`clone("2")`

CompSys 2016

Oleks

DIKU

November 9, 2016

Disclaimer

This presentation shamelessly cites
release 4.07 (2016-07-17) of the Linux man-pages project.

The original presentation was given on a
Linux 4.4.28, bundled with GNU libc 2.24.

fork(“2”)

```
int main() {
  printf("Parent’s perceived PID: %d\n", getpid());

  pid_t retval = fork();
  if (retval < 0) {
    error(1, errno, "system b0rked ;-/");
  }

  if (retval == 0) {
    printf("Child’s perceived PID: %d\n", getpid());
  } else {
    printf("Child’s real PID: %d\n", retval);
  }
}
```

`make fork-trace` (i.e., `strace -f ./fork.bin`)

```
10094 write(1, "Parent’s perceived PID: 10094\n", 30) = 30
10094 clone(child_stack=NULL,
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x7fbdb7d839d0) = 10095
10094 write(1, "Child’s real PID: 10095\n", 24) = 24
10095 write(1, "Child’s perceived PID: 10095\n", 29 <unfinished
    ...>
```

`man "2" fork`

C library/kernel differences

Since version 2.3.3, rather than invoking the kernel's `fork()` system call, the glibc `fork()` wrapper that is provided as part of the NPTL threading implementation invokes `clone(2)` with flags that provide the same effect as the traditional system call. (. . .)

- NPTL stands for Native POSIX Thread Library.
- POSIX stands for Portable Operating System Interface.

man “2” clone

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
```

`clone()` creates a new process, in a manner similar to `fork(2)`.

(...)

Unlike `fork(2)`, `clone()` allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.
(...)

When the child process is created with `clone()`,
it executes the function `fn(arg)`.
(...)

When the `fn(arg)` function application returns, the child process terminates. The integer returned by `fn` is the exit code for the child process.

Let’s do some work!

```
int work(void *arg) {
  arg = arg;
  printf("Child’s perceived PID: %d\n", getpid());
  return 0;
}

int main() {
  work(NULL);
}
```

gcc -fstack-usage?

```
%.su: %.c Makefile
  $(CC) $(CFLAGS) -fstack-usage $< -o $*.bin
```

- Results for our `work.c` above:

  ```
  work.c:7:5:work 16 static
  work.c:13:5:main 16 static
  ```

- Size is given in bytes...
- May be `static` or `dynamic` (but hopefully, `bounded`).

Default Stack Sizes

This metric doesn't take into account:

- Thread-local storage.
- C Standard Library overhead.

So, we typically, we use some heuristic (sigh):

- 8MB in GNU libc
- 80KB in musl libc
- 65KB in my sample code

```
#define STACK_SIZE (0x10000) // 65KB.
```

Degree of Multiprogramming

How many threads can we run simultaneously?

- Kernel thread handling overhead limits this.
- Exuberant stack sizes (e.g., GNU libc) limits this.
- This is why more efficiency-concerned libraries (e.g., musl libc), might prefer a smaller *default* stack size.

clone(“2”)

```
int work(void *arg) {
  arg = arg;
  printf("Child’s perceived PID: %d\n", getpid());
  return 0;
}

char stack[STACK_SIZE];

int main() {
  printf("Parent’s perceived PID: %d\n", getpid());

  long retval = clone(work, stack + STACK_SIZE, 0, NULL);
  if (retval < 0) {
    error(1, errno, "system b0rked ;-/");
  }

  printf("Child’s real PID: %ld\n", retval);
}
```

man “2” clone

C library/kernel differences

The raw `clone()` system call corresponds more closely to `fork(2)` in that execution in the child continues from the point of the call. As such, the `fn` and `arg` arguments of the `clone()` wrapper function are omitted. Furthermore, the argument order changes. The raw system call interface on x86 and many other architectures is roughly:

```
long clone(unsigned long flags, void *child_stack,
           void *ptid, void *ctid,
           struct pt_regs *regs);
```

Another difference for the raw system call is that the `child_stack` argument may be zero, in which case copy-on-write semantics ensure that the child gets separate copies of stack pages when either process modifies the stack. In this case, for correct operation, the `CLONE_VM` option should not be specified.

For some architectures, the order of the arguments for the system call differs from that shown above.

```
man “2” clone
```

Control groups

Resource Limits, The Interface

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

struct rlimit {
  rlim_t rlim_cur;  /* Soft limit */
  rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

- Per-process resource limits.
- Inheritted by subprocesses.
- A process can only *irreversibly* lower the “hard limit”.
- (But still arbitrarily manipulate the “soft limit”.)

Resource Limits, Examples

`RLIMIT_STACK`	Maximum stack size stack[1].
`RLIMIT_NPROC`	Maximum number of subprocesses.
`RLIMIT_FSIZE`	Maximum file size.
`RLIMIT_NOFILE`	Maxmimum number of open files.
`RLIMIT_CPU`	CPU time limit in seconds.
`RLIMIT_AS`	Address space (VM) size limit.
`RLIMIT_DATA`	Maximum data segment size.

[1] Used by NTPL `clone()`.

“Containers”

- There is no unified notion of a “container” in the Linux kernel.
- Instead, there is a mixture of kernel-level virtualisation mechanisms, enabling the virtualisation of various system resources.
- Intricately combined, these give rise to a userland notion of “containers”.

This is the sort of stuff that, for instance, Docker builds on top of.

- Enough C already? Want to do this in pure x86_64?
  — See `http://nullprogram.com/blog/2015/05/15/`
- You might need some tools.
  — `https://github.com/oleks/compsys16-assembly`