Synchronization and Scheduling in ALPS Objects

Prasad Vishnubhotla
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210
Phone: (614) 292-1553
vishnu@ohio-state.edu

Abstract

This paper describes the synchronization and scheduling mechanisms used in an object oriented concurrent programming language called ALPS. The main contributions of ALPS are the concept of a manager and hidden procedure arrays. An object can have a special high-priority process called a manager which implements the necessary synchronization and scheduling for the object. The concept of hidden procedure arrays allows an entry procedure to be exported as a single procedure but implemented as a procedure array. When multiple calls to the entry procedure arrive at the object, each call gets attached to a different element of the hidden procedure array. This simplifies the programming of concurrency within an object. The language mechanisms are illustrated using several examples.

1. Introduction

ALPS is an object oriented concurrent programming language that is being designed as part of an effort to develop a language for parallel and distributed systems. To facilitate efficient programming of shared memory multiprocessors, ALPS supports data sharing within an object. An object consists of a data part and a set of entry procedures which share the data part. To support distributed programming, ALPS provides both remote procedure calls and message passing. Calls to the entry procedures of an object are implemented as remote procedure calls. A user can further communicate with an executing remote procedure using message passing on point-to-point channels.

Two issues should be addressed in any object oriented design for parallel and distributed systems. The first is the concurrency of entry calls and the other is the concurrent execution of entry calls. Most object oriented systems implement synchronization and scheduling for entry calls using semaphores or conditional critical regions. When an entry call arrives the runtime system starts the execution of the called procedure right away. The procedure should be programmed to delay its own execution if the synchronization conditions are not satisfied. This approach has the drawback that the scheduling algorithm gets scattered across the various procedures of the object and is difficult to understand and modify. For this reason in ALPS we do not use semaphores or conditional critical regions for scheduling within an object. Instead we use a special process called a manager which intercepts entry calls and implements the synchronization and scheduling for the object.

When an entry procedure of an object is called, the procedure is not executed immediately but the call is directed to the manager. The call is delayed until it is accepted by the manager. The manager may delay accepting a call until certain synchronization conditions hold, or if the scheduling of the call requires further processing based on the invocation parameters. The manager can request the call and then delay it until it is mature for execution. When an entry call is mature for execution, the manager can execute the procedure concurrently with respect to itself. When an executing entry procedure terminates, the manager can recognize this and can update its synchronization state. Thus the manager provides a facility for pre- and post-processing of entry calls which can be used not only to implement scheduling but also to monitor the object.

If the scheduling policy allows concurrent execution of different entry calls then a separate server process is required in the object to handle each call. If all processes that are potentially required in an object are created statically at the time the object itself is created then the operating system may be burdened with too many processes of which only a few might be active at any given time. A common solution to this problem is to create a server process dynamically at the time an entry procedure is called. If this method is used in conjunction with the manager mechanism then we also need a facility to explicitly name the dynamically created processes so that the manager can implement the scheduling. However it is too cumbersome in a language framework to create process descriptors dynamically and to use process variables to store these process descriptors. Instead in ALPS we use the notion of hidden procedure arrays. Using this mechanism an entry procedure can be exported as a single procedure but can be implemented as an array of procedures. Each call to the entry procedure gets automatically attached to a distinct element of the procedure array which uniquely identifies the call and the process that is dynamically created to service the call.

Apart from implementing the initial scheduling for entry calls, the manager can be programmed to exchange messages with the executing processes and to implement other forms of scheduling. This can be done by programming each entry procedure in the object so that before entering a critical section it sends a request message to the manager and awaits a permission message from the manager. To minimize the synchronization overhead due to process switches between the manager and the other processes in the object, it is recommended that the whole object is stored in a single address space and all processes in the object are implemented as light weight processes. Also to ensure that the synchronization requests are delivered to the manager with minimum delay, the implementation should execute the manager at a higher priority compared to the other processes in the object.

The object/manager facility in ALPS is a generalization of the well-known synchronization abstractions monitor [1,2], serializer [3] and path expressions [4,5]. Monitor-like mutual exclusion can be implemented by programming the manager to execute each
request to completion before accepting another request. A variety of scheduling policies that require condition (queue) variables in monitors can be implemented in ALPS by programming the manager appropriately. An ALPS object is a resource protected by the manager. The manager can be programmed to allow multiple users to access the resource simultaneously - a facility sought in the design of the serializer mechanism. In ALPS it is possible to design objects such that all entry procedures of the object are sequential procedures and all scheduling is implemented separately in the procedures that are scheduled was first used in path expressions of scheduling policies that require condition (queue) variables in CSP. Though a task entry could create a new task dynamically and make it calls to the processes defined in the resource. Instead the exported procedures of the resource have to explicitly communicate with the manager-like process which implements the necessary scheduling. This affects the modularity and modifiability of scheduling because a change in the scheduling policy might require a change in several processes in the resource. Another important difference is that an SR resource that uses a process array to service simultaneous multiple requests must reserve one server process for each user of the resource and should make the whole process array visible to each user of the resource. In ALPS, it is possible for an object to export a single entry procedure but implement it as a procedure array.

The manager mechanism has close similarities to the mediator mechanism proposed recently [14] but there are significant differences. The similarities are in identifying the need for a flexible scheduling mechanism and in the use of primitives. The first difference is in the structure for the manager and mediator constructs. A manager in our model is a single CSP-like process whereas a mediator is a concurrent program using shared variables between its concurrent actions and conditional critical regions for synchronization between them. While it can be argued that the internal concurrency within a mediator could reduce contention, we believe that the manager mechanism is much cleaner and that contention can be reduced by programming the manager to do only minimal processing. The second difference is the use of hidden procedure arrays with managers, a facility not available with mediators. Instead a mediator uses an interface record which does not provide a proper level of abstraction that is compatible with object oriented programming. The interface records in mediators and the hidden procedure arrays in ALPS both serve the same purpose. However the hidden procedure arrays offer a better abstraction by exploiting the separation of definition and implementation of an object. In fact an implementation of a hidden procedure array has to maintain an interface record for each request that is attached to the array.

Several other concurrent programming language designs have been reported in the literature. For a survey of these languages see [15]. The rest of the paper is organized as follows. Section 2 presents the language notation and illustrates the ideas with several examples. Section 3 discusses some of the implementation issues. Section 4 concludes the paper.

2. The ALPS Language

2.1 Procedures

A procedure is defined by the following syntax:

\[
\text{proc name(name(params)) returns(results);} \\
\quad \text{body}
\]

Data values can be passed as parameters (params) when the procedure is called or data values can be obtained as results (results) when the procedure terminates. The procedure declared above can be called using the call command which is of the form

\[
\text{name(param-list, result-list)}
\]

where param-list and result-list are positionwise type compatible with params and results respectively. The call command terminates only when the called procedure terminates. Both params and results are optional (i.e., they can be empty).

2.1.1 Parallel Execution

Two commands are provided for parallel execution of procedures. The command

\[
\text{par P(...), Q(...) and R(...) end par}
\]

starts the executions of the procedures P, Q and R in parallel and terminates only when all the three procedures terminate. The command

\[
\text{par i = m to n do P(i) end par}
\]

executes P(m), P(m+1), ... , P(n) in parallel and terminates only when all the (n-m+1) executions of procedure P terminate.
2.1.2 Communication and Synchronization

Parallel executions of procedures can communicate either through shared memory or by using message passing on point-to-point communication channels. Message passing can also be used to synchronize access to shared data. Unlike Occam [16] which uses synchronous message passing on point-to-point channels, ALPS uses asynchronous message passing. A channel variable C may be declared as

```
var C: chan(T1, ..., Tn);
```

where \(T1, ..., Tn\) are types. An output on \(C\) can be sent by the command `send C(v1, ..., vn)` where \(v1, ..., vn\) are values of type \(T1, ..., Tn\) respectively. The message is buffered and the sending process continues with its execution. An input on \(C\) can be received by the command `receive C(x1, ..., xn)` where \(x1, ..., xn\) are variables of type \(T1, ..., Tn\) respectively. The receiving process is delayed until the message is available. Each channel should be used either for input or for output but not both. A receive command can be used in a guard of an alternative or repetitive command (to be discussed later). Channel variables may be used to compose arbitrary data structures (e.g., arrays of channels). Channels can be passed as procedure parameters and also as message values.

2.2 Objects

An object is described in two parts: definition and implementation. The definition part describes the user interface for the externally visible procedures - also called entry procedures - of the object, and constants and types that may be exported from the object. The user interface for a procedure consists of the name and formal parameter specification (it does not include the code for the procedure). The implementation part consists of an implementation for the entry procedures, any local procedures that may be used in implementing the entry procedures, an optional `manager process` that implements the synchronization and scheduling for all procedures in the object, a data part that is shared by all procedures of the object and the manager, and an optional initialization code which is implicitly executed when the object is created. The implementation part is transparent to the users of the object.

We use the following notation to describe the definition and implementation parts of an object:

```
object name defines
  constant and type declarations
  procedure declaration
  procedure declaration
  procedure declaration
end name

object name implements
  constant and type declarations
  shared data declarations
  procedure body
  procedure body
  procedure body
  manager process
begin
  initialization code
end name;
```

An object can be imported into a program or another object by a `use` clause. An entry procedure \(P\) defined in an object \(X\) can be called from outside the object using the statement \(X.P(...).\) A user can call entry procedures belonging to the same or different objects in parallel using the `par` statement discussed earlier. A user can also communicate with an executing entry procedure using messages.

In the present version of ALPS an object has a single instance. Object types can be provided in a future version either in the style of Clu clusters [17] or in the style of Ada package types [10].

2.3 Managers

The manager process has a declaration part followed by a code body. The purpose of the manager is to intercept calls to the entry procedures of the object and to implement the necessary synchronization and scheduling by delaying the execution of the entry procedures until they are mature for execution. The manager process should be executed at a high priority compared to the other processes in the object so that the manager is more receptive to entry calls. When an object is created, its initialization code is first executed and then its manager process is implicitly created and started. The manager is never specified in the object's definition and is optional in the object's implementation. If the manager is omitted then each time an entry procedure is called a process is created implicitly and is made to execute the procedure. In that case either the object should not be shared between parallel processes or the entry procedures should implement the necessary synchronization themselves using message passing primitives. Thus it is possible to design an object which does not use a manager but encapsulates a set of procedures and some shared data.

Suppose \(X\) is an object which has a manager process and let \(P\) be an entry procedure of \(X\). When an outside process invokes \(P\), \(P\) is not actually executed until the invocation is accepted by \(X\)'s manager. The manager shows its acceptance by executing the primitive `accept P(...)`. If the manager reaches an accept statement first, it awaits a user invocation, and if a user invocation arrives first, it is delayed until the manager executes a corresponding `accept` statement. This is somewhat similar to the rendezvous mechanism in Ada [10] except that the rendezvous that takes place here is between the user process and the manager process that is hidden inside the object rather than between a user process and an explicitly named process as happens in Ada. Moreover, the purpose of the `accept` primitive is simply to inform the manager of the entry call and to supply some of the invocation parameters to the manager which it can use for scheduling purposes. The actual service is implemented in the procedure \(P\) which is textually separated from the manager. Subsequently the manager is expected to execute the primitive `start P(...)` which starts the execution of the procedure \(P\) asynchronously with respect to the manager. When the manager executes a `start primitive` it also supplies all the invocation parameters that it received from the caller. The asynchronous nature of the `start primitive` allows the manager to accept other remote calls while the execution of \(P\) is in progress. This avoids the problem of nested (monitor) calls [18]. For example, two objects \(X\) and \(Y\) can be programmed without deadlock such that an entry procedure \(P\) in \(X\) calls a procedure \(Q\) in \(Y\) which in turn calls another entry \(R\) in \(X\). Deadlock can be avoided because \(X\)'s manager can be programmed such that after starting the execution of \(P\) it can be ready to accept calls to \(R\). Note that DP, Ada and SR suffer from the nested calls problem.

When a procedure \(P\) that is started by the manager is ready to terminate, it waits until the manager recognizes its readiness for termination and also grants permission for it to terminate. We require the manager's endorsement of \(P\)'s termination so that the manager can update its synchronization state and also monitor the results being returned by \(P\). The manager can use the primitive `await P(...)` to recognize \(P\)'s readiness for termination and to receive some of the results from \(P\). Subsequently the manager should execute the
The invocation parameters are not given to the manager but are sent which call a common local procedure R, then the manager can to supply to the caller of P the portion of the results that it received from P. If the manager does not want to influence the results being returned by P to the caller, then the parameters to the await and finish primitives can be empty but the use of await and finish primitives is required to complete the protocol necessary for P's termination. When the manager executes a finish P(...), P returns to the caller the portion of the results that it still has and then both the finish P(...) and P terminate together so that the manager can accept another pending call to P. Notice that when the manager executes a finish P(...), it never blocks because the caller of P is simply waiting for the results. For some applications the manager has to execute a called procedure in exclusion. To simplify coding in these situations, we provide a packaged construct "execute P(params, results)" which is equivalent to "start P(params); await P(results); finish P(results)".

The manager has to specify which procedures it wishes to intercept. This is done by using an intercept clause in the manager which is of the form "intercepts P, Q, R". This means that all calls to the procedures P, Q and R are directed to the manager. Calls to a procedure that is not listed in the intercepts clause are not directed to the manager but the procedure execution is started implicitly. This provides the flexibility to define entry procedures that are not intercepted by the manager (e.g. a procedure that returns the object's status) and to intercept even local procedures when they are invoked from other entry/local procedures. Intercepting local procedures is somewhat similar to naming local procedures in path expressions [4]. If P and Q are two entry procedures of the object which call a common local procedure R, then the manager can control the execution of P and Q even after starting them by intercepting the calls to R. This allows programming the the object so that the manager is solely responsible for the scheduling.

When a manager intercepts a call to an entry procedure, by default the invocation parameters are not given to the manager but are sent by the caller directly to the called procedure after it is started by the manager. In ALPS, the manager can receive some or all of the invocation parameters itself and use them for scheduling purposes and later send them to the called procedure at the time it is started. If the manager wishes to exert such control on a procedure P, its intercepts clause should have the form "intercepts P(params)"; where params is an initial subsequence of the parameter list of P as it appears in the object's definition. We allow the manager to receive an initial subsequence of the parameters because it is wasteful to require the manager to receive all the parameters. The manager can also intercept the results returned by an entry procedure. The full form of the intercepts clause is described in Section 2.6.

2.4 Nondeterministic Selection

We provide two nondeterministic control structures select and loop which are similar to the alternative and repetitive constructs of CSP [7]. Just as CSP uses input commands in guards, we use the accept, await and receive primitives in the guards of the selection and loop statements. Use of the receive primitive in a guard is the same as in CSP. Use of the accept primitive in the guards allows the manager to select entry calls nondeterministically. Similarly use of the await primitive in the guards allows the manager to nondeterministically select among the entry procedures that are ready to terminate. A guard using accept P(...) can be selected only if there is a pending call for P. A guard using await P(...) can be selected only if P is ready to terminate. Call guards [19] are not provided in our language.

In ALPS we allow the boolean condition appearing in a guard to depend not only on the current state of the object (as in CSP) but also on the values (parameters, results or message values) received by an accept, await or receive command. Such conditions may be called acceptance conditions. An acceptance condition succeeds if it evaluates to true when evaluated after the corresponding accept, await or receive command is executed, otherwise it fails. To test the success of an acceptance condition, it is necessary for the implementation to first receive the input values in temporary variables and then to evaluate the acceptance condition by substituting the temporary variables for the actual variables. This concept has been used in SR [12]. The need for such a mechanism in Ada has been pointed out in [20].

For some applications it is important to prioritize the selection of guards. Such priorities cannot always be specified as compile-time constants, so it is necessary to allow specification of run-time evaluable priorities with the guarded alternatives. The language SR provides such a facility. We have incorporated this facility in ALPS by allowing a priority clause of the form "pri E" where E is an integer valued expression to appear in a guard. The meaning of this is that among the guarded commands that are eligible for selection, one with the smallest pri value will be selected by the implementation.

The select and loop statements in ALPS have the following structure:

select G1 => S1 or G2 => S2 or ... or Gn => Sn end select

loop G1 => S1 or G2 => S2 or ... or Gn => Sn end loop

Each Gi is a guard and each Si is a statement sequence. We sometimes use the notation "(i=1..n) Gi => Si" to mean "G1 => S1 or ... or Gn => Sn". If B is a boolean expression, P is an entry procedure and C is a channel then a guard can take one of the forms: "when B", "accept P(...)", "await P(...)", "receive C(...)", "accept P(...) when B", "await P(...) when B" or "receive C(...) when B". In addition a priority specification of the form "pri E" can appear at the end of the guard where E is an integer expression which can possibly use values received by an accept, await or receive appearing in the guard. The semantics of the select and loop statements are similar to those in CSP except that we do not use any form of distributed termination.

2.4.1 Example : Bounded Buffer

A producer and a consumer exchange messages via a bounded buffer object which defines two entry procedures Deposit and Remove. Since Deposit and Resume can be called concurrently by the producer and the consumer respectively, their executions need to be synchronized. This synchronization can be implemented by a manager in the buffer object as follows:

object Buffer defines
proc Deposit(Message);
proc Remove returns (Message);
end Buffer
2.5 Hidden Procedure Arrays

Since an object can be shared across different users, there can be multiple calls to an entry procedure $P$. It is necessary to identify these multiple requests separately so that the manager can schedule them appropriately. One way to program this is to declare $P$ as a procedure array with one procedure in the array reserved for each user process. In this approach each user is expected to call only that procedure in the procedure array which is reserved for that user. However this method makes the array structure of $P$ visible to the users and assumes some discipline on the part of the users so that a user does not call a procedure reserved for some other user. Moreover in parallel processing applications each user program can have several component processes competing for the shared resources in the system and it is not practical for each resource to reserve a entry procedure for each possible user process in the system.

We use procedure arrays to implement multiple calls to $P$ but take the view that the user processes should not be aware of the array structure of $P$ - rather they should get the impression that $P$ is a single procedure. To facilitate this we allow $P$ to be used as a hidden procedure array. A procedure array is hidden simply by declaring it as a single procedure $P$ in the object’s definition and as a procedure array $P[i..1..N]$ in the object’s implementation. The number $N$ of procedures used in the array should be a constant but can be independent of the number of user processes in the system. Whenever a user invocation for $P$ arrives, it automatically gets attached to one of the procedures $P[i]$ (1<=i<=N) selected arbitrarily by the implementation. However $P[i]$ is executed only under the manager’s control. The manager can recognize the request attached to $P[i]$ by executing an accept $P[i](...)$. Another request is not attached to $P[i]$ until the currently attached request is processed by $P[i]$, i.e., until the manager executes a finish $P[i]$. If there are more requests than can be accommodated in the procedure array $P$, the remaining requests continue to wait until one of the $P[i]$’s is executed to completion. If $P[i]$ does not have a request attached and an accept $P[i]$ is executed, it is delayed until a request is attached to $P[i]$.

2.5.1 Example : Readers- Writers Problem

A set of readers and writers access a database. A reader’s request gets delayed only if a writer is updating the database or there are too many readers already using the database. The constant $ReadMax$ denotes the maximum number of readers that can access the database simultaneously. A writer’s request gets delayed only if a reader or writer is currently using the database. No reader or writer should be delayed indefinitely. Otherwise no priorities are assumed between readers and writers.

```
object Database implements { the database is declared here }
proc Read(Key: KeyType; Data: DataType); // read data from item Key and return it to caller
begin read data from item Key end Read;
proc Write(KeyType; Data:DataType); // begin write value of Data into item Key end Write;
manager Intercepts Read, Write;
var Readcount: int; WriterLast: bool;
begin
  Readcount := 0; WriterLast := false;
  loop
    (i:1..ReadMax) accept Read[i] when Readcount < ReadMax
    execute Read[i];
  end loop;
end Database;

object Database implements { the database is declared here }
proc Read(Key: KeyType) returns (DataType);
proc Write(KeyType, DataType);
end Database;
```

The use of the intercepts clause in the manager indicates that calls to Deposit and Remove should be directed to the manager. The loop statement is used so that the manager can accept calls either to Deposit or to Remove nondeterministically and repetitively. However a call to Deposit is accepted only if the buffer is not full and a call to Remove is accepted only if the buffer is not empty. The variable Count - which is local to the manager - is used to maintain the state of the buffer. Though Inptr and Outptr can be used in the implementation of the procedures do not affect the manager. When the manager accepts a call to Deposit or Remove, it starts the procedure execution but waits until the procedure terminates before accepting another call. This first example shows the basic synchronization possible in a manager but does not illustrate parallel execution within an object.
Since Read appears as a single procedure in the object's definition, the users are unaware of its array structure in the implementation and can read the database by executing a Database.Read(Key, Data). The notation #Read and #Write is used to refer to the number of pending calls to Read and Write respectively (#Read includes any read request that may have been attached to some Read[i] but not yet accepted and also any read request that may be waiting to be attached to a process). This facility is similar to the COUNT attribute in Ada and the # notation in SR. The above program is adapted from a similar one in [12].

A read request is accepted either if there are no pending write requests or if a writer has just used the database (the latter disjunction is necessary to avoid starvation). After accepting a read request, the corresponding Read[i] procedure is executed asynchronously with respect to the manager. In this manner, upto ReadMax readers can be allowed to access the database simultaneously. The variable ReadCount maintains the number of active readers currently using the database. The manager uses the await primitive to recognize if any of the Read procedures are ready for termination. The actual termination of a Read procedure, say Read[i], occurs only when the manager executes a finish Read[i]. A write request is accepted by the manager only if there are no active readers using the database and either there are no pending read requests or a writer is due its turn even if there are pending read requests.

Two points about this program need further explanation. Notice that each time a reader starts or terminates, the major action taken by the manager is only to update the variable ReadCount, but it does not maintain any information specific to each reader. It might appear that using Read as a procedure array is unnecessary at least for this example. It is possible to tune the language to allow "accept Read" and "start Read" instead of "accept Read[i]" and "start Read[i]" and to interpret the "start Read" as "start the execution of the call that is previously accepted". We chose not to consider such special cases as it would only complicate the semantics. It is much simpler to treat each reader process as calling a separate Read procedure. The second point is that this example does not really require to use await Read[i] and finish Read[i] separately. A combined call would have been more useful. Again we chose not to introduce a special primitive for this purpose. The next example shows the full power of the await and finish primitives.

2.6 Intercepts Parameters and Results

The Intercepts clause can take the form "Intercepts P(params; results)" where params is an initial subsequence of the invocation parameter list of P as described in the object's definition and results is an initial subsequence of the result list of P as described in the object's definition. Either or both of params and results can be empty. The use of params in the Intercepts clause means that when the manager intercepts a call to P using an accept primitive, the manager should receive the first few invocation parameters as specified by params and that the manager should supply these invocation parameters to P when it is started using a start primitive. The remaining invocation parameters are supplied by the caller directly to P after it is started by the manager. We allow the manager to receive an initial subsequence of the parameters because it is wasteful to require the manager to receive all the parameters. In fact in many cases the manager may not require any invocation parameters at all. The use of results in the Intercepts clause means that when the manager recognizes the termination of P using an await primitive the manager should receive the first few results from P as specified by results and that the manager should subsequently supply these results to the caller of P using a finish primitive. The remaining results are supplied by P directly to the caller.

2.7 Combining Invocation Requests

A manager need not start a procedure execution for every entry call that it accepts. For some applications it is more economical if the manager can combine some of the pending requests and synthesize a single request - perhaps to an altogether different procedure that is transparent to the users - so that a single procedure execution would serve the needs of several users. This idea is a software adaptation of the memory combining that is used in the NYU Ultracomputer [21]. Combining can be programmed using managers in the following way. After accepting a call to an entry procedure P, the manager can execute a finish P(...) without even starting the execution of P. In this case the manager is responsible to receive all invocation parameters in the accept primitive, to generate all the results that the caller expects and to supply them as parameters to the finish primitive. A useful application of this is to combine multiple read requests to the same record in a database as illustrated by the following example.

2.7.1 Example: Dictionary Database

A dictionary database is implemented on a multiprocessor by servicing multiple queries to the dictionary simultaneously whenever possible. Each time a request arrives asking for the meaning of a word, a new process is created which then searches the dictionary for that particular word and returns its meaning. The dictionary is defined as an object that hides the database and exports an entry procedure Search that performs the dictionary search. The object's implementation, however, implements Search as a procedure array so that multiple requests to Search can be recognized. Since it is wasteful to execute multiple Search processes that search for the meaning of the same word, the object's manager can be programmed to recognize such requests and to combine them. The following program is written in a pseudo-code in order to avoid the details.

```plaintext
object Dictionary defines
  proc Search(String) returns (String);
  end Dictionary;

object Dictionary Implements
  [ the dictionary database is declared here ]
  proc Search(1..SearchMax)(Word: String) returns (String);
  begin search the dictionary for Word & return its meaning and Search;
  end Search;

manager
  Intercepts Search(String; String);
  begin loop
    (i:1..SearchMax) accept Search[i](Word: String) =>
      if "Word is already being searched on behalf of another search request" then
        "record that Word is now being searched on behalf of Search[i]"
      else
        start Search[i](Word);
      end if;
    end if;
  end loop;
end

Dictionary;
```

261
Note the use of the Intercepts clause in the manager. This means that the manager wishes to intercept the parameter supplied by the caller as well as the result returned by the Search procedure. The manager receives the intercepted parameter in the accept command and receives the intercepted result in the await command.

### 2.8 Hidden Parameters and Results

For some applications the manager may need to supply a few additional parameters to the entry procedures at the time of executing the start primitive. By additional parameters we mean values which the manager desires to supply to the entry procedure over and above any invocation parameters which it received from the caller. Though such additional information could be supplied as messages after starting the procedure execution, it is more elegant if additional parameters can be supplied at the time of starting the procedure execution. Similarly the manager may need to receive some additional results from the process at the time of executing a await primitive - results which are not intended to be delivered to the original caller. In both these cases the additional parameters/results should be included in the object's implementation. However since the user is unaware of these additional parameters/results, they should not be included in the object's definition.

We allow an object's implementation to specify additional parameters/results over and above those specified in the object's definition. Since these additional parameters are hidden from the users of the object, we refer to them as hidden parameters and results respectively. The hidden parameters/results should appear after the regular parameters in the formal parameter/result lists in the object's implementation.

#### 2.8.1 Example: A Printer Spooler

Consider a printer spooler object which uses a pool of printers and schedules print requests onto these printers. The spooler object defines a procedure Print which takes the file to be printed as its only parameter. The object's implementation declares Print as an array of procedures so that multiple print requests can be handled simultaneously. After accepting a print request, the object's manager assigns a free printer and supplies the printer number along with the file descriptor to the corresponding Print procedure. However since the object defines the Print procedure as an array of procedures, the manager accepts a call to Print[i] by allocating a free printer and supplying the printer number and file descriptor to the corresponding Print[i] procedure. Notice that the Print procedure also returns the printer number as a hidden result back to the manager. This eliminates a lot of bookkeeping for the manager to remember which printer has been allocated to which procedure.

#### 2.8.2 Example: Parallel Bounded Buffer

We present a parallel bounded buffer which is a variation of the bounded buffer given in Section 2.4.1 but is more useful in parallel processing. Several producers and consumers are allowed to call the Deposit and Remove procedures of a shared buffer that can hold a finite number of potentially long messages. A consumer can be given a message deposited by any producer. In the implementation of the Buffer object, Deposit and Remove are realized as procedure arrays so that producers and consumers can be serviced in parallel. When the manager accepts a call to Deposit[i], it allocates a free buffer slot and supplies its index as a hidden parameter to Deposit[i]. Similarly when the manager accepts a call to Remove[i], it supplies the index of a message available in the buffer as a hidden parameter to Remove[i]. Once the manager starts a Deposit[i] or Remove[i] in this manner, it can access the buffer without further synchronization. Instead of programming the manager to keep track of buffer slot availability, which is a hidden result back to the manager, the procedures Deposit and Remove are programmed to return the slot index as a hidden result to the manager.

```plaintext
object Buffer defines
  proc Deposit(Message); proc Remove returns (Message); end Buffer

object Buffer implements
  var Buf: array 0..N-1 of Message; (N is a constant)
  proc Deposit[i]::[ProducerMax]::M::Message::Place: int returns (int);
    begin Buf[Place] := M;
      return (Place);
    end Deposit;
    proc Remove[i]::[ConsumerMax]::Place: int returns (Message, int);
      var M::Message;
      begin M := Buf[Place];
        return (M, Place);
      end Remove;
```
The manager maintains two integer arrays Free and Full. Free is a list of indices of Buf whose slots are empty (i.e., in which a new message can be deposited) and Full is a list of indices of Buf whose slots are full (i.e., from which a message can be removed). Initially Free contains indices of all buffer slots. Each time the manager accepts a call to some Deposit[i], it removes the next buffer slot index from Free and supplies it to Deposit[i] as a hidden parameter. The manager does not keep track of which slot index it gave to which procedure but expects each procedure to return the slot index it was given as a hidden result when it terminates (this simplifies the manager's code). While finishing the execution of Deposit[i], the manager stores the returned slot index in Full. Each time the manager accepts a call to some Remove[i], it removes the next buffer slot index from Full and supplies it to Remove[i] as a hidden parameter. While finishing the execution of Remove[i], the returned slot index is stored back in Free. Notice that in the implementation of Buffer, the procedures Deposit and Remove specify a hidden formal parameter Place of type int as well as a hidden result of type int. The corresponding actual parameters are supplied by the manager when it executes the start and await statements respectively.

3. Implementation Issues

Throughout this paper we have been assuming that a remote procedure is executed by a dynamically created process. Unfortunately in many operating systems dynamic process creation is expensive. One solution to this problem is to create a pool of processes at the time the object is created and to assign a process from this pool to execute a remote procedure. The hidden procedure array mechanism provides an ideal abstraction for such an implementation. For example if a procedure P is declared as a procedure array P[1..N] in an object's implementation then between the compiler and the run-time system a pool of N processes could be created at the time the object is created. In this case the mapping between the procedures and processes is one-to-one and a remote call that is attached to the procedure P[i] is executed by the corresponding process. It is also possible to implement the hidden procedure array P by preallocating a pool of M processes where M is much less than N and to assign a process to a remote call at the time it is started rather than when the call arrives. This alternative helps to minimize the number of processes that are required to implement a resource which may be attractive for resources in high demand where the average number of waiting requests (queue length) is significant. The programmer may be allowed to choose between these alternative implementations using compiler switches (in the latter case a value for M may have to be specified).

In recent years it is coming to be widely recognized that operating systems should support light weight processes in addition to conventional processes. A light weight process exists only in the context of a parent process and shares its address space but otherwise has very little contextual information of its own. Due to this, both the dynamic creation of light weight processes as well as switching between light weight processes can be implemented very efficiently. For example the Mach operating system [22] supports two kinds of processes: tasks which are like conventional processes and threads which are the light weight processes that share the address space of the parent task. In such environments an object can be implemented as a single address space owned by the manager process which may be implemented as a conventional process whereas calls to the entry procedures of the object can be executed by dynamically created light weight processes.

Another implementation issue is the polling of remote calls particularly when hidden procedure arrays are used. Note that a hidden procedure array P[1..N] may have only a small number of requests attached to it on the average and that it is wasteful to implement a guarded command of the form "((i:1..N) accept P[i] or (i:1..ConsumerMax) await Deposit[i](Full[FullIn]) => finish Deposit[i]; FullIn := (FullIn + 1) mod N; Min := Min + 1; or ((i:ProducerMax) await Remove[i](Full[FullOut]) => finish Remove[i]; FreeOut := (FreeOut + 1) mod N; Max := Max + 1; or ((i:ConsumerMax) finish Remove[i]; or (i:ProducerMax) accept Deposit[i] when Max > 0 => start Deposit[i](Free[FreeIn]); FreeIn := (FreeIn + 1) mod N; Max := Max - 1; or ((i:ConsumerMax) accept Remove[i] when Min > 0 => start Remove[i](Full[FullOut]); FullOut := (FullOut + 1) mod N; Min := Min - 1; or ((i:ConsumerMax) accept Remove[i] when Min > 0 => start Remove[i](Full[FullOut]); FullOut := (FullOut + 1) mod N; Min := Min - 1; or ((i:ProducerMax) await Deposit[i](Full[FullIn]) => finish Deposit[i]; FullIn := (FullIn + 1) mod N; Min := Min + 1; or ((i:ConsumerMax) await Remove[i](Full[FullOut]) => finish Remove[i]; FullOut := (FullOut + 1) mod N; Max := Max - 1;) end loop; end manager; end Buffer;

4. Conclusions

We have described a synchronization and scheduling mechanism called manager for object oriented concurrent programming. It is the basis for the ALPS language which is a part of an effort to develop an object oriented programming environment for parallel and distributed systems. The initial ideas of this object model were documented in [23]. Since then we have designed an operating system kernel that forms the run-time support for the ALPS language and which can be used directly from other languages like C. The ALPS kernel is currently being implemented in C on a 16-node transputer network [24]. In the future we plan to implement it on Sun workstations, Encore/Multimax, Intel iMPC and IBM Butterfly.

The version of ALPS presented here uses strong typing and is based on a Pascal-like notation. A compiler for this version in the initial stages of development. We intend to use ALPS as a design
language at least until a compiler for the complete language becomes available. A version of the ALPS language based on C, which we call ALPS/C, appears to arouse enough enthusiasm and might be used to design an operating system as a part of the ALPS project.

References


[23] P.R. Vishnubhotla, An Object Model of Parallel and Distributed Programming, OSU-CISRC-86TR1PV, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio, August 1986.