آریا خلیق
۹۵۲۴۰۱۴
bartararya@gmail.com
تمرین سوم درس سیستم‌عامل

تمرین ۱)
بررسی منطق:
در این روش از flag استفاده شده است به گونه‌ای که مشخص کند دقیقاً کدام process آماده ورود به ناحیه بحرانی خود می‌باشد.
زمانی که یک فرایند آماده و متقاضی ورود به C.S خود می‌باشد flag خود را true می‌کند سپس در شرط while به فرایند دیگر نگاه می‌کند. اگر فرایند دیگر متقاضی ورود به C.S خود نباشد یعنی flag آن برابر false باشد وارد C.S خود می‌شود در غیر این صورت وارد حلقه while می‌شود. اگر متغیر turn برابر شماره فرایند دیگر بود به این معنی است که نوبت فرایند دیگر است و با توجه به اینکه flag آن هم ۱ بوده است فرایند فعلی flag خود را موقتاً برابر ۰ می‌کند تا فرایند دیگر بتواند وارد C.S خود شود و فرایند اصلی ما در شرط turn == 1 گیر می‌کند و تا زمانی که فرایند دیگر از C.S خود خارج شود و turn را برابر فرایند اصلی کند منتظر می‌ماند. پی از آنکه از while بیرون آمد flag خود را به نشانه درخواست ورود و ورود به ناحیه بحرانی برابر ۱ می‌کند و وارد ناحیه بحرانی می‌شود. در آخر نوبت را به دیگری داده و flag خود را ۰ می‌کند.
در حقیقت با استفاده از ترکیب flag و turn مشکل deadlock و livelock را رفع می‌کنیم.

بررسی ارضای شروط:
شرط انحصار متقابل)
این شرط را به درستی رعایت می‌کند.
از زمانی که هر فرایند بخواهد وارد ناحیه بحرانی خود شود flag خود را ۱ می‌کند که همین کار باعث ماندن فرایند دیگر در while شده و مانع ورود آن می‌شود.
شرط پیش‌روی)
این شرط را به درستی رعایت می‌کند.
زمانی که فرایندی نه داخل ناحیه بحرانی است و نه منتظر ورود فرایند دیگر می‌تواند بارها وارد و خارج شود چون در این حالت شرط while اشتباه است و مانع دیگری هم در راه خود ندارد.
شرط انتظار محدود)
این شرط را به درستی رعایت می‌کند.
فرض کنیم فرایند ۲ در انتظار ورود است و فرایند ۱ داخل C.S شده. در این حالت فرایند ۱ پس از خروج turn را برابر ۱ می‌کند و وقتی دوباره می‌خواهد وارد C.S خود شود وارد حلقه while شده و شرط if آن درست است و وارد if شده و داخل while با شرط turn == 1 بود گیر می‌کند به همین دلیل فرایند ۲ می‌تواند وارد C.S شود.
بدترین حالت زمانی است که فرایند ۲ در شرط turn == 0 گیر کرده و flag خود را عمداً ۰ کرده است که فرایند ۱ بعد از تمام شدن C.S خود turn را برابر ۱ می‌کند و دوباره وارد C.S خود می‌شود (چون فعلاً flag فرایند دیگر برابر ۰ است) ولی این موضوع زمانی که فرایند ۲ دارای CPU شده و از حلقه با شرط turn == 0 بیرون می‌آید و flag خود را برابر ۱ می‌کند تمام می‌شود و دیگر فرایند اول نمی‌تواند تا زمانی که فرایند دوم وارد C.S خود شده و turn را تغییر دهد وارد C.S شود.

تمرین ۲)
در semaphor یک صف داریم که فرایندهایی که بلاک شده‌اند داخل آن صف رفته و به حالت waiting می‌روند تا زمانی که از آن بیرون بیایند.
در سمافور قوی این صف به صورت FIFO پیاده‌سازی می‌شود یعنی فرایندی که اول به صف وارد شده اول از همه از صف خارج می‌شود ولی در سمافور ضعیف ترتیب خروج از صف waiting مشخص نیست.
سمافور ضعیف نیاز به حافظه کم‌تری دارد ولی سمافور قوی حافظه بیشتری برای نگه‌داری ترتیب نیاز دارد ولی در عوض عدالت بیشتری هم دارد.

تمرین ۳)
فرض کنیم در مانیتور سناریو زیر اتفاق می‌افتد:
یک فرایند در مانیتور در حال انجام کاری است و به دلیل این که باید شرطی برقرار باشد در حال حاضر نمی‌تواند آن کار را انجام دهد تا زمانی که شرط درست شود.
در این حالت درست و منطقی نیست که مانیتور را به این فرایند اختصاص دهیم زیرا منابع ارزشمندمان هدر می‌رود این فرایند باید بلاک شود و فرایند دیگر کارش را در مانیتور انجام دهد و زمانی که شرط درست شد فرایند اول کارش را ادامه دهد.
این کار به وسیله دو متغیر شرطی انجام می‌شود:
Cwait: وظیفه این متغیر شرطی بلاک کردن فرایندی است که شناسه آن فرایند به عنوان آرگومان ورودی به آن داده می‌شود.
Csignal: شناسه فرایندی که بلاک شده به عنوان آرگومان ورودی به آن داده می‌شود و آن را از حالت بلاک در می‌آورد و فرایند مانند قبل می‌تواند به کار خود ادامه دهد.

تفاوت این متغیرهای شرطی با سمافور:
یا wait و signal در سمافور متفاوت است چون برخلاف سمافور ساختمان‌داده‌ای برای نگه‌داری signal یا wait که برای یک فرایند ارسال می‌شود وجود ندارد.

تمرین ۴)

بخش الف)

انحصار متقابل: این نیاز را ارضاء می‌کند.

زمانی که flag یکی true باشد دیگری نمی‌تواند از شرط if آخر رد شود به همین دلیل دو فرایند نمی‌توانند هر دو هم‌زمان از شرط if آخر رد شوند

پیشرفت: این نیاز را ارضاء می‌کند.

زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند فرایند دیگر بارها می‌تواند وارد و خارج شود چون flag فرایند دیگر 0 است.

انتظار محدود: این نیاز را ارضاء نمی‌کند.

فرایند اول می‌تواند بارها وارد ظود و زمانی که flag آن برابر true است فرایند دیگر شرط flag[A]!=0 را چک کرده و چون درست است همواره به خط ۱ می‌رود و فرایند اول C.S خود را تمام می‌کند و دوباره خواستار ورود می‌شود و فلگ را true می‌کند و فرایند دوم دوباره if را چک کرده و به خط اول می‌رود و بی‌نهایت بار می‌تواند اتفاق بیافتد.

بخش ب)

انحصار متقابل: این نیاز را ارضاء می‌کند.

هر زمان یکی وارد C.S شود flag آن قطعاً ۱ است و به همین دلیل دیگری در شرط while گیر می‌کند.

پیشرفت: این نیاز را ارضاء می‌کند.

زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند شرط while غلط خواهد بود و فرایند دیگر بارها و بارها می‌تواند وارد و خارج شود.

انتظار محدود: این نیاز را ارضاء نمی‌کند.

فرایندهای ۱ و ۲ می‌توانند هم‌زمان flag خود را true کنند. سپس شرط while چک شود و هر دو داخل روند. هر دو flag خود را برابر false کنند و سپس برابر true کنند. دوباره در دام while می‌افتند و این قضیه تا ابد می‌تواند ادامه داشته باشد.

بخش ج)
انحصار متقابل: این نیاز را ارضاء می‌کند.
زمانی که هر کدام دو خط اول را طی کنند هیچ‌گاه حالتی نمی‌توان یافت که یکی while را رد کرده و سپس دیگری هم رد کند و اگر هم هم‌زمان یا یکی بعد از دیگری به while برسد با توجه به نامساوی تنها یکی می‌تواند رد شود.
حالت n1==0 هم برای حالت ابتدایی است و متأسفانه مشکلی ایجاد نمی‌کند.
پیشرفت: این نیاز را ارضاء نمی‌کند.
زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند فرایند دیگر می‌خواهد وارد C.S خود شود. فرض کنیم این فرایند P2 است. N2 در این حالت بیشتر از N1 می‌شود و به همین دلیل شرط while خاتمه نمی‌یابد و نمی‌تواند وارد شود.
انتظار محدود: این شرط را ارضاء می‌کند.
زمانی که هر فرایند بخواهد وارد C.S شود در ابتدا N خود را بیشتر می‌کند و همین باعث می‌شود طرف مقابل بتواند وارد شود و دوباره وقتی طرف مقابل برگشت نوبت این طرف می‌شود و ...

بخش د)
انحصار متقابل: این نیاز را ارضاء نمی‌کند.
هر دو فرایند می‌توانند while را هم‌زمان رد کنند (چون در ابتدا هر دو false هستند) و سپس flagها را برابر true کرده و وارد C.S شوند!
پیشرفت: این شرط را ارضاء می‌کند.
زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند فرایند دیگر می‌تواند بدون مشکل بارها وارد و خارج شود.
انتظار محدود: این شرط را ارضاء نمی‌کند.
فرایند P0 وارد C.S شده، فرایند P1 در داخل while گیر می‌کند، پردازنده به P0 داده می‌شود و شرط while را رد می‌کند (flag فرایند P1 برابر false است) و داخل C.S خود می‌رود. پردازنده به P1 داده می‌شود ولی دوباره در حلقه while می‌ماند. این حالت بارها و بارها می‌تواند رخ دهد و نوبت به P1 نرسد.

بخش ه)
انحصار متقابل: این نیاز را ارضاء نمی‌کند.
فرض کنیم دو فرایند P0 و P1 داریم. Turn در ابتدا ۱ است به همین دلیل فرایند اول وارد while شده و شرط (while(flag[1] را هم رد می‌کند و به قبل از turn=0 می‌رسد. در این حال فرایند دوم flag خود را برابر ۱ کرده و شرط while را هم رد می‌کند (فعلاً turn برابر همان مقدار اولیه ۱ است) و به C.S می‌رسد. در همین حین فرایند اول مقدار turn=0 می‌کند و از while بیرون می‌آید و وارد C.S خود می‌شود!
پیشرفت: این نیاز را ارضاء می‌کند.
زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند فرایند می‌تواند بارها و بارها وارد و خارج شود. Flag خود را true و در انتها false می‌کند و turn را هم به شماره خود تغییر می‌دهد.
انتظار محدود: این شرط را ارضاء نمی‌کند.
فرض کنیم فرایند P0 یک‌بار اجرا شده پس turn برابر ۰ است. دوباره وارد می‌شود و داخل C.S است فرایند P1 مقدار flag خود را برابر 1 کرده و شرط داخل while بیرونی می‌شود و در while درونی آن گیر می‌کند. در همین حین CPU به P0 داده می‌شود و از C.S خود خارج شده و چون turn برابر 0 است بدون گیر افتادن درون while بیرونی به C.S می‌رود. CPU دوباره به P1 داده می‌شود و شرط را چک کرده و باز هم درون while می‌ماند. این اتفاق بی‌نهایت می‌تواند اتفاق بیافتد.

بخش و)

شرط انحصار متقابل: این شرط را به درستی رعایت می‌کند.
از زمانی که هر فرایند بخواهد وارد ناحیه بحرانی خود شود flag خود را ۱ می‌کند که همین کار باعث ماندن فرایند دیگر در while شده و مانع ورود آن می‌شود.
شرط پیش‌روی: این شرط را به درستی رعایت می‌کند.
زمانی که فرایندی نه داخل ناحیه بحرانی است و نه منتظر ورود فرایند دیگر می‌تواند بارها وارد و خارج شود چون در این حالت شرط while اشتباه است و مانع دیگری هم در راه خود ندارد.
شرط انتظار محدود: این شرط را به درستی رعایت می‌کند.
فرض کنیم فرایند ۲ در انتظار ورود است و فرایند ۱ داخل C.S شده. در این حالت فرایند ۱ پس از خروج turn را برابر ۱ می‌کند و وقتی دوباره می‌خواهد وارد C.S خود شود وارد حلقه while شده و شرط if آن درست است و وارد if شده و داخل while با شرط turn == 1 بود گیر می‌کند به همین دلیل فرایند ۲ می‌تواند وارد C.S شود.
بدترین حالت زمانی است که فرایند ۲ در شرط turn == 0 گیر کرده و flag خود را عمداً ۰ کرده است که فرایند ۱ بعد از تمام شدن C.S خود turn را برابر ۱ می‌کند و دوباره وارد C.S خود می‌شود (چون فعلاً flag فرایند دیگر برابر ۰ است) ولی این موضوع زمانی که فرایند ۲ دارای CPU شده و از حلقه با شرط turn == 0 بیرون می‌آید و flag خود را برابر ۱ می‌کند تمام می‌شود و دیگر فرایند اول نمی‌تواند تا زمانی که فرایند دوم وارد C.S خود شده و turn را تغییر دهد وارد C.S شود.

بخش ز)
انحصار متقابل: این نیاز را ارضاء می‌کند.
زمانی که یک فرایند داخل C.S خود است به این معنی است که دو wait قبلی خود را رد کرده و مقدار سمافورهای A و B برابر ۰ می‌باشد به همین دلیل فرایند دیگر اجازه ورود نخواهد داشت و wait می‌شود.
پیشرفت: این نیاز را ارضاء می‌کند.
زمانی که هیچ فرایندی در C.S نباشند و در انتظار ورود به آن هم نباشند فرایند دیگر می‌تواند بارها و بارها وارد C.S خود شود و دوباره از آن خارج گردد.
انتظار محدود: این نیاز را ارضاء نمی‌کند.
فرایند P0 دستور wait(A) را رد کرده و فرایند دوم دستور wait(B) را رد کرده. هر دو قفل می‌کنند و هیچ‌کس به کس دیگری اجازه ورود نمی‌دهد و می‌مانند.

تمرین ۵)

**سوال اول:**

۱)

در این سیستم‌عامل PCB داخل فایل proc.h تعریف شده. PCB به صورت یک struct در C می‌باشد.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

۲)

**sz:**

unit یک تایپ از نوع unsigned int می‌باشد که به صورت زیر در فایل types.h تعریف شده:

typedef unsigned int uint;

خود sz هم نشان‌دهنده میزان حجم یک فرایند (بر اساس بایت) در حافظه می‌باشد.

**state:**

تایپ متغیر state به صورت enum می‌باشد. تعریف procstate به صورت زیر است:

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

پس یک فرایند می‌تواند stateای از موارد بالا داشته باشد.

**Context:**

خود متغیر context یک نوع struct می‌باشد که زمانی که CPU از یک فرایند پس گرفته می‌شود وضعیت فعلی فرایند در آن ذخیره می‌شود که دوباره وقتی cpu را به دست آورد لود شده و از حالت قبلی ادامه دهد.

struct context { uint edi; uint esi; uint ebx; uint ebp; uint eip; };

**ofile:**

این متغیر یک آرایه به طول NOFILE می‌باشد که با مقدار زیر تعریف شده است:

```
#define NOFILE 16         // open files per process
```

struct فایل به صورت زیر می‌باشد:

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

با توجه به دو قطعه کد زیر:

```
  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc→cwd);
```



```
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }
```

proc از struct درون ofile استفاده می‌کند.

**Killed:**

همان‌طور که در کامنت توضیح داده شده است اگر هر مقدار غیر 0 داشته باشد به معنی کشته شدن و اتمام فرایند است.

```
int killed;                 // If non-zero, have been killed
```

تمرین ۶)
در عملیات fork کل فرایند حال حاضر کپی شده و پس از آن زمانی که مموری و حالت برنامه در هر یک از فرایندها تغییر کند بر روی فرایند دیگر تأثیر نمی‌گذارد و هر یک memory image جدای خود را دارند.
در thread، مموری و منابع دیگر به صورت مشترک به وسیله thread های مختلف به صورت مشترک استفاده می‌شود و دیگر memory image های جدا برای چند فرایند نداریم.
Overhead در نخ نسبت به fork پایین‌تر می‌باشد.