

به نام خدا
گزارش پروژه درس بی‌درنگ
گروه ۱۶

توضیحات پیاده‌سازی

وظیفه‌ها:

برای ایجاد هر وظیفه، کلاس‌های زیر پیاده‌سازی شده‌اند؛

❖ کلاس `base_task`: کلاس پایه وظیفه‌هایمان می‌باشد و شامل اطلاعاتی درباره تناوب و سطح بحرانیت هر وظیفه است. همچنین برای آن یک سری متودها به صورت انتزاعی تعریف شده‌اند که در کلاس‌های یکه از آن ارث‌بری می‌کنند، در ادامه توضیح داده خواهند شد. تابع `get_utilization`، بهره‌وری وظیفه که حال تقسیم‌زمان اجرا بر تناوب است را برمی‌گرداند.

❖ کلاس `high_critically_task`: از کلاس `base_task` ارث‌بری می‌کند و علاوه بر اطلاعات کلاس پدر، شامل اطلاعاتی نظیر زمان اجرای بزرگ و کوچک و ددلاین مجازی می‌باشد. برای راحتی پیاده‌سازی کد، تابع `get_computation_time` پیاده‌سازی شده و بر اساس حالت فعلی، زمان اجرا را برمی‌گرداند. (در حالت نرمال -> زمان اجرای کوچک، در حالت سرریز -> زمان اجرای بزرگ) همچنین تابع `get_deadline` نیز ددلاین وظیفه را براساس حالت فعلی برمی‌گرداند. (در حالت نرمال -> ددلاین مجازی، در حالت سرریز -> ددلاین اصلی)

❖ کلاس `low_critically_task`: از کلاس `base_task` ارث‌بری می‌کند و تابع `get_computation_time` زمان اجرای آن و تابع `get_deadline` زمان ددلاین آن را برمی‌گرداند.

حال برای ساخت این وظایف، از الگوریتم `uunifast` استفاده شده که به تشریح این الگوریتم می‌پردازیم. در این الگوریتم مقدار بهره‌وری کلی وجود دارد و برای هر وظیفه یک بهره‌وری رندوم تعریف می‌شود. دقت می‌شود که مجموع بهره‌وری همه وظایف کمتر مساوی بهره‌وری کل شود و ددلاین هر وظیفه نیز کمتر مساوی بهره‌وری هر هسته شود. از آنجایی که همه هسته‌ها همگن می‌باشند، بهره‌وری کل از حاصلضرب تعداد هسته‌ها در بهره‌وری هر هسته محاسبه می‌شود.

سپس وظایف تولید شده را به نسبت گفته شده برای وظایف با سطح بحرانیت بالا به وظایف با سطح بحرانیت پایین تقسیم بندی می‌کنیم. سپس یک عدد رندوم برای تناوب هر وظیفه نیز تولید می‌کنیم. حال با توجه به اینکه بهره‌وری از حاصل تقسیم زمان اجرا (زمان اجرای کوچک برای وظایف با سطح بحرانیت بالا) بدست می‌آید، این مقدار را محاسبه کرده و برای وظایف با سطح بحرانیت بالا نیز یک عدد رندوم با بازه‌های مشخص برای زمان اجرای بزرگ تولید می‌کنیم.

منابع:

برای منبع یک مدل به نام resource تعریف شده است و شامل اطلاعاتی درباره ظرفیت منبع (اینکه از آن منبع در کل چند واحد موجود است) و سطح دسترسی آن در هر لحظه (اینکه در حال حاضر آزاد است یا خیر) می‌باشد. برای ظرفیت منبع یک عدد رندوم تولید می‌شود که این کار در resource generation انجام می‌شود.

ادغام وظایف تولید شده با منابع درخواستی:

حال برای وظایف شده باید تعیین کنیم که هر وظیفه نیاز به کدام منابع برای انجام دارد و هر منبع را برای چند واحد زمانی نیاز دارد. برای این کار در ماژول resource_usage با تقسیم بندی زمان اجرای هر وظیفه بین منابع موجود، این امر را محقق کرده‌ایم. دقت شود که لزوماً هر وظیفه، همه منابع را نیاز نخواهد داشت و واحد زمانی مورد نیاز برای منابعی که از آن استفاده نمی‌کند، برابر با صفر خواهد بود.

هسته ها:

برای هسته ها، یک مدل به نام core تعریف شده است و شامل اطلاعاتی درباره وظایفی که انجام داده، وظایفی که به آن نگاشت شده است، ازدحام منابع بر روی آن و بهره‌وری هسته می‌باشد. نحوه مقدار دهی به این فیلد ها در ادامه در ماژول simulator توضیح داده شده است.

اکنون که وظایف تولید شده اند و منابع و هسته ها نیز مشخص هستند، نوبت به شبیه سازی می‌رسد. برای این امر، ماژول simulator نوشته شده است و وظایف و منابع و هسته ها به آن داده می‌شود تا کار شبیه سازی را شروع کند. این ماژول در ادامه تشریح می‌شود. همچنین برای تست این ماژول و گزارش دهی نتایج، ورودی هایی به برنامه داده می‌شود که این ورودی ها به فرمت json در دایرکتوری inputs قابل مشاهده هستند.

ماژول Simulator

پیاده سازی اصلی الگوریتم زمان بندی و نگاشت به هسته ها را در اینجا پیاده سازی کردیم. در ابتدا در مقدار دهی این تسک های با بحرانیّت بالا، تسک ها با بحرانیّت پایین، منابع و هسته ها را میگیریم. همینطور برای کل شبیه ساز یک mode در نظر میگیریم که نشان دهنده وضعیت بحرانیّت سیستم هست. و برای جلو بردن همه تغییرات و نگه داشتن زمان از current_time استفاده میکنیم.

در ابتدا در مقدار دهی این کلاس مقدار x برای EDF-VD را محاسبه میکنیم. برای مصاحبه این مقدار Utilization تسک های low critical و Utilization تسک های high critical را محاسبه میکنیم. سپس از رابطه زیر برای محاسبه x استفاده میکنیم:

$$x \leftarrow \frac{U_{HI}^{LO}(\tau)}{1 - U_{LO}^{LO}(\tau)}$$

در قدم بعدی ددلاین تسک‌های با بحرانیت بالا را محاسبه و مقدار دهی میکنیم. در تابع `update_high_critical_tasks_` این کار را انجام میدهیم.

همینطور یک مپ نگه میداریم که نشان‌دهنده تسک‌های اساین شده به هر هسته میباشد. در هر مرحله شبیه‌سازی اگر تسکی را به یک هسته اساین کردیم این مپ را آپدیت نگه میداریم.

در قدم بعدی جدول مربوط به MSRP را تشکیل میدهیم. در ایجاد این جدول که یک جدول دو بعدی هست تابع `CR` برای برای هر منبع و با `n_k` های مختلف محاسبه میکنیم. در این محاسبه ابتدا ماکسیمم پریود را محاسبه میکنیم. در پیاده‌سازی چون از توابع `max` و `min` استفاده کردیم در صورت لزوم ممکن هست نیاز باشد تا یک دیفالت برای این تابع در نظر بگیریم و چون پریود بیشتر یعنی اولویت کمتر(۰) در نتیجه از ماکسیمم پریود برای دیفالت توابع `max` استفاده میکنیم.

در ادامه برای محاسبه این جدول حساب میکنیم اگر تسکی از این منبع به تعداد بیشتر از `i` استفاده مینمود مینیمم همه پریودهای این جنس تسک‌ها را میگیریم و برابر با آن خانه منبع قرار میدهیم. در واقع در اینجا اولویتی برای هر تسک تعریف نکردیم به صورت جدا بنابراین پریود کمتر معنی اولویت بالاتر میدهد. بدین شکل جدول مربوط به MSRP را تشکیل میدهیم.

همینطور یک `system_preemption_level` تعیین میکنیم که در ابتدا `math.inf` هست. بدلیل استفاده از پریود برای نشان دادن اولویت هر تسک این فیلد ماکسیمم بودنش به معنی کمترین اولویت هست.

در ادامه شبیه‌سازی را با تابع `execute` شروع میکنیم. در این تابع یک `While current_time < time_limit` داریم که درواقع به جای `time_limit` اندازه پنجره شبیه‌سازی ما قرار میگیرد. در ابتدای شبیه‌سازی `system_preemption_level` را با تابع `update_system_preemption_level_` آپدیت مینماییم. در این تابع `preemption_level` هر تسک را میگیریم و بین همه آن‌ها مینیمم میگیریم. که در واقع بالاترین اولویت را بدست میاوریم.

در قدم بعدی `update_mode` را کال میکنیم. در واقع در این تابع بررسی میکنیم که اگر تسکی از جنس بحرانیت بالا وجود داشت و اکتیو بود(جاب فعالی از آن در دست اجرا بود) و همینطور ددلاین آن گذشته بود به حالت `overrun` در میاییم. اگر هم این شرایط برقرار نبود به حالت `normal` میبریم سیستم را.

سپس تابع `handle_done_tasks` صدا زده میشود که برای مدیریت تسک‌های پایان یافته از قبل هست. در این تابع تسک‌هایی که در `currently_assigned_tasks` هستند را بررسی میکنیم و اگر نتیجه متد `is_finished` آن تسک مثبت بود، `job` بعدی آن را میسازیم(این بدان معنی نیست که این جاب لزوما وارد پروسه میشود. بلکه صرفا مشخص میشود وقت انجام جاب بعدی هست و اگر به `release time` مانده باشد تابع `is_active` منفی برخواهد گرداند و استفاده نمیشود تا زمان `release time`).

همینطور اگر تسک از جنس بحرانیت بالا بود و ددلاین آن گذشته بود، سیستم به حالت panic mode میرود و کل شبیه‌سازی به اتمام میرسد. در غیر اینصورت اگر ددلاین یک تسک با بحرانیت پایین گذشته شده بود آن تسک نیز advanced_forward میشود تا جاب بعدی آن آماده شود.

همه تابع‌ها تا این مرحله برای بررسی صحت عملکرد سیستم می‌باشد. در ادامه get_scheduled_tasks کل تسک‌های فعلی را به شرط is_active بودن به ترتیب ددلاینشان مرتب میکنیم. دقت کنید is_active بودن یعنی جاب فعال داشته باشد آن تسک. و ددلاین بر حسب اینکه mode چه باشد برای تسک‌های با بحرانیت بالا فرق خواهد کرد. در نهایت در صورتی که سیستم در مد نرمال باشد تسک‌های مرتب شده با بحرانیت کم و زیاد را باهم جمع کرده و اگر در مد overrun باشد صرفاً تسک‌های با بحرانیت بالا را برای زمان‌بندی برمیگردانیم.

در قدم بعدی تابع assign_to_core صدا زده میشود که در ادامه این تابع ابتدا بررسی میشود اگر در مود overrun بودیم همه تسک‌ها با بحرانیت پایین preempt میشوند. در قدم بعدی تسک‌های مرتب شده برحسب edf-vd به کورها اساین میشوند. که آن هم بدین شکل هست که ابتدا از بین تسک‌های مناسب برای اساین شدن آن‌هایی که در حال حاضر اساین شده‌اند را حذف میکنیم. سپس برای هر تسک قابل اساین شدن بین هسته‌ها ایتريت کرده و چک میکنیم اگر هسته خالی بود یا ددلاین تسک روی هسته بیشتر از هسته در حال اساین شدن بود، آن هسته به لیست هسته‌های قابل استفاده اضافه میشود. دقت کنید که سطح اولویت هر تسک که امکان اساین شدن داشته باشد در مرحله قبلی برحسب اینکه preemptio_level آن تسک بیشتر از preemptio_level سیستم باشد مقایسه شده و فیلتر اعمال شده روی لیست کل تسک‌ها. در نهایت از بین هسته‌های مناسب برای هر تسک congestion را برای هر تسک و هسته محاسبه کرده و همینطور utilization را محاسبه میکنیم. این هسته‌ها را به شرط داشتن utilization مناسب فیلتر کرده و بر اساس congestion مرتب میکنیم. در نهایت بهترین هسته را به آن تسک اساین میکنیم. در قدم آخر تابع calculate کل تسک‌های اساین شده را صدا میزنیم.

نتایج

در کل برای ارزیابی از ۶ سناریو مختلف استفاده کردیم:

1. تعداد وظایف: ۴۰۰، تعداد هسته‌ها: ۲، نسبت وظایف با سطح بحرانیت بالا به وظایف با سطح بحرانیت پایین: برابر، بهره‌وری هسته: ۰.۲۵
2. تعداد وظایف: ۴۰۰، تعداد هسته‌ها: ۲، نسبت وظایف با سطح بحرانیت بالا به وظایف با سطح بحرانیت پایین: برابر، بهره‌وری هسته: ۰.۵
3. تعداد وظایف: ۴۰۰، تعداد هسته‌ها: ۴، نسبت وظایف با سطح بحرانیت بالا به وظایف با سطح بحرانیت پایین: برابر، بهره‌وری هسته: ۰.۲۵
4. تعداد وظایف: ۴۰۰، تعداد هسته‌ها: ۴، نسبت وظایف با سطح بحرانیت بالا به وظایف با سطح بحرانیت پایین: برابر، بهره‌وری هسته: ۰.۵

5. تعداد وظایف: ۴۰۰، تعداد هسته ها: ۸، نسبت وظایف با سطح بحرانیّت بالا به وظایف با سطح بحرانیّت پایین: برابر، بهره وری هسته: ۰.۲۵
6. تعداد وظایف: ۴۰۰، تعداد هسته ها: ۸، نسبت وظایف با سطح بحرانیّت بالا به وظایف با سطح بحرانیّت پایین: برابر، بهره وری هسته: ۰.۵

