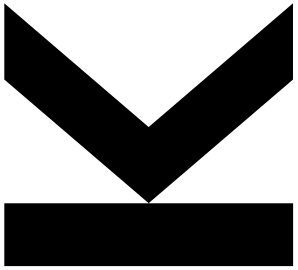
Memory allocation



Systems Programming

JOHANNES KEPLER
UNIVERSITÄT LINZ

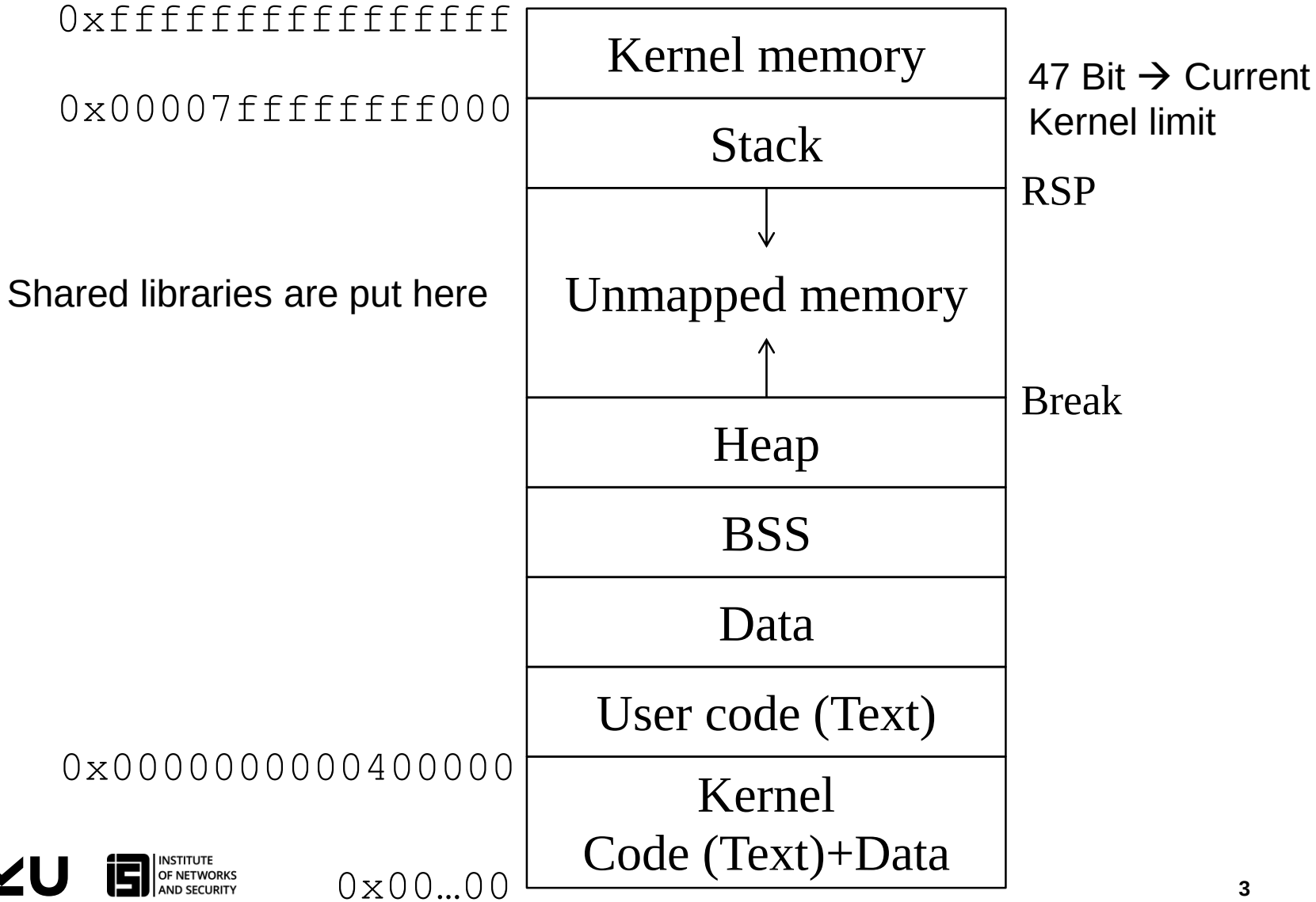
Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

Memory organization

- **Physical memory addresses**
  - Memory addresses referring to RAM chips (“real”)
  - Exist always in full size (as far as chips are present)
  - Only visible to OS – we will never see it in our programs!
- **Virtual memory addresses**
  - Memory addresses used by programs (“logical”, “virtual”)
    - What we use here!
  - Mapping to physical memory addresses
    - Support by OS together with CPU’s Memory Management Unit (MMU)
  - Mapping to disk space: swap space/partition
  - Flexibility for programmers
    - Do not care about how much memory is physically available
    - Do not care about how virtual addresses are mapped to physical ones
  - “Exists” only as far as actually used
    - No mapping to physical addresses if not reserved; access causes fault

Linux memory layout



Memory – factorial-main.s

- Start program in debugger and set breakpoint at start
- Find process ID through **`ps -A`**
- `cat /proc/25364/maps`

Code →
Data →

```
00400000-00401000 r-xp 00000000 00:25 514                /mnt/factorial-main
00600000-00601000 r-xp 00000000 00:25 514                /mnt/factorial-main
00601000-00602000 rwxp 00001000 00:25 514                /mnt/factorial-main
7ffff7bda000-7ffff7bdb000 r-xp 00000000 00:25 512        /mnt/libfactorial.so
7ffff7bdb000-7ffff7dda000 ---p 00001000 00:25 512        /mnt/libfactorial.so
7ffff7dda000-7ffff7ddb000 r-xp 00000000 00:25 512        /mnt/libfactorial.so
7ffff7ddb000-7ffff7ddc000 rwxp 00001000 00:25 512        /mnt/libfactorial.so
7ffff7ddc000-7ffff7dfc000 r-xp 00000000 fd:00 37071      /usr/lib64/ld-2.17.so
7ffff7ff6000-7ffff7ffa000 rwxp 00000000 00:00 0
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0          [vdso]
7ffff7ffc000-7ffff7ffe000 rwxp 00020000 fd:00 37071      /usr/lib64/ld-2.17.so
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

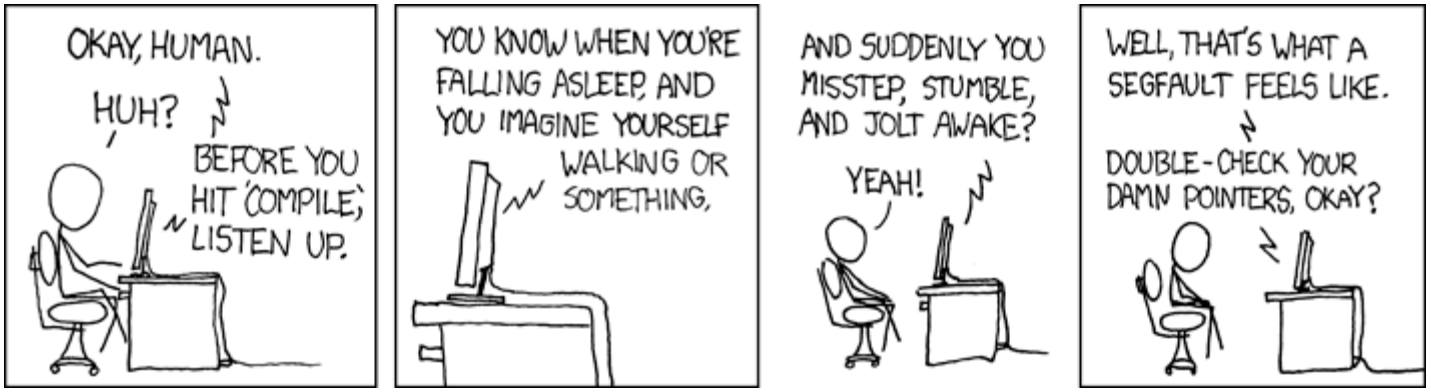
Linux memory layout – X86-64

- **Code**: Instructions of the program
  - Read-only
- **Data**: Data of the program (=initialized)
  - Read-Write
- **BSS**: Buffers of the program (=uninitialized; usually zeroed)
  - Read-Write
- **Heap**: Dynamically allocated memory
  - Read-Write
- **Stack**: Temporary data, procedures
  - Read-Write
- **Unmapped memory**: Memory not mapped to physical addresses
  - Access leads to segmentation fault
- **Break**: First non-usable (mapped) memory address

Dynamic memory allocation

- **Grow/Shrink Mapped Address Space**
  - **`brk`** system call
  - RAX contains **`12`** (system call number of **`brk`**)
  - RDI contains requested break
  - **`brk`** returns the new break in RAX or zero, if there is not enough physical memory or swap space
    - Actual new break might be larger than requested (Linux “might”=will round up to the nearest page, typ. 4 kB)
- **Problem**
  - Increment break for space of new object 1
  - Increment break for space of new object 2
  - What if object 1 is no longer needed?
    - Gap of mapped memory addresses that are not used anymore
- **Solution**
  - Memory manager keeping track of used memory; reuses gaps

Be careful where you point to...



XKCD, Compiler Complaint, https://xkcd.com/371/

Example
Trivial memory manager

alloc.s

```
#######GLOBAL VARIABLES########
# This points to the beginning of the memory we are managing
heap_begin:      .quad 0
# This points to one location past the memory we are managing
current_break:   .quad 0


######STRUCTURE INFORMATION####
    .equ HEADER_SIZE,16     # Size of space for memory region header
    .equ HDR_AVAIL_OFFSET,0 # Offset of the "available" flag in the header
    .equ HDR_SIZE_OFFSET,8  # Offset of the size field in the header


###########CONSTANTS###########
    .equ UNAVAILABLE,0  # This is the number we will use to mark
                        # space that has been given out
    .equ AVAILABLE,1    # This is the number we will use to mark
                        # space that has been returned, and is
                        # available for giving out
    .equ SYS_BRK,12     # System call number for the break
```

alloc.s

```
    ##allocate_init##
    #PURPOSE: Call this function to initialize the functions (specifically,
    #         this sets heap_begin and current_break).  This has no
    #         parameters and no return value.
    .globl allocate_init
    .type allocate_init,@function
allocate_init:
    pushq %rbp                # Standard function stuff
    movq  %rsp,%rbp
    # If the brk system call is called with 0 in %rdi, it
    # returns the last valid usable address
    movq  $SYS_BRK,%rax       # Find out where the break is
    movq  $0,%rdi
    syscall
    movq  %rax,current_break # Store the current break
    movq  %rax,heap_begin     # Store the current break as our
                              # first address. This will cause
                              # the allocate function to get
                              # more memory from Linux the
                              # first time it is run
    movq  %rbp,%rsp           # Exit the function
    popq  %rbp
    ret
```

alloc.s

```
    #PURPOSE:    This function is used to grab a section of memory. It
    # checks to see if there are any free blocks, and, if not, it asks Linux
    # for a new one.
    #PARAMETERS: One parameter - size of memory block to allocate
    #RETURN VALUE: This function returns the address of the allocated memory
    # in %rax.  If there is no memory available, it will return 0 in %rax
    #Variables used:
    #   %rdi - hold the size of the requested memory (first/only parameter)
    #   %rax - current memory region being examined
    #   %rdx - current break position
    #   %rcx - size of current memory region
    # We scan through each memory region starting with heap_begin. We look
    # at the size of each one, and if it has been allocated.  If it's big
    # enough for the requested size, and its available, it grabs that one.
    # If it does not find a region large enough, it asks Linux for more
    # memory.  In that case, it moves current_break up
    .globl allocate
    .type allocate,@function
allocate:
    pushq %rbp                  # Standard function stuff
    movq  %rsp,%rbp
    movq  heap_begin,%rax       # %rax will hold the current search location
    movq  current_break,%rdx    # %rdx will hold the current break
```

alloc.s

```
alloc_loop_begin:                # Here we iterate through each memory region
    cmpq   %rdx,%rax             # Need more memory if these are equal
    je     move_break
    movq   HDR_SIZE_OFFSET(%rax),%rcx  # Grab the size of this memory
    cmpq   $UNAVAILABLE,HDR_AVAIL_OFFSET(%rax)
    je     next_location         # If the space is unavailable, go to next one
    cmpq   %rdi,%rcx             # If the space is available, compare
    jle    allocate_here         # the size to the needed size. If its
                                 # big enough, go to allocate_here
next_location:
    addq   $HEADER_SIZE,%rax     # The total size of the memory region is the
    addq   %rcx,%rax             # sum of the size requested (currently stored
                                 # in %rcx), plus another 16 bytes for the
                                 # header (8 for the AVAILABLE/UNAVAILABLE flag,
                                 # and 8 for the size of the region). So, adding
                                 # %rcx and $16to %rax will get the address
                                 # of the next memory region
    jmp    alloc_loop_begin      # Go look at the next location
```

alloc.s

```
allocate_here:          # If we've made it here, that means that the
                        # region header of the region to allocate is in %rax
    # Mark space as unavailable
    movq   $UNAVAILABLE,HDR_AVAIL_OFFSET(%rax)
    addq   $HEADER_SIZE,%rax   # Move %rax past the header to the usable
                               # memory (since that's what we return)
    movq   %rbp,%rsp           # Return from the function
    popq   %rbp
    ret

move_break:   # If we've made it here, that means that we have exhausted
              # all addressable memory, and we need to ask for more.
              # %rdx(=%rax) holds the current endpoint of the data, and %rdi
              # holds its size
    # We need to increase %rdx to where we _want_ memory  to end, so we
    addq   $HEADER_SIZE,%rdx   # add space for the headers structure, and
    addq   %rdi,%rdx           # add space for the data size requested
    pushq  %rax                # Save needed registers
    pushq  %rcx
    pushq  %rdx
    pushq  %rdi
    movq   %rdx,%rdi           # Prepare parameter
    movq   $SYS_BRK,%rax       # Reset break (%rdi has requested break point)
```

alloc.s

```
    syscall  # Under normal conditions, this should return the new break in
             # %rax, which will be either 0 if it fails, or it will be equal
             # to or larger than we asked for.  We don't care  in this
             # program where it actually sets the break, so as long as %rax
             # isn't 0, we don't care what it is
    cmpq   $0,%rax              # Check for error conditions
    je     error
    popq   %rdi                 # Restore saved registers
    popq   %rdx
    popq   %rcx
    popq   %rax                 # Note: We throw away actual new break here!
    # Set this memory as unavailable, since we're about to give it away
    movq   $UNAVAILABLE,HDR_AVAIL_OFFSET(%rax)
    movq   %rdi,HDR_SIZE_OFFSET(%rax)      # Set the size of the memory
    # Move %rax to the actual start of usable memory.
    # %rax now holds the return value
    addq   $HEADER_SIZE,%rax
    movq   %rdx,current_break # Save the new break
    # Fall through to return from function
error:
    movq   %rbp,%rsp            # On error we return zero,
    popq   %rbp                 # which it already is
    ret
```

alloc.s

```
    ##deallocate##
    #PURPOSE: The purpose of this function is to give back a region of
    #         memory to the pool after we're done using it.
    #PARAMETERS: The only parameter is the address of the memory
    #         we want to return to the memory pool.
    #RETURN VALUE: There is no return value
    #PROCESSING: If you remember, we actually hand the program the
    #         start of the memory that they can use, which is 16 storage
    #         locations after the actual start of the memory region.  All we
    #         have to do is go back 16 locations and mark that memory as
    #         available, so that the allocate function knows it can use it.
    .globl deallocate
    .type deallocate,@function
    .equ ST_MEMORY_SEG,4        # Stack position of the memory region to free
deallocate:
    # Since the function is so simple, we
    # don't need any of the fancy function stuff
    # Get the pointer to the real beginning of the memory
    subq  $HEADER_SIZE,%rdi
    # Mark it as available
    movq  $AVAILABLE,HDR_AVAIL_OFFSET(%rdi)
    ret    # Return
```

alloc-demo.s

```
    .section .text

    .globl _start
_start:
    call allocate_init # Initialize memory manager
    movq $8,%rdi       # Alloc takes size as argument
    call allocate      # Allocate space for one quad (64 bits)
    cmpq $0,%rax       # Check success
    jne  use_memory
    movq $-1,%rdi      # Return -1 on error
    jmp  end
use_memory:
    movq $8,(%rax)     # Write immediate 8 into the allocated space
    movq (%rax),%rdi   # Read the value from memory and load it into %rdi
    movq %rax,%rdi     # Free all used memory
    call deallocate    # Everything reserved should (must) be freed again
end:
    movq $60,%rax      # Call the kernel's exit function
                       # Return value should be 8
    syscall
```

Notes on alloc.s

- How do we get the **current break**?
  - Call **`brk`** with **`0`** in RDI
- **Allocate**
  - 1. Start at beginning of heap
  - 2. Check if we are at the end of the heap
  - 3. If at end of heap → grab new memory and mark as unavailable
  - 4. If current region is unavailable → go to next; then to step 2
  - 5. If current region is too small → go to next; then to step 2
  - 6. If available memory is large enough → mark as unavailable + return it
- **Deallocate**
  - Just mark region as available
    - Note: We don't check whether the address passed in is really the start of a valid (and used) block – we just assume this!

Notes on alloc.s

- Every mapped memory region has an additional **header** of 16 bytes
  - Status: 8 bytes
    - Available → 1
    - Unavailable → 0
  - Size: 8 bytes
- Example: 56 Bytes in total, 40 bytes of memory have been reserved

Notes on alloc.s

- Problems of the simple allocator
  - Too **slow** if there are many allocations
    - Linear search for fitting region
    - Regions are possibly swapped to disk
      - Bringing them back to memory takes a lot of time
  - Number of system calls should be minimized
    - Expensive context switches
    - `brk` called, even if break might be much higher from the previous call
  - Memory wasted
    - Use a 1KB region for 4 bytes
      - 1020 - 4 - 16 (header) bytes wasted
      - Splitting needed
  - No verification whether a block freed is actually the start address of a valid block

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at