امیرعلی فرازمند

99522329

تمارین تئوری فصل O.S. 6

---

1. دو پردازه همروند P_0 و P_1 کد زیر را اجرا می‌کنند و برای ورود به ناحیه بحرانی دارای رقابت هستند. i می‌تواند صفر یا یک را اختیار کند. آیا در کد زیر انحصار متقابل برقرار است؟ توضیح دهید.

```
bool flag[2];
int turn;
process i
        flag[i] = true
        while (turn != i){
                while (flag[j])
                ;
                Turn = I;
        }
        /* Critical Section */
        flag[i] = false;
```

turn	Flag [1]	Flag [0]	P1	po
0	0	0	فرض مقادیر اولیه	
0	1	1	هر دو به لوپ اول میرسند	
0	1	1	داخل لوپ اول میرود، وارد لوپ دوم هم میشود و انجا باید wait کند	وارد لوپ نمیشود و به critical-section میرود
0	1	0		از critical-section خارج میشود
0	1	0	از لوپ دوم خارج میشه(هنوز turnرو تغیر نداده)	
0	1	1		پراسس 0 همین لحظه دوباره اجرا شده و flagاش را 1 میکند،به داخل لوپ اول نمیرود و دوباره وارد ناحیه بحرانی میشود.
1	1	1	خط بعد لوپ دوم turn را تغیر میدهد	
1	1	1	از لوپ اول می اید بیرون و به ناحیه بحرانی میرود	همچنان در ناحیه بحرانی هست

در ردیف آخر همانطور که دیدید 2 پراسس همزمان داخل critical-sectionهستند و این به این معنی هست که mutual-exclusion در این الگوریتم رعایت نمیشود.

2. راه‌حل زیر را برای مسئله‌ی انحصار متقابل شامل دو فرآیند P_0 و P_1 در نظر بگیرید. فرض کنید که مقدار اولیه turn، صفر می‌باشد. کد Process P_0 در ادامه آمده است.

```
/* Other code */
while (turn != 0) /* Do nothing and wait*/
/* Critical Section */
turn = 0;
/* Other code */
```

برای فرآیند P_1، 0 را با 1 در کد بالا عوض کنید. تعیین کنید که آیا راه‌حل ارائه شده تمام شرایط مورد نیاز برای یک راه‌حل انحصار متقابل صحیح را دارد یا خیر.

Turn	P1	Po
0	وارد critical section میشود.	وارد لوپ میشود و wait میخورد
1	از ناحیه بحرانی خارج میشود و turn را 1 میکند	
1	وارد لوپ میشود و wait میخورد	وارد critical section میشود.
0		از ناحیه بحرانی خارج میشود و turn را 0 میکند

وارد حالت اولیه میشویم...

بررسی 3 شرط:

:Mutual exclusion

وقتی یکی از پراسس ها داخل ناحیه بحرانی باشه turn نمیگذاره آن یکی داخل ناحیه بحرانی اش بشود . متغیر turn این را تعیین میکنه که کدام داخل لوپ wait بخوره.

با فرض اینکه برای 2 پراسس عمل میکنیم این شرط برآورده میشود و نه برای n پراسس که معلومه که نمیتونه. 😐

:Progress & Bounded-waiting

وقتی p0 داخل ناحیه بحرانی هست و turn همپنان 0 هست اگر قبل اینکه مقدار turn را 1 کنیم ، p0 فراخوانی شود بخاطر اینکه turn همچنان مقدارش عوض نشده است ،دوباره وارد ناحیه بحرانی میشود و این موضوع هم میتواند تا ابد ادامه پیدا کند ، پس این 2 شرط را برآورده نمیشوند.

3. کد زیر را برای تخصیص (allocating) و آزادسازی (releasing) فرآیندها در نظر بگیرید.

```
while (true){
    while (true){
        flag[i] = want_in;
        j = turn;
        while (j != i){
            if (flag[j] != idle)
                j = turn;
            else
                j = (j + 1) % n;
        }
        flag[i] = in_cs;
        j = 0;
        while ((j < n) && (j == i || flag[j] != in_cs) )
            j++;
        if ((j >= n ) && (turn == i || flag[turn] == idle))
            break;
    }
    /* Critical Section */
    j = (turn + 1) % n;
    while (flag[j] == idle)
        j = (j + 1) % n;
    turn = j;
```

```
        flag[j] = idle;
        /* Remainder Section*/
}
```

الف) race condition(s) را شناسایی کنید.

ب) فرض کنید یک mutex lock با نام mutex با عملیات acquire() و release() دارید. محل قرار دادن قفل را برای جلوگیری از race condition(s) مشخص کنید.

پ) آیا می‌توانیم با جابه‌جا کردن integer variable (int number_of_processes = 0) با atomic integer (atomic_t number_of_processes =0) از race condition(s) جلوگیری کنیم؟

الف)race condition رخ نمیدهد و 3 شرط را هم برآورده میکند:

This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process i requires access to critical section, it first set sits flag variable to want in to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between turn and i are idle. (2) If so, it updates its flag to in_cs and checks whether there is already some other process that has updated its flag to in_cs. (3) If not and if it is this process's turn to enter the critical section or if the process indicated by the turn variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:
Mutual exclusion is ensured: Notice that a process enters the critical section only if the following requirements are satisfied: no other process has its flag variable set to in_cs. Since the process

sets its own flag variable set to in_cs before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.
Progress requirement is satisfied: Consider the situation where multiple processes simultaneously set their flag variables to in_cs and then check whether there is any other process has the flag variable set to in_cs. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer while(1) loop and reset their flag variables to want_in. Now the only process that will set its turn variable to in_cs is the process whose index is closest to turn. It is however possible that new processes whose index values are even closer to turn might decide to enter the critical section at this point and therefore might be able to simultaneously set its flag to in_cs. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their flag variables to in_cs become closer to turn and eventually we reach the following condition: only one process (say k) sets its flag to in_cs and no other process whose index lies between turn and k has set its flag to in_cs. This process then gets to enter the critical section.
Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the fact that when a process k desires to enter the critical section, its flag is no longer set to idle. Therefore, any process whose index does not lie between turn and k cannot enter the critical section. In the meantime, all processes whose index falls between turn and k and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the turn value monotonically becomes closer to k. Eventually, either turn becomes k or there are no processes whose index values lie between turn and k, and therefore process k gets to enter the critical section.

منبع جواب(سوال 2ش میشود)

ب) دو خط

turn =j;

flag[j] = idle;

در خط های آخر برنامه را با mutex lock کنترل میکنیم که اگر 2 پراسس به آنجا رسیدند به race-condition برنخورند.

پ) چون ما 2 متغیر را در 2 خط مختلف باعث race-condition میشوند ، استفاده از متغیر های اتمی فایده ای ندارد و بهتر است از یکی از انواع لاک استفاده شود.

4. توضیح دهید که چرا interruptها برای اجرای اصول synchronization در سیستم‌های چند پردازنده‌ای مناسب نیستند.

مشکل ما در synchronization این بود که پراسس ها به resource های یکسانی دسترسی دارند و تغیر دادن در آنها وقتی که همزمان اجرا میشوند میتواند باعث race - condition شود واسه همین پراسس ها باید بعضی وقتا صبر کنند تا کار پراسس دیگه ای روی دیتای مشترک تمام بشه. در سیستم های مالتی پروسسور وقتی روی پراسس interrupt بزنیم روی پروسسور دیگه ای میتونه اجرا کنه و وقفه ای به اون شکل ایجاد نمیشه که بتونه شرط mutual-exclusion بین 2 پراسس برآورده کنه.

7. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

   Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

5. توضیح دهید deadlock چگونه در مسئله dining-philosophers امکان‌پذیر است.

هنگامی که همه ی فیلسوف ها باهم همزمان تصمیم به غذا خوردن بگیرند، هرکدامشان چاپ استیک یکطرفش را بردارد(مثلا همه سمت چپیشان را برمیدارند). اینطور هر چاپ استیک دست یک نفر هست و همه منتظر میمانند تا آن یکی آزاد شود و آنرا بردارند که همچین چیزی رخ نمیدهد ، پس ددلاک رخ میدهد.

جواب کامل تر:

The problem arises when all of the philosophers decide to eat at the same time. Consider the case where all of the philosophers independently decide that they will try to grab the fork to their left first. When this happens, assuming all of the places at the table are occupied, then all of the forks have been taken. That is, each of the five forks shown are to the left of exactly one of the five philosophers. At this point, every philosopher has exactly one fork, but there are none available for anyone to get their second fork. Unless one of the philosophers decides to give up on eating and put a fork down, all of the philosophers will starve.

6. دو فرآیند P_1 و P_2 به صورت زیر هستند اگر فرآیندها بتوانند به صورت همروند اجرا شوند و امکان اجرای آن‌ها به صورت چند در میان نیز وجود داشته باشد، در صورتی که مقدار اولیه متغیر مشترک a برابر صفر باشد، بعد از اجرای کامل رو فرآیند، مقادیر a، b و c چه تغییری می‌کنند؟

P_1	P_2
b = a;	a = 2;
c = a + 1;	

---

a was initialized to 0;

All 3 possible cases:

1)

P2

P1(line1)

P1(line2)

➔ a = 2, b = 2, c = 2

2)

P1(line1)

P2

P1(line2)

➔ a = 2, b = 0, c = 3

3)

P1(line1)

P1(line2)

P2

➔ a = 2, b = 0, c = 1