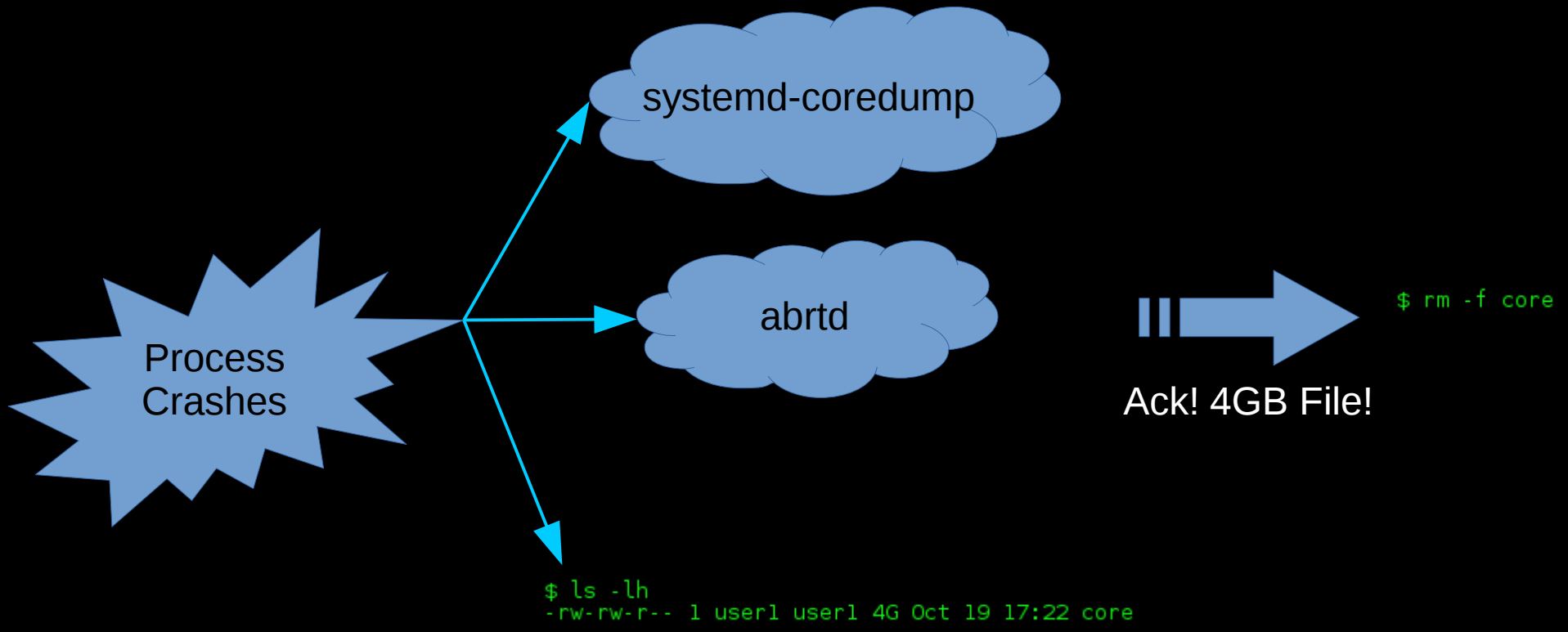
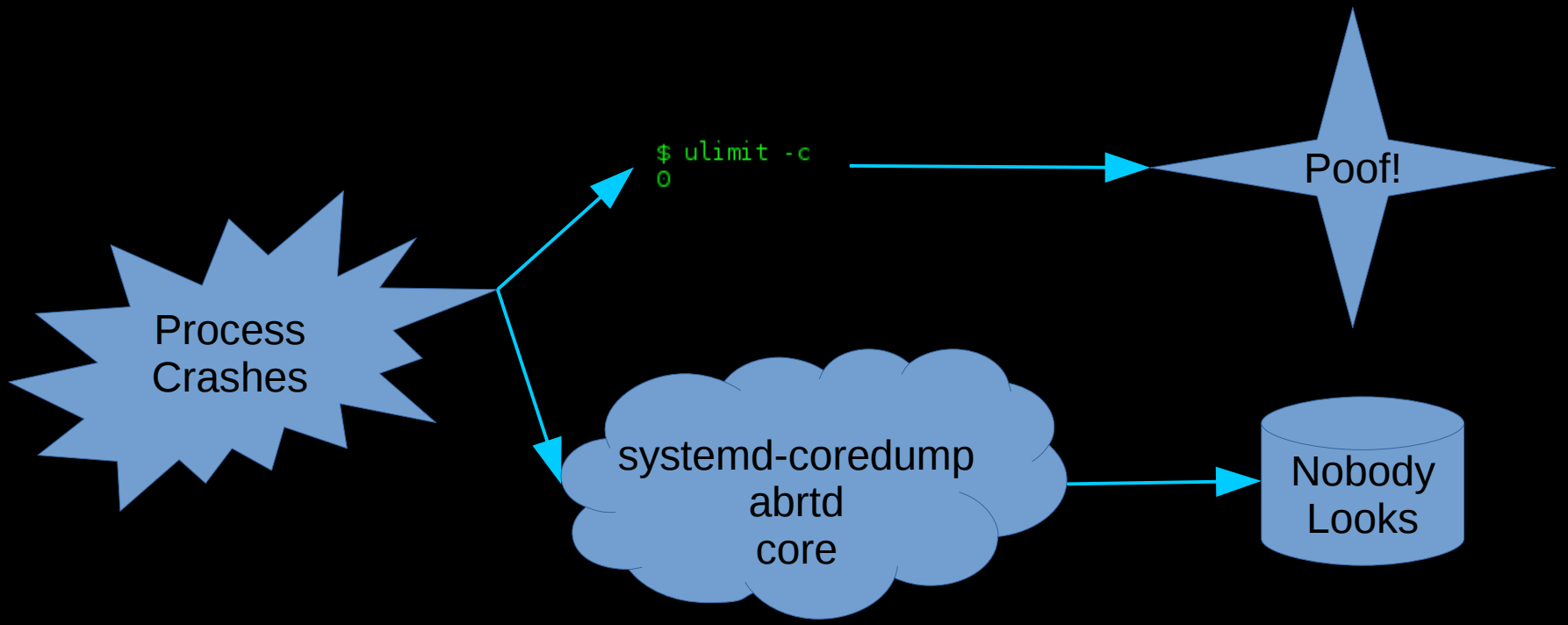
Linux Core Dumps

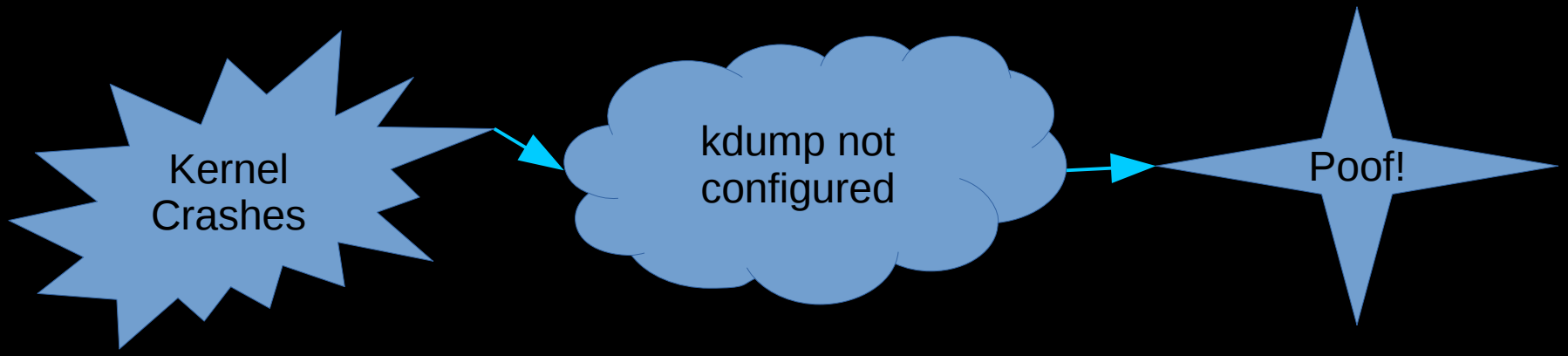
Kevin Grigorenko
kevgrig@gmail.com

Many Interactions with Core Dumps



Most Interactions with Core Dumps



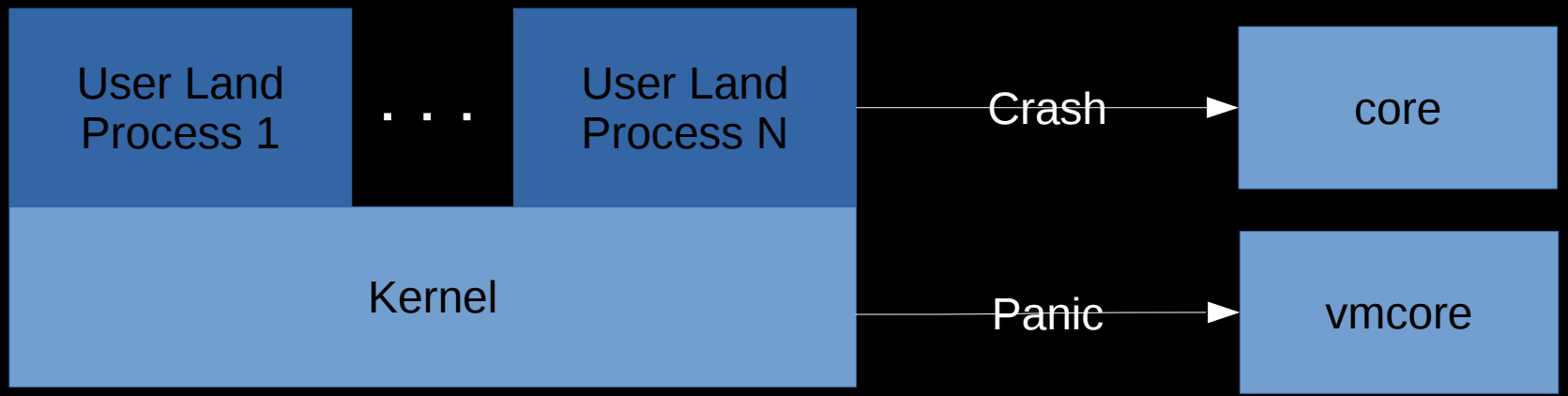


So what?

- Crashes are problems!
  - May be symptoms of security vulnerabilities
  - May be application bugs
    - Data corruption
    - Memory leaks
  - A hard crash kills outstanding work
  - Without automatic process restarts, crashes lead to service unavailability
    - With restarts, a hacker may continue trying.
- We shouldn't be scared of core dumps.
  - When a dog poops inside the house, we don't just `rm -f $poo` or let it pile up, we try to figure out why or how to avoid it again.

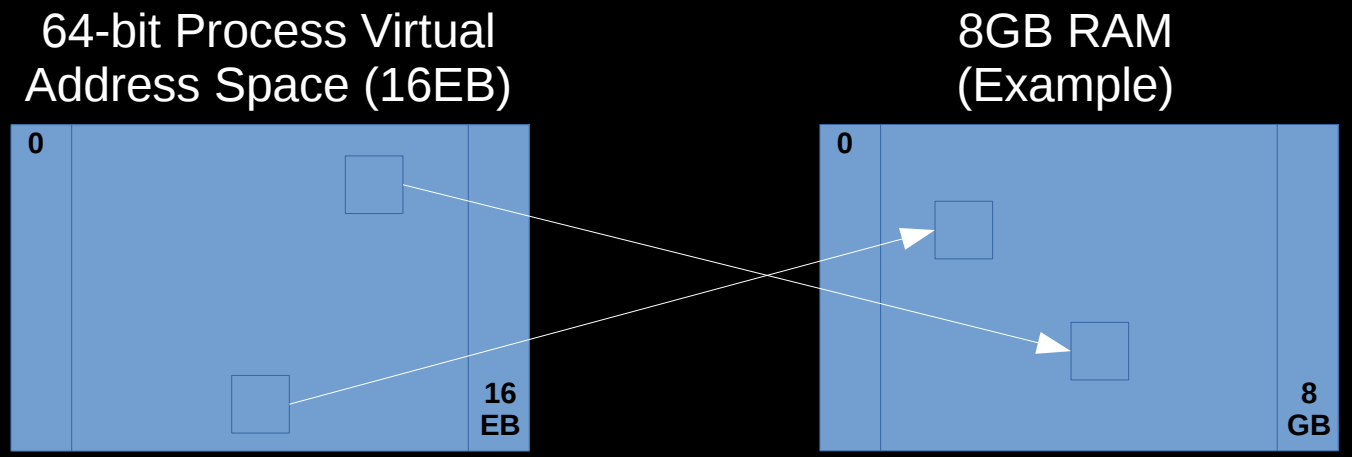
What is a core dump?

- It's just a file that contains virtual memory contents, register values, and other meta-data.
  - User land core dump: Represents state of a particular process (e.g. from crash)
  - Kernel core dump: Represents state of the kernel (e.g. from panic) and process data
- ELF-formatted file (like a program)



What is Virtual Memory?

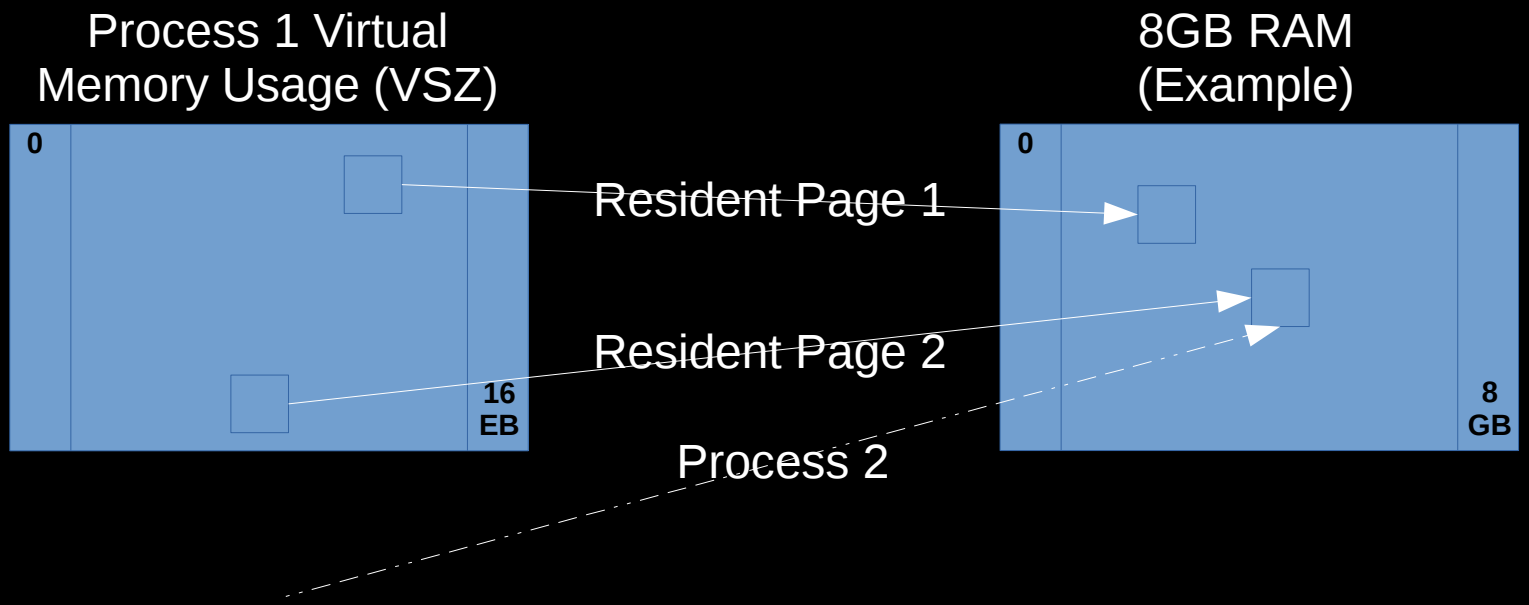
- Virtual Memory is an abstraction over physical memory (RAM/swap)
  - Simplifies programming
  - User land: process isolation
  - Kernel/processor translate virtual address references to physical memory locations



How much virtual memory is used?

- Use `ps` or similar tools to query user process virtual memory usage (in KB):
  - $ ps -o pid,vsz,rss -p 14062

```
  PID    VSZ      RSS
14062  44648   42508
```

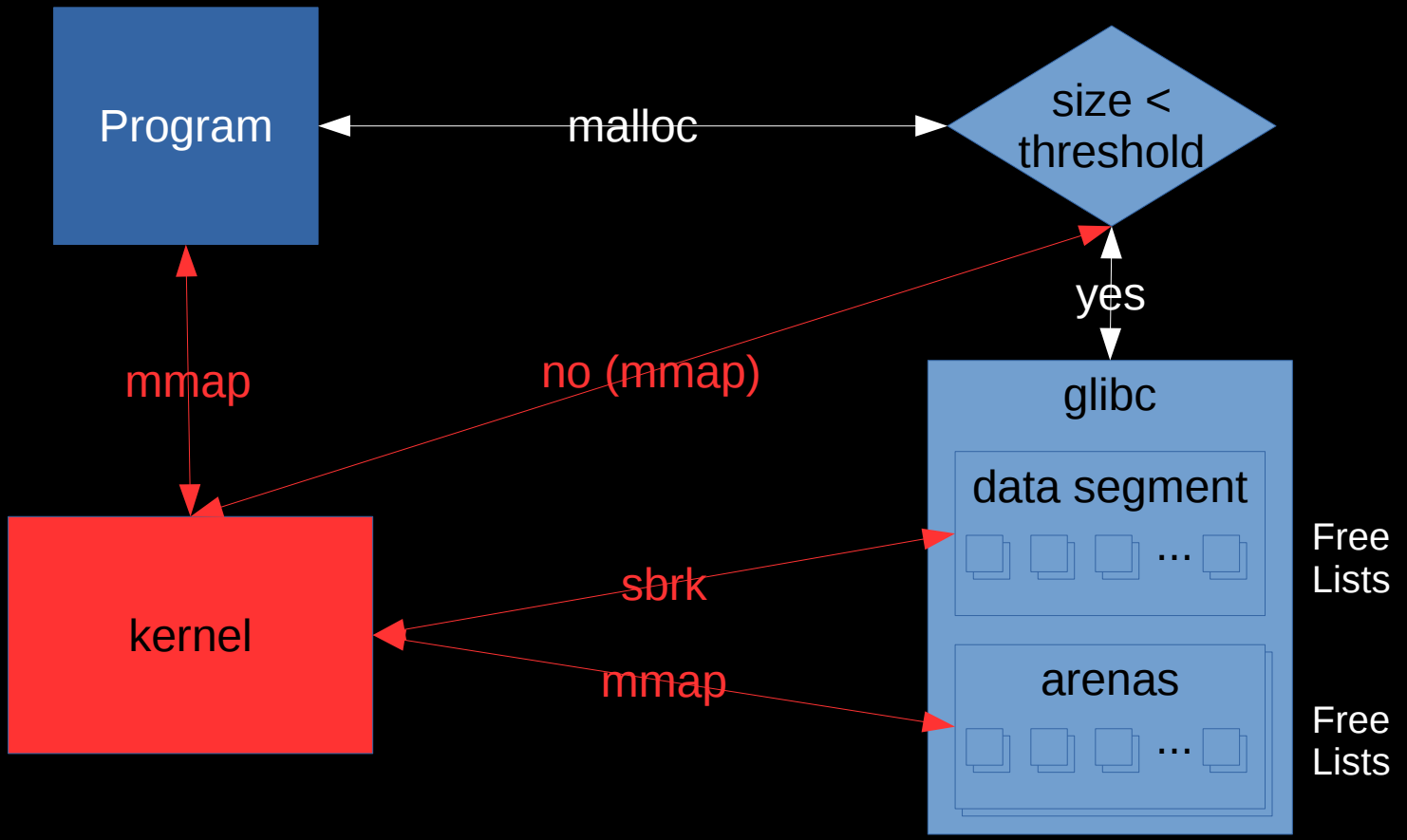


How much virtual memory is used?

- Virtual memory is broken up into virtual memory areas (VMAs), the sum of which equal VSZ and may be printed with:
  - $ cat /proc/${PID}/smaps
    00400000-0040b000 r-xp 00000000 fd:02 22151273 /bin/cat
    Size: 44 kB
    Rss: 20 kB
    Pss: 12 kB...
    - The first column is the address range of the VMA.
    - The second column is the set of permissions (read, write, execute, private copy on write).
    - The final column is the pathname if the VMA is a file mapping. If it's [heap], that's the data segment (primary malloc arena).
    - The Rss value shows how much of the VMA is resident in RAM.
    - The Pss value divides Rss by the total number of processes sharing this VMA.

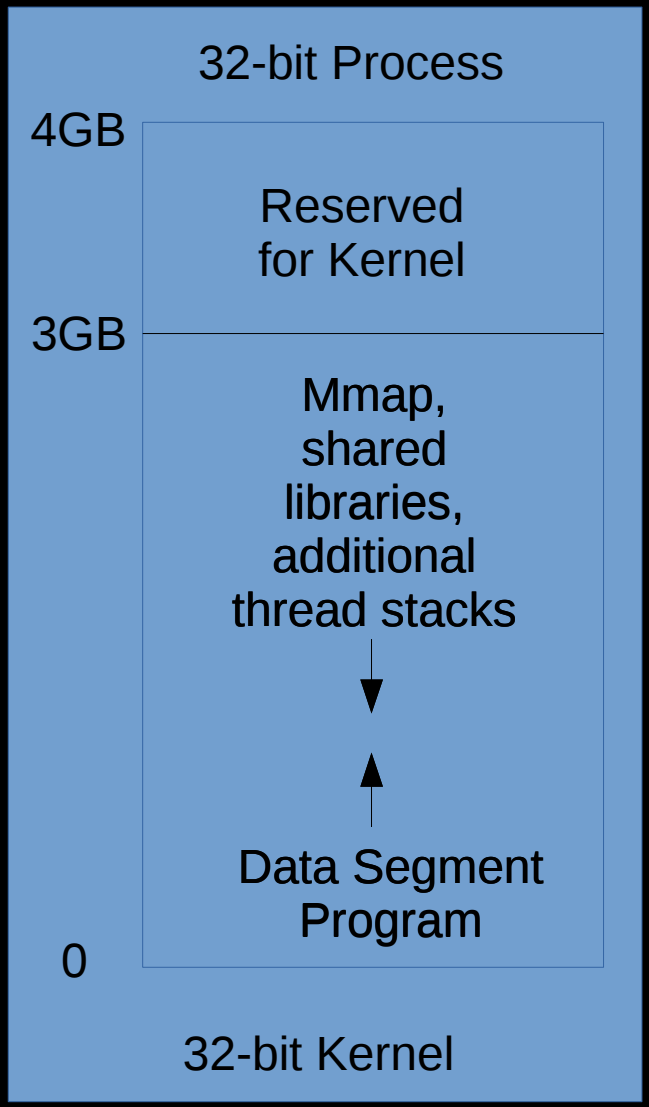
How to request virtual memory?

- malloc: request process virtual address space
  - May suffer fragmentation
- mmap (syscall): size rounded up to page size and zero'd



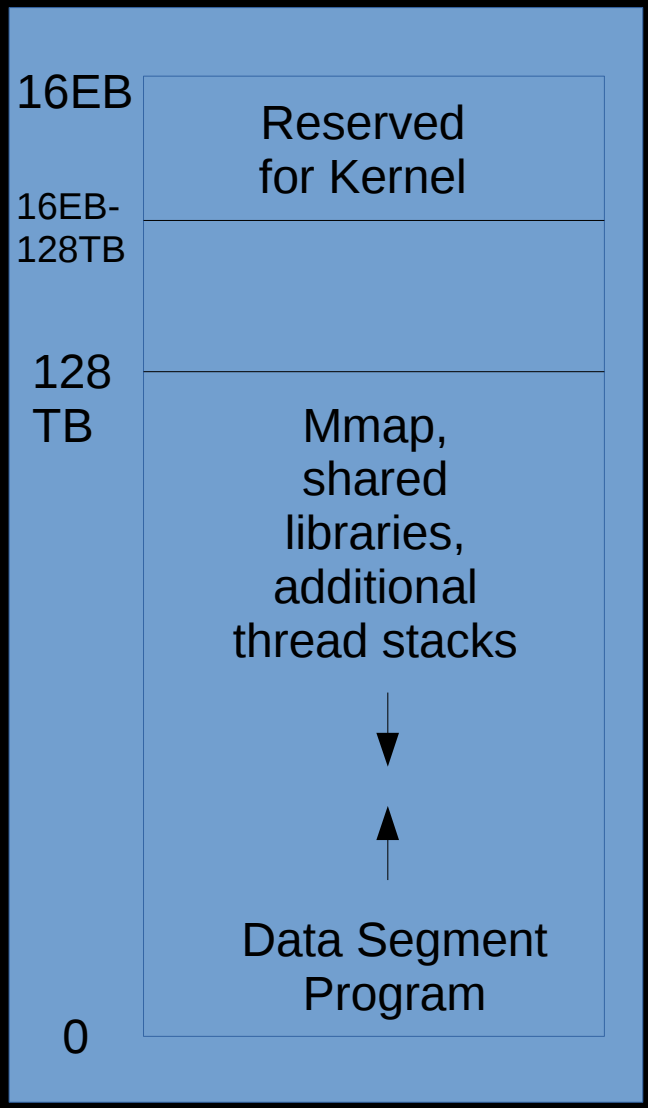
Linux 32-bit Virtual Memory Layout

- 3GB user space (2^^32), or 4GB if:
  - 32-bit process on 64-bit kernel
  - 32-bit hugemem kernel



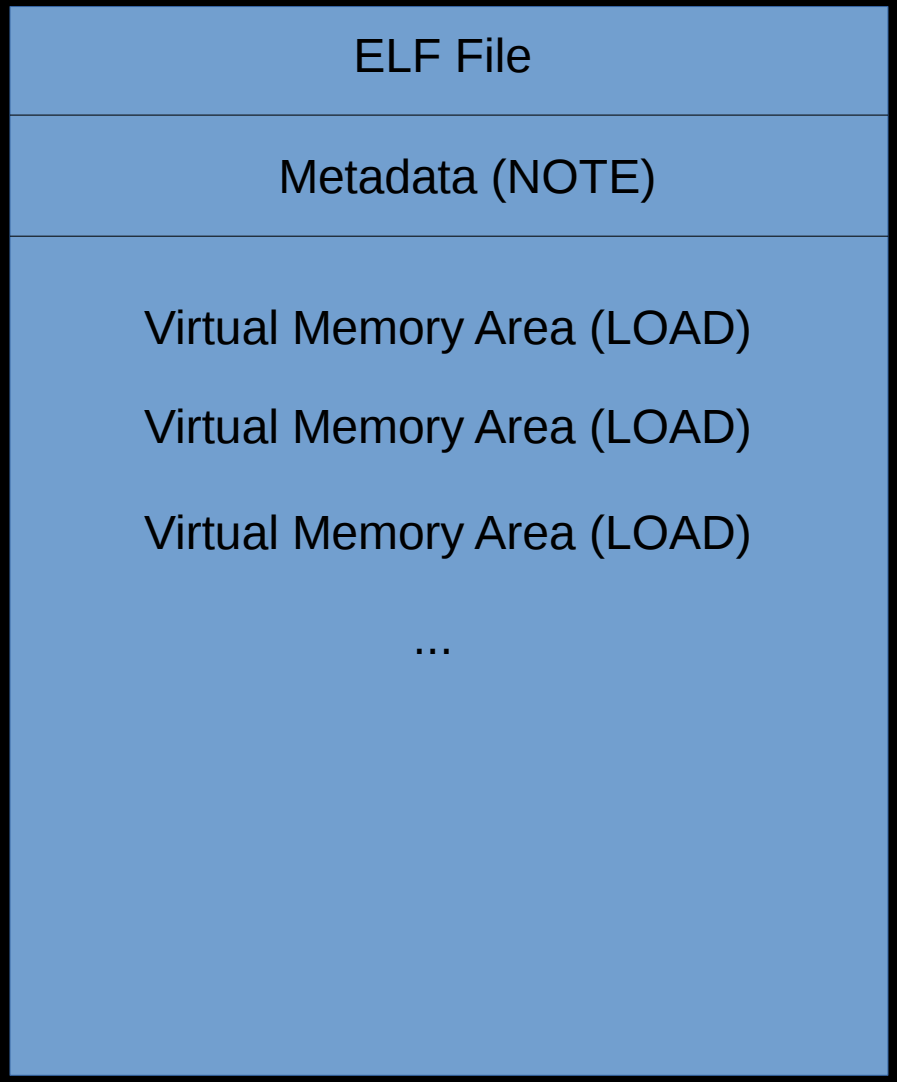
Linux 64-bit Virtual Memory Layout

- The x86_64 processor memory management unit supports up to 48-bit virtual addresses (256TB).
  - https://www.kernel.org/doc/ols/2001/x86-64.pdf
- 128TB for the program
  - 0x through 0x00007FFF'FFFFFFFF
- 128TB for the kernel
  - 0xFFFF8000'00000000 through 0xFFFFFFFF'FFFFFFFF
  - $ sudo ls -lh /proc/kcore
    -r-------- 1 root root 128T /proc/kcore



Diving in!

- Before going through the boring details of how to produce coredumps, let's assume we have one.
- Since it's an ELF-formatted file, let's see the details:
- $ readelf -h core.14391.dmp
  Class: ELF64
  Type: CORE (Core file)...
- This confirms we've got a coredump from a 64-bit process.



User Coredumps

- Next, we'll need to know which program crashed. This may be in logs, but let's just read the notes:
- $ readelf -n core.14391.dmp

```
  CORE                 0x000001de      NT_FILE (mapped files)
       Start            End              Page                 Offset
    0x400000  0x401000  0x00000000       /work/toorcon/a.out …
```

- In this case, the program is /work/toorcon/a.out

Debugging User Coredumps

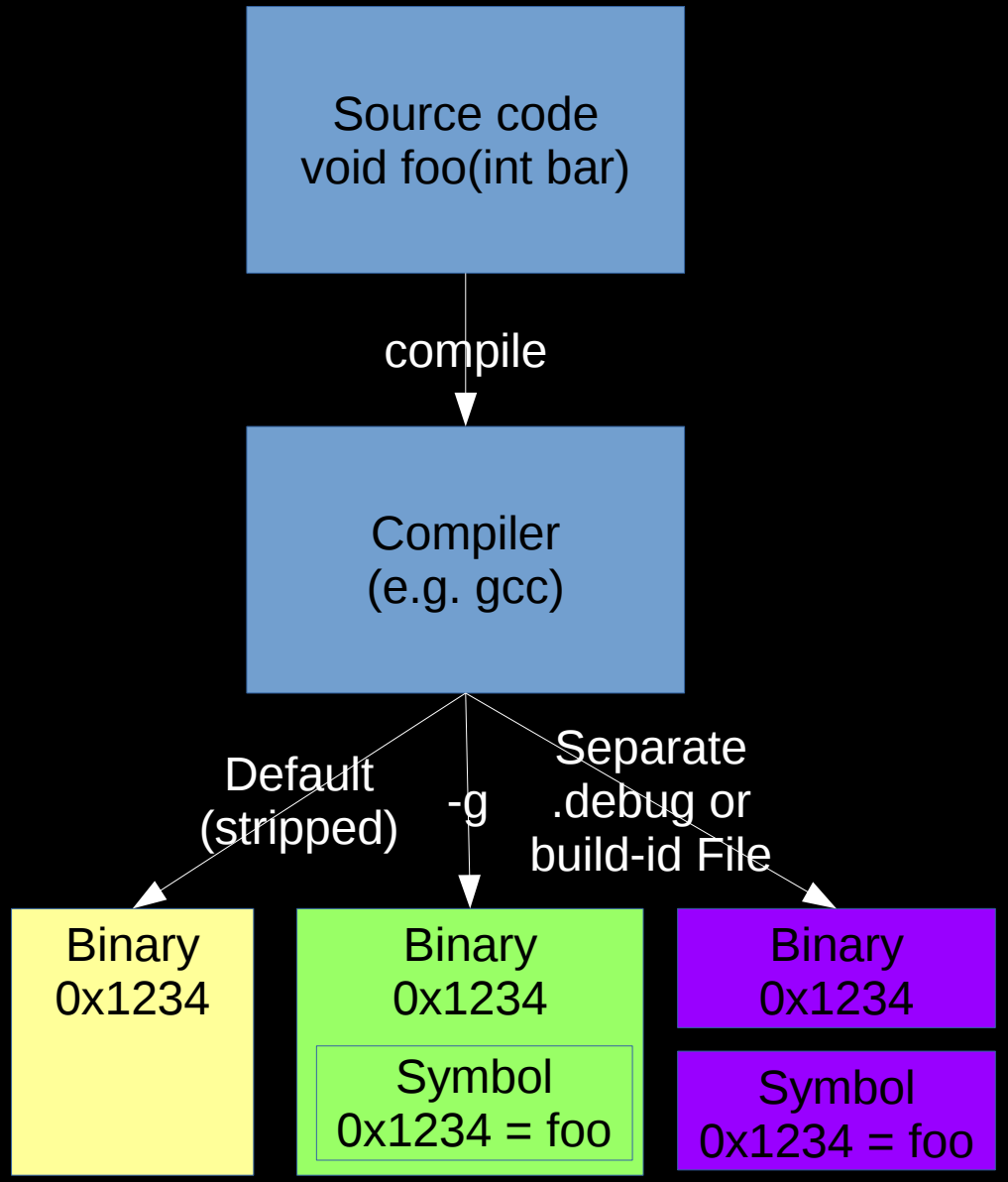
- Now that we know the program that produced the coredump, simply load `gdb` with the program and the coredump. For example:
  - $ gdb /work/toorcon/a.out core.14391.dmp
    Program terminated with signal SIGSEGV,
    Segmentation fault.
    #0 0x00007f6526f1ec8a in strlen () from /lib64/libc.so.6
    Missing separate debuginfos, use: debuginfo-install
    glibc-2.20-8.fc21.x86_64
- The (gdb) prompt awaits instructions. Type `help` for a list of commands. Type `quit` to exit.

Debugging User Coredumps

- If you're not a developer of the program, you'll just need to send them the coredump, libraries, and a stacktrace
- (gdb) bt
  - #0 0x00007f6526f1ec8a in strlen () from /lib64/libc.so.6
    #1 0x00007f6526f03d3c in puts () from /lib64/libc.so.6
    #2 0x0000000000400563 in main (argc=1, argv=0x7ffebc36a128) at test.c:6
- Even better: all stacks
  - (gdb) thread apply all bt

Symbols

- Symbols map virtual addresses to human-understandable names (functions, structures, etc.)
- Without symbols, you'll just get a bunch of addresses
- -g doesn't affect optimizations. “We recommend that you always use ‘-g’ whenever you compile a program." https://www.sourceware.org/gdb/current/onlinedocs/gdb.html



Debugging User Coredumps

- It's best to load the coredump on the same machine where it was produced since gdb will find the loaded shared libraries and any installed debuginfo symbols.
- If copying the coredump for processing on another machine, also copy the program, all shared libraries in the NOTE section and expand those files into a similar folder structure and point to that:
  - $ gdb # no parameters
    (gdb) set solib-absolute-prefix ./
    (gdb) set solib-search-path .
    # (gdb) set debug-file-directory ./path_to_debug
    (gdb) file ./path_to_program
    (gdb) core-file ./path_to_coredump

GDB: Querying virtual memory

- gdb can query a core file and produce output about the virtual address space which is similar to /proc/${PID}/smaps, although it is normally a subset of all of the VMAs:
  - (gdb) info files
    Local core dump file:
    `core.16721.dmp', file type elf64-x86-64.
    0x0000000000400000 - 0x0000000000401000 is load1
    0x0000000000600000 - 0x0000000000601000 is load2
    0x0000000000601000 - 0x0000000000602000 is load3
    0x00007fe288ca5000 - 0x00007fe288ca6000 is load4a
    0x00007fe288ca6000 - 0x00007fe288ca6000 is load4b
    0x00007fe288e58000 - 0x00007fe288e58000 is load5...

GDB Details

- Switch to a frame (list threads with `info thread` and switch threads with `thread N`):
  - (gdb) frame 2
    #2 0x0000000000400563 in main (argc=3, argv=0x7ffd47508d18) at test.c:6
    6   printf("%s\n", p);
- Check why the printf crashed:
  - (gdb) print p
    $10 = 0x0
- Understand the type of argv and then print string contents:
  - (gdb) ptype argv
    type = char **
    (gdb) print argv[0]
    $7 = 0x7ffd4750a17c "./a.out"
    (gdb) print argv[1]
    $8 = 0x7ffd4750a184 "arg1"

User coredump ulimits

- Ensure process ulimits for coredumps (-c) and files (-f) are unlimited
  - The coredump ulimit (-c) often defaults to 0, suppressing cores
  - A coredump is a file so the file ulimit (-f) also applies
- Ulimits may be soft or hard
  - Hard: the maximum value a non-root user can set
  - Soft: Sets the current limit (must be <= hard for non-root)
- Ulimits for the current shell may be queried:
  - $ ulimit -c -f
    core file size (blocks, -c) 0
    file size (blocks, -f) unlimited
- Or by process:
  - $ cat /proc/${PID}/limits | grep -e Limit -e core -e "Max file size"

Limit	Soft Limit	Hard Limit	Units
Max file size	unlimited	unlimited	bytes
Max core file size	0	unlimited	bytes

User Coredump Ulimits

- Ulimits may be set in limits.conf on a user or group basis.
- Commonly set in /etc/security/limits.conf or /etc/security/limits.d/99-cores.conf
- The following example sets file and core soft and hard ulimits to unlimited for all users
  - * - core unlimited
    * - file unlimited
- Alternatively, run the command `ulimit -c unlimited -f unlimited` in the shell that launches the program
- systemd-started processes use LimitCORE/LimitFSIZE

What produces a user coredump?

- When the kernel handles certain signals (`man 7 signal`):
  - SIGQUIT (kill -3)
  - SIGILL (kill -4)
  - SIGABRT (kill -6)
  - SIGGFPE (kill -8)
  - SIGSEGV (kill -11)
    - This is one of the most common causes of a crash when a program references invalid memory (e.g. NULL)
  - Others: SIGBUS, SIGSYS, SIGTRAP, SIGXCPU, SIGXFSZ, SIGUNUSED
- Outside the kernel: use `gcore $PID` (part of gdb)
  - Different code than the kernel: attaches gdb and dumps memory
  - Non-destructive (i.e. process continues after detach)

Where is the user coredump?

- The coredump goes to core_pattern (see `man 5 core`):
  - $ sysctl kernel.core_pattern
    kernel.core_pattern = |/usr/lib/systemd/systemd-coredump %p %u %g %s %t %e
- The default is `core` (sometimes with %p) which writes a file named `core` to the current directory of the PID
  - May include a path to use a dedicated coredump directory
- If the value starts with a `|`, then the coredump bytes are piped to that program
- Often specified in /etc/sysctl.conf or {/etc/sysctl.d|/usr/lib/sysctl.d|/run/sysctl.d}/*.conf

What's in a user coredump?

- The memory dumped is controlled with a bit mask in /proc/$PID/coredump_filter (see `man 5 core`)
  - Inherited from parent process, so you may set in the script/shell that starts the process. Example:
    - $ echo 0x7F > /proc/self/coredump_filter
- Never dumped:
  - Anything madvise'd with MADV_DONTDUMP
  - Memory the process can't read (see the `r` permission in `cat /proc/$PID/smaps`)
  - Memory-mapped I/O pages such as frame buffers

systemd-coredump

- systemd-coredump is a common user coredump handler which handles coredumps
- Configured in /etc/systemd/coredump.conf
- Defaults:
  - Store coredumps in /var/lib/systemd/coredump/
  - Use no more than 10% of that disk's space
  - Ensures cores don't cause that disk's free space to go below 15%
- systemd-tmpfiles may remove old cores

abrtd

- abrtd is an older user coredump handler
- Like systemd-coredump, modified core_pattern to something like:
  - |/usr/libexec/abrt-hook-ccpp % s % c % p % u % g % t e
- Configured in /etc/abrt/abrt.conf
- Defaults:
  - DumpLocation=/var/spool/abrt/
  - MaxCrashReportsSize=1000M

Read Memory in GDB

- Virtual memory may be printed with the `x` command:
  - (gdb) x/32xc 0x00007f3498000000
    0x7f3498000000: 32 ' ' 0 '\000' 0 '\000' 28 '\034' 54 '6' 127 '\177' 0 '\000' 0 '\000'
    0x7f3498000008: 0 '\000' 0 '\000' 0 '\000' -92 '\244' 52 '4' 127 '\177' 0 '\000' 0 '\000'...
- Another option is to dump memory to a file and then spawn an xxd process from within gdb to dump that file which is easier to read (install package vim-common):
  - (gdb) define xxd
    >dump binary memory dump.bin $arg0 $arg0+$arg1
    >shell xxd dump.bin
    >shell rm -f dump.bin
    >end
    (gdb) xxd 0x00007f3498000000 32
    0000000: 2000 001c 367f 0000 0000 00a4 347f 0000 ...6.....4...
    0000010: 0000 0004 0000 0000 0000 0004 0000 0000 ....
- For large chunks, these may be dumped to a file directly:
  - (gdb) dump binary memory dump.bin 0x00007f3498000000 0x00007f34a0000000
- Large VMAs often have a lot of zero'd memory. A simple trick to filter those out is to remove all zero lines:
  - $ xxd dump.bin | grep -v "0000 0000 0000 0000 0000 0000 0000 0000" > dump.bin.txt

Eye catchers

- Well written programs put eye catchers at the start of structures to make finding problems easiers
  - (gdb) xxd 0xF2E010 128

```
00000000: 4445 4144 4641 4444 0000 0000 0000 0000  DEADFADD........
00000010: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
00000020: 4445 4144 4641 4444 0000 0000 7b00 0000  DEADFADD....{...
00000030: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
00000040: 4445 4144 4641 4444 0000 0000 f600 0000  DEADFADD........
00000050: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
00000060: 4445 4144 4641 4444 0000 0000 7101 0000  DEADFADD....q...
00000070: 0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
```

Debugging glibc malloc

- (gdb) p mp_
  - $5 = {trim_threshold = 4202496, top_pad = 131072, mmap_threshold = 2101248, arena_test = 0, arena_max = 1, n_mmaps = 14, n_mmaps_max = 65536, max_n_mmaps = 16, no_dyn_threshold = 0, pagesize = 4096, mmapped_mem = 18333696, max_mmapped_mem = 22536192, max_total_mem = 0, sbrk_base = 0xd83000 ""}
- (gdb) p main_arena
  - $4 = {mutex = 0, flags = 3, fastbinsY = {...}, top = 0x7f650e165000, last_remainder = 0x7f65952d4740, bins = {...}, binmap = {...}, next = 0x368e58ee80, next_free = 0x368e58ee80, system_mem = 3022028800, max_system_mem = 3022028800}
- (gdb) p &main_arena
  - $2 = (struct malloc_state *) 0x368e58ee80
- (gdb) p main_arena.next
  - $3 = (struct malloc_state *) 0x368e58ff80
- (gdb) p *((struct malloc_state *) 0x368e58ff80)
  - $4 = (struct malloc_state *) 0x368e58ee80
- (gdb) p *(mchunkptr) 0x10c5c90
  - $5 = {prev_size = 0, size = 145, fd = 0x10c4030, bk = 0x312258fed8, fd_nextsize = 0x7fd3f0d5b000, bk_nextsize = 0x7fd3f0d5b4e8}

Configure Kernel Coredumps

- Install `kexec-tools`
- Add `crashkernel=256M` to the kernel cmdline – This amount of RAM is no longer available to your live kernel
  - grub2 example:
    - Edit /etc/default/grub
      - Add `crashkernel=256M` to GRUB_CMDLINE_LINUX
    - # grub2-mkconfig -o /boot/grub2/grub.cfg
    - Reboot and verify with `cat /proc/cmdline`
- To customize kdump, edit /etc/kdump.conf
  - For example, often useful to get user process data:
    - core_collector makedumpfile -l --message-level 1 -d 23,31
- Enable and start the kdump service
  - # systemctl enable kdump.service
  - # systemctl start kdump.service

How to Create a Kernel Coredump?

- Once the kdump service is running, a kernel panic will automatically produce a kernel coredump
- To manually produce a kernel coredump:
  - Enable sysrq (`man 5 proc`):
    - # echo 1 > /proc/sys/kernel/sysrq
  - Emulate a crash:
    - # echo c > /proc/sysrq-trigger
- kdump will dump the vmcore and reboot

Reading a Kernel Coredump

- Switch to the root user
- Kernel coredumps normally in /var/crash/
  - Check the version of the core:
  - # cd /var/carsh/${VMCORE_DIRECTORY}/
  - # strings vmcore | grep "Linux version"
    - Linux version 4.2.3-200.local.fc22.x86_64
- Install the kernel debuginfo/dbgsym packages matching the version of the vmcore

Reading a Kernel Coredump

- You may install the `crash` package, but best to compile from source:
  - https://github.com/crash-utility/crash/releases
  - $ tar xzf crash* && cd crash*
  - Recent vmcores may be compressed with lzop so best to compile in that support:
    - Install lzo, lzo-devel and lzo-minilzo packages
    - echo '-DLZO' > CFLAGS.extra
    - echo '-llzo2' > LDFLAGS.extra
  - $ make
  - # make install

Reading a Kernel Coredump

- Run crash on the matching vmlinux file and vmcore
  - crash ${PATH_TO_VMLINUX} ${PATH_TO_VMCORE}
  - Example:
    - $ crash /usr/lib/debug/lib/modules/4.2.3-200.local.fc22.x86_64/vmlinux /var/crash/*/vmcore
      CPUS: 4
      LOAD AVERAGE: 1.45, 0.72, 0.27
      TASKS: 444
      RELEASE: 4.2.3-200.local.fc22.x86_64
      PANIC: "sysrq: SysRq : Trigger a crash"
      PID: 12868
      COMMAND: "bash"
      CPU: 3
    - Last few lines are the current context

Crash Commands

- Type `help` for command list. `alias` to list aliases. `quit` to exit.
- Print the kernel log
  - crash> dmesg
    [90.266362] sysrq: SysRq : Trigger a crash
- Print processes
  - crash> ps
    ```
      PID    PPID  CPU       TASK        ST  %MEM     VSZ    RSS  COMM
    >     0      0       0   ffffffff81c124c0  RU   0.0         0         0  [swapper/0]
    ```
- Change current context to another PID:
  - crash> set 10042
    ```
        PID: 10042
    COMMAND: "gnome-terminal-"
       TASK: ffff8800482c3b00  [THREAD_INFO: ffff880044d24000]
        CPU: 3
      STATE: TASK_RUNNING
    ```
- Change context to the task executing on CPU #N (0-based), or the panic'ed task:
  - crash> set -c 0
  - crash> set -p

Crash Commands

- Print the stack trace of the current context:
  - crash> bt -l
    PID: 12868 TASK: ffff88007a0a0000 CPU: 3   COMMAND: "bash"
     #0 [ffff88004832f9f0] machine_kexec at ffffffff8105802b
        /usr/src/debug/kernel-4.2.fc22/linux-4.2.3-200.local.fc22.x86_64/arch/x86/kernel/machine_kexec_64.c: 322
     #1 [ffff88004832fa60] crash_kexec at ffffffff81127f42
        /usr/src/debug/kernel-4.2.fc22/linux-4.2.3-200.local.fc22.x86_64/kernel/kexec.c: 1500
     #2 [ffff88004832fb30] oops_end at ffffffff810180e6
        /usr/src/debug/kernel-4.2.fc22/linux-4.2.3-200.local.fc22.x86_64/arch/x86/kernel/dumpstack.c: 232 ...

Crash Commands

- Print virtual memory areas of the current context
  - crash> vm
    PID: 12868 TASK: ffff88007a0a0000 CPU: 3   COMMAND: "bash"
    MM PGD RSS TOTAL_VM
    ffff880044d5d800 ffff88007b15b000 4816k 118400k
    VMA START END FLAGS FILE
    ffff880060b3eda8 55c1a01eb000 55c1a02e3000 8000875
    /usr/bin/bash
- Print open files of the current context:
  - crash> files
    PID: 12868 TASK: ffff88007a0a0000 CPU: 3   COMMAND: "bash"
    ROOT: / CWD: /root
    FD FILE DENTRY INODE TYPE PATH
    0 ffff88005518ba00 ffff88005170a000 ffff88007c6a1f10 CHR /dev/pts/0

Crash Commands

- Print general memory information:
  - crash> kmem -i

	PAGES	TOTAL	PERCENTAGE
TOTAL MEM	479480	1.8 GB	----
FREE	218470	853.4 MB	45% of TOTAL MEM
USED	261010	1019.6 MB	54% of TOTAL MEM
BUFFERS	8096	31.6 MB	1% of TOTAL MEM
CACHED	93047	363.5 MB	19% of TOTAL MEM
TOTAL SWAP	64511	252 MB	----
SWAP USED	0	0	0% of TOTAL SWAP
SWAP FREE	64511	252 MB	100% of TOTAL SWAP
COMMIT LIMIT	304251	1.2 GB	----
COMMITTED	828252	3.2 GB	272% of TOTAL LIMIT

- Print kernel memory slab information:
  - crash> kmem -s

CACHE	NAME	OBJSIZE	ALLOCATED	TOTAL	SLABS	SSIZE
ffff88007d3c5e00	TCP	1984	30	32	2	32k

Crash Commands

- Print each CPU's run queue:
  - crash> runq
    CPU 0 RUNQUEUE: ffff88007fd967c0
      CURRENT: PID: 12868 TASK: ffff88007a0a0000 COMMAND: "bash"
      RT PRIO_ARRAY: ffff88007fd96960
        [no tasks queued]
      CFS RB_ROOT: ffff88007fd96860
        [120] PID: 224   TASK: ffff880036939d80 COMMAND: "kworker/3:2"
        [120] PID: 10042 TASK: ffff8800482c3b00 COMMAND: "gnome-terminal-"
- Print swap information:
  - crash> swap
    SWAP_INFO_STRUCT   TYPE   SIZE   USED   PCT PRI FILENAME
    ffff880036629400 PARTITION   258044k   0k   0%   -1 /dev/dm-0
- Display X bytes from a start address (in this example, 32 bytes):
  - crash> rd -8 0xffffffff814821f6 32
    ffffffff814821f6:   c6 04 25 00 00 00 00 01 5d c3 0f 1f 44 00 00 55   ..%.....]...D..U
    ffffffff81482206: 48 89 e5 53 8d 5f d0 48 c7 c7 60 48 a9 81 48 83   H..S._.H..`H..H.

Crash Commands

- Print stack contents for each frame:
  - crash> bt -f
    #11 [ffff880079d03de0] write_sysrq_trigger at ffffffff81482e98...
    #12 [ffff880079d03e00] proc_reg_write at ffffffff81286f62
    ffff880079d03e08: ffff8800420e3800 ffff880079d03f18
    ffff880079d03e18: ffff880079d03ea8 ffffffff8121d8d7
- Print definition of something like a stack frame method:
  - crash> whatis write_sysrq_trigger
    ssize_t write_sysrq_trigger(struct file *, const char *, size_t, loff_t *);
- In this case, the four arguments to write_sysrq_trigger will be the four addresses at the top of the stack of the lower frame (respectively, ffff8800420e3800, ffff880079d03f18, etc.)
- Since we know the first argument is a file, let's print its dentry struct and then from that its name:
  - crash> struct file.f_path.dentry ffff8800420e3800
    f_path.dentry = 0xffff880060a2d0c0
    crash> struct dentry.d_name.name 0xffff880060a2d0c0
    d_name.name = 0xffff880060a2d0f8 "sysrq-trigger"

Live Kernel Debugging

- If proper symbols are installed, simply run the `crash` command without arguments to debug the live kernel
- # crash

OOM Killer

- “By default [/proc/sys/vm/overcommit_memory=0], Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer” (`man 3 malloc`).
- Watch your system logs for messages such as:
  - kernel: Out of Memory: Killed process 123 (someprocess).
- Or set /proc/sys/vm/panic_on_oom=1 to cause a kernel panic instead
  - Then use the `bt` command to see who requested memory and how much and the `ps` command to see what is using memory

swappiness

- Linux aggressively uses physical memory for transient data such as file cache.
  - $ free -m

	total	used	free	shared	buffers	cached
Mem:	15699	4573	11126	0	86	1963
-/+ buffers/cache:		2523	13176			

- However, /proc/sys/vm/swappiness (default 60) controls how much the kernel will prefer to page programs out rather than filecache
- Set lower (e.g. 0) to avoid paging out programs

Memory Leaks

- "Currently debugging native-memory leaks on Linux with the freely available tools is more challenging than doing the same on Windows. Whereas UMDH allows native leaks on Windows to be debugged in situ, on Linux you will probably need to do some traditional debugging rather than rely on a tool to solve the problem for you." http://www.ibm.com/developerworks/library/j-nativememory-linux/
- ltrace might help, but no stacks:
  - $ ltrace -f -tt -p ${PID} -e malloc,free -o ltrace.out
- valgrind might work in a test environment, but not production
- mtrace overhead too high. SystemTap good option
- Find largest Rss VMAs in smaps and dump them in gdb

Summary

- Set `core` (-c) and `file` (-f) ulimits to unlimited for users or groups that run programs you're concerned about.
  - Either run `ulimit -c unlimited -f unlimited` in the shell or script that starts the process, or set it globally in /etc/security/limits.conf or /etc/security/limits.d/
  - Confirm the ulimits are set correctly by running `cat /proc/$PID/limits`
- If using systemd-coredump, ensure enough disk space is available or modify the configuration
- If using abrtd, increase MaxCrashReportsSize or set to unlimited
- Install debuginfo/dbgsym packages for kernel* packages and all the programs you're concerned about

Summary

- Monitor for coredumps
- Enable kdump and monitor for vmcores
- Don't be afraid to load cores and vmcores and review the stack traces
  - Otherwise, report the issues to the owner(s) of the code

Questions?

Appendix

Tips

- Review the size of thread stacks when investigating memory usage
- If using gcore, also gather /proc/$PID/smaps beforehand
- Creating coredumps is mostly disk I/O time, so if performance is important, allocate additional RAM so that coredumps are written to filecache and written out asynchronously
- If no memory leak, but RSS increases, may be fragmentation. Consider MALLOC_MMAP_THRESHOLD_/MALLOC_MMAP_MAX_ and/or MALLOC_ARENA_MAX=1