# slides 3

## IPC

### Revision of process related concepts

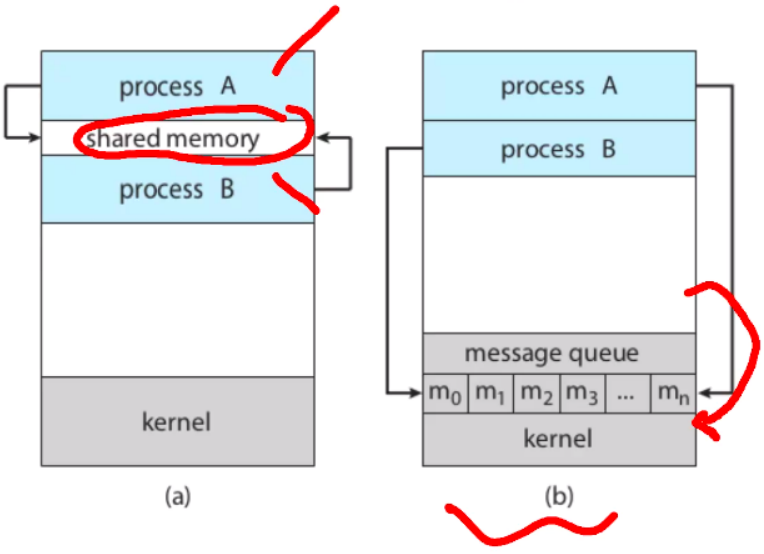
- PCB, struct proc
- Process lifecycle – different states
- Memory layout
- Memory management
- Interrupts handling, system call handling, code from xv6
- Scheduler, code of scheduler in xv6

# IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, e.g. copy paste
  - Computation speedup, e.g. matrix multiplication
  - Modularity, e.g. chrome – separate process for display, separate for fetching data
  - Convenience,
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

---

# Shared Memory Vs Message Passing




**Each requires OS to provide system calls for**

- Creating the IPC mechanism
- To read/write using the IPC mechanism
- Delete the IPC mechanism

**Note: processes communicating with each other with the help of OS!**

**Figure 3.11** Communications models. (a) Shared memory. (b) Message passing.

## Example of co-operating processes: Producer Consumer Problem

- **Shared data**

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **Can only use BUFFER_SIZE-1 elements**

## Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {
      /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count)  == out)
      ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

## Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

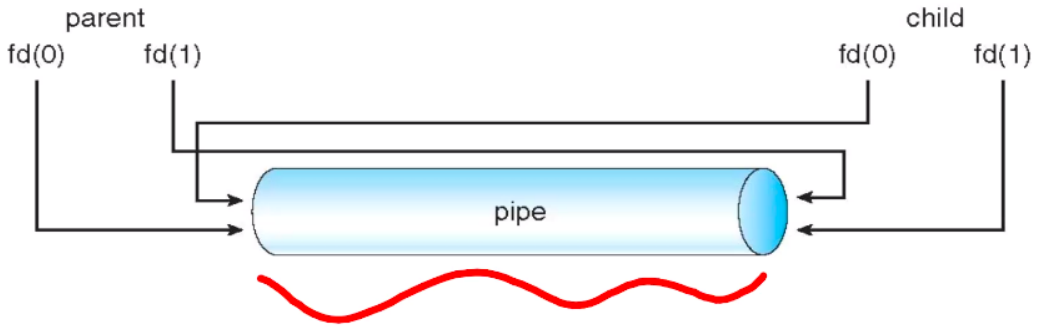
```
while (true) {
        while (in == out)
                ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

# Pipes for IPC

- **Two types**
  - **Unnamed Pipes or ordinary pipes**
  - **Named Pipe**

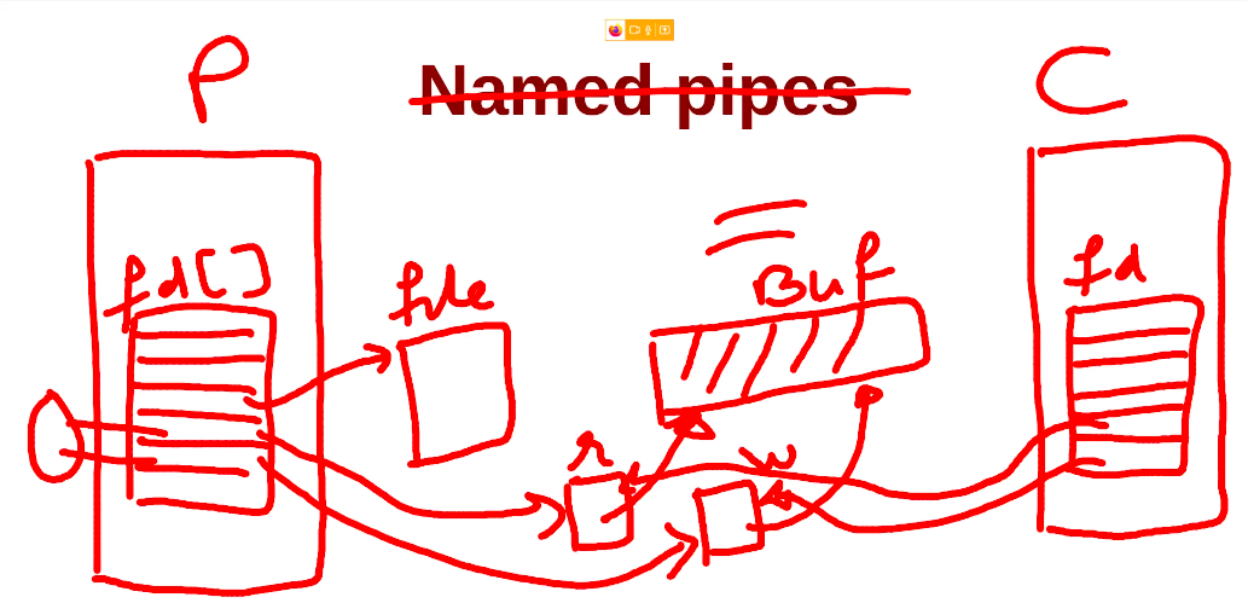
# Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a  parent-child (or sibling, etc) kind of relationship between communicating processes

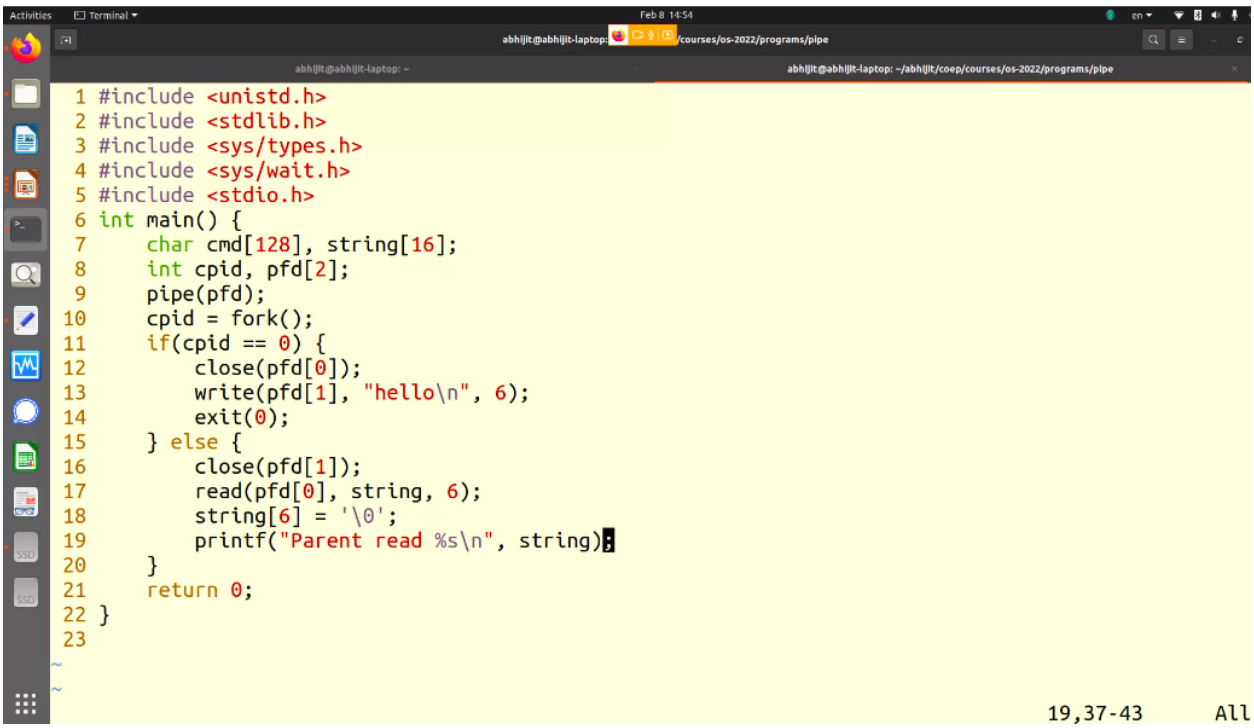


```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd, fd2;
    char str[1600];
    int pfd[2];

    pipe(pfd);
    /*pfd[0] - read file descriptor
    pfd[1] - write file descriptor   */
    write(pfd[1], "hello\n", 6);
    read(pfd[0], str, 6);
    str[6] = '\0';
    printf("%s", str);
}
```

"pipe.c" 20L, 378C 1,1 Al

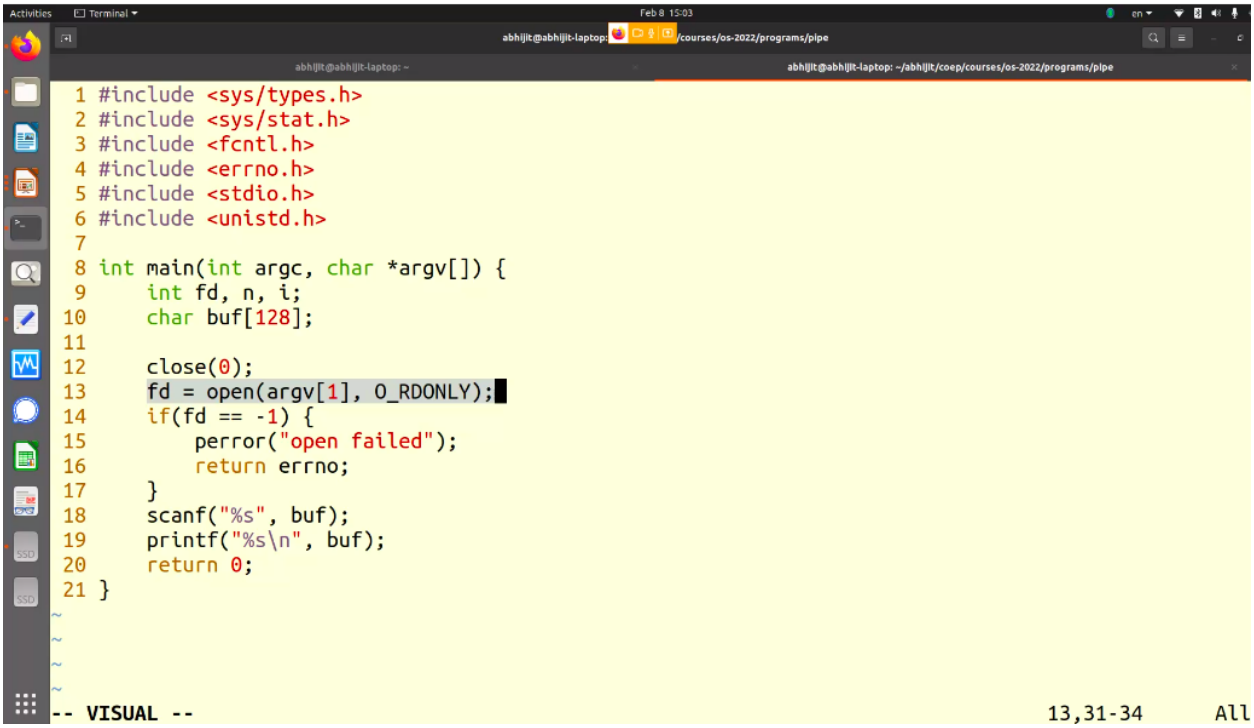



pipe is unidirectional, so all unused ends should be closed



```c
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
int main() {
    char cmd[128], string[16];
    int cpid, pfd[2];
    pipe(pfd);
    cpid = fork();
    if(cpid == 0) {
        close(pfd[0]);
        write(pfd[1], "hello\n", 6);
        exit(0);
    } else {
        close(pfd[1]);
        read(pfd[0], string, 6);
        string[6] = '\0';
        printf("Parent read %s\n", string);
    }
    return 0;
}
```

all the writes are delayed by kernel, and it will do it on its own convinience.

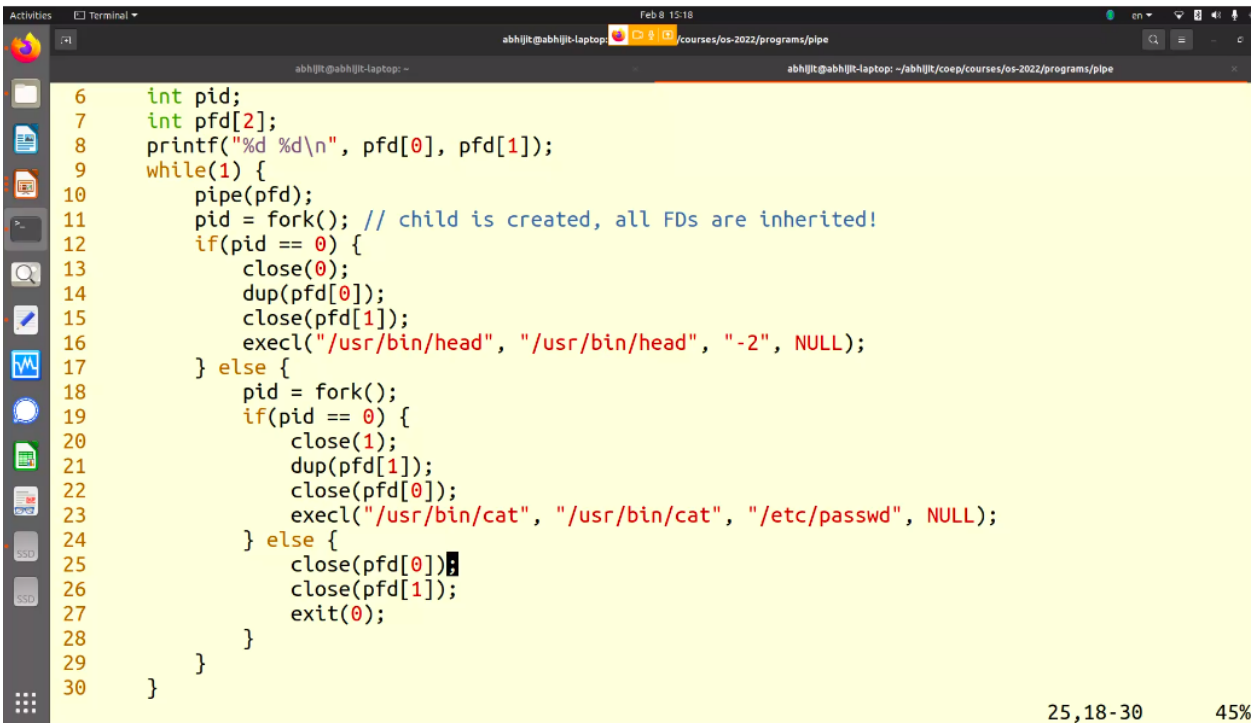


```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(0);
    fd = open(argv[1], O_RDONLY);
    if(fd == -1) {
        perror("open failed");
        return errno;
    }
    scanf("%s", buf);
    printf("%s\n", buf);
    return 0;
}
```

for this program, open will return file descriptor 0, and hence will read from file

see strace

execl, copies the file descriptors



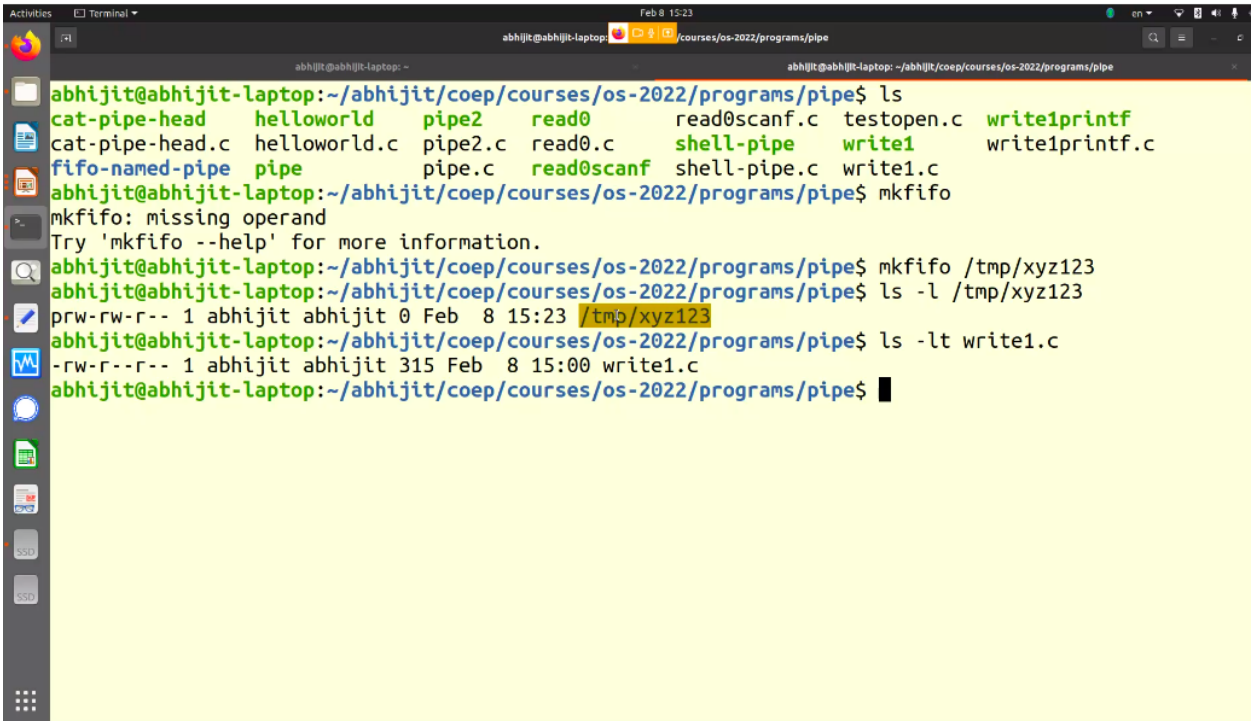

when we use process1 | process2

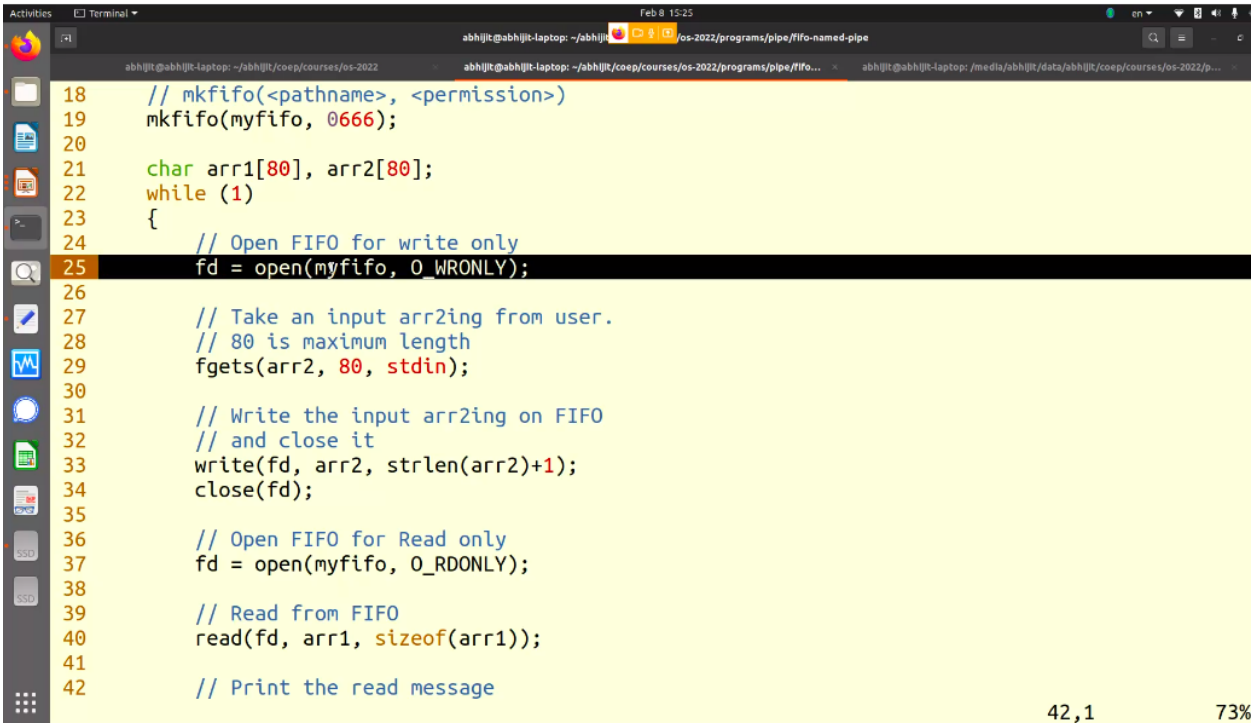
process 1 is continuously writing and process2 is continuously reading from pipe

# Named pipes

- **Also called FIFO**
- **Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO**
- **Named Pipes are more powerful than ordinary pipes**
- **Communication is bidirectional**
- **No parent-child relationship is necessary between the communicating processes**
- **Several processes can use the named pipe for communication**
- **Provided on both UNIX and Windows systems**
- **Is not deleted automatically by OS**

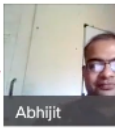
named pipe files are prefixed by p in file permissions




# Shared memory

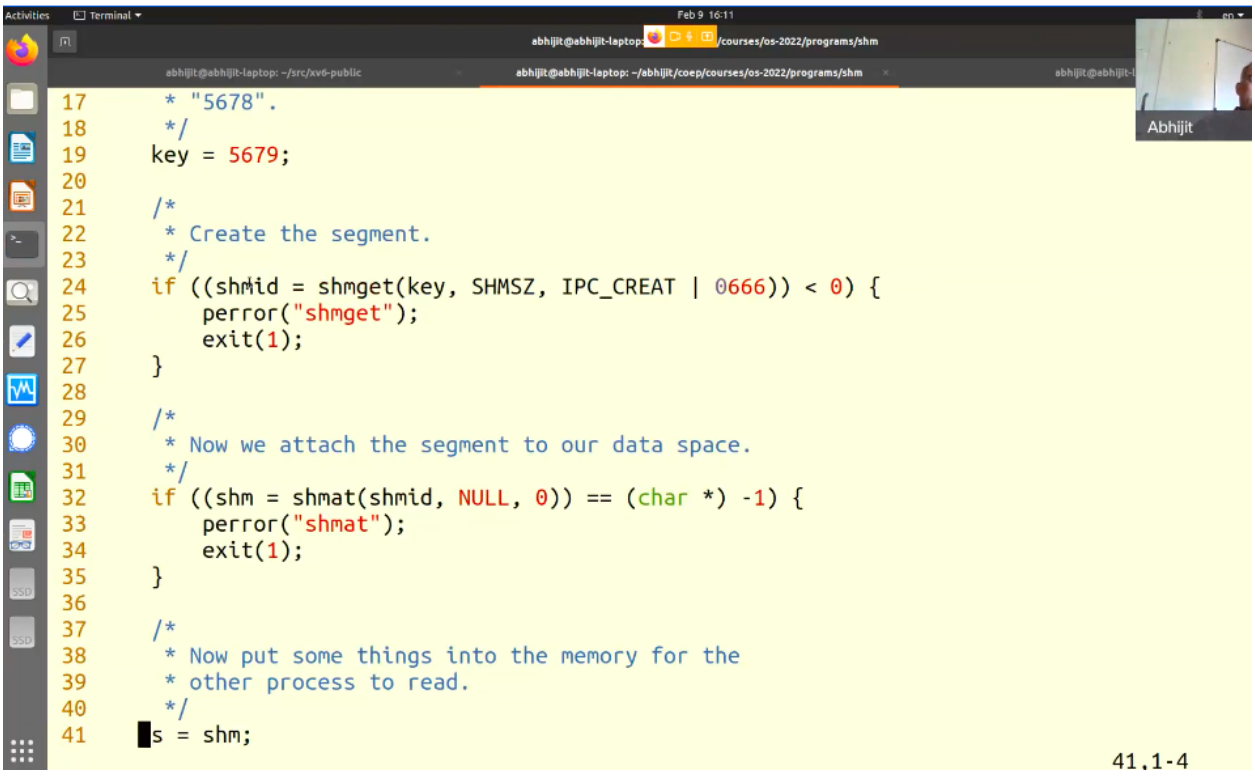
# System V shared memory

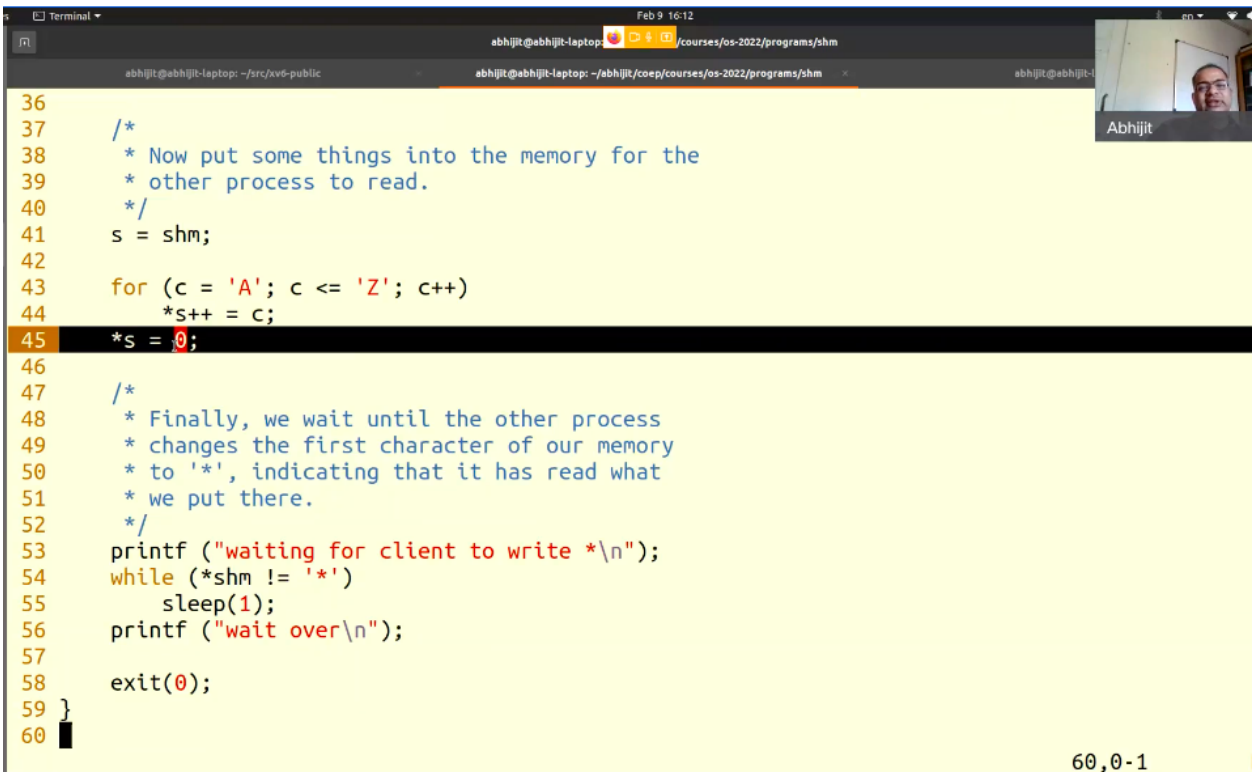



- **Process first creates shared memory segment**
  **segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);**
- **Process wanting access to that shared memory must attach to it**
  **shared_memory = (char *) shmat(id, NULL, 0);**
- **Now the process could write to the shared memory**
  **sprintf(shared_memory, "Writing to shared memory");**
- **When done, a process can detach the shared memory from its address space**

shmdt() detaches

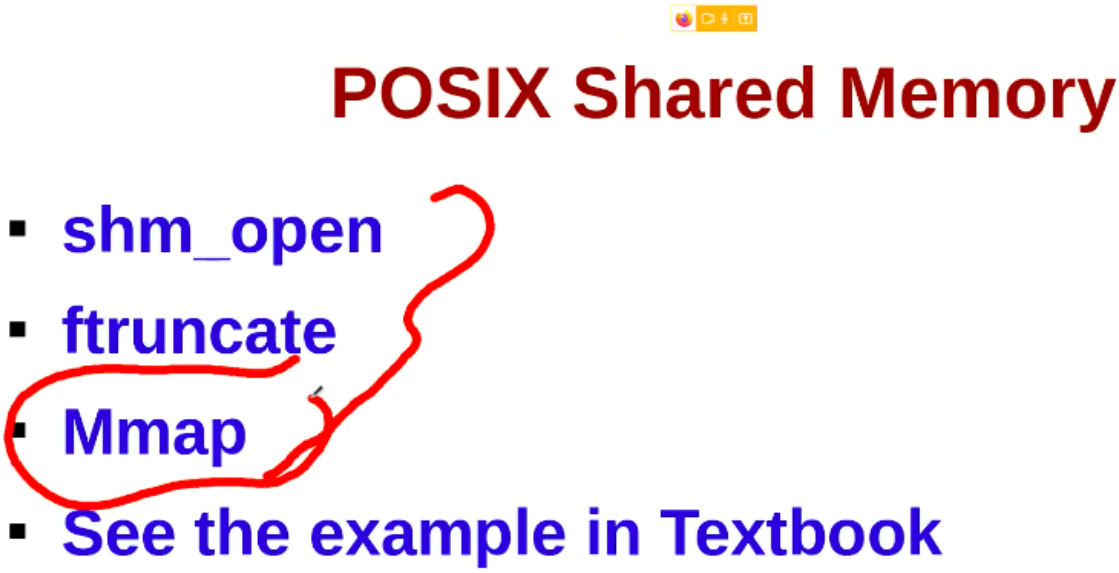
shm is a char*,key should be common with all process sharing the shared memory






the above shared memory systemcall were system 5 compliant,

linux gives POSIX(portable operating system interfaces) systemcalls too




ordinary pipes are unidirectional,whereas named pipes are bidirectional

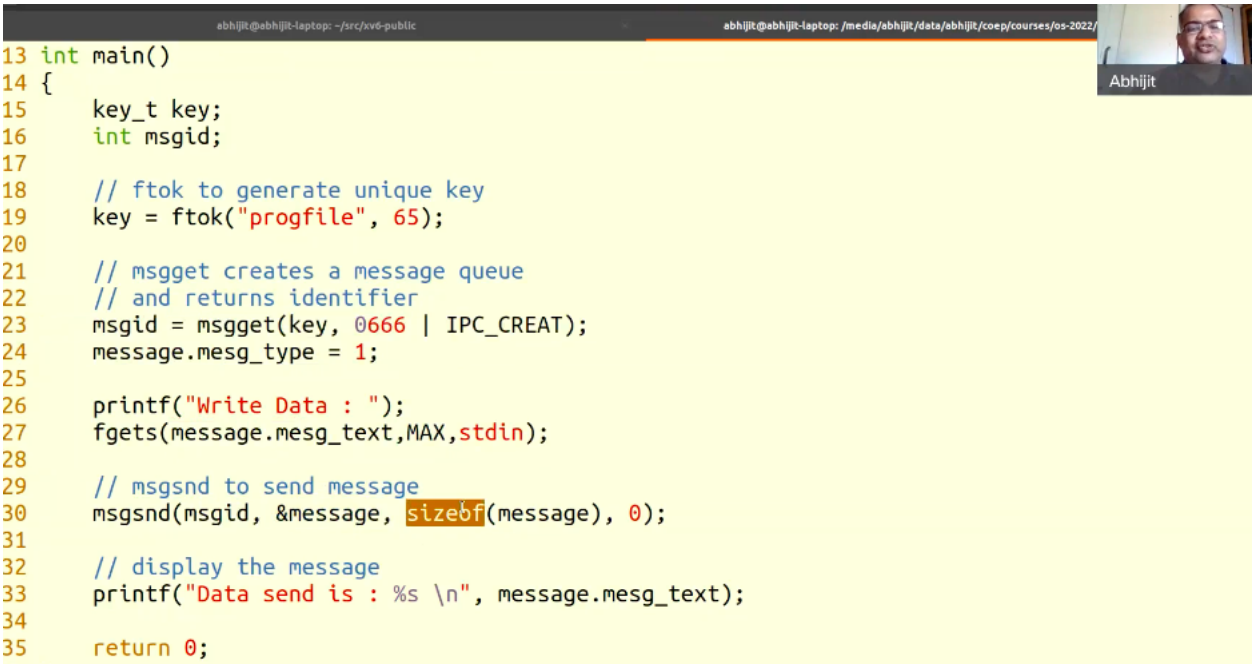
## Message Passing

indirect message passing example

receiver


```c

int main()
{
    key_t key;
    int msgid;

    //ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

```

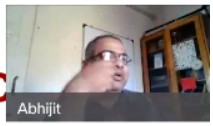
sender




```c
int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    fgets(message.mesg_text,MAX,stdin);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
```

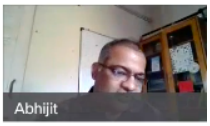
message passing using naming

## Message passing using direct communic




- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

## Message passing using IN-direct communication




- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

here A is the mailbox, not a sender or receiver

# Message passing using IN-direct communication

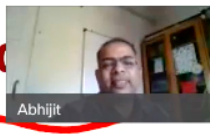

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1 sends; P2 and P3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

---

# Message Passing implementation: Synchronization issues


- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

## Producer consumer using blocking send receive




**Producer**

```
message next_produced;
while (true) {
/* produce an item in
next_produced */
send(next_produced);
}
```

**Consumer**

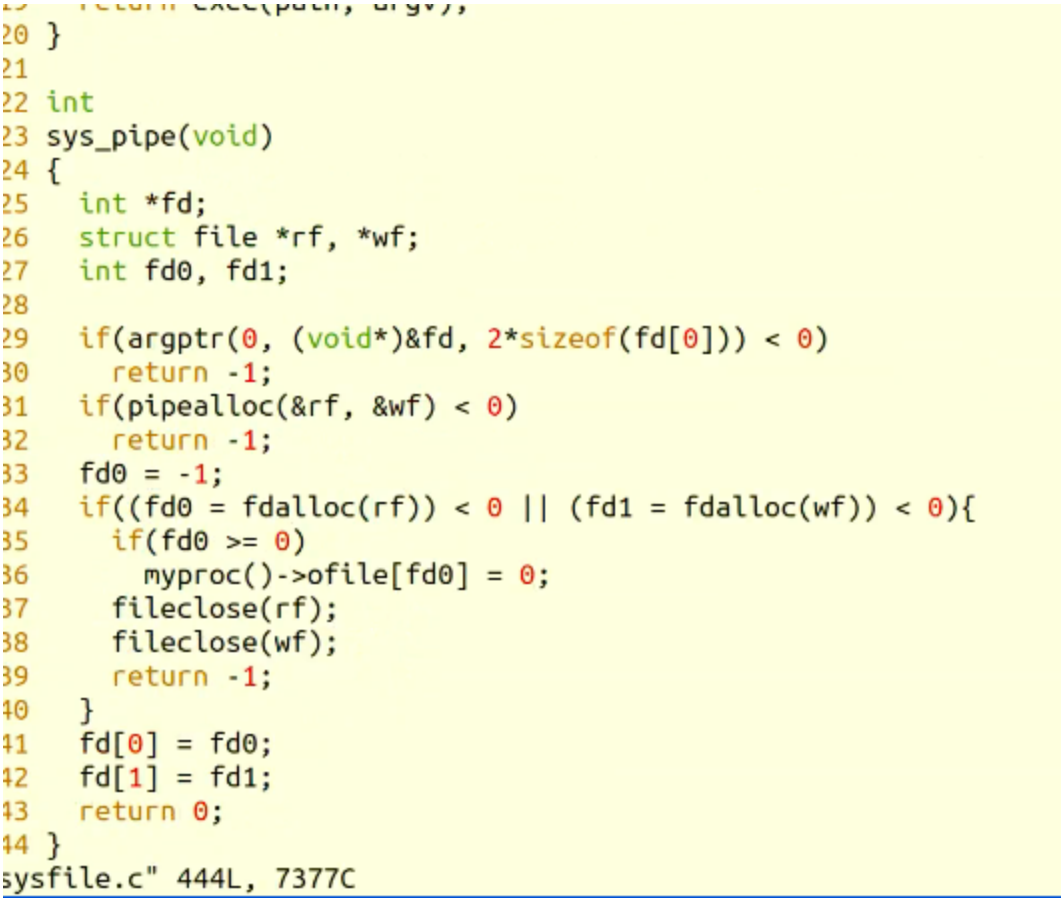
```
message
next_consumed;
while (true) {
receive(next_consumed);
}
```

writing blocking code is much easier.

## Message Passing implementation: choi Buffering

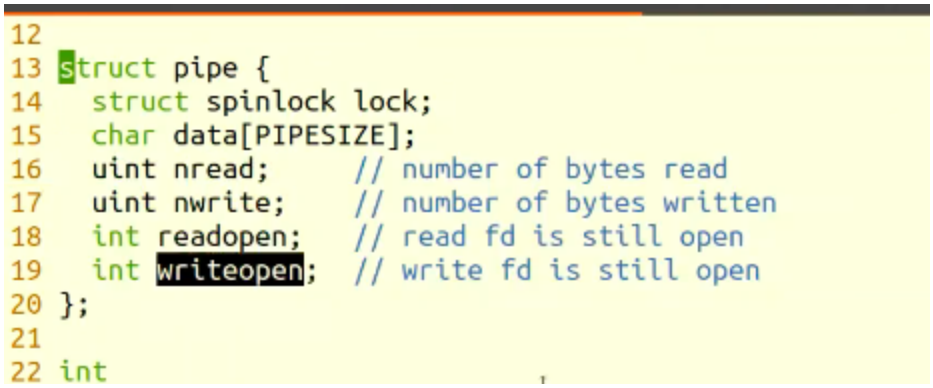

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     - Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of n messages
     - Sender must wait if link full
  3. Unbounded capacity – infinite length
     - Sender never waits

34:07



```
20 }
21
22 int
23 sys_pipe(void)
24 {
25   int *fd;
26   struct file *rf, *wf;
27   int fd0, fd1;
28
29   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
30     return -1;
31   if(pipealloc(&rf, &wf) < 0)
32     return -1;
33   fd0 = -1;
34   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
35     if(fd0 >= 0)
36       myproc()->ofile[fd0] = 0;
37     fileclose(rf);
38     fileclose(wf);
39     return -1;
40   }
41   fd[0] = fd0;
42   fd[1] = fd1;
43   return 0;
44 }
sysfile.c" 444L, 7377C
```

sys_pipe is called for pipe systemcall,which creates 2 file pointers and then calls pipealloc, does error checking and then allocate fd array and returns;

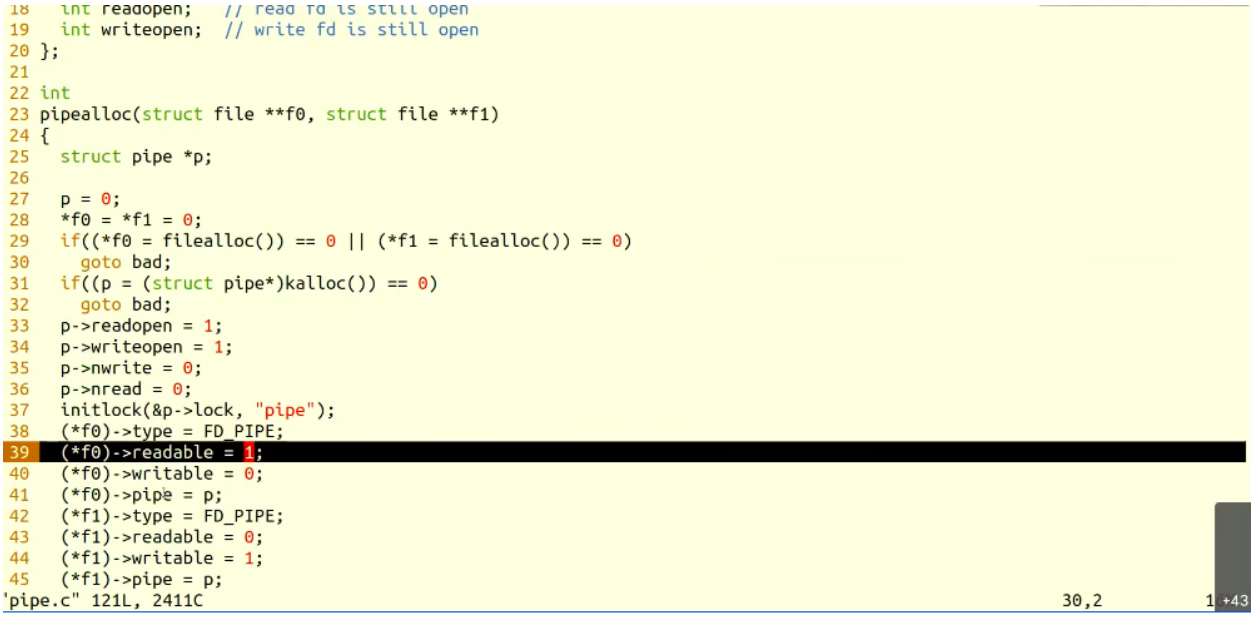


```
12
13 struct pipe {
14   struct spinlock lock;
15   char data[PIPESIZE];
16   uint nread;     // number of bytes read
17   uint nwrite;    // number of bytes written
18   int readopen;   // read fd is still open
19   int writeopen;  // write fd is still open
20 };
21
22 int
```

PIPESIZE is 512, data is the buffer used to store data,

inside pipealloc, it calls filealloc, which gives a struct file, then there is a call to kalloc() which will give you a 4kb page, and it is typecasted in pipe pointer.

then variables are initialized, then file structure, f0 and f1 are being initialized, one for writing and another for reading and then just return



```
18   int readopen;   // read fd is still open
19   int writeopen;  // write fd is still open
20 };
21
22 int
23 pipealloc(struct file **f0, struct file **f1)
24 {
25   struct pipe *p;
26
27   p = 0;
28   *f0 = *f1 = 0;
29   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
30     goto bad;
31   if((p = (struct pipe*)kalloc()) == 0)
32     goto bad;
33   p->readopen = 1;
34   p->writeopen = 1;
35   p->nwrite = 0;
36   p->nread = 0;
37   initlock(&p->lock, "pipe");
38   (*f0)->type = FD_PIPE;
39   (*f0)->readable = 1;
40   (*f0)->writable = 0;
41   (*f0)->pipe = p;
42   (*f1)->type = FD_PIPE;
43   (*f1)->readable = 0;
44   (*f1)->writable = 1;
45   (*f1)->pipe = p;
'pipe.c" 121L, 2411C                                          30,2          1 +43
```

struct file:→ which either points to a actual file(inode pointer is used) else a pipe pointer is used

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};


// 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;          // inode has been read from disk?

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

when you read from a file, read is done through sys_read, which calls fileread

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}

int
sys_write(void)
{
  struct file *f;
  int n;
```

fileread calls piperead, if f→type was FD_PIPE

```
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}

//PAGEBREAK!
// Write to file f.
int
filewrite(struct file *f, char *addr, int n)
{
```

this piperead is like a queue with 2 indices that are nread and nwrite

```
}

int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

similar code for pipewrite, which is called by filewrite, which is called by sys_write

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}

int
piperead(struct pipe *p, char *addr, int n)
```

# paging

Viewing Abhijit's application

## Review of last class

- MMU : Hardware features for MM
- OS: Sets up MMU for a process, then schedules process
- Compiler : Generates object code for a particular OS + MMU architecture
- MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS

Viewing Abhijit's application

# Review of last class

- Different types of MMUs
  - No translation (no MMU)!
  - Base + limit (also called relocation + offset) scheme
    - Leads to what type of OS ?
  - Multiple base + limit scheme --> segmentation
    - Offers which benefits?
    - Problem: fragmentation
  - Paging – logical division of the physical memory into equal sized pages, in memory page table + PTBR in hardware, OS's job and compiler's job in this case

Viewing Abhijit's application

Review of last class

- Different types of MMUs
  - No translation (no MMU)!
  - Base + limit (also called relocation + offset) scheme
    - Leads to what type of OS ?
  - Multiple base + limit scheme --> segmentation
    - Offers which benefits?
    - Problem: fragmentation
  - Paging – logical division of the physical memory into equal sized pages, in memory page table + PTBR in hardware, OS's job and compiler's job in this case

linux kernel is dynamically linked and dynamically loader

xv6 is static loading and linking

Viewing Abhijit's application

# Dynamic Linking, Loading

- **Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking**
  - **Hence called 'link-loader'**
  - **Static or dynamic loading is still a choice**
- **Question: which of the MMU options will alllow for which type of linking, loading ?**

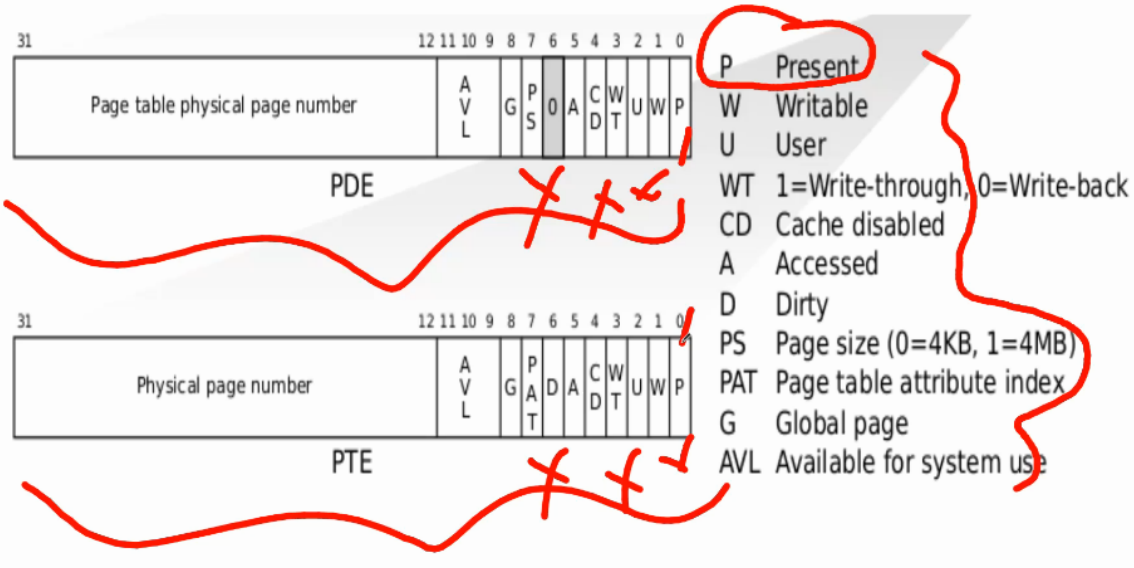
process is discontinuous is the main distinction between segmentation and paging

paging still has internal fragmentation, but the process is no longer continuous

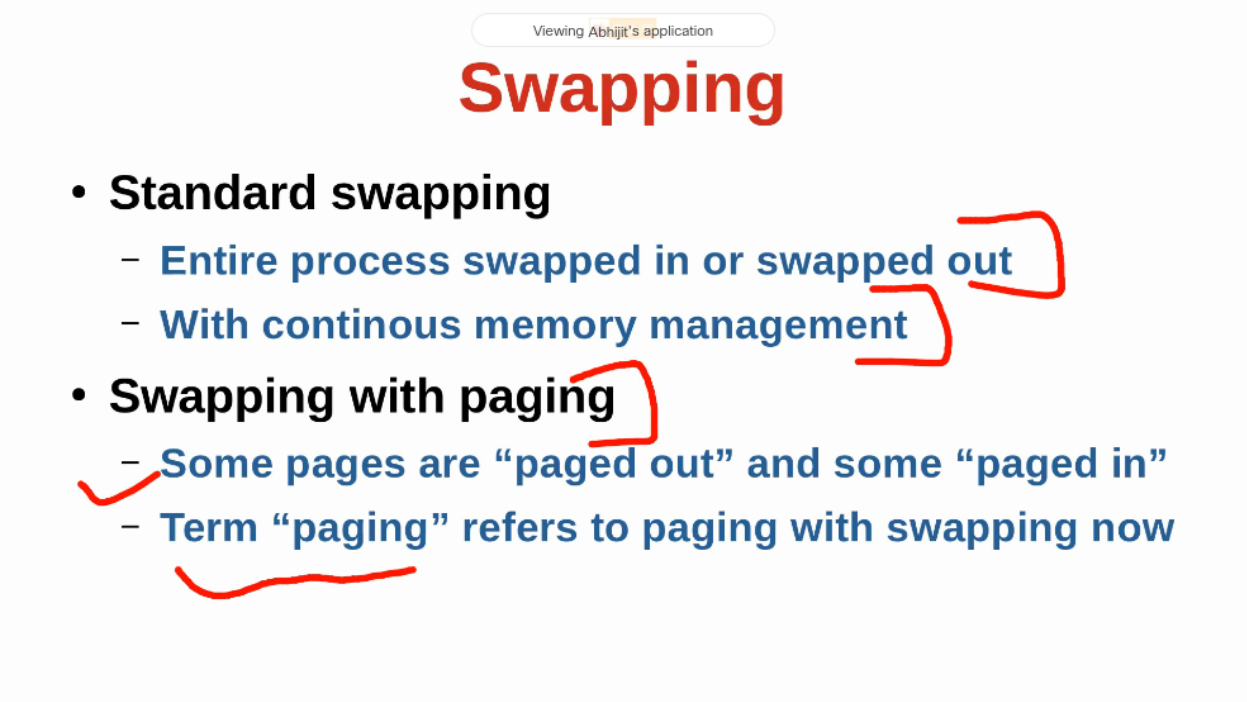
every process has a separate page table

Viewing Abhijit's application

## X86 PDE and PTE



problem with paging is page table is continuous




# 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

# /hat is virtual memory?

Virtual memory – separation of user logical memory from physical memo

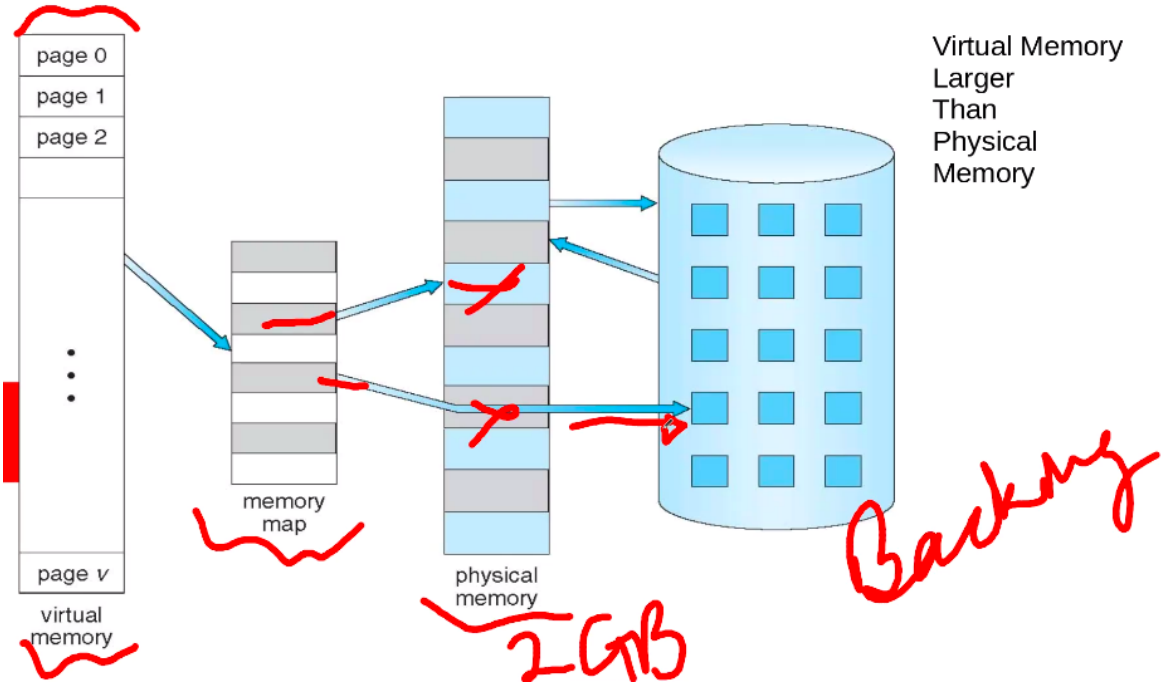
- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical ad space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

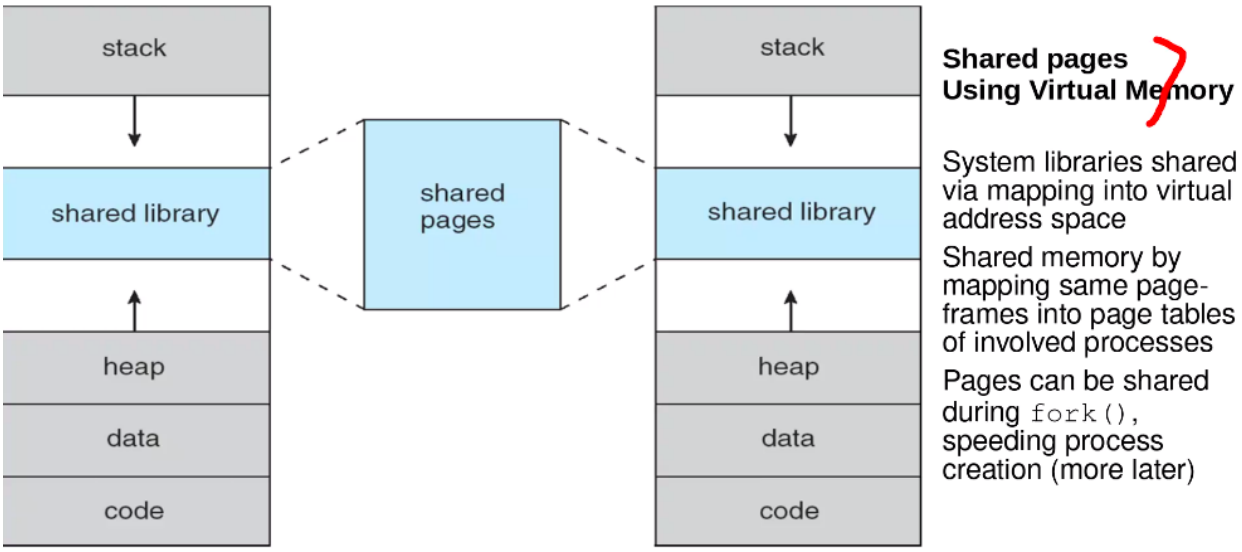
Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

---

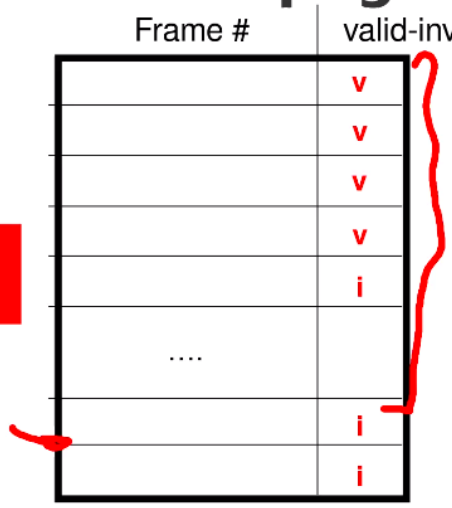
this is how Virtual memory is created,cylinder is backing store






syscall called sbrk ,which asks the kernel to allocate some region and returns a pointer to it.

## New meaning for valid/invalid bits in page table




- With each page table entry a valid–invalid bit is associated

  v: in-memory – memory resident

  i : not-in-memory or illegal

- During address translation, if valid–invalid bit in page table entry is I : **raises trap called page fault**

1 instruction could cause 3 page faults, 1 for instruction, and then for mem1,mem2

## 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**)

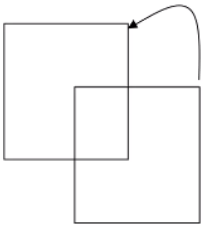
  Instruction restart

## Instruction restart

- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

# copy 20 bytes from address 0x100 to address 0x200

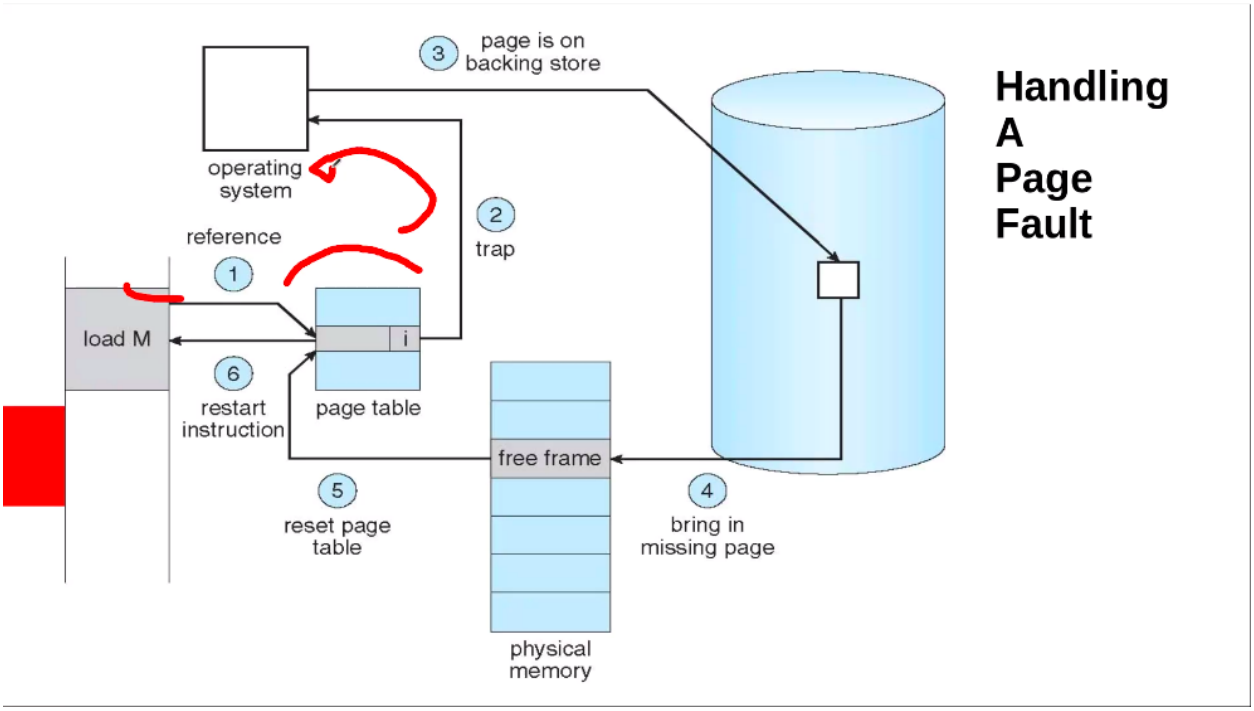


Hardware support needed for demand paging

Page table with valid / invalid bit

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

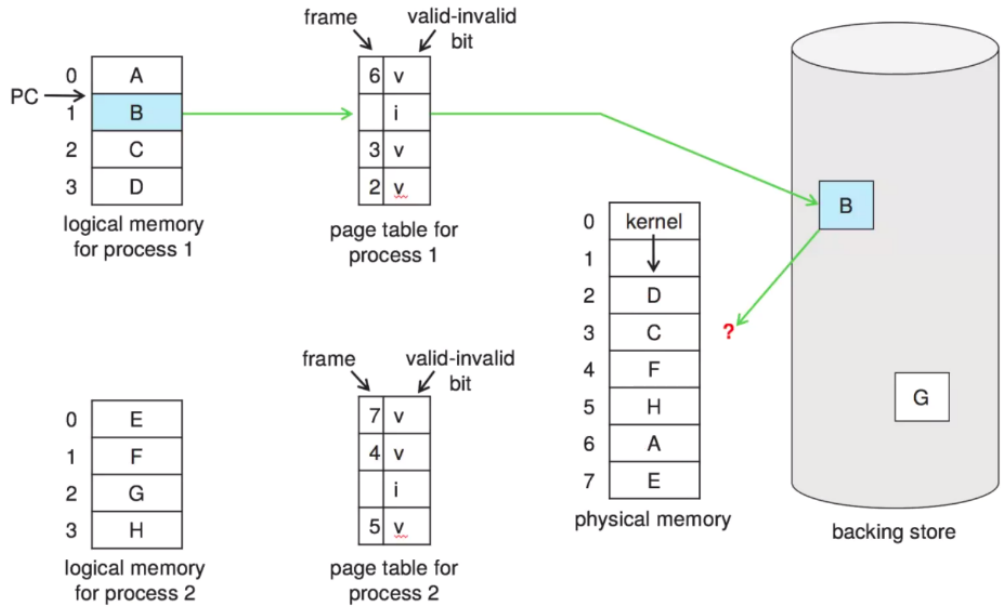
Instruction restart



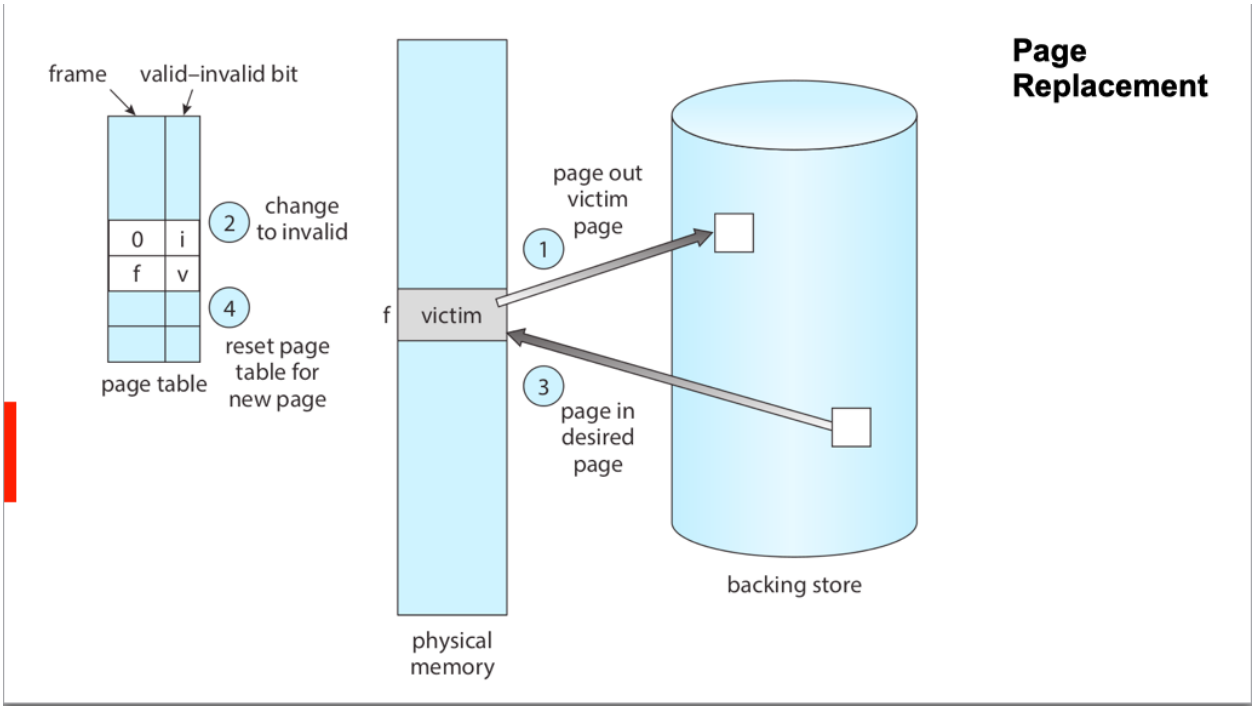

NUMBER 5 IS THE KERNEL EXECUTING,

## What if no free frame found on page fault?

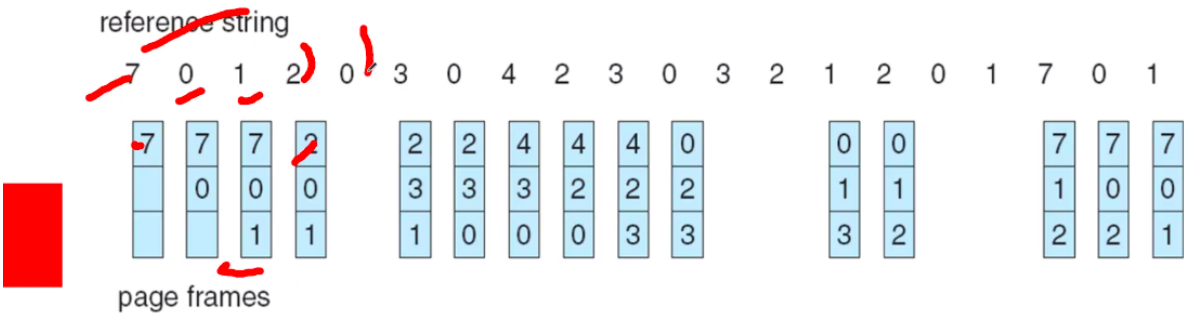
- Page frames in use depends on “Degree of multiprogramming”
  - More multiprogramming -> overallocation of frames
  - Also in demand from the kernel, I/O buffers, etc
  - How much to allocate to each process? How many processes to allow?
- Page replacement – find some page(frame) in memory, but not 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




strategies for page replacement



# FIFO Algorithm




balady's anamoly

FIFO suffers with belady's anamoly

optimal and LRU doesnt

# 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

# paging in xv6

kinit1() takes end, virtual address  of 4mb [range of address i.e virtual address-end<4mb]

why virtual address? because code for this will be made by compiler which uses virtual addresses

end is defined in kernel symbol table which is the last address of ELF file

P2V will add KERNBASE,

why is this needed? because once the paging is set up, you need to use virtual address

kinit1 calls freerange, which runs a for loop on range of virtual address as shown and calls kfree for each page

```
36   freerange(vstart, vend);
37 }
38
39 void
40 kinit2(void *vstart, void *vend)
41 {
42   freerange(vstart, vend);
43   kmem.use_lock = 1;
44 }
45
46 void
47 freerange(void *vstart, void *vend)
48 {
49   char *p;
50   p = (char*)PGROUNDUP((uint)vstart);
51   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
52     kfree(p);
53 }
54 //PAGEBREAK: 21
55 // Free the page of physical memory pointed at by v,
56 // which normally should have been returned by a
57 // call to kalloc().  (The exception is when
58 // initializing the allocator; see kinit above.)
59 void
60 kfree(char *v)
```

PGROUNDUP wiill align address to page boundary like 4k,8k,12k

kfree() is called, v is the address of the chunk that will be part of linkedlist now

struct run{ struct run *next;}

entire xv6 code uses physical memory 0 to 224mb only

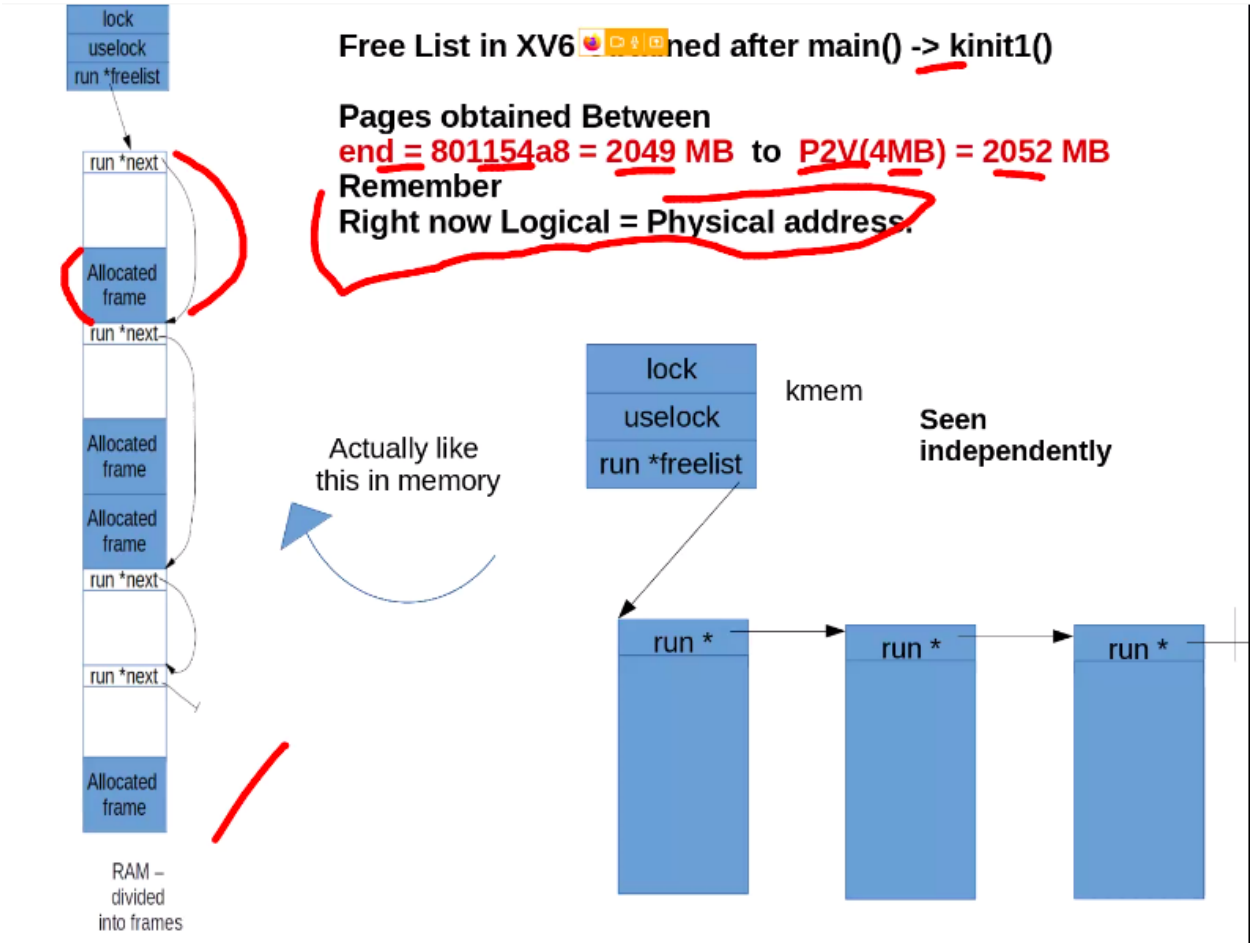
panic() runs a infinite loop after disabling interrupts, hence nothing will work after calling panic.

```
53 }
54 //PAGEBREAK: 21
55 // Free the page of physical memory pointed at by v,
56 // which normally should have been returned by a
57 // call to kalloc().  (The exception is when
58 // initializing the allocator; see kinit above.)
59 void
60 kfree(char *v)
61 {
62   struct run *r;
63
64   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
65     panic("kfree");
66
67   // Fill with junk to catch dangling refs.
68   memset(v, 1, PGSIZE);
69
70   if(kmem.use_lock)
71     acquire(&kmem.lock);
72   r = (struct run*)v;
73   r->next = kmem.freelist;
74   kmem.freelist = r;
75   if(kmem.use_lock)
76     release(&kmem.lock);
77 }
```

memset fills the entire page with ‘1’, virtual address v is typecasted into struct run*, then here is kmem,which has pointer call freelist which is global pointer, so line 72,73,74 creates a linkedlist

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

so 2052-2049== 3mb, which will have 3mb/4kb =1536 pages




we have created a free list out of 3mb.

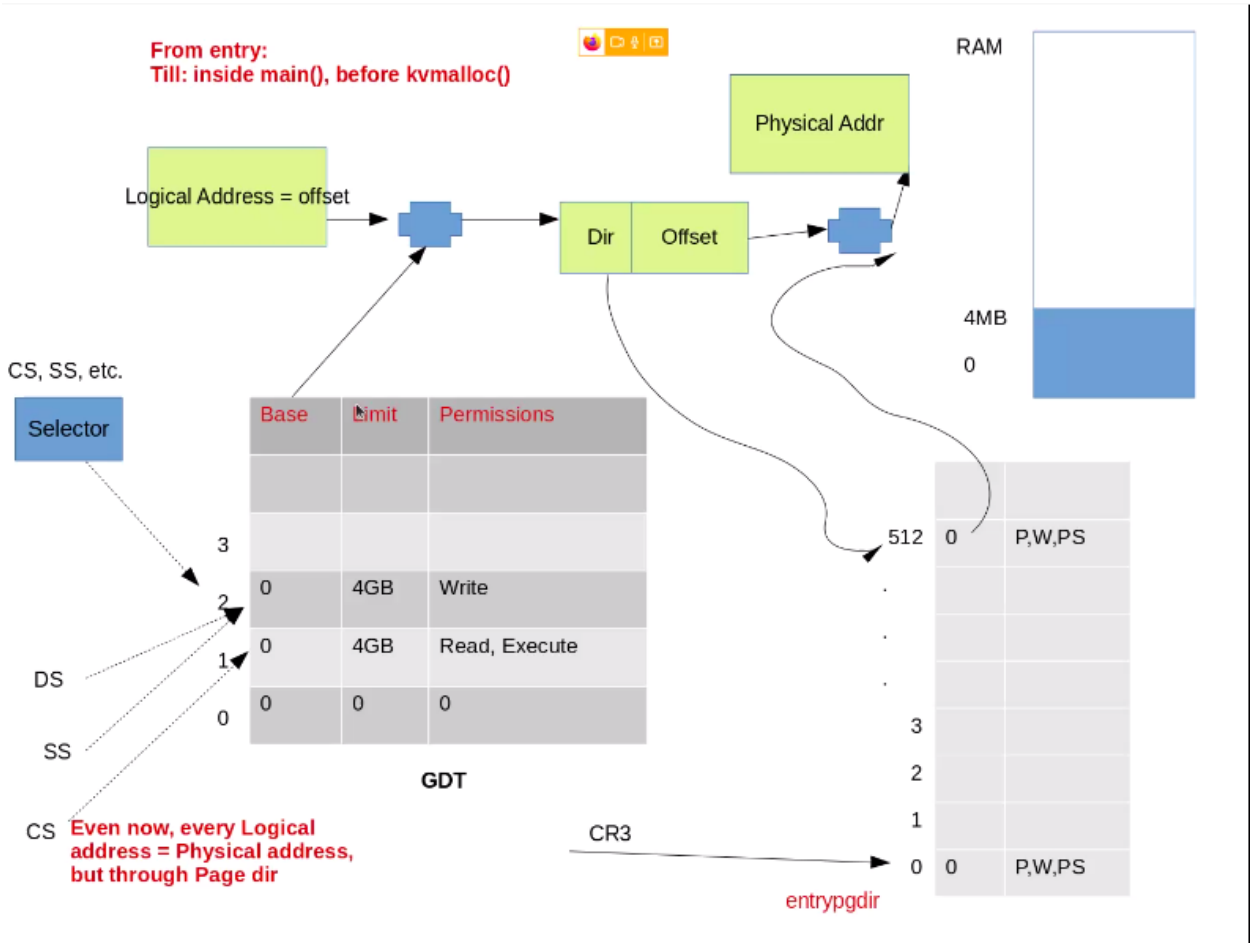
all the addresses 0 to 4mb, 2gb to 2gb+4mb will be mapped to 0 to 4mb

we also call kinit2() in bootmain()

here beginning address is 4mb, ending address is PHYSTOP

kvmalloc() will setup kernel's page table

this is the mmu before kvmalloc()




As of now only kernel is running, no applications are running.

```
7
8 // Allocate one page table for the machine for the kernel address
9 // space for scheduler processes.
0 void
1 kvmalloc(void)
2 {
3   kpgdir = setupkvm();
4   switchkvm();
5 }
6
7 // Switch h/w page table register to the kernel-only page table,
8 // for when no process is running.
9 void
0 switchkvm(void)
1 {
2   lcr3(V2P(kpgdir));   // switch to the kernel page table
3 }
n.c" 394L, 9918C
```

kvmalloc creates the mmu setup for scheduler, it calls switchkvm() lcr3() is load into cr3

setupkvm is called from many functions.

setupkvm() will setup the page directory for the kernel and it’s address is stored in kpgdir

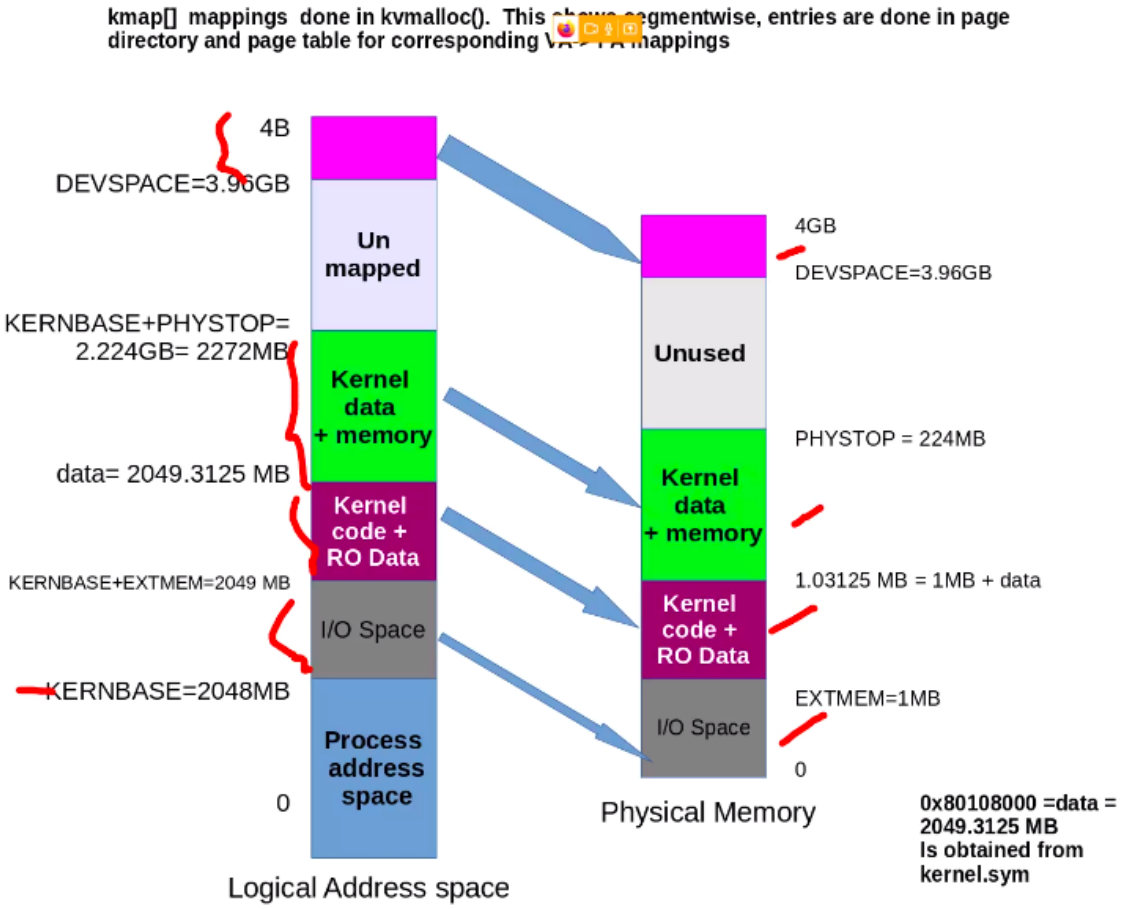
kpgdir is a global uint*

```
// Set up kernel part of a page table.
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;
}
```

k is a pointer to struct kmap

```
//                                     rw data + free physical memory
//   0xfe000000..0: mapped direct (devices such as ioapic)
//
// The kernel allocates physical memory for its heap and for user memo
// between V2P(end) and the end of physical memory (PHYSTOP)
// (directly addressable from end..P2V(PHYSTOP)).

// This table defines the kernel's mappings, which are present in
// every process's page table.
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+roo
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+mem
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

it is array of structures




EXTMEM is 1MB, which is mapped to 0 to 1MB in physical space

0 as permission is read permission

kalloc() is counterpart of kfree

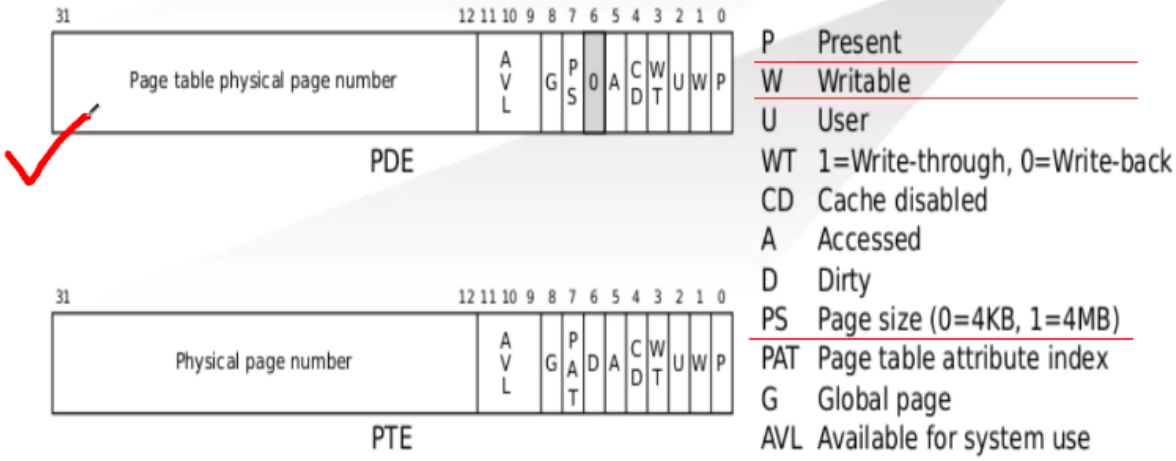
```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

it picksup the first frame on freelist and returns a pointer to it. kalloc() will give you a 4K frame

pgdir points to frame returned by kalloc()

we call memset() which sets 0 in all pgdir, so we make P=0,PS=0 i.e present entry=0, PS=0 i.e. 4KB page size

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




for loop iterates through all kmap entries and calls mappages() which creates PDE and PTE entries, allocate page tables, and setup entries in PT and PD

```
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
.c" 394L, 9918C                                                    66,1
```

inside mappages() it calls walkpgdir() for all pagedir entries

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

41.17

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

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

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

// construct virtual address from indexes and offset
```

PDX will give page directory index and PTX will give page table index

if (Present bit is 1) which is false as we memset to 0

if this was true, then you will get a page number

but in else part we allocate a 4kb page [using kalloc]

we set up a page directory entry with physical address of pgtable,with  flags

returns pointer to page table entry

then we return in mappages, which fills a entry in the page table [as *pde] with physical address pa passed as argument[page number]

which returns in setupkvm, and thus 4 entries of kernel are setup.

**summary**

walkpgdir checks if va is mapped inside the pgdir entry, if yes return entry, no then allocate a page table then return address to the entry in pgdir of that page table

kvmalloc switches from 1 level paging to 2 level paging.

kvmalloc → setupkvm → mappages → walkpgdir

if alloc is 1, walkpgdir allocates a page table and returns its address

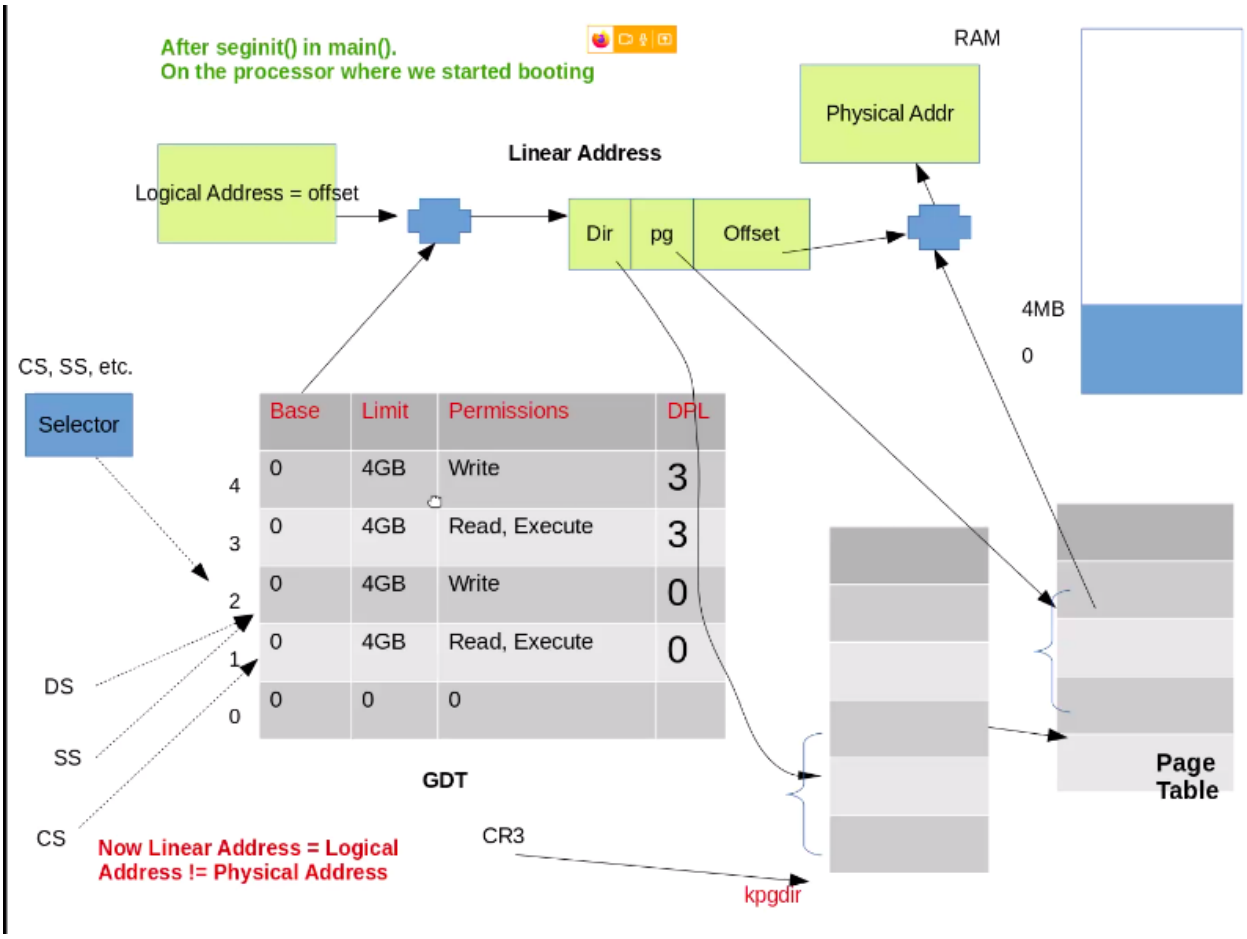
mappages fills the entry returned by walkpgdir with physical address pa ,permissions and sets Present bit.

setupkvm runs mappages for all the 4 entries of kmap.

kvmalloc sets the address of pgdir returned by setupkvm into global variable called kpgdir and loads the same address into the cr3 register as well.

```
// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));
}
```

seginit will change segmentation table, now there are 4 entries




when application is running you ensure CS,DS is 3 and 4, for kernel is 1 and 2

- **Thread-local storage** (**TLS**) 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)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static** data
  - TLS is unique to each thread

## Scheduler activations for threads

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

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