Principes des systèmes d'exploitation et programmation concurrente

Introduction

UFR IM^{2}AG M1 MIAGE & M1 MEEF NSI 2023-2024

Renaud Lachaize

https://m1-miage-os.gitlab.io

https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=350

Crédits et remerciements

- **Le contenu de ce support de cours est partiellement inspiré, voire emprunté aux travaux d'autres personnes :**

  - Vincent Danjean, Sacha Krakowiak, Arnaud Legrand, Vania Marangozova-Martin (UFR IM^{2}AG)
  - David Mazières (Stanford University)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
  - Gernot Heiser (UNSW)

  - Ouvrages de référence (voir détails sur la page web) :
    - Silberschatz et al. Operating systems concepts with Java.
    - Tanenbaum. Modern operating systems.
    - Bryant and O'Hallaron. Computer systems: a programmer's perspective.

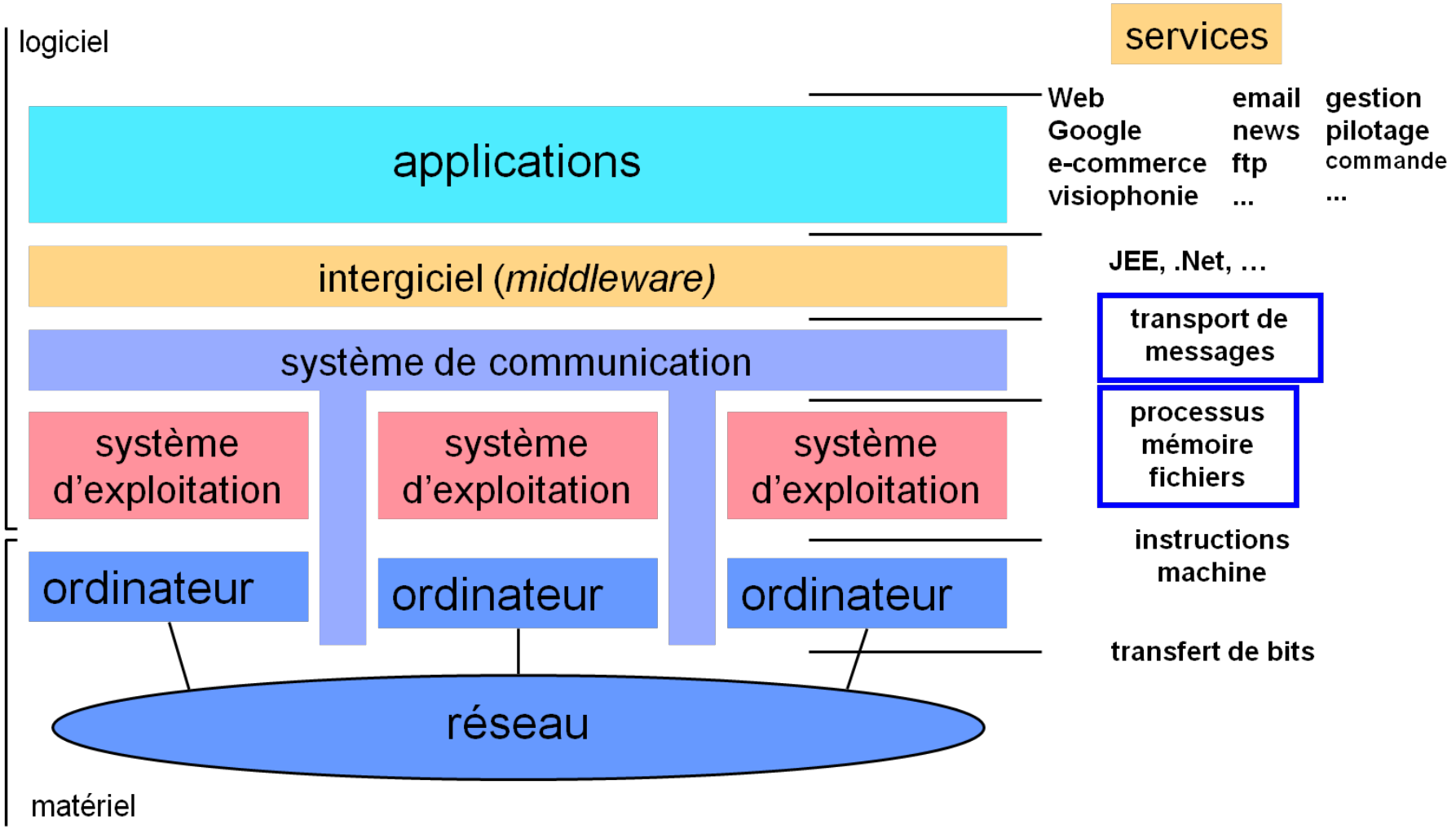
Objectifs du cours

- **Comprendre les principaux concepts et mécanismes des systèmes d'exploitation, ainsi que leurs éléments de réalisation**
- **Mieux comprendre les interactions entre les couches matérielles et logicielles**
- **Mieux comprendre les interactions entre couches logicielles**
  - exemples : environnements Java, Python
- **Maîtriser la programmation concurrente**
- **En Résumé, acquérir des connaissances nécessaires pour :**
  - Programmer (de manière fiable et efficace) et structurer des logiciels complexes
  - Appréhender des domaines connexes (intergiciels, systèmes répartis ...)

Plan

- Introduction : position et rôle d'un système d'exploitation
- Notions de base: processus, protection et partage de ressources
- Threads
- Machines virtuelles

Composants d'un système informatique



Qu'est-ce qu'un système d'exploitation ?

- **Au sens large : les couches logicielles intermédiaires entre le matériel (processeurs et périphériques) et les applications**

- **Plus précisément : les couches logicielles de plus bas niveau sur une machine**
  - Noyau
  - Bibliothèques de base et quelques programmes utilitaires

- **Deux rôles principaux et complémentaires**
  - Adaptation d'interface
  - Gestion de ressources

- **Présent sur la plupart des équipements munis d'un processeur**
  - Serveurs, ordinateurs personnels, tablettes, téléphones, carte à puces ...

Adaptation d'interface

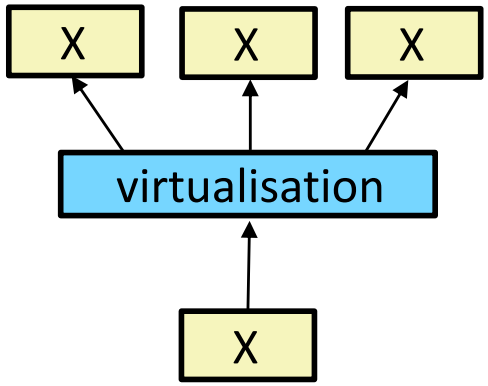
- **But : simplifier l'écriture et l'exécution (lancement, débogage, administration ...) de programmes sur une machine**

- **Pour cela, le système d'exploitation, décharge les programmeurs/utilisateurs d'applications de la gestion de certains aspects complexes :**
  - Les interactions avec le matériel (interfaces de très bas niveau, diversité et hétérogénéité des périphériques)
  - Les limites physiques des machines et leurs variations d'une machine à l'autre (taille de la mémoire vive, nombre de processeurs/cœurs)
  - Le partage des ressources entre applications/utilisateurs (cf. gestion de ressources)

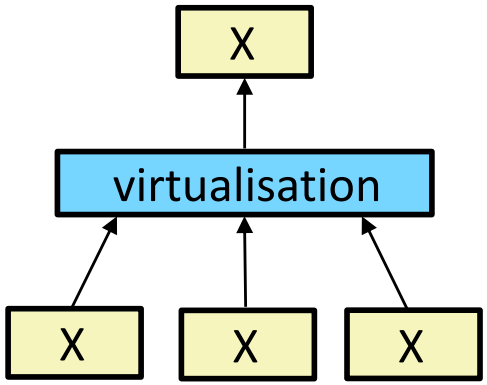
- **Idée générale : masquer les détails complexes derrière des abstractions, souvent de plus haut niveau**

Virtualisation

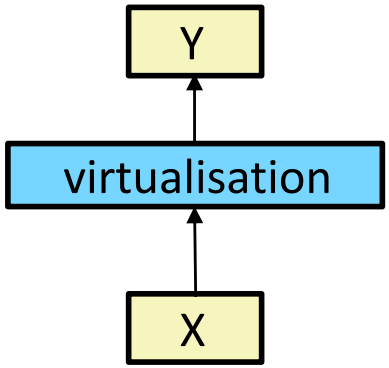
- **Idée générale : abstraire les ressources sous-jacentes**
- **Il existe différentes formes de virtualisation**
  - (qui peuvent être combinées)



(a) Multiplexage



(b) Agrégation



(c) Émulation

Inspired (with modifications) from books and slides by: J. Salzer & M.F. Kaashoek, E. Bugnion & J. Nieh & D. Tsafrir, S. Krakowiak

Gestion de ressources

- **But: permettre un fonctionnement stable/sécurisé/efficace/équitable (...) des applications**
  - malgré leur exécution concurrente et des demandes/besoins potentiellement contradictoires
  - malgré d'éventuels bogues/pannes/tentatives de malveillance

- **Plusieurs types de ressources**
  - Physiques : processeur(s), mémoire, périphériques, énergie ...
  - Logiques : programmes, données, communications ...
- **Plusieurs aspects**
  - Allocation
  - Protection
  - Partage (spatial et temporel)

- **Idée générale : la gestion de chaque ressource/aspect repose sur des mécanismes (génériques) et sur des politiques (configurables)**

Fonctions d'un système d'exploitation (1/2)

- **Gestion d'activités**
  - Déroulement de l'exécution
  - Evénements particuliers (matériel, erreurs, interactions entre activités ...)
  - Abstraction clé : processeur => processus

- **Gestion d'information**
  - Accès dynamique (exécution)
  - Partage
  - Conservation permanente
  - Abstractions clés :
    - Mémoire principale => mémoire virtuelle
    - Disque => fichiers

Fonctions d'un système d'exploitation (2/2)

- **Gestion des communications**
  - Interface avec l'utilisateur
  - Interface avec les périphériques d'entrées/sorties
  - Interactions avec d'autres machines via le réseau
  - Abstraction clé : périphérique => flot d'entrées/sorties

- **Protection**
  - Pour les ressources matérielles et les informations
  - Notions clés : domaines de protection et politiques associées

Interfaces d'un système d'exploitation (1/2)

- **Un système exporte typiquement deux types d'interfaces**
  - Une interface de commande
  - Une interface programmatique

- **Interface de commande**
  - Conçue pour les interactions avec les utilisateurs
  - Diverses formes : interfaces textuelles (shells), interfaces graphiques
  - Composée d'un ensemble de commandes
    - Exemple : supprimer un fichier
    - Version textuelle (shell Unix) : **`rm myfile.txt`**
    - Version graphique : faire glisser l'icône du fichier vers la corbeille

Interfaces d'un système d'exploitation (2/2)

- **Interface programmatique**
  - Utilisée/invoquée par les applications qui s'exécutent sur le système
    - Y compris les programmes qui implémentent les interfaces de commande
  - Composée d'un ensemble de procédures/fonctions
    - Bibliothèques
    - Appels système (voir détails un peu plus loin)
  - Définie à deux niveaux
    - Au niveau du code source : *Application Programming Interface* (API) – Exemple : norme POSIX
    - Au niveau du code binaire : *Application Binary Interface* (ABI) – Exemple : ABI Linux x86 32bits

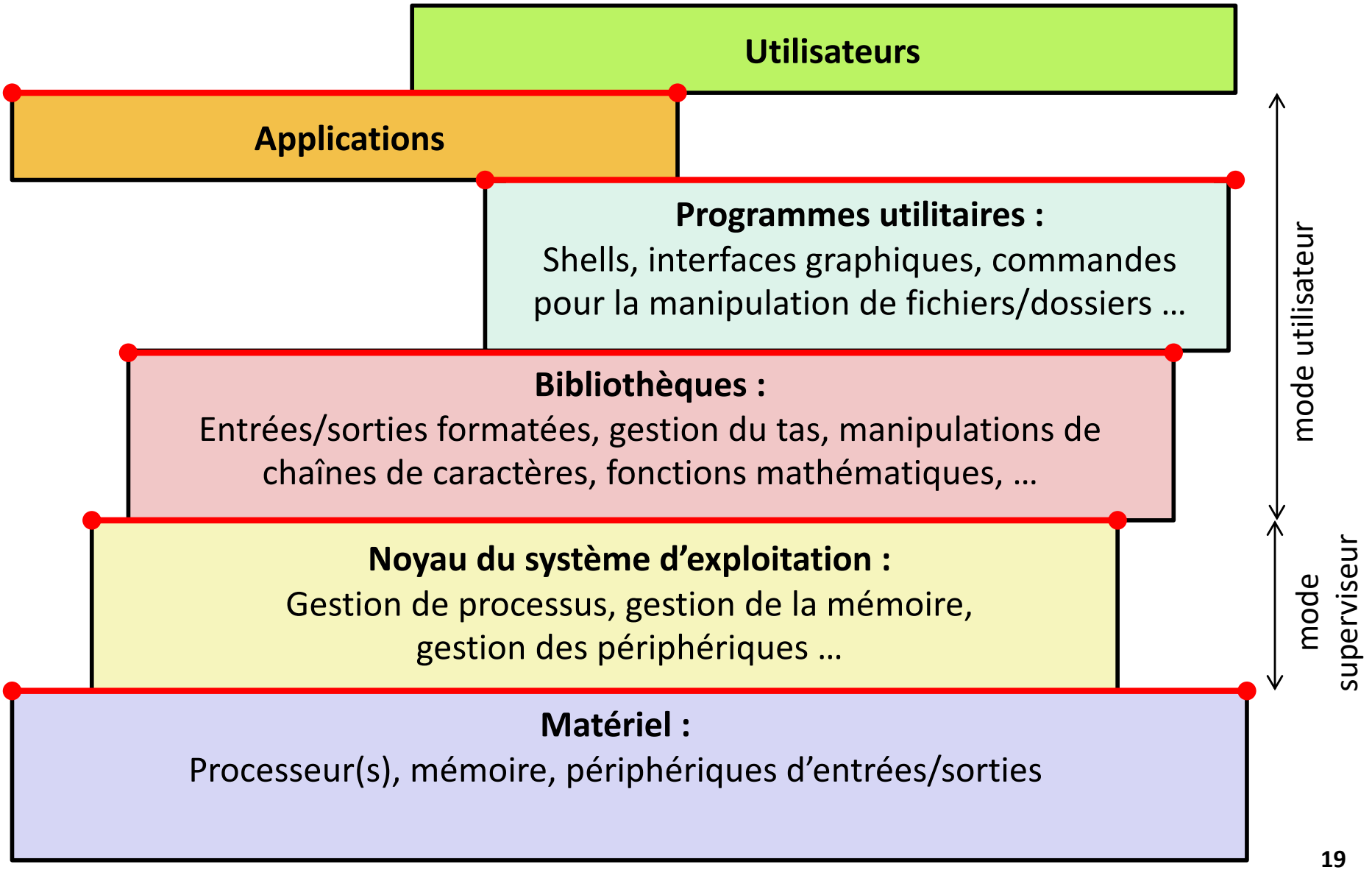
Un peu d'histoire

- **Premiers systèmes d'exploitation**
  - De simples bibliothèques de code permettant de simplifier l'écriture des programmes
  - Par exemple, pour la gestion des périphériques

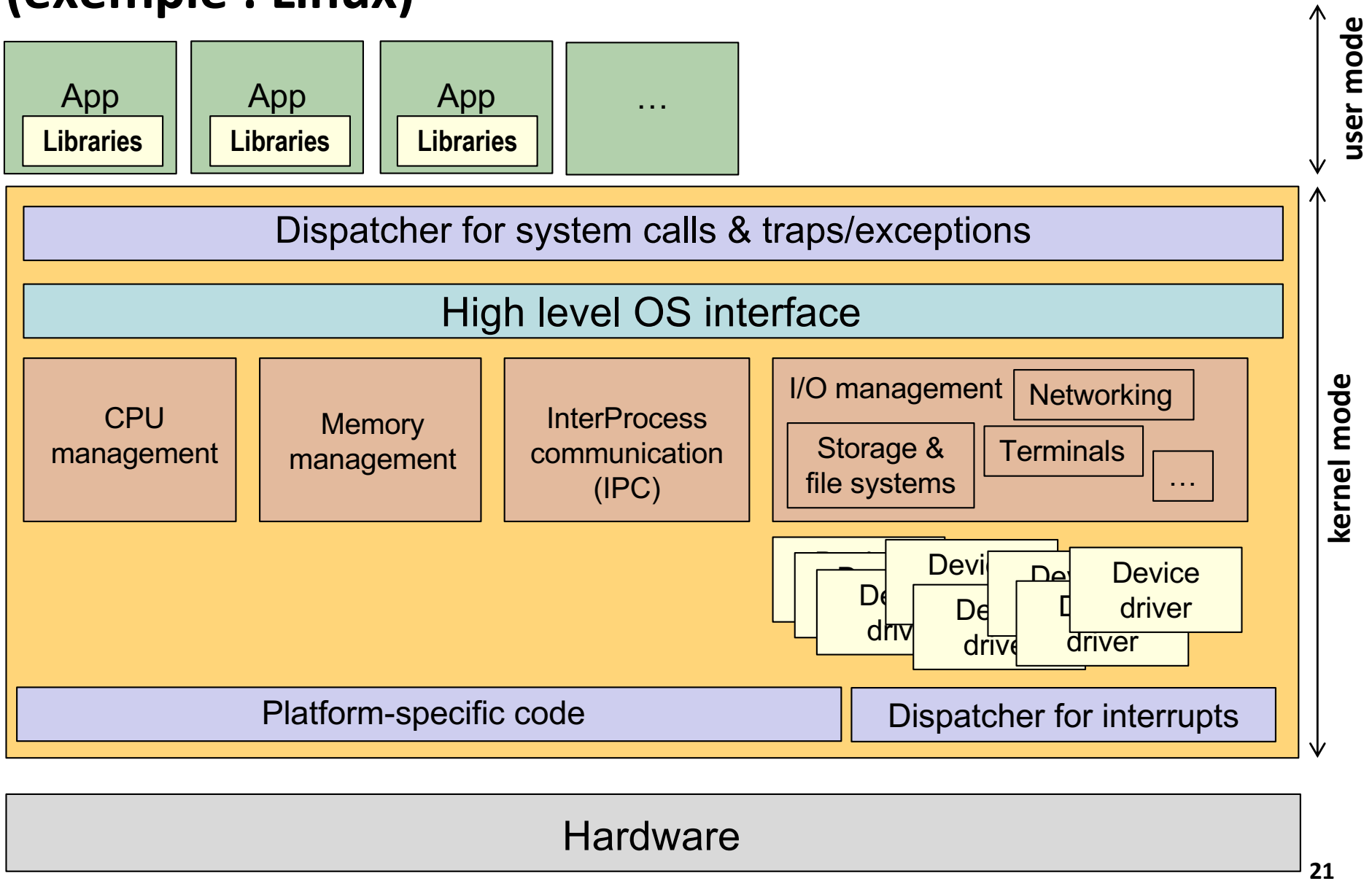
- **2ème étape : prise en compte du rôle central du système d'exploitation**
  - Un utilisateur/une application ne doit pas rendre le système global inutilisable
  - Introduction d'une nouvelle interface (matérielle et logicielle) pour protéger le code et les données critiques du système

- **3ème étape : amélioration de l'utilisation des ressources et de la réactivité**
  - Ne pas nécessairement attendre la terminaison d'une application avant d'en lancer une autre
  - Empêcher les interférences entre différentes applications

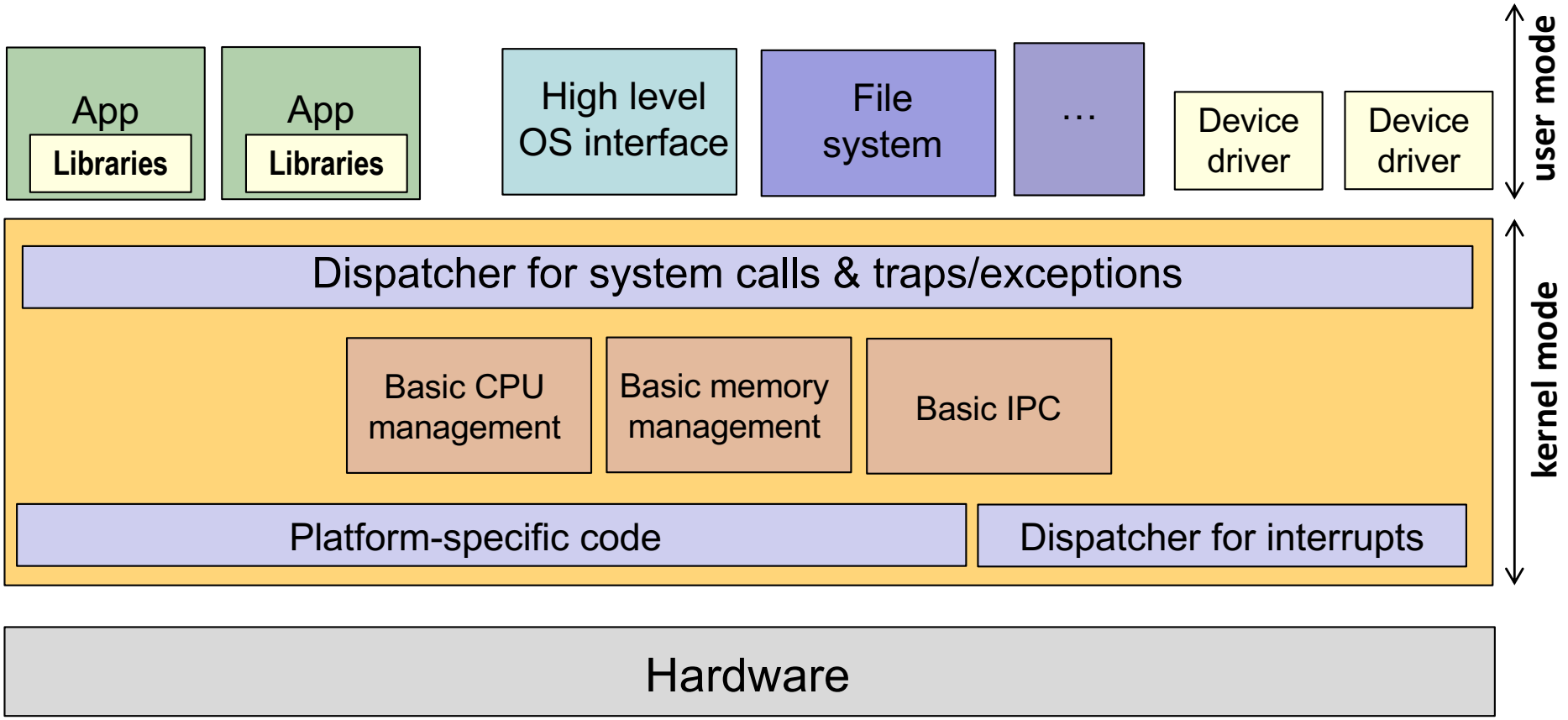
Structure typique d'un système d'exploitation



Structure typique d'un noyau monolithique (exemple : Linux)



Structure de type micro-noyau (exemple : L4)



Une abstraction clé des systèmes d'exploitation : les processus (1/2)

- **Un processus est une abstraction associée à une instance de programme en cours d'exécution**

- **Cette abstraction sert essentiellement à virtualiser un processeur (CPU)**
  - Une machine physique ne dispose que de quelques CPU (ou même d'un seul)
  - ... mais le système d'exploitation fournit l'illusion d'une capacité quasi-infinie de CPU « logiques » (un par processus), indépendants les uns des autres
  - Cette abstraction est aussi très utile pour le contrôle de l'exécution d'un programme, et donc plus globalement pour la gestion des ressources

Une abstraction clé des systèmes d'exploitation : les processus (2/2)

- **Un processus est essentiellement constitué des éléments suivants :**
  - Un contexte d'exécution
    - Etat courant de la machine : ensemble des valeurs stockées dans les registres du processeur, dont notamment le compteur programme (PC) et le pointeur de pile (SP)
    - Une pile d'exécution
  - Un espace de mémoire (ou « espace d'adressage », ou encore «une mémoire virtuelle »)
  - Un état courant :
    - En cours d'exécution
    - Prêt – en attente d'un CPU
    - Bloqué (cause ?)
  - D'autres informations nécessaires pour le système
    - Exemples : liste des fichiers ouverts, autorisations ...

Espace mémoire d'un processus
Vue simplifiée

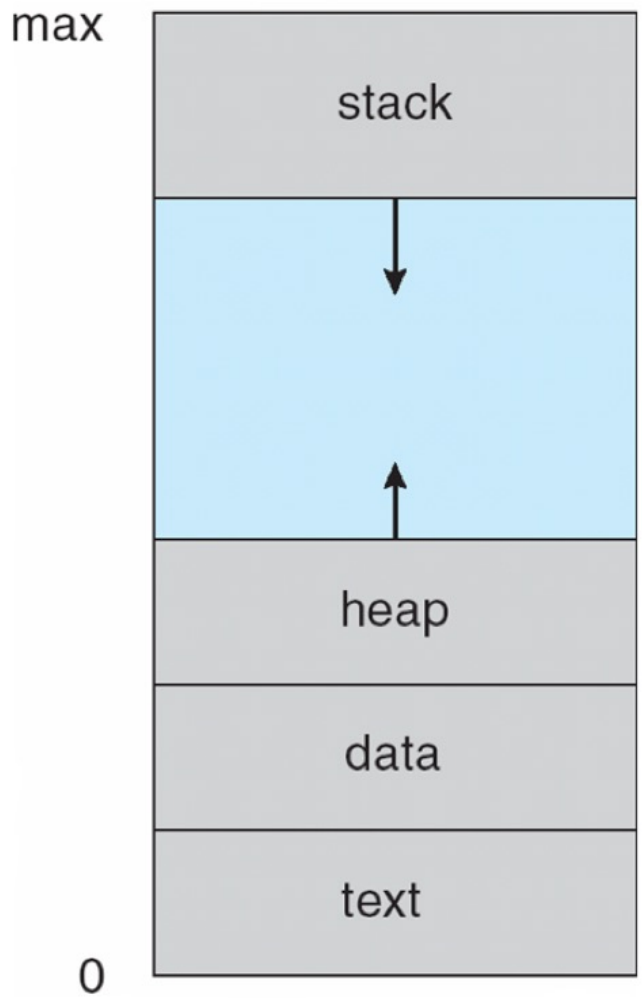


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Espace mémoire d'un processus
Vue détaillée

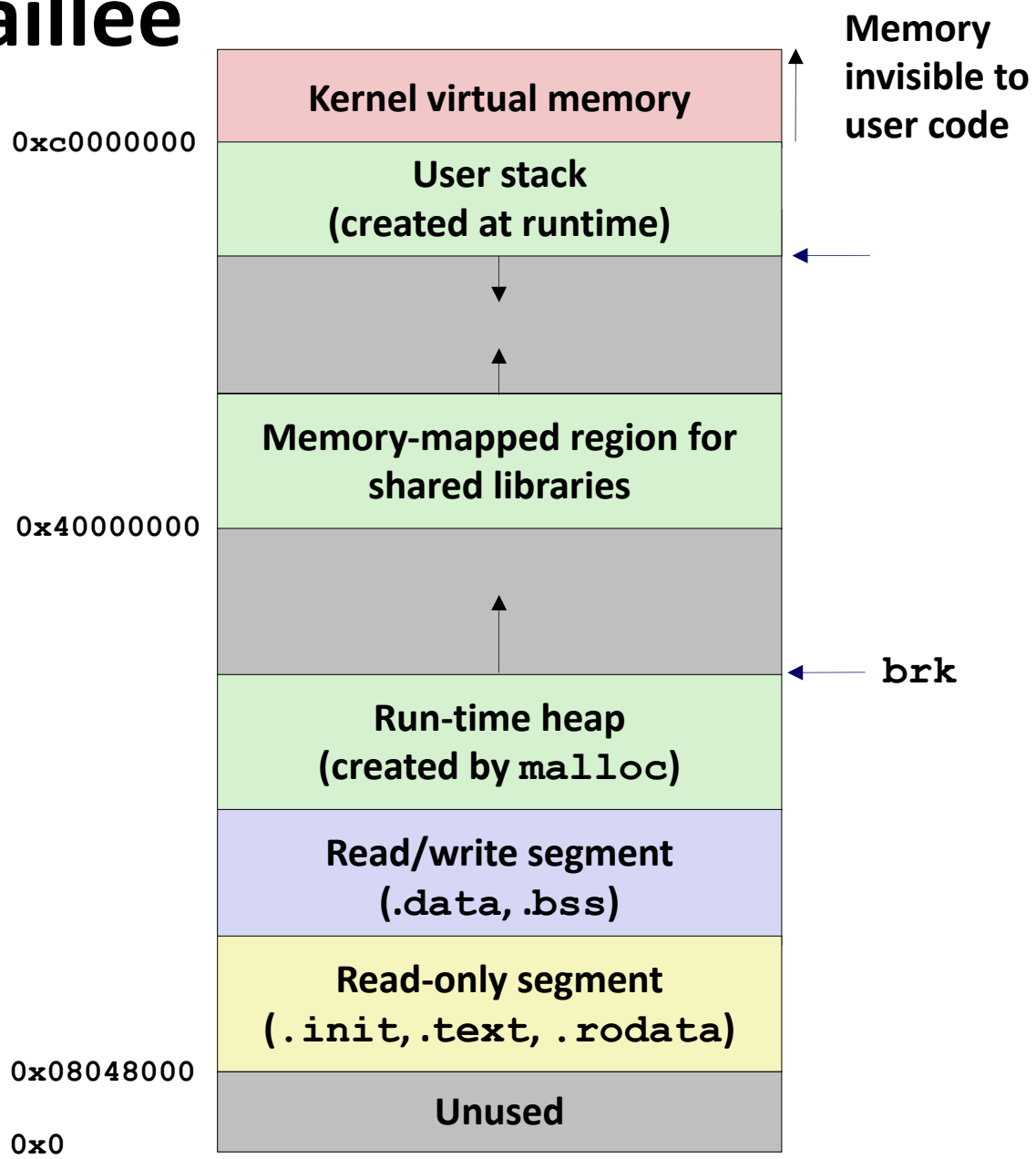


Schéma extrait de :
Bryant & O'Hallaron, Computer systems: a programmer's perspective, Prentice Hall, 2003.

Techniques clés pour la protection

- **Objectif général : empêcher des processus incorrects (code bogué ou volontairement malveillant) d'impacter le système d'exploitation ou d'autres processus**

- **Préemption**
  - Principe : attribuer une ressource à un processus et lui reprendre plus tard (si besoin, de force) si elle est requise pour autre chose
  - Exemple : contrôle de l'attribution du CPU
- **Interposition**
  - Principe : le système est un passage obligé entre les applications et les ressources ... et peut ainsi contrôler la légitimité de chaque demande applicative
  - Exemple : appels système

Support matériel pour la protection (1/2)

- **Un processeur moderne a (au moins) deux modes d'exécution**
  - Mode privilégié (ou « mode superviseur », ou encore « mode noyau/kernel »)
  - Mode non-privilégié (ou « mode utilisateur »)
- **Le noyau s'exécute en mode privilégié**
- **Les applications (et les bibliothèques) s'exécutent en mode utilisateur**

- **Le code et les données critiques/privilégiées (y compris les informations qui définissent ces privilèges) ne doivent être accessibles qu'en mode privilégié**
  - Ces règles sont vérifiées par le matériel

Support matériel pour la protection (2/2)

- **Le code privilégié peut demander (librement) au processeur de basculer l'exécution vers du code en mode non-privilégié**
- **Les transitions du mode non-privilégié vers le mode privilégié sont contrôlées . Il y a seulement deux façons d'opérer un basculement dans ce sens :**
  - **Appels système et exceptions (synchrones) :**
    - Transition synchrone entre le mode utilisateur et le mode noyau, liée à la dernière instruction exécutée
    - Causée volontairement par une instruction spéciale (cas d'un appel système)
    - Ou bien causée implicitement par une erreur (cas d'une exception) - exemple : erreur d'accès mémoire
  - **Interruptions (asynchrones) :**
    - Le basculement est lié à une cause externe, indépendante du code utilisateur en cours d'exécution
    - Typiquement, la cause est un signal matériel envoyé par un périphérique

Appels système (1/5)

- **Les applications peuvent invoquer des services fournis par le noyau**
  - En utilisant une instruction spéciale qui déclenche un basculement de mode (« *trap* »)
  - (Le basculement peut aussi être causé implicitement par une instruction erronée ou interdite en mode utilisateur)
  - L'exécution du code utilisateur est alors suspendue – le processeur se met à exécuter une procédure spéciale définie par le noyau (« *trap handler* »), qui :
    - Détermine la cause du basculement en analysant les paramètres passés à l'instruction de trap (ou les circonstances de l'erreur)
    - Appelle la sous-procédure de traitement concernée
      - En cas d'erreur, le traitement consiste souvent à tuer le processus

Appels système (2/5)

- **Intérêt : effectuer des opérations pour les applications, sans leur fournir un accès direct aux ressources (principe d'interposition)**
  - Un appel système ressemble à l'appel d'une fonction de bibliothèque mais déclenche un changement de mode d'exécution
- **Le noyau fournit une interface précise et vaste pour les appels systèmes (*system call interface*)**
  - Définie aux niveau de l'API (code source) et de l'ABI (code binaire)
  - Plusieurs centaines de procédures dans les noyaux modernes
  - Le code utilisateur prépare les arguments et exécute l'instruction trap
  - Le noyau analyse les paramètres et vérifie si l'opération est autorisée pour le processus appelant, effectue l'opération puis déclenche un basculement de retour vers le code utilisateur

Appels système (3/5)

- **Illustration avec l'appel `open` (pour ouvrir un fichier)**

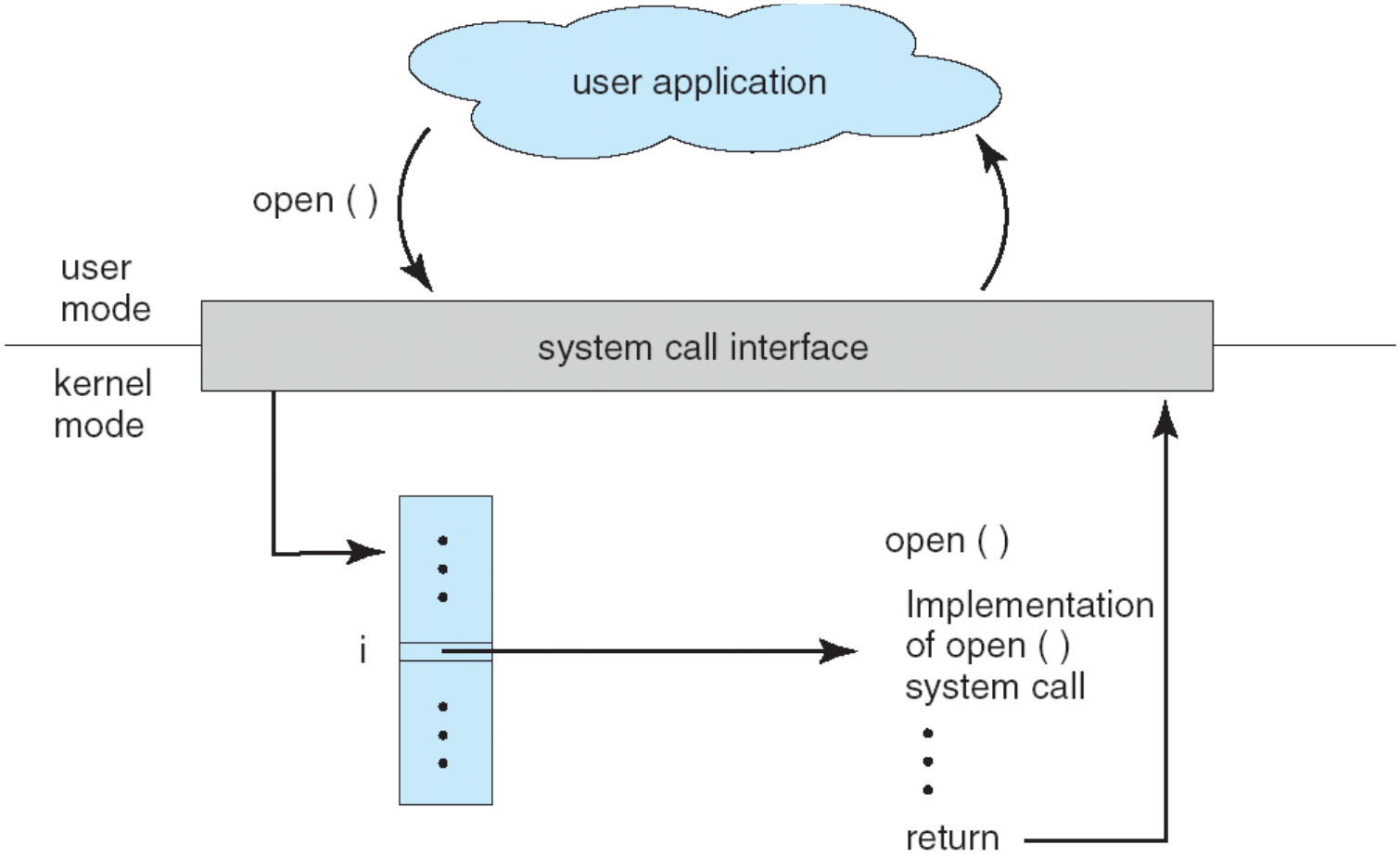


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Appels système (4/5)

- **Le code applicatif peut invoquer directement des appels système**
- **En général, la plupart des appels systèmes sont effectués via des bibliothèques**
- **... et beaucoup des fonctions de bibliothèques (notamment la bibliothèque C standard – libC) s'appuient sur des appels système**

- **Exemple (Unix/Linux) : E/S formatées**
  - **`printf`**/**`fprintf`** implémentées sous forme de code utilisateur qui invoque le noyau via l'appel système **`write`**
  - Idem pour **`scanf`**/**`fscanf`** avec l'appel **`read`**

Appels système (5/5)

- **Exemple : E/S formatées (suite)**
  - **`printf/fprintf`** :
    - Génère une chaîne de caractères (octets) à partir d'une description de format
    - A les mêmes privilèges que l'application
    - Invoque l'appel système **`write`**
  - **`write`** :
    - Ecrit une séquence d'octets dans un fichier (ou sur la console)
    - Dispose des privilèges du noyau (qui autorisent la manipulation des ressources : fichiers/console)

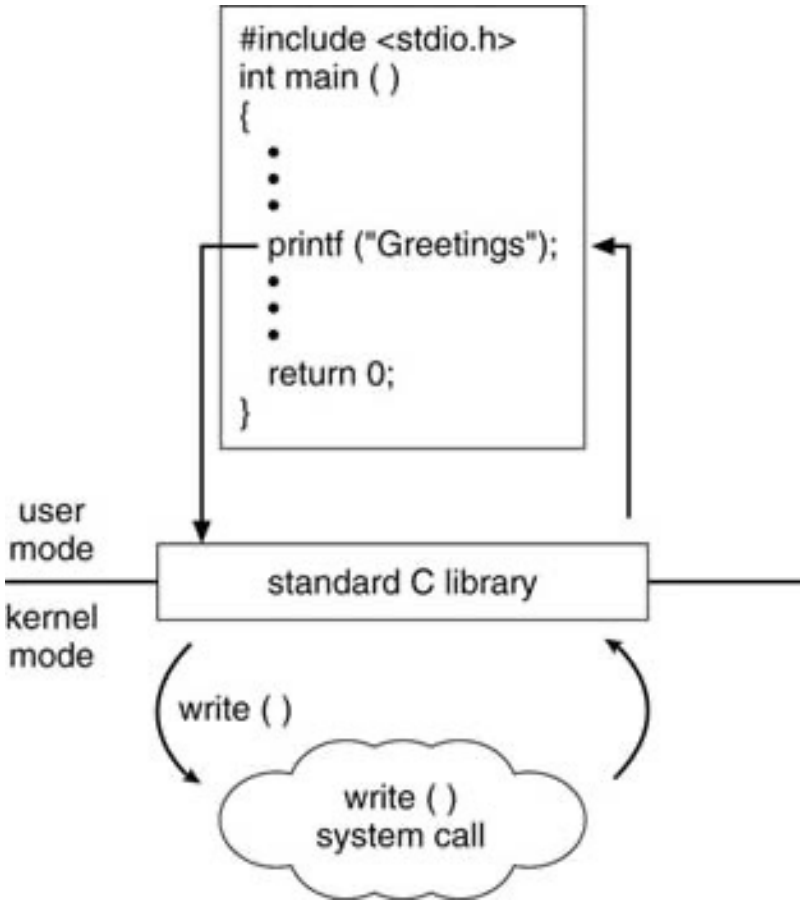


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Contrôle/préemption du CPU (1/2)

- **Problème :**
  - Pour garantir le contrôle de la machine, lorsque le système bascule vers du code utilisateur, il doit avoir la garantie de pouvoir reprendre le contrôle du CPU au bout d'un temps borné
  - Que faire si le code d'un processus n'effectue pas spontanément d'appels système ?
    - (processus bogué, malveillant ou simplement très long)

- **Solution :**
  - Utiliser le mécanisme d'interruptions matérielles couplé à un périphérique « *timer* »
  - Le noyau programme le timer pour générer des interruptions périodiques (par exemple, période = 10 ms)
    - Le contrôle du timer est une opération privilégiée

Contrôle/préemption du CPU (2/2)

- **Le noyau configure le processeur pour exécuter une procédure particulière (traitant/*handler*) à exécuter lors de l'arrivée de chaque interruption timer**
  - Ce traitant s'exécute en mode privilégié
  - Le code utilisateur n'a pas les privilèges nécessaires pour ignorer/désactiver les interruptions, ni pour redéfinir le traitant
  - On a donc la garantie qu'une procédure du noyau sera exécutée au moins une fois par période du timer
  - Le traitant peut décider de redonner immédiatement la main au processus interrompu ou bien de suspendre ce processus et d'attribuer le CPU à un autre
- **Résultat :**
  - Un processus ne peut pas monopoliser le CPU avec une boucle infinie
  - Au pire, il n'utilisera qu'une fraction du temps CPU disponible

Partage du processeur (1/2)

- **L'ordonnanceur (*scheduler*) est un composant du noyau chargé de décider de l'attribution des CPU (une décision par CPU)**

- **À quel moment l'ordonnanceur est-il invoqué ?**
  - Périodiquement, suite à chaque interruption matérielle du timer
  - Ponctuellement, en réaction à certains appels système
    - Terminaison d'un processus (**`exit`**)
    - Restitution spontanée du CPU par un processus (**`yield`**, **`sleep`** ...)
    - Action bloquante
    - Création d'un nouveau processus avec une priorité supérieure
    - ...
  - Ponctuellement, suite à certaines interruptions matérielles

Partage du processeur (2/2)

- **Lors de chaque invocation de l'ordonnanceur :**
  - Décision : à quel processus P2 attribuer le CPU, en fonction de :
    - la liste des processus prêts à s'exécuter
    - ... et d'une politique d'ordonnancement
  - Sauvegarde du contexte d'exécution du processus interrompu (P1)
    - (Sauf si l'invocation est liée à la terminaison de P1)
  - Injection/restauration du contexte de P2 sur le CPU
  - La séquence ci-dessus est appelée « changement de contexte » (*context switch*)

  - Remarque : à l'issue du changement de contexte, P2 est encore en mode noyau. Le système doit rebasculer en mode utilisateur en utilisant une instruction spéciale (retour d'interruption ou d'appel système – selon le cas)

Changement de contexte (1/2)

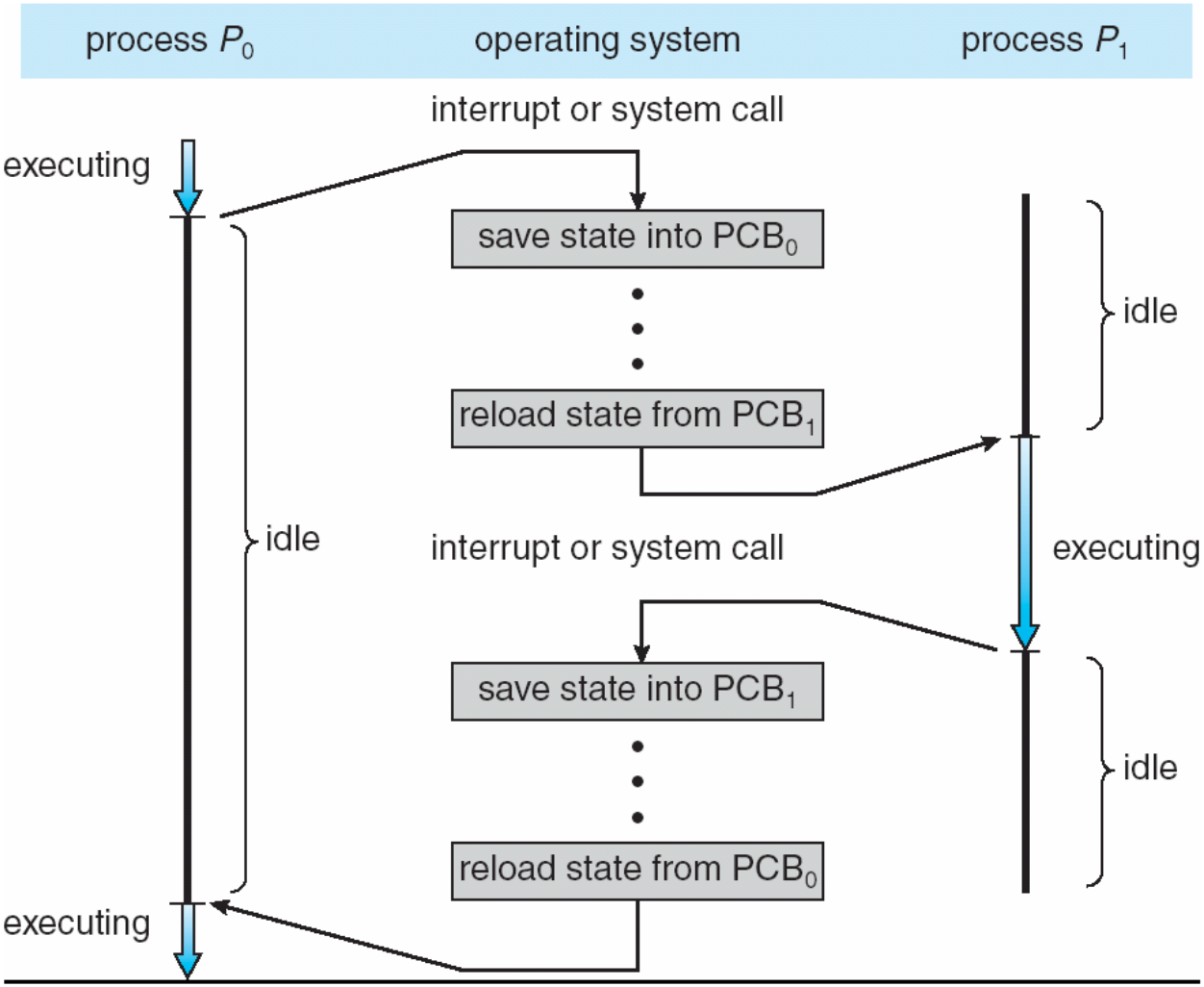
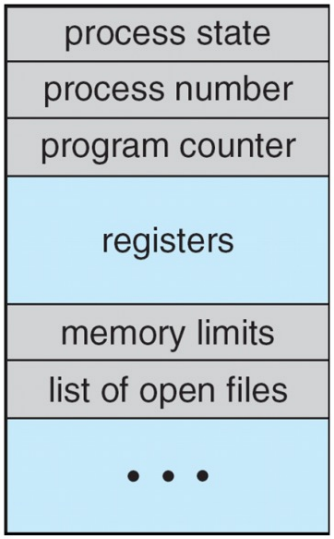


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.



PCB
(process control block)

Changement de contexte (2/2)

- **Remarques :**
  - Les détails d'implémentation sont spécifiques à chaque processeur mais les principes généraux sont les mêmes
  - Les changements de contexte ont un coût non-négligeable et ne doivent pas se produire trop souvent afin de pas dégrader les performances
  - Attention, ne pas confondre :
    - **Changement de contexte** : transition entre deux contextes d'exécution
    - **Changement de mode** : transition entre le mode utilisateur et le mode noyau (ou inversement), dans le même contexte d'exécution
      - Un changement de mode a aussi un coût important

Sauvegarde/restauration de contexte

Un exemple simplifié de code (tiré du mini-système Unix pédagogique "xv6" du MIT – version pour processeur Intel x86 32 bits)

```
# void swtch(struct context *old, struct context *new);
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax.    # put old ptr into eax
  popl 0(%eax)  # %eip. # save the old instruction pointer
  movl %esp, 4(%eax)     # and stack
  movl %ebx, 8(%eax)     # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)
```

```
struct context {
  // Don't need to save
  // %eax, %ecx, %edx,
  // because the
  // x86 convention is
  // that the caller has
  // saved them.
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
  # Load new registers
  movl 4(%esp), %eax  # put new prt into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

Sauvegarde/restauration de contexte

Un exemple simplifié de code (tiré du mini-système Unix pédagogique "xv6" du MIT – version pour processeur RISC-V 64 bits)

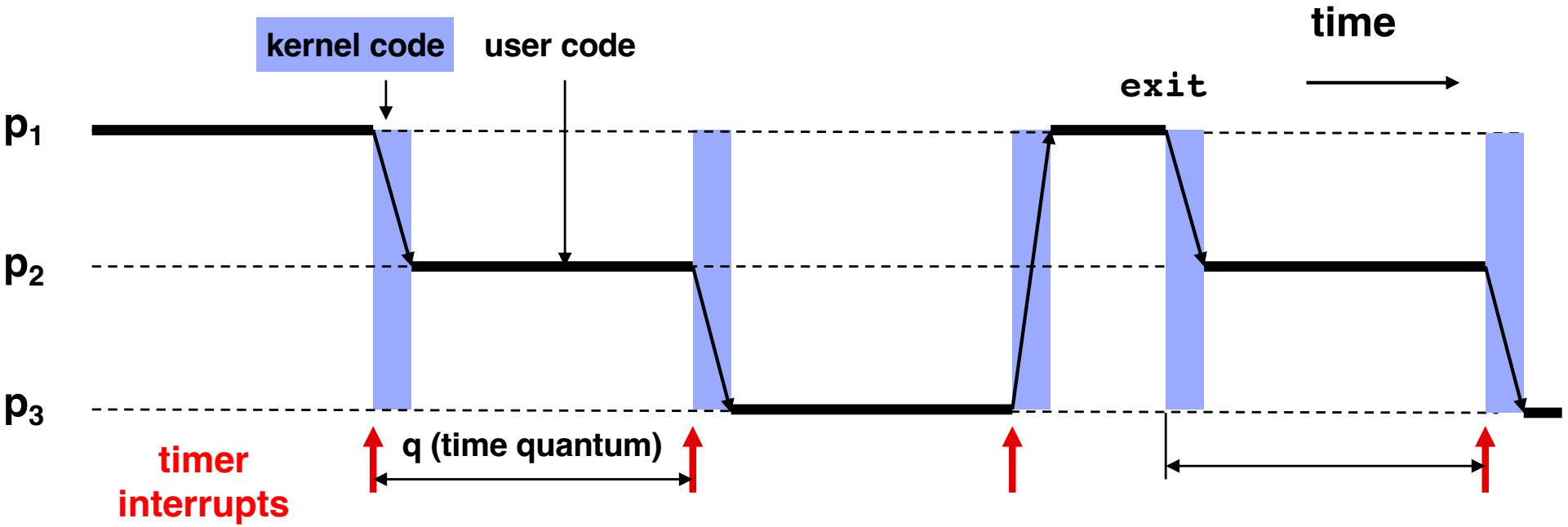
```
struct context {
  uint64 ra;
  uint64 sp;
  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

```
# void swtch(struct context *old, struct context *new);
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  sd ra, 0(a0)
  sd sp, 8(a0)
  sd s0, 16(a0)
  sd s1, 24(a0)
  sd s2, 32(a0)
  sd s3, 40(a0)
  sd s4, 48(a0)
  sd s5, 56(a0)
  sd s6, 64(a0)
  sd s7, 72(a0)
  sd s8, 80(a0)
  sd s9, 88(a0)
  sd s10, 96(a0)
  sd s11, 104(a0)
```

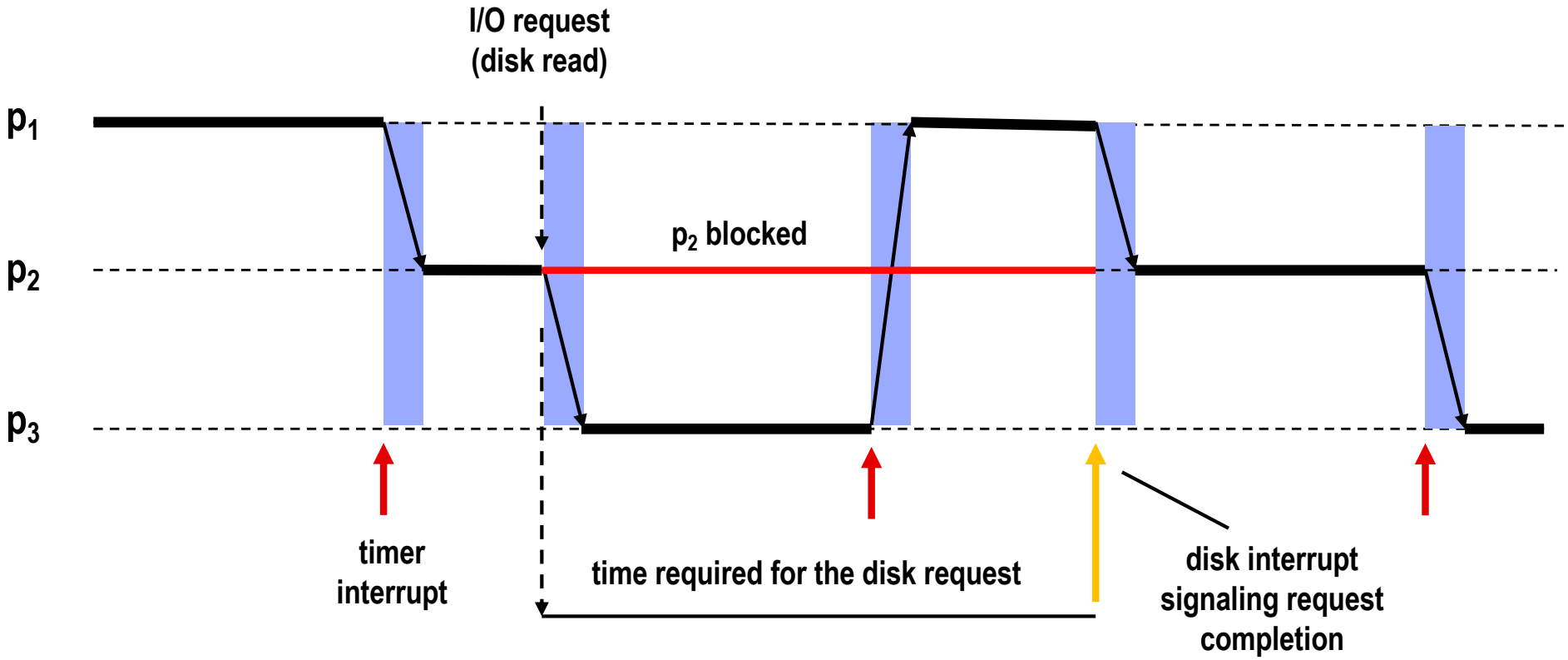
```
# Load new registers
ld ra, 0(a1)
ld sp, 8(a1)
ld s0, 16(a1)
ld s1, 24(a1)
ld s2, 32(a1)
ld s3, 40(a1)
ld s4, 48(a1)
ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)

# Finally return into new ctxt
ret
```

Exemples d'ordonnancement (1/2)



Exemples d'ordonnancement (2/2)



Création de processus

- **Dans les systèmes Unix/Linux, un processus peut créer un autre processus (« fils ») via la fonction `fork`**

- **Le processus ainsi créé est identique au processus père et commence à s'exécuter au retour de l'appel à `fork` (comme s'il avait lui même appelé `fork`)**
- **Seule différence entre les deux processus :**
  - Pour le père, **`fork`** retourne l'identifiant (*PID*) du fils
  - Pour le fils, **`fork`** retourne 0
- **Fonction surprenante : un appel, deux retours d'appel**

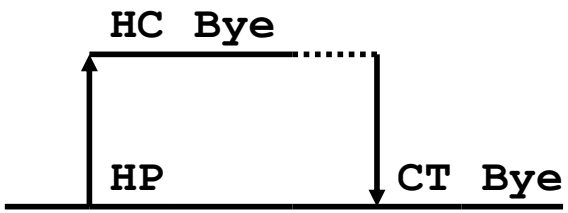
- **Chaque processus à son propre espace de mémoire virtuelle et donc son propre exemplaire des variables**

- **Un processus père peut attendre la fin de l'un ses fils en appelant `wait` ou `waitpid`**

Création de processus - Exemple

```
int main() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Exécution d'un nouveau programme

- **La fonction `execve` permet de changer/remplacer le programme exécuté par un processus**
  - (comme d'autres fonctions de la même famille : **`execl`**, **`execvp`** ...)

- **Si l'appel réussit, `execve` :**
  - Efface et réinitialise l'espace mémoire du processus appelant
    - Code, données, tas, pile ...
  - Charge le code et les données du fichier exécutable indiqué en paramètres
  - ... avec la liste d'arguments et les variables d'environnement spécifiées en paramètre

- **Un appel réussi à `execve` a donc la particularité d'être sans retour !**

Exécution d'un nouveau programme – Exemple

```
int main() {
   int res;
   if (fork() == 0) {
      res = execl("/usr/bin/cp", "cp", "foo", "bar", 0);
      if (res < 0) {
          printf("error: execl failed\n");
          exit(-1);
      }
   }
   wait(&res);
   if (res == 0) {
       printf("copy completed\n");
   }
   exit(0);
}
```

Processus et mémoire virtuelle

- **Rappel : chaque processus dispose de son propre espace de mémoire virtuelle**

- **Protection/isolation mémoire :**
  - Un processus ne peut pas accéder à l'espace mémoire d'un autre processus
  - En fait, un processus ne peut même pas désigner la mémoire d'un autre processus
    - Les adresses de mémoire virtuelle sont contextuelles
    - La même adresse virtuelle correspond à des informations distinctes pour deux processus distincts
  - Au sein d'un processus, le code utilisateur n'a pas accès au code et aux données du noyau

Interactions entre processus

- **Comment permettre à des processus d'interagir malgré l'isolation mémoire ?**

- **Mécanismes de synchronisation**
  - Permettent à des processus de coordonner leur actions
  - (Seront étudiés dans un cours ultérieur)

- **Mécanismes de communication**
  - Permettent l'échange de données entre processus
  - (Détails page suivante)

- **Signaux**
  - Evénements synchrones ou asynchrones utilisés essentiellement :
    - Par le noyau pour la gestion d'erreurs
    - Pour le contrôle des tâches dans un shell/terminal
    - Pour notifier un processus père de la fin d'un de ses fils

Mécanismes de communication

- **Transfert de données via des canaux de communication**
  - Sous forme de flux d'octets
    - Tubes (*pipes*) et tubes nommés
    - *Sockets* de type « stream »
  - Sous forme de messages
    - Files de messages
    - *Sockets* de type « datagrammes »
- **Transfert de données via des fichiers**
- **Transfert de données via mémoire partagée**
  - Des processus peuvent décider de mettre en commun une sous-partie de leurs espaces mémoires respectifs
  - Cf. appel système `mmap` (qui sert également à d'autres choses)
- **<u>Attention :</u> pour fonctionner correctement, certaines communications (notamment par fichiers et mémoire partagée) nécessitent des précautions particulières (synchronisation) – voir détails dans cours ultérieur**

Threads – Introduction (1/2)

- **Un thread est un contexte d'exécution (aussi appelé *fil* ou *flot d'exécution*)**
  - Associé à un état : registres (dont PC et SP), pile, ...
- **Par défaut (et jusqu'à présent dans ce cours), un processus n'englobe qu'un seul thread**
- **Mais il est également possible d'écrire un programme qui s'exécute avec plusieurs threads au sein du même espace de mémoire virtuelle**

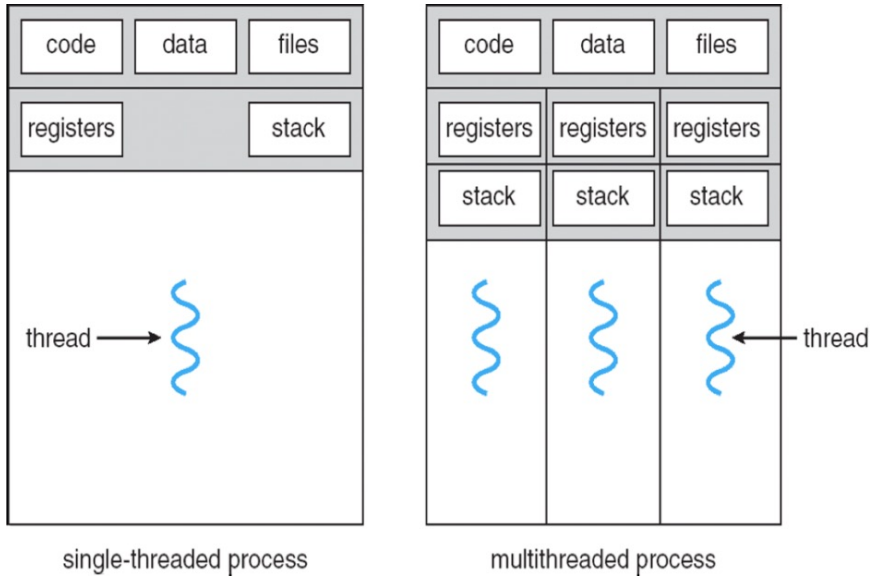


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Threads – Introduction (2/2)

- **Au sein d'un processus, les différents threads partagent tout, notamment :**
  - Code
  - Données (variables globales/statiques)
  - Données allouées dynamiquement (tas)
  - Fichiers ouverts

- **Cependant**
  - Chaque thread peut éventuellement exécuter une fonction/tâche différente
  - Chaque thread dispose d'une zone mémoire distincte pour sa pile (et donc pour ses variables locales)
  - Le système gère une sauvegarde de contexte distincte pour chaque thread au sein d'un processus

Threads – Discussion

- **Permettent la programmation d'applications concurrentes**
  - Gestion de plusieurs activités simultanées
  - Exploitation possible des multiprocesseurs (parallélisme matériel)
- **Applications multi-threads ou multi-processus ?**
  - Threads : plus légers
    - Nécessitent moins de ressources (1 seul espace mémoire)
    - Meilleur passage à l'échelle
    - Création/destruction plus rapides
  - Threads : communications plus efficaces (espace mémoire complètement mutualisé)
- **Cependant**
  - Programmer de manière correcte et efficace par mémoire partagée est non-trivial (cf. détails dans cours ultérieurs)
  - Un bogue ou une faille de sécurité au sein d'un thread peut mettre en péril tous les autres threads du même processus

Threads – Stratégies d'implémentation

- **L'abstraction de threads peut être implémentée de différentes façons**
  - Il y a deux dimensions principales à considérer
  - ... qui conditionnent le fonctionnement des threads et la façon dont on les utilise

- **Préemptibilité : un thread peut-il être suspendu à n'importe quel étape de son exécution pour laisser la main à un autre thread du même processus ?**

- **Niveau d'implémentation : les threads sont-ils visibles par le noyau ?**

Threads préemptifs ou coopératifs (1/2)

- **Threads préemptifs**
  - Un thread peut être préempté à n'importe quelle étape de son exécution pour laisser le CPU à un autre thread du même processus
  - La préemption est réalisée grâce à des interruptions matérielles (threads noyau) ou à des signaux (threads utilisateurs)
  - Plusieurs threads du même processus peuvent s'exécuter en parallèle sur différents CPU
- **Threads coopératifs**
  - Au maximum, un seul thread d'un même processus peut s'exécuter à un instant donné
  - Un thread T ne laisse le CPU à un autre thread du même processus que dans les cas suivants :
    - T libère volontairement le CPU (**`yield`**/**`sleep`** ou **`exit`**)
    - T effectue un appel système bloquant

Threads préemptifs ou coopératifs (2/2)

- **Remarque : un thread préemptif ou coopératif peut toujours être suspendu pour laisser le CPU à un autre processus**
- **Discussion**
  - Les threads préemptifs sont plus délicats à programmer car ils rendent possibles de nombreux entrelacements de threads et donc augmentent les risques de bogues.
  - Les threads coopératifs ne permettent pas à un processus d'exploiter les architectures multiprocesseurs/multicœurs.
  - Avec des threads coopératifs, un seul thread peut monopoliser le temps CPU attribué au processus correspondant.
  - Dans la passé, la majorité des implémentations de threads étaient coopératives. Aujourd'hui, elles sont principalement préemptives, en raison notamment de la généralisation des architectures multicœurs.

Threads noyau ou threads utilisateur (1/4)

- **Threads « noyau »**
  - L'ordonnanceur du noyau est informé qu'un processus peut éventuellement englober plusieurs threads
  - Les décisions et les changements de contextes effectués par l'ordonnanceur se font à l'échelle des threads

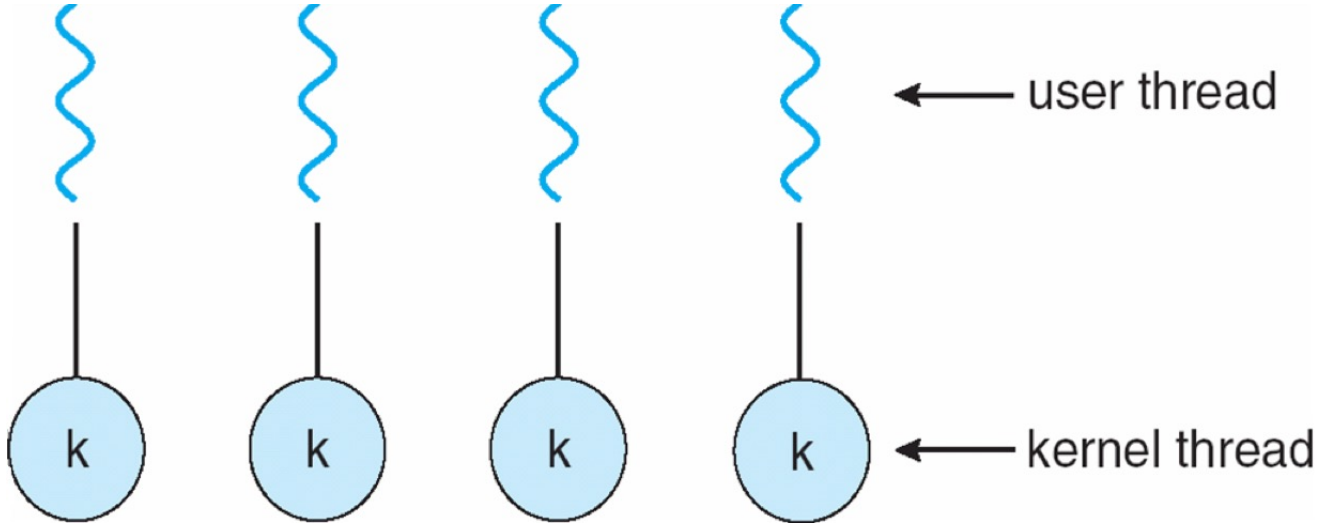


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Threads noyau ou threads utilisateur (2/4)

- **Threads « utilisateur »**
  - La gestion des threads est effectuée au niveau d'une bibliothèque exécutée en mode utilisateur
  - L'ordonnanceur du noyau ne gère qu'un seul contexte d'exécution par processus
  - La bibliothèque multiplexe ce contexte fourni par l'ordonnanceur du noyau entre plusieurs contextes de threads (au sein d'un processus)

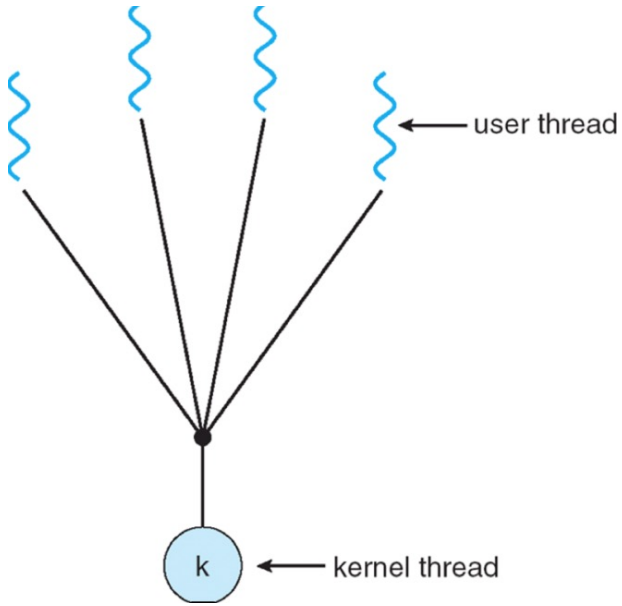


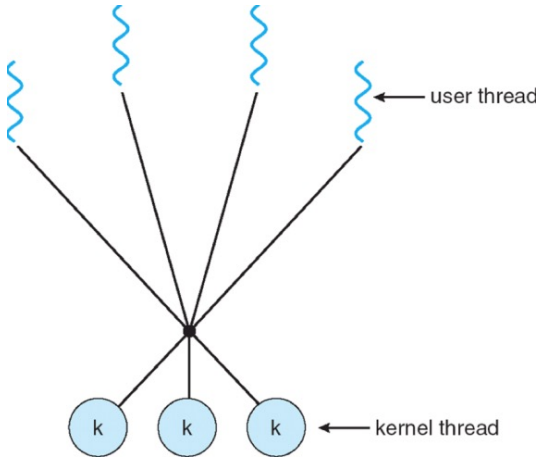
Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Threads noyau ou threads utilisateur (3/4)
Discussion

- **Les threads noyau sont plus lourds/lents**
  - Chaque opération (création/destruction/...) nécessite un appel système
  - ... et le changements de mode sur le processeur sont coûteux
  - Nécessitent plus de ressources (par exemple, un pile dans l'espace du noyau pour chaque thread, en plus de la pile utilisateur)
- **Les threads utilisateur ont aussi des inconvénients**
  - Ne peuvent pas exploiter le parallélisme matériel (pourquoi ?)
  - Certains événements (liés au noyau) qui imposent un blocage (défauts de page, certains appels système ...) bloquent non seulement le thread concerné mais tous les autres threads du processus
    - Mauvaises performances
    - Risques d'interblocages entre threads

Threads noyau ou threads utilisateur (4/4)

- **Une autre possibilité : Modèle hybride N:M**
  - N threads utilisateur multiplexés sur M threads noyau (avec N > M)
  - En comparaison, threads noyau = 1:1 et threads utilisateur = N:1



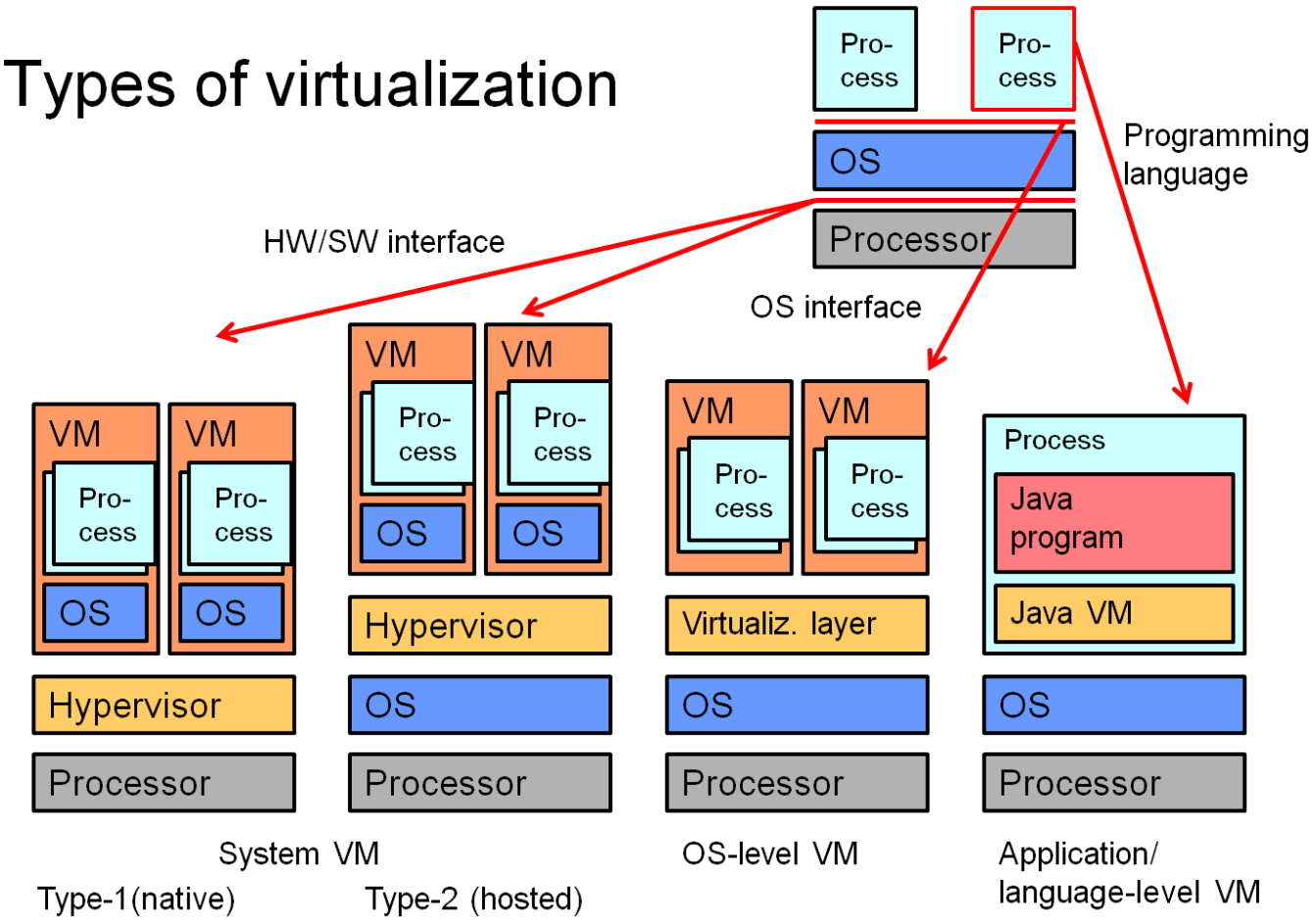
  - Potentiellement un bon compris mais difficiles à implémenter de manière efficace et sans retomber dans les mêmes problèmes que ceux des threads utilisateur

- **Remarque : en raison de la généralisation des architectures multicœurs, la majorité des implémentations de threads sont de type « noyau » (ou hybrides)**

Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Machines virtuelles

- **Des intergiciels qui étendent les fonctions classiques d'un système d'exploitation**
  - Des approches et des niveaux d'abstraction différents pour différents besoins



Picture courtesy of Gernot Heiser, UNSW.

Machines virtuelles

1er exemple : hyperviseurs (1/2)

- **Un hyperviseur**
  - est une couche logicielle
  - fournit l'abstraction d'une ou plusieurs machines « nues » (processeurs, mémoire et périphériques) au dessus d'une véritable machine physique
- **Intérêt :**
  - Faire tourner plusieurs systèmes d'exploitation simultanément sur la même plateforme matérielle
  - Faciliter le déploiement pour le « cloud computing » (IAAS: Infrastructure as a service)
  - Faciliter le test/débogage/prototypage de systèmes d'exploitation et d'applications réparties
  - Sauvegarder/rembobiner l'état d'un système
  - Sécurité (renforcement de l'isolation de certaines applications)
  - ...

Machines virtuelles
1er exemple : hyperviseurs (2/2)

- **Deux types d'implémentations**
  - Type I : hyperviseur natif
  - Type II : hyperviseur hébergé
- **Hyperviseur de type I (natif)**
  - Un hyperviseur natif est la couche logicielle de plus bas niveau
  - C'est en fait un système d'exploitation spécialisé
  - Approche la plus efficace
  - Exemples : VMware ESX, Xen
- **Hyperviseur de type II (hébergé)**
  - Un hyperviseur hébergé est une application intermédiaire (intergiciel) qui s'appuie sur un système hôte sous-jacent
  - Moins efficace (plus de couches) mais plus simple à installer/utiliser
  - Exemples : VMware workstation/fusion/player, VirtualBox

Machines virtuelles

2ème exemple : machine virtuelle Java

- **Une machine virtuelle Java (JVM) :**
  - Est implémentée sous la forme d'un intergiciel qui s'appuie sur les services d'un système hôte
    - S'exécute sous la forme d'un processus applicatif
    - Les threads d'un programme java sont (généralement) implémentés à partir des threads du système sous-jacent
  - Joue le rôle d'un interprète :
    - lit en entrée le code généré par le compilateur javac (format bytecode portable, indépendant de la plateforme sous-jacente)
    - traduit les instructions de bytecode en séquences d'actions adaptées à la plateforme sous jacente (ABI)
  - Fournit des services aux programmes Java
    - Exemples : ramasse-miettes mémoire, gestion d'interface graphique ...
    - Ces services peuvent être implémentés sous forme de threads

3ème exemple : Conteneurs (*OS-level containers*)

- **Extensions intégrées au sein d'un système d'exploitation « classique » comme Linux**
- **Création de compartiments cloisonnés**
  - Isolation de sécurité : Espace de nom (fichiers, processus) distincts
  - Isolation de performances : Règles d'allocation des ressources globales
- **En complément du support au niveau de l'OS, des outils pour simplifier la création de paquets applicatifs légers et auto-suffisants, et leur déploiement**
  - Gestion des exécutables, des bibliothèques et des fichiers de configuration
  - Gestion des numéros de version et des dépendances
  - Système de fichiers imbriqué
  - Exemple : Docker

3ème exemple : Conteneurs (*OS-level containers*) - Détails

- **Comparaison avec un hyperviseur**
  - Un seul noyau partagé entre tous les conteneurs
  - Moins de sécurité
  - Moins de flexibilité (choix de la configuration logicielle)
  - Moins de contrôle (pause/reprise, migration, rollback)
  - Mais une architecture plus simple à déployer (moins de couches logicielles) et plus légère (meilleures performances)