پاسخ نامه تمرین ششم درس سیستم‌های عامل

استاد درس: دکتر زرندی

پاییز ۹۹

سوال ۱

بر اساس قانون آمدال اگر p درصد از برنامه توسط c هسته احرا شود افزایش سرعت (speed up) برابر است با:

$$\frac{1}{(1-p)+\frac{p}{c}}$$

الف)

$$\text{Speed up} = \frac{1}{(1-0.5)+\frac{0.5}{2}} = 1.33$$

ب)

$$\text{Speed up} = \frac{1}{(1-0.5)+\frac{0.5}{4}} = 1.6$$

سوال ۲

از آنجایی که این ریسمان ها به صورت همروند اجرا می شوند ، نمیتوان از قبل در مورد نوع و ترتیب اجرای کد در هر کدام حرفی زد.

در صورتی که ریسمان ها به ترتیب ایجاد شدن کارشان را انجام دهند (یعنی اول th0 و بعد th1 و بعد th2) مقدار count به حالت **بیشینه** خود می رسد یعنی ۳۰۰۰

اما برای به دست آوردن بدترین حالت باید به معادل اسمبلی دستورات دقت کنیم. می توان دستور افزایش count را به سه دستور ۱ـ خواندن محتوای count ۲ـ افزایش آن ۳ـ نوشتن آن در حافظه تقسیم کرد.

حال اگر th0 دستور اول را اجرا کند (count=0) سپس th1 کامل اجرا شود و th2 ۹۹۹ بار اجرا شود مقدار count = 1999 است. اما در th0 مقدار count هنوز صفر است. حال اگر th0 خط دوم و سوم را انجام دهد count برابر ۱ می شود.

سپس اگر th1 برای بار ۱۰۰۰ ام (و پایانی) خط اول را اجرا کند مقدار count = 1 را می خواند.

سپس اگر th0 همه ۹۹۹ بار باقی مانده را اجرا کند و در نهایت th2 ۲ خط پایانی را اجرا کند count=2 خواهد بود (چرا که در حافظه th2 مقدار count خوانده شده برابر ۱ بود) پس count=2 مقدار **کمینه** آن خواهد بود.

سوال ۳

الگوریتم ارائه شده انحصار متقابل **ندارد** ، پیش روی **ندارد** ، انتظار محدود دارد.

انحصار متقابل نداریم ، برای مثال :

فرایند ۰ وارد enter_region می شود ، به دلیل برقرار نبود شرط:

Interested[other1] || interested[other2]

وارد بخش بحرانی می شود. سپس فرایند ۲ وارد این تابع شده و تا خط قبل از while اجرا می شود. سپس فرایند ۱ وارد شده و تا خط قبل از while اجرا می شود. حال مقدار turn برابر id فرایند ۲ است و به همین خاطر شرط turn != process برای فرایند ۲ برقرار نیست و وارد ناحیه بحرانی می شود (که فرایند ۰ هم هنوز ممکن است داخل آن باشد)

سوال ۳

پیش روی نداریم ، برای مثال :

فرض کنید دو فرایند ۰ و ۱ درخواست ورود به بخش بحرانی را داشته باشند اما فرایند ۲ نه. درین صورت هرگز فرایند های ۰ و ۱ از خط while عبور نخواهند کرد. چرا که turn برابر فرایند ۲ است و بخش interested[other1] || interested[other2] هم صحیح است.

انتظار محدود داریم چرا که تعداد دفعاتی که یک فرایند می تواند پشت سر هم وارد ناحیه بحرانی شود **محدود** است و اگر فرایند بعد درخواست ورود داشته باشد ، همان فرایند وارد خواهد شد (پس دچار قحطی نمی شویم) چرا که با هر دفعه ورود فرایند قبلی turn به فرایند بعدی داده می شود ، هرچند برای ورود فرایند بعدی **نیاز به ورود فرایند قبلی هم داریم** که مشکل عدم پیش روی را ایجاد می کرد.

سوال ۴

این راه حل که راه حل Dekker نام دارد ، همه شروط مد نظر را دارا می باشد.

روند کلی الگوریتم به این صورت است که اگر دو فرایند همزمان قصد ورود به ناحیه بحرانی را داشته باشند ، بر اساس آنکه نوبت کدام یک است فقط یکی می تواند وارد شود. و دیگری با استفاده از شروط

Wants_to_enter[0] and wants_to_enter[1]

دچار busy waiting می شود تا فرایند دیگر از ناحیه خارج شود. همچنین اولویت و نوبت ورود فرایند ها با turn تعیین می شود که با خروج هر یک از فرایند ها از ناحیه بحرانی ، این متغیر به دیگری داده می شود.

الگوریتم انحصار متقابل دارد چرا که هیچ کدام از دو فرایند قبل ازین که flag های نشان دهند قصد ورودشان را روشن کنند ، نمی توانند وارد ناحیه بحرانی شوند پس اگر هر دو قصد ورود داشته باشند لااقل یکی از دو فرایند وارد while اول خواهند شد.

این موضوع همینطور پیشروی را هم تضمین می کند چرا که اگر هردو فرایند قصد ورود داشته باشند با توجه به turn اول یکی و بعد دیگری وارد می شود، فرایندی که نوبتش نیست flag مربوط به خودش را به صفر تغییر می دهد تا وقتی که فرایندی که نوبتش است از ناحیه بحرانی خارج شود و این کار باعث می شود فرایند دیگر (که نوبتش بود) از loop خارج شده و وارد ناحیه بحرانی شود. همچنین اگر یکی از دو فرایند قصد ورود به ناحیه بحرانی را نداشته باشد ، دیگری هم دلیل برای صبر کردن ندارد و بلافاصله اجرا می شود.

سوال ۴

الگوریتم انتظار محدود دارد و دچار قحطی نمی شود. می دانیم که اگر یکی از دو فرایند قصد ورود به ناحیه بحرانی را نداشته باشند هیچ مشکلی نداریم پس فرض کنید فرایند دوم وارد ناحیه بحرانی شده و فرایند اول در حلقه while wants_to_enter[1] گیر کرده باشد. (تمام این سناریو را می توان برعکس هم تصور کرد پس فرقی ندارد) بعد ازینکه ناحیه بحرانی فرایند دوم تمام شد متغیر turn برابر صفر می شود (و تا وقتی که فرایند ۱ پیش روی نداشته باشد تغییر نخواهد کرد). پس سر انجام فرایند ۱ از while turn != 0 عبور می کند (تازه به فرض آنکه اینجا گیر کرده باشد) و wants_to_enter[0] را true می کند و منتظر می ماند تا flag مربوط به فرایند دوم (wants_to_enter[1]) هم false شود (و از آنجایی که turn برابر صفر است هرگز دستورات داخل while اولش را انجام نمی دهد.) دفعه بعد که فرایند دوم قصد ورود به ناحیه بحرانی را داشته باشد مجبود خواهد بود دستورات داخل while wants_to_enter[0] را اجرا کند یعنی اول flag خود را false کند و در حلقه while turn != 1 گیر کند (چرا که ممکن است فرایند ۱ کارش تمام نشده باشد و turn تغییر نکرده باشد) و فرایند اول که حالا دلیلی برای متوقف بودنش نیست و شروط حلقه ها برقرار نمی باشد ناحیه بحرانی خودش را اجرا کرده و سرانجام نوبت را دوباره به فرایند ۲ می دهد.

این روند باعث می شود که اگر هردو فرایند قصد ورود به ناحیه بحرانی را داشته باشند هیچ فرایندی بیش از ۱ بار پشت سر هم اجرا نشود.