CSE 511: Operating Systems Design

Lecture 8

System Calls, LibC

Thread Models

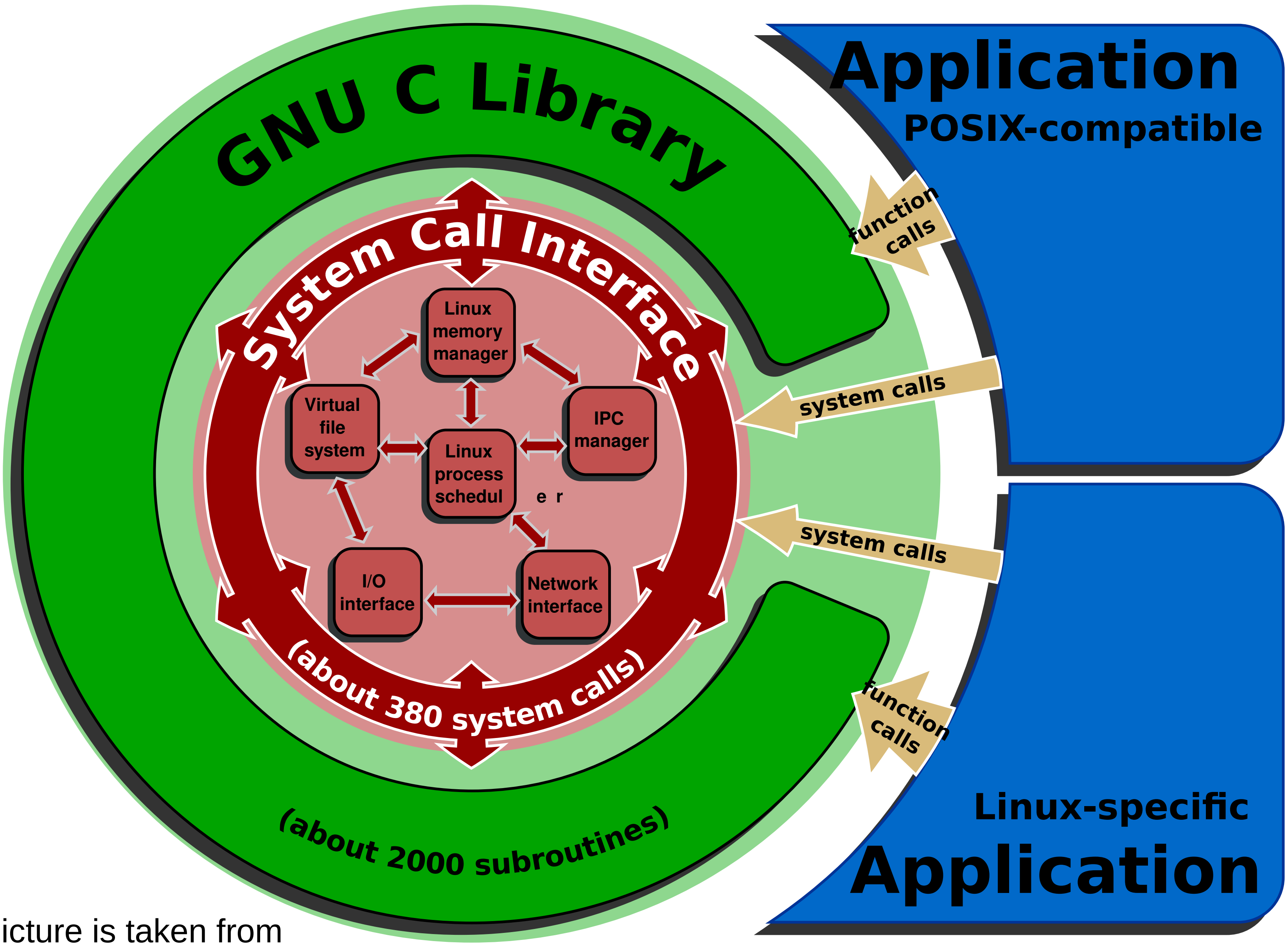
Thread Local Storage (TLS)

In-kernel per-CPU variables

System Calls

- Switching from the *privileged* to *unprivileged* mode can happen anywhere in the program code (e.g., ***sysret*** in x86-64)
- Switching from the *unprivileged* to *privileged* mode can only happen through special CPU-controlled *gates* (e.g., ***syscall*** in x86-64)
  - They are also known as '*system calls*'
  - Each system call has a special entry point (an address programmed in memory, CPU exception, etc)
  - The handler typically does not trust any input parameters and subject them to additional verification

System Calls



* The picture is taken from
https://en.wikipedia.org/wiki/Linux_kernel_interfaces

Why do we need LibC?

- The system call interface is platform-dependent
  - Each CPU architecture provides its own mechanisms
  - Do not want to use assembly code in our C programs, much easier to call functions
- The API is (mostly) platform-independent
  - POSIX in microkernels will use IPCs, not system calls
  - May want to replace some functions in LibC by using LD_PRELOAD
- LibC is more than a wrapper for system calls
  - Higher-level functions: printf(), malloc(), etc

Example: printf and malloc

```
#include <stdio.h>
#include <malloc.h>

int main()
{
    void *ptr;
    printf("Before allocating!\n");
    ptr = malloc(1024);
    printf("After allocating!\n");
    free(ptr);
    printf("After freeing\n");
    return 0;
}
```

Compile with -O0 (to avoid any optimization of malloc) and run:

```
strace ./test
```

Example: printf and malloc

```
execve("./test", ["./test"], 0x7ffcb310f800 /* 49 vars */) = 0
…
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
…
mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fa4f42a6000
…
close(3)                                = 0
…
brk(0x55db4415c000)                     = 0x55db4415c000
write(1, "Before allocating!\n", 19)    = 19
write(1, "After allocating!\n", 18)     = 18
write(1, "After freeing\n", 14)         = 14
```

Example: printf and malloc

```
execve("./test", ["./test"], 0x7ffcb310f800 /* 49 vars */) = 0
…
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
…
mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 
0x7fa4f42a6000
…
close(3)                                = 0
…
brk(0x55db4415c000)                     = 0x55db4415c000
write(1, "Before allocating!\n", 19)    = 19
write(1, "After allocating!\n", 18)     = 18
write(1, "After freeing\n", 14)         = 14
```

- Where is malloc?
  - Size is too small, try to increase it to 1024*1024 (1MB)

Example: printf and malloc

```
write(1, "Before allocating!\n", 19)    = 19
mmap(NULL, 1052672, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0x7fd295daf000
write(1, "After allocating!\n", 18)     = 18
munmap(0x7fd295daf000, 1052672)         = 0
write(1, "After freeing\n", 14)         = 14
```

Example: printf and malloc

```
write(1, "Before allocating!\n", 19)    = 19
mmap(NULL, 1052672, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0x7fd295daf000
write(1, "After allocating!\n", 18)     = 18
munmap(0x7fd295daf000, 1052672)         = 0
write(1, "After freeing\n", 14)         = 14
```

- What is mmap?
  - Maps files to the virtual address space (e.g., LibC itself)
  - But does not necessarily have to be backed by actual files and can be used to allocate memory (MAP_ANONYMOUS)

GLibC’s mmap()

sysdeps/unix/sysv/linux/mmap.c:

```
void *
__mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)
{
  MMAP_CHECK_PAGE_UNIT ();

  if (offset & MMAP_OFF_LOW_MASK)
    return (void *) INLINE_SYSCALL_ERROR_RETURN_VALUE (EINVAL);

#ifdef __NR_mmap2
  return (void *) MMAP_CALL (mmap2, addr, len, prot, flags, fd,
                offset / (uint32_t) MMAP2_PAGE_UNIT);
#else
  return (void *) MMAP_CALL (mmap, addr, len, prot, flags, fd,
                MMAP_ADJUST_OFFSET (offset));
#endif
}
weak_alias (__mmap, mmap)
```

GLibC’s mmap()

sysdeps/unix/sysv/linux/mmap_internal.h:

```
#ifndef MMAP_CALL
# define MMAP_CALL(__nr, __addr, __len, __prot, __flags, __fd, __offset) \
  INLINE_SYSCALL_CALL (__nr, __addr, __len, __prot, __flags, __fd, __offset)
#endif
```

sysdeps/unix/sysdep.h:

```
#define INLINE_SYSCALL_CALL(...) \
  __INLINE_SYSCALL_DISP (__INLINE_SYSCALL, __VA_ARGS__)

#define __INLINE_SYSCALL_DISP(b,...) \
  __SYSCALL_CONCAT (b,__INLINE_SYSCALL_NARGS(__VA_ARGS__))
(__VA_ARGS__)

#define __INLINE_SYSCALL_NARGS(...) \
  __INLINE_SYSCALL_NARGS_X (__VA_ARGS__,7,6,5,4,3,2,1,0,)

#define __INLINE_SYSCALL_NARGS_X(a,b,c,d,e,f,g,h,n,...) n
```

GLibC’s mmap()

sysdeps/unix/sysdep.h:

```
#define __INLINE_SYSCALL0(name) \
  INLINE_SYSCALL (name, 0)
#define __INLINE_SYSCALL1(name, a1) \
  INLINE_SYSCALL (name, 1, a1)
...
#define __INLINE_SYSCALL7(name, a1, a2, a3, a4, a5, a6, a7) \
  INLINE_SYSCALL (name, 7, a1, a2, a3, a4, a5, a6, a7)
```

sysdeps/unix/sysv/linux/sysdep.h:

```
#define INLINE_SYSCALL(name, nr, args...)                \
  ({                                    \
    long int sc_ret = INTERNAL_SYSCALL (name, nr, args);        \
    __glibc_unlikely (INTERNAL_SYSCALL_ERROR_P (sc_ret))        \
    ? SYSCALL_ERROR_LABEL (INTERNAL_SYSCALL_ERRNO (sc_ret))      \
    : sc_ret;                                \
  })
```

GLibC’s mmap()

sysdeps/unix/sysv/linux/x86_64/sysdep.h:

```
#define INTERNAL_SYSCALL(name, nr, args...)          \
    internal_syscall##nr (SYS_ify (name), args)

#define internal_syscall0(number, dummy...)          \
({                                                   \
    unsigned long int resultvar;                     \
    asm volatile (                                   \
    "syscall\n\t"                                    \
    : "=a" (resultvar)                               \
    : "0" (number)                                   \
    : "memory", REGISTERS_CLOBBERED_BY_SYSCALL);     \
    (long int) resultvar;                            \
})
…
#define SYS_ify(syscall_name)   __NR_##syscall_name
```

sysdeps/unix/sysv/linux/x86_64/64/arch-syscall.h:

```
#define __NR_mmap 9
…
#define __NR_write 1
```

Linux's side of mmap()

arch/x86/kernel/cpu/common.c:

```
wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
```

arch/x86/include/asm/msr-index.h:

```
#define MSR_LSTAR         0xc0000082 /* long mode SYSCALL target */
```

arch/x86/entry/entry_64.S:

```
 Registers on entry:
 rax  system call number
 rcx  return address
 r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 rdi  arg0
 rsi  arg1
…
SYM_CODE_START(entry_SYSCALL_64)
…
    /* IRQs are off. */
    movq    %rax, %rdi
    movq    %rsp, %rsi
    call    do_syscall_64       /* returns with IRQs disabled */
```

Linux's side of mmap()

arch/x86/entry/common.c:

```
__visible noinstr void do_syscall_64(unsigned long nr, struct pt_regs *regs)
{
    nr = syscall_enter_from_user_mode(regs, nr);

    instrumentation_begin();
    if (likely(nr < NR_syscalls)) {
        nr = array_index_nospec(nr, NR_syscalls);
        regs->ax = sys_call_table[nr](regs);
...
}
```

arch/x86/entry/syscall_64.c:

```
#define __SYSCALL_COMMON(nr, sym) __SYSCALL_64(nr, sym)
#define __SYSCALL_64(nr, sym) [nr] = __x64_##sym,

asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &__x64_sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

Linux's side of mmap()

arch/x86/include/generated/asm/syscalls_64.h:

```
__SYSCALL_COMMON(1, sys_write)
...
__SYSCALL_COMMON(9, sys_mmap)
```

arch/x86/include/asm/syscall_wrapper.h:

```
#define __X64_SYS_STUBx(x, name, ...)                    \
    __SYS_STUBx(x64, sys##name,                 \
        SC_X86_64_REGS_TO_ARGS(x, __VA_ARGS__))

#define __SYS_STUBx(abi, name, ...)                \
    long __##abi##_##name(const struct pt_regs *regs);      \
    ALLOW_ERROR_INJECTION(__##abi##_##name, ERRNO);         \
    long __##abi##_##name(const struct pt_regs *regs)       \
    {                               \
        return __se_##name(__VA_ARGS__);            \
    }
```

Linux's side of mmap()

arch/x86/kernel/sys_x86_64.c:

```
SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,
        unsigned long, prot, unsigned long, flags,
        unsigned long, fd, unsigned long, off)
{ … }
```

arch/x86/include/asm/syscall_wrapper.h:

```
#define __SYSCALL_DEFINEx(x, name, ...)                 \
    static long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__)); \
    static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__));\
    __X64_SYS_STUBx(x, name, __VA_ARGS__)               \
    __IA32_SYS_STUBx(x, name, __VA_ARGS__)              \
    static long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__))  \
    {                               \
        long ret = __do_sys##name(__MAP(x,__SC_CAST,__VA_ARGS__));\
        __MAP(x,__SC_TEST,__VA_ARGS__);                 \
        __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));   \
        return ret;                         \
    }                               \
    static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

Linux’s side of mmap()

arch/x86/kernel/sys_x86_64.c:

```
SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,
        unsigned long, prot, unsigned long, flags,
        unsigned long, fd, unsigned long, off)
{ … }
```

This is __do_sys_mmap

Threads

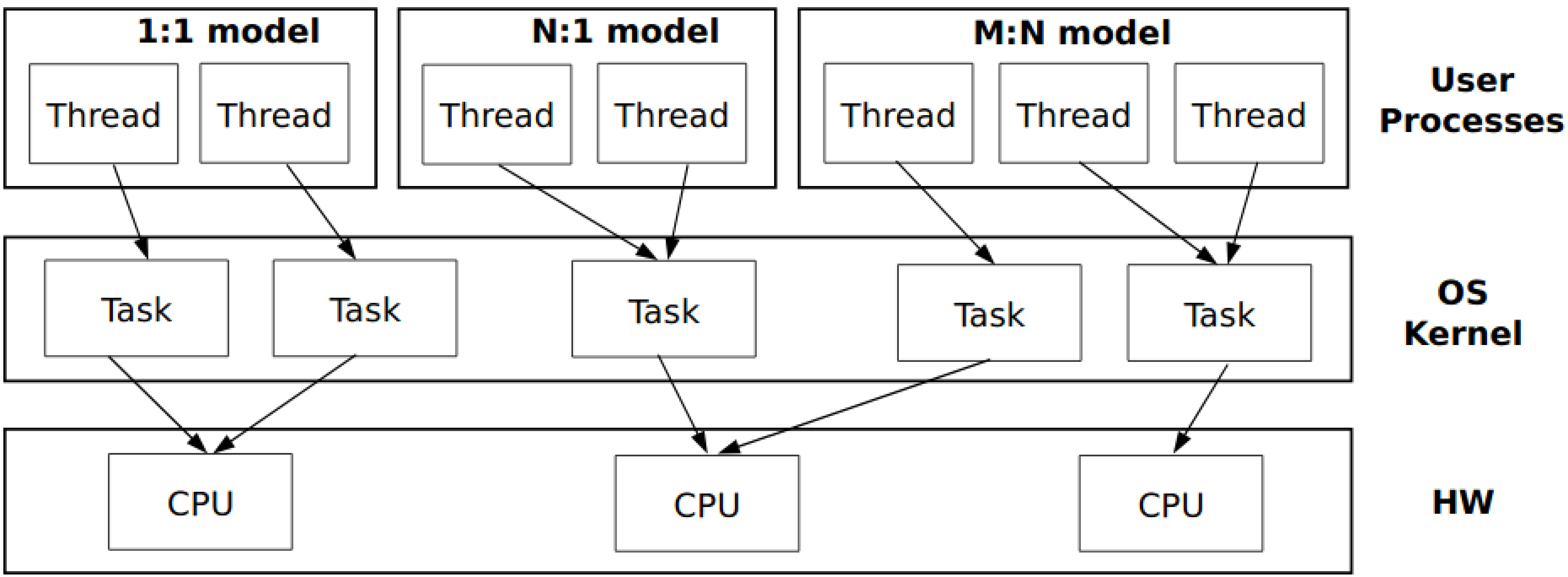
- A system call executes in the context of a user-space thread
- Threads are managed by POSIX threads (libpthread)
  - A similar idea to GlibC but only with respect to thread-related system calls
- The Linux kernel uses different abstractions
  - It defines different abstractions known as “tasks”

Threads

- Each process consists of 1 or more threads
  - ***Threads*** are entities that share a process's virtual memory address space, file descriptor table, and other resources
  - Threads can be executed concurrently but are **not** isolated from other threads in their respective processes
- The kernel defines one or more "tasks"
  - ***Tasks*** comprise the execution state of entities that can be executed concurrently and possibly on different CPUs
    - Can be used within the same or different processes
  - The OS scheduler executes tasks by suspending some tasks and resuming others from time to time
    - Task context switch

Threads

- Linux systems use the 1:1 threading model
  - Each thread is assigned its own task
  - Other models are N:1 and M:N



Threads

- N:1 – all threads are mapped to one task
  - Needs only minimal kernel support but it only allows the use of a single CPU for all threads
- M:N – M threads are mapped to N tasks
  - Reconciles the N:1 and 1:1 models
  - Normally M ≥ N
    - Needs a user-mode scheduler (e.g., in libpthread) to suspend some threads and resume others
  - Capable to context switch between different threads in a process without using the underlying OS kernel
    - Difficult to implement, e.g., blocking system calls can monopolize the underlying “task”

Problems with system calls

- Can pollute TLB
  - User-space virtual addresses are different from kernel-space addresses
- Can pollute cache
  - Kernel data and code are different
- Direct cost of system calls
  - You need to switch CPU between different modes
  - Additional checks, different stacks in kernel space and user space, etc

Thread Local Storage (TLS)

- All threads in one process share the ***same*** address space
- But how do we maintain per-thread variables?
  - e.g., ***errno*** in C must contain an error number when an operation/system call fails, operations can execute concurrently
- Thread Local Storage (TLS) is a special per-thread area which provides private memory address space for each thread
  - Should be accessed efficiently, i.e., no page table switches
  - Not supposed to be ***fully*** isolated from other threads if they somehow figure out the base address of this area, i.e., they can still corrupt this area since there is no isolation

Thread Local Storage (TLS)

- TLS is architecture- and OS-specific, see "ELF Handling for Thread-Local Storage" for x86-64/Linux and other architectures (https://akkadia.org/drepper/tls.pdf)
- To support TLS, you typically need a special register
  - It specifies a thread-specific offset (base) in the memory
- x86 historically supported "segmentation", where memory pointers would be bound to a specific segment
  - Segments can be more trivial, i.e., change the base address
  - x86-64 made segmentation obsolete but still supports base addresses for fs: and gs: segment registers

Thread Local Storage (TLS)

- Many x86 instructions support segment override, by default ds: (data segment), es: (another data segment), or ss: (stack segment) are used depending on the instruction
  - Historically, it was supported for 16-bit and 32-bit CPUs
    - Segmentation was almost never used with 32-bit CPUs
  - ds:, es:, and ss: will now have their base=0 for x86-64
  - However, fs: and gs: can still change their base using WRMSR (FS.base is C000_0100h, GS.base is C000_0101h)
- fs: and gs: segments are still used
  - fs: is used by TLS in Linux
  - gs: is used by TLS in Windows

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a = 5;

int func() {
    return a;
}
```

Compile:

gcc -Wall -O2 -S tls.c

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movl    %fs:a@tpoff, %eax
    ret
    .cfi_endproc

…

    .section    .tdata,"awT",@progbits
    .align 4
    .type   a, @object
    .size   a, 4
a:
    .long   5
```

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a = 5;

int func() {
    return a;
}
```

Compile:

gcc -Wall -O2 -S tls.c

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movl    %fs:a@tpoff, %eax
    ret
    .cfi_endproc

…

    .section    .tdata,"awT",@progbits
    .align 4
    .type   a, @object
    .size   a, 4
a:
    .long   5
```

Segment override, use **fs:** instead of **ds:**

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a = 5;

int func() {
    return a;
}
```

Compile:

gcc -Wall -O2 -S tls.c

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movl    %fs:a@tpoff, %eax
    ret
    .cfi_endproc

…

    .section    .tdata,"awT",@progbits
    .align 4
    .type   a, @object
    .size   a, 4
a:
    .long   5
```

a is a symbol
@tpoff – a relocation with an offset relative to the TLS block

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a = 5;

int func() {
    return a;
}
```

Compile:

gcc -Wall -O2 -S tls.c

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movl    %fs:a@tpoff, %eax
    ret
    .cfi_endproc

…

    .section    .tdata,"awT",@progbits
    .align 4
    .type   a, @object
    .size   a, 4
a:
    .long   5
```

.tdata – a TLS .data section, works as an initialization image

Thread Local Storage (TLS)

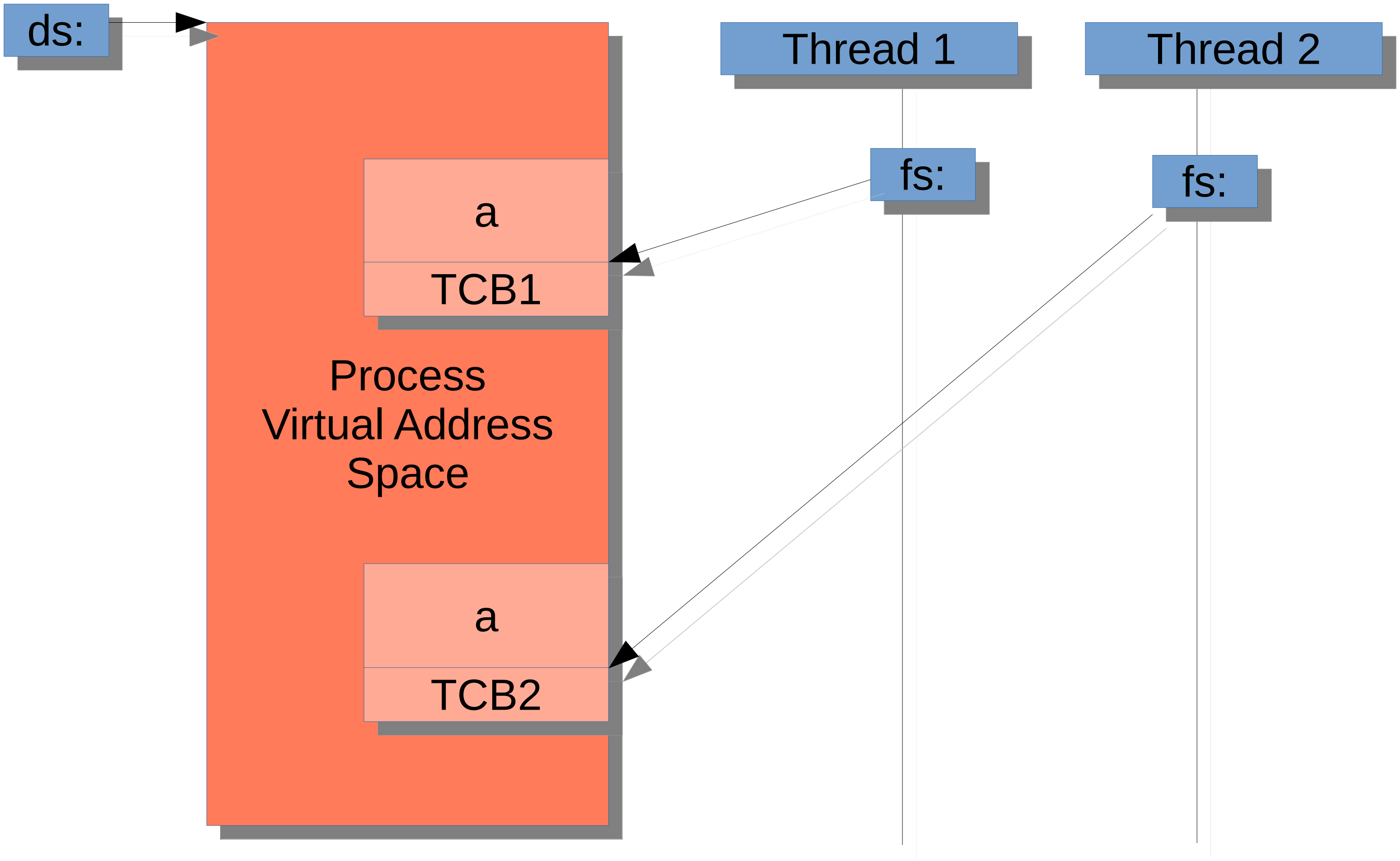
- What about errno? We can find the following trick in /usr/include/errno.h:

```
#include <features.h>

/* The system-specific definitions of the E* constants, as macros.  */
#include <bits/errno.h>

/* When included from assembly language, this header only provides the
   E* constants.  */
#ifndef __ASSEMBLER__

__BEGIN_DECLS

/* The error code set by various library functions.  */
extern int *__errno_location (void) __THROW __attribute_const__;
# define errno (*__errno_location ())

# ifdef __USE_GNU

/* The full and simple forms of the name with which the program was
   invoked.  These variables are set up automatically at startup based on
   the value of argv[0].  */
extern char *program_invocation_name;
extern char *program_invocation_short_name;
```

Thread Local Storage (TLS)

- Each address is calculated using a segment register
  - [ds:] + addr = addr, since [ds:]=0
  - [fs:] + addr = addr + TLS_block, since [fs:] points to a TLS block
- sym@tpoff is addr in case of TLS
  - Non-negative addresses are used ***internally*** for a thread control block (TCB)
    - fs:0 points to TLS_block itself
  - Negative addresses used for TLS variables
    - @tpoff is actually negative, -4 in the example above when linking is complete

What is really happening here?



Thread Local Storage (TLS)

Create tls.c:

```c
__thread int a[100];

int func() {
    int sum = 0;
    for (int i = 0; i < 100; i++)
        sum += a[i];
    return sum;
}
```

Compile:

```
gcc -Wall -O2 -S tls.c
```

- Linux segment overrides are used sparingly, fs:0 is sometimes preferred

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rdx
    xorl    %r8d, %r8d
    leaq    a@tpoff(%rdx), %rax
    addq    $400+a@tpoff, %rdx
    .p2align 4,,10
    .p2align 3
.L2:
    addl    (%rax), %r8d
    addq    $4, %rax
    cmpq    %rdx, %rax
    jne .L2
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a[100];

int func() {
    int sum = 0;
    for (int i = 0; i < 100; i++)
        sum += a[i];
    return sum;
}
```

Compile:

gcc -Wall -O2 -S tls.c

- Linux segment overrides are used sparingly, fs:0 is sometimes preferred

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rdx
    xorl    %r8d, %r8d
    leaq    a@tpoff(%rdx), %rax
    addq    $400+a@tpoff, %rdx
    .p2align 4,,10
    .p2align 3
.L2:
    addl    (%rax), %r8d
    addq    $4, %rax
    cmpq    %rdx, %rax
    jne .L2
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

%rdx – TLS_block

Thread Local Storage (TLS)

Create tls.c:

```c
__thread int a[100];

int func() {
    int sum = 0;
    for (int i = 0; i < 100; i++)
        sum += a[i];
    return sum;
}
```

Compile:

gcc -Wall -O2 -S tls.c

- Linux segment overrides are used sparingly, fs:0 is sometimes preferred

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rdx
    xorl    %r8d, %r8d
    leaq    a@tpoff(%rdx), %rax
    addq    $400+a@tpoff, %rdx
    .p2align 4,,10
    .p2align 3
.L2:
    addl    (%rax), %r8d
    addq    $4, %rax
    cmpq    %rdx, %rax
    jne .L2
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

%rax – the beginning of the array

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a[100];

int func() {
    int sum = 0;
    for (int i = 0; i < 100; i++)
        sum += a[i];
    return sum;
}
```

Compile:

```
gcc -Wall -O2 -S tls.c
```

- Linux segment overrides are used sparingly, fs:0 is sometimes preferred

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rdx
    xorl    %r8d, %r8d
    leaq    a@tpoff(%rdx), %rax
    addq    $400+a@tpoff, %rdx
    .p2align 4,,10
    .p2align 3
.L2:
    addl    (%rax), %r8d
    addq    $4, %rax
    cmpq    %rdx, %rax
    jne .L2
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

%rdx – the end of the array

Thread Local Storage (TLS)

Create tls.c:

```
__thread int a[100];

int func() {
    int sum = 0;
    for (int i = 0; i < 100; i++)
        sum += a[i];
    return sum;
}
```

Compile:

gcc -Wall -O2 -S tls.c

- Linux segment overrides are used sparingly, fs:0 is sometimes preferred

tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rdx
    xorl    %r8d, %r8d
    leaq    a@tpoff(%rdx), %rax
    addq    $400+a@tpoff, %rdx
    .p2align 4,,10
    .p2align 3
.L2:
    addl    (%rax), %r8d
    addq    $4, %rax
    cmpq    %rdx, %rax
    jne .L2
    movl    %r8d, %eax
    ret
    .cfi_endproc
```

the actual loop

Other usages of [fs:0]

Create tls.c:

```
__thread int a = 5;

int func() {
    return a;
}
```

Compile:

gcc -Wall -O2 -mno-tls-direct-seg-refs -S tls.c

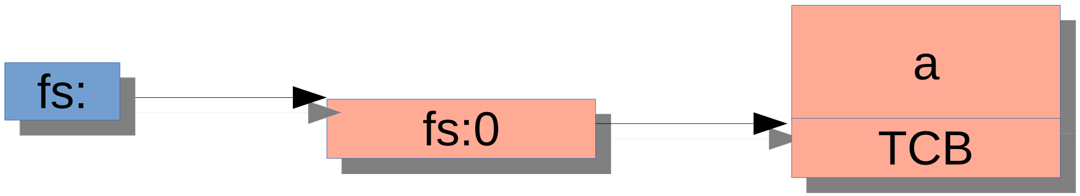
tls.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movq    %fs:0, %rax
    movl    a@tpoff(%rax), %eax
    ret
    .cfi_endproc
```

- It is even possible to disable direct access through segment registers, and **always** use fs:0
- Why?
  - Until recently fs: and gs: could only be updated in privileged mode through WRMSR

Other usages of [fs:0]

- But what if you need to context switch in unprivileged mode?
  - Not in Linux since the 1:1 threading model is used
  - Consider M:N threading with a user-space scheduler
- Also useful in other places
  - Microkernels, hypervisors that use paravirtualization
- ***Example***: an OS can use indirection by creating a “dummy” TCB, which points to a real TCB
  - Still works in a regular case (Linux), fs:0 points to itself



In-kernel usage

- We have a similar (but not identical) problem in the kernel
  - We want to have per-CPU variables
  - There is a similar set of variables but each CPU will have its own value (e.g., interrupt stack context)
- Why not use the other register (gs:) for that purpose?
  - Linux implements exactly that in percpu.h

In-kernel usage

arch/x86/include/asm/percpu.h:

```
#ifdef CONFIG_X86_64
#define __percpu_seg        gs
…
#ifdef CONFIG_SMP
#define PER_CPU_VAR(var)    %__percpu_seg:var
…
#ifdef CONFIG_SMP
#define __percpu_prefix     "%%"__stringify(__percpu_seg)":"
#define __my_cpu_offset     this_cpu_read(this_cpu_off)
...
```

Variable accesses become gs:addr

In-kernel usage

ruslan@ruslan-ThinkPad-T470p: ~/linux-5.9.12/arch/x86

```
{
    BUG_ON(slot >= KVM_NR_SHARED_MSRS);
    shared_msrs_global.msrs[slot] = msr;
    if (slot >= shared_msrs_global.nr)
        shared_msrs_global.nr = slot + 1;
}
EXPORT_SYMBOL_GPL(kvm_define_shared_msr);

static void kvm_shared_msr_cpu_online(void)
{
    unsigned int cpu = smp_processor_id();
    struct kvm_shared_msrs *smsr = per_cpu_ptr(shared_msrs, cpu);
    u64 value;
    int i;

    for (i = 0; i < shared_msrs_global.nr; ++i) {
        rdmsrl_safe(shared_msrs_global.msrs[i], &value);
        smsr->values[i].host = value;
        smsr->values[i].curr = value;
    }
}

int kvm_set_shared_msr(unsigned slot, u64 value, u64 mask)
                                                      332,33-36     2%
```

arch/x86/kvm/x86.c

In-kernel usage

- Rumprun-SMP unikernel also uses gs: to access per-CPU variables
  - https://github.com/ssrg-vt/rumprun-smp
  - e.g., values related to a timer and per-CPU clock cycle counter

platform/hw/include/arch/x86_64/pcpu.h

ruslan@ruslan-ThinkPad-T470p: ~/test/rumprun-smp

```
#ifndef _BMK_PCPU_PCPU_H_
#define _BMK_PCPU_PCPU_H_

#define BMK_PCPU_PAGE_SHIFT 12UL
#define BMK_PCPU_PAGE_SIZE (1<<BMK_PCPU_PAGE_SHIFT)
#define BMK_PCPU_L1_SHIFT 7
#define BMK_PCPU_L1_SIZE 128

struct bmk_cpu_info {
    __attribute__ ((aligned(BMK_PCPU_L1_SIZE))) struct bmk_cpu_info *self;
    struct bmk_thread *idle_thread;
    unsigned long cpu;
    /* Interrupt enabling/disabling. */
    unsigned long spldepth;
    /* Base time values at the last call to tscclock_monotonic(). */
    unsigned long long time_base;
    unsigned long long tsc_base;
    __attribute__ ((aligned(BMK_PCPU_L1_SIZE))) char _pad[0];
};

#define bmk_get_cpu(x)                                              \
    __extension__ ({                                                \
        __typeof__ (((struct bmk_cpu_info *)0)->x) __val;           \
                                                  1,1          Top
```

In-kernel usage

```
        __res;                                                                    \
    })

#define bmk_get_cpu_info()  bmk_get_cpu(self)

static inline void bmk_set_cpu_info(struct bmk_cpu_info *cpu)
{
    unsigned long p = (unsigned long) cpu;

    cpu->self = cpu;
    __asm__ __volatile__ ("wrmsr" ::
        "c" (0xc0000101),
        "a" ((unsigned)(p)),
        "d" ((unsigned)(p >> 32))
    );
}

static inline void bmk_cpu_relax(void)
{
    __asm__ __volatile__ ("pause" ::: "memory");
}

#endif /* _BMK_PCPU_PCPU_H_ */
```

platform/hw/include/arch/x86_64/pcpu.h