<table>
<thead>
<tr>
<th>Item</th>
<th>Multiprocessor</th>
<th>Multicomputer</th>
<th>Distributed System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node configuration</td>
<td>CPU</td>
<td>CPU, RAM, net interface</td>
<td>Complete computer</td>
</tr>
<tr>
<td>Node peripherals</td>
<td>All shared</td>
<td>Shared exc. maybe disk</td>
<td>Full set per node</td>
</tr>
<tr>
<td>Location</td>
<td>Same rack</td>
<td>Same room</td>
<td>Possibly worldwide</td>
</tr>
<tr>
<td>Internode communication</td>
<td>Shared RAM</td>
<td>Dedicated interconnect</td>
<td>Traditional network</td>
</tr>
<tr>
<td>Operating systems</td>
<td>One, shared</td>
<td>Multiple, same</td>
<td>Possibly all different</td>
</tr>
<tr>
<td>File systems</td>
<td>One, shared</td>
<td>One, shared</td>
<td>Each node has own</td>
</tr>
<tr>
<td>Administration</td>
<td>One organization</td>
<td>One organization</td>
<td>Many organizations</td>
</tr>
</tbody>
</table>

Figure 1: Comparison of three kinds of multiple CPU systems.

CIS2326 Multiple Processor Systems Week 10

Three kinds of multiple process systems are looked at - for a comparison, see Figure 1.

Definitions

**Multiprocessor**: a computer system in which two or more CPUs share full access to a common RAM.

In this model every CPU has equal access to the entire physical memory, and can read and write individual words using LOAD and STORE instructions. Accessing a memory word takes approximately 10-50 nsec.

**Multicomputers**: tightly coupled CPUs that do not share memory.

The CPUs are connected by some kind of high-speed interconnect. The machines communicate by sending multiword messages over the interconnect. When communications are good, a short message can take 10-50 μsec.

**Distributed System**: complete computer systems are connected over a wide area network. The systems are loosely coupled, as opposed to the tightly coupled systems of multicomputers. Message times are of the order of 10-50 msec.

**Multiprocessors**

**Hardware**

An important property is the ability for every memory word to be read as fast as every other memory word. Multiprocessors which have this property are called **Uniform Memory Access (UMA)** multiprocessors. Not all multiprocessors have this property and ones which do not are called **Non Uniform Memory Access (NUMA)**.

The simplest UMA is bus-based and there is an associated problem in a CPU having to wait for input/output to memory until the bus becomes idle. Alternative designs which solve this problem are CPUs which have their own cache and CPUs which have a cache and an additional private memory - which it accesses via a private bus.

More complex UMA multiprocessors use **switching networks**; these are similar to those used for many years within telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way. Another multiprocessor design is based on the ‘2 by 2’ switch. There are two outputs and two inputs - messages from either input can be switched to either output. From this can be built larger **multistage switching networks**.
The UMA model is limited to about 100 CPUs - because of expense. A more realistic system allows different memory modules different access times. Each CPU has its own 'local memory' and also has 'remote' access to the local memories of other CPUs - for all share a single address space. However for each CPU, access to remote memory is slower than access to local memory.

**OS Types** are as follows:

(i) Each CPU has its own operating system - see Figure 2. The memory is divided into as many partitions as there are CPUs - each CPU has its own private memory and its own private copy of the OS. An optimisation is to allow all the CPUs to share the OS code and make private copies of the data only. There are various problems with this very primitive approach, including the fact that one CPU might be idle while another is loaded with work and that if a certain disc block is 'dirty' in multiple memory caches this could lead to inconsistencies.

(ii) Master-Slave Multiprocessors (Figure 3.)

One copy of the OS and its tables are present on CPU 1 and not on any of the others. This solves most of the problems of the first model. The CPUs are coordinated by the 'master' so that work is shared. For a similar reason there is only one buffer cache which eliminates inconsistencies. The problem here is that the master can become a bottleneck.

(iii) Symmetric Multiprocessors (SMP) (Figure 4)

Any multiprocessor can use the one copy of the OS. This eliminates the master CPU bottleneck problem - but introduces its own problems. In particular - two CPUs could pick the
same process! A simple solution is to associate a lock with the OS - so only one CPU can use the OS at a time. Greater efficiency is obtained by partitioning the OS into critical sections which can run independently of each other. This still leads to problems, and great care is needed in order to avoid deadlocks - all of which leads to an examination of synchronisation of multiprocessors.

Synchronisation
This is far from trivial - a problem with (for example) the TSL (test and set lock) operation is that it takes two separate bus cycles - first it reads the memory word lock and stores it in a register then it stores a nonzero value in lock.

In Figure 5 we see that CPU 1 reads a ‘0’, however before CPU 1 has a chance to write ‘1’, CPU 2 reads ‘0’ ... this means both CPUs can access critical region and mutual exclusion fails. One way round it is to give each CPU its own private lock as in figure 6. A CPU that fails to acquire the lock allocates a lock variable and attaches itself to the end of a CPU list waiting for the lock. As each lock holder exits the critical section, it frees the private lock being tested by the CPU at the head of the list. A problem with any synchronisation of CPUs is that it can involve a CPU ‘spinning’ (waiting) which wastes its time. A solution is to ‘switch’ the CPU to another thread - however this also wastes time for the state of the current thread must be saved. Researchers have discovered that the best decision is to keep track of previous ‘spin’ times when deciding whether to ‘spin’ or ‘switch’.

Figure 5: The TSL instruction can fail if the bus is already locked. The four steps show a sequence of events where the failure is demonstrated.

Figure 6: Use of multiple locks to avoid cache thrashing.
Figure 7: Various Interconnection Topologies: (a) single switch (b) ring (c) grid (d) double taurus (e) cube (f) hypercube

Figure 8: Steps in making a remote procedure call. The stubs are grey.

Multicomputers

The basic node of a multicomputer consists of a CPU, memory, network interface and (possibly) a hard disc. The nodes are interconnected using various topologies (Figure 7). Communication is via messages and messages can either be split into packets or a route can be determined between nodes and the message pumped through non-stop using circuit switching. However the procedures of send/receive of messages are built around input/output.

Remote Procedure Call (RPC)

Often, a better way of sharing processing between CPUs is RPC. When a process on CPU 1 calls a process on CPU 2, the calling process on CPU 1 and execution of the process takes place on CPU 2. The idea is to make a remote procedure call look as much as possible like a local one. The client program is bound with a small library procedure called the client stub that represents the server procedure in the client’s address space. Similarly for the server. The steps are shown on Figure 8, where CPU 1 is the client CPU and CPU 2 is the server.

Step 1: client calls client stub, a library procedure representing the called procedure. This is a local procedure call with the parameters pushed onto the stack.

Step 2: client stub packs the parameters into a message and makes the system call to send the message (packing the parameters is called marshalling);

Step 3: kernel 1 sends message from client to server;
Step 4: kernel 2 passes message to server stub;
Step 5: Server stub calls procedure.
Reply is in reverse direction.

**Load Balancing**

The decision about which process should go on which node is important - this is in contrast to multiprocessor systems in which all processes live in the same memory and can be scheduled on any CPU. This is known as load balancing and the algorithms which aid the decision making are shown in Figures 9 and 10. Figure 9 demonstrates a graph-theoretic deterministic algorithm. Each vertex is a process and each arc represents the flow of messages between two processes. The idea is to minimize network traffic. A partition is arranged of the graph into disjoint subgraphs - one for each node. In Figure 9 we see nine processes allocated in 2 different ways. In each case, the arc between processes is labelled by the mean communication time between the processes. The total network traffic is the sum of those arcs which are intersected by the partition boundaries (dashed lines). In Figure 9(a) it is 30 and in Figure 9 (b) it is 28 - therefore (b) is preferable - given that it meets other constraints.

Figure 10 demonstrates two distributed algorithms. The first is in Figure 10(a) a Sender Initiated Distributed Heuristic Algorithm. In this algorithm, a process is normally run on the node which created it. If that node is overloaded it selects another node at random and asks
what its load is (using some metric). If the selected node’s load is below some threshold value, the new process is sent there. The problem is the overhead generated if the whole system is overloaded - when nodes send ultimately futile requests to other machines.

The second algorithm is complementary and demonstrated in Figure 10(b). When a process completes, its node asks for further work. If no work is found, the node completes any queued up work and tries again after a fixed time interval. The advantage of this system is that no extra load is put on the system at critical times. It is also possible to combine the two distributed load balancing algorithms. Thus machines try to offload work if they have too much and take on work if they have too little.

Distributed Computing

Distributed systems are essentially different from the previous two kinds of multiple processor systems, which are mostly concerned with solving computationally intensive problems. In contrast distributed systems are mainly concerned with *communication*. Since the computers comprising the nodes of the system can be different with different hardware and OSs there needs to be a unifying paradigm. Such a paradigm is called *middleware*. The layer provides data structures and operations that allows processes and users on remote machines to work together in a consistent way. There are two kinds of service:

**Connection oriented services**: work by the user establishing a connection, using it then releasing it. This can be compared to the telephone system.

**Connectionless services**: work by sending the data as a message, a *datagram*, with each one independant of the rest. This can be compared to the postal system.

**Network Protocols**:

There are highly specialised rules for transferring messages and their responses over any network. The set of rules is called a *protocol*. Two important protocols are the *Internet Protocol (IP)* and *Transmission Control Protocol (TCP)*. IP is a datagram protocol by which a sender transmits a datagram of up to 64 KB into the network and hopes it will arrive - but no guarantee is given. (It is an unreliable connectionless service.) In contrast TCP is built on top of IP - first a connection is established to a remote process and the bytes are sent to this process. The TCP implementation guarantees that the bytes will be in the correct order and undamaged by using sequence numbers, checksums etc. (A less reliable protocol is *User Datagram Protocol or UDP*.)

**Sockets** are data structures which allow users to interface to the network. They can be thought of as analogous to mailboxes and telephone wall sockets. They support both byte streams and packet streams and are created and destroyed dynamically. They are created on both the source and destination computers. The sequence of operations for both server and client is as follows:

**Create a Socket** There are two important types of sockets. The first is connection oriented and is implemented in the Internet domain by TCP. The second is connectionless (datagram) and implemented by UDP - as described above.

**Bind an Address** A connection between two sockets is fully specified by source IP address, source port (specific socket), protocol in use and destination IP address, destination port.

**Connecting Sockets** A stream socket can wait passively to be contacted or it can actively connect to another socket.
Accepting Connections: The process which created the socket blocks, listening for requests to the port associated with the socket.

Setting up a connection: A client can ask the system to connect it to a remote socket. For this it needs to know its address for subsequent data transfer.

Close socket The socket is removed from the system.

An alternative method of data transfer is a Remote Procedure Call (described earlier).

Distributed Mutual Exclusion: We need to be able to implement mutual exclusion in a distributed system - for example to control access to a file by two or more processes. Mechanisms such as semaphores for exclusion are difficult to distribute. They rely on shared variables which exist in one place. It is difficult to devise atomic (indivisible) access over a network. Examples of algorithms are as follows:

(i) Centralised: - one of the processes is chosen as coordinator and each process has to ask its permission to enter its critical section (e.g. write to file). When the process receives a reply
message from the coordinator it can proceed. Until that happens the request is queued. After exiting the critical section the process sends a release message to the coordinator and proceeds with its non-critical execution. The coordinator ensures that only one process at a time has access. When the coordinator receives a release message it removes a request from the queue and sends a reply message to the requesting process. For this to work there needs to be a way of ensuring that coordinator failure is recognised. If the coordinator process fails another is elected. One algorithm for election is the bully algorithm in which if a process sends a request which is not answered within a certain time limit it is assumed that the coordinator has failed - the requesting process tries to elect itself as coordinator. It does this by sending an election message to every process with a higher priority. Another algorithm is the ring algorithm - the requesting process sends a message to its right-hand neighbour. (Details of both are in Silberschatz.)

(ii) Fully distributed algorithm - each process takes its share of responsibility for arranging mutual exclusion. When a process want to enter its critical section it multicasts to all processes including itself. A process may reply immediately or (if it is in its critical section for example) defer sending a reply. Only when all have responded and given permission is entry to a critical section allowed. After exiting its critical section the process sends reply messages back to all its deferred requests. Advantages are:

(i) Mutual exclusion is achieved (ii) freedom from deadlock and from starvation is achieved.
Disavantages are (i) that the processes need to know the identity of all other processes in the system; (ii) If a process fails then it will never respond to messages and the system collapses. There needs to be monitoring of processes for failure; (iii) There are frequent pauses while processes enter their critical sections. It is suitable for small, stable sets of cooperating processes.

NB See tutorial example on number of messages involved in each case (i) and (ii).

Shared Object-Based Middleware
Paradigms for middleware include document-based, file-based, object-based and coordination-based middleware. The first two relate to the world wide web, which can be thought of either as a huge directed graph of documents or as huge file system.

In contrast, the third paradigm regards everything as an object. One such system is CORBA (Common Object Based Request Broker Architecture) which is based on runtime objects. CORBA is a client-server system in which client processes on client machines can invoke operations on objects located on remote server machines - which can be remote.
Processes are required to access the variables of an object through the objects methods. The arrangement of CORBA parts is shown in Figure 11 (Note the similarity to RPCs.) For communication over the Internet the protocol is called Internet InterOrb Protocol (IIOP). The object adapter is a wrapper that handles chores such as registering the object, generating object references etc. This makes it possible to handle non-CORBA objects. However a problem with CORBA is that every object is located on only one server which means that performance will be poor on heavily invoked objects.

Globe is an example of a distributed object system designed for heavy use. There are two key ideas to scaling to larger systems. The first is replication and the second is flexibility. Replication means that with many instances of an object, the load is shared. Flexibility allows for different users and objects to behave differently. Direct LOADs and STOREs to an objects internal state are prohibited and all accesses are forced to go through one of the methods associated with the object.

Co-ordination Based Middleware
The first example of this paradigm is Linda, an academic research project which originated the idea. In Linda, independent processes communicate via tuple spaces. Processes can insert tuples into tuple space or remove tuples from the space. A tuple consists of one or more fields, each of which is a value of some ‘type’. Types can be (for example), integers, floating point numbers, arrays (including strings) and structures. (See Figure 12.) Tuples do not have associated methods. Four primitive operations are provided and are ‘out’, ‘in’, ‘read’ and ‘eval’. The fields of ‘out’ are constants, variables or expressions - it puts a tuple into tuple space. For example

```
out(‘abc’, 2, 5)
```

The fields of ‘in’ can be expressions or formal parameters. A ‘matching’ algorithm is used to compare the fields of ‘in’ with the fields of every tuple in the tuple space. The operation ‘in’
retrieves tuples from tuple space. For example

\[ \text{in(‘‘abc’’, 2, ?i)} \]

where the third tuple component is a formal (integer) parameter. In the example the operation “searches” the tuple space for a tuple consisting of string “abc” integer 2 and a third field containing any integer. If found, the tuple is removed and the variable i is assigned the value of the third field. The matching and removal are atomic so if two processes execute the same ‘in’ operation simultaneously, only one will succeed - unless there is more than one matching tuple. The ‘read’ operation copies from tuple space and ‘eval’ performs an arbitrary computation.

**Publish/Subscribe** was inspired by Linda and consists of a number of processes connected by a broadcast network. Each process can be a producer of information, a consumer or both. When an information consumer has a new piece of information it broadcasts (ie publishes) this as a tuple on the network. Processes that are interested in certain information can subscribe to certain subjects, telling a tuple daemon process on the same machine that monitors published tuples what to look for. Figure 13 shows how processes publish tuples by broadcasting them on the local LAN. The tuple daemons on each machine copies all of these tuples onto its RAM. It then forwards copies of appropriate tuples to interested processes. The tuples can also be broadcast over a WAN using a router - the tuples are then forwarded to other LANs for re-broadcasting.

**Jini** is an attempt to change the ‘CPU-centric’ view of computing to a ‘network-centric’ view. It is a product of SUN Microsystems and consists of a large number of self-contained Jini devices, each of which offers services to others. A Jini device can be plugged into a network and begin offering services instantly - and note that devices are plugged into a network rather than a computer. A device can be a computer, a palmtop computer, cell phone, printer, or any device with a CPU, memory and a network connection. A Jini system is a federation of Jini devices and when a device wishes to join this federation it broadcasts a packet on the local LAN or in the local wireless cell. When a new device wants to register it is sent a piece of code that can perform the registration. The code sent is JVM (Java Virtual Machine) which all Jini devices must be capable of running.

Jini clients communicate and synchronise using **JavaSpaces**, which are modelled on Linda tuple spaces. The difference between JavaSpaces and Linda tuple spaces is that JavaSpaces are strongly typed whereas Linda tuple spaces are untyped. In JavaSpaces, each of the fields
in a tuple has a basic Java type. The four methods defined on a JavaSpace are ‘write’, which puts a new entry into the JavaSpace, ‘read’, which copies an entry, ‘take’ which copies and removes an entry, and ‘notify’ which notifies the caller when a matching entry is written.

JavaSpace can be used for synchronisation between communicating processes. For example, in a producer-consumer situation, the producer puts items in a JavaSpace as it produces them. The producer removes them with ‘take’, blocking if none are available. JavaSpace guarantees that each of the methods is executed atomically, so there is no danger of one process trying to read an entry that has only been half entered.

Remaining Weeks
Week 11 (after Easter) - Unix/Linux and Windows XP
Week 12 Revision