دانشگاه صنعتی اصفهان
تکلیف سوم درس سیستم عامل (ترم اول ۱۳۹۹)
سارا برادران (شماره دانشجویی : ۹۶۲۴۱۹۳)

۱ سوال

الف)

سطح ۱ : $Max[1] < 1 + E \Rightarrow 3 < 1 + E \Rightarrow 2 < E$

سطح ۲ : $Max[2] + Max[3] + Max[4] < 3 + E \Rightarrow 3 + 2 + 2 < 3 + E \Rightarrow 4 < E$

سطح ۳ : $Max[5] < 1 + E \Rightarrow 8 < 1 + E \Rightarrow 7 < E$

نتیجه : حداقل مقداری که E می تواند اخذ کند برابر ۸ است لذا حداقل به ۸ نسخه از منبع R۱ نیازمند هستیم.

ب)

۲ سوال

ابتدا برای هر فرآیند اختلاف حداکثر منبع مورد نیاز و منابع اختصاص یافته را به ازای هر کدام از منابع R۱ و R۲ محاسبه می کنیم.

جدول ۱: اختلاف حداکثر منبع مورد نیاز و منبع اختصاص یافته

فرآیند	R۱	R۲
p1	۵	۰
p2	۰	۵
p3	۰	۵
p4	۰	۳
p5	۹	۵

با توجه به اینکه در گام اول تنها تعداد ۳ منبع آزاد از R۲ موجود است پس اولین فرآیند اجرایی نمی تواند p۲، p۳، p۵ باشد حال اگر اولین فرآیند اجرایی p۱ باشد در همین مرحله می توان گفت مقدار x در این مرحله می بایست حداقل ۵ باشد اما اگر اولین فرآیند اجرایی فرآیند p۴ باشد آنگاه می توان گفت تعداد ۳ منبع باقی مانده R۲ را در اختیار گرفته و بدون نیاز به منبع اضافه از R۱ اجرا می شود در انتها ۴ منبع R۲ و ۱ منبع R۱ را ازاد خواهد کرد پس تا بدین مرحله می توان گفت حداقل مقدار x می تواند ۰ باشد در گام دوم همچنان تعداد منبع ازاد R۲ کمتر از ۵ و برابر ۴ است پس تنها فرآیندی که می تواند در گام دوم اجرا شود فرآیند p۱ است و از آنجایی که این فرآیند به حداکثر ۵ منبع R۱ اضافه برای اجرا نیازمند است و تعداد x+۱ منبع R۱ فی الحال ازاد داریم پس مقدار x تا به این جا می تواند حداقل ۴ باشد.فرآیند p۴ هیچ منبع اضافه ی R۲ احتیاج ندارد پس از خاتمه اجرای فرآیند p۱ تعداد ۲ منبع R۲ و ۵ منبع R۱ آزاد می گردد و تعداد منابع R۲ باقی مانده برابر ۶ و تعداد منابع باقی مانده R۱ با احتساب x=۴ برابر ۵ خواهد شد در این مرحله هر یک از فرآیند های p۲ ، p۳ می توانند اجرا شوند اگر فرآیند p۲ در گام سوم اجرا شود ۵ منبع اضافه R۲ را اخذ کرده و به هیچ منبع اضافه R۱ احتیاج ندارد در انتها نیز ۱۰ منبع R۲ و ۲ منبع R۱ آزاد می کند و لذا در این گام تعداد منابع باقی مانده R۲ برابر ۱۱ و تعداد منابع باقی مانده R۱ برابر ۷ خواهد بود در گام بعدی تنها فرآیند p۳ قابل اجرا است که به ۵ منبع R۲ اضافه برای اجرا نیازمند بوده و هیچ منبع اضافه ای از R۱ برای اجرا نیاز ندارد پس از خاتمه آن نیز تعداد ۵ منبع R۲ و تعداد ۴ منبع R۱ آزاد می کند و لذا در این مرحله تعداد

۱۱ منبع R۲ و تعداد ۱۱ منبع آزاد موجود خواهد بود به عنوان آخرین فرآیند p۵ می تواند اجرا گردد و به حداکثر ۵ منبع اضافه R۲ و حداکثر ۹ منبع اضافه R۱ برای اجرا نیازمند است پس از خاتمه نیز ۵ منبع R۲ و ۹ منبع R۱ را آزاد می کند و تعداد کل منابع باقی ماند در انتهای اجرای همه فرایند ها برای هر کدام از R۱ و R۲ برابر ۱۱ خواهد بود.
نتیجه : پس با در نظر گرفتن مقدار حداقل ۴ برای x یک مسیر امن برای اجرای فرآیند ها وجود خواهد داشت.

۳ سوال

الف) با توجه به اینکه عملیات فرآیند p۱ موجب افزایش مقدار s و عملیات فرآیند p۲ موجب کاهش مقدار s در هر گام می شود لذا می توان گفت برای رسیدن به حداکثر مقدار s می بایست پس از هر دستور move s, reg که توسط فرآیند p۲ اجرا می شود حتما یک دستور move s, reg توسط فرآیند p۱ اجرا گردد به علاوه دستور move reg, s توسط فرآیند p۱ نیز می بایست همواره پیش از دستور ذخیره سازی رجیستر در متغیر s فرآیند p۲ و پس از دستور ذخیره سازی رجیستر در متغیر s فرآیند p۱ اجرا گردد یک توالی مناسب از وقفه ها برای رسیدن به بیشترین مقدار s را در زیر می توان مشاهده کرد. در حالت کلی مقدار هایی که فرآیند p۲ در رجیستر می نویسد همواره می بایست توسط فرآیند p۱ بازنویسی(rewrite) شود. نهایتا حداکثر مقداری که می توان برای s بدست آورد ۳۶ می باشد بدین ترتیب می توان به گونه ای وقفه ها را تنظیم کرد که در انتها به نظر برسد فرآیند p۲ هیچ عملیاتی بر روی متغیر s نداشته است.

جدول ۲: نحوه اجرای فرآیند ها

P۲	P۱
	Move reg, s
	Mul reg, 3
Move reg, s	
Div reg, 2	
Move s, reg	
	Move s, reg
	Move reg, s
	Mul reg, 3
Move reg, s	
Div reg, 2	
Move s, reg	
	Move s, reg

ب) در این قسمت هر دو فرآیند p۱ و p۲ موجب افزایش مقدار s می گردند لذا می بایست به دنبال فرآیندی بود که مقدار S را کمتر افزایش می دهد و گام اول هر دو فرآیند منجر به ۴ شدن مقدار s می شود اما در گام دوم فرآیند p۱ منجر به ۸ شدن مقدار s و فرآیند p۲ منجر به ۶ شدن مقدار s می شود پس کافیست دستورات را در جایی متوقف کرده و تعویض متن انجام دهیم که دومین دستور ذخیره سازی رجیستر در متغیر s مربوط به فرآیند p۲ پس از دومین دستور ذخیره سازی رجیستر در متغیر s مربوط به فرآیند p۱ اجرا شود تا منجر به rewrite شدن مقدار ۸ با ۶ شود.یک توالی مناسب از وقفه ها برای رسیدن به کمترین مقدار s را در زیر می توان مشاهده کرد.

جدول ۳: نحوه اجرای فرآیند ها

P۱	P۲
	Move reg, s
	ADD reg, 2
Move reg, s	
MUL reg, 2	
Move s, reg	
	Move s, reg
	Move reg, s
	ADD reg, 2
Move reg, s	
MUL reg, 2	
Move s, reg	
	Move s, reg

۴ سوال

الف) در راه حل قسمت الف اگر فرض شود برای نخستین بار ترد ۱ قصد ورود به ناحیه بحرانی را داشته باشد و فی الحال هیچ ترد دیگری در حال استفاده از ناحیه بحرانی نباشد لذا ترد ۱ مقدار n[۱] را true کرده و از آنجایی که مقدار اولیه turn برابر ۰ و مخالف ۱ است وارد حلقه اول می گردد به علاوه n[۰] نیز مقدار false دارد و لذا شرط حلقه دوم بر قرار نبوده در نتیجه ترد ۱ از حلقه دوم عبور خواهد کرد حال اگر پیش از آنکه مقدار turn را برابر id خود قرار دهد عمل context switch رخ داده و ترد ۰ بر روی cpu قرار گیرد آنگاه این ترد نیز مقدار n[۰] را true کرده و از آنجایی که مقدار turn هنوز برابر ۰ است و $0 = !0$ نمی باشد ترد ۰ از حلقه اول عبور کرده و وارد ناحیه بحرانی می گردد حال اگر پیش از آنکه ترد ۰ ناحیه بحرانی را رها کند مجددا context switch رخ داده و تردا بر روی cpu اجرا شود بلافاصله مقدار turn را ۱ کرده و مجددا شرط حلقه اول چک می گردد و از آنجایی که $1 = !1$ نیست ترد ۱ نیز از حلقه اول عبور کرده و وارد CS می گردد به این ترتیب شرط mutual exclusion عملا نقض شده و دو ترد همزمان توانسته اند وارد ناحیه بحرانی شوند.

ب) در راه حل قسمت ب اگر فرض شود برای نخستین بار ترد ۱ قصد ورود به ناحیه بحرانی را داشته باشد و فی الحال هیچ ترد دیگری در حال استفاده از ناحیه بحرانی نباشد لذا ترد ۱ مقدار n[۱] را true کرده و از آنجایی که مقدار اولیه turn برابر ۰ و مخالف ۱ است وارد حلقه اول می گردد به علاوه n[۰] نیز مقدار false دارد و لذا شرط حلقه دوم بر قرار نبوده در نتیجه مقدار turn هیچ تغییری نمی کند و همچنان ترد ۱ در حلقه اول گردش خواهد کرد و در صورتی که ترد ۰ هرگز قصد ورود به ناحیه بحرانی را نداشته باشد عملا ترد ۱ نه تنها نمی تواند وارد CS شود بلکه در داخل یک spin lock گیر خواهد افتاد. به علاوه اگر فرض شود برای نخستین بار ترد ۰ قصد ورود به ناحیه بحرانی را داشته باشد آنگاه مقدار n[۰] را true کرده و با توجه به اینکه مقدار turn برابر ۰ است لذا $0 = !0$ نبوده و ترد ۰ از حلقه اول عبور کرده و وارد ناحیه بحرانی می شود حال اگر پیش از آنکه ترد ۱ قصد کند وارد CS شود ترد ۰ ناحیه بحرانی را رها کرده و n[۰] را false کند مجددا سناریو ذکر شده در بالا تکرار خواهد شد و لذا امکان ورود ترد ۱ به CS تنها زمانی فراهم می گردد که همزمان با رسیدن ترد ۱ ترد ۰ ناحیه بحرانی را در دست داشته باشد و n[۰] برابر true باشد در غیر اینصورت هرگز مقدار turn تغییری نخواهد کرد.

۵ سوال

اگر فرض شود در حین یکبار اجرای barrier-done تمامی ترد ها بلافاصله پس از اجرای دستور count++ متوقف شده و contxtswitch رخ دهد و ترد دیگری بر روی cpu قرار گیرد آنگاه در ادامه اجرای همه نخ ها زمانی خواهد بود که count مقدار N اتخاذ کرده است و لذا هیچ یک از ترد ها وارد بلاک if نشده و همگی وارد بلاک else می شوند سپس به ازای هر ترد $N-1$ مرتبه post(barrier) اجرا می شود و لذا عملا مقدار متغیر barrier برابر $N \times (N-1)$ خواهد شد حال اگر بار بعدی barrier-done فراخوانی شود و تمام ترد ها به جز ترد آخر پس از عمل count++ وارد بلاک if شوند آنگاه علی رغم آنکه wait(barrier) می بایست تمام آن ها را تا رسیدن ترد آخر متوقف کند اما به دلیل اینکه مقدار barrier اکنون بیش از ۰ و حتی بیش از N است لذا هیچ تردی در این مرحله متوقف نشده و همگی ترد ها از barrier عبور خواهند کرد پیش از آنکه ترد های دیگر به آن نقطه رسیده باشند.

۶ سوال

الف) توابع به صورت زیر نوشته شده اند.

```
semaphore empty1 = 1;
semaphore empty2 = 1;
semaphore full1 = 0;
semaphore full2 = 0;

Proccess_A(){
    while(1){
        wait(empty1);
        read(buffer1, "file.txt");
        post(full1);

    }
}
Proccess_B(){
    while(1){
        wait(full1);
        copy(buffer2, buffer1);
        post(full2);
    }
}
Proccess_C(){
    while(1){
        wait(full2);
        print(buffer2);
        post(empty1);
    }
}
```

ب) توابع به وسیله conditionVariable بازنویسی شده است.

```
pthread_cond_t cv1;
pthread_cond_t cv2;
pthread_cond_t cv3;
pthread_mutex_t mutex;

int A_done = 0;
int B_done = 0;
int C_done = 1;

Proccess_A(){
    while(1){
        mutex_lock(&mutex);
        while(C_done == 0)
            cond_wait(&cv1, &mutex);

        read(buffer1, "file.txt");
        A_done = 1;
        C_done = 0;
        cond_signal(&cv2);
        mutex_unlock(&mutex);
    }
}
Proccess_B(){
    while(1){
        mutex_lock(&mutex);
        while(A_done == 0)
            cond_wait(&cv2, &mutex);

        copy(buffer2, buffer1);
        B_done = 1;
        cond_signal(&cv3);
        mutex_unlock(&mutex);
    }
}
Proccess_C(){
    while(1){
        mutex_lock(&mutex);
        while(B_done == 0)
            cond_wait(&cv3, &mutex);

        print(buffer2);
        C_done = 1;
        cond_signal(&cv1);
        mutex_unlock(&mutex);
    }
}
```

۷ سوال

در این الگوریتم واضح است که هر ترد بلافاصله پس از ورود به بلاک do مقدار turn را برابر id خود قرار می دهد و فلگ مربوط به خودش را نیز True می کند این در حالی است که شرط while موجود در بلاک do تنها زمانی برقرار است که هم فلگ مربوط به ترد دیگر True بوده و هم مقدار turn برابر id ترد دیگری باشد (علی رغم اینکه همواره پس از ورود هر ترد turn برابر id همان ترد قرار می گیرد) در غیر اینصورت شرط حلقه برقرار نبوده و ترد وارد ناحیه بحرانی میگردد واضح است که اگر مثلا ترد صفر در ابتدای کار flag[۰] را True کرده و turn را نیز ۰ کند آنگاه به دلیل اینکه turn == ۱ نیست شرط حلقه نقض شده و از حلقه عبور کرده و ناحیه بحرانی را در اختیار می گیرد حال در همین حین عمل contextswitch رخ داده و ترد ۱ بر روی cpu قرار گرفته و قصد ورود به ناحیه بحرانی را دارد لذا flag[۱] را True کرده و turn را برابر ۱ قرار می دهد و لذا شرط turn == ۰ نقض شده و این ترد هم وارد ناحیه بحرانی میگردد پس عملا mutual exclosion نقض می شود. در حالت کلی الگوریتم پترسون مبتنی بر دادن نوبت به ترد مقابل می باشد در حالی که در این پیاده سازی هر ترد نوبت را به خودش می دهد و لذا نیاز است دستور turn = i با turn = j جایگزین گردد.

۸ سوال

خیر الگوریتم ذکر شده عملکرد صحیحی ندارد چراکه اگر یک ترد زمانی که ناحیه بحرانی در اختیار هیچ ترد دیگری قرار ندارد قصد کند ناحیه بحرانی را در اختیار بگیرد وارد حلقه اول ((while(true)) شده و چون شرط $lock! = 0$ برقرار نبوده از حلقه دوم عبور می کند حال چنانچه contextswitch رخ دهد و پیش ازینکه ترد مورد نظر مقدار lock را یک کند ترد بعدی بر روی cpu ران شود آنگاه به دلیل اینکه همچنان مقدار lock صفر بوده ترد دوم نیز از حلقه داخلی عبور و مقدار lock را یک کرده و وارد ناحیه بحرانی می شود حال اگر زمانی که ترد دوم در ناحیه بحرانی است مجددا ترد اول بر روی cpu قرار گیرد آنگاه او نیز

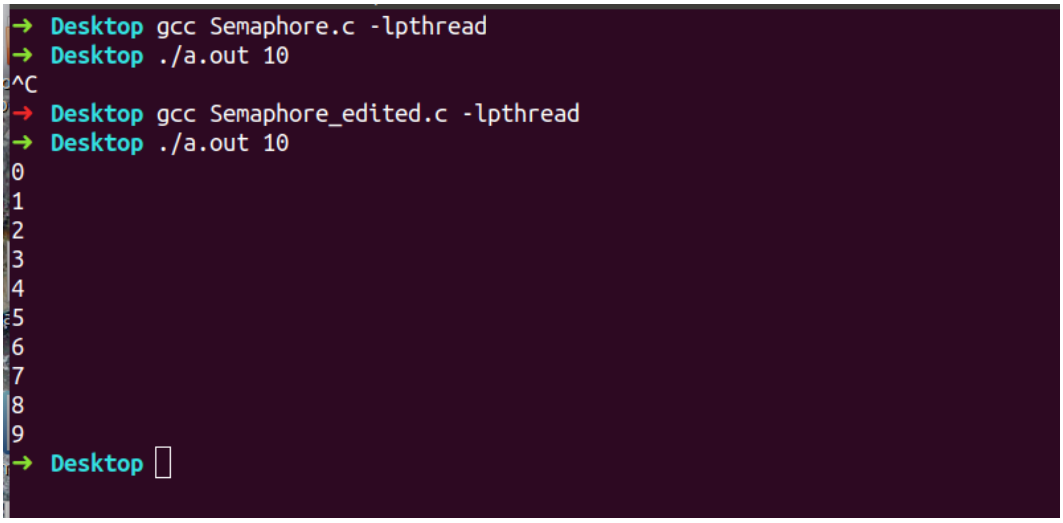
lock را یک کرده و وارد ناحیه بحرانی می شود غافل از آنکه یک ترد دیگر داخل ناحیه بحرانی است و بدین ترتیب شرط اولیه قفل یعنی mutual excposion نقض می گردد.

۹ سوال

الف) همانطور که مشخص است سه عدد سمافور در این کد تعریف شده است سمافور full مقدار اولیه ۰ دارد و لذا این بدان معناست که پیش از اینکه حداقل یکبار تابع sem-post(&full) فراخوانی شود هرگز نمیتوان وارد بلاکی شد که با sem-wait(&full) محافظت می شود. به علاوه سمافور mutex با مقدار اولیه ۱ می تواند نقش قفل را ایفا نماید و تنها اجازه ورود همزمان یک ترد به بلاکی که با sem-wait(&mutex) محافظت می شود را بدهد. با توجه به اینکه در توابع تولید کننده و مصرف کننده ابتدا sem-wait(&mutex) فراخوانی شده لذا تنها یک ترد تولید کننده و یا یک ترد مصرف کننده می تواند وارد بلاک شود و دیگری در تابع sem-wait(&mutex) متوقف خواهد شد به علاوه در گام بعد تولید کننده یا مصرف کننده وارد شده با sem-wait(&full) مواجه می گردد و لذا در این مکان قفل شده و از آنجایی که هیچ یک از این دو ترد نمی تواند تابع sem-post() را فرابخواند پس هر گز نمی توانند از sem-wait() ها خارج شوند لذا مسئله با بن بست مواجه خواهد شد.

ب) این کد بدین گونه اصلاح شده است که از یک سمافور (سمافور empty با مقدار اولیه MAX) استفاده می شود به این سبب که درهر بار ورود به تابع تولید کننده چک کنیم که بافر پر نشده باشد در غیر اینصورت اجازه تولید داده نمی شود تا زمانی که یک مصرف کننده حداقل یک خانه از بافر را مصرف کرده و مجددا تولید کننده بتواند تولید خود را داخل بافر ذخیره نماید. لذا پس از ورود به تابع تولید کننده sem-wait(&empty) را فراخوانی می نماییم تا در صورتی که تمام MAX خانه بافر پر شده باشد ترد تولید کننده در sem-wait متوقف گردد. به علاوه پس از اینکه یک مصرف کننده به وسیله تابع get یک خانه از بافر را خالی می نماید دستور sem-post(&empty) فراخوانی می گردد تا اگر یک تولید کننده در sem-wait(&empty) متوقف شده آزاد گشته و بتواند تولید خود را انجام دهد. به علاوه از یک سمافور دیگر (سمافور full با مقدار اولیه ۰) استفاده می شود به این سبب که در هر بار ورود به تابع مصرف کننده چک کنیم که بافر خالی نشده باشد در غیر اینصورت اجازه مصرف داده نمی شود تا زمانی که یک تولید کننده حداقل یک خانه از بافر را پر کرده و مجددا مصرف کننده بتواند مقدار تولید شده را مصرف کند. لذا پس از ورود به تابع مصرف کننده sem-wait(&full) را فراخوانی می نماییم تا در صورتی که بافر خالی شده باشد ترد مصرف کننده در sem-wait متوقف گردد. به علاوه پس از اینکه یک تولید کننده به وسیله تابع put یک خانه از بافر را پر می نماید دستور sem-post(&full) فراخوانی می گردد تا اگر یک مصرف کننده در sem-wait(&full) متوقف شده آزاد گشته و بتواند مصرف خود را انجام دهد.

از سمافور mutex با مقدار اولیه ۱ نیز به منظور قفل استفاده شده است تا عمل put و get به شکل atomic صورت پذیرد. کد اصلاح شده تحت هنوان semaphore-edited.c ضمیمه شده و نتجه اجرای دو کد اولیه و اصلاح شده را در تصویر زیر می توان مشاهده کرد.



۱۰ سوال

الف) در تابع make-thread نخ ها توسط pthread-create ایجاد شده و قرار بر آن است که تابع entry را اجرا نمایند به علاوه همه نخ ها یک متغیر مشترک از جنس ساختار *shared را به عنوان آرگومان به تابع entry پاس می دهند. تابع

join-thread نیز با استفاده از pthread-join در انتظار خاتمه کار نخ ها می ایستد. تابع entry آرگومان ورودی که متغیر مشترک از جنس *shared بود را دریافت کرده و تابع ()child-code را با آرگومان مذکور فراخوانی می کند. حال به طور دقیق تر نحوه کار و هدف اجرای این برنامه را بررسی می کنیم : آنچه هدف این برنامه به نظر می رسد این است که در وهله اول یک ساختار تحت عنوان shared ایجاد می کنیم این ساختار شامل یه متغیر counter یک متغیر end و یک آرایه array از اعداد صحیح است در داخل متغیر end تعداد اعضای آرایه نگهداری می شود. تعداد ۲ ترد ایجاد گشته و حال قصد داریم هربار یکی از ترد ها یکی از اعضای آرایه مذکور را که در ابتدا تماما با صفر مقدار دهی شده اند یک واحد اضافه نماید و توسط متغیر counter کنترل می گردد که هر ترد مقدار کدام یک از خانه های آرایه را([array[counter) را یک واحد اضافه نماید به علاوه هر ترد پس از انجام عملیات افزایش یک واحدی [array[counetr مقدار counter را یک واحد زیاد می کند تا ترد بعدی خانه counter+۱ ام از آرایه را افزایش دهد و به این ترتیب تمام خانه های آرایه که ابتدا با صفر مقدار دهی شده اند در انتها می بایست مقدار ۱ داشته باشند زیرا هر خانه توسط یکی از ترد ها یک واحد افزایش یافته در انتهای تابع child-code نیز هر زمان که مقدار counter به مقدار end ساختار رسید تابع خاتمه می پذیرد زیرا این بدان معناست که همه اعضای آرایه پیموده شده و یک واحد افزایش یافته اند. پس از انتهای کار دو ترد تابع check-array مقدار تمام اعضای ارایه (۰ تا ۱-end) را چک کرده و درصورتی که هر کدام مقدار ۱ نداشته باشد یک واحد به متغیر error اضافه می نماید. حال مشاهده می شود که پس از خاتمه کار ترد ها مقدار error برابر صفر نیست و تعداد زیادی از خانه های آرایه مقدار ۱ ندارند علت این موضوع مشخص است در حقیقت ۲ دستور shared->array[shared->counter] ++ و counter ++ شامل بیش از یک دستور اسمبلی هستند و لذا اگر به شکل atomic اجرا نگردد و میانه هر یک ازین دستورات اسمبلی context switch رخ دهد آنگاه نتیجه کاملا در تضاد با آنچیزی خواهد بود که مورد انتظار است.

```
99930000
99930000
99940000
99940000
99950000
99950000
99960000
99960000
99970000
99970000
99980000
99980000
99990000
99990000
99990000
100000000
Child done.
100000000
Child done.
Checking...
12319387 errors.
→ Desktop gcc counter.c -lpthread
```

ب) از ساختار semaphore موجود در کد استفاده کرده و یک lock برای دو دستوری که در بالا ذکر شد و ناحیه بحرانی محسوب می شوند قرار می دهیم در ادامه تصویر اجرای کد تصحیح شده نشان می دهد تعداد ارور ها به صفر رسیده است. کد تصحیح شده تحت عنوان counter.c ضمیمه شده است.

```
99900000
99900000
99910000
99910000
99920000
99920000
99930000
99930000
99940000
99950000
99960000
99960000
99970000
99970000
99980000
99990000
99990000
100000000
Child done.
100000000
Child done.
Checking...
0 errors.
→ Desktop gcc counter_edited.c -lpthread
```

۱۱ سوال

الف)

Mutex : The full form of Mutex is Mutual Exclusion Object. It is a special type of binary semaphore which used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. It allows current higher priority tasks to be kept in the blocked state for the shortest time possible. However, priority inheritance does not correct priority- inversion but only minimizes its effect.

Semaphore : Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1)wait, and 2) signal for the process synchronization. A semaphore either allows or disallows access to the resource, which depends on how it is set up.

DIFFERENCES

- Mutex is a locking mechanism whereas Semaphore is a signaling mechanism
- Mutex is just an object while Semaphore is an integer
- Mutex has no subtype whereas Semaphore has two types, which are counting semaphore and binary semaphore
- Semaphore supports wait and signal operations modification, whereas Mutex is only modified by the process that may request or release a resource
- Semaphore value is modified using wait () and signal () operations, on the other hand, Mutex operations are locked or unlocked

در حقیقت semaphore می تواند در نقش mutex ظاهر شود تنها کافیست مقدار اولیه آن را برابر ۱ تنظیم کرده و از تابع sem-wait() در نقش mutex-lock() و از تابع sem-post() در نقش mutex-unlock() استفاده نماییم.

۱۲ سوال

کد نوشته شده تحت عنوان Question۱۲.c ضمیمه شده است.