Interprocess Communication

Peter J. Denning
May 2019

When you launch a program, the OS creates a process for you that encapsulates your program and data in its own address space. The OS ensures the new address space is completely isolated from all other processes. As you have seen, the OS also provides dozens of *dæmons* -- service processes that run in the background taking care of tasks for Internet, printing, graphics, databases, and more. To cooperate, these otherwise isolated processes must communicate. To do that, they need a message-exchange medium that works outside their individual address spaces. To accomplish this the OS provides *interprocess communication* (IPC), the means for these client and server processes to make their requests and responses. The IPC level allows communications between processes on the same machine or across the network. All the kernel levels above the IPC level use IPC to make requests of service processes on local or remote machines.

A special case occurs when a group of processes on the same machine want to share data. Each process's page table contains a page pointing to a single shared page frame. Reading or writing that page can be done with normal virtual memory operations. The content of that page is visible in all the participating processes. While tempting with its apparent simplicity, this is difficult to use and very limiting to the messages exchanged within the process group. A much better approach is to design a message system that is completely outside every address space, making it easier to guarantee delivery and prevent tampering.

We will discuss four models for interprocess communication. The shared page model allows the operating system to let processes communicate by reading and writing a page shared among their virtual memories. The messages model assumes every process is by default a service process with a built-in request queue; the process takes the messages one at a time and responds to the service requests they contain. The Internet model uses the Internet protocol stack to send messages to processes on any host machine in the Internet. The remote procedure call (RPC) model is similar to the Internet model but with an interface that appears the same as a local procedure call.

Protected Service Processes

Many processes in the operating system are service processes: you send them a request and they respond with an answer after performing the requested service. This is such a common pattern we have a name for it: *protected service process*. We add the stipulation "protected" because the data and hardware resources needed to provide the service are confined to the address space of the service process and cannot be accessed without making a request of the service process. A service process has an input queue for receiving requests; each request is stamped with the process ID of the sender. For example, a disk service process would have an input queue in which any other process can deposit a read or write request; the disk service takes the requests one at a time, interacts with the disk itself, and responds to the sender when done. A process needing a disk transaction cannot access the disk device driver directly: it must ask the disk service process to do so. This protects against hanging the system due to improper use of the device driver interface, and against race conditions when multiple processes make disk requests in parallel. The protocol for interacting with a service process contains a number of interesting synchronization issues.

Protected Service Process Protocol

A process `P` interacting with a service process `S` follows the protocol outlined in the pseudo code below. The input queue of `S`, `inqueue[S]`, is protected by two semaphores

```
insem[S], initially 0 (queue length)
mutex[S], initially 0 (mutual exclusion)
```

The count of `S` is the number of entries in the input queue `inqueue[S]`. The operations for manipulating the input queue when `P` is the caller are:

```
insert(R,S) = append (P,R) to the end of inqueue[S]
(P,R)=remove() = remove and return the first request of
inqueue[self]
```

The pseudocodes for the interaction between processes P and S are:

```
P: loop {
       do some computation
       prepare service request R
       wait(mutex[S])
       insert(R,S)
       signal(mutex[S])
       signal(insem[S])
       SLEEP
       }

S:  loop {
       wait(insem[self])
       wait(mutex[S])
       (P,R)=remove()
       signal(mutex[S])
       start device with parameters R
       SLEEP
       WAKEUP(P)
       }

DCI: WAKEUP(S)
     return
```

The user process `P` sends a request to the service process and goes to sleep until the service process awakens it. Adding the request to the input queue must be protected by mutual exclusion.

The service process `S` waits for a request at its homing position. It removes the request and activates the service device with parameters `R`. (Its access to its input queue is regulated by the mutex semaphore to prevent racing with processes placing requests.) `S` then goes to sleep until the device completion interrupt awakens it.

The service device is activated by a hardware signal transmitted by the device driver contained in `S.` When done with the specified transaction, it awakens the service process by sending the device completion interrupt.

Once `S` is awakened, it awakens the process `P,` which can then continue.

This protocol cannot deadlock because the processes involved wait in the priority order `P, S, D`; no circular wait is possible.

Messages Model

The messages model standardizes the service process model by treating every process as a potential service process. By default every process has a request queue as in the protected service process pattern discussed above. A user process calls on a service process by sending it a request message as above. The service process can prepare a response to the caller and send it in a return message that appears in the caller's input queue. This interaction with messages simulates a traditional procedure call.

The operations of the user process protocol above can be combined into a "local service process call" in the kernel

```
B = LPC(S,A)
```

which means: "send request message `A` to `S and` return its response `B`." This model is restricted to sending and receiving messages from processes on the same system (managed by the same kernel). To send and receive messages with remote processes, the Internet or RPC model must be used.

Shared Page Model

The shared page model allows the same page to appear in multiple address spaces. This is accomplished by putting a pointer to the shared page-frame in each of the page tables referring to that frame. Those page table entries can have different page numbers in each address space. When the MMU maps a reference to the shared page, it accesses the shared frame of memory.

The interface to the kernel for shared pages reveals the complexity inherent in this model. First of all, the OS needs to know exactly which processes share a given page; otherwise it cannot inform them if the contents of the shared frame move to a different memory location or are paged out from inactivity. To do this, a parent process sets up the group of child processes with a kernel call

```
g=CREATE_GROUP(P1,P2,…,Pn)
```

where `g` is a group code and `P1,P2,…,Pn` are the members; the OS also sets aside a page frame `f` of memory for the group's shared page and a mutual exclusion semaphore `s` to be used by group processes reading or writing the shared page. Then each member can request a shared page with the kernel call `s=JOIN_GROUP(i,g)`, which asks the OS to make the caller's page table entry `i` point to `f` and return the name of the group's mutual-exclusion semaphore `s`. After all the group members have set up their pointers to the shared page in this way, they can now share data by reading and writing it in the shared page as if it were like any other page it its address space.

The OS must provide a way for any child to leave the group, for example,

```
EXIT_GROUP(g)
```

which deletes the calling process from the group and removes the link from its page table to the shared frame. Similarly, the parent can dissolve the whole group and release its shared page and semaphore to the OS with

```
DISSOLVE_GROUP(g)
```

Once this is all set up, the dynamics of using the shared page can be quite tricky and confusing to users. The first issue is preventing race conditions as different, concurrent processes read and write the shared page. The mutual exclusion semaphore is the main tool. A process accessing the share paged locks it first and releases the lock when done. Obviously this creates a potential performance bottleneck if the lock can be set for a significant period of time.

The second issue concerns keeping all the processes in the group properly informed if the OS decides to relocate the shared page to a new frame or delete it entirely from main memory. To properly execute any manipulation of the location of the shared page, the OS must access each page table in the group and update the frame pointer in the entry pointing to the shared page. All the processes in the group must be suspended until that update is completed. This could incur a significant overhead and performance penalty for the group.

A third issue concerns overflows on the shared page. What happens if a process tries to copy N bytes into the shared page when N is larger than the page size? Here the behavior becomes indeterminate and depends on details of how the operating system and memory hardware work. One possible behavior is that the process copies the excess bytes into the next page of the address space. In that case only the first segment of bytes will appear on the shared page; other processes in the group will not see the remaining bytes. Another possible behavior is that the memory hardware takes over the writing of a sequence of bytes, placing the overflow in the next frame of memory. Unfortunately, that frame is not shared. The overflow bytes overwrite another process's page, causing that process to crash or to receive leaked information. Both these cases are unacceptable . The first does not share all the bytes among the group, as claimed by the page sharing protocol. The second overwrites data in another address space, violating the guarantee that address spaces are isolated and cannot leak information from one to another.

Because of these difficulties, the shared page model is not widely used. It was used in operating systems in the 1980s and 1990s under the guise of "multithreading packages", but it fell out of use because it caused many problems and was difficult to work with.

Internet Model

The standard protocols of the Internet are used to send messages to a remote process and receive responses in return. We will not cover the details of theTCP/IP stack here; you can learn about them in a networks course. We will discuss the interface and some of the design issues that the model must deal with. Most OS have a TCP/IP stack interface in them but use an RPC model to hide it from everyone except skilled systems programmers.

The basic network protocols use a datagram model for message passing. A datagram is a single packet sent to the remote host; the datagram contains a complete specification of a request for a specific service. The remote host has a network listener service process that receives datagrams and routes each one to the input queue of the service process for the request encoded in the datagram. When the service is done, it transmits its response to the sender in another datagram.

Here is a simplified view of how the user can interface with communication according to this model. Suppose process `P1` on host with IP address `A1` wants to send message `m1` to process `P2` on host with IP address `A2`. Here `P1` acts as client and `P2` as server. Both `P1` and `P2` are set up as service processes. The notation `A2/P2` means service process `P2` on host `A2`. (In network jargon, the combination `A2/P2` is called a *socket*.) The sender issues a kernel call `SEND_DG(m1, A2/P2)`, which sends a datagram `(A1/P1, m1, A2/P2)` to represent the message "socket `A1/P1` has a message `m1` for socket `A2/P2`." On host `A2`, a listener process receives the datagram and routes it to the input queue of local process `P2`. `P2` performs the task requested in `m1`, encodes its response into message `m2`, and uses the kernel call `SEND_DG(m2, A1/P1)` to respond. The return socket, `A1/P1`, was specified as the source in the first datagram.

RPC Model

The remote procedure call is an interface for the Internet model that looks like a local procedure call. That means that you can interface to local and remote processes with the same interface that appears in your programming languages.

Many OSs support remote procedure call (RPC) as part of their general IPC systems. RPC allows a process to make a standard procedure call, which is then implemented with a message exchange with the service process, which might be on a remote host. When translating a procedure call, the compiler inserts code called a "stub". The stub determines whether the called procedure is local or remote. If local, it uses the LPC to call it. If remote, it uses RPC. The RPC protocol encodes the parameters into a datagram, which it sends to a socket of the remote procedure. The remote procedure sends a return datagram to the caller. You can read about RPC in more detail from online sources.

The implementation of RPC is much more complex than a regular local procedure call because of all the synchronization issues across the Internet. An RPC name service must be available to get a socket number for a remote process given its name. A standard method is needed to encode parameters into the datagrams. The protocols for message exchange must be capable of dealing with lost datagram packets, by using time-outs and resends; they use the TCP/IP system, which already includes this. On receipt at the host of the remote procedure, the incoming datagrams are routed to the input queues of their service processes.

Summary

IPC is an interprocess communication system by which processes with distinct, nonoverlapping address spaces can exchange messages. The most common way to organize this is with service processes. A service process has an input queue that receives all requests for its service. Requests are stamped with the sender's identification so that the service can respond properly. The model of a service process is incorporated into the message model, the Internet model, and the RPC model.

The shared page model is not much used because it is clumsy and error prone.