۱۹ سمافور (Semaphore)

این تکنیک شبیه قفل mutex است اما روش پیچیده‌تری را برای همگام‌سازی فرآیندها استفاده می‌کند با این حال ابزار قوی‌تری را نیز معرفی می‌کند که در ادامه نشان داده خواهد شد.

سمافور یک ساختار (structure) شامل فیلدهای زیر دارد:

۱ شمارنده صحیح (count): از این شمارنده برای شمارش تعداد WakeUp هایی که ~~[illegible]~~ انجام می‌شود، استفاده می‌گردد. با این کار سیگنال‌های دریافتی برای استفاده‌های بعدی ذخیره می‌شود.

۲ صف (queue): از صف برای نگهداری فرآیندهای مسدود شده روی سمافور استفاده می‌شود.

```
typedef struct{
    int count;
    struct Process *list;
}semaphore;
```

همچنین دو تابع wait و signal را نیز شامل می‌شود که به ترتیب تعمیم یافته تابع sleep (برای بلاک کردن یک فرآیند) و WakeUp (برای اضافه کردن فرآیند به مجموعه فرآیندهای آماده سیستم) است.

در این تکنیک به جای اینکه فرآیندها برای ورود به ناحیه بحرانی خود مجبور باشند در حلقه انتظار مشغول (Busy waiting) گرفتار شوند، ~~[illegible]~~ به کمک استراتژی هوشمندانه و با استفاده از توابع و ساختار یک سمافور، مسدود شده و از انتظار مشغول کمی اتلاف پردازنده می‌شود را می‌گیرد.

۹

توابع Wait و Signal به صورت زیر تعریف می‌شوند:

```
Void wait (semaphore *S)
{
   S.count = S.count - 1;
   if (S.count < 0)
   {
      add this process to s.list;
      sleep();
   }
}

Void signal (semaphore *S)
{
   S.count = S.count + 1;
   if (S.count <= 0)
   {
      remove a process P from S.list;
      wakeup(P);
   }
}
```

عملیات sleep و wakup هر دو به عنوان فراخوان‌های سیستم (system call) پایه در سیستم عامل وجود دارند و به طور دقیق به شرح زیر هستند:

sleep() : فرآیندی که این تابع را فراخوانی کند، مسدود می‌شود.

wakeup(P) : اجرای فرآیند معلق شده P از سر گرفته می‌شود.

نکته: مقدار متغیر Count در یک سمافور می تواند منفی باشد و در واقع ارزش آن می تواند بیانگر تعداد فرآیندهایی باشد که منتظر ~~منتظر~~ روی سمافور هستند (به نحوی مقدار فرآیندهایی که در صف سمافور قرار گرفته اند).

نکته: در برخی منابع از حرف P به جای wait و از حرف V به جای signal استفاده شده است. همچنین در برخی کتب از اصطلاح down به جای wait و از UP به جای signal استفاده شده است.

نکته: در تابع signal، انتخاب فرآیند از صف فرآیندهای مسدود شده می بایست طبق استراتژی انجام شود. اگر انتخاب فرآیند از این صف براساس استراتژی FIFO (FCFS) صورت گیرد، به آن سمافور قوی اطلاق می شود و اگر این چنین نباشد به آن سمافور ضعیف می گویند. لازم به ذکر است که در سمافور ضعیف (متناسب با استراتژی انتخاب) امکان پدیده گرسنگی برای فرآیندهای مسدود شده سمافور وجود دارد.

نکته: عملیات signal و wait به عنوان عملیات اتمیک (غیرقابل تقسیم) در نظر گرفته می شوند.

(۲۰) می توان یک نسخه دودویی برای سمافور شمارشی در نظر گرفت که در آن متغیر Count تنها مقادیر ۰ و ۱ را اختیار می کند. توابع wait و signal برای سمافور باینری (دودویی) به صورت زیر است:

```
Void wait (semaphore *S)
{
  if (S.count == 1)
      S.count = 0;
  else
  {
    add this Process to S.list;
    sleep();
  }
}
```

```
Void Signal (semaphore *S)
{
  if (S.list is empty)
      S.count = 1;
  else
  {
     remove a process P from S.list;
     wakup(P);
  }
}
```

(۴۱) می‌توان یک شخص مشغول برانتظار (Busy waiting) را نیز برای تکنیک سمافور در نظر گرفت که البته در چنین شرایطی کاربرد صفی برای مسدود کردن فرآیندها نخواهیم داشت و تنها از متغیر count بهره می‌گیریم. توابع Wait و Signal برای چنین شرایطی به صورت زیر است. لازم به ذکر است که در این شرایط، تعریف کلاسیک سمافور نیز اطلاق می‌شود.

```
Void wait (semaphore *S)
{
   while (S.count <= 0) do; /* Busy wait */
   S.count = S.count - 1;
}

Void signal (semaphore *S)
{
   S.count = S.count + 1;
}
```

نکته ۵: متغیر count در سمافور کلاسیک تنها مقدار ۵ و مثبت ۵ ترند میگیرد در حالیکه در سطح عمیق سمافور، برای این متغیر امکان ارزش گذاری منفی نیز وجود دارد.

(۲۲) حل مسأله ناحیه بحرانی با استفاده از سمافورها

برای حل یک مسأله ناحیه بحرانی به کمک سمافور، می‌بایست پارامترهای زیر را تعیین کنیم:

۱. به چند سمافور نیاز داریم؟

۲. مقدار اولیه شمارنده هر سمافور چند باید باشد؟

۳. استراتژی انتخاب فرآیند از صف فرآیندهای مسدود شده روی سمافور چه باشد، که ما در ادامه صحبت مطالب فرض را بر روی استراتژی FCFS قرار می‌دهیم.

۴. عملیات‌های Signal و wait مربوط به هر سمافور در کجای برنامه باید قرار گیرد.

مشابه تکنیک‌های معرفی شده قبلی، فرض کنید دو فرآیند P_0 و P_1 داریم که همزمان صدا زده و هر کدام، یک ناحیه بحرانی دارند. فرآیندهای P_0 و P_1 در این تکنیک به صورت زیر باز تعریف می‌شوند:

```
Semaphore mutex;
mutex.count = 1;
```

```
P0(void)
{
  while (TRUE)
  {
    wait(mutex);
    critical_section();
    signal(mutex);
    non_critical_section();
  }
}
```

```
P1(void)
{
  while (TRUE)
  {
    wait(mutex);
    critical_section();
    signal(mutex);
    non_critical_section();
  }
}
```

فرض کنید فرآیند P_1 اول اجرا شود، در اینصورت فرآیند P_1 با اجرای تابع wait، چون مقدار سمافور $\neq 1$ است آن را صفر کرده و وارد ناحیه بحرانی می‌شود. در زمانیکه P_1 در ناحیه بحرانی است، اگر P_2 سعی به ورود به ناحیه بحرانی داشته باشد، چون سمافور برابر صفر است، این فرآیند مسدود شده و در صف قرار می‌گیرد. بعد از خروج P_1 از ناحیه بحرانی، تابع Signal را اجرا کرده و چون صف خالی نیست، P_2 که در صف قرار دارد را آزاد می‌کند و P_2 می‌تواند وارد ناحیه بحرانی شود.

تمرین: مشابه دهید سمافور باینری و عمومی برای فرآیندهای P_1 و P_2 ارایه شده در بالا، مساله ناحیه بحرانی را برای دو فرآیند مذکور حل می‌کند