به نام خدا

دانشکده مهندسی برق

تمرین ۲

درس سیستم‌های نهفته بی درنگ

محمدامین حاجی خداوردیان

۹۷۱۰۱۵۱۸

استاد:دکتر غلامپور

نیمسال دوم ۱۴۰۰-۱۴۰۱

۱ - پیاده سازی Rate – Monotonic

۱-۱ الگوریتم پیاده سازی

برای پیاده سازی این الگوریتم ابتدا تابعی تحت عنوان گرفتن ورودی از کاربر نوشته شده است که تعداد Taskها به همراه دوره زمانی (t_i) و بدترین هزینه پردازش(WCET) (c_i) در آن دریافت و ذخیره می‌شود. این بخش کد در زیر آورده شده است:

```
def Read_Data():

    """
    Reading the details of the tasks to be scheduled from the user as
    Number of tasks N:
    Period of task P:
    Worst case excecution time C:
    """
    global N
    global HP
    global Tasks

    N = int(input("\n \t\tEnter number of Tasks:"))

    for i in range(N):
        Tasks[i] = {}
        print("\n Enter Period of task T",i,":")
        P = input()
        Tasks[i]["Period"] = int(P)
        print("Enter the WCET of task C",i,":")
        C = input()
        Tasks[i]["WCET"] = int(C)
        Complete_Period.append(0)
```

تابع اصلی که عملیات اصلی بر دوش آن است و نام این تابع simulation است به این صورت است که با توجه به بازه زمانی که برای آن با استفاده از الگوریتم fixed Priority در نظر می‌گیریم برنامه، زمان را به یک واحد زمان بازه بندی می‌کند و در ابتدای هر واحد زمانی اولویت Task با توجه به پایان یافتن Task و دوره زمانی آن توسط تابع PriorityCalc محاسبه می‌شود که به صورت جداگانه این تابع توضیح داده می‌شود. سپس با توجه به اینکه در این الگوریتم ما فرض بر این داشته ایم که T = D بوده است چک می‌شود که آیا انجام Task از ددلاین خود گذشته است یا خیر. متغیری تحت عنوان RealTime_task وجود دارد که به عنوان ورودی تابع PriorityCalc است که در آن همه ویژگی های Task وجود دارد اما با گذشت هر واحد زمانی اگر پردازنده کاری از آن را پردازش کرده باشد از مقدار پردازش آن کم می‌شود. در انتهای کد نیز اگر Taskای به طور کامل

انجام شده باشد و دوره تناوب آن به سر رسیده باشد مجدد مقدار پردازش آن ریست می‌شود تا آمدن Task جدید به این نحو پیاده سازی شده باشد. کد این تابع در شکل زیر آورده شده است:

```
def Simulation(hp):

    """
    The real time schedulng based on Rate Monotonic scheduling is simulated here.
    """
    # Real time scheduling are carried out in RealTime_task
    global RealTime_task
    RealTime_task = copy.deepcopy(Tasks)

    # main loop for simulator
    for t in range(hp):
        # Determine the priority of the given tasks
        Priority = PriorityCalc(RealTime_task)

        if (Priority != -1):
            # Update WCET after each clock cycle
            RealTime_task[Priority]["WCET"] -= 1
            # For plotting the results
            if t < (Complete_Period[Priority] + 1 )*Tasks[Priority]["Period"]:
                y_axis.append("TASK%d"%(Priority+1)+" (" + "C =%d"%Tasks[Priority]["WCET"] + ")")
                from_x.append(t)
                to_x.append(t+1)
            else:
                y_axis_Miss.append("TASK%d"%(Priority+1)+" (" + "C =%d"%Tasks[Priority]["WCET"] + ")")
                from_x_Miss.append(t)
                to_x_Miss.append(t+1)

        # Update Period after each clock cycle
        for i in RealTime_task.keys():
            RealTime_task[i]["Period"] -= 1
            if (RealTime_task[i]["Period"] == 0 ):
                if (RealTime_task[i]["WCET"] == 0):
                    Complete_Period[i] += 1
                    RealTime_task[i] = copy.deepcopy(Tasks[i])
```

در ادامه به توضیح تابع PriorityCalc می‌پردازیم. در این تابع متغیر RealTime_task را دریافت می‌کند که در آن میزان پردازش Taskها تا زمانی که گذشته است ذخیره شده است. سپس اگر یک Task پایان نیافته باشد با توجه به دوره تناوب آن اولویت بندی می‌شود. کد این بخش در تصویر زیر مشاهده می‌شود:

```
def PriorityCalc(RealTime_task):

    """
    Estimates the priority of tasks at each real time period during scheduling
    """
    HP = 10000
    TempPeriod = HP
    P = -1    #Returns -1 for idle tasks
    for i in RealTime_task.keys():
        if (RealTime_task[i]["WCET"] != 0):
            if (TempPeriod > Tasks[i]["Period"]):
                TempPeriod = Tasks[i]["Period"] #Checks the priority of each task based on period
                P = i
    return P
```

در بخش تابع Simulation به الگوریتم fixed Priority اشاره کردیم که این الگوریتم در تابعی با نام MinimumPeriod پیاده سازی شده است. با توجه به گفته‌های درس رابطه‌ی اصلی این الگوریتم به صورت زیر است:

$R_i^{(k)}$: worst-case response time for τ_i at step k.

$$R_i^{(0)} = C_i + \sum_{j=1}^{i-1} C_j$$

$$R_i^{(k)} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil C_j$$

کد پیاده سازی شده نیز چیزی فراتر از این قطعه کد نیست و تنها برای Task با بدترین دوره تناوب که کمترین اولویت را دارد نوشته شده است. ورودی این تابع مقدار K است که تعداد Iteration هایی که قرار است برای الگوریتم بالا زده شود را مشخص می‌کند که به صورت پیش فرض برابر ۲۰ در نظر گرفته شده است و اگر در دو Iteration متوالی مقدار R ثابت بماند این مقدار برگردانده می‌شود. کد این بخش به صورت زیر است:

```
def MinimumPeriod(K):
    #Fixed Priority Algorithm
    R = []
    TempPeriod = 0
    P = -1 #Returns -1 for idle tasks
    for i in Tasks.keys():
        if Tasks[i]["Period"] > TempPeriod:
            TempPeriod = Tasks[i]["Period"]
            P = i
    for i in range(K):
        if i==0:
            Sum = 0
            for j in range(N-1):
                Sum = Tasks[j]["WCET"] + Sum
            R.append(Tasks[P]["WCET"] + Sum)
        else:
            Sum = 0
            for j in range(N-1):
                Sum = ceil(R[i-1]/Tasks[j]["Period"])*Tasks[j]["WCET"] + Sum
            R.append(Tasks[P]["WCET"] + Sum)

        if R[i] == R[i-1] and i != 0:
            return R[i]

    return R[K]
```

علاوه بر بخش بالا یک تابع تحت عنوان UtilizationCalc وجود دارد که میزان Utilization را حساب می-کند که در انتها با استفاده از شروط گفته شده به قابل برنامه ریزی بودن یا نبودن یک سری مجموعه از Task ها پی ببریم. این تابع به صورت زیر است:

```
def UtilizationCalc():

    """
    For Finding system Utilization and Check Bounds
    """
    U = 0
    for i in range(N):
        U = U + Tasks[i]["WCET"]/Tasks[i]["Period"]

    return U
```

تابع آخری که استفاده شده است برای رسم است که توضیح خاصی ندارد و کد آن در زیر آورده شده است:

```
def DrawGraph():
    """
    The scheduled results are displayed in the form of a
    gantt chart for the user to get better understanding
    """

    fig = plt.figure()
    ax = fig.add_subplot(111)
    # the data is plotted from_x to to_x along y_axis
    ax = plt.hlines(y_axis, from_x, to_x, linewidth=20, color = 'green')
    if from_x_Miss != []:
        ax = plt.hlines(y_axis_Miss, from_x_Miss, to_x_Miss, linewidth=20, color = 'red')
    plt.title('Rate Monotonic scheduling')
    plt.grid(True)
    plt.xlabel("Real-Time clock")
    plt.ylabel("HIGH<-----------------Priority--------------------->LOW")
    plt.xticks(np.arange(min(from_x), max(to_x)+1, 1.0))
    plt.show()
```

۱-۲ نحوه به کارگیری و استفاده از کد

در این بخش با اجرای کد در کنسول از شما تعداد Task به همراه دوره تناوب و بدترین زمان پردازش خواسته می‌شود که پس از وارد کردن این اطلاعات دیاگرام زمانی رسم می‌شود. برای ایجاد تغییر در تعداد Iteration تابع MimimumPeriod که در بخش قبل توضیحات آن داده شده است کافی است که در انتهای کد ورودی تابع را مقدارش را تغییر دهیم:

```
143    Read_Data()
144    hp = MinimumPeriod(20)
145    print("Minimum Peri0d is:", hp)
146    Simulation(hp)
147    DrawGraph()
```

کافی است عدد ۲۰ را با عدد دلخواه خود عوض کنید.

همچنین اگر بخواهید بازه زمانی رسم شده را به دلخواه تغییر دهید کافی است به جای hp در تابع Simulation مقدار دلخواه خود را قرار دهید.

فایل شبیه سازی این بخش با نام RM.py موجود است و باید این فایل اجرا شود.

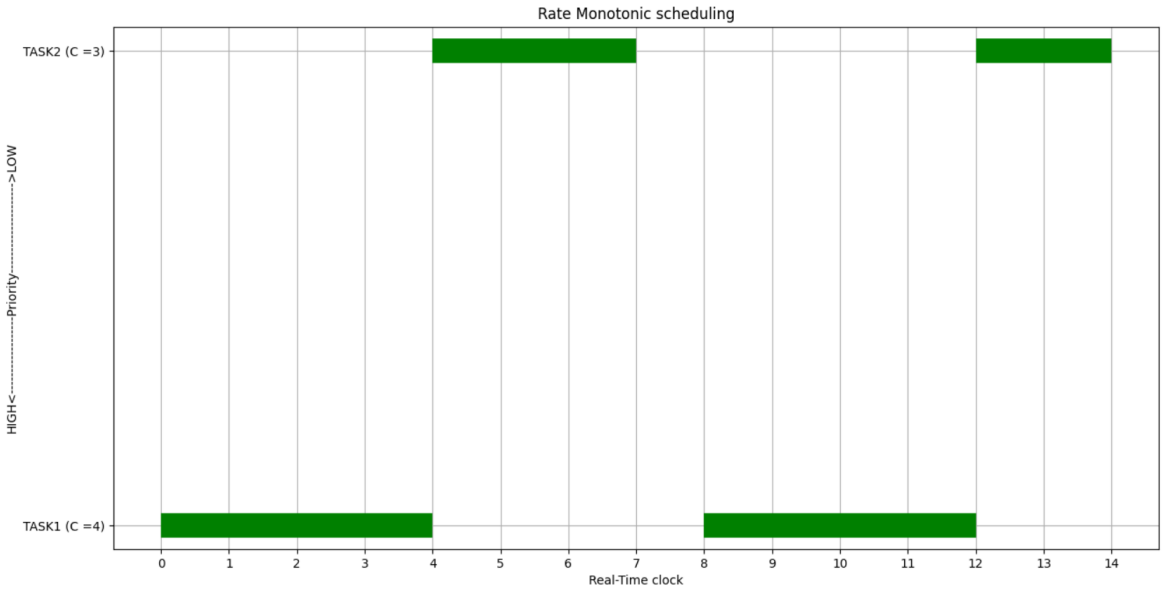
۱-۳ آزمون و مثال‌های متنوع

در این بخش ابتدا برای سنجیدن صحت کد به سراغ مثال هایی که در کلاس درس زده شد می‌رویم. اولین مثال ما مقادیر زیر را دارد:

مثال ۱:

T1 = 8 , C1 = 4　　T2 = 12 , C2 = 3

دیاگرام زمانی به شکل زیر خواهد بود:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
worst-case response time 6
We can Use RM for scheduling because Us < Ub
 Ub: 0.8284271247461903 Us: 0.75
```

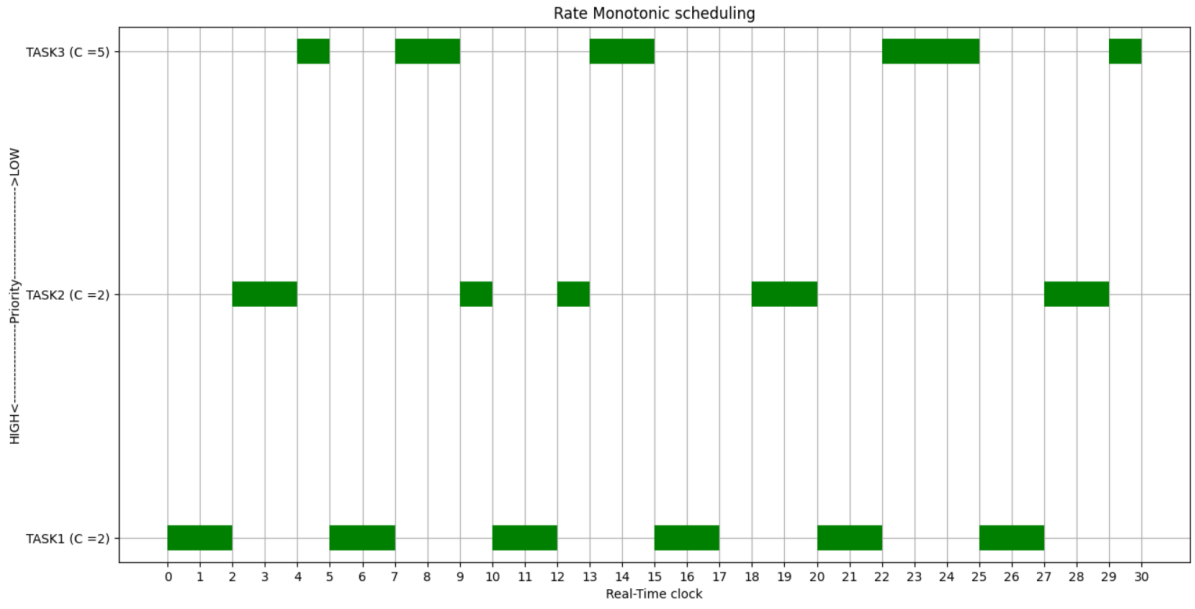
Task با اولویت بیشتر در پایین تابع قرار گرفته است و Task با اولویت کمتر در بالای تابع قرار می‌گیرد.

مثال دومی که در نظر گرفته شده است به صورت زیر است:

مثال ۲

T3 = 20, C3 = 5　　T2 = 9, C2 = 2　　T1 = 5, C1 = 2

دیاگرام زمانی مثال بالا در صفحه بعد آورده شده است. انتظار ما این است که بیشترین اولویت برای Task1 و کمترین اولویت برای Task3 باشد با توجه به نحوه تصمیم گیری الگوریتم Rate Monotonic که این قضیه در تصویر صفحه بعد مشخص است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

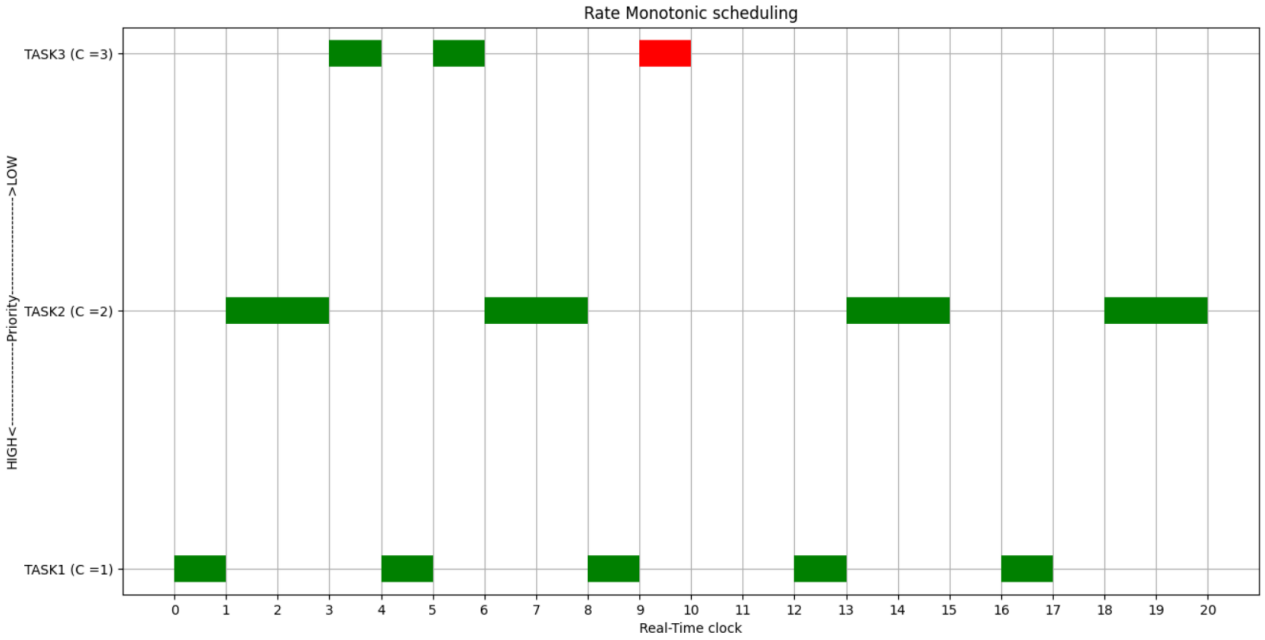
```
worst-case response time 15
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.7797631496846196 Us: 0.8722222222222222
```

برای مثال سوم به سراغ مثالی می‌رویم که در کلاس درس زده شد و در آن استفاده از الگوریتم RM باعث از دست دادن یک سری از عملیات‌ها می‌شود که در شبیه ساز ما این عملیات ها با رنگ قرمز مشخص خواهند شد.

مثال ۳

C3 = 3, T3 = 8　　C2 = 2, T2 = 6　　C1 = 1, T1 = 4

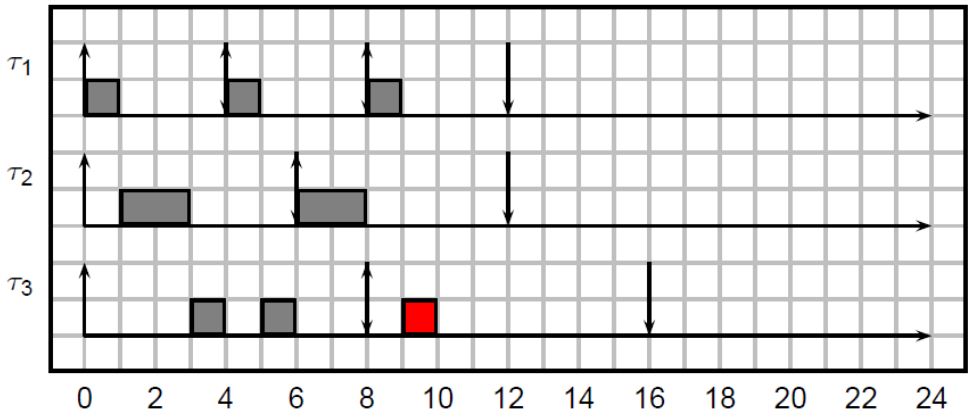
دیاگرام زمانی بالا به صورت زیر است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
worst-case response time 10
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.7797631496846196 Us: 0.9583333333333333
```

دقیقا مشابه مثال کلاس در بازه زمانی (9,10) یک عملیات از دست رفته خواهیم داشت که پس از آن دیگر زمان بندی ها معتبر نیست. تصویر زیر نتیجه موجود در جزوه است(تفاوت در شکل به علت این است که در شبیه ساز پیاده سازی شده توسط بنده Task با اولویت بالاتر در بخش پایین تری قرار می‌گیرد):

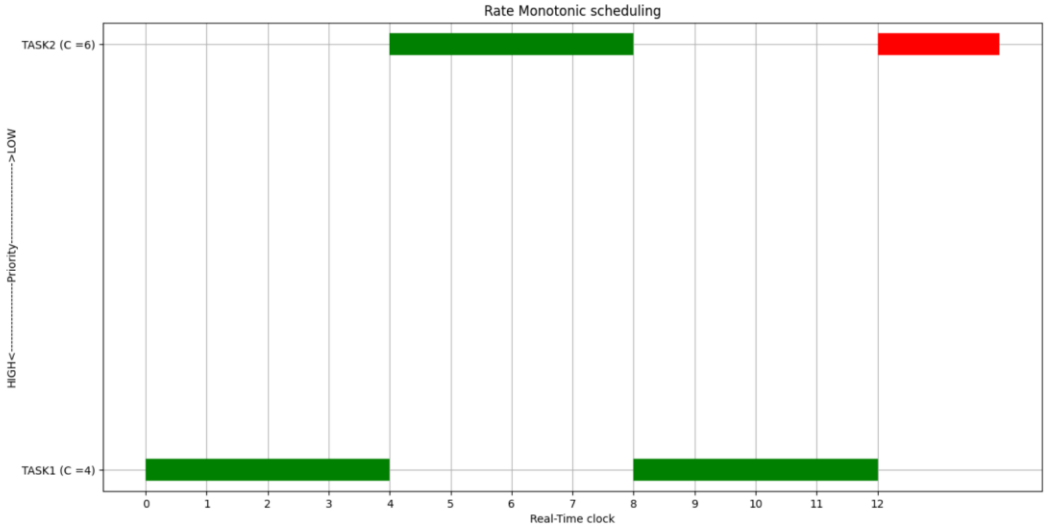


یک مثال دیگری که در کلاس بررسی شد و در آن الگوریتم RM جوابگو نیست به صورت زیر است:

مثال ۴:

T1 = 8, C1 = 4 T2 = 12 , C2 = 6

دیاگرام زمانی مثال بالا به صورت زیر است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
worst-case response time 14
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.8284271247461903 Us: 1.0
```

که نتیجه آن مشابه مثالی است که در کلاس آورده شده است.

۲ - پیاده سازی Earliest Deadline First

۲-۱ الگوریتم پیاده سازی

برای پیاده سازی این الگوریتم ابتدا تابعی تحت عنوان گرفتن ورودی از کاربر نوشته شده است که تعداد Taskها به همراه دوره زمانی (t_i) ، بدترین هزینه پردازش(WCET) (c_i) و ددلاین (D_i) در آن دریافت و ذخیره می‌شود. این بخش کد در زیر آورده شده است:

```
def Read_Data():

    """
    Reading the details of the tasks to be scheduled from the user as
    Number of tasks N:
    Period of task P:
    Worst case excecution time C:
    """
    global N
    global HP
    global Tasks

    N = int(input("\n \t\tEnter number of Tasks:"))
    # Storing data in a dictionary

    for i in range(N):
        Tasks[i] = {}
        print("\n Enter Period of task T",i,":")
        P = input()
        Tasks[i]["Period"] = int(P)
        print("Enter the WCET of task C",i,":")
        C = input()
        Tasks[i]["WCET"] = int(C)
        print("Enter the DeadLine of task D",i,":")
        D = input()
        Tasks[i]["DeadLine"] = int(D)
        Complete_Period.append(0)
```

بدنه اصلی این بخش نیز مشابه الگوریتم قبلی است و تفاوت چندانی در آن به وجود نیامده است و تنها تفاوت در بخشی است که آیا عملیاتی از دست می‌رود یا خیر که در این الگوریتم این بخش با توجه به ددلاین‌ها تصمیم گیری رخ می‌دهد. در هر واحد زمانی تابعی با عنوان PriorityCalc به ددلاین‌ها نگاه می‌کند و عملیاتی که نزدیک ترین ددلاین را دارد را به عنوان عملیاتی که باید انجام دهد انتخاب می‌کند. در تابع PriorityCalc ممکن است حالتی رخ دهد که یک Task که در حال انجام آن بوده‌ایم ددلاین آن با Task جدیدی که دریافت می‌کنیم یکسان باشد در این صورت تصمیم گیری ما به این صورت است که Task در حال اجرا را به پایان برسانیم و سپس به سراغ Task جدید برویم که این بخش و کل تابع PriorityCalc در صفحه بعد قطعه کد آن آمده است:

```python
def PriorityCalc(RealTime_task , PPriority):

    """
    Estimates the priority of tasks at each real time period during scheduling
    """
    HP = 10000
    TempDeadLine = HP
    P = -1    #Returns -1 for idle tasks
    for i in RealTime_task.keys():
        if (RealTime_task[i]["WCET"] != 0):
            if (TempDeadLine > RealTime_task[i]["DeadLine"]):
                TempDeadLine = RealTime_task[i]["DeadLine"] #Checks the priority of each task based on DeadLine
                P = i
            elif (TempDeadLine == RealTime_task[i]["DeadLine"] and PPriority != -1 and RealTime_task[PPriority]["WCET"] != 0 ):
                #Give priority to task that is undergoing
                P = PPriority

    return P
```

ورودی این تابع مشابه بخش قبل تابع RealTime_task را دارد اما با این تفاوت که یک ورودی جدید دیگر تحت عنوان PPriority وجود دارد که اولویت Task ای که در واحد زمانی قبل در حال اجرا بود به آن داده می‌شود که در صورتی که شرایط گفته شده در صفحه قبل برقرار شد بتواند Task در حال اجرا را ادامه دهد.

در انتهای تابع simulation به علت اضافه شدن ددلاین تغییراتی باید لحاظ شود که یکی از آن‌ها این است که در صورتی که دوره تناوب یک Task تمام شد ددلاین بعدی محاسبه و در متغیر RealTime_task لحاظ شود ولی اگر این Task تمام نشده بود فقط دوره تناوب آن بروزرسانی شود. که این بخش در زیر آورده شده است:

```python
def Simulation(hp):

    """
    The real time schedulng based on Rate Monotonic scheduling is simulated here.
    """
    # Real time scheduling are carried out in RealTime_task
    global RealTime_task
    RealTime_task = copy.deepcopy(Tasks)
    PPriority = -1
    # main loop for simulator

    for t in range(hp):

        # Determine the priority of the given tasks
        Priority = PriorityCalc(RealTime_task , PPriority)
        PPriority = Priority

        if (Priority != -1):
            # Update WCET after each clock cycle
            RealTime_task[Priority]["WCET"] -= 1
            # For plotting the results

            #print("T is:" , t , "  Deadline is:" , (Complete_Period[Priority] + 1 )*Tasks[Priority]["DeadLine"])
            if t < RealTime_task[Priority]["DeadLine"]:
                y_axis.append("TASK%d"%(Priority+1)+" (" + "C =%d"%Tasks[Priority]["WCET"] + ")")
                from_x.append(t)
                to_x.append(t+1)
            else:
                y_axis_Miss.append("TASK%d"%(Priority+1)+" (" + "C =%d"%Tasks[Priority]["WCET"] + ")")
                from_x_Miss.append(t)
                to_x_Miss.append(t+1)
```

```
# Update Period after each clock cycle
for i in RealTime_task.keys():
    RealTime_task[i]["Period"] -= 1
    if (RealTime_task[i]["Period"] == 0 ):
        if (RealTime_task[i]["WCET"] == 0):
            Complete_Period[i] += 1
            RealTime_task[i]["DeadLine"] = Tasks[i]["Period"] * (Complete_Period[i]) + Tasks[i]["DeadLine"]
            RealTime_task[i]["WCET"] = copy.deepcopy(Tasks[i]["WCET"])
            RealTime_task[i]["Period"] = copy.deepcopy(Tasks[i]["Period"])
        else:
            RealTime_task[i]["Period"] = copy.deepcopy(Tasks[i]["Period"])
```

تابع Utilization چون در این بخش T != D است تغییر می‌کند و نحوه محاسبه آن تفاوت می‌کند و به صورت زیر است:

```
def UtilizationCalc():

    """
    For Finding system Utilization and Check Bounds
    """
    U = 0
    for i in range(N):
        U = U + Tasks[i]["WCET"]/Tasks[i]["DeadLine"]

    return U
```

تابع رسم نیز با بخش قبل هیچ تفاوتی ندارد به همین دلیل از این بخش صرف نظر می‌کنیم.

۲-۲ نحوه به کارگیری و استفاده از کد

در این بخش با اجرای کد در کنسول از شما تعداد Task به همراه دوره تناوب و بدترین زمان پردازش خواسته می‌شود که پس از وارد کردن این اطلاعات دیاگرام زمانی رسم می‌شود.:

```
133    Read_Data()
134    hp = 30
135    Simulation(hp)
136    DrawGraph()
```

همچنین اگر بخواهید بازه زمانی رسم شده را به دلخواه تغییر دهید کافی است به جای hp در تابع Simulation مقدار دلخواه خود را قرار دهید.

فایل شبیه سازی این بخش با نام EDF_T_diff_D.py موجود است و باید این فایل اجرا شود.

۳-۲ آزمون و مثال‌های متنوع

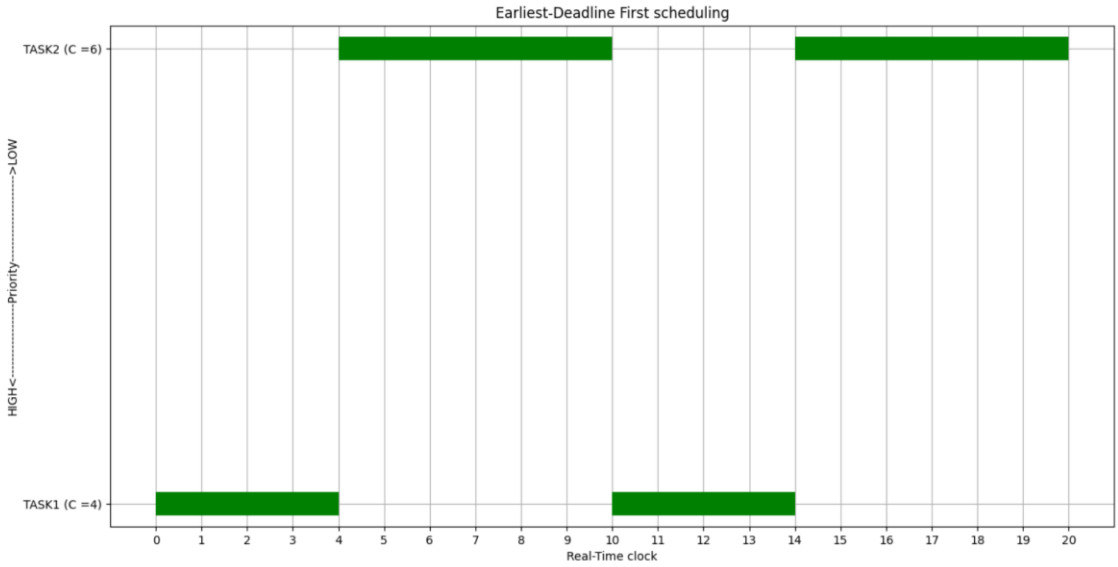
در این بخش ابتدا برای سنجیدن صحت کد به سراغ مثال هایی که در کلاس درس زده شد می‌رویم. اولین مثال ما مقادیر زیر را دارد:

مثال ۱

این مثال دقیقا مثال ۴ بخش قبل است که با توجه به دیاگرام زمانی دیدیم که الگوریتم RM توانایی زمان بندی آن را ندارد. این مثال مقدار Utilization آن برابر ۱ است و می‌دانیم توسط الگوریتم EDF به درستی زمان بندی باید بشود. دیاگرام زمانی این مثال در صفحه بعد آورده شده است:

T1 = 8, C1 = 4 T2 = 12 , C2 = 6

در ورودی ها باید دقت کنیم که مقدار ددلاین ها را در این مثال برابر با T قرار دهیم: (T = D)



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

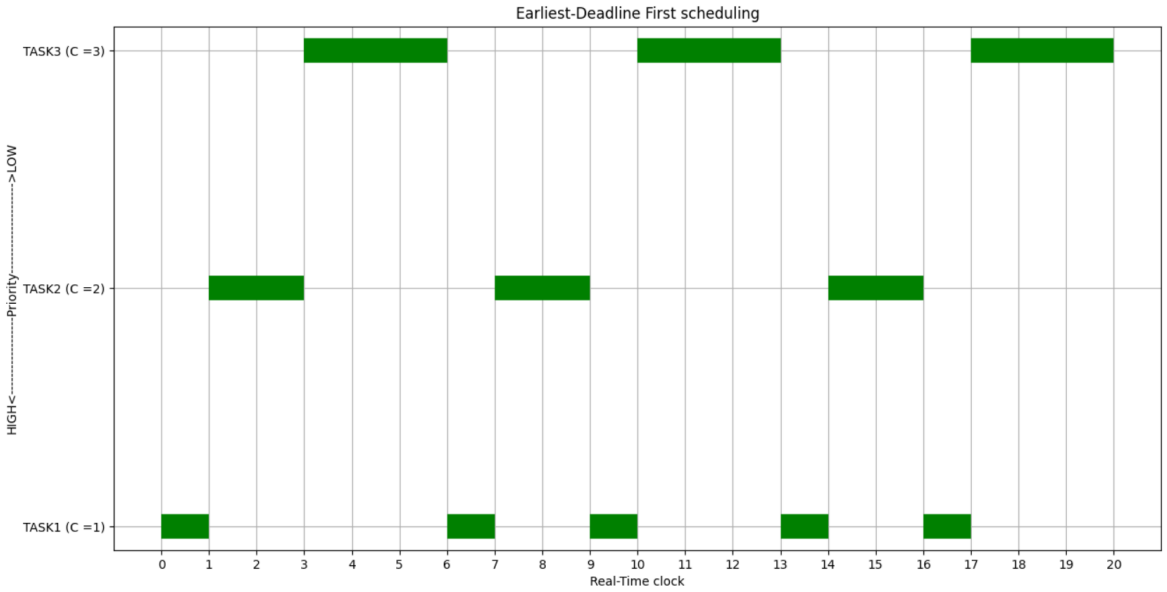
```
We can Use EDF for scheduling because Us <= 1
 Us: 1.0
```

همانطور که دیده می‌شود مشکلی که در الگوریتم RM وجود داشت در این مثال برطرف شده است.

مثال ۲

در این مثال هم به سراغ مثال ۳ بخش قبل می‌رویم که همانطور که دیده شد توسط RM به درستی زمان بندی رخ نمی‌دهد. در ورودی ها باید دقت کنیم که مقدار ددلاین ها را در این مثال برابر با T قرار دهیم: (T = D)

C3 = 3, T3 = 8　　C2 = 2, T2 = 6　　C1 = 1, T1 = 4



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

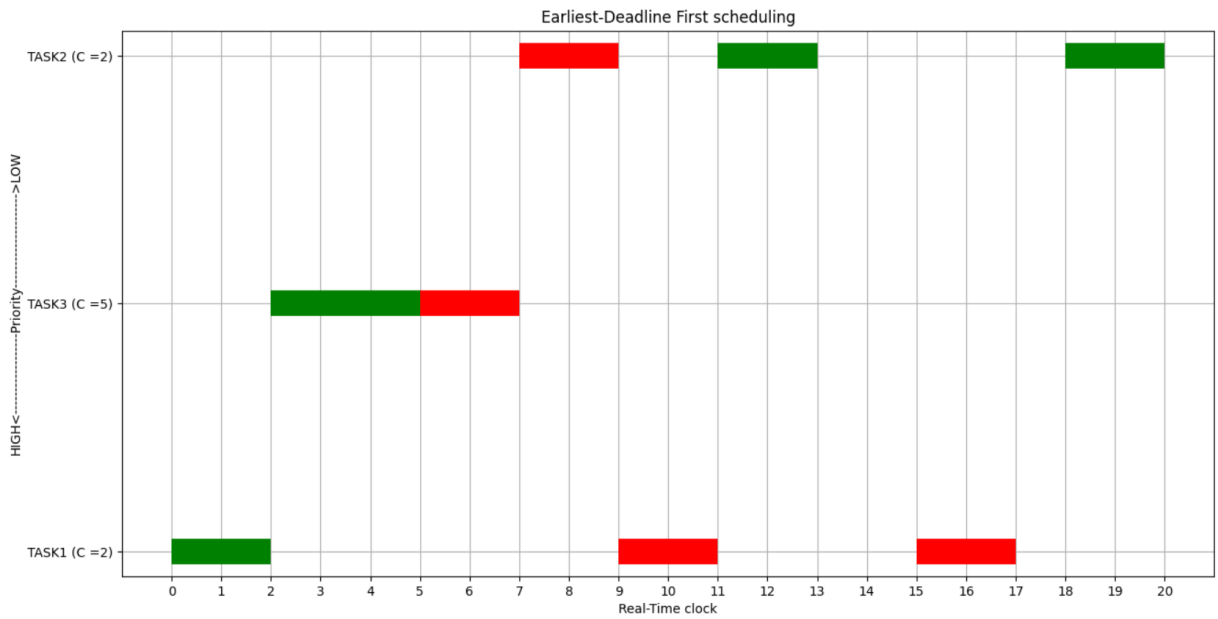
```
We can Use EDF for scheduling because Us <= 1
 Us: 0.9583333333333333
```

همانطور که در شکل دیده می‌شود عملیات‌ها به درستی زمان بندی شدند و مشابه نتیجه جزوات درس است.

مثال ۳

T3 = 20, C3 = 5, D3 = 5　　T2 = 9, C2 = 2, D2 = 6　　T1 = 5, C1 = 2, D1 = 3

دیاگرام زمانی برای مثال بالا به شکل صفحه بعد است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
We Can't use EDF for scheduling because Us > 1
 Us: 2.0
```

با توجه به ددلاین نزدیک Task 3 باید تا واحد زمانی ۵ این Task به پایان برسد ولی Task1 چون ددلاین نزدیک تری دارد زودتر باید انجام شود و از طرف دیگر باعث می‌شود که Task3 ددلاین خودش را از دست بدهد و این عملیات به همین گونه ادامه میابد و زمان بندی دیگر معتبر نیست.

مثال ۴

T3 = 20, C3 = 5, D3 = 15 T2 = 9, C2 = 2, D2 = 6 T1 = 5, C1 = 2, D1 = 3

همان مثال قبل ولی این بار مشکلی که پیش آمده بود در ددلاین Task 3 بود که آن را تغییر دادیم و افزایش دادیم تا فرصت انجام عملیات به Task داده شود. حال دیاگرام زمانی مثال بالا را رسم می‌کنیم که تصویر دیاگرام در صفحه بعد آورده شده است:

شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
We Can't use EDF for scheduling because Us > 1
 Us: 1.3333333333333333
```

در کلاس و جزوه نیز گفته شده است که اگر T != D باشد رابطه Utilization تنها تفاوت آن جایگزینی T و D است ولی این حالت بدبینانه است و همانطور که در مثال بالا دیده می‌شود با اینکه چک کردن مرز با روابط این را نشان می‌داد که قابلیت برنامه ریزی وجود ندارد ولی به درستی زمان بندی انجام شد

در این مثال به درستی زمان بندی انجام شده است و هیچ Taskای ددلاین خود را از دست نمی‌دهد.

۳ - پیاده سازی Deadline Monotonic

۳-۱ الگوریتم پیاده سازی

برای پیاده سازی این الگوریتم ابتدا تابعی تحت عنوان گرفتن ورودی از کاربر نوشته شده است که تعداد Taskها به همراه دوره زمانی (t_i) ، بدترین هزینه پردازش(WCET) (c_i) و ددلاین (D_i) در آن دریافت و ذخیره می‌شود. این بخش کد در زیر آورده شده است:

```
def Read_Data():

    """
    Reading the details of the tasks to be scheduled from the user as
    Number of tasks N:
    Period of task P:
    Worst case excecution time C:
    """
    global N
    global HP
    global Tasks

    N = int(input("\n \t\tEnter number of Tasks:"))

    for i in range(N):
        Tasks[i] = {}
        print("\n Enter Period of task T",i,":")
        P = input()
        Tasks[i]["Period"] = int(P)
        print("Enter the WCET of task C",i,":")
        C = input()
        Tasks[i]["WCET"] = int(C)
        print("Enter the DeadLine of task D",i,":")
        D = input()
        Tasks[i]["DeadLine"] = int(D)
        Complete_Period.append(0)
```

تمامی بخش ها مشابه بخش قبلی پیاده سازی شده است و تنها تفاوت در تابع PriorityCalc است که در آن مشابه بخش ۱ است ولی در این الگوریتم به جای دوره تناوب از ددلاین برای اولویت بندی استفاده شده است. که قطعه کد آن در زیر آورده شده است:

```
def PriorityCalc(RealTime_task):

    """
    Estimates the priority of tasks at each real time period during scheduling
    """
    HP = 10000
    TempPeriod = HP
    P = -1    #Returns -1 for idle tasks
    for i in RealTime_task.keys():
        if (RealTime_task[i]["WCET"] != 0):
            if (TempPeriod > Tasks[i]["DeadLine"]):
                TempPeriod = Tasks[i]["DeadLine"] #Checks the priority of each task based on Deadline
                P = i
    return P
```

۲-۳ نحوه به کارگیری و استفاده از کد

در این بخش با اجرای کد در کنسول از شما تعداد Task به همراه دوره تناوب و بدترین زمان پردازش خواسته می‌شود که پس از وارد کردن این اطلاعات دیاگرام زمانی رسم می‌شود. برای ایجاد تغییر در تعداد Iteration تابع MimimumPeriod که در بخش قبل توضیحات آن داده شده است کافی است که در انتهای کد ورودی تابع را مقدارش را تغییر دهیم:

```
152   Read_Data()
153   hp = MinimumPeriod(20)
154   print("Minimum Peri0d is:", hp)
155   Simulation(hp)
156   DrawGraph()
```

کافی است عدد ۲۰ را با عدد دلخواه خود عوض کنید.

همچنین اگر بخواهید بازه زمانی رسم شده را به دلخواه تغییر دهید کافی است به جای hp در تابع Simulation مقدار دلخواه خود را قرار دهید.

فایل شبیه سازی این بخش با نام DM.py موجود است و باید این فایل اجرا شود.

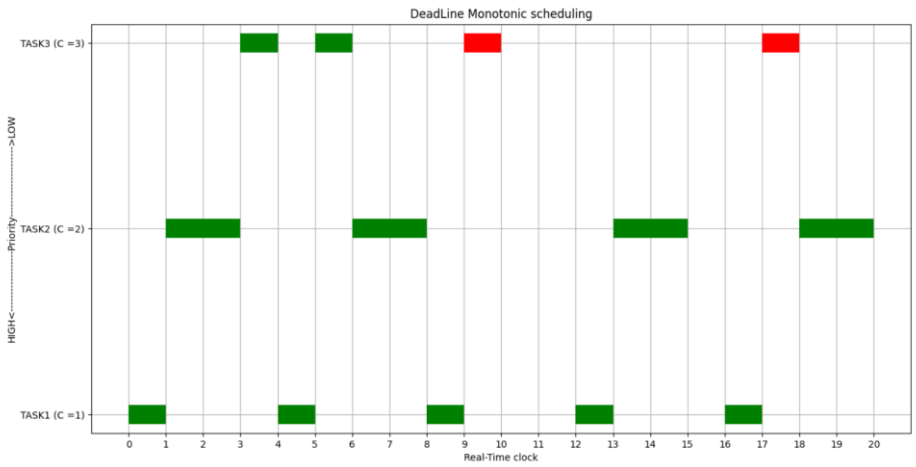
۳-۳ آزمون و مثال‌های متنوع

اگر ددلاین و دوره تناوب یکسان باشد باید رفتار این الگوریتم مشابه با RM باشد به همین دلیل در مثال ۱ به سراغ مثال ۳ بخش اول می‌رویم.

مثال ۱

C3 = 3, T3 = 8　　C2 = 2, T2 = 6　　C1 = 1, T1 = 4

دیاگرام زمانی بالا به صورت زیر است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

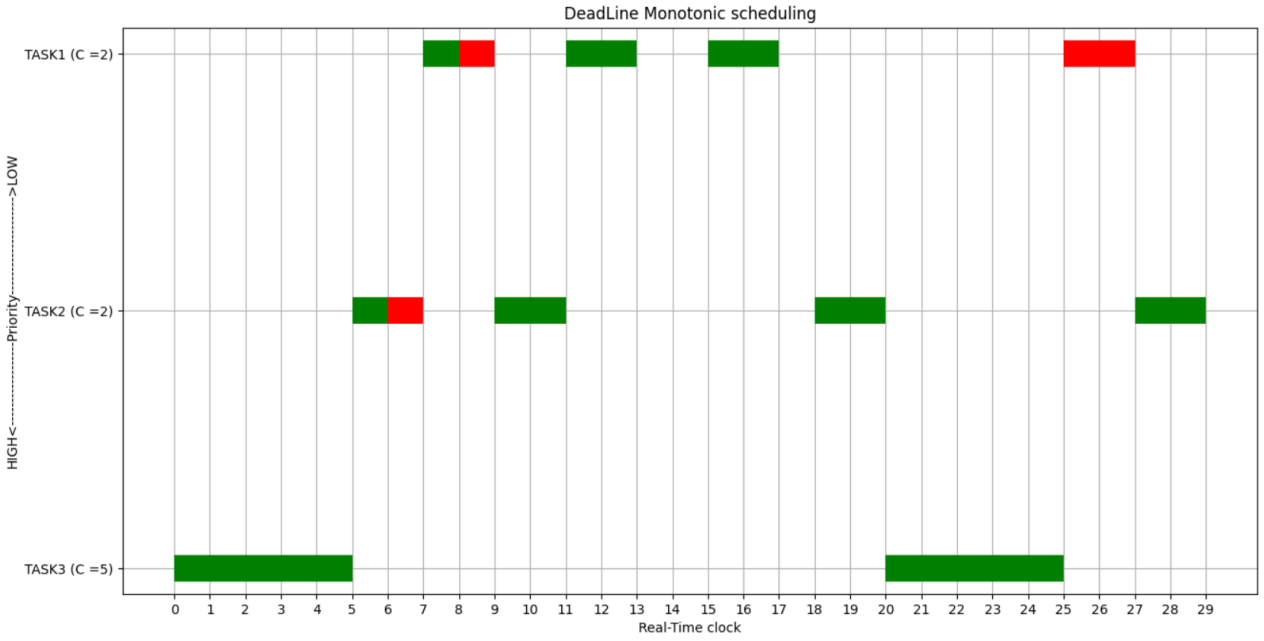
```
worst-case response time 10
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.7797631496846196 Us: 0.9583333333333333
```

پس در صورتی که T = D در نظر بگیریم این الگوریتم مشابه RM رفتار می‌کند.

مثال ۲

T3 = 20, C3 = 5, D3 = 5 T2 = 9, C2 = 2, D2 = 6 T1 = 5, C1 = 2, D1 = 8

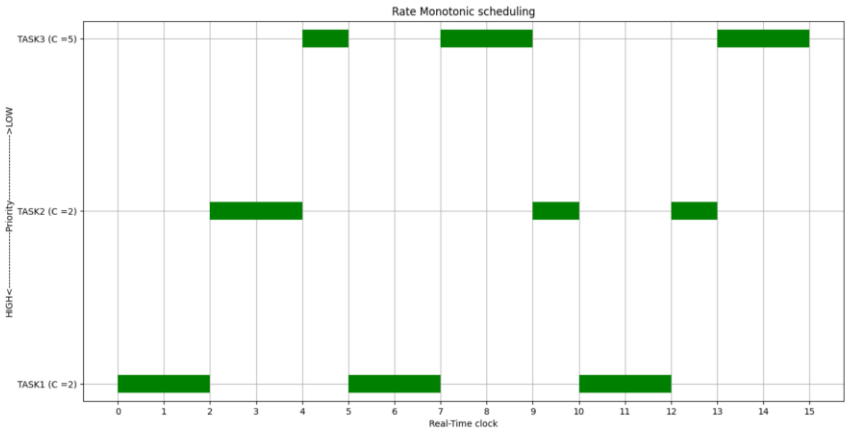
دیاگرام زمانی برای مثال بالا در صفحه بعد آورده شده است



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
worst-case response time 15
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.7797631496846196 Us: 1.5833333333333333
```

همانطور که دیده می‌شود این الگوریتم با توجه به ددلاین اولویت بندی خود را انجام داده است حال همین مثال را با الگوریتم RM ولی با این تفاوت که در آن T = D است تست می‌کنیم تا اولویت Taskها و تفاوت آن با این مثال را متوجه شویم:

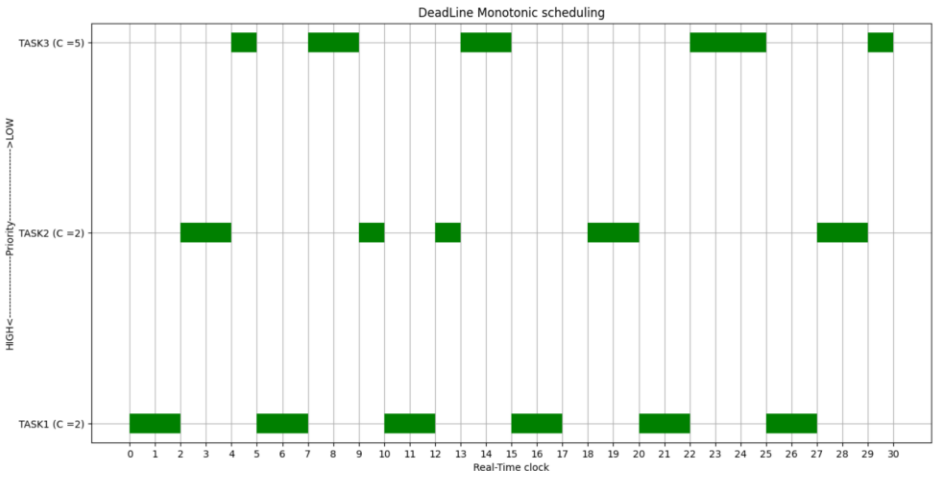


همانطور که دیده شد اولویت ها در الگوریتم RM که شکل بالا است کاملا تفاوت دارد پس الگوریتم DM به درستی در حال رفتار است.

مثال ۳

T3 = 20, C3 = 5, D3 = 15 T2 = 9, C2 = 2, D2 = 6 T1 = 5, C1 = 2, D1 = 3

این مثال۴ بخش ۲ است و دیاگرام زمانی آن با الگوریتم DM به صورت زیر است:



شروط نیز به صورت زیر چک و نمایش داده می‌شوند:

```
worst-case response time 15
We don't know that we can use RM for scheduling or not and we need exact analysis
 Ub: 0.7797631496846196 Us: 1.3333333333333333
```

همانطور که دیده می‌شود زمان بندی آن به درستی انجام شده است و مشابه حالت قبل شروط چک شده بدبینانه است