Using the Operating System

The Airplane Pilot’s Abstract Machine
Basic Abstractions

Abstract Machine Entities

- **Process**: A sequential program in execution
- **Resource**: Any abstract resource that a process can request, and which may cause the process to be blocked if the resource is unavailable.
- **File**: A special case of a resource. A linearly-addressed sequence of bytes. “A byte stream.”
Classic Process

- OS implements {abstract machine} – one per task
- *Multiprogramming* enables N programs to be space-muxed in executable memory, and time-muxed across the physical machine processor.
- Result: Have an environment in which there can be multiple programs in execution *concurrently*, each as a processes

* Concurrently: Programs appear to execute simultaneously
Process Abstraction

Processes Sharing a Program
Multithreaded Accountant

(a) Separate Processes
- First Accountant
  - Purchase Orders
- Second Accountant
  - Invoice

(b) Double Threaded Process
- Invoice
- Accountant & Clone
- Purchase Orders

Modern Process & Thread

- Divide classic process:
  - *Process* is an infrastructure in which execution takes place – address space + resources
  - *Thread* is a program in execution within a process context – each thread has its own stack

- Data
- Program
- Process
- Stack
- Operating System
A Process with Multiple Threads

More on Processes

- Abstraction of *processor* resource
  - Programmer sees an *abstract machine environment* with spectrum of resources and a set of resource addresses (most of the addresses are memory addresses)
  - User perspective is that its program is the only one in execution
  - OS perspective is that it runs one program with its resources for a while, then switches to a different process (*context switching*).

- OS maintains
  - A *process descriptor* data structure to implement the process abstraction
    - Identity, owner, things it owns/accesses, etc.
    - Tangible element of a process
  - Resource descriptors for each resource
Address Space

- Process must be able to reference every resource in its abstract machine
- Assign each unit of resource an address
  - Most addresses are for memory locations
  - Abstract device registers
  - Mechanisms to manipulate resources
- Addresses used by one process are inaccessible to other processes
- Say that each process has its own address space

Shared Address Space

- Classic processes sharing program ⇒ shared address space support
- Thread model simplifies the problem
  - All threads in a process implicitly use that process’s address space, but no “unrelated threads” have access to the address space
  - Now trivial for threads to share a program and data
    - If you want sharing, encode your work as threads in a process
    - If you do not want sharing, place threads in separate processes
Creating a Process

• Here is the classic model for creating processes:

  FORK(label) – Create another process in the same address space beginning execution at instruction label
  QUIT() – Terminate the process.
  JOIN(count) – Merge processes into one.

  Equivalent Code:
  ```c
  disableInterrupts();
  count--;
  if(count > 0) QUIT();
  enableInterrupts();
  ```
Example

```
procA() { 
  while(TRUE) { 
    <compute section A1>; 
    update(x); 
    <compute section A2>; 
    retrieve(y); 
  } 
}
```

```
procB() { 
  while(TRUE) { 
    retrieve(x); 
    <compute section B1>; 
    update(y); 
    <compute section B2>; 
  } 
}
```

Example (cont)

```
L0:  count = 2; 
    <compute section A1>; 
    update(x); 
    FORK(L2); 
    <compute section A2>; 
L1:  JOIN(count); 
    retrieve(y); 
    goto L0; 
L2:  retrieve(x); 
    <compute section B1>; 
    update(y); 
    FORK(L3); 
    goto L1; 
L3:  <compute section B2> 
    QUIT(); 
```
Example (cont)

L0:  count = 2;
     <compute section A1>;
     update(x);
     FORK(L2);
     <compute section A2>;
L1:  JOIN(count);
     retrieve(y);
     goto L0;
L2:  retrieve(x);
     <compute section B1>;
     update(y);
     FORK(L3);
     goto L1;
L3:  <compute section B2>
     QUIT();
     L0:  count = 2;
     <compute section A1>;
     update(x);
     FORK(L2);
     retrieve(y);
L1:  <compute section B1>;
     update(y);
     FORK(L3)
     goto L0;
L2:  <compute section A2>;
     goto L1;
L3:  <compute section B2>
     QUIT();

UNIX Processes

UNIX Processes
UNIX Processes

- Each process has its own address space
  - Subdivided into text, data, & stack segment
  - a.out file describes the address space
- OS kernel creates descriptor to manage process
- Process identifier (PID): User handle for the process (descriptor)
- Try “ps” and “ps -aux” (read man page)
Creating/Destroying Processes

- UNIX `fork()` creates a process
  - Creates a new address space
  - Copies text, data, & stack into new address space
  - Provides child with access to open files
- UNIX `wait()` allows a parent to wait for a child to terminate
- UNIX `execve()` allows a child to run a new program

Creating a UNIX Process

```c
int pidValue;
...
pidValue = fork();    /* Creates a child process */
if(pidValue == 0) {
    /* pidValue is 0 for child, nonzero for parent */
    /* The child executes this code concurrently with parent */
    childsPlay(...);  /* A procedure linked into a.out */
    exit(0);
}
/* The parent executes this code concurrently with child */
parentsWork(...);
wait(...);
...```
Child Executes a Different Program

```c
int pid;
...
/* Set up the argv array for the child */
...
/* Create the child */
if((pid = fork()) == 0) {
    /* The child executes its own absolute program */
    execve(childProgram.out, argv, 0);
    /* Only return from an execve call if it fails */
    printf("Error in the exec ... terminating the child ...");
    exit(0);
}
...
wait(...); /* Parent waits for child to terminate */
...```

Example: Parent

```c
#include <sys/wait.h>

#define NULL 0

int main (void)
{
    if (fork() == 0) { /* This is the child process */
        execve("child", NULL, NULL);
        exit(0); /* Should never get here, terminate */
    }
    /* Parent code here */
    printf("Process[%d]: Parent in execution ...
", getpid());
    sleep(2);
    if(wait(NULL) > 0) /* Child terminating */
        printf("Process[%d]: Parent detects terminating child
", getpid());
    printf("Process[%d]: Parent terminating ...
", getpid());
}
Example: Child

```c
int main (void)
{
    /* The child process's new program
       This program replaces the parent's program */

    printf("Process[%d]: child in execution ...\n", getpid());
    sleep(1);
    printf("Process[%d]: child terminating ...\n", getpid());
}
```

Threads -- The NT Model
Windows NT Process

```c
#include <cthreads.h>
...
int main(int argc, char *argv[]) {
    ...
    STARTUPINFO startInfo;
    PROCESS_INFORMATION processInfo;
    ...
    strcpy(lpCommandLine, "C:\WINNT\SYSTEM32\NOTEPAD.EXE temp.txt");
    ZeroMemory(&startInfo, sizeof(startInfo));
    if(!CreateProcess(NULL, lpCommandLine, NULL, NULL, FALSE,
                        HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE,
                        NULL, NULL, &startInfo, &processInfo)) {
        fprintf(stderr, "CreateProcess failed on error %d\n", GetLastError());
        ExitProcess(1);
    }
    /* A new child process is now executing the lpCommandLine program */
    ...
    CloseHandle(processInfo.hThread);
    CloseHandle(processInfo.hProcess);
    t_handle = CreateProcess(..., lpCommandLine, ...);
}
```

NT Threads

```c
#include <cthreads.h>
...
int main(int argc, char *argv[]) {
    t_handle = CreateThread;
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
        // pointer to thread security attributes
    DWORD dwStackSize,
        // initial thread stack size, in bytes
    LPTHREAD_START_ROUTINE lpStartAddress,
        // pointer to thread function
    LPVOID lpParameter,     // argument for new thread
    DWORD dwCreationFlags,  // creation flags
    LPDWORD lpThreadId      // pointer to returned thread identifier
    );
    /* A new child thread is now executing the tChild function */
    Sleep(100)      /* Let another thread execute */
}
```

DWMRDS WINAPI tChild(LPVOID me) {
    /* This function is executed by the child thread */
    ...
    SLEEP(100);    /* Let another thread execute */
    ...
}
_beginthreadex()

- Single copy of certain variables in a process
- Need a copy per thread

```c
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned (*_stdcall *start_address)( void * ),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr
);
```

Resources

- Anything that a process requests from an OS
  - Available ⇒ allocated
  - Not available ⇒ process is blocked
- Examples
  - Files
  - Primary memory address space (“virtual memory”)
  - Actual primary memory (“physical memory”)
  - Devices (e.g., window, mouse, kbd, serial port, …)
  - Network port
  - … many others …
Files

- Data must be read into (and out of) the machine – I/O devices
- Storage devices provide persistent copy
- Need an abstraction to make I/O simple – the file
- A file is a linearly-addressed sequence of bytes
  - From/to an input device
  - Including a storage device

The File Abstraction
**UNIX Files**

- UNIX and NT try to make every resource (except CPU and RAM) look like a file
- Then can use a common interface:

  ```
  open  Specifies file name to be used
  close Release file descriptor
  read  Input a block of information
  write Output a block of information
  lseek Position file for read/write
  ioctl Device-specific operations
  ```

**UNIX File Example**

```c
#include <stdio.h>
#include <fcntl.h>
int main() {
    int inFile, outFile;
    char *inFileName = "in_test";
    char *outFileName = "out_test";
    int len;
    char c;

    inFile = open(inFileName, O_RDONLY);
    outFile = open(outFileName, O_WRONLY);
    /* Loop through the input file */
    while ((len = read(inFile, &c, 1)) > 0) {
        write(outFile, &c, 1);
    }
    /* Close files and quit */
    close(inFile);
    close(outFile);
}
```
Windows File Manipulation Program

#include <windows.h>
#include <stdio.h>
#define BUFFER_LEN ... // # of bytes to read/write
/* The producer process reads information from the file name
   in_test then writes it to the file named out_test.
*/
int main(int argc, char *argv[]) {
    // Local variables
    char buffer[BUFFER_LEN+1];
    // CreateFile parameters
    DWORD dwShareMode = 0;  // share mode
    LPSECURITY_ATTRIBUTES lpFileSecurityAttributes = NULL;
    HANDLE hTemplateFile = NULL;  // handle to file with attributes to copy
    // ReadFile parameters
    HANDLE sourceFile;  // Source of pipeline
    DWORD numberOfBytesRead;  // number of bytes read
    LPOVERLAPPED lpOverlapped = NULL;  // Not used here

    // Open the source file
    sourceFile = CreateFile ("in_test",
        GENERIC_READ,
        dwShareMode,
        lpFileSecurityAttributes,
        OPEN_ALWAYS,
        FILE_ATTRIBUTE_READONLY,
        hTemplateFile
    );
    if(sourceFile == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "File open operation failed\n");
        ExitProcess(1);
    }

    // WriteFile parameters
    HANDLE sinkFile;  // Source of pipeline
    DWORD numberOfBytesWritten;  // # bytes written
    // Open the sink file
    sinkFile = CreateFile ("out_test",
        GENERIC_WRITE,
        dwShareMode,
        lpFileSecurityAttributes,
        OPEN_ALWAYS,
        FILE_ATTRIBUTE_READONLY,
        hTemplateFile
    );
    if(sinkFile == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "File open operation failed\n");
        ExitProcess(1);
    }
    // Read from the source file
    numberOfBytesRead = ReadFile(sourceFile, buffer, BUFFER_LEN, NULL, NULL);
    // Write to the sink file
    if(numberOfBytesRead > 0) {
        numberOfBytesWritten = WriteFile(sinkFile, buffer, numberOfBytesRead, NULL, NULL);
    }
    // Close the files
    if(numberOfBytesWritten != numberOfBytesRead) {
        fprintf(stderr, "Error writing to file\n");
        ExitProcess(1);
    }
    close(sourceFile);
    close(sinkFile);
    return 0;
}
Windows File Manipulation Program(3)

// Open the sink file
sinkFile = CreateFile (  
    "out_test",  
    GENERIC_WRITE,  
    dwShareMode,  
    lpSecurityAttributes,  
    CREATE_ALWAYS,  
    FILE_ATTRIBUTE_NORMAL,  
    hTemplateFile  
);  
if(sinkFile == INVALID_HANDLE_VALUE) {  
    fprintf(stderr, "File open operation failed\n");  
    ExitProcess(1);  
}

Windows File Manipulation Program(4)

// Main loop to copy the file
while  
(  
    ReadFile(  
        sourceFile, buffer,  
        BUFFER_LEN, &numberOfBytesRead,  
        lpOverlapped  
    )  
    &&  
    numberOfBytesRead > 0  
) {  
    WriteFile(sinkFile, buffer, BUFFER_LEN,  
        &numberOfBytesWritten, lpOverlapped);  
}  

// Terminating. Close the sink and source files
CloseHandle(sourceFile);  
CloseHandle(sinkFile);  
ExitProcess(0);  
}
Shell Command Line Interpreter

Interactive User

Shell Program

Application & System Software

OS System Call Interface

OS

The Shell Strategy

% grep first f3

read keyboard

Shell Process

fork a process

Process to execute command

f3

read file
**Bootstrapping**

- Computer starts, begins executing a *bootstrap program* -- *initial process*
- Loads OS from the disk (or other device)
- Initial process runs OS, creates other processes
Objects

- A recent trend is to replace processes by objects
- Objects are autonomous
- Objects communicate with one another using messages
- Popular computing paradigm
- Too early to say how important it will be ...