

# *Process Management in TOS*

# Objectives

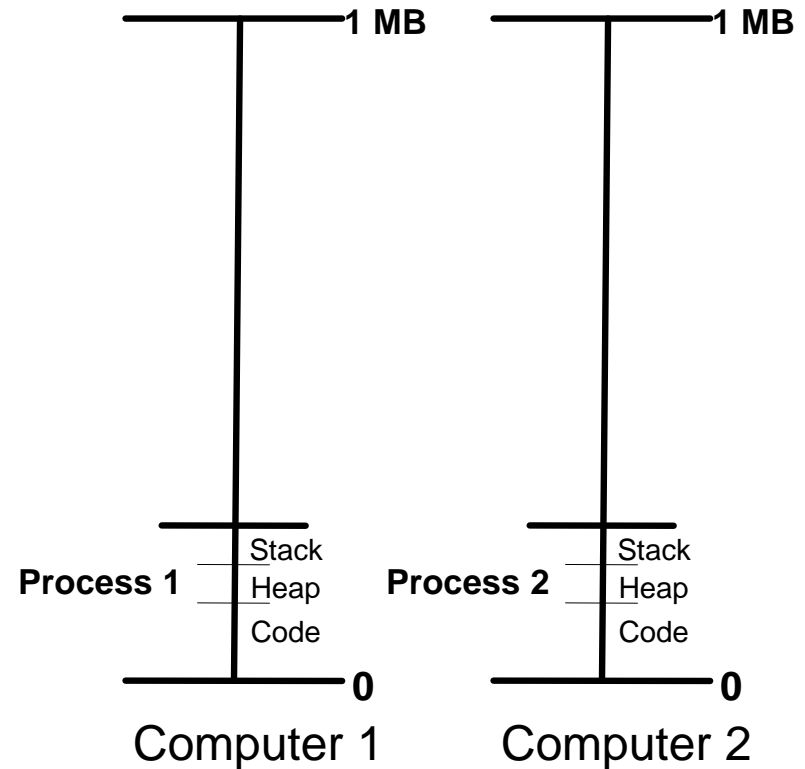
- Introduction to process management of an operating system
- Explain step-by-step how processes are created in TOS

# Introduction to Processes

- What is a process?
  - A process consists of the program code, the data, the heap and the stack
  - A process image is created out of a program with the intention to be executed (e.g., \*.exe files under Windows)
- Single tasking
  - There is only one process. The CPU dedicates complete processing time to one process

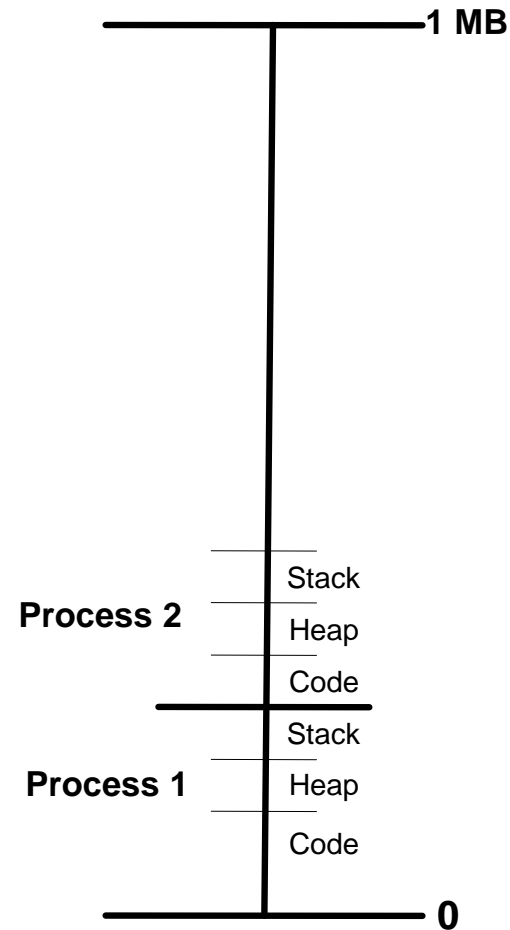
# From Single to Multitasking

- Two identical computers each run a process independently.
- Each computer has its own RAM and CPU, and therefore its own registers EIP, ESP, etc.
- Note that each process has its own code, heap and stack space.
- Idea: move both processes into one computer.
- Problem: then we only have one CPU and one set of registers but two processes!
- Solution: multitasking (time sharing; work a bit on one process and then work a bit on the other process)

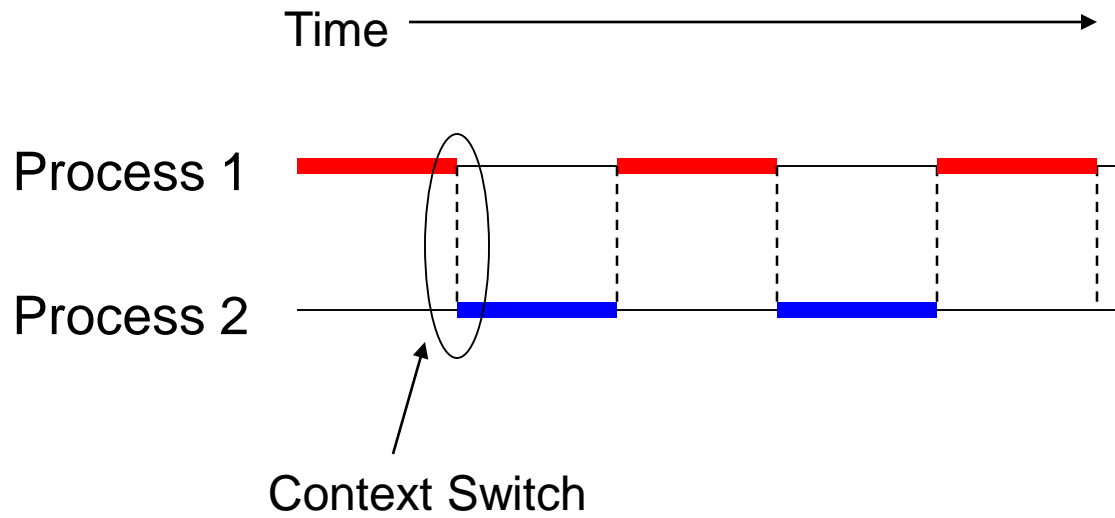


# Multitasking

- Being able to run more than one process “at the same time”
- Several processes time-share one CPU
- Each process has its own program code, memory and stack as shown in the picture



# Sharing the CPU



# Types of Multitasking

- Cooperative multitasking – running process voluntarily relinquishes control
- Preemptive multitasking – running process is suspended involuntary (e.g., because of a hardware interrupt) and control given to another process
- In both cases, “program scheduler” decides next process to run
- Deciding which process to run next is called *Scheduling*

# Process Context

- What is the process context?
  - State of a process consisting of all the x86 registers, heap and stack.
- Where is process context used?
  - Each process needs its own EIP and ESP registers. Since there is only one “copy” of those registers, they must be shared by all processes. EIP and ESP are set for the currently running process. If another process is scheduled to run, the values of EIP and ESP are saved and re-loaded for the next process to run. This is called a *context switch*.

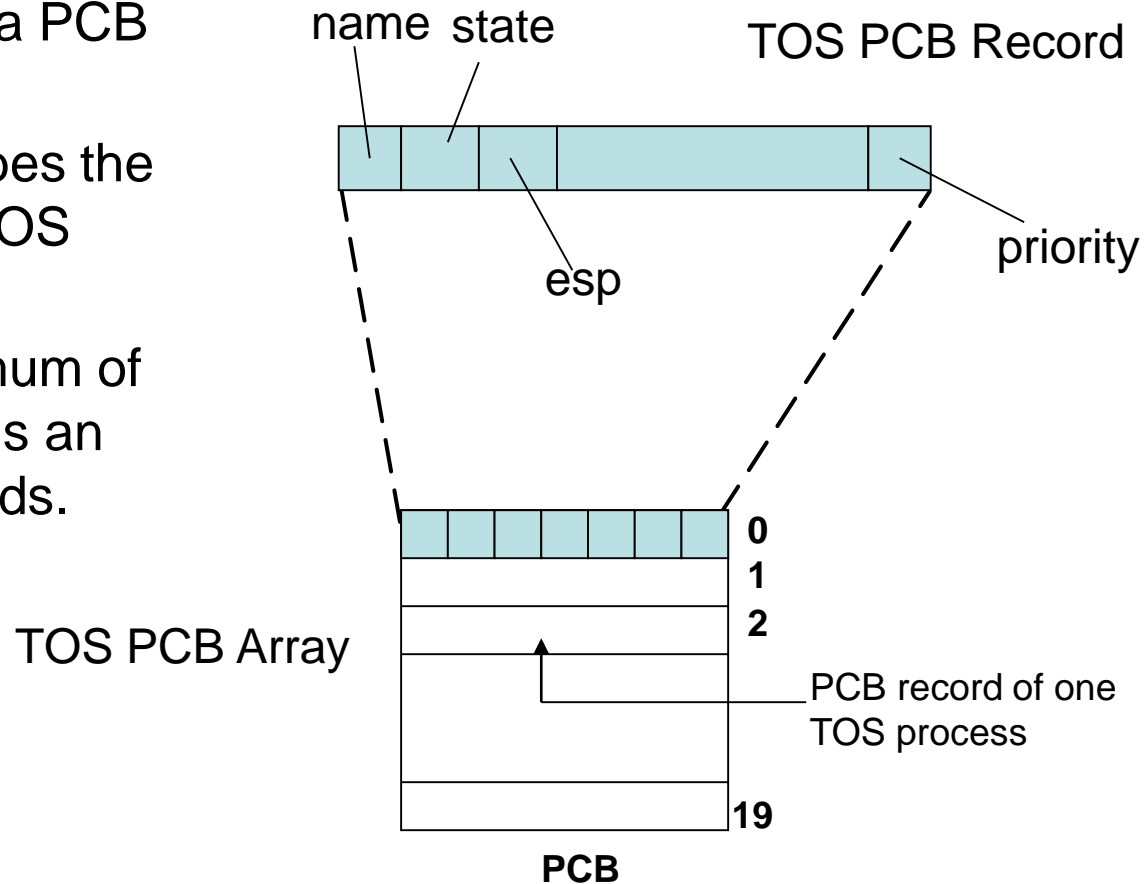


# Introducing TOS Processes

- TOS can have up to 20 processes.
- A process is a C-function (remember pointers to functions?)
- All TOS processes share the global variables, but each process has its own stack
- The context of each process is stored in a PCB (Process Control Block)
- Each process has a priority.
  - Priorities are used to decide which process to run next
  - Priorities must be between 0 (lowest) and 7 (highest)
- All runnable processes are added to a ready queue
- Relevant API:
  - `create_process()`: Creates a new process
  - `add_ready_queue()`: Add a process to the ready queue
  - `remove_ready_queue()`: Remove a process from the ready queue
  - `become_zombie()`: Turn the calling process into a zombie. 😊
  - `dispatcher()`: Select a runnable process
  - `resign()`: Voluntarily relinquish control of the CPU

# Process Control Block (PCB)

- Each TOS process has a PCB record
- One PCB record describes the context of exactly one TOS process
- Since there are a maximum of 20 processes, the PCB is an array with 20 PCB records.



# TOS PCB: struct PCB

- This is the definition of the PCB structure in TOS as defined in the common header `~/tos/include/kernel.h`
- `PROCESS` is a C-pointer to a PCB entry
- Many of the fields will become clear later
- Where are the x86 registers saved? On the stack!

```
typedef struct _PCB {
    unsigned          magic;
    unsigned          used;
    unsigned short    priority;
    unsigned short    state;
    MEM_ADDR          esp;
    PROCESS           param_proc;
    void*            param_data;
    PORT             first_port;
    PROCESS           next_blocked;
    PROCESS           next;
    PROCESS           prev;
    char*            name;
} PCB;

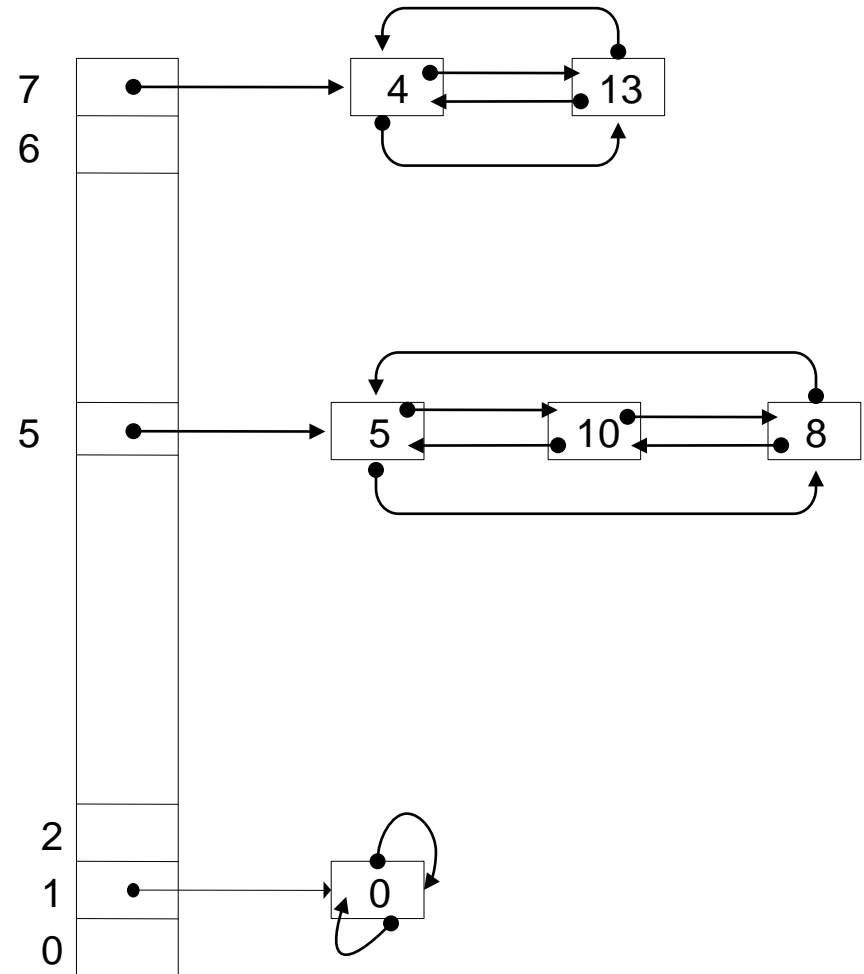
typedef PCB* PROCESS;
```

# TOS Process States

- Each process has a state that gives a high-level indication on its current activity.
- A TOS process can be in exactly one state.
- For now, TOS supports two states (defined in `include/kernel.h`):
  - `STATE_READY`: the process is ready to run and is willing to be scheduled CPU cycles.
  - `STATE_ZOMBIE`: a process has reached the end of its lifespan. It is removed from the ready queue permanently and should no longer be scheduled CPU cycles.
- Note: TOS does not have a `kill_process()` function (the reason for this will become apparent later). Being a zombie is the closest thing for a process to be “dead”.

# TOS Ready Queue

- TOS Ready Queue maintains list of runnable processes
- `ready_queue` is an 8 element-sized array ordered by process priority (0 == lowest; 7 == highest)
- Priority is required to react quickly to interrupts
- Processes with same priority form circular double-linked list within that level
- Ready queue used to select next process to run
- Global variable `active_proc` points to process that the CPU is currently executing
- `PCB.next` and `PCB.prev` are used to implement the double linked list.



`ready_queue`

# Maintaining the Ready Queue

- `void add_ready_queue (PROCESS p)`
  - Changes the state of process `p` to ready (`p->state = STATE_READY`)
  - Process `p` is added to the ready queue.
  - Process `p` is added at the tail of the double linked list for the appropriate priority level.
  - `p->priority` determines to which priority level the process is added in the ready queue
  - automatically maintains the double-linked list.
- `void remove_ready_queue (PROCESS p)`
  - Process `p` is removed from the ready queue.
  - After the removal of the process, the ready queue should be a double-linked list again with process `p` removed.
  - The caller of `remove_ready_queue()` should change the state of process `p` to an appropriate state. Right now, the only state is `STATE_ZOMBIE`.
- `become_zombie()`
  - Turns the caller (`active_proc`) into a “zombie” (`active_proc->state = STATE_ZOMBIE`).
  - Then it should do “nothing”: `while(1);`
  - Will be revisited later once TOS supports context switches.

# CPU Scheduling

- Round Robin Scheduling: Tasks just take turns, rotate through all tasks
- Priority Scheduling: Higher-priority tasks are scheduled before lower-priority tasks (e.g., interactive applications get favored over background computation)
- Real-Time Scheduling: OS makes specific guarantees about when a task will be scheduled

# Scheduling

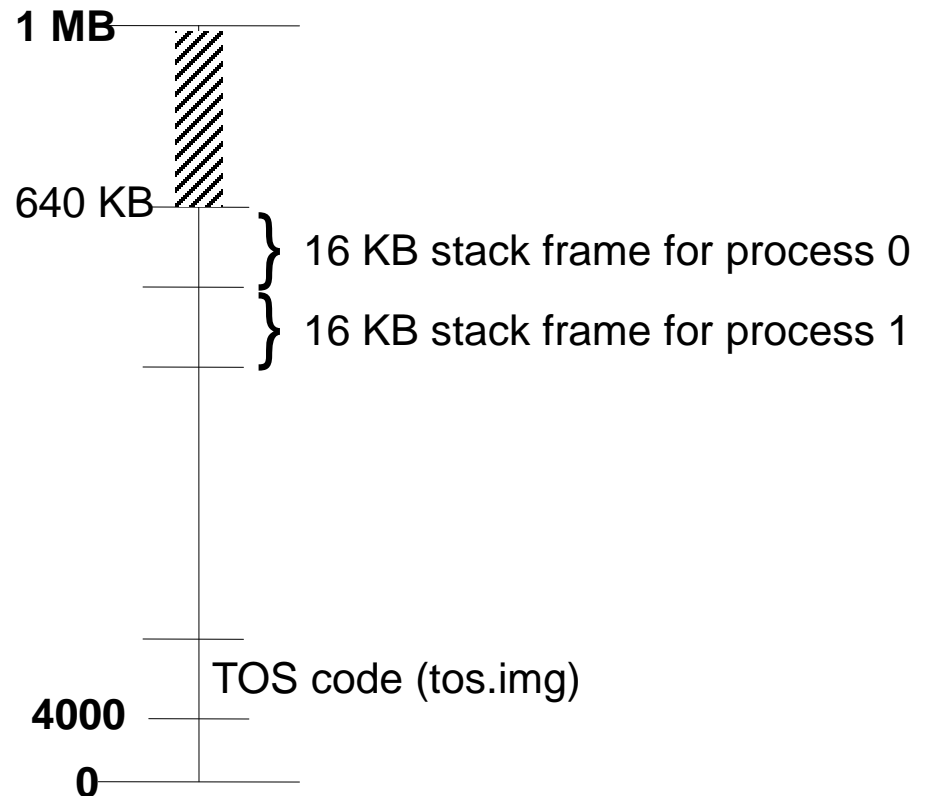
- `PROCESS dispatcher()` returns the next to be executed process.
- Only processes that are on the ready queue are eligible for selection. The assumption is that there is always at least one process on the ready queue.
- Global variable `active_proc` of type `PROCESS` always points to the process (i.e., PCB slot) that currently owns the CPU.
- The next process is selected based on `active_proc`:
  - If there is a process with a higher priority than `active_proc->priority`, then that process will be chosen.
  - Otherwise, `active_proc->next` will be chosen (Round-Robin within the same priority level).



# TOS Process Creation

- TOS is one executable file that gets created by the linker
- The name of this executable is `tos.img`
- During booting, this file gets loaded to address 4000 into RAM.
- A process in TOS is a C-function with the following signature:  

```
void process_a (PROCESS,  
               PARAM) ;
```
- All TOS processes share the same code and heap space but still need different stack space.



# TOS Process Entry Point

- A C-function designates a process in TOS.
- The process needs to be explicitly created (next slide).
- If `void process_a (PROCESS, PARAM)` is a TOS process, then `process_a` defines the entry point of this process, i.e. the new process will start executing from `process_a()`.
- The new process is given the input parameters of type `PROCESS` and `PARAM`. The latter is defined as an `unsigned long` in `~/tos/include/kernel.h`
- The first parameter points to the PCB entry for the newly created process and the second parameter is an application-specific parameter passed from parent to child process.
- Since a TOS process is implemented by a C-function, special care must be taken to ensure that that function is never exited (there is no caller of that function in the traditional sense). By convention, at the end of the function `become_zombie()` should be called.

# Creating a TOS process

- New TOS process can be created via `create_process()`
- This function is located in file `~/tos/kernel/process.c`
- **Signature:**

```
PORT create_process(void (*func) (PROCESS, PARAM),
                    int prio, PARAM param, char* name)
```

***Input:*** `func`: function pointer that defines the entry point of the process to be created.  
`param`: a parameter that the parent process can pass to the child process.  
`prio`: Priority of the process.  $0 \leq \text{prio} \leq 7$   
`name`: Clear text name for the process (e.g. “Boot process”)

***Output:*** For now, this function simply returns a NULL pointer. The meaning of data type `PORT` will be explained later.

# Example: create\_process()

```
void test_process(PROCESS self, PARAM param)
{
    // assert(k_memcmp(self->name,
                       "Test process",
                       k_strlen("Test process")+1)
              == 0);

    // assert(param == 42)
    // ...
    become_zombie();
}

void kernel_main()
{
    // ...
    create_process(test_process, 5, 42,
                  "Test process");
}
```

# create\_process()

What does `create_process (func, prio, param, name)` do?

- Allocates an available PCB entry

- Initializes the elements of this PCB entry

```
PCB.magic      = MAGIC_PCB
PCB.used       = TRUE
PCB.state      = STATE_READY
PCB.priority   = prio
PCB.first_port = NULL
PCB.name       = name
```

- Allocates an available 16KB stack frame for this process

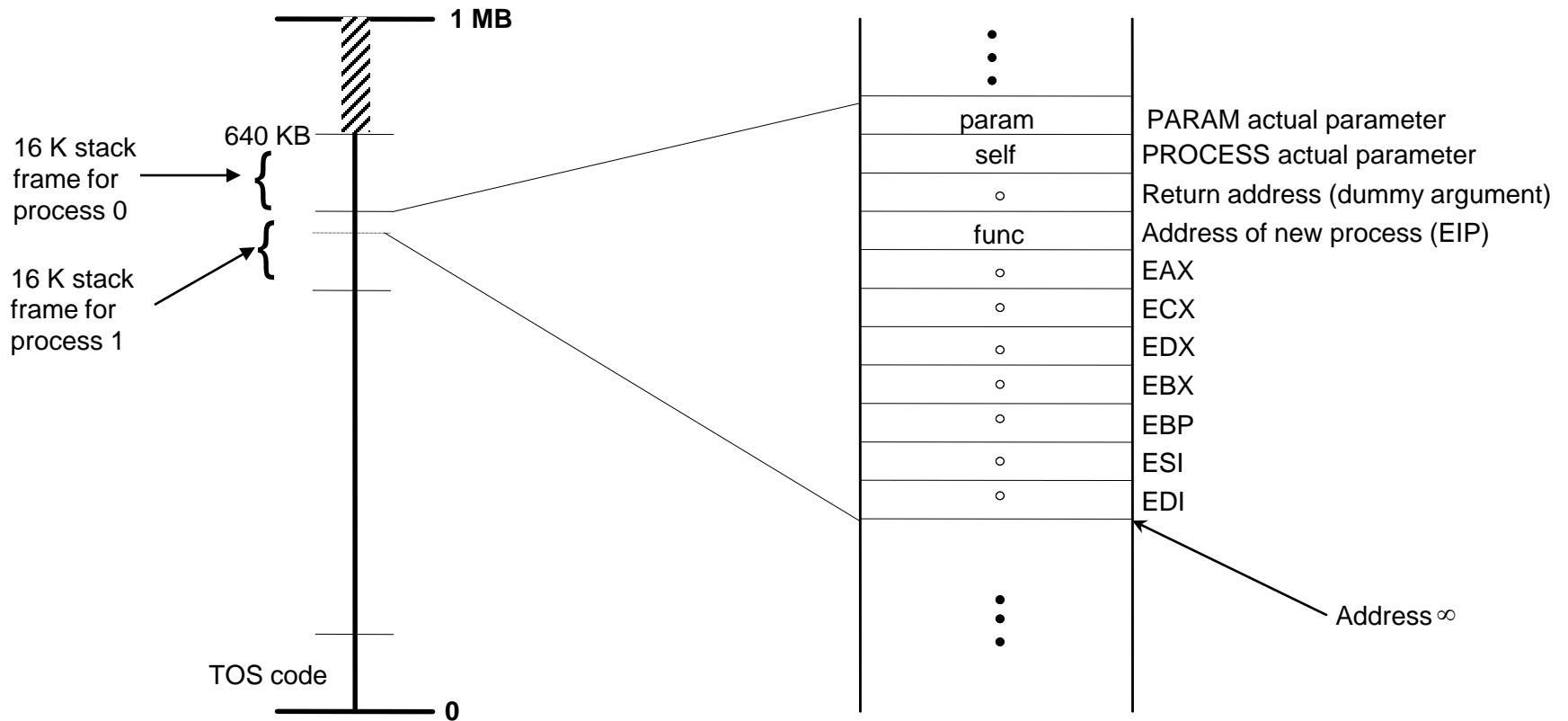
- Initializes an initial stack frame (see next slide)

- Saves the stack pointer to `PCB.esp`

- Adds the new process to the ready queue

- Returns a `NULL` pointer

# Stack of the new process

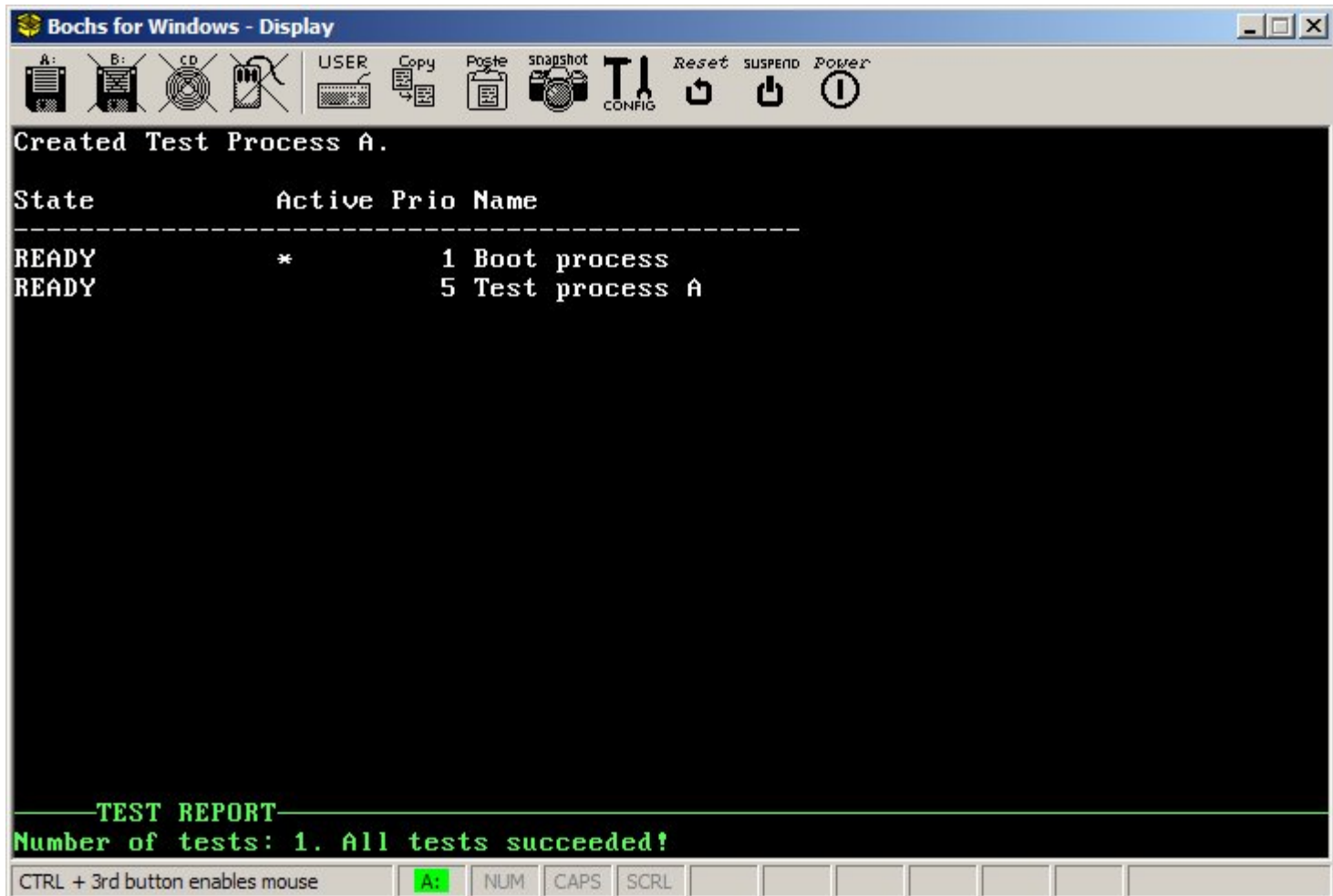


Address  $\infty$  will be saved in PCB.esp

# Printing the PCB

- It is useful to get a list of all processes. Similar to 'ps' command in Unix
- This is done via functions `print_process()` and `print_all_processes()` located in file `~/tos/kernel/process.c`
- TOS functions:
  - `void print_process (WINDOW* wnd, PROCESS proc)`  
print the details of process `proc` to window `wnd`
  - `void print_all_processes (WINDOW* wnd)`  
print the details of all processes currently existing in TOS to window `wnd`
- The following process information should be displayed:
  - Name of the process (`PCB.name`)
  - State of the process (`PCB.state`)
  - Priority of the process (`PCB.priority`)
- The process that is currently active (i.e., the process where `active_proc` is pointing to) should also be marked by a flag
- The following screenshot show how the output could look like. You don't necessarily have to follow the exact same layout. The screenshot is taken from `test_create_process_3`.

# Sample Layout for `print_all_processes()`



The screenshot shows a window titled "Bochs for Windows - Display". The window contains a terminal-like interface with the following text:

```
Created Test Process A.
```

State	Active	Prio	Name
READY	*	1	Boot process
READY		5	Test process A

At the bottom of the window, there is a green text line: `TEST REPORT` and `Number of tests: 1. All tests succeeded!`. The bottom status bar shows keyboard shortcuts: CTRL + 3rd button enables mouse, A:, NUM, CAPS, SCRL.



# Note on Initializing

- There are some global variables in TOS that need to be initialized at startup.
- Those variables are initialized in C-functions called `init_*`():
  - `init_dispatcher()`: initialize the global variables associated with the ready queue
  - `init_process()`: initialize the global variables associated with process creation
- Those init-functions need to be called from `kernel_main()` in order to initialize everything correctly.
- The test functions included in TOS call these functions for you. If you run a test program there is no need to call the init-functions explicitly

# ~/tos/kernel/main.c

```
#include <kernel.h>

void kernel_main()
{
    init_process();
    init_dispatcher();
    init_ipc();
    init_interrupts();
    init_null_process();
    init_timer();
    init_com();
    init_keyb();
    init_shell();
    become_zombie();
}
```

# init\_process()

- When initializing the process sub-system, the first (i.e., current) process becomes the boot process.
- Use PCB[0] for the boot process. Use the following parameters to initialize the PCB entry for the boot process:

```
PCB[0].magic      = MAGIC_PCB
PCB[0].used       = TRUE
PCB[0].state      = STATE_READY
PCB[0].priority   = 1
PCB[0].first_port = NULL
PCB[0].name       = "Boot process"
```

- **Note:** PCB[0].esp does not need to be initialized (why?)

# Assignment 3

- Implement the functions located in `~/tos/kernel/dispatch.c`:  
`add_ready_queue()`, `remove_ready_queue()`,  
`dispatcher()`, `init_dispatcher()`
- Implement the functions located in `~/tos/kernel/process.c`:  
`create_process()`, `print_process()`,  
`print_all_processes()`, `init_process()`
- Implementation for `become_zombie()` is already provided.
- Test cases:
  - `test_create_process_1`
  - `test_create_process_2`
  - `test_create_process_3`
  - `test_dispatcher_1`
  - `test_dispatcher_2`
  - `test_dispatcher_3`
- Hint: no inline assembly necessary for this assignment!



# PacMan (1)

- Remember: The purpose of the PacMan application is a simple game as a showcase for some of the TOS API. You are encouraged to implement it in order to gain a deeper understanding of the TOS API.
- The next stage of PacMan can be implemented when assignment 3 is completed.
- In this next stage, a new TOS process is created for each ghost.



# PacMan (2)

- Here is how to define the ghost process:

```
void ghost_proc(PROCESS self, PARAM param)
{
    create_new_ghost();
}
```

- Note that the signature of `ghost_proc` follows the conventions of a TOS process.
- Now you can create several ghost processes in `init_pacman()` via:

```
int i;
for (i = 0; i < num_ghosts; i++)
    create_process(ghost_proc, 3, 0, "Ghost");
```

- Note: since we have not yet implemented a context switch, creating a ghost will not do anything! Ghost processes will be created, but not yet scheduled. This will only be possible after the next assignment!