Chapter 5

Inter-process communication

5.1 Hardware support for synchronization

You should have already seen a number of hardware mechanisms for synchronizing threads. The most basic one available to an OS is to disable interrupts:

\textbf{Algorithm 5.1} Protecting a critical section by disabling interrupts

1: Disable all interrupts and traps
2: Access state in a critical section
3: Enable interrupts

Remarks:

- This technique doesn’t work in a situation with multiple cores or hardware threads running concurrently. Neither does it take into account DMA devices.

- Processes can’t be rescheduled inside critical section. Indeed, this is effectively a mutex on the entire state of the machine.

- That said, inside the kernel on a uniprocessor it is extremely efficient for short critical sections.

To provide synchronization on a multiprocessor in user space, you need help from the memory system and instruction set.

5.1.1 Shared-memory synchronization instructions

Most of this section should be a recap from the Systems Programming and Parallel Programming courses.
5.1. HARDWARE SUPPORT FOR SYNCHRONIZATION

Algorithm 5.2 Test-And-Set

1: inputs
2: \( p \) \{Pointer to a word in memory\}
3: outputs
4: \( v \) \{Flag indicating if the test was successful\}
5: do atomically:
6: \( v \leftarrow *p \)
7: \( *p \leftarrow 1 \)
8: end do atomically
9: return \( v \)

Remarks:

- Plenty of processors provide an instruction for this, or something equivalent like Read-And-Clear.
- Some Systems-on-Chip also provide peripheral hardware registers that function this way as well when you read from them.

Algorithm 5.3 Compare-and-Swap

1: inputs
2: \( p \) \{Pointer to a word in memory\}
3: \( v1 \) \{Comparison value\}
4: \( v2 \) \{New value\}
5: outputs
6: \{Original value\}
7: do atomically:
8: if \( *p = v1 \) then
9: \( *p \leftarrow v2 \)
10: return \( v1 \)
11: else
12: return \( *p \)
13: end if
14: end do atomically
Remarks:

- This is available on most modern processors as an instruction.
- It is strictly more powerful than TAS. In fact, it is as powerful as needed: you can show [HFP02] that any other atomic operation can be efficiently simulated with CAS, though not with TAS.

---

**Algorithm 5.4 Load-linked / Store Conditional (LL/SC)**

**Load-linked**

1. **inputs**
2. \( p \) {Pointer to a word in memory}
3. **outputs**
4. \( v \) {Value read from memory}
5. **do atomically:**
6. \( v \leftarrow \ast p \)
7. mark \( p \) as “locked”
8. **end do atomically**
9. **return** \( v \)

**Store-conditional**

10. **inputs**
11. \( p \) {Pointer to a word in memory}
12. \( v \) {Value to store to memory}
13. **outputs**
14. \( r \) {Result of store}
15. **do atomically:**
16. **if** \( \ast p \) has been updated since load-linked **then**
17. \( r \leftarrow 1 \)
18. **else**
19. \( \ast p \leftarrow v \)
20. \( r \leftarrow 0 \)
21. **end if**
22. **end do atomically**
23. **return** \( r \)

Remarks:

- Also known as “Load-locked”, “Load-reserve”, etc.
- Well-suited to RISC load-store architectures
- Often implemented by marking the line in the cache

---

**5.1.2 Hardware Transactional Memory**

LL/SC can be viewed as providing highly restricted form of transaction (on a single word), which aborts if a conflicting update to the word has taken place during the transaction.
Definition 5.5 (Transactional Memory). Transactional Memory is a programming model whereby loads and stores on a particular thread can be grouped into transactions. The read set and write set of a transaction are the set of addresses read from and written to respectively during the transaction. A data conflict occurs in a transaction if another processor reads or writes a value from the transaction’s write set, or writes to an address in the transaction’s read set. Data conflicts cause the transaction to abort, and all instructions executed since the start of the transaction (and all changes to the write set) to be discarded.

Example 5.6. Intel’s Transactional Synchronization Extensions (TSX) provide three basic instructions for implementing “Restricted Transactional Memory” or RTM: XBEGIN (which starts a transaction), XEND (which commits a transaction), and XABORT (which forces the transaction to abort). There is also XTEST, which returns whether it is executing under a transaction.

If a transaction aborts, the processor rolls back the write set and jumps to a fallback instruction address specified by the XBEGIN instruction, with register information saying why the transaction aborted. This code can then choose to retry the transaction, or so something else (like take out a conventional lock instead).

TSX also provides an alternative to RTM called Hardware Lock Elision (HLE). Under HLE, the code is written to take out locks using atomic instructions as above, but the processor doesn’t actually do this the first time. Instead, it executes the critical section under a transaction, and only if it aborts does it try again, this time really taking out the lock.

Remarks:

- As with LL/SC, HTM is usually implemented using the cache coherency protocol to make lines in the cache as part of the read and write sets. Coherency messages signal remote accesses, which then cause aborts. For this reason, conflict detection is actually done at the granularity of entire cache lines rather than just words.

- As with many speculation-based CPU features, HTM is notoriously difficult to get right. The first Intel Haswell and Broadwell processors to be sold supporting TSX had to have the functionality disabled in microcode after serious bugs came to light.

- There are many things other than conflicts or an explicit instruction that can cause an HTM transaction to abort (false sharing, interrupts, etc.) The “false abort rate” is an important measure of the effectiveness of a HTM implementation.

- There is a limit to the size of read and write sets that can be checked (such as in the L1 cache). If this is exceeded, the transaction aborts. It’s important not to retry a transaction like this, since it’s always going to abort. The abort handling code is therefore usually supplied with information about whether the CPU thinks the transaction should be retried or not.
5.2 Shared-memory synchronization models

We’ll assume you’re already familiar with semaphores (and P, V operations), mutexes (Acquire, Release), condition variables (Wait, Signal/Notify, Broadcast/NotifyAll), and monitors (Enter, Exit).

Our focus here is the interaction of these operations with the rest of the OS, in particular the scheduler. Assuming for a moment a priority-based scheduler (as in Unix, Windows, etc.).

**Definition 5.7. Spinlock.** A spinlock is a multiprocessor mutual exclusion primitive based on one processor spinning on a memory location written by another.

**Algorithm 5.8** TAS-based spinlock

1: inputs
2: p is an address of a word in memory
3: Acquire the lock
4: repeat
5: v ← TAS(*p)
6: until v = 0
7: . . .
8: Release the lock
9: *p ← 0

**Remarks:**

- Spinlocks only make sense on a multiprocessor: if you’re spinning, nobody else is going to release the lock.
- A pure spinlock only makes sense if the duration that any process holds the lock is short, otherwise, it’s better to block.

**Definition 5.9** (Spin-block problem). The spin-block problem is to come up with a strategy for how long a thread should spin waiting to acquire a lock before giving up and blocking, given particular values for the cost of blocking, and the probability distribution of lock hold times.

**Theorem 5.10.** (Competitive spinning): in the absence of any other information about the lock hold time, spinning for a time equal to the cost of a context switch results in overhead at most twice that of the optimal offline algorithm (which has perfect knowledge about the future). This bound is also tight: no online algorithm can do better than a factor of two. Proof: see Anna Karlin et al. [KMMO90].

**Remarks:**

- The proof is subtle (and worth reading!), but the intuition is as follows: in the best case, you avoid a context switch and save time. Otherwise, your overhead is at worst twice as bad as immediately blocking.
5.3 Messages: IPC without shared memory

The alternative to communication using shared data structures protected by thread synchronization primitives is to send messages instead. You’ve already seen this in networking using sockets.

Message passing is best thought of as an abstraction: it’s perfectly possible to implement thread synchronization using only messages, and vice versa. This was famously demonstrated by Hugh Lauer and Roger Needham \cite{LN79}, which showed that the two models are essentially equivalent, but can vary greatly in performance based on the properties of the underlying hardware.

**Definition 5.11 (Asynchronous IPC).** In *asynchronous* or *buffered* IPC the sender does not block, but the send operation instead returns immediately. If the receiving process is not waiting for the message, the message is buffered until the receive call is made. On the receive side, the receive call blocks if no message is available.

**Definition 5.12 (Synchronous IPC).** In contrast, in a *synchronous* or *unbuffered* IPC system, both sender and receiver may block until both are ready to exchange data.

**Remarks:**

- Asynchronous IPC is the model you’re probably most familiar with from network sockets. “Asynchronous” and “synchronous” are heavily overloaded terms in computer science, but here “asynchronous” means that the send and receive operation do not need to overlap in time: a send can complete long before the corresponding receive starts.

- Asynchronous IPC implies a buffer to hold messages which have been sent but not yet received. If this buffer becomes full, it’s not clear what to do: you can discard messages (as in UDP), or block the sender from sending (as in TCP) until the buffer is drained.

- Synchronous IPC, on the other hand, requires no buffering, merely two threads synchronizing in the OS kernel.

- You’ve probably heard of “non-blocking” I/O, which is an orthogonal concept.

**Definition 5.13 (Non-blocking I/O).** *Blocking* communication operations may block the calling thread (such as the asynchronous receive call described above). *Non-blocking* variants of these operations instead immediately return a code indicating that the operation should be retried.

**Remarks:**

- Non-blocking operations can be thought of as *polling*: there’s usually some kind of operation that can tells which potential non-blocking operations would succeed if they were tried right now: see \texttt{select()} or \texttt{poll()} in UNIX.

- You can have *synchronous, non-blocking* operations: the send call only succeeds when the receiver is waiting.
Example 5.14. Unix pipes: Pipes are the more fundamental IPC mechanism in UNIX, and are closely related to \texttt{fork()}; one might reasonable claim that UNIX is any OS based on \texttt{fork()} and \texttt{pipe()}.

A pipe is a unidirectional, buffered communication channel between two processes, created by:

\begin{verbatim}
int pipe(int pipefd[2])
\end{verbatim}

Each end is identified by a file descriptor, returned by reference in the array \texttt{pipefd}. One sets up a pipe between two processes by creating the pipe, then forking the other the process. This is, at heart, how the shell works.

When you create a pipe, you immediately get both end-points in one go. We can make this model more flexible, for example allowing the processes at each end of the pipe to be created independently.

Example 5.15. Unix domain sockets. Like network sockets, UNIX domain sockets can be bound to an address, which in this case is a filename. The filename can then be used by clients to connect to the socket.

\begin{verbatim}
int s = socket(AF_UNIX, type, 0);
\end{verbatim}

This allows us to split up, on the client side, the name of a communication end-point (the filename in this case) from the reference you use to actually send and receive data (the file descriptor you get back from open).

Example 5.16. Unix named pipes. named pipes (also called “FIFO”s) go one step further by allowing both client and server to open the FIFO based on its name (it’s also a special file type). You can create a FIFO from the command line:

\begin{verbatim}
$ mkfifo /tmp/myfifo
\end{verbatim}

5.4 Upcalls

So far, every operation to do with communication that we have seen has involved the sender or receiver (in other words, a userspace process) calling “down” into the OS to perform an operation (send, receive, etc.).

Definition 5.17 (Upcall). An upcall is an invocation by the operating system (usually the kernel) of a function inside a user process. The called function in the user program is called the entry point, or the upcall handler.

Remarks:

- This is the inverse of a regular system call: the OS calls the program. It is a very important structuring concept for systems, and yet not widely known among non-systems programmers. One way to view an upcall is as the generalization of an interrupt.
Obviously, the kernel has to know what to call in the user program, i.e. the address of the upcall handler.

If the OS is running conventional processes, and the process has been preempted when the kernel is entered, this naturally raise the question of what happens to the previous thread context that was saved by the OS.

One approach is to keep this around, and treat the upcall as running in a “special” context which only exists until it returns (to the kernel).

Alternatively, the OS might choose to pass the previously-saved thread context to the user program, in case it wants to do something with it (like resume the thread).

Example 5.18 (Unix signals), are an example of a the first type of upcall mechanism: the user program registers “signal handlers” as functions that can be called to deliver a signal. Unix systems have a fixed number of signal types (see “man 7 signal”). For each signal, an action can be specified: ignore the signal, call a handler function, terminate the process completely, etc.

Remarks:

Signals raise some interesting concurrency questions. Which stack does the signal handler run on, for example?

Another is: what happens if the signal handler issues a system call? Since it’s not really part of the regular process, what happens? It turns out that signal handlers are quite limited in what they are allowed to do. For example, “man 7 signal-safety” will list the system calls that a signal handler is allowed to make, and there are not many of them. They do include signal() and sigaction(), however.

Signal handlers can’t, in general, safely access program global or static variables, since the main process might have these protected under a lock when the signal handler is called. This includes many standard C library calls cannot (including the reentrant “_r” variants of functions like strtok()).

It is possible to longjmp() out of a signal handler (and into the process) if you are careful. It’s a good exercise to figure out what the OS needs to do so that the process keeps running smoothly.

As with all upcalls, what happens if another signal arrives? If multiple signals of the same type are to be delivered, Unix will discard all but one – signals of the same type are basically indistinguishable. If signals of different types are to be delivered, Unix will deliver them all, but is free to do so in any order.

Example 5.19 (Scheduler activations), take the idea of upcalls much further than signals. Every time a process is resumed by the kernel, instead of simply restoring its registers, the kernel upcalls into the process letting it know where the previous execution state has been stored. This allows the process to resume
it, or do something different: the original motivation for this design was to run highly efficient user-level threads that were aware of when the process itself was preempted and rescheduled. Indeed, the first implementations also upcalled the process (on one core) whenever it was descheduled (on another core), just to let it know. The upcall handler is basically the entry point to a user-level thread scheduler.

Remarks:

• Scheduler activations allow a thread implementation that elegantly combines the performance of user-space threads, and the predictability and flexibility of kernel threads, and this is why they were adopted in recent versions of Windows.

• As with signals, what happens if more than one scheduler activation is pending? The original implementation used a stack and a reentrant activation handler to allow multiple scheduler activations to be active at a time; but an alternative approach (published at the same time, but under a different name) simply disables upcalls until the activation handler tells the kernel it’s OK. In the meantime, the kernel simply resumes the process instead.

5.5 Client-Server and RPC

Message-passing can be fast, and has quite nice semantics (either a message is sent or it isn’t, and it’s either received or it isn’t). Moreover, over a network (as we’ll see later in the course), it’s the only way to communicate. We will soon encounter cases which, for the moment at least, we do not see in a single-machine OS: lost messages, reordered messages, or messages that are delayed by some unbounded time.

Until then, consider just two parties communicating by messages (often the common case). Typically, this interaction is asymmetric: one end of the communication is offering a service, while the other is using it.

Definition 5.20 (Client-Server). In the client-server paradigm distributed computing, a server offers a service to potentially multiple clients, who connect to it to invoke the service.

Remarks:

• This is a distributed computing concept, but it applies even in the single-machine OS case (which is why we introduce it here). Indeed, the distinction between inter-process communication (with a single OS) and networked communication (between machines over a network) is increasingly blurred these days, and we’ll see more of this later. Rather than focussing on whether one or more machines is involved, it’s better to think about what network model is assumed between endpoints: can messages be lost? Reordered? Delayed indefinitely? etc.
Pipes can’t handle client-server communication, since either the client or server (or a common ancestor) must have forked the other. Client-server requires a way to name the end-point where the server is offering the service, so that clients can connect to it. You have seen one way to deal with this: sockets, where the server address is passed to the `connect()` call by the client.

If you write client-server code using sockets, however, you immediately encounter an issue: you find yourself writing the same “wrapper” code over and over again for every service and every client.

**Definition 5.21** (Remote Procedure Call). Remote Procedure Call or RPC is a programming technique whereby remote client-server interactions are made to look to the programmer of both the client and the server as simple procedure calls: the client program calls the server using a simple procedure call, and the server program implements the service purely as a procedure with the appropriate name.

How this works is as follows: the *signature* of the remote procedure is fed into a *stub compiler*, which outputs two chunks of code that go into libraries. The first is the client stub (or *proxy* which implements the client side procedure: this takes its arguments, *marshals* them into a buffer, sends the buffer to the server, and waits for a message in reply. When this comes back, it *unmarshals* the return value(s) from the call, and returns to the client program.

The second is the server stub, which performs the corresponding actions on the server side: wait for a message, unmarshal the arguments, call the server code with the arguments, marshal the return value(s) into a buffer, and send it back to the client.

**Remarks:**

- As described, this allows only one procedure to be made available remotely. In practice, this generalizes. The stub compiler can generate code for a *interface*, which is a collection of related procedures. All RPCs to the same interface go over the same connection, with an extra, hidden argument marshalled in: the *procedure number* of the function to call on the other side.

- For languages without strong type systems like C, a separate language is needed to define interfaces.

**Definition 5.22** (Interface Definition Language). An *Interface Definition Language* or *IDL* is a small, domain-specific language for writing RPC interface definitions.

**Remarks:**

- If the RPC is to be carried over a network, both sides need to agree on a common representation for the arguments and results of the RPC. This requires, in networking terms, a *presentation-layer protocol*. An example is XDR, the eXternal Data Representation used for, among other applications, the UNIX Network File System (NFS).
• If the RPC is, instead, local to a machine (in which case it is called, without apparent irony, Local RPC), the use of a presentation-layer protocol is less important. However, the performance tradeoff is now different. For classical networked RPC, the time to execute the simplest possible RPC (the “Null RPC”, which just returns) is dominated by the network propagation delay. On a single machine, through the kernel, it can be dominated by the cost of entering and exiting the kernel. For a server which is executing lots of requests, or a client which needs to send many requests in sequence, this can be a bottleneck.

Example 5.23 (RPC over synchronous IPC). A client executing an RPC needs to perform two operations, which are typically system calls: first, the send call, and second, the receive call to get the reply. Similarly, a server processing requests need to receive the message, then execute another system call to send the result back.

For this reason, high-performance local RPC systems allow the OS the combine two operations in a single system call. There are two important cases.

The first is sometimes called “send with closed receive”: the operation sends a message to a given destination, and then the thread blocks waiting for a reply from that destination. This performs the whole of the client side of an RPC communication in one syscall.

The second is sometimes called “send with open receive”: this sends a message to a destination and then blocks waiting for a message from any source. This is both halves of the server side combined, but in reverse order: the server calls this to send a reply and then block waiting for the next call.

5.6 Distributed objects

How does this get set up? As with TCP network connections, a server needs to create an end-point (in the TCP case, a listening socket) and advertise its address somewhere, while a client has to look up this address and connect to it. This usually requires a 3rd party.

Definition 5.24 (Name server). A name server is a service (usually invoked using RPC) which holds the addresses of other RPC services. Servers register their services with the name server, and clients lookup the service they want to get the address to connect to.

The data that the name server stores and hands out for a service is sometimes called an interface reference. It’s a name for the service, and it can be passed around freely (which is why the nameserver can be an RPC server like any other).

Definition 5.25 (RPC binding). To contact an RPC service, a client has to acquire an interface reference for the service, and then establish a binding to the service. Establishing the binding is basically setting up a connection, and results in an invocation reference, which is the required client stub.

Binding can be explicit, in which case the client has to call some kind of “bind” or “connect” procedure to establish it. However, implicit binding is also possible: as part of unmarshalling an interface reference, the binding is established immediately and an invocation reference returned to the client (or server) instead.
Remarks:

- This is similar to the notion of a binding we saw early on, but here the binding is an association between the client stub and the remote service: the local pointer or reference the client program has to its stub is now, implicitly, bound to the remote server.

- We didn’t talk about this in Chapter 2 but the client binding (and the analogous binding on the server side) are often first-class objects themselves: you can perform operations on them to manipulate the connection or query its status, for example.

By now, you’re probably thinking that this is beginning to feel somewhat object-oriented.

**Definition 5.26.** Distributed object system. A *distributed object system* is an RPC system with implicit binding where interface references are viewed as object references, and the IDL (if present) defines classes of which services are instances.

A local datastructure called the *object table* holds a mapping from interface references to invocation references, so that when an interface reference arrives twice, only a single client stub is created.

Remarks:

- Note that a distributed object system need not be tied to a particular language: well-known examples like CORBA and DCOM are not. However, they do need to define their own type system if they cannot lift one from a single programming language. Most examples use C primitive types, plus records and (discriminated) unions, plus variable length arrays (sequences) and, of course, interface/object references.

- The type system for interfaces can be arbitrarily sophisticated, supporting subtyping (interface inheritance), and (in some cases) distributed garbage collection.

RPC, and distributed objects, are intuitive for programmers, but also are predicated on things mostly working: either your RPC was delivered, was executed, and you got a reply, or you get a program exception saying it didn’t.

However, when implementing some distributed algorithms which are designed to handle lost messages, delays, node failures, etc., this model isn’t realistic (often because these algorithms are intended to create this reliability). Hence, when discussing things like consensus, we tend to talk in terms of messages, whereas talking to web services is couched in terms of RPCs.

**Bibliography**
