**فایل Segment.java**

این کلاس مربوط به قطعات حافظه میشود که بر حسب نیاز بین فرآیند ها تقسیم میشوند. برای object های این کلاس سایز قطعه حافظه(size)، آدرس شروع قطعه(initAddress)، فضایی که فرآیند از این قطعه اشغال کرده(usedSize) و همینطور id فرآیندی که این قطعه به آن اختصاص پیدا کرده(pdi) درنظر گرفته شده است. برای مثال اگر فرآیندی که درخواست 19KB داده مقدار usedSize برابر 19KB است و مقدار size برابر 32 خواهد بود.

**تابع assignSegment** تابعی است که موقعی که این segment به فرآیندی اختصاص داده میشود اجرا میشود و مقادیر pid و usedSize مقدار دهی میشوند. تابع releaseSegment تقریبا کار برعکس تابع assignSegment را انجام میدهد و مقدار pid و usedSize را صفر میکند تا این segment به فرآیندی اختصاص نداشته باشد. توابع دریگر getter های property های این کلاس هستند که از توضیح آنها صرف نظر میشود.

**فایل ReadWriteLock.java**

این کلاس برای ایجاد ناحیه بحرانی ایجاد شده که در آن از تعریف تابع با کمک synchronized استفاده شده. **تابع lockRead** برای ایجاد قفل برای reader هاست و اگر writer و یا درخواست write در سیستم موجود باشد reader باید wait کند. **تابع unlockRead** برای آزاد کردن قفل reader استفاده شده که یکی از تعداد reader ها کم میکند و تمام رشته های در انتظار را بیدار میکند تا اگر رشته ای شرایط اجرا شدن دارد، اجرا شود. **تابع lockWrite** نیز به شرط اینکه تعداد reader ها و writer ها صفر باشد یکی از تعداد درخواست های write کم میکند و تعداد writer ها رو یکی افرایش میدهد در غیر اینصورت wait میکند. **تابع unlockWrite** یکی از تعداد writer ها کم میکند و تمام رشته های در انتظار را بیدار میکند تا اگر رشته ای شرایط اجرا شدن دارد، اجرا شود.

**فایل Memory.java**

این کلاس برای شبیه سازی memory نوشته شده. برای هر object از این کلاس سایز کل حافظه(MEMORY\_SIZE)، یک آرایه از Segment ها به نام segments که تمام قطعه های حافظه را درون خود ذخیره میکند، حداقل سایز قطعه های حافظه(MIN\_SEG\_SIZE) که با توجه به متن پروژه 32 در نظر گرفته شده، یکReaderWriterLock که برای ایجاد محافظت برای آرایه segments استفاده میشود.

ابتدا توابعی که در تابع allocate استفاده شده شرح داده میشوند.

**تابع checkPocessSpace** چک میکند که فرآیند درخواستی داده باشد که با حافظه های فعلی که در اختیار دارد نتواند آن را رفع کند. یعنی اگر از حافظه 32KB خود تنها 10KB استفاده کرده باشد حق ندارد برای فضای 20KB ای درخواست دهد. اگر این نیاز قابل رفع باشد مقدار usedSize در segment مربوطه افزایش میابد و true برگردانده میشود تا نشان دهیم کار تخصیص توانشته انجام شود در غیر اینصورت در نهایت false برگردانده میشود.

**تابع splitMemorySegment** یک اندیس آرایه segments را میگیرد و آن segment را به قطعات کوچکتر تقسیم کند تا جایی که به سایز needeSize برسد. اندیس segment ای که باید تقسیم شود توسط segmentIndex به تابع داده میشود. و تا زمانی که سایز مورد نظر نرسیده باشیم segment فعلی remove میشود و به جای آن دو segment با سایز نصف درون segments قرار میگیرند. در هر مرحله نیز اگر به سایز مورد نظر رسیده باشیم return میکنیم.

**تابع findOptimumSize** که وظیفه پیدا کردن سایز segment بهینه را دارد با توجه به اندازه مورد نیاز(size) هر بار سایز کل حافظه را نصف میکند و با سایز درخواستی مقایسه میکند و باتوجه به اینکه سایز از مقدار 32KB نباید کمتر شود، نزدیک ترین سایز مورد نظر را برای سایز درخواستی برمیگرداند.

**تابع allocate** وظیفه پیدا کردن و نسبت دادن segment ها به فرآیند را دارد. این تابع ابتدا با کمک تابع checkProcessSpace بررسی میکند که درخواست مجاز باشد. و اگر خروجی تابع true باشد پیغام مناسب چاپ میکند. در غیر اینصورت پس از گرفتن lockWriter به جستوجوی segment مناسب میپردازد و اگر آنرا پیدا کرد آنرا به عنوان segment منتخب برمیگرداند. در این مراحل جستوجو دو متغیر splitCandidateIndex و splitCandidateSize را به روزرسانی میکند تا درصورتی که segment مورد نظر وجود نداشت آنرا با تابع splitMemorySegment بسازد. با توجه به splitCandidateSize اگر segment ای ببینیم که استفاده نشده و سایز کمتری دارد یعنی segment مناسب تری برای تقسیم است پس مقادیر را به روز میکنیم. در نهایت اگر segment مناسب پیدا نشد تابع splitMemorySegment را فراخوانی میکنیم و segment با سایز مناسب را بسازد و در نهایت lockWriter را آزاد میکند. اگر هم این segment چه برای تخصیص و چه برای split کردن پیدا نشود خطای NotEnoughMemoryError میدهد و پیغام مناسب نیز چاپ میشود.

**تابع deallocate** با دریافت pid و initAddress به جستوجوی segment مربوطه میپردازد و با پیدا کردن آن تابع releaseSegment را برای آن segment فرامیخواند تا segment آزاد شود.

**تابع deallocateAll** مشابه تابع deallocate عمل میکند با این تفاوت که تمام segment های مربوطه به pid را releaseSegment میکند. این تابع موقع اتمام فرآیند استفاده میشود.

**فایل Process.java**

این کلاس که از کلاس Thread ارث بری میکند برای شبیه سازی کار های یک فرآیند ایجاد شده. Object های این کلاس ویژگی های pid، زمان شروع(initTime)، وضعیت اجرای فرآیند(isTerminated)، زمان پایان یافتن(terminatedtime)، آرایه ای از segment های(segments)، memory و حداقل زمان بین عملیات های مختلف فرآیند(MIN\_INTERVAL\_DELAY) را دارا هستند.

مهم ترین تابع این کلاس **تابع run** است. در این تابع یک while(true) قرار دارد که تا زمانی که فرآیند حضور دارد این حلقه اجرا میشود. درون این حلقه ابتدا یک عدد رندوم بین 0 یا 1 یا 2 تولید میشود که به معنی این است که فرآیند در این دور از حلقه کدام عملیات را باید انجام دهد. 0 برای allocate، 1 برای deallocate و 3 برای اتمام فرآیند است.

برای 0 یک عدد رندوم بین 5 تا MEMORY\_SIZE/4 ایجاد میشود. دلیل تقسیم بر 4 این است که اعداد خیلی بزرگ نباشند. سپس از memory به مقدار آن عدد رندوم allocate میکنیم و segment ای که انتخاب میشود را در segments قرار میدهیم. برای 1 بین segment های موجود در segments یکی را به صورت رندوم انتخاب میکنیم و از memory آنرا deallocate میکنیم. برای 2 نیز اگر فرآیند segment نداشته باشد و یا با شانس 50 درصد(nextInt(2)) فرآیند باقی میماند در غیر اینصورت فرآیند terminate میشود.

در انتهای این دور از حلقه نیز یک عدد رندوم برای تاخیر تا عملیات بعدی ایجاد میشود.

**تابع terminate** برای پاک کردن کل segment های فرآیند و مقدار دهی isTerminated و deallocateAll استفاده میشود.

**تابع getusedSpace** تمامی usedSize های segment های این فرآیند را جمع میکند و برمیگرداند. تابع getInternalFragmentation نیز از هر segment مربوط به فرآیند، مقداری که استفاده نشده را جمع میکند و برمیگرداند.

**فایل ProcessPool.java**

این فایل مجموعه ای از process ها را نگه میدارد و وظیفه log کردن memory را هر 5 ثانیه برعهده دارد.

در constructor این کلاس به تعداد processCount فرآیند ایجاد میکنیم. و pid هرکدام را از 1000 تا 1000+processCount-1 قرار میدهیم و آنها را در processes که آرایه ای از فرآیند ها است اضافه میکنیم. پس از اینکار تمام فرآیند ها را start میکنیم تا شروع شوند.

در **تابع run** این کلاس یک while(true) قرار دارد که در هر دور برای هر فرآیند مقدار usedSpace و internalFragmentation را محاسبه و جمع میکند تا ای مقادیر برای کل حافظه بدست بیاید. سپس **تابع log** فراخوانی میشود تا این مقادیر چاپ شوند و علاوه بر آن برای همه فرآیند ها مقادیر pid، usedSpace، runtime و terminatedTime چاپ شود. در نهایت نیز آرایه segments متعلق به memory چاپ میشود تا وضعیت تمام segment ها قابل مشاهده باشد. در هر دور از حلقه اگر تمام فرآیند ها terminate شده بودند تابع run متوقف میشود و برنامه به اتمام میرسد.

**فایل Main.java**

این فایل اصلی برنامه است که یک object از کلاس Memory به سایز 1024KB و یک object از کلاس ProcessPool میسازد و آنرا start میکند.

**نمونه خروجی برنامه**

خروجی زیر برای تعداد 3 فرآیند و 1024KB حافظه و تصاویر زیر عکس از خروجی کنسول است که به ترتیب در چند عکس قرار گرفته.







