An Overview of EGOS: The Earth and Grass Operating System

Robbert van Renesse

1 Quick Overview

The Earth and Grass Operating System (EGOS) is a "user mode operating system" in that it runs inside an ordinary (Linux or Mac OS X) process. Instead of approx. 17M lines of code in an OS like Linux, EGOS proper has approx. 3,000 lines of code, making it relatively easy to wrap your brain around the whole thing.

EGOS consists of three layers (bottom to top):

1. *Earth* emulates an x86 hardware platform, replacing its rather complicated low-level hardware management facilities with a simple software-loaded TLB, interrupt handling, (emulated) kernel/user mode, and a few simple devices such as a terminal, clock, disk, and so on.

2. *Grass* is a simple but working operating system microkernel that manages multiple processes (including paging), files, directories, and so on.

3. On top of all this run the applications, and EGOS comes with some simple Unix-style apps like a shell.

Much of Grass is structured as a set of "kernel processes" that interact via message passing. (A Grass user process is a special kind of kernel process that manages a virtual address space—more on this later.) The kernel processes implement simple services such as a disk service, a "tty" (terminal) service, and a service to start new processes (the "spawn" service). Grass is a microkernel, and it runs a complete file service in user space. (More on this later as well.)

There are three message types:

1. `MSG_REQUEST`: a request message sent from a client process to a server process;

2. `MSG_REPLY`: a reply message sent by a server process, in response to a request message, to the client process;

3. `MSG_EVENT`: "unsolicited messages" that can be sent in an unstructured fashion. These messages are currently only used to notify deaths of processes.

Because services are implemented by processes, only seven system calls suffice:

1. `gpid_t sys_getpid()`: returns the process identifier of the current process;

2. `int sys_recv(type, max_time...)`: wait for a message of the given `type` (a request or an event) and then return it. `max_time` specifies a time-out in milliseconds, with 0 meaning that the process is willing to wait for a message indefinitely;

3. `int sys_send(dst, type, ...)`: send a message of the given `type` (a reply or an event) to process `dst`;

4. `int sys_rpc(dst, ...)`: send a request message and wait for a response;

5. `void sys_exit(int status)`: terminates the process with the given status;

6. `unsigned long sys_gettime()`: returns the time since the kernel was booted in milliseconds;

7. `void sys_print(char *string)`: prints the given string, which is useful when debugging and calls like `printf()` no longer work.

Typically server processes run a loop in which they wait for a request message, then perform the requested service, and send a reply. They are not required to send a reply before waiting for the next request. For example, the terminal server can accept requests for writing to the terminal screen while another client is waiting for terminal input. A client sends a request message and then immediately blocks waiting for a response. It is not possible for a single process to have multiple outstanding requests.

There are a variety of kernel processes that can roughly be grouped by the type of interface they implement:

- Block services: these provide interfaces to read and write disks by "inode number" (partition identifier). Grass runs two of these: one to support the user level file server, and one for paging.

- File services: these provide interfaces to read and write files by "file number." (A file is identified by the process identifier of the server and its file number.) There are currently two file servers:

    1. the "tty" or terminal server that is used to read from the keyboard or write to the screen;
    2. the "ram file" server is a very simple file server that keeps files in memory. It is used for /tmp as well as to hold some executables that cannot be paged, such as the main file server that runs in user space.

- The "spawn server" supports starting new user processes.

- The "gate server" provides an interface to the Earth layer and the outside world. For example, through the gate server it is possible to read files from the underlying Linux or Mac OS X file system, or to read the absolute time.

Other important operating services run in user space:

- A block server is a storage server that presents a "block device" as a simple array of blocks. A block server can serve multiple block devices, each identified by an inode number. A block server can support the file server listed above but also paging. Each block server actually consists of a set of "block layers," which will be explained later. A block server uses the same interface as the kernel block servers.
- The "block file server" implements a full-featured file system on top of a block server. It uses the same interface as the kernel file servers.
- The directory server maps file names to file numbers and provides interfaces to insert, lookup, or delete mappings. Directories are kept on files. While processes could in theory update directories themselves using the file interface, doing so would lead to all kinds of race conditions when multiple processes try to do so. The directory server serializes all those updates.
- The "sync server" synchronizes the caches of all files every 30 seconds.
- The "init server" is the first user level process that runs: it initializes the file system and runs the login process, which in turn launches a shell.

2 Building and Running EGOS

First install the code in a directory. You can build EGOS either for 64-bit Linux or 64-bit Mac OS X. (We recommend Linux—it has better debugging tools like `gdb` readily available.) Run '`make`', and everything should build just so. If you make changes to any source file, you need to run `make` again. However, there are bugs in the "Makefile" and sometimes some files do not get recompiled properly. In that case, run:

```
make clean
make
```

The easiest way to launch EGOS is to run '`make run`'. A login prompt should come up, and you can log in using the guest account "guest" with password "guest". A "shell" should come up, and you can type in commands such as

- `ls`: list the current directory;
- `echo hello world`: print "hello world";
- `cat README.md`: print file "README.md";
- `shell`: launch another shell, nested in the current one. (The number in the prompt is the process identifier, so you can tell which is which. As with Linux and Mac OS X, if you type '`<ctrl>d`', you indicate an 'END-OF-FILE' terminating the shell.)
- `shutdown`: synch all files (if using a write-back cache) then kill all processes and quit (may cause kernel crash due to dependencies).

If you type '<ctrl>l', all processes will be listed, and you should see each of the kernel processes mentioned above as well as some user processes. If you're done with EGOS, type '<ctrl>q' and EGOS will quit without synching for write-back cache. (Note that the <ctrl> commands are interpreted by the terminal server, not the shell.)

For example, if you run `ls -l` you should see something like this:

```
S:I    SIZE    NAME
===================
5:20   128     ..dir
5:6    160     ...dir
5:23   2682    README.md
5:24   64      script.bat
```

Each line lists an entry in the current working directory. In front of each file name, "`S:I`" indicates the server (by process identifier), and the "file number" of the file. Together, these identify a file, so we call the pair a *file identifier*. Most names end in ".<*type*>", indicating the file type. For example, directories have a ".dir" suffix. By convention, "..dir" is the current directory and "...dir" is the parent directory.

In this case, process 5 is the file server. You can see that if you type '<ctrl>l', which should print something like this:

```
Processes (current = 2):
PID   DESCR    UID STATUS      OWNER ALARM   EXEC
   1: main       0 AWAIT EVENT     1
   2: tty        0 AWAIT REQST     1
   3: blockfil   0 AWAIT REQST     1
   4: ramfile    0 AWAIT REQST     1
   5: diskfile   0 AWAIT REQST     1
   6: dir        0 AWAIT REQST     1
   7: spawn      0 AWAIT REQST     1
   8: user       0 AWAIT EVENT     1         5:7
   9: user       0 AWAIT REQST     8         5:10
  10: user       0 AWAIT EVENT     8         5:8
  11: user     666 AWAIT     2    10         5:19
```

In this case there are 11 processes, with process 2 being the current process (the "tty" or terminal server). None of them are actually running—they're all waiting for something (a message to arrive). Processes 1, 8, and 10 are waiting for death notifications of the processes that they own (`AWAIT EVENT`). Process 1 is the first kernel process for EGOS. Process 8, the "init" process (executable "/etc/init.exe"), is the first and only user process that is spawned during initialization by the Grass kernel. "init" then spawns the "login" process ("/etc/login.exe") using the spawn server (process 7), which in turn launches the shell ("/bin/shell.exe"). Process 11 is the shell, which is waiting for a response from process 2, the tty server. Most other processes are servers waiting for a request (`AWAIT REQST`).

Processes 8 through 11 are user processes. The listing contains the file identifier of the executable they are running. You can find what those are by running `ls /bin` and `ls /etc`.

3 Directory Organization

The directory under Linux or Mac OS X is organized as follows:

- `.`: the top level directory contains the Makefile and, when run, also the executables that are generated. The Makefile invokes three other Makefile (in `src/make`): Makefile.apps (to generate a library called `lib/libgrass.a`) as well as a set of applications (in the `bin` directory)), Makefile.grass (to generate the Grass kernel called `build/grass/k.exe`), and Makefile.earth (to generate the Earth virtual machine, which is `build/grass/earthbox`). Note that EGOS executables end in the `.exe` suffix. These will be loaded into the EGOS file system.
- `src/make`: Makefiles for building the EGOS system;
- `src/earth`: the source files for the Earth layer;
- `src/grass`: the source files for the Grass layer;
- `src/include`: C standard include files;
- `src/h`: EGOS-specific include files;
- `src/lib`: the source files for the library for applications;
- `src/block`: the source files for the various block layers;
- `src/apps`: the source files for the various apps;
- `src/tools`: some useful tools;
- `bin`: EGOS executables;
- `lib`: EGOS runtime libraries and linkable objects;
- `storage`: virtualized storage devices;
- `tcc`: Tiny C Compiler;
- `usr`: EGOS home directories;
- `docs`: source files for documents.

All these are copied into the EGOS file system when EGOS is being initialized.

4 The Earth Layer

The Earth layer emulates the underlying hardware devices that are not actually present inside a Linux or Mac OS X process. The Earth layer is designed to support different operating system kernels running on top of it, although we shall focus on the case where Grass is running on top. Earth currently only supports a single core.

Earth creates two address spaces. The kernel is loaded at address 0xA,000,000,000, while user process will run at address 0x9,000,000,000. The kernel can interface with Earth through the "earth" interface (see `src/earth/intf.c`).

4.1 The TLB

The TLB (Translation Lookaside Buffer) is a fixed-size associative array that essentially maps virtual addresses to physical addresses. It also supports protections bits (`P_READ`, `P_WRITE`, `P_EXEC`), although the Grass layer currently simply sets all bits whenever it maps a page. When the current process makes an access to a virtual address, it may or may not be mapped. If it is mapped, the TLB specifies which physical page is associated with the virtual page in which the virtual address falls. If it is not mapped (or the protection bits do not allow access), an `INTR_PAGE_FAULT` signal is thrown. The TLB is not backed by a hardware-implemented page table, so it is up to the operating system to handle the interrupt, add a new entry to the TLB, and then return from interrupt. The operating system is free to implement single- or multi-level page tables, segmented or inverted page tables, or whatever it likes. (Grass implements a simple single-level level page table.) The TLB does not know about different processes, so it needs to be "flushed" whenever context switching between processes.

4.2 Interrupts

There are currently just four types of interrupts:

- `INTR_PAGE_FAULT`: a page fault, as discussed above. It specifies the address at which the page fault happens, but unfortunately nothing else. (But you can guess based on the protection bits of the page.) The kernel page fault handler should either map the address or decide to kill the process.
- `INTR_CLOCK`: clock interrupts currently happen 100 times per second. They are needed for the kernel to get control back from a running user process.
- `INTR_IO`: This means some I/O is ready. Which kind is unfortunately not specified, however, each device can separately specify a callback function to be invoked when it has I/O available for processing. The `INTR_IO` interrupt is intended to allow a scheduling decision to happen.
- `INTR_SYSCALL`: A system call interrupt is thrown when a user process want to execute one of the system calls listed above. System calls are implemented with an illegal x86 instruction (`.long 6` to be precise). The interrupt handler is passed a pointer to a `struct syscall` record that specifies what system call is invoked, and also allows for a return value to be sent back to the user process.

Interrupts are *only* enabled in two cases: when EGOS is executing in user space, or when EGOS is waiting for I/O. Thus, when the operating system (Grass) is executing, interrupts are disabled. Interrupts are never "dropped," even when disabled. Once enabled, they fire and need to be handled.

4.3 User Space vs. Kernel Space

Ok, there really is no user space and kernel space—it's all make believe because a Linux or Mac OS X process runs in user space only. But the emulation is not bad. Earth reserves a range of address space, `VIRT_BASE` to `VIRT_TOP`, to be used in "user space". There is also a range of "physical memory" reserved that can be used for user process pages.

When not executing in user space, the process state is saved on the "interrupt stack" stack of the process. (In Grass, each user process ends up with three stacks: the kernel stack, the interrupt stack, and the user stack. Only the user stack is paged.) In Linux and Mac OS X, one cannot change the interrupt stack while the operating system thinks the process is executing on it. Therefore the contents of the interrupt stack must be saved and restored by copying. The initial interrupt stack is created in `main()` (in "grass/main.c()") by explicitly accessing address 1. (By using `ctx_switch()`, we also are able to move back off the interrupt stack so we can easily copy it.) This interrupt stack then becomes the initial interrupt stack of all user processes.

4.4 Devices

Earth supports a few simple devices:

- `clock`: the clock device that interrupts 100 times per second;
- `disk`: a disk device for storing and retrieving blocks;
- `gate`: a pseudo-device that is useful for interactions with the "outside world";
- `log`: allows the kernel to log or print messages;
- `tty`: the terminal device for reading from keyboard and writing to the screen. It interrupts when there is input from the keyboard;
- `udp`: a device for sending and receiving UDP packets, which interrupts when a UDP packet has been received. (Not currently used by Grass.)

5 The Grass Layer

The Grass layer runs on top of Earth. It implements multi-processing, messages passing, paging, files, and directories.

5.1 A Grass Kernel Process

A Grass kernel process has a kernel stack and is "non-preemptive": a process runs until it explitly yields to another process using function `proc_yield()`. Function `proc_yield()` searches for another process to run and then executes a so-called *context switch* by saving its registers on its stack followed by restoring the registers that were previously saved by the other process. The other process is now running.

A process is always in one of the following states:

- `PROC_RUNNABLE`: ready to run or in fact running. Because EGOS is single core, only one process can run at a time.
- `PROC_WAITING`: waiting for a message to arrive. Currently a process has to specify the type of message that it is waiting for. (Separate state variables indicate what type of message the process is waiting for.)
- `PROC_ZOMBIE`: this is a process that is dying but not quite dead. When a zombie process context switches to another process, the latter cleans up the zombie once and for ever.

The current process, which may or may not be runnable, is pointed to by pointer `proc_current`. This pointer should always be valid. All runnable processes, except any that is current, are on the `proc_runnable` queue. That is, processes are scheduled in round-robin order. When a process yields, it tries to dequeue a runnable process. If there is none, the process either returns (if itself is still runnable), or it waits for I/O interrupts or timeouts (using Earth function `intr_suspend(timeout)`).

5.2 A Grass User Process

A Grass user process is simply another kernel process, but one that has a page table and an interrupt stack attached to it. Its code is in `user_proc` in file "grass/spawn.c". The page table maps "pages" to "frames." It is a simple one-level page table, with one entry for each virtual page. The page table starts out empty. When the process runs, page faults happen whenever the process accesses an unmapped page in its virtual address space range. The job of function `proc_pagefault()` is to allocate a frame, save it in its page table, add a new entry to the TLB, and then return from the interrupt. For pages in the code and data section of the process, the corresponding frames are initialized by reading the data out of an executable file. Other frames are zero-initialized.

The top of the user stack of the process is initialized with a block of data that contains the arguments to the process, as well as the "Grass environment" pointed to by macro `GRASS_ENV` (see file `src/h/egos/syscall.h`), which has the following fields:

- `self`: the process identifier of the process, which should be accessed by function `sys_getpid()`;
- `servers`: an array of process identifiers for various handy server processes;

- `stdin`, `stdout`, `stderr`: file identifiers for standard input, output, and error output;
- `cwd`: file identifier of the current working directory;
- `argc`, `argv`, `envp`: point to the arguments of the process.

A process cannot directly access its own virtual memory when executing in kernel mode (because page fault interrupts are disabled in kernel mode). Instead, it should use function `copy_user()`.

5.3 Grass File Systems

Grass currently offers two file systems. The first, "ramfile," is a simple file system that keeps each file in contiguously allocated memory. It is simple, but not durable, and appending to a file may require the entire file to be copied to new place. It runs as a kernel process.

The other file system, called "bfs" (for "block file server") provides the same interface but is based on a sophisticated layered storage subsystem. It runs in user space, and is layered over a "block server."

A block server in turn provides a layered abstraction of a "disk." Each disk is partitioned into one or more "i-nodes", and each i-node consists of a sequence of blocks, numbered $0, 1, ...$ A block is typically a power of 2 bytes. The block interface allows one to read or write a block at a time—that is, there are no interfaces to access individual bytes.

In Grass, disks are virtual. Each "disk module" implements the same block interface. The bottom layer would typically be a kernel block server that actually keeps blocks in a file. One can also use a "ram block server" if you simply want to keep the data stored in memory. Grass provides a wide variety of disk module types, including:

- Check disk module: keeps a hash of all the blocks and checks to make sure that every block read has the same contents as the last time it was written.
- CLOCK disk module: implements a write-through cache based on the CLOCK algorithm.
- Combine disk module: combines an array of underlying block stores into a single block store.
- Debug disk module: prints all the read and write operations that flow through it.
- Map disk module: forwards calls to a particular inode in the block store below.
- Partition disk module: cuts up an underlying disk module into multiple fixed size partitions (each is identified by "inode number"). You can layer a disk module on top of each of those partitions.
- Protocol disk module: this module allows access to a remote disk module using Grss RPC.

- RAID0 disk module: this is layered on top of multiple underlying disk modules and simply stripes operations across them for better throughput.
- RAID1 disk module: this is layered on top of multiple underlying disk modules and mirrors everything written on all disks for better fault tolerance, and distributes read operations for better throughput.
- RAM disk module: this a disk module that stores the blocks in memory, which has good performance but poor durability.
- Statistics disk module: keeps track of the rates of block read and block write operations that flow through it.
- Tree disk module: like the Partition disk module, but cuts up a disk module into multiple variable size "partitions," each of which is a tree of blocks organized under a single inode.

You can take these disk modules and stack them in any way you like, creating a DAG (Directed Acyclic Graph) of disk modules. You may even want to stack multiple instantiations of the same disk module type.

The tree disk module is the one that looks closest to a Linux or Windows file system. In fact, it would be good if eventually we had an EXT4 disk module, a FAT disk module, and so on, but for now there is only the tree disk module. It implements each partition as a tree of blocks.

The tree disk module does not (knowingly) keep track of which inodes are free or allocated, how many bytes exactly they contain, when it was last written, or any of that information. The block file server keeps this information on partition 0 of the tree disk, and thus implements the file interface. But the file server would need no changes if it ran on top of an EXT4 disk module, say.

6 Service Protocols

Looking up file names, reading and writing files, and spawning processes are all implemented by a specific protocol between a client that requests a service and a server that implements the service. This section lists both the service interfaces that are available to clients and some specifics of the underlying protocols. The interfaces and the protocols are specified in the "shared" directory.

6.1 File Interface and Protocol

To create a file, use the following interface:

```
bool_t file_create(gpid_t server, gmode_t mode,
                   /* OUT */ unsigned int *p_fileno);
```

Here `server` is the process identifier of the file server and `mode` contains the access control bits. It returns `True` if successful, and if so it returns a file number in `*p_fileno`.

To read a file identified by file identifier `fid`, a client invokes:

```
bool_t file_read(fid.server, fid.file_no, offset,
                 /* OUT */    *data,
                 /* IN/OUT */ *psize);
```

Argument `offset` specifies the byte offset into the file, `data` points to a region of memory in which the data is to be copied, and `*psize` indicates the size of that region and thus the maximum amount of data to be read. The function returns either `False` if there is an error, or `True`, in which case `*psize` is filled with the amount of data read by the function. If `*psize == 0`, then End-Of-File (EOF) has been reached.

To write a file identified by `fid`, a client invokes:

```
bool_t file_write(fid.server, fid.file_no, offset,
             /* IN */ *data, size);
```

where once again `offset` is a byte offset into the file, `data` points to a region of memory containing the data to write, and `size` is the size of that region in bytes.

Other handy interfaces:

```
bool_t file_getsize(gpid_t svr, unsigned int file_no,
                                unsigned long *psize);
bool_t file_setsize(gpid_t svr, unsigned int file_no,
                                unsigned long size);
bool_t file_delete(gpid_t svr, unsigned int file_no);
bool_t file_sync(gpid_t svr, unsigned int file_no);
```

The file service protocol uses request and reply messages specified in "<egos/file.h>" (which you can find in `h/egos/file.h`). The protocol is implemented by file servers and also the tty server. The tty server ignores the offset, and uses file number 0 for standard input, file number 1 for standard output, and file number 2 for error output.

6.2 Directory Interface and Protocol

Directories are simply maintained in ordinary files (by convention named with the ".dir" extension). However, because multiple processes might simultaneously try to update the same directory, update operations to insert and remove entries in a directory all go through the directory service, whose process identifier is available to a user process as `GRASS_ENV->dir`. By convention, most file services will use file number 1 to contain a "root directory" for that file service.

An entry in a directory file has the following format (see "<egos/dir.h>"):

```
struct dir_entry {
    fid_t fid;
    char name[DIR_NAME_SIZE];
};
```

Some entries in a directory may be all zero bytes, indicating that the entry is not currently in use.

The interface to the directory service is as follows:

```
bool_t dir_lookup(gpid_t svr, fid_t dir, char *path,
                                  /* OUT */ fid_t *pfid);
bool_t dir_insert(gpid_t svr, fid_t dir, char *path,
                                  fid_t fid);
bool_t dir_remove(gpid_t svr, fid_t dir, char *path);
```

Here `svr` is typically `GRASS_ENV->dir`, `dir` is the file identifier of the directory, and `path` is a (relative) path name. (The current directory server does not yet support actual path names with slashes in them.) There is no `dir_list` interface—the client is supposed to simply read the corresponding file using `file_read()`.

The same header file specifies the format of the protocol messages.

6.3 Spawn Interface and Protocol

To spawn a new process, a client invokes:

```
bool_t spawn_exec(gpid_t svr, fid_t executable,
               char *argb, unsigned int size,
               /* OUT */ gpid_t *ppid);
```

Here `svr` is typically `GRASS_ENV->spawn`, `executable` the file identifier of the file containing the executable (with a format specified in "<egos/exec.h>"), and `argb` is the initial contents of the user stack consisting of `size` bytes.

`argb` should contain the process argument and the "grass environment." It can be generated with the following function:

```
bool_t spawn_load_args(struct grass_env *ge_init,
           int argc, char **argv,
           /* OUT */ char **p_argb, unsigned int *p_size);
```

Here `ge_init` is the initial grass environment, which is usually just `GRASS_ENV`. The spawn server will overwrite field `self` with the process identifier of the new process. `argc` and `argv` specify the argument vector. The result is allocated by `malloc()` (and must be freed when no longer in use) and placed in `*p_argb`. `*p_size` is filled with the size of this region.

There is no `spawn_wait()` interface—instead a process can simply wait for `MSG_EVENT` messages, which have type `struct msg_event` and contain the process identifiers and exit statuses of the processes that die (see "<egos/syscall.h>"). Status `-1` is used for the case when the process dies because it accessed an illegal address.

6.4 Block Interface and Protocol

The block interface is for disk modules. Each server can manage multiple disk module instances that are identified by inode number. The interface is similar to the file interface but manipulates blocks of `BLOCK_SIZE` bytes:

```
bool_t block_read(gpid_t svr, unsigned int ino,
        unsigned int offset, void *addr);
bool_t block_write(gpid_t svr, unsigned int ino,
        unsigned int offset, const void *addr);
bool_t block_getsize(gpid_t svr, unsigned int ino,
        unsigned int *p_nblocks);
bool_t block_setsize(gpid_t svr, unsigned int ino,
        unsigned int nblocks);
bool_t block_sync(gpid_t svr, unsigned int ino);
```

Note that the `offset` is measured in blocks, not bytes.

6.5 Password Service and Protocol

The password file "/etc/passwd" should not be directly updated for two reasons. First, concurrent updates could leave the password file corrupted. Second, it is of course a security issue if everybody could update the password file. The password file is therefore updated by the "password server," which runs in user space. The `passwd` application allows users to update their password. This application sends a request to update the password file to the password server.

The password file has a similar format as what is used in Linux, with one line for each user. For example, the line for user `guest` might look like:

```
guest:CDS56G1ck?kWezL13Wg3Btei5Hkfad0M:666:/usr/guest
```

A line consists of four entries:

1. user name;
2. hashed password;
3. user identifier;
4. home directory.

The hashed password consist of a four character random salt and a 28 character hash based on a SHA256 computation over the salt and the user's password. (By the way, using SHA256 is not a great choice because of the availability of hardware that can quickly compute SHA256 hashes over a large number of inputs.)

The first line in the password file is for user `root`, aka the *superuser*. The superuser has certain managerial powers, like being able to read or write any file and kill any process.

7 The Application Layer

The sources of the applications in the "apps" directory in the host operating system, while the executables are installed in the `/bin` and `/etc` directories under Grass. The sources use library routines that are stored in the host `src/lib` directory directory as specified in the previous section. Each application is loaded with `lib/crt0.o`, in which `_start()` is the entry point. The address of `_start()` is in the header of the executable (see "<egos/exec.h>"). Function `_start()` invokes `main()`. Should `main()` return, then it automatically invokes `sys_exit(status)` using the return value of `main()`.

The applications are compiled with a standard C compiler (such as gcc or clang), and loaded with `lib/libgrass.a`. The resulting executable has a ".int" extension and is in either ELF format (Linux) or MACHO format (Mac OS X). Tools in the `src/tools` directory then convert these into ".exe" files with headers as specified in "<egos/exec.h>". The ".exe" files are automatically copied into the EGOS file system. `main()` (in `src/grass/main.c`) starts the "init.exe" process as mentioned above, which in turn spawns an instance of `bin/shell.exe`.

The library supports `malloc()` and `printf()` and a bunch of other common standard C functions. There is also a rudimentary Posix compatibility layer (see `lib/unistd.c`).

7.1 The Shell

The shell is a typical command-line interface to the operating system. The shell accepts one command per line. A typical command is:

```
cat README.md
```

This will try to locate the file "cat.exe" and then contact the spawn server to run it. In this case, function `int main(int argv, char **argv)` in "cat.exe" will be invoked with two arguments: "cat" and "README.md". The shell will then wait for the process to terminate before printing a new prompt and waiting for the next line of input.

It is possible to run commands in the background by ending the line with a "&". So, for example:

```
loop&
```

will run `loop.exe` but the shell will *not* wait for its completion. (`loop.exe` in `/bin` is handy for testing.) It just runs in a loop for a number of times that amounts to maybe 5-15 seconds.) If, at a later time, you want to explicitly wait for commands that may or may not still be running in the background, the shell as a built-in command:

```
wait [pid]
```

If you specify a process identifier, then the shell will wait for that process identifier to finish. If you don't specify a process identifier, the shell will wait for all background processes to finish.

Currently, the only other built-in command is

`exit [status]`

(status 0 is default). For example, try:

```
shell
exit 3
```

The (parent) shell should print something like `Process X terminated with status 3`. The shell prints a line like this everything it sees a process terminating with a non-zero status or if it sees a process terminating it was not explicitly waiting for.

7.2 Other Apps

- `cat [file ...]`: print the contents of the given files. If no files are specified, read standard input (terminated by <ctrl>d. Also, the "–" file name is interpreted as standard input. Returns 1 if a file does not exists.
- `cc ...`: Yes, there is a C compiler. Most apps and the Grass kernel itself can be recompiled under Grass, a sure sign of its maturity.
- `cp file1 file2`: copy `file1` to `file2`, creating `file2` if it didn't exist before.
- `echo [args ...]`: simply print the arguments.
- `ed file`: simple file editor like Unix v7 `ed`. Simpler even.
- `kill pid ...`: kill one or more processes.
- `loop [#]`: loop for the given number of iterations in a simple tight loop. `loop 0` means loop for ever. If no argument is given, the default loops for about 5-15 seconds.
- `ls`: list the contents of the current directory. Output is in the form `S:N name`, where `S` is the process identifier of the server, and `N` is a file number.
- `passwd`: to change the password of a user.
- `pwd`: print working directory name.
- `shutdown`: syncs all files (if write-back cache is enabled) and quit EGOS.
- `sync`: syncs all files (if write-back cache is enabled).

8 History and Acknowledgments

Much of EGOS was written by Robbert van Renesse in the summer of 2018 to replace the OS for the Cornell CS4411 class that he was teaching in the Fall semester of that year. However, he borrowed some code from earlier projects he had done. In particular, the "block layer stack" was largely drawn from code he had written for a CS4410 homework project in Fall 2015. Yunhao Zhang helped Robbert significantly in developing the educational aspects of EGOS and in particular refined many of the projects. He also wrote the fatdisk block layer, and taught CS4411 by himself in Spring 2019 and Fall 2020. Edward Tremel taught CS4411 in Spring 2020 and made various improvements.

Jason Liu (CS'19) took CS4411 in Fall 2018 and contributed cipherdisk.c, a block layer that encrypts/decryps blocks. Alice Chen (CS'19) also took the course that semester and, during Spring 2019, in collaboration with Robbert and Yunhao, made two important improvements to the block layer. First, she changed the block layer API so each layer instantiation can serve multiple partitions or inodes. Second, she added support for writeback caching. She also developed several unit tests for the block layer. Yizhou Yu helped with porting the Tiny C Compiler to EGOS. Kenneth Fang and Mena Wang contributed a Unix FS disk layer.