User program and OS interaction
Multiprocessing

Jinyang Li
User program and OS interaction
Applications, OS, Hardware

Applications
- Firefox
- Safari
- Adobe
- Skype

Operating System
- Windows
- Linux

Hardware
- CPU
- Memory
- I/O
The role of OS

Purpose of the OS software
1. Manage resources among running programs
2. Hide messy hardware details

Concrete jobs of the OS
1.1 Scheduling (give each process the illusion of exclusive CPU use)
1.2 VM management (give each process the illusion of exclusive memory use)
2. file systems, networking, I/O
Process

• Process is an instance of a running program
  – when you type `./a.out`, OS launches a process
  – when you type Ctrl-C, the foreground process is killed
• Each process corresponds to some state in OS
  – process identifier (process id)
  – user id
  – status (e.g. runnable or blocked)
  – saved rip and other registers
  – VM structure (including its page table)

Only OS can modify these data
How to protect the OS from user processes?

• Hardware provides privileged vs. non-privileged mode of execution
  also called
  supervisor/kernel mode
  also called
  user mode

• OS runs in privileged mode
  – can change content of CR3 (points to root page table)
  – can access VA marked as supervisor only
  – ...

• User programs run in non-privileged mode
  – cannot access kernel data structures because they are stored in VA marked as supervisor only
How to get into privileged mode?

Hardware provides 3 controlled mechanisms to switch from non-privileged to privileged execution:

1. Exception
   – e.g. divide by zero, page fault
2. Traps
   – syscalls (user programs explicitly ask for OS help)
3. Interrupt
   – timer, device events such as keyboard process, packet arrival
How to get out of privileged mode?

- OS uses the special hardware instruction `iret`
- OS may return to the same program or context switch to execute a different program
**Syscall: User → OS**

- User programs ask for OS services using syscalls — it’s like invoking a function in OS
- Each syscall has a known number

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
</tr>
<tr>
<td>60</td>
<td>exit</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
</tr>
</tbody>
</table>

C library wraps around these syscalls to provide file I/O

*linux syscall number*
Syscall: user → OS

Assuming OS wants to execute the same process next, but it does not have to.

User code:
- `movq %rax, %rbx`
- `inc %rbx`
- `syscall 2`
- `movq %rax, %r8`
- `add %r8, %r9`
- ...

OS code:
- `iret`
- Code to open the requested file...
OS also takes control upon exceptions

user code

```assembly
addq %rax, %rbx
...
mov (%rbx) %r8
```

OS code

```assembly
... check process VM structure. If VA is legit,
... create page table mapping.
... Otherwise kill process
... iret
```
Multi-processing
Goal of multi-processing

• Run multiple processes “simultaneously”
• Why?
  – listening to music while writing your lab
  – Running a web server, a database server, a PHP program together
Modern CPUs have multiple cores

CPU
PC: 0x00...0058
IR: instruction
GPRs: %rax, ...
%rsp

TLB
CPU Cache

Memory

Your mental model of the CPU as a single core machine
Modern CPUs have multiple cores

CPU core 1
- CPU
- PC: 0x00...0058
- IR: instruction
- GPRs: %rax, %rsp

per-core TLB cache
per-core L1/L2 Cache

CPU core 2
- CPU
- PC: 0x00...0058
- IR: instruction
- GPRs: %rax, %rsp

per-core TLB cache
per-core L1/L2 Cache

shared L3 Cache
Memory
How to multi-process?

• Execute one process exclusive on each core?
  – 2 cores → 2 processes only 😞

• How to “simultaneously” execute more processes than there are cores?
Multiprocessing
(on a single core machine)

**Process 1**
- stack
- heap
- data
- code
- PCB

**Process 2**
- stack
- heap
- data
- code
- PCB

User
---
OS

Process Control Block (PCB) stores saved register values

CPU
- PC: 0x00...0058
- IR: instruction
- GPRs: %rax, %rsp

Memory state

CPU state
Context switch

1. timer interrupt

   1. timer interrupt
   ...
   ...
   4. context switch to where P1 previously left off

   • decide it’s P’s turn
   • save current process’ CPU state
   • restore P’s saved CPU state
   iret

2. context switch to where P2 previously left off

3. timer interrupt

   3. timer interrupt
   ...
   ...
   ...

OS code

Process P1

Process P2

every 10ms
Creating and killing processes

- One process creates another process via syscall fork()
  - All processes are created by some processes (a tree). The first process is a special one (init) and is created by OS.
  - Launch a program via command-line (the shell program creates the process)
The fork syscall

- Creates a new child process (almost completely) identical to the parent process
- Same code, data, heap, stack, register state except different return values of the fork syscall
- Returns child process’s id in parent process
- Returns zero in the child process

“called once, returned twice”
Example fork call

```c
void
main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent, child pid=%d\n", pid);
    }
}
```
Example fork call

void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...
");
  }
}
Example fork call

void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}

process 1

void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}

process 2
Example fork call

process 1

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...");
    }
}
```

process 2

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...");
    }
}
```
Example fork call

process 1

```c
void main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...
");
  }
}
```

process 2

```c
void main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...
");
  }
}
```
Example fork call

process 1

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}
```

output:

```
In parent...
```

process 2

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}
```
Example fork call

process 1
void
main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}

output:
In parent...

process 2
void
main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}
Example fork call

process 1

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}
```

process 2

```c
void main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...
");
    }
}
```

output:

```
In parent...
```
Example fork call

process 1

```c
void
main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...\n");
    }
}
```

process 2

```c
void
main() {
    pid_t pid = fork();
    assert(pid >= 0);
    if (pid == 0) {
        printf("In child");
    } else {
        printf("In parent...\n");
    }
}
```

output:

```
In parent...
In child
```
Notes on fork

• Execution of parent and child are concurrent
  – interleaving is non-deterministic.
  – In the example, both outputs are possible

  In parent...
  In child
  In child
  In parent...

• Parent and child have separate address space
  (but their contents immediately after fork are identical)
Another fork example

```cpp
void main()
{
  printf("hello\n");
  fork();
  printf("world\n");
  fork();
  printf("bye\n");
}
```

How many processes are created in total?
void main()
{
    L1: printf("hello\n");
    L2: fork();
    L3: printf("world\n");
    L4: fork();
    L5: printf("bye\n");
}

What are the possible printouts?

✔ hello world world bye bye
✔ hello world world bye bye
✗ hello world world bye bye bye
Exercise

```c
void main()
{
    L1: printf("hello\n");
    L2: if (fork() == 0) {
        L3: printf("big\n");
        L4: if (fork() == 0) {
            L5: printf("world\n");
        }
    }
    L6: printf("bye\m");
}
```

What are the possible printouts?

- ✔ hello
- ✔ big
- ✔ world
- ✔ bye
- ✔ bye
- ✔ bye

- ✗ hello
- ✗ bye
- ✗ big
- ✗ big
- ✗ bye
- ✗ bye
- ✗ world

Diagram:

```
L1 ---- L2 ---- L3 ---- L4 ---- L5 ---- L6
     |            |            |
     |            |            |
     |            |            |
     |            |            |
     L6
```

L1: hello
L4: big
L5: world
L6: bye
wait: synchronize with child

• Parent process could wait for the exit of its child process(es).
  – int waitpid(pid_t pid, int * child_status, ...)

• Good practice for parent to wait
  – Otherwise, some OS process state about the child cannot be freed even after child exits
  – leaks memory
Exercise

void main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("child");
  } else {
    printf("parent");
  }
}

What are the possible printouts?

✔ child
✔ parent
Exercise

void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("child");
  } else {
    waitpid(pid, NULL, 0);
    printf("parent");
  }
}

What are the possible printouts?

✔ child  ❌ parent

✔ parent  ❌ child
execv: load program in current process

• int execv(char *filename, char *argv[])
  – overwrites code, data, heap, stack of existing process (retains process pid)

• called once, never returns
Example

```c
void main() {
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        execv("/bin/echo", "hello");
        printf("world\n");
    }
    waitpid(pid, NULL, 0);
    printf("bye\n");
}
```

How many processes are created in total? output?

2

hello bye