Context Switch in TOS
Objectives

- Explain non-preemptive scheduling
- Explain step-by-step how a context switch works 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 is missing: running those processes!
• What needs to be done: implement a function that switches the context, so that another process gets the chance to run.
Context switching in TOS

- First step: cooperative multi-tasking
  - Pre-emptive multi-tasking will come later
  - For now, a process voluntarily gives up the CPU by calling the function \texttt{resign()}

- Eventually control is passed back to the original caller because it is assumed that other processes also call \texttt{resign()}

- Therefore, from a process’ perspective, \texttt{resign()} is not doing anything, except causing a delay before \texttt{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.

```c
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`

```
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);
}
```

Output

<table>
<thead>
<tr>
<th>Location A</th>
<th>Location C</th>
</tr>
</thead>
</table>
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 |

```c
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);
}
```
Understanding resign()

• \texttt{resign()} implements a context switch, i.e. it gives another process the chance to run.
• Conceptually, \texttt{resign()} is doing the following:
  – Save the context of the current process pointed to by \texttt{active\_proc}
  – \texttt{active\_proc = dispatcher()}  
  – Restore the context
  – RET

But how does it work exactly?
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

![Stack Diagram]

- Used stack
  - return addr
  - EAX
  - ECX
  - EDX
  - EBX
  - EBP
  - ESI
  - EDI

- Stack frame of process 2
  - Used stack
    - EIP (RET)
  - %ESP

- Stack frame of process 1
Implementing `resign()`

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

```c
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()`
Understanding resign()

• It is especially important to note that the context pushed is not necessarily the same as the context popped
  – recall that active_proc and (hence) %ESP register changed in between push and pop context.
  – then we aren’t looking at the same stack now!
  – but how can we be sure that the ESP register is pointing to some stack?
Understanding resign()

• We made the assumption that wherever `active_proc->esp` points to is where context of the current process is saved

• To satisfy this assumption, we always need to save the context of a process so that it can be popped at some time in the future

• We have already done this!
  – for a new process we setup the stack (see `create_process()`)
  – for process calling `resign()` we setup the stack (identical to the way we did it for `create_process()`) before call to `dispatch()`
  – now you should be able to connect the dots
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.

<table>
<thead>
<tr>
<th>param</th>
<th>Stack frame of process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td></td>
</tr>
<tr>
<td>func</td>
<td></td>
</tr>
<tr>
<td>(EAX)</td>
<td></td>
</tr>
<tr>
<td>(ECX)</td>
<td></td>
</tr>
<tr>
<td>(EDX)</td>
<td></td>
</tr>
<tr>
<td>(EBX)</td>
<td></td>
</tr>
<tr>
<td>(EBP)</td>
<td></td>
</tr>
<tr>
<td>(ESI)</td>
<td></td>
</tr>
<tr>
<td>(EDI)</td>
<td></td>
</tr>
<tr>
<td>%ESP</td>
<td>Stack frame of process 1</td>
</tr>
<tr>
<td>EIP (RET)</td>
<td></td>
</tr>
</tbody>
</table>
Understanding resign()

- And don’t forget – because the context popped was different than the context pushed in the beginning of `resign()`, the return address also is different
- So `resign()` pushed one return address and popped another return address by clever ESP register manipulation
- What does this mean? `resign()` returns to some other address, not to the caller process
- tada! we have a context switch!
Notes on inline assembly

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

```c
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:

```c
/* 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”
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.
Example of resign()

- First step: save the registers for process 1
Example of resign()

- First step: save the registers for process 1
Example of resign()

- Next step: save the stack pointer for process 1
Example of resign()

- Next step: choose new process - dispatcher()
Example of resign()

- Next step: choose new process- `dispatcher()`
Example of resign()

- Next step: restore the stack pointer for process 2
Example of resign()

- Next step: restore the stack pointer for process 2
Example of resign()

• Next step: restore the registers for process 2
Example of resign()

- Next step: restore the registers for process 2
Example of resign()

- Finished! We return from `resign()` and process 2 continues where it left off
Context Switch

- Context switch is implemented by one function:
  
  ```c
  void resign()
  ```

- This function is located in the file `~/tos/kernel/dispatch.c`
Assignment 4

• **Implement** `resign()` *(in dispatch.c)*

• **Test cases:**
  - `test_resign_1`
  - `test_resign_2`
  - `test_resign_3`
  - `test_resign_4`
  - `test_resign_5`
  - `test_resign_6`

• **Hint:** the tests for assignment 4 may fail because of errors in assignment 3!
Assignment 4 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:

```c
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:

```c
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()`
Inline Assembly

• For simple self-contained instructions:
  
  \[
  \text{asm("pushl \%eax");}
  \]

• But sometimes we need to refer to a C expression inside the inline assembly:
  
  \[
  \text{asm("movl \%esp, active_proc->esp");}
  \]

• Things get really messy here, just cut-and-paste from the next slide!
Inline Assembly

- The middle steps of `resign()`:
  
  /* 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 `%`
Revisiting `become_zombie()`

- The current `become_zombie()` implementation is as follows:
  ```c
  void become_zombie()
  {
      active_proc->state = STATE_ZOMBIE;
      while (1);
  }
  ```

- The endless loop is just needlessly burning CPU cycles. With `resign()` this can done more efficiently:
  ```c
  void become_zombie()
  {
      active_proc->state = STATE_ZOMBIE;
      remove_ready_queue(active_proc);
      resign();
      // Never reached
      while (1);
  }
  ```
PacMan (1)

- Earlier you were told to create several ghost processes in `init_pacman()` via:
  ```c
  int i;
  for (i = 0; i < num_ghosts; i++)
    create_process(ghost_proc, 3, 0, "Ghost");
  ```
- It was said although you create several ghost processes, you will not see them yet, because they will not yet get scheduled.
- After the for-loop, add a call to `resign()` as the next experiment.
- Because the ghost process has a higher priority than the boot process, you should see one ghost.
- Note: you will only see one ghost, even though you might have created several ghost processes (why?)
PacMan (2)

- The reason you will see only one ghost is because TOS only supports cooperative multitasking at this point.
- In order to see the other ghosts, each ghost needs to voluntarily relinquish control of the CPU by making a call to `resign()`.
- Earlier you were told to implement a function called `create_new_ghost()` according to the following pseudo code:

  ```
  void create_new_ghost()
  {
    GHOST ghost;
    init_ghost(&ghost);
    while (1) {
      remove ghost at old position (using remove_cursor())
      compute new position of ghost
      show ghost at new position (using show_cursor())
      do a delay
      resign()
    }
  }
  ```

- Add a call to `resign()` in that function as indicated above. Now you should see several ghosts!