LECTURE 5: PROCESS API (SYSTEM CALL)

PROCESS CREATION

Chapters 5 OSTEP: https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

AGENDA

- HOW TO CREATE AND CONTROL PROCESSES?

- What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable powerful functionality, ease of use, and high performance

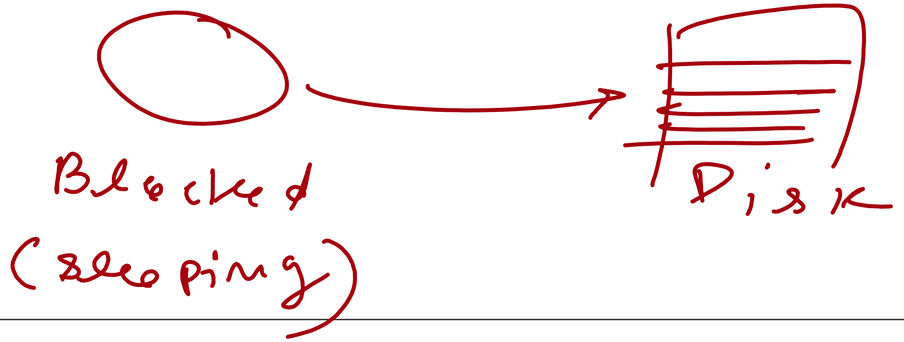
Operating systems APIs

- This group of APIs defines how applications use the resources and services of operating systems.
- Every OS has its set of APIs, for instance, Windows API or Linux API (kernel user-space API and kernel internal API).

API

- API provided by OS is a set of "system calls"
  - System call is a function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level

    only at higher privilege level.
  - Some "blocking" system calls cause the process to be blocked and descheduled (e.g., read from disk)



POSIX API

- We want to avoid writing programs for each OS
- POSIX API: a standard set of system calls that an OS must implement
  - Programs written to the POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Ensures program portability
- Program language libraries hide the details of invoking system calls
  - The printf function in the C library calls the write system call to write to screen
  - User programs usually do not need to worry about invoking system calls

System calls for process creations and manipulations

init → ancestor

- Unix based
- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - The init process is ancestor of all processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

fork()

- System call **fork()** is used to create processes.
- It takes no arguments and returns a process ID.
- The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller.
- After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.
- Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:
  - If **fork()** returns a negative value, the creation of a child process was unsuccessful.
  - **fork()** returns a zero to the newly created child process.
  - **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.
- Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.**

fork() system call return a process ID.

int k = fork();

k<0 child creat failed

k==0 child created

type => pid_t

getpid() to get the process id.

Parent Address space ⟷ child has separate address space.

Fork Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("parent of %d (pid:%d)\n",
            rc, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p1
hello (pid:29146)
parent of 29147 (pid:29146)
child (pid:29147)
prompt>
```

wait() System Call

- `wait()` System call is used for a parent to wait for a child process to finish what it has been doing.
- Process termination scenarios
  - By calling `exit()` (exit is called automatically when end of main is reached)
  - OS terminates a misbehaving process

terminated process exists as a ZOMBIE

- Terminated process exists as a zombie
- When a parent calls `wait()`, zombie child is cleaned up or "reaped"
- `wait()` blocks in parent until child terminates (non-blocking ways to invoke wait exist)
- What if parent terminates before child?
  - `init` process adopts orphans and reaps them

fork() and wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {           // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
  } else {                // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Parent fork child

exec() System Call

same same

- The exec() system call is useful when you want to run a program that is different from the calling program.
  - Calling fork() only makes the child executing the same code, not useful.

Parent exec() different

- A process can run exec() to load another executable to its memory image
  - So, a child can run a different program from parent
  - The exec() family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.

Variants of exec():

- Functions in the exec() family have different behaviours:
  - *l* : arguments are passed as a list of strings to the main()
  - *v* : arguments are passed as an array of strings to the main()
  - *p* : path/s to search for the new running program
  - *e* : the environment can be specified by the caller

environment can be specified by the caller.

Mixing them

- You can mix them, therefore you have:
  - int execl(const char *path, const char *arg, ...);
  - int execlp(const char *file, const char *arg, ...); It will search automaticaly.
  - int execle(const char *path, const char *arg, ..., char * const envp[]);
  - int execv(const char *path, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - int execvpe(const char *file, char *const argv[], char *const envp[]);
- For all of them the initial argument is the name of a file that is to be executed.

fork(), wait(), and exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc"
    myargs[1] = strdup("p3.c"); // arg: input file
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p3
hello (pid:29383)
child (pid:29384)
      29     107    1030 p3.c
parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

Case study: How does a shell work?

first process is the

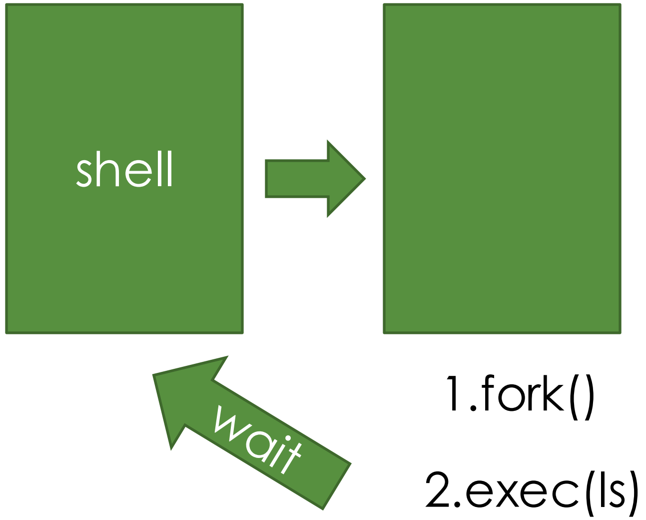
$\boxed{\text{init process}}$

forks

$\boxed{\text{Shell}}$ → this is the shell what we see, when xv6 boot on qemu.

- In a basic OS, the *init* process is created after initialization of hardware
- The *init* process spawns a shell like *bash*
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like *ls* are all executables that are simply exec'ed by the shell

```
prompt>ls
a.txt      b.txt      c.txt
```



More funky things about the shell

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file
- `prompt>ls > foo.txt`
- Shell spawns a child, rewires its standard output to a file, then calls exec on the child

ls is a child.

rewire it's stdout

file

foo.txt.

All Of The Above With Redirection

```
prompt> ./p4
prompt> cat p4.output
      32      109      846 p4.c
prompt>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
         S_IRWXU);
    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: wc
    myargs[1] = strdup("p4.c");  // arg: file to count
    myargs[2] = NULL;            // mark end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

redirecting

executing

Process Control and Users

- Beyond fork(), exec(), and wait(), there are a lot of other interfaces for interacting with processes in UNIX systems
- kill() system call : used to send signals to a process
- Dont think that kill() is to terminate a process only. It can send all kinds of signals.

Kill() used to Send Signal

Thank You