Entrées/sorties et multithreading : interruptions vs scrutations

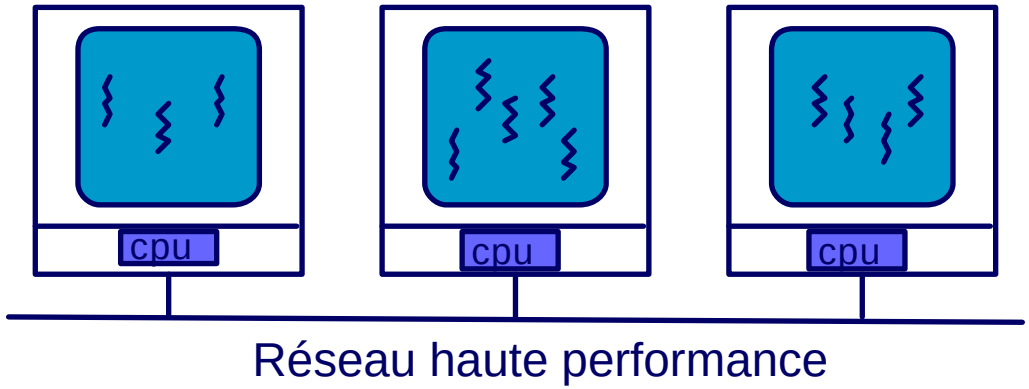
Vincent Danjean
ReMaP, LIP, ENS-Lyon
Email : Vincent.Danjean@ens-lyon.fr

Plan

- Contexte
- Problèmes abordés
  - Réactivité
  - Interruption / scrutation
- Contribution
  - Marcel
  - Activations
- Réactivité
- Travaux futurs

Contexte

- Environnements multithreads distribués
  - Outils de programmation des grappes/machines parallèles

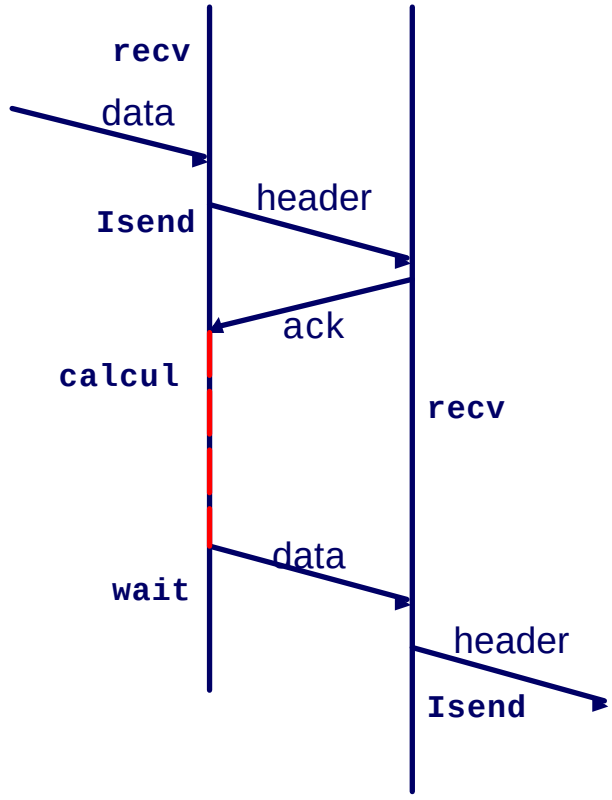


  - Efficacité
    - → Processeur/Bande passante réseau
  - Réactivité
    - → Temps de réponse réseaux courts
  - Athapascan , PM^2

Réactivité

- Permet de faire progresser le calcul
  - Cas des requètes asynchrones
  - Ex : MPI

```
If (mynode==0){
        send(mynode+1);
        calcul(1s);
        recv();
} else {
        recv();
        Isend(mynode+1,&req);
        calcul(1s);
        wait(req);
}
```



Entrées/sorties

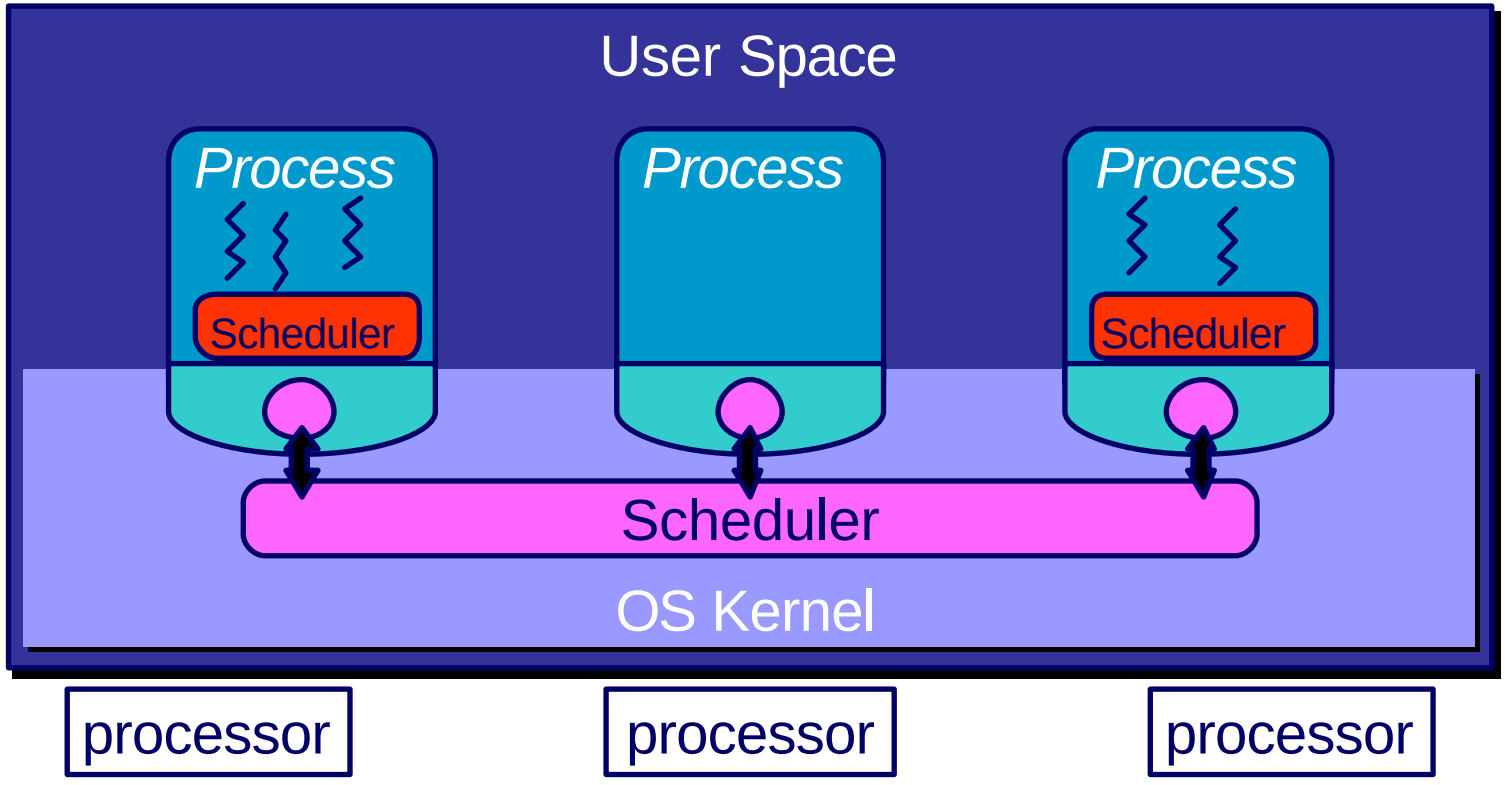
- Deux modes d'interaction pour les réseaux
  - Interruptions
    - → La carte réseau interrompt le processeur
    - → Le système d'exploitation réveille l'application
  - Scrutation
    - → Vérification périodique du réseau par l'application
    - → Compromis entre réactivité et surcharge
- Principes également valables pour les autres types d'entrées/sorties
    - → fichiers, ...

Les interruptions

- Pas toujours disponibles ou coûteuses
  - → SiSCI, BIP
- Pas toujours en mode utilisateur
  - Utilisation des exploitables signaux (SIG_IO)
  - Utilisation d'appels systèmes bloquants
- Environnements multithreadés
  - Threads de niveau noyau
    - → pertes de performances
    - → utilisation de l'ordonnanceur du noyau (peu flexible)

Les interruptions (2)

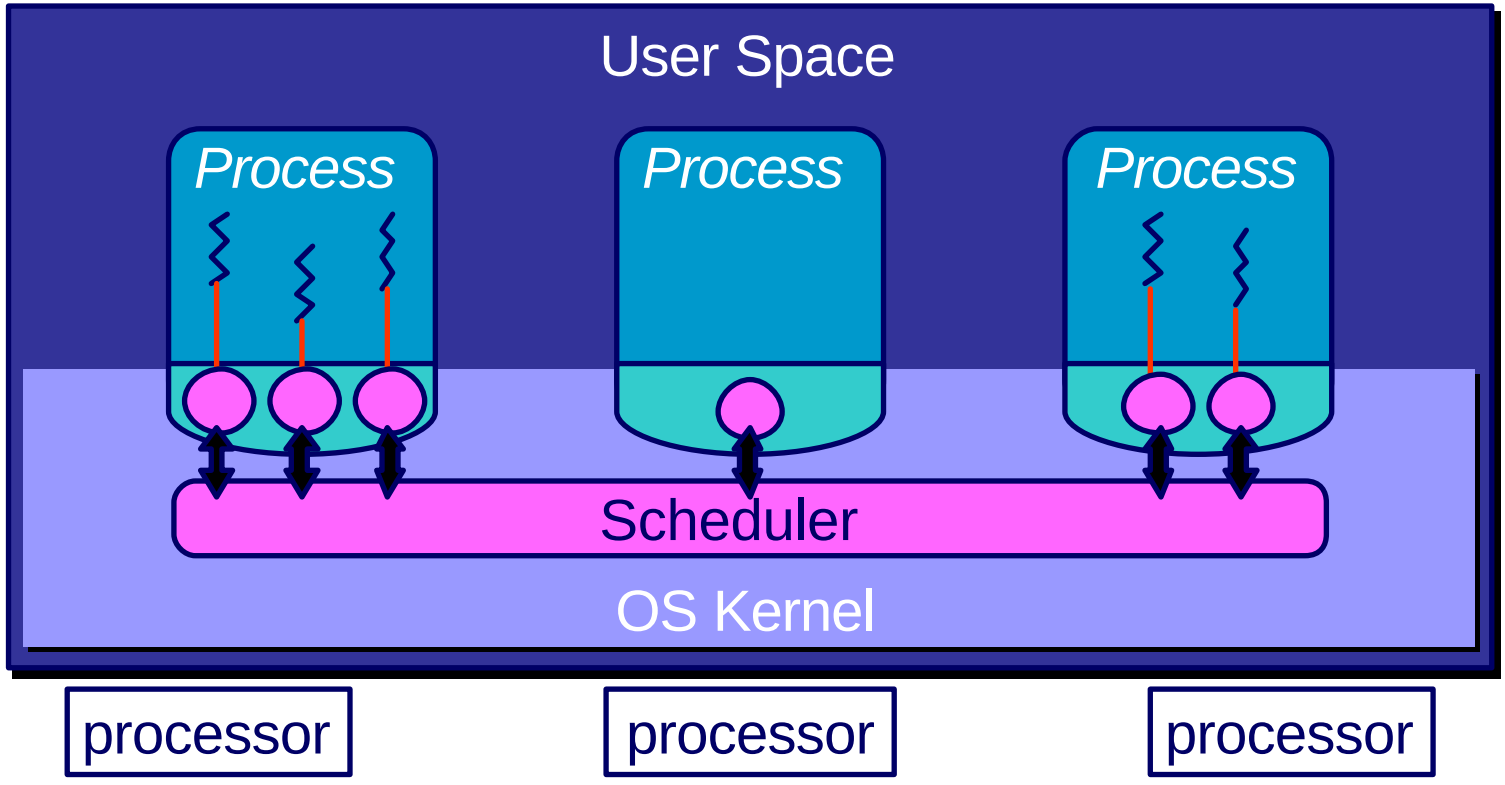
- Multithreading utilisateur



→ Efficacité mais appels systèmes bloquants mal gérés

Les interruptions (3)

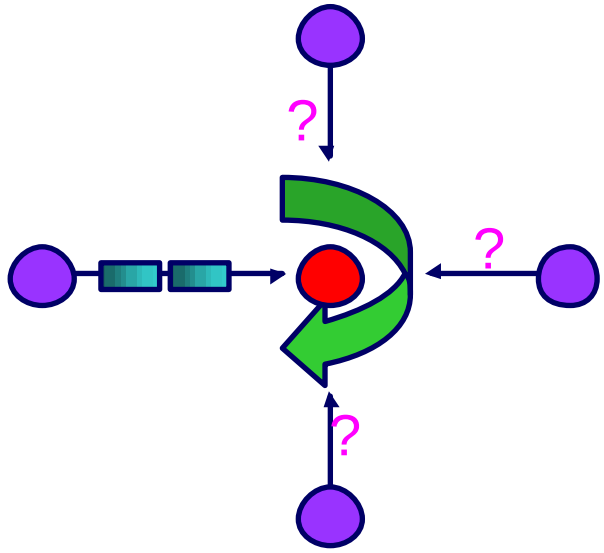
- Multithreading noyau



→ Appels systèmes bloquants gérés mais inefficacité

La scrutation

- La scrutation est
  - intéressante
    - → Efficacité « brute »
  - nécessaire
    - → API réseau ne fournissant pas d'appel bloquant
    - → Appels systèmes bloquants inexploitables (threads utilisateurs)



- Problèmes
  - → Fréquence difficile à assurer
  - → Coûteux en présence de multiple "*pollers* "
  - → Boucles actives consommatrices de CPU

Proposition

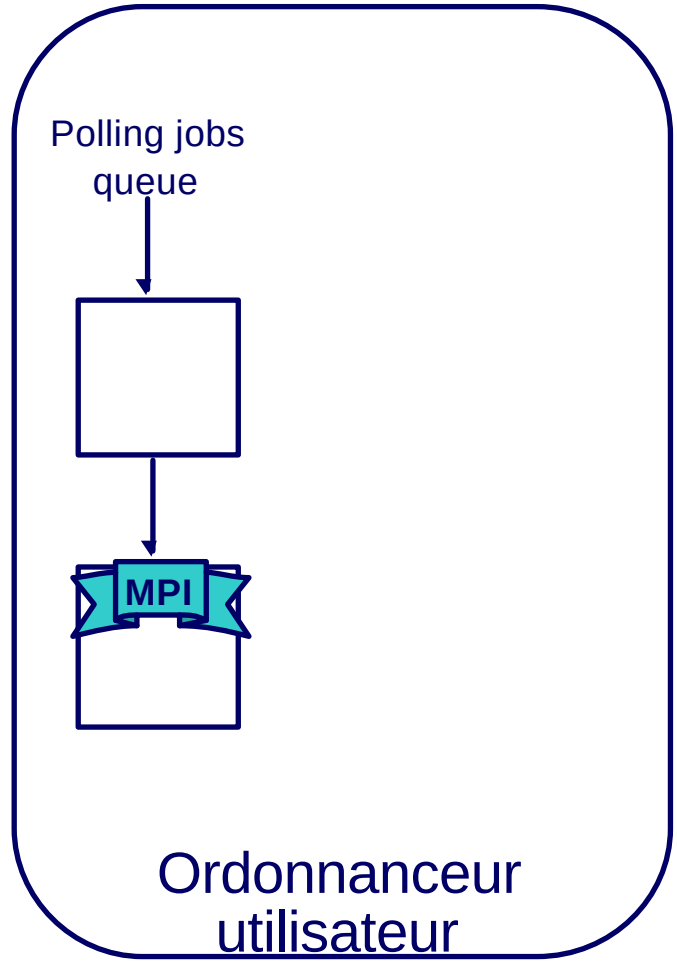
- Une interface unifiée pour la récupération des événements (réseaux)
  - Mécanisme unifiant scrutations et interruptions
  - Extension des fonctionnalités de l'ordonnanceur et de l'OS
- Prototype
  - PM^2/Marcel
  - Linux

Extensions de l'ordonnanceur

- Marcel : un serveur pour la gestion des événements réseaux
  - → Méthode d'accès choisie par l'ordonnanceur
    - scrutation/interruption
- Support pour la scrutation
  - → Fréquence contrôlée
  - → Factorisation des scrutations multiples
- Support pour les interruptions
  - → Utilisation des activations
  - → Support nécessaire dans le système d'exploitation

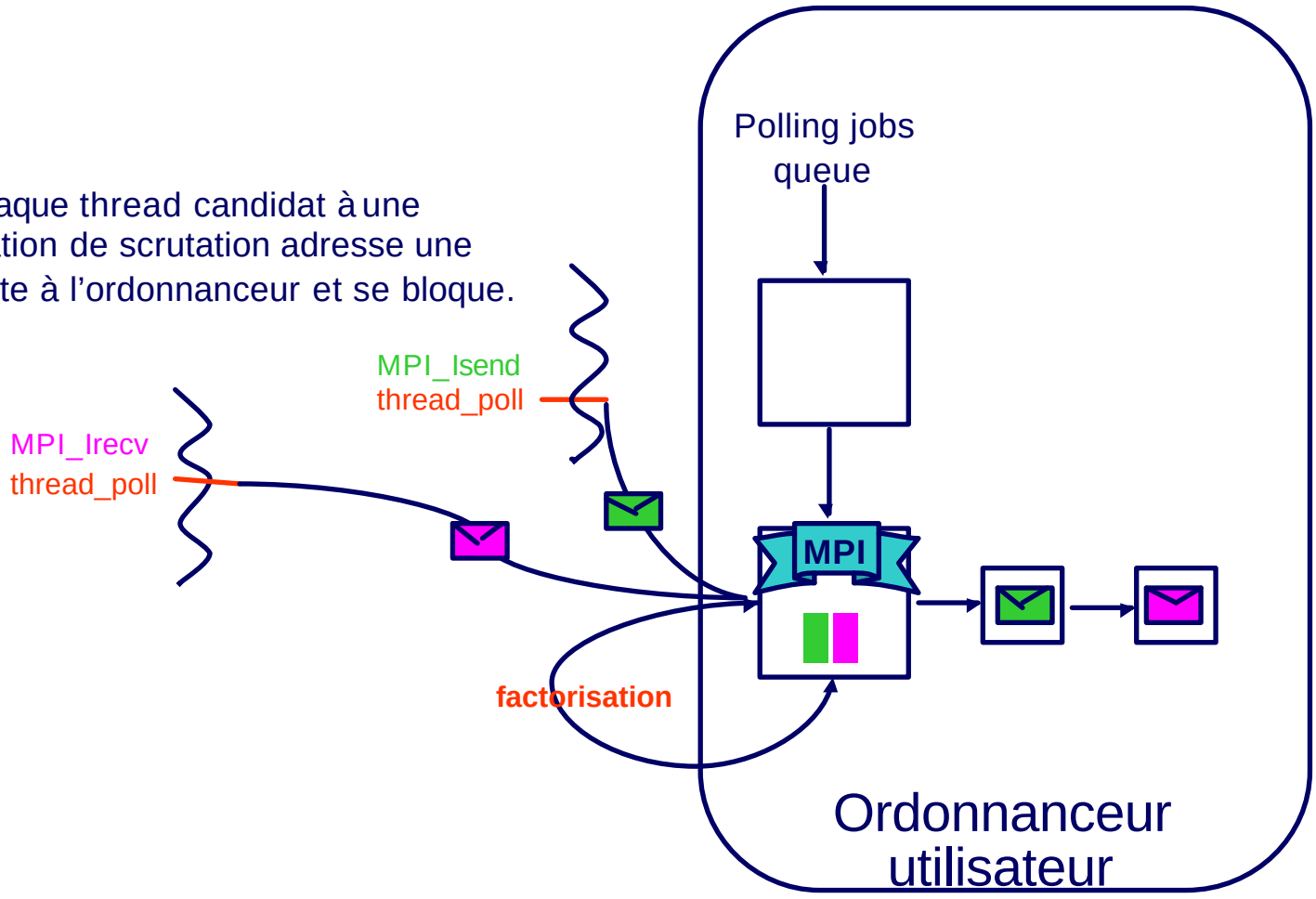
La scrutation

❶ Création d'une **catégorie** de scrutation (ex : MPI), assignation d'une **fréquence** et enregistrement de **callbacks.**

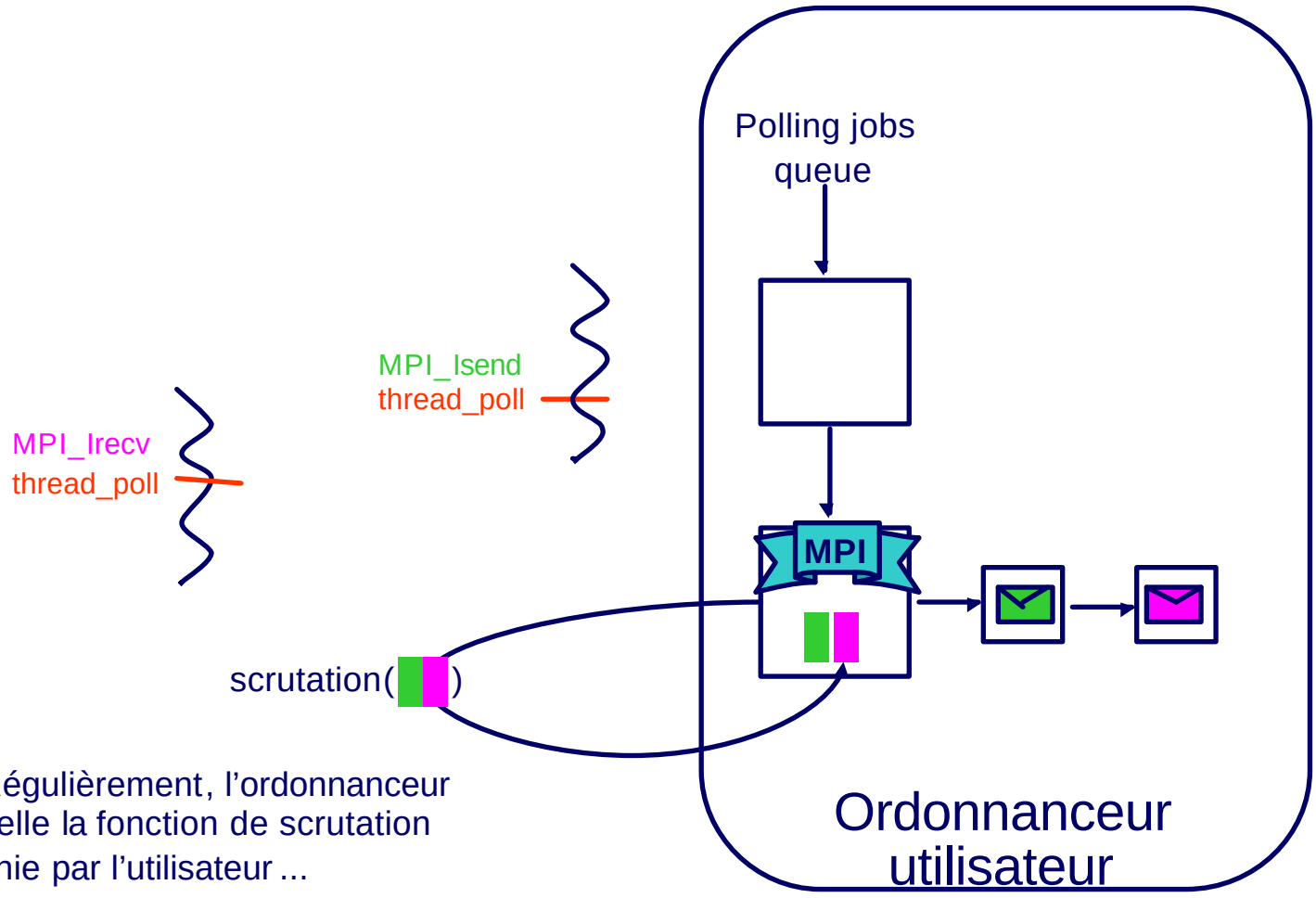


La scrutation

❷ Chaque thread candidat à une opération de scrutation adresse une requête à l'ordonnanceur et se bloque.

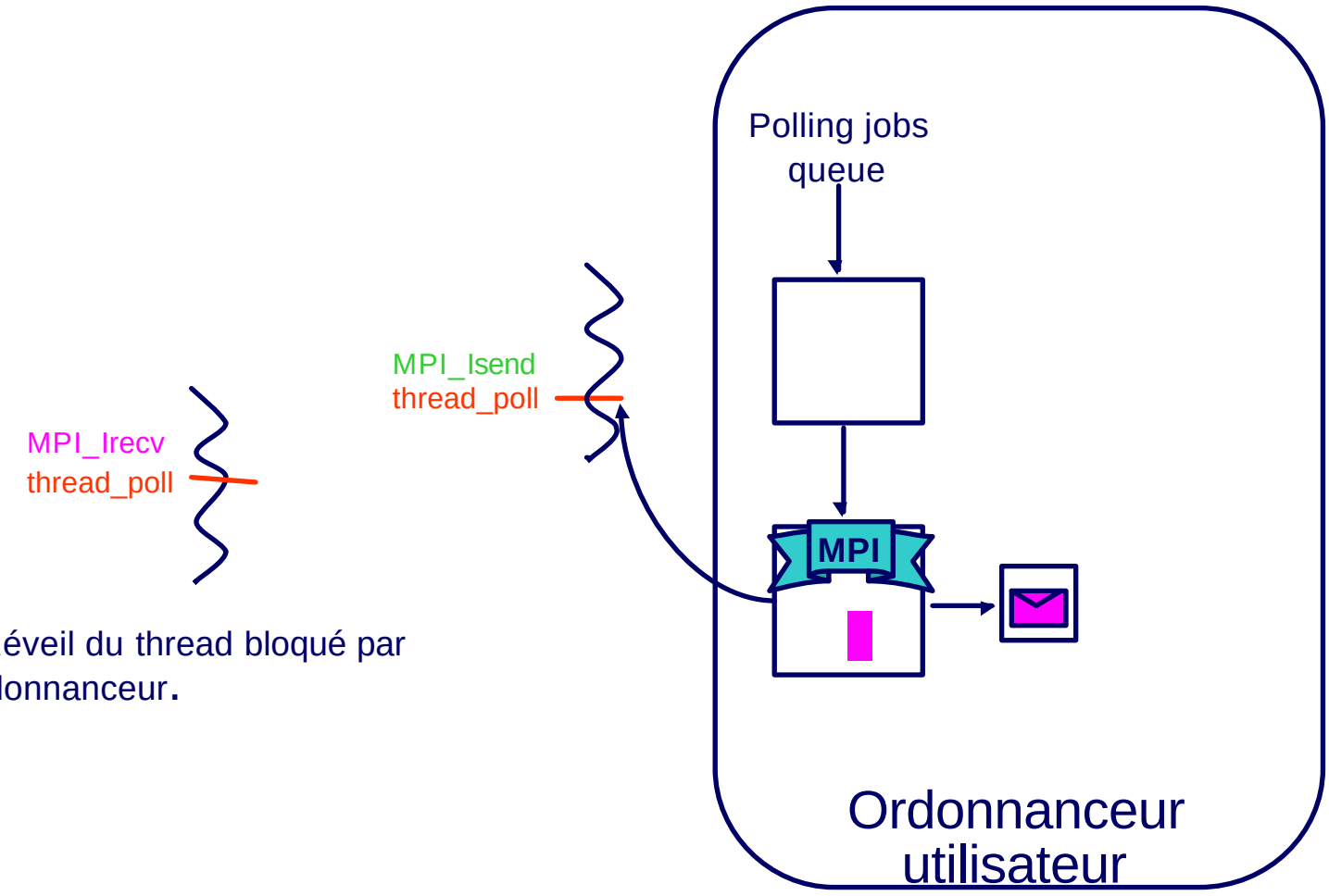


La scrutation



❸ Régulièrement, l'ordonnanceur appelle la fonction de scrutation définie par l'utilisateur ...

La scrutation



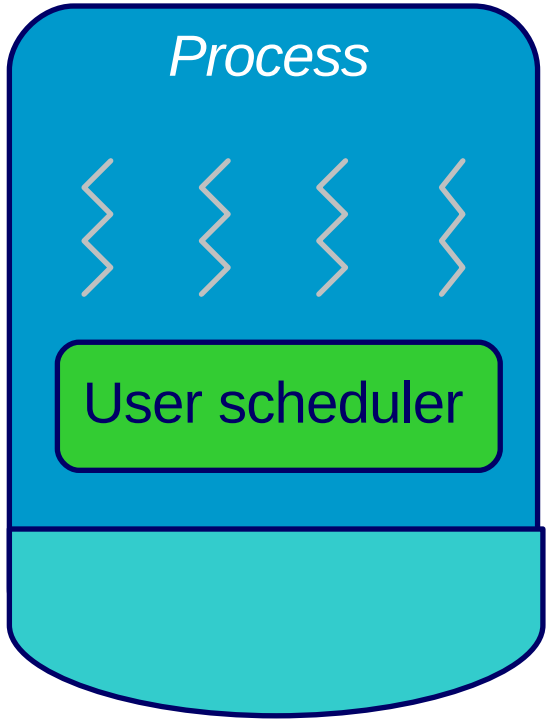
❹ Réveil du thread bloqué par l'ordonnanceur.

Interruptions et système

- *Scheduler activation* [Anderson et al. 91]
  - Idée : la coopération entre les deux ordonnanceurs est bidirectionnelle
    - → L'ordonnanceur utilisateur effectue des appels systèmes
    - → L'ordonnanceur noyau utilise des *upcalls* !
- *Upcall*
  - Informe l'application des événements noyaux
- Activations (~ processeurs virtuels)
  - Autant d'activations en exécution que de processeurs
  - Le noyau contrôle les créations/destructions des activations

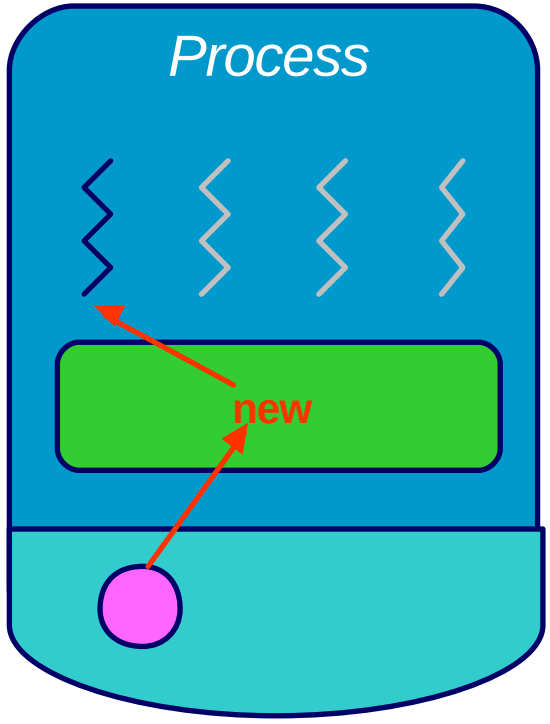
Principe des activations

- Appel système bloquant / 2 processeurs



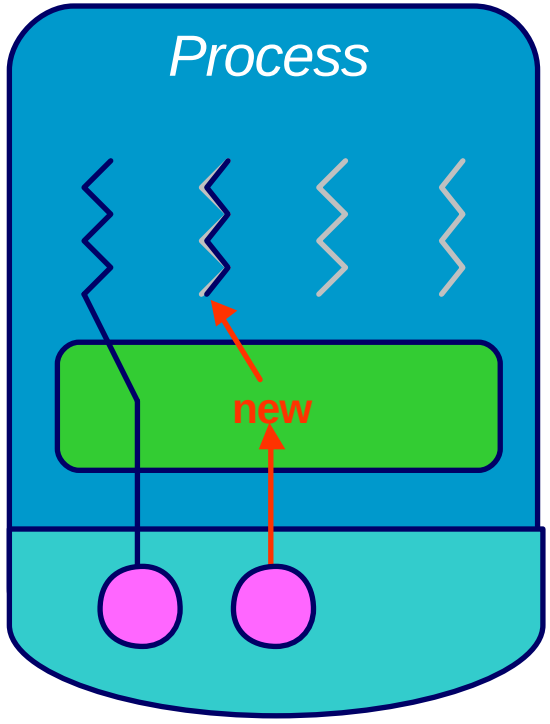
Principe des activations

- Appel système bloquant / 2 processeurs



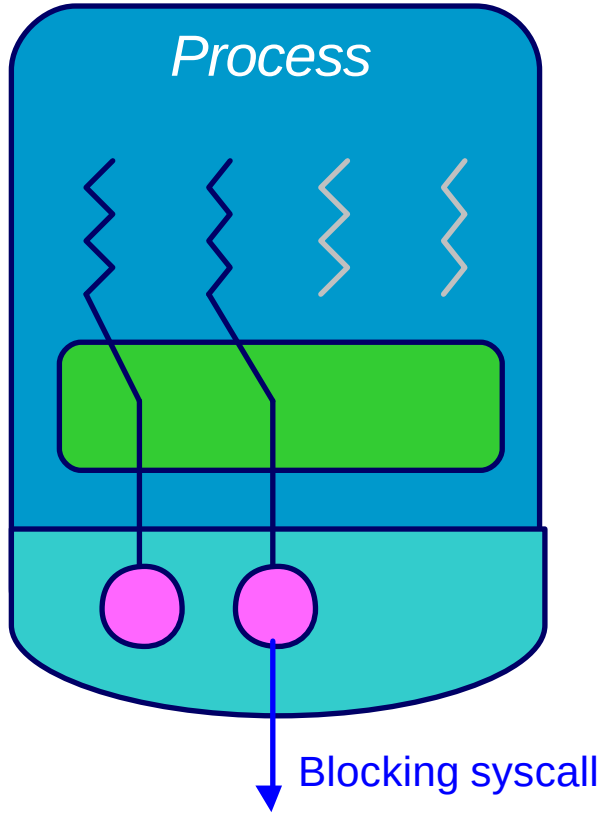
Principe des activations

- Appel système bloquant / 2 processeurs



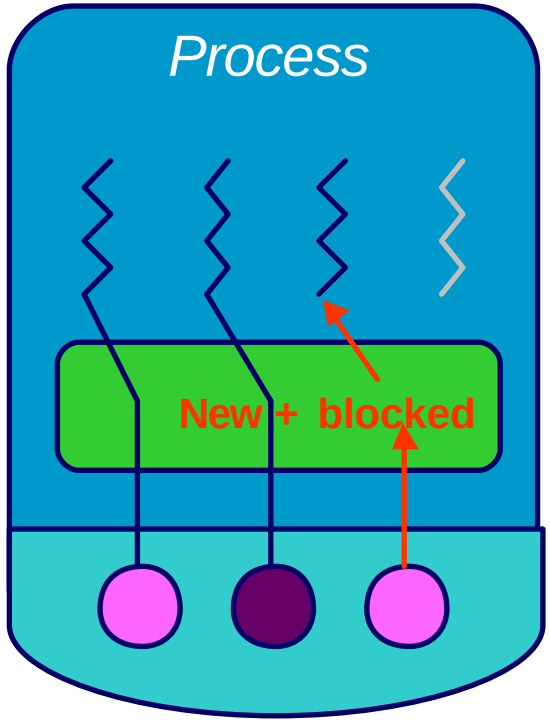
Principe des activations

- Appel système bloquant / 2 processeurs



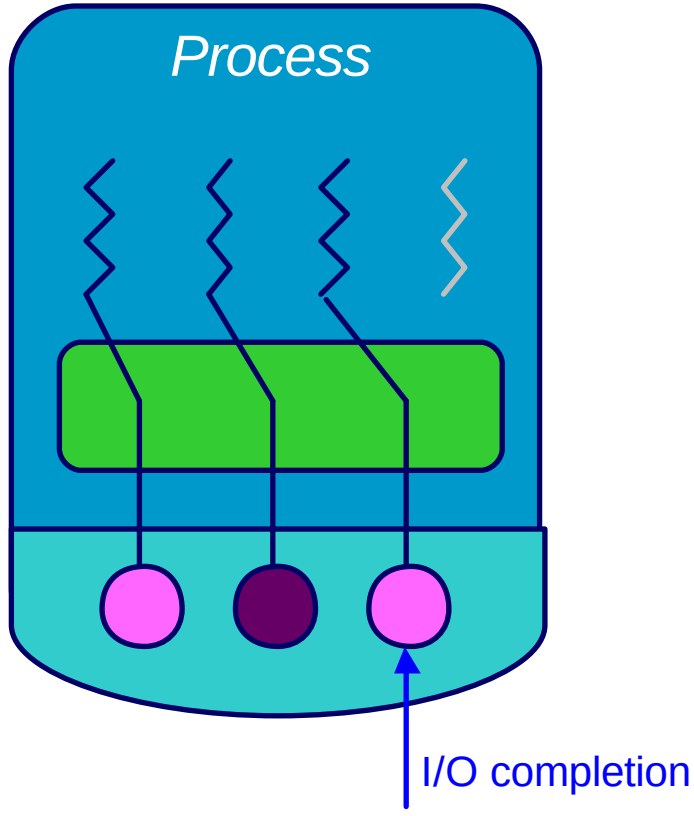
Principe des activations

- Appel système bloquant / 2 processeurs



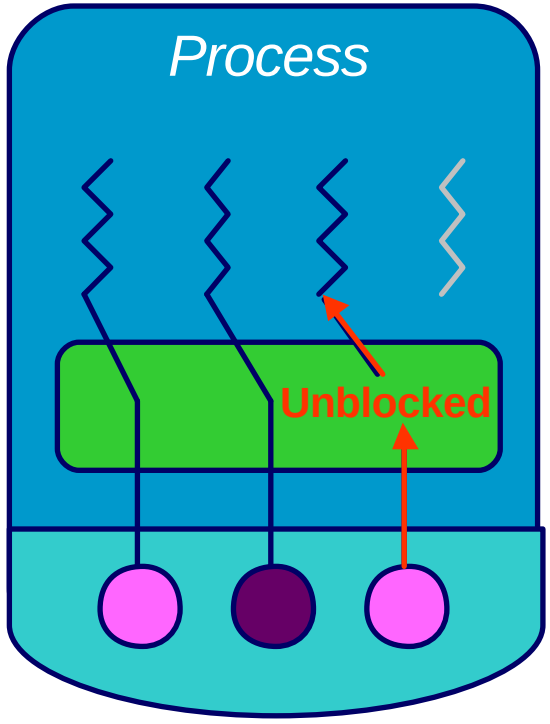
Principe des activations

- Appel système bloquant / 2 processeurs



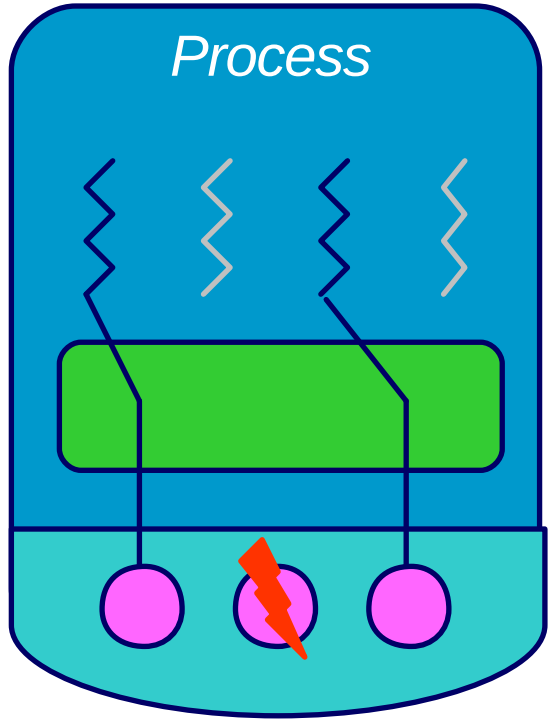
Principe des activations

- Appel système bloquant / 2 processeurs



Principe des activations

- Appel système bloquant / 2 processeurs



Amélioration des activations

- Extensions du modèle d'Anderson
  - Indépendance noyau/applications
  - Nombre d'activations non borné
    - → Tous les appels systèmes bloquants sont gérés
  - Meilleure gestion des *upcalls*
    - → *new*, *block* , *preempt*, *unblock*
- Optimisation
  - Réserve d'activations prêtes dans le noyau

Mise en œuvre

- Marcel
  - Bibliothèque de *threads* utilisateurs de PM^2
  - 10 000 lignes de code modifées
- Noyau Linux
  - Parties modifiées :
    - → schedule(), do_fork() et do_exit()
  - Code ajouté :
    - → gestion des upcalls et des nouveaux appels systèmes
  - 2 000 lignes de code ajoutées sur 30 000

Performances

Bibliothèque	Opérations sur les *threads* intensives	Opérations bloquantes
Marcel/mono	330 us	infini
Marcel/SMP	440 us	14 ms
Marcel/activation	440 us	13 ms
LinuxThread	15000 us	15 ms

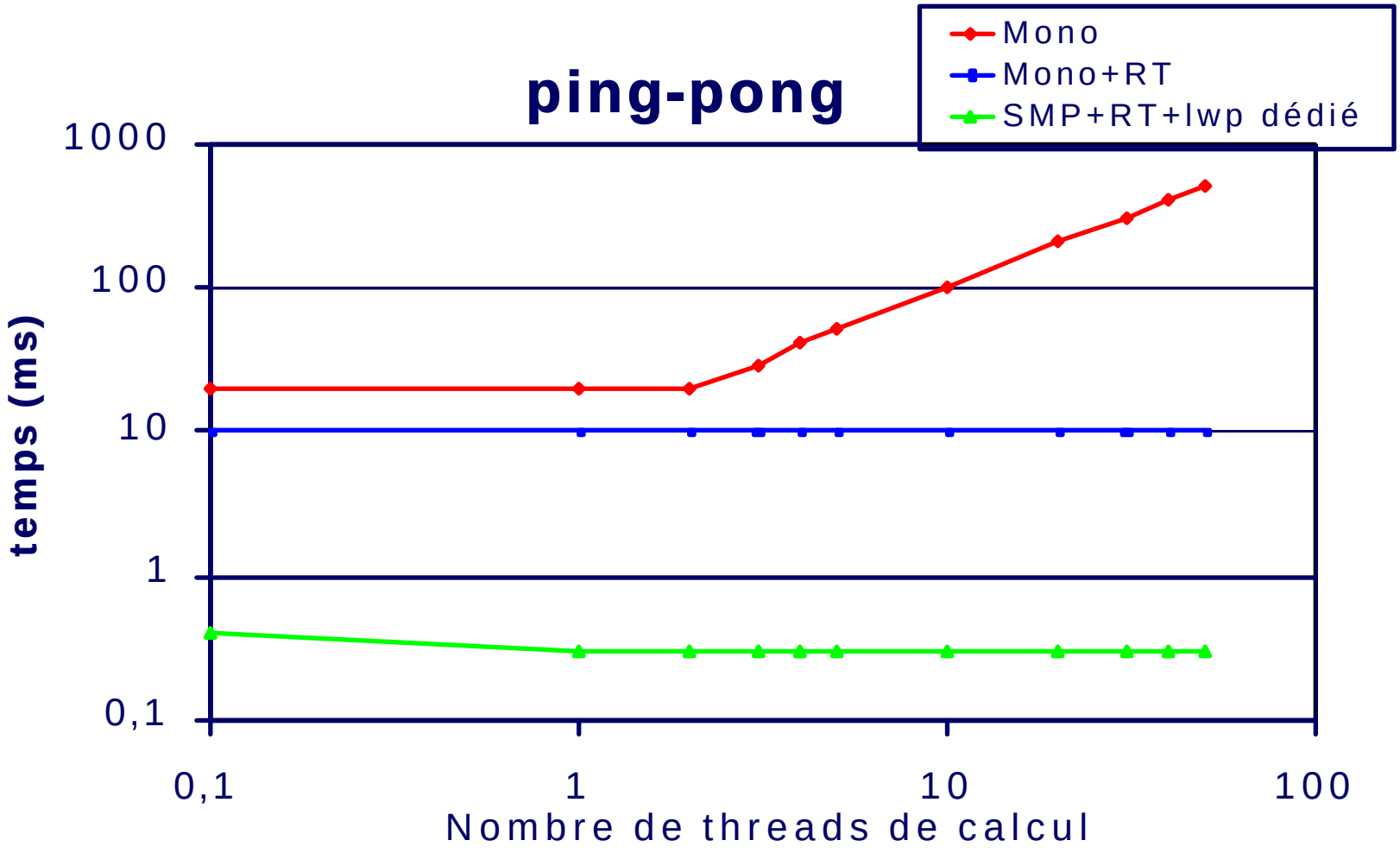
Et la réactivité ?

- Détecter un événement ne suffit pas
  - Il faut donner la main au(x) thread(s) concerné(s)
  - On ne peut pas le faire dès la détection
    - ➔ section critique => pas de `switch_to` immédiat
    - ➔ un thread peu déléguer à un autre les communications réseaux
- Deux classes de threads
  - La classe RealTime
    - ➔ threads exécutés en priorité
    - ➔ threads réseaux
      - Threads d'écoute et threads de service
  - La classe Standard
    - ➔ threads non prioritaires
      - Threads de calcul

Et sans activations ?

- Ne pas utiliser d'appels bloquants
  - Scrutation
- Utiliser des threads noyaux spécifiques
  - Amélioration de la version SMP de Marcel
  - « lwps » dédiés aux threads de communication
    - → bonnes performances, meilleure réactivité
    - → mais pas de contrôle sur l'ordonnancement noyau
      - risque d'attente active
      - impossibilité de traiter ainsi tous les appels systèmes bloquants

Expériences



Travaux actuels et futurs

- Optimisation au réveil d'une activation
  - Une fois signalé (upcall), il faut réveiller le thread
  - Quel thread était dans l'activation réveillée ?
  - Les upcalls sont asynchrones
    - → section critique possible => `switch_to` retardé
- Mesures des performances
  - Applications de test diversifiées
  - Analyse précise des coûts des différents mécanismes
- Généralisation à d'autres types d'E/S
  - Systèmes de fichiers distribués sur un cluster, ...