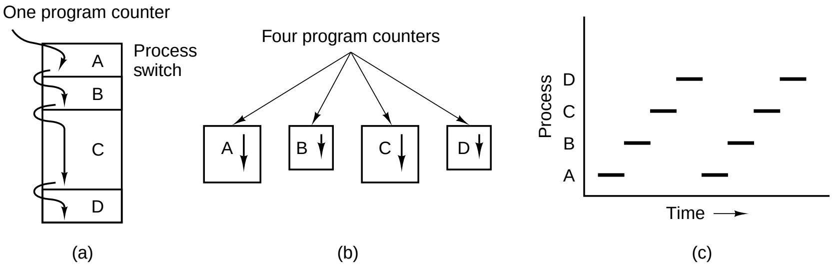
Sistemas Operativos: Limited Direct Execution

Pedro F. Souto (pfs@fe.up.pt)

February 27, 2020

Multiprocess Execution



- The processor is **time shared** among processes
- The OS provides the illusion that each process executes in its own processor, i.e. each process executes in a virtual processor.

Kernel Data Structures (xv6 toy-OS)

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

src: Arpaci-Dusseau & Arpaci-Dusseau

Multiprocess Execution: Challenges

Performance How to implement virtualization **efficently**?

Multiprocess Execution: Challenges

Performance How to implement virtualization **efficently**?

Protection How to **protect** the OS from processes and processes from one another?

Multiprocess Execution: Challenges

Performance How to implement virtualization **efficently**?

Protection How to **protect** the OS from processes and processes from one another?

Approach **Limited Direct Execution**

Direct Execution

- Just run the program directly on the CPU:

OS	Program
Create entry for process list	

Direct Execution

- Just run the program directly on the CPU:

OS	Program
Create entry for process list Allocate memory for program	

Direct Execution

- Just run the program directly on the CPU:

OS	Program
Create entry for process list	
Allocate memory for program	
Setup stack with argc/argv	

Direct Execution

- Just run the program directly on the CPU:

OS	Program
Create entry for process list Allocate memory for program Setup stack with argc/argv Clear registers	

Direct Execution

- Just run the program directly on the CPU:

OS	Program
Create entry for process list Allocate memory for program Setup stack with argc/argv Clear registers Call `main()`	

Direct Execution

- Just run the program directly on the CPU:

OS	**Program**
Create entry for process list	
Allocate memory for program	
Setup stack with argc/argv	
Clear registers	
Call `main()`	
	Run `main()`

Direct Execution

- Just run the program directly on the CPU:

OS	**Program**
Create entry for process list	
Allocate memory for program	
Setup stack with argc/argv	
Clear registers	
Call `main()`	
	Run `main()`
	Return from `main()`

Direct Execution

- Just run the program directly on the CPU:

OS	**Program**
Create entry for process list	
Allocate memory for program	
Setup stack with argc/argv	
Clear registers	
Call `main()`	
	Run `main()`
	Return from `main()`
Free memory of process	

Direct Execution

- Just run the program directly on the CPU:

OS	**Program**
Create entry for process list	
Allocate memory for program	
Setup stack with argc/argv	
Clear registers	
Call `main()`	
	Run `main()`
	Return from `main()`
Free memory of process	
Remove entry from process list	

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?
   - If a process is allowed to access the entire disk, then it will not be possible to protect files from access by non-authorized users

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?
   - If a process is allowed to access the entire disk, then it will not be possible to protect files from access by non-authorized users
   - And yet, processes usually need to access files on disk

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?
    - If a process is allowed to access the entire disk, then it will not be possible to protect files from access by non-authorized users
    - And yet, processes usually need to access files on disk
2. In particular, how does the OS stop a process and context switch to another process?

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?
   - If a process is allowed to access the entire disk, then it will not be possible to protect files from access by non-authorized users
   - And yet, processes usually need to access files on disk
2. In particular, how does the OS stop a process and context switch to another process?
   - This is necessary to ensure that all processes are able to run

Direct Execution: Issues

1. How can the OS prevent a process from doing something it does not want the process to do?
    - If a process is allowed to access the entire disk, then it will not be possible to protect files from access by non-authorized users
    - And yet, processes usually need to access files on disk
2. In particular, how does the OS stop a process and context switch to another process?
    - This is necessary to ensure that all processes are able to run
    - But, to ensure efficiency it is important to run processes directly on the HW.

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Kernel mode when the CPU executes in this mode:

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Kernel mode when the CPU executes in this mode:

- It is allowed to execute all instructions, including privileged instructions;

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Kernel mode when the CPU executes in this mode:

- It is allowed to execute all instructions, including privileged instructions;
- It is allowed to access the entire memory.

The kernel always executes in kernel mode (hence the name)

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Kernel mode when the CPU executes in this mode:

- It is allowed to execute all instructions, including privileged instructions;
- It is allowed to access the entire memory.

The kernel always executes in kernel mode (hence the name)

Issue How can a user process do I/O?

Processor Modes (Privilege Levels)

How can the OS prevent a process from doing what it should not?

User mode when the CPU executes in this mode:

- It is not allowed to execute privileged instructions, such as I/O
- It is not allowed to access the entire memory, in particular the region that contains kernel code and data, i.e. kernel space

Kernel mode when the CPU executes in this mode:

- It is allowed to execute all instructions, including privileged instructions;
- It is allowed to access the entire memory.

The kernel always executes in kernel mode (hence the name)

Issue How can a user process do I/O?

- Via system calls

Traps to the Kernel

- **Upon a system call**, the CPU must change from user mode to kernel mode
  - Once in kernel mode, it can perform privileged instructions
  - Upon returning from a system call, the CPU must change back to user mode

Traps to the Kernel

- **Upon a system call**, the CPU must change from user mode to kernel mode
  - Once in kernel mode, it can perform privileged instructions
  - Upon returning from a system call, the CPU must change back to user mode
- To support this, modern CPUs provide special instructions:

  trap (to kernel) jumps to the kernel code and raises the privilege level to kernel mode:

  1. Save CPU state (PC and a few registers)
     - On the x86, the CPU pushes the PC and some other registers to a per-process kernel stack

     Some of this must be done by HW, rest may be done by SW.
  2. Raise privilege level to kernel mode
  3. Jump to appropriate trap-handler

Traps to the Kernel

- **Upon a system call**, the CPU must change from user mode to kernel mode
  - Once in kernel mode, it can perform privileged instructions
  - Upon returning from a system call, the CPU must change back to user mode
- To support this, modern CPUs provide special instructions:

  trap (to kernel) jumps to the kernel code and raises the privilege level to kernel mode:

  1. Save CPU state (PC and a few registers)
     - On the x86, the CPU pushes the PC and some other registers to a per-process kernel stack

     Some of this must be done by HW, rest may be done by SW.
  2. Raise privilege level to kernel mode
  3. Jump to appropriate trap-handler

  return-from-trap returns to the calling user process:

  1. Lowers the privilege level to user mode (may be done by 2)
  2. Restores the CPU state from before the system call
  3. Jumps to the instruction after the trap in user code

Trap Table

Issue How does the kernel know which code to run upon a trap?

- The caller **must not** provide the specific address (like in a function call)

Trap Table

Issue How does the kernel know which code to run upon a trap?

- The caller **must not** provide the specific address (like in a function call)

Solution Use a table which is initialized by the kernel at boot-time

OS @ boot (kernel mode)	Hardware
initialize trap table	

Trap Table

Issue How does the kernel know which code to run upon a trap?

- The caller **must not** provide the specific address (like in a function call)

Solution Use a table which is initialized by the kernel at boot-time

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of syscall handler

Trap Table

Issue How does the kernel know which code to run upon a trap?

- The caller **must not** provide the specific address (like in a function call)

Solution Use a table which is initialized by the kernel at boot-time

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of syscall handler

- This is similar to an interrupt table
- Upon handling a HW interrupt, the CPU also switches to kernel mode
  - Usually, the interrupt handler needs to do I/O
- On the x86, Linux uses the software interrupt instruction INT as trap instruction (x86 offers other trap instructions)

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main()

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ...

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	
		...

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	
		... Return from `main()`

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	
		... Return from `main()` **trap** (via `exit()`)

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

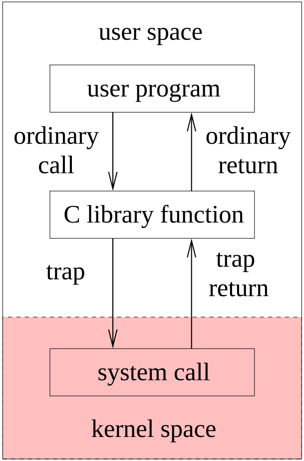
OS (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	
		... Return from `main()` **trap** (via `exit()`)
Free memory of process		

Limited Direct Execution Protocol

Assumption: Upon switching into/out of the kernel, the CPU saves/restores its registers in a **per process kernel stack**

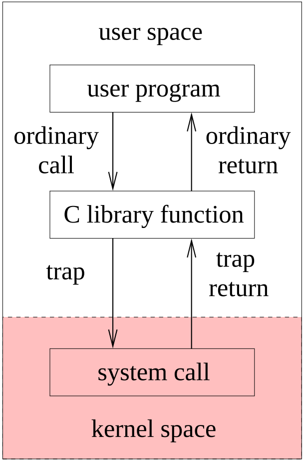
OS (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup stack with argc/argv Fill kernel stack with reg/PC		
	move to user mode restore regs from kernel stack (jump to main)	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	move to user mode restore regs from kernel stack (jump to PC after trap)	
		... Return from `main()` **trap** (via `exit()`)
Free memory of process Remove entry from process list		

System Call Implementation



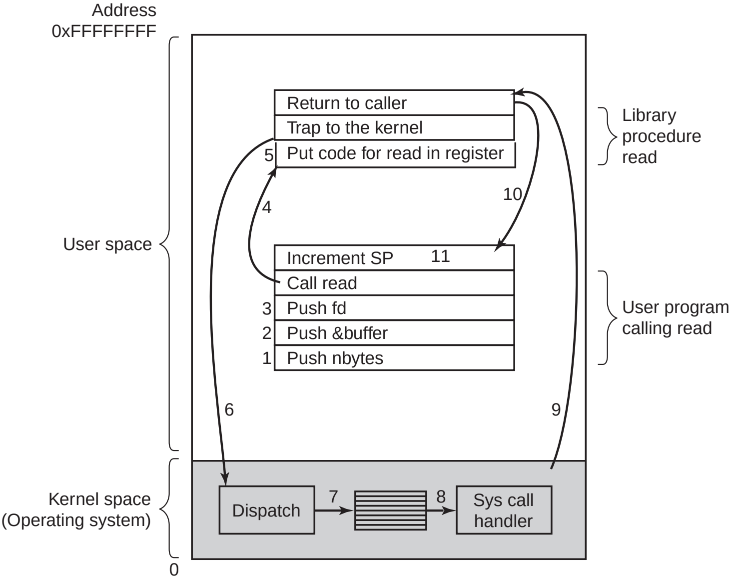
- Uses special HW instructions, e.g. (*call gates* ou *sw interrupts*, in the case of the x86) that switch automatically from one privilege mode to another

System Call Implementation



- Uses special HW instructions, e.g. (*call gates* ou *sw interrupts*, in the case of the x86) that switch automatically from one privilege mode to another
- For a programmer it is as if it had invoked a C library function:
  - Actually, that is what s/he does

```
ssize_t read(int fd, void *buffer,
size_t nbytes)
```



`read()` Execution Steps

1, 2, 3 push the arguments onto the stack;

4 call C library function `read()`;

5 setup register with system call #

6 switch CPU execution mode

7 dispatch to appropriate *handler*;

8 execute *handler*;

9 return back to the C library;

10 return from the C library function `read()`;

11 adjust *stack*.

LDE Issue # 2: Stopping a Running Process

Issue When a process is running the kernel is not

LDE Issue # 2: Stopping a Running Process

Issue When a process is running the kernel is not

Alternatives

Cooperative Provide a system call, e.g. yield

- By calling `yield` a process allows the kernel to run and switch to another process.
- Someone (?Tanenbaum?) wrote that **`nice`** is the least used Unix command
- Malicious or buggy processes may prevent other processes from running

LDE Issue # 2: Stopping a Running Process

Issue When a process is running the kernel is not

Alternatives

Cooperative Provide a system call, e.g. yield

- By calling `yield` a process allows the kernel to run and switch to another process.
- Someone (?Tanenbaum?) wrote that **`nice`** is the least used Unix command
- Malicious or buggy processes may prevent other processes from running

Non-cooperative Use timer interrupts

- Program a timer device to interrupt periodically (typically, a few ms)
- Upon an interrupt (by the timer or otherwise):
  - The currently running process is suspended
  - A pre-configured interrupt handler (IH) runs
  - At this point the kernel runs and can do what it wants

Limited Direct Execution Protocol: Timer Interrupt

- At boot time, the kernel must:

Limited Direct Execution Protocol: Timer Interrupt

- At boot time, the kernel must:
  - initialize the interrupt table, with the interrupt handlers, including the timer interrupt
  - configure the timer to interrupt every X ms
  - start the timer

Limited Direct Execution Protocol: Timer Interrupt

- At boot time, the kernel must:
  - initialize the interrupt table, with the interrupt handlers, including the timer interrupt
  - configure the timer to interrupt every X ms
  - start the timer
- Upon an interrupt, the HW must:

Limited Direct Execution Protocol: Timer Interrupt

- At boot time, the kernel must:
  - initialize the interrupt table, with the interrupt handlers, including the timer interrupt
  - configure the timer to interrupt every X ms
  - start the timer
- Upon an interrupt, the HW must:
  - Save the state of the process that is running (at least processor registers)
    - Process must be able to resume in the point it was, once the kernel has handled the interrupt
    - This is very similar to what is done upon a system call
      No wonder, Linux uses the `INT` instruction

Context Switch

- The kernel, or better the scheduler decides:
  1. Whether to allow the currently running process to continue
  2. Or to suspend it (move it to the READY state) and run another process instead.

Context Switch

- The kernel, or better the scheduler decides:
  1. Whether to allow the currently running process to continue
  2. Or to suspend it (move it to the READY state) and run another process instead.
- In case 2, the kernel must do a context switch:
  - Save the state of the currently running process
    - So that it can be resumed later, as if it had not been preempted
  - Restore the state of the soon-to-be-running process

Context Switch

- The kernel, or better the scheduler decides:
  1. Whether to allow the currently running process to continue
  2. Or to suspend it (move it to the READY state) and run another process instead.
- In case 2, the kernel must do a context switch:
  - Save the state of the currently running process
    - So that it can be resumed later, as if it had not been preempted
  - Restore the state of the soon-to-be-running process
- By switching the kernel stacks, the kernel exits to the context of a process different of the one that was executing when of the call to the switch code

Context Switch

- The kernel, or better the scheduler decides:
  1. Whether to allow the currently running process to continue
  2. Or to suspend it (move it to the READY state) and run another process instead.
- In case 2, the kernel must do a context switch:
  - Save the state of the currently running process
    - So that it can be resumed later, as if it had not been preempted
  - Restore the state of the soon-to-be-running process
- By switching the kernel stacks, the kernel exits to the context of a process different of the one that was executing when of the call to the switch code
- When the kernel executes the return-from-trap instruction, the soon-to-be-running process becomes the currently running process

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A . . .
	timer interrupt	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A)	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap Call `switch`		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap Call `switch` save regs(A) to proc-struct(A)		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap Call `switch` save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B)		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		

	remember addresses of
	    syscall handler
	    timer handler	

start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Handle the trap
Call `switch`
    save regs(A) to proc-struct(A)
    restore regs(B) from proc-struct(B)
switch to k-stack(B)		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		

	remember addresses of
	    syscall handler
	    timer handler	

start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Handle the trap
Call `switch`
    save regs(A) to proc-struct(A)
    restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)		

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		

	remember addresses of
	    syscall handler
	    timer handler	

start interrupt timer		
	start timer interrupt CPU every x ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Handle the trap
Call `switch`
    save regs(A) to proc-struct(A)
    restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)		

	move to user mode	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		

	remember addresses of
	    syscall handler
	    timer handler	

start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Handle the trap
Call `switch`
    save regs(A) to proc-struct(A)
    restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)		

	move to user mode restore regs(B) from k-stack(B)	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap Call `switch` save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	move to user mode restore regs(B) from k-stack(B) (jump to B's PC)	

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	**Hardware**	**Program (user mode)**
initialize trap table		

	remember addresses of
	    syscall handler
	    timer handler	

start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	

Handle the trap
Call `switch`
    save regs(A) to proc-struct(A)
    restore regs(B) from proc-struct(B)
switch to k-stack(B)
return-from-trap (into B)		

	move to user mode restore regs(B) from k-stack(B) (jump to B's PC)	
		Process B . . .

Limited Direct Execution Protocol: Timer Interrupt

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember addresses of syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU every *x* ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A . . .
	timer interrupt save regs(A) to k-stack(A) switch to kernel mode jump to trap handler	
Handle the trap Call `switch` save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	move to user mode restore regs(B) from k-stack(B) (jump to B's PC)	
		Process B . . .

Context Switch

- Note that there are 2 types of register saves/restores

  Upon the timer-interrupt the **hardware** saves the **user registers** of the running process onto the **kernel stack** of that process

  - On the IA32 architecture, this happens every time the processor moves into kernel mode

  On context switching the **kernel**

  1. saves the current values of the **registers** of the running process into the **process structure** of that process
  2. restores the previously saved values of the **registers** of the soon-to-run process from the **process structure** of that process

Context Switch: xv6 code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
    # Save old registers
    movl 4(%esp), %eax    # put old ptr into eax
    popl 0(%eax)          # save the old IP
    movl %esp, 4(%eax)    # and stack
    movl %ebx, 8(%eax)    # and other registers
    movl %ecx,12(%eax)
    movl %edx,16(%eax)
    movl %esi,20(%eax)
    movl %edi,24(%eax)
    movl %ebp,28(%eax)

    # Load new registers
    movl 4(%esp), %eax    # put new ptr into eax
    movl 28(%eax), %ebp   # restore other registers
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp    # stack is switched here
    pushl 0(%eax)         # return addr put in place
    ret                   # finally return into new ctx
```

Kernel Data Structures (xv6 toy-OS)

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

src: **Arpaci-Dusseau & Arpaci-Dusseau**