User program and OS interaction
Multiprocessing

Jinyang Li
What we’ve learnt so far

• Machine instructions
  – compiler translates C to x86 instructions
  – x86 instructions are executed by CPU hardware only

• Dynamic memory allocator
  – realized as a library implementation

• Virtual memory
  – each process has its own virtual address space
  – VM is realized by a combination of hardware mechanism and OS implementation
    • MMU performs address translation
    • OS populates page table
Today’s lesson plan

1. Interaction between user programs and OS
2. Multiprocessing
Interaction between user programs and OS

I mean OS kernel
Applications, OS, Hardware

Applications

Operating System

Hardware

I/O
CPU
Memory
Applications, OS, Hardware

Applications

Operating System

Hardware

I/O  CPU  Memory
The role of OS

What does the OS do?

1. Resource management
   - scheduling: give each process illusion of exclusive CPU use
   - VM management: give each process illusion of exclusive memory use

2. Hide messy hardware details
   - file system
   - networking
Process

• Process is an instance of a running program
  – when you type `.a.out`, OS launches a process
  – when you type Ctrl-C, OS kills the process
• OS maintains some state for each process
  process identifier (process id)
  – user id
  – status (e.g. runnable or blocked)
  – saved rip and other registers
  – VM structure (including its page table)
How to protect the OS from user processes?

• Hardware provides privileged vs. non-privileged mode of execution
  also called supervisor or 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. Traps
   - syscalls (user programs explicitly ask for OS help)
2. Exception (caused by the current running program)
   - e.g. divide by zero, page fault
3. Interrupt (caused by external events)
   - timer, keyboard press, 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
#1 Traps: 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>Syscall</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 these syscalls to provide file I/O

Linux syscall number
Syscall: user → OS

Assuming OS wants to execute the same process next; it does not have to
#2 exceptions: OS takes control upon exceptions

Hardware exception because %rbx contains a non-readable address

User code:
```
addq %rax, %rbx
...
```
```
mov (%rbx) %r8
...
```

OS code:
```
check process VM structure. If VA is legit, create page table mapping. Otherwise kill process
```
```
iret
```
#3 interrupts:
OS takes control upon interrupts

Interrupts due to packet arrival from the network card

User code:
```
addq %rax, %rbx
... ...
... ...
... ...
... ...
... ...
```

OS code:
```
... ...
... ...
... ...
... ...
... ...
... ...
iret
```

Process packets:
e.g. send acknowledgment packets
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

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
(e.g. on a single core machine)

Process Control Block (PCB) stores process meta-data, e.g. process id, saved register values
Context switch

Process P1

1. timer interrupt

... ...

4. context switch to where P1 previously left off

Process P2

2. context switch to where P2 previously left off

3. timer interrupt

every 10ms

OS code

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

... iret...
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.
  – When launching a program via command-line, the shell program creates the process
The fork syscall

• OS 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\n");
    } else {
        printf("In parent, child pid=%d\n", pid);
    }
}
```
Example fork call

process 1

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

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

process 1

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

process 2
Example fork call

process 1

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

process 2

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

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

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

process 1

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

output:

In parent...

process 2

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

process 1

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

output:

```
In parent...
```

process 2

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

process 1

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

output:

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

process 2

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

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

• Parent and child have separate address space
  (but their contents immediately after fork are identical)
Execution of parent and child are concurrent

```c
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 bye bye world bye
✗ hello world bye bye world bye
Yet another example

```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
- ✔ hello big bye bye
- ✗ hello bye big bye bye world
Parent and child have separate address space with (initially) identical content

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

What are the possible printouts?

- ✔ child 1
- ➕ parent 1
- ✗ child 1
- ✗ parent 2
- ✗ parent 1
- ✗ child 2

parent

Total=0
void main()
{
    int total = 0;
    pid_t pid = fork();
    assert(pid >= 0);
    total++;
    if (pid == 0)
        printf("child %d\n");
    else
        printf("parent %d\n");
}
Parent and child have separate address space with (initially) identical content

```c
void main()
{
    int total = 0;
    pid_t pid = fork();
    assert(pid >= 0);
    total++;
    if (pid == 0)
        printf("child %d\n");
    else
        printf("parent %d\n");
}
```

What are the possible printouts?

✔ child 1
  parent 1

✗ child 1
  parent 2

✗ parent 1
  child 2

- Physical memory
- total=0
- child
- parent
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

What are the possible printouts?

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

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

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

Never executed because execv has replaced process’s memory with that of the echo program
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");
}
```
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");
}
```
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");
}
```