Systèmes d'Exploitation Avancés

Pablo Oliveira [pablo.oliveira@uvsq.fr]

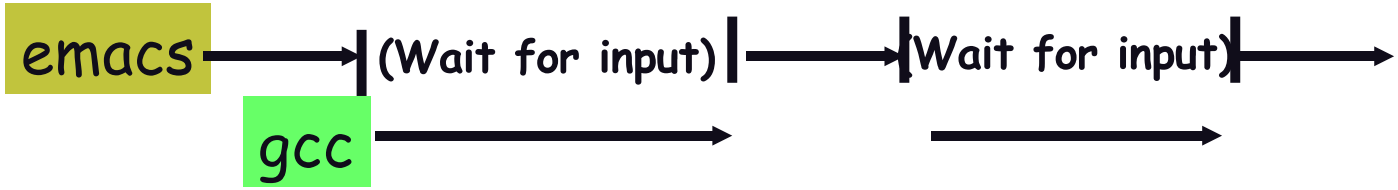
ISTY

Processus

- Un *processus* est une instance en exécution d'un programme
- SE modernes sont multi-processus (plusieurs processus en même temps)
- Exemples (ces processus peuvent tourner simultanement) :
  - `gcc file_A.c` – compilateur sur fichier A
  - `gcc file_B.c` – compilateur sur fichier B
  - `vim` – editeur
  - `firefox` – navigateur
- Contre-exemples (implemémentés comme un seul processus) :
  - Geany : plusieurs tabulations avec des fichiers ouverts
- Pourquoi les processus ?
  - Isolation
  - Simples a programmer
  - Meilleure utilisation CPU et latence

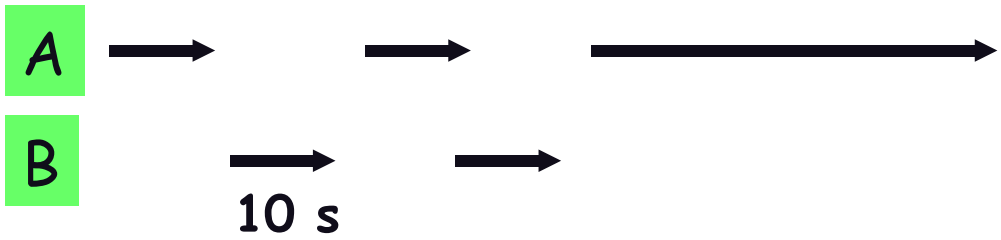
Performance

- Plusieurs processus améliorent l'utilisation du CPU
  - Recouvrement des temps d'attentes et temps de calcul



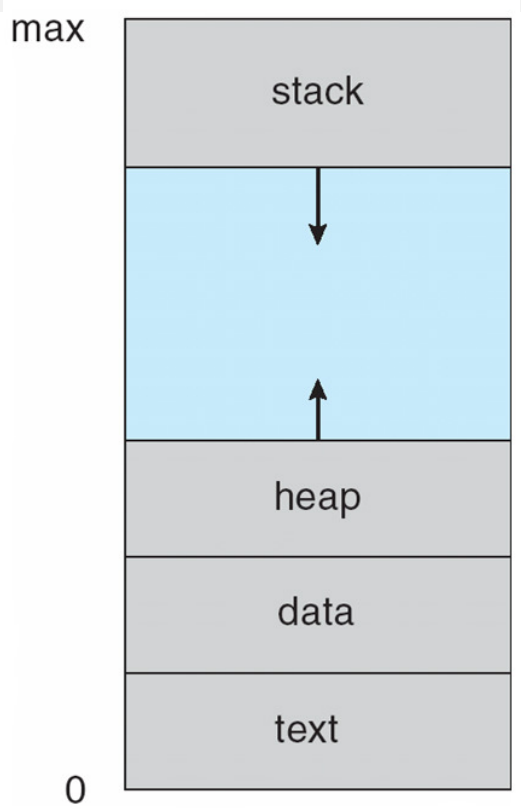
- Plusieurs processus peuvent réduire la latence
  - Exécuter A puis B nécessite 100 s pour que B termine

  - Exécuter A puis B de manière concurrente réduit la latence



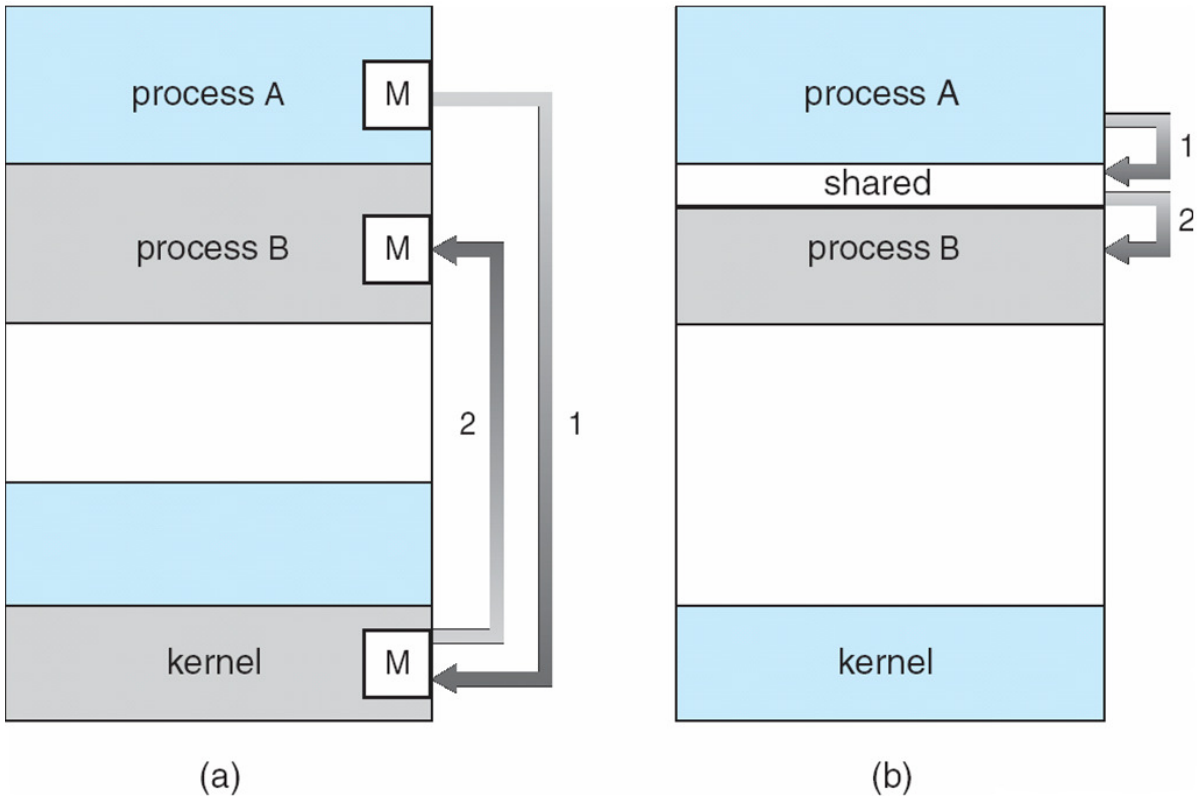
  - A est légèrement plus lent, mais moins que 100 s (excepté si A et B utilisent uniquement le CPU).

Point de vue du processus



- Chaque processus a sa propre vue de la machine
  - Son propre espace d'adressage
  - Ses propres fichiers ouverts
  - Son propre CPU virtuel (par ordonnancement préemptif)
- `*(char *)0xc000` valeur différente entre P_1 & P_2
- Simplifie énormément la programmation
  - `gcc` n'a pas besoin de se soucier que `firefox` tourne
- Parfois l'intéraction entre processus est désirable
  - Le plus simple est de communiquer à travers un fichier : `vim` édite un fichier, `gcc` le compile
  - Exemples plus compliqués : Shell/commande, Gestionnaire de Fenêtres/Application.

Communication Inter Processus (IPC)



- Comment les processus peuvent communiquer en temps réel ?
  - (a) Par passage de message à travers le noyau (eg. signaux asynchrones ou alertes)
  - (b) En partageant une zone commune de la mémoire

Plan du cours

- Vue utilisateur des processus
  - Rappels sur l'interface d'appels systèmes Unix/Linux
  - Comment créer, tuer et communiquer entre processus
- Vue noyau des processus
  - Implémentation des processus dans le noyau
- Threads
- Comment implémenter des threads

Créer un processus

- `int fork (void);`
  - Crée un processus fils qui est une copie conforme du père
  - Retourne *l'identifiant (pid)* du fils dans le père
  - Retourne 0 dans le fils
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – processus à attendre, -1 pour n'importe quel processus
  - `stat` – contient la valeur de sortie
  - `opt` – souvent 0 ou `WNOHANG`
  - Retourne pid du processus attendu ou -1 en cas d'erreur
- `int wait(int *stat) <=> int waitpid(-1, int *stat, 0);`

Terminer un processus

- `void exit (int status);`
  - Termine le processus courant
  - `status` est retourné dans le `waitpid` du père
  - Convention : `status` est 0 pour une sortie normale
- `int kill (int pid, int sig);`
  - Envoie le signal `sig` au processus `pid`
  - `SIGTERM` tue le processus (le signal peut être intercepté de manière à sortir proprement)
  - `SIGKILL` tue le processus (le signal ne peut pas être intercepté)

Exécution de programmes

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – path du programme à exécuter
  - `argv` – tableau des arguments
  - `envp` – variables d'environnement, e.g., `PATH`, `HOME`
- Possède plusieurs wrappers plus simples
  - `int execvp (char *prog, char **argv);`
    Cherche dans `PATH` le programme, réutilise l'environnement courant
  - `int execlp (char *prog, char *arg, ...);`
    Permet de passer les arguments un par un, doit finir par `NULL`
- Exemple : `minish.c`
  - Boucle qui lit des commandes et les exécute

minish.c (simplifié)

```
pid_t pid; char **args;
void doexec () {
  execvp (args[0], args);
  perror (args[0]);
  exit (1);
}
    /* ... main loop: */
    for (;;) {
      parse_next_line_of_input (&args, stdin);
      switch (pid = fork ()) {
      case -1:
        perror ("fork"); break;
      case 0:
        doexec ();
      default:
        waitpid (pid, NULL, 0); break;
      }
    }
```

Manipulation de descripteurs de fichiers

- `int dup2 (int oldfd, int newfd);`
  - Ferme `newfd`, si descripteur valide
  - Écrase `newfd` avec une copie conforme de `oldfd`
  - Les deux descripteurs partagent l'offset de lecture (un appel à `lseek` affecte les deux `fd`)
- `int fcntl (int fd, F_SETFD, int val)`
  - Active (`val=1`) ou Désactive le mode *close on exec* sur fd.
  - Le `fd` ne peut pas être hérité par les programmes lancés avec execv.
- Exemple : `redirsh.c`
  - Loop qui lit une commande et l'éxécute
  - Reconnaît `command < input > output 2> errlog`

redirsh.c

```
void doexec (void) {
  int fd;
  if (infile) {        /* non vide pour "command < infile" */
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  /* ... pareil pour outfile -> fd 1, errfile -> fd 2 ... */

  execvp (args[0], args);
  perror (args[0]);
  exit (1);
```

Pipes / Tubes

- `int pipe (int fds[2]);`
  - Retourne deux descripteurs dans `fds[0]` et `fds[1]`
  - Une écriture sur `fds[1]` sera lisible sur `fds[0]`
  - Retourne 0 en cas de succès
- Opération sur les Tubes
  - `read`/`write`/`close` – comme pour un fichier
  - Si `fds[1]` fermé, `read(fds[0])` retourne 0
  - Si `fds[0]` fermé, `write(fds[1])` :
    - Envoie le signal `SIGPIPE` au processus
    - Si le signal est ignoré lève la faute EPIPE
- Exemple : `pipesh.c`
  - `command1 | command2 | command3 ...`

pipesh.c (simplifié)

```
void doexec (void) {
  while (outcmd) {
    int pipefds[2]; pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break; /* lancement command1 */
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&args, &outcmd);
      break; /* lancement command2, etc... */
    }
  }
```

Pourquoi fork ?

- Souvent `fork` suivi de `execve`
- On pourrait les combiner dans un *spawn* system call
- Parfois utile de forker un processus sans `execve`
  - Unix *dump* fait des sauvegardes sur bande
  - Lorsque la bande est pleine, il faut revenir à un point de sauvegarde
  - Fork pour revenir à l'état sauvegardé après changement de la bande.
- Fork propose une interface très simple
  - Plein de choses possibles à faire sur le fils :
    manipuler les fd, l'environnement, les limites de ressource, etc.
  - Pourtant `fork` ne nécessite aucun argument

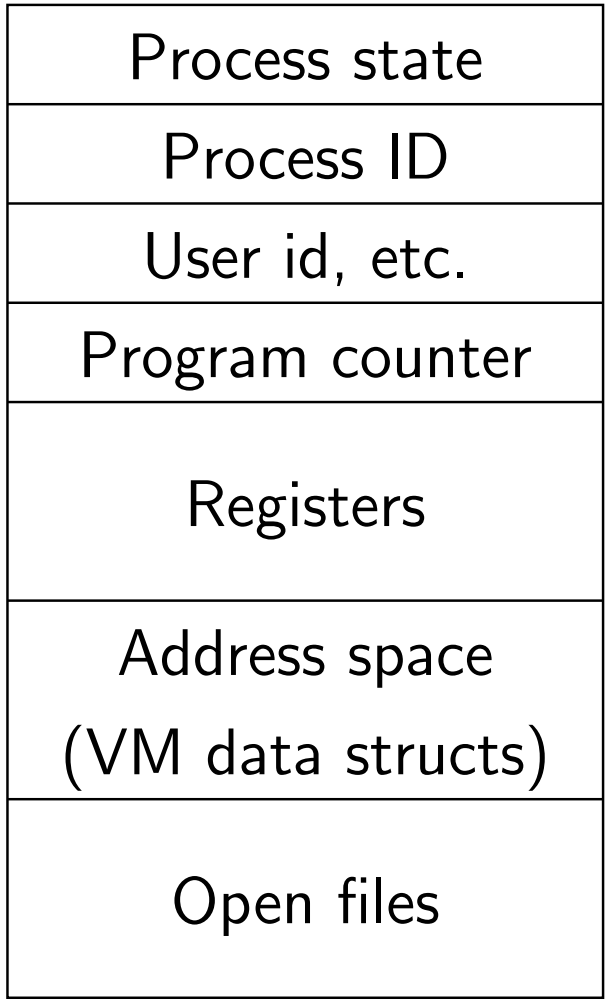
Lancer des processus sans fork

- Sans fork, pléthore d'arguments
- Exemple : Windows appel système `CreateProcess`
  - et `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`, ...

```
BOOL WINAPI CreateProcess(
  _In_opt_        LPCTSTR lpApplicationName,
  _Inout_opt_     LPTSTR lpCommandLine,
  _In_opt_        LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_        LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_            BOOL bInheritHandles,
  _In_            DWORD dwCreationFlags,
  _In_opt_        LPVOID lpEnvironment,
  _In_opt_        LPCTSTR lpCurrentDirectory,
  _In_            LPSTARTUPINFO lpStartupInfo,
  _Out_           LPPROCESS_INFORMATION lpProcessInformation
);
```

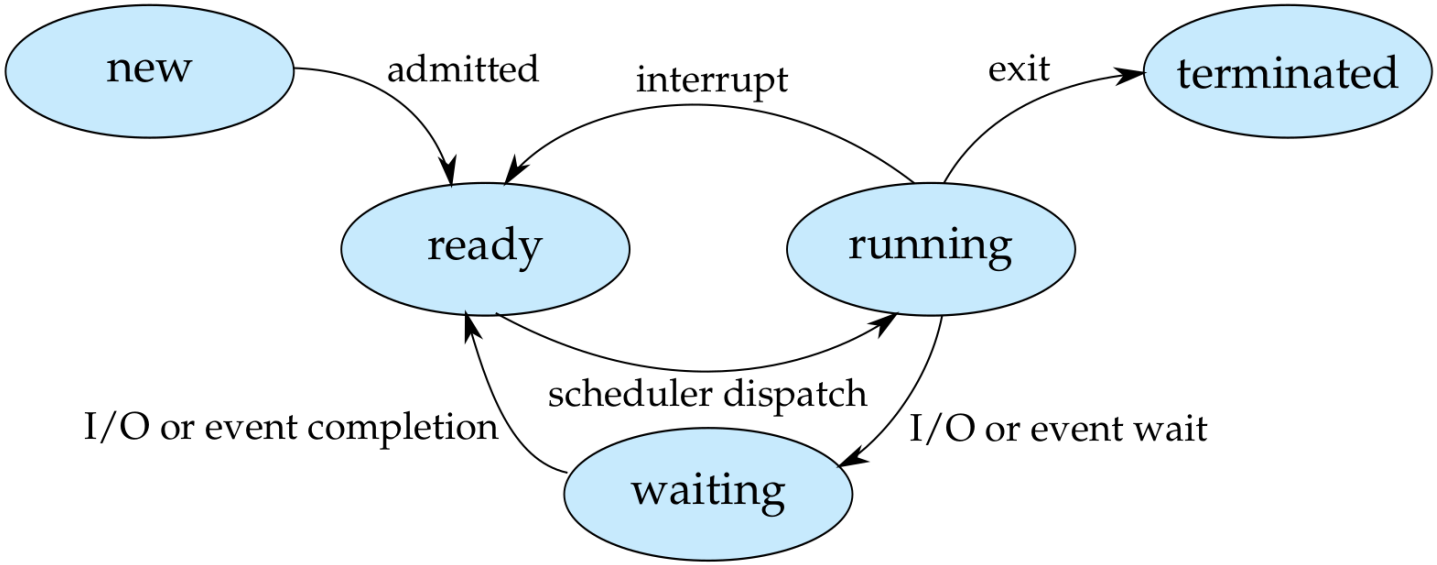
Implementation des processus

- Le SE connait la liste des processus
  - Process Control Block (PCB)
  - Appellé `proc` dans Unix et `task_struct` dans Linux.
- Suit *l'état* du processus
  - En Exécution, Prêt, Bloqué, etc.
- Inclus l'information sur le contexte du processus
  - Banc de registres, traductions d'adresses virtuelles, etc.
  - Fichier ouverts
- D'autres informations sur le processus
  - Capabilités (user/group ID), masque de signal, terminal, priorité, ...



PCB

État d'un processus



- Les processus sont dans un des états suivants
  - *nouveau & terminé* – en début et en fin de vie
  - *en exécution* – en cours d'exécution
  - *prêt* – en attente d'être exécuté
  - *en attente* – bloqué en attente d'un évenemment (e.g., lecture disque)
- Quel processus le SE doit-il choisir ?
  - Si 0 processus prêt, tourne à vide ou éteint le processeur.
  - Si >1 processus prêts, décision d'ordonnancement.

Ordonnancement

- Quel processus exécuter ?
- Choisir le premier prêt dans la table des processus ?
  - Coûteux. Biais de priorité (petits pids)
  - Deux tables séparées : prêts / en attente
- FIFO ?
  - File d'attente

- Priorité ?
  - Donner la priorité à certains threads

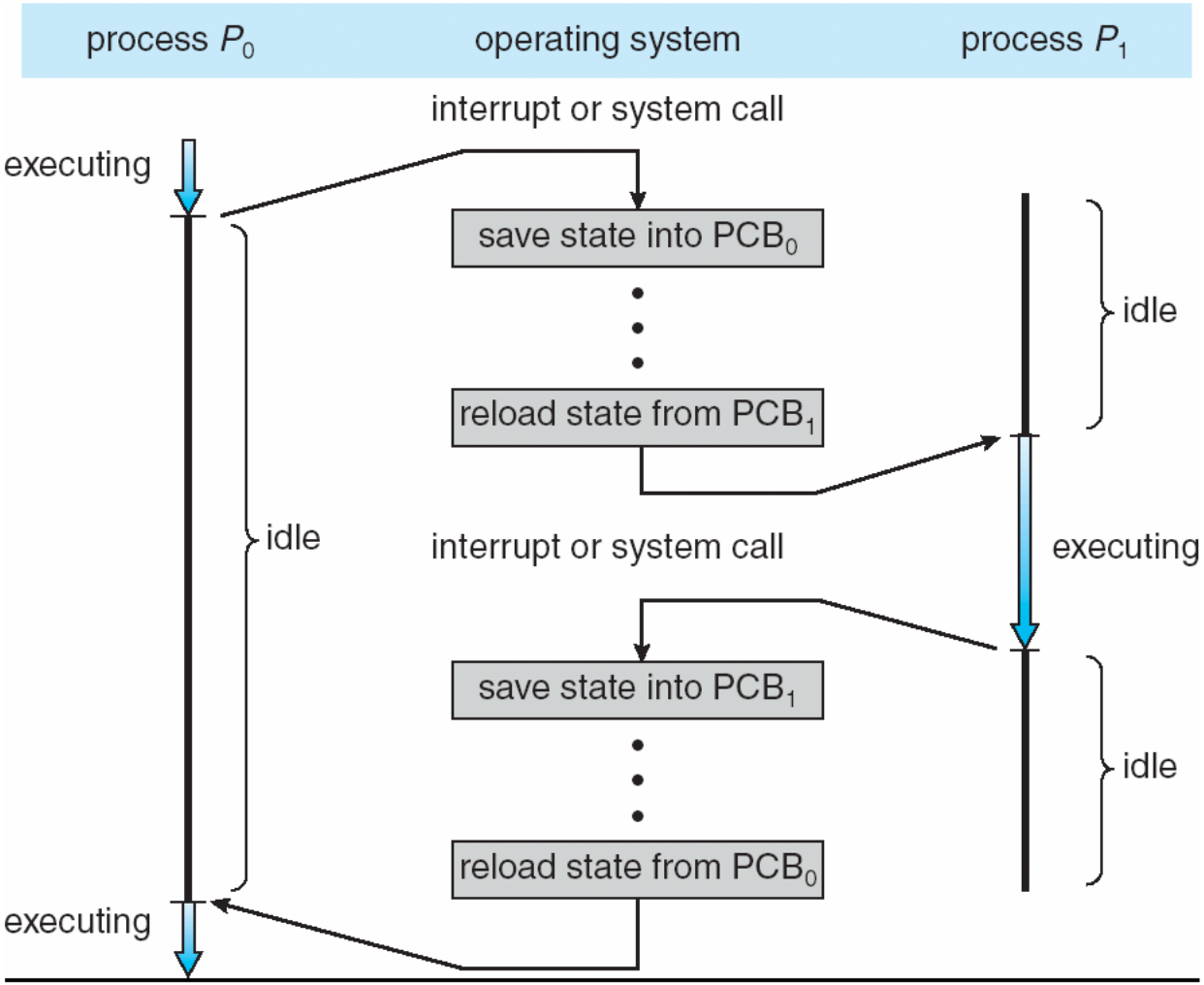
Buts de l'Ordonnancement

- Compromis entre plusieurs objectifs
  - *Équité* – pas de famine
  - *Priorité* – certains processus sont plus importants (QOS)
  - *Échéances* – important de faire x (eg. audio) avant une date
  - *Débit* – bonne capacité de calcul
  - *Efficience* – réduire le surcoût de l'ordonnanceur
- Pas de martingale
  - Multiobjectifs, on ne peut pas satisfaire tous les critères
  - Objectifs contradictoires (e.g., priorité vs. équité)
- Voir cours sur l'ordonnancement

Préemption

- Le noyau peut préempter lorsqu'il a le contrôle
- Le processus en exécution peut être intérrompu
  - Appel système, Faute de page, Instruction illégale
  - Peut bloquer le processus—e.g., lecture sur disque
  - Peut débloquer des processus—e.g., fork, écriture sur pipe
- Timer périodique (interruption horloge)
  - Interrompre le processus en exécution si *quantum* consommé
- Interruption matérielle
  - Requête disque complète, arrivée d'un paquet réseau
  - Débloque un processus
  - qui sera schédulé si ça priorité est plus haute que le processus courant
- C'est un changement de contexte

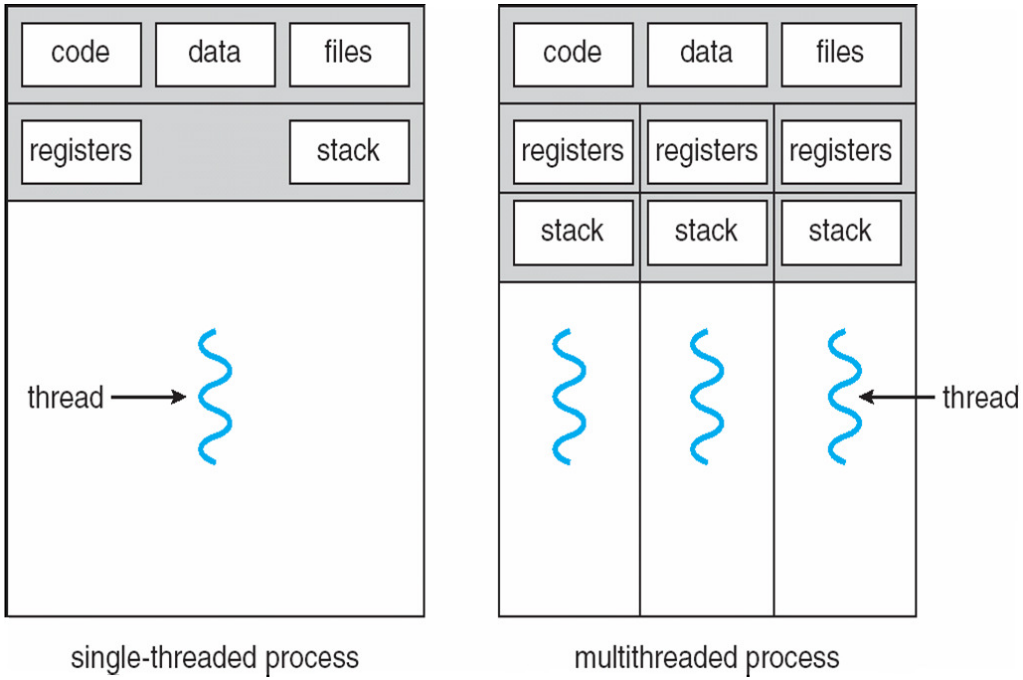
Changement de contexte



Détails du changement de contexte

- Dépend de l'architecture. Scénario typique :
  - Sauver le compteur ordinal et les registres entiers
  - Sauver les registres flottants et les registres spéciaux
  - Sauver les flags conditionnelles
  - Sauver les traductions d'adresse virtuelles
- Coût non négligeable
  - Sauver les registres flottants est coûteux
    - Optimisation : sauvés uniquement si utilisés
  - Nécessite parfois un flush TLB (cache traduction adresses)
    - Optimisation : ne pas flusher le TLB avec les traductions du noyau
  - Cause des miss dans les caches (processus travaillent sur des données différentes)

Threads



- Un thread encapsule un contexte d'execution
  - compteur ordinal, pile, registres, ...
- Programmes simples 1 Thread par Processus
- Programmes multi-threads N Threads par Processus
  - Les threads se partagent le même espace mémoire

Pourquoi les threads ?

- Abstraction efficace pour la concurrence
  - Plus légers que les processus
  - Permettent le partage de mémoire (calcul parallèle)
- Permettent d'exploiter plusieurs coeurs de calcul
- Processus : recouvrement des E/S et du calcul
  - Comme pour un SE qui tourne `emacs & gcc` simultanement
  - E.g., un serveur WEB threadé possède un thread par client

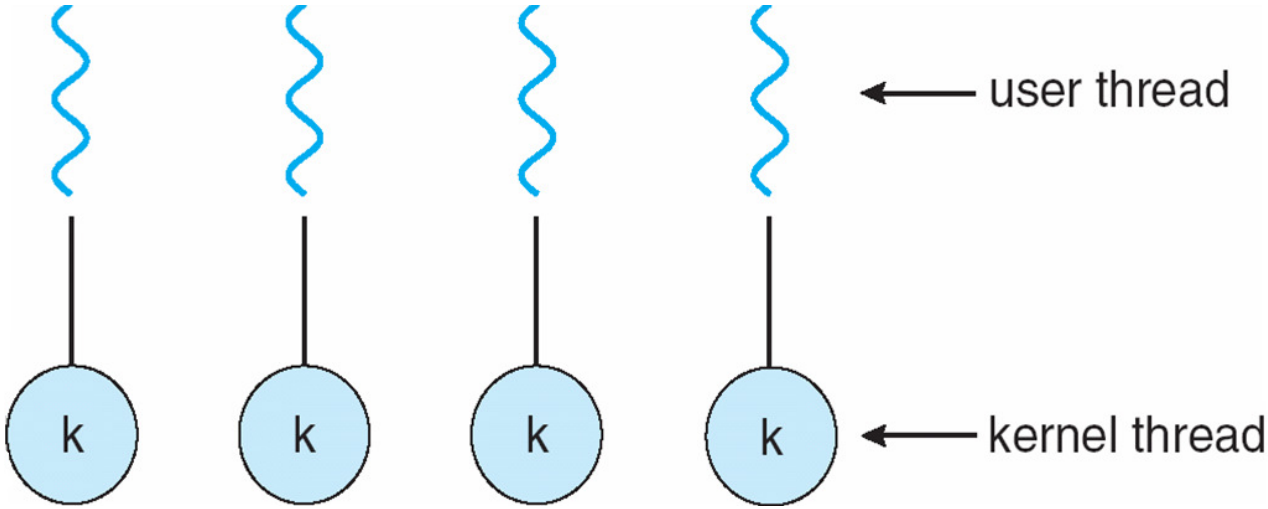
```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

- La plus part des noyaux sont écrits avec des threads

Thread POSIX API

- `tid thread_create (void (*fn) (void *), void *arg);`
  - Crée un nouveau thread qui exécute `fn(arg)`
- `void thread_exit ();`
  - Termine le thread courant
- `void thread_join (tid thread);`
  - Attends la terminaison d'un autre thread
- Primitives de synchronization (vues dans le cours 4)
- Consulter [Birell] pour une bonne introduction
- Threads Préemptifs ou Coopératifs

Threads Noyau

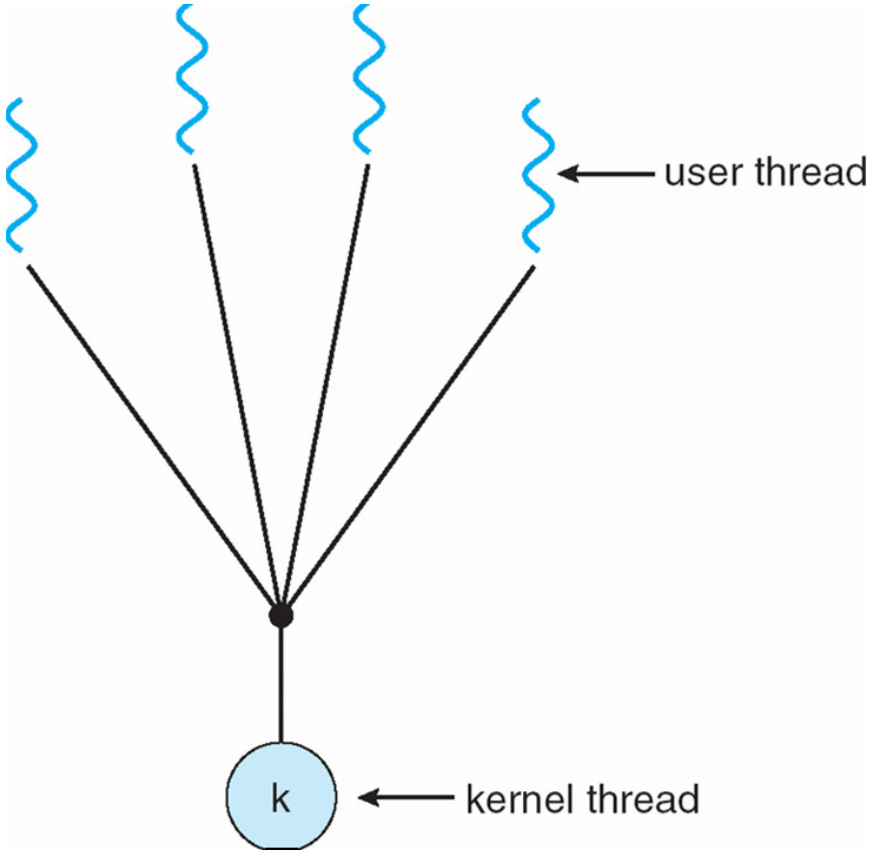


- On pourrait implémenter `thread_create` comme un appel système
- Pour ajouter `thread_create` à un SE où la fonction est absente :
  - Prendre pour point de départ un processus Noyau
    - Partager le même espace d'adresse, table des fichiers, etc. dans le nouveau processus
    - Appel système `clone` sous linux
- Plus léger qu'un processus mais assez coûteux tout de même

Limitations des threads noyau

- Chaque opération sur le thread passe par le noyau
  - creation, sortie, join, synchro, changement de contexte
  - Sur un laptop : appel système 100 cycles, appel fonction 5 cycles
  - Conclusion : threads 10x-30x plus lents si implémentés dans le noyau
- Hérite des structures de données lourdes des processus
  - E.g., taille fixe de pile

Threads utilisateurs



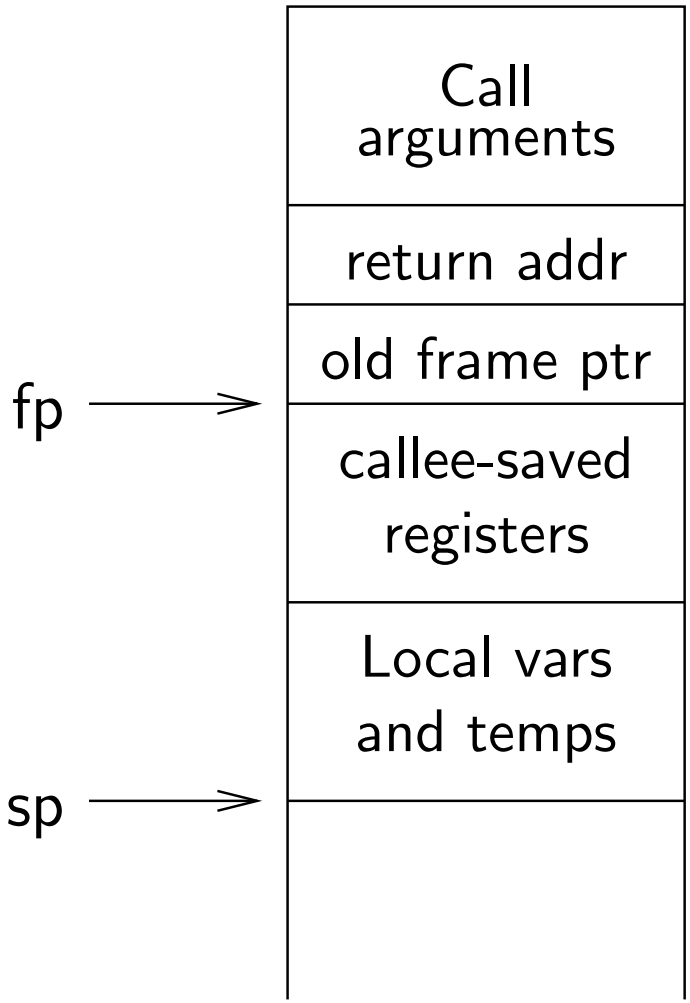
- Les threads peuvent être implémentés en espace utilisateur
  - Un seul thread noyau par processus
  - `thread_create`, `thread_exit`, etc., simples appels de fonctions

Implémentation des threads utilisateur

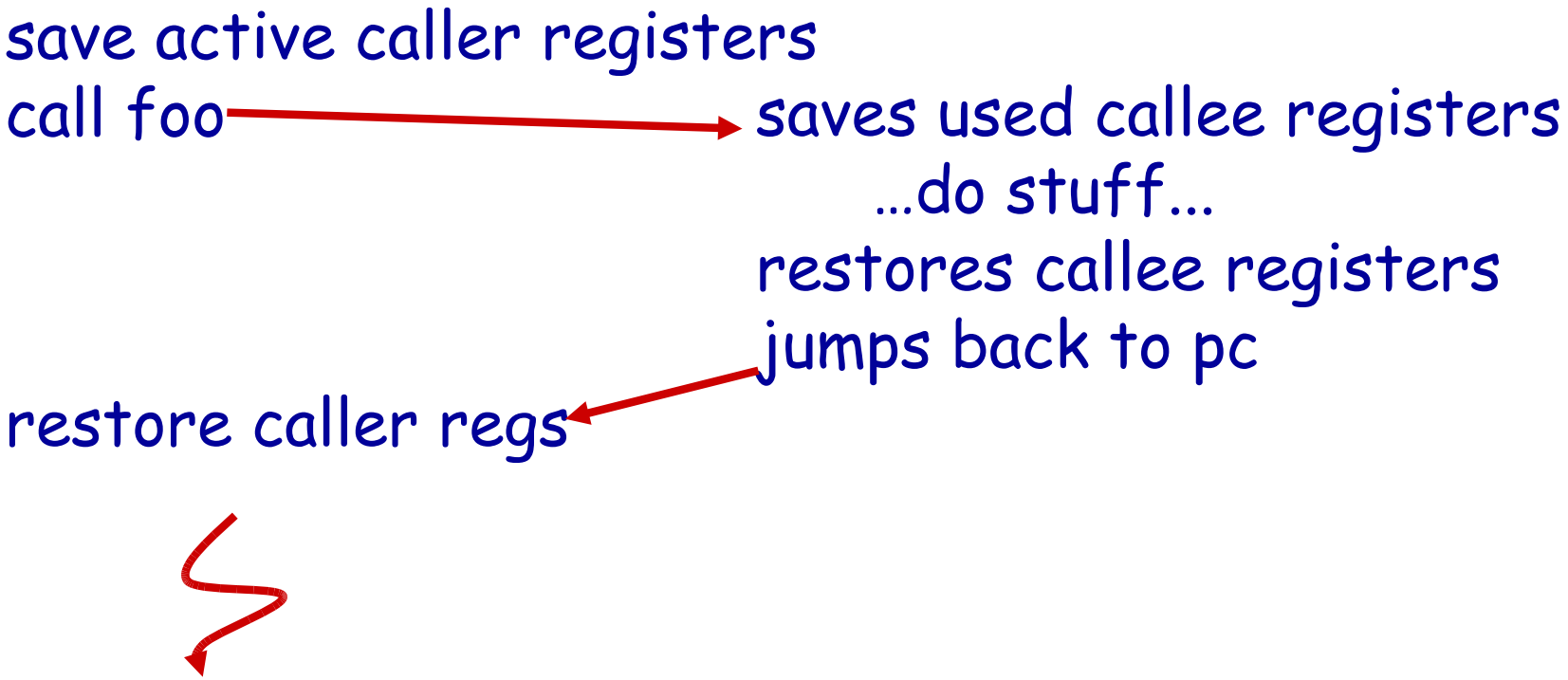
- Allouer une nouvelle pile pour chaque `thread_create`
- Gérer une pile de threads en attente
- Intercepter les appels d'E/S (`read`/`write`/etc.)
  - Si une opération bloque, changer de thread
- Demande un signal d'horloge (`setitimer`)
  - Change de thread à chaque réception de signal (preémption)
- Example d'un serveur WEB multithréadé
  - Le thread appelle `read` pour recevoir une requête
  - L'appel est intercepté
  - Si pas de données encore reçues : on passe au thread suivant
  - À chaque signal d'horloge on vérifie quels threads on reçu des données
- Comment changer le contexte de deux threads ?

Présentation des Convention d'appel

- On sépare les registres en deux groupes
  - Les fonctions peuvent écraser les registres *caller-saved* (%eax [return val], %edx, & %ecx)
  - Mais doivent préserver les registres *callee-saved* (%ebx, %esi, %edi, %ebp et %esp)
- *sp* pointe vers le début de la pile
- Variables locales dans des registres et sur la pile
- Arguments dans des registres caller-saved et sur la pile
  - Sur x86, tous les arguments sur la pile



Appels de fonctions



- État de l'appelant sauvegardé sur la pile
  - Adresse de retour, registres caller-saved
- Une partie de l'état est ailleurs
  - Registres callee-saved, variables globales, position de la pile

Threads vs. fonctions

- Les threads peuvent être débloqués en un ordre quelconque
  - Une pile (LIFO) ne permet donc pas de sauver l'état
  - Solution générale : une pile par thread
- Changement de thread moins fréquent que changement de fonction
  - On ne partitionne pas les registres (Pourquoi ?)
- L'interruption d'une thread peut-être volontaire (`sleep`)
  - Synchrone : l'appel de fonction sauve une partie de l'état
  - Asynchrone : le code de changement de contexte sauve tous les registres

Example d'implémentation de threads utilisateurs

- Thread control block (TCB)

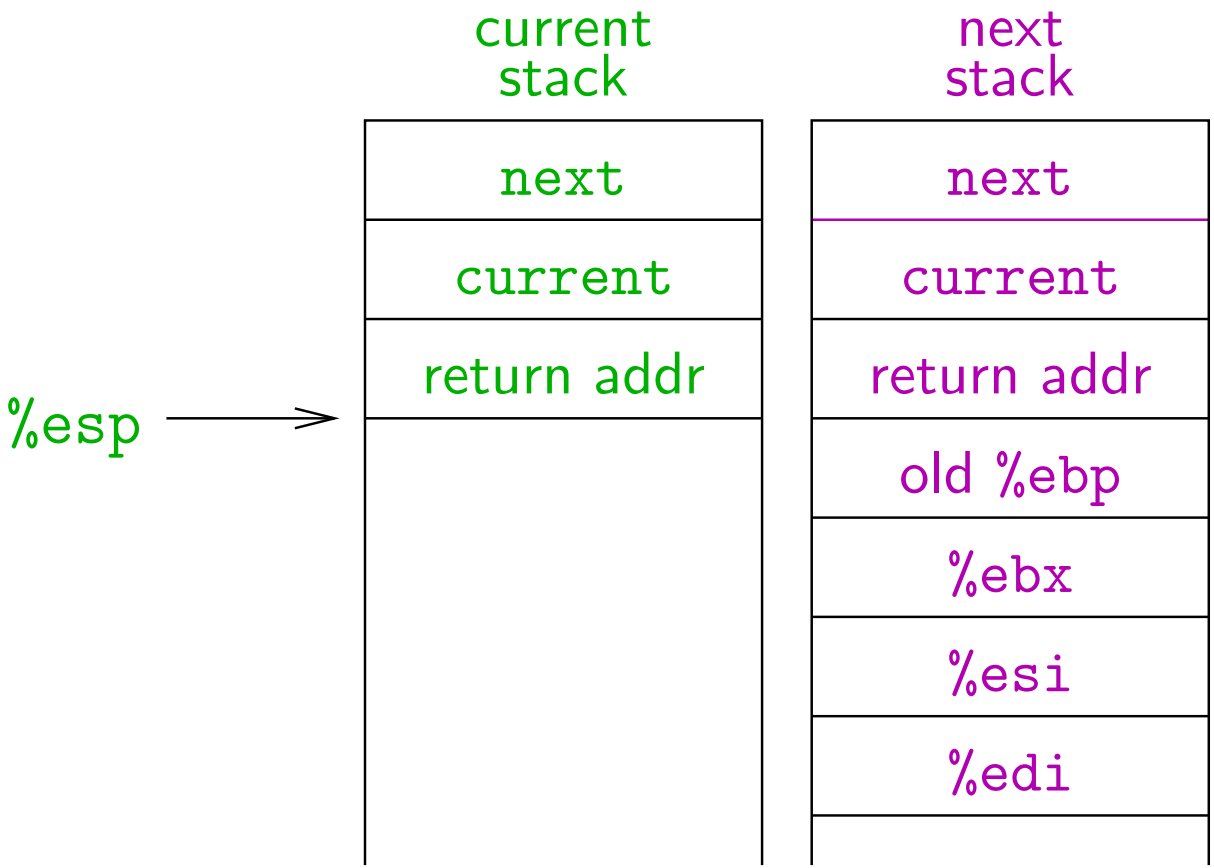
```
typedef struct tcb {
  unsigned long md_esp;  /* Pointeur de pile*/
  char *t_stack;         /* Pile */
  /* ... */
};
```

- Changement de conxtexte (dépendant de la machine)
  - `void thread_md_switch (tcb *current, tcb *next);`
- Initialization d'un thread (dépendant de la machine)
  - `void thread_md_init (tcb *t, void (*fn) (void *), void *arg);`

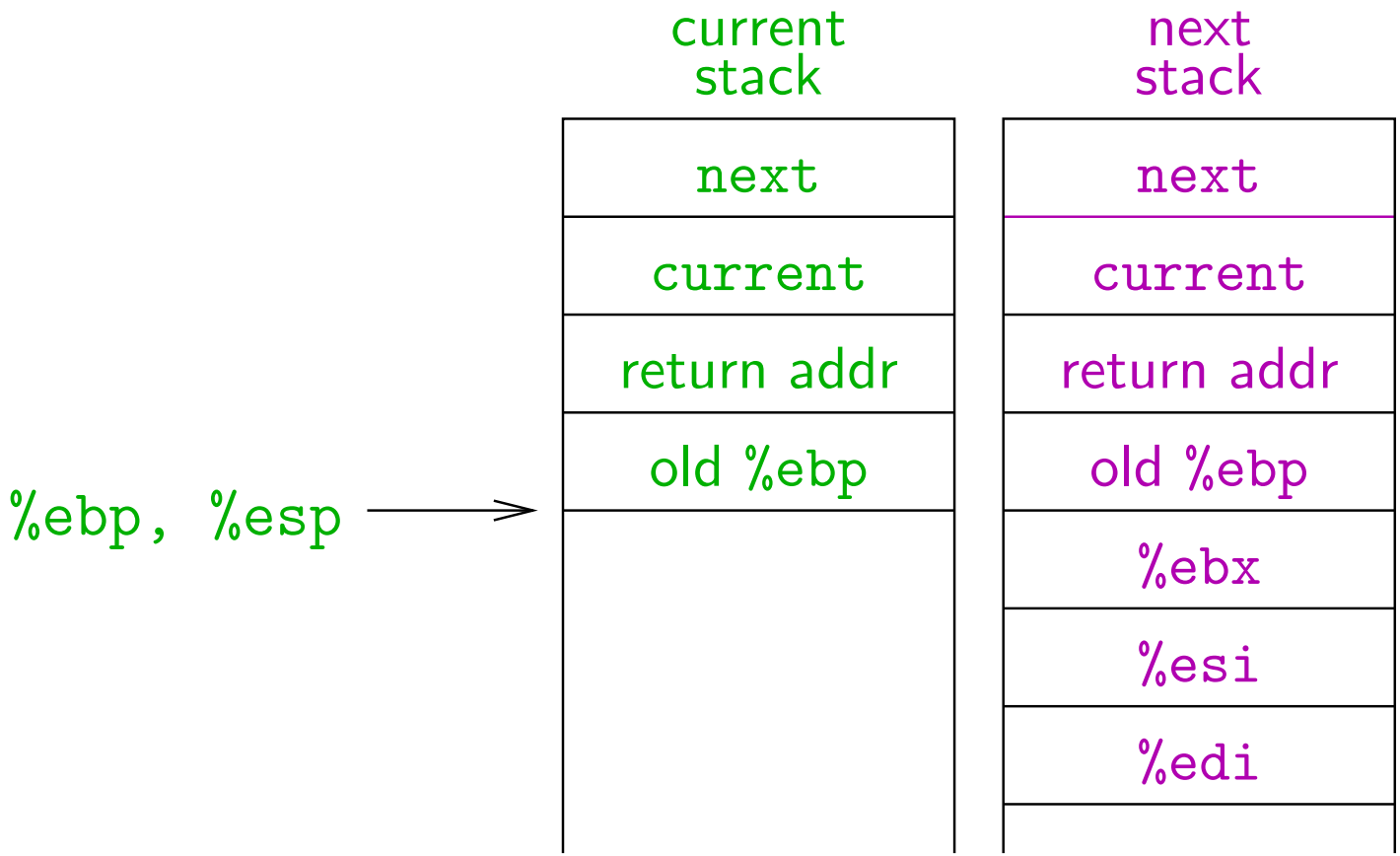
i386 thread_md_switch

```
pushl %ebp;                                  # On sauve old_epb
movl %esp,%ebp                               # new_ebp pointe sur old_ebp
pushl %ebx; pushl %esi; pushl %edi           # Sauve callee-saved

movl 8(%ebp),%edx                            # %edx = thread_current
movl 12(%ebp),%eax                           # %eax = thread_next
movl %esp,(%edx)                             # %edx->md_esp = %esp
movl (%eax),%esp                             # %esp = %eax->md_esp

popl %edi; popl %esi; popl %ebx              # Restaure callee-saved
popl %ebp                                    # Restaure ebp
ret                                          # continue l'execution
```

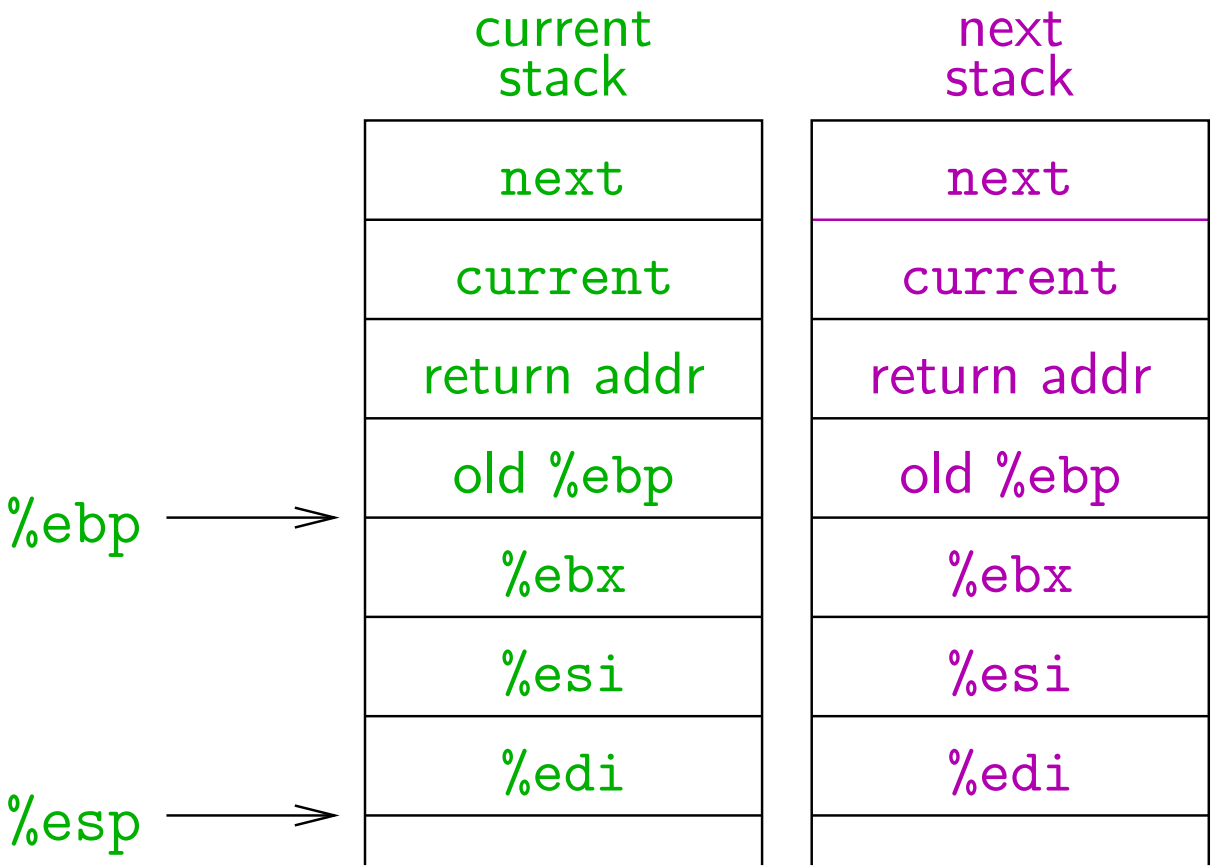
i386 thread_md_switch



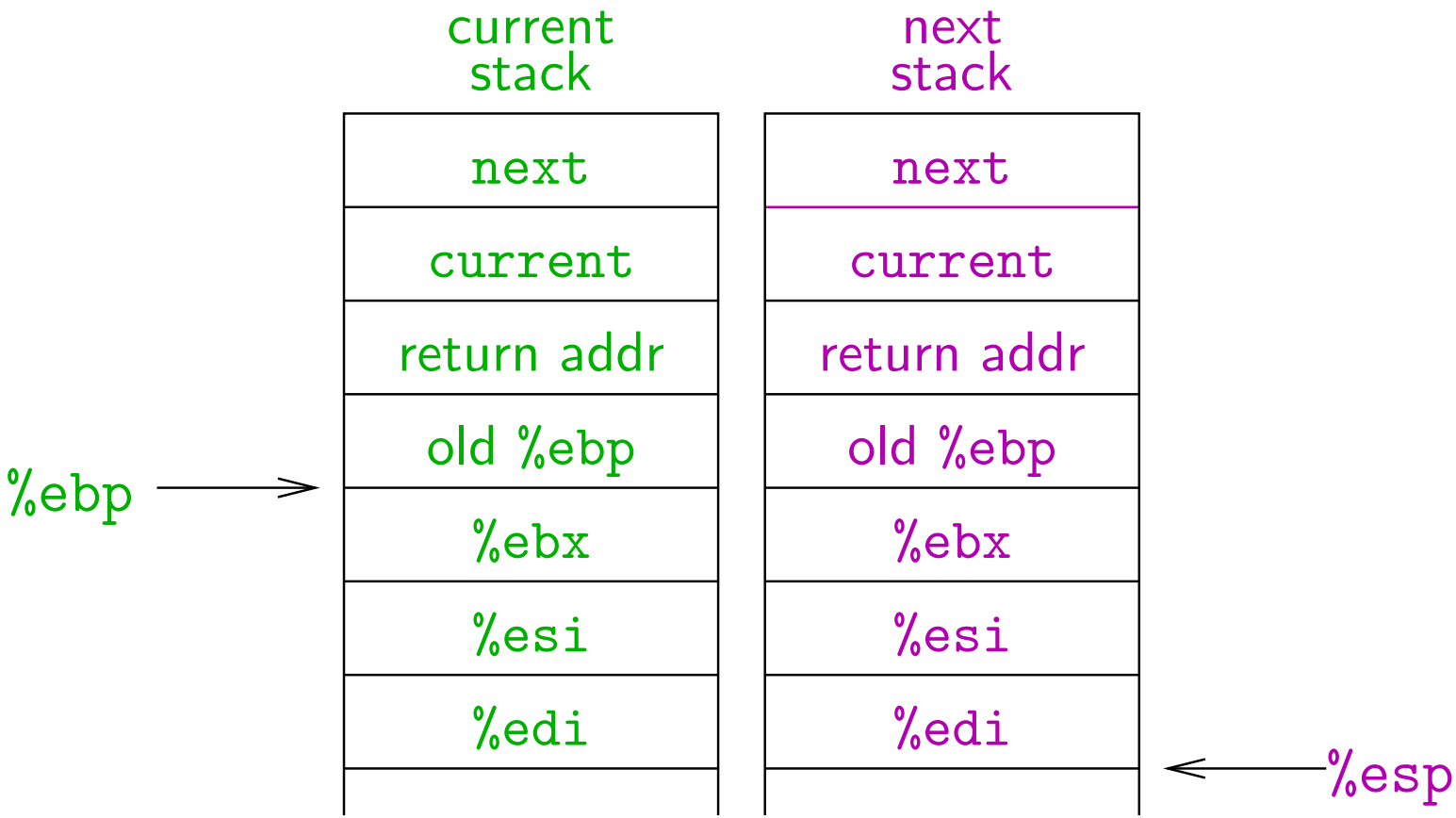
i386 thread_md_switch



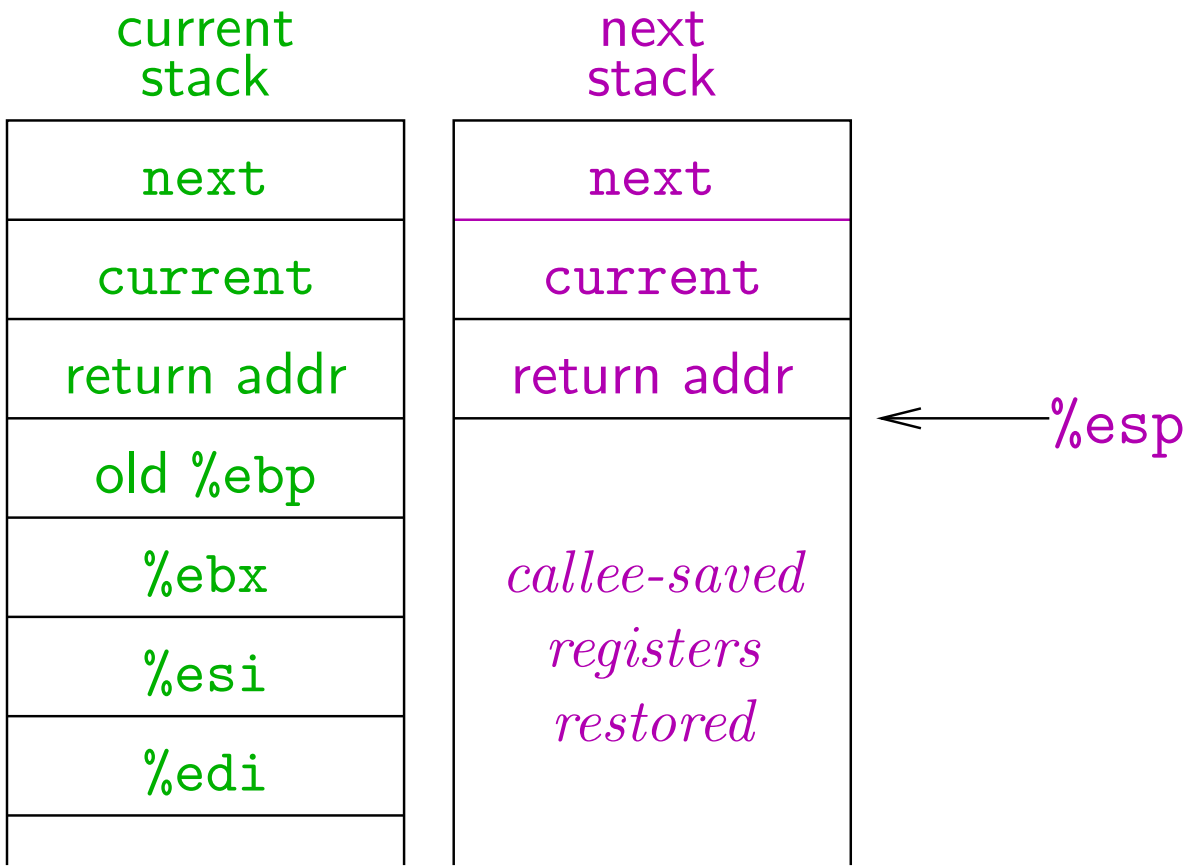
i386 thread_md_switch



i386 thread_md_switch



i386 thread_md_switch



Limitations des threads utilisateurs

- Ne peuvent pas utiliser plusieurs CPUs
- Un appel système bloquant, bloque l'ensemble des threads
  - Peut remplacer `read` pour les connexions reseau
  - Mais la plus part des SE ne permettent pas l'accès asynchrone au disque
- Une faute de page bloque tous les threads

Résumé

- Threads peuvent être facilement implémentés avec une librairie
  - mais les threads noyau ne sont pas forcément la meilleure interface
- Les threads sont tout de même très utiles
  - Performant pour la plus part des usages
  - Thread noyaux si l'objectif est d'exploiter la concurrence des E/S
  - Thread utilisateur pour des application avec beaucoup de changements de contexte (e.g,. applications calcul scientifique)
- Mais les programmes concurrents sont difficiles à débugger...