System Call Examples

Yajin Zhou (http://yajin.org)

Zhejiang University

Pstree

```
work@yajin:~$ pstree -p
systemd(1)─┬─PM2 v5.2.2: God(195505)─┬─python(228993)───{python}(229000)
           │                        ├─{PM2 v5.2.2: God}(195506)
           │                        ├─{PM2 v5.2.2: God}(195507)
           │                        ├─{PM2 v5.2.2: God}(195508)
           │                        ├─{PM2 v5.2.2: God}(195509)
           │                        ├─{PM2 v5.2.2: God}(195510)
           │                        ├─{PM2 v5.2.2: God}(195511)
           │                        ├─{PM2 v5.2.2: God}(195512)
           │                        ├─{PM2 v5.2.2: God}(195513)
           │                        ├─{PM2 v5.2.2: God}(195514)
           │                        └─{PM2 v5.2.2: God}(195515)
           ├─accounts-daemon(90)─┬─{accounts-daemon}(102)
           │                     └─{accounts-daemon}(113)
```

fork

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

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

```

fork

- A way to create a new process
- Odd part
  - the newly created process is an exact copy of the calling process
  - **return twice**
  - new process has its own memory address space and etc

fork + wait

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

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

```

fork + wait

- Parent process can use the **wait** system call to wait the child process finishes executing

fork + wait + exec


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

int
main(int argc, char *argv[])
{
    printf("hell world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");
        myargs[1] = strdup("p3.c");
        myargs[2] = NULL;
        execvp(myargs[0], myargs);
        printf("this shouldn't print out");
    } else {
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
    }
    return 0;
}

```

fork + wait + exec

- Exec is useful when you want to run a program that is different from the calling program
- Exec never **returns**
- Why separating fork and exec?
  - Essential building UNIX shell
- Shell: a user program
  - Wait for inputs
  - execute commands: fork, exec, and wait

```
prompt> wc p3.c > newfile.txt
```

  - Separating fork and exec can make the shell something interesting and useful — make something happen after fork but before exec

fork + wait + exec

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

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

        char *myargs[3];
        myargs[0] = strdup("wc");
        myargs[1] = strdup("p4.c");
        myargs[2] = NULL;
        execvp(myargs[0], myargs);
        printf("this shouldn't print out");
    } else {
        int wc = wait(NULL);
    }
    return 0;
}
```

File descriptor

Integer value	Name	<stdio.h> file stream
0	Standard input	**stdin**
1	Standard output	**stdout**
2	Standard error	stderr

fork + wait + exec

- Read the man page

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an execve(2) (i.e., the **FD_CLOEXEC** file descriptor flag described in fcntl(2) is initially disabled); the **O_CLOEXEC** flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see lseek(2)).

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

ptrace

NAME top

ptrace - process trace

SYNOPSIS top

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

ptrace

A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

```
ptrace(PTRACE_foo, pid, ...)
```

where *pid* is the thread ID of the corresponding Linux thread.

(Note that in this page, a "multithreaded process" means a thread group consisting of threads created using the clone(2) **CLONE_THREAD** flag.)

A process can initiate a trace by calling fork(2) and having the resulting child do a **PTRACE_TRACEME**, followed (typically) by an execve(2). Alternatively, one process may commence tracing another process using **PTRACE_ATTACH** or **PTRACE_SEIZE**.

While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is **SIGKILL**, which has its usual effect.) The tracer will be notified at its next call to waitpid(2) (or one of the related "wait" system calls); that call will return a *status* value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

ptrace

```c
int main(int argc, char* argv[]) {
  pid_t child;

  if (argc == 1) {
    exit(0);
  }

  char* chargs[argc];
  int i = 0;

  while (i < argc - 1) {
    chargs[i] = argv[i+1];
    i++;
  }
  chargs[i] = NULL;

  child = fork();
  if(child == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execvp(chargs[0], chargs);
  } else {
    int status;

    while(waitpid(child, &status, 0) && ! WIFEXITED(status)) {
      struct user_regs_struct regs;
      ptrace(PTRACE_GETREGS, child, NULL, &regs);
      fprintf(stderr, "system call %d from pid %d\n", (REG(regs)), child);
      ptrace(PTRACE_SYSCALL, child, NULL, NULL);
    }
  }
  return 0;
}
```

Debugger

A debugger or debugging tool is a computer program that is used to test and debug other programs (the “target” program)

https://en.wikipedia.org/wiki/Debugger

Debugger

I don't like debuggers. Never have, probably never will.

— Linus Torvalds <torvalds@transmeta.com> (2000)

Ptrace (again)

- `long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);`

  The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

  — man 2 ptrace

- Used by gdb, strace, DynInst...

Attaching to a process

- `(gdb) start` PTRACE_TRACEME – makes parent a tracer (called by a tracee)
- `(gdb) attach PID` PTRACE_ATTACH – attach to a running process
- Watch out for Yama security module:

```
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
ptrace: Operation not permitted.
```

- `prctl(PR_SET_PTRACER, pid, ...)`

Basic Control

- `(gdb) stop` `kill(child_pid, SIGSTOP)` (or PTRACE_INTERRUPT)
- `(gdb) continue` PTRACE_CONT
- `(gdb) info registers` PTRACE_GET(FP)REGS(ET) and PTRACE_SET(FP)REGS(ET)
- `(gdb) x` PTRACE_PEEKTEXT and PTRACE_POKETEXT

Setting a breakpoint

- `(gdb) br *ADDRESS`
- Instruction at the given address is read, saved and replaced with a breakpoint:
  - either a special instruction,
  - or an undefined encoding.

Hitting a breakpoint

- Executing breakpoint instruction causes:
  - SIGTRAP when using special instruction,
  - SIGILL for undefined instructions.
- Any signal destined for the tracee stops its execution.
  - Tracer is notified about it via `waitpid(PID)` result.
  - Breakpoint-related signals are suppressed (otherwise would be delivered to the tracee after continuing).
  - To continue, the original instruction is temporarily restored and single stepped.