Week12– Real-Time Scheduling

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

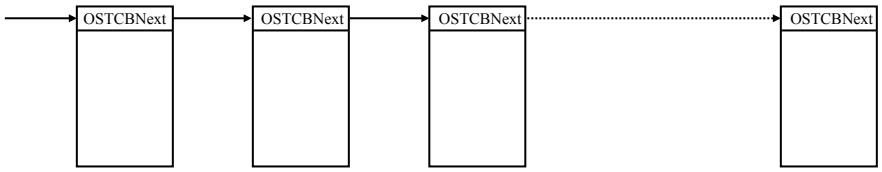
Electrical & Computer ENGINEERING

Lab 4 Requirements

- Build a real-time operating system called Gravelv2
- Part I
  - Port your libc, swi and irq installation from lab 3
  - Create, schedule tasks according to rate monotonic policy
  - Create, manage mutexes
  - Schedule tasks according to a fixed pre-assigned priority
- Part II
  - Utilization bound (UB) test – to be covered in "Advanced Real-Time" lecture
  - Highest locker priority protocol – to be covered in "Advanced Real-Time" lecture

Run queues

- Gravelv2 maintains the list of currently runnable tasks on run queues
- Typically there will be a run queue consisting of tasks that are ready to run
  - On a context switch, the scheduler can go through this run queue and look for the highest priority task that is ready to run
- Maintain run queue as a linked list of TCBs of all tasks in your system



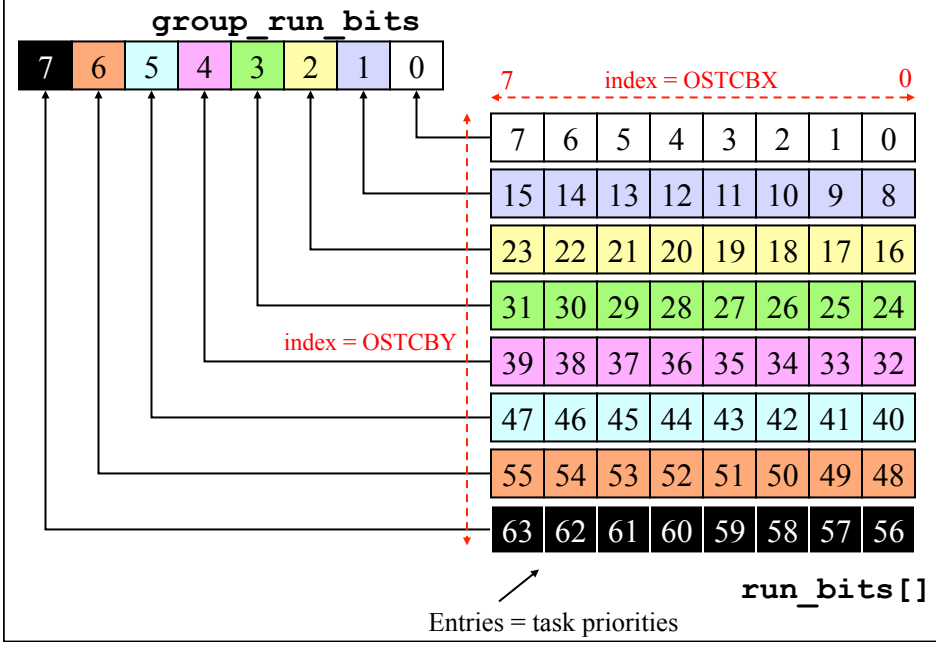
- How do you find the highest priority task in this list?
- Disadvantages?

Runqueue in Gravelv2

Where is the runqueue maintained in Gravelv2?

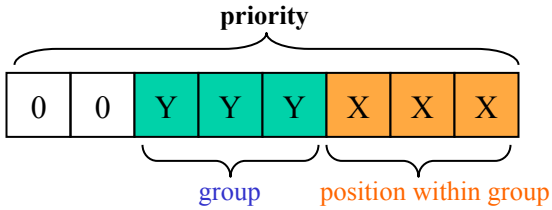
Using the array `run_bits[8]` and variable `group_run_bits` in `run_queue.c`

Ready Queue



Task Groups

- Gravelv2 divides its 64 priority levels into 8 groups
  - How do you determine a task's group?
  - **priority >> 3**
  - This value is stored in the **OSTCBY** variable
    - Task priorities 0, 1, 2, ….., 7 ⇨ task group = 0
    - Task priorities 8, 9, 10, ….., 15 ⇨ task group = 1
  - i.e., **OSTCBY should** contain the 3 MSBs of the task's priority number
- What is the task's position within the group?
  - **priority & 0x07**
  - This value is stored in the **OSTCBX** field of the task's OS_TCB
  - **OSTCBX** contains the 3 LSBs of the task's priority number



priority = OSTCBY << 3 + OSTCBX

Ready List

- Gravelv2 maintains two variables `group_run_bits` and `run_list[]` which are used to store the list of tasks ready to run
- When a task is created or is made ready to run, the appropriate bits of `group_run_bits` and `run_list[]` are set to indicate to the scheduler that the task is ready to run
- Example: Assume `group_run_bits` and `run_list[8]={0,0,…0}` and a task of priority `prio` = 17 is created, in order to let the scheduler know the presence of the task at `prio = 17`, `TaskCreate` must
  - Compute priority group (`OSTCBY` field) of the task
    `OSTCBY = (prio >> 3)  = 2`
  - Compute task's position in the priority group (`OSTCBX` field)
    `OSTCBX = (prio & 0x07) = 1`
  - Set bit number `OSTCBY` of `group_run_bits` to 1
  - Set bit number `OSTCBX` of `run_bits[OSTCBY]` equal to `1`

Example

- What will be the values of `group_run_bits` and `run_bits[]` when 3 tasks of priorities 16, 17 and 25 are created?
- Assume that initially group_run_bits = 0, and run_bits = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

Finding the Highest-Priority Ready Task

- Suppose that you want to find the highest-priority task (let's denote its priority by `prio`) that is ready to run
- Can be done quickly (without scanning the entire linked list of created tasks) through `group_bits` and `run_bits[]`
  - `Algorithm:`
    - Find the least significant bit set in `group_bits` (call it `y`)
    - Find the least significant bit set in `run_bits[y]` (say `x`)
    - Highest priority task ready to run is `(y<<3)+x`
- Optimization – Since you will be doing this often, to make this process faster, there is a special table called `prio_unmap_table[]` with 256 entries that helps you do the above

```
y   = prio_unmap_table[group_bits];
x   = prio_unmap_table[run_bits[y]]
prio = (y << 3) + x
```

- You are likely to need this kind of code in `run_queue()`

Finding the Highest-Priority Ready Task (contd.)

- Assume `group_bits=12` and `run_bits={0,0,3,2,0,0,0,0}`
- How do you determine the highest priority task to run

```
y   =  UnMapTbl[group_bits]
x   =
UnMapTbl[run_bits[y]]
prio = (y << 3) + x

Once prio is known, you
can get
to the TCB of this task by
accessing system_tcb[prio]
```

```
UnMapTbl[] = {
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

- Question: Do you need to do anything to `group_run_bits` and `run_bits[]` when suspending a task?

Summary

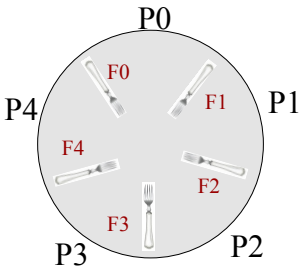
- Lab 4 overall summary
- Please follow course mailing list for more details
  - You may have several questions
  - Please use mailing list rather than sending individual requests
  - You may be helping others with similar problem
- Aim to finish task management first – you should try and use what we've covered today to do this as soon as possible
  - Lab is even more challenging than Lab 3
  - Very useful to understand embedded OS concepts if you have never done a course on OS

Dining Philosophers

- *n* philosophers *P0, P1, ..., Pn-1*
- To the left of each philosopher is a fork, and in the center is a bowl of spaghetti
- A philosopher was expected to spend most of his time thinking; but when he felt hungry, he needed to pick up both the left fork and right fork to eat the spaghetti
- When he was done with his meal, he puts down both sticks and continues thinking
- A fork can be used by only one philosopher at a time
  - If a neighboring philosopher wants it, he has to wait until the fork is available again

Putting All the Pieces Together

```
sem_t mayEat[5]; // how should we initialize this?
sem_t mutex; // how should we initialize this?
int state[5] = {THINKING};
take_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = HUNGRY;
    testSafetyAndLiveness(i); // check if I can run
    signal(&mutex); // exit critical section
    wait(&mayEat[i]);
}
release_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = THINKING;
    testSafetyAndLiveness (i+1 %5); // check if my neighbor can run now
    testSafetyAndLiveness(i+4 %5);
    signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
    if (state[i]==HUNGRY && state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
          state[i] = EATING;
          signal(&mayEat[i]);
    }
}
```



- What about starvation?

Necessary Conditions for a Deadlock

- Formally: (***all*** of these conditions ***must*** hold for deadlock to occur)
  - **Mutual exclusion**: some resource is non-sharable, (eg, a printer)
  - **Hold and wait**: there must exist a process holding for a resource (eg, a printer) and waiting for another resource (eg, a plotter)
  - **No preemption**: the system will not preempt the resources in contention
  - **Circular wait**: there must exist a circular chain of processes.
    - One is holding a resource and waiting for the next process

Deadlock Handling

- **Prevention**: structure the system in such a way as to avoid deadlocks (i.e., in a way to avoid one of the conditions above).
  - This is done in the design phase: design a system and ensure that there is no deadlock in the system
- **Avoidance**: does not make deadlock impossible (as in prevention)
  - Instead, it rejects requests that cause deadlocks by examining the requests before granting the resources. If there will be a deadlock, reject the request
- **Detection and Recovery**: after the deadlock has been detected, break one of the 4 conditions above. The mechanisms of deadlock detection and deadlock recovery are very much tied to each other

Deadlock Prevention

- Ensure one of the four necessary condition is never satisfied
  - Allow all resources to be shared (so prevent mutual exclusion)
    - Some resources are inherently non-sharable
  - Don't allow hold and wait i.e. force processes to either acquire all the needed resources or to release the acquired (currently held) resources if one or more requests are not granted
    - Starvation of processes that need many resources
  - Allow preemption of resources
  - Enforce total ordering in resource acquisition process and require each process to request resources in an increasing order
    - Example: Printer order=3, Tape drive order = 7
      - If process 1 needs printer and tape drive in any order it must request printer first and then tape drive

Can cause deadlock

```
P1::        P2::
P(s1);      P(s2);
P(s2);      P(s1);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

Prevents deadlock

```
P1::        P2::
P(s1);      P(s1);
P(s2);      P(s2);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

Deadlock Avoidance

- All processes need to declare in advance all the resources that they will need
- Given all the resources all the processes will need, the system can decide for each resource request whether or not a process should wait for the request to be granted
- **Banker's Algorithm**
- Based on how a bank would work with several clients that borrow and invest money
  - It differentiates between **safe** state (resources can be allocated to the processes in such a way that there will be no deadlock) and **unsafe states** (the opposite)
  - The OS will refuse requests for resources from processes if the request will take the system to an unsafe state.
  - If the system is always in a safe state, there will never be a deadlock.
  - When a request arrives, the Banker's algorithm checks whether there will be enough resources (money) to cover all the money to be returned to the clients in case they decide to withdraw.

Banker's Algorithm

- A, B, C, D are four bank customers
  - Each customer has been given a line of credit
  - Each customer may need all their credit units to do their job
    - Customers may not request their maximum credit units all at once
    - After using their credit, customers repay the loan
- Bank has 10 dollars even though 22 dollars are needed to satisfy each customer's request in the worst case
  - Customers must wait if their request cannot be satisfied
  - Deadlock occurs when all customers have to wait for each other
- Check whether the following bank states are safe

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

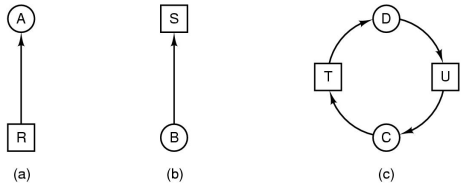
	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Source: Modern Operating Systems by A. Tanenbaum

Deadlock Detection

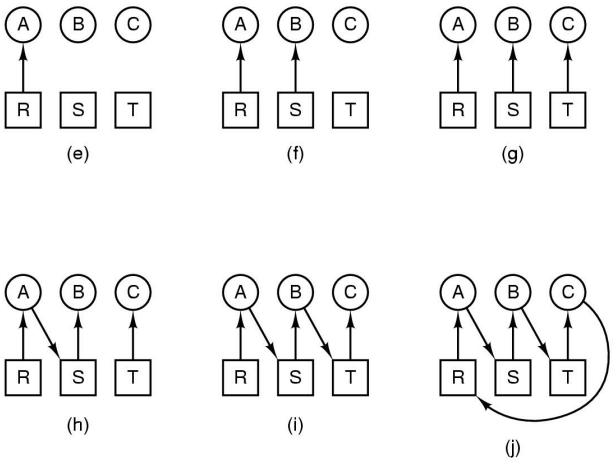
- Graph models for deadlock detection
- If each resource has a single instance in a system, a *directed* graph model can directly tell you if there is a deadlock in the system
  - Called resource allocation graph



  - resource R assigned to process A
  - process B is requesting/waiting for resource S
  - process C and D are in deadlock over resources T and U

Source: Modern Operating Systems by A. Tanenbaum

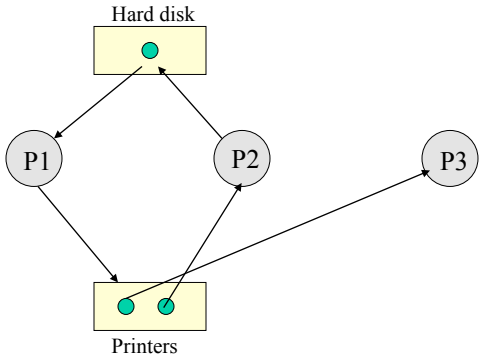
Deadlock detection with one resource of each type



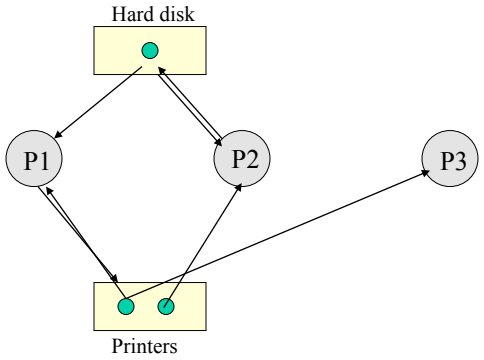
Source: Modern Operating Systems by A. Tanenbaum

Deadlock detection with multiple resources

- Assume a system with 2 printers and 1 hard disk – the resource allocation graph is shown below
  - Is there a deadlock?



Deadlock detection with multiple resources



With multiple resources of each type
If resource allocation graph does not have a cycle then the system is not in deadlock state.
If resource allocation graph has a cycle then system *may* or may not be in deadlock

Real-Time Basics

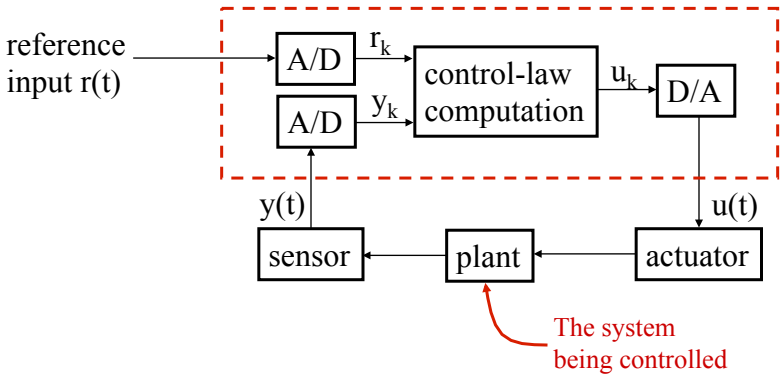
- Real-time systems: correctness of the system depends not only upon the logical correctness of the output but also upon the time at which the results are delivered
  - Subject to both temporal and logical constraints
  - Example: Not enough to say that "brakes were applied", a real-time system should say "brakes were applied at correct time"

- Examples of real-time systems
  - Transportation: automobiles, railways, subways, aircraft, ships, elevators
  - Traffic control: airspace, highways, shipping
  - Medical: pacemakers, radiation, patient monitoring
  - Military: command and control, missile defense, radar tracking
  - Manufacturing: automated plants

Example: Real-Time System

Many real-time systems are **control systems**

Example: A simple one-sensor, one-actuator control system



Simple Control System (cont'd)

Pseudo-code for this system:

```
set timer to interrupt periodically with period T;
at each timer interrupt do
    do analog-to-digital conversion to get y;
    compute control output u;
    output u and do digital-to-analog conversion;
end do
```

T is called the **sampling period**

Other Real-Time Applications

- **Multimedia Systems**
- Digital Set-top box: receives compressed (MPEG-2) video and audio data (bits) for each frame
- Frame rate is 30 frames/sec (i.e. 30 frames are displayed on the TV screen per second)
- Each frame has to be decoded within 1/30 = 33 ms to ensure that video data is available when the frame needs to be displayed
  - Audio data needs to be processed too, and decryption (if any) needs to be done during this time frame

What Makes a Real-Time Program Different?

- What does real-time mean?
  - A real-time service is one that is required to be delivered within time intervals dictated by the environment
  - Temporal constraints are a part of the results' correctness criteria
- What makes a real-time system different?
  - Timing constraints
    - Must satisfy timing constraints involving relative and absolute times
      - Example: a deadline is a limit on the amount of time for completing an operation or a computation
  - Concurrency
    - Must deal with the natural concurrency of the physical world
      - Sensors can fire simultaneously
      - Environmental signals can arrive at the same time

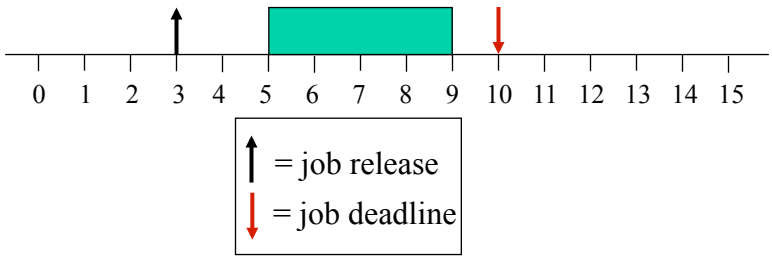
"Real-Time" Does Not Mean "Fast"

- "Real-time" does not mean "fast"
  - Real-time = meeting timing constraints
  - Fast = doing something quickly
- Real-time systems can be slow
- Fast systems can be non-real-time

Real Time Terminology

- **Job:** Instance of a task
- **Release time of a job:** The time instant the job becomes ready to execute
- **Absolute Deadline of a job:** The time instant by which the job must complete execution
- **Relative deadline of a job:** "Deadline - Release time"
- **Response time of a job:** "Completion time - Release time"

Example



- Job is released at time 3.
- Its (absolute) deadline is at time 10.
- Its relative deadline is 7.
- Its response time is 6.

Hard Real-Time Systems

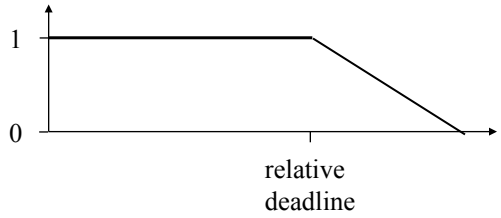
- A **hard deadline** *must* be met
  - If *any* hard deadline is *ever* missed, then the system is **incorrect**
  - Requires a means for **validating** that deadlines are met
- **Hard real-time system:** A real-time system in which at least one deadline is hard
  - We mostly consider hard real-time systems in this course
- **Examples:** Automotive braking systems, flight control, …

Soft Real-Time Systems

- A **soft deadline** may *occasionally* be missed
  - **Question:** Define "occasionally"?
- **Soft real-time system:** A real-time system in which all deadlines are soft
- **Examples:** Multimedia Systems

Defining "Occasionally"

- **One Approach:** Use probabilistic requirements
  - For example, 99% of deadlines will be met
- **Another Approach:** Define a "usefulness" function for each job:



Kinds of Real-Time Tasks

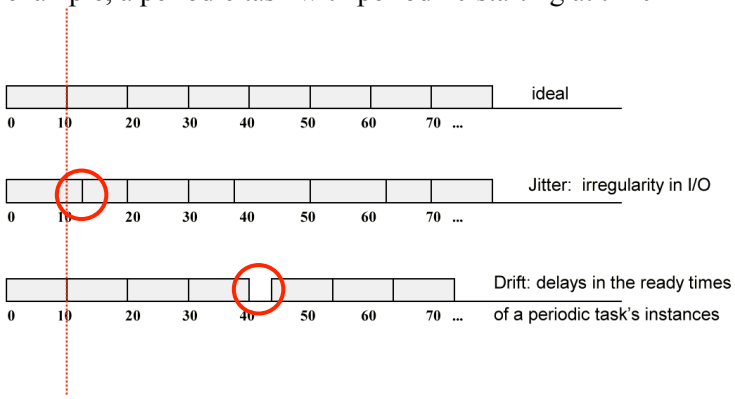
- Periodic processes/tasks
  - Time-driven
  - Activated on a regular basis between fixed time intervals
  - Specified by $\{C_i, T_i, D_i\}$
    - C_i = worst-case compute time (execution time) for task τ_i
    - T_i = period of task τ_i
    - D_i = deadline (relative) for task τ_i
  - Example: periodic monitoring or sampling of sensors
- Sporadic processes/tasks
  - Event-driven
  - Activated by an external entity or an environmental change
  - Example: Events such as faults
- Aperiodic processes/tasks
  - Event-driven
  - Multiple simultaneously or closely arriving events
  - For bursty events or bursty actions

Why Are Deadlines Missed?

- For a given task, consider
  - **Preemption**: time waiting for higher priority tasks
  - **Execution**: time to do its own work
  - **Blocking**: time delayed by lower priority tasks
- The task is schedulable if the sum of its preemption, execution, and blocking is less than its deadline.
- **Focus**: identify the biggest hits among the three and reduce, as needed, for schedulability

Drift and Jitter

- A periodic task should repeat regularly according to a given period.
  - For example, a periodic task with period 10 starting at t = 0

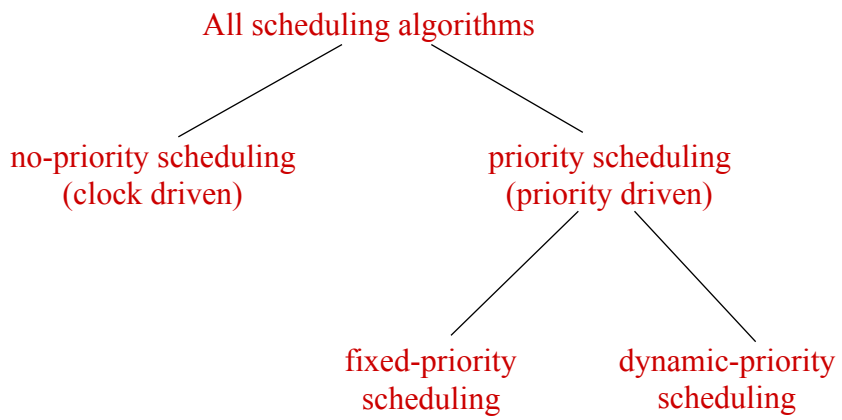


Scheduling

- Scheduler = resource allocator that affects the timing of real-time services
- Offline scheduler = does allocation at design time
- Online scheduler = does allocation at run-time
- Schedulers need to know dependencies and worst-case behavior

- Worst-case execution time (WCET) is often required to be known in real-time systems
  - How do you determine worst-case behavior?

Classification of Scheduling Algorithms



Scheduling Algorithms (No Priority)

- Non-priority-based
  - All tasks are created equal
  - There is no way to indicate which tasks are more "important" (more critical) than others
- First-In-First-Out (FIFO) or First-Come-First-Served (FCFS)
  - Ready tasks are inserted into a list
  - Tasks are dispatched from the list in their order of entry in the list
  - No preemption, i.e., a task runs to completion before the next task runs
  - No consideration of task priorities
- Round-Robin Preemptive
  - Ready tasks are dispatched in turn
  - Each task given its fair share of fixed execution time
  - Preemption of running task at the end of the fixed interval
  - No consideration of task priorities

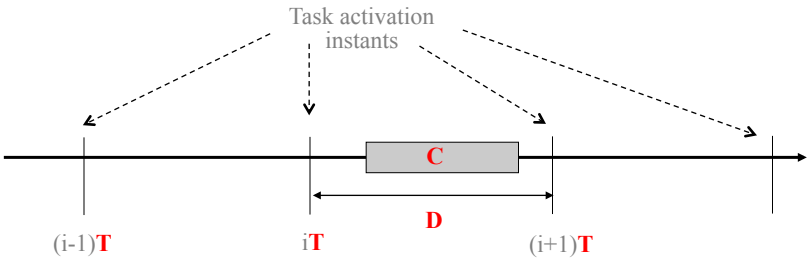
Scheduling Algorithms (Fixed-Priority)

- Fixed-priority based
  - All tasks are not created equal
  - Some tasks have more importance (higher priority) than others
  - Once a task's priority is assigned, it cannot change during run-time
- Rate Monotonic Scheduling (RMS)
  - Shorter the period, the higher the priority of the task
  - Assigns fixed priorities in reverse order of period length
  - Tasks requiring frequent attention have higher priority and get scheduled earlier
- Least Compute Time (LCT)
  - Shorter the computation time, the higher the priority of the task
  - Assigns fixed priorities in reverse order of computation length
  - Tasks finishing quickly have higher priority and get scheduled earlier

Scheduling Algorithms (Dynamic-Priority)

- Dynamic-priority based
  - All tasks are not created equal
  - Some tasks have more importance (higher priority) than others
  - Task's priority (and thus, the schedule) may change during execution
- Shortest Completion Time (SCT)
  - Ready task with the smallest remaining compute time gets scheduled first
- Earliest Deadline First (EDF)
  - Ready task with the earliest future deadline gets scheduled first
- Least Slack Time (LST)
  - Ready task with the smallest amount of free/slack time within the cycle gets scheduled first

Representation of a Real-Time Task



- Real-time periodic task represented as (C, T, D)
  - C = worst-case computation/execution time of process/task P
  - T = period or cycle time (how often the process/task P is activated)
  - D = deadline for completing execution of process/task P
- Constraints
  - $C <= D <= T$

Periodic Task: {C, T}

- Periodic task
  - Initiated at fixed intervals
  - Must finish before the relative deadline of the task
- Specifying a task $\{C_i , T_i\}$
  - C_i = worst-case compute time (execution time) for task τ_i
  - T_i = period of task τ_i

We assume that the relative deadline D_i of a task is the same as the period T_i

- Individual task's CPU utilization: $U_i = \frac{C_i}{T_i}$
- Total CPU utilization for a set of tasks

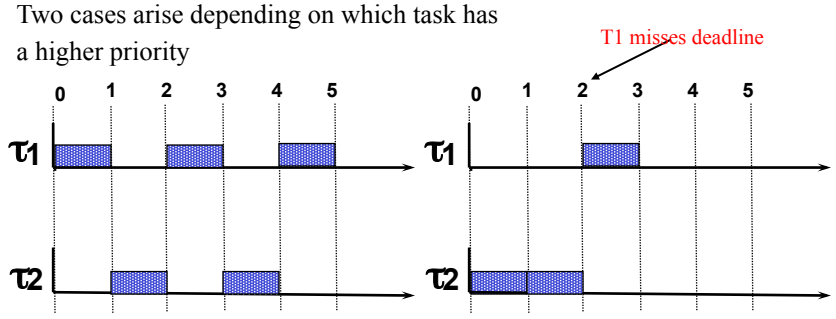
$$U = U_1 + U_2 + ... + U_n$$

- Assumption: Fixed priority preemptive scheduler
- Problem: Given $\{C_i , T_i\}$ for a set of tasks determine
  - **Whether the tasks are schedulable?**
  - **How to assign priorities to the tasks so that the tasks are schedulable?**

Task Schedulability – Priority Assignment Matters

- A set of tasks is schedulable if all tasks are guaranteed to meet their deadlines
- Why will a set of tasks be not schedulable?
  - Priority assignment
  - The values of C and Ts for the tasks (their utilization factor)

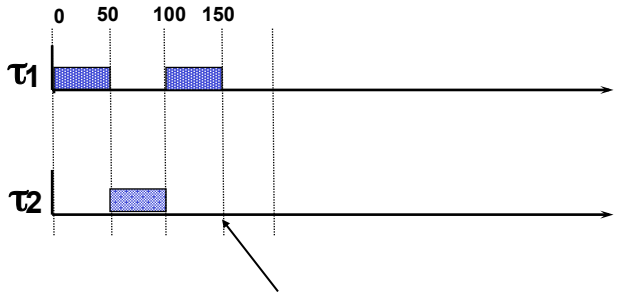
Example – T1 (1,2) & T2 (2,5) are two tasks.
Two cases arise depending on which task has a higher priority



T1 has higher priority **T2 has higher priority**

Task Schedulability – CPU Utilization Matters

Consider 2 periodic tasks
{C , T} for the first task (50,100)
{C , T} for the second task (60,150)
Assume both tasks arrive simultaneously at t=0
Overall utilization factor for these tasks = 0.9



T2 misses deadline!