

Nouvelle techniques d'accélération du réseau

Antoine BLIN

January 16, 2019

Introduction

- 20 dernières années :
- puissance des CPUs multipliée par 370
 - débit des cartes réseau multiplié par 100 000

Introduction



- 20 dernières années :
- puissance des CPUs multipliée par 370
 - débit des cartes réseau multiplié par 100 000

Facteur limitant : ❌ la bande passante réseau

✅ le CPU : envoi/réception des paquets

Introduction

- 20 dernières années :
- puissance des CPUs multipliée par 370
 - débit des cartes réseau multiplié par 100 000



Facteur limitant :  la bande passante réseau
 le CPU : envoi/réception des paquets

Piles réseau OS et API socket conçues pour :

- des cartes réseau avec des bande passante limitée (1GbE)
- minimiser la consommation mémoire

Introduction

- 20 dernières années :
- puissance des CPUs multipliée par 370
 - débit des cartes réseau multiplié par 100 000

Facteur limitant :  la bande passante réseau
 le CPU : envoi/réception des paquets



Piles réseau OS et API socket conçues pour :

- des cartes réseau avec des bande passante limitée (1GbE)
- minimiser la consommation mémoire

⇒ exploiter les cartes réseau à hautes performances (40/100 GbE)

Introduction

- 20 dernières années :
- puissance des CPUs multipliée par 370
 - débit des cartes réseau multiplié par 100 000

Facteur limitant :  la bande passante réseau
 le CPU : envoi/réception des paquets

Piles réseau OS et API socket conçues pour :

- des cartes réseau avec des bande passante limitée (1GbE)
- minimiser la consommation mémoire

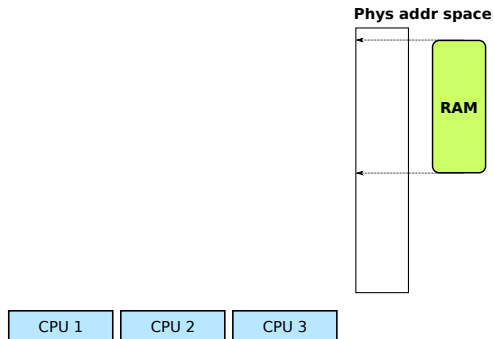
⇒ exploiter les cartes réseau à hautes performances (40/100 GbE)

Solutions :

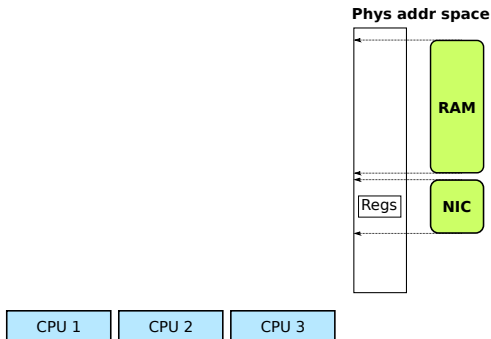
- diminuer le temps de réception/émission
- libérer du temps CPU pour le traitement des données

Historique

Initialisation de la carte réseau (NIC)

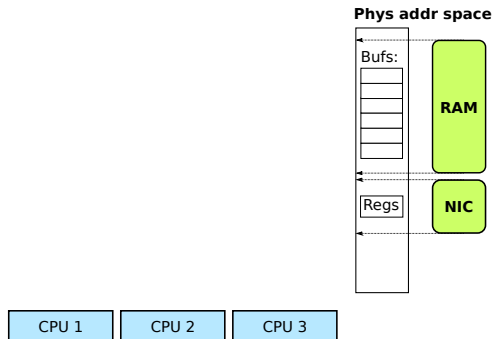


Initialisation de la carte réseau (NIC)



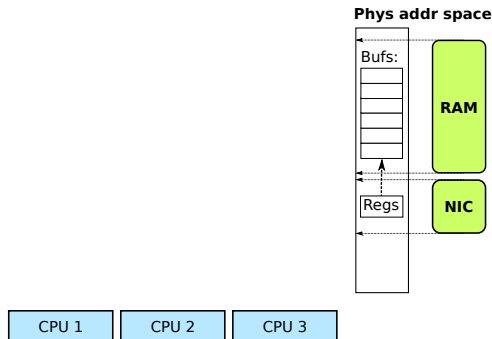
1. Les registres du NIC sont mappés dans l'espace d'@ physique

Initialisation de la carte réseau (NIC)



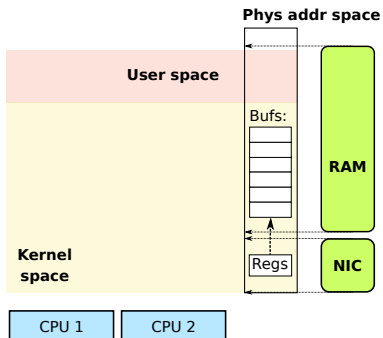
1. Les registres du NIC sont mappés dans l'espace d'@ physique
2. Un tableau de « packet buffers » est alloué en RAM

Initialisation de la carte réseau (NIC)



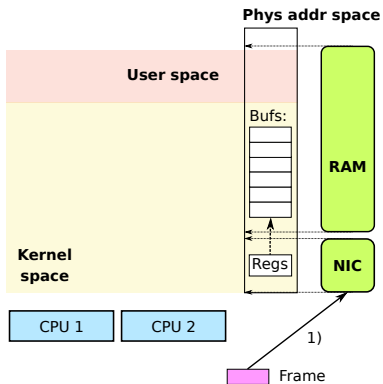
1. Les registres du NIC sont mappés dans l'espace d'@ physique
2. Un tableau de « packet buffers » est alloué en RAM
3. Les registres du NIC registers sont mis à jours avec l'@ du tableau

Pile réseau de Linux : réception



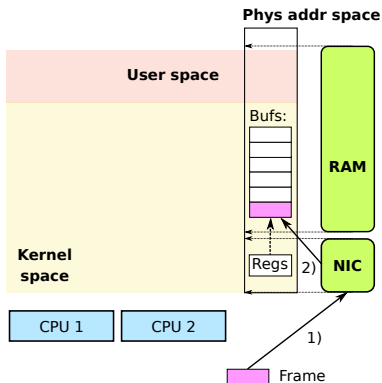
Pile réseau de Linux : réception

1. Une trame est reçue par le NIC



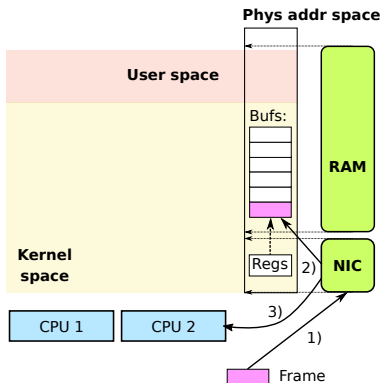
Pile réseau de Linux : réception

1. Une trame est reçue par le NIC
2. Le NIC effectue un accès DMA pour écrire la trame en RAM.

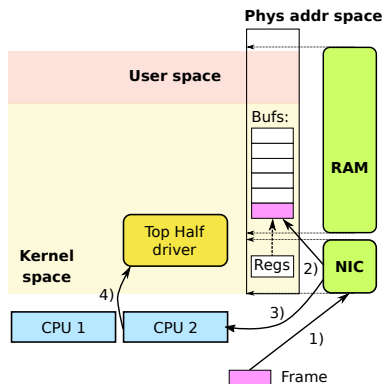


Pile réseau de Linux : réception

1. Une trame est reçue par le NIC
2. Le NIC effectue un accès DMA pour écrire la trame en RAM.
3. Le NIC lève une IRQ

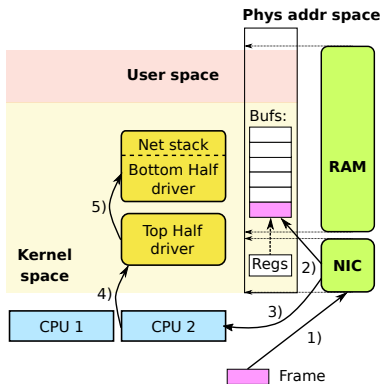


Pile réseau de Linux : réception



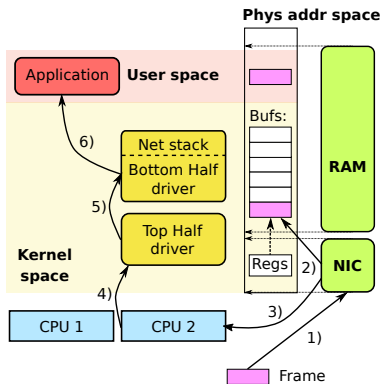
1. Une trame est reçue par le NIC
2. Le NIC effectue un accès DMA pour écrire la trame en RAM.
3. Le NIC lève une IRQ
4. Le handler d'interruption (top half) est déclenché et l'IRQ est acquittée auprès du NIC

Pile réseau de Linux : réception



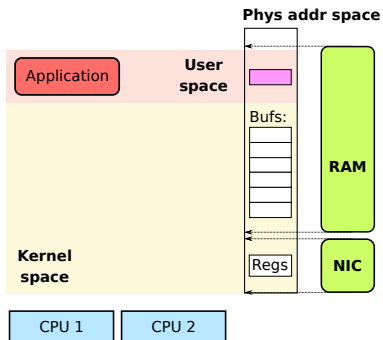
1. Une trame est reçue par le NIC
2. Le NIC effectue un accès DMA pour écrire la trame en RAM.
3. Le NIC lève une IRQ
4. Le handler d'interruption (top half) est déclenché et l'IRQ est acquittée auprès du NIC
5. La trame est transmise gestionnaire bottom Half

Pile réseau de Linux : réception



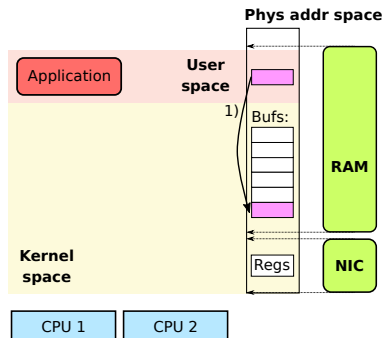
1. Une trame est reçue par le NIC
2. Le NIC effectue un accès DMA pour écrire la trame en RAM.
3. Le NIC lève une IRQ
4. Le handler d'interruption (top half) est déclenché et l'IRQ est acquittée auprès du NIC
5. La trame est transmise gestionnaire bottom Half
6. Les données sont copiées dans l'application

Pile réseau de Linux : transmission



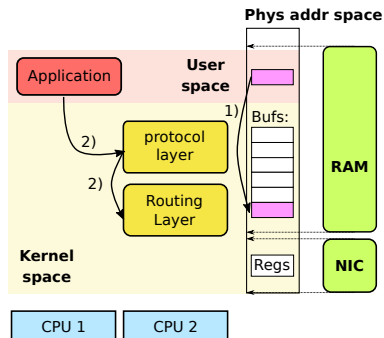
Pile réseau de Linux : transmission

1. Les données sont copiées en utilisant un appel système



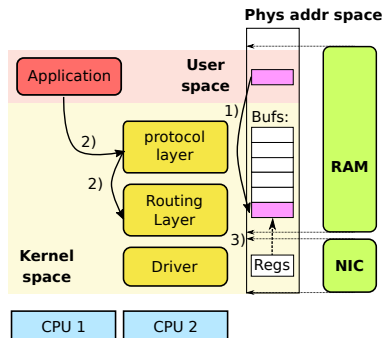
Pile réseau de Linux : transmission

1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau

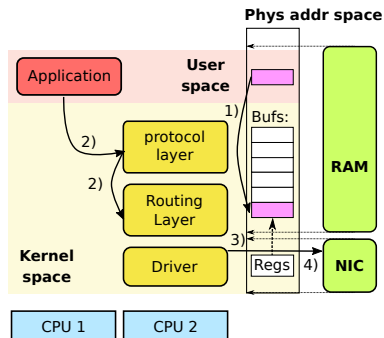


Pile réseau de Linux : transmission

1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau
3. Le driver crée un mapping DMA vers la zone mémoire des données

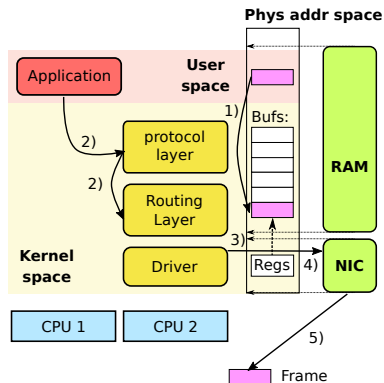


Pile réseau de Linux : transmission



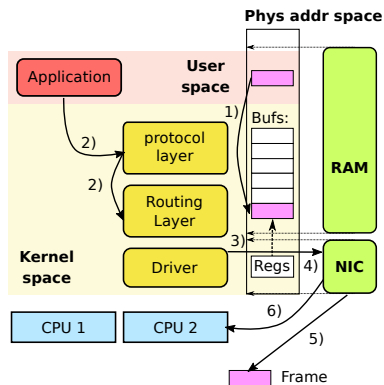
1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau
3. Le driver crée un mapping DMA vers la zone mémoire des données
4. Et signale au NIC que des données peuvent être transmises.

Pile réseau de Linux : transmission



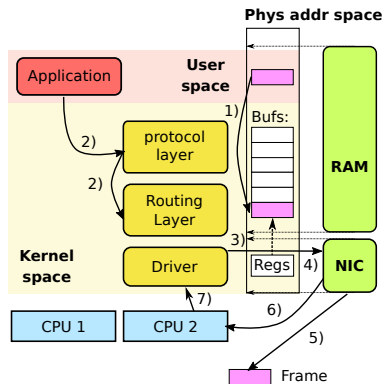
1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau
3. Le driver crée un mapping DMA vers la zone mémoire des données
4. Et signale au NIC que des données peuvent être transmises.
5. Le NIC récupère les données depuis la RAM et commence la transmission.

Pile réseau de Linux : transmission



1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau
3. Le driver crée un mapping DMA vers la zone mémoire des données
4. Et signale au NIC que des données peuvent être transmises.
5. Le NIC récupère les données depuis la RAM et commence la transmission.
6. Le NIC envoie une interruption pour signaler la fin du transfert.

Pile réseau de Linux : transmission



1. Les données sont copiées en utilisant un appel système
2. Les données transitent par la pile réseau
3. Le driver crée un mapping DMA vers la zone mémoire des données
4. Et signale au NIC que des données peuvent être transmises.
5. Le NIC récupère les données depuis la RAM et commence la transmission.
6. Le NIC envoie une interruption pour signaler la fin du transfert.
7. Le driver dé-map la région DMA et désalloue les données

Pile réseau de Linux : limitations

La pile réseau de Linux est adaptée aux cartes réseau génériques

Pile réseau de Linux : limitations

La pile réseau de Linux est adaptée aux cartes réseau génériques

L'arrivée des nouveaux NICs: 40GbE, 100GbE a mis en exergue certaines limitations :

Pile réseau de Linux : limitations

La pile réseau de Linux est adaptée aux cartes réseau génériques

L'arrivée des nouveaux NICs: 40GbE, 100GbE a mis en exergue certaines limitations :

- Copies entre l'espace mémoire noyau et utilisateur
- Tempêtes d'interruptions
- Coût des changements de contextes entre le mode noyau et utilisateur :
 - ▶ Interruptions
 - ▶ Appels système utilisé pour envoyer/recevoir
- Surcoûts de la pile réseau génériques
- Allocation mémoire par paquet

Solution 1 : Kernel Bypass

Solution 1 : Kernel Bypass

Suppression de la couche noyau

Solution 1 : Kernel Bypass

Suppression de la couche noyau

L'application récupère un accès direct au NIC :

- Utilisation de polling en place des interruptions
- Suppression des appels systèmes
- Suppression des copies mémoire
- Utilisation d'une pile réseau conçue pour l'application

Solution 1 : Kernel Bypass

Suppression de la couche noyau

L'application récupère un accès direct au NIC :

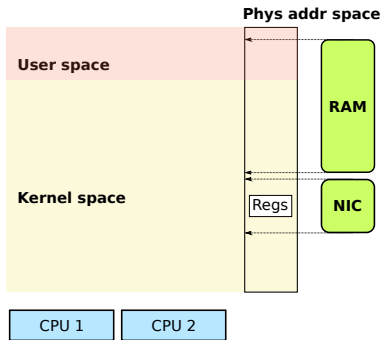
- Utilisation de polling en place des interruptions
- Suppression des appels systèmes
- Suppression des copies mémoire
- Utilisation d'une pile réseau conçue pour l'application

Différents frameworks :

- netmap: Conserve une partie du code dans le noyau
- DPDK/PF_RING: Donne un contrôle total à l'application

Kernel Bypass : reception

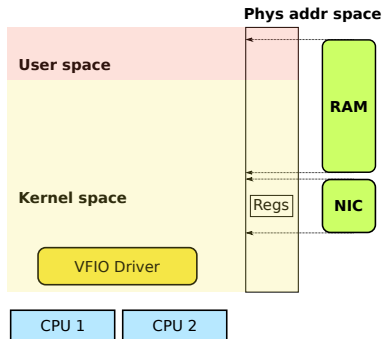
Configuration :



Kernel Bypass : reception

Configuration :

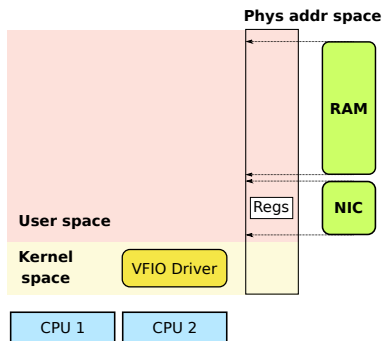
- Le module DPDK est chargé



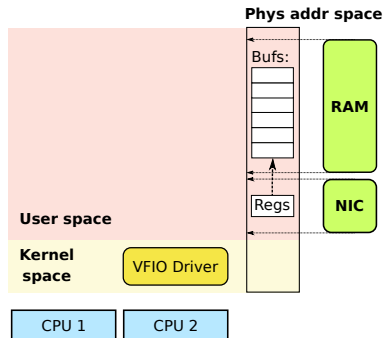
Kernel Bypass : reception

Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC



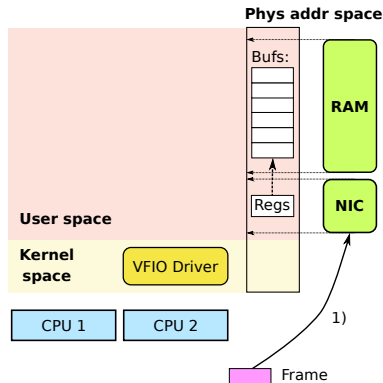
Kernel Bypass : reception



Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Kernel Bypass : reception



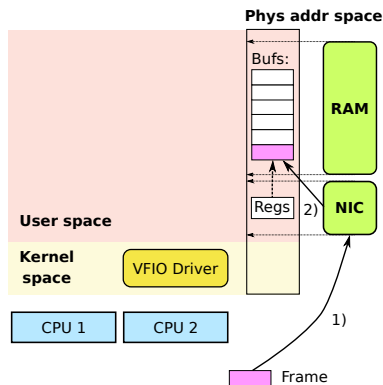
Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Réception :

1. Une trame est reçue par le NIC

Kernel Bypass : reception



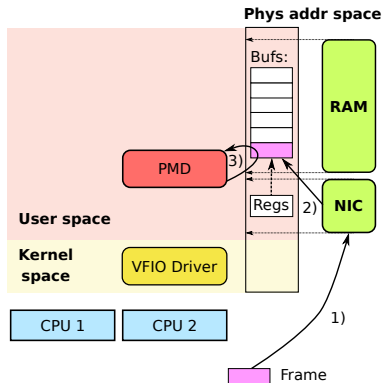
Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Réception :

1. Une trame est reçue par le NIC
2. Le NIC utilise le DMA pour l'écrire en RAM.

Kernel Bypass : reception



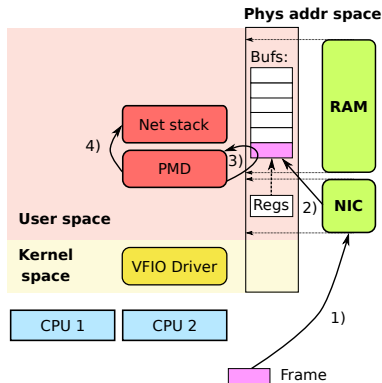
Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Réception :

1. Une trame est reçue par le NIC
2. Le NIC utilise le DMA pour l'écrire en RAM.
3. Polling est effectué pour lire la trame

Kernel Bypass : reception



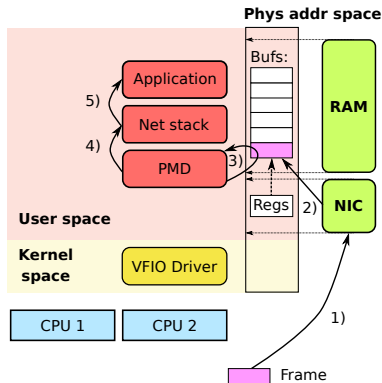
Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Réception :

1. Une trame est reçue par le NIC
2. Le NIC utilise le DMA pour l'écrire en RAM.
3. Polling est effectué pour lire la trame
4. La trame est décodée par la pile réseau optionnelle

Kernel Bypass : reception



Configuration :

- Le module DPDK est chargé
- Les registres du NIC sont mappés dans l'espace d'adressage du NIC
- Un tableau de « packet buffers » est alloué en RAM

Réception :

1. Une trame est reçue par le NIC
2. Le NIC utilise le DMA pour l'écrire en RAM.
3. Polling est effectué pour lire la trame
4. La trame est décodée par la pile réseau optionnelle
5. L'application reçoit la donnée

Kernel Bypass : partage du matériel

Sans kernel bypass : la pile réseau est utilisée pour partager le NIC

Kernel Bypass : partage du matériel

Sans kernel bypass : la pile réseau est utilisée pour partager le NIC

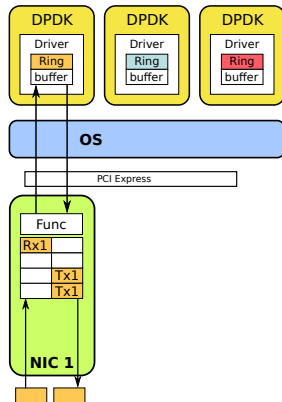
Avec kernel bypass ?

Kernel Bypass : partage du matériel

Sans kernel bypass : la pile réseau est utilisée pour partager le NIC

Avec kernel bypass ?

- Utiliser un NIC dédié par application: Pertinent pour des applications à hautes performances

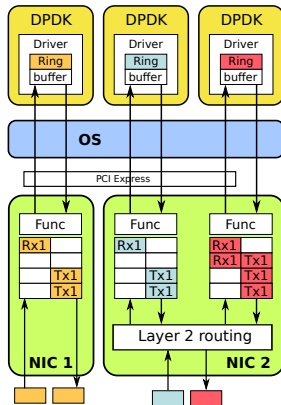


Kernel Bypass : partage du matériel

Sans kernel bypass : la pile réseau est utilisée pour partager le NIC

Avec kernel bypass ?

- Utiliser un NIC dédié par application: Pertinent pour des applications à hautes performances
- Utiliser SR-IOV:
 - ▶ Spécification pour partager des périphériques PCIe
 - ▶ Le NIC est configuré pour créer des NICs Virtuels
 - ▶ Un commutateur de niveau 2 est embarqué dans le NIC pour router les trames



Kernel Bypass : résumé

- ✓ Copies entre l'espace noyau et utilisateur \Rightarrow Zero copy
- ✓ Tempêtes d'interruptions \Rightarrow Polling
- ✓ Coût des contextes switch \Rightarrow pas d'appels système
- ✓ Surcoût de la pile réseau générique \Rightarrow Pile réseau dédiée
- ✓ Allocation mémoire par paquet \Rightarrow Pre-allocation des paquets

Kernel Bypass : résumé

- ✓ Copies entre l'espace noyau et utilisateur \Rightarrow Zero copy
- ✓ Tempêtes d'interruptions \Rightarrow Polling
- ✓ Coût des contextes switch \Rightarrow pas d'appels système
- ✓ Surcoût de la pile réseau générique \Rightarrow Pile réseau dédiée
- ✓ Allocation mémoire par paquet \Rightarrow Pre-allocation des paquets
- ✗ Le partage nécessite du matériel dédié et est borné
- ✗ la création de pile réseau dédiée est laborieuse
 - \Rightarrow nécessite de redévelopper l'outillage
 - \Rightarrow pertinent pour des applications spécifiques (NFV)
- ✗ Les problèmes de sécurité sont plus difficiles à gérer

Solution 2 : extended BPF

Solution 2 : extended BPF

Machine virtuelle/bytecode intégré dans le noyau

Solution 2 : extended BPF

Machine virtuelle/bytecode intégré dans le noyau

Permet à un programme d'injecter du bytecode dans des hooks
noyau :

- Rapide : compilation en natif (JIT)
- Sûr : vérification statique effectuée sur le bytecode

Solution 2 : extended BPF

Machine virtuelle/bytecode intégré dans le noyau

Permet à un programme d'injecter du bytecode dans des hooks noyau :

- Rapide : compilation en natif (JIT)
- Sûr : vérification statique effectuée sur le bytecode

Originellement utilisé pour filtrer/capturer des paquets réseau

Solution 2 : extended BPF

Machine virtuelle/bytecode intégré dans le noyau

Permet à un programme d'injecter du bytecode dans des hooks noyau :

- Rapide : compilation en natif (JIT)
- Sûr : vérification statique effectuée sur le bytecode

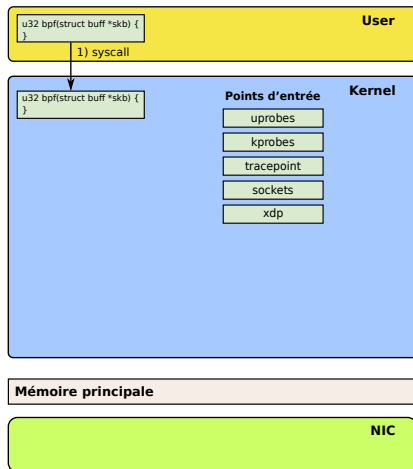
Originellement utilisé pour filtrer/capturer des paquets réseau

Maintenant utilisé pour d'autres usages plus génériques :

- Trace (fonctions du noyaux)
- Fonctions réseau
- Sécurité (Réalisation de politique)
- Monitoring d'applications

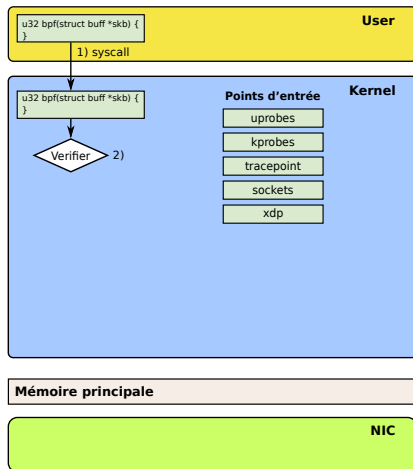
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF



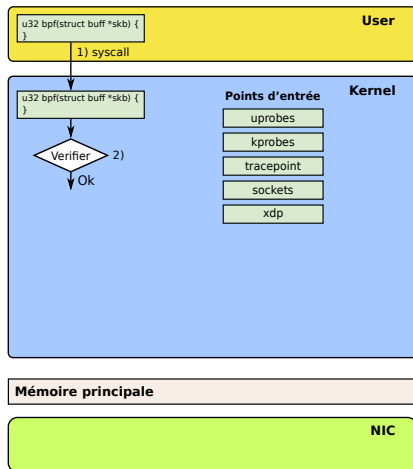
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme



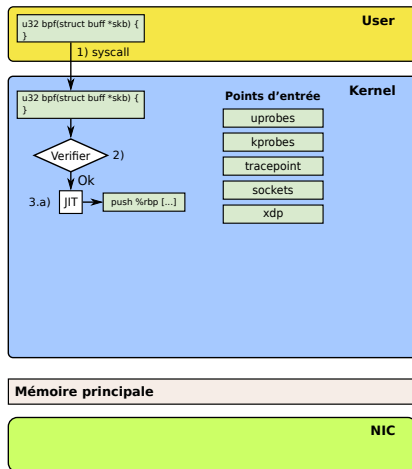
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :



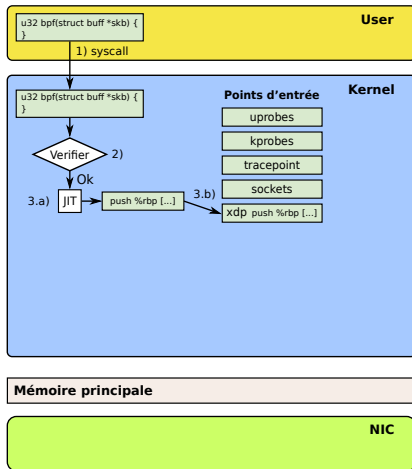
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)



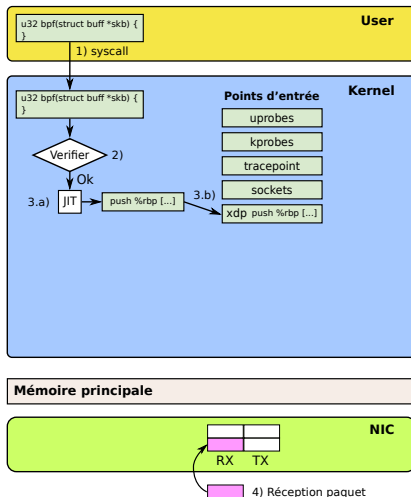
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée



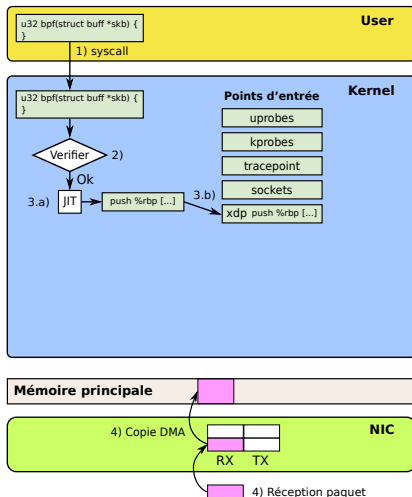
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée
4. Déclenchement du chemin d'exécution du programme



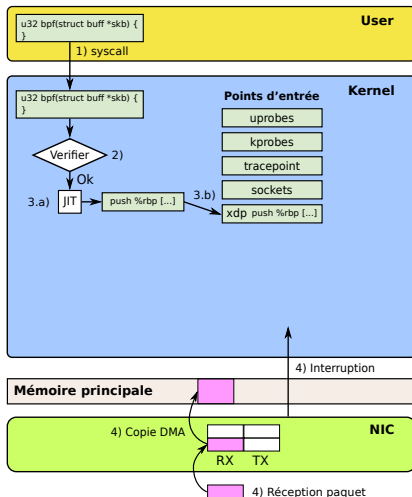
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée
4. Déclenchement du chemin d'exécution du programme



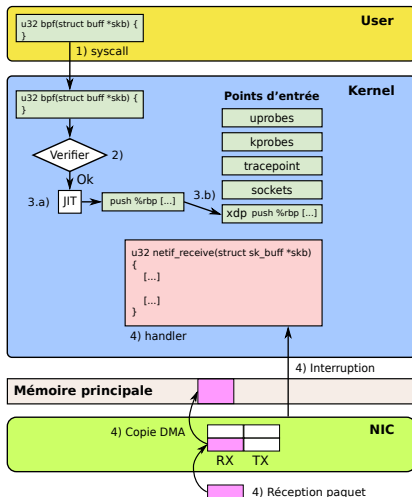
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée
4. Déclenchement du chemin d'exécution du programme



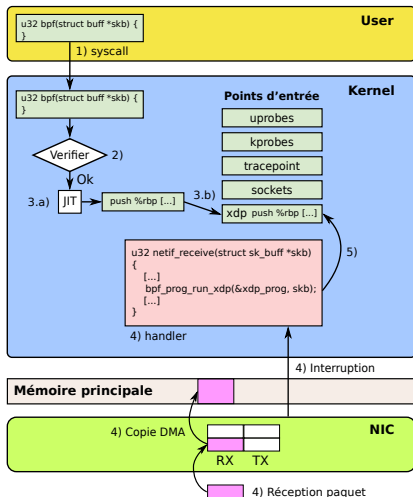
eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée
4. Déclenchement du chemin d'exécution du programme



eBPF : Principes de fonctionnement

1. Syscall : charger un programme BPF
2. Verification du programme
3. Si le programme est valide :
 - a) Compilation dans le langage de la machine (JIT)
 - b) Mise en place dans le point d'entrée
4. Déclenchement du chemin d'exécution du programme
5. Exécution du programme



Architecture de eBPF : bytecode

Jeu d'instruction RISC conçu :

- pour être facilement généré depuis un compilateur C/Rust/P4...
- pour être compilé en code natif
(JIT présent dans x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm)

Architecture de eBPF : bytecode

Jeu d'instruction RISC conçu :

- pour être facilement généré depuis un compilateur C/Rust/P4...
- pour être compilé en code natif
(JIT présent dans x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm)

Instructions 64 bits (87 instructions pour le moment)

Architecture de eBPF : bytecode

Jeu d'instruction RISC conçu :

- pour être facilement généré depuis un compilateur C/Rust/P4...
- pour être compilé en code natif
(JIT présent dans x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm)

Instructions 64 bits (87 instructions pour le moment)

10 registres 64 bits d'usage générique + 1 PC + 1 Pointeur de pile

Architecture de eBPF : bytecode

Jeu d'instruction RISC conçu :

- pour être facilement généré depuis un compilateur C/Rust/P4...
- pour être compilé en code natif
(JIT présent dans x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm)

Instructions 64 bits (87 instructions pour le moment)

10 registres 64 bits d'usage générique + 1 PC + 1 Pointeur de pile

Pile d'exécution d'une taille limitée à 512 octets.

Fonctions helpers : principes

Fonctions définies dans le noyau

Fonctions helpers : principes

Fonctions définies dans le noyau

Appelables par un programme BPF

- Interagir avec des objets du noyau
- Effectuer des opérations impossibles à faire en eBPF

Fonctions helpers : principes

Fonctions définies dans le noyau

Appelables par un programme BPF

- Interagir avec des objets du noyau
- Effectuer des opérations impossibles à faire en eBPF

5 arguments maximum (r1-r5) :

- Mapping byte code \Rightarrow calling conventions System V AMD64
- Gain de performances

Fonctions helpers : principes

Fonctions définies dans le noyau

Appelables par un programme BPF

- Interagir avec des objets du noyau
- Effectuer des opérations impossibles à faire en eBPF

5 arguments maximum (r1-r5) :

- Mapping byte code \Rightarrow calling conventions System V AMD64
- Gain de performances

Les helpers appelables peuvent varier selon le type de programme eBPF

Fonctions helpers : exemple

Listing 1: Programme BPF

```
static inline int rewrite(struct __sk_buff *skb) {  
    int cgroup = bpf_get_cgroup_classid(skb)  
}
```

Listing 2: Déclaration d'un helper

```
BPF_CALL_1(bpf_get_cgroup_classid, const struct sk_buff *, skb) {  
    return task_get_classid(skb);  
}  
  
static const struct bpf_func_proto bpf_get_cgroup_classid_proto = {  
    .func          = bpf_get_cgroup_classid,  
    .gpl_only      = false,  
    .ret_type       = RET_INTEGER,  
    .arg1_type      = ARG_PTR_TO_CTX,  
};  
  
struct bpf_func_proto :  
    • Validation statique par le verifier  
    • Les arguments passés dans le programme  
      BPF correspondent aux arguments attendus
```


Maps

Stockage clés \Rightarrow valeur de taille fixée dans le noyau :

- persistance de données en mémoire
- partage de données entre des programmes utilisateur et eBPF
- partage de données entre des programmes eBPF

Maps

Stockage clés \Rightarrow valeur de taille fixée dans le noyau :

- persistance de données en mémoire
- partage de données entre des programmes utilisateur et eBPF
- partage de données entre des programmes eBPF

Plusieurs implémentation :

- Génériques : `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_ARRAY` ...
- Spécifiques : `BPF_MAP_TYPE_CGROUP_ARRAY` ...

Maps

Stockage clés \Rightarrow valeur de taille fixée dans le noyau :

- persistance de données en mémoire
- partage de données entre des programmes utilisateur et eBPF
- partage de données entre des programmes eBPF

Plusieurs implémentation :

- Génériques : `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_ARRAY` ...
- Spécifiques : `BPF_MAP_TYPE_CGROUP_ARRAY` ...

Interface :

- Utilisateur (`man bpf`) : syscalls `create`, `update`, `delete`, `lookup`, `iterate`
- eBPF : helpers génériques (`update`, `delete`, `lookup`), et spécifiques (`bpf_current_task_under_cgroup()`) ...

Verifier

Vérifie **au chargement** la validité du code BPF

Verifier

Vérifie **au chargement** la validité du code BPF

Pas de préemption des programmes (sauvegarde de contexte) :

- ⇒ Assurer la terminaison : boucles/récursivité interdite (non turing complet)
- ⇒ Temps d'exécution courts : nombre d'instructions limité à 4096

Verifier

Vérifie **au chargement** la validité du code BPF

Pas de préemption des programmes (sauvegarde de contexte) :

- ⇒ Assurer la terminaison : boucles/récursivité interdite (non turing complet)
- ⇒ Temps d'exécution courts : nombre d'instructions limité à 4096

Pas de buffer overflow :

- ⇒ Vérifie que les accès effectués à la mémoire sont sûrs
- ⇒ Tracer et valider le type de variables (arguments des helpers)

Verifier

Vérifie **au chargement** la validité du code BPF

Pas de préemption des programmes (sauvegarde de contexte) :

- ⇒ Assurer la terminaison : boucles/récurtivité interdite (non turing complet)
- ⇒ Temps d'exécution courts : nombre d'instructions limité à 4096

Pas de buffer overflow :

- ⇒ Vérifie que les accès effectués à la mémoire sont sûrs
- ⇒ Tracer et valider le type de variables (arguments des helpers)

Interdiction des accès à des variables non initialisées :

- ⇒ Vérifie les accès à la pile
- ⇒ Vérifié les accès aux registres

Chaine d'outillage

Programme C \xrightarrow{LLVM} programme ELF \xrightarrow{Loader} SYSCALL

Compilation : LLVM

Loader :

- bcc python
- perf trace
- iproute2 réseaux
- Noyau linux samples/bpf/

LLVM : Contraintes C

Pas de bibliothèques partagées autre que dans des .h

Pas de bibliothèques standard :

- Certaines fonctions built-in sont autorisées (`memset()`, `memcpy()`, `memmove()`, `memcmp()`)
- **lorsque la taille est constante.**

Boucles déroulées :

```
#pragma clang loop unroll(full)
for(i=0; i<bufsz >> 1; i++)
    checksum += *buf++;
```

Pas de variables globales.

Pas de strings/tableaux constants

LLVM : Différences

Plusieurs programmes BPF peuvent être dans un seul fichier C

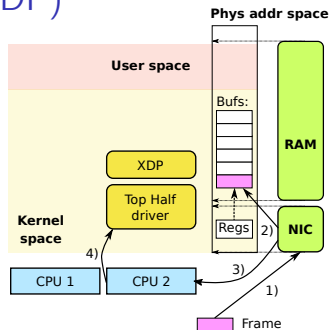
- Communication inter-programmes via maps

Tail calls :

- Appels « récursifs » sans retour :
 - ▶ Rediriger le flot d'exécution d'un 1er programme BPF vers un 2eme
 - ▶ Le nouveau programme écrase le contexte d'exécution (pile) de l'ancien
 - ▶ limité à 32 appels
- Découper un traitement en sous-traitements

Hook réseau : eXpress Data Path (XDP)

Callback : exécuter un programme eBPF sur un paquet reçu AVANT la pile réseau

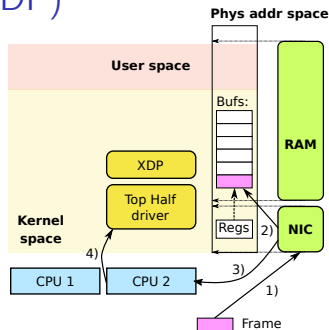


Hook réseau : eXpress Data Path (XDP)

Callback : exécuter un programme eBPF sur un paquet reçu AVANT la pile réseau

Action pouvant être effectuée par un programme XDP BPF :

- DROP, PASS
- TX/REDIRECT : rerouter le paquet en dehors du NIC



Hook réseau : eXpress Data Path (XDP)

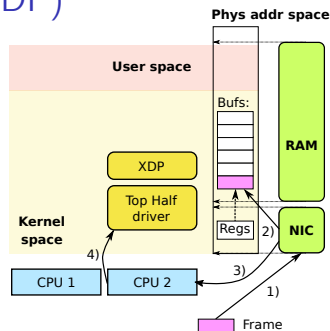
Callback : exécuter un programme eBPF sur un paquet reçu AVANT la pile réseau

Action pouvant être effectuée par un programme XDP BPF :

- DROP, PASS
- TX/REDIRECT : rerouter le paquet en dehors du NIC

Points d'entrées :

- Offload : dans la carte réseau
- Native : dans le driver après la réception du paquet
- Generic : après le driver



Hook réseau : eXpress Data Path (XDP)

Callback : exécuter un programme eBPF sur un paquet reçu AVANT la pile réseau

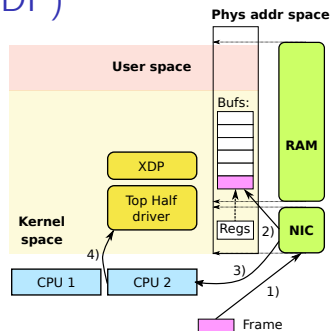
Action pouvant être effectuée par un programme XDP BPF :

- DROP, PASS
- TX/REDIRECT : rerouter le paquet en dehors du NIC

Points d'entrées :

- Offload : dans la carte réseau
- Native : dans le driver après la réception du paquet
- Generic : après le driver

Partage du matériel: SR-IOV ou tail calls



Hook réseau : eXpress Data Path (XDP)

Callback : exécuter un programme eBPF sur un paquet reçu AVANT la pile réseau

Action pouvant être effectuée par un programme XDP BPF :

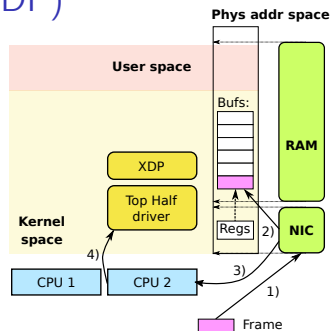
- DROP, PASS
- TX/REDIRECT : rerouter le paquet en dehors du NIC

Points d'entrées :

- Offload : dans la carte réseau
- Native : dans le driver après la réception du paquet
- Generic : après le driver

Partage du matériel: SR-IOV ou tail calls

Cas d'utilisation: firewalling, forwarding, load-balancing



Conclusion

Les techniques utilisées pour améliorer les performances réseau fonctionnent en court-circuitant des couches logicielles :

=> Kernel bypass : couche noyau

=> eBPF/XDP : pile réseau, application

Conclusion

Les techniques utilisées pour améliorer les performances réseau fonctionnent en court-circuitant des couches logicielles :

- => Kernel bypass : couche noyau

- => eBPF/XDP : pile réseau, application

Les techniques de Kernel bypass sont plus flexibles mais nécessitent de redévelopper un sous-ensemble de la pile réseau

Conclusion

Les techniques utilisées pour améliorer les performances réseau fonctionnent en court-circuitant des couches logicielles :

- => Kernel bypass : couche noyau

- => eBPF/XDP : pile réseau, application

Les techniques de Kernel bypass sont plus flexibles mais nécessitent de redévelopper un sous-ensemble de la pile réseau

eBPF & XDP peut être utilisé pour enrichir la pile réseau existante