

MOSS:

A Mini Operating System Simulator

Fred Barnes (frmb@kent.ac.uk)
S05, Computing Lab.

CO501 MOSS, slide 1

Fred Barnes, February 2004.

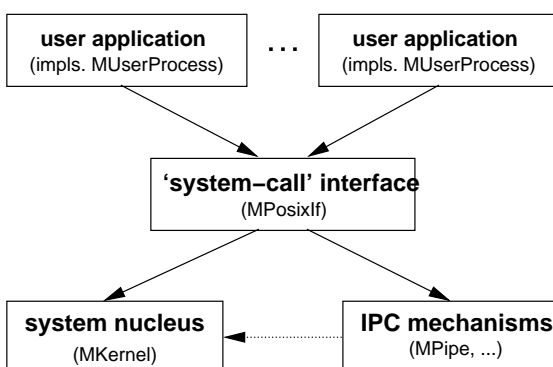
Overview

- An operating system *simulator* written in Java
- *Emulates* certain aspects of an operating system:
 - user and system processes
 - operating system 'nucleus'
 - system-call interface
 - signals
 - process scheduling
- Some aspects are hard (or pointless) to emulate:
 - memory management
- Others are incomplete:
 - file-system

CO501 MOSS, slide 2

Fred Barnes, February 2004.

Structure



CO501 MOSS, slide 3

Fred Barnes, February 2004.

OO and Operating Systems

- Operating systems are **not** generally object oriented
- Reasons for this vary
 - suitable languages for implementation: low-level vs. high-level
 - overheads (e.g. garbage collection)
 - not exactly object oriented in the first place
- Most operating systems are written in C, with large amounts of assembler for low-level interaction with hardware.

CO501 MOSS, slide 4

Fred Barnes, February 2004.

MOSS and Java

- Source files:
 - in Linux, for example, logical functionality is divided across different files:
 - * scheduler, memory-management,
 - * file-system interface (VFS),
 - * process management, device drivers, ...
- MOSS does similar; different files contain logically different parts of the operating system
- Explicit use of object-orientation is avoided, but we still have to be “Java friendly”
 - classes provide wrappers for **static** methods
 - no ‘instance’s are required

Run-Time Environment

- MOSS is quite similar to a UNIX/POSIX system (e.g. Linux)
 - POSIX flavoured system-call interface
 - provision of **processes** with numeric IDs (PID)
 - process signalling mechanism
 - concept of process hierarchy (e.g. parent process)
 - pipes/streams – stdin, stdout and stderr
 - multiple virtual processors
- Process scheduling is somewhat awkward, but emulated in a tidy way (virtual CPU)

- `MKernel.java` – largely concerned with process scheduling/management and signal delivery
- `MPosixIf.java` – provides the ‘system-call’ interface for (primarily) user processes
- `MInitTask.java` – the first process (initial task) of the system
- `MProcess.java` – ‘process control block’ (PCB)
- `MSystem.java` – system constants
- `MiniOSSim.java` – where ‘`main()`’ lives
- `MFile.java` – interface for ‘file operations’
- `MJavaConsole.java` – provides ‘`MFile`’ access to the “Java console”

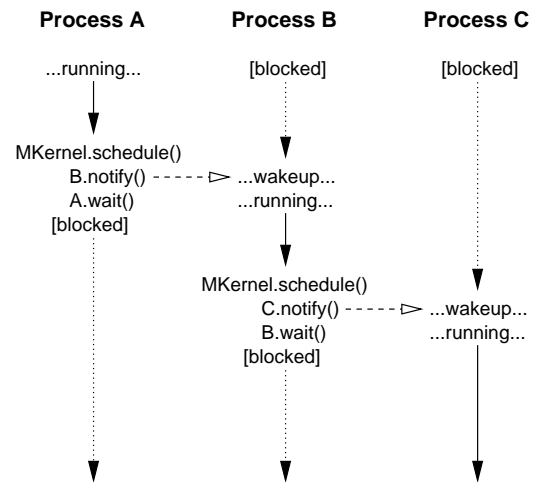
Java Threads and MOSS Processes

- In MOSS, the ‘`MProcess`’ class provides the “base” for a process
 - it extends the Java ‘`Thread`’ class, and thus exists as its own ‘thread of control’ within the JVM
- A MOSS ‘process’ consists of two things:
 - an instance of a class that implements either ‘`MUserProcess`’ or ‘`MKernelProcess`’ (interfaces)
 - an ‘`MProcess`’ instance that provides the ‘thread’ (emulated virtual machine)

Scheduling

- Free-wheeling execution of Java threads is not an option
 - operating systems don't work like that
- Processes in MOSS 'run' on a **virtual CPU**
 - in actual fact, a rather null entity – the JVM takes care of running threads
 - virtual CPU retains a reference to the current process
 - kernel maintains an array of 'current' processes, indexed by CPU

- Context switching (changing from one process to another) is done using Java monitor methods (executed on the 'MProcess' object)



- New processes are picked from the run-queue

- 'core' context-switch code is:

```

if (new_p != old_p) {
    synchronized (old_p) {
        if (new_p != null) {
            synchronized (new_p) {
                new_p.notify ();
            }
        }
    }
    try {
        old_p.wait ();
    } catch (InterruptedException e) {
        panic ("MKernel::schedule(). interrupted:"
            + e.getMessage());
    }
}
}

```

- 'old_p' is the **current** process, 'new_p' is the **next** process (i.e. the one we're switching to)

- Processes are added to the run-queue by calling 'add_to_run_queue' in MKernel

- A **voluntary reschedule** is performed by calling 'reschedule' in MPosixIf. The implementation of this is:

```

MProcess current =
    MKernel.current [MProcessor.currentCPU()];
MProcess.sync_process_signals (current);
MKernel.add_to_run_queue (current);
MKernel.schedule ();
MProcess.sync_process_signals (current);

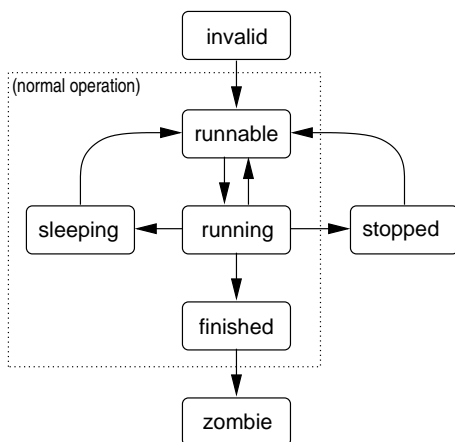
```

- The first line shows how the kernel can discover the current 'process context' – i.e. which process is executing

- The 'sync_process_signals' method is used to deliver signals to a process
- Signals are generated by calling 'queue_signal' in MKernel
- **Note:** MOSS is a non-preemptive system – if a user-process gets stuck in a loop, it stays there
 - there are ways around this (kernel-timer coupled with 'dead' handling if the thread ever interacts with the kernel again), but they're messy..
 - better to write nice non-polling code :-)

- Processes have a **state**; one of:
 - TASK_INVALID – invalid
 - TASK_STOPPED – stopped process (job control)
 - TASK_RUNNABLE – runnable process (on the run-queue waiting for execution)
 - TASK_RUNNING – currently running process
 - TASK_SLEEPING – waiting for I/O or timeout
 - TASK_FINISHED – process that has terminated
 - TASK_ZOMBIE – terminated and waiting for parent to collect status

- Process state transitions:



Kernel Interfacing

- User processes interact with the kernel through **static** methods in 'MPosixIf'
 - processes view each other as PIDs (simple integers)
 - file-descriptors (typically I/O streams) are also simple integers
 - return values from methods are mostly integer; conventionally, negative values indicate error (error codes defined in 'MSystem.java')
- User processes **should not** directly interact with other parts of the system

A Note on File Interfacing

- From a user-process perspective:
 - a simple integer (file-descriptor)
 - methods in MPosixIf such as 'open', 'read', 'write', 'lseek' and 'close'
 - allows interfacing with I/O streams, pipes, files, ...
- From the MOSS kernel perspective:
 - an instance of a Java class implementing 'MFileOps'
 - provides a 'back-end' for the user-visible methods

Blocking Processes

- When executing code inside the MOSS 'kernel', it is typically in the context of a MOSS process
 - the MOSS kernel code is executing in the Java **thread** provided by an MProcess
 - current process is accessed in a common way:

```
MProcess current =
    MKernel.current[MProcessor.currentCPU()];
```
- Sometimes, will want to suspend (deschedule) the current process
 - e.g. blocking reads and writes

- When a process is descheduled, a reference to it must be stored somewhere so that it can be re-summed
- Related code, in the context of a **different** process will generally be responsible for waking it up (re-schedule)
- There are two ways to sleep/wakeup a process in MOSS:
 - simple: not signal friendly
 - complex: very signal friendly

- Simple blocking:
 - store reference to current process
 - set process state to 'sleeping'
 - reschedule (MKernel.schedule())
- Simple wakeup:
 - recover reference of blocked process
 - if state is not sleeping, return
 - otherwise add to run-queue
- Any IPC must be worked in around this general framework
- Signal friendly..?

- Complex blocking: done in two parts, booleans flag involved:

```

boolean do_sleep = false;
synchronized (local) {
    synchronized (current) {
        // store reference to 'current' process
        current.state = TASK_SLEEPING;
        do_sleep = true;
    }
    // maybe wake up other blocked process
}
if (do_sleep) {
    boolean xsleep;
    synchronized (current) {
        xsleep =
            ((current.state == MProcess.TASK_SLEEPING)
             && !current.signalled);
    }
    if (xsleep) {
        MKernel.schedule ();
    }
    if (current.signalled) {
        synchronized (local) {
            // remove stored reference to 'current'
        }
        // return indicating interrupted
    }
}

```

- Corresponding 'wakeup' is much simpler, however:

```

synchronized (local) {
    ...
    // recover reference to blocked process
    MKernel.addTo_run_queue (...);
}

```

- The complexity in the 'sleep' occurs because a process may be awoken as the result of an asynchronous signal

The Assignment

- Choice of 5 different things:
 - A better 'pipe' IPC mechanism
 - A 'mail-box' IPC mechanism
 - A 'semaphore' IPC mechanism
 - A virtual device-driver
 - 'mail-box' demonstrator program(s)
- The first 3 (IPC implementations) are the most 'guided'
- The virtual device-driver provides scope for doing your own thing
- As does the demonstrator program, to some extent

Example: the pipe

- User-processes call 'MPosixIf.pipe()' to create
 - this creates a new 'MPipe' object and stores the MFileOps reference inside the process's 'file-table' (indexed by descriptor)
 - two (integer) descriptors are returned
- Read, write, etc. operations result in calls to the underlying MPipe **object**
 - user-process ↔ posix-if:


```
public static int read
    (int fd, byte buffer[], int count)
```
 - posix-if ↔ pipe-implementation:


```
public int read
    (MFile handle, byte buffer[], int count)
```

- The given implementation of a pipe (in 'MPipe.java') is not entirely trivial. It does, however, attempt to be correct. This includes being safe against asynchronous signal delivery...
- The implementation supports multiple readers and multiple writers on the same pipe (although such usage is not recommended)

Option 1: a better pipe

- This can be based on the existing pipe (more or less), although it does not need to support multiple readers/writers
- I'd suggest creating a new file/class, 'MPipe2', implementing MFileOps, that contains the code
 - and add a new method to MPosixIf to create the new pipe:


```
public static int pipe2 (int fds[])
```
- To test, modify the existing UPipeTest/UPipeTest2 'programs'

Option 2: mail-box IPC

- This is slightly different, and should be done using **static** methods
- A good place to start would be to create a new file/class, e.g. 'MMailBox', with static methods and 'attributes' (data) that holds most of the code
 - 'sendmsg' and 'recvmsg' at **this level** require four bits of information: source PID, destination PID, message type, and message 'Object'
- The methods added to MPosixIf would simply invoke the static methods inside 'MMailBox', passing relevant information

- Internally, the MMailBox class needs to store messages
- A sensible way to do this would be to create a small private class for a 'message':


```
private static class MailMessage {
    public int from_pid;    // source
    public int to_pid;     // target
    public int type;       // type
    public Object msg;     // message
}
```
- Then maintain an ArrayList (or other suitable 'container') of 'MailMessage' objects, for example
- Blocked processes (receivers) held on a local static queue

Option 3: semaphore IPC

- This is similar to the mail-box IPC idea, except that the mechanism is different (signalling instead of data-communication)
- Similarly, a good starting point would be to create a 'MSemaphore' class/file with static methods called through from MPosixIf
- The interface hints towards the implementation:
 - an internal 'Sema4' data-type (like 'MailMessage')
 - 'Sema4' objects held in a static ArrayList or other container

- 'Sema4' data-type can handle blocked processes itself, e.g.:

```
private static class Sema4 {  
    public int key;  
    public int value;  
    public MProcess waiting;  
}
```

Option 4: virtual device-driver

- There is a lot of scope for interesting things here...
- Interface should be provided through 'MFileOps'
 - for interfacing with user-applications, add a specific creation method to 'MPosixIf' that creates an instance of your virtual device-driver
 - similar to how 'pipe()' works, except that only one descriptor need be involved
- Small test/demonstration programs would be nice, but aren't strictly necessary

Option 5: mail-box demonstrator

- This is not necessarily an easier option
 - simple "hello world" sender/receiver – not good for marks :- (
 - 'simple' user-to-user chat via mail-boxes (roughly like UNIX's 'write') – good for marks :-)
- Typically only interfacing with 'MPosixIf'
- Could do something interesting by making part of the test a kernel-process (requires 'in-kernel' coding to start one of these, however)