گزارش پروژه سوم آزمایشگاه سیستم عامل

پریا خوش‌تاب ۸۱۰۱۹۸۳۸۷ - علی اخگری ۸۱۰۱۹۸۳۴۱ - پرنیان فاضل ۸۱۰۱۹۸۵۱۶

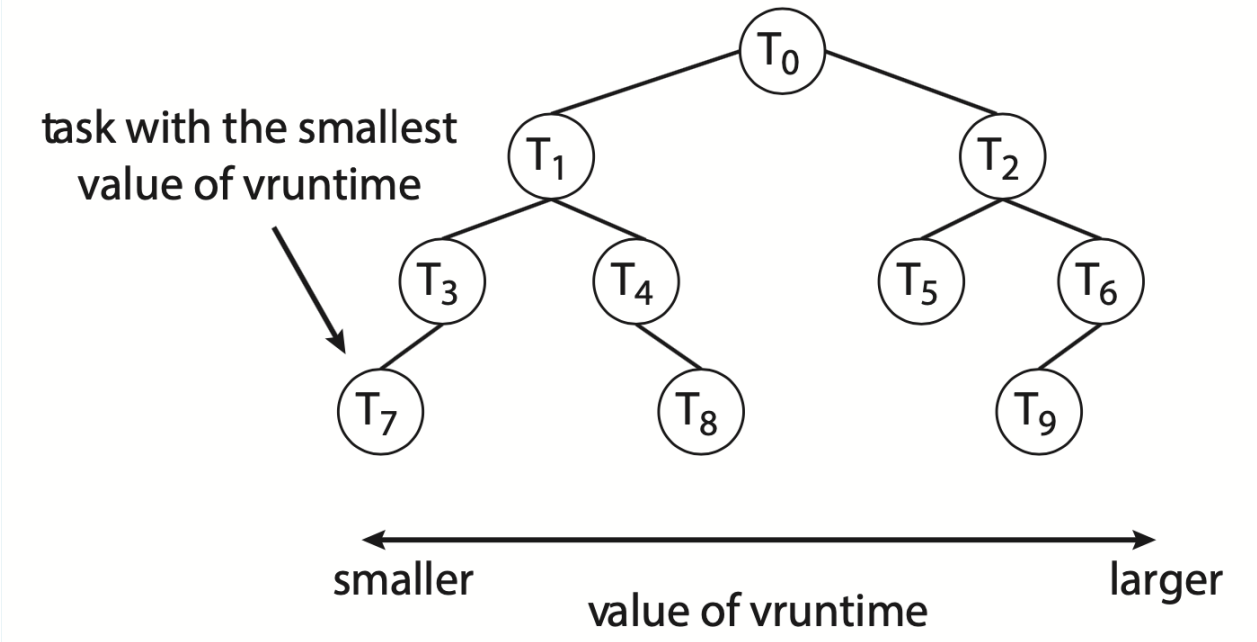
زمان بندی در xv6

سوال ۱) چرا فراخوانی تابع sched() منجر به فراخوانی تابع scheduler() می‌شود؟ (منظور توضیح شیوه اجرای فرایند است.)

در تابع yield وقتی که حالت یک پردازه به RUNNABLE تغییر پیدا میکند، تابع sched صدا زده میشود. در تابع sched ابتدا شروطی مثل lock بودن، running بودن و وجود داشتن امکان وقفه چک می شود و در صورتی که هیچ کدام از این شروط برقرار نباشند با استفاده از تابع swtch عمل context switch انجام می شود و پس از ذخیره کردن context فعلی به scheduler سوییچ می کنیم، پس scheduler جایگزین پردازه فعلی می شود. در scheduler پردازه مناسب توسط الگوریتم RR مشخص می شود و cpu در اختیار آن قرار میگیرد. توجه شود که برای این که پردازه مناسب توسط RR یافت شود، روی صف پردازه ها (ptable.proc) پیمایش می کنیم و پردازه ای که در وضعیت RUNNABLE باشد را به عنوان پردازه بعدی که باید اجرا شود، در نظر میگیریم.

سوال ۲) صف پردازه‌هایی که تنها منبعی که برای اجرا کم دارند پردازنده است، صف آماده یا صف اجرا نام دارد. در xv6 صف آماده مجزا وجود نداشته و از صف پردازه‌ها بدین منظور استفاده می‌گردد. در زمان‌بند کاملا منصف در لینوکس، صف اجرا چه ساختاری دارد؟

لینوکس برای اینکه مشخص کند چه پردازه ای برای باید در ادامه اجرا شود، از داده ساختار red-black tree استفاده می شود. به این صورت که هر پردازه در این درخت قرار داده می شود که کلید آن بر اساس مقدار vruntime است. vruntime یا virtual runtime، برای اولویت بندی پردازه ها استفاده می شود، به این صورت که در هر بار زمان بندی، پردازه ای که کمترین vruntime داشته باشد اجرا می شود و همچنین هر چه اولویت پردازه بالاتر باشد، زمان vruntime آن به صورت کندتر افزایش می یابد.



سوال ۳) همانطور که در پروژه اول مشاهده شد، هر هسته پردازنده در xv6 یک زمان‌بند دارد. در لینوکس نیز به همین گونه است. این دو سیستم عامل را از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی بررسی نمایید. و یک مزیت و یک نقص صف مشترک نسبت به صف مجزا را بیان کنید.

در xv6 همه پردازه ها در یک صف مشترک قرار می‌گیرند. این صف proc نام داشته و در استراکت ptable قرار دارند. این ساختمان داده با استفاده از یک قفل، دسترسی های همزمان را کنترل میکند. در هنگام زمان‌بندی در این صف پیمایش میکنیم و الگوریتم RR روی همین صف اجرا می‌شود.

در سیستم عامل لینوکس هر پردازنده یک صف مخصوص به خودش دارد.

مزیت: هنگامی که از صف مجزا استفاده می‌شود، مسئله‌ای که وجود دارد Load Balancing است. در واقع باید پردازه یا ریسه‌هایی که به پردازنده‌ها assign می‌شوند، بالانس باشند چون در غیر اینصورت بعضی پردازه‌ها overload شده درصورتی که ممکن است دیگر پردازه ها بیکار باشند و عملا استفاده از چند صف مشترک هزینه‌ای بی جهت بوده است. اما نکته‌ای که وجود دارد این است که بالانس و متعادل کردن این صف ها هم پیچیدگی خاص خودش را دارد.

هنگامی که از صف مشترک استفاده می‌شود دیگر نیازی به متعادل کردن پردازه ها نیست چون فقط یک صف برای پردازه ها وجود دارد.

نقص: به دلیل وجود تنها یک صف، به صف مشترک به صورت همزمان دسترسی وجود خواهد داشت که این موضوع باید هندل شود. در xv6 با استفاده از قفل کردن هندل می‌شود.

سوال ۴) در هر اجرای حلقه برای مدتی وقفه فعال میشود. علت چیست؟ آیا در سیستم های تک هسته‌ای به آن نیاز است؟

علت فعال شدن این وقفه ها در هر بار اجرای حلقه این می باشد که ممکن است حالتی پیش بیاید که هیچ پردازه RUNNABLE نداشته باشیم یعنی هیچ پردازه ی در انتظار اجرا در صف پردازه ها(ptable.proc) وجود نداشته باشد. برای مثال همه پردازه ها در حال گرفتن ورودی یا منتظر خروجی دادن(صف waiting) باشند. در این حالت اگر وقفه غیرفعال باشد، هیچوقت عملیات ورودی و خروجی به پایان نمیرسد. بنابراین نیاز داریم تا وقفه در هر حلقه برای مدتی فعال شود تا مواردی از این دست پیش نیاید.

بله، زیرا مشکل ذکر شده در بالا وابسته به تعداد هسته ها نیست و ممکن است در سیستم تک هسته ای هم این حالت رخ بدهد. بنابراین وقفه باید مدتی فعال شود.

سوال ۵) وقفه ها اولویت بالاتری نسبت به پردازه ها دارند. به طور کلی مدیریت وقفه ها در لینوکس در دو سطح صورت می گیرد. آن ها را نام برده و به اختصار توضیح دهید.

در فاز اول هسته generic interrupt handler را اجرا می‌کند که عدد وقفه را مشخص می‌کند که عدد وقفه، interrupt handler برای این وقفه‌ی خاص و interrupt controller را تعیین می‌کند. در این مرحله هرگونه اقدام حیاتی مانند تعیین وقفه در سطح interrupt controller انجام خواهد شد. وقفه‌های پردازنده‌های محلی (local) در مدت این فاز غیر فعال می‌شوند و تا فاز بعدی غیر فعال باقی می‌مانند. در فاز دوم تمام device drivers handler های مرتبط با این وقفه اجرا می‌شوند در پایان این فاز متد "end of interrupt" در interrupt controller فراخوانی می‌شود تا به interrupt controller اجازه دهد تا این وقفه را دوباره برقرار کند. وقفه‌های پردازنده‌ی محلی در این مرحله فعال هستند.

در نهایت در آخرین فاز از مدیریت وقفه اقدامات قابل تعویق در زمینه وقفه اجرا خواهند شد. این‌ها گاهی اوقات به عنوان Bottom Half شناخته می‌شوند (Top Half بخشی از مدیریت وقفه می‌باشد که با وقفه‌های غیر فعال اجرا میشود) در این مرحله وقفه‌ها در پردازنده‌ی محلی فعال می‌شوند.

همچنین طبق کتاب مرجع Linux Kernel Development می توان ISR ها در لینوکس را به دو بخش زیر تقسیم بندی کرد:

Top Halves: در Top Halves کارها بلافاصله پس از دریافت وقفه اجرا می شود و فقط کارهایی را انجام می دهد که از نظر زمانی حیاتی هستند، مانند تایید دریافت وقفه یا تنظیم مجدد سخت افزار. در این بخش Interrupt handler ها توسط کرنل به صورت asynchronous و به سرعت به وقفه‌ها پاسخ می‌دهند.

Bottom Halves: کارهایی که می توانند بعدا انجام شوند به Bottom Halves موکول می شود. Bottom Halves در آینده و در زمان مناسب تر با فعال بودن تمام وقفه ها اجرا می شوند.

اولویت این دو سطح مدیریت نسبت به هم و نسبت به پردازه ها چگونه است؟

اولویت Top Halves از Bottom Halves بالاتر است و اولویت این دو سطح، از پردازه ها بالاتر است، چرا که این سطوح وقفه ها را مدیریت می کنند که اولویت بالاتری نسبت به پردازه ها دارند.

در واقع وقتی یک مدیریت کننده وقفه Top Half فراخوانی می‌شود، آن ابتدا عملیات‌های سخت‌افزاری را هندل می‌کند و سپس مدیریت کننده‌های وقفه‌های Bottom Half را به صف وقفه‌ها اضافه می‌کند.

برای اطلاعات بیشتر این لینک مطالعه شود.

مدیریت وقفه ها در صورتی که بیش از حد زمان بر شود، می تواند منجر به گرسنگی پردازه ها گردد. این می تواند به خصوص در سیستم های بی درنگ مشکل ساز شود. چگونه این مشکل حل شده است؟

این مسئله در سیستم عامل لینوکس مانند دیگر سیستم عامل ها توسط ساز و کار افزایش سن (aging) حل می‌شود. به این صورت که اگر وقفه‌ای پس از گذشت زمان طولانی هندل نشده بود، ISR مربوط به این وقفه را یک سطح بالاتر می بریم، مثلا اگر ISR این وقفه در بخش bottom half باشد، آن را به بخش top half منتقل می‌کنیم تا مشکل گرسنگی برطرف شود.

زمان‌بندی بازخوردی چندسطحی

● زمان‌بندی RR:

ابتدا در ساختار proc، یک متغیر جدید به نام queue اضافه می کنیم و مقدار پیش فرض آن را در تابع allocproc دو در نظر میگیریم. سپس برای تعیین کردن پردازه بعدی توسط الگوریتم RR، روی صف پردازه ها پیمایش می‌کنیم و اولین پردازه RUNNABLE را که در صف ۱ قرار دارد، خروجی می‌دهیم. دقت شود که اگر چنین پردازه‌ای یافت نشود، عدد ۰ را خروجی می‌دهیم.

● زمان‌بندی LCFS:

ابتدا در ساختار proc، یک متغیر جدید به نام arrival_time اضافه می کنیم و مقدار پیش فرض آن را در تابع allocproc مقدار زمان سیستم در نظر میگیریم. سپس برای تعیین کردن پردازه بعدی توسط الگوریتم LCFS، روی صف پردازه ها پیمایش می‌کنیم و آخرین پردازه RUNNABLE را که وارد سیستم شده است و در صف ۲ قرار دارد،خروجی می‌دهیم.

دقت شود که اگر چنین پردازه‌ای یافت نشود، عدد ۰ را خروجی می‌دهیم.

● زمان‌بندی HRRN:

ابتدا در ساختار proc، متغیر های executed_cycle_number و MHRRN_priority را اضافه می کنیم و مقادیر پیش فرض executed_cycle_number و MHRRN_priority را در تابع allocproc به ترتیب یک و صفر در نظر می گیریم. سپس برای تعیین کردن پردازه بعدی توسط الگوریتم HRRN، روی صف پردازه ها پیمایش می‌کنیم و طبق فرمول گفته شده، پردازه RUNNABLE را که مقدار MHRRN آن بیشینه می‌باشد و در صف ۳ قرار دارد، خروجی می دهیم.

دقت شود که اگر چنین پردازه‌ای یافت نشود، عدد ۰ را خروجی می‌دهیم.

سازوکار افزایش سن

ابتدا در ساختار proc، یک متغیر جدید به نام waiting_cycles اضافه می کنیم و مقدار پیش فرض آن را در تابع allocproc صفر در نظر میگیریم. برای پیاده سازی ساز و کار افزایش سن در تابع scheduler اگر پردازه موردنظر توسط یکی از زمان‌بندها قابل اجرا باشد، روی صف پردازه ها پیمایش میکنیم. اگر پردازه RUNNABLE باشد، waiting_cycles را یک واحد زیاد می‌کنیم و اگر waiting_cycles از ۸۰۰۰ بیشتر و در صف ۲ یا ۳ باشد، پردازه را به یک صف بالاتر انتقال می‌دهیم و waiting_cycles را صفر میکنیم. در آخر پس از خروج از حلقه، waiting_cycles پردازه مورد نظر را صفر می‌کنیم.

```
for (op = ptable.proc; op < &ptable.proc[NPROC]; op++)
{
  if(op->pid == 0 && op->state != RUNNABLE)
    continue;

  op->waiting_cycles++;

  if(op->waiting_cycles > 8000 && op->queue > 1)
  {
    op->queue -= 1;
    op->waiting_cycles = 0;
  }
}
p->waiting_cycles = 0;
```

فراخوانی‌های سیستمی مورد نیاز

۱- تغییر صف مورد نیاز:

تابع سیستم کال change_queue با گرفتن ۲ پارامتر pid و queue اولویت صف پردازه با شناسه pid آن را به queue تغییر میدهد. برای این کار ابتدا با استفاده از تابع acquire پردازه ها را قفل کرده سپس روی همه پردازه ها در استراکت proc پیمایش می‌کنیم و پردازه‌ای که شناسه آن با pid یکی است را در نظر گرفته و فیلد queue را که خودمان به استراکت proc اضافه کردیم را به مقدار صف در پارامتر ورودی تغییر می‌دهیم. در نهایت با استفاده از release قفل را باز میکنیم.

۲- مقداردهی پارامتر MHRRN در سطح پردازه:

با استفاده از این سیستم کال، یک ضریب و pid یک پردازه را از کاربر می گیریم و به عنوان HRRNpriority در پردازه مورد نظر تاثیر می دهیم.

۳- مقداردهی پارامتر MHRRN در سطح سیستم:

با استفاده از این سیستم کال، یک ضریب را از کاربر می گیریم و به عنوان HRRNpriority در تمام پردازه ها تاثیر می دهیم.

۴- چاپ اطلاعات

با استفاده از این سیستم کال، اطلاعاتی مانند نام پردازه، شماره پردازه و ... را پرینت می کنیم.

نمونه‌ای از چاپ اطلاعات پردازه ها:

```
$ print_info
name       pid      state          queue_level      exec_cycles     waiting_cycles      arrival      HRRNpriority          MHRRN
.....................................................................................................................................
init       1     SLEEPING             1                34              15843              0              0                132
sh         2     SLEEPING             1                41              1              53              0                109
foo        5     RUNNABLE             2                4001              7796              1049              0                1
foo        4     SLEEPING             1                17              7793              1042              0                233
foo        6     RUNNABLE             2                3994              7811              1049              0                1
foo        7     RUNNING              1                3896              3              1049              0                1
foo        10     RUNNABLE             2                1              7796              5067              0                1954
foo        11     RUNNABLE             2                1              7796              5067              0                1954
foo        12     RUNNABLE             2                1              7796              5067              0                1954
foo        13     RUNNABLE             2                1              7796              5067              0                1954
foo        14     RUNNABLE             2                3881              5              5068              0                1
foo        15     RUNNABLE             2                1              7795              5068              0                1954
foo        16     RUNNABLE             2                1              7795              5068              0                1954
print_info 19     RUNNING              2                2              0              8974              0                0
```

برای چاپ، پس از وارد کردن دستور foo 10 & در xv6 با دستور print_info اطلاعات پردازه ها چاپ خواهند شد. ۱۰ تعداد پردازه‌های ایجاد شده در برنامه تست foo است که این تعداد از کاربر گرفته می‌شود.

در برنامه‌ی foo تعدادی پردازه ساخته‌ایم و پردازه‌ها عملیات پردازشی (cpu-intensive) انجام میدهند که در اینجا ما با سه حلقه‌ی تو در تو این کار را انجام دادیم.

شناسه آخرین کامیت:

25fec031260f0c88c87b00865753b9b49b06883f

آدرس مخزن:

https://gitlab.com/ali_akhgari/os-lab