CSE 323: Operating System Design

Virtual Memory (Segmentation)

Salman Shamil

North South University (NSU)
Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]

Lecture Topics

- Abstraction: address space
- Policy: isolation
- Mechanism: address translation
- Mechanism: heap management

This slide deck covers chapters 13–17 in OSTEP.

Virtualization

Goal: isolate processes (and their faults) from each other.

Virtualization

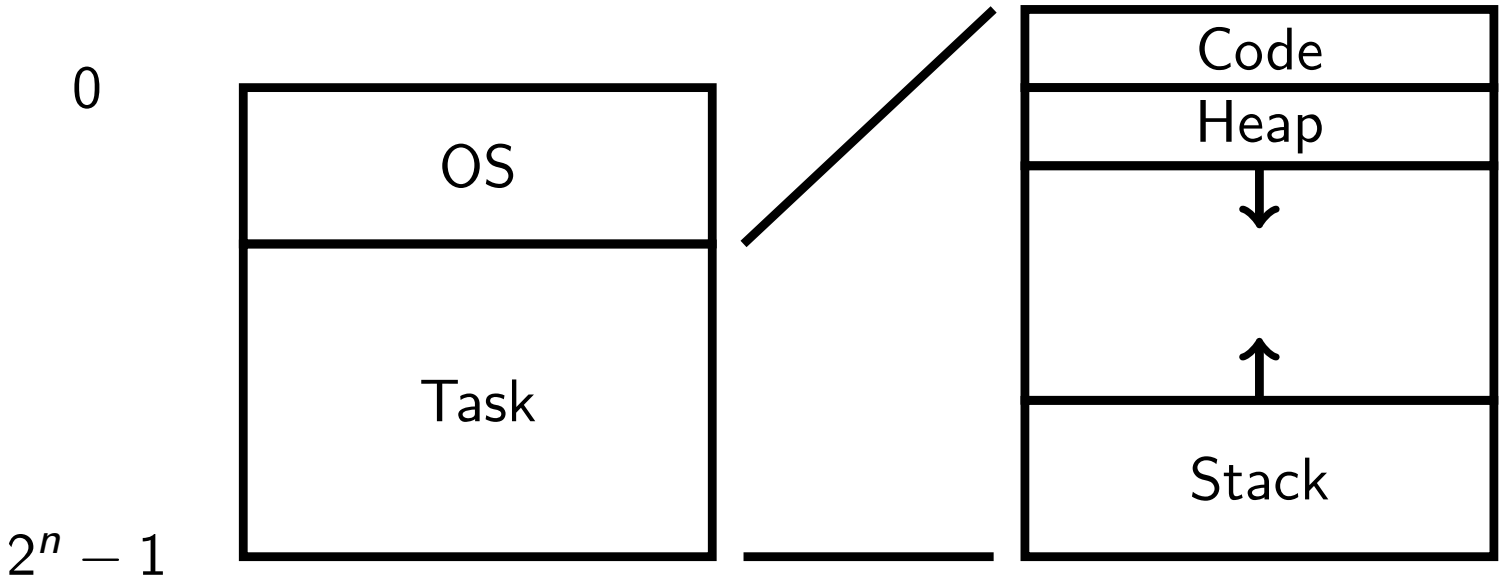
Goal: isolate processes (and their faults) from each other.

Virtualization enables isolation, but isolation requires separation. A process must be prohibited to access memory/registers of another process.

- Step 1: Virtual CPU provides illusion of private CPU registers (mechanisms and policy)
- Step 2: Virtual RAM provides illusion of private memory

History: Uniprogramming

- Initially the OS was a set of library routines
- Issue 1: only one task at a time
- Issue 2: no isolation between OS / task



Sharing Memory

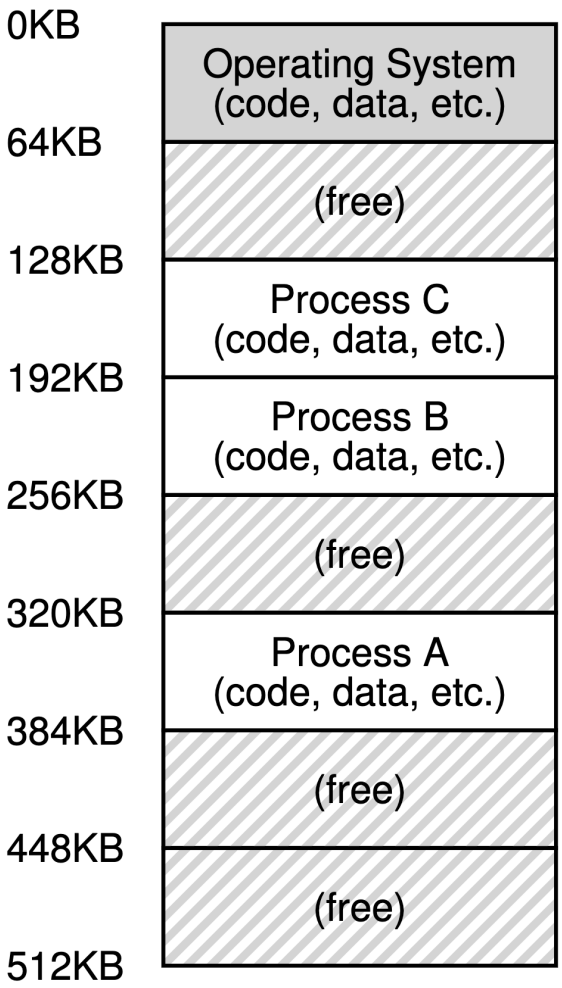


Figure 1: Three processes sharing memory

- A simplified way to share memory among multiple processes.
- Assumption: all three processes fit inside the physical memory.
- Space sharing has its own challenges.
  - Q: Why can't we do time-sharing?
  - Q: How to protect each process (and OS) from other processes?

Goals for Multiprogramming

- **Transparency:** processes are unaware of memory sharing and the existence of other processes
- **Protection:** OS/other processes are isolated from any process (read/write)
- **Efficiency (1):** do not waste too much memory for structures (e.g., fragmentation)
- **Efficiency (2):** do not make programs too slow (use hardware support if necessary)
- **Sharing:** processes *may* share part of address space

Abstraction: Address Space

Address Space: each process has a set of addresses that map to data (i.e., a map from pointers to bytes)

- Static: code and global variables
- Dynamic: stack, heap

Abstraction: Address Space

Address Space: each process has a set of addresses that map to data (i.e., a map from pointers to bytes)

- Static: code and global variables
- Dynamic: stack, heap

Why do we need dynamic memory?

Abstraction: Address Space

Address Space: each process has a set of addresses that map to data (i.e., a map from pointers to bytes)

- Static: code and global variables
- Dynamic: stack, heap

Why do we need dynamic memory?

- The amount of required memory may be task dependent
- Input size may be unknown at compile time
- Conservative pre-allocation would be wasteful
- Recursive functions (invocation frames)

Dynamic Data Structure: Stack

- Data is returned in reverse order from insertion
  - `push(1); push(2); push(3);`
  - `pop()->3; pop()->2; pop()->1;`
- Memory is freed in reverse order from allocation
  - `a=alloc(20); b=alloc(10);`
  - `free(b); free(a);`

Dynamic Data Structure: Stack

- Data is returned in reverse order from insertion
  - `push(1); push(2); push(3);`
  - `pop()->3; pop()->2; pop()->1;`
- Memory is freed in reverse order from allocation
  - `a=alloc(20); b=alloc(10);`
  - `free(b); free(a);`
- Straight-forward implementation: bump or decrement a pointer
  - Advantage: no fragmentation, no metadata
  - Note: deallocations ***must*** be in ***reverse order***

Excursion: Procedure Invocation Frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

Excursion: Procedure Invocation Frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

- Slot for `int tmp`
- Slots for the parameters a, b
- Slot for the return code pointer
- Usual ordering: b, a, RIP, tmp

Excursion: Procedure Invocation Frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

- Slot for `int tmp`
- Slots for the parameters a, b
- Slot for the return code pointer
- Usual ordering: b, a, RIP, tmp

The compiler creates the necessary code.

Stack for Procedure Invocation Frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

Stack for Procedure Invocation Frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

What happens to the data when function returns?

Stack for Procedure Invocation Frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

What happens to the data when function returns?

- Data from previous function lingers, overwritten when the next function initializes its data

Quiz: Scopes and Stack

```c
int a = 2;
int called(int b) {
  int c = a * b;
  printf("a: %d b: %d c: %d\n", a, b, c);
  a = 5;
  return c;
}
int main(int argc, char* argv) {
  int b = 2, c = 3;
  printf("a: %d b: %d c: %d\n", a, b, c);
  b = called(c);
  printf("a: %d b: %d c: %d\n", a, b, c);
  return 0;
}
```

Dynamic Data Structure: Heap

A heap of randomly allocated memory objects with *statically unknown size* and *statically unknown allocation patterns*. The size and lifetime of each allocated object is unknown.

API: `malloc` creates an object, `free` indicates it is no longer used.

Dynamic Data Structure: Heap

A heap of randomly allocated memory objects with *statically unknown size* and *statically unknown allocation patterns*. The size and lifetime of each allocated object is unknown.

API: `malloc` creates an object, `free` indicates it is no longer used.

How would you manage such a data structure?

Heap: Naive Implementation

```
char storage[4096], *heap = storage;
char *alloc(size_t len) {
  char *tmp = heap;
  heap = heap + len;
  return tmp;
}

void free(char *ptr) {}
```

- Advantage: simple
- Disadvantage: no reuse, will run out of memory
  - unused fragments

Heap: Free List

Idea: abstract heap into list of free blocks.

- Keep track of free space, program handles allocated space
- Keep a list of all available memory objects and their size

Implementation:

- `malloc`: take a free block, split, put remainder back on free list
- `free`: add block to free list

Heap: Implementation Strategies

- Allocation: find a fitting object (first, best, worst fit)
  - first fit: find the first object in the list and split it
  - best fit: find the object that is closest to the size
  - worst fit: find the largest object and split it

Heap: Implementation Strategies

- Allocation: find a fitting object (first, best, worst fit)
  - first fit: find the first object in the list and split it
  - best fit: find the object that is closest to the size
  - worst fit: find the largest object and split it
- Free: merge adjacent blocks
  - if the adjacent region is free, merge the two blocks

Quiz: where is it?

```
int g;
int main(int argc, char *argv[]) {
  int foo;
  char *c = (char*)malloc(argc*sizeof(int));
  free(c);
}
```

Possible storage locations: stack, heap, globals, code

Quiz: where is it?

```
int g;
int main(int argc, char *argv[]) {
  int foo;
  char *c = (char*)malloc(argc*sizeof(int));
  free(c);
}
```

Possible storage locations: stack, heap, globals, code

- Stack: argc, argv, foo, c
- Heap: *c
- Globals: g
- Code: main

Virtualizing Memory

- Challenge: how can we run multiple programs in parallel?
  - Addresses are hard coded in code
  - Static allocation? What about executing the same task twice?
- Possible sharing mechanisms:
  - Time sharing
  - Static relocation/allocation
  - Base (+ bounds)
  - Segmentation
  - Virtual memory

Virtualizing memory: time sharing

- Reuse idea from CPU virtualization
  - OS virtualizes CPU by storing register state to memory
  - Could virtualize memory by storing state to disk
- Disadvantage: incredibly bad performance due to I/O latency
- Better: space sharing (divide memory among processes)

Tangent: track that memory access

- How many memory accesses are executed?
- What kind of memory accesses (read or write)?

```
0x10: mov -0x4(%rbp),%edx
0x13: mov -0x8(%rbp),%eax
0x16: add %edx,%eax
0x18: mov %eax,-0x8(%rbp)
```

Tangent: track that memory access

- How many memory accesses are executed?
- What kind of memory accesses (read or write)?

```
0x10: mov -0x4(%rbp),%edx
0x13: mov -0x8(%rbp),%eax
0x16: add %edx,%eax
0x18: mov %eax,-0x8(%rbp)
```

```
0x10: mov -0x4(%rbp),%edx   # Load 0x10 Exe Load *(%rbp-4)
0x13: mov -0x8(%rbp),%eax   # Load 0x13 Exe Load *(%rbp-8)
0x16: add %edx,%eax         # Load 0x16 Exe
0x18: mov %eax,-0x8(%rbp)   # Load 0x18 Exe Store *(%rbp-8)
```

Virtualizing memory: static relocation

```
0x10: mov -0x4(%rbp),%edx
0x13: mov -0x8(%rbp),%eax
0x16: add %edx,%eax
0x18: call 60 <printf@plt>
```

OS relocates text segment (code area) when new task is started:

Virtualizing memory: static relocation

```
0x10: mov -0x4(%rbp),%edx
0x13: mov -0x8(%rbp),%eax
0x16: add %edx,%eax
0x18: call 60 <printf@plt>
```

OS relocates text segment (code area) when new task is started:

```
# Task 1                          # Task 2
0x1010: mov -0x4(%rbp),%edx       0x5010: mov -0x4(%rbp),%edx
0x1013: mov -0x8(%rbp),%eax       0x5013: mov -0x8(%rbp),%eax
0x1016: add %edx,%eax             0x5016: add %edx,%eax
0x1018: call 1060 <printf>        0x5018: call 5060 <printf>
```

Virtualizing memory: static relocation

- When loading a program, relocate it to an assigned area
- Carefully adjusts all pointers in code and globals, set the stack pointer to the assigned stack

Virtualizing memory: static relocation

- When loading a program, relocate it to an assigned area
- Carefully adjusts all pointers in code and globals, set the stack pointer to the assigned stack

- There is only one address space, no physical/virtual separation
- Issue 1: no separation between processes (no integrity or confidentiality)
- Issue 2: fragmentation, address space remains fixed as long as program runs
- Issue 3: programs have to be adjusted when loaded (e.g., target of a jump will be at different addresses depending on the location in the address space)

Challenge: illusion of private address space

How can the OS provide the illusion of a private address space to each process?

Virtualizing memory: dynamic relocation

- What if, instead of relocating the memory accesses ahead of time, the hardware could help us relocate accesses just-in-time?
- In dynamic relocation, a hardware mechanism translates each memory address from the program's viewpoint to the hardware's viewpoint.

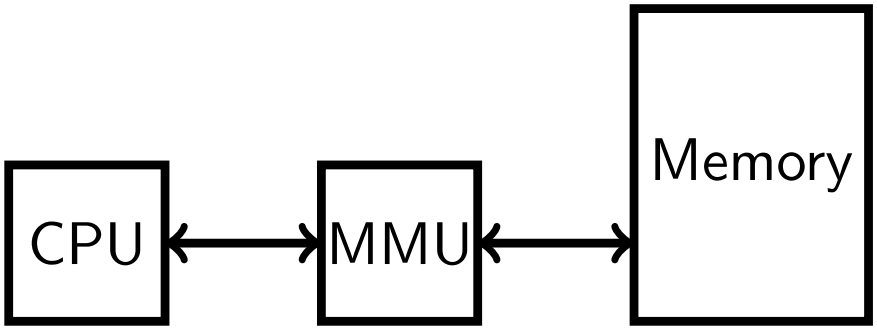
Interposition: the hardware will intercept each memory access and dynamically and transparently translate for the program from virtual addresses (VA) to physical addresses (PA). The OS manages the book keeping of which physical addresses are associated with what processes.

Indirection

> *We can solve any problem by introducing an extra level of indirection. [Except for the problem of too many layers of indirection.]*

(Andrew Koenig attributed the quote to Butler Lampson who attributed it to David J. Wheeler, adding another layer of indirection.)

MMU: Memory Management Unit



- Process runs on the CPU
- OS controls CPU and MMU
- MMU translates virtual addresses (logical addresses) to physical addresses

Privilege modes

How do you keep the process from modifying the MMU configuration?

Privilege modes

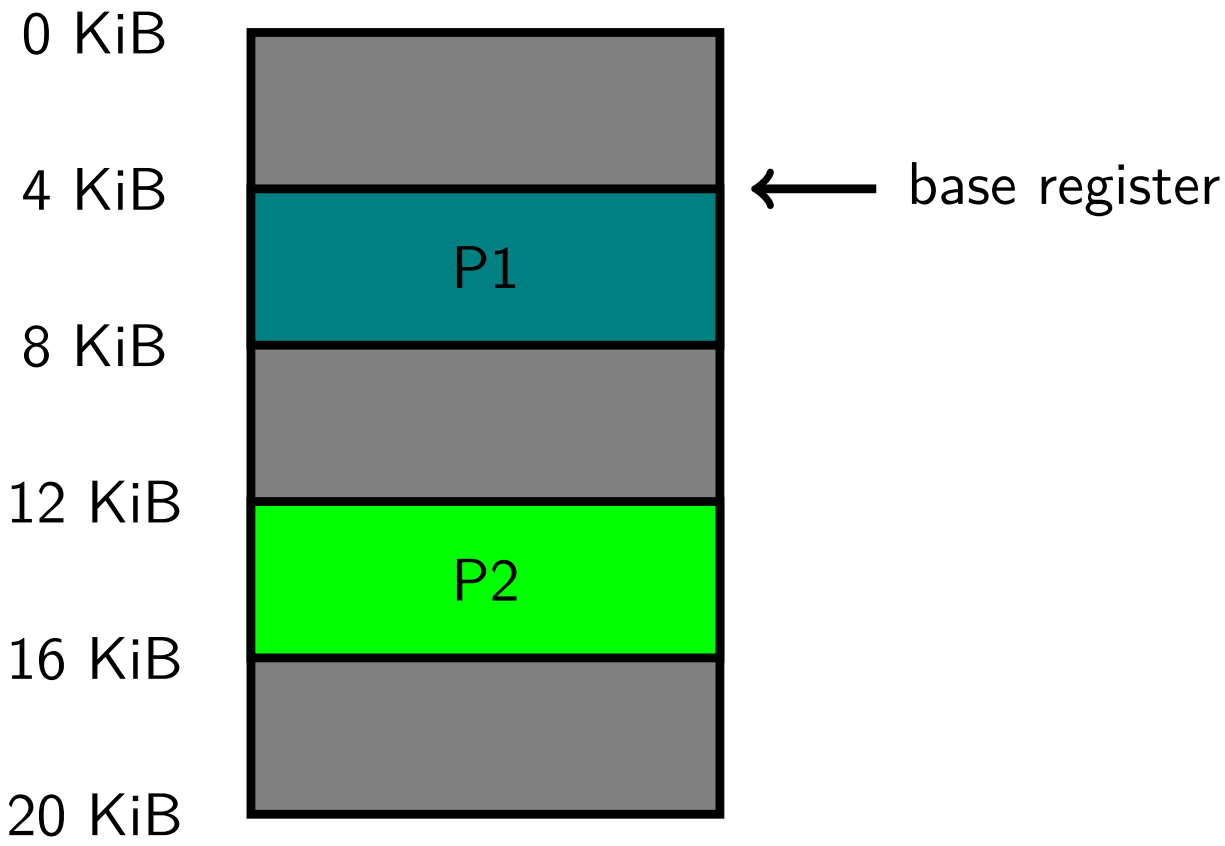
How do you keep the process from modifying the MMU configuration?

- Separation: OS runs at higher privileges than process
- OS privileges include special instructions for MMU config
- Switch from user-space (process) to kernel space through system call (special call instruction)
- OS returns to unprivileged user mode (with special return)
- Exceptions in user space (e.g., illegal memory access, division by 0) switch to privileged mode, OS handles the exception

A simple MMU: base register

- Idea: translate virtual to physical addresses by adding offset.
- Store offset in special register (OS controlled, used by MMU).
- Each process has a different offset in their base register

A simple MMU: base register



A simple MMU: base register

- Set base register to $0x1000$ for P1
- Load of address $0x100_v$ becomes $0x1100_p$
- Set base register to $0x3000$ for P2
- Load of address $0x52_v$ becomes $0x3052_p$

A simple MMU: base register

- Is this design free from security issues?
  - Are processes P1 and P2 truly separated?

A simple MMU: base register

- Is this design free from security issues?
  - Are processes P1 and P2 truly separated?

No! P1 can access the memory of P2 as the base register is simply added. In the previous example, with `base=0x1000`, accessing address $0x2000_v$ will access the first byte of memory of P2 while P1 is executing!

A simple MMU: base and bounds

- Simple solution: base and bounds
  - Base register sets minimum address
  - Bounds register sets (virtual) limit of the address space, highest physical address that is accessible becomes `base+bounds`
- New concept: access check

```
if (addr < bounds) {
  return *(base+addr);
} else {
  throw new SegFaultException();
}
```

Note: bounds can either store the size of the address space or the upper memory address; this is an implementation choice.

A simple MMU: base and bounds

- Achieves security (isolation property is satisfied)
- Achieves performance (translation and check are cheap)
- What's the remaining problem?

A simple MMU: base and bounds

- Achieves security (isolation property is satisfied)
- Achieves performance (translation and check are cheap)
- What's the remaining problem?

- All memory must be continuously allocated
  - Waste of physical memory (all must be allocated)
  - No (easy) sharing between processes

A simple MMU: segmentation

Instead of a single base/bounds register pair, have one pair per memory area:

- Code Segment (CS on x86, default for instructions)
- Data Segment (DS on x86, default for data accesses)
- Stack Segment (SS on x86, default for push/pop)
- Extra Segments (ES, FS, and GS on x86, for anything else)

Allow a process to have several regions of continuous memory mapped from a virtual address space to a physical address space.

Note that hardware also allows to override default segment registers, allowing the programmer to specify which segment should be used. E.g., loading data from the code segment.

Summary

- OS manages access to constrained resources
  - Principle: limited direct execution (bare metal when possible, intercept when needed)
  - CPU: time sharing between processes (low switching cost)
  - Memory: space sharing (disk I/O is slow, so time sharing is expensive)
- Programs use dynamic data
  - Stack: program invocation frames
  - Heap: unordered data, managed by user-space library (allocator)
- Time sharing: one process uses all of memory
- Base register: share space, calculate address through offset
- Base + bounds: share space, limit process' address space
- Segments: movable segments, virtual offsets to segment base