Linux Process Control
Signals and Signal Handling in programs

Prof. Rossano Pablo Pinto

March 2014 - v0.1

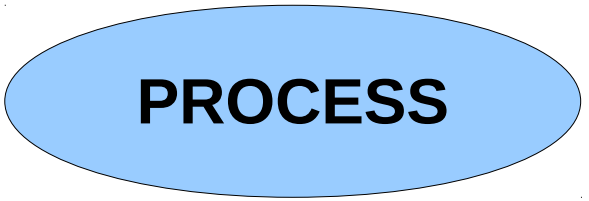
March 2017 - v0.8

Agenda

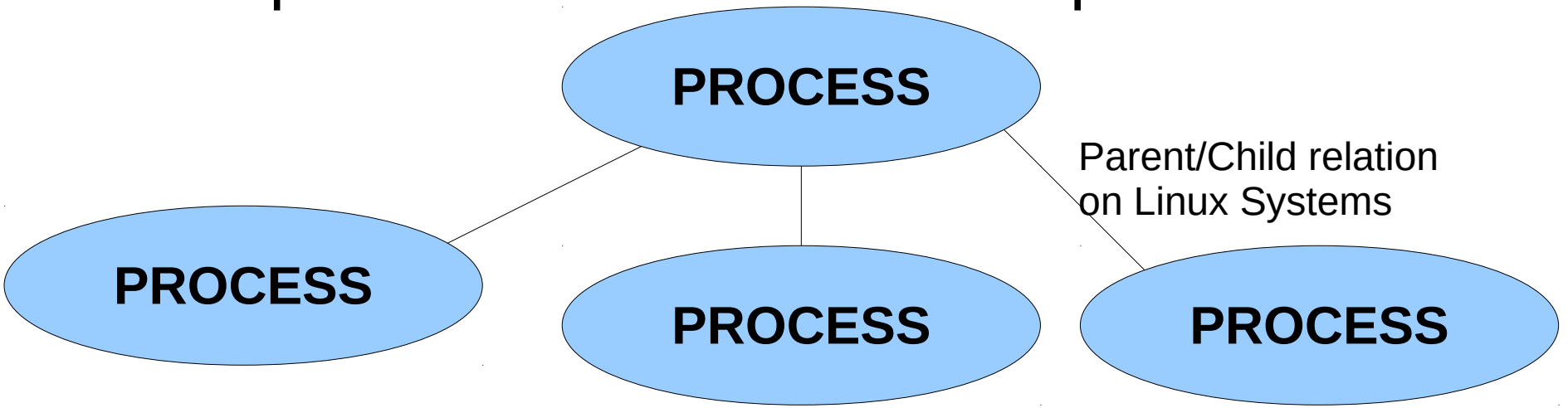
- Introduction
- Linux booting - 1st Proc. Creation
- Process Attributes
- Linux Process States
- Listing Processes
- Signals (Controlling processes)
- Using signals in programs
- Threads

Introduction

- Multiprogramming systems use an abstraction to control several concurrent running programs:



- Each process can create other processes



Introduction : Process Definition

- Definition (a simple one): a running program!
- Definition (a not so simple one):
  - An address space +
  - A set of kernel data structures to keep information about the running program. Ex.:
    - The process address space map
    - The process current state, execution priority
    - resources, signal mask
    - Process owner ….
- Reminder: PCB (Process Control Block)

Introduction

How are processes created ?

Is there a life-cycle ?

How are they controlled ?

Introduction

How are processes created ?

- 1st process: kernel creates the first process
- 2nd + are created by OTHER PROCESSES

Is there a life-cycle ?

- Yes. In each stage of the life-cycle, the process is in a different STATE.

How are they controlled ?

- Using SIGNALS

Linux booting:
1st Process creation

Linux booting - 1st Proc. Creation

- Basic concepts
  - Kernel space
    - Kernel
  - User space
    - Init
      - All services
- Hardware support in PC for multiprogramming
  - IA32e (AMD64) + EFI
    - Protected mode
    - Memory protection
    - 4 rings of protection
    - Instructions: Priviledged x non-priviledged

Linux booting - 1st Proc. Creation

- Example: Linux/IA32e (AMD64) with UEFI and GRUB
  - Power-on (real-mode)
  - CPU fetches first instruction <- ROM (UEFI)
  - UEFI switches processor to protected-mode
  - UEFI switches processor to Long (64 bits)
  - UEFI looks for a partition of type ESP (EFI System Partition - EF00)
  - UEFI loads an EFI application (for instance, GRUB)
  - GRUB loads the linux kernel to the memory and hands-off the control to Linux
  - Linux executes a bunch of routines to configure itself
  - The very last thing Linux does during initialization is the creation of the first process of the system:
    - the init
    - `/usr/src/linux-4.4.5/init/main.c (line 960)`

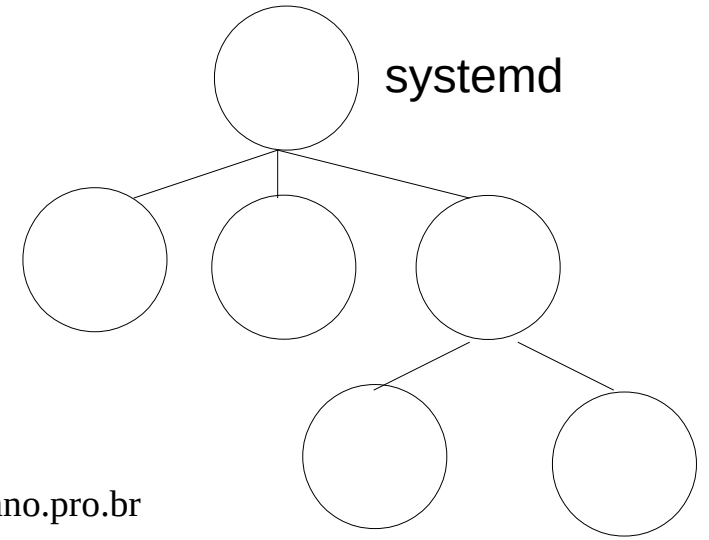
Linux booting - 1st Proc. Creation

```
// /usr/src/linux-4.4.5/init/main.c (line 960):
if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
                return 0;
        panic("Requested init %s failed  (error %d).",
                 execute_command, ret);
}
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
          return 0;

panic("No working init found.  Try passing init= option to kernel."
       “See Linux Documentation/init.txt for guidance.”);
```

Linux booting - 1st Proc. Creation

- systemd reads the directories `/usr/lib/systemd/system`, `/etc/systemd/system` and `/etc/systemd/system/[name.type].d/*.conf`, and loads all services that must run at boot time
- systemd becomes the father (grandfather/great grandfather...) of all processes of the system:



Process Attributes

- Process Important Attributes:
  - PID (Process IDentification)
  - PPID (Parent Process IDentification)
  - UID (User ID)
  - EUID (Effective User ID)
  - Status
  - Niceness
  - Control Terminal

Process Attributes

- `/proc` filesystem
  - An interface to get and configure the system attributes
  - `/proc/[PID]/`
    - Information for the process with PID. Some files:

      `cmdline` – the complete command line for the process

      `stat` – Status information about the process (used by ps - see ps reading /proc with `strace ps |& grep /proc`)

      `status` – information in /proc/[pid]/stat and /proc/[pid]/statm in a format that's easier for humans

Process Attributes

- … Information for the process with PID. Some files:

  `maps` – currently mapped memory regions and their access permissions

  Example:

  ```
  ps ax | grep emacs
  ```

```
16002 pts/0    S      0:14 emacs create-timelapse.sh
```

  ```
  cat /proc/16002/maps
  ```

  Output next page

Process Attributes

```
08048000-0820c000  r-xp   00000000 08:01 17487     /usr/bin/emacs23-x
0820c000-08691000  rw-p   001c3000 08:01 17487     /usr/bin/emacs23-x
091a3000-09537000  rw-p   00000000 00:00 0         [heap]
b26c6000-b2726000  rw-s   00000000 00:04 6717478   /SYSV00000000 (deleted)
b2726000-b2741000  r--s   00000000 08:01 262568    /usr/share/mime/mime.cache
b2741000-b2f9f000  r--p   00000000 08:01 262580    /usr/share/icons/hicolor/icon-theme.cache
b2f9f000-b5f05000  r--p   00000000 08:01 294494    /usr/share/icons/gnome/icon-theme.cache
b5f05000-b6143000  r--p   00000000 08:01 261373    /usr/share/icons/Tango/icon-theme.cache
...
```

Process Attributes

The second column provides the address space permissions. The permissions are represented by the following letters:

`r = read`

`w = write`

`x = execute`

`s = shared`

`p = private (copy on write)`

Linux Process States

- R = Running/ Runnable (on run queue)
- S = Sleeping
- Z = Zombie (not reaped by it's parent)
- D = Uninterruptible sleep (usually IO)
- T = Stopped
- t = Tracing stop
- X = dead (never appears or should never be seen)
- W = paging (not valid since 2.6.xx)

Linux Process States

- Some flags
  - < = Higher than normal priority
  - N = Lower than normal priority
  - L = pages r locked in memory (can't be paged out)
  - s = session leader
  - l = is multi-threaded (uses CLONE_THREAD flag in clone syscal)
  - + = is in the foreground process group

Listing Processes

- Most common commands
  - ps - instant system photography
  - top - shows system photography every 5 seconds (default)
  - These programs depend on the proc filesystem mounted on /proc
    - **systemd automatically mounts /proc**
    - with systemd it's not possible to umount /proc anymore as it was with SysV init
    - `fuser -v -m /proc` shows that PID 1 is using /proc (you can only umount unused fs and you can't SIGKILL PID 1)

Listing Processes

- ps examples (several others from '`man ps`')
  - `ps` - PID, TTY, TIME, CMD (from that shell only)
  - `ps aux` - USER, PID, %CPU, %MEM, VSZ, RSS, TTY, STAT, START, TIME, COMMAND
  - `ps ax` - PID, TTY, STAT, TIME, COMMAND
  - `ps -ejH` - PID, PGID, SID, TIME, CMD
  - `ps -auroot` - PID, TTY, TIME, CMD (all processes owned by root)

Listing Processes

- ps example with customizable fields
  - `ps -Luroot -o ppid,pid,tid,stat,wchan,cmd`
    - Shows all processes from User root
    - It shows the columns
      - Parent Process ID
      - Process ID
      - Thread ID
      - Status
      - The name of the event the “S state” is waitng for
      - Command line

Signals (Controlling processes)

- What's out there as a process controlling mechanism?
  - Signals - it's a special message that is sent to a process to sinalize some condition
  - `kill` - command to send a signal to a process
  - List available signals: `kill -l`
  - `man 7 signal`

Signals (Controlling processes)

- Important ones (bare minimal to master):
  - SIGHUP (1) - nowadays, usually used to signal a process to reread it's conf file
  - SIGINT (2) - Ctrl-C from terminal (terminates the process)
  - SIGQUIT (3) - similar to SIGTERM but generates a core dump (some programs catch this signal and do some other thing...)
  - SIGKILL (9) - destroy process from the kernel

Signals (Controlling processes)

- Important ones (bare minimal to master):
  - SIGSTOP (19) - gets it off from the run queue
  - SIGTSTP(20) - Ctrl-Z/Terminal Stop
  - SIGCONT (18) - reenables stopped process
  - SIGSEGV (11) - sent by kernel to offending process
  - SIGTERM (15) - similar to kill but gives a chance to the process to terminate “nicelly” (for instance to do some finishing task and invoke exit syscall)

Signals (Controlling processes)

- Important ones (bare minimal to master):
  - SIGUSR1 (10), SIGUSR2 (12) - They don't have a default meaning
    - Ex.: Apache uses SIGUSR1 as a request to restart
- Default behavior
- Programmed behavior (trapped/caught signals)
- “Untoucheable” behaviors - Ex.: SIGKILL and SIGSTOP

Signals (Controlling processes)

- Sending a signal to a process
  - Most common programs: `kill`, `pkill`
- kill sintax

```
kill SIGNAL PID
```

- Example:

```
kill -SIGKILL [SOMEPROCESSID]

kill -9 [SOMEPROCESSID]
```

Signals (Controlling processes)

`xcalc &`

`pgrep xcalc` (suppose it returns 5533)

`kill -SIGSTOP 5533`

`ps ax | grep xcalc` (observe the T status - try to use the calculator)

`kill -SIGCONT 5533`

`ps ax | grep xcalc` (observe the S status - try to use the calculator)

Signals (Controlling processes)

```
kill -SIGKILL 5533
```

```
pgrep xcalc
```
(process has died)

Signals (Controlling processes)

- pkill sintax:
  - `pkill -SIGNAL [ATTRIBUTES]`
- Example
  - `pkill –SIGHUP syslogd`
    - make syslog rereads its conf file
  - `pkill –SIGTERM –u albert`
    - sends a SIGTERM to all albert processes

Using signals in programs

- Reminder: PID and PPID

```
00 /* Author: rossano at gmail dot com */
01 #include <stdio.h>
02 #include <unistd.h>
03
04 int main() {
05   printf("This process PID is %d\n", (int) getpid());
06   printf("This process PPID is %d\n", (int) getppid());
07   return 0;
08 }
```

Using signals in programs

- Reminder: fork, parent/child relation

```
00 /* Author: rossano at gmail dot com */
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04
05 int main() {
06   pid_t pid=0;
07   printf("Parent PID is %d\n", (int) getpid());
08
09   pid = fork();
10   if(pid != 0) {
11     printf("This is the parent process, PID is %d\n", (int) getpid());
12     printf("Child PID is %d\n", (int) pid);
13   }
14   else printf("This is the child process, PID is %d\n", (int) getpid());
15   return 0;
16 }
```

Using signals in programs

- Send signal to process: kill - `man 2 kill`

NAME

kill - send signal to a process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

DESCRIPTION

The kill() system call can be used to send any signal to any process group or process.

Using signals in programs

- Send signal to process: kill example:

```
//send_signal.c - Author: rossano at gmail dot com
#include <stdio.h> // printf
#include <stdlib.h> // atoi
#include <sys/types.h> // pid_t type definition used in kill syscall
#include <signal.h> // kill

int main(int argc, char **argv) {
  int sig=0, pid=0, ret=0;

  if (argc < 2) {
    printf("Usage: %s SIGNAL_NUMBER PID\n",argv[0]);
    exit(0);
  }

  sig=(int)atoi(argv[1]);
  pid=(int)atoi(argv[2]);
  ret=kill(pid, sig);

  return ret;
}
```

Using signals in programs

- Send signal to process: kill example: Compile/test

  `gcc -o send_signal send_signal.c`

  `xterm &`

  `ps ax | grep xterm` (returns the PID 23152)

  `./send_signal 19 23152` (it's the SIGSTOP)

  `ps ax | grep xterm` (state changed to T)

  `./send_signal 18 23152` (it's the SIGCONT)

  `ps ax | grep xterm` (state changed to S)

  `./send_signal 9 23152` (xterm is eliminated)

Using signals in programs

- Control signal behavior: signal - man signal

SYNOPSIS

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION

The behavior of signal() varies across Unix versions, and has also varied historically across different versions of Linux. Avoid its use: use sigaction(2) instead.

Using signals in programs

- Control signal behavior: signal - man signal

SYNOPSIS

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION

The behavior of signal() varies across Unix versions, and has also varied historically across different versions of Linux. **Avoid its use: use sigaction(2) instead**.

Using signals in programs

- Control signal behavior: signal example (capture CTRL-C from terminal):

```
// conf_signal.c - Author: rossano at gmail dot com
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
void my_sigint_handler() {
  int c;
  printf("Are you sure you want to terminate program [y/n]?");
  c = getchar();
  if(c == 'y')   exit(0);
}
int main(int argc, char **argv) {
    signal(SIGINT, my_sigint_handler);
  while(1) {}
  return 0;
}
```

Using signals in programs

- Control signal behavior: sigaction - man sigaction

NAME

sigaction - examine and change a signal action

SYNOPSIS

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

DESCRIPTION

The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. (See signal(7) for an overview of signals.)

Using signals in programs

- Control signal behavior: sigaction example:

```
/* Author: rossano at gmail dot com */
#include <signal.h>    // sig_atomic_t
#include <string.h>    // memset
#include <stdio.h>     // printf

sig_atomic_t counter = 0;

void my_handler(int signum) {
  ++counter;
  printf("I received signal %d\n",signum);
}

int main() {
  struct sigaction sa;
  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = &my_handler;
  sigaction(SIGUSR1, &sa, NULL);

  while(counter < 3) {}

  printf("I received %d SIGUSR1 signals. Terminating!!!\n", counter);
  return 0;
}
```

Using signals in programs

- Control signal behavior: sigaction example: compile/test

```
gcc -o sigusr1 sigusr1.c
```

```
./sigusr1
```

- From another terminal:

```
pgrep sigusr1
```
(returns 23880)

```
kill -SIGUSR1 23880
```

```
kill -SIGUSR1 23880
```

```
kill -SIGUSR1 23880
```

Using signals in programs

- For every `kill -SIGUSR1 23880` from the other terminal, the program sigusr1 prints:

  “I received signal 10”

- The last `kill -SIGUSR1 23880` makes sigusr1 print:

  “I received signal 10”

  “I received 3 SIGUSR1 signals. Terminating!!!”

Threads

- Linux creates SCHEDULING ENTITIES with the system call CLONE or FORK
  - FORK - used by glibc < 2.3.3 (uses wrapper fork())
  - CLONE - used by glibc >= 2.3.3 (still uses wrapper fork())
  - CLONE can be used to create PROCESSES and THREADS
  - CLONE offers several flags
  - Depending on the flags, the created entity is called a PROCESS or a THREAD

Threads

- Syscall clone with flags SIGCHLD is equivalent to a syscall fork.

Threads: CLONE flags for PROCESSES

- CLONE_PARENT_SETTID
- CLONE_CHILD_CLEAR_TID
- SIGCHILD

Threads: CLONE flags for THREADS < kernel 2.6

- CLONE_VM
- CLONE_FS
- CLONE_FILES
- CLONE_SIGHAND

Threads: CLONE flags for THREADS >= kernel 2.6

- CLONE_VM
- CLONE_FS
- CLONE_FILES
- CLONE_SIGHAND
- CLONE_THREAD
- CLONE_SYSVSEM
- CLONE_SETTLS
- CLONE_PARENT_SETTID
- CLONE_CHILD_CLEAR_TID

Threads: Libraries

- POSIX.1 Specification
- Thread Libraries for Linux
  - LinuxThreads
  - NPTL (Native POSIX Threads Library)
- Both libraries are a 1:1 implementation (each thread maps to a kernel scheduling entity)
- Both libraries uses CLONE in a way that a SIGKILL (and other signals when each thread has the same signal handlers) affects all the process threads (AS IT SHOULD BE)

Threads: LinuxThreads

- Original Pthreads Linux implementation
- Some compliance with POSIX
- No longer supported since glibc 2.4
- Each process (when multithreaded) is composed of: main thread, **“manager” thread**, other threads
- Signals may be sent only to specific threads
- `getpid()` returns a DIFFERENT PID for each thread

Threads: NPTL

- A little bit more compliant with POSIX
- Available since glibc 2.3.2
- Depends on kernel 2.6+
- Each process (when multithreaded) is composed of: main thread, other threads
- Signals may be sent to
  - specific threads (tgkill system call)
  - process (kill system call)
- `getpid()` returns THE SAME PID for each thread
- Creation time is 4 times as fast as LinuxThreads

Tests

```
//Author : rossano at gmail dot com
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *func1(void *uu) {
  while(1) {printf("func1 - %i\n",getpid());}
  return NULL;
}
void *func2(void *uu) {
  while(1) {printf("func2 - %i\n",getpid());}
  return NULL;
}
void *func3(void *uu) {
  while(1) {printf("func3 - %i\n",getpid());}
  return NULL;
}
```

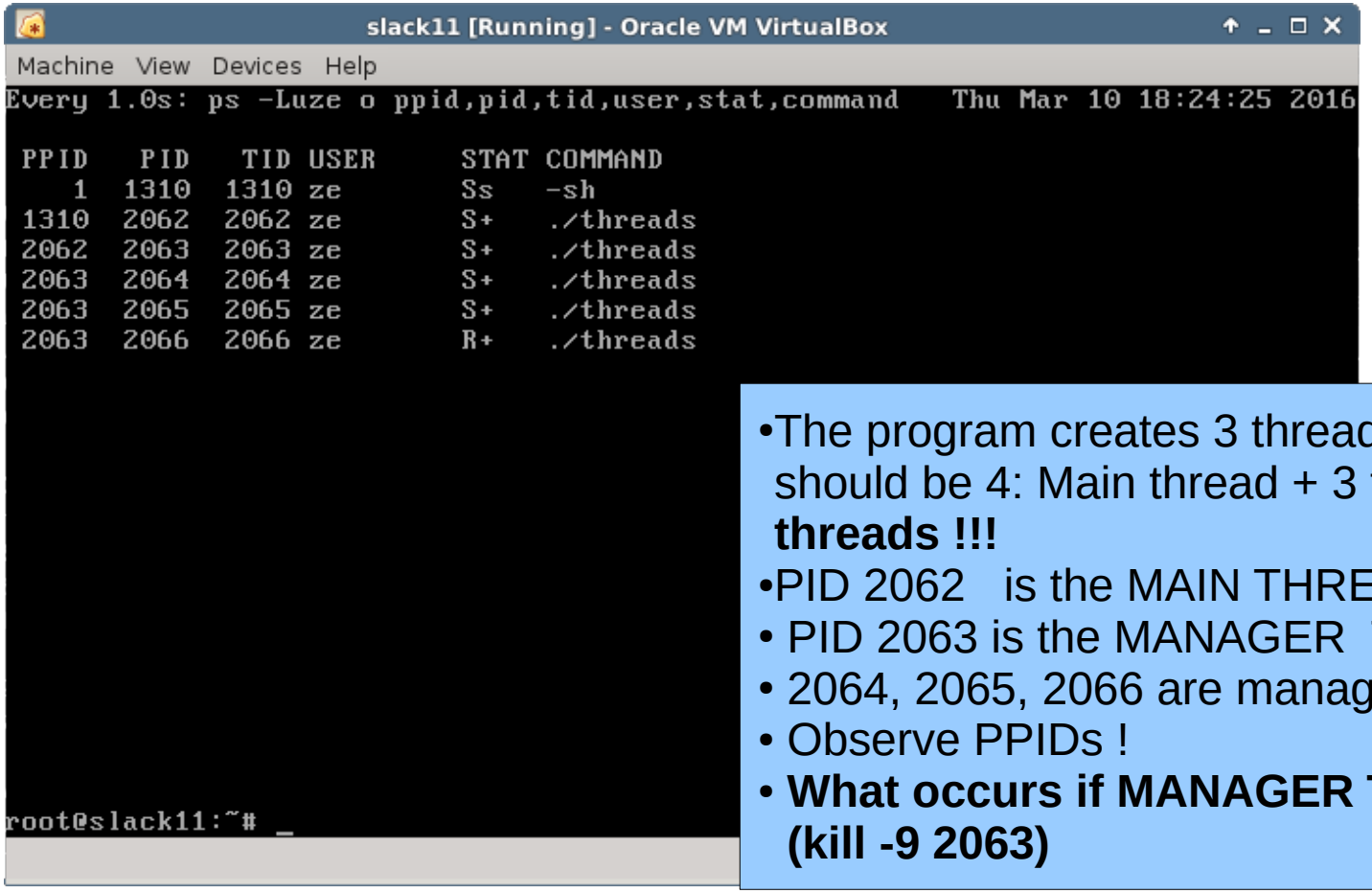
```
int main() {
  pthread_t t1, t2, t3;
  pthread_create(&t1, NULL, &func1, &"x");
  pthread_create(&t2, NULL, &func2, &"x");
  pthread_create(&t3, NULL, &func3, &"x");

  while(1) {printf("Main - %i\n",getpid());}
  return 0;
}
```

Tests

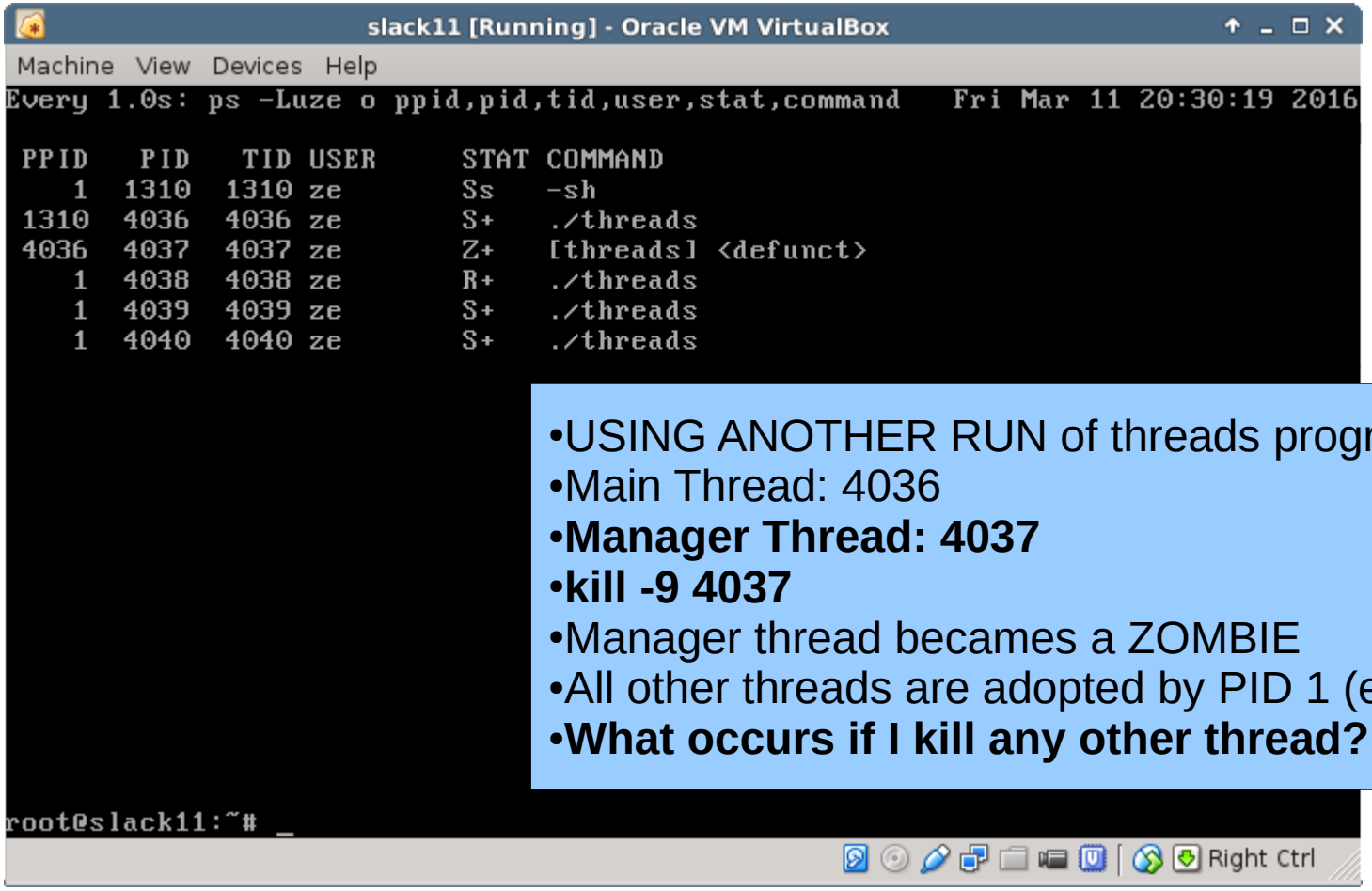
- `gcc threads.c -o threads -pthread`
- Terminal 1
  - `./threads`
- Terminal 2
  - `watch -n 1 ps -Luze o ppid,pid,tid,user,stat,command`
  - Obs.: uze means **User ze**

Tests: Slackware 11 - LinuxThreads



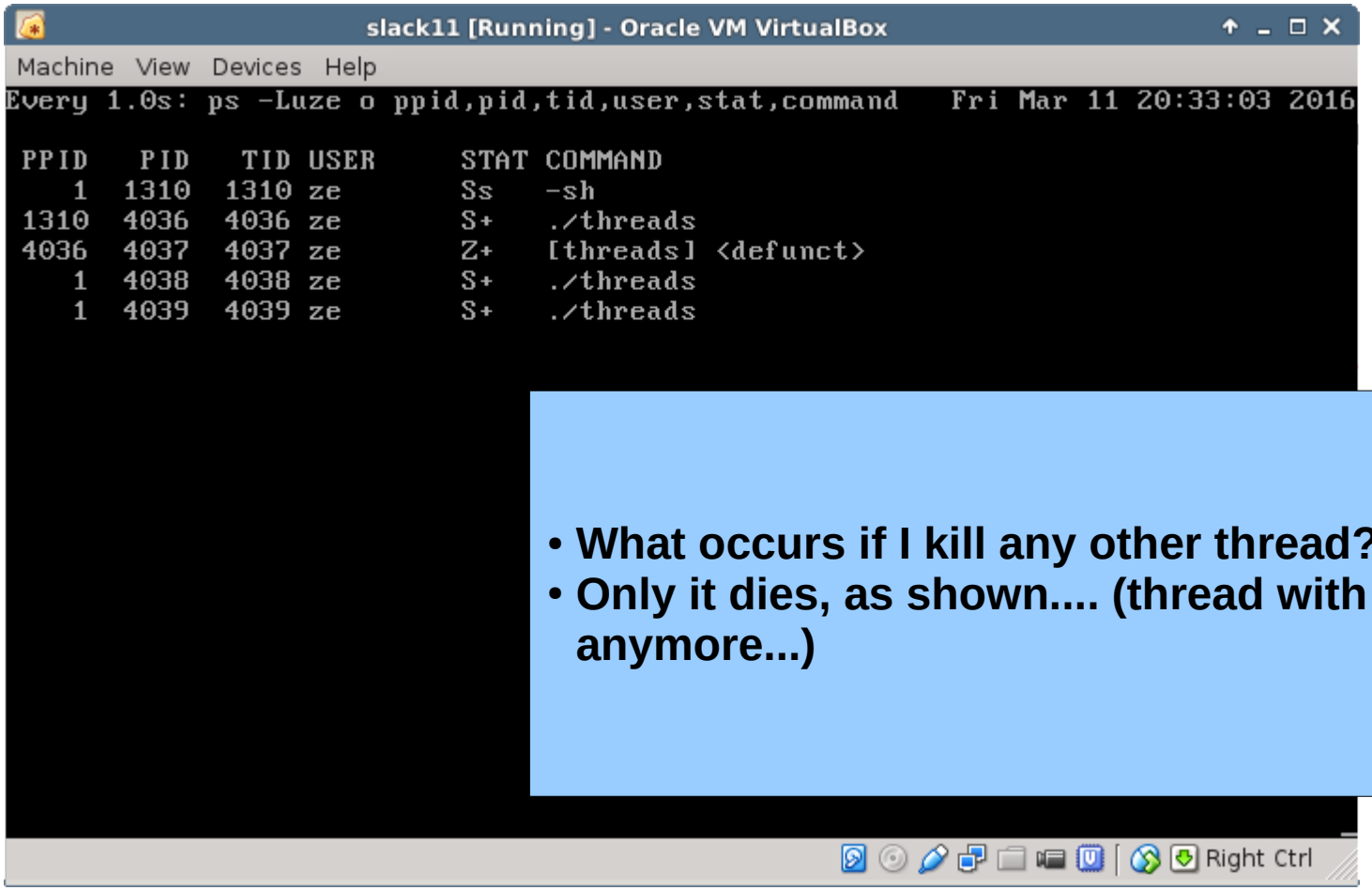
- The program creates 3 threads. The total thread count should be 4: Main thread + 3 threads! **But it shows 5 threads !!!**
- PID 2062 is the MAIN THREAD
- PID 2063 is the MANAGER THREAD
- 2064, 2065, 2066 are managed threads
- Observe PPIDs !
- **What occurs if MANAGER THREAD is killed ? (kill -9 2063)**

Tests: Slackware 11 - LinuxThreads



- USING ANOTHER RUN of threads program:
- Main Thread: 4036
- **Manager Thread: 4037**
- **kill -9 4037**
- Manager thread becames a ZOMBIE
- All other threads are adopted by PID 1 (except the main thread)
- **What occurs if I kill any other thread? (kill -9 4040)**

Tests: Slackware 11 - LinuxThreads



- **What occurs if I kill any other thread? (kill -9 4040)**
- **Only it dies, as shown.... (thread with PID 4040 is not present anymore...)**

Tests: Slackware 11 - LinuxThreads

- What occurs if any other thread, other than the manager thread, is killed?
  - All other threads associatted with the manager thread is killed, including the main thread.
- See manager thread code next slide!

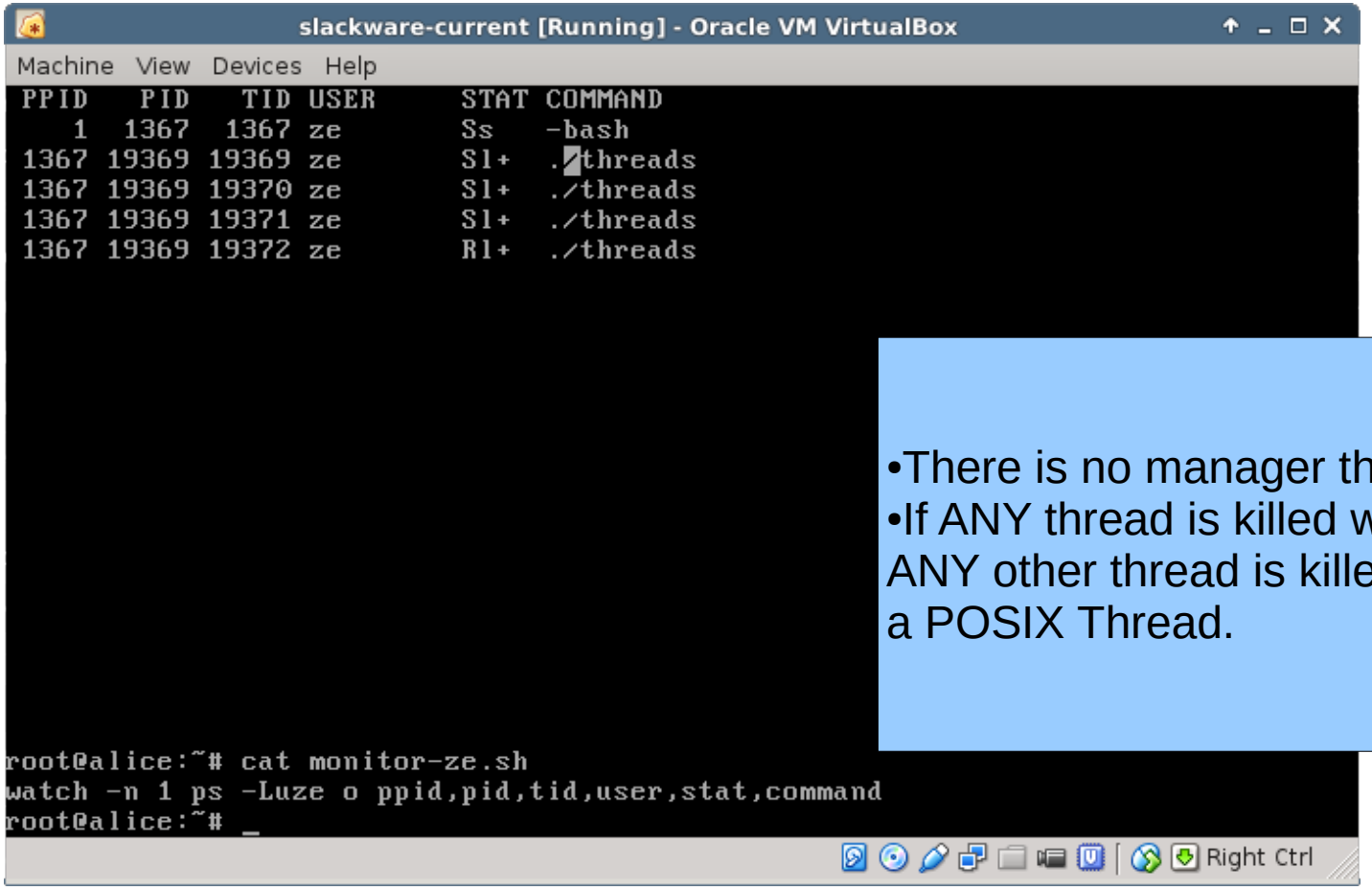
Tests: Slackware 11 - LinuxThreads

```
....
while(1) {
    n = __poll(&ufd, 1, 2000);

    /* Check for termination of the main thread */
    if (getppid() == 1) {
      pthread_kill_all_threads(SIGKILL, 0);
      _exit(0);
    }
    /* Check for dead children */
    if (terminated_children) {
      terminated_children = 0;
      pthread_reap_children();
    }
    /* Read and execute request */
    if (n == 1 && (ufd.revents & POLLIN)) {
      n = TEMP_FAILURE_RETRY(__libc_read(reqfd, (char
*)&request,
 sizeof(request)));
...
```

- Check the bold red code!
  - It shows that if the manager thread is adopted by PID 1, then main thread is dead, so kill all other threads.
  - Something similar occurs if any other managed thread is killed.

Tests: slackware 14.2 - NPTL



- There is no manager thread
- If ANY thread is killed with the “kill” syscall, then ANY other thread is killed. That's what expected for a POSIX Thread.

That's all

Priorities (under construction!!!!!!!!)

- It's all about priorities!!!
- It's possible to set soft priorities to Linux processes (we do not touch REAL-TIME priorities here)
- UNDER CONSTRUCTION

A lot of stuff

- Credentials: man 7 credentials
  - Process Group, Process Group Leader, Process Session, Process Session Leader
- man 3 exit
  - What happens when a process terminates?