# **Unix Implementation**

## **Process Control**

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no functional benefit from shared-text segments. An efficiency benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the text table. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A diagram of a data structure

Description automatically generated

## **Process creation and program execution**

Processes are created by the system primitive fork. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the fork are truly shared after the fork. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may wait for the termination of any of its children.

A process may exec a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an exec does not change processes; the process that did the exec persists, but after the exec it is executing a different program. Files that were open before the exec remain open after the exec.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply execs the second program. This is analogous to a ‘‘goto.’’ If a program wishes to regain control after execing a second program, it should fork a child process, have the child exec the second program, and have the parent wait for the child.

## Synchronization and Scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent’s process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra’s P and V synchronization operations,2 in that no memory is associated with events. Thus, there need be no allocation of events prior to their use. Events exist simply by being used.

**On the negative side**, because there is no memory associated with events, no notion of ‘‘how much’’ can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

**SOURCE**

“UNIX Implementation” -K. Thompson