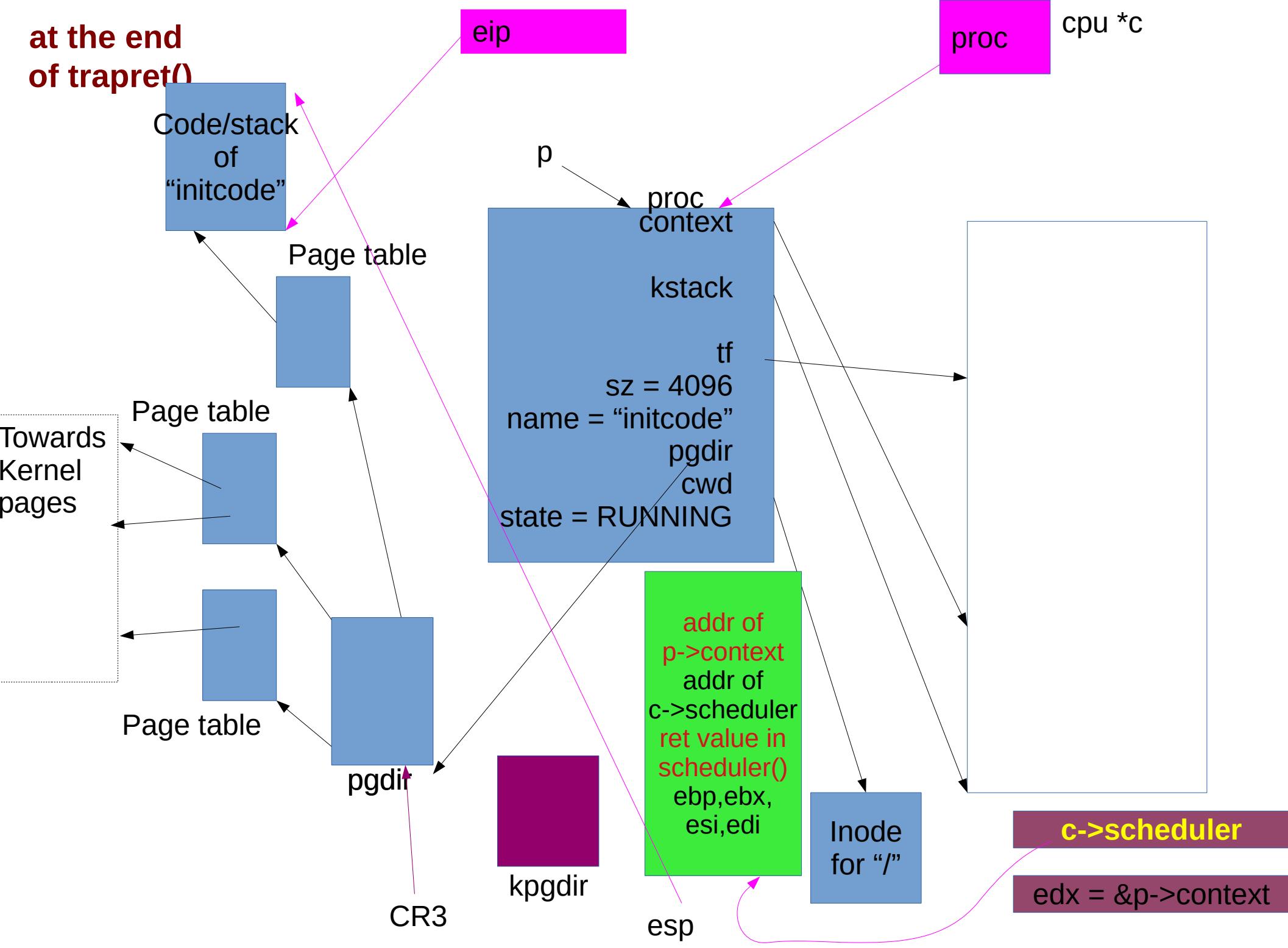
after forkret()
just before
trapret()
begins



After iret in trapret

- The CS, EIP, ESP will be changed
- to values already stored on trapframe
- this is done by iret
- Hence after this user code will run
- On user stack!
- Hence code of *initcode* will run now

at the end
of trapret()



initcode

```
# char init[] = "/init\0";
```

init:

```
.string "/init\0"
```

```
# char *argv[] = { init, 0 };
```

```
.p2align 2
```

argv:

```
.long init
```

```
.long 0
```

start:

```
pushl $argv
```

```
pushl $init
```

```
pushl $0 // where caller pc  
would be
```

```
movl $SYS_exec, %eax
```

```
int $T_SYSCALL
```

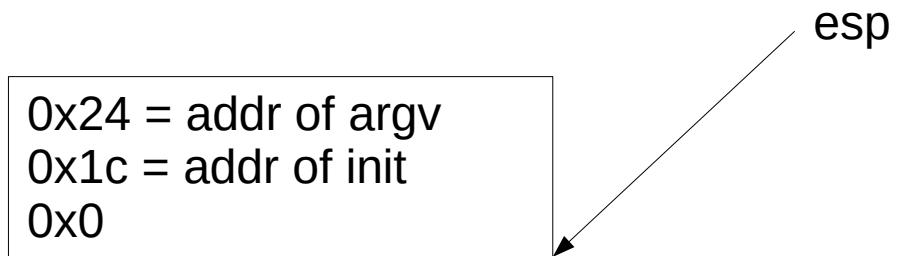
```
# for(;;) exit();
```

exit:

```
movl $SYS_exit, %eax
```

```
int $T_SYSCALL
```

```
jmp exit
```



00000000 <start>:

0:	68 24 00 00 00	push \$0x24
5:	68 1c 00 00 00	push \$0x1c
a:	6a 00	push \$0x0
c:	b8 07 00 00 00	mov \$0x7,%eax
11:	cd 40	int \$0x40

00000013 <exit>:

13:	b8 02 00 00 00	mov \$0x2,%eax
18:	cd 40	int \$0x40
1a:	eb f7	jmp 13 <exit>

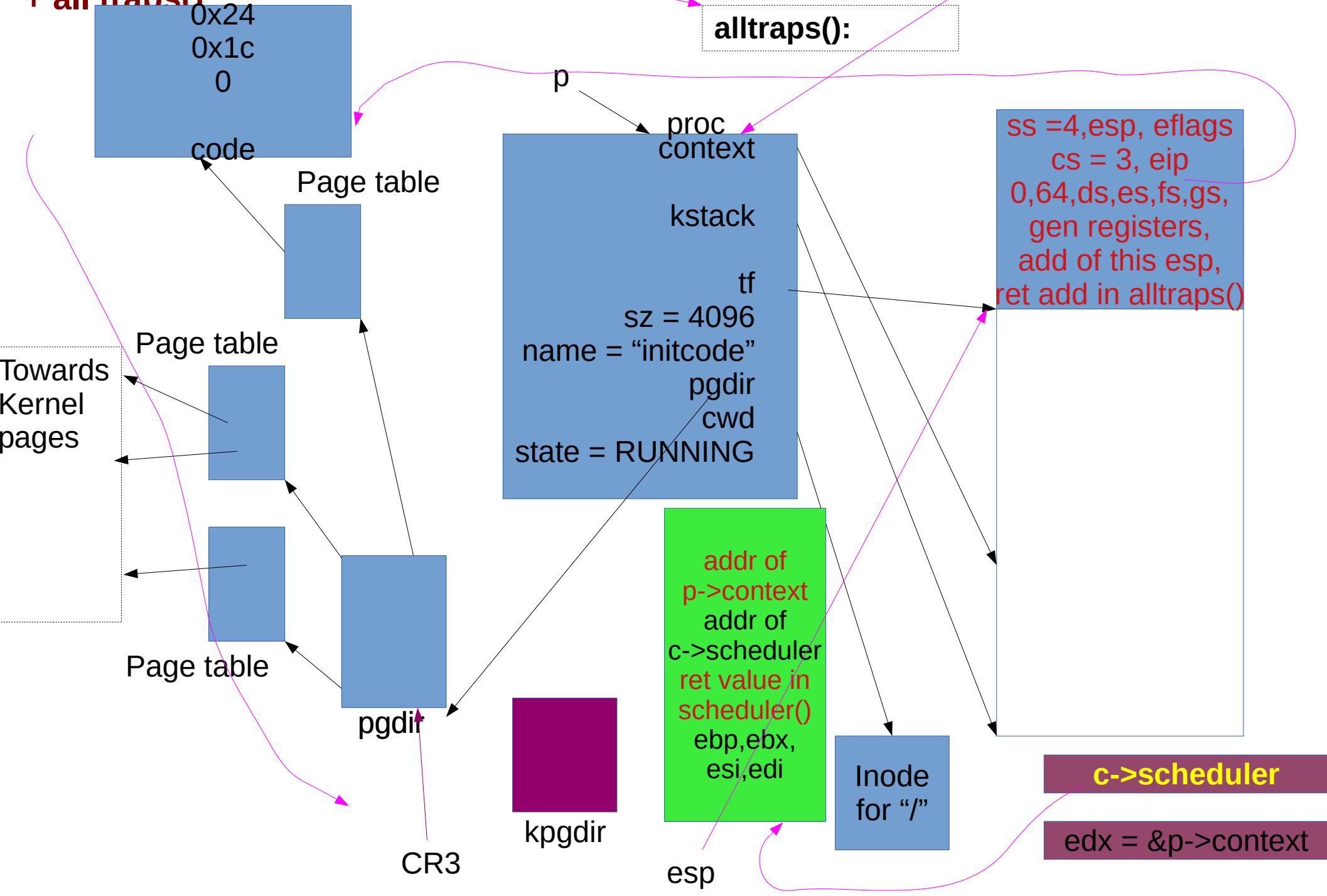
0000001c <init>:

"/init\0"

00000024 <argv>:

1c 00
00 00

on sys_exec()
+ all traps()



Understanding fork() and exec()

**First, revising some concepts already learnt
then code of fork(), exec()**

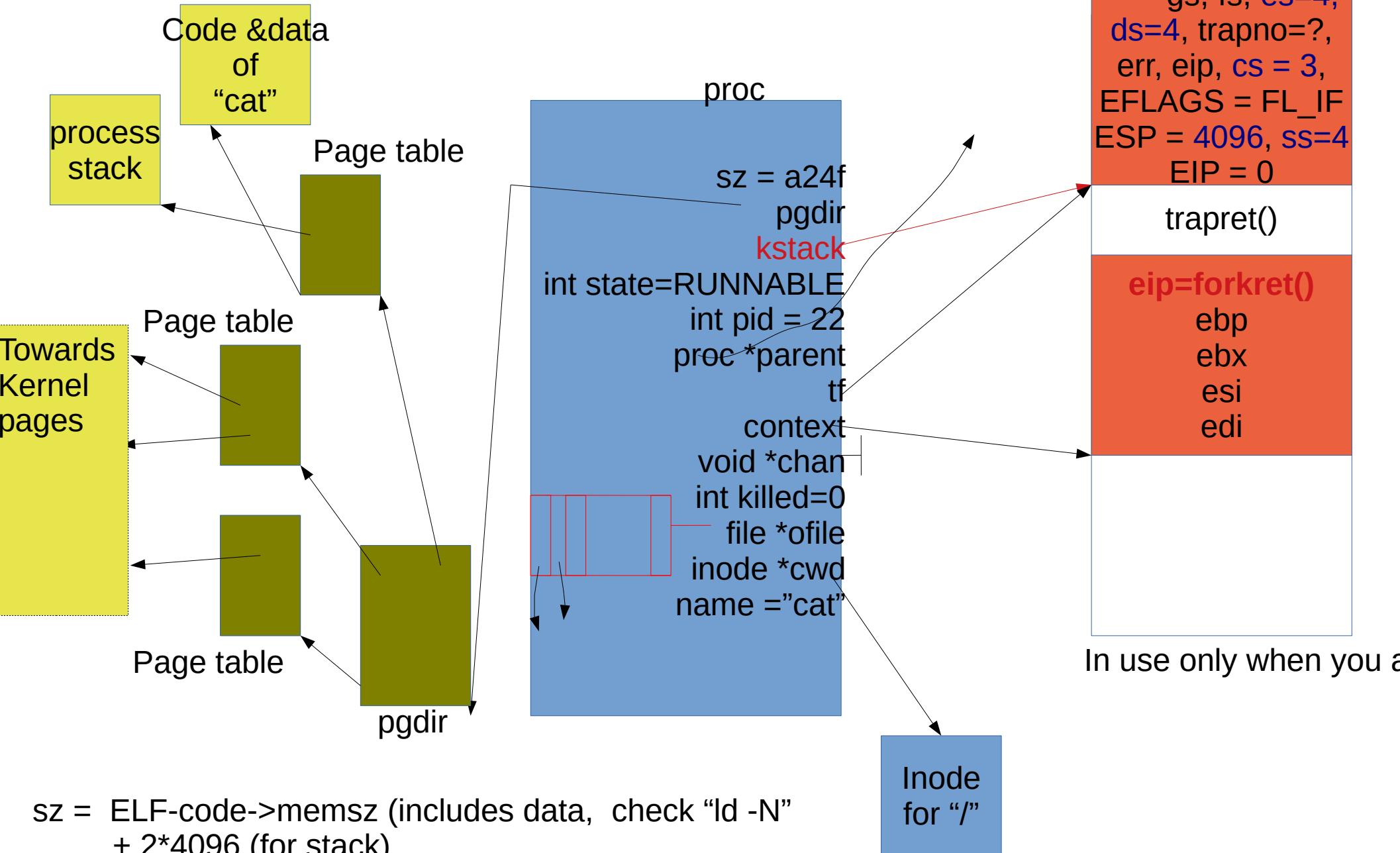
First process creation

Let's revisit 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, waiting 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[NFILE];    // Open files, used by open(), read(),...
    struct inode *cwd;            // Current directory, changed with "chdir()"
    char name[16];                // Process name (for debugging)
};
```

struct proc diagram



fork()/exec() are syscalls. On every syscall this happens

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- 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, during idtinit()
 - Remember vectors.S – gives jump locations for each interrupt
 - “int 64” ->jump to 64th entry in vector table
- 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
  -

```
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
```

## alltraps:

- 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 stack

```
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){

```

## trap()

- 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
  - `UatINTR()`
  - If Timer
    - Call `yield()` -- calls `sched()`
    - If process was killed (how is that done?)
      - Call `exit()`!

# when trap() returns

```
#Back in alptraps
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,

# understanding fork()

- What should fork do?
- Create a copy of the existing process
- child is same as parent, except pid, parent-child relation, return value (pid or 0)
- Please go through every member of struct proc, understand it's meaning to appreciate what fork() should do
- create a struct proc, and
- duplicate pages, page directory, sz, state, trapframe, context, ofile (and files!), cwd, name
- modify: pid, parent, trapframe, state

# understanding fork()

```
int
sys_fork(void)
{
 return fork();

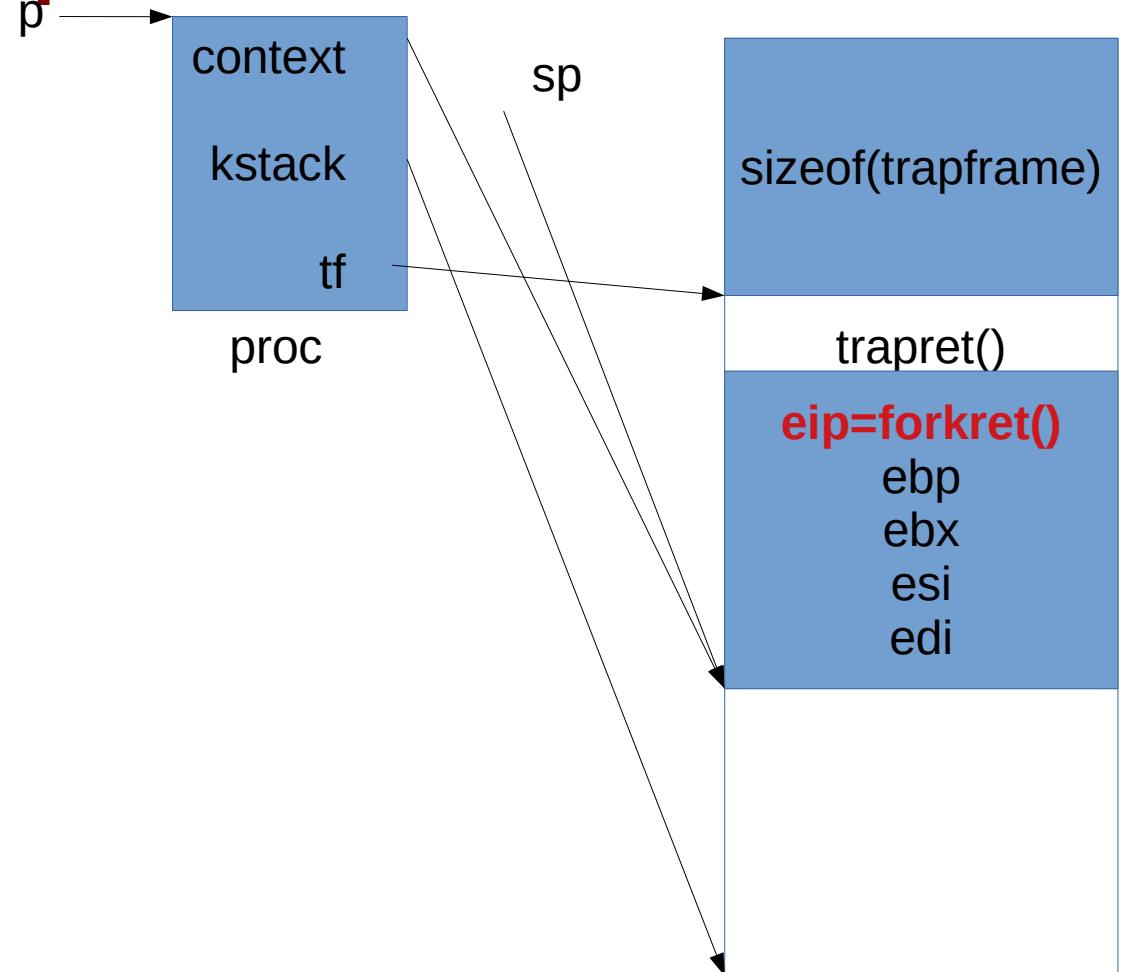
}

int
fork(void)
{
 int i, pid;
 struct proc *np;
 struct proc *curproc = myproc();

 // Allocate process.
 if((np = allocproc()) == 0){
 return -1;
 }
}
```

# after allocproc()

-- we studied this -- same as creation of first process



# understanding fork()

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
 kfree(np->kstack);
 np->kstack = 0;
 np->state = UNUSED;
 return -1;
}

np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
```

- copy the pages, page tables, page directory
- no copy on write here!
- Rewind if operation of copyuvm() fails
- copy size
- set parent of child
- copy trapframe (structure is copied)

# understanding fork()->copyuvm()

```
pde_t*

copyuvm(pde_t *pgdir, uint sz)
{
 pde_t *d; pte_t *pte; uint pa, i, flags;
 char *mem;

 if((d = setupkvm()) == 0)
 return 0;

 for(i = 0; i < sz; i += PGSIZE){
 if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
 panic("copyuvm: pte should exist");

 if(!(*pte & PTE_P))
 panic("copyuvm: page not present");

 pa = PTE_ADDR(*pte);

 flags = PTE_FLAGS(*pte);

 if((mem = kalloc()) == 0)
 goto bad;

 memmove(mem, (char*)P2V(pa), PGSIZE);

 if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
 kfree(mem);

 goto bad;
 }
 }

 return d;

bad:

 freevm(d);

 return 0;
}
```

- Map kernel pages
- for every page in parent's VM address space
- allocate a PTE for child
- set flags
- copy data
- map pages in child's page directory/tables

# understanding fork()

```
np->tf->eax = 0;
for(i = 0; i < NOFILE; i++)
 if(curproc->ofile[i])
 np->ofile[i] = filedup(curproc->ofile[i]);
 np->cwd = idup(curproc->cwd);
 safestrcpy(np->name, curproc->name,
 sizeof(curproc->name));
 pid = np->pid;
 acquire(&phtable.lock);
 np->state = RUNNABLE;
 release(&phtable.lock);
```

- set return value of child to 0
- eax contains return value, it's on TF
- copy each struct file
- copy current working dir inode
- copy name
- set pid of child
- set child “RUNNABLE”

# exec() - different prototype

- **int exec(char\*, char\*\*);**
- **usage: to print README and test.txt using “cat”**

```
int main(int argc, char *argv[])
{
 char *cmd = "/cat";
 char *argstr[4] = { "/cat", "README", "test.txt",
0};
 exec(cmd, argstr);
}
```

note: to really run this code in xv6, you need to make changes to Makefile. First, add this program to the CFILES variable:

```

int
sys_exec(void)
{
 char *path, *argv[MAXARG];
 int i;
 uint uargv, uarg;
 if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
 return -1;
 }
 memset(argv, 0, sizeof(argv));
 for(i=0;; i++){
 if(i >= NELEM(argv))
 return -1;
 if(fetchint(uargv+4*i, (int*)&uarg) < 0)
 return -1;
 if(uarg == 0){
 argv[i] = 0;
 break;
 }
 if(fetchstr(uarg, &argv[i]) < 0)
 return -1;
 }
 return exec(path, argv);
}

```

# sys\_exec()

- **argstr(n,), argint(n,)**
- **Fetch the n'th argument from *process stack* using p->tf->esp + offset**
- **Again: revise calling conventions**
- **0'th argument: name of executable file**
- **1<sup>st</sup> Argument: address of the array of arguments**
- **store in *uargv***

```

int sys_exec(void)
{
 char *path, *argv[MAXARG];
 int i; uint uargv, uarg;
 if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
 return -1;
 }
 memset(argv, 0, sizeof(argv));
 for(i=0;; i++){
 if(i >= NELEM(argv)) return -1;
 if(fetchint(uargv+4*i, (int*)&uarg) < 0)
 return -1;
 if(uarg == 0){
 argv[i] = 0; break;
 }
 if(fetchstr(uarg, &argv[i]) < 0)
 return -1;
 }
 return exec(path, argv);
}

```

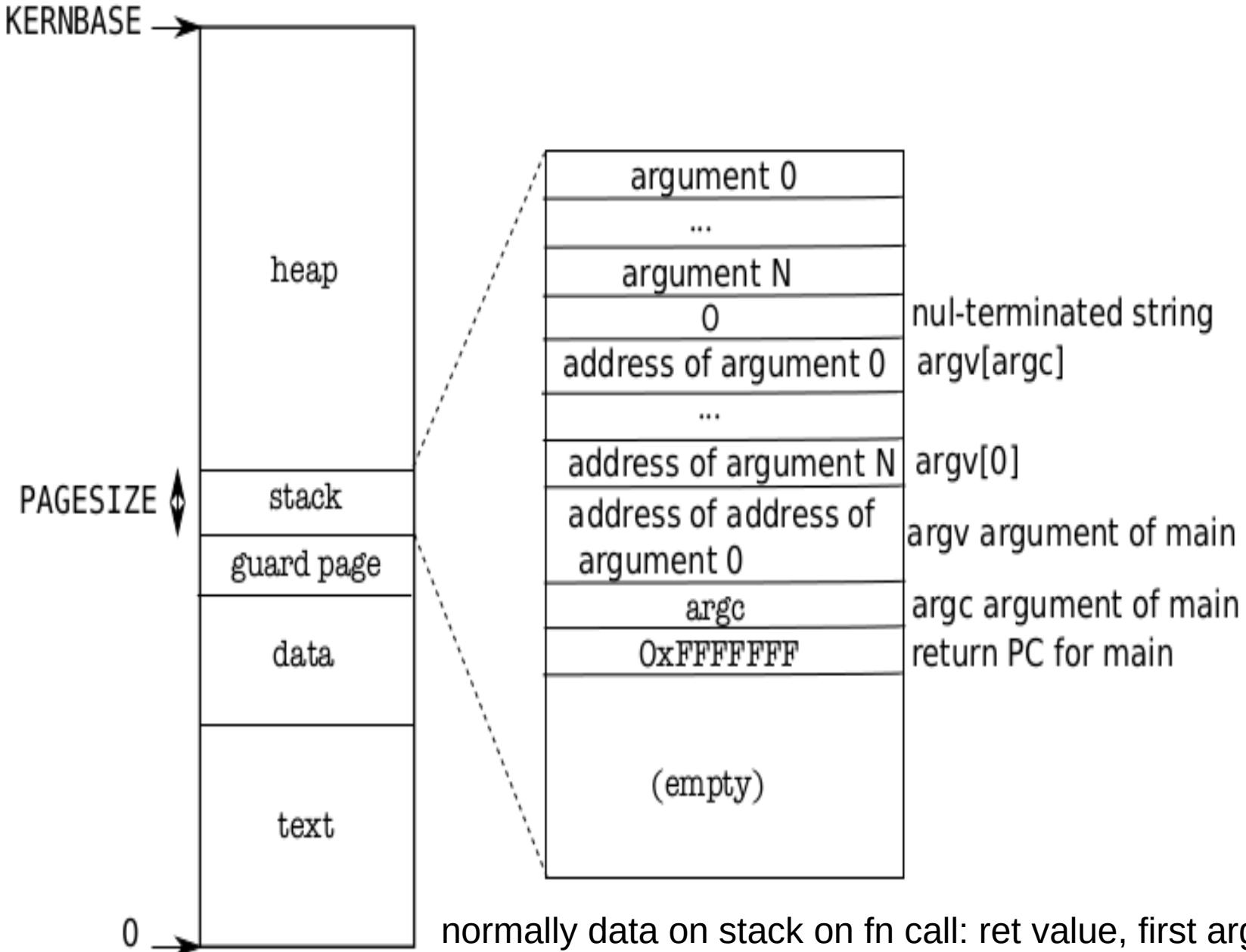
# sys\_exec()

- the local array argv[] (allocated on kernel stack, obviously) set to 0
- fetch every next argument from array of arguments
- Sets the address of argument in argv[1]
- call exec
- beware: mistake to assume that this exec() is the exec() called from user code! NO!

# What should exec() do?

- Remember, it came from fork()
- so proc & within it tf, context, kstack, pgdir-tables-pages, all exist.
- Code, stack pages exist, and mappings exist through proc->pgdir
- Hence
- read the ELF executable file (argv[0])
- create a new page dir – create mappings for kernel and user code+data; copy data from ELF to these pages (later discard old pagedir)
- Copy the argv onto the user stack – so that when new process starts it has its main(argc, argv[]) built
- set values of other fields in proc to start program correctly

User stack after  
call  
to exec()  
is over



normally data on stack on fn call: ret value, first arg, second arg, ...  
**main(int argc, char \*argv[])**  
argv[] is address of array of string; string itself is an address. Hence 2

# exec()

```
int
exec(char *path, char **argv)
{
...
uint argc, sz, sp, ustack[3+MAXARG+1];
...
if((ip = namei(path)) == 0){
 end_op();
 cprintf("exec: fail\n");
 return -1;
}
```

- **ustack**
- **used to build the arguments to be pushed on user-stack**
- **namei**
- **get the inode of the executable file**

# exec()

```
// Check ELF header
if(readi(ip, (char*)&elf, 0,
sizeof(elf)) != sizeof(elf))
 goto bad;
if(elf.magic != ELF_MAGIC)
 goto bad;

if((pgdir = setupkvm()) == 0)
 goto bad;
```

- **readi**
- **read ELF header**
- **setupkvm()**
- **creating a *new* page directory and mapping kernel pages**

# exec()

```
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
 if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
 goto bad;
 if(ph.type != ELF_PROG_LOAD)
 continue;
 if(ph.memsz < ph.filesz)
 goto bad;
 if(ph.vaddr + ph.memsz < ph.vaddr)
 goto bad;
 if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
 goto bad;
 if(ph.vaddr % PGSIZE != 0)
 goto bad;
 if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
 goto bad;
}
```

- Read ELF program headers from ELF file
- Map the code/data into pagedir-pagetable-pages
- Copy data from ELF file into the pages allocated

# exec()

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz +
2*PGSIZE)) == 0)
 goto bad;
clearpteu(pgdir, (char*)(sz -
2*PGSIZE));
sp = sz;
```

- Allocate 2 pages on top of proc->sz
- One page for stack
- one page for guard page
- Clear the valid flag on guard page

// Push argument strings, prepare rest of stack in  
ustack.

```
for(argc = 0; argv[argc]; argc++) {
 if(argc >= MAXARG)
 goto bad;

 sp = (sp - (strlen(argv[argc]) + 1)) & ~3;

 if(copyout(pgdir, sp, argv[argc], strlen(argv[argc])
+ 1) < 0)
 goto bad;

 ustack[3+argc] = sp;
}

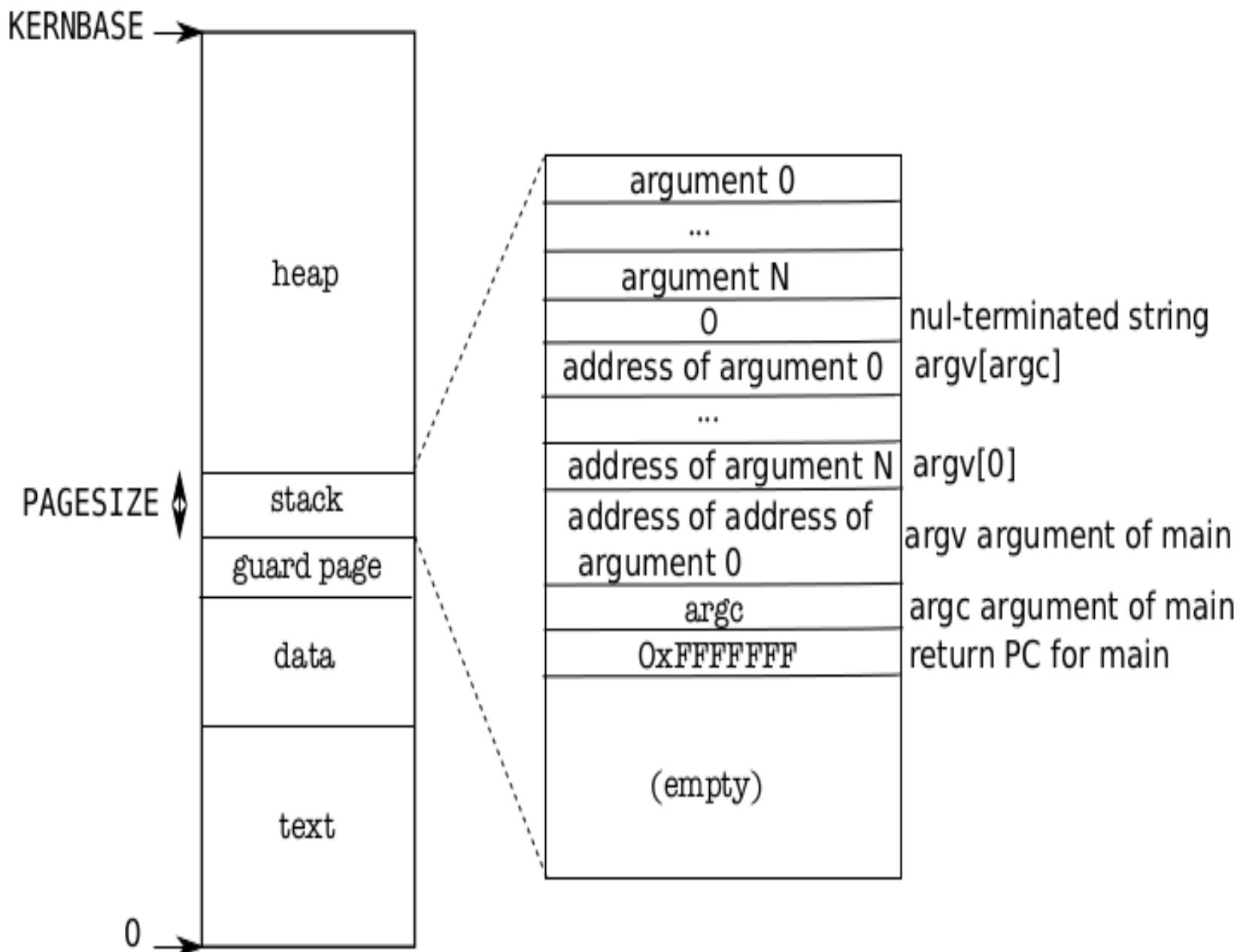
ustack[3+argc] = 0;
ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer
sp -= (3+argc+1) * 4;

if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
 goto bad;
```

# exec()

- For each entry in argv[]
- copy it on user-stack
- remember it's location on user stack in ustack
- add extra entries (to be copied to user stack) to ustack
- copy argc, argv pointer
- take sp to bottom
- copy ustack to user stack

This is what the code on ear



```
// Save program name for debugging.
```

```
for(last=s=path; *s; s++)
```

```
if(*s == '/')
```

```
last = s+1;
```

```
safestrcpy(curproc->name, last,
sizeof(curproc->name));
```

```
// Commit to the user image.
```

```
oldpgdir = curproc->pgdir;
```

```
curproc->pgdir = pgdir;
```

```
curproc->sz = sz;
```

```
curproc->tf->eip = elf.entry; // main
```

```
curproc->tf->esp = sp;
```

```
switchuvvm(curproc);
```

```
freevm(oldpgdir);
```

```
return 0;
```

# exec()

- copy name of new process in proc->name

- change to new page directory

- change new size

- tf->eip will be used when we return from exec() to jump to user code. Set to first instruction of code, given by elf.entry

- Set user stack pointer to "sp" (bottom of stack of arguments)

- Update TSS, change CR3 to newpagedir

- free old page dir

# return 0 from exec()?

- We know exec() does not return !
- This was exec() function !
- Returns to sys\_exec()
- sys\_exec() also returns , where?
- Remember we are still in kernel code, running on kernel stack. p->kstack has the trapframe setup
- There is context struct on stack. Why?
- sys\_exec() returns to trapret(), the trap frame will be popped !
- with “iret” jump into new program !
- New program is not old program , which could have accessed return value of sys\_exec()

# File Systems

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

# Introduction

- **Human end user's view of file system on a modern desktop operating system**
  - Files, directories(folders), hierarchy – acyclic graph like structure
  - Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at “/”

# Introduction

- **Secondary and Tertiary memory**
  - Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc.  
Used for storing files
  - Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary
  - IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms
  - Controllers provide a block/sector based read/write access
    - Block size is most typically 512 bytes
    - Can't read byte no. 33 directly. Must read sector 0 (byte 0 to 511) in memory and then access the byte no 33 in memory.

# Introduction

- **OS and File system**
  - OS bridges the gap between end user and storage hardware controller
  - Provides data structure to map the logical view of end users onto disk storage
  - Essentially an implementation of the acyclic graph on the sequential sector-based disk storage
    - Both in memory and on-disk
  - Provides system calls (open, read, write, ..., etc. ) to enable access to files, folders

# **What we are going to learn**

- **The operating system interface (system calls, commands/utilities) for accessing files in a file-system**
- **Design aspects of OS to implement the file system**

# What is a file?

- **A sequence of bytes , with**
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- **Types: Text files, binary files**
  - Text: All bytes are human readable
  - Binary: Non-text
- **Types: ODT, MP4, TXT, DOCX, etc.**
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application

| 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 to be**
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records
  - 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

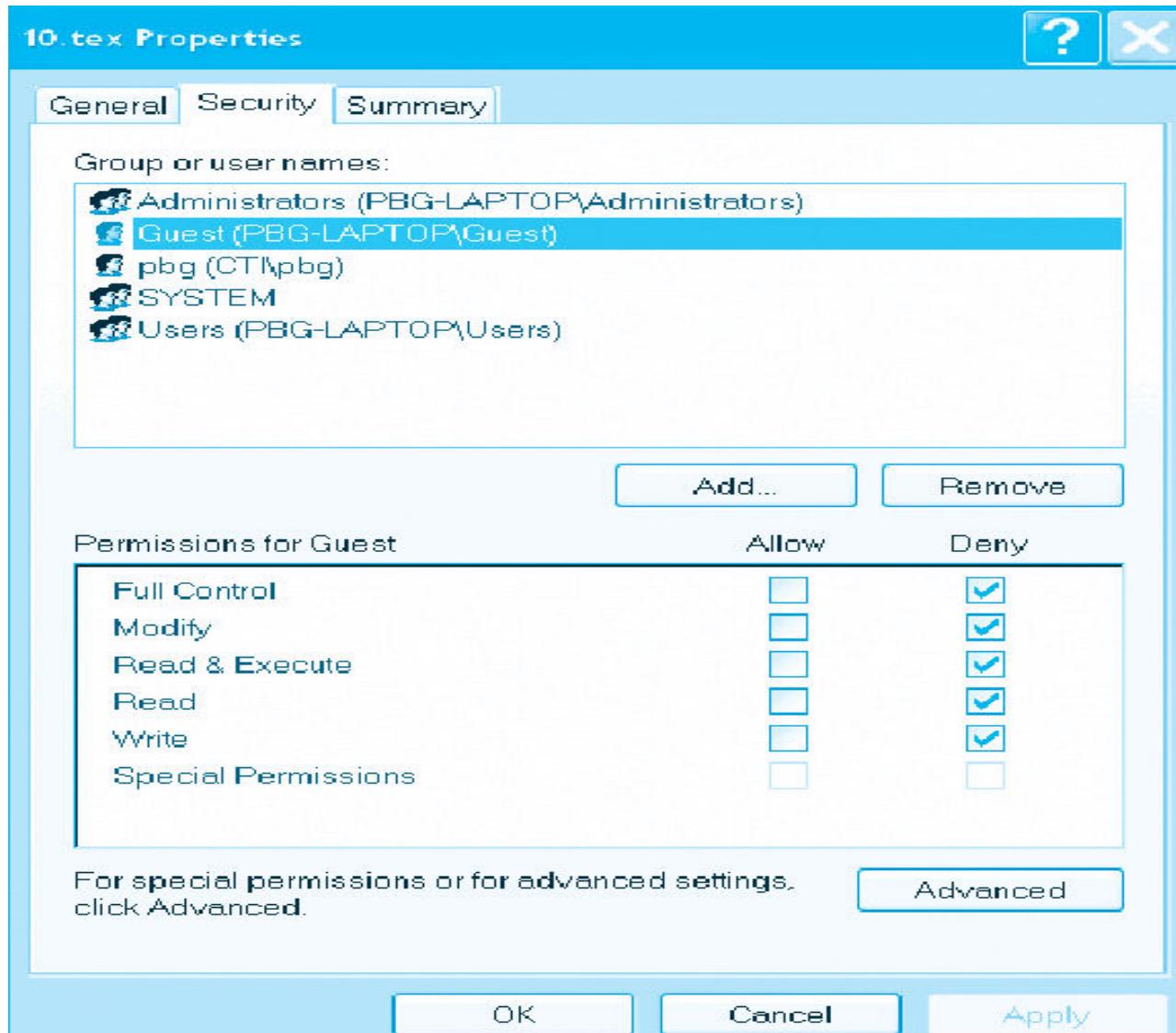
- Run
  - \$ ls -l
  - on Linux
  - To see file listing with different attributes
- Different OSes and file-systems provide different sets of file attributes
  - Some attributes are common to most, while some are different
  - E.g. name, size, owner can be found on most systems
  - “Executable” permission may not be found on all systems

# File protection attributes

- File owner/creator should be able to control:
    - what can be done
    - by whom
  - Types of access
    - Read
    - Write
    - Execute
    - Append
    - Delete
    - List
  - Linux: See commands: “chown”, “chgrp” and “chmod”
- 
- **Linux file permissions**
    - For owner, group and others
    - Read, write and execute
    - Total 9 permissions

# A Sample UNIX Directory Listing

|            |   |     |         |       |              |               |
|------------|---|-----|---------|-------|--------------|---------------|
| -rw-rw-r-- | 1 | pbg | staff   | 31200 | Sep 3 08:30  | intro.ps      |
| drwx-----  | 5 | pbg | staff   | 512   | Jul 8 09:33  | private/      |
| drwxrwxr-x | 2 | pbg | staff   | 512   | Jul 8 09:35  | doc/          |
| drwxrwx--- | 2 | pbg | student | 512   | Aug 3 14:13  | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff   | 9423  | Feb 24 2003  | program.c     |
| -rwxr-xr-x | 1 | pbg | staff   | 20471 | Feb 24 2003  | program       |
| drwx--x--x | 4 | pbg | faculty | 512   | Jul 31 10:31 | lib/          |
| drwx-----  | 3 | pbg | staff   | 1024  | Aug 29 06:52 | mail/         |
| drwxrwxrwx | 3 | pbg | staff   | 512   | Jul 8 09:35  | test/         |



# Windows XP Access List

# 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**

# Device Drivers

- **Hardware manufacturers provide “hardware controllers” for their devices**
- **Hardware controllers can operate the hardware, as instructed**
- **Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions**
  - This bridges the hardware-software gap
- **OS programmers, typically, write one “device driver” code that interacts with one hardware controller**
  - This is pure C code
  - Which calls I/O instructions for hardware controller
  - This code is independent of most of the OS code
  - Often this code is like an add-on Module , eg. Linux kernel

# Disk device driver

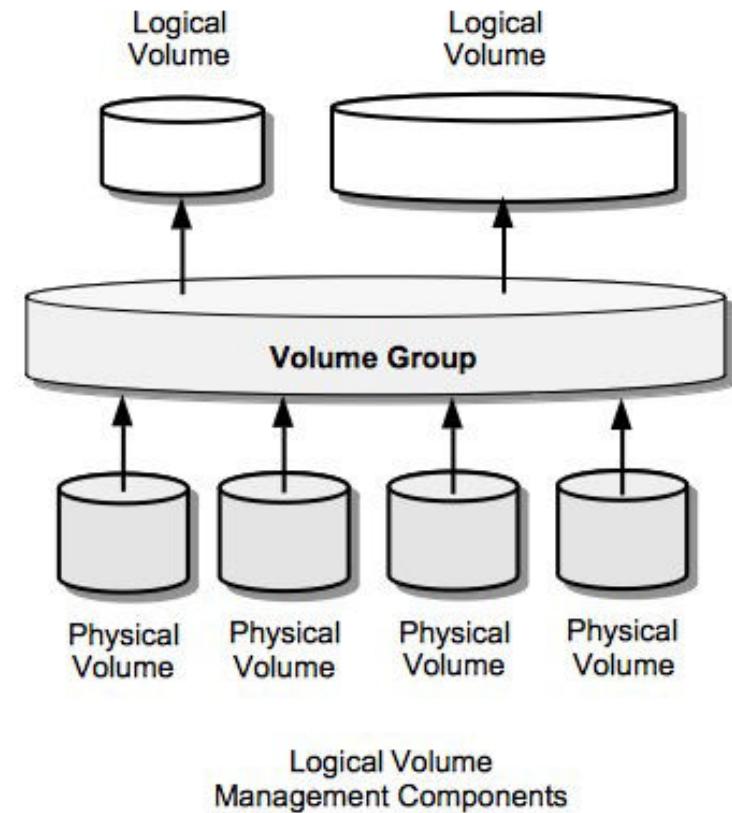
- OS views the disk as a logical sequence of blocks
  - OS's assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it's view of “logical block” of the disk, into physical sector numbers
- Acts as a translator between rest of OS and hardware controller
- xv6: ide.c

# OS's job now

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

# Volume Managers

- **Special type of kernel device drives, which reside on top of disk device drivers**
- **Provide a more abstract view of the underlying hardware**
- **E.g. Can combine two physical hard disks, and present them as one**
- **Allow end users to**
  - “combine one or more physical disks to create physical volumes”
  - “create volume groups out of a set of physical volumes”,
  - “create logical volumes within volume groups”

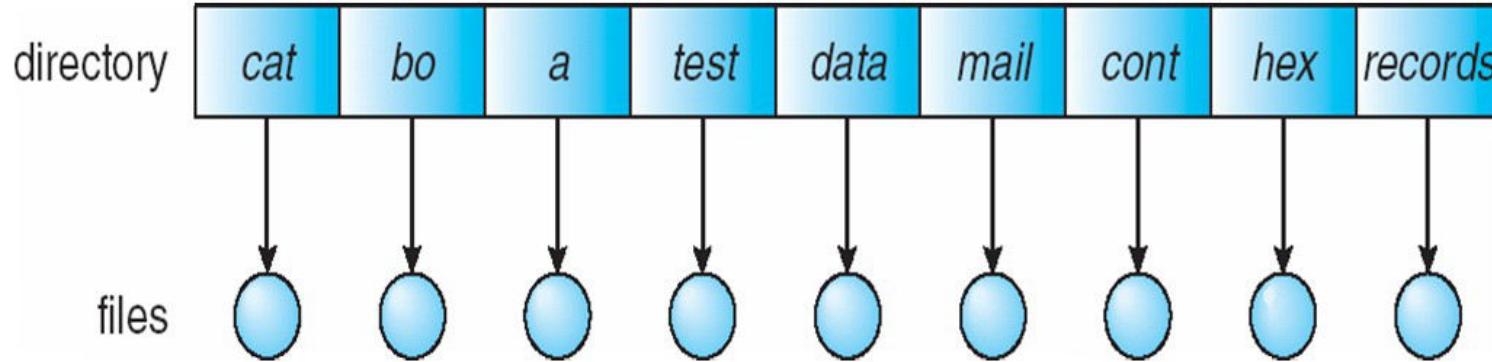


# Formatting

- **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

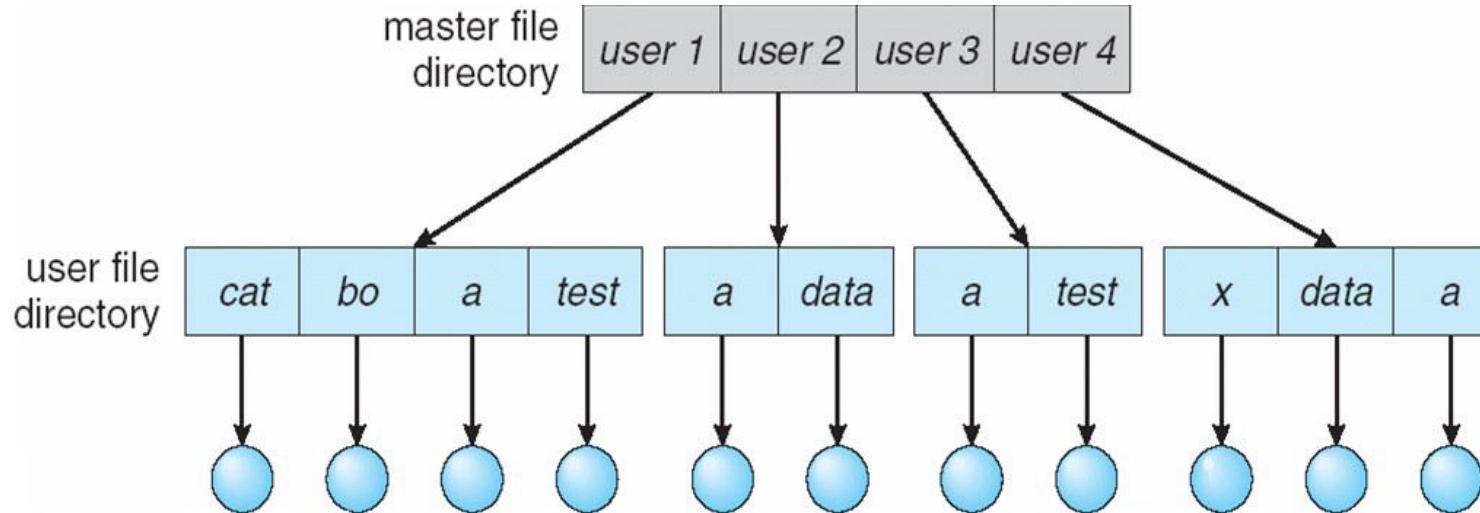


Naming problem

Grouping  
problem

# Different types of “layouts”

## Two level directory



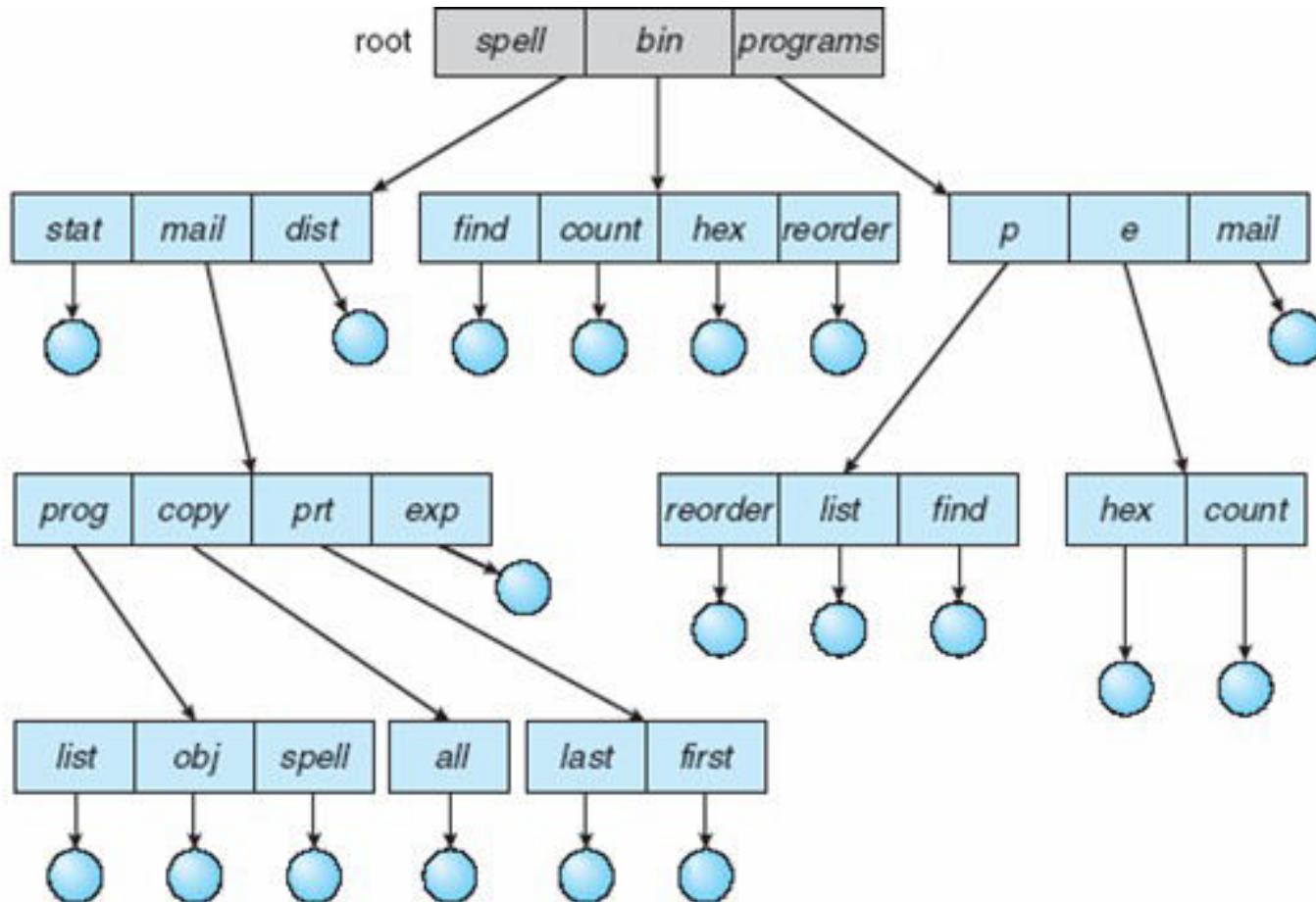
Path name

Can have the same file name for different user

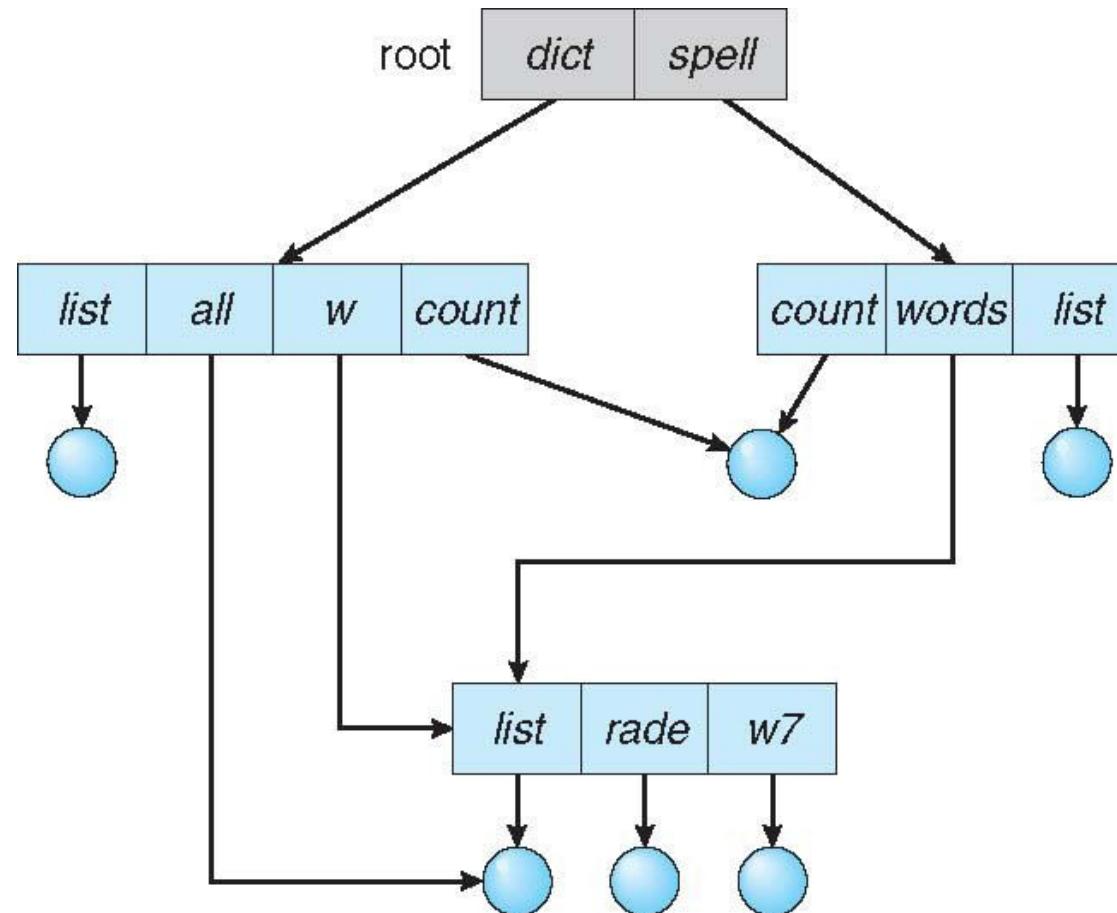
Efficient searching

No grouping capability

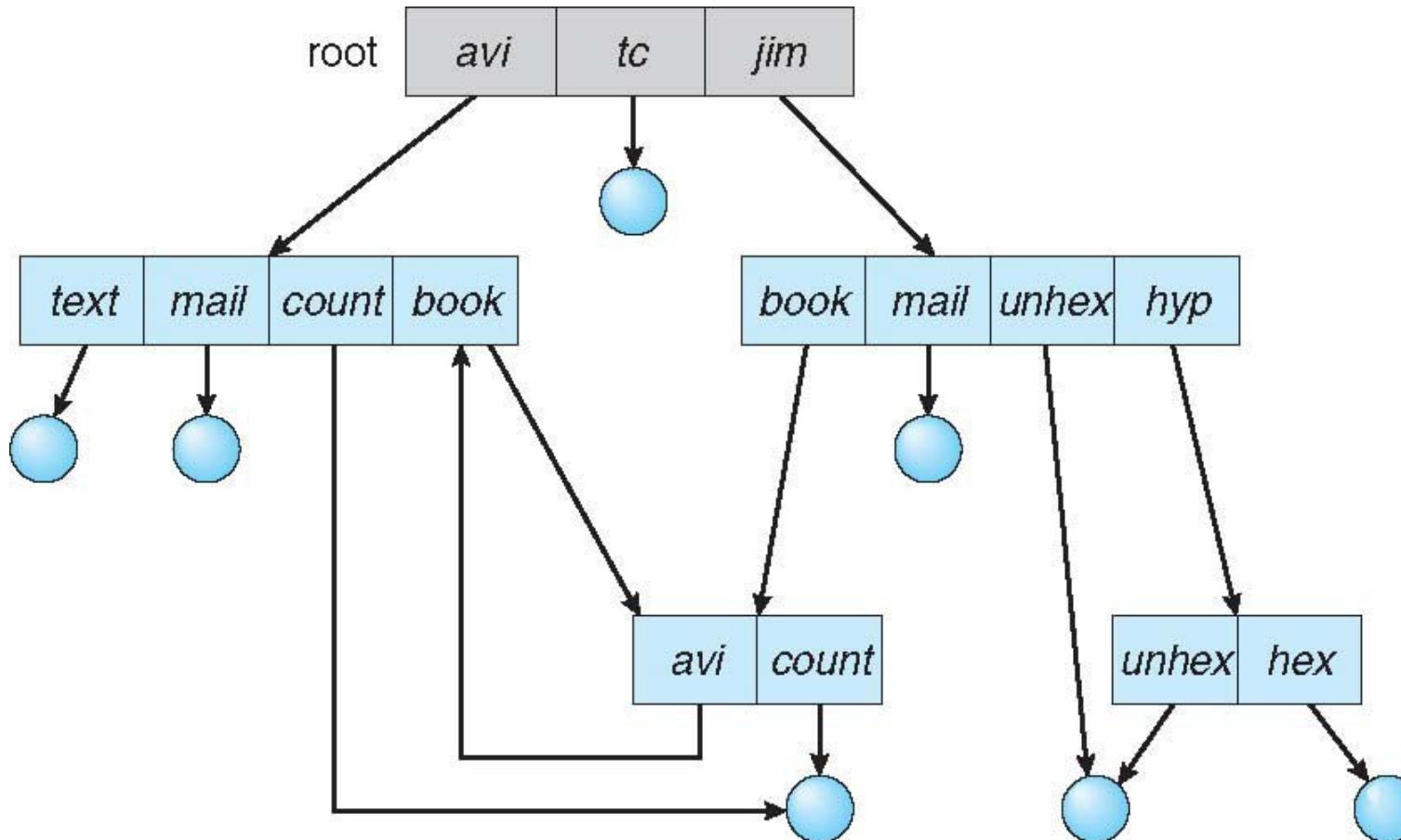
# Tree Structured directories



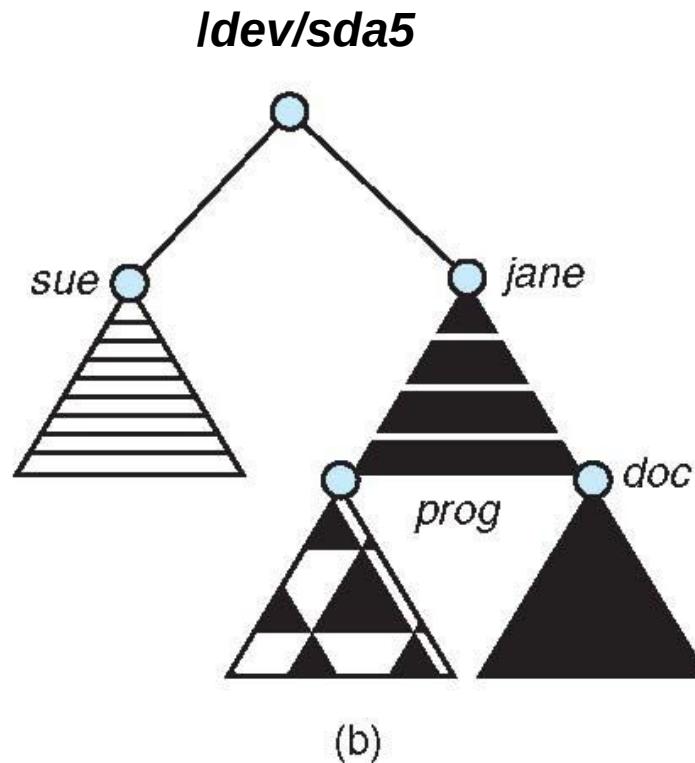
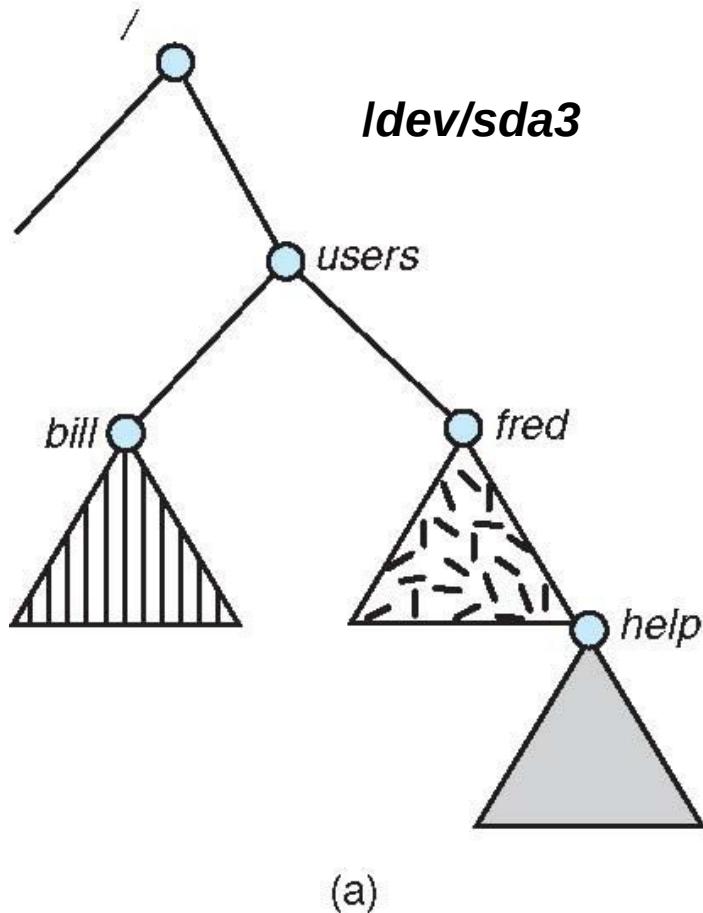
# Acyclic Graph Directories



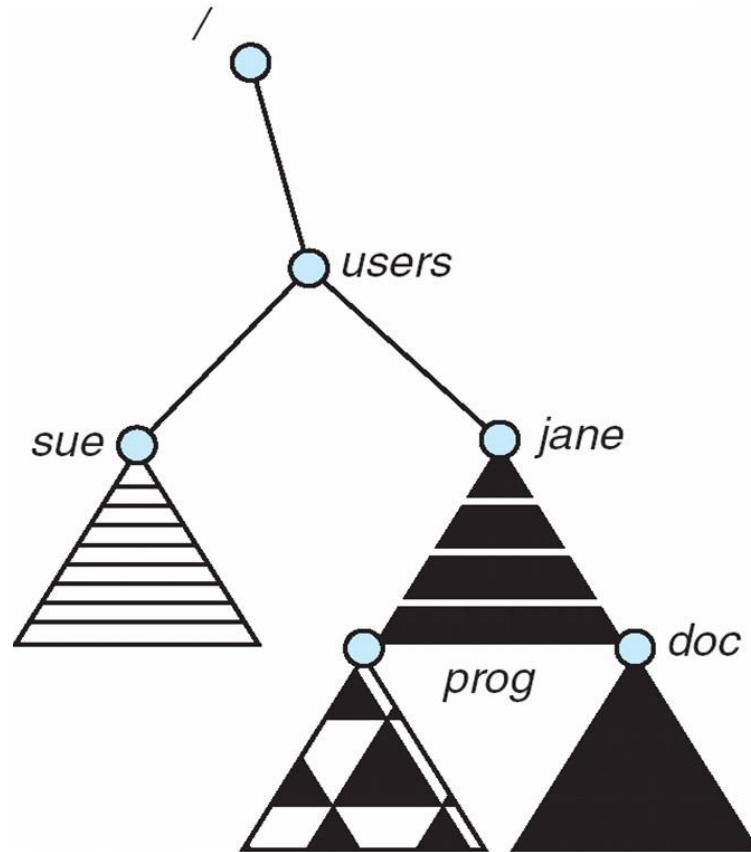
# General Graph directory



# 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 folder */a/b* on this computer
-

# File sharing semantics

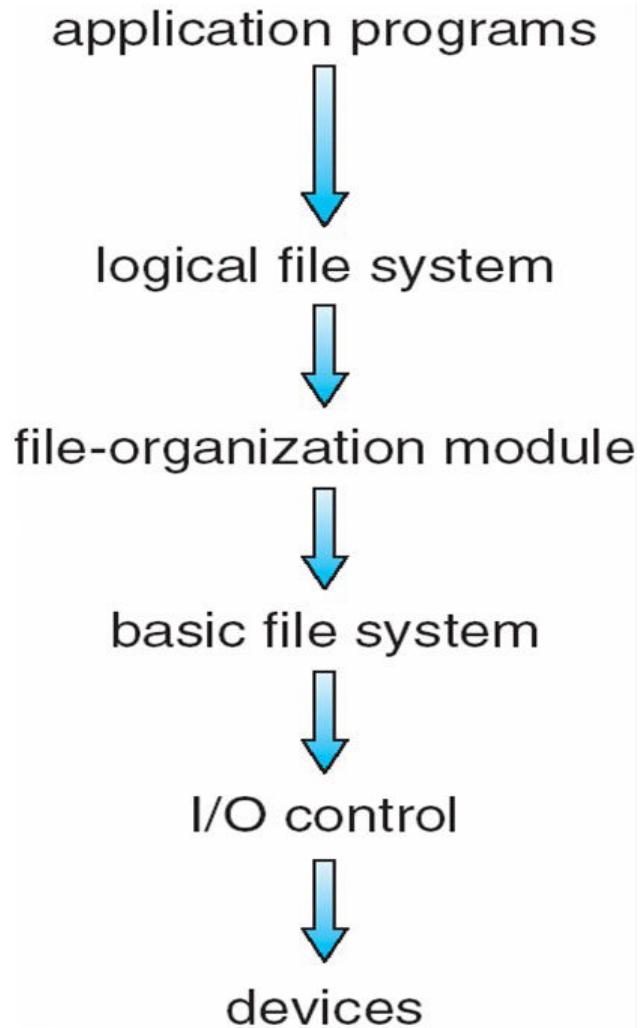
- **Consistency semantics specify how multiple users are to access a shared file simultaneously**
- **Unix file system (UFS) implements:**
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write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

# **Implementing file systems**

# **File system on disk**

- **Disk I/O in terms of sectors (512 bytes)**
- **File system: implementation of acyclic graph using the linear sequence of sectors**
- **Device driver: available to rest of the OS code to access disk using a block number**

# 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

## 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(sectorno, 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 slightly differently, but following the layering principle!*

# Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
  - The Virtual File System (to be discussed later) helps us combine multiple physically different file systems into one logical layer at the top

# File system implementation: Different problems to be solved

- **What to do at boot time ?**
- **How to store directories and files on the partition ?**
  - Complex problem. Hierarchy + 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**
- **How to mount a file system , how to unmount**

# File system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures. Let's see some of the important ones.
  - Boot control block contains info needed by system to boot OS from that volume
    - Not always needed. Needed if volume contains OS, usually first block of volume
  - Volume control block (superblock, master file table) contains volume details
    - Total # of blocks, # of free blocks, block size, free block pointers or array
  - Per-file File Control Block (FCB) contains many details about the file
    - Inode(FCB) number, permissions, size, dates
  - Directory structure organizes the files
    - Names and inode numbers, master file table

# 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

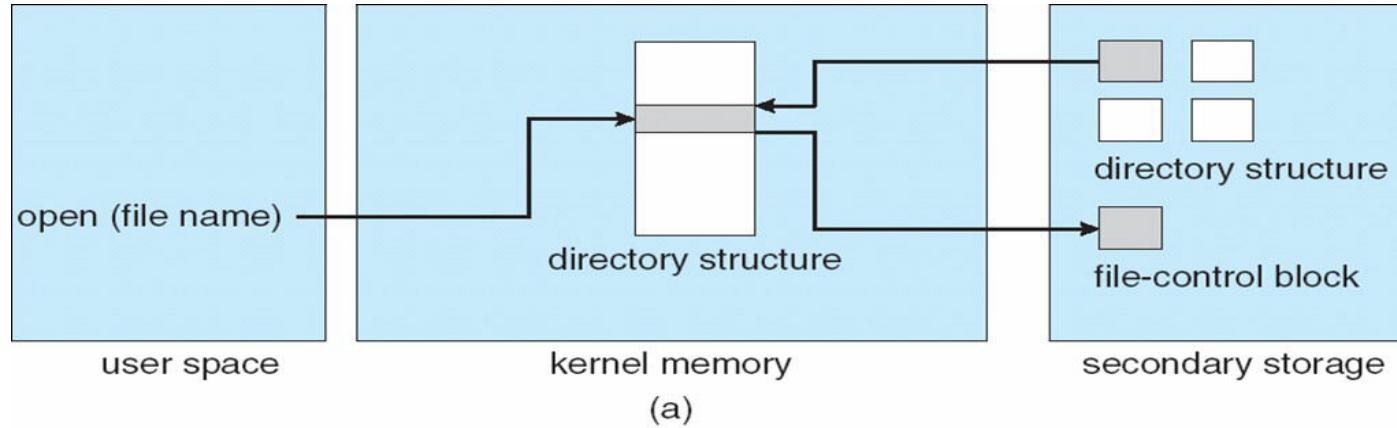
Why does it NOT  
contain the

Name of the file ?

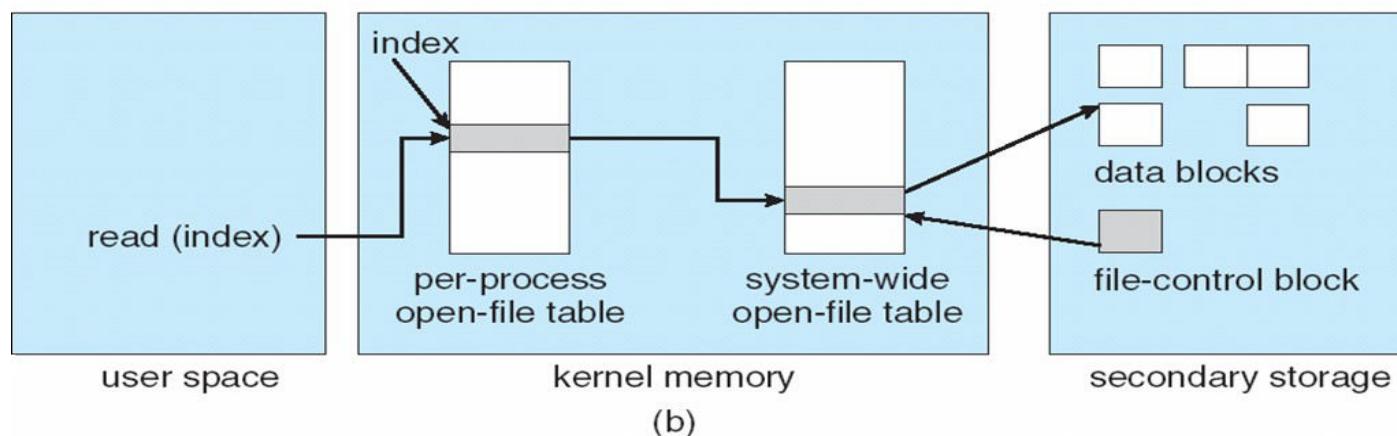
# In memory data structures

- **Mount table**
  - storing file system mounts, mount points, file system types
- **See next slide for “file” realated 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

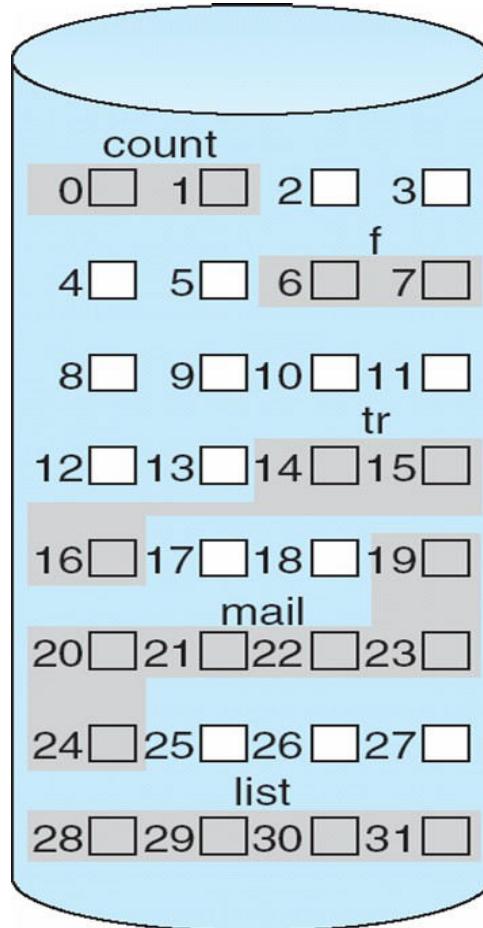
# Directory Implementation

- **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



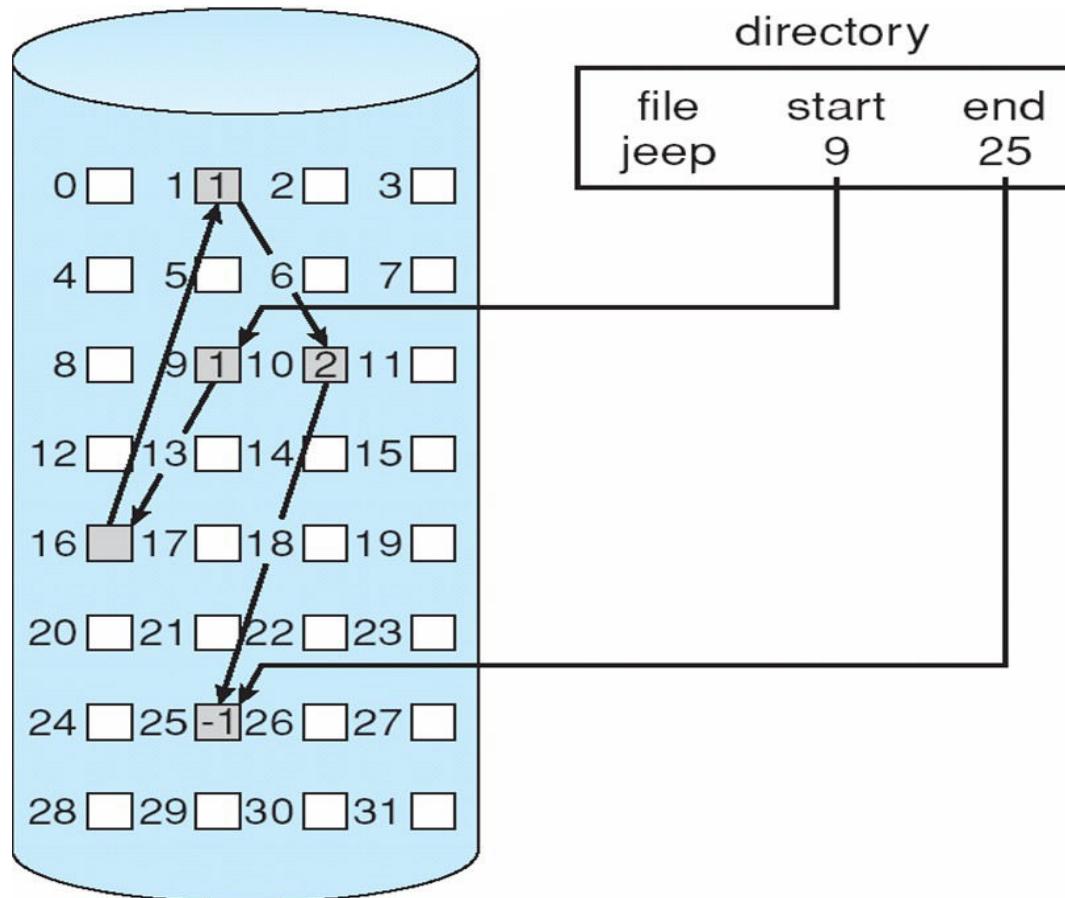
directory

| 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

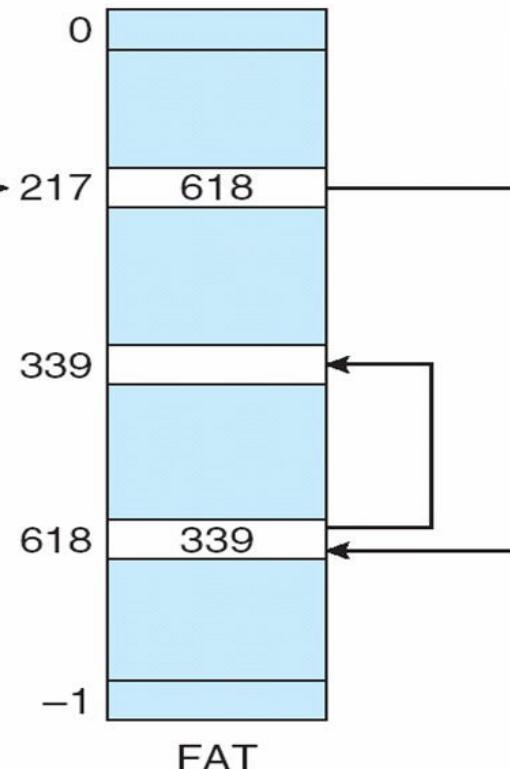
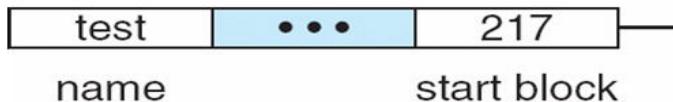


# Linked allocation of blocks to a file

- **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

directory entry



- 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

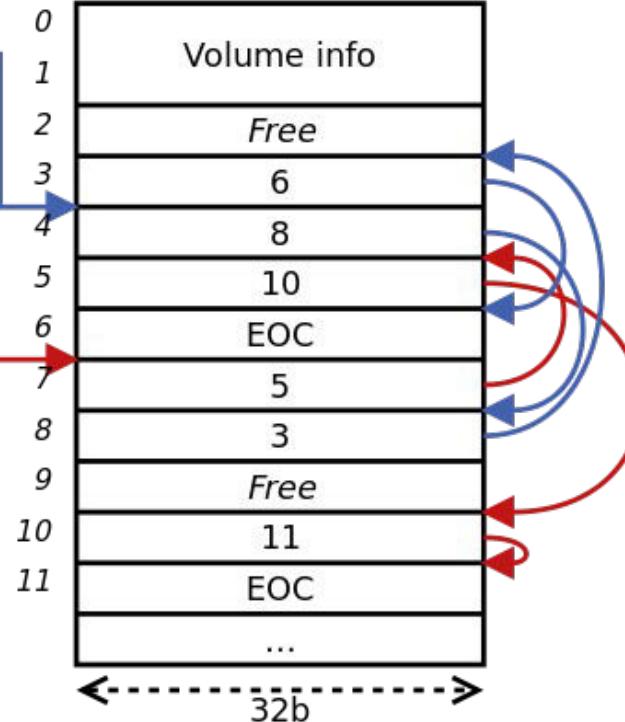
# FAT: File Allocation Table

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

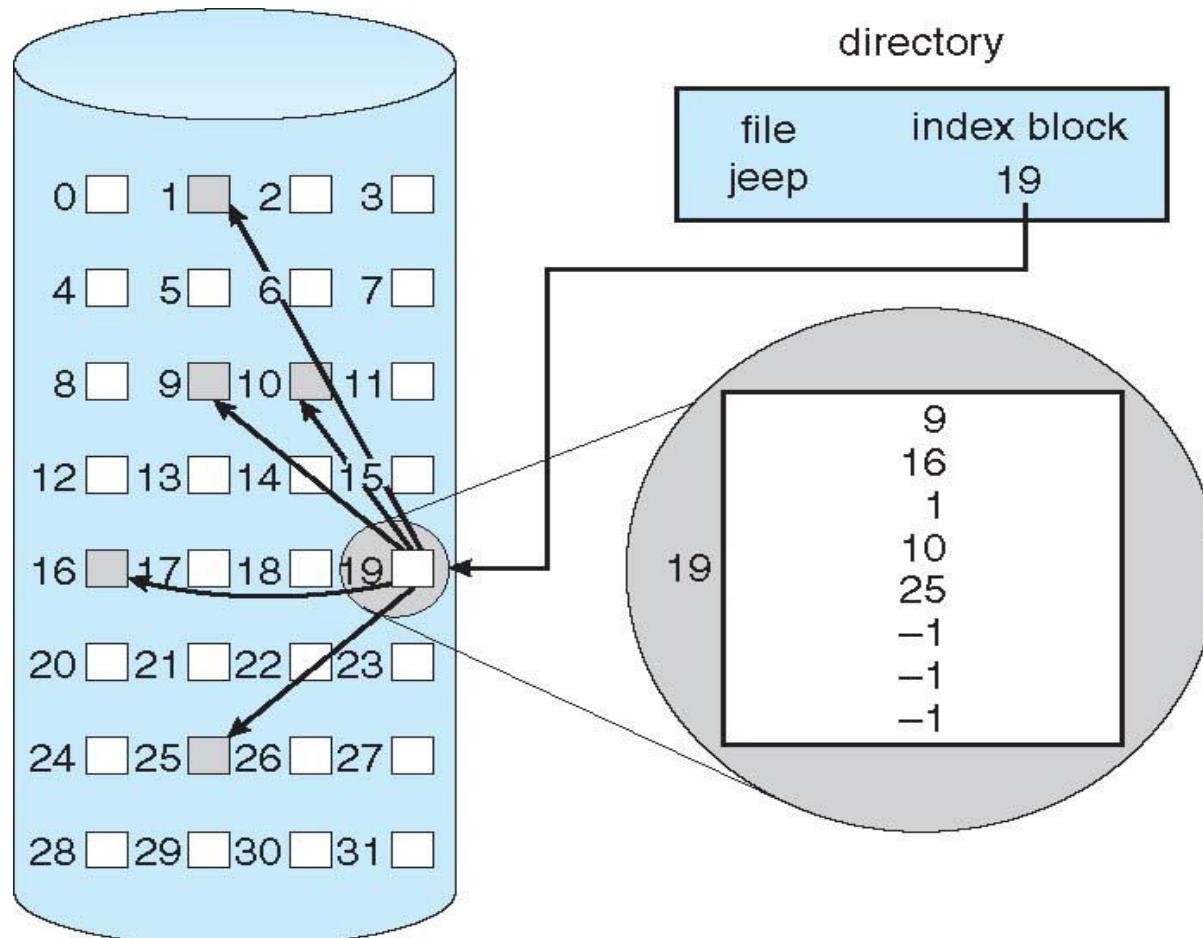
Directory table entry (32B)

|                           |
|---------------------------|
| Filename (8B)             |
| Extension (3B)            |
| Attributes (1B)           |
| Reserved (1B)             |
| Create time (3B)          |
| Create date (2B)          |
| Last access date (2B)     |
| First cluster # (MSB, 2B) |
| Last mod. time (2B)       |
| Last mod. date (2B)       |
| First cluster # (LSB, 2B) |
| File size (4B)            |

File allocation table



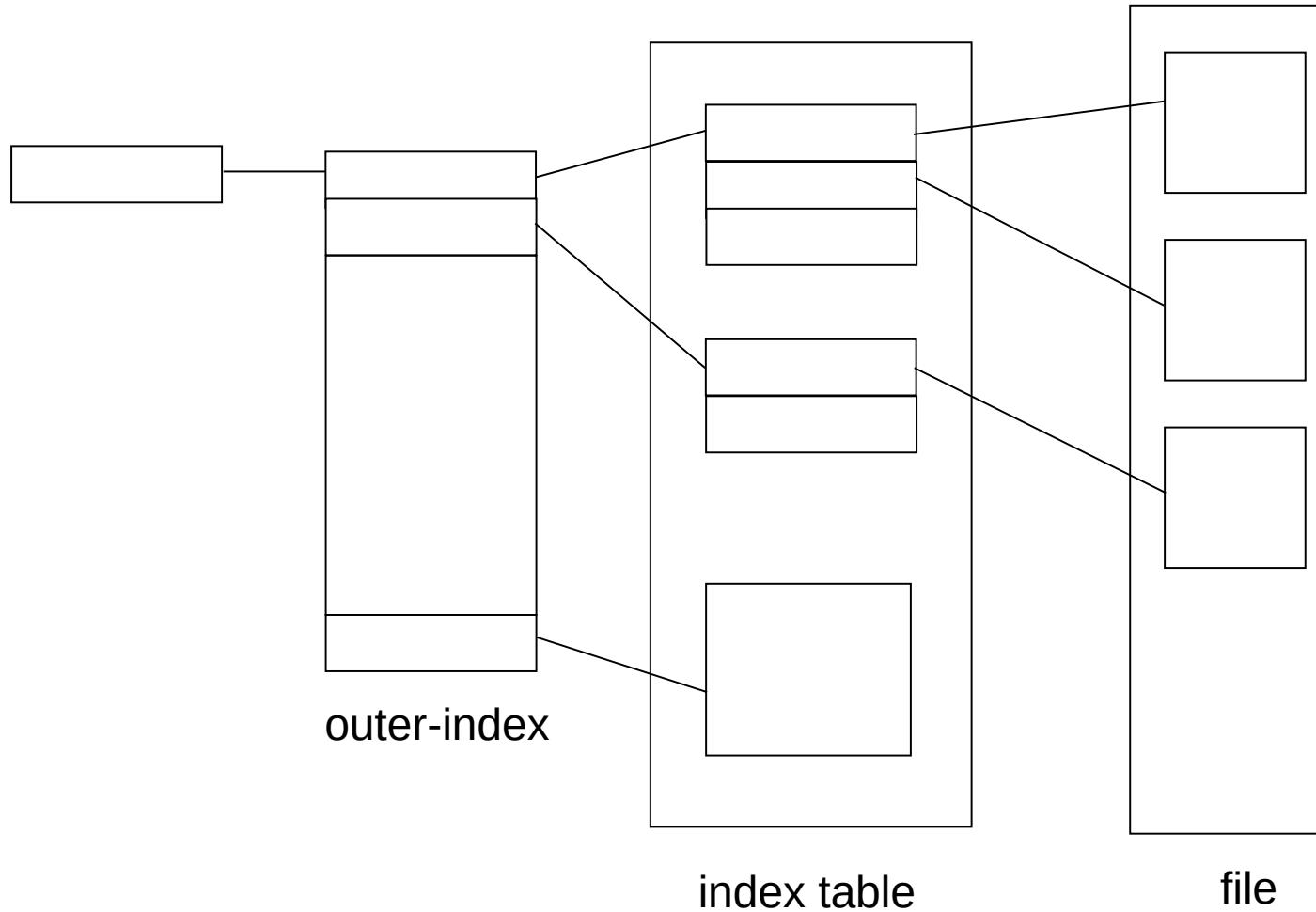
# Indexed allocation



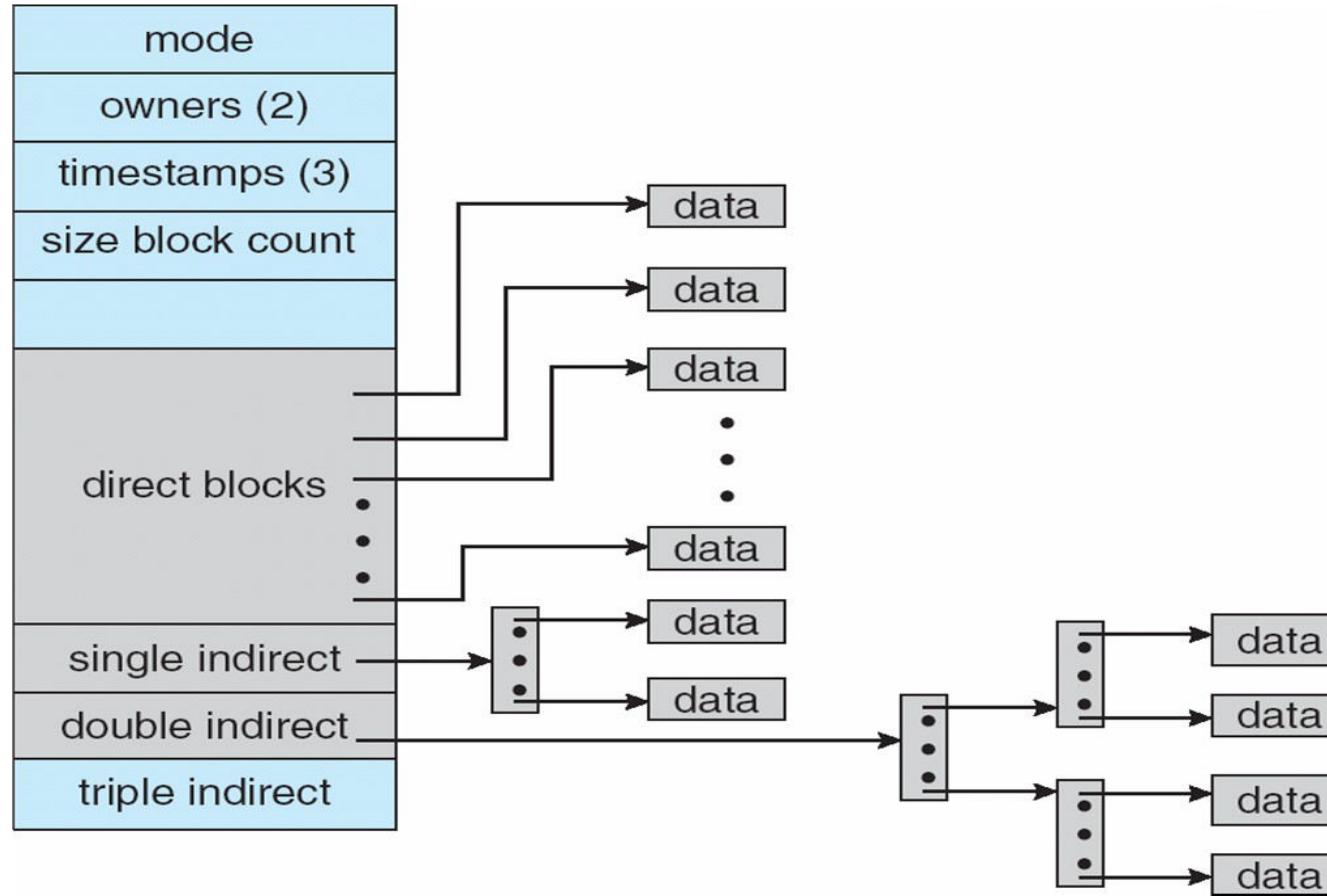
# Indexed allocation

- 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



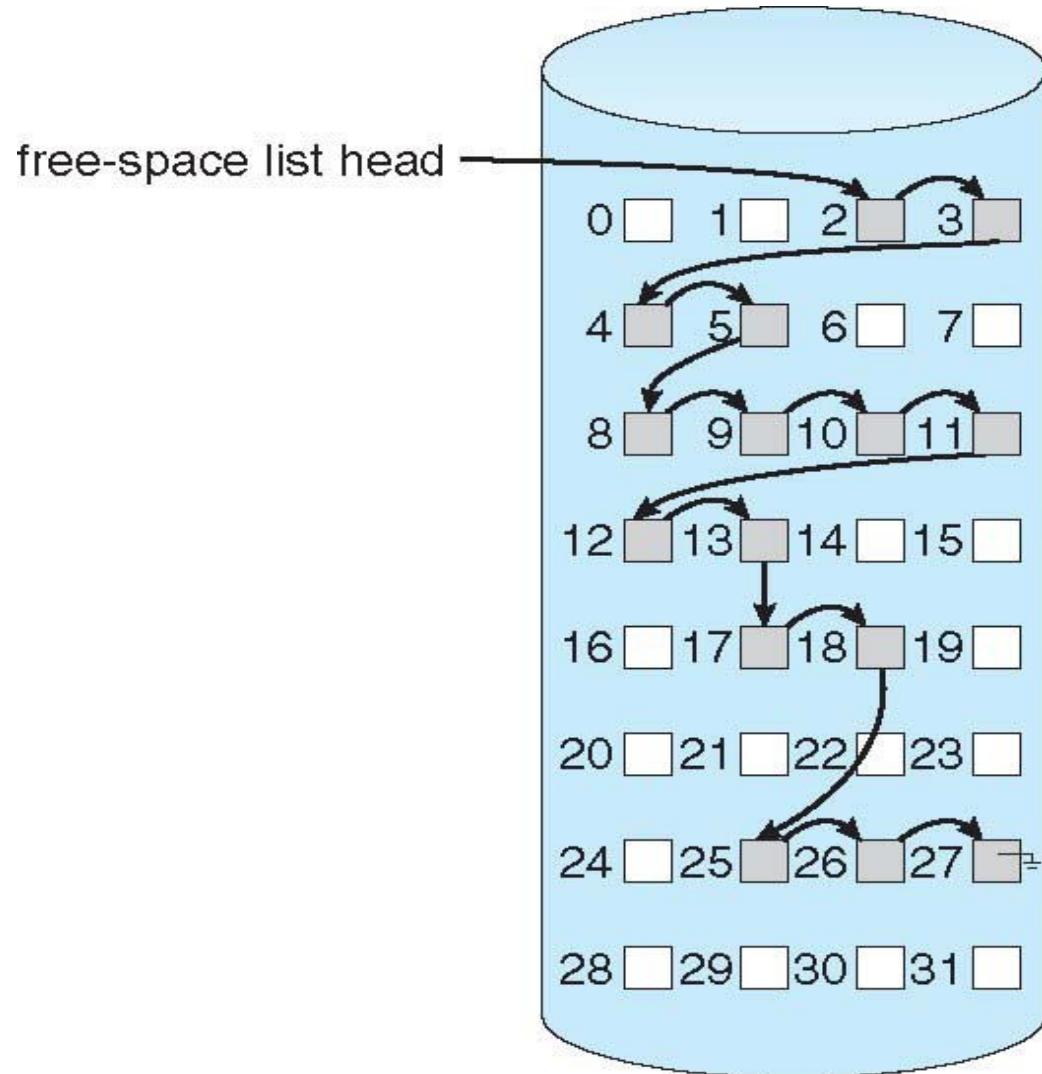
# Free Space Management

- **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!)

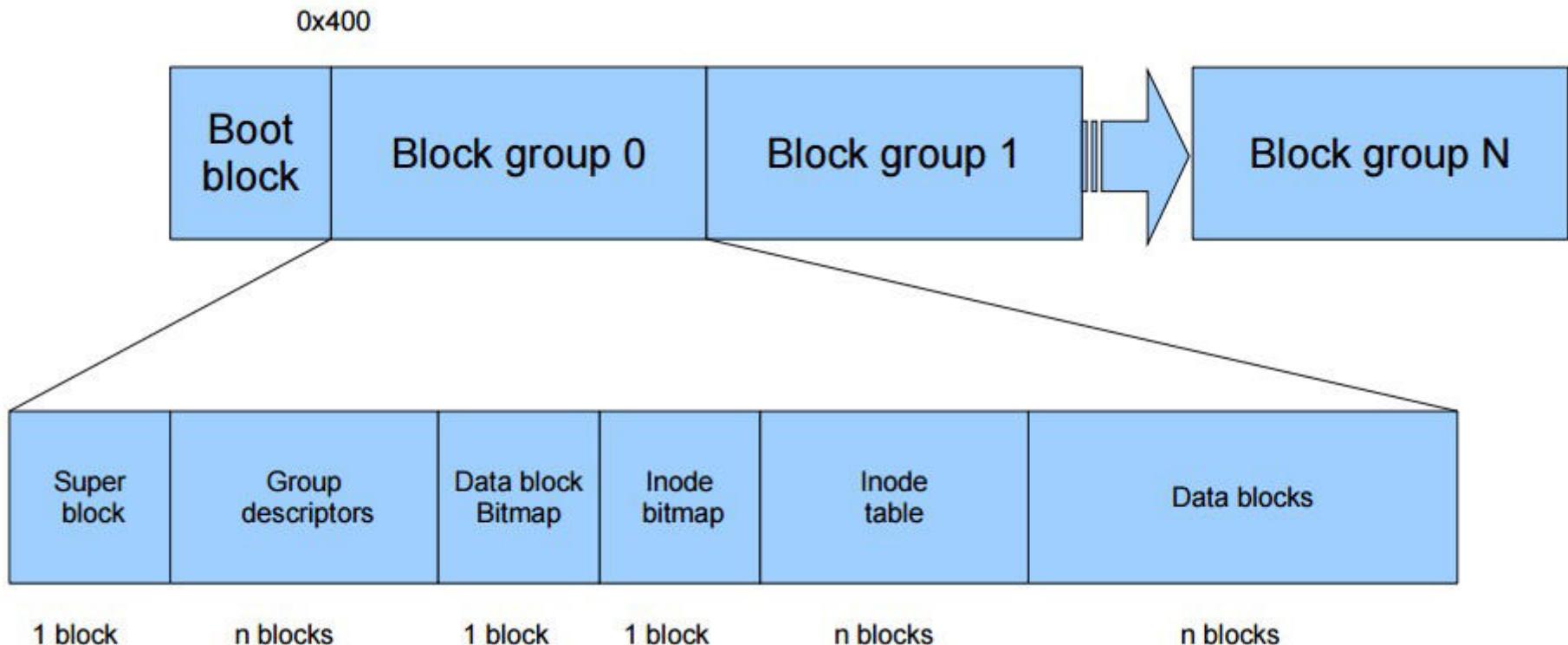


# Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

# Ext2 FS layout

# Ext2 FS Layout



```
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 */
```

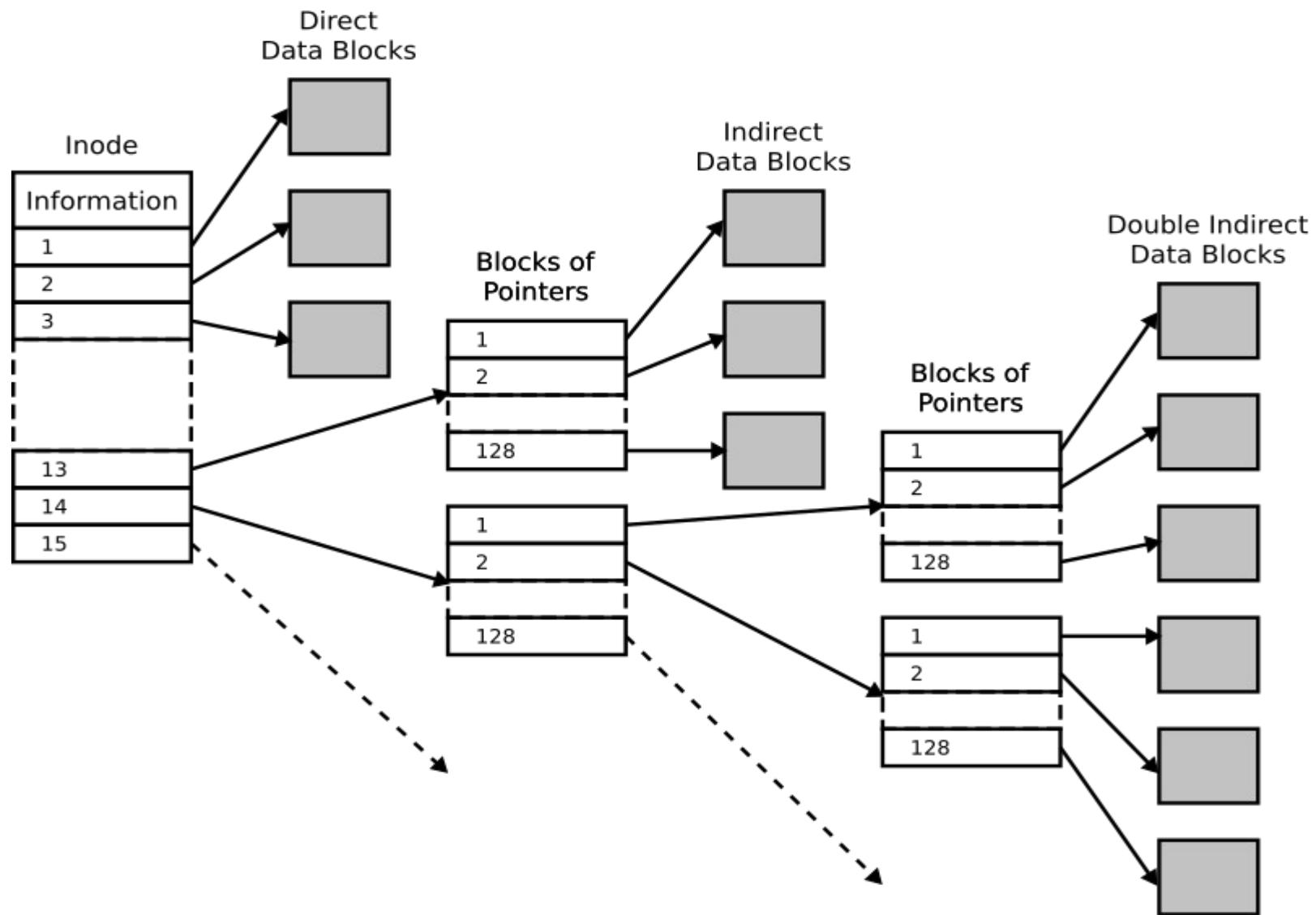
```
struct ext2_super_block {
...
 __le16 s_minor_rev_level; /* minor revision level */
 __le32 s_lastcheck; /* time of last check */
 __le32 s_checkinterval; /* max. time between checks */
 __le32 s_creator_os; /* OS */
 __le32 s_rev_level; /* Revision level */
 __le16 s_def_resuid; /* Default uid for 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];
};
```

```
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: Directory entry

|    | inode | rec_len | file_type | name_len | 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 |
| 40 | 67    | 28      | 3         | 2        | u    | s  | r  | \0 |   |    |    |
| 52 | 0     | 16      | 7         | 1        | o    | l  | d  | f  | i | 1  | e  |
| 68 | 34    | 12      | 4         | 2        | s    | b  | i  | n  |   |    |    |

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

**Efficiency and Performance  
(and the risks created  
while trying to achieve it!)**

# Efficiency

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
  -

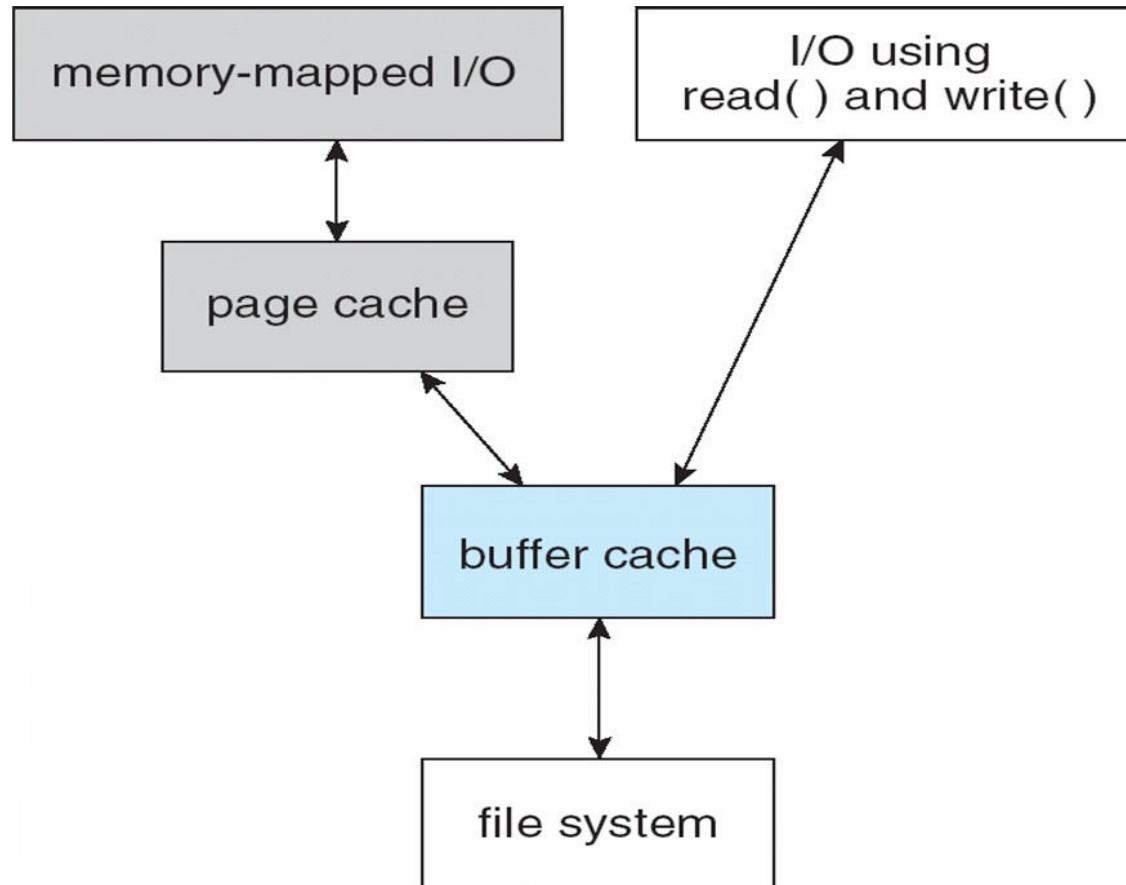
# Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

# Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

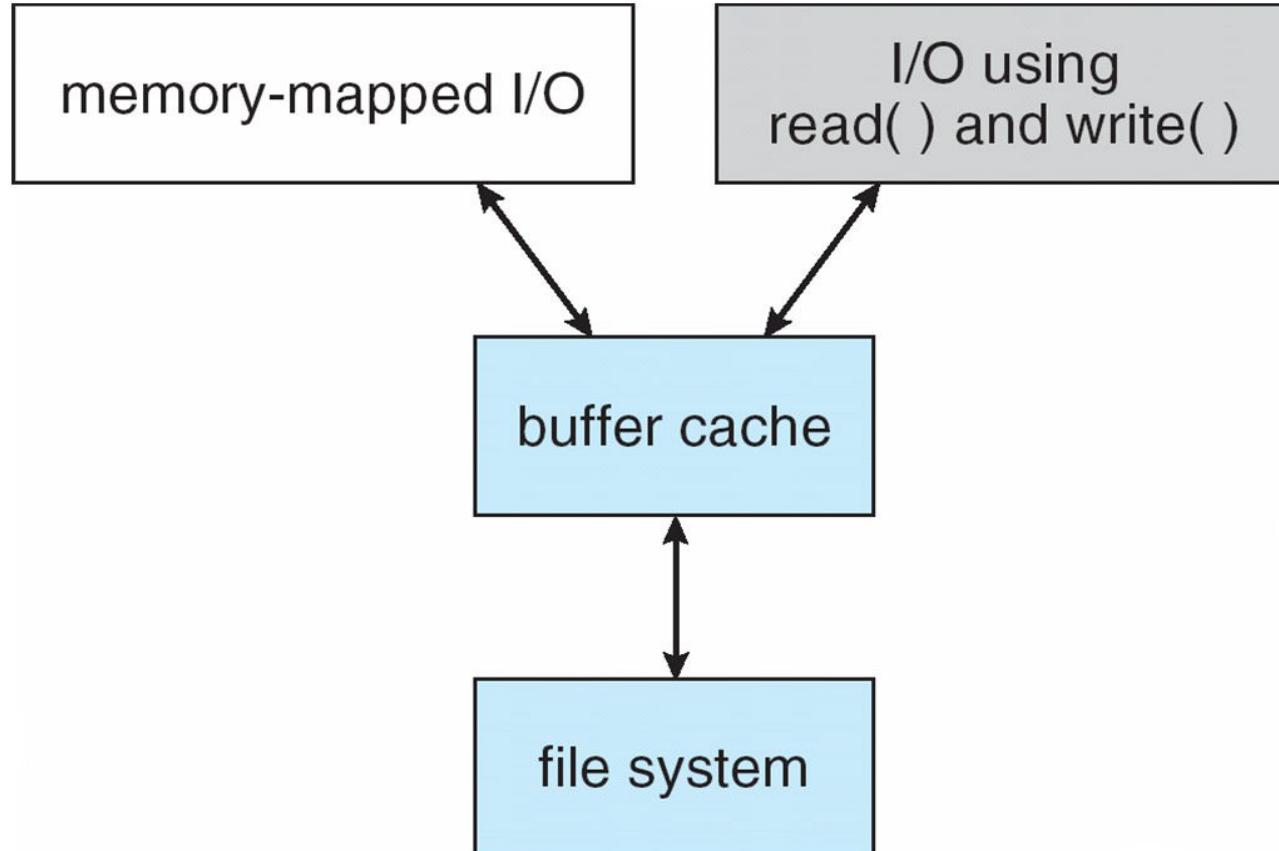
# I/O Without a Unified Buffer Cache



# Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

# I/O Using a Unified Buffer Cache



# Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

# Recovery

- **Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data from backup**

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

# Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!















# File Systems

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

# **System calls related to files/file-system**

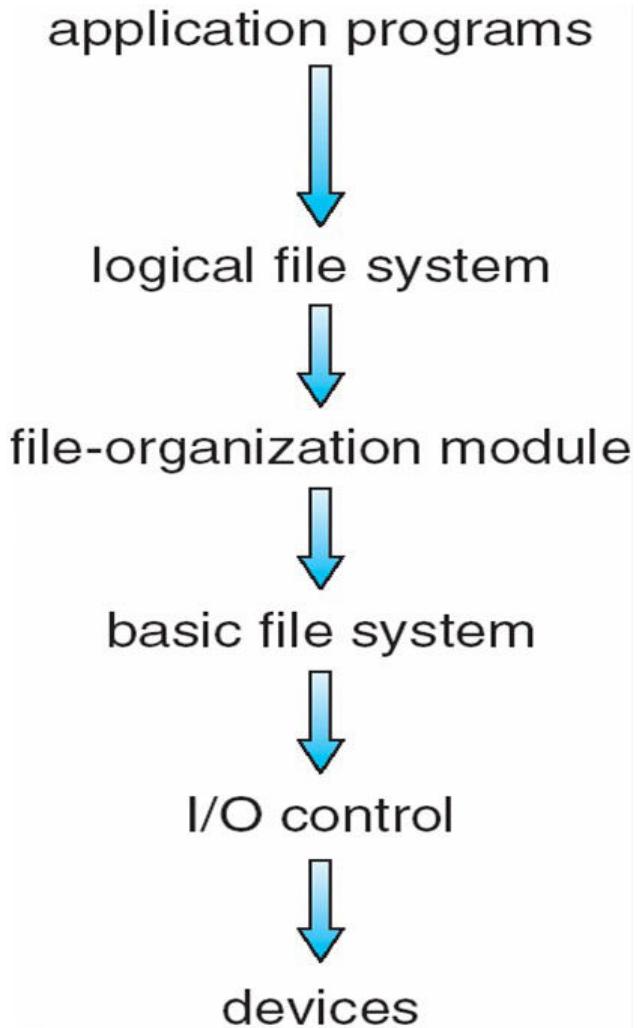
- Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...**

# **Implementing file systems**

# **File system on disk**

- **Disk I/O in terms of sectors (512 bytes)**
- **File system: implementation of acyclic graph using the linear sequence of sectors**
- **Device driver: available to rest of the OS code to access disk using a block number**

# 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(sectorno, 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 slightly differently, but following the layering principle!*

# 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

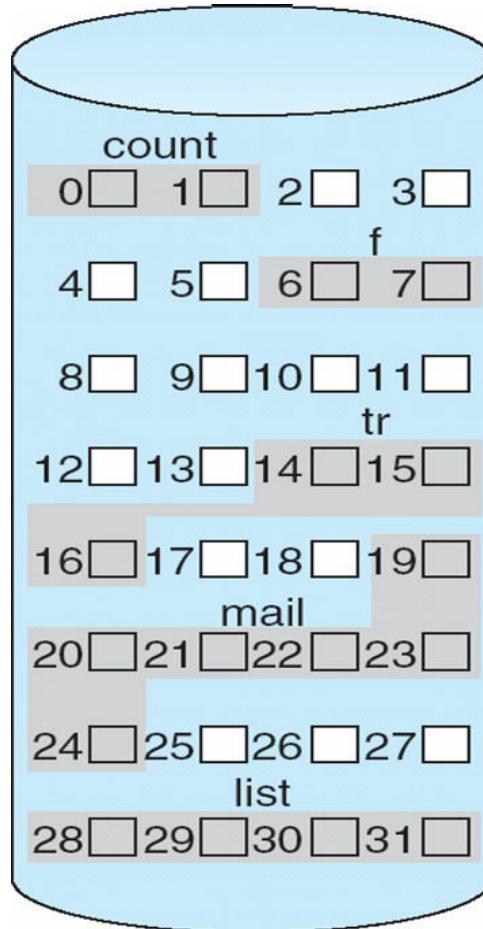
Why does it NOT  
contain the

Name of the file ?

# 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



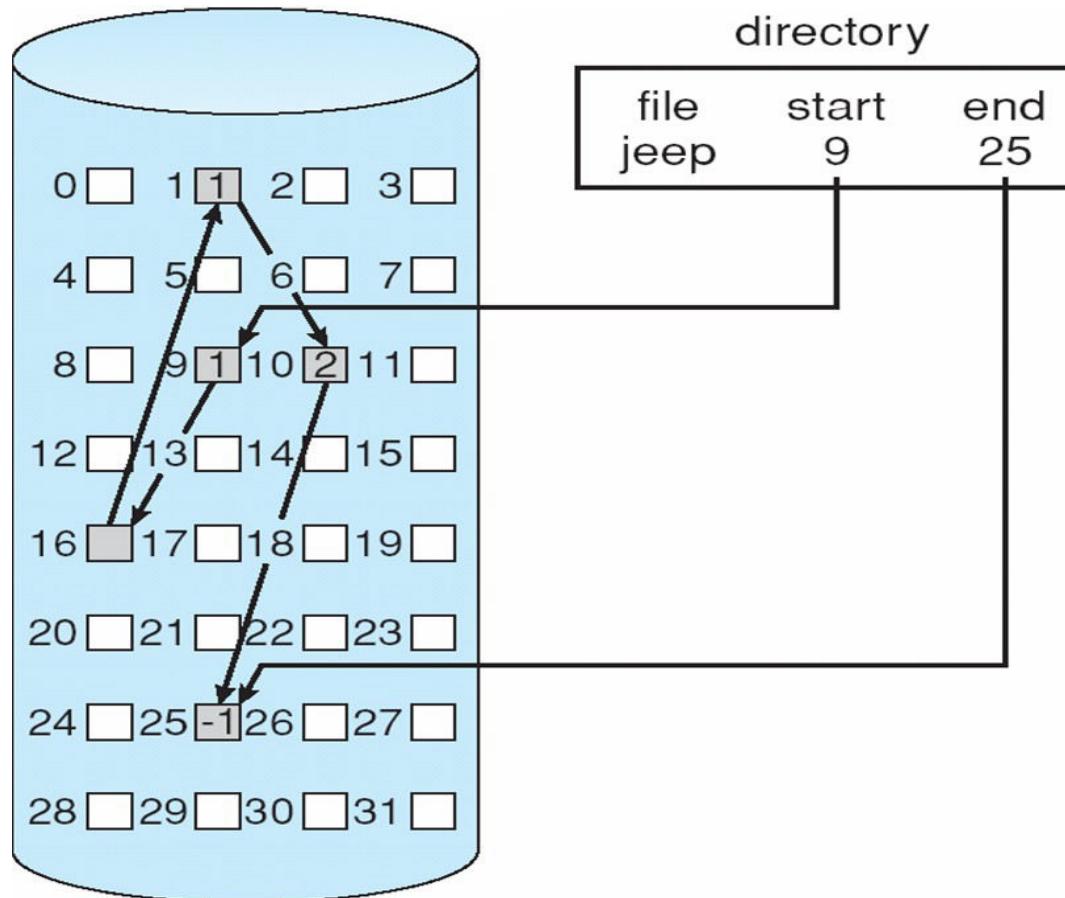
directory

| 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

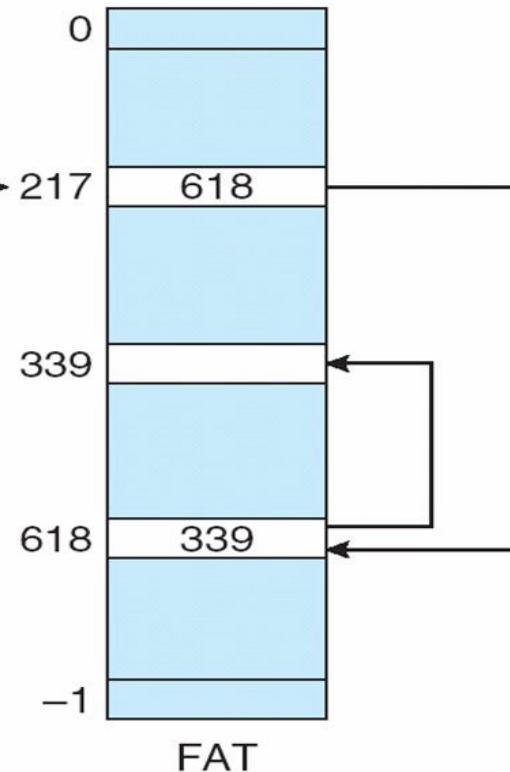
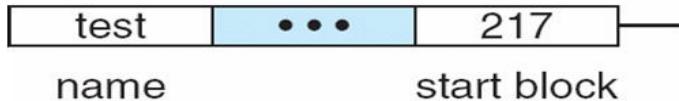


# Linked allocation of blocks to a file

- **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

directory entry



- 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

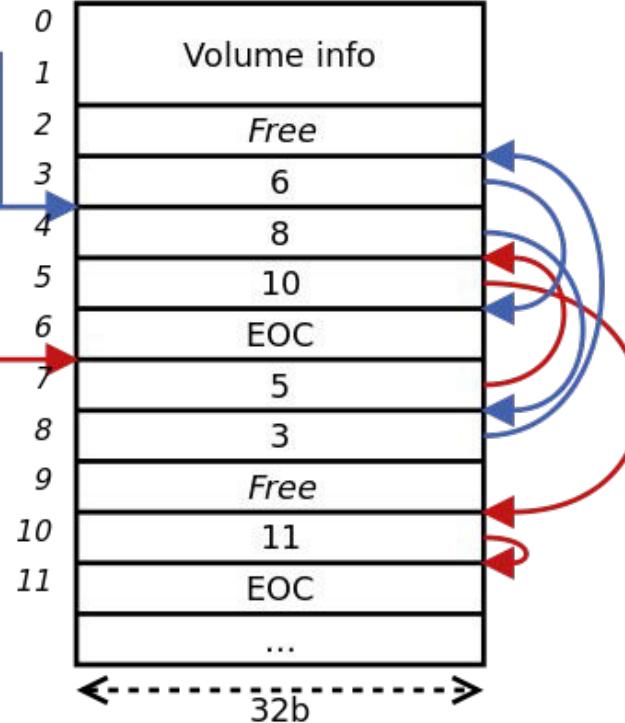
# FAT: File Allocation Table

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

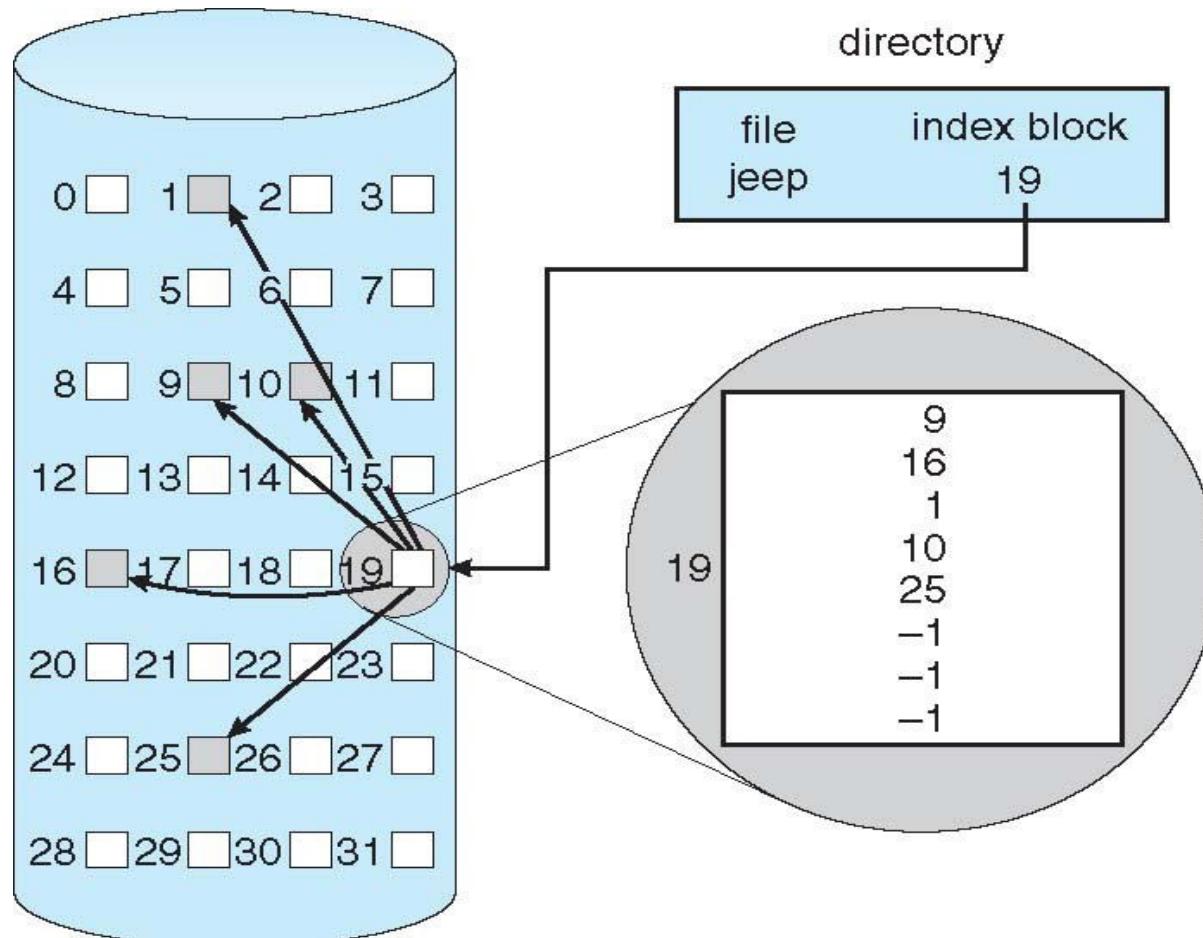
Directory table entry (32B)

|                           |
|---------------------------|
| Filename (8B)             |
| Extension (3B)            |
| Attributes (1B)           |
| Reserved (1B)             |
| Create time (3B)          |
| Create date (2B)          |
| Last access date (2B)     |
| First cluster # (MSB, 2B) |
| Last mod. time (2B)       |
| Last mod. date (2B)       |
| First cluster # (LSB, 2B) |
| File size (4B)            |

File allocation table



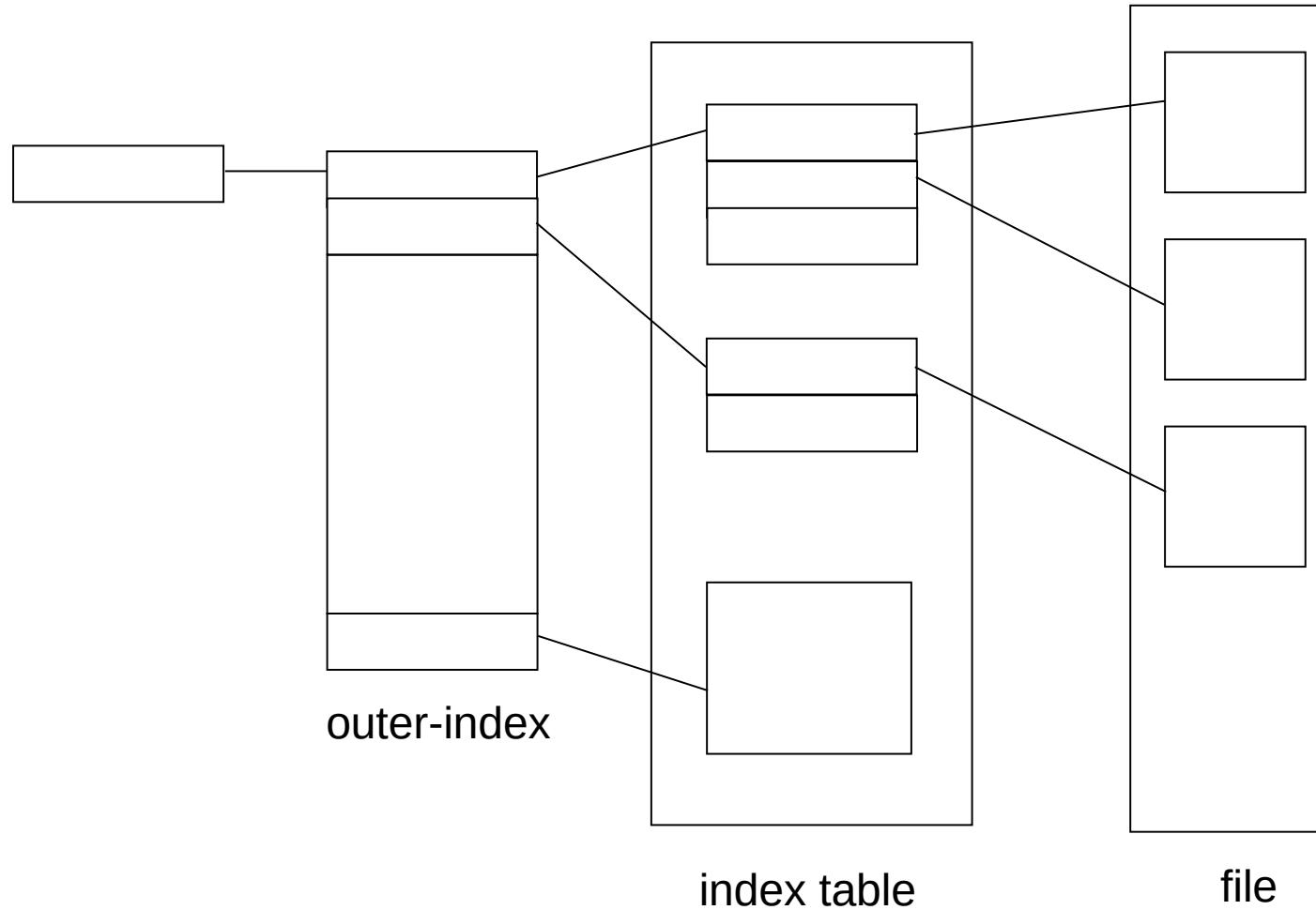
# Indexed allocation



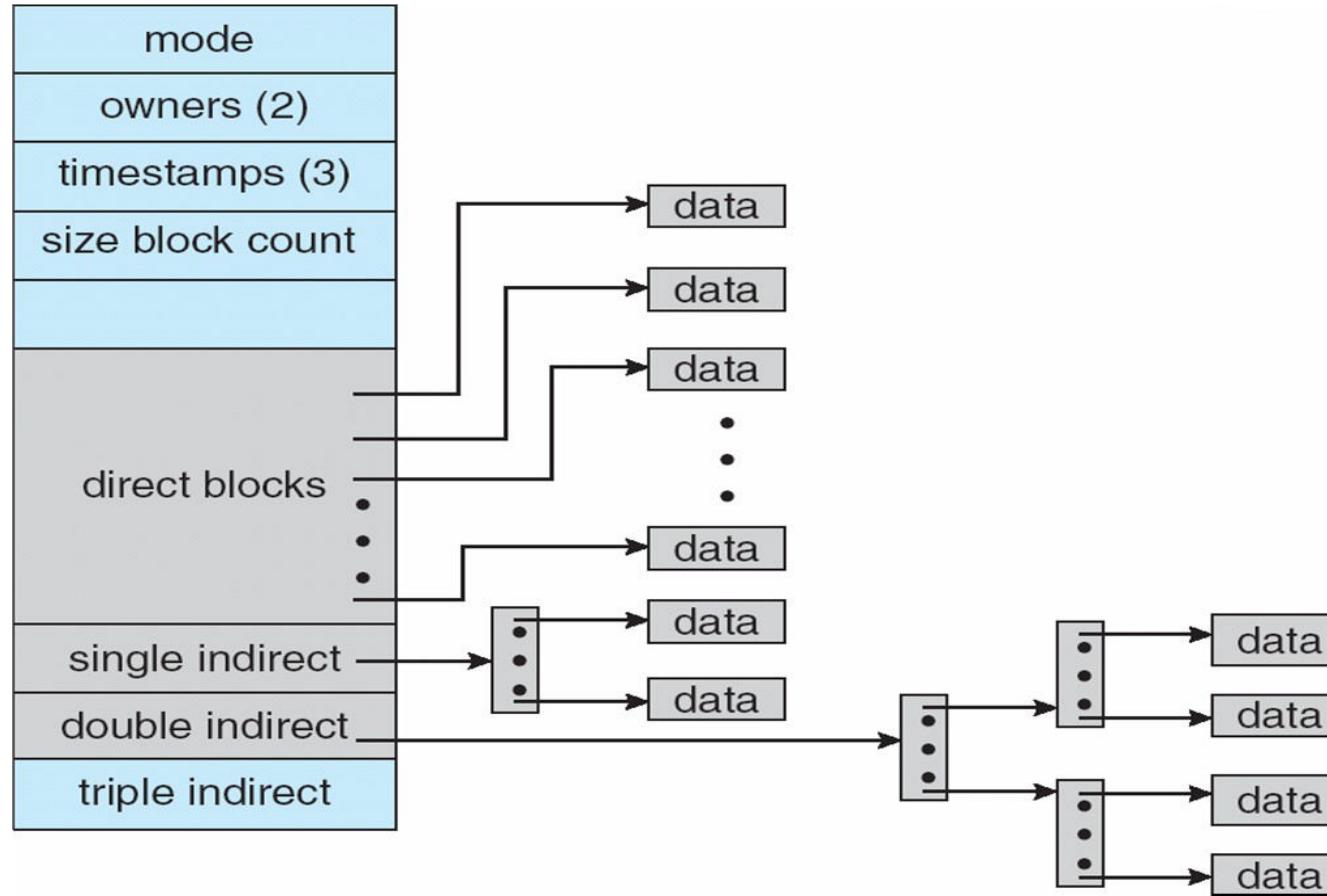
# Indexed allocation

- 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



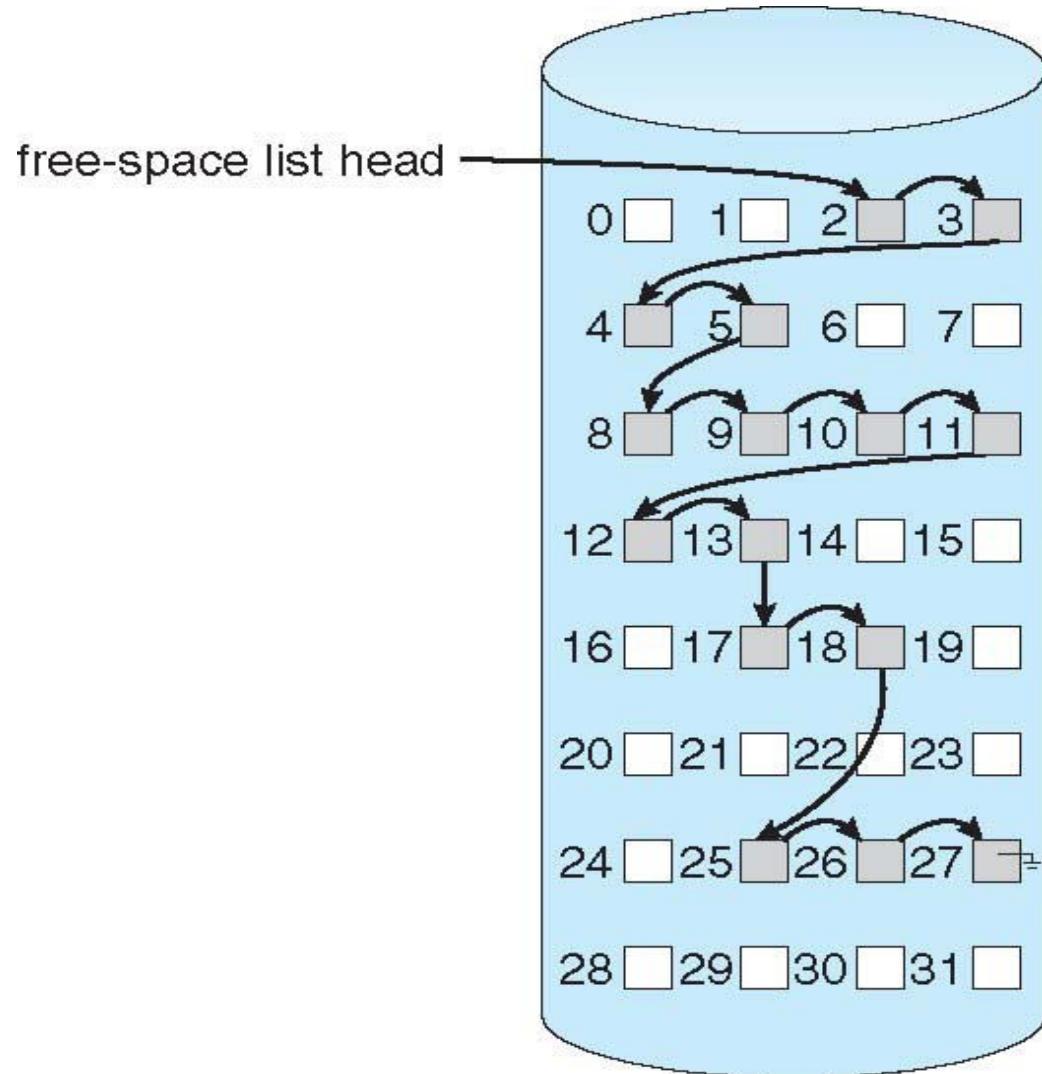
# Free Space Management

- 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!)



# Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

# Directory Implementation

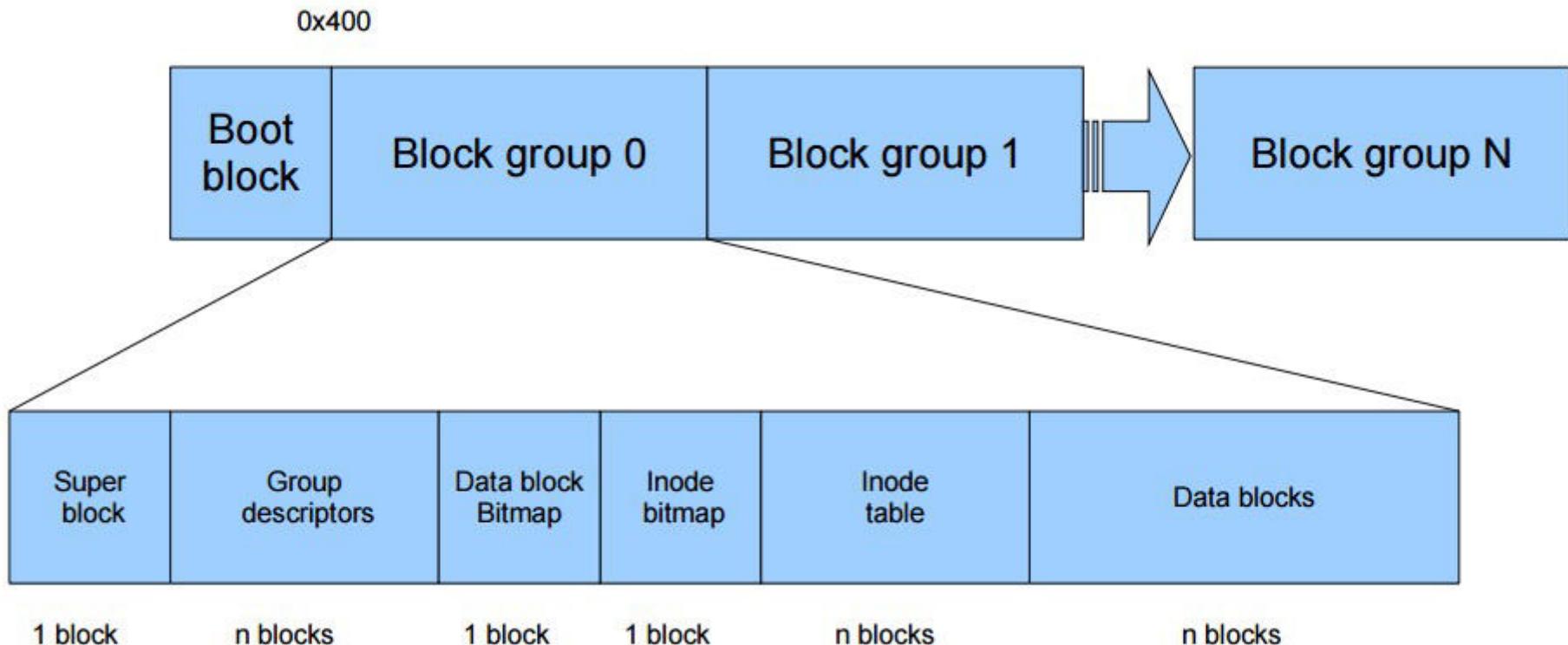
- **Problem**
  - Directory contains files and/or subdirectories
  - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
  - Directory needs to give location of each file on disk

# 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

# Ext2 FS layout

# Ext2 FS Layout



```
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 */
```

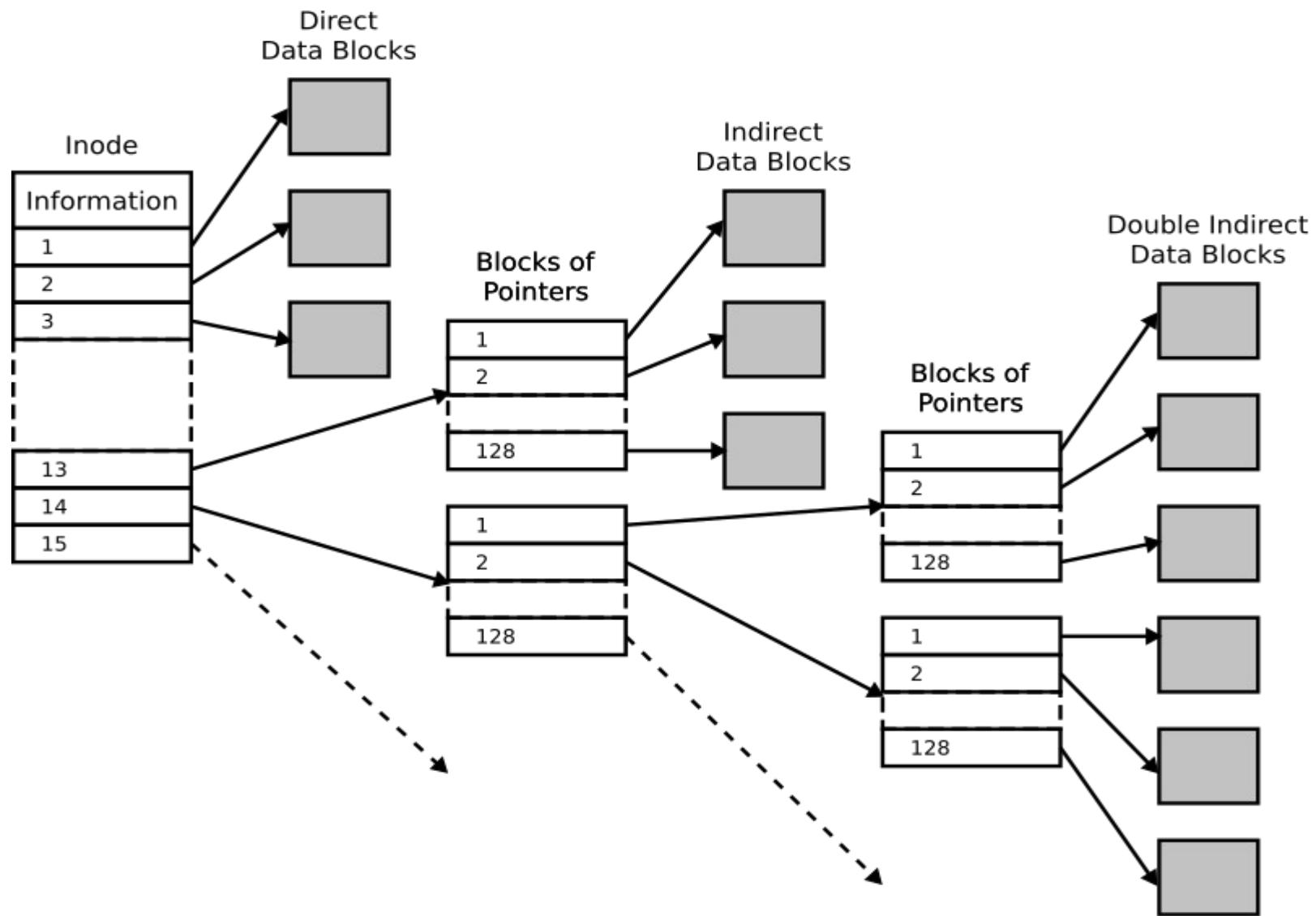
```
struct ext2_super_block {
...
 __le16 s_minor_rev_level; /* minor revision level */
 __le32 s_lastcheck; /* time of last check */
 __le32 s_checkinterval; /* max. time between checks */
 __le32 s_creator_os; /* OS */
 __le32 s_rev_level; /* Revision level */
 __le16 s_def_resuid; /* Default uid for 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];
};
```

```
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: Directory entry

|    | inode | rec_len | file_type | name_len | 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 |
| 40 | 67    | 28      | 3         | 2        | u    | s  | r  | \0 |   |    |    |
| 52 | 0     | 16      | 7         | 1        | o    | l  | d  | f  | i | 1  | e  |
| 68 | 34    | 12      | 4         | 2        | s    | b  | i  | n  |   |    |    |

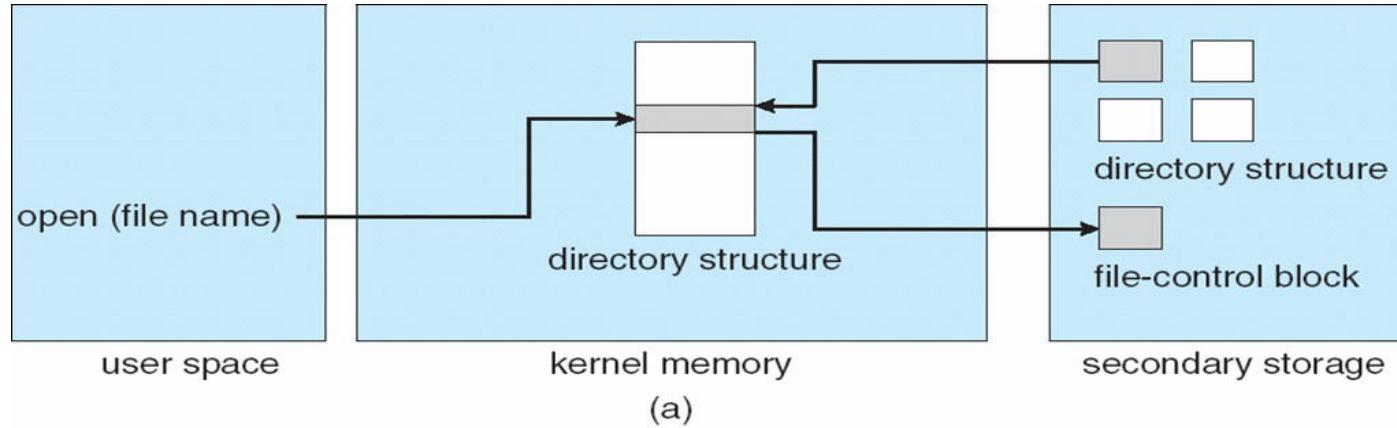
Let's see a program to read superblock of an ext2  
file system.

**Efficiency and Performance  
(and the risks created  
while trying to achieve it!)**

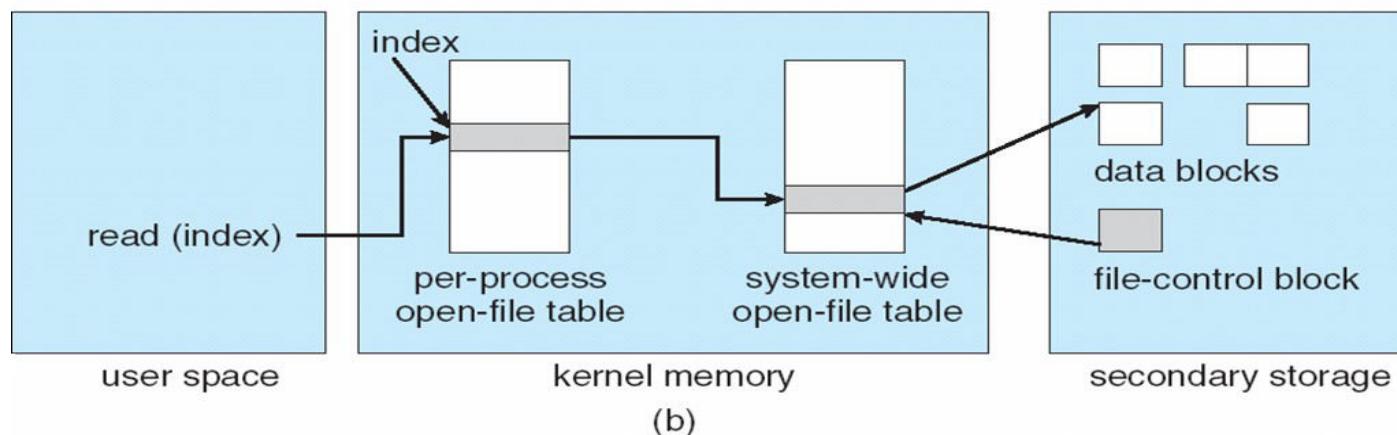
# In memory data structures

- **Mount table**
  - storing file system mounts, mount points, file system types
- **See next slide for “file” realated 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

# Efficiency

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
  -

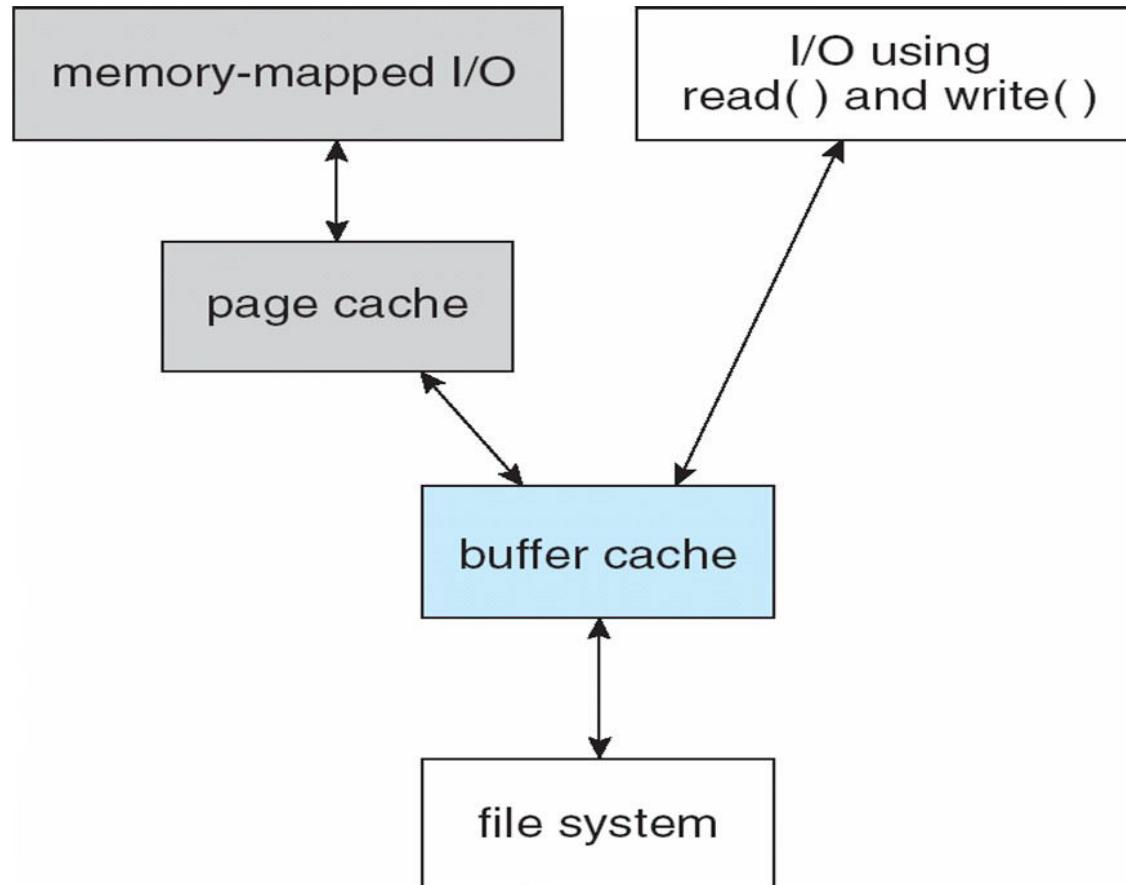
# Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

# Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

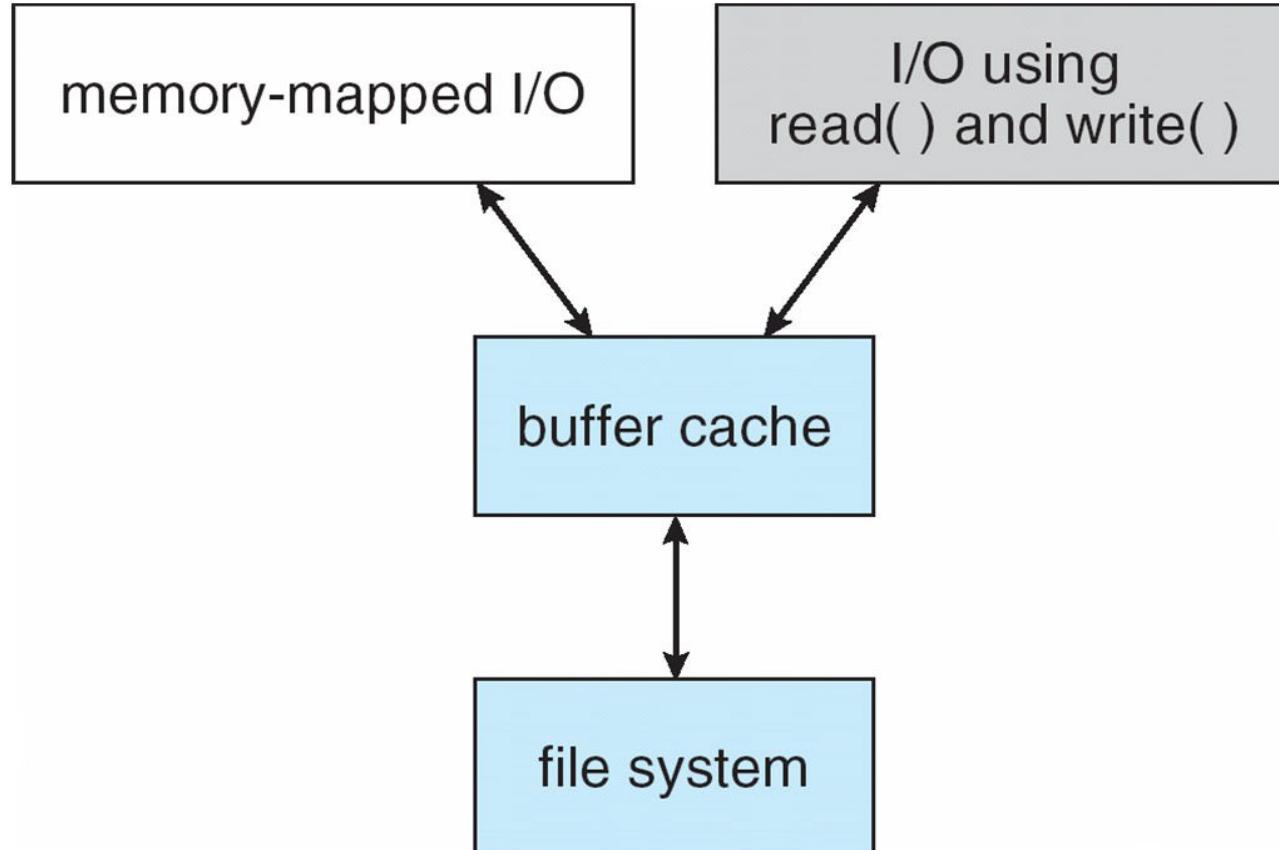
# I/O Without a Unified Buffer Cache



# Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

# I/O Using a Unified Buffer Cache



# Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

# Recovery

- **Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data from backup**

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

# Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!















# **File System Code**

**open, read, write, close, pipe, fstat, chdir, dup,  
mknod, link, unlink, mkdir,**

**Files, Inodes, Buffers**

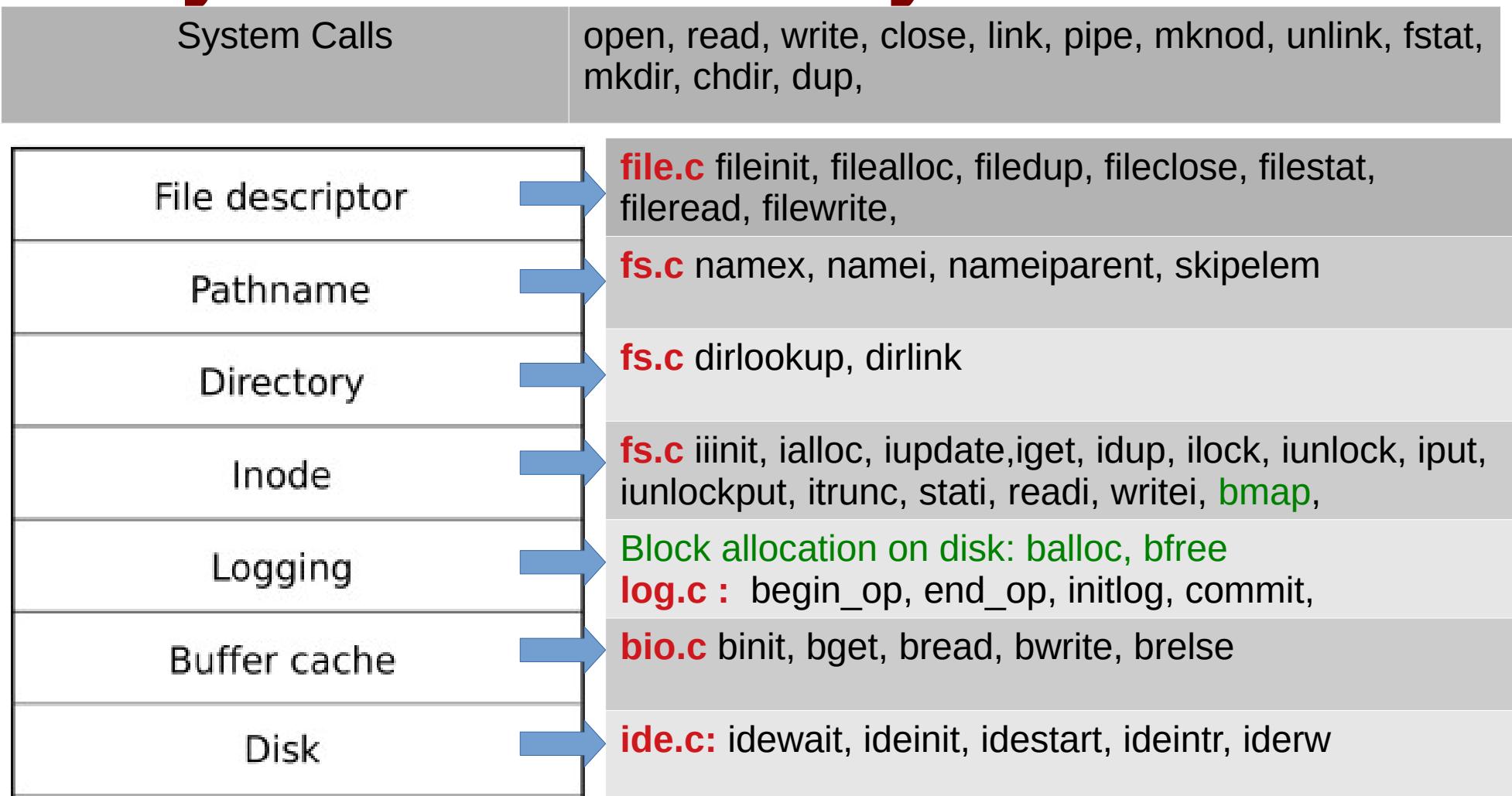
# What we already know

- **File system related system calls**
  - deal with ‘**fd**’ arrays (**ofile** in xv6). **open()** returns first empty index. **open** should ideally locate the **inode** on disk and initialize some data structures
  - maintain ‘**offsets**’ within a ‘file’ to support sequential read/write
  - **dup()** like system calls duplicate pointers in fd-array
  - read/write like system calls, going through ‘**ofile**’ array, should locate data of file on disk
  - We need functions to read/write from disk – that is **disk driver**
  - cache data of files in OS data structures for performance : **buffering**
  - Need to handle on disk data structures as well
- **Faster recovery (like journaling in ext3) is desired**

# xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartmentalized

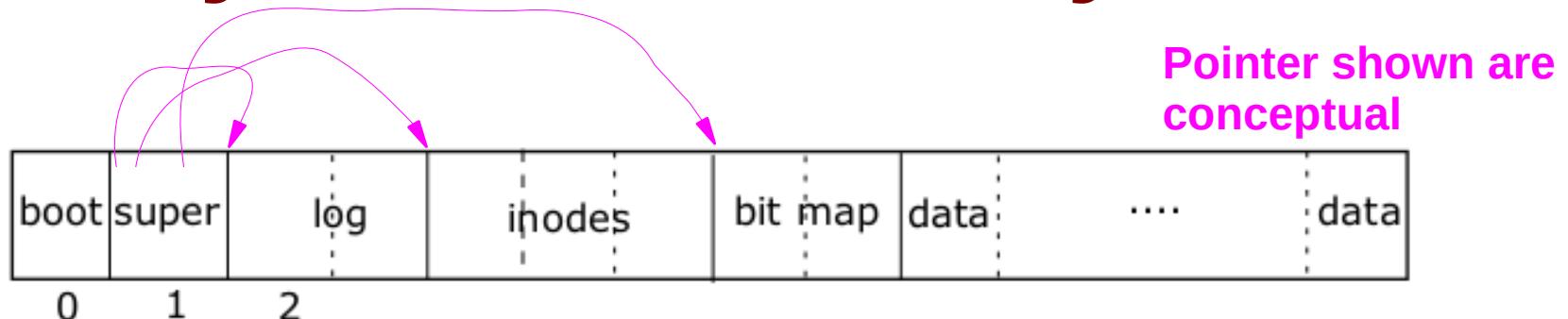
# Layers of xv6 file system code



Normally, any upper layer can call any lower layer below

**Abhijit: Block allocator should be considered as another Layer!**

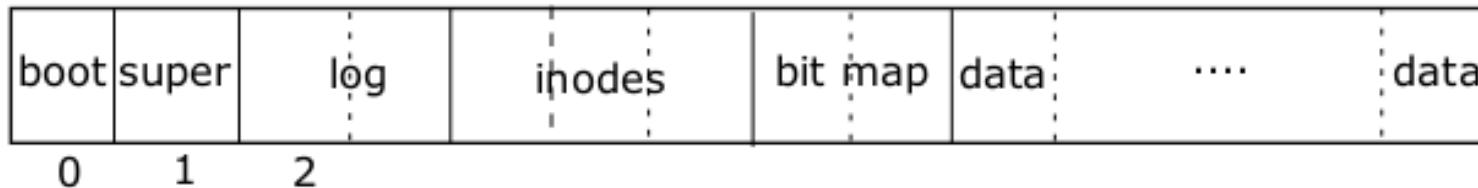
# Layout of xv6 file system



May see the code of `mkfs.c` to get insight into the layout

```
struct superblock {
 uint size; // Size of file system image (blocks)
 uint nblocks; // Number of data blocks
 uint ninodes; // Number of inodes.
 uint nlog; // Number of log blocks
 uint logstart; // Block number of first log block
 uint inodestart; // Block number of first inode block
 uint bmapstart; // Block number of first free map block
};
#define ROOTINO 1 // root i-number
#define BSIZE 512 // block size
```

# Layout of xv6 file system



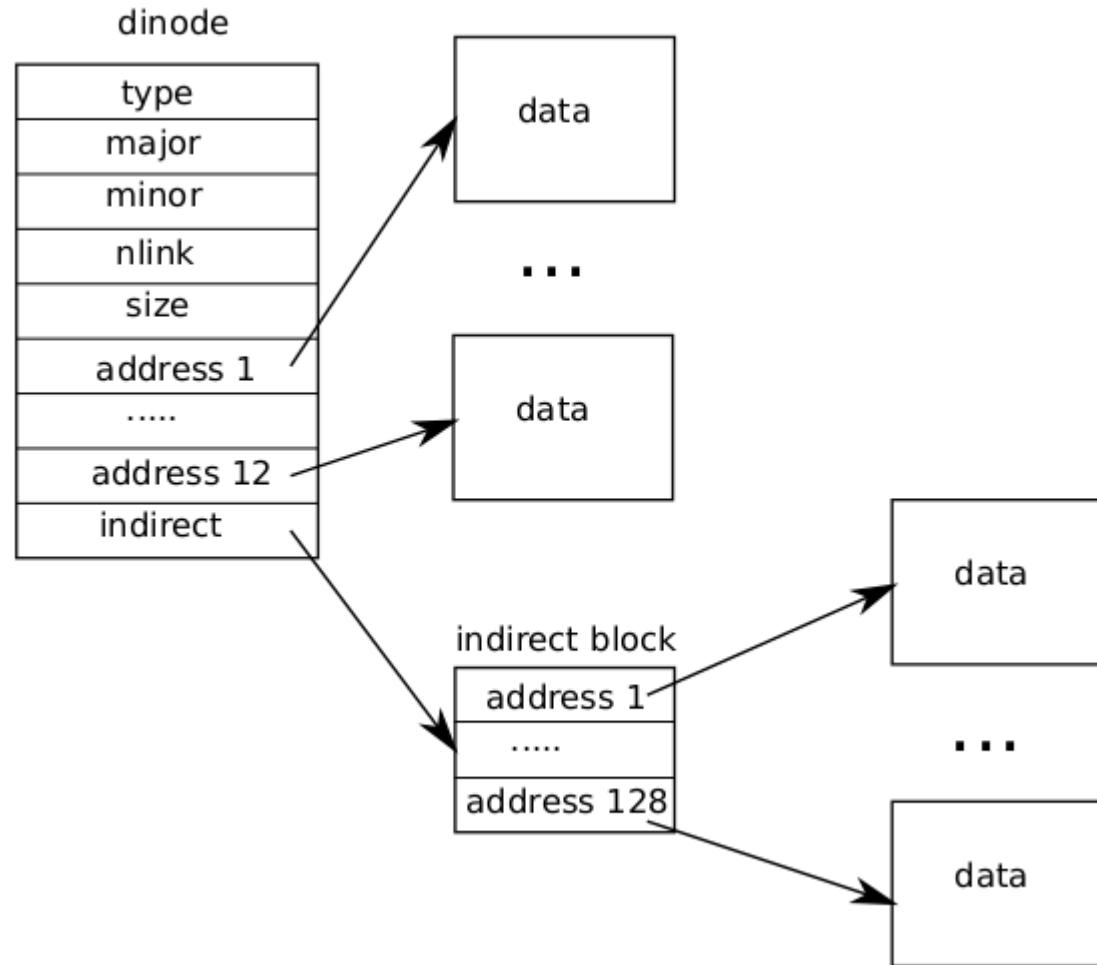
```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
 short type; // File type
 short major; // Major device number (T_DEV only)
 short minor; // Minor device number (T_DEV only)
 short nlink; // Number of links to inode in file system
 uint size; // Size of file (bytes)
 uint addrs[NDIRECT+1]; // Data block addresses
};


```

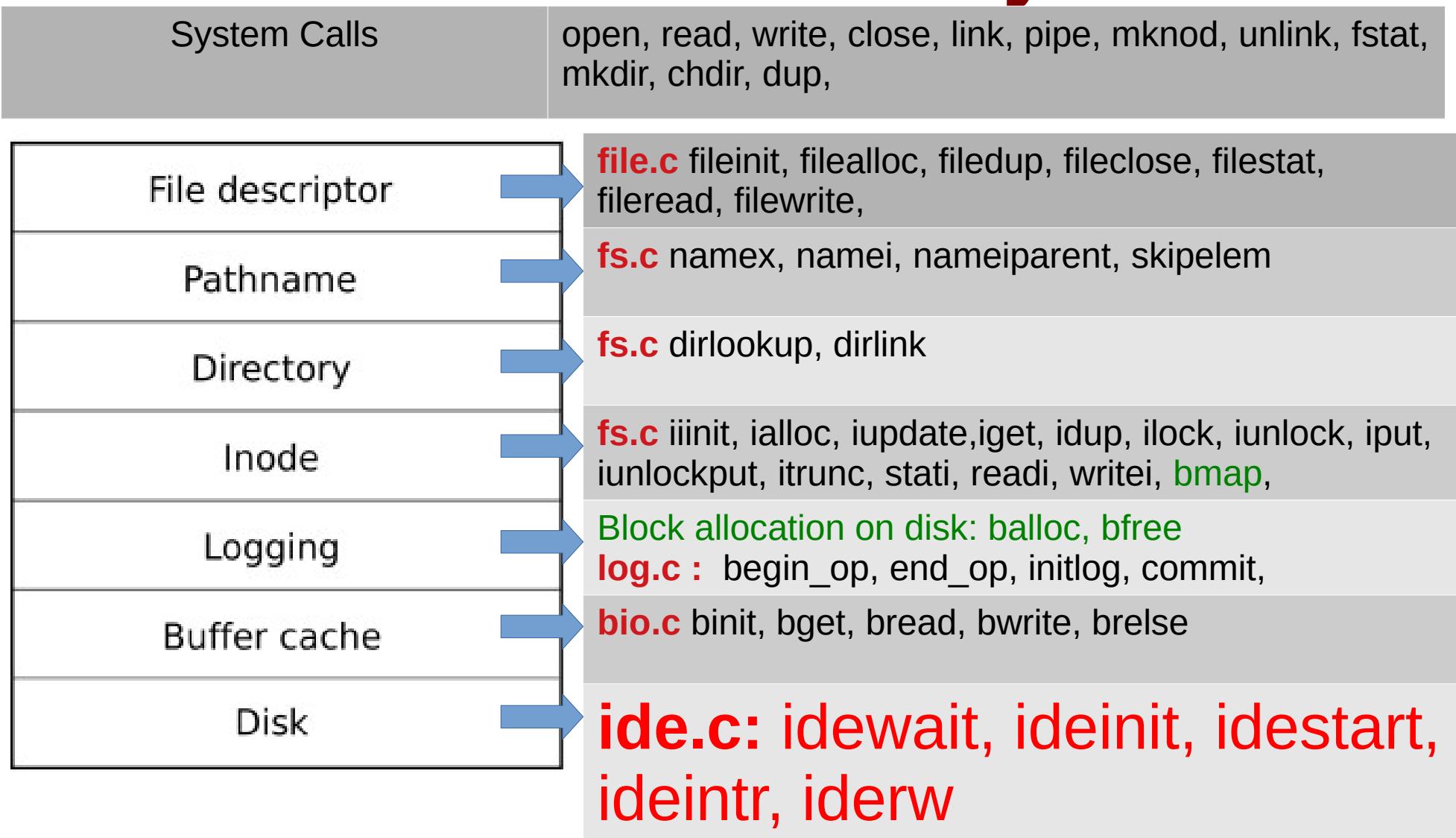
```
#define DIRSIZ 14
```

```
struct dirent {
 ushort inum;
 char name[DIRSIZ];
};
```

# File on disk



# Let's discuss lowest layer first



Normally, any upper layer can call any lower layer below

# **ide.c: idewait, ideinit, idestart, ideintr, iderw**

**static struct spinlock idelock;**

**static struct buf \*idequeue;**

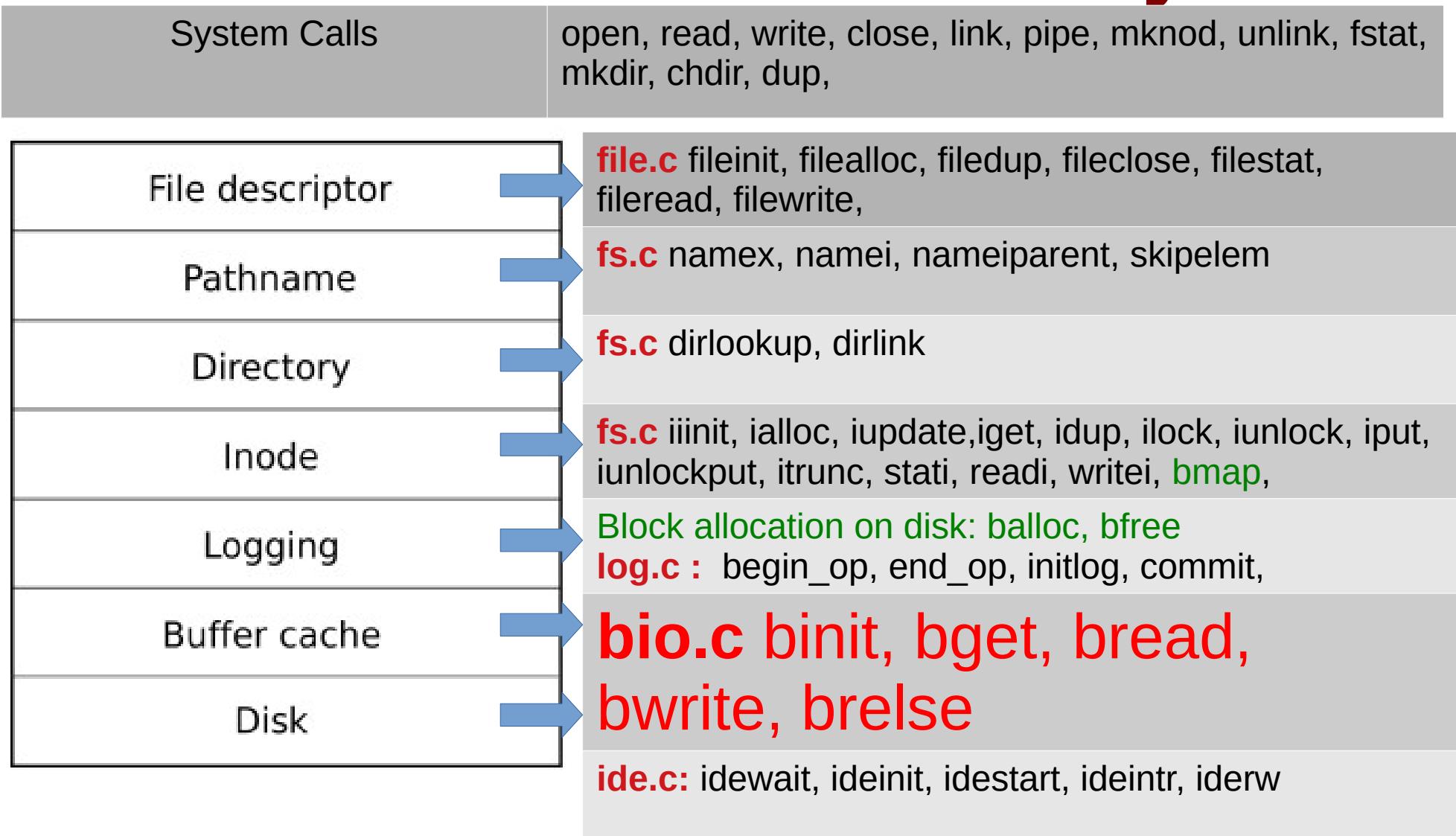
**static int havedisk1;**

- **ideinit**
  - was called from **main.c: main()**
  - Initialized IDE controller by writing to certain ports
  - **havedisk=1** setup
  - Initialize **idelock**
- **idewait**
  - **BUSY** loop waiting for IDE to be ready

# ide.c: idewait, ideinit, idestart, ideintr, iderw

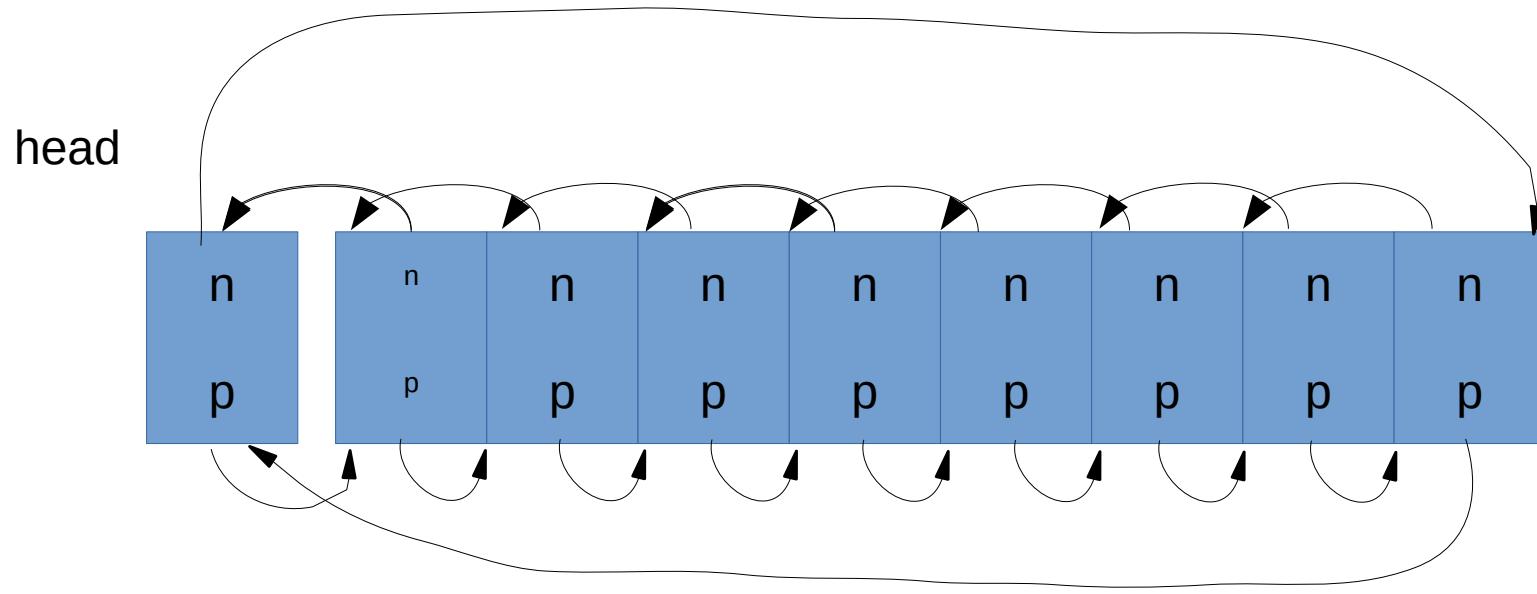
- **void idestart(buf \*b)**
  - **static void idestart(struct buf \*b)**
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on **idequeue**)
- **ideintr**
  - Take **idelock**. Called on IDE interrupt (through alltraps()->trap())
  - Wakeup the process waiting on first buffer in **buffer \*idequeue**;
  - call **idestart()**. Release **idelock**.
- **iderw(buf \*b)**
  - Move **buf b to end of idequeue**
  - Call **idestart()** if not running, sleep on **idelock**

# Let's see buffer cache layer



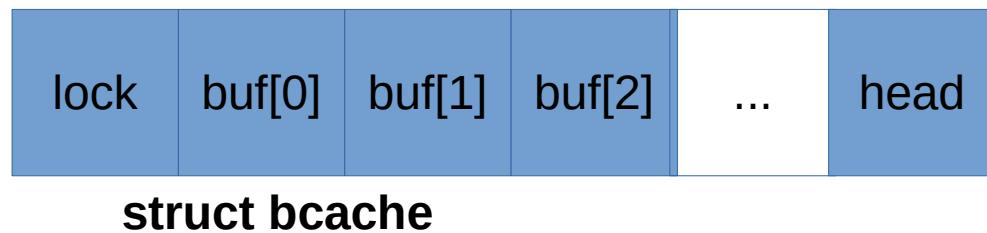
Normally, any upper layer can call any lower layer below

# Reminder: After main() -> binit()



Conceptually  
Linked lists  
like this

Buffers keep  
moving on  
list, as LRU



# struct buf

```
struct buf {
 int flags; // 0 or B_VALID or B_DIRTY
 uint dev; // device number
 uint blockno; // seq block number on device
 struct sleeplock lock; // Lock to be held by process using it
 uint refcnt; // Number of live accesses to the buf
 struct buf *prev; // cache list
 struct buf *next; // cache list
 struct buf *qnext; // disk queue
 uchar data[BSIZE]; // data 512 bytes
};
#define B_VALID 0x2 // buffer has been read from disk
#define B_DIRTY 0x4 // buffer needs to be written to disk
```

# buffer cache:

## static struct buf\* bget(uint dev, uint blockno)

- The **bcache.head** list is maintained on Most Recently Used (MRU) basis
  - **head.next** is the Most Recently Used (MRU) buffer
  - hence **head.prev** is the Least Recently Used (LRU)
- Look for a buffer with **b->blockno = blockno** and **b->dev = dev**
  - Search the **head.next** list for existing buffer (MRU order)
  - Else search the **head.prev** list for empty buffer
  - **panic()** if found in-use or empty buffer
- Increment **b->refcnt** ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

# buffer cache: **struct buf\* bread(uint dev, uint blockno)**

```
struct buf*
bread(uint dev, uint blockno)
{
 struct buf *b;
 b = bget(dev, blockno);
 if((b->flags & B_VALID) == 0) {
 iderw(b);
 }
 return b; // locked buffer
}
```

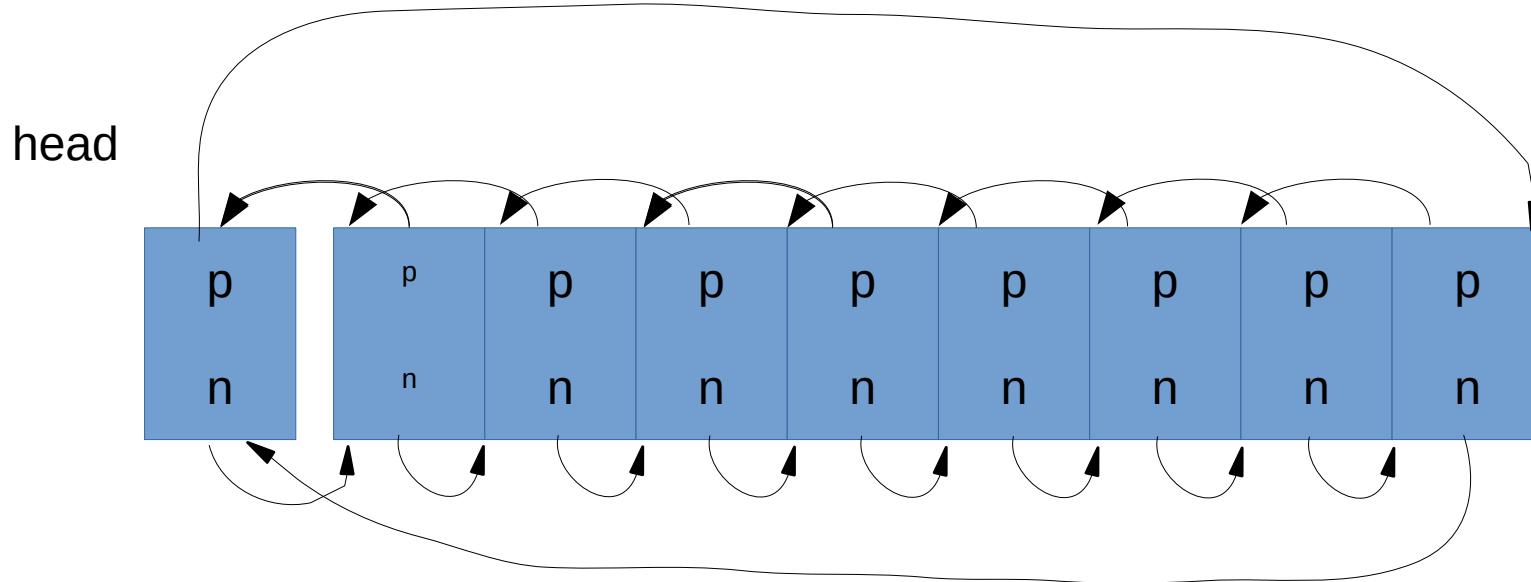
```
void
bwrite(struct buf *b)
{
 if(!holdingsleep(&b->lock))
 panic("bwrite");
 b->flags |= B_DIRTY;
 iderw(b);
}
```

Recollect: **iderw** moves buf to tail of **idequeue**, calls **idestart()** and **sleep()**

## buffer cache: **void brelse(struct buf \*b)**

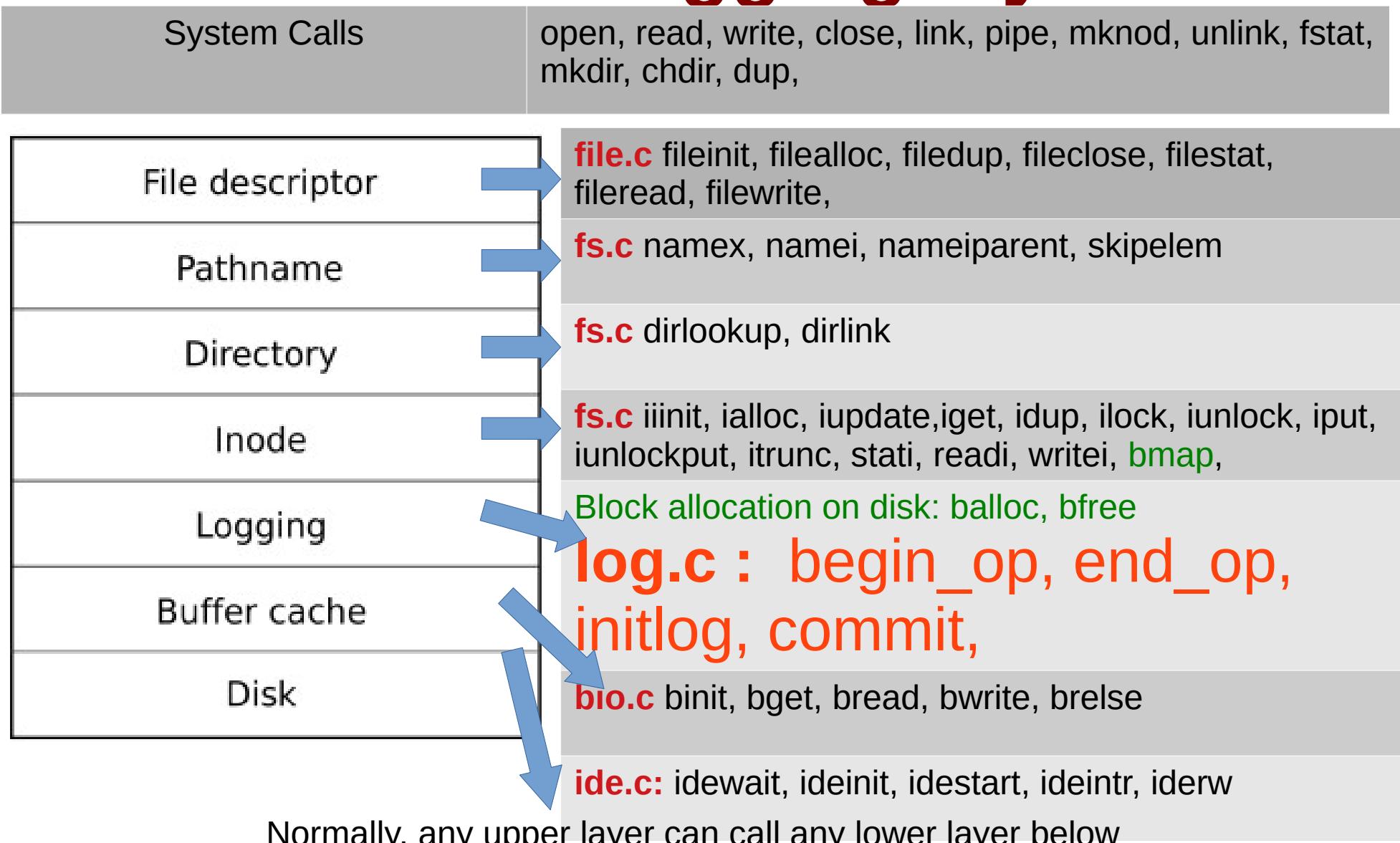
- **release lock on buffer**
- **b->refcnt = 0**
- **If b->refcnt = 0**
  - Means buffer will no longer be used
  - Move it to **front** of the **front** of **bcache.head**

# Overall in this diagram



Buffers keep moving to the front of the list and around  
The list always contains **NBUF=30** buffers  
**head.next** is always the MRU and **head.prev** is always LRU  
buffer

# Let's see logging layer



# log in xv6

- **a mechanism of recovery from disk**
- **Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)**
  - some writes succeed and some don’t
  - leading to inconsistencies on disk
- **In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all**
- **During recovery, completed operations can be “rerun” and incomplete operations neglected**

# log in xv6

- **xv6 system call does not directly write the on-disk file system data structures.**
- **A system call calls begin\_op() at beginning and end\_op() at end**
  - begin\_op() increments log.outstanding
  - end\_op() decrements log.outstanding, and if it's 0, then calls commit()
- **During the code of system call, whenever a buffer is modified, (and done with)**
  - log\_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in its actual place in FS as of now
- **when finally commit() is called, all modified blocks are copied to disk in the file system**

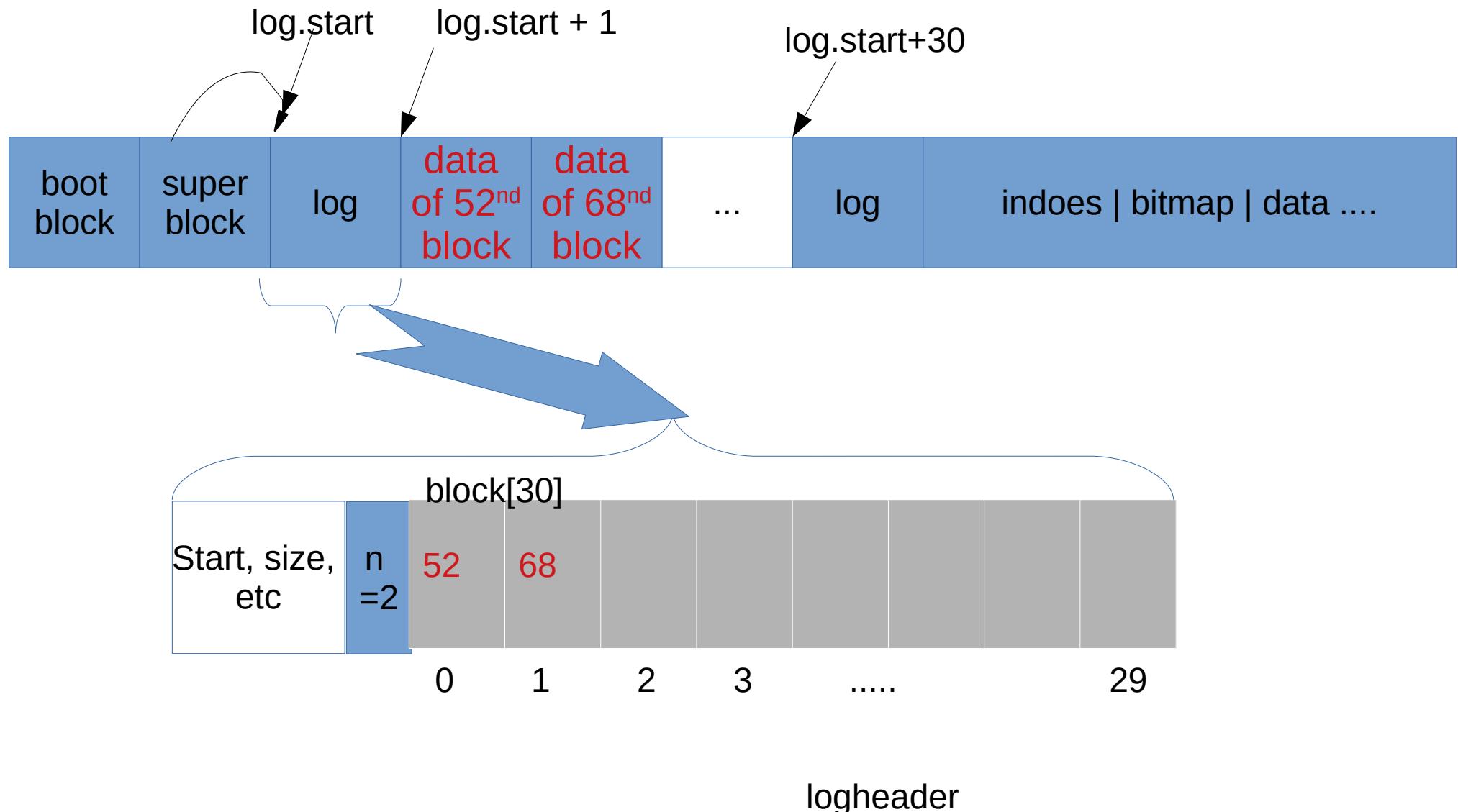
# log

```
struct logheader { // ON DISK
 int n; // number of entries in use in block[] below
 int block[LOGSIZE]; // List of block numbers stored
};

struct log { // only in memory
 struct spinlock lock;
 int start; // first log block on disk (starts with logheader)
 int size; // total number of log blocks (in use out of 30)
 int outstanding; // how many FS sys calls are executing.
 int committing; // in commit(), please wait.
 int dev; // FS device
 struct logheader lh; // copy of the on disk logheader
};

struct log log;
```

# log on disk



# Typical use case of logging

**/\* In a system call code \*/**

**begin\_op();**

...

**bp = bread(...);**

**bp->data[...] = ...;**

**log\_write(bp);**

...

**end\_op();**

prepare for logging. Wait if  
logging system is not ready or  
'committing'. ++outstanding

read and get access to a data  
block – as a buffer

modify buffer

note down this buffer for  
writing, in log. proxy for  
bwrite(). Mark B\_DIRTY. Absorb  
multiple writes into one.

Syscall done. write log and all  
blocks. --outstanding.

If outstanding = 0, commit().

# Example of calls to logging

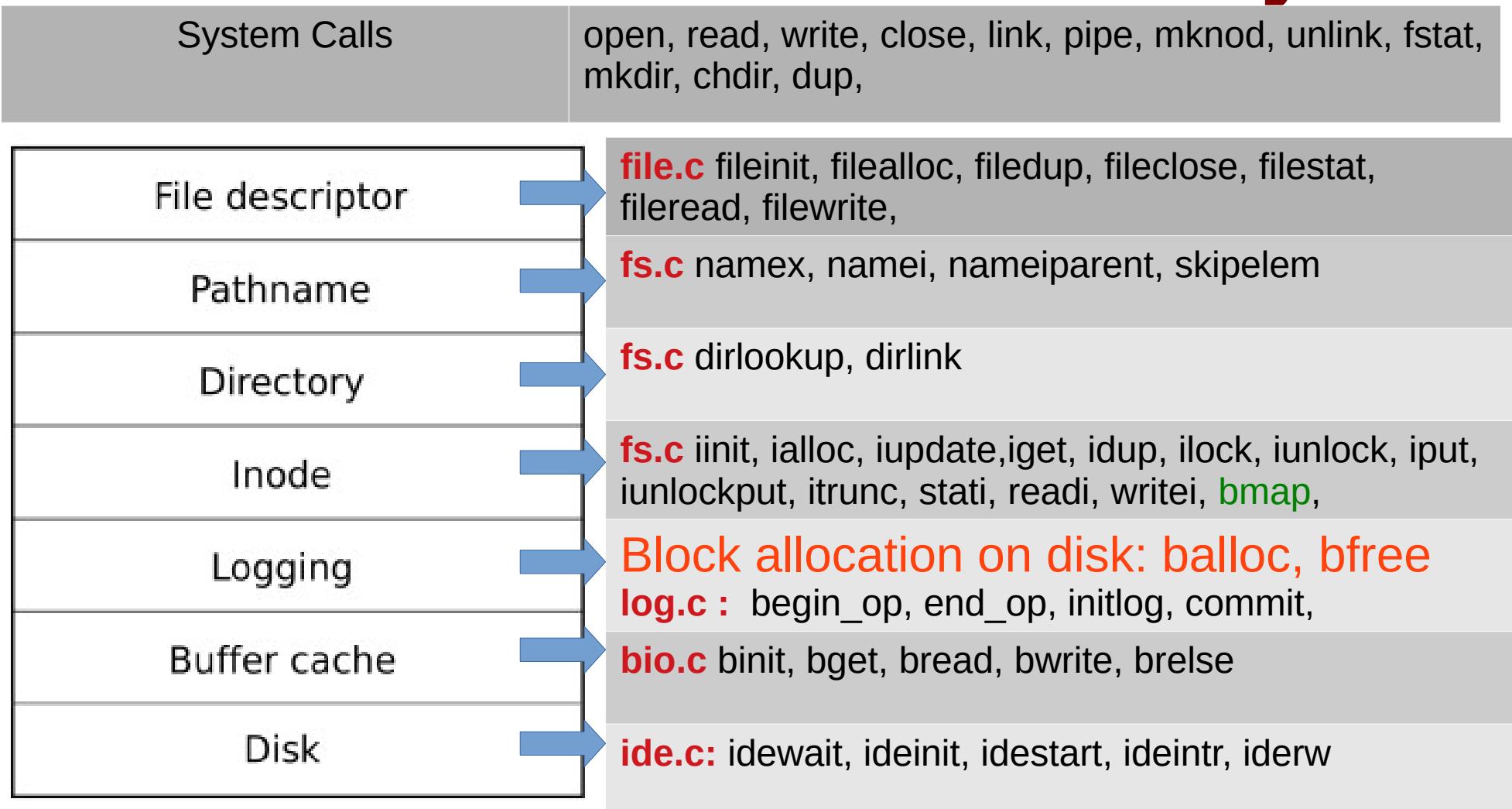
```
//file_write() code
begin_op();
ilock(f->ip);
/*loop */ r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

- each `writei()` in turn calls `bread()`, `log_write()` and `brelse()`
  - also calls `iupdate(ip)` which also calls `bread`, `log_write` and `brelse`
- Multiple writes are combined between `begin_op()` and `end_op()`

# Logging functions

- **Initlog()**
  - Set fields in global **log.xyz** variables, using FS superblock
  - Recovery if needed
  - **Called from first forkret()**
- **Following three called by FS code**
- **begin\_op(void)**
  - Increment **log.outstanding**
- **end\_op(void)**
  - Decrement **log.outstanding** and call **commit()** if it's zero
- **log\_write(buf \*)**
  - Remember the specified block number in **log.lh.block[]** array
  - Set the block to be dirty
- **write\_log(void)**
  - **Called only from commit()**
  - Use block numbers specified in **log.lh.block** and copy those blocks from memory to log-blocks
- **commit(void)**
  - **Called only from end\_op()**
  - **write\_log()**
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again

# Let's see block allocation layer



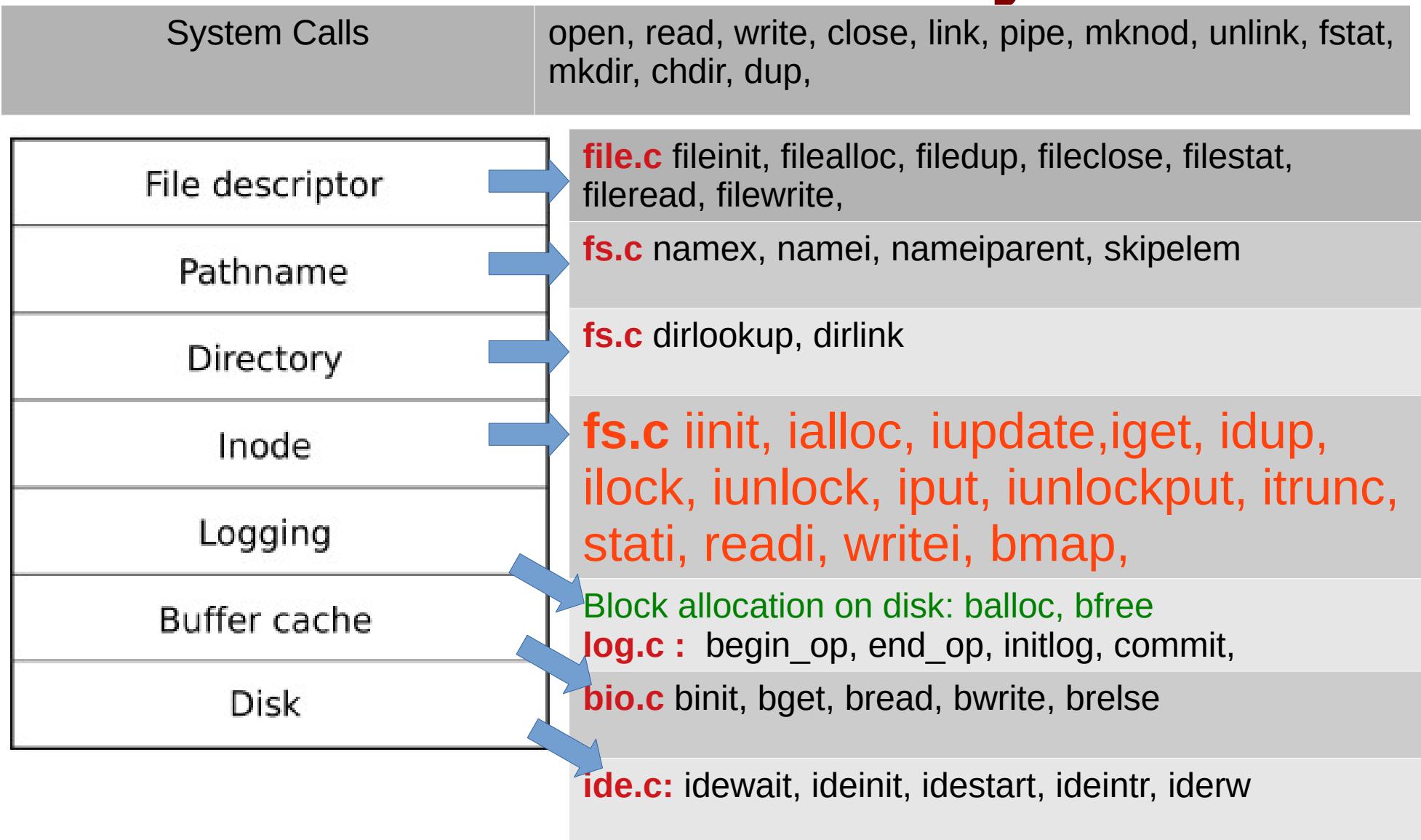
Normally, any upper layer can call any lower layer below

**Abhijit: Block allocator should be considered as another Layer!**

# allocating & deallocating blocks on DISK

- **balloc(devno)**
  - looks for a block whose bitmap bit is zero, indicating that it is free.
  - On finding updates the bitmap and returns the block.
  - **balloc()** calls **bread()->bget** to get a block from disk in a buffer.
    - Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.
  - Calls **log\_write(bp);**
    - Thus writes to bitmap blocks are also logged
- **bfree(devno, blockno)**
  - finds the right bitmap block and clears the right bit.
  - Also calls **log\_write()**

# Let's see Inode Layer



# On disk & in memory inodes

```
struct {
 struct spinlock lock;
 struct inode inode[NINODE];
} icache;

// On-disk inode structure
struct dinode {
 short type; // File type
 short major; // T_DEV Major device
 number
 short minor; // Minor device number
 short nlink; // Number of links
 uint size; // Size of file (bytes)
 uint addrs[NDIRECT+1]; /
};
```

```
// in-memory copy of an inode
struct inode {
 uint dev; // Device number
 uint inum; // Inode number
 int ref; // Reference count
 struct sleeplock lock; // protects
 everything below here
 int valid; // been read from disk?

 short type; // copy of disk inode
 short major;
 short minor;
 short nlink;
 uint size;
 uint addrs[NDIRECT+1];
};
```

# In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The **iget** and **iput** functions acquire and release pointers to an inode, modifying the **ref count**.
- See the caller graph of **iget()**
  - all those who call **iget()**
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

# iget and iupdate

- **iget**
  - **searches for an existing/free inode in icache and returns pointer to one**
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock
- **iupdate(inode \*ip)**
  - **read on disk block of inode**
  - **get on disk inode**
  - **modify it as specified in ‘ip’**
  - **modify disk block of inode**
  - **log\_write(disk block of inode)**

# itrunc , iput

- **iput(ip)**
  - if ref is 1
    - **itrunc(ip)**
    - **type = 0**
    - **iupdate(ip)**
    - **i->valid = 0 // free in memory**
  - **else**
    - **ref--**
- **itrunc(ip)**
  - **write all data blocks of inode to disk**
    - **using bfree()**
  - **ip->size = 0**
    - **Inode is freed from use**
  - **iupdate(ip)**
  - **called from iput() only when 'ref' becomes zero**

# race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
  - and won't be prepared to find the inode is not longer allocated
- This is not possible. Why?
  - no way for a syscall to get a ref to a inode with ip->ref = 1

```
void
iput(struct inode *ip)
{
 acquiresleep(&ip->lock);
 if(ip->valid && ip->nlink == 0){
 acquire(&icache.lock);
 int r = ip->ref;
 release(&icache.lock);
 if(r == 1){
 // inode has no links and no other
 references: truncate and free.
 itrunc(ip);
 }
 }
}
```

# buffer and inode cache

- to read an **inode**, its block must be read in a buffer
- So the buffer always contains a copy of the on-disk dinode
  - duplicate copy in in-memory **inode**
- The inode cache is write-through,
  - code that modifies a cached inode must immediately write it to disk with **iupdate**
- Inode may still exist in the buffer cache

# allocating inode

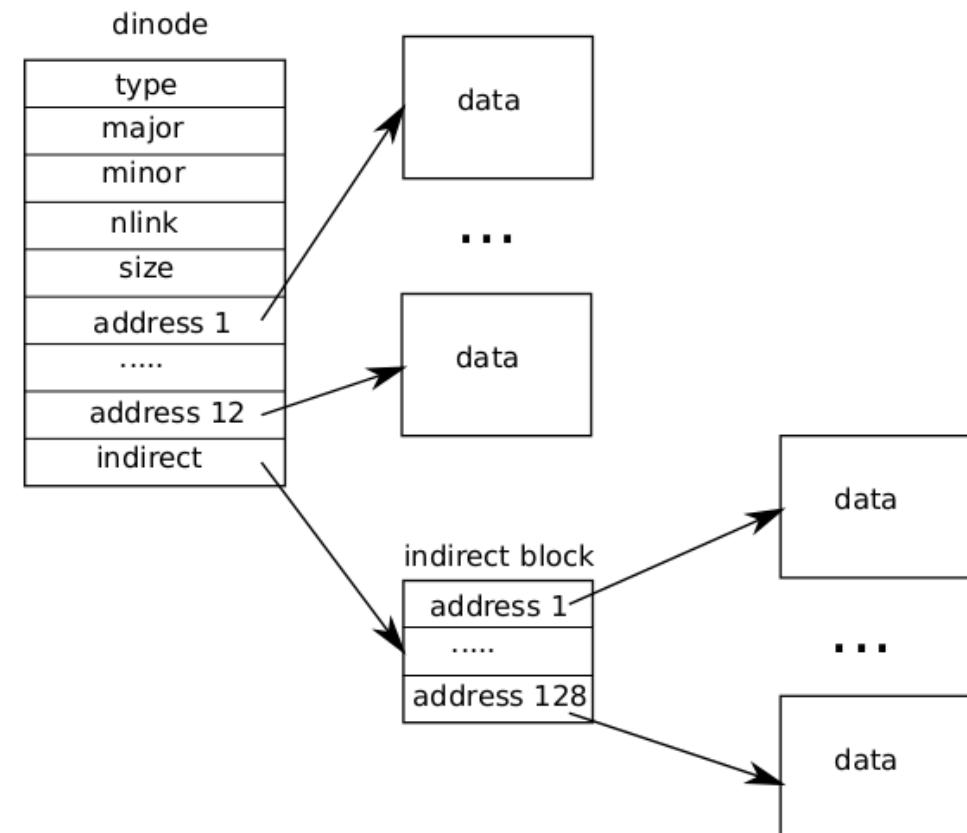
- **ialloc(dev, type)**
  - Loop over all disk inodes
  - read inode (from its block)
  - if it's free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- **ilock**
  - code must acquire ilock before using inode's data/fields
  - **Ilock reads inode if it's already not in memory**

# Trouble with iput() and crashes

- **iput() doesn't truncate a file immediately when the link count for the file drops to zero, because**
  - some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.
- **if a crash happens before the last process closes the file descriptor for the file,**
  - then the file will be marked allocated on disk but no directory entry points to it
- **Unsolved problem.**
- **How to solve it?**

# Get Inode data: bmap(ip, bn)

- **Allocate ‘bn’th block for the file given by inode ‘ip’**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**
  - **allocate block of indirect entries if needed using balloc()**



# writing/reading data at a given offset in file

**readi(struct inode \*ip,  
char \*dst, uint off, uint  
n)**

**writei(struct inode \*ip,  
char \*src, uint off, uint  
n)**

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
- using bread(), brelse()
- Call devsw.read if inode is a device Inode.
- Writei() also updates size if required

# Reading Directory Layer

|                 |                                                                                                                             |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------|
| System Calls    | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                              |
| File descriptor | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                       |
| Pathname        | <b>fs.c</b> namex, namei, nameiparent, skipelem                                                                             |
| Directory       | <b>fs.c</b> dirlookup, dirlink                                                                                              |
| Inode           | <b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, unlock, iput, unlockput, itrunc, stati, readi, writei, <b>bmap</b> , |
| Logging         | Block allocation on disk: <b>ballo</b> , <b>bfree</b><br><b>log.c</b> : begin_op, end_op, initlog, commit,                  |
| Buffer cache    | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                             |
| Disk            | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                                   |

# directory entry

```
#define DIRSIZ 14
```

```
struct dirent {
 ushort inum;
 char name[DIRSIZ];
};
```

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

**struct inode\***  
**dirlookup(struct inode \*dp, char \*name, uint \*poff)**

- **Given a pointer to directory inode (dp), name of file to be searched**
  - return the pointer to inode of that file (NULL if not found)
  - set the ‘offset’ of the entry found, inside directories data blocks, in poff
- **How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?**
  - During resolution of pathnames?
- **Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num**

int  
**dirlink(struct inode \*dp, char \*name, uint inum)**

- **Create a new entry for ‘name’ \_ ‘inum’ in directory given by ‘dp’**
  - inode number must have been obtained before calling this. How to do that?
- **Use dirlookup() to verify entry does not exist!**
- **Get empty slot in directory’s data block**
- **Make directory entry**
- **Update directory inode! writei()**

# namex

- **Called by namei(), or nameiparent()**
- **Just iteratively split a path using “/” separator and get inode for last component**
- **iget() root inode, then**
- **Repeatedly calls**
  - **split on “/”, dirlookup() for next component**
-

# races in namex()

- **Crucial. Called so many times!**
- **one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink**
  - when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- **Deadlock? next points to the same inode as ip when looking up "..". Locking next before releasing the lock on ip would result in a deadlock.**
  - namex unlocks the directory before obtaining a lock on next.

# File descriptor layer code

| System Calls    | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                                                                                          |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| File descriptor | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                                                                                   |
| Pathname        | <b>fs.c</b> namex, namei, nameiparent, skipelem                                                                                                                                         |
| Directory       | <b>fs.c</b> dirlookup, dirlink                                                                                                                                                          |
| Inode           | <b>fs.c</b> iiinit, ialloc, iupdate, igeet, idup, ilock, unlock, iput, unlockput, itrunc, stati, readi, writei, <b>bmap</b> ,<br>Block allocation on disk: <b>balloc</b> , <b>bfree</b> |
| Logging         | <b>log.c</b> : begin_op, end_op, initlog, commit,                                                                                                                                       |
| Buffer cache    | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                                                                                         |
| Disk            | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                                                                                               |

# data structures related to “file” layer

```
struct file {
 enum { FD_NONE, FD_PIPE,
FD_INODE } type;
 int ref; // reference count
 char readable;
 char writable;
 struct pipe *pipe; // used only if it
works as a pipe
 struct inode *ip;
 uint off;
};
// interesting no lock in struct file !
```

```
struct proc {
 ...
 struct file *ofile[NOFILE]; // Open files
per process
 ...
}
struct {
 struct spinlock lock;
 struct file file[NFILE];
} ftable; //global table from which ‘file’
is allocated to every process
Lock is used to protect updates to
every entry in the array
```

# Multiple processes accessing same file.

- **Each will get a different ‘struct file’**
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- **File can be a PIPE (more later)**
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!
- **ref**
  - used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

# file layer functions

- **filealloc**
  - find an empty struct file in ‘ftable’ and return it
  - set ref = 1
- **filedup(file \*)**
  - simply ref++
- **fileclose**
  - --ref
  - if ref = 0
    - free struct file
    - iput() / pipeclose()
    - note – transaction if iput() called
- **filestat**
  - simply return fields from inode, after holding lock. on inodes for files only.

# file layer functions

- **fileread**
  - call `readi()` or `piperead()`
  - `readi()` later calls device-  
read or inode read (using  
`bread()`)
- **filewrite**
  - call `pipewrite()` or `writei()`
  - `writei()` is called in a loop,  
within a transaction
- **Why does readi()  
call read on the  
device , why not  
fileread() itself call  
device read ?**

# pipes

```
struct pipe {
 struct spinlock lock;
 char data[PIPE_SIZE];
 uint nread;
 // number of bytes read
 uint nwrite;
 // number of bytes written
 int readopen;
 // read fd is still open
 int writeopen;
 // write fd is still open
};
```

- **functions**
  - pipealloc
  - pipeclose
  - piperead
  - pipewrite
-

# pipes

- **pipealloc**
  - allocate two struct file
  - allocate pipe itself using kalloc (it's a big structure with array)
  - init lock
  - initialize both struct file as 2 ends (r/w)
- **pipewrite**
  - wait if pipe full
  - write to pipe
  - wakeup processes waiting to read
- **piperead**
  - wait if no data
  - read from pipe
  - wakeup processes waiting to write
- **Good producer consumer code !**

# **Further to reading system call code now**

- Now we are ready to read the code of system calls on file system
  - sys\_open, sys\_write, sys\_read , etc.
- Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.
- Also think of locks that need to be held.

# Synchronization

# My formulation

- OS = data structures + 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  $c_1 + c_2$ , 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 parallelly (not concurrently)**
- **What about non-interruptible kernel on multiprocessor systems?**
- **What about non-interruptible kernel on uniprocessor systems?**

# Critical Section Problem

- Consider system of n processes {p<sub>0</sub>, p<sub>1</sub>, ... p<sub>n-1</sub>}
- 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

# Critical Section problem

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

Figure 6.1 General structure of a typical process  $P$ .

# Expected solution characteristics

- **1. Mutual Exclusion**
  - If process  $P_i$  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);
```

# 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

# **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) {
 spinunlock(sl);
 currprocess->state = WAITING
 move current process to m->q
 Sched();
 spinlock(sl);
}

release(mutex *m) {
 spinlock(m->spinlock);
 m->islocked = 0;
 Some process in m->queue =RUNNABLE;
 spinunlock(m->spinlock);
}
```

# 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 lk;
```

```
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;
```

```

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.
};


```

# Spinlock on xv6

```

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();

}

```

```

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;
}

```

# spinlocks

- **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**

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();

}

# spinlocks

- **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

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);

}

# sleep()

- 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
- Here ptable lock is acquired but even then it is blocked , violation of thumb rule. But it is held until yield()

# 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);

```

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;

}

```

# Wakeup()

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

# 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)

# 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 semantics of the semaphore.

# Semaphore implementation? - 1

```
struct semaphore {
 int val;
 spinlock lk;
};

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(semaphore *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; spinunlock;

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

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

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

circular wait. Deadlock.

None of them will proceed.

# Semaphore implementation? - 2

```
struct semaphore {
 int val;
 spinlock lk;
};

sem_init(semaphore *s, int initval) {
 s->val = initval;
 s->sl = 0;
}

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

# Semaphore implementation? - 3b

```
struct semaphore {
 int val;
 spinlock lk;
 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(semaphore *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 going to change the
s->sl;
```

# Semaphore implementation? - 3c

```
struct semaphore {
 int val;
 spinlock lk;
 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(semaphore *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;
}
....
```

# **Different uses of semaphores**

# For mutual exclusion

**/\*During initialization\*/**

**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 */
```

# **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 programme 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
- 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

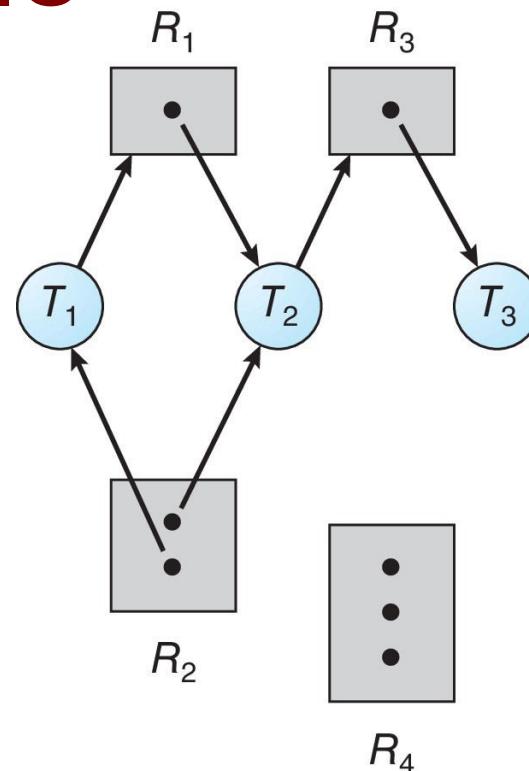
- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$
- CPU cycles, memory space, I/O devices
- Resource: Most typically a lock, synchronization primitive
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

# Deadlock characterisation

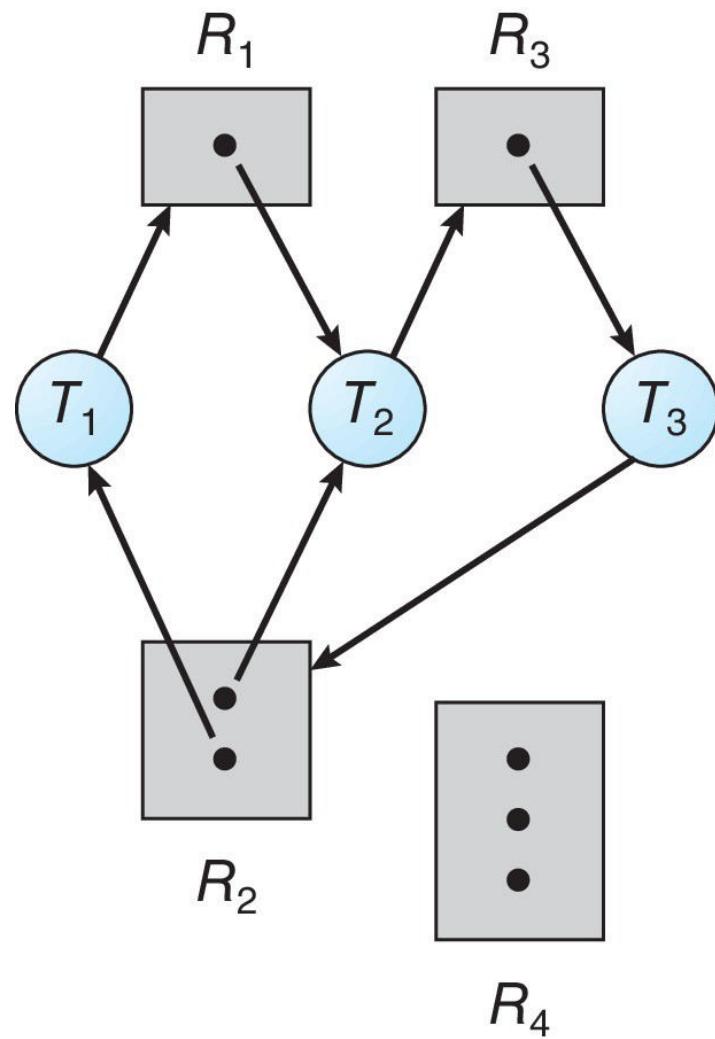
- Deadlock is possible only if ALL of these conditions are TRUE at the same time
- Mutual exclusion: only one process at a time can use a resource
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- Circular wait: there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Resource Allocation Graph Example

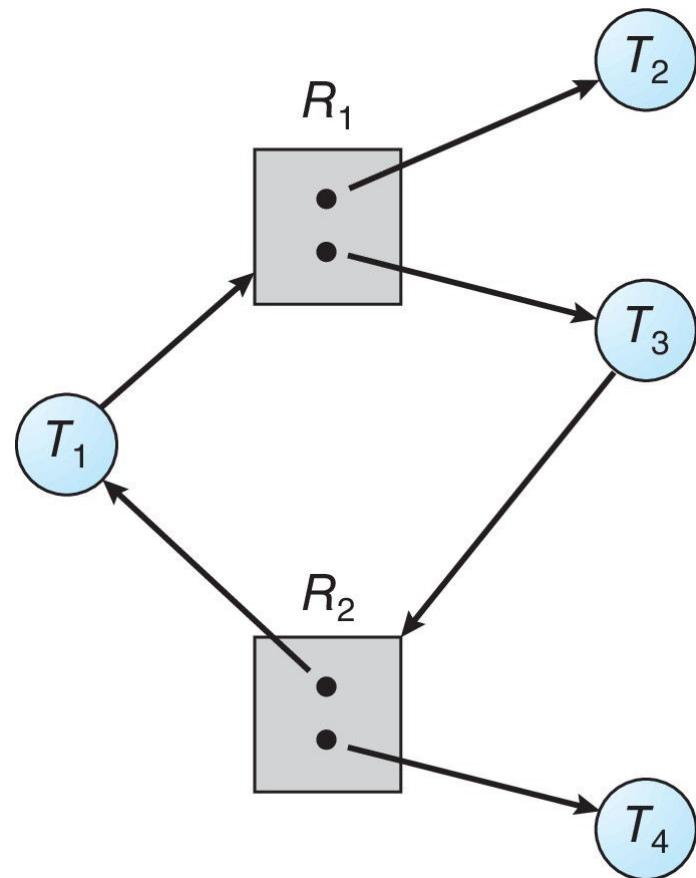
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holding one instance of R3



# 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 = 1
  - second\_mutex = 5
- code for thread\_two could not be written as follows:

```
/* 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 log lock, idelock, and ptable.lock.**

## (2) Deadlock avoidance

- Requires that the system has some additional a priori information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

## (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 all threads 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

## 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

# **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 nPendingReads;
```

```
 int nPendingWrites;
```

```
 spinlock_t sl;
```

```
 condition canRead;
```

```
 condition canWrite;
```

```
};
```

```
void lockShared (struct rwlock *r)
(
 spin_lock (&r->sl);
 r->nPendingReads++;
 if (r->nPendingWrites > 0)
 wait (&r->canRead, &r->sl); /*don't starve
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) void lockExclusive (struct rwlock *r)
{
 spin_lock (&r->sl);
 r->nActive--;
 if (r->nActive == 0) {
 spin_unlock (&r->sl);
 do signal (&r->canWrite);
 } else
 spin_unlock (&r->M);
}

(
 spin_lock (&r->sl);
 r->nPendingWrtes++;
 while (r->nActive)
 wait (&r->canWrite, &r->sl);
 r->nPendingWrites--;
 r->nActive = -1;
 spin_unlock (&r->sl);
}
```

# Code for reader-writer locks

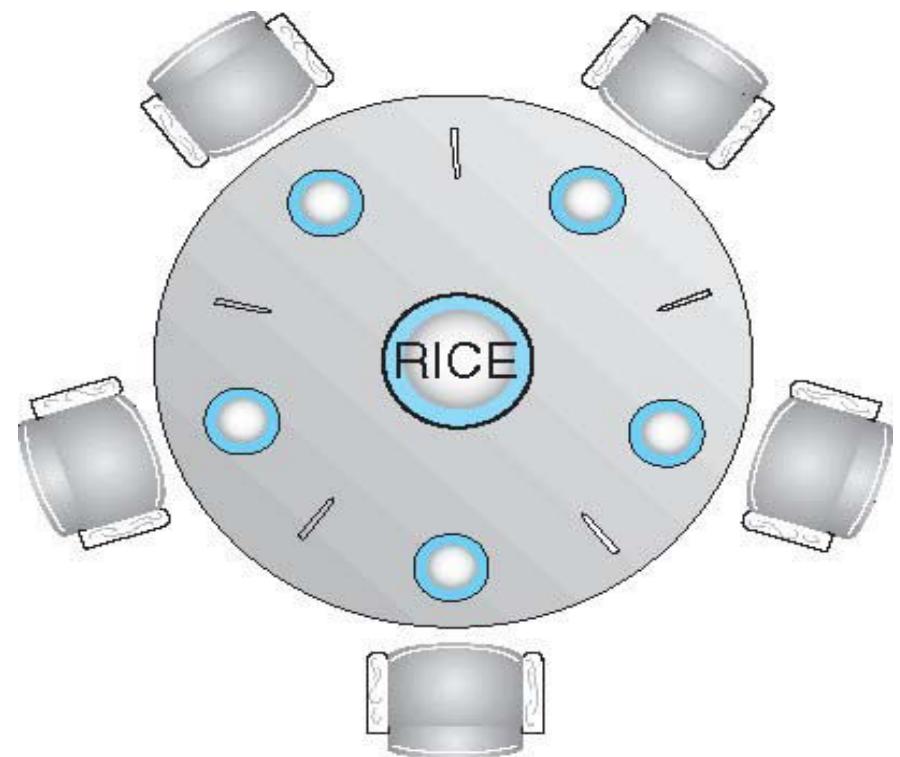
```
void unlockExclusive (struct rwlock *r){
 boolean t wakeReaders;
 spin_lock (&r->sl);
 r->nActive = 0;
 wakeReaders = (r->nPendingReads != 0);
 spin_unlock (&r->sl);
 if (wakeReaders)
 do broadcast (&r->canRead); /* wake
allreaders */
 else
 do_signal (&r->canWrite);
/*wakeasinglewrir */
}
```

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

- 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 sleep(), 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.

# Lost Wakeup problem

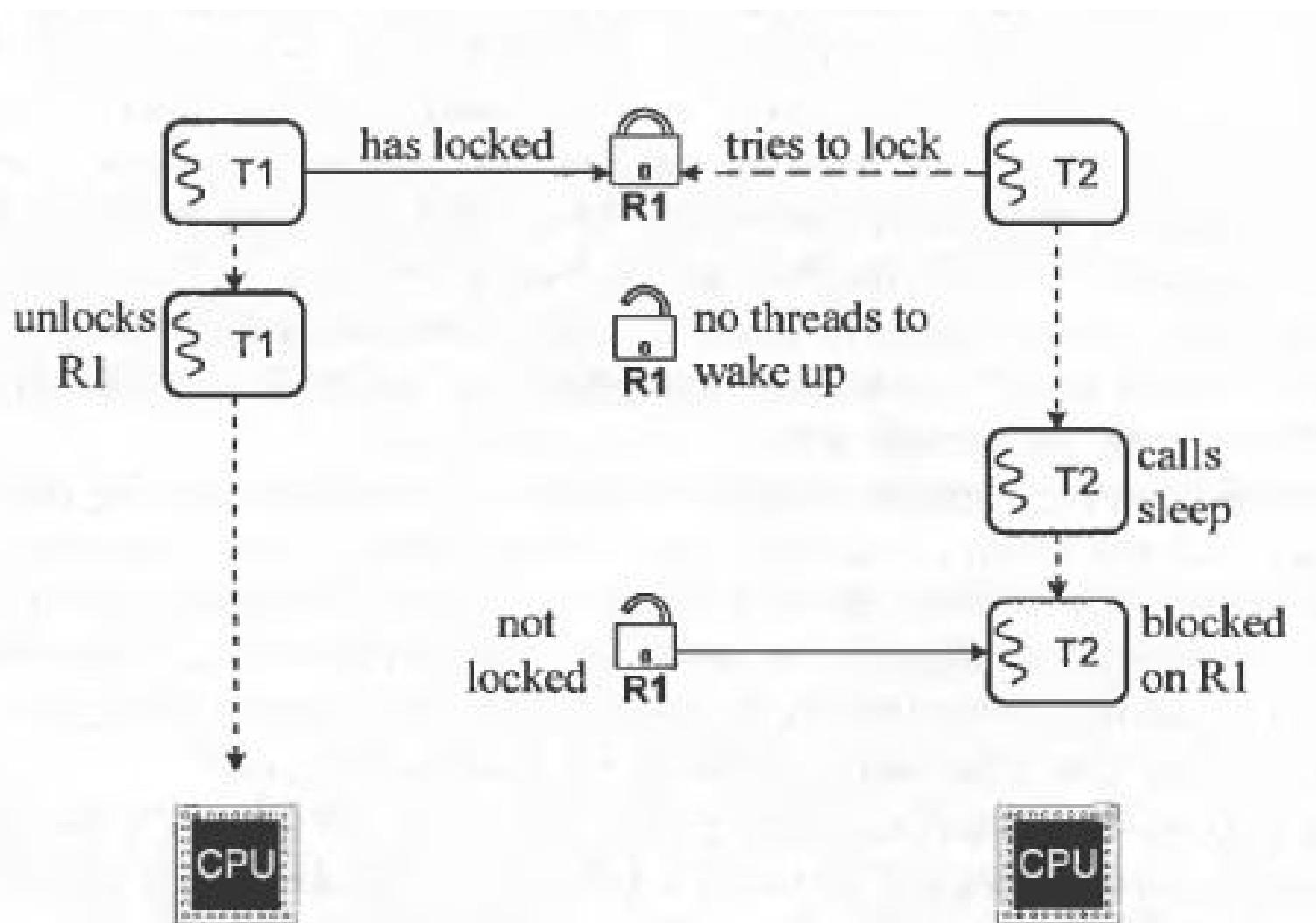


Figure 7-6. The lost wakeup problem.

# 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;**

| <i>Atomic Operation</i>        | <i>Effect</i>          |
|--------------------------------|------------------------|
| 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 inefficient
- 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/disabling 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 one 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 --> Trouble!
- 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!**

# Scheduling

Abhijit A.M.

**abhijit.comp@coep.ac.in**

Credits: Slides from os-book.com

# Necessity of scheduling

## Multiprogramming

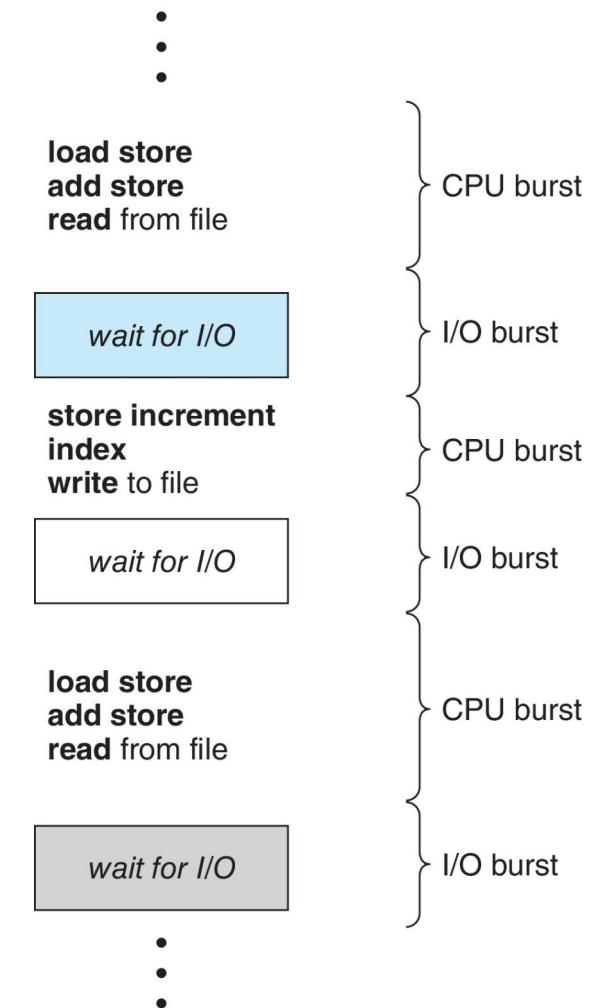
- Ability to run multiple programs at a time
- Increase use of CPU
  - CPU utilisation
-

# CPU Scheduling

- The task of selecting ‘next’ process/thread to execute on CPU and doing a context switch
- Scheduling algorithm
  - Criteria for selecting the ‘next’ process/thread and it’s implementation
- Why is it important?
  - Affects performance !
  - Affects end user experience !
  - Involves money!

# Observation: CPU, I/O Bursts

- Process can ‘wait’ for an event (disk I/O, read from keyboard, etc.)
- During this period another process can be scheduled
- CPU–I/O Burst Cycle:
- Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



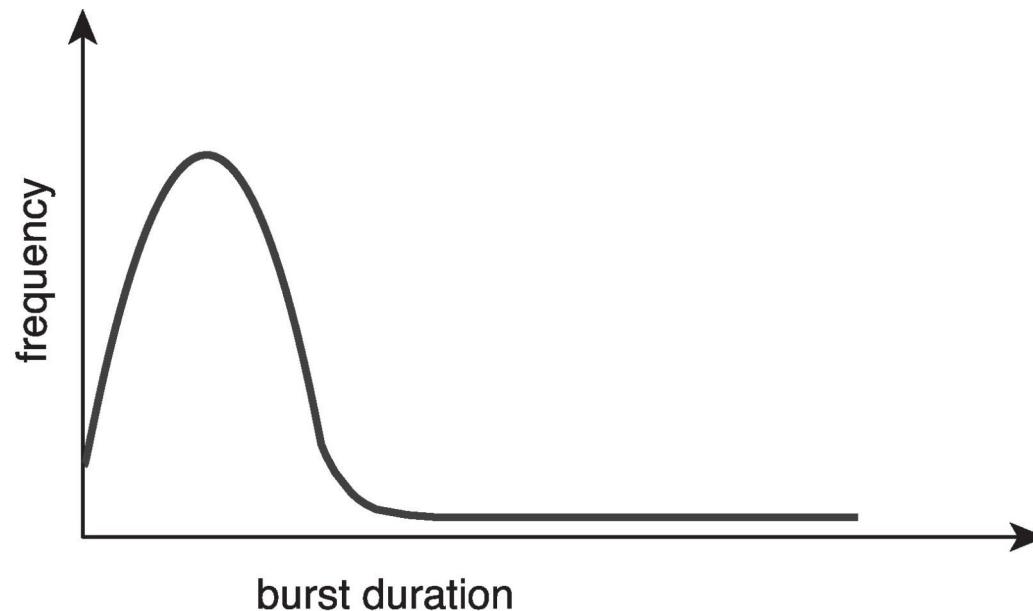
# Let's understand the problem

- Programs have alternate CPU and I/O bursts
  - Some are CPU intensive
  - Some are I/O intensive
  - Some are mix of both

- A C program example:

```
f(int i, int j, int k) {
 j = k * i; // CPU burst
 scanf("%d", &i); // I/O
 burst
 k = i * j;// CPU burst
 printf("%d\n", k);// I/O
 burst
 return k;
}
```

# CPU bursts: observation



Large number of short bursts

Small number of longer  
bursts

# Scheduler, what does it do?

- From a list of processes, ready to run
  - Selects a process for execution
  - Allocates a CPU to the process for execution
  - Does “context switch”
    - Context: Set of registers
    - Switch from context of one process to another process
    - May be done like this: P1 context -> scheduler context -> P2 context

# When is scheduler invoked?

## 1) Process Switches from running to waiting state

- Waiting for I/O, etc.

## 2) Switches from running to ready state

- E.g. on a timer interrupt

## 3) Switches from waiting to ready

- I/O wait completed, now ready to run

## 4) Terminates

▪ Scheduling under 1 and 4 is **nonpreemptive**

▪ All other scheduling is **preemptive**

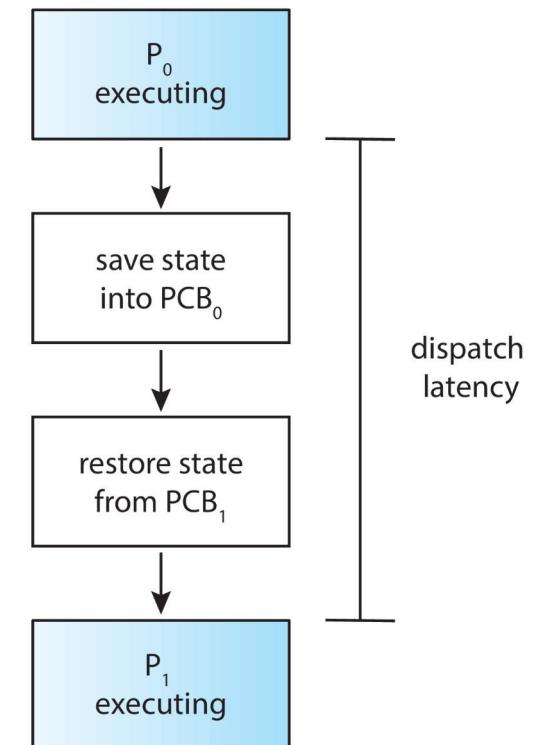
- Consider access to shared data

- Consider preemption while in kernel mode

- Consider interrupts occurring during crucial OS activities

# Dispatcher: A part of scheduler

- Gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency
  - time taken to stop one process and start another running
- Xv6: swtch(), some tail end parts of sched(), yield(), trap(), trapret()



# Dispatcher in action on Linux

- Run “`vmstat 1 3`”
  - Means run `vmstat` 3 times at 1 second delay
- In output, look at CPU:cs
  - **Context switches every second**
- Also for a process with pid 3323
  - Run
  - “`cat /proc/3323/status`”
  - See
    - `voluntary_ctxt_switches`
    - --> Process left CPU
    - `nonvoluntary_ctxt_switche`
    - --> Process was preempted
    -

# Scheduling criteria

- **CPU utilization: Maximise**

- keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.

- **Throughput : Maximise**

- # of processes that complete their execution per time unit

- **Turnaround time : Minimise**

- amount of time to execute a particular process

- **Waiting time : Minimise**

- amount of time a process has been waiting in the ready queue

- **Response time : Minimise**

- amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

# Calculations of different criteria

- If you want to evaluate an algorithm practically, you need a proper workload !
  - Processes with CPU and I/O bursts
  - Different durations of CPU bursts
  - Different durations of I/O bursts
    - How to do this programmatically?
    - How to ensure that after 2 seconds an I/O takes place?
  - Need periods when system will be “idle” – no process schedulable !

# Calculations of different criteria

## ▪ CPU Utilization

- % time spent in doing ‘useful’ work
- What is useful work?
  - On linux
    - there is an “idle” thread, scheduled when no other task is RUNNABLE
    - Not running idle thread is productive work
    - Includes process + scheduling time + interrupts
  - On other systems?

## ▪ Need to define

- On xv6
  - We can say that time spent in the loop selecting a process is idle work

# Calculations of different criteria

- **Throughput**

- # processes that complete execution per unit time
- Formula: total # processes completed / total time
- Simply divide by your total workload that completed by the time taken
- Depends on the workload as well. ‘long’ or ‘short’ processes.
- If too many short processes , then throughput may appear to be high, like 10s of processes per second
-

# Calculations of different criteria

## ▪ Turnaround time

- Amount of time required for one process to complete
- For every process, note down the starting and ending time, difference is TA-time
- For process P<sub>1</sub> -> Time when process ended – time when process started
  - = Sum of time spent in (ready queue + running + waiting for I/O)

## ▪ Do the average TA-time

# Calculations of different criteria

- **Waiting time**
- amount of time a process has been waiting in the *ready* queue.
- To be minimised.
- Part of turn around time
- CPU scheduling does not affect waiting time in I/O queues, it affects time in ready queue
- 
-

# Scheduling Criteria

## ▪ Response time

- amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).
- To be minimised.
- E.g. time between your press of a key , and that key being shown in screen

# **Challenges in implementing the scheduling algorithms**

- **Not possible to know number of CPU and I/O bursts and the duration of each before the process runs !**
  - **Although when we do numerical problems around each algorithm, we assume some values for CPU and I/O bursts !**
  -

# GANTT chart

- A timeline chart showing the sequence in which processes get scheduled
- Used for analysing a scheduling algorithm

# Scheduling Criteria: Differing requirements

- Different uses need different treatment of the scheduling criteria
- Knowing the workload is a challenge
- E.g. a desktop system
  - Response time is important
    - Minimize average response time Vs minimize variance in response time?
    - Sometimes fast, sometimes slow Vs Reasonable and predictable response time

# **Our discussion on scheduling algorithms**

- Analysis with only one CPU burst per process**
  - Ideally we should do for hundreds of CPU bursts**
- Only waiting time considered as criteria**

# Scheduling Algorithms

# First- Come, First-Served (FCFS) Scheduling

| Process | Burst Time |
|---------|------------|
| P1      | 24         |
| P2      | 3          |
| P3      | 3          |

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:

Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time:  $(0 + 24 + 27)/3 = 17$

Non Pre-emptive algorithm

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P2 , P3 , P1

The Gantt chart for the schedule is:

Waiting time for P1 = 6; P2 = 0; P3 = 3

Average waiting time:  $(6 + 0 + 3)/3 = 3$

Much better than previous case

# FCFS: Convoy effect

- Consider one CPU-bound and many I/O-bound processes
- CPU bound process is scheduled, I/O bound processes are waiting in I/O queues
- I/O bound processes finish I/O and move to ready queue, and wait for CPU bound process to finish
  - I/O devices Idle
- CPU bound process over, goes for I/O. I/O bound processes run quickly, move to I/O queues again
  - CPU idle
- CPU bound process will run when it's ready to run
- Same process will repeat
- --> Lower CPU utilisation
  - Better if I/O bound processes run first

# FCFS: further evaluation

- Troublesome for interactive processes
  - CPU bound process may hog CPU
  - Interactive process may not get a chance to run early and response time may be quite bad

# **Shortest-Job-First (SJF) Scheduling**

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time. Better name – Shortest Next CPU Burst Scheduler
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user – bad idea, unlikely to know!

# Example of SJF

| Process | Burst Time |
|---------|------------|
| P1      | 6          |
| P2      | 8          |
| P3      | 7          |
| P4      | 3          |

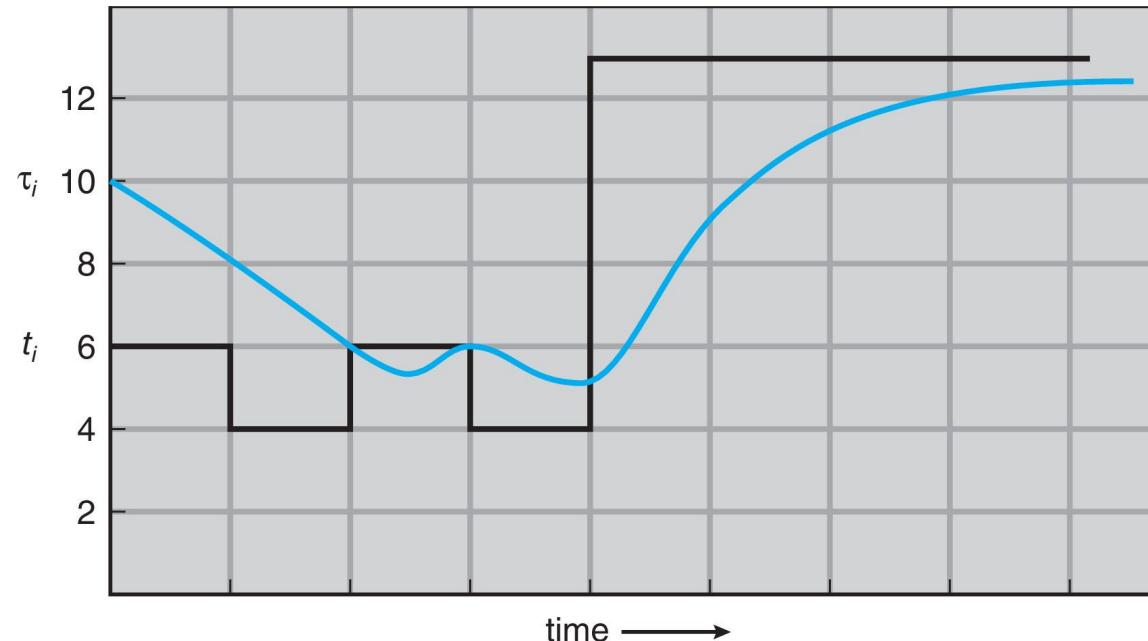
SJF scheduling chart

$$\text{Average waiting time} = (3 + 16 + 9 + 0) / 4 = 7$$



# Prediction of the Length of the Next CPU Burst

- $A = 1/2$ ,  $\tau_0 = 10$



# Examples of Exponential Averaging

- $\alpha = 0$
- $\Delta_{n+1} = \Delta_n$
- Recent history does not count
- $\alpha = 1$
- $\Delta_{n+1} = \alpha t_n$
- Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $\Delta_{n+1} = \alpha t_n + (1 - \alpha) \Delta_{n-1} + \dots$
  - $+ (1 - \alpha)^j \Delta_{n-j} + \dots$
  - $+ (1 - \alpha)^{n+1} \Delta_0$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Example of Shortest-remaining-time-first

Preemptive SJF = SRTF. Now we add the concepts of varying arrival times and preemption to the analysis

| Process | Arrival Time | Burst Time |  |
|---------|--------------|------------|--|
| P1      | 0            | 8          |  |
| P2      | 1            | 4          |  |
| P3      | 2            | 9          |  |
| P4      | 3            | 5          |  |

Preemptive SJF Gantt Chart

$$\text{Average waiting time} = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 \text{ msec}$$

# Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (time quantum  $q$ ), usually 10-100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once.
  - No process waits more than  $(n-1)q$  time units.

# Round Robin (RR) Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
- $q$  large ↴ FIFO
  - $q$  small ↴  $q$  must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

| <u>Process</u> | <u>Burst Time</u> |    |
|----------------|-------------------|----|
| $P_1$          |                   | 24 |
| $P_2$          | 3                 |    |
| $P_3$          | 3                 |    |

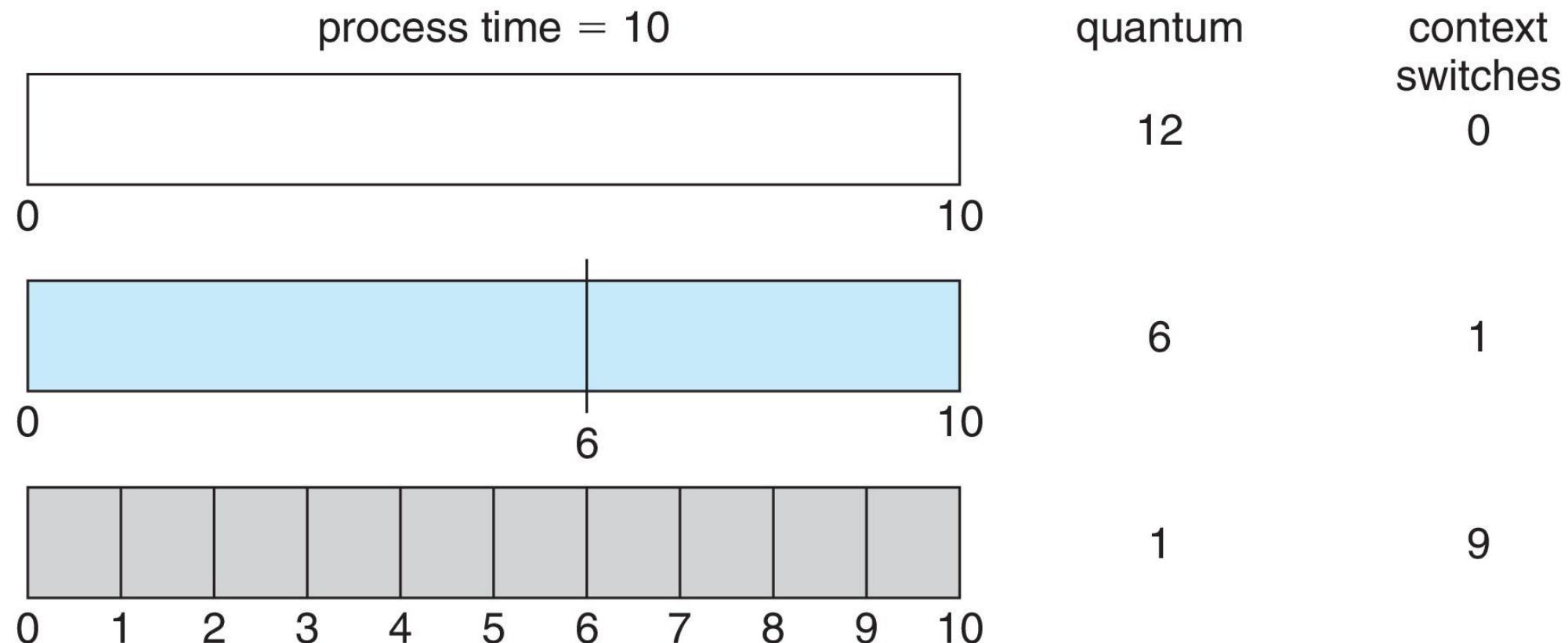
The Gantt chart is:

Typically, higher average turnaround than SJF, but better *response*

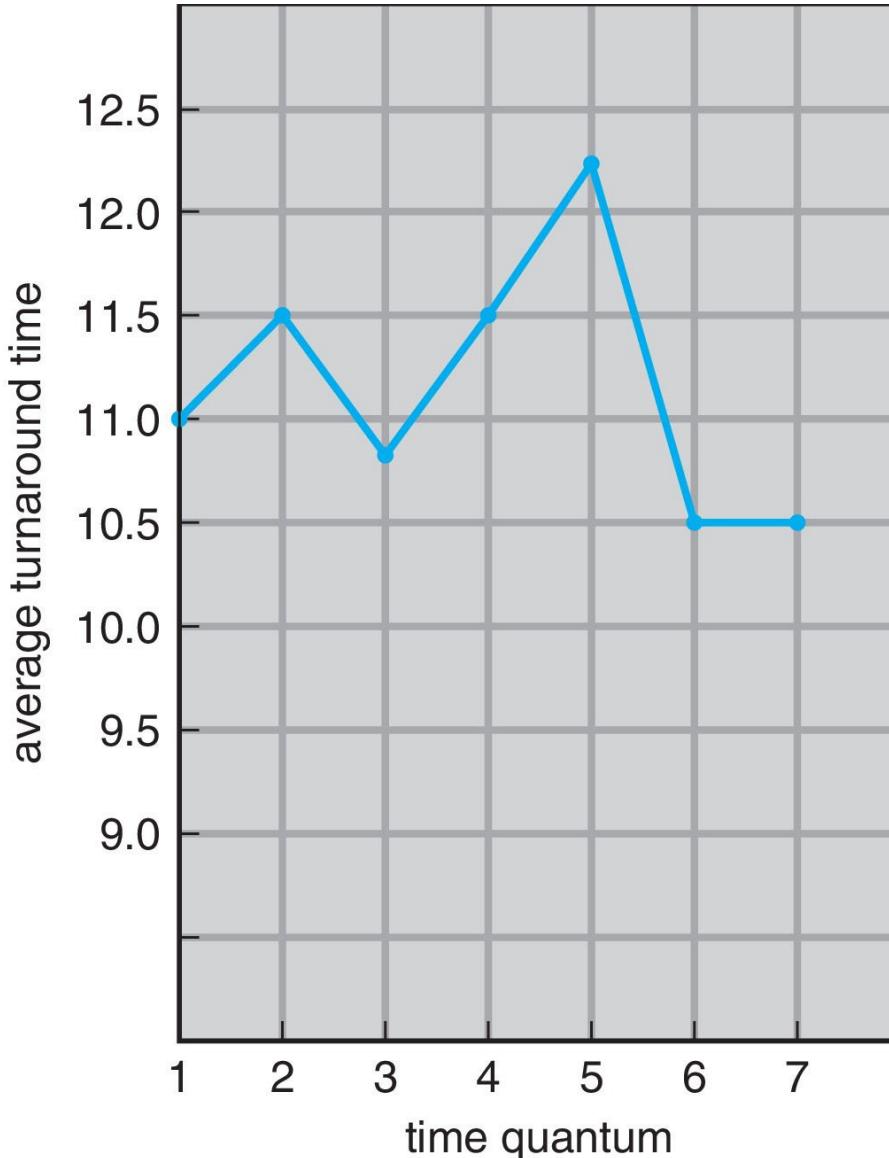
q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



| process | time |
|---------|------|
| $P_1$   | 6    |
| $P_2$   | 3    |
| $P_3$   | 1    |
| $P_4$   | 7    |

80% of CPU bursts should be shorter than quantum

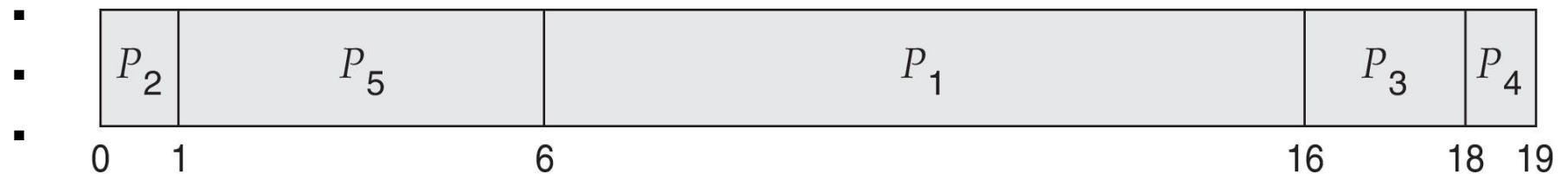
# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  highest priority)
- Preemptive (timer interrupt, more time for more priority)
- Nonpreemptive (no timer interrupt, just schedule process with highest priority)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  Starvation – low priority processes may never execute
- Solution  Aging – as time progresses increase the priority of the process
-

# Example of Priority Scheduling

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| $P_1$          | 3                 | 10              |
| $P_2$          | 1                 | 1               |
| $P_3$          | 2                 | 4               |
| $P_4$          | 1                 | 5               |
| $P_5$          | 5                 | 2               |

- Priority scheduling Gantt Chart



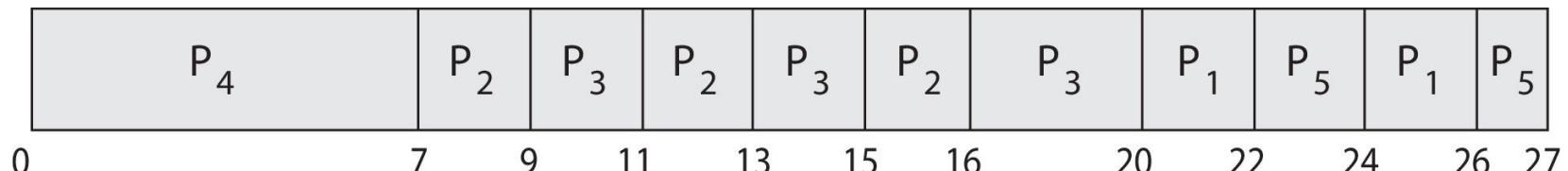
- Average waiting time = 8.2 msec

# Priority Scheduling with Round-Robin

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| $P_1$          | 4                 | 3               |
| $P_2$          | 5                 | 2               |
| $P_3$          | 8                 | 2               |
| $P_4$          | 7                 | 1               |
| $P_5$          | 3                 | 3               |

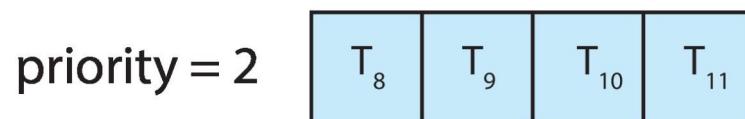
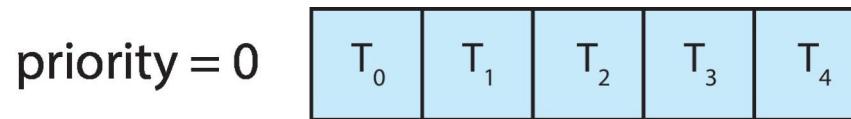
Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum



# Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



•

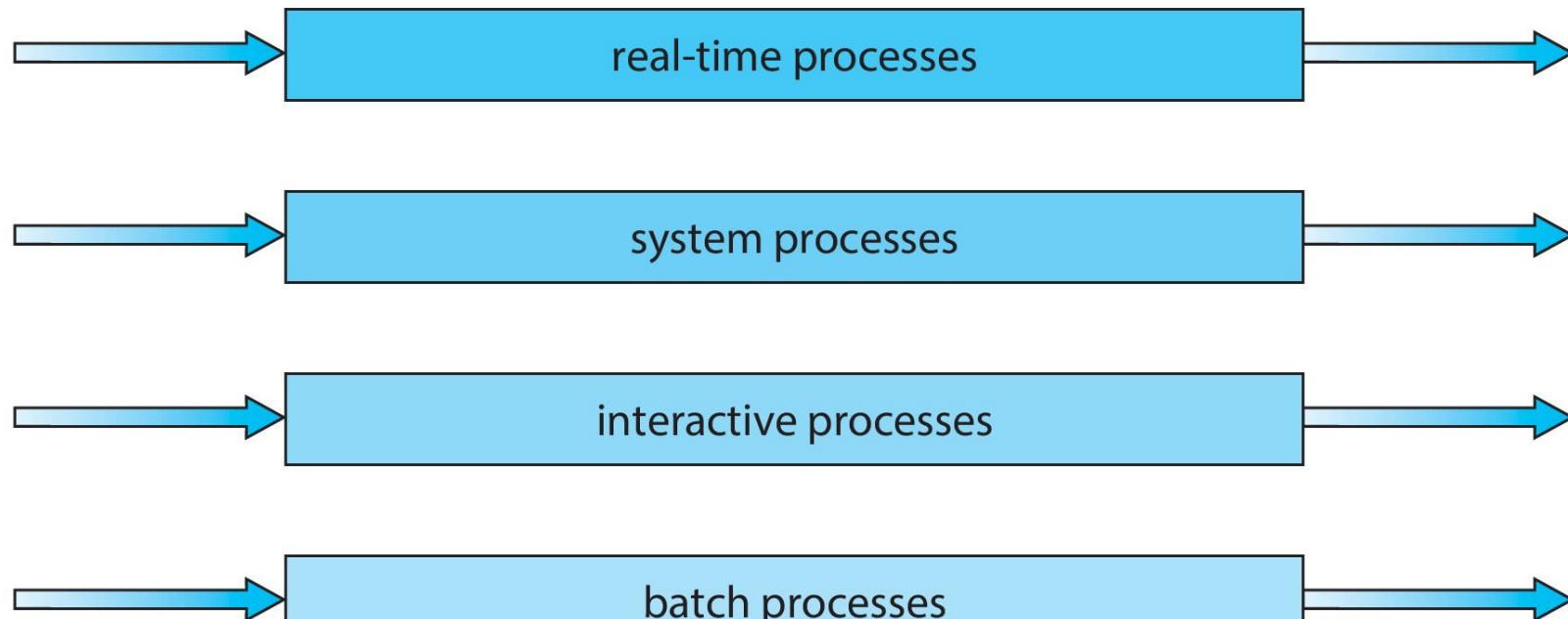
•

•



# Multilevel Queue

highest priority



lowest priority

# Implementing multilevel queue

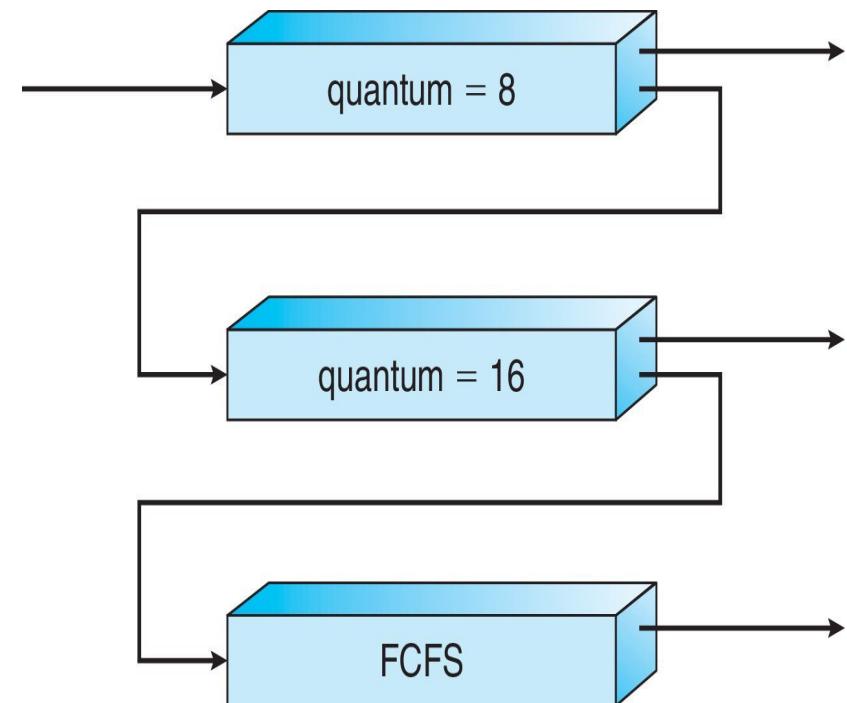
- Processes need to have a priority
  - Either modify fork()/exec() to have a priority
  - Or add a nice() system call to set priority
- How to know the priority?
  - The end user of the computer system needs to know this from needs of real life
  - E.g. on a database system, the database process will have a higher priority than other processes

# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
    - method used to determine which queue a process will enter when that process needs service

# Example of Multilevel Feedback Queue

- Three queues:
- $Q_0$  – RR with time quantum 8 milliseconds
- $Q_1$  – RR time quantum 16 milliseconds
- $Q_2$  – FCFS
- Scheduling rules
- Serve all processes in  $Q_0$  first
- Only when  $Q_0$  is empty, serve processes in  $Q_1$
- Only when  $Q_0$  and  $Q_1$  are empty, serve processes in  $Q_2$



# Example of Multilevel Feedback Queue

- **Scheduling**

- **A new job enters queue  $Q_0$**

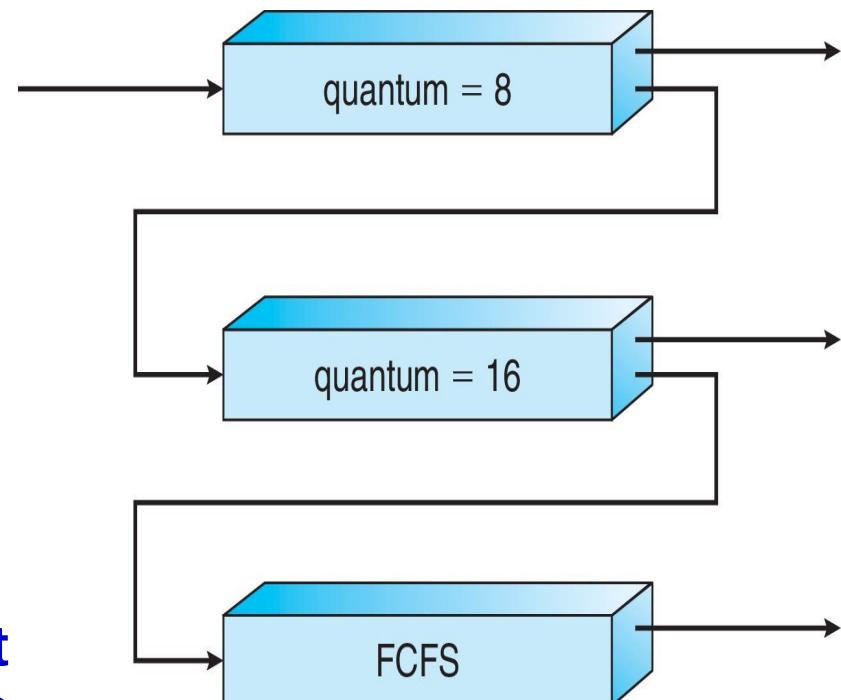
- **When it gains CPU, job receives 8 milliseconds**

- **If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$**

- **At  $Q_1$  job receives 16 additional milliseconds**

- **If it still does not complete, it is preempted and moved to queue  $Q_2$**

- **To prevent starvation, move a process from lower-priority queue to higher priority queue after it has waited for too long**



# Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as process-contention scope (PCS) since scheduling competition is within the process
- Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

# Pthread Scheduling

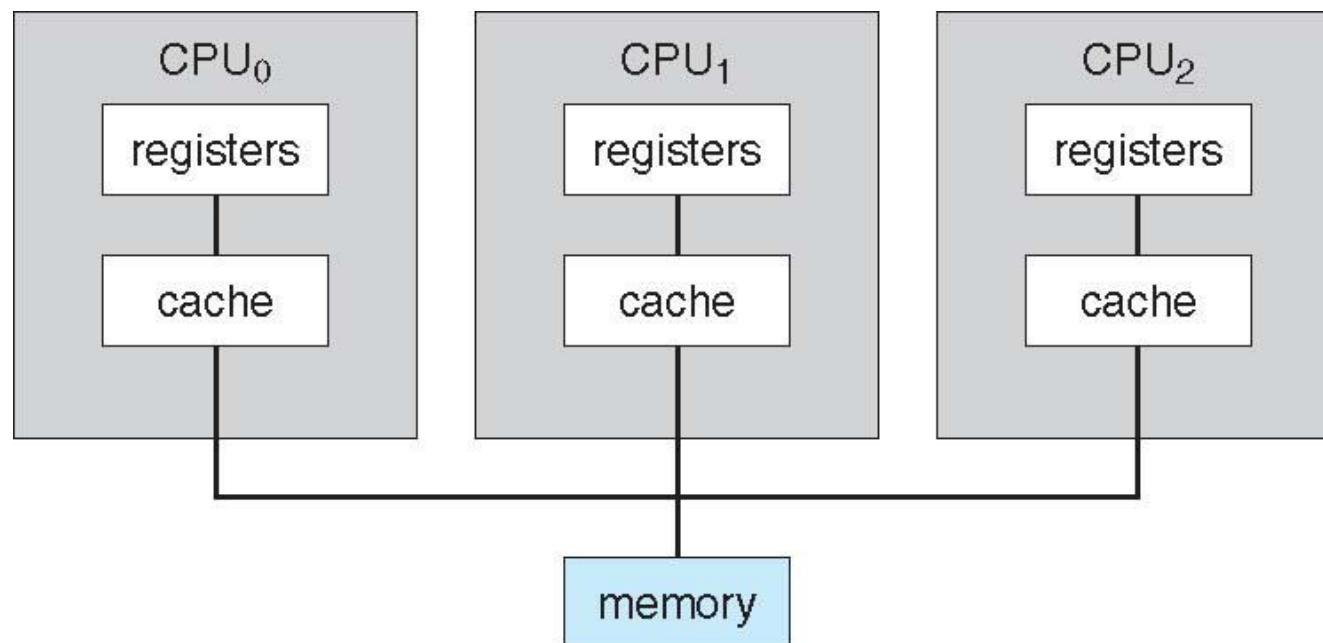
- **PTHREAD\_SCOPE\_PROCESS** schedules threads using PCS scheduling
- **PTHREAD\_SCOPE\_SYSTEM** schedules threads using SCS scheduling
- Linux and macOS only allow **PTHREAD\_SCOPE\_SYSTEM**
- Let's see a Demo using a program

# **Multi Processor Scheduling**

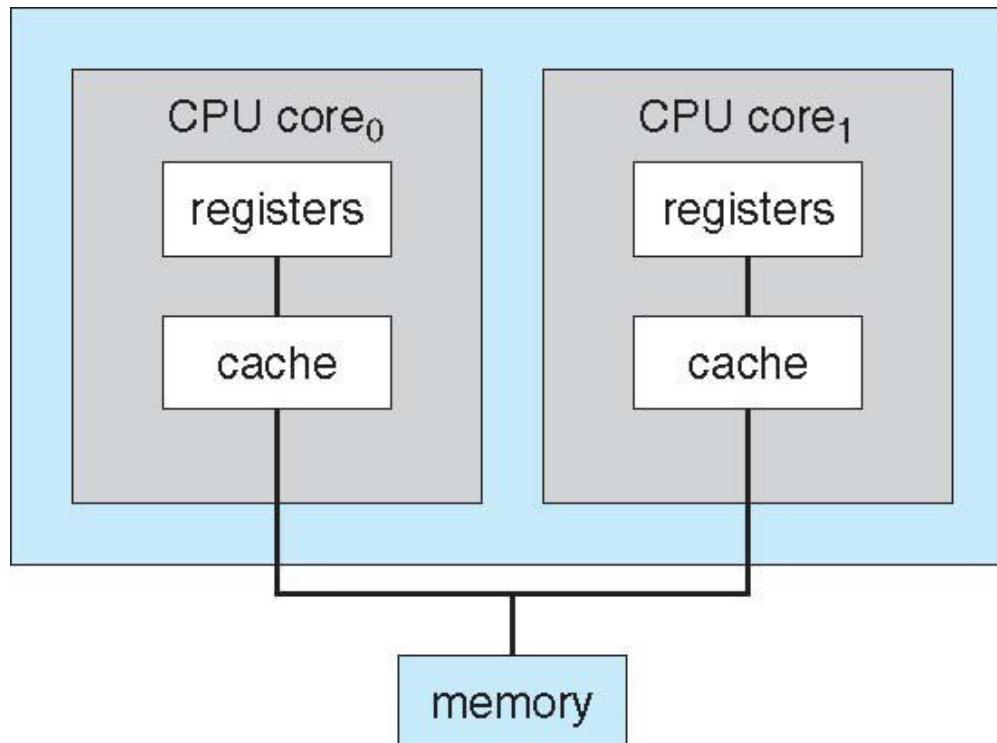
# Multiprocessor systems

- **Each processor has separate set of registers**
  - All: eip, esp, cr3, eax, ebx, etc.
- **Each processor runs independently of others**
- **Main difference is in how do they access memory**

# Symmetric multiprocessing (SMP)



# Multicore systems (also SMP)



No difference from the perspective of OS. The hardware ensures that OS sees multiple processors and not multiple cores.

# Booting multi-processor systems

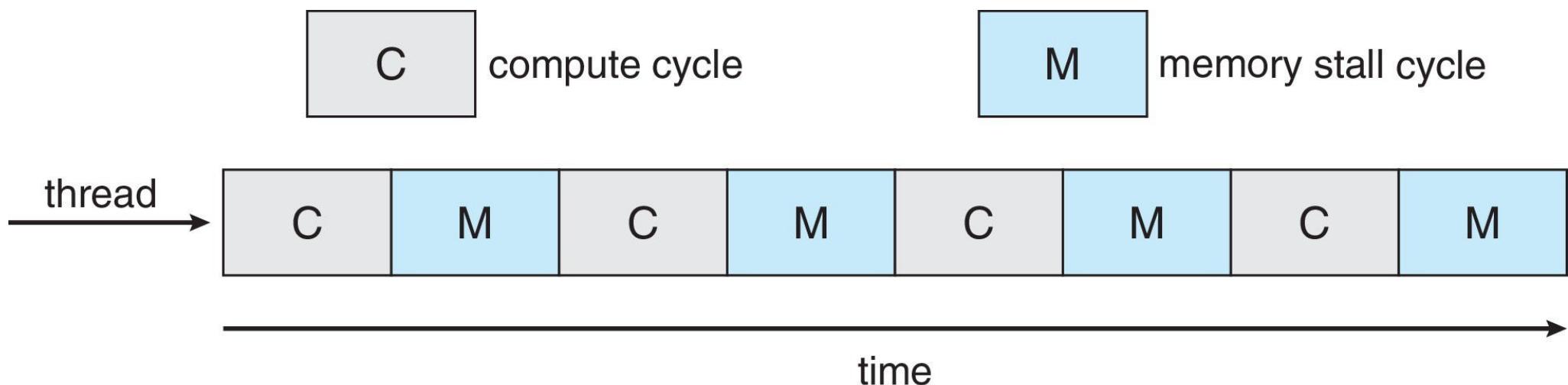
- X86 + xv6

- One processor starts, runs BIOS, loads kernel, initializes kernel data structures
- **Mpinit()** : first processor scans certain memory addresses for information about other processors and obtains configuration and configures them
- **Startothers()** : First processor initializes kernel data structures for other processors , and ensures that other processors start in scheduler() and starts them
- First processor “starts” other processors

-

# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens



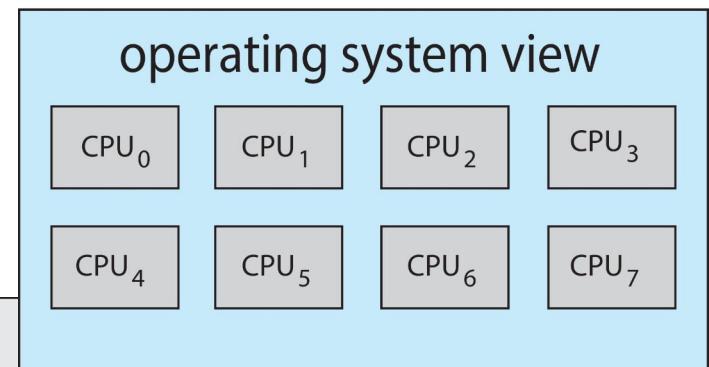
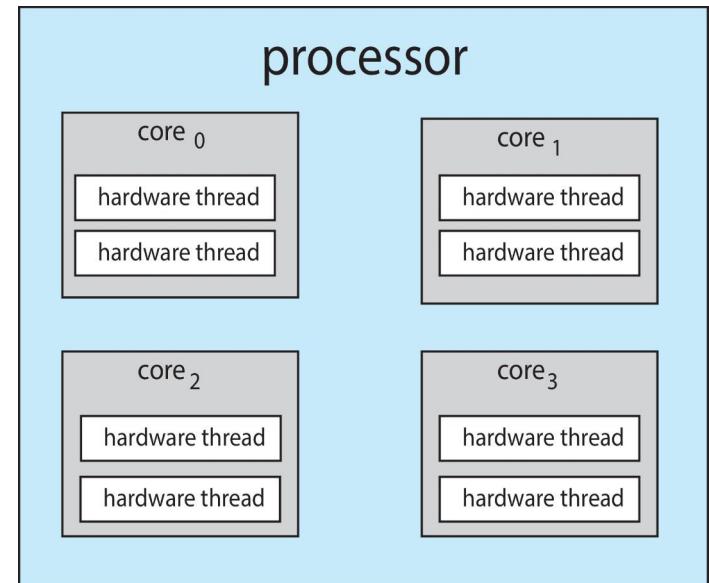
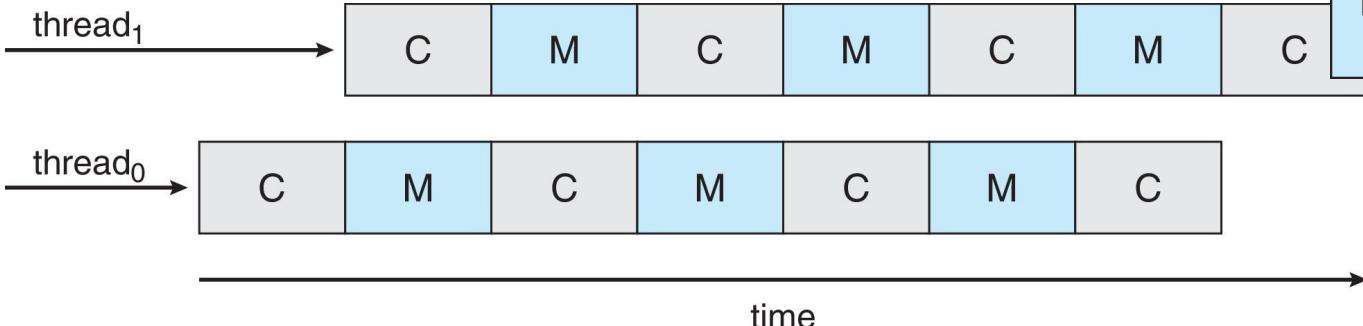
# Multithreaded Multicore System

Each core has > 1 hardware threads.

**Chip-multithreading (CMT)** assigns each core multiple hardware threads.  
(Intel refers to this as **hyperthreading**.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

If one thread has a memory stall, switch to another thread!



# Non-Uniform Memory Architecture (NUMA)

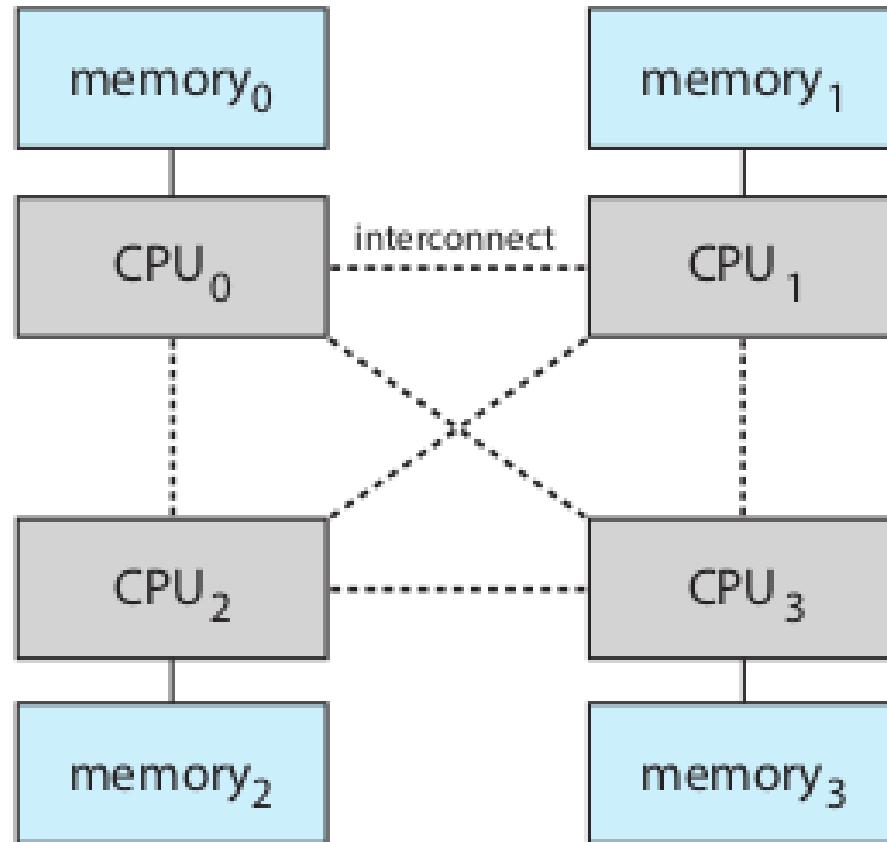


Figure 1.10 NUMA multiprocessing architecture.

# More on SMP systems

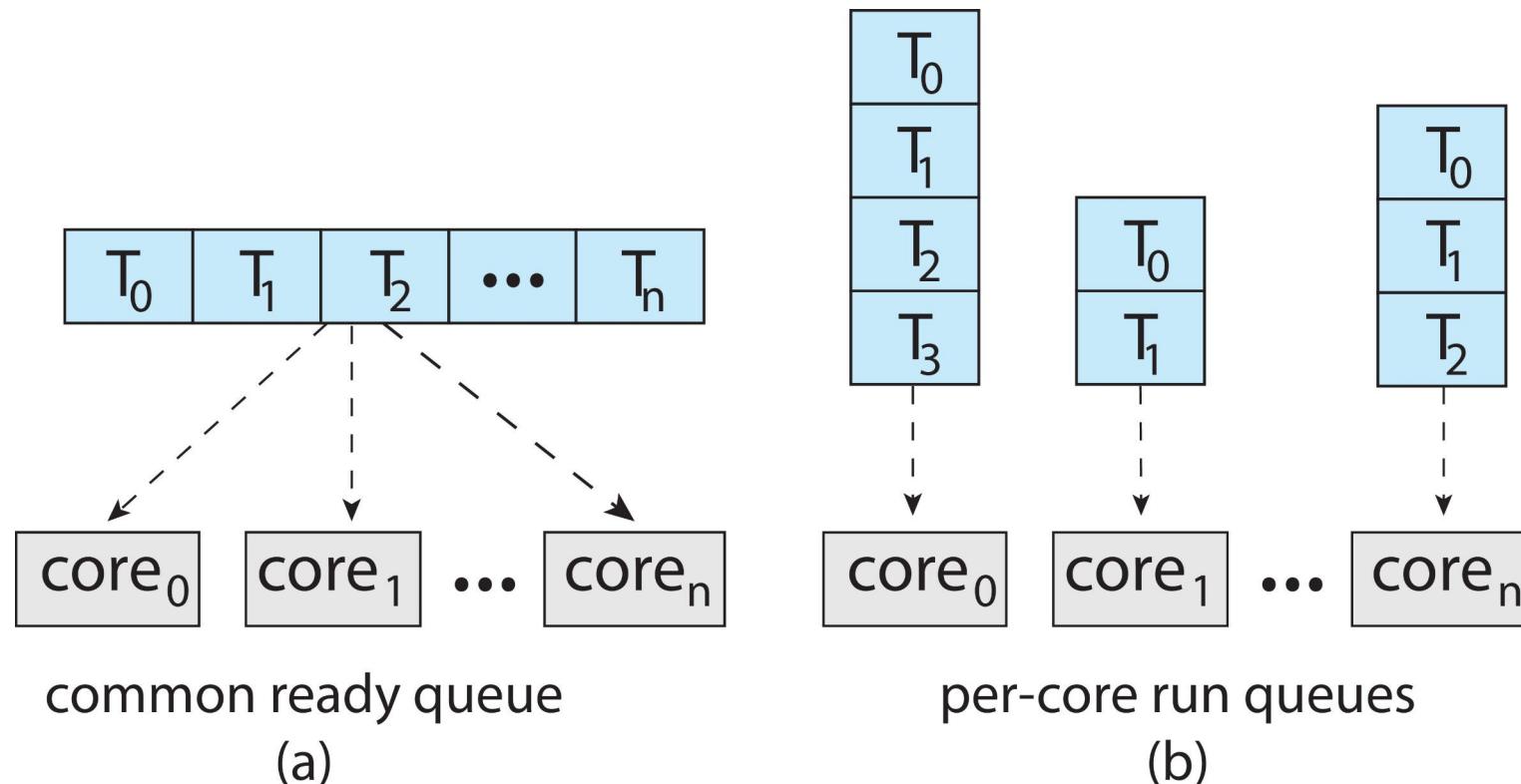
- During booting – each CPU needs to be turned on
  - Special I/O instructions writing to particular ports
  - See `lalicstartap()` in xv6
  - Need to setup CR3 on each processor
  - Segmentation, Page tables are shared (same memory for all CPUs)
- All processors will keep running independently of each other
- Different interrupts on each processor – each needs IDT setup
- Each processor will be running processes, interrupt handlers, syscalls
- Synchronization problems !
- How to do scheduling ?

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
    - Heterogeneous multiprocessing

# Multiple-Processor Scheduling

- **Symmetric multiprocessing (SMP) is where each processor is self scheduling.**
- **All threads may be in a common ready queue (a)**
- **Each processor may have its own private queue of threads (b)**



# Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration – idle processors pulls waiting task from busy processor

# Multiple-Processor Scheduling – Load Balancing

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
-

# Multiple-Processor Scheduling – Load Balancing

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

# SMP in xv6

- Only one process queue
- No load balancing, no affinity
- A process may run any CPU burst /allotted-time-quantum on any processor randomly
- See the code of:
  - `Startothers()`, `mpenter()`, `mpmain()`
- Different scheduler's kernel stack on each processor
  - Done in `startothers()`
- Each processor calls `scheduler()` from `mpmain()`

**End**