Operating Systems And Applications For Embedded Systems

Managing Memory

Katedra Inżynierii Komputerowej

Plan

- Profiling and Tracing
  - Profling with top
  - Poor man's profler
- perf
  - perf
  - Confguring the kernel for perf
  - Building perf with the Yocto Project
  - Building perf with Buildroot
  - Profling with perf
  - perf user interfaces
  - OProfle and gprof
  - LTTng components
  - LTTng and the Yocto Project
  - LTTng and Buildroot
  - Callgrind
  - Helgrind
  - Using strace to show system calls

Kernel space memory layout

- The kernel itself, in other words, the code and data loaded from the kernel image file at boot time. This is shown in the preceding code in the segments .text, .init, .data, and .bss. The .init segment is freed once the kernel has completed initialization.
- Memory allocated through the slab allocator, which is used for kernel data structures of various kinds. This includes allocations made using kmalloc(). They come from the region marked lowmem.
- Memory allocated via vmalloc(), usually for larger chunks of memory than is available through kmalloc(). These are in the vmalloc area.
- Mapping for device drivers to access registers and memory belonging to various bits of hardware, which you can see by reading /proc/iomem. These come from the vmalloc area but since they are mapped to physical memory that is outside of main system memory, they do not take any real memory.
- Kernel modules, which are loaded into the area marked modules.
- Other low level allocations that are not tracked anywhere else.

User space memory layout I

Listing 1: Listing

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#define BUFFER_SIZE (1024 * 1024)
void print_pgfaults(void)
{
    int ret;
    struct rusage usage;
    ret = getrusage(RUSAGE_SELF, &usage);
    if (ret == -1) {
        perror("getrusage");
    } else {
```

User space memory layout II

```c
            printf ("Major page faults %ld\n", usage.ru_majflt);
            printf ("Minor page faults %ld\n", usage.ru_minflt);
        }
    }
    int main (int argc, char *argv[])
    {
        unsigned char *p;
        printf("Initial state\n");
        print_pgfaults();
        p = malloc(BUFFER_SIZE);
        printf("After malloc\n");
        print_pgfaults();
        memset(p, 0x42, BUFFER_SIZE);
        printf("After memset\n");
        print_pgfaults();
```

User space memory layout III

```
    memset(p, 0x42, BUFFER_SIZE);
    printf("After 2nd memset\n");
    print_pgfaults();
    return 0;
}
```

Initial state
Major page faults 0
Minor page faults 172
After malloc
Major page faults 0
Minor page faults 186
After memset
Major page faults 0
Minor page faults 442
After 2nd memset

User space memory layout IV

Major page faults 0
Minor page faults 442

Process memory map

cat /proc/1/maps
00008000-0000e000 r-xp 00000000 00:0b 23281745 /sbin/init
00016000-00017000 rwxp 00006000 00:0b 23281745 /sbin/init
00017000-00038000 rwxp 00000000 00:00 0 [heap]
b6ded000-b6f1d000 r-xp 00000000 00:0b 23281695 /lib/libc-2.19.so
b6f1d000-b6f24000 —p 00130000 00:0b 23281695 /lib/libc-2.19.so

1. r = read
2. w = write
3. x = execute
4. s = shared
5. p = private (copy on write)

Swap

The idea of swapping is to reserve some storage where the kernel can place pages of memory that are not mapped to a file, so that it can free up the memory for other uses. It increases the effective size of physical memory by the size of the swap file. It is not a panacea: there is a cost to copying pages to and from a swap file which becomes apparent on a system that has too little real memory for the workload it is carrying and begins disk thrashing.

Swap to compressed memory (zram)

- CONFIG_SWAP
- CONFIG_CGROUP_MEM_RES_CTLR
- CONFIG_CGROUP_MEM_RES_CTLR_SWAP
- CONFIG_ZRAM

mtrace I

Listing 2: Listing

```
#include <mcheck.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int j;
    mtrace();
    for (j = 0; j < 2; j++)
        malloc(100); /* Never freed:a memory leak */
    calloc(16, 16); /* Never freed:a memory leak */
    exit(EXIT_SUCCESS);
}
```

mtrace II

```
export MALLOC_TRACE=mtrace.log
./mtrace-example
mtrace mtrace-example mtrace.log
Memory not freed:
-----------------
Address Size Caller
0x0000000001479460 0x64 at /home/chris/mtrace-example.c:11
0x00000000014794d0 0x64 at /home/chris/mtrace-example.c:11
0x0000000001479540 0x100 at /home/chris/mtrace-example.c:15
```

Valgrind I

1. memcheck: This is the default tool, and detects memory leaks and general misuse of memory
2. cachegrind: This calculates the processor cache hit rate
3. callgrind: This calculates the cost of each function call
4. helgrind: This highlights misuse of the Pthread API, potential deadlocks, and race conditions
5. DRD: This is another Pthread analysis tool
6. massif: This profiles usage of the heap and stack

Valgrind II

To find our memory leak, we need to use the default memcheck tool, with the option --leakcheck=full to print out the lines where the leak was found:

```
valgrind --leak-check=full ./mtrace-example

==17235== Memcheck, a memory error detector
==17235== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17235== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17235== Command: ./mtrace-example
==17235==
==17235==
==17235== HEAP SUMMARY:
==17235== in use at exit: 456 bytes in 3 blocks
==17235== total heap usage: 3 allocs, 0 frees, 456 bytes allocated
==17235==
```

Valgrind III

```
==17235== 200 bytes in 2 blocks are definitely lost in loss record 1 of 2
==17235== at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-linux.so)
==17235== by 0x4005FA: main (mtrace-example.c:12)
==17235==
==17235== 256 bytes in 1 blocks are definitely lost in loss record 2 of 2
==17235== at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-linux.so)
==17235== by 0x400613: main (mtrace-example.c:14)
==17235==
==17235== LEAK SUMMARY:
==17235== definitely lost: 456 bytes in 3 blocks
==17235== indirectly lost: 0 bytes in 0 blocks
==17235== possibly lost: 0 bytes in 0 blocks
==17235== still reachable: 0 bytes in 0 blocks
==17235== suppressed: 0 bytes in 0 blocks
==17235==
```

Valgrind IV

==17235== For counts of detected and suppressed errors, rerun with: -v
==17235== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

Additional reading

- Linux Kernel Development, 3rd Edition, by Robert Love, Addison Wesley, O'Reilly Media; (Jun. 2010) ISBN-10: 0672329468
- Linux System Programming, 2nd Edition, by Robert Love, O'Reilly Media; (8 Jun. 2013) ISBN-10: 1449339530
- Understanding the Linux VM Manager by Mel Gorman: https://www.kernel.org/doc/gorman/pdf/understand.pdf
- Valgrind 3.3 - Advanced Debugging and Profiling for Gnu/Linux Applications by J Seward, N. Nethercote, and J. Weidendorfer, Network Theory Ltd; (1 Mar. 2008) ISBN 978-0954612054

References

C. Simmonds.
Mastering Embedded Linux Programming.
Packt Publishing, 2015.

KATEDRA INŻYNIERII KOMPUTEROWEJ

The End