CS 162: Operating Systems and Systems Programming

Lecture 2: Multiprogramming & Dual Mode Operation

June 25, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Recall: What is an OS?

• **Referee**
  • Resource sharing, protection, isolation

• **Illusionist**: clean, easy to use abstractions
  • Infinite memory, dedicated machine
  • Higher level objects: files, users, messages
  • Masking limitations, virtualization

• **Glue**: Common Services
  • Storage
  • Window system
  • Networking
  • Authorization
Recall: Virtual Machines

• Definition: Software emulation of abstract machine
  • Programs believe they own the machine
  • Simulated "hardware" has the features we want

• Two types:
  1. Process VM: Run a single program in the VM, e.g. an OS
  2. System VM: Run entire OS & its apps in the VM, e.g. Virtualbox, VMWare, Xen
Recall: Protecting Processes

• Use two features offered by hardware:
  1. Dual Mode Operation
     • Processes run in user or kernel mode
     • Some operations prohibited in user mode
  2. Address Translation
     • Each process has a distinct and isolated address space
     • Hardware translates from virtual to physical addresses

• Policy: No program can read or write memory of another program or of the OS
Today: Four Fundamental OS Concepts

• **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  - Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  - Address Space + One or more Threads

• **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other
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

Program Source

```
int main()
{
    ...
}
```

Executable

```
data
instructions
```

Memory

```
OS
| stack
| heap
| data
| instructions
```

PC:

```
| registers
| Processor
```

Compiler

- `foo.c` is compiled into `a.out`
The instruction cycle

Processor
- PC:
- Instruction fetch
- Decode
- Execute

Memory
- instruction
- data

Registers

ALU

next

decode
Review (61C): What happens during program execution?

- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat
Today: Four Fundamental OS Concepts

• **Thread: Execution Context**
  • Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  • Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  • Address Space + One or more Threads

• **Dual mode operation / Protection**
  • Only the “system” can access certain resources
  • Combined with translation, isolates programs from each other
Threads of Control

• Definition: **A single, unique execution context**
  • Program counter, registers, stack

• A thread is *executing* on a processor when it is resident in that processor's registers

• Registers hold the root state of the thread:
  • The rest is "in memory"
  • Including program counter – the currently executing instruction

• Registers point to thread state in memory:
  • Stack pointer to the top of the thread's (own) stack
Multiprogramming - Multiple Threads of Control

Thrd. 1  Thrd. 2  ...  Thrd. n

OS

stack
heap
Static Data
code

stack
heap
Static Data
code

stack
heap
Static Data
code
Illusion of Multiple Processors

- Threads are *virtual cores*
- Multiple threads: **Multiplex** hardware in time
Illusion of Multiple Processors

- Contents of virtual core (thread):
  - Program counter, stack pointer
  - Registers

- Where is it?
  - On the real (physical) core, or
  - Saved in memory – called the Thread Control Block (TCB)
Illusion of Multiple Processors

- At T1: vCPU1 on real core, vCPU2 in memory
- At t2: vCPU2 on real core, vCPU1 in memory
- What happened?
  - OS Ran [how?]
  - Saved PC, SP, … in vCPU1's thread control block (memory)
  - Loaded PC, SP, … from vCPU2's TCB, jumped to PC
Very Simple Multiprogramming

• All vCPU's share non-CPU resources
  • Memory, I/O Devices
• Each thread can **read/write data of others**
• Threads **can overwrite OS functions**
• Unusable? No. This approach is used in
  • Embedded applications
  • MacOS 1-9/Windows 3.1 (switch only with voluntary yield)
  • Windows 95-ME (switch with yield or timer)
Today: Four Fundamental OS Concepts

• **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  - Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  - Address Space + One or more Threads

• **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other
Second OS Concept: Address Space

- Definition: **Set of accessible addresses and the state associated with them**
  - $2^{32} = \sim 4$ billion on a 32-bit machine

- What happens when you read or write to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Crash
  - Communicate with another program
Address Space: In a Picture

Processor registers

PC:

SP:

Code Segment

Static Data

instruction

heap

stack

0xFFF...

0x000...
Base and Bound w/o Translation

- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path
Relocation: Quick Review

jal printf

000011 xx xxxxxxx x xxxxxxx x xxxxxxx

opcode for jal  address of printf (shifted right by 2)

• Have to modify program instructions when loading
• Addresses need to point to the right place!
Address Space Translation

• Program operates in an address space that is distinct from the physical memory space of the machine
• Can the program touch OS?
• Can it touch other programs?
Virtual Address Translation/Paging

• Ostensibly introduced in CS 61C
  • Much more on this later in 162!

• Gives every process **the whole address range**

• Breaks memory into pages (~4K chunks)

• Allows unallocated "holes" and other tricks
Today: Four Fundamental OS Concepts

• **Thread**: Execution Context
  • Program Counter, Registers, Execution Flags, Stack

• **Address space** *(with translation)*
  • Program's view of memory is distinct from physical machine

• **Process**: an instance of a running program
  • Address Space + One or more Threads

• **Dual mode operation / Protection**
  • Only the “system” can access certain resources
  • Combined with translation, isolates programs from each other
The Process

• Definition: 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

• Tradeoff: protection vs. efficiency
  • Threads more efficient than processes (more later)

• Application: one or more processes
Single and Multithreaded Processes

- Threads encapsulate **concurrency**: “Active” component
- Address spaces encapsulate **protection**: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
Protection

• Why?
  • Reliability: buggy programs only hurt themselves
  • Security and privacy: trust programs less
  • Fairness: enforce shares of disk, CPU

• Mechanisms:
  • Address translation: address space only contains its own data
  • Privileged instructions, registers
  • Syscall processing (e.g., enforce file access rights)
Today: Four Fundamental OS Concepts

• **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  - Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  - Address Space + One or more Threads

• **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other
Dual Mode Operation: HW Support

• **1 bit** of state (user/kernel mode bit)
• Certain actions only permitted in kernel mode
• User->Kernel mode sets kernel mode, saves user PC
  • Only transition to **OS-designated addresses**
  • OS can save the rest of the user state if necessary
• Kernel->User mode sets user mode, restores user PC
## UNIX Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>Standard Libs</td>
<td>Kernel Interface to the Hardware</td>
</tr>
<tr>
<td>(the users)</td>
<td>shells and commands, compilers and interpreters, system libraries</td>
<td>CPU scheduling, page replacement, demand paging, virtual memory</td>
</tr>
<tr>
<td></td>
<td>system-call interface to the kernel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>signals terminal handling, character I/O system, terminal drivers</td>
<td>terminal controllers, terminals</td>
</tr>
<tr>
<td></td>
<td>file system, swapping block I/O system, disk and tape drivers</td>
<td>device controllers, disks and tapes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory controllers, physical memory</td>
</tr>
</tbody>
</table>
Break
3 Types of U→K Mode Switches

• System Call (syscall)
  • "Function call" into the kernel
  • Example: exit, read
  • Kernel reads syscall id # and arguments from user registers

• Interrupt
  • External asynchronous events: Timer, I/O

• Trap or Exception
  • Special event in process
  • Protection violation (segfault), divide by zero

• All 3 are an *implicit transfer of control*
Where do mode transfers go?

- Cannot let user program specify (why?)
- Solution: **Interrupt Vector**

```c
intrpHandler_i() {
    ...
}
```
Implementing Safe Kernel Mode Transfers

• *Carefully* constructed kernel code packs up the user process state and sets it aside

• Must handle weird/buggy/malicious user state
  • Syscalls with null pointers
  • Return instruction out of bounds
  • User stack pointer out of bounds

• Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

• User program should not know interrupt has occurred (*transparency*)
The Kernel Stack

• Interrupt handlers want a stack
• System call handlers want a stack
• Can't just use the user stack [why?]
The Kernel Stack

• Solution: two-stack model
  • Each OS thread has kernel stack (located in kernel memory) plus user stack (located in user memory)

• Place to save user registers during interrupt
Before Interrupt

User-level Process

code:

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

Kernel

code:

handler() {
    pusha
    ...
}

Exception Stack
During Interrupt

User-level Process

code:

```c
foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}
```

stack:

Kernel

code:

```c
handler() {
    pusha
    ...
}
```

Exception Stack

Registers

- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

- SS
- ESP
- EFLAGS
- CS
- EIP
- error
Kernel System Call Handler

• Vector through well-defined syscall entry points!
  • Table mapping system call number to handler

• Locate arguments
  • In registers or on user (!) stack

• Copy arguments
  • From user memory into kernel memory – carefully checking locations!
  • Protect kernel from malicious code evading checks

• Validate arguments
  • Protect kernel from errors in user code

• Copy results back
  • Into user memory – carefully checking locations!
Hardware support: Interrupt Control

• Interrupt processing not visible to the user process:
  • Occurs between instructions, restarted transparently
  • No change to process state
  • What can be observed even with perfect interrupt processing?

• Interrupt Handler invoked with interrupts ‘disabled’
  • Re-enabled upon completion
  • Non-blocking (run to completion, no waits)
  • Pack up in a queue and pass off to an OS thread for hard work
    • wake up an existing OS thread
Hardware support: Interrupt Control

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

- HW may have multiple levels of interrupts
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    - e.g., kernel segmentation fault
    - Also: Power about to fail!
How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - “Single instruction”-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred
Today: Four Fundamental OS Concepts

• **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  - Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  - Address Space + One or more Threads

• **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other
Putting It Together: Mode Transfer & Translation

• Mode transfer should change address translation mapping

• Examples:
  • Ignore base and bound in kernel mode
  • Page tables with "kernel mode only" bits
Base & Bound: About to Switch

- Privileged Inst: set special registers
- RTU

![Diagram showing memory regions and addressing]

- Base: 1000 ...
- Bound: 1100...
- uPC: 0001...
- PC: 0000 ...
- stack: 00FF ...
- heap: FFFF ...
- code: RTU
- Static Data:
  - heap
  - stack
  - code
Simple B&B: User Code Running

Proc 1  Proc 2  Proc n

OS

sysmode 0
Base 1000 …
Bound 1100…
uPC xxxx…
PC
regs ...

code
Static Data
heap
stack

code
Static Data
heap
stack

0000…
1000…
1100…
3000…
3080…
FFFF…
• Save registers and set up system stack
How do we switch between processes?

• We already have all the necessary machinery!

• Just requires two mode transfers
Base & Bound: Switch User Process

Need to save registers of Proc 1 and restore those of Proc 2

Sysmode

Base: 3000 ...
Bound: 0080 ...
uPC: 0000 0248
PC: 0001 0124

RTU

0000 ...
1000 ...
1100 ...
3000 ...
3080 ...
FFFF ...

Code
Static Data
Heap
Stack

0000 ...
FFFF ...
Base & Bound: Switch User Process

- **Proc 1**, **Proc 2**, ..., **Proc n**
- **OS**

- **sysmode**: 0
- **Base**: 1000 ...
- **Bound**: 1100 ...
- **uPC**: xxxxxx
- **PC**: 0000 100

Diagram shows memory regions:
- **Code**
- **Stack**
- **Static Data**
- **Heap**

Memory addresses:
- **0000** ...
- **FFFF** ...
- **1000** ...
- **1100** ...
- **3000** ...
- **3080** ...
- **0000 100**
Representing Processes: PCB

- **Proc 1**
- **Proc 2**
- **Proc n**

**OS**

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Static Data</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000...</td>
<td>0000...</td>
<td>0000...</td>
</tr>
<tr>
<td>1000...</td>
<td>1000...</td>
<td>1000...</td>
</tr>
<tr>
<td>Bound</td>
<td>xxxxxx</td>
<td>uPC</td>
</tr>
<tr>
<td>1100...</td>
<td>0000 1234</td>
<td>0000 100</td>
</tr>
<tr>
<td>Base</td>
<td>0000 100</td>
<td>PC</td>
</tr>
<tr>
<td>1000...</td>
<td>1000...</td>
<td>1000...</td>
</tr>
<tr>
<td>1100...</td>
<td>1100...</td>
<td>1100...</td>
</tr>
<tr>
<td>0000...</td>
<td>0000...</td>
<td>0000...</td>
</tr>
<tr>
<td>000...</td>
<td>000...</td>
<td>000...</td>
</tr>
<tr>
<td>1000...</td>
<td>1000...</td>
<td>1000...</td>
</tr>
<tr>
<td>1100...</td>
<td>1100...</td>
<td>1100...</td>
</tr>
<tr>
<td>0000...</td>
<td>0000...</td>
<td>0000...</td>
</tr>
<tr>
<td>000FF...</td>
<td>000FF...</td>
<td>000FF...</td>
</tr>
</tbody>
</table>
```

- **PC**
- **sysmode**
- **regs**
Process Control Block

- Kernel representation of each process
  - Status (running, ready, blocked)
  - Register state (if not running)
  - Thread control block(s)
  - Process ID
  - Execution time
  - Memory translations

- **Scheduler** maintains a data structure of PCBs

- *Scheduling algorithm*: Which process should the OS run next?
Scheduling: Mechanism for deciding which processes/threads receive the CPU

Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

```c
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
```
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Request

Reply
Conclusion: Four OS Concepts

• **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack

• **Address space (with translation)**
  - Program's view of memory is distinct from physical machine

• **Process: an instance of a running program**
  - Address Space + One or more Threads

• **Dual mode operation / Protection**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other