slides4

Kinit1, kinit2 -> free list

Kalloc,kfree ->managing free frame list

setupkvm()

Kpgdir-> entries of pgdir point to page tables,

Can we move seginit to line 35?

userinit() -> creation of user processes

userinit() creates init process, it is an application process, we have to set it up without using fork,

_binary_initcode_start[] is a starting address of 'start’ symbol in binary file called initcode, similarly _binary_initcode_size[] is the size (2c)

It can be searched in kernel.sym

kernel: $(OBJS) entry.o entryother initcode kernel.ld

$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

- b means include symbols from these files

Essentially initcode.S is the initcode file.

We are creating a process called initcode, whose only job is to exec init

userinit calls allocproc()

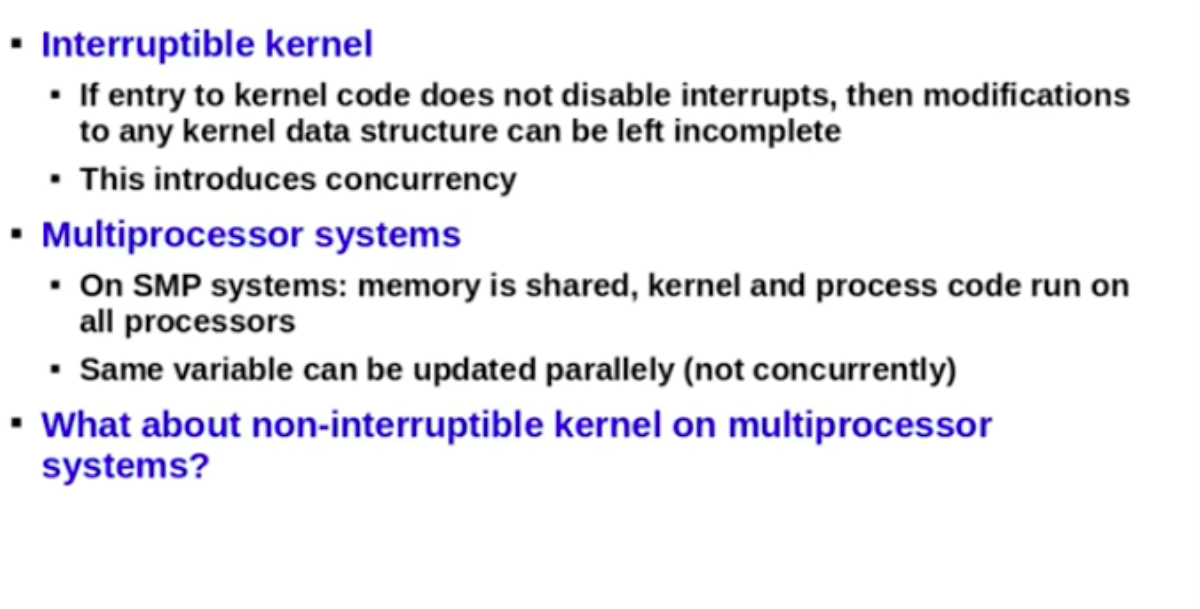
```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  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);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // 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;

  return p;
}
```

nextpid is global variable initialized to 1.

Only way a process is created is using trap

synchronization

reason for races



non-interupptible kernel on multiprocessor system doesnt work because then there is nopoint in using multiprocessor system.

non-interrruptible kernel on uniprocessor will solve the problem of races, but then you lose concurrency.

Expected solution characteristics

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

peterson’s solution mai assume both the flags are false in the begining.

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;
}
```

some rules for spinlock

never block a process which holds the spinlock

hold a spinlock only for short period of time.

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 therad to wait queue
  - The thread holding the lock will wake up one of them

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;
```

ncli keeps a count of number of locks are being held

while acquiring spinlocks always disable interrupts.

if any other process is holding the same lock on the same cpu, then acquire calls panic(),note we can hold locks of different types on same cpu no problem, that is the count ncli, whereas we cannot acquire same lock twice on same cpu

__sync_synchronize() is a memory barrier instruction

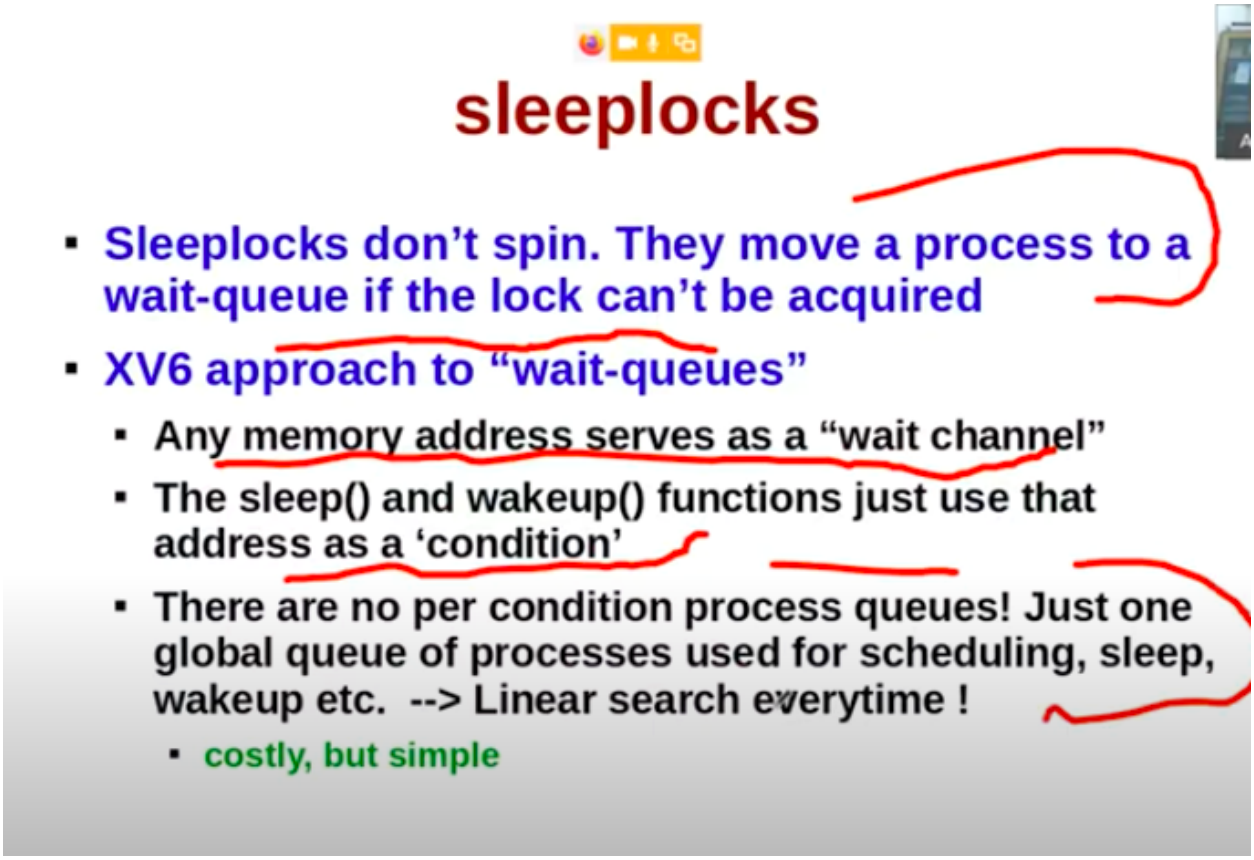
sleep function must release the lock before it calls scheduler.

RULE:if a spinlock is used by an interrupt handler then then processor must never hold that spin-lock with interrupts enabled, or it will lead to a deadlock.

why you should disable interrupts before acquiring a spinlock?

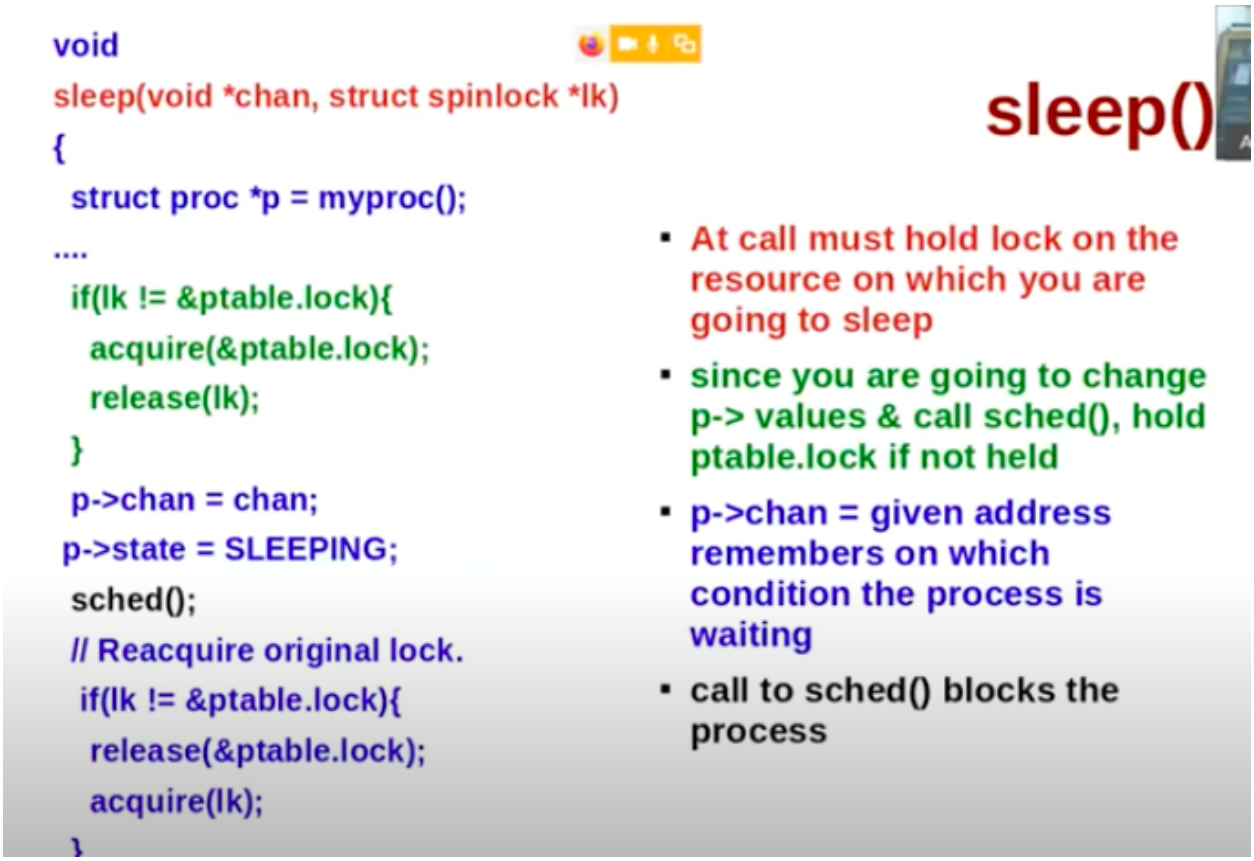
consider a situation of iderw() which acquires idelock,now since interrupts are enabled, after acquiring the lock, ideintr() is called as a result of interrupt and it also tries to

acquire(idelock), but will busy wait, since iderw() has never realeased the spin lock so this is a deadlock situation created here.



chan is a variable used by xv6 to indicate that process is waiting for the address pointed by this variable.

remember: whenever chan is not null, state ==WAITING.



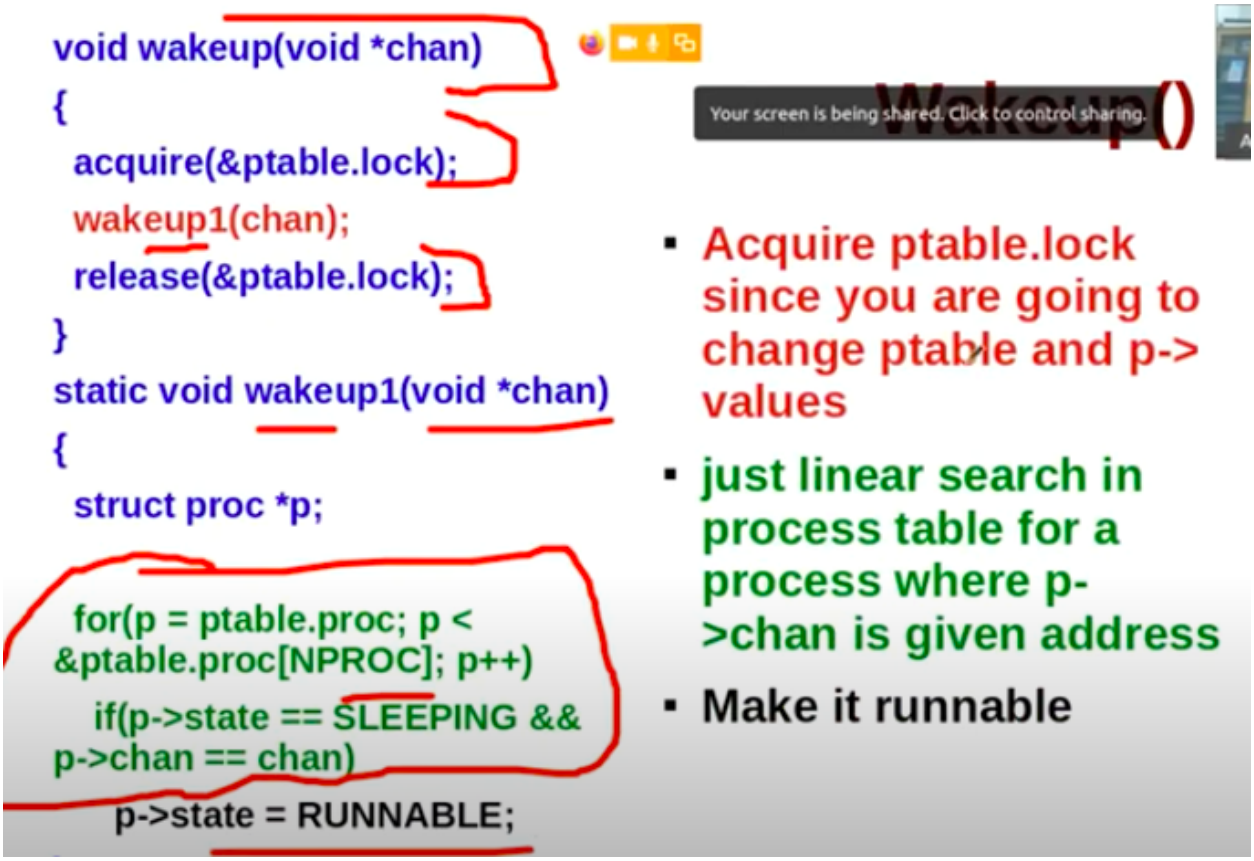
if chan is address of inode, then process is waiting for the inode to be read.

sleep() changes the state to WAITING(SLEEPING inxv6)

```
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);  //race here
  }
  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);
}

int
holdingsleep(struct sleeplock *lk)
{
```

sleep will reacquire the spinlock of sleeplock at the end of sleep() so that there is no race at acquiring sleeplock again in while(lk→locked) sleep(lk,&lk→lk)

```
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  /* Abhijit: the below will push c->scheduler, then address of p->context, and then return address Y on sta
ck */
  swtch(&p->context, mycpu()->scheduler);
  /* Address, say, Y */
  /* Abhijit: the control comes here, only from scheduler() !, not otherwise */
  mycpu()->intena = intena;
}
```


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;
Your screen is being shared. Click to control sharing.
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

RULE: Never hold a spinlock if the process is about to block. this rule is broken by ptable.lock, but in sched we check that we only hold ptable.lock and ncli==1 which means no other spinlock is hold,except the ptable.lock

ptablelock is released in yield, for ptable lock there is an exception that it can be hold even if the process is about to block, and sched() checks if ncli≠1, which makes sure that only ptable is allowed for this exception

on a uniprocessor system, sleep lock/blocking lock is much better than a spin lock

sleeplocks are used in only 2 places, struct buf and struct 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.

sleeplocks cannot be used by interrupt handler because if we are running a interrupt handler we want the interrupts to be disabled, thus ide intr handler,console intr handler cannot use sleeplocks

Semaphore

wait() and signal() , orignally called P() and V()

wait(S){

while (S≤0);

S- -;

}

signal(S){

S++;

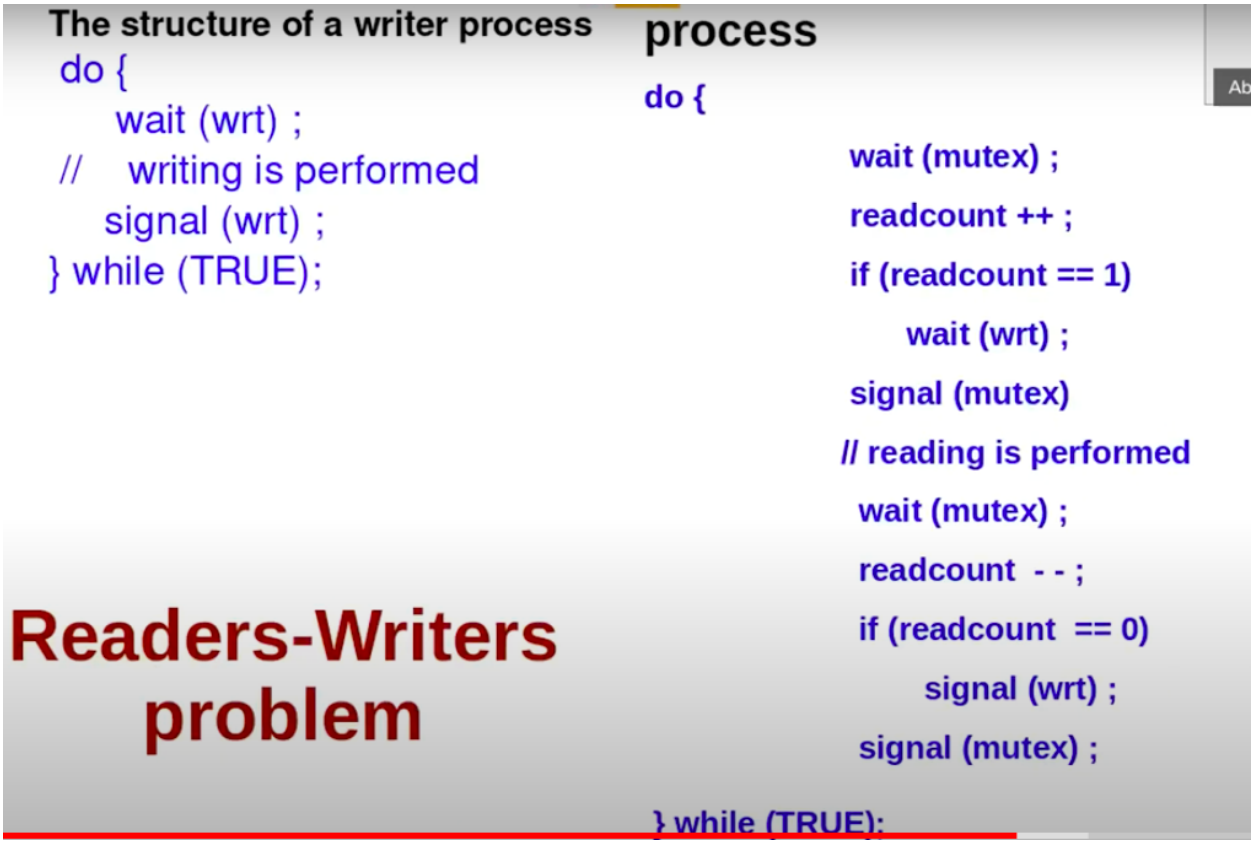
}

cycle in graph is necessary condition but not sufficient condition for deadlock

acquire loglock before idelock.

classic synchronization problems-

reader writer problem



above code gives preference to writers,aswriter can directly wait() and do job then signal, whereas reader has to first acquire mutex, then acquire wrt then do reading, also if a wrt==0, 1st reader has acquired mutex, 2nd will wait untill 1st releases mutex, which is waiting on wrt for writer to complete writing.

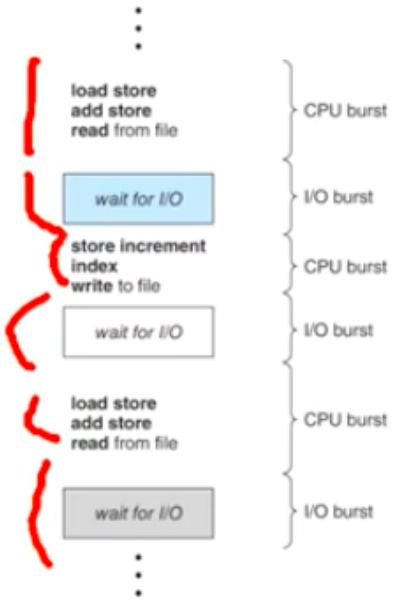
condition is just a wait queue.

Volume groups are abstraction of partition on hard disks

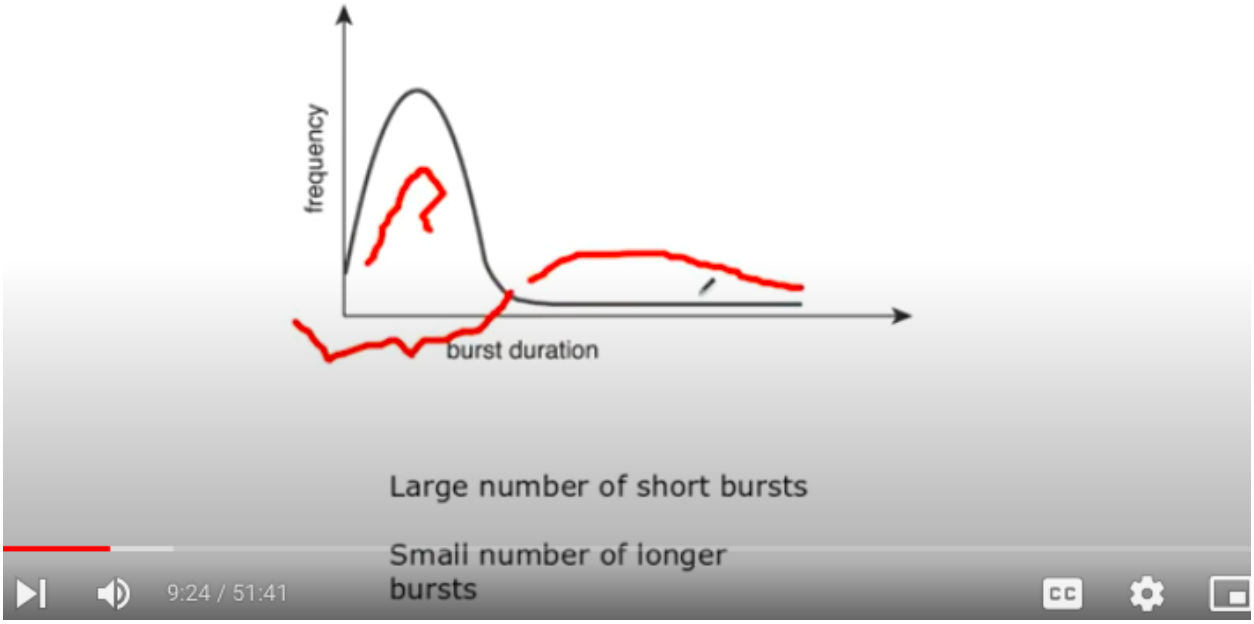
ACL is access control lists

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



CPU bursts: observation



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

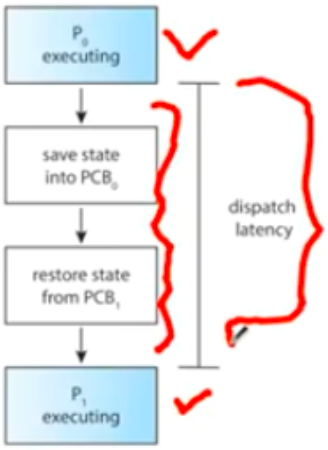
- Scheduling under 1 and 4 is nonpreemptive
- All other scheduling is preemptive
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

if interrupts are allowed then it is preemptive scheduling else it is non-preemptive scheduling

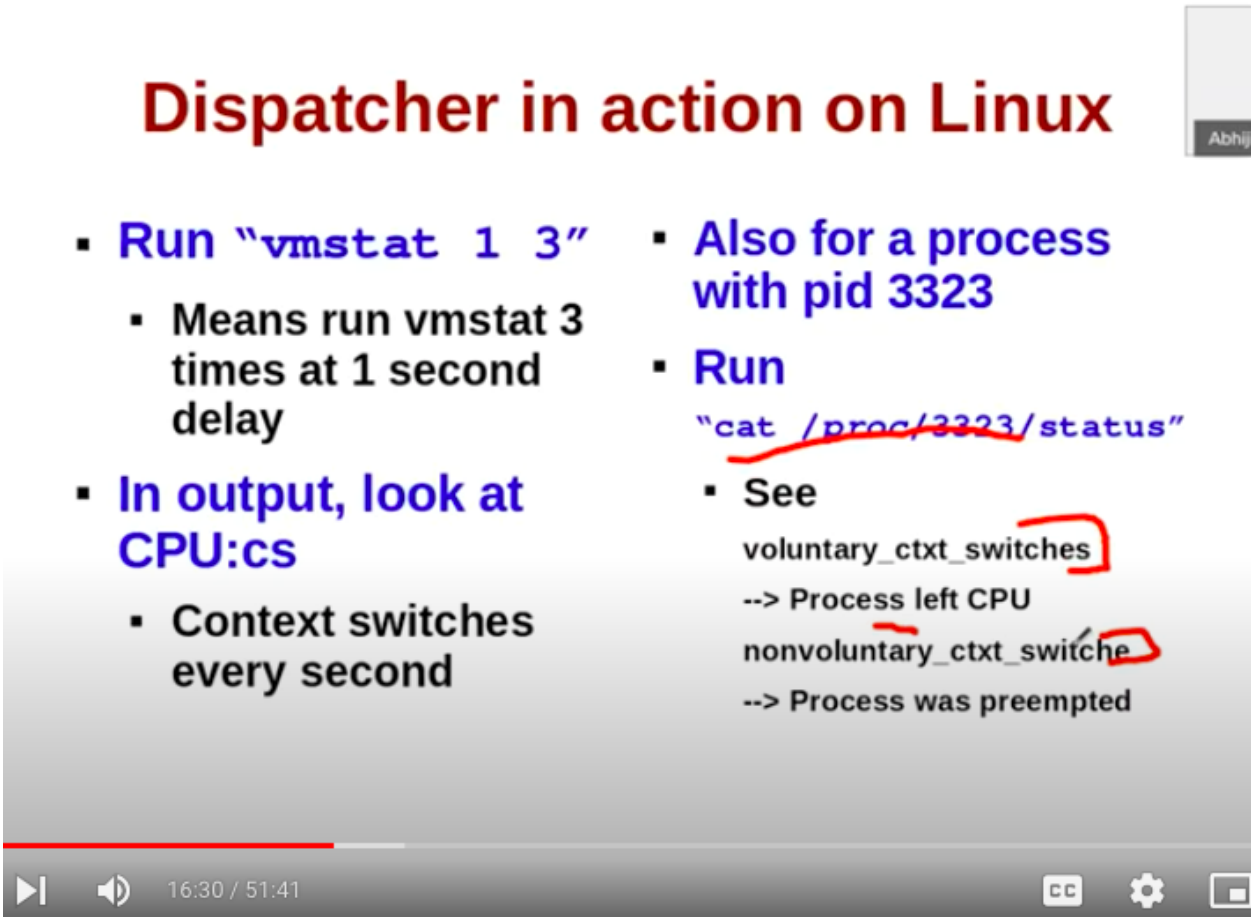
scheduler in xv6 is a short term scheduler.

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(), trap(), trapret()



Dispatcher in action on Linux



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 !

throughput=total process complete / total time

waiting time is amount of time spent in ready queue and is the part of turnarround time.

scheduling algorithms

convoy effect in FCFS

FCFS are bad for interactive process, and response time is bad for these algorithms.

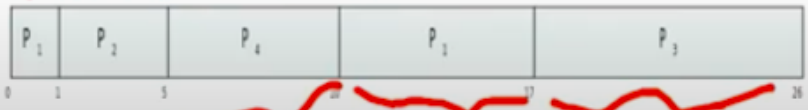
shortest job first is an optimal algorithm

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



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