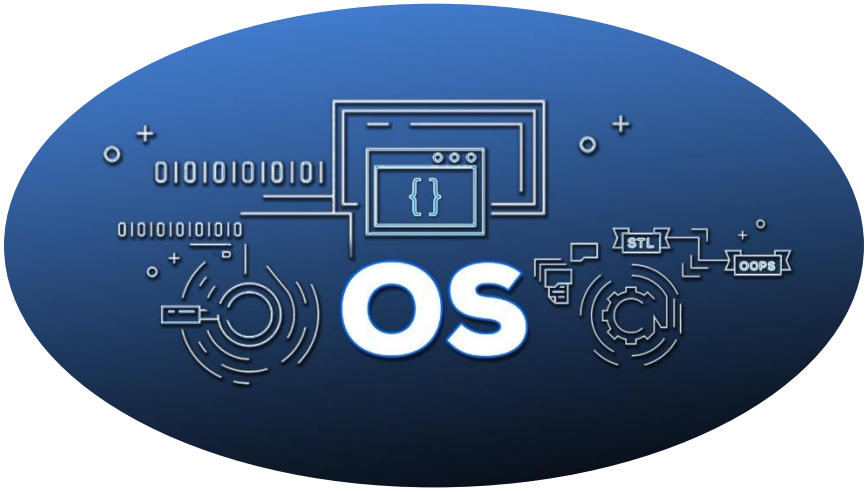
مهیار محمدی متین

گزارش کار پروژه درس اصول سیستم‌های عامل ۶۱۰۳۹۸۱۶۶

mahyarmohmmadi@ut.ac.ir

مقدمه:

انتخاب من از بین الگوریتم های سیستم عامل، الگوریتم مولتی تسکینگ(همزمانی فرایندها) است. از گذشته، برای من همزمان انجام دادن کارها توسط سیستم عامل جذاب و خارق العاده می‌نمود و از این نظر خواستم که دستی بر آتش برم و اینبار خودم این الگوریتم را پیاده سازی کنم. در این پروژه الگوریتم موازی سازی و همروندی هر دو پیاده سازی می‌شوند. موازی سازی یعنی یک core از CPU آنقدر سریع بین تسک‌ها جا به جا شود که ما فکر کنیم همه به طور همزمان در حال اجرا هستند. اما همروندی یعنی چند core را به جان task ها بیاندازیم تا همزمان انجام شوند.
همچنین برای ساخت اعداد رندوم یک الگوریتم به صورت دستی نوشته شده است که از ساعت سیستم کمک گرفته و به تولید آنها می‌پردازد. آزمون درستی این تولید اعداد نیز در کد آمده است تا مطمئن شویم اعداد رندوممان واقعا رندوم هستند!
در این پروژه ما یک task را مجموعه ای از پروسس ها(process) در نظر میگیریم که این پروسس ها هرکدام یک حافظه مورد نیاز و یک زمان لازم برای اجرا دارند. هنگامی که یک پروسس در حال انجام است، روی ram قرار می‌گیرد و طی تبادلات آن با CPU محاسبه می‌شود. مهم است که ظرفیت ram محدود است و توسط کاربر ورودی داده میشود و در صورتی که ram جایی برای پروسس ها نداشته باشد، سیستم عامل از virtual ram استفاده میکند که مطابق هارد است و سرعت به نسبت پایین تری دارد. علت آن، انتقال از هارد به ram است که یک مدت زمانی طول میکشد. کاربر می‌تواند این مقدار زمان را نیز دستی ست کند. همچنین به اثبات قانون Amdahl خواهیم پرداخت.



بررسی عملکرد الگوریتم:

بخش اصلی اجرای الگوریتم، هماهنگ کردن core های CPU با همدیگر و تقسیم وظیفه درست بین انها است. بدین منظور، چون تمام core ها به صورت همزمان میخواهند پروسس هارا انجام دهند، ممکن است تداخل پیش بیاید و حین انجام یک پروسس، یک core مزاحم core دیگر شود. بدین منظور باید از lock استفاده کنیم. بدین صورت که پروسسی که در حال انجام توسط یک core است، توسط آن core قفل می شود و پس از اجرا دوباره قفل آن باز می‌شود.

دقت کنید که در الگوریتمی که ما پیاده میکنیم، پروسس را کوچکترین واحد قابل محاسبه در نظر میگیریم که نمیتواند بین دو core تقسیم شود. اما یک task می‌تواند به صورت همزمان توسط چند core محاسبه شود.

مسئله بعدی در الگوریتم آن است که انجام یک پروسس پیشنیاز داشته باشد. یعنی پروسس های قبلی باید انجام شوند تا بتوانیم پروسس جدید را حل کنیم. در این صورت است که عملا core های بیشتر به کار نخواهد آمد! مگر آنکه سعی کنیم core هارا به سراغ task دیگری بفرستیم.

با وجود اینکه چند core داریم اما همچنان قانون موازی سازی باید توسط تمام core ها پیاده شود. یعنی اگر یک core یک پروسس از تسکی را انجام داد، دیگر حق ندارید در آن تسک ادامه بدهد و باید سراغ تسک دیگری برود. بدین صورت علاوه بر اجرای همروند تسک ها، موازی سازی هم خواهیم داشت.

توضیح کد شبیه سازی سیستم عامل:

ابتدا لایبرری های مورد نیاز را import میکنیم:

```
import time
import numpy as np
import pandas as pd
import timeit
import threading
import matplotlib.pyplot as plt
```

حال به تعریف class های مورد نیاز که سازنده سیستم عاملمان است می‌پردازیم:

کلاس process:

```
class Process:

    def __init__(self,task_id,memory_needed,time_needed,need_prev_prob):
        #the memory that this process is need(by byte)
        self.memory = create_random(*memory_needed)[0]

        #the time that this process need to be done(by second)
        self.time = create_random(*time_needed)[0]

        #this process blongs to which task?
        self.task_id = task_id

        #is this process lock?
        self.lock = False

        #if true, this process can't be done until all previous process in that task get done
        self.need_prev = np.random.choice([False,True],1,p=[1-need_prev_prob,need_prev_prob])

        #true when this process get done
        self.done = False
```

این کلاس نماینده یک پروسس ورودی به سیستم عامل است. همانطور که اشاره شد یک ظرفیتی از حافظه را اشغال میکند و یک زمان لازم برای انجام دارد که هردوی اینها باید ست شوند. علاوه بر آن، قابلیت قفل شدن برای آن وجود دارد(که در پیاده سازی الگوریتم همروندی به ما کمک میکند)

همچنین متغییر مهم دیگر need_prevاست که اگر true باشد، بدین معناست که این پروسس نمیتواند انجام شود، مگر آنکه تمام پروسس های قبل از آن در این تسک انجام شده باشند. این مقدار به صورت احتمالی که کاربر ورودی میدهد ست میشود. این پارامتر نقش کلیدی در ایجاد وابستگی بین پروسس ها ایجاد میکند. هرچه پروسس ها بهم وابسته تر باشند، همروندی کم اثرتر خواهد بود. این مورد را با آزمایش ثابت خواهیم کرد.

کلاس task:

```python
class Task:

    def __init__(self,id,proc_count,memory_needed=(0,0),time_needed=(0,1000),prev_prob=0):

        self.id = id

        #how many process this task had?
        self.proc_count = proc_count

        #create random process for this task
        self.process_queue = [Process(id,memory_needed,time_needed,prev_prob) for _ in range(proc_count)]

        #true if the task is done
        self.done = False

    #just a helper function to see that which core is working
    def state(self,core_id):
        print(f'core{core_id}:',end=' ')
        if(self.done):
            print(f'task {self.id} is completely done!')
            return
        c=0
        for proc in self.process_queue:
            if(proc.done):
                c+=1

        if(c*100/self.proc_count==100): print(f'task {self.id} is completely done!')
        else: print(f'task {self.id} is {int(c*100/self.proc_count)}% done.')

    #check if task is done
    def is_done(self):
        if(self.done):return True
        for proc in self.process_queue:
            if(proc.done==False):return False
        self.done = True
        return True
```

همانطور که اشاره شد هر task از چند process تشکیل میشود. این تعداد پروسس توسط کاربر ورودی داده میشود و همچنین مشخص میشود که به طور میانگین هر پروسس در این task چقدر حافظه اشغال میکند و چقدر زمان نیاز برای حل شدن دارد.

در constructor این کلاس همانطور که مشخص است به تعدادی که کاربر بخواهد پروسس رندوم تولید شده و در صف مربوط به این task قرار میگیرد.

همچنین تمام پروسس ها از نظر وابستگی به یکدیگر از احتمال prev_prob تبعیت میکنند. هرچه این احتمال بیشتر باشد، توانایی ما برای اجرای همروند دستورات کمتر خواهد شد.

حال قبل از آنکه به سراغ تعریف CPU برویم، مهم ترین کلاس این پروژه یعنی کلاس core را تعریف میکنیم که به نوعی هوش سیستم عاملمان است. قرار گیری چند دسته از این core ها داخل CPU است که باعث میشود پروسس ها به خوبی هندل شوند.

کلاس core:

```
class Core:
    id = None
    busy = None
    cur_task = None

    def __init__(self,id):
        self.id = id
        self.time_spend = 0

    def run(self):
        #print('i am core:',self.id)
        finished = False
        while(finished==False):
            finished = True
            for task in tasks:
                time.sleep(0.000001)
                #check if task is already done or not
                if(task.is_done()):continue
                finished = False
                #try solve one process of task
                prev_proc_lock = False
                for proc in task.process_queue:
                    if(proc.done):continue
                    #process is lock, so another core is working on it. we check next process
                    if(proc.lock): prev_proc_lock = True; continue

                    #if we aren't in top of queue and we need prev proc to do this proc,
                    #so other core is working on it and we should wait for it's response.
                    #so we go to other task
                    if(proc.need_prev and prev_proc_lock): break

                    #else we run core on this process
                    proc.lock=True
                    time.sleep(proc.time*10**(-3))
                    self.time_spend+=proc.time*10**(-3)
                    proc.done = True
                    proc.lock = False
                    break #we break to do another task. we want to do tasks parallel

```

ایده اجرای core بدین صورت است که هر core به صورت موازی روی task ها حرکت میکند. وقتی به یک task رسید، چک میکند که انجام شده است یا خیر. اگر انجام شده بود، به سراغ task بعدی میرود. در غیر این صورت به سراغ اولین پروسس انجام نشده از آن تسک می‌رود. چک میکند که آیا این پروسس لاک هست یا نه. اگر لاک باشد، یعنی یک core دیگر قبل از آن به سراغ آن رفته و مشغول به حل آن پروسس است.

اگر پروسس لاک باشد، به سراغ پروسس بعدی همان تسک میرود و سعی میکند آن را حل کند. اما دقت کنید! شاید نتوانیم پروسس های بعدی تسک را حل کنیم اگر وابسته به پروسس قبلی باشد که هنوز حل نشده است! پارامتر need_prev به ما میگوید که اگر میخواهیم این پروسس را حل کنیم به جواب پروسس های قبل از آن نیاز داریم یا خیر. اگر نیاز داشتیم و پروسس قبلی توسط یک core دیگر لاک شده و در حال حل بود، از خیر آن پروسس میگذریم و چون دیگر نمیتوانیم پروسس دیگری را انجام دهیم، دیگر در این task کاری انجام نداده و به سراغ task بعدی میرویم.

اما بیایید فرض کنیم که مشکلات بالا پیش نیاید و core ما بالاخره یک پروسس غیر لاک پیدا کند که نیازی هم به حل پروسس های قبل از آن نداشته باشد یا خودش پروسس سر صف باشد. در این صورت ابتدا پروسس را لاک میکنیم و سپس به میزانی که آن پروسس زمان لازم دارد، core را sleep میکنیم(مثلا در حال انجام آن پروسس است). پس از انجام شدن پروسس آن را آنلاک میکنیم.

دقت کنید که در این مرحله ما توانستیم یک پروسس از یک تسک را انجام دهیم. حال برای اعمال قاعده موازی سازی، دیگر نباید به ادامه حل پروسس های این تسک بپردازیم! از حلقه break کرده و به سراغ تسک های دیگر میرویم.

کلاس CPU:

```
class CPU:

    def __init__(self,core_count=1,ram=2,vir_ram=200):
        #set cores
        self.cores = [Core(i) for i in range(core_count)]

        #set ram capacity
        self.ram = ram

        #set virtual ram capacity
        self.vir_ram = vir_ram

    #run cpu on global list tasks
    def run(self):
        t = []
        #set timer to calculate the time needed for this cpu to handle all tasks
        start = timeit.default_timer()
        def doWork(i):
            self.cores[i].run()

        #run all cores together!
        for i in range(len(self.cores)):
            thread = threading.Thread(target=doWork, args=(i,))
            t.append(thread)
            thread.start()

        #we need for cores to finish
        for thread in t:
            thread.join()

        #stop timer
        stop = timeit.default_timer()
        time = int((stop - start)*1000)*10**(-3)
        return time
```

در این کلاس ابتدا coreها را به تعداد ورودی داده شده توسط کاربر میسازیم. همچنین مقدار ram و virtual ram را ست میکنیم.

در تابع run به صورت موازی همه core های cpu را ران میکنیم. همچنین زمان آغاز و پایان را حساب میکنیم تا میزان زمانی که طول کشیده است تا تمام task ها انجام شود.

تابع ساخت اعداد رندوم:

```
def create_random(Min,Max,RandomSize):
    RandomNumber = []
    power_rate = len(str(Max - Min))
    x_0 = int(time.time()*int(10e5))
    temp = RandomSize
    while(temp > 0):
        x_temp = ((19673528492*x_0 + 10041399)% 2**30)/2**30 * 10**power_rate
        if( Min <= x_temp <= Max):
            RandomNumber.append(int(x_temp))
            temp -= 1
        x_0 = x_temp
    return RandomNumber
```

با استفاده از ساعت سیستم و یک رندوم سید و ضرب دو عدد چند رقمی در هم به ساخت عدد رندوم میپردازیم.
همچنین پس از ساخت عدد رندوم اول، از آن برای ساخت اعداد رندوم بعدی استفاده میکنیم.

کد تست آزمون آماری برای صحت رندوم بودن تابع فوق:

```
def runsTest(l, l_median):
    runs, n1, n2 = 0, 0, 0
    for i in range(len(l)):
        if (l[i] >= l_median and l[i-1] < l_median) or \
                (l[i] < l_median and l[i-1] >= l_median):
            runs += 1

        if(l[i]) >= l_median: n1 += 1
        else: n2 += 1

    runs_exp = ((2*n1*n2)/(n1+n2))+1
    stan_dev = math.sqrt((2*n1*n2*(2*n1*n2-n1-n2))/(((n1+n2)**2)*(n1+n2-1)))
    z = (runs-runs_exp)/stan_dev
    return z

l = create_random(1,100,100000)
l_median= statistics.median(l)

Z = abs(runsTest(l, l_median))

if(Z>1.96):
    print('not random')
else:
    print('totally random!')


```

```
totally random!
```

تابع ساخت task های رندوم:

```
#create tasks
#memory_needed: at most process have this memory(in byte)
#time_needed: at most process have this time to execute(in mili second)
def make_tasks(task_count=10,proc_count=20,time_range=(1,1000),prev_prob=0):
    tasks = [Task(i,proc_count,(0,1),time_range,prev_prob) for i in range(1,task_count)]
    s = 0
    for task in tasks:
        c=0
        for proc in task.process_queue:
            c+=proc.time*10**(-3)
        #print(c)
        s+=c
    #print(s)
    return (tasks,s)
```

اجرای برنامه:

```
tasks,expected = make_tasks(task_count=5,time_range=(1,100),prev_prob=1)
cpu = CPU(core_count=1,ram=2,vir_ram=200)
cpu.run()
```

کاربر میتواند ابتدا task هارا بر اساس رنج دلخواه و شرایط مخصوص به خود بسازد و سپس CPU با شرایط دلخواه را طراحی کند و سپس CPU را روی تسک ها ران کند.

انجام آزمایش و نتایج:

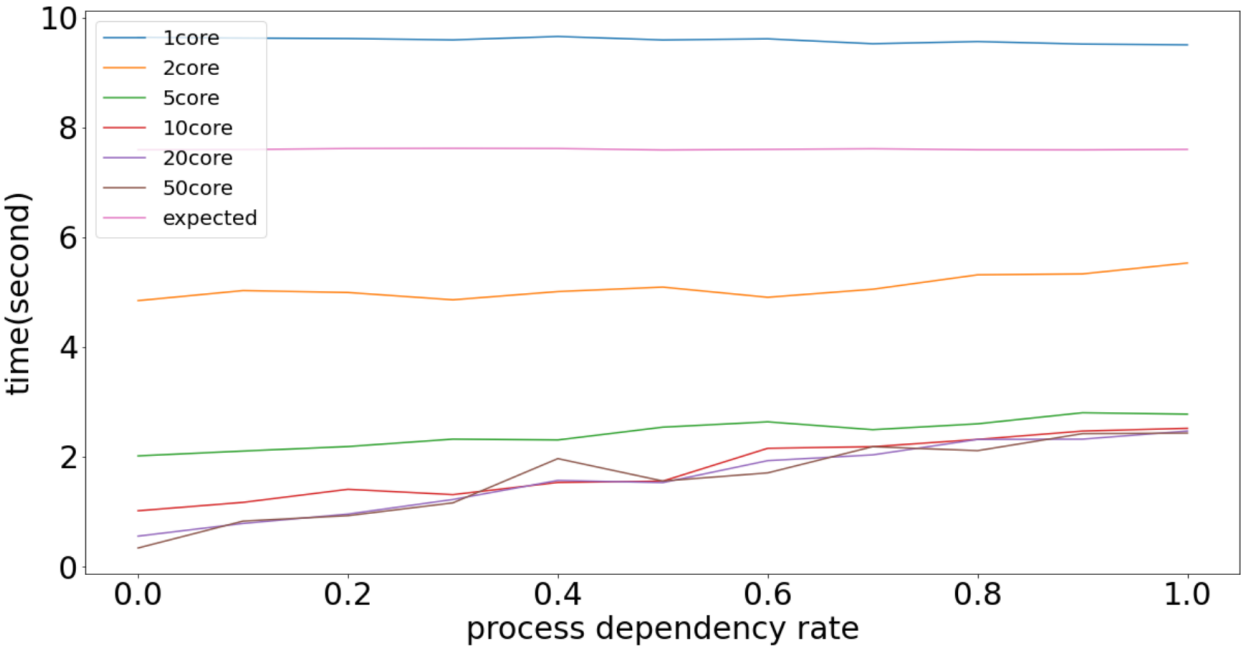
```
times = [[],[],[],[],[],[]]
probs = [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
expecteds = []
for prev_prob in probs:
    ex = []
    for i,core_count in enumerate([1,2,5,10,20,50]):
        tasks,expected = make_tasks(task_count=5,time_range=(90,100),prev_prob=prev_prob)
        cpu = CPU(core_count)
        times[i].append(cpu.run())
        ex.append(expected)
    expecteds.append(int(sum(ex)*1000/len(ex))*10**(-3))

df = pd.DataFrame({
    '1core':times[0],
    '2core':times[1],
    '5core':times[2],
    '10core':times[3],
    '20core':times[4],
    '50core':times[5],
    'prev_prob':probs,
    'expected':expecteds
})
display(df)
ax = df.set_index(df.prev_prob).drop('prev_prob',axis=1).plot(
    figsize=(20,10),
    fontsize=30)
ax.set_ylabel('time(second)',fontdict={'fontsize':30})
ax.set_xlabel('process dependency rate',fontdict={'fontsize':30})
plt.legend(loc=2, prop={'size': 20})
pass
```

طی کد نوشته شده فوق ما به صورت میانگین زمان لازم برای حل تسک های ثابتی توسط CPU با شرایط مختلف بدست می‌آوریم. یکبار CPU هارا از نظر تعداد core ها برابر اعداد ۱، ۲، ۵، ۱۰، ۲۰ و ۵۰ قرار میدهیم . یک بار میزان وابستگی پروسس ها به یکدیگر را در رنج ۰ تا ۱ دست خوش تغییر میکنیم.

در این صورت جدول زیر حاصل خواهد شد:

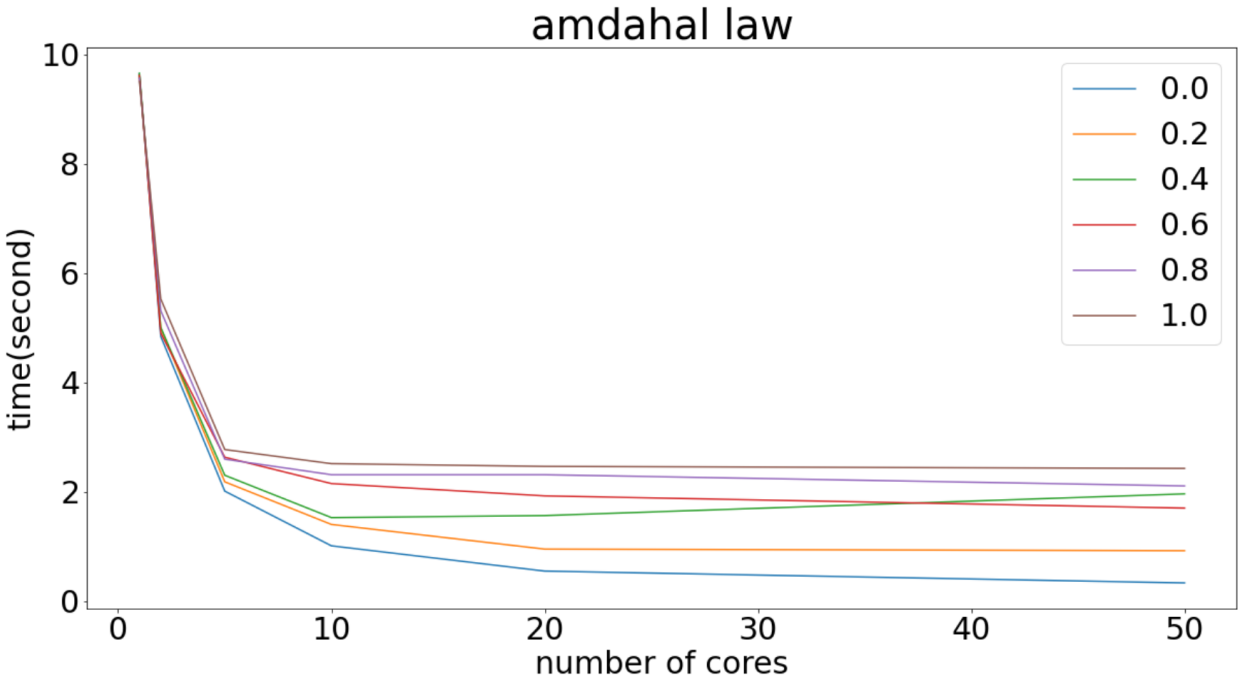
	1core	2core	5core	10core	20core	50core	prev_prob	expected
0	9.640	4.842	2.014	1.014	0.552	0.336	0.0	7.591
1	9.625	5.025	2.102	1.167	0.783	0.826	0.1	7.595
2	9.616	4.990	2.184	1.405	0.954	0.925	0.2	7.613
3	9.592	4.857	2.319	1.308	1.219	1.158	0.3	7.616
4	9.654	5.007	2.304	1.530	1.567	1.964	0.4	7.613
5	9.591	5.088	2.537	1.554	1.527	1.556	0.5	7.587
6	9.611	4.903	2.634	2.152	1.927	1.704	0.6	7.598
7	9.522	5.049	2.491	2.181	2.032	2.181	0.7	7.609
8	9.562	5.312	2.598	2.314	2.314	2.109	0.8	7.591
9	9.516	5.329	2.800	2.466	2.319	2.419	0.9	7.589
10	9.502	5.527	2.774	2.516	2.466	2.429	1.0	7.598

هرچه ستون prev_prob به سمت ۱ برود، میزان وابستگی پروسس ها بیشتر شده و همانطور که می‌بینید زمان لازم برای انجام آنها بیشتر میشود. یک نمودار از جدول فوق رسم میکنیم تا به طور دقیق تری بررسی کنیم:



همانطور که مشخص است، هرچه میزان وابستگی پروسس‌ها به یکدیگر بیشتر شود، افزایش تعداد core ها بی ثمرتر خواهد بود! زیرا امکان موازی سازی کمتر شده و یکسری core منتظر انجام دستورات بیکار می‌مانند. هرچند که میبینیم زمان انجام پروسس ها توسط یک core ۱۰ ثانیه طول میکشد، اما به نظرم یرسد بیش از ۵ core تغییر آنچنانی در سرعت ایجاد نمیکند. اللخصوص وقتی که وابستگی پروسس ها به یکدیگر زیاد باشد. حال به اثبات قانون Amdahl میپردازیم:

```
ax = pd.DataFrame(df.iloc[[0,2,4,6,8,10]].drop(['prev_prob','expected'],axis=1).to_numpy().T,
            columns = [0,0.2,0.4,0.6,0.8,1],index=[1,2,5,10,20,50]).plot(
                figsize=(20,10),
                fontsize=30,
                )
ax.set_ylabel('time(second)',fontdict={'fontsize':30})
ax.set_xlabel('number of cores',fontdict={'fontsize':30})
plt.legend(loc=1, prop={'size': 30})
plt.title('amdahal law',size=40)
pass
```



این مشابه همان نمودار Amdahl است که بیان میدارد، لزوما ۵۰برابر کردن تعداد هسته های CPU به افزایش ۵۰درصدی منجر نمیشود. بلکه وابسته به S است که S میزان وابستگی پروسس ها به یکدیگر است. همچنین هرچه دیتاها به یکدیگر وابسته تر باشند، در زمان بالاتری همگرا میشوند.