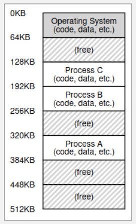
Memory

Supporting Multiprogramming

- *time sharing* means that the currently running process gets the entire resource (CPU, memory) to itself

- time sharing is effective for the CPU
  - saving and restoring registers on each context switch is relatively fast
- time sharing is not effective for memory
  - saving and restoring all of memory to disk is prohibitively slow
  - *space sharing* is needed instead


0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)

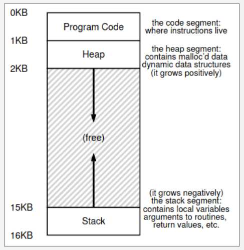
https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf

Goals

- a transparent (i.e. hard to see) system
  - processes should think they have memory to themselves
  - not dependent on the number or location of processes
- protection
  - a process should not be able to corrupt the OS or other processes
  - a process should not be able to read the data of other processes
- sharing
  - cooperating processes can share portions of their memory
- efficiency
  - in time – not taking too long and slowing programs down
  - in space – not too much overhead for supporting virtualization

Abstraction: The Address Space

- each process has a set of addresses (its *address space*) that map to bytes in memory
  - starts at 0
- the address space contains all of the memory state of the running process
  - program *code*
    - placed at the top of the address space because its size is static
  - program call *stack*
  - *heap* for dynamically allocated memory
    - stack and heap are both dynamic – putting them at opposite ends of the address space lets them both grow without having to move

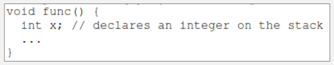

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.

- the OS (and hardware) translates addresses within the address space to physical addresses

https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf

Memory API

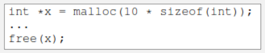
- two types of memory, distinguished by how allocations and deallocations are managed
  - *stack* memory is managed by the compiler
    - memory usage known at compile time
  - *heap* memory is managed by the programmer
    - memory usage is only known at runtime



```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

the compiler allocates memory on the stack when `x` is declared

the memory is deallocated when `x` goes out of scope (in this case, when the function returns)



```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

`malloc` results in memory being allocated on the heap

`x` going out of scope on deallocates only `x` itself (the box holding `int *`) – the memory allocated by `malloc` remains until freed by `free`

https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf