Ch6: Processes

Processes and Programs

A process is an instance of an executing program.

A program is a file containing a range of information that describes how to construct a process at run time.

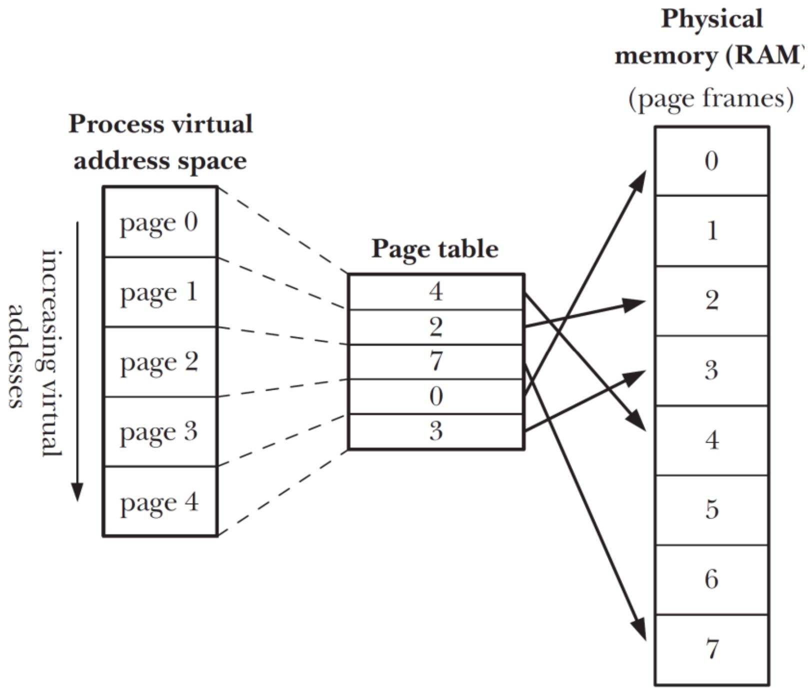
Information exists in a program file:

- Binary format identification (a.out, COFF, ELF).
- Machine-language instructions.
- Program entry-point address.
- Data (initial values and literal strings).
- Symbol and relocation tables.
- Shared-library and dynamic-linking information.

Process components

- **A process consists of:**
  - User-space memory containing program code and variables.
  - kernel data structures that maintain information about the state of the process:
    - Process IDs (/proc/sys/kernel/pid_max, getpid(), getppid()).
    - Virtual memory tables.
    - table of open file descriptors.
    - Resource usage and limits.
    - Others.

Virtual Memory Management



Locality of reference: a typical property of most programs.

- Spatial locality: reference memory address nearby.
- Temporal locality: access same address in near future.

We can execute a program while only part of its address space in RAM.

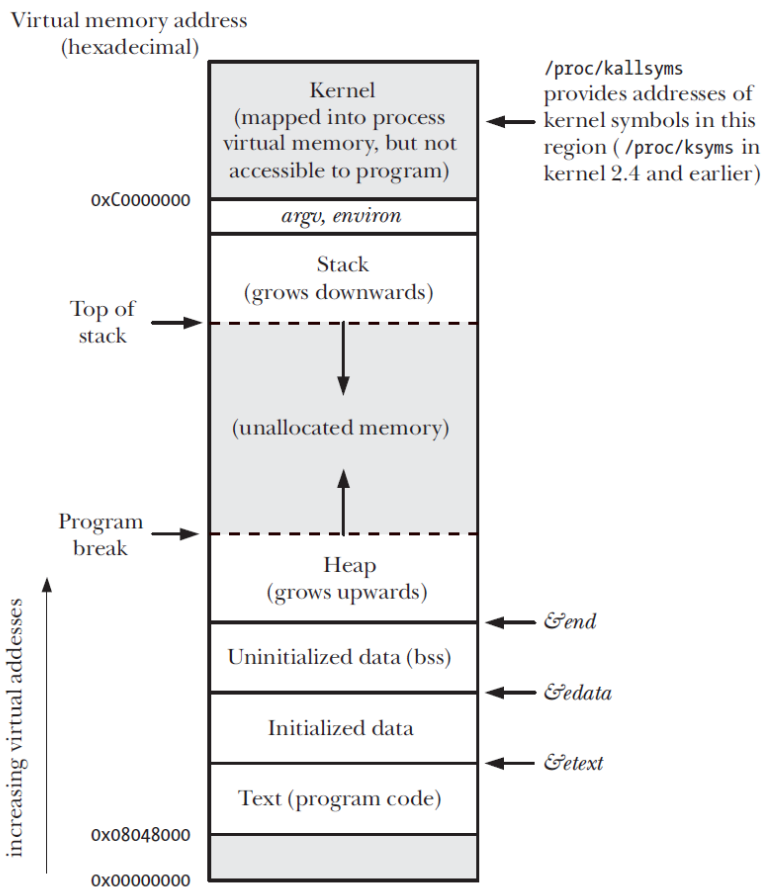
How does it work?

- Memory is divided into pages.
- Kernel maintains a page table for each process.
- Vary over program lifetime (stack, heap, shared memory).
- Page faults.
- Swap area.
- MMU.

Virtual Memory Advantages

- Process isolation (from other process and from the kernel).
- Memory sharing.
  - Executing the same program.
  - IPC.
- Memory Protection (read-only, execute-only, RW).
- Compiler and linker don’t need to be concerned with the physical layout.
- Loading programs faster.
- Better CPU utilization.

Memory Layout of a Process



Process memory layout consists of:

- Text (read-only, sharable).
- Data.
- BSS (Block Started by Symbol).
- Stack.
- Heap (Program Break)

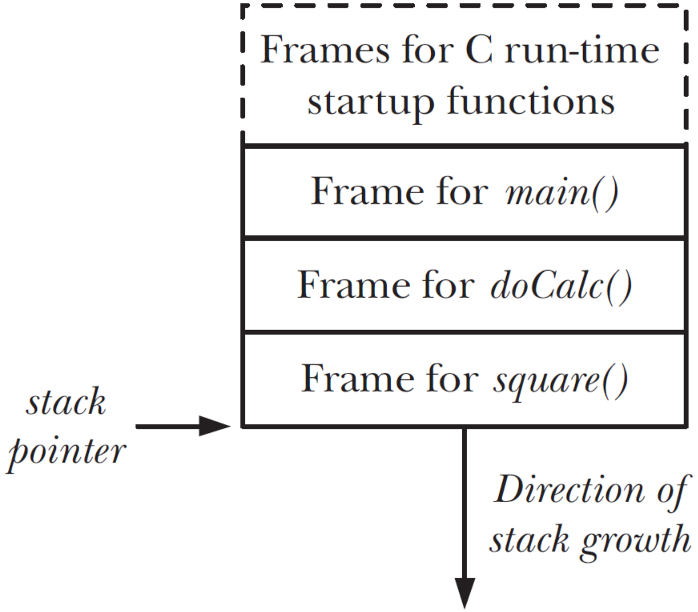
Notes:

→ Size command.

→ etext, edata, end.

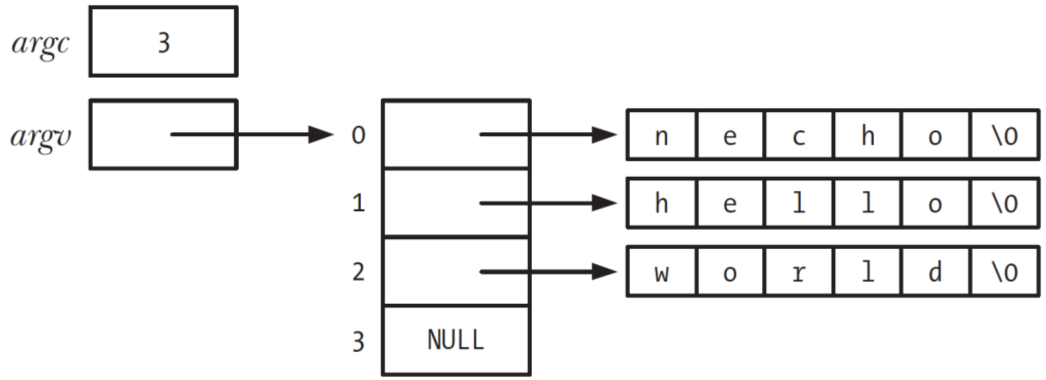
→ Reentrant functions.

Stack and Stack Frames



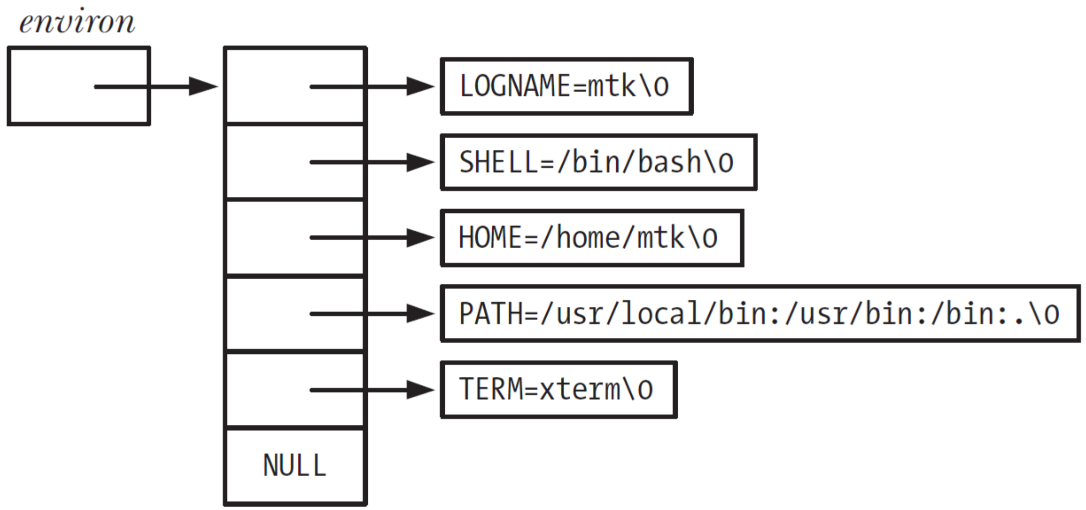
- → User Stack vs. Kernel Stack.
- → Stack Frames
- → Stack Frame contents
  - → Function arguments.
  - → Local(automatic) variables.
  - → Call linkage info.
- → Recursion.

Command Line Arguments



- → Using argv[0] (gzip, gunzip, zcat).
- → /proc/PID/cmdline.
- → program_invocation_name.
- → program_invocation_short_name.
- → Place of argv and environ arrays.
- → getopt().

Environment List



- → Place of argv and environ arrays.
- → One-way, one-time transfer from parent to child.
- → environ variable.
- → It can be passed to main() but with limited scope.
- → /proc/PID/environ.
- → export, printenv, unset commands.
- → Modifying env in C: setenv, unsetenv.
- → Clear env (possible leaks).

Ch7:
Memory Allocation

Heap

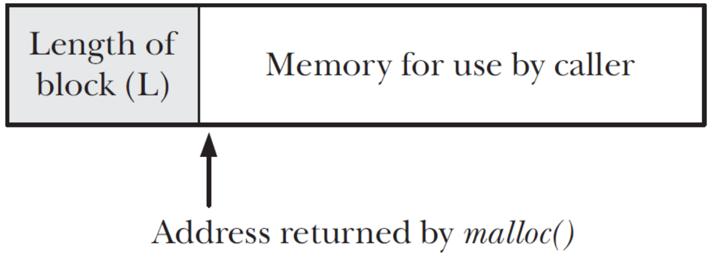
- **The heap** is a variable size segment of contiguous virtual memory that begins just after the uninitialized data segment of a process and grows and shrinks as memory is allocated and freed.
- **Program Break** is The current limit of the heap.
- **Adjusting Program break** using brk() and sbrk(). Kernel allocates pages when the new memory is accessed.
- Program break upper and lower limits (end of data, shared memory, process limits).
- Getting the current program break (sbrk(0)).

Allocating Memory on the Heap: malloc() and free()

- **malloc()** allocates the needed size and may adjust the program break.
- **free()** usually will not lower the program break.
  - Freed memory might be in the middle of the heap.
  - Minimize the number of sbrk() calls.
  - Programs tend to repeatedly release and reallocate memory.
- When a process terminates, all of its memory is returned to the system. Should we call free()?
  - Readability and maintainability.
  - Long-running programs (Daemons and shells).

Implementation of malloc()

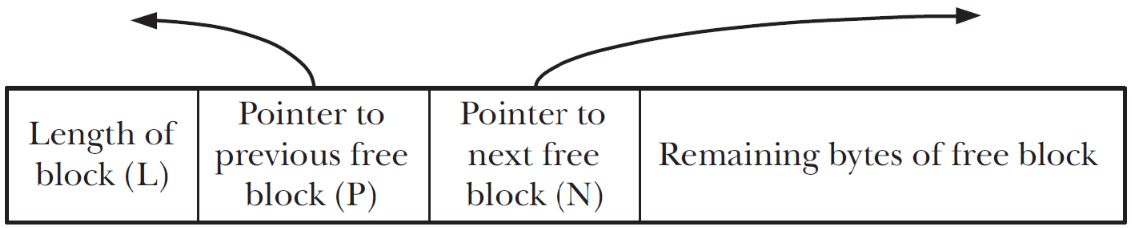
- → Search the free blocks for a good candidate (first-fit, best-fit, etc.).
  - → Split the block if its size is larger than needed.
- → Call sbrk() if no free block matching (with larger size).
- → How free() will know the size of the block?



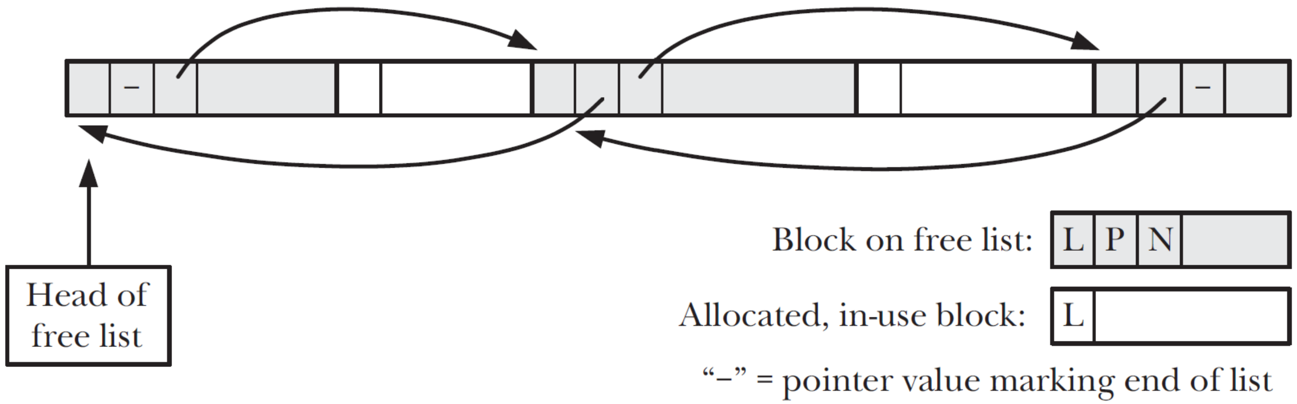
Implementation of free()

→ Get the block length allocated and set by malloc().

→ Construct the free blocks list.



→ The free list will become intermingled with blocks of allocated, in-use memory:



Calloc() and realloc()

- **calloc()** allocates memory for an array of identical items and initializes the memory to zero.
- **realloc()** resizes a block of allocated memory.
  - Attempts to coalesce the block with an immediately following block of memory.
  - Might copy all existing data from old to new block.
  - Must use the returned pointer.
  - Do not assign directly as realloc might fail.

Surviving Rules in Dynamic memory Allocation

- → DO NOT touch any memory outside the allocated block range.
- → DO NOT free an allocated block twice.
- → Free with the same pointer returned from malloc. NOT with offset.
- → Free the allocated memory.

Allocating Memory on the Stack

- → **alloca()**
  - → Allocates memory dynamically on the stack.
  - → Obtains memory from the stack by increasing the size of the stack frame.
  - → DO NOT call free().
  - → Allocated memory is valid only within the function.

Debugging Dynamic Memory Allocation

- → **mtrace(), muntrace(), MALLOC_TRACE**
  - → record tracing information about memory allocation and deallocation.
- → **mcheck(), mprobe()**
  - → Perform consistency checks on blocks of allocated memory.
- → **malloc debugging libraries**
  - → Valgrind
  - → Electric Fence
  - → Insure++
  - → dmalloc