

# Processes

## 1 What is a Process?

A process is an encapsulation of a program in progress. An OS needs this encapsulation because the OS may have to suspend a program, resume a program, or swap a program out to let another program run for a bit. The process is the data structure that the kernel uses to keep track of a program's state. So, now the question is: what exactly does the OS have to keep track of? Well, there's the actual **code**, which is usually called the **program text** when it's loaded in memory. Then there's the state of all the registers on the processor (including the program counter, called the IP on intel), this is known as the **processor state** or **hardware context**. What's left? All the stuff in memory, so the memory state also needs to be saved. Memory itself is divided into two main chunks: **the stack** and **the heap**.

### 1.1 Program Text

The text is actually the memory that contains the compiled code for the program. These are the actual instructions that the processor executes. Because the text shouldn't be changing (i.e. self-modifying code is a no-no), you can actually share program text between multiple instances of the same program. For instance, if you're logged into a big multi-user server and everyone is using emacs to edit their code, the system will actually only have to load one copy of the emacs code even though each emacs instance is a separate process. This is fine, because the memory containing the program text is shared out (using virtual memory tricks) in read/execute mode. This means that a rogue process can't mess with the code (which would be bad), and it means that the OS saves memory by sharing common code (a good thing).

### 1.2 Hardware Context

The hardware context is the programmer visible processor state. Most processors have a bunch of registers, but only a subset is available to instructions running in user mode. The kernel has access to all the other registers, but the user code can only write a subset, and read a slightly larger subset (usually). The specific contents of the hardware context varies from platform to platform, but it usually consists of:

1. General Purpose Registers
2. PC (IP on intel)
3. Memory registers (segment registers, mmu settings registers, etc)
4. Debugging registers (not available on all architectures)
5. Some portion of the Processor Status Word, sometimes (condition codes, usually).

Most hardware platforms provide some handy supervisor-only registers that make backing these things up simple. Some architectures even include special instructions to dump programmer-visible state to portions of memory (intel systems have this, and Linux uses it heavily).

### 1.3 The Stack

We've talked about the stack before, but as my mother always said, you can never know enough about the stack! The stack is an old programming language technique to elegantly deal with function scope and linkage. Abstractly, when a function is called, it comes with a new scope. Local variables declared in that function are available to that function and inner scopes, but not the function that called it (in C, as in Java, new scope is usually encompassed with curly brackets `{, }`). Additionally, because any function can be called by any other function, you need some way of keeping track of who called you so that you can return to the appropriate function (at the hardware level, this is called the return address). Both of these problems can be handled elegantly with a stack. Abstractly, there is a stack of function contexts somewhere in memory. When a function is called, a new function context is pushed to the top of the stack. This context has the appropriate return address loaded into it by the calling function. The calling function also loads copies of the function arguments into the context. Then the calling function transfers control to the called function. The called function can allocate local variables on the stack. If you need an `int`, you just push one on the top of the stack. Since the stack can keep growing 'up', you basically have unlimited space. When the called function needs to return, it pops its context from the top of the stack and THEN jumps to the return address. In actuality, the stack is just a region in memory. The first difference, is that rather than the top of the stack growing 'up' towards larger addresses in memory, the stack grows 'down' towards smaller addresses. This is done purely for efficiency reasons (although it may have been done to make OS professor's jobs harder :). Additionally, the stack has structure, but it's structured like a C struct, rather than a Java object. So, a stack frame is divided into regions of bytes that hold various values. Here's a stack frame layout for a function, in C defined as: `int strcmp(char* str1, char* str2, int n)`

So, the stack holds the function arguments, the return address, local variables, and the stack pointer. What is the stack pointer? The stack pointer is the value of the top or the bottom of the caller's frame (it depends on the architecture and/or OS). The stack pointer is used to address stuff in the stack and is restored to the processor when the function returns (this is how you pop a frame off the stack). You push a new frame on the stack when you call a function, and you pop your frame off the stack just before you return. Therefore the stack takes care of: local variable allocation and deallocation, function argument passing, and function linkage. From the point of view of the process, the stack contains the current function-calling history of the program as well as the local state of all running functions.

### 1.4 The Heap

The heap is where all of the data goes that doesn't fit into the stack. Every time you call `malloc`, you've been allocating something in the heap. In Java, every time you call `new` you allocated something in the heap (modulo JIT optimizations). The heap is for allocating and storing all the

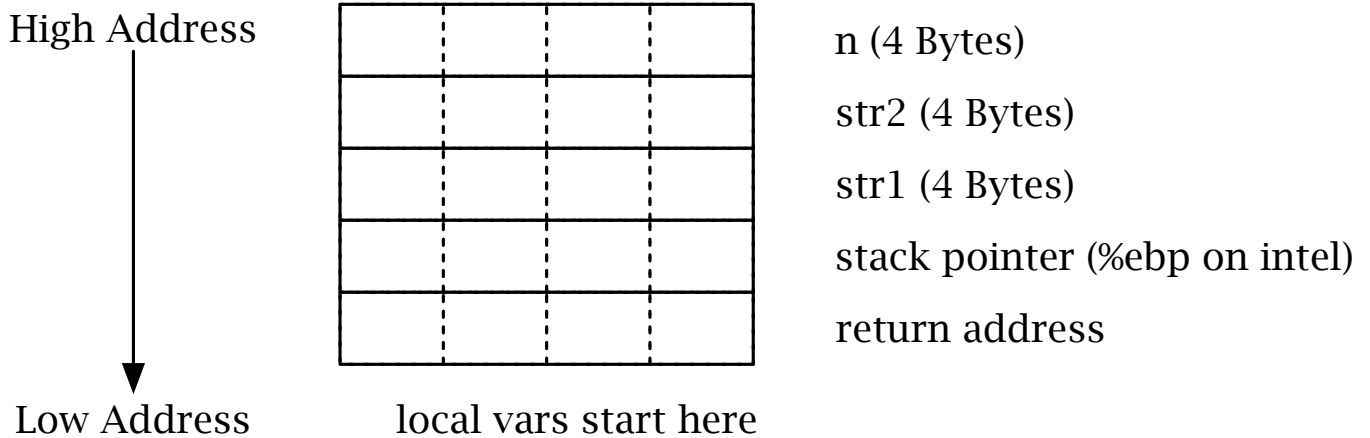


Figure 1: An Example Stack Frame

highly dynamic data. This is data whose total size is unknown at compile time. Dynamic data structures that grow and shrink (a binary tree that you're inserting and deleting from, for instance) are allocated in the heap. Because the heap is for dynamic allocation of data, it is as unstructured as the stack is structured. The stack grows down from the top of memory, and the heap grows up from the bottom. What this means is that if the top of the stack meets or passes the top of the heap, you've run out of memory!

## 1.5 Resources

Resources encapsulates all of those hardware resources abstracted by the OS and exported to user space. An open file is a kind of resource. So is a network socket. Processes request resources from the OS, and the OS is responsible for allocating them fairly among requesting processes and for reclaiming them when processes die/exit. The OS maintains a record of the currently allocated resources so that it can reclaim them on process exit and so that the OS can make sure that the process doesn't exceed resource limits. For instance, UNIX systems usually place a hard limit on the number of files that a process can have open at any given time. This is done because when one process has opened a file that excludes other processes from accessing that file (modulo standard reader/writer semantics). These limits prevent one process from starving other processes from access to the filesystem.

## 2 Threads and LWPs

Traditionally processes were defined as a stream of control and data. However, modern processors encourage multi-threading, which is basically multiple streams of control over the same shared data. Each separate stream (or thread) of control is called a thread. Threads within the same

process share the same code and the same data (heap and static/global data). However, they each are executing their own path through the code, so each thread needs its own stack. Threads are usually mapped onto an OS structure called a light-weight process or LWP. A normal process is then considered to be a heavy-weight process. A modern process is then an address space and a resource table plus one or more LWPs. An LWP is then all the rest of the process components, namely: a stack, a hardware context, and a status (more on status later). LWPs are great for multi-threaded code because they allow the OS to swap running threads without having to perform a full-on process swap (essentially, they eliminate the need to swap address spaces which can have large overhead).

### **3 An Example**

Here's a simple example, consider a chess game. Assume that it's graphical game, so there's some window that contains a graphical representation of the game in progress. The process consists of the program code/text, the process status (currently running, say), the heap which would contain a representation of the current state of the board and multiple threads (LWPs) that share this representation. What would the threads be doing? This system can support at least two threads. One thread would be doing the actual chess computation(figuring out which move to make next), the other thread would be responsible for converting the chess-board data structure into a graphical representation to display on the screen. It turns out that you can add many more threads. Chess can be parallelized fairly simply. You can turn a chess game into a tree of moves by either side. This tree is gigantic (on the order of  $10^{40}$  nodes), so you can't fully explore it. However, you can search ahead a few moves to get an idea of which moves would be better. Rather than just iteratively stepping through each move in a loop, you could have several threads exploring the tree simultaneously. As long as the threads aren't evaluating parts of the tree that another thread already visited, you can increase your throughput.