ICS143A: Principles of Operating Systems

Final recap, sample questions

Anton Burtsev
March, 2017
What is operating system?
What is a process?
What mechanisms are involved into implementation of a process?
Describe the x86 address translation pipeline (draw figure), explain stages.
Address translation

CPU

Selector
Offset
Logical Address

Segment Translation

Linear Address

Page Translation

Physical Address

x GB
RAM
What is the linear address? What address is in the registers, e.g., in %eax?
Logical and linear addresses

- Segment selector (16 bit) + offset (32 bit)
What segments do the following instructions use? push, jump, mov
What's on the stack? Describe layout of a stack and how it changes during function invocation?
Example stack

| 10 | [ebp + 16] (3rd function argument) |
|  5 | [ebp + 12] (2nd argument)        |
|  2 | [ebp + 8]  (1st argument)        |
| RA | [ebp + 4]  (return address)      |
| FP | [ebp]      (old ebp value)       |
|    | [ebp - 4]  (1st local variable)  |
|    | [ebp - X]  (esp - the current stack pointer) |
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)
Interrupt path

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- EIP

Last stack frame

Process
- Argument 1
- Argument 2
- Calling EIP ++
- Old EBP
- Local variables
- Saved local values, e.g. push EAX, etc

User stack of a process (can grow up to 2GBs)

Code, data, heap

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel Stack of a process (4K)

GDT
- NULL: 0x0
- KCODE: 0 - 4GB
- KDATA: 0 - 4GB
- K_CPU: 4 bytes
- CODE: 0 - 4GB
- DATA: 0 - 4GB
- TSS: sizeof(tss)

IDT
- CS: HANDLER ADDR
- ...
- ...

TSS
- SS:
- EFLAGS:
- ESP:
- ...

Page table
- Level 1
- 0 - 4MB
- 4 - 4MB
- ...
- 2GB - 2GB + 4MB

Level 2

Kernel code

vector32

CPU register:
- CS: #1
- SS: #2
- EIP: <kernel>
- GDT: gdt
- TSS: tss
- IDT: idt
- CR3: pt
Which stack is used for execution of an interrupt handler? How does hardware find it?
Interrupt path

User stack of a process (can grow up to 2GBs)

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel code
Describe organization of the memory allocator in xv6?
Physical page allocator

Virtual

end (0x801126fc)
kmem.freelist

2GB + 4MB
0x80400000

0x80000000

Protected Mode

CS : 0x8  EIP: main
SS : 0x10  ESP: stack
GDT: 0x7c78  TSS: 0x0
IDT: 0x0  CR3: entrypgdir
Where did free memory came from?
How do we switch between processes?
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx

2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi

2967
2968 # Switch stacksh
2969 movl %esp, (%eax)
2970 movl %edx, %esp

2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret

swtch()

2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
Stack inside `swtch()`

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- CS
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP
- EIP (alltraps)
- ...
- EIP (trap)
- ...
- EIP (yield)
- ...
- &proc->context
- cpu->scheduler
- EIP (sched)

Kernel Stack of a process (4K)

Trap frame

Call stack:
- vector32()
- alltraps()
- trap()
- yield()
- sched()
- switch(&proc->context, cpu->scheduler)

Context
- EIP (line: 2479)
- EBP
- EBX
- ESI
- EDI
What is the interface between the kernel and user-level processes?
void syscall(void) {
    int num;
    num = proc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        proc->tf->eax = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n",
                 proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}
void syscall(void)
{
    int num;

    num = proc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // proc->tf->eax = syscalls[num]();
        proc->tf->esp -= 4;
        *(int*)ptoc->tf->esp = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d
", proc->pid, proc->name, num);
        // proc->tf->eax = -1;
        proc->tf->esp -= 4;
        *(int*)ptoc->tf->esp = -1;
    }
}

Why do we need locks?
Request queue (e.g. incoming network packets)

• Linked list, list is pointer to the first element
CPU 1 and 2 allocate new request
CPU 1 and 2 update
next pointer

l->next = list

l->next = list
CPUs 1 updates
head pointer

list = l
CPU2 updates head

list = l
State after the race
```
insert(int data) {
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}
```

- Critical section
Correct implementation

1573 void
1574 acquire(struct spinlock *lk)
1575 {
...  
1580    // The xchg is atomic.
1581    while(xchg(&lk->locked, 1) != 0)
1582        ;
...  
1592 }
Xchg instruction

• Swap a word in memory with a new value
  • Atomic!
  • Return old value
Deadlocks

```
acquire(A)  ←

acquire(B) {  
    while(xchg(&B->locked, 1) != 0) 
}  -->

acquire(A) {  
    while(xchg(&A->locked, 1) != 0) 
}  ←

acquire(B)
```
Lock ordering

• Locks need to be acquired in the same order
Locks and interrupts

network_packet()

....
insert() {
  acquire(A)
}

Network interrupt

network_packet()

....
insert() {
  acquire(A)
}

...
Locks and interrupts

- Never hold a lock with interrupts enabled
Send/receive queue

201 void *
202 send(struct q *q, void *p)
203 {
204   while(q->ptr != 0)
205     ;
206   q->ptr = p;
207   wakeup(q); /*wake recv*/
208 }

210 void *
211 recv(struct q *q)
212 {
213   void *p;
214
215   while((p = q->ptr) == 0)
216     sleep(q);
217   q->ptr = 0;
218   return p;
219 }
Lost wakeup problem

recv

215

216 wait for wakeup forever

test

sleep

send

206

207

204

205

store p

wakeup

test

spin forever

Time
The role of file systems

- Sharing
  - Sharing of data across users and applications
- Persistence
  - Data is available after reboot
Architecture

• On-disk and in-memory data structures represent
  • The tree of named files and directories
  • Record identities of disk blocks which hold data for each file
  • Record which areas of the disk are free
Crash recovery

- File systems must support crash recovery
  - A power loss may interrupt a sequence of updates
  - Leave file system in inconsistent state
    - E.g. a block both marked free and used
Multiple users

- Multiple users operate on a file system concurrently
  - File system must maintain invariants
Speed

• Access to a block device is several orders of magnitude slower
  • Memory: 200 cycles
  • Disk: 20,000,000 cycles

• A file system must maintain a cache of disk blocks in memory
Block layer

- Read and write data
- From a block device
- Into a buffer cache
- Synchronize across multiple readers and writers
## Transactions

- Group multiple writes into an atomic transaction

<table>
<thead>
<tr>
<th>System calls</th>
<th>File descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathnames</td>
<td>Recursive lookup</td>
</tr>
<tr>
<td>Directories</td>
<td>Directory inodes</td>
</tr>
<tr>
<td>Files</td>
<td>Inodes and block allocator</td>
</tr>
<tr>
<td>Transactions</td>
<td>Logging</td>
</tr>
<tr>
<td>Blocks</td>
<td>Buffer cache</td>
</tr>
</tbody>
</table>
Files

- Unnamed files
- Represented as inodes
- Sequence of blocks holding file's data

<table>
<thead>
<tr>
<th>System calls</th>
<th>File descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathnames</td>
<td>Recursive lookup</td>
</tr>
<tr>
<td>Directories</td>
<td>Directory inodes</td>
</tr>
<tr>
<td>Files</td>
<td>Inodes and block allocator</td>
</tr>
<tr>
<td>Transactions</td>
<td>Logging</td>
</tr>
<tr>
<td>Blocks</td>
<td>Buffer cache</td>
</tr>
</tbody>
</table>
Directories

- Special kind of inode
- Sequence of directory entries
- Each contains name and a pointer to an unnamed inode

<table>
<thead>
<tr>
<th>System calls</th>
<th>File descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathnames</td>
<td>Recursive lookup</td>
</tr>
<tr>
<td>Directories</td>
<td>Directory inodes</td>
</tr>
<tr>
<td>Files</td>
<td>Inodes and block allocator</td>
</tr>
<tr>
<td>Transactions</td>
<td>Logging</td>
</tr>
<tr>
<td>Blocks</td>
<td>Buffer cache</td>
</tr>
</tbody>
</table>
Pathnames

- Hierarchical path names
  - `/usr/bin/sh`
  - Recursive lookup

<table>
<thead>
<tr>
<th>System calls</th>
<th>File descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathnames</td>
<td>Recursive lookup</td>
</tr>
<tr>
<td>Directories</td>
<td>Directory inodes</td>
</tr>
<tr>
<td>Files</td>
<td>Inodes and block allocator</td>
</tr>
<tr>
<td>Transactions</td>
<td>Logging</td>
</tr>
<tr>
<td>Blocks</td>
<td>Buffer cache</td>
</tr>
</tbody>
</table>
System call

- Abstract UNIX resources as files
  - Files, sockets, devices, pipes, etc.
- Unified programming interface
File system layout on disk

- Block #0: Boot code
- Block #1: Metadata about the file system
  - Size (number of blocks)
  - Number of data blocks
  - Number of inodes
  - Number of blocks in log
File system layout on disk

- Block #2 (inode area)
- Bit map area: track which blocks are in use
- Data area: actual file data
- Log area: maintaining consistency in case of a power outage or system crash
Typical use of transactions

```c
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```
Strawman scheduler (xv6)

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?
scheduler(void)
{
    for(;;){
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
            proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&cpu->scheduler, proc->context);
            switchkvm();
            proc = 0;
        }
    }
}
Strawman scheduler (xv6)

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?
Strawman scheduler

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?
  - Only allows round-robin scheduling
  - Can’t prioritize tasks
Priority based scheduling

- Higher-priority processes run first
- Processes within the same priority are round-robin
O(1) scheduler (Linux 2.6 – 2.6.22)

- Priority based scheduling
- Goal: decide who to run next, independent of number of processes in system
  - Still maintain ability to prioritize tasks, handle partially unused quanta, etc
**O(1) data structures**

- runqueue: a list of runnable processes
  - Blocked processes are not on any runqueue
  - A runqueue belongs to a specific CPU
  - Each task is on exactly one runqueue
  - Task only scheduled on runqueue’s CPU unless migrated
- $2 \times 40 \times \#\text{CPUs}$ runqueues
  - 40 dynamic priority levels (more later)
  - 2 sets of runqueues – one active and one expired
O(1) data structures (contd)
O(1) intuition

- Take the first task off the lowest-numbered runqueue on active set
  - Confusingly: a lower priority value means higher priority
- When done, put it on appropriate runqueue on expired set
- Once active is completely empty, swap which set of runqueues is active and expired
- Constant time, since fixed number of queues to check; only take first item from non-empty queue
O(1) example

**Active**

- 139
- 138
- 137
- .
- .
- 101
- 100

**Expired**

- 139
- 138
- 137
- .
- .
- 101
- 100

Pick first, highest priority task to run

Move to the expired queue
What now?

Flip active and expired queues
Blocked tasks

• What if a program blocks on I/O, say for the disk?
  • It still has part of its quantum left
  • Not runnable, so don’t waste time putting it on the active or expired runqueues

• We need a “wait queue” associated with each blockable event
  • Disk, lock, pipe, network socket, etc.
Blocking example

Active

139
138
137
.
.
101
100

Expired

139
138
137
.
.
101
100

Process goes on disk wait queue
Buddy memory allocator
Buddy allocator
What's wrong with buddy?
What's wrong with buddy?

- Buddy allocator is ok for large allocations
  - E.g. 1 page or more
- But what about small allocations?
  - Buddy uses the whole page for a 4 bytes allocation
    - Wasteful
  - Buddy is still slow for short-lived objects
Slab:
Allocator for object of a fixed size
Slab

- A 2 page slab with 6 objects
Keeping track of free objects

- kmem_bufctl array is effectively a linked list
  - First free object: 3
  - Next free object: 1
A cache is formed out of slabs
Slab is fine, but what's wrong?
Slab is fine, but what's wrong?

- We can only allocate objects of one size
Kmalloc(): variable size objects

- A table of caches
  - Size: 32, 64, 128, etc.
Thank you!