Announcements

- Homework 2 should be released soon. It will be due on Feb. 28th at 5pm CST.
- Lab 1 reports are due at the beginning of Lab 2 this week.
- Lab 2 starts this week.

Overview

- Last lecture: basic concepts in periodic tasks.
  - Real-time periodic tasks and the problems of jitter and drifts.

- Today: Implementation of real time periodic task in POSIX RT.
  - Timer signals (software interrupt)
  - Signal handlers (Interrupt Service Routines – ISR)

Review: Potential Timing Problems

- E1: Process could be swapped out of memory (causing drift and jitter).
- Solution: Pin it down in memory! In LynxOS, processes with priority greater than 16 (default) will not be swapped out.

```
1. current_time = read_clock()                    //E2 if preempted, drift
2. If (START_TIME - current_time < 10 msec) {//report too late and exit}
3. sleep(START_TIME - current_time)               //E3: if preempted, drift
loop
    4. current_time = read_clock()                // same problem as line 1
    5. wake_up_time = current_time + 20 msec      //E4: drift if current time is delayed at line 4
    6.  Read sensor data                          //E5: if preempted, input jitter
    7.    //do work
    8. current_time = read_clock()                // same problem as line 1
    9.    //send control data to the device        //E6: output jitter (caused by preemption
                                                       or variable execution time)
    10. sleep(wake_up_time - current_time)        // same as line 3
 end_loop
```

Summary: Every line that manipulates time or performs I/O has problems.

Solution Approach: When I/O is Short (1)

- To solve the drift problem, use a periodic, hardware based timer to kick start each instance of periodic tasks. The tasks will be ready at the correct time instants.
- To minimize the jitter problem, perform the I/O in the timer interrupt handlers.
- Inside the handler, the processor will only be interrupted by another interrupt with a higher priority. See nested interrupts from previous lecture.
- As long as each task finishes before its end of period, I/O can be done at nearly the regular instants of timer interrupt, the highest regularity.
- The I/Os will still interfere with each other. But when the I/O times are short, this method works just fine. Also, you may use condition variables which will simplify the actual implementation.

Solution Approach: When I/O is Short (2)

- Initialization: // Lock mutex *wake_task_1_up.*

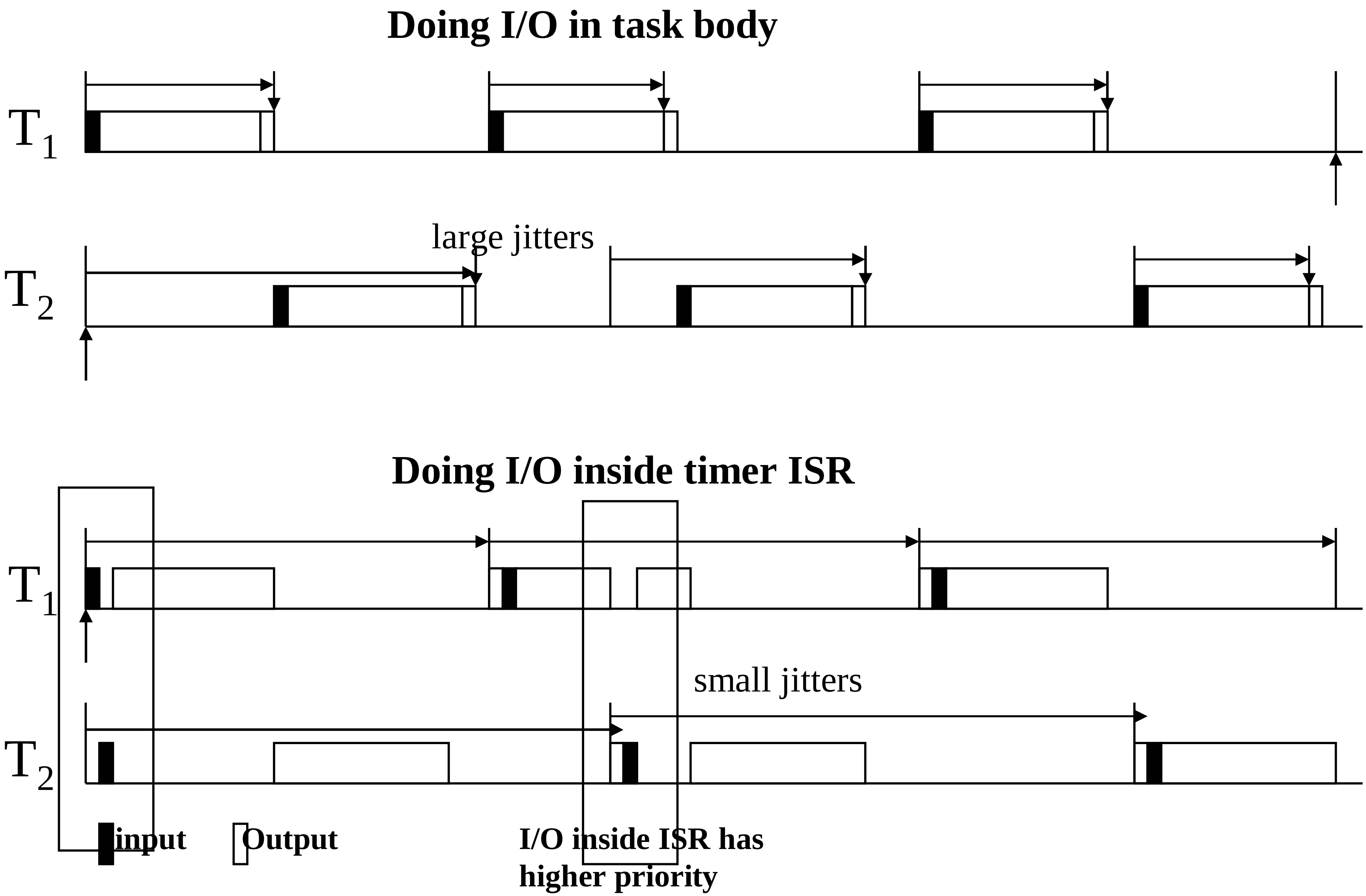
- Timer_interrupt_handler() // *by convention*, handlers execute before application tasks

```
{
        // Do I/O ONLY – Why can't we do both work and I/O in the handler?
        // Unlock mutex wake_task_1_up
}
```

- Task_1()

```
{
        Loop
                // Lock mutex wake_task_1_up
                // Do non-I/O computation
        End loop
}
```

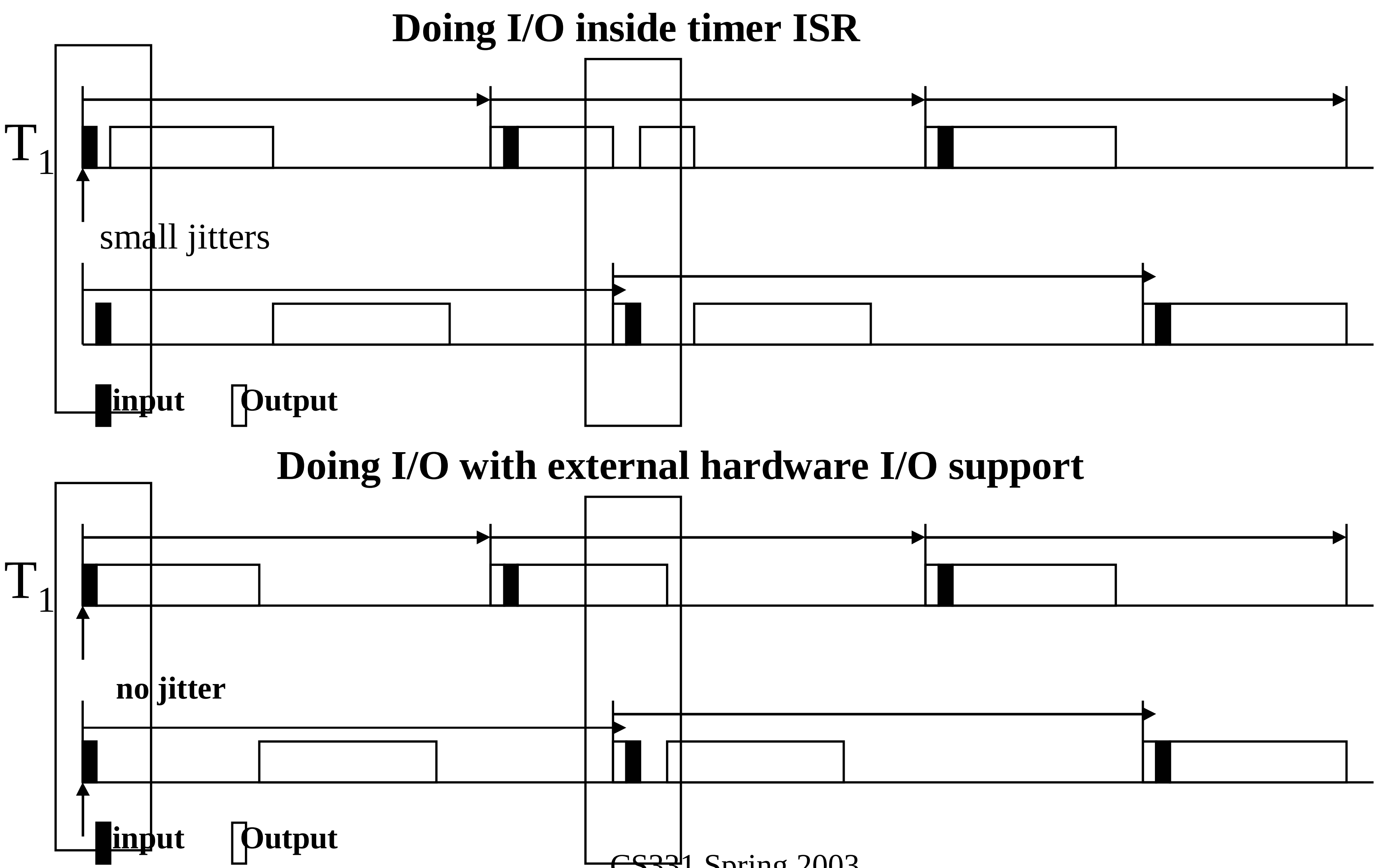
Solution Approach: When I/O is Short (3)



Solutions Approach: When RT I/O is Too Long (1)

- In this case, hardware solution is called for. In high end A/D – D/A cards, there is a hardware timer for each input and output channel. For example,
  - ch 0 is input.
    - We can program its timer to sample the input line every 20 ms, starting at time 0.
    - put the data in ch 0’s input buffer. (jitter free read)
    - Generate an interrupt to trigger Task_1.
  - Task_1 ISR: kickoff the Task_1 body.
  - Task_1 body: read the data, do work and put result to ch1 buffer.
  - ch 1 is output. We program its timer to output data from its buffer every 20 ms starting at 20 ms. (jitter_free output. Do not start at time zero, why?)
- As long as Task_1 body finishes its work within 20 ms, there is no jitter even if Task_1 body is preempted, why?

Solutions Approach: When RT I/O is Too Long (2)



Signal Architecture (1)

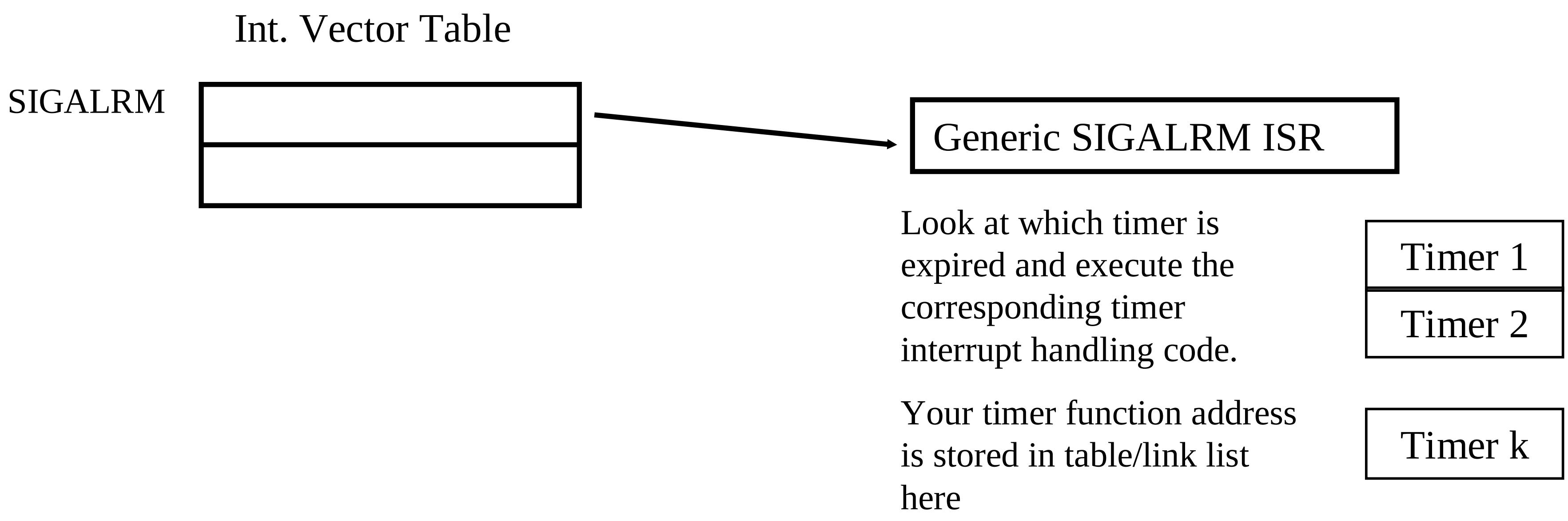
- POSIX signals are implemented by software interrupts (e.g., SIGALRM)
  - Each periodic task thread will use a timer interrupt.
  - There can be as many threads as memory allows.

- How many real time clocks do we need? Why?

- How many entries in the interrupt vector table do we need?

Signal Architecture (2)

- A simple design: put the address of your ISR into the interrupt vector table. This is a valid and simple design. The drawback is that you need one alarm signal per thread. This can be done but not very efficient.
- An improved design. Recall that each thread has its own software timer. And all the threads wait on the same SIGALRM. To support the sharing of SIGALRM:



POSIX RT Timer Interrupt Handler

- POSIX Timer generates signal (software interrupts), SIGALRM. Action (ISR) for SIGALRM is therefore needed.

- Recall that you first define an interrupt type number which stores the address of your interrupt service routine.

- By POSIX coding convention, you define a generic action and then bind the action to a specific handler your write.

```
…
struct sigaction my_action, old_action  //actions to catch a given signal arrives
...
// establish signal handler for SIGALRM
memset(&my_action, 0, sizeof(my_action)); // clear up the memory, a convention

my_action.sa_handler = timer_handler;  // the "action" is to handle timer interrupt
//your ISR address for timer interrupt is copied into the sa_handler field

return_code = sigaction(SIGALRM, &my_action, &old_action); //binding action & signal
```

Summary

- Today
  - We have reviewed the issues in programming periodic tasks.
    - jitters and drifts.
  - How to use timers, and signal handler to construct a real time periodic tasks.
  - You are now ready to do Lab 2.
- Next Time
  - Basic signal processing.

Appendix: POSIX Real Time Clocks

- In POSIX RT defines the structure of time representation. There is at least 1 real time clock.
- Clock resolution is hardware dependent. (10 msec in PC without add-on high resolution real time clock cards)

- #include time.h
- ...

```
struct timespec  current_time, clock_resolution
return_code = clock_gettime(CLOCK_REALTIME, &current_time);
//check return_code...
printf("%d %d current time in CLOCK_REALTIME is \n",
        current_time.tv_sec,            // the second portion
        current_time.tv_nsc);           // the fractional portion in nano-sec presentation

return_code  =  clock_getres(CLOCK_REALTIME, &clock_resolution)
//check return_code...
printf("%d %d  CLOCK_REALTIME's  resolution is \n",
clock_resolution.tv_sec, clock_resolution.tv_nsc);
```

Appendix: Interval Timer Structure

POSIX timers are used to generate SOFTWARE INTERRUPTS (signals)

```
#include <signal.h>
#include <time.h>

//CREATE THE TIMER
timer_t timer_id;
//under POSIX, each thread defines its own timer for itself.
return_code=  timer_create(CLOCK_REALTIME, NULL, &timer_id);
//NULL: no signal mask used to block other signals, if any, sent to this task
-------------
//INITIALIZE THE TIMER
struct itimerspec timer_spec, old_timer_spec;
//old timer allows us to save the old timer definition so you restore it if you need to
timer_spec.it_value.tv_sec = 10;                    // 1st expiration time
timer_spec.it_value.tv_nsec = 0;
timer_spec.it_interval.tv_sec = 0;                  // task period, 50Hz
timer_spec.it_interval.tv_nsec = 20000000;

timer_settime(timer_id, 0, &timer_spec, &old_timer_spec)        //initialize the periodic timer
```

Appendix: Putting It Together

```
#include header files signal.h, time.h, errno.h, stdio.h //errno.h allows to decode the return code

void timer_handler( ) // it is invoked by the timer, not called by your software
{do your device I/O}  // use global variables to communicate between main and handler

void main ()
{
    //ask user to input the task rate, max volt and min volt. Check for validity. If the resolution is too high
    and if the voltages are too high/low
    //create your timer and initialize it
    //sets up your SIGALRM handler

    while (1) {
            sigpause(SIGALRM); //wait for signal SIGALRM
            //do your computation, make the handler code as small as possible to reduce jitter.
    }
}
```

Appendix: Using Signal Mask

Sigpause( ) is an old fashioned Unix function. POSIX RT defines sigsuspend( ) that uses a flexible signal mask. sigpause(signal_number) is still supported for backward compatibility.

```
void main ()
{
    sigset_t wait_signals, old_signals; //set of signals to wait for
    ...
    sigemptyset(&wait_signals); //initialize wait_signals to be an empty set
    sigaddset(&wait_signals, SIGALRM); //add SIGALRM to the set
    sigprocmask(SIG_BLOCK, &wait_signals, &old_signals); //install the signal set as a
    blocking mask
    while (1) {
            return_code = sigsuspend(&wait_signals); //wait for signal SIGALRM
            //do computation, make the handler code as small as possible to reduce jitter.
    }
}
```