ICS143A: Principles of Operating Systems

Final recap, sample questions

Anton Burtsev
December, 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
Logical Address (or Far Pointer)

Segment Selector

Offset

Linear Address Space

Global Descriptor Table (GDT)

Segment Descriptor

Segment Base Address

Segment

Lin. Addr.

Page Directory

Entry

Page Table

Entry

Physical Address Space

Page

Phy. Addr.

Segmentation

Paging
Logical Address (or Far Pointer)

Segment Selector

Offset

Linear Address Space

Global Descriptor Table (GDT)

Segment Descriptor

Segment Base Address

Segment

Linear Address

Dir Table Offset

Page Table

Entry

Page Directory

Page

Phy. Addr.

Physical Address Space

Segmentation

Paging
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[ebp + 16] (3rd function argument)</td>
</tr>
<tr>
<td>5</td>
<td>[ebp + 12] (2nd argument)</td>
</tr>
<tr>
<td>2</td>
<td>[ebp + 8] (1st argument)</td>
</tr>
<tr>
<td>RA</td>
<td>[ebp + 4] (return address)</td>
</tr>
<tr>
<td>FP</td>
<td>[ebp] (old ebp value)</td>
</tr>
<tr>
<td></td>
<td>[ebp - 4] (1st local variable)</td>
</tr>
<tr>
<td></td>
<td>[ebp - X] (esp - the current stack pointer)</td>
</tr>
</tbody>
</table>
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)
Which stack is used for execution of an interrupt handler? How does hardware find it?
Describe organization of the memory allocator in xv6?
Where did free memory came from?
How do we switch between processes?
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

2978

struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
Stack inside `swtch()`

User state (saved by hardware)
- SS
- ESP
- EFLAGS
- CS
- CS
- 0
- 32

Kernel Stack of a process (4K)

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

Stack frame
- `Proc`
- `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->esp -= 4;
    *(int*)ptoc->tf->esp = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            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

Diagram showing a list structure with pointers connecting nodes, and two CPUs at the ends.
CPU2 updates head pointer

\[ \text{list} = 1 \]
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}
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

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

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

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

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

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

```c
network_packet(){
    ...
    insert() {
        acquire(A)
    }
}
```

Network interrupt

```c
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   while((p = q->ptr) == 0)
215     sleep(q);
216   q->ptr = 0;
217   return p;
218 }
Lost wakeup problem

recv

215 test
216 sleep
wait for wakeup forever

send

206 store p
207 wakeup
204 test
205 spin forever
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>System calls</td>
<td>File descriptors</td>
</tr>
<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
## 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

<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>
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
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;
        switchuvvm(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)

Active

139
138
137
.
.
101
100

Expired

139
138
137
.
.
101
100
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**
  - Tasks: 139, 138, 137, 101, 100
  - Pick first, highest priority task to run

- **Expired**
  - Tasks: 139, 138, 137, 101, 100
  - Move to the expired queue
What now?

Active

139
138
137

Expired

139
138
137

Flip active and expired queues

101
100

101
100
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

Process goes on disk wait queue
Buddy memory allocator
Buddy allocator

\[ 2^n \text{ pages} \]
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!