Systems I – CS 6013
Computer Architecture and Operating Systems
Lecture 15: Virtual Memory

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*adapted from slides by Ryan Stutsman, Andrea Arpaci-Dusseau, and Sarah Diesburg, and MSD presentations

Lecture 15 – Topics

- Virtual Memory
  - Motivation
  - Address Space
    - Beginnings of Address Translation

Questions

- Code Review
- Questions

Virtualization

- The OS is all about *virtualizing* the computer…
- So far this semester:Virtual CPU
  - So far this course as discussed Virtualization of the CPU. What does this mean?
    - Each process believes it has its own CPU. Keyword for this?
      - LDE – Limited Direct Execution
    - How is this done?
      - Only a single process can execute at a given time.
      - Each process has its own registers in the CPU.
        - Well – the illusion of its own registers… How do processes share the CPU (and its registers)?
        - Context switching
- Next couple of weeks: Virtual Memory:
  - Illusion of private addresses and memory

Motivation for Virtualization

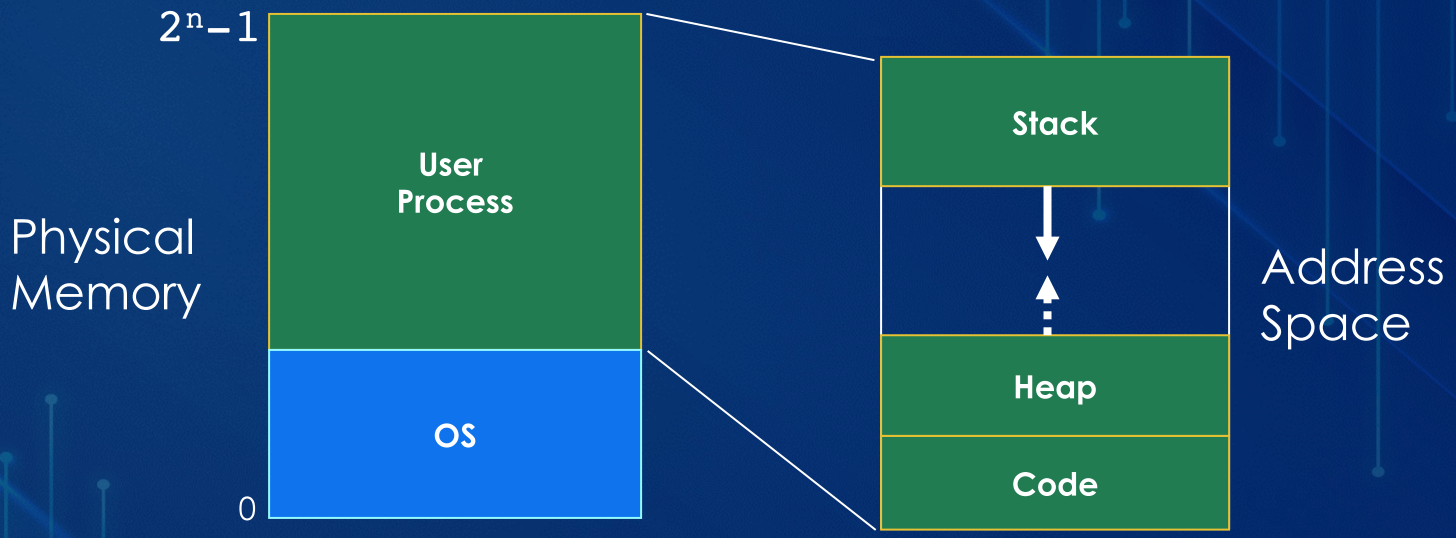
- A long, long time ago…
  - **Uniprogramming**: One process runs at a time

Disadvantages:

- Only one process runs at a time
- Process can destroy OS

Question:

- What is n?
  - Number of bits in an address

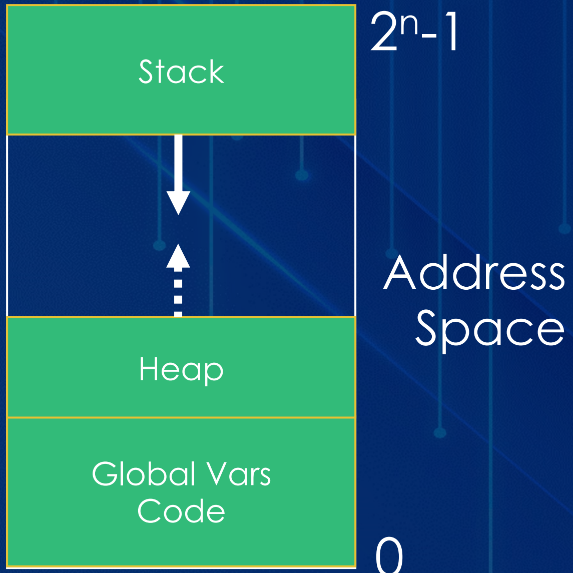


Multiprogramming Goals

- Transparency
  - Processes are not aware that memory is shared
  - Works regardless of number and/or location of processes
- Protection
  - Secrecy: Cannot read data of OS or other processes
  - Integrity: Cannot change data of OS or other processes
- Efficiency
  - Do not waste memory resources (minimize fragmentation)
  - Do not waste CPU cycles
- Sharing
  - Cooperating processes can share portions of address space

Abstraction: Address Space

- **Address space**: process' set of addresses (that *map* to the machine's real physical memory).
  - How does OS provide illusion of private address space to each process?
- Review: What is in an address space?
- Address space has static and dynamic components
  - Static: Code and some global variables
  - Dynamic: Stack and Heap
- *Note: Our book draws this picture upside down…



Motivation for Dynamic Memory

- Give the process (and programmer) the illusion that they have their own private address space.
- Why do processes need dynamic memory allocation?
  - Do not know amount of memory needed at compile time
  - Must be pessimistic when allocating memory statically
  - If we allocate for worst case, storage is used inefficiently
- Recursive procedures
  - What is a recursive procedure?
    - Function that calls itself… eg: Fibonacci
  - Do not know how many times procedure will be nested
- Complex data structures: lists and trees
  - `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`
  - Can build them on the fly.

- Two types of dynamic allocation
  - Stack
  - Heap

Stack Organization

- **Stack**: memory freed in opposite order from allocation
  - As functions are called, their memory is pushed on the stack
  - When a function returns, its memory is popped from the stack.
- Simple and efficient implementation:
  - Pointer separates allocated and free space
  - Allocate space: decrement pointer
  - Free up space: increment pointer
- Which pointer?
  - SP – Stack Pointer
- No fragmentation
- "Automatic"
  - Compiler adjusts stack pointer on entry/exit to calls* (allocates/free space)

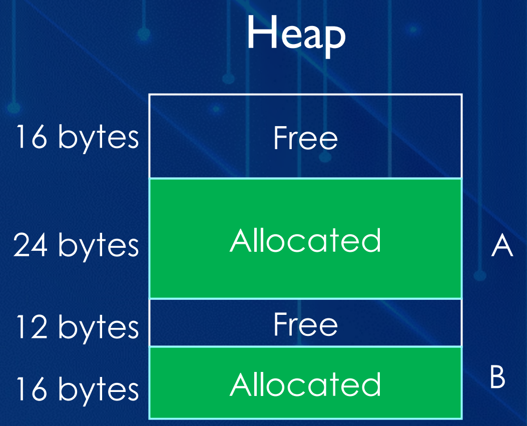
*Calls / Functions / Procedures / Methods

Where Are Stacks Used?

- Processes use the stack for procedure call frames… which includes?
  - local variables and parameters, and…
  - At the assembly level?
    - information about the stack itself (SP, BP)
- Note, the OS is not (directly) involved in the use of the Stack

Heap Organization

- **Heap**: memory region where allocation/free are *explicit*
  - Heap consists of allocated areas and free areas (holes)
- Advantage
  - Allocation lifetime is independent of function call / return
    - Ie, data can exist / persist outside of a function call.
  - Works for all data structures
- Disadvantages
  - Allocation can be slow
  - End up with small chunks of free space – fragmentation
  - Leaks (program forgets free), double free
- What is OS's role in managing heap?
  - OS gives big chunks of free memory to process (sbrk/mmap);
  - Memory Library manages individual allocations (malloc/free)



Heap Fragmentation

- How do you allocate memory on the heap?
  - C: malloc(SIZE)
  - C++: new (or malloc)
- What is required from the memory system when allocating a large array?
  - Memory must be contiguous
- If the program alternates between allocating small blocks and big blocks, even when some memory is freed (especially small blocks) that memory may not be useful to the program (as any needed "big" block won't be able to find contiguous memory… This is called fragmentation.

Memory Accesses – Look at Assembly

```
int main( int argc, char *argv[] ) {
  int x;
  x = x + 3;
}
```

```
0x10: mov edi, 0x4+rbp   ; Put X into edi
0x13: add edi, 3         ; Add 3 to edi
0x19: mov 0x4+rbp, edi   ; Move edi back into X
```

- Quick quiz question…
  - What can go here?
    - env

Reminder: `bp` is the base pointer: points to base of current stack frame

Memory Accesses?

ip = 0x10 (Initial)

bp = 0x200

```
➡ 0x10: mov edi, [0x4+rbp]
  0x13: add edi, 3
  0x19: mov [0x4+rbp], edi
```

ip is instruction pointer (aka, the program counter), bp is the stack base pointer

CPU control unit state machine has two main states…

1. fetch instruction (from?)
   - memory
2. execute

Fetch instruction at addr 0x10

Execution: load from addr 0x204

Fetch instruction at addr 0x13

Execution: no memory access

Fetch instruction at addr 0x19

Execution: store to addr 0x204

Virtualizing Memory

- How do we run multiple processes when addresses are *hardcoded* into process binaries?
- Possible solutions
  - Time Sharing
  - Static Relocation
  - Dynamic Relocation
    - Base & Bound
    - Segmentation
    - Paging (Coming Soon)

Time Sharing **Memory**

- Try similar approach to how OS virtualizes CPU
- So, given the observation that…
  - OS gives the illusion of many CPUs by saving CPU registers to memory when a process isn't running
- What could the OS do to give a process the illusion of having all memory on the machine to itself?
  - Actually give ***all*** memory to the process!
  - The OS could save all memory to disk when process isn't running, and load (all of) a new process's memory back from disk when it is about to run.

Time Sharing is No Good

- Why is this a (really) bad idea for memory, but works for CPU registers?
  - Ridiculously poor performance due to slow speeds of memory and disk data transfer
  - Memory "footprint" on CPU is much (much) smaller – Only a few registers, and transfer from CPU to Memory is much faster than Memory to Disk.
- A Better Alternative: Space sharing
  - Physical (*real*) memory is divided across processes at the same time.
- Note, the remainder of approaches we will discuss all use space sharing

Static Relocation

- Idea: OS rewrites program's memory references before loading in memory
  - Program compiled to use addresses 0 to 0x7fffff…
  - When program is loaded into memory (to run), the OS modifies (offsets) all memory references by some amount.
  - Every process is offset to a different (real/physical) location.
  - Must change jumps, loads of static data, etc
- For example:
  - Each process might be given 3000 bytes. So the 3rd process will begin at physical address…?
    - Assume physical addresses 0-2999 are used by the OS itself.
  - 9000 - The OS adds 9000 to every address in the program.

```
0x10:  mov   edi,
0x4+rbp
0x13:  add   edi, 3
0x19:  mov 0x4+rbp,
edi
```

Program loader rewrites

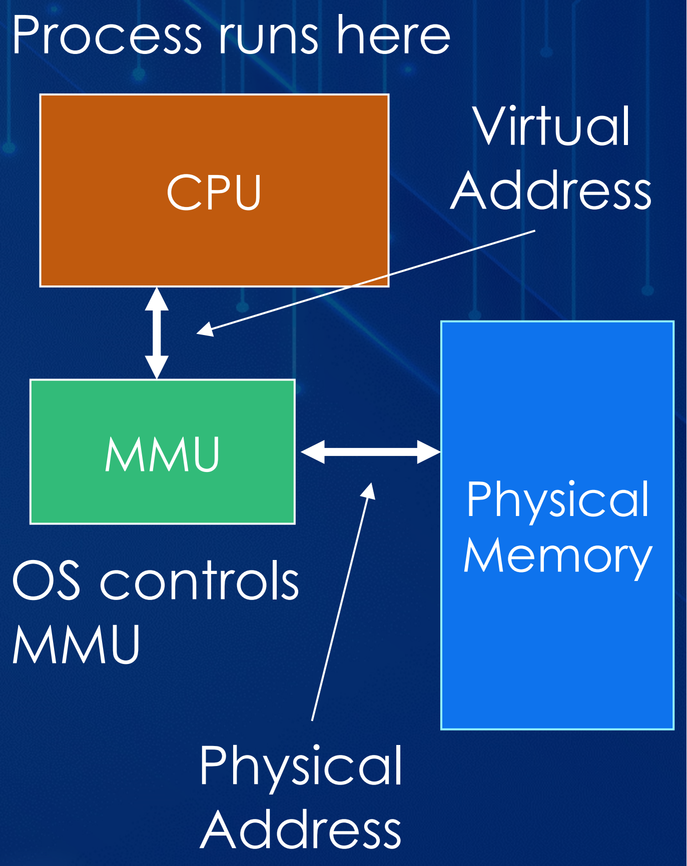
```
0x9010: mov edi, 0x4+rbp
0x9013: add  edi, 3
0x9019: mov 0x4+rbp, edi
0x901C:   jmp 0x9010
```

Static Relocation Problems

- Cannot move address space after it has been placed.
- If a process (memory footprint) grows over time, this may require re-computation of addresses on the fly.
  - Or risk violation of other processes!

Dynamic Relocation

- **Goal**: Relocation at ***runtime*** and protect processes from one another
- Requires hardware support: **Memory Management Unit (MMU)**
- MMU dynamically changes every process address at every memory reference
  - Process/compiler generates **virtual** (**logical**) addresses (in their address space)
  - Memory hardware uses **physical** or **real** addresses



```
0x10:  mov  edi, [0x4+rbp] ; # of memory accesses here?
0x13:  add  edi, 3
0x19:  mov 0x4+rbp, edi
```

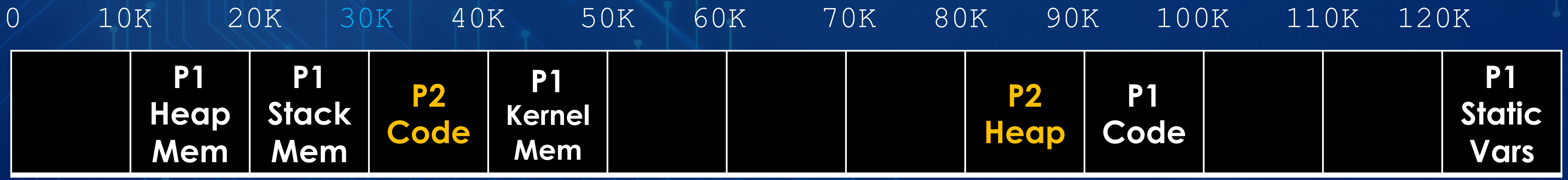
Virtual Addressing

- Review the slides from lecture 10 as necessary.
- A process's virtual address space runs from 0 to 0x7fffffffff.
- MMU translates these addresses to actual physical addresses
  - P2 code starts at address 0x0 – MMU translates that to real address 0x30K.
- Physical addresses are allocated by the OS wherever the OS has space available.

Process 2

Virtual Address	Present	Writable	User	Physical Address
0x0	1	0	-	0x30000
0x1000	0	-	-	-
0x2000	0	-	-	-
0x40000	1	1	1	0x80000
…	…	…	…	…
0x7fffffffff	0	-	-	-
0x8008000	0	-	-	-

Page Table



Physical (real) memory (shown in *Pages*)

Virtual Memory – What's the point?

- Let's reiterate the point of virtual addresses
  - (As opposed to just using physical addresses).
- Program can assume its address space is 0 to 0x7ffff…
  - This makes it much easier to write programs!
- Program can actually "use" more memory then exists… how?
  - Only using some of the memory at any given time.
- Process cannot access a *physical* address that belongs to another process (isolation)
- Processes can share the physical memory of the machine without getting in each other's way.
- Review: What's the difference between Program and Process?

Next Time

- Techniques for Address Translation, such as
  - Base / bound
  - Segmentation

~ *Fin* ~