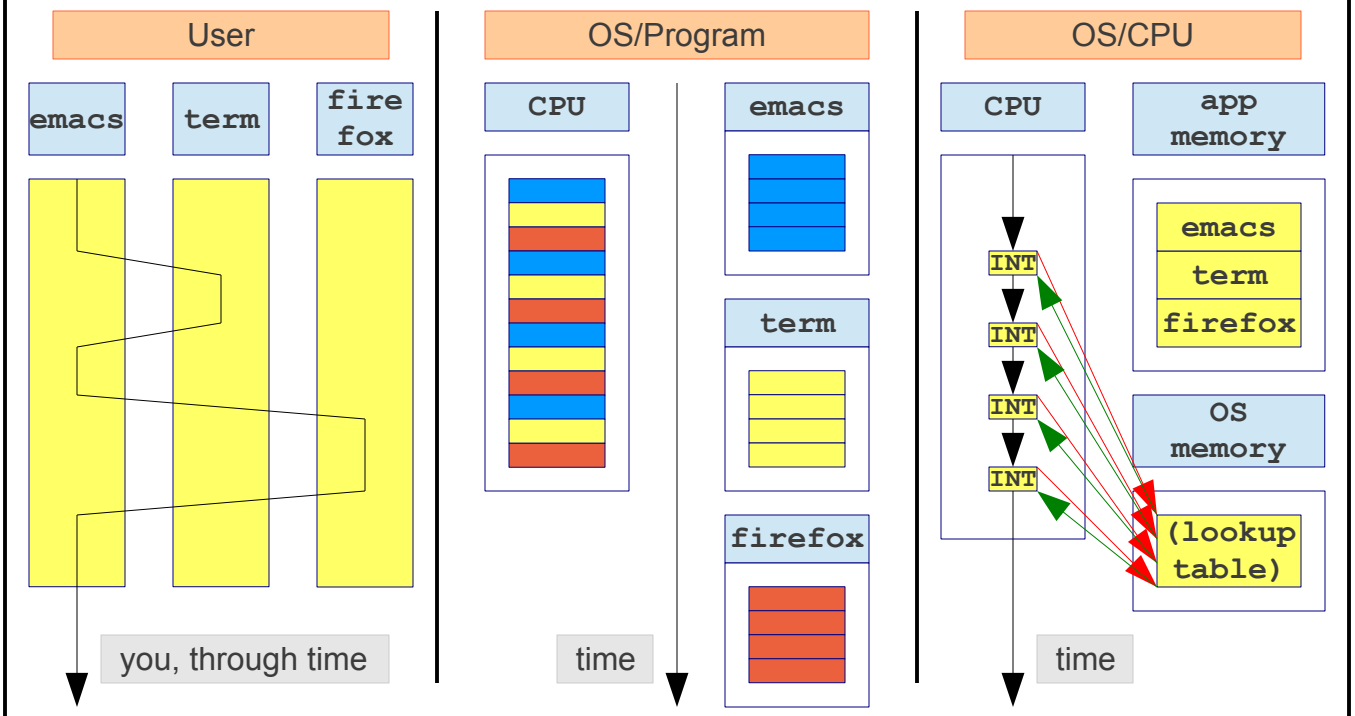
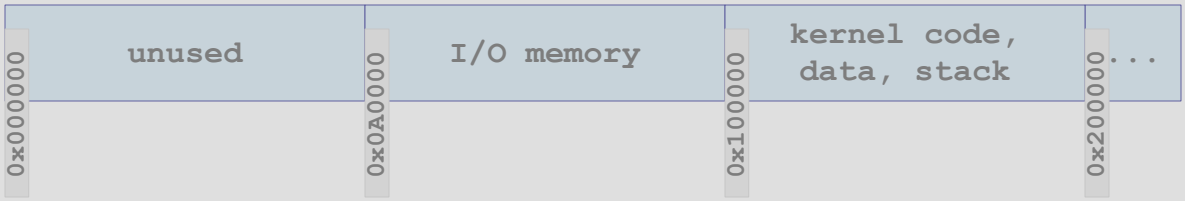
Mini 1	Context switching	1/8

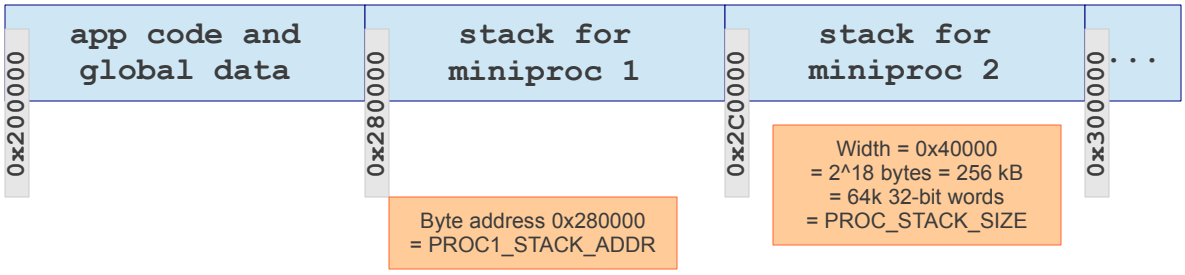


```
// MINIPROCOS MEMORY MAP
//
// +--------------------------+-------------+-----------------------------+-/
// |    Base Memory (640K)    |  I/O Memory | Kernel              Kernel  |
// |        (unused)          |             | Code + Data          Stack  |
// +--------------------------+-------------+-----------------------------+-/
// 0                       0xA0000       0x100000                      0x200000
//
//         /-+----------------+-----------+-----------+-----------+---/
//           |   Application  | Miniproc 1 | Miniproc 2 | Miniproc 3 |
//           | Code + Globals |      Stack |      Stack |      Stack |
//         /-+----------------+-----------+-----------+-----------+---/
//       0x200000          0x280000     0x2C0000     0x300000     0x340000
//                            |            |            |
//                    PROC1_STACK_ADDR     |     PROC1_STACK_ADDR
//                                         |    + 2*PROC_STACK_SIZE
//                                         |
//                                  PROC1_STACK_ADDR
//                                + PROC_STACK_SIZE
//
// There is also a shared 'cursorpos' variable, located at 0x60000 in the
// kernel's data area.  (This is used by 'app_printf' in mpos-app.h.)
```

Don't care about right now



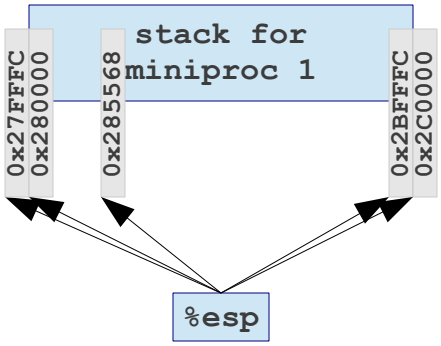
DO care about right now



Byte address 0x280000
= PROC1_STACK_ADDR

Width = 0x40000
= 2^18 bytes = 256 kB
= 64k 32-bit words
= PROC_STACK_SIZE

Mini 1	Miniproc stack (from `copy_stack()`)	4/8



Each process stores its own copy of `%esp` (and also all other registers).

0x280000 is a memory address, so e.g.

```
int x = *((int *) 0x280000);
```

will set x = the contents of the word at memory location 0x280000.

`%esp = 0x2C0000`: empty stack
`%esp = 0x2BFFFC`: stack has 4 bytes = 1 word
`%esp = 0x285568`: stack has 60070 words
`%esp = 0x280000`: full stack
`%esp = 0x27FFFF`: stack overflow in proc 1

A stack overflow in proc 1 will overwrite the app data.
An overflow in proc 2 will overwrite proc 1's stack.

1. `pid_t = int32_t = long` *(defined in types.h)*
   – used for process IDs

2. `procstate_t`: the available process states *(defined in mpos-kern.h)*

   1. `P_EMPTY`: the process is empty
      – we can start a new process in this location

   2. `P_RUNNABLE`: there is a process here
      – and if we want, we can context-switch to it

   3. `P_BLOCKED`: the process is blocked
      – we need to unblock it to use it
      – we change the state to `P_RUNNABLE` when it's no longer blocked

   4. `P_ZOMBIE`: used to mark for cleanup by sys_wait()

3. `process_t`: process descriptor *(defined in mpos-kern.h)*
   – there is a `process_t` instantiated for each process

   Contents:

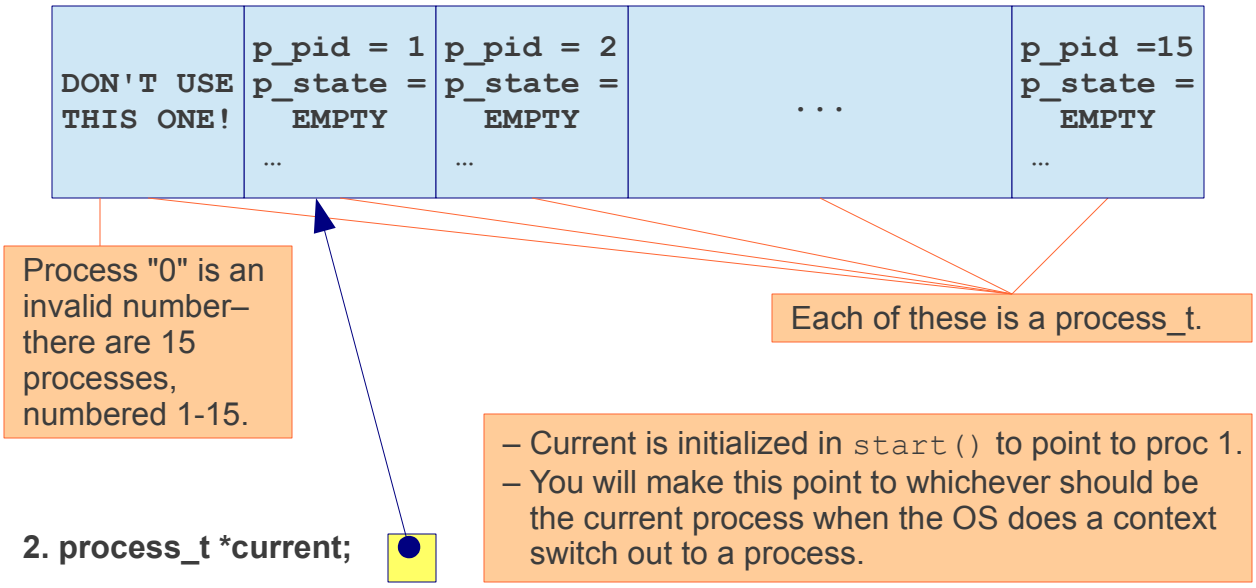
   1. `pid_t pid`: the ID of the corresponding process
      – we set this to its `proc_array[]` index in `mpos-kern.c:start()`.

   2. `registers_t p_registers`: the state of the registers for this process.
      – proc_array[1].p_registers.reg_eax stores what process 1 expects its `%eax` to be when it starts running again.

   3. `procstate_t p_state`: the state: `P_EMPTY`, …, `P_ZOMBIE`

   4. `int p_exit_status`: we set this in the `INT_SYS_EXIT` handler for other processes to read in `INT_SYS_WAIT`.

1. **static process_t proc_array[NPROCS];** *// NPROCS = 16, mpos.h*

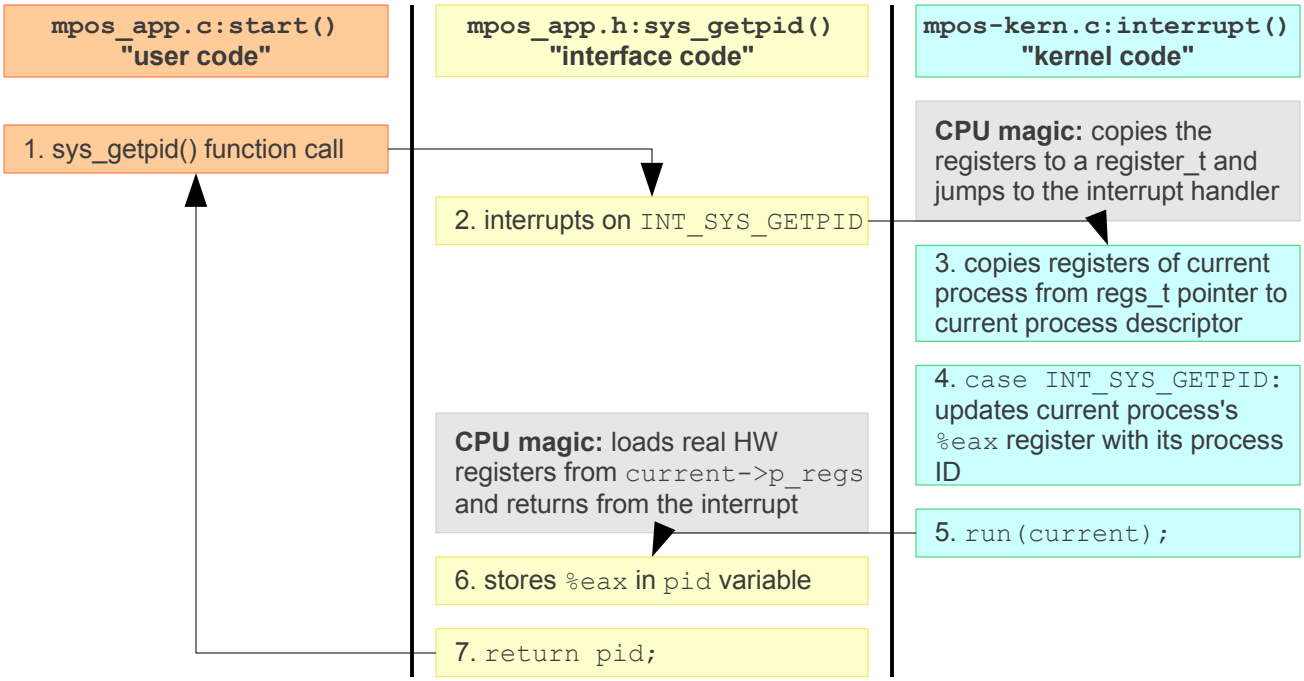


Process "0" is an invalid number– there are 15 processes, numbered 1-15.

Each of these is a process_t.

2. process_t *current;

– Current is initialized in `start()` to point to proc 1.
– You will make this point to whichever should be the current process when the OS does a context switch out to a process.


mpos_app.c:start()
"user code"
mpos_app.h:sys_getpid()
"interface code"
mpos-kern.c:interrupt()
"kernel code"
1. sys_getpid() function call
2. interrupts on INT_SYS_GETPID
CPU magic: copies the registers to a register_t and jumps to the interrupt handler
3. copies registers of current process from regs_t pointer to current process descriptor
4. case INT_SYS_GETPID: updates current process's %eax register with its process ID
5. run(current);
CPU magic: loads real HW registers from current->p_regs and returns from the interrupt
6. stores %eax in pid variable
7. return pid;