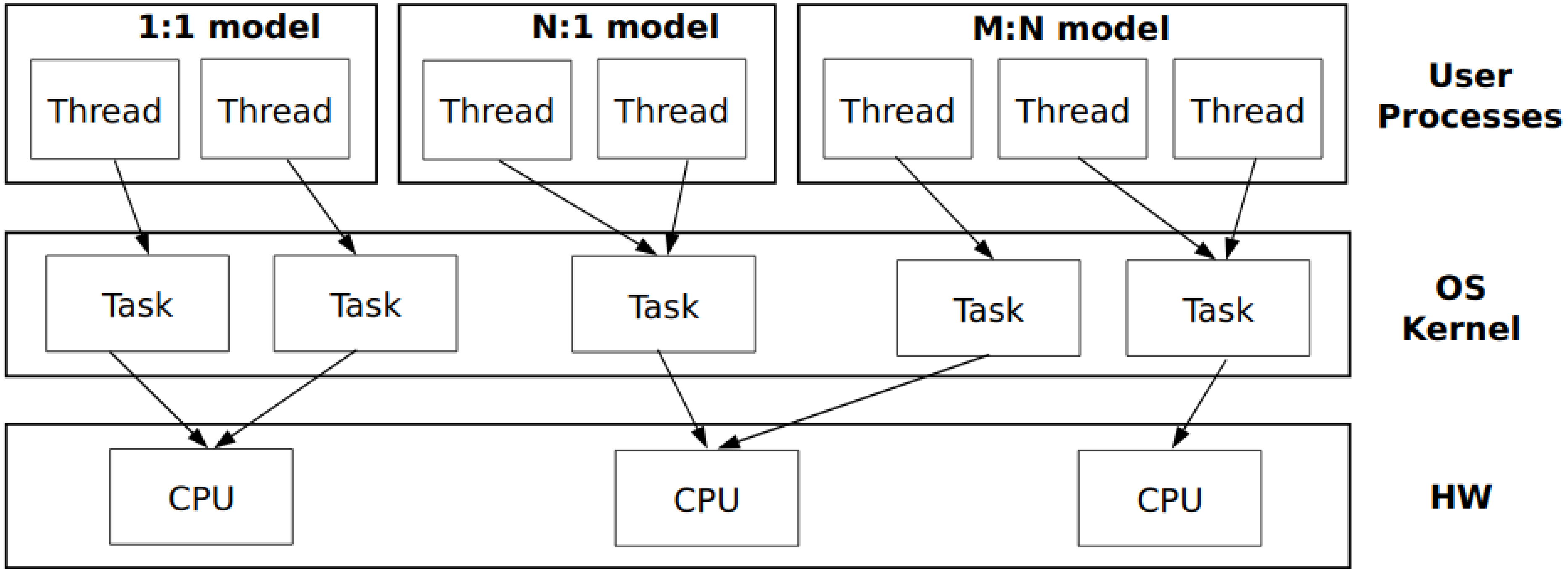
CSE 511: Operating Systems Design

Lectures 9-10

Thread Local Storage (TLS)
In-kernel per-CPU variables
Global Descriptor Table (GDT)
Task State Segment (TSS)
Interrupt Descriptor Table (IDT)

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
```

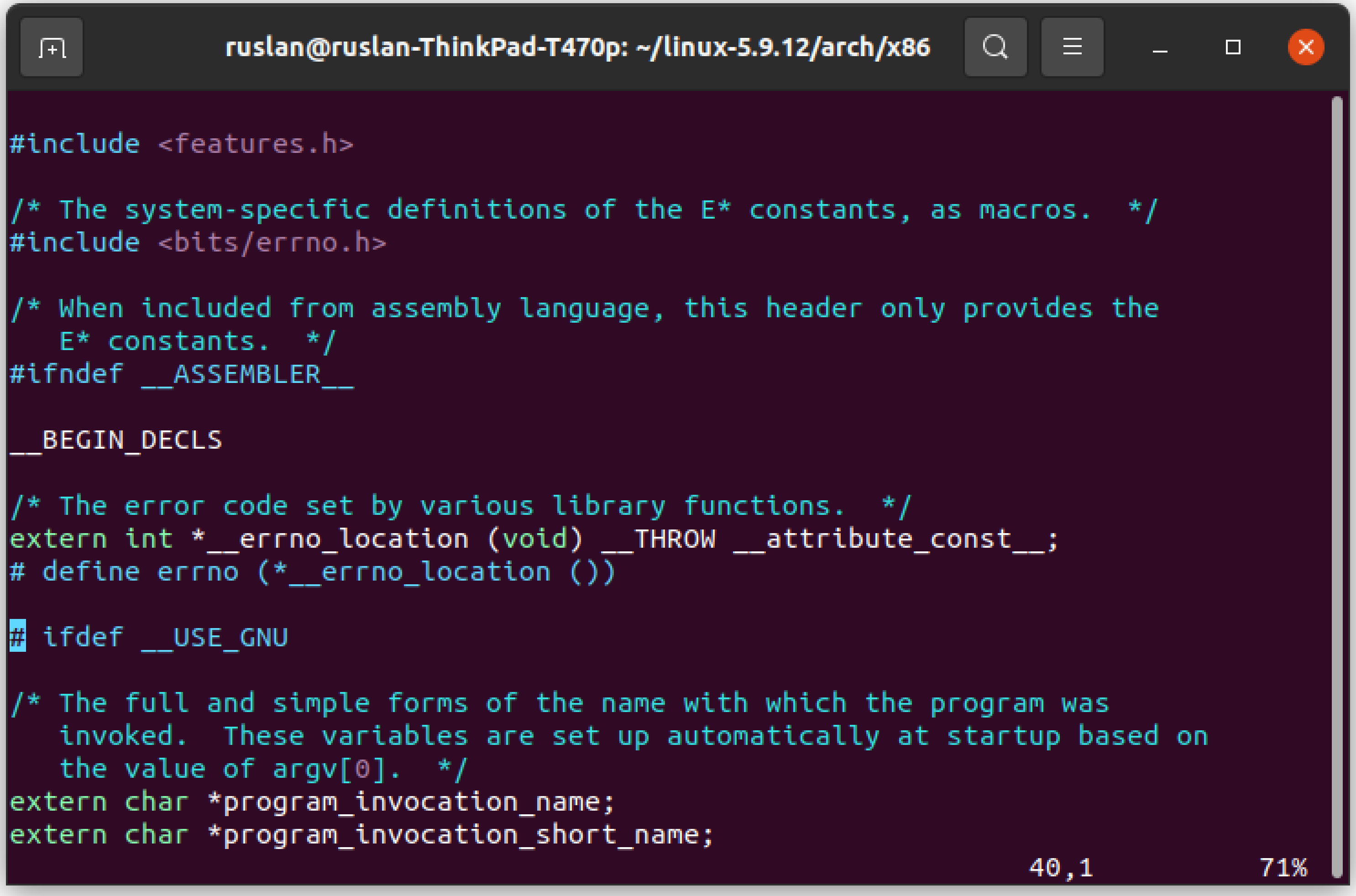
tls.s:

```
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:



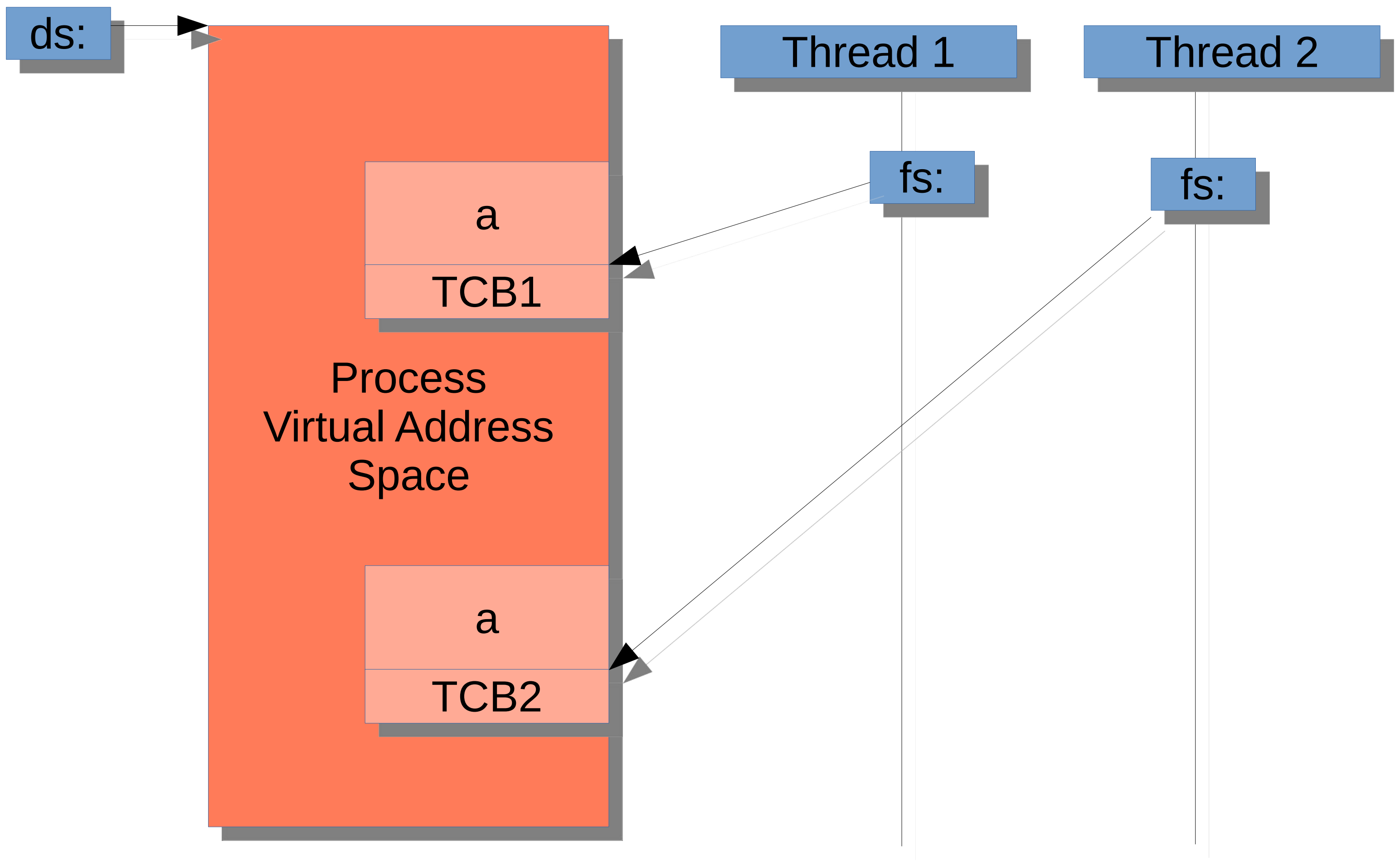
ruslan@ruslan-ThinkPad-T470p: ~/linux-5.9.12/arch/x86

```
#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;
                                                             40,1          71%
```

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:

```
__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:

```
__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
```

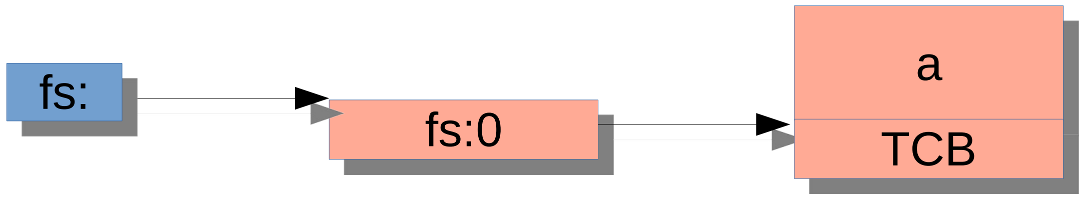
tls.s:

```
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

Global Descriptor Table (GDT)

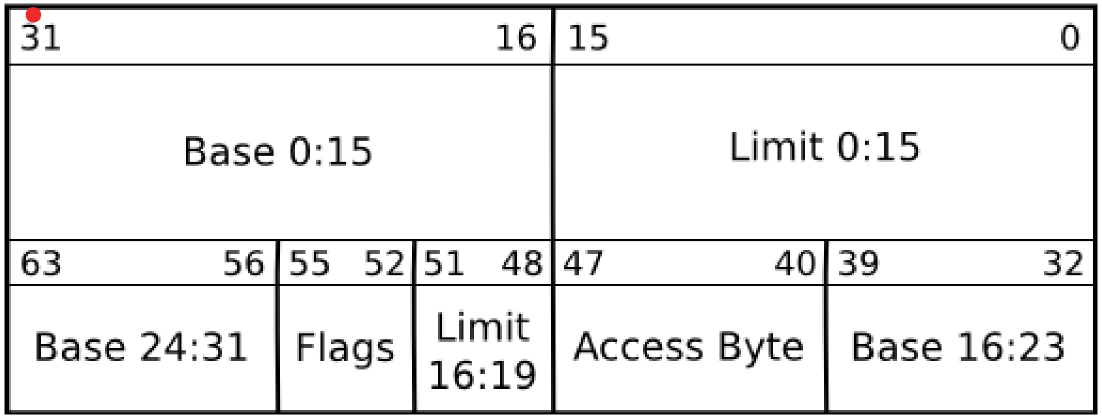
- Historically, x86 provided a very elaborated virtual memory system
  - Paging (e.g., Windows, Linux, and most other OSes)
  - Segmentation
  - Or even both (e.g., OS/2)
- Even though segmentation is now obsolete in x86-64, certain pieces have to be initialized correctly
  - Thread Local Storage (TLS) uses fs: or gs: segment registers
  - All other segment registers have to be initialized appropriately (cs, ds, es, ss)

Global Descriptor Table (GDT)

- A segment is an ***isolated*** contiguous piece of memory space, which has a ***base*** address and ***limit*** (size – 1)
  - You can restrict access to segments (e.g., each process can access just its own segments)
  - Compare that with a per-process page table (and TLB), though no complex translation is needed
    - An address will be segment_base + offset, offset is your program’s (segment’s) virtual address
- A segment register refers to a segment ***selector***
  - Cannot be arbitrarily replaced without proper permissions
- Except TLS, you want to select segments with base = 0 and limit = MAX_ADDRESS for x86-64 (“flat memory”)

Global Descriptor Table (GDT)

- GDT is a table of segment ***descriptors***
  - Each entry is 8 bytes
  - The first entry is reserved (0)
  - Other entries can define segments for code and data
  - For x86-64, base and limit are always 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF, respectively

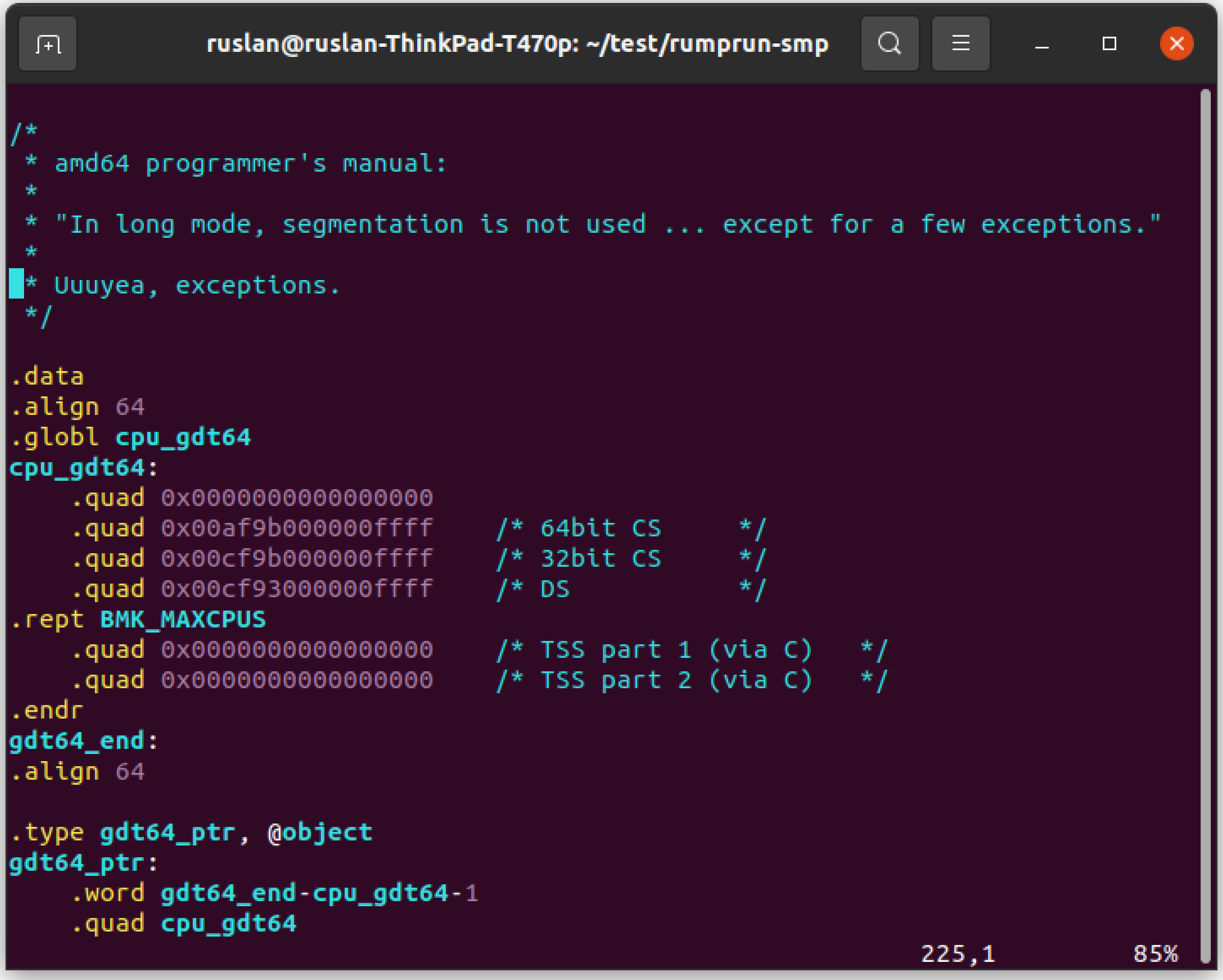


* The picture is taken from https://wiki.osdev.org/Global_Descriptor_Table

Rumprun-SMP

- A unikernel based on NetBSD
- Do not copy any code but you can check it as an example to get some idea
- A version that supports SMP is available at
  - https://github.com/ssrg-vt/rumprun-smp

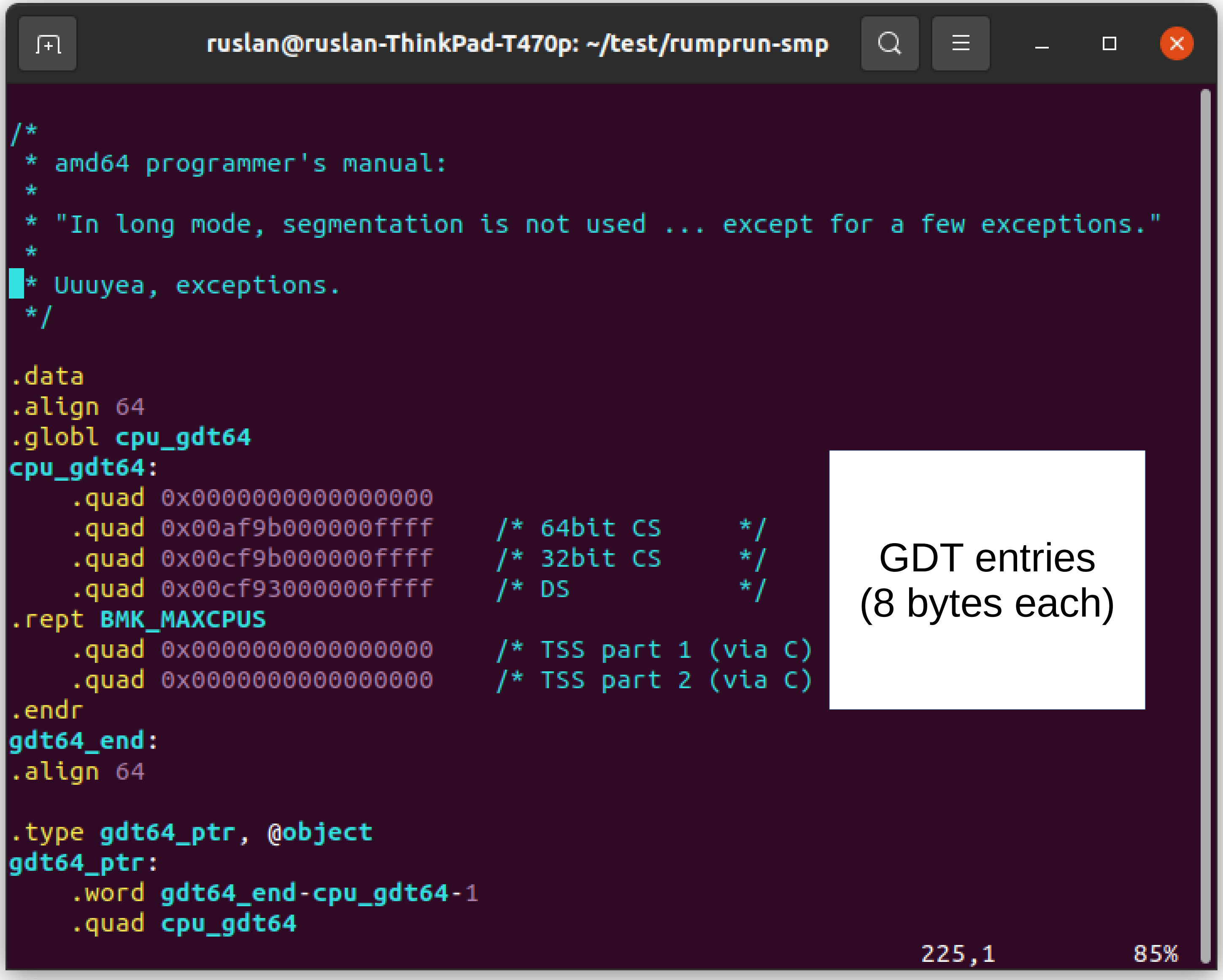
Example of GDT (rumprun-smp)



```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)   */
    .quad 0x0000000000000000    /* TSS part 2 (via C)   */
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
                                                           225,1          85%
```

platform/hw/arch/amd64/locore.S

Example of GDT (rumprun-smp)

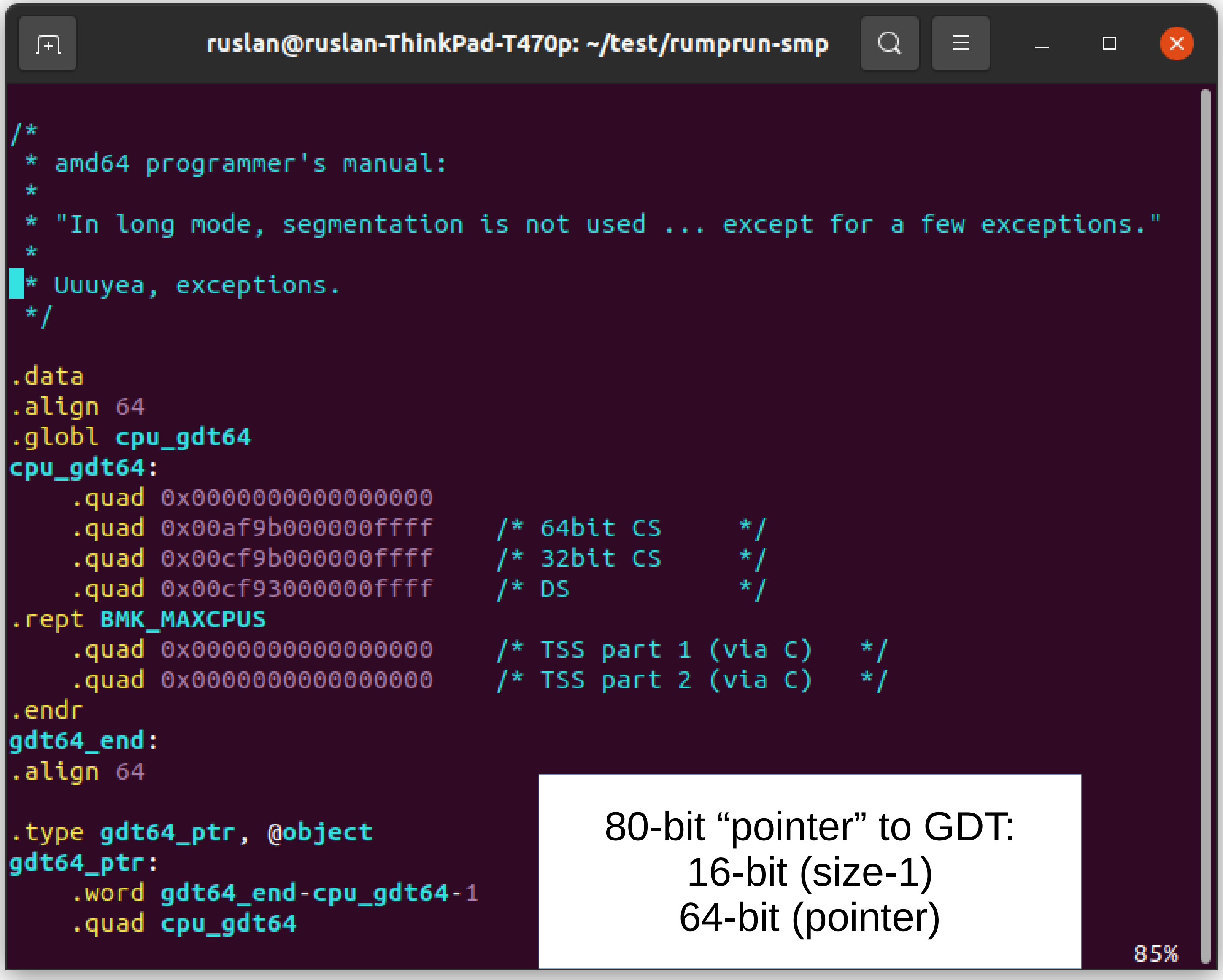


```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)
    .quad 0x0000000000000000    /* TSS part 2 (via C)
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
                                                           225,1          85%
```

GDT entries
(8 bytes each)

platform/hw/arch/amd64/locore.S

Example of GDT (rumprun-smp)


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

```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)   */
    .quad 0x0000000000000000    /* TSS part 2 (via C)   */
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
```

85%

80-bit “pointer” to GDT:
16-bit (size-1)
64-bit (pointer)

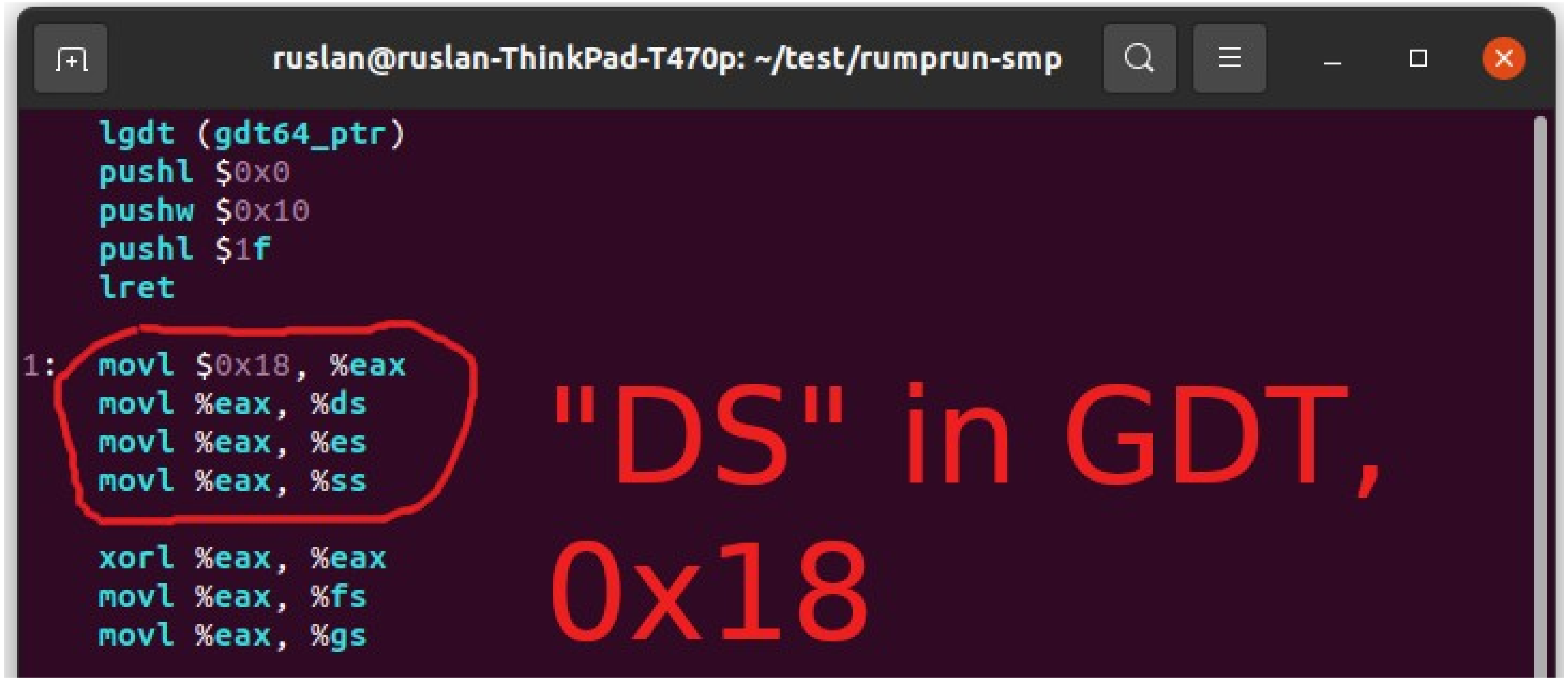
platform/hw/arch/amd64/locore.S

Loading GDT with:

lgdt (gdt64_ptr)

Segment Registers

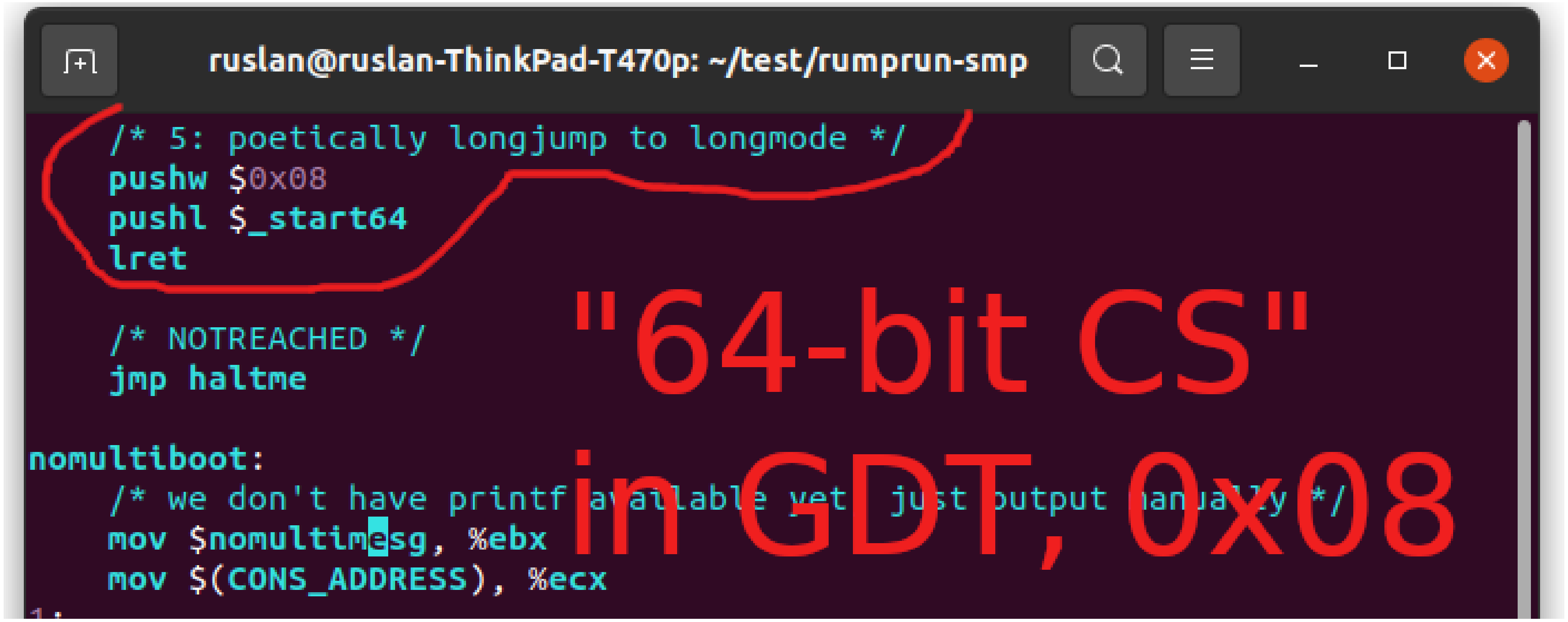
- Need to initialize all data-related segment registers
  - Use (byte) offsets in GDT
  - FS and GS are handled differently (TLS), you can change their 64-bit base



```
    lgdt (gdt64_ptr)
    pushl $0x0
    pushw $0x10
    pushl $1f
    lret

1:  movl $0x18, %eax
    movl %eax, %ds
    movl %eax, %es
    movl %eax, %ss

    xorl %eax, %eax
    movl %eax, %fs
    movl %eax, %gs
```

Segment Registers

- Need to initialize the code segment register
  - You cannot write to %cs, use “long/far jump” or “long/far return”, i.e., change both the instruction pointer and %cs
  - For long mode (your code is already in 64-bit mode), both *pushw* and *pushl* here **must be replaced** with *pushq*

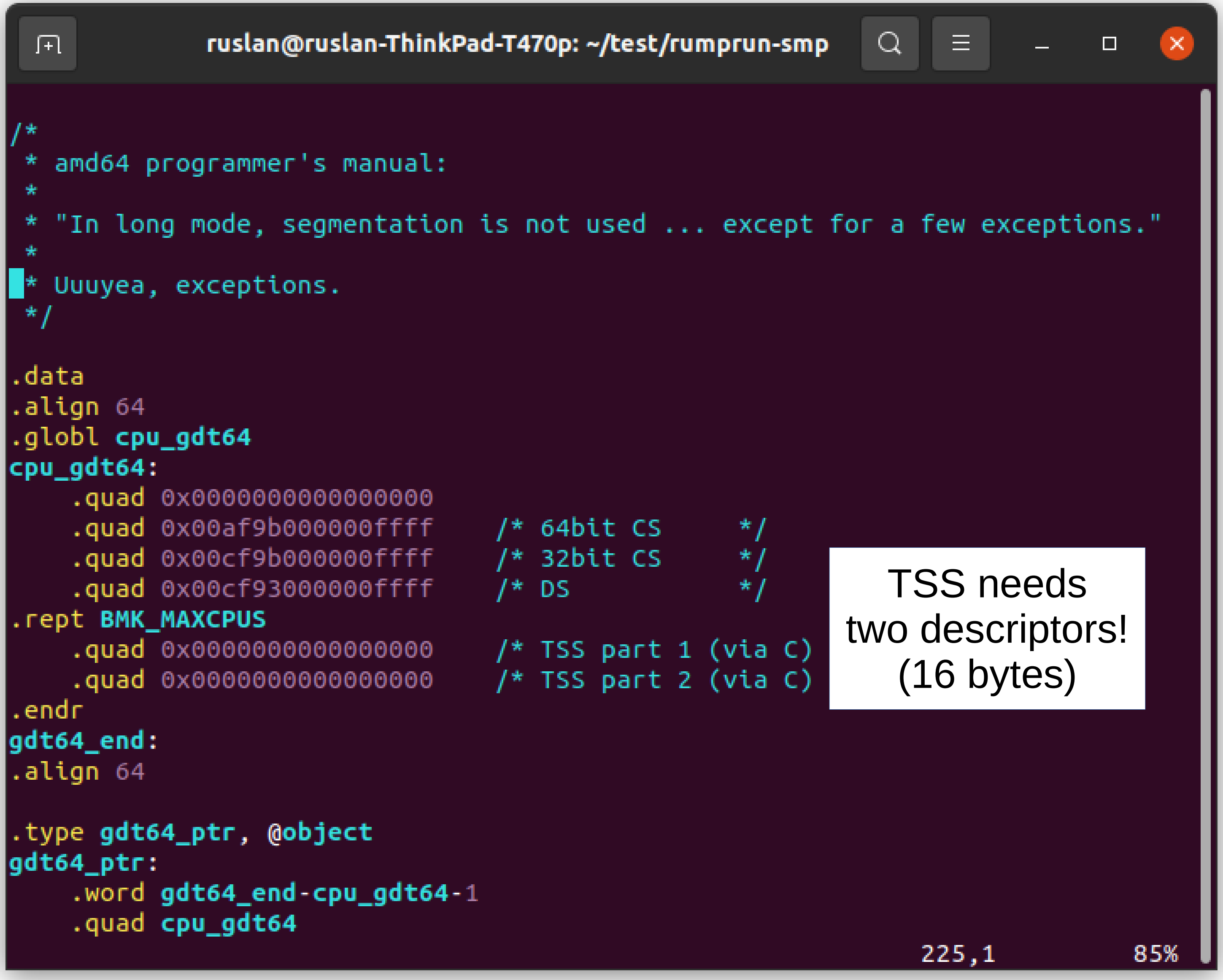


```
    /* 5: poetically longjump to longmode */
    pushw $0x08
    pushl $_start64
    lret

    /* NOTREACHED */
    jmp haltme

nomultiboot:
    /* we don't have printf available yet, just output manually */
    mov $nomultimesg, %ebx
    mov $(CONS_ADDRESS), %ecx
```

Task State Segment (TSS)

- Historically, ***hardware*** context switches were supported
  - But this approach is no longer preferred, ***software*** context switches can be more optimized (e.g., in Linux)
  - Hardware “tasks” use TSS
- Even though they are mostly obsolete in x86-64, you still need to set up TSS, at least one TSS per each CPU
- Why do you still need TSS?
  - If an interrupt happens while you are in user mode, you must set up a correct stack pointer for privileged mode
    - Remember that you cannot trust user-space applications! (And their stack pointer values.)

Task State Segment (TSS)


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

```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
	.quad 0x0000000000000000
	.quad 0x00af9b000000ffff	/* 64bit CS		*/
	.quad 0x00cf9b000000ffff	/* 32bit CS		*/
	.quad 0x00cf93000000ffff	/* DS			*/
.rept BMK_MAXCPUS
	.quad 0x0000000000000000	/* TSS part 1 (via C)
	.quad 0x0000000000000000	/* TSS part 2 (via C)
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
	.word gdt64_end-cpu_gdt64-1
	.quad cpu_gdt64
                                                        225,1          85%
```

TSS needs two descriptors! (16 bytes)

platform/hw/arch/amd64/locore.S

Task State Segment (TSS)

- In the previous example, TSS descriptors are not yet initialized
- They are initialized later in the code

See platform/hw/arch/amd64/machdep.c:

```
struct taskgate_descriptor {
    unsigned long td_lolimit:16;    /* segment extent (lsb) */
    unsigned long td_lobase:24; /* segment base address (lsb) */
    unsigned long td_type:5;    /* segment type */
    unsigned long td_dpl:2;     /* segment descriptor priority level */
    unsigned long td_p:1;       /* segment descriptor present */
    unsigned long td_hilimit:4; /* segment extent (msb) */
    unsigned long td_xx1:3;     /* avl, long and def32 (not used) */
    unsigned long td_gran:1;    /* limit granularity (byte/page) */
    unsigned long td_hibase:40; /* segment base address (msb) */
    unsigned long td_xx2:8;     /* reserved */
    unsigned long td_zero:5;    /* must be zero */
    unsigned long td_xx3:19;    /* reserved */
} __packed;
```

Task State Segment (TSS)

See platform/hw/arch/amd64/machdep.c:

```
static inline void
_init_taskgate(unsigned long cpu)
{
    unsigned long d = 4 + cpu * 2; /* TSS descriptor offset. */
    struct taskgate_descriptor *td = (void *)&cpu_gdt64[d];

    td->td_lolimit = 0;
    td->td_lobase = 0;
    td->td_type = 0x9;
    td->td_dpl = 0;
    td->td_p = 1;
    td->td_hilimit = 0xf;
    td->td_gran = 0;
    td->td_hibase = 0xffffffffffUL;
    td->td_zero = 0;
    amd64_ltr(d*8);
}
```

Assume just one CPU (cpu=0) for simplicity, i.e. d=4

amd64_ltr(0x20), i.e. 4*8=32

Global Descriptor Table (GDT)

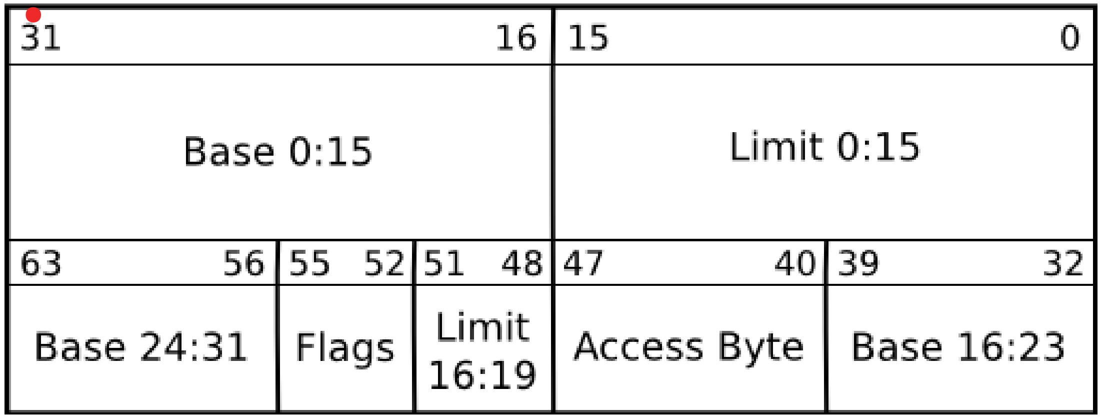
- Historically, x86 provided a very elaborated virtual memory system
  - Paging (e.g., Windows, Linux, and most other OSes)
  - Segmentation
  - Or even both (e.g., OS/2)
- Even though segmentation is now obsolete in x86-64, certain pieces have to be initialized correctly
  - Thread Local Storage (TLS) uses fs: or gs: segment registers
  - All other segment registers have to be initialized appropriately (cs, ds, es, ss)

Global Descriptor Table (GDT)

- A segment is an ***isolated*** contiguous piece of memory space, which has a ***base*** address and ***limit*** (size – 1)
  - You can restrict access to segments (e.g., each process can access just its own segments)
  - Compare that with a per-process page table (and TLB), though no complex translation is needed
    - An address will be segment_base + offset, offset is your program’s (segment’s) virtual address
- A segment register refers to a segment ***selector***
  - Cannot be arbitrarily replaced without proper permissions
- Except TLS, you want to select segments with base = 0 and limit = MAX_ADDRESS for x86-64 (“flat memory”)

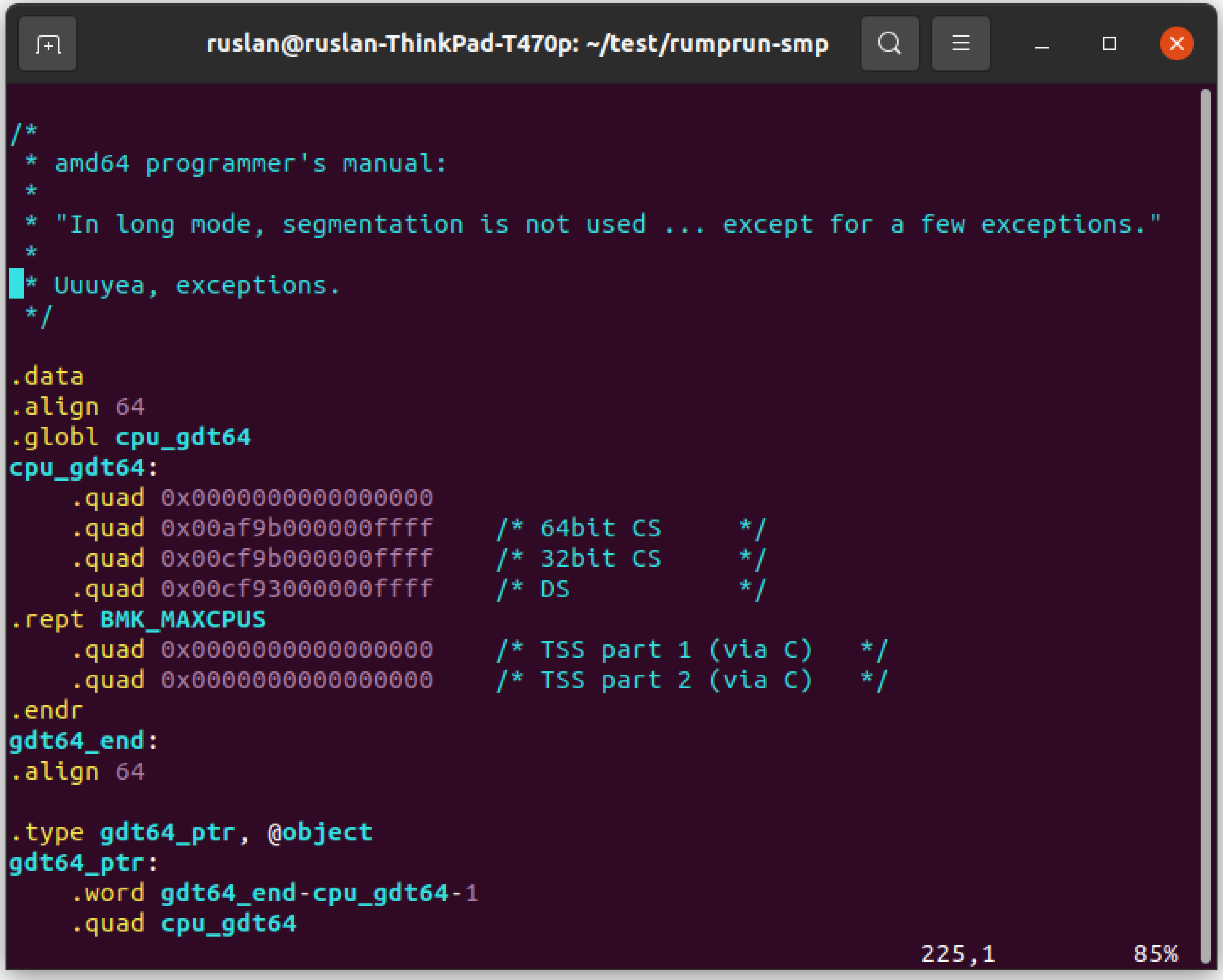
Global Descriptor Table (GDT)

- GDT is a table of segment ***descriptors***
  - Each entry is 8 bytes
  - The first entry is reserved (0)
  - Other entries can define segments for code and data
  - For x86-64, base and limit are always 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF, respectively



* The picture is taken from https://wiki.osdev.org/Global_Descriptor_Table

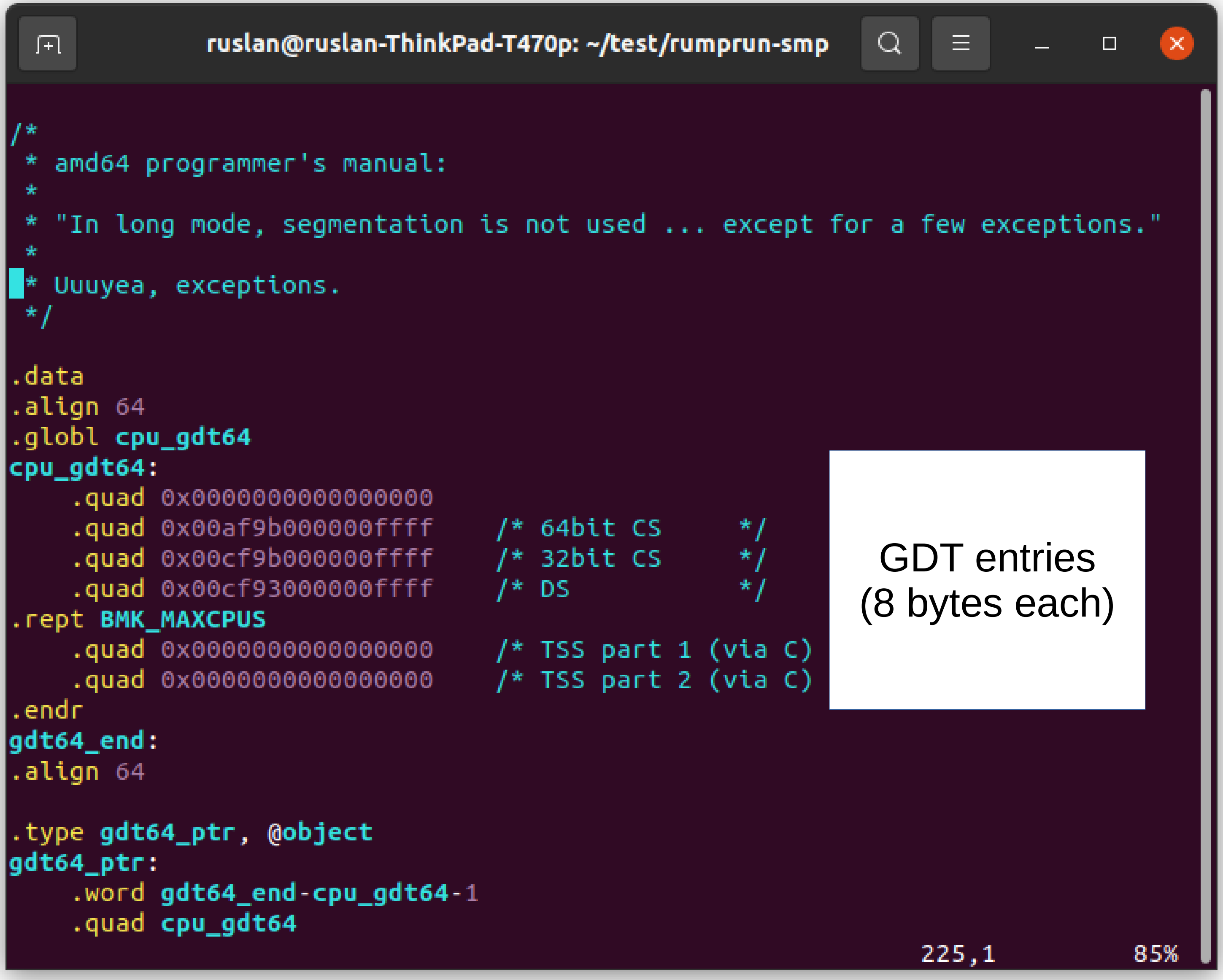
Example of GDT (rumprun-smp)



```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)   */
    .quad 0x0000000000000000    /* TSS part 2 (via C)   */
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
                                                           225,1          85%
```

platform/hw/arch/amd64/locore.S

Example of GDT (rumprun-smp)

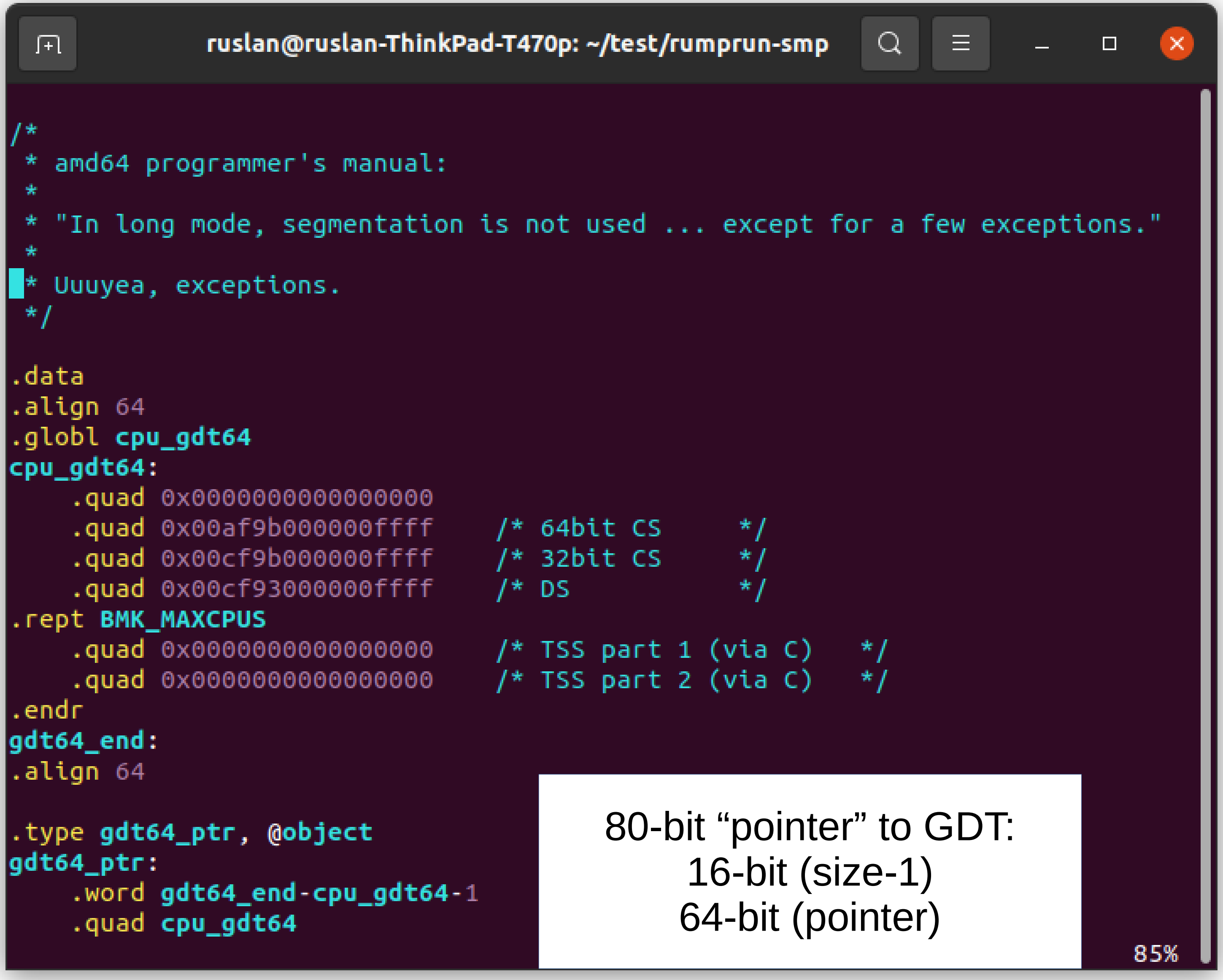


```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)
    .quad 0x0000000000000000    /* TSS part 2 (via C)
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
                                                                  225,1         85%
```

GDT entries
(8 bytes each)

platform/hw/arch/amd64/locore.S

Example of GDT (rumprun-smp)



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

```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)   */
    .quad 0x0000000000000000    /* TSS part 2 (via C)   */
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
```

85%

80-bit "pointer" to GDT:
16-bit (size-1)
64-bit (pointer)

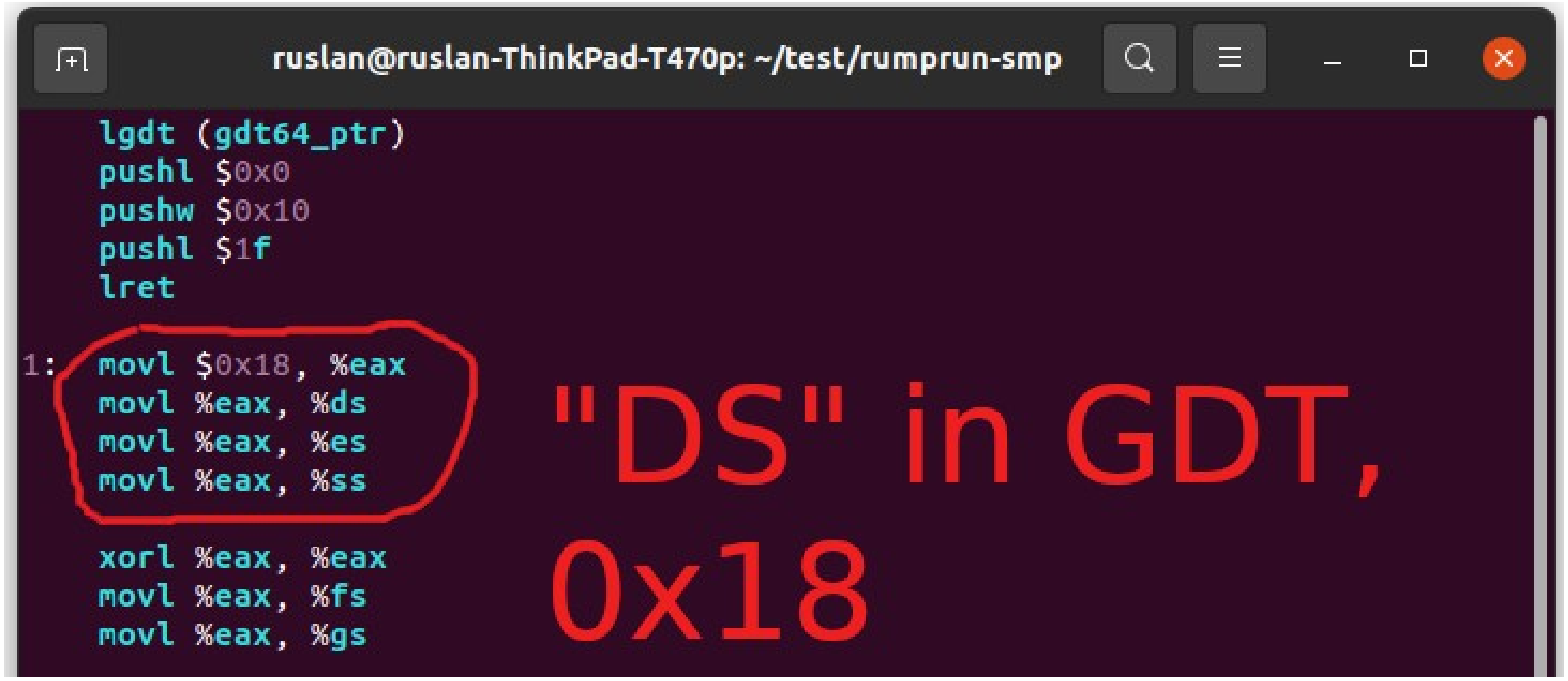
platform/hw/arch/amd64/locore.S

Loading GDT with:

lgdt (gdt64_ptr)

Segment Registers

- Need to initialize all data-related segment registers
  - Use (byte) offsets in GDT
  - FS and GS are handled differently (TLS), you can change their 64-bit base

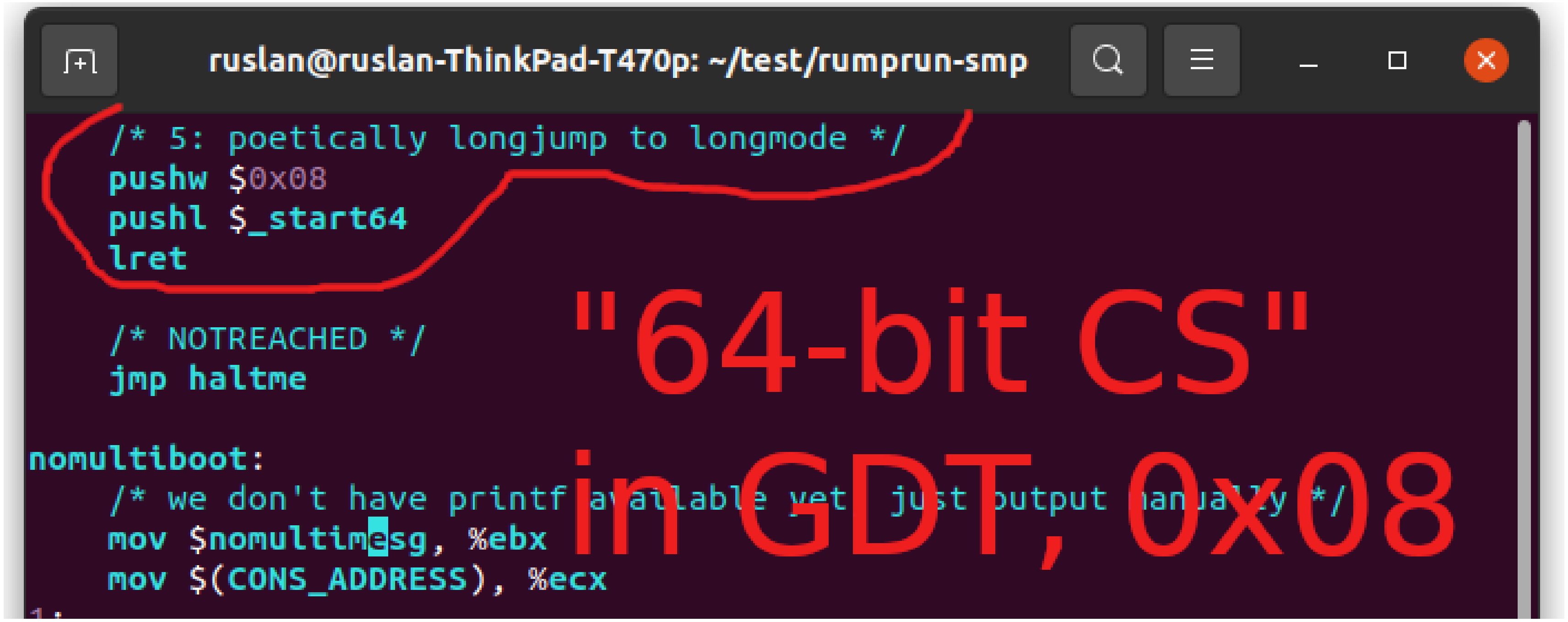


```
    lgdt (gdt64_ptr)
    pushl $0x0
    pushw $0x10
    pushl $1f
    lret

1:  movl $0x18, %eax
    movl %eax, %ds
    movl %eax, %es
    movl %eax, %ss

    xorl %eax, %eax
    movl %eax, %fs
    movl %eax, %gs
```

"DS" in GDT, 0x18

Segment Registers

- Need to initialize the code segment register
  - You cannot write to %cs, use “long/far jump” or “long/far return”, i.e., change both the instruction pointer and %cs
  - For long mode (your code is already in 64-bit mode), both *pushw* and *pushl* here **must be replaced** with *pushq*

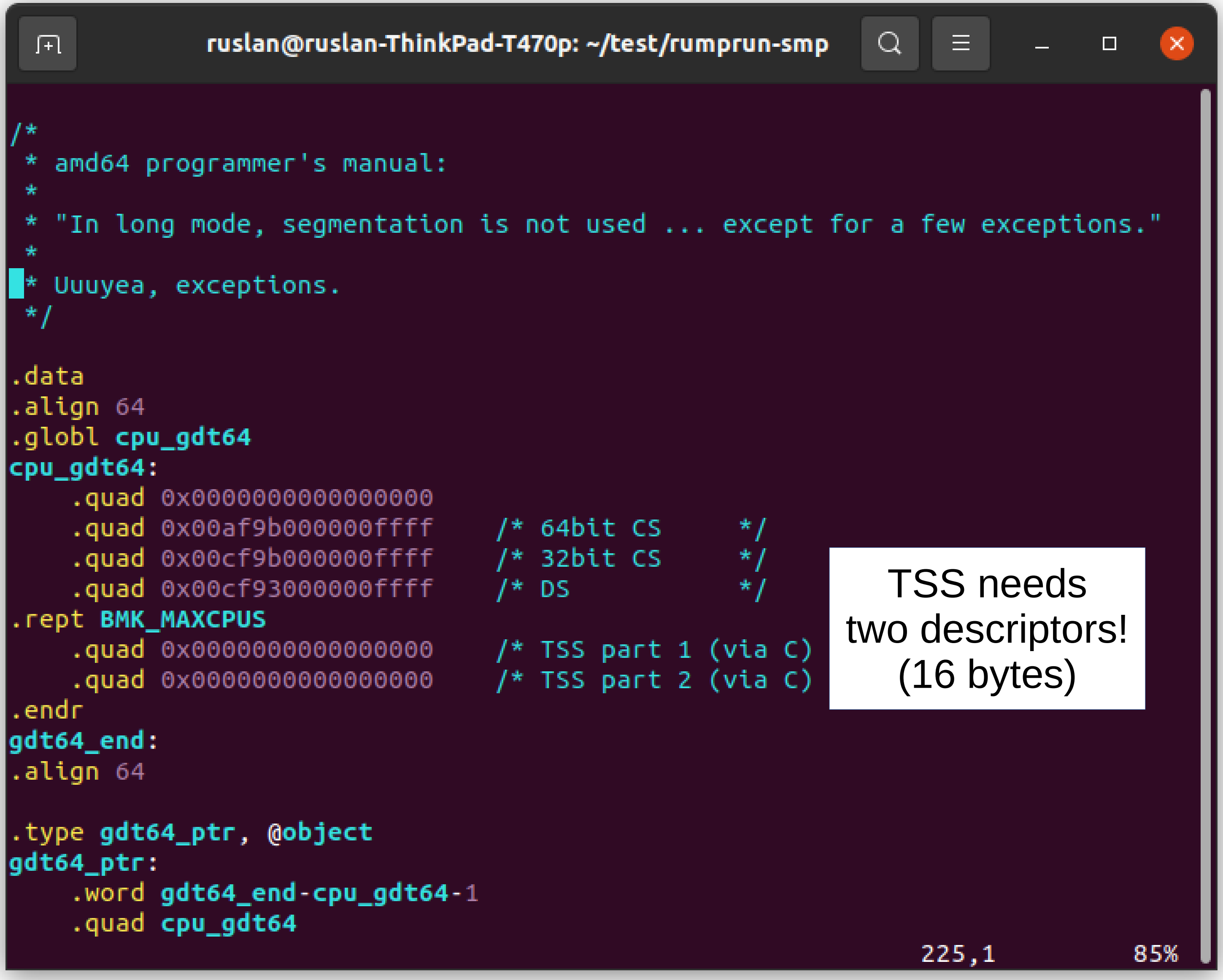


```
    /* 5: poetically longjump to longmode */
    pushw $0x08
    pushl $_start64
    lret

    /* NOTREACHED */
    jmp haltme

nomultiboot:
    /* we don't have printf available yet, just output manually */
    mov $nomultimesg, %ebx
    mov $(CONS_ADDRESS), %ecx
```

Task State Segment (TSS)

- Historically, ***hardware*** context switches were supported
  - But this approach is no longer preferred, ***software*** context switches can be more optimized (e.g., in Linux)
  - Hardware "tasks" use TSS
- Even though they are mostly obsolete in x86-64, you still need to set up TSS, at least one TSS per each CPU
- Why do you still need TSS?
  - If an interrupt happens while you are in user mode, you must set up a correct stack pointer for privileged mode
    - Remember that you cannot trust user-space applications! (And their stack pointer values.)

Task State Segment (TSS)



```
/*
 * amd64 programmer's manual:
 *
 * "In long mode, segmentation is not used ... except for a few exceptions."
 *
 * Uuuyea, exceptions.
 */

.data
.align 64
.globl cpu_gdt64
cpu_gdt64:
    .quad 0x0000000000000000
    .quad 0x00af9b000000ffff    /* 64bit CS     */
    .quad 0x00cf9b000000ffff    /* 32bit CS     */
    .quad 0x00cf93000000ffff    /* DS           */
.rept BMK_MAXCPUS
    .quad 0x0000000000000000    /* TSS part 1 (via C)
    .quad 0x0000000000000000    /* TSS part 2 (via C)
.endr
gdt64_end:
.align 64

.type gdt64_ptr, @object
gdt64_ptr:
    .word gdt64_end-cpu_gdt64-1
    .quad cpu_gdt64
                                                            225,1         85%
```

TSS needs
two descriptors!
(16 bytes)

platform/hw/arch/amd64/locore.S

Task State Segment (TSS)

- In the previous example, TSS descriptors are not yet initialized
- They are initialized later in the code

See platform/hw/arch/amd64/machdep.c:

```
struct taskgate_descriptor {
    unsigned long td_lolimit:16;    /* segment extent (lsb) */
    unsigned long td_lobase:24; /* segment base address (lsb) */
    unsigned long td_type:5;    /* segment type */
    unsigned long td_dpl:2;     /* segment descriptor priority level */
    unsigned long td_p:1;       /* segment descriptor present */
    unsigned long td_hilimit:4; /* segment extent (msb) */
    unsigned long td_xx1:3;     /* avl, long and def32 (not used) */
    unsigned long td_gran:1;    /* limit granularity (byte/page) */
    unsigned long td_hibase:40; /* segment base address (msb) */
    unsigned long td_xx2:8;     /* reserved */
    unsigned long td_zero:5;    /* must be zero */
    unsigned long td_xx3:19;    /* reserved */
} __packed;
```

Task State Segment (TSS)

See platform/hw/arch/amd64/machdep.c:

```
static inline void
_init_taskgate(unsigned long cpu)
{
    unsigned long d = 4 + cpu * 2; /* TSS descriptor offset. */
    struct taskgate_descriptor *td = (void *)&cpu_gdt64[d];

    td->td_lolimit = 0;
    td->td_lobase = 0;
    td->td_type = 0x9;
    td->td_dpl = 0;
    td->td_p = 1;
    td->td_hilimit = 0xf;
    td->td_gran = 0;
    td->td_hibase = 0xffffffffffUL;
    td->td_zero = 0;
    amd64_ltr(d*8);
}
```

Assume just one CPU (cpu=0) for simplicity, i.e. d=4

amd64_ltr(0x20), i.e. 4*8=32

Interrupt Descriptor Table (IDT)

- Loading IDT is similar to that of GDT. You need to have an 80-bit “pointer”. Below is an example for this in C.

From platform/hw/arch/amd64/machdep.c:

```
struct region_descriptor {
    unsigned short rd_limit;
    unsigned long rd_base;
} __attribute__((__packed__));  <-- “packed” is important here to avoid padding

struct region_descriptor region;
...
region.rd_limit = sizeof(idt)-1;
region.rd_base = (uintptr_t)(void *)idt;
amd64_lidt(&region);
```

Where amd64_lidt is just the **lidt** instruction

```
ENTRY(amd64_lidt)
    lidt (%rdi)
    ret
END(amd64_lidt)
```