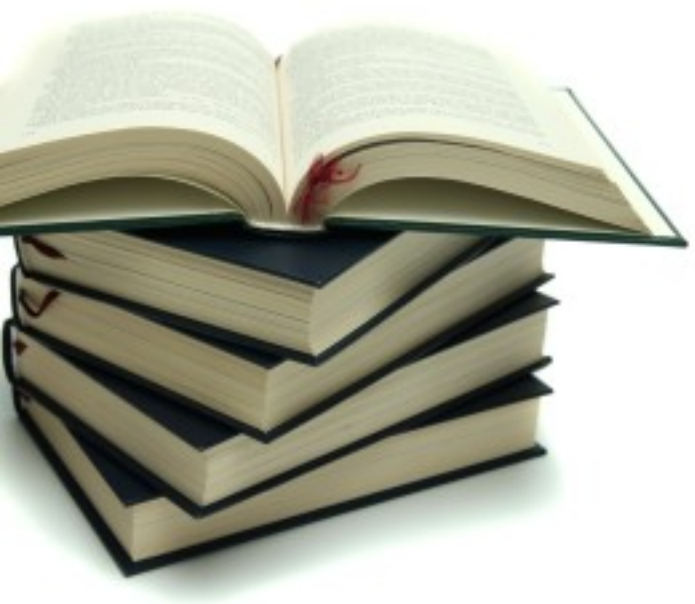
Test curs 2

- De ce **settimeofday** nu poate fi implementat ca un virtual system call (dar **gettimeofday** poate fi)?
- Pentru care din cele două secvențe de cod de mai jos se generează un page fault şi / sau un kernel BUG()?
  - memcpy(NULL, NULL, 1024);
  - copy_from_user(NULL, NULL, 1024);
- Care este stack frame-ul pentru funcția de mai jos?

```
int f(int a, int b) { int c, d; .... }
```

so2
kernel crunch

Cursul 3



3

Procese

11 martie 2010

Procese

- Implementarea proceselor şi threadurilor
- Schimbare de context
- Blocarea şi trezirea threadurilor
- Bibliografie
  - LKD: capitolul 3,4
  - UTLK: capitolul 3

Proces

- Un spațiu de adresă
- Fişiere, socketi
- Alte resurse: semafoare, zone de memorie partajată, etc.
- O stare (rulează, aşteaptă la I/O)
- Unul sau mai multe fire de execuție
- Structura din kernel ce descrie procesul este generic denumită Process Control Block (PCB)

Proces (2)

```
$ ls -1 /proc/self/
cmdline
cwd
environ
exe
fd
fdinfo
maps
mem
root
stat
statm
status
task
wchan
```

```
Name:   cat
State:  R (running)
Tgid:   18205
Pid:    18205
PPid:   18133
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
```

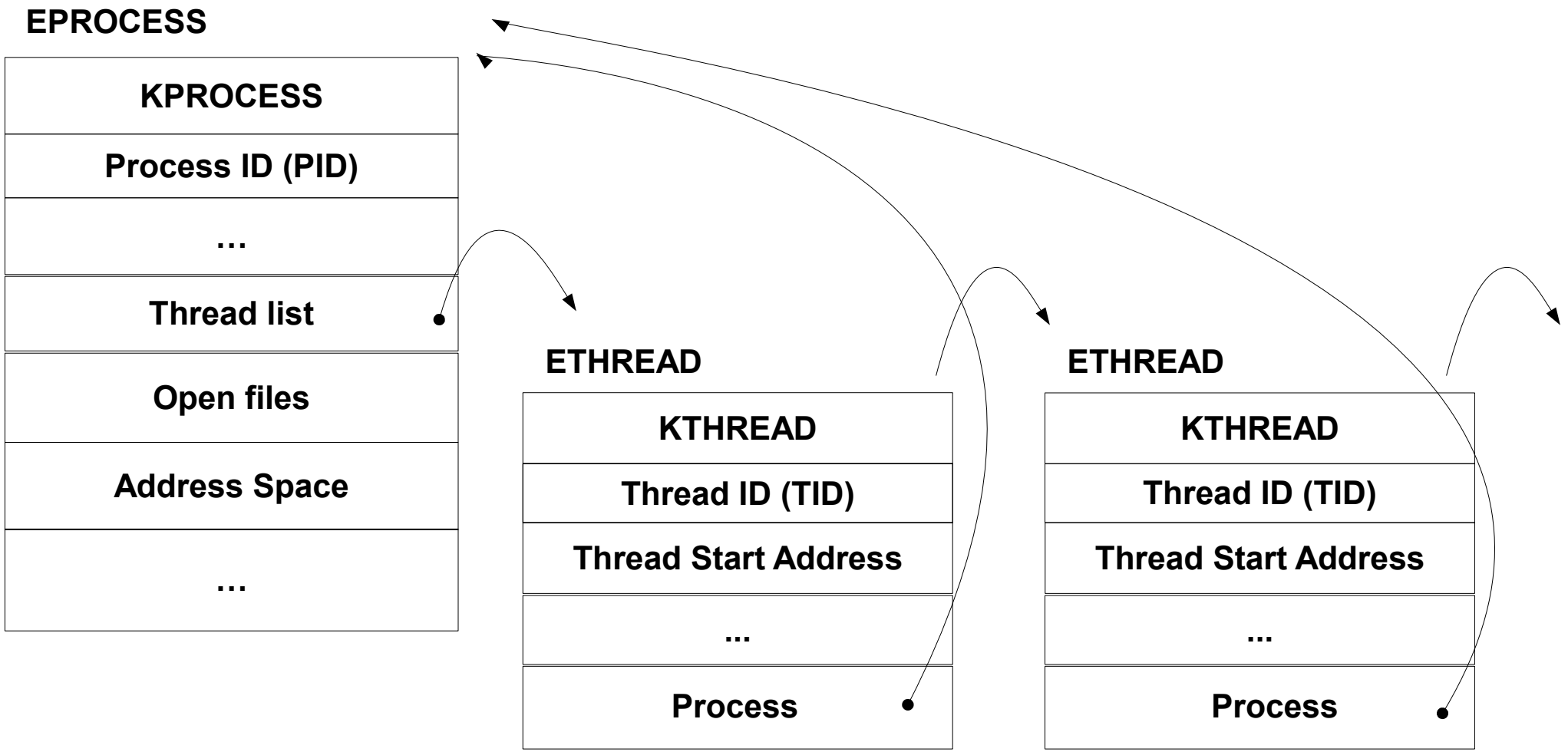
```
dr-x------ 2 tavi tavi  0 2008-03-27 12:34 .
dr-xr-xr-x 6 tavi tavi  0 2008-03-27 12:34 ..
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 0 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 1 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 2 -> /dev/pts/4
lr-x------ 1 tavi tavi 64 2008-03-27 12:34 3 -> /proc/18312/fd
```

```
08048000-0804c000 r-xp 00000000 08:02 16875609   /bin/cat
0804c000-0804d000 rw-p 00003000 08:02 16875609   /bin/cat
0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
...
b7f46000-b7f49000 rw-p b7f46000 00:00 0
b7f59000-b7f5b000 rw-p b7f59000 00:00 0
b7f5b000-b7f77000 r-xp 00000000 08:02 11601524   /lib/ld-2.7.so
b7f77000-b7f79000 rw-p 0001b000 08:02 11601524   /lib/ld-2.7.so
bfa05000-bfa1a000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

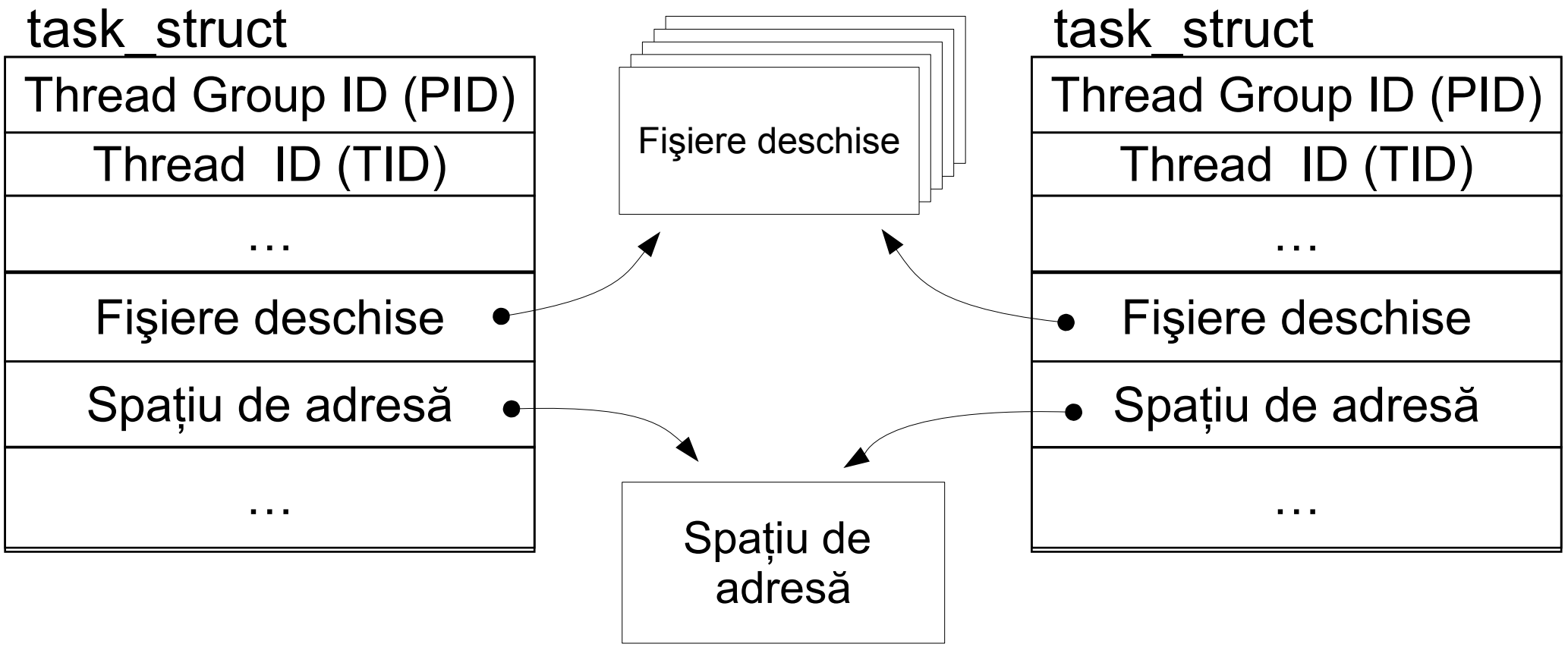
Fire de execuție

- Reprezintă un context de execuție
  - Regiştri, stivă
- Scheduler-ul planifică fire de execuție (nu procese)
- Un fir de execuție rulează în contextul unui proces (e.g. în spațiul de adresă al unui proces, are acces la fişierele deschise, etc.)

Implementare „clasică” fire de execuție (Windows)



Implementarea firelor de execuție în Linux



clone()

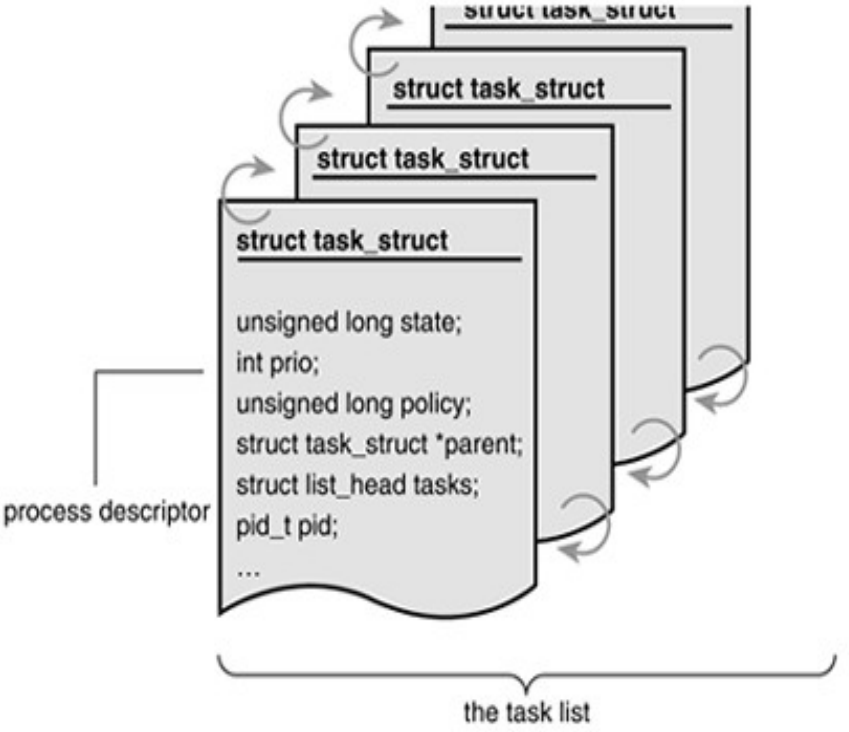
- În Linux diverse resurse ale unui proces pot fi fie partajate fie copiate în urma unui apel de sistem clone
  - CLONE_FILES – se partajează **tabela** descriptorilor de fişier
  - CLONE_VM – se partajează spațiul de adresă
  - CLONE_FS – se partajează informațiile despre sistemul de fişiere (directorul rădăcină, directorul curent, etc.)
  - CLONE_NEWNS – „mount namespace”-ul noului proces **nu** este partajat
  - CLONE_NEWIPC – spațiul de nume pentru IPC-uri **nu** este partajat (procesele nu pot vedea decât IPC-urile din acelaşi namespace)
  - CLONE_NEWNET – spațiul de nume pentru rețea **nu** este partajat (interfețele, stiva de Ipv4/IPV6, tabelele de rutare, setările de rețea, etc. nu sunt vizibile decât proceselor din acelaşi namespace)
  - CLONE_NEWPID – spațiul de nume pentru procese **nu** este partajat
- Mai multe detalii: man 2 clone

Containere

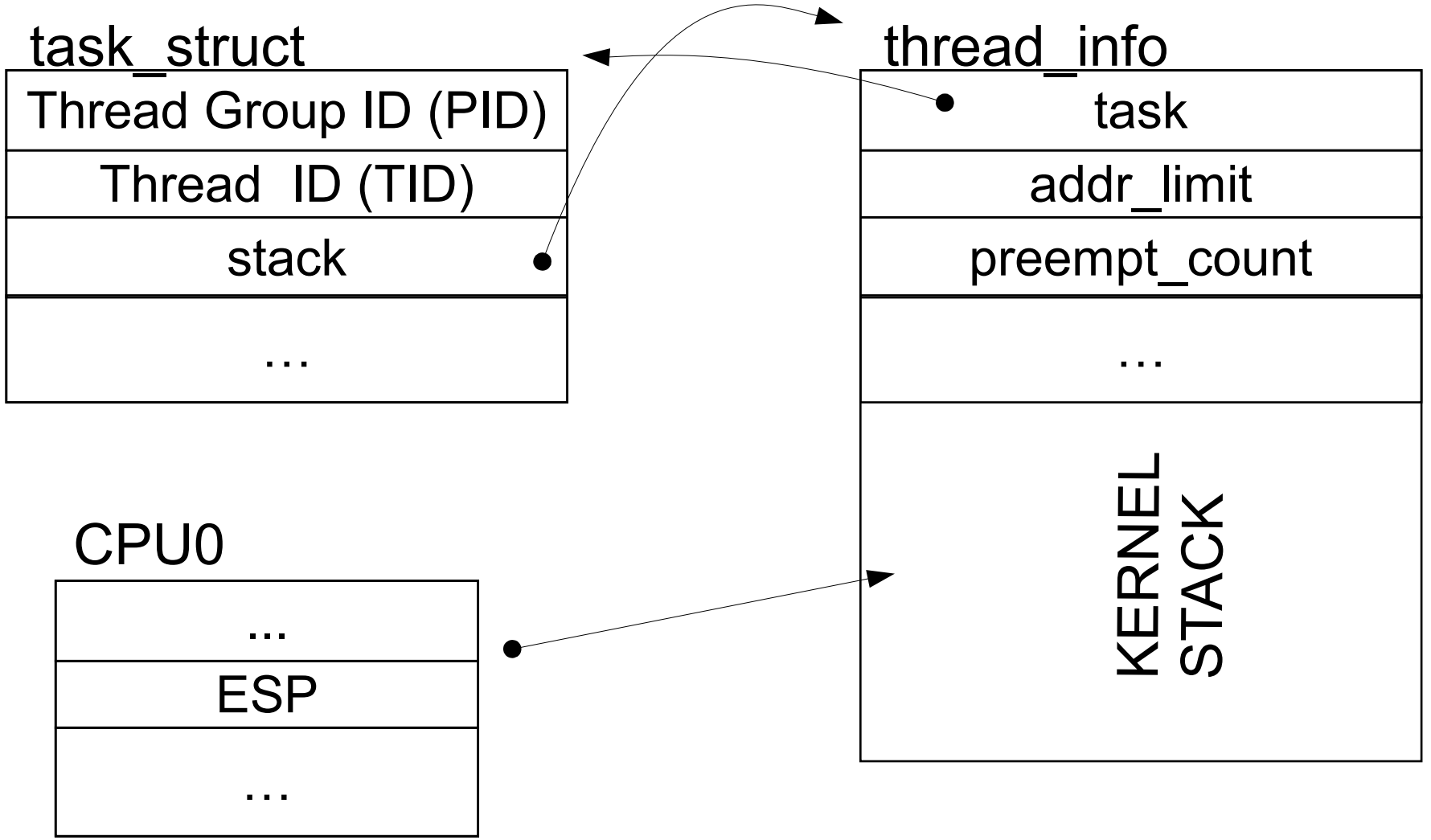
- O tehnologie de virtualizare la nivel de kernel
- Avantaje: mai light decât virtualizarea „clasică”
- Dezavantaje: nu e la fel de reliable, un bug în kernel va afecta toate containerele
- Sistemul de operare este partiționat, la nivel de kernel, în mai multe containere
- Containerele sunt izolate între ele (nu „se văd” decât procesele, interfețele de rețea, obiectele IPC, etc.)
- Izolarea / partiționarea se face prin încapsularea diverselor structuri ce descriu spațiul de nume al proceselor, interfețelor, obiectelor IPC într-o structură specială
  - Un procese este legat la o astfel de structură prin câmpul ns_proxy
- Mai multe detalii: http://lxc.sourceforge.net/lxc.html

task_struct

- Kernelul menține toate informațiile despre un proces în **task_struct**
- Această structură este independetă de arhitectură
- Câmpul **stack** conține un pointer la o structură **thread_info** ce conține informații dependente de arhitectură şi stiva kernel



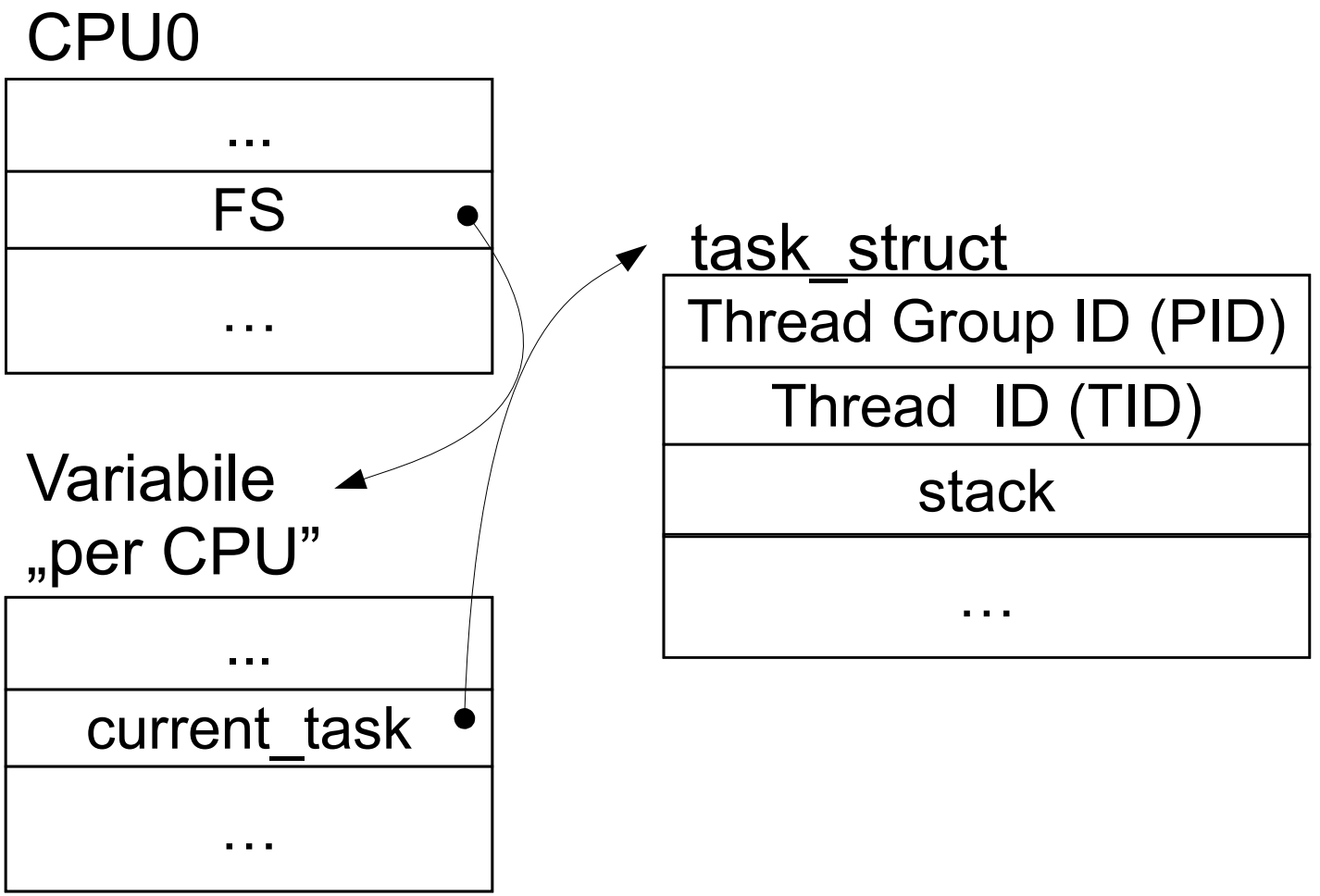
Legătura dintre thread_info şi task_struct (x86)



Acces la procesul current

- Accesul la structura ce descrie procesul curent este frecventă
- Exemple:
  - Deschiderea unui fişier are nevoie de accesul fişierele deschise (care se țin în PCB)
  - Maparea unui fişier în spațiul de adresă are nevoie de accesul la spațiul de adresă (care se ține în PCB)
- Peste 90% din apelurile de sistem au nevoie de acces la structura ce descrie procesul curent

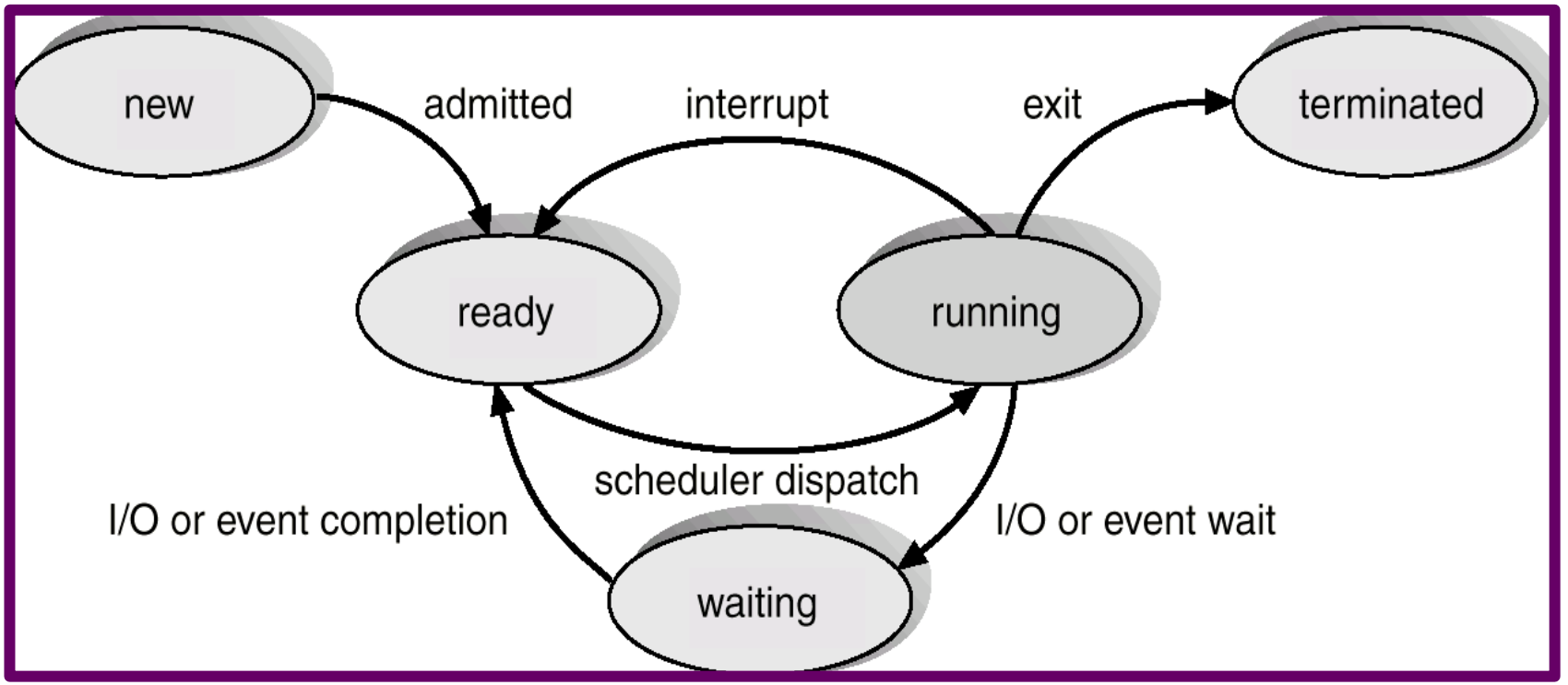
Accesul la procesul curent (x86, 32biți)



Alternativă (folosită în versiunile anterioare)

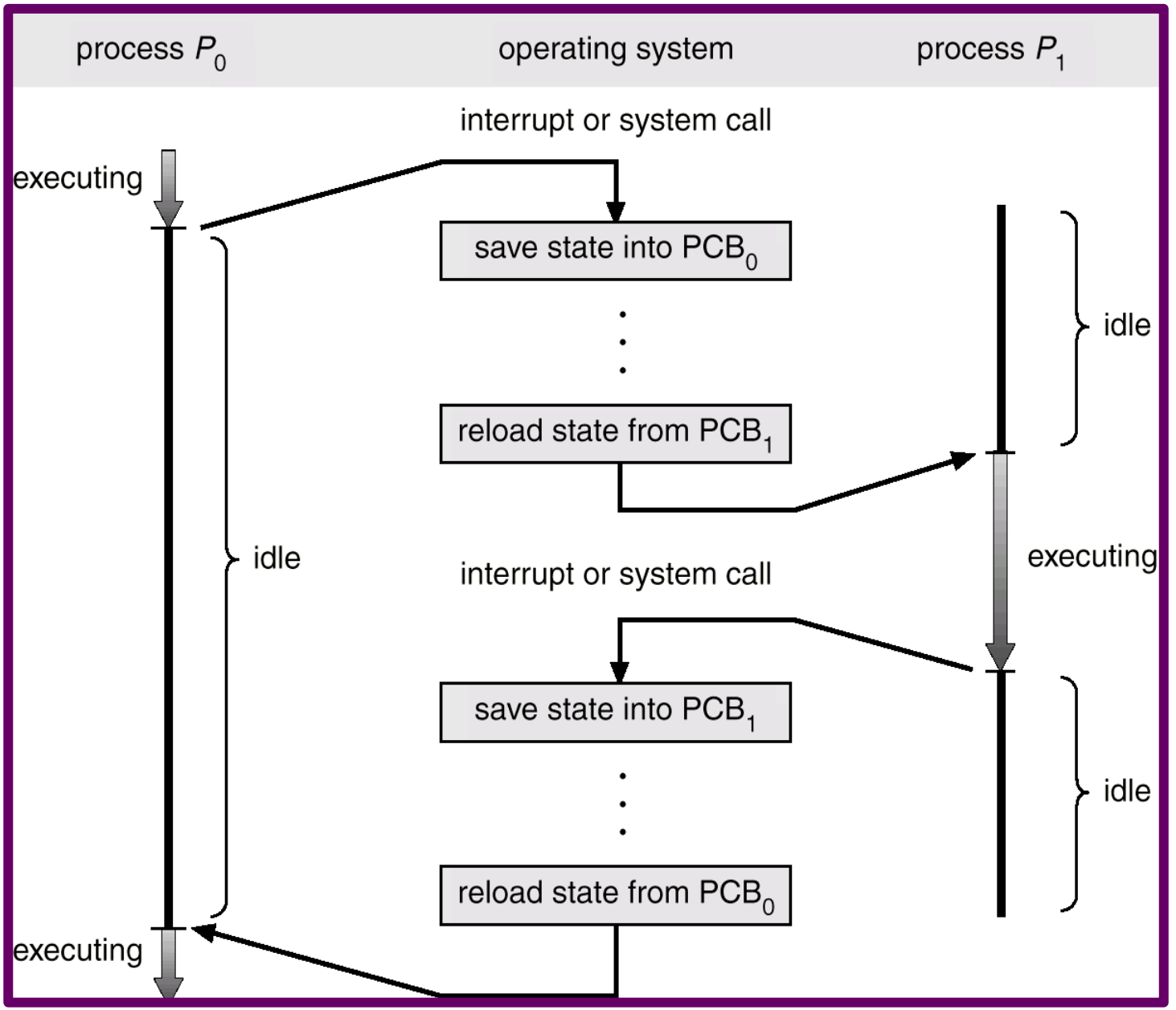
```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp")
  __attribute_used__;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
  return (struct thread_info *)(current_stack_pointer &
  ~(THREAD_SIZE - 1));
}

#define current current_thread_info()->task
```

Stări ale procesului



- READY: gata de execuție
- RUNNING: rulează
- WAITING: așteaptă terminarea unui operații de I/E

Schimbare de context



context_switch

```
static inline void
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
 {
     struct mm_struct *mm = next->mm, *oldmm = prev->active_mm;

     if (likely(!mm)) {
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next);
     } else
         switch_mm(oldmm, mm, next);

     if (likely(!prev->mm)) {
         prev->active_mm = NULL;
         rq->prev_mm = oldmm;
     }

     /* Here we switch the register state and the stack. */
     switch_to(prev, next, prev);

 }
```

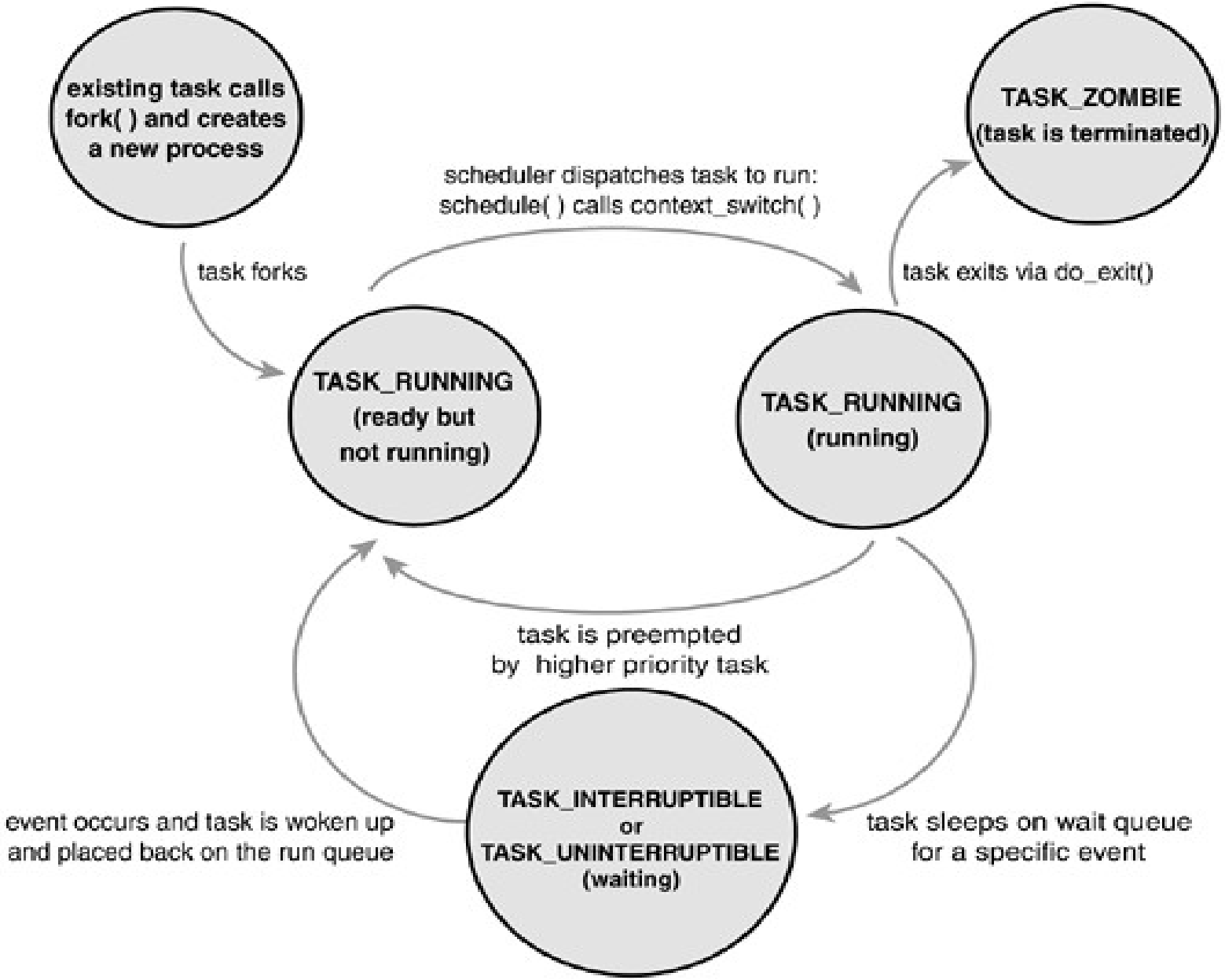
switch_to (x86)

```
unsigned long ebx, ecx, edx, esi, edi;
asm volatile(
          "pushfl\n\t"                          /* Save flags */
          "pushl %%ebp\n\t"                     /* Save EBP */
          "movl %%esp,%[prev_sp]\n\t"           /* save ESP */
          "movl %[next_sp],%%esp\n\t"           /* restore ESP */
          "movl $1f,%[prev_ip]\n\t"             /* save EIP */
          "pushl %[next_ip]\n\t"                /* restore EIP */
          "jmp __switch_to\n"                   /* regparm call */
          "1:\t"
          "popl %%ebp\n\t"                      /* restore EBP */
          "popfl"                               /* restore flags */
    /* output parameters */
    :[prev_sp] "=m" (prev->thread.esp),
     [prev_ip] "=m" (prev->thread.eip), "=a" (last),

    /* clobbered output registers: */
    "=b" (ebx), "=c" (ecx), "=d" (edx),
    "=S" (esi), "=D" (edi)

    /* input parameters */
    :[next_sp] "m" (next->thread.esp),
     [next_ip] "m" (next->thread.eip),
    /* regparm parameters for __switch_to */
     [prev] "a" (prev), [next] "d" (next)
);
```

Schimbare de context (2)

- Salvarea stării procesului curent
  - Majoritatea informațiilor despre procesul curent sunt deja în task_struct, nu mai trebuie salvate
  - Salvarea contextului de execuție: regiştrii kernel
- Încărcarea stării procesului de rulat
  - Încărcarea contextului de execuție pentru noul proces (setup MMU dacă se schimbă spațiul de adresă)
  - Setarea noului proces curent
- Regiştrii userspace sunt salvați pe stiva kernel şi vor fi restaurați la întoarcerea în userspace
- În __switch_to se salvează/încarcă restul contextului (fs, gs, FPU, se seteaza noul proces curent, etc.)

Diagrama de stări a proceselor în Linux



Blocarea threadului curent

- Se schimbă starea threadului în WAITING
- Se şterge threadul din coada READY
- Se pune threadul într-o coadă de aşteptare (se inserează thread-ul într-o listă)
- Se apelează scheduler-ul, care caută şi selectează un nou thread din coada READY
- Se face schimbarea de context către noul thread

Blocarea threadului curent - pseudocod

```
sleep_on_common(wait_queue_head_t *q, int state, long timeout)
{
  unsigned long flags;
  wait_queue_t wait;
  init_waitqueue_entry(&wait, current);

  __set_current_state(state);
  spin_lock_irqsave(&q->lock, flags);
  __add_wait_queue(q, &wait);
  spin_unlock(&q->lock);
  timeout = schedule_timeout(timeout);
  spin_lock_irq(&q->lock);
  __remove_wait_queue(q, &wait);
  spin_unlock_irqrestore(&q->lock, flags);

  return timeout;
}
```

Blocarea threadului curent – pseudocod (2)

```
static inline void __add_wait_queue(wait_queue_head_t *head,
  wait_queue_t *new)
{
  list_add(&new->task_list, &head->task_list);
}

static inline void __remove_wait_queue(wait_queue_head_t *head,
  wait_queue_t *old)
{
  list_del(&old->task_list);
}

void __sched sleep_on(wait_queue_head_t *q)
{
 sleep_on_common(q, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

Trezirea unui thread blocat

- Trezirea se face indirect, prin semnalizarea cozii de aşteptare
- Se selectează un thread din coada de aşteptare
- Se setează starea threadului în READY
- Se introduce threadul în coada READY

Kernel nepreemptiv

- La fiecare tick de ceas se verifică dacă procesul curent şi-a depăşit cuanta de timp
- În caz afirmativ se setează un flag ce indică acest lucru
- Înainte de întoarcerea în user-space se verifică acest flag şi în cazul în care este setat se apelează schedulerul
- Kernelul nu trebuie preemptat, nu există probleme de sincronizare pe maşini uniprocesor

Kernel preemptiv

- Procesul curent poate fi preemptat chiar şi atunci când rulează în kernel
- Trebuie folosite primitive de sincronizare
- În zonele critice trebuie dezactivată preempția:
  - Atunci când se ia un spinlock, când se dezactivează bottom-half handlerele sau întreruperile, când se intră într-o regiune RCU
  - Se foloseşte un contor de preempție; la ieşierea din zona critică se decrementează contorul
- Dacă apare o condiție ce necesită preempție se setează un flag în cadrul procesului curent
- Preempția se face doar atunci când contorul de preempție ajunge la zero şi când flag-ul este setat; condiția de preemptivitate este verificată:
  - La întoarcerea dintr-o întrerupere
  - Atunci când contorul de preempție ajunge la zero
  - (Atunci când procesul curent se blochează)

Context proces

- În kernel, rulăm în context process atunci când:
  - se face un apel de sistem
  - se trezeşte un thread blocat
- În context proces există un proces curent bine determinat
- În context proces putem face sleep (e.g. să comutăm contextul către alt proces)
- În context proces putem accesa spațiul utilizator (e.g. copy_from_user)

Kernel threads

- Anumite operații pe care kernelul trebuie să le facă pot fi blocante (e.g. swap-in, swap-out, etc.) şi deci trebuie executate în context proces
- Kernel thread-urile nu au un spațiu de adresă utilizator
- Kernel thread-urile aparțin unui proces special (init)

Planificatorul de procese

- Completely Fair Scheduler
- Introdus in 2.6.23 împreună cu o abordare modulară pentru planificare bazată pe clase de scheduling
- Înlocuieşte planificatorul precedent, introdus în 2.6.0 (cunoscut sub numele de „O(1) scheduler”)
  - Principalul dezavantaj al planificatorului precedent este faptul că foloseşte o abordare euristică pentru a prezice modul în care se comportă un proces
- Bazat pe Rotary Staircase Deadline Scheduler
  - Se renunța la estimare, criteriul de planificare fiind bazat pe conceptul de fairness

Completely Fair Scheduler

- Arbore sortat după timpul de aşteptare (wait_runtime)
- Se menține un timp virtual (fair_clock), care ia în considerare numărul de task-uri din sistem
  - Pentru 4 task-uri, fair_clock „merge” de patru ori mai încet decât timpul real
- În timp ce un task rulează wait_runtime-ul este decrementat
- Algoritmul selectează un task astfel încât să obțină wait_runtime cât mai apropiat de zero pentru fiecare task
- Prioritatea task-urilor este luată în considerare prin alterarea vitezei de scurgere a timpului pentru task-ul respectiv (pentru un task mai prioritar timpul se scurge mai încet)
- Mai multe detalii:
  - http://www.ibm.com/developerworks/linux/library/l-cfs/
  - http://kerneltrap.org/node/8059

Intrebări

?