Literature Review for "The Nucleus of a Multiprogramming System"

Luke Levesque
COP 5611

1. Problem Statement

Designers of new software are often stuck with the operating system (OS) that comes with the computer they are developing for. This can be problematic if the OS does not behave in a way or offer services that the designer or software developer wants. Examples of this include modes of scheduling (real time or batch, for instance) and resource allocation. [NMS]

"When the need arises, the user often finds it hopeless to modify an operating system that has made rigid assumptions in its basic design." Often, OSes are tied to a particular piece of hardware, so changing platforms might make the problem worse. Likewise, installing a different OS on the same hardware, if one is even available, would be difficult, costly, and could introduce compatibility issues with existing software or hardware. [NMS]

2. The Solution

The solution created, the Nucleus, was designed very simply: "Our basic attitude during the designing was to make no assumptions about the particular strategy needed to optimize a given type of installation, but to concentrate on the fundamental aspects of the control of an environment consisting of parallel, cooperating processes." [NMS] In other words, the Nucleus was designed to support only the most basic parts of an operating system, leaving the rest of it to be implemented as a program rather than in the kernel. In a way, the Nucleus is a sort of microkernel.

The Nucleus is described below and partitioned into various aspects of its operation. Process communication (an important feature of the Nucleus) is described first, followed by the processes themselves and some details of implementation.

2.1 Process Communication

Semaphores were considered for process communication, but it was decided they were too insecure. A badly written program could cause deadlocks and other problems using semaphores. Therefore, the only process communication and synchronization method available is messaging. [NMS]

"The four primitives of message communication are as follows:

send message (recv, msg, buf)
wait message (sender, msg, buf)
send answer (result, ans, buf)
wait answer (result, ans, buf)" [NMS]

Send Message() takes the "first available message buffer from the pool" and copies the message into it. The message is then put into the queue of the receiver and the caller may continue execution. *Wait Message()* puts the calling process to sleep until a "message arrives in its queue" (if a message is already in the queue, the process immediately gets it without sleeping). The calling process, when a message is available, is made active again and gets the name of the sender and the contents of the buffer. The process can then deal with the message as needed.

Send Answer() will reply to a message sent with Send Message() and reuses the buffer returned from Wait Message() to conserve resources. An answer cannot be sent unless the process has received a message previously that has not yet been answered (you cannot send more than one answer with the same message) and typically every message has an answer. The caller continues immediately after the answer copied. *Wait Answer()* puts the calling process to sleep until an answer is available in the buffer given (this works much like Wait Message()). When the answer has been processed, its buffer is freed automatically. [NMS]

The Nucleus checks buffers to make sure faulty programs do not overrun or misuse them and a per-process limit on active messages is set to insure that a single process cannot use up the entire buffer pool. While messages are put into a queue on the receiving process, the process may elect to store the buffers in another data structure after it has received them (such as being sorted by process name or priority). It may then send answers in any order that it wishes. The Nucleus has procedures to handle situations where processes were removed when messages are pending; typically a dummy 'answer' is sent back to let the other process know what has occurred or a message is ignored entirely. [NMS]

An example of using messages instead of semaphores is presented below. The algorithms are implementing the classic producer-consumer problem using three processes: producer, buffer, and consumer. Note that significantly more code would be required to add better error handling or the ability to stop processes when the producer is finished.

```
PROCEDURE Producer()
        LOOP
                create item in temporary memory
                send message()  // Sends item to buffer
                wait answer()   // Confirms buffer has the item
                if error in answer, retry or exit
        END LOOP
END PROCEDURE

PROCEDURE Buffer()
        LOOP
                wait message()   // Wait for request from producer or consumer
                if message from producer
                        put item in queue
                        send answer()    // Indicates success / failure of queue insertion
                else
                        get item from queue
                        send answer()   // sends next item to consumer
                endif
        END LOOP
END PROCEDURE

PROCEDURE Consumer()
        LOOP
                send message()    // To buffer, make request
                wait answer()      // Has item in message buffer
                process item
        END LOOP
END PROCEDURE
```

2.2 Processes

There are actually two types of processes in the system: internal and external. Internal processes are the typical process found on other operating systems. That is, they are programs in execution. An external process references a block of data (registers, files, or a TTY, for instance) on a peripheral device, known as a document. Internal processes can then use this external process, which has a unique ID, to access whatever document it has opened for them. Processes communicate via messages to one another and it is the responsibility of the Nucleus to insure these messages are delivered. [NMS]

2.2.1 Internal Processes

Internal processes may be created, controlled (started or stopped), and removed. These processes are created by other internal processes (there is a process, S, that is created at system startup to handle any initial process creations). The internal process that called for the creation of another internal process is known as the parent. Parents own the resources (such as memory) of all child processes and therefore are responsible for the allocation or multiplexing of those resources among their children. Child processes can only use resources owned by their parent. [NMS]

One of the resources shared is processor time. Though the Nucleus services all active processors with round robin scheduling [NMS], parents may stop individual child processes so that other processes will get more CPU time, thus allowing a crude implementation of a priority system. As an example, if process B has priority over process C in a particular operating system (parent process) A, then A may choose to let C be active only every third iteration of the round robin scheduling while B is always active. This would result in B getting more processing time than C.

Another positive side effect of the parent/child relationship is that parents may do per-process swapping. This allows parents to have more processes than their memory allocation would normally allow, providing that not all processes are active at once (because one or more of them would be swapped out to a backing store). [NMS] Note that this does not allow a process to have more memory than the parent owns and that per-process swapping is not virtual memory. This is because the machine (the RC 4000) the Nucleus runs on was designed for real time applications and therefore there is no hardware support for virtual memory [Supplemental].

2.2.2 External Processes

External processes are used to communicate with the 'outside world', such as printers, TTYs, disk drives, and the system clock. Each external process executes code (the device driver) in the Nucleus that allows it to communicate with the device it needs to access. The external processes are created by the request of internal processes and have a unique name. The internal processes use the standard messaging system as described in section 2.1 to communicate with external processes in order to read or write data. [NMS]

Each external process typically handles one document. Documents can be anything from registers in a device, individual files on a disk or drum, or even a TTY. Internal processes may request exclusive access to a external process to insure a particular document is not accessed by

anyone else. If more advanced access methods are desired (such as disk scheduling or access control), an internal process can take the place of an external process by having the same or similar name. [NMS] Messages (I/O requests) can then be sent to the internal process, where it will do processing and either handle the I/O itself or send it to the real external process with which it has exclusive access.

The TTY is the only external process that can send a message to an internal process (the other external processes can only send answers and receive messages) and is used to start programs or enter commands by sending messages to specific internal processes. As mentioned previously, files are also represented as external processes as are other related devices such as tape drives. Finally, as an example to show how powerful this paradigm is, an internal process can message the clock external process with a time delay as the contents. The clock will send an answer back when the time has elapsed, allowing a form of process synchronization. [NMS]

2.2.3 Process Hierarchy

Processes are arranged in a hierarchical (tree) fashion, with parents creating their children. The most important feature of this hierarchy is that parents retain complete control over their children. The hierarchy can extend as far down and across as needed, with children being the parents of other processes. This allows parents to control all resource allocation (such as memory) and the general process status (start, stop, swap in and out) of all children, including descendants of children, created. This means that children cannot use resources outside of the parent's control (I.E.: If the parent has 32KB assigned to it, a child process cannot allocate 48KB). Removing a child releases the resources back to the parent. The hierarchy does not explicitly control CPU time usage, as that is done with round robin scheduling for all processes. See section 2.2.1 for a technique to emulate scheduling priorities. Also, messages may be sent to any process in the tree, regardless of what parent owns it. [NMS]

In order to initialize the hierarchy, the Nucleus auto-starts a process known as S, which is a very basic operating system and the root of the tree. S can be messaged by a TTY and be told to start other processes - the real operating systems. This allows new operating systems to be created as programs and started and stopped as needed without any modification to the Nucleus, allowing for debugging or upgrades without bringing down the whole system. "Multiple operating systems can even be active simultaneously." Interestingly enough, this allows for "user programs to run under different operating systems if they have a common set of interfaces (messages)." [NMS]

2.3 Implementation

The Nucleus was implemented successfully on the RC 4000 computer, a 24-bit word system. The system supported a "real-time clock, TTYs, paper tape in/out, printer, tape, and a backing store." [NMS]

The Nucleus and it's supporting software (external process code, S, etc.) consume about 18.5KB when running, leaving the rest of memory for user programs and external processes. On average, either of the four messaging primitives take about 0.5 msec to execute. The Nucleus operations taking the most time are process creation and removal because each word allocated to a process must have a protection bit set individually. [NMS]

3. Critique of the Solution

While the solution likely helped pave the way for the modern microkernel, it suffers from several significant problems. A major flaw is messaging security: any process can message any other process in the hierarchy, which could cause many undesired effects. Another security issue involves the creation of external processes: any internal process can create an external process, implying there are no file permissions, etc. It also seems that the round robin scheduling is a little inflexible. Perhaps more control of process scheduling could be delegated to the direct children of S (each parent could dole out CPU time between its children rather than each child being given CPU time directly by the Nucleus) to increase flexibility. A major weakness is that the lack of virtual memory limits the usefulness of the operating system. The main reason for this is because the RC 4000 computer is designed for real time applications and thus has no virtual memory [Supplemental]. It's possible that implementation on a different computer platform would solve this issue. Finally, another issue is the per-process message limit. If all processes messaged the same process, one that was not calling Wait Message(), then it is possible the system may run out of message buffers and be in a sort of deadlock situation. Perhaps the message limit could be for the receiving queue of each process instead to avoid the deadlock.

4. Conclusion

The Nucleus places only basic operating system services in the kernel, allowing for the system to be easily extended with one or more 'real' operating systems. Because the processes are in hierarchy form, parents have complete control over children, thus making the parents operating systems. This arrangement allows for multiple active operating systems in memory and the ability to start, stop, and change them without affecting the system globally (no reboots or modifications to the Nucleus are needed). The process scheduling scheme used is round robin.

The processes described above are known as internal processes. The other type of process, known as external processes, are different in that they are not part of any hierarchy (flat structure) and are used to interface with the outside world (files, tape drives, real time clock, etc.). They can be created by any internal process and are communicated with using messages, described below.

Processes may only communicate using messages. The Nucleus imposes an active message limit for each process to avoid excessive resource consumption. The receiver of a message may have an answer sent back to the initiator if desired, reusing the same buffer to conserve resources.

5. References

[NMS] Per Brinch Hansen., **The Nucleus of a Multiprogramming System.** Communications of the ACM 13(4), April 1970. Pp. 238-241, 250.

[Supplemental] P. Brinch Hansen, **The RC 4000 real-time control system at Pulawy**. BIT 7, 4 (1967), 279-288.