浙江大學
ZHEJIANG UNIVERSITY

Light-Weight Processes: Dissecting Linux Threads

Yajin Zhou (http://yajin.org)

Zhejiang University

source: https://opensourceforu.com/2011/08/light-weight-processes-dissecting-linux-threads/

Thread

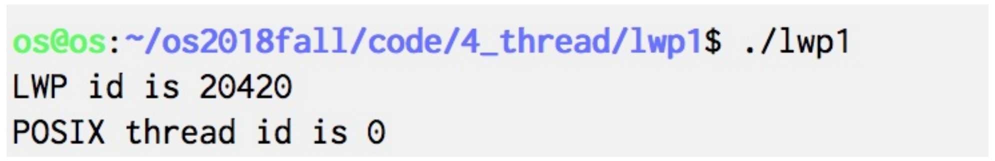
- Threads have their own identity (thread ID), and can function independently.
- They share the address space within the process, and reap the benefits of avoiding any IPC (Inter-Process Communication) channel (shared memory, pipes and so on) to communicate.
- Threads of a process can directly communicate with each other
    - for example, independent threads can access/update a global variable.
- This model eliminates the potential IPC overhead that the kernel would have had to incur. As threads are in the same address space, a thread context switch is inexpensive and fast.

How does Linux implement threads?

- User-level threads in Linux follow the open POSIX (Portable Operating System Interface for uniX) standard, designated as IEEE 1003. The user-level library (on Ubuntu, glibc.so) has an implementation of the POSIX API for threads.
- Threads exist in two separate execution spaces in Linux — in **user space** and the **kernel.**
    - User-space threads are created with the pthread library API (POSIX compliant).
    - In Linux, kernel threads are regarded as **"light-weight processes"**. An LWP is the unit of a basic execution context. Unlike other UNIX variants, including HP-UX and SunOS, there is no special treatment for threads. **A process or a thread in Linux is treated as a "task"**, and shares the same structure representation (list of struct task_structs).
    - These user-space threads are mapped to kernel threads.

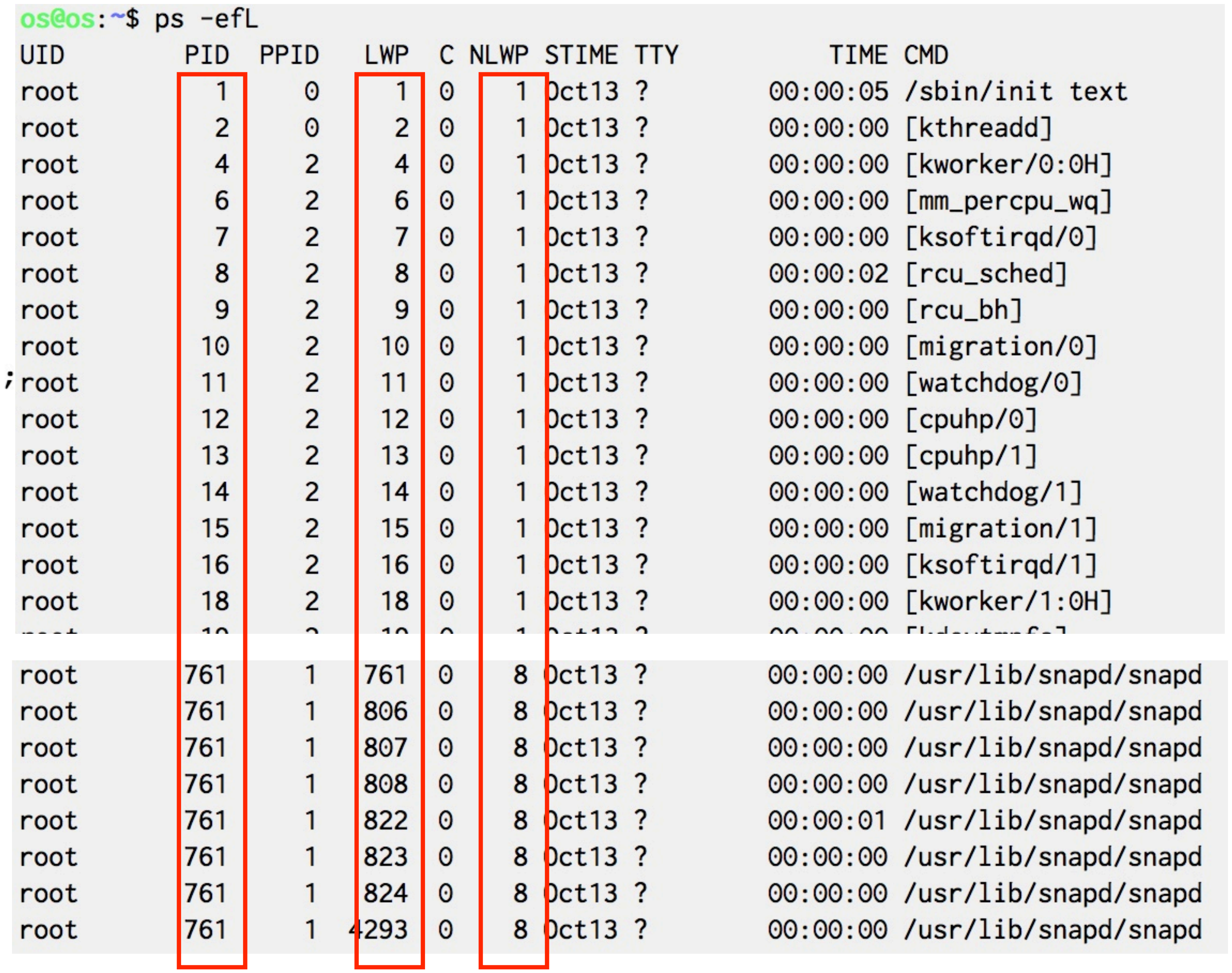
How does Linux implement threads?

- For a set of user threads created in a user process, there is a set of corresponding LWPs in the kernel



```
os@os:~/os2018fall/code/4_thread/lwp1$ ./lwp1
LWP id is 20420
POSIX thread id is 0
```

```c
#include <stdio.h>
#include <syscall.h>
#include <pthread.h>

int main()
{
    pthread_t tid = pthread_self();
    int sid = syscall(SYS_gettid);
    printf("LWP id is %dn", sid);
    printf("POSIX thread id is %dn", tid);
    return 0;
}
```



```
os@os:~$ ps -efL
UID          PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root           1     0     1  0    1 Oct13 ?        00:00:05 /sbin/init text
root           2     0     2  0    1 Oct13 ?        00:00:00 [kthreadd]
root           4     2     4  0    1 Oct13 ?        00:00:00 [kworker/0:0H]
root           6     2     6  0    1 Oct13 ?        00:00:00 [mm_percpu_wq]
root           7     2     7  0    1 Oct13 ?        00:00:00 [ksoftirqd/0]
root           8     2     8  0    1 Oct13 ?        00:00:02 [rcu_sched]
root           9     2     9  0    1 Oct13 ?        00:00:00 [rcu_bh]
root          10     2    10  0    1 Oct13 ?        00:00:00 [migration/0]
root          11     2    11  0    1 Oct13 ?        00:00:00 [watchdog/0]
root          12     2    12  0    1 Oct13 ?        00:00:00 [cpuhp/0]
root          13     2    13  0    1 Oct13 ?        00:00:00 [cpuhp/1]
root          14     2    14  0    1 Oct13 ?        00:00:00 [watchdog/1]
root          15     2    15  0    1 Oct13 ?        00:00:00 [migration/1]
root          16     2    16  0    1 Oct13 ?        00:00:00 [ksoftirqd/1]
root          18     2    18  0    1 Oct13 ?        00:00:00 [kworker/1:0H]

root         761     1   761  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1   806  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1   807  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1   808  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1   822  0    8 Oct13 ?        00:00:01 /usr/lib/snapd/snapd
root         761     1   823  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1   824  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761     1  4293  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
```

What is a Light-Weight Process?

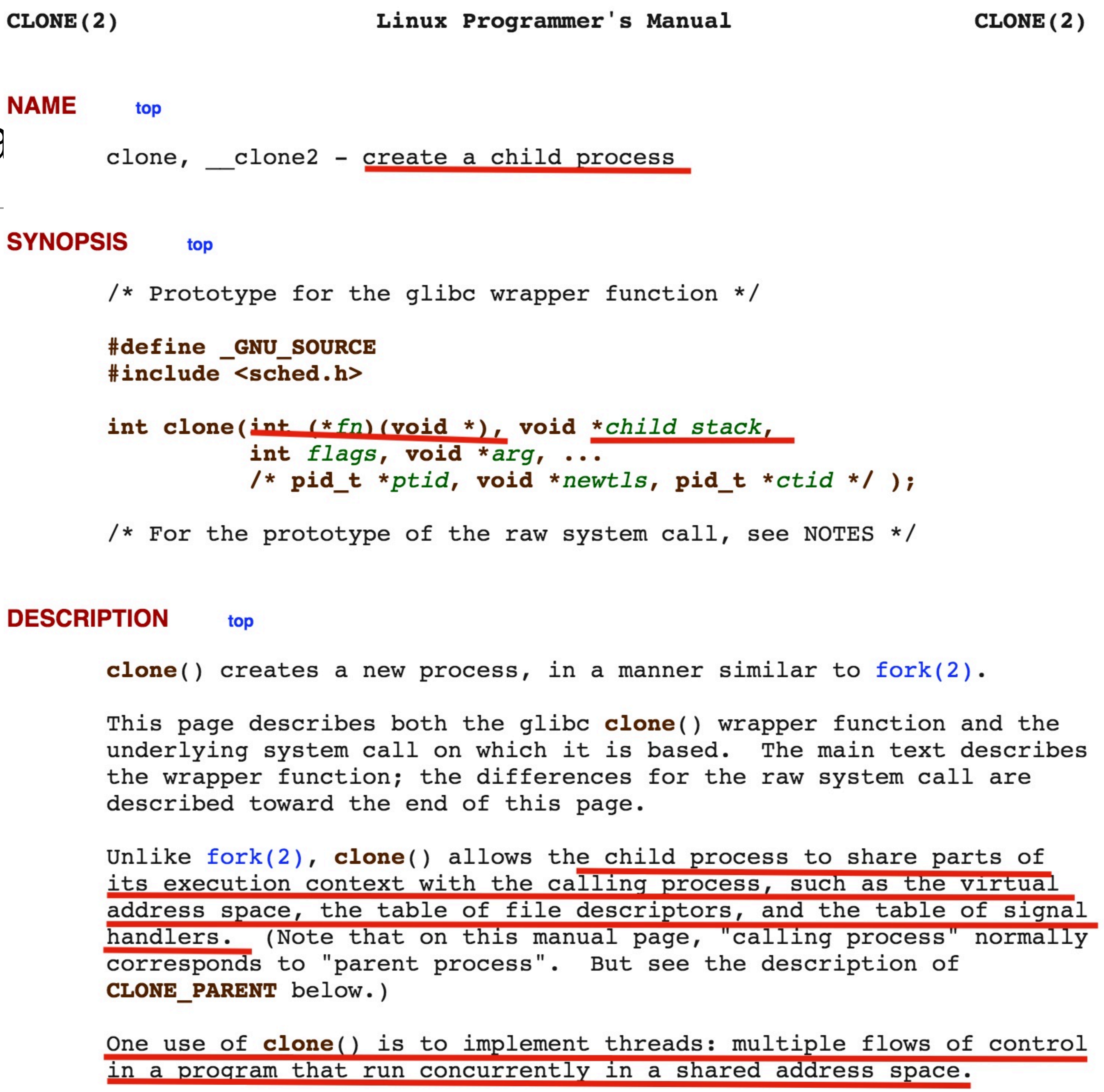
- An LWP is a process created to facilitate a user-space thread. **Each user-thread has a 1×1 mapping to an LWP**.
- The creation of LWPs is different from an ordinary process; for a user process “P”, **its set of LWPs share the same group ID**. Grouping them allows the kernel to enable resource sharing among them (resources include the address space, physical memory pages (VM), signal handlers and files). This further enables the kernel to avoid context switches among these processes. Extensive resource sharing is the reason these processes are called **light-weight processes**.

How does Linux create LWPs

- Linux handles LWPs via the non-standard **clone() system call**. It is similar to fork(), but more generic. Actually, fork() itself is a manifestation of clone(), which allows programmers to choose the resources to share between processes.

- The clone() call creates a process, but the **child process shares its execution context with the parent**, including the memory, file descriptors and signal handlers. The pthread library too uses clone() to implement threads. Refer to ./nptl/sysdeps/pthread/createthread.c in the glibc version 2.11.2 sources.

Crea



CLONE(2) Linux Programmer's Manual CLONE(2)

NAME top

clone, __clone2 - create a child process

SYNOPSIS top

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );

/* For the prototype of the raw system call, see NOTES */
```

DESCRIPTION top

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

This page describes both the glibc **clone**() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike fork(2), **clone**() allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of **CLONE_PARENT** below.)

One use of **clone**() is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.

Create your own LWP

```
// 64kB stack
#define STACK 1024*64

// The child thread will execute this function
int threadFunction( void* argument ) {
     printf( "child thread entering\n" );
     close((int*)argument);
     printf( "child thread exiting\n" );
     return 0;
}

int main() {
     void* stack;
     pid_t pid;
     int fd;

     fd = open("/dev/null", O_RDWR);
     if (fd < 0) {
         perror("/dev/null");
         exit(1);
     }

     // Allocate the stack
     stack = malloc(STACK);
     if (stack == 0) {
         perror("malloc: could not allocate stack");
         exit(1);
     }
     printf("Creating child thread\n");
```

Create your own LWP

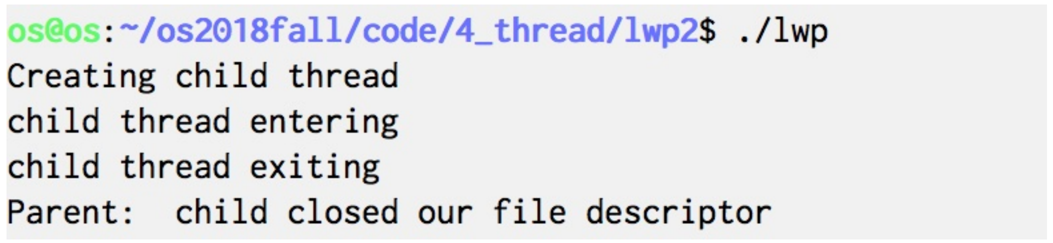
```c
    // Call the clone system call to create the child thread
    pid = clone(&threadFunction,
                (char*) stack + STACK,
                SIGCHLD | CLONE_FS | CLONE_FILES |\
                 CLONE_SIGHAND | CLONE_VM,
                (void*)fd);

    if (pid == -1) {
         perror("clone");
         exit(2);
    }

    // Wait for the child thread to exit
    pid = waitpid(pid, 0, 0);
    if (pid == -1) {
        perror("waitpid");
        exit(3);
    }

    // Attempt to write to file should fail, since our thread has
    // closed the file.
    if (write(fd, "c", 1) < 0) {
        printf("Parent:\t child closed our file descriptor\n");
    }

    // Free the stack
    free(stack);

    return 0;
}
```

Create your own LWP

- `SIGCHLD` : The thread sends a `SIGCHLD` signal to the parent process after completion. It allows the parent to `wait()` for all its threads to complete.
- `CLONE_FS` : Shares the parent's filesystem information with its thread. This includes the root of the filesystem, the current working directory, and the umask.
- `CLONE_FILES` : The calling and caller process share the same file descriptor table. Any change in the table is reflected in the parent process and all its threads.
- `CLONE_SIGHAND` : Parent and threads share the same signal handler table. Again, if the parent or any thread modifies a signal action, it is reflected to both the parties.
- `CLONE_VM` : The parent and threads run in the same memory space. Any memory writes/mapping performed by any of them is visible to other process.

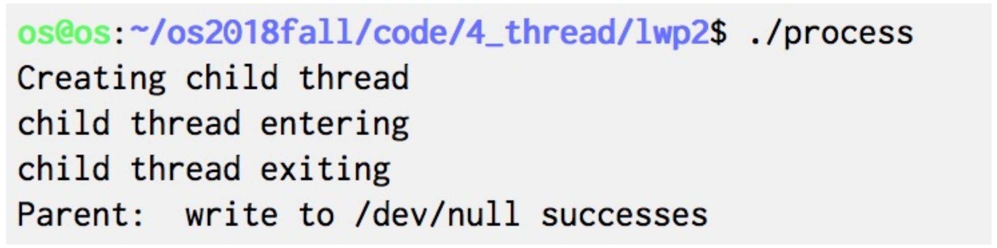
A Slight Change to the Code

```
// Call the clone system call
    pid = clone(&threadFunction,
                (char*) stack + STACK,
                SIGCHLD | CLONE_FS | CLONE_FILES |\
                 CLONE_SIGHAND | CLONE_VM,
                (void*)fd);
```



```
os@os:~/os2018fall/code/4_thread/lwp2$ ./lwp
Creating child thread
child thread entering
child thread exiting
Parent:  child closed our file descriptor
```

```
// Call the clone system call
    pid = clone(&threadFunction,
                (char*) stack + STACK,
                SIGCHLD | CLONE_FS |\
                 CLONE_SIGHAND | CLONE_VM,
                (void*)fd);
```



```
os@os:~/os2018fall/code/4_thread/lwp2$ ./process
Creating child thread
child thread entering
child thread exiting
Parent:  write to /dev/null successes
```

Another Example: CLONE_VM

```c
static int child_func(void* arg) {
  char* buf = (char*)arg;
  printf("Child sees buf = \"%s\"\n", buf);
  strcpy(buf, "hello from child");
  return 0;
}

int main(int argc, char** argv) {
  // Allocate stack for child task.
  const int STACK_SIZE = 65536;
  char* stack = malloc(STACK_SIZE);
  if (!stack) {
    perror("malloc");
    exit(1);
  }

  // When called with the command-line argument "vm", set the
CLONE_VM flag on.
  unsigned long flags = 0;
  if (argc > 1 && !strcmp(argv[1], "vm")) {
    flags |= CLONE_VM;
  }

  char buf[100];
  strcpy(buf, "hello from parent");
  if (clone(child_func, stack + STACK_SIZE,
flags | SIGCHLD, buf) == -1) {
    perror("clone");
    exit(1);
  }

  int status;
  if (wait(&status) == -1) {
    perror("wait");
    exit(1);
  }

  printf("Child exited with status %d. buf =
\"%s\"\n", status, buf);
  return 0;
}
```

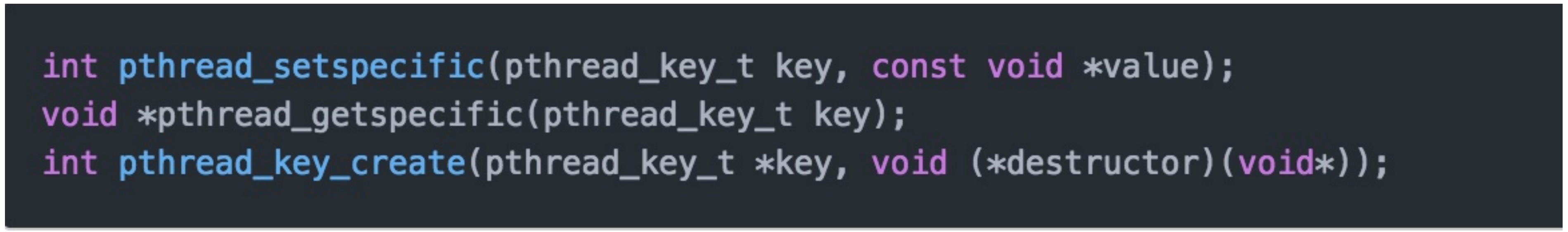
Why COW (Copy on Write) Is Not Enough

- For processes, there's a bit of copying to be done when **fork** is invoked, which costs time. The biggest chunk of time probably goes to copying the memory image due to the lack of **CLONE_VM**. **Note, however, that it's not just copying the whole memory**; Linux has **an important optimization by using COW** (Copy On Write) pages. The child's memory pages **are initially mapped to the same pages shared by the parent**, **and only when we modify them the copy happens**. This is very important because processes will often use a lot of shared read-only memory (think of the global structures used by the standard library, for example).

- But still, **the page tables still have to be copied.** This overhead is not applied to thread, since **threads inside a process are sharing address space** — using same page tables and mappings

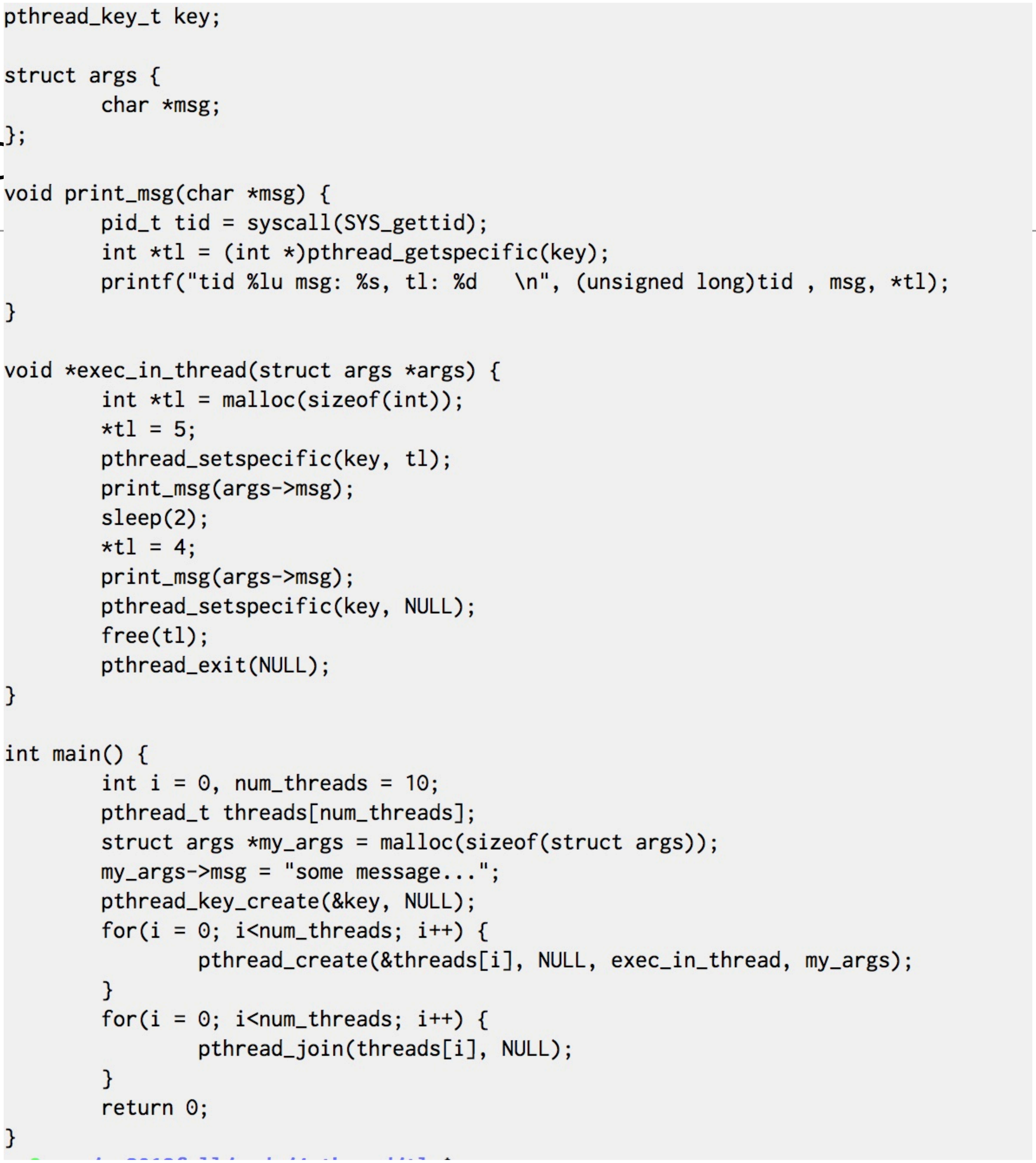
source: https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/

Pthread: TLS



```c
int pthread_setspecific(pthread_key_t key, const void *value);
void *pthread_getspecific(pthread_key_t key);
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

Pthreac



```c
pthread_key_t key;

struct args {
        char *msg;
};

void print_msg(char *msg) {
        pid_t tid = syscall(SYS_gettid);
        int *tl = (int *)pthread_getspecific(key);
        printf("tid %lu msg: %s, tl: %d   \n", (unsigned long)tid , msg, *tl);
}

void *exec_in_thread(struct args *args) {
        int *tl = malloc(sizeof(int));
        *tl = 5;
        pthread_setspecific(key, tl);
        print_msg(args->msg);
        sleep(2);
        *tl = 4;
        print_msg(args->msg);
        pthread_setspecific(key, NULL);
        free(tl);
        pthread_exit(NULL);
}

int main() {
        int i = 0, num_threads = 10;
        pthread_t threads[num_threads];
        struct args *my_args = malloc(sizeof(struct args));
        my_args->msg = "some message...";
        pthread_key_create(&key, NULL);
        for(i = 0; i<num_threads; i++) {
                pthread_create(&threads[i], NULL, exec_in_thread, my_args);
        }
        for(i = 0; i<num_threads; i++) {
                pthread_join(threads[i], NULL);
        }
        return 0;
}
```

Pthread: TLS

```
__thread int x = 3;

void print_msg() {
        pid_t tid = syscall(SYS_gettid);
        printf("tid %lu x: %d   \n", (unsigned long)tid , x);
}

void *exec_in_thread(struct args *args) {

        x += 1;
        print_msg();

        sleep(3);
        pthread_exit(NULL);
}

int main() {
        int i = 0, num_threads = 10;
        pthread_t threads[num_threads];
        for(i = 0; i<num_threads; i++) {
                pthread_create(&threads[i], NULL, exec_in_thread, NULL);
        }
        for(i = 0; i<num_threads; i++) {
                pthread_join(threads[i], NULL);
        }
        return 0;
}
```

```
int x = 3;

void print_msg() {
        pid_t tid = syscall(SYS_gettid);
        printf("tid %lu x: %d   \n", (unsigned long)tid , x);
}

void *exec_in_thread(struct args *args) {

        x += 1;
        print_msg();

        sleep(3);
        pthread_exit(NULL);
}

int main() {
        int i = 0, num_threads = 10;
        pthread_t threads[num_threads];
        for(i = 0; i<num_threads; i++) {
                pthread_create(&threads[i], NULL, exec_in_thread, NULL);
        }
        for(i = 0; i<num_threads; i++) {
                pthread_join(threads[i], NULL);
        }
        return 0;
}
```