

دانشگاه علم و صنعت

دانشکده مهندسی کامپیوتر

درجه تحصیلی: کارشناسی

تمرین تئوری1 OS

گردآورنده:

پرنیان شاکریان - 99400064

استاد:

دکتر انتظاری

سال تحصیلی: اردیبهشت 1402

بخش اول) سوالات تشریحی

1. multitasking و Multiprogramming را توضیح دهید و تفاوت آنها را بیان کنید.

Multiprogramming یک طراحی معماری کامپیوتری است که در آن سیستم کامپیوتری می‌تواند چندین برنامه را به طور همزمان اجرا کند و در CPU برای اجرای دستورالعمل‌ها از هر برنامه به روشی interleaved استفاده کند. در یک سیستم Multiprogramming ، چندین برنامه می‌توانند همزمان در حافظه اصلی کامپیوتر قرار بگیرند. سیستم عامل بخشی از حافظه را برای هر برنامه اختصاص می دهد و در صورت نیاز بین برنامه ها سوئیچ می کند و به هر برنامه اجازه می دهد بدون انتظار برای اتمام برنامه دیگر پیشرفت کند. ایده کلیدی پشت Multiprogramming، بهینه کردن و به حداکثر رساندن استفاده از CPU و سایر منابع سیستم، مانند حافظه و دستگاه های I/O است. با اجازه دادن به چندین برنامه برای اجرای همزمان، سیستم می تواند CPU را مشغول نگه دارد و از هدر رفتن قدرت پردازش آن جلوگیری کند. Multiprogramming یکی از ویژگی‌های اساسی سیستم‌عامل‌های مدرن است که در بسیاری از برنامه‌ها استفاده می‌شود، مانند سیستم‌های time-sharing که در آن چندین کاربر می‌توانند به طور همزمان یک کامپیوتر را به اشتراک بگذارند. عملکرد اصلی آن در یک سیستم عامل، به حداکثر رساندن استفاده از منابع سیستم مانند CPU، حافظه و دستگاه‌های I/O است. در یک سیستم Multiprogramming ، سیستم عامل حافظه را به چندین برنامه اختصاص می‌دهد و به آن‌ها اجازه می‌دهد تا به طور همزمان اجرا شوند. CPU بخش کوچکی از یک برنامه را اجرا می کند، سپس به برنامه دیگر سوئیچ می کند و بعد به برنامه اول برای اجرای قسمت بعدی برمی گردد. این اجرای بهم پیوسته از چندین برنامه، به CPU اجازه میدهد تا مشغول بماند و از بیکاری در زمانی که برنامه منتظر عملیات I/O یا منابع دیگر است، جلوگیری می کند. Multiprogramming همچنین پیاده‌سازی سیستم‌های time-sharing را تسهیل میبخشد. سیستم عامل زمان CPU را به هر کاربر اختصاص می دهد و برنامه کاربر برای مدت کوتاهی، معمولاً چند میلی ثانیه، قبل از اینکه CPU به برنامه کاربر دیگر سوئیچ شود، اجرا می شود. یکی دیگر از عملکردهای آن بهبود توان عملیاتی و زمان پاسخگویی سیستم است. با اجازه دادن به چندین برنامه برای اجرای همزمان، سیستم می تواند تعداد بیشتری از درخواست ها را پردازش کند و سریعتر به درخواست های کاربر پاسخ دهد.

Multitasking یک نوع دیگری از طراحی معماری کامپیوتری است که به کامپیوتر اجازه می دهد تا چندین کار یا فرآیند را به طور همزمان اجرا کند. در Multitasking، CPU به سرعت بین وظایف یا فرآیندهای مختلف سوئیچ می کند تا ظاهری شبیه به اجرای موازی(parallel execution) داشته باشد، حتی اگر CPU در واقع فقط یک وظیفه را در یک زمان اجرا می کند. این به کاربران اجازه می دهد تا چندین برنامه را به طور همزمان اجرا کرده و به سرعت بین آنها جابجا شوند، بدون اینکه منتظر بمانند تا هر برنامه قبل از شروع برنامه دیگر کامل شود. Multitasking را می توان به روش های مختلفی از جمله preemptive multitasking و cooperative multitasking به دست آورد. در preemptive multitasking، سیستم عامل بر اساس سطوح اولویت و الگوریتم‌های زمان‌بندی تصمیم می‌گیرد که کدام کار بعدی را اجرا کند، در حالی که در cooperative multitasking، هر کار به‌طور داوطلبانه CPU را تحویل می‌دهد تا به وظایف دیگر اجازه اجرا بدهد. معمولاً در برنامه هایی مانند سیستم عامل ها، مرورگرهای وب، نرم افزارهای ویرایش ویدیو و بسیاری موارد دیگر استفاده می شود. عملکرد آن به این صورت است که سیستم عامل اجرای Multitasking را با تخصیص زمان CPU به هر برنامه بر اساس سطوح اولویت و الگوریتم های زمان بندی مدیریت می کند. CPU به سرعت بین وظایف مختلف سوئیچ می کند و بخش کوچکی از هر کار را به نوبه خود انجام می دهد و ظاهری شبیه به اجرای موازی می دهد. همچنین با مشغول نگه داشتن CPU و اجتناب از زمان بیکاری، توان عملیاتی کلی سیستم را بهبود می بخشد. این بدان معنی است که سیستم می تواند تعداد بیشتری از درخواست ها را پردازش کند و سریعتر به درخواست های کاربر پاسخ دهد. یکی دیگر از عملکردهای مهم Multitasking بهبود پایداری و قابلیت اطمینان سیستم است. در این محیط، اگر یک برنامه از کار بیفتد یا با خطا مواجه شود، بر اجرای برنامه های دیگر تأثیری نمی گذارد. سیستم عامل می تواند خطاها را شناسایی و بازیابی کند و از پایداری و اطمینان سیستم اطمینان حاصل کند.

تفاوت ها: در حالی که هر دو تکنیک‌هایی هستند که توسط سیستم‌عامل‌ها برای بهبود کارایی سیستم استفاده می‌شوند، اما در تمرکز و پیاده‌سازی متفاوت هستند مانند:

* تخصیص حافظه: در Multiprogramming به هر برنامه مقدار مشخصی از حافظه اختصاص می‌یابد و برنامه‌ها نمی‌توانند از این محدودیت تجاوز کنند. در مقابل، در Multitasking، سیستم عامل به صورت پویا حافظه را در صورت نیاز به هر کار اختصاص می دهد.
* اولویت و زمان‌بندی: در Multiprogramming، اولویت و زمان‌بندی معمولاً بر اساس مدت زمانی است که یک برنامه منتظر زمان CPU بوده است. در مقابل، در Multitasking، اولویت و زمان‌بندی معمولاً بر اساس ترکیبی از عواملی مانند سطح اولویت، کوانتوم زمانی و الگوریتم‌های زمان‌بندی است.
* اشتراک منابع: در Multiprogramming، منابعی مانند حافظه و دستگاه‌های I/O بین برنامه‌ها به اشتراک گذاشته می‌شوند، اما تنها یک برنامه می‌تواند در یک زمان از یک منبع استفاده کند. در Multitasking، منابع بین وظایف به اشتراک گذاشته می شود و چندین کار می توانند از یک منبع به طور همزمان استفاده کنند.
* نحوه اجرا: Multiprogramming معمولاً از یک CPU برای اجرای چندین برنامه استفاده می کند، در حالی که Multitasking را می توان با استفاده از چندین CPU یا یک CPU واحد که به سرعت بین وظایف سوئیچ می کند، پیاده سازی کرد.
* تعامل کاربر: در Multiprogramming، کاربران معمولاً با یک برنامه در یک زمان تعامل دارند. در مقابل، Multitasking، کاربران می توانند با چندین برنامه به طور همزمان تعامل داشته باشند و به سرعت بین آنها جابجا شوند.

2. مفهوم critical section را توضیح دهید.

در سیستم عامل، یک بخش بحرانی به بخشی از کد یا بخشی از یک برنامه اشاره دارد که به یک منبع مشترک مانند یک متغیر مشترک یا یک فایل مشترک دسترسی دارد که می تواند توسط چندین processes concurrent یا thread همزمان modified شود. بخش بحرانی یک مفهوم اساسی در برنامه نویسی concurrent است، زیرا اطمینان از دسترسی به منبع مشترک به شیوه ای ایمن و سازگار توسط چندین processes یا thread ضروری است. مشکل بخش بحرانی زمانی ایجاد می شود که چندین فرآیند یا رشته سعی می کنند به طور همزمان به منبع مشترک دسترسی پیدا کنند. این می تواند منجر به شرایط race، ناسازگاری داده ها و سایر مشکلات همگام سازی شود. برای جلوگیری از چنین مشکلاتی، سیستم عامل مکانیسم هایی را فراهم می کند تا اطمینان حاصل شود که تنها یک فرآیند یا رشته می تواند در یک زمان به بخش بحرانی دسترسی داشته باشد. یکی از راه حل های رایج برای مشکل بخش بحرانی استفاده از mutual exclusion (حذف متقابل) است، که تضمین می کند که تنها یک فرآیند یا رشته می تواند بخش بحرانی را در یک زمان اجرا کند. سیستم عامل‌ها همگام‌سازی‌های گوناگونی مانند lock، سمافورها و مانیتورها برای اجرای mutual exclusion فراهم می‌کنند. هنگامی که یک فرآیند یا رشته می خواهد وارد بخش بحرانی شود، ابتدا باید lock یا سمافور مرتبط با بخش بحرانی را بدست آورد. اگر lock یا semaphore قبلاً توسط فرآیند یا رشته دیگری نگه داشته شده باشد، فرآیند یا رشته درخواست کننده مسدود می‌شود تا زمانی که lock یا semaphore در دسترس قرار گیرد. هنگامی که process یا نخ بخش بحرانی را کامل می کند، lock یا semaphore را آزاد می کند و به process یا نخ دیگری اجازه می دهد تا وارد بخش بحرانی شود. بخش بحرانی برای اطمینان از صحت و سازگاری برنامه های همزمان ضروری است. با استفاده از مکانیسم های حذف و هماهنگ سازی متقابل، سیستم عامل می تواند از شرایط race جلوگیری کند و اطمینان حاصل کند که منابع مشترک به روشی ایمن و سازگار قابل دسترسی هستند.

3. شرایطی که باعث به وجود آمدن بن بست (deadlock) میشود را نام برده و توضیح دهید.

deadlock وضعیتی است در برنامه نویسی concurrent و زمانی رخ می دهد که دو یا چند process یا رشته مسدود می شوند و به طور نامحدود منتظر یکدیگر می مانند تا منابع را آزاد کنند.

deadlock زمانی رخ می دهد که چهار شرط موجود باشد:

* Mutual Exclusion: منابع درگیر در deadlock باید غیرقابل اشتراک باشند. یعنی فقط یک process می تواند در هر زمان از یک منبع استفاده کند.
* Hold and Wait: یک process باید حداقل یک منبع را در خود نگه دارد و منتظر باشد تا منابع دیگری را که توسط processهای دیگر نگهداری می شود به دست آورد.
* Preemption Non: منابع را نمی‌توان به زور از process که آنها را نگه می‌دارد حذف کرد. یک process فقط می تواند منابع را به صورت داوطلبانه آزاد کند.
* Circular Wait: یک زنجیره دایره ای متشکل از دو یا چند process یا رشته وجود دارد که در آن هر process، یک یا چند منبع را در خود نگه می دارد که توسط process بعدی در زنجیره درخواست شده است.

برای مثال، سیستمی را با دو منبع A و B و دو process P1 و P2 در نظر بگیرید. فرض کنید P1 منبع A را نگه می دارد و منتظر منبع B است، در حالی که P2 منبع B را نگه می دارد و منتظر منبع A است. در این شرایط هر دو process مسدود شده و به طور نامحدود منتظر می مانند تا process دیگر منبع را آزاد کند و در نتیجه به deadlock می رسد. deadlock میتواند در شرایط مختلفی رخ دهد، مانند:

* رقابت بر سر منابع: هنگامی که چندین process یا رشته برای مجموعه محدودی از منابع با هم رقابت می کنند. اگر منابع به شیوه ای امن تخصیص داده نشود، ممکن استdeadlock رخ دهد.
* Synchronization نادرست: زمانی که از اصول اولیه همگام سازی مانند lock ها، سمافورها و مانیتورها به درستی استفاده نشود، deadlock ممکن است رخ دهد.
* Circular Dependencies: زمانی که processها یا رشته ها به صورت دایره ای به یکدیگر وابسته باشند، اگر وابستگی ها به درستی مدیریت نشوند، deadlock می تواند رخ دهد.
* Deadlock Prevention Algorithms: زمانی که الگوریتم‌های پیشگیری به درستی اجرا نشوند، ممکن است deadlock رخ دهند.

برای جلوگیری از deadlock می توان از رویکردهای مختلفی مانند deadlock avoidance، deadlock detection و deadlock recovery استفاده کرد. این رویکردها تضمین می‌کنند که چهار شرط بن‌بست برآورده نمی‌شوند و منابع به شیوه‌ای امن و درست تخصیص می‌یابند.

4. variables Atomic چیست و کجا استفاده میشود؟

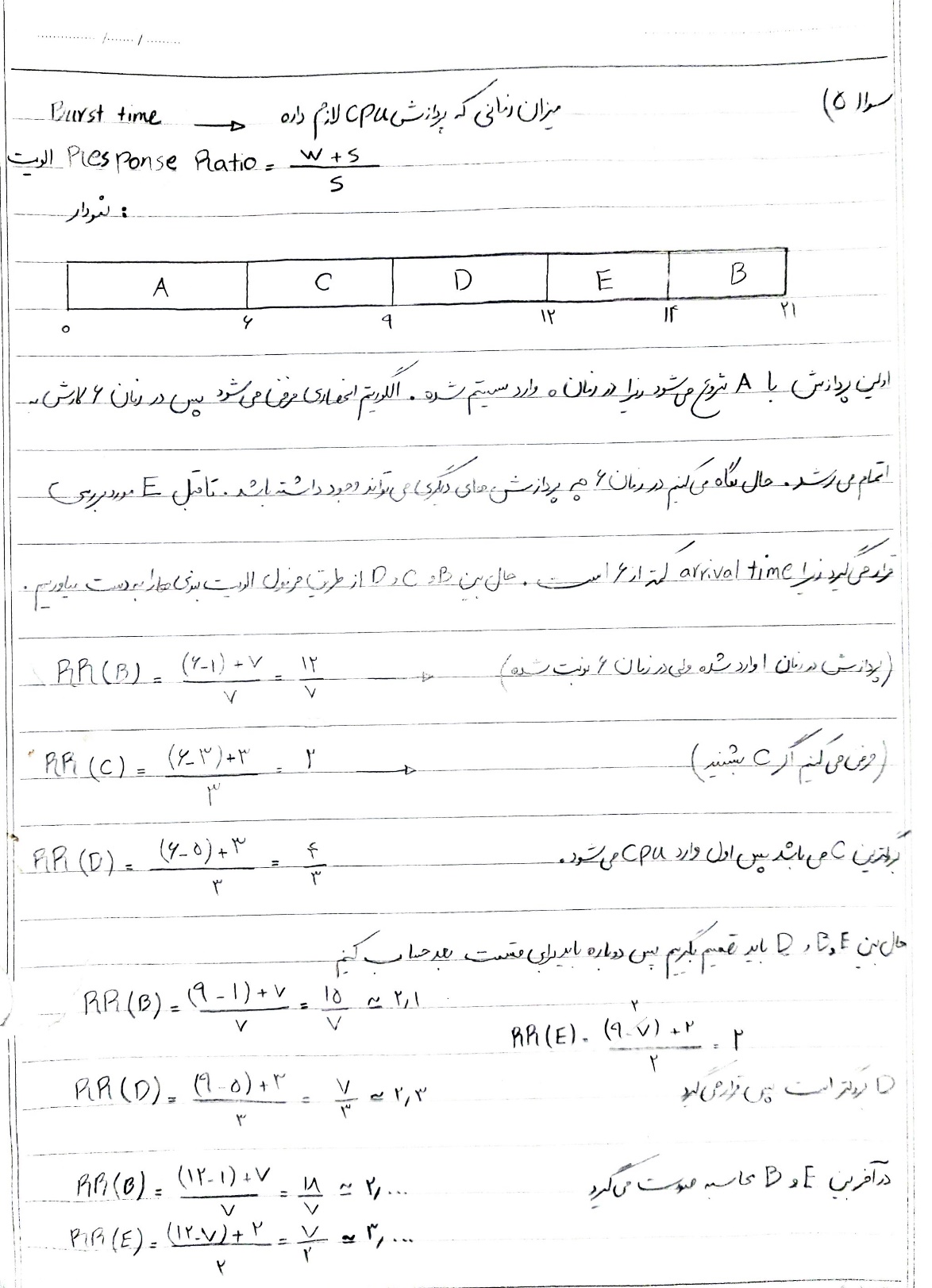
Atomic variables متغیرهایی هستند که دسترسی به آنها به گونه ای است که ثبات را تضمین کرده و از شرایط race در برنامه‌ریزی concurrent جلوگیری می کند. از آنها در برنامه‌های multithreaded استفاده می شود که در آن، چندین رشته ممکن است به طور همزمان به یک متغیر دسترسی داشته باشند. بدون مکانیسم‌های synchronization مناسب، مانند Atomic variables، این دسترسی‌های همزمان می‌تواند به شرایط race منجر شود، تا جایی که مقدار متغیر می‌تواند ناسازگار یا نامشخص شود. Atomic variable ها تضمین می کنند که عملیات روی یک متغیر به عنوان یک عملیات منفرد و غیرقابل تقسیم، بدون دخالت رشته های دیگر اجرا می شود. این امر ثبات را تضمین می کند و با اطمینان از اینکه فقط یک رشته می تواند متغیر را در یک زمان تغییر دهد، از شرایط race جلوگیری می کند. سیستم‌ عامل ها انواع مختلفی از Atomic variables، مانند atomic types، mutexes و semaphores را ارائه میدهند. این مکانیسم ها تضمین می کنند که عملیات اتمی روی متغیرها به گونه ای اجرا می شود که از شرایط race جلوگیری کرده و ثبات را تضمین می کند و کارایی و صحت برنامه های همزمان را بهبود می بخشد.

از کاربردهای اصلی Atomic variable ها میتوان به موارد زیر اشاره کرد:

* Counters: از Atomic variable می توان برای پیاده سازی شمارنده هایی استفاده کرد که توسط رشته های متعدد افزایش یا کاهش می یابند. این تضمین می کند که شمارنده حتی در صورت دسترسی همزمان ثابت و دقیق باقی بماند.
* Flags: می‌توان برای پیاده‌سازی Flag ‌هایی استفاده کرد که نشان‌دهنده درست یا نادرست بودن یک شرط خاص است. به عنوان مثال، یک Flag می تواند برای نشان دادن اینکه یک منبع برای استفاده توسط چندین رشته در دسترس است استفاده شود.
* Pointers: Atomic variables را می توان برای تغییر اشاره گرها به منابع مشترک استفاده کرد و اطمینان حاصل کرد که منابع به شیوه ای ایمن و سازگار توسط رشته های متعدد قابل دسترسی هستند.

بخش دوم) سوالات مسئله‌ای

5. پراسس‌های زیر را با مشخصات ذکر شده در نظر بگیرید و آنها را بر اساس الگوریتم HRRN زمانبندی کنید.



6. دو پراسس مستقل A,B را درنظر بگیرید که به سمافورهای S1=2,S2=1 و نیز مستقل a=1 دسترسی دارند و به طور همزمان روی یک پردازنده اجرا میشوند. با توجه به اطلاعات جدول به سوالات زیر پاسخ دهید.

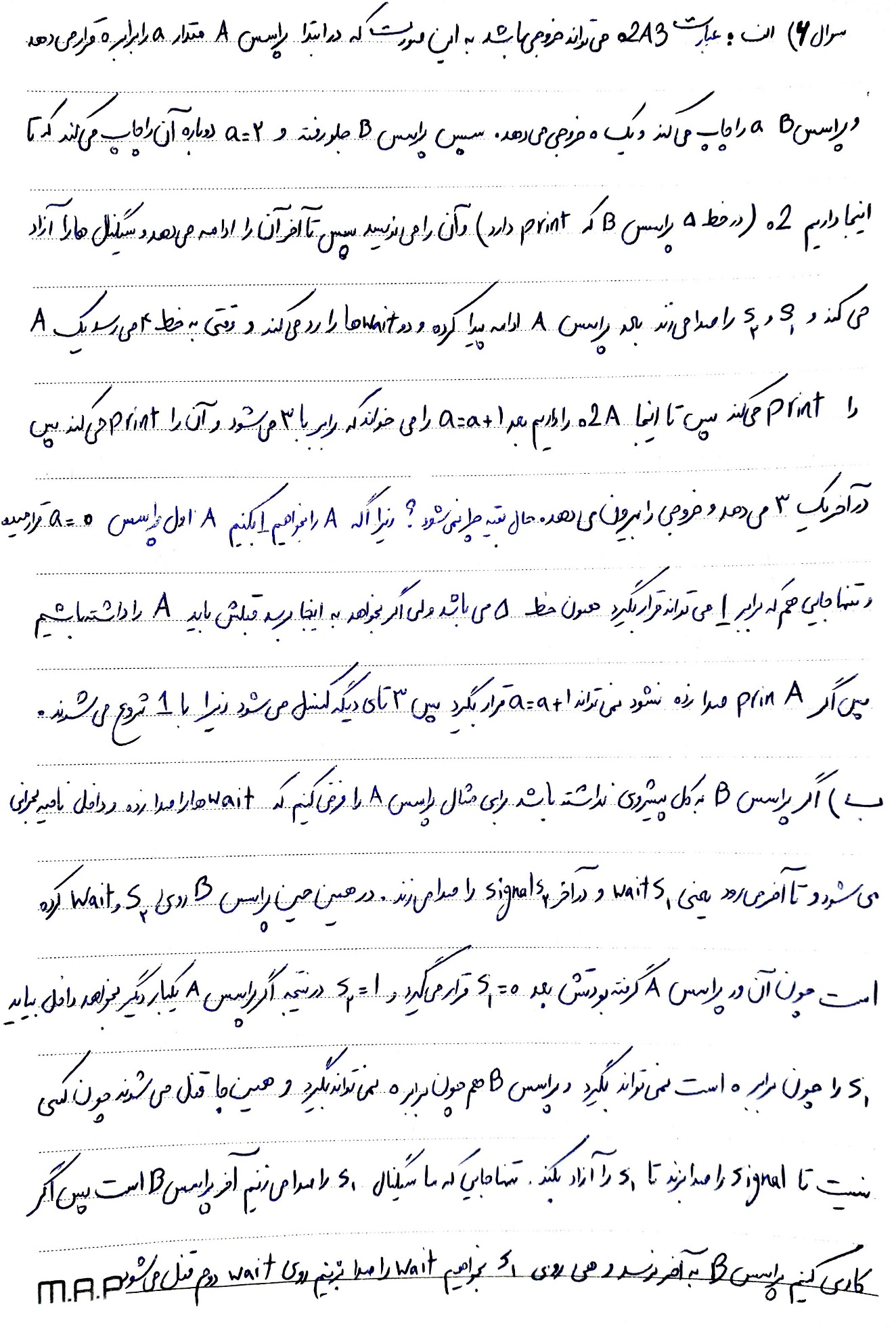
الف) روند فرآیند A:

* + A کاراکتر "A" و مقدار ۱ (زیرا مقدار اولیه a برابر ۱ است) را چاپ می‌کند.
  + فرآیند B منتظر می ماند تا سمافور S2 سیگنال داده شود.
  + فرآیند A منتظر می ماند تا سمافور S1 سیگنال داده شود.
  + فرآیند A سمافور S2 را سیگنال می‌دهد.

روند فرآیند B :

* + B مقدار 1 (مقدار اولیه a) را چاپ می کند.
  + B منتظر می ماند تا سمافور S1 سیگنال داده شود.

فرآیند A منتظر می ماند تا سمافور S1 سیگنال داده شود (از قبل سیگنال داده شده است، بنابراین بدون انتظار ادامه می دهد). فرآیند A مقدار 1 (بدون تغییر) را چاپ می کند. فرآیند A به سمافور S2 سیگنال می دهد. خروجی این دنباله برابر است با 02A3 به شرح زیر است :



ب) برای اینکه یکی از فرآیندها به بن بست برسد و بیشتر پیش نرود، باید شرایط زیر رعایت شود:

1. فرآیند A ابتدا شروع می شود و به انتظار خط (s1) می رسد.
2. فرآیند B پس از آن شروع می شود و به انتظار خط (s2) می رسد.
3. فرآیند A سمافور S1 را به دست می آورد و به خط بعدی ادامه می دهد.
4. فرآیند B تلاش می کند سمافور S2 را به دست آورد، اما نمی تواند این کار را انجام دهد زیرا فرآیند A سمافور S2 را در اختیار دارد و هنوز آن را منتشر نکرده است.
5. فرآیند B مسدود می شود و تا زمانی که سمافور S2 در دسترس نباشد نمی تواند ادامه یابد.

در این سناریو، فرآیند B به دلیل مسدود شدن در سمافور S2 قادر به ادامه کار نیست و در نتیجه فرآیند B به بن بست می رسد. در همین حال، فرآیند A می تواند به اجرای خود ادامه دهد. خروجی سیستم در این مورد به اجرای کد بعدی فرآیند A پس از بدست آوردن سمافور S1 بستگی دارد. زمانی که فرآیند B به بن بست می رسد نمی توان خروجی خاص سیستم را تعیین کرد.

ج) مشکلات انحصاری متقابل، که به عنوان شرایط مسابقه نیز شناخته می‌شوند، زمانی رخ می‌دهند که چندین فرآیند یا رشته به طور همزمان و بدون همگام‌سازی مناسب به منابع مشترک دسترسی داشته باشند. در سیستم داده شده، منبع مشترک متغیر "a" و سمافورهای S1 و S2 است.

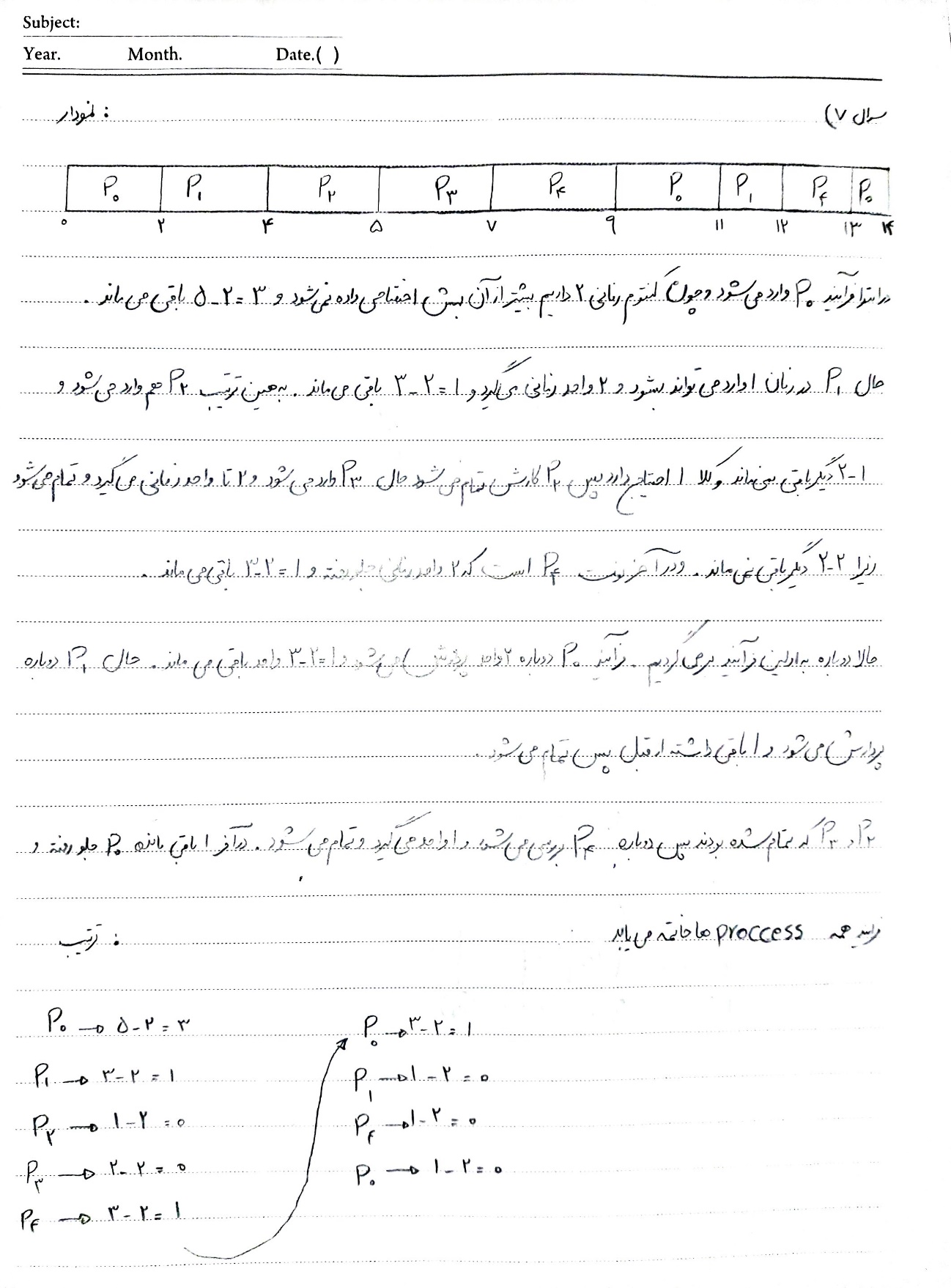
سناریوی زیر را در نظر بگیریم که می تواند منجر به شرایط مسابقه شود:

1. فرآیند A و فرآیند B تقریباً به طور همزمان شروع می شوند.
2. فرآیند A به خط چاپ (a) می رسد و سعی می کند مقدار "a" را چاپ کند.
3. در همان زمان، فرآیند B به خط چاپ (a) می رسد و همچنین سعی می کند مقدار "a" را چاپ کند.
4. هر دو فرآیند به طور همزمان به متغیر مشترک "a" دسترسی پیدا می کنند که منجر به شرایط مسابقه می شود.

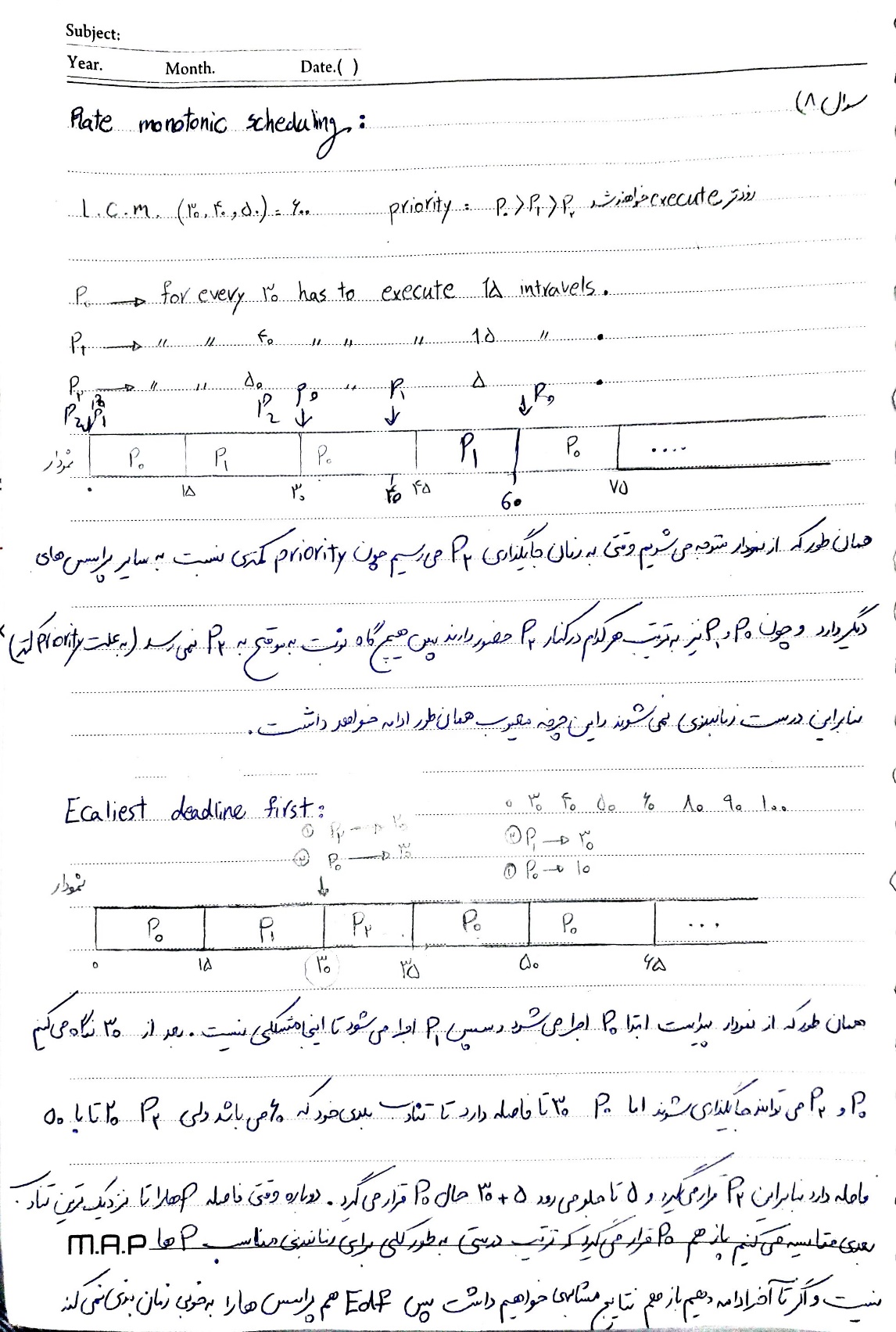
خروجی در این مورد به زمان بندی دقیق و ترتیب اجرای فرآیندها بستگی دارد. نتایج احتمالی می تواند این باشد:

1. فرآیند A مقدار اولیه "a" (1) را می خواند و آن را چاپ می کند و سپس فرآیند B همان مقدار اولیه "a" (1) را می خواند و چاپ می کند. خروجی می تواند "1 1" باشد.
2. فرآیند B مقدار اولیه "a" (1) را می خواند و آن را چاپ کرده و سپس فرآیند A همان مقدار اولیه "a" (1) را می خواند و چاپ می کند. خروجی می تواند "1 1" باشد.
3. فرآیند A مقدار اولیه "a" (1) را می خواند و آن را چاپ می کند. سپس، قبل از اینکه فرآیند B بتواند مقدار "a" را بخواند، فرآیند A اجرای خود را کامل می کند و مقدار "a" را به 0 یا 2 به روز می کند. سپس فرآیند B مقدار به روز شده "a" را می خواند و چاپ می کند. خروجی می تواند " 10" یا " 12" باشد.
4. فرآیند B مقدار اولیه "a" (1) را می خواند و آن را چاپ می کند. سپس، قبل از اینکه فرآیند A بتواند مقدار "a" را بخواند، فرآیند B اجرای خود را کامل می کند و مقدار "a" را به 0 یا 2 به روز می کند. سپس فرآیند A مقدار به روز شده "a" را می خواند و چاپ می کند. خروجی می تواند " 10" یا " 12" باشد.

7. پراسس های زیر را با مشخصات ذکر شده در نظر بگیرید و آنها را بر اساس الگوریتم Round Robin با کوانتوم زمانی ۲ واحد زمانی زمانبندی کنید.

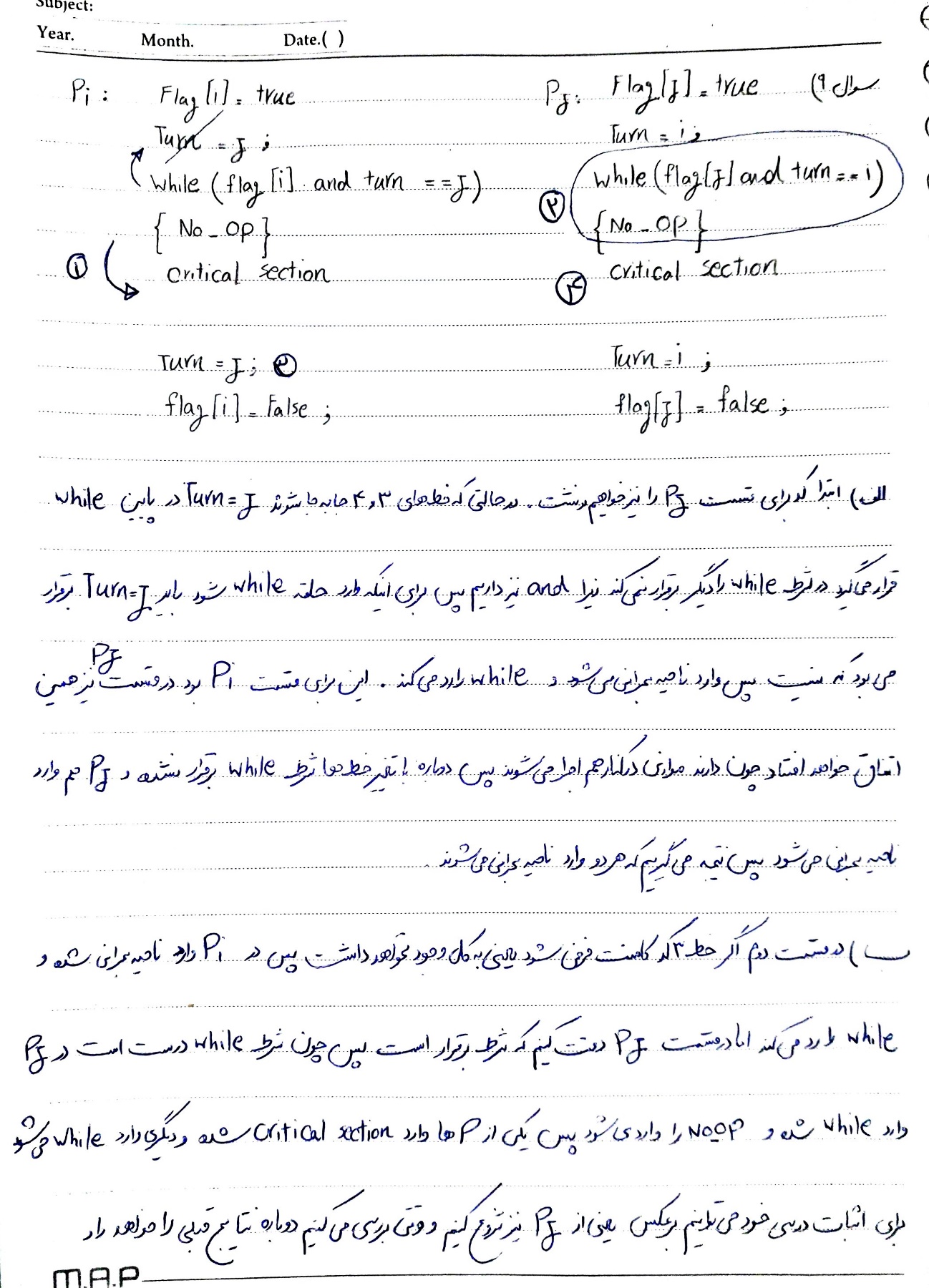


8. پراسس های زیر را با مشخصات ذکر شده در نظر بگیرید و آنها را با استفاده از دو الگوریتم EDF , RMS زمان بندی کنید و نتایج را با هم مقایسه کنید. آیا دو الگوریتم میتوانند این دو پراسس را به درستی زمان بندی کنند ؟





9. الگوریتم زیر برای ایجاد برقراری شروط مورد نیاز ناحیه بحرانی بین دو فرایند Pi , Pj نوشته شده است. برای هر یک از حالت های زیر آن را تحلیل کنید و در صورت برقرار نبودن هر یک ازشرط های مورد نیاز برای ناحیه بحرانی ، دلیل آن را بیان کنید .



:منابع

<https://www.youtube.com/watch?v=xgW8VhEOpFg>

<https://www.youtube.com/watch?v=ejPXTOcMRPA>

<https://www.youtube.com/watch?v=z2bMgy8_LIk>

<https://www.youtube.com/watch?v=-jFGYDfWkXI>

<https://en.wikipedia.org/wiki/Round-robin_scheduling>

<https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/>