جامعة المنارة

كلية: .....الهندسة.....

قسم: ..... الهندسة المعلوماتية.....

اسم المقرر: ..... نظم تشغيل 2.....

رقم الجلسة (...3...)

عنوان الجلسة

....السيمافور في نظم التشغيل.....

م.عمار مصطفى م.فاطمة جنيدي

الفصل الدراسي العام الدراسي 2024/ 2025

جدول المحتويات

Contents

الغاية من الجلسة: تعريف الطلاب باستخدام اشارات السيمافور في نظم التشغيل من أجل التغلب على مشاكل العمليات المتزامنة في المنطقة الحرجة ومشكلة المنتج و المستهلك، مشكلة القارئ و الكاتب

مزامنة العمليات وإدارة المنطقة الحرجة هما مكونان أساسيان في تصميم أنظمة التشغيل والتطبيقات التي تتعامل مع تعدد العمليات (Processes) أو الخيوط.(Threads) .

1-مزامنة العمليات

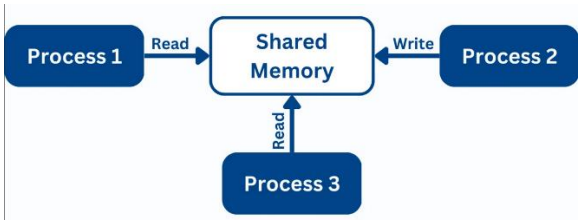
مزامنة العمليات هي العملية التي يتم من خلالها تنسيق العمليات المختلفة لضمان أن تعمل معًا بدون تعارض، خاصة عندما تشترك في موارد النظام.

الآليات المستخدمة لمزامنة العمليات:

1. السيمافور :(Semaphore) أداة قوية لإدارة التزامن.
2. الأساليب القائمة على القفل :(Lock-based mechanisms) مثل الأقفال (Locks) والمتغيرات الشرطية (Condition Variables).
3. المراقب :(Monitor) يوفر مراقبة ذاتية للتزامن ويحتوي على قفل داخلي ومتغيرات شرطية.
4. الحواجز :(Barriers) تستخدم لتنظيم نقطة التزامن حيث تتوقف كل العمليات وتنتظر أن يصل الجميع إلى نقطة محددة قبل المتابعة

2-المنطقة الحرجة(Critical Section)

المنطقة الحرجة هي جزء من الكود الذي يمكن أن يؤدي إلى مشاكل إذا تم تشغيله بواسطة أكثر من عملية أو خيط في نفس الوقت. تحتاج المنطقة الحرجة إلى حماية لضمان أن عملية واحدة فقط تقوم بتنفيذها في وقت واحد.



القواعد لضمان سلامة المنطقة الحرجة:

1. المنع المتبادل :(Mutual Exclusion) يجب ضمان أن عملية واحدة فقط تستطيع الدخول إلى المنطقة الحرجة في أي وقت.
2. التقدم :(Progress) إذا لم تكن أي عملية في المنطقة الحرجة، وأرادت عملية الدخول إليها، فيجب أن تستطيع الدخول بدون تأخير غير مبرر.
3. الانتظار المحدود :(Bounded Waiting) يجب وضع حد على عدد المرات التي يمكن لعملية أخرى الدخول إلى المنطقة الحرجة قبل أن يتم منح الدخول للعملية التي تنتظر.

3-حالة السباق Race Condition:

تحدث حالة سباق عندما تقوم عمليات أو خيوط متعددة بالوصول إلى الموارد المشتركة وتعديلها في وقت واحد، وتعتمد النتيجة النهائية على الترتيب غير المتوقع للتنفيذ.

الموارد المشتركة: هذه هي الموارد مثل الملفات أو المتغيرات أو مكونات الأجهزة التي يمكن الوصول إليها من خلال عمليات/خيوط متعددة .الوصول المتزامن: تحاول عمليات أو خيوط متعددة تعديل المورد المشترك في وقت واحد .الترتيب غير المتوقع: التسلسل الدقيق الذي يتم به تنفيذ هذه العمليات/الخيوط غير حتمي، مما يعني أنه يمكن أن يختلف في كل مرة .نتيجة غير صحيحة: بسبب الترتيب غير المتوقع للتنفيذ، قد تكون الحالة النهائية للمورد المشترك غير متسقة أو غير صحيحة.

مثال: لنفترض وجود نظام مصرفي بسيط تحاول فيه عمليتان سحب الأموال من نفس الحساب :العملية **P1** :قراءة رصيد الحساب (لنقل 100 دولار) .خصم المبلغ المراد سحبه (لنقل 50 دولارًا) .كتابة الرصيد الجديد (50 دولارًا) مرة أخرى في الحساب .العملية **P2** :قراءة رصيد الحساب (100 دولار) .خصم المبلغ المراد سحبه (50 دولارًا) .كتابة الرصيد الجديد (50 دولارًا) مرة أخرى في الحساب .**حالة السباق** :إذا تم تشغيل العمليتين **P1** و**p2** في نفس الوقت، فإن السيناريو التالي ممكن : تقرأ العملية**P1** الرصيد (100 دولار) .تقرأ العملية **P2** الرصيد (100 دولار) .تكتب العملية **P2** الرصيد الجديد (50 دولارًا) . تكتب العملية **P1** الرصيد الجديد (50 دولارًا) .النتيجة: سيكون الرصيد النهائي في الحساب 50 دولارًا، بينما يجب أن يكون 0 دولارًا (نظرًا لأن العمليتين سحبتا 50 دولارًا) .يرجع ذلك إلى أن ترتيب التنفيذ غير متوقع، مما يؤدي إلى نتيجة غير صحيحة.

3-1-حلول لمنع حالةالسباق:

آليات المزامنة: استخدام آليات مثل أدوات المزامنة المتبادلة mutexes أو السيمافور semaphores أو أجهزة المراقبة لضمان أن عملية/خيط واحد فقط يمكنه الوصول إلى المورد المشترك في كل مرة.

3-1-1-السيمافور.(Semaphore)

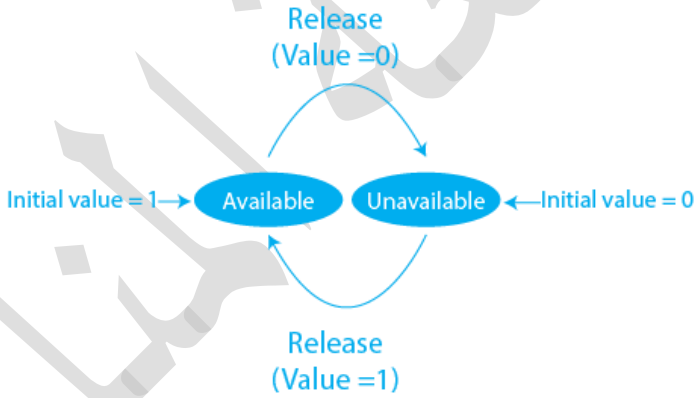
Semaphore (السيميافور) هو مفهوم أساسي في علوم الكمبيوتر. يُعتبر السيما فور واحداً من آليات التزامن الأساسية في نظم التشغيل.

1-تعريف السيميافور

السيما فور هو متغير أو كائن يُستخدم للتحكم في الوصول إلى مورد مشترك من قبل عمليات متعددة في نظام تشغيل متعدد المهام. يمكن استخدامه لمنع الشروط المتسارعة والتأكد من أن العمليات لا تتداخل مع بعضها البعض.

2- أنواع السيمافور

- **السيما فور الثنائي (Binary Semaphore)**: يمكن أن يأخذ فقط القيمتين 0 و1، وهو يُستخدم للتحكم في الوصول إلى مورد يمكن استخدامه مرة واحدة فقط في الوقت نفسه (مثل القفل).
  - ✓ التهيئة: يتم تهيئة السيمافور الثنائي بقيمة 1، مما يشير إلى أن المورد المشترك متاح.



- **السيمافور العددي (Counting Semaphore)**: هي نوع من **السيما فور الثنائي** التي تسمح لعدد محدود من العمليات أو الخيوط بالوصول إلى مورد مشترك في وقت واحد. إنها شكل أكثر عمومية من **السيما فور الثنائي** ، والتي تسمح فقط لعملية واحدة أو خيط واحد بالوصول إلى المورد في وقت واحد.
  - ✓ التهيئة: يتم تهيئة السيما فور العددي بقيمة، تُعرف باسم count، والتي تحدد الحد الأقصى لعدد العمليات أو الخيوط التي يمكنها الوصول إلى المورد المشترك في وقت واحد.
  - ✓ الاستحواذ: عندما تريد عملية أو خيط الوصول إلى المورد المشترك، تحاول الاستحواذ على السيمافور. إذا كانت قيمة السيمافور أكبر من 0، يمكن للعملية/الخيط الوصول إلى المورد ويتم تقليل قيمة السيمافور بمقدار 1. إذا كانت قيمة السيمافور 0، يتم حظر العملية/الخيط حتى يتم تحرير السيمافور.
  - ✓ التحرير: عندما تنتهي عملية/سلسلة من استخدام المورد المشترك، فإنها تقوم بتحرير السيمافور، مما يزيد قيمة السيمافور بمقدار 1. وهذا يشير إلى العمليات/الخيوط الأخرى أن المورد أصبح متاحًا مرة أخرى.

3-آلية عمل السيمافور:

- **إشارة :(Signal)** تُستخدم لزيادة قيمة السيما فور. عندما يتم تنفيذ عملية الإشارة، فهذا يشير إلى أن عملية قد انتهت من استخدام المورد أو أنه تم تحرير المورد.
- **انتظار :(Wait)** تُستخدم لإنقاص قيمة السيما فور. عندما يتم تنفيذ عملية الانتظار، فهذا يعني أن العملية تحاول الوصول إلى المورد، وإذا كانت القيمة صفرًا، فستنتظر العملية حتى يصبح المورد متاحًا.

مثال بسيط باستخدام لغة C :

هذا الكود ينشئ ثلاثة خيوط، وكل خيط ينتظر حتى يصبح السيمافور متاحًا، ثم يدخل القسم الحرج، يقوم بطباعة رسالة، وأخيرًا يحرر السيمافور. يتم استخدام السيمافور هنا لضمان أن خيطًا واحدًا فقط يمكنه الدخول إلى القسم الحرج في نفس الوقت.

```c
#include <stdio.h>
 2  #include <pthread.h>
 3  #include <semaphore.h>
 4
 5  sem_t semaphore;
 6
 7  void* thread_function(void* arg) {
 8      sem_wait(&semaphore); // الانتظار للحصول على المورد
 9      // القسم الحرج
10      printf("Thread %d is in the critical section.\n", *(int*)arg);
11      sem_post(&semaphore); // تحرير المورد
12      return NULL;
13  }
14
15  int main() {
16      pthread_t threads[3];
17      int thread_ids[3] = {1, 2, 3};
18
19      sem_init(&semaphore, 0, 1); // 1 تهيئة السيمي فور بقيمة
20
21      for (int I = 0; I < 3; i++) {
22          pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
23      }
24
25      for (int I = 0; I < 3; i++) {
26          pthread_join(threads[i], NULL);
27      }
28
29      sem_destroy(&semaphore); // تدمير السيمي فور
30      return 0;
31  }
```

stdio.h مكتبة الإدخال والإخراج القياسية في C.

semaphore.h مكتبة لمعالجة السيما فور.(semaphores)

تعريف متغير من نوع sem_t يسمى semaphore، وهو السيما فور الذي سنستخدمه لمزامنة الوصول إلى القسم الحرج.

sem_wait(&semaphore) تنتظر حتى يصبح السيما فور متاحًا (قيمة أكبر من الصفر)، ثم تقلل من قيمته بمقدار واحد.

printf("Thread %d is in the critical section.\n", *(int*)arg) تطبع رسالة تشير إلى أن الخيط قد دخل القسم الحرج.

sem_post(&semaphore) تزيد قيمة السيميافور بمقدار واحد، مما يسمح للخيوط الأخرى بالدخول إلى القسم الحرج

pthread_t threads[3] تعريف مصفوفة من ثلاث خيوط.

int thread_ids[3] = {1, 2, 3} تعريف مصفوفة من ثلاث معرفات للخيوط.

sem_init(&semaphore, 0, 1) تهيئة السيما فور بقيمة ابتدائية 1. القيمة 0 تشير إلى أن السيما فور مشترك بين العمليات في نفس الذاكرة (داخل نفس العملية).

pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]) إنشاء ثلاثة خيوط، كل خيط يستدعي thread_function مع تمرير معرّف الخيط كمعامل.

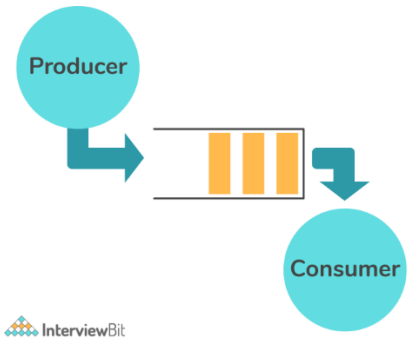
pthread_join(threads[i], NULL) انتظار انتهاء كل خيط قبل مواصلة تنفيذ البرنامج الرئيسي

sem_destroy(&semaphore) تدمير السيمي فور لتحرير الموارد.

return 0; إنهاء البرنامج بنجاح.

مشكلة المنتج والمستهلك Producer-Consumer

تعتبر مشكلة المنتج والمستهلك مشكلة مزامنة كلاسيكية في علوم الكمبيوتر، وخاصة في سياق أنظمة التشغيل والبرمجة المتزامنة. وهي تنطوي على نوعين من العمليات: المنتجون والمستهلكون، الذين يشتركون في مخزن مؤقت مشترك. ويتمثل التحدي في ضمان عدم قيام المنتج بإضافة عناصر إلى مخزن مؤقت ممتلئ وعدم قيام المستهلك بإزالة عناصر من مخزن مؤقت فارغ.



المفاهيم الأساسية

- المنتج: عملية تقوم بتوليد البيانات ووضعها في المخزن المؤقت.
- المستهلك: عملية تأخذ البيانات من المخزن المؤقت وتعالجها.
- المخزن المؤقت: منطقة تخزين ذات حجم محدود يضع فيها المنتج العناصر ويستعيد المستهلك العناصر منها.

المتطلبات

- المنع المتبادل Mutual Exclusion: يجب أن تتمكن عملية واحدة فقط (إما المنتجة أو المستهلكة) من الوصول إلى المخزن المؤقت في كل مرة لمنع عدم تناسق البيانات.
- سعة المخزن المؤقت Buffer Capacity: يجب أن ينتظر المنتج إذا كان المخزن المؤقت ممتلئًا، ويجب أن ينتظر المستهلك إذا كان المخزن المؤقت فارغًا.

نهج الحل

يمكن حل المشكلة باستخدام آليات المزامنة مثل السيمافور أو المزامنة المتبادلة. فيما يلي حل نموذجي باستخدام السيمافور:

السيمافور:

- فارغة: تحسب عدد الفتحات الفارغة في المخزن المؤقت.
- ممتلئة: تحسب عدد الفتحات الممتلئة في المخزن المؤقت.
- مزامنة: تضمن المنع المتبادل عند الوصول إلى المخزن المؤقت.

مثال على الكود بلغة C++

فيما يلي تنفيذ بسيط لمشكلة المنتج والمستهلك باستخدام لغة C++:

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <semaphore>
5  #include <vector>
6  #include <chrono>

8  const int BUFFER_SIZE = 5;
9   std::vector<int> buffer(BUFFER_SIZE);
10  std::counting_semaphore<BUFFER_SIZE> empty_slots;
11  std::counting_semaphore<0> full_slots;
12  std::mutex buffer_mutex;
13
14  void producer() {
15      for (int i = 0; i < 10; ++i) {
16          empty_slots.acquire(); // Wait for an empty slot
17          {
18              std::lock_guard<std::mutex> lock(buffer_mutex);
19              buffer[i % BUFFER_SIZE] = i; // Produce an item
20              std::cout << "Produced: " << i << std::endl;
21          }
22          full_slots.release(); // Signal that a new item is available
23          std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Simulate
production time
24      }
25  }
26
27  void consumer() {
28      for (int i = 0; i < 10; ++i) {
29          full_slots.acquire(); // Wait for a full slot
30          {
31              std::lock_guard<std::mutex> lock(buffer_mutex);
32              int item = buffer[i % BUFFER_SIZE]; // Consume an item
33              std::cout << "Consumed: " << item << std::endl;
34          }
35          empty_slots.release(); // Signal that an item has been consumed
```

```
36          std::this_thread::sleep_for(std::chrono::milliseconds(150)); // Simulate
consumption time
37      }
38  }
39
40  int main() {
41      std::thread prod(producer);
42      std::thread cons(consumer);
43      prod.join();
44      cons.join();
45      return 0;
46
```

الرؤوس والثوابت

- يتضمن الكود الرؤوس اللازمة للخيوط، و الاستبعاد المتبادل ، والسيمافور، والمتجهات.
- يحدد الثابت BUFFER_SIZE حجم المخزن المؤقت المشترك.
- المتغيرات المشتركة للمخزن المؤقت والمزامنة
- يعد متجه المخزن المؤقت vector<int> buffer المخزن المؤقت المشترك الذي يحتفظ بالعناصر المنتجة.
- يتم استخدام السيمافور empty_slots لحساب عدد الفتحات الفارغة في المخزن المؤقت. يتم تهيئتها باستخدام BUFFER_SIZE للإشارة إلى أن المخزن المؤقت فارغ في البداية.
- يتم استخدام السيمافور full_slots لحساب عدد الفتحات الكاملة في المخزن المؤقت. يتم تهيئتها باستخدام 0 للإشارة إلى أن المخزن المؤقت فارغ في البداية.
- يتم استخدام السيمافور buffer_mutex لحماية الوصول إلى المخزن المؤقت المشترك.

دالة المنتج void producer()

- يتم تشغيل دالة المنتج في خيط منفصل وتنتج 10 عناصر.
- تنتظر فتحة فارغة في المخزن المؤقت باستخدام empty_slots.acquire(). **عملية الانتظار**: تقوم الدالة acquire() بإنقاص قيمة السيمافور بمقدار واحد. **السلوك**: إذا كانت قيمة السيمافور أكبر من الصفر، ستستمر الخيط في التنفيذ. إذا كانت القيمة صفرًا، ستنتظر الخيط حتى تصبح قيمة السيمافور أكبر من الصفر (أي أن هناك موارد متاحة).
- بمجرد توفر فتحة فارغة، يقوم بقفل buffer_mutex للوصول إلى المخزن المؤقت.
- ينتج عنصرًا عن طريق تعيين قيمة للمخزن المؤقت ويطبع رسالة تشير إلى أنه تم إنتاج عنصر.
- يحرر السيمافور full_slots للإشارة إلى أن عنصرًا جديدًا متاحًا في المخزن المؤقت.
- يحاكي وقت الإنتاج باستخدام std::this_thread::sleep_for(). وضع الخيط في حالة انتظار لفترة زمنية قصيرة (100 ملي ثانية) قبل إنتاج العنصر التالي. هذا يساعد على تقليل استخدام المعالج ويمنح الوقت لتفريغ المخزن من قبل المستهلكين.

دالة المستهلك `void consumer()`

- تعمل دالة المستهلك في خيط منفصل وتستهلك 10 عناصر.
- تنتظر فتحة كاملة في المخزن المؤقت باستخدام full_slots.acquire().
- بمجرد توفر فتحة كاملة، يقوم بقفل buffer_mutex للوصول إلى المخزن المؤقت.
- يستهلك عنصرًا عن طريق قراءة قيمة من المخزن المؤقت ويطبع رسالة تشير إلى أنه تم استهلاك عنصر.
- يحرر السيمافور empty_slots للإشارة إلى أنه تم استهلاك عنصر وأن الفتحة فارغة الآن.
- يحاكي وقت الاستهلاك باستخدام std::this_thread::sleep_for().

الوظيفة الرئيسية `main()`

- تنشئ الوظيفة الرئيسية خيطين، أحدهما للمنتج والآخر للمستهلك.
- تبدأ الخيطين باستخدام كائنات std::thread.
- تنتظر حتى ينتهي الخيطان باستخدام join().

مشكلة القارئ-الكاتب (Readers-Writers Problem) :

هي مشكلة كلاسيكية في التزامن (Synchronization) تصف سيناريو حيث تحتاج عمليات متعددة إلى الوصول إلى مورد مشترك (مثلاً ملف). بعض هذه العمليات تريد فقط قراءة المورد (Readers) ، بينما تريد عمليات أخرى كتابة فيه (Writers)

القواعد الأساسية هي:

- يمكن لعدة قراء الوصول إلى المورد في نفس الوقت.
- يمكن لكاتب واحد فقط الوصول إلى المورد في أي وقت.
- لا يمكن للقراء والكتاب الوصول إلى المورد في نفس الوقت.

يمكن حل هذه المشكلة باستخدام السيمافور (Semaphores) كما يلي:

المتغيرات:

- mutex سيمافور ثنائي (Binary Semaphore) مهيأ بالقيمة 1، يستخدم لحماية readcount.
- rw_mutex سيمافور مهيأ بالقيمة 1، يستخدم لمنع الكتاب والقراء من الوصول المتزامن للمورد.
- readcount متغير عددي يحفظ عدد القراء الذين يقرأون حاليًا من المورد

```
#include <iostream>    // لإظهار المخرجات على الشاشة
#include <pthread.h>   // لإنشاء وإدارة الخيوط (Threads)
#include <semaphore.h> // لاستخدام السيمافورات (Semaphores)
#include <unistd.h>    // لاستخدام الدالة sleep()
```

تعريف السيمافورات والمتغيرات المشتركة:

```
sem_t rw_mutex;      // Semaphore (قرّاء وكتّاب) يتحكم في الوصول للكتابة. لحماية قسم الكتابة.
sem_t mutex;         // Semaphore لحماية المتغير read_count من التداخل بين الخيوط.

int read_count = 0;  // عدد القراء الحاليين الذين يقرؤون في نفس الوقت.
```

دالة القارئ:

```
void* reader(void* arg) {
    int id = *((int*)arg);  // استخراج رقم القارئ من المؤشر
```

الدخول إلى قسم حرج لحماية read_count

```
 sem_wait(&mutex);  // إغلاق (lock) السيمافور لحماية read_count من التعديل المتوازي
 read_count++;      // زيادة عدد القراء الحاليين

 if (read_count == 1)
     sem_wait(&rw_mutex);  // أول قارئ يغلق باب الكتابة

 sem_post(&mutex);  // فتح السيمافور
```

القسم الخاص بالقراءة:

```
cout << "📖 القارئ " << id << " يقرأ البيانات..." << endl;
sleep(1); // تمثيل وقت القراءة
```

الخروج من القراءة:

```
sem_wait(&mutex);  // الدخول مرة أخرى لحماية read_count
read_count--;      // تقليل عدد القراء

if (read_count == 0)
    sem_post(&rw_mutex);  // آخر قارئ يفتح باب الكتابة للكتّاب

sem_post(&mutex);  // فتح السيمافور
```

إنهاء الخيط:

```
    pthread_exit(0); // إنهاء الخيط
}
```

دالة الكاتب:

```
// الدالة التي ينفذها الكاتب
void* writer(void* arg) {
    int id = *((int*)arg);

    sem_wait(&rw_mutex); // كاتب ينتظر حتى لا يوجد قراء أو كتاب آخرين

    // (قسم الكتابة) كتابة البيانات
    cout << "✍️ الكاتب " << id << " يكتب البيانات..." << endl;
    sleep(2); // تمثيل وقت الكتابة

    sem_post(&rw_mutex); // السماح للآخرين بالدخول

    pthread_exit(0);
}
```

```
int main() {
    pthread_t rtid[5], wtid[5];  // معرفات الخيوط
    int ids[5];

    // تهيئة السيمافورات
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);

    for (int i = 0; i < 5; i++) {
        ids[i] = i + 1;
        pthread_create(&rtid[i], NULL, reader, &ids[i]);
        pthread_create(&wtid[i], NULL, writer, &ids[i]);
    }

    // الانتظار لانتهاء الخيوط
    for (int i = 0; i < 5; i++) {
        pthread_join(rtid[i], NULL);
        pthread_join(wtid[i], NULL);
    }

    // تدمير السيمافورات
    sem_destroy(&rw_mutex);
    sem_destroy(&mutex);

    return 0;
}
```

عيوب هذا الحل:

قد يتسبب في تجويع الكتاب (Writer Starvation) إذا كان هناك تدفق مستمر من القراء. بمعنى أن الكتاب قد يضطرون للانتظار لفترة طويلة قبل أن يتمكنوا من الكتابة.

هذا الحل يمثل أحد أبسط حلول مشكلة القراء-الكتاب باستخدام السيمافور. توجد حلول أخرى أكثر تعقيدًا تعالج مشكلة تجويع الكتاب .