The Kernel
wants to be your friend
Boxing them in

- Buggy apps can crash other apps
- Buggy apps can crash OS
- Buggy apps can hog all resources
- Malicious apps can violate privacy of other apps
- Malicious apps can change the OS
The Process

- An abstraction for protection the execution of an application program with restricted rights
- But there are tradeoffs (there always are tradeoffs!)
- Must not hinder functionality
  - still efficient use of hardware
  - enable safe communication
The Process

- An abstraction for protection
  - the execution of an application program with restricted rights

- But there are tradeoffs
  (there always are tradeoffs!)

- Must not hinder functionality
  - still efficient use of hardware
  - enable safe communication
I know what you are thinking...

This looks like a job for Superman!
Special

- Part of the OS
  - all kernel is in the OS
  - not all the OS is in the kernel
    - (why not? robustness)
  - widgets libraries, window managers etc
Process:
Getting to know you

- A process is a program during execution
  - program is a static file
  - process = executing program = program + execution state

Source code

compiler

Physical memory

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
</table>
| OS copy

Executable Image

- Header
- Code
- Initialized data
- 
- 
- 
- 

- Physical memory

- Code
- Data
- Heap
- Stack
Keeping track of a process

- A process has code
  - OS must track program counter

- A process has a stack
  - OS must track stack pointer

- OS stores state of process in Process Control Block (PCB)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB
How can the OS enforce restricted rights?

Easy: kernel interprets each instruction!
- slow
- many instructions are safe: do we really need to involve the OS?
How can the OS enforce restricted rights?

- Easy: kernel interprets each instruction!
  - slow
  - many instructions are safe: do we really need to involve the OS?

- Dual Mode Operation
  - hardware to the rescue: use a mode bit
    - in user mode, processor checks every instruction
    - in kernel mode, unrestricted rights
  - hardware to the rescue (again) to make checks efficient
Efficient protection in dual mode operation

- Privileged instructions
  - in user mode, no way to execute potentially unsafe instructions

- Memory protection
  - in user mode, memory accesses outside a process’ memory region are prohibited

- Timer interrupts
  - kernel must be able to periodically regain control from running process

+ Efficient mechanism for switching modes
I. Privileged instructions

- Examples: Set mode bit; set accessible memory; disable interrupts; etc

- But how can an app do I/O then?
  - System calls achieve access to kernel mode only at specific locations specified by OS

- Executing a privileged instruction while in user mode causes a processor exception....
  - ...which passes control to the kernel
II. Memory Protection

Step 1: Virtualize Memory

- Virtual address space: set of memory addresses that process can “touch”
  - CPU works with virtual addresses

- Physical address space: set of memory addresses supported by hardware

<table>
<thead>
<tr>
<th>Virtual address space</th>
<th>mapped segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>DLL’s</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>Initialized data</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
</tr>
</tbody>
</table>
II. Memory Protection

Step 2: Address Translation

Implement a function mapping
into

Virtual

Physical

Advantages:
- protection
- relocation
- data sharing
- multiplexing

a486d9

5e3a07
Protection

At all times, the functions used by different processes map to disjoint ranges.
Relocation

The range of the function used by a process can change over time
Relocation

The range of the function used by a process can change over time.
Data Sharing

Map different virtual addresses of different processes to the same physical address
Multiplexing

Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses.
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time
The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
A simple mapping mechanism: Base & Bound

- CPU
- Logical addresses ≤ yes
- No Memory Exception
- Physical addresses +
- Bound Register 500
- Base Register 1000
- p’s physical address space
- MAXsys
- 0 1000 1500
- ≤ 500 1000
On Base & Limit

- Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses

- Protection is easy, but sharing is hard
  - Two copies of emacs: want to share code, but have data and stack distinct...

- And there is more...
  - Hard to relocate
  - We want them as far as possible in virtual address space, but...
III. Timer Interrupts

- Hardware timer
  - can be set to expire after specified delay (time or instructions)
  - when it does, control is passed back to the kernel

- Other interrupts (e.g. I/O completion) also give control to kernel
Crossing the line

user process

- user process executing
- calls system call
- return from system call

mode bit := 1

trap

mode bit := 0

execute system call

kernel

mode bit = 0

return

mode bit := 1
From user mode to kernel mode...

**Exceptions**
- user program acts silly (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous

**System calls/traps**
- user program requests OS service
- synchronous

**Interrupts**
- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous
...and vice versa

Resume after exception, interrupt or syscall
- restore PC, SP, registers;
- toggle mode

Switch to different process
- load PC, SP, registers from 's PCB
- toggles mode

If new process
- copy program in memory,
- set PC and SP
- toggle mode

User-level upcall
- a sort of user-level interrupt handling
Making the transition: Safe mode switch

- Common sequences of instructions to cross boundary, which provide:
  - Limited entry
    - entry point in the kernel set up by kernel
  - Atomic changes to process state
    - PC, SP, memory protection, mode
  - Transparent restartable execution
    - user program must be restarted exactly as it was before kernel got control
Hardware identifies why boundary is crossed
- trap?
- interrupt (which device)?
- exception?
Hardware selects entry from interrupt vector
Appropriate handler is invoked
Interrupt stack

- Stores execution context of interrupted process
  - HW saves SP, PC
  - Handler saves remaining registers
- Stores handler’s local variables
- Pointed by privileged register
- One per process (or per thread!)
  - Why not use the stack in user’s space?
Interrupt masking

What if an interrupt occurs while running an interrupt handler?

- Disable interrupts via privileged instruction
  - Overdramatic... it actually defers them
- Just use the current SP of Interrupt stack
Mode switch on x86

User-level Process

Code

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

Stack

Registers

```
Stack segment

SS:ESP
CS:EIP
EFLAGS
Other
Registers:
EAX, EBX,
...
```

Kernel

Code

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

Interrupt Stack
Mode switch on x86

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
    ... 
}

Interrupt Stack

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Mode switch on x86

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
    ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
**Mode switch on x86**

### User-level Process
- **Code**
  ```
  foo() {
    while(...) {
      x = x+1;
      y = y-2
    }
  }
  ```
- **Stack**

### Registers
- **CS:EIP**
- **SS:ESP**
- **EFLAGS**
- **Other Registers:** EAX, EBX, ...

### Kernel
- **Code**
  ```
  handler() {
    pusha
    ...
  }
  ```
- **Interrupt Stack**
  ```
  SS:ESP
  CS:EIP
  EFLAGS
  ```

### Process
- Disable interrupts
- Save key registers
- Switch onto the kernel interrupt stack
- Push key registers onto new stack
Mode switch on x86

**User-level Process**

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

**Kernel**

- Code
  - `handler()` {
    - `pusha`
    - ...
  }
- Interrupt Stack
  - SS:ESP
  - CS:EIP
  - EFLAGS

**Registers**

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers:
  - EAX, EBX,
  - ...

1. Disable interrupts
2. Save key registers
3. Switch onto the kernel interrupt stack
4. Push key registers onto new stack
5. Save error code (optional)
Mode switch on x86

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
    ...
}

Interrupt Stack

SS:ESP

CS:EIP

EFLAGS

Error

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Mode switch on x86

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
  ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS
Error

Disable interrupts
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Invoke interrupt handler
Mode switch on x86

User-level Process

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

Kernel

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

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Interrupt Stack

- SS:ESP
- CS:EIP
- EFLAGS
- Error

Error

1. Disable interrupt
2. Save key registers
3. Switch onto the kernel interrupt stack
4. Push key registers onto new stack
5. Save error code (optional)
6. Invoke interrupt handler
7. Handler pushes all registers on stack
Mode switch on x86

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
    ...
}

Interrupt Stack

Disable interrupt
Save key registers
Switch onto the kernel interrupt stack
Push key registers onto new stack
Save error code (optional)
Invoke interrupt handler
Handler pushes all registers on stack
Switching back

- From an interrupt, just reverse all steps!
- From exception and system call, increment PC on return
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user level state
System calls

- Programming interface to the services provided by the OS
- Mostly accessed through an API (Application Programming Interface)
  - Win32, POSIX, Java API
- Parameters passed according to calling convention
  - registers, stack, etc.
System call stubs

User

- Set up parameters

- call int 080 to context switch

open:
  movl #SysCall_Open, %eax
  int 080
  ret

Kernel

- Validate parameters
  - defend against errors in content and format of args

- Copy before check
  - prevent TOCTOU

- Copy back any result
The Skinny

- Syscall interface allows separation of concern
- Innovation
- Narrow
  - simple
  - powerful
  - highly portable
  - robust
- Portable OS Kernel
- Portable OS Library
- System call interface
- Compilers
- Web Servers
- Databases
- Word Processing
- Web Browsers
- Email
- 10Mbps/100Mbps/1Gbps Ethernet
- 1802.11 a/b/g/n
- SCSI
- Graphics accelerators
- LCD Screens
## Upcalls: virtualizing interrupts

<table>
<thead>
<tr>
<th>Interrupts/Exceptions</th>
<th>Upcalls/Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware-defined</td>
<td>Kernel-defined signals</td>
</tr>
<tr>
<td>Interrupts &amp; exceptions</td>
<td>Handlers (user)</td>
</tr>
<tr>
<td>Interrupt vector for handlers (kernel)</td>
<td></td>
</tr>
<tr>
<td>Interrupt stack (kernel)</td>
<td>Signal stack (user)</td>
</tr>
<tr>
<td>Interrupt masking (kernel)</td>
<td>Signal masking (user)</td>
</tr>
<tr>
<td>Processor state (kernel)</td>
<td>Processor State (user)</td>
</tr>
</tbody>
</table>
Signaling

Why?

- To terminate an application
- To suspend it/resume it (e.g., for debugging)
- To alert of timer expiration

Upon receipt:

- Ignore
- Terminate process
- Catch through handler
## More on signals

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., CTRL-C from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Tmer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>20</td>
<td>SIGSTP</td>
<td>Stop until SIGCONT</td>
<td>Stop signal from terminal (e.g., CTRL-Z from keyboard)</td>
</tr>
</tbody>
</table>
Unix signals

User-level Process

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

Stack

Registers

Code
    signal_handler() {
        ...
    }

Kernel

Interrupt Stack

HW copies current user state in Interrupt stack

User-level Process

When a signal occurs, the processor enters an interrupt mode.

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

HW copies current user state in Interrupt stack.
Unix signals

User-level Process

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

Stack

Registers

Code

EAX, EBX,
...

EFLAGS

Other Registers:
EAX, EBX,
...

Interrupt Stack

Kernel

signal_handler() {
  ...
}

EAX, EBX,
...

Kernel copies user state on user stack
Unix signals

User-level Process

Code

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

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Signal Handler

Code

signal_handler() {
    ...
}

Interrupt Stack

Kernel changes PC saved on Interrupt stack to point to handler and SP to point after state saved on user stack
Unix signals

User-level Process

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

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, …

Code

Stack

Interrupt Stack

Kernel

signal_handler() {
    ...
}

Kernel exits; Interrupt stack copied back into registers
Unix signals

User-level Process

Code

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

Stack

Registers

SS:ESP'
CS:EIP'
EFLAGS
Other Registers: EAX, EBX, ...

Kernel

Interrupt Stack

Kernel exits; Interrupt stack copied back into registers
Unix signals

User-level Process

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

Stack

Code

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX,
 ...

Kernel

Interrupt Stack

5 Signal handler returns

User-level Process

Code

signal_handler() {
  ...
}

Stack

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX,
 ...

Interrupt Stack
Booting an OS Kernel

- Basic Input/Output System
- In ROM, includes the first instructions fetched and executed
- BIOS copies bootloader, using a cryptographic signature to make sure it has not been tampered with
Booting an OS Kernel

- Bootloader copies OS kernel, checking its cryptographic signature
Booting an OS Kernel

- Kernel initializes its data structures
- Starts first process by copying it from disk
- Let the dance BEGIN!
Shall we dance?

- All processes are progeny of that first process
- Created with a little help from its friend...

...via system calls!
Starting a new process

- A simple recipe:
  - Allocate & initialize PCB
  - Create and initialize a new address space
  - Load program into address space
  - Allocate user-level and kernel-level stacks
  - Initialize hw context to start execution at "start"
  - Copy arguments (if any) to the base of the user-level stack
  - Inform scheduler process new process is ready
  - Transfer control to user-mode
Which API?

Windows: CreateProcess System Call

```c
if (!CreateProcess(
    NULL,            // No module name (use command line)
    argv[1],         // Command line
    NULL,            // Process handle not inheritable
    NULL,            // Thread handle not inheritable
    FALSE,           // Set handle inheritance to FALSE
    0,               // No creation flags
    NULL,            // Use parent's environment block
    NULL,            // Use parent's starting directory
    &si,             // Pointer to STARTUPINFO structure
    &pi )            // Ptr to PROCESS_INFORMATION structure
)
```

Everything you might want to control... but wait!

- CreateProcessAsUser
- CreateProcessWithLogonW
Which API?

Unix: `fork()` and `exec()`

`fork()`

Creates a complete copy (child) of the invoking process (parent) — but for return value:

child := 0;
parent := child’s pid

`exec()`

Loads executable in memory & starts executing it

- code, stack, heap are overwritten
- the process is now running a different program!
The genius of fork() and exec()

- To redirect stdin/stdout:
  - fork, close/open files, exec

- To switch users:
  - fork, setuid, exec

- To start a process with a different current directory:
  - fork, chdir, exec

You get the idea!

But what about overhead?
wait() and exit()

- wait() causes parent to wait until child terminates
  - parent gets return value from child
  - if no children alive, wait() return immediately

- exit() is called after program terminates
  - closes open files
  - deallocates memory
  - deallocates most OS structures
  - checks if parent is alive. If so...
In action

/* See Figure 3.5 in textbook*/

#include <stdio.h>
#include <unistd.h>

int main() {

    int child_pid = fork();

    if (child_pid == 0) {  // child process
        printf("I am process #%d\n", getpid());
        return 0;
    } else {  // parent process
        printf("I am the parent of process #%d\n", child_pid);
        return 0;
    }
}

In action

Process 13
Program A

PC
pid
?

pid = fork();
if (pid==0)
  exec(B);
else
  wait(pid);

Process 13
Program A

PC
pid
14

pid = fork();
if (pid==0)
  exec(B);
else
  wait(pid);

Process 13
Program A

PC
pid
0

pid = fork();
if (pid==0)
  exec(B);
else
  wait(pid);

Process 14
Program A

PC
pid
0

main() {
  ...
}

Process 14
Program B
Shell

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh: Original Unix shell (Bourne, 1977)
  - csh: BSD Unix C shell (tcsh enhances it)
  - bash: “Bourne again” shell
- Every command typed in the shell starts a child process of the shell