Memory Leakage

The concept of memory leakage is not uncommon. It occurs when a program allocates memory to itself (to perform a specific task) but never deallocates it[1]. Thus, what can happen is that all of the system memory can be consumed because an operation being performed is continually having to obtain more and more memory each time the same part of the program has to execute. When this happens, the system may well become unresponsive and depending on the operating system present, may well crash the entire system.

Luckily, this is not always the case. Different operating systems handle memory and process execution differently, and so there have been different approaches in trying to prevent memory leaks. Also, memory management (one possible aspect of controlling memory leakage) can be linked to the way processes are managed by the OS.

Windows

Under Microsoft Windows (NT/2000) memory is mostly allocated via the concept of *virtual memory* (VM). Virtual memory is an area of non-physical memory stored on hard disk to give the allusion that there is more physical RAM available. The concept here is via the use of *pages* which programs use to access and store their data to and from the VM[2]. Virtual memory in this case may well be stored as a swapfile, which contain a number of pages that a process can use.

It has already been defined that *memory leakage* occurs when an application uses up all the memory allocated to it, because memory that is no longer used is never freed. Indeed, when an application is started under Windows what happens is it is allocated as much of the VM as is necessary (usually about 4GB[3]) via "chunks" known as *page frames*. Typically (although this depends on the architecture type) these pages are in chunks of 8kb. So there is a theoretical limit of this much RAM an application can use (clearly 4GB includes what small amount of physical RAM might be present).

Interestingly enough, memory leakage can occur not only via the application itself, but via a number of the key *.dll* files that an application uses. Any application that runs under Windows will use a core set of *.dll* files. If a call to one of these files is frequent, that can push the amount of memory that *.dll* needs to reference over the page size, and memory can be leaked that way. This has an added disadvantage in that many applications can also use the same *.dll* files. This has a knock-on effect of disallowing other applications

1 The assumption here, is of course that memory is handled by the programmer. This assumption is dependent on the programming language used. Some languages (such as C++ and Java) automatically dereference memory dynamically. But for the purposes of demonstration, it is assumed a language such as C has been used, where one must explicitly free up memory that is no longer in use.

2 Tanenbaum, A. "*Operating Systems: Design and Implementation*.", Prentice Hall, 1987

3 4GB because it is defined by the number of bytes in 32 bits, hence 2^{32} (or 4GB – hence why VM is so powerful).

resources to that file, and so can cause issues. Of course, throughout all of these operations, the Windows kernel is continually controlling how the paged requests are handled in the VM area. In theory, the kernel should kill off applications that start to consume a lot of VM, although it is often the case that this doesn't happen, and the whole system crashes.

Linux

Linux's approach to memory management is much the same as Windows. As before, programs (which we'll call *processes*), when they are started are allocated pages via the VM. Unlike Windows, however, Linux uses the concept of a *swap partition* which acts much the same as the swapfile under Windows, but is partitioned off from the disk.

If memory leakage does occur under Linux, it won't usually be for long. The Linux kernel is continuously "monitoring" how memory is allocated to processes and also monitors the state a process is in. If the kernel sees that a process is using too much memory it will kill off the process via a generic concept in the kernel known as the "OOM (Out Of Memory) killer[4]".

The job of the OOM-killer is to establish (via the paged files from the swap partition) which process it is which is using up all of the available memory. Historically, this second guessing has been problematic[5], but is now considered more or less fixed. As soon as the process has been killed, the memory (pages in this case, since the definition of out of memory is for the VM to have been filled) is freed and made available to other applications.

The killing of processes in this manner can have some side-effects, though. Under Linux (and any Unix-like operating system) processes are organised in a hierachy. Thus a parent/child relationship is built up, with processes being parents and children of each other. If a process is scheduled by the OOM-killer to be killed, it is the responsibility of the *parent* process to ensure this is done correctly[6].

Sometimes though, if the process has to be killed directly, the process might not be killed off entirely. When this happens the process enters into an *uninterruptable state* (also known as state 'D' (D for dead). This is where the process has effectively died, but the memory is never freed. In such cases, the only way to ensure the memory is freed is for the *parent* process to be killed off as well[7]. This can cause problems since if that parent process had other child processes attached to it, they would also disappear and stop running, even though they might not have been affected by the OOM condition in the

4 http://www.rossfell.co.uk/~rickp/oom/
5 http://kerneltrap.org/node/view/142
6 Such instances are handled via signals (man 1 kill) – Unix man page, reference.
7 Note that a process can enter into a "D" state for other reasons besides being out of memory. The assumption shoud not be made that getting rid of a "D" process always implies killing off its parent.

first place.

The termination of processes however is vital. It's done primarily for the kernel's own self-preservation. If the kernel realises that it is going to run out of memory and hence cannot function it will do what it can to avoid that. Hence this design strategy differs from windows in that the system is allowed to continue to have its memory used until it crashes.

Mac OS

[** Not sure what to say about this – OS X seems to use the same memory management (and hence leakage fix) as Linux does. **]