Étude de cas « Simulation de programmes pour la détection d'interblocage »

Denis Conan

Revision : 3580

CSC4102

Télécom SudParis Janvier 2019

1 Lettre de mission

Un groupe d'étudiants en première année d'une école d'ingénieur est mandaté par une communauté de communes pour un audit de la gestion des baignades pendant l'été sur la plage d'un plan d'eau d'une base de loisirs locale. Leur mission est de proposer une gestion apaisée d'un vestiaire extérieur. La communauté de communes souhaite éviter de dédier trop de personnels pour la gestion du vestiaire tout en évitant les confusions.

Les étudiants commencent leur observation de la plage en fonctionnement et remarquent rapidement une situation problématique : plusieurs familles arrivent sur la plage ; chaque famille prend un premier panier puis cherche un second panier, voire un troisième, avant d'entrer dans une cabine pour changer les enfants ; assez rapidement, tout le monde est bloqué en attente d'un panier tout en ayant déjà pris un panier. La fréquence d'apparition de ce premier problème est diminuée en ajoutant des paniers. Ensuite, les étudiants observent que le nombre de paniers permet de contrôler peu ou prou le nombre de personnes se baignant, ce qui est une propriété intéressante. Mais, lorsque l'affluence est grande et qu'il n'y a presque plus de paniers disponibles, les étudiants notent une seconde situation problématique : certains baigneurs « réservent » une cabine avant de chercher les paniers nécessaires tandis que d'autres commencent par prendre les paniers avant la recherche de cabines ; les premiers se retrouvent bloqués en attente de paniers et les seconds en attente de cabines.

Par ailleurs, la communauté de communes prévoit d'améliorer l'attractivité et la convivialité de la plage en mettant à disposition des brassards, des tapis flottants, etc. Aussi, les étudiants se posent la question suivante : « qu'est-ce que ce sera lorsque de multiples types de ressources partagées seront mis à disposition ? »

Lors de leur réflexion, les étudiants analysent ces situations comme un problème d'interblocage aussi appelé étreinte fatale (en anglais, *deadlock*). Ils vous demandent donc de les aider à étudier le problème de manière systématique par simulation. L'idée de départ trouvée dans la littérature est l'utilisation du concept de sémaphore tel qu'introduit par Edsger Dijkstra dans l'annexe d'un article publié dans la revue *Communication of the ACM* en mai 1968[1]. Voici ces éléments :

> "*Explicit mutual synchronization of parallel sequential processes is implemented via so-called 'semaphores.' They are special purpose integer variables allocated in the universe in which the processes are embedded; they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the P-operation and the V-operation.*
> *A process, 'Q' say, that performs the operation 'P(sem)' decreases the value of the semaphore called 'sem' by 1. If the resulting value of the semaphore concerned is nonnegative, process Q can continue with the execution of its next statement; if, however, the resulting value is negative, process Q is stopped and booked on a waiting list associated with the semaphore concerned. Until further notice (i.e. a V-operation on this very same semaphore), dynamic progress of process Q is not logically permissible and no processor will be allocated to it.*
> *A process, 'R' say, that performs the operation 'V(sem)' increases the value of the semaphore called 'sem' by 1. If the resulting value of the semaphore concerned is positive, the V-operation in question has no further effect; if, however, the resulting value of the semaphore concerned is nonpositive, one of the processes booked on its waiting list is removed from this waiting list, i.e. its dynamic progress is again logically permissible and in due time a processor will be allocated to it.*"

Votre mission : créer un simulateur permettant d'écrire des programmes[2] avec des sections critiques (c'est-à-dire des portions de code en exclusion mutuelle) pour établir, par exemple des quatre programmes qui suivent, (1) pourquoi, avec chacun des trois premiers programmes pris séparément, l'exécution concurrente de plusieurs processus exécutant ce programme est sujette à interblocage, et (2) si le dernier programme ne présente pas le même défaut d'autoriser des situations d'étreinte fatale. (Voir l'annexe pour ces explications.) Vous pourrez aussi établir qu'un système composé d'un mélange de processus exécutant le programme 3 ou le programme 4 ci-dessous peut entrer en interblocage.

Programme 1 (avec 2 cabines)

0	acquérir une cabine
1	acquérir une cabine
2	rendre une cabine
3	rendre une cabine

Programme 2

0	acquérir une cabine
1	acquérir un panier

Programme 3

0	acquérir une cabine
1	acquérir un panier
2	rendre une cabine
3	acquérir une cabine
4	rendre un panier
5	rendre une cabine

Programme 4

0	acquérir un panier
1	acquérir une cabine
2	rendre une cabine
3	acquérir une cabine
4	rendre un panier
5	rendre une cabine

Vous appellerez votre simulateur « SimInt » (pour SIMulation de programmes pour la détection d'INTerblocage).

1. `https://doi.org/10.1145/363095.363143`
2. Dans cette étude, uniquement des séquences d'instructions sans structure de contrôle : pas de `if`, `loop`, etc.

2 Cahier des charges

L'objectif du logiciel SimInt est l'étude et la validation de programmes concurrents partageant des ressources de différents types et présentes en plusieurs exemplaires. Le concept de sémaphore tel que présenté dans l'extrait de l'article de E. Dijkstra inséré ci-avant, définit le sémaphore dit binaire (avec le compteur valant 0 ou 1). Vous devez généraliser cette présentation car le compteur d'un sémaphore de E. Dijkstra peut bien sûr être initialisé à une valeur ≥ 1 pour modéliser un ensemble de ressources de même type, un sémaphore initialisé à n gérant alors la disponibilité d'un ensemble de n ressources de même type. Pour appréhender la modélisation de programmes concurrents ainsi que les concepts de sémaphore et d'interblocage, veuillez faire une première lecture de l'annexe A maintenant.

SimInt doit simuler l'exécution d'un système concurrent composé de processus exécutant chacun un programme composé d'instructions (uniquement des manipulations de sémaphores). Une simulation avec SimInt simule donc un seul système concurrent ; il y a donc un seul état global initial du système et un seul graphe d'états globaux atteignables à partir de cet état global initial.

Pour simplifier le travail à réaliser dans le module, qui couvre les deux premières itérations (sprints) du développement de SimInt, les seules variables manipulées par les programmes sont les sémaphores. Nous ignorons donc les ressources gérées par les sémaphores. Un sémaphore est composé d'un compteur (correspondant au nombre de ressources d'un même type qui sont disponibles) et d'une file d'attente (composées des processus bloqués en attente de la cession d'une ressource du type géré par le sémaphore).

Dans une première itération de développement (premier sprint), la solution SimInt doit permettre :

- de créer des sémaphores qui contrôlent l'accès à des ensembles de ressources du même type ;
- d'écrire des programmes composés d'instructions qui manipulent des sémaphores (P(sem) ou proberen(sem) et V(sem) ou verhogen(sem)) ;
- de créer des processus, qui chacun exécute un programme donné ;
- d'exécuter pas-à-pas les instructions de cet ensemble de processus pour faire transiter le système d'un état global vers un état global successeur, un état global étant défini comme la concaténation des états des processus (un état par processus) et des états des sémaphores (un état par sémaphore). Ce mode est appelé le mode « pas-à-pas » ;
- d'établir si le système est en situation d'interblocage dans un état global donné (au moins un processus est bloqué en attente sur un sémaphore et tous les autres processus sont soit aussi bloqués soit déjà terminés).

Dans une seconde itération de développement (second sprint), la solution SimInt doit en plus permettre de construire automatiquement le graphe ou l'arbre des états globaux atteignables à partir de l'état global initial. Ce dernier mode est appelé le mode « validation de modèle » (en anglais, *model checking*). Comme le nombre d'états globaux atteignables à partir d'un état global est important, vous proposerez deux stratégies de construction du graphe des états globaux (jusqu'à la découverte d'une situation d'interblocage ou jusqu'à la fin de l'exécution du système sans interblocage) : (1) étant donné un état global, calcul de tous les états globaux atteignables à partir de cet état global, et (2) choix aléatoire d'un des états globaux atteignables à partir de cet état global.

A Modèle du système à simuler, et concepts de sémaphore et d'interblocage (ou étreinte fatale)

Voici des éléments trouvés dans la littérature par les étudiants de première année qui vous commandent la réalisation du logiciel SimInt. C'est le résultat de leur réflexion pour l'établissement du cahier des charges. Veuillez les étudier avant la séance 2 du module CSC4102 et y revenir autant que nécessaire.

A.1 Modèle de transition

Dans cette étude, nous proposons d'utiliser le modèle dit des transitions, que nous décrivons maintenant.

Un *programme* est une séquence d'instructions. Une instruction assigne des *valeurs* à des *variables*. Un *état* est l'affectation de valeurs à des variables. En cours d'exécution, une *instruction* i représente la relation entre un ancien état s et un nouvel état t notée $s\mathsf{i}t$. L'exécution d'une instruction est atomique : l'instruction i provoque le changement « instantané » de l'état de s à t ; en d'autres termes, le système n'est pas « observable » pendant l'exécution de i. Une *exécution* dans un *processus* p du programme P débutant à l'instant t à partir de l'état initial s^0, est la succession d'un nombre infini d'instructions sur des états notée $\mathsf{i}^0 s^0 \mathsf{i}^1 s^1 \mathsf{i}^2 s^2 \mathsf{i}^3$, etc. L'état s^0 du processus p est appelé l'*état initial*. L'instruction initiale i^0 est fictive et correspond à la création du processus.

Un *système concurrent* est composé de n processus partageant des ressources. L'*état global* (encore appelé *configuration*) s du système concurrent à l'instant τ est composé des états locaux de tous les processus à l'instant τ. L'exécution d'un système concurrent débutant à l'instant physique τ à partir de la configuration s est constituée de la juxtaposition des exécutions des processus.

NB : nous proposons de nes pas modéliser les ressources dans SimInt, mais de modéliser uniquement les sémaphores gérant les ressources (acquisition, cession), un sémaphore par type de ressources. Aussi, en plus de l'état des processus, les seules variables des états globaux sont les sémaphores.

A.2 Sémaphore

Veuillez suivre l'énoncé et le corrigé de l'exercice « La piscine » du module CSC3102 que la plupart d'entre vous avez suivi l'année dernière. Voici l'URL :
— `https://moodle.imtbs-tsp.eu/mod/url/view.php?id=51420`.

Notons que dans l'exercice « La piscine », les shells scripts `P.sh` et `V.sh` utilisent les fonctionnalités du système d'exploitation pour réaliser dans les scripts `acquireresource.sh` et `releaseresource.sh` les fonctionnalités d'un sémaphore. Si vous avez peur de les confondre avec les opérations P et V sur les sémaphores, vous pouvez par exemple nommer les instructions d'acquisition et de cession sur les sémaphores selon leur appelation d'origine en néerlandais[3] : proberen pour P et verhogen pour V.

Les instructions et shell scripts qui correspondent aux manipulations des sémaphores de notre étude de cas sont les suivants :

- P(sem) ou proberen(sem) ≡ `./acquireresource.e.sh sem`, et
- V(sem) ou verhogen(sem) ≡ `./releaseresource.d.sh sem`.

A.3 Interblocage

Dans un système concurrent, le fait que des processus accèdent concurremment à des ressources partagées est une cause possible de blocage. Un interblocage se produit lorsqu'un ensemble de processus est tel que chacun d'eux tient au moins une ressource, et pour poursuivre sa progression, est en attente d'une ressource tenue par l'un des autres.

Pour votre information, il existe trois modèles d'interblocage, chacun d'eux étant illustré par un des petits programmes proposés par les étudiants dans la présentation de la mission :

3. `https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names`

- le modèle « OU » dans lequel un processus doit obtenir plusieurs ressources d'un même type. Un interblocage dans ce modèle est montré ci-après avec le programme 1 : le progamme 1 requiert l'acquisition de deux cabines, et uniquement des cabines ;
- le modèle « ET » dans lequel un processus doit obtenir plusieurs ressources, chacune d'elles étant d'un type différent, et chaque type de ressources étant en un unique exemplaire. Un interblocage dans ce modèle est montré ci-après avec le programme 2 : le programme 2 requiert que le processus acquière l'unique cabine puis l'unique panier ;
- le modèle général « OU–ET » dans lequel un processus peut choisir entre plusieurs ressources d'un même type (le « OU »), et il doit obtenir des ressources de plusieurs types (le « ET »). Un interblocage dans ce modèle est montré ci-après avec le programme 3 : le programme 3 requiert une cabine (deux fois, mais ce n'est obligatoire pour l'exemple) et un panier parmi un ensemble de cabines et un ensemble de paniers.

Voici quatre exemples d'exécution : une exécution pour chacun des trois premiers programmes donnés à titre d'exemple dans la présentation de la mission et qui montre une situation d'interblocage dans les modèles « OU », « ET », « OU–ET » ; une exécution pour le quatrième programme qui ne présente pas d'interblocage. Dans chaque exécution donnée ci-après, les processus exécutent le même programme. Certains programmes sont clairement incorrects, mais encore faut-il s'en convaincre et c'est le rôle de votre logiciel.

Programme 1

```
0  acquérir une cabine
1  acquérir une cabine
2  rendre une cabine
3  rendre une cabine
```

Exemple d'exécution avec interblocage pour 2 processus, 2 cabines

```
1  exécution d'un pas de p1
2  exécution d'un pas de p2
3  exécution d'un pas de p1  // p1 est bloqué en attente
4  exécution d'un pas de p2  // p2 est aussi bloqué en attente ⟹ interblocage
```

Programme 2

```
0  acquérir une cabine
1  acquérir un panier
```

Exemple d'exécution avec interblocage pour 2 processus, 1 cabine, 1 panier

```
1  exécution d'un pas de p1
2  exécution d'un pas de p2  // p2 est bloqué en attente
3  exécution d'un pas de p1  // p1 est terminé ⟹ p2 reste bloqué en attente ⟹ interblocage
```

Programme 3

```
0  acquérir une cabine
1  acquérir un panier
2  rendre une cabine
3  acquérir une cabine
4  rendre un panier
5  rendre une cabine
```

Exemple d'exécution avec interblocage pour 5 processus, 2 cabines, 2 paniers

```
 1  p1 demande et obtient une cabine   // p1 à l'instruction 1
                                       // il reste 1 cabine et 2 paniers
 2  p2 demande et obtient une cabine   // p2 à l'instruction 1
                                       // il reste 2 paniers
 3  p3 demande une cabine              // p3 est bloqué en attente à l'instruction 0
                                       // il reste 2 paniers, file_cabine = {p3}
 4  p4 demande une cabine              // p4 est bloqué en attente à l'instruction 0
                                       // il reste 2 paniers, file_cabine = {p3, p4}
 5  p5 demande une cabine              // p5 est bloqué en attente à l'instruction 0
                                       // il reste 2 paniers, file_cabine = {p3, p4, p5}
 6  p1 demande et obtient un panier    // p1 à l'instruction 2
                                       // il reste 1 panier, file_cabine = {p3, p4, p5}
 7  p2 demande et obtient un panier    // p2 à l'instruction 2
                                       // file_cabine = {p3, p4, p5}
 8  p1 rend sa cabine et débloque p3   // p1, p3 respectivement à l'instruction 3, 1
                                       // file_cabine = {p4, p5}
 9  p2 rend sa cabine et débloque p4   // p2, p4 respectivement à l'instruction 3, 1
                                       // file_cabine = {p5}
10  p3 demande un panier               // p3 est bloqué en attente à l'instruction 1
                                       // file_cabine = {p5}, file_panier = {p3}
11  p4 demande un panier               // p4 est bloqué en attente à l'instruction 1
                                       // file_cabine = {p5}, file_panier = {p3, p4}
12  p1 demande une cabine              // p1 est bloqué en attente à l'instruction 3
                                       // file_cabine = {p5, p1}, file_panier = {p3p4}
13  p2 demande une cabine              // p2 est bloqué en attente à l'instruction 3
                                       // file_cabine = {p5, p1, p2}, file_panier = {p3p4}
                                       // ⟹ interblocage
```

Programme 4

0	acquérir un panier
1	acquérir une cabine
2	rendre une cabine
3	acquérir une cabine
4	rendre un panier
5	rendre une cabine

Exemple d'exécution sans interblocage pour 5 processus, 2 cabines, 2 paniers

1	p_1 demande et obtient un panier	// p_1 à l'instruction 1 // il reste 1 panier et 2 cabines
2	p_2 demande et obtient un panier	// p_2 à l'instruction 1 // il reste 2 cabines
3	p_1 demande et obtient une cabine	// p_1 à l'instruction 2 // il reste 1 cabine
4	p_3 demande un panier	// p_3 est bloqué en attente à l'instruction 0 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_3\}$
5	p_2 demande et obtient une cabine	// p_2 à l'instruction 2 // $\text{file}_{\text{panier}} = \{p_3\}$
6	p_4 demande un panier	// p_4 est bloqué en attente à l'instruction 0 // $\text{file}_{\text{panier}} = \{p_3, p_4\}$
7	p_1 rend sa cabine	// p_1 à l'instruction 3 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_3, p_4\}$
8	p_2 rend sa cabine	// p_2 à l'instruction 3 // il reste 2 cabines, $\text{file}_{\text{panier}} = \{p_3, p_4\}$
9	p_5 demande un panier	// p_5 est bloqué en attente à l'instruction 0 // il reste 2 cabines, $\text{file}_{\text{panier}} = \{p_3, p_4, p_5\}$
10	p_1 demande et obtient une cabine	// p_1 à l'instruction 4 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_3, p_4, p_5\}$
11	p_2 demande et obtient une cabine	// p_2 à l'instruction 4 // $\text{file}_{\text{panier}} = \{p_3, p_4, p_5\}$
12	p_1 rend son panier et débloque p_3	// p_1, p_3 respectivement à l'instruction 5, 1 // $\text{file}_{\text{panier}} = \{p_4, p_5\}$
13	p_1 rend sa cabine	// p_1 est terminé // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_4, p_5\}$
14	p_2 rend son panier et débloque p_4	// p_2, p_4 respectivement à l'instruction 5, 1 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_5\}$
15	p_2 rend sa cabine	// p_2 est terminé // il reste 2 cabines, $\text{file}_{\text{panier}} = \{p_5\}$
16	p_3 demande et obtient une cabine	// p_3 à l'instruction 2 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_5\}$
17	p_4 demande et obtient une cabine	// p_4 à l'instruction 2 // $\text{file}_{\text{panier}} = \{p_5\}$
18	p_3 rend sa cabine	// p_3 à l'instruction 3 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_5\}$
19	p_4 rend sa cabine	// p_4 à l'instruction 3 // il reste 2 cabines, $\text{file}_{\text{panier}} = \{p_5\}$
20	p_3 demande et obtient une cabine	// p_3 à l'instruction 4 // il reste 1 cabine, $\text{file}_{\text{panier}} = \{p_5\}$
21	p_4 demande et obtient une cabine	// p_4 à l'instruction 4 // $\text{file}_{\text{panier}} = \{p_5\}$
22	p_3 rend son panier et débloque p_5	// p_3, p_5 respectivement à l'instruction 5, 1
23	p_4 rend son panier	// p_4 à l'instruction 5 // il reste 1 panier
24	p_3 rend sa cabine	// p_3 est terminé // il reste 1 panier et 1 cabine
25	p_4 rend sa cabine	// p_4 est terminé // il reste 1 panier et 2 cabines
26	p_5 demande et obtient une cabine	// p_5 à l'instruction 2 // il reste 1 panier et 1 cabine
27	p_5 rend sa cabine	// p_5 à l'instruction 3 // il reste 1 panier et 2 cabines
28	p_5 demande et obtient une cabine	// p_5 à l'instruction 4 // il reste 1 panier et 1 cabine
29	p_5 rend son panier	// p_5 à l'instruction 5 // il reste 2 paniers et 1 cabine
30	p_5 rend sa cabine	// p_5 est terminé // il reste 2 paniers et 2 cabines // $\Longrightarrow$ l'exécution du système est terminée

NB : la manière de détecter l'interblocage proposée pour SimInt, c'est-à-dire par simulation, est différente de celle utilisée dans l'exercice « La piscine » du module CSC3102[4] :

- dans l'exercice du module CSC3102, le shell script `detectdeadlock.sh` vérifie la formule suivante : $\forall r \in R = \{\text{cabine}, \text{panier}\}, \forall p, p$ détenant une ressource de type r $\Longrightarrow$ $p \in \text{file}_{\text{R}\setminus\{\text{r}\}}$;
- dans SimInt, pour généraliser en nous affranchissant des ressources manipulées, nous proposons de détecter l'interblocage avec la condition nécessaire et suffisante qui suit :
  étant donné un état global obtenu par simulation, il n'est plus possible d'avancer dans l'exécution du système alors que tous les processus ne sont pas à la fin de leur programme.

Veuillez observer que dans l'exercice du module CSC3102, les ressources sont « modélisées » alors qu'elles ne le seront pas dans SimInt. En outre, remarquez que, dans l'exercice du module CSC3102, nous ne détectons pas la fin des processus, contrairement à ce que nous ferons dans la simulation avec SimInt.

4. `https://moodle.imtbs-tsp.eu/mod/url/view.php?id=51420`