
Linux 2.6 interne

Concurrence & ordonnancement

Ronan.Keryell@enstb.org

—

Département Informatique
École Nationale Supérieure des Télécommunications de Bretagne

11 mai 2005

Plan

1

- ✍ Introduction concurrence & ordonnancement
- Tâches, processus lourds et légers dans Linux
- Ordonnancement dans Linux
- Conclusion



Ce cours est la première partie d'un cours plus complet incluant :

- ▶ Ordonnancement : politiques, différences entre Linux 2.4 et 2.6
- ▶ Aspect NETFILTER et IProute-2 et IPsec
- ▶ Aspects noyau et temps réel
- ▶ Fonctions auxiliaires : tasklets, softirq,...
- ▶ Rajouter un protocole dans la pile IP



Bibliographie

- Que les sources soient avec vous ! ☺
 - ▶ Récupérer le noyau Linux <http://kernel.org>
 - ▶ Le code hypertextuel <http://lxr.linux.no> (et plus proche de nous <http://kernel.enstb.org>)
 - ▶ Penser à utiliser les *tags* dans son éditeur favori
 - `make TAGS` crée un fichier TAGS pour les Emacs
 - `make tags` crée un fichier tags pour les vi
- Livres
 - ▶ « *Linux Kernel Development* » Robert LOVE. Novell Press, 2^{ème} édition, 12 janvier 2005
 - ▶ « *Linux Device Drivers* », Jonathan CORBET, Alessandro RUBINI et Greg KROAH-HARTMAN. O'Reilly, 3^{ème} édition, février 2005. <http://lwn.net/Kernel/LDD3>
- Gazette de discussion sur Linux <http://www.kernel-traffic.org>



- <http://lwn.net/Kernel> Linux kernel development
- The linux-kernel mailing list FAQ <http://www.tux.org/lkml> (un peu vieux)
- Index of Documentation for People Interested in Writing and/or Understanding the Linux Kernel
<http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html>
- <http://lwn.net/Articles/2.6-kernel-api> *2.6 API changes. A regularly-updated summary of changes to the internal kernel API*
- <http://lwn.net/Articles/driver-porting> The Porting drivers to 2.6 series: over 30 articles describing, in detail, how the internal kernel API has changed in the 2.6 release



Système d'exploitation

5

Fournit 2 types de services

- Machine virtuelle étendue plus agréable que la vraie machine brute
 - ▶ Plusieurs programmes fonctionnent en même temps
 - ▶ Plusieurs utilisateurs
 - ▶ Mémoire arbitrairement grande
 - ▶ Fichiers
 - ▶ Interfaces (graphiques) sympathiques
 - ▶ ...

Détails cachés de manière *transparente*

- Gestion optimale des ressources
 - ▶ Processeurs
 - ▶ Mémoire



- ▶ Périphériques d'entrée-sortie
- ▶ Gestion de la qualité de service (surtout en mode multi-utilisateur...)
 - Latence, temps de réponse : mode interactif
 - Débit : centre de calcul
 - Contraintes temps réel : gestion d'un processus industriel
 - Tolérance aux pannes : centrales nucléaires, avions
 - ...



Machine Virtuelle Étendue

- Processus : abstraction de processeur virtuel
 - ▶ Programme s'exécutant sur un processeur
 - ▶ Contexte d'exécution
 - ▶ Protection
- Atomicité : qui *paraît* insécable
 - ▶ Lecture, écriture
 - ▶ Transactions : début, fin, abandon possible sans casse
- Fichier abstrait
 - ▶ Conteneur de données, de programmes, répertoires
 - ▶ Périphérique (disquette)
 - ▶ Contrôle de n'importe quoi dans Unix...



- Applications & algorithmes complexes avec des choses indépendantes
- Simplification de la programmation : détacher les tâches les unes des autres
- Gestion explicite des tâches
 - ▶ Co-routines
 - ▶ `setjmp()/longjmp()`
 - ▶ Gestion à la main... ☹
- Si plusieurs processeurs : possibilité d'exécuter plusieurs tâches en parallèle



- Besoin de simplification de la programmation (productivité)
- Abstraction de la notion de tâche
 - ▶ Programme s'exécutant (actif) sur un processeur virtuel \rightsquigarrow processus (lourd) dans Unix
 - ▶ Contexte d'exécution : ressources virtuelles (espace mémoire, fichiers,...)
 - ▶ Atomicité possible (transactions,...)
- Gestion directe par le système d'exploitation

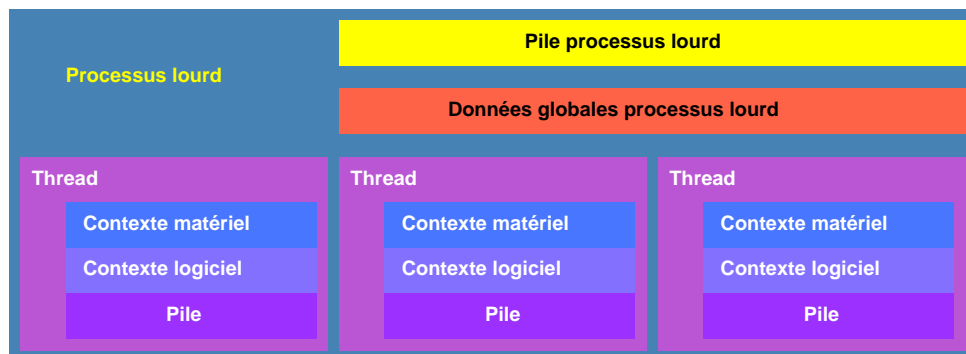


- Besoin de concurrence plus fine dans un programme
- Processus lourd peut contenir plusieurs fibres d'exécutions (*threads of execution*, processus légers)
- Partagent au sein d'un processus lourd Unix
 - ▶ Mémoire (données globales)
 - ▶ Fichiers
 - ▶ Signaux
 - ▶ Données internes au noyau (état processeur,...)
- Mais les threads disposent en propre
 - ▶ Numéro (*thread-id*)
 - ▶ Contexte matériel : image des registres de la machine (compteur ordinal, pointeur de pile,...)
 - ▶ Pile



- ▶ Masque de signaux UNIX
- ▶ Priorité
- ▶ Mémoire locale via le tas global et la pile





Des processus sans système d'exploitation ?

13

- Processus et concurrence \equiv concept de programmation important
- Difficile de porter tous les SE sur toutes les plateformes ☺
- Pour des systèmes embarqués, pas toujours les ressources pour : microcontrôleur minuscule,...
- Idée si on ne veut pas écrire des co-routines à la main : traduire (compiler) les appels systèmes thread POSIX du programme en programme qui gère les tâches à la main dans espace utilisateur
- Génère un programme C séquentiel compilable par un compilateur C pour n'importe quoi
 - ▶ *Atomic execution block* (AEB)
 - ▶ séparés par du code qui gère le choix des AEB à exécuter
- Alexander G. DEAN. *Compiling for concurrency: Planning and*

performing software thread integration. In Proceedings of the 23rd IEEE Real-Time Systems Symposium, Austin, TX, Dec 2003.

- *Lightweight Multitasking Support for Embedded Systems using the Phantom Serializing Compiler*, André C. NÁCUL and Tony GIVARGIS, DATE2005



- Introduction concurrence & ordonnancement
- ✍ Tâches, processus lourds et légers dans Linux
- Ordonnancement dans Linux
- Conclusion



- Plusieurs tâches concurrentes dans le noyau, voire parallèles si plusieurs processeurs
- Interruptions asynchrones
- Préemption en standard dans noyau Linux 2.6 : tâches du noyau peuvent être aussi interrompues
- Meilleure réactivité... ☺
- ...source de bugs améliorée ☺
- Bien soigner partage de ressources pour éviter interblocages
- Ne pas tomber dans excès inverse d'avant 2.6 : gros cli/sti



Synchronisation & concurrence

- Application utilisateur
 - ▶ Si application mono-thread pas de problème de partage de ressource
 - ▶ Si multithread/multiprocessus, possibilité de conflits d'accès à des ressources, interblocages,...
 - ▶ ⇨ Utilisation de primitives atomiques : verrous,...
 - ▶ Dans le pire des cas, arrêt des tâches possibles
- Tâche dans le noyau
 - ▶ Intrinsèquement multitâche
 - ▶ ⇨ Utilisation de primitives atomiques : verrous,...
 - ▶ Pas de système d'exploitation pour veiller...
 - ▶ ... Si conflits d'accès ou étreintes mortelles : plantage du système ☹

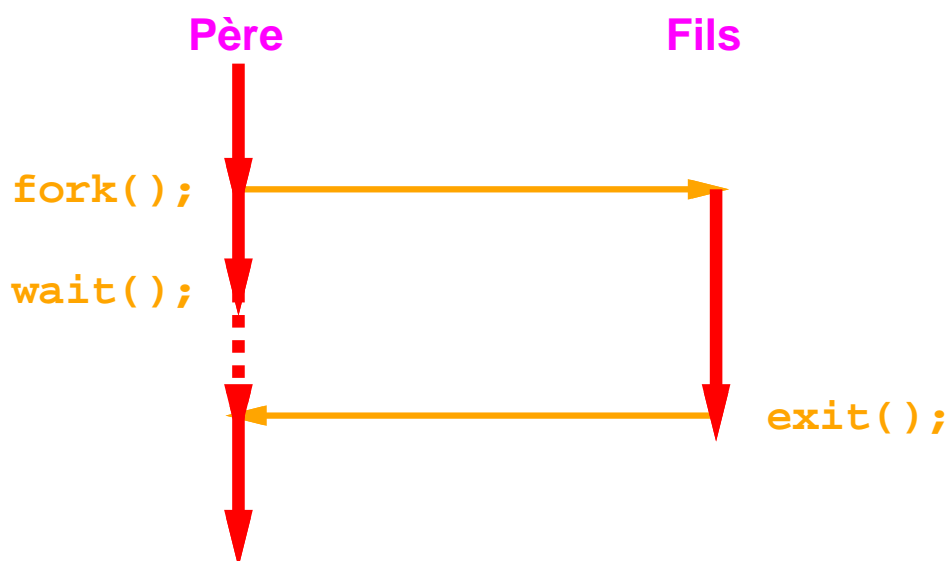


- Un processus lourd ou léger en cours d'exécution en mode utilisateur
- Un processus lourd ou léger en cours d'exécution en mode noyau
- Une tâche noyau en cours d'exécution
- Mode interruption dans noyau



Vie et mort d'un processus version Unix

19



- `fork()` crée un clone du père
 - Ne partagent que les valeurs initiales

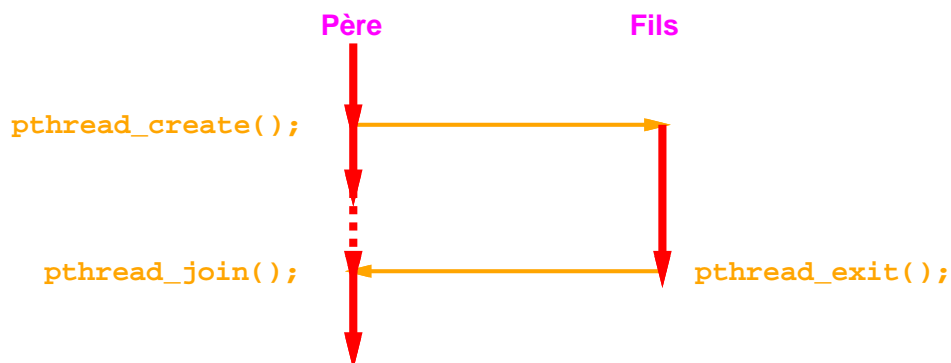


- Souvent suivi d'un `exec()` pour exécuter un autre programme dans le processus
 - ▶ En fait optimisation de `fork()`: partage des pages de mémoire et copie seulement lors de la première écriture (COW *Copy On Write*)
 - ▶ Par le passé pré-COW, introduction du `vfork()` BSD optimisé pour un `exec()` : le fils *utilise* l'espace mémoire du père qui est rendu après l'`exec()`... mais appel système toujours présent



Vie et mort d'un processus version thread

Thread POSIX



- Créer un *thread*

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr = null,
                  void *(*entry) (void *),
                  void *arg)
```

entry indique la fonction à exécuter avec les paramètres arg

- Pour terminer

```
void pthread_exit(void *status)
```

Transmet dans status une cause de terminaison

- Attendre la fin d'un fils *thread*

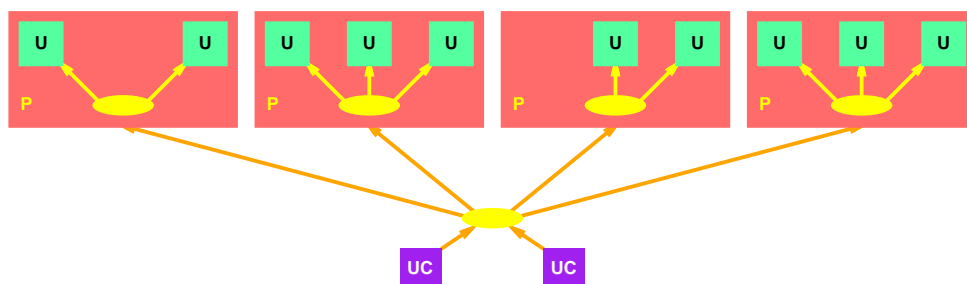
```
int pthread_join(pthread_t *thread,
                 void **status) ;
```

Récupère dans status une information sur la cause de la terminaison




2 niveaux d'ordonnanceur

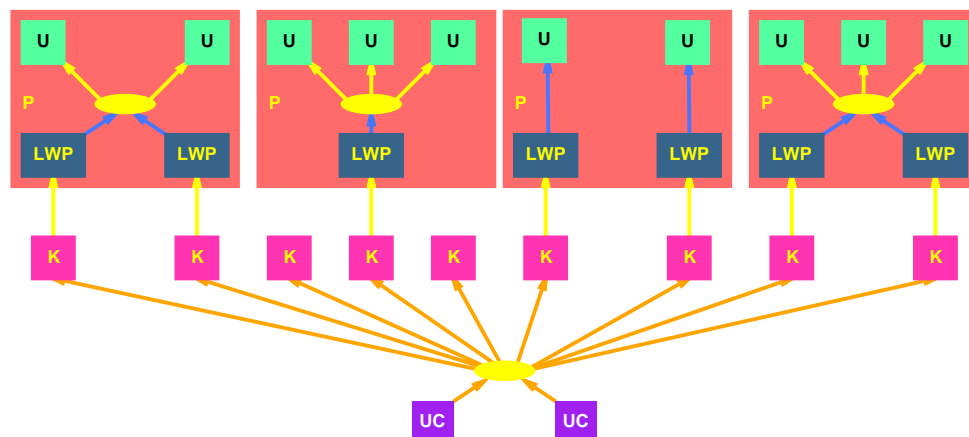
Où choisir de multiplexer exécution concurrente ?



- Ordonnanceur du noyau : processus plutôt lourds
- Ordonnanceur utilisateur dans chaque processus lourds :
threads plutôt légères

 blocages sur E/S... ☹





Les processus légers dans Linux

25

- Pas de gestion spécifique dans Linux (contrairement à Solaris (LWP) ou Windows)
- Processus léger \equiv processus lourd comme un autre... où on précise qu'on partage certaines choses (mémoire,...)
- Suppose que processus gérés de manière naturellement « légère » (création, changement de contexte,...)
- Si pas suffisant, utiliser une bibliothèque niveau utilisateur



```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

Linux propose appel système `clone()` permettant de choisir ce qui est partagé

- ▶ Espace mémoire
- ▶ Descripteurs de fichiers (fichiers ouverts)
- ▶ Gestion des signaux
- ▶ Espace de nommage du système de fichiers



Différents usages de clone()

27

- `sys_fork`(arch/i386/kernel/process.c)

```
asm linkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
```

- `sys_vfork`(arch/i386/kernel/process.c)

```
asm linkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
```

- `sys_clone`(arch/i386/kernel/process.c)

```
asm linkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;
```



```

clone_flags = regs.ebx;
newsp = regs.ecx;
parent_tidptr = (int __user *)regs.edx;
child_tidptr = (int __user *)regs.edi;
if (!newsp)
    newsp = regs.esp;
return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
}

```

Le vrai boulot : do_fork(kernel/fork.c)

```

long do_fork(unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs *regs,
            unsigned long stack_size,
            int __user *parent_tidptr,
            int __user *child_tidptr)
{
    struct task_struct *p;
    long pid = alloc_pidmap();

```



```

...
p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, pid,
...
if (!(clone_flags & CLONE_STOPPED))
    wake_up_new_task(p, clone_flags);
else
    p->state = TASK_STOPPED;
if (clone_flags & CLONE_VFORK)
    wait_for_completion(&vfork);
return pid;
}


```

- ▶ Le fils est réveillé : essaye de le faire tourner avant le père pour optimiser le COW en cas d'exec() rapide
- ▶ copy_process() fait un dup_task_struct(current) et initialise la structure de la tâche




- Même le noyau peut avoir besoin de processus pour faire des choses de manière concurrente
- \rightsquigarrow tâches noyau (convention de nommage : tâche comme raccourci de processus noyau)
- Création ? Comme les processus utilisateurs
 - ▶ Si processus utilisateur, création de processus utilisateur avec `clone()` ou `(v)fork`
 - ▶ Si tâche noyau, `clone()` crée une tâche noyau
Exemple du chargement dynamique de module (si rajout de périphériques,...)

Remarques

-  Un processus utilisateur ne peut créer que des processus utilisateurs... Si besoin tâche noyau : modifier le noyau ou mettre dans un module



-  Une tâche noyau n'a aucun mode de protection, accède à toutes les ressources,... Sans filet !



- Outils ps, top,...

```
keryell@an-dro:~$ ps auxww
```

UUSER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	1584	80	?	S	04:46	0:00	init [5]
root	2	0.0	0.0	0	0	?	SN	04:46	0:00	[ksoftirqd/0]
root	3	0.0	0.0	0	0	?	S<	04:46	0:00	[events/0]
root	2141	0.0	0.0	2984	176	?	Ss	04:46	0:00	/sbin/klogd
bind	2171	0.0	0.1	29740	736	?	Ssl	04:46	0:00	/usr/sbin/named
...										
keryell	9620	0.0	0.1	4612	904	pts/4	R+	08:52	0:00	ps auxww

- Des entrées dans /proc pour chaque processus dont *self*



Représentation des tâches en interne

- Chaque processus est représenté par une `task_struct` (include/linux/sched.h) de 1,7 Ko sur ordinateur 32 bits
- Contient tout ce que le noyau a besoin de connaître sur un processus pour le faire fonctionner
 - ▶ État (bloqué ou pas)
 - ▶ Espace mémoire
 - ▶ Exécutable associé
 - ▶ Parenté
 - ▶ Droits, capacités
 - ▶ Fichiers ouverts
 - ▶ Espace de nommage (montages particuliers, autre racine,...)
 - ▶ Domaines d'exécutions (simulation d'autres systèmes,...)



- ▶ Audit
- ▶ Files d'attentes d'entrées/sorties
- ▶ Statistiques d'usage
- ▶ Priorité
- ▶ Gestion des machines parallèles NUMA
- ▶ ...
- Chaque processus a une (petite) pile dans le noyau
- Chaque processus utilisateur a aussi une pile dans espace mémoire utilisateur
- Reliés par une double liste chaînée
- Tout n'est pas nécessaire pour faire des changement de tâches
 - ▶ Hiérarchisation
 - ▶ Le minimum vital est en haut de la pile noyau du processus



- ▶ Rapide (cache) et facile (déplacement pointeur de pile) à accéder

thread_info(linux/thread_info.h)

```
struct thread_info {
    struct task_struct *task;           /* main task structure */
    struct exec_domain *exec_domain;    /* execution domain */
    unsigned long flags;                /* low level flags */
    unsigned long status;               /* thread-synchronous flags */
    __u32 cpu;                          /* current CPU */
    __s32 preempt_count;                /* 0 => preemptable, <0 => BUG */
    mm_segment_t addr_limit;            /* thread address space */
    struct restart_block restart_block;
    unsigned long previous_esp;         /* ESP of the previous stack in case
                                         of nested (IRQ) stacks */
    __u8 supervisor_stack[0];
};
```



- `current_thread_info()` pointe vers le descripteur courant

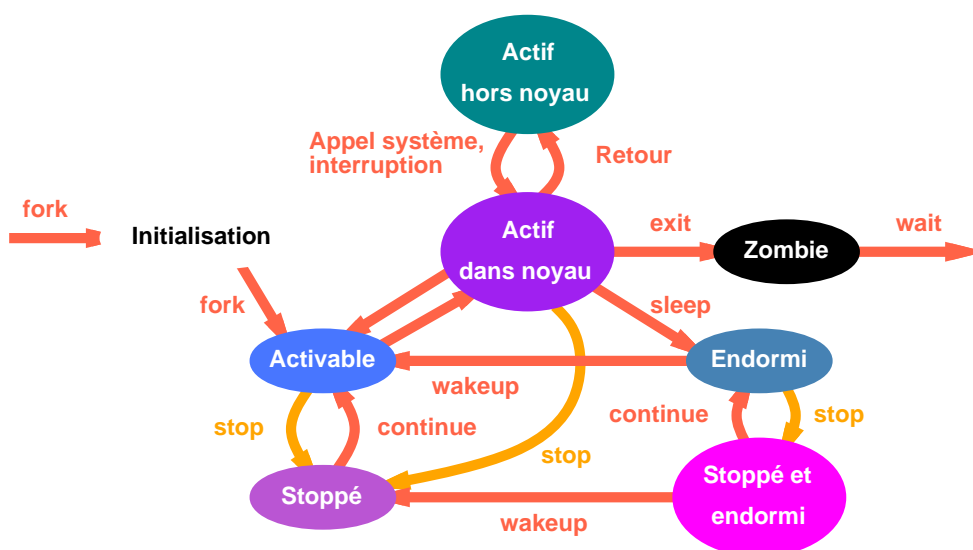
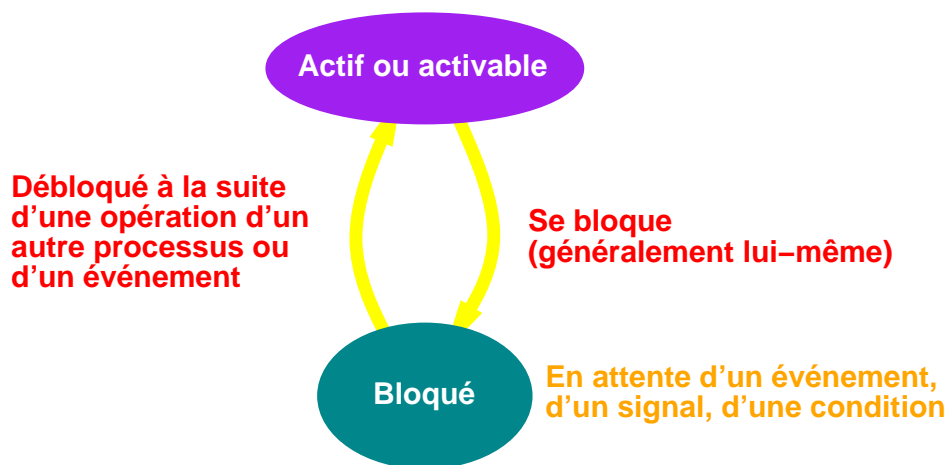
```
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__ ("andl_%%esp,%0;_r": "=r" (ti) : "" (~(THREAD_SIZE - 1)));
    return ti;
}
```

- `current()` pointe vers le descripteur courant de la `task_info` complète
- Adresse pas pratique pour un utilisateur (si processus migre,...)
 - ↪ notion de *pid* sur 16 ou 32 bits dans `task_info`
 - ▶ Limite dans `/proc/sys/kernel/pid_max` changeable pour gros serveurs 64 bits...
 - ▶ Allocation de *pid* par un bitmap
 - ▶ Fonction de hachage pour retrouver rapidement une tâche à partir de son *pid*



- `kernel/pid.c`





Dans descripteur processus `task_struct` (include/linux/sched.h) :
champ `task`

- ▶ `#define TASK_RUNNING 0` : le processus est en train de fonctionner ou est sur une file d'attente pour. Si processus en mode utilisateur forcé en train de... s'exécuter ☺
R dans `ps/top`
- ▶ Le processus est bloqué en attente (endormi) d'un événement ou entrée-sortie pour repasser dans l'état `TASK_RUNNING`
 - `#define TASK_INTERRUPTIBLE 1` : le processus peut être réveillé aussi par un signal
S dans `ps/top`
 - `#define TASK_UNINTERRUPTIBLE 2` : idem mais ne peut pas être réveillé par un signal. Utile si processus veut dormir sans interruption, si événement attendu rapidement et réserve de

ressources importantes,...

D dans `ps/top`



Un signal `SIGKILL` ne peut même pas en venir à bout...

Mais peut-être voulu si des ressources ont été bloquées et ne seraient pas libérées. Problème si bug néanmoins ☺

↪ Éviter si possible

- ▶ `#define TASK_STOPPED 4` : est stoppé suite à `SIGTSTOP` (`^Z` du shell,...) `SIGSTOP` (ne peut pas être contré) ou suite à une entrée-sortie alors qu'il est en tâche de fond (`SIGTTIN` & `SIGTTOU`). Continue si `SIGCONT`
T dans `ps/top`
- ▶ `#define TASK_TRACED 8` : un processus est en train de tracer tout ce qu'il fait (`strace` pour lister appels systèmes, `gdb` pour déboguer,...)

- ▶ `#define EXIT_ZOMBIE 16` : le processus n'existe plus. État juste pour retourner `task_struct.exit_code` au processus parent quand il fera un `wait()`
 - Si pas de parent, `init` joue le rôle de parrain et fait un `wait()` de complaisance
 - Si parent mal écrit et ne fait jamais de `wait()`, développement des zombies... ☹
- ▶ `#define EXIT_DEAD 32` : paix à son âme



- Introduction concurrence & ordonnancement
- Tâches, processus lourds et légers dans Linux
- ✍ Ordonnancement dans Linux
- Conclusion



- Système d'exploitation multitâche
 - ▶ Plein de processus veulent tourner
 - ▶ Un ou plusieurs processeurs
 - ▶ De nombreuses possibilités
 - ▶ Lesquels faire tourner en premier ?
- 2 objectifs difficiles à atteindre
 - ▶ S'assurer d'un bon taux d'utilisation des processeurs
 - ▶ S'assurer que chaque processus a le service qu'il souhaite
- Satisfaction des processus interactif au détriment des travaux par lots
- Coût des changements de contexte, des transferts de mémoire principale-mémoire secondaire,...
- Processus souvent connus qu'à l'exécution ☹ ~→



ordonnancement dynamique



- ▶ Multitâche coopératif
 - Chaque processus a prévu de passer la main aux autres
 - Nécessite une architecture logicielle précise
 - Comportement assez prédictible
 - Difficile de faire des choses complexes
- ▶ Multitâche préemptif
 - Même si une tâche n'a pas prévu de s'arrêter le système peut en faire tourner une autre à la place après un quantum de temps
 - Globalement plus simple à mettre en place
 - Pas de garantie facile à assurer sur les contraintes

Systèmes d'exploitations modernes généralistes : font les deux
(Linux 2.6, Solaris,...)



Quelques politiques d'ordonnancement

- Premier Arrivé Premier Servi (PAPS) ou *First In First Out* (FIFO)
- Tourniquet (*round robin*) ou partage du temps
- Priorité statique (temps réel) ou dynamique
- *Shortest job first...* Mais nécessite de connaître la durée de la tâche (future)
- *Earliest deadline first...* Mais nécessite de connaître la date de fin (future)
- *Smaller period first (Rate Monotonic scheduling)...* Nécessite de préciser les intervalles de lancement
- ...



Programme \equiv calculs + E/S

... mais rarement équilibré. Souvent 2 aspects se dégagent :

- Processus orienté calcul
 - ▶ Gros calculs
 - ▶ Prémption fréquente gâcherait du temps (système, cache, swap,...)
 - ▶ Réactivité moins évidente à l'utilisateur
- Processus orienté entrées-sorties
 - ▶ Calculs souvent bloqués par des E/S
 - ▶ Prémption souvent évitée par blocage sur E/S
 - ▶ Réactivité plus évidente à l'utilisateur (si c'est lui l'E/S !)

D'un point de vue rentabilité processeur, intéressant d'avoir les 2 en même temps

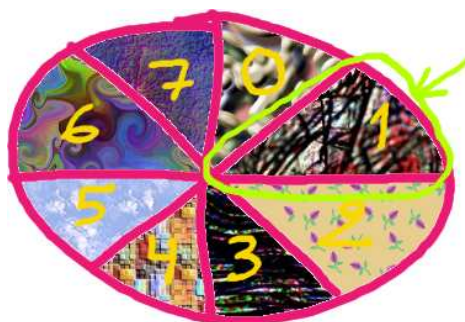


Temps partagé – tourniquet

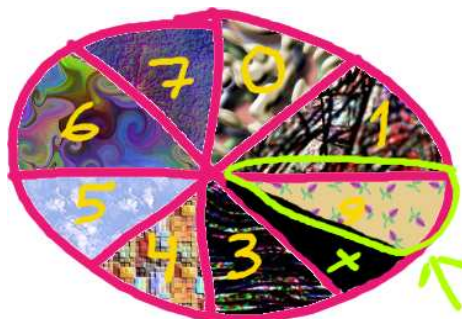
49

- Donner illusion de disposer d'une machine à soi tout seul
- Lié à l'invention du terminal interactif
- Changer de programme à chaque quantum de temps
- Privilégier les requêtes peu gourmandes par rapport aux programmes de calcul : faire

des heureux facilement avec
caisse moins de 10 Articles



- Blocage possible d'un processeur avant la fin de son quantum de temps
- Libère du temps pour les autres

**Politique par priorité****51**

Associer une priorité à chaque processus et faire fonctionner processus voulant tourner qui est le plus prioritaire

Priorité choisie en fonction de

- Objectifs globaux du système (système à temps partagé, traitement par lots, contrôle de procédés industriels,...)
- Caractéristiques de chaque processus (échéance temporelle, périodicité, temps processeur récemment consommé, temps récemment passé en sommeil,...) si on veut une politique à priorité plus dynamique
- Caractéristiques de chaque périphérique : ne pas ralentir des périphériques qui sont déjà lents,...

Priorité (p_{base})	Exemple	Type
+19 (+ faible)	Utilisateur d'écran	Plutôt utilisateur
:	:	
+1		
0 (standard)	Jeux vidéo	Plutôt système
-1	Numérisation d'un cours	
:	:	
-20 (+ forte)		

- Processus « noyau » ont une priorité (négative en Unix...) forte
- Processus utilisateurs ont une priorité (positive en Unix...) faible
- Un utilisateur peut seulement baisser la priorité d'un processus utilisateur ☺(modifie sa base) à l'aide de la commande `nice`



- Le super utilisateur peut augmenter la priorité
- Ordonnancement :
 - ▶ Faire tourner les processus de plus forte priorité entre eux en tourniquet
 - ▶ Un processus utilisateur qui a dépassé son quantum de temps est remis en queue de sa file d'attente (tourniquet)
 - ▶ Toutes les secondes, on recalcule les priorités sur le thème

$$p_{base} + \text{temps}_{\text{utilisation processeur}}$$

Avec p_{base} donné par la commande `nice` (0 par défaut)

- Rajout aussi d'ordonnanceur « temps réel » en plus dans les Unix modernes : permet d'avoir aussi des processus qui tournent avec des contraintes fortes



```
include/linux/sched.h
```

```
/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL tasks are
 * in the range MAX_RT_PRIO..MAX_PRIO-1. Priority values
 * are inverted: lower p->prio value means higher priority.
 *
 * The MAX_USER_RT_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space. This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */
#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO          MAX_USER_RT_PRIO

#define MAX_PRIO              (MAX_RT_PRIO + 40)
```

```
kernel/sched.c
```

```
/*
```



```
/* Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)

/*
 * 'User priority' is the nice value converted to something we
 * can work with better when scaling various scheduler parameters,
 * it's a [ 0 ... 39 ] range.
 */
#define USER_PRIO(p)            ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p)       USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO           (USER_PRIO(MAX_PRIO))
```



top permet de visualiser les processus de manière interactive

Tasks: 134 total, 5 running, 129 sleeping, 0 stopped, 0 zombie
 Cpu(s): 97.4% user, 2.6% system, 0.0% nice, 0.0% idle
 Mem: 514572k total, 505464k used, 9108k free, 41872k buffers
 Swap: 2048248k total, 57620k used, 1990628k free, 150544k cached

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
26487	even	16	0	185m	185m	784	R	95.7	36.8	158:20.05	prose
15440	keryell	9	0	2800	1944	1704	R	3.3	0.4	8:23.54	sshd
26540	keryell	11	0	1000	1000	768	R	0.7	0.2	0:00.60	top
17441	keryell	9	0	31244	28m	27m	S	0.3	5.6	12:18.96	mozilla-bin
1	root	9	0	436	412	388	S	0.0	0.1	1:26.93	init
2	root	9	0	0	0	0	S	0.0	0.0	0:00.14	keventd
3	root	9	0	0	0	0	S	0.0	0.0	0:36.84	kapmd
4	root	19	19	0	0	0	S	0.0	0.0	1:37.74	ksoftirqd_CPU0
5	root	9	0	0	0	0	S	0.0	0.0	59:20.41	kswapd



6	root	9	0	0	0	0	S	0.0	0.0	0:00.00	bdfush
7	root	9	0	0	0	0	S	0.0	0.0	1:29.13	kupdated
381	root	5	-10	90400	1480	1408	S	0.0	0.3	33:38.10	XFree86

Autres commandes : ps -ef (Système V), ps auxww (BSD),...



- Un quantum (*time slice*) identique pour tout le monde n'est pas la solution optimale
 - ▶ Trop long : peu interactif pour les autres
 - ▶ Trop court : trop de temps perdu en changements de contextes
- Clair qu'idéalement un petit quantum est préférable \rightsquigarrow souvent autour de 50 ms dans les Unix
- Mais tâche plus prioritaire tournera plus souvent même si petit quantum
- Idée : allouer un quantum variable en fonction de la priorité ou interactivité et faire un tourniquet sur tous ces processus jusqu'à épuisement des quanta
 - ▶ Quantum par défaut : 100 ms
 - ▶ Processus plus interactif ou plus prioritaire : tend vers 800 ms



- ▶ Processus moins interactif ou prioritaire : tend vers 5 ms
- ▶ Un quantum peut être consommé en plusieurs changement de contexte dans un tour de tourniquet (50 passage de 1 ms pour un processus très interactif par exemple)
- ▶ Recalcul des quanta après chaque tour
- Prémption lorsque
 - ▶ Un processus passe dans l'état `RUNNING` et que sa priorité est supérieure à celle du processus en cours d'exécution : changement de processus
 - ▶ Un processus a terminé son quantum de temps



- Bonne interactivité même si forte charge
- Essaye de respecter une certaine équité : pas de misère
- Optimise le cas courant de quelques processus actifs mais tient la charge
- Algorithme d'ordonnancement indépendant du nombre de processeurs : $\mathcal{O}(1)$
- Chaque processeur a sa liste de processus : bonne montée en charge parallèle (extensibilité) car pas de conflit
- Exploite la localité des processus sur processeurs (affinité) : meilleure utilisation des caches,...
- Tâche de migration de processus vers des processeurs moins chargés
- Gère le rajout et la disparition des processeurs



File d'exécution (*runqueue*)

61

- Tous les processus actifs/activables sont rangés dans une file d'exécution (*runqueue*)/processeur
- Un extrait de *runqueue*(kernel/sched.c) :

```
struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load;
#endif
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;

    unsigned long expired_timestamp; /* Date de l'échange d'arrays */
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm; /* Espace mémoire de la tâche précédente */
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
```



```
...  
/* Des choses pour le multiprocesseur et des statistiques */  
}
```



Tableaux de priorité

63

- Sur chaque file d'exécution (*runqueue*) avec n processus il faut
 - ▶ Trouver tâche plus prioritaire
 - ▶ Exécuter tâche plus prioritaire
 - ▶ La mettre après exécution si quantum expiré dans une liste des tâches ayant consommé leur quantum de temps

↪ Besoin d'une bonne structure de données pour éviter d'aller à la pêche en $\mathcal{O}(n)$ comme dans Linux 2.4...
- Idée : s'inspirer du *bucket sort* des facteurs de la poste
 - ▶ Mettre autant de files qu'il y a de niveaux de priorité
 - ▶ Mettre chaque tâche dans la file correspondante
 - ▶ Exécuter tâche de la file non vide de plus forte priorité

↪ Recherche en $\mathcal{O}(\#prio)$
- Certains processeurs ont des instructions pour trouver position



premier bit à 1 dans une chaîne (ffs,...) en $\mathcal{O}(\log \text{sizeof}(\text{int}))$,
en général 1 cycle


- ▶ Construire 1 tableau de bits associé aux files avec bit à 1 si file non vide
- ▶ Trouver en $\mathcal{O}(\frac{\#prio}{\text{sizeof}(\text{int})} \log \text{sizeof}(\text{int}))$ la position de la première file non vide
- ▶ Si 140 niveaux de priorité et mots de 32 bits, au plus 5 ffs pour parcourir le champ de 160 bits dans sched_find_first_bit()
- Comme cela ne dépend plus de n , on parle d'ordonnanceur à temps constant en $\mathcal{O}(1)$

```
#define BITMAP_SIZE (((MAX_PRIO+1+7)/8)+ sizeof(long)-1)/sizeof(long)
struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
};
```



```
struct list_head queue[MAX_PRIO];
};
```



- Idée de base
 - ▶ Itérer sur toutes les tâches
 - ▶ Si une tâche a usé son quantum de temps, le recalculer (en fonction de plein de paramètres : priorité, interactivité, histoire,...)
 - ▶  Grosse boucle $\mathcal{O}(n)$ avec des mécanismes d'exclusion mutuelle partout assez incompatible avec du temps réel... ☹
- Version Linux 2.6


```
struct runqueue {
    ...
    prio_array_t *active, *expired, arrays[2];
    ...
}
```

 - ▶ On alloue un nouveau tableau de priorité `expired` pour les tâches ayant usé leur temps



- ▶ Dès qu'une tâche de `active` a usé son quantum, on recalcule son quantum et on la met au bon endroit dans le tableau `expired`
- ▶ Lorsque le tableau de priorité `active` est vide on échange les 2 tableaux et on recommence
- ▶ Fait dans `schedule()(kernel/sched.c)`

```
array = rq->active;
if (unlikely (!array->nr_active)) {
    /*
     * Switch the active and expired arrays.
     */
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
} else
```



```
schedstat_inc(rq, sched_noswitch);
```

- Devient donc un petit $\mathcal{O}(n)$ moins violent que dans le 2.4



La fonction d'ordonnancement schedule()

- Appelée
 - Explicitement par une tâche noyau altruiste
 - À la fin d'un quantum de temps d'un processus
 - Après chaque changement de priorité ou d'état

- schedule()(kernel/sched.c)

```
idx = sched_find_first_bit(array->bitmap);  
queue = array->queue + idx;  
next = list_entry(queue->next, task_t, run_list);
```



- Un processus utilisateur a par défaut une priorité de sympathie envers les autres ($nice \in [-20, 19]$) : `static_prio`
- `effective_prio()` (kernel/sched.c) calcule la priorité dynamique `prio`

```
/*
 * effective_prio — return the priority that is based on the static
 * priority but is modified by bonuses/penalties.
 *
 * We scale the actual sleep average [0 .... MAX_SLEEP_AVG]
 * into the -5 ... 0 ... +5 bonus/penalty range.
 *
 * We use 25% of the full 0...39 priority range so that:
 *
 * 1) nice +19 interactive tasks do not preempt nice 0 CPU hogs.
 * 2) nice -20 CPU hogs do not get preempted by nice 0 tasks.
 *
 * Both properties are important to certain workloads.
 */
#define CURRENT_BONUS(p) \
```



```
(NS_TO_JIFFIES ((p) -> sleep_avg) * MAX_BONUS / \
MAX_SLEEP_AVG)
static int effective_prio (task_t *p)
{
    int bonus, prio;

    if (rt_task (p))
        return p -> prio;

    bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;

    prio = p -> static_prio - bonus;
    if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO - 1)
        prio = MAX_PRIO - 1;
    return prio;
}
```

- En gros si une tâche dort beaucoup elle est interactive et donc



elle gagne un bonus de priorité

- Un nouveau processus part avec un grand `sleep_avg` pour bien démarrer dans la vie
- ```
/*
 * task_timeslice() scales user-nice values [-20 ... 0 ... 19]
 * to time slice values: [800ms ... 100ms ... 5ms]
 *
 * The higher a thread's priority, the bigger timeslices
 * it gets during one round of execution. But even the lowest
 * priority thread gets MIN_TIMESLICE worth of execution time.
 */
#define MIN_TIMESLICE max(5 * HZ / 1000, 1)
#define DEF_TIMESLICE (100 * HZ / 1000)
#define SCALE_PRIO(x, prio) \
 max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)

static unsigned int task_timeslice(task_t *p)
{
 if (p->static_prio < NICE_TO_PRIO(0))
```



```
 return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
 else
 return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```



- Les tâches ont aussi besoin de se reposer
- Évite les attentes actives inutiles
- Permet de faire du travail utile pendant ce temps
- Gestion des états TASK\_INTERRUPTIBLE et TASK\_UNINTERRUPTIBLE
- Introduction de files d'attente (*wait queue*) pour collectionner processus à réveiller sur un événement particulier
- Le réveil des tâches en attente est fait par `__wake_up()`(kernel/sched.c)

Exemple de `do_clock_nanosleep()`(kernel/posix-timers.c)

```
long
do_clock_nanosleep(clockid_t which_clock, int flags, struct timespec *tsave)
{
```



```
...
/* éCre un éélément de liste abs_wqueue avec la tâche courante comme
 valeur, style abs_wqueue = CONS(current, NIL) */
DECLARE_WAITQUEUE(abs_wqueue, current);
...
init_timer(&new_timer);
new_timer.expires = 0;
new_timer.data = (unsigned long) current;
new_timer.function = nanosleep_wake_up;
abs = flags & TIMER_ABSTIME;
...
if (abs && (posix_clocks[which_clock].clock_get !=
 posix_clocks[CLOCK_MONOTONIC].clock_get))
 /* En gros fait
 nanosleep_abs_wqueue = CONS(current, nanosleep_abs_wqueue) */
 add_wait_queue(&nanosleep_abs_wqueue, &abs_wqueue);

new_timer.expires = jiffies + left;
__set_current_state(TASK_INTERRUPTIBLE);
add_timer(&new_timer);
```



```
schedule ();
...
return 0;
}
```



## Préemption

77

- Effectué par `schedule()`(kernel/sched.c) qui appelle `context_switch()`(kernel/sched.c)
- Peut arriver
  - ▶ Processus utilisateur : au moment de revenir dans l'espace utilisateur depuis un appel système ou une interruption
  - ▶ Tâche noyau : après un retour d'interruption, sur blocage, appel explicite à `schedule()` ou lorsque le noyau redevient préemptif
- Une tâche dans le noyau est préemptible si elle ne possède aucun verrou de posé (comptabilisés par le champ `preempt_count` de la `thread_info`)
- La préemption est globalement contrôlée par le drapeau `need_resched` qui est positionné si quelqu'un a besoin d'avoir la main



- Multiprocesseur capable d'exécuter plusieurs processus en parallèle
- Pour des raisons d'efficacité, tâches associées à un processus
- Il se peut que des processeurs soient beaucoup plus chargés que d'autres... ☹
- `schedule()`(kernel/sched.c) appelle régulièrement `load_balance()`(kernel/sched.c)
  - ▶ Si pas de déséquilibre de plus de 25 % ne fait rien
  - ▶ Sinon prend une tâche de haute priorité (à équilibre principalement) qui n'a pas tourné depuis longtemps (cache...) et essaye de la bouger



## Affinité tâche-processeur

## 79

- Une tâche tourne sur *un* processeur
- Est-ce indépendant de la localisation ?
  - ▶ Fonctionnellement oui...
  - ▶ ... ⚠ performances !
- Architecture sous-jacente complexe
  - ▶ Mémoires caches dans les processeurs
  - ▶ Architectures NUMA (*Non Uniform Memory Access*)
  - ▶ Architectures hétérogènes : cartes réseaux et processeurs
    - Faire tourner un processus qui traite des paquets d'une interface réseau sur un processeur proche de celle-ci

↪ Besoin de contrôler finement la localisation

Cf. man de `taskset(1)`, `sched_setaffinity(2)`,  
`sched_getaffinity(2)`





Interaction avec mémoire virtuelle et mémoire secondaire car 1 processeur ne peut exécuter que des instructions en mémoire principale

- Au niveau bas : ordonnanceur choisit de rentre actif 1 processus activable *présent en mémoire principale*
- Au niveau haut : ordonnanceur gère le va-et-vient (*swap*) entre la mémoire principale et la mémoire secondaire. Modifie les priorités pour favoriser processus qui viennent de rentrer en mémoire principale, etc.

Choix averti de l'ordonnanceur  $\rightsquigarrow$  noyau a constamment en mémoire principale une table de l'ensemble des processus avec données nécessaire au fonctionnement de l'ordonnanceur



## **Ordonnancement et va et vient**

**81**

- Certaines informations liées à un processus restent en mémoire principale
  - ▶ Paramètres d'ordonnancement
  - ▶ Adresses sur disques des segments/pages du processus
  - ▶ Informations sur les signaux UNIX acceptés
  - ▶ ...
- D'autres informations suivent le processus quand il est mis (*swap out*) en mémoire secondaire
  - ▶ Contexte matériel
  - ▶ Table de descripteurs de fichiers
  - ▶ Pile du noyau (stockage des variables locales du noyau lorsqu'il traite le processus) et la pile « utilisateur » (stockage des variables locales lors des calculs)



En plus de la politique non temps réel `SCHED_NORMAL` il y a 2 politiques plus temps réel dans Linux 2.6

- `SCHED_FIFO`
    - ▶ Une telle tâche passera toujours avant une `SCHED_NORMAL`
    - ▶ Prémption
    - ▶ Tourne tant qu'elle n'est pas bloquée ou fait un `sched_yield()` ou une tâche temps réel plus prioritaire veut tourner
  - `SCHED_RR`
    - ▶ Comme `SCHED_FIFO` mais avec des quanta de temps
    - ▶ Tourniquet entre processus de même priorité
- Pas de garantie dure mais permet de faire plus de chose que `SCHED_NORMAL` classique de base



Cf. `man chrt(1)`, `man sched_setscheduler(2)`



Supposons

- 3 processus  $P_1$ ,  $P_2$  et  $P_3$
- Un système à priorités fixes dures
- Priorités  $p(P_1) > p(P_2) > p(P_3)$



Cas pathologique :



- $P_1$  attend un message de  $P_3$
- $P_2$  fonctionne à la place de  $P_3$  car plus prioritaire
- Paradoxe :  $P_2$  fonctionne à la place de  $P_1$  et est donc plus prioritaire !



Solutions possibles :

- Héritage de priorité : faire hériter (provisoirement !)  $P_3$  de la priorité de  $P_1$
- *Priority Ceiling Algorithms* : associer à une ressource une priorité qui sera prêtée aux tâches qui utilisent ou attendent cette ressource
- Pas géré dans Linux de base mais facile d'introduire des mécanismes de propagation de priorité (cf. TimeSys,...)



## Mars Pathfinder Mission on July 4th, 1997

87

[http://www.research.microsoft.com/~mbj/Mars\\_Pathfinder](http://www.research.microsoft.com/~mbj/Mars_Pathfinder)

The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. Successes included its unconventional "landing"-bouncing onto the Martian surface surrounded by airbags, deploying the Sojourner rover, and gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web.



A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus (1553 bus). Access to the bus was synchronized with mutual exclusion locks (mutexes).



The meteorological data gathering task (ASI/MET) ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.



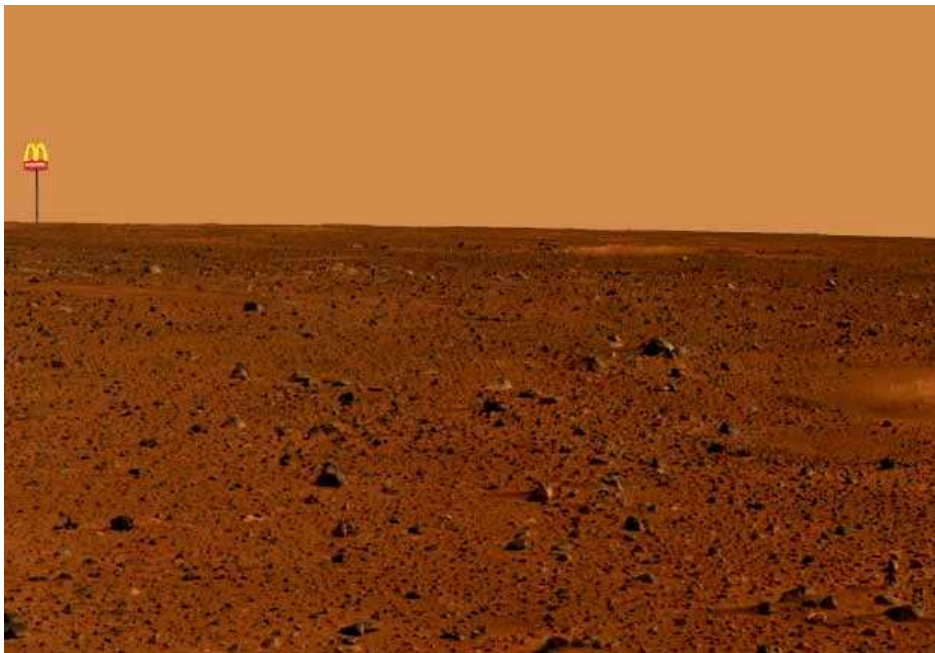
The spacecraft also contained a bus communications task that ran with medium priority.



Most of the time this combination worked fine. However, very infrequently, it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.



- Vitesse de la lumière Terre-Mars  $\approx 14$  mn ☹
- Récupérer une version du système temps réel (vxWorks) spécialement instrumentée pour le debug avec collecte de trace d'exécutions
- Faire tourner avec les mêmes tâches que sur Mars une maquette identique
- Bug apparu au bout de 18 heures
- Analyse : sémaphore utilisé sans option d'héritage de priorité (non mis par défaut pour des raisons d'optimisation)
- Besoin de modifier le code sur... Mars !
- Conception d'une rustine logicielle (*patch*)
- Mise en place avec une procédure spéciale... qui avait été prévue !



- Introduction concurrence & ordonnancement
- Tâches, processus lourds et légers dans Linux
- Ordonnancement dans Linux
- ✎ Conclusion



- De gros progrès dans Linux 2.6 !
- Noyau préemptif
- Rajout de priorités fixes temps réel
- Ordonnanceur efficace et rapide : regarder  
<http://developer.osdl.org/craiger/hackbench/index.html>
- Nouvelles politiques d'ordonnancement temps réel en plus de Unix dynamique classique
- Parti du monde du PC Linux aborde sans complexe toute l'étendue informatique : des systèmes embarqués aux super-calculateurs parallèles NUMA
- Monde du logiciel libre
  - ▶ Pas captif d'un produit fermé
  - ▶ On a les sources pour regarder dedans et adapter !





- ▶ Plein de documentation et d'exemples disponibles
- ▶ Grande communauté (support gratuit ou payant)



## List of Slides

- 1 Plan
- 2 Déroulement
- 3 Bibliographie

### Introduction

- 5 Système d'exploitation
- 7 Machine Virtuelle Étendue
- 8 Concurrence et parallélisme
- 9 Tâches
- 10 Processus légers
- 12 Threads utilisateurs dans un processus
- 13 Des processus sans système d'exploitation ?
- 15 Plan
- 16 Préemption dans le noyau
- 17 Synchronisation & concurrence

### Tâches, processus lourds et légers dans Linux

- 18 4 états d'exécution possible
- 19 Vie et mort d'un processus version Unix
- 21 Vie et mort d'un processus version thread
- 22 Threads POSIX
- 23 2 niveaux d'ordonnanceur
- 24 Avec ordonnanceur de threads noyau
- 25 Les processus légers dans Linux
- 26 Quid entre processus Linux lourd et léger ?
- 27 Différents usages de clone()
- 30 Tâches noyau
- 32 Exemple de liste de processus
- 33 Représentation des tâches en interne
- 36 Accès aux tâches
- 38 Graphe des états d'un processus
- 39 Un processus Unix dans tous ses états
- 40 États internes dans Linux
- 43 Plan
- 44 Ordonnancement



## Ordonnancement dans Linux

- 46 Types de multitâche
- 47 Quelques politiques d'ordonnancement
- 48 Processus dirigé par le calcul ou les E/S
- 49 Temps partagé – tourniquet
- 50 Blocage dans le tourniquet
- 51 Politique par priorité
- 52 Politique à priorités dans Unix
- 54 Toutes les priorités dans Linux
- 56 Exemple de processus Unix avec top
- 58 Choix du quantum de temps dans Linux
- 60 Ordonnanceur Linux 2.6
- 61 File d'exécution (*runqueues*)
- 63 Tableaux de priorité
- 66 Recalculer les quantas de temps
- 69 La fonction d'ordonnancement `schedule()`
- 70 Calcul des priorités dynamiques et quantas
- 74 Sommeil & réveil

- 77 Prémption
- 78 Équilibrage de charge
- 79 Affinité tâche-processeur
- 80 Unix : 2 ordonnancement
- 81 Ordonnancement et va et vient
- 82 Politiques temps réel
- 84 Des ennuis : inversion de priorité
- 86 Solutions possibles
- 87 Mars Pathfinder Mission on July 4th, 1997
- 88 Tâche martienne de priorité forte
- 89 Tâche martienne de priorité faible
- 90 Tâche martienne de priorité moyenne
- 91 Bug martien
- 92 Debug martien
- 94 Plan
- 95 Conclusion

## Conclusion

- 97 Table des matières

