Chapitre 03 : Synchronisation, Inter-blocage et Communication de Processus

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme D'Ingénieur de Conception (DIC, 1^{e} année)
en Informatique / Télécommunications-Réseaux
Licence Professionnelle
en Génie Logiciel et Systèmes d'Information (GLSI)

ECOLE SUPERIEURE POLYTECHNIQUE
www.esp.sn

Plan du Chapitre

Sommaire

Accès concurrents

- ☛ Le problème des accès concurrents se produit quand deux processus partagent une ressource, matérielle ou logicielle, alors que celle-ce ne peut pas être partagée (accès exclusif).
  - ➠ L'accès en lecture ne pose de problème contrairement aux accès en écriture

Problèmes de la vie courante

- Les problèmes de synchronisation sont légions dans la vie courante :
  1. Quand on crédite son compte en banque, ce crédit ne doit pas être perdu parce qu'en parallèle la banque débite un chèque
  2. Un parking de capacité N places ne doit pas laisser entrer $N+1$ véhicules
  3. Roméo et Juliette ne peuvent se prendre par la main que s'ils se retrouvent à leur rendez-vous
  4. Moussa et Doudou font la vaisselle. Moussa lave. Doudou essuie. L'égouttoir les synchronise.
  5. Certaines lignes de train possèdent des sections de voie unique. Sur ces sections, on ne peut avoir que des trains circulant dans un même sens à un instant donné
- On rencontre des problèmes similaires lorsqu'on utilise un système d'exploitation

Correspondance problèmes vie courante/informatique

- **Banque** $\Rightarrow$ Problème d'exclusion mutuelle : une ressource ne doit être accessible que par une entité à un instant donné. Cas, par exemple, d'une zone mémoire contenant le solde d'un compte.
- **Parking** $\Rightarrow$ Problème de cohorte : un groupe de taille bornée est autorisé à offrir/utiliser un service. Cas, par exemple, d'un serveur de connexions Internet qui ne doit pas autoriser plus de N connexions en parallèle.
- **Roméo et Juliette** $\Rightarrow$ Problème de passage de témoin : on divise le travail entre des processus. Cas, par exemple, de 2 processus qui doivent s'échanger des informations à un moment donné de leur exécution avant de continuer.

Correspondance problèmes vie courante/informatique

- **Moussa et Doudou** $\Rightarrow$ Problème de producteurs/consommateurs : un consommateur ne peut consommer que si tous les producteurs ont fait leur travail. Cas, par exemple, d'un processus chargé d'envoyer des tampons qui ont été remplis par d'autres processus
- **Train** $\Rightarrow$ Problème de lecteurs/rédacteurs où l'on doit gérer, de manière cohérente, une compétition entre différentes catégories d'entités. Cas, par exemple, d'une tache de fond périodique de « nettoyage » qui ne peut se déclencher que quand les taches principales sont inactives

Généralités [E. W. Dijkstra, 1965]

- Sémaphore = objet composé
  1. D'une variable (sa valeur)
  2. D'une file d'attente (les processus bloqués)
- Primitives associés :
  1. Initialisation (avec une valeur positive ou nulle)
  2. Manipulation :
     - Prise (P ou Wait) = demande d'autorisation
     - Validation (V ou Signal) = fin d'utilisation
- Principe : sémaphore associé à une ressource
  1. Prise = demande d'autorisation (Puis-je ?)
     - si valeur > 0 accord, sinon blocage
  2. Validation = restitution d'autorisation (Vas-y)
     - si valeur < 0 déblocage d'un processus

Analogie

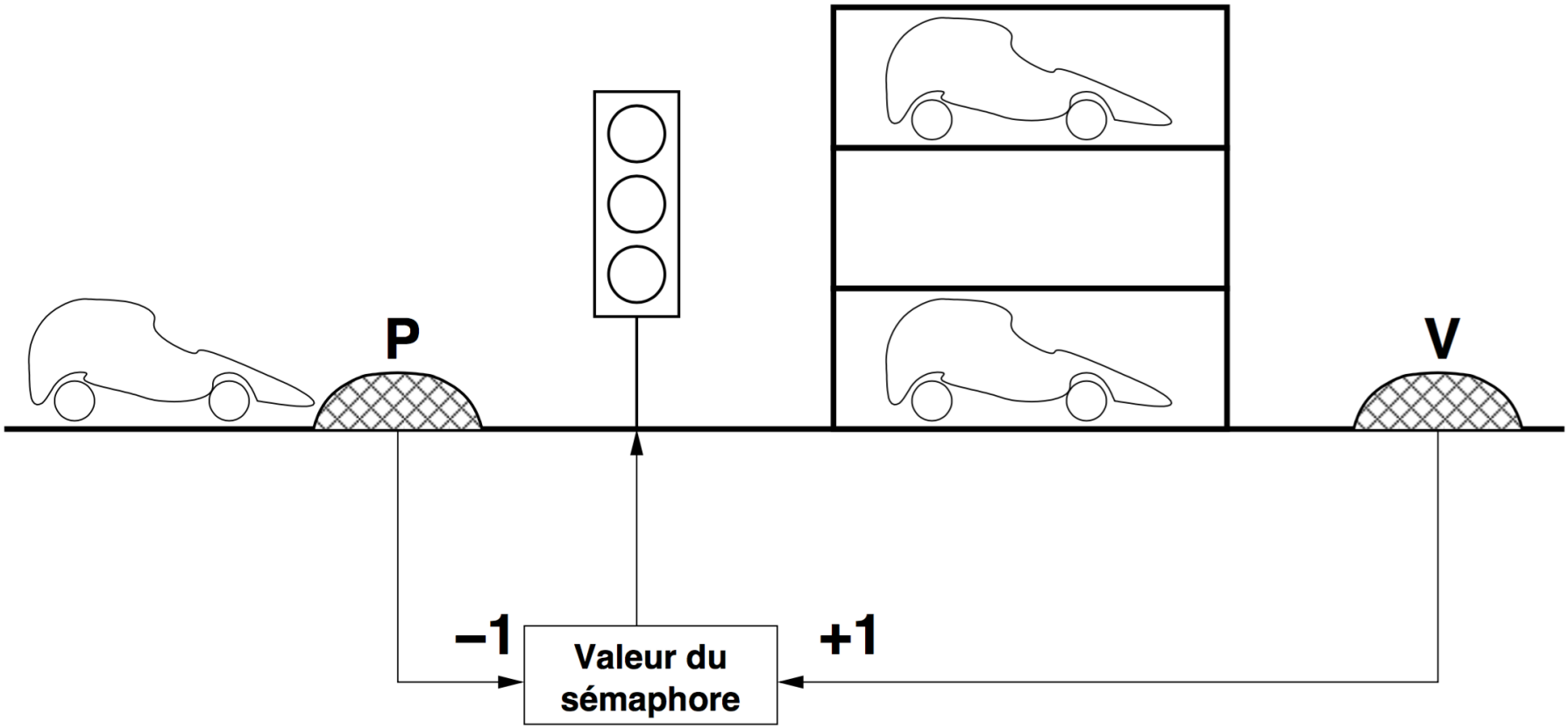


Figure: Parking de *N* places contrôlé par un feu

Manipulation des Sémaphores : Algorithmes P ET V

- ☛ Initialisation(sem,n)
- ☛ sem = n

P(sémaphore)

- sem = sem -1
  - si (sem < 0) alors
    1. étatProcessus = Bloqué
    2. mettre processus en file d'attente
  - finSi
  - invoquer l'ordonnanceur

V(sémaphore)

- sem = sem +1
- Si (sem <= 0) alors
  1. extraire processus de file d'attente
  2. étatProcessus = Prêt
- finSi
- invoquer l'ordonnanceur

Cas à étudier

1. Exclusion mutuelle
2. Cohorte
3. Passage de témoin
4. Producteurs/Consommateurs
5. Lecteurs/rédacteurs

Exclusion mutuelle : définition

Ressource critique et Section critique

- Une ressource est dite **ressource critique** lorsque des accès concurrents à cette ressources peuvent résulter dans un état incohérent
- On parle aussi de situation de compétition (race condition) pour décrire une situation dont l'issue dépend de l'ordre dans lequel les opérations sont effectuées
- Une section critique est une section de programme manipulant une ressource critique.
- **Section critique** = partie du processus où il peut y avoir un conflit d'accès
  - Contient des variables et/ou ressources partagées

Exclusion mutuelle : définition

Définition

☛ **Exclusion mutuelle** : si une ressource a été accédée par un processus P_1, aucun autre processus ne peut y accéder tant qu'elle n'a pas été libérée par P_1.

- Si P_1 est interrompu, il faut attendre à ce qu'il reprenne son exécution pour qu'il puisse libérer la ressource
- Une section de programme est dite atomique lorsqu'elle ne peut pas être interrompue par un autre processus manipulant les mêmes ressources critiques.
- C'est donc une atomicité relative à la ressource
- Un mécanisme d'exclusion mutuelle sert à assurer l'atomicité des sections critiques relatives à une ressource critique

Principe de l'exclusion mutuelle

- Gestion d'accès concurrent à des ressources partagées
  1. Sémaphore **mutex** initialisé à 1
  2. Primitive *P* en début de section critique
  3. Primitive *V* en fin de section critique

- Sémaphore mutex initialisé à 1

Prog1

- P(mutex)
  - x=lire (cpte)
  - x = x + 10
  - écrire (cpte=x)
- V(mutex)

Prog2

- P(mutex)
  - x=lire (cpte)
  - x = x - 100
  - écrire (cpte=x)
- V(mutex)

Principe de l'exclusion mutuelle

☛ Sémaphore mutex initialisé à 1

Prog1

- P(mutex)
  - x=lire (cpte)
  - x = x + 10
  - écrire (cpte=x)
- V(mutex)

Prog2

- P(mutex)
  - x=lire (cpte)
  - x = x - 100
  - écrire (cpte=x)
- V(mutex)

1. Quand Prog1 s'exécute sans que Prog2 soit activé, alors :
   ☛ P(mutex) fait passer mutex à 0 et Prog1 entre en section critique.
2. Si Prog2 est activé quand Prog1 est en section critique, alors
   ☛ P(mutex) fait passer mutex à -1.
   ☛ Donc Prog2 se retrouve bloqué. Quand Prog1 exécute V(mutex), alors mutex passe à 0 et Prog2 est débloqué.

Principe du Cohorte

- Permet la coopération d'un groupe de taille maximum donnée
  1. Sémaphore **parking** initialisé à N
  2. Primitive P en début de besoin
  3. Primitive V en fin de besoin
- Sémaphore parking initialisé à N

Programme Véhicule

- Programme véhicule
  - ...
  - P(parking)
  - | ...
  - V(parking)
  - ...

- Le $(N+1)$ véhicule qui cherche à entrer dans le parking se retrouve bloque par son P(parking).
- Il n'est débloqué que quand un autre véhicule sort en faisant V(parking)

Principe du passage de témoin

☛ Il existe trois types de synchronisation par passage de témoin :

1. Envoi de signal
2. Rendez-vous entre 2 processus
3. Appel procédurale entre processus

Passage de témoin : Envoi de signal

- Permet à un processus d'envoyer un « top » à un autre pour signaler la présence/disponibilité d'une information
- Principe :
  1. Sémaphore top initialisé à 0
  2. Primitive P pour attente « top »
  3. Primitive V pour signal « top »

- Sémaphore top initialisé à 0

Prog1

- ...
- calcul(info)
- V(top)
- ...

Prog2

- ...
- P(top)
- utilisation(info)
- ...

Passage de témoin : Envoi de signal

☛ Sémaphore top initialisé à 0

Prog1

- ...
- calcul(info)
- V(top)
- ...

Prog2

- ...
- P(top)
- utilisation(info)
- ...

☛ Ce mécanisme peut être utilisé par *Prog*1 pour donner à *Prog*2 le droit d'accès à une ressource quand *Prog*1 estime que cette ressource est prête

Passage de témoin : Rendez-vous entre deux processus

☛ Permet à deux processus d'établir un point de synchronisation

1. Sémaphore Roméo initialisé à 0
2. Sémaphore Juliette initialisé à 0

Prog1

- ...
- V(juliette)
- P(romeo)
- ...

Prog2

- ...
- V(romeo)
- P(juliette)
- ...

☛ Ce patron permet à Prog1 et Prog2 de s'échanger des informations à un moment précis de leur exécution avant de continuer.

Passage de témoin : Rendez-vous entre N processus

☛ Généralisation du principe d'un rendez-vous entre N processus

Initialisation

- Sémaphore **rdv** initialisé à 0
- Sémaphore **mutex** initialisé à 1
- Entier **nbAttente** initialisé à 0

- NB : cet algorithme ne fonctionne qu'une seule fois

Exécution de l'algorithme

- ...
- P(mutex)
- Si (nbAttente < N - 1) alors
  - nbAttente = nbAttente + 1
  - V(mutex)
  - P(rdv)
- Sinon
  - V(mutex)
  - répéter (N-1) fois
    - V(rdv)
  - FinRépéter
- FinSi

Passage de témoin : Appel procédurale entre processus

☛ Permet à un processus de faire un « appel de procédure », alors que le code de cette procédure est localisé dans un autre processus

1. Sémaphore **appel** initialisé à 0
2. Sémaphore **retour** initialisé à 0

Serveur

- Répéter :
  - P(appel)
  - analyseParamAppel()
  - ...
  - préparationParamRetour()
  - V(retour)
- FinRépéter

Client

- ...
- preparationParamAppel()
- V(appel)
- P(retour)
- analyseParamRetour()
- ...

Passage de témoin : Appel procédurale entre processus

Serveur

- Répéter :
  - P(appel)
  - analyseParamAppel()
  - ...
  - préparationParamRetour()
  - V(retour)
- FinRépéter

Client

- ...
- preparationParamAppel()
- V(appel)
- P(retour)
- analyseParamRetour()
- ...

1. Serveur est le serveur d' « appel de procédure ». Il se met en attente d'un appel en faisant P(appel).
2. Client démarre. Il prépare ses paramètres d'appel en les mettant, par exemple, dans une zone de mémoire partagée.

Passage de témoin : Appel procédurale entre processus

Serveur

- Répéter :
  - P(appel)
  - analyseParamAppel()
  - ...
  - préparationParamRetour()
  - V(retour)
- FinRépéter

Client

- ...
- preparationParamAppel()
- V(appel)
- P(retour)
- analyseParamRetour()
- ...

3 Avec V(appel), le Client prévient Serveur de la disponibilité de ces informations

4 Le Serveur est alors réactivé. Il analyse les paramètres d'appel, effectue son traitement et stocke les paramètres de retour dans une (autre) zone de mémoire partagée

Passage de témoin : Appel procédurale entre processus

Serveur

- Répéter :
  - P(appel)
  - analyseParamAppel()
  - ...
  - préparationParamRetour()
  - V(retour)
- FinRépéter

Client

- ...
- preparationParamAppel()
- V(appel)
- P(retour)
- analyseParamRetour()
- ...

4 Le Serveur est alors réactivé. Il analyse les paramètres d'appel, effectue son traitement et stocke les paramètres de retour dans une autre zone de mémoire partagée

5 Client est réactivé, analyse les paramètres retour et poursuit son traitement

Producteurs/Consommateurs

- Objectif
- Principe
- Disposer et extraire
- Solution complète

Producteurs/Consommateurs : Objectif

☛ Permettre le contrôle de flux entre un (ou des) producteur(s) et un (ou des) consommateur(s) dans le cas où ils communiquent via un tampon mémoire de *N* cases

1. Exécution `Produc` : il produit `info0`

info0				

2. Exécution `Produc` : il produit `info1`

info0	info1			

3. Exécution `Conso` : il consomme `info0`

	info1			

4. Exécution `Produc` : il produit `info2`

	info1	info2		

Producteurs/Consommateurs : Principe

- ☛ Sémaphore **infoPrete** initialisé à 0
- ☛ Sémaphore **placeDispo** initialisé à N

Producteur

- Répéter
  - ...
  - calcul(info)
  - P(placeDispo)
  - Déposer(info)
  - V(infoPrete)
  - ...
- finRépéter

Consommateur

- Répéter
  - P(infoPrete)
  - extraire(info)
  - V(placeDispo)
  - utiliser(info)
- finRépéter

- ☛ Ici, le producteur Producteur peut écrire N informations avant d'être bloqueé (degré de liberté N).
- ☛ Chaque itération de Consommateur libère une case.

Producteurs/Consommateurs : Déposer/Extraire

- ☛ Le tampon ne peut s'étendre à l'infini
- ☛ Le tampon est géré de façon circulaire, c'est-à-dire que quand dernière case utilisée, retour à la première
- ☛ Il faut :
  1. un indice de dépôt iDépot
  2. un indice d'extraction iExtrait
  3. ajouter à l'initialisation iDépot = 0 et iExtrait = 0

déposer(info)

- tampon[iDépot] = info
- iDépot = (iDépot + 1) modulo N

extraire(info)

- info = tampon[iExtrait]
- iExtrait = (iExtrait + 1) modulo N

Producteurs/Consommateurs : cas de 2 producteurs

Solution complète

- Il faut une exclusion mutuelle sur déposer() et extraire() autour de iDépot et iExtrait
- Pour K producteurs :
  - Sémaphore mutex initialisé à 1
  - disposer(info)
    - P(mutex)
    - tampon[iDépot] = info
    - iDépot = (iDépot + 1) modulo N
    - V(mutex)
- Pour P consommateurs, idem pour extraire
- Pour K producteurs et P consommateurs, utiliser deux sémaphores mutexP et mutexC

Lecteurs/rédacteurs : Points abordés

- ☛ Objectif
- ☛ Solution de base
- ☛ Analyse
- ☛ Solution avec priorités égales

Lecteurs/rédacteurs : Objectif

☛ Permettre une compétition cohérente entre deux types de processus (les « lecteurs » et les « rédacteurs »)

1. Plusieurs lecteurs peuvent accéder simultanément à la ressource
2. Les rédacteurs sont exclusifs entre eux pour leur exploitation de la ressource
3. Un rédacteur est exclusif avec les lecteurs

Lecteurs/rédacteurs : Objectif

- Ce patron permet, par exemple, d'interdire à un processus d'écrire dans un fichier, si des processus sont en train de lire (simultanément) ce fichier
- De plus, deux processus ne peuvent écrire simultanément dans le fichier.
- En revanche, plusieurs processus peuvent lire simultanément le fichier.
- Le paradigme Lecteurs/Rédacteur permet de réaliser une exclusion mutuelle entre un groupe d'entités (les lecteurs) et une entité (le rédacteur).
- Il est généralisable à l'exclusion mutuelle entre deux groupes d'entités

Lecteurs/rédacteurs : Solution de base

- Sémaphore mutexG initialisé à 1
- Sémaphore mutexL initialisé à 1
- Entier NL initialisé à 0

Lecteurs/rédacteurs : Solution de base

Lecteur

- P(mutexL)
- NL = NL + 1
- si NL == 1 alors
  - P(mutexG)
- finSi
- V(mutexL)
- lectures()
- P(mutexL)
- NL = NL - 1
- si NL == 0 alors
  - V(mutexG)
- finSi
- V(mutexL)

Rédacteur

- P(mutexG)
- ecrituresEtLectures()
- V(mutexG)

Lecteurs/rédacteurs : Analyse

- La solution de base fonctionne, mais on constate qu'il y a une possibilité de famine pour les rédacteurs
- Pour éviter cette famine, on ajoute les contraintes suivantes
- Si la ressource est utilisée par un lecteur :
  1. Tout écrivain est mis en attente.
  2. Tout lecteur est accepté s'il n'y a pas d'écrivain en attente
  3. Tout lecteur est mis en attente s'il y a un écrivain en attente

Lecteurs/rédacteurs : Solution avec priorités égales

- Sémaphore mutexG initialisé à 1
- Sémaphore mutexL initialisé à 1
- Sémaphore fifo initialisé à 1
- Entier NL initialisé à 0

Lecteurs/rédacteurs : Solution avec priorités égales

Lecteur

- P(fifo)
  - P(mutexL)
    - NL = NL + 1
    - Si NL == 1 alors
    - P(mutexG)
    - finSi
    - V(mutexL)
  - V(fifo)
  - lectures()
  - P(mutexL)
    - NL = NL - 1
    - si NL == 0 alors
    - V(mutexG)
    - finSi
  - V(mutexL)

Rédacteur

- P(fifo)
  - P(mutexG)
- V(fifo)
- ecrituresEtLectures()
- V(mutexG)

Sommaire

Définitions

- ☛ Un ensemble de processus est en inter-blocage si chaque processus attend un événement que seul un autre processus de l'ensemble peut engendrer
- ☛ 2 (ou +) processus sont en attente de la libération d'une ressource attribuée à l'autre
- ☛ Un ensemble de processus est bloqué !

Vers un situation d'inter-blocage

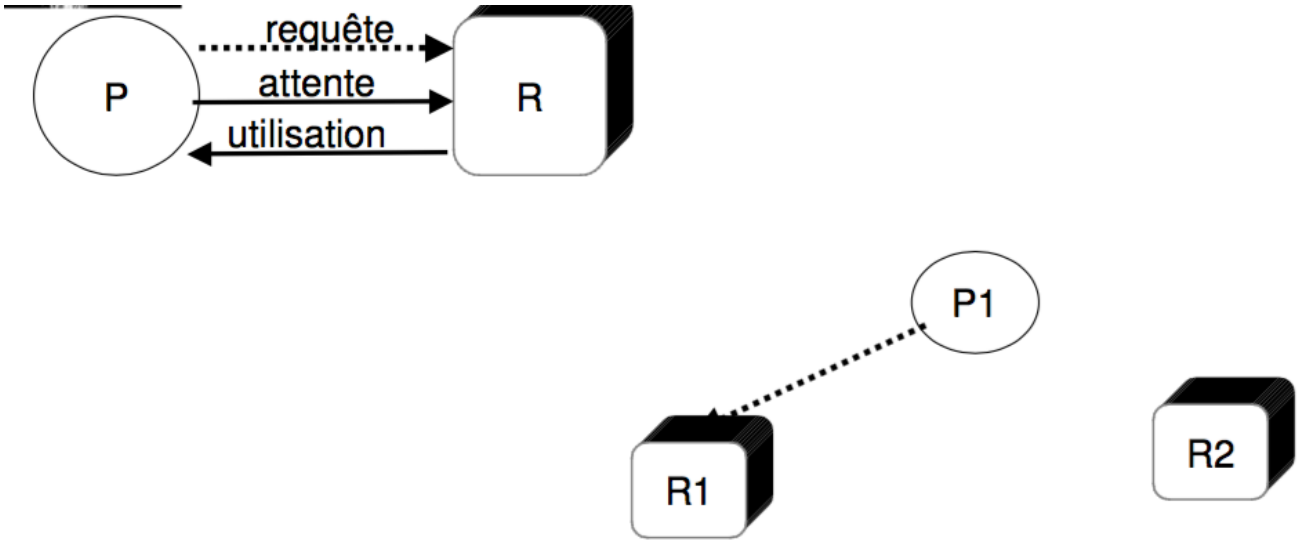


Figure: Vers un situation d'inter-blocage : 1

Vers un situation d'inter-blocage

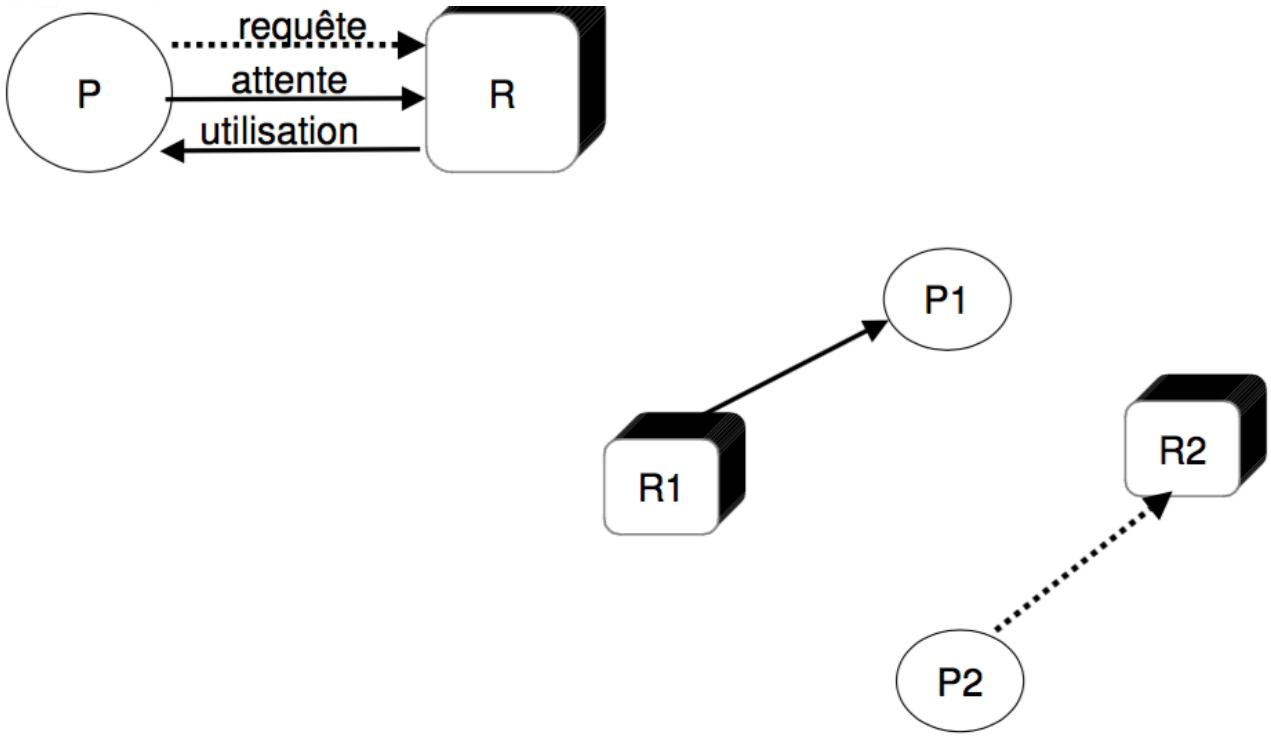


Figure: Vers un situation d'inter-blocage : 2

Vers un situation d'inter-blocage

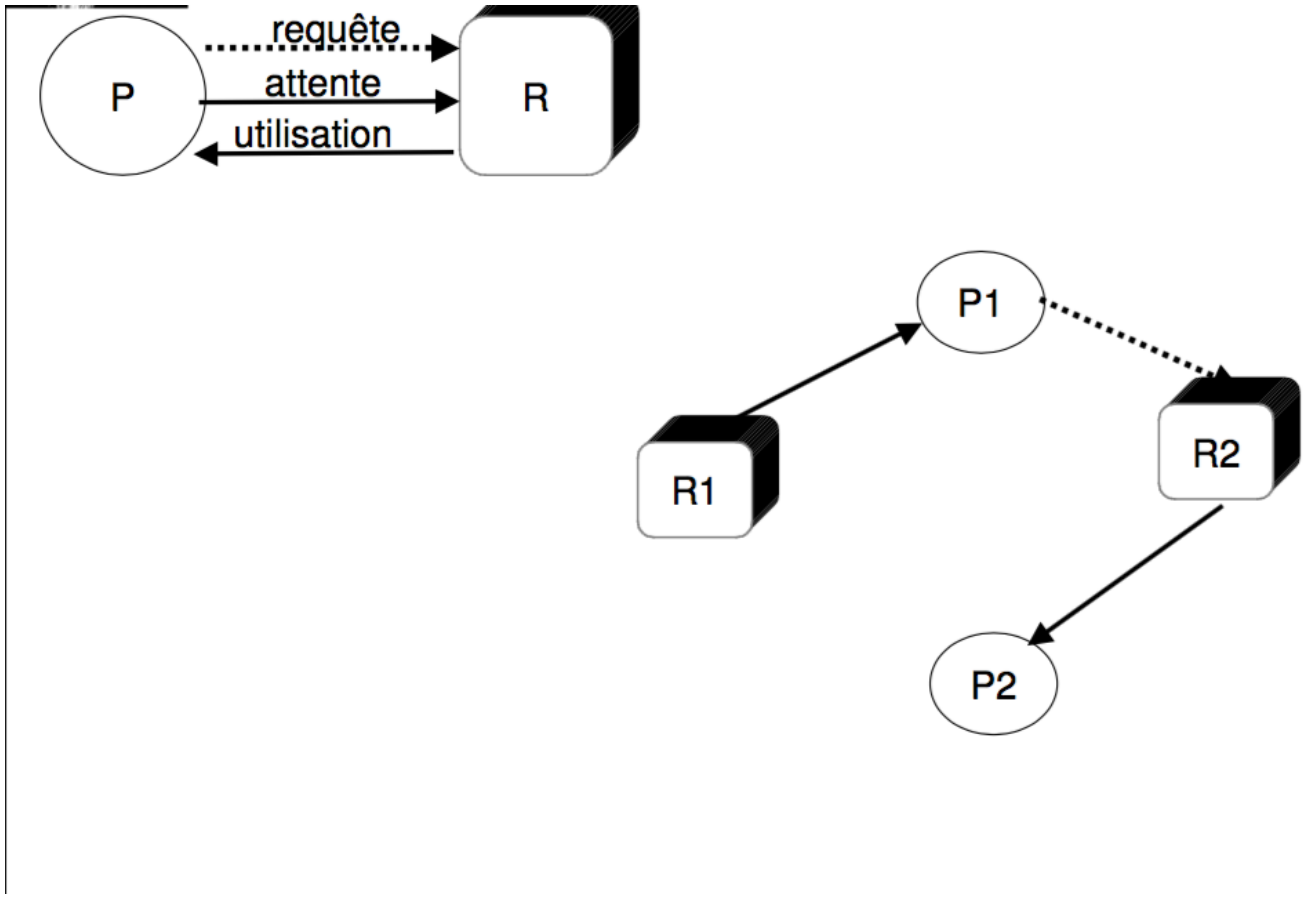


Figure: Vers un situation d'inter-blocage : 3

Vers un situation d'inter-blocage

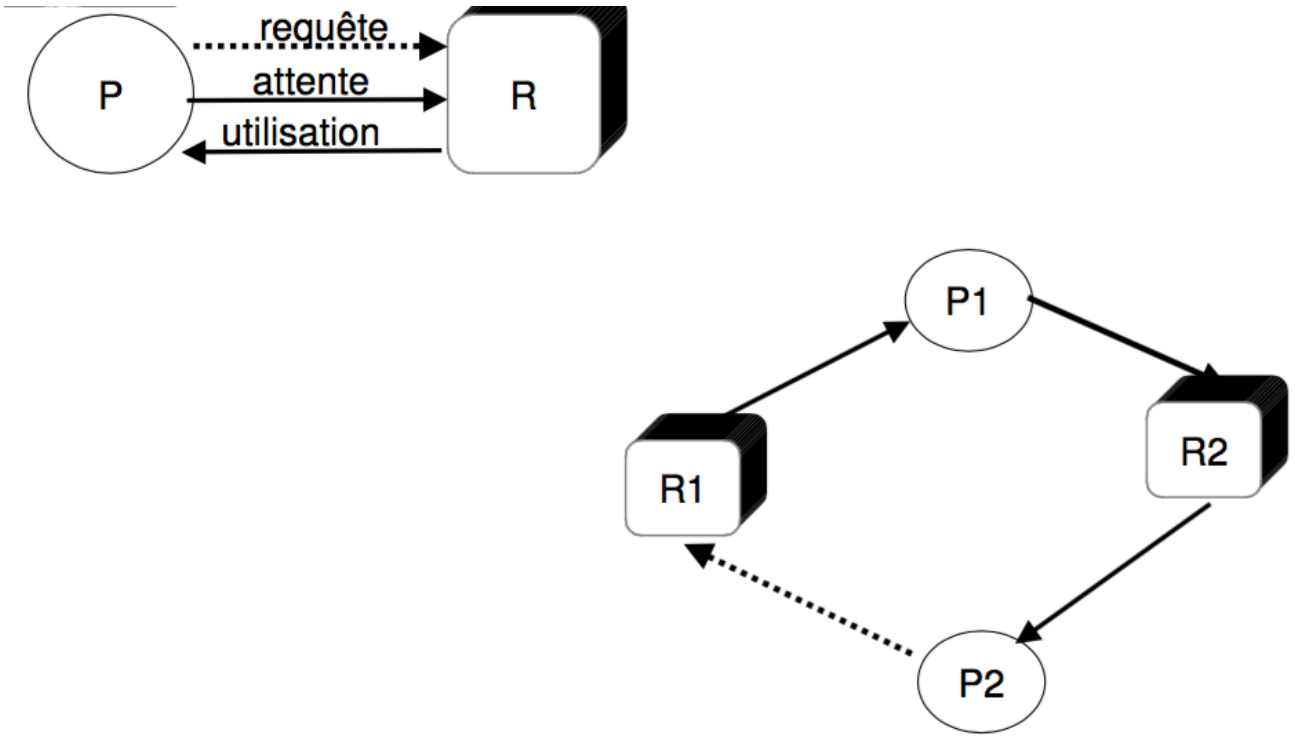


Figure: Vers un situation d'inter-blocage : 4

Vers un situation d'inter-blocage

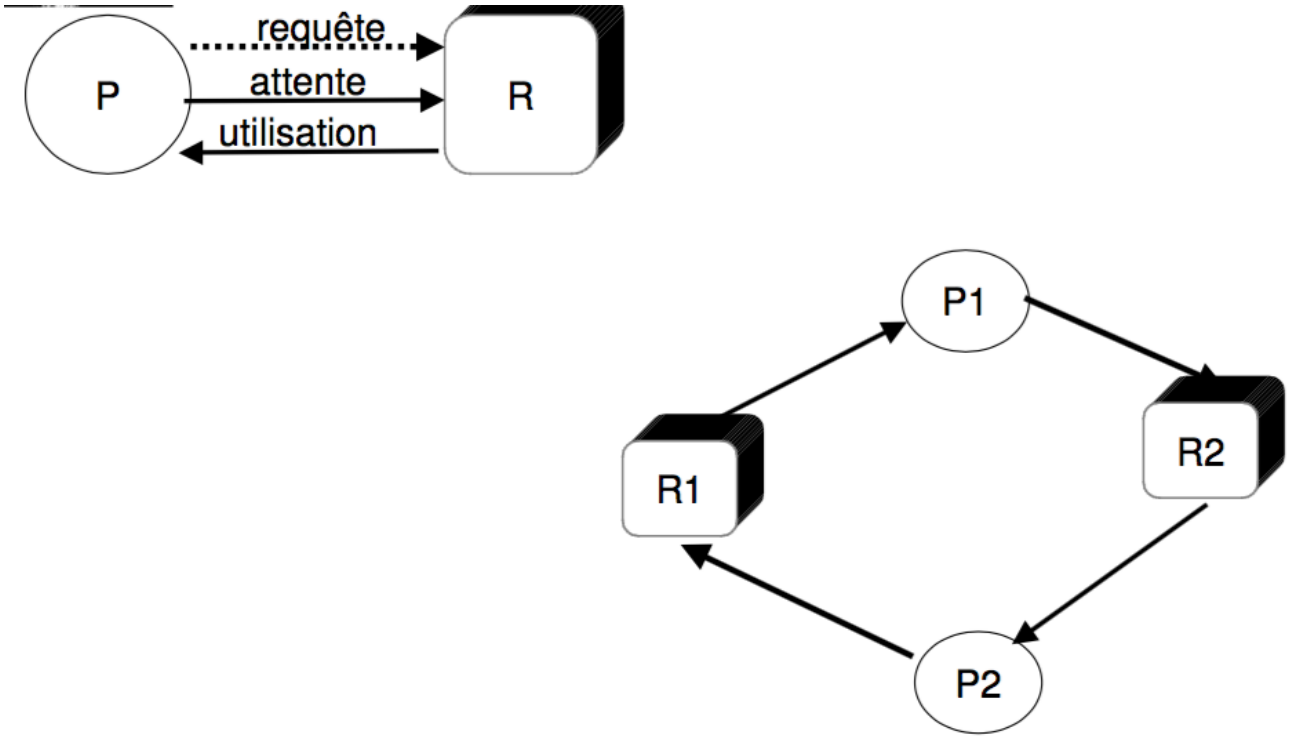


Figure: Vers un situation d'inter-blocage : 5

4 Conditions conduisant à un inter-blocage

1. Ressource limitée (exclusion mutuelle)
2. Attente circulaire : il doit y avoir au moins deux processus chacun attendant une ressource détenu par un autre
3. Occupation et attente : les processus qui détiennent des ressources peuvent en demander de nouvelles
4. Pas de réquisition(non préemption) : les ressources sont libérées par le processus

Stratégie de gestion

- ☛ Détecter et résoudre
- ☛ Prévenir la formation d'inter-blocages en empêchant l'apparition d'une des 4 conditions
- ☛ Éviter les inter-blocages en allouant de manière « intelligente » les ressources

Sommaire

Introduction

- Les processus coopèrent souvent pour traiter un même problème
- Ces processus s'exécutent en parallèle sur un même ordinateur (mono-processeur ou multiprocesseurs) ou bien sur des ordinateurs différents.
- Ils doivent alors s'échanger des informations : **communication inter-processus**
- Il existe plusieurs moyens de communication inter-processus :
  1. les signaux
  2. les messages :
     - les tubes de communication
     - les sockets

les signaux

- Les **interruptions logicielles** ou **signaux** sont utilisées par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important.
- De nombreuses erreurs détectées par le matériel, comme l'exécution d'une instruction non autorisée (une division par 0, par exemple) ou l'emploi d'une adresse non valide, sont converties en signaux qui sont envoyés au processus fautif.
- Ce mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée.
- Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal

les signaux

- ☛ On peut ainsi ignorer le signal, ou bien le prendre en compte, ou encore laisser le SE appliquer le comportement par défaut :
  - en général tuer le processus
- ☛ Certains signaux ne peuvent être ni ignorés, ni capturés
- ☛ Si un processus choisit de prendre en compte les signaux qu'il recroit, il doit alors spécifier la procédure de gestion de signal.
- ☛ Quand un signal arrive, la procédure associée est exécutée.
- ☛ A la fin de l'exécution de la procédure, le processus s'exécute à partir de instruction qui suit celle durant laquelle l'interruption a eu lieu

Synthèse

- ☛ Les signaux sont utilisés pour établir une communication minimale entre processus, une communication avec le monde extérieure et faire la gestion des erreurs.

les signaux

Décision prise à l'arrivée d'un signal

- Tous les signaux ont une routine de service, ou une action par défaut. Cette action peut être du type :
  - terminaison du processus
  - ignorer le signal
  - Créer un fichier *core*
  - Stopper le processus
  - La procédure de service ne peut pas être modifiée
  - Le signal ne peut pas être ignoré

Les messages

- Un autre mécanisme de communication entre processus est l'échange de **messages**
- Chaque message véhicule des données
- Un processus peut envoyer un message à un autre processus se trouvant sur la même machine ou sur des machines différentes
- Unix offre plusieurs mécanismes de communication pour l'envoie de messages : les tubes de communication sans nom, nommés et les sockets

Présentation générale

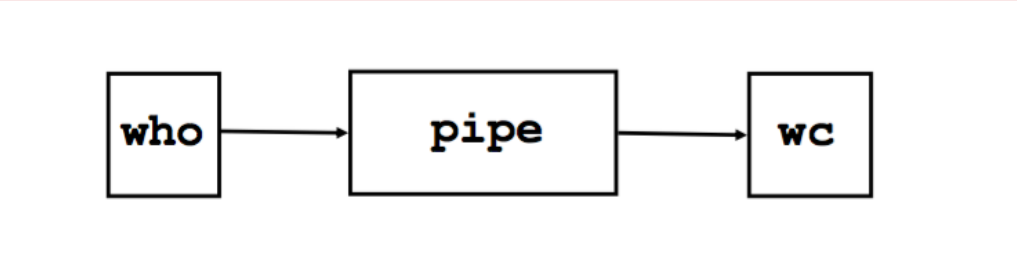
- Les tubes de communication ou pipes permettent à deux ou plusieurs processus d'échanger des informations.
- On distingue deux types de tubes :
  1. **tubes sans nom** (unamed pipe)
  2. **tubes nommés** (named pipe)

Les tubes sans nom

- Les tubes sans nom sont des liaisons de communication unidirectionnelles.
- La taille maximale d'un tube sans nom varie d'une version à une autre d'Unix, mais elle est approximativement égale à 4 Ko
- Les tubes sans nom peuvent être créés par le shell ou par l'appel système pipe()
- Les tubes sans nom du shell sont crées par l'opérateur binaire « | » :
  - la sortie standard d'un processus vers l'entrée standard d'un autre processus
  - Les tubes de communication du shell sont supportés par toutes les versions d'Unix

Les tubes sans nom : exemple

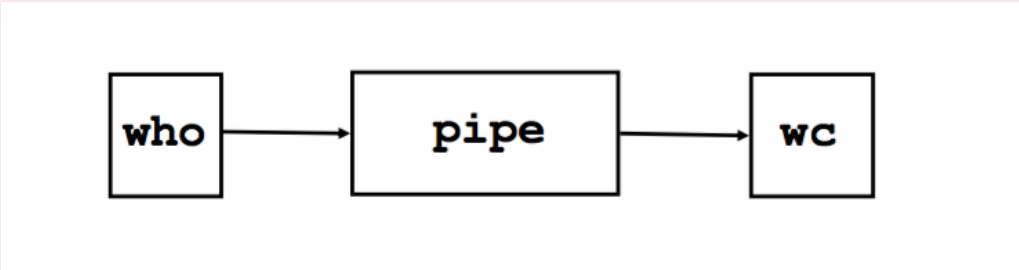
- Soit la commande suivante :
  - **who | wc -l**
- Elle crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe



1. Elle détermine le nombre d'utilisateurs connectés au système en appelant who, puis en comptant les lignes avec wc
2. Le premier processus réalise la commande who. Le second processus exécute la commande wc -l

Les tubes sans nom : exemple

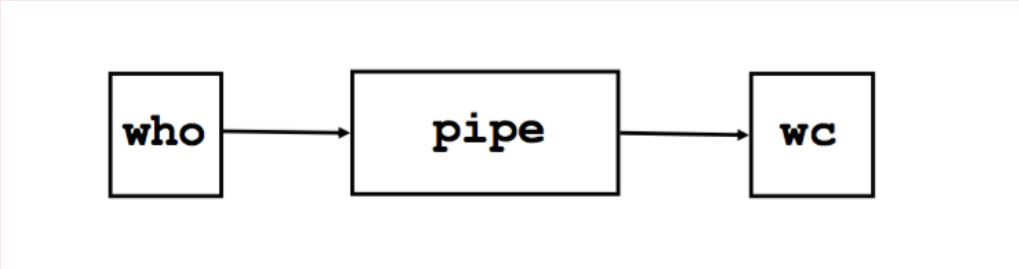
- Soit la commande suivante :
  - **who | wc -l**
- Elle crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe



4. Le processus réalisant la commande **who** ajoute dans le tube une ligne d"information par utilisateur du système
5. Le processus réalisant la commande **wc -l** récupère ces lignes d'information pour en calculer le nombre total
6. Le résultat est affiché à l'écran

Les tubes sans nom : exemple

- Soit la commande suivante :
  - **who | wc -l**
- Elle crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe

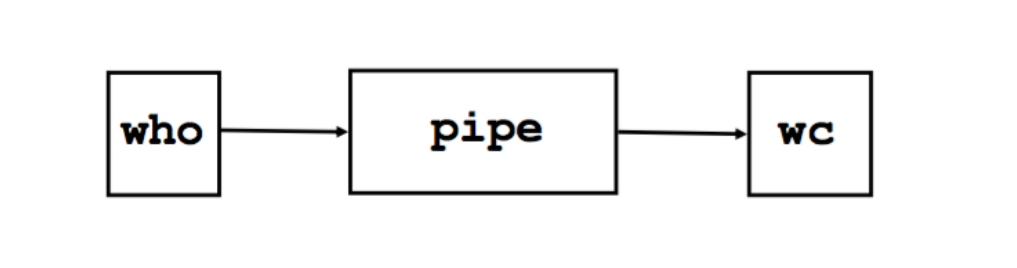


7 Les deux processus s'exécutent en parallèle, les sorties du premier processus sont stockées sur le tube de communication.

8 Lorsque le tube devient plein, le premier processus est sus- pendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker une ligne d'information

Les tubes sans nom : exemple

- Soit la commande suivante :
  - **who | wc -l**
- Elle crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe



8 Lorsque le tube devient plein, le premier processus est sus- pendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker une ligne d'information

9 De même, lorsque le tube devient vide, le second processus est suspendu jusqu'à ce qu'il y ait au moins une ligne d'information sur le tube.

Tubes de communication nommés

- Linux supporte un autre type de tubes de communication, beaucoup plus performants
- Il s'agit des tubes nommés (named pipes).
- Les tubes de communication nommés fonctionnent aussi comme des files du type FIFO (First In First Out)
- Ils sont plus intéressants que les tubes sans nom car ils offrent, en plus, les avantages suivants :
  1. Ils ont chacun un nom qui existe dans le système de fichiers (une entrée dans la Table des fichiers)
  2. Ils sont considérés comme des fichiers spéciaux.
  3. Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine
  4. Ils existeront jusqu'à ce qu'ils soient supprimés explicitement
  5. Ils sont de capacité plus grande, d'environ 40 Ko.

Sockets

- Les tubes de communication permettent d'établir une communication unidirectionnelle entre deux processus d'une même machine
- Le système Unix/Linux permet la communication entre processus s'exécutant sur des machines différentes au moyen de **sockets**.
- Les sockets ou prises sont les mécanismes traditionnels de communication inter-processus du système Unix
- Elles permettent la communication entre processus s'exécutant sur des machines différentes connectées par un réseau de communication.
- Par exemple, l'utilitaire **rlogin** qui permet à un utilisateur d'une machine de se connecter à une autre machine, est réalisé au moyen de sockets
- Les sockets sont également utilisé pour imprimer un fichier se trouvant sur une autre machine et pour transférer un fichier d'une machine à autre.

Sommaire

Les Appels systèmes

- Un appel système est un moyen de communiquer directement avec le noyau de la machine
- Le noyau regroupe toutes les opérations vitales de la machine
- Ainsi il est impossible d'écrire directement sur le disque dur
- L'utilisateur doit passer par des appels systèmes qui contrôlent les actions qu'il fait
- Ceci permet de garantir :
  1. la sécurité des données car le noyau interdira à un utilisateur d'ouvrir les fichiers auxquels il n'a pas accès
  2. l'intégrité des données sur le disque. Un utilisateur ne peut pas par mégarde effacer un secteur du disque ou modifier son contenu
- Ainsi, les appels système sont les fonctions permettant la communication avec le noyau
  - open, read, write, etc.

Les Appels système : utilisation

- Travaillent en relation directe avec le noyau
- Retournent un entier positif ou nul en cas de succès et -1 en cas d'échec
- Par défaut le noyau peut bloquer les appels systèmes et ainsi bloquer l'application si la fonctionnalité demandée ne peut pas être servie immédiatement
- Ne resservent pas de la mémoire dans le noyau.
- Les résultats sont obligatoirement stockés dans l'espace du processus (dans l'espace utilisateur)
- il faut prévoir cet espace par allocation de variable (statique, pile) ou de mémoire (malloc(). . .)

Chapitre 03 : Synchronisation, Inter-blocage et Communication de Processus

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme D'Ingénieur de Conception (DIC, 1^{e} année)
en Informatique / Télécommunications-Réseaux
Licence Professionnelle
en Génie Logiciel et Systèmes d'Information (GLSI)

ECOLE SUPERIEURE POLYTECHNIQUE
www.esp.sn