Embedded Systems Programming
Memory Management

Iwona Kochańska
Gdansk University of Technology
Physical memory - “flat”

- All processes share the same address space
- CPU examples: 8086-80206, ARM Cortex-M, 8- and 16-bit PIC, AVR, most of 8- and 16 bit systems
Physical memory - x86
Physical memory - x86

- Physical memory is divided into pages
- Page size is different for different architectures. 4096 B in most cases.
- Limited portability of C programs
- Total amount of memory should be known
- You need to make sure that processes have separate memory areas
- Badly written programs can cause a system failure
Virtual memory

- Linux configures the memory management unit of the CPU to present a virtual address space to a running program that begins at zero and ends at the highest address (0xffffffff on a 32-bit processor).
- Address space is divided into pages of 4 KiB
- Linux divides this virtual address space into **user space** (an area for applications), and **kernel space**.
- Mapping of virtual memory onto:
  - physical memory (RAM)
  - device memory
- Advantages:
  - each process may have its own address space;
  - the address spaces are separated from each other, and the address space of the kernel is invisible to the user’s address space,
  - the physical RAM can be mapped by many processes - shared memory
  - the memory areas may have different access settings (read, write, execute)
Kernel space and user space

- The split between the two is set by a kernel configuration parameter named `PAGE_OFFSET`.
  - In a typical 32-bit embedded system, `PAGE_OFFSET = 0xc0000000`, giving the lower three GiB to user space and the top one GiB to kernel space.
  - The user address space is allocated per process, so that each process runs in a sandbox, separated from the others.
  - The kernel address space is the same for all processes.
Kernel space and user space

- **Data segment** - global and static variables with defined, unchanging values; any variables not defined inside the function
- **BSS segment** - global and static variables with a value of 0 or without values
- **Heap** - dynamic variables - space managed by malloc, calloc, realloc, free; area shared by all processes, shared libraries, modules
- **Stack** - local variables; if the pointer to stack is equal to the pointer to heap, it means the end of the free memory.
Memory Management Unit

- **Memory Management Unit (MMU)** - a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses.

- **MMU tasks:**
  - translation of the virtual memory into the physical memory,
  - memory protection,
  - management of data buses
  - switching memory banks (in 8-bit systems)

- MMU divides a virtual memory space into pages

- A physical page of memory is identified by the **Page Frame Number (PFN)**. The PFN can be easily computed from the physical address by dividing it with the size of the page (or by shifting the physical address with PAGE_SHIFT bits to the right).
MMU - Translation Lookaside Buffer

- Translation Lookaside Buffer - associative cache memory used for mapping virtual / physical addresses
  - If the TLB does not have an appropriate assignment (size TLB memory is limited) slower hardware processor mechanisms are used, searching data structures found in memory, which sometimes requires help from the site software (operating system).
  - Positions in these structures are called page table entries (PTE), and the whole the structure is called the table of pages.
  - The complete address in the physical memory is determined by adding bits offset to the translated page number.
- If in TLB or PTE structures there is not a proper description of the currently used logical page of the memory or if there is an entry prohibiting access in a given mode, the MMU signals to CPU an exception connected with incorrect access to the memory page, so-called page fault.
Virtual memory

Each page of virtual memory may be:

- unmapped, in which access will result in a SIGSEGV
- mapped to a page of physical memory that is private to the process
- mapped to a page of physical memory that is shared with other processes
- mapped and shared with a copy on write flag set: a write is trapped in the kernel which makes a copy of the page and maps it to the process in place of the original page before allowing the write to take place
- mapped to a page of physical memory that is used by the kernel

The kernel may additionally map pages to reserved memory regions, for example, to access registers and buffer memory in device drivers.
Virtual memory

- Mapping and sharing with the flag "Copy on write"
  - when there is a need to share a large amount of data;
  - instead of real, expensive copying, the memory is returned is the indicator disappears to the original data;
  - copying is performed only when you need to modify them.

- Example: fork()
  - creates a child process that has an exact copy of the context of the parent process, as well as a copy of its memory (mapped to the same physical space).
  - memory that can be modified by both the process and its child, receive a "copy-on-write" tag.
  - when one of the processes modifies memory, the kernel intercepts this call and copies the modified pages so that changes made by one process are invisible to the other. From now on, the parent and child processes begin to refer to physically different pages.
Virtual memory

Julia Evans
@bork

every time you start a new process on Linux, it does a

fork() "clone" which copies the parent process

old new

fork() 3GB of RAM old new

copy on write

copying all the memory every time we fork would be slow and a waste of space.

the new process isn’t even gonna use that memory most of the time.

so Linux lets them share RAM instead of copying

Oh no! won’t the processes pollute each others’ memory?

how do we make this work?!

Linux marks all the memory for both processes as read-only (in the page table)

I’m going to write to the shared memory!

CPU Uh oh! That is not allowed! Linux! PAGE FAULT!

no problem! I will just make a copy of that piece of memory.

everyone is happy
Advantages of virtual memory:

- Invalid memory accesses are trapped and applications alerted by SIGSEGV
- Processes run in their own memory space, isolated from others
- Efficient use of memory through the sharing of common code and data, for example, in libraries
- The possibility of increasing the apparent amount of physical memory by adding swap files, although swapping on embedded targets is rare
Virtual memory

Disadvantages of virtual memory:

▶ It is difficult to determine the actual memory budget of an application

▶ The default allocation strategy is to over-commit, which leads to tricky out-of-memory situations,

▶ The delays introduced by the memory management code in handling exceptions – page faults – make the system less deterministic, which is important for real-time programs.
Kernel space memory layout

- Kernel memory is managed in a non-demand-paged way
  - For every allocation using `kmalloc()` or similar function, there is real physical memory.
  - Kernel memory is never discarded or paged out.

- Consumers of kernel-space memory include:
  - The kernel itself (the code and data loaded from the kernel image file at boot time)
    - Memory allocated through the slab allocator, which is used for kernel data structures of various kinds.
    - Segments: `.text`, `.init`, `.data`, `.bss`
  - Memory allocated through the slab allocator, which is used for kernel data structures of various kinds.
    - allocations made using `kmalloc()`.
    - They come from the region marked lowmem.
Consumers of kernel-space memory include:

- 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`.
  - 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.
How much memory does the kernel use?

- **Size of kernel image**

  iwona@iwona2:/opt/yoctoIG/debian/quark$ size bzImage
  BFD: bzImage: Warning: Ignoring section flag IMAGE_SCN_MEM_NOT_PAGED in section .bss
  text  data  bss  dec  hex  filename
  2348064  0  5810656  8158720  7c7e00  bzImage

  - Usually, the size is small when compared to the total amount of memory.
  - If that is not the case, you need to look through the kernel configuration and remove those components that you don’t need.
  - **Building small kernels:**
    - search for Linux-tiny or Linux Kernel Tinification.
    - Project https://tiny.wiki.kernel.org/.
### How much memory does the kernel use?

- **Reading /proc/meminfo**

```bash
iwona@iwona2:~$ cat /proc/meminfo

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemTotal</td>
<td>2018684 KB</td>
</tr>
<tr>
<td>MemFree</td>
<td>246924 KB</td>
</tr>
<tr>
<td>MemAvailable</td>
<td>546684 KB</td>
</tr>
<tr>
<td>Buffers</td>
<td>45952 KB</td>
</tr>
<tr>
<td>Cached</td>
<td>499364 KB</td>
</tr>
<tr>
<td>SwapCached</td>
<td>12752 KB</td>
</tr>
<tr>
<td>Active</td>
<td>1037772 KB</td>
</tr>
<tr>
<td>Inactive</td>
<td>521352 KB</td>
</tr>
<tr>
<td>Active(anon)</td>
<td>832792 KB</td>
</tr>
<tr>
<td>Inactive(anon)</td>
<td>376368 KB</td>
</tr>
<tr>
<td>Active(file)</td>
<td>204980 KB</td>
</tr>
<tr>
<td>Inactive(file)</td>
<td>144984 KB</td>
</tr>
<tr>
<td>Unevictable</td>
<td>32 KB</td>
</tr>
<tr>
<td>Mlocked</td>
<td>32 KB</td>
</tr>
<tr>
<td>SwapTotal</td>
<td>4882428 KB</td>
</tr>
<tr>
<td>SwapFree</td>
<td>4604100 KB</td>
</tr>
<tr>
<td>Dirty</td>
<td>112 KB</td>
</tr>
<tr>
<td>Writeback</td>
<td>0 KB</td>
</tr>
<tr>
<td>AnonPages</td>
<td>1010760 KB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>KernelStack</td>
<td>7776 KB</td>
</tr>
<tr>
<td>PageTables</td>
<td>32092 KB</td>
</tr>
<tr>
<td>NFS_Unstable</td>
<td>0 KB</td>
</tr>
<tr>
<td>Bounce</td>
<td>0 KB</td>
</tr>
<tr>
<td>WritebackTmp</td>
<td>0 KB</td>
</tr>
<tr>
<td>CommitLimit</td>
<td>5891768 KB</td>
</tr>
<tr>
<td>Committed_AS</td>
<td>5587976 KB</td>
</tr>
<tr>
<td>VmallocTotal</td>
<td>34359738367 KB</td>
</tr>
<tr>
<td>VmallocUsed</td>
<td>539424 KB</td>
</tr>
<tr>
<td>VmallocChunk</td>
<td>34359191516 KB</td>
</tr>
<tr>
<td>HardwareCorrupted</td>
<td>0 KB</td>
</tr>
<tr>
<td>Anon Huge Pages</td>
<td>0 KB</td>
</tr>
<tr>
<td>HugePages_Total</td>
<td>0 KB</td>
</tr>
<tr>
<td>HugePages_Free</td>
<td>0 KB</td>
</tr>
<tr>
<td>HugePages_Rsvd</td>
<td>0 KB</td>
</tr>
<tr>
<td>HugePages_Surp</td>
<td>0 KB</td>
</tr>
<tr>
<td>Hugepagesize</td>
<td>2048 KB</td>
</tr>
<tr>
<td>DirectMap4k</td>
<td>62448 KB</td>
</tr>
<tr>
<td>DirectMap2M</td>
<td>1024752 KB</td>
</tr>
</tbody>
</table>
```

How much memory does the kernel use?

The **kernel memory usage** is the sum of:

- **Slab**: The total memory allocated by the slab allocator
  - get more information by reading /proc/slabinfo.
- **KernelStack**: The stack space used when executing kernel code
- **PageTables**: The memory used for storing page tables
- **VmallocUsed**: The memory allocated by vmalloc()
  - more information in /proc/vmallocinfo
**How much memory does the kernel use?**

**lsmod** - to find out the memory space taken up by the code and data of modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Size</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>12927</td>
<td></td>
</tr>
<tr>
<td>ccm</td>
<td>17577</td>
<td></td>
</tr>
<tr>
<td>xt_conntrack</td>
<td>12681</td>
<td></td>
</tr>
<tr>
<td>ipt_MASQUERADE</td>
<td>12594</td>
<td></td>
</tr>
<tr>
<td>iptable_nat</td>
<td>12646</td>
<td></td>
</tr>
<tr>
<td>nf_conntrack_ipv4</td>
<td>18448</td>
<td></td>
</tr>
<tr>
<td>nf_defrag_ipv4</td>
<td>12483</td>
<td>nf_conntrack_ipv4</td>
</tr>
<tr>
<td>nf_nat_ipv4</td>
<td>12912</td>
<td>iptable_nat</td>
</tr>
<tr>
<td>xt_addrtype</td>
<td>12557</td>
<td></td>
</tr>
<tr>
<td>iptable_filter</td>
<td>12536</td>
<td></td>
</tr>
<tr>
<td>ip_tables</td>
<td>26011</td>
<td>iptable_filter,iptable_nat</td>
</tr>
</tbody>
</table>
User space memory layout

- Linux maps physical pages of memory only when the program accesses it.
- Example: allocating a buffer of 1 MiB using malloc(3) returns a pointer to a block of memory addresses but no actual physical memory! A flag is set in the page table entries such that any read or write access is trapped by the kernel. This is known as a page fault.
- Only at this point does the kernel attempt to find a page of physical memory and add it to the page table mapping for the process.
- A page fault is generated when the kernel traps an access to a page that has not been mapped.
- Two kinds of page fault:
  - minor - the kernel just has to find a page of physical memory and map it into the process address space
  - major - the virtual memory is mapped to a file (for example using mmap(2)). Reading from this memory means that the kernel not only has to find a page of memory and map it in, but it also has to...
User space memory layout

Example program

```c
#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 {
        printf ("Major page faults %l
", usage.ru_majflt);
        printf ("Minor page faults %l

   }

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();
    memset(p, 0x42, BUFFER_SIZE);
    printf("After 2nd memset\n");
    print_pgfaults();
    return
```
User space memory layout

Run example program

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
Major page faults 0
Minor page faults 442

- Increase of page faults when filling the memory with data: $442 - 186 = 256$.
  The buffer is 1 MiB (256 pages)

- The second call to memset(3) makes no difference because all the pages are now mapped.
Process memory map

Memory map for a process - through the proc filesystem.

▶ example: the map for the init process, PID 1:

```
iwona@iwokocha:~$ sudo cat /proc/1/maps
7f441c000000-7f441c029000 rw-p 00000000 00:00 0
7f441c029000-7f4420000000 ---p 00000000 00:00 0
7f4420000000-7f4424029000 rw-p 00000000 00:00 0
7f4424029000-7f4428000000 ---p 00000000 00:00 0
7f4428000000-7f442b6e0000 ---p 00000000 00:00 0
7f442b6e0000-7f442b865f0000 ---p 00000000 00:00 0
7f442b865f0000-7f442c9e700000 rw-p 00000000 00:00 0
7f442c9e700000-7f442f670000 rw-p 00000000 00:00 0
7f442f670000-7f442f674000 r-xp 00000000 00:02 54001692 /lib/x86_64-linux-gnu/libattr.so.1.1.0
7f442f674000-7f442f873000 ---p 00004000 00:02 54001692 /lib/x86_64-linux-gnu/libattr.so.1.1.0
7f442f873000-7f442f874000 r-p 00003000 00:02 54001692 /lib/x86_64-linux-gnu/libattr.so.1.1.0
7f442f874000-7f442f8870000 00:04000 00:02 54001692 /lib/x86_64-linux-gnu/libattr.so.1.1.0
7f442f887000-7f442f8870000 00:02 54002789 /lib/x86_64-linux-gnu/libdl-2.19.so
```

▶ First three columns: start and end virtual addresses and the permissions for each mapping.
▶ Permissions: r = read w = write x = execute s = shared p = private (copy on write)
Process memory map

- Mapping associated with a file
  - the filename appears in the final column,
  - columns four, five, and six contain:
    - the offset from the start of the file,
    - the block device number
    - the inode of the file.

- Most of the mappings are to the program itself and the libraries it is linked with.
Process memory map

- Two areas where the program can allocate memory:
  - marked [heap] - memory allocated using malloc(3) (except for very large allocations)
  - marked [stack] - allocations on the stack

- The maximum size of both areas is controlled by the process’s **ulimit**:
  - heap: ulimit -d, default unlimited
  - stack: ulimit -s, default 8 MiB

Allocations that exceed the limit are rejected by SIGSEGV.

- When running out of memory, the kernel may decide to discard pages that are mapped to a file and are read-only. If that page is accessed again, it will cause a major page fault and be read back in from the file.
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.

- 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 is seldom used on embedded devices because it does not work well with flash storage where constant writing would wear it out quickly.
Swap to compressed memory (zram)

- **zram driver** creates RAM-based block devices named /dev/zram0, /dev/zram1, and so on.
- Pages written to these devices are compressed before being stored.
- Compression ratios in the range of 30% to 50%
- Overall increase in free memory of about 10%, at the expense of more processing and a corresponding increase in power usage.
- Used in some low memory Android devices.
Swap to compressed memory (zram)

To enable zram, configure the kernel with these options:

```
CONFIG_SWAP
CONFIG_CGROUP_MEM_RES_CTRLR
CONFIG_CGROUP_MEM_RES_CTRLR_SWAP
CONFIG_ZRAM
```

Mount zram at boot time by adding to `/etc/fstab`:

```
/dev/zram0 none swap defaults zramsize=<size in bytes>,swapprio=<swap part>
```

Turn swap on and off:

```
# swapon /dev/zram0
# swapoff /dev/zram0
```
Mapping memory with mmap

- A process begins life with a certain amount of memory mapped to the text (the code) and data segments of the program file, together with the shared libraries that it is linked with. (OBRAZEK!)
- It can allocate memory
  - on its heap at runtime using `malloc(3)`
  - on the stack through locally scoped variables and memory allocated through `alloca(3)`
- It may also load libraries dynamically at runtime using `dlopen(3)`.  
- All of these mappings are taken care of by the kernel.
A process can also manipulate its memory map in an explicit way using `mmap(2)`:

```c
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- It maps **length** bytes of memory from the file with the descriptor **fd**, starting at **offset** in the file, and returns a pointer to the mapping, assuming it is successful.

- Since the underlying hardware works in pages, the length is rounded up to the nearest whole number of pages.

- The protection parameter, **prot**, is a combination of read, write, and execute permissions and the flags parameter contains at least `MAP_SHARED` or `MAP_PRIVATE`. 
Using mmap to allocate private memory

- Using mmap to allocate an area of private memory - by setting the MAP_ANONYMOUS flag and the fd file descriptor to -1.
- This is similar to allocating memory from the heap using malloc(3) except that the memory is page-aligned and in multiples of pages.
- The memory is allocated in the same area as that used for libraries (that area is referred to by some as the mmap area for this reason)
- Anonymous mappings are better for large allocations because they do not pin down the heap with chunks of memory, which would make fragmentation more likely.
- Interestingly, you will find that malloc(3) (in glibc at least) stops allocating memory from the heap for requests over 128 KiB and uses mmap in this way, so in most cases, just using malloc is the right thing to do.
- The system will choose the best way of satisfying the request.
Using mmap to share memory

POSIX shared memory requires mmap to access the memory segment. In this case, you set the MAP_SHARED flag and use the file descriptor from shm_open():

```c
int shm_fd;
char *shm_p;

shm_fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, 65536);
shm_p = mmap(NULL, 65536, PROT_READ | PROT_WRITE,
             MAP_SHARED, shm_fd, 0);
```
Using mmap to access device memory

- It is possible for a driver to allow its device node to be mmaped and so share some of the device memory with an application.
- The exact implementation is dependent on the driver.
- **Example 1:** the Linux frame buffer, `/dev/fb0`. The interface is defined in `/usr/include/linux/fb.h`, including an ioctl function to get the size of the display and the bits per pixel.
  - Use mmap to ask the video driver to share the frame buffer with the application and read and write pixels:

```c
int f;
int fb_size;
unsigned char *fb_mem;

f = open("/dev/fb0", 0_RDWR);
/* Use ioctl FIOGET_VSCREENINFO to find the display dimensions and calculate fb_size */
fb_mem = mmap(0, fb_size, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
/* read and write pixels through pointer fb_mem */
```
Using mmap to access device memory

Example 2: the streaming video interface, Video 4 Linux, version 2, or V4L2, which is defined in /usr/include/linux/videodev2.h.

- Each video device has a node named /dev/videoN, starting with /dev/video0.
- ioctl function to asks the driver to allocate a number of video buffers which you can mmap into user space.
- Then, it is just a question of cycling the buffers and filling or emptying them with video data, depending on whether you are playing back or capturing a video stream.
How much memory does my application use?

- Ask the kernel how much memory it thinks is available - `free` command

  ```bash
  iwona@iwokocha:$ free
  total    used    free    shared   buffers
  Mem: 8079276   6840000  1239276  213792   1726864
  +/- buffers/cache: 1849080  6230196
  Swap: 8271868 0 8271868
  ```

- Linux believes that free memory is wasted memory and so the kernel uses free memory for buffers and caches, in the knowledge that they can be shrunk when the need arises.

- Removing buffers and cache from the measurement gives the true free memory. When using free, the numbers on the second line marked +/- buffers/cache are the important ones.
How much memory does my application use?

- Force the kernel to free up caches by writing a number between 1 and 3 to /proc/sys/vm/drop_caches:
  ```bash
echo 3 > /proc/sys/vm/drop_caches
```
  - The number is actually a bit mask which determines which of the two broad types of cache you want to free:
    - 1 for the page cache
    - 2 for the dentry and inode caches combined.
Per-process memory usage

Two metrics to measure the amount of memory a process is using:

▶ **Vss (virtual set size)** - called **VSZ** in the **ps** command and **VIRT** in **top**
  ▶ total amount of memory mapped by a process
  ▶ sum of all the regions shown in /proc/<PID>/map
  ▶ number of limited interest, since only part of the virtual memory is committed to physical memory at any one time.

▶ **Rss (resident memory size)** - called **RSS** in **ps** and **RES** in **top**
  ▶ sum of memory that is mapped to physical pages of memory.
  ▶ this gets closer to the actual memory budget of the process, but there is a problem, if you add up the Rss of all the processes, you will get an overestimate the memory in use because some pages will be shared.
Using ps

- The versions of `top` and `ps` from BusyBox give very limited information.
- Package `procps` - full version of `top` and `ps`
- The `ps` command shows Vss (VSZ) and Rss (RSS)

```
iwona@iwokocha:~$ ps -eo pid,tid,class,rtprio,stat,vsz,rss,comm
   PID  TID  CLS RTPRIO STAT  VSZ   RSS COMMAND
     1    1   TS    - Ss   176232  5140 systemd
     2    2   TS    - S    0     0   kthreadd
```
Using **top**

- **top** shows a summary of the free memory and memory usage per process:

```
top - 13:40:30 up 3 days, 4:58, 3 users, load average: 0.13, 0.12, 0.11
Tasks: 187 total, 1 running, 184 sleeping, 0 stopped, 2 zzz
%Cpu(s): 1.1 us, 0.8 sy, 0.0 ni, 98.1 id, 0.0 wa, 0.0 hi, 0.0 id
KiB Mem: 8079276 total, 1034928 free, 1183852 used, 5860492 buffers
KiB Swap: 8271868 total, 8271868 free, 0 used, 6441624 available
```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>12336</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>253120</td>
<td>47204</td>
<td>32212 S</td>
<td>2.0</td>
<td>0.6</td>
<td>9:00</td>
</tr>
<tr>
<td>14918</td>
<td>iwona</td>
<td>20</td>
<td>0</td>
<td>570100</td>
<td>174336</td>
<td>75472 S</td>
<td>2.0</td>
<td>2.2</td>
<td>5:10</td>
</tr>
<tr>
<td>18642</td>
<td>iwona</td>
<td>20</td>
<td>0</td>
<td>431172</td>
<td>68804</td>
<td>41688 S</td>
<td>1.3</td>
<td>0.9</td>
<td>2:00</td>
</tr>
<tr>
<td>23176</td>
<td>iwona</td>
<td>20</td>
<td>0</td>
<td>41592</td>
<td>3604</td>
<td>2892 R</td>
<td>1.0</td>
<td>0.0</td>
<td>0:00</td>
</tr>
<tr>
<td>18480</td>
<td>iwona</td>
<td>20</td>
<td>0</td>
<td>1147948</td>
<td>220348</td>
<td>65880 S</td>
<td>0.7</td>
<td>2.7</td>
<td>0:03</td>
</tr>
<tr>
<td>14683</td>
<td>www-data</td>
<td>20</td>
<td>0</td>
<td>364816</td>
<td>4080</td>
<td>2496 S</td>
<td>0.3</td>
<td>0.1</td>
<td>0:00</td>
</tr>
</tbody>
</table>
```

Rss of a process keeps on increasing - > **memory leak**!
However, they are not very accurate in the absolute measurements of memory usage.
Using sMem

In 2009 proposed two new metrics

- **Uss (unique set size)** - amount of memory that is committed to physical memory and is unique to a process;
  - not shared with any other.
  - amount of memory that would be freed if the process were to terminate.

- **Pss (propotional set size)** - splits the accounting of shared pages that are committed to physical memory between all the processes that have them mapped.
  - if an area of library code is 12 pages long and is shared by six processes, each will accumulate two pages in Pss.
  - actual amount of memory being used by N processes.
Using smem

The information is available in `/proc/<PID>/smaps`, which contains additional information for each of the mappings shown in `/proc/<PID>/maps`.

```bash
iwona@iwokocha:$ sudo cat /proc/1/smaps | more
7f441c000000-7f441c029000 rw-p 00000000 00:00 0
Size: 164 kB
Rss: 12 kB
Pss: 12 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 12 kB
Referenced: 12 kB
Anonymous: 12 kB
AnonHugePages: 0 kB
Swap: 0 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB
Locked: 0 kB
VmFlags: rd wr mr mw me ne nr sd
```
Using smem

**smem** - a tool that collates the information from the smaps files and presents it in various ways, including as pie or bar charts.

- The project page: https://www.selenic.com/smem.
- It is available as a package in most desktop distributions.
- Written in Python, installing it on an embedded target requires a Python environment, which may be too much trouble for just one tool.

**smemcap** - small program that captures the state from /proc on the target and saves it to a TAR file which can be analyzed later on the host computer.

- part of BusyBox, but it can also be compiled from the **smem** source.
Using smem

iwona@iwokocha:~/Downloads/smem-1.4$ ./smem -t

<table>
<thead>
<tr>
<th>PID</th>
<th>User</th>
<th>Command</th>
<th>Swap</th>
<th>USS</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>14854</td>
<td>iwona</td>
<td>/bin/sh /etc/xdg/xfce4/xini</td>
<td>0</td>
<td>108</td>
<td>1</td>
</tr>
<tr>
<td>14887</td>
<td>iwona</td>
<td>/usr/bin/dbus-launch --exit</td>
<td>0</td>
<td>292</td>
<td>3</td>
</tr>
<tr>
<td>14906</td>
<td>iwona</td>
<td>/usr/bin/gpg-agent --sh --d</td>
<td>0</td>
<td>256</td>
<td>4</td>
</tr>
<tr>
<td>1116</td>
<td>iwona</td>
<td>/usr/bin/gpg-agent --sh --d</td>
<td>0</td>
<td>292</td>
<td>5</td>
</tr>
<tr>
<td>15189</td>
<td>iwona</td>
<td>/usr/bin/dbus-daemon --conf</td>
<td>0</td>
<td>412</td>
<td>5</td>
</tr>
<tr>
<td>18322</td>
<td>iwona</td>
<td>/usr/lib/dconf/dconf-service</td>
<td>0</td>
<td>576</td>
<td>6</td>
</tr>
<tr>
<td>14902</td>
<td>iwona</td>
<td>/usr/lib/x86_64-linux-gnu/x</td>
<td>0</td>
<td>592</td>
<td>6</td>
</tr>
<tr>
<td>15180</td>
<td>iwona</td>
<td>/usr/lib/at-spi2-core/at-spi</td>
<td>0</td>
<td>596</td>
<td>7</td>
</tr>
<tr>
<td>15214</td>
<td>iwona</td>
<td>/usr/lib/x86_64-linux-gnu/g</td>
<td>0</td>
<td>724</td>
<td>8</td>
</tr>
<tr>
<td>15193</td>
<td>iwona</td>
<td>/usr/lib/at-spi2-core/at-spi</td>
<td>0</td>
<td>696</td>
<td>9</td>
</tr>
</tbody>
</table>
Using smem

iwona@iwokocha:~/Downloads/smem-1.4$ ./smem --bar pid -c "pss uss"
Using smem

iwona@iwokocha:~/Downloads/smem-1.4$ ./smem --pie name -s rss

name by rss

unused 80.82%

other 6.98%

lyx 0.73%

dropbox 1.06%

opera 2.18%

skype 3.21%

icedove
Using smemcap

iwona@iwokocha:~/Downloads/smem-1.4$ sudo smemcap > smem-bb-cap.tar
iwona@iwokocha:~/Downloads/smem-1.4$ ./smem -t -S smem-bb-cap.tar