جامعة المنارة

كلية: .....الهندسة.....

قسم: ..... الهندسة المعلوماتية.....

اسم المقرر: ..... نظم تشغيل 2.....

رقم الجلسة (...4...)

عنوان الجلسة

....المراقب في نظم التشغيل.....

إعداد: م.عمار مصطفى

الفصل الدراسي العام الدراسي 2024/ 2025

جدول المحتويات

Contents

الغاية من الجلسة: التعرف على بنية المراقب وكيفية استخدامه كأدراة مزامنة بين العمليات للوصول الى الموارد المشتركة كما سيتم اجراء مقارنة مع تقنية السيمافور وفق مشكلة عشاء الفلاسفة

المراقب Monitor هي بنية مزامنة تستخدم في أنظمة التشغيل لإدارة الوصول إلى الموارد المشتركة من خلال عمليات أو خيوط متعددة. وهي تساعد في منع حالات السباق، مما يضمن أن خيطًا واحدًا فقط يمكنه الوصول إلى مورد في كل مرة مع توفير تجريد أعلى مستوى من المزامنة المتبادلة أو السيمافور. قد تكون البيانات داخل Monitor إما عالمية لجميع الاجرائيات داخل Monitor أو محلية لإجراء معين. أي طريقة ضمن المراقب لا يمكنها الوصول إلى أي بيانات خارج المراقب.

يجمع المراقب بين ثلاث ميزات:

- البيانات المشتركة.
- العمليات على البيانات.
- المزامنة والجدولة.

وهي ملائمة بشكل خاص للمزامنة التي تتضمن الكثير من الحالات.

المفاهيم الأساسية للمر اقب:

- التغليف Encapsulation: يغلف المراقب المتغيرات المشتركة والإجراءات التي يعمل عليها، مما يضمن أنه لا يمكن تغيير الحالة المشتركة إلا بطريقة خاضعة للرقابة. لا يمكن الوصول إلى هذه المتغيرات إلا من خلال الإجراءات المحددة داخل المراقب.
- الإجراءات Procedure: تحدد المراقبين الأساليب (أو الإجراءات) التي تعمل على المتغيرات المشتركة. هذه الإجراءات هي الوسيلة الوحيدة التي يمكن للخيوط من خلالها التفاعل مع البيانات المشتركة.
- الاستبعاد المتبادل Mutual Exclusion: لا يمكن أن يكون سوى خيط واحد نشطًا في المراقب في أي وقت. يتم تحقيق ذلك من خلال استخدام الأقفال locks.
- المتغيرات الشرطية Condition Variables: غالبًا ما يتضمن المراقب متغيرات شرطية تسمح للخيوط بالانتظار حتى تصبح شروط معينة صحيحة قبل المتابعة. وهذا مفيد في السيناريوهات التي يحتاج فيها الخيط إلى الانتظار حتى يصبح المورد متاحًا.

يمكن إجراء عمليتين على المتغيرات الشرطية wait و signal

Wait(condition): تحرير قفل المراقب، ووضع العملية في وضع السكون. وعندما تستيقظ العملية مرة أخرى، تكتسب قفل المراقب على الفور.

Signal(condition): ايقاظ عملية واحدة قيد الانتظار على المتغير الشرطي (FIFO)، إذا لم يكن هناك أي عملية قيد الانتظار، فلا تفعل شيئًا

Broadcast(condition): ايقاظ جميع العمليات التي تنتظر على المتغير الشرطي. إذا لم يكن هناك أي عملية قيد الأنتظار ينتظر، فلا تفعل شيئًا.

- القفل التلقائي Automatic Locking: عندما يدخل الخيط إلى المراقب، فإنه يكتسب قفل المراقب، وعندما يخرج، يتم تحرير القفل تلقائيًا.

بنية المراقب Monitor Structure:

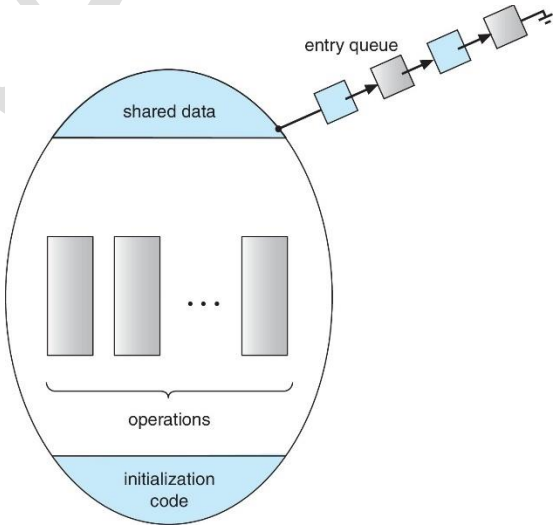
قد تتضمن بنية المراقب النموذجية ما يلي:

- المتغيرات المشتركة: وهي الموارد المشتركة بين الخيوط.
- الإجراءات: الوظائف التي تعمل على المتغيرات المشتركة.
- المتغيرات الشرطية: تستخدم للإشارة بين الخيوط.

يمكن الوصول إلى بيانات Monitor فقط داخل Monitor. يمكن حماية بنية البيانات المشتركة عن طريق وضعها في Monitor. إذا كانت البيانات الموجودة في Monitor تمثل بعض الموارد، فإن Monitor يوفر مرفق استبعاد متبادل للوصول إلى المورد. يمكن للإجراء المحدد داخل المراقب الوصول فقط إلى تلك المتغيرات المعلنة محليًا داخل المراقب ومعامِلاتها الرسمية.

عندما تستدعي عملية إجراء مراقب، فإن التعليمات القليلة الأولى للإجراء ستتحقق لمعرفة ما إذا كانت هناك أي عملية أخرى نشطة حاليًا داخل المراقب. إذا كانت العملية نشطة، فسيتم تعليق العملية المستدعاة حتى تغادر العملية الأخرى المراقب. إذا لم تكن هناك عملية أخرى تستخدم المراقب، فيمكن للعملية المستدعاة الدخول.

الشكل التالي يُظهر مخططًا للمراقب، مع قائمة إدخال للعمليات التي تنتظر دورها لتنفيذ عمليات المراقب (الطرق methods).

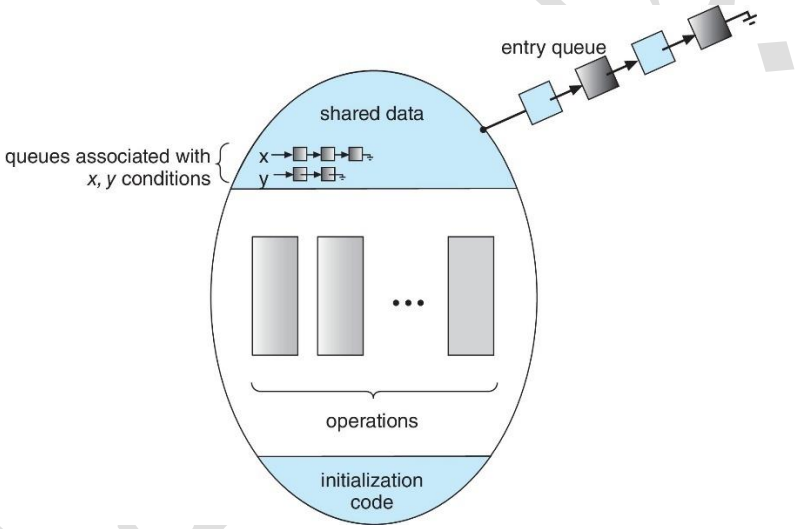


يدعم Monitor المزامنة باستخدام المتغيرات الشرطية الموجودة داخل Monitor والتي يمكن الوصول إليها فقط من داخل Monitor. كل متغير شرطي له قائمة انتظار مرتبطة به. تعمل المتغيرات الشرطية على اثنين، من أجل تحقيق إمكانات المراقبين بالكامل، نحتاج إلى تقديم نوع

بيانات جديد إضافي، يُعرف باسم الشرط condition. يحتوي المتغير من النوع الشرطي على عمليتين قانونيتين فقط، الانتظار wait والإشارة signal . أي إذا تم تعريف X من النوع الشرطي، فإن العمليات القانونية ستكون X.wait() و X.signal()

- تمنع عملية الانتظار العملية حتى تستدعي عمليات أخرى الإشارة signal، وتضيف العملية المحظورة إلى قائمة مرتبطة بهذا الشرط.
- لا تفعل عملية الإشارة signal أي شيء إذا لم تكن هناك عمليات تنتظر هذا الشرط. وإلا فإنها توقظ عملية واحدة فقط من قائمة العمليات المنتظرة للشرط.

يوضح الشكل التالي أدناه مراقبًا يتضمن متغيرات شرط داخل مساحة البيانات الخاصة به. لاحظ أن متغيرات الشرط، إلى جانب قائمة العمليات التي تنتظر حاليًا الشروط، موجودة في مساحة البيانات الخاصة بالمراقب - والعمليات الموجودة في هذه القوائم ليست "داخل" المراقبة، بمعنى أنها لا تنفذ أي تعليمات برمجية في المراقبة.



ولكن الآن هناك مشكلة محتملة - إذا أصدرت العملية P داخل المراقب إشارة signal من شأنها إيقاظ العملية Q أيضًا داخل المراقب، فسيكون هناك عمليتان تعملان في وقت واحد داخل المراقب، مما ينتهك متطلب الاستبعاد. وفقًا لذلك، هناك حلين محتملين لهذه المعضلة:

- الإشارة signal والانتظار wait - عندما تصدر العملية P الإشارة لإيقاظ العملية Q، تنتظر P بعد ذلك، إما حتى تخرج Q من المراقب أو في حالة أخرى.
- الإشارة signal والاستمرار continue - عندما تصدر P الإشارة، تنتظر Q، إما حتى تخرج P من المراقب أو في حالة أخرى.

هناك بارامترات لصالح وضد أي من الخيارين. يقدم باسكال المتزامن بديلاً ثالثًا - يتسبب استدعاء الإشارة signal في خروج العملية على الفور من المراقب، بحيث يمكن لعملية الانتظار بعد ذلك الاستيقاظ والمتابعة.

مثال: استخدام المتغير الشرطي :

لنفترض أن P1 وP2 يحتاجان إلى تنفيذ الحالتين S1 وS2 والمطلب حدوث S1 قبل S2، أنشئ مراقبًا بإجراءين F1 وF2 يتم استدعاؤهما بواسطة P1 وP2 على التوالي

- متغير شرطي واحد "x" تم تهيئته إلى 0
- متغير منطقي واحد "done"

```
1  F1:
2     S1;
3     done = true;
4     x.signal();
5  F2:
6     if done = false
7        x.wait()
8     S2;
```

- ✓ S1: هذا عنصر نائب لعملية يقوم بها الخيط (threat 1). يمكن أن تكون هذه أي مهمة، مثل إنتاج البيانات أو معالجة عنصر، إلخ.
- ✓ done = true: يضبط هذا السطر علامة (done) على true، مما يشير إلى أن العملية التي يمثلها S1 قد اكتملت. تعمل هذه العلامة كشرط للخيط الآخر (threat 2) لمعرفة ما إذا كان يمكنه المتابعة.
- ✓ x.signal: يشير هذا التابع إلى المتغير الشرطي x. إنها توقظ أحد الخيوط التي قد تكون في انتظار متغير الشرط هذا، مما يسمح لها باستئناف التنفيذ.
- ✓ إذا done = false: يتحقق هذا الشرط من حالة علامة done. إذا كانت false، فسينفذ الخيط (threat 2) طريقة wait.
- ✓ x.wait: يضع هذا التابع الخيط في حالة انتظار، ويحرر القفل المرتبط بالمتغير الشرطي x. سيظل الخيط في حالة الانتظار هذه حتى يتم الإشارة إليه بواسطة خيط آخر (في هذه الحالة، threat 1) مما يشير إلى أنه يمكنه المتابعة (عندما يصبح done = true).
- ✓ S2: بمجرد استيقاظ الخيط بنجاح (بمعنى أن علامة done تم تعيينها على true)، فسوف ينفذ العملية التي يمثلها S2، مثل استهلاك البيانات.

منطق المزامنة

تفاعلات الخيوط threats:

- ينفذ الخيط 1 عمليته، ويضبط علامة "done" على "true"، ويرسل إشارة إلى الخيط 2 للاستيقاظ.
- يتحقق الخيط 2 من حالة "done" قبل المتابعة. إذا كانت حالة "done" تساوي false ، فإنه ينتظر الإشارة من الخيط 1.

استخدام متغيرات الشرط:

- يساعد هذا النمط في منع الانتظار المشغول busy waiting، حيث يتحقق الخيط threat باستمرار من شرط ما. بدلاً من ذلك، سوف ينام threat 2 ولا يستيقظ إلا عند إخطاره، مما يجعل استخدام موارد وحدة المعالجة المركزية أكثر كفاءة.

فيما يلي مثال بسيط لمراقب يتحكم في الوصول إلى عداد مشترك: يحدد هذا الكود مراقب يسمىCounter ، والذي ينفذ عدادًا آمنًا للخيوط

```
1  monitor Counter {
2      int count = 0; // Shared variable
3      condition not_empty;
4
5      procedure increment() {
6          count = count + 1;
7      }
8
9      procedure decrement() {
10         while (count == 0) {
11             wait(not_empty); // Wait until count is not zero
12         }
13         count = count - 1;
14     }
15
16     procedure getCount() {
17         return count;
18     }
19 }
20
```

شرح المثال

المتغير المشترك: يتم مشاركة متغير count بين جميع الخيوط التي تستخدم مراقب العداد.

الإجراءات:

- increment: يزيد من العدد.
- decrement: يقلل من العدد ولكنه ينتظر إذا كان العدد صفرًا.
- getCount: يعيد القيمة الحالية للعدد.
- المتغير الشرطي: يسمح المتغير الشرطي not_empty لإجراء التناقص بالانتظار حتى يكون هناك عدد موجب.
- while (count == 0) { ... } تضمن هذه الحلقة عدم استمرار عملية التناقص إذا كان العداد صفرًا بالفعل. وهذا يمنع الأخطاء المحتملة مثل القيم السلبية.
- wait(not_empty); إذا كان العداد صفرًا، ينتظر الخيط على متغير الشرط .not_empty وهذا يحظر الخيط حتى يرسل خيط آخر إشارة signal إلى متغير الشرط.

عندما تستدعي خيوط متعددة increment() أو decrement() في نفس الوقت، يمكن لخيط واحد فقط الوصول إلى متغير count المشترك في كل مرة بسبب خاصية الاستبعاد المتبادل للمراقب.

- إذا استدعى خيط ما decrement() ووجد أن العداد يساوي صفرًا، فسوف ينتظر على متغير الشرط.not_empty
- عندما يستدعي خيط آخر increment() ، فإنه يزيد العداد ثم يشير إلى متغير الشرط.not_empty

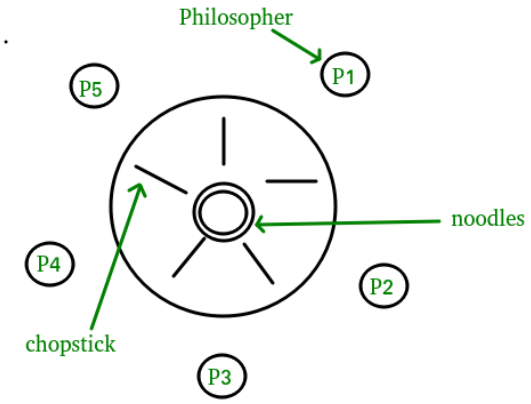
تعمل هذه الإشارة على إيقاظ الخيط المنتظر من حالة الانتظار، مما يسمح له بمواصلة عملية.decrement()

مزايا المراقب

- البساطة: توفر تجريدًا أعلى مستوى من بدائيات المزامنة ذات المستوى الأدنى، مما يجعل الكود أسهل في الفهم.
- المعالجة التلقائية للأقفال: تتم معالجة قفل وفتح الموارد المشتركة تلقائيًا عند الدخول إلى المراقب والخروج منه.

مشكلة عشاء الفلاسفة باستخدام المراقب:

لدينا n فيلسوف يجلس حول طاولة دائرية، يوجد عود طعام واحد بين كل فيلسوف، يجب على الفيلسوف أن يلتقط عودين طعام أقرب إليه ليأكل، يجب على الفيلسوف أن يلتقط عود طعام أولاً، ثم الثاني، وليس الاثنين في وقت واحد



مشكلة الفلاسفة في تناول الطعام هي مشكلة مزامنة كلاسيكية توضح تحديات تقاسم الموارد والجمود في البرمجة المتزامنة. تتضمن المشكلة خمسة فلاسفة يجلسون على طاولة، يحتاج كل منهم إلى شوكتين لتناول الطعام. والهدف هو تصميم حل يسمح للفلاسفة بتناول الطعام دون التسبب في جمود أو مجاعة.

الحل باستخدام المراقب

في هذا الحل، سنستخدم المراقب لتغليف سلوك كل فيلسوف والموارد المشتركة (الشوك Forks). سينتظر كل فيلسوف الشوك (الممثلة بمتغيرات الشرط) ويرسل إشارة Signal عند الانتهاء من تناول الطعام.

```
1  monitor DiningPhilosophers {
2      boolean[] fork_available = {true, true, true, true, true}; // Fork availability
3      condition[] fork = new condition[5]; // Condition variables for each fork
4
5      procedure pickup(int philosopher) {
6          // Wait until both forks are available
7          while (!fork_available[philosopher] || !fork_available[(philosopher + 1) % 5]) {
8              wait(fork[philosopher]); // Wait for the condition signal
9          }
10         // Take the forks
11         fork_available[philosopher] = false; // Forks are now taken
12         fork_available[(philosopher + 1) % 5] = false;
13     }
14
15     procedure putdown(int philosopher) {
16         // Put down the forks
17         fork_available[philosopher] = true; // Forks are now available
18         fork_available[(philosopher + 1) % 5] = true;
```

```
19          signal(fork[philosopher]); // Signal waiting philosophers
20          signal(fork[(philosopher + 1) % 5]); // Signal waiting philosophers
21      }
```

سلوك الفيلسوف

سيستخدم كل فيلسوف إجراءات الالتقاط pickup والإنزال putdown المحددة في المراقب، يتناوب كل فيلسوف بين التفكير والأكل. تضمن إجراءات pickup وputdown احترام قواعد مشاركة الموارد، مما يمنع الجمود والجوع.

- توافر الشوكةFork Availability : تتعقب مجموعة fork_available ما إذا كانت كل شوكة متاحة أم لا. يحتاج كل فيلسوف إلى التحقق من توافر شوكاته اليسرى واليمنى.
- متغيرات الشرطية Condition Variables: لكل فيلسوف متغير شرطي مطابق في مجموعة الشوك. ينتظر الفلاسفة هذه المتغيرات الشرطية عندما لا يتمكنون من التقاط الشوكتين.

إجراء الالتقاط pickup:

- قبل التقاط الشوكتين، يتحقق الفيلسوف مما إذا كانت الشوكتان اليسرى واليمنى متاحتين. إذا لم يكن الأمر كذلك، ينتظر الفيلسوف متغير الشرطي الخاص بهما.
- إذا كانت الشوكتان متاحتين، يضبط الفيلسوف الإدخالات المقابلة في fork_available على false، مما يشير إلى أن الشوكتين مأخوذتان الآن.

إجراء الإنزال putdown

عندما ينتهي الفيلسوف من الأكل، يضع الشوكتين. يتم تحديث توافر الشوكتين، ويتم إرسال إشارات إلى المتغيرات الشرطية الحالة لإيقاظ أي فيلسوف منتظر.

Monitor DiningPhilosophers: يتم هنا إنشاء مراقب يسمى DiningPhilosophers.

boolean[] fork_available = {true, true, true, true, true}: تمثل هذه المصفوفة مدى توفر الشوكات الخمس. في البداية، تكون جميع الشوكات متاحة (true).

condition[] fork = new condition[5]: تحتوي هذه المصفوفة على متغيرات شرطية، متغير واحد لكل شوكة. تُستخدم متغيرات الشرط للإشارة والانتظار بطريقة متزامنة.

procedure pickup(int philosopher): تحاكي هذه العملية فيلسوفًا يلتقط عيدان تناول الطعام الخاصة به (الشوك).

while (!fork_available[philosopher] || !fork_available[(philosopher + 1) % 5]): تتحقق هذه الحلقة مما إذا كانت كل من الشوكة اليسرى للفيلسوف (philosopher) والشوكة اليمنى (philosopher + 1) % 5 والتي تلتف حول 0 إذا كان الفيلسوف هو 4) متاحة.

wait(fork[philosopher]): إذا كانت أي من الشوكتين غير متاحة، ينتظر الفيلسوف متغير الشرط المرتبط بالشوكة اليسرى. يؤدي هذا إلى تحرير القفل على المراقب، مما يسمح للفلاسفة الآخرين بالاستيلاء على الشوكات المحتملة.

fork_available[philosopher] = false; fork_available[(philosopher + 1) % 5] = false;: بمجرد توفر الشوكين، يلتقطهما الفيلسوف ويضع علامة عليهما على أنهما غير متاحين.

procedure putdown(int philosopher): يحاكي هذا الإجراء الفيلسوف وهو يضع عيدان تناول الطعام الخاصة به.

fork_available[philosopher] = true; fork_available[(philosopher + 1) % 5] = true;: يضع الفيلسوف علامة على الشوكين على أنهما متاحان مرة أخرى.

signal(fork[philosopher]); signal(fork[(philosopher + 1) % 5]);: يرسل الفيلسوف إشارات إلى متغيرات الشرط المرتبطة بكلا الشوكين. يؤدي هذا إلى إيقاظ أي فيلسوف ينتظر تلك الشوكين.

متى سيحدث الجمود deadlock ؟

يحدث الجمود deadlock عندما يتم حظر عمليتين أو أكثر، في انتظار أن تطلق كل منهما الأخرى موردًا تحتاجه. يمكن أن يحدث هذا في الحلول التي تسمح للفيلسوف بالتقاط عيدان تناول الطعام واحدة في كل مرة، أو التي تسمح لجميع الفلاسفة بالتقاط عيدان تناول الطعام اليسرى في نفس الوقت.

متى ستحدث المجاعة؟

تحدث المجاعة عندما يتم حظر عملية إلى أجل غير مسمى من الوصول إلى مورد تحتاجه. يمكن أن يحدث هذا في الحلول التي تعطي الأولوية لفلاسفة معينين أو التي تسمح لبعض الفلاسفة بالحصول على عيدان تناول الطعام باستمرار بينما لا يتمكن آخرون من ذلك.

كيف يمنع الجمود

- الاستبعاد المتبادل: يضمن المراقب أن يتمكن فيلسوف واحد فقط من الاحتفاظ بالقفل في كل مرة، مما يمنع الوصول المتزامن إلى الشوكات المشتركة.
- متغيرات الشرط: يسمح استخدام متغيرات الشرط للفلاسفة بالانتظار بأمان إذا لم تكن الشوكات متاحة. عندما تصبح الشوكات متاحة، يتم إخطار الفيلسوف المنتظر.

<u>الخلاصة</u>

يدير هذا التنفيذ لمشكلة Dining Philosophers باستخدام المراقب المزامنة بين الفلاسفة بشكل فعال. من خلال تغليف توفر الشوكة والتحكم في الوصول داخل المراقب، نضمن أن الفلاسفة لا يمكنهم المضي قدمًا إلا عندما يكون لديهم كلتا الشوكتين، وبالتالي منع الجمود وضمان العدالة في مشاركة الموارد. يسلط هذا النهج الضوء على مزايا استخدام بنيات المزامنة عالية المستوى في البرمجة المتزامنة. يستخدم هذا التنفيذ قاعدة "اليد اليسرى"، حيث يلتقط الفيلسوف دائمًا الشوكة اليسرى أولاً. يتجنب هذا السيناريو حيث يحاول فيلسوفان التقاط نفس الشوكات في نفس الوقت. يضمن المراقب العدالة في الوصول إلى الشوكات باستخدام متغيرات الشرط لإخطار الفلاسفة المنتظرين.

الحل القائم على السيمافور:

يمكن أيضًا حل مشكلة الفلاسفة أثناء تناول الطعام باستخدام السيمافور. في هذا النهج، سنستخدم السيمافور الثنائي لتمثيل توفر الشوكات. سيحاول كل فيلسوف الحصول على الشوكات (السيمافور) اللازمة قبل تناول الطعام وسيقوم بإطلاقها بعد الانتهاء.

```
sem_t fork[5]; // Binary semaphores for each fork

void initialize() {
    for (int i = 0; i < 5; i++) {
        sem_init(&fork[i], 0, 1); // Initialize each fork semaphore to 1 (available)
    }
}

void pickup(int philosopher) {
    // Pick up the left fork
    sem_wait(&fork[philosopher]); // Wait (decrement) for left fork
    // Pick up the right fork
    sem_wait(&fork[(philosopher + 1) % 5]); // Wait (decrement) for right fork
}

void putdown(int philosopher) {
    // Put down the right fork
    sem_post(&fork[(philosopher + 1) % 5]); // Signal (increment) right fork
    // Put down the left fork
    sem_post(&fork[philosopher]); // Signal (increment) left fork
}

void philosopher(int id) {
    while (true) {
        think(); // Philosophers think
        pickup(id); // Try to pick up forks
        eat(); // Philosophers eat
        putdown(id); // Put down forks
    }
```

يستخدم هذا الكود السيمافور لحل مشكلة Dining Philosophers. وفيما يلي تفصيل:

sem_t fork[5]: يتم إعلان مجموعة من 5 سيمافور، تمثل الشوكات الخمسة. يتم تهيئة كل سيمافور إلى 1، مما يشير إلى أن الشوكة متاحة.

void initialize(): تقوم هذه الوظيفة بتهيئة كل سيمافور باستخدام sem_init(&fork[i], 0, 1).

تتطلب sem_init ثلاثة بارامترات:

- &fork[i]عنوان السيمافور المراد تهيئته.
- 0: يشير إلى سيمافور مشتركة(يمكن الوصول إليها من قبل عمليات متعددة).
- 1: القيمة الأولية للسيمافور (1 تعني متوفرة).

void pickup(int philosopher): تحاكي هذه العملية فيلسوفًا يلتقط كلا الشوكتين:

sem_wait(&fork[philosopher]) و sem_wait(&fork[(philosopher + 1) % 5]): تنتظر هذه الأسطر (تتناقص) على السيمافور التي تمثل الشوكتين اليسرى واليمنى. إذا كانت السيمافور 0 (غير متوفرة)، فإن الفيلسوف يحظر حتى تصبح قيمة السيمافور 1 (متوفرة).

void putdown(int philosopher): تحاكي هذه العملية فيلسوفًا يضع كلا الشوكتين:

sem_post(&fork[philosopher]) و sem_post(&fork[(philosopher + 1) % 5]): تشير هذه الأسطر (تزيد) من قيمة السيمافور المرتبطة بالشوكتين اليمنى واليسرى. وهذا يجعل الشوكتين متاحتين للفلاسفة الآخرين.

void philosopher(int id): تمثل هذه الدالة تصرفات كل فيلسوف:

think(): عنصر نائب للفيلسوف الذي يفكر.

pickup(id): يحاول التقاط الشوكتين.

eat(): عنصر نائب للفيلسوف الذي يأكل.

putdown(id): يضع الشوكتين.

كيف يمنع الجمود deadlock

- السيمافور الثنائي: تمثل كل سيمافور شوكة واحدة، مما يسمح لفيلسوف واحد فقط بحمل القفل في كل مرة.
- الانتظار wait: تضمن عملية sem_wait أنه في حالة عدم توفر شوكة، يقوم الفيلسوف بحظرها، مما يمنع الجمود.
- الإشارة signal: تطلق عملية sem_post الشوكة عندما ينتهي الفيلسوف، مما يسمح لفيلسوف آخر بالحصول عليها.

النقاط الرئيسية

- يستخدم هذا التنفيذ قاعدة "اليد اليسرى" لالتقاط الشوكات، على غرار حل المراقب.
- يعمل استخدام السيمافور على تبسيط منطق المزامنة مقارنة بالمراقب، حيث توفر آليات الانتظار والإشارة بشكل مباشر.
- يعد نهج السيمافور هذا طريقة شائعة وفعالة لحل مشكلة Dining Philosophers. وهو يوضح كيف يمكن استخدام السيمافور للمزامنة في البرمجة المتزامنة.
- الحل القائم على السيمافور يوفر تحكمًا أكبر في المزامنة ولكن على حساب زيادة التعقيد. يجب على المبرمج إدارة السيمافور بشكل صريح والتعامل مع حالات الجمود المحتملة من خلال ضمان الترتيب المناسب عند الحصول على الموارد

مشكلة القراءة والكتابة باستخدام المراقب

مشكلة القراءة والكتابة هي مشكلة مزامنة كلاسيكية تتضمن إدارة الوصول إلى مورد مشترك حيث يمكن لعدة خيوط القراءة في وقت واحد، ولكن يمكن لخيط واحد فقط الكتابة في كل مرة. التحدي هو ضمان عدم تدخل القراء مع الكتاب والعكس صحيح، مع السماح أيضًا لعدة قراء بالقراءة في وقت واحد.

الحل القائم على المراقب:

في هذا الحل، سننفذ مشكلة القراءة والكتابة باستخدام المراقب لإدارة المزامنة بين القراء والكتاب.

```
monitor ReadWrite {
    int readers = 0; // Count of active readers
    boolean writerActive = false; // Flag to indicate if a writer is active
    condition canRead; // Condition variable for readers
    condition canWrite; // Condition variable for writers

    procedure startRead() {
        // Wait until there are no active writers
        while (writerActive) {
            wait(canRead);
        }
        readers++; // Increment the count of active readers
    }

    procedure endRead() {
        readers--; // Decrement the count of active readers
        if (readers == 0) {
            signal(canWrite); // Signal waiting writers if no readers are left
        }
    }

    procedure startWrite() {
        // Wait until there are no active readers or writers
        while (writerActive || readers > 0) {
            wait(canWrite);
        }
        writerActive = true; // Set the writer as active
    }

    procedure endWrite() {
        writerActive = false; // Writer has finished
        signal(canWrite); // Signal waiting writers
        signal(canRead); // Signal waiting readers
    }

```

شرح الحل

عدد القراء **Reader Count**: يتتبع عدد القراء الذين يقومون حاليًا بالوصول إلى المورد المشترك.

علم الكاتب النشط **Writer Active Flag**: يشير القيمة المنطقية writerActive إلى ما إذا كان الكاتب يكتب حاليًا إلى المورد المشترك.

متغيرات الشرط:

canRead: يستخدم للإشارة إلى متى يمكن للقارئ البدء في القراءة (أي عندما لا يكون هناك كتاب نشطون).

canWrite: يستخدم للإشارة إلى متى يمكن للكاتب البدء في الكتابة (أي عندما لا يكون هناك قراء أو كتاب نشطون).

إجراء بدء القراءة Read Procedure:

- يتحقق القارئ من وجود كاتب نشط. إذا كان موجودًا، فإنه ينتظر شرط canRead حتى يتم الإشارة إليه.
- بمجرد منح الوصول، يزيد القارئ عدد القراء النشطين.

إجراء نهاية القراءة End Read Procedure:

يقلل القارئ عدد القراء النشطين. إذا كان هو آخر قارئ (أي، يصل القراء إلى 0)، فإنه يشير إلى canWrite للسماح للكتاب المنتظرين بالمتابعة.

إجراء بدء الكتابة Write Procedure:

- ينتظر الكاتب إذا كان هناك أي قراء نشطين أو إذا كان هناك كاتب آخر نشط حاليًا. يضمن هذا الوصول الحصري.
- بمجرد منح الوصول، يضبط الكاتب علامة writerActive على true.

إنهاء إجراء الكتابة End Write Procedure:

بعد الانتهاء من الكتابة، يضبط الكاتب علامة writerActive على false ويرسل إشارات إلى كل من canWrite وcanRead لإعلام الخيوط المنتظرة.

الخلاصة

يدير هذا التنفيذ القائم على المراقب لمشكلة Read-Writer المزامنة بين القراء والكتاب المتعددين بفعالية. فهو يسمح لقراء متعددين بالوصول إلى المورد المشترك في نفس الوقت مع ضمان حصول الكتاب على وصول حصري عند الكتابة. يساعد استخدام الشاشات في تغليف منطق المزامنة، مما يقلل من خطر الجمود ويجعل الكود أسهل في الفهم والصيانة.