Memory Models

Unix Memory Model

For simplicity, we will say that a processor has a restricted mode and an privileged mode. Some instructions are only available in privileged mode. In fact, some processors are more complicated, but that is sufficient for this discussion. The Operating System code that executes to switch between processes, to handle transmitting signals from one process to another, and to swap memory pages from disk, operates in privileged mode. The code inside a process operates in restricted mode.

Each process is allocated a segment of memory. No process can use any memory that is not in its segment. In user mode, attempts to access memory outside the segment cause a segmentation fault. The operating system uses some of the memory to store the environment. The environment consists of the environment variables and handles for the I/O channels. I/O channels are managed in the operating system. Most I/O channels, be they files, terminal devices, internet connections, etc, require some book keeping to manage the position, cached input and output, etc. This information is stored internally by the OS, usually in a table. Each process using the I/O channel has a handle to the resource. A handle is a token that the process uses to tell the OS which resource it is referencing.

C Memory Model

Aside from the memory used by the system for the environment, C divides the memory into three parts: the Code, the Stack, and the Heap. Most modern languages use this division of memory. The Von Neumann architecture calls for the same memory addresses to be used for both code and data. This leads to the Von Neumann bottleneck – the fact that instructions and data must be moved through the same data bus to the CPU. However, this well-documented flaw has not stopped almost all computer manufacturers from adopting the Von Neumann architecture. Some programming languages do not use this architecture, like ml or stackless python, because their stack frames do not obey a stack discipline. Usually, these languages are designed for massive multi-threading or stored continuations. In a conventional language, like C or Java, each execution thread is given its own stack, but the heap is shared. In stackless languages, the stack is not present and all memory not used to store code is in a shared heap. Instead of stack frames for each function call, activation frames are allocated in the heap in no particular configuration.

Code

The code is maintained on disk as a relocatable image. Most versions of Unix use the Executable and Linkable Format (ELF). This contains information about linkages. Exportable symbols are listed. Unresolved linkages are listed along with the source for the symbols. ELF
also includes a relocation table, a list of the places in the code that must be patched to reflect the actual location of the code. The code is placed into memory by the loader in a two-step process. First the code is linked, so all dynamic linkages are resolved. Second, the code is patched for its particular location in memory. Once these processes are complete, the code is loaded and ready to receive control.

**Stack**
Most CPUs have stack hardware. This allows the CPU to set up a LIFO structure in memory. Each time a function is entered in code, the hardware creates a stack frame (also called an activation frame) on the stack. This frame holds the values of the registers prior to the activation of the function. The stack frame also holds space for local variables in the function. After the function is done, the frame is popped from the stack, thus restoring the registers to their state in the calling program. The return value from the function is stored temporarily in an accessible location in the stack.

Some processors build the stack from low to high memory; some build from high to low memory, but in any event, the hardware creates a LIFO structure. The hardware has some kind of mark to indicate the fullest possible extent of the stack. A request to create a stack frame outside of the bounds of the stack results in a stack overflow error.

**Heap**
The remainder of a process’s memory segment is called the heap. Languages generally implement a heap manager as part of the process. In Java, for example, the heap manager is part of the JVM. In C, the heap manager automatically compiled into programs that use malloc, calloc, free, etc. The heap is not organized in any particular manner. When a request for heap space is received, the heap manager identifies a contiguous block of memory to satisfy the request. Some heap managers will allocate memory in a sequential manner, some have more complex algorithms. When a request for contiguous heap space exceeds the largest block of memory available in the heap, the heap manager will generate an out of memory error. Thus, it is possible to have 2K available in the heap, but not be able to honor a request for 2K, if the heap memory is not available in a contiguous block.

### Example C Program

```
int main(int argc, char* argv[]) {
    a = 3;
    b = a + 2;
    printf("%d", b);
    return 0;
}
```

Prior to running the code above, the following steps are performed

1. The code for the program is put into memory
2. A new stack frame is created on the stack
    a. A slot for the return value
b. A slot for argc and argv

c. A slot for a and b

3. argv is pointed to an array in the heap
   a. The first element of the array points to the name of the program
   b. The second element points to the parameter given

4. argc is set to the size of argv

5. The code for main receives control

These steps create the memory image above. The five slots on the stack comprise the stack frame for main. Other functions will have other frame sizes, but the general layout will be the same. Some computer architectures may lay out a stack frame differently, but those elements are always present along with the saved registers (not shown). Once the memory segment is set-up, control is given to the loaded code for main.

The first instruction “a = 3;” copies the 3 from within the code into the slot in the stack for the variable a, leading to the following memory diagram.
“b = a + 2;” copies the 3 from the stack and adds the 3 from the code.
The next instructions calls the function printf, which causes a stack frame to be allocated. The stack frame must have space for a return value (printf return an int value), the two parameters and however many local variables printf uses. The space for the return value is left as is, the first parameter is a pointer to “%d” in the code. The second parameter is the value from the b variable slot, which is currently 5.

Printf executes with those parameters, and returns 1, resulting in the following memory diagram.
Finally, the program puts zero in the return slot and returns control to the Unix.