Faculté des Sciences & Techniques — Université de Limoges

Licence 3ème année

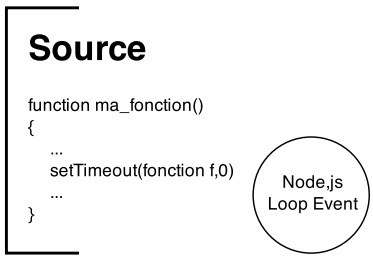
Programmation Concurrente

TD n°3

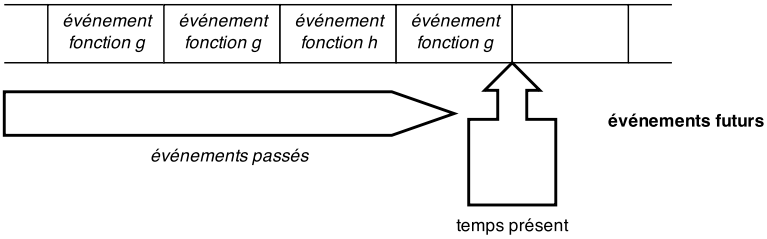
Programmation asynchrone

Exécution asynchrone et boucle d'événement

1 –



a. À quoi correspond `setTimeout(fonction,0)` ?
b. Montrer les transformations de la liste d'événements ;
c. Qu'est-ce que la « boucle d'événements », « *loop event* » ? Comment utilise-t-elle la liste d'événements ?
d. Quand un événement est-il pris en compte ?



Fonction à conservation d'état

2 – Soit le code du « producteur/consommateur » :

```
1  var produit = 0;
2
3  function appel_producteur() {
4      producteur.next();
5  }
6
7  function* p() {
8      var compteur = 0;
9      while(true)
10     {
11         compteur++;
12         produit = compteur;
13         console.log("Production " + pro
   duit);
14         setTimeout(appel_consommateur,0);
15         yield null;
16     }
17 }
```

```
1  function appel_consommateur() {
2      consommateur.next();
3  }
4
5  function* c() {
6      while(true)
7      {
8          console.log("Consommation "+pro
   duit);
9          setTimeout(appel_producteur,
   0);
10         yield null;
11     }
12 }
13
14 var producteur = p();
15 var consommateur = c();
16
17 producteur.next();
```

a. À quoi sert le caractère « * » dans la déclaration `function *p()` ?
b. À quoi sert l'instruction « `yield` » ?
c. Pourquoi déclare-t-on une variable en lui affectant l'exécution d'une fonction : `var producteur = p()` ?
d. Qu'est-ce que fait la méthode « `next()` » ?
e. Pourquoi a-t-on besoin de la fonction « `appel_producteur()` » ?
f. Étudier l'exécution du « producteur/consommateur » où vous détaillerez l'évolution de la liste d'événement en fonction de l'exécution du producteur et du consommateur.

Équité et « Scheduling »

3 – Soit le code des philosophes mangeur de ramen :

```
var liste_philosophes = [];
var liste_philosophes_a_traiter = null;
var philosophe_a_reveiller = null;

var compteur_chaises = 4;

function reveil_philosophe()
{
    philosophe_a_reveiller.next();
}

function* philosophe(n)
{
    while(true)
    {
         if (compteur_chaises > 0)
         {
             compteur_chaises --;
             console.log("Philosophe : "+n+" prend une chaise");
             yield null;
             console.log("Philosophe : "+n+" mange");
             yield null;
             console.log("Philosophe : "+n+" se leve");
             compteur_chaises ++;
         }
         else
         {
             console.log("Philosophe : "+n+" reflechit");
         }
         yield null;
    }
}

for(i=0; i<5; i++)
        liste_philosophes.push(philosophe(i));

liste_philosophes_a_traiter = liste_philosophes.slice(0);

function scheduler()
{
     var numero_philosophe = Math.floor(Math.random() * liste_philosophes_a_trai
ter.length);
     philosophe_a_reveiller = liste_philosophes_a_traiter[numero_philosophe];
     liste_philosophes_a_traiter.splice(numero_philosophe, 1);
     if (liste_philosophes_a_traiter.length == 0)
     {
             liste_philosophes_a_traiter = liste_philosophes.slice(0);
     }
     setTimeout(reveil_philosophe,0);
     setTimeout(scheduler,0);
}

scheduler();
```

Étudier l'exécution du programme résolvant le problème des philosophes :

a. Quelle activité est réalisée par chaque philosophe ?

b. À quoi servent les différents « `yield` » ?

c. Comment est utilisée la liste d'événements et la boucle d'événement ?

d. Pourquoi parle-t-on de « *scheduling* », « ordonnancement » ?

e. Est-il équitable ? Est-ce qu'il y a des risques de famine ? Pourquoi ?

4 – Proposez une solution au problème du producteur/consommateur :

- ☐ équitable ;
- ☐ utilisant un buffer de 5 cases.