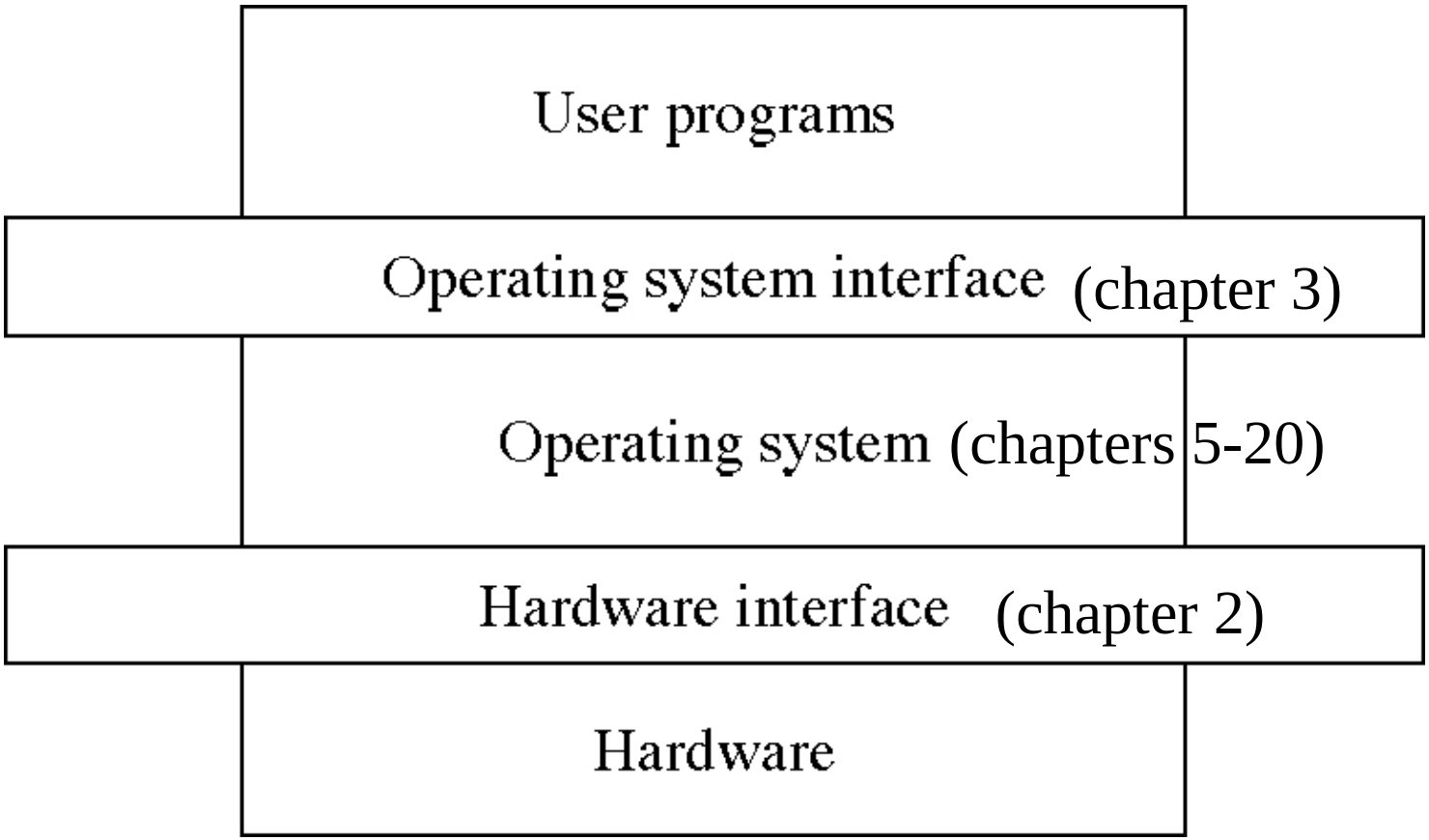
The Operating System Interface

Chapter 3

Key concepts in chapter 3

- System calls
- File and I/O system
  - hierarchical file naming
  - file interface: open, read, write, lseek, close
  - file versus open file
  - devices as files (in naming and in interface)
- Process
  - operations: create, exit, wait
- Shell

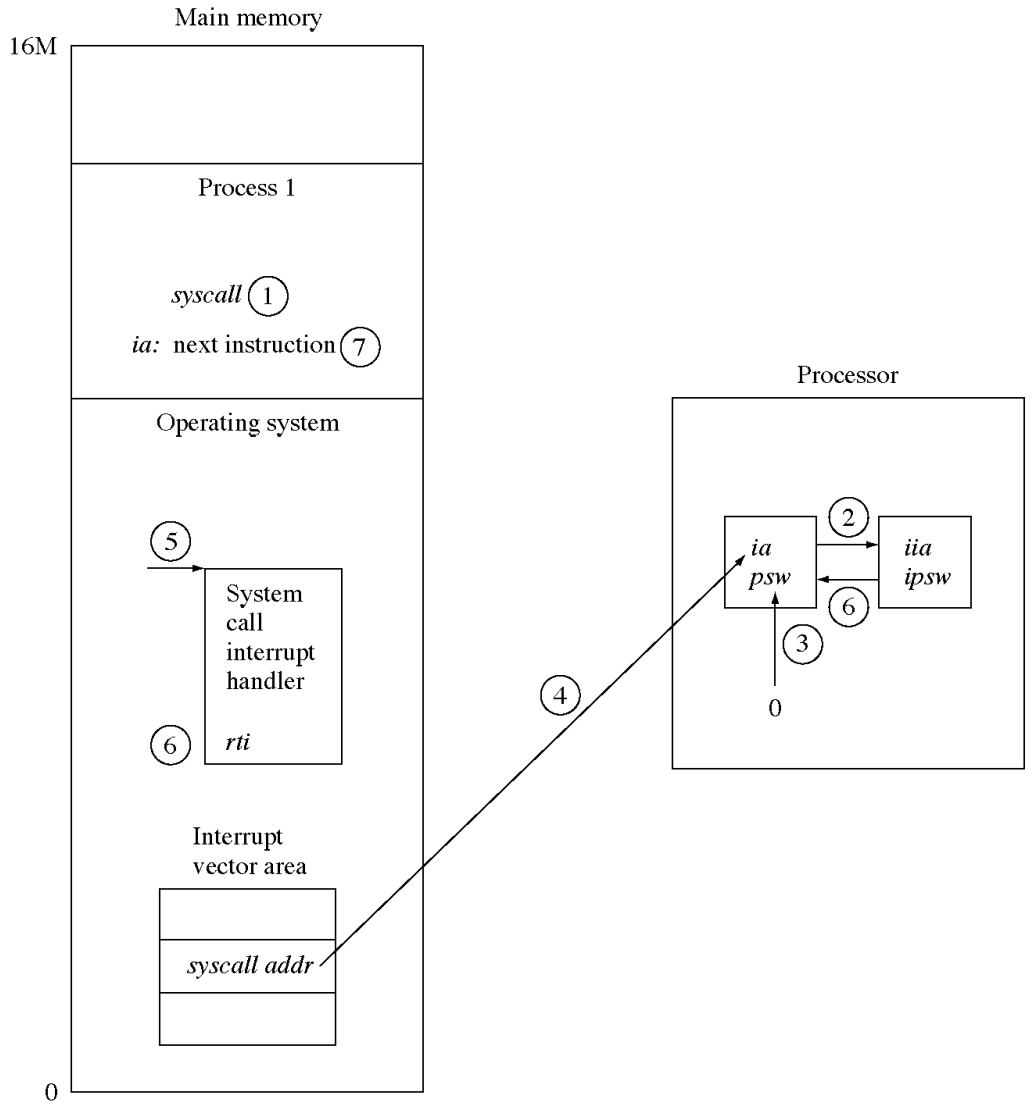
The OS Level Structure



System calls

- A special machine instruction
  - that causes an interrupt
  - various names: syscall, trap, svc
- Usually not generated by HLLs
  - but in assembly language functions

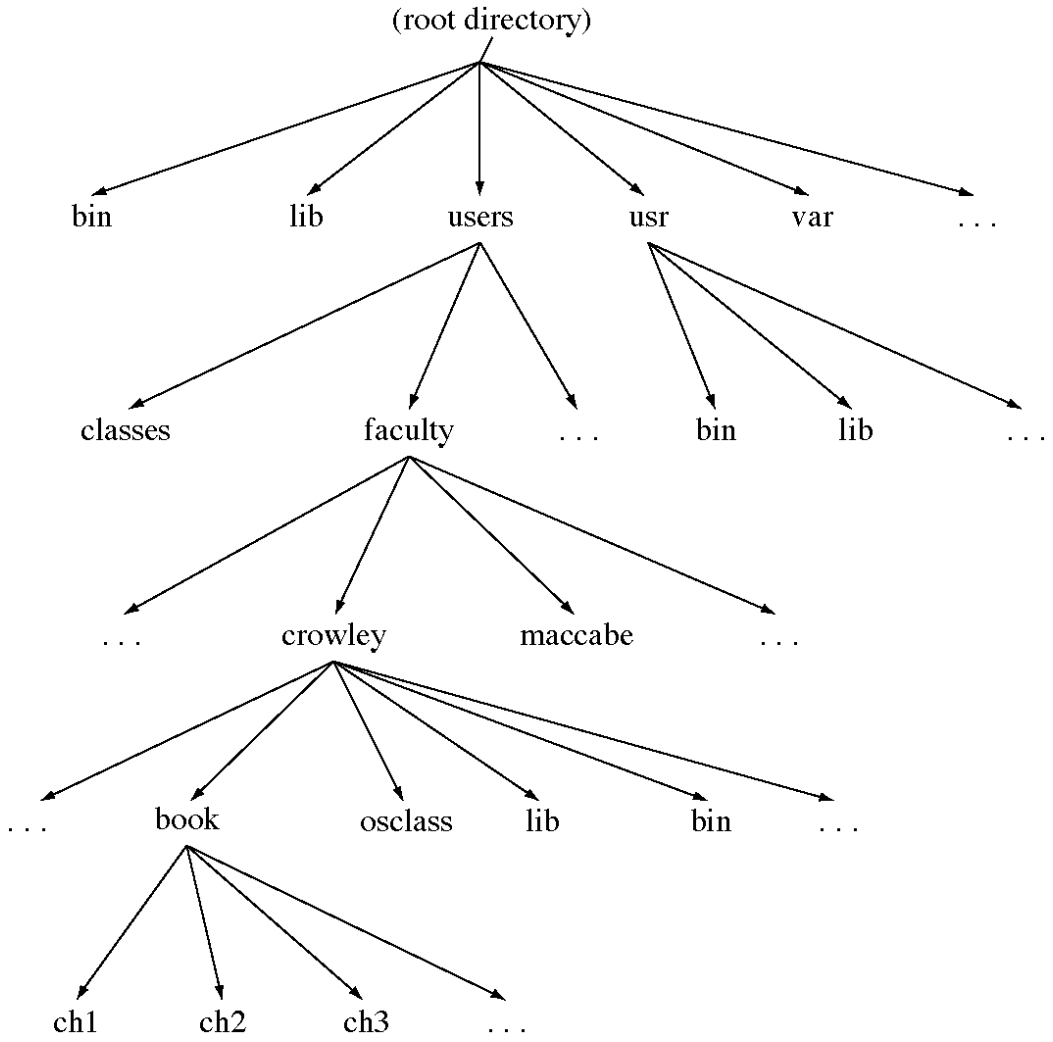
System call flow of control



Hierarchical file naming systems

- A tree of directories and files
  - directory: contains file and directory names
- Objects (files and directories) are named with path names
  - later: other kinds of objects (e.g. devices)
- Path names contains a component name for each directory in the path

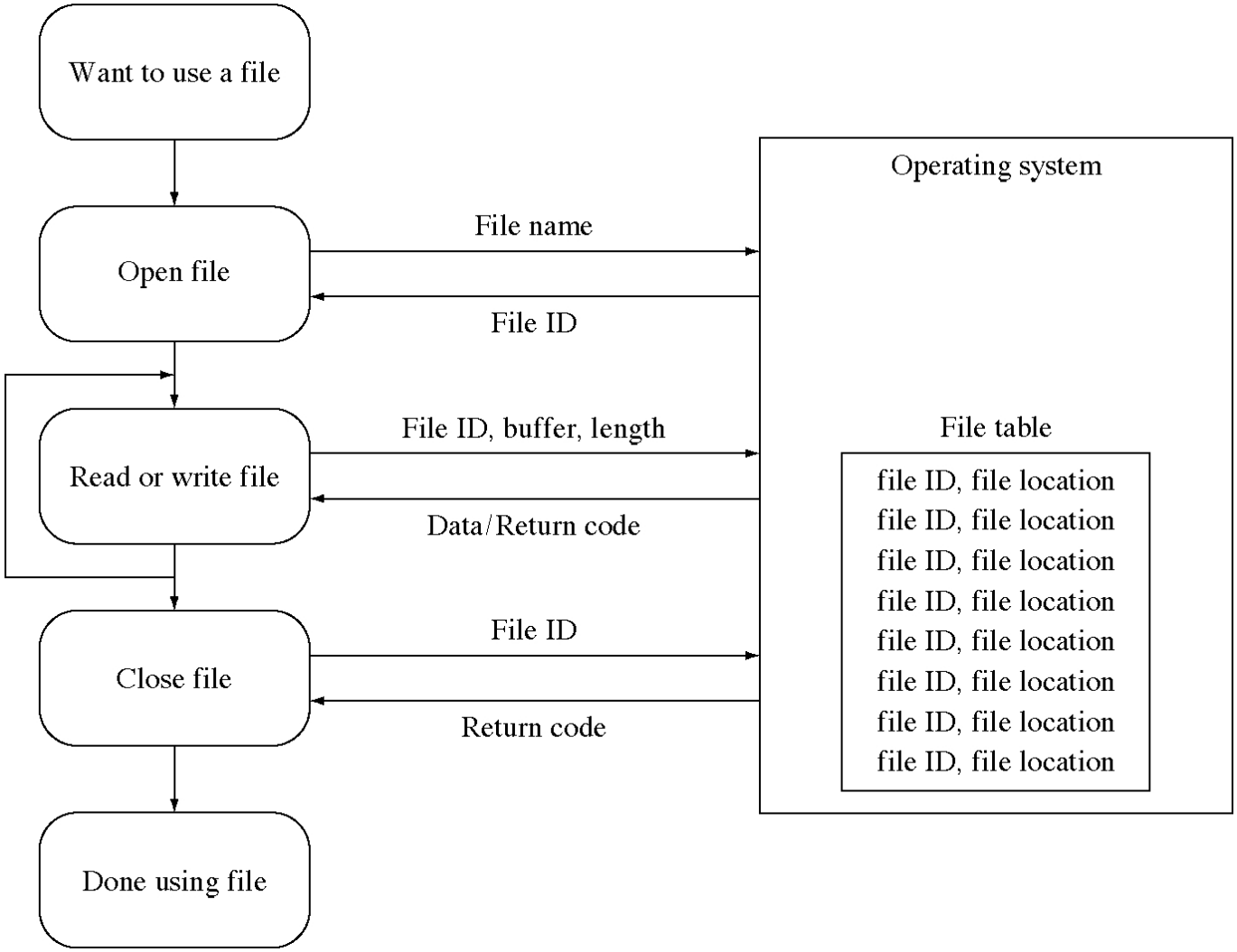
A file naming system



File and I/O system calls

- int open(char *name, int flags)
- int read(int fid, char *buffer, int count)
- int write(int fid, char *buffer, int count)
- int lseek(int fid, int offset, int from)
- int close(int fid)
- int unlink(char *name)

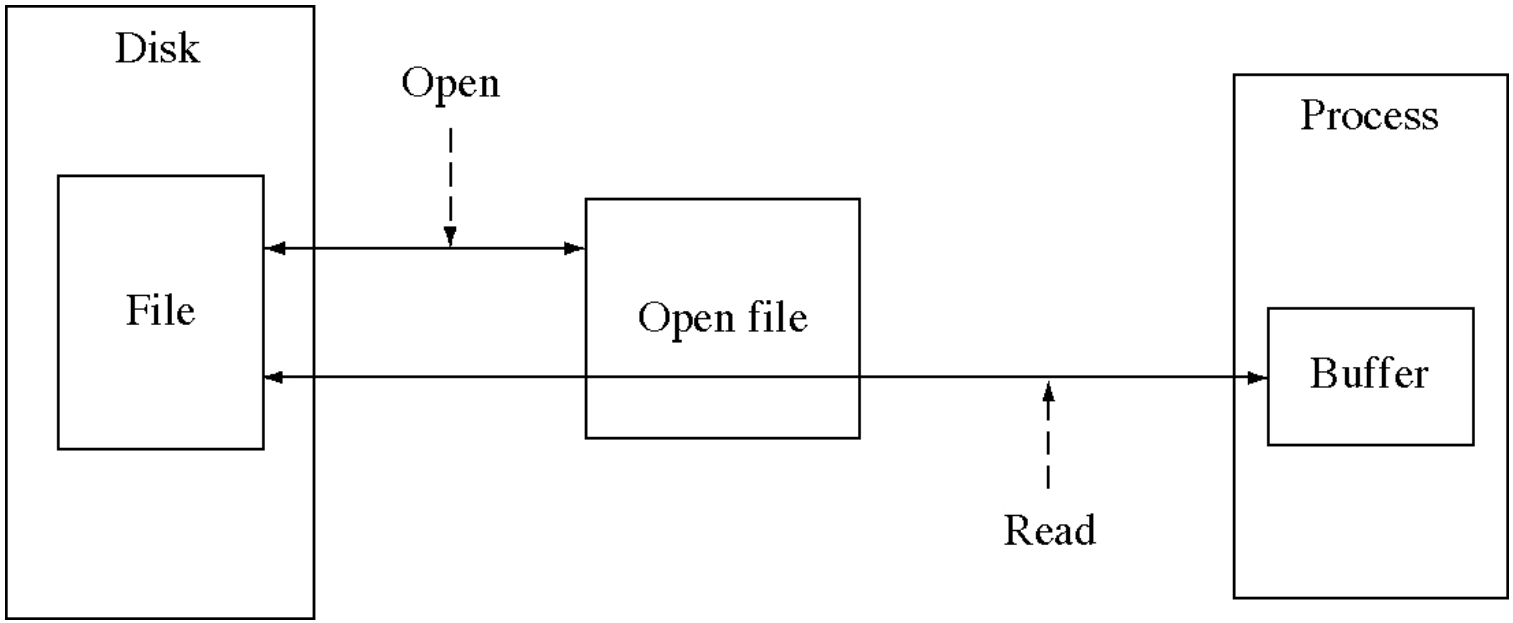
Steps in using a file



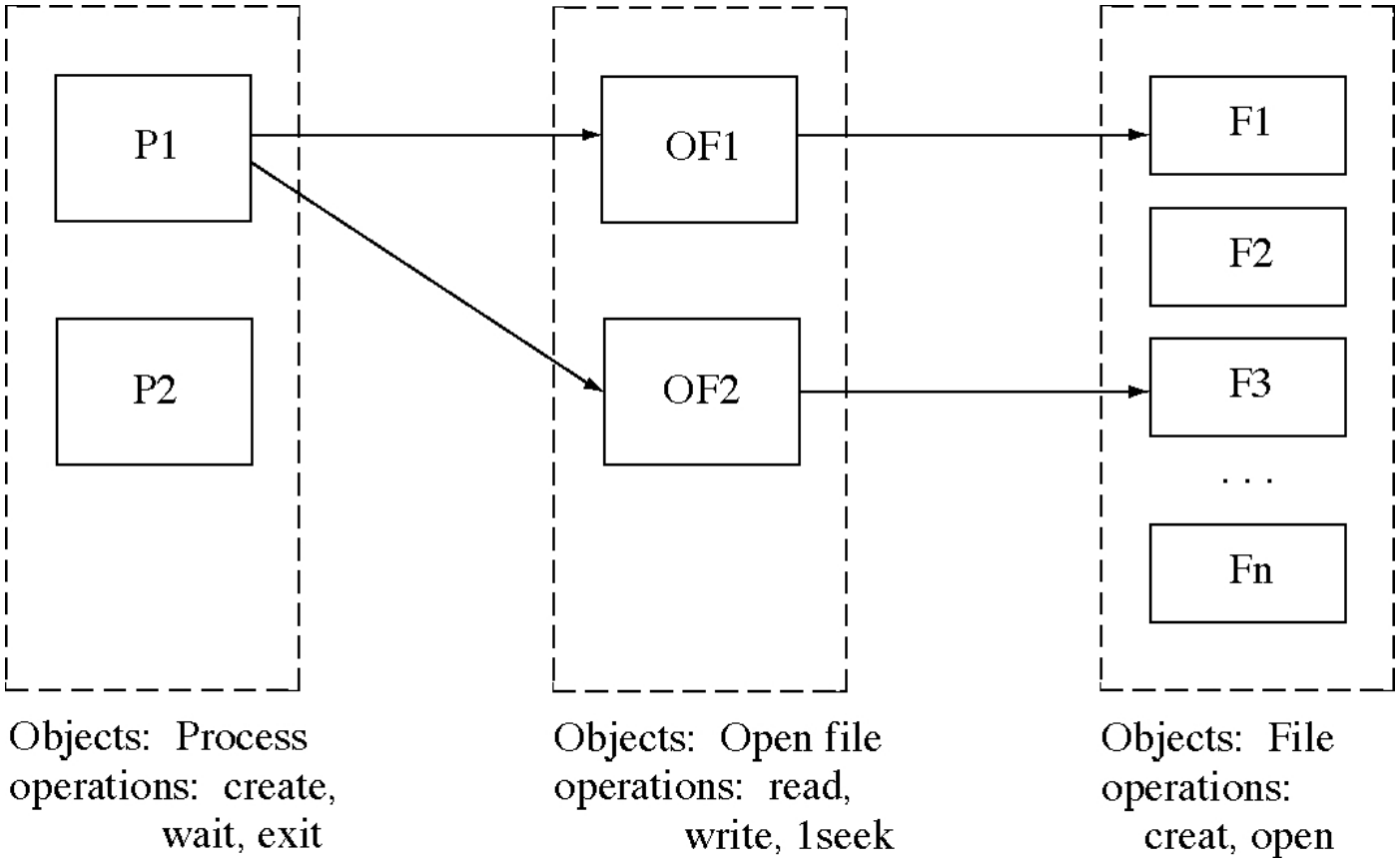
Files versus open files

- *File*: passive container of bytes on disk
- *Open file*: active source (or sink) of bytes in a running program
  - usually connected to a file
  - but can be connected to a device or another process

Files and open files



OS objects and operations



File copy

```
enum { Reading=0, Writing=1, ReadAndWrite=2,
       ReadWriteFile=0644 };
void FileCopy( char * fromFile, char * toFile ) {
  int fromFD = open( fromFile, Reading );
  if( fromFD < 0 ) {
    cerr << "Error opening " << fromFile << endl;
    return; }
  int toFD = creat( toFile, ReadWriteFile );
  if( toFD < 0 ) {
    cerr << "Error opening " << toFile << endl;
    close( fromFD ); return; }
  while( 1 ) {
    char ch; int n = read( fromFD, &ch, 1 );
    if( n <= 0 ) break;
    n = write( toFD, &ch, 1 );
    if( n < 0 ) {
      cerr << "Error writing " << toFile << endl;
      return; }
  }
  close( fromFD ); close( toFD );
}
```

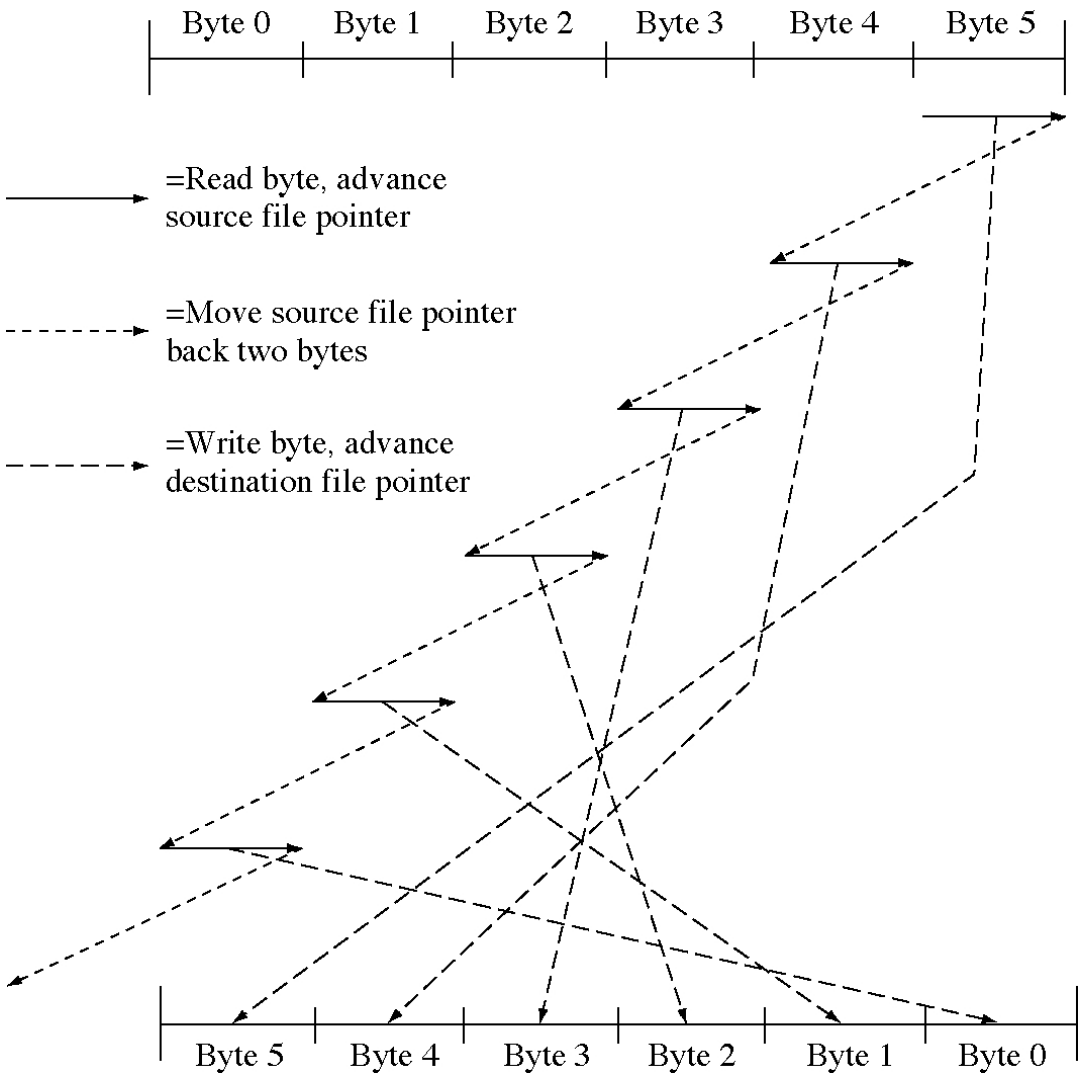
File reverse (1 of 2)

```
enum { Reading=0, Writing=1, ReadAndWrite=2 };
enum {SeekFromBeginning=0,SeekFromCurrent=1,SeekFromEnd=2};
void Reverse( char * fromFile, char * revFile ) {
  int fromFD = open( fromFile, Reading );
  if( fromFD < 0 ) {
    cerr << "Error opening " << fromFile << endl;
    return;
  }
  // move the internal file pointer so the next character
  // read will be the last character of the file
  int ret lseek( fromFD, -1, SeekFromEnd );
  if( ret < 0 ) {
    cerr << "Error seeking on " << fromFile << endl;
    close( fromFD );
    return;
  }
  int revFD = creat( revFile, 0 );
  if( revFD < 0 ) {
    cerr << "Error creating " << revFile << endl;
    close( fromFD );
    return;
  }
```

File reverse (2 of 2)

```
    while( 1 ) {
      char ch;
      int n = read( fromFD, &ch, 1 );
      if( n < 0 ) {
        cerr << "Error reading " << fromFile << endl;
        return;
      }
      n = write( revFD, &ch, 1 );
      if( n < 0 ) {
        cerr << "Error writing " << revFile << endl;
        return;
      }
      // exit the loop if lseek returns an error.
      // The expected error is that the computed offset will
      //   be negative.
      if( lseek(fromFD, -2, SeekFromCurrent) < 0 )
        break;
    }
    close( fromFD );
    close( revFD );
  }
```

Reversing a file



Design technique: Interface design

- There are many different sets of system calls with the same functionality
  - which one is best depends on how they will be used
  - we try to make them easy to use and efficient (minimize the number of system calls necessary to get the job done)
- One should always consider several design alternatives and evaluate them

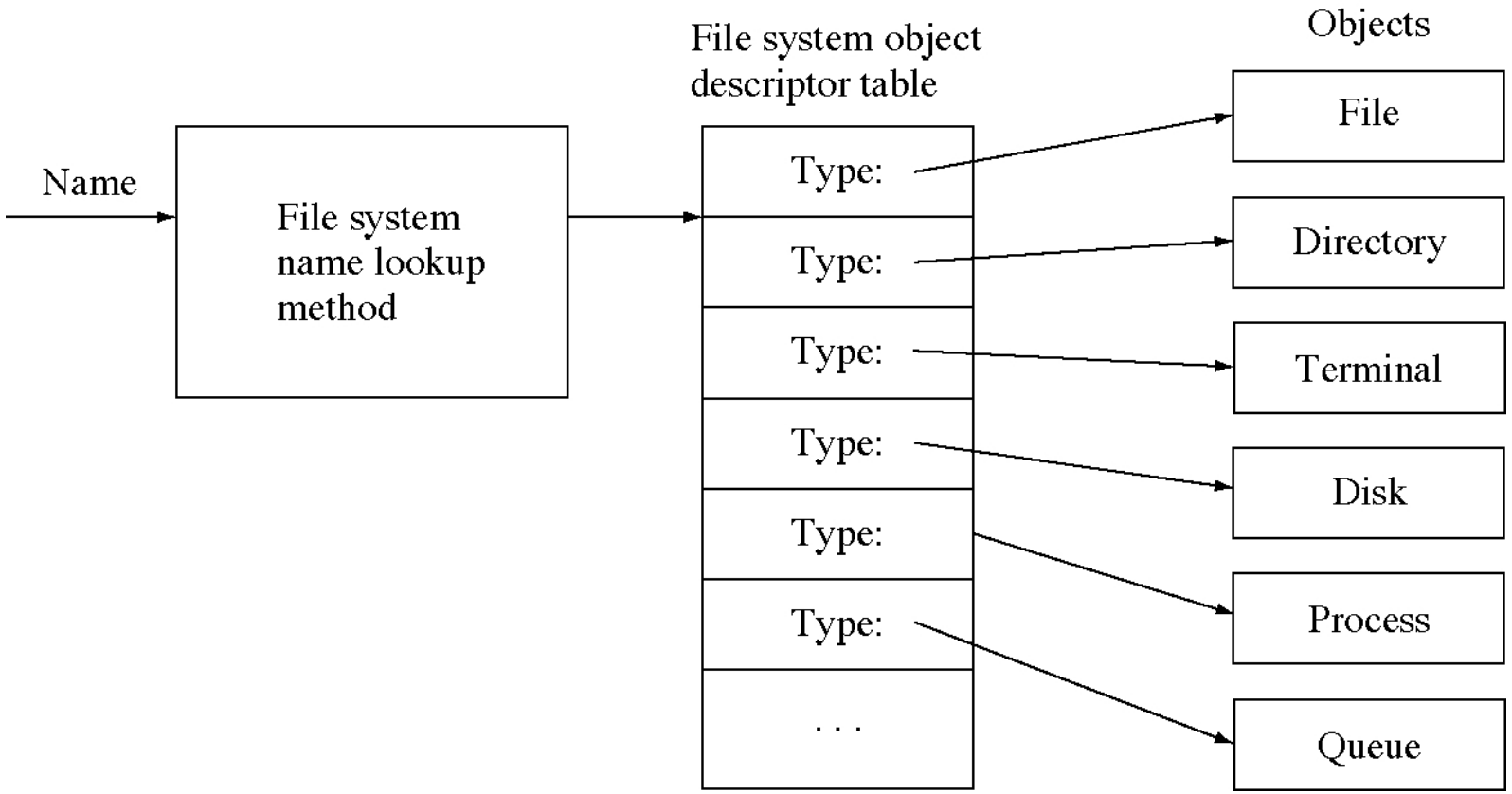
Meta-data

- Meta-data describes the file rather than being the data in file itself
  - also called meta-information
- Examples of meta-data
  - Who owns the file
  - Who can use the file and how
  - When the file was created, last used, last modified
- int stat(char * name, StatInfo *statInfo)
  - this calls returns the file meta-data

Naming OS objects

- File naming system names files (and directories)
  - but why limit it to that
- Other OS objects need names:
  - processes
  - devices
  - IPC: message queues, pipes, semaphores

Mapping names to objects



Devices as files

- Devices are named as files
  - they can be opened as files, to create open files
  - they can be used as byte streams: sources of bytes and sinks for bytes
- Examples
  - copy someFile /dev/tty17
  - copy aFile /dev/tape01

The process concept

- *Program*: a static, algorithmic description, consists of instructions
  -
```
int main() {
   int i, prod=1;
   for(i=0; i<100; ++i)
     prod = prod*i;
}
```
- *Process*: dynamic, consists of instruction executions

Simple create process

```
void CreateProcess1( void ) {
  int pid1 = SimpleCreateProcess( "compiler" );
  if( pid1 < 0 ) {
    cerr << "Could not create process \"compiler\""
         << endl;
    return; }
  int pid2 = SimpleCreateProcess( "editor" );
  if( pid2 < 0 ) {
    cerr << "Could not create process \"editor\""
         << endl;
    return; }
  // Wait until they are both completed.
  SimpleWait( pid1 );
  SimpleWait( pid2 );
  // "compiler" and "editor" also end by making
  //  SimpleExit system calls
  SimpleExit();
}
```

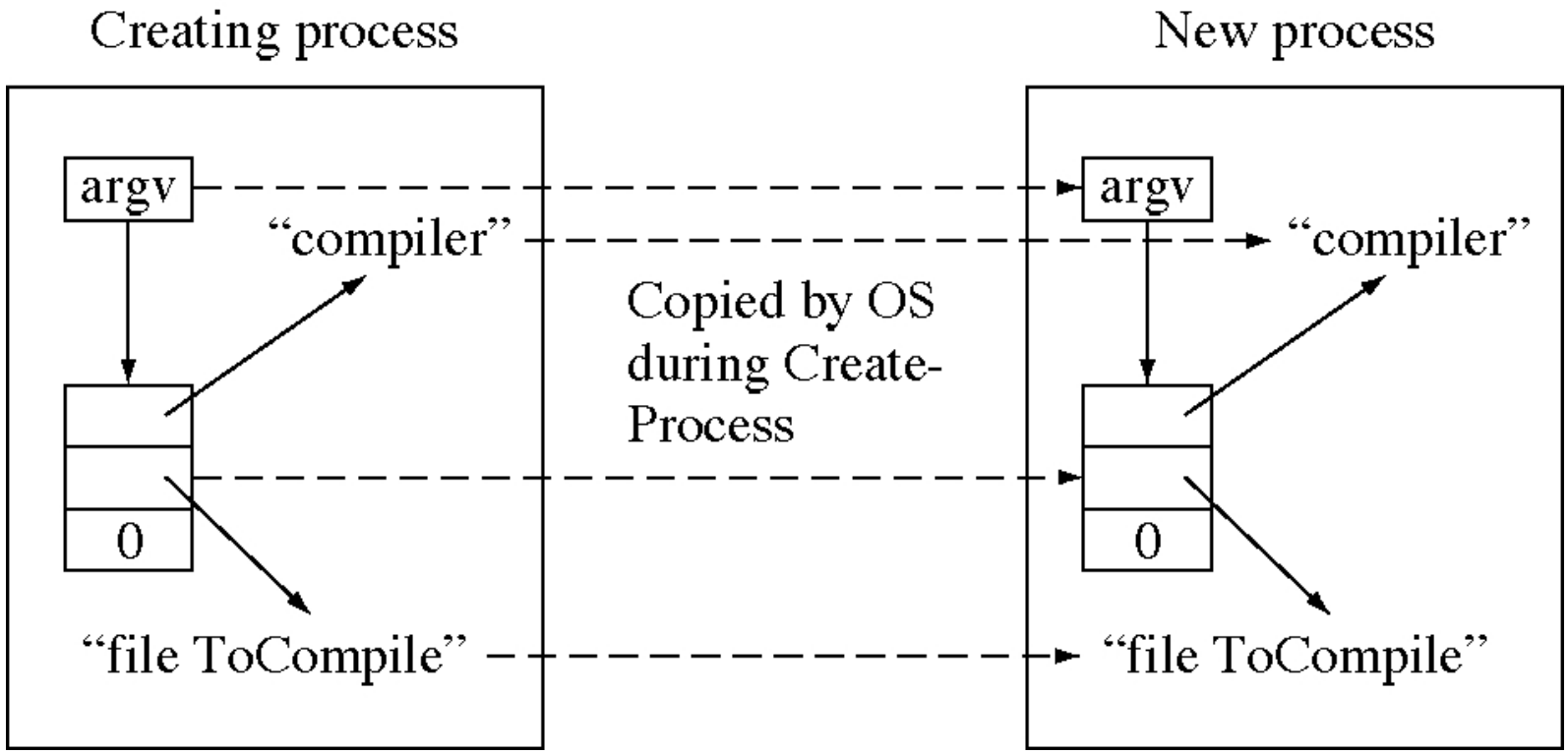
Process system calls

- int CreateProcess(
  char *progName, int argc, char *argv[ ])
  - progName is the program to run in the process
  - returns a process identifier (pid)
- void Exit(int returnCode)
  - exits the process that executes the exit system calls
- int Wait(int pid)
  - waits for a child process to exit

Create process

```
void CreateProcess2( void ) {
  static char * argb[3]
    = { "compiler", "fileToCompile", (char *) 0 };
  int pid1 = CreateProcess( "compiler", 3, argb );
  if( pid1 < 0 ) {
    cerr << "Could not create process \"compiler\"”
         << endl;
    return;
  }
  char * argv[3];
  argv[0] = "editor";
  argv[1] = "fileToEdit";
  argv[2] = (char *) 0;
  int pid2 = CreateProcess( "editor", 3, argv );
  if( pid2 < 0 ) {
    cerr << "Could not create process \"compiler\"”
         << endl;
    return;
  }
  (void) Wait( pid1 );
  (void) Wait( pid2 );
  Exit( 0 );
}
```

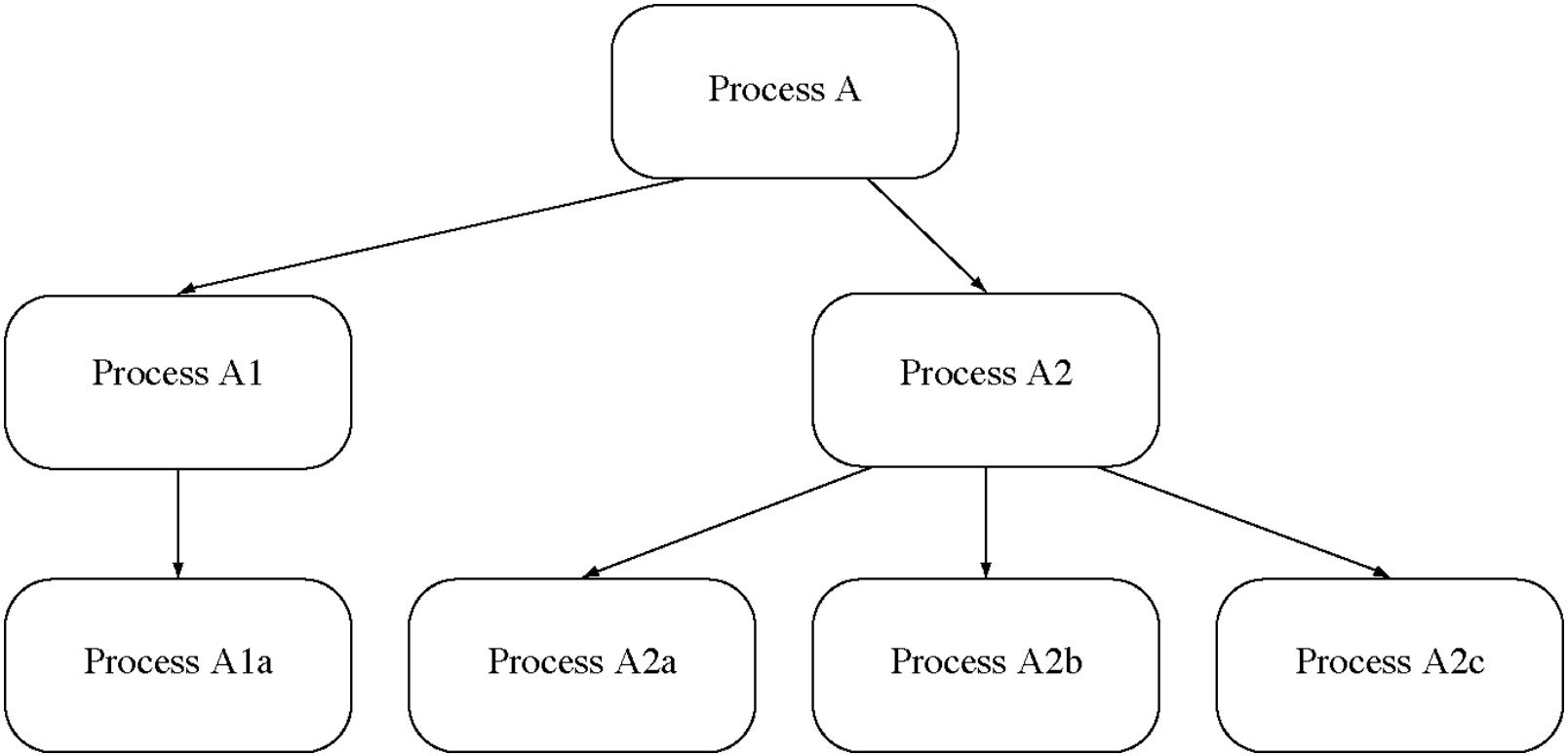
How arguments are passed



Print arguments

```
// This program writes out it arguments.
#include <iostream.h>
void main( int argc, char * argv[ ] ) {
  int i;
  for( i = 0; i < argc; ++i ) {
    cout << argv[i] << " ";
  }
  cout << "\n";
}
```

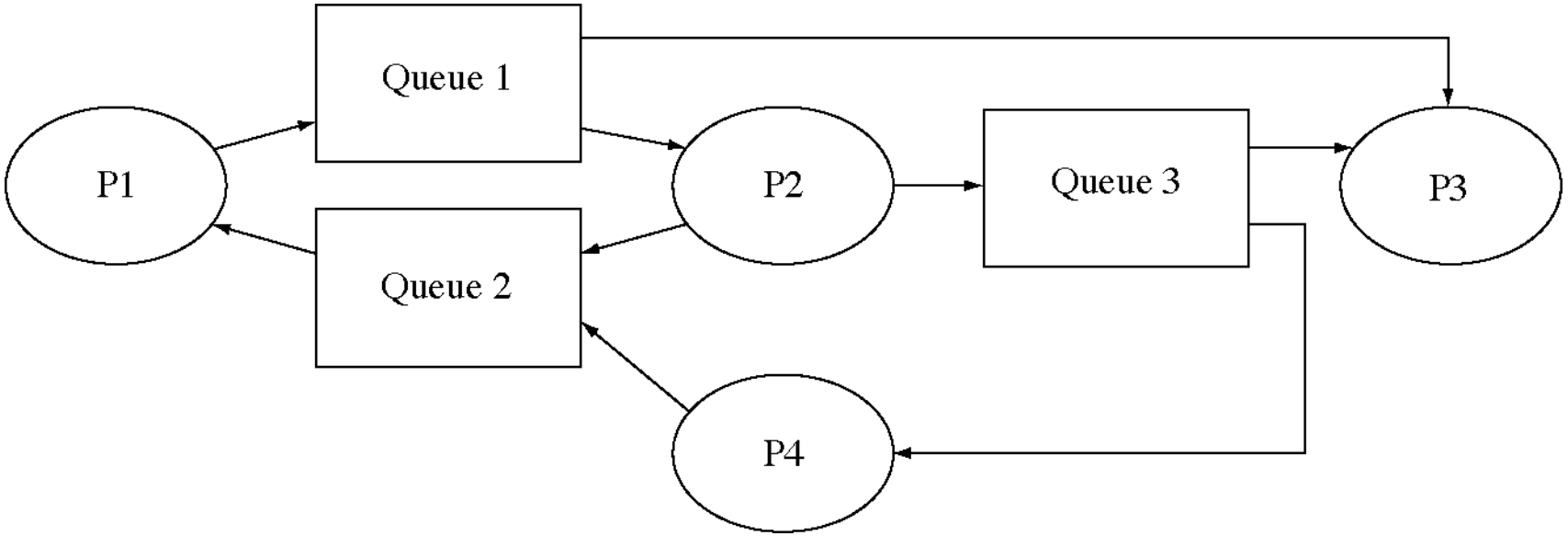
A process hierarchy



Interprocess communication (IPC)

- Many methods have been used: messages, pipes, sockets, remote procedure call, etc.
- Messages and message queues:
  - The most common method
  - Send messages to message queues
  - Receive messages from message queues

Example of message passing paths



Message passing system calls

- int CreateMessageQueue(void)
  - returns a message queue identifier (mqid)
- int SendMessage(int mqid, int *msg)
  - send to a message queue (no waiting)
- int ReceiveMessage(int mqid, int *msg)
  - receive from a message queue
  - wait for a message if the queue is empty
- int DestroyMessageQueue(int mqid)

Message: file sender (1 of 2)

```
void SendMsgTo( int msg_q_id, int msg0=0, int
msg1=0, int msg2=0 ) {
  int msg[8];
  msg[0] = msg0; msg[1] = msg1; msg[2] = msg2;
  (void)SendMessage( msg_q_id, msg );
}
enum { Reading=0, Writing=1, ReadAndWrite=2 };
enum{ FileToOpen=1, SendQueue=2, ReceiveQueue=3 };
void main( int argc, char * argv[ ] ) {
  int fromFD = open( argv[FileToOpen], Reading );
  if( fromFD < 0 ) {
    cerr << "Could not open file ”
         << argv[FileToOpen] << endl;
    exit( 1 );
  }
  int to_q = atoi(argv[SendQueue]);
```

Message: file sender (2 of 2)

```
  while( 1 ) {
    char ch;
    int n = read( fromFD, &ch, 1 );
    if( n <= 0 )
      break;
    SendMsgTo( to_q, ch );
  }
  close( fromFD );
  SendMsgTo( to_q, 0 );
  int msg[8];
  int from_q = atoi(argv[ReceiveQueue]);
  ReceiveMessage( from_q, msg );
  cout << msg[0] << " characters\n";
  exit( 0 );
}
```

Message: file receiver

```
enum{ SendQueue=1, ReceiveQueue=2 };
void main( int argc, char * argv[ ] ) {
  // start the count at zero.
  int count = 0;
  int msg[8];
  int from_q = atoi(argv[SendQueue]);
  while( 1 ) {
    ReceiveMessage( from_q, msg );
    if( msg[0] == 0 )
      break;
    // Any message with nonzero content
    //   is a character to count.
    ++count;
  }
  // Send the count back to the sender.
  int to_q = atoi(argv[ReceiveQueue]);
  (void) SendMsgTo( to_q, count );
  exit( 0 );
}
```

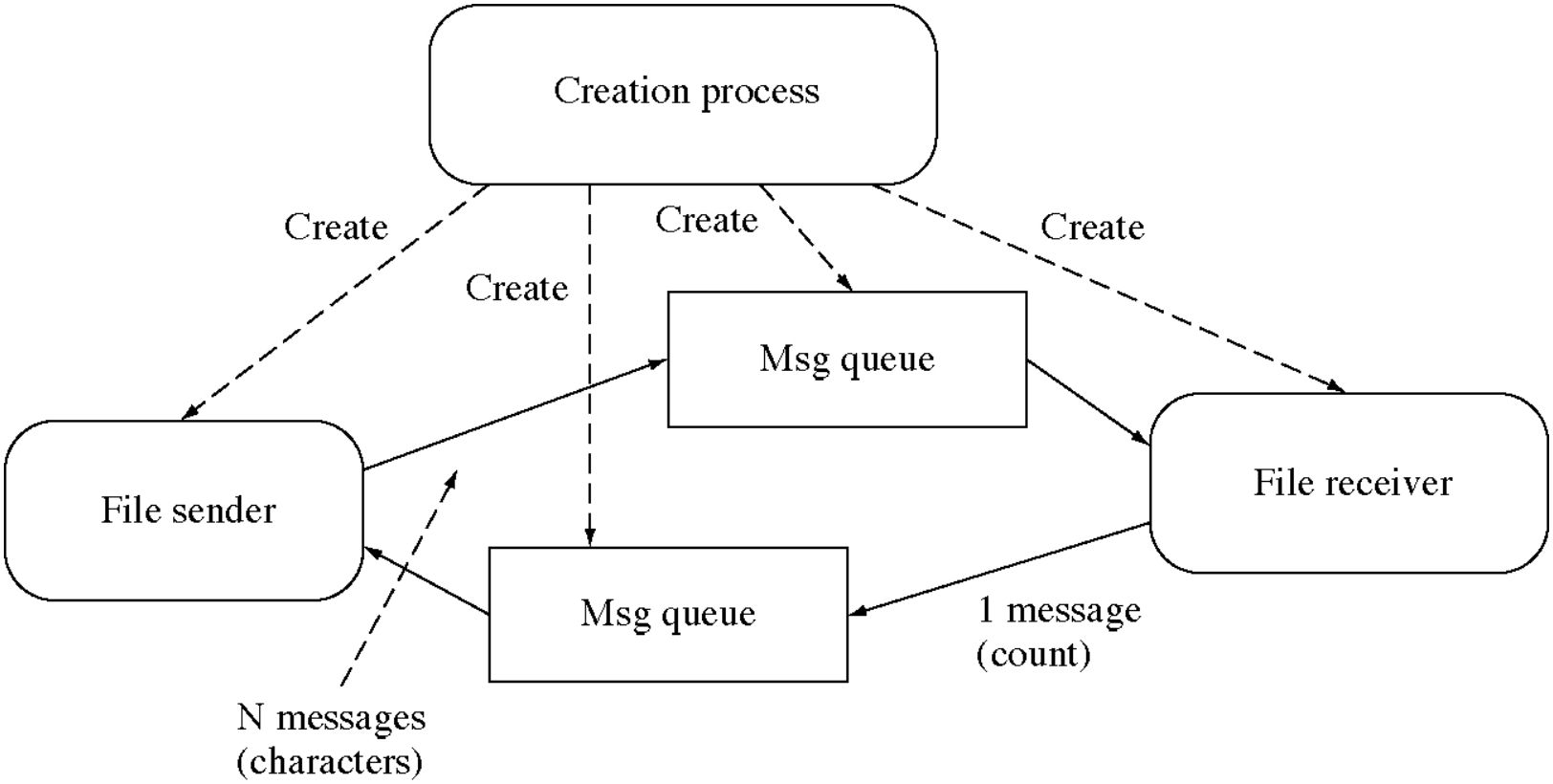
Message: start processes (1 of 2)

- ```
  int CreateProcessWithArgs(char * prog_name,
      char * arg1=0, char * arg2=0, char * arg3=0) {
    char *args[5];
    args[0] = prog_name;
    args[1] = arg1;
    args[2] = arg2;
    args[3] = arg3;
    args[4] = 0;
    int argc = 4;
    if( arg3 == 0) --argc;
    if( arg2 == 0) --argc;
    if( arg1 == 0) --argc;
    return CreateProcess( prog_name, argc, args );
  }
  char * itoa( int n ) {
    char * result = new char[8];
    sprintf( result, "%d", n );
    return result;
  }
  ```
  ```

Message: start processes (2 of 2)

```
void main( int argc, char * argv[ ] ) {
  //Create the message queues the processes will
use.
  int q1 = CreateMessageQueue();
  int q2 = CreateMessageQueue();
  // Create the two processes, sending each the
  // identifier for the message queues it will use.
  int pid1 = CreateProcessWithArgs( "FileSend",
    "FileToSend", itoa(q1), itoa(q2) );
  int pid2 = CreateProcessWithArgs( "FileReceive",
    itoa(q1), itoa(q2) );
  // Wait for the two processes to complete.
  int ret1 = wait( pid1 );
  int ret2 = wait( pid2 );
  // We do not use the return code ret1 and ret2
  //   in this example.
  // Destroy the message queues.
  DestroyMessageQueue( q1 );
  DestroyMessageQueue( q2 );
  Exit( 0 );
```

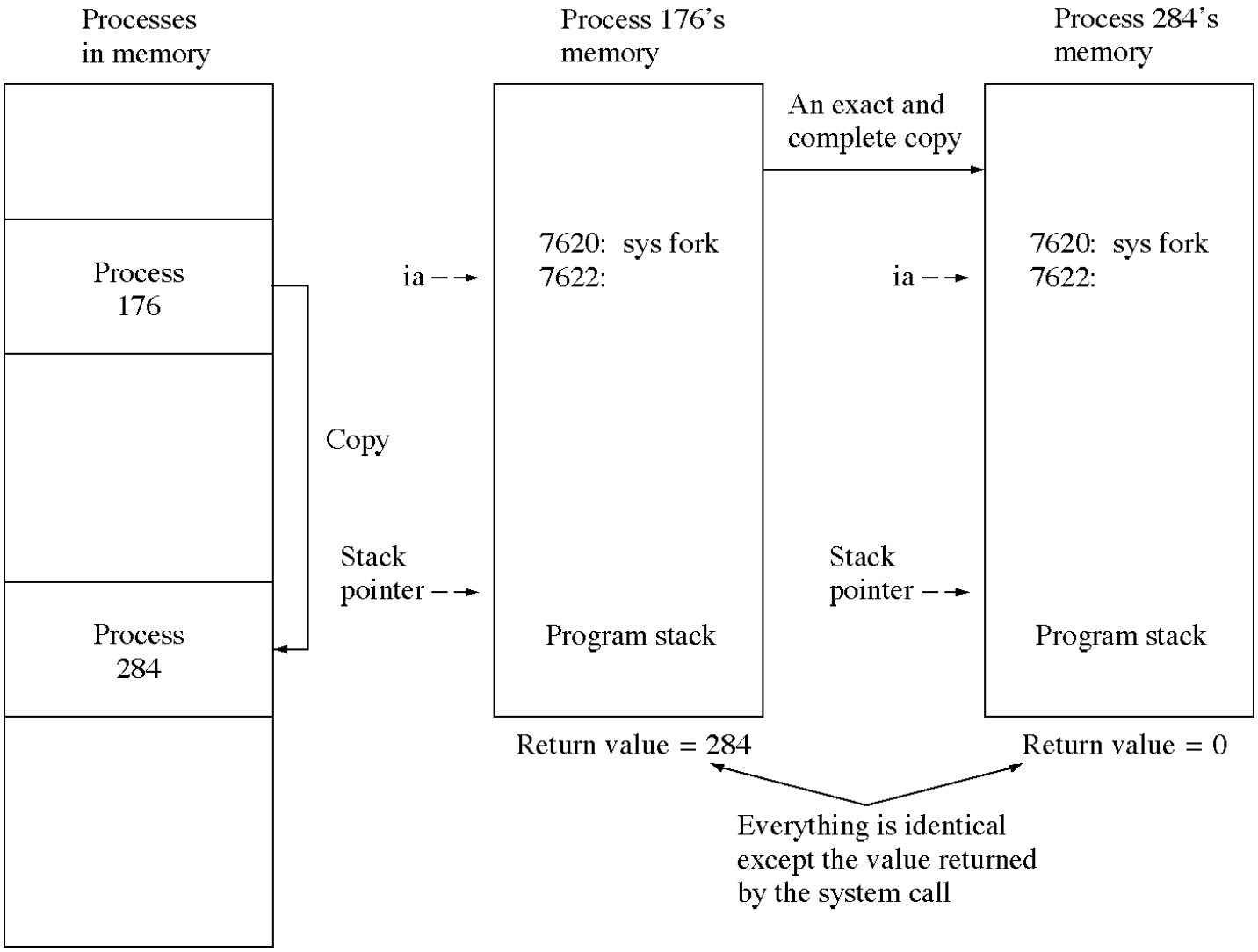
Objects for sending a file with messages



UNIX-style process creation

- int fork()
  - creates an exact copy of the calling process
- int execv(char *progName, char *argv[ ])
  - runs a new program in the calling process
  - destroying the old program
- int exit(int retCode)
  - exits the calling process
- int wait(int *retCode)
  - waits for any exited child, returns its pid

UNIX fork


Processes in memory
Process 176's memory
Process 284's memory
An exact and complete copy
Process 176
Process 284
ia
7620: sys fork
7622:
Copy
Stack pointer
Program stack
Return value = 284
Return value = 0
Everything is identical except the value returned by the system call

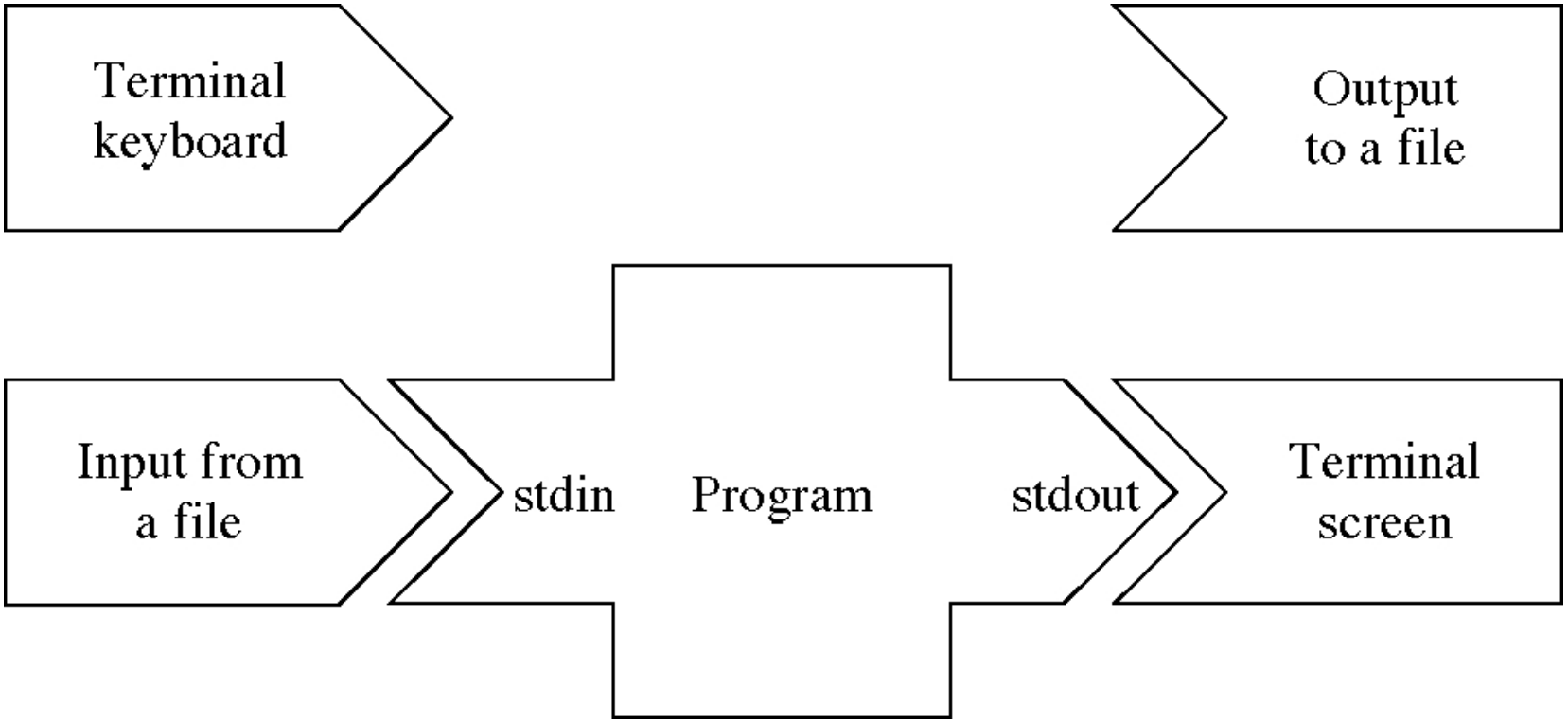
Create process (UNIX-style)

```
void CreateProcess3( void ) {
  int pid1, pid2;
  char *argv[3] = {"compiler", "fileToCompile", 0};
  pid1 = fork();
  if( pid1 == 0 ) {// Child process code begins here
    execv( "compiler", argv ); // execute compiler
    // Child process code ends here.
    // execv does not return
  }
  // Parent executes here because pid1 != 0
  argv[0] = "editor";
  argv[1] = "fileToEdit";
  argv[2] = 0;
  if( (pid2 = fork()) == 0 )
    execv( "editor", argv );
  int reta, retb;
  int pida = wait( &reta );
  int pidb = wait( &retb );
}
```

Standard input and output

- Most programs are filters:
  - one input stream (standard input)
  - some processing
  - one output stream (standard output)
- So the OS starts a program out with two open files, standard input and standard output
- `grep helvetica <fontList >helvList`

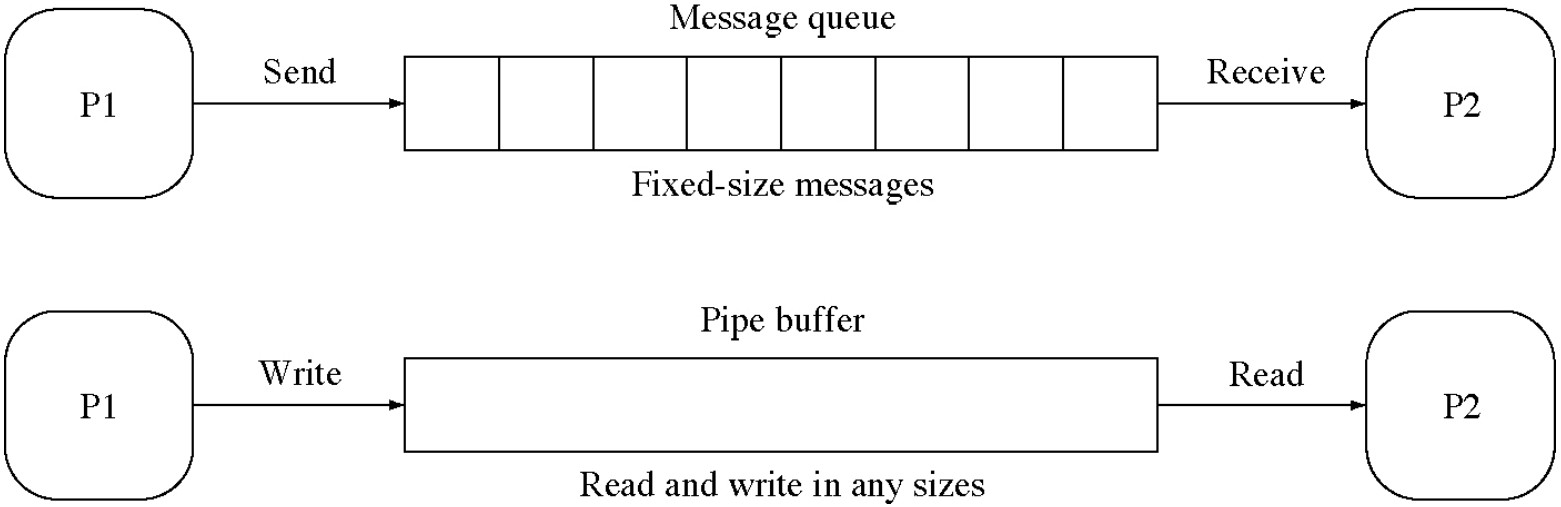
Redirection of standard input and output



Pipes

- Pipe: another IPC mechanism
  - uses the familiar file interface
  - not a special interface (like messages)
- Connects an open file of one process to an open file of another process
  - Often used to connect the standard output of one process to the standard input of another process

Messages and pipes compared



Pipe: file sender

```
enum { Reading=0, Writing=1, ReadAndWrite=2 };
void main( int argc, char * argv[ ] ) {
  int fromFD = open( argv[1], Reading );
  int to_pipe = open( argv[2], Writing );
  while( 1 ) {
    char ch;
    int n = read( fromFD, &ch, 1 );
    if( n == 0 ) break;
    write( to_pipe, &ch, 1 );
  }
  close( fromFD );
  close( to_pipe );
  int n, from_pipe = open( argv[3], Reading );
  // int n is four bytes long, so we read four bytes

  read( from_pipe, &n, 4 );
  close( from_pipe );
  cout << n << " characters\n";
  exit( 0 );
}
```

Pipe: file receiver

```
enum { Reading=0, Writing=1, ReadAndWrite=2 };
void main( int argc, char * argv[ ] ) {
  int count = 0;
  // The first argument is the pipe to read from.
  int from_pipe = open( argv[1], Reading );
  while( 1 ) {
    char ch;
    int n = read( from_pipe, &ch, 1 );
    if( n == 0 )
      break;
    ++count;
  }
  close( from_pipe );
  // send the count back to the sender.
  int to_pipe = open( argv[2], Writing );
  write( to_pipe, &count, 4 );
  close( to_pipe );
  exit( 0 );
}
```

Pipe: create processes

```
void main( int argc, char * argv[ ] ) {
  int pid1 = CreateProcessWithArgs("FileSend",
    "FileToSend", "PipeToReceiver", "PipeToSender");
  int pid2 = CreateProcessWithArgs( "FileReceive",
    "PipeToReceiver", "PipeToSender" );
  int ret1 = wait( pid1 );
  int ret2 = wait( pid2 );
  exit( 0 );
}
```

More on naming

- We have seen three naming systems
  - *Global character names* in the file system: named pipes
  - *Process-local names* (file identifiers): anonymous pipes
  - *Global integer names* picked by the OS: message queues
- We can use any of the systems to name objects

Design technique: Connection in protocols

- File interface is a *connection protocol*
  - open (setup), use, close
  - Best for tightly-coupled, predictable connections
- WWW interface is a *connection-less protocol*
  - Each interaction is independent
  - For loose, unpredictable connections

OS examples

- UNIX (ATT, Bell Labs)
  - Basis of most modern OSes
- Mach (CMU)
  - Microkernel
  - Research system, now widely used
- MS/DOS (Microsoft)
  - Not a full OS

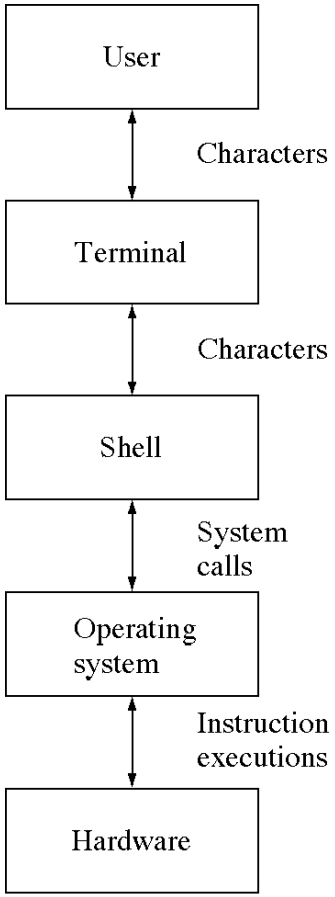
More OS examples

- Windows NT (Microsoft)
  - Successor to MS/DOS
- OS/2 (IBM)
- Macintosh OS (Apple)
  - Innovations in the GUI
  - To be replaced by Rhapsody (Mach)

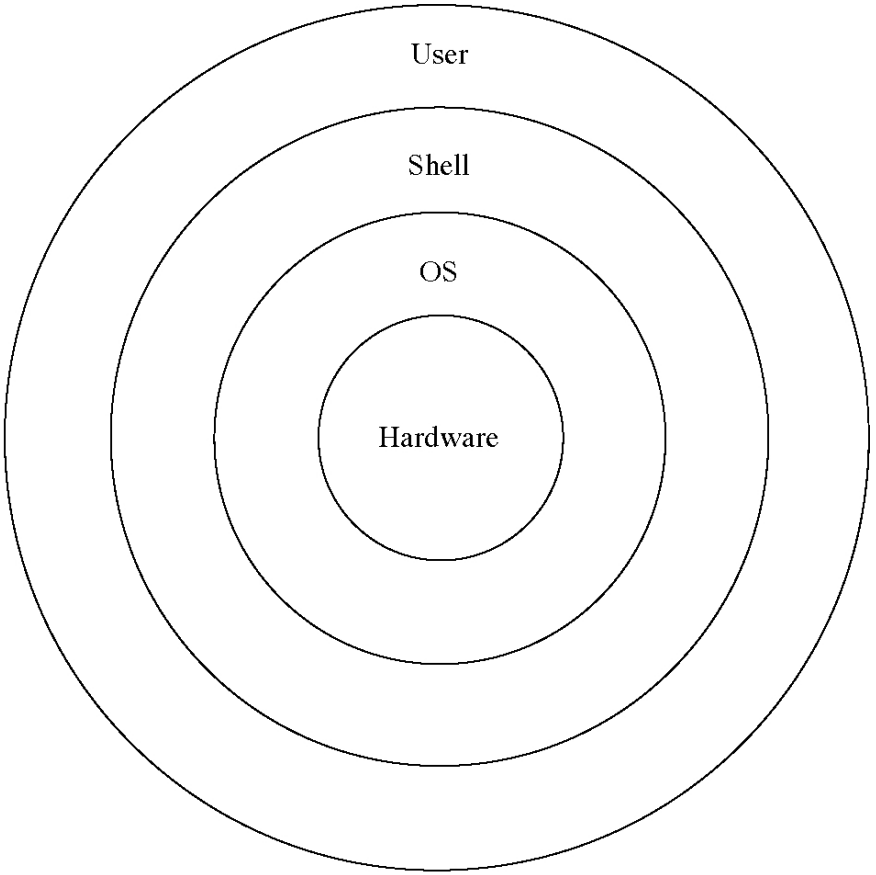
Shell: an OS interface

- Interactive access to the OS system calls
  - copy fromFile toFile
- Contains a simple programming language
- Popularized by UNIX
  - Before UNIX: JCL, OS CLs (command languages)
  - Bourne shell, C shell (csh), Korn shell (ksh), Bourne-again shell (bash), etc.

Two views of a shell



(a) Shell as a level



(b) Shell as a covering

Shell: globals

```
#include <iostream.h>
// some constants
// maximum size of any one argument
const int ARGSIZE 50
// maximum number of arguments
const int NARGS   20

// token types returned by getWord()
const int STRING    1
const int INREDIR   2
const int OUTREDIR  3
const int NEWLINE   4

// define the argv structure
char *argv[NARGS]; // space for argv vector
char args[NARGS][ARGSIZE]; // space for arguments
```

Shell (1 of 3)

```
void main( int argcount, char *arguments[ ] ) {
  int wasInRedir, wasOutRedir;
  char inRedir[ARGSIZE], outRedir[ARGSIZE];
  // each iteration will parse one command
  while( 1 ) {
  // display the prompt
  cout << "@ ";
  // So far we have not seen any redirections
  wasInRedir = 0;
  wasOutRedir = 0;
  // Set up some other variables.
  int argc = 0;
  int done = 0;
  char word[ARGSIZE];
  // Read one line from the user.
  while( !done ) {
    // getWord gets one word from the line.
    int argType = getWord(word);
    // getWord returns the type of the word it read
```

Shell (2 of 3)

```
switch( argType ) {
case INREDIR:
  wasInRedir = 1;
  (void)getWord(inRedir);
  break;
case OUTREDIR:
  wasOutRedir = 1;
  (void)getWord(outRedir);
  break;
case STRING:
  strcpy(args[argc], word);
  argv[argc] = &args[argc][0];
  ++argc;
  break;
case NEWLINE:
  done = 1;
  break;
}
}
argv[argc] = NULL;
if( strcmp(args[0], "logout") == 0 )
  break;
```

Shell (3 of 3)

```
    if( fork() == 0 ) {
      if( wasInRedir ) {
        close(0); // close standard input
        open(inRedir, 0); //reopen as redirect file }
      if( wasOutRedir ) {
        close(1); // close standard output
        enum { UserWrite=0755 };
        creat(outRedir, UserWrite); }
      char cmd[60];
      strcpy(cmd, "./"); strcat(cmd, args[0]);
      execv(cmd, &argv[0]);
      strcpy(cmd, "/bin/"); strcat(cmd, args[0]);
      execv(cmd, &argv[0]);
      cout << "Child: could not exec \"" << args[0]
           << "\"\n";
      exit(1);
    }
    int status; (void) wait(&status);
  }
  cout << "Shell exiting.\n";
}
```

Design technique: Interactive and programming interfaces

- Interactive interfaces have advantages:
  - for exploration
  - for interactive use
- Programming interfaces have advantages :
  - for detailed interactions
  - Inter-application programming
  - Scripting, COM, CORBA
- It is useful for a program to have both