Threads

Thread a fundamental unit of cpu utilization.
Parallelism is different from concurrency,
With concurrency it is progress at same time,done by scheduler.
Parallelism needs multiprocessors [runs 2 functions at same time on 2 processors]

Kernels are generally multithread

Single vs multithreaded process

Single threaded process has code,data,files, register,stack.
In multithreaded, seperate registers,program counter and stack are for each threads,code data, files are shared among threads

Difference between process and threads

Process is heavy weight, while thread is light weight.

In multicore systems, each core will have seperate timer interrupts.

User vs Kernel Threads

User threads will give a typedef called threads, the scheduling of threads to be implemented user level library.
Systemcalls->
In many-one model, if one thread does blocking task, all threads are blocked, as process is assumed to block by the kernel

Many-many ,one-one will solve this problem
Tow level model is some are many-many, some one-one

clone-> syscall to create a child process, which takes argument called flags, fork duplicates all, whereas clone can select what exactly you want to clone.
Fork is a wrapper on clone with some flags setup.
clone(files,fs,code,..)

Thread libraries

Library like pthreads.follow syntax of pthreads in your project to make it POSIX compliant.

Issues with threads

Does fork() duplicate only calling thread, or all threads?

-> depends on OS
Thread cancellations[]terminating a thread before it has finished]
Cancel immediately
Check periodicallly and cancel itself.

Clone duplicates currently running process, with our requirements,
By default clone() duplicates stack, which duplicates context of process.

For many-one you should not use clone() syscall, using only 1 kernel thread you have to run threads.

For this you need a notion of eip,struct thread to keep a track of all the threads

pthread_create(…,..,fn,….){
t=mallloc(th);
threads[i]=t;
swtch();//library functions available
//similar to xv6's scheduler
}
Struct threads{
Char stack[];
Context c;
Function ptr;
}
First use these libraries and then write those again.
Determine what is context and done.

Library functions->
Getcontext,setcontext, makecontext

Longjump and setjump to change eip values.

Signals

Synchronous and asynchronous
Signal handler processes signals

signal.c

```c
#include <stdio.h>
#include <signal.h>
```

```c
int *p = 1234, i = 1234;
void seghandler(int signo)
{
    printf("seg fault occured \n");
    return;
}
void inthandler(int signo)
{
    printf("INT signal received\n");
    return;
}
int main()
{
    signal(SIGINT, inthandler);
    getchar();
    i = 10;
    signal(SIGSEGV, seghandler);
    *p = 100;
    return 0;
}
```

Ctrl +z ->SIGSTOP
Ctrl +c -> SIGINT
SEGMENT FAULT-> SIGSEV
SIGCONT should make the process continue
int kill(pid_t pid,int sig);

man 7 signals

SIGKILL is number 9
kill -9 <pid>
kill -s SIGINT <pid>
SIGCONT -> continue process after ctrl+z
strace kill -s SIGINT <pid>
strace will show all the syscalls that comand makes

terminal
➜ OS kill -s SIGINT 12569
➜ OS kill -s SIGSTOP 12569
➜ OS kill -9 12569

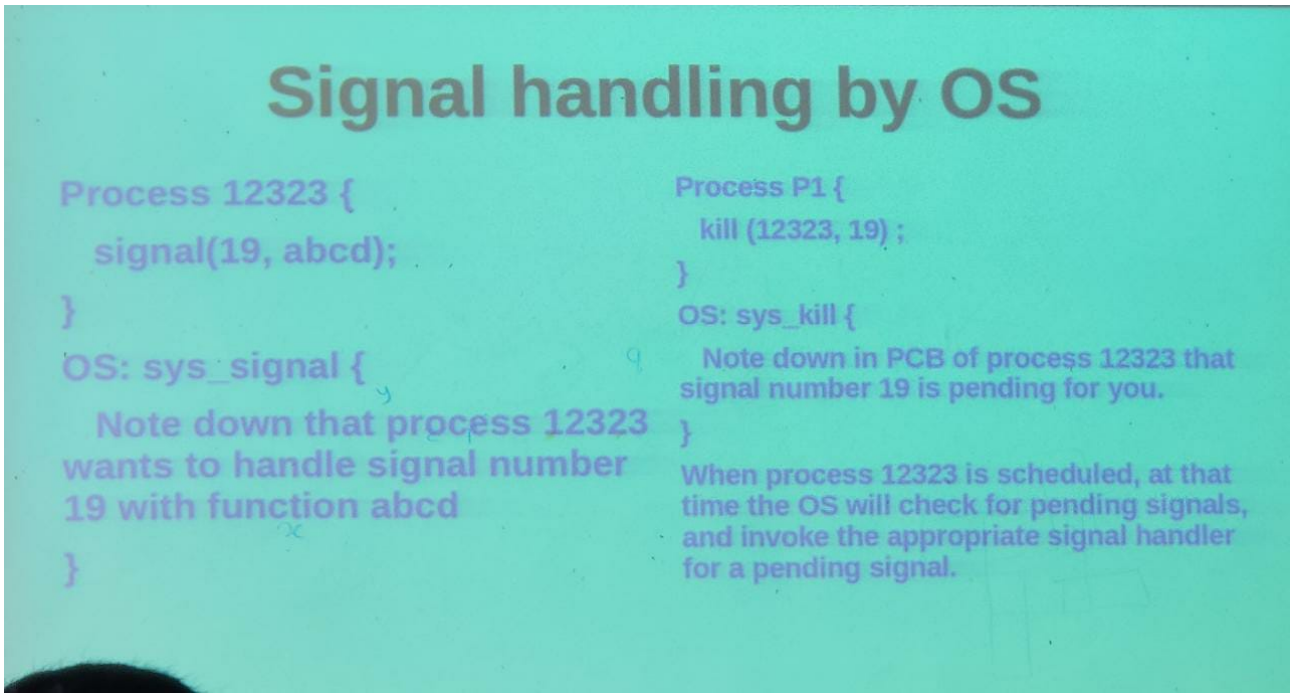
signal() is no longer used, posix uses sigaction()
Signals are identified using numbers too
kill() is used by a process to send another process a signal.
one user cannot send signals to other user's process

man 7 signal
SIGCHILD is sent by kernel to process when child dies

SIGUSR1 is user-defined signal ,send to dd will show statistics

SIGALRM is important for project, it does something like timer

Inside struct proc, the user program that should be run on signal is noted



Thread pools

Thread in a thread pool will not run function immediately, it is going to wait, the point is chrome know with every thread needs 4 threads,if it didn't creates n number of threads initially, then it may happen enough resource is not available, hence it holds some number of threads when it begins and thus becomes faster, it allows the number of threads in application to be bound to size of pool

Thread local storage

main(){
th(f)
th(g)
}

Global variables are shared among all threads, which may create problem,
So we will have a data global to all functions of a thread

[global is accessible in all files, global static is accessed inside the same file only]
Solved by thread-local storage

Scheduler activations for threads

Suppose all the kernel threads in the many-many mapping gets blocked, then how will the application know that the kernel threads are blocked, so kernel has to tell the application, so some communication from kernel to user is required[backward communication]
For this we have scheduler activations
It provides upcalls which is a backward communication mechanism from kernel to thread library

So between user thread and kernel thread is a lightweight process,
How does this solve?
LWP is a part of kernel data structure,

In linux there is no difference between thread and process, there is only 1 term called task

See fork's syscall using strace

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++)
// UNUSED=0
    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

The data structure used in kalloc() and kfree() in xv6 is
. Singly linked NULL terminated list

In xv6, The struct context is given as
struct context {
  uint edi;
  uint esi;
  uint ebx;
uint ebp;
uint eip; };
Select all the reasons that explain why only these 5 registers are included in the struct context.
->The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Hierarchical Paging → More memory access time per hierarchy, Inverted Page table → Linear/Parallel search using page number in page table, Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically

the status of segmentation setup in xv6
entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor

Page fault handling
Process is kept in wait state → 6, The reference bit is found to be invalid by MMU → 1, Restart the instruction that caused the page fault → 9, Disk interrupt wakes up the process → 7, A hardware interrupt is issued → 2, Page tables are updated for the process → 8, OS makes available an empty frame → 4, OS schedules a disk read for the page (from backing store) → 5, Operating system decides that the page was not in memory → 3

correct statements about sched() and scheduler() in xv6 code
sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

return a free page, if available; 0, otherwise → kalloc(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Setup kernel part of a page table, and switch to that page table → kvmalloc(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

paging with demand paging and select the correct s
Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The kernel code and data take up less than 2 MB space: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

xv6 uses physical memory upto 224 MB only: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
Cumulative size of all programs can be larger than physical memory size: True One Program's size can be larger than physical memory size: True
Logical address space could be larger than physical address space: True Relatively less I/O may be possible during process execution: True
Virtual access to memory is granted: False Virtual addresses are available: False
Code need not be completely in memory: True

the actions done as part of code of swtch() in swtch.S
Switch from old process context to new process context: False
Restore new callee saved registers from user stack of new context: False Jump to code in new context: False
Save old callee saved registers on kernel stack of old context: True
Switch from one stack (old) to another(new): True
Save old callee saved registers on user stack of old context: False
Restore new callee saved registers from kernel stack of new context: True

which explains how paging works.
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The physical address may not be of the same size (in bits) as the logical address: True The page table is indexed using frame number: False
The page table is itself present in Physical memory: True
The page table is indexed using page number: True
The PTBR is present in the CPU as a register: True
Maximum Size of page table is determined by number of bits used for page number: True

statements about MMU and it's functionality

MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

the compiler's view of the process's address space, for each of the following MMU schemes:
Segmentation, then paging → many continuous chunks of variable size, Paging → one continuous chunk, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk

a processor supports base(relocation register) + limit scheme of MMU. The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are: end, P2V(PHYSTOP)

Order events in xv6 timer interrupt code
alltraps() will call iret → 18, Process P2 is executing → 14, change to context of the scheduler, scheduler's stack in use now → 11, trap() is called → 7, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, P2's yield() will return in trap() → 16, Change of stack from user stack to kernel stack of P1 → 3, Trapframe is built on kernel stack of P1 → 6, jump to alltraps → 5, P1 is marked as RUNNABLE → 9, jump in vector.S → 4, P2 will return from

sched() in yield() → 15, P2 is selected and marked RUNNING → 12, Timer interrupt occurs → 2, Process P1 is executing → 1, sched() is called, → 10, yield() is called → 8

```
lockexcl(rwlock *1) {
        P(l->mutex);
        while (1->nReaders <0 or 1->nReaders > 0) {
        V(l->mutex);
        P(l->pendingWriters);
        P(l->mutex);
        }
        l->nReaders =-1;
        P(l->mutex);
}
 lockshared (rwlock *1){
        P(I-> mutex);
        while (nReaders < 0) {
        v(1->mutex); // not holding mutex here
        P(I->pending Readers); // not holding mutex
        P(I->mutex);
        } else {
        nReaders++;
        }
        v(I-> mutex);
}
```

```
unlockexcl(rwlock *1) {
        P(I->mutex);
        while(l->nPendingReaders){
        l->nPendingReaders--;
        V(l->pendingReaders); // wakeup readers
        }
        //assert: l->nPendingReaders=0
        if(nPendingwriters){
        L->nPendingwriters--;
        V(l->pendingwriters)
        }
        V(l->mutex);
        }
        unlockshared (rwlock *1) {
        P(I->mutex);
        I->nReaders--;
        if(1->nReaders == 0) {
        if(1->nPendingWriters > 0)
        V(I->pendingWriters);
        }
        V(I->mutex);
        }
        Typedef Struct read-write lock{
         int nReaders;
        int nPending Readers;
        int nPendingWriters;
        seamphore mutex;
```

semaphore pendingReaders;

semaphore pendingWriters

}rwlock;

initlock(rwlock *1) {

initsemaphore (I->mutex,1);

initsemaphore (I->pendingReaders,10);

initsemaphore (I-> pendingWriters,1);

nReaders = nPendingReaders = nPendingWriters = 0;

}

Cat /proc/partitions gives the partition list

Fdisk is used to create partition

Mkfs to create filesystem

Data block bitmap is 1 block ,if size of block =4k then 32kBits, thus 32k blocks is the size of the block group, including superblock of size 1 block and so on.

Fsck is consistency checking recovery related program

begin_op() is related to logging that indicates transaction is about to begin
end_op() marks transaction end
File system functions

Root inode number is 1,block size is 512 in xv6

There are 12 direct blocks and 1 indirect block in xv6

Thus 512/4 +12-> 140 blocks -> 140blocks * 512=70KB max file size

Ideinit -> initializes disk dirver,checks disk1(fs.img) is present
Iderw-> input or output,called by bread, bwrite
Idestart->called by iderw, conversion from block number to sector number,assure sector per block<7, and calls outsl which takes data from buf to output port if dirty,else read
Idewait->checks if disk controller is ready to accept next request, called by idestart
Ideintr->called by interrupt handler when I/o is completed
Insl and outsl are loop instructions

Xv6 assumes that the I/o interrupt always occurs for the first buffer on the queue
If it is first I/o then iderw calls idestart,else every ideintr calls idestart
Iderw does read and write depending on the flags specifically B_DIRTY flag.
Bget-> reads a buffer for block number ,it checks if buf is already cached, if found use, else use a new buffer,increments refcnt
Binit->initializes buffer cache
Bread-> calls bget, to get a buffer, if VALID is not set then calls iderw() else directly returns.
Bwrite-> sets the flag to B_DIRTY and calls iderw
For new buffer, pick the buffer in LRU fashion, but the list is mantained in MRU fashion, so previous pointers will give a list in LRU fashion,start from tail and use prev pointers
Dev field of struct dev is device number, as there may be multiple devices of same type so same device driver can be used to read from those buffers.
The job of buffer cache layer is to avoid unnecessary disk I/O which is done mainly by bread,when it checks VALID flag.
Bcache list always contains 30 buffers, NBUF=30.

Brelse-> it simply decrements refcnt, and removes the buffer from its current position and place it at the begining, to mantain a MRU list
Initlog initializes the in memory struct log

begin_op-> increments log.outstanding,indicating one more FS syscall has been called.
end_op->decrements log.outstanding,and if 0 it will set do_commit,and call commit()
log_write-> only writes block number in the array, doesn't write data in front of log array yet.

write_log-> called by commit, it writes data in front of log array from bcache. [copy data blocks in log area]
write_head->wrote first_log block on the disk

init_log->calls recovery_from_log which is main function to recover fs
LOGSIZE=30
log.outstanding will be atmost 3

Balloc-> gives a empty block from Devno. log_write called by it writes the bitmap block
Bfree->finds the right bitmap block and clears the right bit

Inode

major_num is device number which is only console in xv6

Iget-> gets inode from inode_num, it doesnt read inode from the disk. It finds in icache,increments ref and returns, if not found returns empty inode.

Iupdate->copy a modified in memory inode to disk.
Inode cache is write through

dinode is the diskinode struct
Ilock-> if valid bit ==0 then reads the inode from disk to in memory inode

Iput-> decrements ref, but if ref==1 then calls itrunc, valid=0,iupdate()

Itrunc->writes all data block to disk using bfree

If type of inode ==0, then it is a free inode,since xv6 doesnt have inode bitmap, 3types T_DIR =1, T_FILE->2, T_DEV=3

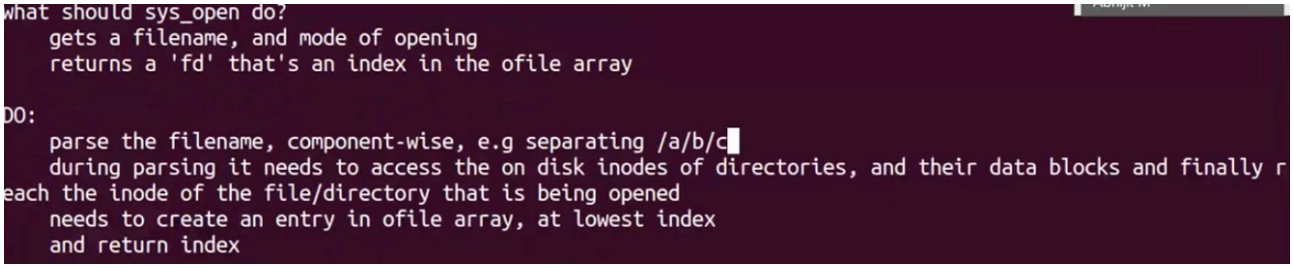
Ialloc-> finds a free inode In device, and calls log_write.

Bmap->allocates nth block for file given by inode. Allocate block on disk and store it in direct entry or indirect entry.

If type =T_DEV it calls read and write from devsw[] array which is initialized in consoleinit to consoleread,consolewrite

Size of struct dirent in 16 bytes, name=14bytes

namei returns a inode pointer from path
namei calls namex(path,0,name)
Idup increments ref and returns same inode
Skipelem -> cut path on '/' and give next eg see comment
Dirlookup-> returns inode ptr to name that is passed, sets poff to byte offset of entry in directory



Filealloc-> returns empty struct file from global array of struct file
Fdalloc->sets lowest array element to point to struct file that we got from filealloc
Create-> nameiparent->returns pointer to parentdir

Dirlink,links file newly created to create a entry in parentdir
sys_write writes max 3 blocks at a time 1536 bytes

Map the function in xv6's file system code, to it's perceived logical layer.
namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time. Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

All processes accessing the same file share the file descriptor among themselves: No
The pointer entry in the file descriptor array entry points to the data of the file directly: No
There is only one global file structure per on-disk file.: No
All file accesses are made using only global variables: No
The 'file offset' is shared among all the processes that access the file.: No
No synchronization is implemented so that changes are made available immediately.: No
A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No
There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes

The file descriptors in every PCB are pointers to the same global file structure.: No
The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No
All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes
The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Statements about scheduling and scheduling algorithms
The nice() system call is used to set priorities for processes: True
Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True
In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True
xv6 code does not care about Processor Affinity: True
In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
Response time will be quite poor on non-interruptible kernels: True
Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Pre-emptive scheduling leads to many race conditions in kernel code.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Map ext2 data structure features with their purpose
Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore

consistency, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Combining file type and access rights in one variable → saves 1 byte of space, rec_len field in directory entry → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, Inode bitmap is one block → limits total number of files that can belong to a group, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, A group → Try to keep all the data of a directory and it's file close together in a group

Mark whether the concept is related to scheduling or not.
timer interrupt: Yes context-switch: Yes ready-queue: Yes file-table: No
runnable process: Yes

This question is based on your general knowledge about operating systems/related concepts and their features.
Java thereads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Map the block allocation scheme with the problem it suffers from (Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others
Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical partitions: True

Mark statements True/False w.r.t. change of states of a process. Reference: The process state diagram (and your understanding of how kernel code works)
A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True
Every process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

The given semaphore implementation faces which problem? Assume any suitable code for signal()
Note: blocks means waits in a wait queue.

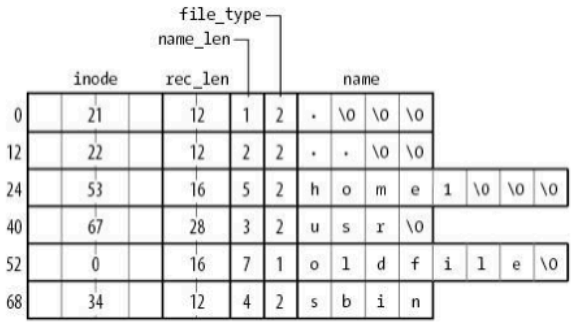
```
struct semaphore { int val;
spinlock lk;
};
sem_init(semaphore *s, int initval) {
s->val = initval; s->sl = 0;
}
wait(semaphore *s) {

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

Ans: deadlock

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer. For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence. Assume that the kernel code is non-interruptible and uniprocessor system.
Process P1 executing a system call Timer interrupt
Generic interrupt handler runs Scheduler runs
Scheduler selects P2 for execution
P2 returns from timer interrupt handler Process p2, user code executing

This sequence of events is: not-possible Because

Consider the structure of directory entry in ext2, as shown in this diagram.



Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)%4)
- b. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (strlen(name)- 4) %4
- c. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + 4 - (strlen(name)%4)
- d. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)
- e. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) %4
- f. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + strlen(name)

Your answer is incorrect.

The correct answer is: rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)

Select all the correct statements about log structured file systems. are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

The permissions -rwx--x--x on a file mean
: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side
is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

All statements are in the context of preventing deadlocks.
A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Circular wait is avoided by enforcing a lock ordering: True
Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Consider this program.
Some statements are identified using the // comment at the end.
Assume that = is an atomic operation.
#include <stdio.h> #include <pthread.h> long c = 0, c1 = 0, c2 =
void *thread1(void *arg)
while(run == 1) {//E c = 10; //A
c1 = c2 + 5; //B }
}
void *thread2(void *arg)
while(run == 1) {//F c = 20;//C
c2 = c1 + 3;//D }

}
int main() {
pthread_t th1, th2; pthread_create(&th1, pthread_create(&th2,
sleep(2);
run = 0; fprintf(stdout, "c = fflush(stdout);
}
0, run = 1; {
{
NULL, thread1, NULL); NULL, thread2, NULL);
%ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
Which statements are part of the critical Section?
F: No D: Yes C: No A: No B: Yes E: No

Mark the statements about device drivers by marking as True or False.

It's possible that a particular hardware has multiple device drivers available for it.: True xv6 has device drivers for IDE disk and console.: True
A disk driver converts OS's logical view of disk into physical locations on disk.: True
A device driver code is specific to a hardware device: True
All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True Device driver is an intermediary between the end-user and OS: False

Select all the correct statements about bootloader.
The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.
The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid
The correct answer is: 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");
Atomic compare and swap instruction (to be expanded inline into code) ?
If you don't do this, a process may be running on two processors parallely ?
Tell compiler not to reorder memory access beyond this line ?

return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void sleep(void *chan, struct spinlock *lk)
{
...
if(lk != &ptable.lock){
acquire(&ptable.lock);
release(lk);
} → Avoid a self-deadlock, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize(); → Tell compiler not to reorder memory access beyond this line