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

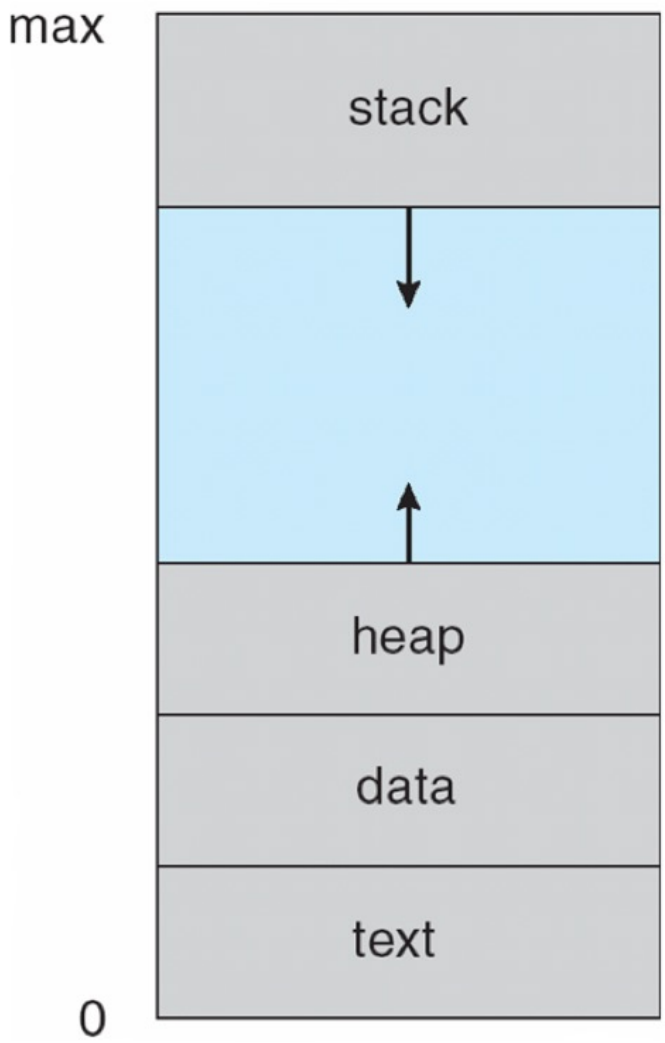


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Espace mémoire d'un processus
Vue détaillée

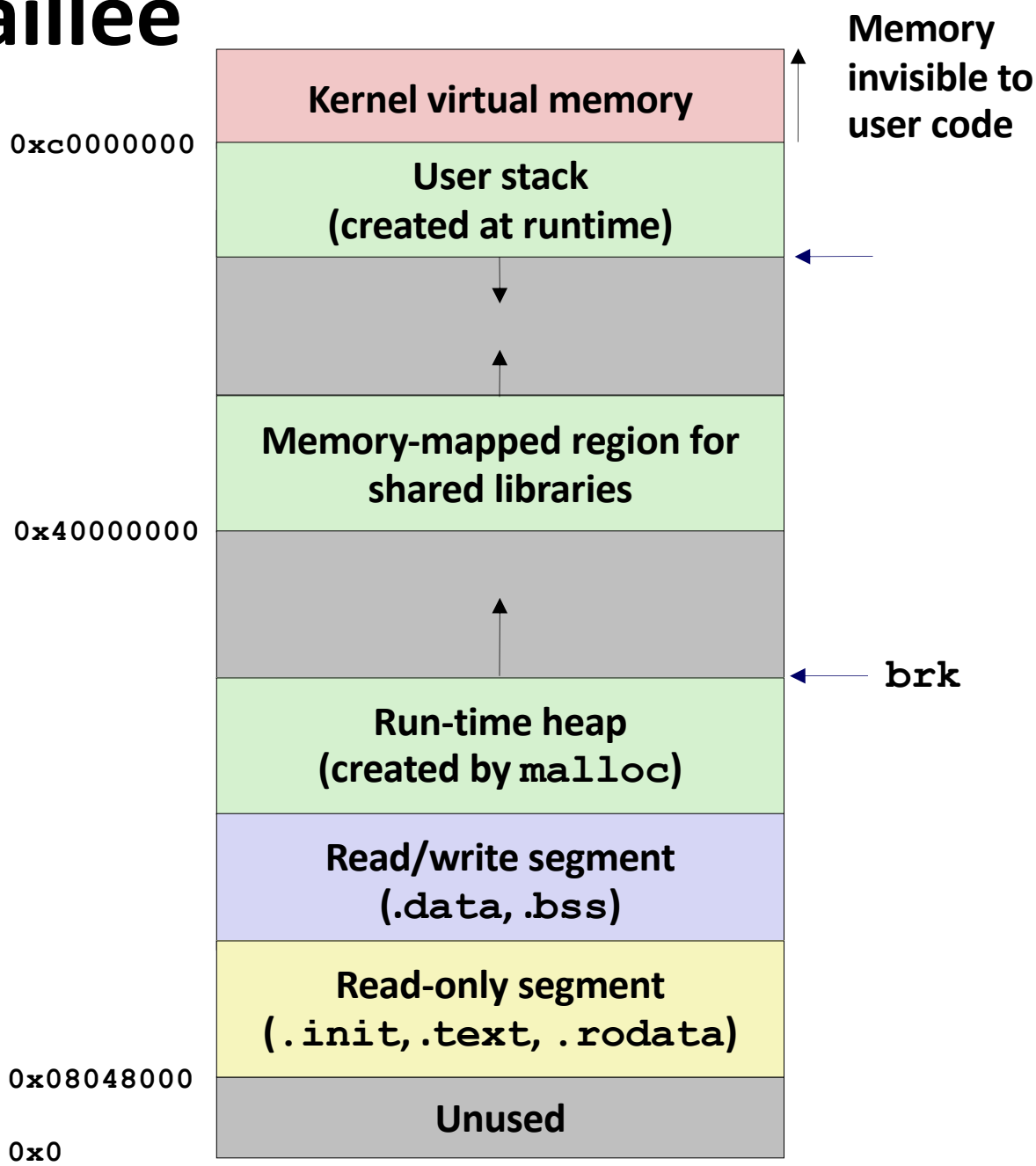


Schéma extrait de :
Bryant & O'Hallaron, Computer systems: a programmer's perspective, Prentice Hall, 2003.

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)**

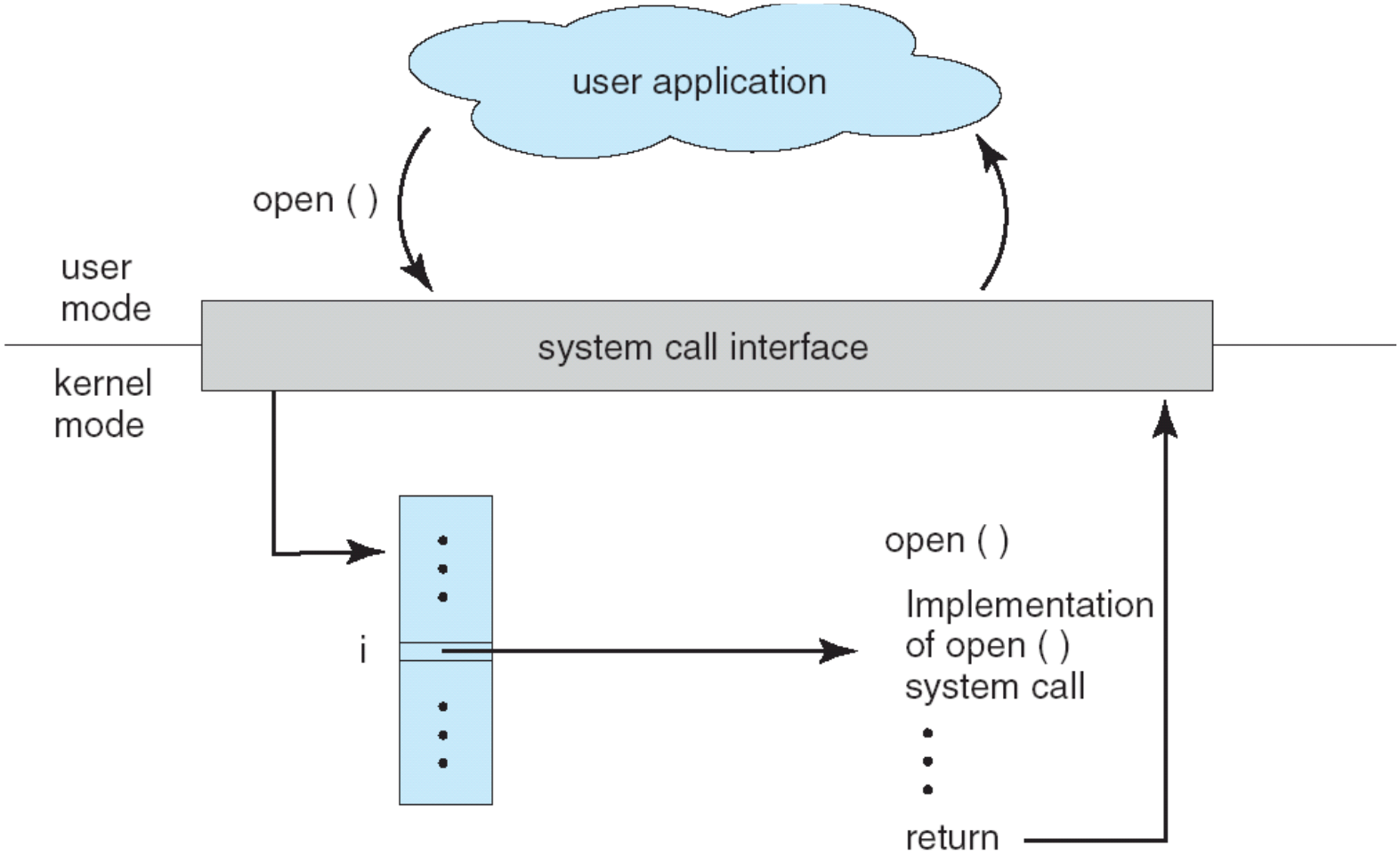


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

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)

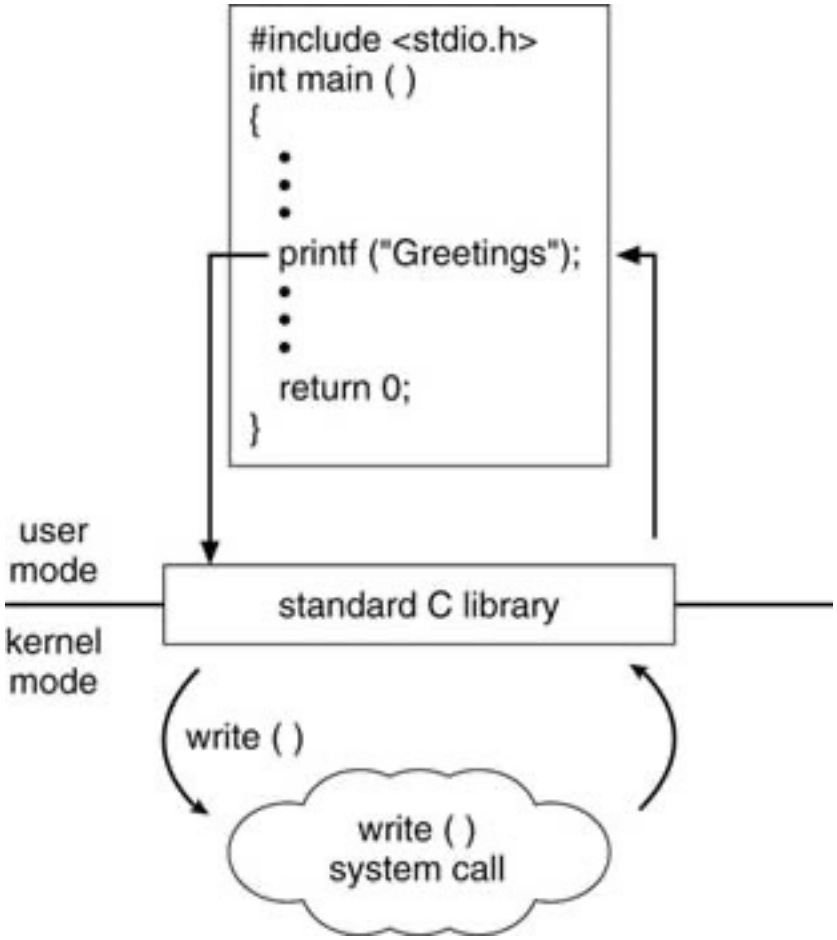


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

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)**

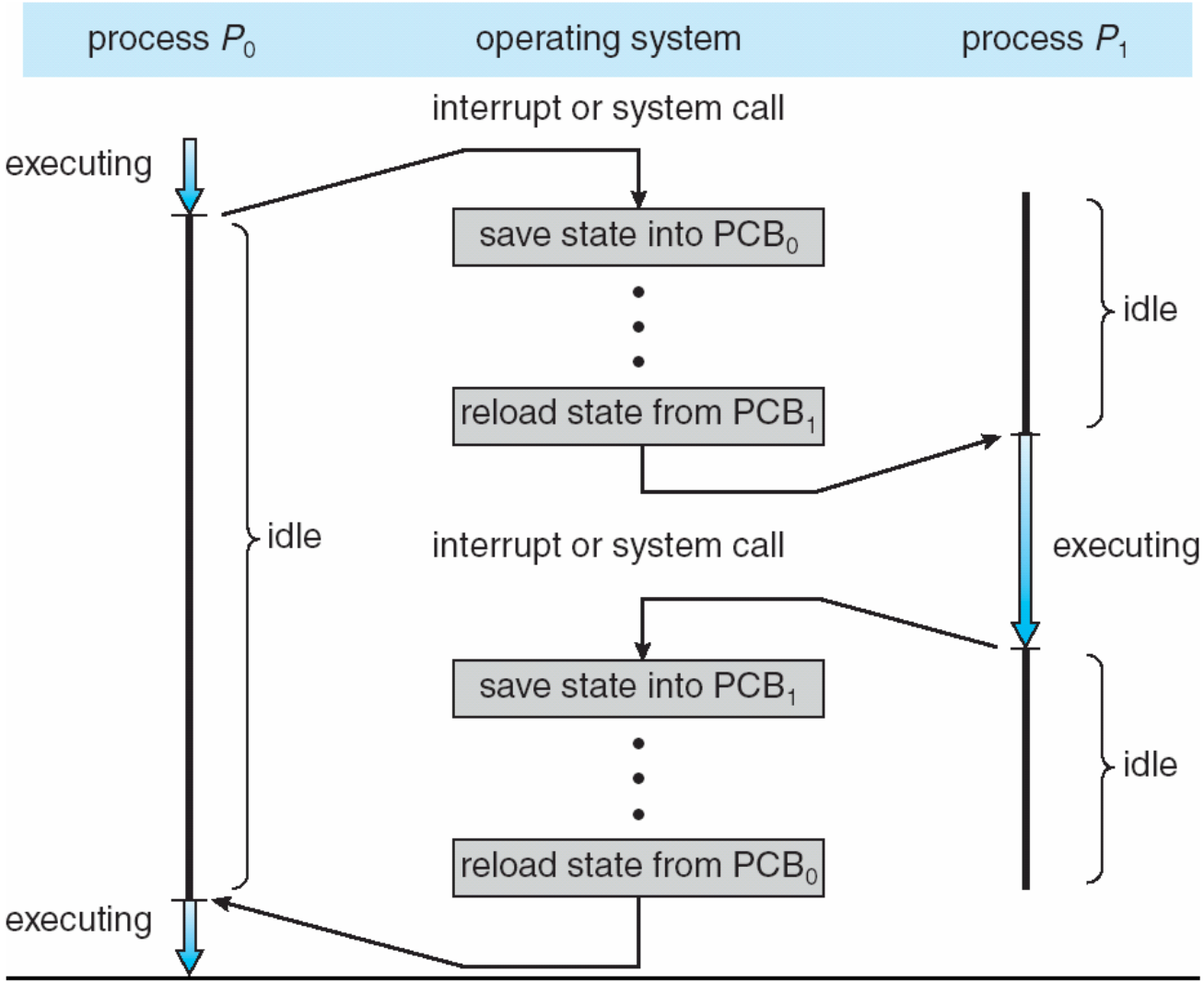
- **À quel moment l'ordonnanceur est-il invoqué ?**
  - Périodiquement, suite à chaque interruption matérielle du timer
  - Ponctuellement, en réaction à certains appels système
    - Terminaison d'un processus (**`exit`**)
    - Restitution spontanée du CPU par un processus (**`yield`**, **`sleep`**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

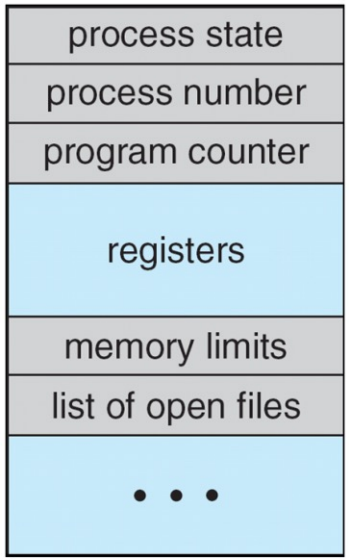
Partage du processeur (2/2)

- **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)





PCB
(process control block)

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

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)

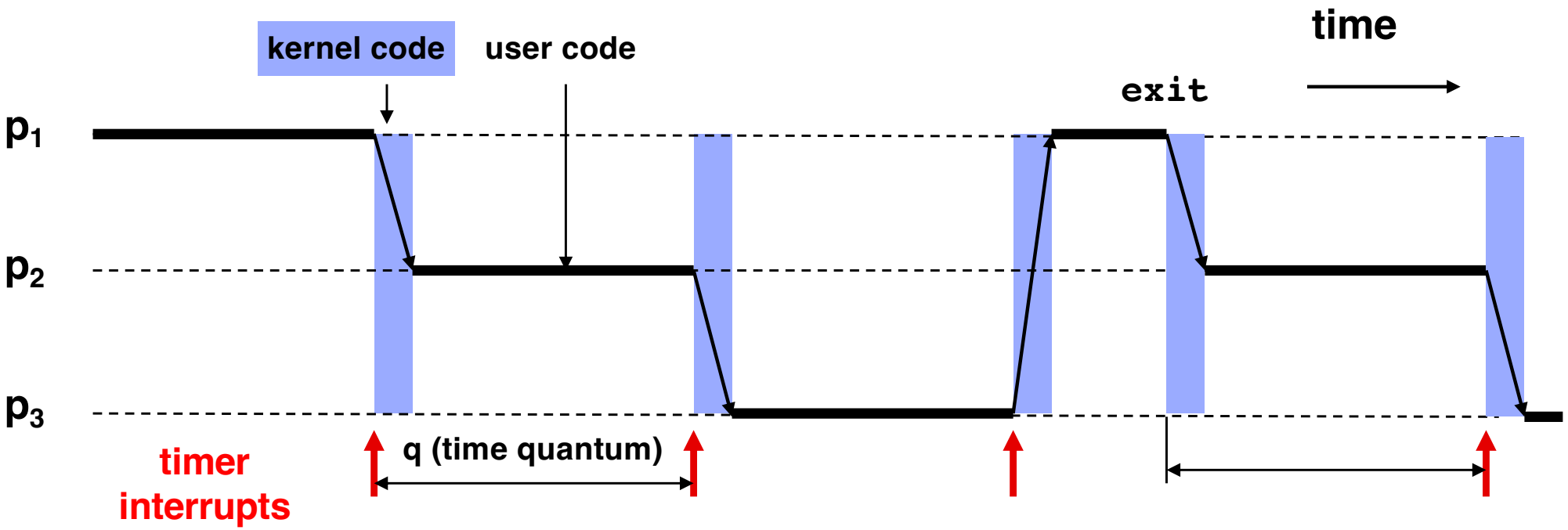
```
struct context {
  uint64 ra;
  uint64 sp;
  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

```
# 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)
```

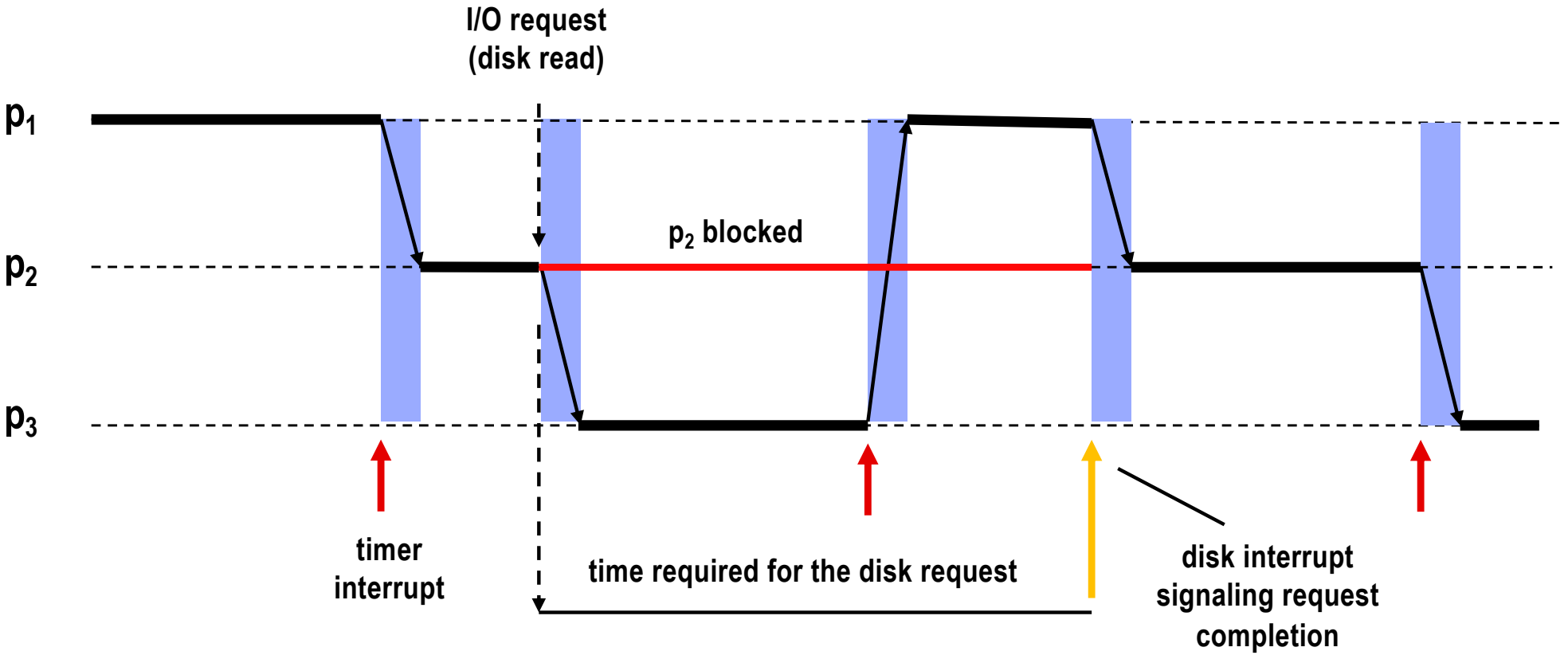
```
  # Load new registers
  ld ra, 0(a1)
  ld sp, 8(a1)
  ld s0, 16(a1)
  ld s1, 24(a1)
  ld s2, 32(a1)
  ld s3, 40(a1)
  ld s4, 48(a1)
  ld s5, 56(a1)
  ld s6, 64(a1)
  ld s7, 72(a1)
  ld s8, 80(a1)
  ld s9, 88(a1)
  ld s10, 96(a1)
  ld s11, 104(a1)

  # 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**

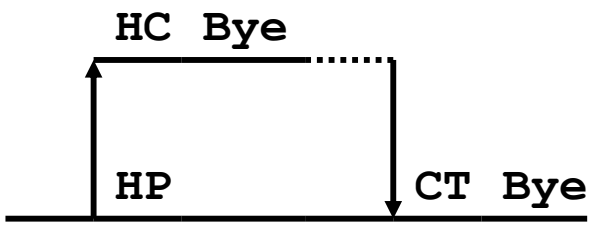
- **Chaque processus à son propre espace de mémoire virtuelle et donc son propre exemplaire des variables**

- **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**