

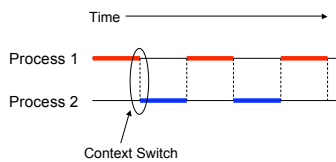
Processes

- An important job of the OS is to provide a service that lets each application think it has its own dedicated computer.
- Each “dedicated computer” has a CPU and some memory.
- For other hardware, OS changes applications’ view to provide more convenient abstractions
 - E.g., hard disk contains fixed-size blocks indexed by number but OS provides a hierarchy of arbitrary-size named files.

Manging the CPU

- Applying OS tasks to the CPU:
- Sharing:
 - Given just 1 CPU, share it among many applications to give the illusion that we have many independent CPUs
- Protection:
 - Don’t let one rogue program “hog” the CPU and exclude others

Sharing the CPU



Context Switching

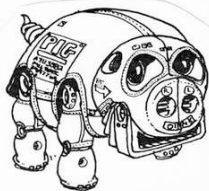
- OS must occasionally switch from one process to another (a *context switch*)
- Since the CPU has just one set of registers, the contents of all registers must be saved and the stored contents of all registers for the new process must be loaded
- The stored contents of all registers are part of the *process context*.

Sharing the CPU

- Cooperative multitasking – running process voluntarily relinquishes control
- Preemptive multitasking – running process is suspended involuntarily (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*

CSC 415

Arno Puder



Process Management in TOS

Introducing TOS Processes

- Unlike a “real” OS, processes in TOS don’t have private memory
 - Hence all global variables are shared
 - Much like kernel threads rather than processes
- The entry point for a process is a C function (remember pointers to functions?)
- The context of each process is stored in a PCB (Process Control Block)

Dynamic Data Structures

- New processes may be created at any time
- When a new process is created, we must allocate a PCB to hold its state
- Ordinarily we would use `malloc()` (like `new` in C++) to allocate memory at runtime
- In OS hacking we don’t have those functions, so we have to simulate dynamic memory management.

Dynamic Data Structures

- Solution: Pick an arbitrary maximum number of items and create a static array
 - We still need a way to tell which items are in use and which are free
- Technique #1: Include a ‘used’ flag in each item. When a new item is needed, look for one where `used==FALSE`
- Technique #2: use a linked list to track which items are available

Part 1: Data Structures

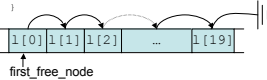
```
typedef struct _Node
{
    int x;
    struct _Node* next_free;
} Node;

Node l[20];
Node* first_free_node;
```

- Compiler allocates array with 20 data items (this is how we get around using malloc())
- I.e., there can be a maximum of 20 data items at any given time
- Each data item contains a pointer to the next available data item

Part 2: Initialization

```
void init ()
{
    int i;
    /* Create single linked list
    of the first 19 elements */
    for (i = 0; i < 19; i++)
        l[i].next_free = &l[i+1];
    /* 0 terminates the list at
    the last element */
    l[19].next_free = (Node *) 0;
    /* Initialize the pointer to the
    first free node */
    first_free_node = &l[0];
}
```



- Before the data structure can be used, it needs to be initialized once
- Function init() creates a single linked list of the first 19 elements of the list.
- Global variable first_free_node is initialized to point to the first array element

Part 3: Allocating a Data Item

```
Node* alloc_data_item ()
{
    Node* tmp;
    tmp = first_free_node;
    first_free_node = tmp->next_free;
    return tmp;
}
```

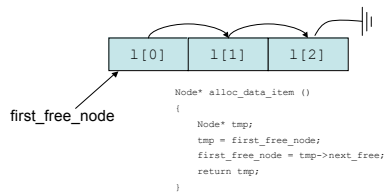
- Function alloc_data_item() returns a free data item from the list of available data items
- The next free data item is determined via global variable first_free_node
- I.e., the next free element is taken from the head of the single linked list
- If no more data items are available, the function returns 0
- The caller is responsible to check that the value returned is not 0

Part 4: Deleting a Data Item

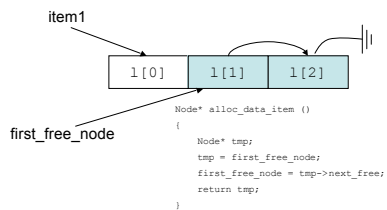
```
void delete_data_item (Node* elem)
{
    elem->next_free = first_free_node;
    first_free_node = elem;
}
```

- A data item is deleted by returning it to the free list
- This is done by adding the data item to be deleted to the beginning of the single linked list

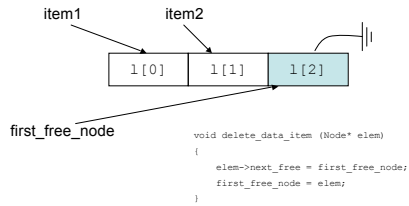
Static array with free list



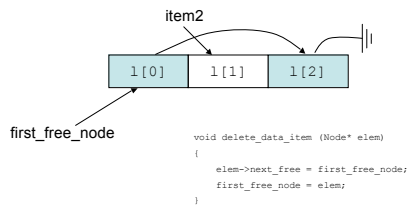
Static array with free list



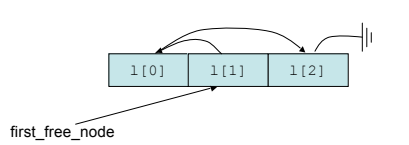
Static array with free list



Static array with free list



Static array with free list



TOS

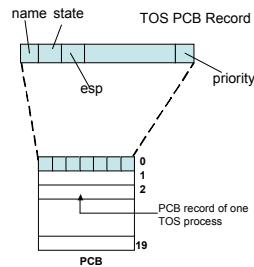
Process State

- Every process has a *state* that indicates what the process is doing.
- For now, all processes will be STATE_READY which means they are ready to execute.
- More states will come later...
- All processes in STATE_READY should be on a list of runnable processes called the *ready queue*

TOS

Process Control Block (PCB)

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



TOS

TOS PCB: struct PCB

- This is the definition of the TOS PCB structure from kernel.h
- PROCESS is a pointer to a PCB entry
- Many of the fields will become clear later

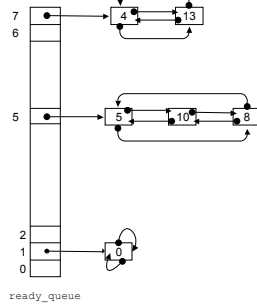
```
typedef struct _PCB {
    unsigned magic;
    unsigned used;
    unsigned short priority;
    unsigned short state;
    MEM_ADDR esp;
    PROCESS param_proc;
    void* param_data;
    PORT first_port;
    PROCESS next_blocked;
    PROCESS next;
    PROCESS prev;
    char* name;
} PCB;

typedef PCB* PROCESS;
```

TOS

TOS Ready Queue

- Ready queue used to select next process to run
- Global variable `active_proc` points to process that the CPU is currently executing
- `ready_queue` is an 8 element-sized array ordered by process priority
- Processes with same priority form circular double-linked list within that level
- `PCB.next` and `PCB.prev` are used to implement the double linked list.



TOS

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 at the tail of the double linked list for the appropriate priority level (determined by `p->priority`).
 - Must maintain 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.

TOS

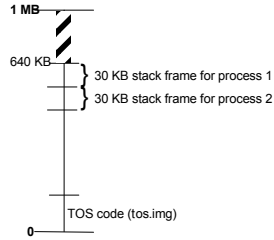
Scheduling

- `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.
- The next process is selected based on `active_proc`:
 - If there is a process with a higher priority than `active_proc`, then that process will be chosen.
 - Otherwise, `active_proc->next` will be chosen (Round-Robin within the same priority level).



TOS Process Creation

- The TOS kernel is one executable file (`tos.img`) created by the linker
- During booting, this file gets loaded to address 0 into RAM.
- All TOS processes share the same code and data space but need different stack space.



TOS Process Entry Point

- A process is created with `create_process()`
- The code that a process will execute is specified by a C function like the following:

```
void process_a(PROCESS proc,
              PARAM param)
```

- The first parameter (`proc`) is the PCB entry for the new process
- The second parameter (`param`) allows the parent process to pass an argument to the new process
 - `PARAM` is defined as `unsigned long` in `kernel.h`



Creating a TOS process

- New TOS process are created via `create_process()`
- **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 (self->name == "Test process")
    assert (param == 42)
}

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

TOS

create_process()

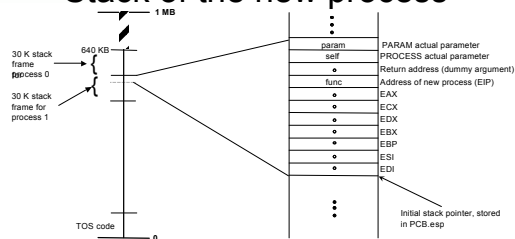
What does create_process (func, prio, param, name) do?

- 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 30KB 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

TOS

Stack of the new process



TOS

Printing the PCB

- It is useful to get a list of all processes, like the Unix command 'ps'
- This is done via functions `print_process()` and `print_all_processes()` located in file `process.c`:
 - `void print_process (PROCESS proc)`
print the details of process `proc`
 - `void print_all_processes ()`
print the details of all processes
- 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 `active_proc` points to) should also be marked

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_processes()`: 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

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 2 Part I

- 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()`
- Test cases:
 - `test_create_process_1`
 - `test_create_process_2`
 - `test_create_process_3`
 - `test_create_process_4`
 - `test_create_process_5`
- Hint: no inline assembly necessary for this assignment!

CSC 415

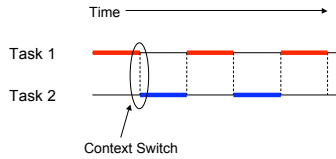
Arno Puder

Context Switch in TOS

Status Quo

- We can create new processes in TOS.
- New processes are added to the ready queue.
- The ready queue contains all runnable processes.
- BUT: so far, none of these new processes ever gets executed.
- What needs to be done: implement a function that switches the context, so that another process gets the chance to run.

Review: Context Switch



Context switching in TOS

- First iteration: cooperative multi-tasking
 - Pre-emptive multi-tasking will come later
 - For now, a process voluntarily gives up the CPU by calling the function `resign()`
- Eventually control is passed back to the original caller because it is assumed that other processes also call `resign()`
- Therefore, from a process' perspective, `resign()` is not doing anything, except causing a delay before `resign()` returns

`resign()` example

- Assumption: there is only one process in the ready queue
- In this example, `resign()` simply does nothing, like a function call that immediately returns.
- `active_proc` is not changed

```
.  
. .  
kprintf ("Location A\n");  
resign();  
kprintf ("Location B\n");  
. .  
.
```

Output

```
Location A  
Location B
```

resign() example

- Assumption: after the call to `create_process()`, there are two processes on the ready queue and `process_a` has a higher priority
- Call to `resign()` does a context switch to `process_a` because it has the higher priority
- `active_proc` changes after `resign`

Output

```
Location A
Location C
```

```
void process_a (PROCESS self, PARAM param)
{
    kprintf ("Location C\n");
    assert (self == active_proc);
    while (1);
}

void kernel_main()
{
    init_process();
    init_dispatcher();
    create_process (process_a, 5, 0,
                  "Process A");
    kprintf ("Location A\n");
    resign();
    kprintf ("Location B\n");
    while (1);
}
```

resign() example

- Assumption: after the call to `create_process()`, there are two processes on the ready queue and `process_a` has a higher priority
- First call to `resign()` switches context to `process_a`
- `process_a` removes itself from the ready queue and then calls `resign()` again. This will do a context switch back to the first process.
- If `remove_ready_queue(self)` were not called, the program would print "Location D" instead of "Location B"

Output

```
Location A
Location C
Location B
```

```
void process_a (PROCESS self, PARAM param)
{
    kprintf ("Location C\n");
    remove_ready_queue (self);
    resign();
    kprintf ("Location D\n");
    while (1);
}

void kernel_main()
{
    init_process();
    init_dispatcher();
    create_process (process_a, 5, 0,
                  "Process A");
    kprintf ("Location A\n");
    resign();
    kprintf ("Location B\n");
    while (1);
}
```

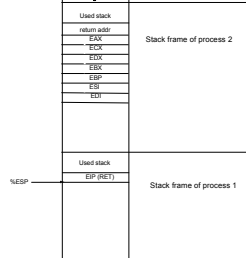
TOS

Understanding resign()

- `resign()` in a nutshell:
 - Save the context of the current process
 - `active_proc = dispatcher();`
 - Restore the context of the new process
 - RET
- Important: `resign()` takes no arguments and has no local variables so nothing is put on the stack by the compiler

Implementing resign()

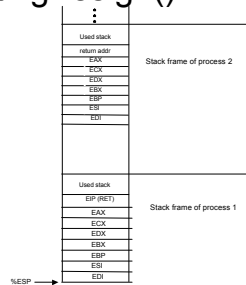
- Process 2 previously called `resign()`
- Process 1 calls `resign()`, the stacks are as shown
- The goal is to "suspend" process 1 within `resign()` and "resume" where process 2 left off in `resign()`
- First step: save the registers for process 1



Implementing resign()

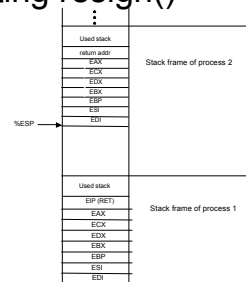
- State of process 1 is saved -- now we actually make the switch:

```
active_proc->esp = %ESP;
active_proc =
dispatcher();
%ESP = active_proc->esp;
```



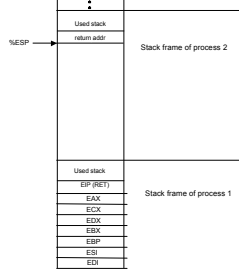
Implementing resign()

- Finally, we restore the state of process 2 by popping the saved register values from the stack
- Note, the registers were stored on the stack when process 2 entered `resign()`



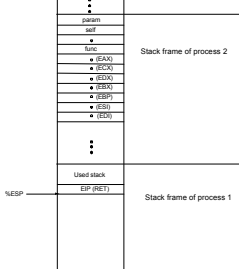
Implementing resign()

- We're done -- when we finish with the `ret` instruction, we jump back to where process 2 called `resign()`

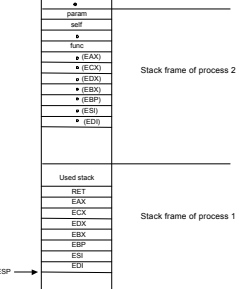


Implementing resign()

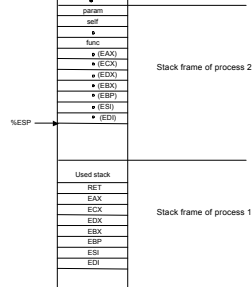
- By creating the initial stack frame carefully in `create_process()`, we ensure that `resign()` can switch to a brand new process as well as one that previously called `resign()`
- Process 1 is active
- Process 2 was created with `create_process()` but has never run.



Implementing resign()

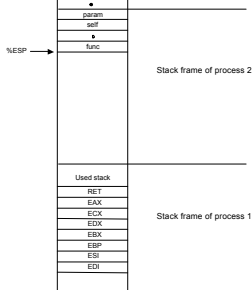


Implementing `resign()`



Implementing `resign()`

- Note the stack frame in process 2 as we return from `resign()` looks just like we are calling `func` for the first time!



Notes on inline assembly

- As explained earlier, `resign()` does amongst others the following:

```
active_proc->esp = %ESP;
active_proc = dispatcher();
%ESP = active_proc->esp;
```

- The first and the third instruction require inline assembly, because the `%ESP` register is accessed.
- There is no C-instruction with which this could be achieved, that is why inline assembly is necessary.



Accessing the Stack Pointer

- This can be accomplished with the following instructions:

```
/* Save the stack pointer to the PCB */  
asm ("movl %esp,%0" : "=r" (active_proc->esp) : );  
/* Select a new process to run */  
active_proc = dispatcher();  
/* Load the stack pointer from the PCB */  
asm ("movl %0,%esp" : "=r" (active_proc->esp));
```

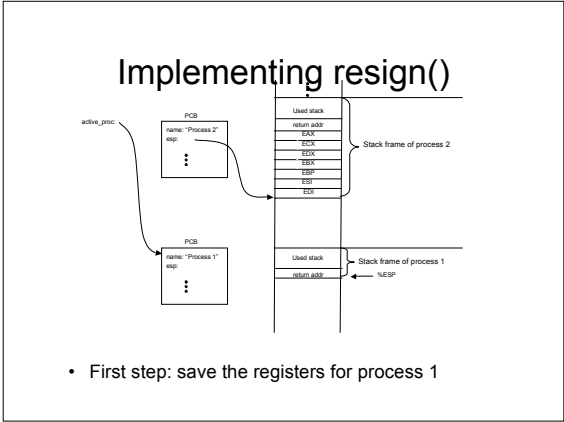
- Notes:
 - The register name %ESP has to be prefixed with another %
 - The specifier "=r" means "an output parameter that should be placed in an x86 register"
 - The specifier "r" means "an input parameter that should be placed in an x86 register"

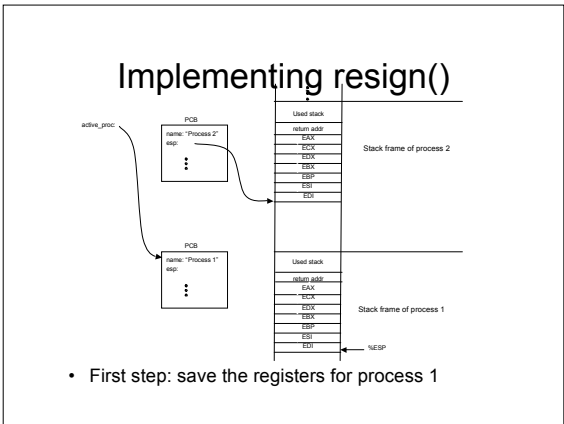
Assembler for Push/Pop

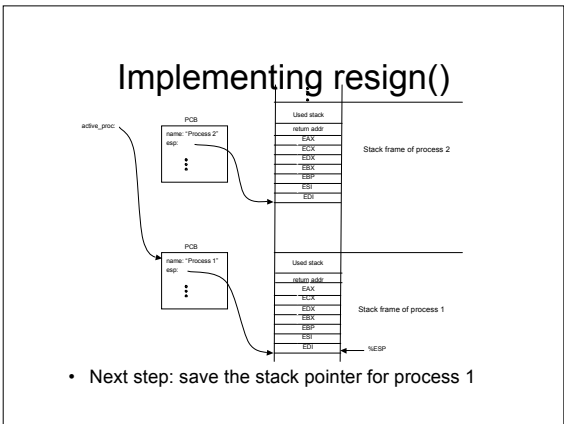
- Register names
 - eax, ebx, ecx, edx, esi, edi, ebp
 - Need a % in the assembler instructions
- Push Function
 - asm ("pushl %eax");
- Pop Function
 - asm ("popl %eax");

Example of resign()

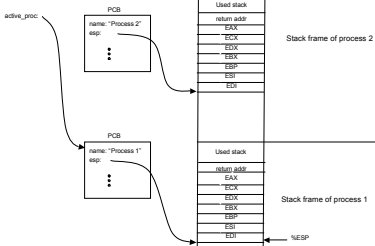
- Process 1 is active, it calls `resign()`
- Process 2 previously called `resign()`, it is ready to run but not currently running.
- Inside `resign()`, assume that `dispatcher()` returns process 2 so we must perform a switch from process 1 to process 2.





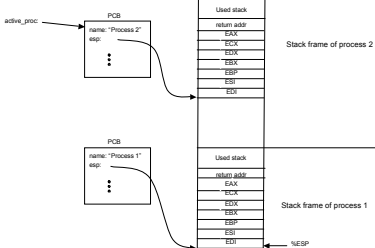


Implementing resign()



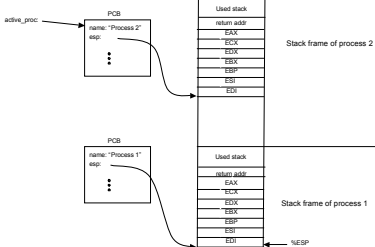
- Next step: choose new process- dispatcher ()

Implementing resign()



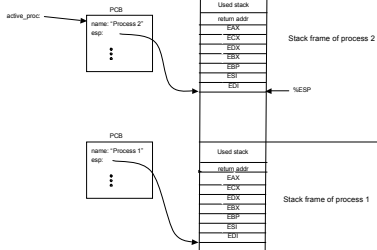
- Next step: choose new process- dispatcher ()

Implementing resign()



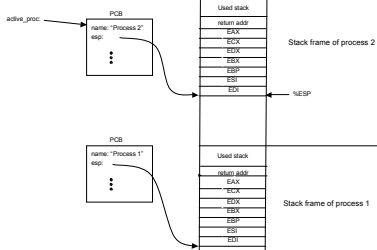
- Next step: restore the stack pointer for process 2

Implementing resign()



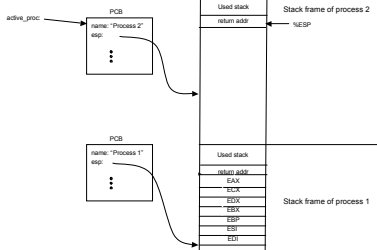
- Next step: restore the stack pointer for process 2

Implementing resign()

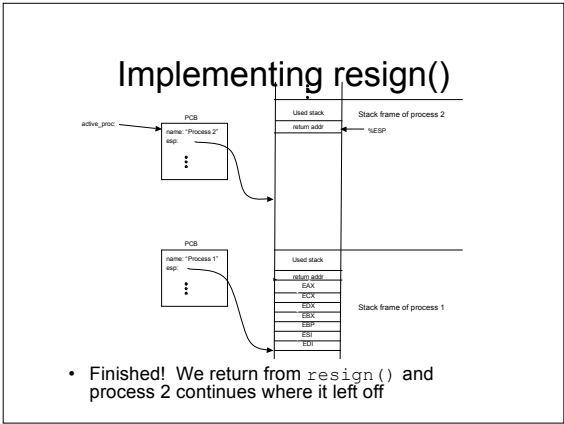


- Next step: restore the registers for process 2

Implementing resign()



- Next step: restore the registers for process 2



Switching to a new process

- Previous example: new process previously called `resign()`
- The context switch is exactly the same if the new process was created with `create_process()` but has never run!

Assignment

Assignment 2 Part II

- Implement `resign()` (in `dispatch.c`)
- Test cases:
 - `test_dispatcher_1`
 - `test_dispatcher_2`
 - `test_dispatcher_3`
 - `test_dispatcher_4`
 - `test_dispatcher_5`
 - `test_dispatcher_6`
 - `test_dispatcher_7`
- Hint: the tests for assignment 2 part II may fail because of errors in assignment 2 part I!

Assignment 2 part II Hints

- This project is relatively straightforward to code, but difficult to debug
- In general, using `assert` is a good thing but here it is dangerous:

```
active_proc = dispatcher();  
assert(active_proc != NULL);
```

- Calling `assert` pushes arguments on the stack but we are trying to manually manage the stack!

Safe assertions in resign

- In this case, we can get work around the problem:

```
void check_active() {  
    assert(active_proc != NULL);  
}
```

```
...  
active_proc = dispatcher();  
check_active();
```

- Inside `resign()`, we call `check_active()` which has no arguments so no stack problems
- This approach is only necessary inside `resign()`
