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

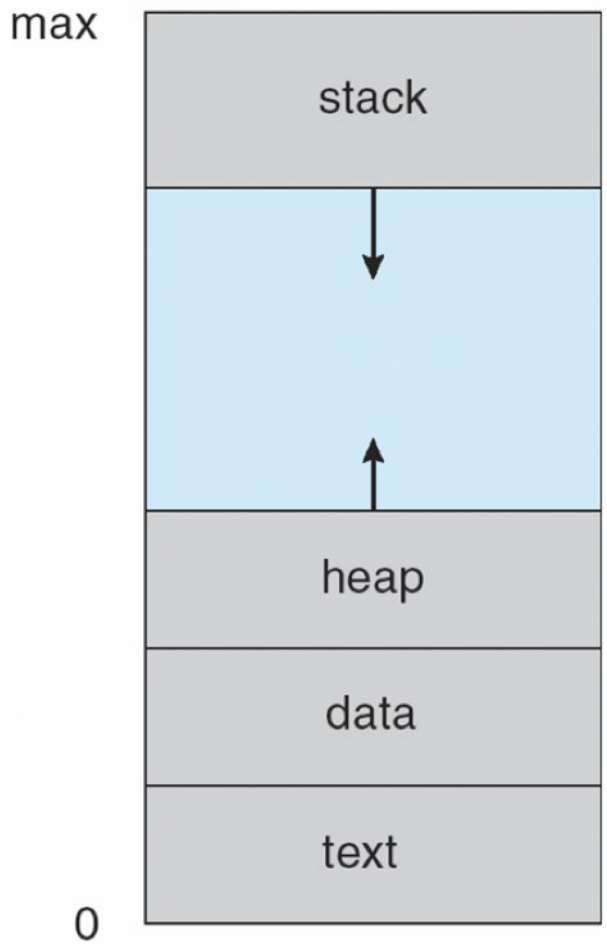


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Espace mémoire d'un processus
Vue détaillée

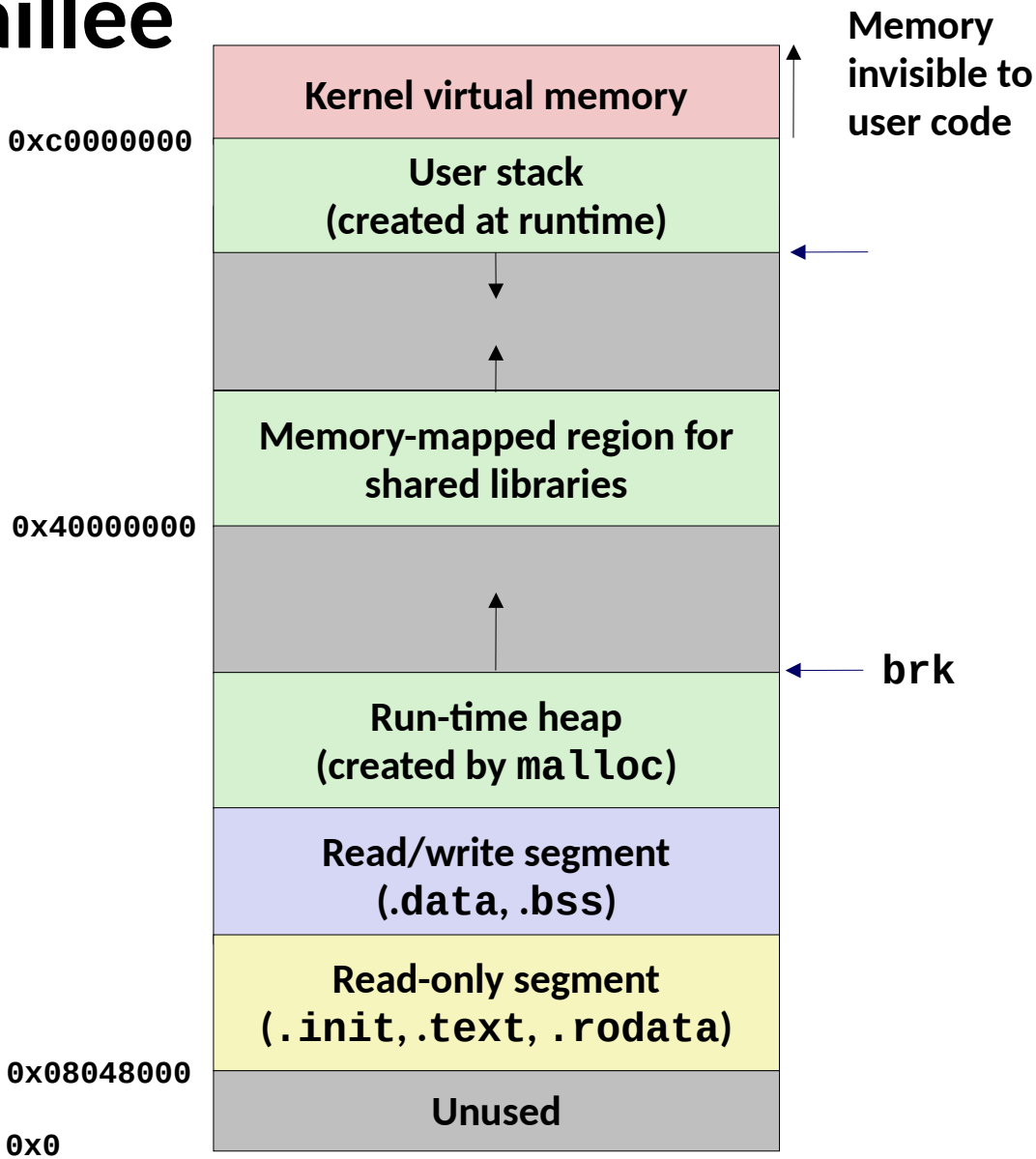


Schéma extrait de :
Bryant & O'Hallaron, Computer systems: a programmer's perspective, Prentice Hall, 2003.

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**

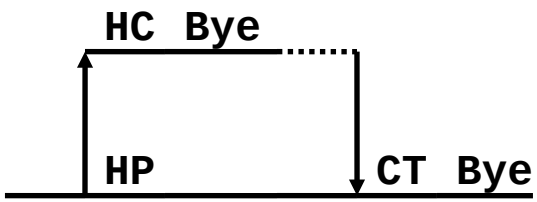
- **Chaque processus à son propre espace de mémoire virtuelle et donc son propre exemplaire des variables**

- **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**