

به نام پروردگار یگانه

پروژه‌ی سیستم‌های بی‌درنگ
Real-Time Systems
دانشکده‌ی مهندسی کامپیوتر
مدرس: دکتر صفری



دانشگاه صنعتی شریف

گزارش پروژه ۲ فاز دوم

میتراقلی‌پور ۴۰۱۱۰۶۳۶۳

ملیکا علیزاده ۴۰۱۱۰۶۲۵۵

موضوع: تحلیل تأثیر تجاوز از زمان اجرای اسمی (NET) بر زمان پاسخ در سامانه‌های بی‌درنگ

مقدمه:

با توجه به این موضوع که در سامانه‌های بی‌درنگ رعایت صحت زمانی چالشی اساسی است و به دلیل پیچیدگی سخت افزارهای مدرن، تعیین دقیق زمان اجرای بدترین حالت WCET دشوار بوده و به جای آن، زمان اجرای اسمی NET که بر اساس اندازه‌گیری‌های تجربی تعیین می‌شود، مورد استفاده قرار می‌گیرد. از آنجایی که تجاوز از NET می‌تواند منجر به افزایش غیرخطی زمان پاسخ و نقض مهلت‌های زمانی شود در این پروژه سعی داریم به تحلیل اثرات تجاوز از NET در سامانه‌های بی‌درنگ چندهسته‌ای ناهمگن و ارائه روشی سیستماتیک برای شناسایی و بررسی این اثرات پردازیم.

در این پروژه روش پیشنهادی را با سیاست‌های مختلف زمان‌بندی مثل Preemptive, RM, EDF, FIFO و در مدل‌های Non-Preemptive و تحت شرایط Partitioned Scheduling, Global Scheduling, بررسی می‌کنیم. همچنین تجاوز از NET مدل‌سازی شده و اثرات آن تا زمان از بین رفتن پیامدهای آن بررسی می‌شود. برای هر مقدار تجاوز e کران بالایی زمان پاسخ (e) Ri به عنوان یک رویداد تجاوز محاسبه شده و مقادیر بحرانی e با استفاده از الگوریتم‌های جستجوی نمایی و دودویی شناسایی می‌شوند.

پیاده‌سازی:

در این گزارش به تشریح پیاده‌سازی تعدادی از الگوریتم زمان‌بندی با در نظر گرفتن امکان تجاوز زمانی (overrun) در زمان اجرای تسك‌ها پرداخته ایم. در کد پیاده‌سازی شده یک شبیه‌ساز چندپردازنده‌ای است که الگوریتم‌های مختلف تخصیص تسك به هسته‌ها و زمان‌بندی‌های متفاوت را پیاده‌سازی کرده ایم. در بخش اول به توضیح کلاس‌های اصلی می‌پردازیم.

بخش اول: توضیح کلاس‌های اصلی

۱ - کلاس Task

این کلاس برای پیاده‌سازی تسك‌ها تعریف شده است.
متغیرهای اصلی ورودی:

• **Id:** شناسه منحصر به فرد تسك

• **Period:** دوره تناوب تسك

(Worst-Case Execution Time) **Wcet:** •

(Normal Execution Time) **Net:** •

Utilization: • میزان استفاده از پردازنده

متغیرهای اصلی خروجی:

تعداد مهلت‌های از دست رفته **missed_deadlines:** •

زمان پاسخ **response_time:** •

میزان تجاوز زمانی اتفاق افتاده **overrun:** •

```
class Task:  
    def __init__(self, id, period, wcet, net, utilization):  
        self.id = id  
        self.period = max(1, period)  
        self.wcet = max(1, wcet)  
        self.net = min(max(1, net), self.wcet)  
        self.utilization = min(utilization, 1.0)  
        self.missed_deadlines = 0  
        self.response_time = 0  
        self.total_jobs = 0  
        self.core_assignment = -1  
        self.total_execution = 0  
        self.overrun = 0  
        self.max_response = 0  
  
    def __lt__(self, other):  
        return self.period < other.period
```

۲ - کلاس Job

نماینده یک نمونه از اجرای تسك.

صفات اصلی ورودی:

زمان آزادسازی **release_time:** •

• **deadline:** مهلت زمانی (زمان آزادسازی + دوره تناوب)

صفات اصلی نگهدارنده حالت:

زمان باقیمانده برای اجرا **Remaining:** •

میزان تجاوز زمانی این نمونه اجرا **Overrun:** •

```

class Job:
    def __init__(self, task, release_time):
        self.task = task
        self.release_time = release_time
        self.deadline = release_time + task.period
        self.remaining = task.wcet
        self.actual_execution = 0
        self.core = -1
        self.start_time = -1
        self.overrun = 0

    def __lt__(self, other):
        return self.deadline < other.deadline

```

Core - ۳ - کلاس

نماینده یک هسته پردازنده.
صفات اصلی هسته:

• لیست تسک‌های تخصیص یافته Tasks:

صفات نگهدارنده حالت هسته:

• جاب در حال اجرا current_job:

متغیرهای اصلی خروجی:

• تاریخچه زمانبندی schedule:

• میزان بهره‌وری از هسته utilization:

```

class Core:
    def __init__(self, id):
        self.id = id
        self.tasks = []
        self.current_job = None
        self.time = 0
        self.utilization = 0
        self.actual_utilization = 0
        self.schedule = []
        self.next_round_robin = 0 # For RR scheduling

    def add_task(self, task):
        if self.utilization + task.utilization <= 1.0:
            self.tasks.append(task)
            self.utilization += task.utilization
            task.core_assignment = self.id
            return True
        return False

    def reset(self):
        self.current_job = None
        self.time = 0
        self.actual_utilization = 0
        self.schedule = []
        self.next_round_robin = 0
        for task in self.tasks:
            task.missed_deadlines = 0
            task.response_time = 0
            task.total_jobs = 0
            task.total_execution = 0
            task.overrun = 0
            task.max_response = 0

```

کلاس اصلی که منطق زمانبندی را پیادهسازی می‌کند. این کلاس شامل توابعی مشترک برای تمام الگوریتم‌ها و توابعی خاص برای پیادهسازی دقیق هر کدام از الگوریتم‌های زمانبندی می‌باشد که در این بخش به توابع مشترک و در بخش بعد به توابع انحصاری اشاره می‌کنیم.

```
class Scheduler:
    def __init__(self, num_cores, scheduling_policy, allocation_policy, is_preemptive):
        self.num_cores = num_cores
        self.cores = [Core(i) for i in range(num_cores)]
        self.scheduling_policy = scheduling_policy
        self.allocation_policy = allocation_policy
        self.is_preemptive = is_preemptive
        self.hyper_period = 800
        self.total_utilization = 0
        self.overrun_stats = defaultdict(list)
```

تولید Task متد generate_tasks تعدادی تسك تولید می‌کند که برای هر کدام از آنها فرایند زیر انجام می‌شود:

- دوره‌های تناوب از مجموعه ثابتی انتخاب می‌شوند.
- زمان اجرا بر اساس میزان استفاده (utilization) محاسبه می‌شود.
- NET به صورت تصادفی بین ۷۰ تا ۱۰۰ درصد از WCET انتخاب می‌شود.

```
def generate_tasks(self, num_tasks, target_utilization=None):
    tasks = []
    periods = [8, 10, 20, 25, 50, 100, 40, 80, 16, 32]

    if target_utilization is None:
        target_utilization = 0.9 * self.num_cores

    utilizations = []
    remaining_util = target_utilization
    for i in range(num_tasks):
        if i == num_tasks - 1:
            util = remaining_util
        else:
            util = remaining_util * (1 - random.random() ** (1 / (num_tasks - i)))
        util = max(0.05, min(0.95, util))
        utilizations.append(util)
        remaining_util -= util

    for i in range(num_tasks):
        period = random.choice(periods)
        wcet = max(1, int(period * utilizations[i]))
        net = random.randint(max(1, int(0.7 * wcet)), wcet)

        task = Task(i, period, wcet, net, utilizations[i])
        tasks.append(task)

    self.total_utilization = sum(t.utilization for t in tasks)
    return tasks
```

تخصیص Task ها به هسته‌ها برای این کار دو حالت زیر را پیادهسازی کرده‌ایم:

- تخصیص به صورت Global: که در این حالت هر تسك روی هر پردازنده آزاد می‌تواند زمانبندی و اجرا شود.
 - تخصیص به صورت Partitioned: که در این حالت هر تسك به هسته‌ای مشخص تخصیص یافته می‌شود. برای تخصیص به هسته‌ها، چندین سیاست قابل تصور است که آنها را پیادهسازی کرده‌ایم:
- Best-Fit: تسك را به هسته‌ای تخصیص می‌دهد که کمترین فضای خالی پس از تخصیص داشته باشد.

: تسک را به هسته‌ای تخصیص می‌دهد که بیشترین فضای خالی پس از تخصیص داشته باشد. **Worst-Fit** –

: تسک را به اولین هسته‌ای که ظرفیت دارد تخصیص می‌دهد. **First-Fit** –

: تسک‌ها را به صورت چرخشی بین هسته‌ها توزیع می‌کند. **Round-Robin** –

```
def allocate_tasks(self, tasks):
    if 'partitioned' in self.allocation_policy:
        for core in self.cores:
            core.tasks = []
            core.utilization = 0.0
    if 'best_fit' in self.allocation_policy:
        self._allocate_best_fit(tasks)
    elif 'worst_fit' in self.allocation_policy:
        self._allocate_worst_fit(tasks)
    elif 'first_fit' in self.allocation_policy:
        self._allocate_first_fit(tasks)
    elif 'round_robin' in self.allocation_policy:
        self._allocate_round_robin(tasks)
    else:
        # Default to best-fit decreasing
        self._allocate_best_fit_decreasing(tasks)
    else: # global scheduling
        for core in self.cores:
            core.tasks = tasks.copy()
            core.utilization = self.total_utilization
            for task in tasks:
                task.core_assignment = core.id
```

```
def _allocate_best_fit_decreasing(self, tasks):
    """Best-fit decreasing heuristic for task allocation"""
    sorted_tasks = sorted(tasks, key=lambda x: -x.utilization)

    for task in sorted_tasks:
        best_core = None
        min_remaining = float('inf')

        for core in self.cores:
            remaining = 1.0 - (core.utilization + task.utilization)
            if remaining >= 0 and remaining < min_remaining:
                best_core = core
                min_remaining = remaining

        if best_core:
            best_core.add_task(task)
        else:
            # If no core has enough space, put it on the least utilized core
            min_core = min(self.cores, key=lambda x: x.utilization)
            min_core.add_task(task)
```

```

def _allocate_best_fit(self, tasks):
    """Best-fit heuristic for task allocation"""
    for task in tasks:
        best_core = None
        min_remaining = float('inf')

        for core in self.cores:
            remaining = 1.0 - (core.utilization + task.utilization)
            if remaining >= 0 and remaining < min_remaining:
                best_core = core
                min_remaining = remaining

        if best_core:
            best_core.add_task(task)
        else:
            min_core = min(self.cores, key=lambda x: x.utilization)
            min_core.add_task(task)

```

```

def _allocate_worst_fit(self, tasks):
    """Worst-fit heuristic for task allocation"""
    for task in tasks:
        worst_core = None
        max_remaining = -1

        for core in self.cores:
            remaining = 1.0 - (core.utilization + task.utilization)
            if remaining >= 0 and remaining > max_remaining:
                worst_core = core
                max_remaining = remaining

        if worst_core:
            worst_core.add_task(task)
        else:
            min_core = min(self.cores, key=lambda x: x.utilization)
            min_core.add_task(task)

```

```

def _allocate_first_fit(self, tasks):
    """First-fit heuristic for task allocation"""
    for task in tasks:
        allocated = False
        for core in self.cores:
            if core.add_task(task):
                allocated = True
                break
        if not allocated:
            min_core = min(self.cores, key=lambda x: x.utilization)
            min_core.add_task(task)

```

پیاده‌سازی الگوریتم زمان‌بندی این قسمت برای هر کدام از الگوریتم‌های زمان‌بندی متفاوت بوده و به صورت مجزا در بخش بعد توضیح داده می‌شود.

شبیه‌سازی عملی به کمک تابع simulate فرایند ورود تسك‌ها، زمان‌بندی آنها و انجام‌شان روی هسته‌ها را شبیه‌سازی کرده و نتایج عملی را استخراج کرده‌ایم. این شبیه‌سازی با توجه به پیشگیرانه بودن یا نبودن، صورت گرفته است که روند کلی هر دو به صورت زیر است:

- ابتدا job‌ها را تولید کرده و آنها را بر اساس زمان ورودشان مرتب کردیم.
- با جلو رفتن روی زمان، job‌ها را روی هسته‌ها زمان‌بندی می‌کنیم. (در صورت پیشگیرانه بودن سیاست، ممکن است یک job چندین بار زمان‌بندی شود)
- در صورت پایان job قبل از ددلاین، آن را موفقیت‌آمیز و در غیر این صورت آن را از دست رفته محسوب می‌کنیم. توجه شود که هر کدام از job‌ها ممکن است با احتمال ۱۰ درصد Overrun داشته باشد و میزان Overrun به صورت تصادفی بین ۰ تا (NET – WCET) انتخاب می‌شود.

```

def _allocate_round_robin(self, tasks):
    """Round-robin task allocation"""
    core_idx = 0
    for task in tasks:
        allocated = False
        start_idx = core_idx
        while True:
            if self.cores[core_idx].add_task(task):
                allocated = True
                core_idx = (core_idx + 1) % self.num_cores
                break
            core_idx = (core_idx + 1) % self.num_cores
            if core_idx == start_idx:
                break
        if not allocated:
            min_core = min(self.cores, key=lambda x: x.utilization)
            min_core.add_task(task)

```

```

def simulate(self, tasks, max_time=800):
    if 'partitioned' in self.allocation_policy:
        self._simulate_partitioned(tasks, max_time)
    else:
        self._simulate_global(tasks, max_time)

def _simulate_partitioned(self, tasks, max_time):
    for core in self.cores:
        if core.tasks:
            self._simulate_core_edf(core, max_time)

```

```

def _simulate_core_edf(self, core, max_time):
    jobs = []
    for task in core.tasks:
        releases = range(0, max_time, task.period)
        task.total_jobs = len(releases)
        for release in releases:
            jobs.append(Job(task, release))

    jobs.sort(key=lambda x: x.release_time)
    ready_queue = []
    current_job = None
    time = 0

    while time < max_time and (jobs or ready_queue or current_job):
        # Add newly released jobs
        while jobs and jobs[0].release_time <= time:
            job = jobs.pop(0)
            heapq.heappush(ready_queue, job)

        # Handle preemption if enabled
        if self.is_preemptive and current_job and ready_queue:
            if ready_queue[0].deadline < current_job.deadline:
                heapq.heappush(ready_queue, current_job)
                current_job = None

        # Get next job to execute
        if not current_job and ready_queue:
            current_job = heapq.heappop(ready_queue)
            current_job.start_time = time

```

محاسبه زمان پاسخ به صورت نظری دوتابع برای محاسبه نظری زمان پاسخ پیاده‌سازی شده است:

- برای تحلیل در حالت پیشگیرانه `calculate_response_time_preemptive`:

- برای تحلیل در حالت غیرپیشگیرانه `calculate_response_time_nonpreemptive`:

```

def _simulate_global(self, tasks, max_time):
    jobs = []
    for task in tasks:
        releases = range(0, max_time, task.period)
        task.total_jobs = len(releases)
        for release in releases:
            jobs.append(Job(task, release))

    jobs.sort(key=lambda x: x.release_time)
    ready_queue = []
    current_jobs = {core.id: None for core in self.cores}
    time = 0

    while time < max_time and (jobs or ready_queue or any(current_jobs.values())):
        # Add newly released jobs
        while jobs and jobs[0].release_time <= time:
            job = jobs.pop(0)
            heapq.heappush(ready_queue, job)

        # Handle preemption if enabled
        if self.is_preemptive:
            for core_id, current_job in current_jobs.items():
                if current_job and ready_queue:
                    if ready_queue[0].deadline < current_job.deadline:
                        heapq.heappush(ready_queue, current_job)
                        current_jobs[core_id] = None

    # Assign jobs to idle cores

```

```

def _get_execution_time(self, job):
    execution = job.task.net
    if random.random() < 0.1: # 10% chance of overrun
        overrun = random.randint(0, job.task.wcet - job.task.net)
        execution += overrun
        job.overrun = overrun
        job.task.overrun += overrun
        self.overrun_stats[job.task.id].append(overrun)
    return execution

```

```

def analyze_results(self, tasks, max_time, scenario_name):
    task_metrics = []
    critical_overruns = []
    nonlinear_points = []

    for task in tasks:
        # Get tasks on the same core for response time calculation
        core_tasks = [t for t in tasks if t.core_assignment == task.core_assignment]

        # Calculate theoretical response time based on scheduling policy
        if self.is_preemptive:
            theoretical_response = calculate_response_time_preemptive(task, core_tasks)
        else:
            theoretical_response = calculate_response_time_nonpreemptive(task, core_tasks)

        expected_completions = max(1, task.total_jobs)
        actual_completions = expected_completions - task.missed_deadlines

        # Calculate miss rate (task is missed if it has more than one missed deadline job)
        miss_ratio = 1.0 if task.missed_deadlines > 1 else 0.0

        avg_response = 0
        if actual_completions > 0:
            avg_response = task.response_time / actual_completions

```

```

def calculate_response_time_premptive(task, core_tasks):
    """Calculate response time for preemptive EDF with NET overrun"""
    C_i = task.wcet
    Δ_i = task.wcet - task.net # maximum possible overrun
    T_i = task.period
    D_i = task.period # assuming deadline equals period

    # Initialize variables for iterative calculation
    t_prev = 0
    t_current = C_i + Δ_i

    while t_prev != t_current and t_current <= D_i:
        t_prev = t_current
        sum_interference = 0

        for other_task in core_tasks:
            if other_task.id == task.id:
                continue

            C_j = other_task.wcet
            Δ_j = other_task.wcet - other_task.net
            T_j = other_task.period
            D_j = other_task.period

            # Calculate interference term
            term = math.floor((t_prev - D_j) / T_j) * (C_j + Δ_j)
            sum_interference += max(0, term)

        t_current = C_i + Δ_i + sum_interference

    return min(t_current, D_i)

```

```

def calculate_response_time_nonpreemptive(task, core_tasks):
    """Calculate response time for non-preemptive EDF with NET overrun"""
    # J_i is the maximum blocking time from lower priority tasks
    J_i = max((t.wcet + (t.wcet - t.net)) for t in core_tasks if t.period > task.period) if any

    C_i = task.wcet
    Δ_i = task.wcet - task.net

    # Sum of all higher priority tasks that could be ready
    sum_hp = sum((t.wcet + (t.wcet - t.net)) for t in core_tasks if t.period < task.period)

    return J_i + C_i + Δ_i + sum_hp

```

بخش دوم: توضیح نحوه پیاده‌سازی الگوریتم‌های زمان‌بندی

۱- الگوریتم EDF (Earliest Deadline First)

این الگوریتم را با استفاده از یک صف اولویت‌دار (heap) بر اساس مهلت زمانی (deadline) مدیریت می‌کنیم. با ورود یک جاب جدید، آن را بر اساس مهلت زمانی آن داخل صف قرار می‌دهیم (این کار به صورت بهینه در زمان $O(\log n)$ صورت می‌گیرد که در اینجا n تعداد جاب‌های درون صف است). در زمانی که هسته خالی باشد، جاب با نزدیک‌ترین مهلت زمانی از این صف استخراج شده (این کار به صورت بهینه در زمان $O(1)$ صورت می‌گیرد) و پردازش می‌شود. در صورتی که سیاست پیشگیرانه داشته باشیم، با ورود یک جاب جدید، مهلت زمانی آن با مهلت زمانی job در حال پردازش روی هسته مقایسه می‌شود و در صورت نزدیک‌تر بودن دلاین جاب جدید، آن را پردازش کرده و جاب قبلی را متوقف کرده مجدداً وارد صف می‌کنیم. توجه شود که این کار نیز با توجه به global یا partitional سیاست صورت می‌گیرد. اگر سیاست global داشته باشیم، جاب وارد یک صف عمومی که برای تمامی هسته‌های است و ممکن است روی هر کدام از هسته‌ها پردازش شود وارد می‌شود. اما در صورت partitional بودن، فقط در صف هسته‌ای که مجاز هستیم وارد می‌کنیم.

۲- الگوریتم RM (Rate Monotonic)

الگوریتم RM (Rate Monotonic) در این کد با محوریت اولویت ثابت بر اساس دوره زمانی وظیفه fixed-priority based (on task period) پیاده‌سازی شده است: وظیفی با دوره‌های کوتاه‌تر (shorter periods) بالاترین اولویت را دریافت می‌کند. این اصل در متدهای `lt` کلاس‌های Task و Job انجام شده است، به طوری که مقایسه و مرتب‌سازی هر دو وظیفه و نمونه‌های وظیفه (Jobs) بر اساس مقدار period وظیفه والد انجام می‌شود. در شبیه‌سازی، چه در حالت پارتیشن‌بندی شده (partitioned) (که هر هسته صف آماده خود را دارد) و چه در حالت گلوبال (global) (با یک صف آماده مشترک)، هرگاه هسته‌ای بیکار شود یا نیاز به انتخاب وظیفه جدید باشد، Job با بالاترین اولویت RM (یعنی کوتاه‌ترین دوره) از صف اولویت (که با `heapq` پیاده‌سازی شده) انتخاب و اجرا می‌شود. در صورت فعلی بودن پیشگیری (preemption)، اگر Jobی با اولویت بالاتر (دوره کوتاه‌تر) منتشر شود، می‌تواند اجرای Job با اولویت پایین‌تر را متوقف کند. این رویکرد تضمین می‌کند که وظایف با نرخ (rate) بالاتر، همیشه از اولویت اجرایی بالاتری برخوردار باشند.

۳- الگوریتم FIFO (First-In, First-Out)

الگوریتم FIFO (First-In, First-Out) با تغییر رویکرد صف اولویت پیاده‌سازی شده است. برخلاف EDF که بر اساس دلاین و RM که بر اساس دوره اولویت‌بندی می‌کند، FIFO وظایف را بر اساس زمان انتشار (release time) آن‌ها اولویت‌بندی می‌کند. این تغییر در متد `lt` کلاس Job اعمال شده است، به طوری که `return self.release_time < other.release_time` تضمین می‌کند که Job‌ها با زمان انتشار زودتر، اولویت بالاتری دارند. صف آماده (`ready queue`) که با `heapq` مدیریت می‌شود، به طور خودکار Job‌ها را بر اساس این معیار FIFO مرتب می‌کند. هرگاه هسته‌ای آماده اجرای Job جدیدی باشد، همیشه Job را که اولین بار منتشر شده است (یعنی قدیمی‌ترین Job موجود در صف) از بالای صف بیرون می‌کشد. در زمان‌بندی پیشگیرانه (FIFO)، اگر Jobی با زمان انتشار زودتر از Job در حال اجرا وارد صف شود، می‌تواند Job فعلی را متوقف کرده و به جای آن اجرا شود. در حالت غیر پیشگیرانه (non-preemptive FIFO)، Job در حال اجرا تا اتمام یا دلاین خود ادامه می‌یابد و پیشگیری رخ نمی‌دهد. شبیه‌سازی برای هر دو حالت پارتیشن‌بندی شده و گلوبال، این منطق FIFO را در انتخاب و اجرای Job‌ها اعمال می‌کند.

بخش سوم: بررسی نتایج تحلیل نتایج (analyze_results)

پس از شبیه‌سازی، این متدها عملکرد سیستم را ارزیابی می‌کنند. شامل محاسبه نرخ از دست دادن ددلاين (miss_ratio)، زمان پاسخ (actual_response)، شناسایی اضافه بارهای بحرانی (critical_overruns) و نقاط غیرخطی (nonlinear_points) در زمان پاسخ. همچنین آمار هسته (core_stats) و آمار کلی سیستم را تولید می‌کنند.

```
def analyze_results(self, tasks, max_time, scenario_name):
    task_metrics = []
    critical_overruns = []
    nonlinear_points = []

    for task in tasks:
        # Get tasks on the same core for response time calculation
        core_tasks = [t for t in tasks if t.core_assignment == task.core_assignment]

        # Calculate theoretical response time based on scheduling policy
        if self.is_preemptive:
            theoretical_response = calculate_response_time_preemptive(task, core_tasks)
        else:
            theoretical_response = calculate_response_time_nonpreemptive(task, core_tasks)

        expected_completions = max(1, task.total_jobs)
        actual_completions = expected_completions - task.missed_deadlines

        # Calculate miss rate (task is missed if it has more than one missed deadline job)
        miss_ratio = 1.0 if task.missed_deadlines > 1 else 0.0

        avg_response = 0
        if actual_completions > 0:
            avg_response = task.response_time / actual_completions

        task_metrics.append({
            'id': task.id,
            'period': task.period,
            'wcet': task.wcet,
            'net': task.net,
```

رسم نمودارها (plot_schedulability, plot_response_vs_overrun)

ایجاد نمودارهای بصری برای تحلیل رابطه بین اضافه بار و زمان پاسخ، و همچنین تحلیل قابلیت زمان‌بندی سیستم.

```
def plot_response_vs_overrun(self, results, scenario_name):
    plt.figure(figsize=(12, 8))

    for task in results['task_metrics']:
        if task['overrun'] > 0:
            overruns = results['overrun_stats'].get(task['id'], [])
            if overruns:
                responses = []
                for o in overruns:
                    responses.append(task['net'] + o + task['period'] * 0.3) # Simplified response model

                plt.scatter(overruns, responses, label=f"Task {task['id']}", alpha=0.6)

    plt.xlabel('Overrun Amount (e)')
    plt.ylabel('Response Time')
    plt.title(f'Response Time vs NET Overrun\nScenario: {scenario_name}')
    plt.grid(True)
    plt.legend()
    plt.savefig(f'./results/response_vs_overrun/response_vs_overrun_{scenario_name}.png')
    plt.close()
```

```

def plot_schedulability(self, results, scenario_name):
    plt.figure(figsize=(12, 8))

    util_bins = np.linspace(0, 1, 11)
    miss_ratios = []

    for u in util_bins:
        tasks_in_bin = [t for t in results['task_metrics'] if (u-0.1) <= t['utilization'] < u]
        if tasks_in_bin:
            count_gt_0 = sum(1 for t in tasks_in_bin if t['miss_ratio'] > 0)
            miss_ratios.append(count_gt_0)
        else:
            miss_ratios.append(0)

    plt.bar(util_bins, miss_ratios, width=0.1)
    plt.xlabel('Task Utilization')
    plt.ylabel('Miss count')
    plt.title(f'Schedulability Analysis\nScenario: {scenario_name}')
    plt.grid(True)
    plt.savefig(f'./results/schedulability/schedulability_{scenario_name}.png')
    plt.close()

```

تحلیل مسیر بحرانی با MILP (generate_milp_report, plot_milp_analysis, _analyze_critical_paths)

این بخش از کتابخانه OR-Tools برای مدل‌سازی و حل یک مسئله برنامه‌ریزی خطی عدد صحیح (MILP) استفاده می‌کند. هدف این است که سناریوهای بدترین حالت را برای زمان پاسخ وظایف، به ویژه در حضور اضافه بار، شناسایی کند و امتیاز بحرانی (criticality score) را برای هر وظیفه تعیین کند. این به شناسایی وظایفی که بیشترین تأثیر را بر پایداری سیستم دارند کمک می‌کند.

```

def _setup_milp_solver(self):
    """Initialize MILP solver for critical path analysis"""
    self.milp_solver = pywraplp.Solver('SCIP')
    self.milp_vars = {}

def _add_milp_constraints(self, tasks):
    """Add constraints for critical path analysis"""
    # Create variables for each task
    for task in tasks:
        self.milp_vars[task.id] = self.milp_solver.IntVar(0, task.period, f't{task.id}')

    # Add constraints for dependencies and deadlines
    for task in tasks:
        # Constraint: Execution time <= Period
        self.milp_solver.Add(
            self.milp_vars[task.id] <= task.period - task.wcet
        )

```

```

def _analyze_critical_paths(self, tasks):
    """Use MILP to find critical paths and worst-case scenarios"""
    try:
        self._setup_milp_solver()
        self._add_milp_constraints(tasks)

        # Objective: Maximize response time considering overruns
        objective = self.milp_solver.Objective()
        for task in tasks:
            # Weight by both utilization and potential overrun impact
            weight = task.utilization * (task.wcet - task.net)/task.wcet
            objective.SetCoefficient(self.milp_vars[task.id], weight)
        objective.SetMaximization()

        status = self.milp_solver.Solve()

        if status == pywraplp.Solver.OPTIMAL:
            critical_times = {}
            for task in tasks:
                crit_time = self.milp_vars[task.id].solution_value()
                # Calculate criticality score (0-1)
                criticality = min(1.0, crit_time/(task.period - task.net))
                critical_times[task.id] = {
                    'critical_time': crit_time,
                    'criticality': criticality,
                    'period': task.period,
                    'wcet': task.wcet,
                    'net': task.net
                }
            return critical_times
        except Exception as e:
            print(f'MILP analysis failed: {str(e)}')
    return None

```

```

def plot_milp_analysis(self, critical_paths, tasks, scenario_name):
    """Visualize MILP critical path analysis results"""
    if not critical_paths:
        return

    # Prepare data
    task_ids = []
    criticalities = []
    periods = []
    wcets = []
    nets = []

    for task_id, data in critical_paths.items():
        task_ids.append(task_id)
        criticalities.append(data['criticality'])
        periods.append(data['period'])
        wcets.append(data['wcet'])
        nets.append(data['net'])

    # Create figure
    plt.figure(figsize=(15, 10))

    # Criticality heatmap - FIXED COLORBAR ISSUE
    plt.subplot(2, 2, 1)
    sorted_idx = np.argsort(criticalities)[::-1]
    sorted_crit = np.array(criticalities)[sorted_idx]
    bars = plt.bar(range(len(criticalities)), sorted_crit,
                  color=plt.cm.viridis(sorted_crit))
    plt.xticks(range(len(criticalities)), np.array(task_ids)[sorted_idx], rotation=45)
    plt.xlabel('Task ID')
    plt.ylabel('Criticality Score (0-1)')
    plt.title('Task Criticality Ranking')

```

```
def generate_milp_report(self, critical_paths):
    """Generate report of critical paths identified by MILP"""
    report = "Critical Path Analysis (MILP):\n"
    report += "Task ID | Critical Time | Period | WCET | NET\n"
    report += "-----|-----|-----|-----\n"

    for task_id, crit_time in critical_paths.items():
        task = next(t for t in self.tasks if t.id == task_id)
        report += (f"{task_id:7} | {crit_time:13.2f} | {task.period:6} | "
                  f"{task.wcet:4} | {task.net:3}\n")

    return report
```

اجرای سناریوهای مختلف و تولید گزارش جامع

این تابع مجموعه‌ای از سناریوهای از پیش تعریف شده را اجرا می‌کند. هر سناریو شامل پارامترهایی مانند تعداد هسته‌ها، تعداد وظایف، سیاست زمانبندی و سیاست تخصیص است. برای هر سناریو: یک شیء Scheduler ایجاد می‌شود. وظایف تولید و تخصیص می‌یابند. شبیه‌سازی اجرا می‌شود. نتایج تحلیل شده و نمودارها تولید می‌شوند. در نهایت یک گزارش کامل شامل فایل متّنی و تصاویر نمودارها و تولید CSV می‌شود.

```
def run_comprehensive_scenarios():
    scenarios = []

    # Vary number of tasks
    for n in [10, 50, 100, 200, 400, 500]:
        for alloc_policy in ['partitioned_best_fit', 'partitioned_worst_fit', 'partitioned_first_fit', 'partitioned']:
            scenarios.append({
                'name': f'{alloc_policy}_EDF_{n}tasks_4cores',
                'num_cores': 4,
                'num_tasks': n,
                'scheduling_policy': 'EDF',
                'allocation_policy': alloc_policy,
                'is_preemptive': True,
                'max_time': 800,
                'target_util': 0.8 * 4 # 80% system utilization
            })

    # Vary number of cores
    for c in [4, 8, 16, 32]:
        for alloc_policy in ['partitioned_best_fit', 'partitioned_worst_fit', 'partitioned_first_fit', 'partitioned']:
            scenarios.append({
                'name': f'{alloc_policy}_EDF_100tasks_{c}cores',
                'num_cores': c,
                'num_tasks': 100,
                'scheduling_policy': 'EDF',
                'allocation_policy': alloc_policy,
                'is_preemptive': True,
                'max_time': 800,
                'target_util': 0.8 * c
            })

def generate_detailed_report(all_results):
    with open('./results/comprehensive_edf_report.txt', 'w') as f:
        f.write("Comprehensive EDF Scheduling with NET Overruns Report\n")
        f.write("=====\n")

        # 1. Upper bounds for response time with NET overruns
        f.write("1. Upper Bounds for Response Time with NET Overruns:\n")
        for scenario, results in all_results.items():
            f.write(f"{scenario}:\n")
            f.write(f"- Avg Response/Period: {results['avg_response_ratio']:.2f}\n")
            f.write(f"- Miss Rate: {results['avg_miss_ratio']:.2%}\n")
            f.write(f"- Max Response/Period: {max(t['response_ratio'] for t in results['task_metrics']):.2f}\n")
            f.write(f"- Worst-case Response: {max(t['max_response']) for t in results['task_metrics']} if t['period']\n")

        # 2. Critical overrun values (e)
        f.write("\n2. Critical Overrun Values Causing Nonlinear Response:\n")
        for scenario, results in all_results.items():
            if results['critical_overruns']:
                f.write(f"{scenario}:\n")
                for task in sorted(results['critical_overruns'], key=lambda x: -x['overrun'])[:5]:
                    f.write(f"Task {task['task_id']}: Overrun={task['overrun']} ")
                    f.write(f"(NET={task['net']}, WCET={task['wcet']}) ")
                    f.write(f"Response={task['max_response']} (Period={task['period']})\n")
                f.write("\n")
```

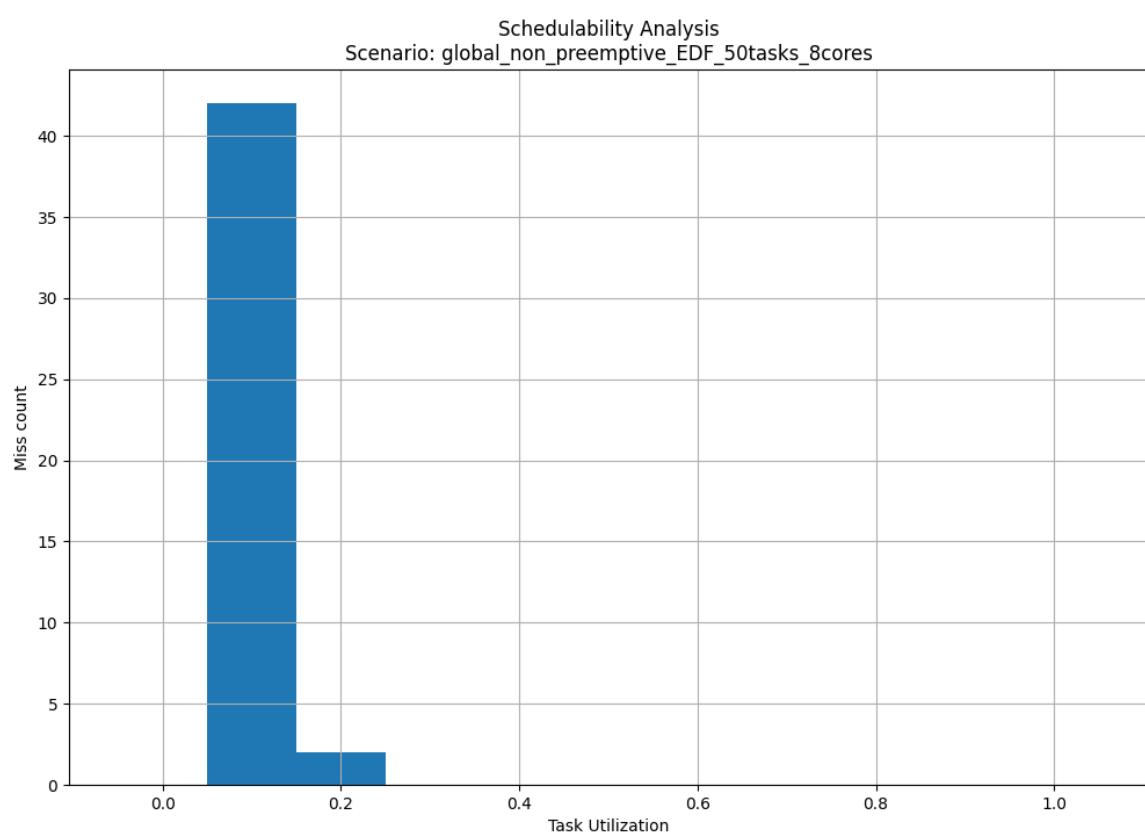
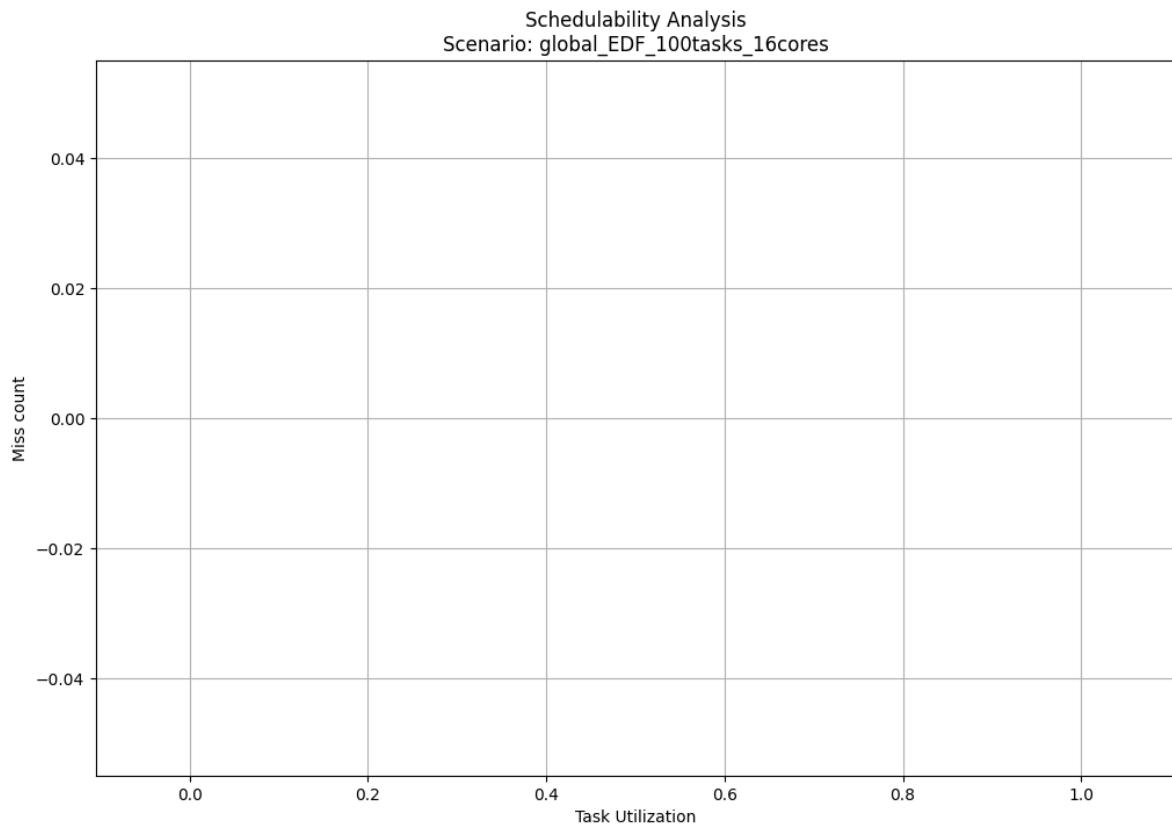
تمامی فایل‌های نتایج در پوشه **result** که ضمیمه شده است وجود دارد. به دلیل زیاد بودن تعداد حالاتی که باید بررسی می‌شد و محدودیت کامپایل لتك، در ادامه تنها بخشی از نتایج آورده شده است و باقی نتایج در پوشه **result** قابل مشاهده است.

نتایج: EDF

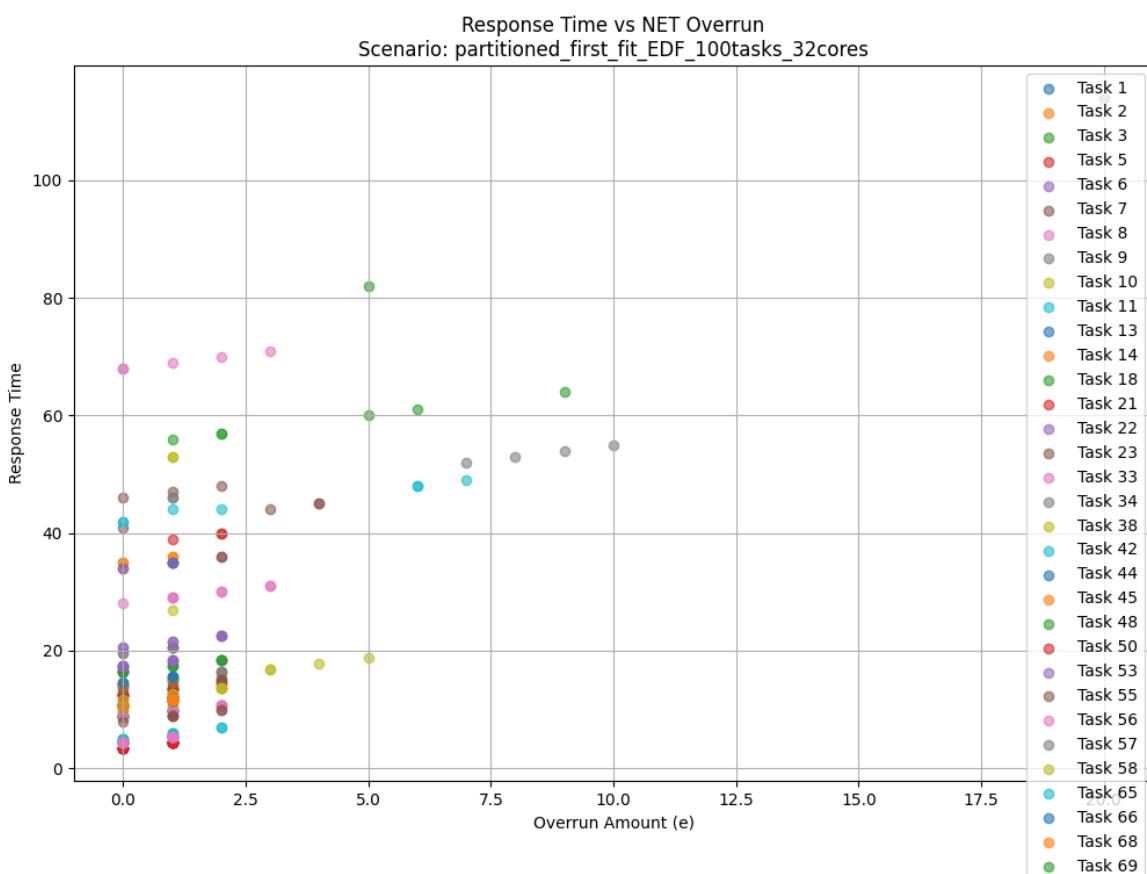
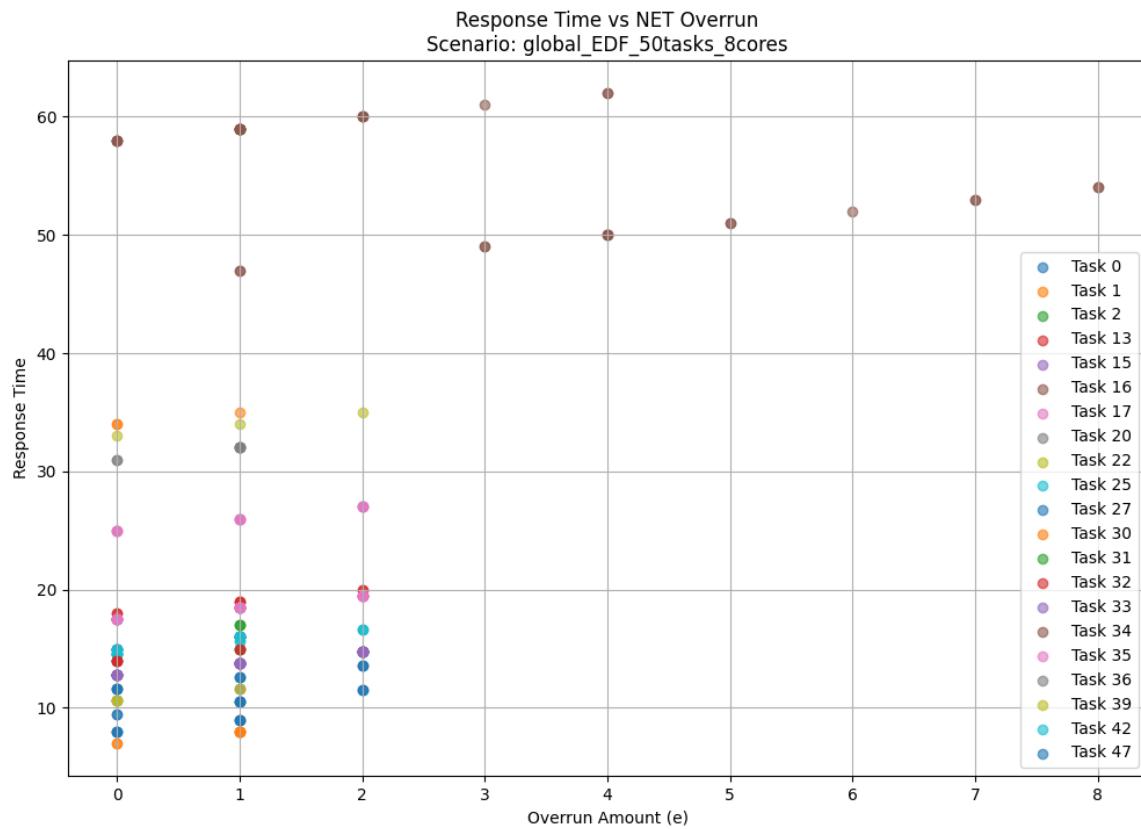
نتایج تکمیلی:

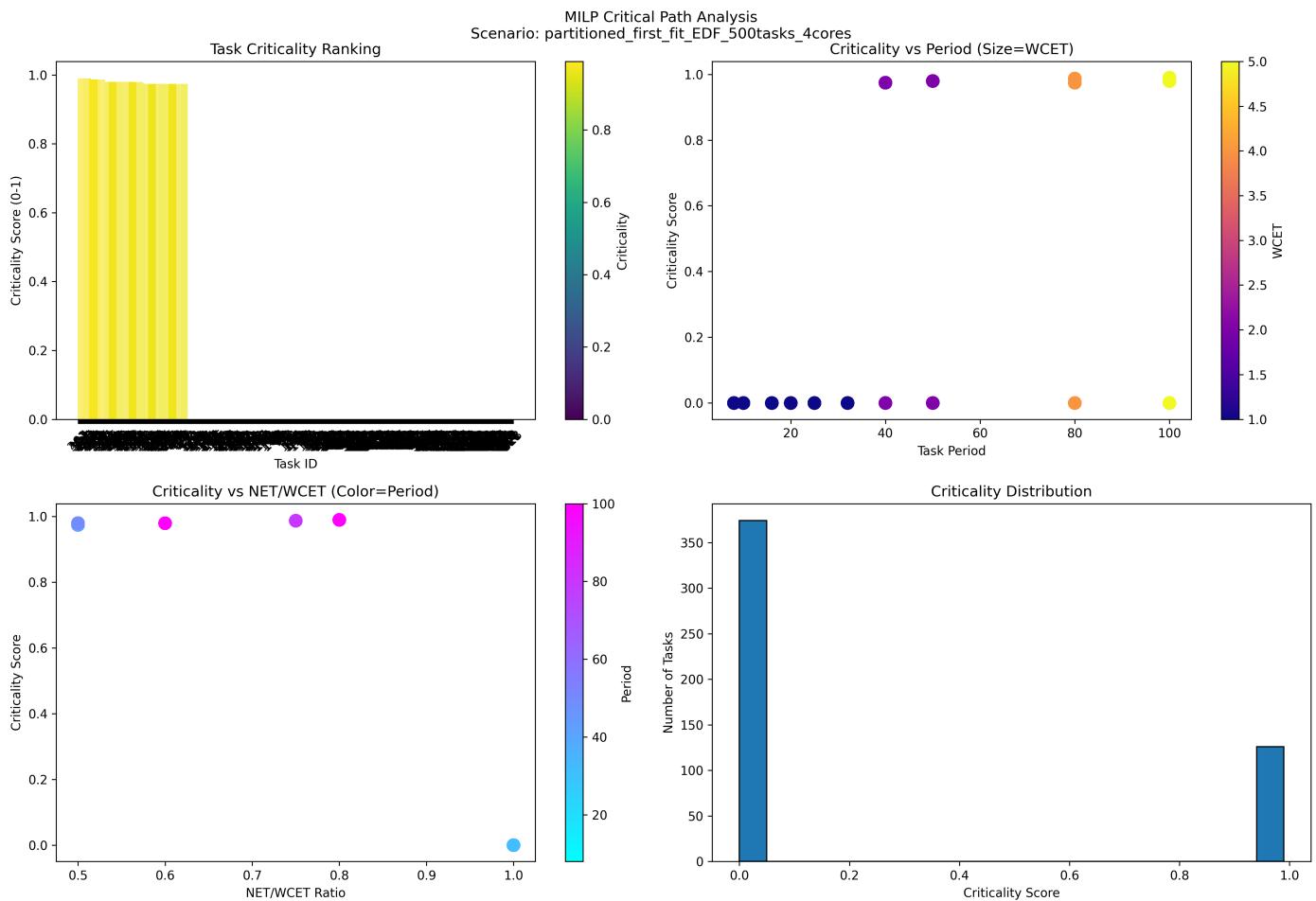
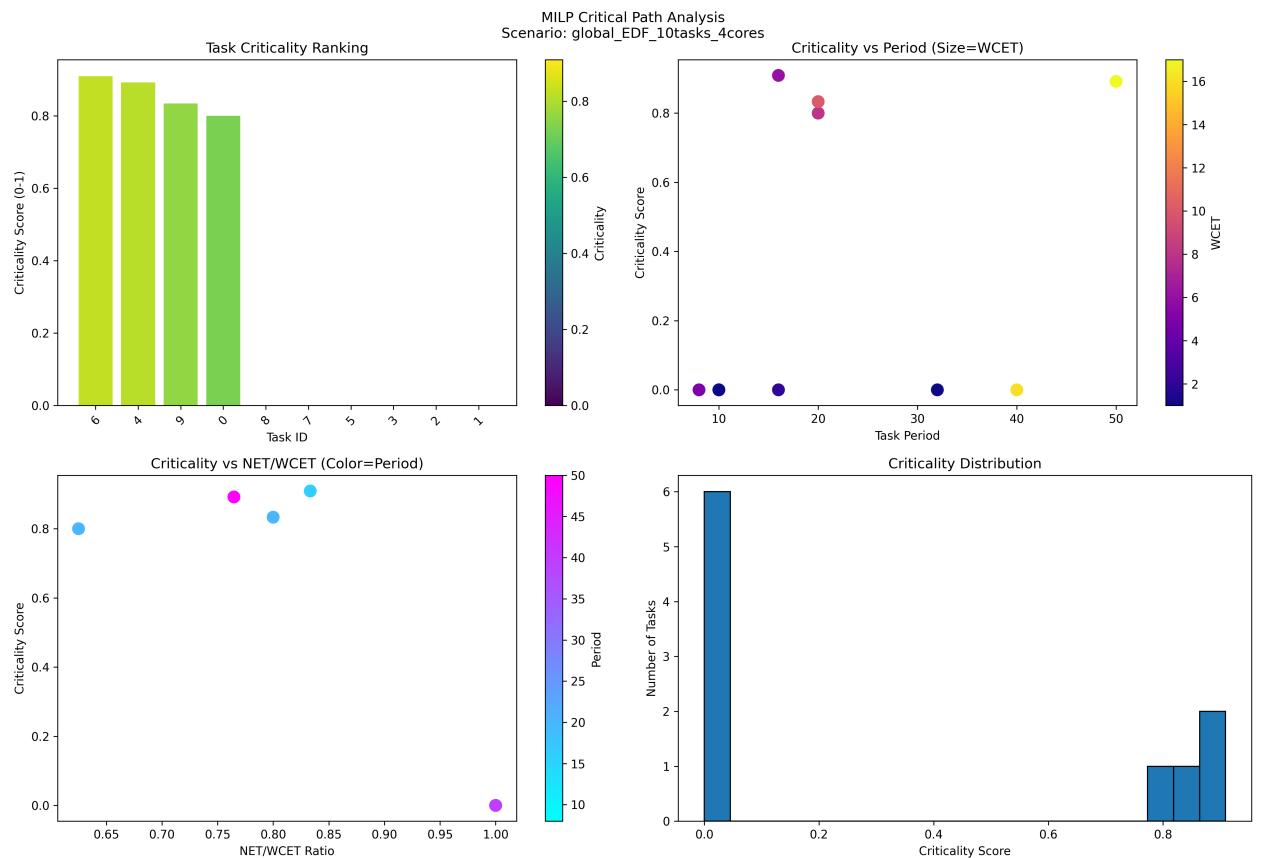
```
results > detailed_results.csv
1 scenario,task_id,period,wcet,net,utilization,miss_ratio,max_response,overrun,core
2 partitioned_best_fit_EDF_10tasks_4cores,0,40,26,23,0.6733089580519214,0.0,32,22,0
3 partitioned_best_fit_EDF_10tasks_4cores,1,8,1,1,0.07949094176898093,0.0,1,0,0
4 partitioned_best_fit_EDF_10tasks_4cores,2,25,15,10,0.6193319974589614,1.0,25,17,1
5 partitioned_best_fit_EDF_10tasks_4cores,3,50,16,12,0.3380989337795098,0.0,50,11,1
6 partitioned_best_fit_EDF_10tasks_4cores,4,20,1,1,0.05,0.0,6,0,0
7 partitioned_best_fit_EDF_10tasks_4cores,5,20,5,5,0.2903877162548367,0.0,10,0,2
8 partitioned_best_fit_EDF_10tasks_4cores,6,10,3,3,0.3173674249385011,0.0,5,0,2
9 partitioned_best_fit_EDF_10tasks_4cores,7,50,2,2,0.05,0.0,35,0,0
10 partitioned_best_fit_EDF_10tasks_4cores,8,8,1,1,0.18738477066846781,0.0,4,0,2
11 partitioned_best_fit_EDF_10tasks_4cores,9,80,47,35,0.594629257078821,0.0,80,22,3
12 partitioned_worst_fit_EDF_10tasks_4cores,0,10,3,2,0.3975232932000481,0.0,5,12,0
13 partitioned_worst_fit_EDF_10tasks_4cores,1,20,1,1,0.06270238420119867,0.0,2,0,1
14 partitioned_worst_fit_EDF_10tasks_4cores,2,32,1,1,0.057496479859610164,0.0,1,0,2
15 partitioned_worst_fit_EDF_10tasks_4cores,3,80,7,7,0.09601550066788674,0.0,8,0,3
16 partitioned_worst_fit_EDF_10tasks_4cores,4,100,27,19,0.2730065059604946,0.0,44,13,2
17 partitioned_worst_fit_EDF_10tasks_4cores,5,32,2,1,0.06481858236418103,0.0,4,2,1
18 partitioned_worst_fit_EDF_10tasks_4cores,6,16,1,1,0.08855167582154501,0.0,1,0,3
19 partitioned_worst_fit_EDF_10tasks_4cores,7,20,18,16,0.9153114784121954,0.0,0,0,-1
20 partitioned_worst_fit_EDF_10tasks_4cores,8,8,1,1,0.2169918559175728,0.0,1,0,1
21 partitioned_worst_fit_EDF_10tasks_4cores,9,8,7,6,0.95,0.0,0,0,-1
22 partitioned_first_fit_EDF_10tasks_4cores,0,10,1,1,0.200739674247244,1.0,10,0,0
```

<pre>results > comprehensive_edf_report.txt 4 1. Upper Bounds for Response Time with NET Overruns: 5 partitioned_best_fit_EDF_10tasks_4cores: 6 - Avg Response/Period: 0.03 7 - Miss Rate: 10.00% 8 - Max Response/Period: 0.10 9 - Worst-case Response: 80 units 10 11 partitioned_worst_fit_EDF_10tasks_4cores: 12 - Avg Response/Period: 0.01 13 - Miss Rate: 0.00% 14 - Max Response/Period: 0.06 15 - Worst-case Response: 44 units 16 17 partitioned_first_fit_EDF_10tasks_4cores: 18 - Avg Response/Period: 0.02 19 - Miss Rate: 30.00% 20 - Max Response/Period: 0.04 21 - Worst-case Response: 32 units 22 23 partitioned_round_robin_EDF_10tasks_4cores: 24 - Avg Response/Period: 0.04 25 - Miss Rate: 20.00% 26 - Max Response/Period: 0.12 27 - Worst-case Response: 100 units 28 29 global_EDF_10tasks_4cores: 30 - Avg Response/Period: 0.01 31 - Miss Rate: 0.00% 32 - Max Response/Period: 0.04 33 - Worst-case Response: 31 units 34 35 partitioned_best_fit_EDF_50tasks_4cores: 36 - Avg Response/Period: 0.05 37 - Miss Rate: 28.00% 38 - Max Response/Period: 0.17 39 - Worst-case Response: 100 units</pre>	<pre>results > comprehensive_edf_report.txt 238 239 partitioned_worst_fit_EDF_100tasks_32cores: 240 - Avg Response/Period: 0.03 241 - Miss Rate: 11.00% 242 - Max Response/Period: 0.50 243 - Worst-case Response: 100 units 244 245 partitioned_first_fit_EDF_100tasks_32cores: 246 - Avg Response/Period: 0.07 247 - Miss Rate: 45.00% 248 - Max Response/Period: 1.00 249 - Worst-case Response: 100 units 250 251 partitioned_round_robin_EDF_100tasks_32cores: 252 - Avg Response/Period: 0.03 253 - Miss Rate: 10.00% 254 - Max Response/Period: 0.50 255 - Worst-case Response: 100 units 256 257 global_EDF_100tasks_8cores: 258 - Avg Response/Period: 0.02 259 - Miss Rate: 0.00% 260 - Max Response/Period: 0.07 261 - Worst-case Response: 55 units 262 263 global_EDF_100tasks_16cores: 264 - Avg Response/Period: 0.01 265 - Miss Rate: 0.00% 266 - Max Response/Period: 0.05 267 - Worst-case Response: 40 units 268 269 global_EDF_100tasks_32cores: 270 - Avg Response/Period: 0.01 271 - Miss Rate: 0.00% 272 - Max Response/Period: 0.07 273 - Worst-case Response: 58 units 274</pre>
--	--



تغییرات زمان پاسخ وظایف بر حسب میزان تجاوز از NET:





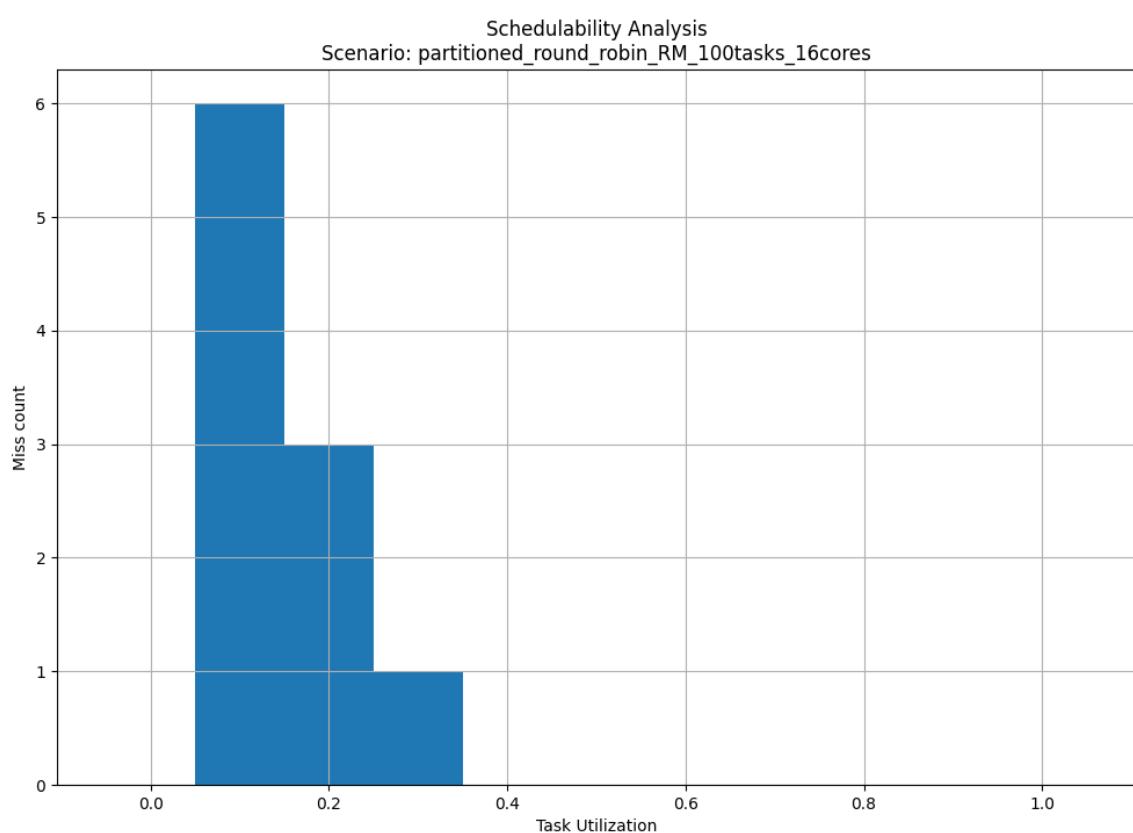
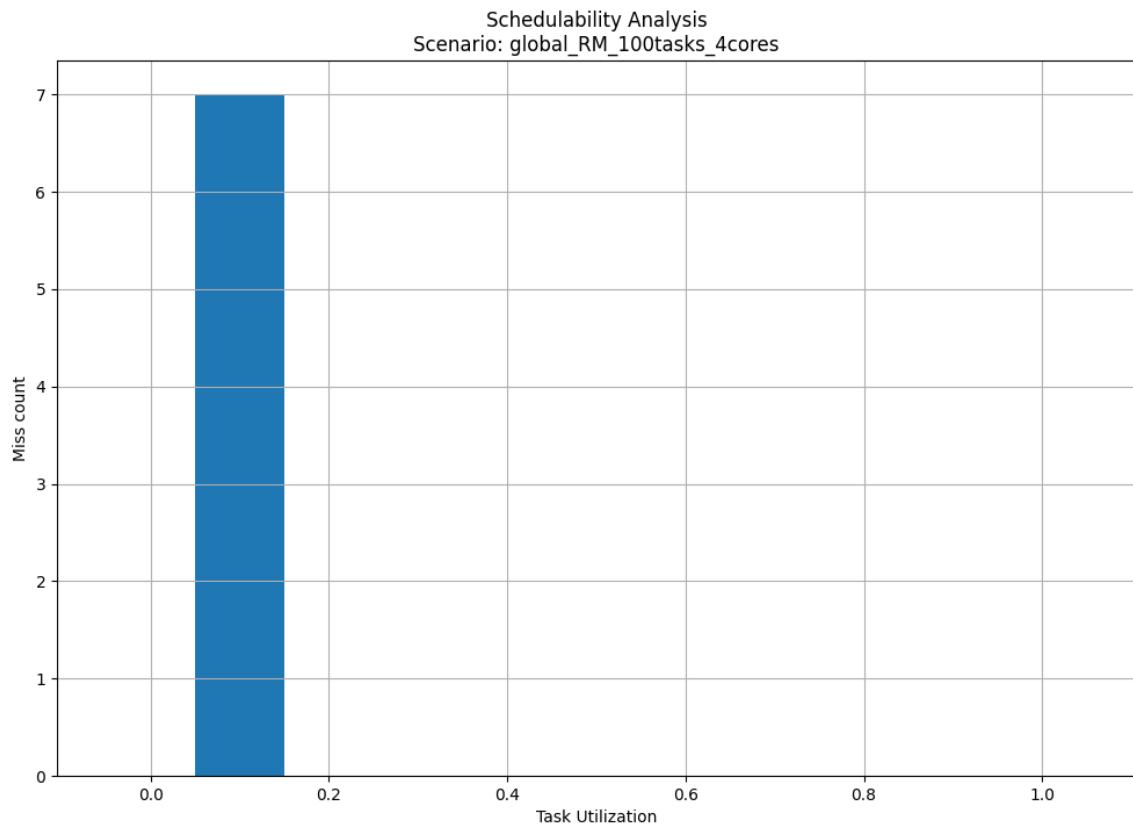
نتایج RM:

نتایج تکمیلی:

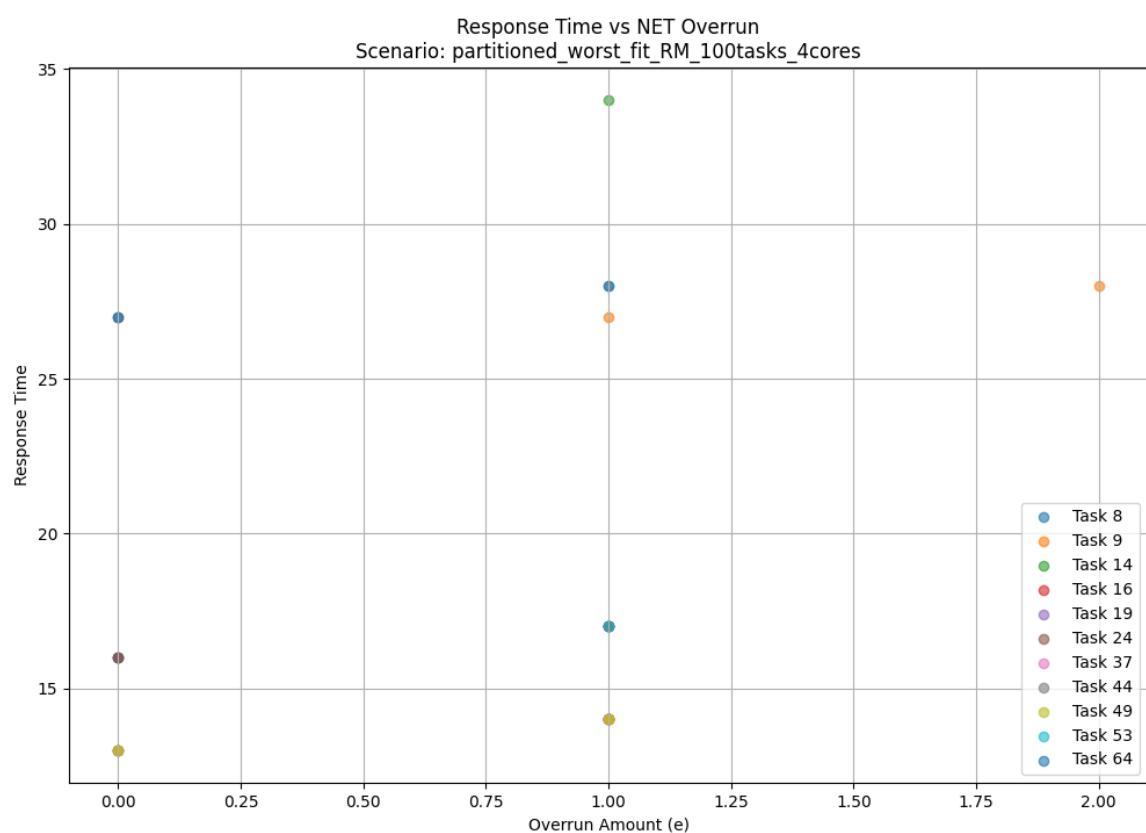
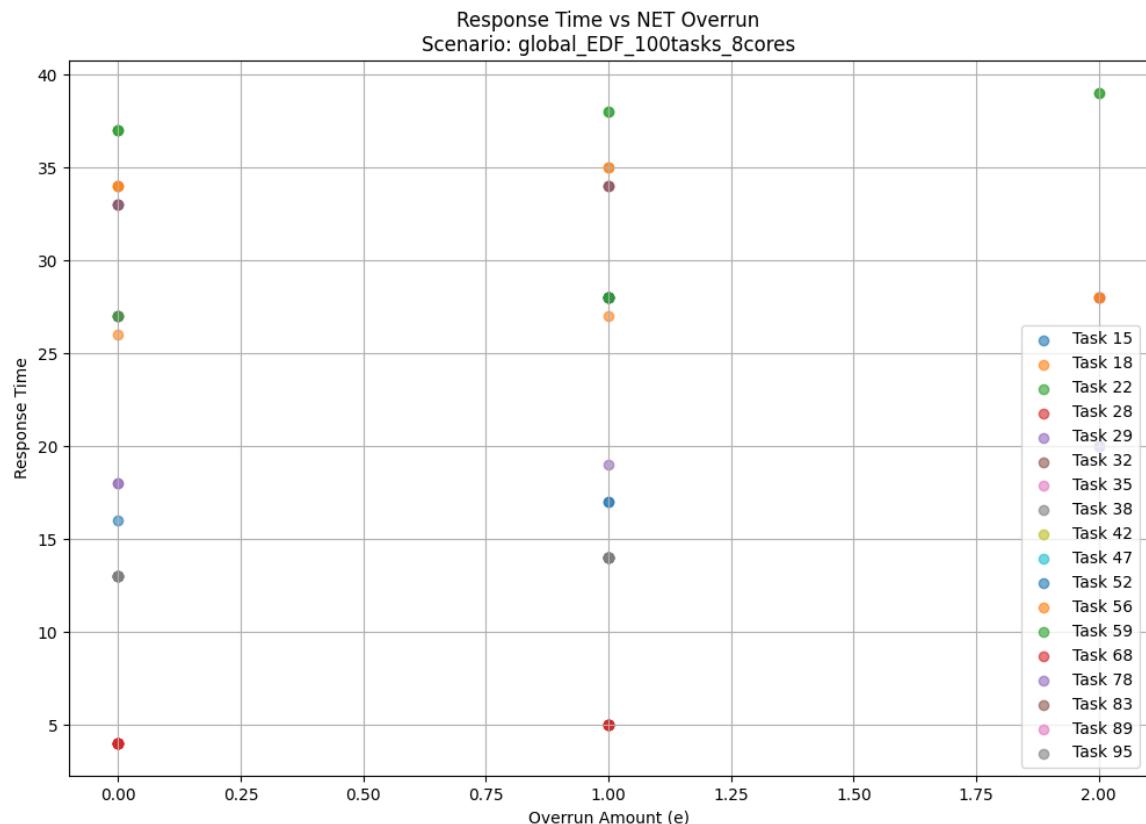
```
results > detailed_results.csv
1 scenario,task_id,period,wcet,net,utilization,miss_ratio,max_response,overrun,core
2 partitioned_best_fit_RM_10tasks_4cores,0,25,1,1,0.05,0.0,3,0,0
3 partitioned_best_fit_RM_10tasks_4cores,1,50,9,9,0.18795984198645407,1.0,50,0,0
4 partitioned_best_fit_RM_10tasks_4cores,2,50,7,7,0.1571466551410596,1.0,50,0,0
5 partitioned_best_fit_RM_10tasks_4cores,3,50,2,2,0.05,1.0,50,0,0
6 partitioned_best_fit_RM_10tasks_4cores,4,40,14,10,0.3500716047378163,0.0,24,13,0
7 partitioned_best_fit_RM_10tasks_4cores,5,16,1,1,0.11187706461969035,0.0,2,0,0
8 partitioned_best_fit_RM_10tasks_4cores,6,80,32,26,0.40483240058548253,0.0,59,7,1
9 partitioned_best_fit_RM_10tasks_4cores,7,40,28,24,0.7142456323988936,0.0,40,1,2
10 partitioned_best_fit_RM_10tasks_4cores,8,8,1,1,0.07990712200152944,0.0,1,0,0
11 partitioned_best_fit_RM_10tasks_4cores,9,25,23,20,0.95,0.0,25,10,3
12 partitioned_worst_fit_RM_10tasks_4cores,0,50,29,25,0.5931944061877531,0.0,50,4,0
13 partitioned_worst_fit_RM_10tasks_4cores,1,16,3,3,0.21592112226745552,0.0,5,0,1
14 partitioned_worst_fit_RM_10tasks_4cores,2,10,8,6,0.8687560015326552,0.0,10,18,2
15 partitioned_worst_fit_RM_10tasks_4cores,3,32,5,5,0.18502561690457578,0.0,12,0,3
16 partitioned_worst_fit_RM_10tasks_4cores,4,80,11,8,0.14348447781514925,0.0,30,4,3
17 partitioned_worst_fit_RM_10tasks_4cores,5,16,3,3,0.218194724267434,0.0,8,0,1
18 partitioned_worst_fit_RM_10tasks_4cores,6,10,3,3,0.3846990789237129,0.0,5,0,3
19 partitioned_worst_fit_RM_10tasks_4cores,7,50,5,4,0.10237884633241574,0.0,30,0,1
20 partitioned_worst_fit_RM_10tasks_4cores,8,50,9,8,0.18414422934022126,0.0,30,0,1
21 partitioned_worst_fit_RM_10tasks_4cores,9,32,9,9,0.30420149642862726,0.0,17,0,0
```

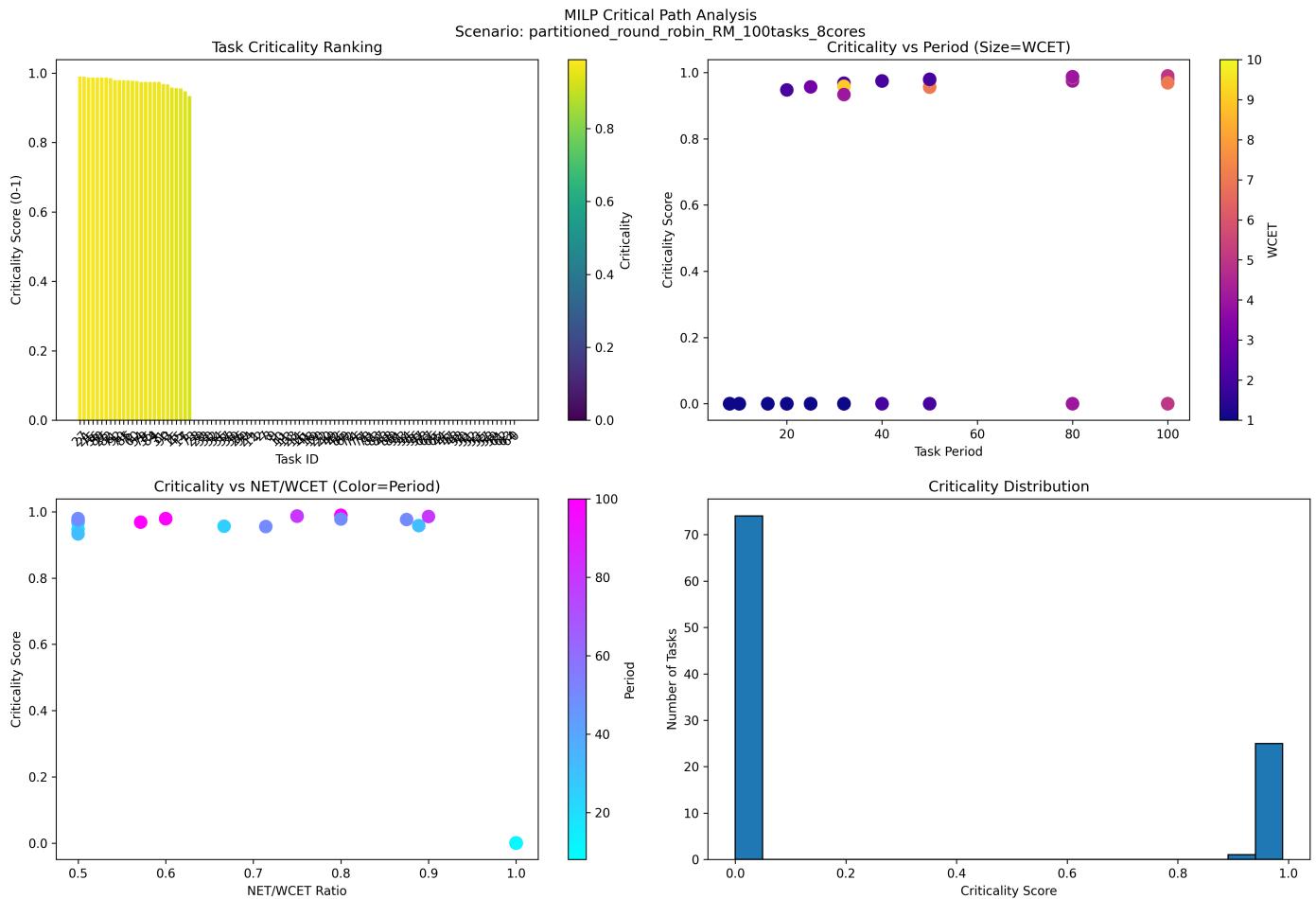
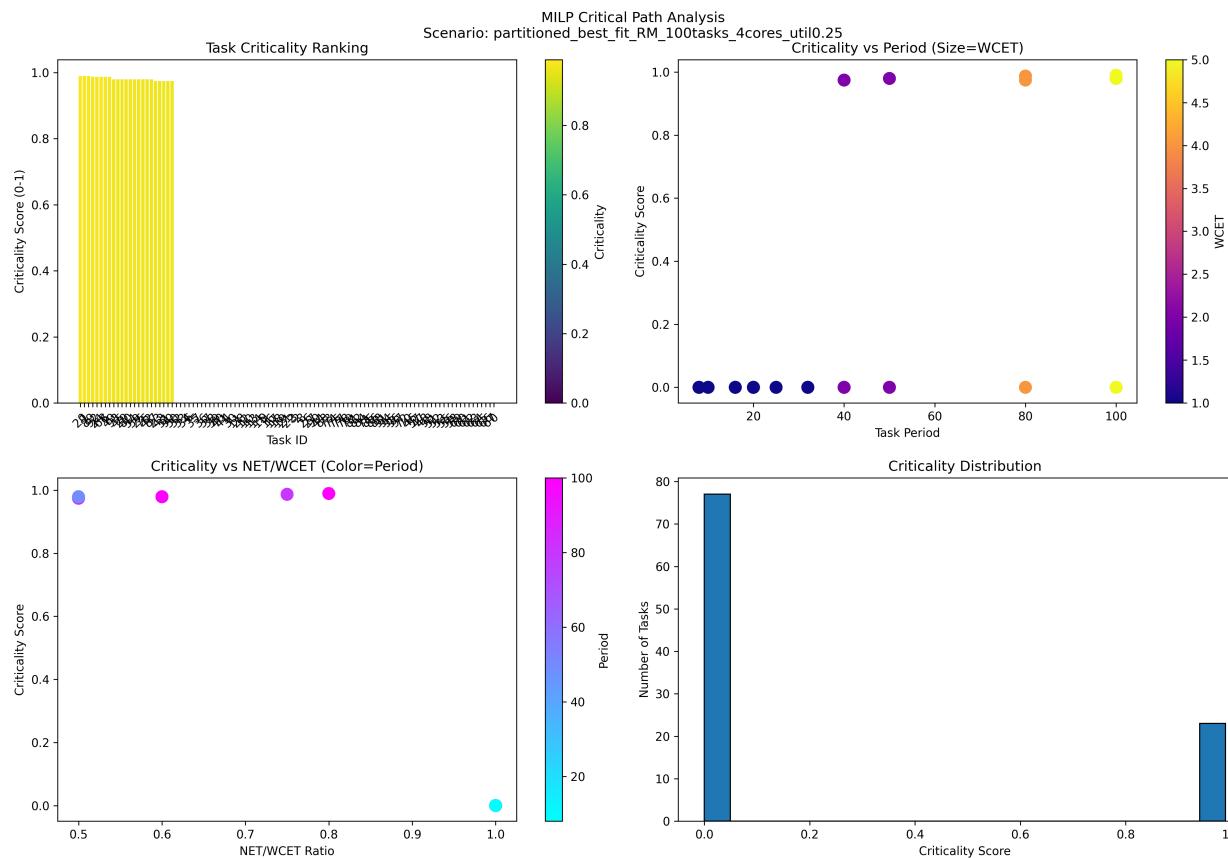
```
results > comprehensive_rm_report.txt
1 Comprehesive RM Scheduling with NET Overruns Report
2 =====
3
4 1. Upper Bounds for Response Time with NET Overruns:
5 partitioned_best_fit_RM_10tasks_4cores:
6 - Avg Response/Period: 0.05
7 - Miss Rate: 30.00%
8 - Max Response/Period: 0.12
9 - Worst-case Response: 59 units
10
11 partitioned_worst_fit_RM_10tasks_4cores:
12 - Avg Response/Period: 0.02
13 - Miss Rate: 0.00%
14 - Max Response/Period: 0.06
15 - Worst-case Response: 50 units
16
17 partitioned_first_fit_RM_10tasks_4cores:
18 - Avg Response/Period: 0.06
19 - Miss Rate: 20.00%
20 - Max Response/Period: 0.50
21 - Worst-case Response: 100 units
22
23 partitioned_round_robin_RM_10tasks_4cores:
24 - Avg Response/Period: 0.02
25 - Miss Rate: 0.00%
26 - Max Response/Period: 0.06
27 - Worst-case Response: 50 units
28
29 global_RM_10tasks_4cores:
30 - Avg Response/Period: 0.02
31 - Miss Rate: 10.00%
32 - Max Response/Period: 0.05
33 - Worst-case Response: 26 units
34
35 partitioned_best_fit_RM_50tasks_4cores:
36 - Avg Response/Period: 0.09
37 - Miss Rate: 14.00%
38 - Max Response/Period: 1.00
39 - Worst-case Response: 100 units
40
41 partitioned_worst_fit_RM_50tasks_4cores:
42 - Avg Response/Period: 0.02
43 - Miss Rate: 0.00%
44 - Max Response/Period: 0.10
45 - Worst-case Response: 80 units
```

```
results > comprehensive_rm_report.txt
1
2 35
3 36 2. Critical Overrun Values Causing Nonlinear Response:
4 37 global_RM_100tasks_4cores:
5 38 Task 71: Overrun=3 (NET=1, WCET=2) Response=59 (Period=40)
6 39 Task 85: Overrun=1 (NET=1, WCET=2) Response=78 (Period=40)
7 40 Task 87: Overrun=1 (NET=1, WCET=2) Response=160 (Period=40)
8
9 41 partitioned_best_fit_non_preemptive_RM_50tasks_8cores:
10 42 Task 12: Overrun=1 (NET=3, WCET=4) Response=191 (Period=80)
11 43
12 44
13 45 3. System Sensitive Points (Nonlinear Response):
14 46 partitioned_first_fit_RM_10tasks_4cores:
15 47 Task 1: Threshold=0.4 Slope=1.00
16 48 Task 6: Threshold=0.8 Slope=1.00
17 49 Task 8: Threshold=0.5 Slope=1.00
18
19 50 partitioned_round_robin_RM_10tasks_4cores:
20 51 Task 1: Threshold=0.4 Slope=1.00
21 52 Task 4: Threshold=0.2 Slope=1.00
22
23 53 global_RM_10tasks_4cores:
24 54 Task 1: Threshold=0.3 Slope=1.00
25 55 Task 6: Threshold=0.1 Slope=31.50
26 56 Task 7: Threshold=0.9 Slope=1.00
27 57 Task 9: Threshold=0.6 Slope=1.00
28
29 58 partitioned_best_fit_RM_50tasks_4cores:
30 59 Task 0: Threshold=0.6 Slope=1.00
31 60 Task 7: Threshold=0.7 Slope=1.00
32 61 Task 24: Threshold=0.2 Slope=1.00
33 62 Task 40: Threshold=0.5 Slope=1.00
34 63 Task 45: Threshold=0.5 Slope=1.00
35
36 64 partitioned_worst_fit_RM_50tasks_4cores:
37 65 Task 5: Threshold=0.8 Slope=1.00
38 66 Task 24: Threshold=0.4 Slope=1.00
39 67 Task 29: Threshold=0.4 Slope=1.00
40
41 68 partitioned_first_fit_RM_50tasks_4cores:
42 69 Task 24: Threshold=0.5 Slope=1.00
43 70 Task 30: Threshold=0.6 Slope=1.00
44 71 Task 40: Threshold=0.4 Slope=1.00
45 72 Task 46: Threshold=0.5 Slope=1.00
46
47 73
48 74
49 75
50 76
51 77
52 78
53 79
54 80
55 81
56 82
57 83
58 84
59 85
60 86
61 87
62 88
63 89
64 90
65 91
66 92
67 93
68 94
69 95
70 96
71 97
72 98
73 99
74 100
75 101
76 102
77 103
78 104
79 105
80 106
81 107
82 108
83 109
84 110
85 111
86 112
87 113
88 114
89 115
90 116
91 117
92 118
93 119
94 120
95 121
96 122
97 123
98 124
99 125
100 126
```



تغییرات زمان پاسخ وظایف بر حسب میزان تجاوز از.NET:





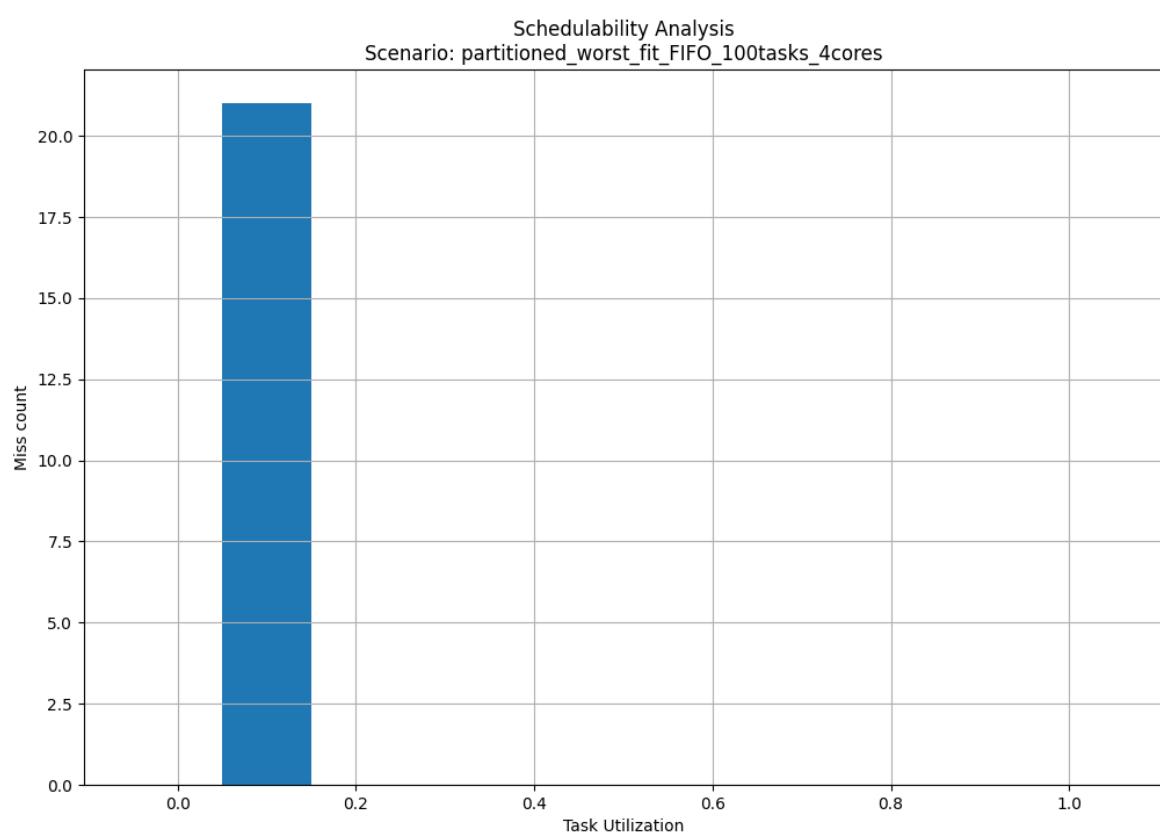
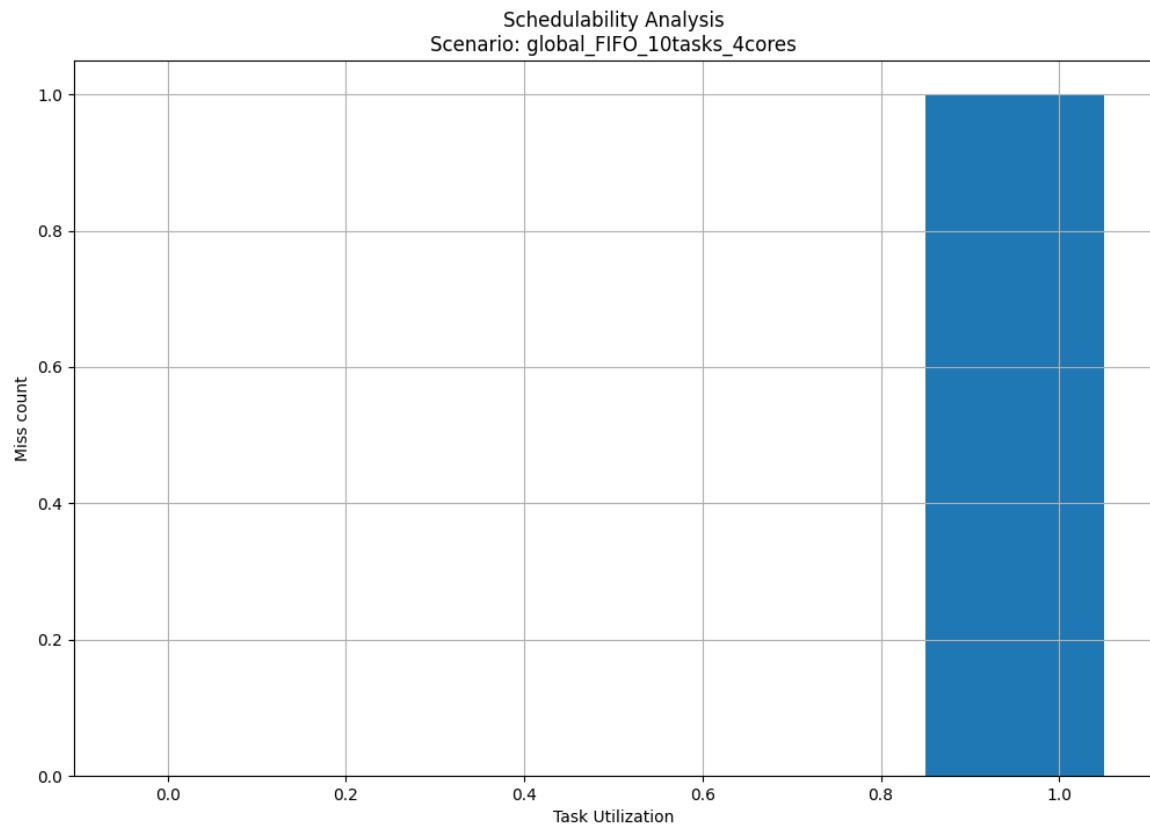
نتایج :FIFO

نتایج تکمیلی:

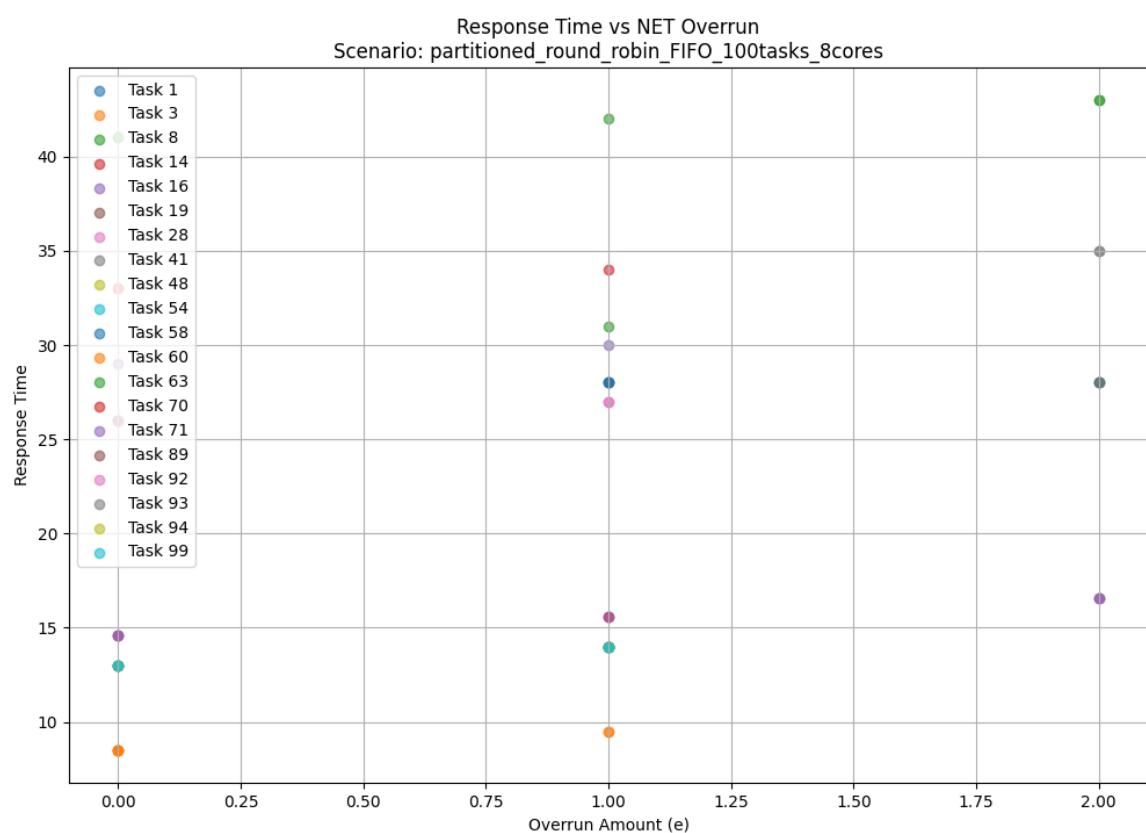
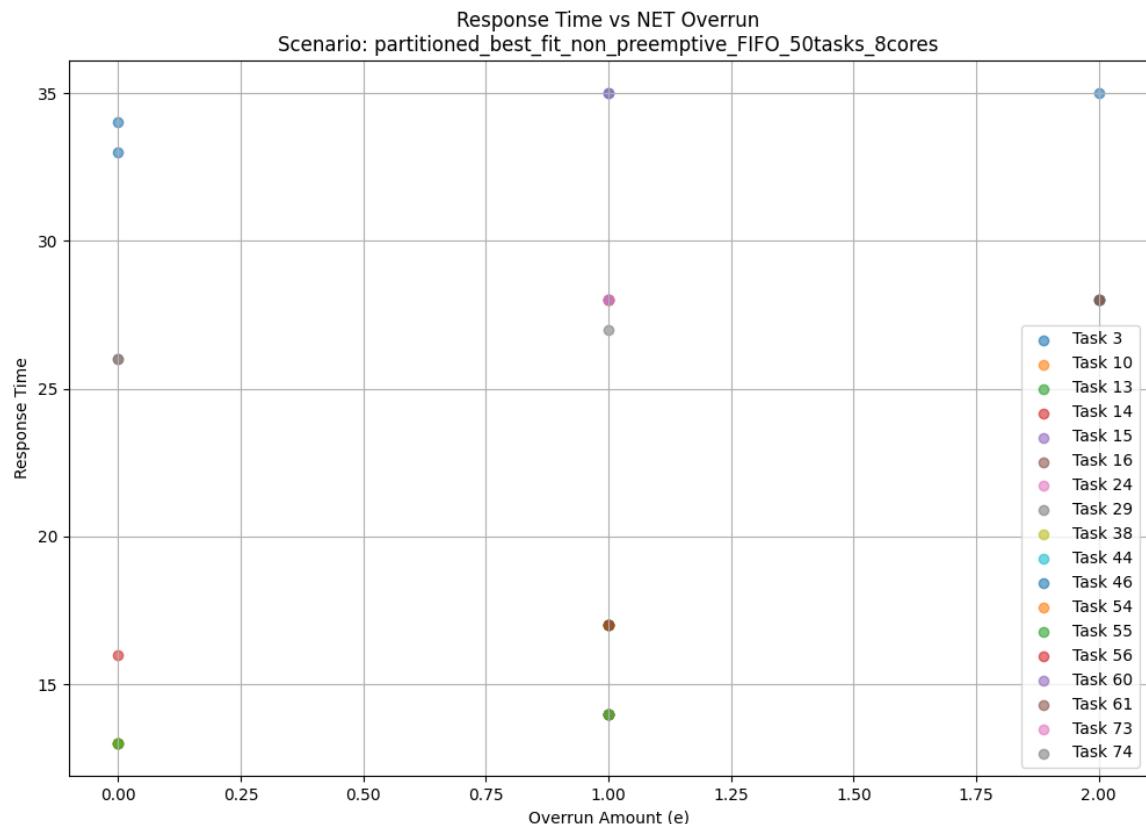
```
results > detailed_results.csv
1 scenario,task_id,period,wcet,net,utilization,miss_ratio,max_response,overrun,core
2 partitioned_best_fit_FIFO_10tasks_4cores,0,40,6,6,0.16537495404547364,0.0,38,0,0
3 partitioned_best_fit_FIFO_10tasks_4cores,1,32,2,1,0.06868552681555411,1.0,32,4,0
4 partitioned_best_fit_FIFO_10tasks_4cores,2,40,7,4,0.17560305836526502,0.0,38,9,0
5 partitioned_best_fit_FIFO_10tasks_4cores,3,50,4,4,0.08359116592962451,0.0,46,0,0
6 partitioned_best_fit_FIFO_10tasks_4cores,4,50,15,12,0.3025145384642374,0.0,42,3,0
7 partitioned_best_fit_FIFO_10tasks_4cores,5,50,14,12,0.28193335385127577,0.0,20,3,1
8 partitioned_best_fit_FIFO_10tasks_4cores,6,16,1,1,0.05,0.0,38,0,0
9 partitioned_best_fit_FIFO_10tasks_4cores,7,100,95,76,0.95,0.0,152,0,2
10 partitioned_best_fit_FIFO_10tasks_4cores,8,10,3,2,0.35385010695949076,1.0,15,7,1
11 partitioned_best_fit_FIFO_10tasks_4cores,9,80,61,56,0.7684472955690786,0.0,112,2,3
12 partitioned_worst_fit_FIFO_10tasks_4cores,0,25,3,2,0.13959856926981296,0.0,10,2,0
13 partitioned_worst_fit_FIFO_10tasks_4cores,1,8,1,1,0.2484013207612276,0.0,17,0,1
14 partitioned_worst_fit_FIFO_10tasks_4cores,2,32,3,2,0.1026769665580484,0.0,19,3,2
15 partitioned_worst_fit_FIFO_10tasks_4cores,3,100,52,42,0.5260632900309127,0.0,84,0,3
16 partitioned_worst_fit_FIFO_10tasks_4cores,4,25,16,15,0.6701396872884728,0.0,34,2,2
17 partitioned_worst_fit_FIFO_10tasks_4cores,5,16,1,1,0.11359504124976294,0.0,12,0,0
18 partitioned_worst_fit_FIFO_10tasks_4cores,6,20,1,1,0.07586643978389292,0.0,14,0,1
19 partitioned_worst_fit_FIFO_10tasks_4cores,7,20,8,5,0.4252617440068296,0.0,15,7,0
20 partitioned_worst_fit_FIFO_10tasks_4cores,8,50,16,13,0.32227104318038413,0.0,22,5,1
```

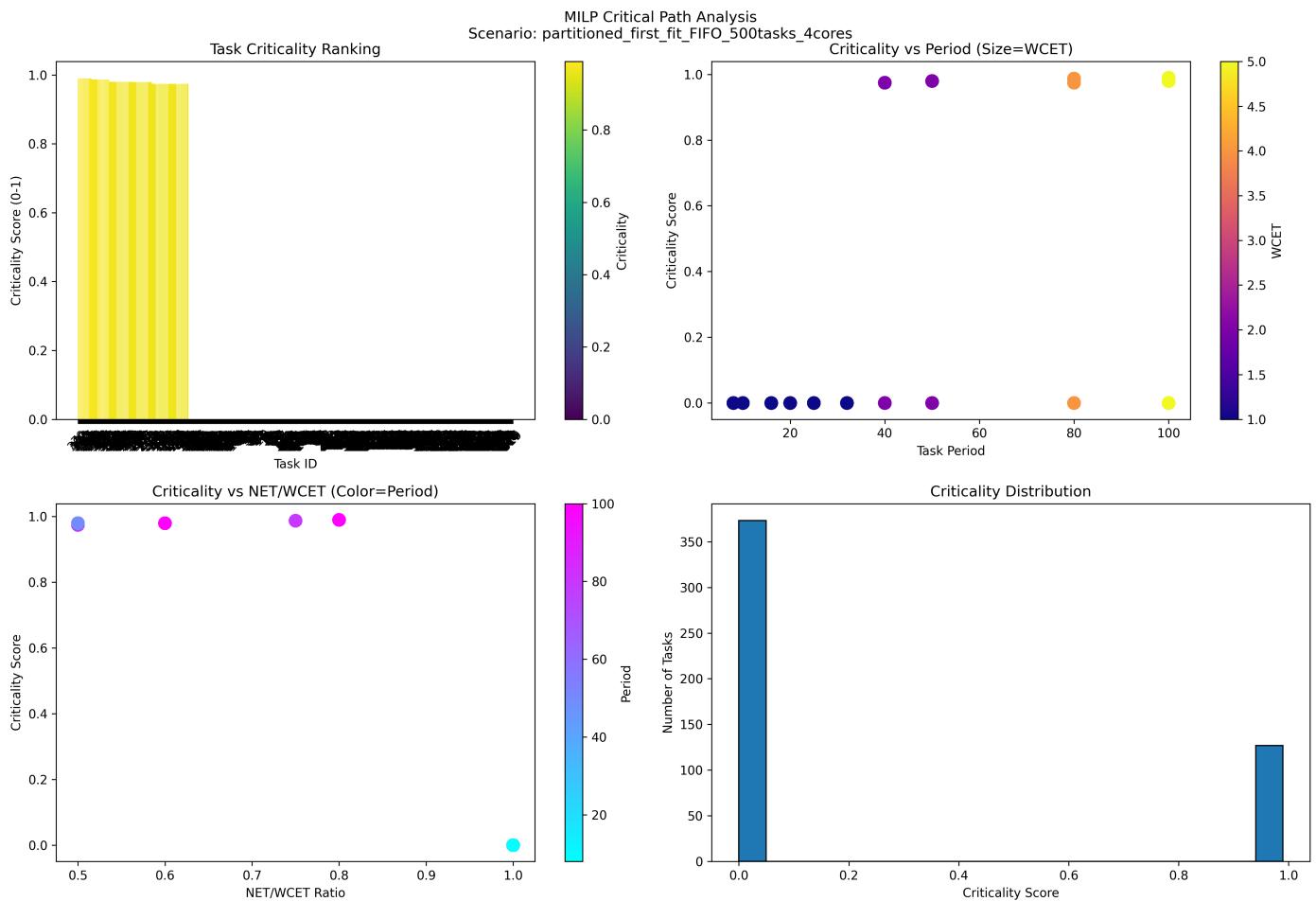
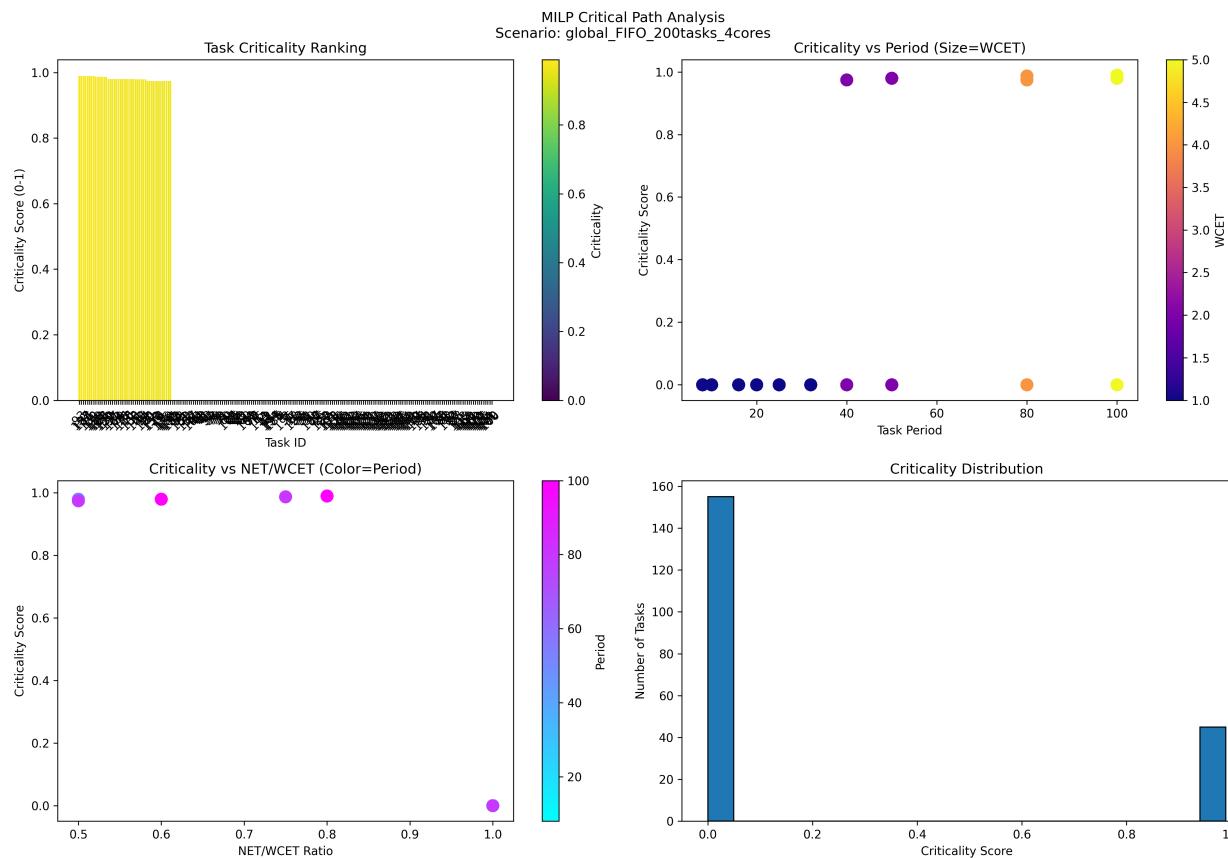
```
results > comprehensive_fifo_report.txt
1 Comprehensive FIFO Scheduling with NET Overruns Report
2 =====
3
4 1. Upper Bounds for Response Time with NET Overruns:
5 partitioned_best_fit_FIFO_10tasks_4cores:
6 - Avg Response/Period: 0.07
7 - Miss Rate: 20.00%
8 - Max Response/Period: 0.19
9 - Worst-case Response: 152 units
10
11 partitioned_worst_fit_FIFO_10tasks_4cores:
12 - Avg Response/Period: 0.03
13 - Miss Rate: 0.00%
14 - Max Response/Period: 0.10
15 - Worst-case Response: 84 units
16
17 partitioned_first_fit_FIFO_10tasks_4cores:
18 - Avg Response/Period: 0.06
19 - Miss Rate: 20.00%
20 - Max Response/Period: 0.13
21 - Worst-case Response: 80 units
22
23 partitioned_round_robin_FIFO_10tasks_4cores:
24 - Avg Response/Period: 0.03
25 - Miss Rate: 20.00%
26 - Max Response/Period: 0.08
27 - Worst-case Response: 64 units
28
29 global_FIFO_10tasks_4cores:
30 - Avg Response/Period: 0.02
31 - Miss Rate: 20.00%
32 - Max Response/Period: 0.04
33 - Worst-case Response: 31 units
34
35 partitioned_best_fit_FIFO_50tasks_4cores:
36 - Avg Response/Period: 0.08
37 - Miss Rate: 12.00%
38 - Max Response/Period: 0.28
39 - Worst-case Response: 71 units
40
41 partitioned_worst_fit_FIFO_50tasks_4cores:
42 - Avg Response/Period: 0.06
43 - Miss Rate: 14.00%
44 - Max Response/Period: 0.20
45 - Worst-case Response: 76 units
```

```
results > comprehensive_fifo_report.txt
1 Comprehensive FIFO Scheduling with NET Overruns Report
2 =====
3
4 1. Upper Bounds for Response Time with NET Overruns:
5 Task 34: Overrun=3 (NET=1, WCET=2) Response=95 (Period=40)
59 Task 11: Overrun=1 (NET=1, WCET=2) Response=127 (Period=50)
67 Task 67: Overrun=1 (NET=1, WCET=2) Response=122 (Period=50)
396
397 partitioned_first_fit_FIFO_400tasks_4cores:
398 Task 34: Overrun=3 (NET=1, WCET=2) Response=109 (Period=40)
399 Task 11: Overrun=1 (NET=1, WCET=2) Response=109 (Period=50)
400 Task 59: Overrun=1 (NET=1, WCET=2) Response=95 (Period=40)
401
402 partitioned_round_robin_FIFO_400tasks_4cores:
403 Task 46: Overrun=2 (NET=1, WCET=2) Response=90 (Period=50)
404
405 partitioned_best_fit_FIFO_500tasks_4cores:
406 Task 3: Overrun=2 (NET=1, WCET=2) Response=53 (Period=40)
407 Task 15: Overrun=2 (NET=1, WCET=2) Response=65 (Period=50)
408 Task 73: Overrun=1 (NET=1, WCET=2) Response=130 (Period=50)
409
410 partitioned_worst_fit_FIFO_500tasks_4cores:
411 Task 60: Overrun=2 (NET=1, WCET=2) Response=59 (Period=40)
412 Task 73: Overrun=1 (NET=1, WCET=2) Response=82 (Period=40)
413
414 partitioned_first_fit_FIFO_500tasks_4cores:
415 Task 67: Overrun=3 (NET=1, WCET=2) Response=54 (Period=40)
416
417 partitioned_round_robin_FIFO_500tasks_4cores:
418 Task 3: Overrun=3 (NET=1, WCET=2) Response=88 (Period=40)
419 Task 29: Overrun=2 (NET=1, WCET=2) Response=69 (Period=40)
420
421 partitioned_best_fit_FIFO_100tasks_8cores:
422 Task 30: Overrun=4 (NET=1, WCET=2) Response=29 (Period=20)
423 Task 53: Overrun=2 (NET=1, WCET=2) Response=47 (Period=32)
424
425 partitioned_first_fit_FIFO_100tasks_8cores:
426 Task 65: Overrun=6 (NET=2, WCET=3) Response=15 (Period=10)
427
428 partitioned_round_robin_FIFO_100tasks_8cores:
429 Task 16: Overrun=5 (NET=5, WCET=7) Response=63 (Period=32)
430 Task 14: Overrun=4 (NET=5, WCET=7) Response=71 (Period=32)
431 Task 48: Overrun=3 (NET=1, WCET=2) Response=56 (Period=40)
432
433 partitioned_best_fit_FIFO_100tasks_16cores:
434 Task 71: Overrun=19 (NET=21, WCET=26) Response=45 (Period=32)
435 Task 26: Overrun=8 (NET=1, WCET=2) Response=38 (Period=16)
436 Task 6: Overrun=4 (NET=1, WCET=2) Response=74 (Period=40)
437 Task 35: Overrun=4 (NET=2, WCET=4) Response=34 (Period=20)
```



تغییرات زمان پاسخ وظایف بر حسب میزان تجاوز از.NET:





بخش چهارم: بررسی با مثال

تست EDF

با توجه به این بخش، نحوه عملکرد زمان‌بندی First Deadline (Earliest Deadline) و نتایج خواسته شده را بررسی می‌کنیم. با استفاده از یک سناریوی شبیه‌سازی ساده، اجرای وظایف را مصوّر سازی کرده و معیارهای عملکردی را ارائه می‌دهیم. به طور کلی در این بخش یک سیستم تک‌هسته‌ای با سه وظیفه نمونه را با استفاده از کلاس Scheduler شبیه‌سازی می‌کند. یک نمودار گانت تولید می‌کند که نشان می‌دهد هر وظیفه در چه زمانی و برای چه مدتی روی هسته اجرا شده است. اطلاعات دقیقی در مورد زمان‌بندی هر نمونه وظیفه (Job) و خلاصه‌ای از عملکرد کلی هر وظیفه (مانند دلاین‌های از دست رفته و حداکثر زمان پاسخ) ارائه می‌دهد. بررسی می‌شود آیا در صورت قابلیت زمان‌بندی سیستم، دلاینی از دست نرفته است. همچنین یک مثال ساده برای نشان دادن نحوه عملکرد صفت اولویت (heap) برای مدیریت اولویت‌ها از آن استفاده می‌کند، ارائه می‌دهد.

در تابع `test_edf_scheduling` یک سناریوی EDF خاص را اجرا می‌کند. ابتدا یک نوع زمان‌بندی و تsekها به صورت زیر تعریف می‌شود. شبیه‌سازی برای ۶۰ واحد زمانی اجرا می‌شود. تابع `_simulate_core_edf` به طور مستقیم برای هسته فراخوانی می‌شود تا فرآیند زمان‌بندی را اجرا شود.

```
def test_edf_scheduling():
    # Create a simple test scenario with 1 core and 3 tasks
    scheduler = Scheduler(
        num_cores=1,
        scheduling_policy='EDF',
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    # Manually create tasks with different periods and WCETs
    tasks = [
        Task(id=0, period=18, wcet=3, net=2, utilization=0.3), # 3/10 = 0.3
        Task(id=1, period=15, wcet=4, net=3, utilization=0.267), # 4/15 ≈ 0.267
        Task(id=2, period=20, wcet=5, net=4, utilization=0.25) # 5/20 = 0.25
    ]

    # Allocate tasks to core (since we're testing partitioned)
    scheduler.cores[0].tasks = tasks.copy()
    for task in tasks:
        task.core_assignment = 0

    # Simulate for 60 time units (enough to see several periods)
    max_time = 60
    scheduler._simulate_core_edf(scheduler.cores[0], max_time)
```

نمودار گانت تولید شده نشان می‌دهد هر وظیفه (با یک رنگ مجزا) در چه بازه‌های زمانی روی هسته اجرا شده است. خطوط نقطه‌چین عمودی برای هر وظیفه، دلاین‌های آن را روی نمودار نشان می‌دهند. متد تأیید ویژگی‌های EDF این تابع، صحت عملکرد EDF را بر اساس اصول آن بررسی می‌کند:

- بررسی ترتیب اولویت EDF
- بررسی بهره‌برداری و از دست دادن دلاین
- تحلیل زمان پاسخ

در این شبیه‌سازی، اضافه بار (overrun) به طور تصادفی اتفاق می‌افتد. بنابراین، حتی اگر بهره‌برداری نظری کمتر از ۱۰٪ باشد، ممکن است دلاین‌ها به دلیل این اضافه بار تصادفی از دست بروند.

```

# Collect and visualize the schedule
schedule = scheduler.cores[0].schedule
task_colors = {0: 'red', 1: 'green', 2: 'blue'}

# Create a timeline visualization
plt.figure(figsize=(15, 5))

# Plot each job execution
for start, end, task_id, _ in schedule:
    plt.barh(y=0, width=end-start, left=start, color=task_colors[task_id], edgecolor='black')
    plt.text((start+end)/2, 0, f'T{task_id}', ha='center', va='center', color='white')

# Mark deadlines for each task
for task in tasks:
    deadlines = range(task.period, max_time + task.period, task.period)
    for d in deadlines:
        if d <= max_time:
            plt.axvline(x=d, color=task_colors[task.id], linestyle='--', alpha=0.5)

plt.yticks([])
plt.xlabel('Time')
plt.title('EDF Schedule Visualization')
plt.grid(True, axis='x')
plt.show()

# Print job execution details
print("\nJob Execution Details:")
print("Time | Task | Execution | Deadline")
print("-----|-----|-----|-----")
for start, end, task_id, exec_time in schedule:
    task = tasks[task_id]
    job_deadline = start - (start % task.period) + task.period
    print(f"{start:4}-{end:4} | T{task_id} | {exec_time:4}/{task.wcet:4} | {job_deadline:4}")

```

```

def verify_edf_properties(schedule, tasks, max_time):
    print("\nVerifying EDF Properties:")

    # 1. Check that jobs are scheduled in deadline order
    deadlines = []
    current_job = None
    prev_deadline = -1
    violations = 0

    for start, end, task_id, _ in schedule:
        task = tasks[task_id]
        # Calculate absolute deadline for this job
        job_release = start - (start % task.period)
        job_deadline = job_release + task.period

        if current_job is None or task_id != current_job:
            deadlines.append((job_deadline, task_id, start))
            current_job = task_id

        # Check if this deadline is earlier than previous
        if job_deadline < prev_deadline and prev_deadline != -1:
            violations += 1
            print(f"Warning: EDF violation at time {start} - T{task_id} (deadline {job_deadline}) sch")

        prev_deadline = job_deadline

    if violations == 0:
        print("1. No EDF priority violations found - jobs always scheduled in deadline order")
    else:
        print(f"1. Found {violations} EDF priority violations")

    # 2. Check that no deadlines are missed when utilization <= 1
    total_utilization = sum(task.wcet / task.period for task in tasks)
    missed_deadlines = sum(task.missed_deadlines for task in tasks)

```

تصویرسازی عملیات (visualize_heap_operations) : که در پایتون با heapq پیاده‌سازی می‌شود و برای صفات اولویت EDF ضروری است) طراحی شده است. کوچکترین عنصر همیشه در بالای آن است. خروجی نشان می‌دهد که Job ها به ترتیب دلاین صعودی (یعنی از نزدیکترین دلاین به دورترین) خارج می‌شوند، که رفتار مورد انتظار یک صفت اولویت EDF است.

```
def visualize_heap_operations():
    """Visualize how the heap works in EDF scheduling"""
    print("\nVisualizing EDF Heap Operations:")

    # Create some jobs with different deadlines
    jobs = [
        Job(Task(id=0, period=10, wcet=2, net=1, utilization=0.2), 0),
        Job(Task(id=1, period=15, wcet=3, net=2, utilization=0.2), 0),
        Job(Task(id=2, period=20, wcet=4, net=3, utilization=0.2), 0),
        Job(Task(id=0, period=10, wcet=2, net=1, utilization=0.2), 10),
        Job(Task(id=1, period=15, wcet=3, net=2, utilization=0.2), 15)
    ]

    # Manually set deadlines (normally done in Job.__init__)
    for job in jobs:
        job.deadline = job.release_time + job.task.period

    print("\nInitial Jobs (unordered):")
    for i, job in enumerate(jobs):
        print(f"Job {i}: Task {job.task.id}, Release={job.release_time}, Deadline={job.deadline}")

    # Create a heap
    heap = []
    for job in jobs:
        heapq.heappush(heap, job)
```

و مثال برای حالت چند هسته‌ای:

```
def test_multicore_edf():
    # Create a test scenario with 4 cores and 8 tasks
    print("\n==== Testing Multicore EDF Scheduling (4 cores, 8 tasks) ====")
    scheduler = Scheduler(
        num_cores=4,
        scheduling_policy='EDF',
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    # Create tasks with different characteristics
    tasks = [
        # High priority (short period) tasks
        Task(id=0, period=10, wcet=3, net=2, utilization=0.3),
        Task(id=1, period=12, wcet=2, net=1, utilization=0.17),
        Task(id=2, period=15, wcet=4, net=3, utilization=0.27),
        Task(id=3, period=20, wcet=3, net=2, utilization=0.15),

        # Medium priority tasks
        Task(id=4, period=30, wcet=6, net=4, utilization=0.2),
        Task(id=5, period=40, wcet=5, net=3, utilization=0.125),

        # Low priority (long period) tasks
        Task(id=6, period=60, wcet=8, net=5, utilization=0.13),
        Task(id=7, period=80, wcet=10, net=7, utilization=0.125)
    ]

    # Allocate tasks to cores using the specified policy
    scheduler.allocate_tasks(tasks)
```

:global برای تست

```

def test_global_edf():
    """Test global EDF scheduling"""
    print("\n== Testing Global EDF Scheduling (2 cores, 4 tasks) ===")
    scheduler = Scheduler(
        num_cores=2,
        scheduling_policy='EDF',
        allocation_policy='global',
        is_preemptive=True
    )

    tasks = [
        Task(id=0, period=10, wcet=3, net=2, utilization=0.3),
        Task(id=1, period=15, wcet=4, net=3, utilization=0.27),
        Task(id=2, period=20, wcet=5, net=4, utilization=0.25),
        Task(id=3, period=30, wcet=6, net=4, utilization=0.2)
    ]

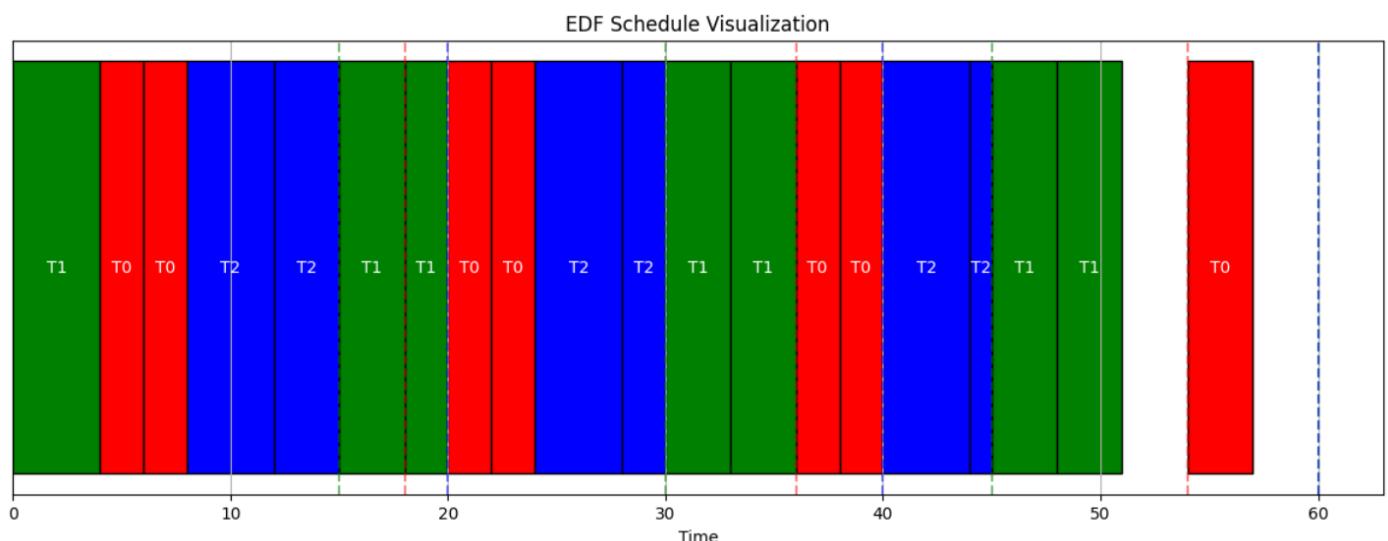
    # For global scheduling, tasks aren't assigned to specific cores initially
    scheduler.allocate_tasks(tasks)

    # Simulate for 100 time units
    max_time = 100
    scheduler.simulate(tasks, max_time)

```

نتایج بدست آمده از این تست:

در چند هسته‌ای چون مقدار بهره‌وری طبق الگوریتم UUniFast بدست آمده است در هنگامی که تعداد تسک‌ها زیاد شود مقدار نزدیک به صفر است تا مقدار بهره‌وری در کل به صورت uniform باشد.



Job Execution Details:

Time	Task	Execution	Deadline
0-4	T1	4/4	15
4-6	T0	2/3	18
6-8	T0	4/3	18
8-12	T2	4/5	20
12-15	T2	7/5	20
15-18	T1	3/4	30
18-20	T1	5/4	30
20-22	T0	2/3	36
22-24	T0	4/3	36
24-28	T2	4/5	40
28-30	T2	6/5	40
30-33	T1	3/4	45
33-36	T1	6/4	45
36-38	T0	2/3	54
38-40	T0	4/3	54
40-44	T2	4/5	60
44-45	T2	5/5	60
45-48	T1	3/4	60
48-51	T1	6/4	60
54-57	T0	3/3	72

Task Metrics:

Task	Period	WCET	NET	Jobs	Missed	Max Response
T0	18	3	2	4	0	8
T1	15	4	3	4	0	6
T2	20	5	4	3	0	15

Verifying EDF Properties:

1. No EDF priority violations found - jobs always scheduled in deadline order

2. Utilization Check: Total utilization = 0.68

- No deadlines missed as expected ($\text{utilization} \leq 1$)

3. Response Time Analysis:

T0: Max response = 8 (period = 18) ✓ Within expected bound

T1: Max response = 6 (period = 15) ✓ Within expected bound

T2: Max response = 15 (period = 20) ✓ Within expected bound

Visualizing EDF Heap Operations:

Initial Jobs (unordered):

Job 0: Task 0, Release=0, Deadline=10
Job 1: Task 1, Release=0, Deadline=15
Job 2: Task 2, Release=0, Deadline=20
Job 3: Task 0, Release=10, Deadline=20
Job 4: Task 1, Release=15, Deadline=30

Heap Order (after push operations):

This shows the order jobs would be popped based on deadlines:

Heap[0]: Task 0, Deadline=10
Heap[1]: Task 1, Deadline=15
Heap[2]: Task 2, Deadline=20
Heap[3]: Task 0, Deadline=20
Heap[4]: Task 1, Deadline=30

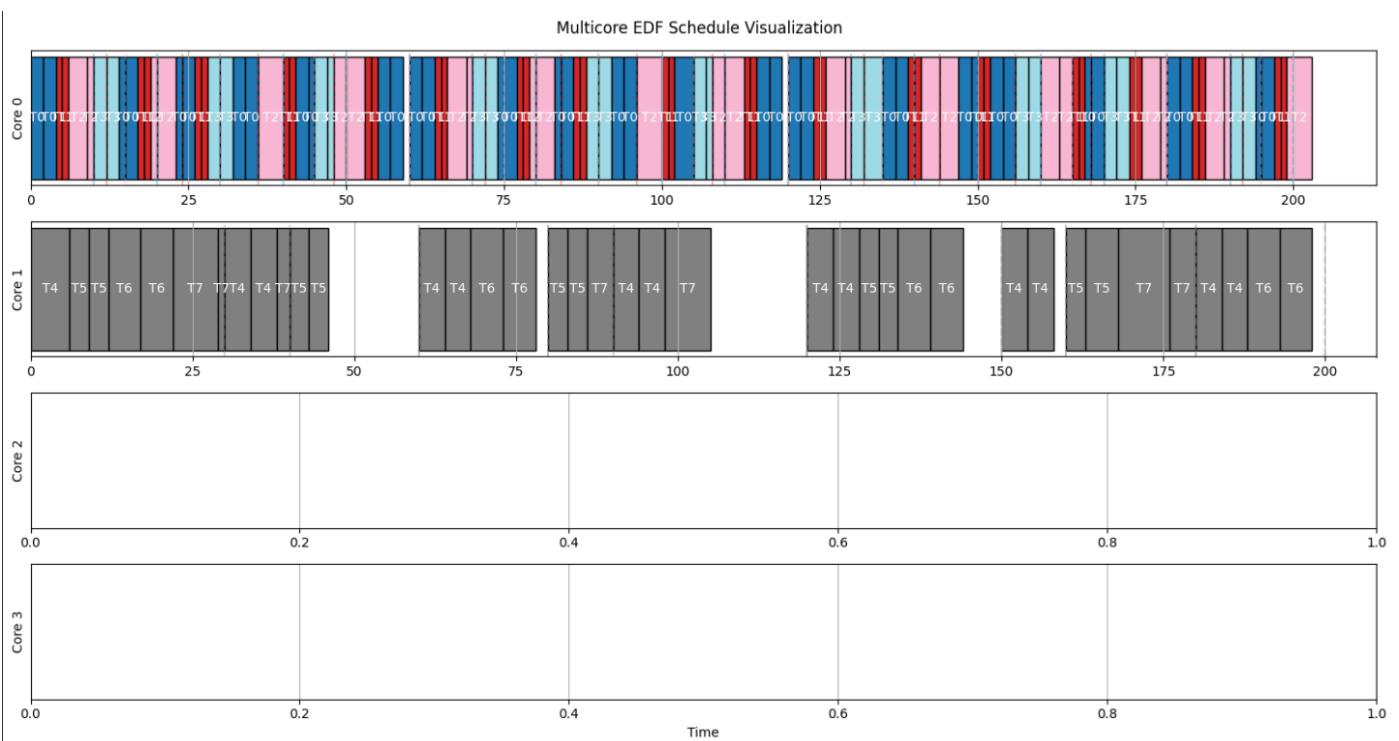
Popping jobs from heap (should be in deadline order):

Popped: Task 0, Deadline=10
Popped: Task 1, Deadline=15
Popped: Task 2, Deadline=20
Popped: Task 0, Deadline=20
Popped: Task 1, Deadline=30

==== Testing Multicore EDF Scheduling (4 cores, 8 tasks) ====

Task Allocation to Cores:

Core 0: Tasks [0, 1, 2, 3], Utilization: 0.89
Core 1: Tasks [4, 5, 6, 7], Utilization: 0.58
Core 2: Tasks [], Utilization: 0.00
Core 3: Tasks [], Utilization: 0.00



```

Core Statistics:
Core | Tasks | Planned Util | Actual Util | Missed Deadlines
----|-----|-----|-----|-----
  0 |    4 |   0.89 |   1.00 |      0
  1 |    4 |   0.58 |   0.80 |      0
  2 |    0 |   0.00 |   0.00 |      0
  3 |    0 |   0.00 |   0.00 |      0

Task Statistics (Tasks with missed deadlines):
Task | Core | Period | WCET | NET | Jobs | Missed | Max Response
----|-----|-----|-----|-----|-----|-----|-----

```

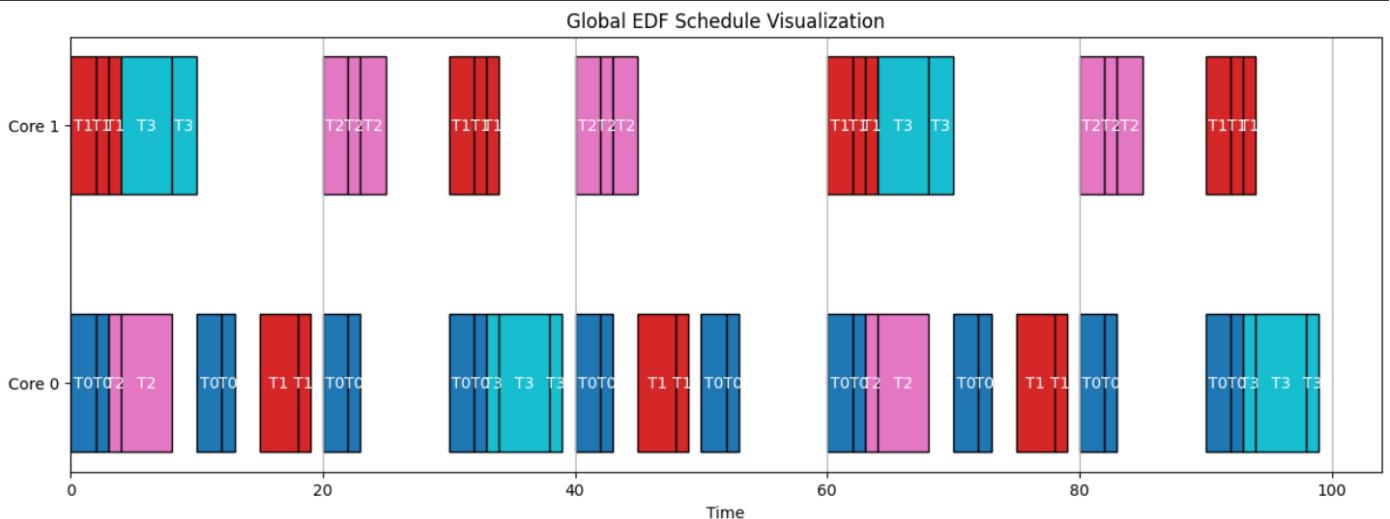
Verifying EDF Properties per Core:

Core 0:
✓ No EDF priority violations
Core utilization: 0.88
✓ Utilization ≤ 1.0 - all deadlines should be met

Core 1:
✗ 4 EDF priority violations
Core utilization: 0.58
✓ Utilization ≤ 1.0 - all deadlines should be met

Core 2:
No jobs scheduled on this core

Core 3:
No jobs scheduled on this core



```

== Global EDF Results ==
Total System Utilization: 1.02
Total Missed Deadlines: 0
✓ System within theoretical schedulability bound

Task Migration Counts:
Task 0: 19 migrations
Task 2: 12 migrations
Task 1: 17 migrations
Task 3: 9 migrations

```

تست :RM

این تست هم مانند قبلی پیاده‌سازی شده است ولی با تسک‌های زیر برای حالت تک هسته‌ای:

```
def test_rm_scheduling():
    # Create a simple test scenario with 1 core and 3 tasks
    scheduler = RMScheduler(
        num_cores=1,
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    # Manually create tasks with different periods and WCETs
    tasks = [
        Task(id=0, period=5, wcet=1, net=1, utilization=0.2),           # Highest priority (shortest period)
        Task(id=1, period=7, wcet=2, net=1, utilization=2/7),            # Medium priority
        Task(id=2, period=12, wcet=3, net=2, utilization=0.25)          # Lowest priority
    ]

    # Allocate tasks to core (since we're testing partitioned)
    scheduler.cores[0].tasks = tasks.copy()
    for task in tasks:
        task.core_assignment = 0

    # Simulate for 40 time units (enough to see several periods)
    max_time = 40
    scheduler._simulate_core_preemptive_rm(scheduler.cores[0], max_time)
```

و برای جند هسته‌ای:

```
def test_multicore_rm():
    """Test multicore RM scheduling with partitioned approach"""
    print("\n==== Testing Multicore RM Scheduling (3 cores, 9 tasks) ====")
    scheduler = RMScheduler(
        num_cores=3,
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    # Create tasks with harmonic periods (better for RM)
    tasks = [
        # Core 0 candidates (high priority)
        Task(id=0, period=5, wcet=1, net=1, utilization=0.2),
        Task(id=1, period=10, wcet=2, net=1, utilization=0.2),
        Task(id=2, period=20, wcet=3, net=2, utilization=0.15),

        # Core 1 candidates
        Task(id=3, period=6, wcet=1, net=1, utilization=0.17),
        Task(id=4, period=12, wcet=2, net=1, utilization=0.17),
        Task(id=5, period=24, wcet=3, net=2, utilization=0.125),

        # Core 2 candidates
        Task(id=6, period=7, wcet=1, net=1, utilization=0.14),
        Task(id=7, period=14, wcet=2, net=1, utilization=0.14),
        Task(id=8, period=28, wcet=3, net=2, utilization=0.11)
    ]

    # Allocate tasks using best-fit partitioning
    scheduler.allocate_tasks(tasks)
```

و برای :non-preemptive

```

def test_nonpreemptive_rm():
    """Test non-preemptive RM scheduling"""
    print("\n==== Testing Non-Preemptive RM (1 core, 3 tasks) ====")
    scheduler = RMScheduler(
        num_cores=1,
        allocation_policy='partitioned_best_fit',
        is_preemptive=False
    )

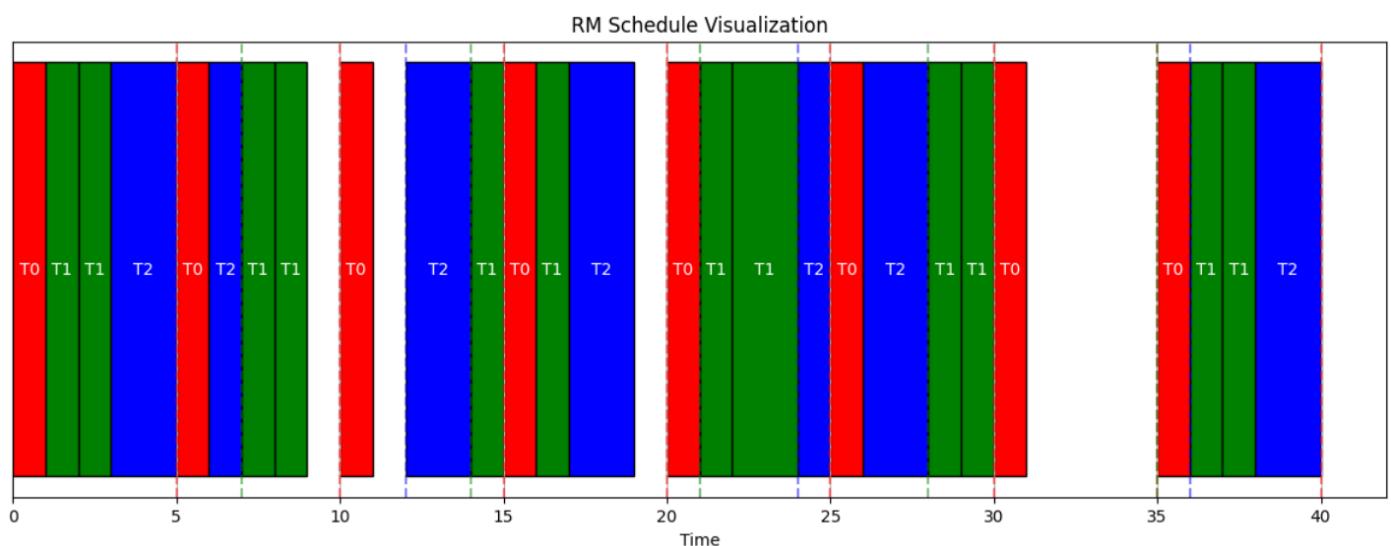
    # Create tasks with harmonic periods
    tasks = [
        Task(id=0, period=5, wcet=1, net=1, utilization=0.2),
        Task(id=1, period=10, wcet=1, net=1, utilization=0.1),
        Task(id=2, period=20, wcet=2, net=1, utilization=0.1)
    ]

    # Allocate to core
    scheduler.cores[0].tasks = tasks.copy()
    for task in tasks:
        task.core_assignment = 0

    # Simulate for 50 time units
    max_time = 50
    scheduler._simulate_core_nonpreemptive_rm(scheduler.cores[0], max_time)

```

نتایج بدست آمده از این تست:



JOB EXECUTION DETAILS:

Time	Task	Execution	Deadline
0-1	T0	1/1	5
1-2	T1	1/2	7
2-3	T1	2/2	7
3-5	T2	2/3	12
5-6	T0	1/1	10
6-7	T2	3/3	12
7-8	T1	1/2	14
8-9	T1	2/2	14
10-11	T0	1/1	15
12-14	T2	2/3	24
14-15	T1	1/2	21
15-16	T0	1/1	20
16-17	T1	2/2	21
17-19	T2	4/3	24
20-21	T0	1/1	25
21-22	T1	1/2	28
22-24	T1	3/2	28
24-25	T2	1/3	36
25-26	T0	1/1	30
26-28	T2	3/3	36
28-29	T1	1/2	35
29-30	T1	2/2	35
30-31	T0	1/1	35
35-36	T0	1/1	40
36-37	T1	1/2	42
37-38	T1	2/2	42
38-40	T2	2/3	48

Task Metrics:

Task	Period	WCET	NET	Jobs	Missed	Max Response
T0	5	1	1	8	0	1
T1	7	2	1	6	0	3
T2	12	3	2	4	0	7

Verifying RM Properties:

Warning: Invalid preemption at time 1 - T1 (period 7) preempted T0 (period 5)
 Warning: Invalid preemption at time 3 - T2 (period 12) preempted T1 (period 7)
 Warning: Invalid preemption at time 6 - T2 (period 12) preempted T0 (period 5)
 Warning: Invalid preemption at time 12 - T2 (period 12) preempted T0 (period 5)
 Warning: Invalid preemption at time 16 - T1 (period 7) preempted T0 (period 5)
 Warning: Invalid preemption at time 17 - T2 (period 12) preempted T1 (period 7)
 Warning: Invalid preemption at time 21 - T1 (period 7) preempted T0 (period 5)
 Warning: Invalid preemption at time 24 - T2 (period 12) preempted T1 (period 7)
 Warning: Invalid preemption at time 26 - T2 (period 12) preempted T0 (period 5)
 Warning: Invalid preemption at time 36 - T1 (period 7) preempted T0 (period 5)
 Warning: Invalid preemption at time 38 - T2 (period 12) preempted T1 (period 7)

1. Preemption Analysis:

T0 preempted T2 3 times (correct priority order)
 T1 preempted T2 3 times (correct priority order)
 T0 preempted T1 3 times (correct priority order)

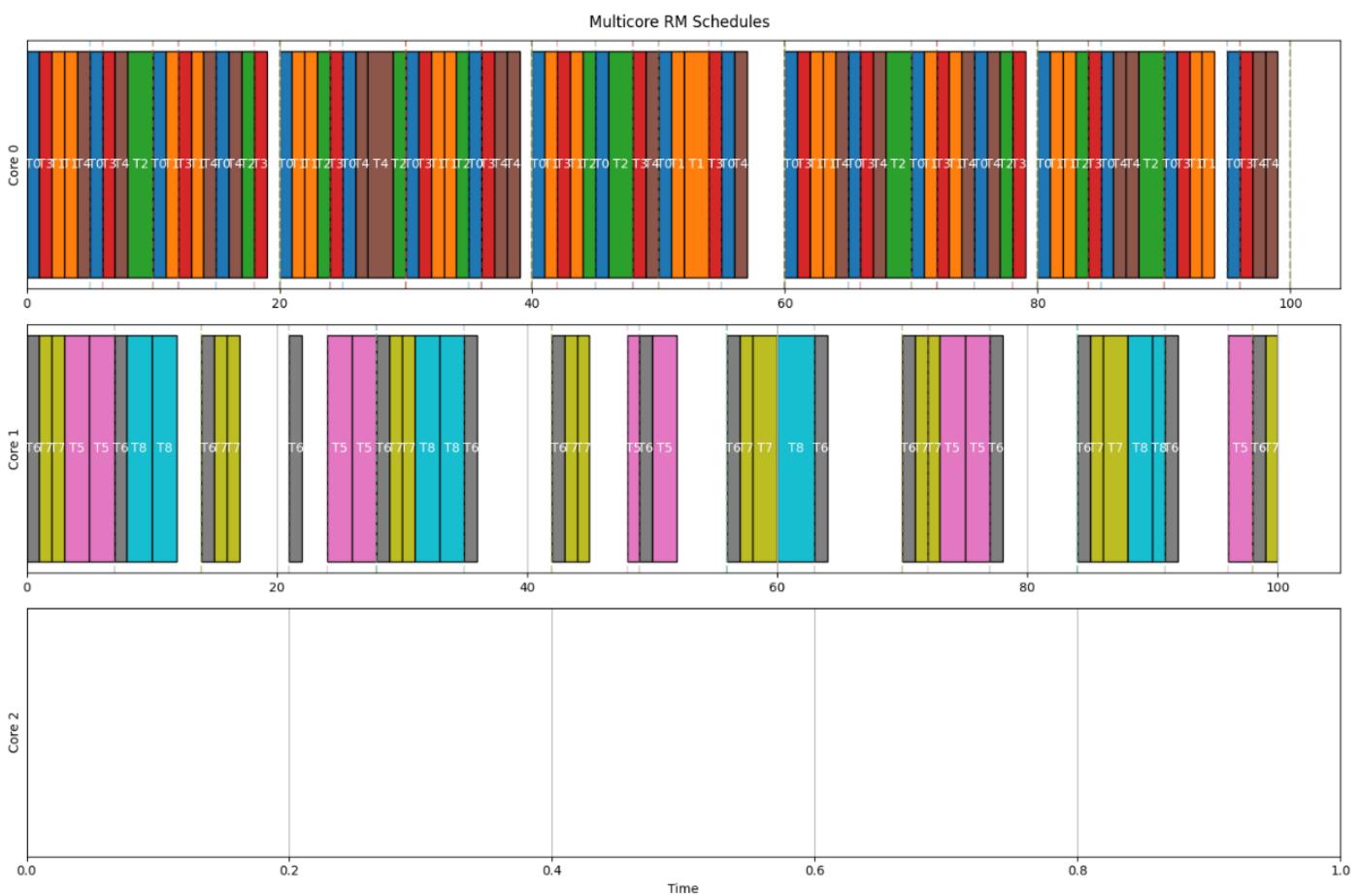
2. Liu & Layland Schedulability Test:

Total utilization: 0.736
 Liu & Layland bound for 3 tasks: 0.780
 ✓ System guaranteed schedulable under RM ($U \leq n(2^{(1/n)-1})$)

3. Response Time Analysis:
T0: Period=5, Max response=1 ✓ Within expected bound
T1: Period=7, Max response=3 ✓ Within expected bound
T2: Period=12, Max response=7 ✓ Within expected bound

==== Testing Multicore RM Scheduling (3 cores, 9 tasks) ===

Task Allocation:
Core 0: Tasks [0, 3, 1, 4, 2], Utilization: 0.89
Core 1: Tasks [6, 7, 5, 8], Utilization: 0.52
Core 2: Tasks [], Utilization: 0.00



==== Multicore RM Results ===

Core Statistics:
Core	Tasks	Utilization	Missed Deadlines
0 | 5 | 0.89 | 0
1 | 4 | 0.52 | 0
2 | 0 | 0.00 | 0

Tasks with Missed Deadlines:
Task	Core	Period	WCET	Jobs	Missed	Response

RM Property Verification per Core:

```
RM Property Verification per Core:
```

Core 0:

Tasks: [0, 1, 2, 3, 4]

Utilization: 0.890

Liu & Layland bound: 0.743

✗ Fails LL test (may still be schedulable)

Note: System is empirically schedulable despite failing LL test

Core 1:

Tasks: [5, 6, 7, 8]

Utilization: 0.515

Liu & Layland bound: 0.757

✓ Passes LL schedulability test

Core 2:

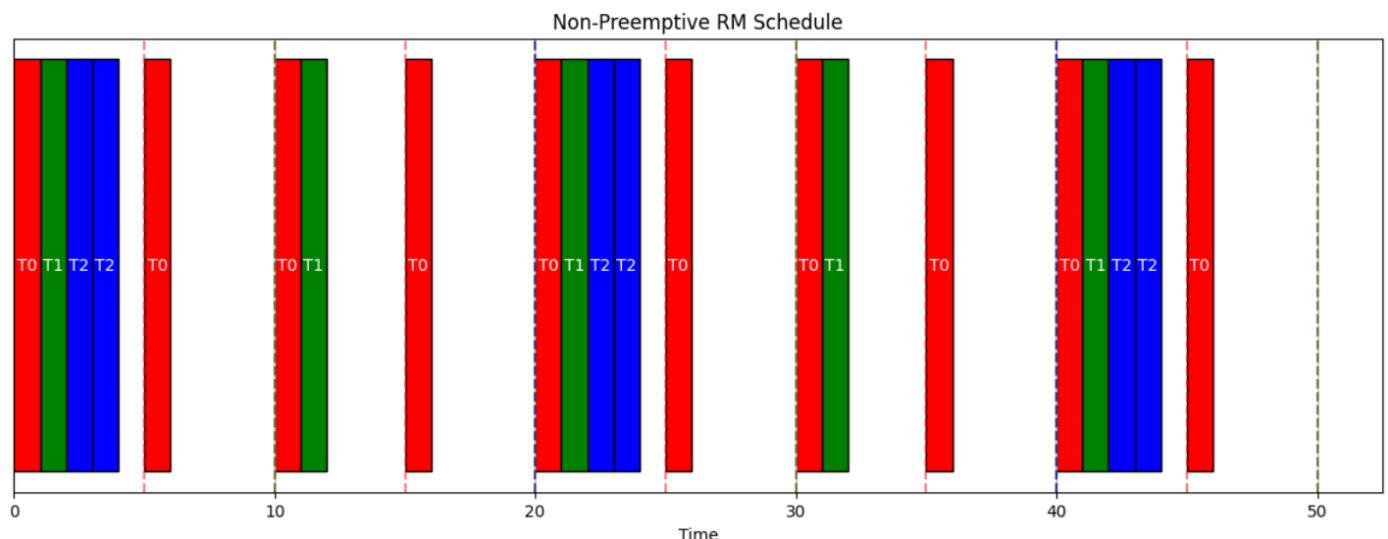
Tasks: []

Utilization: 0.000

Liu & Layland bound: 0.000

✓ Passes LL schedulability test

```
== Testing Non-Preemptive RM (1 core, 3 tasks) ==
```



```
Non-Preemptive RM Results:
```

Task	Period	WCET	Jobs	Missed	Max Response
T 0	5	1	10	0	1
T 1	10	1	5	0	2
T 2	20	2	3	0	4

Priority Inversion Check:

Priority inversion at time 3-4: T2 (period 20) ran before T0 (period 5)

Priority inversion at time 11-12: T1 (period 10) ran before T0 (period 5)

Priority inversion at time 23-24: T2 (period 20) ran before T0 (period 5)

Priority inversion at time 31-32: T1 (period 10) ran before T0 (period 5)

Priority inversion at time 43-44: T2 (period 20) ran before T0 (period 5)

تست :FIFO

این تست هم مانند قبلی پیاده سازی شده است تا این شکل تعریف شده اند

```
def test_fifo_scheduling():
    # Create a simple test scenario with 1 core and 3 tasks
    scheduler = Scheduler(
        num_cores=1,
        scheduling_policy='FIFO',
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    # Manually create tasks with different periods and WCETs
    tasks = [
        Task(id=0, period=18, wcet=3, net=2, utilization=0.3),
        Task(id=1, period=15, wcet=4, net=3, utilization=4/15),
        Task(id=2, period=20, wcet=5, net=4, utilization=0.25)
    ]

    # Allocate tasks to core (since we're testing partitioned)
    scheduler.cores[0].tasks = tasks.copy()
    for task in tasks:
        task.core_assignment = 0

    # Simulate for 60 time units (enough to see several periods)
    max_time = 60
    if scheduler.is_preemptive:
        scheduler._simulate_partitioned(tasks, max_time) # Changed from _simulate_core_pfifo
    else:
        scheduler._simulate_partitioned(tasks, max_time)
```

```
def visualize_heap_operations():
    """Visualize how the heap works in FIFO scheduling"""
    print("\nVisualizing FIFO Heap Operations:")

    # Create some jobs with different release times
    jobs = [
        Job(Task(id=0, period=10, wcet=2, net=1, utilization=0.2), 0),
        Job(Task(id=1, period=15, wcet=3, net=2, utilization=0.2), 0),
        Job(Task(id=2, period=20, wcet=4, net=3, utilization=0.2), 0),
        Job(Task(id=0, period=10, wcet=2, net=1, utilization=0.2), 10),
        Job(Task(id=1, period=15, wcet=3, net=2, utilization=0.2), 15)
    ]

    for job in jobs:
        job.deadline = job.release_time + job.task.period
```

```

def test_multicore_fifo():
    print("\n==== Testing Multicore FIFO Scheduling (4 cores, 8 tasks) ====")
    scheduler = Scheduler(
        num_cores=4,
        scheduling_policy='FIFO',
        allocation_policy='partitioned_best_fit',
        is_preemptive=True
    )

    tasks = [
        Task(id=0, period=10, wcet=3, net=2, utilization=0.3),
        Task(id=1, period=12, wcet=2, net=1, utilization=0.17),
        Task(id=2, period=15, wcet=4, net=3, utilization=0.27),
        Task(id=3, period=20, wcet=3, net=2, utilization=0.15),
        Task(id=4, period=30, wcet=6, net=4, utilization=0.2),
        Task(id=5, period=40, wcet=5, net=3, utilization=0.125),
        Task(id=6, period=60, wcet=8, net=5, utilization=0.13),
        Task(id=7, period=80, wcet=10, net=7, utilization=0.125)
    ]
    scheduler.allocate_tasks(tasks)

```

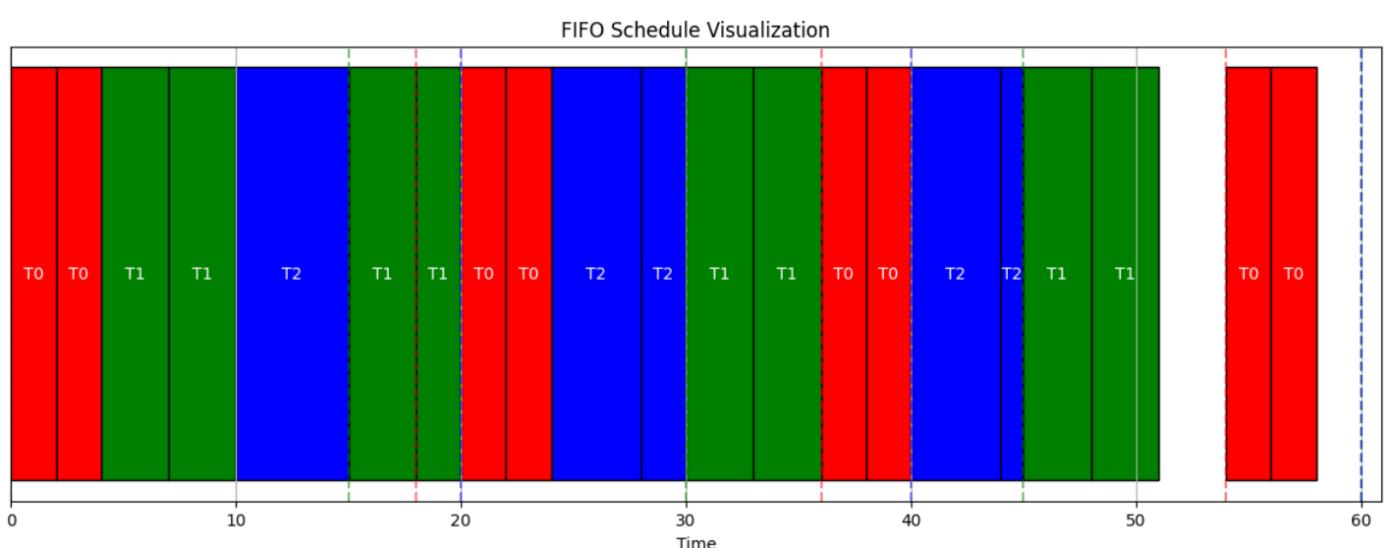
```

def test_global_fifo():
    print("\n==== Testing Global FIFO Scheduling (2 cores, 4 tasks) ====")
    scheduler = Scheduler(
        num_cores=2,
        scheduling_policy='FIFO',
        allocation_policy='global',
        is_preemptive=True
    )

    tasks = [
        Task(id=0, period=10, wcet=3, net=2, utilization=0.3),
        Task(id=1, period=15, wcet=4, net=3, utilization=0.27),
        Task(id=2, period=20, wcet=5, net=4, utilization=0.25),
        Task(id=3, period=30, wcet=6, net=4, utilization=0.2)
    ]
    scheduler.allocate_tasks(tasks)
    max_time = 100
    scheduler.simulate(tasks, max_time)

```

نتایج بدست آمده از این تست:



Time	Task	Execution	Release	Deadline
0-2	T0	2/3	0	18
2-4	T0	4/3	0	18
4-7	T1	3/4	0	15
7-10	T1	6/4	0	15
10-15	T2	5/5	0	20
15-18	T1	3/4	15	30
18-20	T1	5/4	15	30
20-22	T0	2/3	18	36
22-24	T0	4/3	18	36
24-28	T2	4/5	20	40
28-30	T2	6/5	20	40
30-33	T1	3/4	30	45
33-36	T1	6/4	30	45
36-38	T0	2/3	36	54
38-40	T0	4/3	36	54
40-44	T2	4/5	40	60
44-45	T2	5/5	40	60
45-48	T1	3/4	45	60
48-51	T1	6/4	45	60
54-56	T0	2/3	54	72
56-58	T0	4/3	54	72

Task Metrics:						
Task	Period	WCET	NET	Jobs	Missed	Max Response
T0	18	3	2	4	0	6
T1	15	4	3	4	0	10
T2	20	5	4	3	0	15

Verifying FIFO Properties:

- No FIFO priority violations found - jobs always scheduled in release order
- Utilization Check: Total utilization = 0.68
 - No deadlines missed as expected (utilization ≤ 1)

3. Response Time Analysis:

T0: Max response = 6 (expected ≤ 8) ✓ Within expected bound
 T1: Max response = 10 (expected ≤ 4) ✗ Exceeds expected bound!
 T2: Max response = 15 (expected ≤ 14) ✗ Exceeds expected bound!

Visualizing FIFO Heap Operations:

Initial Jobs (unordered):

```

Job 0: Task 0, Release=0, Deadline=10
Job 1: Task 1, Release=0, Deadline=15
Job 2: Task 2, Release=0, Deadline=20
Job 3: Task 0, Release=10, Deadline=20
Job 4: Task 1, Release=15, Deadline=30
  
```

Heap Order (after push operations):

This shows the order jobs would be popped based on release times:

```

Heap[0]: Task 0, Release=0
Heap[1]: Task 1, Release=0
Heap[2]: Task 2, Release=0
Heap[3]: Task 0, Release=10
Heap[4]: Task 1, Release=15
  
```

Popping jobs from heap (should be in release time order):

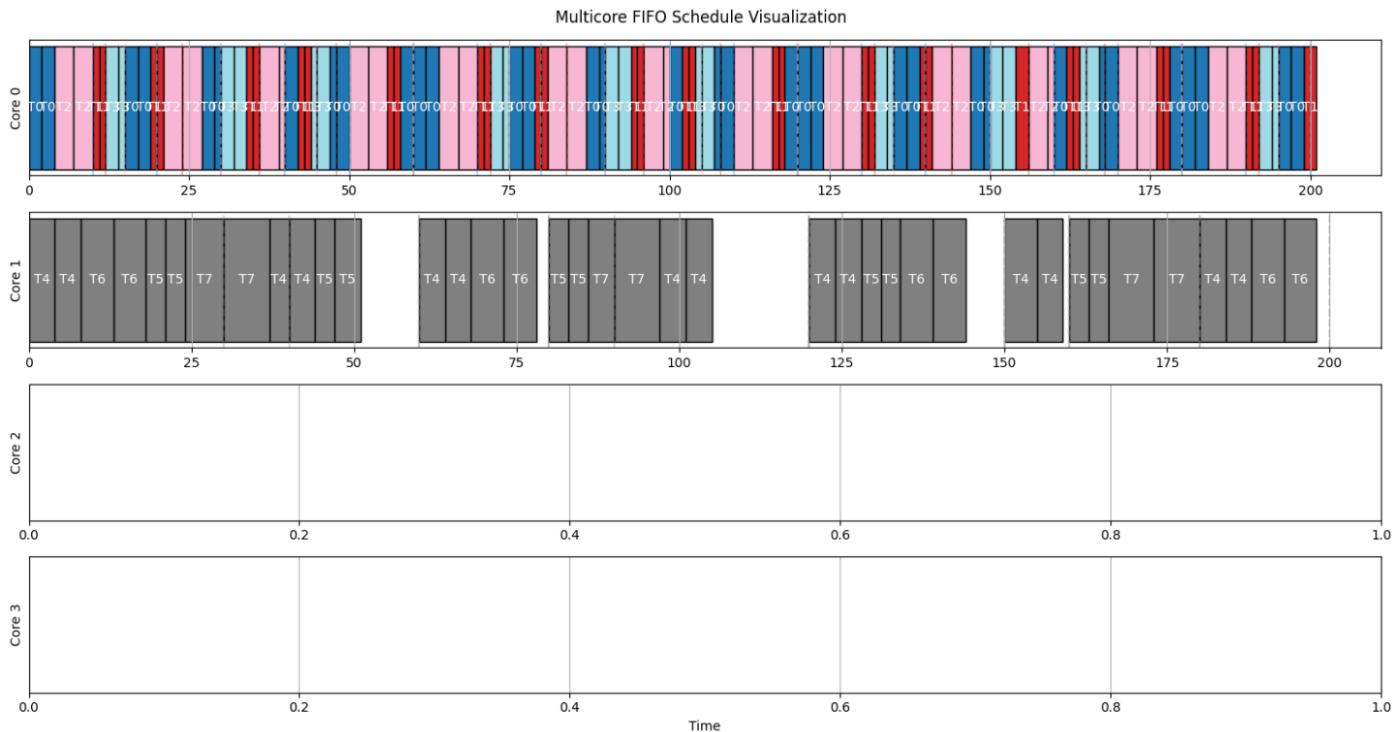
```

Popped: Task 0, Release=0
Popped: Task 2, Release=0
Popped: Task 1, Release=0
Popped: Task 0, Release=10
Popped: Task 1, Release=15
  
```

```
== Testing Multicore FIFO Scheduling (4 cores, 8 tasks) ==
```

Task Allocation to Cores:

Core 0: Tasks [0, 1, 2, 3], Utilization: 0.89
Core 1: Tasks [4, 5, 6, 7], Utilization: 0.58
Core 2: Tasks [], Utilization: 0.00
Core 3: Tasks [], Utilization: 0.00



```
== Multicore Scheduling Results ==
```

Core Statistics:

Core	Tasks	Planned Util	Actual Util	Missed Deadlines
0	4	0.89	1.00	6
1	4	0.58	0.82	0
2	0	0.00	0.00	0
3	0	0.00	0.00	0

Task Statistics (Tasks with missed deadlines):

Task	Core	Period	WCET	NET	Jobs	Missed	Max Response	
T0	0	0	10	3	2	20	6	10

Verifying FIFO Properties per Core:

Core 0:

X 3 FIFO priority violations

Core utilization: 0.88

✓ Utilization ≤ 1.0 - all deadlines should be met

Core 1:

✓ No FIFO priority violations

Core utilization: 0.58

✓ Utilization ≤ 1.0 - all deadlines should be met

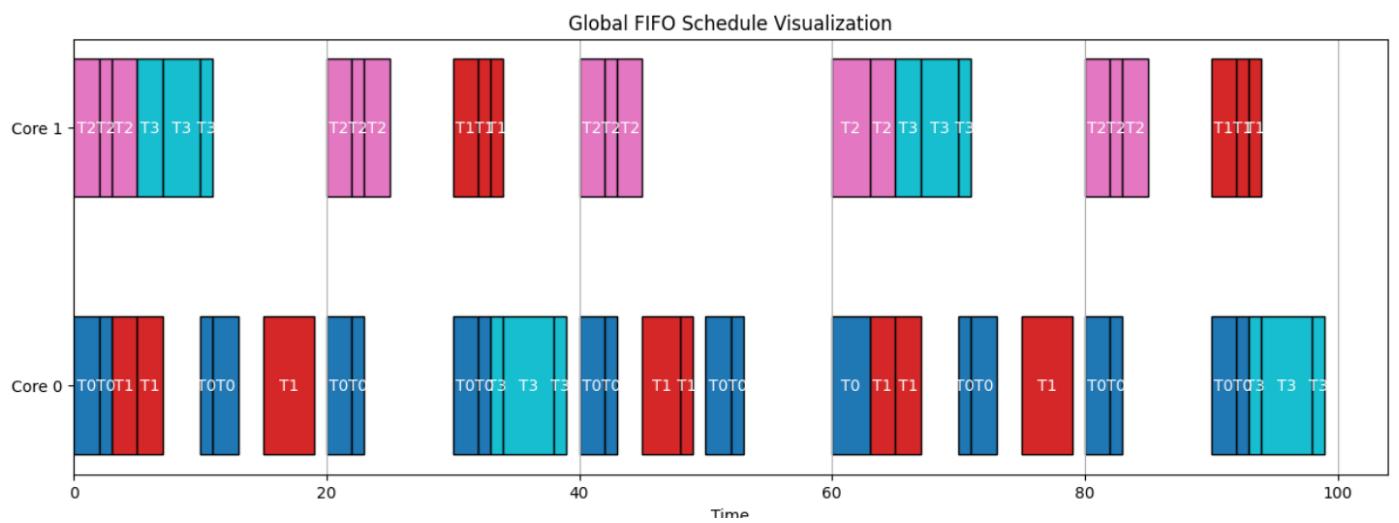
Core 2:

No jobs scheduled on this core

Core 3:

No jobs scheduled on this core

```
== Testing Global FIFO Scheduling (2 cores, 4 tasks) ==
```



```
==== Global FIFO Results ====
Total System Utilization: 1.02
Total Missed Deadlines: 0
✓ System within theoretical schedulability bound
```