Laboratory on processes
OS - Operating System

Simone Rossi

October 24, 2016

Date Performed:	13/20 October 2016
Partners:	Simone Rossi

1 Information on Operating Systems

1.1 History of Windows

According to the Wikipedia page of Windows NT, it is written in C, C++ and Assembly language. The spectum of programming languages used is quite wide and is needed to explore different levels of abstraction with the right language. For example, probably boot operations and high performance applications (like "critical" drivers, such as memory management) were been written in assembly, while "normal" drivers (power management, audio, video, network, ...) and system libraryies were been written in a higher level language like C. Finally, GUI may been written in C++.

1.2 Linux Kernel

1 `uname -a` prints system informations, such as the version of the operating system and the name of the machine in use. To print only the system's version the command to be run is `uname -v`, which in my case showed `121-Ubuntu SMP Wed Jan 20 10:50:42 UTC 2016`

4 `/home/Local_Data` is a local directory on the PC, since is not part of the mounted filesystem

6 I didn't know the difference between "z" compression and "tar" compression. After a fast research, seems that tar compression is not a real compression: it is just an utility to collect many files into just one archive; on the contrary, with "z" the data is compressed with the LZW algorithm.

7 Yes, all the source file are prepared to be configure to target a specific platform. After the configuration, which I left with default parameters,

the kernel has been compiled. At the end, to use the new kernel, it has to be mount on a new root partition.

2 Analyzing processes

2 Here is reported the result of the command `vmstat -s`.

```
        0 swap ins
        0 swap outs
        0 pages swapped in
        0 pages swapped out
   783503 total address trans. faults taken
      550 page ins
        0 page outs
      783 pages paged in
        0 pages paged out
   296894 total reclaims
   296894 reclaims from free list
        0 micro (hat) faults
   783503 minor (as) faults
      550 major faults
   167649 copy-on-write faults
   290103 zero fill page faults
   216060 pages examined by the clock daemon
        0 revolutions of the clock hand
        0 pages freed by the clock daemon
     1269 forks
      850 vforks
     2020 execs
 31435117 cpu context switches
 51201789 device interrupts
   976804 traps
 10213769 system calls
  3874932 total name lookups (cache hits 96%)
    11256 user   cpu
    79993 system cpu
 30751051 idle   cpu
        0 stolen cpu
     1306 forks
      850 vforks
```

All the these informations are statistics that the OS keeps track since bootup: some of them are related to memory management (like pages in and out) and others to forks and system calls. Since bootup, the system has executed 1269 `forks` and 850 `vforks`.

4 The compilation ended with two errors:

```
fork01.c: In function   m a i n   :
fork01.c:4:3: error: unknown type name   p i d _ t
```

```
fork01.c:19:1: error: expected    ;    before    }
```

These errors are solved including the library `<unistd.h>` and a semicolomn in line 18.

5 To simply get rid the warning I added a `return 0;`.

6 There is a run time error, a segmantation fault. Looking at the strings of the executable file, this is the result:

```
  d4 /usr/lib/ld.so.1
 9b8 child
 9be parent
 9c5 process %s ret  = %ld
 b7e YZ]
 bb8 []
 c01 []
 c18    &
 cbd  $  U
 d64 [^]
```

The strings we are interested in start with offset 0009B8. Reading the ELF file, I found that the correspondent section is read-only and it has not the privilege to be written

```
[ 8] .SUNW_dynsymsort  LOOS+ffffff1    08050918 000918
     000038 04   A  3   0  4
[ 9] .SUNW_reloc       REL             08050950 000950
     000028 08   A  4   0  4
[10] .rel.plt          REL             08050978 000978
     000040 08  AI  4  16  4
[11] .rodata           PROGBITS        080509b8 0009b8
     000024 00   A  0   0  1
[12] .plt              PROGBITS        080509dc 0009dc
     000090 10  AX  0   0  4
[13] .text             PROGBITS        08050a70 000a70
     0002fc 00  AX  0   0 16
```

These lines, finally, are dump of the binary file in the section `.rodata`.

```
Contents of section .rodata:
 80509b8 6368696c 64007061 72656e74 0070726f  child.
    parent.pro
 80509c8 63657373 20257320 72657420 203d2025  cess %s
    ret  = %
 80509d8 6c640a00                             ld..
```

7 To make a more robust code, I simply added a control on the return value of the fork.

```
if(ret < 0){
   printf("Error in process creation");
   return -1;
 }
```

8 These are a few lines of a long list of function and system calls:

```
CPU     ID                    FUNCTION:NAME
  0  81524             tsort_unsigned:entry
  0  81523                      tsort:entry
  0  81461                 set_environ:entry
  0  81333                  lookup_sym:entry
  0  81353                    elf_hash:entry
  0  81332                 _lookup_sym:entry
  0  81330            core_lookup_sym:entry
  0  81460                    callable:entry
  0  81354                elf_find_sym:entry
  0  81755                      strcmp:entry
  0  81755                      strcmp:entry
  0  81755                      strcmp:entry
  0  81394                      calloc:entry
```

Some of them are system calls (like `getpid`) and others are function calls (like `mutex_lock` and `mutex_unlock`). After that, I monitored the system call of my program:

```
forksys
fstatat64
ioctl
lwp_sigmask
nanosleep
rexit
schedctl
write
```

For instance, `forksys` refears to the fork function and `nanosleep` to sleep function. To run the provided script, I also wrote a little script to automate the retrive of the PID

```
./rossi_fork &\
read pid <<<  $( ps | grep rossi_fo | awk '{print $1;}')
    ;\
echo $pid;\
./dtracescript $(echo $pid) >> 2x08.txt
```

The result of this is script is the following (just only few lines).

```
__forkx
__lwp_sigmask
__nanosleep
__schedctl
```

```
__systemcall
__write
_calloc
_cleanup
_exithandle
_fflush_l_iops
_fflush_u
_findbuf
_getfp
_malloc
_ndoprnt
_postfork_parent_handler
_prefork_handler
_setbufend
_setorientation
_ti_critical
_xflsbuf
aplist_append
aplist_delete
aplist_insert
aplist_test
```

9 Using the same script and adding a couple of instruction in the source file, I can monitor the right moment in which the child is killed.

```
forkx
free
fstat64
fstatat64
ioctl
isatty
isseekable
kill
libc_fini
```

10 Using the provided script, this is the resut.

```
SIGKILL was sent to fork_palmiero by Christian Palmiero
SIGKILL was sent to rossi_fork by Simone Rossi
SIGKILL was sent to fork_palmiero by Christian Palmiero
```

11 The program I wrote is the following.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

#define MAX 3
#define SPL 10
int main(void) {
```

```
  int i;
  int j;

  for (i = 0; i < MAX; i++)
  {
    pid_t ret = fork();
    if (ret == 0)
    {
      if ( i == 0 || i == MAX - 1 )
      {
        for (j = 0; j < 2; j++)
        {
          pid_t ret2 = fork();
          if (ret2 == 0)
          {
            printf("%d", j);
            sleep(SPL);
            exit(0);
          }
        }
        printf("%d/n", i);
        sleep(SPL);
        exit(0);
      }
    }
  }
  sleep(200);
  printf("Parent");
  return(0);
}
```

Checking with `ps` and `dtrace` I verified that It correctly spawns three children.

```
 CPU     ID    FUNCTION:NAME
   0  80418       fork:entry
   3  80418       fork:entry
   1  80418       fork:entry
```

I also traced two function calls (`printf` and `exit`) and one system call (`write`)

```
 CPU     ID    FUNCTION:NAME
   0  80158       exit:entry

 CPU     ID    FUNCTION:NAME
   1  80159     printf:entry

 CPU     ID    FUNCTION:NAME
   0  80159      write:entry
```

12 To display information about the number and/or types of processors installed on the system, the commad to run is `psrinfo -p -v`. For the machine `calibra`, the result is the following:

```
The physical processor has 2 virtual processors (0-1)
  x86 (GenuineIntel 206D7 family 6 model 45 step 7 clock
      1800 MHz)
        Intel(r) Xeon(r) CPU E5-2650L 0 @ 1.80GHz
The physical processor has 2 virtual processors (2-3)
  x86 (GenuineIntel 206D7 family 6 model 45 step 7 clock
      1800 MHz)
        Intel(r) Xeon(r) CPU E5-2650L 0 @ 1.80GHz
```

To map a process to a given set of virtual (or physical) processors the command to run is `pbind -b -c 0-1 ‘‘pid’’`. In this case, the "pid" process is mapped to virtual processors 0 and 1 (physical processor 0).

3 Communication between processes

2 In the original program, only the "sender" creates the file while the "receiver" opens it only in read mode. This will bring to a condition race: the order of execution of the two processes is not subjected to any constrain: the scheduler can first schedule the sender and then the receiver or in the opposite order. In the first case, everything work properly while in the second case the receiver tries to open a file which doesn't exist yet. The solution could be introduce a synchonization between processes (likely using semaphores) or create a delay in the receiver before it starts to work on the queue (using for example a `sleep` system call.

```
int main(void) {
  int pid;
  // Create two processes: one for sending data, and
     another one for receiving data
  pid = fork();
  if (pid != 0) { /* Parent */
    sleep(1);
        receiver();
        // delete the message queue
        mq_unlink(MQ_NAME);

  } else { /* Child */
        sender();
  }
  return(0);
}
```

Finally, this is the `dtrace` result run on this program (the complete dump is provided as additional material).

```
./mqtest
I am in sender
0 ?--->
1 ?--->
2 ?--->
3 ?--->
4 ?--->
5 ?--->
6 ?--->
7 ?--->
8 ?--->
9 ?--->
10 ?--->
11 ?--->
12 ?--->
13 ?--->
14 ?--->
I am in receiver
---> 0
---> 1
---> 2
---> 3
---> 4
---> 5
---> 6
---> 7
---> 8
---> 9
---> 10
---> 11
---> 12
---> 13
---> 14
 CPU     ID                    FUNCTION:NAME
   2  85273             tsort_unsigned:entry
   2  85272                      tsort:entry
   2  85210                set_environ:entry
   2  85082                 lookup_sym:entry
   2  85102                   elf_hash:entry
   2  85081                _lookup_sym:entry
```

3 For this exercise, there are three main actors: a sender (son11), a forwarder (son2) and a receiver (son12).
This is source code of the sender:

```
int n;
mqd_t mqfd;

//printf("I am in sender\n");
init_queue (&mqfd, name, O_CREAT | O_WRONLY);
```

```
for (n = 0; n < MAX_MSG; n++) {
    printf ("%d␣?--->\n", n);
    put_integer_in_mq (mqfd, n);
  }
put_integer_in_mq (mqfd, OVER);

/* close queue */
mq_close (mqfd);
```

This is source code of the forwarder:

```
int data_from_s11;
mqd_t mq11_2, mq2_12;

init_queue (&mq2_12, "/mq2_12", O_CREAT | O_WRONLY);
init_queue (&mq11_2, "/mq11_2", O_CREAT | O_RDONLY);

 while (1) {
    data_from_s11 = get_integer_from_mq (mq11_2);
    put_integer_in_mq(mq2_12, data_from_s11);

    if (data_from_s11 == OVER)
      break;

    printf("--->␣%d␣--->\n", data_from_s11);
 }
```

This is source code of the receiver:

```
int d;
mqd_t mqfd;

// printf("I am in receiver\n");
init_queue (&mqfd, name, O_CREAT | O_RDONLY);
while (1) {
    d = get_integer_from_mq (mqfd);
    if (d == OVER)
      break;
    printf ("--->␣%d\n", d);
  }
/* Close queue */
mq_close(mqfd);
```

This is the source code needed to spawn the correct process tree:

```
int main(void) {
  pid_t ret1, ret2;

  ret1 = fork();
  if (ret1 < 0)
    perror("Error␣in␣process␣creation:␣SON1␣cannot␣be␣
        created");
```

```
  else if (ret1 == 0)
    son1();
  else {
      ret2 = fork();
      if (ret2 < 0)
        perror("Error in process creation: SON2 cannot 
            be created");
      else if (ret2 == 0)
        son2();
      else {
        int i;
        for (i = 0; i < 2; i++)
          wait(NULL);
      }
  }

  return(0);
}

void son1(void) {
  pid_t ret11, ret12;

  ret11 = fork();
  if (ret11 < 0)
    perror("Error in process creation: SON11 cannot be 
        created");
  else if (ret11 == 0)
    son11();
  else {
     ret12 = fork();
     if (ret12 < 0)
       perror("Error in process creation: SON12 cannot 
           be created");
     else if (ret12 == 0)
       son12();
     else {
       int i;
       for (i = 0; i < 2; i++)
         wait(NULL);
       exit(0);
     }
```

This is the terminal dump of a run of this program:

```
0 ?--->
1 ?--->
2 ?--->
3 ?--->
4 ?--->
5 ?--->
6 ?--->
```

```
7 ?--->
8 ?--->
9 ?--->
10 ?--->
11 ?--->
12 ?--->
13 ?--->
14 ?--->
---> 0 --->
---> 1 --->
---> 2 --->
---> 3 --->
---> 4 --->
---> 5 --->
---> 6 --->
---> 7 --->
---> 8 --->
---> 9 --->
---> 10 --->
---> 11 --->
---> 12 --->
---> 13 --->
---> 14 --->
---> 0
---> 1
---> 2
---> 3
---> 4
---> 5
---> 6
---> 7
---> 8
---> 9
---> 10
---> 11
---> 12
---> 13
---> 14
```

Bonus For the bonus question, the same forwarding operation is now performed also by the son1 and this is the source code to do that:

```
int data_from_s11;
mqd_t mq11_1, mq1_12;

init_queue (&mq1_12, "/mq2_12", O_CREAT |
    O_WRONLY);
init_queue (&mq11_1, "/mq11_2", O_CREAT |
    O_RDONLY);
```

```
    while (1) {
      data_from_s11 = get_integer_from_mq (mq11_1);
      put_integer_in_mq(mq1_12, data_from_s11);

      if (data_from_s11 == OVER)
        break;

      printf("--->s1:␣%d␣--->\n", data_from_s11);
    }

    int i;
    for (i = 0; i < 2; i++)
      wait(NULL);
    exit(0);
```

The only different with respect to the previous program is the inclusion of two `OVER` number from the sender. This is need to guarantee that both son1 and son2 terminate their executions.

```
  int n;
  mqd_t mqfd;

  //printf("I am in sender\n");
  init_queue (&mqfd, name, O_CREAT | O_WRONLY);
  for (n = 0; n < MAX_MSG; n++) {
      printf ("%d␣?--->\n", n);
      put_integer_in_mq (mqfd, n);
    }
  put_integer_in_mq (mqfd, OVER);
  put_integer_in_mq (mqfd, OVER);
  /* close queue */
  mq_close (mqfd);
  return ;
```

Finally, this is the terminal dump of this last program.

```
0 ?--->
1 ?--->
2 ?--->
3 ?--->
4 ?--->
5 ?--->
6 ?--->
7 ?--->
8 ?--->
9 ?--->
10 ?--->
11 ?--->
12 ?--->
--->s1: 1 --->
```

```
--->s2: 0 --->
--->s1: 2 --->
--->s2: 3 --->
13 ?--->
--->s1: 4 --->
--->s2: 5 --->
--->s2: 7 --->
14 ?--->
--->s1: 6 --->
--->s2: 8 --->
--->s1: 9 --->
--->s2: 10 --->
---> 0
---> 1
---> 2
---> 3
---> 4
---> 5
--->s1: 11 --->
---> 6
---> 7
---> 8
---> 9
--->s2: 12 --->
--->s2: 14 --->
--->s1: 13 --->
---> 10
---> 11
---> 12
---> 13
---> 14
```