Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23

Show one page at a time

Finish review

Started on	Thursday, 20 April 2023, 7:42 AM
State	Finished
Completed on	Thursday, 20 April 2023, 8:06 AM
Time taken	23 mins 7 secs
Grade	**17.81** out of 30.00 (**59.37**%)

Question **1**

Partially correct

Mark 1.75 out of 2.00

Flag question

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
panic(char *s)
{
...
 panicked = 1;
```

Ensure that no printing happens on other processors ✔

```
void
acquire(struct spinlock *lk)
{
...
 getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls ✔

```
static inline uint
xchg(volatile uint *addr, uint
newval)
{
 uint result;

 // The + in "+m" denotes a read-
modify-write operand.
 asm volatile("lock; xchgl %0, %1" :
          "+m" (*addr), "=a"
(result) :
          "1" (newval) :
          "cc");
 return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
acquire(struct spinlock *lk)
{
 pushcli();
```

Disable interrupts to avoid deadlocks ✔

```
struct proc*
myproc(void) {
...
 pushcli();
 c = mycpu();
 p = c->proc;
 popcli();
```

Disable interrupts to avoid another process's pointer being returned ✔

```
popcli();
...
}
```

```
void
sleep(void *chan, struct spinlock
*lk)
{
 ...
 if(lk != &ptable.lock){
  acquire(&ptable.lock);
  release(lk);
 }
```

Release the lock held by some another process

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: void

```
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, void

```
panic(char *s)
{
...
 panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
...
 getcallerpcs(&lk, lk->pcs);
```

→ Traverse ebp chain to get sequence of instructions followed in functions calls, static inline uint

```
xchg(volatile uint *addr, uint newval)
{
 uint result;

 // The + in "+m" denotes a read-modify-write operand.
 asm volatile("lock; xchgl %0, %1" :
          "+m" (*addr), "=a" (result) :
          "1" (newval) :
          "cc");
 return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
acquire(struct spinlock *lk)
{
 pushcli();
 → Disable interrupts to avoid deadlocks, struct proc*
myproc(void) {
...
 pushcli();
 c = mycpu();
 p = c->proc;
 popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, 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

Question **2**
Incorrect
Mark 0.00 out of 1.00
Flag question

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. log may be kept on same block device or another block device ✔
- [ ] b. file system recovery may end up losing data
- [ ] c. file system recovery recovers all the lost data
- [x] d. a transaction is said to be committed when all operations are written to file system ✘
- [x] e. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔

Your answer is incorrect.

The correct answers 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

Question **3**
Correct
Mark 1.00 out of 1.00
Flag question

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
void
acquire(struct spinlock *lk)
{
...
 getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls ✔

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
panic(char *s)
{
...
 panicked = 1;
```

Ensure that no printing happens on other processors ✔

Your answer is correct.

The correct answer is: void
acquire(struct spinlock *lk)
{

```
...
 getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls, void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process, void
panic(char *s)
{
...
 panicked = 1; → Ensure that no printing happens on other processors
```

Question **4**

Partially correct

Mark 1.50 out of 2.00

Flag question

Select all the correct statements about synchronization primitives.

Select one or more:

- [x] a. Spinlocks are good for multiprocessor scenarios, for small critical sections ✔
- [x] b. Blocking means moving the process to a wait queue and calling scheduler ✔
- [x] c. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [ ] d. Semaphores are always a good substitute for spinlocks
- [x] e. Mutexes can be implemented using blocking and wakeup ✔
- [ ] f. All synchronization primitives are implemented essentially with some hardware assistance.
- [x] g. Spinlocks consume CPU time ✔
- [ ] h. Mutexes can be implemented without any hardware assistance
- [ ] i. Blocking means one process passing over control to another process
- [ ] j. Mutexes can be implemented using spinlock
- [x] k. Thread that is going to block should not be holding any spinlock ✔
- [ ] l. Blocking means moving the process to a wait queue and spinning

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question **5**

Partially correct

Mark 0.88 out of 1.00

Flag question

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- [x] a. Recovery is possible due to redundancy in file system data structures ✔
- [x] b. It is possible to lose data as part of recovery ✔
- [x] c. Even with a write-through policy, it is possible to need a recovery program. ✔
- [x] d. A recovery program, most typically, builds the file system data structure and checks for inconsistencies ✔
- [x] e. They may take very long time to execute ✔
- [ ] f. They are used to recover deleted files
- [ ] g. Recovery programs are needed only if the file system has a delayed-write policy.
- [x] h. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✔
- [x] i. They can make changes to the on-disk file system ✔

Your answer is partially correct.

You have correctly selected 7.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **6**
Correct
Mark 1.00 out of 1.00
Flag question

Assuming a 8- KB page size, what is the page numbers for the address 428517 reference in decimal :
(give answer also in decimal)

Answer: 52 ✔

The correct answer is: 52

Question **7**
Correct
Mark 1.00 out of 1.00
Flag question

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.
Consider each request as first request.

150 KB, best fit	200 KB	✔
50 KB, worst fit	300 KB	✔
200 KB, first fit	300 KB	✔
220 KB, best fit	250 KB	✔
150 KB, first fit	300 KB	✔
100 KB, worst fit	300 KB	✔

The correct answer is: 150 KB, best fit → 200 KB, 50 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 220 KB, best fit → 250 KB, 150 KB, first fit → 300 KB, 100 KB, worst fit → 300 KB

Question **8**
Partially correct
Mark 0.25 out of 1.00
Flag question

Match the code with it's functionality

S1 = 0; S2 = 0; P2: Statement1; Signal(S2); P1: Wait(S2); Statemetn2; Signal(S1); P3: Wait(S1); Statement S3;	Counting semaphore	✘
S = 5 Wait(S) Critical Section Signal(S)	Binary Semaphore for mutual exclusion	✘
S = 1 Wait(S) Critical Section Signal(S);	Binary Semaphore for mutual exclusion	✔
S = 0 P1: Statement1; Signal(S) P2: Wait(S) Statment2;	Execution order P3, P2, P1	✘

Your answer is partially correct.
You have correctly selected 1.
The correct answer is: S1 = 0; S2 = 0;
P2:
Statement1;

```
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3; → Execution order P2, P1, P3, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion, S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2
```

Question **9**

Partially correct

Mark 0.80 out of 1.00

Flag question

Mark the statements as True or False, w.r.t. thrashing

True	False		
☑	☐	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔
☐	☒	During thrashing the CPU is under-utilised as most time is spent in I/O	✖
☑	☐	Thrashing can be limited if local replacement is used.	✔
☐	☑	mmap() solves the problem of thrashing.	✔
☑	☐	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
☐	☑	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✔
☑	☐	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
☑	☐	The working set model is an attempt at approximating the locality of a process.	✔
☐	☑	Thrashing occurs because some process is doing lot of disk I/O.	✔
☒	☐	Thrashing can occur even if entire memory is not in use.	✖

Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing can be limited if local replacement is used.: True
mmap() solves the problem of thrashing.: False
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

The working set model is an attempt at approximating the locality of a process.: True
Thrashing occurs because some process is doing lot of disk I/O.: False
Thrashing can occur even if entire memory is not in use.: False

Question **10**
Partially correct
Mark 0.76 out of 2.00
Flag question

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. Demand paging always increases effective memory access time.
- [x] b. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] c. Both demand paging and paging support shared memory pages. ✔
- [ ] d. Demand paging requires additional hardware support, compared to paging.
- [x] e. With paging, it's possible to have user programs bigger than physical memory. ✖
- [x] f. Paging requires some hardware support in CPU ✔
- [ ] g. Paging requires NO hardware support in CPU
- [x] h. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] i. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [ ] j. TLB hit ration has zero impact in effective memory access time in demand paging.

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: 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.

Question **11**
Correct
Mark 1.00 out of 1.00
Flag question

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- [ ] a. Journals must be maintained on the same device that hosts the file system
- [x] b. The purpose of journal is to speed up file system recovery ✔
- [x] c. Journals are often stored circularly ✔
- [x] d. Most typically a transaction in journal is recorded atomically (full or none) ✔
- [ ] e. A different device driver is always needed to access the journal
- [ ] f. Journal is hosted in the same device that hosts the swap space
- [x] g. the journal contains a summary of all changes made as part of a single transaction ✔

Your answer is correct.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **12**
Correct
Mark 1.00 out of 1.00
Flag question

For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No		
☑	☐		✔

: Yes

: Yes

Question **13**

Incorrect

Mark 0.00 out of 1.00

Flag question

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 1101001000

The correct answer is: 11011110

Question **14**

Incorrect

Mark 0.00 out of 1.00

Flag question

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
```

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

too much spinning, bounded wait not guaranteed

```
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));
}
```

blocks holding a spinlock

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

not holding lock after unblock

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

deadlock

Your answer is incorrect.

The correct answer is:

```
struct semaphore {
        int val;
```

```
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

```
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));
}
```

→ deadlock,

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

→ blocks holding a spinlock,

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed

Question **15**
Correct
Mark 1.00 out of 1.00
Flag question

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5.00 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **16**
Correct
Mark 2.00 out of 2.00
Flag question

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- a. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✔
- b. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✔
- c. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup. ✔
- d. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems. ✔
- e. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- f. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✔
- g. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- h. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✔

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **17**
Complete
Mark 1.00 out of 3.00
Flag question

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a) int chown(char* path, int owner_id, int group_id);

(b) int sys_chown(void);

(c) in syscall.c add "[SYS_chown] sys_chown," and in user.h add declaration "int chown(char*, int, int);" and in usys.S add SYSCALL(chown) and in syscall.h add #define SYS_chown [num+1], add int sys_chown in sysfile.c and int chown in fs.c

(d) NIL

(e) chown() - changes the owner and group of a file.

(f) $(UPROG)/chown\ and the declaration at the end of makefile

Comment:

Question **18**
Incorrect
Mark 0.00 out of 1.00
Flag question

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms, The effective memory access time in nanoseconds is:

Answer: 5600 ✗

The correct answer is: 5600060.00

Question **19**
Correct
Mark 1.00 out of 1.00
Flag question

Map the technique with it's feature/problem

static linking	large executable file	✓
dynamic linking	small executable file	✓
dynamic loading	allocate memory only if needed	✓
static loading	wastage of physical memory	✓

The correct answer is: static linking → large executable file, dynamic linking → small executable file, dynamic loading → allocate memory only if needed, static loading → wastage of physical memory

Question **20**
Partially correct
Mark 0.63 out of 1.00
Flag question

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
☐	✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✗
☑	☐	The arguments are accessed in the kernel code using esp on the trapframe.	✓
✗	☐	Integer arguments are stored in eax, ebx, ecx, etc. registers	✗
✗	☐	The arguments to system call are copied to kernel stack in trapasm.S	✗
☐	☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✓
☑	☐	The arguments to system call originally reside on process stack.	✓
☑	☐	The functions like argint(), argstr() make the system call arguments available in the	✓

		kernel.	
☑	✗	Integer arguments are copied from user memory to kernel memory using argint()	✔

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The arguments are accessed in the kernel code using esp on the trapframe.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
The arguments to system call are copied to kernel stack in trapasm.S: False
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call originally reside on process stack.: True
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
Integer arguments are copied from user memory to kernel memory using argint(): True

Question **21**
Complete
Mark 0.25 out of 2.00
Flag question

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a) void* kalloc(int n); void kfree(void* addr);

(b) int sys_buddy(void);

(c) kinit1(KERNBASE, PHYSTOP);

(d) structrun { structrun *next; int len; }; structrun *freelist[24]; char* heap_end;

(e) NIL

(f) NIL

(g) NIL

Comment:

Question **22**
Correct
Mark 1.00 out of 1.00
Flag question

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

True	False			
☑	✗	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
☑	✗	A logical volume can be extended in size but upto the size of volume group	✔	
☑	✗	The volume manager stores additional metadata on the physical disk partitions	✔	
☑	✗	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP
☑	✗	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
☑	✗	A logical volume may span across multiple physical volumes	✔	

A volume group consists of multiple physical volumes ✔

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
The volume manager stores additional metadata on the physical disk partitions: True
A logical volume may span across multiple physical partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A logical volume may span across multiple physical volumes: True
A volume group consists of multiple physical volumes: True

Question **23**
Incorrect
Mark 0.00 out of 1.00
Flag question

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- a. A process blocks in many-one model even if a single thread makes a blocking system call
- b. A process may not block in many-one model, if a thread makes a blocking system call ✖
- c. many-one model can be implemented even if there are no kernel threads
- d. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- e. one-one model can be implemented even if there are no kernel threads
- f. one-one model increases kernel's scheduling load ✔
- g. many-one model gives no speedup on multicore processors

Your answer is incorrect.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Finish review

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1 2 3 4 5
6 7 8 9 10
11 12 13

Show one page at a time

Finish review

Started on	Friday, 31 March 2023, 6:58 PM
State	Finished
Completed on	Friday, 31 March 2023, 7:00 PM
Time taken	1 min 14 secs
Grade	**5.10** out of 15.00 (**33.97**%)

Question **1**
Incorrect
Mark 0.00 out of 1.00
Flag question

The lines

```
if(ip->type != T_DIR){
  iunlockput(ip);
  return 0;
}
```

in namex() function

mean

- a. ilock is held on the inode, and hence it's an error if it is a directory
- b. One of the sub-components on the given path name, did not exist, hence it's an error
- c. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return ✗
- d. One of the sub-components on the given path name, was not a directory, hence it's an error
- e. There was a syntax error in the pathname specified
- f. No directory entry was found for the file to be opened, hence an error ✗
- g. One of the sub-components on the given path name, was a directory, but it was not supposed to be a directory, hence an error ✗

Your answer is incorrect.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

Question **2**
Partially correct
Mark 0.42 out of 1.00
Flag question

Select all the actions taken by iget()

- a. Returns the inode locked
- b. Returns an inode with given dev+inode-number from cache, if it exists in cache ✔
- c. Returns a valid inode if not found in cache
- d. Panics if inode does not exist in cache ✗
- e. Returns a free-inode , with dev+inode-number set, if not found in cache
- f. Returns the inode with inode-cache lock held
- g. Returns the inode with reference count incremented ✔

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question **3**

Suppose an application on xv6 does the following:

Partially correct

Mark 0.50 out of 1.00

Flag question

```
int main() {
  char arr[128];
  int fd = open("README, O_RDONLY);
  read(fd, arrr, 100);
}
```

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True	False			
✗	✓	The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers	✗	No. data is copied into arr.
✗	✓	The loop in readi() will always read a different block using bread()	✗	No. the offsets can overlap over the same block
✓	✗	value of fd will be 3	✓	
✓	✗	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread	✓	There is no other device file in xv6
✗	✓	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers	✗	It will transfer from kernel buffer to user memory (arr)
✓	✗	The process will be made to sleep only once	✓	Yes. Reading 100 bytes means reading only one disk block. So bread()->iderw() is called only once.

The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers: False
The loop in readi() will always read a different block using bread(): False
value of fd will be 3: True
The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread: True
The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False
The process will be made to sleep only once: True

Question **4**

Partially correct

Mark 1.50 out of 2.00

Flag question

Marks the statements as True/False w.r.t. "struct buf"

True	False			
✓	✗	A buffer can be both on the MRU/LRU list and also on idequeue list.	✗	
✗	✓	B_VALID means the buffer is empty and can be reused	✓	No. it means it contains data, same as the data on disk
✓	✗	The reference count (refcnt) in struct buf is = number of processes accessing the buffer	✓	
✓	✗	B_DIRTY flag means the buffer contains modified data	✓	
✓	✗	Lock on a buffer is acquired in bget, and released in brelse	✓	
✗	✓	The "next" pointer chain gives the buffers in LRU order	✗	No. MRU order.
✓	✗	The buffers are maintained in LRU order, in the function brelse	✓	
✗	✓	A buffer can have both B_VALID and B_DIRTY flags set	✓	only one will be set

A buffer can be both on the MRU/LRU list and also on idequeue list.: True
B_VALID means the buffer is empty and can be reused: False
The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True
B_DIRTY flag means the buffer contains modified data: True
Lock on a buffer is acquired in bget, and released in brelse: True
The "next" pointer chain gives the buffers in LRU order: False
The buffers are maintained in LRU order, in the function brelse: True
A buffer can have both B_VALID and B_DIRTY flags set: False

Question **5**
Partially correct
Mark 0.92 out of 1.00
Flag question

Map the function in xv6's file system code, to it's perceived logical layer.

dirlookup	directory	✔
balloc	logging	✘
sys_chdir()	system call	✔
skipelem	pathname lookup	✔
stati	inode	✔
ialloc	inode	✔
ideintr	disk driver	✔
namei	pathname lookup	✔
filestat()	file descriptor	✔
commit	logging	✔
bread	buffer cache	✔
bmap	inode	✔

Your answer is partially correct.

You have correctly selected 11.
The correct answer is: dirlookup → directory, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, stati → inode, ialloc → inode, ideintr → disk driver, namei → pathname lookup, filestat() → file descriptor, commit → logging, bread → buffer cache, bmap → inode

Question **6**
Incorrect
Mark 0.00 out of 1.00
Flag question

Select all the actions taken by ilock()

- [x] a. Read the inode from disk, if needed ✔
- [x] b. Lock all the buffers of the file in memory ✘
- [ ] c. Take the sleeplock on the inode, always
- [x] d. Get the inode from the inode-cache ✘
- [x] e. Take the sleeplock on the inode, optionally ✘
- [x] f. Mark the in-memory inode as valid, if needed ✔
- [ ] g. Copy the on-disk inode into in-memory inode, if neeed

Your answer is incorrect.

The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if neeed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question **7**
Incorrect
Mark 0.00 out of 1.00
Flag question

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. ext4 is a log structured file system ✘ it's a journaled file system, not log structured
- [x] b. log structured file systems considerably improve the recovery time ✔

b. log structured file systems considerably improve the recovery time

c. ext2 is by default a log structured file system

d. xv6 has a log structured file system ✔

e. file system recovery recovers all the lost data

Your answer is incorrect.

The correct answers are: xv6 has a log structured file system, log structured file systems considerably improve the recovery time

Question **8**

Partially correct

Mark 1.43 out of 2.00

Flag question

Select T/F w.r.t physical disk handling in xv6 code

True	False		
✔	✘	only 2 disks are handled by default	✔
✘	✔	device files are not supported	✘
✔	✘	the superblock does not contain number of free blocks	✔
✔	✘	log is kept on the same device as the file system	✔
✔	✘	disk driver handles only one buffer at a time	✔
✘	✔	only direct blocks are supported	✘
✔	✘	The code supports IDE, and not SATA/SCSI	✔

only 2 disks are handled by default: True
device files are not supported: False
the superblock does not contain number of free blocks: True
log is kept on the same device as the file system: True
disk driver handles only one buffer at a time: True
only direct blocks are supported: False
The code supports IDE, and not SATA/SCSI: True

Question **9**

Incorrect

Mark 0.00 out of 1.00

Flag question

An inode is read from disk as a part of this function

a. ilock

b. sys_read

c. iget

d. readi ✘

e. iread

Your answer is incorrect.

The correct answer is: ilock

Question **10**

Partially correct

Mark 0.33 out of 1.00

Flag question

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False		
✔	✘	xv6 contains journal, ext2 does not	✔
✘	✔	Ext2 contains superblock but xv6 does not.	✘

✖	☑	In both ext2 and xv6, the superblock gives location of first inode block	✖
☑	✖	Ext2 contains group descriptors but xv6 does not	✔
✖	☑	xv6 contains inode bitmap, but ext2 does not	✖
✖	☑	Both xv6 and ext2 contain magic number	✖

xv6 contains journal, ext2 does not: True
Ext2 contains superblock but xv6 does not.: False
In both ext2 and xv6, the superblock gives location of first inode block: False
Ext2 contains group descriptors but xv6 does not: True
xv6 contains inode bitmap, but ext2 does not: False
Both xv6 and ext2 contain magic number: False

Question **11**
Incorrect
Mark 0.00 out of 1.00
Flag question

Arrange the following in their typical order of use in xv6.

1. ✖ ilock
2. ✖ iput
3. ✖ iget
4. ✖ iunlock
5. ✖ use inode

Your answer is incorrect.

Grading type: Relative to the next item (including last)

Grade details: 0 / 5 = 0%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

Question **12**
Incorrect
Mark 0.00 out of 1.00
Flag question

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 30 ✖

The correct answer is: 71680

Question **13**
Incorrect
Mark 0.00 out of 1.00
Flag question

Match function with it's functionality

dirlink	Lookup (search) for a given directory	✖
dirlookup	return on-disk inode for a given pathname	✖

nameiparent	Lookup (search) for a given directory ✖
namex	return in-memory inode for parent directory of a given pathname ✖

Your answer is incorrect.

The correct answer is: dirlink → Write a new entry in a given directory, dirlookup → Search a given name in a given directory, nameiparent → return in-memory inode for parent directory of a given pathname, namex → return in-memory inode for a given pathname

Finish review

◄ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management

Jump to...

(Random Quiz - 7) Pre-Endsem Quiz ►

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1 2 3 4 5
6 7 8 9 10

Show one page at a time

Finish review

Started on	Thursday, 9 March 2023, 6:20 PM
State	Finished
Completed on	Thursday, 9 March 2023, 7:30 PM
Time taken	1 hour 10 mins
Overdue	14 mins
Grade	**2.18** out of 10.00 (**21.79**%)

Question **1**

Partially correct

Mark 0.30 out of 1.00

Flag question

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode
entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [x] a. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✔
- [x] b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] c. The bootmain() code does not read the kernel completely in memory ✖
- [ ] d. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] e. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔
- [x] f. The kernel.asm file is the final kernel file ✖
- [ ] g. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [x] h. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] i. readseg() reads first 4k bytes of kernel in memory

Your answer is partially correct.

You have selected too many options.
The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **2**
Incorrect
Mark 0.00 out of 1.00
Flag question

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The ljmp instruction does a divide by 8 on the first argument
- b. The value 8 is stored in code segment
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The code segment is 16 bit and only upper 13 bits are used for segment number

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **3**
Incorrect
Mark 0.00 out of 1.00
Flag question

For each line of code mentioned on the left side, select the location of sp/esp that is in use

cli in bootasm.S	0x7c00 to 0	✗
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack'	✗
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x7c00 to 0x10000	✗
call bootmain in bootasm.S	0x10000 to 0x7c00	✗
jmp *%eax in entry.S	Immaterial as the stack is not used here	✗

Your answer is incorrect.

The correct answer is: cli
in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0);
in bootmain.c → 0x7c00 to 0, ljmp $(SEG_KCODE<<3), $start32
in bootasm.S → Immaterial as the stack is not used here, call bootmain
in bootasm.S → 0x7c00 to 0, jmp *%eax
in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack'

Question **4**
Incorrect
Mark 0.00 out of 1.00
Flag question

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- [x] a. xv6.img is the virtual processor used by the qemu emulator ✔
- [x] b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- [ ] c. The bootblock is located on block-0 of the xv6.img
- [ ] d. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- [ ] e. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- [ ] f. The size of the xv6.img is nearly 5 MB
- [x] g. Blocks in xv6.img after kernel may be all zeroes. ✗
- [ ] h. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- [ ] i. The kernel is located at block-1 of the xv6.img

- j. The size of the kernel file is nearly 5 MB
- k. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question **5**
Partially correct
Mark 0.33 out of 1.00
Flag question

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 5 entries (0 to 4) on one processor	✗
entry.S	gdt setup with 5 entries (0 to 4) on all processors	✗
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 all processors	✗
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → 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, bootasm.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

Question **6**
Incorrect
Mark 0.00 out of 1.00
Flag question

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- d. The code for reading ELF file can not be written in assembly

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **7**
Correct
Mark 1.00 out of 1.00
Flag question

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler's addresses
- b. A frame of memory that contains all the trap handler code
- c. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- d. A frame of memory that contains all the trap handler code's function pointers
- e. The IDT table
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **8**
Partially correct
Mark 0.55 out of 1.00
Flag question

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

True	False		
☑	✗	The free page-frame are created out of nearly 222 MB	✔
✗	☑	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗
☑	✗	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✔
☑	✗	The kernel code and data take up less than 2 MB space	✔
☑	✗	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
☑	✗	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✗
✗	☑	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✗
☑	✗	xv6 uses physical memory upto 224 MB only	✔
☑	✗	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✗
✗	☑	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✗
☑	✗	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔

The free page-frame are created out of nearly 222 MB: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
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 process's address space gets mapped on frames, obtained from ~2MB:224MB range: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
xv6 uses physical memory upto 224 MB only: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: 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
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

Question **9**
Incorrect
Mark 0.00 out of

Select the correct statements about interrupt handling in xv6 code

a. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt ✗

1.00

Flag question

a. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt

b. The trapframe pointer in struct proc, points to a location on process's kernel stack ✔

c. On any interrupt/syscall/exception the control first jumps in trapasm.S ✘

d. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt

e. The function trap() is the called only in case of hardware interrupt

f. On any interrupt/syscall/exception the control first jumps in vectors.S

g. The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs ✔

h. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✔

i. xv6 uses the 0x64th entry in IDT for system calls ✘

j. Before going to alltraps, the kernel stack contains upto 5 entries. ✔

k. xv6 uses the 64th entry in IDT for system calls

l. The trapframe pointer in struct proc, points to a location on user stack ✘

m. All the 256 entries in the IDT are filled in xv6 code

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **10**

Incorrect

Mark 0.00 out of 1.00

Flag question

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

a. The stosb() is used here, to fill in some space in memory with zeroes ✔

b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✘

- [ ] c. The readseg finally invokes the disk I/O code using assembly instructions
- [ ] d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [x] e. The kernel file has only two program headers ✔
- [ ] f. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [ ] g. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [x] h. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✖
- [x] i. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✔
- [x] j. The kernel file gets loaded at the Physical address 0x10000 in memory. ✔
- [x] k. The condition if(ph->memsz > ph->filesz) is never true. ✖

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Finish review

← Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state

Jump to...

Random Quiz - 6 (xv6 file system) →

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1 2 3 4 5
6 7 8 9 10
11 12 13 14

Show one page at a time

Finish review

Started on	Thursday, 16 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 16 February 2023, 10:59 PM
Time taken	1 hour 59 mins
Grade	**12.97** out of 15.00 (**86.49**%)

Question **1**
Correct
Mark 1.00 out of 1.00
Flag question

Mark whether the concept is related to scheduling or not.

Yes	No		
✗	✓	file-table	✓
✓	✗	context-switch	✓
✓	✗	runnable process	✓
✓	✗	ready-queue	✓
✓	✗	timer interrupt	✓

file-table: No
context-switch: Yes
runnable process: Yes
ready-queue: Yes
timer interrupt: Yes

Question **2**
Correct
Mark 1.00 out of 1.00
Flag question

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk	✓
Relocation + Limit	one continuous chunk	✓
Segmentation	many continuous chunks of variable size	✓
Segmentation, then paging	many continuous chunks of variable size	✓

Your answer is correct.

The correct answer is: Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size

Question **3**
Partially correct
Mark 0.50 out of 1.00
Flag question

Select all the correct statements about signals

Select one or more:

- [x] a. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✓
- [x] b. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓
- [ ] c. Signal handlers once replaced can't be restored
- [ ] d. The signal handler code runs in user mode of CPU
- [x] e. The signal handler code runs in kernel mode of CPU ✗
- [ ] f. Signals are delivered to a process by another process

f. Signals are delivered to a process by another process

g. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU

h. Signals are delivered to a process by kernel ✔

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **4**

Partially correct

Mark 1.78 out of 2.00

Flag question

Select all the correct statements about the state of a process.

a. A waiting process starts running after the wait is over

b. A process changes from running to ready state on a timer interrupt or any I/O wait

c. A process in ready state is ready to be scheduled ✔

d. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers

e. Changing from running state to waiting state results in "giving up the CPU" ✔

f. A process that is running is not on the ready queue ✔

g. A process can self-terminate only when it's running ✔

h. Typically, it's represented as a number in the PCB ✔

i. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔

j. Processes in the ready queue are in the ready state ✔

k. A running process may terminate, or go to wait or become ready again ✔

l. A process in ready state is ready to receive interrupts

m. A process waiting for any condition is woken up by another process only

n. A process changes from running to ready state on a timer interrupt

Your answer is partially correct.

You have correctly selected 8.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **5**

Partially correct

Mark 0.60 out of 1.00

Flag question

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True **False**

		A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✔
		Only a process in READY state is considered by scheduler	✖
		A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
		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.	✖
		Every forked process has to go through ZOMBIE state, at least for a small duration.	✔

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
Only a process in READY state is considered by scheduler: True
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
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
Every forked process has to go through ZOMBIE state, at least for a small duration.: True

Question **6**
Correct
Mark 1.00 out of 1.00
Flag question

Which of the following statements is false ?

Select one:

- a. Real time systems generally use non preemptive CPU scheduling. ✔
- b. Response time is more predictable in preemptive systems than in non preemptive systems.
- c. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- d. Time sharing systems generally use preemptive CPU scheduling.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

Question **7**
Correct
Mark 1.00 out of 1.00
Flag question

Map each signal with it's meaning

SIGCHLD	Child Stopped or Terminated	✔
SIGALRM	Timer Signal from alarm()	✔
SIGPIPE	Broken Pipe	✔
SIGUSR1	User Defined Signal	✔
SIGSEGV	Invalid Memory Reference	✔

The correct answer is: SIGCHLD → Child Stopped or Terminated, SIGALRM → Timer Signal from alarm(), SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGSEGV → Invalid Memory Reference

Question **8**
Correct
Mark 1.00 out of 1.00
Flag question

Match the names of PCB structures with kernel

linux	struct task_struct	✔
xv6	struct proc	✔

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **9**
Partially correct
Mark 0.43 out of 1.00
Flag question

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
		Response time will be quite poor on non-interruptible kernels	✗
		Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✗
		Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✓
		A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✓
		On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✓ It's the negation of this. Time NOT spent in idle thread.
		xv6 code does not care about Processor Affinity	✗
		Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✗

Response time will be quite poor on non-interruptible kernels: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False
xv6 code does not care about Processor Affinity: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

Question **10**
Correct
Mark 1.00 out of 1.00
Flag question

Select the state that is not possible after the given state, for a process:

New: Running ✓

Ready : Waiting ✓

Running: : None of these ✓

Waiting: Running ✓

Question **11**
Correct
Mark 1.00 out of 1.00
Flag question

Select all the correct statements about zombie processes

Select one or more:

- [x] a. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✓
- [x] b. init() typically keeps calling wait() for zombie processes to get cleaned up ✓
- [ ] c. A zombie process remains zombie forever, as there is no way to clean it up
- [ ] d. Zombie processes are harmless even if OS is up for long time
- [x] e. A zombie process occupies space in OS data structures ✓
- [ ] f. A process becomes zombie when it's parent finishes
- [x] g. If the parent of a process finishes, before the process itself, then after finishing the process is typically ✓

attached to 'init' as parent

- [x] h. A process can become zombie if it finishes, but the parent has finished before it ✔

Your answer is correct.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **12**
Correct
Mark 1.00 out of 1.00
Flag question

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Check the program for syntactical errors
- [ ] b. Process the # directives in a C program
- [x] c. Suggest alternative pieces of code that can be written ✔
- [ ] d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [x] e. Check the program for logical errors ✔
- [ ] f. Convert high level langauge code to machine code

Your answer is correct.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **13**
Partially correct
Mark 0.67 out of 1.00
Flag question

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. expressions
- [ ] b. pointer dereference
- [ ] c. global variables
- [ ] d. function calls
- [x] e. #directives ✔
- [x] f. typedefs ✔
- [ ] g. local variable declaration

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

Question **14**
Correct
Mark 1.00 out of 1.00
Flag question

Order the sequence of events, in scheduling process P1 after process P0

Process P0 is running	1 ✔
Process P1 is running	6 ✔
context of P0 is saved in P0's PCB	3 ✔
Control is passed to P1	5 ✔
context of P1 is loaded from P1's PCB	4 ✔
timer interrupt occurs	2 ✔

Your answer is correct.

The correct answer is: Process P0 is running → 1, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3, Control is passed to P1 → 5, context of P1 is loaded from P1's PCB → 4, timer interrupt occurs → 2

Finish review

◄ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management ►

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

Show one page at a time

Finish review

Started on	Thursday, 2 February 2023, 9:06 PM
State	Finished
Completed on	Thursday, 2 February 2023, 11:00 PM
Time taken	1 hour 53 mins
Grade	**14.06** out of 20.00 (**70.3**%)

Question **1**
Complete
Mark 0.30 out of 1.00
Flag question

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- [ ] a. named pipes are more efficient than ordinary pipes
- [x] b. named pipe exists even if the processes using it do exit()
- [x] c. named pipes can be used between multiple processes but ordinary pipes can not be used
- [x] d. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- [ ] e. named pipe can be used between any processes
- [x] f. a named pipe exists as a file on the file system
- [x] g. ordinary pipe can only be used between related processes

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question **2**
Complete
Mark 1.80 out of 2.00
Flag question

Match the elements of C program to their place in memory

Function code	Code
Local Variables	Stack
Global variables	Data
#define MACROS	No Memory needed
Arguments	Stack
#include files	No memory needed
Local Static variables	Data
Code of main()	Main_Code
Global Static variables	Data
Malloced Memory	Heap

The correct answer is: Function code → Code, Local Variables → Stack, Global variables → Data, #define MACROS → No Memory needed, Arguments → Stack, #include files → No memory needed, Local Static variables → Data, Code of main() → Code, Global Static variables → Data, Malloced Memory → Heap

Question **3**
Complete
Mark 0.14 out of 1.00
Flag question

Order the events that occur on a timer interrupt:

Change to kernel stack of currently running process	2
Select another process for execution	4
Jump to a code pointed by IDT	6
Execute the code of the new process	7
Jump to scheduler code	3
Save the context of the currently running process	1
Set the context of the new process	5

The correct answer is: Change to kernel stack of currently running process → 1, Select another process for execution → 5, Jump to a code pointed by IDT → 2, Execute the code of the new process → 7, Jump to scheduler code → 4, Save the context of the currently running process → 3, Set the context of the new process → 6

Question **4**
Complete
Mark 0.67 out of 1.00
Flag question

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [ ] a. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [x] b. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return
- [x] c.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] d. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] e. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt

Interrupt unblocks P1
Interrupt returns
P3 running
Timer interrupt
Scheduler
P1 running

f. P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **5**
Complete
Mark 1.00 out of 1.00
Flag question

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
  int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

  fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
  fd2 = open("/tmp/2", O_RDDONLY);
  fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
  close(0);
  close(1);
  dup(fd2);
  dup(fd3);
  close(fd3);
  dup2(fd2, fd4);
  printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
  return 0;
}
```

1	/tmp/3
fd2	/tmp/2
fd4	/tmp/2
fd3	closed
2	stderr
fd1	/tmp/1
0	/tmp/2

The correct answer is: 1 → /tmp/3, fd2 → /tmp/2, fd4 → /tmp/2, fd3 → closed, 2 → stderr, fd1 → /tmp/1, 0 → /tmp/2

Question **6**

Complete

Mark 0.30 out of 1.00

Flag question

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

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

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

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

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [ ] a. Only Program 1 is correct
- [ ] b. Both programs are correct
- [ ] c. Both program 1 and 2 are incorrect
- [x] d. Program 2 makes sure that there is one file offset used for '2' and '1'
- [ ] e. Program 2 is correct for > /tmp/ddd but not for 2>&1
- [ ] f. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [x] g. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] h. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] i. Program 2 does 1>&2
- [x] j. Program 1 makes sure that there is one file offset used for '2' and '1'
- [ ] k. Program 1 does 1>&2
- [ ] l. Only Program 2 is correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **7**

Complete

Mark 0.80 out of 1.00

Remove flag

Select the order in which the various stages of a compiler execute.

Linking	4
Loading	4
Syntatical Analysis	2
Intermediate code generation	3
Pre-processing	1

The correct answer is: Linking → 4, Loading → does not exist, Syntatical Analysis → 2, Intermediate code generation → 3, Pre-processing → 1

Question **8**
Complete
Mark 0.80 out of 1.00
Flag question

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A zombie process remains zombie forever, as there is no way to clean it up
- [x] b. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [x] c. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [ ] d. A process can become zombie if it finishes, but the parent has finished before it
- [x] e. A zombie process occupies space in OS data structures
- [ ] f. Zombie processes are harmless even if OS is up for long time
- [ ] g. A process becomes zombie when it's parent finishes
- [x] h. init() typically keeps calling wait() for zombie processes to get cleaned up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **9**
Complete
Mark 0.50 out of 1.00
Flag question

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Relocation + Limit	one continuous chunk
Segmentation	many continuous chunks of variable size
Segmentation, then paging	Many continuous chunks of same size
Paging	Many continuous chunks each of page size

The correct answer is: Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size, Paging → one continuous chunk

Question **10**
Complete
Mark 4.00 out of 5.00
Flag question

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
  int pid1, pid2;
  int pfd[ [2] ][2];

  pipe( pfd[0] );
  pid1 = fork() ;
  if(pid1 != 0) {
    close(pfd[0] [0] );
    close( 1 );
    dup( [0][1] );
    execl("/bin/ls", "/bin/ls", " -l ", NULL);
  }
  pipe( pfd[1] );
  pid2 = fork();
  if(pid2 == 0) {
    close( pfd[0][1] ;
```

```
    close(0);
    dup( pfd[0][0] );
    close(pfd[1] [0] );
    close( 1 );
    dup( pfd[1][1] );
    execl("/usr/bin/head", "/usr/bin/head", " -3 ", NULL);
  } else {
    close(pfd [1][1] );
    close( 0 );
    dup( pfd[1][0] );
    close(pfd [1][1] );
    execl("/usr/bin/tail", "/usr/bin/tail", " -1 ", NULL);
  }
}
```

Question **11**

Complete

Mark 0.75 out of 1.00

Flag question

Consider the image given below, which explains how paging works.

Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

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

The page table is indexed using frame number: False
The page table is itself present in Physical memory: True
Size of page table is always determined by the size of RAM: False
Maximum Size of page table is determined by number of bits used for page number: True
The page table is indexed using page number: True

The physical address may not be of the same size (in bits) as the logical address: True
The PTBR is present in the CPU as a register: True
The locating of the page table using PTBR also involves paging translation: False

Question **12**
Complete
Mark 0.75 out of 1.00
Flag question

Select the state that is not possible after the given state, for a process:

New: Running
Ready : Waiting
Running: : None of these
Waiting: Ready

Question **13**
Complete
Mark 1.00 out of 1.00
Flag question

A process blocks itself means

- a. The kernel code of system call calls scheduler
- b. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The application code calls the scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **14**
Complete
Mark 1.00 out of 1.00
Flag question

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Suggest alternative pieces of code that can be written
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] d. Check the program for syntactical errors
- [x] e. Check the program for logical errors
- [ ] f. Process the # directives in a C program

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **15**
Complete
Mark 0.25 out of 1.00
Flag question

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- [ ] a.
  P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P3 running
  Hardware interrupt
  Interrupt unblocks P1
  Interrupt returns
  P3 running
  Timer interrupt
  Scheduler
  P1 running
- [x] b.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [x] c. P1 running

P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

d. P1 running
keyboard hardware interrupt
keyboard interrupt handler running
interrupt handler returns
P1 running
P1 makes sytem call
system call returns
P1 running
timer interrupt
scheduler
P2 running

e. P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

f. P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Finish review

◄ Random Quiz - 2: bootloader, system calls, fork-exec, open-read-write, linux-basics, processes

Jump to...

Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state ►

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Started on	Monday, 16 January 2023, 9:02 PM
State	Finished
Completed on	Monday, 16 January 2023, 10:08 PM
Time taken	1 hour 6 mins
Grade	**11.13** out of 15.00 (**74.22**%)

Question **1**
Correct
Mark 1.00 out of 1.00
Flag question

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
   execl("/bin/ls", "/bin/ls", NULL);
  printf("hello\n");
}
```

Compiled as

```
cc   exec1.c  -o exec1
cc   exec2.c  -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec1 fails
- b. Execution fails as one exec can't invoke another exec
- c. Program prints hello
- d. "ls" runs on current directory ✔
- e. Execution fails as the call to execl() in exec2 fails

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question **2**
Correct
Mark 1.00 out of 1.00
Flag question

A process blocks itself means

- a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔
- b. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- c. The kernel code of system call calls scheduler
- d. The application code calls the scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **3**

Is the terminal a part of the kernel on GNU/Linux systems?

Correct

Mark 1.00 out of 1.00

Flag question

a. yes

b. no ✔ wrong

The correct answer is: no

Question **4**

Correct

Mark 1.00 out of 1.00

Flag question

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. LILO is a bootloader ✔
- [x] b. Modern Bootloaders often allow configuring the way an OS boots ✔
- [ ] c. Bootloader must be one sector in length
- [x] d. Bootloaders allow selection of OS to boot from ✔
- [ ] e. The bootloader loads the BIOS

Your answer is correct.

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

Question **5**

Correct

Mark 1.00 out of 1.00

Flag question

which of the following is not a difference between real mode and protected mode

a. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit

b. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT

c. processor starts in real mode

d. in real mode the addressable memory is more than in protected mode ✔

e. in real mode the addressable memory is less than in protected mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question **6**

Correct

Mark 2.00 out of 2.00

Flag question

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

a. run ls twice ✔

b. run ls once

c. run ls twice and print hello twice, but output will appear in some random order

d. one process will run ls, another will print hello

e. run ls twice and print hello twice

Your answer is correct.

The correct answer is: run ls twice

Question **7**

Correct

Mark 1.00 out of 1.00

Flag question

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

a. The choice of booting slowly or fast

b. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✔

c. The choice of which OS to boot from

c. The choice of which OS to boot from

d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **8**
Correct
Mark 1.00 out of 1.00
Flag question

Compare multiprogramming with multitasking

a. A multiprogramming system is not necessarily multitasking ✔
b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **9**
Partially correct
Mark 0.67 out of 1.00
Flag question

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Soft link shares the inode of actual file
- [ ] b. Hard links can span across partitions while soft links can't
- [ ] c. Deleting a hard link deletes the file, only if link count was 1
- [x] d. Hard links share the inode ✔
- [x] e. Hard links increase the link count of the actual file inode ✔
- [x] f. Deleting a soft link deletes the link, not the actual file ✔
- [ ] g. Deleting a hard link always deletes the file
- [x] h. Soft links can span across partitions while hard links can't ✔
- [ ] i. Deleting a soft link deletes both the link and the actual file
- [ ] j. Hard links enforce separation of filename from it's metadata in on-disk data structures.
- [ ] k. Deleting a soft link deletes only the actual file
- [ ] l. Soft links increase the link count of the actual file inode

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **10**
Partially correct
Mark 0.80 out of 1.00
Flag question

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✔
- [x] b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✔
- [ ] c. The two modes are essential for a multiprogramming system
- [x] d. The two modes are essential for a multitasking system ✔
- [x] e. There is an instruction like 'iret' to return from kernel mode to user mode ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **11**
Incorrect
Mark 0.00 out of 3.00
Flag question

Select correct statements about mounting

Select one or more:

- a. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system.
- b. It's possible to mount a partition on one computer, into namespace of another computer.
- c. On Linuxes mounting can be done only while booting the OS
- d. Even in operating systems with a pluggable kernel module for file systems, the code for mounting any particular file system must be already present in the operating system system kernel ✗
- e. Mounting deletes all data at the mount-point
- f. The mount point can be a file as well ✗
- g. Mounting makes all disk partitions available as one name space
- h. The mount point must be a directory
- i. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✓
- j. The existing name-space at the mount-point is no longer visible after mounting

Your answer is incorrect.

The correct answers are: Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space, The mount point must be a directory, The existing name-space at the mount-point is no longer visible after mounting, Mounting makes all disk partitions available as one name space, In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system., It's possible to mount a partition on one computer, into namespace of another computer.

Question **12**
Partially correct
Mark 0.67 out of 1.00
Flag question

Order the following events in boot process (from 1 onwards)

Shell	5	✗
OS	3	✓
Init	4	✓
Login interface	6	✗
Boot loader	2	✓
BIOS	1	✓

Your answer is partially correct.

You have correctly selected 4.
The correct answer is: Shell → 6, OS → 3, Init → 4, Login interface → 5, Boot loader → 2, BIOS → 1

Finish review

← Random Quiz - 1 (Pre-Requisite Quiz)

Jump to...

Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes →

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Started on	Friday, 17 March 2023, 2:29 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:56 PM
Time taken	2 hours 27 mins
Grade	**5.19** out of 10.00 (**51.91**%)

Question **1**
Partially correct
Mark 0.42 out of 0.50
Flag question

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False			
✓	✗	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✓	
✗	✓	free page list is a singly circular linked list.	✓	it's singly linked NULL terminated list.

✓	✗	if(kmem.use_lock)
		acquire(&kmem.lock);
		is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
			✓	

✗	✗	the kmem.lock is used by kfree() and kalloc() only.	✗	

✗	✓	if(kmem.use_lock)
		acquire(&kmem.lock);
		this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
			✓	No. kinit2() calls kfree() and then initializes use_lock.

✓	✗	The pointers that link the pages together are in the first 4 bytes of the pages themselves	✓	

kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True
free page list is a singly circular linked list.: False
if(kmem.use_lock)
acquire(&kmem.lock);
is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True
the kmem.lock is used by kfree() and kalloc() only.: True
if(kmem.use_lock)
acquire(&kmem.lock);
this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False
The pointers that link the pages together are in the first 4 bytes of the pages themselves: True

Question **2**
Correct
Mark 0.25 out of 0.25
Flag question

Match function with it's meaning

ideintr	disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer ✓
idestart	tell disc controller to start I/O for the first buffer on idequeue ✓
ideinit	Initialize the disc controller ✓
idewait	Wait for disc controller to be ready ✓
iderw	Issue a disk read/write for a buffer, block the issuing process ✓

Your answer is correct.

The correct answer is: ideintr → disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer, idestart → tell disc controller to start I/O for the first buffer on idequeue, ideinit → Initialize the disc controller, idewait → Wait for disc controller to be ready, iderw → Issue a disk read/write for a buffer, block the issuing process

Question **3**
Partially correct
Mark 0.23 out of 0.50
Flag question

Which of the following is DONE by allocproc() ?

- [x] a. ensure that the process starts in forkret() ✔
- [x] b. setup the contents of the trapframe of the process properly ✘
- [x] c. allocate kernel stack for the process ✔
- [x] d. allocate PID to the process ✔
- [ ] e. setup kernel memory mappings for the process
- [ ] f. ensure that the process starts in trapret()
- [ ] g. setup the trapframe and context pointers appropriately
- [x] h. Select an UNUSED struct proc for use ✔

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **4**
Incorrect
Mark 0.00 out of 0.50
Flag question

The first instruction that runs when you do "make qemu" is

```
cli
```

from bootasm.S

Why?

- a. "cli" that is Command Line Interface needs to be enabled first
- b. "cli" enables interrupts, it is required because the kernel must handle interrupts.
- c. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available ✘
- d. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- e. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- f. "cli" enables interrupts, it is required because the kernel supports interrupts.
- g. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.
- h. "cli" stands for clear screen and the screen should be cleared before OS boots.

Your answer is incorrect.

The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

Question **5**
Incorrect
Mark 0.00 out of 0.25
Flag question

Which of the following call sequence is impossible in xv6?

- a. Process 1: write() -> sys_write()-> file_write() -- timer interrupt -> trap() -> yield() -> sched() -> switch() (jumps to)-> scheduler() ->swtch()(jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code ✘
- b. Process 1: write() -> sys_write()-> file_write() -> writei() -> bread() -> bget() -> iderw() -> sleep() -> sched() -> switch() (jumps to)-> scheduler() ->swtch()(jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code
- c. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...

Your answer is incorrect.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...

Question **6**
Partially correct
Mark 0.60 out of 1.00
Flag question

Mark the statements as True/False w.r.t. swtch()

True	False			
☑	✘	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.	✔	
☑	✘	swtch() changes the context from "old" to "new"	✔	yeah, that's the definition
✘	☑	movl %esp, (%eax)	✔	No. it means c->scheduler = contents of

	means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()		esp.
	swtch() is called only from sched() or scheduler()	✔	
	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.	✔	any function can push anything on stack, but remove it properly, that will not affect calling convention.
	switch stores the old context on new stack, and restores new context from old stack.	✖	old goes on old, new comes from new stack
	p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch()	✔	That's the only place when p->context is changed.
	push in swtch() happens on old stack, while pop happens from new stack	✖	
	sched() is the only place when p->context is set	✖	
	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	✖	it does reverse!

swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True
swtch() changes the context from "old" to "new": True
movl %esp, (%eax)
means, *(c->scheduler) = contents of esp
When swtch() is called from scheduler(): False
swtch() is called only from sched() or scheduler(): True
swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False
switch stores the old context on new stack, and restores new context from old stack.: False
p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch(): True
push in swtch() happens on old stack, while pop happens from new stack: True
sched() is the only place when p->context is set: False
swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False

Question **7**
Partially correct
Mark 0.40 out of 0.50
Flag question

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode
entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain().
- [ ] b. The kernel.asm file is the final kernel file
- [x] c. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✔
- [ ] d. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is

all zeroes.

- [x] e. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [x] f. readseg() reads first 4k bytes of kernel in memory ✔
- [ ] g. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- [ ] h. The bootmain() code does not read the kernel completely in memory
- [x] i. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **8**

Correct

Mark 0.50 out of 0.50

Flag question

With respect to this diagram, mark statements as True/False.

True	False			
◉	○	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	✔	
◉	○	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	✔	
◉	○	The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	✔	
◉	○	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	✔	
◉	○	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	✔	
◉	○	"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.	✔	"Kernel data + memory" on LEFT side, here refers to the virtual addresses of kernel used at run time.

✔	✘	The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable	✔

When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: True
PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: True
The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.: True
The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM: True
This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True
"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True
The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True

Question **9**
Correct
Mark 0.25 out of 0.25
Flag question

The variable 'end' used as argument to kinit1 has the value

- a. 81000000
- b. 8010a48c
- c. 80110000
- d. 80000000
- e. 801154a8 ✔
- f. 80102da0

The correct answer is: 801154a8

Question **10**
Partially correct
Mark 0.31 out of 0.50
Flag question

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False		
✘	✔	The value of 'chan' is changed only in sleep()	✘
✔	✘	'chan' is used only by the sleep() and wakeup1() functions.	✔
✔	✘	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✔
✔	✘	When chan is not NULL, the 'state' in struct proc must be SLEPING	✔
✔	✘	chan stores the address of the variable, representing a condition, for which the process is waiting.	✔
✘	✔	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✘
✘	✔	Changing the state of a process automatically changes the value of 'chan'	✘
✘	✔	when chan is NULL, the 'state' in proc must be RUNNABLE.	✔

The value of 'chan' is changed only in sleep(): True
'chan' is used only by the sleep() and wakeup1() functions.: True
in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True
When chan is not NULL, the 'state' in struct proc must be SLEPING: True
chan stores the address of the variable, representing a condition, for which the process is waiting.: True
chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False
Changing the state of a process automatically changes the value of 'chan': False
when chan is NULL, the 'state' in proc must be RUNNABLE.: False

Question **11**
Incorrect

Why is there a call to kinit2? Why is it not merged with knit1?

Mark 0.00 out of 0.25

Flag question

a. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames

b. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

c. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2() ✗

d. Because there is a limit on the values that the argumets to knit1() can take.

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **12**

Partially correct

Mark 0.13 out of 0.50

Flag question

when is each of the following stacks allocated?

kernel stack for the scheduler, on other processors	during main()->kinit2()	✗
kernel stack of process	during fork() in allocproc()	✓
user stack of process	during exec()	✗
kernel stack for scheduler, on first processor	during main()->kinit1()	✗

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: kernel stack for the scheduler, on other processors → in main()->startothers(), kernel stack of process → during fork() in allocproc(), user stack of process → during fork() in copyuvm(), kernel stack for scheduler, on first processor → in entry.S

Question **13**

Partially correct

Mark 0.45 out of 1.00

Flag question

Given below is code of sleeplock in xv6.

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

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


void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

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

Mark the statements as True/False w.r.t. this code.

True	False			
✗	✓	A process has acquired the sleeplock when it comes out of sleep()	✓	
✓	✗	The spinlock lk->lk is held when the process comes out of sleep()	✗	
✗	✓	Wakeup() will wakeup the first process waiting for the lock	✓	Wakeup() will wakeup all processes waiting for the lock
✓	✗	The process which called acquiresleep() and then got blocked, is woken up by the	✗	it's woken up by another process which called releasesleep() and then wakeup()

		timer interrupt	
✗	✓	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep()	✓
✓	✗	sleep() is the function which blocks a process.	✗

✗	✓	acquire(&lk->lk);
		while (lk->locked) {
		sleep(lk, &lk->lk);
		}
		could also be written as
		acquire(&lk->lk);
		if (lk->locked) {
		sleep(lk, &lk->lk);
		}
			✓ loop is required because other process might have obtained the lock before this process returns from sleep().

✓	✗	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✗
✓	✗	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up	✗ wakeup() wakes up all processes, and they "thunder" to take the spinlock.
✗	✓	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✓
✓	✗	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✗

A process has acquired the sleeplock when it comes out of sleep(): False
The spinlock lk->lk is held when the process comes out of sleep(): True
Wakeup() will wakeup the first process waiting for the lock: False
The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True
sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep(): False
sleep() is the function which blocks a process.: True
acquire(&lk->lk);
while (lk->locked) {
sleep(lk, &lk->lk);
}
could also be written as
acquire(&lk->lk);
if (lk->locked) {
sleep(lk, &lk->lk);
}: False
Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True
All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True
the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False
the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True

Question **14**
Incorrect
Mark 0.00 out of 0.25
Flag question

Select the odd one out

- a. Process stack of running process to kernel stack of running process
- b. Kernel stack of scheduler to kernel stack of new process
- c. Kernel stack of running process to kernel stack of scheduler ✗
- d. Kernel stack of new process to kernel stack of scheduler
- e. Kernel stack of new process to Process stack of new process

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **15**
Correct
Mark 0.50 out of 0.50
Flag question

We often use terms like "swtch() changes stack from process's kernel stack to scheduler's stack", or "the values are pushed on stack", or "the stack is initialized to the new page", etc. while discussing xv6 on x86.

Which of the following most accurately describes the meaning of "stack" in such sentences?

- a. The "stack" variable declared in "stack.S" in xv6
- b. The ss:esp pair ✓

c. The region of memory where the kernel remembers all the function calls made

d. the region of memory which is currently used as stack by processor

e. The stack variable used in the program being discussed

f. The stack segment

g. The region of memory allocated by kernel for storing the parameters of functions

Your answer is correct.

The correct answer is: The ss:esp pair

Question **16**

Incorrect

Mark 0.00 out of 0.25

Flag question

Which of the following is not a task of the code of swtch() function

a. Load the new context

b. Save the return value of the old context code

c. Jump to next context EIP ✗

d. Switch stacks

e. Change the kernel stack location

f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **17**

Correct

Mark 0.50 out of 0.50

Flag question

The struct buf has a sleeplock, and not a spinlock, because

a. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available.

b. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf. ✔

c. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context

d. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock

e. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.

Your answer is correct.

The correct answer is: struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question **18**

Partially correct

Mark 0.17 out of 0.50

Flag question

Mark statements as True/False, w.r.t. the given diagram

True	False		
✗	☑	The "context" yellow coloured box,	✗

		pointed to by cpu->scheduler is on the kernel stack of the scheduler.		
☑	✗	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	✔	
✗	☑	The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate	✗	diagram shows only kernel stack
✗	☑	This is a diagram of swtch() called from scheduler()	✗	No. diagram of swtch() called from sched()
☑	✗	The diagram is correct	✗	
☑	✗	The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.	✔	

The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False
The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True
The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False
This is a diagram of swtch() called from scheduler(): False
The diagram is correct: True
The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True

Question **19**
Partially correct
Mark 0.38 out of 0.75
Flag question

Mark statements as True/False w.r.t. ptable.lock

True	False			
☑	✗	ptable.lock protects the proc[] array and all struct proc in the array	✔	
☑	✗	It is taken by one process but released by another process, running on same processor	✔	
☑	✗	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6	✗	sched() is called only if you hold ptable.lock
☑	✗	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)	✔	One process slept, another was scheduled and it came out of timer interrupt.
✗	☑	A process can sleep on ptable.lock if it can't aquire it.	✗	It's a spinlock!
✗	☑	ptable.lock is acquired but never released	✔	how is that possible?
✗	☑	The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched()	✗	No. it's always held. sched() will hold the lock.
✗	☑	ptable.lock can be held by different processes on different processors at the same time	✗	No lock can be held like this!

ptable.lock protects the proc[] array and all struct proc in the array: True
It is taken by one process but released by another process, running on same processor: True
the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True
One sequence of function calls which takes and releases the ptable.lock is this:
iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock): True
A process can sleep on ptable.lock if it can't aquire it.: False
ptable.lock is acquired but never released: False
The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
ptable.lock can be held by different processes on different processors at the same time: False

Question **20**

Partially correct

Mark 0.11 out of 0.75

Flag question

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

movw %ax, %gs	protected mode with only segmentation	✔
inb $0x64,%al	protected mode with segmentation and 4 MB pages	✖
jmp *%eax	real mode	✖
movl $(V2P_WO(entrypgdir)), %eax	real mode	✖
readseg((uchar*)elf, 4096, 0);	protected mode with segmentation and 4 MB pages	✖
orl $CR0_PE, %eax	protected mode with only segmentation	✖
ljmp $(SEG_KCODE<<3), $start32	protected mode with segmentation and 4 MB pages	✖

The correct answer is: movw %ax, %gs → protected mode with only segmentation, inb $0x64,%al → real mode, jmp *%eax → protected mode with segmentation and 4 MB pages, movl $(V2P_WO(entrypgdir)), %eax → protected mode with only segmentation, readseg((uchar*)elf, 4096, 0); → protected mode with only segmentation, orl $CR0_PE, %eax → real mode, ljmp $(SEG_KCODE<<3), $start32 → real mode

Finish review

◄ Quiz-1(24 Feb 2023)

Jump to...

Pre-requisite Quiz (old) - use it for practice ►

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Operating Systems-even-sem-22-23 ऑपरेटींग सिस्टीम्स

Quiz navigation

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Show one page at a time

Finish review

Started on	Friday, 24 February 2023, 2:45 PM
State	Finished
Completed on	Friday, 24 February 2023, 5:07 PM
Time taken	2 hours 22 mins
Grade	**7.63** out of 10.00 (**76.27**%)

Question **1**

Incorrect

Mark 0.00 out of 0.50

Flag question

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

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

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

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

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- [ ] a. Program 2 does 1>&2
- [x] b. Program 2 is correct for > /tmp/ddd but not for 2>&1 ✗
- [x] c. Program 1 does 1>&2 ✗
- [x] d. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- [ ] e. Both programs are correct
- [ ] f. Only Program 1 is correct
- [ ] g. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] h. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [ ] i. Only Program 2 is correct
- [ ] j. Both program 1 and 2 are incorrect
- [ ] k. Program 1 makes sure that there is one file offset used for '2' and '1'
- [x] l. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **2**
Correct
Mark 0.50 out of 0.50
Flag question

Which of the following parts of a C program do not have any corresponding machine code ?

- [x] a. global variables ✔
- [ ] b. local variable declaration
- [ ] c. pointer dereference
- [ ] d. expressions
- [ ] e. function calls
- [x] f. #directives ✔
- [x] g. typedefs ✔

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **3**
Correct
Mark 0.50 out of 0.50
Flag question

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- (●) a. It prohibits a user mode process from running privileged instructions ✔
- ( ) b. It disallows hardware interrupts when a process is running
- ( ) c. It prohibits one process from accessing other process's memory
- ( ) d. It prohibits invocation of kernel code completely, if a user program is running

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **4**
Partially correct
Mark 0.44 out of 0.50
Flag question

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Static variables	Immediately after the text, along with globals	✔
Global variables	Immediately after the text	✔
#define	No memory allocated, they are handled by pre-processor	✔
#include files	No memory allocated, they are handled by linker	✘
Text	starting with 0	✔
Local variables	An offset with respect to stack pointer (esp)	✔
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✔
typedef	No memory allocated, as they are not variables, but only conceptual definition of a type	✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Static variables → Immediately after the text, along with globals, Global variables → Immediately after the text, #define → No memory allocated, they are handled by pre-processor, #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, Text → starting with 0, Local variables → An offset with respect to stack pointer (esp), malloced memory → Heap (handled by the malloc-free library, using OS's system calls), typedef → No memory allocated, as they are not variables, but only conceptual definition of a type

Question **5**

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and

Correct

Mark 0.50 out of 0.50

Flag question

argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

Figure 3.2 Diagram of process state.

True	False		
☑	☒	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✔
☒	☑	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.	✔
☑	☒	Only a process in READY state is considered by scheduler	✔
☑	☒	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
☒	☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✔

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
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
Only a process in READY state is considered by scheduler: True
Every forked process has to go through ZOMBIE state, at least for a small duration.: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

Question **6**

Correct

Mark 0.50 out of 0.50

Flag question

Map each signal with it's meaning

SIGALRM	Timer Signal from alarm()	✔
SIGUSR1	User Defined Signal	✔
SIGCHLD	Child Stopped or Terminated	✔
SIGSEGV	Invalid Memory Reference	✔
SIGPIPE	Broken Pipe	✔

The correct answer is: SIGALRM → Timer Signal from alarm(), SIGUSR1 → User Defined Signal, SIGCHLD → Child Stopped or Terminated, SIGSEGV → Invalid Memory Reference, SIGPIPE → Broken Pipe

Question **7**

Correct

Mark 0.50 out of

Which of the following instructions should be privileged?

0.50

Flag question

Select one or more:

- [x] a. Switch from user to kernel mode. ✔ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- [ ] b. Set value of a memory location
- [ ] c. Read the clock.
- [x] d. Access memory management unit of the processor ✔
- [x] e. Modify entries in device-status table ✔
- [x] f. Turn off interrupts. ✔
- [ ] g. Access a general purpose register
- [x] h. Access I/O device. ✔
- [x] i. Set value of timer. ✔

Your answer is correct.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **8**

Partially correct

Mark 0.30 out of 0.50

Flag question

Mark the statements about named and un-named pipes as True or False

True	False		
✖	✔	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✖
✖	✔	The pipe() system call can be used to create either a named or un-named pipe.	✖
✔	✖	Both types of pipes provide FIFO communication.	✔
✖	✔	Named pipes can be used for communication between only "related" processes.	✔
✔	✖	Un-named pipes are inherited by a child process from parent.	✖
✔	✖	Named pipes can exist beyond the life-time of processes using them.	✔
✔	✖	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✔
✔	✖	Both types of pipes are an extension of the idea of "message passing".	✔
✔	✖	Named pipe exists as a file	✔
✖	✔	A named pipe has a name decided by the kernel.	✖

The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
The pipe() system call can be used to create either a named or un-named pipe.: False
Both types of pipes provide FIFO communication.: True
Named pipes can be used for communication between only "related" processes.: False
Un-named pipes are inherited by a child process from parent.: True
Named pipes can exist beyond the life-time of processes using them.: True
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Both types of pipes are an extension of the idea of "message passing".: True
Named pipe exists as a file: True

Named pipe exists as a file: True
A named pipe has a name decided by the kernel.: False

Question **9**
Partially correct
Mark 0.36 out of 0.50
Flag question

You must have seen the error message "Segmentation fault, core dumped" very often.

With respect to this error message, mark the statements as True/False.

True	False			
☑	✗	The image of the process is stored in a file called "core", if the ulimit allows so.	✔	see ulimit -a
✗	☑	The illegal memory access was detected by the kernel and the process was punished by kernel.	✔	"detection" is done by CPU, not kernel.
☑	✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.	✖	
☑	✗	The process has definitely performed illegal memory access.	✖	
✗	☑	On Linux, the message is printed only because the memory management scheme is segmentation	✔	No, it's just a term used, even if paging is used for memory management.
☑	✗	The core file can be analysed later using a debugger, to determine what went wrong.	✔	use gdb ./core ./executable-filename
✗	☑	The term "core" refers to the core code of the kernel.	✔	core means memory, all memory for the process.

The image of the process is stored in a file called "core", if the ulimit allows so.: True
The illegal memory access was detected by the kernel and the process was punished by kernel.: False
On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.: True
The process has definitely performed illegal memory access.: True
On Linux, the message is printed only because the memory management scheme is segmentation: False
The core file can be analysed later using a debugger, to determine what went wrong.: True
The term "core" refers to the core code of the kernel.: False

Question **10**
Correct
Mark 0.50 out of 0.50
Flag question

What is meant by formatting a disk/partition?

- a. creating layout of empty directory tree/graph data structure ✔
- b. writing zeroes on all sectors
- c. erasing all data on the disk/partition
- d. storing all the necessary programs on the disk/partition

The correct answer is: creating layout of empty directory tree/graph data structure

Question **11**
Correct
Mark 0.50 out of 0.50
Flag question

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging	one continuous chunk	✔
Relocation + Limit	one continuous chunk	✔
Segmentation	many continuous chunks of variable size	✔
Segmentation, then paging	many continuous chunks of variable size	✔

Your answer is correct.

The correct answer is: Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size

Question **12**
Incorrect
Mark 0.00 out of 0.50
Flag question

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

```
good

output
```

should be written as `good output`

--

```
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one ✗

The correct answer is: hi one one

Question **13**
Partially correct
Mark 0.45 out of 0.50
Flag question

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No		
☑	✗	PID	✓
☑	✗	Memory management information about that process	✓
☑	✗	Pointer to the parent process	✓
✗	☑	Pointer to IDT	✓
☑	✗	List of opened files	✓
✗	☑	PID of Init	✓
☑	✗	EIP at the time of context switch	✗
✗	☑	Function pointers to all system calls	✓
☑	✗	Process state	✓
☑	✗	Process context	✓

PID: Yes
Memory management information about that process: Yes

Pointer to the parent process: Yes
Pointer to IDT: No
List of opened files: Yes
PID of Init: No
EIP at the time of context switch: Yes
Function pointers to all system calls: No
Process state: Yes
Process context: Yes

Question **14**
Incorrect
Mark 0.00 out of 0.50
Flag question

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 5 ✗ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **15**
Correct
Mark 0.50 out of 0.50
Flag question

Match the elements of C program to their place in memory

Global variables	Data	✓
Function code	Code	✓
Code of main()	Code	✓
#include files	No memory needed	✓
Arguments	Stack	✓
Local Static variables	Data	✓
Local Variables	Stack	✓
Malloced Memory	Heap	✓
Global Static variables	Data	✓
#define MACROS	No Memory needed	✓

The correct answer is: Global variables → Data, Function code → Code, Code of main() → Code, #include files → No memory needed, Arguments → Stack, Local Static variables → Data, Local Variables → Stack, Malloced Memory → Heap, Global Static variables → Data, #define MACROS → No Memory needed

Question **16**
Correct
Mark 0.50 out of 0.50
Flag question

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

True	False		
☑	✗	Device driver is part of OS code	✓
✗	☑	Device driver is part of hardware	✓
☑	✗	Device driver is an intermediary between the hardware controller and OS	✓
✗	☑	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.	✓
☑	✗	It's possible that a particular hardware has multiple device drivers available for it.	✓
☑	✗	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✓

Device driver is part of OS code: True
Device driver is part of hardware: False
Device driver is an intermediary between the hardware controller and OS: True
Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False
It's possible that a particular hardware has multiple device drivers available for it.: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Question **17**
Correct
Mark 0.50 out of 0.50
Flag question

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 ✔
Indexed Allocation	Overhead of reading metadata blocks ✔
Linked allocation	Too many seeks ✔

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Indexed Allocation → Overhead of reading metadata blocks, Linked allocation → Too many seeks

Question **18**
Partially correct
Mark 0.48 out of 0.50
Flag question

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
  int pid1, pid2;
  int pfd[ 2 ✔ ][2];

  pipe( pfd[0] ✔ );
  pid1 = fork() ✔ ;
  if(pid1 != 0) {
    close(pfd[0] [0] ✔ );
    close( 1 ✔ );
    dup( pfd[0][1] ✔ );
    execl("/bin/ls", "/bin/ls", " -l ✔ ", NULL);
  }
  pipe( pfd[1] ✔ );
  pid2 ✔ = fork();
  if(pid2 == 0) {
    close( pfd[0][1]) ✔ ;
    close(0);
    dup( pfd[0][0] ✔ );
    close(pfd[1] [0] ✔ );
    close( 1 ✔ );
    dup( pfd[1][1] ✔ );
    execl("/usr/bin/head", "/usr/bin/head", " -3 ✔ ", NULL);
  } else {
    close(pfd [1][1] ✔ );
    close( 0 ✔ );
    dup( pfd[1][0] ✔ );
    close(pfd [1][1] ✖ );
```

```
    execl("/usr/bin/tail", "/usr/bin/tail", " -1 ", NULL);
  }
}
```

Question **19**
Partially correct
Mark 0.36 out of 0.50
Flag question

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
☑	☒	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✔
☑	☒	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✔
☒	☑	The kernel refers to the page table for converting each virtual address to physical address.	✖
☒	☑	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✔
☑	☒	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✔
☒	☑	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✔
☒	☑	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✖

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The kernel refers to the page table for converting each virtual address to physical address.: False
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

Question **20**
Partially correct
Mark 0.25 out of 0.50
Flag question

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.

An option has to be correct entirely to be marked "Yes"

Possibly one block bitmap corresponding to the parent directory — Yes ✔

Superblock — Yes ✔

Data blocks of the file	Yes	✗
One or more data bitmap blocks for the parent directory	Yes	✗
One or multiple data blocks of the parent directory	No	✓
Block bitmap(s) for all the blocks of the file	No	✗

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 3.

The correct answer is: Possibly one block bitmap corresponding to the parent directory → Yes, Superblock → Yes, Data blocks of the file → No, One or more data bitmap blocks for the parent directory → No, One or multiple data blocks of the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes

Finish review

You are logged in as 112003093 Div2 Chinmay Sandeep Naik (Log out)
OS-even-sem-22-23
Data retention summary
Get the mobile app
Give feedback about this software
(C) COEP Technological University. All Rights Reserved.

Chapter-I and II
Introduction to OS

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

OS or kernel?

- Debate on terminology
- What you use in daily life is not kernel, but
  - GUI, Shell, Applications, ...
  - One view: OS = kernel + (GUI, Shell, Libraries, System programs, Minimum Applications,...)
  - Correct, IMO!
- What we study in this course is kernel
  - Not how is “Ubuntu” built!

Building an "OS"

- E.g. Debian
  - A collection of thousands of applications, libraries, system programs, ... and Linux kernel !
  - Linux kernel is at the heart, but heart is not a human without the body !
  - Job of "Debian Developers"
    - Collect the source code of all things you want
    - Create an "Environment" for compiling things : bootstrap challenge
    - Compile everything
    - Ensure that things work "with each other"
      - Very difficult! Versions, dependencies !
    - "Package" things (e.g. make .deb files)

Components of a computer system Abstract view

Figure 1.1 Abstract view of the components of a computer system.

Today, different type of kernels to serve different needs

- Each "Computing Environment" has a different requirement of managing resources, processes, etc
  - Desktop / Laptop
  - Thin Clients
  - Workstations
  - Handheld
  - "Servers"
  - Distributed systems
  - Peer to peer systems
  - Virtualization
  - Cloud
  - Real time

Von Neumann architecture

A von Neumann architecture

A key to “understanding”

- Everything happens on processor
- Processor is always running some instruction
- We should be able to tell possible execution sequences on processor

Memory hierarchy

Memory hierarchy

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

System calls

- We have seen
  - Fork(), exec(): system calls to create processes
  - Open(), read(), write() : system calls to play with files
- System call: A service given by kernel

System calls: the problem

- On a time shared system
  - Applications and kernel run on the CPU taking turns
  - Kernel allocates time to application , by setting a value in timer register, and then makes application run
    - Timer ticks down every clock cycle
  - *Timer hardware interrupt* occurs when timer = 0 , that is time of application is over
  - Kernel runs again

System calls: the problem

- Kernel must provide “services” to applications
  - Create processes
  - Grant access to resources to processes
  - Etc
- Process should be able to “call” kernel’s service
- But everything runs on processor!
  - That is both processes and kernel
- Kernel should be able to run particular instructions which processes should not be able to !

System calls: the problem

- Privileged Instructions
  - Set the timer register
  - Set a value in registers of I/O devices
  - Set the MMU
  - Etc
- How to ensure that only kernel code can run privilged instructions?
  - Not possible without hardware support in CPU!

System calls: the problem

- CPU: 2 modes
  - User mode and kernel mode
  - All instructions can be executed in kernel mode
  - Non-privileged instructions can be executed in User mode
  - How to ensure this?
- Special instruction: software interrupt instruction
  - Changes the mode from User Mode to Kernel mode and changes the CS+IP (or PC) to a pre-defined address (where the kernel already exists)
  - Two things at a time – that's the key

2 modes of operation of CPU needed for multi-tasking system

Application – kernel interaction in system calls

System call – parameter passing

Sharing of data between “user mode” and “kernel mode” code !

What a modern multi-processor, multi-tasking kernel does

- **Process Management**
  - Create an environment (fork, exec, exit, etc) so that processes can be created, exited, ...
- **Resource management**
  - Processes can access "resources" only through kernel
  - System calls like: open(), brk(), ...
  - Includes storage management – enable a "tree" type view of files and folders
  - Manage memory for itself and all processes
  - I/O Management – make I/o devices accessible to processes and itself
  - Manage users (uid, gid, setuid(), seteuid(), ..)
- **Protection and Security**
  - Limit what a process can do
  - Prevent "attacks" by internal and external agents

System calls exist for ...

- Process control
  - create process (fork), terminate process (exit)
  - end, abort (kill)
  - load, execute (exec)
  - get process attributes (getpid, ..), set process attributes (setrlimit, ...)
  - wait for time (sleep, ...)
  - wait event (wait, ...), signal event (signal, kill, ...)
  - allocate and free memory (brk, ...)
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

System calls exist for ...

- File management
  - create file (open, create), delete file (unlink)
  - open, close (close()) file
  - read, write, reposition (read, write, lseek, ...)
  - get and set file attributes (chown, chmod, stat, ...)
- Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
  - On Linux: mostly done using file systecalls, but special calls are also there: mknod, mkfifo, ...

System calls exist for ...

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
  - time, fcntl, ...
- Communications
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
  - From client to server
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
  - Examples: socket, bind, listen, shmget, send, recv,

System calls exist for ...

- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access
- **More**
  - On LinuxL “man syscalls”
  - POSIX: Portable Operating System Interface
  - See output of strace

Traditional Unix system structure

Linux system structure

- **Linux kernel is extendible**
  - **Kernel modules**
  - **lsmod, insmod, ..**

Layered approach

- **xv6**

Microkernel approach

- Mach, Mac OS X (based on Mach)
- GNU herd
- Against “monolithic” approach (Linux kerenel)

Skipped

- **IOS, Android structure**

Clustered sytem

Free Software / Open Source kernels

- **Linux**
- **BSD Unix**
- **GNU Herd**
- **xv6**
- **Minix**
- **FreeRTOS**
- **OpenSolaris**
- **Many others**

https://en.wikipedia.org/wiki/Comparison_of_open-source_operating_systems

Virtualization

- A general concept!
- Create a “virtual reality” out of “reality”
  - Black boxes that do something
  - End users need not know “how”, but only “what”
- Following all are “virtualizations”
  - Function
  - Class
  - An application
  - Debian GNU/Linux (or any OS)

Virtual Machines, an example

Introduction to Linux and Basics of System Calls

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

Why GNU/Linux ?

Few Key Concepts

- *Files* don't open themselves
  - Always some application/program open()s a file.
- Files don't display themselves
  - A file is displayed by the program which opens it. Each program has it's own way of handling ifles

Few Key Concepts

- **Programs don't run themselves**
  - **You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program**
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - **and programs open files**

Path names

- Tree like directory structure
- Root directory called /
- A complete path name for a file
  - /home/student/a.c
- Relative path names

  concept: every running program has a *current* working directory

  . current directory

  .. parent directory

  ./Desktop/xyz/../p.c

A command

- Name of an executable file
  - For example: 'ls' is actually “/bin/ls”
- Command takes arguments
  - E.g. ls /tmp/
- Command takes options
  - E.g. ls -a

A command

- Command can take both arguments and options
  - E.g. ls -a /tmp/
- Options and arguments are basically argv[] of the main() of that program

Basic Navigation Commands

- pwd
- ls
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a

    \ls -F
- cd
  - cd /tmp/
  - cd
  - cd /home/student/Desktop
- notation: ~
  - cd ~
  - cd ~/Desktop
  - ls ~/Desktop

Map these commands to navigation using a graphical file browser

Before the command line, the concept of Shell and System calls

- System Call
  - A function from OS code
  - Does specific operations with hardware (e.g. reading from keyboard, writing to screen, *opening* a file from disk, etc.)
  - Applications can't access hardware directly, they have to request the OS using system calls
  - Examples
    - open(*"/x/y", ..)*
    - *read(fd, &a, ...);*
    - *fork()*
    - *exec("/usr/bin/ls" ...)*

The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the *Applications***
- **Talks with Users and Applications and does some talk with OS**

Not a very accurate diagram !

http://code.snipcademy.com/tutorials/linux-command-line/environment/

The Shell

Shell waits for user's input

Requests the OS to run a program which the user has asked to run

Again waits for user's input

GUI is a Shell !

Let's Understand fork() and exec()

```
#include <unistd.h>
int main() {
        fork();
        printf("hi\n");
        return 0;
}
```

```
#include <unistd.h>
int main() {
        printf("hi\n");
        execl("/bin/ls", "ls",
NULL);
        printf("bye\n");
        return 0;
}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
        char string[128];
        int pid;
        while(1) {
                printf("prompt>");
                scanf("%s", string);
                pid = fork();
                if(pid == 0) {
                        execl(string, string, NULL);
                } else {
                        wait(0);
                }
        }
```

Users on Linux

- Root and others
  - root
    - superuser, can do (almost) everything
    - Uid = 0
  - Other users
    - Uid != 0
  - UID, GID understood by kernel
- Groups
  - Set of users is a group
  - Any number of groups is possible
- users/groups data in Text files: /*etc*/passwd /*etc*/shadow /*etc*/group ...

File Permissions on Linux

- 3 sets of 3 permission
  - Octal notation: Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- chmod command, used to change permissions, uses these notations
  - It calls the chmod() system call
- Permissions are for processes started by the user, but in common language often we say "permissions are for the user"

File Permissions on Linux

```
-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01_linux_basics.ppt
-rw-r--r-- 1 abhijit abhijit  341736 May 17 10:39 Debian Family Tree.svg
drwxr-xr-x 2 abhijit abhijit    4096 May 17 11:16 fork-exec
-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp
```

3 sets of 3 permissions

3 sets = user (owner), group, others

3 permissions = read, write, execute

Owner

size

name

last-modification

hard link count

File Permissions on Linux

- r on a file : can read the file
  - open(.... O_RDONLY) works
- w on a file: can modify the file
  - open(.... O_WRONLY) works
- x on a file: can ask the os to run the file as an executable program
  - exec(...) works
- r on a directory: can do 'ls'
- w on a directory: can add/remove files from that directory (even without 'r'!)
- x on a directory: can 'cd' to that directory

Access rights examples

- **-rw-r--r--**
  Readable and writable for file owner (actually a process started by the owner!), only readable for others

- **-rw-r-----**
  Readable and writable for file owner, only readable for users belonging to the file group.

- **drwx------**
  Directory only accessible by its owner

- **-------r-x**
  File executable by others but neither by your friends nor by yourself. Nice protections for a trap...

Permissions: more !

- Setuid/setgid bit

  $ ls -l /usr/bin/passwd

  -rwsr-xr-x 1 root root 68208 Nov 29 17:23 /usr/bin/passwd

- How to set the s bit?
  - chmod u+s <filename>
- What does this mean?
  - Any user can run this process, but the process itself runs as if run by the the owner of the file
    - passwd runs as if run by “root” even if you run it

Man Pages

- Manpage
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir
  - 2 System calls
    - int mkdir(const char *, …);
  - 3 Library calls
    - int printf(const char *, …);
  - 4 Device drivers and network protocols
    - /dev/tty
  - 5 Standard file formats
    - /etc/hosts
  - 6 Games and demos
    - /usr/games/fortune
  - 7 Misc. files and docs
    - man 7 locale
  - 8 System admin. Cmds
    - /sbin/reboot

GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
/dev/	Files representing devices
	/dev/hda: first IDE hard disk
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

GNU / Linux filesystem structure

/lost+found	Corrupt files the system tried to recover
/media	Mount points for removable media:
	/media/usbdisk, /media/cdrom
/mnt/	Mount points for temporarily mounted filesystems
/opt/	Specific tools installed by the sysadmin
	/usr/local/ often used instead
/proc/	Access to system information
	/proc/cpuinfo, /**proc/version** …
/root/	root user home directory
/sbin/	Administrator-only commands
/sys/	System and device controls
	(cpu frequency, device power, etc.)

GNU / Linux filesystem structure

/tmp/	Temporary files
/usr/	Regular user tools (not essential to the system) **/usr/bin/**, **/usr/lib/**, **/usr/sbin**…
/usr/local/	Specific software installed by the sysadmin (often preferred to **/opt/**)
/var/	Data used by the system or system servers **/var/log/**, **/var/spool/mail** (incoming mail), **/var/spool/lpd** (print jobs)…

Files: cut, copy, paste, remove,

- cat <filenames>
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- cp <source> <target>
  - cp a.c b.c
  - cp a.c /tmp/
  - cp a.c /tmp/b.c
  - cp -r ./folder1 /tmp/
  - cp -r ./folder1 /tmp/folder2
- mv <source> <target>
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- rm <filename>
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a
- mkdir
  - mkdir /tmp/a /tmp/b
- rmdir
  - rmdir /tmp/a /tmp/b

Useful Commands

- echo
  - echo hi
  - echo hi there
  - echo “hi there”
  - j=5; echo $j
- sort
  - sort
  - sort < /etc/passwd
- firefox
- libreoffice
- grep
  - grep bash /etc/passwd
  - grep -i display /etc/passwd
  - egrep -i 'a|b' /etc/passwd
- less <filename>
- head <filename>
  - head -5 <filename>
  - tail -10 <filename>

Useful Commands

- **alias**

  alias ll='ls -l'

- **tar**

  tar cvf folder.tar folder

- **gzip**

  gzip a.c

- **touch**

  touch xy.txt

  touch a.c

- **strings**

  strings a.out

- **adduser**

  sudo adduser test

- **su**

  su administrator

Useful Commands

- df

  df -h

- du

  du -hs .

- bc
- time
- date
- diff
- wc
- dd

Network Related Commands

- ifconfig
- ssh
- scp
- telnet
- ping
- w
- last
- whoami

Unix job control

- **Start a background process:**
  - **gedit a.c &**
  - **gedit**
    ***hit* ctrl-z**
    bg
- **Where did it go?**
  - **jobs**
  - **ps**
- **Terminate the job: kill it**
  - **kill %*jobid***
  - **kill *pid***
- **Bring it back into the foreground**
  - **fg %1**

Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in */etc*
- */etc/*passwd and /*etc*/shadow
  - Text files containing user accounts
- */etc/*resolv.conf
  - DNS configuration
- */etc/network/interfaces*
  - *Network configuration*
- */etc/*hosts
  - Local database of Hostname-IP mappings
- */etc/*apache2/apache2.conf
  - Apache webserver configuration

~/.bashrc file

- **~/.bashrc**

  Shell script read each time a **bash** shell is started
- You can use this file to define
  - Your default environment variables (**PATH**, **EDITOR**...).
  - Your aliases.
  - Your prompt (see the **bash** manual for details).
  - A greeting message.
- Also ~/.bash_history

Mounting

Partition

- What is C:\ , D:\, E:\ etc on your computer ?
  - "Drive" is the popular term
  - Typically one of them represents a CD/DVD RW
- What do the others represent ?
  - They are "partitions" of your "hard disk"

Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- A logical chunk of storage is partition
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

Partitions

Disk Management
File Action View Help
Volume | Layout | Type | File System | Status | Capacity | Free Space | % Fr
(E:) | Partition | Basic | FAT32 | Healthy | 9.76 GB | 8.37 GB | 85 %
(F:) | Partition | Basic | FAT32 | Healthy | 9.76 GB | 7.24 GB | 74 %
OLDDRIVE (H:) | Partition | Basic | FAT32 | Healthy (A... | 4.99 GB | 586 MB | 11 %
WINDOWS XP (C:) | Partition | Basic | FAT32 | Healthy (S... | 9.76 GB | 2.61 GB | 26 %
Windows Vista (G:) | Partition | Basic | NTFS | Healthy | 8.00 GB | 1.61 GB | 20 %
XPBACKUP (I:) | Partition | Basic | FAT32 | Healthy | 4.99 GB | 4.33 GB | 86 %
XXCOPY (J:) | Partition | Basic | FAT32 | Healthy | 9.00 GB | 4.32 GB | 47 %
Disk 0
Basic
37.30 GB
Online
WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)
(E:)
9.77 GB FAT32
Healthy
(F:)
9.77 GB FAT32
Healthy
Windows Vista (G:)
8.00 GB NTFS
Healthy
Disk 1
Basic
19.01 GB
Online
OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)
XPBACKUP (I:)
5.00 GB FAT32
Healthy
XXCOPY (J:)
9.01 GB FAT32
Healthy
Primary partition
Extended partition
Logical drive

Managing partitions and hard drives

- System → Administration → Disk Utility
- Use gparted or fdisk to partition drives on Linux
- Had drive partition names on Linux
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, ... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- Formatting: creating an empty layout on disk, layout capable of storing the tree of files/folders
  - There are different layouts named ext4, ext2, ntfs, etc.

Windows Namespace

c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”

Windows Namespace

- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data

- One “tree” per partition
  - Together they make a “forest”

Linux Namespace: On a partition

/usr/songs/xyz.mp3

- On every partition:
  - Root is “/”
  - Separator is also “/”

Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”

Linux namespace
Mounting a partition

Mounting another partition on */home/guest/mydir/* directory

Mounted disk partition

/home/guest/mydir/x/b.cpp → way to access the file on the other disk partition

Mounting across network!

Using Network File System (NFS)

sudo apt install nfs-common

$ sudo mount 172.16.1.75:/mnt/data */myfolder*

Files that are not regular/directory

Special devices (1)

Device files with a special behavior or contents

- **/dev/null**
  The data sink! Discards all data written to this file.
  Useful to get rid of unwanted output, typically log information:
  mplayer black_adder_4th.avi &> /dev/null

- **/dev/zero**
  Reads from this file always return **\0** characters
  Useful to create a file filled with zeros:
  dd if=/dev/zero of=disk.img bs=1k count=2048

See **man null** or **man zero** for details

Special devices (2)

- **/dev/random**
  Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness ("entropy").
  Reads can be blocked until enough entropy is gathered.

- **/dev/urandom**
  For programs for which pseudo random numbers are fine.
  Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See **man random** for details.

Files names and inodes

Hard Links Vs Soft Links

Creating “links”

- **Hard link**

```
$ touch m
$ ls -l m
-rw-rw-r-- 1 abhijit abhijit 0 Jan  5 16:18 m
$ ln m mm
$ ls -l m mm
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 m
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 mm
$ ln mm mmm
$ ls -l m mm mmm
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 m
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 mm
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 mmm
$ echo Hi > m
$ ls -l m mm mmm
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 m
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 mm
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 mmm
$ cat m
Hi
$ cat mm
Hi
```

- **Soft Link**

```
$ ln -s m y
$ ls -l m y
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 m
lrwxrwxrwx 1 abhijit abhijit 1 Jan  5 16:18 y -> m
```

Understanding Boot Process
BIOS, UEFI, etc.

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

All credits: “Hands on Booting” by Yogesh Babar

What do we already know about “boot sequence”?

- When POWERed-ON
  - The CPU’s IP (or CS+IR, etc) has manufacturer defined value
    - The Motherboard manufacturer has ensured that a code called “Basic Input Output System(BIOS)” is at this location
  - CPU starts doing fetch-decode-execute-repeat
    - So CPU is running BIOS
  - BIOS
    - code does a Power On Self Test (POST)
    - Initializes hardware by writing to control ports, etc.
    - Reads list of boot-devices, and finds the first available device

What do we already know about “boot sequence”?

- BIOS (contd)
  - Reads “Sector-0” of the “Boot-device”, and loads it in RAM
    - Sector-0 is “boot-loader’ (at least that’s what we said!)
  - Jumps to it. Now boot-loader is running
- Boot-loader
  - Loads the OS kernel (boot-loader somehow knows where kernel is!) in RAM
  - Jumps to kernel code
  - Now kernel is running
- Kernel
  - Creates first process... and then fork-exec will create other processes

Legacy boot
The world of BIOS + MBR

Primary and Logical Partitions

- Max 4 primary partitions
- Want more?
  - 2nd or 3rd or 4th partition can be made "extended"
  - Within extended partition create logical partitions

Primary and Logical Partitions

Partition	File System	Label	Size	Used	Unused	Flags
/dev/sda1	fat32	XP2009	9.77 GiB	9.78 MiB	9.76 GiB	
/dev/sda2	ext3	Solaris 2009	9.77 GiB	307.23 MiB	9.47 GiB	
/dev/sda3	ext3	PC-BSD 9	9.77 GiB	307.23 MiB	9.47 GiB	
/dev/sda4	extended		45.70 GiB	---	---	
/dev/sda5	ntfs	windows 2003	9.77 GiB	50.75 MiB	9.72 GiB	
/dev/sda6	ntfs	windows 7	9.77 GiB	50.75 MiB	9.72 GiB	
/dev/sda7	ext3	RHEL 6	9.77 GiB	307.23 MiB	9.47 GiB	
/dev/sda8	ext3	Fedora 15	9.77 GiB	307.23 MiB	9.47 GiB	
/dev/sda9	linux-swap	swap	6.63 GiB	0.00 B	6.63 GiB	

Primary and Logical Partitions

Figure 2-6. *The disk layout on a BIOS-based system*

OS	Rules
Unix	Unix operating systems (OpenSolaris and BSD) have to be installed on a primary partition only.
Linux	Linux does not have any installation rules. It can be installed on any primary or logical partition.
Windows	The Windows operating system can be installed on any partition (primary or logical), but the predecessor of the Windows family has to be present on the first primary. That means you can install Windows 7 on a logical partition, but its predecessor, which is XP or win2k3, has to be present on the first primary partition. Also, you cannot break the Windows operating system sequence of installation. For example, one cannot install Windows 7 first and then the older win2k3 or XP. It has to be in this sequence: 98, then 2000, and then XP.

Windows XP installer

```
Windows XP Professional Setup

The following list shows the existing partitions and
unpartitioned space on this computer.

Use the UP and DOWN ARROW keys to select an item in the list.

   • To set up Windows XP on the selected item, press ENTER.

   • To create a partition in the unpartitioned space, press C.

   • To delete the selected partition, press D.

76796 MB Disk 0 at Id 1 on bus 0 on atapi [MBR]

   C:  Partition1 (XP2009) [FAT32]        10000 MB (  9990 MB free)
   G:  Partition2 [Unknown]               10000 MB ( 10000 MB free)
   H:  Partition3 [Unknown]               10000 MB ( 10000 MB free)
   D:  Partition4 (windows 200) [NTFS]    10000 MB (  9949 MB free)
   E:  Partition5 (windows 7) [NTFS]      10000 MB (  9949 MB free)
   I:  Partition6 [Unknown]               10000 MB ( 10000 MB free)
   J:  Partition7 [Unknown]               10000 MB ( 10000 MB free)
   K:  Partition8 [Unknown]                6794 MB (  6794 MB free)

ENTER=Install  D=Delete Partition  F3=Quit
```

Figure 2-4. *Partition layout shown by XP's installer*

“Boot sector” is a misnomer here! it’s more than a sector!

Figure 2-7. *The boot sector*

Bootloader code is in three parts!

Figure 2-7. *The boot sector*

Bootloader code is in three parts!

Location	Size	Part	Information
Bootstrap	440 bytes	NTLDR part-1	The tiniest part
Bootloader	31 KB	NTLDR part-2	Bigger compared to part-1
Inside an actual OS partition	No size limitation	NTLDR part-3	The biggest part

See C:\NTLDR (NTLDR-3 file) file on Windows XP
Try deleting this file and booting

C:\Windows\winload.exe is Windows kernel

Boot sequence of Win-Xp

Figure 2-9. *The boot sequence of Windows XP*

Note: The diagram is “conceptual”. In reality, each stage loads the next stage in RAM and “jumps” (that is calls “jump” instruction) to the location of the next stage in RAM. NO jumps happen on disk!

More on boot region

- Vendor signature: Seagate, WD, etc.
- NULL (is 0, else disk is bad)
- MBR: reason why only 4 primary partitions!

Size	Parts	Stores
16 bytes	Part-1	First partition's information
16 bytes	Part-2	Second partition's information
16 bytes	Part-3	Third partition's information
16 bytes	Part-4	Fourth partition's information

1 sector = 512 Bytes

More on boot region

- **Fdisk signature**
  - **Can be "*" or something else**
  - **When *, called boot flag, or active/inactive flag**
  -

More on boot region

- **Fdisk signature**
  - **Can be "*" or something else**
  - **When , called boot flag, or active/inactive flag, OS kernel is supposed to be loaded from partition with "*" flag**
    - **Makes sense only when you have multi-boot system**
  - **BIOS -> Bootstrap code (440 bytes) -> Bootloader (31 kB) of first partition -> where – partition which has "*" on it**

Flowchart

- Run: BIOS
- Run: Partition-1, Bootloader part-1: Sector-0 , 440 bytes
- Run: Partition-1, Bootlaoder part-2: 31k
  - Checks where is the “*” mark, that is the active flag
    - If on partition-1
      - Run Partition-1, Bootlaoder part-3
    - Else
      - Run, Partition-x, Bootloader, part-0
- The Bootloader-3 part of whichever partition in the end
  - should be able to recognize the other operating systems
  - Linux recognizes all, windows recognizes only itself, for other OS: depends

UEFI
Unified Extended Firmware Interface

BIOS Limitations

- BIOS will only be able to jump to the first sector, which is 512 bytes.
- BIOS cannot generate good graphics/GUIs.
- You cannot use a mouse in the BIOS.
- The maximum partition size is 2.2 TB.
- The BIOS is dumb because it does not understand the bootloader or the OS.
- It struggles to initialize the new-generation hardware devices.

UEFI advantages

- UEFI can use the full CPU. Unlike the BIOS (which is stuck with 16 bits of processor), UEFI can access up to 64 bits.
- UEFI can use a full RAM module. Unlike 1 MB of address space of the BIOS, UEFI can support and use terabytes of RAM.
- Instead of 64 bytes of a tiny MBR, UEFI uses the GPT (GUID) partition table, which will provide an infinite number of partitions, and all will be primary partitions. In fact, there is no concept of primary and logical partitions.
- A maximum partition size is 8 zettabytes.
- UEFI has enterprise management tools.
  - Boot remotely
  - Browse internet
  - Change firmware remotely

UEFI advantages

- It's a small OS
  - a) You will have full access to audio and video devices.
  - b) You will be able to connect to WiFi.
  - c) You will be able to use the mouse.
  - d) In terms of the GUI, UEFI will provide a rich graphics interface.
  - e) UEFI will have its own app store like we have for Android and Apple phones.
  - f) You will be able to download and use the applications from the UEFI app store, just like with Android and Apple phones. Hundreds of apps are available such as calendars, email clients, browser, games, shells, etc.
  - g) UEFI is able to run any binary that has an EFI executable format.
  - h) It boots operating systems securely with the help of the Secure Boot feature. We will discuss the Secure Boot feature in depth later in this book.
  - i) UEFI is backward compatible, meaning it will support the "BIOS way" of booting. In other words, operating systems that do not have UEFI support will also be able to boot with UEFI.

Figure 2-105. *ASUS UEFI implementation*

Figure 2-106. *The UEFI structure*

```
root@yogesh-virtual-machine:/home/yogesh# ls -l /boot/efi/EFI/ubuntu/
total 3724
-rwx------ 1 root root     108 Dec  2 17:47 BOOTX64.CSV
drwx------ 2 root root    4096 Dec  2 17:47 fw
-rwx------ 1 root root   75992 Dec  2 17:47 fwupx64.efi
-rwx------ 1 root root     126 Dec  2 17:47 grub.cfg
-rwx------ 1 root root 1116024 Dec  2 17:47 grubx64.efi
-rwx------ 1 root root 1269496 Dec  2 17:47 mmx64.efi
-rwx------ 1 root root 1334816 Dec  2 17:47 shimx64.efi
root@yogesh-virtual-machine:/home/yogesh#
```

Figure 2-111. *The EFI directory of Ubuntu*

Figure 2-112. *The boot priority window of UEFI*

Figure 2-114. *Ubuntu's boot sequence*

Boot Manager

Boot normally

Continue to boot using the default boot order.

ubuntu
Windows Boot Manager
EFI VMware Virtual SCSI Hard Drive (0.0)
EFI VMware Virtual SATA CDROM Drive (1.0)
EFI Network

EFI Internal Shell (Unsupported option)

Enter setup
Reset the system
Shut down the system

↑↓=Move Highlight <Enter>=Select Entry

Figure 2-121. *The boot sequence of Windows 10*

Processes

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

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
  - List of all processes
  - Memory management details for each, files opened by each etc.
  - Scheduling information about the process
  - Status of the process
  - List of processes “waiting” for different events to occur,
  - Etc.

Figure 3.3 Process control block (PCB).

Process Control Block

- A record representing a process in operating system's data structures
- OS maintains a "list"of PCBs, one for each process
- Called "`struct task_struct`" in Linux kernel code and "`struct proc`" in xv6 code

Figure 3.3 Process control block (PCB).

Fields in PCB

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```

List of open files

- The PCB contains an array of pointers, called file descriptor array (fd[ ]), pointers to structures representing files
- When open() system call is made
  - A new file structure is created and relevant information is stored in it
  - Smallest available of fd [ ] pointers is made to point to this new struct file
  - The index of this fd [ ] pointer is returned by open
- When subsequent calls are made to read(fd, ...) or write(fd, ...) , etc.
  - The kernel gets the “fd” as an index in the fd[ ] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

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

```
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;
};
```

Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
  - List of processes currently running on multiple CPUs
  - Etc.

```
// Linux data structure
struct task_struct {
    long state;/*state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /*this process's parent */
    struct list_head children; /*this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm;/*address space */
```

```
struct list_head {
    struct list_head
*next, *prev;
};
```

Figure 3.4 The ready queue and wait queues.

Figure 3.5 Queueing-diagram representation of process scheduling.

Figure 3.2 Diagram of process state.

See state in output of ps axu
(BSD style options, without -)

Conceptual diagram

On Linux

“Giving up” CPU by a process or blocking

```
int main() {
i = j + k;
scanf("%d", &k);
}
int scanf(char *x, ...) {
...
read(0, ..., ...);
}
int read(int fd, char *buf, int len) {
...
__asm__ { "int 0x80..."}
...
}
```

OS Syscall

```
sys_read(int fd, char *buf, int len) {
  file f  = current->fdarray[fd];
  int offset = f->position;
  ...
  disk_read(... , offset, ...);
  // Do what now?
  //asynchronous read
  //Interrupt will occur when the disk read is complete
  // Move the process from ready queue to a wait queue and call scheduler!
  // This is called "blocking"
  Return the data read ;
}
disk_read(...., offset, .... ) {
__asm__("outb PORT ..");
return;
}
```

“Giving up” CPU by a process or blocking

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

To be seen later

Context Switch

- Context
  - Execution context of a process
  - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
  - Change the context from one process/OS to OS/another process
  - Need to save the old context and load new context
  - Where to save? --> PCB of the process

Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go

Figure 3.6 Diagram showing context switch from process to process.

Pecularity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention
- When an interrupt occurs
  - It can occur anytime
  - Context switch can happen in the middle of execution of any function
- After context switch
  - One process takes place of another
  - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
  - Code for context switch must be in assembly!

System Calls, fork(), exec()

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

(C) Abhijit A.M.
Available under Creative Commons Attribution-ShareAlike License V3.0+

Credits: Slides of “OS Book” ed10.

Figure 2.1 A view of operating system services.

WHAT IS AN OPERATING SYSTEM?

Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

System Calls

- Services provided by operating system to applications
  - Essentially available to applications by calling the particular software interrupt application
    - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
    - Different arguments specified in EAX register inform the kernel about different system calls
- The C library has wrapper functions for each of the system calls
  - E.g. open(), read(), write(), fork(), mmap(), etc.

Types of System Calls

- **File System Related**
  - **Open(), read(), write(), close(), etc.**
- **Processes Related**
  - **Fork(), exec(), ...**
- **Memory management related**
  - **Mmap(), shm_open(), ...**
- **Device Management**
- **Information maintainance – time,date**
- **Communication between processes (IPC)**
- **Read man syscalls**

https://linuxhint.com/list_of_linux_syscalls/

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	write()
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Code schematic

```
int main() {
    int a = 2;
    printf("hi\n");
}
-----
C Library
-----
int printf("void *a, ...) {
    ...
    write(1, a, ...);
}
int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl  %eax, -4(%ebp)");
# -4ebp is ret
    return ret;
}
```

```
-------user-kernel-mode-boundary----
//OS code
int sys_write(int fd, char *, int len) {
    figure out location on disk
    where to do the write and
    carry out the operation,
    etc.
}
```

Figure 2.6 The handling of a user application invoking the `open()` system call.

Two important system calls
Related to processes

fork() and exec()

Process

- A program in execution
- Exists in RAM
- Scheduled by OS
  - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- The “ps” command on Linux

Process in RAM

- **Memory is required to store the following components of a process**
  - **Code**
  - **Global variables (data)**
  - **Stack (stores local variables of functions)**
  - **Heap (stores malloced memory)**
  - **Shared libraries (e.g. code of printf, etc)**
  - **Few other things, may be**

Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system
Only one program in RAM at a time, and only one program can run at a time

A multi tasking system
With multiple programs loaded in memory
Along with kernel
(A very simplified conceptual diagram. Things are more complex in reality)

fork()

- A running process creates it’s duplicate!
- After call to fork() is over
  - Two processes are running
  - Identical
  - The calling function returns in two places!
  - Caller is called parent, and the new process is called child
  - PID is returned to parent and 0 to child

exec()

- Variants: execvp(), execl(), etc.
- Takes the path name of an executable as an argument
- Overwrites the existing process using the code provided in the executable
- The original process is OVER ! Vanished!
- The new program starts running overwritting the existing process!

Shell using fork and exec

- **Demo**
- **The only way a process can be created on Unix/Linux is using fork() + exec()**
- **All processes that you see were started by some other process using fork() + exec() , except the initial *"init" process***
- ***When you click on "firefox" icon, the user-interface program does a fork() + exec() to start firefox; same with a command line shell program***
- **The "bash" shell you have been using is nothing but an advanced version of the shell code shown during the demo**
- **See the process tree starting from "init"**
- **Your next assignment**

The boot process, once again

- BIOS
- Boot loader
- OS – kernel
- Init created by kernel by Hand(kernel mode)
- Kernel schedules init (the only process)
- Init fork-execs some programs (user mode)
  - Now these programs will be scheduled by OS
- Init -> GUI -> terminal -> shell
  - One of the typical parent-child relationships

Event Driven kernel
Multi-tasking, Multi-programming

Earlier...

- **Boot process**
  - **BIOS -> Boot Loader -> OS -> “init” process**
- **CPU doing**

```
for(;;) {
    fetch from @PC;
    deode+execute;
    PC++/change PC
}
```

Understanding hardware interrupts

- Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”
- Basically create an electrical signal on some connection to CPU (/bus)
- This is notified to CPU (in hardware)
- Now CPU’s normal execution is interrupted!
  - What’s the normal execution?
  - CPU will not continue doing the fetch, decode, execute, change PC cycle !
  - What happens then?

Understanding hardware interrupts

- On Hardware interrupt
  - The PC changes to a location pre-determined by CPU manufacturers!
  - Now CPU resumes normal execution
    - What's normal?
    - Same thing: Fetch, Decode, Execute, Change PC, repeat!
    - But...
  - But what's there at this pre-determined address?
  - OS! How's that ?

Boot Process

- **BIOS runs “automatically” because at the initial value of PC (on power ON), the manufacturers have put in the BIOS code in ROM**
  - **CPU is running BIOS code . BIOS code also occupies all the addresses corresponding to hardware interrupts.**
- **BIOS looks up boot loader (on default boot device) and loads it in RAM, and passes control over to it**
  - **Pass control? - Like a JMP instruction**
  - **CPU is running boot loader code**
- **Boot loader gets boot option from human user, and loads the selected OS in memory and passes control over to OS**
  - **Now OS is running on the processor !**

Hardware interrupts and OS

- **When OS starts running initially**
  - **It copies relevant parts of it's own code at all possible memory addresses that a hardware interrrupt can lead to!**
  - **Intelligent, isnt' it?**
- **Now what?**
  - **Whenever there is a hardware interrupt – what will happen?**
  - **The PC will change to predetermined location, and control will jump into OS code**
- **So remember: whenever there is a hardware interrupt, OS code will run!**
- **This is "taking control of hardware"**

Key points

- **Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware**
- **Most features of computer systems / operating systems are derived from hardware features**
  - **We will keep learning this throught the course**
  - **Hardware support is needed for many OS features**

Time Shared CPU

- Normal use: after the OS has been loaded and Desktop environment is running
- The OS and different application programs keep executing on the CPU alternatively (more about this later)
  - The CPU is time-shared between different applications and OS itself
- Questions to be answered later
  - How is this done?
  - How does OS control the time allocation to each?
  - How does OS choose which application program to run next?
  - Where do the application programs come from? How do they start running ?
  - Etc.

Multiprogramming

- **Program**
  - **Just a binary (machine code) file lying on the hard drive. E.g. /bin/ls**
  - **Does not do anything!**
- **Process**
  - **A program that is executing**
  - **Must exist in RAM before it executes. Why?**
  - **One program can run as multiple processes. What does that mean?**

Multiprogramming

- **Multiprogramming**
  - **A system where multiple processes(!) exist at the same time in the RAM**
  - **But only one runs at a time!**
    - **Because there is only one CPU**
- **Multi tasking**
  - **Time sharing between multiple processes in a multi-programming system**
  - **The OS enables this. How? To be seen later.**

Question

- **Select the correct one**

  1) A multiprogramming system is not necessarily multitasking

  2) A multitasking system is not necessarily multiprogramming

Events , that interrupt CPU's functioning

- **Three types of "traps" : Events that make the CPU run code at a pre-defined address**

  1) Hardware interrupts

  2) Software interrupt instructions (trap)

  E.g. instruction "int"

  3) Exceptions

  e.g. a machine instruction that does division by zero

  Illegal instruction, etc.

  Some are called "faults", e.g. "page fault", recoverable

  some are called "aborts", e.g. division by zero, non-recoverable

- **The OS code occupies all memory locations corresponding to the PC values related to all the above events!**

Multi tasking requirements

- Two processes should not be
  - Able to steal time of each other
  - See data/code of each other
  - Modify data/code of each other
  - Etc.
- The OS ensures all these things. How?
  - To be seen later.

But the OS is “always” “running”
“in the background”
Isn’t it?

Absolutely No!

Let’s understand
What kind of
Hardware, OS interplay
makes
Multitasking possible

Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types
- Normal instructions
  - mov, jmp, add, etc.
- Privileged instructions
  - Normally related to hardware devices
  - E.g.
    - IN, OUT # write to I/O memory locations
    - INTR # software interrupt, etc.

Two types of CPU instructions and two modes of CPU operation

- CPUs have a mode bit (can be 0 or 1)
- The two values of the mode bit are called: User mode and Kernel mode
- If the bit is in user mode
  - Only the normal instructions can be executed by CPU
- If the bit is in kernel mode
  - Both the normal and privileges instructions can be executed by CPU
- If the CPU is “made” to execute privileged instruction when the mode bit is in “User mode”
  - It results in a illegal instruction execution
  - Again running OS code!

Two types of CPU instructions and two modes of CPU operation

- The opearting system code runs in kernel mode.
  - How? Wait for that!
- The application code runs in user mode
  - How? Wait !
  - So application code can not run privileged hardware instructions
- Transition from user mode to kernel mode and vice-versa
  - Special instruction called “software interrupt” instructions
  - E.g. INT instruction on x86

Software interrupt instruction

- E.g. INT on x86 processors
- Does two things at the same time!
  - Changes mode from user mode to kernel mode in CPU

    + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
    - Close to the way a hardware interrupt works. Isn't it?
  - Why two things together?
  - What's there are the pre-defined location?
    - Obviously, OS code. OS occupied these locations in Memory, at Boot time.

Software interrupt instruction

- What's the use of these type of instructions?
  - An application code running INT 0x80 on x86 will now cause
    - Change of mode
    - Jump into OS code
  - Effectively a request by application code to OS to do a particular task!
  - E.g. read from keyboard or write to screen !
  - OS providing hardware services to applications !
  - A proper argument to INT 0x80 specified in a proper register indicates different possible request

Figure 1.10 Transition from user to kernel mode.

Software interrupt instruction

- How does application code run INT instruction?
  - C library functions like printf(), scanf() which do I/O requests contain the INT instruction!
  - Control flow
    - Application code -> printf -> INT -> OS code -> back to printf code -> back to application code

Example: C program

```
int main() {
    int i, j, k;
    k = 20;
    scanf("%d", &i);  // This jumps into OS and returns back
    j = k + i;
    printf("%d\n", j); // This jumps into OS and returns back
    return 0;
}
```

Interrupt driven OS code

- **OS code is sitting in memory , and runs intermittantly . When?**
  - **On a software or hardware interrupt or exception!**
  - **Event/Interrupt driven OS!**
  - **Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code**

Interrupt driven OS code

- **Timer interrupt and multitasking OS**
  - **Setting timer register is a privileged instruction.**
  - **After setting a value in it, the timer keeps ticking down and on becoing zero the timer interrupt is raised again (in hardware automatically)**
  - **OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !**
  - **When time allocated to process is over, the timer interrupt occurs and control jumps back to OS (hardware interrupt mechanism)**
  - **The OS code that runs here (the ISR) is called “scheduler”**
  - **OS can now schedule another program**

What runs on the processor ?

- **4 possibilities.**

Compilation, Linking, Loading

Abhijit A M

Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

- Boot sequence: Process world
  - kernel->init -> many forks+execs() -> ....
- Hardware interrupts, system calls, exceptions
- Event driven kernel
- System calls
  - Fork, exec, ... open, read, ...

What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

Standard C Library

- A collection of some of the most frequently needed functions for C programs
  - scanf, printf, getchar, system-call wrappers (open, read, fork, exec, etc.), ...
- An machine/object code file containing the machine code of all these functions
  - Not a source code! Neither a header file. More later.
- Where is the C library on your computer?
  - /usr/lib/x86_64-linux-gnu/libc-2.31.so

Compiler

- application program, which converts one (programming) language to another
  - Most typically compilers convert a high level language like C, C++, etc. to Machine code language
- E.g. GCC /usr/bin/gcc
  - Usage: e.g.
  - $ gcc main.c -o main
  - Here main.c is the C code, and "main" is the object/machine code file generated

- Input is a file and output is also a file.
- Other examples: g++ (for C++), javac (for java)

Assembler

- application program, converts assembly code into machine code
- What is assembly language?
  - Human readable machine code language.
- E.g. x86 assembly code
  - mov 50, r1
  - add 10, r1
  - mov r1, 500
- Usage. eg..
  - $ as something.s -o something

Compilation Process

Figure 2.11 The role of the linker and loader.

- From the textbook

Example

try.c

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
	int i, j, k;
	scanf("%d%d", &i, &j);
	k = f(i, j) + MAX;
	printf("%d\n", k);
	return 0;
}
```

f.c

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
	return ADD(m,n) + g(10);
}
```

g.c

```
int g(int x) {
	return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to understand what is happening

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

More about the steps

- Pre-processor
  - #define ABC XYZ
    - cut ABC and paste XYZ
  - # include <stdio.h>
    - copy-paste the file stdio.h
    - There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.
- Linking
  - Normally links with the standard C-library by default
  - To link with other libraries, use the -l option of gcc

Using gcc itself to understand the process

- Run only the preprocessor
  - cc -E test.c
  - Shows the output on the screen
- Run only till compilation (no linking)
  - cc -c test.c
  - Generates the “test.o” file , runs compilation + assembler
  - gcc -S main.c
  - One step before machine code generation, stops at assembly code
- Combine multiple .o files (only linking part)

Linking process

- Linker is an application program
  - On linux, it's the "ld" program
  - E.g. you can run commands like $ ld a.o b.o -o c.o
  - Normally you have to specify some options to ld to get a proper executable file.
- When you run gcc
  - $ cc main.o f.o g.o -o try
  - the CC will internally invoke "ld" . ld does the job of linking

Linking process

- The resultatnt file "try" here, will contain the codes of all the functions and linkages also.
- **What is linking?**
  - "connecting" the call of a function with the code of the function.
- What happens with the code of printf()?
  - The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

Executable file format

- An executable file needs to execute in an environment created by OS and on a particular processor
  - Contains machine code + other information for OS
  - Need for a structured-way of storing machine code in it
- Different OS demand different formats
  - Windows: PE, Linux: ELF, Old Unixes: a.out, etc.

- ELF : The format on Linux.
- Try this
  - $ file */bin/ls*
  - $ file /usr/lib/x86_64-linux-gnu/libc-2.31.so

Exec() and ELF

- When you run a program
  - $ ./try
  - Essentially there willl be a fork() and exec("./try", ...")
  - So the kernel has to read the file "./try" and understand it.
  - So each kernel will demand it's own object code file format.
  - Hence ELF, EXE, etc. Formats
- ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6
- What is a.out?
  - "a.out" was the name of a format used on earleir Unixes.
  - It so happened that the early compiler writers, also created executable with default name 'a.out'

Utilities to play with object code files

- objdump
  - $ objdump -D -x /bin/ls
  - Shows all disassembled machine instructions and “headers”
- hexdump
  - $ hexdump /*bin*/ls
  - Just shows the file in hexadecimal
- readelf
  - Alternative to objdump
- ar
  - To create a “statically linked” library file
  - $ ar -crs libmine.a one.o two.o
- Gcc to create shared library
  - $ gcc hello.o -shared -o libhello.so
- To see how gcc invokes as, ld, etc; do this
  - `$ gcc -v hello.c` -o hello
  - /* https://stackoverflow.com/questions/1170809/how-to-get-gcc-linker-command

Linker, Loader, Link-Loader

- Linker or linkage-editor or link-editor
  - The “ld” program. Does linking.
- Loader
  - The exec(). It loads an executable in the memory.
- Link-Loader
  - Often the linker is called link-loader in literature. Because where were days when the linker and loader’s jobs were quite over-lapping.

Static, dynamic / linking, loading

- Both linking and loading can be
  - Static or dynamic
  - More about this when we learn memory management
- An important fundamental:
  - memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.
  - They all should fit into each other to make a system work
  - That's why the phrase "system programs"

Cross-compiler

- Compiler on system-A, but generate object-code file for system-B (target system)
  - E.g. compile on Ubuntu, but create an EXE for windows
- Normally used when there is no compiler available on target system
  - see gcc -m option
- See https://wiki.osdev.org/GCC_Cross-Compiler

Inter Process Communication

Revision of process related concepts

- PCB, struct proc
- Process lifecycle – different states
- Queues/Lists of processes
- What is “Blocking”
- Event driven kernel

Before IPC, let's learn more about file related system calls

- **Redirection**

  ls > /*tmp*/file

  cat < /etc/passwd

  how does this work?

- **File descriptors are inherited across fork**

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)
- Broadly Two models of IPC
  - Shared memory (examples: shared memory, pipes, ...)
  - Message passing (examples: send/recv on socket, send-recv messages, ...)

Shared Memory Vs Message Passing

Figure 3.11 Communications models. (a) Shared memory. (b) 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!

Typical code of shared memory type of solutions

Process P1

```
x = getshm(ID)
*x = 100;
```

Process P2

```
x = getshm(ID)
y = *x
```

Typical code of message passing type solutions

Process P1

```
send(P2, x)
or
broadcast(x)
```

Process P2

```
y = receive(P1)
or
y = receive(anyone)
```

Example of co-operating processes: Producer Consumer Problem

- **Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process**
  - **unbounded-buffer places no practical limit on the size of the buffer**
  - **bounded-buffer assumes that there is a fixed buffer size**

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

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

parent
fd(0)
fd(1)
child
fd(0)
fd(1)
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 pipes

- int mkfifo(const char *pathname, mode_t mode);
- Example

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(shared_memory);
```

Example of Shared memory
POSIX Shared Memory

- **What is POSIX?**
  - **Portable Operating System Interface (POSIX)**
  - **family of standards**
  - **specified by the IEEE Computer Society**
  - **for maintaining compatibility between operating systems.**
  - **API (system calls), shells, utility commands for compatibility among UNIXes and variants**

POSIX Shared Memory

- shm_open
- ftruncate
- Mmap
- See the example in Textbook

Message passing

Message Passing

- Message system – processes communicate with each other using send(), receive() like syscalls given by OS
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - Receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Communication link can be implemented in a variety of ways

Message Passing using "Naming"

- Pass a message by "naming" the receiver
  - A) Direct communication with receiver
    - Receiver is identified by sender directly using it's name
  - B) Indirect communication with receiver
    - Receiver is identified by sender in-directly using it's 'location of receipt'

Message passing using direct communication

- **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

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or 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**

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 and 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);
}
```

Message Passing implementation: choice of 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**

Memory Management – Continued

More on Linking, Loading, Paging

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**

More on Linking and Loading

- **Static Linking: All object code combined at link time and a big object code file is created**
- **Static Loading: All the code is loaded in memory at the time of exec()**
- **Problem: Big executable files, need to load functions even if they do not execute**
- **Solution: Dynamic Linking and Dynamic Loading**

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - **function code to function calls**
    - **references to global variables with “extern” declarations**
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
    11d4:        callq  1080 <printf@plt>
```

Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
    1080:      endbr64
    1084:      bnd jmpq *0x2f3d(%rip)        # 3fc8
<printf@GLIBC_2.2.5>
    108b:       nopl   0x0(%rax,%rax,1)
```

Dynamic Loading

- **Loader**
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- **Dynamic Loading**
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

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 ?**

Continous memory management

What is Continous memory management?

- **Entire process is hosted as one continous chunk in RAM**
- **Memory is typically divided into two partitions**
  - **One for OS and other for processes**
  - **OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !**

Hardware support needed: base + limit (or relocation + limit)

Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

Variable partition scheme

Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- Suppose there are 3 free memory regions of sizes 30k, 40k, 20k
- The newly created process (during fork() + exec()) needs 15k
- Which region to allocate to it ?

Strategies for finding a free chunk

- **6k, 17k, 16k, 40k holes . Need 15k.**
- **Best fit: Find the smallest hole, larger than process. Ans: 16k**
- **Worst fit: Find the largest hole. Ans: 40k**
- **First fit: Find the “first” hole larger than the process. Ans: 17k**

Problem : External fragmentation

- **Free chunks: 30k, 40k, 20k**
- **The newly created process (during fork() + exec()) needs 50k**
- **Total free memory: 30+40+20 = 90k**
  - **But can’t allocate 50k !**

Solution to external fragmentation

- **Compaction !**
- **OS moves the process chunks in memory to make available continous memory region**
  - **Then it must update the memory management information in PCB (e.g. base of the process) of each process**
- **Time consuming**
- **Possible only if the relocation+limit scheme of MMU is available**

Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
  - **If total 1M memory, then 20 such chunks**
- **Allocate one or more chunks to a process, such that the total size is >= the size of the process**
  - **E.g.if request is 50k, allocate 1 chunk**
  - **If request is 40k, still allocate 1 chunk**
  - **If request is 60k, then allocate 2 chunks**
- **Leads to internal fragmentation**
  - **space wasted in the case of 40k or 60k requests above**

Fixed partition scheme

- **OS needs to keep track of**
  - **Which partition is free and which is used by which process**
  - **Free partitions can simply be tracked using a bitmap or a list of numbers**
  - **Each process's PCB will contain list of partitions allocated to it**

Solution to internal fragmentation

- **Reduce the size of the fixed sized partition**
- **How small then ?**
  - **Smaller partitions mean more overhead for the operating system in allocating deallocating**

Paging

An extended version of fixed size partitions

- **Partition = page**
  - Process = logically continuous sequence of bytes, divided in ‘page’ sizes
  - Memory divided into equally sized page ‘frames’
- **Important distinction**
  - Process need not be continous in RAM
  - Different page sized chunks of process can go in any page frame
  - Page table to map pages into frames

Logical address seen as

Paging hardware

Figure 9.8 Paging hardware.

MMU’s job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- **Allocate a page table for the process, at time of fork()/exec()**
  - **Allocate frames to process**
  - **Fill in page table entries**
- **In PCB of each process, maintain**
  - **Page table location (address)**
  - **List of pages frames allocated to this process**
- **During context switch of the process, load the PTBR using the PCB**

Job of OS

- **Maintain a list of all page frames**
  - **Allocated frames**
  - **Free Frames (called frame table)**
  - **Can be done using simple linked list**
  - **Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)**
  -

Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- **Each memory access results in two memory accesses!**
  - **One for page table, and one for the actual memory location !**
  - **Done as part of execution of instruction in hardware (not by OS!)**
  - **Slow down by 50%**

Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number

Speedup due to TLB

- **Hit ratio**
- **Effective memory access time**

  = Hit ratio * 1 memory access time + miss ratio * 2 memory access time

- **Example: memory access time 10ns, hit ratio = 0.8, then**

  effective access time = 0.80 × 10 + 0.20 × 20

  = 12 nanoseconds

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

X86 PDE and PTE

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

Paging: problem of large PT

- **64 bit address**
- **Suppose 20 bit offset**
  - **That means 2^20 = 1 MB pages**
  - **44 bit page number: 2^44 that is trillion sized page table!**
  - **Can’t have that big continuous page table!**

Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means 2^12 = 4 KB pages
  - 20 bit page number: 2^20 that is a million entries
  - Can’t always have that big continuous page table as well, for each process!

Hierarchical paging

Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
p_1	p_2	d
42	10	12

More hierarchy

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Problems with hierarchical paging

- **More number of memory accesses with each level !**
  - **Too slow !**
- **OS data structures also needed in that proportion**

Hashed page table

Figure 9.17 Hashed page table.

Inverted page table

Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page
tables include the 64-bit Ultra SPARC and Power PC

virtual address
consists of a triple:
<process-id, page-number, offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- **Cashing levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)**
  - **CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.**
  - **A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page**
  - **When a virtual address reference occurs, the hardware searches the TLB for a translation.**
  - **If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup**

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- **Standard swapping**
  - Entire process swapped in or swapped out
  - With continous memory management
- **Swapping with paging**
  - Some pages are “paged out” and some “paged in”
  - Term “paging” refers to paging with swapping now

Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

- **Not as simple as it sounds when it comes to implementation**
  - **Writing OS code for this is challenging**

Memory Management Basics

Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program

$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other’s code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case

- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case

Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme

- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continous segment
  - Fill in the 'base' register with the actual address of the process in RAM.
  - Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **"Relocatable code"** – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme

It should be possible to have relocatable code even with “simplest case”

By doing extra work during “loading”.

How?

Next scheme: Multiple base +limit pairs

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it's in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more "base+limits" in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continous in memory!
- Divide the continuous process image in smaller chunks (let’s say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

Figure 9.9 Paging model of logical and physical memory.

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management

Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

Paging in x86

Figure 9.23 Paging in the IA-32 architecture.

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

Scheduling

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

Credits: Slides from os-book.com

Necessity of scheduling

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

CPU Scheduling

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

Observation: CPU, I/O Bursts

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

Let’s understand the problem

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

A C code example:

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

CPU bursts: observation

Large number of short bursts

Small number of longer bursts

Scheduler, what does it do?

- **From a list of processes, ready to run**
  - **Selects a process for execution**
  - **Allocates a CPU to the process for execution**
  - **Does “context switch”**

Context and Context Switch

- **Context**
  - **Set of registers. Which ones?**
  - **All or some ?**
  - **Following registers on xv6 kernel: edi, esi, ebx, ebp, eip**
    - **Related to calling convention!**
  - **Linux kernel context: Much bigger!**
  - **Also includes: MMU setup**

- **Context switch**
  - **Process context -> kernel context**
    - **On interrupt, system call, or exception**
  - **Kernel context -> process context**
    - **Returning from system call, returning from interrupt handler, scheduling a process**
  - **In every switch, need to change to the set of registers "last in use" by that context, and also the MMU setup**

When is scheduler invoked?

1) Process Switches from running to waiting state
   - Waiting for I/O, etc.
2) Switches from running to ready state
   - E.g. on a timer interrupt
3) Switches from waiting to ready
   - I/O wait completed, now ready to run
4) Terminates

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

Dispatcher: A part of scheduler

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

Dispatcher in action on Linux

- Run "`vmstat 1 3`"
  - Means run vmstat 3 times at 1 second delay
- In output, look at CPU:cs
  - Context switches every second
- Also for a process with pid 3323
- Run

  "`cat /proc/3323/status`"
  - See

    voluntary_ctxt_switches

    --> Process left CPU

    nonvoluntary_ctxt_switche

    --> Process was preempted

Scheduling criteria

- **CPU utilization: Maximise**
  - **keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.**
- **Throughput : Maximise**
  - **# of processes that complete their execution per time unit**
- **Turnaround time : Minimise**
  - **amount of time to execute a particular process**
- **Waiting time : Minimise**
  - **amount of time a process has been waiting in the ready queue**
- **Response time : Minimise**
  - **amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)**
-

To be studied later

- Evaluation of scheduling criteria
- Different scheduling algorithms, and their characteristics
  - Round Robin, FIFO, Shortest Job First, Priority, Multi-level Queue, etc.
- Thread scheduling

Multi Processor Scheduling

Multiprocessor systems

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

Symmetric multiprocessing (SMP)

Multicore systems (also SMP)

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

Multicore Processors

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

Multithreaded Multicore System

Each core has > 1 hardware threads.

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

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

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

More on SMP systems

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

Multiple-Processor Scheduling

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

Multiple-Processor Scheduling

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

common ready queue
(a)

per-core run queues
(b)

SMP in xv6

- Only one process queue
- No load balancing, no affinity (more later)
- A process may run any CPU-burst /alloted-time-quantum on any processor randomly

End

Signals

Signals

- Can be termed as a way of Inter-process communication, but often it is not categorized as IPC (why!?)
- Processes can send each other “signals”, that is indicators of an event and receiver can “act” on the receipt of the signal
  - Integers are used to denote/signify each event

Let’s see signals in action using Linux command line

- Use of “kill” command
  - Does not mean “kill” literally every time
  - “kill” means send signal!

Signals

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

Signals

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

Signals

- **Actions**
  - **Term Default action is to terminate the process.**
  - **Ign Default action is to ignore the signal.**
  - **Core Default action is to terminate the process and dump core (see core(5)).**
  - **Stop Default action is to stop the process.**
  - **Cont Default action is to continue the process if it is currently stopped.**

Demo

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

Signal handling by OS

Process 12323 {

signal(19, abcd);

}

OS: sys_signal {

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

}

Process P1 {

kill (12323, 19) ;

}

OS: sys_kill {

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

}

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

Sigaction: POSIX

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t
handler);
```

Threads and Signals

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

The “errno”
And error conventions

Notes on reading xv6 code

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

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

Introduction to xv6

Structure of xv6 code

Compiling and executing xv6 code

About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- No : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

Use cscope and ctags with VIM

- Go to folder of xv6 code and run

  `cscope -q *.[chS]`

- Also run

  `ctags *.[chS]`

- Now download the file http://cscope.sourceforge.net/cscope_maps.vim as `.cscope_maps.vim` in your ~ folder
- And add line "`source ~/.cscope_maps.vim`" in your `~/.vimrc` file
- Read this tutorial http://cscope.sourceforge.net/cscope_vim_tutorial.html

Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
  - This creates the file “doxyconfig”

Use call graphs (using doxygen)

- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

```
PROJECT_NAME           = "XV6"
OUTPUT_DIRECTORY       = ./doxygen
CREATE_SUBDIRS         = YES
EXTRACT_ALL            = YES
EXCLUDE                = usertests.c cat.c yes.c echo.c forktest.c
grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c wc.c
zombie.c
CALL_GRAPH             = YES
CALLER_GRAPH           = YES
```

- Now run “doxygen doxyconfig”
- Go to “doxygen”/html and open “firefox index.html” --> See call graphs in files -> any file

Xv6 follows monolithic kernel approach

qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
  - BIOS
  - Virtual CPU, RAM, Disk controller, Keyboard controller
  - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -
m 512
```

  - Invoked when you run “make qemu”

qemu

- **Understanding qemu command**
  - **-serial mon:stdio**
    - **the window of xv6 is also multiplexed in your normal terminal.**
    - **Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt**
  - **-drive file=fs.img,index=1,media=disk,format=raw**
    - **Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format**
  - **-smp 2**
    - **Two cores in SMP mode to be simulated**
  - **-m 512**
    - **Use 512 MB ram**

About files in XV6 code

- `cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c`
  - User programs for testing xv6
- `Makefile`
  - To compile the code
- `dot-bochsrc`
  - For running with emulator bochs

About files in XV6 code

- `bootasm.S` `entryother.S` `entry.S` `initcode.S` `swtch.S` `trapasm.S` `usys.S`
  - Kernel code written in Assembly. Total 373 lines
- `kernel.ld`
  - Instructions to Linker, for linking the kernel properly
- `README` `Notes` `LICENSE`
  - Misc files

Using Makefile

- **make qemu**
  - **Compile code and run using “qemu” emulator**
- **make xv6.pdf**
  - **Generate a PDF of xv6 code**
- **make mkfs**
  - **Create the mkfs program**
- **make clean**
  - **Remove all intermediary and final build files**

Files generated by Makefile

- `.o files`
  - Compiled from each .c file
  - No need of separate instruction in Makefile to create .o files
  - _%: %.o $(ULIB) line is sufficient to build each .o for a _xyz file
-

Files generated by Makefile

- **asm files**
  - **Each of them has an equivalent object code file or C file. For example**

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
        ./sign.pl bootblock
```

Files generated by Makefile

- `_ln, _ls, etc`
  - **Executable user programs**
  - **Compilation process is explained after few slides**

Files generated by Makefile

- `xv6.img`
  - Image of xv6 created

```
xv6.img: bootblock kernel
        dd if=/dev/zero of=xv6.img count=10000
        dd if=bootblock of=xv6.img conv=notrunc
        dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Files generated by Makefile

- **bootblock**

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
        ./sign.pl bootblock
```

Files generated by Makefile

kernel

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
        $(LD) $(LDFLAGS) -T kernel.ld -o kernel  entry.o $(OBJS) -b binary initcode entryother
        $(OBJDUMP) -S kernel > kernel.asm
        $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Files generated by Makefile

- `fs.img`
  - A disk image containing user programs and README

```
fs.img: mkfs README $(UPROGS)
        ./mkfs fs.img README $(UPROGS)
```

- `.sym files`
  - Symbol tables of different programs
  - E.g. for file “kernel”

```
$(OBJDUMP) -t kernel | sed '1,/SYMBOL
TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Size of xv6 C code

- **wc *[ch] | sort -n**
  - **10595 34249 278455 total**
  - **Out of which**
    - **738 4271 33514 dot-bochsrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
  - **2849 6864 51993 total**
- So total code is 10595 – 2849 – **738 = 7008 lines**

List of commands to try (in given order)

usertests # Runs lot of tests and takes upto 10 minutes to run

stressfs # opens , reads and writes to files in parallel

ls # out put is filetyep, inode number, type

cat README

ls;ls

cat README | grep BUILD

echo hi there

echo hi there | grep hi

echo "hi there

List of commands to try (in this order)

echo README | grep Wa

echo README | grep Wa | grep ty # does not work

cat README | grep Wa | grep bl # works

ls > out # takes time!

mkdir test

cd test

ls # fails

ls ../ # works from inside test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both directories

ln cat xyz; ls

rm xyz; ls

User Libraries: Used to link user land programs

- Ulib.c
  - Strcpy, strcmp,strlen, memset, strchr, stat, atoi, memove
  - Stat uses open()
- Usys.S -> compiles into usys.o
  - Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.

    Run following command see the last 4 lines in the output

```
objdump -d usys.o
00000048 <open>:
  48:   b8 0f 00 00 00          mov    $0xf,%eax
  4d:   cd 40                   int    $0x40
  4f:   c3                      ret
```

User Libraries: Used to link user land programs

- printf.c
  - Code for printf()!
  - Interesting to read this code.
    - Uses variable number of arguments. Normal technique in C is to use va_args library, but here it uses pointer arithmetic.
  - Written using two more functions: printint() and putc() - both call write()
    - Where is code for write()?

User Libraries: Used to link user land programs

- umalloc.c
  - This is an implementation of malloc() and free()
  - Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
  - Uses sbrk() to get more memory from xv6 kernel

Understanding the build process in more details

- **Run**

  make qemu | tee make-output.txt

- **You will get all compilation commands in make-output.txt**

Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

        $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

        $(OBJDUMP) -S $@ > $*.asm

        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d;
s/ .* / /; /^$$/d' > $*.sym
```

$@ is the name of the file being generated

$^ is dependencies . i.e. $(ULIB) and %.o in this case

Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing
-O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-
pointer -fno-stack-protector -fno-pie -no-pie   -c -o
cat.o cat.c

ld -m    elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o

objdump -S _cat > cat.asm

objdump -t _cat | sed '1,/SYMBOL TABLE/d;
s/ .* / /; /^$/d' > cat.sym
```

Compiling user land programs

Mkfs is compiled like a Linux program !

```
gcc -Werror -Wall -o mkfs mkfs.c
```

How to read kernel code ?

- Understand the data structures
  - Know each global variable, typedefs, lists, arrays, etc.
  - Know the purpose of each of them
- While reading a code path, e.g. exec()
  - Try to ‘locate’ the key line of code that does major work
  - Initially (but not forever) ignore the ‘error checking’ code
- Keep summarising what you have read
  - Remembering is important !
- To understand kernel code, you should be good with concepts in OS , C, assembly, hardware

Pre-requisits for reading the code

- **Understanding of core concepts of operating systems**
  - **Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization**
- **2 approaches:**

  1) Read OS basics first, and then start reading xv6 code
  - **Good approach, but takes more time !**

  2) Read some basics, read xv6, repeat
  - **Gives a headstart, but you will always have gaps in your understanding of the code, until you are done with everything**
  - **We normally follow this approach**
- **Good knowledge of C, pointers, function pointers particularly**
  - **Data structures: doubly linked lists, queues, structures and pointers**

File Systems

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

What we are going to learn

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

What is a file?

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

File types and kernel

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

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

What is a file?

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

File attributes

- Run

  $ ls -l

  on Linux

  To see file listing with different attributes

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

Access methods

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

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

Disk

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

Boom
Head
Sector
Spindle
Track
Platter
Cylinder

Device Driver

File system implementation: layering

Application programs

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

OS

Logical file system:

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

File organization module:

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

}
```

Basic File system:

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

}
```

IO Control, Device driver:

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

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

OS’s job now

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

Formatting

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

Different types of “layouts”
Single level directory

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

Different types of “layouts”
Two level directory

Path name

Can have the same file name for different user

Efficient searching

No grouping capability

Tree Structured directories

Acyclic Graph Directories

General Graph directory

Mounting of a file system: before

Mounting of a file system: after

$sudo mount */dev/sda5 /users*

Remote mounting: NFS

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

File sharing semantics

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

Implementing file systems

File system on disk

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

File system implementation: layering

File system: Layering

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

File system implementation: Different problems to be solved

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

File system implementation: Different problems to be solved

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

The hard link problem

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

The hard link problem

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

A typical file control block (inode)

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

Name is stored separately

Where?

IN data block of directory

In memory data structures

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

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

Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

At boot time

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

Directory Implementation

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

Directory Implementation

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

Disk space allocation for files

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

Contiguous Allocation of Disk Space

directory

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

Contiguous allocation

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

Linked allocation of blocks to a file

Linked allocation of blocks to a file

- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table

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

FAT: File Allocation Table

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

Indexed allocation

Indexed allocation

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

Multi level indexing

Unix UFS: combined scheme for block allocation

Free Space Management

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

Free Space Management: bit vector

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

    00111100111111000110000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap

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

Ext2 FS layout

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

Inode in ext2

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

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

Ext2 FS Layout: Entries in directory's data blocks

Ext2 FS Layout

Size of block 4k,1k, etc

Determined at time of mkfs (formatting)

Calculations done by “mkfs” like this

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

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

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

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

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

Traversal / path-name resolution

```
//resolving /a/b

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

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

Processes in xv6 code

Process Table

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

- One single global array of processes
- Protected by

  `ptable.lock`

Layout of process's VA space

xv6 schema!

different from Linux

Logical layout of memory for a process

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

Kernel mappings in user address space actual location of kernel

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

  VA 0x80000000 to PA 0x00000 and

  VA 0x8010000 to 0x100000

Kernel mappings in user address space actual location of kernel

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

Imp Concepts

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

Struct proc

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

struct proc diagram: Very imp!

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

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

Memory Layout of a user process

Memory Layout of a user process

After exec()

Note the argc, argv on stack

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

Handling Traps

Some basic steps

- Xv6.img is created by "make"
  - Contains bootsector, kernel code, kernel data
- QEMU boots using xv6.img
  - First it runs bootloader
  - Bootloader loads kernel of xv6 (from xv6.img)
  - Kernel starts running
- Kernel running..
  - Kernel calls main() of kernel (NOT a C application!) & Initializes:
    - memory management data structures
    - process data structures
    - file handling data structures
    - Multi-processors
    - Multi-processor data structures
    - Interrupt Descriptor Table
    - ...
  - Then creates init() ...
  - Init() fork-execs shell

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

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

Segmentation + Paging

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

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

Handling traps

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

Handling traps

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

Privilege level

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

Privilege level

- **Changes automatically on**

  “int” instruction

  hardware interrupt

  exeception

- **Changes back on**

  iret

- **“int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt’**
- **Xv6 uses “int 64” for actual system calls**

Interrupt Descriptor Table (IDT)

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

IDT setup
done by tvinit() function

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

The tvinit() is called during
kernel initialization

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

IDTR and IDT

IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

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

Setting IDT entries

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

Setting IDT entries

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

Setting IDT entries

Vectors.S

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

trapasm.S

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

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

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

After “int” ‘s job is done

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

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

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

alltraps:

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

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

trap()

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

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

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler
    - Timer
      - wakeup(&ticks)
    - IDE: disk interrupt
      - Ideintr()
    - KBD
      - Kbdintr()
    - COM1
      - Uatrintr()
    - If Timer
      - Call yield() -- calls sched()
    - If process was killed (how is that done?
      - Call exit()!

when trap() returns

- #Back in alltraps

```
  call trap
  addl $4, %esp

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

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

XV6 bootloader

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

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

Boot-process

- **Bootloader itself**
  - **gets loaded by the BIOS at a fixed location in memory and BIOS makes it run**
  - **Our job, as OS programmers, is to write the bootloader code**
- **Bootloader does**
  - **Pick up code of OS from a ‘known’ location and loads it in memory**
  - **Makes the OS run**
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

bootmasm.S bootmain.c: Steps

1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.
2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.
3) Reads kernel ELF file and loads it in RAM, as per instructions in ‘kernel’ ELF file
4) Sets up paging (4 MB pages)
5) Runs main() of kernel (which later on sets up 4KB pages)

bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
  - How to make sure that your bootloader is loaded at 0x7c00 ?
  - Makefile has

```
bootblock:  bootblock.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....
    ...
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o
    bootmain.o
```

Resuls in:

```
00007c00 <start>:  in bootblock.asm
```

Processor starts in real mode

- **Processor starts in real mode – works like 16 bit 8088**
- **eight 16-bit general-purpose registers,**
- **Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.**

  addr = seg << 4 + addr

Effective memory translation in the beginning
At _start in bootasm.S:

%cs=0 %ip=7c00.

So effective address = 0*16+ip = ip

bootloader

- First instruction is ‘cli’
  - disable interrupts
- So that until your code loads all hardware interrupt handlers, no interrupt will occur

Zeroing registers

```
# Zero data segment registers DS, ES, and SS.
xorw    %ax,%ax          # Set %ax to zero
movw    %ax,%ds          # -> Data Segment
movw    %ax,%es          # -> Extra Segment
movw    %ax,%ss          # -> Stack Segment
```

- zero ax and ds, es, ss
- BIOS did not put in anything perhaps

A not so necessary detail Enable 21 bit address

```
seta20.1:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1
  movb    $0xd1,%al            # 0xd1 -> port
0x64
  outb    %al,$0x64
seta20.2:
  inb     $0x64,%al            # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2
  movb    $0xdf,%al            # 0xdf -> port 0x60
  outb    %al,$0x60
```

- **Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2^20), 8086 wrapped addresses to 0.**
- **80286 introduced 21st bit of address. But older software required 20 bits only. BIOS disabled 21st bit. Some OS needed 21st Bit. So enable it.**
- **Write to Port 0x64 and 0x60 -> keyboard controller**
  - **to enable 21st bit out of address translation**
  - **Why? Before the A20, i.e. 21st bit was introduced, it belonged to keyboard controller**
  - **For more details see https://en.wikipedia.org/wiki/A20_line**

    https://en.wikipedia.org/wiki/A20_line

After this

Some instructions are run to enter protected mode

And further code runs in protected mode

Real mode Vs protected mode

- Real mode 16 bit registers
- Protected mode
  - Enables segmentation + Paging both
    - No longer seg*16+offset calculations
    - Segment registers is index into segment descriptor table. But segment:offset pairs continue
      - mov %esp, $32 # SS will be used with esp
    - More in next few slides
    - Other segment registers need to be explicitely mentioned in instructions
      - Mov FS:$200, 30
  - 32 bit registers
    - can address upto 2^{32} memory
    - Can do arithmetic in 32 bits

X86 address : protected mode address translation

Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging
Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways

X86 segmentation

Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

Figure 9.8 Paging hardware.

Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

Figure 8.23 Paging in the IA-32 architecture.

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

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

CR3

31
12 11
5 4 3 2 0
CR3
Page-Directory-Table Base Address
PCD
PWT
PWT Page-level writes transparent
PCT Page-level cache disable

CR4

31
11 10 9 8 7 6 5 4 3 2 1 0
CR4
OSXM
OSFX
PCE
PGE
MCE
PAE
PSE
DE
TSD
PVI
VME
VME Virtual-8086 mode extensions
PVI Protected-mode virtual interrupts
TSD Time stamp disable
DE Debugging extensions
PSE Page size extensions
PAE Physical-address extension
MCE Machine check enable
PGE Page-global enable
PCE Performance counter enable
OSFXSR OS FXSAVE/FXRSTOR support
OSXMM-EXCPT OS unmasked exception support

Segmentation + Paging

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

Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB
  - translate logical to linear addresses without change, so that they are always equal.
  - Segmentation is pratically off
- Once paging is enabled, the only interesting address mapping in the system will be linear to physical.
  - In xv6 paging is NOT enabled while loading kernel
  - After kernel is loaded 4 MB pages are used for a while
  - Later the kernel switches to 4 kB pages!

GDT Entry

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

asm.h

```
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register

Status flags
- CF Carry flag
- PF Parity flag
- AF Auxiliary carry flag
- ZF Zero flag
- SF Sign flag
- OF Overflow flag

Control flags
- DF Direction flag

System flags
- TF Trap flag
- IF Interrupt enable flag
- IOPL I/O privilege level
- NT Nested task
- RF Resume flag
- VM Virtual-8086 mode
- AC Alignment check
- VIF Virtual intr. flag
- VIP Virtual intr. pending
- ID ID flag

lgdt

```
lgdt    gdtdesc
...
# Bootstrap GDT
.p2align 2   # force 4 byte alignment
gdt:
  SEG_NULLASM     # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)

   # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)
# sizeof(gdt) - 1
  .long   gdt
```

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- table gdt : The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At gdtdesc we have this data

  2, <4 byte addr of gdt>

  Total 6 bytes

  - GDTR is : <address 4 byte>, <table limit 2 byte>
  - So lgdt gdtdesc loads these two values in GDTR

bootasm.S after “lgdt gdtdesc” till jump to “entry”

Base	Limit	Permissions
0	4GB	Write
0	4GB	Read, Execute
0	0	0

GDT

Still
Logical Address = Physical address!

But with GDT in picture and
Protected Mode operation

During this time,

Loading kernel from ELF into physical memory

Addresses in “kernel” file translate to same physical address!

Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0_PE) in register %cr0

```
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

CR0

PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not **WP: Write protecion on/off**

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

```
ljmp   $(SEG_KCODE<<3), $start32
```

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that ‘start32’ is the address of next instruction after ljmp.

Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

Jumping to “C” code

```
 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
 movw    %ax, %ds                # -> DS: Data Segment
 movw    %ax, %es                # -> ES: Extra Segment
 movw    %ax, %ss                # -> SS: Stack Segment
 movw    $0, %ax                 # Zero segments not ready for use
 movw    %ax, %fs                # -> FS
 movw    %ax, %gs                # -> GS

 # Set up the stack pointer and call into C.
 movl    $start, %esp
 call    bootmain
```

- **Setup Data, extra, stack segment with SEG_KDATA (=2), FS & GS (=0)**
- **Copy “$start” i.e. 7c00 to stack-ptr**
  - **It will grow from 7c00 to 0000**
- **Call bootmain() a C function**
  - **In bootmain.c**

Setup now

Logical Address = offset
Physical Address
CS, SS, etc.
Selector
3
2
1
0
DS
SS
CS
Base
Limit
Permissions
0
4GB
Write
0
4GB
Read, Execute
0
0
0
GDT

bootmain(): already in memory, as part of ‘bootblock’

- bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).
  - Why?
- The kernel is an ELF format binary
- Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000
- readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);
```

Drivers and Controllers

bootmain()

- Check if it's really ELF or not
- Next load kernel code from ELF file "kernel" into memory

```
  // Is this an ELF
executable?

  if(elf->magic != ELF_MAGIC)

    return;  // let bootasm.S
handle error
```

ELF

```
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff; // where is program header table
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;  // no. Of program header entries
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};
```

ELF

```
// Program  header
struct proghdr {
  uint type; // Loadable segment , Dynamic linking information , Interpreter information , Thread-Local Storage template , etc.
  uint off; //Offset of the segment in the file image.
  uint vaddr; //Virtual address of the segment in memory.
  uint paddr; // physical address to load this program, if PA is relevant
  uint filesz; //Size in bytes of the segment in the file image.
  uint memsz; //Size in bytes of the segment in memory. May be 0.
  uint flags;
  uint align;
};
```

Run ‘objdump -x -a kernel | head -15’ & see this

kernel: file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Code to be loaded at KERNBASE + KERNLINK

Diff between memsz & filesz, will be filled with zeroes in memory

Program Header:
LOAD **off** 0x00001000 **vaddr** 0x80100000 **paddr** 0x00100000 align 2**12
filesz 0x0000a516 **memsz** 0x000154a8 **flags** rwx
STACK **off** 0x00000000 **vaddr** 0x00000000 **paddr** 0x00000000 align 2**4
filesz 0x00000000 **memsz** 0x00000000 **flags** rwx

Stack : everything zeroes

Load code from ELF to memory

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
// Abhijit: number of program headers
for(; ph < eph; ph++){
  // Abhijit: iterate over each program header
  pa = (uchar*)ph->paddr;
  // Abhijit: the physical address to load program
  /* Abhijit: read ph->filesz bytes, into 'pa',
     from ph->off in kernel/disk */
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph-
>filesz); // Zero the reminder section*/
}
```

Jump to Entry

```
// Call the entry point from the ELF header.
// Does not return!
/* Abhijit:
 * elf->entry was set by Linker using kernel.ld
 * This is address 0x80100000 specified in
kernel.ld
 * See kernel.asm for kernel assembly code).
 */
entry = (void(*)(void))(elf->entry);
entry();
```

Figure 8.23 Paging in the IA-32 architecture.

To understand further code

Remember: 4 MB pages are possible

From entry:
Till: inside main(), before kvmalloc()

From entry:
Till: inside main(), before kvmalloc()

Even now, every Logical address = Physical address, but through Page dir

entrypgdir in main.c, is used by entry()

```
#define PTE_P        0x001   // Present
#define PTE_W        0x002   // Writeable
#define PTE_U        0x004   // User
#define PTE_PS       0x080   // Page Size
#define PDXSHIFT     22      // offset of
PDX in a linear address
```

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {

  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,

  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

This is entry page directory during entry(), beginning of kernel
Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.
This mapping restricts the kernel instructions and data to 4 Mbytes.

entry() in entry.S

```
entry:
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0
  movl $(stack + KSTACKSIZE), %esp
  mov $main, %eax
  jmp *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

More about entry()

```
movl    $(V2P_WO(entrypgdir)), %eax
movl    %eax, %cr3
```

-> Here we use physical address using V2P_WO because paging is not turned on yet

- **V2P is simple: substract 0x80000000 i.e. KERNBASE from address**

More about entry()

```
movl    %cr0, %eax
orl     $(CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

entry()

```
  movl $(stack + KSTACKSIZE), %esp

  mov $main, %eax

  jmp *%eax

.comm stack, KSTACKSIZE

# Abhijit: allocate here 'stack' of size = KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

bootmasm.S bootmain.c: Steps

1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.
2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.
3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file
4) Sets up paging (4 MB pages)
5) Runs main() of kernel

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

main() in main.c

- Initializes “free list” of page frames
  - In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
  - Just like the first processor
- Creates the first process!

- Initializes
  - LAPIC on each processor, IOAPIC
  - Disables PIC
  - “Console” hardware (the standard I/O)
  - Serial Port
  - Interrupt Descriptor Table
  - Buffer Cache
  - Files Table
  - Hard Disk (IDE)

main() in main.c

```
int

main(void) {

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

  kvmalloc();      // kernel page table
```

```
void

kinit1(void *vstart, void *vend) {

  initlock(&kmem.lock, "kmem");

  kmem.use_lock = 0;

  freerange(vstart, vend);

}
```

main() in main.c

```
void

freerange(void *vstart, void *vend)

{

  char *p;

  p =
(char*)PGROUNDUP((uint)vstart);

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

    kfree(p);

}
```

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

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

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

Actually like this in memory

Seen independently

Back to main()

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

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

Back to main()

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

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

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

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

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

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

Remidner: PDE and PTE entries

PDE

PTE

P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Before mappages()

What mappages() Does

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

main()->seginit()

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

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

After seginit()

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

Scheduler

Scheduler – in most simple terms

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

Steps in scheduling scheduling

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

```
F() {  i++; j++;}
```

- P2 was doing

```
G() { x--; y++; }
```

- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!

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

scheduler()

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

swtch

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

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

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

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

scheduler()

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

swtch(old, new)

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

  ret

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

scheduler()

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

sched()

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

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

sched() and schduler()

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

}
```

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

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

sched() and scheduler() as co-routines

- In sched()

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

- In scheduler()

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

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

To summarize

- On a timer interrupt during P1
  - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
  - trap()->yield()
  - yield()->sched()
  - sched() -> swtch(&p->context, c->scheduler()
  - Stack changes to scheduler's kernel stack.
  - Switches to location "Y" in scheduler().
-
- Now the loop in scheduler()
  - calls switchkvm()
  - Then continues to find next process (P2) to run
  - Then calls switchuvm(p): changing the page table to the P2's page tables
  - then calls swtch(&c->scheduler, p2's->context)
  - Stack changes to P2's kernel stack.
  - P2 runs the last instruction it was was in ! Where was it?
    - mycpu()->intena = intena; in sched()
    - Then returns to the one who called sched() i.e. exit/sleep, etc
    - Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code

File System Code

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

Files, Inodes, Buffers

What we already know

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

xv6 file handling code

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

Layers of xv6 file system code

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

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

Layout of xv6 file system

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

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

Layout of xv6 file system

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

#define DIRSIZ 14

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

File on disk

dinode
type
major
minor
nlink
size
address 1
.....
address 12
indirect
data
...
data
indirect block
address 1
.....
address 128
data
...
data

Let’s discuss lowest layer first

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

Normally, any upper layer can call any lower layer below

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

```
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
```

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

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

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

Let’s see buffer cache layer

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

Normally, any upper layer can call any lower layer below

Reminder: After main()->binit()

Conceptually Linked liks this

Buffers keep moving on list, as LRU

struct bcache

struct buf

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

buffer cache:
static struct buf* bget(uint dev, uint blockno)

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

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

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

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

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

buffer cache:
void brelse(struct buf *b)

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

Overall in this diagram

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

File descriptor layer code

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

data structures related to “file” layer

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

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

Lock is used to protect updates to every entry in the array

Multiple processes accessing same file.

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

file layer functions

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

file layer functions

- **fileread**
  - **call readi() or piperead()**
  - **readi() later calls device-read or inode read (using bread())**
- **filewrite**
  - **call pipewrite() or writei()**
  - **writei() is called in a loop, within a transaction**

- **Why does readi() call read on the device , why not fileread() itself call device read ?**

Reading Directory Layer

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

directory entry

```
#define DIRSIZ 14

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

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

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

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

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

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

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

namex

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

races in namex()

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

Let’s see Inode Layer

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

On disk & in memory inodes

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

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

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

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

In memory inodes

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

iget and iupdate

- iget
  - searches for an existing/free inode in icache and returns pointer to one
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock

- iupdate(inode *ip)
  - read on disk block of inode
  - get on disk inode
  - modify it as specified in ‘ip’
  - modify disk block of inode
  - log_write(disk block of inode)

itrunc , iput

- iput(ip)
  - if ref is 1
    - itrunc(ip)
    - type = 0
    - iupdate(ip)
    - i->valid = 0 // free in memory
  - else
    - ref--

- itrunc(ip)
  - write all data blocks of inode to disk
    - using bfree()
  - ip->size = 0
    - Inode is freed from use
  - iupdate(ip)
  - called from iput() only when ‘ref’ becomes zero

race in iput ?

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

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

buffer and inode cache

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

allocating inode

- ialloc(dev, type)
  - Loop over all disk inodes
  - read inode (from it's block)
  - if it's free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- panic if no free inodes

- ilock
  - code must acquire ilock before using inode's data/fields
  - llock reads inode if it's already not in memory

Trouble with iput() and crashes

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

Get Inode data: bmap(ip, bn)

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

writing/reading data at a given offset in file

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

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

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

Let’s see block allocation layer

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

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

allocating & deallocating blocks on DISK

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

- bfree(devno, blockno)
  - finds the right bitmap block and clears the right bit.
  - Also calls log_write()

Let's see logging layer

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

Normally, any upper layer can call any lower layer below

Recovery

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

Recovery

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

Recovery

- **Is a critical problem!**
- **Downtime is un-desired!**
- **A attempt at the solution: log structured / journaling file systems, e.g. ext3**

Log structured file systems

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

Journaling file systems

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

log in xv6

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

log in xv6

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

log on disk

log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

log

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

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

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

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

modify buffer

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

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

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

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

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

Logging functions

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

pipes

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

- functions
  - pipealloc
  - pipeclose
  - piprea d
  - pipewrite
-

pipes

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

Further to reading system call code now

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

Threads

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

Threads

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

Threads

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

Single vs Mulththreaded process

single-threaded process

multithreaded process

A mulththreaded server

Benefits of threads

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Single vs Multicore systems

Single core : Concurrency possible

Multicore : parallel execution possible

Multicore programming

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

User vs Kernel Threads

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

User threads vs Kernel Threads

- User threads
  - User level library provides a "typedef" called threads
  - The scheduling of threads needs to be implemented in the user level library
  - Need some type of timer handling functionality at user level execution of CPU
    - OS needs to provide system calls for this
  - Kernel does not know that there are threads!

- Kernel Threads
  - Kernel implements concept of threads
  - Still, there may be a user level library, that maps kernel concept of threads to "user concept" since applications link with user level libraries
  - Kernel does scheduling!

Mulththreading models

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

Many-One Model

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

One-One Model

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

Many-Many Model

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

Two Level Model

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

Thread Libraries

Thread libraries

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

pthreads

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

Demo of pthreads code

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

Other libraries

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

Issues with threads

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

More on threads

Thread pools

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

Thread Local Storage (TLS)

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

```
static __thread int threadID;
```

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

Thread Local Storage (TLS) in pthreads

- **Functions**
  - `pthread_key_create`
  - `pthread_key_delete`
  - `pthread_setspecific`
  - `pthread_getspecific`
- **See**
  - thrd_specific.c file

Scheduler activations for threads

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

Scheduler activations for threads

Library

--

```
upcall_handler() {
  Create one more LWP and schedule threads on this;
}
th_setup(int n) {
   max_LSW = n;
   curr_LWP = 0;
   register_upcall(upcall_handler);
}
th_create(...., fn,....) {
   if(curr_LWP < max_LWP)
      create LWP;
   schedule fn on one of the LWP;
}
```

application

```
f() {
   scanf();
}
g() {
   recv();
}
h() {...}; i() {...}
main()
   th_setup(2);
   th_create(...,f,...);
   th_create(...,g,...);
   th_create(...,h,...);
   th_create(...,i,...);
}
```

Kernel

```
register_upcall(function f) {

   proc->upcall = f;

}

sys_write() {

// before calling sleep() going to block

myproc()->upcall();  // tricky!

}
```

Issues with threads

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

Issues with threads

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

Issues with threads

- **Example NetBSD**
  - **"An Implementation of Scheduler Activations on the NetBSD Operating System"**
    - **https://web.mit.edu/nathanw/www/usenix/freenix-sa/freenix-sa.html**

Linux threads

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

Linux threads

- clone() takes options to determine sharing on process create
- struct task_struct points to process data structures (shared or unique depending on clone options)
- fork() is a wrapper on top of clone()
  - Use ‘strace’ to see this.

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

Issues in implementing threads project

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

Issues in implementing threads project

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

Signals

Signals

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

Signals

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

Demo

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

Signal handling by OS

Process 12323 {

signal(19, abcd);

}

OS: sys_signal {

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

}

Process P1 {

kill (12323, 19) ;

}

OS: sys_kill {

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

}

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

Threads and Signals

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

Synchronization

My formulation

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

Race problem

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

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

Race problem

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

Race problem

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

  mov c, r1

  add r1, 1

  mov r1, c

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

  thread1: mov c, r1

  thread2: mov c, r1

  thread1: add r1, 1

  thread1: mov r1, c

  thread2: add r1, 1

  thread2: mov r1, c

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

Races: reasons

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

Critical Section problem

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

Figure 6.1 General structure of a typical process P_i.

Critical Section Problem

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

Expected solution characteristics

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

suggested solution - 1

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

- What's wrong here?
- Assumes that

  while(flag ==) ; flag = 0

  will be atomic

suggested solution - 2

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

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

Peterson's solution

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

  int turn;

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

Peterson's solution

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

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

Hardware solution – the one actually implemented

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

Solution using test-and-set

```
lock = false; //global

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

Definition:

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

Solution using swap

```
lock = false; //global

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

Spinlock

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

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

Bounded wait M.E. with T&S

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

sleep-locks

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

Sleep locks/mutexes

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

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

Some thumb-rules of spinlocks

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

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

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

Locks in xv6 code

struct spinlock

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

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

spinlocks in xv6 code

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

Spinlock in xv6

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

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

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

spinlocks

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

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

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

spinlocks

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

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

spinlocks

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

sleeplocks

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

sleep()

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

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

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

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

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

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

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

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

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

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

Wakeup()

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


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

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

sleeplock

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

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

Sleeplock acquire and release

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

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

Where are sleeplocks used?

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

Sleeplocks issues

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

<u>Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.</u>

More needs of synchronization

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

Semaphore

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

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

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

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

Semaphore for synchronization

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

  Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

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

Semaphore implementation

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

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

Semaphore implementation? - 1

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

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

- suppose 2 processes trying wait.

val = 1;

Th1: spinlock Th2: spinlock-waits

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

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

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

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

circular wait. Deadlock.

None of them will proceed.

Semaphore implementation? - 2

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

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

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

Spinlocks are not good for a long wait.

Semaphore implementation? - 3, idea

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

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

Semaphore implementation? - 3a

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

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

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

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

Semaphore implementation? - 3b

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

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

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

Semaphore implementation? - 3c

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

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

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

Semaphore Implementation

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

Semaphore in Linux

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};
static noinline void __sched
__down(struct semaphore *sem)
{
    __down_common(sem,
TASK_UNINTERRUPTIBLE,
MAX_SCHEDULE_TIMEOUT);
}
```

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

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

Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem-
>wait_list);
    waiter.task = task;
    waiter.up = false;
    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
    ....
}
```

Different uses of semaphores

For mutual exclusion

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

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

Event-wait

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

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

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

Control countable resources

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

Drawbacks of semaphores

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

Deadlocks

Deadlock

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

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

Example of deadlock

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

Non-deadlock, but similar situations

- **Starvation – indefinite blocking**
  - **A process may never be removed from the semaphore queue in which it is suspended**
- **Priority Inversion**
  - **Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task**
  - **Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest**

Livelock

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

Livelock example

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

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

More on deadlocks

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

System model for understanding deadlocks

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

Deadlock characterisation

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

Resource Allocation Graph Example

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

Resource Allocation Graph with a Deadlock

Graph with a Cycle But no Deadlock

Basic Facts

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

Methods for Handling Deadlocks

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

  1) Deadlock prevention

  2) Deadlock avoidance

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

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

(1) Deadlock Prevention

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

(1) Deadlock Prevention (Cont.)

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

(1) Deadlock prevention: Circular Wait

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

  first_mutex = 1

  second_mutex = 5

  code for thread_two could not be written as follows:

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

   pthread_exit(0);
}

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

   pthread_exit(0);
}
```

(1) Preventing deadlock: cyclic wait

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

(1) Prevention in Xv6: Lock Ordering

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

(2) Deadlock avoidance

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

(2) Deadlock avoidance

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

(3) Deadlock detection and recovery

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

“Condition” Synchronization Tool

What is condition variable?

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

```
Struct condition {
   Proc *next
   Proc *prev
   Spinlock *lock
}
```

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

Code for condition variables

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

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

Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

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

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

Classical Synchronization Problems

Bounded-Buffer Problem

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

Bounded-Buffer Problem: solution with semaphores

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

Bounded-buffer problem

The structure of the producer process

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

The structure of the Consumer process

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

Bounded buffer problem

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

Readers-Writers problem

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

Readers-Writers problem

The structure of a writer process

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

The structure of a reader process

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

Readers-Writers Problem Variations

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

Reader-write lock

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

Code for reader-writer locks

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

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

Code for reader-writer locks

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

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

Code for reader-writer locks

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

Try writing code for downgrade and upgrade

Try writing a reader-writer lock using semaphores!

Dining-Philosophers Problem

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

Dining philosophers: One solution

The structure of Philosopher i:

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

What is the problem with this algorithm?

Dining philosophers: Possible approaches

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

Other solutions to dining philosopher's problem

- **Using higher level synchronization primitives like 'monitors'**
-

Practical Problems

Lost Wakeup problem

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

Lost Wakeup problem

Figure 7-6. The lost wakeup problem.

Thundering herd problem

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

Case Studies

Linux Synchronization

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

Linux Synchronization

- **Atomic variables**

  atomic_t is the type for atomic integer

- **Consider the variables**

  atomic_t counter;

  int value;

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

Pthreads synchronization

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

Synchronization issues in xv6 kernel

Difference approaches

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

Three types of code

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

Interrupts enabling/disablilng in xv6

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

Memory ordering

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

- Consider this

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

Lost Wakeup?

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

Code of sleep()

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

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

Exercise question : 1

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

Suppose the special case were eliminated by replacing

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

with

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

Doing this would break sleep. How?

`

bget() problem

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

iget() and ilock()

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

Interesting cases in namex()

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

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

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

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

One process acquires, another releases!

Giving up CPU

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

Interesting race if ptable.lock is not held

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

Homework

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

Revision of Memory management concepts

Revision

- Memory layout of C program
- Address binding times: compile, link, load
- MMU Schemes
  - No MMU, Base+Relocation, Multiple Base+Relocation = Segmentation, Paging
    - Hierarchical paging
    - TLB,
  - External and Internal fragmentation
- X86 memory management
- Xv6 paging + segmentation

More on Linking, Loading

More on Linking and Loading

- Static Linking: All object code combined at link time and a big object code file is created
- Static Loading: All the code is loaded in memory at the time of exec()
- Problems
  - Static linking: Big executable files,
  - Static loading: need to load functions even if they do not execute
- Solution: Dynamic Linking and Dynamic Loading

Dynamic Linking

- Linker is normally invoked as a part of compilation process
  - Links
    - function code to function calls
    - references to global variables with “extern” declarations
- Dynamic Linker
  - Does not combine function code with the object code file
  - Instead introduces a “stub” code that is indirect reference to actual code
  - At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)

Dynamic Linking on Linux

```
#include <stdio.h>
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d %d\n", a, b);
return 0;
}
```

PLT: Procedure Linkage Table
used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
    11d4:        callq  1080 <printf@plt>
```

Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
    1080:       endbr64
    1084:       bnd jmpq *0x2f3d(%rip)        # 3fc8
<printf@GLIBC_2.2.5>
    108b:       nopl   0x0(%rax,%rax,1)
```

Dynamic Loading

- Loader
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- Dynamic Loading
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

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 ?

Virtual Memory

Introduction

- Virtual memory != Virtual address
  - Virtual address is address issued by CPU's execution unit, later converted by MMU to physical address
  - Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

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

All parts of array a[] not accessed

Function f() may not be called

Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used
  - Error code, unusual routines, large data structures are rarely used
- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program
  - One Program no longer constrained by limits of physical memory
  - One Program and collection of programs could be larger than physical memory

What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger Than Physical Memory

Virtual Address space

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

Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

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

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

Demand Paging

Demand Paging

- Load a “page” to memory when it’s neded (on demand)
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
-

Demand Paging

- Options:
  - Load entire process at load time : achieves little
  - Load some pages at load time: good
  - Load no pages at load time: pure demand paging

New meaning for valid/invalid bits in page table

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

- 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**

Page Table With Some pages Not In memory

Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!
  - Change of stack to kernel stack
  - Saving the context of process
  - Switching to kernel code

Important (not all) steps on a Page fault

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

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

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

2) Get empty frame (this may be complicated, may need evicting a frame if no free frames available!)

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

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

6) Restart the instruction that caused the page fault

Issues with page fault handling

- 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. Less response time ?**
- 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

Problem with Instruction restart

- A critical Problem
- Consider an instruction that could access several different locations
  - `movarray 0x100, 0x200, 20`
  - # copy 20 bytes from address 0x100 to address 0x200
  - `movarray 0x100, 0x110, 20`
  - # what to do in this case?

Handling A Page Fault

Page fault handling (detailed)

1) Trap to the operating system

2) Default trap handling():

   Save the process registers and process state

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

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

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

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

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

Page fault handling (detailed)

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

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

8) Default interrupt handling():

- Save the registers and process state for the other user
- Determine that the interrupt was from the disk

9) Disk interrupt handler():

- Figure out that the interrupt was for our waiting process
- Make the process runnable

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

- Kernel restores the page table of the process, marks entry as “v”
- Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate 0 <= p <= 1

if p = 0 no page faults

if p = 1, every reference is a fault

Effective (memory) Access Time (EAT)

EAT = (1 – p) * memory access time +

p * (page fault overhead // Kernel code execution time

+ swap page out // time to write an occupied frame to disk

+ swap page in // time to read data from disk into free frame

+ restart overhead) // time to reset process context, restart it

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

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

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p

p < .0000025

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

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

```
        scanf("%s", cmd);
        if(strcmp(cmd, "exit") == 0)
            return 0;
        pid = fork(); // A->B
        if(pid == 0) {
            ret = execl(cmd, cmd, NULL);
            if(ret == -1) {
                perror("execution
failed");
                exit(errno);
            }
        } else {
            wait(0);
        }
```

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

An optimization: Copy on write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Before Process 1 Modifies Page C

After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store
  - For stack, heap pages: on swap partition
  - For code, shared library? : swap partition or the actual executable file on the file-system?
  - If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system
- Is the page table itself pagable?
  - If no, good
  - If Yes, then there can be page faults in accessing the page tables themselves! More complicated!
- Is the kernel code pagable?
  - If no, good
  - If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.

Page replacement

Review

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

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created (kinit1, knit2 in xv6)
- Frames are used for allocating memory to a process
  - But may also be used for managing kernel's own data structures also (in xv6, all kernel data is statically allocated)
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”
  - More multiprogramming -> overallocation of frames
  - Also in demand from the kernel, I/O buffers, etc
  - How much to allocate to each process? How many processes to allow?
- Page replacement – find some page(frame) in memory, but not really in use, page it out
  - Questions : terminate process? Page out process? replace the page?
  - For performance, need an algorithm which will result in minimum number of page faults
- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
  - Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

1) Find the location of the desired page on disk

2) Find a free frame:

- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty

3) Bring the desired page into the free frame; update the page table of process and global frame table/list

4) Continue the process by restarting the instruction that caused the trap

- Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

Two problems to solve

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

Evaluating algorithm: Reference string

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

An Expectation

More page Frames Means less faults

FIFO Algorithm

reference string

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

page frames

15 page faults

FIFO Algorithm

1	5	1 4 5
2	3	2 1 3
3	4	3 2 4

9 PF

1	4	1 5 4
2	5	2 1 5
3	2	3 2
4	3	4 3

10 PF

- **Belady's Anomaly**
- Adding more frames can cause more page faults!
  - Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

FIFO Algorithm: Balady's anamoly

Optimal Algorithm

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

Optimal page replacement

reference string

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

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

page frames

Least Recently Used: an approximation of optimal

reference string

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

page frames

Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page
12 faults – better than FIFO but worse than OPT
Generally good algorithm and frequently used
But how to implement?

LRU: Counter implementation

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

LRU: Stack implementation

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

Use of a Stack to Record The Most Recent Page Ref

Stack algorithms

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

LRU: Approximation algorithms

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

LRU: Approximation algorithms

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

Second-Chance (clock) Page-Replacement Algorithm

(a)

(b)

Counting Algorithms

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

Page buffering algorithms

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

Major and Minor page faults

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

```
$ ps -eo min_flt,maj_flt,cmd
```

Special rules for special applications

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

Allocation of frames

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

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

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

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

Priority Allocation of frames

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

Global Vs Local allocation

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

Virtual Memory – Remaining topics

Agenda

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

Thrashing

Thrashing

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

Thrashing

Locality In A Memory-Reference Pattern

Demand paging and thrashing

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

Working set model

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

Working set model

- D = Σ WSS $_i$ ≡ total demand frames
  - Approximation of locality
- if D > m (total available frames) ⇒ Thrashing
- Policy if D > m, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy (to memory) and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If ac

Working Sets and Page Fault Rates

Memory Mapped Files

Memory-Mapped Files

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

Memory-Mapped Files

Use MAP_SHARED flag

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

Memory-Mapped Files

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

Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
  - I.e. for device I/O
-

Buddy Allocator

Buddy Allocator

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

Buddy Allocator

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

Slab Allocator

Slab Allocator

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

Other considerations

Other Considerations -- Prepaging

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

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

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

Program Structure

- Program structure
- `Int[128,128] data;`
- Each row is stored in one page
- Program 1

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

128 x 128 = 16,384 page faults

- Program 2

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

128 page faults

I/O Interlock

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

VFS

Figure 15.5 Schematic view of a virtual file system.

VFS

- **Consider this**

***/dev/*sda1 is "/"**

***/dev/*sda2 is mounted on "/a/b" folder**

How does this work in kernel?

open("/a/b/c/d", O_RDONLY)

- **Consider xv6 code**
- **sys_open -> namei -> namex -> (skipelem, dirlookup, ilock)**
- **Dirlookup() of "c" in "/a/b" should return : Not the inode of "c" on */dev/*sda1 but inode of "/" on /dev/sda2**

VFS

- Object Oriented Programming in C (let's see example of this)
- Clever use of function pointers
- There is an "abstract" file system class (VFS), and there are concrete file system classes (ext2, vfat, ...)
- sys_read → fileread → readi() becomes
- sys_read → fileread → (i->-i_ops->read)()
- Inode is a generic inode
- Contains file system specific inode pointer
- And file system specific inode operations
- Fields setup during namei()

```
struct inode_operations {
        int (*readi) (int, char *, int);
        int (*writei) (int, char *, int);
        ....
}
struct inode {
    int mode,
    Int uid;
     ....
    void *inode_specific;
    struct inode_ops i_ops;
}
```

Efficiency and Performance

(and the risks created

while trying to achieve it!)

Efficiency

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

Performance

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

Page cache

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

I/O Without a Unified Buffer Cache

Unified buffer cache

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

I/O Using a Unified Buffer Cache

Recovery

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

Recovery

- fsck: Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup
- Faster recovery? - “log structured file system” or “journaling file system” can help

Log structured file systems

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

Journaling file systems

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

log in xv6

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

log on disk

log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

log in xv6

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

log

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

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

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

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

modify buffer

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

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

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

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

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

Logging functions

- Initlog()
- Set fields in global log.xyz variables, using FS superblock
- Recovery if needed
- Called from first forkret()
- Following three called by FS code
- begin_op(void)
- Increment log.outstanding
- end_op(void)
- Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
- Remember the specified block number in log.lh.block[] array
- Set the block to be dirty

- write_log(void)
- Called only from commit()
- Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
- Called only from end_op()
- write_log()
- Write header to disk log-header
- Copy from log blocks to actual FS blocks
- Reset and write log header again

Scheduling Algorithms

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

Credits: Slides from os-book.com

Calculations of different scheduling criteria

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

Calculations of different criteria

- CPU Utilization
  - % time spent in doing ‘useful’ work
  - What is useful work?
    - On linux
      - there is an “idle” thread, scheduled when no other task is RUNNABLE
      - Not running idle thread is productive work
      - Includes process + scheduling time + interrupts
    - On other systems?
      - Need to define
    - On xv6
      - We can say that time spent in the loop selecting a process is idle work

Calculations of different criteria

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

Calculations of different criteria

- **Turnaround time**
  - **Amount of time required for one process to complete**
  - **For every process, note down the starting and ending time, difference is TA-time**
  - **For process P1 : (Time when process ended – time when process started)**

    = Sum of time spent in (ready queue + running + waiting for I/O)
  - **One can find the average T.A. time**

Calculations of different criteria

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

Scheduling Criteria

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

Challenges in implementing the scheduling algorithms

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

GANTT chart

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

Scheduling Algorithms

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

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

The Gantt Chart for the schedule is:

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

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

Non Pre-emptive algorithm

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P2 , P3 , P1

The Gantt chart for the schedule is:

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

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

Much better than previous case

FCFS: Convoy effect

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

FCFS: further evaluation

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

Shortest-Job-First (SJF) Scheduling

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

Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF scheduling chart

Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- **Not possible to implement SJF as can't know "next" CPU burst. Can only estimate the length – should be similar to the previous one**
  - **Then pick process with shortest predicted next CPU burst**
- **Can be done by using the length of previous CPU bursts, using exponential averaging**

1. t_n = actual length of nth CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

- **Commonly, α set to ½**
- **Preemptive version called shortest-remaining-time-first**

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$

    $+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$

    $+ (1 - \alpha)^{n+1} \tau_0$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst

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

CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

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

Round Robin (RR) Scheduling

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

Round Robin (RR) Scheduling

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

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

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

q should be large compared to context switch time

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

Time Quantum and Context Switch Time

quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than quantum

Priority Scheduling

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

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

Priority Scheduling with Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

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

Gantt Chart with 2 ms time quantum

Multilevel Queue

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

Multilevel Queue

Implementing multilevel queue

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

Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

- **Three queues:**
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- **Scheduling rules**
  - Serve all processes in Q0 first
  - Only when Q_0 is empty, serve processes in Q_1
  - Only when Q_0 and Q_1 are empty, serve processes in Q_2

Example of Multilevel Feedback Queue

- **Scheduling**
  - **A new job enters queue Q_0**
    - **When it gains CPU, job receives 8 milliseconds**
    - **If it does not finish in 8 milliseconds, job is moved to queue Q_1**
  - **At Q_1 job receives 16 additional milliseconds**
    - **If it still does not complete, it is preempted and moved to queue Q_2**
  - **To prevent starvation, move a process from lower-priority queue to higher priority queue after it has waited for too long**

Thread Scheduling

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

Pthread Scheduling

- **PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling**
- **PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling**
- **Linux and macOS only allow PTHREAD_SCOPE_SYSTEM**
- **Let’s see a Demo using a program**

Multiple-Processor Scheduling – Load Balancing

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

End

Device files
Volume Manager

Device Files

- See
  - $ ls -l */dev*
  - “c” character special device files
  - ‘b’ block device files
- Device file represents a hardware device
- Open() and then read(), write() on device files will read/write from the device (if supported)

Device Files

- Major Number and Minor Number
  - The size field in inode is reused as major-minor number field for device files
  - Major number: type of device (identifies the device driver)
  - Minor number: device number of that type (tells the device driver, which device)
- Xv6
  - The sys_read()->file_read()->readi() will redirect the read operation to

```
return devsw[ip->major].read(ip, dst, n);
 where
 struct devsw {
   int (*read)(struct inode*, char*, int);
   int (*write)(struct inode*, char*, int);
 };
```

Block Device Files

- For “block” devices
  - read() ,write() happen in multiples of “block”
- E.g.
  - */dev/*sda1
  - */dev/*hda1
  - */dev/nvmep01*

Issues with disk partitions

- Use of disk partitions

```
$ fdisk /dev/sdb  # create partitions
$ mkfs -t ext2 /dev/sdb1  # format
$ mount -t ext2 /dev/sdb1  /a/b # mounted on /a/b
```

- Fixed size
- Cant' easily grow them
  - Without risk of losing file system
- Can't easily shrink them

Logical Volume Mananger (LVM)

- A layer (indirection) between physical partitions (physical volumes) and file system
  - Enables easy grouping , re-grouping, extending, shrinking, ...
- Logical Volume Group:
  - Parallel to a physical disk (but extendible)
- Logical Volume
  - Parallel to a physical partision (but extendible!)

File system
File system
Logical Volume
Logical Volume
Volume Group
Physical Volume
Physical Volume
Physical Volume

LVM
File Systems
Logical Volumes
Volume Group
Physical Volumes
Partitions
Hard Drives
/mnt/data (ReiserFS)
/mnt/backup (EXT3)
Unallocated Space
/dev/datavg/data
/dev/datavg/backup
datavg
/dev/sda1
/dev/sdb1
/dev/sdc1
/dev/sdd1
/dev/sde1
/dev/sda1
/dev/sdb1
/dev/sdc1
/dev/sdd1
/dev/sde1
/dev/sda
/dev/sdb
/dev/sdc
/dev/sdd
/dev/sde

LVM
File Systems
Sistema de arquivos
/home
(ext4)
/data
(xfs)
Logival Volume (LV)
Volume Lógico
/dev/vgroup/lv_home
/dev/vgroup/lv_data
Volume Groups (VG)
Grupos de volumes
vgroup
www.linuxnaweb.com
Physical Volumes (PV)
Volumes físicos
/dev/sdb1
/dev/sdb2
/dev/sdc1
/dev/sdc2
Partitions
Partições
/dev/sdb1
8e LVM
/dev/sdb2
8e LVM
/dev/sdc1
8e LVM
/dev/sdc2
8e LVM
Physical Drives
Drivers físicos
/dev/sdb
/dev/sdc

The point of this document:

I'm actually not so sure. I think the point is to explain why xv6 does stuff, in broad strokes, as a complement to the explanations of the actual code of xv6.

Important-ish note:

Often, I'll refer to things that haven't been explained yet. Just keep in mind that not everything is explained at first, and that's okay.

These explanations are not full. They may not even be accurate. I'm just jotting these down during class, hoping it more or less gives a picture of what's going on.

Anyways, here we go.

Kernel

When I say "kernel", I mean the operating system. This is the mother of all programs that hogs the computer to itself and manages all the other programs and how they access the computer resources (such as memory, registers, time (that is, who runs when), etc.).

The kernel also keeps control over the "mode" bit (in the `%cs` register), which marks whether current commands are running in user-mode (and therefore can't do all kinds of privileged stuff) or kernel-mode (and therefore can do whatever it wants).

Boot

The processor (that is, the actual computer) does not know what operating system is installed, what it looks like, or how large it is.

So how can the processor load xv6 into the memory?

The processor instruction pointer `%eip` register points - by default - to a certain memory place in the ROM, which contains a simple program that:

1. Copies the very first block (512 bytes. AKA the boot block) from the disk to the memory
2. Sets the instruction pointer `%eip` to point to the beginning of the newly-copies data

So what?

Every operating system needs to make sure that its first 512 bytes are a small program (it has to be small; it's only 512 bytes!) that loads the rest of the operating system to the memory and sets the PC to whatever place it needs to be in. If 512 bytes aren't enough for this, the program can actually call a slightly larger program that can load and set up more.

In short:

1. `%eip` points to hard-coded ROM code
2. ROM code loads beginning of OS code

3. Beginning of OS code loads the rest of the code
4. Now hear the word of the Lord!

Processes

A process is the running of a program, including the program's state and data. The state includes such things as:

- Memory the program occupied
- Memory contents
- Register values
- Files
- Kernel structures

This is managed by the kernel. The kernel has a simple data structure for each process, organized in some list. The kernel juggles between the processes, using a context switch, which:

1. Saves state of old process to memory
2. Loads state of new process from memory

Context switch can be preemptive (i.e. the kernel decides "next guy's turn!", using the scheduler (there'll be a whole lot of talk about this scheduler guy later on)) or non-preemptive (i.e. the hardware itself decides). In non-preemptive, it is asked of the programmers to make calls to the kernel once in a while, in order to let the kernel choose to let the next guy run.

Xv6 is preemptive.

Processes are created by the kernel, after another process asks it to. Therefore, the kernel needs to run the first process itself, in order to create someone who will ask for new processes to be created.

`fork()`

Every process has a process ID (or pid for short).

A process can call the kernel to do `fork()`, which creates a new process, which is entirely identical to the parent process (registers, memory and everything). The only differences are:

- The pid
- The value returned from fork:
- In the new process - 0
- In the parent process - the new pid
- In case of failure - some negative error code

`exec()`

`Fork()` creates a new process, and leaves the parent running. `Exec()`, on the other hand, replaces the process's program with a new program. It's still the same process, but with new code (and variables, stack, etc.). Registers, pid, etc. remain the same.

It is common practice to have the child of `fork` call `exec` after making sure it is the child. So why not just make a single function that does both fork and exec together? The exec system call replaces the entire memory of the parent process except for the open files. This allows a parent process to decide what the stdin, stdout and stderr for the child process. This trick is used in the /init process in xv6 and the sh to pipe outputs to other processes.

Process termination

Trigger warning: sad stuff ahead. And also zombies.

A process will be terminated if (and only if) one of the following happens:

1. The process invokes `exit()`
2. Some other process invokes `kill()` with its pid
3. The process generates some exception

Note that `kill` does not actually terminate the process. What it does is leave a mark of "You need to kill yourself" on the process, and it'll be the process itself that commits suicide (after it starts running again, when the scheduler loads it).

Note also that not any process can `kill` any other process. The kernel makes sure of that.

Once `kill`ed, the process's resources (memory, etc.) are not released yet, until its parent (that is, the process which called for its creation) allows this to happen. A process that `kill`ed itself but whose parent did not acknowledge this is called a zombie.

In order to a parent to "acknowledge" its child's termination, it needs to call `wait()`. When it calls `wait()`, it will not continue until one of its children exits, and then it will continue. If there are a few children, the parent will need to call `wait` once for each child process.

`Wait` returns the pid of the exited process.

What happens if a parent process `exit`s before its children? Its children become orphans, and the The First Process (whose pid is 1) will make them His children.

System calls

Many commands will only run if the "mode" bit is set to kernel-mode. However, all processes run on user-mode only; xv6 makes sure of that.

In order to run a privileged command, a process must ask the kernel (which runs in kernel-mode, of course) to carry out the command. This "asking" is called a system call.

Here's an example of system call on Linux with an x86 processor:

```
movl flags(%esp), %ecx
lea name(%esp), %ebx
movl $5,%eax    ; loads the value 5 into eax register, which is the command "open" in Linux
int $128    ; invokes system call. Always 128!
; eax should now contain success code
```

The kernel has a vector with a bunch of pointers to functions (Yay, pointers!).

Addresses

This one's a biggie. Hold on to your seatbelts, kids.

Programs refer to memory addresses. They do this when they refer to a variable (that's right; once code's compiled, all mentions of the variable are turned into the variable's address). They do this in every `if` or loop. When the good old `%eip` register holds the address of the next instruction, it's referring to a memory address.

There are two issues that arise from this:

- The compiled code does not know where in the memory the program is going to be, and therefore these addresses must be relative to the program's actual address. (This is a problem with loops and ifs, not with the `%eip`.)
- We'll want to make sure no evil program tries to access the memory of another program.

So, each process has to have its "own" addresses, which it thinks are the actual addresses. It has *nothing* to do with the actual RAM, just with the *addresses* that the process knows and refers to. (**Process**, not **program**; this includes the kernel.)

Behold! A sketch of what a process's addresses looks like in xv6:

Address	Who uses this
`[0xFFFF FFFF]`	Kernel
`[0xFFFF FFFE]`	Kernel
...	Kernel
`[0x8000 0000]`	**Kernel**
`[0x7FFF FFFF]`	**Process**
...	Process
`[0x0000 0000]`	Process

In order to pull off this trick, we use a hardware piece called the Address Translation Unit, which actually isn't officially called that.
Its real name is the MMU (Memory Management Unit), for some reason.

The MMU is actually comprised of two units:

1. Segmentation Unit
2. Paging Unit.

The MMU sits on the address bus between then CPU and the memory, and decides which actual addresses the CPU accesses when it reads and writes. Each of the smaller units (segmenataion and paging) can be turned on or off. Note that Paging can only be turned on if Segmentation is on. (We actually won't really use the Segmentation Unit in xv6. LATER EDIT:

This is a lie. A LIE! We actually use it for all kinds of unwholesome tricks.)

Addresses coming from CPU are called **virtual/logical addresses**. Once through the Segmentation Unit, they're called **linear addresses**. Once through the Paging Unit, they're **physical addresses**.
In short: CPU [virtual] -> Segmentation [linear] -> Paging [physical] -> RAM

Here's a bunch of 16-bit registers that are used by the Segmentation Unit:

- `%cs` - **C**ode. This guy actually messes with our `%eip` register (that's the guy who points to the next command!).
- `%ds` - **D**ata. By default, messes with all registers except `%eip`, `%esp` and `%ebp`. (In assembly, we can override the choice of messing register.)
- `%ss` - **S**tack. By default, messes with `%esp` and `%ebp` registers.
- `%es`
- `%fs`
- `%gs`

The address `%eip` points to after going through `%cs` is written as `CS:EIP`.

In the Segmentation Unit there is a register called `%GDTR`. The kernel uses this to store the address of a table called GDT (Global Descriptor Table). Every row is 8 bytes, and row #0 is not used. It can have up to 8192 rows, but no more.

The first two parts of each row are **Base** and **Limit**.

When the Segmentation Unit receives and address, the CPU gives it an *index*. This index is written in one of the segmentation registers (`%cs`, `%ds`, ... `%gs`), thusly:

- Bits 0-1: permissions
- Bit 2: "use GDT or LDT?" (Let's pretend this doesn't exist, because it does not interest us at all.)
- Bits 4-15: The index

The Segmentation Unit receives a logical address and uses the index to look at the GDT. Then:

- If the logical address is greater than the `Limit`, crash!
- Else, the Segmentation adds the `Base` to the logical address, and out comes a linear address.

Note: In the past, the Segmentation Unit was used in order to make sure different processes had their own memory. For example, they could do this by making sure that each time the kernel changes a process, its segmentation registers would all point to an index that "belongs" to that process (and each row in the GDT would contain appropriate data). Another way this could be done would be by maintaining a seperate GDT for each process. Or maintaining a single row in the GDT and updating it each time we switch a process. There is no single correct way.

Note that all the addresses used by each process must be consecutive (along the physical memory).

In xv6, we don't want any of this.

Therefore, we will make sure that the GDT `Limit` is set to max possible, and the `Base` is set to 0.

In order to allow consecutive virtual addresses to be mapped to different areas in the physical memory, we use **paging**.

In the Paging Unit, there is a register named `%CR3` (Control Register 3), which points to the **physical** address of the Page Table (kinda like GDT). A row in the Page Table has a whole bunch of data, such as page address, "is valid", and some other friendly guys.

When the Paging Unit receives a linear address, it acts thusly:

- The left-side bits are used as an *index* (AKA "page number") for the Page Table. (There are 20 of these.)
- In the matching row, if the "is valid" bit = 0, crash!
- (There are also "permission" bits, but let's ignore them for now.)
- Those "page number" bits from the linear address are replaced by the actual page in our row (which is already *part* of the actual real live physical address)
- The page is "glued" to the right-side bits of the linear address (you know, those that aren't the page number. There are 12 of these.)
- Voila! We have in our hands a physical address.

Note that each page can be in a totally different place in the physical memory. The pages can be scattered (in page-sized chunks) all along the RAM.
Also note: Hardware demands that the 12 right-most bits of `%CR3` be 0. (If not, the hardware'll zero 'em itself.)

Uh oh:
Each row in the Page Table takes up 4KB (that's 12 bits).
The Page Table has 1024 rows.
4KB * 1024 = 4MB. That's 4 whole consecutive MBs. That's quite a large area in the memory, which kind of defeats the whole purpose of the Page Table. Well, not the *whole* purpose, but definitely some of it.

The solution: The Page Table gets its very own Page Table!

- First (small) page table contains - in each row - the (physical) address of another (small) page table.
- Each of the (small) 2nd-level page tables (which are scattered) contain actual page addresses.
- So: instead of 20 bits for single index, we have 10 bits for 1st-level table index and 10 bits for 2nd-level table index.

Thus, a virtual address can be broken down like so:

`[i0][i1][offset]`

- `i0` is the index of the First Table
- `i1` is the index of a second table
- `offset` is the offset

Addresses - double mapping

Every single physical address is mapped by the kernel to virtual address by adding KERNBASE to it.

When a process is given a virtual address, this new virtual address is *in addition* to the kernel mapping.

So when we want to get the physical address of a virtual address:

1. If the user-code is asking, it needs to access the page tables.

2. Is the kernel is asking, it can simply subtract KERNBASE from the address.

Likewise, the kernel can determine *its* virtual address of *any* physical address by adding KERNBASE.

KERNBASE = 0x80000000.

1217 main

In the beginning, we know that from `[0x0000 0000]` till `[0x0009 FFFF]` there are 640KB RAM.
From `[0x000A 0000]` till `[0x000F FFFF]` is the "I/O area" (384KB), which contains ROM and stuff we must not use (it belongs to the hardware).
From `[0x0010 0000]` (1MB) till `[0xFF00 0000]` (4GB - 1MB in total) there is, once again, usable RAM.
After that comes "I/O area 2".

(Why the 640KB, the break, and then the rest? Because in the olden days they thought no one would ever use more than 640KB.)

Address	Who uses this
`[Who cares]`	I/O
...	I/O
`[0xFF00 0001]`	I/O
`[0xFF00 0000]`	Usable RAM
...	Usable RAM
`[0x0010 0000]`	Usable RAM
`[0x000F FFFF]`	I/O
...	I/O
`[0x000A 0000]`	I/O
`[0x0009 FFFF]`	Usable RAM
...	Usable RAM
`[0x0000 0000]`	Usable RAM

Remember Mr. Boot? He loads xv6 to `[0x0010 0000]`.
By the time xv6 loads (and starts running `main`), we have the following setup:

1. `%esp` register is pointing to a stack with 4KB, for xv6's use. (That's not a lot.)

2. Segmentation Unit is ready, with a (temporaray) GDT that does no damage (first row inaccessible, and another two with 0 `Base` and max `Limit`.

3. Paging Unit is ready, with a temporary page table. The paging table works thusly:

    - Addresses from `[0x800- ----]` till `[0x803- ----]` are mapped to `[0x000- ----]` through `[0x003- ----]` repectively. (That is, the left-most bit is simply zeroed.)
    - ALL the the above addresses have 1000000000b as their 10 left-most bits, so our 1st-level page table has row 512 (that's 1000000000b) as "valid", and all the rest marked as "not valid".
    - Row 512 points to a single 2nd-level table.
    - The 2nd-level table uses all 1024 of its rows (that's exactly the next 10 bits of our virtual addresses), so they're all marked as valid.

- Each row in 2nd-level table contains a value which - coincidentally - happens to be the exact same number as the row index.

Let's look at some sample virual address just to see how it all works:
`[0x8024 56AB]` -> `[1000 0000 0010 0100 0101 0110 1010 1011]` -> `[1000 0000 00` (that's row 512) `10 0100 0101` (that's row 581) `0110 1010 1011` (and that's the offset) `]` -> row 581 in the 2nd-level table will turn `[1000 0000 0010 0100 0101...]` to `[0000 0000 0010 0100 0101...]`, which is exactly according to the mapping rule we mentioned a few lines ago.

Available pages (free!)

Processes need pages to be mapped to. Obviously, we want to make sure that we keep track of which page are available. The free pages are managed by a **linked list**. This list is held by a global variable named `kmem`. Each item is a `run` struct, which contains only a pointer to the next guy.

`kmem` also has a lock, which makes sure (we'll learn later how) that different processes don't work on the memory in the same time and mess it up. (An alternative would be to give each processor its own pages. However, that could cause some processors to run out of memory while another has spare. (There are ways to work around it.))

`main` calls `kinit1` and `kinit2`, which call `freerange`, which calls `kfree`.
In `kfree`, we perform the following 3 sanity checks:

- We're not in the middle of some page
- We're not trying to free part of the kernel
- We're not pushing beyond the edge of the physical memory

Building a page table

Let's examine what needs to be done (not necessarily in xv6) in order to make our very own paging table.

Our table should be able to "translate" virtual address *va* to physical address *vp* according to some rule.
Note that we'll be using `v2p`, which is a hard-coded mother-function that translates **virtual** addresses to **physical** by subtracting `KERNBASE` (which equals `0x8000 0000`) from the virtual addresses.

Step 1: Call `kalloc` and get a page for our First Table. (Save (virtual!) address of new table in `pgdir` variable)

Step 2: Call `memset` to clear entire page (thus marking all rows as invalid).

Step 3: Do the following for **every single *va*** we want to map:

- **Step 3.1**: Create and clear subtable, and save address in `pgtab` (similar to what we did in steps 1 and 2). (SEE NOTE AFTER THIS LIST)

- **Step 3.2**: Figure out i0 (index in `pgdir`) (using *va* & `v2p` function), write `pgtab` there, mark as valid.

- **Step 3.3**: Figure out i1 (index in `pgtab`) (using *va* & `v2p` function), write *pa* there, mark as valid.

Step 4: Set `%CR3` to point at `pgdir`, using `v2p`.

THE NOTE MENTIONED EARLIER: After we already have some subtables, we only need to create new subtables if the requested subtable does not exist yet.
How do we know whether it exists already? Simply by looking at the current i0 and seeing whether it's already marked as valid.

ANOTHER NOTE: What if two different *va->vp* rules clash? xv6 will crash (deliberately).

For more details about how the kernel builds its mapping tables, please refer to [xv6 Code Explained.md] (`kvmalloc`).

Moar GDT stuff!

Remember how we said we'd make sure GDT has `base=0` (so it won't alter addresses) and `limit=max` (so it won't interfere)?
Well, it turns out the hardware still uses the GDT to check various user-mode/kernel-mode stuff.

We need four rows in the GDT:

1. Kernel-mode, can only execute and read
2. Kernel-mode, can write
3. User-mode, can only execute and write
4. User-mode, can write.

There is a macro named `SEG`, which helps us build all these annoying bits.

Okay, that's enough of this.
Don't pretend you understand, because you don't and it doesn't matter.

Per-CPU variables

We've got an array of CPU data, `cpus`.

We can access specific CPU stuff via `cpus[SOME_CPU_IDENTIFIER]`.
In order to get current CPU identifier, we call `getcpu()`, which is slow.

When we do this, we *MUST* stop interrupts from happening, to make sure we stay within the same CPU.

Problem: Calling `getcpu()` is slow, and stopping (and then resuming) interrupts can be extremely slow.

Solution: Instead of using `getcpu()`, we can use a special register in each CPU!

Problem: Registers can be overriden by users.

Solution: Special register that only kernel can use!

Problem: *There are no such registers.*

Solution: Cheating, using the GDT!

So we have a list of GDTs, one per CPU.

When we initialize a CPU (just once, yeah?), we set up its very own GDT.

In this GDT, we add a *fifth* row (remember, we have four rows for kernel/user x read/write).

In this new row, we set the base to point at the place in the memory where the CPU data sits, and set the limit to 8 (because we have two important variables of CPU data, and each one take 4).

We do this in `seginit()`.

```
struct cpus *c = &cpus[getcpu()]; // costly, but used just once!

// Set first four rows...
c->gdt[SEG_KCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA]=SEG(STA_S, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
c->gdt[SEG_UDATA]=SEG(STA_S, 0, 0xffffffff, DPL_USER);

// Set our special row
c->gdt[SEG_KCPU]=SEG(STA_W, &c->proc, 8, 0);

// Load the GDT
lgdt(c->gdt, sizeof(c->gdt));

// Set %gs register to point at our fifth row in the GDT.
loadgs(SEG_KCPU << 3); // shift 3 left so we get value and not permission bits.

// Declare out two variables,
// telling code generator to use gs register
// whenever we access proc or cpu
extern struct proc *proc asm("%gs:0");
extern struct proc *cpu asm("%gs:4");

proc = NULL;
cpu = c;
```

So how does this affect our accessing per-CPU data?

Normally, if we do `proc = 3;`, then it would be complied to `movl $3, proc`.

However, now that we did that weird `extern struct proc *proc asm("%gs:0");` part, it is compiled to `movl $3, %gs:0`.

Therefore, whenever in the code we call `proc` or `cpu`, we will automagically be referring to the address held in `GS:0` or `GS:4`, which happens to be the `proc` or `cpu` belonging to the specific CPU in which the code is executed.

Note that variable `cpu` points to the variables `cpu` and `proc`, so calling `proc` is the same as (yet faster than) calling `cpu->proc`.

Note also that this means that calling `cpu->cpu->cpu->cpu` is the same as calling `cpu`.

Important note: `GS` can be changed by code in user-mode.

So, every time there is an interrupt, we reload our value to `%gs` using `loadgs(SEG_KCPU << 3);`. (Don't worry, we back up all user-mode's registers beforehand.)

Processes

Every process has a struct `proc` that holds a bunch of its data:

- `sz` - memory size
- `pgdir` - its very own page table
- `kstack` - small *kernel* stack (4KB), for data we don't want the process to touch and break (such as return address from system call)
- `state`
- `pid` - process ID
- `parent` - parent process
- `tf` - pointer to place where register values are saved during interrupt
- `context` = another interrupt thing
- `chan` - channel, marks what event process is waiting for (while sleeping). If none, then set to 0.
- `killed` - 0 if not killed yet
- `ofile` - vector of open files
- `cwd` - current working directory (like when doing `cd...` in command)
- `name`

A process can move through its processes thusly:

Embryo -> Runnable -> Running

Running -> Zombie

Running -> Sleeping (waiting for whatever event is marked in `chan`)

Running -> Runnable

The processes are held in a struct named `ptable`, who has a vector of proceses named `proc`.

Preparing the First Process

When running The First Process, we do the following in `userinit`:

- `allocproc`: allocate `proc` structure

- Allocate process kernel stack
- Set up data on kernel stack (in the "trapframe" area) *as if the process already existed and went into kernel mode*. As a result, the data will be loaded to proc in due time

- `setupkvm`: create page table (without knowing yet where to map user addresses (only kernel addresses))
- `inituvm`:

- Allocate free page
- Copy code to page
- Update page table to map user memory

- Fix the data on the kernel stack for some reason.

Trapframe contains all register data.

When we return from an interrupt (or, in our case, start the process for the first time), we do the following:

1. Pop the first 4 values to their appropriate registers
2. Pop `eip` field to `%eip` register (now `%eip` register is pointing to `forkret` function)
3. Goto `forkret` function, which (somehow) takes us to the next guy in the stack...
4. ...Trapret which pops another whole bunch of values from the stack to the registers
5. And so on. We end up with all registers holding good values, and `%eip` pointing to 0.

The First Process

All it does it execute the program "init" (with the arguments {"/init", 0} (the 0 is to mark the end of the list)).

This is how the code *would* have looked like if it were written in c:

```
char *argv[] = {"/init", 0};
main() {
        exec(argv[0], argv);
        for (;;) exit();
}
```

...But it's not written in c.
It's written in assembly.

Here's the arguments code:

```
# these are the arguments:

init:
    .string "/init\0"

    .p2align 2
argv:
    .long init
    .long 0
```

...And here's the *code* code:

```
    .globl start
start:
    pushl       $argv ; push second argument
        pushl   $init ; push first argument
        pushl   $0
        movl    $SYS_exec, %eax ; load the canons (perpare to call "exec" system-call)
        int     $T_SYSCALL ; FIRE!

# once end up back here, we need to quit.
exit:
        movl    $SYS_exit, %eax
        int     $T_SYSCALL
        jmp     exit
```

So we have this code, but we need to load it during `inituvm`.

This is done in the script in the **initcode.S** file, which compiles that code to binary and shoves it all into the data section, with the exact same labels that are used in `inituvm`.

Actually running the First Process

This is done in `mpmain`, which calls the scheduler, which loops over process tables and runs processes.

Since at this point there's only one process at this point, it's the only one that'll run.

Accessing process's kernel stack during interrupt

When we're in user-mode and there's an interrupt, the kernel needs its own stack on the process to push real data onto it (before leaving the process).

We need this kernel-stack to be accessable only by the kernel.

The hardware supports this, with a magnificent Rube Goldberg machine:

- `%tr` register can only be accessed in kernel-mode
- `%tr` contains `SEG_TSS`, which points to special place in GDT
- special place in GDT contains address and size of some weird struct
- weird `TSS` struct contains field `esp0`, which actually (finally) points to top of process's kernel stack (and `ss0` with stack size)

So during `switchuvm` we set up this whole thing, so that in due time the kernel (and only the kernel) can access its stack on the proc.

Locks - the problem

Because we have multiple CPUs, we need to make sure two CPUs don't both grab the same data (such as the same `proc` struct during `allocproc`, or the same page during `kalloc`).

This could (theoretically) even happen with a single CPU, because of interrupts.

Locks - a possible solution (just for interrupts)

We can block interrupts (!) from happening by setting the `IF` bit in `%eflags` register to 0.

This can be controlled (in kernel-mode, yes?) using the assembly commands:

- `cli` (clears bit)
- `sti` (sets bit to 1)

(Also, the entire `eflags` register can be pushed into `eax` using some other assembly command.)

So each time we want to do something unsafe, we can do:

```
cli();
something_dangerous();
sti();
```

...But this is still dangerous, because of cases such as the following:

```
function b() {
        cli();
        // ...
        sti();
}

function a() {
        cli();
        b(); // at the end of this we set sti()!
        // ... Uh oh. We're open to interrupts!
        sti();
}
```

Locks - an actual solution (just for interrupts)

Solution: Keep track of how many times we called `cli()`, so that `sti()` only releases if down to 0.

We can do this by wrapping all `cli()`s with `pushcli()` and `sti()`s with `popcli()`.

In `popcli()`, if our count reaches 0, we don't automatically call `sti()`, but revert to the **previous setting**.

```
function b() {
        pushcli();
        // ...
        popcli();
}

function a() {
        pushcli();
        b(); // at the end of this we DO NOT set sti()
        // ... the counter is still 1
        popcli(); // NOW the counter became 0, so now we call cli().
}
```

(Note that each CPU runs `pushcli()` and `popcli()` separately for each CPU.)

Locks - an actual solution (for multiple CPUs)

There is a magical assembly word, `lock`.

Usage example:

```
lock ; xchgl mem, reg
```

If `lock` is added to different commands run by different CPUs, then the `lock`ed commands will execute *serially*.

Using this, the following c call:

```
old = xchg(&lock, b);
```

...Does the following:

1. Sets old = lock
2. Sets lock = b (but only if not locked by someone else!)

How on earth does this help us?

I'll tell you how!

We have a struct called `spinlock`, which holds all kinds of locking data.

In the function `acquire`, we do the following:

```
acquire(struct spinlock *lk) {
        pushcli(); // disable interrupts in this CPU, to avoid deadlocks
        while(xchg(&lk->locked, 1) != 0);
}
```

`xchg` will always return the *existing* value of `&lk->locked`, so if it's already locked we'll loop until it's released.

In `release`, we do the following:

```
release (struct spinlock*lk) {
        xchg(&lk->locked, 0);
        popcli();
}
```

How locks are managed during `scheduler`

When `scheduler` searches for RUNNABLE procs, it `acquire`s a lock on the process table.
This is in order to make sure that `scheduler`s on *other* CPUs don't grab the same process we just did.

The lock is held even as we switch to the process; it's the process's responsibility to release the lock (in order to allow interrupts).
Likewise, the process must re-lock the process table before returning to `scheduler`.

Each lock is re-released by the next process that runs, and then re-re-locked.

The final release is done by `scheduler` once there are no runnable processes left.

Sleep

When a process needs a resource in order to continue (such as disk, input or whatever), it must call `sleep` where it marks itself (`proc->chan`) as waiting for the wanted event, sets its `state` to SLEEPING, and returns to `scheduler`.

When the Event Manager (whom we never heard of yet) sees that the event occurs, it marks the proc as RUNNABLE once again.

Note that the process needs to see that its resource is still available (and hasn't been make unavailable again by the time it woke up) before continuing. If the resource is unavailable again, it should go back to sleep (if it knows what's good for it).

`sleep` also demands a locked `spinlock` for a weird reason we don't know yet.
It replaces the locking from *that* lock to the process table (unless they are the same lock, of course).
Once done, it releases the process table and re-locks the supplied `spinlock`.
We'll get back to you folks on this one once we understand why on earth this is needed.

Interrupts - CPU

There are two basic types of interrupts: internal (thrown by CPU) and external (like keyboard strokes).

We can decide whether to listen to external interrupts using kernel-mode functions `sti()` and `cli()` which sets or clears the `IF` bit on the `%eflags` register.
We cannot ignore internal interrupts.

Interrupts can only occur *between* commands.

During an interrupt, the CPU needs to know what kind of interrupt it is. These range between 0-255, when 0-31 are reserved by Intel for all kinds of hardware stuff.

In order to handle these interrupts, the CPU needs a table with pointers to functions that handle 'em (with the index being the interupt number).
This table is called IDT, and the register that points to it is `%IDTR`.

Each row has a whole descriptor of 64 bits. These include:

- `offset` - the actual address of the function (this guy is loaded to `%eip`)
- `selector` - loaded to `%cs`
- `type` - 1 for trap gate, 0 for interrupt gate (if interrupt, we disable other interrupts)

Before an interrupt call, the CPU must do a bunch of stuff to rescue our registers before they are overriden by the IDT guy.
Reminder: the kernel stack's address is saved in a `tss` structure, which is saved in the GDT, in the index held by `%tr` register.
Therefore: the kernel needs to set this before an iterrupt occurs.
SO: The CPU uses the kernel stack to store our registers.
(And after the interrupt handling is over, the CPU needs to pop these guys again. It does this with assembly `iret` command.)

If the interrups occurs during **kernel-mode**, the CPU performs the following:

- push `%eflags`,`%cs` and `%eip` to the **kernel** stack (so we'll have them again after the interrupt).

If the interrups occurs during **user-mode**, the CPU performs the following:

- Store `%ss` in `%ss3`
- Store `$esp` in `%esp3`
- Store `tss.ss0` in `%ss`
- Store `tss.esp0` in `%esp`
- Push `%ss3`, `%esp3`, `%eflags`, `%cs` and `%eip` to **kernel** stack

Question: Since we push a different number of registers for kernel-mode and user-mode, how do we know how many to pop back out?
Answer: In both cases, we need to first pop `%eip`, `%cs` and `%eflags`.
Having done that, we can look at the two right-most bits of `%cs` to see whether we were in user- or kernel-mode before the interrupt! Hurray!

Interrups - xv6

In xv6, all interrups are handled by a *single* C function `trap`.

However:

1. This function needs to end with `iret` (that's the guy who handles all the register poppin' at the end of the call), while C functions end with `ret`!
2. Different interrupts need to send different data to the function (such as interrupt number, error number (which not every interrupt needs)).

In order to accomplish this, we wrap the call to `trap` in assembly code that does the following:

1. Push registers to kernel-stack (except those which were already pushed by CPU (see previous section))
2. Push address of kernel-stack to kernel-stack (so the address is sent as a parameter to `trap`; I'll elaborate more on this in a moment)
3. Call `trap`
4. Pop the registers
5. Do `iret`

The function `trap` receives a pointer to a `trapframe` struct, which is actually the kernel-stack.
The fields in `trapframe` are the actual register values, so `trap` can use them in order to determine all kinds of stuff (such as "were we in user-mode when the interrupt occurred?"). Additionally, we have the interrupt number and error-code (where applicable).

The assembly code that wraps the call to trap is mostly the same for each interrupt, with just the slightest difference: Which interrupt number to push, and do we push an error-number (and if so, which).

So, the shared code is under a label `alltraps` (which, as it says on the label, is for all traps).
Each interrupt does its own little thing and then jumps to `alltraps`.

The code parts for the different interrupts are labelled `vector0`, `vector1`, ... `vector255`, and are generated by a script (in verctors.pl file).
After generating the vectors, the script creates an array (called - surprise, surprise - "vectors").
Later, during `main`, we call `tvinit` which loops over the vectors array and writes the data in the IDT table.

syscall

When user-code calls a system-call, the system-call number is stored in `%eax`.
So in `syscall`, we can access the system-call number via `proc->tf->eax`.

After running the system-call, we put the return value in `proc->tf->eax`; this will cause `alltraps` to pop it back to `%eax`, and then the user-code will know whether the system-call succeeded.

The system-call numbers are saved as constants in syscall.h file.

Then we have a vector that maps each constant to its appropriate function. This vector is used in `syscall` function in order to execute the system-call.

`sleep` system-call

There is a global variable called `ticks`.

Whenever there is a timer interrupt occurs, `ticks` increments by 1.

We use the *address* of `ticks` as the **event number** for the timer.

So...
When a process calls `sleep`, it sets its channel to equal `&ticks`, and stores the current value of `ticks`.
Whenever a timer event occurs, if the current value of `ticks` minus the value of `ticks` that the process stored *reaches* the number of ticks the process asked to sleep for, it becomes RUNNABLE.

Here's a sample of what a call to sleep looks like in assembly (for 10 ticks) :

```
pushl $10
subl $4, %esp ; empty word. Some weird convention
movl $SYS_sleep, %eax
int $64
```

Getting arguments in system-calls

In the example of `sleep`, the argument (how many ticks to wait) is stored on the **user**-stack.

This is accessed via `proc->tf->esp + 4` (the +4 is to skip the empty word, which is on the stack because of some weird convension). We do this in `argint` function.

Because this the data is placed on the stack by *user code* (which is pointed at by `%esp`), we need to check that the user didn't set `%esp` to point at some invalid address! We do this in `fetchint` function.

fork() #2

When we `fork` a proc, we need to copy a bunch of its data.
We want to map *new* physical addresses, but copy the old data.
We want to create a *new* kernel stack.
We want to copy context.
We want the state to be RUNNABLE (and not RUNNING like parent).
We want a new `pid`.

A lot of this stuff is already handled by `allocproc`.

The memory stuff is handled by `copyuvm`.

exec() #2

`exec` replaces the current process with new one.

Because this may fail, we don't want to destroy the current memory data until we know we succeeded.
In order to keep the current memory until the end, we need to create a *new* memory mapping, and only switch to it (and destroy the old mapping) at the very end.

The four stages of `exec`:

1. Load file
2. Allocate user stack (and a guard page to protected data from being overwritten by stack)
3. Pass argv[] arguments
4. Fix trapframe and switch context

In order for `exec` to run the file it's told to run, the file needs to be in ELF format.

ELF

Executable and Linkable Format.

In xv6 we only use *static* ELF, but in real life there are also *dynamic* ones but we don't care about that here.

ELF file have:

1. ELF header at the very start
2. `proghdr` vector
3. Program sections

The program sections include the actual code, as well as external code linked by the linker.
There can be many program sections, scattered along different parts of the file (but not at the beginning).
Each program section needs a *program header*, which says where it is in the file and where it is in the RAM.
All the *program headers* are held in the `proghdr` vector.
The location of `proghdr` is held in the ELF header.

ELFHDR (that's our ELF header) has a bunch of fields, but we'll only look here at a few:

- `uint magic` - must equal `ELF_MAGIC`. Just an assurance all's good.
- `ushort machine` - the type of processor. xv6 actually doesn't pay attention to this, so I don't know why we bothered mentioning this one.
- `uint entry` - virtual address of `main`
- `uint phoff` - PH offset, location of `proghdr`
- `uint phnum` - PH number, length of `proghdr`
- `uint flags` - uh... flags.

Since the ELF is created by user-code, the kernel doesn't trust it.
The kernel treats the data with caution, and fails the `exec` if there are any errors.

`proghdr` has the following important fields in each vector:

- `uint off` - where section starts in file
- `uint filesz` - where sections ends in file (so last byte is in `off+filesz-1`)
- `uint vaddr` - virtual addres to which section is loaded
- `uint memsz` - size section takes in memory. May be larger than `filesz`, because section may include lots of zeros (but not the other way around).

The kernel needs to loop and copy the memory, allocating new memory if either 1) we need more memory or 2) `vaddr` is beyond what is already allocated. This is done using `allocuvm`.

`exec` - *guard page*

After `exec` loads da codes, it allocates another page for the user-stack.

Problem: Stack goes to *lower* addresses as it fills up; eventually, it'll overrun the code-n-data!

Solution: Add guard page, with user-mode bit cleared. That way, user-code will get an error if StackOverflow.

`exec` - *pushing arguments to new process*

New process expects the following to be on the user-stack:

variable	what it holds
`argv`	pointer to vector of pointers to string arguments (with 0 in last place)
`argc`	number of arguments
	return address

...And this is how `exec` fills the user-stack:

user-stack

argument
...
argument
0
pointer to argument
...
pointer to argument
`argv` (pointing to this ↑ guy)
`argc`
-1

Note that the `pointers to arguments` don't really need to be on the stack; we just put 'em there because we have to put them *somewhere* so why not.

Note also that we don't know in advance how many arguments there are and how long they'll be.
Therefore, our code must first place the arguments stuff, and only afterwards copy `argc`, `argv` and return address.
Therefore, we save `argc` & `argv` & return address to a temporary variable `ustack`, which we copy to actual stack at the end of the argument pushing.

I/O

xv6 supplies the following system-call functions for files:

- `open(char *name, int flag)` - opens file and returns file descriptor (i.e. index in file array)
- `close(int fd)` - closes file
- `pipe(int pipefd[2])` - kinda like a file, for sending messages between processes
- `dup(int fd)` - duplicates opening to file (so if we move the cursor/offset of one (such as by reading), it'll affect the other)
- `read`
- `write`
- `stat`

Reading/writing can be much different if we're dealing with keyboard, file, etc., but xv6 strives to make them "act" the same.
We've got the following read/writables:

- pipe
- inode
    - file
    - directory
    - device

Each has its own internal functions for reading, writing, etc., but - as mentioned above - xv6 wraps 'em and hides 'em from user code.

xv6 has an abstract struct `file`, which it uses for all the above read/writables.
xv6 has functions that deal with this `file`, but they accept a *pointer* instead of a file descriptor.
Why? Because:

- xv6 needs a pointer to an actual struct, because it needs to handle data.
- User code isn't trusted to handle pointers; it gives us a file descriptor, and the *kernel* accesses the actual pointer.

So how to we maintain this translation between file descriptors and pointers?
With an array "ofile", that's how.

These `file` guys have basic data that all read/writables contains, such as "type".
Inodes implement their "inheritance" by having their "type" be "FD_INODE", and containing a pointer (in "inode" field) to a `struct inode` that has its very own data (such as "type", which could be "T_FILE" for a file or "T_DEV" for device).

OK, this explanation is pretty lame :(
You should really just see the "I/O" presentation on Carmi's site, which he changes every semester.
Currently, the presentation is at http://www2.mta.ac.il/~carmi/Teaching/2016-7A-OS/Slides/Lec%20io.pdf (http://www2.mta.ac.il/%7Ecarmi/Teaching/2016-7A-OS/Slides/Lec%20io.pdf) (slides 13-23).

Pipe stuff

A `pipe` is pointed to by a *reader* `file struct` and a *writer* `file struct`.
(Both of which exist *once*, and may be pointed to by many different procs.)

The `pipe` has `readopen` and `writeopen` fields, which contain "1" as long as the *reader* and *writer* guys are still holding him open.
When both are "0", the `pipe` is closed.

Moar file stuff

The kernel has an array `ftable`, which holds all the open files (and a lock to make sure two procs don't fight over a slot in the array).

Files are added to `ftable` in the function `filealloc`.

Transactions

When writing to blocks on disk (or deleting), we really don't want the machine to crash or restart, or else there'll be corrupted data.

In order to alleviate our fears, we can wrap our *writing* in a *transaction* (by calling `begin_trans()` before and `commit_trans()` after).
A transaction ensures that the write will either be *complete* or won't be at all.

A transaction is limited to a certain size, so during `writefile` we need to divide the data to managable chunks.
Therefore, if there's a crash in the middle of the write operation, some blocks might be written and the rest not. However, each block will be fully-written or not written at all (but no hybrid mish-mash).

inode Stuff

xv6 considers a directory to be a file like any other.
Every `struct dirent` has an inode number, and a name. (If the directory is not in use, the number is 0.)

The inodes are managed by the inode layer, which we will talk about later.

`struct inode` has the following amongst its fields:

- `uint dev` - device number
- `uint inum` - inode number
- `int ref` - reference count
- `int flags` - flags, such as 0x1 for I_BUSY (that is, *locked*) and 0x2 for I_VALID
- `short nlink` - the number of names ('links') the inode has on the disk (kinda like a shortcut in Windows)
- `uint size` - the size of the inode on the disk

In order to open an inode, we have `namei()`.
In order to search for an inode, we have `dirlookup()`.

Note that `dirlookup` can be supplied with an optional pointer `poff` that gets the offset of the found inode.
Why?
In case of renaming or deleting, we'll need to update the row in our parent inode. (In case of name change, change the num; in case of deletion, change `inum` to 0.)

So... when we open an inode, we need to supply the inode to search in.
Conveniently, we actually *don't* supply an inode pointer, because we have THE CURRENT WORKING DIRECTORY which is "supplied" automatically.

...Actually, that's not exactly true.
It *is* true if we're looking for some "a/b/c" path, but **not** for "/a/b/c".
In the latter case, we use the **root inode** instead of the current working directory.

As you may or may not have guessed, if we have a path with a few parts (such as "a/b/c/d"), we need to loop over the parts and for each part find the matching inode.

In order to split the path into parts, we use `skipelem`.

inode Stuff - The Inode Layer

The inode layer supplies a bunch of funcs:

- `iget` - open
- `iput` - close
- `idup` - increments ref count
- `ilock` - must lock inode whenever we want to access any field
- `iunlock` - unlocks inode
- `ialloc` - allocates inode both on disk and in memory
- `iupdate` - update disk with whatever chanes we made to inode
- `readi`
- `writei`

Our inode structs are saved in `icache`.

They are saved there the first time they are opened (but not before; they are loaded on-demand).

The inodes are added to `icache` in `iget`.

When added, we add them *without the actual data from the disk*.

Why?

Because often we call `iget` just to see if the inode exists, and actually reading from the disk is expensive (so we do it just if we need it).

The Disk - Reading and Writing

The disk in divided to blocks.

Every block on our disk is of 512 bytes; no more, no less.

Reading and writing to and from a block is done *just* in the beginning of a block.

So how do we write (or read) only a few bytes?

We have a buffer layer that takes a whole block to memory, writes (or reads) our few bytes *in memory*, then - if we're writing - re-writes the entire block to the disk.

The buffer layer contains the functions:

- `readsb` - reads super-block. Never mind.
- `bread(dev, sector)` reads an entire block. (function returns a `struct buf` which contains a field `data` with the actual 512 bytes of data)
- `log_write`
- `brlse`
- `balloc`
- `IBLOCK` - we'll explain in a moment:

Remember `struct inode`?

Well, there's also `struct dinode` which represents an inode *on the disk*.

It's similar to the inode, but without the meta-data (such as ref, inum, etc.).

`IBLOCK` is a macro that gives us the block number of an inode.

In order to tell *where* within the block is our inode, we can calculate (inum % IPB). IPB is the number of inodes in a block.

`ilock`

As mentioned before, before accessing inode data we need to `ilock` it.

We don't want to use our regular locks, because they:

- disable interrupts
- when trying to acquire, "spin" over lock till lock is open

This wastes a lot of time.

`ilock` is a *soft* lock, that doesn't do this stuff.

We don't disable interrupts, and instead of spinning, we do `sleep` (so we're not hogging CPU while waiting for acquiring).

Reading from the disk

Every inode on the disk has a vector of `addrs`.

Every entry points at a single block on the disk.

The first 12 entries are *direct* pointers.

The 13th points to a block that serves as vector of pointers.

That is, the last pointer is a pointer to pointers.

It turns out there's actually a reason for having the inode data be partially direct and partially indirect:

- We have the direct blocks because reading from the indirect blocks cost an extra read for each block.
- We have the indirect blocks because the direct blocks are a waste of space for small files.

The buffer layer

The buffer layer supplies the following functions:

- `bread`
- `bwrite`
- `balloc`
- `bfree`

`struct buf` has the following fields:

- `flags` - such as BUSY or DIRTY
- `dev`
- `sector` - device and block num
- `struct buf * prev` - for linked list
- `struct buf * next` - for linked list

- `struct bef * qnext`
- `uchar data[512]`

All our buffers are kept in a cache, `bcache`.
This cache contains a spinlock (obviously), and a linked list of buffers (held in field `head`).

The linked list of buffers is maintained so that the most recently used is in the head, and the least in the tail.

Buffs are never removed from the cache!
If they're not used, they're marked as *not* busy and *not* dirty.
If we need a new buff (such as calling `bget` for a block that hasn't been read yet), we find a `struct buff` at the end of the list that is neither BUSY nor DIRTY, and mark it BUSY.

When we call `brelse` and release a buffer, it is moved to the head of the list (and it's BUSY flag is turned off).

`bcache` is initialized in `binit()`.

The superblock and the bitmap

There is a block on the disk called the *superblock*.
This block is located right after the boot block, and contains meta data regarding the other blocks.

There is another area on the disk called the *bitmap*.
For every block on the disk, there is a bit that marks whether the block is free or not.

The driver layer

Ah, the driver layer. The guy who actually does all the dirty work.

In order to actually access the disk (as well as other peripherals, such as keyboard), we have controllers.
Our guy is the IDE.

The IDE has a bunch of *ports*, which are actually numbers (but "port" sounds a lot more computer-y that "number").
The processor can send/receive values to/from these ports using IN/OUT commands.
The IDE decides what to do for different values it is given to different ports.

If we want the IDE to read (or write) from the disk, we need to tell it:

1. Which disk to read from (turns out there are two)
2. Where on the disk to read (28 bits to tell us which block)
3. How many blocks to read

Each of these needs to be sent to different OUT ports, before we send the command "read" to the port that accepts the command:

- 0x1f2 - number of blocks
- 0x1f3, 0x1f4, 0x1f5 and 0x1f6 - block number (28 bits split to 8 bits per port (0x1f6 gets only 3 bits and device number))
- 0x1f7 - the command

The IDE can only handle one single command at a time; so before we do anything, we need to check its IN 0x1f7 port (the STATUS port).
The left-most bit tells us if the IDE is busy (so shouldn't use).
The second bit tells us if the IDE is ready (so can use).

The IDE has its very own RAM attached to it, which it uses to read/write.
(NOTE: We don't know how much is in this RAM. It depends on the controller we chose to put in our PC.)

So how do we access this RAM from our code?

Accessing this RAM from our code

In order to do this, we use OUT port 0x1f0.
For example:

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
        outl(0x1f0, a[i]);
```

Although it looks like every `long` is being written to the same place, it isn't.

Same goes for reading:

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
        a[i] = inl(0x1f0);
```

Another useful parameter to send is `0` to port 0x3f6.
This tells the IDE to raise an interrupt when it finishes the command.
(Otherwise, the kernel can't tell when the read/write from/to the disk is complete.)

Functions provided by the driver layer

- `idewait` - loops over IDE port 0x1f7 until status is READY
- `idestart`- *starts* requesting to read/write (whether to read or write depends on input)
- `iderw` - the *real* read/write function. Handles a queue of blocks to write, because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk.

In order to handle the queue, the buffer layer has a variable `idequeue`.
This is a linked list of bufs, which are actually read/written by `ideint`, which is run whenever there is an interrupt from IDE due to IDE finishing its previous command.

- `ideint` - handles "I'm finished!" interrupt from IDU and managed `idequeue`.

main *(kernel enty point)*

- **kinit1** *(frees pages)*
    - **freerange** *(frees pages)*
        - **kfree** *(frees single page)*
- **kvmalloc** *(builds new page table)*
    - **setupkvm** *(sets up kernel page table)*
        - **mappages** *(adds translations to page table)*
            - **walkpgdir** *(allocates and maps page)*
                - **kalloc** *(allocates page)*
                - **memset** *(clean new page)*
- **seginit** *(sets up segmentation table)*
- **tvinit** *(initializes interrupt table)*
- **kinit2** *(frees pages)*
    - **freerange** *(frees pages)*
        - **kfree** *(frees single page)*
- **userinit** *(initialize the First Process)*
    - **allocproc** *(allocates new proc)*
    - **setupkvm** *(sets up kernel page table)*
        - **mappages** *(adds translations to page table)*
            - **walkpgdir** *(allocates and maps page)*
                - **kalloc** *(allocates page)*
                - **memset** *(clean new page)*
    - **inituvm** *(allocates and maps single page, and copies First Process code to it)*
- **mpmain**
    - **idtinit** *(sets %IDTR to point at existing interrupt table)*
    - **scheduler** *(runs runnable processes)*
        - **acquire** *(locks process table)*
            - **pushcli** *(makes us ignore interrupts)*

        - **switchuvm** *(prepares proc's kernel stack and makes TSS point to it)*
        - **swtch** *(saves current context on proc, and switch to new proc)*
        - **switchkvm** *(switches back to kernel page table)*
        - **release** *(unlocks process table)*
            - **popcli** *(makes us stop ignoring interrupts)*

---

fork *(creates child process)*

- **allocproc** *(allocates new proc)*
- **copyuvm** *(copies memory)*
    - **setupkvm** *(sets up kernel page table)*
        - **mappages** *(adds translations to page table)
            - **walkpgdir** *(allocates and maps page)*
                - **kalloc** *(allocates page)*
                - **memset** *(clean new page)*
    - **walkpgdir** *(validate that page mapping exists, without allocating or cleaning)*
    - **kalloc** *(allocate new page for user-code)*
    - **memmove** *(copy page data)*
    - **mappages** *(add user-code page)*
        - **walkpgdir** *(maps page, without allocating or cleaning)*
    - **freevm** *(free page table in case of error)*
        - **deallocuvm** ()
            - **walkpgdir** *(get entry of internal page)*
            - **kfree** *(free actual page)*
        - **kfree** *(free inner table)*
        - **kfree** *(free outer table)*
- **kfree** *(if error, free kernel-stack)*
- **filedup**
- **idup**

- **safestrcpy**

Hello, world.

In this document, we'll attempt to explain the actual code of our beloved xv6.
Not all of it, but some of the interesting parts.

God have mercy on us.

1217 main(void)

The entry point of the kernel.
Sets up kernel stuff and starts running the first process.

1219: set up first bunch of pages, for kernel to work with (minimal, because old harware has little memory)

1220: set up all kernel pages

1223: set up segment tables and per-CPU data

1238: set up the rest of the pages for general use (because until now we had just minimal, because other CPUs might not handle high addresses)

1239: set up the First Process

1241: run the scheduler (and the First Process)

2764 struct run

Represents a memory page.
A `run` points to the next available page/`run` (which is actually the *previous* page, because the first available is the last in the memory).

2768 struct kmem

Points to the head of a list of free (that is, available) pages of memory.

2780 kinit1(void *vstart, void *vend)

Frees a bunch of pages.
Also does some locking thing (I'll elaborate once we actually learn this stuff).
Used only when kernel starts up.

Called by `main`.

2788 kinit2(void *vstart, void *vend)

Frees a bunch of pages.

Also does some locking thing (I'll elaborate once we actually learn this stuff).

Used only when kernel starts up.

Called by `main`.

2801 freerange(void *vstart, void *vend)

Frees a bunch of pages.

2804: use PGROUNDUP because `kinit1` in called to start where the kernel finished (which is not likely to end *exactly* at a page end).

Called by:

- `kinit1`
- `kinit2`

2815 kfree(char *v)

Frees the (single!) page that `v` points at.

2819-2820: address validity checks

2823: fill page with 1s, to help in case of bugs

2827-2829: insert our page into the beginning of `kmem` (a linked list with all available pages)

Called by:

- `deallocuvm`
- `freevm`
- `fork`
- `wait`
- `freerange`
- `pipealloc`
- `pipeclose`

2838 kalloc(void)

Removes a page from `kmem`, and returns its (virtual!) address.

2844-2846: remove first free page from `kmem`

2849: return address

Called by:

- `startothers`
- `walkpgdir`
- `setupkvm`
- `inituvm`
- `allocuvm`
- `copyuvm`
- `allocproc`
- `pipealloc`

1757 kvmalloc(void)

Builds new page table and makes `%CR3` point to it.

1759: make table and get its address

1760: make `%CR3` point to returned address

Called by `main`.

1728 kmap[]

Contains data of how kernel pages should look.
Used by `setupkvm` for mapping.

Column 0: Virtual addresses
Column 1: Physical addresses start
Column 2: Physical addresses end
Column 3: Pages permissions

Note: The `data` variable (used in lines 1730-1731 is where the kernel's data start (*data* is plural, by the way).
We do not know during compliation where this will be.

1737 setupkvm(void)

Sets up kernel virtual pages.

Returns page table address if successful, 0 if not.

1742: create outer `pgdir` page table

1744: clear `pgdir`

1745-1746: make sure we didn't map illegal address

1747-1750: loop over `kmap` and map pages using `mappages`

Called by:

- kvmalloc
- copyuvm
- userinit
- exec

1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)

Creates translations from *va* (virtual address) to *pa* (physical address) in existing page table `pgdir`.

Returns 0 if successful, -1 if not.

1684: get starting address

1685: get ending address (which is starting address if `size`=1)

1686: for each page...

1687: get i1 row address (using `walkpgdir`)

1689: make sure i1 row not used already

1691: write *pa* in i1 and mark as valid, with required permissions

Called by:

- setupkvm
- inituvm
- allocuvm

- `copyuvm`

1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)

Looks at virtual address `va`,

finds where where it should be mapped to according to page table `pgdir`,

and returns the **virtual** address of the the *index* i1.

Returns address if successful, 0 if not.

If there is no mapping, then:

if `alloc`=1, mapping is created (and address is returned);

if `alloc`=0, return 0

Some constants and macros used here:

PDX - zeroes offset bits

PTX - uh... some more bit manipluations

PTE_P - "valid" bit

PTE_W - "can write" bit

PTE_U - "available in usermode" bit

1659: get i0 index address

1660: check if i0 is valid

1661: put address of subtable in `pgtab`

1663: (i0 not valid) create new subtable, `pgtab`

1666: (i0 not valid) clear `pgtab` rows

1670: (i0 not valid) point i0 to `pgtab` and mark **i0** as valid, writable, usermodeable.

1672: return address of appropriate row in `pgtab`

Called by:

- `mappages`
- `loaduvm`
- `deallocuvm`
- `clearpteu`
- `copyuvm`
- `uva2ka`

1616 seginit(void)

Sets segmentation table so that it doesn't get in the way, for each CPU. Adds extra row to each segementation table in order to guard CPU-specific data, makes `%gs` resgister point to it, and makes `proc` and `cpu` actually point to `%gs`.

1624-1628: set up regular rows in segmentation table

1631-1634: set up special row and `%gs` register

1637-1638: set up inital `proc` and `cpu` data

Called by `main`.

2252 userinit(void)

Creates and sets up The First Process.

2255: data and size of First Process code. Filled by script during compilation.

2257: allocate `proc` structure and set up data on kernel stack

2258: save proc in `initproc`, so we'll always remember who the First Process is

2259-2260: create page table with kernel addresses mapped

2261: allocate free page, copy process code to page, map user addresses in page table

2262-2275: fix data on kernel-stack, *as if* it were stored there because of an interrupt

- **2264**: make sure we'll be in usermode when process starts
- **2270**: make sure process will start in address 0

Called by `main`.

2205 allocproc(void)

Allocates `proc` structure and sets up data on kernel stack.
Returns proc if succeeds, 0 if not.

2211-2213: find first unused `proc` structure

2217-2219: set to EMBRYO state and give `pid`

2223-2226: allocate and assign kernel stack for process

2227: set stack pointer to bottom of stack (stack bottom is highest address in stack)

2230-2231: make room on stack for `trapframe`

2235-2239: make room on stack for `context`

2240-2241: set `context`, setting `context.eip` to function `forkret`

Called by:

- `userinit`
- `fork`

1803 `inituvm(pde_t *pgdir, char *init, uint sz)`

Allocates and maps single page (4KB), and fills it with with program code.

1807-1808: make sure entire program code fits in single page

1809: allocate page

1810: clear page

1811: map pages in page table `pgdir` (using `v2p`, because we know what memory this is, because this is the First Process, which the kernel always creates on startup)

1812: copy the code

Called by `userinit`.

2458 `scheduler(void)`

Loops over all processes (in each CPU), finds a runnable process, and runs it.
Loops for ever and ever.

2467: lock process table, to prevent multiple CPUs from grabbing same process

2468-2470: find first available process

2475: set *per-CPU* variable `proc` to point to current running process

2476: set up process's kernel stack, and switch to its page table

2477: mark process as running

2478: save current registers (including where to continue on scheduler) and load process's registers, handing the stage over to the process
(NOTE: the running process is responsible to release the process table lock (to enable interrupts) and later re-lock it.)

2479: now that process is switching back to scheduler, switch back to kernel registers and Page Table

2483: set per-CPU variable `proc` back to 0

2485: release process table, allowing other CPUs to grab processes (just in case

Called by `mpmain`.

1773 switchuvm(struct proc *p)

Perpares kernel-stack of process (that is, makes `%tr` register indirectly point to it), and loads process's Page Table to `%cr3`.

1776-1779: set up `%tr` register and `SEG_TSS` section in GDT end up magically (don't ask how) referring us to top of process's kernel stack

Called by:

- `growproc`
- `scheduler`
- `exec`

2708 swtch(struct context **old, struct context *new)

Saves current register context in `old`, then loads the register context from `new`.
Basically gives control to new process.

2709: set `%eax` to contain address of `old` context

2710: set `%edx` to contain `new` context

2713-2716: push `%ebp`, `%ebx`, `%esi`, `%edi` onto current stack (which happens to be `old` stack)

2719: copy value of `%esp` to address held in `%eax`, which is the `old` stack address (see line **2709**)

2720: set current stack pointer (`%esp`) to value of `%edx`, which is the `new` stack address (see line **2710**)

2723-2726: pop `%edi`, `%esi`, `%ebx`, `%ebp` from `new` stack onto the actual stack

Called by:

- `scheduler`
- `sched`

2503 sched(void)

Switches back `scheduler` to return from a process that had enough running.

Called by:

- `exit`
- `yield`
- `sleep`

2522 yield(void)

Gives up the CPU from a running process.

2524: re-lock the process table for scheduler

2525: make self as not running

2526: switch back to scheduler

2527: after scheduler re-ran process, re-ealease process table to enable interrupts

Called by `trap`.

2553 sleep(void *chan, struct spinlock *lk)

Makes process sleep until `chan` event occurs.

2568: lock process table in order to set sleeping state safely

2569: now that process table is locked, release `lk`

2573-2574: set up sleeping state (and alarm clock)

2575: return to scheduler until the event manager marks process as runnable

2578: clean up

2582: release process table

2583: lock `lk` once again

1555 pushcli(void)

Saves state of `%eflags` register's `IF` bit (that is, the current state of "listen to interrupts?" bit),
increments the "how many times did we choose to ignore interrupts" counter,
and clears the "listen to interrups" bit.

1561-1562: save initial state of bit in `interna` var (FL_IF is the location of our bit)

Called by:

- `acquire`
- `switchuvm`

1566 popcli(void)

Decrements the "how many times did we choose to ignore interrupts" counter,
and if it reaches 0 then sets the "listen to interrups" bit to whatever it was before the very first `pushcli` was ever called.

1572-1573: only set our bit if `interna` (initial bit value) was set

Called by:

- `release`
- `switchuvm`

1474 acquire(struct spinlock *lk)

Loops over spinlock until lock is acquired (exclusively by current CPU).

1476: disable interrupts (which will be enabled in `release`)

1483-1484: loop over lock until it has a zero (and write "1" in it)

1502 release(struct spinlock *lk)

Releases spinlock from being held by current CPU.

1519: write "0" in lock

1521: re-enable interrups (which were disabled in `acquire`)

3004 alltraps

Catches and prepares all interrupts for `trap`.
Pushes register data on stack, calls `trap` with the `stack` as a `trapframe` argument, pops register data from stack, and finally calls `iret`.

3005-3010: store registers and build `trapframe`

3013-3018: set up data and per-CPU segments (?)

3021-3023: call `trap`, using stack as argument

- **3023**: skip over top of frame (`%esp` address) without popping it into anything

3027-3034: pop registers and call `iret`

- **3033**: skip over data and per-CPU segments (without popping it into anything)

3101 trap(struct trapframe *tf)

Handles all interrups.

3103-3111: handle system call and return

- **3106**: save `tf` to `proc->tf`, so that we don't need to start passing it around during `syscall`

3113-3143: handle controller interrupts (keyboard, timer, etc.)

3150-3163: handle unexpected interrupt

- **3151**: check if there is no current process (i.e. during `scheduler`) or if we were in kernel-mode during interrupt
- **3158-3162**: print error and kill buggy process

3168-3178: finish up non-system calls

- **3168-3169**: if process is user-process, and killed, and is not in the middle of a system-call, exit (and don't return to `alltraps`)
- **3173-3174**: if process is running, and we had a timer-interrupt, and the process ran for long enough already, yield CPU back to `scheduler`
- **3177-3178**: after previous yield, if process was killed (and not in middle of system-call), exit

Called by `alltraps`

3067 tvinit(void)

Initializes the IDT table.

Called by `main`.

3079 idtinit(void)

Makes `%IDTR` point at existing IDT table.

Called by `mpmain`

3375 syscall(void)

Handles system-calls from user-code.

3379: get system-call number

3380: make sure number is valid

3381: execute system-call and store return value in process's trapframe's `eax` field (which will afterward be popped to `%eax`)

3382-3385: if bad system-call number, print error and store -1 (error) in trapframe's `eax` field

Called by `trap`.

3465 sys_sleep(void)

System call for sleeping a certain amount of ticks.

3470: get number of required ticks (and validate that the user supplied a valid address as an argument)

3472: lock tickslock

3473: store current (initial) value of `ticks` in `ticks0`

3474-3480: while required number of ticks didn't pass, loop

- **3479**: wait for event #`&ticks` (we don't really need the lock in this case, but usually in `sleep` call we need to lock because other cases we `sleep` for disc or something else where we don't want two process's to grab the resource simultaneously)

Can be called by user code.

2614 wakeup(void *chan)

Locks process table, finds all sleeping processes that are waiting for `chan`, makes them runnable, and unlocks process table.

Called by a lot of different functions.

2603 wakeup1(void *chan)

Finds all sleeping processes that are waiting for `chan`, and makes them runnable.

Called by:

- `exit`
- `wakeup`

3295 argint(int n, int *ip)

Gets the `n`th *integer* argument pushed onto the user-stack by user code before user asked for system-call.

3267 fetchint(uint addr, int *ip)

Gets the integer argument in address `addr`, and sets it in `ip`.

Returns 0 if successful, -1 otherwise.

3267: Validates that neither "edge" of integer-containing address space goes beyond valid proc memory.

Called by:

- `argint`
- `sys_exec`

2304 fork(void)

Creates new process, copying lots from its parent, and set stack as if returning from a system-call.

2310: allocate new proc. Proc now contains kernel-stack, context (with trapret address), trapframe and pid

2314-2319: copy memory

- **2314**: copy memory
- **2315-2318**: if error, free kernel stack

2320: copy `sz`

2321: set parent

2322: *copy* trapframe struct to new kernel-stack

2325: clear `%eax` so `fork` will return 0 for child process

2327-2330: do file stuff

2333: make new proc RUNNABLE (at this point, `scheduler` can grab child process before parent)

1953 copyuvm(pde_t *pgdir, uint sz)

Creates copy of parent memory for child process.

Returns address of new page table.

1960: set up kernel virtual pages

1962: loop over all pages:

- **1963**: get address+flags of parent process's page
- **1965**: make sure page is actually present
- **1967**: get physical address of parent process's page
- **1968**: allocate new page
- **1970**: copy memory from old physial page to new physical page
- **1971**: add-n-map new page to new page table

1974: if no errors, return address of new page table

1977-1978: if there were any errors, release all memory and return 0

Called by `fork`

1910 freevm(pde_t *pgdir)

Frees a page table and all the physical memory pages (in its user part).

1916: free user-mode pages

1917-1921: free internal page tables

1923: free external page table

Called by:

- `copyuvm`
- `wait`
- `exec`

1882 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)

Deallocates user pages to bring the process size from `oldsz` to `newsz`.

1891: loop over extra pages we want to deallocate:

- **1892**: get virtual address of internal table entry that points to current page-to-remove

- **1896**: get physical address of page
- **1899**: get virtual address of page
- **1900**: free page
- **1901**: mark internal page table entry as "pointing at no page"

Called by:

- `allocuvm`
- `freevm`
- `growproc`

1853 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)

Allocate page tables and physical memory to grow process from `oldsz` to `newsz`.
returns `newsz` if succeeded, 0 otherwise.

Called by:

- `growproc`
- `exec`

5910 exec(char *path, char **argv)

Replaces current process with new one.

5920-5949: load da code

- **5920**: open file
- **5926**: read ELF header
- **5936-5947**: loop over sections in `proghdr`:
  - **5937**: read section from file
  - **5943**: allocate memory
  - **5945**: load code and data
- **5948**: close file

5952-5956: allocate user-stack and guard page

- **5952**: round up address in order to add stack and guard-page at new page
- **5953**: allocate two pages, for stack and guard page
- **5955**: remove user-mode bit from guard page
- **5956**: set stack pointer to point to stack page

5959-5967: push arguments to new process user-stack

- **5959**: loop over arguments:
    - **5962**: move new process stack pointer so there's room for current argument
    - **5963**: copy argument to actual new process user-stack
    - **5965**: make `argv` vector entry (which is still in temporary `ustack` variable!) point to current argument that we just pushed to stack
- **5967**: put 0 in last entry of `argv` vector (which is still in temporary `ustack` variable!), as is expected by convention
- **5969**: put -1 as return address in appropriate spot in temporary `ustack` variable
- **5970**: put `argc` in appropriate spot in temporary `ustack` variable
- **5971**: put address of where `argv` will be in new user-stack (but isn't there yet) in appropriate spot in temporary `ustack` variable
- **5974**: now that all arguments are copied to new user-stack, and we know where to place return address & `argc` & `argv`, copy `ustack` variable to new process user-stack

5978-5981: copy new process name

- **5978-5980**: get part of the name after all the slashes

5984-5991: switch address space and fix trapframe

- **5984**: save old page table for freeing later
- **5985**: give proc new page table!
- **5986**: give proc new size
- **5987**: set trapframe's `eip` to new process entry point
- **5988**: set trapframe's `sp` to new process user-stack
- **5989**: make `%CR3` point to new page table (which doesn't harm our current running!)
- **5990**: free old page table

- **5991**: return to syscall, which returns to alltraps, which returns to popall, which returns to iret, which pops a bunch of values to actual registers, which include `%eip`, which makes us actually continue with the new process

---

1818 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)

Loads a `sz`-sized program section from `ip` file (in `offset` offset) to address `addr` (which is already mapped in `pgdir`).
Returns 0 if successful, -1 otherwise.

1825-1835: loop page by page (because the pages mapped in `pgdir` are scattered (and we can't rely on the Paging Unit to handle this, because `pgdir` is not our current page table))

- **1826**: validate that page is already mapped
- **1828**: get physical address
- **1829-1832**: take care of case where what's left to read is less than page
- **1833**: copy code-n-data

Called by `exec`

2002 uva2ka(pde_t *pgdir, char *uva)

Returns the kernel virtual address of a user virtual address.
Only works for addresses of pages (and not for middle of page).

2006: get `pte` entry

2011: get offset, do `p2v` to it, and return result

2018 copyout(pde_t *pgdir, uint va, void *p, uint len)

Copies `len` bytes from `p` address to `pgdir->va` address.

2029: get `va` offset within its page

2032: copy data

2354 exit(void)

Exists current process.

2359-2360: make sure we're not the First Process

2363-2368: close all files opened by user-code

2370-2371: close current working-directory

2376: let parent know we're exiting (in case parent called `wait` for child to exit) (The `chan` the parent is waiting for is the parent's `proc` address)

2379-2385: pass abandoned children to the First Process

- **2382-2383**: if child did `exit`, let the First Process know

2388: become zombie

2389: awaken the `scheduler`

5225 `filealloc(void)`

Finds the first free slot in the global file table, and returns its *address*.
If there are none free, returns 0.

5438 `fdalloc(struct file *f)`

Finds the first free slot in the process's file table, points it to `f`, and returns the index (AKA the file descriptor).
If no room, returns -1.

5252 `filedup(struct file *f)`

Increments the reference count of `f`.
(Used as part of the file duplication process)

5451 `sys_dup(void)`

Duplicates proc's reference to file.

5458: actual duplication

5460: increment ref count

5419 `argfd(int n, int *pfd, struct file **pf)`

Gets the `n`th argument sent to the system call, as a file descriptor.

Returns descriptor and the struct file it points to.

(The only reason we need the file descriptor `pfd` is in case we're closing the file and need to make `ofile` point to null.)

5315 fileread(struct file *f, char *addr, int n)

Reads from `f` to `addr`.

5319: make sure can read

5321-5322: handle case when file is pipe

5323-5329: handle case when file is inode:

- **5324**: lock the inode (because we must)
- **5325**: read
- **5326**: update offset
- **5327**: unlock the inode

Called by `sys_read`.

5352 filewrite(struct file *f, char *addr, int n)

Writes from `addr` to `f`.

5358-5359: handle case when file is pipe

5360-5386: handle case when file is inode:

- **5367-5372**: break up data to managable chunks (for transactions)
- **5374-5379**: write chunk:
    - **5374**: begin transaction
    - **5376-5377**: write
    - **5379**: end transaction

5264 fileclose(struct file *f)

Decrements file reference count.
When no references left, actually close file.

5271-5273: decrement ref count

5275: keep backup of struct file, because we're about to release the lock on the file table (and anything can happen after *that*)

5278: release file table because closing file on disk can take a long time

5280-5281: handle case when file is pipe (needs to happen once for `read` and once for `write` descriptors of pipe for it to actually close)

5282-5286: handle case when file is inode (using transaction because this can require writing on device)

5851 sys_pipe(void)

Allocates two files (read pipe and write pipe).
Expects a vector with two entries (from the input), in order to return the descriptors in.

5859: allocate pipe (creates 2 file structs)

5862: allocate slots in `ftable`, and point them to the pipe

5863-5867: remove files in case of failure

5869-5870: put descriptors in vector, for user code

5701 sys_open(void)

Opens or creates inode.

5710-5715: create inode (on disk!) - can only create file (not directory)

5716-5724: open inode

- **5720-5722**: if tried opening directory in *write* mode, close-n-error

5726: create struct for inode, add it to `ftable`

5734-5738: set file struct data

5011 dirlookup(struct inode *dp, char *name, uint *poff)

Finds an inode *under* `dp` with name that's equal to `name`.
(`poff` in an optional pointer to the offset of the found inode.)

5020: read single `struct dirent` in `dp`

5022: check whether `inum` of current `dirent` is active

5024: check whether name of current `dirent` equals `name`

If we found the droid we're looking for:

5026-5027: store offset in `poff` (if we supplied the optional pointer)

5028-5029: get actual inode

...And if we did not find it:

5033: return 0

5115 skipelem(char *path, char *name)

A helper function that helps us take apart "long/path/names".

Returns the value of `path` *without* the first part, and sets `name` to equal the chopped off head.

(Ignores the first instance of "/" in the path: "**/**a/b/c" acts the same as "a/b/c".)

5189 namei(char *path)

Returns the inode with the matching path.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

5192: ask `namex` to do the work

5196 nameiparent(char *path, char *name)

Returns the inode with the matching path *without the last part*.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

5198: ask `namex` to do the work

5154 namex(char *path, int nameiparent, char *name)

Returns the inode with the matching path.

(If `nameiparent` is 0, finds actual inode. Else, ignores last part (for case when we want to *create* a *new* inode).)

5158-5161: check if we need to look in root or current working directory

5163-5180: loop over parts in path:

- **5164**: `ilock` current inode, because we must
- **5165-5168**: make sure we're looking at a *directory*
- **5169-5173**: if we're looking for all but last (and found it), return it
- **5174-5177**: try to get the next part of the path under the current inode
- **5179**: set current inode to be the found path part inode

5181-5184: if looking for all but last (and haven't found id before), return 0

5185: happily return our found inode

5052 dirlink(struct inode *dp, char *name, uint inum)

Writes a new directory entry (name, inum) into the inode pointed at by `dp`.

5059-5062: make sure name doesn't already exist in inode

5065-5070: find empty slot in inode

5072: prepare to write name

5073: prepare to write inum

5074: actually write the data (if we reached the end of the inode, `writei` will increase its size)

5657 create(char *path, short type, short major, short minor)

Creates and returns inode supplied in `path` (expecting `path` to exist up to its last part, which is the inode we're creating).
If inode already exists, opens it.

5663: get parent of requested inode

5667-5674: check if inode exists (and is a file!)
If so, return it.
If exists but is not file, return 0.

5676: get an actual inode from the disk

5679: lock the new inode (because we must)

5680-5682: set some values. nlink is 1, because parent is linked to new inode

5683: updates new values in the disk

5689: add the two links every directory has:

- "." self
- ".." parent

5693: add the new inode to parent

4654 iget(uint dev, uint inum)

Opens (and returns) inode.

4661-4670: loop over `icache`, looking if inode was already loaded once

- **4663-4667**: check if the inode is already in the cache
- **4668-4669**: find first empty slot, in case we'll need to load inode to it

4673-4674: if cache is full (and inode not there), panic (although we could have also gone to sleep till there's room)

4676-4680: load some of the inode data to `icache`. Valid bit is set to 0, because we haven't yet read the data from the disk

---

4689 idup(struct inode *ip)

Increments reference count of inode (and returns it).

4691: lock **`icache`** before use, because we access inode field

---

4756 iput(struct inode *ip)

Decrements reference count of inode, and closes it on the disk if this it will no longer be referenced at all.

4759: check if:

1. We're about to close the inode's last reference
2. We're closing a valid inode
3. There are no more names of / links to the inode (so it must be destroyed)

If so:

- **4761**: mark inode as busy (since there's still one reference!), just in case
- **4765**: actually delete inode on disk
- **4767**: update the inode (???)
- **4770**: do something that isn't really needed

4772: finally decrement the ref count

4603 ialloc(uint dev, short type)

Allocates a new inode *on the disk*, and then *in the memory*.

4610: read super-block from disk.

4612: loop over all inodes on disk:

- **4613**: read block of current inode
- **4614**: calculate pointer to current inode (using casting so that `+ inum%IPB` will add the correct size)
- **4615**: check if inode is free. If so:
    - **4616**: clear inode data
    - **4617**: set type
    - **4618**: re-write entire block, marking new inode as used (because we set type)
    - **4619**: close current block
    - **4620**: allocate inode in memory and return the *memory* inode
- **4622**: close current block

4624: no more inodes on disk?! Panic!

4629 iupdate(struct inode *ip)

Updates inode-on-disk from inode-on-memory `ip`.

4634: read entire block from disk

4635: calculate pointer to inode-on-disk (using casting so that `+ inum%IPB` will add the correct size)

4636-4641: set inode data to copy of block that is on the memory now

4642: re-write entire block on disk

4643: close block

4703 ilock(struct inode *ip)

Locks inode, without spinning or preventing interrupts.

4711-4715: sleep over inode until it's not busy:

- **4712-4713**: while inode is busy, go back to sleep
- **4714**: mark inode as busy

4717-4730: make sure inode is still valid - affects only in-memory inode

- **4718**: read entire block
- **4719**: calculate pointer to inode
- **4720-4725**: set data (just in case it changed meanwhile
- **4726**: release block
- **4727**: mark as valid

4735 iunlock(struct inode *ip)

Unlocks inode.

4741: remove "busy" flag

4742: wakeup all procs waiting for inode lock.

5751 sys_mkdir(void)

Creates a new directory.

5513 sys_link(void)

Creates a new name (or *shortcut*) for a file.
(But not for a directory!)

5601 sys_unlink(void)

Destroys a name of a file or directory.

4902 readi(struct inode *ip, char *dst, uint off, uint n)

Actually reads data from the disk.

4907-4911: (weird stuff beyond the scope of this course)

4913-4914: offset validation

4915-4916: uh...

4918: for each block from offset till block where we want to finish reading:

- **4919**: read entire current block
- **4920**: figure out how much to read (entire block, or just part)
- **4921**: copy the how-much-to-read data to memory (buffer)
- **4922**: close current block

4952 writei(struct inode *ip, char *src, uint off, uint n)

Actually reads data from the disk.

4957-4961: (weird stuff beyond the scope of this course)

4963-4966: validation

4968: for each block from offset will block where we want to finish writing:

- **4969** read entire current block (we actually don't really need to do this when we're writing a whole block, but whatever)
- **4970**: figure out how much to write (entire block, or just part)
- **4971**: copy the how-much-to-write data to memory (buffer)
- **4972**: copy buffer to disk
- **4972**: close current block

4976-4979: if the file grew because of the write, update inode's *size*

4856 itrunc(struct inode *ip)

Destroys inode on disk!
(Must be called only when inode is no longer referenced or held open by anyone.)

4862-4867: free direct blocks (if they're allocated) and mark them as such on inode

4869: if there are also indirect blocks:

- **4870**: read the block with the indirect pointers
- **4871**: cast block to int vector, for convenience

- **4872-4875**: free indirect blocks (if they're allocated) - no need to mark them on indirect vector, because we'll destory him soon
- **4876**: close the block with the indirect pointers
- **4877**: destory the block with the (now freed) indirect pointers
- **4878**: mark the indirect pointer as free on the inode

4881-4882: update inode

4810 bmap(struct inode *ip, uint bn)

Returns the physical block number of `ip`'s `bn`th block.
If block doesn't exist, the block is allocated.

4815-4819: handle case when block is direct

- **4816**: try to get physical address
- **4817**: if no physical address, allocate one

If we reached here, then we know block is indirect

4824-4825: allocate indirect block if it doesn't exist yet

4826: read contents of indirect block

4827: cast the contents as a vector of numbers

4828-4831: try to get physical address

- **4829**: allocate new address if needed
- **4830**: write new address on the indrect block on the disk

4832: close indirect block

4102 bread(uint dev, uint sector)

Gets a block from the disk.

4106: try to get buffer from cache

4107-4108: if buffer is not valid, ask for the actual buffer from the driver (which is the layer that *really* really reads from the disk

---

4114 bwrite(struct buf *b)

Writes a block to the disk.

4116-4117: make sure buffer is marked as busy

4118: mark buffer as dirty (that's our way to tell the driver to *write*)

4119: ask driver layer to write to disk.

4038 binit(void)

Initialize `bcache` buffer cache.

4066 bget(uint dev, uint sector)

Gets a buffer from the cache.
If it's not there, allocate it there.

4070: lock buffer cache

4072-4084: search for buffer in cache

- **4075**: buffer found!
    - **4076-4080**: if buffer is not busy, unlock the buffer cache, mark buffer as busy, and return it
    - **4081-4082**: if buffer is busy, go to sleep till it's unlocked (but will need to search again for case buffer was changed)

Got here?
Buffer not found; allocate new buffer

4087: loop from end of list to beginning

- **4088**: check if current buff is not busy and not dirty
    - **4089-4093**: fill data, release lock, return buff

4125 brelse(struct buf *b)

Release buffer from being BUSY and move to head of linked list.

4130: lock cache

4132-4137: reposition buff to list head

4139: mark buff as not busy

4140: wake up anyone who might be waiting for buff

4142: release cache lock

4454 balloc(uint dev)

Allocates and zeroes a block on the disk.

4461 read super block

---

bfree

---

idewait

---

idestart

3954 iderw(struct buf *b)

Handles a queue of blocks to write
(because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk).

3968-3971: append `b` to end of `idequeue`

3974-3975: if our `b` is at the head of the list, read/write it to IDE

3978-3980: sleep until `b` is VALID and NOT DIRTY

3902 ideintr(void)

Handles the interrupt from the IDE (which means the disk finished reading/writing).
Manages `idequeue`.

3907-3913: get current head of `idequeue`, and move `idequeue` to next guy.

- **3908-3912**: handle false alarms (which happen sometimes)
- **3913**: "increment" queue

3916-3917: read data (if that's what current buff wanted)

3920-3922: clear buff flags and wakeup all those waiting for buff

3925-3926: tell IDE to start running on next guy in queue

DAS BOOT

First things first: in order for a computer to run xv6, we need to load it from disk into memory and tell the processor to start running it. So how does this all happen?

The Boot Process

When you press the power button, the hardware gets initialized by a piece of firmware called the BIOS (Basic Input/Output System) that comes pre-installed on the motherboard on a ROM chip. Nowadays, your computer probably uses UEFI loaded from flash memory, but xv6 pretends like it's 1995 and sticks with BIOS. Since xv6 runs on x86 hardware, we're gonna have to satisfy all the janky requirements that come with that architecture, in addition to the BIOS's requirements.

Now the BIOS has to load some *other* code called the boot loader from disk; then it's the boot loader's job to load the OS and get it running. The boot loader has to act as a middle-man because the BIOS has no idea where on the disk you decided to put the OS.

The BIOS will look for the boot loader in the very first sector (512 bytes) of whatever mass storage device you told it to boot from, which we'll call the boot disk. The processor will execute the instructions it finds there. This means you have to make a choice: either your boot loader has to be less than 512 bytes or you can split it up into smaller parts and have each part load the next one. xv6 takes the first approach.

The BIOS loads the boot loader into memory at address 0x7C00, then sets the processor's `%ip` register to that address and jumps to it. Remember that `%eip` is the instruction pointer on x86? Okay cool. But why did I write `%ip` instead of `%eip`? Well, the BIOS assumes we're gonna be using 16 bits because of the hellscape known as backwards-compatibility, so we've gotta pretend like it's 1975 before we can pretend it's 1995. The irony here is that this initial 16-bit mode is called "real mode". So on top of loading the OS, the boot loader will also have to shepherd the processor from real mode to 32-bit "protected mode".

One last detail: we'll look at the Makefile and linker script later on, but for now just keep in mind that the boot loader will be compiled separately from the kernel, which will be compiled separately from all the user-space programs. This makes it easier to make sure that the entire boot loader will fit in the first 512 bytes on disk. Eventually, the boot loader and the kernel will be stored on the same boot disk together, and the user-space programs will be on a separate disk that holds the file system.

bootasm.S

Boot loader space is tight, and we want to make sure our instructions are exact, so we're gonna start off in assembly. The ".S" file extension means it's gonna be assembled by the GNU assembler `as`, and we're allowed to use C preprocessor directives like `#include` or `#define` or whatever in the assembly code. Also, xv6 uses AT&T syntax, so if you read CS:APP or took the online course then it'll be familiar; if you don't know what that means, then don't worry about it.

Getting Started

First we include some header files to use some constants; I'll point them out later. Next up, we gotta tell the assembler to generate 16-bit code, and set a global label to tell the BIOS where to start executing code.

```
.code16         # Tell compiler to generate 16-bit code
.globl start
start:
```

Next up: you know how sometimes you can press a special key to tell the BIOS to stop what it's doing and let you pick a disk to boot from? Or you move your mouse around in the BIOS menu and you see the pointer moving? Yeah, that needs hardware interrupts in order to work, but right now, we don't have the faintest clue how to handle those if they happen, so let's go ahead and turn those off. There's an x86 instruction to disable them by clearing the interrupt flag in the CPU's flags register.

```
    cli
```

Now we've gotta handle some of x86's quirks. First off, we're gonna need 20-bit memory addresses, but we only have 16 bits to work with. x86 uses six segment registers `%cs` (code segment), `%ds` (data segment), `%ss` (stack segment), `%es` (extra segment), `%fs` and `%gs` (general-purpose segments) to create 20- bit addresses from 16-bit ones; we're gonna need the first four. The BIOS guarantees that `%cs` will be set to zero, but it doesn't make any promises about the others, so we have to clear them ourselves. We're not using `%eax` for anything yet, so we'll use that to clear the others. The `w` at the end of `xorw` and `movw` means we're operating on 16-bit words.

```
    xorw    %ax,%ax
    movw    %ax,%ds     # Data segment
    movw    %ax,%es     # Extra segment
    movw    %ax,%ss     # Stack segment
```

This next part is a total hack for backwards-compatibility: sometimes a virtual address might get converted to a 21-bit physical address, and oh no, what are we gonna do? Well, some hardware can't deal with 21 bits, so it just ignores it, but it's 1995, so we've got fancy hardware that can use that extra bit. Wow, you really know we're in the future when you've got a whole 2 MB of RAM to work with! So we have to tell the processor not to throw away that 21st bit. The way we do that is by setting the second bit of the keyboard controller's output port to line high. I don't know. Don't ask me why. The output ports are 0x64 and 0x60, so we're gonna wait until they're not busy, then set the magic values that will make this all work.

```
seta20.1:
    inb     $0x64,%al   # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.1

    movb    $0xd1,%al   # 0xD1 -> port 0x64
    outb    %al,$0x64

seta20.2:
    inb     $0x64,%al   # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.2

    movb    $0xdf,%al   # 0xDF -> port 0x60
    outb    %al,$0x60
```

Segmentation

Now it's time to switch to 32-bit "protected mode". Up until now, the processor has been converting virtual addresses to physical ones using those segment registers which we cleared, so the mapping has been an identity map. But let's talk about how x86 converts 32-bit virtual addresses to physical ones; this is important for the rest of the boot loader code as well as the OS, so you're gonna have to bear with me for this maelstrom of x86-specific details.

The x86 architecture does the conversion in two steps: first segmentation, then paging. A virtual address starts off life as a *logical address*. Segmentation converts that to a *linear address*, and paging converts that to a physical one.

A logical address consists of a 20-bit *segment selector* and a 12-bit offset, with the segment bits before the offset bits, like `segment:offset`. The CPU's segmentation hardware uses those segment bits to pick one of those four segment registers we cleared earlier, which acts as an index into a *Global Descriptor Table* or GDT. Each entry of this GDT tells you where that segment is found in memory using a base physical address and a virtual address for the maximum or limit.

The GDT entry also has some permission bits for that segment; the segmentation hardware will check whether each address can be written to and whether the process generating the virtual address has the right permissions to access it. These checks compare the GDT entry's *Descriptor Privilege Levels*, also known as *ring levels*, against the *Current Privilege Level*. x86 has four privilege levels (0-3), so if you've ever heard of the kernel operating in ring 0 or user code in ring 3, this is where it comes from.

Okay, so the GDT entry will give us the first 20 bits of the new linear address; the offset bits stay the same. After that, the linear address is ready to be converted to a physical address by the paging hardware. We'll go over this second half of the story in the virtual memory section. For now, the point is this: xv6 is mostly gonna say no thank you to segmentation and stick to paging alone for memory virtualization.

So we're gonna set up our GDT to map all segments the exact same way: with a base of zero and the maximum possible limit (with 32 bits, that works out to a grand total of 4 GB, wow so much RAM, I can't imagine ever needing more). We have to stick this GDT somewhere in our code so we can point the CPU to it, so we'll put it at the end and throw a `gdtdesc` label on it. Now we can tell the CPU to load it up with a special x86 instruction for that.

```
    lgdt    gdtdesc
```

Protected Mode

Good news, everyone! We're finally ready to turn on protected mode, which we do by setting the zero bit of the `%cr0` control register. Note that the `l` at the end of the instructions here means we're now using long words, i.e. 32 bits; `CR0_PE` is defined in the mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) header file as 0x1.

```
    movl    %cr0, %eax      # Copy %cr0 into %eax
    orl     $CR0_PE, %eax   # Set bit 0
    movl    %ax, %cr0       # Copy it back
```

Oh wait, I lied. Enabling protection mode like we just did doesn't change how the processor translates addresses. We have to load a new value into a segment register to make the CPU read the GDT and change its internal segmentation settings. We can do that by using a long jump instruction, which lets us specify a code segment selector. We're just gonna jump to the very next line anyway, but in doing so we'll force the CPU to start using the GDT, which describes a 32-bit code segment, so *now* we're finally in 32-bit mode! Here, `SEG_KCODE` is a constant defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) as segment 1, for `%cs`; we bitshift it left by 3.

```
    ljmp    $(SEG_KCODE<<3), $start32
```

First we signal the compiler to start generating 32-bit code. Then we initialize the data, extra, and stack segment registers to point to the `SEG_KDATA` entry of the GDT; that constant is defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) as the segment for the kernel data and stack. We're not required to set up `%fs` and `%gs`, so we'll just zero them.

```
.code 32    # Tell assembler to generate 32-bit code now
start32:
    movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
    movw    %ax, %ds    # Data segment
    movw    %ax, %es    # Extra segment
    movw    %ax, %ss    # Stack segment
    movw    $0, %ax     # Zero the segments not ready for use
    movw    %ax, %fs
    movw    %ax, %gs
```

The Kernel Stack

Okay, last step in the assembly code now: we have to set up a stack in an unused part of memory. In x86, the stack grows downwards, so the "top" of the stack-- that is, the most-recently-added byte--is actually at the bottom of the stack in physical memory. It's annoying, but we're gonna have to keep track of that. The `%ebp` register points to the base of the stack (i.e., the first byte we pushed onto the stack), and the `%esp` register holds the address of the top of the stack (most-recently-pushed byte).

But where should we put the stack? The memory from 0xA_0000 to 0x10_0000 is littered with a memory regions that I/O devices are gonna be checking, so that's out. The boot loader starts at 0x7C00 and takes up 512 bytes, so that means it ends at 0x7E00. So xv6 is gonna start the stack at 0x7C00 and have it grow down from there, toward 0x0000 and away from the boot loader. Remember how back in the beginning, we started off the assembly code with a `start` label? That means that `start` is conveniently located at 0x7C00.

```
    movl    $start, %esp
```

And we're done with assembly! Time to move on to C code for the rest of the boot loader. We'll take over with a C function called `bootmain()`, which should never return. The linker will take care of connecting the call here to its definition in bootmain.c (https://github.com/mit-pdos/xv6-public/blob/master/bootmain.c).

```
    call    bootmain
```

Handling Errors

Wait, what? There's more assembly code after this? Why?

Well, if something goes wrong in `bootmain()`, then the function will return, so we have to handle that here. Since we usually run OSes we're developing in an emulator like Bochs or QEMU, we'll trigger a breakpoint and loop. Bochs listens on port 0x8A00, so we can transfer control back to it there; this wouldn't do anything on real hardware.

```
    movw    $0x8a00, %ax    # 0x8a00 -> port 0x8a00
    movw    %ax, %dx
    outw    %ax, %dx
    movw    $0x8ae0, %ax    # 0x8ae0 -> port 0x8a00
    outw    %ax, %dx
spin:
    jmp     spin            # loop forever
```

The Global Descriptor Table

Oh, and remember when we promised the hardware that we were gonna give it a GDT? We even told it to load it from address `gdtdesc`, remember? Well, we have to deliver on that promise now by defining the GDT here.

x86 expects that the GDT will be aligned on a 32-bit boundary, so we tell the assembler to do that. Then we use the macros `SEG_NULLASM` and `SEG_ASM` defined in asm.h (https://github.com/mit-pdos/xv6-public/blob/master/asm.h) to create three segments: a null segment, a segment for executable code, and another for writeable data. The null segment has all zeroes; the first argument to `SEG_ASM` has the permission bits, the second is the physical base address, and the third is the maximum virtual address. As we said before, xv6 relies mostly on paging, so we set the segments to go from 0 to 4 GB so they identity-map all the memory.

```
.p2align 2      # force 4-byte alignment
gdt:
    SEG_NULLASM                             # null segment
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code segment
    SEG_ASM(STA_W, 0x0, 0xffffffff)         # data segment

gdtdesc:
    .word   (gdtdesc - gdt - 1)     # sizeof(gdt) - 1
    .long   gdt                     # address of gdt
```

bootmain.c

Okay, the rest of the boot loader is in C now! Most of the code here is just to interact with the disk in order to read the kernel from disk and load it into memory. Let's start off by looking at `waitdisk()`.

waitdisk

```
void waitdisk(void)
{
    while ((inb(0x1F7) & 0xC0) != 0x40)
        ;
}
```

HEAD. DESK. Why all the magic numbers? At least we're lucky that the name makes it obvious what this function does; this won't always be true in xv6. Okay, so this function does only one thing: it loops until the disk is ready. Disk specs are boring as all hell, so feel free to skip to the next section if you don't care about the particulars (I don't blame you).

The usual way to talk to the disk is with Direct Memory Access (DMA), in which devices are hooked up directly to RAM for easy communication. But we haven't initialized the disk at all or set up any drivers for it; that's the OS's responsibility, not the boot loader's. Even if we could ask the disk to give us some data through memory-mapped I/O, we disabled all interrupts, so we wouldn't know when it's ready. So instead, we have to go back to assembly code (ugh, I know) to access the disk directly.

Storage disks have all kinds of standardized specifications, among them IDE (Integrated Drive Electronics) and ATA (Advanced Technology Attachment). The ATA specs include a Programmed I/O mode where data can be transferred between the disk and CPU through I/O ports. This is usually a huge waste of resources because every byte has to be transferred through a port and the CPU is busy the entire time, but right now beggars can't be choosers.

Each disk controller chip has two buses (primary and secondary) for use with ATA PIO mode; the primary bus sends data on port 0x1F0 and has control registers on ports 0x1F1 through 0x1F7. In particular, port 0x1F7 is the status port, which will have some flags to let us know what it's up to. The sixth bit (or 0x40 in hex) is the RDY bit, which is set when it's ready to receive more commands. The seventh bit (i.e., 0x80) is the BSY bit, which if set says the disk is busy.

Since interrupts are disabled, we'll have to manually poll the status port in an infinite loop until the BSY bit is not set but the RDY bit is: `inb()` is a C wrapper (defined in x86.h (https://github.com/mit-pdos/xv6-public/blob/master/bootmain.c)) for the x86 assembly instruction `inb`, which reads from a port. We don't care about any of the other status flags, so we'll get rid of them by bitwise-ANDing the result with 0xC0 = 0x40 + 0x80. If the result of that is 0x40, then only the RDY bit is set and we're good to go.

Phew. That was a lot for just one line of code.

readsect

```
void readsect(void *dst, uint offset)
{
    // Issue command
    waitdisk();
    outb(0x1F2, 1);
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20);

    // Read data
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}
```

If you skipped the last section: this function reads a sector (which in the current-year-according-to-xv6 of 1995 is 512 bytes) from disk. Good to see you again, on to the next section for you!

If you powered through the pain and read about ATA PIO mode above, some of the magic numbers here might be familiar. First we call `waitdisk()` to wait for the RDY bit, then we send some stuff over ports 0x1F2 through 0x1F7, which we know are the command registers for the primary ATA bus.

Note that `uint` is just a type alias for C's `unsigned int`, defined in the header file types.h (https://github.com/mit-pdos/xv6-public/blob/master/bootmain.c). The `offset` argument is in bytes, and determines which sector we're gonna read; sector 0 has to hold the boot loader so the BIOS can find it, and in xv6 the kernel will start on disk at sector 1.

`outb()` is another C wrapper for an x86 instruction from x86.h (https://github.com/mit-pdos/xv6-public/blob/master/x86.h); this one's the opposite of `inb()` because it sends data out to a port. The disk controller register at port 0x1F2 determines how many sectors we're gonna read. Ports 0x1F3 through 0x1F6 are where the sector's address goes. If you *really* must know (why?) they're the sector number register, the cylinder low and high registers, and the drive/head register, in order. Port 0x1F7 was the status port above, but it also doubles as the command register; we send it command 0x20, aka READ SECTORS.

Then we wait for the RDY bit again before reading from the bus's data register at port 0x1F0, into the address pointed to by `dst`. Once again, `insl()` is a C wrapper for the x86 instruction `insl`, which reads from a port into a string. The `l` at the end means it reads one long-word (32 bits) at a time.

readseg

```
void readseg(uchar *pa, uint count, uint offset)
{
    uchar *epa = pa + count;

    // Round down to sector boundary
    pa -= offset % SECTSIZE;

    // Translate from bytes to sectors; kernel starts at sector 1
    offset = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time. We'd write
    // more to memory than asked, but it doesn't matter -- we load in increasing
    // order.
    for (; pa < epa; pa += SECTSIZE, offset++) {
        readsect(pa, offset);
    }
}
```

Okay, finally, we're done with assembly and disk specs. We're gonna read `count` bytes starting from `offset` into physical address `pa`. Note that `uchar` is another type alias for `unsigned char` from types.h (https://github.com/mit-pdos/xv6-public/blob/master/types.h); this means that `pa` is a pointer (which is 32 bits in x86) to some data where each piece is 1 byte.

`epa` will point to the end of the part we want to read. Now, `count` might not be sector-aligned, so we fix that. Declaring `pa` as a `uchar *` lets us do this pointer arithmetic easily because we know that adding 1 to `pa` makes it point at the next byte; if it were a `void *` like in `readsect()`, pointer arithmetic would be undefined. (Actually, GCC lets you do it anyway, but GCC lets you get away with a lot of crazy stuff, so let's not go there.)

Now that we've got everything set up, we just call `readsect()` in a for loop to read one sector at a time, and that's it!

Some people have asked about the structure of some of the for loops in xv6, because they don't always use obvious index variables like `int i`. There are plenty of reasons to hate C, but I think the way it structures for loops is by far one of its most powerful features:

```
for (initialization; test condition; update statements) {
    code
}
```

When evaluating the for loop, C first executes anything in the initialization. Then it checks whether the test condition is true; if so, it executes the code inside the loop. Then it carries out the update statements before checking the test condition again and runnning the code if it's still true.

In the for loop above, the initialization is just an empty statement; all the variables we want to use have already been set up, so we don't need it and C will just move on to the next step. The test condition is simple enough. But the update statement actually increments both `pa` and `offset` at once before going through the loop again.

Okay great, so now we can read from the disk into memory, so we're all set up to load the kernel and start running it!

ELF Files

Before we move on to the star of the show, `bootmain()`, we need to talk about how a computer can actually recognize a file as executable. When you compile some code, the result gets spit out in a format that your machine can recognize, load into memory, and run; it's usually the linker's job to do this. Most Unix and Unix-like systems use the standardized Executable and Linkable Format, or ELF, for this purpose.

ELF divides the executable file into sections: `text` (the code's instructions), `data` (initialized global variables), `bss` (statically-allocated variables that have been declared but not initialized), `stab` and `stabstr` (debugging symbols and similar info), `rodata` (read-only data, usually stuff like string literals).

An ELF file starts with a header which has a magic number: 0x7F followed by the letters "ELF" represented as ASCII bytes; an OS can use this to recognize an ELF file. The header also tells you the file's type: it could be an executable, or a library to be linked with executables, or something else. There's a whole bunch of other info in the header, like the architecture it's made to run on, version, etc., but we're gonna ignore most of that.

The most important parts of the header are the part where it tells us where in the file the processor should start executing instructions and the part that describes the number of entries, on-disk offset, and size of the program header table.

The program header table is an array that has one entry for each of the file sections above that's found in this program. It describes the offset in the file where each section can be found along with the physical and virtual address at which that section should be loaded into memory and the size of the section, both in the file and in memory; these might differ if, e.g. the program contains some uninitialized variables which don't need to be stored in the file but do need to have space in memory.

The kernel (along with all the user-space programs) will be compiled and linked as ELF files, so `bootmain()` will have to parse the ELF header to find the program header table, then parse that to load each section into memory at the right address. xv6 uses a `struct elfhdr` and a `struct proghdr`, both defined in elf.h (https://github.com/mit-pdos/xv6-public/blob/master/elf.h), for this purpose.

Okay, back to the boot loader to finish up now!

bootmain

This is the C function that gets called by the first part of the boot loader written in assembly. Its job will be to load the kernel into memory and start running it at its entry point, a program called `entry()`.

Next up, we're gonna use `readseg()` to load the kernel's ELF header into memory at physical address 0x1_0000; the number isn't too important because the header won't be used for long; we just need some scratch space in some unused memory away from the boot loader's code, the stack, and the device memory-mapped I/O region. We'll read 4096 bytes first at offset 0; `readseg()` turns that offset into sector 1. Remember that we have to convert `elf` into a `uchar *` so that the pointer arithmetic in `readseg()` works out the way we want it to.

```
void bootmain(void)
{
    struct elfhdr *elf = (struct elfhdr *) 0x10000;
    readseg((uchar *) elf, 4096, 0);
    // ...
}
```

While we're at it, let's go ahead and make sure that what we're loading really is an ELF file and not some random other garbage because any of a million things went wrong during the compilation process, or we got some rootkit that totally corrupted the kernel or something. It's not really the most robust of checks, but *eh*. If something went wrong we'll just return, since we know that the code in `bootasm.S` is ready to handle that with some Bochs breakpoints.

```
void bootmain(void)
{
    // ...
    if (elf->magic != ELF_MAGIC) {
        return;
    }
    // ...
}
```

Now we have to look at the program header table to know where to find each of the kernel's segments. The `elf->phoff` field tells us the program header table's offset from the start of the ELF header, so we'll set `ph` to point to that and `eph` to point to the end of the table.

```
void bootmain(void)
{
    // ...
    struct proghdr *ph = (struct proghdr *) ((uchar *) elf + elf->phoff);
    struct proghdr *eph = ph + elf->phnum;
    // ...
}
```

Each entry in the program header table tells us where to find a segment, so we'll iterate over the entries, reading each one from disk and loading it up. In this for loop, note that `ph` is a `struct proghdr *`, so incrementing it with `ph++` increments it by the size of a `struct proghdr` and not by one byte; this makes it automatically point at the next entry in the table.

```
void bootmain(void)
{
    // ...
    for (; ph < eph; ph++) {
        uchar *pa = (uchar *) ph->paddr;    // address to load section into
        readseg(pa, ph->filesz, ph->off);   // read section from disk

        // Check if the segment's size in memory is larger than the file image
        if (ph->memsz > ph->filesz) {
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
        }
    }
    // ...
}
```

That if statement at the end checks if the section's size in memory should be larger than its size in the file, in which case it calls `stosb()`, which is yet another C wrapper from x86.h (https://github.com/mit-pdos/xv6-public/blob/master/x86.h) for the x86 instruction `rep stosb`, which block loads bytes into a string. It's used here to zero the rest of the memory space for that section. Okay, but why would we want to do that? Well, if the reason it's larger is because it has some uninitialized static variables, then we want to make sure those start off holding zero (as the C standard requires) and not whatever garbage value may have been there before.

Last part of the bootloader: let's call the kernel's entry point, `entry()`, and get it running! But remember how the boot loader is compiled and linked separately from the kernel? Yeah, that means we can't just call `entry()` as a function, because then the linker would go "Huh? What entry function? I don't have any `entry` function here in your symbol table. REJECTED." And then it would throw a huge error.

Luckily, the ELF header tells us where to find the entry point in memory, so we could get a pointer to that address. That means... function pointers! If you've never used function pointers in C before, then this won't be the last time you'll see them in xv6, so check it out.

A C function is just a bunch of code to be executed in order, right? That means it shows up in the ELF file's `text` section, which will end up in memory. When you call a regular old C function, the compiler just adds some extra assembly instructions to throw a return address on the stack and update the registers `%ebp` and `%esp` to point to the new function's stack on top of the old one. If the function getting called has any arguments or local variables, they'll get pushed onto the stack too. Then the instruction register `%eip` gets updated to point to the new function section, and that's it. After the compiler is done, the linker will replace the function's name with its memory address in the `text` section, and voila, a function call.

The point of all this is that in C we can use pointers to functions; they just point to the beginning of that function's instructions in memory, where the `%eip` register would end up pointing if the function gets called. So in this case, even though we're not linking with the kernel, we can still call into the entry point by getting its address from the ELF header, creating a function pointer to that address, then calling the function pointer. The compiler will still add all the usual stack magic, but instead of the linker determining where `%eip` should point, we'll do that ourselves.

The first line below declares `entry` as a pointer to a function with argument type `void` and return type `void`. Then we set `entry` to the address from the ELF header, then we call it.

Again, this shouldn't return, but if it does then it's the last part of this function, so this function will return back into the assembly boot loader code.

```
void bootmain(void)
{
    // ...
    void (*entry)(void);
    entry = (void(*) (void)) (elf->entry);
    entry();
}
```

That's it! Starting from `entry()`, we're officially out of the boot loader and into the kernel.

Summary

To summarize, the assembly part of the boot loader (1) disabled interrupts, (2) set up the GDT and segment registers so the segmentation hardware is happy and we can ignore it later, (3) set up a stack, and (4) got us from 16-bit real mode to 32-bit protected mode.

Then the C part of the boot loader just loaded the kernel from disk and called into its entry point.

ELF headers will continue to haunt us in the kernel's linker script and when we load user programs from disk in `exec()`, and function pointers will make another appearance when we get around to handling interrupts. The good news: the boot loader is one of the most opaque parts of the xv6 code, full of boring hardware specs and backwards-compatibility requirements, so if you made it this far, it does get better!

(But it also gets worse... looking at you, mp.c (https://github.com/mit-pdos/xv6-public/blob/master/mp.c) and kbd.c (https://github.com/mit-pdos/xv6-public/blob/master/kb.c)...)

Devices: Disk Driver

At this point, we've seen how xv6 virtualizes memory and the processor to give each user process the illusion of a contiguous, near-infinite memory space and a dedicated CPU to run it; we've also seen how xv6 mediates interactions between most of a computer's hardware components and user processes via system calls. But there's one more piece of hardware that's critically important for an OS that we haven't looked at yet: the disk. All that's left in the kernel code for us to look at is how xv6 manages data storage on the disk and how it presents that data to users in a simplified way.

The function of a disk is to provide *persistence* for an operating system. RAM is volatile memory: it gets erased when the machine is turned off, so any data stored there is fleeting. A disk allows an OS to store and retrieve data across shut-offs. The disk driver we'll go over in this post allows the xv6 kernel direct access to that device so it can read and write data to it.

But unlike other devices, a simple driver isn't enough here. We don't just need to be able to read and write data; we'd like to present users with a simplified, accessible framework to navigate that data. Imagine using a computer where you had to specify which byte of the disk to read or write, then remember that yourself in order to access it again later. It's madness! Enter file systems; "files" don't really exist in any real sense on a disk, but the OS can provide the illusion of discrete, individual files in order to simplify access to data.

We also need to make sure concurrent accesses of the same file don't risk corrupting the file (or even the entire file system). We need to separate out kernel data (like the kernel code itself) from user data on the disk, so that a malicious user process can't just overwrite arbitrary kernel code. Finally, there's that oh-so-famous line about Unix systems, "everything is a file". We'll need a way to present "everything" in the elegant abstraction of a file.

All of these abstractions and security checks will require far more code than a simple driver to implement them, so before we go on to the driver, let's check out how xv6 will organize its file system to get a preview of what's ahead.

File System Organization

Laying the abstraction of a complete file system on top of a physical disk will require several steps. xv6 does this using seven layers. From bottom (direct hardware interaction) to top (user-facing code), they are:

- Disk driver: reads and writes blocks on an IDE hard drive.
- Buffer cache: caches disk blocks in memory and synchronizes access to them.
- Logging: provides atomic disk writes to mitigate the risk of a crash.
- Inodes: turns disk blocks into individual files that the OS can manipulate.
- Directories: creates a tree of named directories that contain other files.
- Path names: provides hierarchical, human-readable path names in the directory tree structure.
- File descriptors: abstracts OS resources like pipes and devices as files to provide a unified API for user programs.

That's a lot of work to do now, but it'll pay off! The kernel will do all this labor so that users are free to be lazy later on and can live in blissful ignorance of the fact that their precious little files actually exist as nothing but ones and zeroes in totally arbitrary locations on the disk.

Note that hard drives are usually divided into *sectors*, which are physical divisions (originally referring to literal geometric sectors), traditionally of 512 bytes. Operating systems can then collect these into larger *blocks* which are multiples of the sector size. xv6 uses 512-byte blocks for simplicity so that the sector and block sizes match up; I'll use the two terms interchangeably.

On the disk, block 0 usually contains the boot sector, so it's not used by xv6 (but remember the Makefile -- xv6 actually stores the boot loader and kernel code on an entirely separate physical disk). Block 1 is called the *superblock* because it contains metadata about the file system like its total size, the size of the log, the number of files, and their location on the disk. Then the log starts at block 2 and on.

buf.h

If you've read any of the previous optional posts on device drivers, you know that interacting directly with the hardware means all kinds of opaque code with seemingly-arbitrary port I/O and cryptic magic numbers. Drivers are also specific to the actual (or virtual) hardware in the machine that xv6 will run on, so it tends to be less useful for showing general OS concepts -- hence why all the other device driver posts were optional. That being said, the disk driver nicely rounds out the rest of the file system code, so I recommend checking it out, but if you're short on time or bored with all the talk about hardware specs, feel free to skip to the summary section below.

Reading and writing disk data is super slow, so the second layer in the file system is the buffer cache, which will store copies of disk blocks in memory for faster access. But we still have to read from the disk to create that buffer, and we still have to write any modified data to the disk once we're done, so we still need a layer below the buffer cache to do that. That layer is the disk driver; its purpose is to copy data from the disk to the in-memory cache and vice versa. A single block is represented in the cache as a `struct buf`, defined in buf.h (https://github.com/mit-pdos/xv6-public/blob/master/buf.h).

```
struct buf {
    int flags;
    uint dev;               // device number
    uint blockno;           // block number (same as sector number)
    struct sleeplock lock;  // sleep-lock to protect buffer reads and writes
    uint refcnt;            // how many processes are using this buffer
    struct buf *prev;       // for use with buffer cache doubly-linked list
    struct buf *next;       // for use with buffer cache doubly-linked list
    struct buf *qnext;      // for use with disk driver queue
    uchar data[BSIZE];      // data stored in the buffer
};

#define B_VALID 0x2
#define B_DIRTY 0x4
```

The two constants defined at the bottom are used in the `flags` field; `B_VALID` indicates that a buffer has been read from disk and should accurately reflect the sector's contents on the disk, and `B_DIRTY` says we've modified the buffer but haven't yet updated the on-disk version of a file, so we need to write the buffer to disk soon.

We'll see later on that the buffer cache uses a doubly-linked list of buffers; the `prev` and `next` fields are used there. However, the disk driver also maintains its own queue of buffers that are waiting to be read from or written to the disk; that's implemented as a singly-linked list using the `qnext` field.

ide.c

We've already seen some code to read and write disk data in the boot loader (boot.md); I know it's been a while, so you can check that out again if you want. We can't reuse the code there for a few reasons, though: (1) the boot loader has to be compiled separately from the kernel, so we can't access any of the functions there, and (2) we need to store data in the buffer cache, so we can't even copy- paste the code we used before since the boot loader barely even knows what memory is, let alone a buffer cache.

ATA Programmed I/O Mode

Modern disk drivers usually talk to the disk via direct memory access (DMA), but to keep things simple xv6 is just gonna talk to it with port I/O. That's much, much slower, and it requires active participation by the CPU (which means it can't do anything else at the same time), but hey, xv6 thinks it's 1995, remember? So PIO mode is still (relatively) cutting edge. Either way, extreme performance isn't the goal here, so we'll just have to suck it up.

Okay, let's do a super-quick summary. `inb` is a C wrapper for an x86 assembly instruction that reads a single byte of data from a port; `outb` writes a byte to a port. The disk controller chip has primary and secondary buses; the primary bus sends data on port 0x1F0 and has control registers on ports 0x1F1 through 0x1F7. Port 0x1F7 doubles as a command register and a status port with some useful flags we can check in order to know what the disk is up to; we saw some of those before, but I'll give you the full list now.

- Bit 0 (0x01) - ERR (indicates an error occurred)
- Bit 1 (0x02) - IDX (index; always set to zero)
- Bit 2 (0x04) - CORR (corrected data; always set to zero)
- Bit 3 (0x08) - DRQ (drive has data to transfer or is ready to receive data)
- Bit 4 (0x10) - SRV (service request)
- Bit 5 (0x20) - DF (drive fault error)
- Bit 6 (0x40) - RDY (ready; clear when drive isn't running or after an error and set otherwise)
- Bit 7 (0x80) - BSY (busy; drive is in the middle of sending/receiving data)

The disk driver defines some of these with preprocessor macros at the top of the file.

```
#define SECTOR_SIZE 512
#define IDE_BSY     0x80
#define IDE_DRDY    0x40
#define IDE_DF      0x20
#define IDE_ERR     0x01
// ...
```

We also saw one command example in the boot loader: sending 0x20 to port 0x1F7 tells the disk to read a sector and send it to us through data port 0x1F0. Now we'll also use commands to write a sector, as well as to read or write multiple sectors at once.

```
// ...
#define IDE_CMD_READ    0x20
#define IDE_CMD_WRITE   0x30
#define IDE_CMD_RDMUL   0xc4
#define IDE_CMD_WRMUL   0xc5
// ...
```

If, for some reason beyond mortal comprehension, you decide you want to know more about the eldritch secrets of ancient hard drives, you can read this resource on ATA disks (https://pdos.csail.mit.edu/6.828/2018/readings/hardware/ATA-d1410r3a.pdf).

After those constants, we find three static global variables: a spin-lock for accessing the disk, the queue of buffers waiting to be synchronized with their on-disk counterparts, and a boolean to track whether xv6 is running with only disk 0 (boot loader and kernel) or with disk 1 (user file system) as well.

```
// ...
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
// ...
```

idewait

This function takes an integer `checkerr` argument that should be a boolean and waits for the disk to be ready to receive more commands. If `checkerr` is true, it'll also check whether the status port includes any error flags.

It starts by reading from the disk's status port and looping until the busy flag is not set but the ready flag is. The bitwise-OR `IDE_BSY | IDE_DRDY` combines both flags, and the bitwise-AND tests whether either one is set in `r`.

```
static int idewait(int checkerr)
{
    int r;
    while (((r = inb(0x1f7)) & (IDE_BSY | IDE_DRDY)) != IDE_DRDY)
        ;
    // ...
}
```

Now if `checkerr` is nonzero we have to check that neither the error nor the drive failure flag is set in the status port. If either one is set, we'll return -1; we'll return 0 otherwise.

```
static int idewait(int checkerr)
{
    // ...
    if (checkerr && (r & (IDE_DF | IDE_ERR)) != 0) {
        return -1;
    }
    return 0;
}
```

ideinit

This function is called by the kernel's `main()` during set-up to initialize the disk. We start by initializing the disk lock, then tell the I/O interrupt controller to forward all disk interrupts to the last CPU. We talked about the `ioapicenable()` function in detail in the post on interrupt controllers.

```
void ideinit(void)
{
    initlock(&idelock, "ide");
    ioapicenable(IRQ_IDE, ncpu - 1);
    // ...
}
```

Then we wait for the disk to be ready to accept commands (ignoring any error flags that may be present).

```
void ideinit(void)
{
    // ...
    idewait(0);
    // ...
}
```

We said above that disk 0 should contain the boot loader and kernel, so we can assume any machine running xv6 should have that present. However, we need to make sure disk 1 is present; the Makefile (https://github.com/mit-pdos/xv6-public/blob/master/Makefile) includes some configurations like `make qemu-memfs` under which xv6 can run without a dedicated disk for the file system, storing files in memory instead.

Port 0x1F6 is used to select a drive. Bits 5 and 7 should always be set, and bit 6 picks the right mode we need to indicate a disk. Bit 4 determines whether we want to select disk 0 or disk 1. So we can select drive 1 by setting bits 5-7 (0xE0 when combined), then bit 4 (`1 << 4`).

```
void ideinit(void)
{
    // ...
    outb(0x1f6, 0xe0 | (1 << 4));
    // ...
}
```

Now we need to wait for disk 1 to be ready; we need to handle this as a special case since `waitdisk()` can't check a specific disk for us, and because an absent disk 1 would make the while loop there continue forever. So we'll check the status register 1000 times; if it ever reports that it's ready, we'll set `havedisk1` to true and break, but otherwise we'll assume disk 1 isn't present and leave `havedisk1` as zero (i.e., false).

```
void ideinit(void)
{
    // ...
    for (int i = 0; i < 1000; i++) {
        if (inb(0x1f7) != 0) {
            havedisk1 = 1;
            break;
        }
    }
    // ...
}
```

Finally, we'll switch back to using disk 0 by changing the fourth bit of the register at port 0x1F6.

```
void ideinit(void)
{
    // ...
    outb(0x1f6, 0xe0 | (0 << 4));
}
```

idestart

This is the core function that will read or write a buffer to or from the disk. It's a `static` function, so it can only be called by other functions in this file; `ideintr()` and `iderw()` will both use it as a helper function. It takes a pointer to a buffer, so the first thing to do is make sure that pointer isn't null. We'll also make sure the buffer's block number is within the maximum limit set by `FSSIZE`, defined in param.h (https://github.com/mit-pdos/xv6-public/blob/master/param.h) as 1000.

```
static void idestart(struct buf *b)
{
    if (b == 0) {
        panic("idestart");
    }
    if (b->blockno >= FSSIZE) {
        panic("incorrect blockno");
    }
    // ...
}
```

Next we need to figure out which disk sector to read from or write to. Since xv6 uses blocks that are the same size as a sector, this should just be `b->blockno`, but we'll add a conversion here in case that gets changed later on (especially if we want higher disk throughput).

```
static void idestart(struct buf *b)
{
    // ...
    int sector_per_block = BSIZE / SECTOR_SIZE;
    int sector = b->blockno * sector_per_block;
    // ...
}
```

If each block fits exactly one sector, then we'll need to use the single-sector read and write commands; otherwise we should use the multi-sector versions of those commands. We'll set `read_cmd` and `write_cmd` to the right versions. We'll also make sure that there are no more than 7 sectors per block.

```
static void idestart(struct buf *b)
{
    // ...
    int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ : IDE_CMD_RDMUL;
    int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
    if (sector_per_block > 7) {
        panic("idestart");
    }
    // ...
}
```

Now let's wait for the disk to be ready, ignoring any error flags.

```
static void idestart(struct buf *b)
{
    // ...
    idewait(0);
    // ...
}
```

Okay, now it's time to brace yourself, because this next part is a hot mess of port I/O operations with lots of magic numbers. First we'll tell the disk controller to generate an interrupt once it's done reading or writing by setting the device control register at 0x3F6 to zero. Then we'll tell it how many total sectors we want to read or write by writing that number (AKA `sector_per_block`) to port 0x1F2.

```
static void idestart(struct buf *b)
{
    // ...
    outb(0x3f6, 0);                 // generate interrupt when done
    outb(0x1f2, sector_per_block);  // number of sectors to read/write
    // ...
}
```

Before sending the read or write command, we have to tell the disk which sector to read from, using our `sector` variable from above. Let's take a second to talk about hard drive geometry. A hard drive consists of a bunch of stacked circular surfaces, where each surface has a corresponding *head* that changes its position to read or write from the right place on the disk. Each surface has a number of *tracks*: concentric circles that contain data. If you pick a track number (i.e. pick a distance from the center of the surfaces) and collect all those tracks from all the surfaces, you get a *cylinder*.

A sector number acts as a kind of address with each part specifying a different geometric component, similar to how linear addresses contain a page directory index, page table index, and offset. The eight most significant bits (24 through 31) identify the drive and/or head that the sector is located on (plus some flags); bits 8 through 23 identify the cylinder,

and bits 0 through 7 pick a sector within that cylinder. Altogether, these define a 3D coordinate system that uniquely identifies all sectors on a machine's disks.

Port 0x1F3 is the sector number register, ports 0x1F4 and 0x1F5 are the cylinder low and high registers, and port 0x1F6 is the drive/head register. We can write the sector number as `sector & 0xFF`; the cylinder low and high numbers can be recovered by bitshifting `sector` down by 8 and 16, respectively.

```
static void idestart(struct buf *b)
{
    // ...
    outb(0x1f3, sector & 0xff);             // sector number
    outb(0x1f4, (sector >> 8) & 0xff);      // cylinder low
    outb(0x1f5, (sector >> 16) & 0xff);     // cylinder high
    // ...
}
```

Now for the drive/head register, we'll use `b->dev` to get the block's device and `(sector >> 24)` to get the head it's on. Finally, we'll set bits 5-7 as required (and as mentioned above in `ideinit()`) with 0xE0. Then we can bitwise-OR all of these together and write them to port 0x1F6.

```
static void idestart(struct buf *b)
{
    // ...
    outb(0x1f6, 0xe0 | ((b->dev & 1) << 4) | ((sector >> 24) & 0x0f));
    // ...
}
```

Okay, that was the worst of it! Deep breath now. The last part is just sending the actual read or write command. But how do we know which one we're supposed to do? The only argument is a pointer to a buffer `b`, not any sort of boolean that might tell us which to carry out. Well, remember the buffer flag `B_DIRTY`? That one indicates that a buffer has been modified and needs to be written to disk. If that flag is set, reading from the disk would overwrite any changes, which probably isn't what we want. So let's just assume that the `B_DIRTY` flag means we should write to disk, and the absence of that flag means we should read from disk.

```
static void idestart(struct buf *b)
{
    // ...
    if (b->flags & B_DIRTY) {
        outb(0x1f7, write_cmd);
        outsl(0x1f0, b->data, BSIZE / 4);
    } else {
        outb(0x1f7, read_cmd);
    }
}
```

Here `outsl()` is another C wrapper for an x86 instruction; this one writes data from a string, four bytes at a time.

That's it! This is by far the most cryptic function in the disk driver; the last two are relatively easy now.

ideintr

We saw in `idestart()` that we set up the disk to send an interrupt whenever it's done reading or writing data. Back when we looked at trap.c (https://github.com/mit-pdos/xv6-public/blob/master/trap.c), we saw that the `trap()` function directs all disk interrupts to the handler function `ideintr()`. It's time to check that one out now.

We'll start by acquiring the disk's spin-lock; note that we don't use a sleep- lock because this is an interrupt handler function, so interrupts should be disabled while it runs.

```
void ideintr(void)
{
    acquire(&idelock);
    // ...
    release(&idelock);
}
```

If we got an interrupt, then it usually means the disk is done with the most recent request. Those requests are stored in the global `idequeue` linked list, with the current request at the front of the queue. So we'll get the head of the queue as `b`, then set `idequeue` to point to the next buffer in the queue. If the head is null, then we'll just return early.

```
void ideintr(void)
{
    // ...
    struct buf *b;
    if ((b = idequeue) == 0) {
        release(&idelock);
        return;
    }
    idequeue = b->next;
    // ...
}
```

The read command in `idestart()` didn't specify where to read the data to, so we do that now. We'll check if the `B_DIRTY` flag was set; if it wasn't (i.e. the operation was a disk read), then we'll wait for the disk to be ready (without any errors, using `idewait(1)` instead of `idewait(0)` as we have before) and read the data into `b->data`.

```
void ideintr(void)
{
    // ...
    if (!(b->flags & B_DIRTY) && idewait(1) >= 0) {
        insl(0x1f0, b->data, BSIZE / 4);
    }
    // ...
}
```

Next, we set the `B_VALID` flag with a bitwise-OR and clear any `B_DIRTY` flag with a bitwise-AND and a bitwise-NOT. Then we'll wake up any user process that went to sleep on a channel for this buffer after requesting a disk I/O operation.

```
void ideintr(void)
{
    // ...
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b);
    // ...
}
```

Finally, we'll get the disk started on the next operation, for the next buffer in the queue.

```
void ideintr(void)
{
    // ...
    if (idequeue != 0) {
        idestart(idequeue);
    }
    // ...
}
```

iderw

The `idestart()` function is `static`, so it can't be called by anything outside of this file; we need to provide a mechanism for both kernel and user threads to read and write disk data. That's what `iderw()` does. Note that processes should never call this function directly; it only gets called by the code for the buffer cache layer of the file system. In other words, processes will use system calls like `open()`, `read()`, `write()`, `close()`, etc., which in turn will use functions from higher layers of abstraction, which in turn call functions from lower layers, and so on, until they reach the buffer cache, which calls `iderw()` to finally read/write directly from/to the disk.

By the time a process gets to `iderw()`, it should already be holding a sleep- lock `b->lock` for the buffer `b` it wants to read or write, and either the `B_DIRTY` flag should be set (to write to disk) or the `B_VALID` flag should be absent (to read from disk). We'll start off with some sanity checks for those, and make sure that we're not trying to read from disk 1 if it's not present on this machine. Then we'll acquire the disk's spin-lock.

```
void iderw(struct buf *b)
{
    if (!holdingsleep(&b->lock)) {
        panic("iderw: buf not locked");
    }
    if ((b->flags & (B_VALID | B_DIRTY)) == B_VALID) {
        // B_VALID is set, so we don't need to read it; B_DIRTY is not set, so
        // we don't need to write it
        panic("iderw: nothing to do");
    }
    if (b->dev != 0 && !havedisk1) {
        panic("iderw: ide disk 1 not present");
    }

    acquire(&idelock);
    // ...
    release(&idelock);
}
```

There may be other buffers waiting in line in the disk queue, so we have to append this buffer `b` to the end of `idequeue`. We can do that by setting `b->qnext` to null, then creating a variable `pp` to traverse the entire queue. When `pp` points to the last element, we'll set its `qnext` field to point to `b`.

```
void iderw(struct buf *b)
{
    // ...
    b->qnext = 0;

    // Traverse the queue
    struct buf **pp;
    for (pp = &idequeue; *pp; pp = &(*pp)->qnext)
        ;

    // Append b to end of queue
    *pp = b;

    // ...
}
```

That traversal might look confusing as all hell, so let's take a closer look. It defines `pp` as a double pointer: a pointer to a pointer to a `struct buf`. (If you've seen the interview of Linus Torvalds where he talks about good style with linked lists, it's similar to the code there; there's a nice summary here (https://github.com/mkirchner/linked-list-good-taste).) `pp` starts off equal pointing to `idequeue`, i.e. the head of the linked list. Each iteration checks that `pp` points to a valid (non-null) pointer, i.e. the loop will end when we reach the end of the list. The body of the loop is empty, so none of the iterations actually do anything; the purpose of the for loop is just to update `pp` several times. At the end of each iteration, `pp` is updated to point to a pointer to the next buffer in the queue.

Suppose the last buffer in the queue is `end`. At the end of the for loop, `pp` will hold the address of `end->qnext`, so `*pp = b` sets `end->qnext = b`. The double indirection makes it easy to update the last buffer in the queue; without it, we would have to stop the loop one step earlier when `pp` points to `end` instead of `end->qnext` then be careful to update the actual buffer at the end of the queue instead of just updating the local variable `pp`. All in all, it's just an elegant way to write a linked list traversal in a single line.

Okay, so now our buffer `b` is at the end of the queue. If there are others in front of it, then `ideintr()` will make sure that each disk interrupt starts the disk on the next operation. But what if `b` is actually the only buffer in the queue? In that case, the disk isn't running yet, so we need to get it started ourselves.

```
void iderw(struct buf *b)
{
    // ...
    if (idequeue == b) {
        idestart(b);
    }
    // ...
}
```

At this point, we can be confident that the disk will either start our request now or get to it eventually (if there are other requests in the queue). This process just has to wait for the disk to finish, so we'll put it to sleep until the buffer has been synchronized with the disk. We'll check that by making sure the `B_VALID` flag is present but `B_DIRTY` is not set. The call to `sleep()` will release `idelock` and reacquire it before returning.

```
void iderw(struct buf *b)
{
    // ...
    while ((b->flags & (B_VALID | B_DIRTY)) != B_VALID) {
        sleep(b, &idelock);
    }
    // ...
}
```

Summary

The disk driver handles direct communication with the hard drive, issuing orders to read or write sectors. It exposes two API functions, `ideintr()` and `iderw()`. The former is called by `trap()` to handle disk interrupts, while the latter is called by the code for the buffer cache layer of the file system to update blocks in the buffer cache with their corresponding sectors on disk. Next up we'll look at the buffer cache itself, as well as the logging layer, which provides crash recovery.

The Beginning: Entry and Paging

xv6's Memory Layout

The whole point of virtualizing memory is to give users the illusion that they can roam freely across a limitless field of memory without worrying their pretty little heads about such boring details as how much physical memory their machine actually has, or where kernel code is stored, or the fact that their seemingly- continuous heap space is actually shattered into tons of tiny pages spread out in possibly random parts of physical memory. As long as user code is well- behaved, that illusion should hold up; if they do a no-no we'll just smack them with a segmentation fault.

One downside is that the kernel also has to use virtual memory, so we're faced with the potentially-complicated challenge of setting things up in physical memory without knowing where anything is actually located in physical memory! So xv6 does something that a lot of OSes do: it sets itself up as a higher-half kernel. That means that in the virtual address space (from 0 to 4 GB), the kernel will reside in the upper half starting at 2 GB, i.e. address 0x8000_0000 and up; user code will start at 0 and end at 2 GB. Because of this, `KERNBASE` is defined in memlayout.h (https://github.com/mit-pdos/xv6-public/blob/master/memlayout.h) as 0x8000_0000.

Then it sets up paging so that all of physical memory is identity-mapped to virtual memory starting at 0x8000_0000. This makes it really convenient for the kernel to figure out the physical address of a virtual address it's using; just subtract `KERNBASE` and you're done. The `V2P` and `V2P_WO` macros defined in memlayout.h (https://github.com/mit-pdos/xv6-public/blob/master/memlayout.h) do just that, and the `P2V` and `P2V_WO` add `KERNBASE` to a physical address to get the kernel virtual address.

Note that I said "kernel virtual address", not just any old virtual address. Users don't get these kinds of fancy privileges, because they shouldn't be worrying about where anything is in physical memory. They're running through a limitless field of virtual memory, remember? So user virtual addresses between 0 and 2 GB will get mapped to totally arbitrary locations in physical memory.

One consequence of this is that xv6 is limited to no more than 2 GB of physical memory (instead of the 4 GB that 32-bit addresses allow for) in order to map it all into the top 2 GB of virtual memory. In reality, it's even less, for two reasons: (1) we also need to map device I/O regions into virtual memory, so it'll be a little less than 2 GB, and (2) it's hard and annoying to figure out how much physical memory is actually present on any given machine, so xv6 just says to hell with all that and picks the totally arbitrary value of a puny 224 MB as the amount of available physical memory (that's `PHYSTOP`, defined in memlayout.h (https://github.com/mit-pdos/xv6-public/blob/master/memlayout.h)).

Paging

Remember when we talked about segmentation, and how we said we'd come back to paging later? Guess what? It's later.

So all virtual addresses are really "logical addresses", and segmentation turns those into "linear addresses". In xv6, the boot loader set up the segmentation hardware to use an identity map, so virtual addresses are the same as logical addresses are the same as linear addresses. Now paging has to turn those linear addresses into physical addresses. Just like segmentation uses a GDT and the segment registers for its mapping, paging uses a page directory, page tables, and the `%cr3` register.

First, imagine a world where every single time some user code throws up an address (maybe it looks up a variable, or it calls a function, or it simply needs to execute the next instruction), the CPU has to stop what it's doing, save all the user's register contents, load up some kernel code, restore its register contents, find out where its stack is, get it running, and

then ask the OS where that virtual address is actually located in physical memory. That would be *so* slow. We don't want that. We want the hardware to do all the address conversions by itself, and involve the OS only minimally to set up a new page directory when it starts a new process.

Instead, the x86 hardware uses one of its control registers, `%cr3`, to store a pointer to a page directory in memory. Then every time it needs to map a linear address to a physical one, it goes to that page directory and grabs the relevant entry. That entry is a pointer to a page *table* somewhere else in memory, so the processor grabs the right entry from there, which points to a 4096-byte page in some other location.

A linear address has a three-part structure: the 10 most significant bits are an index that picks an entry from the page directory, the next 10 bits are an index to pick an entry from whatever page table we've been directed to, and the last 12 bits are an offset that determines where to look in the page that the page table entry pointed to.

For example, let's say we have a virtual address like 0x9C4A_02BF. If we convert to binary, split it up, and convert back to hex, we can see that the 10 most significant bits are 0x271, the next 10 are 0x0A0, and the last 12 are 0x2BF. So the paging hardware would look at wherever `%cr3` is pointing to find the page directory; let's just call it `pgdir`. Then it would take entry `pgdir[0x271]` and go look wherever that's pointing to find the right page table; let's call that `pgtab271`. Then it would take entry `pgtab271[0x0A0]` and look wherever that's pointing to find the right page, `pg`. *Then* it would finally know that the corresponding physical address is `pg + 0x2BF`. Whew.

This still sounds super slow, so the paging hardware uses a cache called the Translation Lookaside Buffer (TLB) to store recently-used mappings and make them faster in the future. Since pages are 4096 bytes, it only needs to map a new page if the addresses some code is asking for crosses a page boundary.

xv6 provides two macros, `PDX` and `PTX` defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h), to recover just the page directory index bits or the page table index bits, respectively, from a virtual address.

Finally: an important aspect of virtual memory is that each process should be isolated from the others, and the kernel should be isolated from user processes. So each process will get its own page directory, and each entry of that page directory will say whether it's present (i.e., mapped) or not. If it's present, then it points to a page table for that process; if it's not present and we try to access it, we'll get a page fault or a general protection fault. Each entry in a page table will also say whether that page is present and what kinds of permissions it has. The bit flags for the permissions are (in order from least to most significant bit):

- Bit 0: present.
- Bit 1: read/write.
- Bit 2: user (otherwise only the kernel can access it).
- Bit 3: write-through.
- Bit 4: cache disabled.
- Bit 5: accessed (for the TLB).
- Bit 6: page size (we'll talk about this later).
- Bit 7: (unused).

This way, since each process has its own page directory, page tables, and pages, and each level has specific permissions set, they should never be able to interfere with each other.

Again, most of the time, the kernel will just happily ignore all this and use the mapping in the higher half of virtual memory for simplicity. Each user process's page directory will have the same mapping in the higher half so that the kernel can keep doing what it's doing no matter which user process is currently running.

Anyway, back to the code! We left off after the boot loader had finished loading the kernel into memory; it ended by calling an `entry()` function in the kernel. We haven't set up paging yet, so that's next on our to-do list. But first, the kernel is compiled and linked using a *linker script*, so we'll have to look at that to understand how that sets up memory the way we want it.

kernel.ld

The gory details of linker scripts as a whole are outside the scope of these posts, so I'm gonna gloss over a lot of the parts of this file and focus on the important pieces.

It's important to understand what a linker does in a rough sense, so I'll just generalize and wave my hands around and say that a compiler takes code in a high-level language and converts it to assembly, an assembler takes that assembly code and turns it into machine code, and a linker takes a whole bunch of machine code files (including any code for library functions) and links them all together into a single executable file.

Linking involves three steps that are important for us here: first, the linker has to assign each piece of code a location in memory, so that different variables, functions, etc. don't end up colliding; then it replaces references to that object with its address. Second, it has to resolve any outstanding symbols (variables, functions, etc.) in each file by looking them up in all the other files and replacing them with those addresses; the linker can define its own symbols too. Third, it has to create an output file in a format that the OS can use, like ELF.

xv6 has decided that command-line flags are too basic for it, so instead it'll use a linker script kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld) for the GNU linker.

We start off by specifying the output format (32-bit ELF), the architecture (x86, also known as i386), and the entry point to start executing code. The convention is to call the entry point `_start`; the ELF header will include its address, which is how we were able to call it from the boot loader.

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
```

Next up come the sections. Remember the ELF sections `text`, `rodata`, `data`, `bss`, and `stab`? Well we've gotta tell the linker where to set them up in memory, using commands like `. = address`. These are virtual addresses, so since we want to set up our kernel in the higher half of virtual memory, we'll tell it to link the code start at 0x8010_0000. Again, we use that address instead of 0x8000_0000 (which maps to physical address 0) because we have to avoid the address spaces of the boot loader and the memory-mapped I/O devices.

We can also tell the linker where in physical memory the code should be placed (in linker script lingo, its "load address") using the `AT(address)` command. We'll use the physical address 0x0010_0000, since that maps to virtual address 0x8010_0000.

```
SECTIONS {
    . = 0x80100000;

    .text : AT(0x100000) {
        /* this part tells the linker which files to include in this section */
    }

    /* more sections here... */
}
```

There's one other detail we should check out: the linker can create its own symbols using the `PROVIDE(symbol = .)` command. If the code happens to declare its own variable `symbol`, then the linker will just throw away its own version of it, but if the code uses `symbol` without defining it, then the linker will replace those references with the contents of that memory location.

```
SECTIONS {
    /* virtual address and text sections are defined as above */

    PROVIDE(etext = .);     /* etext will be at the address right after the end
                            of the text section */

    /* rodata, stab, and stabstr sections defined here */

    PROVIDE(data = .);      /* data will be at the address at the very beginning
                            of the data section */

    /* data section defined here */

    PROVIDE(edata = .);     /* edata will be at the address right after the end
                            of the data section */

    /* bss section defined here */

    PROVIDE(end = .);       /* end will be at the very last address at the end
                            of the entire kernel code */
}
```

Those variables will be used later in the kernel code; not so much for their contents but for their addresses, as pointers to the virtual addresses of specific parts of the kernel's code in memory. On to the kernel!

entry.S

I have bad news. That `entry()` function that the boot loader called? It's in assembly again. :(

Multiboot Header

Okay, so first off, we've got some more hideous specs to deal with for a bit in the form of a multiboot header. Multiboot is a specification that lets boot loaders load up kernel code in a standardized way; the GNU boot loader GRUB uses it. So this part is mostly here in case you want to run xv6 on real hardware using GRUB; feel free to skip to `entry()` below.

The original Multiboot specification has since been replaced with Multiboot 2, but again, it's 1995, so we don't know about that yet.

Multiboot helps compliant kernels and boot loaders identify each other using a special header. The header must be completely contained in the first 8192 bytes of the kernel's image, and it must be 32-bit aligned. The header contains three things: (1) a magic number used for mutual identification and recognition (0x1BADB002 for kernels, 0x2BADB002 for boot loaders), (2) some flags for the kernel to inform the boot loader what the kernel requires in order to run successfully, and (3) a 32-bit unsigned checksum which when added to the other two fields must have a 32-bit unsigned sum of zero. Depending on the flags that are set, there may be other components to the Multiboot header.

So we'll start by creating a `multiboot_header` label at the beginning of the file (and thus, the beginning of the kernel image) and making sure it's aligned to 32 bits.

```
.p2align 2      # Force 4-byte alignment
.text
.globl multiboot_header
multiboot_header:
    # ...
```

Now we'll just add the magic number, set the flags to 0 to indicate no special requirements, and add the checksum.

```
    #define magic 0x1badboo2
    #define flags 0
    .long magic
    .long flags
    .long (-magic-flags)
```

And that's it!

entry

Back in kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld), we said that the linker would set up the kernel's ELF header to specify the kernel's entry point using `_start`, but `_start` itself wasn't actually defined there, so we have to do that first. We don't know where this code will end up in memory, so we'll define an `entry` label and set `_start` to the address of `entry`. Note that the linker script used virtual addresses in the higher half, but we haven't set up paging yet, so we'll have to convert it to a physical address using one of the macros we mentioned earlier.

```
.globl _start
_start = V2P_WO(entry)
.globl entry
```

Next up we want to finish setting up virtual memory by enabling paging, but that's all kinds of complicated, so we're gonna start off with a super simple version of paging. Part of that difficulty is that there's a bootstrap problem: we need to allocate pages to hold the page tables themselves, but we can't use pages without page tables... uhh...

We'll solve that by starting off with a basic, super-simple page directory where only two entries are mapped: the first entry maps virtual addresses 0 to 4 MB to physical addresses 0 to 4 MB, and the second entry maps virtual addresses `KERNBASE` to `KERNBASE` + 4MB to physical addresses 0 to 4 MB. One consequence is that the entire kernel code and data has to fit in 4 MB.

Why the two entries pointing to the same place? It's to solve another bootstrap problem. The kernel is currently running in physical addresses close to 0. Once we enable paging and start using virtual addresses in the higher half, the stack pointer `%esp`, instruction pointer `%eip`, even the pointer in `%cr3` to the page directory itself will all still point to low addresses until we update them. But updating them requires executing instructions, which would require accessing low addresses a few more times. If we left out the low addresses, we'd get a page fault, and since we don't have exception handlers set up yet, that would cause a double fault, which would turn into the dreaded **TRIPLE FAULT**, in which the processor enters an infinite reboot loop. So yeah, point is, we need both the low and high mappings for now; we'll get rid of the low mappings once we're done setting up.

But wait! Aren't page directory entries supposed to point to page tables? How can they point directly to pages here? It turns out that x86 can skip that second layer altogether if we use so-called "huge" pages of 4 MB in size instead of the usual 4 KB. In the long run, this could lead to internal fragmentation, but it does cut down on the overhead and allows a faster set-up. Plus we're only gonna use them for a minute while we get ready for the full paging ordeal.

To use 4 MB pages, we have to enable x86's Page Size Extension (PSE) by setting the fourth bit in the `%cr4` register. `CR4_PSE` is defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) as 0x10, or 00010000 in binary.

```
entry:
    movl    %cr4, %eax
    orl     $(CR4_PSE), %eax
    movl    %eax, %cr4
```

We need a page directory before we can set up paging; again, basic version now, full glorious page directory later. We're gonna do the same thing we did in the boot loader where we tell the processor to load the page directory now but then procrastinate actually writing it; this time, we'll write it in C and call it `entrypgdir`. Then we'll load its physical address into register `%cr3`.

```
    movl    $(V2P_WO(entrypgdir)), %eax
    movl    %eax, %cr3
```

Now we can enable (a basic version of) paging! We tell the CPU to start using the page directory in `%cr3` by setting bit 31 (paging) of register `%cr0`; we can also set bit 16 (write protect) of the same register to prevent writing to any pages that the page directory and page tables have marked as read-only. `CR0_PG` and `CR0_WP` are defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) to set these bits.

```
    movl    %cr0, %eax
    orl     $(CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0
```

Now remember how the processor is still running at low addresses? Yeah, let's fix that. First we'll make a new kernel stack in the higher half that will still be valid even after we get rid of the lower address mappings. We'll have the linker save some space for us under the symbol `stack` and set it up there; `KSTACKSIZE` is defined in param.h (https://github.com/mit-pdos/xv6-public/blob/master/param.h) as 4096 bytes. So we just set the stack pointer register `%esp` to the top of that section in order to let the stack grow down toward the address of `stack`. Again, we'll procrastinate actually defining `stack`.

```
    movl    $(stack + KSTACKSIZE), %esp
```

Now we want to call into the `main()` function, but we don't just want to do that the usual assembly way of `call main`. That would generate a jump relative to the current value of `%eip`, which is still in low addresses. We'll use an indirect jump instead.

```
    mov     $main, %eax
    jmp     *%eax
```

Finally, we need to get around to reserving space for the stack. We can do that with the assembler instruction `.comm symbol, size`:

```
.comm stack, KSTACKSIZE
```

main.c

Awesome, back to C code now! Remember how we procrastinated actually defining `entrypgdir`? Let's do that now; it's at the bottom of main.c (https://github.com/mit-pdos/xv6-public/blob/master/main.c).

entrypgdir

What in the world is this?!

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Okay, bear with me; I promise it's not too bad.

First, the `__attribute__` tells the compiler and linker that the page directory should be placed in memory at an address that's a multiple of `PGSIZE` (4096 bytes); that's just a requirement of the paging hardware.

Next, we define `entrypgdir` as an array of `NPDENTRIES` (1024, according to mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h)), each of type `pde_t` (a type alias for `unsigned int`, according to types.h (https://github.com/mit-pdos/xv6-public/blob/master/types.h)).

Then we initialize the entries: in C, you're allowed to initialize an array by specifying the values of specific enties; all other enties become zero. You specify an entry by putting its index in square brackets before its value, so `[2] 5` will set the entry with index 2 to be 5. Here we initialize the entries with indices 0 and `KERNBASE >> PDXSHIFT`, which is the same thing as `PDX(KERNBASE)`, AKA the page directory index corresponding to the virtual address `KERNBASE`, AKA 0x8000_0000. So basically, we've initialized the page directory entries corresponding to the low virtual address 0 and the high virtual address `KERNBASE`.

We set their value to 0, because we want them to map to physical addresses from 0 up to 4 MB. Oh, and remember how page directories and page tables can also hold permission flags? We want to set flags to say that these pages are present (so that accessing them doesn't cause a page fault), writeable, and 4 MB in size; those are defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) as `PTE_P`, `PTE_W`, and `PTE_PS`. We can combine them all together by bitwise-ORing them.

And we're done!

main

The code in entry.S (https://github.com/mit-pdos/xv6-public/blob/master/entry.S) finished up by calling into the C function `main()`, which is where the core set-up happens before we can start running processes. It calls into basically every single part of the xv6 kernel, so we can't go through all the functions line-by-line yet; instead I'll just give you an overview of what they do.

- `kinit1()` solves another bootstrap problem around paging: we need to allocate pages in order to use the rest of memory, but we can't allocate those pages without first freeing the rest of memory, which requires allocating them... You see what I mean. This function will free the rest of memory between the `end` of the kernel code (defined in kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld), remember?) and 4 MB.
- `kvmalloc()` allocates a page of memory to hold the fancy full-fledged page directory, sets it up with mappings for the kernel's instructions and data, all of physical memory, and I/O space, then switches to that page directory (leaving poor old `entrypgdir` in the trash).
- `mpinit()` detects hardware components like additional CPUs, buses, interrupt controllers, etc. Then it determines whether this machine supports this crazy new idea where you can have multiple CPU cores. Wow, 1995 is crazy.
- `lapicinit()` programs this CPU's local interrupt controller so that it'll deliver timer interrupts, exceptions, etc. when we're ready for them later.
- `seginit()` sets up this CPU's kernel segment descriptors in its GDT; we still won't really use segmentation, but we'll at least use the permission bits.
- `picinit()` disables the *ancient* PIC interrupt controller that literally no one has ever used since the APIC was introduced in 1989. I don't even know what to say. I guess I was mistaken when I assumed it was 1995; I don't know.
- `ioapicinit()` programs the I/O interrupt controller to forward interrupts from the disk, keyboard, serial port, etc., when we're ready for them later. Each device will have to be set up to send its interrupts to the I/O APIC.
- `consoleinit()` initializes the console (display screen) by adding it to a table that maps device numbers to device functions, with entries for reading and writing to the console. It also sets up the keyboard to send interrupts to the I/O APIC.
- `uartinit()` initializes the serial port to send an interrupt if we ever receive any data over it. xv6 uses the serial port to communicate with emulators like QEMU and Bochs.
- `pinit()` initializes an empty process table so that we can start allocating slots in it to processes as we spin them up.
- `tvinit()` sets up and interrupt descriptor table (IDT) so that the CPU can find interrupt handler functions to deal with exceptions and interrupts when they come.
- `binit()` initializes the buffer cache, a linked list of buffers holding cached copies of disk data for more efficient reading and writing.
- `fileinit()` sets up the file table, a global array of all the open files in the system. There are other parts of the file system that need to be initialized like the logging layer and inode layer, but those might require sleeping, which we can only do from user mode, so we'll do that in the first user process we set up.
- `ideinit()` initializes the disk controller, checks whether the file system disk is present (because both the kernel and boot loader are on the boot disk, which is separate from the disk with user programs), and sets up disk interrupts.
- `startothers()` loads the entry code for all other CPUs (in entryothers.S (https://github.com/mit-pdos/xv6-public/blob/master/entryothers.S)) into memory, then runs the whole setup process again for each new CPU.
- `kinit2()` finishes initializing the page allocator by freeing memory between 4 MB and `PHYSTOP`.
- `userinit()` creates the first user process, which will run the initialization steps that have to be done in user space before spinning up a shell.
- `mpmain()` loads the interrupt descriptor table into the CPU so that we're finally completely ready to receive interrupts, then calls the `scheduler()` function in proc.c (https://github.com/mit-pdos/xv6-

public/blob/master/proc.c), which enables interrupts on this CPU and starts scheduling processes to run. `scheduler()` never returns, so at that point we're completely done with setup and we're running the OS proper.

Summary

The entry code in the xv6 kernel had one job: to set up paging. It kind of failed at that job, but not for lack of trying! There are just all kinds of Catch-22s when it comes to paging, so at least it got us partway there by making a temporary page directory to tide us over until we can throw it away and never look back.

We also took a sneak peek at all the setup code in `main()`; we're gonna end up going through it all, but at least now you should have enough of an idea of what's going on that you can more or less skip around and look at what you need.

Page Allocation

When we left off before the lock detour, the boot loader had set up a GDT to ignore segmentation, and the entry code set up some barebones paging with an `entrypgdir`. But that initial page directory is too limiting to keep for long; it only mapped the first 4 MB of physical memory. So we want a new one, but we have to set it up and allocate pages in it before we can actually use it. And until we switch to it, everything has to happen in those first 4 MB.

kalloc.c

We start off in this file by declaring the function `freerange()`, which will be defined below. We have to do this in C in order to call a function in the code before the compiler has actually seen the function's definition, which comes below, or maybe in another file. A *declaration* tells the C compiler "I know I haven't shown you this symbol before, but don't worry; it's just a function that takes this number of arguments with these types and has a return value of this type." That lets the compiler keep calm and carry on with its usual type-checks (weak as they may be in C). A *definition* tells the compiler that this is the function (or variable) we were talking about, so it'll reserve some space in memory for it; it also tells the compiler how to evaluate that function whenever it's called (for variables, an *initialization* will have to tell the compiler what the value the variable should hold). The linker will take care of matching function calls (and variable uses) to their definitions, possibly across files.

Usually you'd stick declarations in a C header file and tell the preprocessor to copy-paste the header into your code with an `#include` directive; then other files could `#include` that header too. So header files should really be more of an API kind of thing, for functions that you want other code to be able to call. This one is just a local helper function, so we'll declare it here instead of in a header so other code can't use it.

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

Okay okay, I know function declarations are like 101-level C, but I wanted to mention them because we're about to see something similar but a little off next when we declare `end` as a global array of characters.

```
extern char end[];
```

The C keyword `extern` lets you define a global variable or function in one file and use it in another, so in that sense it's similar to the function declaration above. In fact, the compiler implicitly assumes there's an `extern` before each function declaration. The difference is that an explicit `extern` lets us do the same thing for global variables: we tell the compiler and linker "hey, I'm gonna use a variable of this type with symbol `end`, but don't worry about reserving a spot in memory for it; that already happened elsewhere."

The really cool thing about `extern` is that the function or variable might not even be defined in C -- it could come from any other language! We just pass the compiled object files from the other language together with the C object files to the linker and it'll match up the definitions and calls.

In this case if you try looking for the place where `end` is defined in the C or assembly code, you're gonna be disappointed. Turns out it's actually defined in kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld), remember? Back then, we said it was gonna be located at the very first memory address right after the end of the kernel code and data in memory. We're about to see why it's needed.

Next up, we define a new `struct` type:

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

Hmm, the only member of this `struct run` is a pointer to another `struct run`. Hopefully, you've seen some singly-linked lists before so you can recognize it as one of those. Usually it would have another member to hold the data in the list, but we won't need any extra data here; we'll find out why soon enough.

Last thing before we get to the functions: we define another `struct` type and declare the global variable `kmem` to be of that type.

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

The syntax here is the usual C thing where we say the type of a variable, then an identifier, like `int i`; it just looks more confusing because we're also defining the type at the same time. This `struct` type doesn't get a name like `struct run` did because we're only gonna need it this one time. The fields are a spin-lock (hence the detour before coming here), a `use_lock` variable that we'll treat as a boolean, and a pointer to a `struct run` called `freelist`.

I'm just gonna go ahead and spoil the next two functions for you: we want to use a better page directory than `entrypgdir`, right? Well then we need to assign a page of memory for it, plus a page for each of its page tables, plus a page for each entry in those page tables that's mapped. That means we'll need some bookkeeping to track which pages have already been assigned. We're gonna use a linked list of free pages (that's what `struct run` is for); we'll allocate a page by popping one off the free list, and we'll free a page by pushing it onto the top of the list.

Note that `kfree()` here is *not* supposed to be a kernel version of the usual C standard library function `free()`, nor is `kalloc()` supposed to be a kernel version of `malloc()`. We have no concept of a heap yet, so heap allocation wouldn't make sense. These functions allocate and free *whole physical pages* to be added to the current page directory and its page tables.

kfree

This function will free a single page (4096 bytes, or `PGSIZE`) of memory by adding it to the front of the free list. It takes an argument `char *v` which is a virtual address; we're using `char *` here instead of `uint *` or `void *` or whatever so that the pointer arithmetic increments by a single byte instead of 4 bytes for `uint` or whatever.

First, some sanity checks: `v` should be page-aligned (because we're freeing a whole page), it should be above `end` (because we don't want to accidentally overwrite the kernel code), and its corresponding physical address should be below `PHYSTOP` (because the only addresses we'll use above the top of physical memory are for memory-mapped I/O devices and we shouldn't be freeing those pages anyway).

```
void kfree(char *v)
{
    if ((uint) v % PGSIZE || v < end || V2P(v) >= PHYSTOP) {
        panic("kfree");
    }
    // ...
}
```

Now, if you've programmed in C, you might have come across the dreaded (but oh- so-common) bug known as a *use-after-free*. This means you called `free()` on some variable (hopefully one you had `malloc()`-ed before), and then used it again. Hmm, very naughty! The problem is that that memory might have been re- allocated to some other variable or even another process, so you might read the wrong values or overwrite something important. This is a *very* common cause of security vulnerabilities in C and C++ to this day; it's also not always easy to spot because huge projects might have you call `malloc()` in one file, then use the variable somewhere else thousands of lines of code later in some other file, then call `free()` in yet another file -- plus it's unlikely that all of these pieces were written by the same person. So let's make this a little easier on ourselves by filling the freed page with junk (a bunch of 1s everywhere) in the hope that a use-after-free leads to a crash (and thus debugging and detection) sooner than it would otherwise.

```
void kfree(char *v)
{
    // ...
    memset(v, 1, PGSIZE);
    // ...
}
```

You might be familiar with `memset()` from the C standard library in *string.h*, but we can't risk using standard library functions here because they assume the code will be provided by the OS, and the implementation might require any of a million features we haven't implemented yet. So we have to make our own version for the kernel in string.c (https://github.com/mit-pdos/xv6-public/blob/master/string.c). We'll get around to looking at that code later on in an optional detour, but for now just know that it sets the memory starting at `v` and continuing for `PGSIZE` bytes to hold a bunch of repeated 1s.

Now let's talk concurrency. At any time, multiple threads might want to allocate or free pages simultaneously; if we're not careful we might accidentally use the same page twice, which would cause bugs in addition to security vulnerabilities, because all the per-process isolation that paging gets us would be lost. So much work down the drain! This is why `kmem` has a lock, which we should use any time we push to or pop from the free list.

But in the early stages of the kernel we only use a single CPU and interrupts are disabled, so there's nothing to fear. Plus, locks add overhead, and the `acquire()` function needs to call `mycpu()`, which we haven't even defined yet, so let's just go ahead and skip them in the beginning. So `kmem.use_lock` is a boolean that will tell us whether we need a lock right now or not.

```
void kfree(char *v)
{
    // ...
    if (kmem.use_lock) {
        acquire(&kmem.lock);
    }
    // ...
}
```

Okay, we're finally at the point where we can free the page. We'll make a `struct run *r` that points to virtual address `v`, then make its `next` point to the first entry of the free list. Then we'll update the head of the list to point at the newly-freed page. This is the standard C idiom to add to the front of a singly-linked list.

```
void kfree(char *v)
{
    // ...
    struct run *r = (struct run *) v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    // ...
}
```

There's something interesting here: where are we storing this entry for the free list? Why, in the free page itself! So each unused page will hold the address of the next one in its first few bytes.

Finally, we're out of the critical section where we updated the free list, so we can release the lock.

```
void kfree(char *v)
{
    // ...
    if (kmem.use_lock) {
        release(&kmem.lock);
    }
}
```

kalloc

Allocating a page means popping off the head of the free list. We acquire the lock first, if we need one.

```
char *kalloc(void)
{
    if (kmem.use_lock) {
        acquire(&kmem.lock);
    }
    // ...
}
```

Next, we get a pointer to the first free page in the list and update the head to point to the next one in the list. But what if the list is empty? In that case, the head would be a null pointer, and dereferencing a null pointer (like we do here in `r->next`) is undefined behavior in C, which means BAD THINGS HAPPEN. I'm serious -- there are absolutely no restrictions on what might happen, so the compiler could literally set your computer on fire if it wanted to. In the real world, that usually means either a segmentation fault or security vulnerability, or both if you're unlucky. So we should check whether `r` is null (i.e. zero). if it's nonzero then we can update `r->next`; otherwise we should just return `r` and hope whoever called us checks whether it's null. Moral of the story: any call to `kalloc()`, just like any call to `malloc()` in regular C code, should always be followed by checking whether the returned pointer is null.

```
char *kalloc(void)
{
    // ...
    struct run *r = kmem.freelist;
    if (r) {
        kmem.freelist = r->next;
    }
    // ...
}
```

Okay, so now we just release the lock, and we're done!

```
char *kalloc(void)
{
    // ...
    if (kmem.use_lock) {
        release(&kmem.lock);
    }
}
```

freerange

`kalloc()` and `kfree()` both handle only one page at a time, which can get annoying if we're trying to free tons of pages at once; also, they can only use page-aligned virtual addresses, which have to be typecast to `char *`. Let's simplify our lives with a simple wrapper function to free multiple pages between two virtual memory addresses `vstart` and `vend` that may not be page-aligned.

Let's assume that `vstart` is the first address after some other data in an already-allocated page; we don't want to free that page, but the next one, so we align it to a page boundary by rounding up, then cast that to a `char *`.

```
void freerange(void *vstart, void *vend)
{
    char *p = (char *) PGROUNDUP((uint) vstart);
    // ...
}
```

Now we can iterate over the pages, starting at `p` and incrementing by `PGSIZE` until we reach or pass `vend`, freeing pages as we go.

```
void freerange(void *vstart, void *vend)
{
    // ...
    for (; p + PGSIZE <= (char *) vend; p += PGSIZE) {
        kfree(p);
    }
}
```

Done, next.

kinit1 and kinit2

Both of these functions get called by the kernel's `main()`. Quick reminder: we've got an `entrypgdir` that maps two virtual address ranges (0 to 4 MB and `KERNBASE` to `KERNBASE` + 4 MB) to the physical addresses range from 0 to 4 MB. We want to leave this baby page directory behind for a grown-up page directory that maps all of physical memory, but first we needed to figure out how to allocate pages.

Okay cool, we already did that. But allocation needs a free list, which for now is just sitting around chilling as an empty list. But we can't free pages if they're not already allocated, right? Ahh, bootstrap problems! This one's not an issue; we'll just cheat this one time and free all the memory between `end` (the end of the kernel code and data in memory) and `PHYSTOP`, even though we didn't get it from a call to `kalloc()`. Sounds good, right?

I hate to burst your bubble, but kernel development *loves* bursting bubbles. Turns out there's yet another bootstrap problem: each page has to store the pointer to the next free page, which means we have to write to that page, which means that page must already be mapped... but we can't map all of memory until we initialize the free list by freeing all of memory...

HEAD. DESK. We're screwed.

Okay, obviously the xv6 authors figured this out already. The trick is that we do have *some* physical memory we can write to: everything between `end` and 4 MB. So we can free that part for now, allocate some of those pages for a fresh page directory and some pages, then use those pages to map the rest of physical memory, then come back later and free those pages.

So we'll have to split up the work of setting up the new page directory into two very similar functions, `kinit1()` and `kinit2()`. The first one will initialize the lock for the free list but make `kmem.use_lock` false so we don't use a lock in the early stages of kernel setup. The second one will set it to true so we start using a lock to allocate and free pages once we have multiple CPUs, a scheduler, interrupts, etc.

Both of them will use `freerange()` to free the pages in a section of physical memory. `main()` calls `kinit1()` with arguments to free the range from `end` to 4 MB, and calls `kinit2()` with arguments for the range from 4 MB to `PHYSTOP`.

```
void kinit1(void *vstart, void *vend)
{
    initlock(&kmem.lock, "kmem");
    kmem.use_lock = 0;
    freerange(vstart, vend);
}

void kinit2(void *vstart, void *vend)
{
    freerange(vstart, vend);
    kmem.use_lock = 1;
}
```

Summary

This whole file was just to set up page allocation for the new page directory we're gonna replace `entrypgdir` with. It uses a free list in `kmem`; freeing a page adds it to the front of the list and allocation pops a page off the front. We have to populate the free list will pages for all of physical memory, but we do that in two steps to avoid some bootstrap issues.

Again, this is a *page* allocator, not a *heap* allocator like `malloc()`, but many heap allocator implementations use linked lists of free heap regions in the same way. We talked about use-after-free bugs above, but now we can also see why *double-frees* (in which you free the same memory region more than once) can cause bugs and security vulnerabilities: they add the same region to it twice, which then might get allocated to two different variables or processes, which might ruin the per-process isolation that virtualization is supposed to provide. In addition, our page allocator handles fixed-size regions, but a heap allocator needs to use variable regions, so when a memory region gets allocated twice after a double-free, it might get split up into differently-sized pieces, of which some parts get allocated to other processes, etc... It's just a nightmare.

Next up, we'll see the full story of virtual memory.

More Paging: The Kernel Side

We've already talked *plenty* about virtual memory, and I bet you're probably so over `entrypgdir` by now; let's wrap up its story and get rid of it!

The vm.c (https://github.com/mit-pdos/xv6-public/blob/master/vm.c) file is HUGE; only proc.c (https://github.com/mit-pdos/xv6-public/blob/master/proc.c) and sysfile.c (https://github.com/mit-pdos/xv6-public/blob/master/sysfile.c) match its length. Some parts deal with the general paging implementation; we'll look at those here. The rest handles the details of paging for processes and user code, we'll need to know a bit more about processes in xv6 for that.

vm.c

After the include directives for the preprocessor, we have a declaraction for an external symbol defined in kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld). This one is the beginning of the data section for the kernel.

```
extern char data[];
```

Next we have a definition for a pointer to a global page directory: this is the fancy new one that's gonna replace `entrypgdir`. Note that `pde_t` is a type for page directory entries defined in types.h (https://github.com/mit-pdos/xv6-public/blob/master/types.h); it's just a type alias for `int`.

```
pde_t *kpgdir;
```

seginit

This first function gets called directly by the kernel's `main()`; it sets up the segment descriptors in the GDT as identity maps to all of memory so that we can ignore them from now on. Wait, didn't we already do that in the boot loader?

Yes, kind of, but that was before the kernel took over, so back then we had no notion of kernel space versus user space. Now that we do, we want to set the permission flags for each segment so that we can use the privilege ring levels, with the kernel in ring 0 and user code in ring 3. That way any misbehaving user code will get slapped with a segmentation fault the way we've all come to know and love in C.

We also have some permission flags for protection in the page directory and page table entries, so maybe we could get away without it? I mean, both kernel code and user code are read-only anyway, so maybe they could both have a Descriptor Privilege Level of 3. But no, x86 is gonna shut that right down by forbidding interrupts that take you from ring level 0 to ring level 3, so all the interrupt handler functions have to be in kernel space with a kernel code segment selector at ring level 0.

So we're just gonna have to do it all over again. Great. Well, maybe it's not too bad, let's take a look... oh god, it's awful. Okay, deep breath.

Each processor has its own GDT, so we're gonna need to call this function once per CPU. First we figure out which CPU we're on with with the `cpuid()` function that we'll see later on; for now it... (drumroll)... gets the CPU's ID. Then we look that up in a global table of CPUs (there's an `extern` declaration for this in the included proc.h (https://github.com/mit-pdos/xv6-public/blob/master/proc.h)) and store it in a `struct cpu`; we saw that before in the spin-lock code, but we'll get around to talking about it more later.

```
void seginit(void)
{
    struct cpu *c = &cpus[cpuid()];
    // ...
}
```

That `struct cpu` has a field to hold the GDT, so we're gonna add entries for the kernel code, kernel data, user code, and user data segment descriptors; those entries are `SEG_KCODE`, `SEG_KDATA`, `SEG_UCODE`, and `SEG_UDATA`, respectively. Recall that the permission bits are `STA_X` (executable), `STA_R` (readable), and `STA_W` (writeable); now we're gonna pile on the descriptor privilege levels for the kernel (0) and user (3, or `DPL_USER`) on top. Besides those ring levels, we want to ignore segmentation, so each segment should be an identity map for all virtual memory from 0 to 4 GB (0xffff_ffff).

```
void seginit(void)
{
    // ...
    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);
    // ...
}
```

The only difference between the `SEG` macro used here and the `SEG_ASM` one from the boot loader is that this one is for C code and the other is for assembly.

Finally, we load up the new GDT into the processor with a C wrapper for the x86 instruction `lgdt`.

```
void seginit(void)
{
    // ...
    lgdt(c->gdt, sizeof(c->gdt));
}
```

Done with segmentation, now on to more paging.

walkpgdir

A page directory lets the paging hardware convert virtual addresses to physical ones, but we're gonna need those mappings in the kernel too while we set up the page directory, so this function does the conversion manually. Wait, but aren't we setting up paging so that all of physical memory is mapped in the higher half of the virtual address space? Can't

we just add or subtract `KERNBASE` to do the conversion? Well, that would work for kernel virtual addresses, but user virtual addresses actually will use page directories and page tables in a non-obvious way, so if we want to figure out where those go, we'll need a function for it.

In C, using the `static` keyword before a function limits its scope and makes it visible only within its own file. The function returns a `pte_t *`, a pointer to a page table entry (the type is defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) as a type alias for `uint`).

Its arguments are a pointer to a page directory, a virtual address, and `alloc` (a boolean variable, but as an `int` instead of `bool`). This `alloc` lets the function play a dual role: if it's set, the function will allocate a page table if needed; otherwise it reports failure if the page table doesn't exist. The `const` keyword lets the compiler know a variable shouldn't be mutated so it'll throw an error if we do. Here, `const void *va` is a pointer to a constant value of any type; the address the pointer holds might change, but we can never write to that address. The opposite is a `void *const va`: the address being pointed to will never change, but we can overwrite the contents of that address all we want. You can combine the two with `const void *const va`. What's that I hear? C syntax is the worst? No, never...

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
}
```

Remember way back when, when we talked about how "linear" addresses are set up and converted to physical ones? The first 10 bits are an index for the page directory to pick a page directory entry, which points to a page table; the next 10 bits pick a page table entry that points to a page, and the last 12 bits are an offset within that page; the `PDX()` and `PTX()` macros get first 10 bits and the next 10 bits from a linear address, respectively. So we start by getting the page directory index and using that to get the page directory entry.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde = &pgdir[PDX(va)];
    // ...
}
```

Okay, so now `pde` points to a page directory entry which has two parts: a pointer to the physical address of a page table, and some flags. But who knows if this page table even exists; most page directory (and page table) entries aren't mapped in order to save space. So we have to check whether `*pde` has the `PTE_P` (present) flag set.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    pte_t *pgtab;
    if (*pde & PTE_P) {
        // ...
    } else {
        // ...
    }
    // ...
}
```

If the page table exists, we should get rid of the flags and recover the pointer to the page table using the `PTE_ADDR()` macro. But the hardware uses physical addresses for these pointers, so we need to convert it to a virtual address first, which is what this function does... recursion? Bootstrap problem? No, it's actually easy because we can access the page table from within the kernel's virtual address space in the higher half by adding `KERNBASE` to the physical address with the `P2V()` macro.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    pte_t *pgtab;
    if (*pde & PTE_P) {
        pgtab = (pte_t *) P2V(PTE_ADDR(*pde));
    } else {
        // ...
    }
    // ...
}
```

Now for the else clause, which happens if the page directory entry doesn't have the `PTE_P` bit set. Well, if the boolean `alloc` is false (zero), then we're done and we should just report failure by returning a null pointer. On the other hand, if it's true, we just allocate a page for the page table. But wait, remember how page allocation might fail and return a null pointer if we're out of free pages in the free list? And remember how I said we should always check for that? Okay well let's check for that; if allocation fails, we also return a null pointer. Oh, and because this is C, we're gonna do a jillion things at once in a single line: check if `alloc` is false, try to allocate a page table, and check if that allocation failed. C lets us assign to a variable and then test that variable's value in a single statement.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    pte_t *pgtab;
    if (*pde & PTE_P) {
        // ...
    } else {
        if (!alloc || (pgtab = (pte_t *) kalloc()) == 0) {
            return 0;
        }
        // ...
    }
    // ...
}
```

Okay, so now suppose: (1) the page table wasn't present, (2) alloc was set, and (3) we successfully allocated a page. Now what? Remember how we filled all free pages with garbage in `kfree()` using `memset()`? Let's undo that now by zeroing it.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    pte_t *pgtab;
    if (*pde & PTE_P) {
        // ...
    } else {
        // ...
        memset(pgtab, 0, PGSIZE);
        // ...
    }
    // ...
}
```

Now we'll update the page directory entry to point to this new page table and add the `PTE_P` flag so it knows it's present. Wait, while we're at it, what other permissions will it need? Is it writeable? Can users access it? Hmm, we'd have to know whether we're looking up a user virtual address or a kernel one, and whether it's gonna be used for code or data. Ah, screw it, we'll just throw all the flags on there at once. Either way, the page table entries will have their own flags too, so we can restrict the page's permissions there instead of here at the page directory entry.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    pte_t *pgtab;
    if (*pde & PTE_P) {
        // ...
    } else {
        // ...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    // ...
}
```

This probably isn't the safest thing ever, because we're saying that only the page table will restrict permissions, so we're throwing all that responsibility over there, but hey, xv6 is supposed to be simple, not ultra-secure. Just don't do this at home, kids.

Finally, we return the address of the corresponding page table entry using the index from the middle bits of `va`:

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    // ...
    return &pgtab[PTX(va)];
}
```

mappages

Okay, so `walkpgdir()` returns a pointer to a page table entry and can even crate a page table if if it doesn't exist. That's not quite enough to add new mappings for pages though; the page itself might not be mapped, and if we just created a new page table, then certainly none of the pages are mapped yet. `mappages()` will finish the job by installing mappings in page tables (possibly newly-allocated ones) for a range of virtual addresses.

The arguments are a page directory, a virtual address for the beginning of the range, the size of the range, a physical address to map it to, and the flags for permissions we want to set. We start off by rounding the virtual address down to the nearest page boundary and getting a pointer to the end of the range, also page-aligned.

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    char *a = (char *) PGROUNDDOWN((uint) va);
    char *last = (char *) PGROUNDDOWN(((uint) va) + size - 1);
    // ...
}
```

Now we're gonna iterate over the pages in that range; `for (;;)` is a common C idiom for an infinite loop. In this case, we need to increment `a` and `pa` by `PGSIZE` each time, and we'll break out of the loop when `a` reaches `last`. To be completely honest, I'm not really sure why the authors chose to write this as an infinite loop with the condition/break statement and update statements inside the loop rather than as a regular old for loop; I think the latter would be more clear, but oh well, I didn't write this.

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    // ...
    for (;;) {
        // ...
        if (a == last) {
            break;
        }
        a += PGSIZE;
        pa += PGSIZE;
    }
    // ...
}
```

Inside the for loop, we'll start each iteration by looking up the right page table entry with `walkpgdir()`, with `alloc` set to true. Remember how that function called `kalloc()`, which might fail, in which case it returns a null pointer? Well that means we've gotta check for a null pointer here too. This time however, we'll return -1 for failure and 0 for success, because why not?

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    // ...
    for (;;) {
        pte_t *pte;
        if ((pte = walkpgdir(pgdir, a, 1)) == 0) {
            return -1;
        }
        // ...
    }
    return 0;
}
```

We're supposed to be allocating brand-new pages for this range of addresses, so if a page has already been allocated, we'll just flip out in rage and panic.

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    // ...
    for (;;) {
        // ...
        if (*pte & PTE_P) {
            panic("remap");
        }
        // ...
    }
    // ...
}
```

The last thing before checking the loop condition and updating `a` and `pa` is to install the mapping to the right physical address with the right permissions in the page table. Then we're done!

```
static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    // ...
    for (;;) {
        // ...
        *pte = pa | perm | PTE_P;
        // ...
    }
    // ...
}
```

Cool, now we have a way to map new pages into a page directory. We're well on our way to leaving poor old `entrypgdir` behind for the shiny new `kpgdir`.

kmap

Each process is gonna have its own page directory, so its mappings in the lower half of the virtual address space might be totally different from those of another process. But the mappings in the higher half (where the kernel lives) will always be the same -- that way, the kernel can always use the existing page directory for whatever process it happens to be running. We'll only use `kpgdir` when the kernel isn't currently running a process, e.g. while it's running the scheduler.

So when we create a new process, we'll need to copy in all the mappings that the kernel expects to find into a fresh page directory for that process. Those are: memory-mapped I/O device space from physical address 0 to 0x10_0000 (the boot loader is also here, but we don't need it any more), kernel code and read-only data from 0x10_0000 to the physical address of `data` (one of the symbols defined in kernel.ld (https://github.com/mit-pdos/xv6-public/blob/master/kernel.ld)), kernel data the rest of physical memory from there to `PHYSTOP`, and more I/O devices from 0xFE00_0000 and up. Each of these ranges needs its own permissions too.

We'll represent each of these mappings with a `struct kmap`, which has fields for the starting virtual address, the starting and ending physical addresses, and the permissions; then the mappings will get stored in a static global variable `kmap`... oh come on, what fresh hell is THIS?

```
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void *)KERNBASE, 0, EXTMEM, PTE_W },
    { (void *)KERNLINK, V2P(KERNLINK), V2P(data), 0 },
    { (void *)data, V2P(data), PHYSTOP, PTE_W },
    { (void *)DEVSPACE, DEVSPACE, 0, PTE_W }.
};
```

Okay, there are a few things going on here. First, the `static` keyword for a variable means that variable has a single fixed location in memory that it's never gonna move out of.

Then it does that thing again where we simultaneously define a `struct` type and define a variable of that type. So the type is

```
struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
};
```

So then the static global variable `kmap` is an array of `struct kmap`s. I guess we ran out of names or something. The array has four entries, and since each one is a `struct`, it needs curly braces around it.

The first entry (for the lower of the two memory-mapped I/O device regions) has a `virt` field of `KERNBASE`, a `phys_start` field of 0, a `phys_end` field of `EXTMEM` (defined as 0x10_0000), and permission flag `PTE_W`. So it maps a virtual address range starting at `KERNBASE` to the physical address range from 0x0 to 0x10_0000 and makes it writeable so we can communicate with the devices there. The next two entries are similar, except that the kernel code isn't writeable.

The last entry has `phys_start` of 0xFE00_0000 and a `phys_end` of 0. That's a little strange, but it's because we want to map all the way up to the end of the virtual address space at 0xFFFF_FFFF. The end should be one byte past that, but it's impossible to represent 0x1_0000_0000 with 32 bits. Setting the end to 0 makes the size calculation (`phys_end -`

`phys_start`) work out nicely: it'll just overflow to the right number. This is okay since we're using unsigned integers, but note that *signed* integer overflow is undefined behavior and thus VERY BAD and the cause of many security vulnerabilities.

Okay, back to getting rid of `entrypgdir`!

setupkvm

This function sets up a fresh new page directory with all the mappings in `kmap` in order to please the kernel when it encounters the page directory. So needy, right?

It takes no arguments and returns a pointer to the new page directory. First, let's allocate a page of memory to hold the new directory. We'll be good and remember to check for null (in which case we return null too) and clear the page of the garbage values we wrote when we freed it.

```
pde_t *setupkvm(void)
{
    pde_t *pgdir;
    if ((pgdir = (pde_t *) kalloc()) == 0) {
        return 0;
    }
    memset(pgdir, 0, PGSIZE);
    // ...
}
```

The upper end of virtual memory after `DEVSPACE` has I/O devices, so `PHYSTOP` should be below that; this is as good a place as any to make sure.

```
pde_t *setupkvm(void)
{
    // ...
    if (P2V(PHYSTOP) > (void *) DEVSPACE) {
        panic("PHYSTOP too high");
    }
    // ...
}
```

Finally, we'll add all the mappings in `kmap` above into this page directory so the kernel is happy. We'll use `mappages()`, which returns -1 if it fails, so we should check for that. The `freevm()` function is defined below, and we'll get to it soon, but for now just know that it gets rid of all the mappings we just made, in case any of them fails.

```
pde_t *setupkvm(void)
{
    // ...
    struct kmap *k;
    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;
}
```

Let's check out that for loop: `k` is a pointer to a `struct kmap`, and `kmap` is an array of `struct kmap`s; in C, arrays decay to pointers, so they have the same type. `k` starts off pointing to the first (zero) entry of `kmap`. Then incrementing it with `k++` shifts its value by the size of a `struct kmap`, so it'll point to the next entry. The loop stops when `k` points beyond the last entry of `kmap`, as determined by the `NELEM()` macro which counts the number of entries in an array. Note that array element-counting only works in C if the array is defined in the same function or as a global variable in the same file, which is why it's so easy to do an out-of-bounds read or write in C (yet another common security vulnerability).

Finally, if everything worked out okay, we return a pointer to the new page directory.

switchkvm

We said above that the kernel would usually just use the page directory of the currently-running process, but it'll use `kpgdir` when no process is running, i.e. during the kernel setup and while it's scheduling a new process. So we need a way to tell the paging hardware to load `kpgdir` into register `%cr3`, which holds a pointer to the page directory. That's this function.

It's a one-liner: get the physical address of `kpgdir` and stick it in `%cr3` with the assembly instruction `lcr3`.

```
void switchkvm(void)
{
    lcr3(V2P(kpgdir));
}
```

kvmalloc

FINALLY, we're here! We're gonna get rid of `entrypgdir`! The kernel's `main()` calls this function right after `kinit1()`.

We already did all the hard work, so this one's a breeze: we call `setupkvm()` to allocate a new page directory and fill it with the kernel's mappings, then call `switchkvm()` to load it into the paging hardware.

```
void kvmalloc(void)
{
    kpgdir = setupkvm();
    switchkvm();
}
```

And we're DONE! Take that, `entrypgdir`, we don't need you anymore. We're big kids now.

Summary

So far, it's been a serious odyssey just to move from no paging in the boot loader, to super basic paging with `entrypgdir` in entry.S (https://github.com/mit-pdos/xv6-public/blob/master/entry.S), to `kpgdir` now. Along the way, we've looked at code to allocate and free pages and install new mappings in page directories and page tables. That'll come in handy when we look at processes next; the virtual memory story still isn't over.

Also, note that `kpgdir` still isn't at the height of its powers: at the point when `main()` calls `kvmalloc()`, the free list only contains pages for physical memory between 0 and 4 MB. The rest will have to wait until `kinit2()` unleashes its full potential. (Maybe some self-actualization seminars would help...)

More Paging: The User Side

It's almost time to turn to interrupts and processes so we can figure out how to work that sweet multiprocessing magic, but unfortunately we have some last pieces of paging to wrap up before we can get there.

I know, we've been talking about virtual memory for what feels like a century now, but so far everything we've done has been on the kernel side, allocating pages and creating new page directories with the same kernel mapping. But what about the lower half of the virtual address space, where user processes live?

This post will go through the rest of vm.c (https://github.com/mit-pdos/xv6-public/blob/master/vm.c) and set up the paging-related machinery we'll need to run processes later on.

Detour: Starting a New Process

When xv6 runs a new process, it will create a brand new virtual memory space for it with a fresh page directory. We haven't talked about processes in xv6 yet, so you might wonder how a process gets started up in the first place.

Let's forget all about xv6 for a second and think about another Unix-like OS: Linux. How do we start a process there? Okay, we also have to forget about GUI applications there. Let's just say you want to run some C code (xv6 maybe?) that you've just compiled; what happens when you run it from the terminal?

Hopefully, you've done the OSTEP project called processes-shell (https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/processes-shell) by now, so you know the answer; if you haven't, I recommend doing that one right now before I give it away. (It's not strictly required, but are you really the kind of person who loves getting movies spoiled for them?)

Okay, are you done?

The answer: it's just an `exec()` system call! The shell finds the executable file in the file system, calls `fork()` to create a new child process, which then calls `exec()` to transform itself into the program you want to run.

We'll get to these system calls later, so for now let's just go over the broad strokes as they relate to virtual memory. `fork()` works by taking the parent process's virtual memory space and making a copy of it for the child process.

`exec()` allocates a new page directory, figures out how much memory the new program will need when it runs, then grows the virtual memory space allocated in that new page directory to the required size. Then it loads the program into memory in the new page directory.

Next, `exec()` skips a page, leaving it mapped but user-inaccessible; then the next page becomes the process's stack. Why that empty page? It's an important one for protection: that way, user programs that blow their stack will trigger a page fault or a general protection fault instead of possibly overwriting random code.

Then `exec()` copies some arguments into the stack before it switches to using the new page directory and gets rid of the old one it had before.

Whew, okay, that's a lot of code to go over later, and that's only the virtual memory part of the story. So let's just make it easier by doing all the work we can right now. According to the above, we have to understand how xv6 does all of the following:

- Makes a copy of a whole page directory,
- Creates a new page directory,
- Grows (or shrinks) the virtual memory space of a page directory,
- Loads program code into a page directory,
- Makes a page inaccessible to users,
- Copies stuff into a page in a page directory,
- Switches to a new process page directory, and
- Gets rid of an unused page directory.

Finally, there's one edge case to think about: running the very first process. We obviously need to start running a shell at some point, so we need a special way to get that started too, so it can in turn run other processes.

vm.c, Again

We're gonna need some new functions! Actually, we already finished one of the requirements -- `setupkvm()` can allocate a new page directory and set up the kernel portion too. `switchkvm()` lets us switch to using `kpgdir` as a page directory, but now we need to switch *away* from that to a page directory for a process, so that'll be `switchuvm()`.

`copyuvm()` creates a copy of an entire page directory for a child process. `allocuvm()` and `deallocuvm()` grow and shrink the virtual memory space that's allocated in a page directory, and `freevm()` clears a page directory we no longer need.

`loaduvm()` will load program code into a page directory; `clearpteu` makes a page inaccessible to users, and `copyout()` copies data into a page in a page directory. `inituvm()` handles the special case of setting up the page directory for the very first process that xv6 will run.

The rest of this post will go over those functions one by one so we can be done with virtual memory, but I know it's a little strange to go through a million helper functions when we haven't seen the code that's gonna use them yet, so if you'd prefer, you can come back to this after reading about processes and system calls.

deallocuvm

The arguments for this function are a page directory, the process's old size, and the new size we want to shrink it down to; it'll return the process's new size. By "shrinking" a virtual memory space, we really mean making sure that the page directory only allocates up to `newsz` worth of pages. So if we think of the sizes as virtual addresses, then the page directory currently maps the virtual space from 0 to `oldsz`, so we should free everything between `newsz` and `oldsz`, leaving behind the space from 0 to `newsz`.

First, we should make sure the new size is actually smaller than the old one; otherwise trying to "shrink" down to the new size might cause integer overflow. There; the sizes are both unsigned integers here, so at least it wouldn't be that scary boogeyman of undefined behavior, but it could still be bad: 0 would wrap around to 2^32 - 1, so "shrinking" to the new size would actually grow the process way beyond what physical memory could handle.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    if (newsz >= oldsz) {
        return oldsz;
    }
    // ...
}
```

We're gonna shrink the physical memory allocated to this page directory by freeing pages until we reach the new size. Let's start with the first page above `newsz`; we can get its virtual address by rounding up `newsz` to a page boundary.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    uint a = PGROUNDUP(newsz);
    // ...
}
```

Now we'll just iterate over the pages between `a` and `oldsz` one at a time and free them. This is a little tricky: `kfree()` takes a virtual address (cast to a `char *`), but it should be a *kernel* virtual address in the higher half, not a user virtual address. Luckily, we already have `walkpgdir()`, which can take an arbitrary virtual address and return its page table entry, so that's a good start.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    for (; a < oldz; a += PGSIZE) {
        pte_t *pte = walkpgdir(pgdir, (char *) a, 0);
        // ...
    }
    // ...
}
```

The page table entry contains the page's physical address, plus some flags to determine whether it's mapped and what permissions are set for it.

Now, a virtual address space isn't laid out contiguously. Think about it: if you sit back and imagine a user process hanging out in memory, what does that address space look like? You're probably imagining the stack at one end of memory and the heap at the other, with each growing toward the center, right? so there will be some pages in the center that aren't mapped; some of the page tables might not exist either, in which case `walkpgdir()` would return a null pointer.

Remember we agreed to never dereference null pointers? Yeah, so we'll have to skip all those unmapped pages. If we got a null pointer, then that means the entire page table doesn't exist, so we need to skip forward to the next page directory entry (and thus the next page table). We'll have to move `a` to the virtual address that corresponds to that next page directory entry.

We can get the page directory index from `a` with the `PDX()` macro we've seen seen before, and then just add 1 to get the next entry in the page directory. Now we need to turn that back into a virtual address. We'll use a new macro, `PGADDR()` (also from mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h)), to do that. So then we'll continue to the next loop iteration, which will get the page table entry for this new virtual address.

Wait wait wait, one last thing! After all that, `a` should now be the first virtual address in the page table for the new page directory entry... except it's get `PGSIZE` added to it because of the for loop's update statement.

Ugh, okay, fine, this is annoying. Let's just fix it with a hack: subtract `PGSIZE` from it now, so that it gets incremented to the right value in the next iteration. Okay, that's it, I swear!

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    for (; a < oldsz; a += PGSIZE) {
        pte_t *pte = walkpgdir(pgdir, (char *) a, 0);

        if (!pte) {
            a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
        } else {
            // ...
        }
    }
    // ...
}
```

Okay, now the else branch: if we don't get a null pointer then at least the page table exists, but that doesn't mean the page itself is mapped. If it's not, then we don't need to do anything else, but if it is mapped, then we need to free it. We can get the page's physical address out of the page table entry with the `PTE_ADDR` macro then make sure it's not null.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    for (; a < oldsz; a += PGSIZE) {
        // ...
        if (!pte) {
            // ...
        } else if ((*pte & PTE_P) != 0) {
            uint pa = PTE_ADDR(*pte);
            if (pa == 0) {
                panic("kfree");
            }
            // ...
        }
    }
    // ...
}
```

The whole point of this was to be able to call `kfree()`, remember? So let's convert `pa` to a kernel virtual address as a `char *` and free it. Then after the loop is done, we'll return the new size.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    for (; a < oldsz; a += PGSIZE) {
        // ...
        if (!pte) {
            // ...
        } else if ((*pte & PTE_P) != 0) {
            // ...
            char *v = P2V(pa);
            kfree(v);
            *pte = 0;
        }
    }
    return newsz;
}
```

allocuvm

This is the reverse of `deallocuvm()`: instead of freeing pages with `kfree()`, we'll allocate them with `kalloc()`. Here too, we start by checking for integer overflow by making sure `newsz` really is larger than `oldsz`. But now we also have to check that we're not gonna grow the process's size into the region where it could access kernel memory; otherwise it might read or modify arbitrary physical memory.

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    if (newsz >= KERNBASE) {
        return 0;
    }
    if (newsz < oldsz) {
        return oldsz;
    }
    // ...
}
```

We're gonna start adding new pages right after `oldsz`, so we have to align that to a page boundary:

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    uint a = PGROUNDUP(oldsz);
    // ...
}
```

The for loop is easier this time around because we already know that the pages aren't mapped. First we allocate a new page. Any call to `kalloc()` needs two things after, remember? We have to check for null, in which case we print an error message to the console (that's `cprintf()`; we'll get to that in the devices section), then undo any allocations we made and return 0. Then we have to zero the page because we filled it with 1s when it was freed.

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    // ...
    for (; a < newsz; a += PGSIZE) {
        char *mem = kalloc();
        if (mem == 0) {
            cprintf("allocuvm out of memory\n");
            deallocuvm(pgdir, newsz, oldsz);
            return 0;
        }

        memset(mem, 0, PGSIZE);
        // ...
    }
    // ...
}
```

We have a page now, but it's not yet mapped in the page directory. We can do that with `mappages()`; that might fail too (because it needs to allocate more pages for the page tables), in which case we do the same as before. Then after the for loop is done, we return the new size.

```
int allocuvm(pde_t *pgdir), uint oldsz, uint newsz) {
    // ...
    for (; a < newsz; a += PGSIZE) {
        // ...

        if (mappages(pgdir, (char *) a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0) {
            cprintf("allocuvm out of memory (2)\n");
            deallocuvm(pgdir, newsz, oldsz);
            kfree(mem);
            return 0;
        }
    }
    return newsz;
}
```

freevm

This function will get rid of a user page directory that we no longer need. Now that we have `deallocuvm()`, it's easy: we just "shrink" the process to a size of zero. Oh and we'll remember the lessons our ancestors have taught us and make sure the pointer to the page directory isn't null before dereferencing it.

```
void freevm(pde_t *pgdir)
{
    if (pgdir == 0) {
        panic("freevm: no pgdir");
    }
    deallocuvm(pgdir, KERNBASE, 0);
    // ...
}
```

Great, so all pages are freed, and we're done!

Now hang on a sec... The page directory itself resides in memory; so do the page tables. We have to free those too. We'll start with the page tables; freeing the page directory first would be a use-after-free vulnerability because we'd need to use it to get to the page tables.

We'll iterate over the page directory's entries, checking whether each one has the "present" flag set (`NPDENTRIES` is defined as 1024 in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/types.h)). If it does, we'll get the page table's physical address from it with the `PTE_ADDR()` macro, then convert that to a virtual address as a `char *` to make `kfree()` happy. We don't have to worry about clearing the "present" flag in the page directory because it's about to be freed anyway.

```
void freevm(pde_t *pgdir)
{
    // ...
    for (uint i = 0; i < NPDENTRIES; i++) {
        if (pgdir[i] & PTE_P) {
            char *v = P2V(PTE_ADDR(pgdir[i]));
            kfree(v);
        }
    }
    // ...
}
```

We wrap up by freeing the page directory itself.

```
void freevm(pde_t *pgdir)
{
    // ...
    kfree((char *) pgdir);
}
```

copyuvm

The `fork()` system call will need to "clone" a process, which includes its virtual address space. This function takes a pointer to the parent process's page directory and the size of the parent process's address space and returns a pointer to a fresh new page directory with everything set up exactly the same.

We start by creating a new page directory and taking care of the kernel's half of the address space with `setupkvm()`. That might fail if it can't allocate a new page, so we have to check for null. Sigh. C code is approximately 40% checking for null return values.

```
pde_t *copyuvm(pde_t *pgdir, uint sz)
{
    pde_t *d;
    if ((d = setupkvm()) == 0) {
        return 0;
    }
    // ...
}
```

Now we'll iterate over the user portion of the parent process's address space from 0 to `sz`, copying everything over as we go. Say we want to copy a page from the parent's virtual address `i` to the child's address `i` (note that they'll map to different physical addresses). We'll have to figure out the corresponding kernel virtual address for the parent's `i` in order

to do that, so we use `walkpgdir()` to get the page table entry, then get the page's physical address.

```
pde_t *copyuvm(pde_t *pgdir, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        pte_t *pte;
        if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) {
            panic("copyuvm: pte should exist");
        }
        if (!(*pte & PTE_P)) {
            panic("copyuvm: page not present");
        }

        uint pa = PTE_ADDR(*pte);
        uint flags = PTE_FLAGS(*pte);
        // ...
    }
    // ...
}
```

In this case we know the parent process is already set up, so we don't really have to worry about `walkpgdir()` failing and returning null, but it's bad C juju to ignore a possibly-null return value, so we just panic if it does fail or if the page isn't present.

Next we allocate a page for the child process (checking for null again...) and copy everything from the parent's page to the new child page.

```
pde_t *copyuvm(pde_t *pgdir, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        // ...
        char *mem;
        if ((mem = kalloc()) == 0) {
            goto bad;
        }
        memmove(mem, (char *) P2V(pa), PGSIZE);
        // ...
    }
    // ...
}
```

You might recognize `memmove()` as a C standard library function that copies the contents of one memory address into another, but we can't use those, remember? So xv6 provides its own implementation of it in string.c (https://github.com/mit-pdos/xv6-public/blob/master/string.c).

If you haven't seen a `goto` statement before, it's basically a holdover from ye olde days before Edsger Dijkstra preached the gospel of structured programming to the world and invented the if statement. It does exactly what it sounds like: you make a label somewhere in code and it takes you there.

Next we stick that new page into the child's page directory, checking for null again. If `mappages()` fails, then the new page won't be in the page directory, so we have to free it here or else we'll never be able to find it again: a memory leak.

```
pde_t *copyuvm(pde_t *pgdir, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        // ...
        if (mappages(d, (void *) i, PGSIZE, V2P(mem), flags) < 0) {
            kfree(mem);
            goto bad;
        }
    }
    // ...
}
```

If none of the allocations failed, we just return a pointer to the new page directory. But if something went wrong, then one of those `goto` statements will send us to the time out corner of `bad`, where we undo all our work by freeing the page directory and returning a null pointer.

```
pde_t *copyuvm(pde_t *pgdir, uint sz)
{
    // ...
    return d;

bad:
    freevm(d);
    return 0;
}
```

Great, another function we'll have to check for null.

switchuvm

Okay, we've got a way to create a new process page directory. We also have a way to switch to using the kernel page directory `kpgdir` with `switchkvm()`. But we need a way to switch to using the process page directory too. Enter `switchuvm()`.

I'll warn you -- `switchkvm()` was nice and short, but `switchuvm()` is an ugly one for sure.

The argument to this function is a pointer to a `struct proc`, which represents a process. We'll talk about that more when we get to processes; two fields are important now: `p->kstack` which holds a pointer to the kernel stack for that process, and `p->pgdir`, which points to that process's page directory.

Okay, well let's start with some sanity checks to make sure that the process `p` actually exists (the pointer is non-null) and its kernel stack and page directory pointers are non-null too.

```
void switchuvm(struct proc *p)
{
    if (p == 0) {
        panic("switchuvm: no process");
    }
    if (p->kstack == 0) {
        panic("switchuvm: no kstack");
    }
    if (p->kstack == 0) {
        panic("switchuvm: no pgdir");
    }
    // ...
}
```

The main function of loading the process's page directory will be the same as in `switchkvm()`: just an `lcr3` instruction. But the difference now is that the x86 architecture requires some additional bookkeeping for processes.

See, when the kernel runs a new process, the CPU will start executing different instructions. But it needs a way to keep track of where it left off in the kernel code so that it can pick the thread back up after the process is done executing. Similarly, interrupts and system calls might change the running process, so the CPU needs to record some metadata about the process's state too before switching to another one. x86 does that by means of a structure called a *Task State Segment*, or TSS.

The TSS holds information like the current state of certain registers (e.g., `%esp`, `%eip`, `%cr3`, etc.), segment descriptors (`%cs`, `%ss`, `%ds`, etc.), the current privilege leve, and I/O privilege levels -- in other words, the process's *context*. It can be located anywhere in memory, but the processor needs to find it, so it uses an entry in the GDT called the TSS segment descriptor that points to the TSS. Remember the GDT from way back when we were talking about segmentation? Good times. The CPU holds a pointer to the GDT's TSS entry in a special register called the task register.

Back in the segmentation days of our youth, we stored the GDT in a `struct cpu` that held information about the current processor. We got that `struct cpu` by calling a `mycpu()` function. We're gonna do the same thing here in order to update the GDT with a segment for the TSS. Getting interrupted in the middle of this might be disastrous: the TSS would be half-updated, so who knows what would happen when the CPU tried to resume execution where it last left off. So we'll use the `pushcli()` and `popcli()` functions we saw with spin-locks to temporarily disable interrupts.

```
void switchuvm(struct proc *p)
{
    // ...
    pushcli();

    mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts, sizeof(mycpu()->ts)-1, 0);
    mycpu()->gdt[SEG_TSS].s = 0;
    // ...

    popcli();
}
```

Whoa okay what is this?

We've seen the `SEG()` and `SEG_ASM()` macros before; they created GDT segments. `SEG16()` does the same with 16 bits (it's defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h)). `STS_T32A` is a flag that sets the segment's type as an available 32-bit TSS. Then we pass in a pointer to the task state with `&mycpu()->ts`, its size, and a descriptor privilege level of 0 (which means ring 0, the kernel level). The GDT's `.s` field is a one-bit flag to determine whether this is a system or application segment, so we set it to system.

Okay, so now the GDT points to the task state. Next we need to update the task state, then load it into the CPU. We'll start by storing a segment selector and the stack pointer in the task state; these should look familiar from the boot loader and `seginit()`.

```
void switchuvm(struct proc *p)
{
    // ...
    mycpu()->ts.ss0 = SEG_KDATA << 3;
    mycpu()->ts.esp0 = (uint) p->kstack + KSTACKSIZE;
    // ...
}
```

The TSS can also specify permissions for accessing I/O ports: for example, setting the I/O privilege level to 0 in the `eflags` register *and* setting a part of the TSS called the I/O map base address to an address beyond the TSS segment forbids I/O instructions like `inb` and `outb` from user space. So we'll set the I/O map base address next.

```
void switchuvm(struct proc *p)
{
    // ...
    mycpu()->ts.iomb = (ushort) 0xFFFF;
    // ...
}
```

So now we have a GDT entry pointing to the TSS, which is now updated. Now we just load it into the task register with the x86 instruction `ltr`; here we use a C wrapper for that assembly instruction, defined in [x86.h (https://github.com/mit-pdos/xv6-public/blob/master/x86.h)](https://github.com/mit-pdos/xv6-public/blob/master/x86.h).

```
void switchuvm(struct proc *p)
{
    // ...
    ltr(SEG_TSS << 3);
    // ...
}
```

Finally, the last thing we do before re-enabling interrupts is to load the process's page directory into the `%cr3` register so we can start using it.

```
void switchuvm(struct proc *p)
{
    // ...
    lcr3(V2P(p->pgdir));
    // ...
}
```

loaduvm

Okay, this is another function that's gonna require extra info we haven't seen yet, but I'm gonna make it a bit easier by waving my hands around and glossing over the details. It's gonna read a program from a file into memory at virtual address `addr` using page directory `pgdir`. The part we want to read has size `sz` and is located at position `offset` within the file.

Now, what about the file? We'll talk more when we get to the file system code, but for now let's just say that files are represented in xv6 as `struct inode`s, and we can read from them with the function `readi()`.

We're gonna run the program from this code, so the address it's stored in needs to be page-aligned.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    if ((uint) addr % PGSIZE != 0) {
        panic("loaduvm: addr must be page aligned");
    }
    // ...
}
```

Next we're gonna iterate over pages starting from `addr`, reading from the file in `ip` into that page. As usual, we'll need to get the kernel virtual address from the user address `addr`, so we start by getting the page table entry via `walkpgdir()`, checking for a null pointer if the corresponding page table doesn't exist. Then we can turn that into a physical address.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        // Get the page table entry
        pte_t *pte;
        if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
            panic("loaduvm: address should exist");
        }
        // Get the page's physical address
        uint pa = PTE_ADDR(*pte);

        // ...
    }
    // ...
}
```

Now we want to read from the file one page at a time using `readi()`, which takes a pointer to an inode (here, `ip`), a kernel virtual address (`P2V(pa)`), the location within the file of the segment we want to read (`offset + i`), and the segment's size.

Now we want to read from the file one page at a time using `readi()`. We have to specify a size in bytes to read; if the remaining unread part of the segment is larger than a page, then the size we pass to `readi()` should be `PGSIZE`, but otherwise it'll be less. So we'll compare `sz` to `i` and define define `n` accordingly.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        // ...
        uint n;
        if (sz - i < PGSIZE) {
            n = sz - i;
        } else {
            n = PGSIZE;
        }
        // ...
    }
    // ...
}
```

The other arguments to `readi()` are a pointer to an inode (`ip`), a kernel virtual address (`P2V(pa)`), and the location within the file of the segment we want to read (`offset + i`). It returns the number of bytes read, so if it's not `n` we'll report an error by returning -1. Otherwise we return 0 after the for loop is done.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    // ...
    for (uint i = 0; i < sz; i += PGSIZE) {
        // ...
        if (readi(ip, P2V(pa), offset + i, n) != n) {
            return -1;
        }
    }
    return 0;
}
```

inituvm

Okay, the next three are nice and easy! This next one is pretty similar to `loaduvm()`, except instead of loading program code from disk, it copies it in from memory. We'll take `sz` bytes from a source address of `init` and stick it in address 0 of the process's page directory `pgdir`.

This function is also easier because we're only gonna call it for programs that are less than one page in size, so we don't have to worry about looping over pages or anything like that. I like it when xv6 keeps things simple.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
    if (sz >= PGSIZE) {
        panic("inituvm: more than a page");
    }
    // ...
}
```

Next we allocate a fresh page of memory, zero it to clear the garbage values, and stick it into `pgdir` at address 0.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
    // ...
    char *mem = kalloc();
    memset(mem, 0, PGSIZE);
    mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W | PTE_U);
    // ...
}
```

And we wrap up by actually loading the code from `init` into the new page.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
    // ...
    memmove(mem, init, sz);
}
```

clearpteu

This function takes a page directory and a user virtual address and clears the "user-accessible" flag so that the process can't touch it. It's used to create an inaccessible page below a new process's stack to guard against stack overflows; this way, a stack overflow will cause a page fault instead of silently overwriting memory.

The `PTE_U` flag is in the page table entry, so we'll have to get that, then set the flag.

```
void clearpteu(pde_t *pgdir, char *uva)
{
    // Get the page table entry
    pte_t *pte = walkpgdir(pgdir, uva, 0);
    if (pte == 0) {
        panic("clearpteu");
    }
    // Clear the user permission flag
    *pte &= ~PTE_U;
}
```

Here `&` is a bitwise-AND and `~` is a bitwise-NOT; for reference, `|` is bitwise-OR and `^` is bitwise-XOR. Contrast these with their logical versions, `&&`, `!`, and `||` (XOR has no logical version). C also has corresponding assignment operators (similar to `+=`, `-=`, `*=`, etc.) for each of them. So the last line of code is equivalent to `*pte = *pte & (~PTE_U)`.

uva2ka

We often need to convert user virtual addresses to kernel ones; `uva2ka()` is a short helper function that does that while checking that the page is actually present and has the user permission flag set.

We'll call walkpgdir to get the page table entry, then check both permission bits before recovering the page address with `PTE_ADDR()` and converting it to a kernel virtual address. We'll return the kernel virtual address as a `char *`, or null if either flag is not set.

```
char *uva2ka(pde_t *pgdir, char *uva)
{
    pte_t *pte = walkpgdir(pgdir, uva, 0);

    if ((*pte & PTE_P) == 0) {  // check that it's present
        return 0;
    }
    if ((*pte & PTE_U) == 0) {  // check that it's user-accessible
        return 0;
    }
    return (char *) P2V(PTE_ADDR(*pte));
}
```

Let me ask you a weird question: how are you feeling right now?

Okay, that was a test of your C coding practices, because if you took those null checks to heart, you should be *really* uncomfortable right about now.

Check it out: `walkpgdir()` returns a pointer to the page table entry. *Any* time a function returns a pointer, you should immediately ask yourself whether that function can return a null pointer. Tons of C functions report an error by returning null. In this case, we *know* `walkpgdir()` can fail and report null if the page table doesn't exist, so we *know* we might get a null pointer out of it -- it'll happen whenever a page table doesn't exist. So what do we do with that knowledge?

Why, we go right ahead and dereference that pointer. WKBW;NQ39Q2A4T8YHMFGRW!!!

Dereferencing a null pointer is undefined behavior. There's literally no telling what might happen. It can cause all kinds of bugs from segmentation faults to security vulnerabilities.

All those null checks in the other functions serve a purpose: if something goes wrong and a function returns a null pointer, they catch it before it gets dereferenced, then either handle it gracefully or simply propagate the error by returning null (or some other error code) and let the caller figure out what to do with it.

Omitting a check for a null pointer like `uva2ka()` does is bad practice in C because it means the programmer has to *guarantee* -- by manually checking -- that no call to this function could *ever possibly* cause a null return value. Except humans are dumb, dumb creatures who make mistakes all the time, especially in big projects: there's no way you'd be able to remember that tiny little detail two years later when you decide to refactor your code or add a new feature or something.

But maybe you can note that in the comments? Okay yeah, but think about it: how often do you go and look up the source code for every single function you call? Yeah, I thought so.

This is why C is so dangerous: there are hundreds of such problems that you need to be aware of and remember to add stuff like null pointer checks to your code. If you don't because you're a normal human who forgets things sometimes, then you'll need to remember that you forgot to do it before and manually check every single call to your code and think about every possible edge case that a malicious adversary might exploit.

Good thing no one ever makes these mistakes in C, or we'd see enormous security vulnerabilities being reported every single day in all kinds of critical software. Oh wait...

So if you ever find yourself looking at C during code review and you come across a function that returns a pointer, you should stop what you're doing and look up the documentation for that function. If that function has any chance of returning a null pointer, then you should yell and kick and scream until somebody adds a null check and figures out how they want to handle it if it's null. Is this annoying? Yes. Hard to remember? Yes. But that's C. *(cough cough use Rust instead cough cough...)*

Now, the xv6 authors are so awesome that I'm gonna give them the benefit of the doubt and assume they left it off because they hand-checked every call to make sure it would never be an issue. But you and me? Nah.

The point of my rant is this: if you're reading this, then you're probably gonna find yourself hacking away at xv6 for a project sooner or later. When you do that, you should treat this function as VERBOTEN. You're not allowed to touch it or call it, at least until you add a null check to it yourself.

The same goes for any functions that call this one, because maybe all the existing calls to `uva2ka()` are fine right now, but then you make some tiny change and now it's no longer guaranteed to never be null. For reference, this function currently only gets called by `copyout()`, and that one only gets called by `exec()`. `exec()` gets called by `sys_exec()`,

the shell, and the initial user-space program `init`. So be careful if you touch any of those.

Whew, okay, /rant.

copyout

This function copies `len` bytes of data from a kernel virtual address `p` to a user virtual address `va` using page directory `pgdir`. `exec()` will use this to copy command-line arguments to the stack for a program it's about to run.

You might be wondering why it's needed -- doesn't `memmove()` do the same thing? Almost, but the difficulty is that `pgdir` may not be the current page directory, so we'll have to manually translate the virtual address `va`. That's where `uva2ka()` comes in, plus it ensures that the page for `va` has the right flags set. *Then* we can use `memmove()`.

First, `p` will be the source address, but `memmove()` requires a `char *` in order to copy data byte-by-byte, so let's convert it now:

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
    char *buf = (char *) p;
    // ...
}
```

Next we need to get the kernel virtual address corresponding to `va`, but there's a challenge: what if the data crosses a page table boundary? It might be spread across separate locations in physical memory (and thus in kernel virtual memory too). So we'll need a loop in which each iteration gets the next kernel virtual address and copies whatever part of the data is in this page.

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
    // ...
    while (len > 0) {
        // ...
        len -= n;
        buf += n;
        // ...
    }
    // ...
}
```

So we'll start each iteration by making `va0` the base address of the page `va` is on and `pa0` the kernel address of `va0`, converted with `uva2ka()`. I... honestly don't know why they used `pa0` as an identifier here. It makes it look like it should be a physical address, but it's not; it's a kernel virtual address. Sigh. Anyway, the call to `uva2ka()` might fail if the page isn't present or it doesn't have a user permission bit, so we have to check for a null pointer and return -1 if we find one.

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
    // ...
    while (len > 0) {
        uint va0 = (uint) PGROUNDDOWN(va);

        char *pa0 = uva2ka(pgdir, (char *) va0);
        if (pa0 == 0) {
            return -1;
        }

        // ...
        va = va0 + PGSIZE;
    }
    // ...
}
```

Now `va` is in between `va0` and the next page, so the length of the data within this page is `PGSIZE - (va - va0)`, unless it's the last page, in which case we should pick the lesser of this value and `len` (since `len` gets decremented on each iteration through the loop).

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
    // ...
    while (len > 0) {
        // ...
        uint n = PGSIZE - (va - va0);
        if (n > len) {
            n = len;
        }
        // ...
    }
    // ...
}
```

Finally, we copy the data from `buf` into the target kernel virtual address for `va`. Hmm, we don't have that yet. Oh wait, `pa0` is the kernel virtual address for `va0`, and `va` is just `va-va0` bytes after that, so we'll use it.

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
    // ...
    while (len > 0) {
        // ...
        memmove(pa0 + (va - va0), buf, n);
        // ...
    }
    return 0;
}
```

We return 0 if everything went okay.

Summary

Okay, that was a lot of helper functions, but we're ALL DONE with virtual memory! From now on, we have all the tools we'll need to manage memory and set up virtual address spaces for new processes.

Processes

It's time to turn our attention to processes in xv6! proc.c (https://github.com/mit-pdos/xv6-public/blob/master/proc.c) is another huge file, so I'm gonna split it up into a few posts. This one will focus on the basic functions we'll need in order to create new processes; later posts will go over scheduling and system calls.

proc.h

I haven't spent much time on the header files in xv6, but proc.h (https://github.com/mit-pdos/xv6-public/blob/master/proc.h) defines some important structures we're gonna be using often, so let's just get those out of the way first.

Let's start off with the definition for `struct context`. The processor will have to switch between different processes during interrupts, system calls, exceptions, etc.; these *context switches* will require saving the contents of some of the CPU registers so that it can reload them when it switches back and resume execution where it left off. It'll save the process's context by pushing those register contents on the stack; that way the stack pointer is effectively a pointer to the context. So the fields of a `struct context` will just list all the registers that were saved on the stack.

Now, which registers do we need to save? Let's look at the full list on the OSDev Wiki (https://wiki.osdev.org/CPU_Registers_x86). We've got some general- purpose registers, the instruction pointer register `%eip`, segment registers, a flags register, control registers, and the GDT and IDT registers (x86 doesn't use the debug, test, or LDT registers).

The flags register, control registers, and GDT/IDT registers shouldn't change between processes, so we don't need to save those. What about the segment registers like `%cs`? Back when we set up segmentation, we made the segments be identity maps that would always stay the same for all processes. There are separate segments for user mode and kernel mode, but context switches will always occur in kernel mode, so the segment registers shouldn't change, and we don't need to save them either.

We should definitely save the program counter (AKA instruction pointer `%eip`), since that will point to the place in the code where we should resume execution.

The only ones left now are the general-purpose registers: the stack base pointer `%ebp` and stack pointer `%esp`, along with `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`. We said above that the stack pointer `%esp` would tell us where to find the context, so that must mean we'll already have it through some other means in order to find the rest of the context, so we don't need to save it again (we'll see how we end up getting it later on). But we do need to save `%ebp`.

There's an x86 convention that the caller of a function has to save `%eax`, `%ecx`, and `%edx`, so those are already taken care of. So we'll just save the others: `%edi`, `%esi`, and `%ebx`.

We end up with this list of saved registers as the fields for `struct context`:

```
// ...
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uip eip;
};
// ...
```

Next up: we might end up with a bunch of processes, some of which are currently running while others aren't. Let's set up some labels to note that. We'll definitely need a `RUNNING` label; we'll also use one called `RUNNABLE` for processes that are ready to be run the next time there's a free CPU. We also need a label for processes that are blocked waiting for something else to happen (e.g., I/O); xv6 calls this `SLEEPING`. Processes that don't exist yet will be called `UNUSED`.

There are two special moments in a process's lifecycle that we should be careful with: birth and death. When we create a new process, we'll have to do a bunch of setup before it's `RUNNABLE`; killing a process requires clean-up before it goes back to `UNUSED`. We'll call those `EMBRYO` and `ZOMBIE`, respectively.

We could use bit flags for these states or just regular integers, except then we'd have to do annoying bit arithmetic or keep track of which number represents which state. And yes, we could use a bunch of `#define` directives for the preprocessor for that, but there's a better way to do it. C lets us create data types for labels using `enum`s. These don't have fields like `struct`s do; they're basically just a mapping between integers and what the labels those integers represent. So it's pretty similar to using a bunch of `#define` directives, except that they're all defined neatly in a single place, so it helps us remember they're all representing the same idea. So we'll use an `enum` like this:

```
// ...
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// ...
```

Now it's time to look at how we'll represent processes themselves together with their metadata. Let's see... what kind of unique data does each process have? We just talked about `struct context`s and `enum procstate`s; each process will have both of those.

We also talked about virtual memory for processes in a previous post, so it should also have its own page directory and stack for the kernel to use, plus a way to track the size of its virtual address space. We said then that processes are created using `fork()`, so let's add a field to point to the parent process.

We'll need a way for the kernel to refer to a process, so let's give it a unique process ID. That's not super helpful when it comes to debugging, so let's also add a name for it as a string.

The rest of the fields are for aspects we haven't seen yet but will talk about soon: a *trap frame* for interrupts and system calls, a boolean to track whether a process should be killed soon, a *channel* to be able to wake up a sleeping process, an array of open files, and a current working directory.

```
// ...
struct proc {
    uint sz;                     // size (in bytes) of virtual address space
    pde_t *pgdir;                // page directory
    char *kstack;                // kernel stack for this process
    enum procstate state;        // process state
    int pid;                     // process ID
    struct proc *parent;         // parent process
    struct trapframe *tf;        // trap frame for current system call
    struct context *context;     // saved register contents for context switches
    void *chan;                  // channel that process is sleeping on, if any
    int killed;                  // boolean: should process be killed soon?
    struct file *ofile[NOFILE]; // array of open files
    struct inode *cwd;           // current working directory
    char name[16];               // process name
}
// ...
```

Okay, next we'll add another structure for metadata representing each CPU.

If you read the previous post, then you know each CPU has its own local interrupt controller with a unique ID, so we'll write that down. The post about process paging talked about the TSS, so we'll need one of those per CPU, plus a GDT too.

At any point in time, a processor will be running one of: its own initialization routine (only once while the kernel is setting up), a user process (or any interrupts or system calls that come up), or a scheduler routine to run the next process. So let's add a pointer to a `struct proc`, which will be null if it's not running a process; a boolean `started` will be false until the CPU finishes its own set-up. The scheduler isn't itself a process; it uses the `kpgdir` page directory and has its own context, so we'll store that context in a field here.

Finally: remember how the spin-lock post talked about nested calls to `pushcli()` and `popcli()` tracking whether interrupts were enabled before the first call to `pushcli()`, and only enabling interrupts after the last call to `popcli()` if they were enabled before? Those were tracked with per-CPU fields `ncli` and `intena`, so we need those too.

```
struct cpu {
    uchar apicid;               // ID of this CPU's local interrupt controller
    struct context *scheduler;  // scheduler's context
    struct taskstate ts;        // task state segment
    struct segdesc gdt[NSEGS];  // global descriptor table
    volatile uint started;      // boolean: has this CPU been initialized yet?
    int ncli;                   // depth of pushcli() nesting
    int intena;                 // were interrupts enabled before pushcli()?
    struct proc *proc;          // currently running process
};
// ...
```

Last but not least, we'll add declarations for the global array of CPUs and the number of CPUs actually present on this machine; these were defined in mp.c (https://github.com/mit-pdos/xv6-public/blob/master/mp.c)

Okay, on to the functions now!

proc.c

xv6 uses a global process table with an array of processes to store all the `struct proc`s in; this means we'll never be able to create more processes than the number of entries in the array, `NPROC`, defined in param.h (https://github.com/mit-pdos/xv6-public/blob/master/param.h) as 64. We'll need a lock too to prevent data races while accessing the process table. The process table's definition does that thing again where you simultaneously define a `struct type` and define a variable using that type in a single statement.

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

Then we define a global static variable to point to the first process that gets run on xv6, so that other files can set it up.

```
// ...
static struct proc *initproc;
// ...
```

Finally, we're gonna need to assign unique process IDs, so we'll use a global counter to know which one we should use next.

```
// ...
int nextpid = 1;
// ...
```

pinit

This function only does one thing: initializes the lock in the process table.

```
void pinit(void)
{
    initlock(&ptable.lock, "ptable");
}
```

mycpu

This function will return a pointer to the `struct cpu` for the current CPU. There's a potential concurrency bug with this function: if it gets interrupted before it returns, then it might get rescheduled on a different CPU, and end up returning an incorrect `struct cpu`. So we need to make sure that interrupts are disabled when we call it. Normally we'd do that with `pushcli()` and `popcli()`, but those functions actually call this one, so we'd get an infinite recursion. So instead we're just gonna have to remember to disable interrupts *before* calling this function.

If you're reading this because you're gonna do some xv6 kernel hacking for an OSTEP project or something, you should read that as "DANGER DANGER DANGER!". If your code calls this function, or calls any other functions that in turn call this one, you *have* to make sure you've disabled interrupts first.

Concurrency bugs are a nightmare because they're not deterministic: for example, if you forget to disable interrupts before calling this function, it might work just fine most of the time until the one unlucky moment when it gets interrupted and rescheduled on a different CPU. So let's make this easier to debug by starting off with a check that interrupts are disabled and panic if they're not. We can check whether the interrupt flag `FL_IF` is set in the `eflags` register.

```
struct cpu *mycpu(void)
{
    if (readeflags() & FL_IF) {
        panic("mycpu called with interrupts enabled\n");
    }
    // ...
}
```

Okay so how do we figure out which CPU we're on? Well, the previous post talked about interrupt controllers; each CPU has a local interrupt controller with a unique ID which we can get with `lapicid()`. Once we have that, we can iterate over the CPU array `cpus` until we find an entry with a matching `apicid`; we'll just panic if none of them match.

```
struct cpu *mycpu(void)
{
    // ...
    int apicid = lapicid();

    for (int i = 0; i < ncpu; ++i) {
        if (cpus[i].apidid == apicid) {
            return &cpus[i];
        }
    }
    panic("unknown apicid\n");
}
```

cpuid

Those local interrupt controller IDs aren't guaranteed to start from 0, so we'll need another way to identify CPUs. We can just use its entry number in the global `cpus` array for that; `cpus` is an array of `struct cpu`s, which in C means it's really a pointer to the entry with index 0. `mycpu()` returns a pointer to the entry for the current CPU, so we can just subtract those pointers to get the index.

```
int cpuid(void)
{
    return mycpu() - cpus;
}
```

myproc

This function returns a pointer to the `struct proc` running on this CPU. We're gonna call `mycpu()` here, so we'll be good and remember to dsable interrupts first with `pushcli()` and reenable them at the end with `popcli()`. Then we'll get the current process from the `struct cpu`'s field.

```
struct proc *myproc(void)
{
    pushcli();

    struct cpu *c = mycpu();
    struct proc *p = c->proc;

    popcli();
    return p;
}
```

allocproc

Okay, we're finally at the code to create a new process! Whew, it's been a long journey.

This is a `static` function, which means it can only be called by functions defined in this same file. Creating a new process will require modifying the process table, so we need to grab the lock so that other threads can't mess with it while we're using it.

```
static struct proc *allocproc(void)
{
    acquire(&ptable.lock);
    // ...
}
```

Now we need to look through the table and find a slot that's `UNUSED`; if we find on, then great, we'll assign that slot to the new process after the `found` label below. But if none of them are free, we'll have to return a null pointer to indicate that. You know what that means, right? Yup, we're gonna have to add null checks every time we call this function! Wooooo!

```
static struct proc *allocproc(void)
{
    // ...
    struct proc *p;

    // Look through process table looking for an UNUSED slot
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == UNUSED) {
            goto found;
        }
    }
    // If none is found, return null pointer
    release(&ptable.lock);
    return 0;

    // ...
}
```

Check out that for loop too: `p` is a pointer to a `struct proc` that starts off pointing to `ptable.proc`; that means it points to the entry and index 0. Then it gets incremented by 1 each iteration; since it's a `struct proc`, the pointer arithmetic will work out so that it points to the next entry in the process table.

Okay now let's check out the `found` label and see what happens if we did find an unused slot. First we set its state to `EMBRYO` (instead of `RUNNABLE`, since we're not done setting it up) and give it a PID. That state means it's neither `UNUSED` nor `RUNNABLE`, so we can be confident that any other threads wouldn't try messing with it right now; they can't allocate the slot to another process, and they can't try to run it yet. So we can stop hogging the process table now and let other threads take a turn.

```
static struct proc *allocproc(void)
{
    // ...
found:
    p->state = EMBRYO;
    p->pid = nextpid++;

    release(&ptable.lock);

    // ...
}
```

Now we need to allocate a page for this process's kernel thread to use as a stack. Remember, `kalloc()` can return null, so we need a null check here.

```
static struct proc *allocproc(void)
{
    // ...
found:
    // ...
    if ((p->kstack = kalloc()) == 0) {
        p->state = UNUSED;
        return 0;
    }
    // ...
}
```

Now, we're not gonna set up its page directory yet; that'll happen in `fork()`, which we'll see later on. But we do need to set up the process so that it'll start executing code somewhere. It needs to start off in kernel mode, then it'll context-switch back into user mode and start running its code.

We haven't looked at the mechanics of context switches yet, so I'll spoil it a little now (I know, I'm sorry). When a process is already running, it can send a system call to ask for the kernel's attention to do whatever it needs, like a baby crying until it gets fed or changed or burped or whatever. Then it'll switch into kernel mode to run the system call, then switch back to where it left off and pick up from there.

Well, xv6 is all about simplicity, right? And what's more simple and elegant than treating a special case (creating a new process and starting it off running some code) the same as the general case (returning from a system call)? So xv6 will set up every new process to start off by "returning" from a (non-existent) system call. That way the context switch code can be reused for new processes too.

New processes are created via `fork()`, so we'll return into a function called `forkret()`. Then that has to return into the function `trapret()`, which closes out a *trap* (interrupt, system call, or exception) by restoring saved registers and switching into user mode. We'll get to `forkret()` and `trapret()` soon.

But first, the challenge: how do we "return" into a function that never called us in the first place? We talked about function calls in x86 in the post on spin-locks with the `getcallerpcs()` function, so make sure to read that now if you need a refresher.

To summarize: when a function `f()` calls another function `g()`, it pushes the arguments of `g()` on the top of its stack. Then it pushes a return address to know where it should continue running the code of `f()` after `g()` returns; that's just the `%eip` register. Then it pushes the base address of the stack for `f()`, i.e. the current `%ebp` register. That's where `g()`'s stack will start off.

When the scheduler first runs the new process, it'll check its context via `p->context` to get its register contents, including the instruction pointer `%eip`. So if we want it to start executing the code in `forkret()`, the `eip` field of its context should point to the beginning of `forkret()`. Then we can trick it into thinking that the previous caller was `trapret()` by setting up arguments and a return address in its stack.

Let's start off by getting a pointer to the bottom of the stack. We had just allocated a new stack page at `p->kstack`, but the stack grows from high to low addresses, so the base of the stack is really at `p->kstack + KSTACKSIZE`. We'll make it a `char *` so we can move around one byte at a time using pointer arithmetic.

```
static struct proc *allocproc(void)
{
    // ...
found:
    // ...
    char *sp = p->stack + KSTACKSIZE;
    // ...
}
```

Now we should push any arguments for `trapret()` on the stack; it takes a `struct trapframe` (which we'll go over later), so we'll leave some room for it and make the process point to it with `p->tf`.

```
static struct proc *allocproc(void)
{
    // ...
found:
    // ...
    sp -= sizeof(*p->tf);
    p->tf = (struct trapframe *) sp;
    // ...
}
```

Then we add a "return address" to the beginning of `trapret()` after that.

```
static struct proc *allocproc(void)
{
    // ...
found:
    // ...
    sp -= 4;
    *((uint *) sp) = (uint) trapret;
    // ...
}
```

The last thing we need is to save some space for the process's context on the stack and point `p->context` to it. Then we'll zero it all out, except for the `eip` field, which will point to the beginning of `forkret()`. And that's it! We just return the pointer to the process now.

```
static struct proc *allocproc(void)
{
    // ...
found:
    // ...
    sp -= sizeof(*p->context);
    p->context = (struct context *) sp;

    memset(p->context, 0, sizeof(*p->context));
    p->context->eip = (uint) forkret;

    return p;
}
```

We can create new processes now!

growproc

What about growing or shrinking the size of a process's address space? We already did most of the hard work with `allocuvm()` and `deallocuvm()` from the post on process paging, so let's take a beat to thank past us for that.

Okay, so first we have to get the current process's size.

```
int growproc(int n)
{
    struct proc *curproc = myproc();
    uint sz = curproc->sz;
    // ...
}
```

Depending on the size of `n`, we'll either grow the process or shrink it by `n` bytes. Both `allocuvm()` and `deallocuvm()` can fail and return zero, so let's add some checks for those and return -1 if they fail.

```
int growproc(int n)
{
    // ...
    if (n > 0) {
        if ((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0) {
            return -1;
        }
    } else if (n < 0) {
        if ((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0) {
            return -1;
        }
    }

    curproc->sz = sz;
    // ...
}
```

Finally, we need to tell the hardware that there's a new page directory in town with a different size than the old one, so we'll use `switchuvm()` to update the page directory and TSS stored by the hardware to reflect the changes. Then we return 0 to indicate everything went okay.

```
int growproc(int n)
{
    // ...
    switchuvm(curproc);
    return 0;
}
```

procdump

This function is for debugging purposes: it'll print a complete listing of any processes in the process table. Quick spoiler: the keyboard interrupt handler function will set things up so that pressing `^P` runs this function. Go ahead, load up xv6 and try it out!

We want to print out the state for each process, but the states in `enum procstate` are just integers, which isn't very debug-friendly. So let's map them all to strings first with a static array of strings.

```
void procdump(void)
{
    static char *states[] = {
        [UNUSED]    "unused",
        [EMBRYO]    "embryo",
        [SLEEPING]  "sleep ",
        [RUNNABLE]  "runble",
        [RUNNING]   "run   ",
        [ZOMBIE]    "sombie",
    };
    // ...
}
```

This array notation might be a little unusual if you haven't seen it before: C lets you initialize arrays by specifying the value of each entry. If you leave any entries out, then they'll get initialized to zero. You can even write the entries out of order by adding their index before them in square brackets. So `{ [1] 5, [0] 2 }` is the same thing as `{2, 5}`. The `enum` turns the states into integers, so they work as indices here.

Now we'll just iterate over the process table to get all the processes, skipping over any `UNUSED` ones.

```
void procdump(void)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == UNUSED) {
            continue;
        }
        // ...
    }
}
```

Next we'll get the process's state (or just use `"???"` if something went wrong and the state isn't recognized).

```
void procdump(void)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        // ...
        char *state;
        if (p->state >= 0 && p->state < NELEM(states) && states[p->state]) {
            state = states[p->state];
        } else {
            state = "???";
        }
        // ...
    }
}
```

Then we can print out its PID, state, and name to the console.

```
void procdump(void)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        // ...
        cprintf("%d %s %s", p->pid, state, p->name);
        // ...
    }
}
```

Finally, we'll see later on that the `sleep()` and `wakeup()` system calls involve some lock trickery, so sleeping processes could be a common cause of concurrency issues like deadlocks. So if a process is sleeping, we'll print out its call stack using the `getcallerpcs()` function.

```
void procdump(void)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        // ...
        if (p->state == SLEEPING) {
            uint pc[10];
            getcallerpcs((uint *) p->context->ebp + 2, pc);

            for (int i = 0; i < 10 && pc[i] != 0; i++) {
                cprintf(" %p", pc[i]);
            }
        }
        cprintf("\n");
    }
}
```

Summary

Whew, we're making good progress. The most important part of this code was how xv6 creates new processes and sets them up to start running: basically, it uses some stack and function call trickery to make the scheduler start running a new process with the code in `forkret()`, then `trapret()`, before switching context into user mode.

We haven't talked about those two functions yet; we'll hold off on that until we do traps and system calls. Next up is scheduling processes!

Scheduling

We've done a lot of talking about context switching and scheduling, but we've procrastinated looking at the code for those. It's time to fix that.

There are all kinds of advanced schedulers out there, but as we've said before, the name of the game in xv6 is simplicity, so xv6 just uses a round-robin scheduling algorithm in which in loops through the exisitng processes in order. Each timer interrupt will force the current process to yield the processor and perform a context switch back into the scheduler so it can run the next available process.

swtch.S

The `struct context` we talked about in the last post is gonna be key here, so let's just look at its fields again:

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

The context switch function is `swtch()`; it's gonna need to save and restore processor registers, so that means it's gonna have to be written in assembly. But let's pretend it's just a C function for a second and talk about what it's going to do.

This function will save the contents of the registers on the stack as a `struct context`, then save that location as the old context. Then it'll load a new context, switch to the new stack, and restore the registers of the new context. Its declaration would look like this in C:

```
void swtch(struct context **old, struct context *new);
```

The first argument is a pointer *to a pointer* to a `struct context`. That double indirection might be confusing, but there's a method to this madness: C passes arguments by value, so if we used `struct context *old` and changed `old` to point to the saved context, it would be lost as soon as we returned from this function. So instead we have to use this kind of double pointer so we can set `*old` to point to the saved context. This way `old` will be lost anyway, but `*old` was changed and will persist beyond this function's return.

Note that, as we've said before, those arguments will be pushed on the stack before `swtch()` is called. So at the beginning of `swtch()`, the stack pointer `%esp` points to a return address; the argument `old` is one space (4 bytes) above that in the stack, and `new` is one space higher than that.

Okay, let's check out the assembly code now. We're gonna start by saving those arguments into registers. We can't just use any old registers here, or we might overwrite some of the data we're trying to save. But in the last post, I said x86 has a convention that the caller has to save the contents of the `%eax`, `%ecx`, and `%edx` registers, so that means we're free to overwrite them all we want since they've already been saved.

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

We haven't seen this number-parenthesis notation in assembly yet, so in case you're not familiar with x86 assembly, it's just a way to add a number to the contents of a register, then treating it as a pointer and dereferencing it. So `4(%esp)` in assembly is the same as `*(esp + 4)` in C. So at this point, `%eax` holds the `struct context **old` pointer, and

`%edx` holds the `struct context *new` pointer.

Now it's time to save all the fields in a `struct context` on the stack. The stack grows from high addresses to low ones, but C `structs` expect their fields to be from low to high, so we'll save them in reverse order. Oh, and hang on -- remember what's at the bottom of the stack right now, after the arguments? That's right, a return address. That's just a saved `%eip`, so that one's already done for us! We just need to save the others.

```
swtch:
    # ...
    pushl   %ebp
    pushl   %ebx
    pushl   %esi
    pushl   %edi
    # ...
```

Next we have to save a pointer to this old `struct context` into `*old`. Well, we pushed them on the stack in reverse order, right? So `%esp` already *is* pointing to it, so that's our pointer; we'll just copy it into `*old` (remember it's stored in `%eax`, and we dereference it in assembly with parentheses).

```
swtch:
    # ...
    movl    %esp, (%eax)
    # ...
```

Now it's time to switch stacks to the `new` context, which we saved in `%edx`. That context must have been saved by a previous call to `swtch()`, so it also happens to be a stack pointer as well.

```
swtch:
    # ...
    movl    %edx, %esp
    # ...
```

At this point, we're using the stack from `new`, which will already have its saved context at the top. So we can load the new context by popping it off the stack in reverse order into the corresponding registers. And again, just like the `call` instruction had already saved `%eip` on the stack as the return address, the `ret` (return) instruction will pop it off and restore it into `%eip` for us.

```
swtch:
    # ...
    popl    %edi
    popl    %esi
    popl    %ebx
    popl    %ebp
    ret
```

And that's it! That's a context switch in xv6.

proc.c

And now, finally, we can look at the scheduling code. Once the kernel is done setting itself up, initializing all the devices and drivers, etc., the very last function that `main()` calls is `scheduler()`. Interrupts were disabled in the boot loader and haven't been enabled yet, so it's also the scheduler's job to enable them for the first time in xv6.

`scheduler()` never returns; it's an infinite loop that just keeps searching through the process table for a `RUNNABLE` process, then runs it. So from that point on, with the exception of interrupts and system calls, the kernel will only ever do one thing: schedule processes to run.

scheduler

A CPU that's running the scheduler isn't running its own process. So we'll start off by setting this CPU's process pointer to null. Note that `mycpu()` requires interrupts to be disabled before it's called, but that's okay here because interrupts were disabled in the boot loader and haven't been re-enabled before the scheduler is called.

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

The order of the next few steps is tricky, and the authors of xv6 had to be extremely careful to do them in the right order to avoid concurrency problems. We need to (1) re-enable interrupts, (2) acquire the process table's lock, and (3) create an infinite loop to iterate over the process table forever, scheduling processes along the way. To see why this is nontrivial, let's check out some different orders (with a `fake_scheduler()` function) and see what problems we get.

ATTEMPT #1: interrupts -> lock -> loop. Let's try it out.

```
void fake_scheduler1(void)
{
    // ...
    sti();                  // enable interrupts
    acquire(&ptable.lock);  // acquire lock
    for (;;) {              // infinite scheduling loop
        // ...
    }
}
```

Interrupts have been disabled since the boot loader used `cli`, so when we call `sti()` here they'll be turned on for the first time in the kernel. At that point we'll find out if there were any interrupts waiting to be acknowledged, and possibly jump into some handler function to take care of it. Then when that's done, we'll come back here and acquire the process table's lock. Acquiring a lock disables interrupts, remember? So they're disabled again in the infinite scheduling loop (but not forever; we'll release the lock before switching to a user process). That sounds okay, right?

Not so fast! There's a hidden problem: suppose we had a situation in which none of the current processes are `RUNNABLE` -- maybe they're all blocked (or `SLEEPING`) waiting for I/O or something, which is not unlikely. In that case, the scheduler would just keep idly looping through the process table until one of them becomes `RUNNABLE` again. But if interrupts are always disabled in the loop, then this processor will never find out about, e.g., a disk interrupt saying it's done reading data which would allow a blocked process to become `RUNNABLE`. That means the process will never find out the condition it's waiting for has already happened, which means the scheduler will never find any `RUNNABLE` processes. It'll just get stuck in an infinite loop, repeatedly and desperately searching every entry of the process table. So basically, the system would freeze while the CPU pointlessly spins at top speed.

Okay okay, so that doesn't work. We'll have to periodically re-enable interrupts before disabling them again. So let's try moving the call to `sti()` inside the infinite loop so interrupts get re-enabled every once in a while.

ATTEMPT #2: lock -> loop -> interrupts.

```
void fake_scheduler2(void)
{
    // ...
    acquire(&ptable.lock);  // acquire lock
    for (;;) {              // infinite scheduling loop
        sti();              // temporarily enable interrupts
        // ...
    }
}
```

Problem solved, right? Actually... this one turns out to be just as bad. The call to `acquire()` disables interrupts, only for `sti()` to enable them again. There's a reason that locks disable interrupts, remember? If an interrupt occurs that switches away from `scheduler()`, then it might call a handler function that needs to access the process table lock,

which is already held by `scheduler()`, so that function would spin forever in a deadlock.

So now we arrive at the correct order: we'll call *both* `sti()` and `acquire()` inside the loop, in that order. That means we'll also need a call to `release()` at the end of the loop before we try to `acquire()` again in the next iteration. We had already said we'd have to release the lock before running a process; now we'll have to acquire it again before context-switching back into the loop.

ATTEMPT #3 (the right one): loop -> interrupts -> lock. This will give us a chance to detect any outstanding interrupts in each iteration of the for loop, but before we've acquired the lock again and thus, before doing so could cause a deadlock.

```
void scheduler(void)
{
    // ...
    for (;;) {
        sti();
        acquire(&ptable.lock);
        // ... pick a process and run it ...
        release(&ptable.lock);
    }
}
```

Whew, okay. Basically, we've learned that concurrency bugs can be hard to predict and can turn seemingly-fine code into impossible-to-diagnose system crashes or freezes.

Okay, so now let's fill in the part of the loop where the scheduling algorithm goes. We'll add an inner for loop to iterate over the process table entries and stop when we find a `RUNNABLE` process.

```
void scheduler(void)
{
    // ...
    for (;;) {
        // ...
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->state != RUNNABLE) {
                continue;
            }
            // ... run that process ...
        }
        // ...
    }
}
```

Next, if we found a process, then we need to switch to that process's virtual address space; that is, we need to start using its page directory.

```
void scheduler(void)
{
    // ...
    for (;;) {
        // ...
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            // ...
            c->proc = p;
            switchuvm(p);
            // ...
        }
        // ...
    }
}
```

Now, if we just switched to an arbitrary page directory in the middle of running other code, we might cause a bunch of problems: all the virtual addresses we're currently using for variables, functions, instructions, etc. might suddenly become invalid and point to random other places in memory. But this is where can see some of the earlier design decisions in xv6 start to pay off: remember how `setupkvm()` made sure every single process would have the exact same mappings for the upper half of the address space, starting at `KERNBASE`? That means that if we're running in kernel mode, we can arbitrarily switch to any process's page directory and know that all of our mappings will be exactly the same. The user mappings in the lower half might be different, but the kernel side will never change. Nice!

Now we can run the process using `swtch()`.

```
void scheduler(void)
{
    // ...
    for (;;) {
        // ...
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            // ...
            p->state = RUNNING;
            swtch(&(c->scheduler), p->context);
            // ...
        }
        // ...
    }
}
```

`swtch()` will *not* return here immediately; instead, it'll pick up execution wherever the process last left off, which will be in kernel mode -- if it stopped running before, it must have been due to a system call, interrupt, or exception, which would have been handled in kernel mode before calling the scheduler.

Note that this process will still be holding the process table lock when it starts running again. For example, that's the main reason for the existence of the `forkret()` function we mentioned before. This is another dangerous detail we'll have to remember, so I'm just gonna go ahead and hope you remember THIS BIG GIANT GLARING WARNING FLAG RIGHT HERE: if you do any xv6 kernel hacking, and you want to add a new system call that will let go of the CPU, then your code *must* release the process table lock at the point at which it starts executing after switching to it from the scheduler.

This is pretty dangerous; if xv6 were a big project, it would be really easy to forget that when adding more features later on. But in this case, there's no easy way to get around it; for example, we can't just release the process table lock before calling `swtch()` and reacquire it after. The problem becomes apparent if you think of locks as protecting some invariant; that invariant might be temporarily violated while you hold the lock, but it should be restored before the lock is released.

The process table protects invariants related to the process's `p->state` and `p->context` fields, e.g. that the CPU registers must hold the process's register values, that a `RUNNABLE` process must be able to be run by any idle CPU's scheduler, etc. These don't hold true while executing in `swtch()`, so we need to hold the lock then; otherwise another CPU might decide to run the process before `swtch()` is done executing.

Now, at some point that process will be done running and will give up the CPU again. Before it switches back into the scheduler, it has to acquire the process table lock again. So here's ONE MORE GIANT WARNING for good measure: you should make sure to do that too if you add your own scheduling-related system call.

Eventually, it'll switch back here with a call with the arguments in reverse, like `swtch(&(p->context), c->scheduler)`. At the point, execution of the scheduler will resume right here, so we need to switch back to using the kernel page directory `kpgdir`.

```
void scheduler(void)
{
    // ...
    for (;;) {
        // ...
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            // ...
            switchkvm();
            c->proc = 0;
        }
        // ...
    }
}
```

After that, the outer for loop just releases the lock before looping to the top again to temporarily re-enable interrupts, then acquire the lock again and check for another process to run.

forkret

Let's take a quick look at one example of where a process might start to execute after being scheduled. All processes (whether the very first process, or any others created later through calls to `fork()`) will start running code in `forkret()`, then return from here into `trapret()`.

Most of the time, this function does just one thing: it releases the process table lock. However, there are two kernel initialization functions that have to be run from user mode, so we can't just call them from `main()` and be done with it. We need a place for a process to call them, and `forkret()` is as good a place as any. So the very first call to `forkret()` will run these two start-up functions, and the rest will ignore them.

The two functions are `iinit()` and `initlog()`, which are part of xv6's file system code; we'll get to them later on. For now, we'll just use a `static int` as a boolean and set it to false after we've run those functions once on our first pass through `forkret()`.

```
void forkret(void)
{
    static int first = 1;

    release(&ptable.lock);

    // Only gets run once, on the first call
    if (first) {
        first = 0;
        iinit(ROOTDEV);
        initlog(ROOTDEV);
    }
    // Returns into `trapret()`
}
```

Any other kernel code that switches into the scheduler (e.g., `sleep()` and `yield()`) will have a similar lock release right after returning from the scheduler.

sched

We saw one example of code that runs after switching *away* from the scheduler, but what about code that runs before switching *to* the scheduler? Any functions that need to call into the scheduler can't just call `scheduler()`, since the scheduler probably left off last time halfway through the loop and should resume in the same place. So `sched()` handles the task of picking up the scheduler wherever it last left off.

`sched()` should be called *after* acquiring the process table lock and without holding any other locks (lest we cause a deadlock somewhere). Also, the process should not be in the `RUNNING` state anymore since we're about to stop running it. So we'll start off by checking that those are all true and that interrupts are disabled.

```
void sched(void)
{
    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");
    }
    // ...
}
```

Next, remember when the `pushcli()` and `popcli()` functions checked whether interrupts were enabled before turning them off while holding a lock? That's really a property of this kernel thread, not of this CPU, so we need to save that now. Then we can call `swtch()` to pick up where the scheduler left off (the line right after its own call to `swtch()`). This process will resume executing after that line eventually, at which point we'll restore the data about whether interrupts were enabled and let it run again.

```
void sched(void)
{
    // ...

    // Save whether interrupts were enabled before acquiring the lock
    int intena = mycpu()->intena;

    // Perform context switch into the scheduler
    swtch(&p->context, mycpu()->scheduler);
    // Execution will eventually resume here

    // Restore whether interrupts were enabled before
    mycpu()->intena = intena;
}
```

yield

Okay, let's see an example of how this all comes together now! The `yield()` function forces a process to give up the CPU for one scheduling round. For example, this will be used to handle timer interrupts later on. Now that we know how scheduling works in xv6, `yield()` is easy. We just acquire the process table lock, set the current process's state to `RUNNABLE` so it can get picked up again in the next scheduling round, and call `sched()` to switch into the scheduler. When we eventually return here, we'll just release the lock again.

```
void yield(void)
{
    acquire(&ptable.lock);

    myproc()->state = RUNNABLE;
    sched();

    release(&ptable.lock);
}
```

Summary

We've now seen how xv6 handles process scheduling with a super-simple round- robin algorithm. The `scheduler()` function had plenty of concurrency pitfalls, but luckily the xv6 authors took care of all the careful coding for us, so we just get to sit back and admire their work.

We also saw how context switches occur in xv6, so now we can understand how, in the previous post, `allocproc()` set up a new process with a context that would result in it starting execution in `forkret()`.

Next up, we'll look at the way xv6 handles interrupts, system calls, and software exceptions.

Sleep Locks

We've used plenty of spin-locks, and a previous post looked at their implementation in xv6. Spin-locks have pretty harsh performance costs: a process that's waiting to acquire a lock will just spin idly in a while loop, wasting valuable CPU time that could be used to run other processes. So far, we've only seen locks for kernel resources like the process table, page allocator, and console, for which all operations should be relatively fast, on the order of a few dozen CPU cycles at most.

Now it's time to look at the disk driver and file system implementation, and we'll need some locks there too. But disk operations are *slow* -- reading from and writing to disk might take milliseconds, which is a literal eternity for a CPU. Imagine a process hogging a spin-lock for the disk while other processes spin around and around waiting *forever* for the disk to finish writing. It would be an enormous waste!

Spin-locks were the best we could do at the time, since we didn't have any infrastructure to support more complex locks, but now we really do need a better alternative. We also have some more kernel building blocks in place relating to processes, including a bunch of system calls.

For example, we've seen the `sleep()` and `wakeup()` system calls, which let a process give up the CPU until some condition is met. Well, hang on a second -- what if that condition is that a lock is free to acquire? Then a process could sleep while another process holds the lock, and wake up when it's ready to be acquired; that would let other processes run instead of forcing a process to spin and spin. xv6 calls these *sleep-locks*, and it's time to find out how they work.

sleeplock.h

If we want a process holding a sleep-lock to give up the processor in the middle of a critical section, then sleep-locks have to work well when held across context switches. They also have to leave interrupts enabled. This couldn't happen with spin-locks: it was important that they disable interrupts to prevent deadlocks and ensure a kernel thread can't get rescheduled in the middle of updating some important data structure.

Leaving interrupts on adds some extra challenges. First, we have to make sure the lock can still be acquired atomically; second, we have to make sure that any operations in the critical section can safely resume after being interrupted.

Let's solve the first problem: how can we make sure a sleep-lock will always be acquired atomically? Well, if we want to do something atomically, we already have a solution: spin-locks! So rather than reinventing the wheel, we'll just make each sleep-lock a two-tiered deal with a spin-lock to protect its acquisition.

We'll use a `locked` field just like the one all spin-locks have, but then we'll add a spin-lock to protect it. We'll also make debugging a little easier by adding a name for the lock and a field for a PID to identify which process is holding it.

```
struct sleeplock {
    uint locked;
    struct spinlock lk;
    char *name;
    int pid;
};
```

sleeplock.c

initsleeplock

We can initialize a sleep-lock by initializing its guard spin-lock, then adding a name for it, setting `locked` to false, and the `pid` field to zero.

```
void initsleeplock(struct sleeplock *lk, char *name)
{
    initlock(&lk->lk, "sleep lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
}
```

acquiresleep

In order to make sure sleep-lock acquisition is atomic, we'll bookend this function by acquiring and releasing a spin-lock. This will also make sure that interrupts are disabled during this function but re-enabled when it's done. It does add some overheard in the form of spinning until this lock is free, but the code here should be relatively short and fast to execute. What we really want is to avoid spinning once the sleep-lock is acquired, i.e. spinning *after* this function is done. So we'll tolerate a little waste here.

```
void acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    // ...
    release(&lk->lk);
}
```

Okay, now we have to do the actual acquisition. We said above that we'd use the `sleep()` function to avoid wasting processor time. Hopefully you remember one important detail about `sleep()`: it must always be called inside a while loop in order to make sure that we don't miss any wakeup calls. So let's check if the sleep-lock is already being held and go to sleep if it is. We'll need a channel and a lock for `sleep()` to release, so let's use the pointer to this lock `lk` as the channel, and the outer spin-lock `lk->lk` as the lock to be released.

```
void acquiresleep(struct sleeplock *lk)
{
    // ...
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    // ...
}
```

It's important to keep the two locks separate in your head right now: `lk` is the sleep-lock, and `lk->lk` is the spin-lock it uses to protect the sleep-lock's acquisition. Note that we're checking `lk->locked` here, *not* the spin-lock `lk->lk` -- this process is already holding `lk->lk`, but we need to acquire `lk` itself by updating `lk->locked`. Phew, try saying that ten

times fast.

Now the process will go to sleep and yield the CPU until the sleep-lock is free. If multiple processes are sleeping waiting on the same sleep-lock, they will all wake up at the same time, but all of them have to reacquire `lk->lk` before returning from sleep, so only one will get to return here and complete the sleep-lock acquisition. The others will spin a bit longer, then return here only to find that `lk->locked` is already being held by another process, so the while loop will put them to sleep again.

Once the sleep-lock is free, the process can exit the while loop and claim the sleep-lock for itself.

```
void acquiresleep(struct sleeplock *lk)
{
    // ...
    lk->locked = 1;
    lk->pid = myproc()->pid;
    // ...
}
```

We don't need fancy atomic operations like `xchg` anymore, since the guarding spin-lock has already made sure that interrupts are disabled and all operations are effectively atomic. So that's all we need! Now we just release the spin-lock and return.

releasesleep

Now that we've seen how a process acquires a sleep-lock, releasing it is easy, we just do the opposite. We'll set `lk->locked` to zero and clear the `lk->pid` field. And what's the opposite of `sleep()`? Well, `wakeup()`, of course! That will check whether there are any processes sleeping on this channel and let them know they can attempt to acquire the sleep-lock now.

```
void releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    lk->locked = 0;
    lk->pid = 0;

    wakeup(lk);

    release(&lk->lk);
}
```

holdingsleep

This function is even more simple: it just checks whether a sleep-lock is being held, and if so, whether it's being held by the current process. The first is done by just checking `lk->locked`; the second is done by checking that `lk->pid` matches the current process's PID. The result is a boolean stored in a temporary variable so we can release the guarding spin-lock before returning the result.

```
int holdingsleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    int r = lk->locked && (lk->pid == myproc()->pid);

    release(&lk->lk);
    return r;
}
```

Summary

Okay, that wasn't too bad! It makes sense why we couldn't use sleep-locks in a kernel without system calls like `sleep()` and `wakeup()`. But xv6 already has those, so why not use them everywhere? If sleep-locks really do cut down on the wasted CPU time, can we just go back and replace all the spin-locks with sleep-locks? Then the only use for spin-locks would be as a guard for the more- sophisticated sleep-locks.

Hold your horses! It's not that easy. Sleep-locks leave interrupts enabled, so they can't be used in interrupt handler functions, or inside a critical section where a spin-lock is being used, since interrupts will be disabled (though spin- locks can be used inside sleep-lock critical sections). They also can't be used by kernel threads like the scheduler, since those aren't processes and thus can't be put to sleep.

Finally, there are some situations in which a sleep-lock might actually add *more* overhead than a spin-lock: it takes some time to put a process to sleep, schedule another process, send a wakeup call, schedule the first process again, and so on, and the process will hold the sleep-lock the entire time. If another process is waiting on the sleep-lock, it might actually end up waiting longer than with a spin-lock, although it'll wait in a sleeping state instead of a running state where it just spins in a loop.

Additionally, sleep-locks can only be used when it's safe to interrupt a process in the middle of a critical section and wake it up later. Sure, no other process can acquire the sleep-lock in the meantime, but it's still not great for time- sensitive operations like getting the current number of `ticks`.

So sleep-locks are great, but their applications are more limited than spin- locks. The perfect use for them is when a process needs to complete an operation atomically, but that operation itself might take a very long time. A great example of that is disk I/O, and we'll see next how xv6 puts them to use in its file system implementation.

Detour: Spin-Locks

So I know I said I wasn't expecting you to have finished the OSTEP section on concurrency, but xv6 uses locks all over the place, so we're gonna have to get comfortable with them right away. Luckily, xv6 primarily uses spin-locks, which are super simple and work on bare metal; a lot of the more complex/more awesome locks that OSTEP talks about require an OS beneath them.

I'll give a brief intro to concurrency first in case you haven't made it to that part in OSTEP; then we'll turn to the spin-lock implementation in xv6.

A Very Brief, Poor-Man's Intro to Concurrency

TL;DR: Concurrency is your worst nightmare. It'll cause bugs in the places where you least expect it, and they won't even be consistent: your code might work 95% of the time, but every once in a while it'll randomly fail and you'll have no idea why. The good news: xv6 handles it in a super-simple way, so we'll get to appreciate it as we go along. If you're like me, you might also see the code use locks when you wouldn't have thought they were needed, and then you'll come to appreciate just how clever the xv6 authors are.

First off, stop reading this and go watch the discussion of data races and locks in the last few minutes of the CS 50 2021 lecture on SQL (https://www.youtube.com/watch?v=LzElj46saa8&t=8762s). I'm serious, go watch it right now; this post will still be here.

Okay, I'm gonna assume you've seen it now; you should have a decent sense of the main issues with data races and how locks solve them. But the CS 50 lecture skipped some details about locks: (1) what Brian (the TA) does when he finds a locked fridge, (2) how locks are implemented in code, and (3) deadlocks.

What Does Brian Do?

Let's say process `david` is running on one thread, and it needs to use some resource (a global variable maybe, or an I/O device like the disk or console) that other threads might want to use too, so `david` acquires the lock for that resource. Then process `brian` comes along and wants to use the same resource at the same time. This could cause a data race, but luckily we've thought ahead and used a lock, so `brian` can't access it until `david` is done with it and releases the lock.

First of all, we better hope `david` remembers to release the lock; otherwise `brian` (and all other processes, even the kernel) will *never* be able to use that resource. But assuming we're smart and remembered to release it, what does the `brian` process do in the meantime?

Well, maybe `brian` has some other work to do that he can get started on in the meantime. But what would that mean for an OS? How would we know, in general, whether the lines of code that follow the use of a shared resource can be safely executed if we haven't used that resource yet? That sounds impossible to figure out without knowing ahead of time what the resource is and how it's used, so let's just go ahead and skip that idea.

Another option that's actually used often in the real world is for `brian` to stop trying and go to sleep. Maybe he can put a note on himself asking `david` to wake him up when he gets back with the milk. So in code, that might look like `brian` signaling the OS and letting it run a different process until the lock is released. That sounds nice and all, but at this early stage in our kernel, we don't even have processes or a scheduler yet, let alone a notion of sleeping.

Okay, another option: what if `brian` just spins around in circles, or twiddles his thumbs, or does jumping jacks or whatever until `david` releases the lock? In code, that means looping over and over forever until the lock is released. That would be horribly inefficient; think of all the CPU time wasted when one process just loops over and over again while another process does something slow while holding a lock! But it's also the approach that xv6 is gonna take, because at the end of the day, our kernel is still in baby stages and beggars can't be choosers. So xv6 uses *spin-locks* with loops that only stop when we acquire a lock.

This means we should be careful when using locks to acquire them only at the last possible moment when they're absolutely needed, and release them as soon as they're no longer required, in order to limit the amount of wasted CPU cycles.

Implementing Locks

We can implement locks as a simple boolean variable: if it's true, then someone else is using the resource behind the lock. If it's false, then it's unused and you can go ahead and take it. So an `acquire()` function sets the lock to `true` and a `release()` function sets it back to `false`. Done!

But it's not so simple: there's actually a race condition hidden in the very idea of a lock. Think about it for a second: a lock protects some shared resource, right? And a shared resource is something that more than one process wants to use? But a lock is itself a thing that more than one process wants to use... so we haven't actually gotten rid of the race condition. (FLIPS TABLE.)

We have another Catch-22 on our hands, but this time we can't get rid of it with a clever software trick like we did with the `entrypgdir`. The issue is that no matter how well we write our code, it will always require more than one step: first we have to check whether the lock is `true`, then we have to set it to `true`. But if someone else is doing the same thing at the same time, our instructions might get executed in parallel and then we'd both acquire the lock at the same time -> RACE CONDITION.

The solution will require hardware support, using *atomic* instructions -- these are hardware instructions that are indivisible; no other code can execute in between ours. One example is the x86 instruction `xchg`, which atomically reads a value from memory, updates it to a new value, and returns the old value.

Now we're good! A lock can still be a boolean variable but now `acquire` has to use `xchg`: it should get the old value while simultaneously updating it to `true`.

Atomic instructions have more overhead than regular ones, so we should only use them when they're required, like in locks, but otherwise we can stick to the regular instructions we've always used.

There's one other detail we should be careful about: a lot of the locks in xv6 protect resources that are needed by both interrupt handlers and kernel or user code. For example, we might use a process table lock to protect the list of all currently running processes; suppose some kernel code has acquired the lock in order to run a new process. What

happens if a timer interrupt goes off at that moment? The timer interrupt handler function might need to acquire the lock in order to switch processes, but it's already being held by the kernel thread. But the timer interrupt might take priority over the kernel thread and refuse to return to the kernel until it finishes executing. The result: that CPU comes to a total halt as the timer interrupt handler function spins forever, never to get the lock it so desperately needs to move on. So sad. :(

xv6 avoids this issue in a really simple way: every time we acquire a lock, we'll just disable interrupts altogether. Problem solved: now a thread can't get interrupted until it's done using the lock and releases it. This does mean that a process which grabs locks often might stick around longer than it should, since we won't have timer interrupts to tell the scheduler to swap it out with another process, but we're just gonna cross our fingers and hope that doesn't happen too often.

Deadlocks

The last concurrency issue we need to be aware of is the problem of deadlocks. Suppose two threads each need locks A and B; this happens often, e.g. when loading a user program the kernel will need to hold a lock for the disk and another for the process table, or a process might be reading from disk and printing to the console at the same time.

Suppose they're running at the same time, and one process acquires lock A while the other one acquires lock B. If they each need the other lock to keep going, they'd spin forever waiting for it. This is a deadlock.

The way to avoid these is to make sure that, if we use more than one lock, we *always* acquire them in the same order. That way, one process would acquire lock A, the second one would be unable to acquire it and would spin, then the first process acquires lock B with no issues. When it's done, it releases both locks and the second process can continue.

This can get complicated though: if we ever acquire a lock in a function, we'd have to check any functions that that function calls to see whether they use any locks, and so on. If they do, and if the order conflicts with another chain of function calls, we'd have to refactor the code until the orders match. xv6 has been carefully written so that the lock acquisition order is always consistent.

spinlock.c

xv6's spin-locks are set up as a `struct spinlock`, defined in spinlock.h (https://github.com/mit-pdos/xv6-public/blob/master/spinlock.h). The `locked` field acts as the boolean variable to determine whether the lock is held; the other fields are for debugging, since we can expect concurrency issues to be the one of the most common causes of bugs in the kernel code because, again, concurrency is your worst nightmare.

Note that `locked` is an `unsigned int` instead of a `bool`; C requires the standard library header *stdbool.h* in order to use the `bool` type, but on bare metal we can't assume we have a standard library to use.

initlock

```
void initlock(struct spinlock *lk, char *name)
{
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}
```

This function is pretty straightforward; it just stores the string `name` in the lock and starts it off as unlocked; the `cpu` field is 0 because no CPU is holding it yet. Next.

pushcli and popcli

For reasons mentioned above, we need to disable interrupts whenever we're using a lock and re-enable them when we release a lock. But if we're not careful, we could end up enabling interrupts too early when we release one lock while still holding another; or if interrupts were already disabled when we acquired a lock, we could unintentionally re-enable them upon releasing it.

xv6 uses paired functions `pushcli()` and `popcli()`.

```
void pushcli(void)
{
    int eflags = readeflags();
    cli();
    if (mycpu()->ncli == 0) {
        mycpu()->intena = eflags & FL_IF;
    }
    mycpu()->ncli += 1;
}
```

`readeflags()` is a C wrapper for some x86 assembly code that reads from the `eflags` register; the 9th bit is the interrupt flag, which is set whenever interrupts are enabled. `cli` is another x86 instruction that clears that flag, thus disabling interrupts.

`mycpu()` returns a pointer to a `struct cpu` with information about the CPU running this code; we'll go over these when we talk about processes; here we increment the `ncli` field in every call to `pushcli()`. If this is the first call, we save the value of the interrupt flag in the `intena` field.

```
void popcli(void)
{
    if (readeflags() & FL_IF) {
        panic("popcli - interruptible");
    }
    if (--mycpu()->ncli < 0) {
        panic("popcli");
    }
    if (mycpu()->ncli == 0 && mycpu()->intena) {
        sti();
    }
}
```

`popcli()` first checks to make sure interrupts aren't already enabled and we're not popping without having pushed. Then it decrements the `ncli` field of the `struct cpu` for this CPU. If this is the last call to `popcli()`, it checks the `intena` field; if it was set (i.e., interrupts were enabled before the first `popcli()`), then it enables interrupts again.

Check out how these two functions are carefully written so that they're matched: it takes two calls to `popcli()` to undo two calls to `pushcli()`. Also, if interrupts were already off before the first call to `pushcli()`, they'll stay off after the last `popcli()`. Pretty neat, right?

holding

This function checks whether this CPU is holding the lock.

```
int holding(struct spinlock *lock)
{
    pushcli();
    int r = lock->locked && lock->cpu == mycpu();
    popcli();
    return r;
}
```

Not much to talk about here; it just checks (inside calls to `pushcli()` and `popcli()`) whether the lock is being held and this is the CPU holding it. If both conditions are true it'll return 1; otherwise 0.

acquire

The first step in this function is to disable interrupts to avoid deadlocks. We also make sure we're not already holding the lock; otherwise we'd deadlock ourselves.

```
void acquire(struct spinlock *lk)
{
    pushcli();

    if (holding(lk)) {
        panic("acquire");
    }

    // ....
}
```

Next up, we've gotta acquire the lock using the atomic `xchg` instruction, defined in x86.h (https://github.com/mit-pdos/xv6-public/blob/master/x86.h). Like we said before, the trick is to atomically set `locked` to 1 while returning the old value. If the returned old value is 1, that means it was already 1 before we got to it, so it's currently being held and we can't acquire it yet -- gotta spin. But if the returned old value is 0, that means the lock was free before we got to it, and our `xchg` just updated it to 1, so we've successfully acquired it. No other instruction can occur between checking the old value and updating it to the new one, so we can be confident that no one else will be holding the lock at the same time.

```
void acquire(struct spinlock *lk)
{
    // ...
    while (xchg(&lk->locked, 1) != 0)
        ;
    // ...
}
```

We do have to be careful about one other thing: compiler optimizations can get pretty wild nowadays, so the order of code on the page isn't necessarily the order it'll get compiled to or executed in. This is a critical section of code, so we need to make sure acquiring the lock forms a barrier between the code that comes before it and the code after it so any reordering doesn't cross the lock acquisition point. We can do that with a special compiler instruction:

```
void acquire(struct spinlock *lk)
{
    // ...
    __sync_synchronize();
    // ...
}
```

Finally, we'll record some info about the CPU and process holding the lock for debugging purposes. Don't worry about `mycpu()` for now, but we'll talk about `getcallerpcs()` below.

```
void acquire(struct spinlock *lk)
{
    // ...
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

release

Releasing a lock is a little easier than acquiring it: to acquire it, we need to check whether it's already held and update its value, with both steps together as an atomic instruction. To release it, we only have to set the value to false. That's only one instruction, so it's automatically atomic!

Well, almost, but not quite. The compiler works some serious magic behind the scenes, so there's no guarantee that a single C operation like `lk->locked = 0` will actually get compiled down to a single assembly instruction. So we're gonna have to make sure it does by writing it directly in assembly.

We start off by making sure we are already holding the lock before releasing a lock held by someone else. Then we clear the debug info stored in the lock, and tell the compiler and processor not to reorder code past the lock release.

```
void release(struct spinlock *lk)
{
    if (!holding(lk)) {
        panic("release");
    }

    lk->pcs[0] = 0;
    lk->cpu = 0;

    __sync_synchronize();

    // ...
}
```

Next we need to release the lock, i.e. an assembly instruction equivalent to `lk->locked = 0` in C. C allows in-line assembly code using the `asm` keyword. We mark it as `volatile`, which prevents the compiler from optimizing the write away and ensures it'll get written to memory. Finally, we call `popcli()` to enable interrupts again.

```
void release(struct spinlock *lk)
{
    // ...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

getcallerpcs

This function exists to store information about the current process in the lock for use in debugging. In particular, we want to record the program counters of the last 10 functions on the call stack so we can try to figure out which functions were called in which order when concurrency issues inevitably bring our world crashing down with data races, or to a grinding halt with deadlocks.

In order to get the program counters, we're gonna have to know a bit about how x86 handles function calls. The `%eip` register (or instruction pointer) holds the program counter, which tracks the next instruction to be executed. The `%ebp` register (or base pointer) holds the address of the base of the stack (i.e., its highest address, since it grows down).

When a function gets called all its arguments are pushed on the stack in reverse order, so that the first argument is at the top (lowest address) of the stack. Then the previous function's `%eip` is pushed on the stack, followed by its `%ebp`:

```
<- low addresses                                                  high addresses ->
...  [new function's data]  [old %ebp]  [old %eip]  [new arg1]  [new arg2]  ...
<- top of stack                                                 bottom of stack ->
```

Anyway, the point is that if we have the address of the first argument to the current function, then we can recover the contents of the previous function's `%ebp` and `%eip` registers: `%eip` is one spot below it on the stack and `%ebp` is two spots below it.

```
void getcallerpcs(void *v, uint pcs[])
{
    uint *ebp = (uint *) v - 2;
    // ...
}
```

Note the type casts here -- `v` is a pointer to the first argument, which can be of any type and size, so we use a `void *`. But both of the `%eip` and `%ebp` registers hold 32-bit pointers, so `ebp` is declared as a pointer to a `uint` (a type alias for `unsigned int`, remember?), which makes the pointer arithmetic work out nicely so that subtracting 2 returns a pointer to the right spot on the stack.

Now, what we really want is the program counter `%eip`, not the pointer to the stack base `%ebp`. But we can use the address of `%ebp` to make sure we haven't gone too far back in the function call history. Remember, we wanna get the program counters for the last 10 functions in the call stack, then save them in the `pcs` array.

```
void getcallerpcs(void *v, uint pcs[])
{
    // ...
    int i;
    for (i = 0; i < 10; i++) {
        // Stop if the %ebp pointer is null or out of range
        if (ebp == 0 || ebp < (uint *) KERNBASE || ebp == (uint *) 0xffffffff) {
            break;
        }
        pcs[i] = ebp[1];
        ebp = (uint *) ebp[0];
    }
    // ...
}
```

Let's talk about those last two lines: the `ebp` pointer in the code holds the location of the saved `%ebp` register, so `ebp[0]` is the value at that address (i.e., the actual value of the saved `%ebp` register) and `ebp[1]` is the value stored one spot above that, i.e. the value of the saved `%eip` register. So each iteration of the loop will get one `%eip` and store it in a `pcs` entry.

Then we update `ebp` to the actual value at the address it points to, which means `ebp` will now point to the address of the saved `%ebp` register for the function one step further back in the call chain. Okay sorry, I know that's confusing, but basically each iteration of the for loop moves us back to the function that called this function, then the function that called that one, and so on.

Okay, whew. So what happens if we break out of the for loop early because we went all the way back in the call stack? The other entries of `pcs` might hold some garbage values, so let's just make them null pointers so we know to ignore them when debugging.

```
void getcallerpcs(void *v, uint pcs[])
{
    // ...
    for (; i < 10; i++) {
        pcs[i] = 0;
    }
}
```

One last little trick: the previous for loop declared the loop variable `i` before the loop -- this means `i` will be in scope for the rest of the function body. If it had been declare inside the for loop like `for (int i = 0; ...)`, it would fall out of scope at the end of the loop. So we can keep using the same `i` in this second for loop (without an initialization statement) and know it'll hold the value it had after finishing the first for loop. If we finished all the iterations, that value will be 10; otherwise it'll be less. So we use that to clear any remaining entries of `pcs`.

Summary

You'll learn to hate concurrency issues in C; newer languages like Rust make data races a thing of the past, though deadlocks can still rear their ugly heads. But for now, the xv6 authors have done all the dirty work for us, so we can just sit back and watch. Note, though, that even the xv6 authors say it's totally possible that something has slipped past them and the thousands of other students and instructors that have looked at xv6, so it's probable that xv6 still has some lingering race conditions. See, even the masters struggle with it. -_-

Anyway, we saw that locks have to be implemented with hardware support using atomic instructions. C and most languages provide high-level atomics that real- world operating systems use, but the point of xv6 is elegance in simplicity, not being a total show-off, so the xv6 spin-locks just use the basic `xchg`.

We took this detour into spin-locks to make sure we all understand some basic details because we're gonna be seeing a lot of them in the rest of the kernel code. They're inefficient (because the processor just spins around waiting for the lock to be released, WHEEEEE), but we gotta make do with the machinery we've built up so far. xv6 will also use some fancier locks called sleep-locks, but we'll cross that bridge when we get to it.

System Calls: Processes

In a previous post, I pointed out some of the most important functions a kernel has to fulfill. System calls take care of two of these: virtualizing resources via virtual memory and processes, and mediating communication between user-mode processes and the hardware. We'll wrap up the former now by looking at the system call functions relating to processes and scheduling.

proc.c

fork

Unlike some of the other functions we'll talk about in this post, `fork()` is used almost exclusively by user code as a system call; the kernel never calls it. That said, it has an extremely important role: after the first process has started, it's the only way to create more processes. It does that by copying the parent process's virtual address space into a new page directory. We haven't talked about the file system yet, but hopefully you're familiar with file I/O in Linux, so you know each process has its own list of open files and a current working directory; `fork()` will clone those as well for the child process.

Let's start off by getting a pointer to the parent process and creating a slot in the process table for the child process with `allocproc()`. Remember, that function returns a pointer to the new process's `struct proc`, but it can fail and return null (e.g., if there is no available slot in the process table, or if its call to `kalloc()` fails), so we'll need to check for that.

```
int fork(void)
{
    // Parent process
    struct proc *curproc = myproc();

    // Allocate process table slot for child process
    struct proc *np;
    if ((np = allocproc()) == 0) {
        return -1;
    }
    // ...
}
```

`allocproc()` also sets up the new process's stack so that it'll return into `forkret()`, then `trapret()`, before context switching into user mode, and sets the process's state to `EMBRYO`.

Next we need a page directory for the new child process; it should be a copy of the parent process's page directory. Luckily, we already did the hard work for this back in the virtual memory posts, so we can just use `copyuvm()` now. That function can also fail, in which case we'll free the stack that `allocproc()` created and set the child process's state back to `UNUSED`.

```
int fork(void)
{
    // ...
    if ((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0) {
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    // ...
}
```

Next we'll copy the parent process's size and trap frame; the latter will make sure the child starts executing after `trapret()` with the same register contents as the parent. We'll set the child process's parent to, well, its parent (the current process).

```
int fork(void)
{
    // ...
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;
    // ...
}
```

The two processes will be nearly identical, so we need a way to distiguish them from user space so that a user program can give different instructions to each. xv6 follows the Unix convention that `fork()` should return the child process's PID to the parent and return 0 for the child. The parent's return value is easy; we'll just literally return the child's PID at the end. But the child didn't actually call `fork()`, so how can we set a return value that it will see?

Well, the x86 convention is for return values to be passed in the `%eax` register, right? And that register will be restored from the trap frame before switching into user mode. So we'll just store the value 0 there.

```
int fork(void)
{
    // ...
    np->tf->eax = 0;
    // ...
}
```

Next we'll copy all the parent process's open files and its current working current working directory. The files are stored in a per-process file array `curproc->ofile` of size `NOFILE`, so we can copy them over with the function `filedup()` (which we'll see later). The current working directory is in `curproc->cwd` and can be copied with `idup()`.

```
int fork(void)
{
    // ...
    for (int i = 0; i < NOFILE; i++) {
        if curproc->ofile[i]) {
            np->ofile[i] = filedup(curproc->ofile[i]);
        }
    }
    np->cwd = idup(curproc->cwd);
    // ...
}
```

Then we'll copy the parent process's name with `safestrcpy()`, defined in string.c (https://github.com/mit-pdos/xv6-public/blob/master/string.c). You might be familiar with the C standard library funtion `strncpy()`; this function is almost identical, except that unlike `strncpy()` it's guaranteed to nul- terminate the string it copies. If you haven't seen this kind of thing before, it's a fairly common practice to write your own safe wrappers for some of the C standard library functions, especially the ones in `string.h` which are so often error-prone and dangerous.

```
int fork(void)
{
    // ...
    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
    // ...
}
```

Finally, we'll set the child process's state to `RUNNABLE` and return its PID for the parent.

```
int fork(void)
{
    // ...
    int pid = np->pid;

    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);

    return pid;
}
```

kill

This is one of the functions that can get called both by the kernel and as a system call. The kernel will use it to terminate malicious or buggy processes, and user code can use it as a system call to kill another process too.

We said before that killing a process immediately would present all kinds of risks (e.g. corrupting any kernel data structures it might be updating, etc.), so all we're gonna do is give it the ominous mark of death with the `p->killed` field. Then the code in `trap()` will handle the actual murder the next time the process passes through there.

The argument is a process ID number, so let's just iterate over the process table until we find a process with a matching PID; we'll return -1 if we don't find any.

```
int kill(int pid)
{
    acquire(&ptable.lock);

    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            // ...
        }
    }

    release(&ptable.lock);
    return -1;
}
```

If we do find a matching process, then we'll set `p->killed`. Also, some of the calls to `sleep()` will occur inside a while loop that checks if `p->killed` has been set since the process started sleeping, so let's hasten the process's death a little by setting its state to `RUNNABLE` so it'll wake up and encounter those checks faster. There's no risk of screwing up by waking up a process too early, since each call to `sleep()` should be in a loop that will just put it back to sleep if it's not ready to wake up yet.

```
int kill(int pid)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            p->killed = 1;

            if (p->state == SLEEPING) {
                p->state = RUNNABLE;
            }

            release(&ptable.lock);
            return 0;
        }
    }
    // ...
}
```

sleep

The last post went over the basics of `sleep()` and `wakeup()`; they act as mechanisms for *sequence coordination* or *conditional synchronization*, which allows processes to communicate with each other by sleeping while waiting for conditions to be fulfilled and waking up other processes when those conditions are satisfied.

Processes can go to sleep on a channel or wake up other processes sleeping on a channel. In many operating systems, this is achieved via channel queues or even more complex data structures, but xv6 makes it as simple as possible by simply using pointers (or equivalently, integers) as channels; the kernel can just use any convenient address as a pointer for one process to sleep on while other processes send a wakeup call using the same pointer.

This does mean that multiple processes might be sleeping on the same channel, either because they are waiting for the same condition before resuming execution or because two different `sleep()`/`wakeup()` pairs accidentally used the same channel. The result would be that a process might be woken up before the condition it's waiting for has been fulfilled. We can solve that problem by requiring every call to `sleep()` to occur inside a loop that checks the condition; that way, if a process receives a spurious wakeup call before it really should have been woken up, the loop will put it right back to sleep anyway. We saw one example of this in the `sys_sleep()` function, in which the while loop checked if the right number of ticks had passed.

A common concurrency danger with conditional synchronization in any operating system is the problem of missed wakeup calls: if the process that's supposed to send the wakeup call runs *before* the process that's supposed to sleep, it's possible that the sleeping process will never be woken up again. The problem is more general than just processes; it applies to devices too.

Imagine this scenario: a process tries to read from the disk; it'll check whether the data is ready yet and go to sleep (inside a while loop) until it is. If the disk gets to run first, then the process will just find the data ready and waiting for it, so it can continue on to use the data. If the process runs before the disk does, then it'll see the data isn't ready yet and sleep in a loop until it is; the disk will wake the process up once the data is ready.

But suppose they run at the same time, or in between each other. The process does its check and finds the data isn't ready, but before it can go to sleep, a timer interrupt or some other trap goes off and the kernel switches processes. *Then* the disk finishes reading and starts a disk interrupt that sends a wakeup call to any sleeping processes, but the process isn't sleeping yet. When the process starts running again later on, it'll go to sleep -- having already missed its wakeup call.

The problem is that the process can get interrupted between checking the condition and going to sleep, right? So why don't we just disable interrupts there with `pushcli()` and `popcli()`? add a lock there? Ah, but there's another problem: what if the disk driver is running simultaneously on another CPU? Disabling interrupts on the process's CPU wouldn't stop the other CPU from sending the disk's wakeup call too early.

Okay fine, so let's use a lock instead. The process will hold the lock while it checks the condition and sleeps, and the disk driver will have to acquire the lock before it can send its wakeup call... Can you see the problem here? If the process holds the lock while it's sleeping, the disk driver will never be able to acquire the lock in order to wake it up. That's a deadlock.

HEAD. DESK.

Ugh, okay, fine, you got me. So let's use a lock, but let's have `sleep()` release it right away, then reacquire it before waking up; that way the lock will be free while the process is sleeping so the disk driver can acquire it. Done, right? Everybody's happy?

Nope. Now we're back to the original problem: if the lock gets released inside `sleep()` before the process is actually sleeping, then the wakeup call might happen in between those and get missed.

@*#&@#\(**&@#%\)!!!

So we need a lock. And we can't hold the lock while sleeping, or we'd get a deadlock. But we also can't release it before sleeping, or we might miss a wakeup call. So... ???

See, I told you: concurrency is your worst nightmare. Ever since we decided we'd like our operating systems to do more than run a single basic process at a time, we introduced all *kinds* of problems we have to reason through. Let's check out how xv6 actually writes the `sleep()` function and think through it ourselves and try to understand if it manages to solve this problem.

We'll start by making sure of two things: (1) this CPU is currently running a process and not the scheduler (which can't ever go to sleep), and (2) the caller passed in a lock (which can be any arbitrary lock).

```
void sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = myproc();
    if (p == 0) {
        panic("sleep");
    }
    if (lk == 0) {
        panic("sleep without lk");
    }
    // ...
}
```

Next we need to release the lock and put the process to sleep. That will require modifying its state, so we should now acquire the lock for the process table. But if the lock that the process is already holding *is* the process table lock, then trying to acquire it again would cause a panic, so let's add a check for that; if we're already holding it then we'll keep using it and we don't need to release it.

```
void sleep(void *chan, struct spinlock *lk)
{
    // ...
    if (lk != &ptable.lock) {
        acquire(&ptable.lock);
        release(lk);
    }
    // ...
}
```

Okay, now it's nap time for this process. We just update its channel to `chan` and its state to `SLEEPING`, then call `sched()` to perform a context switch into the scheduler so it can run a new process. We *have* to be holding the process table lock before calling `sched()`, remember?

```
void sleep(void *chan, struct spinlock *lk)
{
    // ...
    p->chan = chan;
    p->state = SLEEPING;
    sched();
    // ...
}
```

When the process wakes up later on (if indeed it turns out that the code here works and doesn't miss any wakeup calls), it'll eventually be run by the scheduler, at which point it will context switch back here. So at that point we'll reset its channel and reacquire the original lock before returning.

```
void sleep(void *chan, struct spinlock *lk)
{
    // ...
    p->chan = 0;
    if (lk != &ptable.lock) {
        release(&ptable.lock);
        acquire(lk);
    }
}
```

Okay, well I don't know about you, but I'm still not convinced that this implementation won't miss any wakeup calls. After all, we release the original lock before putting the process to sleep, right? We're holding the process table lock at that point, which at least means that interrupts are disabled, but the process that will wake this one up might already be

running on another CPU and might send the wakeup signal in between releasing the original lock and updating this process's channel and state. Hmm... Well, as always, xv6 is brilliant, so we'll see how this gets solved in the code for `wakeup()`.

But wait! Before we move on, I have a warning for you about using this function in your own code when you start hacking away at xv6. Remember that when we first talked about deadlocks, we saw we can cause a deadlock if two processes acquire two locks in opposite orders? If process 1 tries to acquire lock A, then lock B, and process 2 simultaneously tries to acquire lock B, then lock A, then the end result is that process 1 will acquire lock A and process 2 will acquire lock B, but neither will be able to acquire the other lock since it's already being held.

If you look at the code above, the process that called `sleep()` must have already been holding a lock `lk`, then `sleep()` acquires `ptable.lock` before releasing `lk`. You know what that means: there's potential for a deadlock. So in order to avoid that, you should make sure that *any* lock you pass in to `sleep()` must *always* get acquired before `ptable.lock`. If any other function (or chain of function calls) could potentially acquire `ptable.lock` before `lk`, then you might end up with a deadlock. As always, the xv6 authors have been extremely careful to make sure that that never happens in the existing code, so you'll have to do the same thing for any code you add.

wakeup

This function is short and sweet because it procrastinates all the work it has to do by pushing it off to a helper function, `wakeup1()`. It just acquires the process table lock, calls `wakeup1()`, then releases the process table lock. It has to grab that lock since it's gonna modify the process's state in the process table.

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

xv6 has to use this kind of a wrapper function for the real wakeup function `wakeup1()` in order to let processes that are already holding the process table lock send wakeup calls too.

Okay, now before we go look at `wakeup1()`, let's get back to figuring out whether xv6's implementation of `sleep()` and `wakeup()` can lead to missed wakeup calls. Take a look at the code in `sleep()` again where the original lock gets released -- we have to acquire the process table lock *before* we can release the other lock. So now there are always two locks in play whenever we use `sleep()` and `wakeup()`.

Let's go back to the example of a process waiting on a disk read. The process acquires some disk-related lock first, then checks to see if the disk is done reading; if not, it'll call `sleep()` inside a while loop. If the disk driver runs now before the process gets to call `sleep()`, that's okay: the disk driver also has to acquire the same lock before calling `wakeup()`, so the disk would just end up spinning idly. Eventually, the process runs again and gets to call `sleep()`; there, it will first acquire the process table lock before releasing the original disk-related lock.

So what happens if the disk driver's code runs now? Now the disk would be able to acquire the original lock, so there's nothing stopping it from calling `wakeup()`. But the very first thing it has to do there is acquire the process table lock, which the process is already holding, so it just spins idly again! There's no way the disk driver could ever beat the process to acquiring this second lock, because the process already held the first (disk-related) lock before acquiring the second one (the process table lock). Now the process can finish going to sleep and switch into the scheduler, which will eventually release the process table lock. So then the disk driver can acquire it, release the first lock, and finally send its wakeup call.

Moral of the story? There's no way for xv6 to ever have any missed wakeup calls! The trick was to use two locks, and acquire the second before releasing the first. But coming up with that solution isn't as easy as saying "oh, just use two locks!" The solution only works because of the way the process table lock is already being handled by so many other parts of the kernel code. For example, if the context switch into the scheduler wasn't guaranteed to release the process table lock, then the disk driver in the example would never be able to acquire it after the process goes to sleep, resulting in a deadlock. The solution works because of all the design decisions in xv6 up to this point.

wakeup1

Okay, I'll stop fawning over the intricacies of xv6 concurrency management now so we can look at how wakeup calls actually happen. Remember, this is a separate function from `wakeup()` because sometimes the scheduler needs to send a wakeup call while it's already holding the process table lock. So we're gonna assume that every function that ever calls this is already holding it.

The implementation here is actually pretty simple now: we'll just iterate over the process table and set every single process that's sleeping on channel `chan` to `RUNNABLE`.

```
static void wakeup1(void *chan)
{
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == SLEEPING && p->chan == chan) {
            p->state = RUNNABLE;
        }
    }
}
```

Now, there might be multiple processes sleeping on this channel, so this will wake them all up. For some of those processes, this might be a spurious wakeup, so again, we should always make sure to call `sleep()` in a loop that checks for some condition to be satisfied. Even if multiple processes do have their sleep conditions satisfied, they'll have to reacquire their original lock before returning out of `sleep()`, so only one of them will do so and the others will spin until the first one is done.

Why not just wake up the first process we find that's sleeping on `chan`? Then we could avoid the extra overhead of a bunch of processes waking up, checking a condition, and going back to sleep, or even spinning idly waiting to reacquire the lock before returning. The issue is that the channels may not be unique, so there's no way to know which of all the

sleeping processes is the one whose sleep condition has just been fulfilled. If we wake up the wrong process, it'll just go back to sleep, but the right process didn't wake up, so that means we've lost a wakeup call.

exit

Okay, so we saw above that `kill()` doesn't really kill a process immediately; it shirks that responsibility and lets `exit()` handle it instead... except even `exit()` won't really fully kill a process. Whew, a process's death just keeps getting dragged out forever, doesn't it? It's starting to feel like a cheesy death scene in a tragedy; I bet the process is tired of suffering the slings and arrows of outrageous fortune by now.

But it does make sense. Think about what we have to do in order to wrap up a process and recycle its slot in the process table: we have to close out any open files and reset its current working directory, free its kernel stack and its entire page directory, then notify the parent that it's done running.

The trouble comes with freeing the kernel stack and process page directory. This function runs in kernel mode, so while the user stack in the lower half of memory will be unused now, the kernel stack is still needed in order to keep executing the instructions for `exit()`. Also, with the exception of the times when it's running the scheduling algorithm, the kernel uses the page directory of the current process. The moment we free that page directory, the very next memory access will be to an invalid page; the CPU would trigger an exception then. That exception would eventually get routed to `exit()` again, except, oh wait, we can't even run any instructions without generating another exception, because the entire page directory and stack have been freed; that's a double fault. So then the CPU would try to handle *that* exception, which would cause the dreaded boogeyman of OS devs around the world: a triple fault. After a fault triggers a second exception, which itself triggers a third exception, the CPU just decides that the kernel in its current state doesn't have its shit together enough to keep running, so it takes over and reboots the whole system. Oops.

Okay, so let's not do that. That means we can't free the kernel stack nor the page directory until we're running on a different stack/page directory combo. That could happen in `scheduler()` while we're using the page directory `kpgdir`, or it could happen while we're running another process. xv6 does it while it's running the parent process, in the `wait()` system call. If you haven't used that in Linux before, `wait()` lets a parent process sleep until a child process is done running. xv6 will use `wait()` to finish cleaning up after an exited child process too.

Now, the very first process that starts running in xv6 (`initproc`, which loads and runs the shell) obviously has no parent process, but that's okay because that one should never exit as long as the system is up. So let's start this function off by making sure that the process that's exiting isn't the initial process.

```
void exit(void)
{
    struct proc *curproc = myproc();
    if (curproc == initproc) {
        panic("init exiting");
    }
    // ...
}
```

Next we'll close all open files and clear the current working directory; again, we haven't seen the file system functions used here, but we'll get to them soon.

```
void exit(void)
{
    // ...

    // Close all open files
    for (int fd = 0; fd < NOFILE; fd++) {
        if (curproc->ofile[fd]) {
            fileclose(curproc->ofile[fd]);
            curproc->ofile[fd] = 0;
        }
    }

    // Clear the current working directory
    begin_op();
    iput(curproc->cwd);
    end_op();
    curproc->cwd = 0;

    // ...
}
```

Now we only have one thing left to do: notify the parent process that this process has exited. If the parent process is currently sleeping in `wait()`, then we'll need to wake it up. But maybe the parent process is currently in the middle of executing other code before it gets to `wait()`; we don't want it to miss the wakeup call... oh wait, but that's okay, remember? The implementations of `sleep()` and `wakeup()`/`wakeup1()` guarantee that we can't miss a wakeup call as long as we're holding the right lock; `wait()` will use the process table lock for that. So let's acquire it now and send a wakeup call.

```
void exit(void)
{
    // ...
    acquire(&ptable.lock);
    wakeup1(curproc->parent);
    // ...
}
```

Now, remember that a sleeping process needs to check some condition in a loop; how can the parent process know that the child has exited? Hmm, okay, let's set the child's state to `ZOMBIE`. That'll also prevent the scheduler from trying to run it again.

Ah, but hang on a sec... what if the parent process has itself been killed, i.e. the current process has been orphaned? (Again with the melodrama...) A process can't run any more user code after `exit()`, so an undead parent process would never get to call `wait()` to clean up after its children. In that case, we'd have to find another process that could adopt a child.

So let's just solve that problem now: this process is about to shuffle off its mortal coil, so let's figure out if it has any children and pass them off to another process that can keep raising them as its own. But which process is guaranteed to live long enough to clean up after those children once they die? Ah, `initproc`, of course! That first process is immortal, so it should be able to look after any children that this process might leave behind after it makes its quietus with a bare bodkin.

So we'll iterate over the process table, looking for any processes with parent process equal to `curproc`; if we find any, we'll have `initproc` adopt them. If any of our now-abandoned children has already exited before we did, we'll send a wakeup signal to `initproc` too in case it's sleeping in `wait()`.

```
void exit(void)
{
    // ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->parent == curproc) {
            p->parent = initproc;
            if (p->state == ZOMBIE) {
                wakeup1(initproc);
            }
        }
    }
    // ...
}
```

Okay, now it's finally time for this process to find out what dreams may come in that sleep of death. We'll set its state to `ZOMBIE` and context-switch into the scheduler, never to return; if something goes wrong and the scheduler *does* return, we'll panic in order to keep this function from returning into user code again.

```
void exit(void)
{
    // ...
    curproc->state = ZOMBIE;
    sched();

    panic("zombie exit");
}
```

wait

Like we said above, this system call lets a parent process wait for a child process to exit; it also cleans up after the child process has exited.

First, we don't even know if this process has any children, so we'll have to check by iterating through the process table and checking each process's parent to see if it matches the current process. If it does, then we'll check if it's a zombie, in which case we can clean it up and return its process ID.

We should also deal with two edge cases: first, if the process has no children at all, and second, if the process does have children but none of them are dead yet. In the first case, we'll just return -1 to report failure; in the second case we'll put the current process to sleep until one of its children exits. The `sleep()` call means we'll have to do these checks inside an infinite loop.

Alright, let's get started by getting the current process and acquiring the process table lock, then starting an infinite loop.

```
int wait(void)
{
    struct proc *curproc = myproc();

    acquire(&ptable.lock);

    for (;;) {
        // ...
    }
}
```

Inside the loop, we'll use a variable `havekids` as a boolean to track whether we've found any child processes. Then we can iterate over the process table, skipping any processes for which the current process is not the parent. If we find any children, we'll set `havekids` to 1.

```
int wait(void)
{
    // ...
    for (;;) {
        int havekids = 0;

        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->parent != curproc) {
                continue;
            }
            havekids = 1;
            // ...
        }
        // ...
    }
}
```

If we did find a child process, we should check if it's a zombie, in which case it's time to finish its clean-up. That means freeing its kernel stack and its page directory and recycling its `struct proc` so that it can be reallocated to another process later on.

```
int wait(void)
{
    // ...
    for (;;) {
        // ...
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            // ...
            if (p->state == ZOMBIE) {
                int pid = p->pid;

                // Free child's kernel stack
                kfree(p->kstack);
                p->kstack = 0;

                // Free child's page directory
                freevm(p->pgdir);

                // Recycle child's struct proc
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;

                release(&ptable.lock);
                return pid;
            }
        }
        // ...
    }
}
```

Now, if `havekids` is still zero by the time we finish the for loop, that means the process doesn't have any children, so we should report failure. We'll also check if the process has been marked as killed in the meantime.

```
int wait(void)
{
    // ...
    for (;;) {
        // ...
        if (!havekids || curproc->killed) {
            release(&ptable.lock);
            return -1;
        }
        // ...
    }
}
```

Finally, if it *does* have children, but none of them have exited yet, we'll put the process to sleep. It'll get woken up when a child exits, at which point it'll restart the outer for loop at the top and start looking through the process table again.

```
int wait(void)
{
    // ...
    for (;;) {
        // ...
        sleep(curproc, &ptable.lock);
    }
}
```

Summary

By now, we've looked at a good chunk of the system calls available in xv6. These system calls wrap up the mechanisms that xv6 uses to create and exit processes with `fork()`, `kill()`, `exit()`, and `wait()`, and introduced `sleep()` and `wakeup()` as a means for (limited) inter-process communication.

So what's left now? The rest of the kernel code we're gonna look at will just focus on communicating with various hardware devices like the serial port, console, and keyboard. Those drivers are relatively short, but there's one device that will require a lot more work: the disk. Storing files on disk and making sure they persist across reboots require careful planning, and making files conveniently accessible to users requires an entire system of abstractions layered on top of each other, along with a whole host of file-related system calls.

System Calls: Routing

We said in the last post that system calls are the primary means for user processes to request some action by the kernel; system calls mediate processes' access to hardware resources.

If a user process wants to generate a system call, it starts a trap with the trap number for system calls. Then it identifies which of the various xv6 system calls it wants to do and passes any required arguments. The processor will then handle the trap instruction using the code we saw in the last post. Eventually, it'll get to the `trap()` function, which will recognize the trap number as a system call and pass it on to the `syscall()` function.

`syscall()` is itself a routing function like `trap()`; it'll figure out which system call the process created and redirect it again to the appropriate kernel code.

syscall.c

All system calls use the same trap number: 64, or `T_SYSCALL`, but xv6 has multiple system calls, so we need another number for a process to identify which system call it wants to run. The convention on x86 is to use a system call number which the calling process should put in the `%eax` register, which usually holds return values. Then the kernel's handler function (here, `syscall()`) can just check `%eax` to figure out which system call to run. The system call numbers are defined in syscall.h (https://github.com/mit-pdos/xv6-public/blob/master/syscall.h). There you can see that, e.g. `SYS_fork` is defined as 1, `SYS_exit` is 2, and so on.

All the system call functions are defined in other files, so we'll have to import their declarations with the `extern` keyword:

```
// ...
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
// ...
```

Now we've got the numbers and the functions. Note that the numbers start with uppercase `SYS_` and the functions start with lowercase `sys_`, so make sure your kernel hacking adventures don't do anything like `SYS_fork()`; use `sys_fork()` instead.

We'll also need a way to map the numbers to those system call functions so that `syscall()` can call the right one depending on the number. We could use another `switch` statement like we did in `trap()`, but there are 21 system calls here, so that would get pretty long; also, each number will just call the specific function, unlike the different trap numbers which required different responses (e.g., the timer interrupt trap number didn't call any function at all). xv6 does something else this time that's much simpler and more elegant, but it uses some slightly-obscure C features, so we'll go over it carefully.

Remember function pointers from way back in the boot loader? Functions are just a set of instructions in order, loaded somewhere in the kernel's code segment, so C lets us use the function's name as a pointer to the beginning of its code in memory. So if we have a C function like `int func(char c)`, then `func` is its function pointer. We could even assign it to a variable; that variable's type would be a pointer to a function of argument type `char` and return type `int`; then we could call the function using the new pointer too. Here's an example that would print "Match!" to the screen:

```
int m = func('a');

int (*func_ptr)(char) = &func;
int n = (*func_ptr)('a');

if (m == n) {
    printf("Match!\n");
}
```

So instead of a big old `switch` statement, the `syscall()` function will use a static, global array of pointers to all the system call functions we just imported above. (Remember that the `static` keyword in front of a variable means it always occupies the same fixed place in memory.) It'll work because all the functions have the same argument type (`void`) and return type (`int`), so their pointers all have the same type and can fit inside a single array. Then we can get the right function by just using the system call number to index into the array of function pointers.

Now, we'd have to be super careful to add the function pointers into the array in the right order so that the indices match up. Even worse, there is no system call with number zero, so we'd have to skip that entry of the array. This could get complicated. Luckily, even though humans are bad at this kind of thing, computers are *really* good at it. So instead of trying to line them up by hand, we can use the array notation from `procdump()` in the post on processes where we specified the value of each entry of an array like this with the index in square brackets, like this:

```
int arr[] = { [2] 5, [0] 1, [4] -2 };
```

The C compiler will use the indices we wrote there to figure out that the array needs 5 entries (indices 0 to 4), and entry 0 is 1, entry 2 is 5, and entry 4 is -2. Entries 1 and 3 will just be initialized to zero.

So at the end of the day, our array of pointers to system call functions looks like this:

```
// ...
static int (*syscalls[])(void) = {
    [SYS_fork]      sys_fork,
    [SYS_exit]      sys_exit,
    [SYS_wait]      sys_wait,
    [SYS_pipe]      sys_pipe,
    [SYS_read]      sys_read,
    [SYS_kill]      sys_kill,
    [SYS_exec]      sys_exec,
    [SYS_fstat]     sys_fstat,
    [SYS_chdir]     sys_chdir,
    [SYS_dup]       sys_dup,
    [SYS_getpid]    sys_getpid,
    [SYS_sbrk]      sys_sbrk,
    [SYS_sleep]     sys_sleep,
    [SYS_uptime]    sys_uptime,
    [SYS_open]      sys_open,
    [SYS_write]     sys_write,
    [SYS_mknod]     sys_mknod,
    [SYS_unlink]    sys_unlink,
    [SYS_link]      sys_link,
    [SYS_mkdir]     sys_mkdir,
    [SYS_close]     sys_close,
};
// ...
```

Okay great, now we're ready to route system calls to the right function.

syscall

The first thing we need to do is get the system call number so we can figure out which function to call. We said above that the x86 convention is to store it in the `%eax` register, but we might have a problem: by the time we get to `syscall()`, the processor has already executed the code in the trap handler function for trap number `T_SYSCALL`, which sent it to `alltraps()`, which replaced all the register contents with those of `trap()`, so the system call number is probably long gone from `%eax`.

But wait, all is not lost! `alltraps()` saved all the registers in a `struct trapframe` for the current process. So we can just read the value of `%eax` from there. Whew, that was some good forward-thinking.

```
void syscall(void)
{
    struct proc *curproc = myproc();
    int num = curproc->tf->eax;
    // ...
}
```

Now we just need to do one more thing: call the function that corresponds to that number. We're gonna use the array of function pointers above, but we have to be careful: this number was given to us by a user process. A malicious user process might pass in an invalid number in the hopes of getting the kernel to carry out some undefined behavior which might lead to an easy exploit. So in order to keep up good security practices, the kernel should *always* distrust anything originating from user code and handle it carefully, preferably with three-inch-thick lead-lined gloves. So let's think about it: what might go wrong?

First of all, any entries that weren't explicitly initialized above (including the 0 entry) will have been automatically initialized to zero, i.e. a null pointer. Also, a number that's bigger than the highest system call number will make us do an out-of-bounds read from the array, thus possibly executing some arbitrary kernel code that's stored after the array in memory. So we should check that (1) the number is greater than 0, (2) it's smaller than the number of elements in the array, and (3) the entry it points to is not a null pointer.

Finally, the `%eax` register is usually used in x86 to store return values, so we'll put the return value of the system call function there. If any of the above checks failed, we'll just print a message to the console and return -1 to indicate failure.

```
void syscall(void)
{
    // ...
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
        // Invalid system call number
        cprintf("%d %s: unknown sys call %d\n", curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

The system call handler function will store its return value in `%eax`; after that, `syscall()` will return to the line below where it was called in `trap()`. After executing the rest of the code there, `trap()` will return into `trapret()`, which ends with an `iret` (interrupt return) instruction to tell the processor to switch to user mode and resume executing the process's code.

fetchint

Take a look at the `sys_` functions we imported above: they all have argument type `void`. But if you think about it, many system calls need an argument: for example, `open()` needs to know which file to open, `chdir()` needs to know which directory to open, `kill()` needs a PID to know which process to kill, etc. So why did we make them all have argument type `void`?

The trouble is that until we get the system call number in `syscall()` above, we have no way of knowing which function we'll need. And each function takes arguments with different types, e.g. `open()` might need a string for the file to open but `kill()` might need an integer for the PID. So there's no way for the kernel to know which arguments to expect in `syscall()`, even though the arguments were already pushed on the stack. The task of recovering the arguments from the stack will have to fall to each of the `sys_` functions. But let's go ahead and make their lives a little easier by setting up some nice helper functions now.

The system call functions might take integers, strings, or pointers, so we'll need functions to fetch each of those types. `fetchint()` is one example; it takes a user virtual address (an integer argument's location in memory) and a pointer to an integer where we can store the integer we find. Then it returns 0 if it was able to find it, or -1 if it failed.

Just like `syscall()` above, we need to treat anything passed from user space with extreme caution. A user process that tries to read or write memory outside its address space will cause a segmentation fault or page fault and be killed, but the kernel has free reign over memory, so a malicious process might try to trick the kernel into doing that *for* it by putting its "argument" outside of the user's address space. So we have to start by checking that the entire 4 bytes of the integer is inside the process's address space.

```
int fetchint(uint addr, int *ip)
{
    struct proc *curproc = myproc();

    if (addr >= curproc->sz || addr + 4 > curproc->sz) {
        return -1;
    }
    // ...
}
```

Now we can just cast the address to a pointer, dereference it, and store the value in `*ip`.

```
int fetchint(uint addr, int *ip)
{
    // ...
    *ip = *(int *)(addr);
    return 0;
}
```

Note that we can use an address like `addr` which will be in the lower half of memory because traps don't perform a full context switch, so we're still using the process's page directory even though we're in kernel mode (ring 0). If we had switched to a kernel page directory, we'd have to call `walkpgdir()` or `uva2ka()` to figure out the corresponding kernel virtual address for `addr`.

Now hopefully, if you've taken anything away from my past rants about undefined behavior in C, you noticed something wrong with this function. If you didn't, take another look; I'll wait.

Did you see it? We're dereferencing `addr` without checking that it's not null, so if the user passed in a null address, we'd dereference a null pointer! We also dereference `ip` without a similar check, but at the very least `ip` is passed in by the kernel.

This could be very dangerous -- in general, it's undefined behavior in C, but now that we've seen the code for handling traps, we're actually at a point where we can figure out what would happen in xv6 if a null pointer gets dereferenced, so let's take the opportunity to think about it for a bit.

First, what would happen if the kernel dereferenced a null pointer? Well, if the kernel is currently using `kpgdir` as a page directory, the address 0 isn't mapped to anything, so when the paging hardware goes to figure out which physical address corresponds to the kernel virtual address 0, it would fail and generate a "General Protection Fault" (trap number 13, or `T_GPFLT`). That would start running the trap handler code, which would eventually get to the `switch` statement in `trap()` (see the last post). Trap number 13 would fall under the `default` case, and the if statement there would recognize that it originated in the kernel. So it would print an error message to the console, then panic.

Okay, what if we're using a process's page directory, e.g. during a system call? Address 0 is in the lower half of memory, so it's a user virtual address. The result will depend on whether that page and its page table are mapped in the process's page directory. If they are, then dereferencing a null pointer might be fine after all. But if they're not mapped, dereferencing a null pointer will cause a General Protection Fault. This time, `trap()` would print an error message to the console, then mark the process to be killed.

Now, killing a process or causing a kernel panic might not sound like a huge deal. In fact, xv6 does a great job here by killing a process that might have dereferenced a null pointer or caused the kernel to do so. A kernel panic would be much worse -- think about how annoying it would be if that PDF you downloaded from that one sketchy website installed some malware that made your kernel panic all the time -- the OS would become unusable. In fact, this is an example of a "denial of service" vulnerability -- a malicious process might not be able to read or write arbitrary memory or execute arbitrary code, but it can still keep you from using your machine the way you expect to.

Just like `uva2ka()`, this function will only get called by one other function (we'll see it soon), so it just so happens that under the current xv6 code, it'll all be okay because it should never get passed a null pointer. But everything from my rant about `uva2ka()` applies here: if you add any kernel code that calls this function, be *VERY* careful and add your own null checks.

Okay, deep breath now. /rant.

fetchstr

Fetching a string argument is tricky too; strings in C are just pointers to an array of characters that ends in nul, i.e. `'\0'`, so this time we have to make sure that both the pointer *and* the entire string are in the user's address space; otherwise, we could unwittingly read from some arbitrary memory location and pass the data back to the user process.

So we'll start by making sure the pointer itself is in a valid address:

```
int fetchstr(uint addr, char **pp)
{
    struct proc *curproc = myproc();

    if (addr >= curproc->sz) {
        return -1;
    }
    // ...
}
```

Now we'll store the string pointer in `*pp`. We'll also get a pointer to the end of the process's virtual address space so we can make sure the entire string is inside its bounds.

```
int fetchstr(uint addr, char **pp)
{
    // ...
    *pp = (char *) addr;
    char *ep = (char *) curproc->sz;
    // ...
}
```

How can we check if the entire string is inside user memory? Well, a string ends with a nul byte, `'\0'`, so we just have to start scanning the memory starting from `*pp` up to `ep` until we find a zero byte. If we find one in that range, then the entire string is in user memory and we can return its length to indicate success; otherwise the string overflows past the end of the process's virtual address space, so we should return -1 to indicate failure.

```
int fetchstr(uint addr, char **pp)
{
    // ...

    // Scan for a nul byte inside process's address space
    for (char *s = *pp; s < ep; s++) {
        if (*s == 0) {
            // If nul byte found, return the length
            return s - *pp;
        }
    }
    // String is not nul-terminated inside process's memory, so report failure
    return -1;
}
```

Note that again, we're dereferencing `pp` and `addr` without any null checks (where `addr` is definitely the bigger concern, since it's user-generated), and again, it's gonna work out okay (a misbehaving process will just get killed), but once more: be careful if you use this function for your own kernel hacks.

argint

This is the main function that the `sys_` system call functions will use to recover an integer argument; it's basically just a wrapper for `fetchint()`. The arguments are an integer `n` to say we want the nth integer argument, and a pointer `ip` to store the recovered argument in. We have to call `fetchint()` with an address argument, so the main task now is to figure out where in memory the nth integer argument should be.

We're gonna have to use the x86 function call conventions again. Remember how whenever we call a function in x86, its arguments get pushed onto the stack in reverse order (i.e., from right to left), so that the first argument is at the top of the stack (i.e., lowest memory address)? Then we push a return address (`%eip`) and the old stack base pointer `%ebp`. Normally, the stack pointer would just keep going on to the next slot on the callee's stack, but in this case the code in `alltraps()` saved all the registers (including the stack pointer `%esp`) in a `struct trapframe` before calling `trap()` or `syscall()`.

That means we can recover the old value of `%esp` from the trap frame and look one spots below that on the stack (i.e., 4 bytes higher in memory, since `int`s are 4 bytes) to get the first (`n = 0`) argument. The second argument (`n = 1`) would be 8 bytes higher than `%esp`, and so on. Pretty neat.

Okay, now that we've got that down, the code for this function is pretty straightforward.

```
int argint(int n, int *ip)
{
    return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

argptr

Some of the system call functions will have pointer arguments, so this function recovers them. Pointers are 4 bytes in x86, so we can use `argint()` to get the pointer itself before performing some additional checks to make sure the pointer and the address it points to are valid.

The arguments are `n` (to retrieve the nth function argument), a pointer `pp` to an address where we can store the retrieved pointer, and the size of the block of memory that the retrieved pointer points to.

Let's start off by just retrieving the value of the pointer as an integer using `argint()`; that'll make sure that the number `n` is valid.

```
int argptr(int n, char **pp, int size)
{
    struct proc *curproc = myproc();

    int i;
    if (argint(n, &i) < 0) {
        return -1;
    }
    // ...
}
```

Now we have to make sure that the pointer we just retrieved is itself valid, i.e. that the size is nonnegative and the beginning and end of the memory block it points to are both within the process's address space.

```
int argptr(int n, char **pp, int size)
{
    // ...
    if (size < 0 || (uint) i >= curproc->sz || (uint) i + size > curproc->sz) {
        return -1;
    }
    // ...
}
```

Finally, we can store the pointer in `*pp` and return 0.

```
int argptr(int n, char **pp, int size)
{
    // ...
    *pp = (char *) i;
    return 0;
}
```

argstr

A string is just a pointer in C, so we can recover the pointer's value using `argint()` again, then pass it to `fetchstr()`. The former will make sure `n` is valid, and the latter will make sure the string is nul-terminated and resides entirely in the process's address space.

```
int argstr(int n, char **pp)
{
    int addr;
    if (argint(n, &addr) < 0) {
        return -1;
    }
    return fetchstr(addr, pp);
}
```

sysproc.c

So now we know how `syscall()` will route a system call trap to the right `sys_` function, and we've seen how those functions can recover arguments from the process's stack. Let's see some examples in action; most of these will be simple wrapper functions.

sys_fork

All the hard work here is gonna be done by `fork()`, which will create a new child process by cloning the parent process's virtual address space. We don't need any arguments for this, so we'll just call `fork()`.

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

sys_exit

`exit()` closes out a process, but it puts it in the `ZOMBIE` state so that the parent process can call `wait()` to find out it's done running. `exit()` should never return, so we'll add a return value here to make the compiler happy, but it should never get executed.

```
int sys_exit(void)
{
    exit();
    return 0;
}
```

sys_wait

This system call is the parent process's counterpart to `exit()`; it'll do as its name says and wait until the child process exits.

```
int sys_wait(void)
{
    return wait();
}
```

sys_kill

The `kill()` system call sounds like a more aggressive version of `exit()`: after all, we're killing another process against its will, right? But in reality it would be way too complicated to do that: the process might be running on another CPU, midway through updating some kernel data structure, or about to wake up another process that's asleep. Killing it by force might screw up a lot of other things.

So instead `kill()` just tags it with the `killed` field in its `struct proc`; eventually either the process will call `exit()` on its own, or it'll generate another trap, at which point the code in `trap()` will call `exit()` on it.

`kill()` needs an integer argument: the process ID for the process we wish to kill. So now we can see the payoff of writing those functions above.

```
int sys_kill(void)
{
    int pid;
    if (argint(0, &pid) < 0) {
        return -1;
    }
    return kill(pid);
}
```

sys_getpid

The `getpid()` system call is so simple that it doesn't even have another function for this `sys_getpid()` to call. We'll just return the PID for the current process.

```
int sys_getpid(void)
{
    return myproc()->pid;
}
```

sys_sbrk

If you're not familiar with system calls like `brk()` and `sbrk()` on Unix systems, here's what they do: they grow or shrink the virtual address space of a process. `brk()` sets its new size to a specific maximum address; `sbrk()` grows or shrinks the process by a certain size in bytes and returns its old size. They're mostly used to implement higher-level memory management functions like `malloc()`. Heh, "high-level" probably isn't high on your mind when you think of adjectives for `malloc()`, right? Anyway, xv6 only has `sbrk()`, so let's check out its `sys_` wrapper function.

We'll need an integer argument (the number of bytes to grow or shrink by), so let's grab that.

```
int sys_sbrk(void)
{
    int n;
    if (argint(0, &n) < 0) {
        return -1;
    }
    // ...
}
```

Now we can use `growproc()` from our posts on paging to grow the process by `n` bytes. But we want to return the old size, so we'll have to grab that before we change it with the call to `growproc()`.

```
int sys_sbrk(void)
{
    // ...
    int addr = myproc()->sz;

    if (growproc(n) < 0) {
        return -1;
    }

    return addr;
}
```

sys_sleep

The `sleep()` function is pretty interesting; we'll get to the implementation details later, but let's talk about the broad strokes now. You might be familiar with the `sleep()` system call in Unix systems; you pass it an integer (usually in milliseconds) and it puts your process to sleep (i.e., leaves it inactive or not running) for that amount of time.

However, `sleep()` plays a dual role in xv6: the kernel will call `sleep()` for processes that need to wait while something else happens, e.g. waiting for a disk to read or write data. That way the processes don't end up idly spinning in a loop or something and wasting valuable CPU time.

Implementing that is tricky; there's no way to know how long it would take for whatever condition the process is waiting on to be satisfied, so it's not like we can just stick in a random amount of time in the call to `sleep()` and hope the condition is satisfied by then. So instead the `sleep()` function will just "put a process to sleep" (read: make its state `SLEEPING` so it can't be run by the scheduler) on a *channel*, which is just an arbitrary integer. Then later on the kernel can wake up any processes sleeping on that channel. So for example, the kernel can put a process waiting on the disk to sleep using a specific channel that's assigned to the disk; then when the next disk interrupt occurs it can wake up any processes that might be sleeping on the disk channel.

Okay so that's all well and good for the kernel's use of `sleep()`. But what about the regular old `sleep()` system call? The argument is an integer that represents the number of ticks to sleep for; how are we gonna turn that into a channel to sleep on?

The answer is pretty neat (at least I think so): we'll set the channel to the address of the `ticks` counter. Remember, `ticks` is a global variable that gets incremented with every timer interrupt. Go check out the code in `trap()` again: each timer interrupt sends a wakeup call to any processes that might be sleeping on the `&ticks` channel. That should wake the process at every timer interrupt. Then we'll just stick that inside a for loop so it keeps sleeping forever until the right amount of ticks have passed.

Let's start by retrieving the integer argument, which is the number of ticks to sleep for.

```
int sys_sleep(void)
{
    int n;
    if (argint(0, &n) < 0) {
        return -1;
    }
    // ...
}
```

That argument `n` is a relative count, since a user process won't necessarily know how many ticks have already gone by. So let's get the current tick count before we put the process to sleep.

```
int sys_sleep(void)
{
    // ...
    acquire(&tickslock);
    uint ticks0 = ticks;
    // ...
}
```

Now we just have to write that while loop I mentioned above to put the process to sleep until `n` ticks have passed. Since we started counting at `ticks0`, the condition should be satisfied when `ticks - ticks0 == n`.

Two more details: first, we'll add a check inside the while loop to see if the current process has been tagged to be killed; if so, we'll just return -1 so we can hasten the process's actual death by letting it run more code so the kernel will call `exit()` on it at the next trap. Second, the function `sleep()` takes another argument in addition to the channel: a lock. It'll release the lock for us and reacquire it before waking up so that a sleeping process doesn't hog a lock when it doesn't need it.

```
int sys_sleep(void)
{
    // ...
    while (ticks - ticks0 < n) {
        if (myproc()->killed) {
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}
```

sys_uptime

The `uptime()` system call just returns the amount of ticks that have passed since the system started. This is another one that's so simple it doesn't need another function, so we'll take care of it all here.

We just acquire the lock for `ticks`, get its current value, release the lock, and return the value we got.

```
int sys_uptime(void)
{
    acquire(&tickslock);
    uint xticks = ticks;
    release(&tickslock);
    return xticks;
}
```

Running System Calls from User Code

We have system calls now! Well, not quite -- we still have to check out the actual functions like `exit()`, `sleep()`, `kill()`, etc. Plus, we only saw the `sys_` wrapper functions for *some* of the system calls here; the rest are in sysfile.c (https://github.com/mit-pdos/xv6-public/blob/master/sysfile.c), which we'll get to after we understand the xv6 file system.

But let's pause for a second and think about how a user process will send a system call. Like let's say you're writing some C code for a user program that will run on xv6 and you want to create a child process with `fork()`. What should you do?

Well, if you were coding for a Unix system like Linux or macOS, you'd just write a call to `fork()` in your code. But that can't be right in xv6, can it? After all, `fork()` is a kernel function, to be run in kernel mode with a current privilege level of 0. Plus, isn't it supposed to be called by `sys_fork()`? So should we call that?

None of these options will work. Well, yes, you do end up just calling `fork()`, but it's *not* the kernel function `fork()`, so if you're expecting that one, you'll be surprised when it doesn't behave the way you want it to. You won't be able to use any kernel code at all in your user program for xv6. This is a mistake I've seen a *lot* of people make in their xv6 OSTEP projects, so bear with me for a second while I explain why you can't do it; feel free to skip the next section on the Makefile if you already know why.

Makefile

To see why, let's check out the xv6 Makefile (https://github.com/mit-pdos/xv6-public/blob/master/Makefile) to see how xv6 is actually compiled, built, and run. There's a ton of stuff in there, but take a second to think about this: how do you usually run xv6? I bet it's a command like `make qemu` or `make qemu-nox`, right?

If you're not familiar with Makefiles, here's a quick primer: each command like `make qemu`, `make clean`, etc. is specified in the Makefile with a rule that looks like this:

```
mycmd: dependency1 dependency2 ...
    build_cmd1
    build_cmd2
    # ...
```

So if I run `make mycmd`, the `make` program will check that `dependency1`, `dependency2`, etc. are up to date; if they're not, it'll update them by looking up *their* rules and executing those to update them. Then it'll execute `build_cmd1` on the shell, followed by `build_cmd2`, etc.

Okay, I know that might be confusing, so let me simply the `make qemu` command a bit to make it more readable (note that I cut a lot of stuff out here, so don't try to run xv6 with what I wrote below).

```
# ...
qemu: fs.img xv6.img
    qemu -drive file=fs.img,index=1 -drive file=xv6.img,index=0
# ...
```

This just says that in order to run `make qemu` when you type it on the terminal, the `make` program first has to make sure that both `fs.img` and `xv6.img` are fully up to date. Then once they are, it can just run the shell command `qemu` with the options `-drive file=fs.img,index=1` and `-drive file=xv6.img,index=0`. Those options are just regular flags like the ones you're probably used to with stuff like `ls -a` or `rm -rf`. In this case, they tell `qemu` to use the files `fs.img` and `xv6.img` as virtual hard drives, with `xv6.img` as disk number 0 and `fs.img` as disk number 1.

Okay, let's check out the `make` command for `xv6.img` next.

```
# ...
xv6.img: bootblock kernel
    # some dd commands here
# ...
```

Hey, that's interesting, we already saw `bootblock` in a prior post. That's the one we get when we compile the boot loader. `kernel` is, well, all the kernel code. The `dd` command is often used in Unix systems to format and set up disks; the details aren't important here, so I left them out for now. The point is that the boot loader got compiled separately from the kernel code, remember? But their machine code files get smushed into the same (virtual) disk together as `xv6.img`, which will be disk 0 when we run in `qemu`.

Not let's check out the (slightly simplified) `make` command for `fs.img`.

```
# ...
UPROGS = cat echo forktest grep init kill ln ls # ...

fs.img: mkfs README $(UPROGS)
    ./mkfs fs.img README $(UPROGS)
# ...
```

Okay, so `UPROGS` is just a list of all the user programs. Each of those gets compiled separately; e.g. if you look in their source code, you'll see each one has its own `main()` function. Then the shell command says to run `mkfs` to create a file system called `fs.img` with `README` and all the user programs as files.

The point of this detour is this: the boot loader gets compiled as a single unit, as does the entire kernel code. But the user programs are compiled one at a time. So if you write a user program for xv6, you should add it to the list in `UPROGS` (as well as in `EXTRA`) and expect it to get compiled individually and stuck onto the `fs.img` disk.

That means there's no way for a user program to call into any kernel code; the linker wouldn't even be able to match up the call to the right function. So no user program will ever be able to call functions like (the kernel's) `fork()`. Think about it: if you write a program in C and compile it to run on Linux, do you expect to have to recompile the entire Linux kernel just to run your one little program? No, right?

But certainly we can't just expect every single program ever to be totally self- contained. You also don't have to rewrite and recompile all of `malloc()` every time you write a C program. So operating systems provide libraries for users to include and call in their programs. Aha! So all we need to do in order for user processes to execute system calls is to provide a library. That library is usys.S (https://github.com/mit-pdos/xv6-public/blob/master/usys.S).

usys.S

Let's trace back to the beginning of a trap. In order to execute a system call, we're supposed to send the processor an `int` instruction with a specific trap number; that would be `int 64` for system calls on xv6. We're also supposed to stick the system call number in the `%eax` register. Let's say we want to call `fork()`. According to syscall.h (https://github.com/mit-pdos/xv6-public/blob/master/syscall.h), the system call number for fork is `SYS_fork`, or 1. In order to send a specific x86 instruction and manipulate individual registers, we'll have to write our system call library in assembly. Here's what it would look like for the `fork()` system call:

```
.globl fork
fork:
    movl    $1, %eax
    int     $64
    ret
```

Okay, that's easy enough, but we have 21 of these to write, and it would be pretty easy to make a mistake or write the wrong system call number. Let's automate it instead with a C preprocessor macro. We've seen plenty of examples of defining simple constants with `#define` directives for the preprocessor, but we haven't looked at them too closely until

now.

The C preprocessor is a piece of software that edits C (or assembly) code before it's compiled. Preprocessor directives like `#define A 5` create macros that are expanded to replace every instance of `A` in the code with the number 5; directives like `#include "header.h"` expand such that they essentially copy- paste all the code in the file `header.h`. We can also create function-like macros like the `P2V()` and `V2P()` macros we've used often by adding a parameter inside parentheses; unlike functions, these will be expanded *before* compilation to paste the code into every instance of its use, thus avoiding the usual overhead associated with a function call. Function-like macros are also generic, in a sense, since they don't require specifying parameter types or return types (as long as it works within the places where the macro will be used). Note that there are some drawbacks: macros aren't type-checked, they can evaluate their arguments more than once, we can't use pointers to them like we can with functions, and they can result in larger code.

We're gonna use a function-like macro here to create the assembly code for each system call function so that it gets expanded before the code is assembled. We'll use `T_SYSCALL` instead of 64 in the code above, and `SYS_fork` (or its equivalent for each system call) for the system call number. We'll have to replace the part after the underscore in `SYS_` with the name of the system call function; we can do that with the token-pasting operator `##`, which glues two tokens together to form a single token. Also, macros must be defined on a single line, so we'll escape the newline characters with `\` and end each assembly line with a semicolon.

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
    .globl name; \
    name: \
        movl    $SYS_##name, %eax; \
        int $T_SYSCALL; \
        ret
```

Now we can just invoke the macro on the name of each function we want to create:

```
# ...
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
# and so on ...
```

After the preprocessor runs on the file, the result will look like this:

```
.globl fork
fork:
    movl    $1, %eax
    int     $64
    ret

.globl exit
exit:
    movl    $2, %eax
    int     $64
    ret

.globl wait
wait:
    movl    $3, %eax
    int     $64
    ret

# and so on...
```

Great! Now we have 21 functions for the system calls, all written in assembly. All user programs for xv6 will be compiled together with the code for these functions: see `ULIB` in the Makefile. So now, a user program can execute a system call by calling these functions, e.g. `fork()`.

Summary

After all the preparations are handled by the trap handler functions in the IDT, `alltraps()`, and `trap()`, system calls get routed to the `syscall()` function, which uses a system call number to pick the right function out of an array. That function will have to recover any arguments to the system call before passing it on to the real system call function later on.

Next up, we'll take a look at some of those system calls; we'll leave the rest until after we go over xv6's file system.

It's a Trap!

The last post introduced the mechanisms that xv6 uses for scheduling and context switches. User processes can transfer control to kernel code with system calls, potentially switching into the scheduler with `sleep()` or `exit()` to find another process to run. But there are many other system calls besides those two Kernel code can also be invoked during hardware interrupts or software exceptions; these three together are collectively referred to as traps.

We'll go over traps now to understand them more generally. First, about the terminology: depending on the source, interrupts might mean hardware interrupts specifically or any trap generally; similarly, exceptions might mean errors arising from the code, or traps in general. It's super frustrating because it makes it really hard to know what's meant by a word like "interrupt" or "exception" in whatever specification or source you happen to be reading. So I'm gonna try my best to save you that kind of pain in this post by sticking to "interrupt" for the hardware interrupts only, "exception" for software errors, and "trap" for those two combined with system calls.

Interrupt Descriptor Table

Imagine if, after every single time some user code carried out a division, the processor stopped, context switched into the kernel, and asked the kernel to check if there was a division by zero and handle it if necessary. Or every time a hardware interrupt happened, the kernel had to start polling all the devices to figure out which one just yelled. No. Just no. Running kernel code for all this would be way too slow.

So it's the processor that will have to detect traps and decide how to handle them. But what exactly it should do for a specific trap depends on all kinds of of particulars about that OS, e.g. a disk saying it's done reading from a file might require updating some file system data or storing the disk data in a specific buffer or something. That's too much responsibility for the processor.

Okay, so the kernel will set up a bunch of handler functions for every possible type of trap. Then it tells the hardware, "Okay, so if you get a disk interrupt, here are my instructions to handle that. For timer interrupts, use these instructions. If a process tries to access an invalid page, do this..." From then on, the processor can handle the traps without further input from the kernel by looking up the interrupt number in a big table to get the trap handler function that the kernel set up, then just running it.

In the x86 architecture, that table is called the *interrupt descriptor table* or IDT. I know, I'm sorry, I promised I'd say "trap" for the general case, but the x86 specs give it the official name of IDT even though it handles all the traps. Sigh. It has 256 entries (so that's the maximum number of distinct traps we can define); each one specifies a segment descriptor (ugh segmentation again, you know what that means: opaque code) and an instruction pointer (`%eip`) that tell the processor where it can find the corresponding trap handler function.

xv6 won't use all 256 entries; it'll mostly use trap numbers 0-31 (software exceptions), 32-63 (hardware interrupts), and 64 (system calls), all defined in traps.h (https://github.com/mit-pdos/xv6-public/blob/master/traps.h). But we do have to stick all 256 in the IDT anway, so we're the unlucky fools who get to write 256 functions' worth of assembly code by hand. Nah, just kidding: xv6 uses a script in a high-level language to do that for us and spit out the entries into an assembly file.

Unfortunately for us, that high-level language is Perl. Sigh. Perl is infamous as a "write-only" language, so I guess instead we're just the unlucky fools who get to try reading Perl.

vectors.pl

Okay, I'm not gonna assume you know Perl, and either way I really don't wanna go over every single line of this file. The syntax is similar enough to C's (except that somehow they managed to make it even *worse* than C), so you can read it on your own if you want.

Now, no script will be able to generate 256 completely unique assembly functions with enough detail to handle each trap correctly, so each function in the script has to be pretty generic. They're all gonna call the same assembly helper function, which will call a C function where we can more comfortably code up how to handle each interrupt.

The gist of this Perl script is that it prints a bunch of stuff using a for loop with 256 iterations. The xv6 Makefile (https://github.com/mit-pdos/xv6-public/blob/master/Makefile) will run it from the command line with `./vectors.pl > vectors.S` so that the output gets saved in an assembly file, which will then get assembled together with all the other kernel code in `OBJS`.

The resulting assembly file will look like this:

```
.globl alltraps

.globl vector0
vector0:
    pushl   $0
    pushl   $0
    jmp     alltraps

.globl vector1
vector1:
    pushl   $0
    pushl   $1
    jmp     alltraps

.globl vector2
vector2:
    pushl   $0
    pushl   $2
    jmp     alltraps

# ...
```

Except that a handful of entries (8, 10 through 14, and 17) will skip one line (I'll explain why below):

```
# ...

.globl vector8
vector8:
    pushl   $8
    jmp     alltraps

# ...
```

Then at the end, it defines an array `vectors` with each of those entries above:

```
# ...

.data
.globl vectors

vectors:
    .long vector0
    .long vector1
    .long vector2
    # ...
```

Okay, so those are all the handler functions; the `vectors` array holds a pointer to each one. They're all more or less the same: most of them push zero onto the stack, then all they push a *trap number* to indicate which trap just happened, and then they jump to a point in the code called `alltraps`; that's the assembly helper function I mentioned earlier.

A handful of the entries don't push zero on the stack: these are trap numbers 8 (a double fault, which happens when the processor encounters an error while handling another trap), 10 (an invalid task state segment), 11 (segment not present), 12 (a stack exception), 13 (a general protection fault), 14 (a page fault), and 17 (an alignment check). These are special because the processor will actually push an error code on the stack before calling into the corresponding handler function in `vectors`. It doesn't push any error codes on the stack for the others, so we just push 0 ourselves to make them all match up.

trapasm.S

alltraps

The processor needs to run the trap handler in kernel mode, which means we have to save some state for the process that's currently running so we can return to it later (similar to the `struct context` we saw before), then set things up to run in kernel mode. The `alltraps` routine does just that.

Remember how we said the IDT holds segment selectors for `%cs` and `%ss`, plus and instruction pointer `%eip`? (I know we haven't seen the code to create the IDT and store the entries of `vectors` in it yet; we'll get to that below.) The processor will start using those segments (and save the old ones) before running the trap handler function. Each trap handler function in `vectors` above pushed an error code (or 0) followed by a trap number. Now we have to push all the other segment selectors on the stack one at a time, then push all the general- purpose registers at once with the x86 instruction `pushal`.

```
.globl alltraps
alltraps:
    pushl   %ds
    pushl   %es
    pushl   %fs
    pushl   %gs
    pushal

    # ...
```

Cool, all the registers are saved now. So now we'll set up the `%ds` and `%es` registers for kernel mode (`%cs` and `%ss` were already done by the processor).

```
    # ...
    movw    $(SEG_KDATA<<3), %ax
    movw    %ax, %ds
    movw    %ax, %es
    # ...
```

Now we're ready to call the C function `trap()` that's gonna do most of the work. That function expects a single argument: a pointer to the process's saved register contents. Well, we just pushed them all on the stack, so we can just use `%esp` as that pointer.

```
    # ...
    pushl   %esp
    call    trap
    # ...
```

That function will return back here when it's done, so let's ignore the return value by moving the stack pointer just above it (essentially popping it off the stack).

```
    # ...
    addl    $4, %esp
    # ...
```

trapret

We've talked about this function before; when we create a new process, it starts executing in `forkret()`, which then returns into `trapret()`. More generally, any call to `trap()` will return here as well.

This function just restores everything back to where it was before, popping stored registers off the stack in reverse order. We can skip the trap number and error code; we won't need them anymore. Then we use the `iret` or "interrupt return" (though you should read that as "trap return") instruction to close out, return to user mode, and start executing the process's instructions again.

```
.globl trapret
trapret:
    popal
    popl    %gs
    popl    %fs
    popl    %es
    popl    %ds
    addl    $0x8, %esp  # skip the trap number and error code
    iret
```

trap.c

Okay, on to the main part of the code! We have to do two things here: stick the trap handler functions in `vectors` into an IDT, and figure out what to do with each interrupt type.

At the top, we've got four global variables. The IDT is represented as an array of `struct gatedesc`s, defined in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h). It's worth taking a look at because it uses an obscure C feature (bit fields); we'll do that in the next section.

Then we declare the `vectors` array of trap handler (with an `extern` keyword, since it's defined in an assembly file), a global counter `ticks` that tracks the number of timer interrupts so far (basically a rough timer), and a lock to use with `ticks`.

```
struct gatedesc idt[256];
extern uint vectors[];
struct spinlock tickslock;
uint ticks;
// ...
```

Bit Fields

This section will get *deep* into the weeds, so feel free to skip it if you're having a nice day and don't want to spoil it by reading about a bunch of C standards.

So far, we've used bit flags with regular integers by manually doing some bit arithmetic to set one bit at a time. For example, the flags for page table and page directory entries are defined as powers of 2 (e.g., `PTE_P` is 0x1, `PTE_W` is 0x2, `PTE_U` is 0x4, etc.) so that we can set a specific bit using a bitwise- OR like `pte |= PTE_U` or test whether it's set

with a bitwise-AND like `pte & PTE_P`.

But sometimes that can get annoying and hard to keep track of; wouldn't it be nice if we could just have variables that represent a single bit? Or two bits, or any number of bits we want?

The trouble is that most computer architectures don't work with a single bit at a time; they operate on bytes, words (2 bytes), long/double words (4 bytes), or quad words (8 bytes), so it would be nontrivial to compile a line of C like `a = 1` if `a` is a nonstandard size.

In fact, accessing variables that aren't aligned to a standard size (4 bytes on x86 or 8 bytes on x86_64) is much slower than when they are aligned. Compilers often optimize code to correct for this by padding `struct`s so that they'll line up along those standard sizes. For example, one like

```
struct nopadding {
    int n;
};
```

is probably left the same on x86, but one like this:

```
struct padding {
    char a;
    int n;
    char b;
};
```

is probably converted by the compiler into this:

```
struct padding {
    char a;
    char pad0[3];
    int n;
    char b;
    char pad1[3];
};
```

WARNING: We're entering the dark arts of C's unspecified and implementation- defined behavior here. Note that these are different from *undefined* behavior: undefined behavior means you did something BAD BAD BAD like dereferencing a null pointer, freeing a memory region twice, using a variable after freeing it, accessing an out-of-bounds index in a buffer, or overflowing a signed data type. Implementation-defined and unspecified behavior aren't as dangerous as undefined behavior is, but they can cause portability issues.

The C standard is a huge document with a bunch of legalese rules about what makes C, well, C. People who write C compilers need to know exactly how C code should behave under all kinds of different circumstances, so the C standard spells most of it out. But there are some parts it intentionally leaves out.

Implementation-defined behavior means the C standard doesn't set any fixed requirements about how a compiler should handle some behavior or feature; the developers of a C compiler get to decide how to write that part of the code with total freedom. One example is the number of bits in a byte; we've been assuming it's 8, but there are some (dumb) architectures where it's different.

Unspecified behavior, on the other hand, means that the C Standard provides some specific options, and compiler developers have to choose from those options for *each instance* of the behavior in the code they're compiling (that means, don't assume it's always gonna be the same, even with the same compiler).

Structure padding is implementation-defined, and there are often implementation- defined ways to modify it or disable it altogether (i.e., to *pack* the `struct` instead of *padding* it), usually with stuff like `__attribute__`s or `#pragma` directives for the preprocessor.

Wait weren't we gonna talk about bit manipulation? Why are we talking about `struct`s? Well, C does have a workaround to make bit manipulation a little easier by avoiding that slightly-annoying bit arithmetic you have to do to set or clear flags in an `int` or `unsigned int`: it's called a *bit field*, and it takes advantage of `struct` padding.

You can specify the number of bits that a field of a `struct` should occupy by adding a colon and a size after the field name:

```
struct bitfield_example {
    unsigned char a : 1;
    unsigned char b : 7;
};
```

This way, you can set the single-bit flag `a` with simple variable assignments like `var.a = 1`, and the compiler will figure out any necessary magic similar to structure padding to make that happen. Awesome, right? So why haven't we been using it all the time instead of all that opaque bit arithmetic with arcane operators like `<<`, `>>`, `|`, and `&`?

Well, there are some big downsides to bit fields. First, the C standard sets some strict rules on their use to make sure that compilers can figure out how to handle them. Bit fields are only allowed inside of structures. You're not allowed to create arrays of bit fields or pointers to bit fields. Functions aren't allowed to return a bit field. You're not allowed to get the address of a bit field with the `&` operator. You can only operate on a single bit field at a time in any statement; that means you can't set one bit field to equal another, and you can't compare the values of two bit fields.

Second, they're *extremely* implementation-defined. Each implementation (read: compiler + architecture combo) determines what data types and sizes are allowed to be used in bit fields. The data types you *can* use might have different signedness rules from the usual ones for signed and unsigned types. How they're laid out, ordered, and padded in memory can differ. In short: the low-level details are a total black box that you can probably only figure out by reading *deep* into the compiler's specifications.

Now imagine trying to do something that requires specific protocols like sending data over a network, and you come across a bit field. Lolwut. Who knows what you'd have to do. Bit fields make it impossible to port your code.

BUT! Bit arithmetic is annoying, so let's use bit fields anyway!

Okay, so back to `struct gatedesc`. IDT entries have to contain a 16-bit code segment selector (`%cs`), 16 low bits and 16 high bits for an offset in that segment, the number of arguments for the handler function, a type, a system/ application flag, a descriptor privilege level (0 for kernel, 3 for user), and a "present" flag. And x86 is very particular about how it's all laid out, so we have to set up `struct gatedesc` in the exact right order.

```
struct gatedesc {
    uint off_15_0 : 16;
    uint cs : 16;
    uint args : 5;
    uint rsv1 : 3;
    uint type : 4;
    uint s : 1;
    uint dpl : 2;
    uint p : 1;
    uint off_31_16 : 16;
};
```

Well, okay, that's it for now.

tvinit

This function loads all the assembly trap handler functions in `vectors` into the IDT. The `SETGATE()` macro in mmu.h (https://github.com/mit-pdos/xv6-public/blob/master/mmu.h) will organize each entry correctly. We said before that the IDT needs a code segment selector, an instruction pointer (from `vectors`), and a privilege level (0 for kernel mode), so we'll stick those in.

```
void tvinit(void)
{
    for (int i = 0; i < 256; i++) {
        SETGATE(idt[i], 0, SEG_KCODE << 3, vectors[i], 0);
    }
    // ...
}
```

We're basically done now, but there's one last hiccup: user code needs to be able to generate system calls, but we just set all the privilege levels so only the kernel and processor can generate traps. So we'll fix the entry for system calls as a special case.

```
void tvinit(void)
{
    // ...
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE << 3, vectors[T_SYSCALL], DPL_USER);
    // ...
}
```

Oh and while we're at it, let's just go ahead and initialize the lock for the tick counter.

```
void tvinit(void)
{
    // ...
    initlock(&tickslock, "time");
}
```

idtinit

The last function stored all the trap vectors in the IDT, so now we need to tell the processor where to find the IDT. There's a special assembly instruction for that in x86 called `lidt`.

```
void idtinit(void)
{
    lidt(idt, sizeof(idt));
}
```

trap

This last function is the one that gets called by the assembly code in `alltraps`; it's responsible for figuring out what to do based on the trap number we pushed on the stack before. Heads up: it's gonna do that by calling a bunch of other functions, many of which we haven't seen yet. I'll just give a quick summary when we come across them, and we'll get to them later on.

The only argument is a pointer to a `struct trapframe`. Wait, hang on. Up above in the assembly code, the argument we pushed on the stack was `%esp`, the stack pointer, not a pointer to any `struct trapframe`. What's up with that? Did we pass the wrong kind of argument in?

Let's check out the definition for `struct trapframe`, found in x86.h (https://github.com/mit-pdos/xv6-public/blob/master/x86.h). It's got a bunch of fields, starting off with the general purpose registers (those are the fields from `%edi` to `%eax`). Then it has four segment registers (fields `%gs` through `%ds`), plus some unused padding bits in between them to round the 16- bit segment registers up to 32 bits. The next two fields are a trap number and an error code.

All that should sound familiar. Take another look at trapasm.S (https://github.com/mit-pdos/xv6-public/blob/master/trapasm.S): so far, those are the exact same things we pushed on the stack! The other fields are what the processor pushed on the stack before calling the handler function in the IDT. So basically, we're never gonna construct a `struct trapframe` in C code; we already constructed it manually in assembly. It just describes everything that's already on the stack by the time this `trap()` function gets called. In that sense, the `%esp` we pushed as an argument really *is* a pointer to a `struct trapframe`. It's a clever way to read values off the stack.

So we said we're gonna check the trap number and decide which kernel function to call based on that, right? Let's start by checking if the trap number indicates this is a system call (trap number 64, or `T_SYSCALL`).

```
void trap(struct trapframe *tf)
{
    if (tf->trapno == T_SYSCALL) {
        // ...
    }
    // ...
}
```

Well how should we handle system calls? xv6 will have several, and we don't even know what they all are yet. So let's procrastinate again and just call some other function `syscall()` to handle the work of figuring out which system call to execute. Now we'll store the pointer to the `struct trapframe` in that process's `struct proc`, obtained with a call to `myproc()`. Also, processes need to be killed once they're done, or if they cause an exception; that happens by setting a `killed` flag in the `struct proc`. So we'll check for that before and after carrying out the system call and close the process out with `exit()` if it's due to be killed.

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

Okay, now we have all the other trap numbers to think about. We could do them with a ton of `if` statements, but that would be a pain; we'll use a `switch` statement instead. If you haven't seen `switch` statements, they replace big `if-else` blocks with cases instead. The cases can only be indexed by integers, and you have to stick a `break` statement at the end or else you'll fall through to the next case and execute the code found there as well. (To be honest, I don't see a reason why the system call case wasn't just included in this same switch statement; if you see a reason for that, let me know.)

```
void trap(struct trapframe *tf)
{
    // ...
    switch (tf->trapno) {
        // cases go here
    }
    // ...
}
```

First up is the trap number for timer interrupts; the main function of timer interrupts is to schedule a new process, but that will come further down in this function. For now, we'll just increment the `ticks` counter then call `wakeup()`, which checks if any processes went to sleep until the next tick; it'll switch to running any process it finds. There's one detail to deal with here: the system may have multiple processors, each with their own timer and interrupts. We want to use the `ticks` counter as a rough timer, but we don't know whether all the CPU timers will be synchronized, so we'll only update `ticks` using the first CPU to avoid those issues.

If you read the post on interrupt controllers then you'll be familiar with `lapiceoi()`; if you didn't (or you forgot), it just tells the local interrupt controller that we've read and acknowledged the current interrupt so it can clear it and get ready for more interrupts.

```
void trap(struct trapframe *tf)
{
    // ...
    switch (tf->trapno) {
        case T_IRQ0 + IRQ_TIMER:
            if (cpuid() == 0) {
                acquire(&tickslock);
                ticks++;
                wakeup(&ticks);
                release(&tickslock);
            }
            lapiceoi();
            break;
        // ...
    }
    // ...
}
```

Later on, we'll see some interrupt handler functions for various devices: `ideintr()` handles disk interrupts, `kbdintr()` for key presses and releases, and `uartintr()` for serial port data. We'll direct the corresponding interrupts to those functions, then acknowledge and clear them with `lapiceoi()`. Also, devices occasionally generate spurious interrupts due to hardware malfunctions; we'll either ignore them (if they're coming from the Bochs emulator) or print a message about it to the console.

```
void trap(struct trapframe *tf)
{
    // ...
    switch (tf->trapno) {
        // ...
        case T_IRQ0 + IRQ_IDE:      // disk interrupt
            ideintr();
            lapiceoi();
            break;
        case T_IRQ0 + IRQ_IDE + 1:  // spurious Bochs disk interrupt
            break;
        case T_IRQ0 + IRQ_KBD:      // keyboard interrupt
            kbdintr();
            lapiceoi();
            break;
        case T_IRQ + 7:             // spurious interrupt-no break, FALL THROUGH
        case T_IRQ + IRQ_SPURIOUS:  // spurious interrupt
            cprintf("cpu%d: spurious interrupt at %x:%x\n",
                    cpuid(), tf->cs, tf->eip);
            lapiceoi();
            break;
        // ...
    }
    // ...
}
```

Okay, so now we've dealt with system calls and hardware interrupts, so any other trap must be a software exception. `switch` statements allow a catch-all case with `default`, so we'll use that to catch the rest of the trap numbers. Now, this may have come from a kernel error or a misbehaving user process. We can check with `myproc()`, which returns a null pointer if we were running kernel code or a pointer to a `struct proc` if we were in user space, or by checking the current privilege level in the code segment selector. Depending on the source, we'll print out an appropriate error message and either panic (if in the kernel) or mark the process so it gets killed soon.

```
void trap(struct trapframe *tf)
{
    // ...
    switch (tf->trapno) {
        // ...
        default:
            if (myproc() == 0 || (tf->cs & 3) == 0) {
                // Kernel code exception
                cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
                        tf->trapno, cpuid(), tf->eip, rcr2());
                panic("trap");
            }
            // User process exception
            cprintf("pid %d %s: trap %d err %d on cpu %d "
                    "eip 0x%x addr 0x%x--kill proc\n",
                    myproc()->pid, myproc()->name, tf->trapno,
                    tf->err, cpuid(), tf->eip, rcr2());
            myproc()->killed = 1;
    }
    // ...
}
```

The reason we don't kill it immediately is because the process might be executing some kernel code right now; for example, system calls allow other interrupts and exceptions to occur while they're being handled. Killing it now might corrupt whatever it's doing. So instead we just give it the kiss of death for now and come back to finish the job later.

So next up, we'll check if this trap was generated by a user process that's due to be killed, and that process is running in ring 3. If so, we finally do the deed with `exit()`; otherwise if it's running in ring 0, it'll live for now and get killed the next time it generates a trap instead.

```
void trap(struct trapframe *tf)
{
    // ...
    if (myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER) {
        exit();
    }
    // ...
}
```

Up above, the only thing a timer interrupt did was increment `ticks`. But we know a really important function of timer interrupts is to force a process to let go of the CPU and let someone else run. It's time to do that. We'll check if the process's state is `RUNNING` and the trap was a timer interrupt; if so, we call `yield()` to let another process get scheduled on this CPU.

```
void trap(struct trapframe *tf)
{
    // ...
    if (myproc() && myproc()->state == RUNNING &&
            tf->trapno == T_IRQ0 + IRQ_TIMER) {
        yield();
    }
    // ...
}
```

Now we have one last check: a process that yielded, then got picked up again later might have been marked as killed in the meantime, so if it was, we need to finish it off now. So we do the exact same check as above again, and then we're done.

```
void trap(struct trapframe *tf)
{
    // ...
    if (myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER) {
        exit();
    }
}
```

Note that this function will return into `trapret` in the assembly code, which will then send it back to user mode.

Summary

Let's take a moment to assess how much of xv6 we've already covered. Remember, the xv6 kernel has four main functions: (1) finishing the boot process that the boot loader started, (2) virtualizing resources in order to isolate processes from each other, (3) scheduling processes to run, and (4) interfacing between user processes and hardware devices. Let's take that as a checklist and go through those items now.

We've already seen some of the initialization routines that get run on boot in `main()`; most of the code there sets up virtual memory and all the hardware devices. We still have a few more devices to talk about: the keyboard, serial port, console, and disk; each of those has its own boot function that we'll need to go over in order to wrap up point (1).

On the other hand, we're already done with (2) and (3): we spent a lot of time going over virtual memory and paging, and the last post on scheduling showed us how xv6 virtualizes the CPU as well as it runs processes.

The code we saw in this post was our introduction to point (4). Traps are the primary mechanism for user processes to communicate with the hardware; the kernel coordinates that communication by setting up trap handler functions. The code we've seen here basically acts like an usher, directing traps to the right trap handler function depending on its type.

When a trap occurs (x86 instruction `int`), the processor will stop executing code, find the IDT, and looks up the entry for that trap number. The script that xv6 uses to generate the IDT entries just makes them all point to the same function `alltraps()`, which saves all the process's registers, switches into kernel mode, and calls `trap()`. Then that function uses the trap number to figure out how the kernel wants it to respond to this particular trap. So any hardware interrupt, software exception, or user system call will get funneled into the functions here before getting dispatched to some other appropriate kernel code that will know what to do with it.

We haven't finished point (4) yet, though: we have to actually see what each of those trap handler functions does. But we did see some of them: for example, we saw that a software exception either kills the process that caused it or panics if it occurred in kernel code. That already takes care of one of the three types of traps, so we're left with hardware interrupts and system calls. All the system calls got redirected to a `syscall()` function which we haven't seen yet.

We have seen how some of the hardware interrupts are dealt with: a timer interrupt increments a `ticks` counter (if it's on CPU 0), then calls `yield()` to force a process to give up the CPU until the next scheduling round. Spurious interrupts either get ignored or print a message to the console. But we've procrastinated some of the others: disk interrupts call an `ideintr()` function to handle them, keyboard interrupts call `kdbintr()`, and serial port interrupts call `uartintr()`, none of which we've gone over.

So in order to wrap up the xv6 kernel, we still have to understand how system calls are routed in general, as well as how devices are initialized at boot and how the kernel responds to specific system calls that require use of those devices. The general system call routing mechanism is up next.