CS153: Process 2

Chengyu Song

Slides modified from Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

- **Lab**
  - New TA Bojian Du, office hours Th 3:30pm-5:30pm at WCH 465
  - Please participate in the lab session poll
  - Groups: will assign a partner if you don't have one by Friday
Recall: process creation

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Loads the program into the address space (optional)
- Initializes the address space and hardware state
- Initializes other OS resources
- Places the PCB on the ready queue
Process creation: Windows

BOOL WINAPI CreateProcess(
    _In_opt_    LPCTSTR               lpApplicationName,
    _Inout_opt_ LPTSTR                lpCommandLine,
    _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_        BOOL                  bInheritHandles,
    _In_        DWORD                 dwCreationFlags,
    _In_opt_    LPVOID                lpEnvironment,
    _In_opt_    LPCTSTR               lpCurrentDirectory,
    _In_        LPSTARTUPINFO         lpStartupInfo,
    _Out_       LPPROCESS_INFORMATION lpProcessInformation
);

- **CreateProcess** will **loads** the program
**Process creation: Unix**

- In *nix, processes are created using `fork()`
  - No arguments, try `man fork`
    - Program loading is done by `exec`
- Creates and initializes a new PCB
- Creates a new address space
- Inherit all the resources its parent has
  - Virtual memory (content only, different physical page)
  - Opened files (different file descriptor table)
fork()

• What would this program print?

```c
int main(int argc, char *argv[]) {
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```
Example output

[sledge ~]$ gcc test.c -o test
[sledge ~]$ ./test
My child is 486
Child of test is 486
Duplicating address spaces

Parent

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Child

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```
Divergence

```
child_pid = 486
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

```
child_pid = 0
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```
Preview: copy-on-write

• Q: copying memory content is slow, so how can we improve the efficiency

• Sharing the same physical resources is faster, but we need the ability to *diverge*

• Insight: content can be shared as long as it remains the same

• Copy-on-write: sharing by default, diverge on write
Example continued

[sledge ~]$ gcc test.c -o test
[sledge ~]$ ./test
My child is 486
Child of test is 486
[sledge ~]$ ./test
Child of test is 498
My child is 498

• Why is the output in a different order?
Why fork()?

• Very useful when the child ...
  • Is cooperating with the parent
  • Relies upon the parent’s data to accomplish its task

• Example: Web server

```
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        // Handle client request
    }
}
```
Process creation: Unix (2)

- Wait a second. How do we actually start a new program?
  - `int exec(char *prog, char *argv[])`
  - `exec()`
    - Stops the current process
    - Release most OS resources like virtual memory, except file descriptors
    - Loads the program "prog" into the process' address space
    - Initializes hardware context and args for the new program
Process creation: Unix (3)

- `fork()` can return an error. Why might this happen?
  - Try `man clone`

- What does it mean for `exec()` to return?

- What does it mean for `exec()` to return with an error?
  - Try `man execve`

- What happens if you run `exec sh` in your shell?

- What happens if you run `exec ls` in your shell? Try it.
Process termination

- All good processes must come to an end. But how?
  - Unix: `exit(int status)`, Windows: `ExitProcess(int status)`
- Essentially, free resources
  - Terminate all threads (coming up)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note: a process does not need to clean up itself **when terminating**
Unix shell

while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        // Manipulate STDIN/OUT/ERR file descriptors for pipes, redirects
        exec(cmd);
        panic("exec failed");
    } else {
        if (!(run_in_background))
            waitpid(child_pid);
    }
}


Parallel programs

• Recall: concurrency vs. parallel

• On a multiprocessor system, our Web server example that forks multiple copies of itself to handle multiple requests *simultaneous*

• To execute these programs we need to
  • Create several processes that execute in parallel
  • *Explicitly* setup some shared resources (e.g., memory) for communication
  • Have the OS schedule these processes in parallel

• Problem: *inefficient*
Recall process

• Process is the entity of resources allocation
  • Address space, machine states, OS resources, etc.
  • By default, processes are completely isolated from each other
  • Creating a new process is costly because of all of the data structures that must be allocated and initialized
  • Communicating between processes is costly because most communication goes through the OS
    • Overhead of system calls and copying data
Rethinking processes

• What is similar in these cooperating processes in a parallel program?
  • They all share the same code and data (address space)
  • They all share the same privileges
  • They all share the same resources (files, sockets, etc.)

• What don’t they share? - Each has its own execution state: PC, SP, and registers

• **Key idea**: separate resources from execution state
Modern OS (Mac OS, Windows, Linux) separate the concepts of processes and threads.

- The **thread** defines a sequential execution stream (PC, SP, registers).
- The **process** defines the address space and general process attributes (the subject abstraction).

- A thread is bound to a single process.
  - Processes, however, can have multiple threads.

- Threads become the unit of scheduling.
Processes and threads

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One Thread/Process</th>
<th>One Thread/Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Address Space</td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td>(MSDOS)</td>
<td>(Early Unix)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Many Threads/Process</th>
<th>Many Threads/Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Address Space</td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td>(Pilot, Java)</td>
<td>(Mac OS, Unix, Windows)</td>
</tr>
</tbody>
</table>
Summary

• How a new process is created
  • New PCB and address space
  • OS resources
    • Difference between CreateProcess and fork
• How a process is terminated
  • PID, address space, static states, dynamic states
• Limitation of the process abstraction
  • Why thread is invented
Additional question to ponder

• When process would be better?
• Which abstractions do browsers use, process or thread, why?
• Which abstractions do web server use, why?
• Hint: isolation
For next class ...

- Scheduling
- Continue to get familiar with the lab environment
- Textbook
  - Module 7, 8, 9