OS Concepts and structure

Today
- OS services
- OS interface to programmers/users
- OS components & interconnects
- Structuring OSs

Next time
- Processes

OS Views

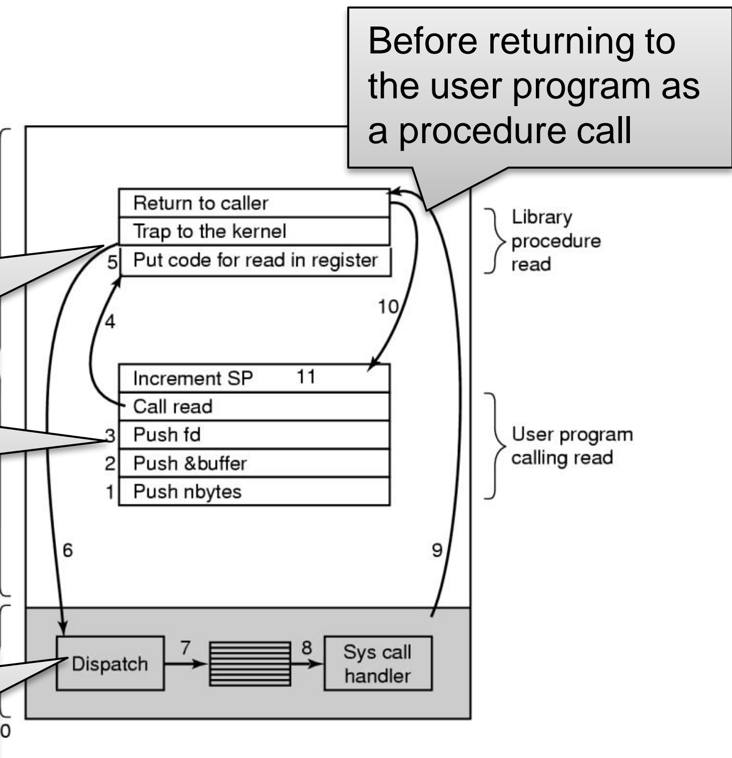
- Vantage points
  - OS as the services it provides
    - To users and applications
  - OS as its components and interactions
- OS provides a number of services
  - To users via a command interpreter/shell or GUI
  - To application programs via system calls
  - Some services are for convenience
    - Program execution, I/O operation, file system management, communication
  - Some to ensure efficient operation
    - Resource allocation, accounting, protection and security

Command interpreter (shell) & GUI

- Command interpreter
  - Handle (interpret and execute) user commands
  - Could be part of the OS: MS DOS, Apple II
  - Could be just a special program: UNIX, Windows XP
    - In this way, multiple options – shells – are possible
  - The command interpreter could
    - Implement all commands
    - Simply understand what program to invoke and how (UNIX)
- GUI
  - Friendlier, through a desktop metaphor, if sometimes limiting
  - Xerox PARK Alto >> Apple >> Windows >> Linux

System calls

- Low-level interface to services for applications
- Higher-level requests get translated into sequence of system calls
- Writing `cp` – copy source to destination
  - Get file names
  - Open source
  - Create destination
  - Loop
    - Read
    - Copy
  - Close des
  - Report co
  - Terminate



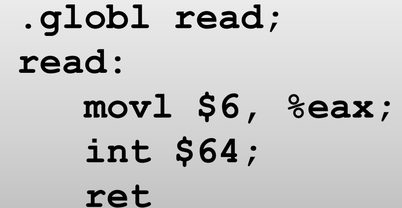
Tracing a system calls in xv6 …

- All starts at the user-level with an app calling a library routine – `read()`
- Library places arguments in relevant registers and issues a trap

```
#define SYSCALL(name) \                    xv6/usys.S
   .globl name; \
    name: \
        movl $SYS_ ## name, %eax; \
       int $T_SYSCALL; \
       ret

SYSALL(read)
```

```
#define SYS_read        6                xv6/syscall.h
```




```
// x86 trap and interrupt constants.      xv6/traps.h

// Processor-defined
#define T_DIVIDE     0   // divide error
#define T_DEBUG      1   // debug exception
…
#define T_GPFLT      13 // general protection
fault
#define T_PGFLT      14 // page fault
….
#define T_SYSCALL    64 // system call
…
```

Value placed in `%eax`, 6, is used to vector to the right system call.

The `int` instruction takes one argument, 64, which tells the hardware which trap type this is.

Other arguments are passed on the stack.

Tracing a system calls in xv6 ...

- Once `int` is executed, hardware gets to do some of the work
  - Raises the privilege level – from Current Privilege Level 3 to 0 in x86
  - Transfers control to trap vectors – as set by the OS in initialization
- To let the hardware know what code to run, the OS must make sure to tell it (at booting) the location of the code for each trap ...

xv6/main.c

```
int
main(void)
{
  …
  tvinit();
  …
}
```

xv6/trap.c

```
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);

   initlock(&tickslock, "time");
}
```

Tracing a system calls in xv6 …

xv6/mmu.h

```
// 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_{TG,IG32,TG32})
 uint s : 1;           // must be 0 (system)
 uint dpl : 2;         // descriptor(meaning new) privilege level
 uint p : 1;           // Present
 uint off_31_16 : 16; // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#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;         \
}
```

Tracing a system calls in xv6 ...

- But we haven't told the hardware yet (just filled in some data structures)
- This is also done in the boot sequence, in `mpmain()` for xv6

xv6/main.c

```
static void
mpmain(void)
{
  …
  idtinit();
  …
}
```

```
static void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

```
static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size – 1;
  pd[1] = (uint) p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

Here is where the hardware is told where to find the Interrupt Descriptor Table in memory!

Tracing a system calls in xv6 ...

- OS has set up its trap handlers, now let's see what happens when a system call is issue via the `int` instruction
- HW does some work for SW, including saving current PC, then OS runs

xv6/vector.sS; generated from vectors.pl

```
.globl vector64
vector64:
   pushl $64
   jmp alltraps
```

First code the OS runs ...
Push trap number onto the stack, and call alltraps to do most of the context saving

xv6/trapasm.S

```
.globl alltraps
alltraps:
   # Build trap frame.
   pushl %ds
   pushl %es
   push %fs
   push %gs
   pushal

   # Set up data and per-cpu segments.
   movw $(SEG_KDATA<<3), %ax
   movw %ax, %ds
   movw %ax, %es
   movw $(SEG_KCPU<<3), %ax
   movw %ax, %ds
   movw %ax, %gs

   # Call trap(tf), where tf=%esp
   pushl %esp
   call trap
   addl %4, %esp
```

... code there saves a few more segments registers onto the stack, before pushing the remaining general registers (pushal)

OS then changes descriptor segments and extra segment registers so that it can access its own (kernel) memory

Then the C trap handler is called

Tracing a system calls in xv6 ...

- Once done with the low-level details, assembly code calls the generic C trap handler

xv6/trap.c

```
void
trap(struct trapframe *tf)
{
   if(tf->trapno == T_SYSCALL){
      if(proc->killed)
         exit();
      proc->tf = tf;
      syscall();
      if(proc->killed)
         exit();
      return;
   }
   …
```

Trap handler handles all type of interrupts and traps, so first checks the trap number.

... If it's T_SYSCALL, handles the system call

... Making sure the process is still there before and after doing the syscall

Tracing a system calls in xv6 ... almost

- Finally, system call number has been pass in `%eax`
- So we use this to call the appropriate routine ...
- In our example, `sys_read()` will be call and the return value is left in `%eax`

xv6/syscall.c

```
static int (*syscalls[] (void) = {
[SYS_chdir]  sys_chdir;
[SYS_close]  sys_close;
…
[SYS_read]   sys_read;
…

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;
   }
}
```

Tracing a system calls in xv6

- And the return ... Just pop what was pushed before to restore the context of the running process and issue `iret`
- Similar to a return from procedure call, but also lowers privilege level to user mode and jumps to instruction after the `int` that invoke syscall
- Now we are back in `read()`

xv6/trapasm.S

```
# Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Yeah!!

Major OS components & abstractions

- Processes
- Memory
- I/O
- Secondary storage
- File systems
- Protection
- Accounting
- Shells & GUI
- Networking

Processes

- A program in execution
  - Address space
  - Set of registers
- To get a better sense of it
  - What data do you need to (re-) start a suspended process?
  - Where do you keep this data?
  - What is the process abstraction I/F offered by the OS
    - Create, delete, suspend, resume & clone a process
    - Inter-process communication & synchronization
    - Create/delete a child process

Call	**Description**
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution & return status

Memory management

- Main memory – the directly accessed storage for CPU
  - Programs must be stored in memory to execute
  - Memory access is fast (e.g., 60 ns to load/store)
    - but memory doesn't survive power failures
- OS must
  - Allocate memory space for programs (explicitly and implicitly)
  - Deallocate space when needed by rest of system
  - Maintain mappings from physical to virtual memory
    - e.g. through page tables
  - Decide how much memory to allocate to each process
  - Decide when to remove a process from memory

Call	**Description**
void *sbrk(intptr_t increment)	Increments program data space by 'increment' bytes

I/O

- A big chunk of the OS kernel deals with I/O
  - Hundreds of thousands of lines in NT
- The OS provides a standard interface between programs & devices
  - file system (disk), sockets (network), frame buffer (video)
- Device drivers are the routines that interact with specific device types
  - Encapsulates device-specific knowledge
    - e.g., how to initialize a device, request I/O, handle errors
  - Examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers, …

Secondary storage

- Secondary storage (disk, tape) is persistent memory
  - Often magnetic media, survives power failures (hopefully)
- Routines that interact with disks are typically at a very low level in the OS
  - Used by many components (file system, VM, …)
  - Handle scheduling of disk operations, head movement, error handling, and often management of space on disks
- Usually independent of file system
  - Although there may be cooperation
  - File system knowledge of device details can help optimize performance
    - e.g., place related files close together on disk

File systems

- Secondary storage devices are hard to work with
- File system offers a convenient abstraction
  - Defines logical abstractions/objects like files & directories
  - As well as operations on these objects
- A file is the basic unit of long-term storage
- A directory is just a special kind of file
  - … containing names of other files & metadata
- Interface:
  - File/directory creation/deletion, manipulation, copy, lock
- Other higher level services: accounting & quotas, backup, indexing or search, versioning

Some I/O related system calls

Call	Description
open(s, flags)	Open a file with mode specified in flags
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd,buf,n)	Write n bytes from an open file into fd
close(fd)	Release fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd’s in p
chdir(s)	Change directory to directory s
mkdir(s)	Create a new directory s
mknod(s, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(s1, s2)	Create another name (s2) for the file s1
unlink(s)	Remove a name

Protection

- Protection is a general mechanism used throughout the OS
  - All resources needed to be protected
    - memory
    - processes
    - files
    - devices
    - …
- Protection mechanisms help to detect and contain errors, as well as preventing malicious destruction

OS structure

- OS made of number of components
  - Process & memory management, file system, …
  - and system programs
    - e.g., bootstrap code, the init program, …
- Major design issue
  - How do we organize all this?
  - What are the modules, and where do they exist?
  - How do they interact?
- Massive software engineering
  - Design a large, complex program that:
    - performs well, is reliable, is extensible, is backwards compatible, …

OS design & implementation

- *User* goals and *System* goals
  - User – convenient to use, easy to learn, reliable, safe, fast
  - System – easy to design, implement, & maintain, also flexible, reliable, error-free & efficient
- Affected by choice of hardware, type of system
- Clearly conflicting goals, no unique solution
- Some other issues complicating this
  - Size: Windows XP ~40G SLOC, RH 7.1 17G SLOC
  - Concurrency – multiple users and multiple devices
  - Potentially hostile users, but some users want to collaborate
  - Long expected lives & no clear ideas on future needs
  - Portability and support to thousands of device drivers
  - Backward compatibility

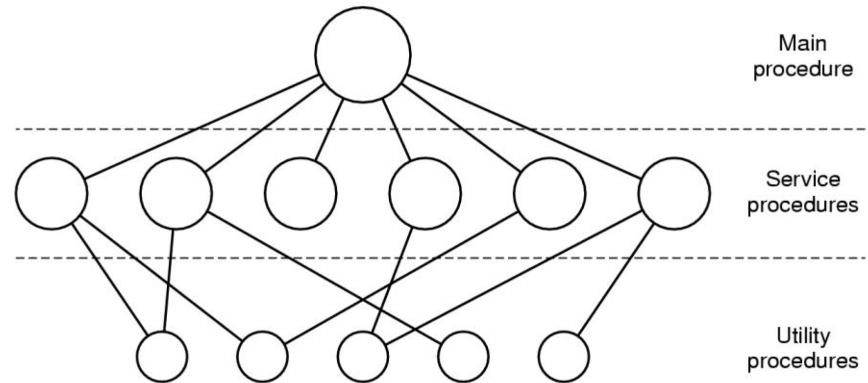
OS design & implementation

- A software engineering principle – separate policy & mechanism
  - Policy: What will be done?
  - Mechanism: How to do it?
  - Why do you care? Max flexibility, easier to change policies
- Implementation on high-level language
  - Early on – assembly (e.g. MS-DOS – 8088), later Algol (MCP), PL/1 (MULTICS), C (Unix, …)
  - Advantages – faster to write, more compact, easier to maintain & debug, easier to port
  - Cost – Size, speed?, but who cares?!

> The greater part of UNIX software is written in the above-mentioned C language [6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multi-programming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.
> …D. Ritchie and K. Thompson, The UNIX time-sharing system, CACM 17(7), July 1974

Monolithic design

- Major advantage:
  - Cost of module interactions is low (procedure call)
- Disadvantages:
  - Hard to understand
  - Hard to modify
  - Unreliable (no isolation between system modules)
  - Hard to maintain
- Alternative?
  - How to organize the OS in order to simplify its design and implementation?



Layering

- The traditional approach
  - Implement OS as a set of layers
  - Each layer shows an enhanced ‘virtual mach’ to layer above
- Each layer can be tested and verified independently

Layer	Description
5: Job managers	Execute users’ programs
4: Device managers	Handle device & provide buffering
3: Console manager	Implements virtual consoles
2: Page manager	Implements virtual memory for each process
1: Kernel	Implements a virtual processor for each process
0: Hardware	

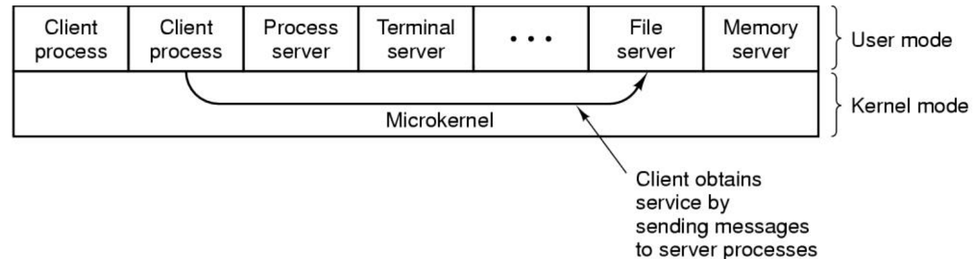
Dijkstra’s THE system

Problems with layering

- Imposes hierarchical structure
  - but real systems have complex interactions
  - Strict layering isn't flexible enough
- Poor performance
  - Each layer crossing implies overhead
- Disjunction between model and reality
  - Systems modelled as layers, but not built that way

Microkernels

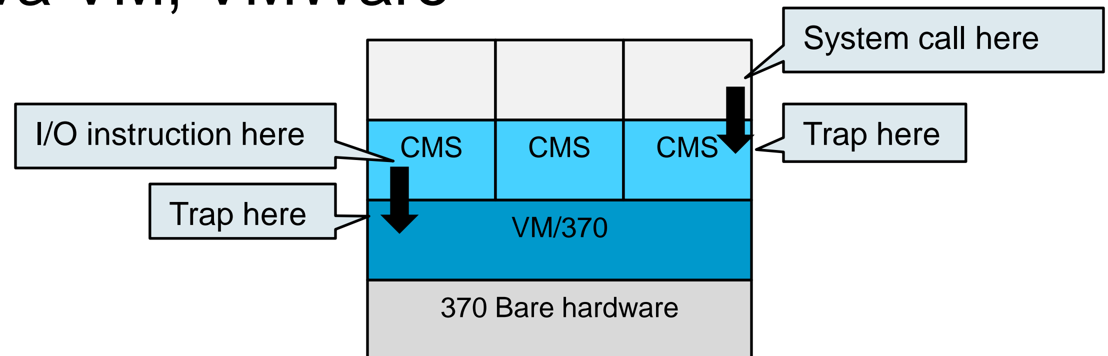
- Popular in the late 80’s, early 90’s
  – Recent resurgence
- Goal
  – Minimize what goes in kernel
  – Organize rest of OS as user-level processes



- This results in
  – Better reliability (isolation between components)
  – Ease of extension and customization
  – Poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  – … Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft), L4 (Karlsruhe), …

Virtual machines

- Initial release of OS/360 were strictly batch but users wanted timesharing
  - IBM CP/CMS, later renamed VM/370 (‘79)
- Note that timesharing systems provides (1) multiprogramming & (2) extended (virtual) machine
- Essence of VM/370 – separate the two
  - Heart of the system (VMM) does multiprogramming & provides to next layer up multiple exact copies of bare HW
  - Each VM can run any OS
- Nowadays – Java VM, VMWare



Summary & preview

- Today
  - The mess under the carpet
  - Basic concepts in OS
  - OS design has been an evolutionary process
  - Structuring OS - a few alternatives, not a clear winner
- Next …
  - Process – the central concept in OS
    - Process model and implementation
    - What it is, what it does and how it does it