# PROCESS MIGRATION IN UNIX ENVIRONMENTS

Chin Lu and J. W. S. Liu

1304 W. Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Contact Person: Chin Lu
Address: same as above
Phone: (217)-333-2518
Net Address: chin@cs.uiucdcs.edu

(NASA-CR-182951) PROCESS MIGRATION IN UNIX ENVIRONMENTS (Illinois Univ.) 23 p CSCL 09B

N88-29311

Unclas
G3/61 0146718

This work was partially supported by the NASA Contract No. NAG 1-613 and a grant from AT & T Information Systems.

# PROCESS MIGRATION IN UNIX ENVIRONMENTS

## I. Introduction

A process is a program in execution. Process migration is the relocation of a process from the host (*the source host*) on which it is executing to another host (*the destination host*) [1-7]. DEMOS/MP [1], V System [2], and ACCENT [3] are well-known systems that support process migration.

We are concerned with how to migrate independent processes (or process groups) on a network of identical UNIX hosts. By an independent process, we mean a background user process that has no direct access to physical devices and interacts with its environment only through system calls. Examples of independent processes are simulation processes, text formating processes, and numerical computation processes. These processes often need long execution time. By providing the capability to migrate them, load balancing can be carried out dynamically. More importantly, this capability allows the processes to survive predictable host failures and down times.

Typically, a set of system server processes resides on every host and provides basic services to user processes on that host. We call these system processes *local-server processes.* Examples of local-server processes include terminal handlers, file servers, daemon processes, etc.. These local-server processes never migrate. When a host fails, local-server processes die. Hereafter, the term *process* refers to a user process unless stated otherwise. We refer to the local-server process that handles the migration of processes on each host as the *migration handler* on that host.

† UNIX is a trade mark of AT & T.

Our attention is confined to the case where all hosts have identical hardware, run identical software and provide identical services. Moreover, reliable file transfer between hosts is supported. Hence a process can run on any host and can be migrated freely from one host to another in principle. However, this does not mean that a process can be migrated correctly in the middle of its execution. In general, a migrated process may execute incorrectly or produce an unacceptable result on its destination host. For example, system clocks on different hosts are not synchronized; process identifiers are not system–wide unique in most cases; a file descriptor usually has some fields that have location dependent values. A process that accesses host dependent variables may produce an incorrect result after it is migrated. For this reason, simple migration packages such as the ones described in [8,9] do not handle the migration of host variable dependent processes correctly.

This paper describes the structure of migration handlers that are capable of migrating independent processes between UNIX hosts correctly. In particular, the migration handlers on the source and destination hosts will migrate a process and will carry out the necessary actions to ensure its correct execution on the destination host whenever it is possible to do so. By recording relevant past behaviors of processes, the migration handlers maintain with enough information to differentiate processes that can be migrated correctly from processes that cannot. By correct migration of a process, we mean that the execution of the process on its destination host is correct according to the criteria defined in Section II. The design of the migration handlers is based on a formal model of interactions between processes and their environments. This model is described in Section III. The structure of the migration handlers and the supporting data structures are described in Section IV. Section V is the conclusion.

## II. Correctness Criteria

From the stand point of any process, a system consists of two entities: the process itself and the environment of the process. The term *environment* loosely refers to the rest of the system with respect to that process. There are at least two aspects of correct process migration. One aspect is concerned with maintaining the state of the migrating process. Another aspect is concerned with the interaction between the process and its environment. We assume here that every process executes correctly on every host if it is not migrated.

It is obvious that in order for a migrated process to execute correctly, the state of the process at resumption time must be the same as the state of the process at suspension time. The state information includes the process's stack, text segment, data segment, register contents, instruction pointer, etc.. We call this criterion of correctness the *state consistency criterion*[10]. Typically, this criterion is satisfied by having the migration handler on the source host suspend the process, save its state information, and transfer this information to the destination host. The migration handler on the destination host restores the state of the process based on this information[1–3,8,9].

It is also obvious that the migration of any process should not leave its source host in an inconsistent state. For instance, the files that are opened by the process must be closed; the resources held by the process, including locks for exclusive access of shared resources and temporary storage, etc., must be released. In addition, all consistency constraints of its environment must be satisfied during the migration of the process. These constraints are both application– and implementation–dependent. Typically, different constraints are enforced by different components of the system regardless. We need not concern with this problem here.

A process running on a UNIX host interacts with its environment through system calls. Let $\Sigma$ be a system call that returns a host-dependent result. Two invocations of $\Sigma$ by a process, one on its source host and one on its destination host, may return results that are inconsistent in some sense. This inconsistency may cause the process to execute incorrectly after it is migrated. For example, suppose that a process $P$ on host $A$ is suspended immediately after it invokes the system clock, and the value returned by the system clock is $t_1$. Suppose that after it is resumed on host $B$, it invokes the system clock again that returns the value $t_2'$. When the clocks on different hosts are not synchronized, $t_2'$ is not necessarily larger than $t_1$. In this case, $P$ sees an inconsistent view of its environment against monotonic constraint of time as a result of its migration. This example illustrates the need for the second criterion of correct migration: A migrating process sees a consistent view of its environment at all times. In other words, the properties of its environment that are visible to a process must satisfy a given set of consistency constraints. We call this *property consistency criterion* [10]. We say that a process is migrated correctly if both the state consistency and property consistency criteria are satisfied.

Unlike the state consistency criterion, property consistency constraints are mainly semantic constraints on certain system properties and hence are typically system dependent. The monotonic property of system clock and the uniqueness of process identifiers are two simple examples. In distributed systems where several versions of a file may exist on different hosts, requiring a migrated process to use the same version (not necessarily the same copy) is another example. In past studies on process migration, processes that access host dependent variables are simply not migrated [1–3,8,9]. Methods to identify such processes are typically not given in these studies. We note that this restriction on the types of processes is often unnecessary. For example, the process $P$ in the previous example cannot be migrated under this restriction. However, $P$ can be migrated correctly either after it no longer needs to access the system clock or before it makes any access to the system clock. Alternatively, if the migration handlers on its source and destination hosts jointly maintain a consistent view of its environment for $P$ by compensating for the difference in system clocks on the two hosts, $P$ can be migrated correctly at any time. A function of the migration handlers is to detect any inconsistency between the results returned by system calls made by a process before and after its migration. This inconsistency is resolved by the migration handler on the destination host whenever it is possible to do so. In other words, the migration handler maintains a host independent virtual environment on the behalf of each migrated process.

## III. Formal Model of Interactions

The design of our migration handlers uses the formal model in [10] to describe the interactions between a process and its environment: The former is an object $P$. The latter consists of a set of objects referred to as *environment objects.* An object consists of a name, a representation of data structures stored in the object, and a set of operations on the data structures in the object. The process interacts with its environment by invoking the operations of the environment objects. This model is similar to the ones used in software specification and verification [11], and in operating system design [12].

An object is modeled as a state machine $M$ [13] represented by a 4-tuple: $M = < \mathbf{S}, S_{0,} O, T >$, where $\mathbf{S}$ is the state space of $M$. $S_0$ denotes the initial state; $\mathbf{O}$ is a set of transition operations; and $T$ is the transition function. Each operation in $\mathbf{O}$, when executed, causes the object to change state. Its effect is expressed by the transition function $T$: $\mathbf{O} \times \mathbf{S} \rightarrow \mathbf{S}$. More specifically, there are two basic types of functions — *$V$-functions* and *$O$-functions.* Each *primitive* $V$-function returns the value of a state variable. The values of all primitive $V$-functions at a particular moment specify the state of the object at that moment. A

$V$–function that returns a value computed from the values of primitive $V$–functions is called a *derived* $V$–function. An $O$–function performs an operation that changes the state of the object. The state transition called effects are described by assertions relating new values of primitive $V$–functions to their prior values.

The interface between an object and its outside world is the set of external operations consisting of all its derived $V$–functions and $O$–functions. A process interacts with its environment by invoking the external operations of the environment objects. In particular, each system call is either a derived $V$–function to examine the state of some environment object, or an $O$–function to change the state of an environment object, or an $O$–function followed by a derived $V$–function of an environment object. We refer to an $O$–function followed immediately by a derived $V$–function as an *$OV$–function.*

Let $S_P(t)$ be the state of the process $P$ at time $t$. (For the sake of convenience, all time references are in terms of a clock external to the system.) $S_P(s)$ and $S_P(r)$ are the states of $P$ at suspension time $s$ and resumption time $r$, respectively. The state consistency criterion means that

$$S_P(s) = S_P(r) \tag{1}$$

Let $E_X(P)$ denotes the set of environment objects with which process $P$ interacts on host $X$. Suppose that there are $n$ environment objects in $E_X(P)$, each of them is denoted by $E_i$, $i = 1, 2, .., n$. Then $E_X(P) = \{ E_{1,} E_{2,} .., E_n \}$. The state space of $E_X(P)$, denoted by $\mathbf{S}_X(P)$, is the product of the state spaces of all environment objects in $E_X(P)$. Process $P$'s *view* $V_X(P, t)$ of its environment $E_X(P)$ on host $X$ at time $t$ is defined as the properties that $P$ has observed about $E_X(P)$ in the time interval starting from the creation time $t_0$ of $P$ to the time $t$. In other words, $V_X(P ,t)$ is the knowledge that $P$ has about its environment $E_X(P)$ up to time $t$ [14].

Since the set of derived $V$–functions of all environment objects defines exactly what a process can observe about its environment, $V_X(P, t)$ is the set of values returned by all the derived $V$–functions of $E_X(P)$ invoked by $P$ in the time interval $(t_0, t)$.

We express property consistency constraints in terms of assertions relating the values of derived $V$–functions before and after the migration. Let $t_1$ and $t_2$ be two time instances with $t_2 > t_1$. When a process migrates from host X to host Y at a time between $t_1$ and $t_2$, its view $V_Y(P, t_2)$ of its environment $E_Y(P)$ on host $Y$ at $t_2$ is said to be consistent with its view $V_X(P, t_1)$ of $E_X(P)$ on host X at $t_1$ if all the property consistency constraints specified for the system are satisfied. We say that $P$ has seen a *consistent view* of its environment at $t_2$ with respect to $V_X(P, t_1)$, or simply the two views are consistent; we use $V_X(P, t_1) => V_Y(P, t_2)$ to denote this fact. In other words, the property consistency constraints serve as invariant conditions that are true for both $V_X(P, t_1)$ and $V_Y(P, t_2)$. Since a process runs correctly on a single host, $V_X(P, t_1) => V_X(P, t_2)$ is always true. The consistency of different views has the transitive property. That is, for different time instances $t_1 < t_2 < t_3$ , if $V_X(P, t_1) => V_Y(P, t_2)$ and $V_Y(P, t_2) => V_Z(P, t_3)$ , then $V_X(P, t_1) => V_Z(P, t_3)$ also holds.

The property consistency criterion is

$$V_X(P, s) \;=>\; V_Y(P, r) \qquad (2)$$

where $s$ and $r$ denote suspension time and resumption time, respectively. Any process may migrate between two hosts, so long as the two views are consistent. This criterion simplifies the implementation of migration handlers and allows more processes to migrate.

The importance of migration transparency has been discusses at length in the literature [1–3,6,7]. Migration tranparency requires that any names related to process identifiers that are passed to user processes in the system including the migrated process itself not be changed when a process is migrated. We note that migration transparency can be specified as one of the property consistency constraints.

## IV. Design of Migration Handlers

This section describes the structure of migration handlers that enforces the criteria (1) and (2) on the behalf of all migrating processes on UNIX hosts. More specifically, when a process $P$ is to be migrated from host $X$ to host $Y$, the migration handler on the source host (1) suspends the process when all pending system calls are completed; (2) identifies if the process can be migrated (correctly); (3) transfers the state of the process and the view of the environment to the destination host, and (4) closes files opened by the process, releases all locks, memory space and so on to leave the source host in a consistent state. Upon receiving the information sent by the migration handler on the source host, the migration handler on the destination host (5) resumes the process and (6) maintains a consistent view on the behalf of the migrated process.

### IV.1. Modeling Objects in C++

We consider here a network of UNIX hosts on which all user programs and system kernels are written in the C++ programming language. The C++ programming language supports data encapsulation [15]. A *class* in C++ is a user defined type. An instantiation of a class is an object of that class. Hence, for every process in the system, there is an instantiation of the user process class. Similarly, each environment object $E_i$ is an instantiation of the corresponding class type $e_i$. For example, **Figure 1** shows the declaration of classes **File** and *clock*. The variables *u_ofile, u_pofile* and *u_cmask* are called *class members.* Class members are variables declared inside an object; only member functions can operate on them. The class members **ktime* and **kztime* of *clock* are kernel variables to indicate time and time zone, respectively. (A kernel variable in

```
class   File       {
        struct     file *u_ofile;     /* file structures for open files     */
        char       u_pofile;          /* per-process flags of open files    */
        short      u_cmask;           /*mask for file creation              */
public:
        open(char*, int, int);        /* a list of system calls for         */
        close();                      /* file operations                    */
        read(char*,int);
        flock(int);
        ..........
        ..........
};      /* File */

class clock        {
#ifdef  KERNEL
        static     struct             timeval *ktime;
        static     struct             timezone *kztime;
#endif
public:
        gettimeofday(struct timeval*, struct timezone*);
        settimeofday(struct timeval*, struct timezone*);
};      /* clock */

/* Implementation of public functions        */
/* similar to old code                       */
```

**Figure 1** Examples: Classes *File* and *Clock*

UNIX is declared between the keyword pair #ifdef KERNEL and #endif.) If a kernel variable is shared among processes on a single host, all objects of the same class refer to a single copy of it. Such a kernel variable is declared *static* as is done for **ktime* and **kztime* in the clock object. Each system call is declared as a *member function* in the *interface* of some environment object $E_i$ [15], i.e., in the *public* part of the declaration of $e_i$. For example, **Figure 1** shows the declaration of classes *File* and *clock*. The system calls *open(), close(), read(), flock()* etc. in **Figure 1** are declared as member functions of the class *File*. According to the model in Section III, these system calls are external operations of any environment object of class *File*, and can be invoked by other objects. Similarly, the system calls *gettimeofday()* and *settimeofday()* are specified as external operations of an object of class *clock*.

C++ is an implementation language and it does not have the $V$–function and $O$–function specification facilities required in our model. However, we can easily model operations defined in C++ as $V$–functions, $O$–functions or $OV$–functions. In particular, we model class members as primitive $V$–functions. Thus each class member corresponds to a state variable. A system call in UNIX usually changes the state of the environment object as well as returning a value that reflexes such a change. For this reason, we model a system call as an $OV$–function call, i.e., an $O$–function call followed immediately by a derived $V$–function call. Since a data structure in C++ may be declared in the argument list of an external operation, a derived $V$–function may not return a single value. In particular, values returned by an invocation of a member function are values returned by the invocation of the corresponding derived $V$–function.

## IV.2. View Maintenance for Each Process

It is relatively easy to implement the functionality required to satisfy the state consistency criterion. Simple packages described in [8,9] carry out process suspension and resumption in such a way that (1) is satisfied. The suspension module restores the source host to a consistent state by executing a procedure call similar to the standard *kill(1)* function provided by UNIX system [16] after the process's image containing its state information is transmitted to the destination host. We will not be concerned hereafter with the suspension and resumption modules, but confine our attention to the part of migration handlers that maintain consistent views on the behalf of migrating processes. We call this function of migration handlers as view maintenance.

In UNIX environments, the values generated for a kernel variable is often location dependent; that is, the values are unique on a single host. Inconsistency in the view of a process can arise only when the process access location dependent kernel variables. If no derived $V$–function value is computed from the value of a kernel variable, that variable cannot be observed by any user process. It cannot cause inconsistency in the views of processes during migration. Hence, in the implementation of migration handlers, we are concerned only with those kernel variables whose values or computed values are returned to user processes through system calls and hence are visible to user processes.

According to the definition given in Section III, a process's view of its environment is the set of values returned by all system calls invoked by the process since its creation time. To facilitate view maintenance, however, it is only necessary to save the values returned by the most recent call of every invoked $V$–function. A simple explanation is given as the following: Suppose that at time $t_{n+1}$, the process $P$ makes a system call that includes an invocation of a $V$–function $\mathbf{V}_\mathrm{i}$. Hence $V_X(P,t_n)$ and $V_X(P,t_{n+1})$ differ only by the value of the $V$–function call $\mathbf{V}_\mathrm{i}$; that is,

$$V_X(P,\ t_{n+1})\ =\ V_X(P,\ t_n) \cup \{\ V_i(t_{n+1})\}$$

Let $V_i(t_m)$ be the values returned by the last invocation of $\mathbf{V}_\mathrm{i}$ before $t_{n+1}$, and $\{ V_i(t_k) \,|\, 0 \leq k \leq m\ \}$ be the set of all values returned by the invocations of $\mathbf{V}_\mathrm{i}$. According to the transitive property, if $V_i(t_{n+1})$ is consistent with $V_i(t_m)$ and $V_i(t_m)$ is consistent with $V_i(t_k)$ for $0 \leq k < m$, then $V_i(t_{n+1})$ is consistent with all $V_i(t_k)$ for $0 \leq k < m$. Therefore, to check consistency of $V_X(P,t_{n+1})$ with $V_X(P,t_n)$, it is only necessary to have the value of $V_i(t_m)$. That is, the values returned by the most recent call of $\mathbf{V}_\mathrm{i}$.

To obtain information required for view maintenance on the behalf of $P$ at any moment, we associate every member function that corresponds to a derived $V$–function with a boolean **marker** and a duplicate of the returned parameter. The markers of all $V$–functions are initially set to false (unmarked). When a $V$–function is called by $P$, its associated marker variable is set to true (marked) and the value returned by the $V$–function is saved in its duplicate (called a marked duplicate). If a $V$–function is called several times, only the most recent value is saved. At

```
/*   U-block:          Per process structure        */

     a list of include files containing interface declaration of environment objects

     struct   user {
              class     pcb u_pcb;                /* process control block as class  */
              class     proc *u_procp;            /* pointer to proc structure       */
              int       *u_ar0;                   /* address of users saved R0       */
              char      u_comm[MAXNAMLEN + 1];

/* syscall parameters, results and catches (override entry block)                    */

              int       u_arg[8];                 /* arguments to current system call*/
              int       *u_ap;                    /* pointer to arglist              */
              char      u_error;                  /* return error code               */
              union {                             /* syscall return values           */
                        struct          {
                                        int       R_val1;
                                        int       R_val2;
                        } u_rv;
                        off_t           r_off;
                        time_t          r_time;
              } u_r;
              ..........
/* data structures to maintain consistent view for system calls                      */
/* that will pass host-dependent properties to user processes                        */

              class     File *u_ofile[NOFILE];    /* file structures for open files  */
              struct    fview           {
                        int             fmarker;  /* for open() file or socket       */

                        a list of marks and duplicates for all derived V-functions
                        of file operations in standard UNIX system calls
                        ...........

              } flview[NOFILE];                   /* array of duplicates for file operations*/

              class     clock           *cptr;                  /* clock object pointer*/
              struct    cview           {
                        int             mark;                   /* gettimeofday() marker*/
                        int             larger;                 /* will be explained later*/
                        struct          timeval *tp;            /* duplicate of time value*/
                        struct          timezone *tzp;          /* duplicate of time zone*/
              }         *clckview;

              ..........
```

**Figure 2** Sample Pseudo code of U-block Declaration

any time, the information needed by a migration handler to carry out view maintenance on the behalf of $P$ can be obtained by scanning the marker of every derived $V$–function to determine whether the function has been called and by checking the value saved in every marked duplicate to determine whether property consistency constraints are satisfied. In other words, the values in $V_X(P,s)$ needed for view maintenance are obtained from all the marked duplicates of derived $V$–functions of $E_X(P)$.

The data structure used to support the interaction of a process with its environment is called a **U–block** in our implementation. A U–block is similar to the *user* structure defined in the header file **user.h** in the Bsd 4.2 version of UNIX [17] except that each U_block includes additional data structures to keep the markers and duplicates of $V$–functions for every instantiated environment object. **Figure 2** shows a sample pseudo code of the U–block declaration. The data structures, *flview[NOFILE]* and **clckview*, are declared as *fview* and *cview* for file objects and clock objects to support view maintenance. (The operations on *flview[NOFILE]* and **clckview* to support view maintenance will be discussed later) When a process invokes a system call, the name of the system call is placed in the U–block along with the list of actual arguments of that call. The execution control is then switched from the user mode into system mode and the system call is executed. Before the control is switched back to user mode, the returned value(s) and error code if there is any are placed in the U–block. In addition, the marker associated with the corresponding $V$–function is marked and the duplicate is updated. **Figure 3** shows how the execution control is switched back and forth between a user process and a system process. **Figure 4** shows the relationship between the content of a U–block and the environment objects at run time.

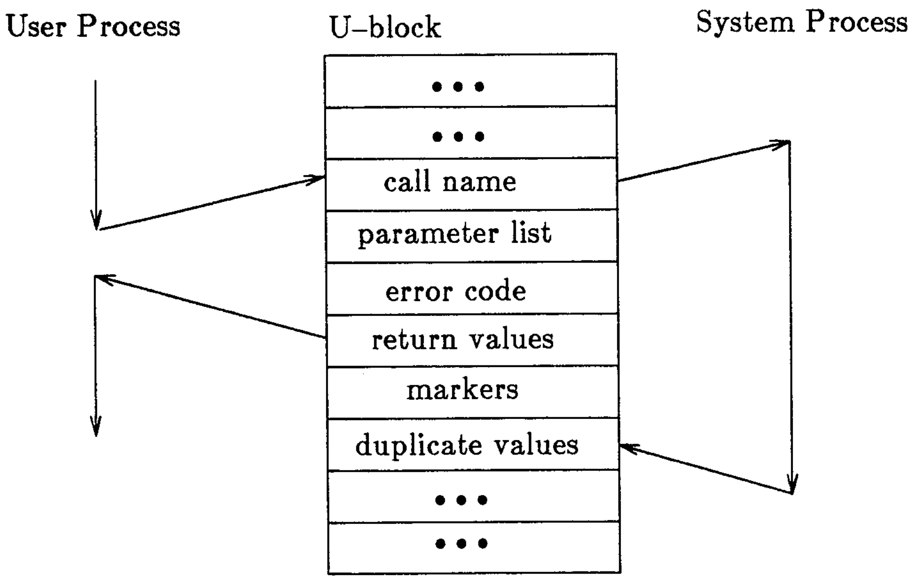


**Figure 3** Execution Control Between a Process and its Environment

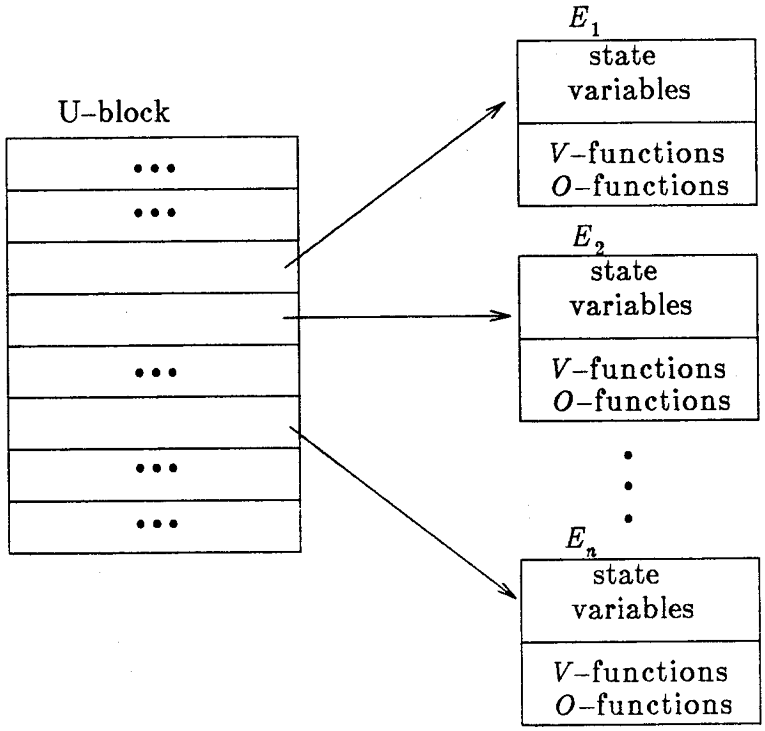


**Figure 4** User Process Object and its Environment Objects

## IV.3 The Functionality of Migration Handlers

In general, property consistency constraints are part of the specification of an operating system. Sometimes these constraints are not explicitly specified even when the implementation fulfills the requirement imposed by these constraints. However, in order to support correct migration of processes, property consistency constraints must be specified explicitly.

To identify whether a suspended process can be migrated, the migration handler on the source host determines what the process has observed about its environment objects by scanning the U_block of the process for markers that are set and for marked duplicates. The process can be migrated if for every environment object instantiated, (1) no property consistency constraint is specified for the object, or (2) the duplicates associated with all derived *V*–functions that have constraints are still unmarked, or (3) the property consistency constraints of invoked *V*–functions can be maintained on the destination host. Otherwise, the process is not migrated.

After the state information and the view of the process (i.e., the stack, text, register content, etc. and its U–block) are transferred from the source host to the destination host, the migration handler on the destination host restores the environment state for the process by doing the following steps.

**Step One:** Bind all kernel variables that are accessed by *P* through external operations of environment objects on the source host to their corresponding local kernel variables on the destination host.

**Step Two:** For every *V*–function or *OV*–function that has its marker set in the U–block, instantiate its environment object by invoking the same *V*–function (or *OV*–function ) on the destination host. Then check if property consistency constraints are satisfied by comparing the

```
/*   Declaration from Figure 1                         */

     class clock          {
     #ifdef  KERNEL
             static    struct          timeval *ktime;
             static    struct          timezone *kztime;
     #endif
     public:
             gettimeofday(struct timeval*, struct timezone*);
             settimeofday(struct timeval*, struct timezone*);
     };      /* clock */

/*   Declaration from Figure 2                         */
     struct  cview     {
             int       mark                      /* marker                      */
             int       larger;                   /* view maintenance indicator  */
             struct    timeval *tp;              /* duplicate of time value     */
             struct    timezone *tzp;            /* duplicate of time zone      */
     }       *clckview;

/*   initially CVIEW->mark=0; CVIEW->larger=1; for migration handler           */

/*   Normal code used each time the system call of                             */
/*   gettimeofday(timeval *tval, timezone *zval) is requested                  */
     {
             C.gettimeofday(tval,zval);
             if (CVIEW->larger)       {          /* consistent                  */
                       copy(tval,CVIEW->tp);  /* update duplicate             */
                       copy(zval,CVIEW->ztp);
             return tval and tzval;
             } else    if (bigger(tval,CVIEW-tp)&&bigger(zval,CVIEW-tzp))       {
                       copy(tval,CVIEW->tp);
                       copy(zval,CVIEW->tzp);
                       CVIEW->larger =1;
                       return tval and zval;
             } else                              /* return duplicates           */
                       return CVIEW->tp and CVIEW->ztp instead of tval and zval
     };

/*   code used by the migration handler on the destination host to             */
/*   instantiate the clock object and check for consistency at resumption time */

     {
             bind *ktime and *kztime to corresponding kernel variables
             C.gettimeofday(timeval *tval,timezone *zval);
             if (bigger(tval, CVIEW->tp))        /* consistent                  */
                       CVIEW->larger =1;
             else CVIEW->larger = 0;             /* indication of possible inconsistency*/
                                                 /* view of P's is not updated then*/
     };
```

**Figure 5** Implementation of View Maintenance for Clock Object

returned values of the $V$–function call (or $OV$–function call ) with the corresponding duplicate values in the U_block transferred from the source host. If an inconsistency should occur, the migration handler must resolve the inconsistency accordingly.

In an actual implementation, the re–instantiation of environment objects and the view maintenance can be done dynamically rather than statically as described above. An environment object does not need to be instantiated until the resumed process invokes some of its external operations. The migration handler on the destination host can save some unnecessary overhead if the environment object is not referenced again by the migrated process. However, the migration handler on the destination host must check for view consistency every time a $V$–function or $OV$–function is called unless the handler can distinguish the first invocation of each $V$–function from subsequent invocations. The solution to resolve any inconsistency depends on what the invariant conditions are. Following are two examples showing how static instantiation and dynamic instantiation are done.

In the example shown in **Figure 5**, the process $P$ is migrated and the view maintenance for the monotonic property of time is done statically at resumption time. Suppose that $C$ is a clock object in $E_X(P,s)$. The variable *CVIEW* is a pointer to **clckview* list of type *cview* and it is used to do view maintenance for $C$. *C.settimeofday()* and *C.gettimeofday()* are external operations of clock object $C$. The procedure call *copy(arg1,arg2)* is used to copy the content from the first argument to the second argument of the same type. The function *bigger(arg1,arg2)* tests if the first argument is bigger than the second argument. The two variables *CVIEW->tp* and *CVIEW->ztp* are used as pointers to the duplicates for the $V$–function call *C.gettimeofday()*. *Marker* is the marker of *C.gettimeofday()*. The variable *larger* is used for the migration handler to maintain view consistency. Initially, *larger* is set to true. *Larger* is set to false only if *C.gettimeofday()* returns a smaller value than the values returned by the last call. When the

```
/*   Declaration from Figure 1                    */

     class    File       {
              struct     file *u_ofile;           /* file structures for open files   */
              char       u_pofile;                /* per-process flags of open files */
              short      u_cmask;                 /* mask for file creation          */
     public:
              open(char*, int, int);              /* a list of system calls for      */
              close();                            /* file operations                 */
              read(char*,int);
              flock(int);
              .........
     };       /* File */

/*   Declaration from Figure 2                    */

     class    File *u_ofile[NOFILE];              /* file structures for open files   */
     struct   fview      {
              int        fmarker;                 /* for open() file or socket        */
              a list of duplicates for each V-function derived
              from file operations of standard UNIX system calls
     } flview[NOFILE];                            /* array of duplicates for file operations */

/*   code of migration handler to maintain consistency and view update           */
/*   For a system call request read(d,buf,nbytes);                               */
/*   FVIEW=flview[d]; FVIEW.offset,FVIEW.flags,FIEW.mode,FVIEW.whence             */
/*   are duplicates; but are not shown in the declaration of fview               */

     {
              if (flview[d].fmarker==0 )
                         report error message;                 /* not opened yet */
              else if (first call after migration)    {        /* get to last read position */
                         u_ofile[d]->open(FVIEW.path,FVIEW.flags,FVIEW.mode);
                         u_ofile[d]->lseek(FVIEW.offset,FVIEW.whence);
                         u_ofile[d]->read(buf,nbytes);
                         update content of FVIEW;              /* the same as flview[d]*/
              }
              else       {                                     /* normal read operation */
                         u_ofile[d]->read(buf,nbytes);
                         update content of FVIEW;              /* normal view update*/
              }
     }

/*   code for other system call request are not shown here            */
     ..........
     ..........
```

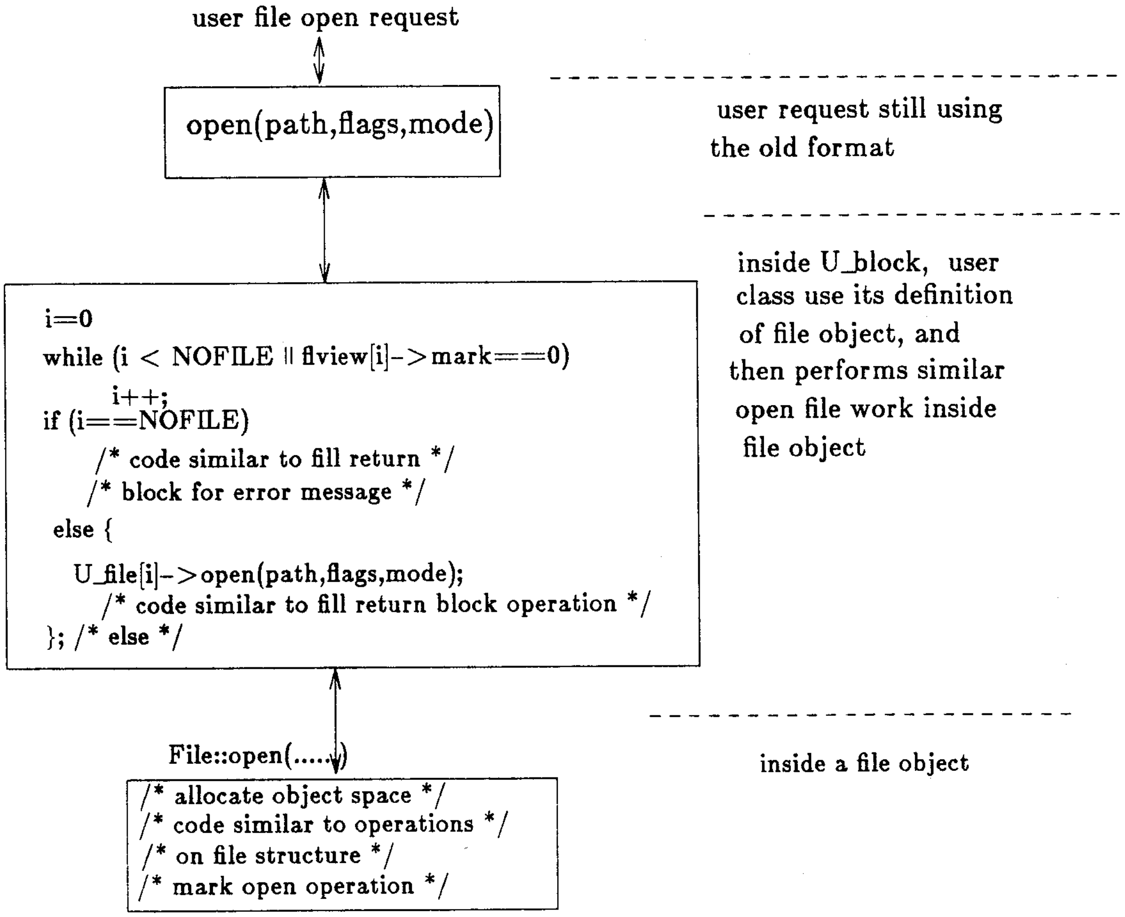
**Figure 6** Sample Pseudo Code of Migration Handler for File Object

migration handler on the destination host resumes the process, it first binds **ktime* and **kztime* to their corresponding kernel variables on the destination host. Then the migration handler invokes *C.gettimeofday()* and checks the returned values against the duplicate values transferred from the source host using the function *bigger()*. If the returned values are smaller than the two duplicates, the handler sets *larger* to false before $P$ is resumed. Thereafter, the values returned to $P$ each time *gettimeofday()* is called are the values from the duplicates instead of the kernel variable values and the two duplicates are not updated until the kernel variable values are bigger. Alternatively, the process can be blocked until the kernel variables are bigger than the duplicates. This alternative is not shown in **Figure 5**.

As an example of migration handlers that handle the re-instantiation of environment objects dynamically, suppose that a file object *u_ofile*[*d*] in $E_X(P,s)$ has opened on the source host by the *open()* system call where $d$ is the index of the file descriptor returned to $P$. The variable *FVIEW* has the type *fview* and *FVIEW* is assigned to *flview*[*d*]. **Figure 6** shows the sample code of the migration handler to do view maintenance for *read()* operation on the behalf of $P$. When $P$ is suspended on the source host, *u_ofile*[*d*] is closed in order to keep the source host in a consistent state. Using the dynamic approach, when the process is resumed on the destination host, *u_ofile[d]* is not opened again until the migrated process attempts to read the file. The migration handler then instantiates the file object by the *open()* operation and by the *lseek()* operation to move the offset of *u_ofile*[*d*] so that the requested *read()* starts from the right place. The overhead of the dynamic approach for the resumption of a migrated process is smaller than that of the static approach, particularly when the migrated process interacts with very few environment objects in $E_X(P,s)$. Take the file object as an example, if the migrated process does not read *u_ofile*[*d*] fR after the migration, there is no need for the migration handler to open the file when the process is resumed. Whereas, in the static approach, the migration handler has to re-open the file at resumption time.

## V. Conclusion

To support process migration in UNIX environments, the main problem is how to encapsulate the location dependent features of the system in such a way that a host independent virtual environment is maintained by the migration handlers on the behalf of each migrated process. An object-oriented approach is used to describe the interaction between a process and its environment. More specifically, we introduce environment objects in UNIX systems to carry out


user file open request
open(path,flags,mode)
user request still using the old format
inside U_block, user class use its definition of file object, and then performs similar open file work inside file object
inside a file object
File::open(.....)

```
i=0
while (i < NOFILE || flview[i]->mark==0)
        i++;
if (i==NOFILE)
      /* code similar to fill return */
     /* block for error message */
 else {
   U_file[i]->open(path,flags,mode);
      /* code similar to fill return block operation */
}; /* else */
```

```
File::open(.....)
/* allocate object space */
/* code similar to operations */
/* on file structure */
/* mark open operation */
```

**Figure 7** The Mapping of System Calls and Object Operations

the user–environment interaction. Our implementation of the migration handlers is based on both the state consistency criterion and the property consistency criterion.

The current UNIX releases such as 4.2Bsd and System V [18] do not support process migration. To make process migration available as an additional feature of the system without affecting current users and existing code ( not the system level code), we need to hide the environment objects from user processes such that the system calls are still invoked in the old way. For this reason, a mapping between the standard system calls and the external operations of environment objects must be done at the system level. **Figure 7** is an example of a *open()* file operation to show how this mapping can be done. The top block in **Figure 7** is the standard *open()* system call. The middle block is the code that searches for an open slot in the U–block for a file object. Then the actual open operation is executed in the file object *U_file*[*i*].

### References

[1] Powell, M. L. and Miller, B. P., "Process Migration in DEMOS/MP", *Operating System Review*, Vol. 17, No. 5, 1983.

[2] Theimer, M. M., Lantz, K. A., and Cheriton, D. R., "Preemptable Remote Execution Facilities for the V–System", *ACM Operating Systems Review*, Vol. 19, No. 5, 1985

[3] Rashid, R. F. and Robertson, G. G., "Accent: A Communication Oriented Network Operating System Kernel", *Proc. of the Eighth Symposium on Operating Systems Principles*, December, 1981.

[4] Ni, L. M., Xu, C.–W. and Gendreau, T. B., "Draft Algorithm – A Dynamic Process Migration Protocol", *Proceedings of the Fifth International Conference on Distributed Computing Systems*, May 13–17, 1985.

[5] Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", *IEEE Transactions on Software Engineering*, SE–3, No. 1, January, 1977.

[6] Rennels, D. A., "Distributed Fault–Tolerant Computer Systems", *IEEE Computer*, Vol. 13, No. 3, March, 1980.

[7] Solomon, M. H. and Finkel, R. A., "The Roscoe Distributed Operating System", *Proceedings of the Seventh Symposium on Operating Systems Principles*, 10–12 December 1979.

[8] Cagle, R. P., "Process Suspension and Resumption in UNIX System V Operating System", *Thesis Report UIUCDCS-R-86-1240*, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1986

[9] Chen, A. Y.-C., "An UNIX 4.2BSD Implementation of Process Suspension and Resumption", *Thesis Report UIUCDCS-R-86-1286*, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1986

[10] Lu, Chin and J. W. S. Liu, "The Notion of Correctness in Process Migration", *Report UIUCDCS-R-86-1307*, Department of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1986

[11] Cheheyl, M. H., Gasser, M., et. al., "Verifying Security", *ACM Computing Surveys*, Vol. 13, No. 3, September 1981

[12] Popek, G.J. and Farber, D.A., "A Model for Verification of Data Security in Operating Systems", *Communication ACM*, Vol. 21, No. 9, September 1978

[13] Parnas, D. L., "A Technique for Software Module Specification with Examples", *Communication ACM*, Vol. 15, No. 5, May 1972

[14] Halpern, J. Y. and Moses, Y., "Knowledge and Common Knowledge in a Distributed Environment", *ACM SIGACT-SIGOPS*, Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1984

[15] Stroustrup, B., "The C++ Programming Language", *Addison-Wesley Publishing Company*, 1986

[16] Bourne, S.R., "The UNIX System", *Addison-Wesley Publishing Company*, 1983

[17] "UNIX Programmer's Manual Reference Guide 4.2 Berkeley Software Distribution",Computer Science Divison, Department of Electrical Engineering and Computer Science, University of California, March 1984

[18] "UNIX System V User Guide", *AT&T Technologies, Inc.* 1984