🇫🇷 Section A — Interruptions sous Linux

A.1 Concepts (synthèse)

Les interruptions sont un mécanisme essentiel permettant aux **périphériques matériels** ou au **logiciel** de signaler au CPU qu'un événement s'est produit et nécessite une attention immédiate.

- **Interruptions matérielles (IRQs)** : Générées par des périphériques physiques (carte réseau, disque, timer, etc.) via le contrôleur d'interruption (PIC/APIC). Elles sont **asynchrones** et peuvent survenir à tout moment, interrompant l'exécution courante du CPU.
- **SoftIRQs** : Mécanisme de **"bottom-half"** (seconde moitié) dans le noyau Linux. Elles sont **synchrones** et ne peuvent être déclenchées que par le noyau lui-même (souvent à la fin d'un *handler* d'IRQ matérielle) pour effectuer des tâches différées qui prennent trop de temps, comme le traitement des paquets réseau (**NET_RX**). Elles s'exécutent dans un contexte spécial (contexte *softirq*), mais pas dans celui d'un processus.
- **Handlers / ISRs (Interrupt Service Routines)** : Fonctions du noyau appelées pour traiter une interruption. Les handlers d'IRQs matérielles sont exécutés dans un contexte de haute priorité (**"top-half"**), où les interruptions de même ou de plus faible priorité sont souvent **masquées** (désactivées) pour garantir une exécution rapide.
- **Latence d'interruption** : Temps écoulé entre l'arrivée d'un signal d'interruption au CPU et le début de l'exécution du *handler* d'interruption correspondant. Une faible latence est cruciale pour les systèmes temps réel ou le réseau.
- **Masquage/Priorités** : Mécanisme pour gérer les interruptions. Le **masquage** (désactivation temporaire) empêche certaines IRQs d'être traitées, souvent dans le *top-half* pour éviter les réentrances et simplifier la synchronisation. Les interruptions ont des **priorités** (gérées par l'APIC) : une interruption de priorité supérieure peut interrompre un *handler* de priorité inférieure si celui-ci n'est pas masqué.
- **Context Switch (Changement de Contexte)** : Mécanisme par lequel le CPU passe de l'exécution d'un processus/thread à un autre. Contrairement à une interruption qui ne fait ès le *handler*, un changement de lu processus courant et le **oftIRQs** et les *handlers* d'IRQs ne **ocessus**.
ment réseau pour réduire la surcharge ors de fortes charges. Au lieu de tilise un mécanisme de lecter et traiter plusieurs paquets en ption coûteux vers le contexte atérielles.

---

mmenter)

Exécution :

Bash

grep . /proc/interrupts | head -n 40
cat /proc/softirqs
Exemple de sortie grep . /proc/interrupts | head -n 40 :

```
           CPU0       CPU1       CPU2       CPU3
  0:         26          0          0          0   IO-APIC-edge      timer
  1:          2          0          0          0   IO-APIC-edge      i8042
  8:          1          0          0          0   IO-APIC-edge      rtc0
 16:    2083451      54012      15034      14002   IO-APIC-fasteoi   nvme0n1
 24:     987654    1001234     998765     999001   PCI-MSI-edge      eth0-rx-0
 25:          0          0          0          0   PCI-MSI-edge      eth0-tx-0
 26:    1543210          0          0          0   PCI-MSI-edge      mlx4-comp-0
 27:          0    1543210          0          0   PCI-MSI-edge      mlx4-comp-1
```

Commentaire :
Le fichier /proc/interrupts affiche les **compteurs d'interruptions matérielles (IRQs)** par numéro d'IRQ et par CPU (CPU0, CPU1, etc.).

- **IRQ Disque (NVMe/SATA) :** Dans cet exemple, **IRQ 16** est associée à nvme0n1. On observe une concentration des interruptions sur le **CPU0**, ce qui est courant par défaut sur certaines architectures ou si irqbalance est désactivé.
- **IRQ Réseau (eth0) :** L'**IRQ 24** est identifiée comme eth0-rx-0. Les cartes réseau modernes utilisent souvent le **Message Signaled Interrupts (MSI/MSI-X)**, permettant d'assigner **plusieurs IRQs** à un seul périphérique (comme eth0-rx-0, eth0-tx-0, etc.) pour améliorer la répartition de la charge sur différents CPUs (Scalable I/O). On note ici que l'IRQ 24 est activement distribuée sur les 4 CPUs.

Exemple de sortie cat /proc/softirqs :

```
                CPU0       CPU1       CPU2       CPU3
      HI:          0          0          0          0
   TIMER:   20000000   20000000   20000000   20000000
  NET_TX:     150000     150000     150000     150000
  NET_RX:    5000000    5000000    5000000    5000000
   BLOCK:     300000     300000     300000     300000
 IRQ_POLL:          0          0          0          0
TASKLET:    1000000    1000000    1000000    1000000
 SCHED:    20000000   20000000   20000000   20000000
```

Commentaire :
Le fichier /proc/softirqs affiche les compteurs de **SoftIRQs** par type (TIMER, NET_RX, NET_TX, etc.) et par CPU. Elles représentent le travail différé (*bottom-half*). **NET_RX** (**Network Receive**) et **NET_TX** (**Network Transmit**) sont cruciales pour le réseau. On observe une répartition équilibrée des SoftIRQs sur les CPUs.

📌 Affinité des IRQ (SMP)

Exécution : (Supposons que l'IRQ réseau est **24** comme dans l'exemple)

Bash

```
IRQ=24
cat /proc/irq/$IRQ/smp_affinity_list
```

Exemple de sortie cat /proc/irq/24/smp_affinity_list :

```
0-3
```

(Indique que l'IRQ 24 peut être traitée par les CPUs 0, 1, 2 et 3).

Explication de l'intérêt d'assigner une IRQ réseau à un CPU local :

Dans un système **Symmetric Multiprocessing (SMP)** avec plusieurs CPUs et souvent plusieurs cartes réseau ou files d'attente (**RX/TX queues**) par carte, l'**affinité d'IRQ** permet de désigner quel CPU doit traiter une IRQ donnée.

L'intérêt principal d'assigner une IRQ réseau à un **CPU local** à la file RX/TX est de garantir la **co-localisation** des données et des traitements :

1. **Meilleure performance du cache (Cache Affinity)** : Le traitement (*handler* d'IRQ, SoftIRQ) du paquet se déroule sur le même CPU qui a peut-être déjà des données liées au flux réseau dans son **cache** (L1/L2). Cela réduit les **cache misses** et évite le coût élevé de la **migration de cache** entre les CPUs (appelé *cache coherency overhead*).
2. **Réduction des latences NUMA** : Sur les architectures **Non-Uniform Memory Access (NUMA)**, si la carte réseau est connectée à un *socket* CPU (Node) particulier, il est plus rapide de traiter l'IRQ sur un CPU de ce même *socket* car l'accès à la mémoire associée à la carte réseau sera plus rapide (*local memory access*).
3. **Éviter les "Hot Spots"** : La répartition des IRQs sur différents CPUs empêche un seul CPU d'être surchargé, améliorant la **capacité de débit globale**. Le mécanisme **RPS (Receive Packet Steering)**, qui s'appuie sur l'affinité, permet de distribuer davantage le traitement des paquets au-delà de la simple IRQ.

📈 Générer une charge IRQ (réseau local)

Exécution :

Bash

```
# Dans une fenêtre A
ping -f -c 2000 127.0.0.1
# Dans une fenêtre B, à lancer avant
watch -n1 'cat /proc/softirqs | sed -n "1p;/NET_RX/p"'
```

Exemple d'observation de NET_RX :

Temps	**CPU0 (Avant)**	**CPU1 (Avant)**	**...**	**CPU0 (Pendant Ping -f)**	**CPU1 (Pendant Ping -f)**
T0	5,000,000	5,000,000		5,000,000	5,000,000
T1	5,000,000	5,000,000		**5,001,500**	**5,001,490**
T2	5,000,000	5,000,000		**5,003,100**	**5,003,050**
T3	5,000,000	5,000,000		**5,004,700**	**5,004,680**

Commentaire sur l'évolution de NET_RX et la répartition par CPU :

1. **Augmentation de NET_RX** : Pendant le ping -f (qui génère un flux rapide de paquets ICMP *localement* via l'interface **loopback**), le nombre de *SoftIRQs* de type **NET_RX** augmente rapidement et de manière significative. Le *loopback* est extrêmement rapide et génère beaucoup de travail de traitement de paquets (même si aucune IRQ matérielle n'est impliquée).
2. **Répartition Équilibrée** : On observe que les compteurs **NET_RX** augmentent sur **tous les CPUs** (CPU0, CPU1, etc.). Cela indique que le noyau Linux répartit efficacement la charge de traitement des paquets reçus (l'exécution des SoftIRQs **NET_RX**) sur les différents cœurs disponibles pour éviter la saturation d'un seul cœur, maximisant ainsi le débit de la boucle locale.

A.4 Question guidée

Que se passe-t-il si un handler est « trop long » ?

Si la partie **« top-half »** d'un *handler* d'interruption matérielle (ISR) prend **trop de temps**, cela a des conséquences graves sur la réactivité du système :

- **Impacts sur la Latence Globale** :
  - **Augmentation de la Latence d'Interruption** : Tant que le *handler* long s'exécute, il bloque le traitement de **toutes les autres interruptions** de même ou de plus faible priorité (car elles sont généralement masquées dans le *top-half*). Le temps de réponse à ces autres événements (e.g., autre périphérique, timer, scheduler) est considérablement retardé.
  - **Impact sur le Scheduler** : Le *handler* s'exécute dans un contexte de haute priorité, **empêchant le *scheduler* de reprendre la main**. Les processus utilisateurs et même les processus noyaux bas-niveau sont mis en attente, ce qui rend le système **lent** ou **non réactif**.
- **Perte de Paquets/Événements** :
  - Pour les cartes réseau, si un *handler* est trop long, le CPU met trop de temps à vider les tampons du contrôleur réseau. Si le matériel reçoit de nouveaux paquets pendant ce temps et que ses tampons sont pleins, il doit **jeter les paquets (perte de paquets)**.
  - Même problème pour les disques ou d'autres périphériques rapides. L'événement (e.g., *completion*) est manqué ou retardé.

Solutions pour les Handlers Trop Longs :

La règle d'or pour un *top-half* est d'être **le plus court possible**. Le travail intensif doit être **déféré** (reporté) à une autre phase, appelée **bottom-half**.

1. **Déférer en Bottom-Half (SoftIRQs, Tasklets, Workqueues)** :
   - Le *top-half* se contente d'acquitter l'interruption, de lire rapidement les registres et de planifier l'exécution d'un **bottom-half**.
   - **SoftIRQs** : Mécanisme très rapide et réentrant (peut s'exécuter sur plusieurs CPUs simultanément) pour les tâches critiques comme le réseau (**NET_RX**). Elles s'exécutent dans un contexte d'interruption.
   - **Tasklets** : Construits sur les SoftIRQs, plus simples et non réentrants pour un même Tasklet (ne peuvent s'exécuter que sur un seul CPU à la fois).
   - **Workqueues** : Défèrent le travail à un **contexte de processus** (un thread noyau dédié). Cela permet d'utiliser des fonctions qui peuvent **dormir** (*sleep*) (comme les mutex, l'allocation mémoire bloquante), ce qui est interdit dans le contexte SoftIRQ/Tasklet. Idéal pour les tâches plus complexes et moins sensibles à la latence.
2. **NAPI (New API) pour le Réseau** :
   - C'est la solution moderne au problème de la surcharge d'IRQs réseau. Après une

première IRQ, NAPI **désactive l'interruption** du périphérique et bascule en mode **sondage (*polling*)** via l'exécution d'une **SoftIRQ NET_RX**.
○ Le **NET_RX SoftIRQ** collecte un **maximum de paquets en bloc** et les traite avant de se terminer. Si la file RX est vide, l'interruption du périphérique est **réactivée**.
○ Ceci **réduit la fréquence des IRQs matérielles** et déplace le travail lourd de gestion des paquets dans le contexte SoftIRQ plus approprié.

Souhaitez-vous que je vous fournisse les commandes détaillées et les explications pour la section A.3 sur l'**Affinité et l'Isolement des IRQ** (qui nécessite *sudo*)?