Unix Ontology: Space, First Mover and Time in Unix v6

Baptiste Mélès

CNRS, Archives Henri-Poincaré, Université de Lorraine

Autumn workshop: "Formalisms at the interface with machines, languages and systems", Bertinoro, Italy, 16-17 October 2018, ANR-17-CE38-0003-01

Outline

What is "Unix philosophy?"

- "Unix philosophy" seems to be an old and well-known topic
  - Ritchie and Thompson 1974, "The UNIX time-sharing system"
  - Doug McIlroy 1978, "Unix Time-Sharing System: Foreword"
  - Peter Salus 1994, *A Quarter-Century of Unix*
  - Eric Raymond 2003, *The Art of Unix Programming*

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

What is "Unix philosophy?"

- In this context, What does "Unix philosophy" mean?
- Basically, "Unix philosophy" means "engineering principles *around* Unix" (in the programming environment, in the community), such as:
  - small is beautiful
  - modularity
  - text streams
  - transparency
- "Philosophy" in a very informal sense ("basic principles")
- "Unix" in a very broad sense (not Unix kernel but the programming environment and the community)

What is "Unix philosophy?"

- Thus, Unix philosophy is neither Unix nor philosophy

Both Unix and philosophy

- My goal is to describe a "Unix philosophy" which be both Unix and philosophy
  - describe Unix' ontology in metaphysical terms
  - as it is encoded in Unix kernel's source code
- But first, what should "ontologies of operating systems" be?
  - Let us define "ontologies" and "operating systems."

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

What is an ontology?

Ontology Theory of "what there is," i.e of what, in a given context, can be:

- observed
- described
- handled

What is an operating system?

Operating system Program which:

- handles the material resources of the machine: processor, memories, inputs, outputs;
- provides the user with an interface allowing to interact with the machine.

This definition has two consequences with respect to ontology:

- the operating system hides the machine's ontology;
- it gives only access to a high-level ontology.

Source code

- The operating system's source code can then be read as a morphism from one ontology to another
  - it is written for a given machine or set of machines (e.g., for Unix v6: PDP11/40, PDP11/45 and PDP11/70), which have a proper ontology (processor, instructions, registers, operands, addresses, ...)
  - it defines another ontology (say, that of Unix: processes, files... or icons, windows etc.)
- The latter ontology is obviously definable in the former...
- ... but generally not uniquely: the high-level ontology may be (and generally is) machine-independent
- This ontology can (at least partially) be abstracted from its technical context.
- Metaphysics gives the abstract language allowing to describe ontologies in general.

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Interest for philosophers

- What is the interest for philosophers?
  - the ontology of an OS implies proper philosophical notions such as: object, act, time, space...
  - these notions are unambiguously defined in a formal language;
  - they can, at least locally, be compared with classical philosophical ontologies.

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Unix v6

- I will describe some bases of Unix v6's ontology, as it is built in the `main()` function
- Why v6?
  - mature Unix, with all essential notions of later Unices (Linux, *BSD...);
  - short and humanly readable source code;
  - well documented code (Lions).
- Now the question is: what kinds of things can be observed, described and handled in Unix v6?

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

From system calls to ontology

- Remember: ontology is the theory of what can be observed, described and handled within a given context
- Good hints can be found in an operating system's system calls
  - indeed, system calls are the handles which allow the user to communicate with the OS

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Unix system calls

- In Unix, the list is given in the file sysent.c

```
2904: /*
2905:  * This table is the switch used to transfer
2906:  * to the appropriate routine for processing a system call
2907:  * Each row contains the number of arguments expected
2908:  * and a pointer to the routine.
2909:  */
2910: int     sysent[]
2911: {
2912:         0, &nullsys,                    /*  0 = indir */
2913:         0, &rexit,                      /*  1 = exit */
2914:         0, &fork,                       /*  2 = fork */
2915:         2, &read,                       /*  3 = read */
2916:         2, &write,                      /*  4 = write */
2917:         2, &open,                       /*  5 = open */
2918:         0, &close,                      /*  6 = close */
```

Unix system calls (continued)

```
2919:           0, &wait,                       /*  7 = wait */
2920:           2, &creat,                      /*  8 = creat */
2921:           2, &link,                       /*  9 = link */
2922:           1, &unlink,                     /* 10 = unlink */
2923:           2, &exec,                       /* 11 = exec */
2924:           1, &chdir,                      /* 12 = chdir */
2925:           0, &gtime,                      /* 13 = time */
2926:           3, &mknod,                      /* 14 = mknod */
2927:           2, &chmod,                      /* 15 = chmod */
2928:           2, &chown,                      /* 16 = chown */
2929:           1, &sbreak,                     /* 17 = break */
2930:           2, &stat,                       /* 18 = stat */
2931:           2, &seek,                       /* 19 = seek */
2932:           0, &getpid,                     /* 20 = getpid */
2933:           3, &smount,                     /* 21 = mount */
2934:           1, &sumount,                    /* 22 = umount */
```

Unix system calls (continued)

```
2935:           0, &setuid,                     /* 23 = setuid */
2936:           0, &getuid,                     /* 24 = getuid */
2937:           0, &stime,                      /* 25 = stime */
2938:           3, &ptrace,                     /* 26 = ptrace */
2939:           0, &nosys,                      /* 27 = x */
2940:           1, &fstat,                      /* 28 = fstat */
2941:           0, &nosys,                      /* 29 = x */
2942:           1, &nullsys,                    /* 30 = smdate; in
2943:           1, &stty,                       /* 31 = stty */
2944:           1, &gtty,                       /* 32 = gtty */
2945:           0, &nosys,                      /* 33 = x */
2946:           0, &nice,                       /* 34 = nice */
2947:           0, &sslep,                      /* 35 = sleep */
2948:           0, &sync,                       /* 36 = sync */
2949:           1, &kill,                       /* 37 = kill */
2950:           0, &getswit,                    /* 38 = switch */
```

Unix system calls (continued)

```
2951:           0, &nosys,              /* 39 = x */
2952:           0, &nosys,              /* 40 = x */
2953:           0, &dup,                /* 41 = dup */
2954:           0, &pipe,               /* 42 = pipe */
2955:           1, &times,              /* 43 = times */
2956:           4, &profil,             /* 44 = prof */
2957:           0, &nosys,              /* 45 = tiu */
2958:           0, &setgid,             /* 46 = setgid */
2959:           0, &getgid,             /* 47 = getgid */
2960:           2, &ssig,               /* 48 = sig */
2961:           0, &nosys,              /* 49 = x */
2962:           0, &nosys,              /* 50 = x */
[...]
2976: };
```

System calls

- What do those system calls talk about?

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Processes

- Some of them refer to processes:

  `exec`, `fork` create processes
  `setuid`, `getuid` handle users' rights on processes
  `nice` set the priority level of processes
  `kill` send signals to processes

- Processes as acts
  - every process is an act
  - in Unix, everything that is an act is a process

Files

- Other system calls refer to files:

  `unlink` erase a file
  `chdir` change directory
  `chmod` modify the rights on a file
  `chown` change the file's owner

- Files as objects
  - every file is an object
  - in Unix, everything that is an object is a file (including "special files")

Is that all?

- Unix ontology seems to include at least:
  - acts (processes)
  - objects (files)
- Is that all?
  - Tanenbaum and Woodhull, p. 15: "The MINIX system calls fall roughly in two broad categories: those dealing with processes and those dealing with the file system."

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Time

- But not only! Some other system calls refer to time:
  - `stime`
  - `gtime`
- In Linux 2.2.14 (2000) [Bovet and Cesati 2001, ch. 5]:

  `time`, `ftime`, `gettimeofday` get date and time;
  `settimer`, `alarm` send periodical signals;
  `adjtimex` synchronise clocks

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

What we learn from system calls

- Unix ontology contains at least:
  - processes as acts
  - files as objects
  - time
- Now, let us examine how this ontology is built: let us look at the `main()` function (in `main.c`)

The main() function

- This code is executed at the very beginning:

```
1532: /*
1533:  * Initialization code.
1534:  * Called from m40.s or m45.s as
1535:  * soon as a stack and segmentation
1536:  * have been established.
```

The main() function (continued)

- Informal specification:

```
1537:  * Functions:
1538:  *      clear and free user core
1539:  *      find which clock is configured
1540:  *      hand craft 0th process
1541:  *      call all initialization routines
1542:  *      fork - process 0 to schedule
1543:  *           - process 1 execute bootstrap
1544:  *
1545:  * panic: no clock -- neither clock responds
1546:  * loop at loc 6 in user mode -- /etc/init
1547:  *      cannot be executed.
1548:  */
1549:
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

The main() function (continued)

- A few variables:

```
1550: main()
1551: {
1552:         extern schar;
1553:         register i, *p;
1554:
```

The main() function (continued)

- Clear memory:

```
1555:           /*
1556:            * zero and free all of core
1557:            */
1558:
1559:           updlock = 0;
1560:           i = *ka6 + USIZE;
1561:           UISD->r[0] = 077406;
1562:           for(;;) {
1563:                   UISA->r[0] = i;
1564:                   if(fuibyte(0) < 0)
1565:                           break;
1566:                   clearseg(i);
1567:                   maxmem++;
1568:                   mfree(coremap, 1, i);
1569:                   i++;
1570:           }
```

The main() function (continued)

- Some PDP11/70-specific lines:

```
1571:           if(cputype == 70)
1572:           for(i=0; i<62; i=+2) {
1573:                   UBMAP->r[i] = i<<12;
1574:                   UBMAP->r[i+1] = 0;
1575:           }
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

The main() function (continued)

- Free memory:

```
1576:           printf("mem = %l\n", maxmem*5/16);
1577:           printf('RESTRICTED RIGHTS\n\n');
1578:           printf('Use, duplication or disclosure is subject
1579:           printf('restrictions stated in Contract with Weste
1580:           printf('Electric Company, Inc.\n');
1581:
1582:           maxmem = min(maxmem, MAXMEM);
1583:           mfree(swapmap, nswap, swplo);
1584:
```

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

The main() function (continued)

- Initialize the process 0:

```
1585:           /*
1586:            * set up system process
1587:            */
1588:
1589:           proc[0].p_addr = *ka6;
1590:           proc[0].p_size = USIZE;
1591:           proc[0].p_stat = SRUN;
1592:           proc[0].p_flag =| SLOAD|SSYS;
1593:           u.u_procp = &proc[0];
1594:
```

The main() function (continued)

- Determine clock:

```
1595:           /*
1596:            * determine clock
1597:            */
1598:
1599:           UISA->r[7] = ka6[1]; /* io segment */
1600:           UISD->r[7] = 077406;
1601:           lks = CLOCK1;
1602:           if(fuiword(lks) == -1) {
1603:                   lks = CLOCK2;
1604:                   if(fuiword(lks) == -1)
1605:                           panic("no clock");
1606:           }
1607:           *lks = 0115;
1608:
```

The main() function (continued)

- Initialize the filesystem:

```
1609:           /*
1610:            * set up 'known' i-nodes
1611:            */
1612:
1613:           cinit();
1614:           binit();
1615:           iinit();
1616:           rootdir = iget(rootdev, ROOTINO);
1617:           rootdir->i_flag =&  ILOCK;
1618:           u.u_cdir = iget(rootdev, ROOTINO);
1619:           u.u_cdir->i_flag =&  ILOCK;
1620:
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

The main() function (continued)

- Make init process:

```
1621:           /*
1622:            * make init process
1623:            * enter scheduling loop
1624:            * with system process
1625:            */
1626:
1627:           if(newproc()) {
1628:                   expand(USIZE+1);
1629:                   estabur(0, 1, 0, 0);
1630:                   copyout(icode, 0, sizeof icode);
1631:                   /*
1632:                    * Return goes to loc. 0 of user init
1633:                    * code just copied out.
1634:                    */
1635:                   return;
1636:           }
```

The main() function (continued)

- Enter scheduling loop:

```
1637:           sched();
1638: }
```

Space, first mover and time

- In ontological terms, the `main()` function successively defines:
  - space
  - first mover
  - time
- Let us begin with space!

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

Memory as a *terra incognita*

- Before running the `main()` function, Unix ignores almost everything about memory:
  - its content is unpredictable;
  - its size is indefinite.
- Space is a big *terra incognita*.
- Unix just knows a few things:
  - where to start, viz. right after the OS kernel image;
  - the linear topology of memory: blocks are all lined up, accessible by incrementation of one variable (`i++`).
- Unix will use this knowledge to domesticate space.

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Clear and free memory

- The first task in the `main` function is supposed to be:

```
1532: /*
[...]
1537:  * Functions:
1538:  *      clear and free user core
[...]
1548:  */
```

- Actually, the code does not only clear and free *core* memory but also *swap* memory.

Identify the first core memory block

```
1550: main()
1551: {
1552:         extern schar;
1553:         register i, *p;
1554:
1555:         /*
1556:          * zero and free all of core
1557:          */
1558:
1559:         updlock = 0;
1560:         i = *ka6 + USIZE;
```

`*ka6` the block following the OS in memory

`USIZE` (= 16 in `systm.h`) blocks reserved for a file descriptor

Free all following blocks

```
1562:           for(;;) {
1563:                   UISA->r[0] = i;
1564:                   if(fuibyte(0) < 0)
1565:                           break;
1566:                   clearseg(i);
1567:                   maxmem++;
1568:                   mfree(coremap, 1, i);
1569:                   i++;
1570:           }
```

`fuibyte()` try to read the first word of the block `i` (= test to know whether `i` is an existing place in memory)

`clearseg(i)` clear the `i` block

`maxmem++` increments the (potential) maximal size allowed per process

`mfree(coremap, 1, i)` declares `coremap`'s block `i` as available

`i++` go to the next block

Core memory as a terra cognita

Now core memory is a terra cognita

- Size is known:

```
1576:           printf("mem = %l\n", maxmem*5/16);
1577:           printf('RESTRICTED RIGHTS\n\n');
1578:           printf('Use, duplication or disclosure is subject
1579:           printf('restrictions stated in Contract with Weste
1580:           printf('Electric Company, Inc.\n');
```

- The content is known:

  from `0` to `*ka6 - 1` kernel image
  from `*ka6` to `*ka6 + USIZE - 1` space allocated to process 0's descriptor
  from `*ka6 + USIZE` to the end finite free space

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

Maximal space per process

- The maximal space per process can be set:

```
1582:           maxmem = min(maxmem, MAXMEM);
```

`MAXMEM` = 64*32 in `param.h`

Clear and free swap memory

- Clear and free the whole swap memory:

```
1583:           mfree(swapmap, nswap, swplo);
```

`swapmap` swap memory
`nswap` number of blocks
`swplo` index of the first block

A new space

- Memory space, as seen by the OS, is now different from that of the underlying machine

	machine space	Unix space
space	topological knowledge (distinguished point, linear topology)	metric knowledge
size	indefinite	finite
content	indeterminate	clear

- Space is now under control: there is a world, in which things and actions will take place.
- Let us now describe how actions can take place therein.

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

The need for a first mover

- In Unix, actions are processes
- Every action is put in motion (`forked`) by a preceding action.
- Processes form a tree.
- Which leads to the following question: does this tree have a root?
- In Unix like in Aristotle, there is no actual infinite: one needs a "first mover."

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Process 0 as a first mover

- Unix' main task is now to initialize the process 0, which is the fundamental process (scheduler)
- We readers of Unix' source code know that this process is created (viz. by the `main` function)
- But from the viewpoint of the OS, it is uncreated
  - it is an act (a process)
  - it puts all other acts in motion
  - there is no way to observe, describe or handle its creator within Unix

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Process 0's descriptor

- Process 0's descriptor is written in memory, right after the OS (`USIZE` blocks are reserved for it)

```
1589:           proc[0].p_addr = *ka6;
1590:           proc[0].p_size = USIZE;
```

Process 0's properties

```
1591:           proc[0].p_stat = SRUN;
1592:           proc[0].p_flag =| SLOAD|SSYS;
```

- The process 0 is set as "ready to run" (`SRUN`)
- It receives two flags:

  `SLOAD` its descriptor is in core memory

  `SSYS` this process must never be moved to the swap memory

- Finally, the process descriptor is associated with its user's descriptor:

```
1593:           u.u_procp = &proc[0];
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Unix ontology at this point

At this point, Unix ontology includes:

1. a metric and finite space;
2. an act which is a first mover (scheduler) and possess all properties of standard Unix "acts" (viz. processes);
3. now Unix will define its proper notion of time.

PDP11 clocks

Unix v6 was exclusively written for the PDP11, which could have one of two clocks:

1. the frequency line clock KW11-L: "The KW11-L accurately divides time into intervals for more efficient use of PDP-11 computer time. The intervals are determined by the line frequency, either 50 or 60 Hz. The accuracy of the clock period is that of the frequency source." (KW11L Handbook, July 1974, p. 2-1)
2. the programmable real-time clock KW11-P, with a crystal oscillator, a counter and a holding register
   - great accuracy
   - tunable count rate: 100 kHz, 10 kHz, 60 Hz...

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Clock addresses

- Those clocks have different physical addresses:
  - frequency line clock KW11-L: 777546;
  - programmable clock KW11-P: 772540.
- Both clocks are defined by their addresses in `main.c` (up to transposition from 0160000-0177777 to 0760000-0777777):

```
1509: #define CLOCK1  0177546
1510: #define CLOCK2  0172540
```

Determine clock

Now, which clock is present in the machine?

- First, let the pointer `lks` point at `CLOCK1`'s address:

```
1601:           lks = CLOCK1;
```

- The system tries to read the value stored at this address (`fuiword()`), and otherwise points at `CLOCK2`'s address:

```
1602:           if(fuiword(lks) == -1) {
1603:                   lks = CLOCK2;
[...]
1606:           }
```

Panic: no clock

- If it fails again, there is a panic exit:

```
1604:                   if(fuiword(lks) == -1)
1605:                           panic("no clock");
```

- Panic exits are very rare in Unix! Time is one of the rare indispensable conditions
- Now, if at least one of the clocks works, then its value is initialized (do not ask me why it is set to 0115 = 77!).

```
1607:           *lks = 0115;
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Unix proper time

- Unix time is now different from that of the machine

	machine	Unix
time scale	10^{-9} second	10^{-2} second
activity	elementary operations	processes

- From now on, the system will receive regular pulses of the clock
  - those interrupts will time Unix activity

"All the clock hardware does is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver." (Tanenbaum and Woodhull 1997, p. 224)

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Interrupts

- At each interrupt, the function `clock()` is executed (file `clock.c` in Unix, `kernel/time.c` and `kernel/sched.c` in Linux)
  - this process is always unexpected: it interrupts either the process 1, or another previous instance of the clock driver process, or any other process; the interrupted process is denoted by the name `ps`
  - the priority of the function `clock()` is the highest one (6). No other device has such a priority: nothing is more urgent than time (Lions 1996, p. 11-1a). The driver can thus reset the clock almost immediately with the command

```
3730:           /*
3731:            * restart clock
3732:            */
3733:
3734:           *lks = 0115;
```

already present in the `main()` function.

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Priority

- Tasks divide into two parts:
  1. high-priority tasks, which are executed immediately
  2. low-priority tasks... which can wait

High-priority tasks

- The system executes some rare functions which are absolutely high-priority, like sending characters to the terminal (`ttrstrt`).
  - These functions and their arguments are registered in the structure array `callout`, defined in `systm.h`
  - This structure defines the `c_time` number of clock ticks which every function must wait before being executed:

High-priority tasks

```
0253: /* The callout structure is for a routine
0254:  * arranging to be called by the clock interrupt
0255:  * (clock.c) with a specified argument,
0256:  * within a specified amount of time.
0257:  * It is used, for example, to time tab delays
0258:  * on teletypes. */
0259:
0260: struct  callo
0261: {
0262:         int     c_time;         /* incremental time */
0263:         int     c_arg;          /* argument to routine */
0264:         int     (*c_func)();    /* routine */
0265: } callout[NCALL];
```

Clock driver

- The clock driver is mostly dedicated to the treatment of this prioritary functions:

```
3742:           /*
3743:            * callouts
3744:            * if none, just return
3745:            * else update first non-zero time
3746:            */
3747:
3748:           if(callout[0].c_func == 0)
3749:                   goto out;
3750:           p2 = &callout[0];
3751:           while(p2->c_time<=0 && p2->c_func!=0)
3752:                   p2++;
3753:           p2->c_time--;
3754:
```

Clock driver (continued)

```
3755:           /*
3756:            * if ps is high, just return
3757:            */
3758:
3759:           if((ps&0340) != 0)
3760:                   goto out;
3761:
```

Clock driver (continued)

```
3762:           /*
3763:            * callout
3764:            */
3765:
3766:           spl5();
```

Clock driver (continued)

```
3767:           if(callout[0].c_time <= 0) {
3768:                   p1 = &callout[0];
3769:                   while(p1->c_func != 0 && p1->c_time <= 0)
3770:                           (*p1->c_func)(p1->c_arg);
3771:                           p1++;
3772:                   }
3773:                   p2 = &callout[0];
3774:                   while(p2->c_func = p1->c_func) {
3775:                           p2->c_time = p1->c_time;
3776:                           p2->c_arg = p1->c_arg;
3777:                           p1++;
3778:                           p2++;
3779:                   }
3780:           }
[...]
3787: out:
```

High-priority functions

- First check that the todo list is not empty:

```
3748:           if(callout[0].c_func == 0)
3749:                   goto out;
```

- Then decrement the number of ticks that the next function must wait, if it not already time to execute it:

```
3750:           p2 = &callout[0];
3751:           while(p2->c_time<=0 && p2->c_func!=0)
3752:                   p2++;
3753:           p2->c_time--;
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

High-priority functions

- Check if the interrupted process is not prioritary:

```
3755:           /*
3756:            * if ps is high, just return
3757:            */
3758:
3759:           if((ps&0340) != 0)
3760:                   goto out;
[...]
3787: out:
```

High-priority functions

- Then switch to the priority level 5:

```
3766:           spl5();
```

High-priority functions

- Now, all the functions which must wait 0 tick are executed, and then replaced by less urgent tasks:

```
3767:           if(callout[0].c_time <= 0) {
3768:                   p1 = &callout[0];
3769:                   while(p1->c_func != 0 && p1->c_time <= 0)
3770:                           (*p1->c_func)(p1->c_arg);
3771:                           p1++;
3772:                   }
3773:                   p2 = &callout[0];
3774:                   while(p2->c_func = p1->c_func) {
3775:                           p2->c_time = p1->c_time;
3776:                           p2->c_arg = p1->c_arg;
3777:                           p1++;
3778:                           p2++;
3779:                   }
3780:           }
```

High-priority functions

- Still in priority 5 or 6, the driver executes other tasks relative to time:
  - increments the time of the `ps` process `u_utime` (user mode) or `u_stime` (kernel mode)

```
3788:           if((ps&UMODE) == UMODE) {
3789:                   u.u_utime++;
[...]
3792:           } else
3793:                   u.u_stime++;
```

High-priority functions

- One of the high-priority tasks is to update official time.
- Unix time is not made of seconds but of interrupts (60 by second, in `param.h`)

```
0128: /* tunable variables */
[...]
0147: #define HZ      60                  /* Ticks/second of the clo
```

- Now, in Linux, for most machines: 100 interrupts by second
  - this is very slow! compare with 10^{-9} second for the processor's operations
- Unix time allows to update official time, and not the reverse

Update official time

- Check that the process `ps` does not take precedence over what follows:

```
3798:                   if((ps&0340) != 0)
3799:                           return;
```

- If so, then update time: if the number of ticks, once incremented, exceeds the value `HZ` (number of ticks by second), then one more second has elapsed since 1st January 1970 at midnight:

```
3797:           if(++lbolt >= HZ) {
3798:                   if((ps&0340) != 0)
3799:                           return;
3800:                   lbolt =- HZ;
3801:                   if(++time[1] == 0)
3802:                           ++time[0];
```

(read `int time[2]` as if it were `long int time`!)

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

High-priority functions

- High-priority tasks are now finished:
  - high-priority functions (`callout`);
  - update time statistics;
  - update official clock.
- Everything that follows is low-priority and can be executed in priority 1:

```
3803:                   spl1();
```

Low-priority tasks

- For instance, several executions of the not prioritary parts of the clock driver may be waiting
  - no problem! high-priority parts will always be executed on time
  - the inconveniences are harmless:
    - loss of console reactivity
    - obsolete information about inactive processes
- Wake up some low-priority processes (some of them at a given time, others every 4 seconds):

```
3804:                   if(time[1]==tout[1] && time[0]==tout[0])
3805:                           wakeup(tout);
3806:                   if((time[1]&03) == 0) {
3807:                           runrun++;
3808:                           wakeup(&lbolt);
3809:                   }
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Low-priority tasks

- Update statistics of processes:

```
3810:                   for(pp = &proc[0]; pp < &proc[NPROC]; pp++
3811:                   if (pp->p_stat) {
3812:                           if(pp->p_time != 127)
3813:                                   pp->p_time++;
3814:                           if((pp->p_cpu & 0377) > SCHMAG)
3815:                                   pp->p_cpu =- SCHMAG; else
3816:                                   pp->p_cpu = 0;
3817:                           if(pp->p_pri > PUSER)
3818:                                   setpri(pp);
3819:                   }
```

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Priority

- Distinction between two kinds of tasks:
  - prioritary tasks
    - restart clock
    - update console
    - execute prioritary functions (typically relative to the terminal)
    - update the information relative to the interrupted process
    - update the system clock
  - all other tasks can wait

Unix proper time

- So, what is time in Unix?
  - it is marked by clock interrupts
  - this rhythm is rather slow (10^{-2} second), compared to that of the machine
  - this rhythm times high level acts (processes), by contrast with elementary operations of the machine
- Thus, time is still the rhythm of activity, but now it is a high-level activity

Conclusion

- Unix' `main()` function gives the bases of Unix ontology:
  1. memory as a metric space;
  2. an alleged first mover, which schedules activity;
  3. a proper time designed for processes (high-level activity).
- All of three replace the machine's ontology:
  1. memory as a topological space;
  2. no first mover;
  3. the time of elementary operations.

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR

References

Aristote (2014), *Physique*, in Pellegrin P. (éd.), *Aristote. Œuvres complètes*, Paris, Flammarion.
Bovet D. et Cesati M. (2001), *Le Noyau Linux*, trad. J. Cornavin et E. Chaput, Paris, O'Reilly.
DEC (1972a), *PDP11/40 Processor Handbook*, Digital Equipment Corporation.
DEC (1972b), *KW11-P Programmable Real-Time Clock Manual*, Digital Equipment Corporation.
DEC (1973), *KW11-L Line Time Clock Manual*, Digital Equipment Corporation.
Kernighan B. and Ritchie D. (2004), *Le Langage C. Norme ANSI*, Paris, Dunod.
Linux (2000), kernel 2.2.14, `http://www.kernel.org`.
Linux (2017), kernel 4.11.8, `http://www.kernel.org`.
...

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

References (continued)

Lions J. (1996), *Lions Commentary on UNIX 6th Edition with Source Code*, Charlottesville, Peer-to-Peer Communications.

MINIX (1997), in (Tanenbaum et Woodhull 1997), appendix A, p. 521-903.

von Neumann J. (1945), « First Draft of a Report on the EDVAC », Contract No. W–670–ORD–4926 Between the United States Army Ordnance Department and the University of Pennsylvania, Moore School of Electrical Engineering, University of Pennsylvania, 30 juin 1945, section 2.

Tanenbaum A. and Woodhull A. (1997), *Operating Systems. Design and Implementation*, Upper Saddle River, Prentice Hall.

Unix (1975), in (Lions 1996), p. 1-90.

PROJET FINANCÉ PAR L'ANR
ANR
PROJECT FUNDED BY THE ANR