



نام و نام خانوادگی:
محمدمبین تیمورپور
۴۰۲۲۱۲۹

نام درس: سیستم های عامل
مدرس: دکتر اله بخش
۱۴۰۳-۱۴۰۴ مهر

داکیومنتیشن پروژه سیستم عامل

هدف پروژه

پیاده سازی یک پلتفرم شبیه سازی سیستم های مدیریت زمانبندی منابع با دیزاین های متفاوت و با استفاده از الگوریتم های زمانبندی متفاوت.

main.py

```

1  class Task:
2      def __init__(self, name, runtime, r1, r2, entry_time, **kwargs):
3          self.name = name
4          self.original_runtime = runtime
5          self.remaining_time = runtime
6          self.r1 = r1
7          self.r2 = r2
8          self.state = 'waiting'
9          self.start_time = None
10         self.end_time = None
11         self.waiting_time = 0
12         self.subsystem = None
13         self.entry_time = entry_time
14         self.failed = False
15         self.failure_reason = None
16         for key, value in kwargs.items():
17             setattr(self, key, value)
18
19
20     def __repr__(self):
21         return f'{self.name} ({self.remaining_time})'


```



```

1  class Core(threading.Thread):
2      def __init__(self, subsystem, core_id):
3          super().__init__(daemon=True)
4          self.subsystem = subsystem
5          self.core_id = core_id
6          self.ready_queue = Queue()
7          self.current_task = None
8          self.lock = threading.Lock()
9          self.processing_event = threading.Event()
10         self.task_completed = threading.Event() # New event for task completion
11         self.should_run = True
12         self.status = 'idle'
13
14     def run(self):
15         while self.should_run:
16             try:
17                 # Only process if we have a task
18                 if self.current_task:
19                     self.current_task.running()
20                     self.processing_event.wait()
21                     self.processing_event.clear()
22                     self.current_task.remaining_time -= 1
23                     self.task_completed.set() # Signal task processing done
24                 else:
25                     self.status = 'idle'
26                     # Signal immediately if idle
27                     self.task_completed.set()
28                     time.sleep(0.1)
29             except Exception as e:
30                 print(f'Core {self.core_id} error: {str(e)}')


```

:Task

این کلاس نماینده یک وظیفه (Task) در سیستم است. هر نمونه از این کلاس شامل تمامی مشخصات و وضعیت فعلی یک تاسک می‌باشد. پارامترهای مهم شامل نام تاسک، زمان اجرای کل، منابع مورد نیاز (R1 و R2)، زمان ورود به سیستم، و وضعیت فعلی (مانند انتظار، آماده، در حال اجرا، تکمیل شده) است. ویژگی‌های اضافی مانند subsystem نشان می‌دهد تاسک به کدام زیرسیستم تعلق دارد، و failure_reason و failed برای مداردهی خطاهای استفاده می‌شوند. متod __init__ تمامی این مقادیر را مقداردهی اولیه می‌کند و kwargs** امکان افزودن ویژگی‌های خاص برای زیرسیستم‌های مختلف را فراهم می‌کند.

:Core

این کلاس که از threading.Thread ارث بری می‌کند، نماینده یک هسته پردازشی در سیستم است. هر هسته دارای یک صف آماده (ready_queue) برای تاسک‌های منتظر اجرا، یک تاسک جاری (current_task)، و مکانیزم‌های هماهنگی مانند lock برای مدیریت دسترسی همزمان و processing_event برای کنترل فرآیند اجرا است. متod run حلقه اصلی اجرا را تشکیل می‌دهد که به صورت مداوم بررسی می‌کند آیا تاسکی برای پردازش وجود دارد یا خیر. در صورت وجود تاسک، زمان باقیمانده آن کاهش می‌یابد و رویداد task_completed برای هماهنگی با زیرسیستم فعال می‌شود. این حلقه با وقفه‌های ۰.۱ ثانیه‌ای اجرا می‌شود تا از مصرف بیش از حد منابع جلوگیری کند.

:Subsystem

این کلاس پایه برای تمامی زیرسیستم‌ها است و شامل ویژگی‌های مشترک مانند منابع اختصاص یافته (R1 و R2)، لیست هسته‌های پردازشی (cores)، و صفحه‌ای مدیریت تاسک (process_event) می‌باشد. متod run حلقه اصلی پردازش را تشکیل می‌دهد که با فعال شدن process_event توسط سیستم اصلی، فرآیند پردازش تاسک‌ها در زمان جاری آغاز می‌شود. پس از تمام پردازش، رویداد cycle_complete برای هماهنگی با سیستم اصلی فعال می‌شود. متod process که در کلاس‌های فرزند پیاده‌سازی می‌شود، مسئولیت منطق خاص هر زیرسیستم (مانند زمان‌بندی، تخصیص منابع) را بر عهده دارد.

زیرسیستم ها:

SS1: این زیرسیستم با سه هسته پردازشی و الگوریتم Round-Robin کار می‌کند. متد load_balance تسكی‌ها را بین هسته‌ها به صورت متوازن توزیع می‌کند. اگر ترسکی در حین اجرا با احتمال $\frac{1}{3}$ قطع شود، به صفت انتظار بازگردانده می‌شود. این زیرسیستم از پیش‌فرض initial_core در تسكی‌ها برای تخصیص اولیه استفاده می‌کند اما ممکن است در حین اجرا تسكی‌ها جایجا شوند.

SS2: این زیرسیستم از دو هسته پردازشی و الگوریتم کوتاهترین زمان باقیمانده (SRTF) استفاده می‌کند. تسک‌ها در یک صف مرکزی (ready queue) اول مذبیریت می‌شوند و در هر لحظه، تسک با کمترین زمان باقیمانده برای اجرا انتخاب می‌شود. اگر در حین اجرا تسک کوتاهتری وارد صف شود، تسک فعلی متوقف شده و تسک جدید اجرا می‌شود (بیش گیری).

SS3: این زیرسیستم تک هسته‌ای امکان قرض‌گیری منابع از سایر زیرسیستم‌ها را فراهم می‌کند. در متدهای process، اگر منابع داخلی کافی نباشد، سیستم به صورت پویا از زیرسیستم‌های دیگر درخواست منابع می‌کند. منابع قرض‌گرفته شده باید تا پایان دوره (period) تسلیم شوند. این زیرسیستم از الگوریتم زمان‌بندی نرخ یکواخت (Rate Monotonic) برای مدیریت تسلک‌های دوره‌ای استفاده می‌کند.

SS4: این زیرسیستم دو هسته‌ای مدیریت وابستگی بین تسک‌ها را بر عهده دارد. هر تسک می‌تواند لیستی از تسک‌های پیش‌نیاز داشته باشد. سیستم به صورت خودکار برسی می‌کند که آیا تمام پیش‌نیازها با موققبت تکمیل شده‌اند یا خیر. اگر هر یک از پیش‌نیازها شکست بخورند، تسک وابسته به طور خودکار لغو می‌شود. این زیرسیستم از صف آماده (ready queue) برای مدیریت تسک‌های آماده احراستفاده می‌کند.

1

1

1

1

SS1

SS2

SS1

```
1 def process(self, current_time):
2     with self.resource_lock:
3         # Process pending tasks
4         pending_to_remove = []
5         for task in list(self.pending_tasks):
6             if task.entry_time == current_time:
7                 task.core = (task.initial_core - 1) % len(self.cores)
8                 if task.r1 <= self.available_r1 and task.r2 <= self.available_r2:
9                     self.available_r1 -= task.r1
10                    self.available_r2 -= task.r2
11                    task.state = 'ready'
12                    self.cores[task.core].ready_queue.append(task)
13                    pending_to_remove.append(task)
14                else:
15                    self.waiting_queue.append(task)
16                    pending_to_remove.append(task)
17
18        for task in pending_to_remove:
19            self.pending_tasks.remove(task)
20
21        # Check waiting queue
22        waiting_to_ready = []
23        for task in list(self.waiting_queue):
24            if task.r1 <= self.available_r1 and task.r2 <= self.available_r2:
25                task.core = (task.initial_core - 1) % len(self.cores)
26                self.cores[task.core].ready_queue.append(task)
27                self.available_r1 -= task.r1
28                self.available_r2 -= task.r2
29                task.state = 'ready'
30                waiting_to_ready.append(task)
31
32        for task in waiting_to_ready:
33            self.waiting_queue.remove(task)
34
35        # Process cores
36        for core in self.cores:
37            if core.current_task:
38                core.current_task.remaining_time -= 1
39                if core.current_task.remaining_time == 0:
40                    # Task completed
41                    self.available_r1 += core.current_task.r1
42                    self.available_r2 += core.current_task.r2
43                    core.current_task.end_time = current_time
44                    core.current_task.state = 'completed'
45                    self.completed_tasks.append(core.current_task)
46                    core.current_task = None
47                    core.status = 'idle'
48                    core.processing_event.clear()
49
50                    if random.random() < 0.3: # 30% chance to preempt
51                        self.waiting_queue.append(core.current_task)
52                        self.available_r1 -= core.current_task.r1
53                        self.available_r2 -= core.current_task.r2
54                        core.current_task = None
55                        core.status = 'idle'
56                        core.processing_event.clear()
57
58                if not core.current_task and core.ready_queue:
59                    core.current_task = core.ready_queue.pop(0)
60                    core.current_task.start_time = current_time
61                    core.status = 'running'
62                    core.processing_event.set()
63
64
```

```

1 def process(self, current_time):
2     with self.resource_lock:
3         # Return any borrowed resources that are past deadline
4         borrowed_to_remove = []
5         for borrowed in self.borrowed_resources:
6             if current_time >= borrowed['deadline']:
7                 for ss, r1, r2 in borrowed['sources']:
8                     with ss.resource_lock:
9                         ss.available_r1 += r1
10                        ss.available_r2 += r2
11                        self.borrowed_r1 -= r1
12                        self.borrowed_r2 -= r2
13                     borrowed_to_remove.append(borrowed)
14         for borrowed in borrowed_to_remove:
15             self.borrowed_resources.remove(borrowed)
16
17     # Process pending tasks
18     pending_to_remove = []
19     for task in list(self.pending_tasks):
20         if task.entry_time <= current_time:
21             self.waiting_queue.append(task)
22             pending_to_remove.append(task)
23
24     for task in pending_to_remove:
25         self.pending_tasks.remove(task)
26
27     # Sort waiting queue by period (Rate Monotonic)
28     waiting_list = list(self.waiting_queue)
29     waiting_list.sort(key=lambda t: t.period)
30
31     core = self.cores[0] # Single core system
32     if not core.current_task:
33         for task in waiting_list:
34             borrowed_r1 = 0 # Initialize borrowing variables
35             borrowed_r2 = 0
36             sources = []
37
38             total_needed_r1 = max(0, task.r1 - self.available_r1)
39             total_needed_r2 = max(0, task.r2 - self.available_r2)
40
41             if total_needed_r1 > 0 or total_needed_r2 > 0:
42                 # Try to borrow resources
43                 for ss in self.main_system.subsystems:
44                     if ss != self:
45                         with ss.resource_lock:
46                             if borrowed_r1 < total_needed_r1:
47                                 borrow_r1 = min(ss.available_r1, total_needed_r1 - borrowed_r1)
48                             if borrow_r1 > 0:
49                                 ss.available_r1 -= borrow_r1
50                                 borrowed_r1 += borrow_r1
51                                 sources.append((ss, borrow_r1, 0))
52
53                             if borrowed_r2 < total_needed_r2:
54                                 borrow_r2 = min(ss.available_r2, total_needed_r2 - borrowed_r2)
55                                 if borrow_r2 > 0:
56                                     ss.available_r2 -= borrow_r2
57                                     borrowed_r2 += borrow_r2
58                                     sources.append((ss, 0, borrow_r2))
59
60             self.borrowed_r1 += borrowed_r1
61             self.borrowed_r2 += borrowed_r2
62
63             # Check if we have enough resources (own + borrowed)
64             if (self.available_r1 + self.borrowed_r1) >= task.r1 and
65                 (self.available_r2 + self.borrowed_r2) >= task.r2:
66                 core.current_task = task
67                 self.waiting_queue.remove(task)
68                 deadline = task.entry_time + task.period * task.num_repetitions
69                 if borrowed_r1 > 0 or borrowed_r2 > 0:
70                     self.borrowed_resources.append({
71                         'sources': sources,
72                         'deadline': deadline,
73                         'r1': borrowed_r1,
74                         'r2': borrowed_r2
75                     })
76                 core.status = 'running'
77                 core.processing_event.set()
78                 break
79
80             if core.current_task:
81                 core.current_task.remaining_time -= 2 # Double speed
82             if core.current_task.remaining_time <= 0:
83                 core.current_task.end_time = current_time
84                 core.current_task.state = 'completed'
85                 self.completed_tasks.append(core.current_task)
86                 self.available_r1 -= min(core.current_task.r1, self.available_r1)
87                 self.available_r2 -= min(core.current_task.r2, self.available_r2)
88                 core.current_task = None
89                 core.status = 'idle'
90                 core.processing_event.clear()
91
92
93
94

```

SS3

```

1 def process(self, current_time):
2     with self.resource_lock:
3         # Process pending tasks
4         pending_to_remove = []
5         for task in list(self.pending_tasks):
6             if task.entry_time <= current_time:
7                 # Check if task requires more resources than system has
8                 if task.r1 > self.r1 or task.r2 > self.r2:
9                     task.failed = True
10                    task.failure_reason = "Insufficient Resources"
11                    task.end_time = current_time
12                    self.completed_tasks.append(task)
13                    pending_to_remove.append(task)
14
15             else:
16                 self.waiting_queue.append(task)
17                 pending_to_remove.append(task)
18
19             if hasattr(task, 'dependencies'):
20                 self.dependency_map[task.name] = task.dependencies
21
22     for task in pending_to_remove:
23         self.pending_tasks.remove(task)
24
25     # Check for dependency failures in waiting queue
26     for task in list(self.waiting_queue):
27         deps = self.dependency_map.get(task.name, [])
28         if deps:
29             for dep_name in deps:
30                 # Check if dependency failed or completed
31                 dep_failed = False
32                 dep_completed = False
33                 for completed in self.completed_tasks:
34                     if completed.name == dep_name:
35                         if completed.failed:
36                             dep_failed = True
37                         else:
38                             dep_completed = True
39                         break
40
41             # If dependency failed, fail this task
42             if dep_failed:
43                 task.failed = True
44                 task.failure_reason = f"Dependency {dep_name} Failed"
45                 task.end_time = current_time
46                 self.completed_tasks.append(task)
47                 self.waiting_queue.remove(task)
48                 break
49
50     # Process remaining waiting tasks
51     waiting_to_ready = []
52     for task in list(self.waiting_queue):
53         if not task.failed: # Skip failed tasks
54             deps = self.dependency_map.get(task.name, [])
55             deps_met = True
56
57             if deps:
58                 for dep_name in deps:
59                     dep_completed = any(t.name == dep_name and not t.failed
60                                         for t in self.completed_tasks)
61                     if not dep_completed:
62                         deps_met = False
63                         break
64
65             if deps_met:
66                 if task.r1 <= self.available_r1 and task.r2 <= self.available_r2:
67                     task.state = 'ready'
68                     self.ready_queue.append(task)
69                     waiting_to_ready.append(task)
70
71     for task in waiting_to_ready:
72         self.waiting_queue.remove(task)
73
74     # Process cores
75     for core in self.cores:
76         if core.current_task:
77             core.current_task.remaining_time -= 1
78             if core.current_task.remaining_time <= 0:
79                 core.current_task.end_time = current_time
80                 core.current_task.state = 'completed'
81                 self.completed_tasks.append(core.current_task)
82                 self.available_r1 -= core.current_task.r1
83                 self.available_r2 -= core.current_task.r2
84                 core.current_task = None
85                 core.status = 'idle'
86                 core.processing_event.clear()
87
88             if not core.current_task and self.ready_queue:
89                 task = self.ready_queue.popleft()
90                 core.current_task = task
91                 self.available_r1 -= task.r1
92                 self.available_r2 -= task.r2
93                 core.current_task.start_time = current_time
94                 core.status = 'running'
95                 core.processing_event.set()
96
97
98
99

```

SS4

MainSystem کلاس

این کلاس مرکز کنترل کلی سیستم است. وظایف اصلی آن شامل تجزیه فایل ورودی (parse_input)، راهاندازی تمامی زیرسیستم‌ها و هسته‌ها، هماهنگی زمانی بین اجزا، و مدیریت چرخه شبیه‌سازی است. متد run حلقه اصلی شبیه‌سازی را تشکیل می‌دهد که در هر سیکل:

۱. به تمام زیرسیستم‌ها سیگنال پردازش ارسال می‌کند.
۲. منتظر می‌ماند تا تمام زیرسیستم‌ها پردازش سیکل جاری را کامل کنند.
۳. زمان سیستم را افزایش می‌دهد.
۴. تاریخچه منابع را به روز می‌کند.
۵. بررسی می‌کند آیا تمام تسک‌ها تکمیل شده‌اند یا خیر.

متد parse_input فایل ورودی را به چهار بخش تقسیم می‌کند: منابع هر زیرسیستم، و چهار بخش تسک‌های مربوط به هر زیرسیستم. برای هر تسک، با توجه به نوع زیرسیستم، پارامترهای خاص (مانند دوره تناب برای SS^۳ یا وابستگی‌ها برای SS^۴) استخراج می‌شوند.

multithreading اصول پیاده سازی

مدیریت چرخه حیات تردها: هر زیرسیستم و هسته پردازشی به عنوان یک ترد مجزا اجرا می‌شود. ویژگی daemon=True تضمین می‌کند که با بسته شدن پنجره اصلی، تمامی تردها به صورت خودکار خاتمه می‌یابند.

هماهنگی بین تردها:

رویدادها (Event): برای هماهنگی مراحل پردازش بین سیستم اصلی و زیرسیستم‌ها استفاده می‌شوند. مثلاً process_event در زیرسیستم‌ها توسط سیستم اصلی فعال می‌شود.

قفل‌ها (Lock): برای محافظت از داده‌های مشترک مانند صفحه و منابع استفاده می‌شوند. مثلاً resource_lock در هر زیرسیستم دسترسی همزمان به منابع را مدیریت می‌کند.

موانع (Barrier): در سیستم اصلی برای همگام‌سازی پایان چرخه تمامی زیرسیستم‌ها استفاده می‌شود.

مدیریت منابع مشترک: داده‌هایی مانند صفحه تسک‌ها و منابع سیستم به دلیل دسترسی همزمان از چند ترد، نیاز به محافظت دارند. تمامی دسترسی‌ها به این داده‌ها در بلوک‌های with lock انجام می‌شود تا از شرایط رقبایی جلوگیری شود.

ارتباط بین تردها: رابط کاربری (در ترد اصلی) و سیستم شبیه‌سازی (در تردهای جداگانه) از طریق مکانیزم‌های thread-safe مانند صفحه‌ای این و کپی‌گیری داده‌ها در قفل‌ها با هم ارتباط برقرار می‌کنند. تاریخچه منابع (resource_history) با استفاده از history_lock محافظت می‌شود.

بهینه‌سازی‌ها:

مکانیزم خواب (Sleep): حلقه‌های پردازشی اصلی (مانند run Core) شامل وقفه‌های کوتاه ((time.sleep(0.1)) هستند تا از مصرف بیش از حد CPU جلوگیری شود.

به روزرسانی غیرهمzman GUI: رابط کاربری به جای به روزرسانی لحظه‌ای، در بازه‌های زمانی مشخص (هر ۱۰۰ میلی‌ثانیه) وضعیت سیستم را بررسی می‌کند تا از کاهش عملکرد جلوگیری شود.

```
1 class MainSystem(threading.Thread):
2     def __init__(self):
3         super().__init__(daemon=True)
4         self.subsystems = []
5         self.current_time = 0
6         self.history_lock = threading.Lock()
7         self.resource_history = []
8         self.running = True
9         self.pause_event = threading.Event()
10        self.barrier = threading.Barrier(
11            1 + # Main thread
12            4 + # 4 subsystems
13            8 # Total cores (3+2+1+2)
14        )
15
16    def start(self):
17        # Single place to start all threads
18        for ss in self.subsystems:
19            # Start subsystem thread
20            if not ss.is_alive():
21                ss.start()
22            # Start core threads
23            for core in ss.cores:
24                if not core.is_alive():
25                    core.start()
26        # Start main system thread last
27        if not self.is_alive():
28            super().start()
29
30    def run(self):
31        while self.running:
32            try:
33                self.pause_event.wait()
34
35                # Clear previous cycle flags
36                for ss in self.subsystems:
37                    ss.cycle_complete.clear()
38
39                # Signal all subsystems to process
40                for ss in self.subsystems:
41                    ss.process_event.set()
42
43                # Wait for all subsystems to complete
44                for ss in self.subsystems:
45                    ss.cycle_complete.wait()
46
47                # Increment time after all subsystems finish
48                self.current_time += 1
49
50                # Update resource history
51                self.current_time += 1
52                with self.history_lock:
53                    current_resources = []
54                    for ss in self.subsystems:
55                        with ss.resource_lock:
56                            current_resources.append((ss.available_r1, ss.available_r2))
57                    self.resource_history.append(current_resources)
58
59                if self.check_all_tasks_completed():
60                    print("All tasks completed")
61                    self.stop()
62                    break
63
64                    time.sleep(0.1)
65                except Exception as e:
66                    print(f"MainSystem error: {str(e)}")
67
68    def check_all_tasks_completed(self):
69        for ss in self.subsystems:
70            if (ss.pending_tasks or ss.waiting_queue or
71                any(core.current_task for core in ss.cores) or
72                any(core.ready_queue for core in ss.cores) or
73                (hasattr(ss, 'ready_queue') and ss.ready_queue)):
74                return False
75        return True
76
77    def stop(self):
78        self.running = False
79        for ss in self.subsystems:
80            ss.running = False
81            # Removed loop stopping cores
82        self.join()
```

```

1  def parse_input(self, input_lines):
2      # Process lines while ignoring comments
3      cleaned_lines = []
4      for line in input_lines:
5          # Split on '#' and take first part, strip whitespace
6          clean_line = line.split('#')[0].strip()
7          if clean_line: # Only keep non-empty lines
8              cleaned_lines.append(clean_line)
9
10     resource_lines = cleaned_lines[:4]
11     ss_resources = []
12     for line in resource_lines:
13         r1, r2 = map(int, line.split())
14         ss_resources.append((r1, r2))
15
16     task_sections = []
17     current_section = []
18     for line in cleaned_lines[4:]:
19         line = line.strip()
20         if line == '$':
21             if current_section:
22                 task_sections.append(current_section)
23                 current_section = []
24             else:
25                 current_section.append(line)
26         if current_section:
27             task_sections.append(current_section)
28
29     for i in range(4):
30         r1, r2 = ss_resources[i]
31         if i == 0:
32             ss = SS1(r1, r2)
33         elif i == 1:
34             ss = SS2(r1, r2)
35         elif i == 2:
36             ss = SS3(r1, r2)
37         else:
38             ss = SS4(r1, r2)
39     ss.attach_main_system(self)
40     self.subsystems.append(ss)
41
42     for i, section in enumerate(task_sections):
43         ss = self.subsystems[i]
44         for task_line in section:
45             parts = task_line.split()
46             name = parts[0]
47             runtime = int(parts[1])
48             if i == 0: # SS1: name runtime r1 r2 entry_time target_core
49                 r1, r2, entry_time, initial_core = map(int, parts[2:6])
50                 task = Task(name, runtime, r1, r2, entry_time, initial_core=initial_core)
51                 ss.add_task(task)
52                 ss.pending_tasks.append(task) # Add to pending, not ready queue
53             elif i == 1: # SS2: name runtime r1 r2 entry_time
54                 r1, r2, entry_time = map(int, parts[2:5])
55                 task = Task(name, runtime, r1, r2, entry_time)
56                 ss.add_task(task)
57                 ss.pending_tasks.append(task)
58             elif i == 2: # SS3: name runtime r1 r2 entry_time period num_repetitions
59                 r1, r2, entry_time, period, num_reps = map(int, parts[2:7])
60                 task = Task(name, runtime, r1, r2, entry_time, period=period, num_repetitions=num_reps)
61                 ss.add_task(task)
62                 ss.pending_tasks.append(task)
63             elif i == 3: # SS4: name runtime r1 r2 entry_time dependencies...
64                 r1, r2, entry_time = map(int, parts[2:5])
65                 dependencies = parts[5:] if len(parts) > 5 else []
66                 dependencies = [d for d in dependencies if d != '-']
67                 task = Task(name, runtime, r1, r2, entry_time, dependencies=dependencies)
68                 ss.add_task(task)
69                 ss.pending_tasks.append(task)
70
71             if i == 3:
72                 name_to_task = {task.name: task for task in ss.tasks}
73                 for task_name, deps in ss.dependency_map.items():
74                     resolved_deps = []
75                     for dep_name in deps:
76                         if dep_name in name_to_task:
77                             resolved_deps.append(name_to_task[dep_name])
78                     ss.dependency_map[task_name] = resolved_deps
79
80     def run_simulation_step(self):
81         if self.running and any(any(task.end_time is None for task in ss.tasks) for ss in self.subsystems):
82             self.current_time += 1
83             for ss in self.subsystems:
84                 ss.process(self.current_time)
85             # Store per-subsystem resources instead of sum
86             current_resources = []
87             for ss in self.subsystems:
88                 current_resources.append((ss.available_r1, ss.available_r2))
89             self.resource_history.append(current_resources)
90             return True
91     return False

```

gui.py

کلاس SubsystemVisualizer

این کلاس مسئول ایجاد رابط کاربری گرافیکی با استفاده از کتابخانه Tkinter است. اجزای اصلی شامل: پنل کنترل: دکمه‌های شروع/توقف، بارگیری ورودی، و انتخاب فایل درخت سلسله مراتب: نمایش ساختار سیستم شامل شناسه‌های تردها

نمودارها:

نمودار تخصیص منابع: نمایش گرافیکی ارتباط بین تردها و منابع با استفاده از کتابخانه NetworkX

نمودار مصرف منابع: نمایش روند مصرف منابع R1 و R2 در طول زمان با استفاده از Matplotlib

تپ‌های زیرسیستم: نمایش وضعیت لحظه‌ای هر زیرسیستم شامل وضعیت هسته‌ها، صفحه‌ها، و منابع

مکانیزم بهروزرسانی:

update_allocation_graph

این متدهای گراف جهت‌دار ایجاد می‌کند که نودها نشان‌دهنده منابع (مربع قرمز) و تردها (دایره سبز) هستند. یالهای آبی نشان‌دهنده تخصیص منابع فعلی، و یالهای نارنجی خطچین نشان‌دهنده درخواست‌های منابع منتظر هستند. این گراف به صورت پویا بر اساس وضعیت فعلی سیستم به روز می‌شود.

این نمودار خطی، مصرف منابع هر زیرسیستم را در طول زمان نمایش می‌دهد. برای هر زیرسیستم، دو خط (یک خط برای R1 و یک خط خطچین برای R2) رسم می‌شود که مقادیر آنها از تاریخچه ذخیره شده در MainSystem.resource_history استخراج می‌شود.

این متدهای گرافیکی مسئول بهروزرسانی تمامی اجزای رابط کاربری است. در هر فراغوانی شناسه تردهای مدیریتی و هسته‌ها به روز می‌شوند، مقادیر منابع فعلی هر زیرسیستم نمایش داده می‌شود، وضعیت هر هسته (در حال اجرا، بیکار، خطأ) با رنگ‌های مختلف نشان داده می‌شود، محتوای صفحه‌ها (انتظار، آماده، تکمیل شده) از روی داده‌های سیستم اصلی پر می‌شود و در نهایت نمودارها مجدداً رسم می‌شوند.

مدیریت رویدادها:

شروع/توقف شبیه‌سازی:

با استفاده از رویداد pause_event در MainSystem، اجرای حلقه شبیه‌سازی کنترل می‌شود. فشردن دکمه Start این رویداد را فعال کرده و فشردن Pause آن را غیرفعال می‌کند.

بارگیری ورودی: کاربر می‌تواند هم از طریق کمپیوت مستقیم در جعبه متن و هم از طریق انتخاب فایل متنی، داده‌های ورودی را به سیستم وارد کند. پس از تجزیه موفقیت‌آمیز، درخت سلسله مراتب و تپ‌ها به طور خودکار به روز می‌شوند.

ریست سیستم: با فشردن دکمه Reset، تمامی تردهای سیستم متوقف شده، داده‌های پاک می‌شوند، و رابط کاربری به حالت اولیه بازمی‌گردد.



