Processes

Memory Organization for an Executed Program

- When a program is loaded into memory, it is organized into three areas of memory, called *segments*:
  - *text segment,*
  - *stack segment*, and
  - *heap segment*
- The text segment (or code segment) is where the compiled code of the program itself resides.
- The stack is where memory is allocated for automatic variables within functions.
- The heap segment provides more stable storage of data for a program since memory allocated in the heap remains in existence for the duration of a program.

Stack

- local variables (variables declared inside a function) are put on the stack - unless they are declared as 'static' or 'register'
- function parameters are allocated on the stack
- local variables that are stored in the stack are not automatically initialized by the system
- variables on the stack disappear when the function exits

Heap

- Global, static, register variables are stored on the heap before program execution begins
- they exist the entire life of the program (even if scope prevents access to them - they still exist)
- they are initialized to zero
  - global variables are on the heap
  - static local variables are on the heap (this is how they keep their value between function calls)
- memory allocated by new, malloc, calloc, etc., are on the heap

Processes

- A process is a program that is executed.
- Each time a process is created, the OS must create a complete independent address space (i.e., processes do not share their heap or stack data)
- The OS maintains a process table to keep track of the active processes in the system. Information maintained for each process:
  - Program id, user id, group id
  - Program status word
  - CPU register values
  - Memory maps
  - Stack pointer
  - Open files
  - Accounting information, etc.

The FORK() System Call

Processes in UNIX are created with the fork() system call.

```
#include <sys/types.h>
#include <unistd.h>

void main(void)
{
    int pid;
    pid = fork();
    if (pid == -1) {
        printf("error in process creation\n");
        exit(1);
    }
    else if (pid == 0) child_code();
    else parent_code();
}
```

The FORK() System Call

```
#include <sys/types.h>
#include <unistd.h>

#define PROCESS 10

void main(void)
{
     int pid, j;

     for (j=0; j < PROCCESS; j++) {
          pid = fork();
          if (pid == -1) {
              printf("error in creation of process %d\n", j);
              exit(1);
          }
          else if (pid == 0)  child_code(j);
     }
      for (j = 0; j < PROCESS; j++) wait(0);
}
```

```
void child_code(int id)
{
        pid_t myid, pid;
        myid = getpid();
        pid = getppid();

        printf("My pid is %d and my parent's id is %d",
                   myid, pid);

        printf("My virtual id is %d\n", id);
        exit(0);
}
```

If we are interested to wait for a particular child, we can use instead of wait(), **waitpid**() (see man pages for more information).

Variables shared to different processes

- Before the creation of the child-processes, the parent process should dynamically allocate some shared memory space. This shared space is assigned in the memory space of the parent process.
- Each child process inherents the shared memory allocated by the parent process.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- **shmget**(): allocation of a segment of shared memory

  (int shmget(key_t key, size_t size, int shmflag;)

  int memid = shmget(IPC_PRIVATE, size, 0600|IPC_CREAT);
  - Size: number of shared bytes to be allocated
  - memid: the id of the allocated shared memory

Variables shared to different processes

- **shmat**(): attaches the shared memory segment associated with memid to the data segment of the calling process; returns a pointer to the first of the allocated shared bytes

  (void *shmat(int shmid, const void *shmaddr, int shmflg);)

  ```
  void *p = shmat(memid, 0, 0);
  ```

  Example:

  ```
  struct whatever {
      int i, j;
      float k;
  } *myvars;
  myvars = (struct whatever *) shmat(memid, 0, 0);
  ```

  Then, we can use myvars->k, myvars->j, myvars->i in the standard way.
- **shmctl**(): called by the parent process to de-allocate the shared memory allocated with shmget
- (shmctl(memid, IPC_RMID, 0);)

  (shmctl(int shmid, int cmd, struct shmid_ds *buf);)

Semaphores

- A **semaphore** is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment.
- The state of the semaphore can be updated by executing any of the following two atomic operations:
  - up() -> increases the value of the semaphore by one.
  - down(): blocks if the value of the semaphore is 0 until its value becomes >0; It decreases the semaphore's value by 1.
- A binary semaphore takes only the values 0 and 1.

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- **semget()**: creation of semaphores

(int semget(key_t key, int nsems, int semflg))

int semid = semget(IPC_PRIVATE, N, 0600|IPC_CREAT);

N: number of semaphores to be created

The initialization of the ith semaphore to the value k is done as follows:

```
union semun {
        int val;  struct semid_ds *buf;   ushort_t        *array;
} arg;
arg.val = k;
semctl(semid, i, SETVAL, arg);
```

Semaphores

- **semctl**(): de-allocation of semaphores (int semctl(int semid,int semnum,int cmd);)

```
semctl(semid, 0, IPC_RMID);
```

- **int semop**(int semid, struct sembuf *sops, unisgned nsops);

```
struct sembuf operation;
semop(semid, &operation, 1);
```

- Fields of struct sembuf: sem_num (which of the N semaphores we refer to), sem_op (-1 for down and +1 for up), sem_flg (usually 0).

```
struct sembuf operation;
operation.sem_flg  = 0;
operation.sem_num  = i;
operation.sem_op   = -1;
semop(semid, &operation, 1);          /* Down */
```

An Example: The Producer – Consumer Problem

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(struct buf
    *pbuf) {
  int item;
  while (TRUE) {
      produce_item(&item);
       down(empty);
       down(mutex);
       enter_item(pbuf, item);
       up(mutex);
       up(full);
   }
}
```

```
void consumer(struct buf
    *pbuf) {
  int item;
  while (TRUE) {
      down(full);
       down(mutex);
       item = remove_item(pbuf);
       up(mutex);
       up(empty);
   }
}
```

Producer-Consumer

```
#include <sys/types.h>
#include <unistd.h>
#define N 100

void main(void)
{
      int pid;



      pid = fork();
      if (pid == 0) consumer(buf);
      else producer(buf);
      wait(0);
      semctl(semid_full, 0, IPC_RMID);
      semctl(semid_empty, 0, IPC_RMID);
      semctl(semid_mutex, 0, IPC_RMID);
      shmctl(memid, IPC_RMID, 0);
}
```

```
int *buf;

int memid = shmget(IPC_PRIVATE, sizeof(int) * N, 0600|IPC_CREAT);
buf = (int *) shmat(memid, 0, 0);
```

```
int semid_full, semid_empty, semid_mutex;
union semun {
            int val;
            struct semid_ds *buf;
            ushort_t        *array;
} arg;

semid_full = semget(IPC_PRIVATE, 1, 0600|IPC_CREAT);
      arg.val = N;
      semctl(semid_empty, 0, SETVAL, arg);

semid_empty = semget(IPC_PRIVATE, 1,
0600|IPC_CREAT);
      arg.val = 0;
      semctl(semid_full, 0, SETVAL, arg);

semid_mutex = semget(IPC_PRIVATE, 1,
0600|IPC_CREAT);
      arg.val = 1;
      semctl(semid_empty, 0, SETVAL, arg);
```

Producer-Consumer

```
void producer(void) {
  int item = 0;
  while (item < 100) {
        down(semid_empty);
        down(semid_mutex);
        *(buf+item) = item;
        up(semid_mutex);
        up(semid_full);
       item++;
   }
}
```

```
void consumer(void) {
  int item = 0;
  while (item < 100) {
        down(semid_full);
        down(semid_mutex);
        printf("Item consumed: %d\n",
                      *(buf+item));
        up(semid_mutex);
        up(semid_empty);
       item++;
   }
}
```

```
void down(int semid) {
          struct sembuf operation;
          operation.sem_flg  = 0;
          operation.sem_num  = 0;
          operation.sem_op   = -1;
          semop(semid, &operation, 1);
    }
```

```
void up(int semid) {
      struct sembuf operation;
      operation.sem_flg  = 0;
      operation.sem_num  = 0;
      operation.sem_op   = +1;
      semop(semid, &operation, 1);
      }
```

Threads

- Threads exist within the resources of their parent processes
- yet are able to be scheduled by the operating system and run as independent entities
  - they duplicate only the bare essential resources that enable them to exist as executable code.
- A thread maintains its own:
  - Thread ID
  - Stack pointer
  - Set of registers
  - Signal masks
  - Scheduling properties (such as policy or priority)
  - stack for local variables, return addresses

Threads

Threads in the same process share:

- Process instructions
- Heap data
- open files (descriptors)
- current working directory
- User and group id
- signals and signal handlers

Because of this sharing of resources:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Why using threads?

- To realize potential program performance gains.
  - a thread can be created with much less operating system overhead than a process
  - managing threads requires fewer system resources than managing processes
- Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
  - Overlapping CPU work with I/O
  - Efficient interleaving of tasks which service events of indeterminate frequency and duration
    - a web server can both transfer data from previous requests and manage the arrival of new requests.

Designing Threaded Programs

- Programs having the following characteristics may be well suited for threads:
  - Work that can be executed, or data that can be operated on by multiple tasks simultaneously
  - Block for potentially long I/O waits
  - Use many CPU cycles in some places but not in others
  - Must respond to asynchronous events
  - Some work is more important than other work (priority interrupts)

POSIX Threads

The subroutines of Pthreads API can be grouped as follows:

- **Thread management**: Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling, etc.)
- **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- **Condition variables**: Routines that address communication between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
- **Synchronization**: Routines that manage read/write locks and barriers.

The PThreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects (stack management)
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

A Simple Example

```
void *PrintHello(void *threadid) {
    long tid;
    tid = *((long *)threadid);

    printf("Hello World! It's me, thread #%ld!\n",
            tid);
    pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
long *taskids[NUM_THREADS];
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++) {
        printf("In main: creating thread %ld\n", t);
        taskids[t] = (long *) malloc(sizeof(long));
        *taskids[t] = t;
        printf("Creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&taskids[t]);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

UNIX Signals

- Signals are various notifications sent to a process in order to notify it of various "important" events.
- They interrupt whatever the process is doing, and force it to handle them immediately.
- When a process receives a signal of some type, it can either take the default response, ignore the signal, or catch the signal.
- If the signal is caught, the system will call a handler function (called signal handler) when the signal is delivered.
- When a signal handler returns, the process continues execution from wherever it happened to be before the signal was received.
- Sending signals using the Keyboard
  - Ctrl-C, Ctrl-Z, fg, bg
- Sending signals from the command line
  - kill

Catchable and Non-Catchable Signals

- Some signals processes cannot catch
  - KILL
  - STOP (sometimes used for de-bugging)
- Other signals are catch-able
  - SEGV, BUS
  - It is possible to catch these signals in order to do some cleanup
- signal(): sets a signal handler for a type of signal.
- Pre-defined signal handlers
  - SIG_IGN (ignore the specified signal)
  - SIG_DFL (set the default signal handler for the given signal)

Example

```
#include <stdio.h>          /* standard I/O functions */
#include <unistd.h>         /* standard unix functions, like getpid() */
#include <sys/types.h>      /* various type definitions, like pid_t */
#include <signal.h>         /* signal name macros, and the signal() prototype */

/* first, here is the signal handler */
void catch_int(int sig_num) {

    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);

    /* and print the message */
    printf("Don't do that\n");
    fflush(stdout);
}

int main(void) {
    /* and somewhere later in the code.... */ . .
    /* set the INT (Ctrl-C) signal handler to 'catch_int' */
    signal(SIGINT, catch_int);

    /* get into an infinite loop of doing nothing. */
    for ( ;; ) pause();
}
```

Masking Signals

- A second signal may occur while a signal handler is executed.
- Masking signals in a global context
  - sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
    - Specify a set of signals to block and returns the list of signals that were blocked
  - Functions to handle sigset_t mask_set:
    - sigemptyset(&mask_set)
    - sigaddset(&mask_Set, SIGINT)
    - sigdelset(&mask_Set, SIGINT)
    - sigismember(&mask_set, SIGINT)
    - sigfillset(&mask_Set)

```
int ctrl_c_count = 0;
#define CTRL_C_THRESHOLD 5

void catch_int(int sig_num) {
    sigset_t mask_set;                     /* used to set a signal masking set. */
    sigset_t old_set;                 /* used to store the old mask set. */

    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    /* mask any further signals while we're inside the handler. */
    sigfillset(&mask_set);
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);

    ctrl_c_count++;
    if (ctrl_c_count >= CTRL_C_THRESHOLD) {
            char answer[30];          /* prompt the user to tell us if to really exit or not */
            printf("\nRealy Exit? [y/N]: ");
            fflush(stdout);
            gets(answer);
            if (answer[0] == 'y' || answer[0] == 'Y') {
                        printf("\nExiting...\n");
                        fflush(stdout); exit(0);
            }
            else {
                        printf("\nContinuing\n"); fflush(stdout); /* reset Ctrl-C counter */
                        ctrl_c_count = 0;
            }
    }
    /* no need to restore the old signal mask - this is done automatically, */
    /* by the operating system, when a signal handler returns.*/
}
```

```
void catch_suspend(int sig_num) {
    sigset_t mask_set;      /* used to set a signal masking set. */
    sigset_t old_set;       /* used to store the old mask set. */

    /* re-set the signal handler again to catch_suspend, for next time */
    signal(SIGTSTP, catch_suspend);

    /* mask any further signals while we're inside the handler. */
    sigfillset(&mask_set);
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);

    /* print the current Ctrl-C counter */
    printf("\n\nSo far, '%d' Ctrl-C presses were counted\n\n", ctrl_c_count);
    fflush(stdout);
    /* no need to restore the old signal mask - this is done automatically, */ /*
    by the operating system, when a signal handler returns. */
 }

int main(void) {
    signal(SIGINT, catch_int);
    signal(SIGTSTP, catch_suspend);
    for ( ;; ) pause();
}
```

Threads and Signals

- If the disposition for a signal type is
  - termination
    - Such signals will terminate all threads, and the process will terminate.
  - ignore
    - Such signals will be ignored by all threads.
  - catch
    - Any thread responding to such signals will enter the same handler function.
- signal masks are maintained per thread.

Non-reentrant functions and errno

- Reentrancy: possibility of a process to attempt to re-enter a function
- Examples of non-reentrant functions with respect to threads:
  - rand() (returns the next pseudo-random in a sequence determined by an initial seed value; as a side effect it updates the seed value enabling the sequence to progress). Different threads executing in parallel may see the same result.
  - Functions that operate on character streams (getc(), getchar(), putc(), putchar())
- Reentrant versions are currently provided (rand_r())
- There is often a trade-off between achieving reentrancy and performance (so sometimes non-reentrant versions are also provided and the user can choose)

Appropriate Compilation

- Solaris: -D_REENTRANT
- AIX: -D_THREAD_SAFE

System Global Variables

- POSIX.1c functions avoid using errno; instead, they return the error number directly as the function return value, with a return value of zero indicating that no error was detected.
- Each reference to errno can be made thread-specific by making errno a macro that expands to a function call.

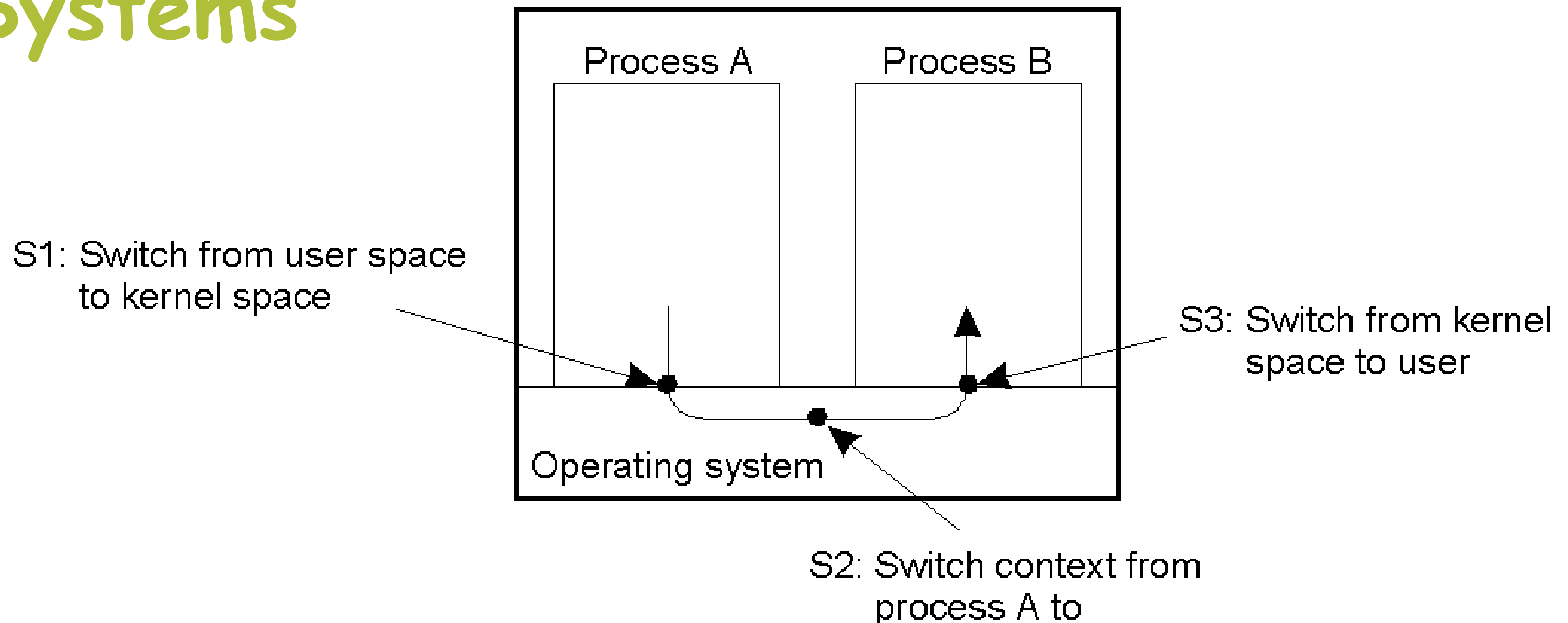
Threads, Signals and non-Reentrant Procedures

- More on UNIX signals:
  http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html
- More on non-reentrant procedures:
  http://www.unix.org/whitepapers/reentrant.html

Thread Usage in one machine

- Whenever a blocking system call is executed by a process, the process is blocked.
- This is not always desirable
  - Excel spreadsheet
- Exploit parallelism when executing on a multiprocessor system
- Many applications are easier to structure as a collection of cooperating threads.
  - Word processor: handling user input, spelling and grammar checking, document layout, index generation, etc.

Thread Usage in Nondistributed Systems



- Some applications are developed as a collection of cooperating programs, each to be executed by a separate process.
- Context switching as the result of IPC

Multithreaded Clients

- distributed systems operating on wide-area networks may need to conceal long inter-process message propagation times
  - Initiate communication and immediately proceed with something else

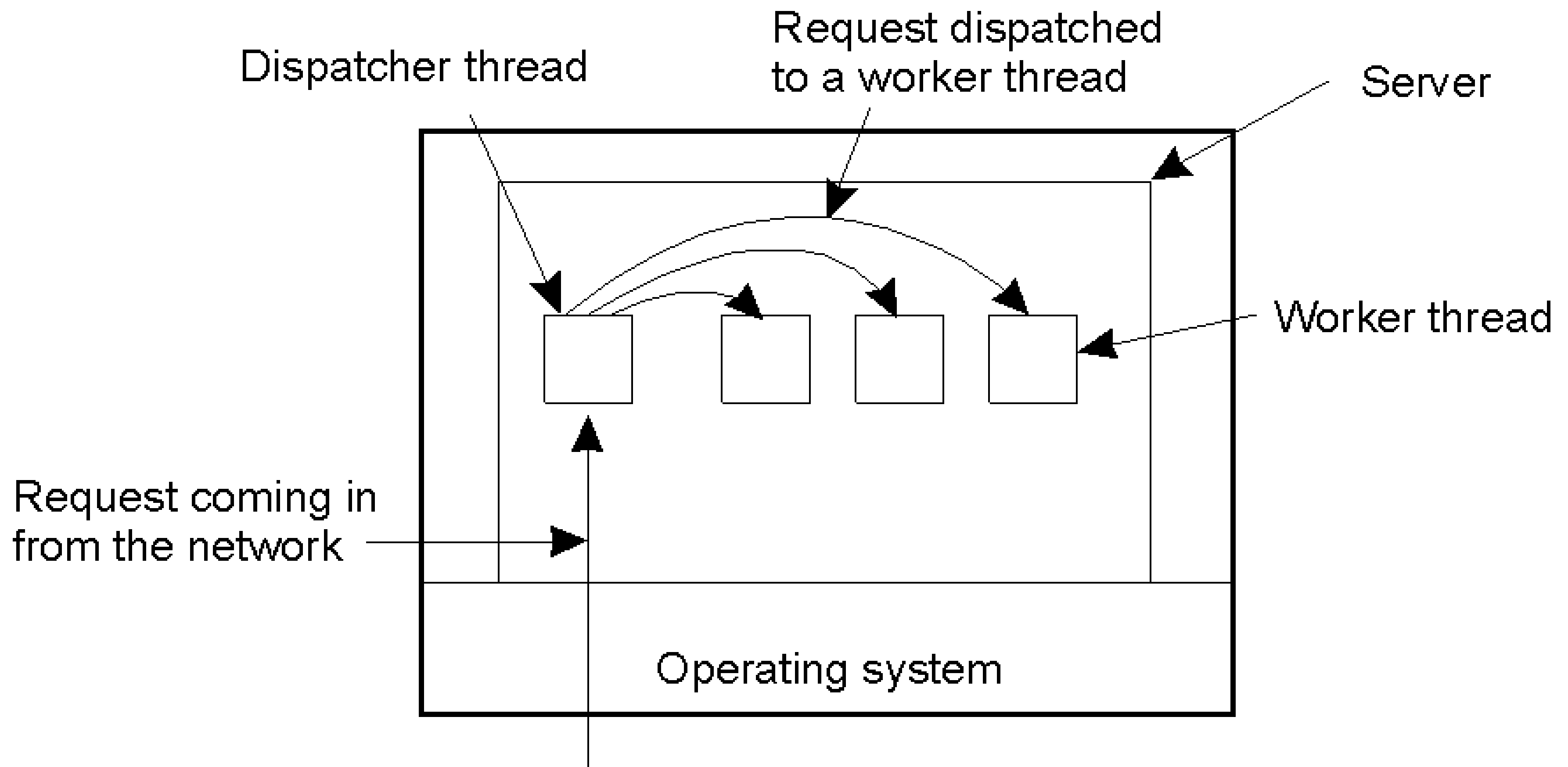
  Example: Web Browsers
  - Some browsers start displaying data while they are still coming in
  - As soon as the main html file has been fetched, different threads can be activated to take care of fetching the other parts.
- Achieve load balancing and increase performance
  - Connections from browsers may be set up to different server replicas
    - Allows data to be transferred in parallel
    - Display the full document much faster

Servers

Model	Characteristics
Single-threaded process	No parallelism, blocking system calls
Threads	Parallelism, blocking system calls

- Iterative Server
  - handles each request and returns a response
- Concurrent Server
  - passes the request to a separate thread or process and immediately waits for the next incoming request

Multithreaded Servers

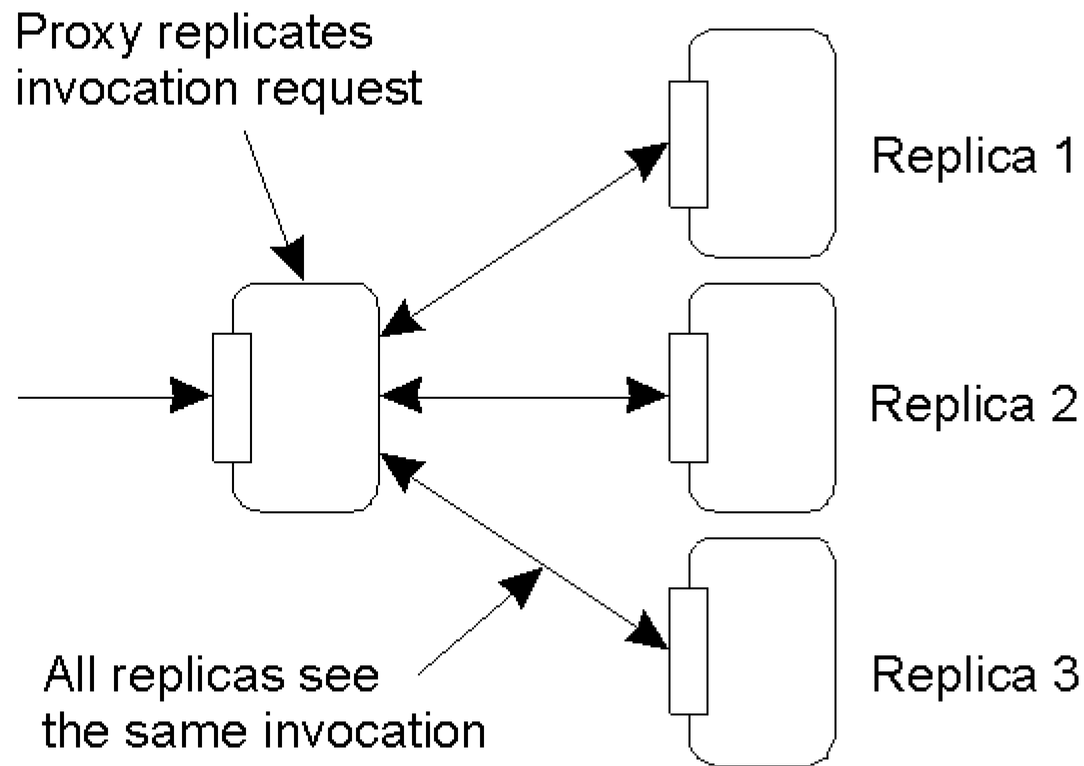


- A multithreaded server organized in a dispatcher/worker model.

Multithreaded Clients

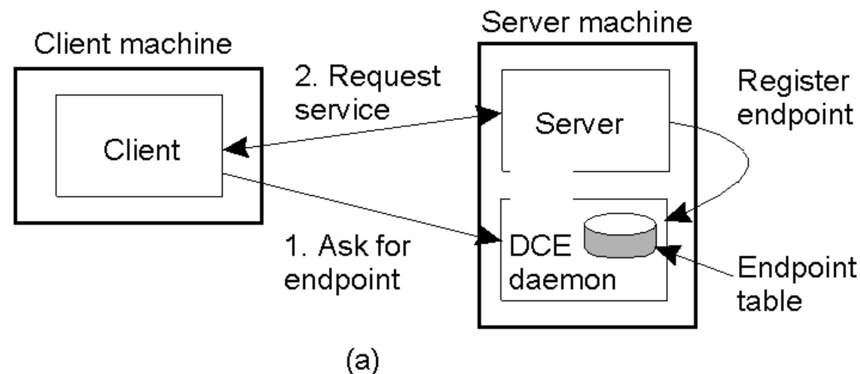
- User Interfaces
  - X Window System, Compound Documents
- Client-Side Software for Distribution Transparency
  - distribution transparency
    - the client is not aware that it is communicating with remote processes
    - Distribution is less transparent to servers for reasons of performance and correctness (e.g., replicated servers may have to communicate, etc.)
  - access transparency -> client stub
  - location-related transparency
    - use a convenient name system
    - When a client is bound to a server, it can be informed about any change to the location of the server
      - Hide server current location
      - Rebind if necessary replication transparency
  - failure transparency
    - client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts.

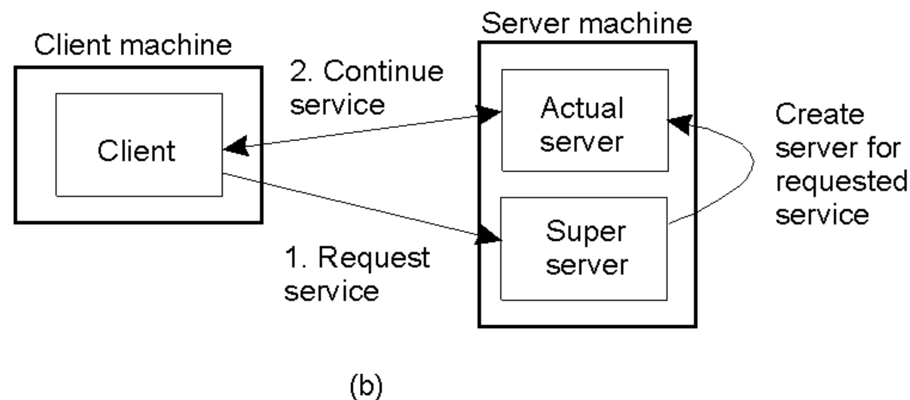
Client-Side Software for Distribution Transparency



- A possible approach to transparent replication of a remote object using a client-side solution.

Servers - How do clients know the port of a service?





- ***Globally assign end-points for well-known services*** (FTP port = 21, web port = 80)
- **Run a special daemon** on each machine to keep track of the current endpoint of each service.
    - The daemon listens to a well-known endpoint
    - Clients first contact the daemon and then the server

- **Superservers** (inetd) -> uses memory more efficiently, since the specific servers run only when needed

How a server can be interrupted?

- Out-of-band data
  - Data that is to be processed by the server with higher priority than any other data from that client.
  - Server listens to a separate control endpoint to which the client sends out-of-band data.
  - With a lower priority the server listens to the endpoint through which the normal data passes

Stateless - Statefull Servers

- Stateless Server
  - Does not keep information on the state of its clients
    - Example: Web servers
- Statefull Server
  - Maintain information on its client
    - Example: file server that allows a client to keep a local copy of a file (even for performing updates)
- ☹ If a crash occurs, it needs to recover its entire state as it was just after the crash -> enabling recovery can introduce considerable complexity
- The choice for a stateless or stateful design should not affect the services provided by the server
  - stateless file servers
- A server may sometimes want to keep a record on a client's behavior so that it can more effectively respond to its requests
  - cookies

Object Servers

- An object server is a server tailored to support distributed objects. It acts as a place where objects live.
  - it provides only the means to invoke local objects based on requests from (probably remote) clients
  - services are implemented by the objects residing in it

Alternatives for Invoking Objects

- To invoke an object, the server needs to know:
  - which code to execute
  - on which data it should operate
  - whether it should start a separate thread to take care of the invocation, etc
- Different Policies
  - Transient objects
    - Create a transient object at the first invocation request, and to destroy it as soon as no clients are bound to it anymore.
    - Create all transient objects at the time the server is initiated
      - What are the advantages and disadvantages of these two policies?
  - Should objects share code or state?
    - place each object in a memory segment of its own
    - let objects at least share code
  - Threading
    - implement server as a simple thread of control, or
    - have several threads, one for each object

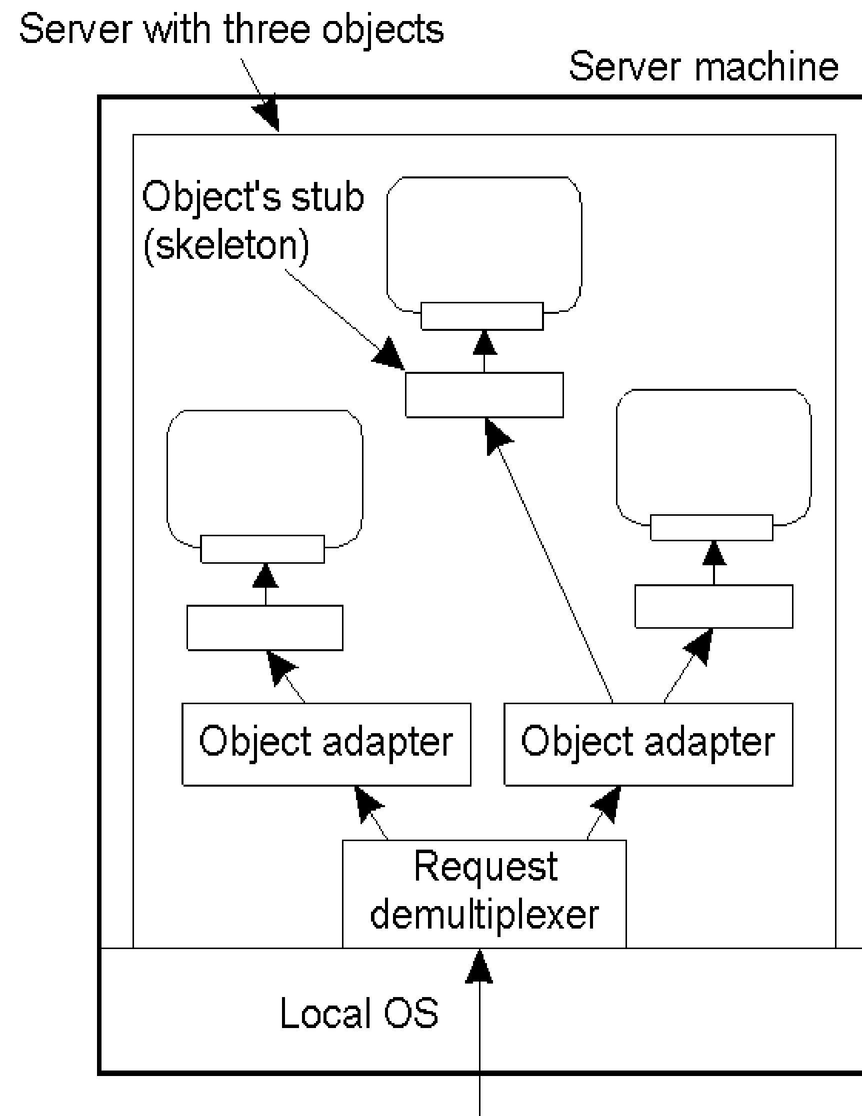
Object Adapters

Decisions on how to invoke an object are referred to as activation policies.

A mechanism to group objects per policy is needed. This mechanism is called object adapter.

An object adapter is software implementing a specific policy.

Object adapters come as generic components to assist developers of distributed objects, and which need only to be configured for a specific policy.



Organization of an object server supporting different activation policies.

Object Adapters - Example

- The adapter manages a number of objects.
- Implemented Policy: Have a single thread of control for each of its objects.
- It expects that each skeleton supports the operation:

  invoke(unsigned in_size, char in_args[],
    unsigned *out_size, char * out_args[])

Object Adapter

```
/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
   long  source                  /* senders identity */
   long  object_id;              /* identifier for the requested object */
   long  method_id;              /* identifier for the requested method  */
   unsigned  size;               /* total bytes in list of parameters */
   char  **data;                 /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned, char*, unsigned*, char**);

long register_object (METHOD_CALL call);        /* register an object  */
void unrigester_object (long object)id);        /* unrigester an object */
void invoke_adapter (message *request);         /* call the adapter */
```

The *header.h* file used by the adapter and any program that calls an adapter.

Object Adapter

```
typedef struct thread THREAD;                    /* hidden definition of a thread
*/

thread *CREATE_THREAD (void (*body)(long tid), long thread_id);
/* Create a thread by giving a pointer to a function that defines the actual  */
/* behavior of the thread, along with a thread identifier  */

void get_msg (unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char **data);
/* Calling get_msg blocks the thread until a message has been put into its  */
/* associated buffer.  Putting a message in a thread's buffer is a nonblocking */
/* operation. */
```

- The *thread.h* file used by the adapter for using threads.

Object Adapter

The implementation of the adapter is independent of the objects for which it handles invocations

- The main part of an adapter that implements a thread-per-object policy.

```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS       100
#define NULL              0
#define ANY               −1

METHOD_CALL invoke[MAX_OBJECTS];      /* array of pointers to stubs      */
THREAD *root;                         /* demultiplexer thread            */
THREAD *thread[MAX_OBJECTS];          /* one thread per object           */

void thread_per_object(long object_id) {
    message      *req, *res;          /* request/response message        */
    unsigned     size;                /* size of messages                */
    char         **results;           /* array with all results          */

    while(TRUE) {
        get_msg(&size, (char*) &req); /* block for invocation request    */

        /* Pass request to the appropriate stub. The stub is assumed to     */
        /* allocate memory for storing the results.                         */
        (invoke[object_id]*)(req→size, req→data, &size, results);

        res = malloc(sizeof(message)+size); /* create response message   */
        res→object_id = object_id;      /* identify object                 */
        res→method_id = req.method_id;  /* identify method                 */
        res→size = size;                /* set size of invocation results  */
        memcpy(res→data, results, size); /* copy results into response    */
        put_msg(root, sizeof(res), res); /* append response to buffer     */
        free(req);                      /* free memory of request          */
        free(*results);                 /* free memory of results          */
    }
}

void invoke_adapter(long oid, message *request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

Code Migration

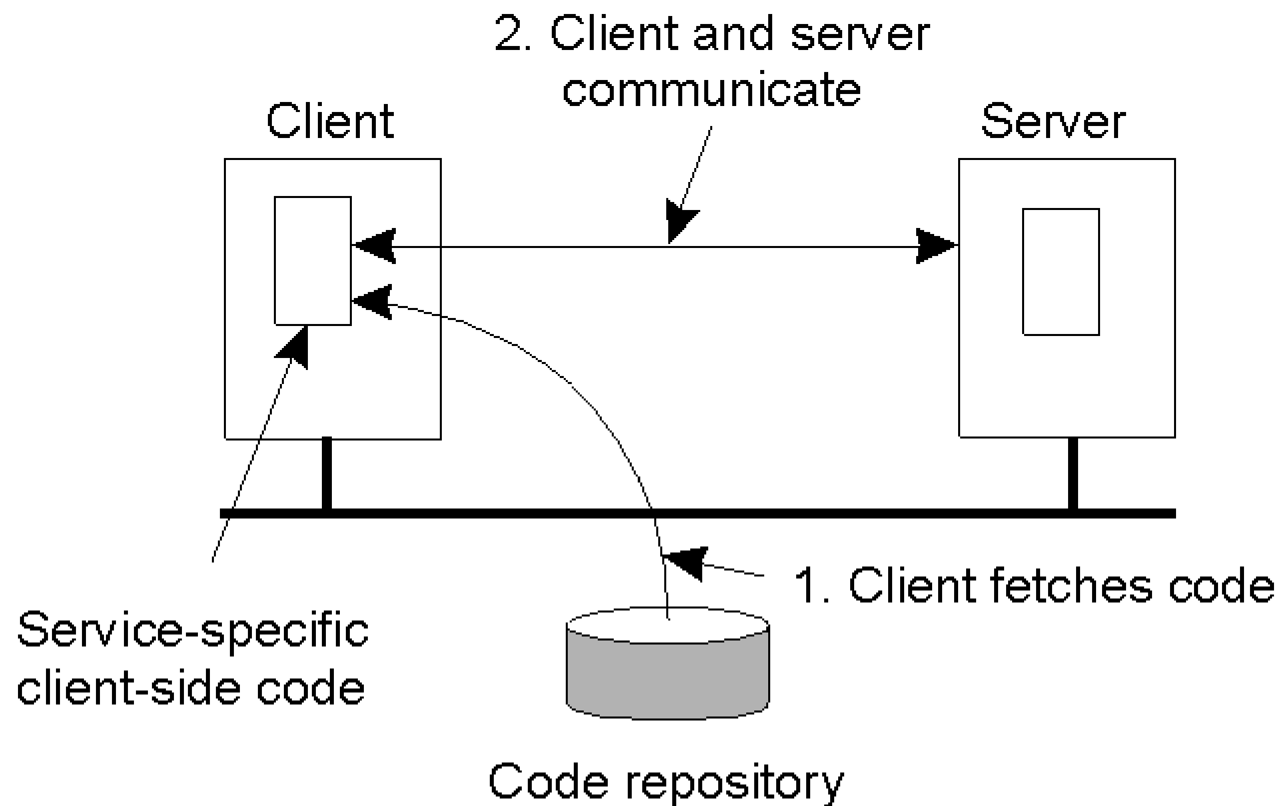
Reasons for Migrating Code

- Load balancing
  - move processes from heavily-loaded to lightly-loaded machines
  - Load-balancing algorithms play an important role in compute-intensive systems
- Minimizing Communication
  - process date close to where those data reside
    - e.g., a server that manages a huge database -> migrate part of the client to server
  - migrate parts of the server to client
- Exploiting parallelism
  - searching for information on the web

Reasons for Migrating Code

Flexibility

- The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.



Models for Code Migration

A process consists of three parts:

- code part
- resource part: contains references to external resources (files, printers, devices, etc.)
- execution part: stores the current execution state of a process (private data, stack, program counter)

Weak Mobility

- Transfer only the code segment + some initialization data
- Example: JAVA applets

Strong Mobility

- transfer the execution part of the process as well

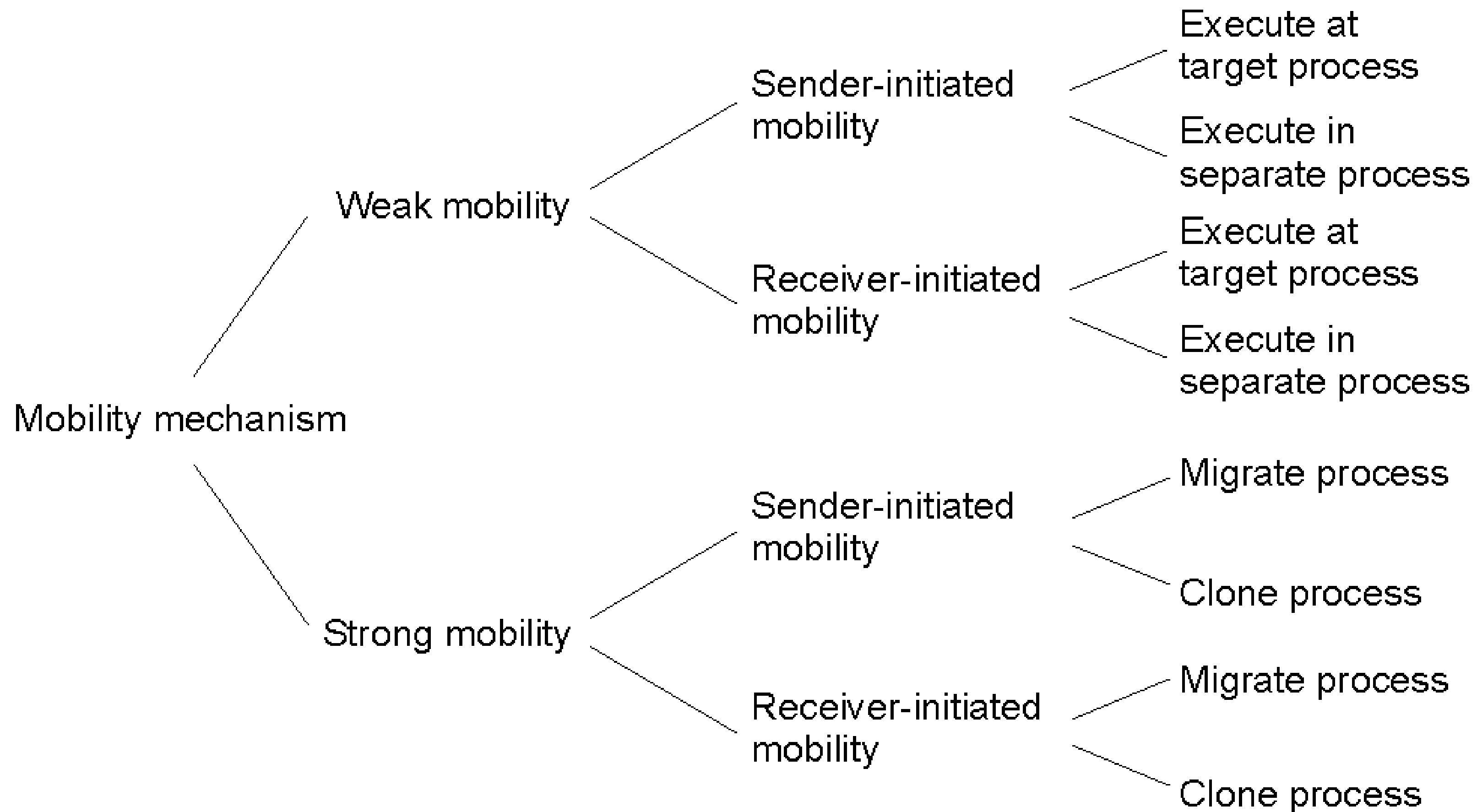
Sender-Initiated

- Examples: upload programs to a compute server, send a search program to a Web server to perform the queries there

Receiver-Initiated

- JAVA applets

Models for Code Migration



Alternatives for code migration.

Migration and Local Resources

Binding by an identifier

- the process requires precisely the referenced resource
- **Example**: use of URLs

Binding by value

- only the value of a resource is needed
- **Example**: use of standard libraries (C or JAVA)

Binding by type

- a process indicates that it needs only a resource of a specific type
- **Example**: references to local devices

Migration and Local Resources

Resource-to-machine Bindings

- Unattached resources
  - can be easily moved between different machines (e.g., data files)
- Fastened resources
  - moving or copying may be possible but only at relatively high costs
  - **Examples**: local databases, complete web sites
- Fixed resources
  - are intimately bound to a specific machine or environment and cannot be moved.
  - **Examples**: local devices

Migration and Local Resources

Resource-to-machine binding

Process-to-resource binding		Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

GR	Establish a global system-wide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally available resource

Actions to be taken with respect to the references to local resources when migrating code to another machine.

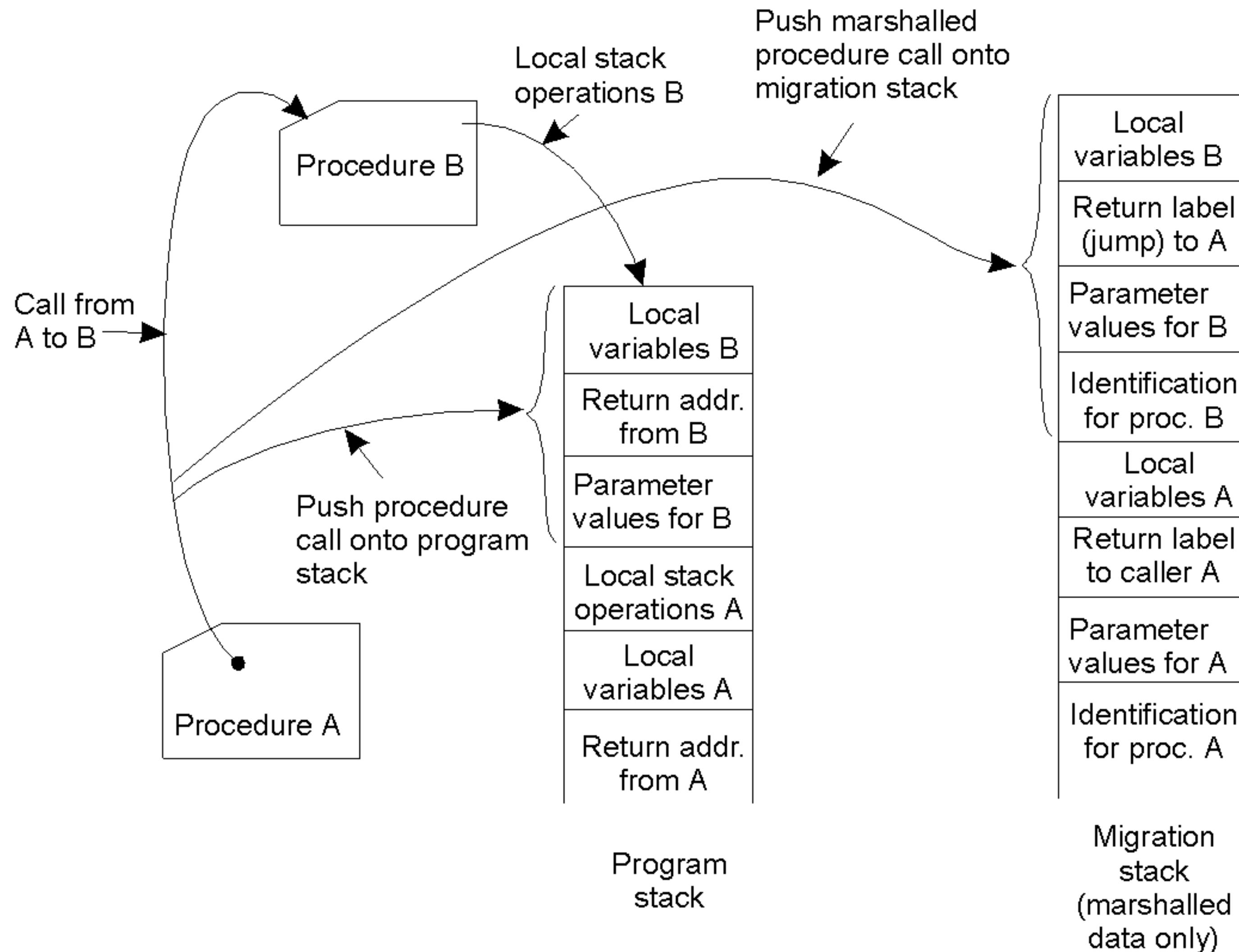
Migration in Heterogeneous Systems

- It should be possible:
  - to execute the code segment on each platform
  - appropriately represent the execution segment at each platform (in case of strong mobility)
- To migrate the execution segment, the target machine should be of the same architecture and run the same operating system.

Solutions

- have the compiler support a migration stack and generate labels allowing the return from a subroutine to be implemented as a (machine-indepenendent) jump.
- Rely on a virtual machine

Migration in Heterogeneous Systems



The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment

Software Agents in Distributed Systems

- A software agent is an autonomous process capable of reacting to, and initiating changes in its environment, possibly in collaboration with users and other agents.

System properties of agents

- A collaborative agent is an agent that forms part of a multi-agent system, in which agents seek to achieve some common goal through collaboration.
- A mobile agent has the capability of moving between different machines.

Classes based on functionality

- Interface agents
  - assist an end user in the use of one or more applications
  - It has learning capabilities
- Information agents
  - manage information from many different sources

Software Agents in Distributed Systems

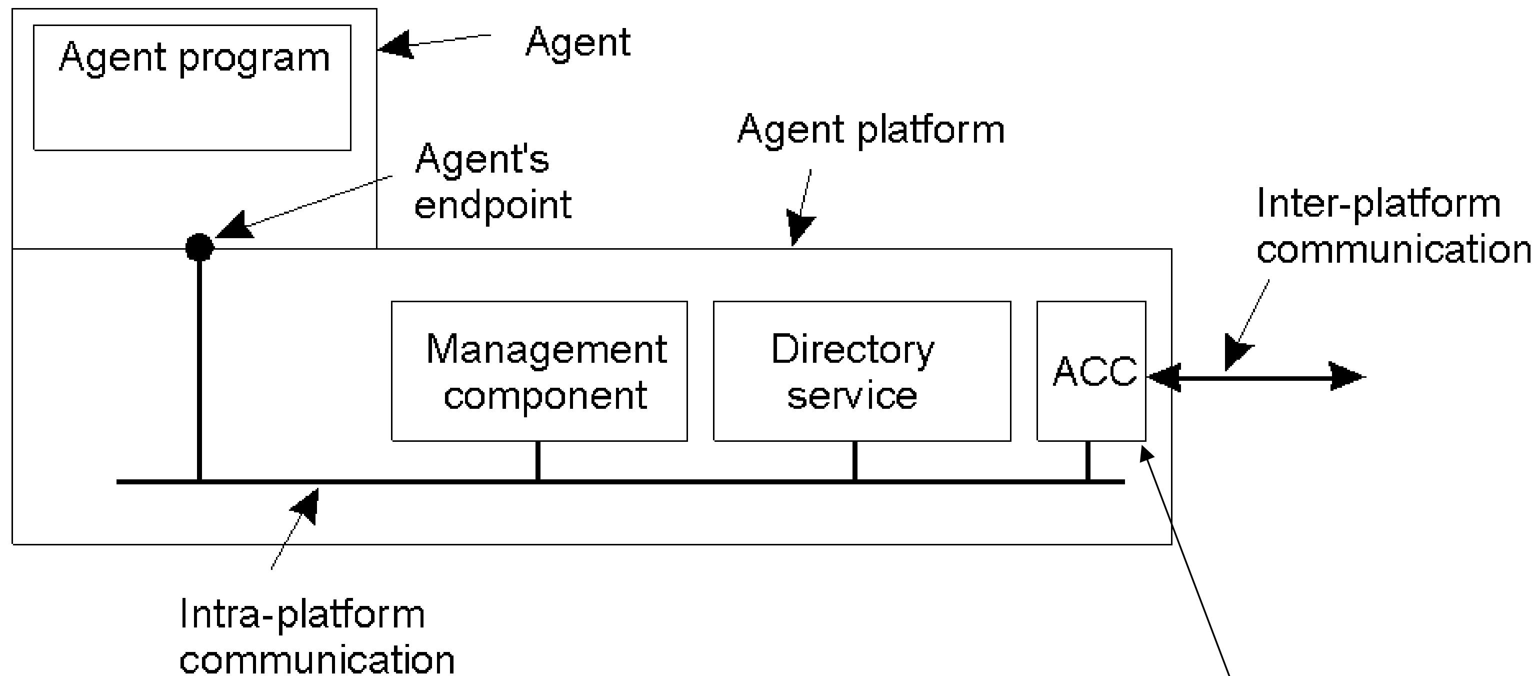
Property	Common to all agents?	Description
Autonomous	Yes	Can act on its own
Reactive	Yes	Responds timely to changes in its environment
Proactive	Yes	Initiates actions that affects its environment
Communicative	Yes	Can exchange information with users and other agents
Continuous	No	Has a relatively long lifespan
Mobile	No	Can migrate from one site to another
Adaptive	No	Capable of learning

- Some important properties by which different types of agents can be distinguished.

Agent Technology

- Foundation for Intelligent Physical Agents (FIPA)
  - developed a general model for software agents
- An agent platform provides the basic services needed for any multiagent system
  - creating and deleting agents
  - facilities to locate agents
  - facilities for inter-agent communication

Agent Technology



The general model of an agent platform

ACC is responsible for reliable and ordered point-to-point communication with other platforms

Agent Communication Languages (ACL)

- The sending and receiving agent have at least the same understanding of the purpose of a message.
- This purpose often determines the reaction of the receiver
- ACL messages consists of a header and the actual content.
  - An ACL message header may contain a field to identify the language or encoding scheme for the content.
  - An additional field may sometimes be included to identify a standardized mapping, called ontology, of symbols to their meaning.

Agent Communication Languages

Message purpose	Description	Message Content
INFORM	Inform that a given proposition is true	Proposition
QUERY-IF	Query whether a given proposition is true	Proposition
QUERY-REF	Query for a give object	Expression
CFP	Ask for a proposal	Proposal specifics
PROPOSE	Provide a proposal	Proposal
ACCEPT-PROPOSAL	Tell that a given proposal is accepted	Proposal ID
REJECT-PROPOSAL	Tell that a given proposal is rejected	Proposal ID
REQUEST	Request that an action be performed	Action specification
SUBSCRIBE	Subscribe to an information source	Reference to source

Examples of different message types in the FIPA ACL, giving the purpose of a message, along with the description of the actual message content.

Agent Communication Languages

Field	Value
Purpose	INFORM
Sender	max@http://fanclub-beatrix.royalty-spotters.nl:7239
Receiver	elke@iiop://royalty-watcher.uk:5623
Language	Prolog
Ontology	genealogy
Content	female(beatrix),parent(beatrix,juliana,bernhard)

A simple example of a FIPA ACL message sent between two agents using Prolog to express genealogy information.