CIS 310 Operating Systems

Week 10: Virtual *Memory*

Dr. Brian C. Ladd

Friday 5th November, 2021

Outline

fork()

- Creates a *new* **process control block** (PCB).
  PCB — OS-specific data structure; access must be privileged.
  `fork()` is a library wrapper around a *system call*.

fork()

- Creates a *new* **process control block** (PCB).
  PCB — OS-specific data structure; access must be privileged.
  `fork()` is a library wrapper around a *system call*.
- Trap to the OS. *k-stack* saved to current (parent) PCB.

`fork()`

- Creates a *new* **process control block** (PCB).
  PCB — OS-specific data structure; access must be privileged.
  `fork()` is a library wrapper around a *system call*.
- Trap to the OS. *k-stack* saved to current (parent) PCB.
- Duplicate parent PCB, duplicate all FD, duplicate memory assignment but separate writable pages.

fork()

- Creates a *new* **process control block** (PCB).
  PCB — OS-specific data structure; access must be privileged.
  `fork()` is a library wrapper around a *system call*.
- Trap to the OS. *k-stack* saved to current (parent) PCB.
- Duplicate parent PCB, duplicate all FD, duplicate memory assignment but separate writable pages.
- **Policy** Prefer parent or child process by returning from the system call to one or the other.

`exec("cmd")`

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations.
  `exec()` is a library wrapper around a *system call*.

`exec("cmd")`

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations.
  `exec()` is a library wrapper around a *system call*.
- Trap to the OS. Save context to PCB. Release user memory (will keep FD and most of the PCB the same).

`exec("cmd")`

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations.
  `exec()` is a library wrapper around a *system call*.
- Trap to the OS. Save context to PCB. Release user memory (will keep FD and most of the PCB the same).
- Load beginning of the *code* segment of `cmd` file. Allocate new memory for *heap* and *stack* for a new program.

exec("cmd")

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations. `exec()` is a library wrapper around a *system call*.
- Trap to the OS. Save context to PCB. Release user memory (will keep FD and most of the PCB the same).
- Load beginning of the *code* segment of `cmd` file. Allocate new memory for *heap* and *stack* for a new program.
- Put the string `cmd` and provided arguments in memory according to OS convention.

`exec("cmd")`

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations.
  `exec()` is a library wrapper around a *system call*.
- Trap to the OS. Save context to PCB. Release user memory (will keep FD and most of the PCB the same).
- Load beginning of the *code* segment of `cmd` file. Allocate new memory for *heap* and *stack* for a new program.
- Put the string `cmd` and provided arguments in memory according to OS convention.
- Modify context so `IP` contains the virtual starting address for a new program.

exec("cmd")

- Loads new program code into memory and starts the loaded program *from the beginning*.
  Will read from *filesystem*. Will modify *context*. Privileged operations. `exec()` is a library wrapper around a *system call*.
- Trap to the OS. Save context to PCB. Release user memory (will keep FD and most of the PCB the same).
- Load beginning of the *code* segment of `cmd` file. Allocate new memory for *heap* and *stack* for a new program.
- Put the string `cmd` and provided arguments in memory according to OS convention.
- Modify context so `IP` contains the virtual starting address for a new program.
- Return from interrupt.

Address Space

Definition

An address space is *all* of the memory that a (process/machine) can **address**.

- If the addresses refer to actual RAM locations, they are **physical** addresses in a physical address space.
- If the addresses must be translated before they refer to actual RAM locations, they are **virtual** addresses in a virtual address space.

The *size* of an address space is determined by the number of address bits it has: *n* **bits** of address $\Rightarrow 2^n$ **bytes** of memory.

Virtual Addresses

- Virtualization should be *transparent*.

Virtual Addresses

- Virtualization should be *transparent*.
- The *process* sees a single, contiguous address space beginning at `0` (hex).

Virtual Addresses

- Virtualization should be *transparent*.
- The *process* sees a single, contiguous address space beginning at `0` (hex).
- All addresses in the process (contents of `IP`, pointers, `load`/`store` instruction targets) are *virtual*.

Virtual Addresses

- Virtualization should be *transparent*.
- The *process* sees a single, contiguous address space beginning at `0` (hex).
- All addresses in the process (contents of `IP`, pointers, `load`/`store` instruction targets) are *virtual*.
- Indirect addressing, through an OS-supported translator, is always applied for user-space machine code.

Address Translation

Base Register

- A **base register** is a CPU or MMU register that permits dynamic memory relocation.

Address Translation

Base Register

- A **base register** is a CPU or MMU register that permits dynamic memory relocation.
- The *physical* address is calculated by **adding** the *virtual* address to the value in the base register.

Address Translation

Base Register

- A **base register** is a CPU or MMU register that permits dynamic memory relocation.
- The *physical* address is calculated by **adding** the *virtual* address to the value in the base register.
- The base register alone can permit user programs to generate dangerous addresses.

Address Translation

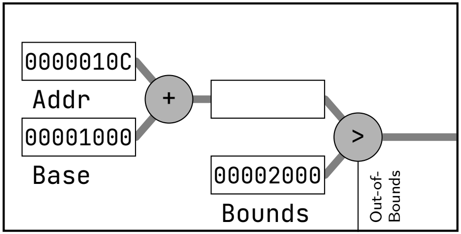
Base Register

- A **base register** is a CPU or MMU register that permits dynamic memory relocation.
- The *physical* address is calculated by **adding** the *virtual* address to the value in the base register.
- The base register alone can permit user programs to generate dangerous addresses.
- Combined with a **bounds register**, the translation is much safer.

Memory Management Unit

Fetch

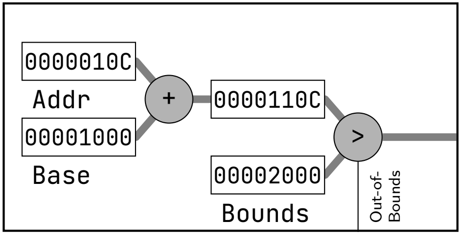
IP	0000010C
Base	00001000
Bounds	00002000



Memory Management Unit

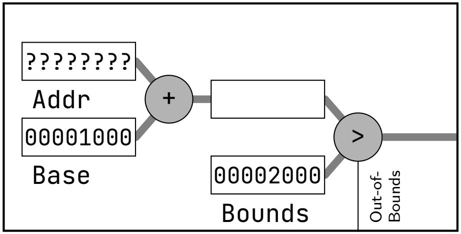
Fetch

IP	0000010C
Base	00001000
Bounds	00002000



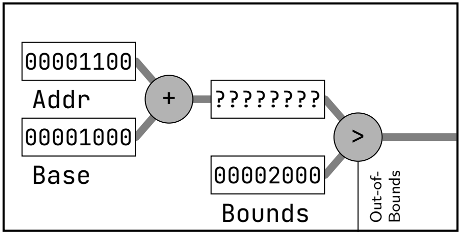
Memory Management Unit

```
load R1, 0x00001100
 Base     00001000
 Bounds   00002000
```



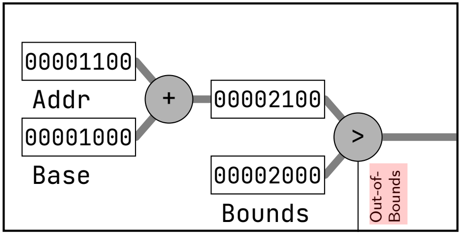
Memory Management Unit

```
load R1, 0x00001100
 Base     00001000
 Bounds   00002000
```



Memory Management Unit

```
load R1, 0x00001100
 Base     00001000
 Bounds   00002000
```



Problems Fitting Processes

- Using a single base/bounds register pair for a process permits dynamic relocation of the *whole virtual address space*.

Problems Fitting Processes

- Using a single base/bounds register pair for a process permits dynamic relocation of the *whole virtual address space*.
- This requires contiguous **physical** memory that can contain the **virtual** address space.

Problems Fitting Processes

- Using a single base/bounds register pair for a process permits dynamic relocation of the *whole virtual address space*.
- This requires contiguous **physical** memory that can contain the **virtual** address space.
- Fixed size process spaces will probably over allocate for many processes, wasting memory in *internal* fragmentation.

Problems Fitting Processes

- Using a single base/bounds register pair for a process permits dynamic relocation of the *whole virtual address space*.
- This requires contiguous **physical** memory that can contain the **virtual** address space.
- Fixed size process spaces will probably over allocate for many processes, wasting memory in *internal* fragmentation.
- Variable-size process spaces will lead to *external* fragmentation.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.
- Can break down by **segment**.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.
- Can break down by **segment**.
- With four base registers, the left-most pair of bits would indicate which segment base to use for relocation.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.
- Can break down by **segment**.
- With four base registers, the left-most pair of bits would indicate which segment base to use for relocation.
- The OS can allocate four blocks, each a quarter of the size of the virtual address space, for the process.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.
- Can break down by **segment**.
- With four base registers, the left-most pair of bits would indicate which segment base to use for relocation.
- The OS can allocate four blocks, each a quarter of the size of the virtual address space, for the process.
- Still variable size. Can still fragment.

Multiple Base/Bounds Pairs

- Easier to fit if virtual address space is broken down into multiple pieces, each with it's own base/bounds pair.
- Can break down by **segment**.
- With four base registers, the left-most pair of bits would indicate which segment base to use for relocation.
- The OS can allocate four blocks, each a quarter of the size of the virtual address space, for the process.
- Still variable size. Can still fragment.
- **Speed**: Another problem is speed of addition. $O(\log(\text{bits}))$

Fixed Size Allocation Blocks

- Avoid external fragmentation by only allocating a single size of block.

Fixed Size Allocation Blocks

- Avoid external fragmentation by only allocating a single size of block.
- Big block v. small block

Fixed Size Allocation Blocks

- Avoid external fragmentation by only allocating a single size of block.
- Big block v. small block
  - Big: fewer base registers needed; more internal fragmentation

Fixed Size Allocation Blocks

- Avoid external fragmentation by only allocating a single size of block.
- Big block v. small block
  - Big: fewer base registers needed; more internal fragmentation
  - Small: less internal fragmentation; more base registers

Fixed Size Allocation Blocks

- Avoid external fragmentation by only allocating a single size of block.
- Big block v. small block
  - Big: fewer base registers needed; more internal fragmentation
  - Small: less internal fragmentation; more base registers
- Small *page size* with translation information accessed indirectly from RAM is **paged virtual memory**.

Address Translation

Page Table

- A **page table base register** (PTBR) is a CPU or MMU register that contains the **physical** address of the beginning of the *page table*.

Address Translation
Page Table

- A **page table base register** (PTBR) is a CPU or MMU register that contains the **physical** address of the beginning of the *page table*.
- The PTBR is part of the context for a process; the page table is a *per process* data structure.

Address Translation

Page Table

- A **page table base register** (PTBR) is a CPU or MMU register that contains the **physical** address of the beginning of the *page table*.
- The PTBR is part of the context for a process; the page table is a *per process* data structure.
- Address translation is done by separating addresses into two parts: the page/frame number and the offset.

`<page #>`	`<offset>`

  Only the *page #* is translated from the virtual to the physical address.

Address Translation

Page Table

- A **page table base register** (PTBR) is a CPU or MMU register that contains the **physical** address of the beginning of the *page table*.
- The PTBR is part of the context for a process; the page table is a *per process* data structure.
- Address translation is done by separating addresses into two parts: the page/frame number and the offset.

`<page #>`	`<offset>`

  Only the *page #* is translated from the virtual to the physical address.
- The *offset* is a fixed number of bits wide so that the bounds register is no longer required.

Paged Translation

- Split *virtual address*: `<page, offset>`

Paged Translation

- Split *virtual address*: `<page, offset>`
- Use `page` as an index into the page table array

Paged Translation

- Split *virtual address*: `<page, offset>`
- Use `page` as an index into the page table array
- Get `frame` out of page table entry

Paged Translation

- Split *virtual address*: `<page, offset>`
- Use `page` as an index into the page table array
- Get `frame` out of page table entry
- Combine `<frame,offset>` into *physical address*

Page Lookup

Fetch

Page Size	`4KB`
`IP`	`0000210C`
`PTBR`	`F0000000`
`sizeof``(PTE)`	`4B`

Offset Bits:

Page Lookup

Fetch

Page Size	`4KB`
IP	`0000210C`
PTBR	`F0000000`
`sizeof``(PTE)`	`4B`

Offset Bits: `12b`

Page #:

Page Lookup

Fetch

Page Size	4KB
IP	0000210C
PTBR	F0000000
sizeof(PTE)	4B

Offset Bits: 12b

Page #: 00002

Offset:

Page Lookup

Fetch

Page Size	`4KB`
IP	`0000210C`
PTBR	`F0000000`
`sizeof``(PTE)`	`4B`

Offset Bits: `12b`

Page #: `00002`

Offset: `10C`

Address of PTE:

Page Lookup

Fetch

Page Size	`4KB`
`IP`	`0000210C`
`PTBR`	`F0000000`
`**sizeof**(PTE)`	`4B`

Offset Bits: `12b`

Page #: `00002`

Offset: `10C`

Address of PTE: `F0000000` +

Page Lookup

Fetch

Page Size	`4KB`
IP	`0000210C`
PTBR	`F0000000`
sizeof(PTE)	`4B`

Offset Bits: `12b`

Page #: `00002`

Offset: `10C`

Address of PTE: `F0000000` $+$ `4` $\times$

Page Lookup

Fetch

Page Size	4KB
IP	0000210C
PTBR	F0000000
sizeof(PTE)	4B

Offset Bits: 12b

Page #: 00002

Offset: 10C

Address of PTE: F0000000 $+$ 4 $\times$ 00002 $=$

Page Lookup

Fetch

Page Size	`4KB`
`IP`	`0000210C`
`PTBR`	`F0000000`
`sizeof``(PTE)`	`4B`

Offset Bits: `12b`

Page #: `00002`

Offset: `10C`

Address of PTE: `F0000000` $+$ `4` $\times$ `00002` $=$ `F0000008`

`RAM[F0000008] = 007F2`

Page Lookup

Fetch

Page Size	`4KB`
IP	`0000210C`
PTBR	`F0000000`
`sizeof``(PTE)`	`4B`

Offset Bits: `12b`

Page #: `00002`

Offset: `10C`

Address of PTE: `F0000000` $+$ `4` $\times$ `00002` $=$ `F0000008`

`RAM[F0000008] = 007F2`

Physical Address:

Page Lookup

Fetch

Page Size	4KB
IP	`0000210C`
PTBR	`F0000000`
`sizeof``(PTE)`	4B

Offset Bits: 12b

Page #: `00002`

Offset: `10C`

Address of PTE: $\texttt{F0000000} + 4 \times \texttt{00002} = \texttt{F0000008}$

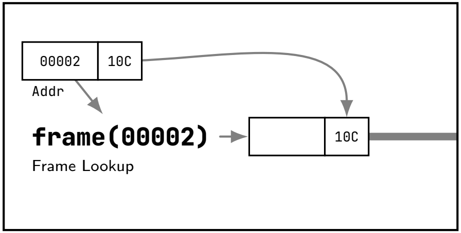
`RAM[F0000008] = 007F2`

Physical Address: `007F210C`

Page Lookup

Fetch

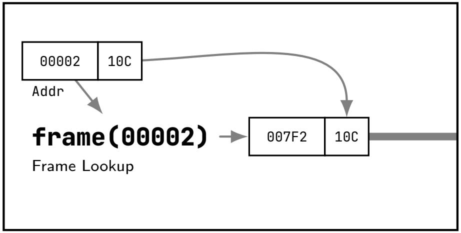
Page Size	4KB
IP	0000210C
PTBR	F0000000
sizeof(PTE)	4B



Page Lookup

Fetch

Page Size	4KB
IP	0000210C
PTBR	F0000000
sizeof(PTE)	4B


00002
10C
Addr
frame(00002)
Frame Lookup
007F2
10C