HW3, OS

Dr Zali

Azar, 1403

Sepehr Ebadi

9933243

۱)

الف)

Non-preemptive : زمانی که یک پروسس از حالت running به حالت waiting میرود و یا در زمانی که پروسس تمام می شود و به حالت terminate می رود گفته می شود که schedular دخیل نبوده در گرفتن cpu از پروسس بلکه پروسس با اختیار خودش به حالتی رفته که cpu را دیگر استفاده نکند و در اختیار دیگر پروسس های داخل سیستم قرار بدهد. به این حالت قبضه نشدنی و یا انحصاری نیز می گویند.

Preemptive : در واقع عکس حالت قبل می باشد به این صورت که زمانی که یک پروسس از حالت running به حالت ready می رود و یا زمانی که یک پروسس از حالت waiting به حالت ready می رود گفته می شود که در این حالت schedular دخیل بوده در این که cpu را از پروسس بگیرد و به پروسس دیگری بدهد. و در این حالت خود پروسس با اختیار خودش باعث ازاد کردن cpu نشده است بلکه schedular این کار را کرده است.

ب)

۱- زمانی که یک پروسس از حالت runnig به حالت waiting سوییچ می کند (برای مثال به درخواست I/O برخورد کرده) : در این حالت خود پروسس با اختیار خود به حالت waiting رفته است و schedular کاره ای نبوده است، پس این حالت Non-Peermptive می باشد.

۲- زمانی که یک پروسس از حالت running به حالت ready سوییچ می کند(برای مثال به interrupt برخورد کرده) : در این حالت چون پروسس کاره ای نبوده و schedular تصمیم گرفته cpu را از پروسسی که در حال اجرا است بگیرد پس این حالت Peermptive می باشد.

۳- زمانی که یک پروسس از حالت waiting به حالت ready سوییچ می کند(برای مثال زمانیکه درخواست I/O تمام شده) : در این حالت چون پروسس کاره ای نبوده و این schedular است که باید تصمیم بگیرد پروسسی که در صف waiting است را به صف ready بیاورد پس این حالت Peermtive می باشد.

۴- زمانی که یک پروسس به اتمام می رسد و به حالت terminate می ورد : در این حالت چون schedular کاره ای نیست در تمام شدن پروسس و تماما به خود پروسس بستگی دارد پس این حالت Non-Peermptive می باشد.

۲)

FCFS : خیر زیرا در این الگوریتم هر پروسسی که می آید به ترتیب آمدن به آنها cpu می دهیم و همانند صف های نانوایی می ماند و در این حالت بالاخره به هر پروسس نوبت میرسد و خدمت دریافت می کند و هیچ گاه دچار گرسنگی نخواهد شد. و این اصلی ترین مزیت این الگوریتم است.

SRTF : در این الگوریتم گرسنگی ممکن است رخ دهد زیرا در این الگوریتم در هر زمان باید چک کنیم ببینیم اگر پروسس جدید که وارد شده اگر burst time کم تری دارد باید cpu را از پروسس در حال اجرا بگیریم و به آن پروسس که جدید آمده و burst کمتری دارد

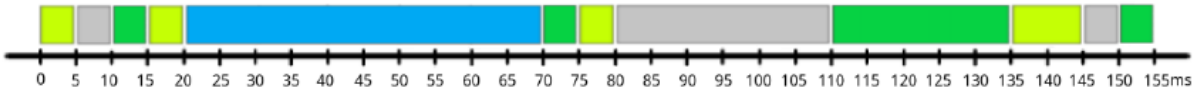
بدهیم. در این حالت ممکن است هیچ گاه نوبت به اختصاص دادن cpu به یک پروسسی که burst زیادی دارد نرسد، زیرا دائما پروسس هایی با burst کوچکتر وارد می شوند.

RR : در این الگوریتم گرسنگی نداریم به دلیل آنکه در یک حلقه مانند پروسس ها قرار می گیرند و به همه پروسس ها یک واحد زمانی یکسان نظیر می کنیم که وقتی این واحد زمانی تمام شد cpu از پروسس گرفته می شود و به پروسس بعدی داده می شود. اگر پروسسی در مرحله اول cpu burst اش تمام نشد و هنوز مانده بود در دور بعدی دوباره بعد از nامین پروسس که دوباره نوبتش می شود(چون مثل یک حلقه قرار گرفته اند پس امکان بازگشت به عضو اول هست) cpu در اختیارش قرار میگیرد و انقدر ادامه می یابد تا بالاخره burst اش تمام شود و در این حالت گرسنگی نداریم چون بالاخره به همه پروسس ها واحد زمانی مشخصی اختصاص داده ایم.

CFS : در این الگوریتم به هر پروسس سهم عادلانه ای time slice نظیر می کنیم و اینجوری نیست که به همه پروسس ها یک time slice یکسان و واحدی بدیم بلکه می خواهیم به پروسس هایی که اولویت بالاتری دارند time slice بالاتری نیز بدهیم اما چون الگوریتم عادلانه عمل می کند بالاخره به همه پروسس ها یک time slice می دهد و فقط ممکن است به پروسسی time slice خیلی کمی بدهد که باعث می شود خیلی کند جوابگوی آن پروسس باشد cpu . پس در این الگوریتم به طور معمول و کلی گرسنگی نخواهیم داشت.

۳)

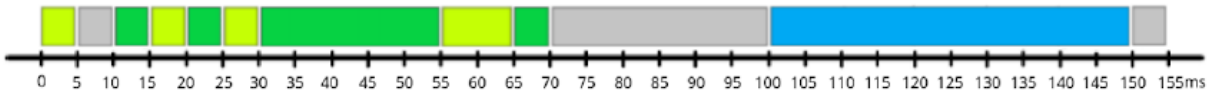
FCFS



$$response\ time = \frac{0 + (5-0) + (20-10) + (10-5)}{4} = 5$$

$$turnaround\ time = \frac{(145-0) + (150-0) + (70-10) + (155-5)}{4} = 126.25$$

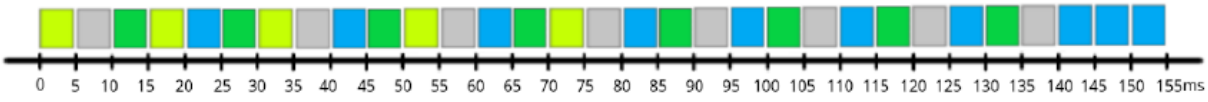
SJF



$$response\ time = \frac{0 + (5-0) + (100-10) + (10-5)}{4} = 25$$

$$turnaround\ time = \frac{(65-0) + (155-0) + (150-10) + (70-5)}{4} = 106.25$$

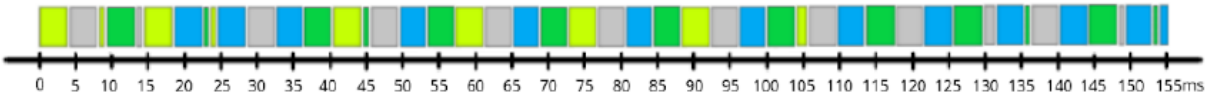
RR(quantum=5)



$$response\ time = \frac{0 + 5 + (20 - 10) + (10 - 5)}{4} = 5$$

$$turnaround\ time = \frac{(75 - 0) + (140 - 0) + (155 - 10) + (135 - 5)}{4} = 122.5$$

RR(quantum=4)



$$response\ time = \frac{0 + 4 + (19 - 10) + (9 - 5)}{4} = 4.25$$

$$turnaround\ time = \frac{(105 - 0) + (149 - 0) + (155 - 10) + (153 - 5)}{4} = 132.25$$

٤)

الف)

سیستم های چند پردازنده ای نا متقارن فقط یک پردازنده توانایی دسترسی به ساختار های داده ای را دارد و در این حالت پیچیدگی اشتراک گذاشتن داده ها کاهش می یابد. و همین طور در این نوع سیستم ها معمولا یک پردازنده به عنوان مستر عمل می کند و دیگر پروسس ها را مدیریت می کند. در این حالت پردازنده های دیگر به پردازنده اصلی وابسته هستند.

اما در سیستم های چند پردازنده ای متقارن (SMP) همه پردازنده ها به صورت مستقل عمل می کنند و می توانند وظایف خود را از یک صف مشترک یا صف های اختصاصی پردازنده دریافت کنند. این روش در سیستم های امروز رواج دارد.و در این روش از روش ها affinity نیز استفاده می کنند. و برای حفظ تعادل کارایی پردازنده ها از load balancer ها استفاده می شود.

ب)

از دید سیستم عامل پردازنده شامل ۴ هسته که هر هسته شامل ۲ ترد سخت افزاری می باشد و سیستم عامل هسته ها و ترد ها را واحد های اجرایی مجزا از هم می داند پس در اینجا سیستم می تواند ۸ ترد سخت افزاری مجزا را به صورت همزمان اجرا کند. اما ترد های

سخت افزاری یک هسته به صورت کامل نمی توان گفت مجزا از هم هستند زیرا شامل منابع مشترکی هستند از نظیر: واحد های محاسباتی و کش و ...اما تکنولوژی هایی مثل هایپر تردینگ این امکان را فراهم می کنند که تا حد امکان ترد های مجزای یک هسته مستقل به نظر بیایند.

ج)

در سطح پروسس PCS که در این سطح، رقابت بین تردهای یک فرآیند خاص برای دسترسی به منابع پردازنده انجام می‌شود. معمولاً توسط کتابخانه‌های تردها در فضای کاربر مدیریت می‌شود. و در مدل‌هایی مانند Many-to-One و Many-to-Many، تردهای کاربر به یک یا چند نخ سبک وزن (LWP) نگاشت می‌شوند و مدیریت آن‌ها داخلی است.

PCS به برنامه‌نویسان اجازه می‌دهد اولویت‌بندی و مدیریت تردها را در محدوده فرآیند خودشان تعریف کنند.

و **در سطح سیستم SCS** که رقابت بین تمام تردهای موجود در سیستم برای دسترسی به پردازنده انجام می‌شود. توسط هسته سیستم عامل مدیریت می‌شود. و معمولاً از تردهای کرنل (Kernel Threads) برای اجرای تردها روی پردازنده‌ها استفاده می‌شود.

SCS از دید کلی سیستم عمل می کند و تلاش می کند که تمام تردها به طور منصفانه یا بر اساس اولویت پردازش شوند.

۵)

الف)

در این نسخه از الگوریتم CFS استفاده شده است. CFS با هدف تحقق یک زمان‌بندی کاملاً عادلانه برای تمام فرآیندها طراحی شده است. در این الگوریتم، هر فرآیند به گونه‌ای برنامه‌ریزی می‌شود که گویی زمان‌بندی روی یک پردازنده مجازی ایده‌آل اجرا می‌شود، که در آن هر فرآیند به‌صورت مساوی زمان پردازشی دریافت می کند.

هر فرآیند یک مقدار vruntime دارد که نشان‌دهنده زمانی است که آن فرآیند روی پردازنده اجرا شده است. این مقدار با اجرای فرآیند به‌صورت خطی افزایش می‌یابد، اما برای فرآیندهایی با اولویت بالا، افزایش vruntime کندتر است. فرآیندی که کمترین vruntime را دارد، اولویت اجرا پیدا می کند.

و همینطور فرآیندها بر اساس مقدار vruntime در یک درخت Red-Black ذخیره می‌شوند. این ساختار اجازه می‌دهد که فرآیندها به ترتیب vruntime مرتب شوند.

اولویت فرآیندها با استفاده از مقدار nice value تعیین می‌شود.

در این روش پروسس هایی که vruntime کمتری دارند در اولویت بالاتری هستند و همچنین پروسس هایی که nicevalue کمتری دارند time slice بیشتری را cpu در اختیارشان قرار میدهد.

ب)

زمان‌بندی در نسخه‌های قدیمی‌تر از O(1) نبود و به تعداد فرآیندها وابسته بود. این باعث افزایش زمان سربار سیستم با افزایش تعداد فرآیندها می‌شد. در نسخه جدید الگوریتم O(1) معرفی شد، که زمان‌بندی فرآیندها در آن مستقل از تعداد فرآیندها بود. در این روش، زمان لازم برای انتخاب فرآیند بعدی برای اجرا ثابت (O(1)) باقی می‌ماند، حتی با وجود صدها یا هزاران فرآیند.
در نسخه قدیمی تر فرآیندهای با اولویت پایین ممکن بود در زمان‌بند قدیمی به دلیل پدیده گرسنگی هرگز زمان اجرا دریافت نکنند. اما در نسخه جدید الگوریتم O(1) از مفهوم (Fair Scheduling) استفاده کرد. به کمک مکانیزم Priority Rebalancing، فرآیندهایی که مدت طولانی منتظر اجرا بودند، اولویت بیشتری دریافت می کردند. این ویژگی پدیده گرسنگی را به طور مؤثری کاهش داد.
در نسخه قدیمی فرآیندها در یک صف واحد یا چند صف محدود (برای اولویت‌ها) مدیریت می‌شدند، که باعث ایجاد ازدحام و کاهش کارایی زمان‌بند می‌شد.
ولی در نسخه جدید دو صف مجزا برای فرآیندها معرفی شد: **صف فعال (Active Queue)** شامل فرآیندهایی که آماده اجرا هستند.**صف غیرفعال (Expired Queue)** شامل فرآیندهایی که سهم زمانی خود را استفاده کرده‌اند. این جداسازی باعث بهبود کارایی و کاهش سربار جابه‌جایی فرآیندها شد.
در نسخه قدیمی زمان‌بند قدیمی پشتیبانی محدود و غیربهینه‌ای از فرآیندهای بلادرنگ داشت.
اما در نسخه جدید اولویت‌های بلادرنگ (Real-Time Priorities) و زمان‌بندی بر اساس سیاست‌های FIFO و Round-Robin برای فرآیندهای بلادرنگ بهبود یافت.

ج)

مقدار-۵ نشان‌دهنده اولویت بالا است، که vruntime را آهسته‌تر افزایش می‌دهد. و مقدار+۵ نشان‌دهنده اولویت پایین است، که vruntimeرا سریع‌تر افزایش می‌دهد.

فرآیندهای CPU-boundبه طور مداوم از CPU استفاده می‌کنند و زمان بیشتری را در حال اجرا هستند، بنابراین vruntime آن‌ها سریع‌تر افزایش می‌یابد. و فرآیندهای I/O-boundبیشتر در انتظار عملیات I/O هستند و سهم کمتری از CPU دریافت می‌کنند، بنابراین vruntime آن‌ها کمتر افزایش می‌یابد.

هر دو cpu-bound :

فرآیند A (nice = -5) : اولویت بالاتری دارد، بنابراین vruntime آن کندتر افزایش می‌یابد. و سهم بیشتری از زمان CPU دریافت می‌کند.

فرآیند B (nice = +5) : اولویت پایین‌تری دارد، بنابراین vruntime آن سریع‌تر افزایش می‌یابد. و سهم کمتری از زمان CPU دریافت می‌کند.

پس فرآیند A زمان بیشتری از CPU دریافت می‌کند و فرآیند B کمتر اجرا می‌شود.

فرآیند A (I/O-bound) و فرآیند B (CPU-bound) :

فرآیند A (I/O-bound, nice = -5) : به دلیل I/O-bound بودن، زمان کمتری از CPU استفاده می‌کند، اما با مقدار nice پایین‌تر (اولویت بالا) به محض آماده شدن برای اجرا، vruntime کمی دارد و سریعاً اجرا می‌شود.

فرآیند B (CPU-bound, nice = +5) : به دلیل CPU-bound بودن، vruntime آن سریع‌تر افزایش می‌یابد و به دلیل niceبالا (اولویت پایین)، سهم کمتری از CPU دریافت می‌کند.

پس فرآیند A سریعاً اجرا می‌شود، اما به دلیل I/O-bound بودن، زمان استفاده از CPU کوتاه است. فرآیند B به آرامی اجرا می‌شود، اما سهم کلی آن از CPU نسبت به حالت اول کمتر خواهد بود.

فرآیند A (CPU-bound) و فرآیند B (I/O-bound) :

فرآیند A (CPU-bound, nice = -5) : اولویت بالایی دارد و vruntime کندتر افزایش می‌یابد، اما به دلیل CPU-bound بودن، زمان زیادی از CPU استفاده می‌کند.

فرآیند B (I/O-bound, nice = +5) : به دلیل I/O-bound بودن، زمان کمی از CPU استفاده می‌کند. با این حال، مقدار niceبالا (اولویت پایین) باعث می‌شود vruntime آن سریع‌تر افزایش یابد.

پس فرآیند A زمان بیشتری از CPU دریافت می‌کند. فرآیند B حتی زمانی که برای اجرا آماده است، به دلیل اولویت پایین‌تر و vruntimeبالاتر، دیرتر اجرا می‌شود.