CSE 511: Operating Systems Design

Lectures 3,4

OS Boot Process

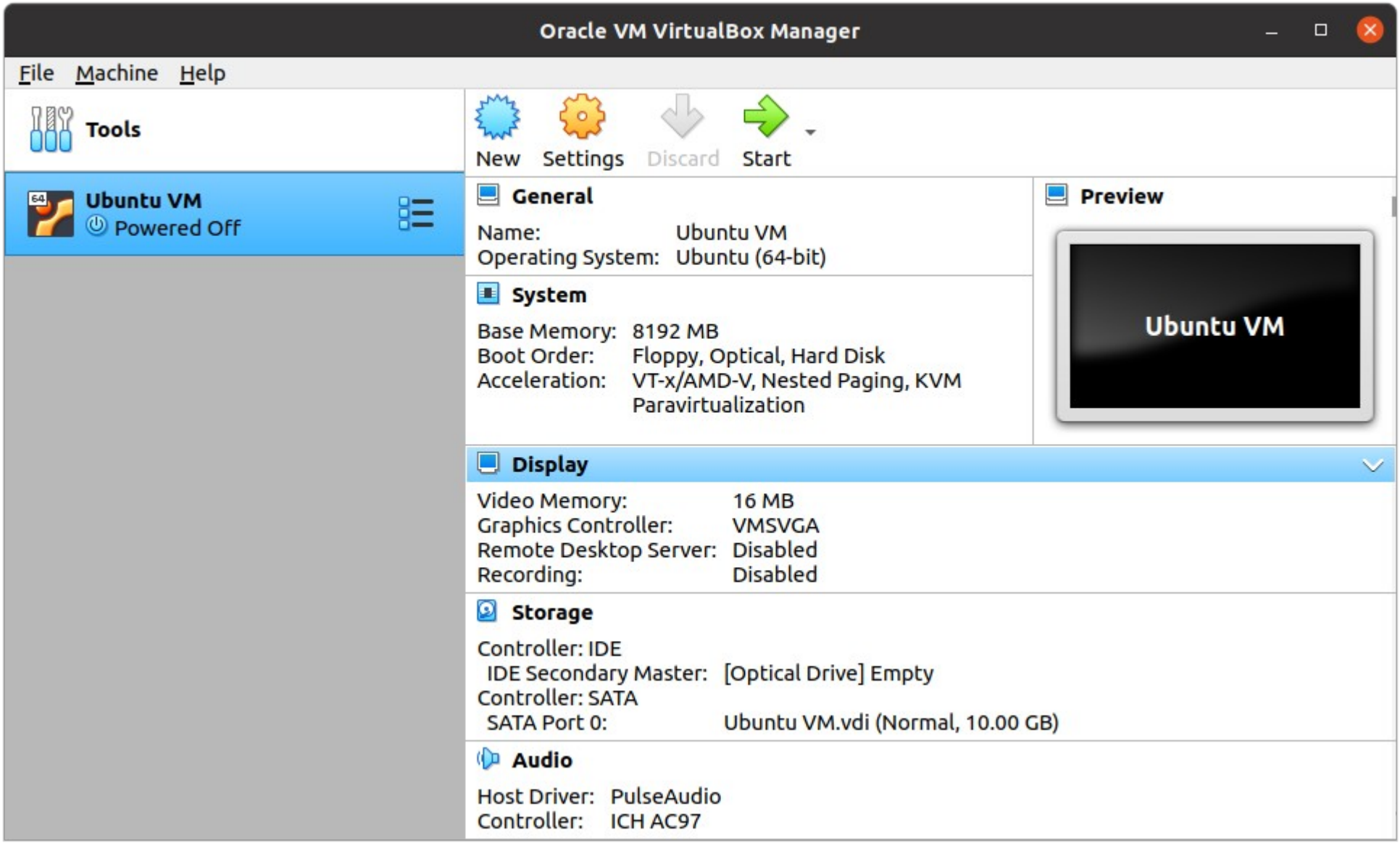
UEFI Programming

Assignment 1

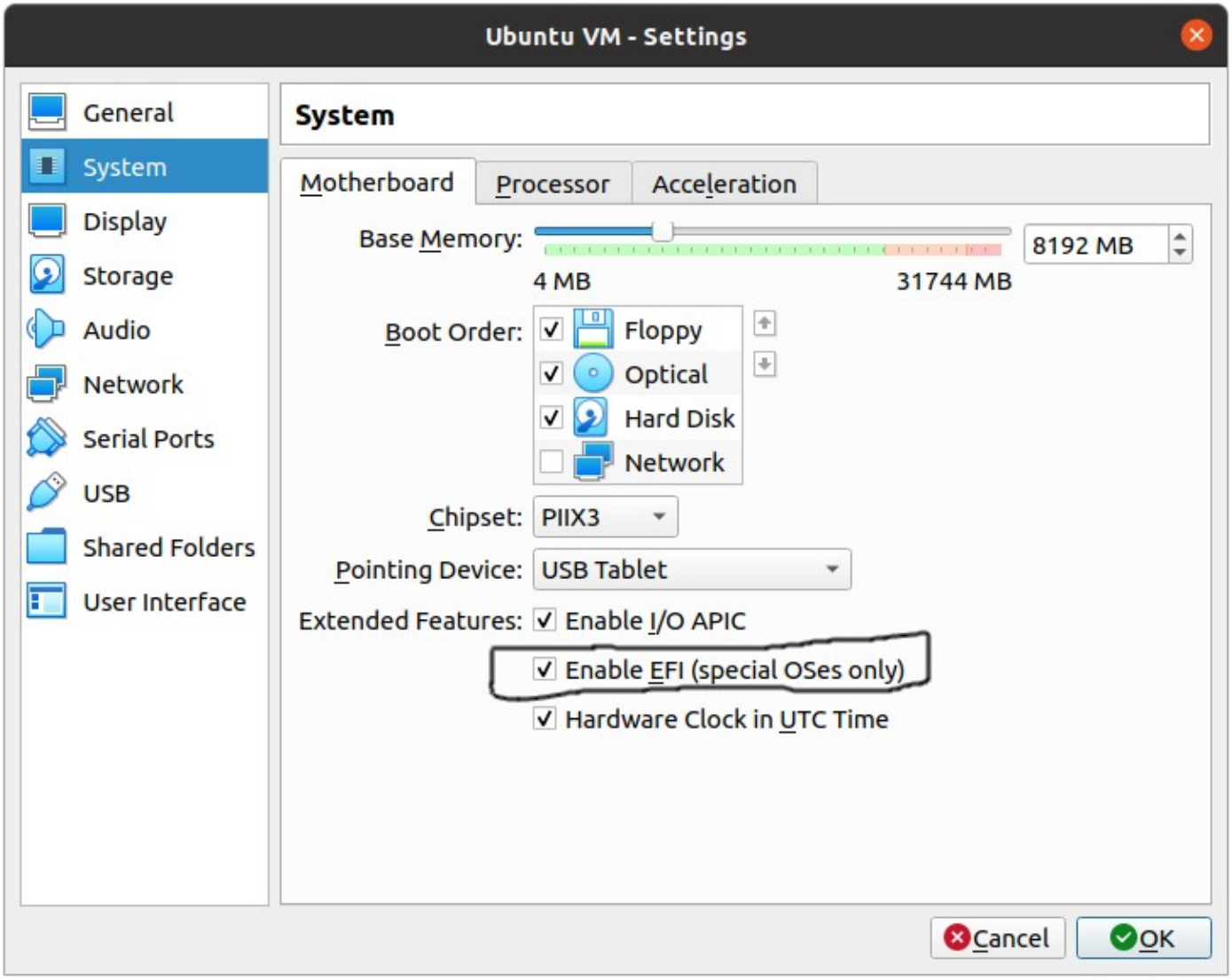
UEFI Firmware

- Used by all newer hardware but legacy (BIOS) boot is still supported
  - Legacy BIOS boot was supposed to be phased out completely in 2020. Cannot boot MS-DOS anymore.
  - Your machine may still use legacy boot!
- **Since all machines vary, and BIOS legacy boot is still widely used, we will use VirtualBox and qemu for our assignments/projects to make things simpler**
  - UEFI boot will work even for legacy machines
  - Ubuntu: `sudo apt-get install virtualbox`

UEFI Firmware



UEFI Firmware



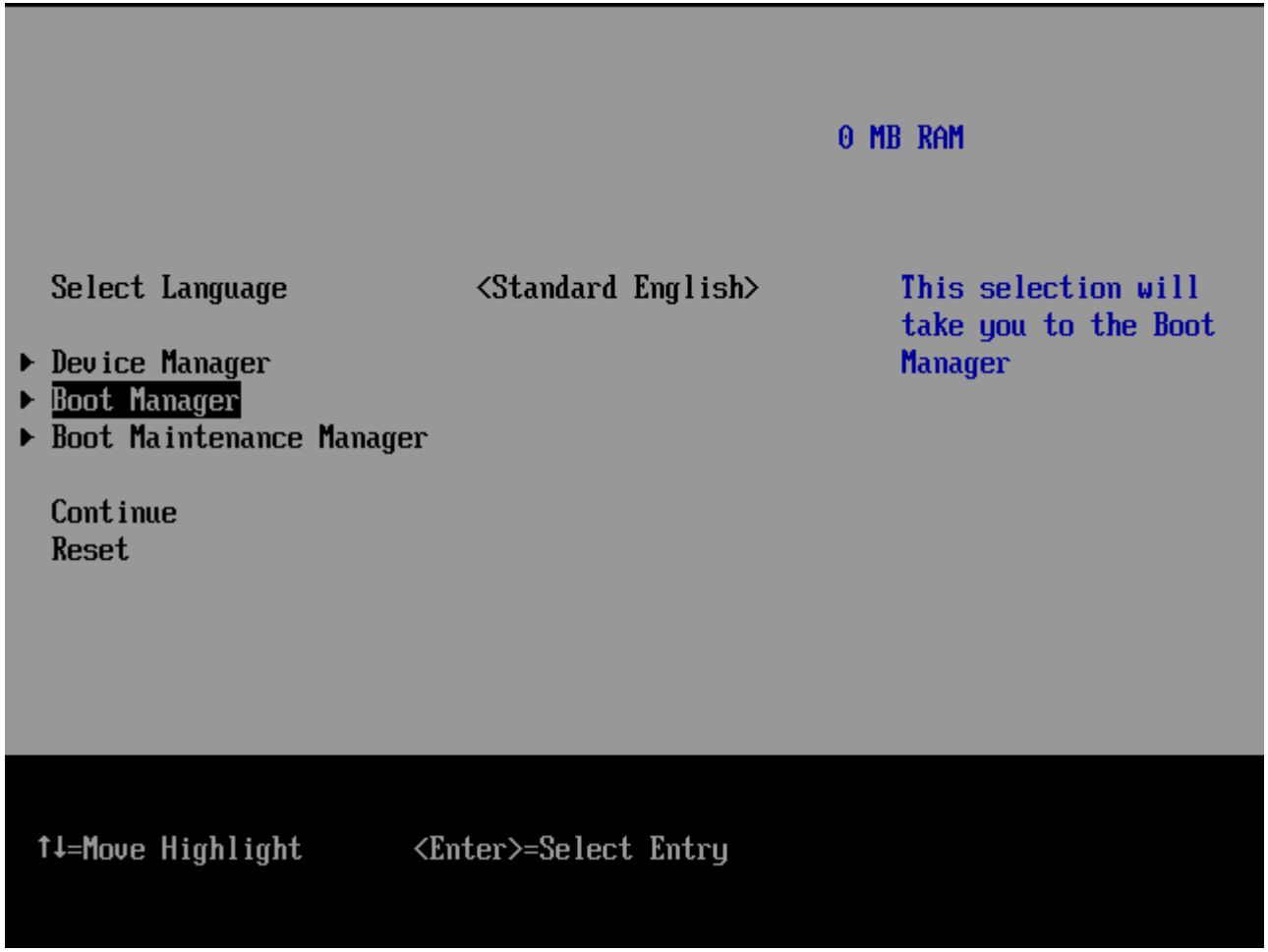
UEFI is disabled by default.

Enable it in Settings!

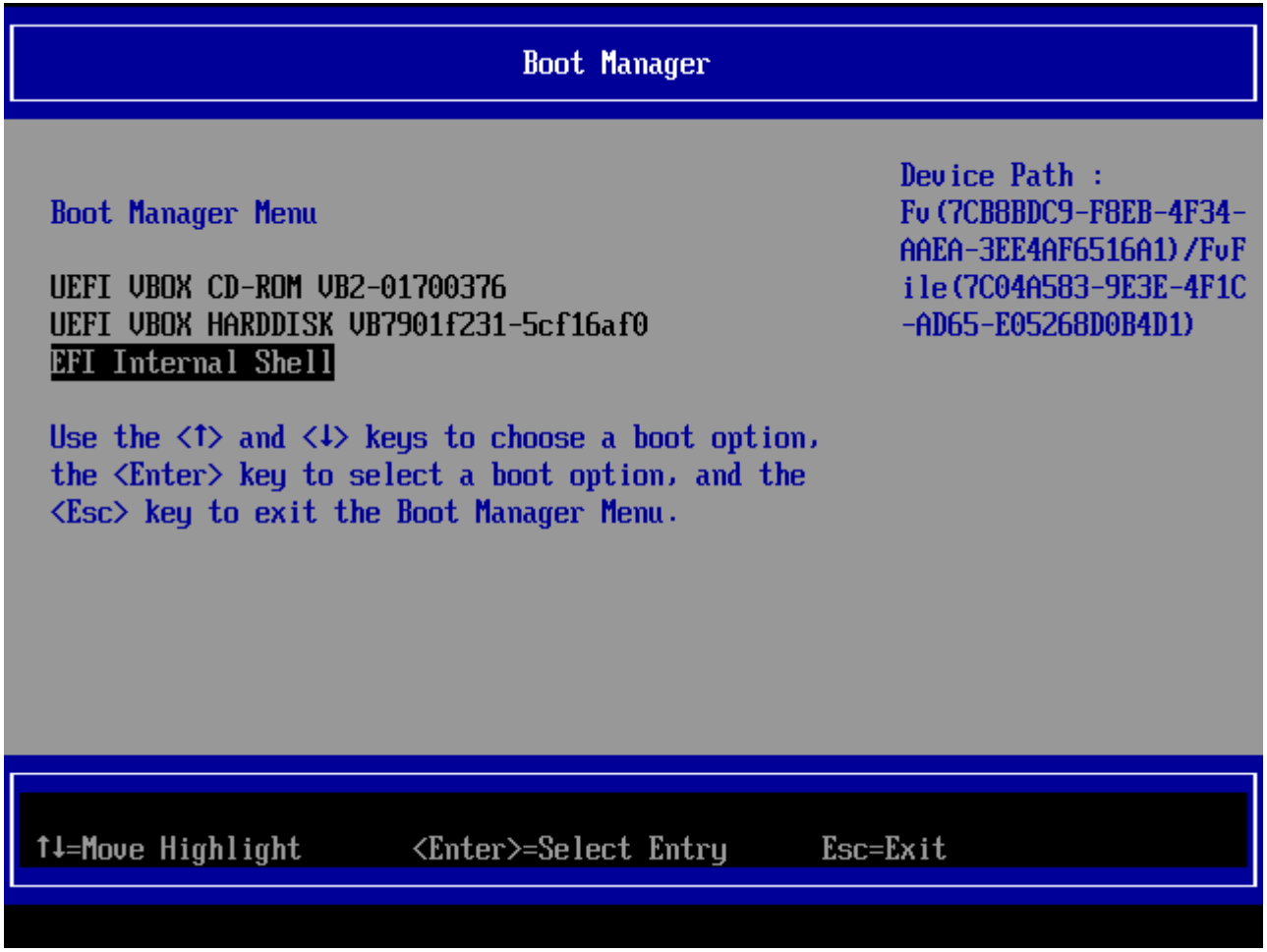
UEFI Shell

Booting from Ubuntu 20.04 ISO image

UEFI Shell


0 MB RAM
Select Language
<Standard English>
This selection will take you to the Boot Manager
Device Manager
Boot Manager
Boot Maintenance Manager
Continue
Reset
↑↓=Move Highlight
<Enter>=Select Entry

UEFI Shell



List of bootable devices

UEFI Shell

```
      FS0: Alias(s):F0c::BLK0:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
      FS1: Alias(s):CD0c0::BLK2:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
      FS2: Alias(s):CD0c1::BLK4:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x1)
     BLK5: Alias(s):
          PciRoot(0x0)/Pci(0xD,0x0)/Sata(0x0,0xFFFF,0x0)
     BLK1: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
     BLK3: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
Press ESC in 4 seconds to skip startup.nsh or any other key to continue.
Shell>
Shell> FS0:
FS0:\> ls
Directory of: FS0:\
00/00/0000  00:00 <DIR>         2,048  .disk
00/00/0000  00:00 <DIR>         2,048  boot
00/00/0000  00:00 <DIR>         2,048  casper
00/00/0000  00:00 <DIR>         2,048  dists
00/00/0000  00:00 <DIR>         2,048  EFI
00/00/0000  00:00 <DIR>         2,048  install
00/00/0000  00:00 <DIR>        34,816  isolinux
00/00/0000  00:00              42,622  md5sum.txt
00/00/0000  00:00 <DIR>         2,048  pics
00/00/0000  00:00 <DIR>         2,048  pool
00/00/0000  00:00 <DIR>         2,048  preseed
00/00/0000  00:00                 231  README.diskdefines
00/00/0000  00:00                   0  ubuntu
          3 File(s)      42,853 bytes
         10 Dir(s)
FS0:\> ls EFI\BOOT
Directory of: FS0:\EFI\BOOT\
00/00/0000  00:00           1,334,816  BOOTx64.EFI
00/00/0000  00:00           1,419,136  grubx64.efi
00/00/0000  00:00           1,269,496  mmx64.efi
          3 File(s)   4,023,448 bytes
          0 Dir(s)
FS0:\> _
```

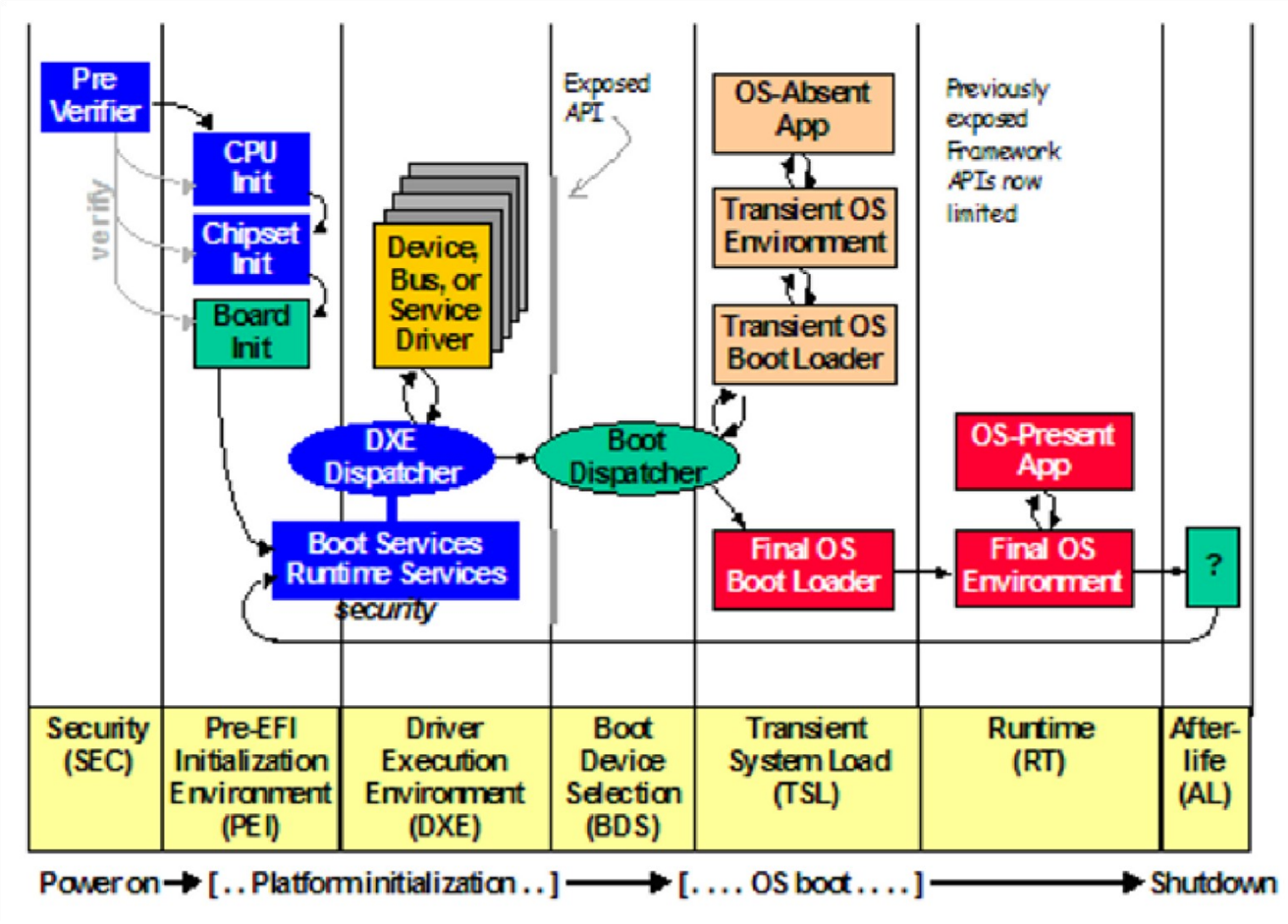
BOOTx64.EFI is the primary ISO boot file

UEFI Shell: Booting example (mmx64)

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
      FS0: Alias(s):F0c::BLK0:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
      FS1: Alias(s):CD0c0::BLK2:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
      FS2: Alias(s):CD0c1::BLK4:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x1)
     BLK5: Alias(s):
          PciRoot(0x0)/Pci(0xD,0x0)/Sata(0x0,0xFFFF,0x0)
     BLK1: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
     BLK3: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/CDROM(0x0)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell> FS0:
FS0:\> EFI\BOOT\mmx64.efi_
```

UEFI Shell: Booting example (mmx64)

UEFI Boot Process



* The picture is taken from EDK2 specification: https://edk2-docs.gitbook.io/edk-ii-build-specification/2_design_discussion/23_boot_sequence

UEFI Services

- Native code
  - No legacy 16-bit BIOS code
  - 32-bit and 64-bit depending on an OS [For x86-64, 32-bit support is optional]
- UEFI boot time services
  - Destroyed when an OS is booted up
  - Cannot access corresponding methods afterwards
- UEFI run time service
  - Can be used within the booted OS
  - Some similarity to what BIOS used to be before its 16-bit mode became obsolete

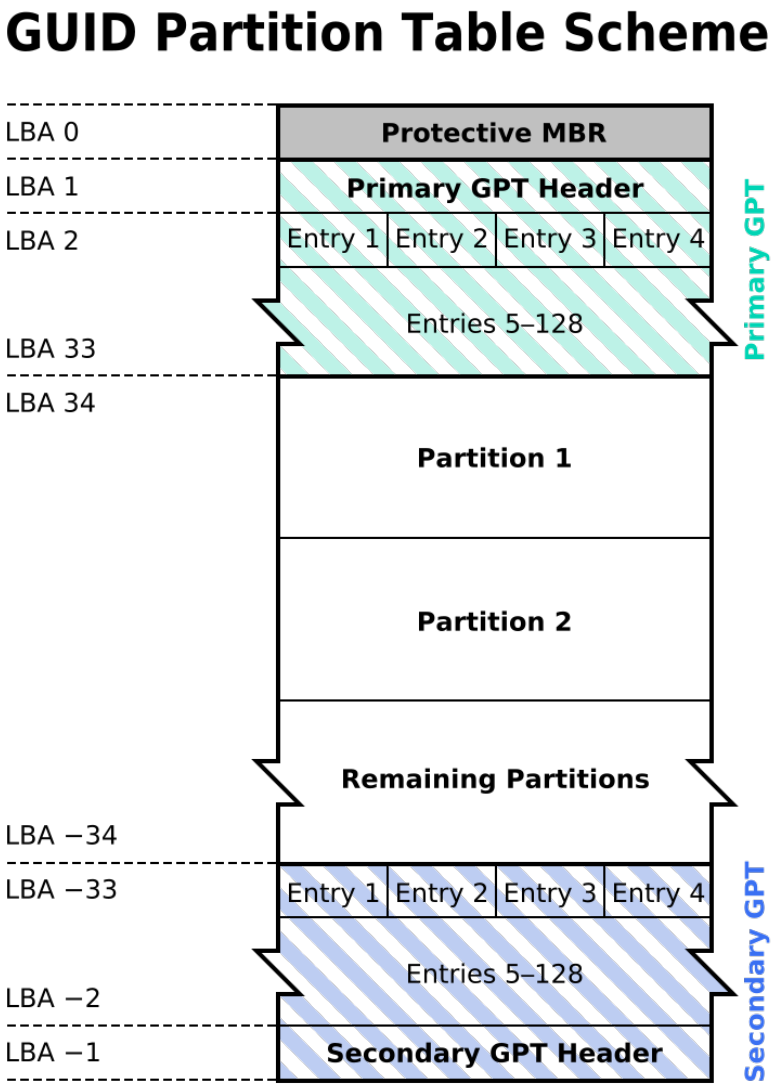
Legacy Master Boot Record (MBR)

- Resides in disk sector #0 and is dual-purposed (sectors are typically 512 bytes for HDD)
  - The first-stage boot loader code that BIOS loads to memory address 0x7C00 – 446 bytes
    - Too small but it is feasible to create multiple stages, etc. Boot sector writing now is a lost art
  - A table of 4 partitions (max), more can be added by extra chaining – 4x16 bytes
    - Due to 32-bit logical block addresses (LBA), the disk space is limited to (2^32 * 512) = 2 TB
- Not used for ISO 9660 (CD/DVD), their sectors are 2048 bytes
  - Another “El-Torito” protocol is used instead

GUID Partition Table (GPT)

- MBR partition table is now obsolete due to the 2 TB limit
  - A dummy, “protective”, MBR sector is created which has no real partition table or boot code
- The protective MBR is followed by GPT (sector #1), a new partition table
  - A dedicated “EFI system partition” stores bootable files
  - Other partitions can be created as necessary
  - All partitions have unique 128-bit GUID identifiers
    - Can be really useful for an OS kernel to map these partitions (especially the bootable partition) to its internal names, e.g., the /dev namespace in Linux

GUID Partition Table (GPT)



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

EFI System Partition

- Typically initialized to FAT32
  - Widely supported across different OSs
- Can have multiple bootable files
  - Multiple OSs can be installed, one .efi program (“boot selector”) can load another .efi program, etc
  - grubx64.efi for the GRUB boot loader (Linux)
  - bootmgfw.efi for the Windows boot manager

Executable Format

- .EFI files are essentially .DLL files (with some minor differences)
  - Microsoft ABI and calling conventions
- .DLL is a *relocatable* binary (similar to ELF relocatable object files)
  - Can be loaded anywhere in the address space
    - UEFI creates a “default page table” already
  - No symbols are imported or exported, one function is the entry point, no symbols

Executable Format

```
ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/build-x64
ruslan@ruslan-ThinkPad-T470p:~/examples/shim/build-x64$ objdump -x shimx64.efi

shimx64.efi:     file format pei-x86-64
shimx64.efi
architecture: i386:x86-64, flags 0x0000012f:
HAS_RELOC, EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x000000018002774a

Characteristics 0x2022
        executable
        large address aware
        DLL

Time/Date               Mon Jan 25 15:32:58 2021
Magic                   020b    (PE32+)
MajorLinkerVersion      0
MinorLinkerVersion      0
SizeOfCode              000000000004a000
SizeOfInitializedData   000000000002da00
SizeOfUninitializedData 0000000000000000
AddressOfEntryPoint     000000000002774a
BaseOfCode              0000000000001000
ImageBase               0000000180000000
SectionAlignment        00001000
FileAlignment           00000200
MajorOSystemVersion     0
MinorOSystemVersion     0
MajorImageVersion       0
MinorImageVersion       0
MajorSubsystemVersion   0
MinorSubsystemVersion   0
Win32Version            00000000
SizeOfImage             0007a000
SizeOfHeaders           00000400
CheckSum                00000000
Subsystem               0000000a        (EFI application)
DllCharacteristics      00000160
SizeOfStackReserve      0000000000000000
SizeOfStackCommit       0000000000000000
SizeOfHeapReserve       0000000000000000
SizeOfHeapCommit        0000000000000000
LoaderFlags             00000000
NumberOfRvaAndSizes     00000010

The Data Directory
Entry 0 0000000000000000 00000000 Export Directory [.edata (or where ever we fou
nd it)]
Entry 1 0000000000000000 00000000 Import Directory [parts of .idata]
Entry 2 0000000000000000 00000000 Resource Directory [.rsrc]
Entry 3 0000000000000000 00000000 Exception Directory [.pdata]
Entry 4 0000000000000000 00000000 Security Directory
Entry 5 0000000000077000 00002858 Base Relocation Directory [.reloc]
Entry 6 0000000000000000 00000000 Debug Directory
Entry 7 0000000000000000 00000000 Description Directory
Entry 8 0000000000000000 00000000 Special Directory
```

```
ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/build-x64
        reloc  125 offset  b50 [75b50] DIR64
        reloc  126 offset  b58 [75b58] DIR64
        reloc  127 offset  b78 [75b78] DIR64
        reloc  128 offset  b80 [75b80] DIR64
        reloc  129 offset  ba0 [75ba0] DIR64
        reloc  130 offset  ba8 [75ba8] DIR64
        reloc  131 offset  bd0 [75bd0] DIR64
        reloc  132 offset  bd8 [75bd8] DIR64
        reloc  133 offset  be0 [75be0] DIR64
        reloc  134 offset  be8 [75be8] DIR64
        reloc  135 offset  bf0 [75bf0] DIR64
        reloc  136 offset  bf8 [75bf8] DIR64
        reloc  137 offset  c10 [75c10] DIR64
        reloc  138 offset  c18 [75c18] DIR64
        reloc  139 offset  c20 [75c20] DIR64
        reloc  140 offset  c40 [75c40] DIR64
        reloc  141 offset  c48 [75c48] DIR64
        reloc  142 offset  c50 [75c50] DIR64
        reloc  143 offset  c70 [75c70] DIR64
        reloc  144 offset  c78 [75c78] DIR64
        reloc  145 offset  c80 [75c80] DIR64
        reloc  146 offset  ca0 [75ca0] DIR64
        reloc  147 offset  ca8 [75ca8] DIR64
        reloc  148 offset  cb0 [75cb0] DIR64
        reloc  149 offset  cd0 [75cd0] DIR64
        reloc  150 offset  cd8 [75cd8] DIR64
        reloc  151 offset  ce0 [75ce0] DIR64
        reloc  152 offset  d00 [75d00] DIR64
        reloc  153 offset  d08 [75d08] DIR64
        reloc  154 offset  d10 [75d10] DIR64
        reloc  155 offset  d30 [75d30] DIR64
        reloc  156 offset  d38 [75d38] DIR64
        reloc  157 offset  d40 [75d40] DIR64
        reloc  158 offset  d60 [75d60] DIR64
        reloc  159 offset  d68 [75d68] DIR64
        reloc  160 offset  d70 [75d70] DIR64
        reloc  161 offset  d90 [75d90] DIR64
        reloc  162 offset  d98 [75d98] DIR64
        reloc  163 offset  da0 [75da0] DIR64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00049e48  0000000180001000  0000000180001000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rdata        00029e30  000000018004b000  000000018004b000  0004a400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00001000  0000000180075000  0000000180075000  00074400  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  3 .reloc        00002858  0000000180077000  0000000180077000  00075400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
no symbols

ruslan@ruslan-ThinkPad-T470p:~/examples/shim/build-x64$
```

Executable Format

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim

```
}

extern EFI_STATUS EFIAPI
efi_main(EFI_HANDLE passed_image_handle, EFI_SYSTEM_TABLE *passed_systab);


EFI_STATUS EFIAPI
efi_main (EFI_HANDLE passed_image_handle, EFI_SYSTEM_TABLE *passed_systab)
{
    EFI_STATUS efi_status;
    EFI_HANDLE image_handle;

    systab = passed_systab;
    image_handle = global_image_handle = passed_image_handle;

    InitializeLib(image_handle, systab);

    verification_method = VERIFIED_BY_NOTHING;

    vendor_authorized_size = cert_table.vendor_authorized_size;
                                                                    2607,1        97%
```

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim

```
endef

%.efi : $(BUILDDIR)/%.efi

src/%.o : $(BUILDDIR)/src/%.o

%.crt : $(BUILDDIR)/certdb/%.crt
%.cer : $(BUILDDIR)/certdb/%.cer

EFI_LDFLAGS = $(EFI_DEBUG_LDFLAGS) /dll /entry:efi_main /safeseh:no /nodefaultlib $(EFI_ARCH_LDFLAGS)

%.efi: %.efi.dll fwimage/fwimage
    ./fwimage/fwimage app $< $@
    @chmod 755 $@

ifneq ($(origin ENABLE_SBSIGN),undefined)
%.efi.signed: %.efi shim.key shim.crt
    @$(SBSIGN) \
        --key $(BUILDDIR)/certdb/shim.key \
        --cert $(BUILDDIR)/certdb/shim.crt \
search hit BOTTOM, continuing at TOP                                89,0-1        77%
```

UEFI Secure Boot

- Malware can hijack the boot process
- One way to alleviate this concern is to use “digital signing” of .efi files with a private key
  - A trusted party (e.g., Microsoft, Red Hat, etc) can sign with their private key
  - The private key is unknown to the public
- The .efi file is then verified with a public key
  - The public key is installed in the UEFI database

EFI_SYSTEM_TABLE

- The topmost data structure which contains function pointers to
  - UEFI boot time services
  - UEFI run time services
  - Protocol services
    - Each protocol is identified with its own GUID

EFI_SYSTEM_TABLE

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/edk2

```
typedef struct {
  ///
  /// The table header for the EFI System Table.
  ///
  EFI_TABLE_HEADER                Hdr;
  ///
  /// A pointer to a null terminated string that identifies the vendor
  /// that produces the system firmware for the platform.
  ///
  CHAR16                          *FirmwareVendor;
  ///
  /// A firmware vendor specific value that identifies the revision
  /// of the system firmware for the platform.
  ///
  UINT32                          FirmwareRevision;
  ///
  /// The handle for the active console input device. This handle must support
  /// EFI_SIMPLE_TEXT_INPUT_PROTOCOL and EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.
  ///
  EFI_HANDLE                      ConsoleInHandle;
  ///
  /// A pointer to the EFI_SIMPLE_TEXT_INPUT_PROTOCOL interface that is
  /// associated with ConsoleInHandle.
  ///
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL    *ConIn;
  ///
  /// The handle for the active console output device.
  ///
  EFI_HANDLE                      ConsoleOutHandle;
  ///
  /// A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL interface
  /// that is associated with ConsoleOutHandle.
  ///
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *ConOut;
  ///
  /// The handle for the active standard error console device.
  /// This handle must support the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.
  ///
  EFI_HANDLE                      StandardErrorHandle;
  ///
  /// A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL interface
  /// that is associated with StandardErrorHandle.
  ///
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *StdErr;
  ///
  /// A pointer to the EFI Runtime Services Table.
  ///
  EFI_RUNTIME_SERVICES            *RuntimeServices;
  ///
  /// A pointer to the EFI Boot Services Table.
  ///
  EFI_BOOT_SERVICES               *BootServices;
  ///
  /// The number of system configuration tables in the buffer ConfigurationTable.
  ///
  UINTN                           NumberOfTableEntries;
  ///
  /// A pointer to the system configuration tables.
  /// The number of entries in the table is NumberOfTableEntries.
  ///
  EFI_CONFIGURATION_TABLE         *ConfigurationTable;
} EFI_SYSTEM_TABLE;
                                                          2010,5        91%
```

EFI_BOOT_SERVICES

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/edk2

```
///
typedef struct {
  ///
  /// The table header for the EFI Boot Services Table.
  ///
  EFI_TABLE_HEADER                Hdr;

  //
  // Task Priority Services
  //
  EFI_RAISE_TPL                   RaiseTPL;
  EFI_RESTORE_TPL                 RestoreTPL;

  //
  // Memory Services
  //
  EFI_ALLOCATE_PAGES              AllocatePages;
  EFI_FREE_PAGES                  FreePages;
  EFI_GET_MEMORY_MAP              GetMemoryMap;
  EFI_ALLOCATE_POOL               AllocatePool;
  EFI_FREE_POOL                   FreePool;

  //
  // Event & Timer Services
  //
  EFI_CREATE_EVENT                  CreateEvent;
  EFI_SET_TIMER                     SetTimer;
  EFI_WAIT_FOR_EVENT                WaitForEvent;
  EFI_SIGNAL_EVENT                  SignalEvent;
  EFI_CLOSE_EVENT                   CloseEvent;
  EFI_CHECK_EVENT                   CheckEvent;

  //
  // Protocol Handler Services
  //
  EFI_INSTALL_PROTOCOL_INTERFACE    InstallProtocolInterface;
  EFI_REINSTALL_PROTOCOL_INTERFACE  ReinstallProtocolInterface;
  EFI_UNINSTALL_PROTOCOL_INTERFACE  UninstallProtocolInterface;
  EFI_HANDLE_PROTOCOL               HandleProtocol;
  VOID                              *Reserved;
  EFI_REGISTER_PROTOCOL_NOTIFY      RegisterProtocolNotify;
  EFI_LOCATE_HANDLE                 LocateHandle;
  EFI_LOCATE_DEVICE_PATH            LocateDevicePath;
  EFI_INSTALL_CONFIGURATION_TABLE   InstallConfigurationTable;

  //
  // Image Services
  //
  EFI_IMAGE_LOAD                    LoadImage;
  EFI_IMAGE_START                   StartImage;
  EFI_EXIT                          Exit;
  EFI_IMAGE_UNLOAD                  UnloadImage;
  EFI_EXIT_BOOT_SERVICES            ExitBootServices;

  //
  // Miscellaneous Services
  //
  EFI_GET_NEXT_MONOTONIC_COUNT      GetNextMonotonicCount;
  EFI_STALL                         Stall;
                                                          1885,3        85%
```

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/edk2

```
  EFI_UNINSTALL_PROTOCOL_INTERFACE  UninstallProtocolInterface;
  EFI_HANDLE_PROTOCOL               HandleProtocol;
  VOID                              *Reserved;
  EFI_REGISTER_PROTOCOL_NOTIFY      RegisterProtocolNotify;
  EFI_LOCATE_HANDLE                 LocateHandle;
  EFI_LOCATE_DEVICE_PATH            LocateDevicePath;
  EFI_INSTALL_CONFIGURATION_TABLE   InstallConfigurationTable;

  //
  // Image Services
  //
  EFI_IMAGE_LOAD                    LoadImage;
  EFI_IMAGE_START                   StartImage;
  EFI_EXIT                          Exit;
  EFI_IMAGE_UNLOAD                  UnloadImage;
  EFI_EXIT_BOOT_SERVICES            ExitBootServices;

  //
  // Miscellaneous Services
  //
  EFI_GET_NEXT_MONOTONIC_COUNT      GetNextMonotonicCount;
  EFI_STALL                         Stall;
  EFI_SET_WATCHDOG_TIMER            SetWatchdogTimer;

  //
  // DriverSupport Services
  //
  EFI_CONNECT_CONTROLLER            ConnectController;
  EFI_DISCONNECT_CONTROLLER         DisconnectController;

  //
  // Open and Close Protocol Services
  //
  EFI_OPEN_PROTOCOL                 OpenProtocol;
  EFI_CLOSE_PROTOCOL                CloseProtocol;
  EFI_OPEN_PROTOCOL_INFORMATION     OpenProtocolInformation;

  //
  // Library Services
  //
  EFI_PROTOCOLS_PER_HANDLE          ProtocolsPerHandle;
  EFI_LOCATE_HANDLE_BUFFER          LocateHandleBuffer;
  EFI_LOCATE_PROTOCOL               LocateProtocol;
  EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES    InstallMultipleProtocolInterfaces;
  EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES  UninstallMultipleProtocolInterfaces;

  //
  // 32-bit CRC Services
  //
  EFI_CALCULATE_CRC32               CalculateCrc32;

  //
  // Miscellaneous Services
  //
  EFI_COPY_MEM                      CopyMem;
  EFI_SET_MEM                       SetMem;
  EFI_CREATE_EVENT_EX               CreateEventEx;
} EFI_BOOT_SERVICES;
                                                          1882,3        87%
```

UEFI Example Program

```
#include <Uefi/UefiSpec.h>

EFI_STATUS EFIAPI
efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable) {
	SystemTable->ConOut->OutputString(SystemTable->ConOut, L"Hello World!\n");
	return EFI_SUCCESS;
}
```

UEFI Example Program

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/edk2

```
  /// The number of modes supported by QueryMode () and SetMode ().
  ///
  INT32   MaxMode;

  //
  // current settings
  //

  ///
  /// The text mode of the output device(s).
  ///
  INT32   Mode;
  ///
  /// The current character output attribute.
  ///
  INT32   Attribute;
  ///
  /// The cursor's column.
  ///
  INT32   CursorColumn;
  ///
  /// The cursor's row.
  ///
  INT32   CursorRow;
  ///
  /// The cursor is currently visbile or not.
  ///
  BOOLEAN CursorVisible;
} EFI_SIMPLE_TEXT_OUTPUT_MODE;

///
/// The SIMPLE_TEXT_OUTPUT protocol is used to control text-based output devices.
/// It is the minimum required protocol for any handle supplied as the ConsoleOut
/// or StandardError device. In addition, the minimum supported text mode of such
/// devices is at least 80 x 25 characters.
///
struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
  EFI_TEXT_RESET                Reset;

  EFI_TEXT_STRING               OutputString;
  EFI_TEXT_TEST_STRING          TestString;

  EFI_TEXT_QUERY_MODE           QueryMode;
  EFI_TEXT_SET_MODE             SetMode;
  EFI_TEXT_SET_ATTRIBUTE        SetAttribute;

  EFI_TEXT_CLEAR_SCREEN         ClearScreen;
  EFI_TEXT_SET_CURSOR_POSITION  SetCursorPosition;
  EFI_TEXT_ENABLE_CURSOR        EnableCursor;

  ///
  /// Pointer to SIMPLE_TEXT_OUTPUT_MODE data.
  ///
  EFI_SIMPLE_TEXT_OUTPUT_MODE   *Mode;
};

extern EFI_GUID gEfiSimpleTextOutProtocolGuid;

#endif
                                                                409,1         Bot
```

ruslan@ruslan-ThinkPad-T470p: ~/examples/shim/edk2

```
**/
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_RESET)(
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL        *This,
  IN BOOLEAN                                ExtendedVerification
  );

/**
  Write a string to the output device.

  @param  This   The protocol instance pointer.
  @param  String The NULL-terminated string to be displayed on the output
                 device(s). All output devices must also support the Unicode
                 drawing character codes defined in this file.

  @retval EFI_SUCCESS             The string was output to the device.
  @retval EFI_DEVICE_ERROR        The device reported an error while attempting to o
utput
                                  the text.
  @retval EFI_UNSUPPORTED         The output device's mode is not currently in a
                                  defined text mode.
  @retval EFI_WARN_UNKNOWN_GLYPH  This warning code indicates that some of the
                                  characters in the string could not be
                                  rendered and were skipped.

**/
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_STRING)(
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL        *This,
  IN CHAR16                                 *String
  );

/**
  Verifies that all characters in a string can be output to the
  target device.

  @param  This   The protocol instance pointer.
  @param  String The NULL-terminated string to be examined for the output
                 device(s).

  @retval EFI_SUCCESS      The device(s) are capable of rendering the output string.
  @retval EFI_UNSUPPORTED  Some of the characters in the string cannot be
                           rendered by one or more of the output devices mapped
                           by the EFI handle.

**/
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_TEST_STRING)(
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL        *This,
  IN CHAR16                                 *String
  );

/**
  Returns information for an available text mode that the output device(s)
  supports.

search hit BOTTOM, continuing at TOP                            171,0-1       46%
```

Toolkits

- We are going to use clang and lld-link
  - sudo apt-get install clang lld
  - clang/lld 7.0.0+ should be OK
- Why clang?
  - UEFI requires to compile PE DLL executables (but they do not use anything from Windows)
  - Clang, unlike gcc, provides both ELF and PE support by default
  - Likewise, lld can link PE in addition to ELF

Toolkits

- In addition, you need to download and compile a tool which converts DLL to EFI
  - Use my version which I ported from Windows to Linux
    - git clone https://github.com/rusnikola/fwimage
  - Run ‘make’
    - It will generate the ‘fwimage’ program

Toolkits

- LLD is a tricky program which behaves differently depending on the program file name
  - “lld.link” for ELF and “lld-link” for PE
  - Ubuntu (and possibly other distributions) does not create lld-link to lld for some reason
  - Do the following (after you install lld)

```
$ cd /usr/bin
$ sudo ln -s lld lld-link
```

[will create lld-link if it does not exist already]

Toolkits

- Download Intel Tianocore/Edk2 headers
  - Create “uefi_projects” directory and make it current
  - wget https://github.com/tianocore/edk2/archive/refs/tags/edk2-stable202208.tar.gz
  - tar xvpf ./edk2-stable202208.tar.gz

    CORRECTION: command to unpack
  - mv edk2-edk2-stable202208/MdePkg/Include ./
  - rm -r edk2-edk2-stable202208
- “Include” in your current directory will have all necessary Edk2 headers

UEFI Example Program

Create hello.c in “uefi_projects”:

```
#include <Uefi.h>

EFI_STATUS EFIAPI
efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable) {
	SystemTable->ConOut->OutputString(SystemTable->ConOut, L"Hello World!\r\n");
	SystemTable->BootServices->Stall(5 * 1000000); // 5 seconds
	return EFI_SUCCESS;
}
```

What is \r\n?

It is a Windows/DOS style for new lines (also adopted by UEFI):
\r (carriage return, CR)
\n (line feed, LF)

Linux uses \n, macOS uses \r, Windows/DOS uses both

UEFI Example Program

Compile:
clang -m64 -O2 -fshort-wchar -I ./Include -I ./Include/X64 -mcmodel=small -mno-red-zone -mno-stack-arg-probe -target x86_64-pc-mingw32 -c hello.c

Link:
lld-link /dll /nodefaultlib /safeseh:no /machine:AMD64 /entry:efi_main hello.o /out:hello.dll

Convert DLL to EFI:
./fwimage/fwimage app hello.dll hello.efi

CORRECTION: there are some problems with ISO9660 when using recent versions of VirtualBox, so we will use FAT!

Create a bootable FAT image (10 MB):
mkdir ./uefi_fat_mnt
dd if=/dev/zero of=boot.img bs=1M count=10
mkfs.vfat ./boot.img
sudo mount -o loop ./boot.img ./uefi_fat_mnt
sudo mkdir -p ./uefi_fat_mnt/EFI/BOOT
sudo cp hello.efi uefi_fat_mnt/EFI/BOOT/BOOTx64.EFI
sudo umount ./uefi_fat_mnt

UEFI Example Program

Instructions for VirtualBox:
Download boot.vmdk (wrapper for boot.img) from Canvas (Modules->VirtualBox Files) and place it in the same directory as boot.img!

Note: If you have multiple copies of boot.vmdk, please change ddb.uuid.image=... in each ‘boot.vmdk’ copy:
Install UUID: sudo apt-get install uuid
Run ‘uuid’ and copy a unique UUID string to boot.vmdk

Instructions for qemu:
Download bios.zip (UEFI image for qemu) from Canvas (Modules->Qemu Files) and unpack it in the same directory as boot.img:
unzip ./bios.zip

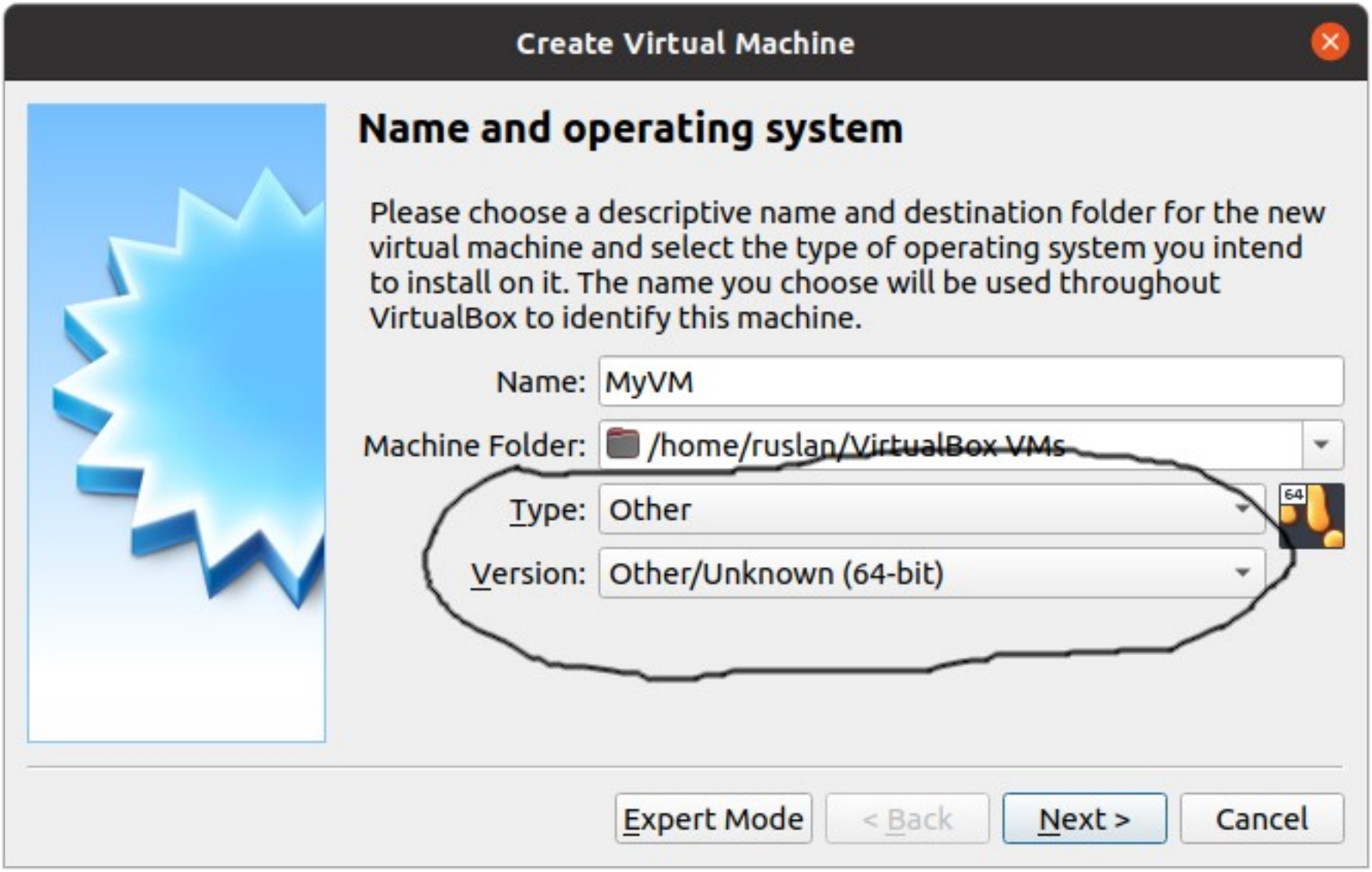
What are those parameters?

- -fshort-wchar to use UTF-16 (2-byte) wide chars, by default Linux assumes UTF-32 (4-byte) wide chars
  - Note that we have L”Hello world!” rather than “Hello world!” since UEFI uses UTF-16, not UTF-8!
- -target x86_64-pc-mingw32 to use Windows executable format rather than ELF
- -I ./Include -I ./Include/X64 to use headers from your “Include” and “Include/X64” (architecture-specific) directories
- Other parameters to disable default system libraries, UEFI-incompatible options, etc

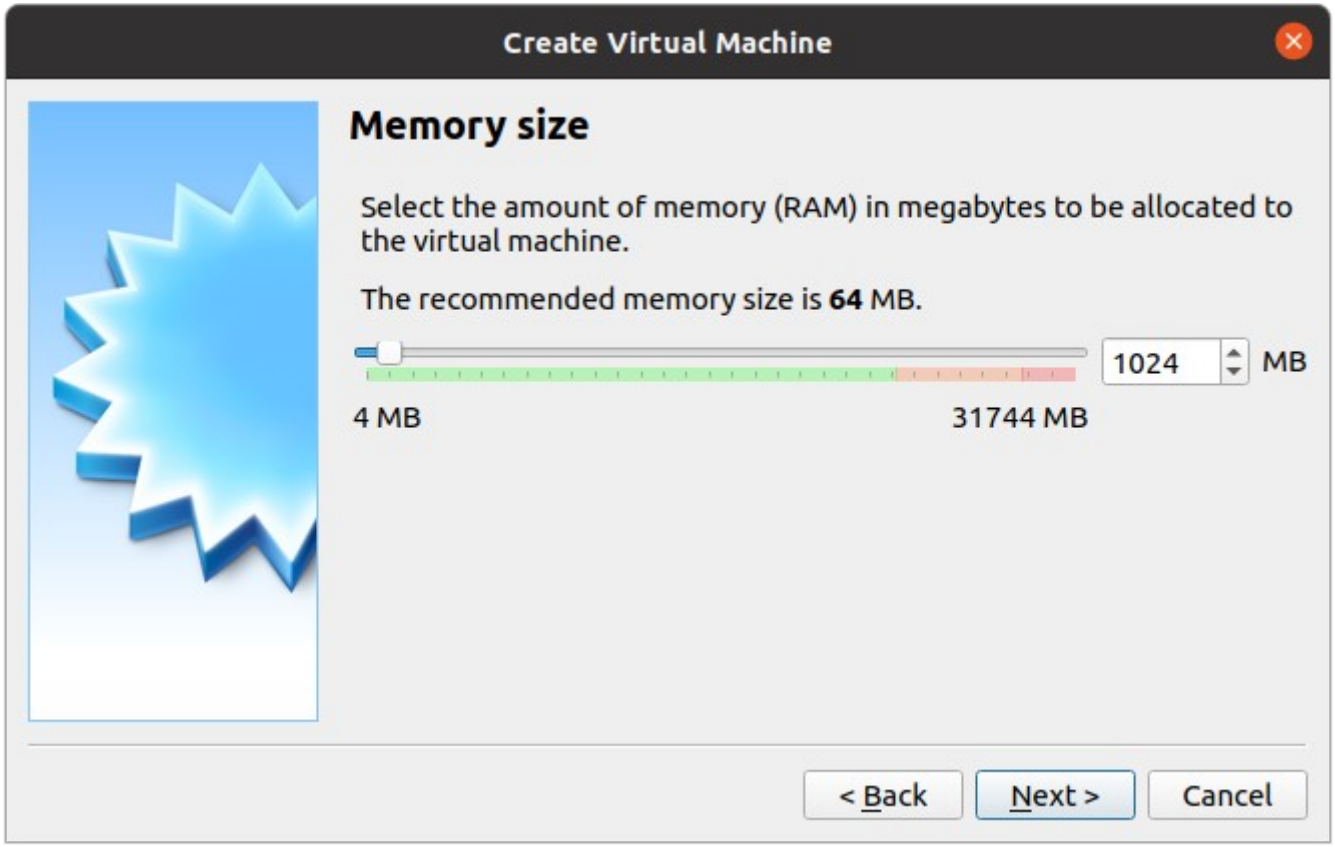
Why does UEFI use UTF-16?

- Probably due to Windows executables and because EFI was developed in the late 90s
- Legacy encodings only required 1-byte characters
- Unicode initially used 2-byte characters
  - Windows adopted Unicode (UTF-16)
  - Unicode later expanded beyond 64K characters, so we really need UTF-32 now. UTF-16 may now require “surrogate pairs.”
  - Why not 1-byte pairs then? This is what UTF-8 does. Linux adopted UTF-8 for simplicity.

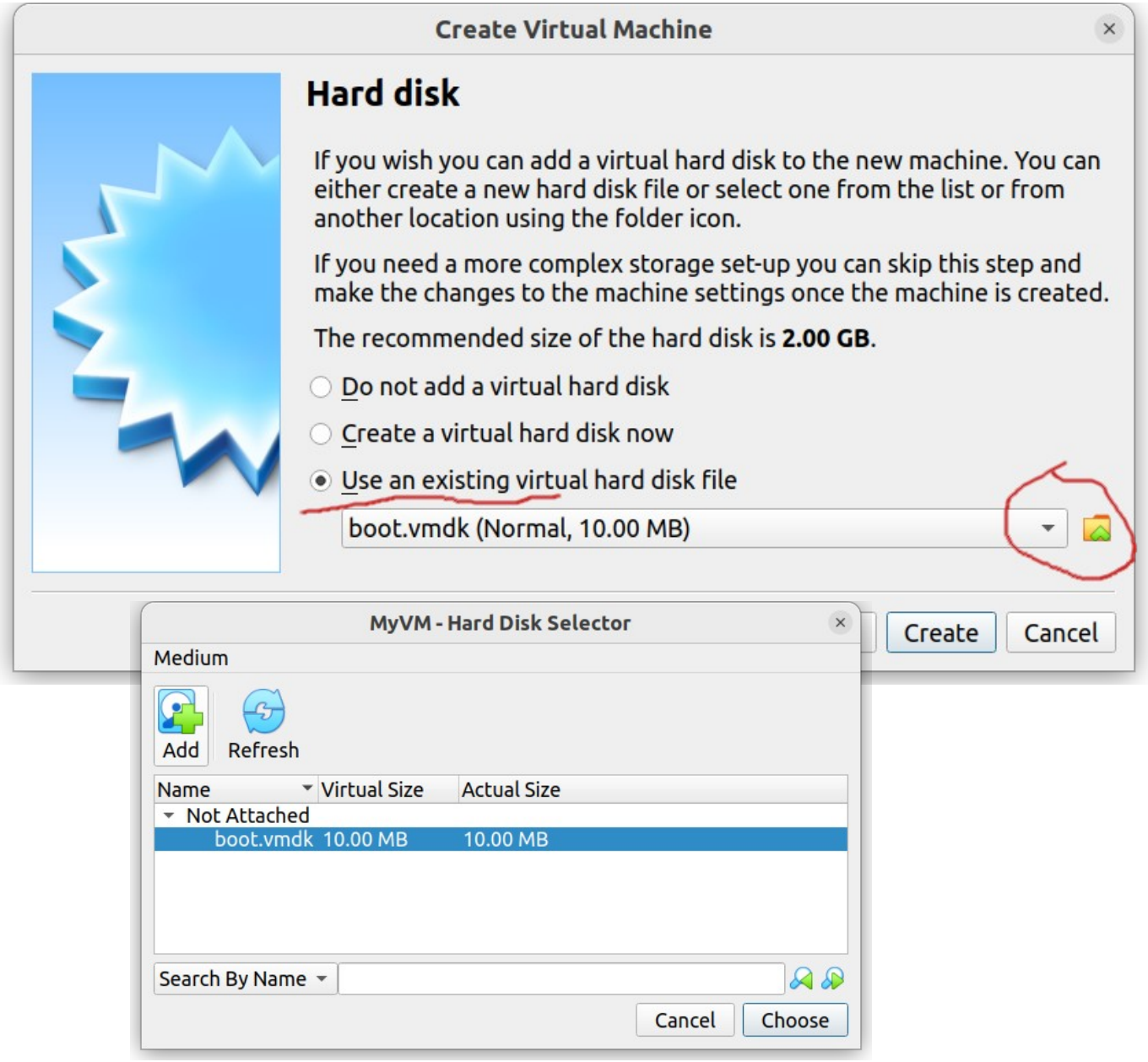
VirtualBox Setup



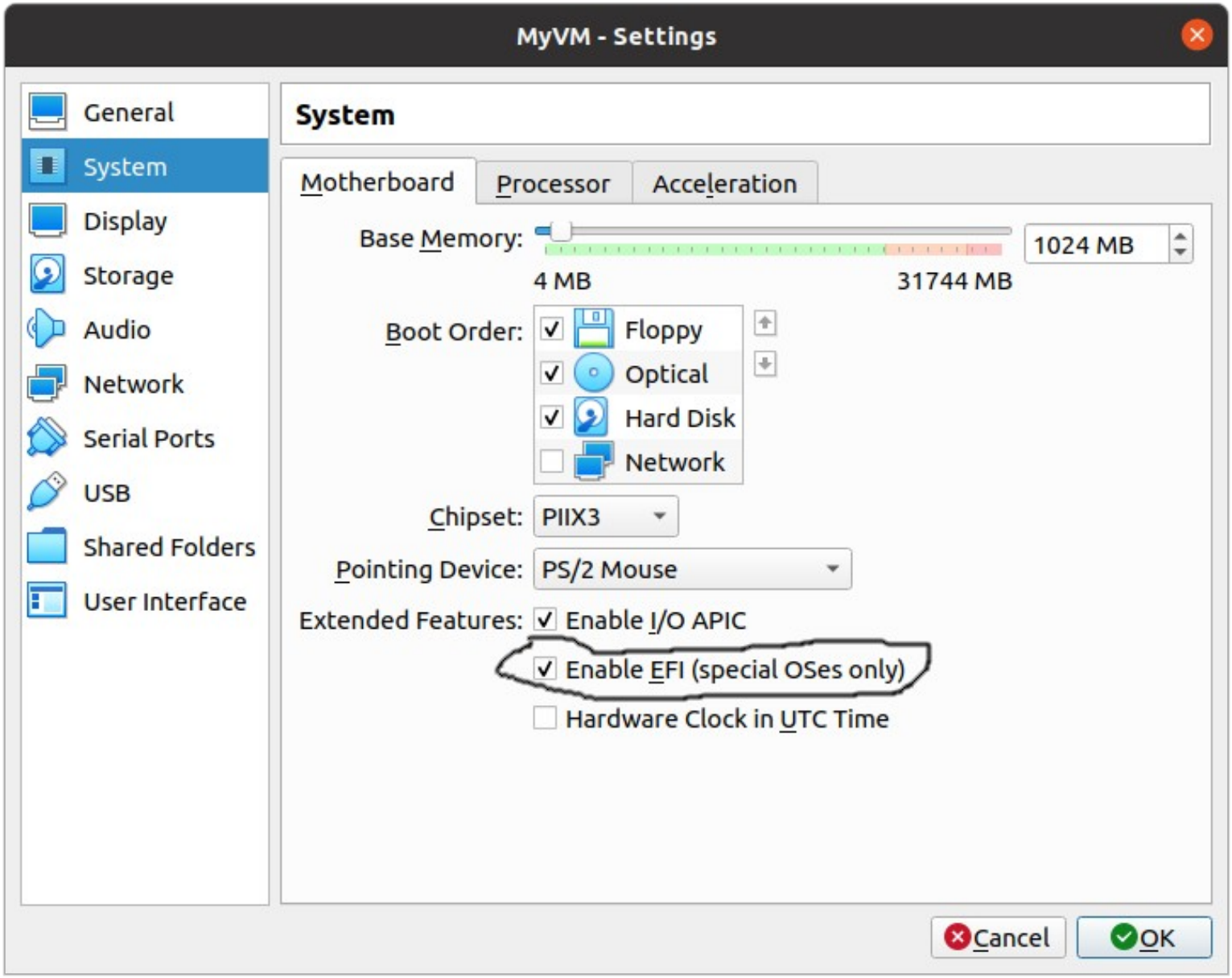
VirtualBox Setup



VirtualBox Setup



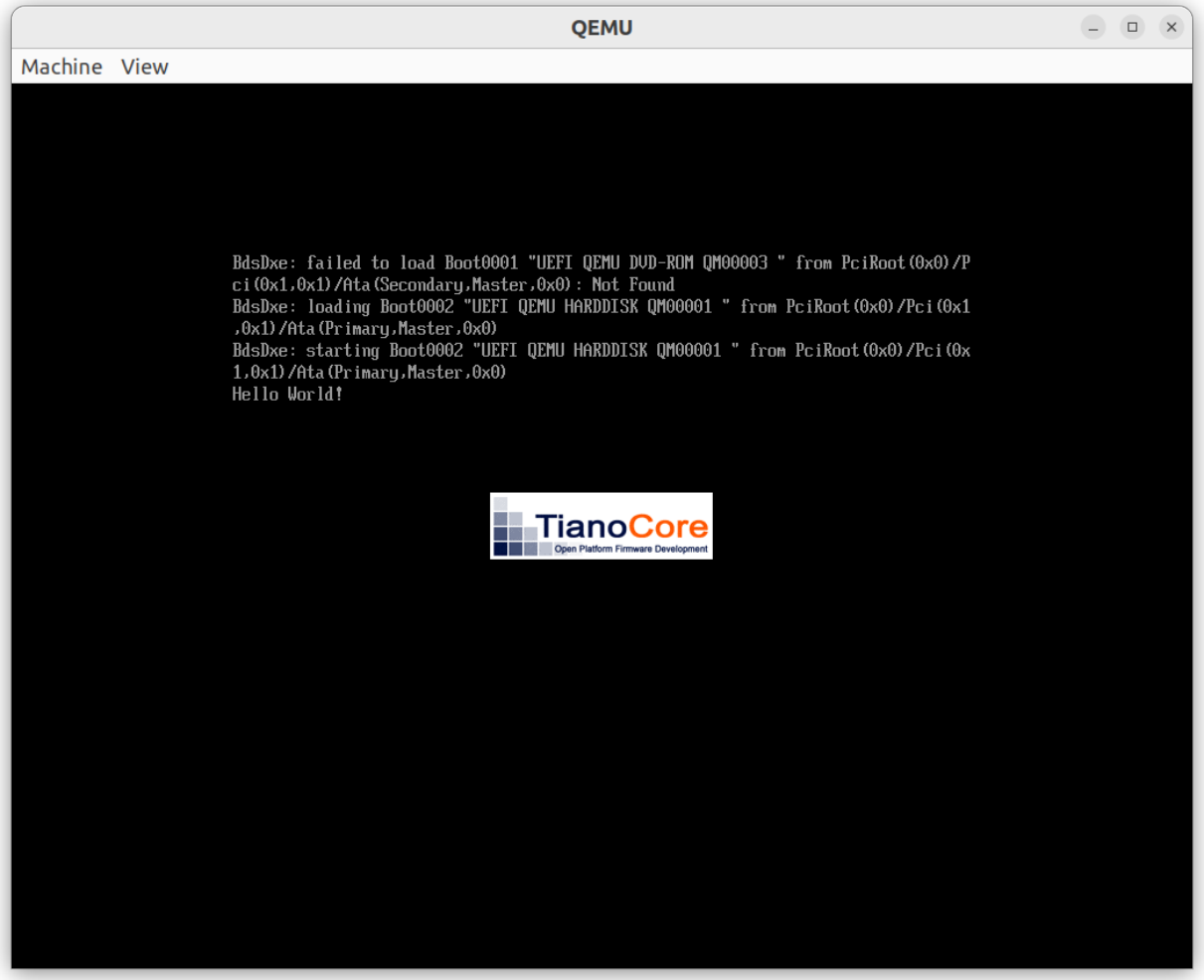
VirtualBox Setup



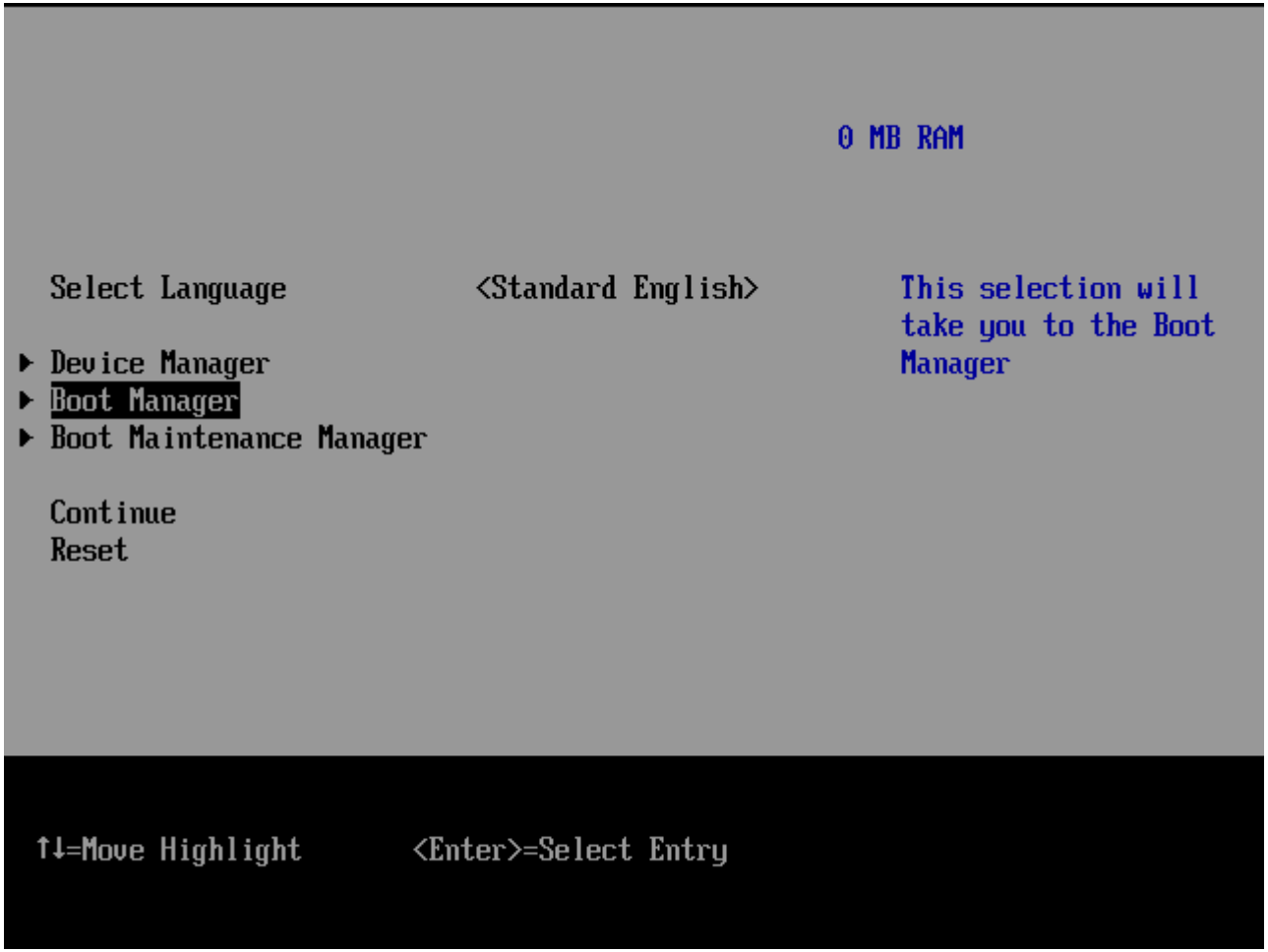
VirtualBox Setup: Execute

qemu: Execute

Run the following command:
qemu-system-x86_64 -bios /path/to/bios.bin -m 1024 -drive format=raw,file=/path/to/boot.img



UEFI Example Program: Complete



UEFI Example Program: Shell

UEFI Example Program: Shell

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
      FS0: Alias(s):F0c::BLK0:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell>
Shell>
Shell>
Shell> FS0:
FS0:\> ls
Directory of: FS0:\
00/00/0000  00:00 <DIR>        2,048  EFI
          0 File(s)          0 bytes
          1 Dir(s)
FS0:\> ls EFI\BOOT
Directory of: FS0:\EFI\BOOT\
00/00/0000  00:00              2,560  BOOTX64.EFI
          1 File(s)       2,560 bytes
          0 Dir(s)
FS0:\> EFI\BOOT\BOOTX64.EFI
Hello World!
FS0:\> _
```

UEFI Documentation

- Go to https://www.uefi.org/specifications/
  - Download UEFI Specification Version 2.9 (released March 2021)
- There also used to be convenient Phoenix (BIOS/UEFI vendor) documentation
  - Still accessible through https://web.archive.org/web/20181012151104/http://wiki.phoenix.com/wiki/index.php/Category:UEFI_2.0

EFI_SYSTEM_TABLE and EFI_BOOT_SERVICES

- The EFI_SYSTEM_TABLE parameter to “efi_main” is the topmost data structure passed by the UEFI firmware which contains pointers to many other structures
- EFI_BOOT_SERVICES deserves to be saved separately
  - Declare a global variable EFI_BOOT_SERVICES *BootServices;
  - BootServices = SystemTable->BootServices;

Assignment 1: UEFI Memory Allocation

```
static VOID *AllocatePool(UINTN size)
{
    VOID *ptr;
    EFI_STATUS ret = BootServices->AllocatePool(EfiBootServicesData, size, &ptr);
    if (EFI_ERROR(ret))
        return NULL;
    return ptr;
}

static VOID FreePool(VOID *buf)
{
    BootServices->FreePool(buf);
}
```

Assignment 1: UEFI Memory Allocation

```
static VOID *AllocatePool(UINTN size)
{
    VOID *ptr;
    EFI_STATUS ret = BootServices->AllocatePool(EfiBootServicesData, size, &ptr);
    if (EFI_ERROR(ret))
        return NULL;
    return ptr;
}

static VOID FreePool(VOID *buf)
{
    BootServices->FreePool(buf);
}
```

- You need to select the allocation type, for this example we assume memory for Boot Services (EfiBootServicesData)

Assignment 1: EFI_LOADED_IMAGE_PROTOCOL_GUID

```
#include <Protocol/LoadedImage.h>

EFI_GUID gEfiLoadedImageProtocolGuid = EFI_LOADED_IMAGE_PROTOCOL_GUID;
```

Get LoadedImage Protocol:

```
EFI_LOADED_IMAGE *li = NULL;
EFI_STATUS efi_status;

efi_status = BootServices->HandleProtocol(ImageHandle, &gEfiLoadedImageProtocolGuid,
                    (void **)&li);
if (EFI_ERROR(efi_status)) {
    SystemTable->ConOut->OutputString(SystemTable->ConOut,
        L"Cannot get LoadedImage for BOOTx64.EFI\r\n”);
    return efi_status;
}
```

Assignment 1: EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

Get SimpleFileSystem Protocol:

```
#include <Protocol/SimpleFileSystem.h>

EFI_GUID gEfiSimpleFileSystemProtocolGuid =
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID;

EFI_FILE_IO_INTERFACE *fio = NULL;

efi_status = BootServices->HandleProtocol(li->DeviceHandle,
                &gEfiSimpleFileSystemProtocolGuid,
                (void **) &fio);
if (EFI_ERROR(efi_status)) {
   SystemTable->ConOut->OutputString(SystemTable->ConOut, L"Cannot get fio\r\n");
   return efi_status;
}
```

Assignment 1: EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

Get Volume and File Handles:

```
EFI_FILE_PROTOCOL *vh = NULL;
EFI_FILE_PROTOCOL *fh = NULL;

efi_status = fio->OpenVolume(fio, &vh);
if (EFI_ERROR(efi_status)) {
    SystemTable->ConOut->OutputString(SystemTable->ConOut, L"Cannot get vh!\r\n");
    return efi_status;
}

efi_status = vh->Open(vh, &fh, L"\\EFI\\BOOT\\KERNEL”,
                EFI_FILE_MODE_READ, 0);
if (EFI_ERROR(efi_status)) {
    SystemTable->ConOut->OutputString(SystemTable->ConOut, L"Cannot get fh!\r\n");
    return efi_status;
}
```

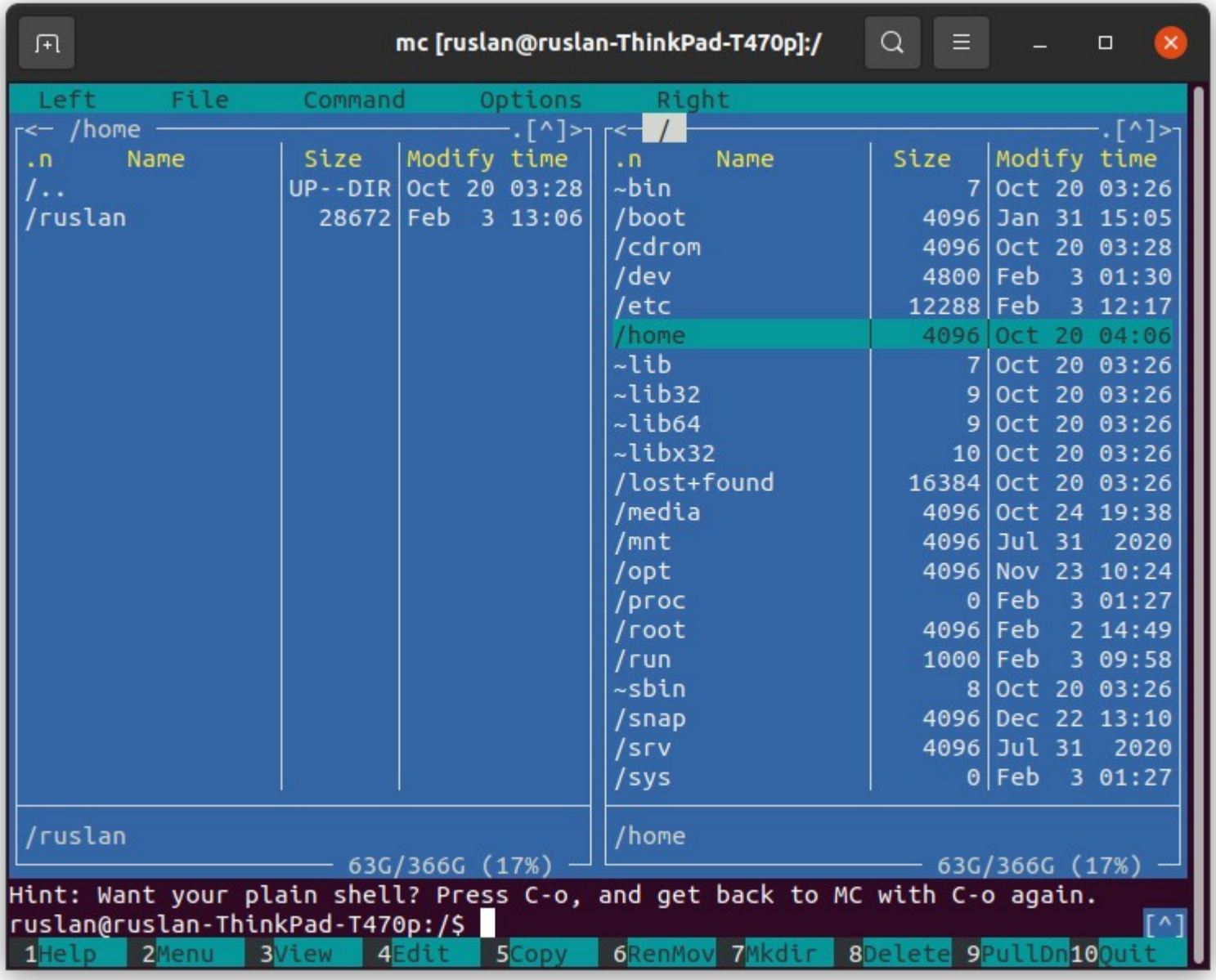
Assignment 1: EFI_SYSTEM_TABLE and EFI_BOOT_SERVICES

- Take a look at EFI_FILE_PROTOCOL
  - Specifically, Read, Write, Close methods, etc
  - Read the kernel file (what memory type do you need to use for the buffer?)
- Take a look at EFI_BOOT_SERVICES (BootServices)
  - ExitBootServices()
  - GetMemoryMap()
  - Jump to the kernel
    - Use an ELF position-independent executable (PIE) and call the entry point function

Assignment 1: Video Framebuffer

- Text mode vs. Graphics Mode
  - Video adapters and/or firmware can work in both modes
  - For VGA adapters, 8-bit ASCII fonts can loaded for 80x25 (standard) or other text modes
  - Colors are supported in both modes
- Text mode manipulates entire **characters**, e.g., 80x25
- Graphics mode manipulates **pixels** (tiny dots on the screen), e.g., 640x480, 800x600, 1024x768, ...

Example: Text mode (TUI)

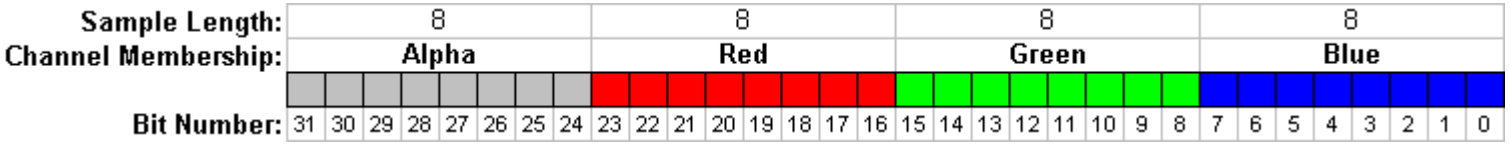


Assignment 1: Video Framebuffer

- Text mode can be emulated in graphics mode
  - Load some font and render its pixels manually
  - Used in Linux for the ”framebuffer console”
  - “Terminal” in GNOME or KDE (Linux)
- Examples and flexibility:
  - OutputString(L”...”) in UEFI is text mode (UTF-16)
  - If we render pixels ourselves, we are not restricted by any specific encoding or font

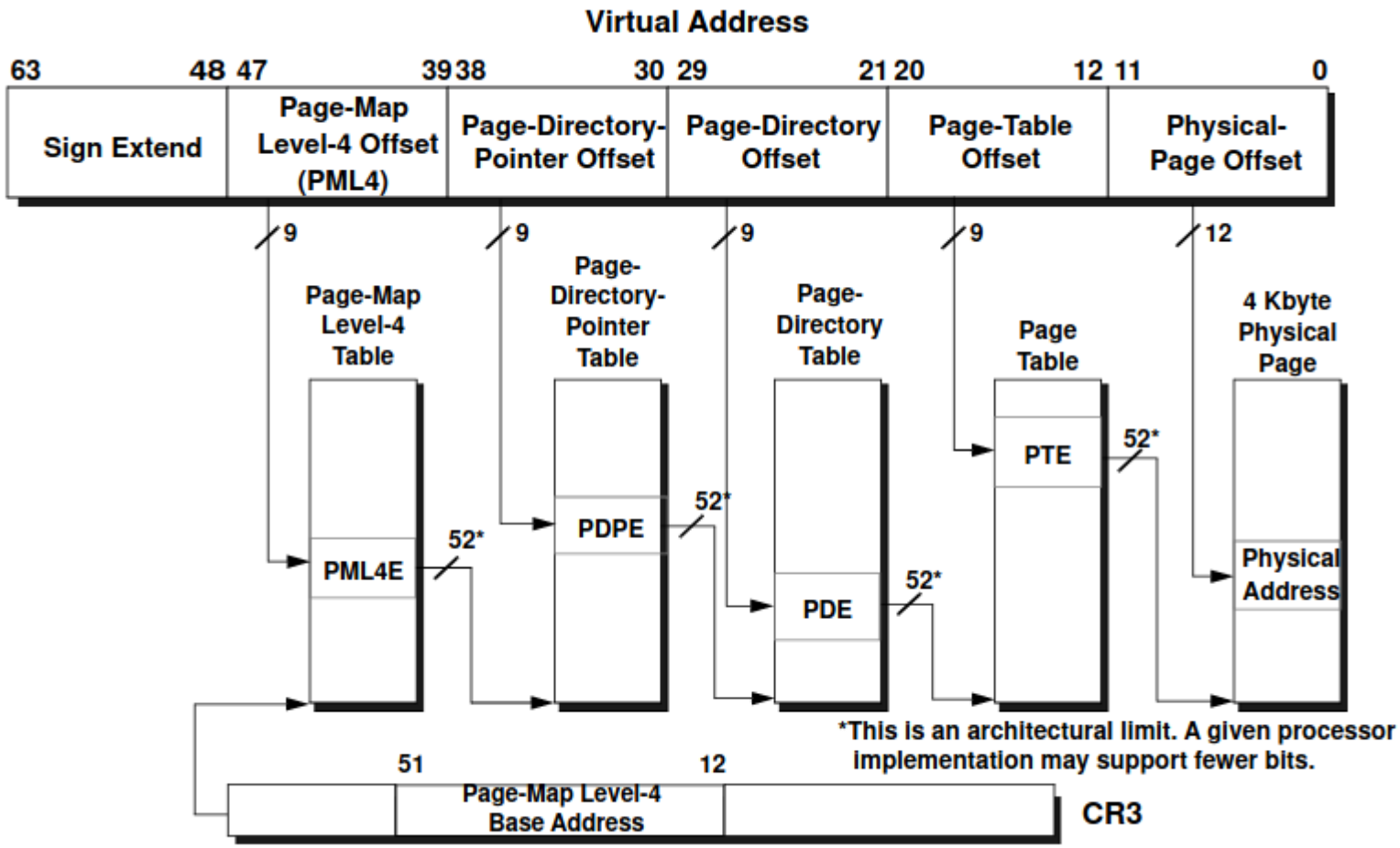
Assignment 1: Video Framebuffer

- Once you switch to graphics mode, you should not be using OutputString() anymore
- Draw some figures using pixels
  - Each pixel is a 32-bit (unsigned int) integer
    - 0xZZZZZZZZ, 4 bytes (BGRA), 2 hexadecimal digits for each color component



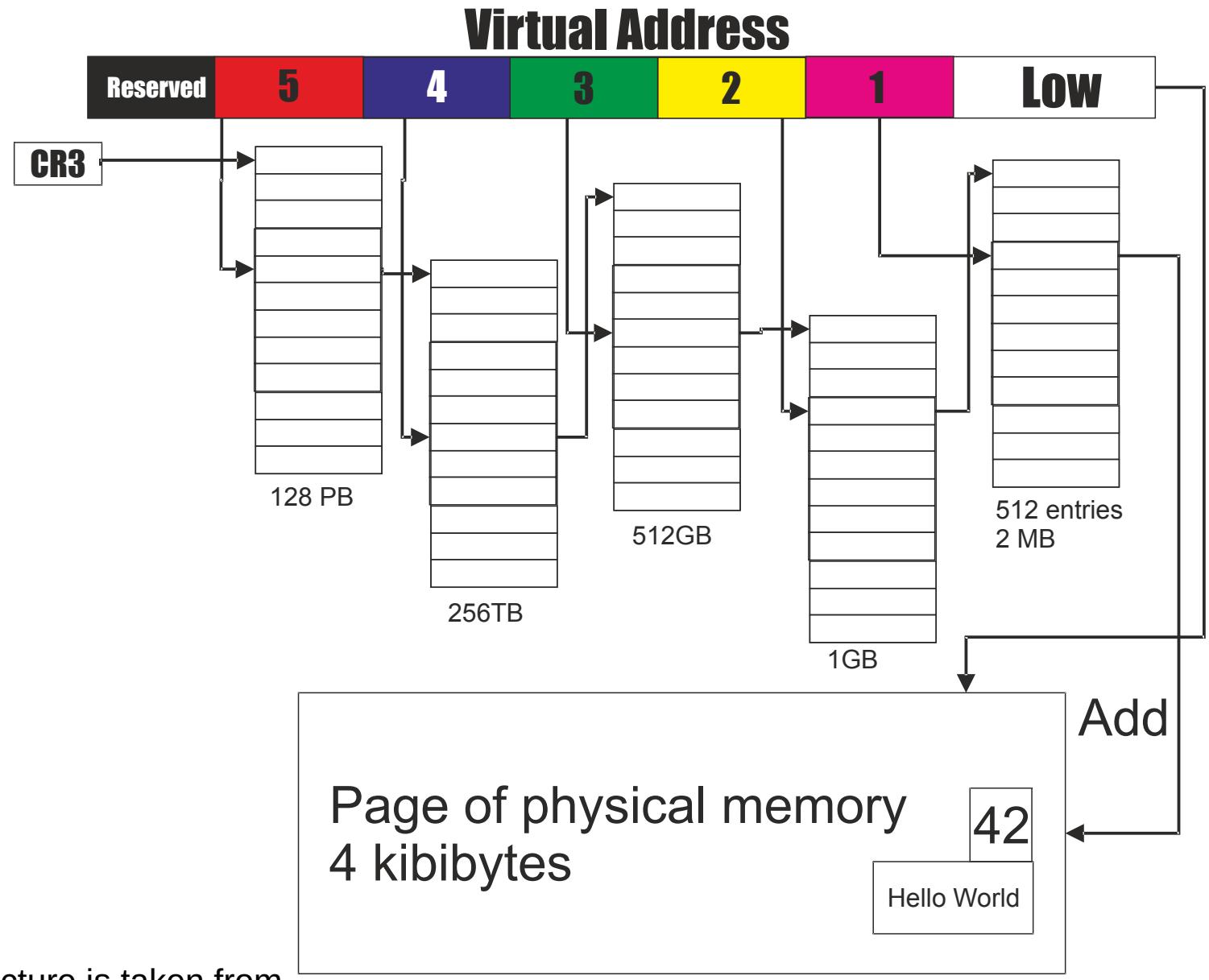
* The picture is taken from https://en.wikipedia.org/wiki/RGBA_color_model

Traditional 48-bit paging (4 levels)



* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

Extension: 57-bit paging (5 levels)



* The picture is taken from https://en.wikipedia.org/wiki/Intel_5-level_paging

Page Table

```
cat /proc/cpuinfo
```

```
ruslan@ruslan-ThinkPad-T470p: ~
c cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdb
g fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer a
es xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_singl
e pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase
 tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap clflush
opt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_noti
fy hwp_act_window hwp_epp md_clear flush_l1d
vmx flags       : vnmi preemption_timer invvpid ept_x_only ept_ad ept_1gb flexpr
iority tsc_offset vtpr mtf vapic ept vpid unrestricted_guest ple shadow_vmcs pml
 ept_mode_based_exec
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs taa itlb_multihit srbds
bogomips        : 5799.77
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

ruslan@ruslan-ThinkPad-T470p:~$
```

Page Table

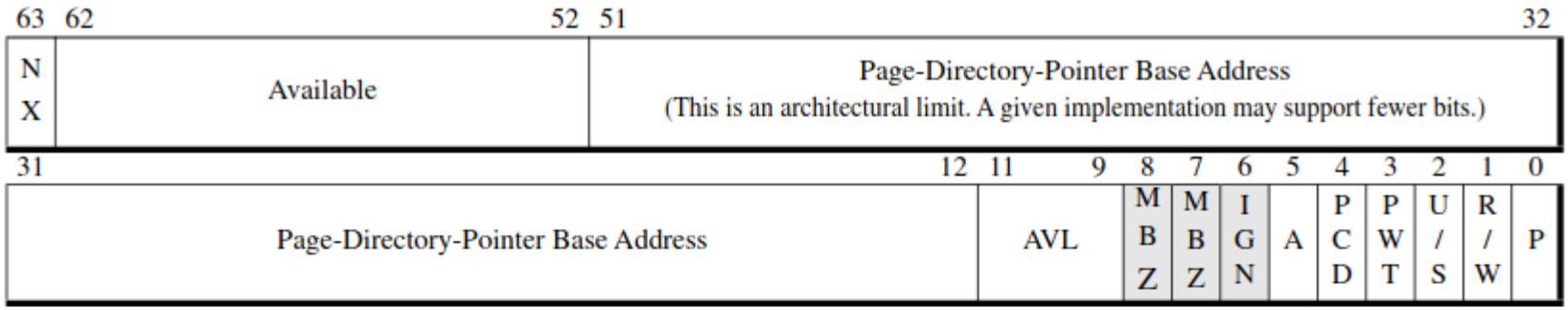


Figure 5-18. 4-Kbyte PML4E—Long Mode

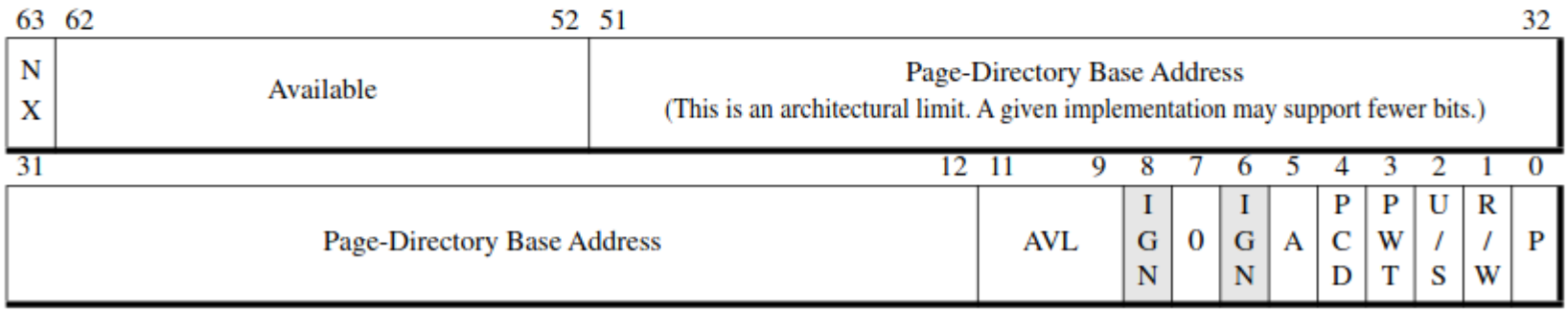


Figure 5-19. 4-Kbyte PDPE—Long Mode

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

Page Table

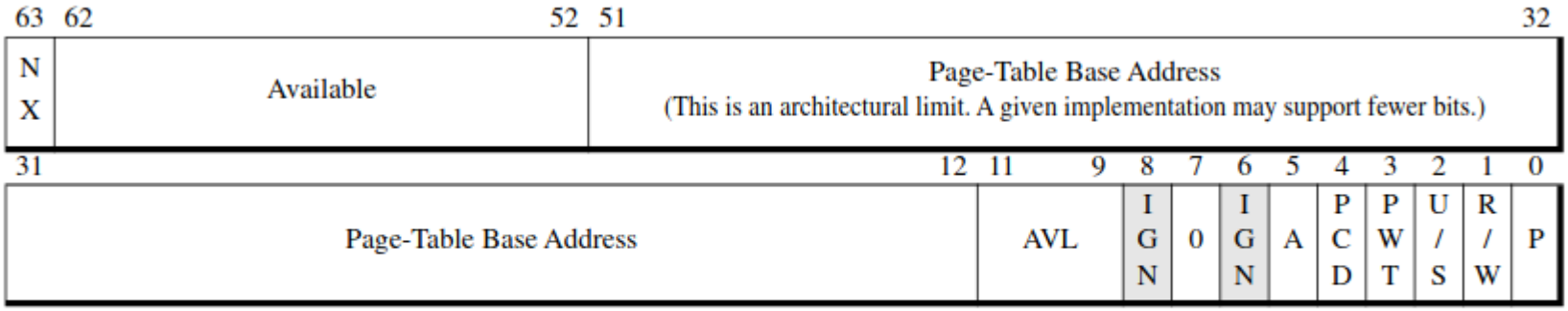


Figure 5-20. 4-Kbyte PDE—Long Mode

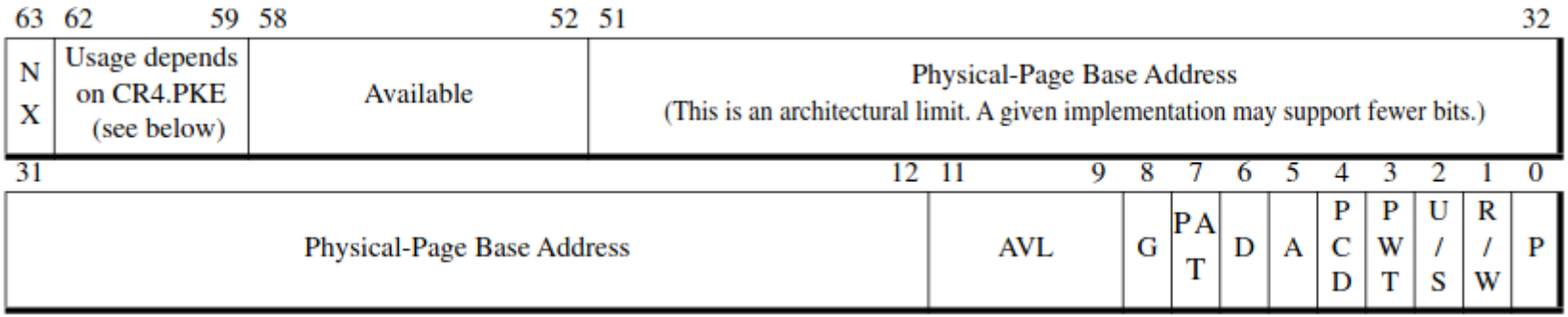


Figure 5-21. 4-Kbyte PTE—Long Mode

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

Example: PTE

```
typedef unsigned long long u64;

struct page_pte {
    u64 present:1;        // Bit P
    u64 writable:1;       // Bit R/W
    u64 user_mode:1;      // Bit U/S
    ...
    u64 page_address:40; // 40+12 = 52-bit physical address (max)
    U64 avail:7;          // reserved, should be 0
    u64 pke:4;            // no MPK/PKE, should be 0
    u64 nonexecute:1;
};
```

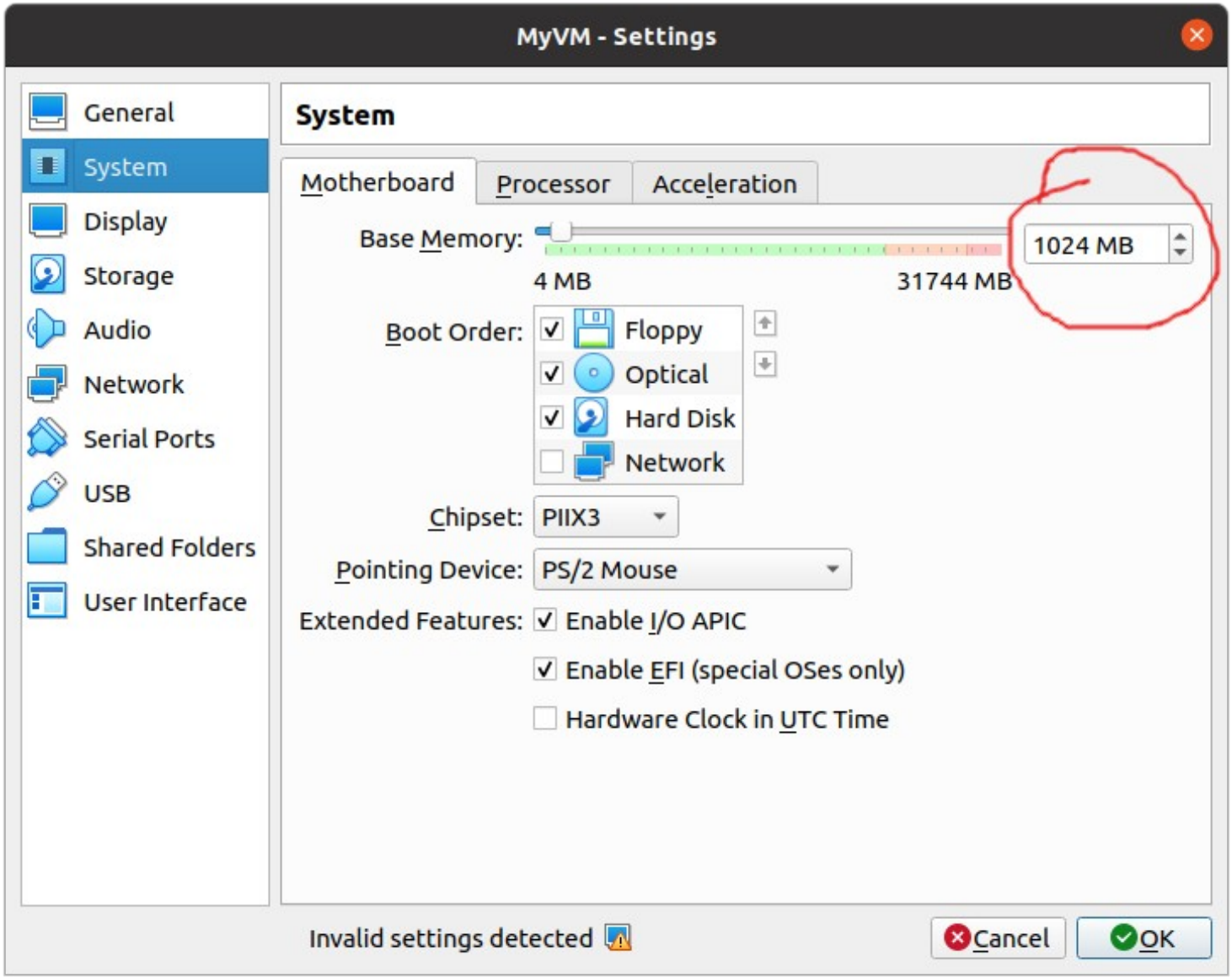
Example: PTE

```
typedef unsigned long long u64;

struct page_pte {
    u64 present:1;       // Bit P
    u64 writable:1;      // Bit R/W
    u64 user_mode:1;     // Bit U/S
    ...
    u64 page_address:40; // 40+12 = 52-bit physical address (max)
    U64 avail:7;         // reserved, should be 0
    u64 pke:4;           // no MPK/PKE, should be 0
    u64 nonexecute:1;
};

// _page_ address, i.e. memory_address / 4096 (or 2^12)
p->page_address = 0xZZZZZZZZZZULL; // ULL is unsigned long long
p->writable = 1;
p->present = 1;
p->user_mode = 0;
...
p->avail = 0;
p->pke = 0;
p->nonexecute = 0;
```

Example: Initializing 4 GB



1GB should be sufficient

Video RAM can be after that

Example: Initializing 4 GB

```
For 4GB, we reference 1048576 physical pages
Using 2048 pages for PTEs, 4 pages for PDEs, 1 for PDPE, 1 for PMLE4E
Total: 2054 pages = 8413184 bytes, align at the 4096 boundary!

PTE:
struct page_pte *p; … // Each entry is 8 bytes
For (int i = 0; i < 1048576; i++) { // 1048576/512 = 2048 pages
    p[i] = … // physical pages 0,1,2,3,…,1048575 (absolute address)
}
PDE:
struct page_pde *pd = (struct page_pde *) (p + 1048576);
for (int j = 0; j < 2048; j++) { // 2048/512 = 4 pages
    struct page_pte *start_pte = p + 512 * j;
    page_addr = (u64) start_pte >> 12; // we record the page address
    …
}
PDPE:
Reference 4 PDEs (1 page), everything else is empty
PMLE4E:
Just one reference to PDPE; everything else is empty
```

Aligning Pages

```
Allocate more space: e.g., 8413184 + 4095

Align the allocated ‘base’:
(void *) (((unsigned long long) base + 4095) & (~4095ULL))
```

- Why long long?
  - The kernel uses 64-bit ‘long’ due to System V’s ABI (aka the LP64 model)
  - The boot loader uses 32-bit ‘long’ due to EFI/Microsoft’s ABI (aka the LLP64 model)
  - ‘int’ is 32 bit and ‘long long’ is 64 bit in either case

Loading Page Table

```
void write_cr3(unsigned long long cr3_value)
{
    asm volatile ("mov %0, %%cr3"
                  :
                  : "r" (cr3_value)
                  : "memory");
}
```

Loading Page Table

```
void write_cr3(unsigned long long cr3_value)
{
    asm volatile ("mov %0, %%cr3"
                  :
                  : "r" (cr3_value)
                  : "memory");
}
```

63 – 52	51 – 32
Reserved, MBZ	Page-Map Level-4 Table Base Address (This is an architectural limit. A given implementation may support fewer bits.)

31 – 12	11 – 5	4	3	2	0
Page-Map Level-4 Table Base Address	Usage depends on state of Processor Context ID enablement (CR4.PCIDE). See below.				

No PCID (PCIDE=0)!

	11 – 5	4	3	2 – 0
CR4.PCIDE=0	Reserved	PCD	PWT	Reserved
CR4.PCIDE=1	Processor Context Identifier (See Section 5.5.1 "Process Context Identifier," on page 147 for more information.)			

Figure 3-6. Control Register 3 (CR3)—Long Mode

* The picture is taken from https://www.amd.com/system/files/TechDocs/24593.pdf

Loading Page Table

Page-Level Writethrough (PWT) Bit. Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

Page-Level Cache Disable (PCD) Bit. Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

```
PCD=0, PWT=0
```