دانشکده مهندسی کامپیوتر

درس سیستم‌های عامل

پاسخنامه تمرین سوم

مدرسان ..... دکتر رضا انتظاری ملکی، دکتر وحید ازهری

تیم طراح ..... عرفان زارع، هانا هاشمی

۱)

الف) thread های سطح کاربر توسط kernel ناشناخته هستند درحالیکه این قضیه برای thread های سطح kernel برعکس است. در سیستم‌هایی که از مدل‌های many-to-one یا many-to-many استفاده می‌کنند، threadهای سطح کاربر در هنگام scheduling از کتابخانه‌های thread استفاده می‌کنند و threadهای سطح kernel از کتابخانه های kernel استفاده می‌کنند. Threadهای سطح kernel نیازی ندارند تا با یک پراسس در ارتباط باشند در حالی که هر thread سطح کاربر برگرفته ازیک پراسس است. به طور کلی، threadهای سطح kernel از لحاظ نگهداری هزینه‌های بیشتری به بار می‌آورند تا thread های سطح کاربر.

ب) چون thread از پراسس کوچک‌تر است، پس می‌توان گفت به منابعی که یک thread به خود اختصاص می‌دهد به مراتب کمتر از منابع اختصاص یافته در یک پراسس است. هنگام ساخت پراسس، PCB که یک ساختمان داده سنگین برای allocating هست، ساخته می‌شود که مدیریت بر روی این ساختمان داده هزینه زمانی سنگین به دنبال می‌آورد درحالیکه thread چه در سطح کاربر چه در سطح kernel ساختمان داده‌ایی که برای memory allocating به اجرا می‌گذارد، کوچک‌تر است پس آنقدر هزینه بر نخواهد بود.

پ) در همه این الگوریتم‌ها به جز RR، امکان starvation وجود دارد:
الگوریتم fcfs یک الگوریتم preemptive است. پس اگر execution یک پراسس هرگز قطع نشود، آنگاه این امر منجر به starvation می‌شود.
در الگوریتم sjf اگر تعداد jobهای کوچکی که باید اجرا شوند بسیار زیاد باشد (بعد از هربار اجرای یک job کوچک دوباره jobایی با همان اندازه در صف ظاهر شود)، آنگاه منجر به starvation می‌شود. راه حلی که برای حل این مشکل پیشنهاد می‌شود، استفاده از aging است. به این شکل که برای هر پراسس جدیدی که initial می‌شود یک مقدار priority در نظر گرفته شود که با گذشت مقداری از زمان، این مقدار افزایش پیدا کند. پس هنگامی که اجرای یک پراسس تمام می‌شود، براساس آن مقدار priority که دارد (که به نوعی بیانگر سن آن پراسس است)، وارد صف می‌شود. این راه حل تا حدی می‌تواند starvation را در sjf حل کند.

ت) در switching بین threadها، stateهای کمتری نیاز به ذخیره و بازیابی دارند. همچنین switching بین threadها این مزیت را به دنبال دارد که می‌توان عمل caching را نیز انجام داد. پس بار هزینه‌ایی بسیار سبک‌تر از switching بین پراسس‌هاست (از آنجایی که در پراسس‌ها cache و TLB وجود ندارد)

ث) اگر فیلسوف‌ها همه به طور همزمان چنگال‌های سمت چپ خود را بردارند، دسترسی به چنگال‌های سمت راست دیگر مقدور نیست و هر کدام باید صبر کنند تا چنگال سمت راستشان (که در واقع چنگال دست چپ فیلسوف بغلی است) آزاد شود تا بتوانند از آن استفاده کنند.

۲)

خروجی همچنان عدد ۵ خواهد بود. چون پردازه فرزند، یک کپی از value را تغییر داده و در واقع حافظه مربوط به این دو پردازه مجزاست. پس هنگامی که در پردازنده والد هستیم، همچنان مقدارمان ۵ است.

۳)

در صورت سوال گفته شده که پراسس به صورت همروند در حال اجرا شدن است. پس threadهای a,b می‌توانند هم به صورت همزمان اجرا شوند و هم به صورت ترتیبی (سریالی). اول فرض می‌کنیم threadها پشت سر یکدیگر و به صورت ترتیبی اجرا می‌شوند؛ به طوریکه وقتی اجرای thread a تمام شد، thread b اجرا شود. اگر حلقه‌ی اجرای thread a تمام شود مقدار x=5 خواهد بود. thread b حلقه خود را اجرا می‌کند و پس از اتمام، مقدار x=10 است. در این حالت پس می‌دانیم مقدار x بیشتر از ۱۰ نخواهد شد. حال فرض می‌کنیم که threadها همروند (concurrent) اجرا شوند. فرض کنیم thread a اول اجرا شود. وقتی i=1 است پس مقدار x=1 خواهد شد و thread a مقدار جدید x را داخل مموری (همان shared memory مطرح شده در صورت سوال) می‌نویسد. حال thread b مقدار x را از مموری می‌خواند و یک واحد آن را افزایش می‌دهد و مقدار جدید x=2 را در مموری می‌نویسد. این روند تا اتمام دو حلقه ادامه پیدا می‌کند. در نهایت مقدار x=10 خواهد شد. پس مقدار x در هردو سناریو در بازه [۲،۱۰] قرار دارد.

۴)

توجه به این نکته ضروری است که در عمل دستیابی به ۱۰۰ درصد از مولفه‌های موازی در یک برنامه، حالت ایده‌آل است و امکان‌پذیر نیست و در هر برنامه‌ای مقدار قابل توجه‌ای از برنامه باید به شکل serial اجرا شود. اگر درصد مولفه‌های serial یک برنامه را با S نمایش دهیم و از رابطه قانون Amdahl کران بالای بهبود سرعت را به ازای core های مختلف محاسبه کنیم، خواهیم داشت:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

$$N = 2 \Rightarrow \text{speedup} \leq \frac{2}{S+1} \qquad N = 4 \Rightarrow \text{speedup} \leq \frac{4}{3S+1}$$

$$N = 8 \Rightarrow \text{speedup} \leq \frac{8}{7S+1} \qquad N = 100 \Rightarrow \text{speedup} \leq \frac{100}{99S+1}$$

همانطور که مشاهده می‌شود، هرچه تعداد هسته‌های محاسباتی (core) افزایش یابد، ضریب پشت S در مخرج کسر با شیب بسیار بیشتری نسبت به صورت کسر درحال افزایش است. این نتایج منجر به این می‌شود که از جایی به بعد به ازای S های غیرصفر، نمودار Speedup برحسب N به عدد خاصی همگرا شود. به عنوان مثال، اگر serial را ۲۵ و parallel را ۷۵ درصد درنظر بگیریم، داریم:

$$N = 2 \Rightarrow \text{speedup} \leq \frac{8}{5} = 1.6 \qquad N = 4 \Rightarrow \text{speedup} \leq \frac{16}{7} \simeq 2.3$$

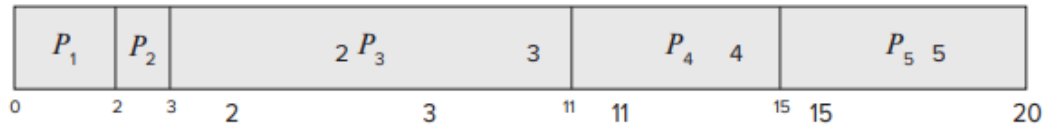
$$N = 8 \Rightarrow \text{speedup} \leq \frac{32}{11} \simeq 2.9 \qquad N = 100 \Rightarrow \text{speedup} \leq \frac{400}{103} \simeq 3.9$$

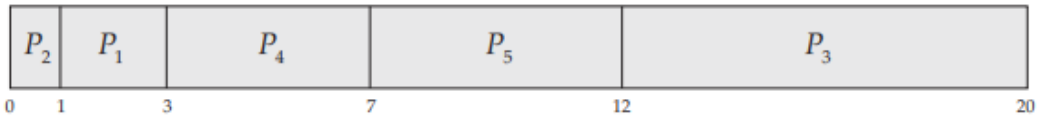
همانطور که مشخص است، speedup به عدد ۴ میل می‌نماید.

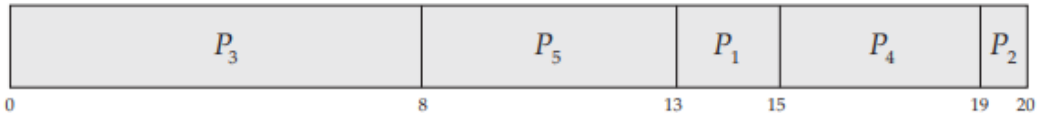
پس می‌توان گفت افزایش بیش از حد تعداد هسته‌ها تاثیر آنچنانی روی افزایش سرعت ندارد و با افزایش هسته ها صرفا هزینه بیشتری را خواهیم داشت.

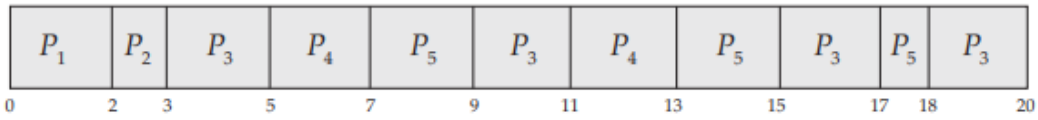
۵)

الف)









ب)

Turnaround time:

	FCFS	SJF	Priority	RR
P_1	2	3	15	2
P_2	3	1	20	3
P_3	11	20	8	20
P_4	15	7	19	13
P_5	20	12	13	18

پ)

Waiting time (turnaround time minus burst time):

	FCFS	SJF	Priority	RR
P_1	0	1	13	0
P_2	2	0	19	2
P_3	3	12	0	12
P_4	11	3	15	9
P_5	15	7	8	13

ت) SJF

۶)

الف) بله، هر تولیدکننده قبل از درج یک آیتم با سایز k، k واحد از فضای بافر را با استفاده از P(empty) نگه می‌دارد.

ب) بله، مقدار سمافور پر (full semaphore) برابر است با تعداد آیتم‌های موجود در بافر. مصرف‌کننده قبل از برداشتن یک آیتم، P(full) را اجرا میکند.

پ) خیر، تا هنگامی‌که برای هر دو آیتم در بافر فضای خالی کافی موجود باشد، دو تولیدکننده می‌توانند به صورت همزمان آیتم را درج نمایند. همچنین اگر بافر شامل تعدادی آیتم باشد ولی پر نباشد، یک تولیدکننده و یک مصرف‌کننده می‌توانند همزمان از بافر استفاده نمایند.

ت) خیر، اگر دو تولیدکننده بخواهند آیتم‌هایی با اندازه بیشتر از N/2 را درج کنند، هر تولیدکننده ممکن است نیمه دیگر فضای خالی را بخواهد. در این صورت هیچ یک از تولیدکننده‌ها نمی‌توانند تولید کنند