TUNIX

Operating System Manual

2024-04-20

Contents

What is TUNIX?

It is UNDER CONSTRUCTION.

TUNIX is a Unix-like KERNAL extension for the Commodore VIC-20 with UltiMem expansion which adds pre-emptive multi-tasking (currently cooperative), loadable drivers and recover-on-reset.

TUNIX performs very simple process, memory and device management. It could be a C program of about 500 lines in size. A schematic C version has been added for documentation ('src/tunix.c'). Perhaps it'll even be used for real one day but it would be rather slow.

The running version, written in ten times as much assembly language, has highly optimized data structures to make most use of the UltiMem's banking architecture without sacrificing simplicity.

Except for the IO23 area, all address space is available to a program. Exented memory can be allocated but it is highly recommended to not touch it at all and use a RAM disk or real drive instead.

Process management

Running processes are scheduled round-robin and switched on system call return. That works surprisingly well with character-oriented I/O. Pre-emptive multitasking is hoped to follow.

Processes can kill others and wait for their exit codes. Waiting takes place on FIFO lists. Before an exit code is returned, the next waiting process is woken up.

When the system runs out multiple processes, process 0 is shutting down the system.

Drivers

Programs can register a new set of vectors that can be applied to any device of a process. The device/-driver configuration is forked like all other resources. A driver assignment cannot be undone. A driver registration cannot be undone until all processes with assignments got killed.

Extended Memory (Consumption)

The rather large 8K-size RAM banks are kept in a singly-linked list and are also tracked per-process. A bank is released as soon it has been unreferenced by all.

Every process occupies six banks (48K). The kernel uses two and eight are reserved for loading non-TUNIX apps designed for the UltiMem expansion.

Raw Performance

(The time spans mentioned here have been guessed.)

Jim Brain's UltiMem expansion makes the TUNIX project possible in the first place. Thanks to it, most process switching can be done in hardware and by automatically generated speed code. [1]

System calls

The most expensive operation is the fork() system call, which can take up to 0.4s to complete since the address space of the forked process is cloned completely. Without speed code this would take about 0.8s.

Task Switches

Switching from one native VIC application to another takes 21ms, from a native to a TUNIX app takes 7ms (175ms the other way around), and a swith from a TUNIX app to another TUNIX app brills at 0.3ms to 10ms.

Time-consuming task switches aren't much of a factor as native apps are not multi-tasking and only run when they are connected to an active console. TUNIX apps can still run alongside unless that has been disabled (e.g. to run programs that take over the machine, to get the maximum performance out of the machine).

[1] Speed code is devoid of instructions that test or take turns. E.g. instead of looping over the same piece of code again and again, speed code instructions are repeated a fixed number of times to spare end-of- loop tests and conditional jumps. The backdraw is that the code is highly- specialized and can gain impractical sizes. Copying an 8K memory block takes about 48K of speed code but is several magnitudes faster than a regular copy loop.

I/O

Calling a driver takes 0.27ms. This leaves us with at least 17.7s overhead for transmitting or receiving 64K characters via device drivers.

The TUNIX system call driver's overhead is parsing and dispatching commands. Resource allocation and deallocation uses fast deque operations at constant execution times. All I/O call performance is reduced by about 1ms per operation in addtion to the operation itself.

Improved Performance

I/O performance can be improved using line- or block-oriented caches and block-wise read and write support at driver level.

Outlook

With tuned applications TUNIX could handle up to 32 processes at the same time. Text and data banks could be cached and/or reused instead of being loaded again. Also excluding low memory areas from task switches could provide further performance boosts.

Installation

TUNIX can not yet be installed.

Running TUNIX

After starting the TUNIX program it'll entertain you with some diagnostic messages and boot an instance of BASIC with process ID 1 and run self-tests that will crash on you after a minute or two.

Developing Applications

TUNIX can be used with any programming language that supports regular file I/O. Implementing drivers will most likely require using assembly language at some point, no matter what the rest of the driver was written in.

cc65

The number one set of programming tools for TUNIX is the cc65 compiler suite for 6502/6518-CPU platforms. It comes with an ANSI-C compiler, assembler and linker, a Unix-like standard C library, exhaustive documentation, and a vibrant community.

Differences Between KERNAL and UNIX I/O

Unlike Unixoids, the KERNAL does not provide default LFNs for standard I/O. Instead, apps reset a the selected input/oupt +device+ pair (aka “channels”) to the keyboard and screen device with help of the CLRCN system call. To make up for this, driver LFNDEV connects LFNs to devices, so a process’ standard I/O can be pipelined.

To get around resetting the channel pair you may decide to OPEN device #0 and #3 to have LFNs for CHKIN and CKOUT. The cc65 standard C library does that on start-up already.

KERNAL I/O Behaviour

This section describes the behaviour of the original KERNAL functions, along with the slightly different behaviour of TUNIX to support multi-tasking better.

Calling Conventions

Functions that return with an error code do so with the carry flag set and an error code in the accumulator (A).

Pointers passed in Y and X registers have the low byte in the X register and the high byte in the Y register respectively.

SETNAM - Set file name.

This routine is used to set the filename for the OPEN, SAVE, or LOAD system calls. A pointer to the filename is expected in the YX register pair (X is the low byte) and the length of the in A.

SETNAM never returns with an error.

SETLFS - Set parameters.

SETLFS will set the logical file number (A), device number (X), and secondary address (Y) for OPEN, LOAD or SAVE.

The logical file numner (or LFN for short) used with OPEN will be translated to the device number by CHKIN/CKOUT for BASIN, BSOUT or GETIN.

The device number may range from 0 to 255 (30 with standard KERNAL I/O). These are the commonly used device numbers:

~~~ 0: Keyboard 1: Tape 2: RS-232 3: Screen 4+5: Printers 6+7: Plotters 8-12: Disk drives 13-30: user defined – 31: TUNIX system calls ~~~
~~~

The secondary (SA) is sent to IEC devices. The c1541 DOS uses it to distinguish between open files like the LFN does. That's why the SA usually has the same value as the LFN in human- readable code.

SETLFS never returns with an error.

OPEN - Open logical file.

OPEN assigns the device and SA to the LFN that was specified when calling SETLFS. A filename must have been set using SETNAM.

The LFN of a successfully opened file can be used with CHKIN and CKOUT to set the input and output device for BASIN, BSOUT and GETIN.

TUNIX copies the filename to the IO area (255 byte maximum) and translates the LFN to a GLFN before calling OPEN.

OPEN clears the STATUS byte. On error, it returns with the carry flag set and one of these codes in A:

- 1: Too many files. 2: File already
- open. 4: File not found. Tape device
- only. 5: Device not present. 6: Not
- an input file. Also if logical file number is 0.
- 9: Illegal device number. Also if tape buffer is below $0200.

CHKIN - Set input device by LFN.

Takes the LFN in X and sets the input device of BASIN and GETIN to the one assigned to that LFN when OPEN was called.

One of these error codes may be returned:

- 3: file not open 5: device not present
- 6: file is not an input file

CKOUT - Set output device by LFN.

Takes the LFN in X and sets the output device of BASIN and GETIN to the one assigned to that LFN when OPEN was called.

One of these error codes may be returned:

- 3: file not open 5: device not present
- 7: file is not an output file

BASIN - Input character from channel

Reads a byte from the device selected by CHKOUT, CLRCN or CLALL and returns it in A. If no device was selected, BASIN reads from device #0.

When reading from the KERNAL keyboard, the input is buffered up to a length of 80 before or a carriage return is typed. Meanwhile, the cursor is blinking.

On error this routine returns with the carry flag set and additional flags in the STATUS zeropage locate (also see the READST system call).

GETIN - Input character from channel

Same as BASIN but returning 0 with the keyboard if no input is available.

BSOUT - output a character to channel

Writes a byte to the device selected by CHKOUT, CLRCN or CLALL, otherwise it writes to device #3.

NOTE: Care must be taken when using routine to send data to a serial device since data will be sent to all open output channels on the bus. Unless this is desired, all open output channels on the serial bus other than the actually intended destination channel must be closed by a call to CLOSE before.

On error BSOUT returns with the carry flag set and additional flags in the STATUS zeropage location (also see the READST system call).

The Y register is not destroyed.

CLRCN - Close default input and output files

Set the the input device to the key- board (0) and the output device to the screen (3), the only means to access standard I/O.

CLRCN never returns with an error.

CLOSE - Close logical file

Closes the LFN passed A. Never returns with an error.

CLALL - close all channels and files

When KERNAL version is called, the pointers into the open file table are reset, closing all files. Also the I/O channels selected with CHKIN/CKOUT are reset to the console, like CLRCN does.

With TUNIX, CLALL does not just drop all open files like the standard KERNAL but calls CLOSE on each driver, and the I/O channels are reset as well.

CLALL never returns with an error.

LOAD - load RAM from a device

Loads a block of bytes to memory and can also be used to verify them with a block in memory withot affecting it (by setting A to 1 instead of 0).

SETNAM and SETLFS must be called beforehand. The LFN used with SETLFS is ignored.

The destination address of the block is specified by either the first bytes of the file (if the SA is not 0). In that case the address is taken from the YX register pair (with X as the low byte). YX will return the address of the byte following the end of the block.

LOAD returns the same error codes as OPEN:

- 1: Too many files.
- 2: File already open.
- 4: File not found. Tape device only.
- 5: Device not present.
- 6: Not an input file. Also if device number is 0 (keyboard).
- 9: Illegal device number. Also if tape buffer is below $0200.

SAVE - save RAM to a device

Mirroring LOAD, this saves a block of memory to a device. SETNAM and SETLFS must have been called beforehand. The LFN used with SETLFN is ignored.

SAVE expects the zeropage location of the address of the memory block in A. The YX register contains the size of the memory block (X is the low byte).

SAVE may return one of these error codes if the carry flag is set:

- 1: Too many files.
- 2: File already open.
- 4: File not found. Tape device only.
- 5: Device not present.
- 7: Not an output file. Also if device number is 3 (screen).
- 9: Illegal device number. Also if tape buffer is below $0200.

System Call Device

Calling Convention

System calls are performed via opening a file on device #31. The first character of the filename denotes the command group, the second the command in that group. Argument bytes may follow. First arguments are passed as the secondary address to SETLFS.

After sending a system call via OPEN, an error code may be read using BASIN. It may be followed by return values.

On assembly level the carry flag tells if an error occured (carry set with code in accumulator instead of the return value). For both ca65 and cc65, system call libraries are around.

```
// Get process ID.
open (31, 31, 0, "P");
// (Never returns with an error.)
chkin (31);
pid = basin ();
```

Most system calls return an error code:

```
// Put process 1 to sleep.
open (31, 31, 1, "PS");
chkin (31);
err = basin ();
if (err)
    printf ("Process 1 already asleep.");
```

Some return an error code and return values:

```
// Fork
open (31, 31, 0, "PF");
chkin (31);
err = basin ();
if (!err) {
    id = basin ();
    if (!id)
        child_stuff ();
}
```

Returned Lists And Tables

Some functions either return a list of key/value pairs or a table. Both are lines of comma-separated values. Tables are headed with a list of column names. Key/value lists always have the keys in the first column.

Column names and keys are composed of upper case ASCII letters.

Values are either decimals, hexadecimal bytes or words of fixed length (2 or 4 characters) starting with a “$”, or strings surrounded by double quotes. Quotes inside strings are printed as quadruple quotes ("").

The order and number of keys may vary, depending on the function and the version of TUNIX. Assuming fixed layouts will jeopardize backwards- compatibility.

An example for a table:

```
ID,PID,NAME
0,0,"INIT"
1,0,"CONSOLE"
1,0,"RAMDISK"
1,0,"C1541"
1,0,"SLIP"
1,0,"UIP"
2,0,"BASIC"
3,0,"SOMETHING ""COOL"""
```

Code Example Notation

Lower case letters in system call file name examples are byte values. Letters in brackets are optional.

C examples are for the cc65 compiler suite and need to include header file ‘cbm.h’.

General

"": Schedule

A filename of length 0. Does nothing but give TUNIX the opportunity to switch to the next process without doing anything else. Like any of the other system calls it may or may not switch unless in single-tasking mode (command “SS”) or the current process has been put to sleep.

```
OPEN31,31,31,""
```

```
lda #31
tax
jsr SETFLS
lda #0
jsr SETNAM
jsr OPEN

; Library version:
jsr tunix_schedule
```

```
cbm_k_open (31, 31, 0, "");
```

“G$”: List general infomation

This function returns a list with these keys:

1. “NPROCS”: Number of processes.
2. “MPROCS”: Max. number of processes.
3. “NDRVS”: Number of drivers.
4. “MDRVS”: Max. number of drivers.
5. “NIOPAGES”: Number of IO pages.
6. “UIOPAGES”: Number of used IO pages.

“GM”: Set Mode (task-switching).

Enables or disables task-switching for the current process. It is disabled if the seconday address of the system call is 0. That shoud be done sparingly in a multi-tasking environment. Switches are scheduled before system calls return and are highly unpredictable.

Processes that did not free the screen will have that switched in for the time they are running.

“GX”: Shut down

Kills all processes and returns to BASIC.

Processes

“P$”: Process list

Returns a list with these fields:

. *ID: Process ID.* . PID: Parent rocess ID. . *NAME: Pathname if the executable.* . FLAGS: (R)running, (S) sleeping, (Z)ombie . *MEM: Allocated memory in bytes.* . NDRV: Number of registered drivers. . *NIOP: Number of allocated IO pages.* . LFNS: Number of logical file numbers.

Running processes are listed before sleeping ones, followed by zombies.

```
ID,PID,NAME,FLAGS,MEM,NDRV,NIOP,LFNS
1,0,"CONSOLE",R,49152,0,0,8
2,1,"BASIC",R,49152,0,0,3
0,0,"INIT",S,49152,0,0,2
```

“P”: Process ID

Returns current process ID.

“PI”: Process info

Secondary address: process ID.

Discloses all internal information about the process specified in the SA. Returns a key/value list with these keys:

- “ID”: Process ID.
- “FLAGS”: One of “BRSZ” (baby, running, sleeping, zombie).
- “EXITCODE”: Exit code. It should have no other value than 0 unless killed.
- “WAITING”: Processes waiting for exit.
- “MEMORY”: Banks assigned to RAM123 (TUNIX), IO23 (TUNIX), RAM123, BLK1, BLK2, BLK3 and BLK5.
- “#BANKS”: Number of extended memory banks.
- “BANKS”: List of extended memory banks.
- “STACK”: Last saved tack pointer.
- “LFNS”: Used LFNs.
- “GLFNS”: Assigned global LFNs.
- “IOPAGES”: Allocated IO pages.
- “DRIVERS”: Registered drivers.
- “DEVICES”: Device drivers.
- “SIGNALS”: Signals to receive.

“PNname”: Process name

The secondary address is the process ID.

Sets the name of process ‘p’ (zero- terminated string) if it follows the command. Returns the current name if none was set.

“PF”: Fork

Makes a copy of the current process with a new ID that is returned to the parent. For the child an ID of 0 is returned. Open files are shared by both processes and be closed by both.

Extended memory banks are inherited but not those that were banked in when the fork was initiated.

“PEname and args”: Execute

Replaces the current process by an executable in TUNIX format. But fear not: ‘runprg’ and ‘basic’ will also launch native programs.

“PX”: Exit

Secondary address: exit code.

Exits the current process with an exit code. All resources are freed but it remains on the process list until its parent exits or returns from a wait() for the exit code and its waiting list is empty (see wait()).

“PW”: Wait

Secondary address: process ID.

Waits for a process to exit and returns its exit code.

“PK[c]”: Kill

Secondary address: process ID.

Kills any process with an optional exit code (default is 255 if none is specified). Parents have to call wait() to make the process leave the system.

“PS”: Suspend

Secondary address: process ID.

Moves process to the sleeping list so that it won’t be executed after the next switch.

“PR”: Resume

Secondary address: process ID.

Moves a process to the running list. It will run again then the running list is restarted in the scheduler.

Extended Memory

Allocation and deallocation of extended memory banks happens fast at an almost constant speed. Banks may be placed anywhere except in the IO23 area.

“M$”: Get memory information

2. “USED”: Number of used banks.
3. “FREE”: Number of free banks
4. “RESERVED”: Number of reserved ones.
5. “FAULTY”: Number of faulty ones.
6. “TOTAL”: Number of banks.
7. “BANKSIZE”: Size of a bank in bytes.

“MA”: Allocate a bank

Allocates a bank and returns its ID. Returns with an error when out of memory.

“MF”: Free a bank

Secondary address: bank ID.

Frees a bank. All processes sharing it need to free it before it can be released back into the global pool. All banks of a process will be freed on exit.

Drivers

Processes may register named KERNAL I/O tables of drivers. Drivers process KERNAL I/O calls for a device they've been assigned to. A driver may be assigned to many devices of different processes simultaneously.

When a driver function is called, TUNIX only banks in BLK1 of the process that registered the driver to make up for the low hardware performance. The context of TUNIX is still that of the calling process. The driver must make sure that used zeropage locations are restored upon return to the caller.

Also, TUNIX translates logical file numbers to global logical file numners (GLFNs), so they won't collide with those used by other processes unless they got shared after a fork. An LFN is always translated to the same GLFN until the process exits.

"DL": Driver list

Returns a list with these fields:

TUNIX returns these fields:

- ID: Process ID.
- PID: Parent rocess ID.
- NAME: Pathname if the executable.
- FLAGS: (R)running, (S) sleeping, (Z)ombie
- MEM: Allocated memory in bytes.
- IOP: Number of allocated IO pages.

Example output:

```
ID,PID,NAME,FLAGS,MEM,IOP
0,0,INIT,S,49152,0
1,0,CONSOLE,R,49152,1
2,1,BASIC,R,49152,0
```

"DRvvname...": Register

The secondary address is unused.

Registers a named KERNAL I/O vector table. Global to all processes. It unregisters when the driver exits.

“DD”: Assign driver to device

Secondary address: process ID.

“DUd”: Unassign driver

Secondary address: process ID.

“DVd”: Get vectors of device

Secondary address: device number.

Used to overlay vectors by a another driver.

“DA” Allocate IO page

Returns the high byte of the page which will be present in all processes. Used to run interrupt handlers. Needs to be commited (to be portable) before use.

“DC”: Commit IO page

Secondary address: IO page.

Copies the page from the current pro- cess to all others.

“DF”: Free IO page

Frees the IO page specified by the secondary address.

Signals

Signals are asynchronuous calls of other processes’ functions. These functions, called *handlers*, can be *registered* by *signal type* (a byte value) or default handlers are called instead when a signal is *delivered*. The exact handler to call is determined at the time it is sent.

Signal type values can be chosen freely as long as they are not used by TUNIX itself. A byte of payload can be sent as well as the source of a signal is not tracked to allow post-mortem delivery (with the process ID of the sender gone).

Pending signals are queued so low-level interrupt handlers can send them with little overhead. Sending is cancelled if a signal of the same type is already on the queue. It is not checked if the target process provides a handler for the signal type. The signal is discarded on delivery when the handler is missing, so handlers can be dropped anytime.

“SSp”: Send signal to process

Secondary address: signal type.

“SRhh”: Register handler

Secondary address: signal type.

“SU”: Unregister handler

Secondary address: signal type.

Internals

Extended Memory Management

Globally, only free banks are tracked via ‘banks[]’ and ‘*free_bank’. The latter could be stored in banks[0] instead. Items in ‘bank_refs’ increment for each process that shares a bank.

Locally (per-process), only allocated banks are tracked in deque ‘lbanks/lbanksb’, starting with ‘first_lbank’.

Task Switching

There are two functions in TUNIX that perform a task switch: machdep_fork() and switch(). machdep_fork() copies the current process in its whole, including the stack and it’s pointer. When the new process is being switched to, it returns from fork_raw() like its parent but with its parent’s ID on the stack. That way fork() can tell if it deals with a returning parent or child.

switch() just exchanges the address space and thus the stack to contiue with the next process.

On the VIC-20 low memory must be copied twice for a task switch as the UltiMem expansion cannot access internal memory.

Scheduler

The running list is circular. Each process takes its turn in random order. When the process list is empty, process 0 is executed, no matter in what state.

Banking Scheme

TUNIX occupies the IO23 area to be around for all processes. BLK1 holds the kernel and global data and RAM123 contains per-process data. These blocks are used by processes as well. So when a programm

enters an I/O function, the TUNIX kernel and data are banked in to work and the process' banks need to come back on return. This seems to be solved easily by saving a process' bank configuration in its kernel tables but it makes calling I/O functions from within drivers impossible, as they are working in the caller's context but in their own address spaces. Calling I/O would overwrite the process' bank configuration with the driver's set, and desaster would be inevitable.

To get around this, the bank number of BLK1 is pushed on the stack and popped off on return. Other blocks are preserved the same way just in time.

Process Creation (fork)

A completey new process is created once when TUNIX starts: process 0. From then on, running processes are created by the fork() system call only. by duplicating the running process. fork() returns the ID of the new process called 'child process'.

Forking an already existing process spares initializations as the whole of the program is copied and has all information required to run. Merely shared resources need updated reference counts.

A freshly forked process that did not yet run is called a 'baby process' (a notion unique to TUNIX). Before it returns from fork(), the duplicated bank configuration on the stack, is overridden by the new banks and the child is marked as being a regular, running process.

Drivers

FIFO

For inter-process communication. Probably the most basic driver for TUNIX. Whatever is written to a device is coming out again in the same order. Writers have to wait when the buffer is full, readers when it is empty.

RAM disk

A RAM disk is essential for good system performance.