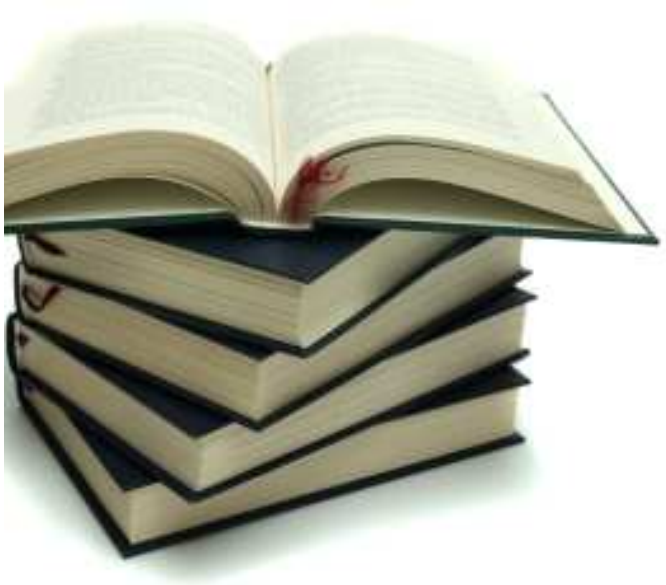
SO2
kernel crunch

Cursul 5



5

Procese

26 martie 2009

Test Curs 4

- Ce apeluri de sistem nu se pot implementa in userspace, folosind mecanismul VDSO?
- De ce în Windows nucleul are nevoie şi de numărul de parametri pentru un apel de sistem?
- Fie AU un pointer în userspace şi AK un pointer în kernel. Ce se întâmpla în cazul următoarelor secvențe de cod:
  - memcpy(AK, AU, len)
  - copy_from_user(AK, AU, len)

dacă adresa AU este validă, respectiv invalidă?

Procese

- Implementarea proceselor şi threadurilor
- Schimbare de context
- Blocarea şi trezirea threadurilor
- Procese şi threaduri în Linux
- Procese şi threaduri în Windows

Bibliografie

- UTLK: capitolul 3
- LKD: capitolul 3,4
- WI: capitolul 6

Procese

- 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

„Conținutul” unui proces

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

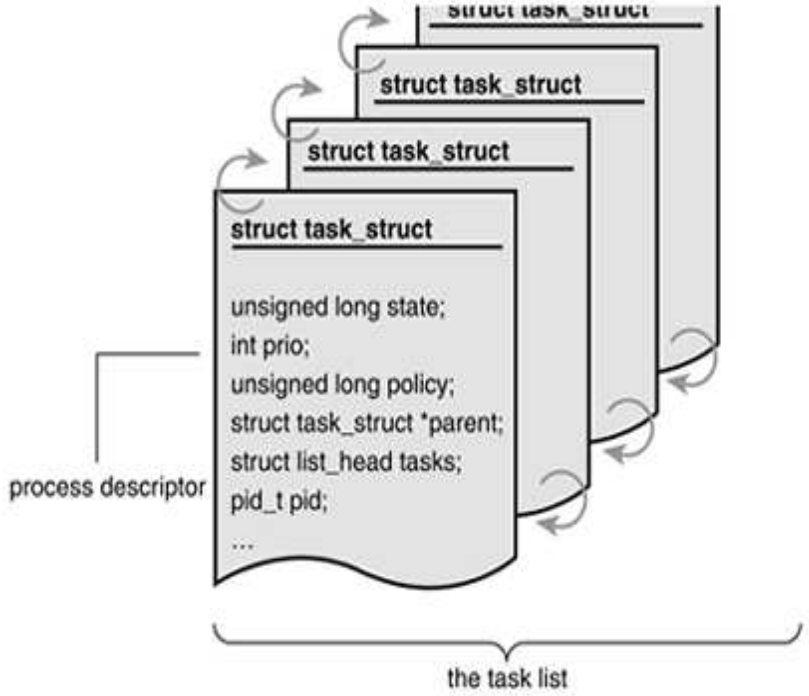
```
Name:   cat
State:  R (running)
Tgid:   18205
Pid:    18205
PPid:   18133
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
```

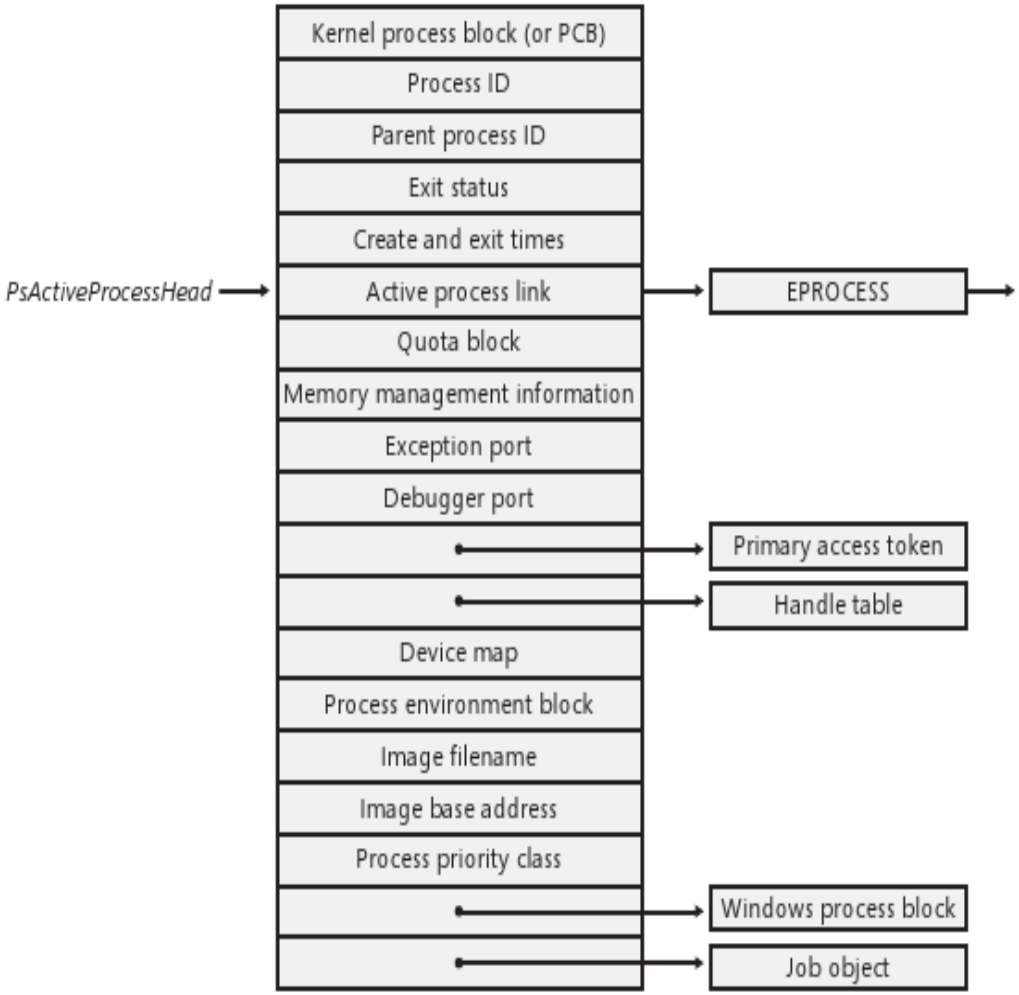
```
dr-x------ 2 tavi tavi  0 2008-03-27 12:34 .
dr-xr-xr-x 6 tavi tavi  0 2008-03-27 12:34 ..
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 0 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 1 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2008-03-27 12:34 2 -> /dev/pts/4
lr-x------ 1 tavi tavi 64 2008-03-27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

Process Control Block (PCB)

- Kernelul menține toate informațiile despre un proces într-o structură

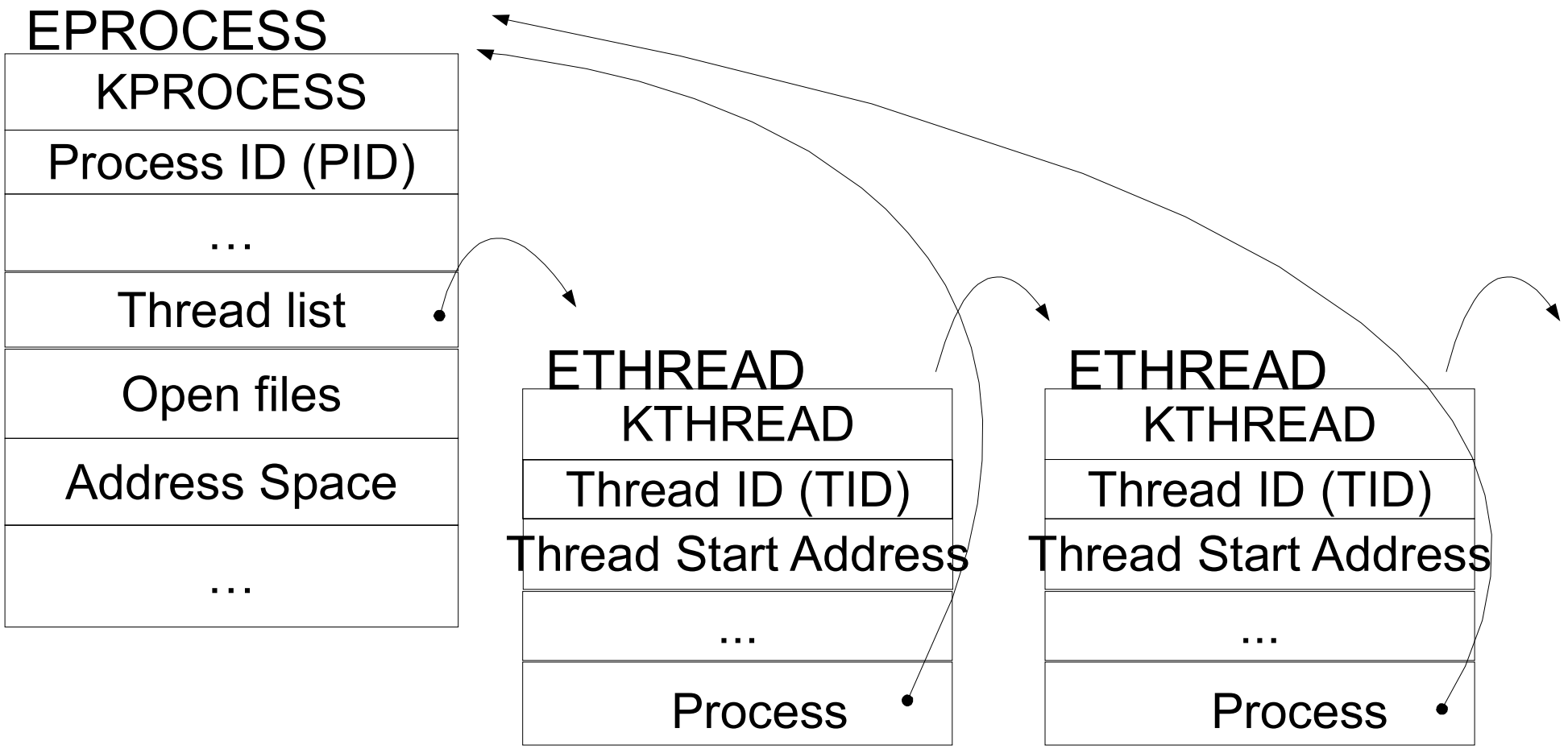




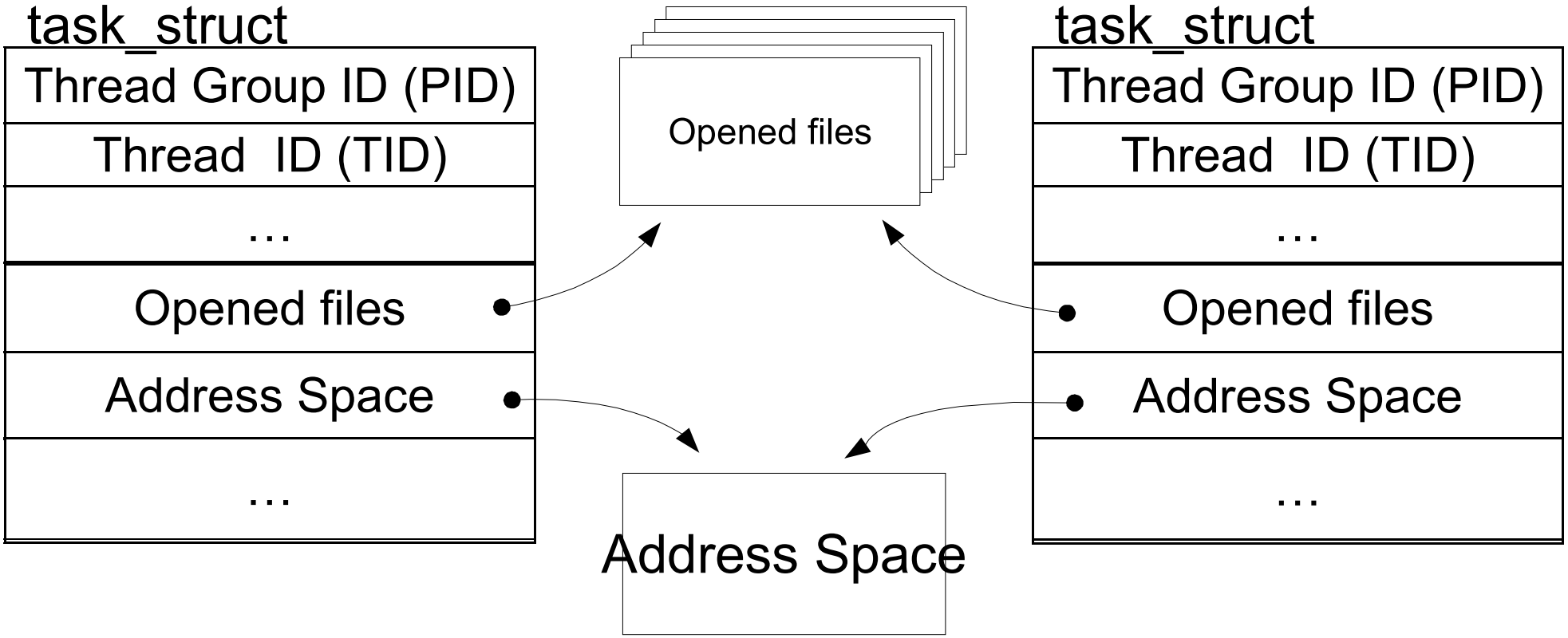
Fire de execuție

- 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.)

Fire de execuție în Windows

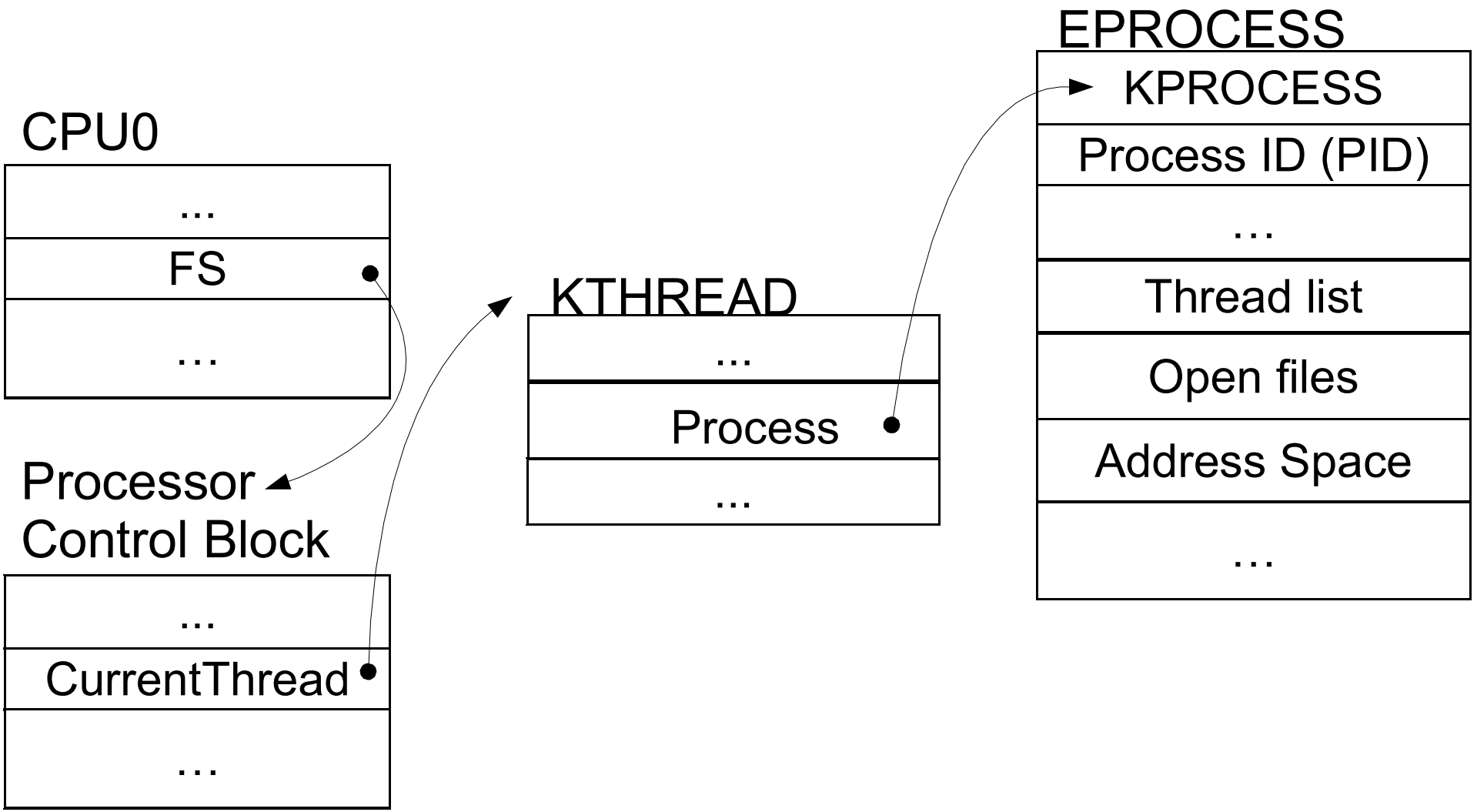


Fire de execuție în Linux

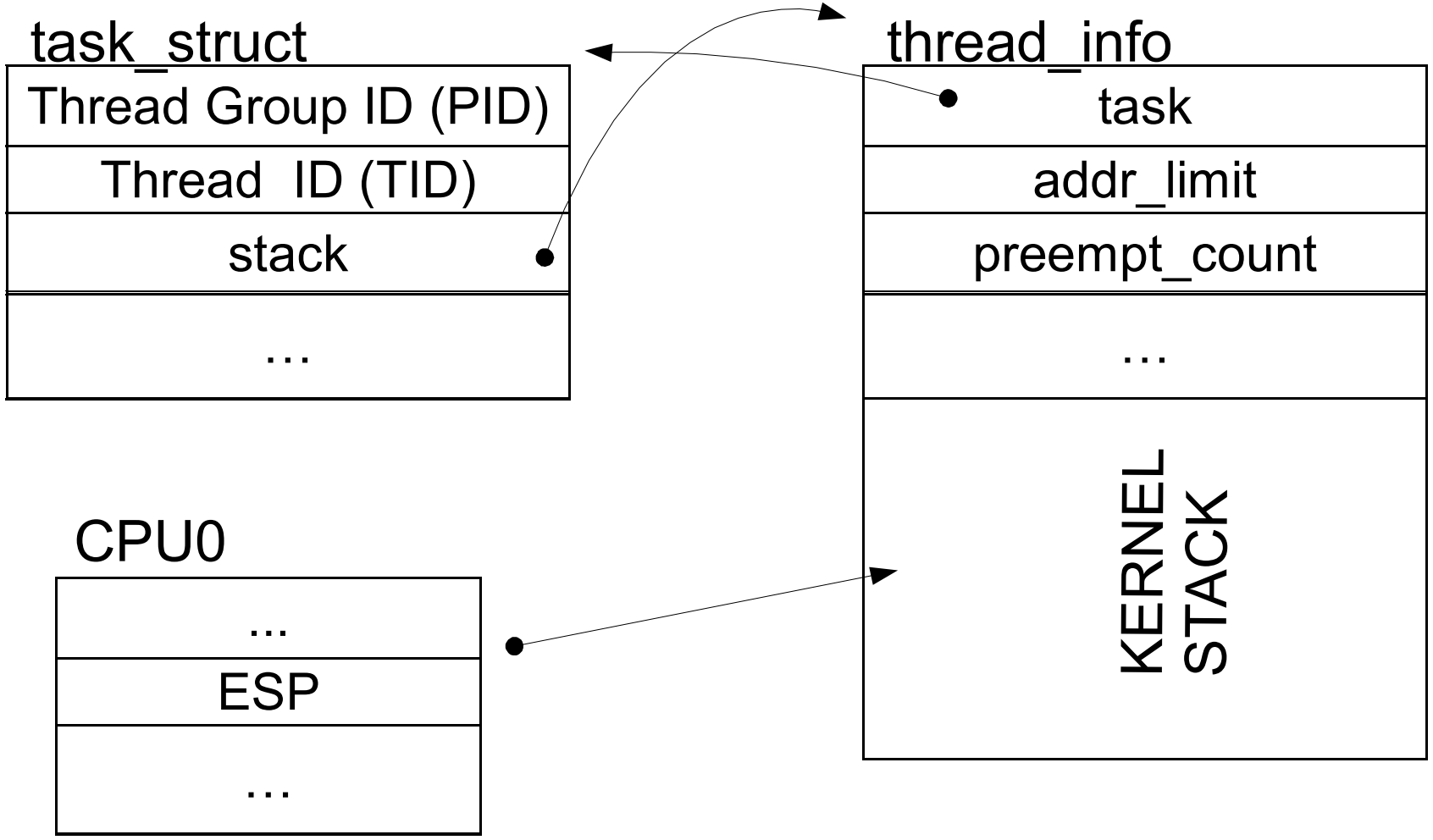


Acces la procesul current

- 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 fişiere 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


EPROCESS
KPROCESS
Process ID (PID)
…
Thread list
Open files
Address Space
…
CPU0
…
FS
…
KTHREAD
…
Process
…
Processor
Control Block
…
CurrentThread
…

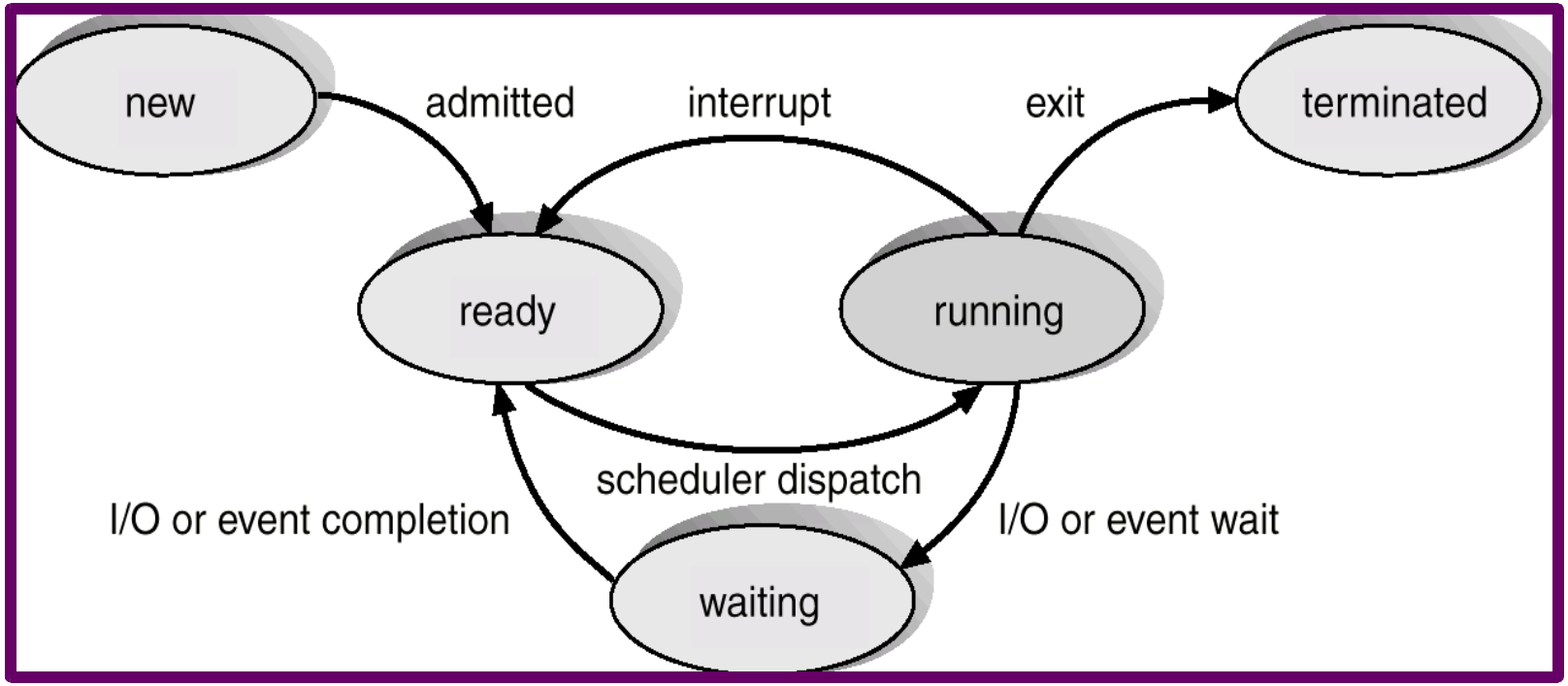
Linux x86 – the (not so) old way



Linux x86 – the (not so) old way (2)

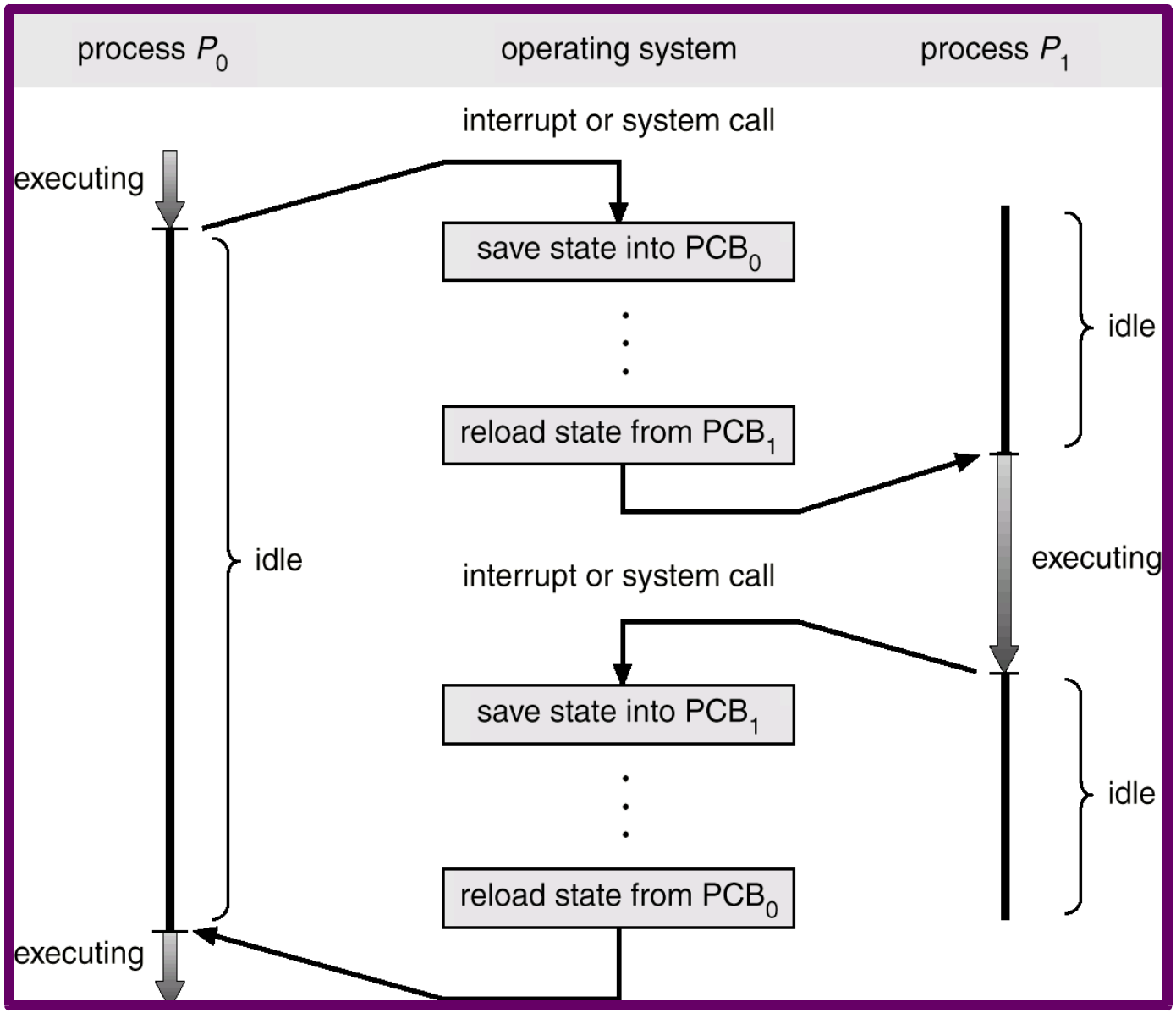
```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp")
  __attribute_used__;

/* 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: asteaptă terminarea unui operații de I/E

Schimbare de context



Schimbare de context (2)

- Salvarea stării procesului curent
  - Majoritatea informațiilor despre procesul curent sunt deja în PCB, nu mai trebuie salvate
  - Salvarea contextului de execuție (regiştri user/kernel space)
- Încărcarea stării procesului de rulat
  - Încărcarea contextului de execuție pentru noul proces (regiştri user/kernel space, setup MMU dacă se schimbă spațiul de adresă)
  - Setarea noului proces curent

switch_to (x86)

```
#define switch_to(prev,next,last) do {
unsigned long esi,edi;
asm volatile("pushfl\n\t"                    /* Save flags */
             "pushl %%ebp\n\t"
             "movl %%esp,%0\n\t"             /* save ESP */
             "movl %5,%%esp\n\t"             /* restore ESP */
             "movl $1f,%1\n\t"               /* save EIP */
             "pushl %6\n\t"                  /* restore EIP */
             "jmp __switch_to\n"
             "1:\t"
             "popl %%ebp\n\t"
             "popfl"
    :"=m" (prev->thread.esp),"=m" (prev->thread.eip),
    "=a" (last),"=S" (esi),"=D" (edi)
    :"m" (next->thread.esp),"m" (next->thread.eip),
    "2" (prev), "d" (next));
```

Schimbare de context în Linux/x86

- Regiştrii userspace sunt salvați pe stiva kernel şi vor fi restaurați la întoarcerea în userspace
- Starea procesului e deja salvată (în task_struct)
- În __switch_to se salvează/încarcă restul contextului (fs, gs, FPU, etc.)
- Schimbarea stivei (ESP) va seta indirect şi noul proces curent
- Comutarea contextului MMU se face anterior, în context_switch()

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 threadul 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

Exemplu: Implementarea în Linux

```
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;
}
```

Exemplu: Implementarea în Linux (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

Preemptivitate procese pentru kernele nepreemptive

- 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

Preemptivitate kernel

- Procesul curent poate fi preemptat chiar şi atunci când rulează în kernel
- Trebuiesc folosite primitive de sincronizare
- În zonele critice trebuie dezactivată preempția:
  - Atunci când se i-a un spinlock, atunci când se dezactivează bottom-half handlerele, sau întreruperile
  - Se foloseşte un conter de preempție
- Dacă apare o condiție ce necesită preempție se setează un flag în cadrul procesului curent

Preemptivitate kernel (2)

- Preempția se face doar atunci când contorul de preempție ajunge la zero
- „Puncte de preempție”:
  - La întoarcerea dintr-o întrerupere
  - Atunci când contorul de preempție ajunge la zero
  - (Atunci când procesul curent se blochează)
- -> Preempția se face imediat ce apare o condiție şi se iese dintr-o zonă critică

Diagrama de stări a proceselor în Linux

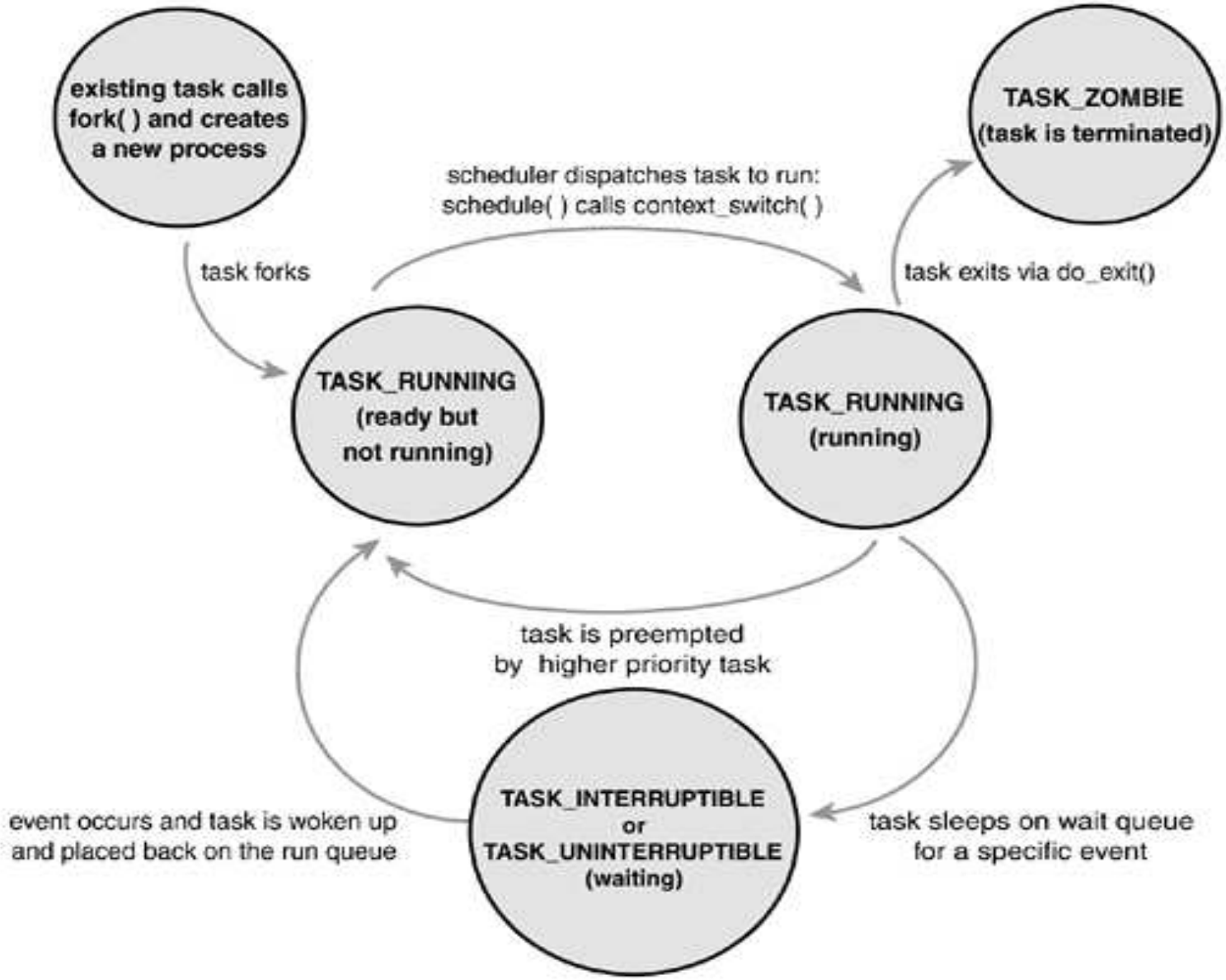
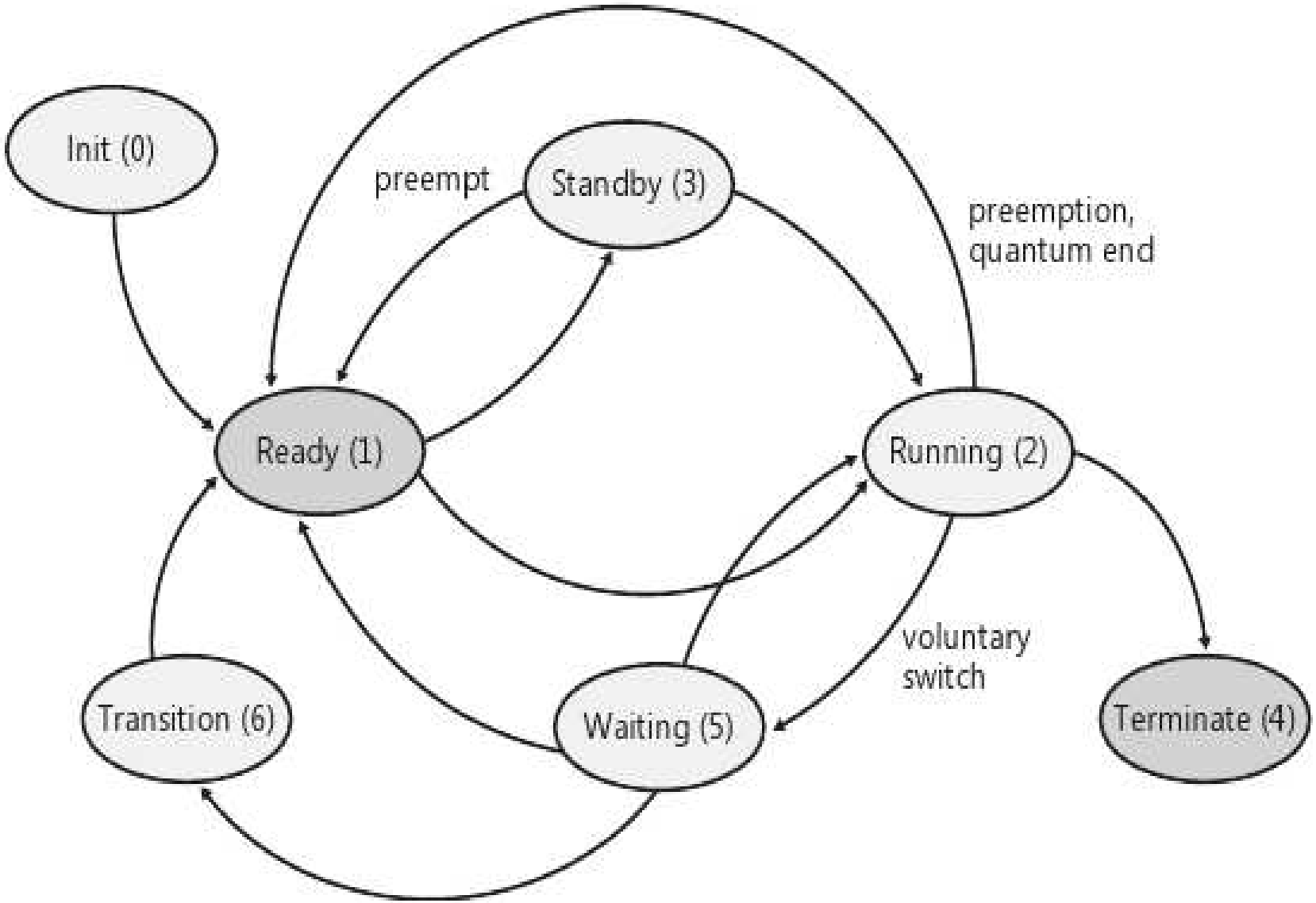


Diagrama de stări a proceselor în Windows



Context proces

- „Se intră” în contextul unui proces atunci când
  - se face un apel de sistem
  - se trezeşte un thread blocat
- În context proces există un proces curent
- 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 (swap-in, swap-out, etc.)
- -> Trebuiesc executate în context proces
- Kernel thread-urile nu au un spațiu de adresă user
- Kernel thread-urile aparțin de un proces special

Intrebări

?