

MOSS:

A Mini Operating System Simulator

SERG talk, 4th March 2004

Fred Barnes (frmb@kent.ac.uk)

S05, Computing Lab.

Contents

- Introduction and motivation
- Structure
- Java, OO, operating systems and MOSS
- Run-time environment
- Threads and scheduling
- Processes, signals and interfacing
- Blocking
- Example: the pipe
- The future

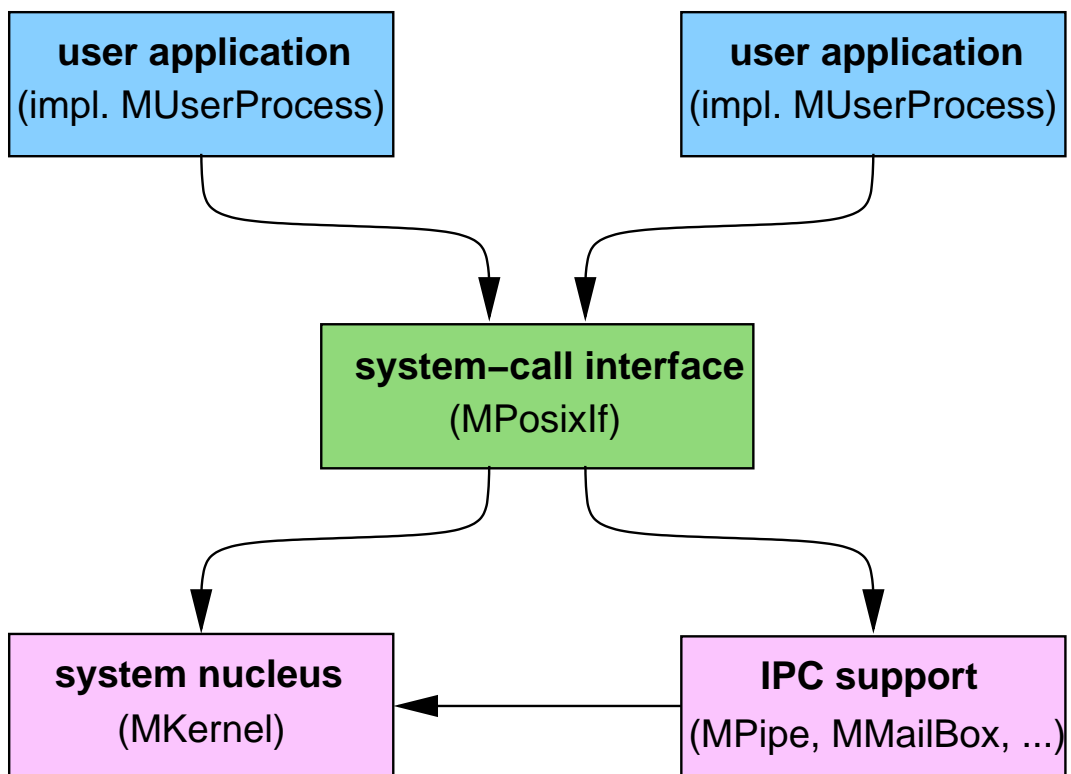
Introduction

- 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

Motivation

- Operating system programming often seen as difficult
 - code complexity
 - low-level nature of code/language
 - lengthy development cycle
- Want it to be easy!
 - ‘hands-on’ programming for students
 - potential prototyping platform

Structure



OO and Operating Systems

- Operating systems are **not** generally object oriented
- Probably reasons for this:
 - 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.

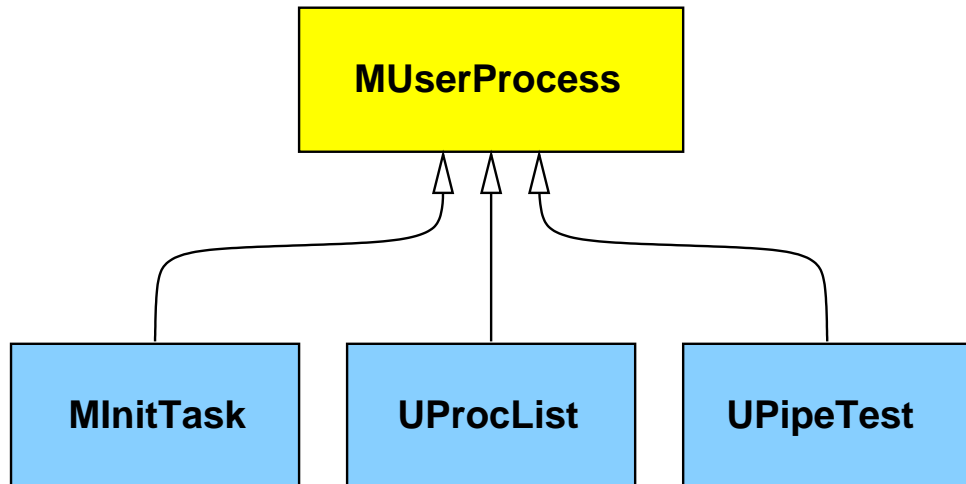
MOSS and Java

- 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

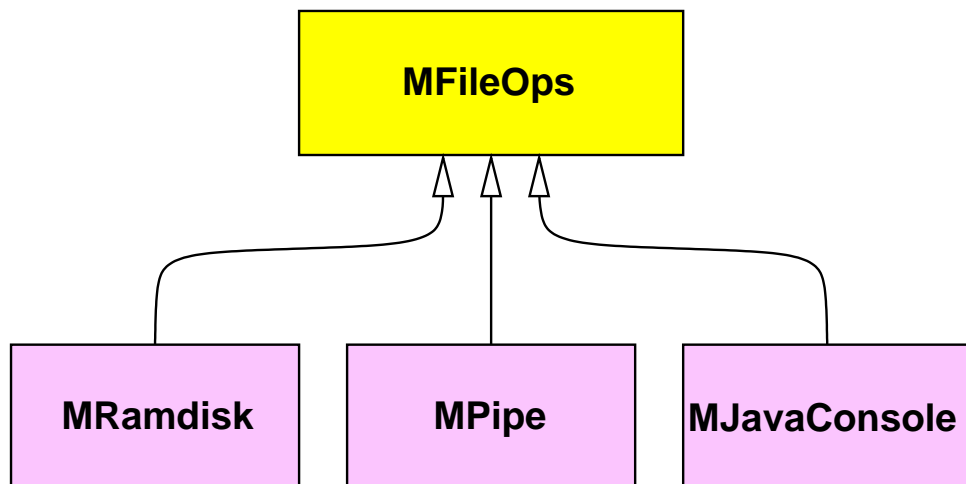
- `MKernel.java` – largely concerned with process scheduling/management and signal delivery
- `MPosixIf.java` – provides the ‘system-call’ interface for (primarily) user processes
- `MInitTask.java` – the first process (initial task) of the system
- `MProcess.java` – ‘process control block’ (PCB)
- `MSystem.java`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

- Some use of interfaces (and *real* objects) is required
 - provides an equivalent to C’s “structure of function pointers”
- `MUserProcess.java` – interface that *user-processes* implement
- `MKernelProcess.java` – interface that *kernel-processes* implement
- `MFileOps.java` – interface for *file-operations*
- `MDirOps.java` – interface for *directory-operations*

- User-processes implement “MUserProcess”:



- Any device that provides ‘file-like’ interfacing implements “MFileOps”:



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
 - 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)

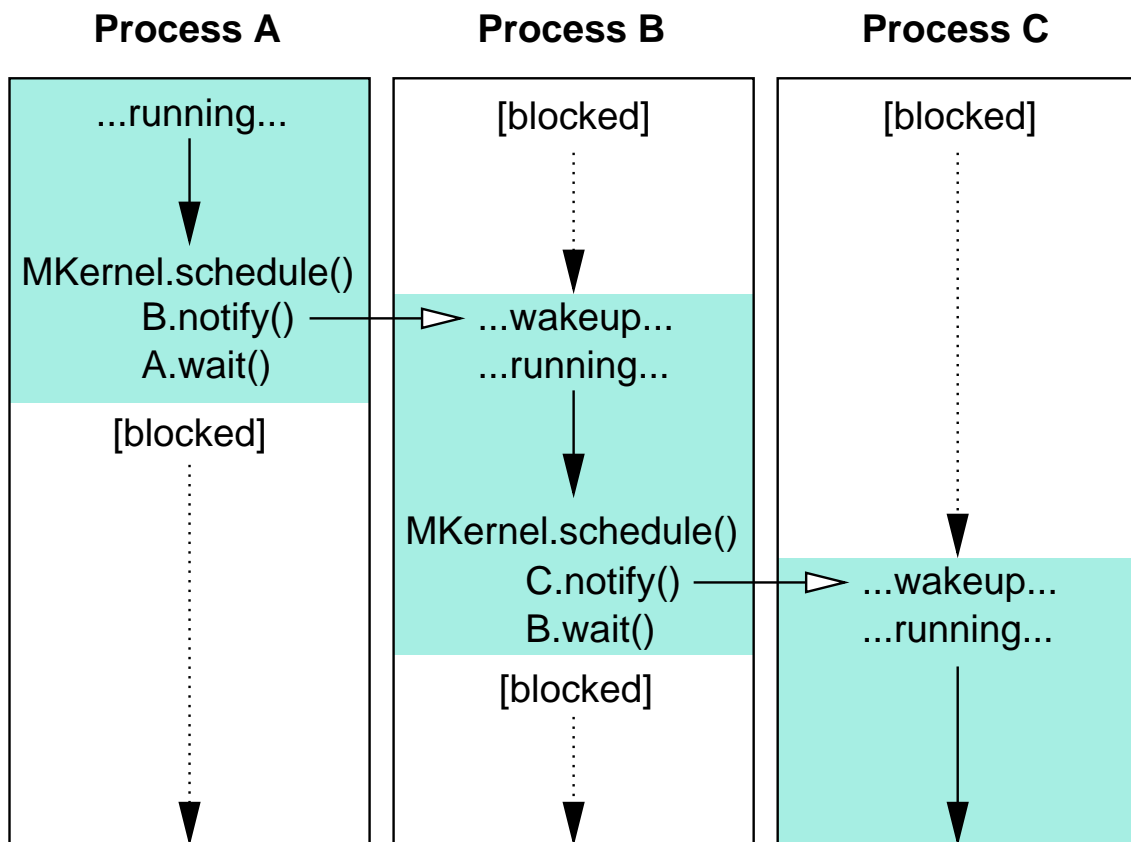
Java Threads and MOSS Processes

- 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:

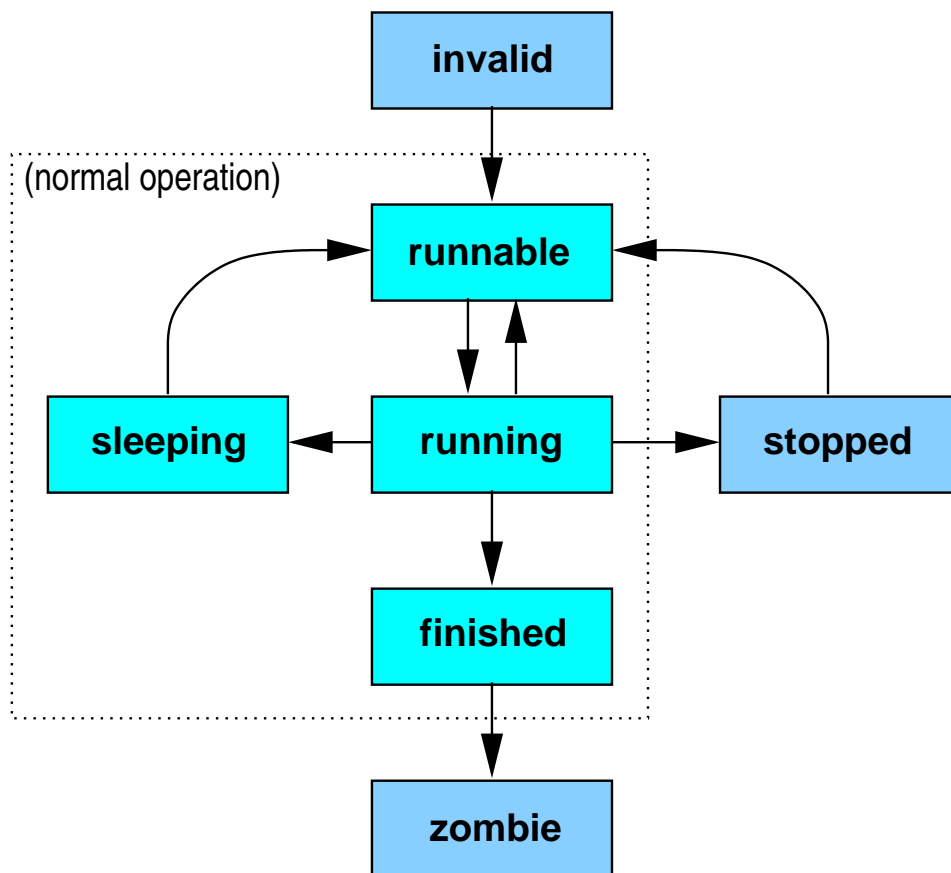
```
MProcess current =  
    MKernel.current[MProcessor.currentCPU()];  
  
MProcess.sync_process_signals (current);  
MKernel.add_to_run_queue (current);  
MKernel.schedule ();  
MProcess.sync_process_signals (current);
```

- 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 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
 - although there is nothing to stop them...

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’, linked by an ‘MFile’
 - 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 (de-schedule) the current process, e.g.:
 - blocking reads and writes (networking)
 - waiting for semaphores
 - other IPC

- When a process is descheduled, a reference to it must be stored somewhere so that it can be resumed
 - 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
- ... race-hazards on multi-processor systems

- 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 'blocking' code must be worked in around this general framework

- Need somewhere to store blocked processes:

```
MWaitQueue blocked = new MWaitQueue();
```

- Simple blocking:

```
MProcess current =  
    MKernel.current[MProcessor.currentCPU()];  
  
blocking.add_to_queue (current);  
current.state = MProcess.TASK_SLEEPING;  
MKernel.schedule ();
```

- Simple wakeup:

```
MProcess other =  
    blocking.get_from_queue ();  
  
if (other.state == MProcess.TASK_SLEEPING) {  
    MKernel.add_to_run_queue (other);  
}
```

- Complex blocking: usually done in two parts, boolean flags involved:

```
boolean do_sleep = false;
synchronized (local) {
    synchronized (current) {
        blocked.add_to_queue (current);
        current.state = MProcess.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) {
            blocked.del_from_queue (current);
        }
        // return indicating interrupted
    }
}
```

- Corresponding ‘wakeup’ is much simpler, however:

```
synchronized (local) {  
    MProcess other =  
        blocked.get_from_queue ();  
  
    if (other.state == MProcess.TASK_SLEEPING) {  
        MKernel.add_to_run_queue (other);  
    }  
}
```

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

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)
- Now handles *end-of-file* and *SIGPIPE*

CO501 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)
- Most people chose to implement the ‘mail-box’ IPC mechanism
- Not marked yet, but there appears to be a wide range of solutions

Early Conclusions

- Going well so far :-)
- Complex in the expected places/ways
 - particularly for multi-CPU handling
- User processes cannot be destructive
 - MPosixIf (and below) are ‘defensive’
 - JVM will handle “null-pointer” exceptions, etc.
- Quite similar to Linux in some of the code logic (and presumably other UNIX systems)

The Future

- More code ...
 - currently around 6000 lines
- File-systems:
 - basic infrastructure for file-system handling is in place
 - “Object File-System” under construction (free-hanging OO)
- Networking:
 - not in place, but most likely a ‘socket()’ in ‘MPosixIf’
 - some operations (bind/accept/listen) might be a bit of a fudge

The Future (2)

- Applications:
 - a “telnet server” or equivalent
 - NFS client (and perhaps a server)
 - graphical interface ?
- Loaded in BlueJ successfully, mostly for the students, who seem strangely unfamiliar with “vi”..
 - might be nice to put BlueJ to some practical use, however – e.g. visualising/fiddling the ‘Object FS’