CPU affinity

- A process have CPU affinity i.e. the CPU on which process is likely to execute.
- This affinity is stored in its PCB (task_struct).
- This affinity is inherited to the child process (from its parent process).
- The CPU affinity can be modified using taskset command and/or sched_setaffinity() syscall.
- taskset command
    - terminal> sudo taskset cpu_affinity_mask ./command
        - cpu_affinity_mask=0x00000001 --> CPU0
        - cpu_affinity_mask=0x00000002 --> CPU1
        - cpu_affinity_mask=0x00000003 --> CPU0 or CPU1
        - cpu_affinity_mask=0xFFFFFFFF --> any CPU
    - terminal> sudo taskset 0x00000001 ./demo12.out

exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- Example:

```
ret = fork();
if(ret == 0) {
    err = execl("/usr/bin/ls", "ls", ..., NULL);
    if(err < 0) {
        perror("execl() failed");
        _exit(1);
    }
}
else {
    wait(&staus);
}
```

- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
    - execl(), execlp(), execle(),
    - execv(), execvp(), execvpe()
    - execve() -- system call
- exec() family multiple functions have different syntaxes but same functionality.
    - l: variable argument List

```
    // ls -l -a /home
    err = execl("/usr/bin/ls", "ls", "-l", "-a", "/home", NULL);
    // execl("exeutable path", ..., NULL);
```

- v: argument Vector (array)

```
    // ls -l -a /home
    char *args[] = { "ls", "-l", "-a", "/home", NULL };
    err = execv("/usr/bin/ls", args);
    // execl("exeutable path", args_vector);
```

- p: find executable in the Path

```
    env

    echo $PATH
```

- PATH contains set of directories separeted by ":".
- When any program/command is executed on shell without giving it's full PATH, shell search it automatically in all directories given in PATH variable.
- execlp() or execvp() automatically search executable in all directories in the PATH variable i.e. the first arg need not to be full path of executable.

```
err = execlp("ls", "ls", "-l", "-a", "/home", NULL);
// OR
err = execvp("ls", args);
```

- e: pass Environment variables to the child program.
  - Env variables contains important information about the system e.g. PATH, HOME, USER, SHELL, etc.

```
    env
    echo $USER
    echo $SHELL
```

- To access Environment variables in a C program, use 3rd of main().

```
    int main(int argc, char *argv[], char *envp[]) {
        int i;
        for(i=0; envp[i]!=NULL; i++)
            puts(envp[i]);
```

```
        return 0;
    }
```

- execve(), execle() and execvpe() can pass Environment variables to the child program.

```
    char *env_array[] = { "USER=test", "SHELL=/bin/csh",
"HOME=/home/test" };
    err = execle("/child.out", "child.out", ..., NULL, env_array);
    // OR
    err = execve("/child.out", args, env_array);
```

Process creation

- fork() syscall
  - allocate pcb for child.
  - copy the pcb of parent into child.
  - assign unique pid to child pcb.
  - allocate memory for child process.
  - copy the parent process into child process memory.
  - return 0 to child and child pid to parent.
- exec() syscall
  - If new program section sizes are not matching with calling process sections, then it release sections of the calling process and reallocate them as per need of new program.
  - Read sections from given executable file and copy them into sections of the calling process.
  - Reset heap and stack sections of the calling process to begin execution of new program.
  - Place command line args and env variables in the stack of the calling process.
  - In PCB modify the execution context so that pc is set to startup (entry point) of the given program.

Assignments

1. Create a multi-file project (main.c, circle.c/.h, square.c/.h, rectangle.c/.h). Compile the program using "gcc" and execute it. No fork(), exec() expected here.
   - Compilation commands:
     - gcc -c circle.c
     - gcc -c square.c
     - gcc -c rectangle.c
     - gcc -c main.c
   - Linking command:
     - gcc -o program.out circle.o square.o rectangle.o main.o
   - Execute command:
     - ./program.out
2. Write a program that compiles above multi-file project. It runs commands "gcc -c circle.c", "gcc -c square.c", "gcc -c rectangle.c", "gcc -c main.c" concurrently.

```
parent
    |- child1 (gcc -c circle.c)
    |- child2 (gcc -c square.c)
    |- child3 (gcc -c rectangle.c)
    |- child4 (gcc -c main.c)
    wait for all child and check exit status. If all exit status 0
(success), then link
    |- child5 (gcc -o program.out circle.o square.o rectangle.o
main.o)
    wait for child and check exit status. If all exit status 0
(success), then run it.
    |- child6 (./program.out)
    wait for child and check exit status. Then print child exit
status.
```