CSE 511: Operating Systems Design

Lecture 7

Symmetric Multiprocessing (SMP)

UEFI Multiprocessing

Symmetric Multiprocessing (SMP)

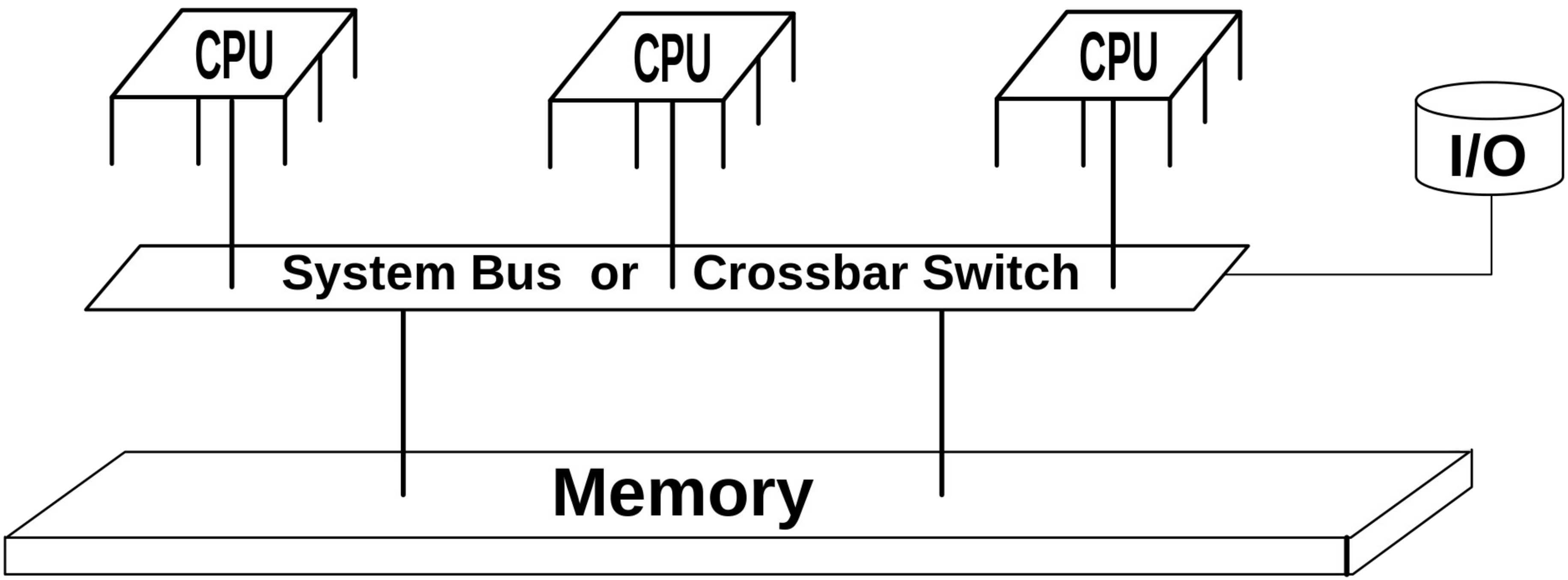
- For almost four decades (!), CPU clock speeds were *exponentially* growing but this growth has stopped around 15 years ago
  - Example: 166 MHz (1997), 733 MHz (2000)
  - After two decades, we have around 4 GHz
- While the number of transistors still continues to grow exponentially, we have to rely more and more on parallelism

Symmetric Multiprocessing (SMP)

- Multiple CPUs were already used to speed up performance on high-end systems
- In early 2000s, multi-core systems were introduced
  - Simultaneous Multithreading (Hyperthreading): share the same physical CPU for two logical CPUs, they appear as two “CPUs” to an OS
    - Floating point vs. integer instructions
  - True multi-core systems wherein the same CPU chip had multiple physical units (“cores”), they also appear as multiple “CPUs” to an OS

Symmetric Multiprocessing (SMP)

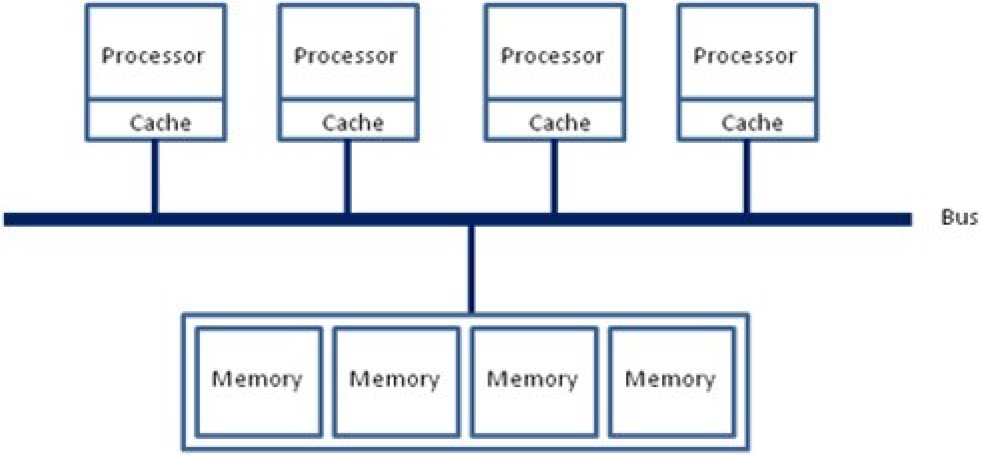
- In SMP, all CPUs are treated equally
  - Compare it with AMP (Asymmetric Multiprocessing)



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

Uniform Memory Access (UMA)

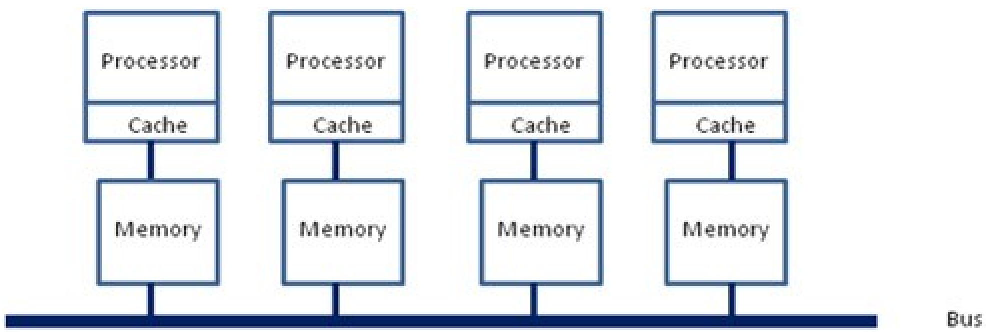
- Memory is shared
- Each CPU/core has its own non-shared cache
  - Sometimes the same cache can be shared by multiple cores in one physical CPU



* The picture is taken from https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html

Non-Uniform Memory Access (NUMA)

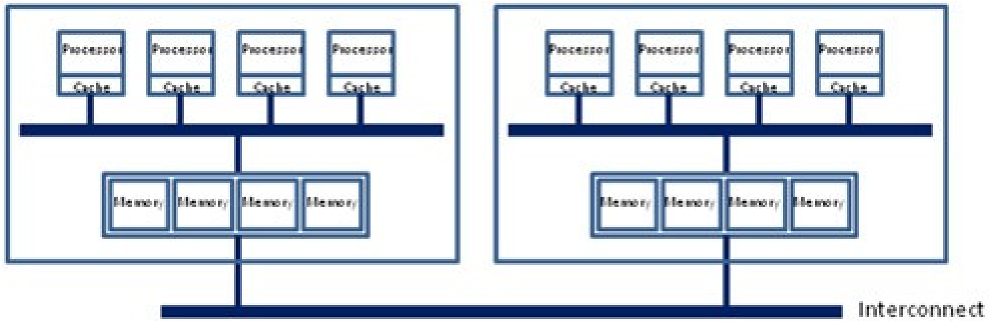
- Each CPU has its local memory which can be accessed faster



* The picture is taken from https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html

Modern Systems

- Combine these two models
  - Interconnect: Intel QPI (QuickPath), Intel UPI (UltraPath), AMD HyperTransport, etc



* The picture is taken from https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html

ccNUMA (cache coherent NUMA)

- Most existing systems are ccNUMA
  - It is much easier to program assuming that cache is coherent across different CPUs

NUMA Commands

- Install numctl: **`sudo apt-get install numactl`**

```
admin_@virginia:~$ numastat
                           node0           node1           node2           node3
numa_hit                10264807         9501974         9427693         9496624
numa_miss                      0               0               0               0
numa_foreign                   0               0               0               0
interleave_hit             17618           17728           17638           17733
local_node              10263586         9468107         9394014         9462638
other_node                  1221           33867           33679           33986

Mappings /proc/<pid>/numa_maps

admin_@virginia:~$ cat /proc/self/numa_maps
5564e2bbc000 default file=/bin/cat mapped=8 N2=8 kernelpagesize_kB=4
5564e2dc3000 default file=/bin/cat anon=1 dirty=1 N2=1 kernelpagesize_kB=4
5564e2dc4000 default file=/bin/cat anon=1 dirty=1 N2=1 kernelpagesize_kB=4
…
admin_@virginia:~$ cat /proc/self/numa_maps
563a633f7000 default file=/bin/cat mapped=8 N2=8 kernelpagesize_kB=4
563a635fe000 default file=/bin/cat anon=1 dirty=1 N3=1 kernelpagesize_kB=4
563a635ff000 default file=/bin/cat anon=1 dirty=1 N3=1 kernelpagesize_kB=4
…
```

NUMA Commands

```
admin_@virginia:~$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
node 0 size: 32038 MB
node 0 free: 30310 MB
node 1 cpus: 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
node 1 size: 32252 MB
node 1 free: 31650 MB
node 2 cpus: 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
node 2 size: 32252 MB
node 2 free: 31393 MB
node 3 cpus: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
node 3 size: 32251 MB
node 3 free: 30677 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

NUMA Commands

- Shows which CPUs/cores are in which NUMA groups (nodes), 4 nodes with 18 cores each
- Around 32 GB RAM per each node, 128 GB in total
- Relative “distances” between nodes, 10..254
  - 10 in the same node
  - 21 is 2.1 slower
  - Defined in ACPI (Advanced Configuration and Power Interface)

UEFI Multiprocessing

- It works but does not seem to be useful **after** ExitBootServices(); **it can change in the future**
  - In the OS kernel, all CPUs should still be started using legacy mechanisms (local APIC interrupt)
    - They start in the 16-bit “real” mode
    - After that you enable the 32-bit protected mode, paging, than the 64-bit (long) mode
- EFI_MP_SERVICES_PROTOCOL
  - UEFI Platform Initialization Specification Version 1.7 (Errata A)

UEFI Multiprocessing

```
#include <Uefi.h>
#include <Pi/PiMultiPhase.h>
#include <Protocol/MpService.h>

EFI_GUID gEfiMpServiceProtocolGuid = EFI_MP_SERVICES_PROTOCOL_GUID;
static EFI_SYSTEM_TABLE *SystemTable;
static EFI_BOOT_SERVICES *BootServices;

EFI_STATUS EFIAPI
efi_main(EFI_HANDLE imageHandle, EFI_SYSTEM_TABLE *systemTable)
{
	EFI_STATUS efi_status;
	SystemTable = systemTable;
	BootServices = systemTable->BootServices;

	EFI_MP_SERVICES_PROTOCOL *mps = NULL;
	efi_status = BootServices->LocateProtocol(&gEfiMpServiceProtocolGuid,
			NULL, (VOID **) &mps);
	if (EFI_ERROR(efi_status)) {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Cannot get the MP protocol!\r\n");
		return efi_status;
	}
```

UEFI Multiprocessing

```
#include <Uefi.h>
#include <Pi/PiMultiPhase.h>
#include <Protocol/MpService.h>

EFI_GUID gEfiMpServiceProtocolGuid = EFI_MP_SERVICES_PROTOCOL_GUID;
static EFI_SYSTEM_TABLE *SystemTable;
static EFI_BOOT_SERVICES *BootServices;

EFI_STATUS EFIAPI
efi_main(EFI_HANDLE imageHandle, EFI_SYSTEM_TABLE *systemTable)
{
	EFI_STATUS efi_status;
	SystemTable = systemTable;
	BootServices = systemTable->BootServices;

	EFI_MP_SERVICES_PROTOCOL *mps = NULL;
	efi_status = BootServices->LocateProtocol(&gEfiMpServiceProtocolGuid,
			NULL, (VOID **) &mps);
	if (EFI_ERROR(efi_status)) {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Cannot get the MP protocol!\r\n");
		return efi_status;
	}
```

UEFI Multiprocessing

```
UINTN total_cpus, enabled_cpus;
efi_status = mps->GetNumberOfProcessors(mps, &total_cpus, &enabled_cpus);
if (EFI_ERROR(efi_status)) {
	SystemTable->ConOut->OutputString(SystemTable->ConOut,
		L"Cannot get the number of processors!\r\n");
	return efi_status;
}
```

UEFI Multiprocessing

```
UINTN total_cpus, enabled_cpus;
efi_status = mps->GetNumberOfProcessors(mps, &total_cpus, &enabled_cpus);
```

EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()

Summary

This service retrieves the number of logical processor in the platform and the number of those logical processors that are currently enabled. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS) (
  IN  EFI_MP_SERVICES_PROTOCOL  *This,
  OUT UINTN                     *NumberOfProcessors,
  OUT UINTN                     *NumberOfEnabledProcessors
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

NumberOfProcessors

Pointer to the total number of logical processors in the system, including the BSP and all enabled and disabled APs.

NumberOfEnabledProcessors

Pointer to the number of logical processors in the platform including the BSP that are currently enabled.

UEFI Multiprocessing

```
union {
	EFI_PROCESSOR_INFORMATION pi;
	UINT8 _pad[256]; // pi can expand in the future
} pi;
for (UINTN i = 0; i < total_cpus; i++) {
	efi_status = mps->GetProcessorInfo(mps, i, &pi.pi);
	if (EFI_ERROR(efi_status)) {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Cannot get a process info!\r\n");
		return efi_status;
	}
	if (!(pi.pi.StatusFlag & PROCESSOR_AS_BSP_BIT)) { // Not BSP
		efi_status = mps->StartupThisAP(mps, cpu_start, i, NULL, 0, (VOID *) i, NULL);
		if (EFI_ERROR(efi_status)) {
			SystemTable->ConOut->OutputString(SystemTable->ConOut,
				L"Cannot start a processor!\r\n");
			return efi_status;
		}
	} else {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Started CPU [BSP]\r\n");
	}
}
```

UEFI Multiprocessing

```
union {
    EFI_PROCESSOR_INFORMATION pi;
    UINT8 _pad[256]; // pi can expand in the future
} pi;
for (UINTN i = 0; i < total_cpus; i++) {
    efi_status = mps->GetProcessorInfo(mps, i, &pi.pi);
    if (EFI_ERROR(efi_status)) {
        SystemTable->ConOut->OutputString(SystemTable->ConOut,
            L"Cannot get a process info!\r\n");
        return efi_status;
    }
    if (!(pi.pi.StatusFlag & PROCESSOR_AS_BSP_BIT)) { // Not BSP
        efi_status = mps->StartupThisAP(mps, cpu_start, i, NULL, 0, (VOID *) i, NULL);
        if (EFI_ERROR(efi_status)) {
            SystemTable->ConOut->OutputString(SystemTable->ConOut,
                L"Cannot start a processor!\r\n");
            return efi_status;
        }
    } else {
        SystemTable->ConOut->OutputString(SystemTable->ConOut,
            L"Started CPU [BSP]\r\n");
    }
}
```

UEFI Multiprocessing

EFI_MP_SERVICES_PROTOCOL.GetProcessorInfo()

Summary

Gets detailed MP-related information on the requested processor at the instant this call is made. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_PROCESSOR_INFO) (
  IN  EFI_MP_SERVICES_PROTOCOL   *This,
  IN  UINTN                       ProcessorNumber,
  OUT EFI_PROCESSOR_INFORMATION  *ProcessorInfoBuffer
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of processor. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

ProcessorInfoBuffer

A pointer to the buffer where information for the requested processor is deposited. The buffer is allocated by the caller. Type **EFI_PROCESSOR_INFORMATION** is defined in "Related Definitions" below.

UEFI Multiprocessing

ProcessorId

The unique processor ID determined by system hardware.

For IPF, the lower 16 bits contains id/eid, and higher bits are reserved.

StatusFlag

Flags indicating if the processor is BSP or AP, if the processor is enabled or disabled, and if the processor is healthy. The bit format is defined below.

Location

The physical location of the processor, including the physical package number that identifies the cartridge, the physical core number within package, and logical thread number within core. Type **EFI_PHYSICAL_LOCATION** is defined below.

```
//*******************************************************
// StatusFlag Bits Definition
//*******************************************************
#define PROCESSOR_AS_BSP_BIT          0x00000001
#define PROCESSOR_ENABLED_BIT         0x00000002
#define PROCESSOR_HEALTH_STATUS_BIT   0x00000004
```

PROCESSOR_AS_BSP_BIT

This bit indicates whether the processor is playing the role of BSP. If the bit is 1, then the processor is BSP. Otherwise, it is AP.

UEFI Multiprocessing

```
for (UINTN i = 0; i < total_cpus; i++) {
    efi_status = mps->GetProcessorInfo(mps, i, &pi.pi);
    if (EFI_ERROR(efi_status)) {
        SystemTable->ConOut->OutputString(SystemTable->ConOut,
            L"Cannot get a process info!\r\n");
        return efi_status;
    }
    if (!(pi.pi.StatusFlag & PROCESSOR_AS_BSP_BIT)) { // Not BSP
        efi_status = mps->StartupThisAP(mps, cpu_start, i, NULL, 0, (VOID *) i, NULL);
        if (EFI_ERROR(efi_status)) {
            SystemTable->ConOut->OutputString(SystemTable->ConOut,
                L"Cannot start a processor!\r\n");
            return efi_status;
        }
    } else {
        SystemTable->ConOut->OutputString(SystemTable->ConOut,
            L"Started CPU [BSP]\r\n");
    }
}
```

UEFI Multiprocessing

```
for (UINTN i = 0; i < total_cpus; i++) {
	efi_status = mps->GetProcessorInfo(mps, i, &pi.pi);
	if (EFI_ERROR(efi_status)) {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Cannot get a process info!\r\n");
		return efi_status;
	}
	if (!(pi.pi.StatusFlag & PROCESSOR_AS_BSP_BIT)) { // Not BSP
		efi_status = mps->StartupThisAP(mps, cpu_start, i, NULL, 0, (VOID *) i, NULL);
		if (EFI_ERROR(efi_status)) {
			SystemTable->ConOut->OutputString(SystemTable->ConOut,
				L"Cannot start a processor!\r\n");
			return efi_status;
		}
	} else {
		SystemTable->ConOut->OutputString(SystemTable->ConOut,
			L"Started CPU [BSP]\r\n");
	}
}
```

UEFI Multiprocessing

EFI_MP_SERVICES_PROTOCOL.StartupThisAP()

Summary

This service lets the caller get one enabled AP to execute a caller-provided function. The caller can request the BSP to either wait for the completion of the AP or just proceed with the next task by using the EFI event mechanism. See the "Non-blocking Execution Support" section in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** for more details. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_STARTUP_THIS_AP) (
  IN  EFI_MP_SERVICES_PROTOCOL*This,
  IN  EFI_AP_PROCEDURE         Procedure,
  IN  UINTN                    ProcessorNumber,
  IN  EFI_EVENT                WaitEvent              OPTIONAL,
  IN  UINTN                    TimeoutInMicroseconds,
  IN  VOID                     *ProcedureArgument     OPTIONAL,
  OUT BOOLEAN                  *Finished              OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

Procedure

A pointer to the function to be run on the designated AP. Type **EFI_AP_PROCEDURE** is defined in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()**.

UEFI Multiprocessing

ProcessorNumber

The handle number of the AP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()**.

WaitEvent

The event created by the caller with **CreateEvent()** service.

If it is **NULL**, then execute in blocking mode. BSP waits until this AP finishes or *TimeoutInMicroSeconds* expires.

TimeoutInMicrosecsond

Indicates the time limit in microseconds for this AP to finish the function, either for blocking or non-blocking mode. Zero means infinity.

If the timeout expires before this AP returns from Procedure, then Procedure on the AP is terminated. The AP is available for subsequent calls to **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP()**.

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

If the timeout expires in non-blocking mode, *WaitEvent* is signaled with **SignalEvent()**.

ProcedureArgument

The parameter passed into *Procedure* on the specified AP.

Finished

If **NULL**, this parameter is ignored.

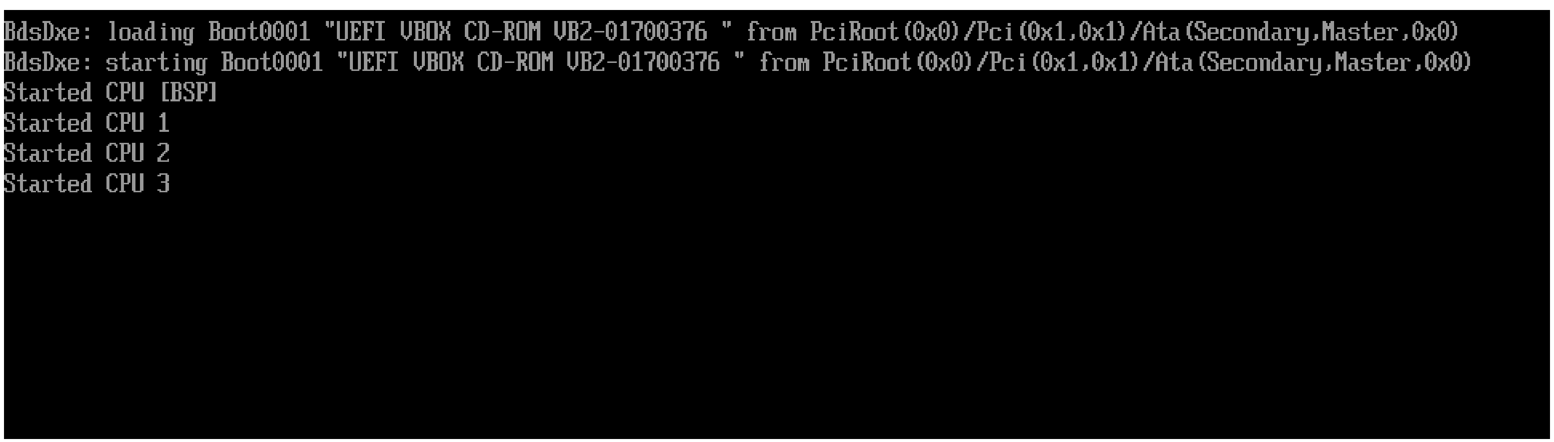
In blocking mode, this parameter is ignored.

UEFI Multiprocessing

```
    BootServices->Stall(5 * 1000000); // 5 seconds

    return EFI_SUCCESS;
}

static VOID EFIAPI
cpu_start(VOID *p)
{
    UINTN cpu = (UINTN) p;
    CHAR16 msg[16] = L"Started CPU ";

    msg[12] = cpu <= 9 ? (cpu + L'0') : L'?';
    msg[13] = L'\r';
    msg[14] = L'\n';
    msg[15] = L'\0';
    SystemTable->ConOut->OutputString(SystemTable->ConOut, msg);
}
```

Execution (4 CPUs)

For qemu, specify **-smp 4** in the command line



Assignment: Debugging

- *Example: Not sure how a bitfield arranges bits in a number*

Create question1.c:

```
typedef unsigned long long u64;

struct my_bitfield {
    u64 reserved:12;
    u64 address:52;
};

void func(struct my_bitfield *v, u64 address)
{
    v->reserved = 0;
    v->address = address;
}
```

Compile:

```
gcc -Wall -O2 -S question1.c
```

Create question2.c:

```
typedef unsigned long long u64;

struct my_bitfield {
    u64 address:52;
    u64 reserved:12;
};

void func(struct my_bitfield *v, u64 address)
{
    v->reserved = 0;
    v->address = address;
}
```

Compile:

```
gcc -Wall -O2 -S question2.c
```

Assignment: Debugging

- *Example: Not sure how a bitfield arranges bits in a number*

question1.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    salq    $12, %rsi
    movq    %rsi, (%rdi)
    ret
    .cfi_endproc
```

Conclusion: Lower bits are 0s (reserved)

Arithmetic/logic shift SAL/SHL to left by 12 (i.e., multiply by 4096)

Note: *Right shifts are different for arithmetic shifts SAR (signed numbers) and logic shifts SHR (unsigned numbers)*

question2.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movabsq $4503599627370495, %rax
    andq    %rax, %rsi
    movq    %rsi, (%rdi)
    ret
    .cfi_endproc
```

Conclusion: Upper bits are 0s (reserved)

%rax = 0xFFFFFFFFFFFFF
A mask which leaves the lower 52 bits intact but clears upper 12 bits

Assignment: Debugging

- *Example: Not sure how pointer arithmetic works*

Create question1.c:

```
typedef unsigned long long u64;

u64 *func(u64 *addr, int offset)
{
    return addr + offset;
}
```

Compile:

gcc -Wall -O2 -S question1.c

Create question2.c:

```
typedef unsigned long long u64;

u64 *func(void *addr, int offset)
{
    return addr + offset;
}
```

Compile:

gcc -Wall -O2 -S question2.c

Assignment: Debugging

- *Example: Not sure how pointer arithmetic works*

question1.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movslq  %esi, %rsi
    leaq    (%rdi,%rsi,8), %rax
    ret
    .cfi_endproc
```

Conclusion: Will multiply ‘offset’ by 8 before adding

LEA (Load Effective Address) will do
%rax = 8 * %rsi + %rdi

%rdi is addr (1st arg)
%rsi is offset (2nd arg)

question2.s:

```
func:
.LFB0:
    .cfi_startproc
    endbr64
    movslq  %esi, %rax
    addq    %rdi, %rax
    ret
    .cfi_endproc
```

Conclusion: Will just add ‘offset’

%rax = %rdi + %esi (sign-extended)

See Also

- x86-64 ABI
  - https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf
  - Specifically “calling conventions”
- Also Wikipedia
  - https://en.wikipedia.org/wiki/X86_calling_conventions