Operating Systems

CSE 231

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 3: Sept. 14 – Sept. 18

More on Process Fork()

- Fork() is used to create a new child process from the parent process.
- Parent process gets PID == PID of child process.
- Child process gets PID == 0.
- Technically child and parent process have IDENTICAL copies of memory pages.

Copy-on-Write (COW)

- In Linux fork() is implemented through copy-on-write functionality.
- Often child process execs another binary, thus wasting all memory that has been copied between the child and parent processes.
- Solution: Copy-on-write(): Create everything for the child process – task_struct, page table entries but NOT the actual memory pages from the parent process…NOT until the child process actually writes into data blocks.
- Linux implements fork() via the clone() system call (so does vfork()).

The steps to fork() a child process

1. Bulk of the work done by clone() → do_fork(), which is defined in kernel/fork.c.
2. do_fork() calls copy_process().
3. copy_process() → dup_task_struct() which allocates new stack for the process and the task_struct.
4. Checks if the spawning of the new processes does not exceed the resource limit of the process.
5. Child processes' state set to TASK_UNINTERRUPTIBLE
6. Various book-keeping activities to set appropriate privileges etc. to the child process.
7. Allocate PID to child process
8. Depending upon the flags passed to clone, the copy_process() either duplicates or shares filesystem information, sig handlers, VMA,etc. Thus creating child process or threads (as required).

Finally control passes back to do_fork() which wakes up the child process and runs first – child-runs-first.

Why ?

Hint: Copy-on-write, child-runs-first.

The vfork()

-Pretty much same as fork(), just that it does not even copy the page table entries, which fork() does.

-Initially introduced in 3BSD.

-Does COW.

-With Linux doing COW for fork(), vfork() is rather redundant.

-Problems with vfork() – complicated semantics – what if exec() fails ?

-No extra benefit by having ``copy-on-write'' page-table entries.

The clone()

Everything is implemented via clone() (fork(), vfork(), pthread_create()) just by varying the flags passed to clone().

- Linux does not have a concept of thread as such – its merely a ``lightweight process'' which shares resources with the parent process.
- Threads have a corresponding task_struct which the kernel keeps track of.
- Other OSes, e.g. Windows, have an explicit thread that the kernel knows of which is used to associated to every process. The threads are scheduled separately from the processes.

How threads are implemented using clone()

Threads are implemented through the clone() system call by passing appropriate flags:

clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

I

n contrast, a normal fork() can be implemented as:

clone(SIGCHLD, 0);

vfork() is implemented as:

clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);

Some clone() flags

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_PARENT	
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.

Kernel Thread

Some threads which are created (often by the kernel) so to function in the background.

Have a task_struct, very much like ordinary processes and are scheduled like an ordinary process but lack the VMA of a normal process.

Very much like the `init' process, the kernel also spawns an kthreadd process which can spawn other kernel threads. One kernel thread spawns other kernel threads.

Kernel threads created using kthread_create() function but does not run until set to running state through the wake_up_process() function.

Process Termination

-Typically a process terminates via the exit() system call.

-Implicitly the C compiler places a call to exit() at the end of the main() function.

How process termination works:

1. exit() □ do_exit() which handles the most part of the work.
2. do_exit() removes any queued up timers – viz. PIC values set/corresponding to the process being terminated.
3. OS MMU/processor MMU together free up the memory pages allocated to the process and also the VMA.
4. Various process / thread synchronization locks are freed up.
5. References to various file system objects e.g. files are deleted.
6. Notification to the parent process which signals the waitpid() function.

Zombies

In case a parent process terminates before the child process, it becomes a zombie.

-If zombies remain *unparented* they would waste way to much memory.

-The solution is to reparent a task's children on exit to either another process in the current thread group or, if that fails, the init process -- do_exit() → exit_notify() → forget_original_parent() → find_new_reaper() to perform the reparenting.

System Calls

-Almost everything you do is via some syscall. The OS does everything for your processes via syscalls.

- open(), read(), write(), close(), fork() etc.

System Call Implementation Example

```
SYSCALL_DEFINE0(getpid)
 {
   return task_tgid_vnr(current); // returns current->tgid

 }
```

SYSCALL_DEFINE0 – Macro specifying the number of arguments to be passed.

Expands to :

asmlinkage long sys_getpid(void)

The asmlinkage modifier on the function definition. This is a directive to tell the compiler to look only on the stack for this function's arguments.

System calls begin the `sys' name.

System Call Implementation Example

System call handler is architecture dependent – for x86 it is done through a soft irq 0x80 while the x86-64 is implemented via a syscall instruction.

Parameter passing

In addition to the system call number, most syscalls require that one or more parameters be passed to them.

Somehow, user-space must relay the parameters to the kernel during the trap.

The easiest way to do this is via the same means that the syscall number is passed: The parameters are stored in registers.

On x86-32, the registers ebx, ecx, edx, esi, and edi contain, in order, the first five arguments. In the unlikely case of six or more arguments, a single register is used to hold a pointer to user-space where all the parameters are stored.

The return value is sent to user-space also via register. On x86, it is written into the eax register

Copy_to_user and Copy_from_user

```
/*  silly_copy - pointless syscall that copies the len bytes from * ‘src’ to ‘dst’ using the kernel
as an intermediary in the copy. * Intended as an example of copying to and from the kernel.
*/

SYSCALL_DEFINE3(silly_copy, unsigned long *, src, unsigned long *, dst, unsigned long len)
{
unsigned long buf; /* copy src, which is in the user’s address space, into buf */
if (copy_from_user(&buf, src, len))
   return -EFAULT; /* copy buf into dst, which is in the user’s address space */

if (copy_to_user(dst, &buf, len))
    return -EFAULT; /* return amount of data copied */

return len;

}
```

Steps to implementing a system call

1. Add an entry to the end of the system call table.This needs to be done for each architecture that supports the system call (which, for most calls, is all the architectures).The position of the syscall in the table, starting at zero, is its system call number.

```
ENTRY(sys_call_table)
  .long sys_restart_syscall /* 0 */
 .long sys_exit
 .long sys_fork
 .long sys_read
.long sys_write
.long sys_open /* 5 */
```

Steps to implementing a system call

2. Add system call number to <asm/unistd.h>

/* * This file contains the system call numbers. */
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
...
#define __NR_signalfd4 327
#define __NR_eventfd2 328
#define __NR_epoll_create1 329
#define __NR_dup3 330

Steps to implementing a system call

2. Add system call number to <asm/unistd.h>

/* * This file contains the system call numbers. */
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
...
#define __NR_signalfd4 327
#define __NR_eventfd2 328
#define __NR_epoll_create1 329
#define __NR_dup3 330

Steps to implementing a system call

3. Add system call.

```
#include <asm/page.h>
 /* * sys_foo – everyone's favorite system call. * * Returns the size of the per-process kernel stack. */
asmlinkage long sys_foo(void)
 { return THREAD_SIZE; }
```

Steps to implementing a system call

4. Invoking from user space.

```
#define __NR_foo 283
__syscall0(long, foo)

int main ()
{ long stack_size;
stack_size = foo ();
printf (“The kernel stack size is %ld\n”, stack_size);
 return 0; }
```