TOS ++

Objectives

- Enhance TOS:
  - Add malloc(), free()
  - Overlapping windows
- Window Manager
- Keyboard support
- TOS shell
- Running TOS on real hardware

Accessing High Memory

- So far, TOS only uses the first MB of RAM. We need more memory for some advanced TOS features.
- Increasing emulated RAM to 8 MB
  via `.bochsrc`:
  `memory: guest=8, host=2`
- Is this sufficient? No! Because of some arcane architectural details of the early IBM PCs, more work is needed to access RAM beyond the first MB.

A20 Address Line

- Early x86 CPUs had 20 address lines. Therefore maximum RAM size was 1 MB (2^{20} = 1 MB)
- Some programs (e.g., MS-DOS) generated addresses higher than 1 MB and relied on a “wrap-around” (i.e., masking of the 20^{th} address bit).
- The 80286 added more address line and the lack of masking of the A20 line caused these programs to fail.
- To allow backward compatibility of such legacy applications, an A20 gate was introduced on the motherboard. It could be enabled (do not mask A20) or disabled (mask A20).
- An operating system wishing to access high memory above 1 MB need to enable the A20 gate.

Enabling A20 Gate

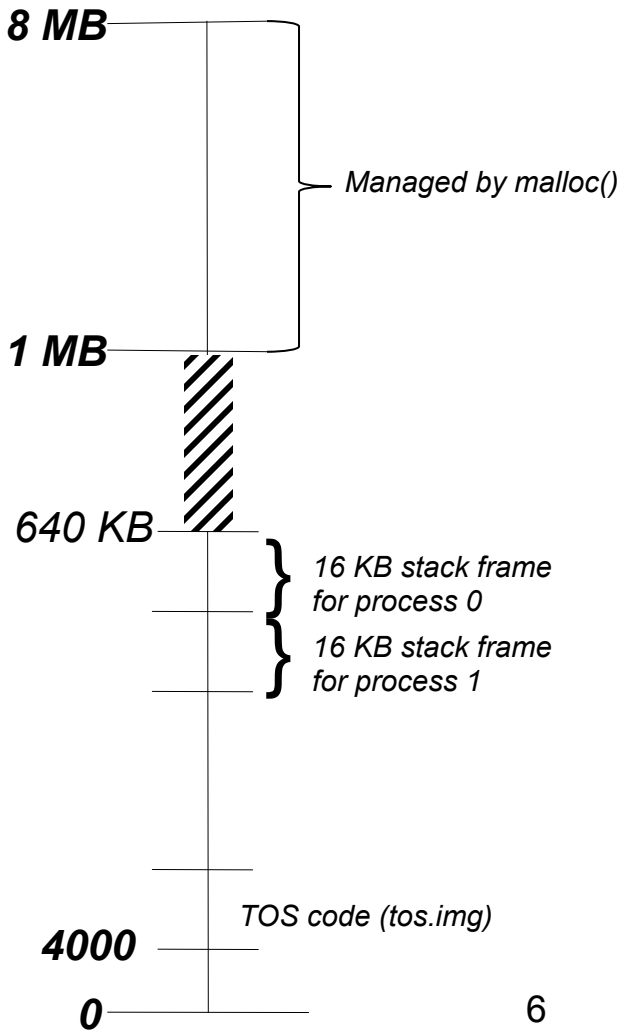
```
enablea20:
    call empty_8042
    mov al,0xd1 ; command write
    out 0x64,al
    call empty_8042
    mov al,0xdf ; A20 on
    out 0x60,al
    call empty_8042
    ret

empty_8042:
    in al,0x64
    test al,2
    jnz empty_8042
    ret
```

X86 subroutine in `tos/tools/boot/utils.s`

Dynamic Memory Management

- Now that TOS can access high memory, next step is to support dynamic memory management via `malloc()` and `free()`.
- Note that those two functions are implemented in C: `tos/kernel/malloc.c`
- `malloc()` keeps track of fragmentation via its own data structure. When a caller requests memory, `malloc()` will find a sufficiently sized region of memory.
- `malloc()` uses a kernel function `sbrk()` to request more memory. If no more memory is available, TOS will panic with `assert(0)`
- `free()` simply returns a region of memory back to the free-list managed by `malloc()`.



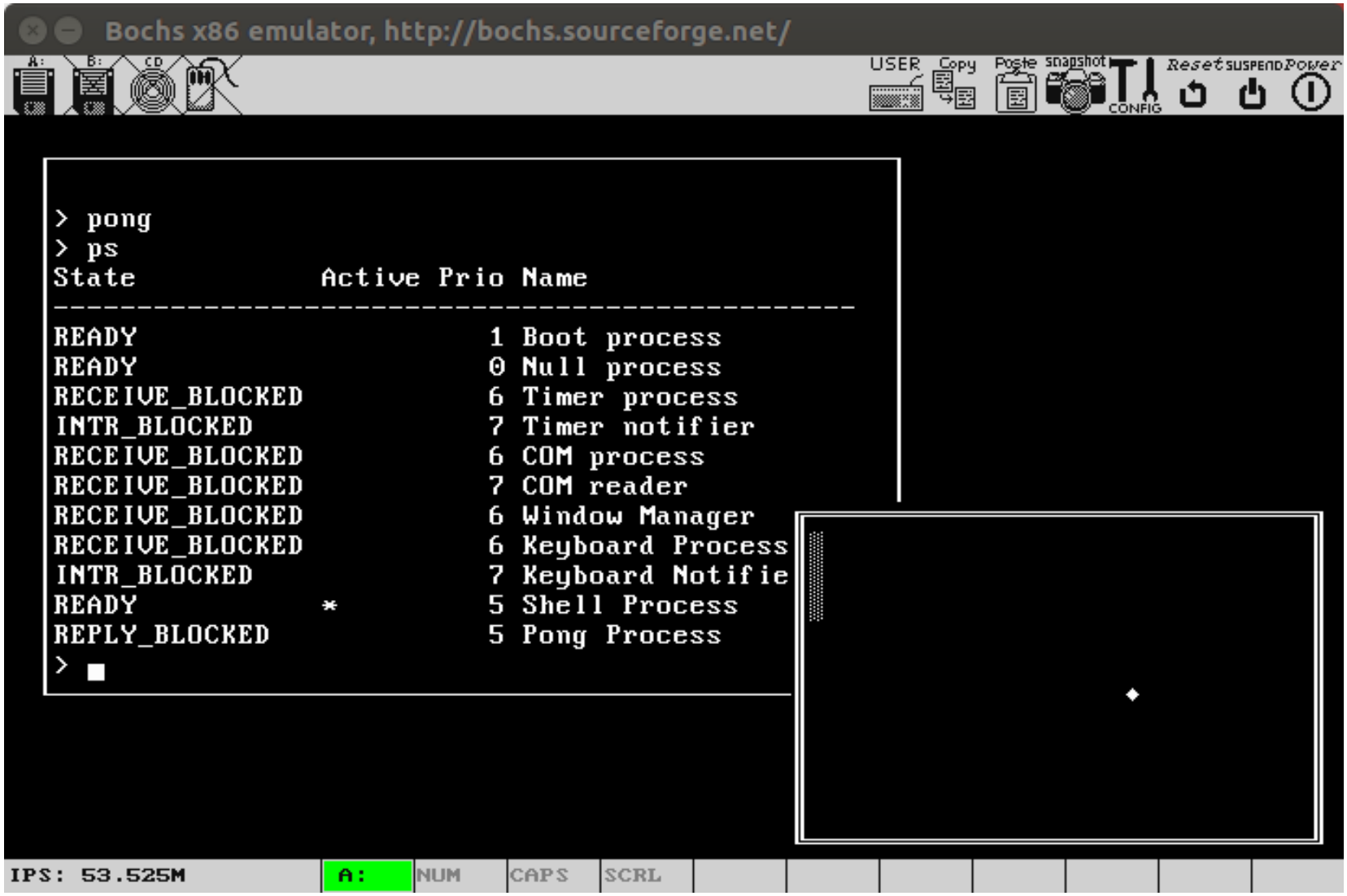
https://danluu.com/malloc-tutorial/

Dynamic Memory Management

- **`void* malloc(size_t size)`**
  Allocates a continuous region of memory of size `size`. The allocated memory is not initialized in any way (e.g., zeroed out). Stops TOS with an assertion if it runs out of available memory. Heap size is 7 MB.
- **`void free(void* ptr)`**
  Frees a previously allocated region of memory. The parameter `ptr` must have been previously returned by `malloc()`. `free()` should be called only once for a given pointer. Memory pointed to by `ptr` should not be accessed anymore after a call to `free()`.

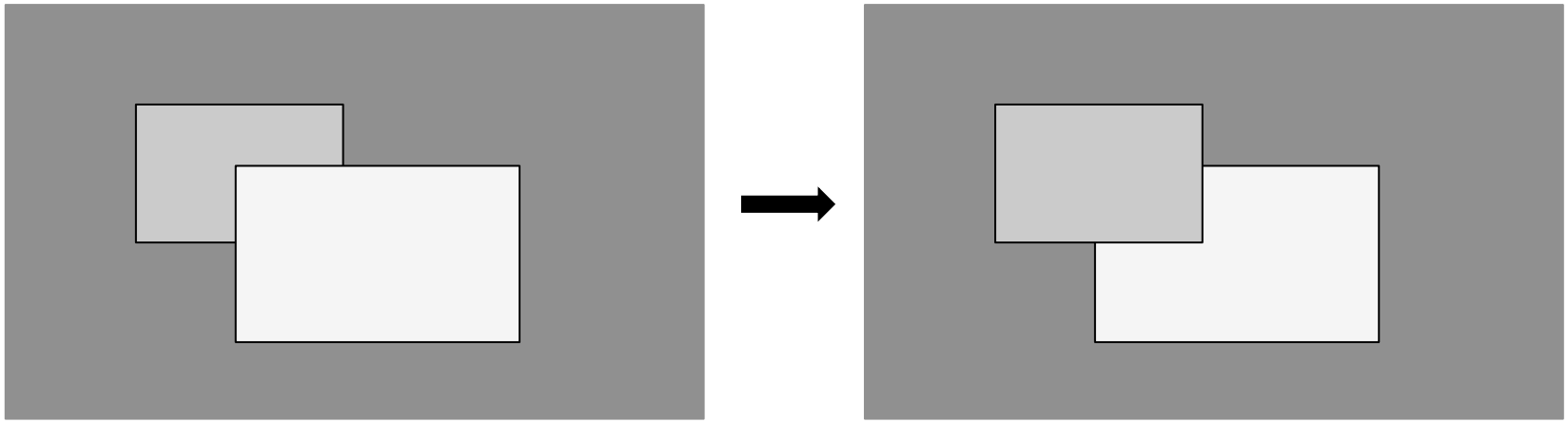
Overlapping Windows

- `wprintf()` has a severe limitation: it can not handle overlapping windows.
- This limits the amount of information visible on the screen.
- New requirements:
  - Handle overlapping windows.
  - Handle shifting of input focus.
  - Handle moving of windows.
- Implementation available in `tos/kernel/wm.c`.



- Window with the current input focus is drawn with double border frame.
- Use <tab> key to shift input focus between windows.
- Use arrow keys to move the current window.

Challenges



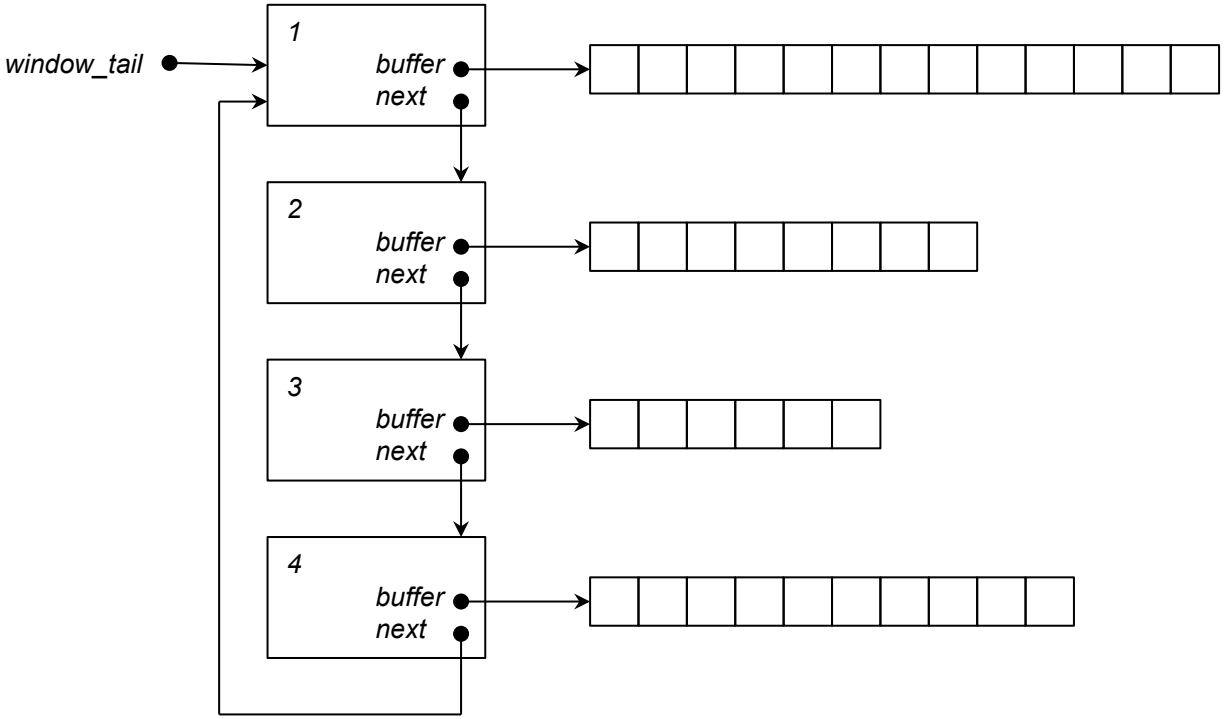
- Only one window has the input focus (i.e., where keystrokes will be sent)
- Windows can be partially moved off the screen.
- Changing Z-order of windows may expose previously hidden parts of a window.
- TOS process can write to a window even if that window is not in the foreground.
- All of this will be implemented in the so-called *Window Manager*.

Window Manager

```
typedef struct __WM {
  int window_id;
  int x, y;
  int width, height;
  int cursor_x, cursor_y;
  char cursor_char;
  char* buffer;
  struct __WM* next;
} WM;
```

- The Window Manager is a TOS process that deals with window output.
- Requests to the WM are sent via TOS' IPC.
- Central data structure `WM`:
  - `window_id`: a unique window ID by which to identify a window.
  - `x`, `y`: top-left corner of the window (relative to the screen)
  - `width`, `height`: width and height of the window.
  - `cursor_x`, `cursor_y`: location of the cursor (relative to the top left corner of the window)
  - `cursor_char`: character to be used for the cursor. If 0, no cursor is drawn.
  - `buffer`: malloc'ed memory of size `width` * `height` that contains the content of the window (only the characters; not the attributes)
  - `next`: pointer to the next window in reverse Z-order.

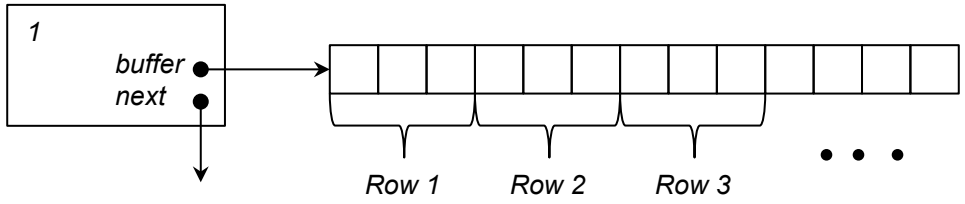
Example



- `window_tail` points to the top-most window. This is the window that will have the input focus and will be drawn with a double border.
- The `next`-chain denotes the Z-order of the windows.
- `buffer`:
  - Content of the window (character only; not attributes)
  - Malloc'ed size is `width * height`.
  - Stored on the heap (not the Video Display Area!)

Scrolling a Window

```
void scroll_wm(WM* window) {
  int to = 0;
  int from = window->width;
  int size = window->width * (window->height - 1);
  for (int i = 0; i < size; i++) {
    window->buffer[to++] = window->buffer[from++];
  }
  for (int i = 0; i < window->width; i++) {
    window->buffer[to++] = 0;
  }
}
```



Redrawing the Screen

```
void redraw_screen() {
  WM* window = window_tail;
  clear_screen_buffer();
  if (window != NULL) {
    do {
      window = window->next;
      draw_window(window);
    } while(window != window_tail);
  }
  copy_screen_buffer();
}

void draw_window(WM* window) {
  BOOL is_top = window == window_tail;
  draw_window_frame(window, is_top);
  draw_window_content(window);
}

void draw_window_content(WM* window) {
  int i = 0;
  for (int y = 0; y < window->height; y++) {
    for (int x = 0; x < window->width; x++) {
      poke_screen_buffer(window->x + x, window->y + y, window->buffer[i++]);
    }
  }
  // …
}

void poke_screen_buffer(int x, int y, char ch) {
  if (x < 0 || y < 0) return;
  if (x >= 80 || y >= 25) return;
  screen_buffer[y * 80 + x] = ch;
}
```

Window Manager

- **`void init_wm()`**
  Initializes the Window Manager. Must be called once from `kernel_main()`.
- **`int wm_create(int x, int y, int width, int height)`**
  Create a new window. `x`, `y` denote the top-left corner of the window. `width` and `height` denote the size of the window. Note: it is not required that the window physically fits the screen. Returns a window ID.
- **`void wm_clear(int window_id)`**
  Clears window and places cursor in the top-left corner of the window.
- **`void wm_set_cursor(int window_id, int x, int y, char cursor_char)`**
  Sets the cursor to a given location within the window. Also allows the cursor character to be changed. If `cursor_char == 0`, no cursor is drawn.
- **`void wm_print(int window_id, const char* fmt, ...)`**
  Prints a formatted string to the window. Correctly handle carriage return (\n) and scrolling.

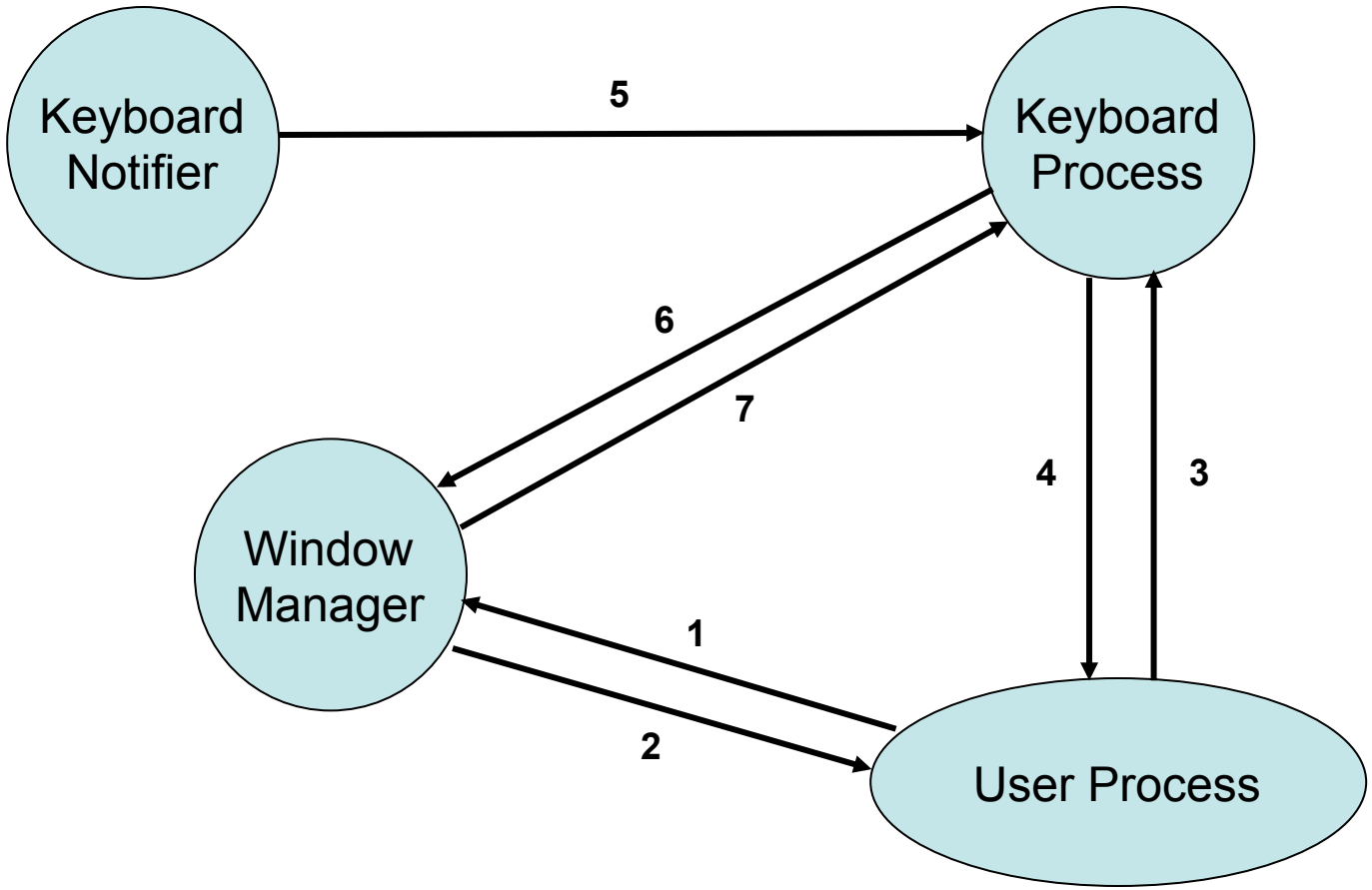
Activating the Keyboard

- In order to use the keyboard, the following steps have to be done:
  - Register the ISR for the keyboard interrupt. This can be accomplished by calling `init_idt_entry (KEYB_IRQ, isr_keyb)` in `init_interrupts()`
  - Make sure that `isr_keyb()` works with your implementation of `wait_for_interrupt()`
  - Call `init_keyb()` from the boot process after calling `init_wm()`.

Keyboard Process

- **`void init_keyb()`**
  Initializes the Keyboard Process. Needs to be called after the initialization of the Window Manager via `init_wm()`.
- **`char keyb_get_keystroke(int window_id, BOOL block)`**
  Will query the Keyboard Process for the next keystroke. `window_id` must be an existing window that was previously created via `wm_create()`. Keystrokes will only be returned when `window_id` has the input focus. If there is no pending keystroke and `block == TRUE`, this function will block the caller until a keystroke is available. When `block == FALSE`, the function will not block when there is no pending keystroke. In that case the function will return 0.

WM/Keyboard Service



WM/Keyboard Process Interactions

1. User process creates a new window via `wm_create()`.
2. Window Manager replies with a unique window ID. User process can use this window ID to print content into the window via `wm_print()`.
3. User process can request keystrokes that are directed to a specific window via `keyb_get_keystroke()`. This request references a window ID.
4. If a keystroke is available for the given window ID, keyboard replies, otherwise user process is kept reply blocked (deferred reply).
5. Keyboard Notifier waits for keyboard interrupts via `wait_for_interrupt()`, processes the keystroke and sends the resulting character to the Keyboard Process via `message()`.
6. If the keystroke is the TAB key, the Keyboard Process will tell the Window Manager to change the input focus next via `wm_change_focus()`. Likewise the arrow keys will be intercepted by the Keyboard Process by calling `wm_move_*()` functions of the Window Manager.
7. Window Manager shifts the focus to the next window and replies with the window ID that has the input focus.

Example

```
void user_process(PROCESS self, PARAM param)
{
   int window_id = wm_create(10, 3, 50, 17);
   wm_print(window_id, ”Hello World!\n");
   while (1) {
     char ch = keyb_get_keystroke(window_id, TRUE);
     wm_print(window_id, "Got key: %c\n", ch);
   }
}
```

- The TOS process above will print out whatever the user types on the keyboard.
- The `TRUE` parameter of `keyb_get_keystroke()` will tell the Keyboard Process to block the caller (deferred reply) in case there is no keystroke.
- Note: keyboard process only returns one character at a time.

TOS Shell

```
void shell_process(PROCESS self, PARAM param)
{
    int window_id = wm_create(10, 3, 50, 17);
    while (1) {
        - read command from keyboard
        - when user hits <enter> execute command
    }
}
```

- The purpose of the TOS shell is to allow a user to type commands.
- The TOS shell is implemented as a TOS process. This process gets created in `start_shell()`.

Assignment “Shell”

- Implement a TOS shell in `~/tos/kernel/shell.c`
- The shell should understand the commands mentioned on iLearn.
- Make sure you initialize the keyboard.
- No test cases are available for this assignment.
- Make sure you cover corner cases (illegal commands, leading and trailing spaces, etc)
- It should be possible to run several shells!

Multiple Process Instances

- It is possible to create the same TOS process multiple times, i.e., call to `create_process()` with the same entry point (function pointer).
- However, special care must be taken in this case since all TOS processes share the same address space.
- A TOS process should NOT have any global variables; only local variables (why?)
- This might require to pass additional parameters to helper functions.
- See `tos/kernel/pong.c` for an example.

```
// Bad
int pong_window_id;

void do_something() {
  wm_clear(pong_window_id);
  //…
}

void init_pong() {
  pong_window_id = wm_create(…);
  do_something();
}
```

```
// Good
void do_something(int window_id) {
  wm_clear(window_id);
  //…
}

void init_pong() {
  int pong_window_id = wm_create(…);
  do_something(pong_window_id);
}
```