UNIVERSITÉ PARIS SUD

IUT D'ORSAY

M3101 · Principes des systèmes d'exploitation

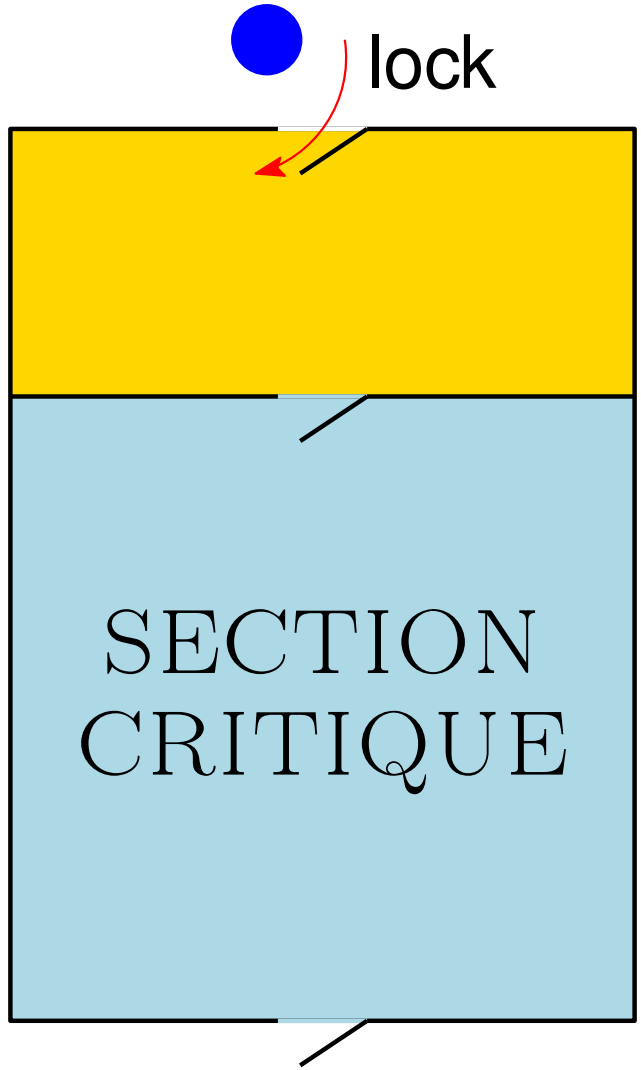
Moniteurs

Producteur – Consommateur

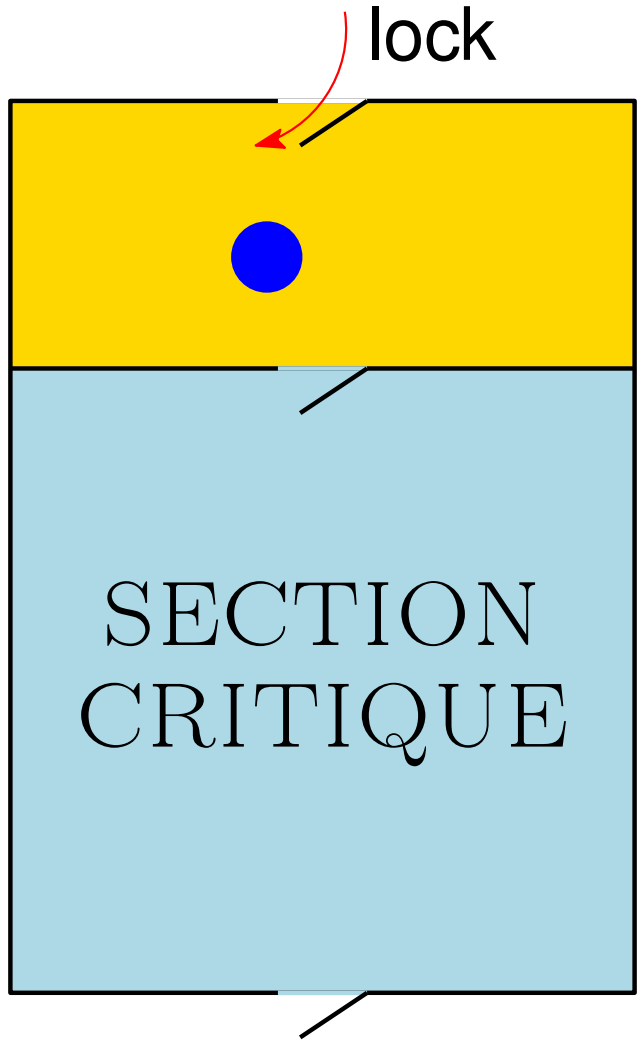
```
unsigned int value;  // le nombre d'unités disponibles

int acquire(int units) {
  if (units > value)          // si pas assez...
    return 0;                 // ...signaler l'échec
  value -= units;             // consommer
  return 1;                   // signaler le succès
}

void release(int units) {
  value += units;             // produire
}
```

Accès simultané aux ressources partagées $\Rightarrow$ risque d'interférence

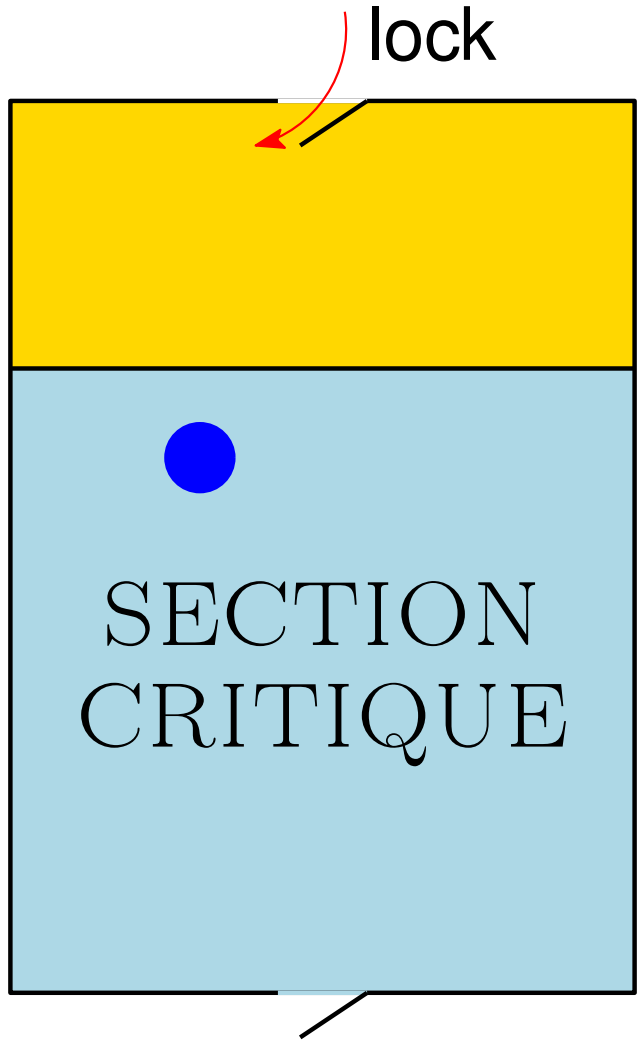
Exclusion mutuelle



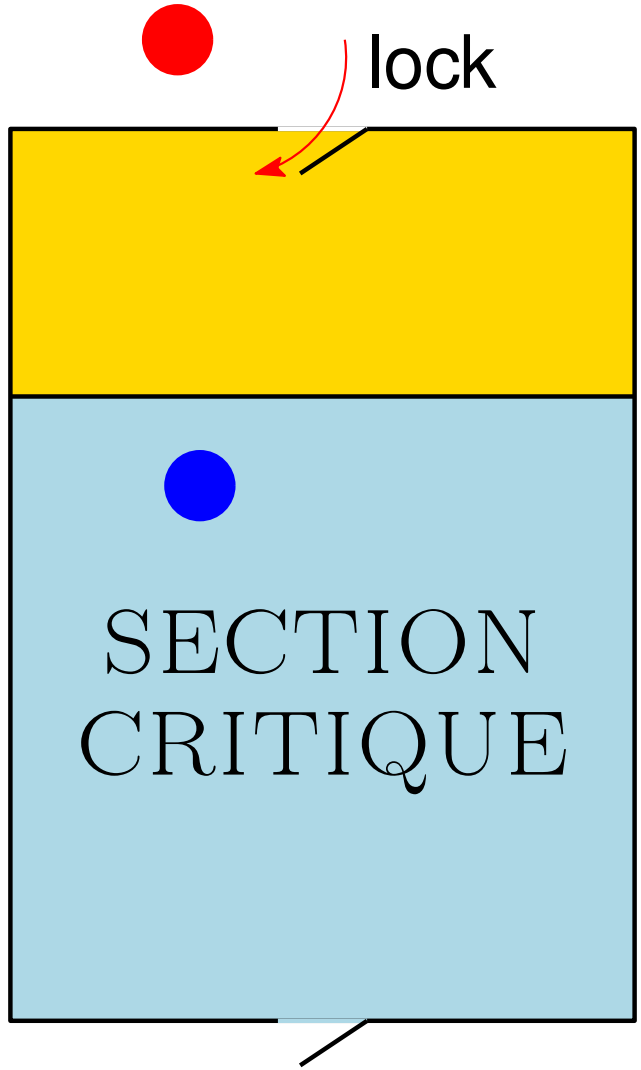
Exclusion mutuelle



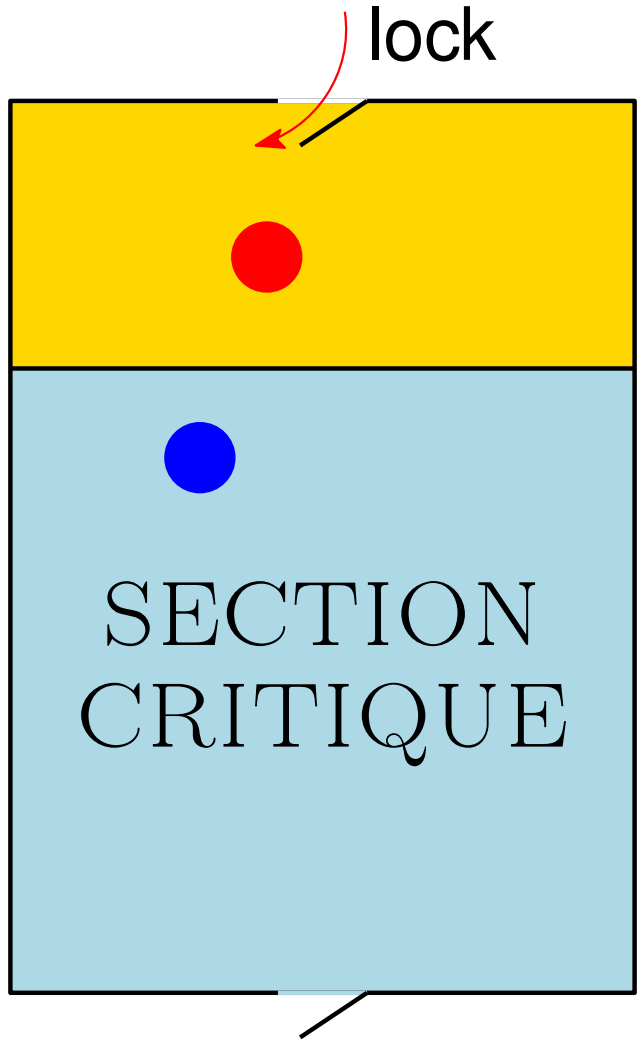
Exclusion mutuelle



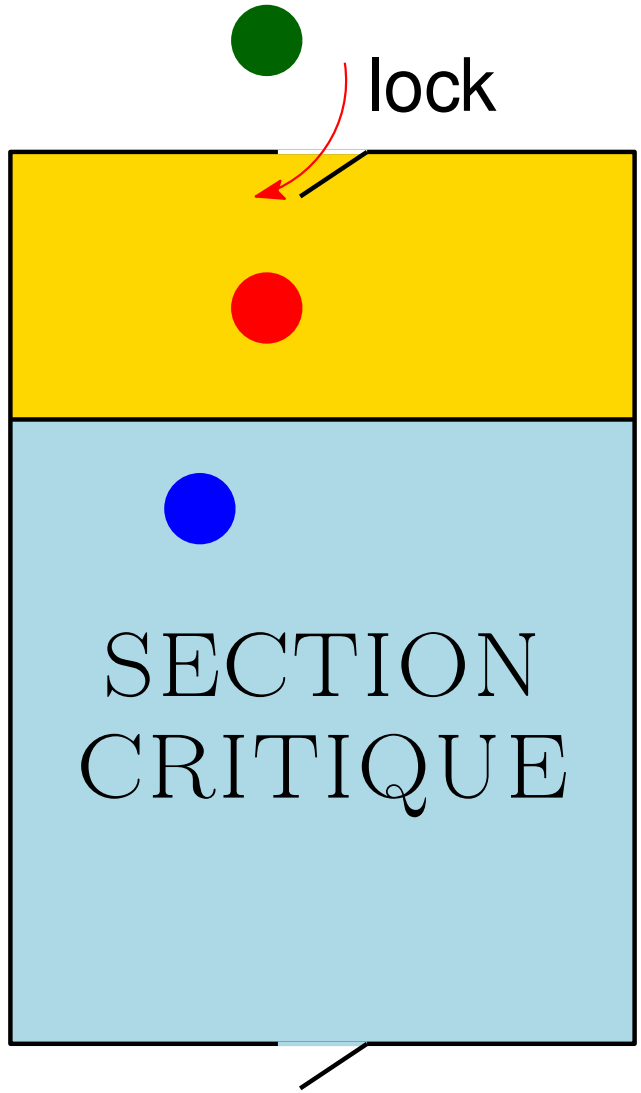
Exclusion mutuelle



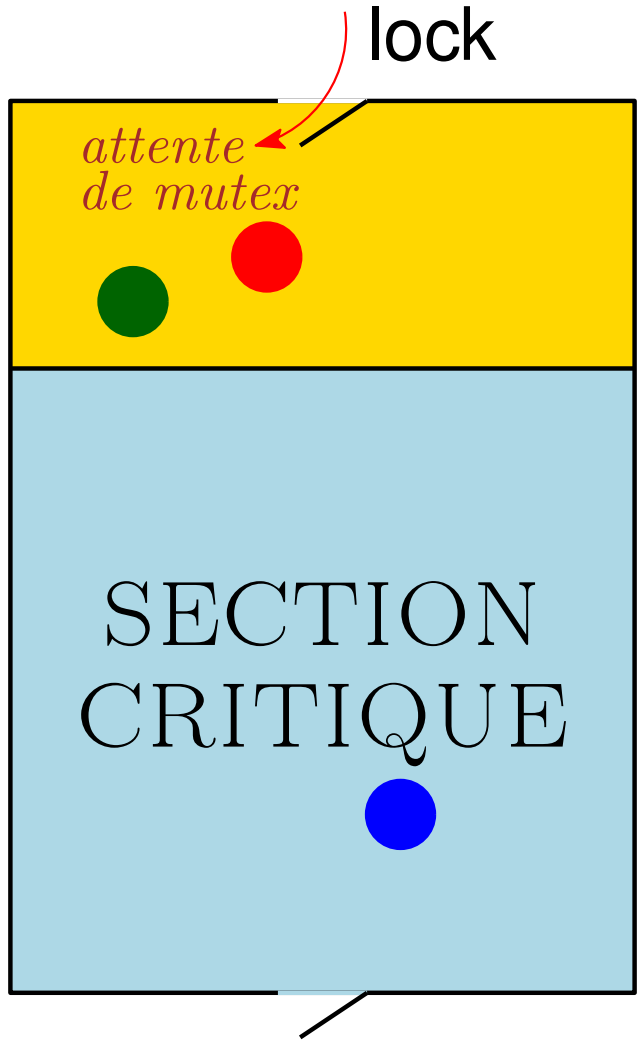
Exclusion mutuelle



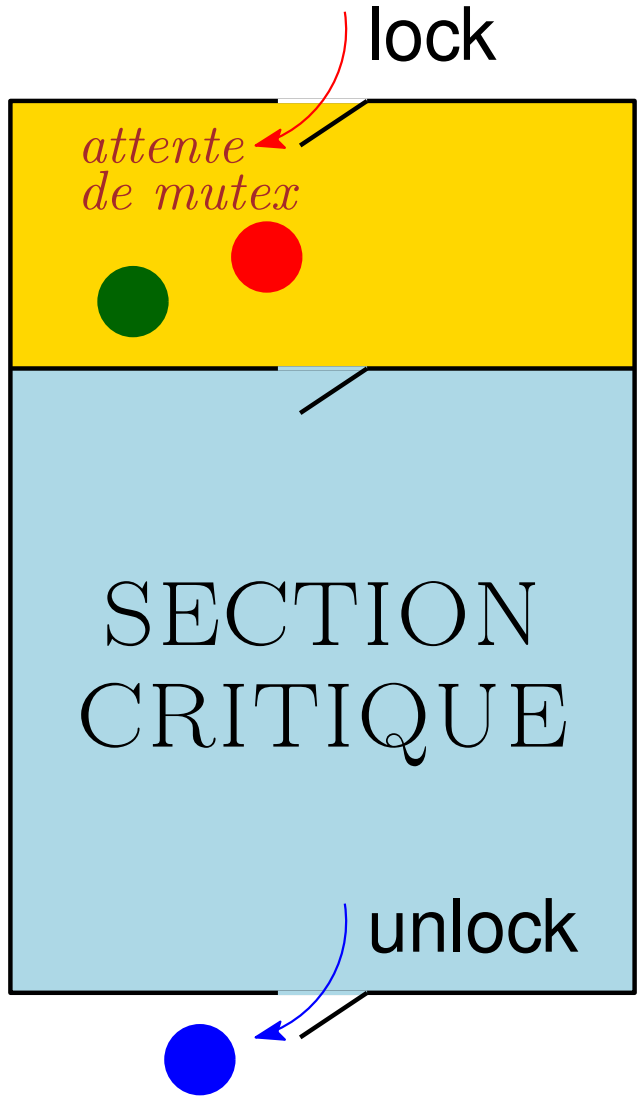
Exclusion mutuelle



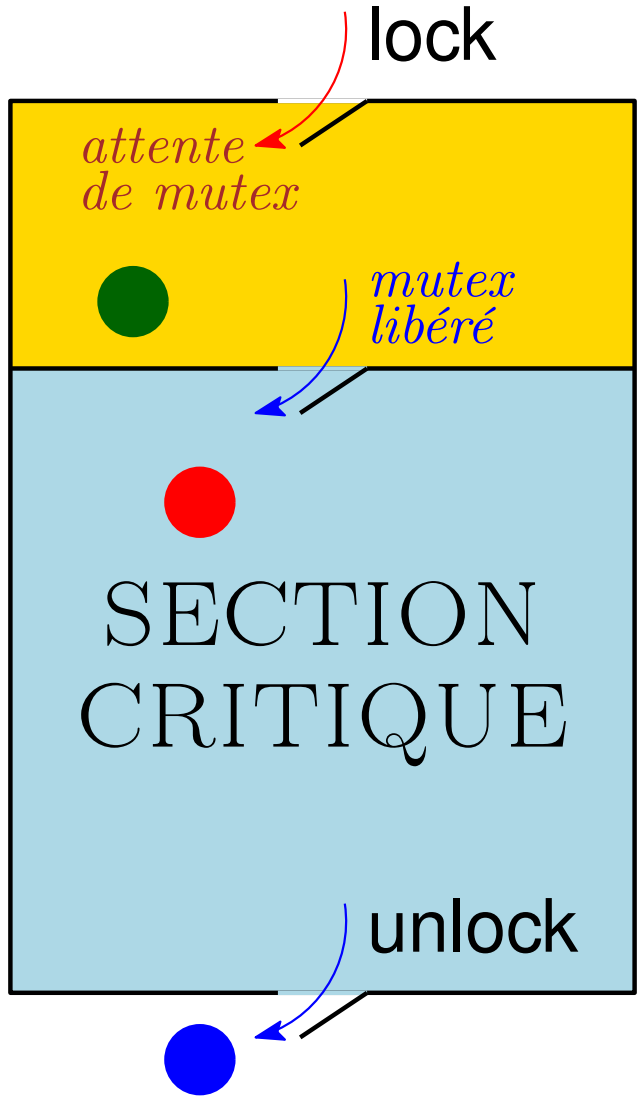
Exclusion mutuelle



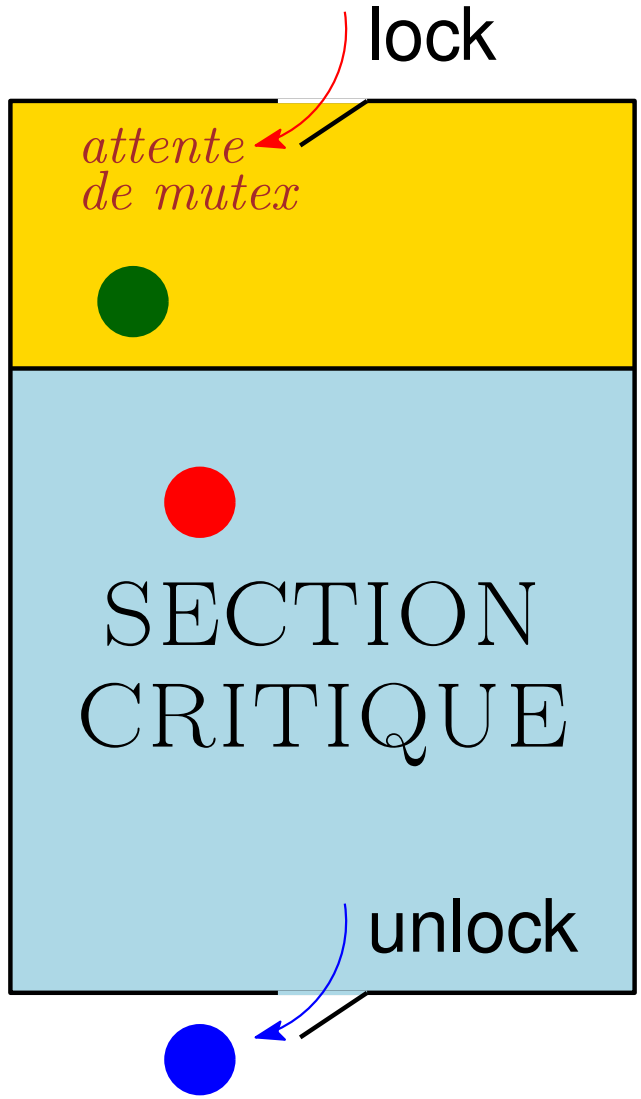
Exclusion mutuelle



Exclusion mutuelle



Exclusion mutuelle



Producteur – Consommateur

```
pthread_mutex_t mutex;  // le mutex du sémaphore
unsigned int    value;  // le nombre d'unités disponibles

int acquire(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  if (units > value) {            // si pas assez...
    pthread_mutex_unlock(&mutex); // ...libérer le mutex
    return 0; }                   // ...signaler l'échec
  value -= units;                 // consommer
  pthread_mutex_unlock(&mutex);   // libérer le mutex
  return 1; }                     // signaler le succès

void release(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  value += units;                 // produire
  pthread_mutex_unlock(&mutex); } // libérer le mutex
```

Accès protégé par l'exclusion mutuelle $\Rightarrow$ pas de risque d'interférence

Producteur – Consommateur

```
pthread_mutex_t mutex;  // le mutex du sémaphore
unsigned int    value;  // le nombre d'unités disponibles

int acquire(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  if (units > value) {            // si pas assez...
    pthread_mutex_unlock(&mutex); // ...libérer le mutex
    return 0; }                   // ...signaler l'échec
  value -= units;                 // consommer
  pthread_mutex_unlock(&mutex);   // libérer le mutex
  return 1; }                     // signaler le succès

void release(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  value += units;                 // produire
  pthread_mutex_unlock(&mutex); } // libérer le mutex
```

Et si on voulais garantir que `acquire()` n'échoue jamais ?

Producteur – Consommateur

```
pthread_mutex_t mutex;  // le mutex du sémaphore
unsigned int    value;  // le nombre d'unités disponibles

void acquire(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  while (units > value)           // tant que pas assez...
  {/* ne rien faire */}           // ...attendre
  value -= units;                 // consommer
  pthread_mutex_unlock(&mutex); } // libérer le mutex

void release(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  value += units;                 // produire
  pthread_mutex_unlock(&mutex); } // libérer le mutex
```

`acquire()` retient le mutex $\Rightarrow$ `release()` ne peut pas accéder à `value`

Producteur – Consommateur

```
pthread_mutex_t mutex;  // le mutex du sémaphore
unsigned int    value;  // le nombre d'unités disponibles

void acquire(int units) {
  while (1) {
    pthread_mutex_lock(&mutex);   // verrouiller le mutex
    if (units <= value) break;    // tester si assez
    pthread_mutex_unlock(&mutex); // libérer le mutex
  }
  value -= units;                 // consommer
  pthread_mutex_unlock(&mutex); } // libérer le mutex

void release(int units) {
  pthread_mutex_lock(&mutex);     // verrouiller le mutex
  value += units;                 // produire
  pthread_mutex_unlock(&mutex); } // libérer le mutex
```

`acquire()` utilise le processeur pendant l'attente $\Rightarrow$ *busy waiting*

Problème : il faut verrouiller le mutex pour voir si on peut avancer

- il faut libérer le mutex si on ne peut pas avancer
- il faut reprendre le mutex plus tard pour refaire le test
- *mais à quel moment ?*

Solution : variables de condition

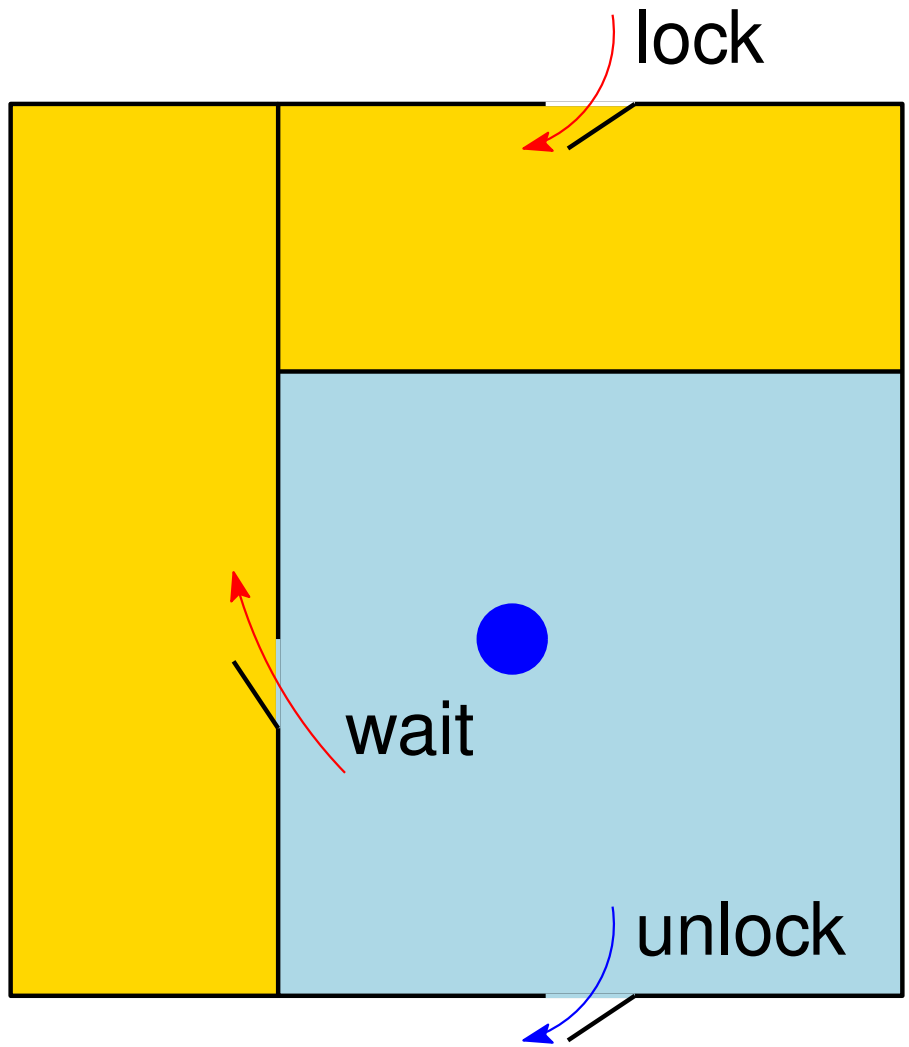
- une « salle d'attente » pour les threads qui demandent une condition

  par exemple, un certain nombre d'unités disponibles
- un thread passe en attente $\Rightarrow$ le mutex est libéré

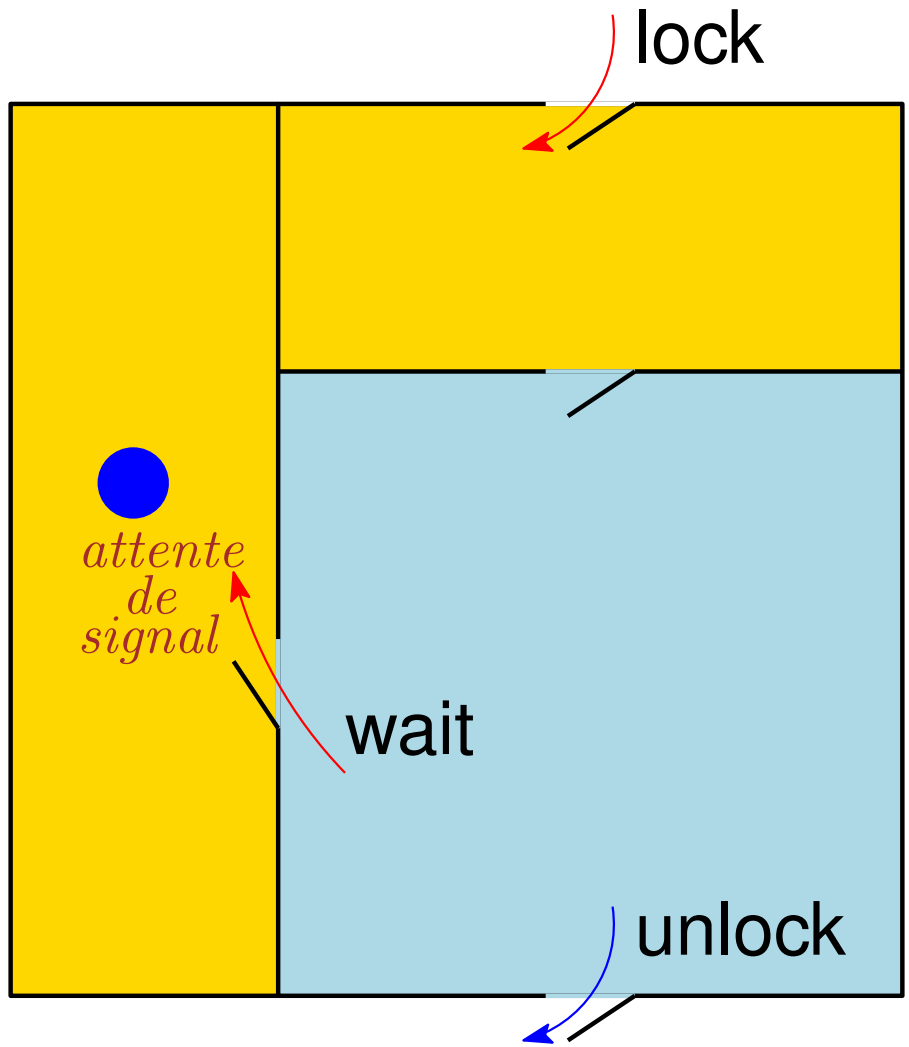
  un autre thread peut intervenir entre-temps et satisfaire la condition
- la condition est satisfaite $\Rightarrow$ on notifie les threads en attente

  les threads notifiés reprennent le mutex et réessayent

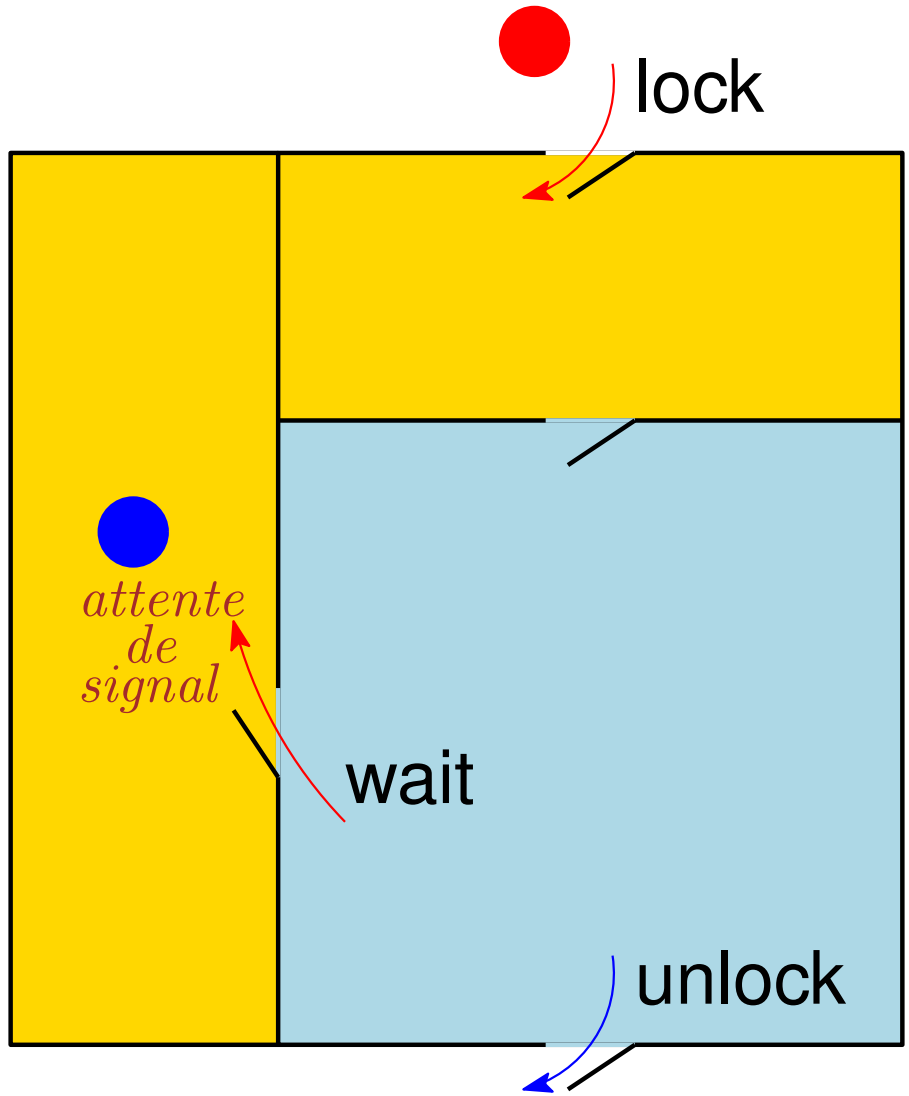
Variables de condition



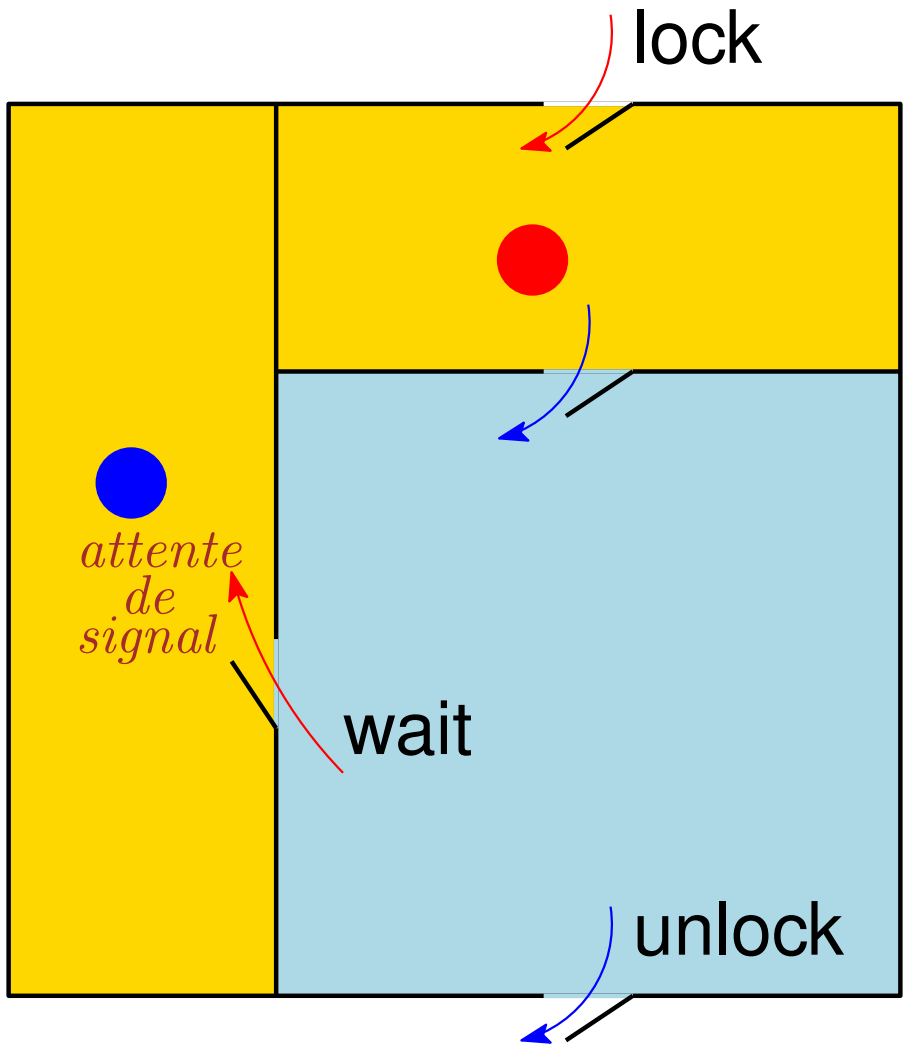
Variables de condition



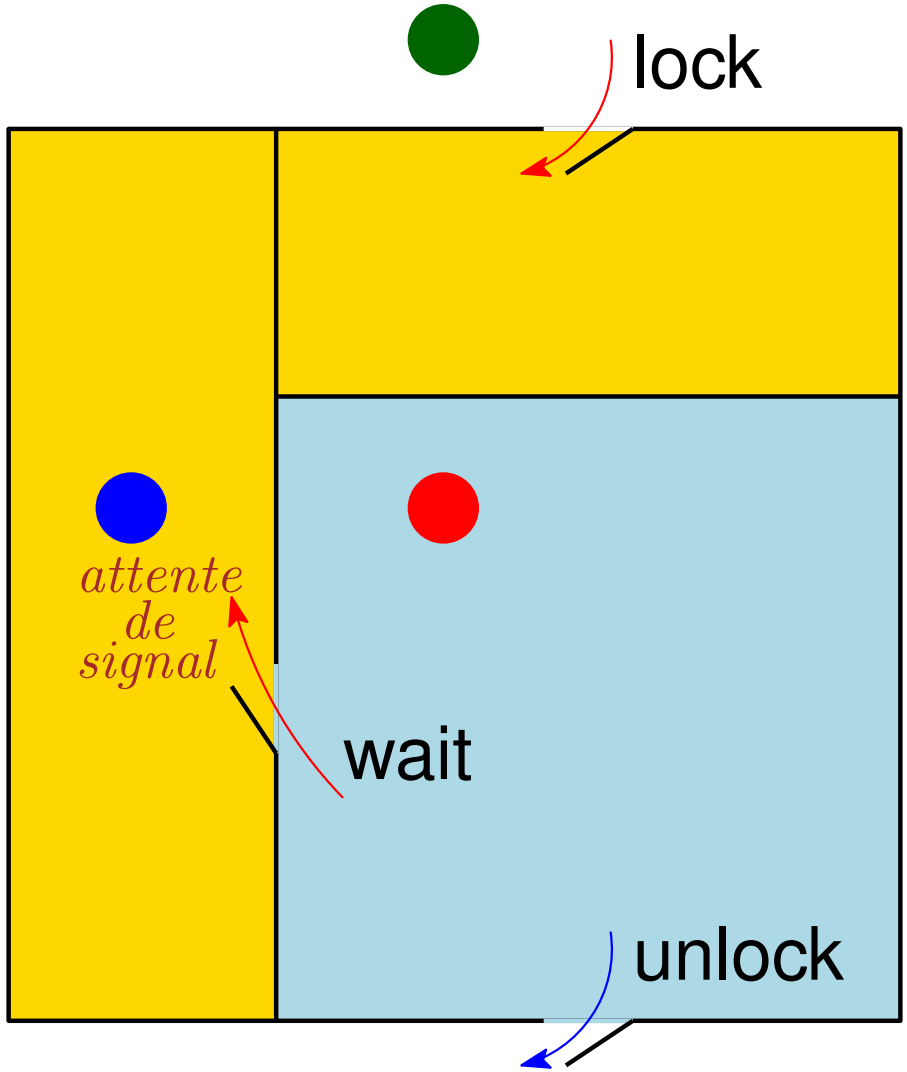
Variables de condition



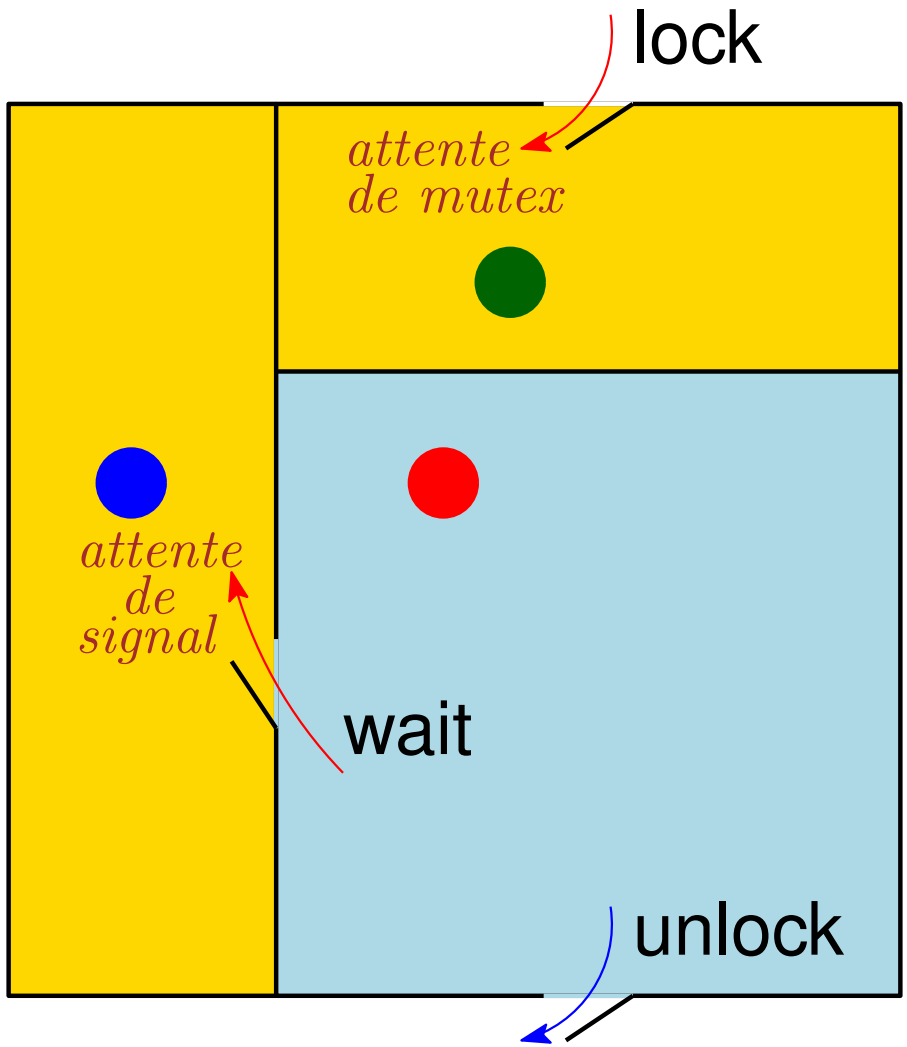
Variables de condition



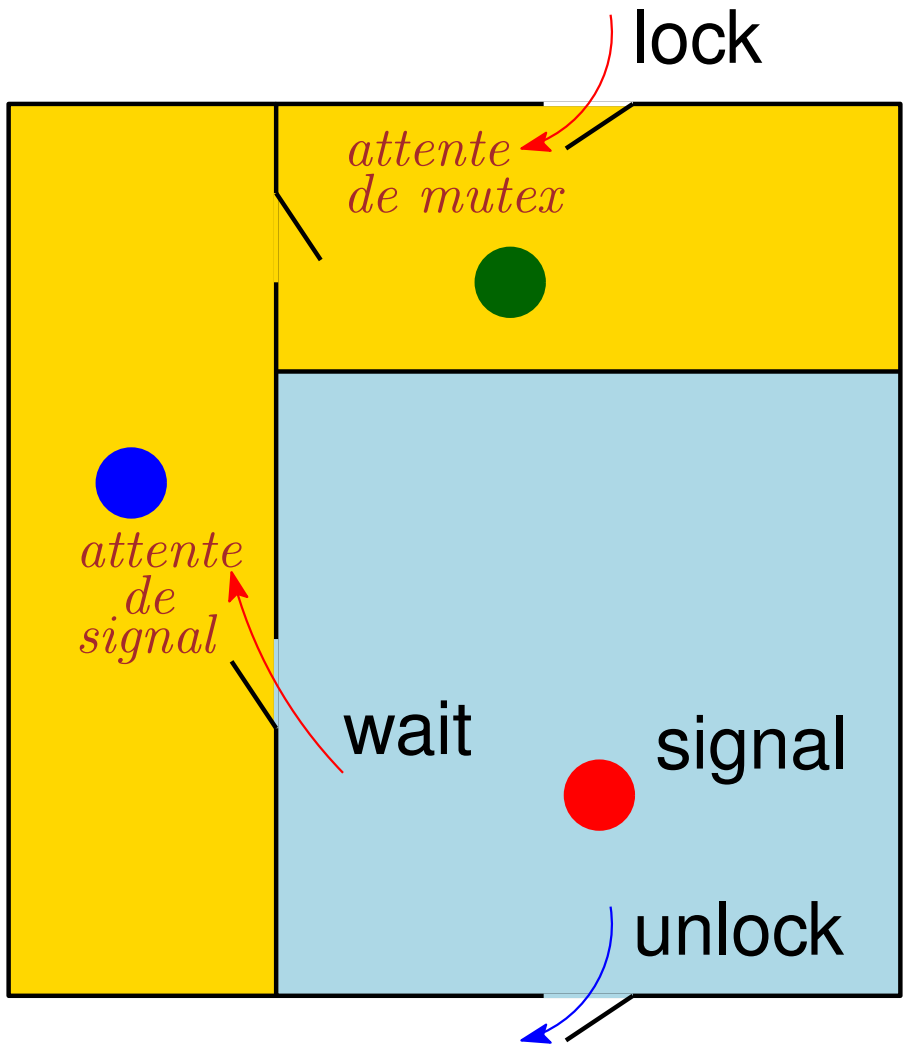
Variables de condition



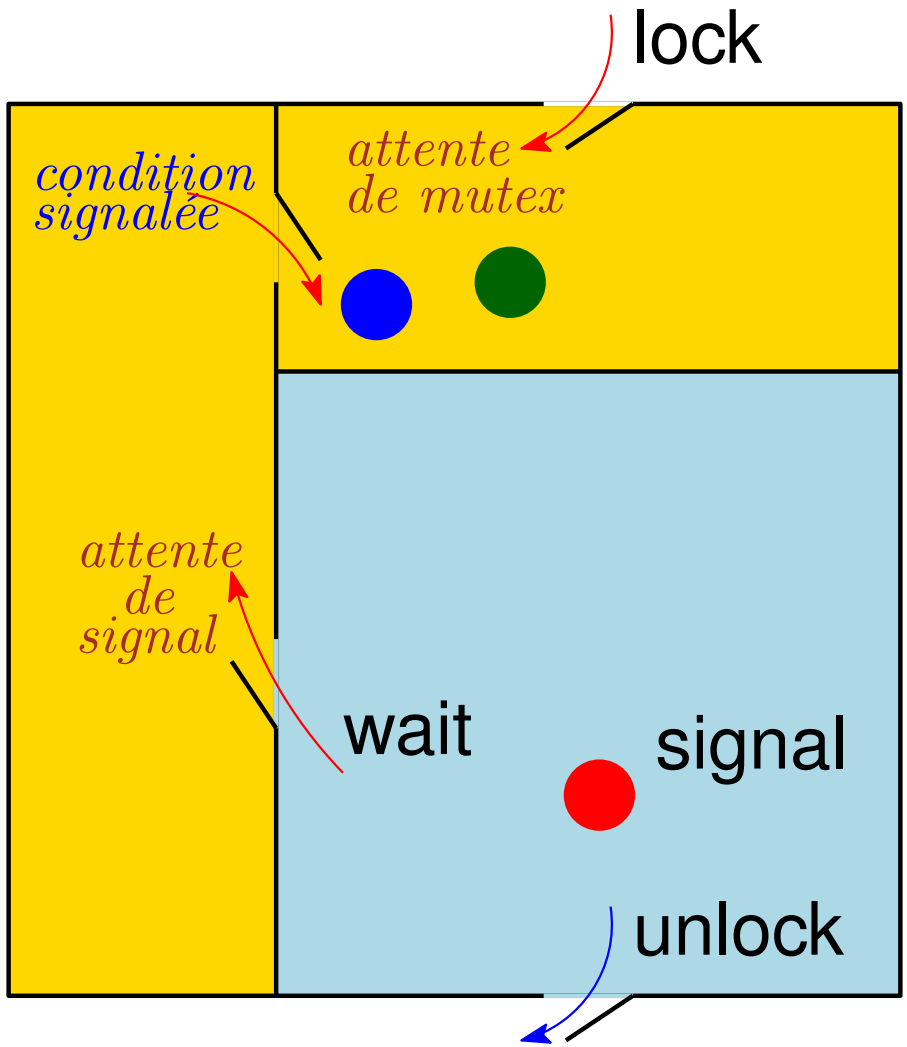
Variables de condition



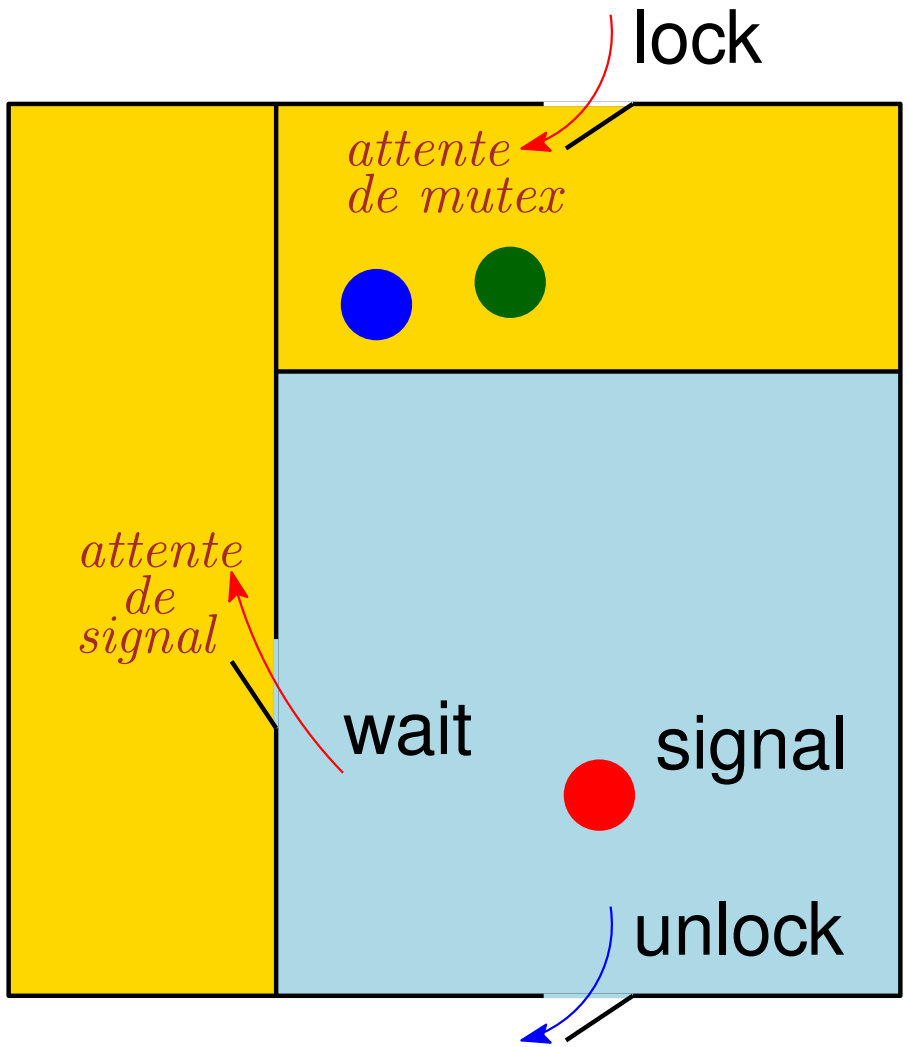
Variables de condition



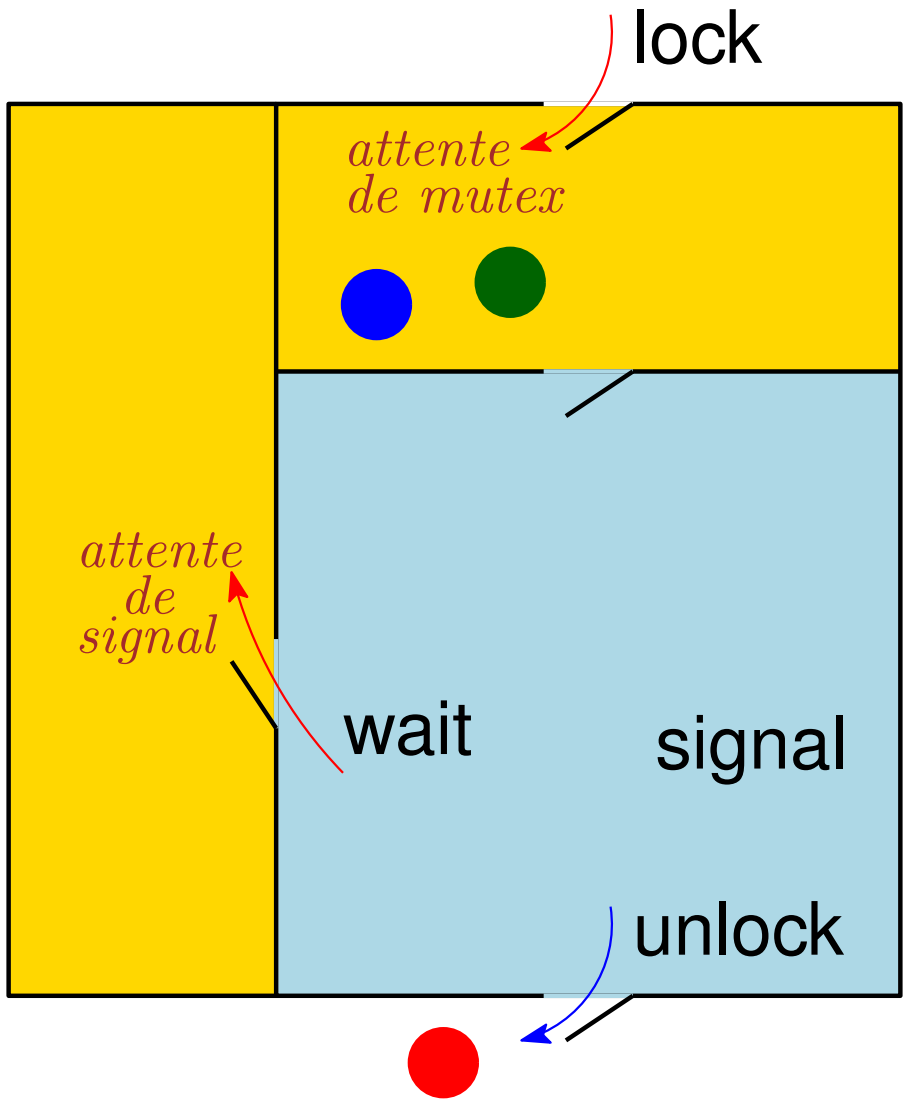
Variables de condition



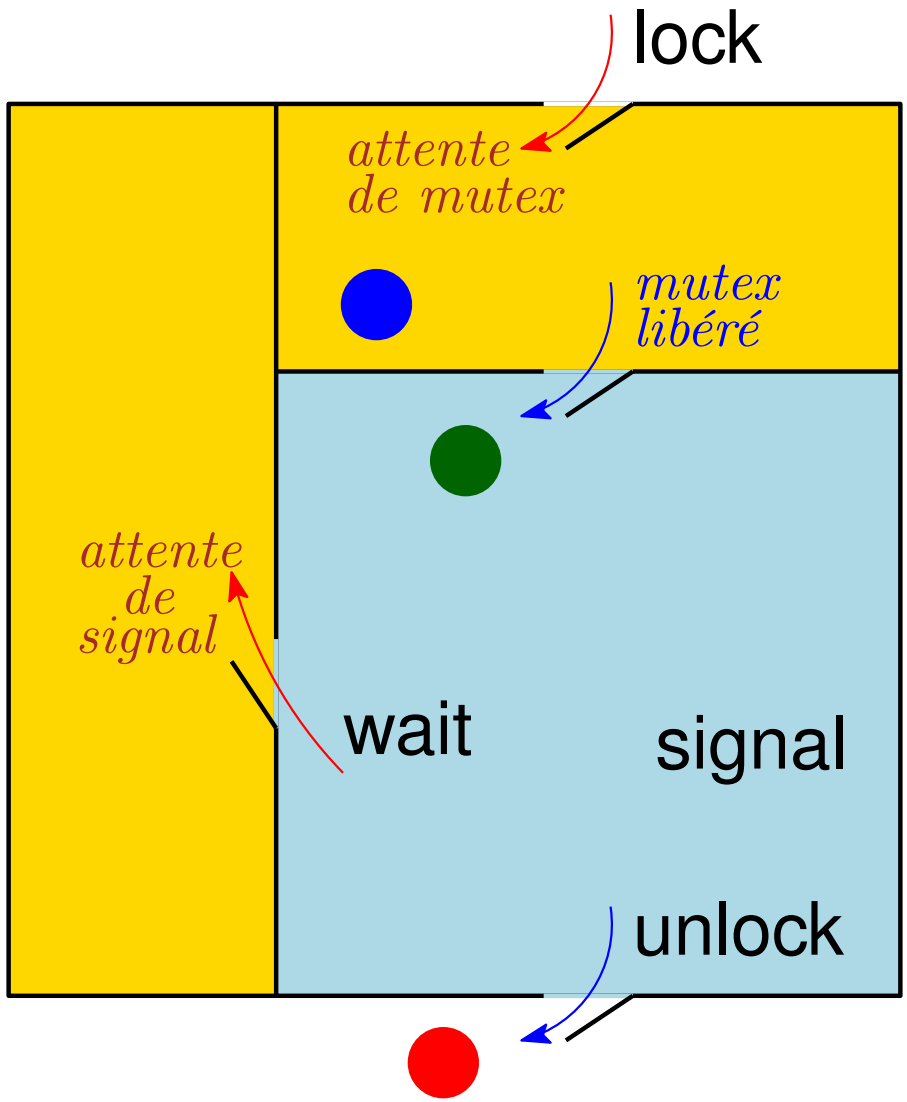
Variables de condition



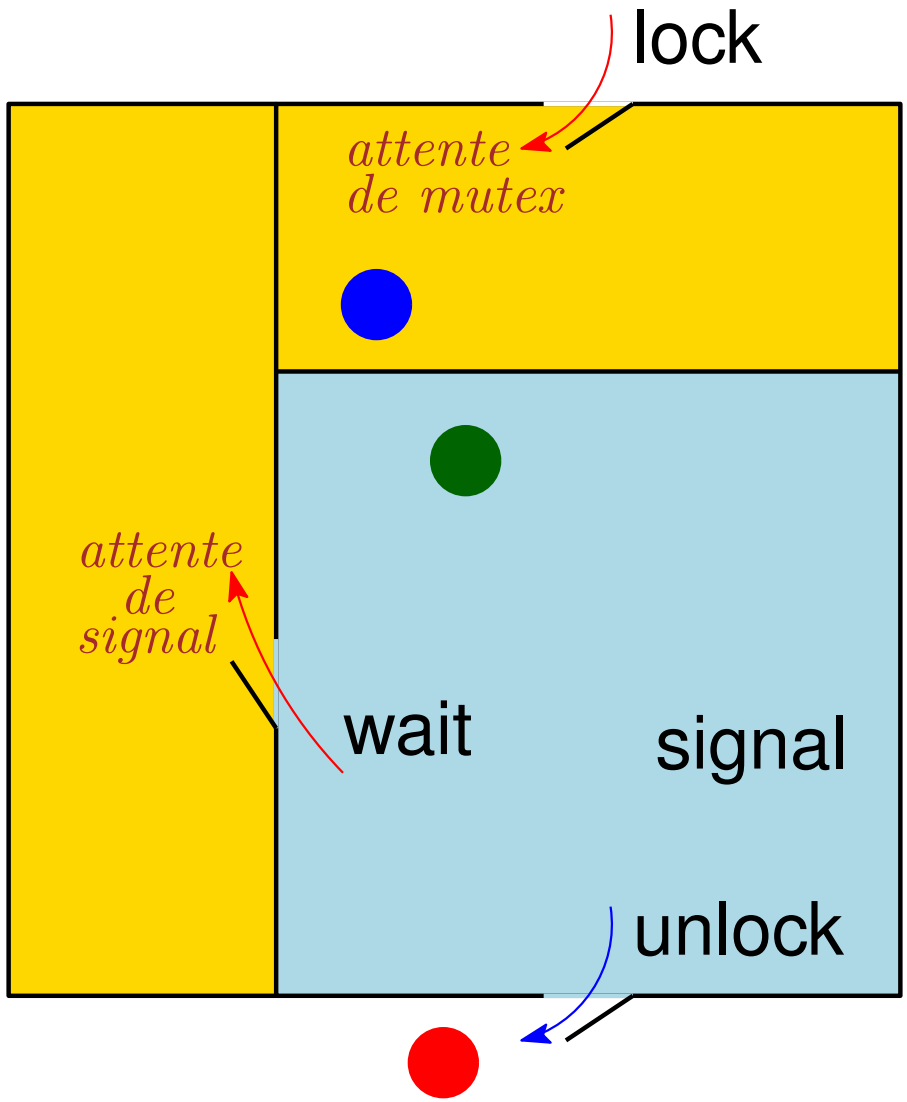
Variables de condition



Variables de condition



Variables de condition



Initialisation et destruction de variables de condition

```
int pthread_cond_init(pthread_cond_t * cond, NULL);
```

Le 2me argument donne les attributs de la condition (NULL par défaut)

```
int pthread_cond_destroy(pthread_cond_t * cond);
```

Attention : aucun thread ne doit attendre sur `cond`

Renvoient 0 au succès, sinon un code d'erreur non-nul

Attente sur une variable de condition

```
int pthread_cond_wait(pthread_cond_t  * cond,
                      pthread_mutex_t * mutex);
```

Attend sur la variable de condition

Attention : `mutex` doit être verrouillé par le thread appelant

- déverrouille `mutex`
- passe en état d'attente jusqu'à une notification sur `cond`
- une fois la notification reçu, reprend `mutex`
- une fois `mutex` repris, retourne 0

Attention : aucun thread ne doit attendre sur `cond` avec un autre mutex

Notification sur une variable de condition

```
int pthread_cond_signal(pthread_cond_t * cond);
```

Notifie un thread en attente sur cond

```
int pthread_cond_broadcast(pthread_cond_t * cond);
```

Notifie tous les threads en attente sur cond

Ne produisent aucun effet s'il n'y a pas de threads en attente

Renvoient 0 au succès, sinon un code d'erreur non-nul

Sémaphore

```
pthread_mutex_t mutex;  // le mutex du sémaphore
pthread_cond_t  cond;   // la condition du sémaphore
unsigned int    value;  // le nombre d'unités disponibles

void acquire(int units) {
  pthread_mutex_lock(&mutex);          // verrouiller le mutex
  while (units > value)                // tant que pas assez...
    pthread_cond_wait(&cond, &mutex); // ...attendre
  value -= units;                      // prendre
  pthread_mutex_unlock(&mutex); }      // libérer le mutex

void release(int units) {
  pthread_mutex_lock(&mutex);          // verrouiller le mutex
  value += units;                      // déposer
  pthread_cond_broadcast(&cond);       // avertir tout le monde
  pthread_mutex_unlock(&mutex); }      // libérer le mutex
```

Attention : toujours refaire le test au retour dans la section critique

Variables de condition