جامعة المنارة

كلية: الهندسة

قسم: المعلوماتية

اسم المقرر: نظم تشغيل ٢

رقم الجلسة (٣)

عنوان الجلسة

استخدام اشارات السيمافور في نظم التشغيل

الفصل الدراسي

الأول

العام الدراسي

٢٠٢٣_٢٠٢٤

جدول المحتويات

Contents

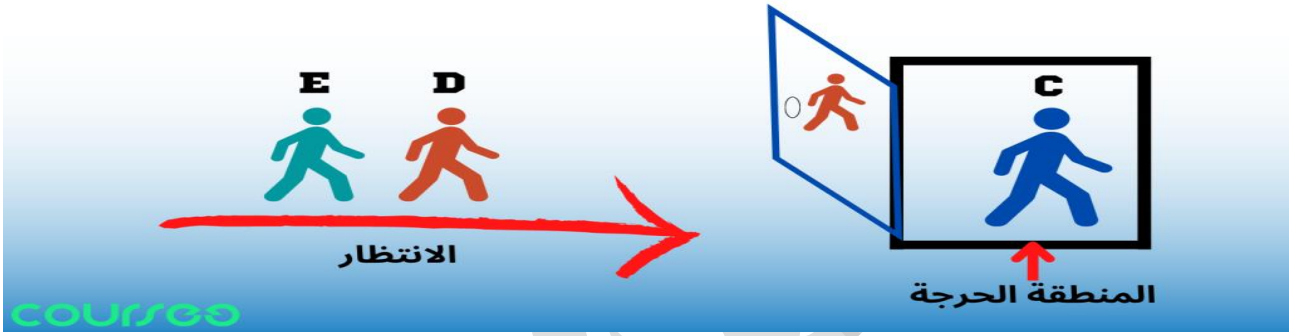
جامعة المنارة

الغاية من الجلسة:

تعريف الطلاب باستخدام اشارات السيما فور في نظم التشغيل من أجل التغلب على مشاكل العمليات المتزامنة في المنطقة الحرجة و مشاكل المنتج و المستهلك

مزامنة العمليات

يُطلق على مزامنة العمليات التنسيق بين عمليتين بحيث يكون هناك وصول إلى عناصر مثل الأجزاء المشتركة من الرموز أو الموارد أو البيانات وأشياء أخرى. على سبيل المثال ، قد يكون هناك مورد مشترك من خلال ٣ عمليات مختلفة ولا يمكن لأي من العمليات تغيير المورد المقابل في نفس الوقت ؛ لأن هذا قد يغير نتائج العمليات الأخرى التي تستخدم نفس المورد.



المنطقة الحرجة؟

في العمليات المتزامنة ، أي العمليات التي تعمل بشكل متزامن ، في حال كانت تحتاج جميعها إلى الوصول إلى جزء من التعليمات البرمجية. يسمى هذا القسم القسم الحرج أو.«Critical Section»

الاستبعاد المتبادل(Mutual Exclusion)

و هو يعني منع الوصول المتزامن إلى مورد مشترك

حالات العمليات

مراحل العملية في دورة حياتها. تكون بأحد الأوضاع التالية.

- تشغيل Running: العملية قيد التنفيذ.
- جاهز Ready: يشير إلى أن العملية تطلب التشغيل.
- خامل IDLE : يتم تشغيل العملية عند عدم وجود أي عمليات قيد التشغيل
- محظور Blocked: العمليات ليست جاهزة وليست مرشحة لعملية جارية. يمكن تنشيطه من خلال بعض الإجراءات الخارجية.
- غير نشط Inactive: الحالة الأولية للعملية غير نشط . يتم تنشيط العملية في مرحلة ما وتصبح جاهزة.

السيمافور Semaphore

هو في الأساس متغير صحيح أو Integer غير سالب يستخدم لحل مشكلة القسم الحرج

(Critical Section Problem) من خلال العمل كإشارة .(Signal)

سيمافور هو مفهوم في نظام التشغيل من أجل التزامن (Synchronization) خلال العمليات المتزامنة (Concurrent Process).

لنفترض أن لدينا عمليتين تعملان في نفس الوقت ، و لديهم متغيرًا باسم Shared وقيمته 5

أولاً ، نحدد هذا المتغير.

```
int shared = 5
```

نكتب الآن العملية الأولى على النحو التالي:

```
int x = shared;
// storing the value of shared variable in the variable x
x++;
sleep(1);
 shared = x;
```

العملية الثانية هي كما يلي:

```
int y = shared;
y--;
sleep(1);
shared = y;
```

لنبدأ بالعملية الأولى. في هذه العملية ، تم التصريح عن المتغير x ، والذي يحتوي في البداية على قيمة المتغير Shared، أي قيمة ٥. بعد ذلك ، تزداد قيمة x وتصبح قيمتها تساوي ٦ ، ثم تنتقل العملية إلى حالة السكون. نظرًا لأن العمليات الحالية متزامنة ، فلن تنتظر الـ CPU وتبدأ عملية المعالجة رقم ٢. المتغير الصحيح y له قيمة المتغير Shared، والتي تظل دون تغيير وقيمتها لا تزال تساوي ٥.

تنخفض قيمة y في العملية رقم ٢ ، وبعد ذلك يتم وضع هذه العملية في حالة السكون. الآن يعود نظام التشغيل إلى العملية رقم واحد وتصبح قيمة المتغير Shared مساوية لـ ٦. عند اكتمال هذه العملية ، في العملية رقم ٢ ، يتم تغيير قيمة المتغير Shared إلى ٤ والتي يجب أن تكون في الواقع ٥.

تسمى هذه الشروط حالة السباق (Race Condition) وبسبب هذه الشروط ، قد تحدث مشاكل مثل .Deadlock بالنظر إلى أن هناك حاجة إلى التزامن المناسب بين العمليات ولتجنب مثل هذه المشاكل ، يتم استخدام متغير إشارة عدد صحيح يسمى سيمافور.Semaphore

ما هو تعريف السيمافور؟

لتعريف السيمافور ، يمكننا القول أن السيمافور هو متغير عدد صحيح من نوع Integer يتم استخدامه بشكل متبادل وفريد من خلال العمليات المتزامنة لتحقيق التزامن المطلوب.

نظرًا لأن الإشارات هي متغيراتInteger ، فإن القيمة المخزنة فيها تعمل كإشارة ، وتصدر هذه الإشارة إذنًا أو لا تسمح للعملية بالوصول إلى قسم مهم من التعليمات البرمجية أو موارد محددة أخرى لعملية ما.

```
ClassSemaphore  {
   int sem;
   WaitQueue q;
}
Semaphore :: P()  {
```

```
    sem؛--
    if (sem< 0) {
        Add this thread t to q;
        block(P);
   }
}
Semaphore :: V() {
    sem؛++
    if (sem>=0) {
         (Remove a thread t from q;
        Wakeup(t)
   }
}
```

ما هي أنواع السيمافور؟

بشكل عام ، هناك نوعان من السيمافورات، أو في الواقع نوعان من المتغيرات Integer للإشارة ، والتي تم سردها أدناه ، ثم يتم وصف كل منها في أقسام فرعية منفصلة.

- السيمافورات الثنائية (Binary Semaphores)
- السيمافورات العددية (Counting Semaphores)

السيمافورات الثنائية

في هذا النوع من السيمافور ، يمكن أن تكون قيمة العدد الصحيح للسيمافور صفرًا أو واحدًا فقط. إذا كانت قيمة الإشارة ١، فهذا يعني أن العملية يمكن أن تدخل القسم الحرج. (المنطقة الحرجة هي المنطقة المشتركة التي يجب الوصول إليه)ا.

ولكن إذا كانت قيمة السيمافور هي صفر ، فلا يمكن للعملية أن تستمر وتدخل المنطقة الحرجة في الرموز. عندما تستخدم العملية منطقة الكود الحرجة ، فإننا نغير قيمة الإشارة إلى الصفر ، وعندما لا تستخدمها العملية ، يمكننا السماح للعملية بالوصول إلى المنطقة الحرجة (Critical Section) عن طريق تغيير قيمة السيمافور إلى ١

و تسمى أيضا.«Mutex Lock»

السيمافورات العددية

هي إشارات من نوع أعداد صحيحة يمكن أن تأخذ أي قيمة عددية، من الممكن تنسيق الوصول إلى الموارد ، وهنا يحدد عداد سيمافورات عدد الموارد المتاحة أو عدد العمليات المسموح بها لاستخدام الموارد. إذا كان قيمة السيمافور هي أي قيمة أكبر من الصفر ، فيمكن للعمليات الوصول إلى المناطق الحرجة أو الموارد المشتركة.

تحدد قيمة semaphore عدد العمليات التي يمكنها الوصول إلى الموارد أو التعليمات البرمجية. ولكن إذا كانت قيمة هذا النوع من السيمافور تساوي الصفر ، فهذا يعني أنه لا يوجد مورد متاح أو أن المنطقة الحرجة يتم الوصول إليها بالفعل من خلال عدد من العمليات ولا يمكن الوصول إليها من خلال المزيد من العمليات. تُستخدم السيمافورات العددية بشكل عام عندما يكون عدد عمليات إنشاء مثيل لمورد أكبر من واحد ويمكن لعمليات متعددة الوصول إلى هذا المورد.

مثال على السيمافور

الآن بعد أن عرفنا ما هي السيمافور وتعرّفنا على أنواع السيمافور ، من الضروري أن نفهم بشكل أفضل كيفية عمل السيمافور وأنواعه. كما ذكرنا سابقًا ، يتمثل هدفنا في مزامنة العمليات وتوفير الحصرية المتبادلة في المنطقة الحرجة من الرموز. لذلك ، نحتاج إلى توفير آلية باستخدام إشارة semaphore Integer لمنع أكثر من عملية واحدة من الوصول إلى المنطقة الحرجة.

```
shared variable semaphore = 1;

process i begin . . P(mutex);

execute Critical Section V(mutex);

. .

end;
```

هنا في هذه القطعة من الكود ، في السطر الأول ، يتم تعريف الإشارة التي يتم تهيئتها بالرقم الأول. ثم يبدأ تنفيذ العملية i ثم ، كما ترى ، تسمى الوظيفة P التي تتلقى قيمة كائن mutex أو السيمافور كمدخل، ثم ندخل المنطقة الحرجة ، وبعد ذلك يتم تنفيذ الوظيفة V ، وهي قيمة كائن المزامنة أو أنه يتلقى نفس الإشارة كمدخلات. بعد ذلك ، يتم تنفيذ الرموز المتبقية وتنتهي العملية.

يجب أن نتذكر أن السيمافور هو متغير إشارة وما إذا كانت العملية يمكن أن تدخل المنطقة الحرجة أم لا تعتمد على قيمتها. في الإشارات الثنائية والعددية ، نقوم بتغيير قيمة السيمافور اعتمادًا على الموارد المتاحة. بهذه الطريقة ، من الأفضل الآن الانتقال إلى وظائف P و V في الكود الكاذب أعلاه وشرحها أكثر.

تنفيذ سيمافورات الثنائية

مبدئيًا، قيمة السيمافور تساوي ١. عندما تدخل العملية P1 المنطقة الحرجة ، تتغير قيمة السيمافور إلى الصفر. إذا أرادت العملية P2 دخول المنطقة الحرجة في هذا الوقت ، فلن يكون ذلك ممكنًا ، لأن قيمة السيمافور ليست أكبر من الصفر.

يجب أن تنتظر العملية P2 حتى تصبح قيمة السيمافور أكبر من ١، ولن يحدث هذا الحدث إلا عندما يغادر P1 المنطقة الحرجة ويتم تنفيذ عملية الإشارة ، مما يؤدي إلى زيادة قيمة السيمافور. لذلك ، يتم تحقيق التفرد المتبادل بهذه الطريقة باستخدام سيمافور ثنائي، بمعنى آخر ، لا يمكن لكلتا العمليتين الوصول إلى المنطقة الحرجة في نفس الوقت.

تنفيذ سيمافورات العددية

في سيمافورات العددية، الاختلاف الوحيد هو أنه سيكون لدينا عدد من الموارد ، أو بعبارة أخرى ، هناك عدد معين من العمليات التي يمكنها الوصول إلى المنطقة الحرجة في وقت واحد وفي نفس اللحظة.

لنفترض أن لدينا موردًا يحتوي على ٤ مثيلات(Instance) ، وبالتالي فإن القيمة الأولية للسيمافور ستكون semaphore = 4. كلما احتاجت عملية ما إلى الوصول إلى منطقة أو مورد حرج ، فإننا نجلب دالة أو وظيفة الانتظار وننقص قيمة السيمافور بواحد فقط إذا كانت قيمة السيمافور أكبر من الصفر.

عندما تصل ٤ عمليات إلى المنطقة الحرجة أو المورد الحرج وتحتاجه العملية الخامسة ، فإننا نضعها في قائمة الانتظار ونسميها (تنبيه) فقط عندما تقوم العملية بتنفيذ الوظيفة أو دالة الإشارة وهذا يعني أن زادت قيمة السيمافور بوحدة واحدة. فيما يلي الرموز المتعلقة بتنفيذ السيمافور العددي:

حل مشكلة المنتج - المستهلك باستخدام السيمافورات

الآن بعد أن أصبح لدينا فهم لكيفية عمل السيمافورات، يمكننا إلقاء نظرة على تطبيقات الحياة الواقعية للسيمافورات في مشاكل المزامنة الكلاسيكية. مشكلة المنتج - المستهلك هي إحدى المشاكل الكلاسيكية لمزامنة العملية.

بيان مشكلة المنتج والمستهلك

في حالة مشكلة المنتج - المستهلك ، هناك مخزنًا مؤقتًا بحجم ثابت وسيقوم المنتج بإنتاج العناصر ووضعها في المخزن المؤقت. يمكن للمستهلك أخذ العناصر من المخزن المؤقت واستهلاكها و هنا يجب الانتباه إلا أن لا يتم تداخل بين عمل المنتج و المستهلك ، المخزن هنا يعبر عن المنطقة الحرجة.

حل مشكلة المنتج - المستهلك باستخدام السيمافورات

لحل هذه المشكلة ، تم استخدام سيمافورين للعد باسم full و empty. ستراقب سيمافور العددي full وتتبع جميع الفتحات المشغولة في المخزن المؤقت ، أو بعبارة أخرى ، ستتتبع جميع العناصر الموجودة في المخزن المؤقت. أيضًا، ستراقب سيمافور empty جميع العناصر الفارغة في المخزن المؤقت وستكون قيمة كائن المزامنة (mutex) مساوية لـ ١.

مبدئيًا ، قيمة سيمافور full تساوي صفرًا ، لأن جميع الفتحات الموجودة في المخزن المؤقت غير مشغولة ، وقيمة المخزن المؤقت empty تساوي n ، حيث n هي حجم المخزن المؤقت ، لأن جميع الفتحات هي في البداية فارغة.

Full = 0

EMPTY=5

على سبيل المثال ، إذا كان حجم المخزن المؤقت يساوي ٥ ، فستكون0 = semaphore full ، لأن جميع الفتحات الموجودة في المخزن المؤقت غير مشغولة و empty تساوي ٥. الحل المستنتج لجزء المنتج من المشكلة هو كما يلي:

في الكود أعلاه ، نسمي عملية الانتظار (Wait Operation) على السيمافورات empty و mutex كائن المزامنة (mutex)عندما ينتج المنتج عنصرًا. نظرًا لتكوين عنصر ، يجب وضعه في المخزن المؤقت وتقليل عدد الفتحات الفارغة بمقدار واحد ، لذلك نسمي عملية الانتظار على السيمافور الفارغة. يجب علينا أيضًا تقليل قيمة كائن المزامنة (mutex)لمنع المستخدم من الوصول إلى المخزن المؤقت.

بعد ذلك ، وضع المنتج العنصر المقابل في المخزن المؤقت ، وبالتالي يمكننا زيادة قيمة سيمافور الـ full بوحدة واحدة وأيضًا زيادة قيمة كائن المزامنة (mutex) ، لأن المنتج قام بعمله ، والآن تمكّن الإشارة هذا سوف تضطر إلى الوصول إلى المخزن المؤقت.

يحتاج المستهلك إلى استهلاك العناصر التي ينتجها المنتج. لذلك ، عندما يزيل المستهلك العنصر من المخزن المؤقت لاستهلاكه ، نحتاج إلى إنقاص قيمة سيمافور full بواحد ، لأنه سيتم تحرير الفتحة ، ونحتاج أيضًا إلى إنقاص قيمة كائن المزامنة (mutex) حتى يتمكن المنتج يمكن الوصول إلى المخزن المؤقت.

الآن بعد أن استهلك المستهلك العنصر ، يمكن تقليل كمية الإشارة الفارغة ويمكن تقليل كمية الإشارة الفارغة بوحدة واحدة. وبهذه الطريقة ، تمكنا من حل مشكلة المنتج - المستهلك وانتهى العمل.

ما هي مزايا السيمافور؟

كما رأينا حتى الآن في هذه المقالة ، تتمتع الإشارات بالكثير من الكفاءة والاستفادة في مجال عمليات المزامنة. في هذا القسم ، نصف بإيجاز مزايا السيمافورات.

- تسمح السيمافورات للعمليات بالدخول إلى المنطقة الحرجة واحدة تلو الأخرى ، كما توفر (في حالة السيمافورات الثنائية) حصرية متبادلة بطريقة دقيقة ومستقرة للغاية.
- لا يتم إهدار أي موارد بسبب الازدحام والانتظار الطويل ، لأنه باستخدام السيمافورات، لا يضيع وقت المعالج في التحقق من الظروف للسماح للعملية بالوصول إلى منطقة حرجة.
- تتم كتابة أكواد السيمافور وتنفيذها في نفس كود المنطقة المستقلة عن الآلة في النواة الدقيقة ، وبالتالي ، فإن السيمافورات مستقلة عن الآلة.

بعد وصف مزايا السيمافورات، من الأفضل مناقشة عيوبها.

ما هي عيوب السيمافور؟

ناقشنا حتى الآن مزايا السيمافورات، لكن للسيمافورات أيضًا عيوبًا ذكرناها أدناه.

- السيمافورات معقدة بعض الشيء ، ونحن بحاجة إلى تنفيذ عمليات الانتظار والإشارة بطريقة تتجنب المآزق.
- قد يؤدي استخدام السيمافورات إلى انعكاس الأولوية (Priority Inversion). في انعكاس الأولوية، تصل العمليات ذات الأولوية العالية إلى المنطقة الحرجة بعد العمليات ذات الأولوية المنخفضة.

تعد مشكلة المنتج والمستهلك مثالاً على مشكلة المزامنة متعددة العمليات. تصف المشكلة عمليتين، المنتج والمستهلك الذي يشترك في مخزن مؤقت مشترك ذو حجم ثابت يستخدمه كقائمة انتظار.

ومهمة المنتج هي توليد البيانات ووضعها في المخزن المؤقت والبدء من جديد.

وفي الوقت نفسه، يستهلك المستهلك البيانات (أي يزيلها من المخزن المؤقت)، قطعة واحدة في كل مرة.

المشكلة: بالنظر إلى المخزن المؤقت الشائع ذو الحجم الثابت، تتمثل المهمة في التأكد من أن المنتج لا يمكنه إضافة بيانات إلى المخزن المؤقت عندما يكون ممتلئًا ولا يستطيع المستهلك إزالة البيانات من المخزن المؤقت الفارغ.

الحل: يتعين على المنتج إما الانتقال إلى وضع السكون أو تجاهل البيانات إذا كان المخزن المؤقت ممتلئًا. في المرة التالية التي يقوم فيها المستهلك بإزالة عنصر من المخزن المؤقت، فإنه يقوم بإخطار المنتج، الذي يبدأ في ملء المخزن المؤقت مرة أخرى. وبنفس الطريقة، يمكن للمستهلك أن ينام إذا وجد أن المخزن المؤقت فارغ. وفي المرة التالية التي يضع فيها المنتج البيانات في المخزن المؤقت، فإنه يوقظ المستهلك النائم.

ملاحظة: قد يؤدي الحل غير المناسب إلى طريق مسدود حيث تنتظر كلتا العمليتين أن يتم إيقاظهما.

المنهج: الفكرة هي استخدام مفهوم البرمجة الموازية

```
// C program for the above approach
#include <stdio.h>
#include <stdlib.h>
// Initialize a mutex to 1
int mutex = 1;
// Number of full slots as 0
int full = 0;
// Number of empty slots as size
// of buffer
int empty = 5, x = 0;
// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;
    // Increase the number of full
    // slots by 1
    ++full;
    // Decrease the number of empty
    // slots by 1
    --empty;
    // Item produced
    x++;
```

```
        printf("\nProducer produces"
                "item %d",
                x);
        // Increase mutex value by 1
        ++mutex;
    }
    // Function to consume an item and
    // remove it from buffer
    void consumer()
    {
        // Decrease mutex value by 1
        --mutex;
        // Decrease the number of full
        // slots by 1
        --full;
        // Increase the number of empty
        // slots by 1
        ++empty;
        printf("\nConsumer consumes "
                "item %d",
                x);
        x--;

        // Increase mutex value by 1
        ++mutex;
    }
    // Driver Code
    int main()
    {
        int n, i;
```


```
        printf("\n1. Press 1 for Producer"
               "\n2. Press 2 for Consumer"
               "\n3. Press 3 for Exit");
    // Using '#pragma omp parallel for'
    // can give wrong value due to
    // synchronization issues.
    // 'critical' specifies that code is
    // executed by only one thread at a
    // time i.e., only one thread enters
    // the critical section at a given time
    #pragma omp critical
        for (i = 1; i > 0; i++) {
                printf("\nEnter your choice:");
                scanf("%d", &n);
                // Switch Cases
                switch (n) {
                case 1:
                        // If mutex is 1 and empty
                        // is non-zero, then it is
                        // possible to produce
                        if ((mutex == 1)
                                && (empty != 0)) {
                                producer();
                        }
                        // Otherwise, print buffer
                        // is full
                        else {
                                printf("Buffer is full!");
                        }
                        break;
```

```
            case 2:
                // If mutex is 1 and full
                // is non-zero, then it is
                // possible to consume
                if ((mutex == 1)
                    && (full != 0)) {
                    consumer();
                }
                // Otherwise, print Buffer
                // is empty
                else {
                    printf("Buffer is empty!");
                }
                break;
            // Exit Condition
            case 3:
                exit(0);
                break;
            }
        }
    }
```

جامعة المنارة