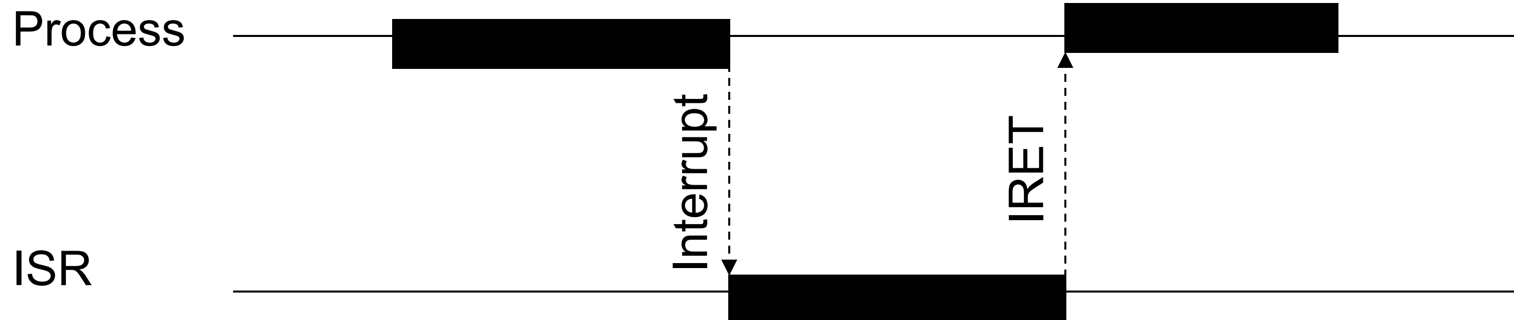
Preemptive Multitasking

Objectives

- Making TOS preemptive
- Avoiding race conditions

Status Quo

- TOS is non-preemptive. i.e., a process has to relinquish control of the CPU voluntarily via `resign()`
- The implication is that if a process never calls `resign()`, no other process will get a chance to run (even if they are of higher priority)
- An ISR is <u>not</u> a process, the ISR runs in the context of a process
- An ISR is only an asynchronous procedure call where a bit of code can be executed in response to an interrupt



Implementing Preemption

- Idea: write an ISR for the timer interrupt. Inside the ISR we call `dispatcher()` to schedule some other process to run
- Idea sounds simple, but requires some deep thinking
- What we want to happen:
  - Process 1 is running
  - Timer interrupt calls the appropriate ISR
  - Call to `dispatcher()` inside ISR schedules another process
  - ISR exits to process 2
  - Process 2 continues running
- What does this really mean? A context switch happens in side the ISR!
- So: the ISR is doing something similar to `resign()`

Preemptive Multitasking

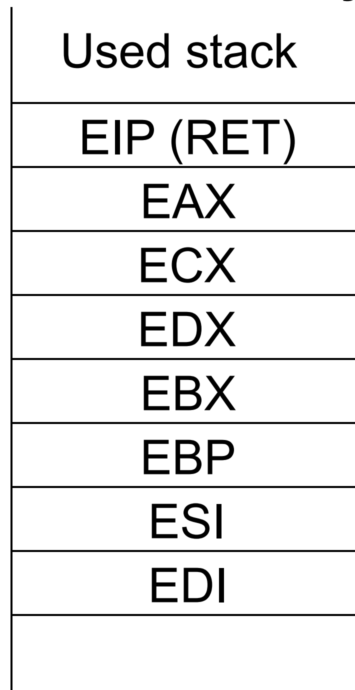
- What to do next: support for preemptive multitasking
- Even though a process is not executing `resign()`, other processes get a chance to run
- Every process should get a "time-slice". When that slice is used up, another process gets a chance to run
- Since the computer is very fast and the time slice is typically short, it gives the illusion of several processes seemingly running concurrently

```
void proc_1 (PROCESS self, PARAM p)
{
  MEM_ADDR screen_offset = 0xB8000;
  while (42)
    poke_b(screen_offset,
           peek_b(screen_offset)+1);
}
```

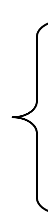
```
void proc_2 (PROCESS self, PARAM p)
{
  MEM_ADDR screen_offset = 0xB8002;
  while (42)
    poke_b(screen_offset,
           peek_b(screen_offset)+1);
}
```

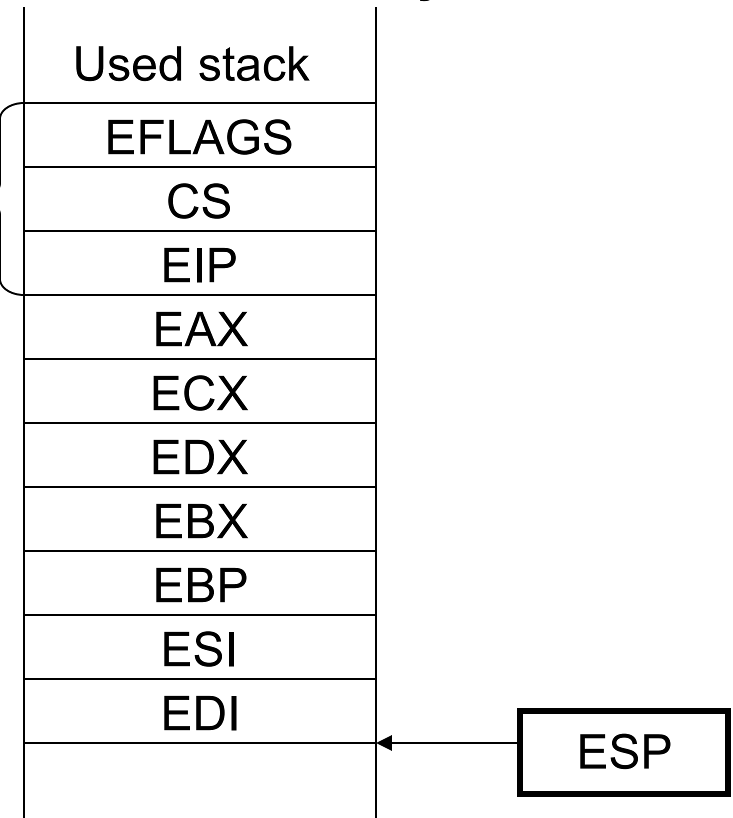
Details of Preemption

Context as saved by `resign()`



Context as saved by `ISR`





Problem: `resign()` and ISR save contexts differently!

Interrupt Preemption

- Problem: `resign()` and `create_process()` build up a different stack frame than an ISR
  - `resign()` and `create_process()` save a 32 bit return address on the stack (intra-segment return address)
  - ISR saves EFLAGS, CS and the return address on the stack (inter-segment return address)
- We can not change the way the x86 handles interrupts (i.e. that an interrupt results in an inter-segment subroutine call)
- Only possible solution: change `resign()` and `create_process()` so that their stack frames are identical to that of an ISR!

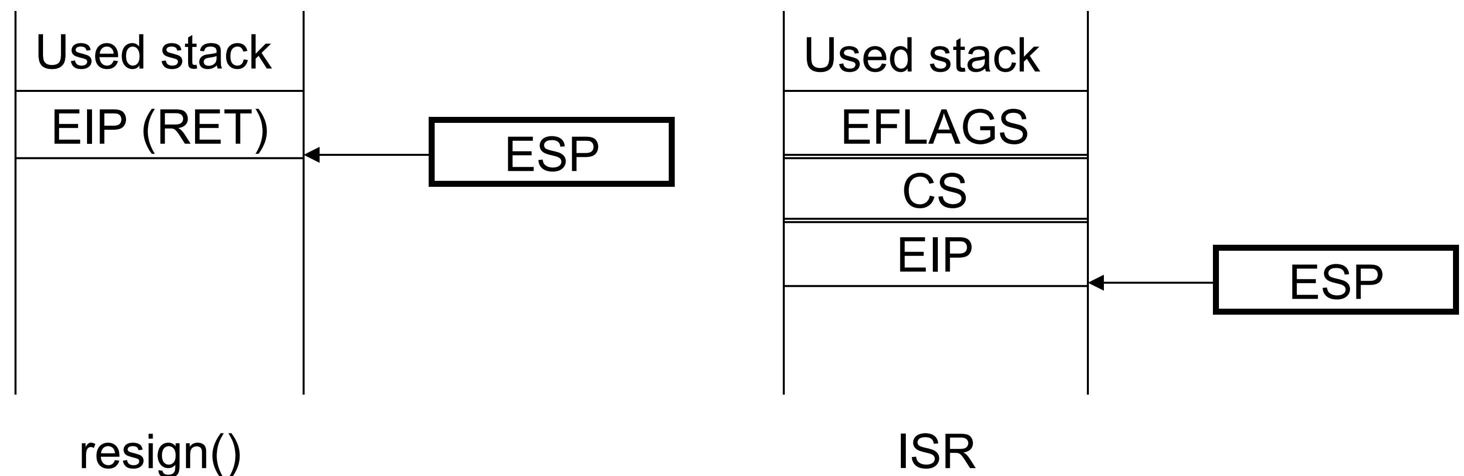
Revisiting create_process()

- Changing the implementation of create_process() is easy
- When create_process() builds up the stack frame for the new process, simply include EFLAGS and CS at the right locations
- For EFLAGS, write the value 512 (0x200). The one bit equal to 1 is IF == 1 (enabled interrupts)
- For CS, write the value 8 (this is the code segment for TOS). Remember to write this value as a long!
- Here is what the stack frame should look like:

param
self
0
512
8
ptr_to_new_proc
0 (EAX)
0 (ECX)
0 (EDX)
0 (EBX)
0 (EBP)
0 (ESI)
0 (EDI)

Revisiting resign() (1)

- Making sure that `resign()` builds up the same stack frame is a bit more complicated.
- Here is the situation right after we have entered `resign()` and the ISR (before pushing the context)



- How can we make the stack frame of `resign()` look like the one of the ISR? Through some assembly magic (next slide)

Revisiting resign() (2)

- Building up the correct stack frame in `resign()` can be achieved with the following assembly instructions (right at the beginning of resign):

```
PUSHFL                          ; Push EFLAGS
CLI                             ; Disable Interrupts
POP      %EAX                   ; EAX == EFLAGS
XCHG     (%ESP), %EAX           ; Swap return address with EFLAGS
                                ; EAX now contains the return
                                        ; address
PUSH     %CS                    ; Push long return address
PUSH     %EAX
```

- Notes:
  - The code above overwrites the original content of `%EAX`. This is OK
  - `XCHG (%ESP), %EAX` swaps the content of `EAX` and the top of the stack
  - After executing the above code, the stack frame looks exactly as if an interrupt had occurred.

Revisiting resign() (3)

- Another thing that needs to be changed is the way we exit from `resign()`
- Recall that the C-compiler emits a RET instruction to exit a function
- This corresponds to an intra-segment jump
- But since we modify the stack to look like that of an ISR, we need to exit `resign()` by inter-segment jump
- This can easily be accomplished by using the assembly instruction IRET

Doing a Context Switch in an ISR

- Once `resign()` and `create_process()` have been changed as described, doing a context switch inside of an ISR is simple:

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

- This is exactly what happens inside of `resign()`!
- So: an ISR behaves similar to `resign()`, except it is triggered by an interrupt, and not by a voluntary call to `resign()`
- That is the difference between preemption and non-preemption

Atomicity

- Are we done yet in order to implement preemptive multitasking?
- NO!
- Problem: several processes use API such as `add_ready_queue()` “concurrently”. There might be race conditions when two processes call this API concurrently.
- Race condition: because of concurrency, the execution of two processes may not always yield the correct result (“race” between two processes). See example on next slide.
- Code that does not exhibit a race condition is said to be reentrant.
- This is similar to the “Too much milk” scenario discussed in an earlier class!

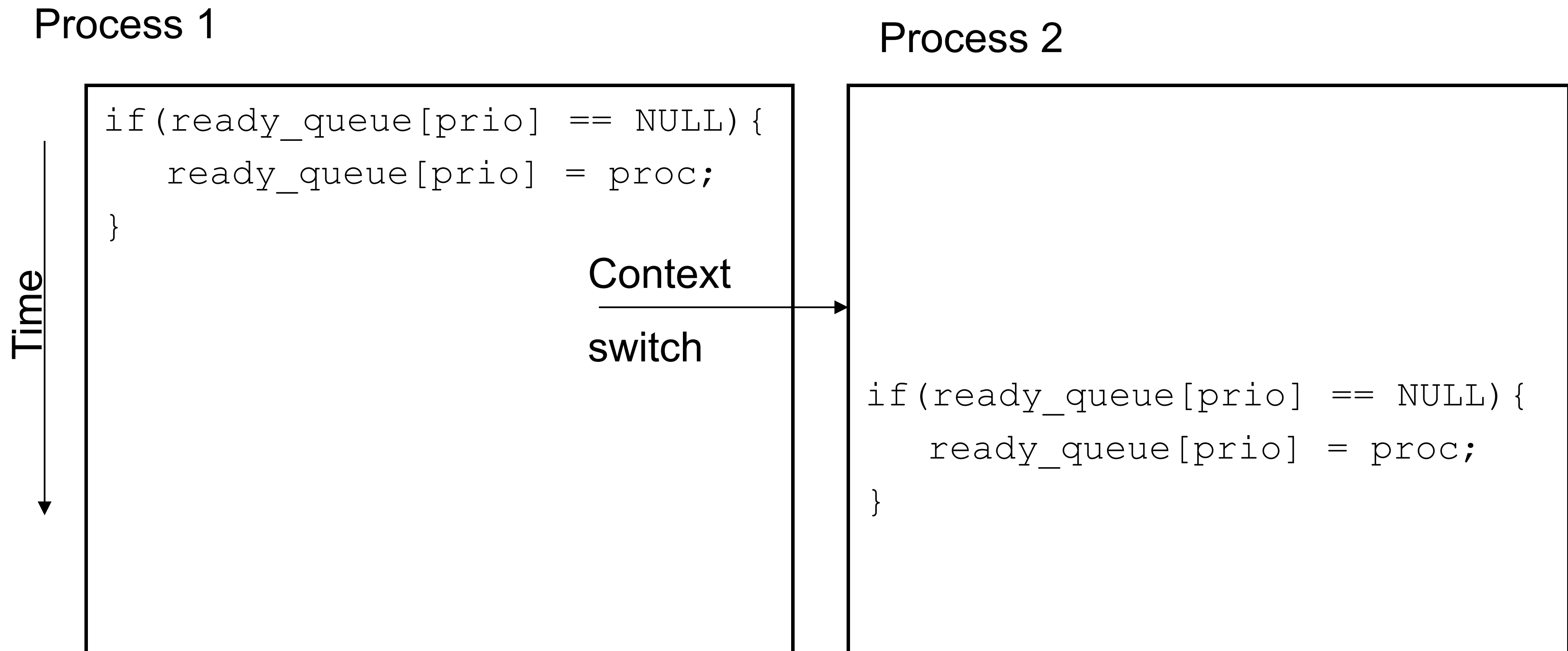
Race condition (1)

- The implementation of `add_ready_queue()` contains the following code:

```
PCB* ready_queue [MAX_READY_QUEUES];
void add_ready_queue (PROCESS proc)
{
    //……
    if (ready_queue[prio] == NULL) {
        //There are no other processes with
        //this priority
        ready_queue[prio] = proc;
    }
    //……
}
```

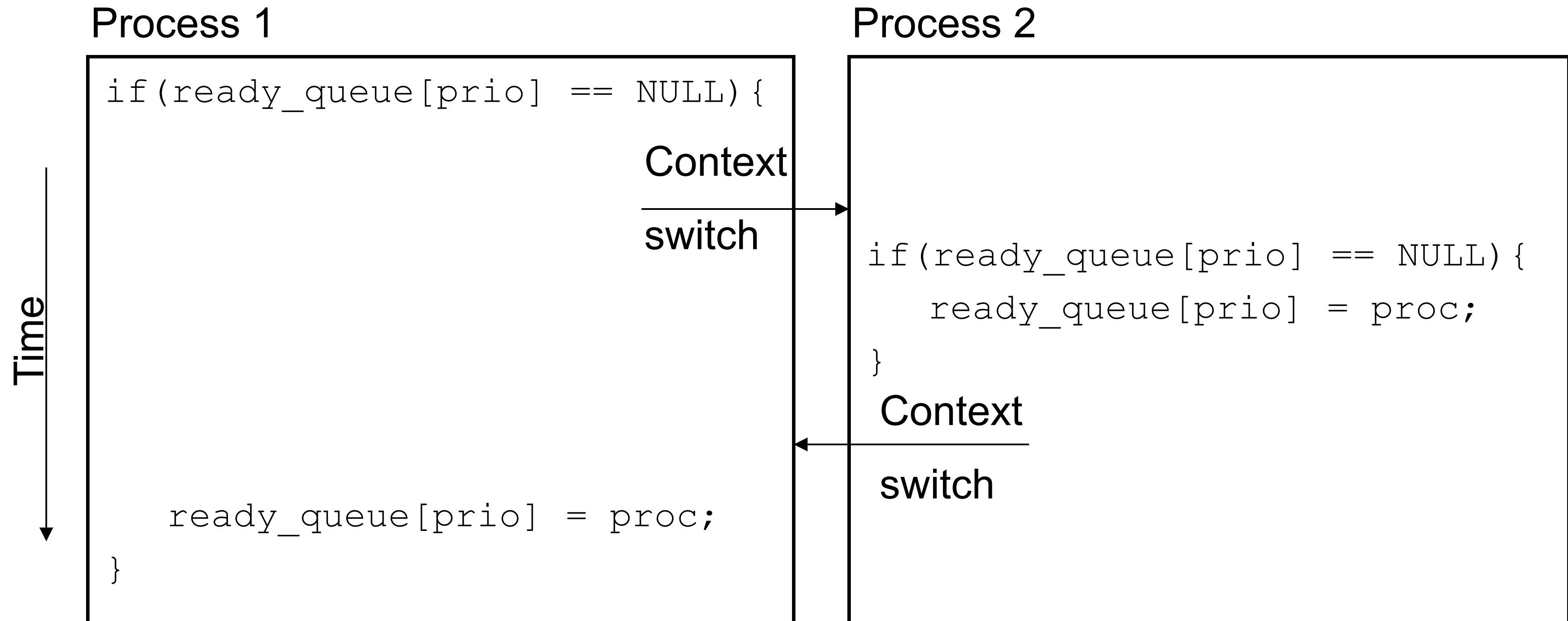
- Remember that because of preemption, a context switch can happen between any two (machine) instructions.
- In the following two slides, process 1 and process 2 both call `add_ready_queue()` at the same time. The slides show the interleaving of instructions.

Race conditions (2)



- Process 1 first executes the if-statement and then process 2
- No race condition

Race conditions (3)



- Process 1 first executes the if-statement, but before it executes the assignment, a context switch happens and process 2 starts to run
- Because the assignment didn't happen yet, process 2 will also enter the if-statement
- After the second context switch, process 1 executes the assignment, overwriting the assignment of process 2: Race Condition!

Reentrant code

- How can we make our code reentrant, i.e. avoid race conditions
- We have to make sure that no context switch happens while in functions that are not reentrant.
- This can be achieved by disabling interrupts while in these functions:

```
void add_ready_queue (PROCESS proc)
{
        volatile int saved_if;
        DISABLE_INTR (saved_if);
        //…
        ENABLE_INTR (saved_if);
}
```

- `DISABLE_INTR()` and `ENABLE_INTR()` are crude versions of *acquire lock* and *release lock* in the "too much milk example". We avoid race conditions simply by turning off interrupts and thereby making sure no context switch can happen inside the timer ISR.
- Remember that `ENABLE_INTR()` needs to be called whenever you exit a function. E.g., if you exit a function via `return`:

```
if (…) {
        // …
        ENABLE_INTR (saved_if);
        return;
}
```

- Instrument all functions that may have race conditions: `remove_ready_queue`, `create_process()`, `output_string()`, `send()`, and many more!

Disabling/Enabling Interrupts

- `DISABLE_INTR()` and `ENABLE_INTR()` are macros defined in `~/tos/include/kernel.h`
- Both these marcos require a parameter of type `volatile int`:

```
volatile int saved_if;
DISABLE_INTR (saved_if);
//…
ENABLE_INTR (saved_if);
```

- `DISABLE_INTR()` saves the current value of the IF bit in `saved_if`, and then executes `CLI`
- `ENABLE_INTR()` restores the IF bit to what was saved in `saved_if`. This guarantees that interrupts will only turned on, if there were turned on before calling `DISABLE_INTR()`
- This is important for nested function calls. Before exiting, the nested function should restore interrupts to either enabled or disabled depending on whether interrupts were enabled or disabled when the nested function was called.

Interrupts and old test cases

- After implementing support for interrupts, some of the old test cases will not work anymore (e.g., `test_dispatcher_*()` and `test_ipc_*()`)
- Why?
  - Those older test cases do not call `init_interrupts()`
  - `create_process()` now pokes the value of 512 for EFLAGS. Reminder: 512 == IF bit is true (Interrupts enabled)
  - The moment a context switch happens, the last instruction in `resign()` is now `IRET`.
  - Because `IRET` pops off EFLAGS *and* `create_process()` poked 512 *and* `init_interrupts()` was not called, disaster strikes.
  - This means: `IRET` implicitly turns on interrupts, but interrupts were not initialized.

Solution

- Make use of global variable `interrupts_initialized` (defined in `~/tos/kernel/intr.c`)
- In `init_interrupts()` set `interrupts_initialized` to true
- In `create_process()`:
  - If `interrupts_initialized` == true: poke 512 for EFLAGS
  - If `interrupts_initialized` == false: poke 0 for EFLAGS
- Poking 0 guarantees that interrupts will remain turned off during a context switch.
- This guarantees that interrupts will remain disabled for older test cases that do not call `init_interrupts()`.

Assignment 7

- Implement the function located in `~/tos/kernel/intr.c`: `isr_timer()`
- Modify the function located in `~/tos/kernel/intr.c`: `init_interrupts()`
- Modify the function located in `~/tos/kernel/process.c`: `create_process()`
- Modify the function located in `~/tos/kernel/dispatch.c`: `resign()`
- Test case:
  - `test_isr_2`

PacMan

- 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()
  }
}
```

- You were asked to add the call to `resign()` in order to force a context switch that would allow other ghosts to move. This is the essence of collaborative multitasking.
- Once you finish assignment 7, you can remove this call to `resign()`. Since TOS is now implementing pre-emptive multitasking, all ghosts should still move ‘concurrently’.