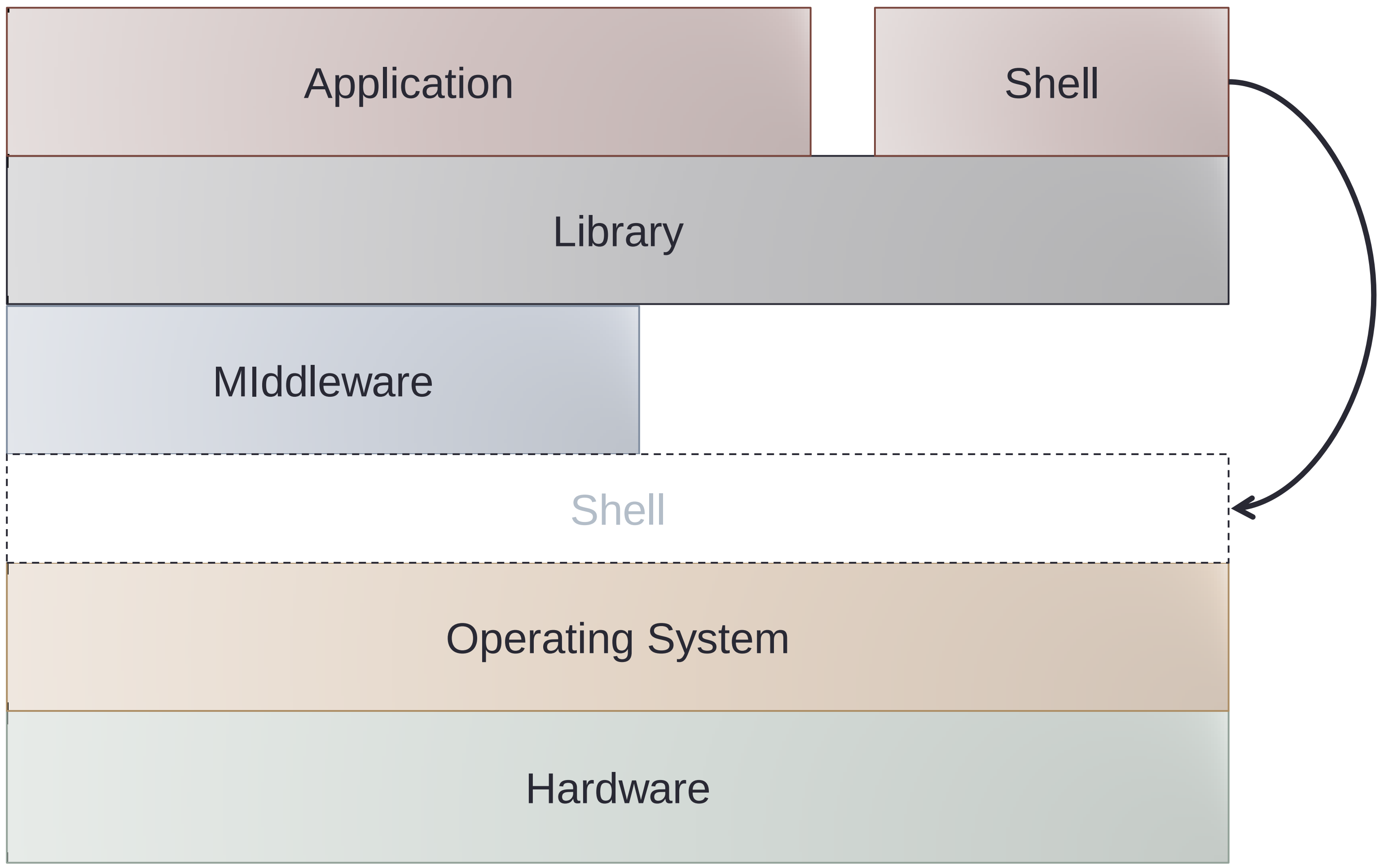
SOFTWARE ARCHITECTURE
3. SHELL

Tatsuya Hagino
hagino@sfc.keio.ac.jp

slides URL

https://vu5.sfc.keio.ac.jp/sa/login.php

Software Layer

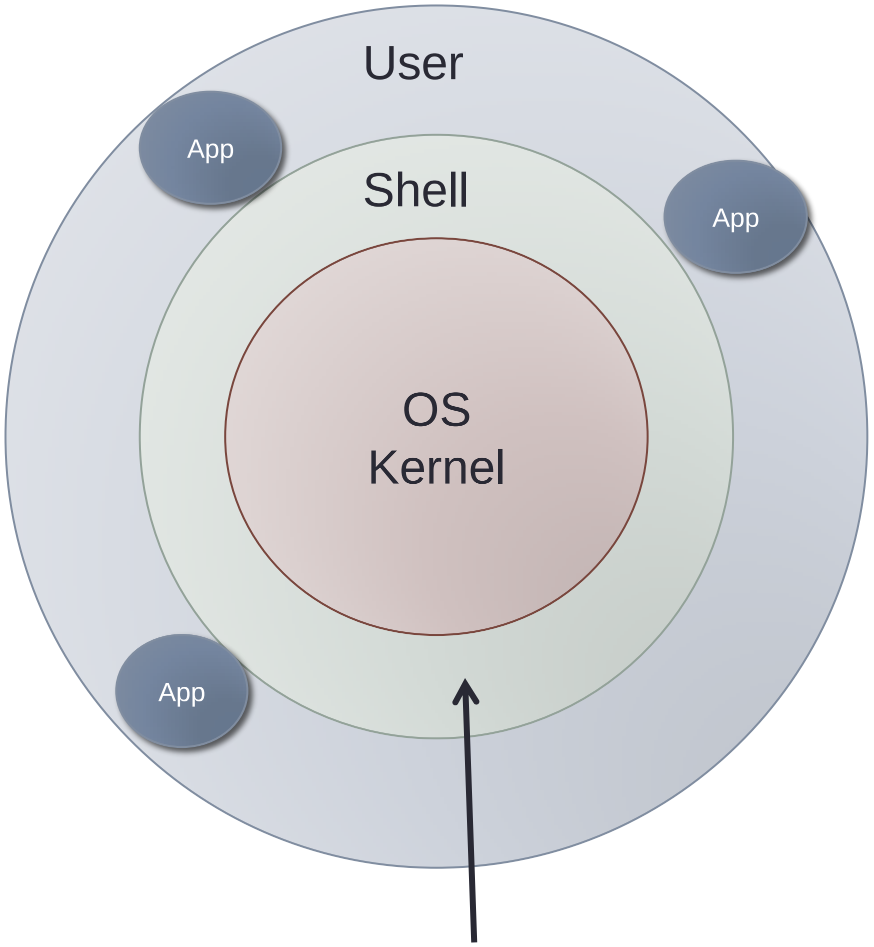


Functions of Shell

• Start programs • Control running programs • background • foreground • Input/output redirection for programs • pipe	Starting and Controlling Programs
• Set environment variables and shell variables	Setting up execution environment
• Expanding wild cards for command line • History of commands • Alias for commands • File name completion	Useful functions
• Repeat and conditional execution • Shell script	Running scripts

Shells

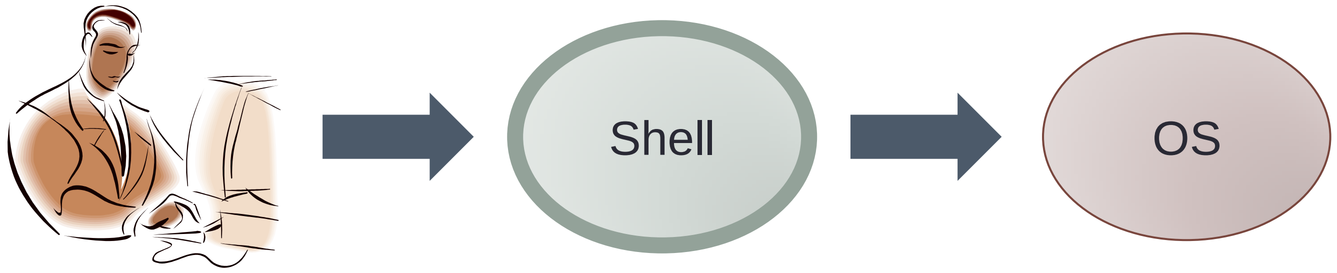
- Windows
  - Command Prompt
  - PowerShell
  - Explorer
- Mac
  - Finder
  - Launcher
- UNIX
  - sh
    - Bourne Shell
    - Korn Shell
    - Z Shell
  - csh
    - C Shell
    - TENEX C Shell



Shell between OS and user

UNIX Shell Commands

- Shell communicates your instruction to OS



- Very few built-in commands
  - **`set`**, **`alias`**, **`cd`**, **`setenv`**, etc.

- Most of commands are ordinary programs.
  - **`ls`** is a program to list files in a directory.
  - **`cat`** is a program to output contents of files.

Shell Command Processing

```
      shell() {
        char buf[512], char *argv[512];
        for (;;) {
(1) →     printf("% ");
(2) →     if (!fgets(buf, sizeof(buf), stdin)) break;
(3) →     parse(buf, argv);
(4) →     execute(argv[0], argv);
        }
      }
```

1. Write a prompt.
2. Read one line from a terminal.
3. Separate the line into a command and its arguments separated by spaces.
4. Execute the command with the arguments.
5. Back to the next input.

Execute a Program

- Create a child process and execute a program.

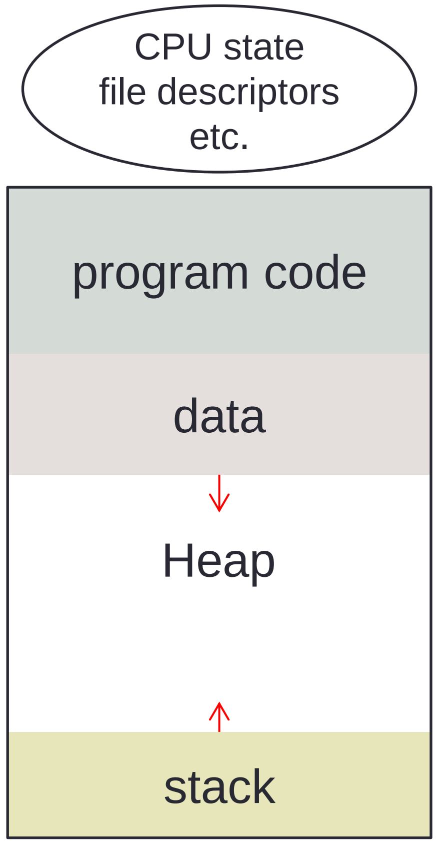
```
execute(char *cmd, char *argv[]) {
  int pid, status;
  pid = fork();
  if (pid == 0) {
    execve(cmd, argv, NULL);
    fprintf(stderr, "command %s not found¥n", cmd);
    exit(1);
  }
  while (wait(&status) != pid);
}
```

- Shell is the parent process and waits for the child process to finish.

Process

- Running state of a program
  - Multiple processes may run the same program.

- Each process consists of:
  - program
  - CPU state (registers, PC, SP)
  - data
  - memory space
  - file descriptors
  - other process related states

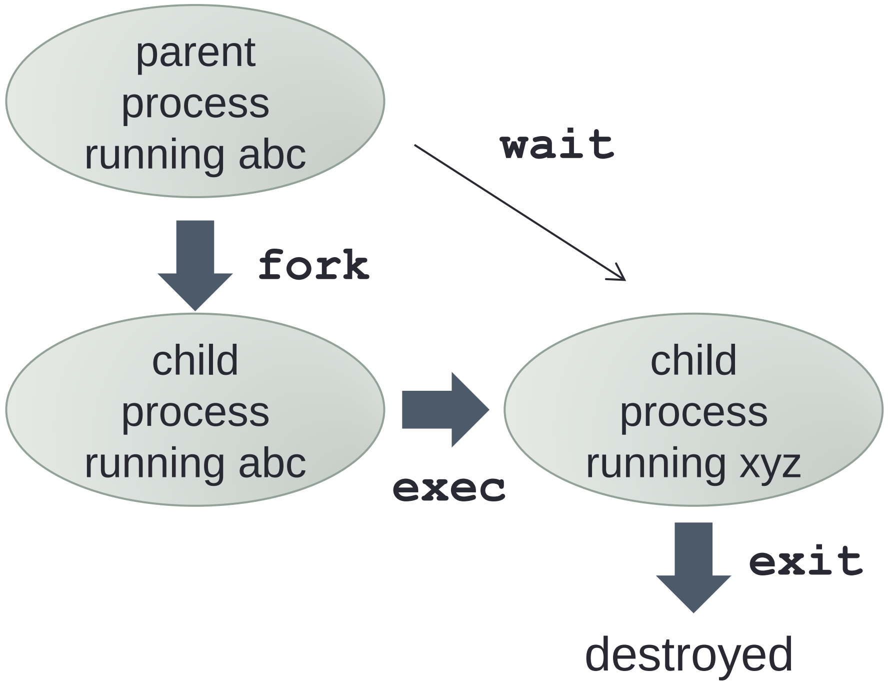


memory space

Process related System calls

- **`fork`**
  - Create a child process.
  - Exact copy of the parent
- **`exec`**
  - Specify a program to execute.
  - Current program is destroyed and replaced with the new one.
- **`wait`**
  - Wait until a child process terminates.
- **`exit`**
  - Terminate the current program.

- other system calls
  - **`signal`**
    - Specify a handler for each interrupt.
  - **`kill`**
    - Send an interrupt.


parent process running abc
wait
fork
child process running abc
exec
child process running xyz
exit
destroyed

parent-child relation in processes

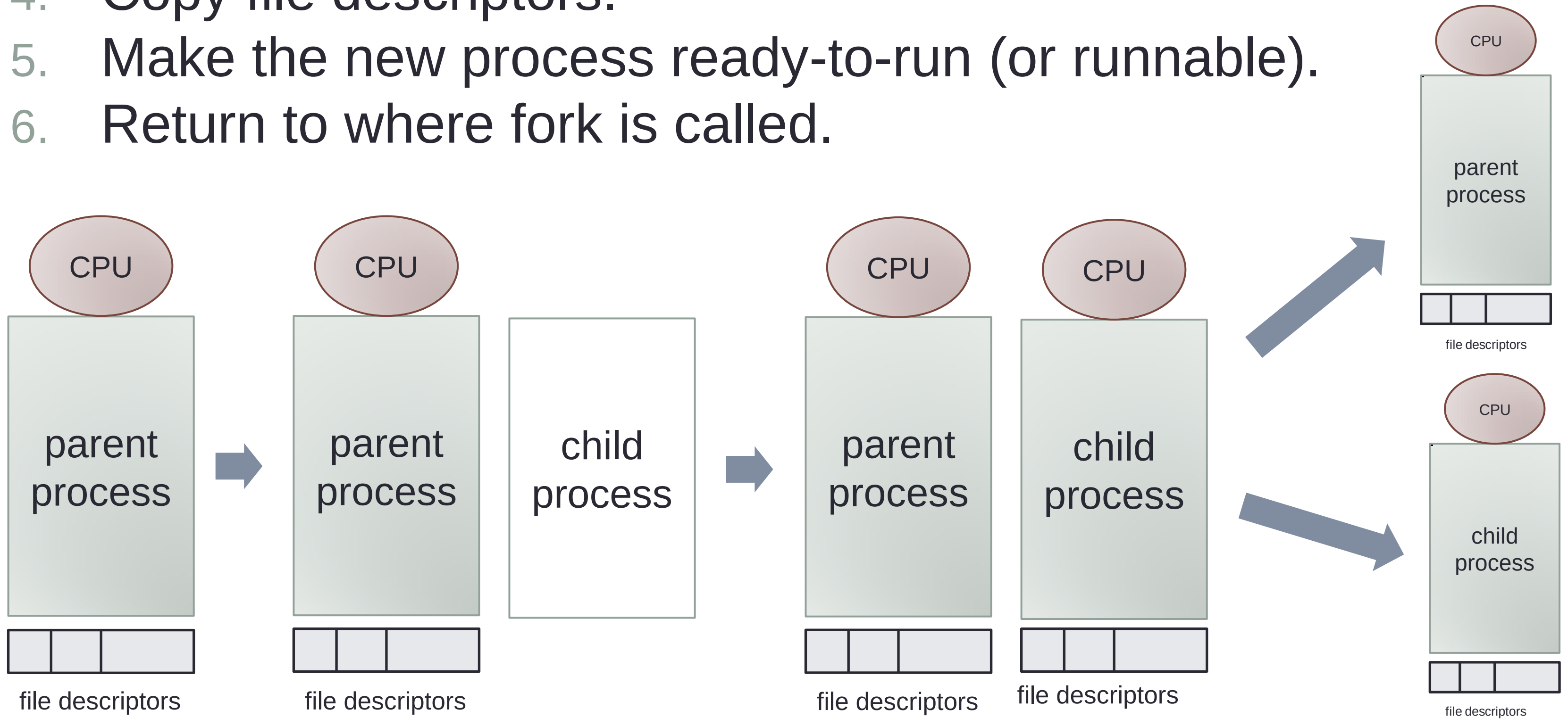
fork and exec

- fork
  - Inherit all from the parent.
    - memory space
      - program
      - data
      - stack
    - file descriptors
    - environment variables
  - Memory spaces are not shared, but copied.
    - *copy-on-write*
    - If either one changes, it is actually copied.

- exec
  - Independence from the parent
    - Destroy the current content of the memory.
    - Replace it with the new program image.
  - Start the new program from its entry point.
    - main
  - What inherits.
    - file descriptors
    - arguments to exec
      - including environment variables
  - The program image is not loaded into the memory immediately.
    - *demand paging*
    - loaded when necessary

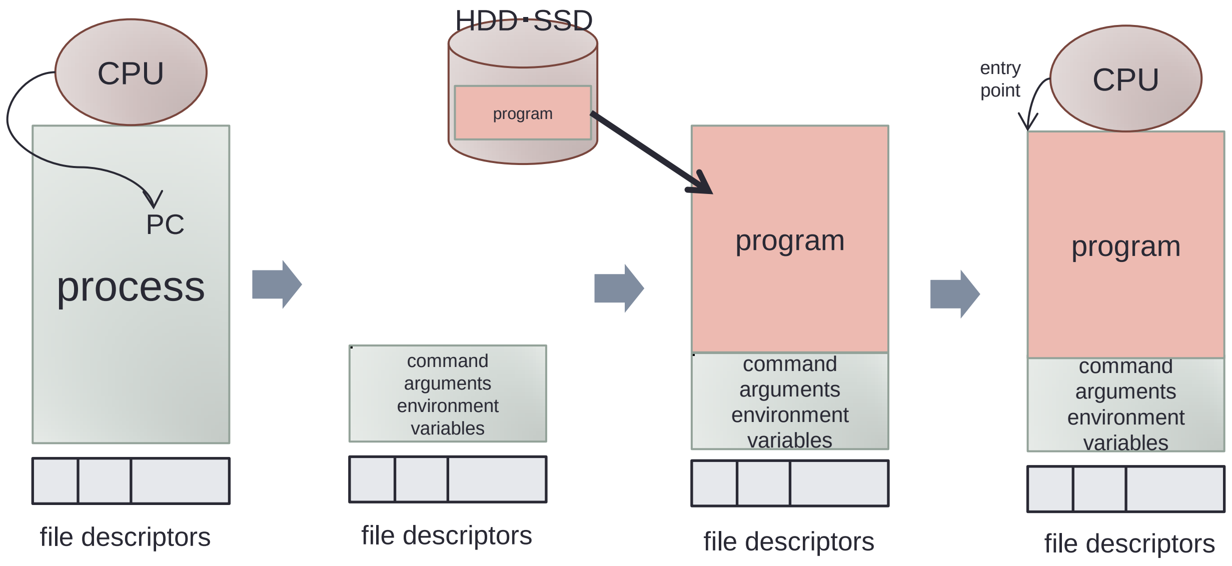
How **fork** works

1. Create a new process and make it a child.
2. Copy CPU state.
   - registers, PC, SP, etc.
3. Copy memory contents.
   - use copy-on-write sharing
4. Copy file descriptors.
5. Make the new process ready-to-run (or runnable).
6. Return to where fork is called.



How **exec** works

1. Allocate a temporary memory space.
2. Copy the command arguments and the environment variables to the temporary memory space.
3. Release all the memory of the current process and create an empty memory space.
4. Set up demand paging of the given program on the new memory space.
5. Copy the command arguments and the environment variables from the temporary memory space to the stack in the new memory space.
6. Release the temporary memory space.
7. Set CPU PC to the entry point.



Command **PATH**

- The first argument of **`execve`** is the path name of the command.
  - **`execve("ls", ...)`** does not work.
  - **`execve("/bin/ls", ...)`** does work.

- Writing full path name is tedious.

- Use an environment variable '**`PATH`**' which contains a list of directories where commands are.

```
/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin
```

- Try each directory to find the command.

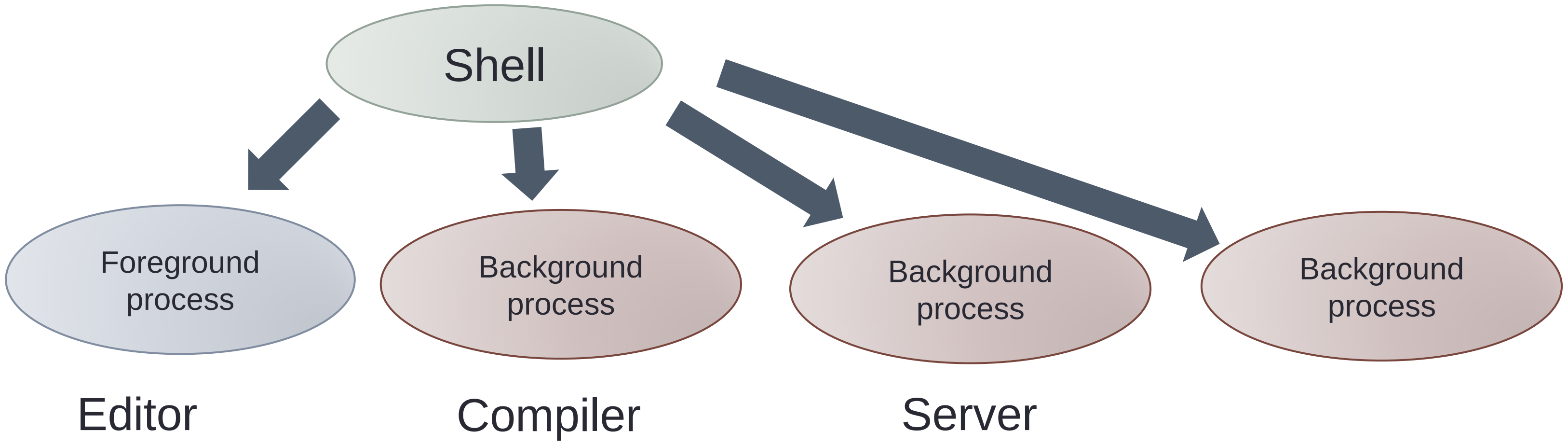
```
execute(char *cmd, char *argv[]) {
  int pid, status;
  pid = fork();
  if (pid == 0) {
    execve("/bin/" + cmd, args, NULL);
    execve("/usr/bin/" + cmd, args, NULL);
    execve("/sbin/" + cmd, args, NULL);
    execve("/usr/sbin/" + cmd, args, NULL);
    execve("/usr/local/bin/" + cmd, args, NULL);
    fprintf(stderr, "command %s not found¥n", cmd);
    exit(1);
  }
  while (wait(&status) != pid);
}
```

After a successful execve the rest will not be executed.

Background and Foreground

- Foreground process
  - Usual execution of a program
  - Execute one by one.

- Background process
  - Put '&' at the end of command.
  - Execute the next command without waiting termination of the current one.
  - Multiple commands can be executed as background.



Implementation of Background

- Shell waits termination of child process with ‘**`wait`**’ system call.
- For background processes, shell does not wait for the termination.

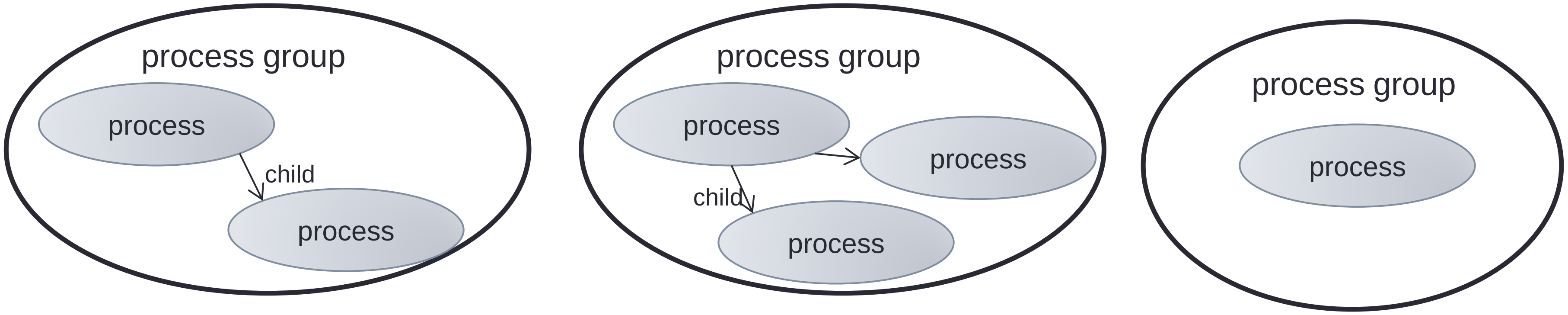
```
execute(char *cmd, char *argv[], int foreground) {
   int pid, status;
   pid = fork();
   if (pid == 0) {
      execve(cmd, argv, NULL);
      fprintf(stderr, "command %s not found¥n", cmd);
      exit(1);
   }
   if (foreground) {
      while (wait(&status) != pid);
   }
}
```

Job Control

- A job may consists of more than one process.
  - Connect commands with pipe.
  - A command may `fork` to create children.

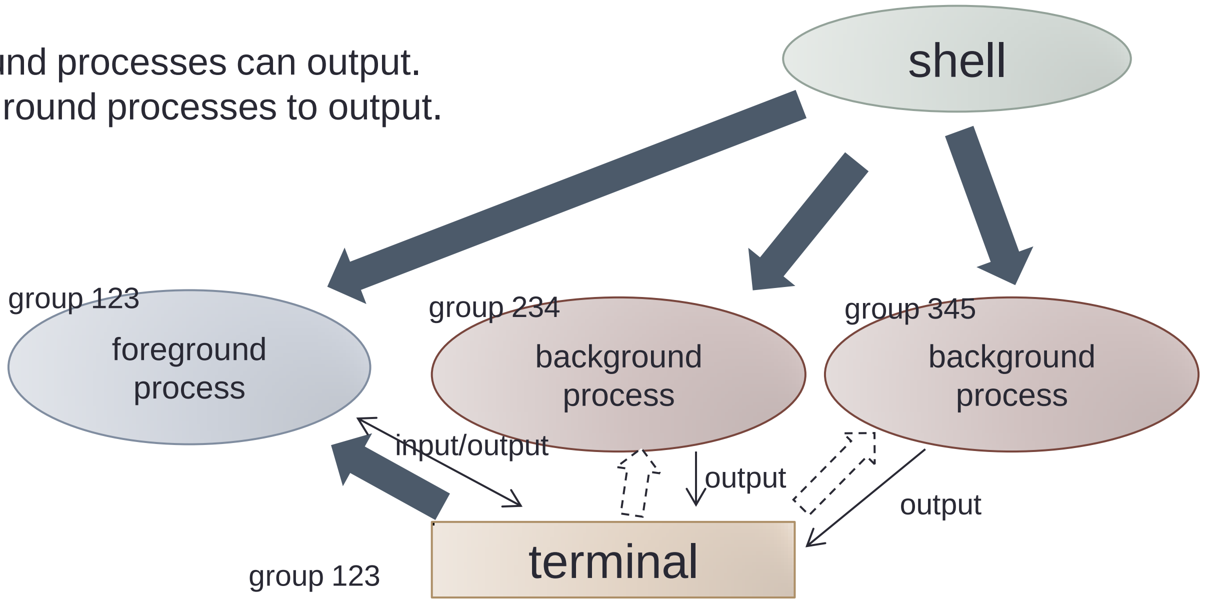
- A *job* is a group of processes.
  - Foreground and background are controlled for jobs (not for each process).

- UNIX uses *process group*.
  - Each process belongs to a process group.
  - Child processes belong to the same process group.
  - Shell creates a new process group for each command.



Job Control for a Terminal

- Each terminal holds one process group.
  - can be set by `ioctl` with `TIOCSPGRP` parameter.
- Foreground
  - terminal process group = process process group
  - Terminal switches process group.
- Input from terminal
  - Sent to foreground processes.
  - Background processes stop when they try input from terminal.
- Output to terminal
  - Foreground and background processes can output.
  - Possible to prohibit background processes to output.



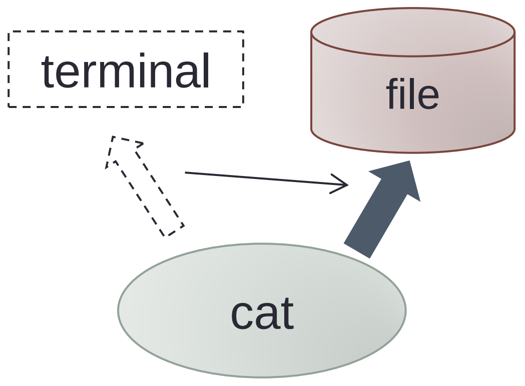
Redirection

- Redirect standard input/output/error to files.

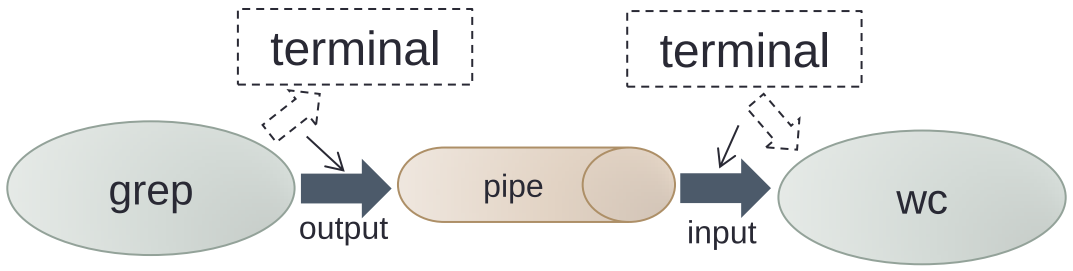
```
% cat /etc/passwd > /tmp/aaa
% wc < /etc/passwd
```

- *Pipe* can combine two commands.
  - Output of one command is connected to input of the other.

```
% grep abc /etc/passwd | wc
```



redirect standard output



connect commands using pipe

Implementation of Redirection

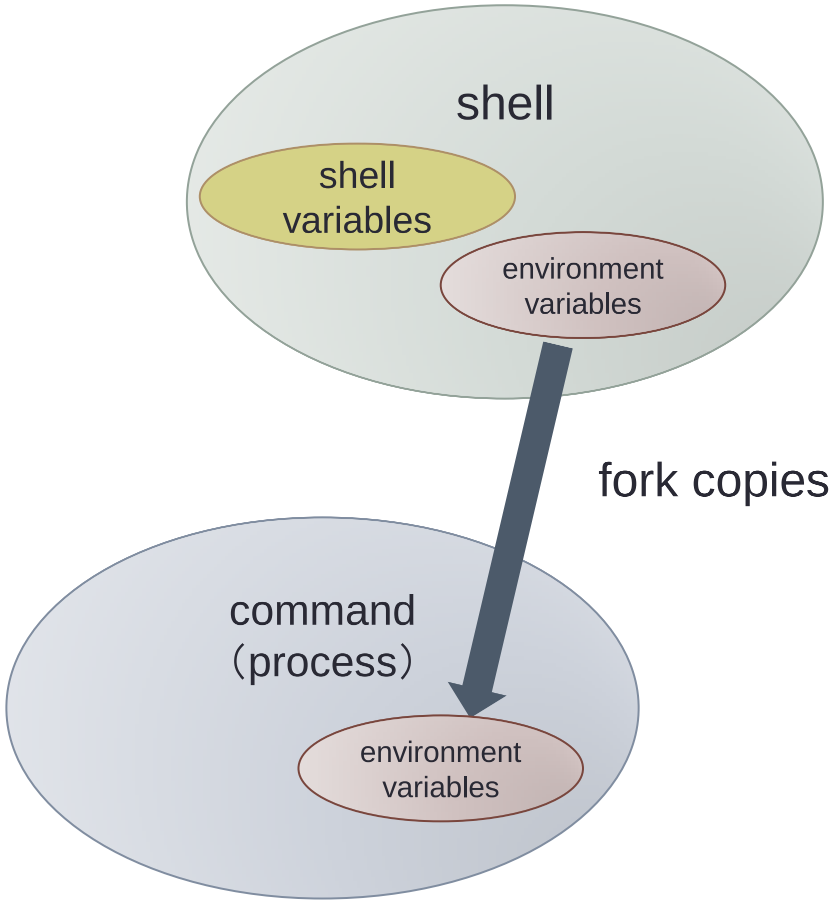
- File descriptors are inherited by **execve**.
- Set file descriptors before **execve**.

```
execute(char *cmd, char *argv[]) {
  int pid, status;
  pid = fork();
  if (pid == 0) {
    fd = open("/etc/passwd", O_RDONLY);
    dup2(fd, 0);
    close(fd);
    execve(cmd, argv, NULL);
    ...
  }
  while (wait(&status) != pid);
}
```

- **dup2** copies file descriptors.
- If changed before **fork**, shell's file descriptors are also changed.
- **fork** and **exec** need to be separated.

Shell variables and Environment varaibles

- Shell variables
  - Used by shell
  - Change shell behavior
  - Often used in shell scripts.
- Environment variables
  - Inherited to commands
    - User name
    - Home directory
    - PATH



Wild Card

- Wide card ‘`*.c`’ can be used to specify multiple files.
- ‘`*.c`’ is expanded by shell

```
% ls *.c
% cat a???.b?
```

```
DIR *dp;
struct dirent *de;
dp = opendir(dir);
while (de = readdir(dp)) {
  if (match(name, de->d_name)) {
    strcpy(argv[argc++], de->d_name);
  }
}
closedir(dp);
```

```
int match(char *pattern, char *p) {
  char ch;
  while (ch = *pattern++) {
    if (ch == '*') {
      while (*p) {
        if (match(pattern, p)) return 1;
        p++;
      }
      return (*pattern == 0);
    }
    else if (ch == '?') {
      if (*p++ == 0) return 0;
    }
    else if (*p++ != ch) return 0;
  }
  return (*p == 0);
}
```

- matching of ‘`*`’ and ‘`?`’

Shell Script

```
test
```

```
echo Hello
date
ls
```

← **`chmod a+x test`**

Mark it as executable

a new command ‘ test’

- Create a new command combining some commands:
  - Create a text file with commands.
  - Mark it as executable using **`chmod`**.
  - OS execute a shell if the given file is not binary.
  - Conditional braches and repetitions are allowed.
  - Shell can be specified in the first line.

```
test
```

```
#!/bin/csh -f
echo Hello
date
ls
```

Summary

- Functions of shell
  - Execute commands
  - Job control
  - Redirection
  - Environment variables
  - Wide card
  - Shell script

- Other functions
  - Command alias
  - File name completion
  - Command history