Timer Service

Objectives

- High-level interrupt API
- Add a Null Process to TOS
- Develop a timer service for TOS

>Sleep- While sleeping take off from ready queue..helps in saving CPU cycles
>U might miss interrupt

High Level Interrupt API

- We now know how to set up and write ISRs. How does the ISR interact with running processes?
- Important: an ISR is NOT a process -- it runs in the context of whatever process happened to be running when the interrupt happened.
- We need a way to synchronize a process with the underlying interrupts: a function called `wait_for_interrupt()`

wait_for_interrupt()

- Goal is to let a process run without worrying about the details of the underlying ISR.
- As in `receive()`, the caller should block until the interrupt occurs.
- We introduce a new process state that a process has while it is blocked: `STATE_INTR_BLOCKED`
- Basic sequence of events:
  - Process calls `wait_for_interrupt()`
  - Process becomes `STATE_INTR_BLOCKED`
  - Interrupt occurs and the appropriate ISR puts the process back to the ready queue.

this process is waiting for some interrupt to happen

Data structure to remember that a particular process is waiting for some Interrupt

Interrupt Handling

- `void wait_for_interrupt(int intr_no)` The process calling this function will become `STATE_INTR_BLOCKED`, then when interrupt `intr_no` occurs, the process is added back to the ready queue.
  - A process can only wait for one interrupt at a time.
  - Only one process can wait for any given interrupt at any given time
  - The only valid values for `intr_no` are `TIMER_IRQ`, `KEYB_IRQ`, and `COM1_IRQ`.

0x60 , 0x61, 0x64

Example

```
void process_a (PROCESS self, PARAM param)
{
      while (1) {
            wait_for_interrupt(TIMER_IRQ);
            kprintf("*");
      }
}
```

- TIMER_IRQ is defined in kernel.h
- The endless loop will print '*' to the screen, but with a short delay after each output.
- Thought experiment: what happens when TIMER_IRQ is changed to KEYB_IRQ?

Keyboard process: A=> press a,shift, release a,shift
Therefore 4 interrupts occurs

Timer Service

```
void sleep()
{
int no_of_times=1000..;
while(ticks-->0)
wait_for_interrupt();
}
```

- Our next goal: Implement a timer service in TOS that mimics `sleep()` on a UNIX system.
- A poor solution: use a long for-loop to burn CPU cycles. This is called *busy waiting*.
- We want the sleeping process to be off the ready queue while it is sleeping and added to the run list when it is time to wake up.

This may voilate the case that ONLY ONE PROCESS CAN SLEEP FOR A PARTICULAR INTERRUPT
Solution: Timer Service Process !! send request to this process, once the no of interrupts are passed, this process replies.

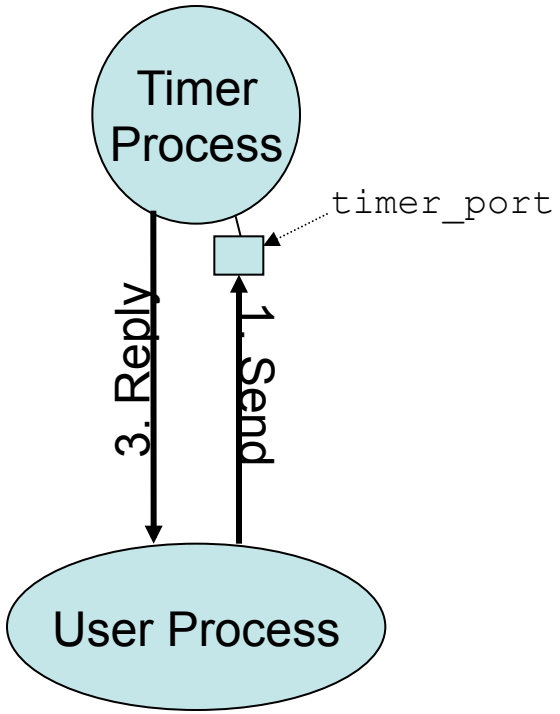
Timer Service Design

- We already have all the pieces needed!
  - Need to use both IPC and the timer interrupt.
- Basic idea:
  - Create a new timer service process.
  - When a process wants to "sleep" for a while, it sends a message to the timer process.
  - The timer process waits until the specified time has passed, and then replies to the process which will "wake it up."
- Note, there is no special `STATE_SLEEP_BLOCKED`. The process is `STATE_REPLY_BLOCKED` while it sleeps.

IPC: receive & reply
Timer Interrupt: wait_for_interrupt

Timer Process
Only one sleeper at a time

```
void timer_process (PROCESS self, PARAM param)
{
    create the timer notifier;
    while(1) {
          msg = receive();
          ticks = parameter transmitted with message;
          while (ticks != 0)
          {
                  wwait_for_interrupt(TIMER_IRQ);
                  ticks--;
          }
          reply to user process;
    }
}
```

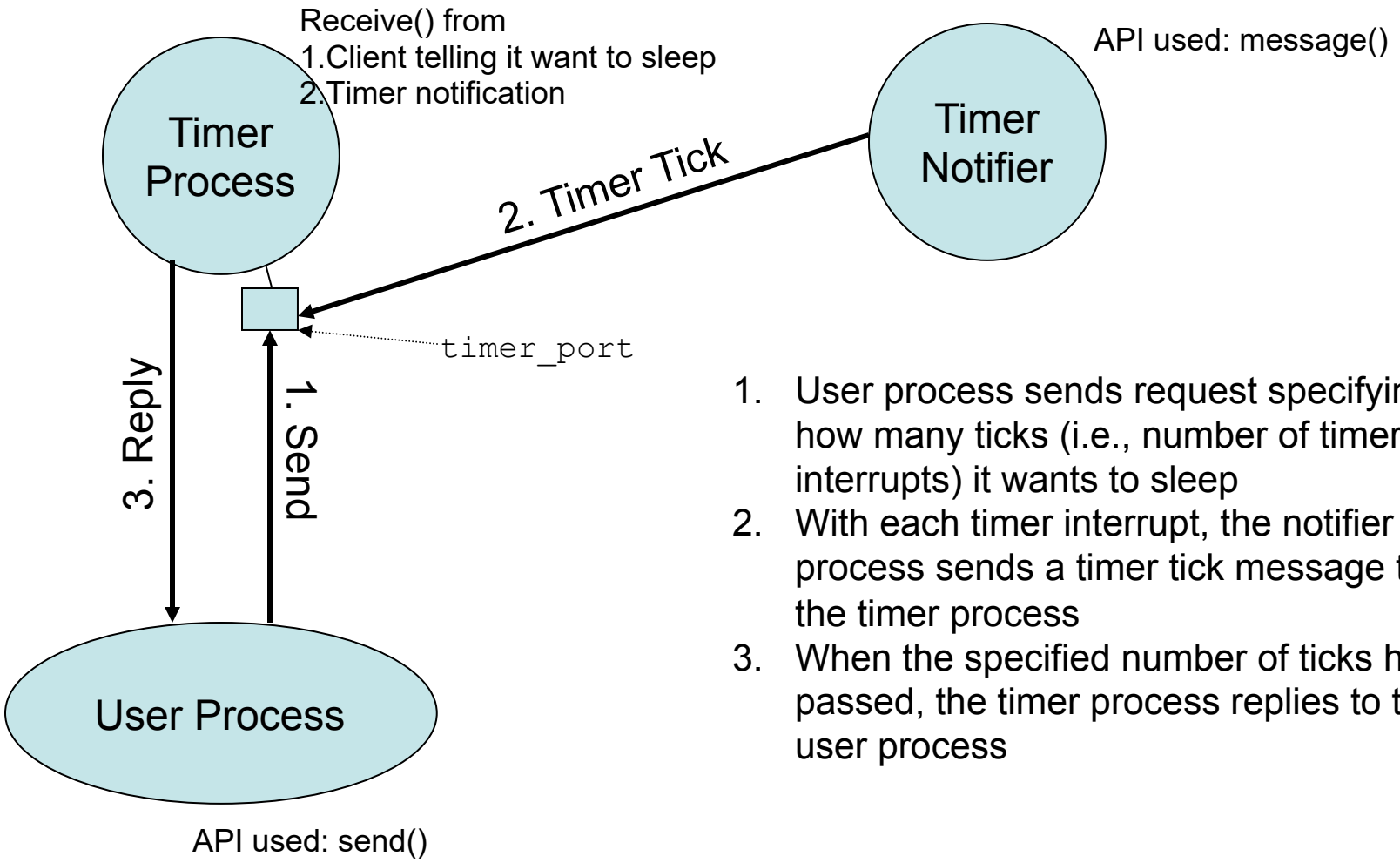


What will happen if multiple clients want to use the timer process simultaneously?

Timer Service Design

- Next step: allow multiple clients that can use the timer process simultaneously.
- Problem: The timer process should do a `receive()` to wait for sleep requests from clients and it needs to call `wait_for_interrupt(TIMER_IRQ)`.
- Big problem: A process can not be `STATE_RECEIVED_BLOCKED` and `STATE_INTR_BLOCKED` at the same time.
- Solution: Introduce a helper process that is waiting for the timer interrupt.
- This helper process is called the *timer notifier*. Its purpose is to notify the timer process whenever a timer interrupt occurs. The notification happens via the standard IPC mechanisms.

Re-Design of Timer Service



1. User process sends request specifying how many ticks (i.e., number of timer interrupts) it wants to sleep
2. With each timer interrupt, the notifier process sends a timer tick message to the timer process
3. When the specified number of ticks has passed, the timer process replies to the user process

Timer Process Message

```
typedef struct _Timer_Message
{
    int num_of_ticks; //no of interrupts
} Timer_Message;
```

- Defined in `kernel.h`
- Instances of `Timer_Message` are sent to the Timer Process by the timer notifier and the user processes
- Member:
  - `num_of_ticks`: number of ticks (i.e., number of timer interrupts) that the process wants to sleep

Timer Notifier

```
void timer_notifier(PROCESS self, PARAM param)
{
      while(42) {
            wait_for_interrupt(TIMER_IRQ);
            send message to timer process;
      }
}
```

- Timer Notifier process should have priority 7 (highest priority)
- Sending the message to the timer process should be done via `message()`
- The message does not need to carry any data

Timer Process
Multiple sleepers (1)

- This slide gives some implementation hints for a Timer Process that can handle multiple clients.
- The Timer Process needs to maintain a list of clients and how many ticks each client wants to sleep.
- Upon arrival of a Timer Notifier message, the tick counter is decremented for each client.
- If a counter reaches 0, that particular client is woken up by replying to it.
- Note: it is perfectly possible that the order in which sleep requests are received is not the same order in which the Timer Process will reply to the clients. This is quite possible with the TOS-IPC and is called *out-of-order-replies*.

Timer Process
Multiple sleepers (2)

```
void timer_process (PROCESS self, PARAM param)
{
    create the Timer Notifier;
    while(1) {
          msg = receive(); //either from notifier or a client
          if (msg was sent from a client) {
                    register number of ticks client wants to sleep;
                    continue;
          } else {
                    // Message must have come from Timer Notifier
                    for (all clients doing a sleep) {
                              decrement tick counter;
                              if (tick counter == 0)
                                        // Wake up client
                                        reply to the client;
                              }
                    }
          }
    }
}
```

Timer Process
Multiple sleepers (3)

- Some implementation hints for the internal book-keeping of the Timer Process:

```
int ticks_remaining[MAX_PROCS];
```

Remember that `MAX_PROCS` is the maximum number of allowable TOS processes.

>Max allowed is 18 removing the notifier and timer process.

poiner to a PCB

`ticks_remaining[i]` corresponds to `pcb[i]`
`ticks_remaining[i] == 0` means that process `pcb[i]` is currently not doing a sleep

// pcb is the global variable indicating base address of pcb array
// client_proc: pointer to client pcb
// i gives the no of elements in pcb

```
PROCESS client_proc;

int i = client_proc - pcb;  // pointer arithmetic!
assert(client_proc == &pcb[i]);
```

`i` can now be used as an index into `ticks_remaining[]`

Timer Process notes

- The timer process gets created in `init_timer()`
- Timer Notifier must have priority 7 (why?)
- Timer service should have priority 6 (why?)
- How can the ticks_remaining[] array be made more efficient? Hint: From O(n) to O(1) by using a differential list.

sleep()

- Create a function `sleep()` similar to the Unix version that wraps the communication with the timer process:

```
void sleep(int ticks)
{
        Timer_Message msg;
        msg.num_of_ticks = ticks;
        send(timer_port, &msg);
}
```

- Global variable `timer_port` should be initialized in `init_timer()`

Timer Interface

- `void init_timer()`
  Initialize the timer process:
  - After initialization the global variable `timer_port` is a communication port owned by the timer process.
  - The timer process should accept and process messages of type `Timer_Message` (defined in `kernel.h`) as explained on earlier slides.
- `sleep()`: wrapper function that clients can use to send `Timer_Message` to the timer process.

Null Process

- TOS function `dispatch()` assumes that there is at least one process on the ready queue.
- It can happen, that all processes are blocked because everyone is waiting for something
- In that case we need to have a special process that gets scheduled, called the *Null Process*
- Some details:
  - It should be created with priority 0 (why?)
  - It must not do anything that may block (why?)

Assignment 8

- Implement the following functions:
  - `wait_for_interrupt()` (in `intr.c`)
  - `init_timer()` (in `timer.c`)
  - `init_null_process()` (in `null.c`)
- Modify existing functions:
  - `init_interrupts()`
  - `isr_timer()`
  - `print_process()`
- Test cases:
  - `test_isr_3`
  - `test_timer_1`

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
   }
}
```

- For the delay you were told to do busy waiting via a long for-loop that does nothing. Replace this delay with a call to `sleep()`. The animation of multiple ghosts moving through the maze should now be smooth.