گزارش کار پروژه ۳ آزمایشگاه درس سیستم عامل

علی علم طلب ۸۱۰۱۰۰۱۸۹

پرهام بدو ۸۱۰۱۰۰۲۴۴

علیرضا کامکار ۸۱۰۱۰۰۲۰۲

پاسخ سوال ۱:

ساختار بلوک کنترل پردازه در سیستم‌عامل xv۶ با نام struct proc شناخته می‌شود که در فایل هدر proc.h تعریف شده است .این ساختار شامل فیلدهایی برای نگهداری اطلاعات حیاتی پردازه است که از مهم‌ترین آن‌ها می‌توان به sz برای اندازه حافظه پردازه و pgdir برای جدول صفحه پردازه و kstackبرای پشته هسته و state برای وضعیت فعلی پردازه و pid برای شناسه منحصربه‌فرد و parent برای اشاره‌گر به پردازه والد و trapframe برای ذخیره وضعیت ثبات‌ها هنگام وقفه و context برای ذخیره وضعیت ثبات‌ها هنگام تعویض متن و chan برای کانال خواب و killed برای وضعیت خاتمه اشاره کرد .وضعیت‌های تعریف شده در این سیستم‌عامل شامل UNUSED برای پردازه استفاده نشده و EMBRYO برای پردازه در حال ساخت و SLEEPINGبرای حالت انتظار و RUNNABLE برای آماده اجرا و RUNNING برای در حال اجرا و ZOMBIE برای خاتمه یافته می‌باشد.

در مقایسه با ساختار استاندارد که در شکل ۳.۳ کتاب منبع درس آمده است شباهت‌های بسیاری وجود دارد زیرا هر دو ساختار وظیفه نگهداری وضعیت پردازه و شمارنده برنامه و ثبات‌ها و اطلاعات زمان‌بندی و مدیریت حافظه را بر عهده دارند .با این حال پیاده‌سازی xv۶ به عنوان یک سیستم‌عامل آموزشی ساده‌تر است و برخی جزئیات پیچیده مانند لیست فایل‌های باز یا اطلاعات حسابداری دقیق که در سیستم‌های مدرن وجود دارد در آن به شکل ساده‌تری پیاده شده یا مدیریت می‌شود اما اصول کلی و فیلدهای اساسی برای مدیریت چرخه حیات پردازه در هر دو ساختار یکسان است.

پاسخ سوال ۲ :

برای نگاشت وضعیت‌های موجود در xv۶ به وضعیت‌های استاندارد نمایش داده شده در شکل ۱ می‌توان گفت که وضعیت EMBRYO در xv۶ معادل وضعیت new در نمودار استاندارد است که نشان‌دهنده مراحل اولیه ایجاد پردازه می‌باشد. وضعیت RUNNABLE در xv۶ دقیقاً معادل وضعیت ready در شکل ۱ است که پردازه در صف آماده قرار دارد و منتظر دریافت پردازنده است .وضعیت RUNNING در xv۶ همان وضعیت running در نمودار استاندارد است که پردازه در حال اجرای دستورات روی پردازنده می‌باشد.

همچنین وضعیت SLEEPING در xv۶ معادل وضعیت waiting در شکل ۱ است که پردازه منتظر وقوع یک رویداد خاص یا تکمیل عملیات ورودی و خروجی است. وضعیت ZOMBIE در xv۶ معادل وضعیت terminated در نمودار استاندارد است که پردازه اجرای خود را به پایان رسانده اما هنوز از جدول پردازه‌ها پاک نشده است .وضعیت UNUSED نیز به خانه‌های خالی جدول پردازه اشاره دارد که در چرخه حیات استاندارد پردازه جایگاه خاصی ندارد و صرفاً برای مدیریت منابع سیستم‌عامل استفاده می‌شود.

پاسخ سوال ۳ :

گذار از حالت new به ready در سیستم‌عامل xv۶ با همکاری توابع allocproc و userinit یا fork انجام می‌شود .در ابتدا تابع allocproc یک اسلات خالی در جدول پردازه‌ها پیدا کرده و وضعیت آن را به EMBRYO تغییر می‌دهد که معادل حالت new است .سپس در تابع userinit برای اولین پردازه یا fork برای سایر پردازه‌ها پس از تنظیمات اولیه حافظه و قاب پشته وضعیت پردازه صراحتاً به RUNNABLE تغییر می‌یابد که همان حالت ready است.

بنابراین توابع اصلی که مستقیماً در تغییر وضعیت و آماده‌سازی نهایی نقش دارند allocproc برای ایجاد وضعیت اولیه و userinit یا fork برای نهایی کردن و تغییر وضعیت به RUNNABLE هستند .در این گذار وضعیت پردازه از حالت UNUSED در ابتدای تخصیص به EMBRYO تغییر کرده و پس از اتمام پیکربندی‌های اولیه نهایتاً به وضعیت RUNNABLE تغییر می‌یابد تا توسط زمان‌بند قابل انتخاب باشد.

پاسخ سوال ۴ :

سقف تعداد پردازه‌ها در سیستم‌عامل xv۶ در فایل param.h با متغیر NPROC تعریف شده است که به طور پیش‌فرض برابر با ۶۴ می‌باشد .این یعنی سیستم‌عامل حداکثر می‌تواند ۶۴ پردازه هم‌زمان را در جدول پردازه‌ها مدیریت کند و تلاش برای ایجاد پردازه بیشتر با محدودیت مواجه خواهد شد.

در صورتی که یک پردازه سعی کند فرزندان زیادی ایجاد کند و از این سقف عبور کند تابع allocproc در یافتن فضای خالی در جدول پردازه‌ها ناکام مانده و مقدار null یا صفر برمی‌گرداند .در نتیجه فراخوانی سیستمی fork با شکست مواجه شده و مقدار منفی ۱ را به برنامه سطح کاربر باز می‌گرداند و معمولاً پیغامی مبنی بر شکست در ایجاد پردازه مانند fork panic در برخی پیاده‌سازی‌ها یا صرفاً بازگشت خطا به کاربر نمایش داده می‌شود.

پاسخ سوال ۵ :

در ابتدای حلقه تابع scheduler نیاز است که جدول پردازه‌ها قفل شود زیرا xv۶ از مدل چند پردازشی متقارن یا همان SMP پشتیبانی می‌کند و چندین هسته پردازشی به صورت هم‌زمان به جدول پردازه‌ها دسترسی دارند .اگر قفل‌گذاری انجام نشود ممکن است دو پردازنده هم‌زمان یک پردازه مشابه که در وضعیت RUNNABLE است را برای اجرا انتخاب کنند که منجر به شرایط رقابتی و خرابی سیستم می‌شود.

در سیستم‌هایی که تنها یک پردازنده دارند اگرچه رقابت بین پردازنده‌ها وجود ندارد اما همچنان بحث وقفه‌ها مطرح است. با این حال در xv۶ مکانیزم قفل‌گذاری ptable به گونه‌ای طراحی شده که وقفه‌ها را نیز در زمان نگه داشتن قفل غیرفعال می‌کند تا از ناسازگاری داده‌ها جلوگیری شود .بنابراین حتی در سیستم تک هسته‌ای هم برای حفظ یکپارچگی ساختارهای داده‌ای کرنل و جلوگیری از تداخل ناخواسته با روتین‌های وقفه استفاده از مکانیزم‌های کنترلی یا غیرفعال کردن وقفه‌ها در بخش‌های بحرانی ضروری است.

پاسخ سوال ۶ :

تابع scheduler در xv۶ یک حلقه بی‌نهایت است که به ترتیب آرایه جدول پردازه‌ها را پیمایش می‌کند .اگر پردازنده‌ای در حال پیمایش جدول باشد و در اندیس فعلی i قرار داشته باشد و در همین حین پردازه جدیدی در اندیسی کمتر از i مثلاً k آماده اجرا شود زمان‌بند در پیمایش جاری آن را نخواهد دید زیرا از آن عبور کرده است.

بنابراین پردازنده باید یک دور کامل پیمایش لیست را تمام کند و پس از آزادسازی قفل و شروع مجدد حلقه جستجو از ابتدای آرایه به پردازه جدید در اندیس k برسد .پس این پردازه در دور بعدی حلقه زمان‌بند بعد از برگشت به ابتدای لیست شانس اجرا پیدا خواهد کرد مگر اینکه اندیس پردازه جدید بیشتر از اندیس فعلی شمارنده زمان‌بند باشد که در این صورت در همان دور دیده می‌شود.

پاسخ سوال ۷ :

در ساختار context که در فایل proc.h تعریف شده است ثبات‌هایی ذخیره می‌شوند که طبق قرارداد فراخوانی توابع در معماری x۸۶ باید توسط تابع فراخوانی‌شونده حفظ شوند .این ثبات‌ها شامل edi و esi و ebx و ebp و eip هستند. این ساختار برای جابه‌جایی بین پردازه‌ها یا بین پردازه و زمان‌بند استفاده می‌شود.

برخلاف trapframe که تمام ثبات‌ها را ذخیره می‌کند ساختار context تنها این پنج ثبات کلیدی را نگه می‌دارد زیرا کامپایلر C تضمین می‌کند سایر ثبات‌ها در صورت نیاز توسط فراخوانی‌کننده ذخیره شده‌اند یا مقدارشان مهم نیست .این طراحی باعث می‌شود عملیات تعویض متن سریع‌تر و سبک‌تر انجام شود.

پاسخ سوال ۸ :

یکی از مهم‌ترین ثبات‌ها برای بازیابی روند اجرای برنامه شمارنده برنامه است که در ساختار context سیستم‌عامل xv۶ با نام eip ذخیره می‌شود .این ثبات آدرس دستورالعمل بعدی که باید اجرا شود را نگه می‌دارد و نقش کلیدی در بازگشت به نقطه توقف قبلی دارد.

شیوه ذخیره و بازنشانی eip در تابع swtch به صورت ضمنی و با استفاده از مکانیزم فراخوانی تابع و پشته انجام می‌شود. زمانی که swtch فراخوانی می‌شود آدرس بازگشت یا همان مقدار eip فعلی به طور خودکار روی پشته قرار می‌گیرد .هنگام بازگشت از تابع swtch در پردازه جدید دستور اسمبلی retاجرا شده و آدرس ذخیره شده از پشته برداشته شده و در ثبات eip قرار می‌گیرد که باعث پرش اجرا به کد پردازه جدید می‌شود.

پاسخ سوال ۹:

اگر در ابتدای تابع scheduler وقفه‌ها با دستور sti فعال نشوند در شرایطی که هیچ پردازه‌ای برای اجرا وجود نداشته باشد سیستم در حلقه زمان‌بند گیر کرده و چون وقفه‌ها غیرفعال هستند هیچ رویداد خارجی مانند تایمر یا کیبورد نمی‌تواند پردازنده را متوجه خود کند .این موضوع باعث می‌شود سیستم‌عامل عملاً قفل شده و پاسخگو نباشد زیرا مکانیزم اصلی تغییر وضعیت و پیشرفت زمان از کار می‌افتد.

در حالتی که وقفه‌ها غیرفعال باشند هیچ وقفه‌ای قابلیت اجرا ندارد زیرا پردازنده سیگنال‌های وقفه سخت‌افزاری را نادیده می‌گیرد .دستور sti به پردازنده اجازه می‌دهد تا به وقفه‌های ماسک‌نپذیر و درخواست‌های وقفه پاسخ دهد. فعال‌سازی وقفه‌ها در زمانی که سیستم بیکار است ضروری است تا پردازنده بتواند مثلاً با دریافت وقفه تایمر یا دیسک وضعیت پردازه‌های منتظر را تغییر دهد و آن‌ها را آماده اجرا کند.

پاسخ سوال ۱۰ :

وقفه تایمر در xv۶ توسط سخت‌افزار LAPIC یا تراشه تایمر برنامه‌ریزی می‌شود و معمولاً با فرکانس ۱۰۰ هرتز تنظیم شده است به این معنی که هر ۱۰ میلی‌ثانیه یک‌بار یک وقفه تایمر صادر می‌شود .این مقدار تعیین می‌کند که سیستم‌عامل هر چند وقت یک‌بار کنترل را برای تصمیم‌گیری‌های زمان‌بندی به دست می‌گیرد.

با بررسی تابع lapicinit در فایل lapic.c و محاسبات مربوط به ثبات‌های تایمر می‌توان مشاهده کرد که مقادیر به گونه‌ای تنظیم شده‌اند که این بازه زمانی رعایت شود .همچنین با افزودن دستور چاپ در هندلر وقفه تایمر در فایل trap.c می‌توان دید که متغیر ticks تقریباً هر ۱۰ میلی‌ثانیه یک واحد افزایش می‌یابد که تأییدکننده این فرکانس است.

پاسخ سوال ۱۱ :

تابعی که منجر به انجام گذار interrupt در شکل ۱ می‌شود تابع yield است .این تابع زمانی فراخوانی می‌شود که یک وقفه به‌ویژه وقفه تایمر رخ دهد و پردازه جاری کوانتوم زمانی خود را مصرف کرده باشد .وظیفه این تابع این است که پردازه را داوطلبانه از حالت اجرا خارج کند تا زمان‌بند بتواند تصمیم جدیدی بگیرد.

وقتی وقفه تایمر رخ می‌دهد تابع trap اجرا شده و با بررسی شرایط تابع yield را فراخوانی می‌کند .تابع yield وضعیت پردازه را از RUNNING به RUNNABLEتغییر می‌دهد و سپس تابع sched را صدا می‌زند تا عملیات تعویض متن انجام شود .این فرآیند دقیقاً معادل یال interrupt در نمودار وضعیت پردازه است که پردازه را از حالت اجرا به صف آماده باز می‌گرداند.

پاسخ سوال ۱۲:

با توجه به اینکه در xv۶ در هر وقفه تایمر که در تابع trap مدیریت می‌شود بررسی می‌شود که آیا پردازه‌ای در حال اجراست یا خیر و اگر باشد تابع yieldفراخوانی می‌شود می‌توان استدلال کرد که کوانتوم زمانی در این پیاده‌سازی برابر با فاصله بین دو وقفه تایمر است .از آنجا که پردازه در اولین وقفه تایمری که دریافت می‌کند پردازنده را رها می‌کند کوانتوم زمانی برابر با یک تیک سیستم است.

با توجه به پاسخ سوال ۱۰ که فرکانس تایمر ۱۰۰ هرتز یعنی هر ۱۰ میلی‌ثانیه است بنابراین کوانتوم زمانی الگوریتم نوبت‌گردشی در xv۶ برابر با ۱۰ میلی‌ثانیه می‌باشد .این بدان معناست که هر پردازه حداکثر ۱۰ میلی‌ثانیه فرصت اجرا دارد و پس از آن در صورت وجود پردازه دیگر تعویض متن رخ می‌دهد.

پاسخ سوال ۱۳:

تابع wait در سیستم‌عامل xv۶ برای اینکه پردازه والد را تا زمان خاتمه یکی از فرزندانش در حالت انتظار نگه دارد در نهایت از تابع sleep استفاده می‌کند .تابع sleep وضعیت پردازه را به SLEEPING تغییر داده و با آزادسازی قفل پردازنده کنترل را به زمان‌بند می‌دهد تا پردازه دیگری را اجرا کند.

در واقع wait در یک حلقه قرار دارد که وجود فرزندان را چک می‌کند و اگر فرزندان هنوز در حال اجرا باشند با فراخوانی sleep روی خودِ پردازه جاری یا همان والد به خواب می‌رود تا زمانی که با یک سیگنال بیداری wakeup که معمولاً توسط تابع exit فرزند ارسال می‌شود دوباره فعال گردد.

پاسخ سوال ۱۴ :

علاوه بر استفاده در تابع wait تابع sleep کاربردهای متعددی در همگام‌سازی و مدیریت منابع ورودی و خروجی دارد .یکی دیگر از استفاده‌های رایج این تابع در خواندن و نوشتن از دیسک یا کنسول است .به عنوان مثال زمانی که یک پردازه درخواستی برای خواندن از دیسک ارسال می‌کند چون عملیات دیسک کند است پردازه با استفاده از sleep به خواب می‌رود تا دیسک کارش تمام شود.

پس از اینکه کنترلر دیسک عملیات را تمام کرد و وقفه زد هندلر وقفه با استفاده از تابع wakeup پردازه منتظر را بیدار می‌کند. این مکانیزم باعث می‌شود تا زمان پردازنده در حین عملیات‌های طولانی ورودی و خروجی هدر نرود و به پردازه‌های دیگر اختصاص یابد.

پاسخ سوال ۱۵ :

تابعی که در سطح کرنل وظیفه آگاه‌سازی پردازه‌ها از وقوع یک رویداد و بیدار کردن آن‌ها را بر عهده دارد تابع wakeup است .این تابع باعث می‌شود پردازه‌هایی که روی یک کانال خاص خوابیده‌اند از وضعیت SLEEPING به وضعیت RUNNABLE تغییر حالت دهند تا در نوبت بعدی زمان‌بندی شانس اجرا داشته باشند.

توابع دیگری که می‌توانند منجر به این گذار شوند یا اثر مشابهی داشته باشند شامل تابع kill است که اگر پردازه خوابیده باشد ممکن است آن را بیدار کند تا خاتمه یابد و همچنین تابع exit که به طور غیرمستقیم با فراخوانی wakeup باعث بیدار شدن پردازه والد که در wait منتظر است می‌شود .اما مکانیزم اصلی و مستقیم تغییر وضعیت از انتظار به آماده همان تابع wakeup است.

پاسخ سوال ۱۶ :

رویکرد xv۶ در مواجهه با پردازه‌های یتیم یا همان Orphan که والد آن‌ها قبل از خودشان خاتمه یافته است واگذاری سرپرستی آن‌ها به پردازه init است .پردازه init اولین پردازه‌ای است که توسط کرنل ساخته می‌شود و همیشه در حال اجراست.

زمانی که یک پردازه با تابع exit خاتمه می‌یابد در بخشی از کد خود تمام فرزندانش را بررسی می‌کند و والد آن‌ها را به پردازه init تغییر می‌دهد .بدین ترتیب پردازه init به عنوان والد جدید مسئولیت wait کردن و پاک‌سازی منابع این فرزندان پس از خاتمه‌شان را بر عهده می‌گیرد تا هیچ پردازه‌ای به صورت زامبی دائمی در سیستم باقی نماند.

پاسخ سوال ۱۷:

ساختار مربوط به پردازنده در xv۶ با نام struct cpu در فایل proc.h تعریف شده است .این ساختار وظیفه نگهداری وضعیت هر هسته پردازشی را بر عهده دارد و برای سیستم‌های چند پردازنده‌ای حیاتی است. با بررسی کد فیلدهای این ساختار شامل apicid برای شناسه سخت‌افزاری پردازنده و schedulerبرای ذخیره کانتکست زمان‌بند آن هسته و ts برای وضعیت وظیفه جهت مدیریت وقفه‌ها در x۸۶ و gdt برای جدول توصیف‌گر سراسری مختص آن هسته و started برای نشانگر راه‌اندازی شدن هسته و ncli برای شمارنده عمق غیرفعال‌سازی وقفه‌ها و intena برای وضعیت فعال بودن وقفه‌ها قبل از pushcli و proc برای اشاره‌گر به پردازه‌ای که هم‌اکنون روی این هسته در حال اجراست می‌باشد.

این ساختار به سیستم‌عامل اجازه می‌دهد تا بداند هر هسته دقیقاً چه کاری انجام می‌دهد و آیا وقفه‌ها روی آن فعال است یا خیر و کدام پردازه را در اختیار دارد .وجود فیلد cpu که آرایه‌ای از این ساختارهاست امکان مدیریت مستقل زمان‌بندی و وقفه‌ها را برای هر هسته در معماری چند پردازنده‌ای فراهم می‌کند.

۴. زمان بندی هسته های ناهمگون:

برای پیاده‌سازی بخش زمان‌بندی هسته‌های ناهمگون ابتدا در فایل proc.c دو تابع کمکی is_ecore و is_pcore تعریف کردیم که فقط با چک کردن زوج یا فرد بودن شماره‌ی CPU تشخیص می‌دهند یک هسته از نوع E-core است یاP-core ، به این ترتیب فرض صورت‌مسئله که هسته‌های با CPUID زوج کم‌مصرف و هسته‌های با CPUID فرد پرقدرت هستند در کد اعمال شد.
سپس در تابع allocproc در همان فایل بعد از پیدا شدن خانه‌ی خالی در ptable برای هر فرایند جدید فیلدهای create_tick و qnticksرا مقداردهی کردیم و با استفاده از تابع count_procs_on_cpu روی همه‌ی هسته‌های E-core چرخیدیم تا کم‌بارترین هسته‌ی زوج را پیدا کنیم و مقدار assigned_cpu فرایند را برابر آن قرار دهیم و queue_type را هم روی CORE_E بگذاریم تا نوع هسته‌ای که فرایند روی آن قرار گرفته در ساختار پردازه ثبت شود.
در نهایت در تابع scheduler در proc.c با استفاده از cpuid و این که خروجی is_ecore یا is_pcore چه باشد تصمیم می‌گیریم برای هر CPU از کدام تابع انتخاب پردازه یعنی pick_proc_ecore یا pick_proc_pcore استفاده شود تا زمان‌بندی بتواند بین دو نوع هسته رفتار متفاوت داشته باشد.

```
static inline int
is_ecore(int id)
{
  return (id % 2) == 0;
}

static inline int
is_pcore(int id)
{
  return (id % 2) == 1;
}
```

```
432   }
433
434   void
435   scheduler(void)
436   {
437     struct cpu *c = mycpu();
438     int id = cpuid();
439     struct proc *p;
440
441     c->proc = 0;
442
443     for(;;){
444       sti();
445
446       acquire(&ptable.lock);
447
448       if(is_ecore(id))
449         p = pick_proc_ecore(id);
450       else
451         p = pick_proc_pcore(id);
452
453       if(p){
454         c->proc = p;
455         switchuvm(p);
456         p->state = RUNNING;
457         swtch(&(c->scheduler), p->context);
458         switchkvm();
459         c->proc = 0;
460       }
461
462       release(&ptable.lock);
463     }
464   }
```

```
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00


Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Alireza Kamkar
Parham Badv
Ali Elm talab
$
```

۱.۴ هسته های E با شماره CPUID زوج :زمانبد نوبت گردشی با کوانتوم زمانی

برای پیاده‌سازی بخش هسته‌های E با شماره CPUID زوج و زمان‌بند نوبت گردشی با کوانتوم زمانی اول در فایل proc.h به ساختار struct procفیلد qnticks را اضافه کردیم تا تعداد تیک‌های سپری شده در کوانتوم هر پردازه ذخیره شود و در فایل proc.c هنگام ساخت پردازه در تابع allocproc این مقدار را صفر کردیم و هر بار که پردازه‌ای روی هسته انتخاب می‌شود در توابع pick_proc_ecore و pick_proc_pcore هم qnticks را ریست کردیم تا از شروع کوانتوم جدید شمارش شود.
سپس در فایل trap.c در رسیدگی به وقفه تایمر case مربوط به IRQ_TIMER را تغییر دادیم به این صورت که اگر پردازه‌ای در حال اجرا بود و روی هسته‌ای با CPUID زوج اجرا می‌شد مقدار qnticks آن را در هر تیک یکی افزایش دادیم و وقتی به مقدار ثابت تعریف شده برای کوانتوم رسید با فراخوانی yield آن را پیش‌دستانه از پردازنده خارج کردیم تا نوبت پردازه بعدی در الگوریتم round robin برسد در حالی که برای هسته‌های با CPUID فرد این پیش‌دستی انجام نشد و رفتار آن‌ها مانند حالت بدون کوانتوم باقی ماند.
برای تست عملی در Makefile یا هنگام اجرای دستور make مقدار CPUS را برابر ۱ قرار دادیم تا فقط یک هسته فعال باشد و با توجه به این که هر تیک حدود ده میلی‌ثانیه است کوانتومی انتخاب کردیم که زمان اجرای آن نزدیک به سی میلی‌ثانیه باشد و با مشاهده خروجی cprintf در trap.c و رفتار جابه‌جایی پردازه‌ها مطمئن شدیم که الگوریتم نوبت گردشی با کوانتوم زمانی روی هسته‌های E به درستی کار می‌کند.

```c
static struct proc*
pick_proc_ecore(int cpu_id)
{
  int start = last_rr_index[cpu_id];
  int idx = (start + 1) % NPROC;
  int scanned = 0;
  struct proc *p;

  while(scanned < NPROC){
    p = &ptable.proc[idx];

    if(p->state == RUNNABLE && p->assigned_cpu == cpu_id){
      last_rr_index[cpu_id] = idx;
      p->qnticks = 0;
      return p;
    }

    idx = (idx + 1) % NPROC;
    scanned++;
  }

  return 0;
}

static struct proc*
pick_proc_pcore(int cpu_id)
{
  struct proc *p, *best = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    if(p->assigned_cpu != cpu_id)
      continue;

    if(best == 0 || p->create_tick < best->create_tick)
      best = p;
  }

  if(best)
    best->qnticks = 0;

  return best;
}
```

```c
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }

  if(myproc() && myproc()->state == RUNNING &&
     (tf->cs & 3) == DPL_USER){
    int id = cpuid();
    if(is_ecore(id)){
      myproc()->qnticks++;
      if(myproc()->qnticks >= ECORE_QUANTUM_TICKS)
        yield();
    } else {
      yield();
    }
  }
```

۲.۴ جدا کردن صف هر هسته:

برای پیاده‌سازی ایده جدا بودن صف هر هسته در بخش دو چهار ابتدا در فایل proc.h به ساختار struct proc یک فیلد assigned_cpuاضافه کردیم تا مشخص شود هر پردازه به کدام هسته نسبت داده شده است و همچنین فیلد queue_type را برای تشخیص نوع صف همان هسته در نظر گرفتیم .

سپس در فایل proc.c در تابع allocproc هنگام ایجاد هر پردازه جدید به جای این که همه پردازه‌ها در یک صف مشترک باشند با استفاده از تابع count_procs_on_cpu روی هسته‌های موجود شمارش انجام دادیم و پردازه را روی هسته‌ای با کمترین بار قرار دادیم و مقدار assigned_cpu آن را برابر شماره همان هسته گذاشتیم .

در ادامه در توابع انتخاب پردازه یعنی pick_proc_ecore و pick_proc_pcore شرط گذاشتیم که فقط پردازه‌هایی انتخاب شوند که assigned_cpuآن‌ها برابر شناسه همان cpu جاری است و در حلقه اصلی تابع scheduler نیز به جای اسکن آزاد کل جدول پردازه‌ها از همین توابع انتخاب استفاده کردیم در نتیجه هر هسته عملا صف مخصوص خودش را دارد و فقط از پردازه‌های صف خودش زمان‌بندی می‌کند که مطابق خواسته این قسمت یعنی جدا شدن صف هر هسته در سیستم چندپردازنده‌ای است.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->create_tick = ticks;
  p->qnticks = 0;

  int best_cpu = 0;
  int best_load = 1000000;
  int c;

  for(c = 0; c < ncpu; c++){
    if(!is_ecore(c))
      continue;

    int load = count_procs_on_cpu(c);
    if(load < best_load){
      best_load = load;
      best_cpu = c;
    }
  }

  p->assigned_cpu = best_cpu;
  p->queue_type = CORE_E;

  release(&ptable.lock);
```

۴. ۳ هسته های P با CPUID فرد : زمانبد اولین ورود اولین رسیدگی

ابتدا در فایل proc.h به ساختار struct proc فیلدی به نام create_tick اضافه کردیم تا لحظه ایجاد هر پردازه را با مقدار متغیر سراسری ticks ذخیره کنیم.

سپس در تابع allocproc در فایل proc.c بعد از انتخاب خانه خالی در جدول پردازه ها این فیلد را برابر ticks قرار دادیم تا برای همیشه زمان ایجاد آن پردازه مشخص بماند در ادامه در تابع pick_proc_pcore در همین فایل که وظیفه انتخاب پردازه برای هسته های پرقدرت را دارد روی کل جدول پردازه ها حلقه زدیم و فقط پردازه های RUNNABLE را که فیلد assigned_cpu آنها برابر شماره همان هسته است در نظر گرفتیم و بین آنها پردازه ای را انتخاب کردیم که کمترین create_tick را دارد .

در نتیجه پردازه ای که زودتر ایجاد شده ولی هنوز تمام نشده در اولویت قرار میگیرد و تا زمانی که آماده اجراست هیچ پردازه جدیدتری جای آن را در صف هسته P نمیگیرد و در تابع scheduler نیز شرط گذاشتیم که برای هسته های با CPUID فرد به جای pick_proc_ecoreاز همین pick_proc_pcore استفاده شود تا این رفتار FCFS فقط روی هسته های P core اعمال شود.

```
static struct proc*
pick_proc_pcore(int cpu_id)
{
  struct proc *p, *best = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    if(p->assigned_cpu != cpu_id)
      continue;

    if(best == 0 || p->create_tick < best->create_tick)
      best = p;
  }

  if(best)
    best->qnticks = 0;

  return best;
}
```

۴.۴ جابه جایی پردازنده میان صف ها

برای جابه جایی پردازه میان صف‌ها و پیاده سازی مکانیزم توازن بار ابتدا در فایل proc.c تابع کمکی count_procs_on_cpu را نوشتیم تا برای هر هسته تعداد پردازه‌های غیر UNUSED و غیر ZOMBIE و غیر EMBRYO را بشمارد و بفهمیم بار هر صف چقدر است.

سپس تابع دیگری به نام balance_load اضافه کردیم که ورودی آن شناسه یک هسته نوع E است و با استفاده از count_procs_on_cpuتعداد پردازه‌های آن هسته و همه هسته‌های نوع P را مقایسه می‌کند و اگر هسته E دست کم سه پردازه بیشتر از سبک ترین هسته P داشت یکی از پردازه‌های قابل جابه جایی آن را که نه init است و نه sh انتخاب می‌کند و فیلد assigned_cpuرا به شماره هسته P و فیلد queue_type را به CORE_P تغییر می‌دهد.

و در نتیجه پردازه به صف هسته پرقدرت منتقل می‌شود در ادامه در تابع allocproc نیز هنگام ایجاد پردازه‌های جدید به جز init و sh آن‌ها را فقط در صف هسته‌های E قرار دادیم تا هل دادن پردازه به صف P فقط از مسیر balance_load انجام شود و در نهایت در تابع schedulerبعد از گرفتن قفل ptable برای هر هسته E به صورت دوره‌ای بر اساس متغیر last_balance_tick بررسی کردیم که اگر از آخرین بار بالانس بیش از آستانه زمان تعیین شده گذشته باشد balance_load روی آن هسته صدا زده شود تا توزیع بار میان هسته‌ها به شکل پویا اصلاح گردد.

```
void
balance_load(int e_cpu)
{
  int c;
  int min_p_cpu = -1;
  int min_p_load = 1000000;

  int e_load = count_procs_on_cpu(e_cpu);

  for(c = 0; c < ncpu; c++){
    if(!is_pcore(c))
      continue;

    int load = count_procs_on_cpu(c);
    if(load < min_p_load){
      min_p_load = load;
      min_p_cpu = c;
    }
  }

  if(min_p_cpu < 0)
    return;

  if(e_load < min_p_load + 3)
    return;

  struct proc *p, *candidate = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED || p->state == ZOMBIE)
      continue;
    if(p->assigned_cpu != e_cpu)
      continue;

    if(p->pid == 1)
      continue;

    if(strncmp(p->name, "sh", 2) == 0)
      continue;

    candidate = p;
    break;
  }

  if(candidate){
    candidate->assigned_cpu = min_p_cpu;
    candidate->queue_type = CORE_P;
  }
}
```

```
static int
count_procs_on_cpu(int cpu_id)
{
  int cnt = 0;
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED || p->state == ZOMBIE || p->state == EMBRYO)
      continue;
    if(p->assigned_cpu == cpu_id)
      cnt++;
  }

  return cnt;
}
```

۴ .۵ و ۵ .۵ ارزیابی الگوریتم زمانبدی و فراخوانی سیستمی:

ابتدا در فایل proc.c دو تابع هسته‌ای tput_start و tput_end اضافه کردیم که در اولی با گرفتن قفل tickslock متغیرهای سراسری tput_active و tput_start_ticks و tput_finished_count را مقداردهی می‌کنیم تا شروع بازه اندازه‌گیری گذردهی مشخص شود.

و در تابع exit هر بار که پردازه‌ای در حالتی که tput_active روشن است تمام می‌شود شمارنده tput_finished_count را زیاد می‌کنیم و در tput_end با محاسبه اختلاف ticks فعلی و tput_start_ticks و تقسیم تعداد پردازه‌های تمام شده بر این تعداد تیک مقدار تقریبی گذردهی را حساب کرده و با cprintf چاپ می‌کنیم.

سپس در sysproc.c توابع رابط sys_tputstart و sys_tputend را نوشتیم که فقط این توابع هسته‌ای را صدا می‌زنند و در فایل‌های syscall.hو syscall.c شماره syscall جدید و ورودی جدول syscalls را برای tputstart و tputend و همین طور برای procinfo اضافه کردیم.

و در user.h اعلام تابع‌های tputstart و tputend و procinfo را گذاشتیم و در usys.S نیز سه خط SYSCALL مربوط به این فراخوانی‌ها افزوده شد .

در نهایت یک برنامه تست سطح کاربر مثل tputtest و schedtest نوشتیم که در ابتدا tputstart را صدا می‌زند و تعدادی پردازه سنگین ایجاد می‌کند و بعد از پایان کار همه فرایندها tputend را فراخوانی کرده و با procinfo وضعیت پردازه‌ها را چاپ می‌کند تا در گزارش بتوانیم معیار گذردهی و اطلاعات زمان‌بندی را به شکل عددی و متنی نشان دهیم.

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  if(tput_active){
    acquire(&tickslock);
    tput_finished_count++;
    release(&tickslock);
  }

  acquire(&ptable.lock);

  wakeup1(curproc->parent);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
int
sys_tputstart(void)
{
  tput_start();
  return 0;
}

int
sys_tputend(void)
{
  tput_end();
  return 0;
}

int
sys_procinfo(void)
{
  procdump_ext();
  return 0;
}
```

```c
// schedtest.c
#include "types.h"
#include "stat.h"
#include "user.h"

void
heavy(int k)
{
  volatile int i, j;
  for(i = 0; i < k; i++){
    for(j = 0; j < 100000; j++){
      // busy work
    }
  }
}

void
run_case(int nprocs, int work)
{
  int i, pid;

  printf(1, "\n=== CASE: nprocs=%d, work=%d ===\n", nprocs, work);

  tputstart();
  for(i = 0; i < nprocs; i++){
    pid = fork();
    if(pid < 0){
      printf(1, "fork failed\n");
      exit();
    }
    if(pid == 0){
      heavy(work);
      exit();
    }
  }

  for(i = 0; i < nprocs; i++)
    wait();

  tputend();
  procinfo();
}

int
main(int argc, char *argv[])
{
  run_case(4, 30);
  run_case(8, 30);
  run_case(4, 80);
  run_case(12, 80);

  exit();
}
```

tputtest.c > main(int, char * [])

```c
#include "types.h"
#include "stat.h"
#include "user.h"

void
heavy(int n)
{
  volatile int i, j;
  for(i = 0; i < n; i++){
    for(j = 0; j < 100000; j++){
    }
  }
}

int
main(int argc, char *argv[])
{
  int nchild = 10;
  int i, pid;

  tputstart();

  for(i = 0; i < nchild; i++){
    pid = fork();
    if(pid < 0){
      printf(1, "fork failed\n");
      exit();
    }
    if(pid == 0){
      heavy(50);
      exit();
    }
  }

  for(i = 0; i < nchild; i++)
    wait();

  tputend();
  procinfo();

  exit();
}
```

۶.۵ برنامه های سطح کاربر

برای بخش برنامه‌های سطح کاربر چند برنامه تست نوشتیم تا الگوریتم‌های زمان‌بندی و فراخوانی‌های tputstart و tputend و procinfoرا در عمل امتحان کنیم.

برای این کار دو برنامه سطح کاربر مثل tputtest و schedtest را در ریشه کد xv۶ اضافه کردیم و در فایل Makefile نام این برنامه‌ها را به لیست UPROGS وارد کردیم تا هنگام ساخت سیستم عامل روی دیسک کاربر قرار بگیرند.

هر برنامه در ابتدای اجرا tputstart را صدا می‌زند و تعداد مشخصی پردازه محاسباتی ایجاد می‌کند که داخل آن‌ها فقط حلقه‌های تودرتو و عملیات ساده روی متغیرها انجام می‌شود تا بدون استفاده از sleep بار واقعی روی پردازنده ایجاد شود .

سپس والد با فراخوانی wait منتظر پایان همه فرزندان می‌ماند و در انتها tputend را برای محاسبه گذردهی و بعد از آن procinfo را برای چاپ وضعیت و صف و نوع هسته هر پردازه فراخوانی می‌کند .

خروجی این برنامه‌ها شامل چند سناریو با تعداد پردازه و حجم کار مختلف است که از آن‌ها برای مقایسه رفتار الگوریتم‌های نوبت گردشی روی هسته‌های E و الگوریتم FCFS روی هسته‌های P و همچنین تاثیر توازن بار و جابه‌جایی میان صف‌ها در گزارش استفاده کردیم.

```
cpu0 (E-core) tick 1464: pid=3, qticks=0
nameopidostateocpuoalgoolifetime_ticks
inito1osleepo0oRRo1464
sho2osleepo0oRRo1459
schedtesto3orunningo0oRRo26

=== CASE: nprocs=4, work=80 ===
Throughput: 4 processes in 8 ticks (~50 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1474: pid=3, qticks=0
nameopidostateocpuoalgoolifetime_ticks
inito1osleepo0oRRo1474
sho2osleepo0oRRo1469
schedtesto3orunningo0oRRo36

=== CASE: nprocs=12, work=80 ===
Throughput: 12 processes in 10 ticks (~120 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1486: pid=3, qticks=0
nameopidostateocpuoalgoolifetime_ticks
inito1osleepo0oRRo1486
sho2osleepo0oRRo1481
schedtesto3orunningo0oRRo48
$ _
```

```
File  Edit  View  Search  Terminal  Help
$ schedtest

=== CASE: nprocs=4, work=30 ===
Throughput: 4 processes in 8 ticks (~50 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1448: pid=3, qticks=0
name    pid     state   cpu     algo    lifetime_ticks
init    1       sleep   0       RR      1448
sh      2       sleep   0       RR      1443
schedtest       3       running 0       RR      10

=== CASE: nprocs=8, work=30 ===
Throughput: 8 processes in 14 ticks (~57 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1464: pid=3, qticks=0
name    pid     state   cpu     algo    lifetime_ticks
init    1       sleep   0       RR      1464
sh      2       sleep   0       RR      1459
schedtest       3       running 0       RR      26

=== CASE: nprocs=4, work=80 ===
Throughput: 4 processes in 8 ticks (~50 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1474: pid=3, qticks=0
name    pid     state   cpu     algo    lifetime_ticks
init    1       sleep   0       RR      1474
sh      2       sleep   0       RR      1469
schedtest       3       running 0       RR      36

=== CASE: nprocs=12, work=80 ===
Throughput: 12 processes in 10 ticks (~120 proc/sec)
=== after fork ===
cpu0 (E-core) tick 1486: pid=3, qticks=0
name    pid     state   cpu     algo    lifetime_ticks
init    1       sleep   0       RR      1486
sh      2       sleep   0       RR      1481
schedtest       3       running 0       RR      48
```