COMPONENTS OF A MULTITASKING SYSTEM

Components of a Multitasking System

A multitasking development environment should provide tools to

- Create separate tasks
- Schedule running of the tasks- e.g., based on priorities
- Share data between tasks
- Synchronize tasks with each other and with external events
- Prevent tasks from corrupting each other
- Control the starting and stopping of tasks

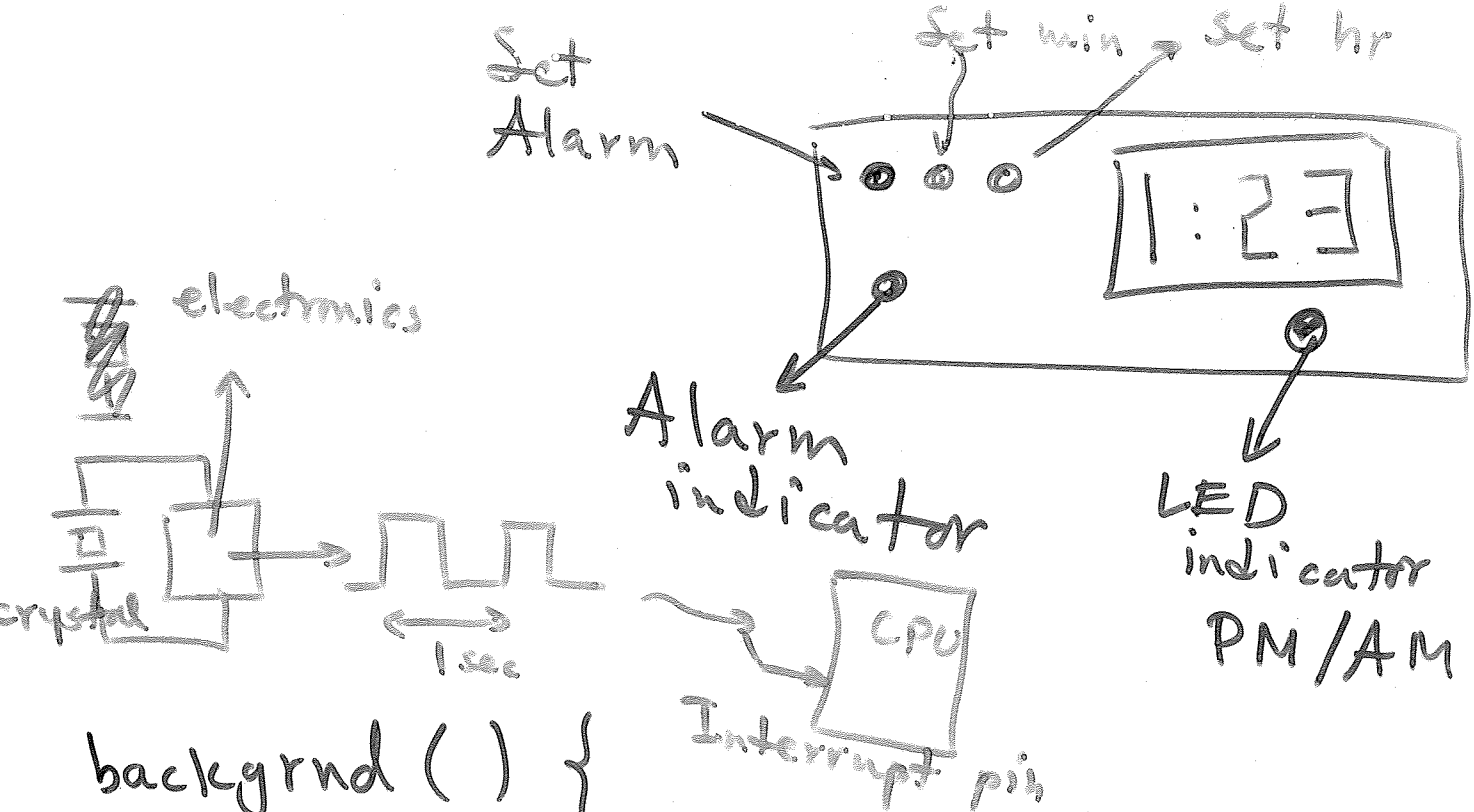
Multitasking Executives

- Foreground-Background Systems
- Co-routines
- Cyclic Executive Approach
- Full-featured RTOS

Foreground-Background Systems

Small systems of low complexity may not need RTOS

- Can be implemented by
  - — a set of interrupt driven tasks: *foreground*
  - — a set of non-interrupt driven tasks: *background*
- Background tasks run in an infinite loop
  - — low-priority, non time-critical
- Foreground tasks are high priority tasks initiated by external requests (interrupt driven).
- Example: digital clock
  - — keeping track of the time → task (1)
  - — display mechanism, push buttons (set time or alarm) → task (2)
  - — task (1) has higher priority than task (2)
  - — time update → foreground interrupt service routine
  - — display, push buttons → background polling loop




```
background ( ) {
    do-init();
    while(1){
        check push buttons; → do-task1();
        do-task2(); →/* display time
        }
    }

foreground ( ) {
    increment time → interrupt service routine triggered by a periodic (1 sec) signal
    }
```

counter++ ;

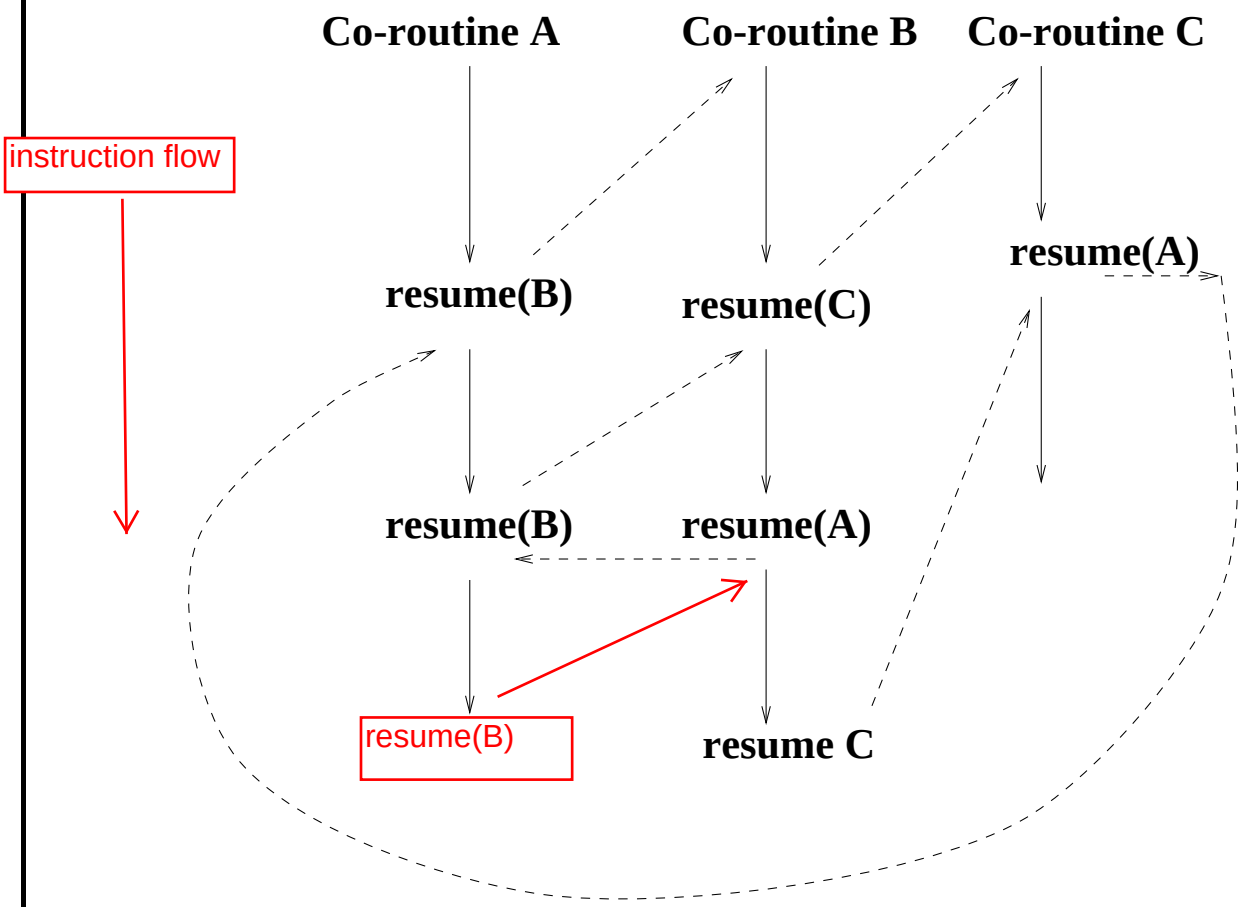
Co-routines

Early multitasking technology used co-routines to perform multitasking.

— *resume* function keeps track of where the program has left, or here it should branch to

— at processor level: achieved by updating the program counter

— not adequate for complicated tasks



Cyclic Executive Approach

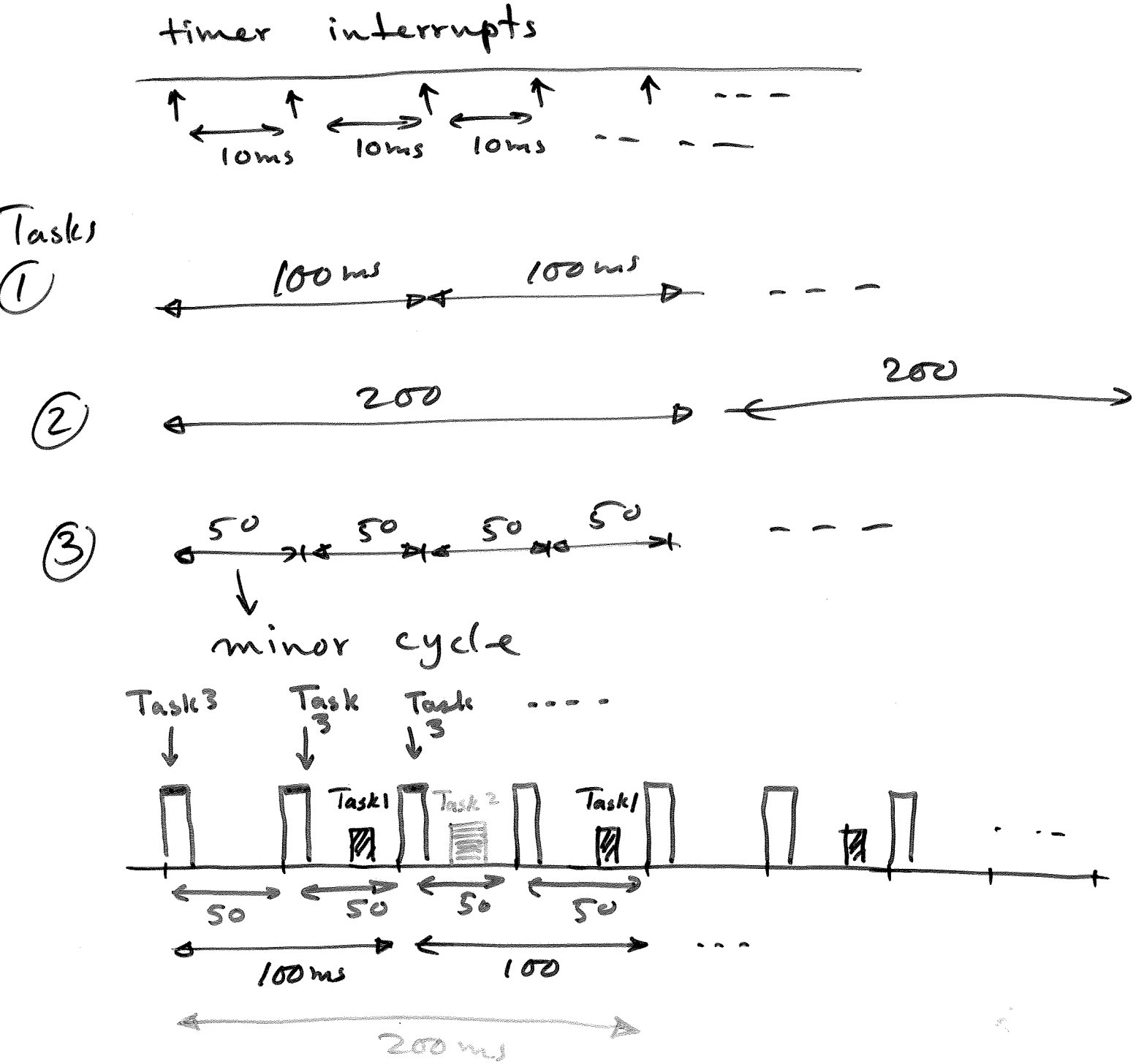
Can be used for simple multitasking applications
Example: Suppose there are three concurrent tasks:

- **First task:** Reads analog data every 100 ms
- **Second task:** Reads a digital input every 200 ms
- **Third task:** Updates an analog output every 50 ms

A timer interrupt updates a global variable every 10 ms, e.g.,

```
/* global long integer */
unsigned long timerTics=0;
....
void timerIsr(void)
        {
                timerTics++;
        }

Question: How long would it take for timerTics
          to overflow?

(0xFFFFFFFF...F= 2^64*10ms ~ 5.8*10^9 years)
```

how many bits?


timer interrupts
10ms
10ms
10ms
Tasks
1
100 ms
100 ms
2
200
200
3
50
50
50
50
minor cycle
Task3
Task3
Task3
Task1
Task2
Task1
50
50
50
50
100ms
100
200 ms

Implementing the tasks:

Data structure- defined for periodic tasks:

```
typedef struct {
  int state; /* initialized to zero */
  int timer; /* next time to run */
  int periodTicks; /*period in 10ms ticks */
  int offsetTicks;/*initial 10ms offset ticks*/
  } pertask_t;

/* function to implement a periodic task */
void Task(pertask_t *t, void (*func)() ){
   switch (t->state) {
   case 0:
     t->timer = timerTics + t->offsetTicks;
     t->state = 1;
     break;
   case 1:
     if (timerTics >= t->timer) {
       t->timer = timerTics + t->periodTicks;
       func();
      }
     break;
    }
 }
```

Code to perform the above tasks:

```
void cycle_exec(void){
  /* Initialization */
   pertask_t  task100, task200, task50;

   task100.state=0; task100.offsetTicks=27;
   task200.state=0; task200.offsetTicks=32;
   task50.state=0;  task50.offsetTicks=20;
   task100.periodTicks = 10;
   task200.periodTicks = 20;
   task50.periodTicks = 5;

   for (;;){
   /* perform analog input task */
   Task(&task100, (FUNCPTR) aInFunction );

   /* perform digital input task */
   Task(&task200, (FUNCPTR) dInFunction );

   /* perform analog output task */
   Task(&task50, (FUNCPTR) aOutFunction );
  }
 }
```

— each function quickly checks if its corresponding timer has expired or not

— if so, it calls `aInFunction()`, `dInFunction()`, or `aOutFunction()`.

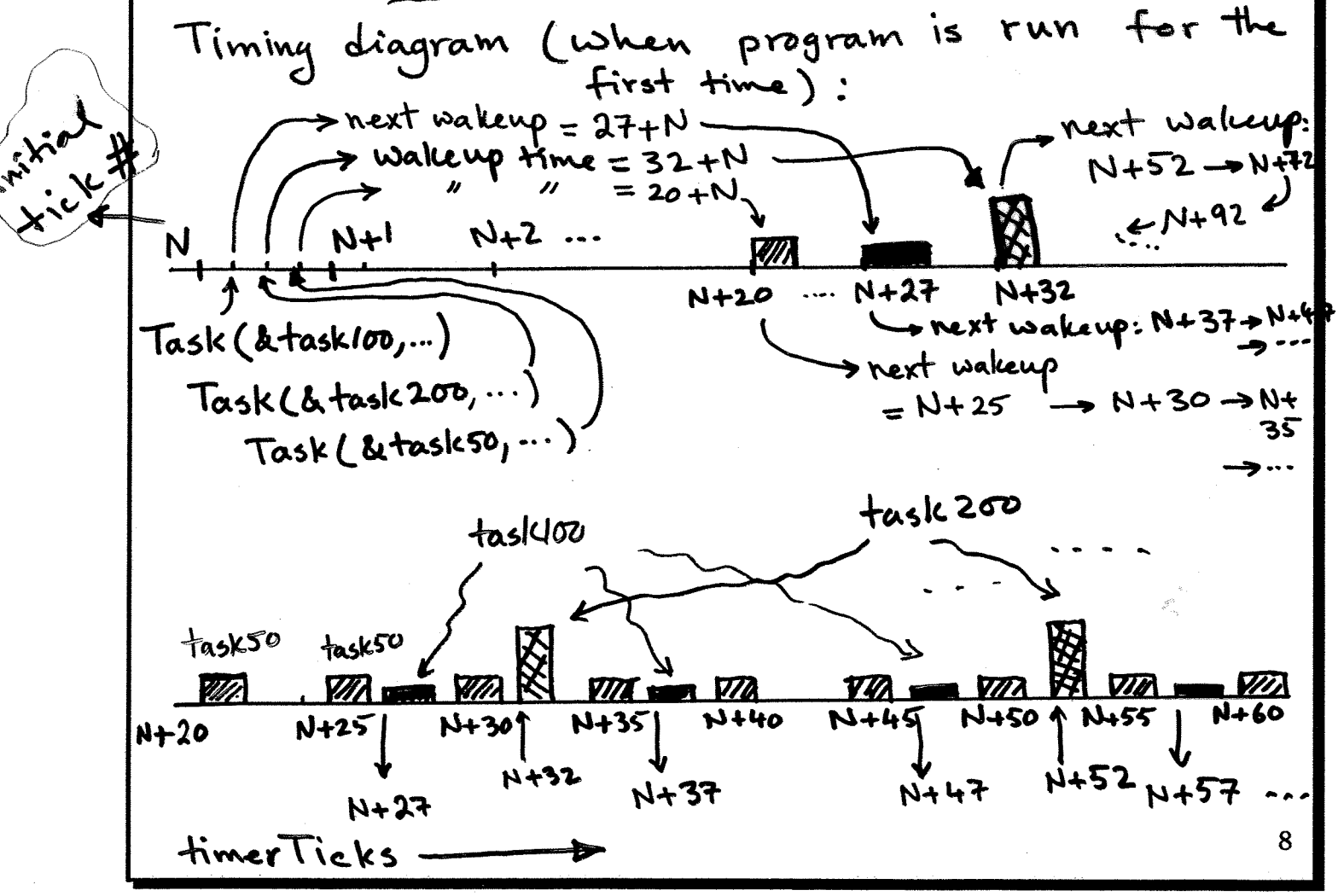
— no priorities assigned to tasks

— tasks are called in turn: *round robin*

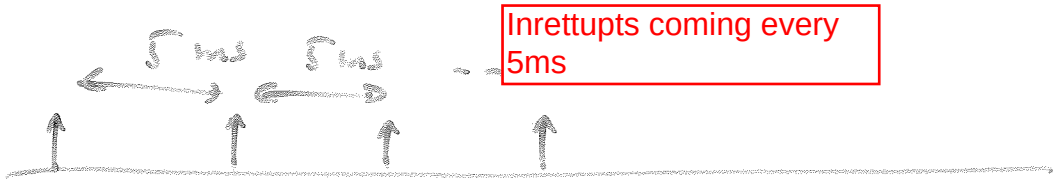
Communication between the tasks:

— can be done through global data

— is up to the receiving task to check for any data



Another method for implementing cyclic executive:
-Assume tasks A, B, C, and D take 2.5ms, 4ms, 2ms, and 2ms, respectively.
- Period of A is 10ms
- B, D, C, have equal 20ms periods

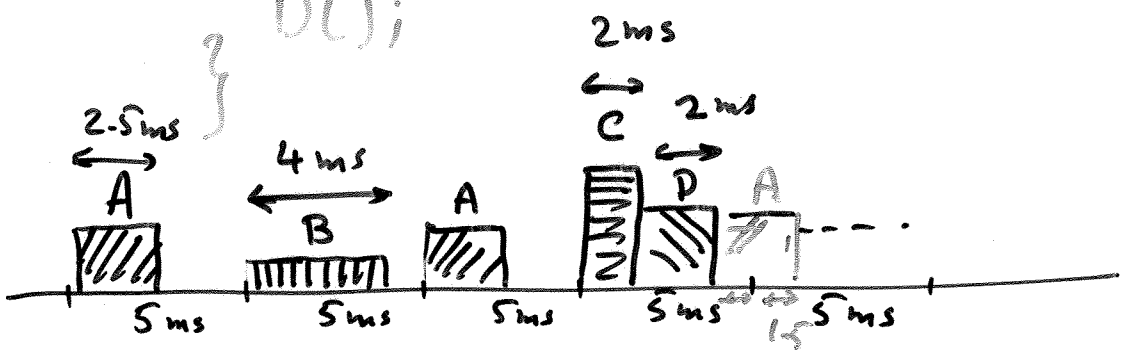


Inrettupts coming every 5ms

```
while (1) {
    wait for interrupt()
    A();
    wait for int().
    B();
    wait --
    A()
    wait
    C();
    D();
}
```

```
while (flag==0);
flag=0
```

Done in ISR:
- Set flag to 1
- Return




```
volatile unsigned char flag = 0
```

Real-Time Operating Systems

Complex real-time systems:

- Consist of a large number of tasks
- Complicated devices and device drivers
- Rich set of debugging tools

Real-time operating system:

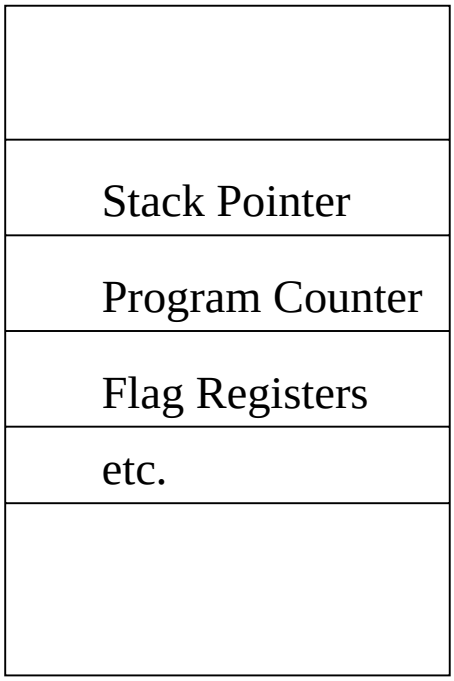
- Real-time kernel: Software calls allowing for
  - — creation of independent tasks
  - — timer management
  - — inter-task communication
  - — memory management
  - — resource management
- Software development/debugging tools: e.g., command line interpreter (Windshell in VxWorks)
- File system

Tasks: Units of work active simultaneously
e.g., Is taking courses a multitasking activity?
YOUR TIME = CPU TIME

Task Control Block (TCB) Model

TCB is a data structure that contains information about the task.

— there is one TCB corresponding to each task:



$\Longrightarrow$ Task creation means the creation of a TCB

==> TCB: A repository for any information that may vary from one task to another

Each TCB is like a page of a notebook where you record important info about different activities

TCB is a data structure that contains:

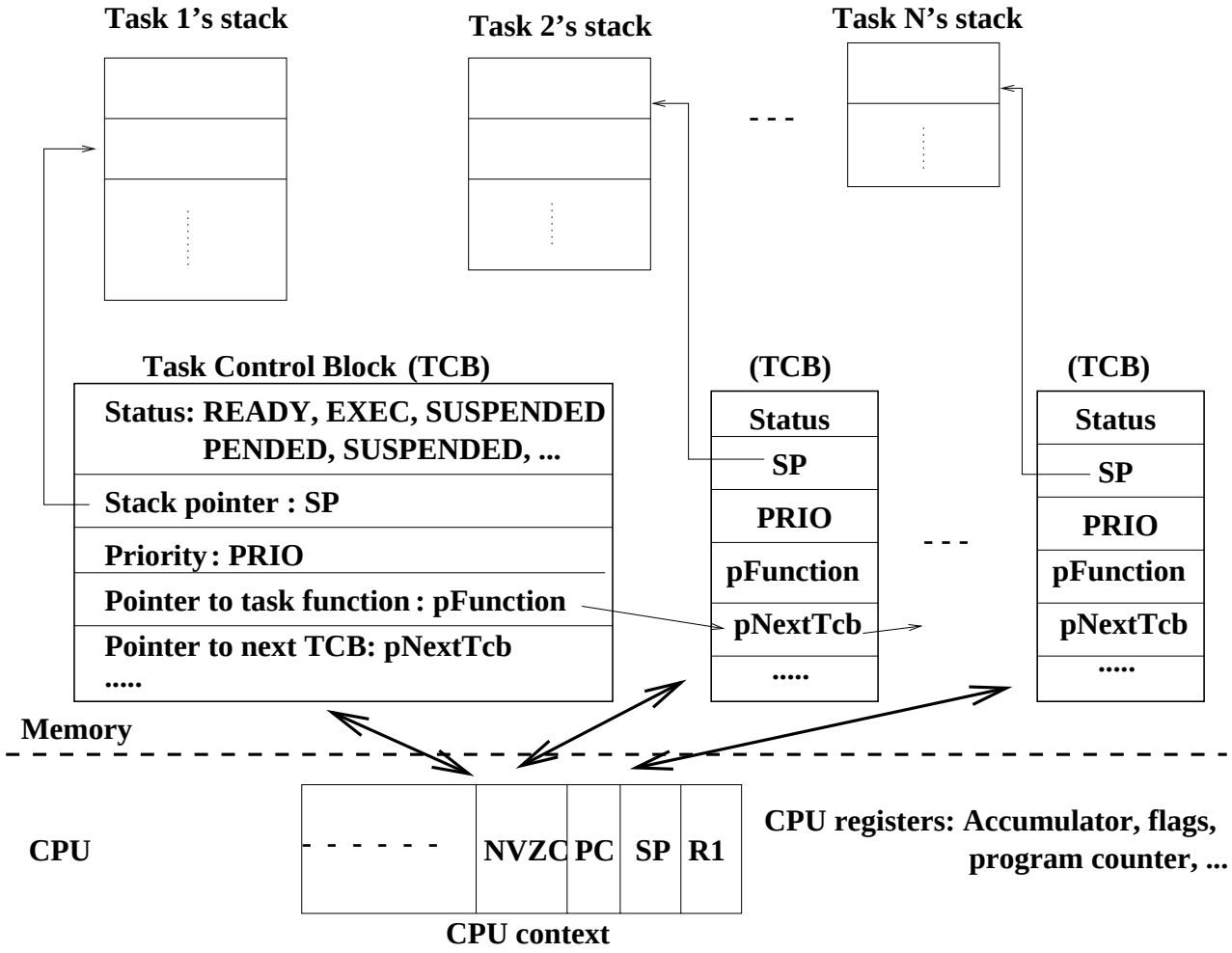
- task's priority
- task number
- a pointer to the task's function
- etc

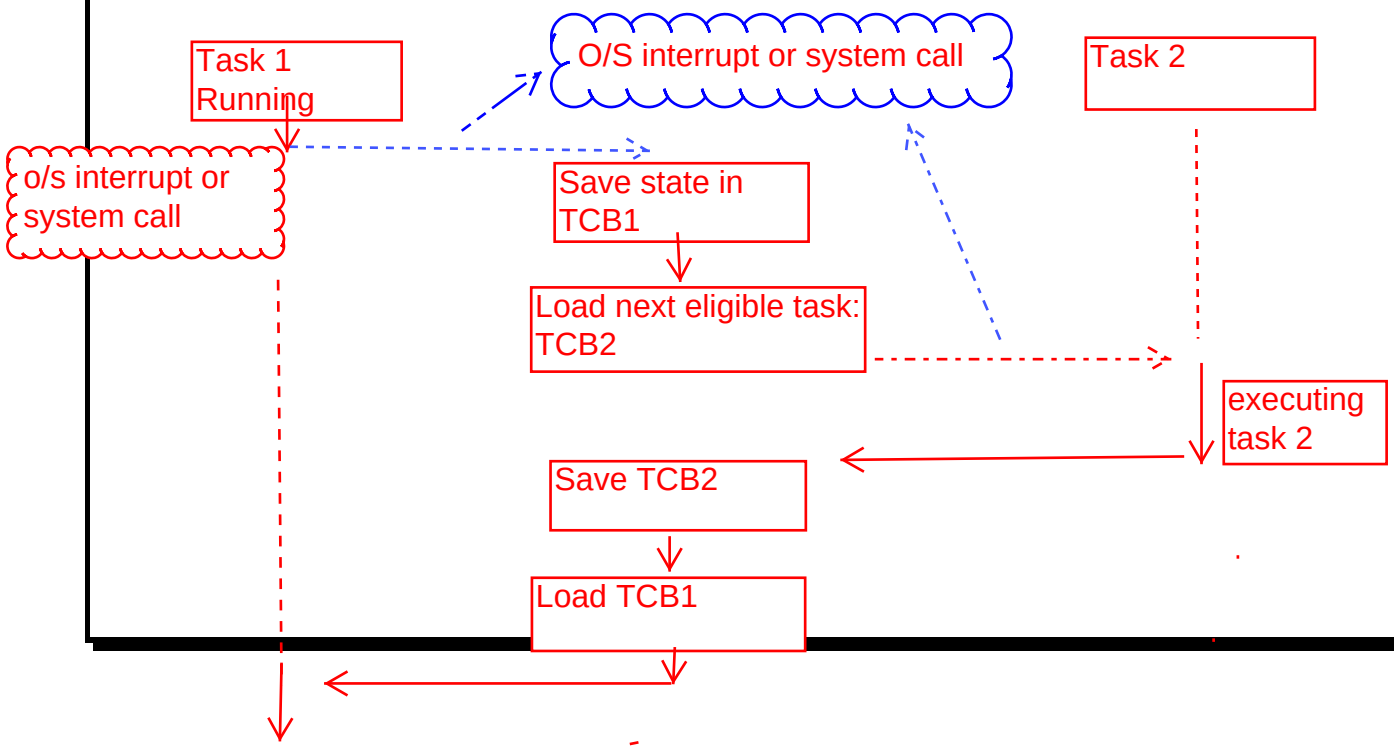
TCB data structure can take the following form

```
typedef struct structTcb {
    int priority;        /* Task priority. */
    int taskNumber;      /* task number. */
    /* Pointer to task function. */
    void (*taskFunctionPtr)();
    /* More data  ... */
} typeTcb;
```

- State: READY, DELAYED, SUSPENDED, PENDED
- Program Counter
- CPU registers
- Memory Limits for Stack

TCB for multiple tasks:





Creating a task: allocate memory, set its priority, assign a task number, ...

```
typeTcb * makeTask(void (*taskFunctionPtr)(),
                                        int taskNum)
 {
        typeTcb * tcb;

        /* allocate a block of memory */
        tcb = getNewTcb();
        /* default priority */
        tcb->pri = DEFAULT_PRIORITY;
        tcb->taskNum = taskNum;
        tcb->taskPtr = taskFunctionPtr;
        return(tcb);
 }
```

Pointer to a data structure

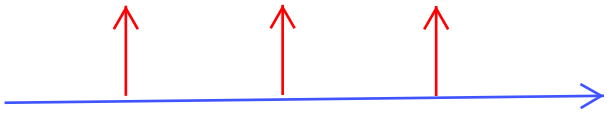
Timer (or Tick) Interrupt

Tick Interrupt: A special ISR triggered periodically by a system call.

After the scheduler determines that another task has to run, ISR performs the following:

1. Stores the interrupted task's program counter in the task's TCB
2. Writes processor's other registers into same TCB

3. Chooses the TCB of a ready task and copies processor' s registers corresponding to that task into the CPU registers.
4. Gets the program counter value (from TCB) and *writes* it onto top of the stack.
5. Executes a return from interrupt



Tick interrupts: During each tick ISR, scheduler is called which may or may not call the dispatcher (dispatcher is a part of scheduler)

Kernel

Responsible for management of tasks and communication between tasks.

Set of function calls that allow for:

- Creation of tasks
- Scheduling of tasks
- Time management: single shot, periodic timers, start/stop a timer
- Inter-task communication
- Resource management

API (Application Programming Interface): An interface to that allows a programmer to use kernel services

Scheduler

Part of the tick ISR: Which task to run next?
Decided based on:

- Priorities assigned to the tasks
- Ready to run?

  — Preemptive scheduling: highest priority *ready* task is executed

Context Switch Mechanism

Context:

- All CPU registers

  — instruction register, program counter, any status registers

- Task's local variables
- Subroutine nesting information

  — a task may be 7 subroutine calls deep when the context switch is made.

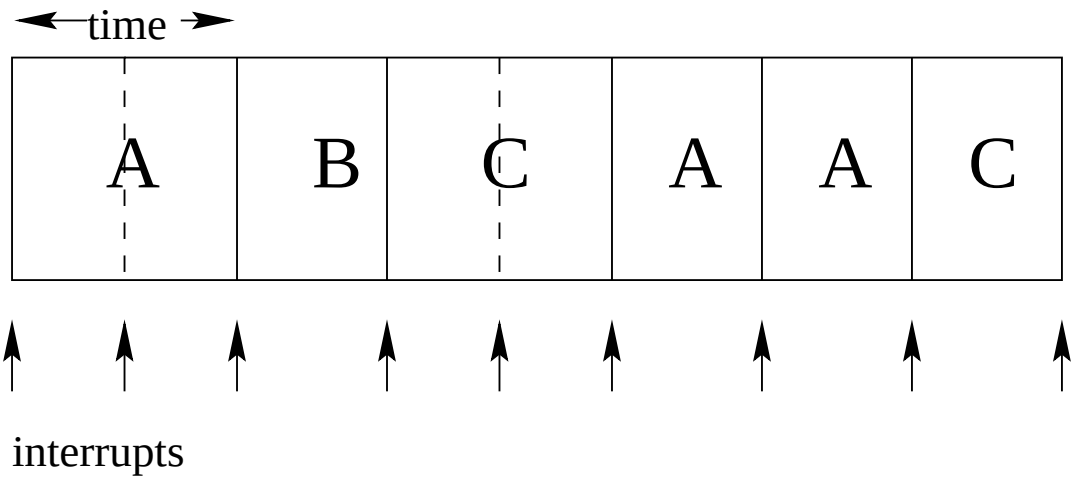
  — when the task is resumed the task's code must be able to return from all 7 subroutines

  — local variables and subroutine nesting level are preserved by giving each task its own private stack

When a task switch occurs:

- its context is saved
- context of the next task to run is restored

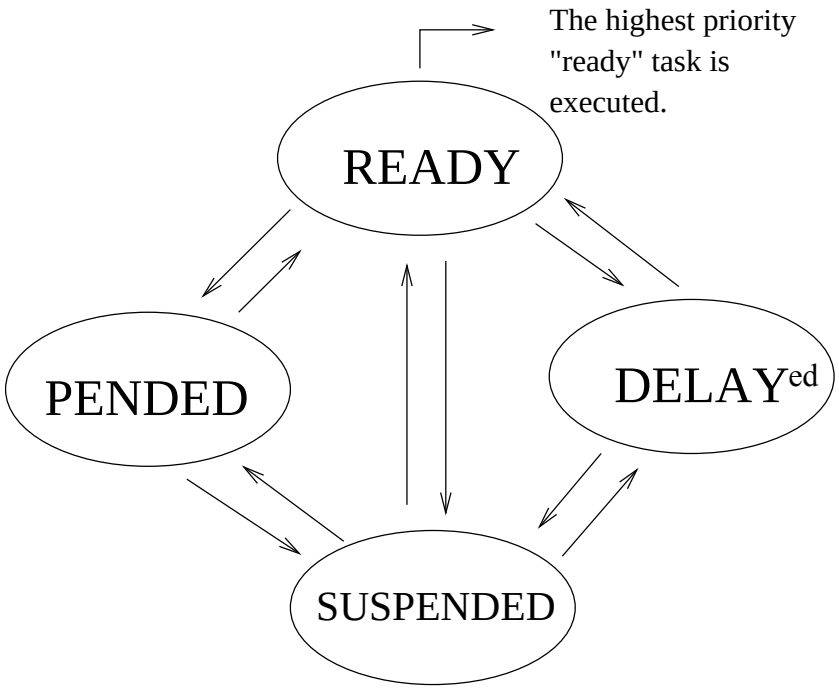
$\longrightarrow$ **Saving/restoring is called context switching**



There are 2 mechanisms that a context switch may occur:
1. O/S ISR (tick interrupt scheduler): Sysnchronous
2. Task function calls an O/S function that results in a context switch (delaying or taking a key that is not available

Task States

O/S is responsible for coordinating tasks activities



Pended: Not running or blocked and placed on a queue, e.g., pending on a semaphore

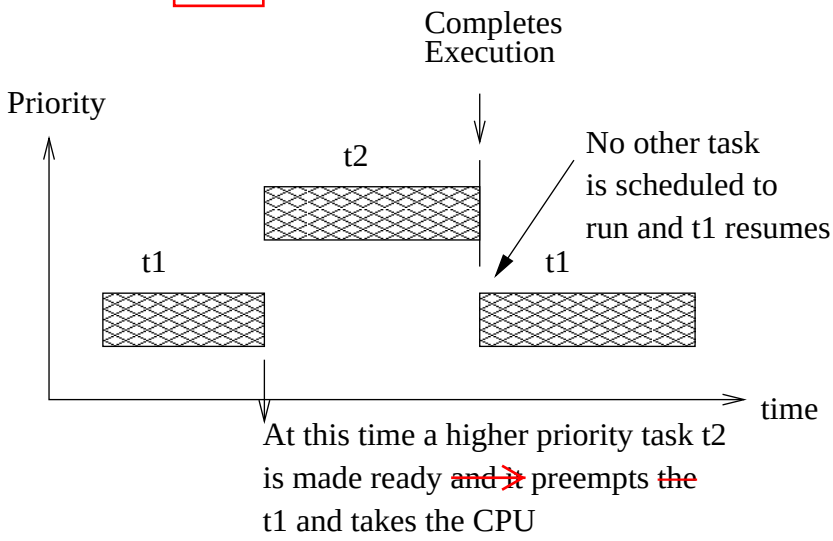
Delayed: Sleeping, blocking and waiting for expiration of a timer (*taskDelay()*)

Suspended: Stopped for debugging or when initialized (*taskInit()* in VxWorks)

Priority Based Kernels

Control of CPU given to one task at a time, i.e., the highest priority, ready task

Pre-emptive Kernel: Higher priority task preempts a lower priority task that is ready



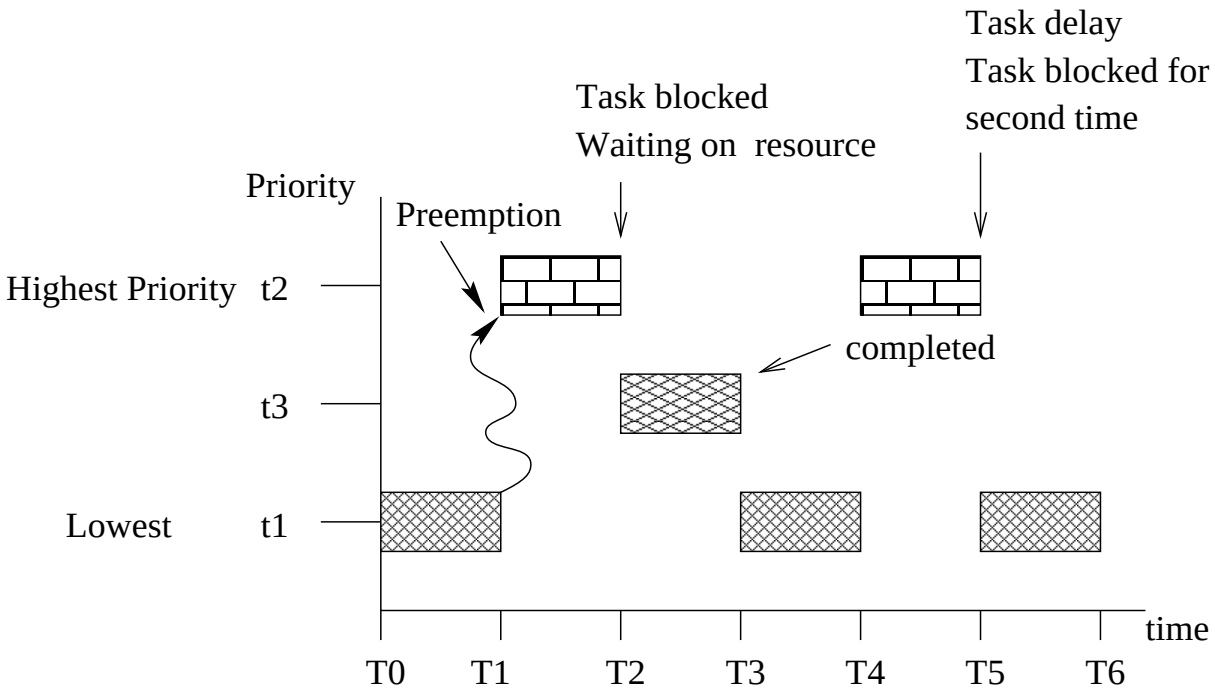
Note: Context switching is not delayed until next tick. Rescheduling can occur at any time as a result of ISR tick interrupt or kernel calls that can cause context switching

Priority of a Task: Each task is given a specific priority in using the CPU which is usually an integer.

Priority assignments in two RTOS's.		
OS	Highest Priority	Lowest Priority
VxWorks	1	256
QNX	30	1

Running tasks with different priorities

Example: Tasks activities vs time



Re-entrant Code

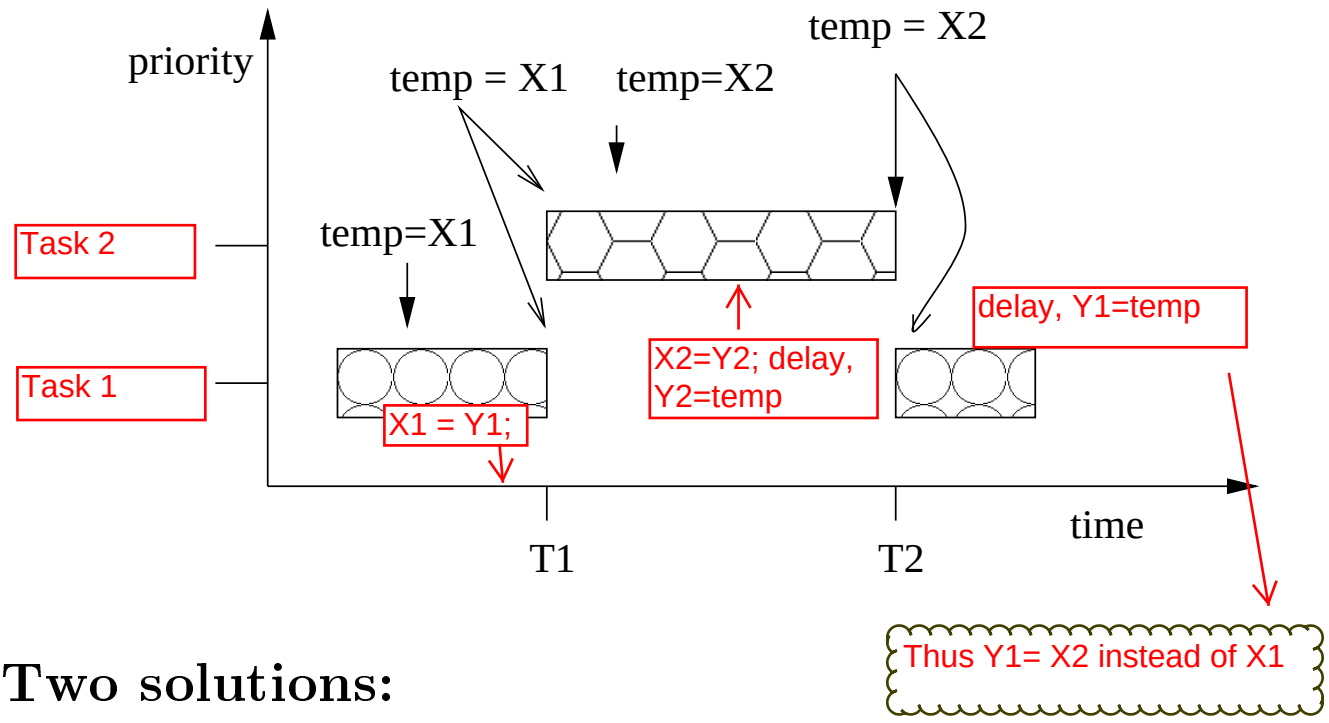
Application code in a preemptive system must be re-entrant

```
int temp; (Global to all tasks/functions)

void swap(int x, int y) {

        temp = x;
        x = y;
        delay()
        y = temp;
}
```

Data corruption can result if swap() is called by several tasks:

```
taskone() {
    ~~~~~
    ~~~~~
    swap(X1, Y1);
}
```

```
tasktwo() {
    ~~~~~
    ~~~~~
    swap(X2, Y2);
}
```



Two solutions:

1. Use local variables
2. Protect the global variable

Scheduling Tasks with the Same Priority

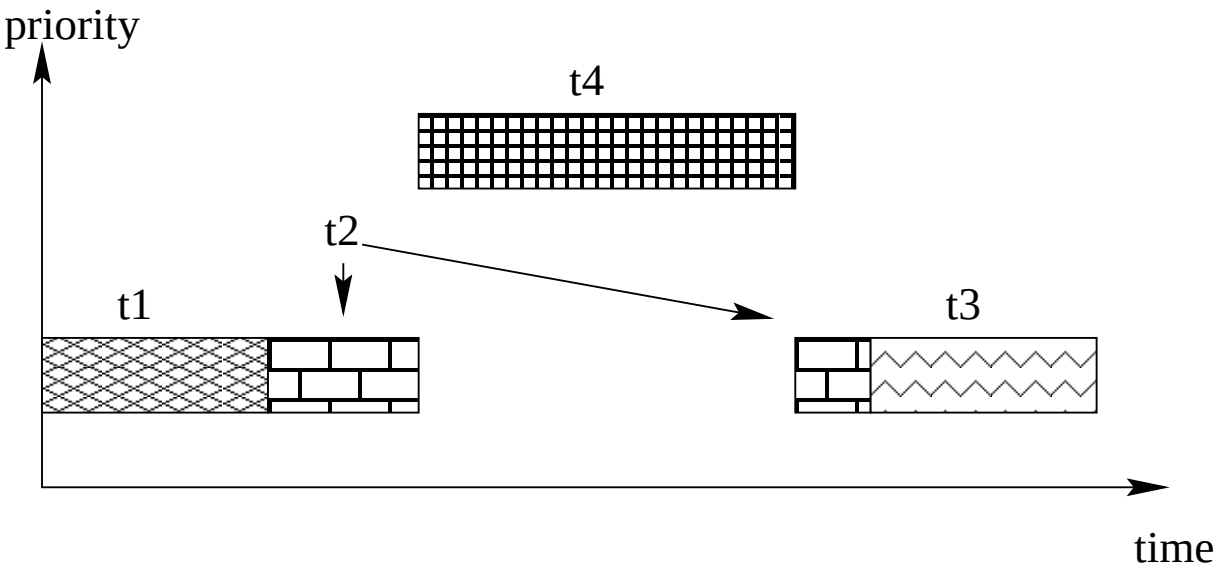
1. FIFO: First In First Out Scheduling
2. Round Robin: time-slicing

First-In First-Out (FIFO) Scheduling

Applies to same PRIORITY, READY tasks

— first-in, first out queue

— Example: t_1,t_2, t_3 have equal priorities



Drawback of FIFO:

A task may hold the CPU for a long time

$\Longrightarrow$ Ideal for:

— tasks which don't take long times

— have to run to completion

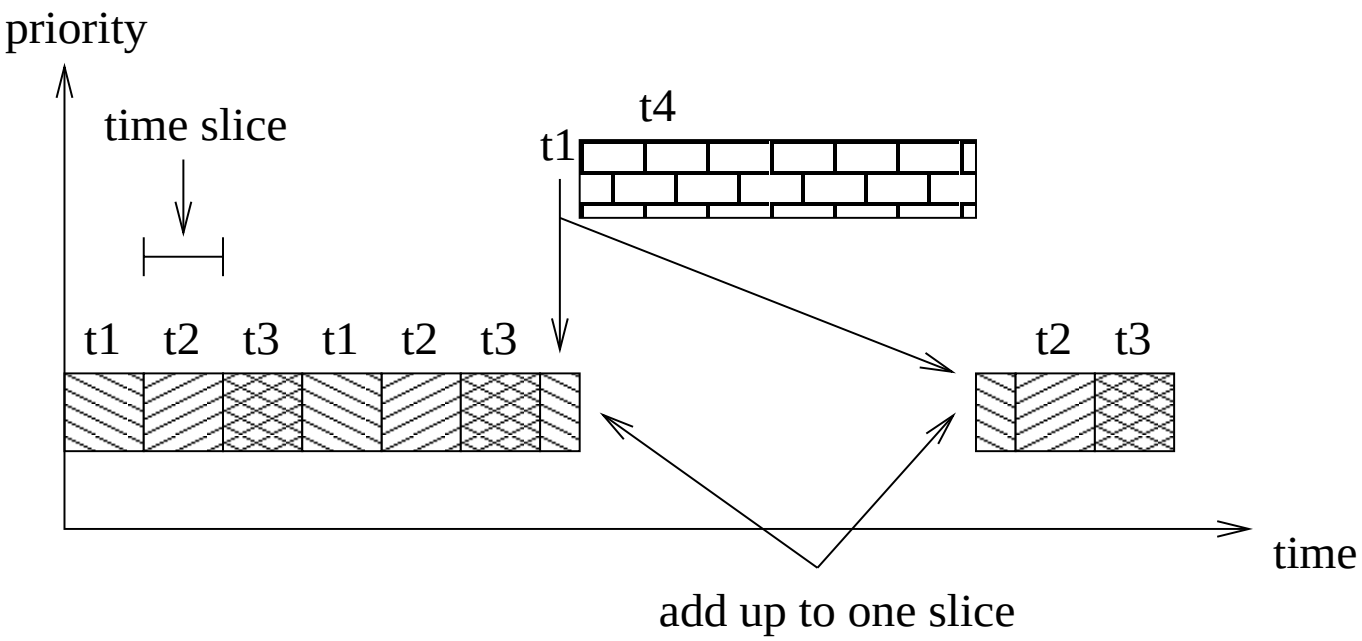
Round Robin (Time-slicing)

Each task executes for a slice of time in each round of the scheduling until:

— it finishes its time slice

— gives up the CPU $\rightarrow$ blocking or completion

— Previous example



OS provides tools for changing the time slice:
e.g., kernelTimeSlice(10) sets the time slice to 10 ticks