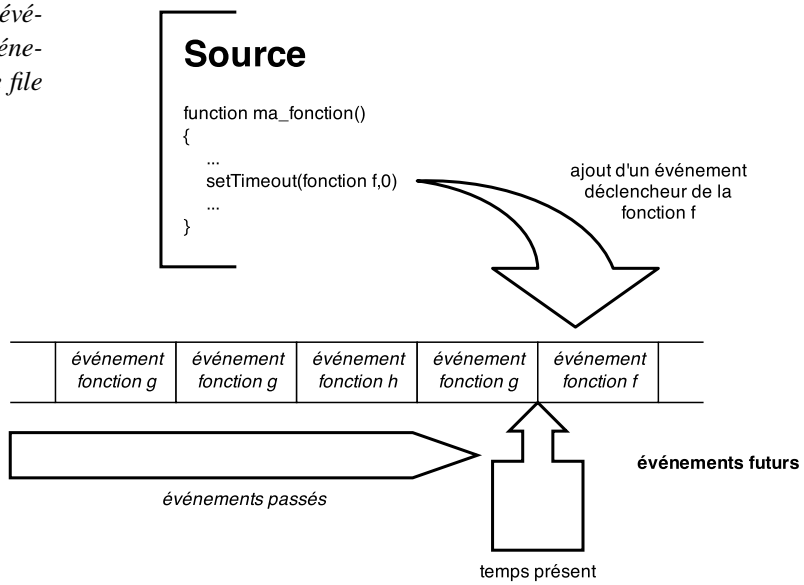
Programmation asynchrone

Exécution asynchrone et boucle d'événement

1 – a. À quoi correspond `setTimeout(fonction,0)` ?
À l'ajout d'un événement temporisé dans la liste d'événements :
- *le déclencheur est automatique, il correspond à attendre pendant une durée nulle ;*
- *lorsque l'événement se déclenche, la fonction « `fonction` » est appelée.*

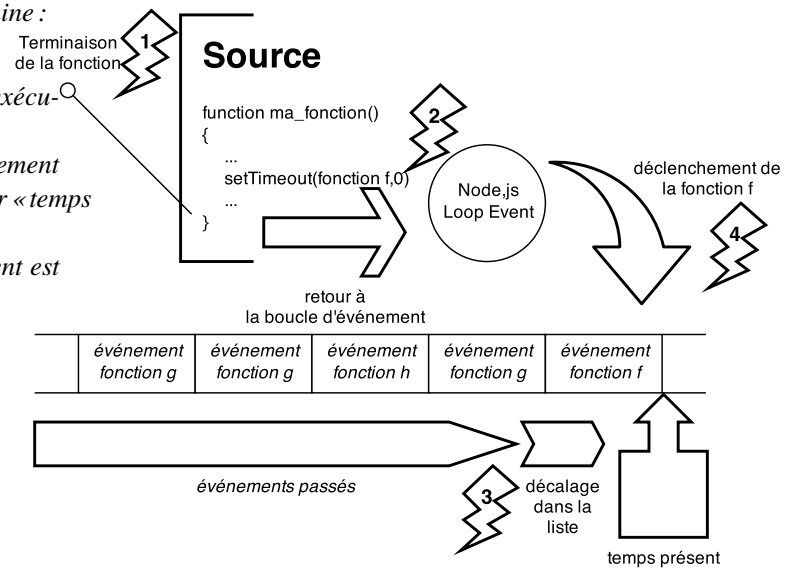
b. Montrer les transformations de la liste d'événements ;

On ajoute l'événement dans les événements futurs dans la liste d'événements qui fonctionne comme une file d'attente :



c. Qu'est-ce que la « boucle d'événements », « *loop event* » ? *C'est le « contrôle » de Node.js : il s'exécute lorsque le programme de l'utilisateur ne s'exécute pas.*
Comment utilise-t-elle la liste d'événements ?
Lorsqu'elle se déclenche, la « boucle d'événement » parcours la liste d'événements et appelle les fonctions associées à chaque événement.

d. Quand un événement est-il pris en compte ?
Lorsque la fonction courante se termine :
- *en 1 : la fonction se termine ;*
- *en 2 : la boucle d'événement est exécutée de nouveau ;*
- *en 3 : elle prend le premier événement « futur » et fait avancer le pointeur « temps présent » ;*
- *en 4 : la fonction `f` de l'événement est appelée.*



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 }
```

```
18 function appel_consommateur() {
19      consommateur.next();
20 }
21
22 function* c() {
23     while(true)
24     {
25          console.log("Consommation "+pro
duit);
26          setTimeout(appel_producteur,
0);
27          yield null;
28     }
29 }
30
31 var producteur = p();
32 var consommateur = c();
33
34 producteur.next();
```

a. À quoi sert le caractère « `*` » dans la déclaration `function *p()` ?
À indiquer que la fonction va conserver son état courant entre plusieurs appels.

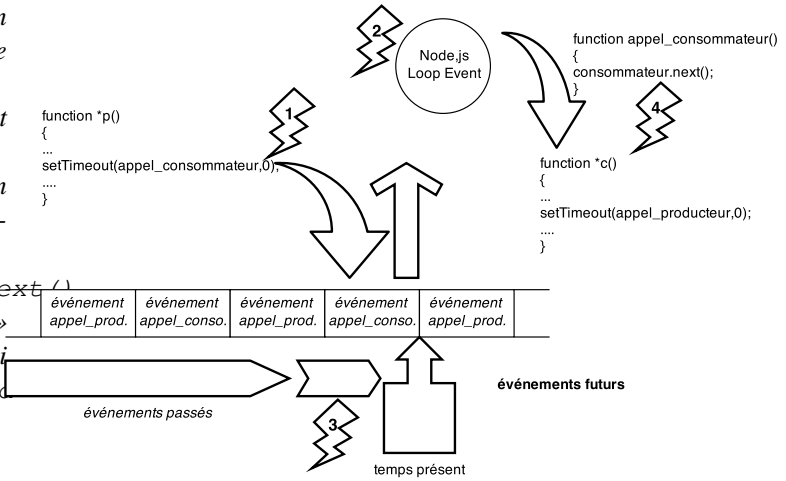
b. À quoi sert l'instruction « `yield` » ?
À suspendre l'exécution de la fonction, à retourner une valeur, ici `null`, et à mémoriser son état courant .

c. Pourquoi déclare-t-on une variable en lui affectant l'exécution d'une fonction :
`var producteur = p()` ? *Pour démarrer le processus de conservation d'état : on récupére un objet qui représente l'état courant de la fonction et qui dispose d'une méthode `next()`*

d. Qu'est-ce que fait la méthode « `next()` » ? *Elle rappelle la fonction qui reprend depuis l'état obtenu avant le « `yield` ».*

e. Pourquoi a-t-on besoin de la fonction « `appel_producteur()` » ? *Parce qu'un événement appelle une fonction uniquement et pas la méthode « `next()` » comme on le voudrait.*

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.
   - ◇ *en 1 : le producteur ajoute un événement (ligne 14), avant de se suspendre (ligne 15) ;*
   - ◇ *en 2 : la boucle d'événement reprend l'exécution :*
   - ◇ *en 3 : elle enlève le prochain événement et exécute la fonction associée (ligne 19) ;*
   - ◇ *en 4 : la fonction exécute un `next()` qui rappelle la fonction « `c` » dans son état sauvegardé qui reprend juste après le « `yield` » (ligne 28) ;*



Équité et « Scheduling »

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

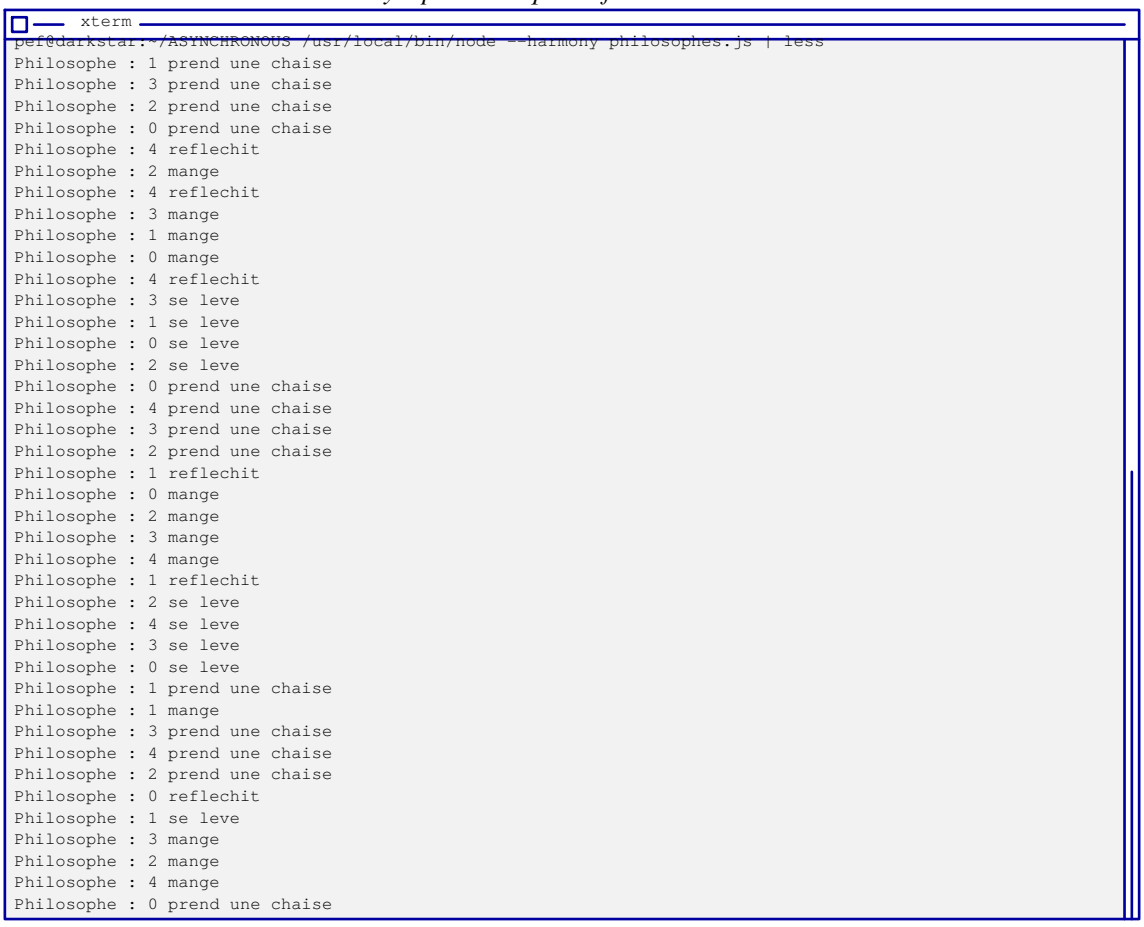
a. Quelle activité est réalisée par chaque philosophe ?
Chaque philosophe effectue une boucle infinie d'actions :
- *« prend une chaise »*
- *« mange »*
- *« se leve »*

b. À quoi servent les différents « `yield` » ?
À suspendre le « philosphe » après chaque action et à laisser la « boucle d'événement » s'exécuter.
Cela permet d'entrelacer les actions des différents philosophes.
L'utilisation du « yield » permet de mémoriser l'état du philosophe qui se suspend, afin de reprendre juste après sa suspension sur la prochaine action qu'il doit effectuer.

c. Comment est utilisée la liste d'événements et la boucle d'événement ?
Uniquement par le « scheduler » : il ajoute un événement de réveil d'un philosophe, suivi d'un événement permettant de le rappeler.
On remarquera que l'état du « scheduler » correspond à l'état de la liste « `liste_philosophes_a_traiter` ».

d. Pourquoi parle-t-on de « *scheduling* », « ordonnancement » ?
Parce que le déroulement de l'utilisation de la liste d'événement permer de passer successivement de l'action d'un philosophe, à la fonction « scheduler » qui sert à choisir de quel philosophe, on exécutera la prochaine action, ce qui par définition est le travail d'un « ordonnanceur »
C'est le « scheduler » qui fait avancer le programme et qui gère l'alternance des actions.

e. Est-il équitable ? Est-ce qu'il y a des risques de famine ? Pourquoi ?
Le traitement est équitable car :
- *c'est le « scheduler » qui choisit quel philosophe va effectuer sa prochaine action ;*
- *le choix du philosophe se fait par sélection d'un philosophe dans une liste qui est ensuite diminuée de ce philosophe : cela veut dire que* ***tous les philosophes*** *vont être traités dans un ordre aléatoire, avant de recommencer. Il n'y a pas de risque de famine.*



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

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

```
var compteur_produits = 0;

function appel_producteur() {
        producteur.next();
}

function appel_consommateur() {
        consommateur.next();
}

function* p() {
        var compteur = 0;
        while(true)
        {
                compteur_produits++;
                console.log("Production : " + compteur_produits);
                yield null;
        }
}

function* c() {
        while(true)
        {
                compteur_produits--;
                console.log("Consommation : " + compteur_produits);
                yield null;
        }
}
```

```
function scheduler()
{
        if (compteur_produits == 0)
        {
                console.log("Tampon vide");
                setTimeout(appel_producteur, 0);
                return setTimeout(scheduler,0);
        }
        if (compteur_produits == 5)
        {
                console.log("Tampon plein");
                setTimeout(appel_consommateur,0);
                return setTimeout(scheduler,0);
        }
        var numero = Math.floor(Math.random() * 2);
        if (numero == 0)
        {
                setTimeout(appel_producteur, 0);
                return setTimeout(scheduler,0);
        }
        else
        {
                setTimeout(appel_consommateur,0);
                return setTimeout(scheduler,0);
        }
}

var producteur = p();
var consommateur = c();

scheduler();
```

xterm

```
pef@darkstar:~/ASYNCHRONOUS /usr/local/bin/node --harmony prod_conso.js | less
Tampon vide
Production : 1
Consommation : 0
Tampon vide
Production : 1
Consommation : 0
Tampon vide
Production : 1
Consommation : 0
Tampon vide
Production : 1
Production : 2
Consommation : 1
Consommation : 0
Tampon vide
Production : 1
Consommation : 0
Tampon vide
Production : 1
Consommation : 0
Tampon vide
Production : 1
Production : 2
Production : 3
Consommation : 2
Production : 3
Production : 4
Consommation : 3
Production : 4
Production : 5
Tampon plein
Consommation : 4
Consommation : 3
Production : 4
Production : 5
Tampon plein
```