CHAPTER 8

Process Management

8.1 INTRODUCTION

In a conventional (centralized) operating system, process management deals with mechanisms and policies for sharing the processor of the system among all processes. Similarly, in a distributed operating system, the main goal of process management is to make the best possible use of the processing resources of the entire system by sharing them among all processes. Three important concepts are used in distributed operating systems to achieve this goal:

1. *Processor allocation* deals with the process of deciding which process should be assigned to which processor.
2. *Process migration* deals with the movement of a process from its current location to the processor to which it has been assigned.
3. *Threads* deals with fine-grained parallelism for better utilization of the processing capability of the system.

The processor allocation concept has already been described in the previous chapter on resource management. Therefore, this chapter presents a description of the process migration and threads concepts.

8.2 PROCESS MIGRATION

Process migration is the relocation of a process from its current location (the *source node*) to another node (the *destination node*). The flow of execution of a migrating process is illustrated in Figure 8.1.

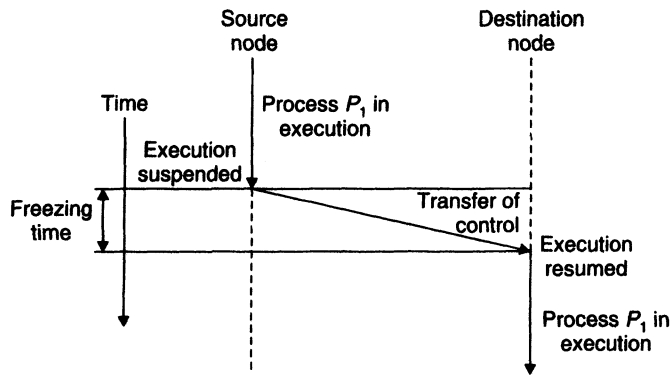


Fig. 8.1 Flow of execution of a migrating process.

A process may be migrated either before it starts executing on its source node or during the course of its execution. The former is known as *non-preemptive* process migration and the latter is known as *preemptive* process migration. Preemptive process migration is costlier than non-preemptive process migration since the process environment must also accompany the process to its new node for an already executing process.

Process migration involves the following major steps:

1. Selection of a process that should be migrated
2. Selection of the destination node to which the selected process should be migrated
3. Actual transfer of the selected process to the destination node

The first two steps are taken care of by the process migration policy and the third step is taken care of by the process migration mechanism. The policies for the selection of a source node, a destination node, and the process to be migrated have already been described in the previous chapter on resource management. This chapter presents a description of the process migration mechanisms used by the existing distributed operating systems.

8.2.1 Desirable Features of a Good Process Migration Mechanism

A good process migration mechanism must possess transparency, minimal interferences, minimal residue dependencies, efficiency, robustness, and communication between coprocesses.

Transparency

Transparency is an important requirement for a system that supports process migration. The following levels of transparency can be identified:

1. *Object access level.* Transparency at the object access level is the minimum requirement for a system to support non-preemptive process migration facility. If a system supports transparency at the object access level, access to objects such as files and devices can be done in a location-independent manner. Thus, the object access level transparency allows free initiation of programs at an arbitrary node. Of course, to support transparency at object access level, the system must provide a mechanism for transparent object naming and locating.

2. *System call and interprocess communication level.* So that a migrated process does not continue to depend upon its originating node after being migrated, it is necessary that all system calls, including interprocess communication, are location independent. Thus, transparency at this level must be provided in a system that is to support preemptive process migration facility. However, system calls to request the physical properties of a node need not be location independent.

Transparency of interprocess communication is also desired for the transparent redirection of messages during the transient state of a process that recently migrated. That is, once a message has been sent, it should reach its receiver process without the need for resending it from the sender node in case the receiver process moves to another node before the message is received.

Minimal Interference

Migration of a process should cause minimal interference to the progress of the process involved and to the system as a whole. One method to achieve this is by minimizing the freezing time of the process being migrated. *Freezing time* is defined as the time period for which the execution of the process is stopped for transferring its information to the destination node.

Minimal Residual Dependencies

No residual dependency should be left on the previous node. That is, a migrated process should not in any way continue to depend on its previous node once it has started executing on its new node since, otherwise, the following will occur:

- The migrated process continues to impose a load on its previous node, thus diminishing some of the benefits of migrating the process.
- A failure or reboot of the previous node will cause the process to fail.

Efficiency

Efficiency is another major issue in implementing process migration. The main sources of inefficiency involved with process migration are as follows:

- The time required for migrating a process
- The cost of locating an object (includes the migrated process)
- The cost of supporting remote execution once the process is migrated

All these costs should be kept to minimum as far as practicable.

Robustness

The process migration mechanism must also be robust in the sense that the failure of a node other than the one on which a process is currently running should not in any way affect the accessibility or execution of that process.

Communication between Coprocesses of a Job

One further exploitation of process migration is the parallel processing among the processes of a single job distributed over several nodes. Moreover, if this facility is supported, to reduce communication cost, it is also necessary that these coprocesses be able to directly communicate with each other irrespective of their locations.

8.2.2 Process Migration Mechanisms

Migration of a process is a complex activity that involves proper handling of several sub-activities in order to meet the requirements of a good process migration mechanism listed above. The four major subactivities involved in process migration are as follows:

1. Freezing the process on its source node and restarting it on its destination node
2. Transferring the process's address space from its source node to its destination node
3. Forwarding messages meant for the migrant process
4. Handling communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration

The commonly used mechanisms for handling each of these subactivities are described below.

Mechanisms for Freezing and Restarting a Process

In preemptive process migration, the usual process is to take a "snapshot" of the process's state on its source node and reinstate the snapshot on the destination node. For this, at some point during migration, the process is frozen on its source node, its state information is transferred to its destination node, and the process is restarted on its destination node using this state information. By freezing the process, we mean that the execution of the process is suspended and all external interactions with the process are deferred. Although the freezing and restart operations differ from system to system, some general issues involved in these operations are described below.

Immediate and Delayed Blocking of the Process. Before a process can be frozen, its execution must be blocked. Depending upon the process's current state, it may be blocked immediately or the blocking may have to be delayed until the process reaches a state when it can be blocked. Some typical cases are as follows [Kingsbury and Kline 1989]:

1. If the process is not executing a system call, it can be immediately blocked from further execution.
2. If the process is executing a system call but is sleeping at an interruptible priority (a priority at which any received signal would awaken the process) waiting for a kernel event to occur, it can be immediately blocked from further execution.
3. If the process is executing a system call and is sleeping at a noninterruptible priority waiting for a kernel event to occur, it cannot be blocked immediately. The process's blocking has to be delayed until the system call is complete. Therefore, in this situation, a flag is set, telling the process that when the system call is complete, it should block itself from further execution.

Note that there may be some exceptions to this general procedure of blocking a process. For example, sometimes processes executing in certain kernel threads are immediately blocked, even when sleeping at noninterruptible priorities. The actual mechanism varies from one implementation to another.

Fast and Slow I/O Operations. In general, after the process has been blocked, the next step in freezing the process is to wait for the completion of all fast I/O operations (e.g., disk I/O) associated with the process. The process is frozen after the completion of all fast I/O operations. Note that it is feasible to wait for fast I/O operations to complete before freezing the process. However, it is not feasible to wait for slow I/O operations, such as those on a pipe or terminal, because the process must be frozen in a timely manner for the effectiveness of process migration. Thus proper mechanisms are necessary for continuing these slow I/O operations correctly after the process starts executing on its destination node.

Information about Open Files. A process's state information also consists of the information pertaining to files currently open by the process. This includes such information as the names or identifiers of the files, their access modes, and the current

positions of their file pointers. In a distributed system that provides a network transparent execution environment, there is no problem in collecting this state information because the same protocol is used to access local as well as remote files using the systemwide unique file identifiers. However, several UNIX-based network systems uniquely identify files by their full pathnames [Mandelberg and Sunderam 1988, Alonso and Kyrimis 1988]. But in these systems it is difficult for a process in execution to obtain a file's complete pathname owing to UNIX file system semantics. A pathname loses significance once a file has been opened by a process because the operating system returns to the process a file descriptor that the process uses to perform all I/O operations on the file. Therefore, in such systems, it is necessary to somehow preserve a pointer to the file so that the migrated process could continue to access it. The following two approaches are used for this:

1. In the first approach [Mandelberg and Sunderam 1988], a link is created to the file and the pathname of the link is used as an access point to the file after the process migrates. Thus when the snapshot of the process's state is being created, a link (with a special name) is created to each file that is in use by the process.

2. In the second approach [Alonso and Kyrimis 1988], an open file's complete pathname is reconstructed when required. For this, necessary modifications have to be incorporated in the UNIX kernel. For example, in the approach described in [Alonso and Kyrimis 1988], each file structure, where information about open files is contained, is augmented with a pointer to a dynamically allocated character string containing the absolute pathname of the file to which it refers.

Another file system issue is that one or more files being used by the process on its source node may also be present on its destination node. For example, it is likely that the code for system commands such as *nroff, cc* is replicated at every node. It would be more efficient to access these files from the local node at which the process is executing, rather than accessing them across the network from the process's previous node [Agrawal and Ezzat 1987]. Another example is temporary files that would be more efficiently created at the node on which the process is executing, as by default these files are automatically deleted at the end of the operation. Therefore, for performance reasons, the file state information collected at the source node should be modified properly on the process's destination node in order to ensure that, whenever possible, local file operations are performed instead of remote file operations. This also helps in reducing the amount of data to be transferred at the time of address space transfer because the files already present on the destination node need not be transferred.

Reinstating the Process on its Destination Node. On the destination node, an empty process state is created that is similar to that allocated during process creation. Depending upon the implementation, the newly allocated process may or may not have the same process identifier as the migrating process. In some implementations, this newly created copy of the process initially has a process identifier different from the migrating process in order to allow both the old copy and the new copy to exist and be accessible at the same time. However, if the process identifier of the new copy of the process is different from its old copy, the new copy's identifier is changed to the original identifier

in a subsequent step before the process starts executing on the destination node. The rest of the system cannot detect the existence of two copies of the process because operations on both of them are suspended. Once all the state of the migrating process has been transferred from the source node to the destination node and copied into the empty process state, the new copy of the process is unfrozen and the old copy is deleted. Thus the process is restarted on its destination node in whatever state it was in before being migrated.

It may be noted here that the method described above to reinstate the migrant process is followed in the most simple and straightforward case. Several special cases may require special handling and hence more work. For example, when obtaining the snapshot, the process may have been executing a system call since some calls are not atomic. In particular, as described before, this can happen when the process is frozen while performing an I/O operation on a slow device. If the snapshot had been taken under such conditions, correct process continuation would be possible only if the system call is performed again. Therefore normally a check is made for these conditions and, if required, the program counter is adjusted as needed to reissue the system call.

Address Space Transfer Mechanisms

A process consists of the program being executed, along with the program's data, stack, and state. Thus, the migration of a process involves the transfer of the following types of information from the source node to the destination node:

- Process's state, which consists of the execution status (contents of registers), scheduling information, information about main memory being used by the process (memory tables), I/O states (I/O queue, contents of I/O buffers, interrupt signals, etc.), a list of objects to which the process has a right to access (capability list), process's identifier, process's user and group identifiers, information about the files opened by the process (such as the mode and current position of the file pointer), and so on
- Process's address space (code, data, and stack of the program)

For nontrivial processes, the size of the process's address space (several megabytes) overshadows the size of the process's state information (few kilobytes). Therefore the cost of migrating a process is dominated by the time taken to transfer its address space. Although it is necessary to completely stop the execution of the migrant process while transferring its state information, it is possible to transfer the process's address space without stopping its execution. In addition, the migrant process's state information must be transferred to the destination node before it can start its execution on that node. Contrary to this, the process's address space can be transferred to the destination node either before or after the process starts executing on the destination node.

Thus in all the systems, the migrant process's execution is stopped while transferring its state information. However, due to the flexibility in transferring the process's address space at any time after the migration decision is made, the existing distributed systems use one of the following address space transfer mechanisms: total freezing, pretransferring, or transfer on reference.

Total Freezing. In this method, a process's execution is stopped while its address space is being transferred (Fig. 8.2). This method is used in DEMOS/MP [Powell and Miller 1983], Sprite [Douglis and Ousterhout 1987], and LOCUS [Popek and Walker 1985] and is simple and easy to implement. Its main disadvantage is that if a process is suspended for a long time during migration, timeouts may occur, and if the process is interactive, the delay will be noticed by the user.

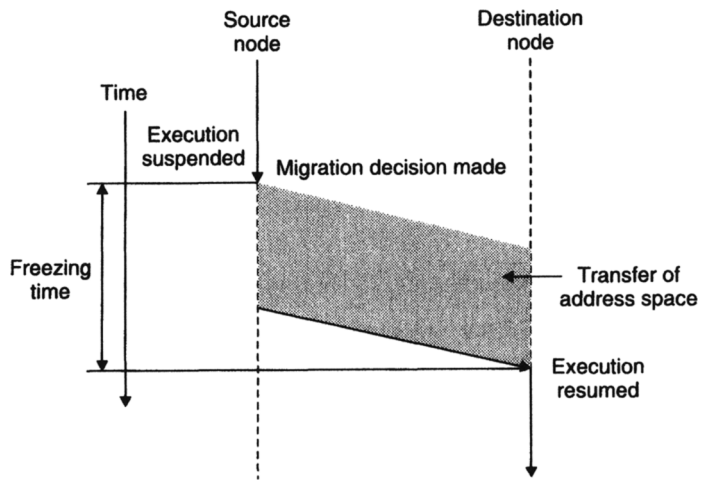


Fig. 8.2 Total freezing mechanism.

Pretransferring. In this method, the address space is transferred while the process is still running on the source node (Fig. 8.3). Therefore, once the decision has been made to migrate a process, it continues to run on its source node until its address space has been transferred to the destination node. Pretransferring (also known as *precopying*) is done as an initial transfer of the complete address space followed by repeated transfers of the pages modified during the previous transfer until the number of modified pages is relatively small or until no significant reduction in the number of modified pages (detected using dirty bits) is achieved. The remaining modified pages are retransferred after the process is frozen for transferring its state information [Theimer et al. 1985].

In the pretransfer operation, the first transfer operation moves the entire address space and takes the longest time, thus providing the longest time for modifications to the program's address space to occur. The second transfer moves only those pages of the address space that were modified during the first transfer, thus taking less time and presumably allowing fewer modifications to occur during its execution time. Thus subsequent transfer operations have to move fewer and fewer pages, finally converging to zero or very few pages, which are then transferred after the process is frozen. It may be noted here that the pretransfer operation is executed at a higher priority than all other

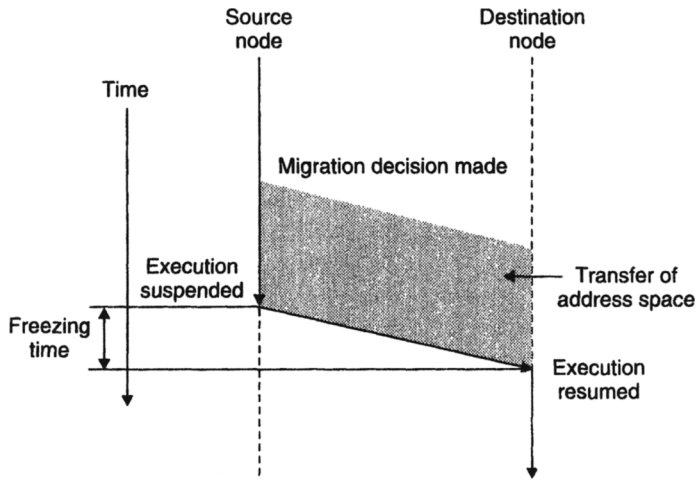


Fig. 8.3 Pretransfer mechanism.

programs on the source node to prevent these other programs from interfering with the progress of the pretransfer operation.

This method is used in the V-System [Theimer et al. 1985]. In this method, the freezing time is reduced so migration interferes minimally with the process's interaction with other processes and the user. Although pretransferring reduces the freezing time of the process, it may increase the total time for migration due to the possibility of redundant page transfers. *Redundant pages* are pages that are transferred more than once during pretransferring because they become dirty while the pretransfer operation is being performed.

Transfer on Reference. This method is based on the assumption that processes tend to use only a relatively small part of their address spaces while executing. In this method, the process's address space is left behind on its source node, and as the relocated process executes on its destination node, attempts to reference memory pages results in the generation of requests to copy in the desired blocks from their remote locations. Therefore in this demand-driven copy-on-reference approach, a page of the migrant process's address space is transferred from its source node to its destination node only when referenced (Fig. 8.4). However, Zayas [1987] also concluded through his simulation results that prefetching of one additional contiguous page per remote fault improves performance.

This method is used in Accent [Zayas 1987]. In this method, the switching time of the process from its source node to its destination node is very short once the decision about migrating the process has been made and is virtually independent of the size of the address space. However, this method is not efficient in terms of the cost of supporting remote execution once the process is migrated, and part of the effort saved in the lazy transfer of an address space must be expended as the process accesses its memory

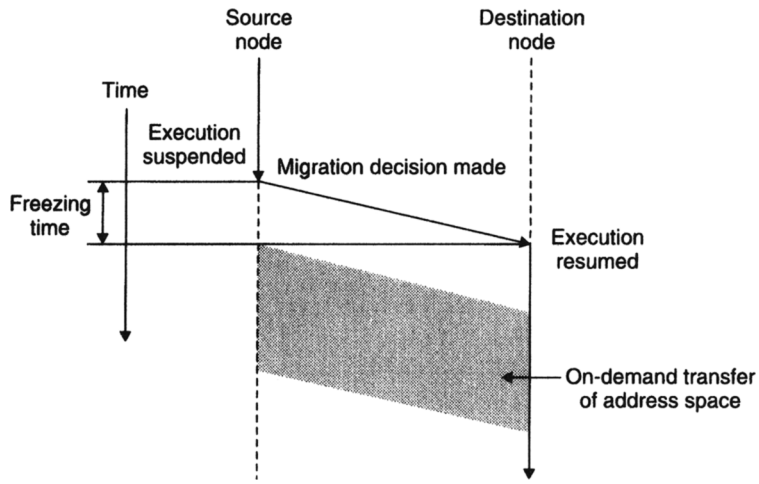


Fig. 8.4 Transfer-on-reference mechanism.

remotely. Furthermore, this method imposes a continued load on the process's source node and results in failure of the process if the source node fails or is rebooted.

Message-Forwarding Mechanisms

In moving a process, it must be ensured that all pending, en-route, and future messages arrive at the process's new location. The messages to be forwarded to the migrant process's new location can be classified into the following:

Type 1: Messages received at the source node after the process's execution has been stopped on its source node and the process's execution has not yet been started on its destination node

Type 2: Messages received at the source node after the process's execution has started on its destination node

Type 3: Messages that are to be sent to the migrant process from any other node after it has started executing on the destination node

The different mechanisms used for message forwarding in existing distributed systems are described below.

Mechanism of Resending the Message. This mechanism is used in the V-System [Cheriton 1988, Theimer et al. 1985] and Amoeba [Mullender et al. 1990] to handle messages of all three types. In this method, messages of types 1 and 2 are returned to the sender as not deliverable or are simply dropped, with the assurance that the sender of the message is storing a copy of the data and is prepared to retransmit it.

For example, in V-System, a message of type 1 or 2 is simply dropped and the sender is prompted to resend it to the process's new node. The interprocess communication mechanism of V-System ensures that senders will retry until successful receipt of a reply. Similarly in Amoeba, for all messages of type 1, the source node's kernel sends a "try again later, this process is frozen" message to the sender. After the process has been deleted from the source node, those messages will come again at some point of time, but this time as type 2 messages. For all type 2 messages, the source node's kernel sends a "this process is unknown at this node" message to the sender.

In this method, upon receipt of a negative reply, the sender does a *locate* operation to find the new whereabouts of the process, and communication is reestablished. Both V-System and Amoeba use the broadcasting mechanism to locate a process (object locating mechanisms are described in Chapter 10). Obviously, in this mechanism, messages of type 3 are sent directly to the process's destination node.

This method does not require any process state to be left behind on the process's source node. However, the main drawback of this mechanism is that the message-forwarding mechanism of process migration operation is nontransparent to the processes interacting with the migrant process.

Origin Site Mechanism. This method is used in AIX's TCF (Transparent Computing Facility) [Walker and Mathews 1989] and Sprite [Douglis and Ousterhout 1987]. The process identifier of these systems has the process's *origin site* (or *home node*) embedded in it, and each site is responsible for keeping information about the current locations of all the processes created on it. Therefore, a process's current location can be simply obtained by consulting its origin site. Thus, in these systems, messages for a particular process are always first sent to its origin site. The origin site then forwards the message to the process's current location. This method is not good from a reliability point of view because the failure of the origin site will disrupt the message-forwarding mechanism. Another drawback of this mechanism is that there is a continuous load on the migrant process's origin site even after the process has migrated from that node.

Link Traversal Mechanism. In DEMOS/MP [Powell and Miller 1983], to redirect the messages of type 1, a message queue for the migrant process is created on its source node. All the messages of this type are placed in this message queue. Upon being notified that the process is established on the destination node, all messages in the queue are sent to the destination node as a part of the migration procedure.

To redirect the messages of types 2 and 3, a forwarding address known as *link* is left at the source node pointing to the destination node of the migrant process. The most important part of a link is the message process address that has two components. The first component is a systemwide, unique, process identifier. It consists of the identifier of the node on which the process was created and a unique local identifier generated by that node. The second component is the last known location of the process. During the lifetime of a link, the first component of its address never changes; the second, however, may. Thus to forward messages of types 2 and 3, a migrated process is located by traversing a series of links (starting from the node where the process was created) that form a chain ultimately leading to the process's current location. The second component of a link is updated when the corresponding

process is accessed from a node. This is done to improve the efficiency of subsequent locating operations for the same process from that node.

The link traversal mechanism used by DEMOS/MP for message forwarding suffers from the drawbacks of poor efficiency and reliability. Several links may have to be traversed to locate a process from a node, and if any node in the chain of links fails, the process cannot be located.

Link Update Mechanism. In Charlotte [Artsy and Finkel 1989], processes communicate via location-independent links, which are capabilities for duplex communication channels. During the transfer phase of the migrant process, the source node sends link-update messages to the kernels controlling all of the migrant process's communication partners. These link update messages tell the new address of each link held by the migrant process and are acknowledged (by the notified kernels) for synchronization purposes. This task is not expensive since it is performed in parallel. After this point, messages sent to the migrant process on any of its links will be sent directly to the migrant process's new node. Communication requests postponed while the migrant process was being transferred are buffered at the source node and directed to the destination node as a part of the transfer process. Therefore, messages of types 1 and 2 are forwarded to the destination node by the source node and messages of type 3 are sent directly to the process's destination node.

Mechanisms for Handling Coprocesses

In systems that allow process migration, another important issue is the necessity to provide efficient communication between a process (parent) and its subprocesses (children), which might have been migrated and placed on different nodes. The two different mechanisms used by existing distributed operating systems to take care of this problem are described below.

Disallowing Separation of Coprocesses. The easiest method of handling communication between coprocesses is to disallow their separation. This can be achieved in the following ways:

1. By disallowing the migration of processes that wait for one or more of their children to complete
2. By ensuring that when a parent process migrates, its children processes will be migrated along with it

The first method is used by some UNIX-based network systems [Alonso and Kyrimis 1988, Mandelberg and Sunderam 1988] and the second method is used by V-System [Theimer et al. 1985]. To ensure that a parent process will always be migrated along with its children processes, V-System introduced the concept of *logical host*. V-System address spaces and their associated processes are grouped into logical hosts. A V-System process identifier is structured as a (*logical-host-id, local-index*) pair. In the extreme, each program can be run in its own logical host. There may be multiple logical hosts associated with a single node; however, a logical host is local to a single node. In V-System, all subprocesses

of a process typically execute within a single logical host. Migration of a process is actually migration of the logical host containing that process. Thus, typically, all subprocesses of a process are migrated together when the process is migrated [Theimer et al. 1985].

The main disadvantage of this method is that it does not allow the use of parallelism within jobs, which is achieved by assigning the various tasks of a job to the different nodes of the system and executing them simultaneously on these nodes. Furthermore, in the method employed by V-System, the overhead involved in migrating a process is large when its logical host consists of several associated processes.

Home Node or Origin Site Concept. Sprite [Douglis and Ousterhout 1987] uses its home node concept (previously described) for communication between a process and its subprocess when the two are running on different nodes. Unlike V-System, this allows the complete freedom of migrating a process or its subprocesses independently and executing them on different nodes of the system. However, since all communications between a parent process and its children processes take place via the home node, the message traffic and the communication cost increase considerably. Similar drawbacks are associated with the concept of origin site of LOCUS [Popek and Walker 1985].

8.2.3 Process Migration in Heterogeneous Systems

When a process is migrated in a homogeneous environment, the interpretation of data is consistent on both the source and the destination nodes. Therefore, the question of *data translation* does not arise. However, when a process is migrated in a heterogeneous environment, all the concerned data must be translated from the source CPU format to the destination CPU format before it can be executed on the destination node. If the system consists of two CPU types, each processor must be able to convert the data from the foreign processor type into its own format. If a third CPU is added, each processor must be able to translate between its own representation and that of the other two processors. Hence, in general, a heterogeneous system having n CPU types must have $n(n-1)$ pieces of translation software in order to support the facility of migrating a process from any node to any other node. An example for four processor types is shown in Figure 8.5(a). This is undesirable, as adding a new CPU type becomes a more difficult task over time.

Maguire and Smith [1988] proposed the use of the *external data representation mechanism* for reducing the software complexity of this translation process. In this mechanism, a standard representation is used for the transport of data, and each processor needs only to be able to convert data to and from the standard form. This bounds the complexity of the translation software. An example for four processor types is shown in Figure 8.5(b). The process of converting from a particular machine representation to external data representation format is called *serializing*, and the reverse process is called *deserializing*.

The standard data representation format is called *external data representation*, and its designer must successfully handle the problem of different representations for data such as characters, integers, and floating-point numbers. Of these, the handling of floating-point numbers needs special precautions. The issues discussed by Maguire and Smith [1988] for handling floating-point numbers in external data representation scheme are given below.

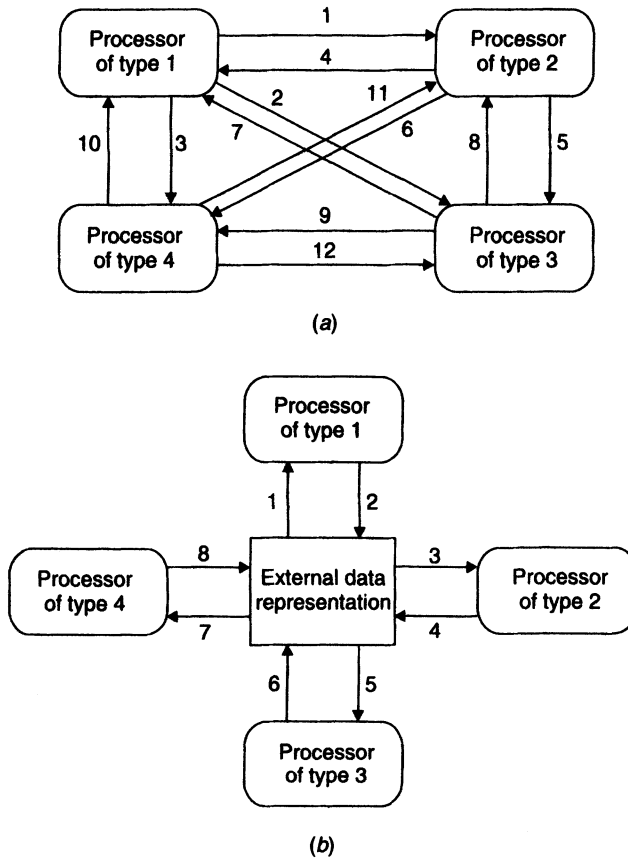


Fig. 8.5 (*a*) Example illustrating the need for 12 pieces of translation software required in a heterogeneous system having 4 types of processors. (*b*) Example illustrating the need for only 8 pieces of translation software in a heterogeneous system having 4 types of processors when the external data representation mechanism is used.

A floating-point number representation consists of an exponent part, a mantissa part, and a sign part. The issue of proper handling of the exponent and the mantissa has been described separately because the side effects caused by an external data representation affect each of the two components differently.

Handling the Exponent

The number of bits used for the exponent of a floating-point number varies from processor to processor. Let us assume that, for the exponent, processor *A* uses 8 bits, processor *B* uses 16 bits, and the external data representation designed by the users of processor

architecture *A* provides 12 bits (an extra 4 bits for safety). Also assume that all three representations use the same number of bits for the mantissa.

In this situation, a process can be migrated from processor *A* to *B* without any problem in representing its floating-point numbers because the two-step translation process of the exponent involves the conversion of 8 bits of data to 12 bits and then 12 bits of data to 16 bits, having plenty of room for the converted data in both steps. However, a process that has some floating-point data whose exponent requires more than 12 bits cannot be migrated from processor *B* to *A* because this floating-point data cannot be represented in the external data representation, which has only 12 bits for the exponent. Note that the problem here is with the design of the external data representation, which will not even allow data transfer between two processors, both of which use 16 bits for the exponent, because the external data representation has only 12 bits for this purpose. This problem can be eliminated by guaranteeing that the external data representation have at least as many bits in the exponent as the longest exponent of any processor in the distributed system.

A second type of problem occurs when a floating-point number whose exponent is less than or equal to 12 bits but greater than 8 bits is transferred from processor *B* to *A*. In this case, although the external data representation has a sufficient number of bits to handle the data, processor *A* does not. Therefore, in this case, processor *A* must raise an overflow or underflow (depending on the sign of the exponent) upon conversion from the external data representation or expect meaningless results. There are three possible solutions to this problem:

1. Ensuring that numbers used by programs that migrate have a smaller exponent value than the smallest processor's exponent value in the system
2. Emulating the larger processor's value
3. Restricting the migration of the process to only those nodes whose processor's exponent representation is at least as large as that of the source node's processor

The first solution imposes a serious restriction on the use of floating-point numbers, and this solution may be unacceptable by serious scientific computations. The second solution may be prohibitively expensive in terms of computation time. Therefore, the third solution of restricting the direction of migration appears to be a viable solution. For this, each node of the system keeps a list of nodes that can serve as destinations for migrating a process from this node.

Handling the Mantissa

The first problem in handling the mantissa is the same as that of handling the exponent. Let us assume that the exponent field is of the same size on all the processors, and for the mantissa representation, processor *A* uses 32 bits, processor *B* uses 64 bits, and the external data representation uses 48 bits. Due to similar reasons as described for handling the exponent, in this case also the migration of a process from processor *A* to *B* will have

no problem, but the migration of a process from processor *B* to *A* will result in the computation being carried out in "half-precision." This may not be acceptable when accuracy of the result is important. As in handling the exponent, to overcome this problem, the external data representation must have sufficient precision to handle the largest mantissa, and the direction of migration should be restricted only to the nodes having a mantissa at least as large as the source node.

The second problem in handling the mantissa is the loss of precision due to multiple migrations between a set of processors, where our example processors *A* and *B* might be a subset. This is a concern only in the mantissa case because loss of one or more bits of the exponent is catastrophic, while loss of bits in the mantissa only degrades the precision of computation. It may appear that the loss in precision due to multiple migrations may be cumulative, and thus a series of migrations may totally invalidate a computation. However, if the external data representation is properly designed to be adequate enough to represent the longest mantissa of any processor of the system, the resulting precision will never be worse than performing the calculation on the processor that has the least precision (least number of bits for the mantissa) among all the processors of the system. This is because the remote computations can be viewed as "extra precision" calculations with respect to the floating point of the processor with least precision.

Handling Signed-Infinity and Signed-Zero Representations

Two other issues that need to be considered in the design of external data representation are the *signed-infinity* and *signed-zero* representations. Signed infinity is a value supported by some architectures that indicates that the generated result is too large (overflow) or too small (underflow) to store. Other architectures may use the sign bit of a value that would otherwise be zero, thus giving rise to a signed zero.

Now the problem is that these representations may not be supported on all systems. Therefore, while designing the translation algorithms, proper decisions must be made about how to deal with these situations. However, in a good design, the external data representation must take care of these values so that a given processor can either take advantage of this extra information or simply discard it.

8.2.4 Advantages of Process Migration

Process migration facility may be implemented in a distributed system for providing one or more of the following advantages to the users:

1. *Reducing average response time of processes.* The average response time of the processes of a node increases rapidly as the load on the node increases. Process migration facility may be used to reduce the average response time of the processes of a heavily loaded node by migrating and processing some of its processes on a node that is either idle or whose processing capacity is underutilized.

2. *Speeding up individual jobs.* Process migration facility may be used to speed up individual jobs in two ways. The first method is to migrate the tasks of a job to the different nodes of the system and to execute them concurrently. The second approach is to migrate a job to a node having a faster CPU or to a node at which it has minimum turnaround time due to various reasons (e.g., due to specific resource requirements of the job). Of course, the gain in execution time must be more than the migration cost involved.

3. *Gaining higher throughput.* In a system that does not support process migration, it is very likely that CPUs of all the nodes are not fully utilized. But in a system with process migration facility, the capabilities of the CPUs of all the nodes can be better utilized by using a suitable load-balancing policy. This helps in improving the throughput of the system. Furthermore, process migration facility may also be used to properly mix I/O and CPU-bound processes on a global basis for increasing the throughput of the system.

4. *Utilizing resources effectively.* In a distributed system, the capabilities of the various resources such as CPUs, printers, storage devices, and so on, of different nodes are different. Therefore, depending upon the nature of a process, it can be migrated to the most suitable node to utilize the system resources in the most efficient manner. This is true not only for hardware resources but also for software resources such as databases, files, and so on. Furthermore, there are some resources, such as special-purpose hardware devices, that are not remotely accessible by a process. For example, it may be difficult to provide remote access to facilities to perform fast Fourier transforms or array processing or this access may be sufficiently slow to prohibit successful accomplishment of real-time objectives [Smith 1988]. Process migration also facilitates the use of such resources by a process of any node because the process can be migrated to the resource's location for its successful execution.

5. *Reducing network traffic.* Migrating a process closer to the resources it is using most heavily (such as files, printers, etc.) may reduce network traffic in the system if the decreased cost of accessing its favorite resources offsets the possible increased cost of accessing its less favored ones. In general, whenever a process performs data reduction (it analyzes and reduces the volume of data by generating some result) on some volume of data larger than the process's size, it may be advantageous to move the process to the location of the data [Smith 1988]. Another way to reduce network traffic by process migration is to migrate and cluster two or more processes, which frequently communicate with each other, on the same node of the system.

6. *Improving system reliability.* Process migration facility may be used to improve system reliability in several ways. One method is to simply migrate a critical process to a node whose reliability is higher than other nodes in the system. Another method is to migrate a copy of a critical process to some other node and to execute both the original and copied processes concurrently on different nodes. Finally, in failure modes such as manual shutdown, which manifest themselves as gradual degradation of a node, the processes of the node, for their continued execution, may be migrated to another node before the dying node completely fails.

7. *Improving system security.* A sensitive process may be migrated and run on a secure node that is not directly accessible to general users, thus improving the security of that process.

8.3 THREADS

Threads are a popular way to improve application performance through parallelism. In traditional operating systems the basic unit of CPU utilization is a process. Each process has its own program counter, its own register states, its own stack, and its own address space. On the other hand, in operating systems with threads facility, the basic unit of CPU utilization is a thread. In these operating systems, a process consists of an address space and one or more threads of control [Fig. 8.6(*b*)]. Each thread of a process has its own program counter, its own register states, and its own stack. But all the threads of a process share the same address space. Hence they also share the same global variables. In addition, all threads of a process also share the same set of operating system resources, such as open files, child processes, semaphores, signals, accounting information, and so on. Due to the sharing of address space, there is no protection between the threads of a process. However, this is not a problem. Protection between processes is needed because different processes may belong to different users. But a process (and hence all its threads) is always owned by a single user. Therefore, protection between multiple threads of a process is not necessary. If protection is required between two threads of a process, it is preferable to put them in different processes, instead of putting them in a single process.

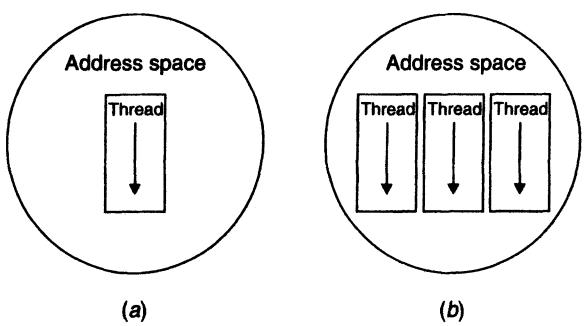


Fig. 8.6 (*a*) Single-threaded and (*b*) multithreaded processes. A single-threaded process corresponds to a process of a traditional operating system.

Threads share a CPU in the same way as processes do. That is, on a uniprocessor, threads run in quasi-parallel (time sharing), whereas on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors. Moreover, like traditional processes, threads can create child threads, can block waiting for system calls to complete, and can change states during their course of execution. At a particular instance of time, a thread can be in any one of several states: running, blocked, ready, or terminated. Due to these similarities, threads are often viewed as miniprocesses. In fact, in operating systems

with threads facility, a process having a single thread corresponds to a process of a traditional operating system [Fig. 8.6(*a*)]. Threads are often referred to as *lightweight processes* and traditional processes are referred to as *heavyweight processes*.

8.3.1 Motivations for Using Threads

The main motivations for using a multithreaded process instead of multiple single-threaded processes for performing some computation activities are as follows:

1. The overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process.
2. Switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address spaces.
3. Threads allow parallelism to be combined with sequential execution and blocking system calls [Tanenbaum 1995]. Parallelism improves performance and blocking system calls make programming easier.
4. Resource sharing can be achieved more efficiently and naturally between threads of a process than between processes because all threads of a process share the same address space.

These advantages are elaborated below.

The overheads involved in the creation of a new process and building its execution environment are liable to be much greater than creating a new thread within an existing process. This is mainly because when a new process is created, its address space has to be created from scratch, although a part of it might be inherited from the process's parent process. However, when a new thread is created, it uses the address space of its process that need not be created from scratch. For instance, in case of a kernel-supported virtual-memory system, a newly created process will incur page faults as data and instructions are referenced for the first time. Moreover, hardware caches will initially contain no data values for the new process, and cache entries for the process's data will be created as the process executes. These overheads may also occur in thread creation, but they are liable to be less. This is because when the newly created thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

Threads also minimize context switching time, allowing the CPU to switch from one unit of computation to another unit of computation with minimal overhead. Due to the sharing of address space and other operating system resources among the threads of a process, the overhead involved in CPU switching among peer threads is very small as compared to CPU switching among processes having their own address spaces. This is the reason why threads are called lightweight processes.

To clarify how threads allow parallelism to be combined with sequential execution and blocking system calls, let us consider the different ways in which a server process can be constructed. One of the following three models may be used to construct a server process (e.g., let us consider the case of a file server) [Tanenbaum 1995]:

1. *As a single-thread process.* This model uses blocking system calls but without any parallelism. In this method, the file server gets a client's file access request from the request queue, checks the request for access permissions, and if access is allowed, checks whether a disk access is needed to service the request. If disk access is not needed, the request is serviced immediately and a reply is sent to the client process. Otherwise, the file server sends a disk access request to the disk server and waits for a reply. After receiving the disk server's reply, it services the client's request, sends a reply to the client process, and goes back to get the next request from the request queue.

In this method, the programming of the server process is simple because of the use of blocking system call; after sending its request, the file server blocks until a reply is received from the disk server. However, if a dedicated machine is used for the file server, the CPU remains idle while the file server is waiting for a reply from the disk server. Hence, no parallelism is achieved in this method and fewer client requests are processed per unit of time.

The performance of a server implemented as a single-thread process is often unacceptable. Therefore, it is necessary to overlap the execution of multiple client requests by allowing the file server to work on several requests simultaneously. The next two models support parallelism for this purpose.

2. *As a finite-state machine.* This model supports parallelism but with nonblocking system calls. In this method, the server is implemented as a single-threaded process and is operated like a finite-state machine. An event queue is maintained in which both client request messages and reply messages from the disk server are queued. Whenever the thread becomes idle, it takes the next message from the event queue. If it is a client request message, a check is made for access permission and need for disk access. If disk access is needed to service the request, the file server sends a disk access request message to the disk server. However, this time, instead of blocking, it records the current state of the client's request in a table and then goes to get the next message from the event queue. This message may either be a request from a new client or a reply from the disk server of a previous disk access request. If it is a new client request, it is processed as described above. On the other hand, if it is a reply from the disk server, the state of the client's request that corresponds to the reply is retrieved from the table, and the client's request is processed further.

Although the method achieves parallelism, it is difficult to program the server process in this method due to the use of nonblocking system calls. The server process must maintain entries for every outstanding client request, and whenever a disk operation completes, the appropriate piece of client state must be retrieved to find out how to continue carrying out the request.

3. *As a group of threads.* This model supports parallelism with blocking system calls. In this method, the server process is comprised of a single *dispatcher* thread and multiple *worker* threads. Either the worker threads can be created dynamically, whenever a request comes in, or a pool of threads can be created at start-up time to deal with as many simultaneous requests as there are threads. The dispatcher thread keeps waiting in a loop for requests from the clients. When a client request arrives, it checks it for access permission. If permission is allowed, it either creates a new worker

thread or chooses an idle worker thread from the pool (depending on whether the worker threads are created dynamically or statically) and hands over the request to the worker thread. The control is then passed on to the worker thread and the dispatcher thread's state changes from running to ready. Now the worker thread checks to see if a disk access is needed for the request or if it can be satisfied from the block cache that is shared by all the threads. If disk access is needed, it sends a disk access request to the disk server and blocks while waiting for a reply from the disk server. At this point, the scheduler will be invoked and a thread will be selected to be run from the group of threads that are in the ready state. The selected thread may be the dispatcher thread or another worker thread that is now ready to run.

This method achieves parallelism while retaining the idea of sequential processes that make blocking system calls. Therefore, a server process designed in this way has good performance and is also easy to program.

We saw the motivation for using threads in the design of server processes. Often there are some situations where client processes can also benefit from the concurrency made possible by threads. For example, a client process may use a divide-and-conquer algorithm to divide data into blocks that can be processed separately. It can then send each block to a server to be processed and finally collect and combine the results. In this case, a separate client thread may be used to handle each data block to interact with the different server processes. Similarly, when a file is to be replicated on multiple servers, a separate client thread can be used to interact with each server. Some client application user interfaces can also benefit by using threads to give the interface back to a user while a long operation takes place. Client processes that perform lots of distributed operations can also benefit from threads by using a separate thread to monitor each operation.

Finally, the use of threads is also motivated by the fact that a set of threads using a shared address space is the most natural way to program many applications. For example, in an application that uses the producer-consumer model, the producer and the consumer processes must share a common buffer. Therefore, programming the application in such a way that the producer and consumer are two threads of the same process makes the software design simpler.

8.3.2 Models for Organizing Threads

Depending on an application's needs, the threads of a process of the application can be organized in different ways. Three commonly used ways to organize the threads of a process are as follows [Tanenbaum 1995]:

1. *Dispatcher-workers model.* We have already seen the use of this model in designing a server process. In this model, the process consists of a single dispatcher thread and multiple worker threads. The dispatcher thread accepts requests from clients and, after examining the request, dispatches the request to one of the free worker threads for further processing of the request. Each worker thread works on a different client request. Therefore multiple client requests can be processed in parallel. An example of this model is shown in Figure 8.7(*a*).

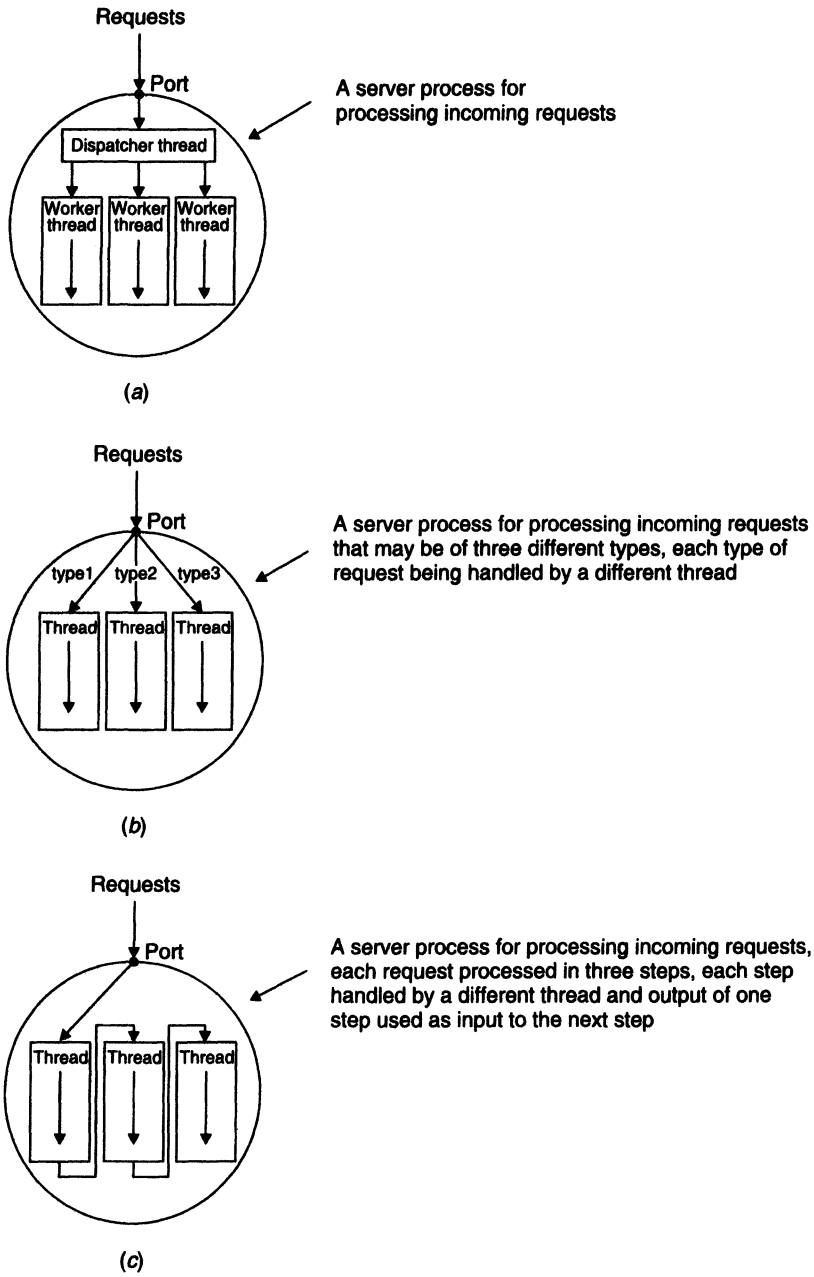


Fig. 8.7 Models for organizing threads: (*a*) dispatcher-workers model; (*b*) team model; (*c*) pipeline model.

2. *Team model.* In this model, all threads behave as equals in the sense that there is no dispatcher-worker relationship for processing clients' requests. Each thread gets and processes clients' requests on its own. This model is often used for implementing specialized threads within a process. That is, each thread of the process is specialized in servicing a specific type of request. Therefore, multiple types of requests can be simultaneously handled by the process. An example of this model is shown in Figure 8.7(*b*).

3. *Pipeline model.* This model is useful for applications based on the producer-consumer model, in which the output data generated by one part of the application is used as input for another part of the application. In this model, the threads of a process are organized as a pipeline so that the output data generated by the first thread is used for processing by the second thread, the output of the second thread is used for processing by the third thread, and so on. The output of the last thread in the pipeline is the final output of the process to which the threads belong. An example of this model is shown in Figure 8.7(*c*).

8.3.3 Issues in Designing a Threads Package

A system that supports threads facility must provide a set of primitives to its users for threads-related operations. These primitives of the system are said to form a *threads package*. Some of the important issues in designing a threads package are described below.

Threads Creation

Threads can be created either statically or dynamically. In the static approach, the number of threads of a process remains fixed for its entire lifetime, while in the dynamic approach, the number of threads of a process keeps changing dynamically. In the dynamic approach, a process is started with a single thread, new threads are created as and when needed during the execution of the process, and a thread may destroy itself when it finishes its job by making an exit call. On the other hand, in the static approach, the number of threads of a process is decided either at the time of writing the corresponding program or when the program is compiled. In the static approach, a fixed stack is allocated to each thread, but in the dynamic approach, the stack size of a thread is specified as a parameter to the system call for thread creation. Other parameters usually required by this system call include scheduling priority and the procedure to be executed to run this thread. The system call returns a thread identifier for the newly created thread. This identifier is used in subsequent calls involving this thread.

Threads Termination

Termination of threads is performed in a manner similar to the termination of conventional processes. That is, a thread may either destroy itself when it finishes its job by making an exit call or be killed from outside by using the kill command and specifying the thread

identifier as its parameter. In many cases, threads are never terminated. For example, we saw above that in a process that uses statically created threads, the number of threads remains constant for the entire life of the process. In such a process, all its threads are created immediately after the process starts up and then these threads are never killed until the process terminates.

Threads Synchronization

Since all the threads of a process share a common address space, some mechanism must be used to prevent multiple threads from trying to access the same data simultaneously. For example, suppose two threads of a process need to increment the same global variable within the process. For this to occur safely, each thread must ensure that it has exclusive access for this variable for some period of time. A segment of code in which a thread may be accessing some shared variable is called a *critical region*. To prevent multiple threads from accessing the same data simultaneously, it is sufficient to ensure that when one thread is executing in a critical region, no other thread is allowed to execute in a critical region in which the same data is accessed. That is, the execution of critical regions in which the same data is accessed by the threads must be *mutually exclusive* in time. Two commonly used mutual exclusion techniques in a threads package are mutex variables and condition variables.

A *mutex variable* is like a binary semaphore that is always in one of two states, locked or unlocked. A thread that wants to execute in a critical region performs a *lock* operation on the corresponding mutex variable. If the mutex variable is in the unlocked state, the *lock* operation succeeds and the state of the mutex variable changes from unlocked to locked in a single atomic action. After this, the thread can execute in the critical region. However, if the mutex variable is already locked, depending on the implementation, the lock operation is handled in one of the following ways:

1. The thread is blocked and entered in a queue of threads waiting on the mutex variable.
2. A status code indicating failure is returned to the thread. In this case, the thread has the flexibility to continue with some other job. However, to enter the critical region, the thread has to keep retrying to lock the mutex variable until it succeeds.

A threads package may support both by providing different operations for actually locking and obtaining the status of a mutex variable.

In a multiprocessor system in which different threads run in parallel on different CPUs, it may happen that two threads perform lock operations on the same mutex variable simultaneously. In such a situation, one of them wins, and the loser is either blocked or returned a status code indicating failure.

When a thread finishes executing in its critical region, it performs an *unlock* operation on the corresponding mutex variable. At this time, if the blocking method is used and if one or more threads are blocked waiting on the mutex variable, one of them is given the lock and its state is changed from blocked to running while others continue to wait.

Mutex variables are simple to implement because they have only two states. However, their use is limited to guarding entries to critical regions. For more general synchronization requirements *condition variables* are used. A condition variable is associated with a mutex variable and reflects a Boolean state of that variable. *Wait* and *signal* are two operations normally provided for a condition variable. When a thread performs a *wait* operation on a condition variable, the associated mutex variable is unlocked, and the thread is blocked until a *signal* operation is performed by some other thread on the condition variable, indicating that the event being waited for may have occurred. When a thread performs a *signal* operation on the condition variable, the mutex variable is locked, and the thread that was blocked waiting on the condition variable starts executing in the critical region. Figure 8.8 illustrates the use of mutex variable and condition variable for synchronizing threads.

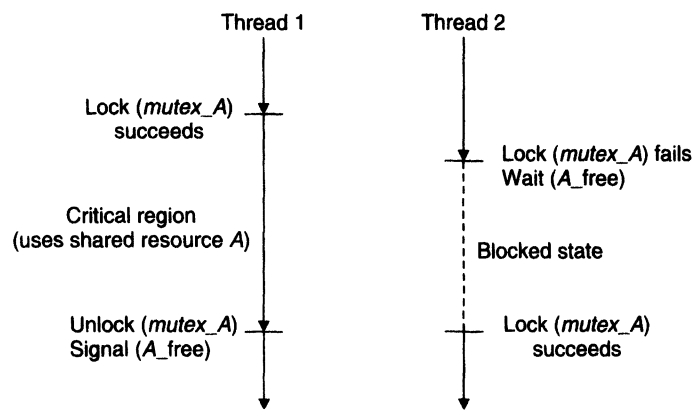


Fig. 8.8 Use of mutex variable and condition variable for synchronizing threads.

Condition variables are often used for cooperation between threads. For example, in an application in which two threads of a process have a producer-consumer relationship, the producer thread creates data and puts it in a bounded buffer, and the consumer thread takes the data from the buffer and uses it for further processing. In this case, if the buffer is empty when the consumer thread checks it, that thread can be made to wait on a *nonempty* condition variable, and when the producer thread puts some data in the buffer, it can signal the *nonempty* condition variable. Similarly, if the buffer is full when the producer thread checks it, that thread can be made to wait on a *nonfull* condition variable, and when the consumer thread takes out some data from the buffer, it can signal the *nonfull* condition variable. In this way, the two threads can work in cooperation with each other by the use of condition variables.

Threads Scheduling

Another important issue in the design of a threads package is how to schedule the threads. Threads packages often provide calls to give the users the flexibility to specify the scheduling policy to be used for their applications. With this facility, an application programmer can use the heuristics of the problem to decide the most effective manner for scheduling. Some of the special features for threads scheduling that may be supported by a threads package are as follows:

1. *Priority assignment facility.* In a simple scheduling algorithm, threads are scheduled on a first-in, first-out basis or the round-robin policy is used to timeshare the CPU cycles among the threads on a quantum-by-quantum basis, with all threads treated as equals by the scheduling algorithm. However, a threads-scheduling scheme may provide the flexibility to the application programmers to assign priorities to the various threads of an application in order to ensure that important ones can be run on a higher priority basis.

A priority-based threads scheduling scheme may be either non-preemptive or preemptive. In the former case, once a CPU is assigned to a thread, the thread can use it until it blocks, exits, or uses up its quantum. That is, the CPU is not taken away from the thread to which it has already been assigned even if another higher priority thread becomes ready to run. The higher priority thread is selected to run only after the thread that is currently using the CPU releases it. On the other hand, in the preemptive scheme, a higher priority thread always preempts a lower priority one. That is, whenever a higher priority thread becomes ready to run, the currently running lower priority thread is suspended, and the CPU is assigned to the higher priority thread. In this scheme, a thread can run only when no other higher priority thread is ready to run.

2. *Flexibility to vary quantum size dynamically.* A simple round-robin scheduling scheme assigns a fixed-length quantum to timeshare the CPU cycles among the threads. However, a fixed-length quantum is not appropriate on a multiprocessor system because there may be fewer runnable threads than there are available processors. In this case, it would be wasteful to interrupt a thread with a context switch to the kernel when its quantum runs out only to have it placed right back in the running state. Therefore, instead of using a fixed-length quantum, a scheduling scheme may vary the size of the time quantum inversely with the total number of threads in the system. This algorithm gives good response time to short requests, even on heavily loaded systems, but provides high efficiency on lightly loaded systems.

3. *Handoff scheduling.* A handoff scheduling scheme allows a thread to name its successor if it wants to. For example, after sending a message to another thread, the sending thread can give up the CPU and request that the receiving thread be allowed to run next. Therefore, this scheme provides the flexibility to bypass the queue of runnable threads and directly switch the CPU to the thread specified by the currently running thread. Handoff scheduling can enhance performance if it is wisely used.

4. *Affinity scheduling.* Another scheduling policy that may be used for better performance on a multiprocessor system is affinity scheduling. In this scheme, a thread is scheduled on the CPU it last ran on in hopes that part of its address space is still in that CPU's cache.

Signal Handling

Signals provide software-generated interrupts and exceptions. Interrupts are externally generated disruptions of a thread or process, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. The two main issues associated with handling signals in a multithreaded environment are as follows:

1. A signal must be handled properly no matter which thread of the process receives it. Recall that in UNIX a signal's handler must be a routine in the process receiving the signal.
2. Signals must be prevented from getting lost. A signal gets lost when another signal of the same type occurs in some other thread before the first one is handled by the thread in which it occurred. This happens because an exception condition causing the signal is stored in a processwide global variable that is overwritten by another exception condition causing a signal of the same type.

An approach for handling the former issue is to create a separate exception handler thread in each process. In this approach, the exception handler thread of a process is responsible for handling all exception conditions occurring in any thread of the process. When a thread receives a signal for an exception condition, it sends an exception occurrence message to the exception handler thread and waits until the exception is handled. The exception message usually includes information about the exception condition, the thread, and the process that caused the exception. The exception handler thread performs its function according to the type of exception. This may involve such actions as clearing the exception, causing the victim thread to resume, or terminating the victim thread.

On the other hand, an approach for handling the latter issue is to assign each thread its own private global variables for signaling exception conditions, so that conflicts between threads over the use of such global variables never occur. Such variables are said to be threadwide global because the code of a thread normally consists of multiple procedures. In this approach, new library procedures are needed to create, set, and read these threadwide global variables.

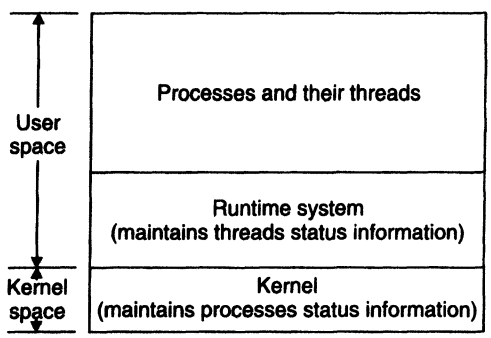
8.3.4 Implementing a Threads Package

A threads package can be implemented either in user space or in the kernel. In the description below, the two approaches are referred to as user-level and kernel-level, respectively. In the user-level approach, the user space consists of a runtime system that is a collection of threads management routines. Threads run in the user space on top of the runtime system and are managed by it. The runtime system also maintains a status information table to keep track of the current status of each thread. This table has one entry per thread. An entry of this table has fields for registers' values, state, priority, and other information of a thread. All calls of the threads package are implemented as calls to the runtime system procedures that perform the functions corresponding to the calls. These procedures also perform thread switching if the

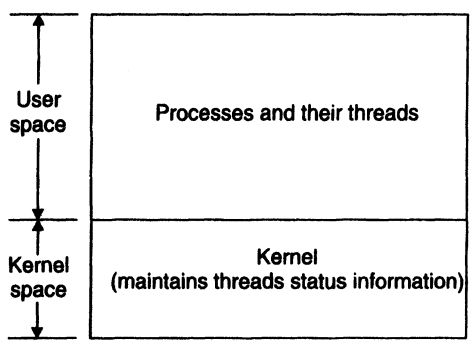
thread that made the call has to be suspended during the call. That is, two-level scheduling is performed in this approach. The scheduler in the kernel allocates quanta to heavyweight processes, and the scheduler of the runtime system divides a quantum allocated to a process among the threads of that process. In this manner, the existence of threads is made totally invisible to the kernel. The kernel functions in a manner similar to an ordinary kernel that manages only single-threaded, heavyweight processes. This approach is used by the SunOS 4.1 Lightweight Processes package.

On the other hand, in the kernel-level approach, no runtime system is used and the threads are managed by the kernel. Therefore, the threads status information table is maintained within the kernel. All calls that might block a thread are implemented as system calls that trap to the kernel. When a thread blocks, the kernel selects another thread to be run. The selected thread may belong to either the same process as that of the previously running thread or a different process. Hence, the existence of threads is known to the kernel, and single-level scheduling is used in this approach.

Figure 8.9 illustrates the two approaches for implementing a threads package. The relative advantages and disadvantages of the approaches are as follows:



(*a*)



(*b*)

Fig. 8.9 Approaches for implementing a threads package: (*a*) user level; (*b*) Kernel level.

1. The most important advantage of the user-level approach is that a threads package can be implemented on top of an existing operating system that does not support threads. This is not possible in the kernel-level approach because in this approach the concept of threads must be incorporated in the design of the kernel of an operating system.

2. In the user-level approach, due to the use of two-level scheduling, users have the flexibility to use their own customized algorithm to schedule the threads of a process. Therefore, depending on the needs of an application, a user can design and use the most appropriate scheduling algorithm for the application. This is not possible in the kernel-level approach because a single-level scheduler is used that is built into the kernel. Therefore, users only have the flexibility to specify through the system call parameters the priorities to be assigned to the various threads of a process and to select an existing algorithm from a set of already implemented scheduling algorithms.

3. Switching the context from one thread to another is faster in the user-level approach than in the kernel-level approach. This is because in the former approach context switching is performed by the runtime system, while in the latter approach a trap to the kernel is needed for it.

4. In the kernel-level approach, the status information table for threads is maintained within the kernel. Due to this, the scalability of the kernel-level approach is poor as compared to the user-level approach.

5. A serious drawback associated with the user-level approach is that with this approach the use of round-robin scheduling policy to timeshare the CPU cycles among the threads on a quantum-by-quantum basis is not possible [Tanenbaum 1995]. This is due to the lack of clock interrupts within a single process. Therefore, once a thread is given the CPU to run, there is no way to interrupt it, and it continues to run unless it voluntarily gives up the CPU. This is not the case with the kernel-level approach, in which clock interrupts occur periodically, and the kernel can keep track of the amount of CPU time consumed by a thread. When a thread finishes using its allocated quantum, it can be interrupted by the kernel, and the CPU can be given to another thread.

A crude way to solve this problem is to have the runtime system request a clock interrupt after every fixed unit of time (say every half a second) to give it control [Tanenbaum 1995]. When the runtime system gets control, the scheduler can decide if the thread should continue running or the CPU should now be allocated to another thread.

6. Another drawback of the user-level approach is associated with the implementation of blocking system calls. In the kernel-level approach, implementation of blocking system calls is straightforward because when a thread makes such a call, it traps to the kernel, where it is suspended, and the kernel starts a new thread. However, in the user-level approach, a thread should not be allowed to make blocking system calls directly. This is because if a thread directly makes a blocking system call, all threads of its process will be stopped, and the kernel will schedule another process to run. Therefore, the basic purpose of using threads will be lost.

A commonly used approach to overcome this problem is to use jacket routines. A *jacket routine* contains extra code before each blocking system call to first make a check

to ensure if the call will cause a trap to the kernel. The call is made only if it is safe (will not cause a trap to the kernel); otherwise the thread is suspended and another thread is scheduled to run. Checking the safety condition and making the actual call must be done atomically.

With some success, few attempts have been made to combine the advantages of user-level and kernel-level approaches in the implementation of a threads package. For example, the FastThreads package [Anderson et al. 1991] and the threads package of the Psyche multiprocessor operating system [Marsh et al. 1991] provide kernel support for user-level thread scheduling. On the other hand, Mach [Black 1990] enables user-level code to provide scheduling hints to the kernel's thread scheduler. The details of threads implementation in Mach is given in Chapter 12.

8.3.5 Case Study: DCE Threads

The C Threads package developed for the Mach operating system and the Lightweight Processes package developed for the SunOS are two examples of commercially available threads packages. In addition, to avoid incompatibilities between threads designs, IEEE has drafted a POSIX (Portable Operating System Interface for Computer Environments) threads standard known as *P-Threads*. Two implementations of this standard are the DCE Threads developed by the Open Software Foundation (OSF) for the OSF/1 operating system and the GNU Threads developed by the Free Software Foundation for the SunOS. The DCE Threads package, which is based on the P1003.4a POSIX standard, is described below as a case study. A description of another threads package, the C Threads package, is given in Chapter 12 as a part of the description of the Mach operating system.

The user-level approach is used for implementing the DCE Threads package. That is, DCE provides a set of user-level library procedures for the creation, termination, synchronization, and so on, of threads. To access thread services from applications written in C, DCE specifies an application programming interface (API) that is compatible to the POSIX standard. If a system supporting DCE has no intrinsic support for threads, the API provides an interface to the DCE threads library that is linked to application procedures. On the other hand, if a system supporting DCE has operating system kernel support for threads, the DCE is set up to use this facility. In this case, the API serves as an interface to the kernel-supported threads facility.

Threads Management

The DCE Threads package has a number of library procedures for managing threads. Some of the important ones are as follows:

- *pthread_create* is used to create a new thread in the same address space as the calling thread. The thread executes concurrently with its parent thread. However, instead of executing the parent's code, it executes a procedure whose name is specified as an input parameter to the *pthread_create* routine.

- *pthread_exit* is used to terminate the calling thread. This routine is called by a thread when it has finished doing its work.
- *pthread_join* is used to cause the calling thread to block itself until the thread specified in this routine's argument terminates. This routine is similar to the *wait* system call of UNIX and may be used by a parent thread to wait for a child thread to complete execution.
- *pthread_detach* is used by a parent thread to disown a child thread. By calling *pthread_detach*, the parent thread announces that the specified child thread will never be *pthread_joined* (waited for). If the child thread ever calls *pthread_exit*, its stack and other state information are immediately reclaimed. In normal cases, this cleanup takes place after the parent has done a successful *pthread_join*.
- *pthread_cancel* is used by a thread to kill another thread.
- *pthread_setcancel* is used by a thread to enable or disable ability of other threads to kill it. It allows a thread to prevent it from getting killed by another thread at such times when killing the thread might have devastating effects, for example, if the thread has a mutex variable locked at the time.

Threads Synchronization

The DCE Threads package provides support for both mutex variables and condition variables for threads synchronization. Mutex variables are used when access to a shared resource by multiple threads must be mutually exclusive in time. On the other hand, condition variables are used with mutex variables to allow threads to block and wait for a shared resource already locked by another thread until the thread using it unlocks it and signals the waiting thread.

The DCE Threads package supports the following types of mutex variables, which differ in how they deal with nested locks:

1. *Fast.* A fast mutex variable is one that causes a thread to block when the thread attempts to lock an already locked mutex variable. That is, nested locking of a fast mutex variable is not permitted. Note that fast mutex variables may lead to deadlock. For instance, if a thread tries to lock the same mutex variable a second time, a deadlock will occur.

2. *Recursive.* A recursive mutex variable is one that allows a thread to lock an already locked mutex variable. That is, nested locking of a recursive mutex variable is permitted with arbitrarily deep nestings. Notice that recursive mutex variables will never lead to deadlock. It is the responsibility of the application programmers to ultimately unlock a recursive mutex variable as many times as it is locked.

3. *Nonrecursive.* A nonrecursive mutex variable is one that neither allows a thread to lock an already locked mutex variable nor causes the thread to block. Rather, an error is returned to the thread that attempts to lock an already locked nonrecursive mutex variable. Notice that nonrecursive mutex variables avoid the deadlock problem associated with fast mutex variables.

Some of the main DCE thread calls for threads synchronization are as follows:

- *pthread_mutex_init* is used to dynamically create a mutex variable.
- *pthread_mutex_destroy* is used to dynamically delete a mutex variable.
- *pthread_mutex_lock* is used to lock a mutex variable. If the specified mutex variable is already locked, the thread that makes this call is blocked until the mutex variable is unlocked.
- *pthread_mutex_trylock* is used to make an attempt to lock a mutex variable. If the mutex variable is already locked, the call returns with an unsuccessful result rather than causing the thread to block.
- *pthread_mutex_unlock* is used to unlock a mutex variable.
- *pthread_cond_init* is used to dynamically create a condition variable.
- *pthread_cond_destroy* is used to dynamically delete a condition variable.
- *pthread_cond_wait* is used to wait on a condition variable. The calling thread blocks until a *pthread_cond_signal* or a *pthread_cond_broadcast* is executed for the condition variable.
- *pthread_cond_signal* is used to wake up a thread waiting on the condition variable. If multiple threads are waiting on the condition variable, only one thread is awakened; others continue to wait.
- *pthread_cond_broadcast* is used to wake up all the threads waiting on the condition variable.

Another area where mutual exclusion is needed is in the use of UNIX library procedures. The standard library procedures of UNIX are not reentrant. Therefore, to prevent inconsistencies that may be caused by threads switching occurring at arbitrary points in time, it is necessary to provide mutual exclusion for the individual calls.

DCE solves this problem by providing jacket routines for a number of nonreentrant UNIX system calls (mostly I/O procedures). Threads call the jacket routines instead of the UNIX system calls. The jacket routines take necessary action on behalf of the thread before or after invoking the system call to avoid any potential problem. For example, the jacket routines ensure that only one thread calls any particular service at a time.

For several other UNIX procedures, DCE provides a single global mutex variable to ensure that only one thread at a time is active in the library.

Threads Scheduling

The DCE Threads package supports priority-based threads scheduling. It allows the users to specify not only the priorities for individual threads but also the scheduling algorithm to be used so that important threads can take priority over other threads, getting the necessary CPU time whenever they need it. A user can choose from one of the following threads-scheduling algorithms:

1. *First in, first out (FIFO)*. In this method, the first thread of the first nonempty highest priority queue is always selected to run. The selected thread continues to execute

until it either blocks or exits. After the thread finishes execution, the same method is used to select a new thread for CPU allocation. The algorithm may cause starvation of low-priority threads.

2. *Round robin (RR).* In this method, also, the first nonempty highest priority queue is located. However, instead of running the first thread on this queue to completion, all the threads on this queue are given equal importance by running each thread for a fixed quantum in a round-robin fashion. This algorithm may also cause starvation of low-priority threads.

3. *Default.* In this method, the threads on all the priority queues are run one after another using a time-sliced, round-robin algorithm. The quantum allocated to a thread varies depending on its priority; the higher the priority, the larger the quantum. Notice that in this algorithm there is no starvation because all threads get to run.

The following system calls allow users to select a scheduling algorithm of their choice and to manipulate individual threads priorities:

- *pthread_setscheduler* is used to select a scheduling algorithm.
- *pthread_getscheduler* is used to know which scheduling algorithm is currently in effect.
- *pthread_setprio* is used to set the scheduling priority of a thread.
- *pthread_getprio* is used to know the scheduling priority of a thread.

Signal Handling

Signals may be generated due to either an exception condition occurring during a thread's execution, such as a segmentation violation, or a floating-point exception or due to an external interrupt, such as when the user intentionally interrupts the running process by hitting the appropriate key on the keyboard. In DCE, an exception condition is handled by the thread in which it occurs. However, an external interrupt is handled by all the concerned threads. That is, when an external interrupt occurs, the threads package passes it to all the threads that are waiting for the interrupt.

DCE also includes the POSIX *sigwait* and *sigaction* services that may be used for signal handling instead of catching signal handlers in the traditional way. These services operate at a different level than signal handlers but can achieve the same results. The *sigwait* service allows a thread to block until one of a specified set of interrupt-based signals (also known as asynchronous signals) is delivered. On the other hand, the *sigaction* service allows for per-thread handlers to be installed for catching exception-based signals (also known as synchronous signals).

Error Handling

The UNIX system calls as well as the standard P1003.4a P-Threads calls report errors by setting a global variable, *errno*, and returning −1. In this method of handling errors, an error may get lost when the global *errno* variable is overwritten by an error occurring in

some other thread before the previous *errno* value is seen and handled by the thread in which it occurred. To overcome this problem, each thread in DCE has its own private *errno* variable for storing its own error status. This variable is saved and restored along with other thread-specific items upon thread switches. The error-handling interface of DCE allows the programmers to inspect the value of this variable. Another method that may be used in DCE for error handling is to have system calls raise exceptions when errors occur.

8.4 SUMMARY

This chapter has presented a description of the two important process management concepts in distributed operating systems: process migration and threads.

Process migration deals with the transparent relocation of a process from one node to another in a distributed system. A process may be relocated before it starts executing or during the course of its execution. The former is called non-preemptive process migration and the latter is known as preemptive process migration. Preemptive process migration is costlier than non-preemptive process migration because the handling of the process's state, which must accompany the process to its new node, becomes much more complex after execution begins.

Process migration policy deals with the selection of a source node from which to migrate a process, a destination node to which the process will be migrated, and the migrant process. These selection decisions are taken by a suitable global scheduling algorithm used for the process migration policy. On the other hand, a process migration mechanism deals with the actual transfer of the process from its source node to its destination node and the forwarding and handling of related messages during and after migration. The commonly used method for preemptive process migration is to freeze the process on its source node, transfer its state information to its destination node, and restart the process on its destination node using this state information.

The cost of migrating a process is dominated by the time taken to migrate its address space. Total freezing, pretransferring, and transfer on reference are the mechanisms used by the existing systems for address space transfer. The different mechanisms used for message forwarding in the existing distributed systems are the mechanism of resending the message, the origin site mechanism, the link traversal mechanism, and the link update mechanism. The two different mechanisms for communication between a process and its subprocesses that might have been migrated and placed on different nodes are the logical host concept and the home node or origin site concept.

Process migration in heterogeneous systems becomes more complex than in homogeneous systems due to the need for data translation from the source node data format to the destination node data format. The external data representation mechanism helps in reducing the software complexity of this translation process. Of the various types of data, such as characters, integers, and floating-point numbers, the handling of floating-point numbers needs special precautions.

The existing implementations of process migration facility have shown that preemptive process migration is possible, although with higher overhead and complexity

than originally anticipated. The cost of migration led some system designers to conclude that it is not a viable alternative, while others disagree. However, the topic is still an active research area with mixed reactions.

Threads are an increasingly popular way to improve application performance through parallelism. In operating systems with threads facility, a process consists of an address space and one or more threads of control. Each thread of a process has its own program counter, its own register states, and its own stack. But all the threads of a process share the same address space. Threads are often referred to as lightweight processes, and traditional processes are referred to as heavyweight processes.

A major motivation for threads is to minimize context switching time, allowing the CPU to switch from one unit of computation to another unit of computation with minimal overhead. Another important motivation for threads is to allow parallelism to be combined with sequential execution and blocking system calls. The use of threads is also motivated by the fact that a set of threads using a shared address space is the most natural way to program many applications.

The three commonly used ways to organize the threads of a process are the dispatcher-workers model, the team model, and the pipeline model.

The set of primitives provided to users for threads-related operations are said to form a threads package. Some of the important issues in designing a threads package are creation, termination, synchronization, and scheduling of threads and handling of signals and errors.

A threads package can be implemented either in user space or in the kernel. Both approaches have their own advantages and limitations.

EXERCISES

8.1. Differentiate between preemptive and non-preemptive process migration. What are their relative advantages and disadvantages? Suppose you have to design a process migration facility for a distributed system. What factors will influence your decision to design a preemptive or a non-preemptive process migration facility?

8.2. What are some of the main issues involved in freezing a migrant process on its source node and restarting it on its destination node? Give a method for handling each of these issues.

8.3. What are the main similarities and differences between the implementation of the following two activities:

(a) Interrupting a process to execute a higher priority process and then restarting the interrupted process after some time on the same node

(b) Freezing a migrant process and then restarting it on a different node

8.4. From the point of view of supporting preemptive process migration facility, is a stateless or stateful file server preferable? Give reasons for your answer.

8.5. When a migrant process is restarted on its destination node after migration, it is given the same process identifier that it had on its source node. Is this necessary? Give reasons for your answer.

8.6. The cost of migrating a process is dominated by the time taken to transfer its address space. Suggest some methods that may be used to minimize this cost.

8.7. A distributed system supports DSM (Distributed Shared Memory) facility. Suggest a suitable address space transfer mechanism that you will use to design a process migration facility for this system.

8.8. Which one or more of the address space transfer mechanisms described in this chapter are suitable for a process migration facility with the following goals?
(a) High performance is the main goal.
(b) High reliability is the main goal.
(c) Effectiveness of process migration policy is the main goal.
(d) Simple implementation is the main goal.
(e) Both reliability and effectiveness of process migration policy are important goals.
If more than one mechanism is suitable for a particular case, which one will you prefer to use and why?

8.9. Which one or more of the message-forwarding mechanisms described in this chapter are suitable for a process migration facility with the following goals?
(a) Transparency is the main goal.
(b) Reliability is the main goal.
(c) Performance is the main goal.
(d) Simple implementation is the main goal.
If more than one mechanisms are suitable for a particular case, which one will you prefer to use and why?

8.10. Which of the mechanisms described in this chapter to handle communication among coprocesses are suitable for a process migration facility with the following goals?
(a) Performance is the main goal.
(b) Reliability is the main goal.
(c) Simple implementation is the main goal.
If more than one mechanisms are suitable for a particular case, which one will you prefer to use and why?

8.11. What are some of the main issues involved in designing a process migration facility for a heterogeneous distributed system?

8.12. The process migration facility of a distributed system does not allow free migration of processes from one node to another but has certain restrictions regarding which node's processes can be migrated to which other nodes of the system. What might be the reasons behind imposing such a restriction?

8.13. When should the external data representation mechanism be used in the design of a process migration facility? Suppose you have to design the external data representation format for a process migration facility. What important factors will influence your design decisions?

8.14. A distributed system has three types of processors *A*, *B*, and *C*. The numbers of bits used for the exponent of a floating-point number by processors of types *A*, *B*, and *C* are 8, 12, and 16, respectively; the numbers of bits used for the mantissa of a floating-point number by processors of types *A*, *B*, and *C* are 16, 32, and 64, respectively. In this system, from which processor type to which processor type should process migration be allowed and from which processor type to which processor type should processor migration not be allowed? Give reasons for your answer.

8.15. List some of the potential advantages and disadvantages of process migration.

8.16. In operating systems in which a process is the basic unit of CPU utilization, mechanisms are provided to protect a process from other processes. Do operating systems in which a thread is the basic unit of CPU utilization need to provide similar mechanisms to protect a thread from other threads? Give reasons for your answer.

8.17. The concept of threads is often used in distributed operating systems for better performance. Can this concept also be useful for better performance in other multiprocessor operating systems and in operating systems for conventional centralized time-sharing systems? Give reasons for your answer.

8.18. List the main differences and similarities between threads and processes.

8.19. What are the main advantages and disadvantages of using threads instead of multiple processes? Give an example of an application that would benefit from the use of threads and another application that would not benefit from the use of threads.

8.20. In a distributed system, parallelism improves performance and blocking system calls make programming easier. Explain how the concept of threads can be used to combine both advantages.

8.21. Give an example to show how a server process can be designed to benefit from the concurrency made possible by threads. Now give an example to show how a client process can be designed to benefit from the concurrency made possible by threads.

8.22. Give a suitable example for each of the following:
(a) An application in which a process uses multiple threads that are organized in the dispatcher-workers model
(b) An application in which a process uses multiple threads that are organized in the team model
(c) An application in which a process uses multiple threads that are organized in the pipeline model

8.23. A file server works in the following manner:
(a) It accepts a client request for file access.
(b) It then tries to service the request using data in a cache that it maintains.
(c) If the request cannot be serviced from the cached data, it makes a request to the disk server for the data and sleeps until a reply is received. On receiving the reply, it caches the data received and services the client's request.

Assume that the hit ratio for the cache is 0.7. That is, 70% of all the requests are serviced using cached data and access to disk server is needed only for serving 30% of all requests. Also assume that, on a cache hit, the request service time is 20 msec and on a cache miss the request service time is 100 msec. How many requests per second can be serviced if the file server is implemented as follows?
(a) A single-threaded process
(b) A multithreaded process

Assume that threads switching time is negligible.

8.24. Differentiate between handoff scheduling and affinity scheduling of threads. In your opinion, which of the two is a more desirable feature for a threads package and why?

8.25. Write pseudocode for a threads-scheduling algorithm that provides the flexibility to vary quantum size dynamically and also supports handoff scheduling.

8.26. What are the main issues in handling signals in a multithreaded environment? Describe a method for handling each of these issues.

8.27. Discuss the relative advantages and disadvantages of implementing a threads package in user space and in the kernel.

8.28. The operating system of a computer uses processes as the basic unit of CPU utilization. That is, it does not support threads. Can threads facility be provided in this computer system without modifying the operating system kernel? If no, explain why. If yes, explain how.

BIBLIOGRAPHY

[Agrawal and Ezzat 1987] Agrawal, R., and Ezzat, A. K., "Location Independent Remote Execution in NEST," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 8 (1987).

[Alonso and Kyrimis 1988] Alonso, R., and Kyrimis, K., "A Process Migration Implementation for a UNIX System," In: *Proceedings of the Winter 1988 Usenix Conference*, Usenix Association, Berkeley, CA (February 1988).

[Anderson et al. 1991] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," In: *Proceedings of the 13th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 95–109 (1991).

[Artsy and Finkel 1989] Artsy, Y., and Finkel, R., "Designing a Process Migration Facility," *IEEE Computer*, Vol. 22, pp. 47–56 (1989).

[Black 1990] Black, D., "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, Vol. 23, pp. 35–43 (1990).

[Butterfield and Popek 1984] Butterfield, D. A., and Popek, G. J., "Network Tasking in the LOCUS Distributed UNIX System," In: *Proceedings of the Summer 1984 Usenix Conference*, Usenix Association, Berkeley, CA, pp. 62–71 (June 1984).

[Cheriton 1988] Cheriton, D. R., "The V Distributed System," *Communications of the ACM*, Vol. 31, No. 3, pp. 314–333 (1988).

[Coulouris et al. 1994] Coulouris, G. F., Dollimore, J., and Kindberg, T., *Distributed Systems Concepts and Design*, 2nd ed., Addison-Wesley, Reading, MA (1994).

[Douglis and Ousterhout 1987] Douglis, F., and Ousterhout, J., "Process Migration in the Sprite Operating System," In: *Proceedings of the 7th International Conference on Distributed Computing Systems*, IEEE, New York, NY, pp. 18–25 (September 1987).

[Douglis and Ousterhout 1991] Douglis, F., and Ousterhout, J., "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice and Experience*, Vol. 21, pp. 757–785 (1991).

[Draves et al. 1991] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W., "Using Continuations to Implement Thread Management and Communication in Operating Systems," In: *Proceedings of the 13th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 122–136 (1991).

[Ferrari and Sunderam 1995] Ferrari, A., and Sunderam, V. S., "TPVM: Distributed Concurrent Computing with Lightweight Processes," In: *Proceedings of the 4th International Symposium on High Performance Distributed Computing* (August 1995).

[Goscinski 1991] Goscinski, A.,"Distributed Operating Systems, The Logical Design," Addison-Wesley, Reading, MA (1991).

[Huang et al. 1995] Huang, C., Huang, Y., and McKinley, P. K., "A Thread-Based Interface for Collective Communication on ATM Networks," In: *Proceedings of the 15th International Conference on Distributed Computing Systems*, IEEE, New York, NY (May–June 1995).

[Hunter 1988] Hunter, C., "Process Cloning: A System for Duplicating UNIX Processes," In: *Proceedings of the Winter 1988 Usenix Conference*, Usenix Association, Berkeley, CA (February 1988).

[Jul 1989] Jul, E., "Migration of Light-Weight Processes in Emerald," *TCOS Newsletter*, Vol. 3, No. 1, pp. 20–23 (1989).

[Jul et al. 1988] Jul, E., Levy, H., Norman, H., and Andrew, B., "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–133 (1988).

[Kingsbury and Kline 1989] Kingsbury, B. A., and Kline, J. T., "Job and Process Recovery in a UNIX-Based Operating System," In: *Proceedings of the Winter 1989 Usenix Conference*, Usenix Association, Berkeley, CA, pp. 355–364 (1989).

[Litzkow 1987] Litzkow, M. J., "Remote UNIX—Turning Idle Workstations into Cycle Servers," In: *Proceedings of the Summer 1987 Usenix Conference*, Usenix Association, Berkeley, CA (June 1987).

[Lo 1989] Lo, V. M., "Process Migration for Communication Performance," *TCOS Newsletter*, Vol. 3, No. 1, pp. 28–30 (1989).

[Lockhart 1994] Lockhart, Jr., H. W., *OSF DCE: Guide to Developing Distributed Applications*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[Maguire and Smith 1988] Maguire, Jr., G. Q. and Smith, J. M., "Process Migration: Effects on Scientific Computation," *ACM-SIGPLAN Notices*, Vol. 23, No. 3, pp. 102–106 (1988).

[Mandelberg and Sunderam 1988] Mandelberg, K. I., and Sunderam, V. S., "Process Migration in UNIX Networks," In: *Proceedings of the Winter 1988 Usenix Conference*, Usenix Association, Berkeley, CA (February 1988).

[Marsh et al. 1991] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P., "First-Class User-Level Threads," In: *Proceedings of the 13th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 110–121 (1991).

[Mullender et al. 1990] Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staverene, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23, No. 5, pp. 44–53 (1990).

[Nutt 1991] Nutt, G. J., *Centralized and Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Popek and Walker 1985] Popek, G. J., and Walker, B. J., *The LOCUS Distributed System Architecture*, MIT Press, Cambridge, MA (1985).

[Powell and Miller 1983] Powell, M. L., and Miller, B. P., "Process Migration in DEMOS/MP," In: *Proceedings of the 9th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 110–119 (November 1983).

[Rosenberry et al. 1992] Rosenberry, W., Kenney, D., and Fisher, G., *OSF DISTRIBUTED COMPUTING ENVIRONMENT, Understanding DCE*, O'Reilly & Associates, Sebastopol, CA (1992).

[Schwan et al. 1991] Schwan, K., Zhou, H., and Gheith, A., "Real-Time Threads," *ACM-SIGOPS Operating Systems Review*, Vol. 25, No. 4, pp. 35–46 (1991).

[Sinha et al. 1991] Sinha, P. K., Park, K., Jia, X., Shimizu, K., and Maekawa, M., "Process Migration Mechanism in the Galaxy Distributed Operating System," In: *Proceedings of the 5th International Parallel Processing Symposium*, IEEE, New York, NY, pp. 611–618 (April 1991).

[Smith 1988] Smith, J. M., "A Survey of Process Migration Mechanisms," *ACM-SIGOPS Operating Systems Review*, Vol. 22, pp. 28–40 (July 1988).

[Stalling 1995] Stalling, W., *Operating Systems*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ (1995).

[Tanenbaum 1995] Tanenbaum, A. S., *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[Theimer et al. 1985] Theimer, M. M., Lantz K. A., and Cheriton, D. R., "Preemptable Remote Execution Facilities for the V System," In: *Proceedings of the 10th ACM Symposium on Operating System Principles*, Association for Computing Machinery, New York, NY, pp. 2–12 (December 1985).

[Thekkath and Eggers 1994] Thekkath, R., and Eggers, S. J., "Impact of Sharing-Based Thread Placement on Multithreaded Architectures," In: *Proceedings of the 21st International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, pp. 176–186 (1994).

[Walker and Mathews 1989] Walker, B. J., and Mathews, R. M., "Process Migration in AIX's Transparent Computing Facility (TCF)," *TCOS Newsletter*, Vol. 3, No. 1, pp. 5–7 (1989).

[Zayas 1987] Zayas, E. R., "Attacking the Process Migration Bottleneck," In: *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, pp. 13–22 (November 1987).

POINTERS TO BIBLIOGRAPHIES ON THE INTERNET

Bibliographies containing references on *Process Migration* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/migrate.html

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Distributed/dshell.html

Bibliography containing references on *Threads and Multithreading* can be found at:

ftp:ftp.cs.umanitoba.ca/pub/bibliographies/Os/threads.html