Threads (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**

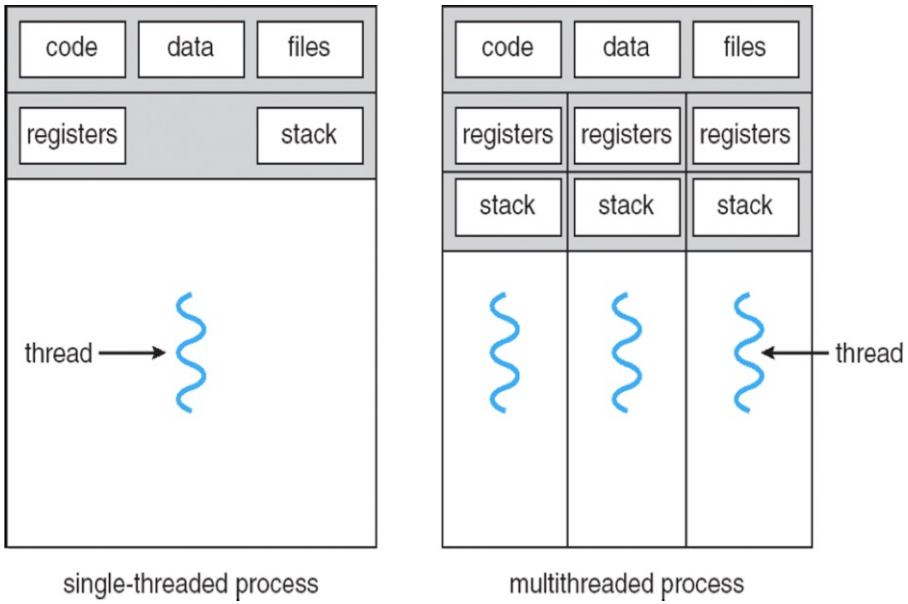


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

Threads (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

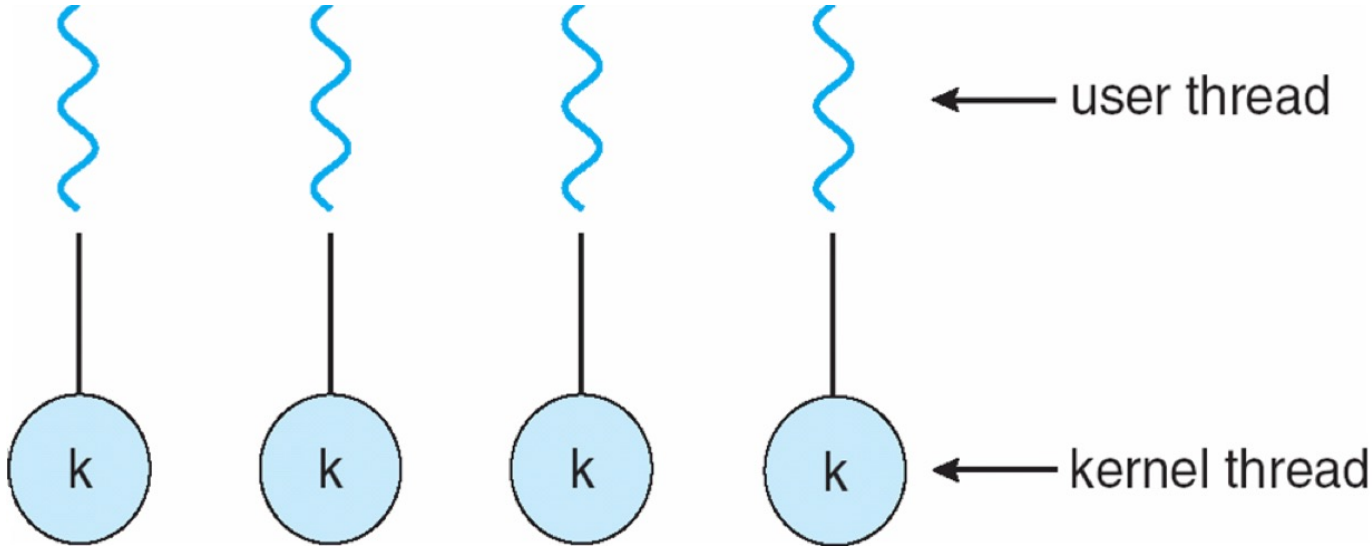


Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

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)

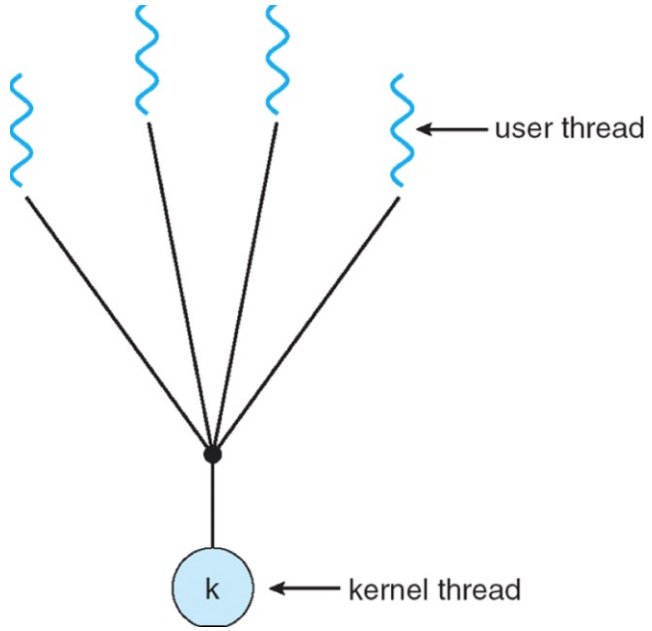


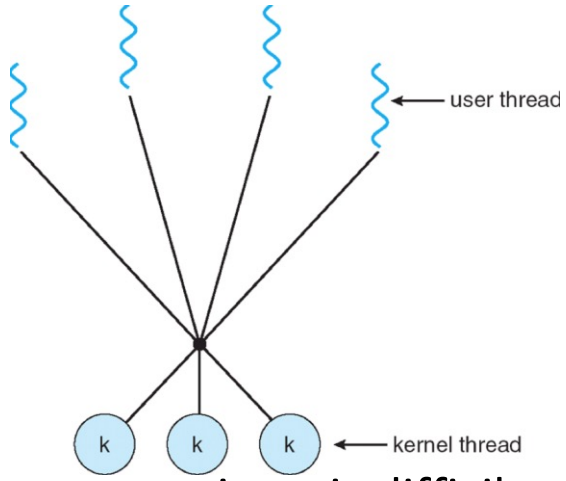
Schéma extrait de :
Silberschatz et al., Operating system concepts (8th edition), Wiley, 2008.

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.