Outline

• Brief history of OS’s
• Four fundamental OS concepts
  • Thread
  • Address space
  • Process
  • Dual-mode operation/protection
Very Brief History of OS

- Several distinct phases:
  - Hardware expensive, humans cheap
  - Eniac, … Multics

“I think there is a world market for maybe five computers.” – Thomas Watson, chairman of IBM, 1943

Thomas Watson was often called “the worlds greatest salesman” by the time of his death in 1956
Very Brief History of OS

• Several distinct phases:
  • Hardware expensive, humans cheap
    • Eniac, … Multics
  • Hardware cheaper, humans expensive
    • PCs, workstations, rise of GUIs
  • Hardware very cheap, humans very expensive
    • Ubiquitous devices, widespread networking
Very Brief History of OS

- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, … Multics
  - Hardware cheaper, humans expensive
    - PCs, workstations, rise of GUIs
  - Hardware very cheap, humans very expensive
    - Ubiquitous devices, widespread networking
- Rapid change in hardware leads to changing OS
  - Batch $\Rightarrow$ multiprogramming $\Rightarrow$ timesharing $\Rightarrow$ GUI $\Rightarrow$ ubiquitous devices
  - Gradual migration of features into smaller machines
- Today
  - Small OS: 100K lines / Large: 20M lines (10M browser!)
  - 100-1000 people-years
OS Archaeology

• Due to high cost of building OS from scratch, most modern OS’s have long lineage

• Multics ⇒ AT&T Unix ⇒ BSD Unix ⇒ Ultrix, SunOS, NetBSD,…

• Mach (micro-kernel) + BSD ⇒ NextStep ⇒ XNU ⇒ Apple OS X, iPhone iOS

• MINIX ⇒ Linux ⇒ Android, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,…

• CP/M ⇒ QDOS ⇒ MS-DOS ⇒ Windows 3.1 ⇒ NT ⇒ 95 ⇒ 98 ⇒ 2000 ⇒ XP ⇒ Vista ⇒ 7 ⇒ 8 ⇒ 10 ⇒ …
Today: Four Fundamental OS Concepts

• Thread
  • Single unique execution context which fully describes program state
  • Program counter, registers, execution flags, stack

• Address space (with translation)
  • Address space which is distinct from machine’s physical memory addresses

• Process
  • Instance of executing program consisting of address space and 1+ threads

• Dual-mode operation/protection
  • Only “system” can access certain resources
  • OS and hardware are protected from user programs
  • User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses
OS Bottom Line: Run Programs

- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to program”
- Provide services to program
- While protecting OS and program
Instruction Cycle:
Fetch, Decode, Execute
What Happens During Program Execution?

- Execution sequence:
  - Fetch instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/memory
  - PC ← Next(PC)
  - Repeat

Next instruction or jump to new address …
Thread (1st OS Concept)

- Thread is single unique execution context
  - Program counter (PC), registers, execution flags, stack
- Thread is executing on processor when it resides in processor's registers
- Registers hold root state of thread (the rest is “in memory”)
- Registers are defined by instruction set architecture (ISA) or by compiler
  - Stack pointer (SP) holds address of top of stack
    - Other conventions: frame pointer, heap pointer, data
  - PC register holds the address of executing instruction in the thread
Address Space (2\textsuperscript{nd} OS Concept)

• Address space is set of accessible addresses and state associated with them
  • For 32-bit processor: $2^{32} \approx 4$ billion addresses

• What happens when you read or write to address?
  • Perhaps nothing
  • Perhaps acts like regular memory
  • Perhaps ignores writes
  • Perhaps causes I/O operation
    • (Memory-mapped I/O)
  • Perhaps causes exception (fault)
Multiprogramming: Multiple Threads

OS

Proc 1  Proc 2  …  Proc n
Time Sharing

- How can we give **illusion** of multiple processors with single processor?
  - Multiplex in time!
- Each virtual “CPU” needs structure to hold
  - PC, SP, and rest of registers (integer, floating point, …)
- How do we switch from one vCPU to next?
  - Save PC, SP, and registers in current **state block**
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, …
The Basic Problem of Concurrency

- The basic problem of concurrency involves resources
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: processes think they have exclusive access to shared resources

- OS should coordinate all activity
  - Multiple processes, I/O interrupts, …
  - How can it keep all these things straight?

- Basic idea is to use virtual machine abstraction
  - Simple machine abstraction for processes
  - Multiplex these abstract machines

- Dijkstra did this for the “THE system”
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)
Properties of This Simple Multiprogramming Technique

• All vCPUs share same non-CPU resources
  • I/O devices, memory, …

• Consequence of sharing
  • Each thread can access data of every other thread
    (good for sharing, bad for protection)
  • Threads can share instructions
    (good for sharing, bad for protection)
  • Can threads overwrite OS functions?

• This (unprotected) model is common in
  • Embedded applications
  • Windows 3.1/Early Macintosh (switch only with yield)
  • Windows 95-ME (switch with both yield and timer)
Protection

• OS must protect itself from user programs
  • Reliability: compromising OS generally causes it to crash
  • Security: limit scope of what processes can do
  • Privacy: limit each process to data it is permitted to access
  • Fairness: enforce appropriate share of resources (CPU time, memory, I/O, etc)

• It must protect user programs from one another

• Primary mechanism is to limit translation from program address space to physical memory space
  • Can only touch what is mapped into process address space

• There are additional mechanisms as well
  • Privileged instructions, in/out instructions, special registers
  • syscall processing, subsystem implementation
    • (e.g., file access rights, etc)
Process (3\textsuperscript{rd} OS Concept)

- Process: execution environment with restricted rights
  - Address Space with one or more threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources

- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Memory protection
  - Threads more efficient than processes (later)
  - Fundamental tradeoff between protection and efficiency
    - Communication easier within a process
    - Communication harder between processes

- Application instance consists of one or more processes
Single and Multithreaded Processes

- Threads encapsulate concurrency and are active components
- Address spaces encapsulate protection and are passive part
  - Keeps buggy program from trashing system
- Why have multiple threads per address space?
  - Processes are expensive to start, switch between, and communicate between
Dual-Mode Operation
(4th OS Concept)

• Hardware provides at least two modes
  • Kernel mode (or “supervisor” or “protected”)
  • User mode, which is how normal programs are executed

• How can hardware support dual-mode operation?
  • A bit of state (user/system mode bit)
  • Certain operations/actions only permitted in system/kernel mode
    • In user mode they fail or trap
  • User to kernel transition sets system mode AND saves user PC
    • OS code carefully puts aside user state then performs necessary actions
  • Kernel to user transition clears system mode AND restores user PC
    • E.g., rfi: return-from-interrupt
User/Kernel (Privileged) Mode

Kernel Mode
- exec
- syscall
- interrupt
- exception
- rfi
- exit

User Mode
- syscall
- interrupt
- exception
- exit

Limited HW access

Full HW access
Simple Memory Protections: Base and Bound (B&B)

Virtual Address

Physical Address

Stack

Heap

Data

Code

Base

Bound

Raise Exception

0x000...

0x001...

0x010...

0x0110...

0x100...

0x101...

0xFFF...

0xFFF...

0x000...
Towards Virtual Addresses

• What are upsides of B&B?
  • OS protection and program isolation
  • Low overhead address translation

• What are downsides of B&B?
  • Expandable heap?
  • Expandable stack?
  • Memory sharing between processes?
  • Non-relative addresses – hard to move memory around
  • Memory fragmentation
Address Space Translation

• Program operates in address space that is distinct from physical memory space of machine
Virtual Address Example

```c
int staticVar = 0;       // a static variable
int main() {
    staticVar += 1;
    usleep(5000000);       // sleep for 5 seconds
    printf("static address: %x, value: %d\n", &staticVar, staticVar);
}
```

- What happens if we run two instances of this program at the same time?
- What if we took the address of a procedure local variable in two copies of the same program running at the same time?
Putting it All Together: OS Loads Process (with B&B)
OS Gets Ready to Execute Process (with B&B)

- Privileged Inst: set special registers
• How does OS switch between processes?
• First question: How to return to OS?
Three Types of Mode Transfer

- **Syscall**
  - Process requests system service, e.g., `exit`
    - Like function call, but outside process
  - Process does not have address of system function to call
    - Like a Remote Procedure Call (RPC) – for later
  - OS marshalls syscall id and args in registers and exec syscall

- **Interrupt**
  - External asynchronous event triggers context switch, e.g., Timer, I/O device
    - Independent of user process

- **Trap or exception**
  - Internal synchronous event in process triggers context switch, e.g., protection violation (segmentation fault), divide by zero, …

- All 3 are UNPROGRAMMED CONTROL TRANSFER

- How do we get address of unprogrammed control transfer?
Interrupt Vector

- Table set up by OS pointing to code to run on different events

```c
void handleTimerInterrupt() {
    /* handle timer interrupt */
}

void handleDivideByZero() {
    /* handle divide by zero error */
}

void handleSystemCall() {
    /* handle system call */
}
```
User to Kernel Switch (with B&B)

- Proc 1
- Proc 2
- OS

sysmode: 0
Base: 0x1000...
Bound: 0x1100...
SP: 0x00FF...
uPC: ...
PC: 0x00001234
Regs: ...
...

- Stack
- Heap
- Data
- Code

0xFFFF...
0x3080...
0x3000...
0x1100...
0x1000...
0x0000...
Interrupt (with B&B)

- How to save registers and set up system stack?

OS stores copy of registers in its memory

- How to save registers and set up system stack?
Summery: Four Fundamental OS Concepts

- **Thread**
  - Single unique execution context which fully describes program state
  - Program counter, registers, execution flags, stack

- **Address space (with translation)**
  - Address space which is distinct from machine’s physical memory addresses

- **Process**
  - Instance of executing program consisting of address space and 1+ threads

- **Dual-mode operation/protection**
  - Only “system” can access certain resources
  - OS and hardware are protected from user programs
  - User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny