Condition de concurrence : Deux processus manipulent des données partagées et le résultat final dépend de l'ordonnancement.
Section critique : Partie du code dans laquelle deux proc ne doivent jamais se trouver en même temps.
4 règles d'or : Pas en même temps en SC — pas de supposition sur la vitesse, le nb de proc — se dépêcher de quitter la SC — jamais attendre indéfiniment l'accès à une SC.
Fork : Crée new entrée ds table des proc. — copie espace d'adressage — copie descripteurs de fichiers — 0 = fils, pidfils = père — père attend fils par wait ou waitpid.
Thread : Nouvelle pile d'exécution.
Test and modify : Ne marche pas **Attente active :** Var verrou a val différente selon le proc. Pas bien
Producteur/Consommateur : Situation classique (ex : spool d'impression)
Solution naïve :

Producteur :
```
TantQue VRAI faire

  item <- Produire_item()

  TantQue nb_lib == 0 faire
    attendre
  FinTantQue

  nb_lib--
  Tab[lib] <- item
  lib++ (mod N)
  nb_occ++

FinTantQue
```

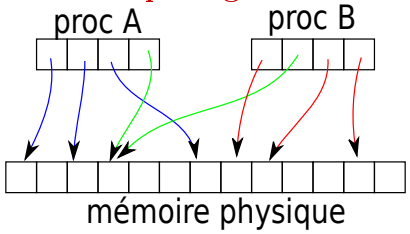
Consommateur :
```
TantQue VRAI faire

  TantQue nb_occ == 0 faire
    attendre
  FinTantQue

  nb_occ--
  item <- Tab[occ]
  occ-- (mod N)
  nb_lib++
  consommer_item(item)

FinTantQue
```

Sémaphore : Un entier + fonctions POST(débloque ou incrémente) et WAIT(bloque ou décrémente) + file d'attente de threads bloqués.

Producteur :
```
Tant Que VRAI Faire

  item <- Produire_item()

  WAIT(nb_lib) //si nb_lib = 0 on bloque,
  sinon on le décrémente et on avance
  WAIT(excl) //début section critique
    Tab[lib] <- item
    lib++ (mod N)
  POST(excl) //fin section critique

  POST(nb_occ) //On débloque un consommateur

FinTantQue
```

Consommateur :
```
Tant Que VRAI Faire

  WAIT(nb_occ)

  WAIT(excl)
    item <- Tab[occ]
    occ++ (mod N)
  POST(excl)

  POST(nb_lib)
  consommer_item(item)

FinTantQue
```

Sémaphore nommé : Fichier virtuel utilisable par tous processus.
Sémaphore non nommé : Variable du processus, utilisable que par les threads.
Mémoire partagée :

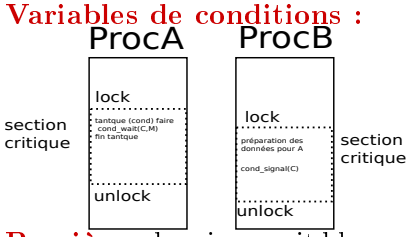


Utilisation : Fichier virtuel

- ouvre
- redimensionne(1x)
- projette
- détruit(1x)

Fork : Le père et le fils partagent la mémoire

Variables de conditions :

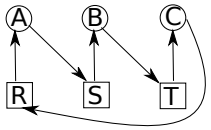


Explication : De manière atomique, on déverrouille le mutex et on passe en attente de la condition. Lorsque la condition est remplie, on passe de façon atomique à mutex verrouillé.

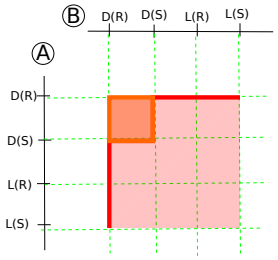
Barrière : barrier_wait bloque le thread jusqu'à ce que suffisament de processus aient atteint barrier_wait
Tubes nommés : mkfifo **Non nommés :** symbole pipe entre 2 proc ou pipe(df[2])
Resource : Objet physique ou virtuel pouvant être alloué à un seul processus à la fois.
Resources retirables : Peut être retirée au processus, utilisée par un autre, puis rendue au 1er sans compromettre son exécution (ex. CPU, RAM).
Resources non retirables : Ne peut être retirée au processus que lorsqu'il en aura décidé.

Famine : Situation où un processus demande l'accès à une ressource mais ne l'obtient jamais car elle est toujours attribuée à d'autres processus.
Solution : Le principe FIFO permet d'éviter la famine mais n'est pas toujours souhaitable.
Interblocage : Situation où un ensemble de processus attendent un événement que seul un des processus de l'ensemble peut provoquer.

Modèle de Holt :
Carré = ressource, Rond = processus



Etats surs : On appelle état sûr un état à partir duquel on peut exécuter tous les processus un après l'autre dans un certain ordre sans qu'aucun ne soit bloqué. Un interblocage survient toujours à partir d'un état non sûr. Pour déterminer les états sûrs, il faut connaitre à l'avance le code que vont exécuter les programmes.



Stratégies possibles pour les interblocages : Ignorer — Détecter et remédier (Holt) — Eviter (états sûrs) — Contraintes (pas d'exclusion mutuelle, ordonner les ressources).

Fork : pid_t fork(void)
pid_t waitpid(pid_t pid, int* status, int options) — pid_t wait(int *status)
Threads : int pthread_create(pthread_t * thread, NULL, (void*(*)(void *))f, (void*)data)
pthread_exit(void* retval) — pthread_join(pthread_t th, void **thread_return)
Sémaphores : int sem_post(sem_t *sem) — int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem) — int sem_getvalue(sem_t *sem, int *sval)
Non nommés : int sem_init(sem_t *sem, int pshared(0=threads seulement), int value)
sem_destroy(sem_t *sem)
Nommés : sem_t *sem_open(const char *name, int oflag(O_CREAT|O_EXCL), mode_t mode(S_IRWXU), int value) — sem_t *sem_open(const char *name, int oflag(O_RDWR))
sem_close(sem_t *sem) — sem_unlink(const char *name).
Mutex : pthread_mutex_init(pthread_mutex_t *mutex)
pthread_mutex_lock(pthread_mutex_t *mutex) — pthread_mutex_unlock(pthread_mutex_t *mutex)
pthread_mutex_trylock(pthread_mutex_t *mutex) — pthread_mutex_destroy(pthread_mutex_t *mutex)
Mémoire partagée : int shm_open(const char *nom, int oflag, mode_t mode)
ftruncate(int df, off_t length)
void *mmap(void *addr, size_t length, int prot(PROT_READ|PROT_WRITE), MAP_SHARED, int df, off_t offset)
int munmap(void *addr, size_t length) — int shm_unlink(const char *nom)
Variables de condition : int pthread_cond_init(pthread_cond_t *cond, NULL)
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_destroy(pthread_condèt *cond)
Barrière : int pthread_barrier_init(pthread_barrier_t *barrier, NULL, count)
int pthread_barrier_wait(pthread_barrier_t *barrier)
int pthread_barrier_destroy(pthread_barrier_t *barrier)
Tube : ssize_t read(int df, void *buf, size_t count)
ssize_t write(int fd, void *buf, size_t count)
Nommé : int mkfifo(const char *pathname, mode_t mode) **Non nommé :** int pipa(int pipefd[2])