Memory Management

Operating-System-Concepts[1]

- **paging** - a memory-management scheme allowing process's physical address space to be non-contiguous.

Linux-System-Programming[2]

- processes do not directly address physical memory
- each process is associates with a (unique) *virtual address space*
- address space is *linear* (start at zero) and *flat*
- **page** smallest addressable unit of memory by MMU (memory management unit)
  - page size is defined by the hardware
  - get pagesize via `/proc`, `getconf`, or syscall `getpagesize()`

    ```
    # cat /proc/1/smaps | grep -i pagesize | head -2
    KernelPageSize:        4 kB
    MMUPageSize:           4 kB
    # getconf PAGESIZE
    4096
    ```

    ```
    #include <unistd.h>
    #include <stdio.h>
    int main() {
       printf("%d\n", getpagesize());
    }
    ```

  - pages are either valid or invalid
    * *valid page* - associated with a page of data (in RAM or secondary storage/swap)
    * *invalid page* - not associated with anything (unused, unallocated address space)
  - **page fault**: - *valid* page but not currently in RAM and needs to be *paged in*
  - **segmentation fault** - caused by trying to access an *invalid* page

  ```
  #include <stdlib.h>
  int main() {
    int *ip;
    ip = (int *) malloc( sizeof(int)*10 );
    ip[100000] = 99; // outside array's 10 elements, seg fault
  }
  ```

- memory regtions (areas/mappings) in every process
  - *text segment* the actual program code, constants etc; ready only data
  - *data segment* or *heap* - dynamic memory (`malloc()`)
  - *bss segment* unitialzed global variables (zeroes)
  - *stack* grows/shrinks dynamically; e.g. local vars, function return data. Thread: one stack per thread

  ```
  $ ulimit -Sa | grep -i stack
  stack size                  (kbytes, -s) 8192
  ```

[1] https://codex.cs.yale.edu/avi/os-book/
[2] https://www.oreilly.com/library/view/linux-system-programming/9781449341527/

- **dynamic memory** - allocated at runtim
  - `void * malloc (size_t size);`
  - `calloc()` - zero's out bytes ('c'=clear)
  - `realloc()` > if unable to enlarge the existing chunk of memory by growing the chunk in situ, > the function may allocate a new region of memory size bytes in length, > copy the old region into the new one, and free the old region
- **anonymous memory mapping**
  - for large allocations, glibc doesn't use the heap; instead creates an anonymous memory mapping
  - `mmap()` creates a memory mapping and the system call `munmap()` destroys a mapping
- **advanced memory allocation**
  - `mallopt()` - sets the memory-management-related parameter e.g.
    - `M_MMAP_THRESHOLD` - malloc allocation request via an anonymous mapping instead of heap
    - default value (Linux) if not set for `M_MMAP_THRESHOLD` = 128k (128*2024)
- **stack-based allocations**
  - one can use the stack for dynamic memory allocations too (as long as it doesn't overflow)
  - `alloca()` system call for dynanimc memory allocation from stack > alloca() returns a pointer to size bytes of memory. > This memory lives on the stack and is automatically freed when the invoking function returns.

    ```
    #include <alloca.h>
    #include <stdlib.h>
    int main() {
      void *p;
      // int size = 8388608; /* will segfault as ulimit: stack size (kbytes, -s) 819
      int size = 8388608 / 2;
      p = alloca(size);
      // free(p); /* will result in Aborted (core dumped) */
    }
    ```

  - very fast compared to `malloc()` (just increases stack pointer)
- **locking memory** - two situations you might want to change default paging behaviour:
  - (1) *determinism* - page fault are costly b/c of I/O operations (disk)
  - (2) *security* - if secrets kept in memory are paged out to (unencrypted) disk
  - `mlock()` - to locks a specific memory area/allocation, unlock with `munlock()`
  - `mlockall()` - locks the whole process space, unlock with `munlockall()`