The Process

A running program

A First Cut at the API

- Create
  - causes the OS to create a new process
- Destroy
  - forcefully terminates a process
- Wait (for the process to end)
- Other controls
  - e.g. to suspend or resume the process
- Status
  - running? suspended? blocked? for how long?

From Program to Process

- To make the program’s code and data come alive
  - need a CPU
  - need memory — the process’ address space
    - for data, code, stack, heap
  - need registers
    - PC, SP, regular registers
  - need access to I/O
    - list of open files

How the OS Keeps 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
You’ll Never Walk Alone

- Machines run (and thus OS must manage) multiple processes
  - how should the machine’s resources be mapped to these processes?
- OS as a referee...

Enter the illusionist!
- give every process the illusion of running on a private CPU
  - which appears slower than the machine’s
- give every process the illusion of running on a private memory
  - which may appear larger(??) than the machine’s

Isolating Applications

- 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

Operating System

Reading and writing memory, managing resources, accessing I/O...

Mechanism and Policy

- Mechanism
  - what the system can do
- Policy
  - what the system should do

Mechanisms should not determine policies!
The Process, Refined

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

The enforcing mechanism must not hinder functionality
- still efficient use of hardware
- enable safe communication

Special

The process abstraction is enforced by the kernel
- all kernel is in the OS
- not all the OS is in the kernel
  - (why not? robustness)
  - widgets libraries, window managers etc

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?

**Mechanism: 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

Amongst our weaponry are such diverse elements as...

- Privileged instructions
  - In user mode, no way to execute potentially unsafe instructions
- Memory isolation
  - 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

I. Privileged instructions

- Set mode bit
- I/O ops
- Memory management ops
- Disable interrupts
- Set timers
- Halt the processor

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 (naughty naughty...) 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

II. Memory Isolation

Step 2: Address Translation
- Implement a function mapping

Isolation
- At all times, functions used by different processes map to disjoint ranges — aka “Stay in your room!”
**Relocation**

- The range of the function used by a process can change over time — “Move to a new room!”

**Data Sharing**

- Map different virtual addresses of distinct processes to the same physical address — “Share the kitchen!”

**Multiplexing**

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?
The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
More Multiplexing

At different times, different processes can map part of their virtual address space into the same physical memory — change tenants!

A simple mapping mechanism: Base & Bound

On Base & Limit

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

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

And there is more...
- Hard to relocate
- Hard to account for dynamic changes in both heap and stack
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

Interrupt Management

Interrupt controllers implement interrupt priorities:
- Interrupts include descriptor of interrupting device
- Priority selector circuit examines all interrupting devices, reports highest level to the CPU
- Controller can also buffer interrupts coming from different devices
  - more on this later...

Interrupt Management

Maskable interrupts
- can be turned off by the CPU for critical processing

Nonmaskable interrupts
- indicate serious errors (power out warning, unrecoverable memory error, etc.)

Types of Interrupts

Exceptions
- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose (breakpoints)
- synchronous/non-maskable

System calls/traps
- user program requests OS service
  - synchronous/non-maskable

Interrupts
- HW device requires OS service
  - timer, I/O device, interprocessor
  - asynchronous/maskable
Interrupt Handling

- Two objectives
  - handle the interrupt and remove the cause
  - restore what was running before the interrupt
    - state may have been modified on purpose
- Two “actors” in handling the interrupt
  - the hardware goes first
  - the kernel code takes control by running the interrupt handler

A Tale of Two Stack Pointers

- Interrupt is a program: it needs a stack!
- so, each process has two stacks pointers:
  - one when running in kernel mode
  - another when running in user mode
- Why not using the user-level stack pointer?
  - user SP may be badly aligned or pointing to non-writable memory
  - user stack may not be large enough, and may spill to overwrite important data
  - security:
    - kernel could leave sensitive data on stack
    - pointing SP to kernel address could corrupt kernel

Handling Interrupts: HW

- On interrupt, hardware:
  - sets supervisor mode (if not set already)
  - disable (masks) interrupts
  - pushes PC, SP, and PSW
  - sets PC to point to the first instruction of the appropriate interrupt handler
    - depends on interrupt type
    - interrupt handler specified in interrupt vector loaded at boot time
  - pushes PC, SP, and PSW
    - of user program on interrupt stack
  - sets PC to point to the first instruction of the appropriate interrupt handler

Handling Interrupts: SW

- We are now running the interrupt handler!
  - IH first pushes the registers’ contents on the interrupt stack (part of the PCB)
  - need registers to run the IH
  - only saves necessary registers (that’s why done in SW, not HW)
Typical Interrupt Handler Code

HandleInterruptX:

```
PUSH %Rn
... only need to save registers not
PUSH %R1
saved by the handler function
CALL _handleX
POP %R1
... restore the registers saved above
POP %Rn
RETURN_FROM_INTERRUPT
```

Returning from an Interrupt

- Hardware pops PC, SP, PSW
- Depending on content of PSW
  - switch to user mode
  - enable interrupts
- From exception and system call, increment PC
  - on return (we don’t want to execute again the same instruction)
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user level state

Starting a new process: the recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base of the user-level stack
6. Simulate an interrupt
   a) push initial PC, user SP
   b) push PSW (supervisor mode off, interrupts enabled)
7. Clear all other registers
8. RETURN_FROM_INTERRUPT

Interrupt Handling on x86

User-level Process

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

- Stack

Registers

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

Kernel

- Code
  handler() {
    ...
  }

- Interrupt Stack
Interrupt Handling on x86

User-level Process

Registers

Kernel

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

Stack

Hardware performs these steps
Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack

Interrupt Stack

Other Registers:
EAX, EBX,
...

EFLAGS

SS:ESP
CS:EIP

Hardware performs these steps
Change mode bit
Disable interrupts
Save key registers to temporary location
Switch onto the kernel interrupt stack
Push key registers onto new stack

EFLAGS

SS:ESP
CS:EIP
Interrupt Handling on x86

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

Stack

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

Kernel
Code
handler() {
  pusha
  ...
}

Interrupt Stack
SS:ESP
CS:EBP
EFLAGS
...

Other Registers:
EAX, EBX,
...

Hardware performs these steps
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step
Handler pushes all registers on stack

Interrupt Handling on x86

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

Stack

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

Kernel
Code
handler() {
  pusha
  ...
}

Interrupt Stack
SS:ESP
CS:EBP
EFLAGS
...

Other Registers:
EAX, EBX,
...

Hardware performs these steps
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step
Handler pushes all registers on stack
Interrupt Safety

- Kernel should disable device interrupts as little as possible
  - Interrupts are best serviced quickly
- Thus, device interrupts are often disabled selectively
  - E.g., clock interrupts enabled during disk interrupt handling
- This leads to potential “race conditions”
  - System's behavior depends on timing of uncontrollable events

Interrupt Race Example

- Disk interrupt handler enqueues a task to be executed after a particular time
  - While clock interrupts are enabled
- Clock interrupt handler checks queue for tasks to be executed
  - May remove tasks from the queue
- Clock interrupt may happen during enqueue

**Concurrent access to a shared data structure (the queue!)**

Making code interrupt-safe

- Make sure interrupts are disabled while accessing mutable data!
- But don’t we have locks?
  - Consider

```c
void function ()
{
    lock(mtx);
    /* code */
    unlock(mtx);
}
```

Is function thread-safe?
- Operates correctly when accessed simultaneously by multiple threads
- To make it so, grab a lock

Is function interrupt-safe?
- Operates correctly when called again (re-entered) before it completes
- To make it so, disable interrupts

Example of Interrupt-Safe Code

```c
void enqueue(struct task *task) {
    int level = interrupt_disable();
    /* update queue */
    interrupt_restore(level);
}
```

Why not simply re-enable interrupts?
- Say we did. What if then we call enqueue from code that expects interrupts to be disabled?
  - Oops...
- Instead, remember interrupt level at time of call; when done, restore that level
Many Standard C Functions are not Interrupt-Safe

- Pure system calls are interrupt-safe
  - e.g., read(), write(), etc.
- Functions that don’t use global data are interrupt-safe
  - e.g., strlen(), strcpy(), etc.
- malloc(), free(), and printf() are not interrupt-safe
  - must disable interrupts before using it in an interrupt handler
  - and you may not want to anyway (printf() is huge!)

System calls

- Programming interface to the services the OS provides:
  - read input/write to screen
  - create/read/write/delete files
  - create new processes
  - send/receive network packets
  - get the time / set alarms
  - terminate current process
  - ...

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
- “Narrow waist” makes it highly portable
  - robust (small attack surface)
- Internet IP layer also offers skinny interface

Executing a System Call

- Process:
  - Calls system call function in library
  - Places arguments in registers and/or pushes them onto user stack
  - Places syscall type in a dedicated register
  - Executes syscall machine instruction

- Kernel
  - Executes syscall interrupt handler
  - Places result in dedicated register
  - Executes RETURN_FROM_INTERRUPT

- Process:
  - Executes RETURN_FROM_FUNCTION
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

note: interrupt stack is empty while process running
### Executing read System Call

```c
int main(argc, argv){
    ... 
    c = read(fd, buffer, nbytes)
    ...
}
```

#### stack frame for main()
- USP
- KSP
- user stack
- UPC

#### stack frame for _read
- USP, UPC, PSW
- saved registers
- stack frame for handleSyscall()

#### interrupt stack
- USP, UPC, PSW
- saved registers

#### HandleIntrSyscall:
- push %Rn
- push %R1
- call __handleSyscall
- pop %R1
- pop %Rn
- return_from_interrupt

### What if read needs to block?
- read may need to block if:
  1. It reads from a terminal
  2. It reads from disk, and block is not in cache
  3. It reads from a remote file server

We should run another process!

### Virtualizing the CPU

- OS keeps a PCB for each process
- It has space to hold a “frozen” version of the state process’s state
  - Program counter
  - Process status (ready, running, etc)
  - CPU registers
  - CPU scheduling info
  - Memory management info
  - Account info
  - I/O status info
  - to be saved when the process relinquishes the CPU
  - and reloaded when the process reacquires the CPU

### Process Life Cycle

- Process Control Block
  - PC
  - Stack Ptr
  - Registers
  - PID
  - UID
  - Priority
  - List of open files
  - Process status
  - Kernel stack ptr
  - Location in Memory
  - Location of executable on disk

- Process Life Cycle:
  - Init
  - Ready
  - Running
  - Waiting
  - Zombie
Process Life Cycle

- **Init**
  - PCB: being created
  - Registers: uninitialized

- **Ready**
  - PCB: on the Ready queue
  - Registers: pushed by kernel code onto interrupt stack

- **Running**
  - PCB: currently executing
  - Registers: popped from interrupt stack into CPU

- **Waiting**
  - PCB: being created
  - Registers: uninitialized

- **Zombie**
  - Admitted to the Ready queue
  - PCB: on the Ready queue

Process Life Cycle

- **Init**
  - PCB: being created
  - Registers: uninitialized

- **Ready**
  - PCB: on the Ready queue
  - Registers: pushed by kernel code onto interrupt stack

- **Running**
  - PCB: currently executing
  - Registers: popped from interrupt stack into CPU

- **Zombie**
  - Admitted to the Ready queue
  - PCB: on the Ready queue

- **Waiting**
  - PCB: being created
  - Registers: uninitialized
Process Life Cycle

Init

Admitted to the Ready queue

Ready

Dispatch

Running

Yield

Init

Admitted to the Ready queue

PCB: on Ready queue

Registers: pushed onto interrupt stack (SP saved in PCB)

Waiting

77

Init

Admitted to the Ready queue

PCB: currently executing

Registers: popped from interrupt stack into CPU

Waiting

78

Process Life Cycle

Init

Admitted to the Ready queue

PCB: on specific waiting queue (I/O device, lock, etc.)

Registers: on interrupt stack

Waiting

79

Init

Admitted to the Ready queue

PCB: on Ready queue

Registers: on interrupt stack

Waiting

80
Process Life Cycle

Init
- Admitted to the Ready queue
- PCB: currently executing
- Registers: popped from interrupt stack into CPU

Ready
- Dispatch
- Waiting
- PCB: on Finished queue, ultimately deleted
- Registers: no longer needed

Running
- Yield
- Blocking call completion
- e.g., read(), wait()

Waiting
- Yield
- Blocking call completion
- e.g., read(), wait()

Zombie
- PCB: on Finished queue, ultimately deleted
- Registers: no longer needed

Invariants to keep in mind
- At most one process/core running at any time
- When CPU in user mode, current process is RUNNING and its interrupt stack is empty
- If process is RUNNING
  - its PCB not on any queue
  - it is not necessarily in USER mode
- If process is RUNNABLE or WAITING
  - its registers are saved at the top of its interrupt stack
  - its PCB is either
    - on the READY queue (if RUNNABLE)
    - on some WAIT queue (if WAITING)
- If process is a ZOMBIE
  - its PCB is on FINISHED queue

Cleaning up Zombies
- Process cannot clean up itself (why?)
- Process can be cleaned up
  - by some other process, checking for zombies before returning to RUNNING state
  - or by parent which waits for it
    - but what if parent turns into a zombie first?
  - or by a dedicated “reaper” process
- Linux uses a combination
  - if alive, parent cleans up child that it is waiting for
  - if parent is dead, child process is inherited by the initial process, which is continually waiting
How to Yield/Wait?

- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING    READY
  - Next process: READY     RUNNING

Save kernel registers of Current on its interrupt stack
Save kernel SP of Current in its PCB
Restore kernel SP of Next from its PCB
Restore kernel registers of Next from its interrupt stack

Yielding

```c
void yield(){
    assert(current->state == RUNNING);
    current->state = RUNNABLE;
    runQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp);
    current = next;
}
```

Starting a New Process

```c
void createProcess(func){
    current->state = READY;
    readyQueue.add(current);
    next = malloc(…);
    next->func = func;
    next->state = RUNNING;
    ctx_start(&current->sp, next->top_of_stack);
    current = next;
}
```

Anybody there?

- What if no process is READY?
  - scheduler() would return NULL — aargh!
- To avoid armageddon
  - OS always runs a low priority process, in an infinite loop executing the HLT instruction
    - halts CPU until next interrupt
  - Interrupt handler executes yield() if some other process is put on the Ready queue
Three Flavors of Context Switching

- **Interrupt:** from user to kernel space
  - on system call, exception, or interrupt
  - Px user stack, Px interrupt stack
- **Yield:** between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Px interrupt stack, Py interrupt stack
- **Return from interrupt:** from kernel to user space
  - with the homonymous instruction
  - Px interrupt stack, Px user stack

Switching between Processes

- **Process 1**
  - User Space
    - read(file)
  - Kernel Space
    - disk_read()
  - Return from interrupt
  - resume
- **Process 2**
  - User Space
  - save Process 1 user registers
  - Kernel Space
  - save Process 1 kernel registers and restore Process 2 kernel registers
  - Restore Process 2 user registers

System Calls to Create a New Process

- **Windows**
  - `CreateProcess(...);`
- **Unix (Linux)**
  - `fork() + exec(...)`

CreateProcess (Simplified)

```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
)
```

[Windows]
fork (actual form)

process identifier

```c
int pid = fork();
```

..but needs exec(…)

[Unix]

Kernel Actions to Create a Process

- **fork()**
  - allocate ProcessID
  - initialize PCB
  - create and initialize new address space
  - inform scheduler new process is READY

- **exec(program, arguments)**
  - load program into address space
  - copy arguments into address space's memory
  - initialize h/w context to start execution at “start”

- **CreateProcess(...)** does both

Creating and managing processes

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child</td>
</tr>
<tr>
<td>exec(prog, args)</td>
<td>Run application prog in the current process with the specified args (replacing any code and data that was present in process)</td>
</tr>
<tr>
<td>wait(&amp;status)</td>
<td>Pause until a child process has exited</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Tell kernel current process is complete and its data structures (stack, heap, code) should be garbage collected. May keep PCB.</td>
</tr>
<tr>
<td>kill(pid, type)</td>
<td>Send an interrupt of a specified type to a process (a bit of an overdramatic misnomer...)</td>
</tr>
</tbody>
</table>

[Unix]

In action

```
Process 13
Program A

pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
```
In action

```c
pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
```

Process 13
Program A

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

Process 14
Program B

What is a 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
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

```
while(! EOF)
    read input
    handle regular expressions
int pid = fork()   // create child
if (pid == 0) { // child here
    exec("program", argc, argv0,...);
} else { // parent here
    ...
}
```
Signals (Virtualized Interrupts)

Asynchronous notifications in user space

<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>Timer 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>

### Handler Example

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}
```

```c
int main() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler); // register handler for SIGINT
    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

### Kernel Operation (conceptual, simplified)

- Initialize devices
- Initialize "first process"
- while (TRUE) {
  - while device interrupts pending
    - handle device interrupts
  - while system calls pending
    - handle system calls
  - if run queue is non-empty
    - select a runnable process and switch to it
  - otherwise
    - wait for device interrupt
}
Booting an OS Kernel

1. BIOS copies Bootloader, checking its cryptographic hash to make sure it has not been tampered with.

2. Bootloader copies OS Kernel, checking its cryptographic hash.

3. OS Kernel initializes its data structures (devices, interrupt vector table, etc.)
Boot an OS Kernel

1. Bootloader
2. OS Kernel
3. Login app

4. Kernel: Copies first process from disk
Changes PC and sets mode bit to 1
And the dance begins!