Arquitectura de Sistemas

Procesos

Gustavo Romero López

Updated: 14 de febrero de 2019

Arquitectura y Tecnología de Computadores

Índice

Lecturas recomendadas

Jean Bacon	Operating Systems (4)
Abraham Silberschatz	Fundamentos de Sistemas Operativos
William Stallings	Sistemas Operativos (3)
Andrew Tanuenbaum	Sistemas Operativos Modernos (2.1)

Definición

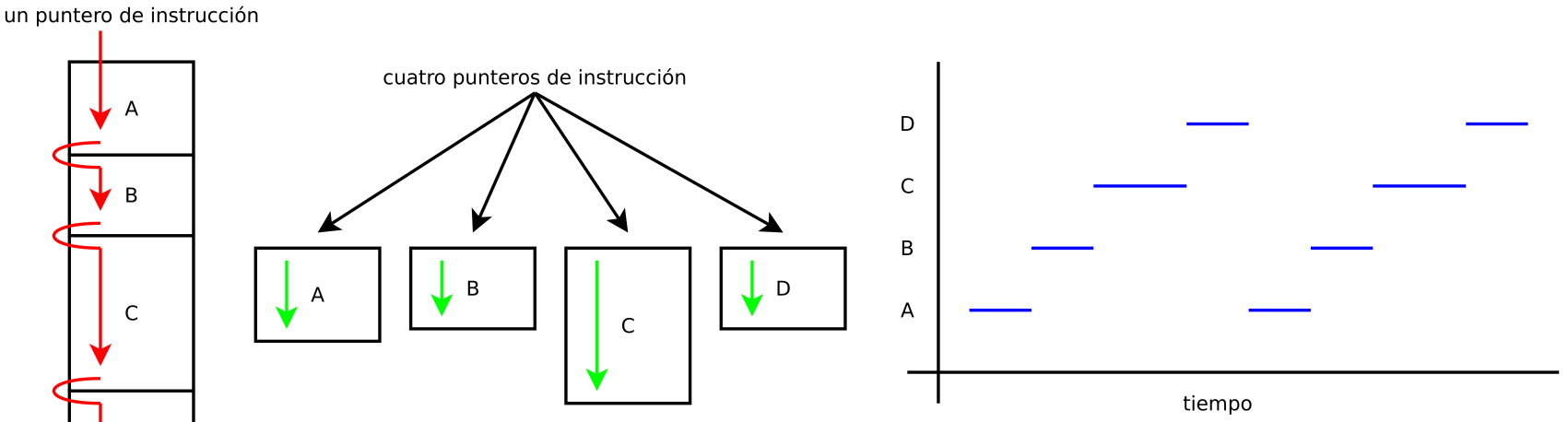
- ¿Qué es un **proceso**?
  - **programa** en ejecución.
  - entorno de **protección**.
  - algo **dinámico**.
- Componentes básicos:
  - **hebras**/hilos de ejecución.
  - **espacio de direcciones**.
- La tarea fundamental de un SO es la gestión de procesos:
  - **creación**.
  - **planificación**.
  - **comunicación** (sincronización).
  - **finalización**.
- Un **programa** es...
  - una lista de **instrucciones** (especie de receta de cocina).
  - algo **estático**.
  - **varios** procesos pueden ejecutar un mismo programa.
- Puede **lanzar**, o ser lanzado por, otros procesos.

Ejemplo: gcc

- **gcc** es un **programa**.
- Muchos usuarios pueden utilizarlo **simultáneamente**.
- Una única copia del programa es **compartida** por todos.
- Los procesos que ejecutan gcc para cada usuarios son **indenpendientes**.
  - un fallo en uno no afecta a los demás.
- gcc para cumplir con sus funciones lanza otra serie de procesos:
  - `cpp`: preprocesador.
  - `as`: ensamblador.
  - `cc`: compilador.
  - `ld`: enlazador.

Modelo de procesamiento

- Todo el software se organiza en forma de procesos.
- proceso = programa + entorno (procesador + memoria).
- Objetivos:
  - **multiprogramación**: maximizar el uso del procesador $\Longrightarrow$ ocupar procesador continuamente $\Longrightarrow$ maximizar rendimiento.
  - **tiempo compartido**: interacción como si cada proceso dispusiera del sistema por completo $\Longrightarrow$ cambiar entre ellos frecuentemente $\Longrightarrow$ cambiar entre ellos en los momentos precisos.



Modelo de procesamiento

- Clasificación en función del coste del cambio de proceso:
  - procesamiento **pesado**: proceso UNIX.
    - hebra de actividad y espacio de direcciones unificados.
    - el cambio de proceso implica 2 cambios de espacio de direcciones.

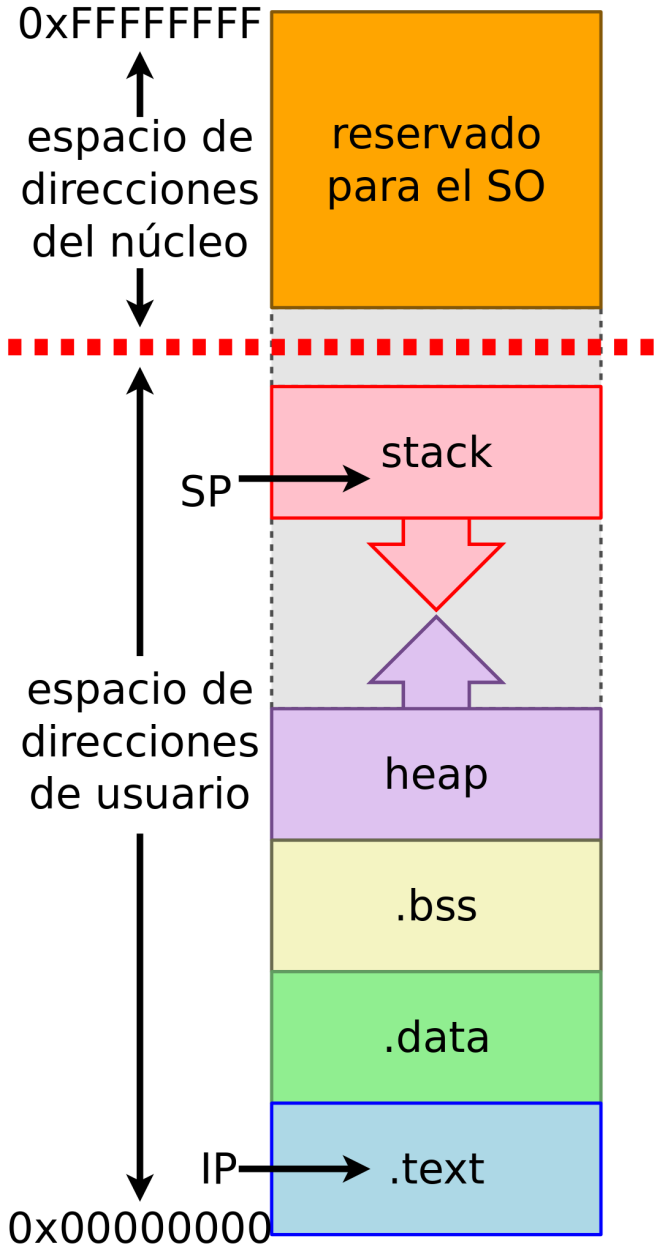
      $\mathrm{ED}_x \longrightarrow \mathrm{ED}_{SO} \longrightarrow \mathrm{ED}_y$
  - procesamiento **ligero**: hebras tipo núcleo.
    - hebra de actividad y espacio de direcciones desacoplados.
    - el cambio de hebra implica 1 ó 2 cambios de espacio de direcciones en función de si las hebras lo comparten o no.

      $\mathrm{ED}_x \longrightarrow \mathrm{ED}_{SO} \longrightarrow \mathrm{ED}_y$ / ED_y
  - procesamiento **superligero/pluma**: hebras tipo usuario.
    - hebra de actividad y espacio de direcciones unificados.
    - el cambio de hebra no implica ningún cambio de espacio de direcciones.

      $\mathrm{ED}_x \longrightarrow \mathrm{ED}_y$

Ejemplo: Espacio de direcciones en Linux

- Espacio de direcciones:
  - código: .text
  - datos:
    - inicializados: .data
    - sin inicializar: .bss
    - dinámicos: heap
  - pila: stack
- Linux se mapea dentro del espacio de direcciones de cada proceso para **ahorrar** cambios de proceso.
- Divisiones típicas (32 bits): 2GB/2GB y 3GB/1GB.



Estructuras de control del sistema operativo

- Para **gestionar** procesos y recursos el SO debe disponer de **información** sobre estos.
- El SO mantiene **tablas** sobre cada entidad que gestiona:
  - Tablas de **memoria**: principal y secundaria, protección, tradución.
  - Tablas de **E/S**: dispositivos y canales, estado de las operaciones.
  - Tablas de **ficheros**: existencia, atributos, localización,...
  - Tablas de **procesos**: localización y atributos.
- Las tablas anteriores no suelen estar separadas sino entrelazadas.
- Normalmente las tablas se inicializan al arrancar el sistema mediante autoconfiguración.

Estructuras de control de procesos

- Representación física de un proceso:
  - **Imagen del proceso**:
    - programa a ejecutar.
    - espacio de direcciones disponible para código, datos y pila.
  - **Bloque de Control del Proceso (PCB)** o **descriptor de proceso**:
    - atributos para la gestión del proceso por parte del SO.
    - **estructura de datos más importante del SO.**
- Atributos de un proceso:
  - **Identificación** del proceso: identificadores del proceso, proceso padre, usuario.
  - Estado del **procesador**: registros de propósito general, de estado y control, puntero de pila.
  - Información de **control** del proceso: estado, planificación, estructuración, comunicación y sincronización, privilegios, gestión de memoria, control de recursos y utilización.
- Ejemplo: Linux, `struct task_struct` en `sched.h`.

Linux task_struct from sched.h

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
  /*
   * For reasons of header soup (see current_thread_info()), this
   * must be the first element of task_struct.
   */
  struct thread_info thread_info;
#endif
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
  atomic_t usage;
  unsigned int flags; /* per process flags, defined below */
  unsigned int ptrace;

#ifdef CONFIG_SMP
  struct llist_node wake_entry;
  int on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
  unsigned int cpu; /* current CPU */
#endif
  unsigned int wakee_flips;
  unsigned long wakee_flip_decay_ts;
  struct task_struct *last_wakee;

  int wake_cpu;
#endif
  int on_rq;

  int prio, static_prio, normal_prio;
  unsigned int rt_priority;
  const struct sched_class *sched_class;
  struct sched_entity se;
  struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
  struct task_group *sched_task_group;
#endif
```

Linux task_struct from sched.h II

```
  struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
  /* list of struct preempt_notifier: */
  struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
  unsigned int btrace_seq;
#endif

  unsigned int policy;
  int nr_cpus_allowed;
  cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
  int rcu_read_lock_nesting;
  union rcu_special rcu_read_unlock_special;
  struct list_head rcu_node_entry;
  struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
  unsigned long rcu_tasks_nvcsw;
  bool rcu_tasks_holdout;
  struct list_head rcu_tasks_holdout_list;
  int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */

#ifdef CONFIG_SCHED_INFO
  struct sched_info sched_info;
#endif

  struct list_head tasks;
#ifdef CONFIG_SMP
  struct plist_node pushable_tasks;
  struct rb_node pushable_dl_tasks;
```

Linux task_struct from sched.h III

```
#endif

  struct mm_struct *mm, *active_mm;
  /* per-thread vma caching */
  u32 vmacache_seqnum;
  struct vm_area_struct *vmacache[VMACACHE_SIZE];
#if defined(SPLIT_RSS_COUNTING)
  struct task_rss_stat  rss_stat;
#endif
/* task state */
  int exit_state;
  int exit_code, exit_signal;
  int pdeath_signal;  /*  The signal sent when the parent dies  */
  unsigned long jobctl; /* JOBCTL_*, siglock protected */

  /* Used for emulating ABI behavior of previous Linux versions */
  unsigned int personality;

  /* scheduler bits, serialized by scheduler locks */
  unsigned sched_reset_on_fork:1;
  unsigned sched_contributes_to_load:1;
  unsigned sched_migrated:1;
  unsigned sched_remote_wakeup:1;
  unsigned :0; /* force alignment to the next boundary */

  /* unserialized, strictly 'current' */
  unsigned in_execve:1; /* bit to tell LSMs we're in execve */
  unsigned in_iowait:1;
#if !defined(TIF_RESTORE_SIGMASK)
  unsigned restore_sigmask:1;
#endif
#ifdef CONFIG_MEMCG
  unsigned memcg_may_oom:1;
#ifndef CONFIG_SLOB
  unsigned memcg_kmem_skip_account:1;
#endif
```

Linux task_struct from sched.h IV

```
#endif
#ifdef CONFIG_COMPAT_BRK
  unsigned brk_randomized:1;
#endif

  unsigned long atomic_flags; /* Flags needing atomic access. */

  struct restart_block restart_block;

  pid_t pid;
  pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
  /* Canary value for the -fstack-protector gcc feature */
  unsigned long stack_canary;
#endif
  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->real_parent->pid)
   */
  struct task_struct __rcu *real_parent; /* real parent process */
  struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
  /*
   * children/sibling forms the list of my natural children
   */
  struct list_head children;  /* list of my children */
  struct list_head sibling; /* linkage in my parent's children list */
  struct task_struct *group_leader; /* threadgroup leader */

  /*
   * ptraced is the list of tasks this task is using ptrace on.
   * This includes both natural children and PTRACE_ATTACH targets.
   * p->ptrace_entry is p's link on the p->parent->ptraced list.
   */
  struct list_head ptraced;
```

Linux task_struct from sched.h V

```
  struct list_head ptrace_entry;

  /* PID/PID hash table linkage. */
  struct pid_link pids[PIDTYPE_MAX];
  struct list_head thread_group;
  struct list_head thread_node;

  struct completion *vfork_done;    /* for vfork() */
  int __user *set_child_tid;    /* CLONE_CHILD_SETTID */
  int __user *clear_child_tid;    /* CLONE_CHILD_CLEARTID */

  cputime_t utime, stime, utimescaled, stimescaled;
  cputime_t gtime;
  struct prev_cputime prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
  seqcount_t vtime_seqcount;
  unsigned long long vtime_snap;
  enum {
    /* Task is sleeping or running in a CPU with VTIME inactive */
    VTIME_INACTIVE = 0,
    /* Task runs in userspace in a CPU with VTIME active */
    VTIME_USER,
    /* Task runs in kernelspace in a CPU with VTIME active */
    VTIME_SYS,
  } vtime_snap_whence;
#endif

#ifdef CONFIG_NO_HZ_FULL
  atomic_t tick_dep_mask;
#endif
  unsigned long nvcsw, nivcsw; /* context switch counts */
  u64 start_time;   /* monotonic time in nsec */
  u64 real_start_time;  /* boot based time in nsec */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
  unsigned long min_flt, maj_flt;
```

Linux task_struct from sched.h VI

```
  struct task_cputime cputime_expires;
  struct list_head cpu_timers[3];

/* process credentials */
  const struct cred __rcu *ptracer_cred; /* Tracer's credentials at attach */
  const struct cred __rcu *real_cred; /* objective and real subjective task
           * credentials (COW) */
  const struct cred __rcu *cred;  /* effective (overridable) subjective task
           * credentials (COW) */
  char comm[TASK_COMM_LEN]; /* executable name excluding path
              - access with [gs]et_task_comm (which lock
                it with task_lock())
              - initialized normally by setup_new_exec */
/* file system info */
  struct nameidata *nameidata;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
  struct sysv_sem sysvsem;
  struct sysv_shm sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
  unsigned long last_switch_count;
#endif
/* filesystem information */
  struct fs_struct *fs;
/* open file information */
  struct files_struct *files;
/* namespaces */
  struct nsproxy *nsproxy;
/* signal handlers */
  struct signal_struct *signal;
  struct sighand_struct *sighand;

  sigset_t blocked, real_blocked;
  sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
```

Linux task_struct from sched.h VII

```
  struct sigpending pending;

  unsigned long sas_ss_sp;
  size_t sas_ss_size;
  unsigned sas_ss_flags;

  struct callback_head *task_works;

  struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
  kuid_t loginuid;
  unsigned int sessionid;
#endif
  struct seccomp seccomp;

/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed,
 * mempolicy */
  spinlock_t alloc_lock;

  /* Protection of the PI data structures: */
  raw_spinlock_t pi_lock;

  struct wake_q_node wake_q;

#ifdef CONFIG_RT_MUTEXES
  /* PI waiters blocked on a rt_mutex held by this task */
  struct rb_root pi_waiters;
  struct rb_node *pi_waiters_leftmost;
  /* Deadlock detection and priority inheritance handling */
  struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
```

Linux task_struct from sched.h VIII

```
  /* mutex deadlock detection */
  struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
  unsigned int irq_events;
  unsigned long hardirq_enable_ip;
  unsigned long hardirq_disable_ip;
  unsigned int hardirq_enable_event;
  unsigned int hardirq_disable_event;
  int hardirqs_enabled;
  int hardirq_context;
  unsigned long softirq_disable_ip;
  unsigned long softirq_enable_ip;
  unsigned int softirq_disable_event;
  unsigned int softirq_enable_event;
  int softirqs_enabled;
  int softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
  u64 curr_chain_key;
  int lockdep_depth;
  unsigned int lockdep_recursion;
  struct held_lock held_locks[MAX_LOCK_DEPTH];
  gfp_t lockdep_reclaim_gfp;
#endif
#ifdef CONFIG_UBSAN
  unsigned int in_ubsan;
#endif

/* journalling filesystem info */
  void *journal_info;

/* stacked block device info */
  struct bio_list *bio_list;
```

Linux task_struct from sched.h IX

```
#ifdef CONFIG_BLOCK
/* stack plugging */
  struct blk_plug *plug;
#endif

/* VM state */
  struct reclaim_state *reclaim_state;

  struct backing_dev_info *backing_dev_info;

  struct io_context *io_context;

  unsigned long ptrace_message;
  siginfo_t *last_siginfo; /* For ptrace use.  */
  struct task_io_accounting ioac;
#if defined(CONFIG_TASK_XACCT)
  u64 acct_rss_mem1;  /* accumulated rss usage */
  u64 acct_vm_mem1; /* accumulated virtual memory usage */
  cputime_t acct_timexpd; /* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
  nodemask_t mems_allowed;  /* Protected by alloc_lock */
  seqcount_t mems_allowed_seq;  /* Seqence no to catch updates */
  int cpuset_mem_spread_rotor;
  int cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
  /* Control Group info protected by css_set_lock */
  struct css_set __rcu *cgroups;
  /* cg_list protected by css_set_lock and tsk->alloc_lock */
  struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
  struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
  struct compat_robust_list_head __user *compat_robust_list;
```

Linux task_struct from sched.h X

```
#endif
  struct list_head pi_state_list;
  struct futex_pi_state *pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
  struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
  struct mutex perf_event_mutex;
  struct list_head perf_event_list;
#endif
#ifdef CONFIG_DEBUG_PREEMPT
  unsigned long preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
  struct mempolicy *mempolicy;  /* Protected by alloc_lock */
  short il_next;
  short pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
  int numa_scan_seq;
  unsigned int numa_scan_period;
  unsigned int numa_scan_period_max;
  int numa_preferred_nid;
  unsigned long numa_migrate_retry;
  u64 node_stamp;     /* migration stamp  */
  u64 last_task_numa_placement;
  u64 last_sum_exec_runtime;
  struct callback_head numa_work;

  struct list_head numa_entry;
  struct numa_group *numa_group;

  /*
   * numa_faults is an array split into four regions:
   * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
   * in this precise order.
   *
```

Linux task_struct from sched.h XI

```
   * faults_memory: Exponential decaying average of faults on a per-node
   * basis. Scheduling placement decisions are made based on these
   * counts. The values remain static for the duration of a PTE scan.
   * faults_cpu: Track the nodes the process was running on when a NUMA
   * hinting fault was incurred.
   * faults_memory_buffer and faults_cpu_buffer: Record faults per node
   * during the current scan window. When the scan completes, the counts
   * in faults_memory and faults_cpu decay and these values are copied.
   */
  unsigned long *numa_faults;
  unsigned long total_numa_faults;

  /*
   * numa_faults_locality tracks if faults recorded during the last
   * scan window were remote/local or failed to migrate. The task scan
   * period is adapted based on the locality of the faults with different
   * weights depending on whether they were shared or private faults
   */
  unsigned long numa_faults_locality[3];

  unsigned long numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */

#ifdef CONFIG_ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH
  struct tlbflush_unmap_batch tlb_ubc;
#endif

  struct rcu_head rcu;

  /*
   * cache last used pipe for splice
   */
  struct pipe_inode_info *splice_pipe;

  struct page_frag task_frag;
```

Linux task_struct from sched.h XII

```
#ifdef  CONFIG_TASK_DELAY_ACCT
  struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
  int make_it_fail;
#endif
  /*
   * when (nr_dirtied >= nr_dirtied_pause), it's time to call
   * balance_dirty_pages() for some dirty throttling pause
   */
  int nr_dirtied;
  int nr_dirtied_pause;
  unsigned long dirty_paused_when; /* start of a write-and-pause period */

#ifdef CONFIG_LATENCYTOP
  int latency_record_count;
  struct latency_record latency_record[LT_SAVECOUNT];
#endif
  /*
   * time slack values; these are used to round up poll() and
   * select() etc timeout values. These are in nanoseconds.
   */
  u64 timer_slack_ns;
  u64 default_timer_slack_ns;

#ifdef CONFIG_KASAN
  unsigned int kasan_depth;
#endif
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
  /* Index of current stored address in ret_stack */
  int curr_ret_stack;
  /* Stack of return addresses for return function tracing */
  struct ftrace_ret_stack *ret_stack;
  /* time stamp for last schedule */
  unsigned long long ftrace_timestamp;
  /*
```

Linux task_struct from sched.h XIII

```
   * Number of functions that haven't been traced
   * because of depth overrun.
   */
  atomic_t trace_overrun;
  /* Pause for the tracing */
  atomic_t tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
  /* state flags for use by tracers */
  unsigned long trace;
  /* bitmask and counter of trace recursion */
  unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_KCOV
  /* Coverage collection mode enabled for this task (0 if disabled). */
  enum kcov_mode kcov_mode;
  /* Size of the kcov_area. */
  unsigned  kcov_size;
  /* Buffer for coverage collection. */
  void    *kcov_area;
  /* kcov desciptor wired with this task or NULL. */
  struct kcov *kcov;
#endif
#ifdef CONFIG_MEMCG
  struct mem_cgroup *memcg_in_oom;
  gfp_t memcg_oom_gfp_mask;
  int memcg_oom_order;

  /* number of pages to reclaim on returning to userland */
  unsigned int memcg_nr_pages_over_high;
#endif
#ifdef CONFIG_UPROBES
  struct uprobe_task *utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
  unsigned int  sequential_io;
```

Linux task_struct from sched.h XIV

```
  unsigned int  sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
  unsigned long task_state_change;
#endif
  int pagefault_disabled;
#ifdef CONFIG_MMU
  struct task_struct *oom_reaper_list;
#endif
#ifdef CONFIG_VMAP_STACK
  struct vm_struct *stack_vm_area;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
  /* A live task holds one reference. */
  atomic_t stack_refcount;
#endif
/* CPU-specific state of this task */
  struct thread_struct thread;
/*
 * WARNING: on x86, 'thread_struct' contains a variable-sized
 * structure.  It *MUST* be at the end of 'task_struct'.
 *
 * Do not put anything below here!
 */
};
```

Control de procesos

- Modos de ejecución.
- Creación de procesos.
- Finalización de procesos.
- Jerarquía de procesos.
- Cambio de procesos.
- Ejecución del sistema operativo.

Modos de ejecución

- La mayor parte de los procesadores proporcionan al menos dos modos de ejecución:
  - **modo usuario**: permite la ejecución de instruciones que no afectan a otros procesos.
  - **modo núcleo**: permite la ejecución de todas las instrucciones.
- Existen otros modos intermedios se solían usarse en controladores de dispositivos y bibliotecas.
- Un bit en la palabra de estado indica en que modo se está ejecutando el procesador.
  - El **bit** puede consultarse como el resto de la **palabra de estado**.
  - Se modifica cuando se produce una llamada al sistema o una interrupción.
  - Al retornar de la llamada al sistema o de la interrupción se devuelve el valor original de dicho bit desde la pila.

Modos de ejecución

- Funciones típicas del núcleo de un sistema operativo:
  - Gestión de procesos:
    - creación y terminación de procesos.
    - planificación y activación de procesos.
    - intercambio de procesos.
    - sincronización y comunicación entre procesos.
    - gestión de los bloques de control de procesos.
  - Gestión de memoria:
    - reserva de espacios de direcciones.
    - intercambio (*swapping*).
    - gestión de páginas y/o segmentos.
  - Gestión de E/S:
    - gestión de almacenes temporales (*buffers*).
    - reserva de canales de DMA y dispositivos.
  - Funciones de soporte:
    - gestión de interrupciones.
    - auditoría.
    - monitorización.

Creación de procesos

- Salvo sistemas extremadamente simples los SO deben tener mecanismos para la creación de nuevos procesos.
- Posibles causas de la creación de un procesos:
  1. Inicialización del sistema.
     - interactivos / no interactivos.
     - primer / segundo plano.
  2. Llamada al sistema para crear un proceso.
     - `fork() + exec() / CreateProcess()`.
  3. Petición de usuario.
     - lanzamiento de una nueva aplicación desde el interfaz de usuario.
  4. Inicio de un proceso por lotes.
     - sistemas de colas de trabajos en servidores.

Creación de procesos

Pasos en la creación de un proceso:

1. Asignar un **identificador de proceso único**.
2. **Reservar espacio** para el proceso:
   - estructuras de datos del SO (**PCB**).
   - imagen del proceso.
3. **Inicialización** del bloque del control del proceso (**PCB**).
   - ppid, estado, ip, sp, prioridad, E/S,...
4. Establecimiento de **enlaces** adecuados:
   - cola de trabajos.
5. Creación o expansión de otras estructuras de datos:
   - auditoría, monitorización, análisis de rendimiento,...

Finalización de procesos

- Una vez creados los procesos se ejecutan y, generalmente, realizan la tarea para la que se lanzarón.
- Causas de finalización de un proceso:
  - Voluntarias:
    - Terminación normal: la mayoría de los procesos realizan su trabajo y devuelven el control al SO mediante la llamada al sistema `exit() / ExitProcess()`.
    - Terminación por error: falta argumento,...
  - Involuntarias:
    - Error fatal: instrucción privilegiada, excepción de coma flotante, violación de segmento,...
    - Terminado por otro proceso: mediante la llamada al sistema `kill() / TerminateProcess()`
- Una vez finalizado es necesario...
  - auditoría/contabilidad de uso de recursos.
  - recuperar/reciclar recursos.

Jerarquía de procesos

UNIX:

- El uso de **fork()** crea una relación jerárquica entre procesos.
- **init** o **systemd** suelen ser primer proceso del sistema y de él dependen todos los demás.
- La relación no puede modificarse.
- Si un proceso padre finaliza antes que sus hijos estos pasan a depender del ancestro previo.
- Útil para llevar a cabo operaciones sobre grupos de procesos.

Windows:

- **CreateProcess()** no establece relación entre procesos.
- Al crear un nuevo proceso se consigue un objeto que permite su control.
  - La propiedad de este objeto puede pasarse a otro proceso.

Jerarquía de procesos

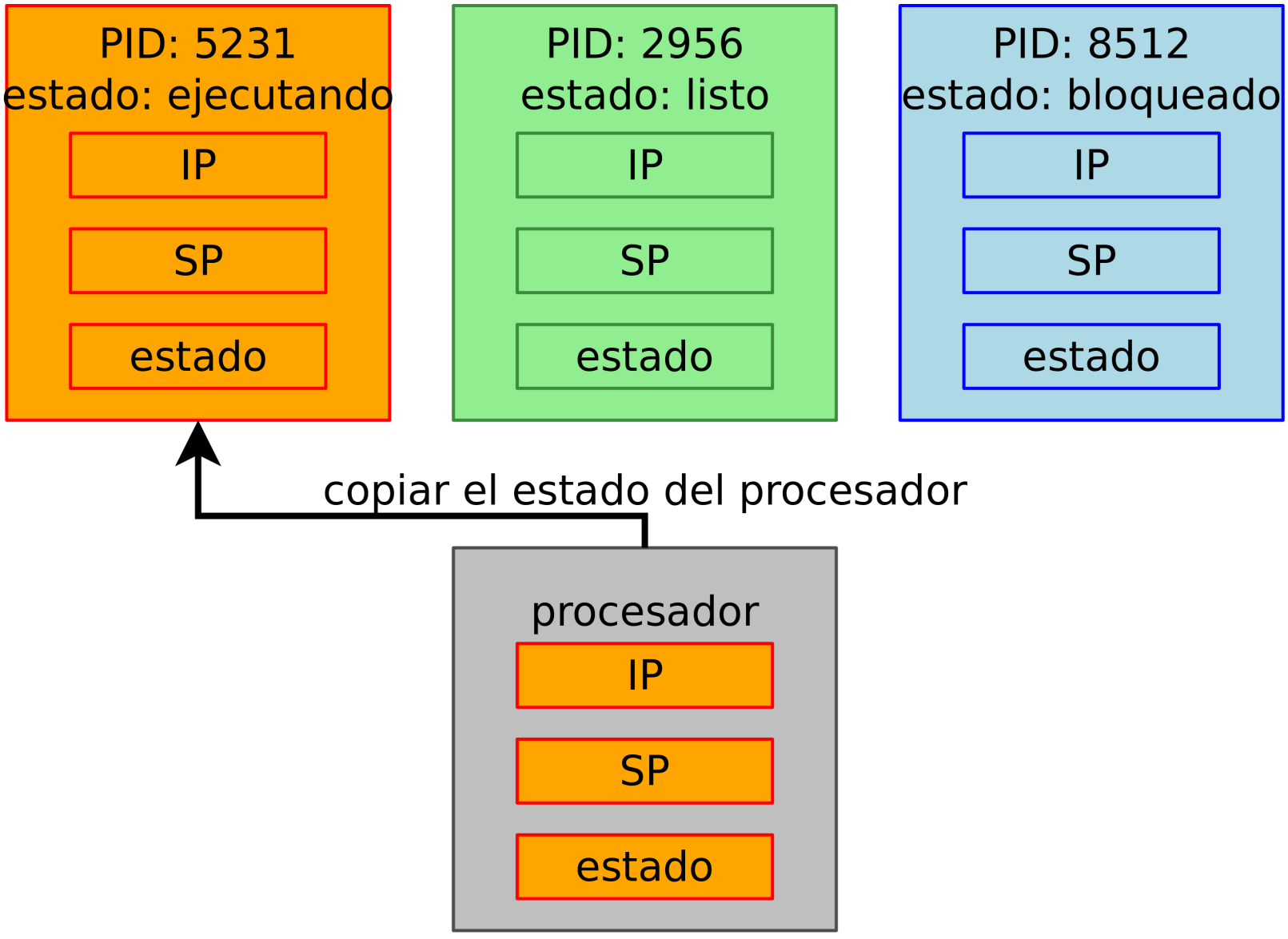
pstree -Ant

```
systemd-+-systemd-journal
        |-systemd-udevd
        |-auditd-+-{auditd}
        |        `-audispd-+-sedispatch
        |                  `-{audispd}
        |-systemd-logind
        |-alsactl
        |-irqbalance---{gmain}
        |-dbus-daemon---{dbus-daemon}
        |-firewalld---{gmain}
        |-ModemManager-+-{gmain}
        |              `-{gdbus}
        |-accounts-daemon-+-{gmain}
        |                 `-{gdbus}
        |-avahi-daemon---avahi-daemon
        |-smartd
        |-rsyslogd-+-{in:imjournal}
        |          `-{rs:main Q:Reg}
        |-rtkit-daemon---2*[{rtkit-daemon}]
        |-abrtd-+-{gmain}
        |       `-{gdbus}
        |-gssproxy---5*[{gssproxy}]
        |-chronyd
        |-polkitd-+-{gmain}
        |         |-{gdbus}
        |         |-{JS GC Helper}
        |         |-{JS Sour~ Thread}
        |         `-{polkitd}
        |-3*[abrt-dump-journ]
        |-NetworkManager-+-{gmain}
        |                |-{gdbus}
        |                `-dhclient
        |-cupsd
        |-php-fpm---7*[php-fpm]
        |-sshd
        |-libvirtd---15*[{libvirtd}]
        |-httpd-+-2*[httpd]
        |       |-3*[httpd---64*[{httpd}]]
        |       `-httpd---80*[{httpd}]
        |-crond
        |-gdm-+-{gmain}
        |     |-{gdbus}
        |     |-gdm-session-wor-+-{gmain}
        |     |                 |-{gdbus}
        ...............................
```

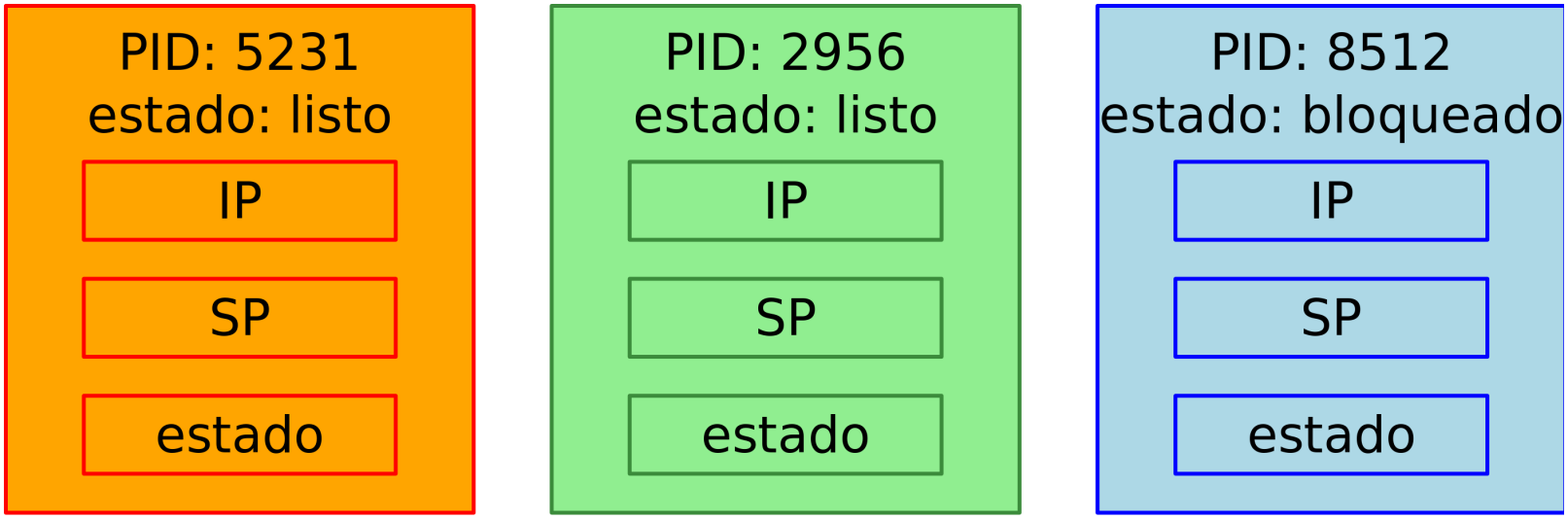
Cambio de proceso

- **Cambio de proceso**:
  - Operación costosa.
    - Linux 2.4.21: 5.4μs/13200 ciclos en un Pentium IV a 2.4GHZ.
  - Eventos que pueden provocar un **cambio de proceso**:
    - Interrupción: interrupción del reloj, finalización de operación de E/S o DMA,...
    - Excepción: fallo de página/segmento, llamada al sistema (`int`/`syscall`), operación de E/S,...
- **Cambio de modo**: cambio del modo de privilegio con el que se ejecuta el procesador.
  - Operación sencilla y poco costosa.
- **Cambio de contexto** es ambiguo... ¿a qué cambio se refiere? ¿proceso? ¿modo? ¿**ambos**?

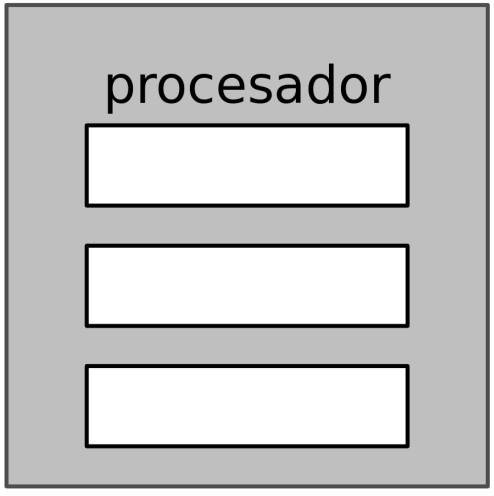
Cambio de proceso



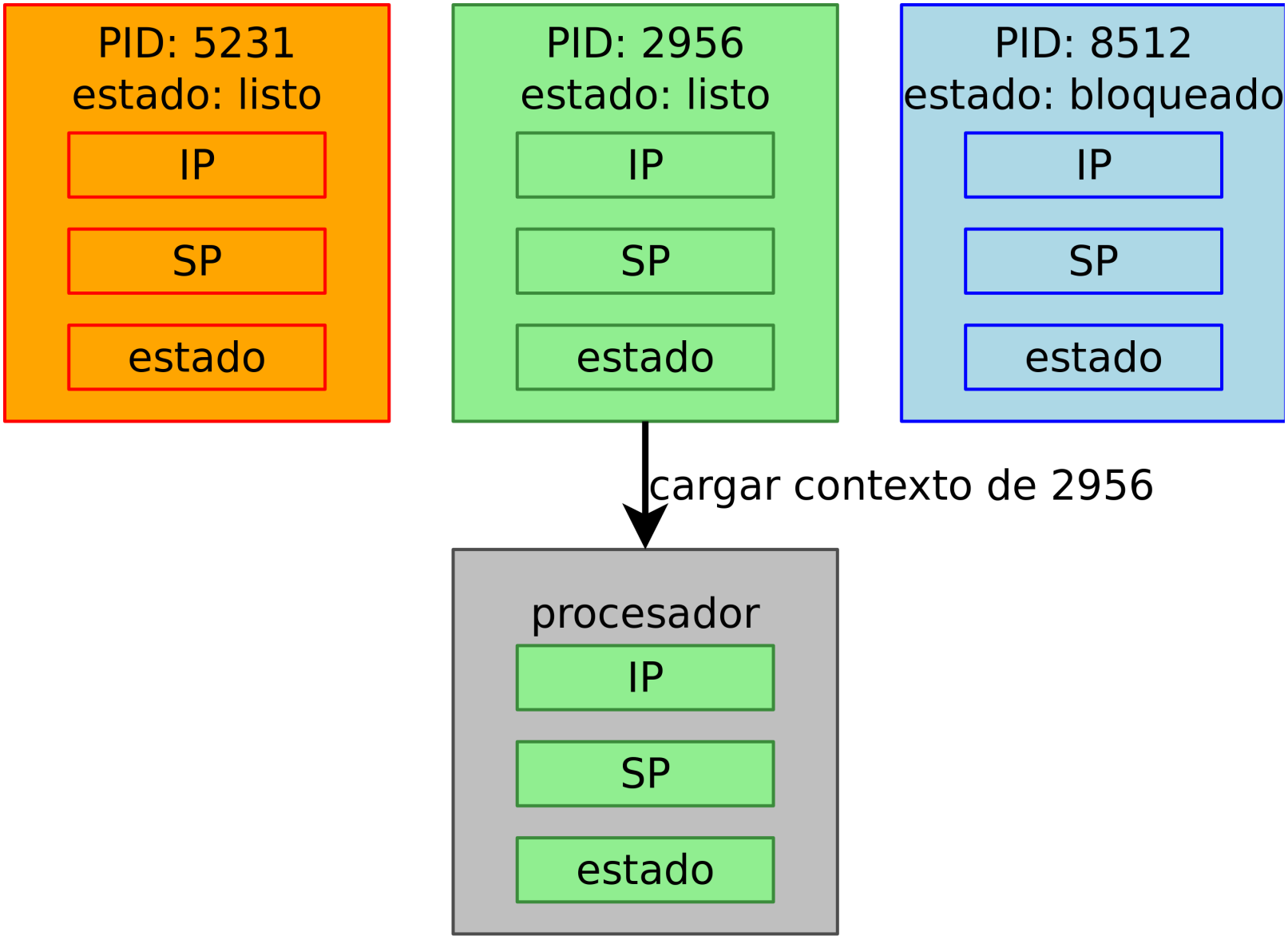
Cambio de proceso



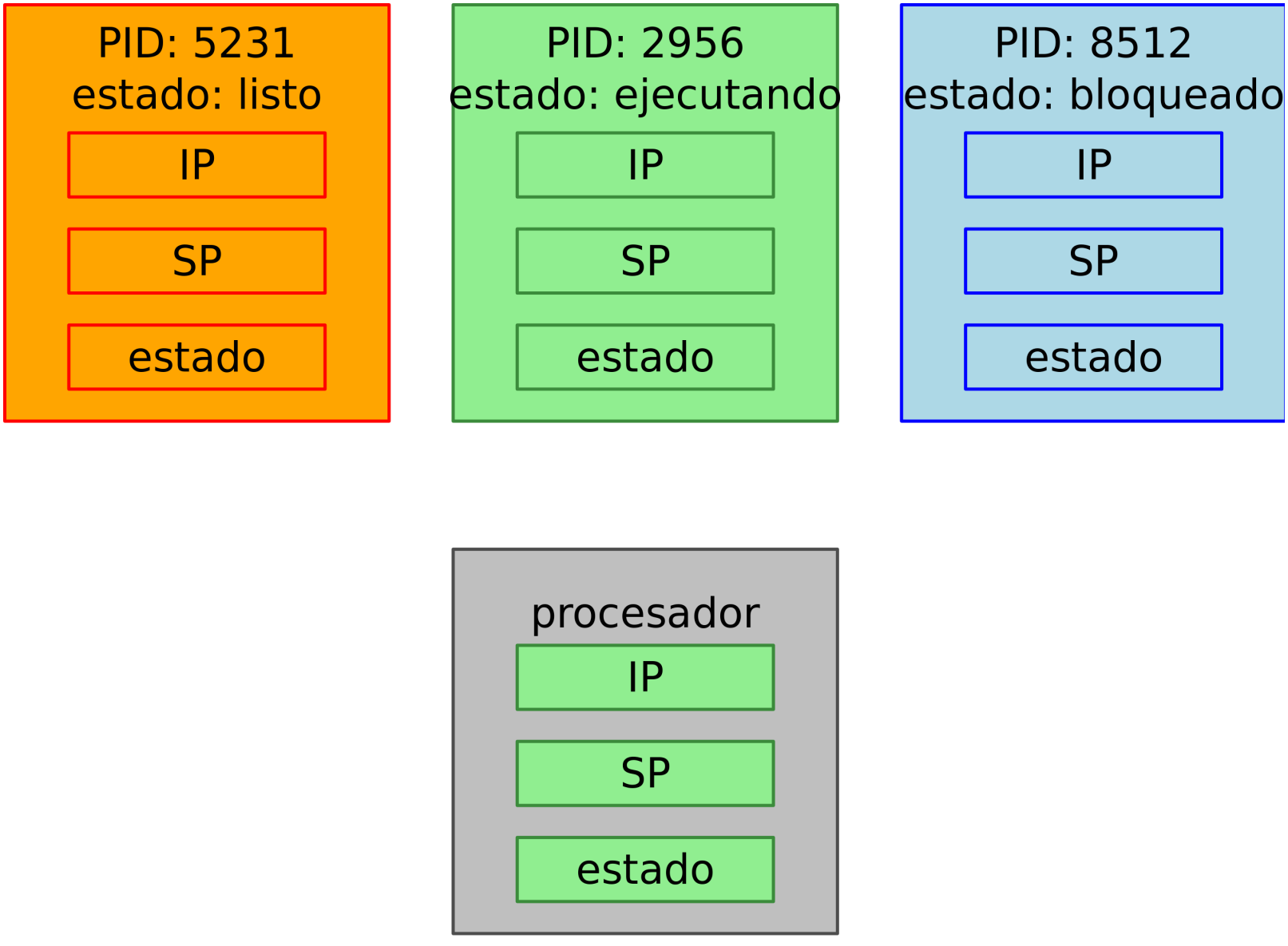
ahora se debe seleccionar un nuevo proceso



Cambio de proceso



Cambio de proceso



Cambio de proceso

Pasos a seguir para realizar el cambio de proceso:

1. Salvar el estado del procesador.
2. Actualizar el estado del bloque de control del proceso.
   - Como mínimo cambiar el estado (ej: ejecutando → preparado).
3. Mover el PCB a la cola adecuada (ej: preparado).
4. Seleccionar el nuevo proceso a ejecutar.
5. Actualizar el estado del bloque de control del proceso.
   - Como mínimo cambiar el estado (ej: preparado → ejecutando).
6. Actualizar las estructuras de datos de gestión de memoria.
7. Restaurar el estado del proceso al que tenía en el momento de abandonar el estado ejecutando.

UNIX: fork() + exec() + wait() + exit()

```
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
  pid_t pid = fork();

  if (pid < 0) // error
    {
      cerr << "error en fork" << endl;
    }
  else if (pid == 0) // hijo
    {
      cout << "hijo" << endl;
      execlp("/bin/ls", "ls", NULL);
    }
  else // padre
    {
      cout << "padre" << endl;
      wait(pid);
    }
}
```

padre

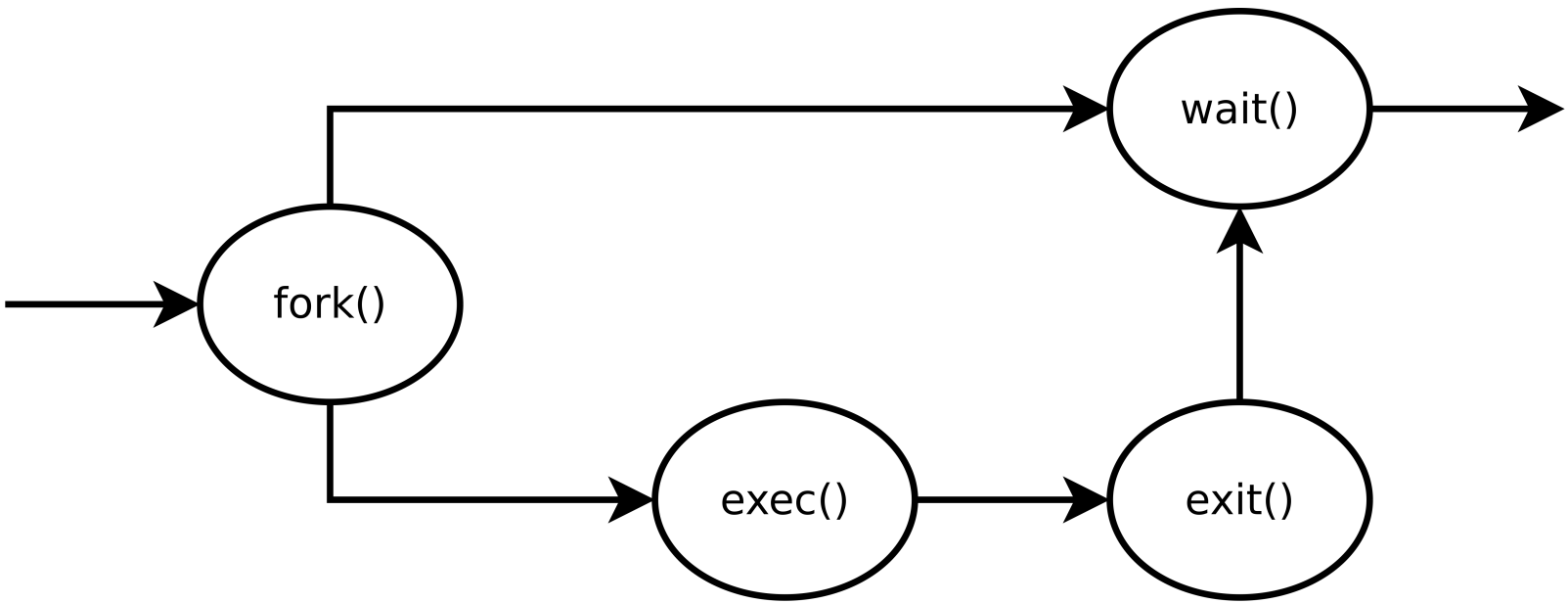
```
cout << "padre" << endl;
wait(pid);
exit(0);
```

hijo

```
cout << "hijo" << endl;
execlp("/bin/ls", "ls", NULL);
exit(0);
```

UNIX: `fork()` + `exec()` + `wait()` + `exit()`

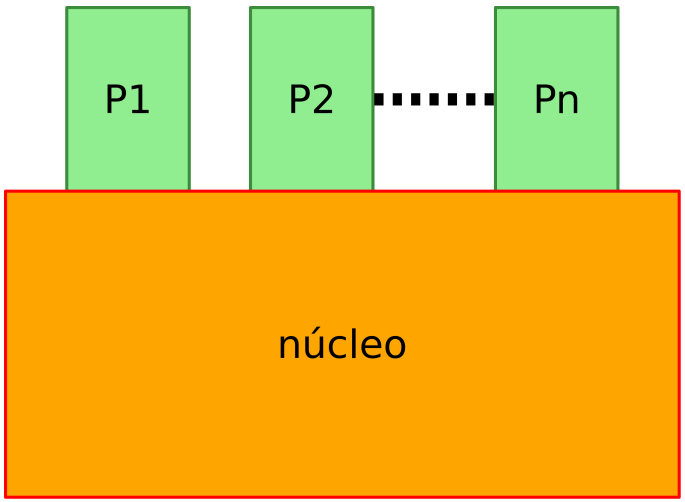
- **fork()**: crea un nuevo proceso.
- **exec()**: cambia la imagen de un proceso.
- **wait()**: permite al padre esperar al hijo.
- **exit()**: finaliza el proceso.



Ejecución del sistema operativo

Núcleo **independiente**

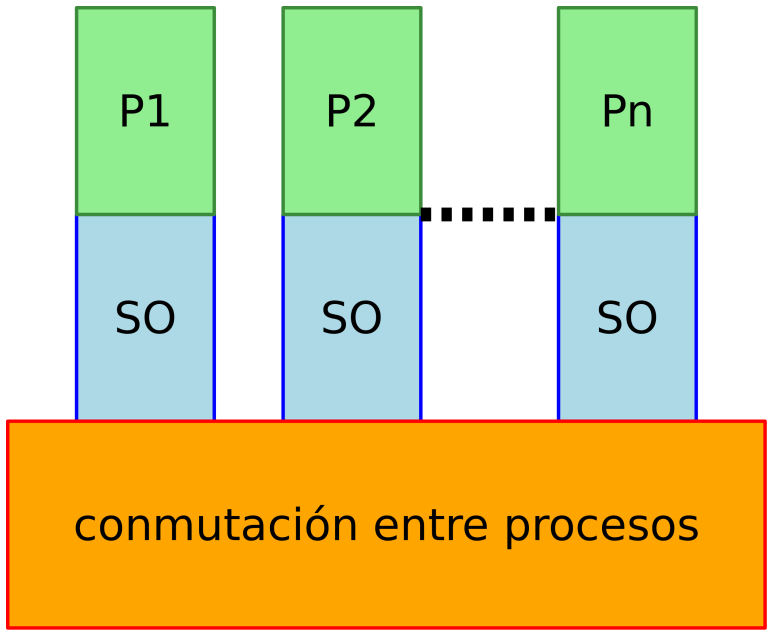
- Método más antiguo
- El SO no es un proceso (aunque se comporte como uno).
- El SO dispone de áreas de memoria y pila propias.
- El concepto de proceso es aplicable sólo a programas de usuario.
- Inconveniente: cada evento cuesta un cambio de proceso y modo.



Ejecución del sistema operativo

Ejecución **dentro** del los procesos de usuario

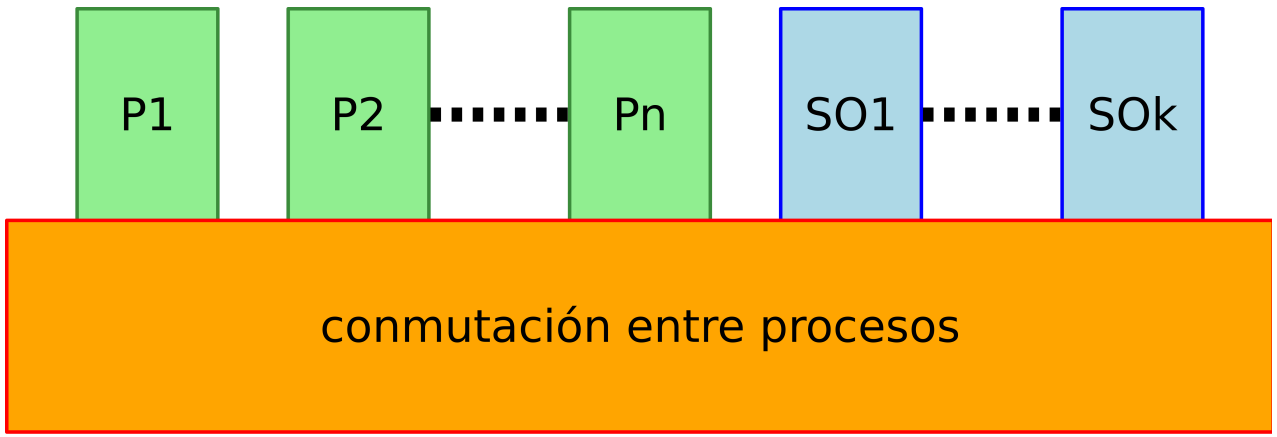
- EL SO es como un conjunto de rutinas que el usuario puede invocar y que están situadas dentro de su entorno.
- A la imagen de cada proceso se une la del SO.
- Ventaja: cada evento cuesta sólo un cambio de modo.
- Inconveniente: restamos espacio al proceso de usuario.



Ejecución del sistema operativo

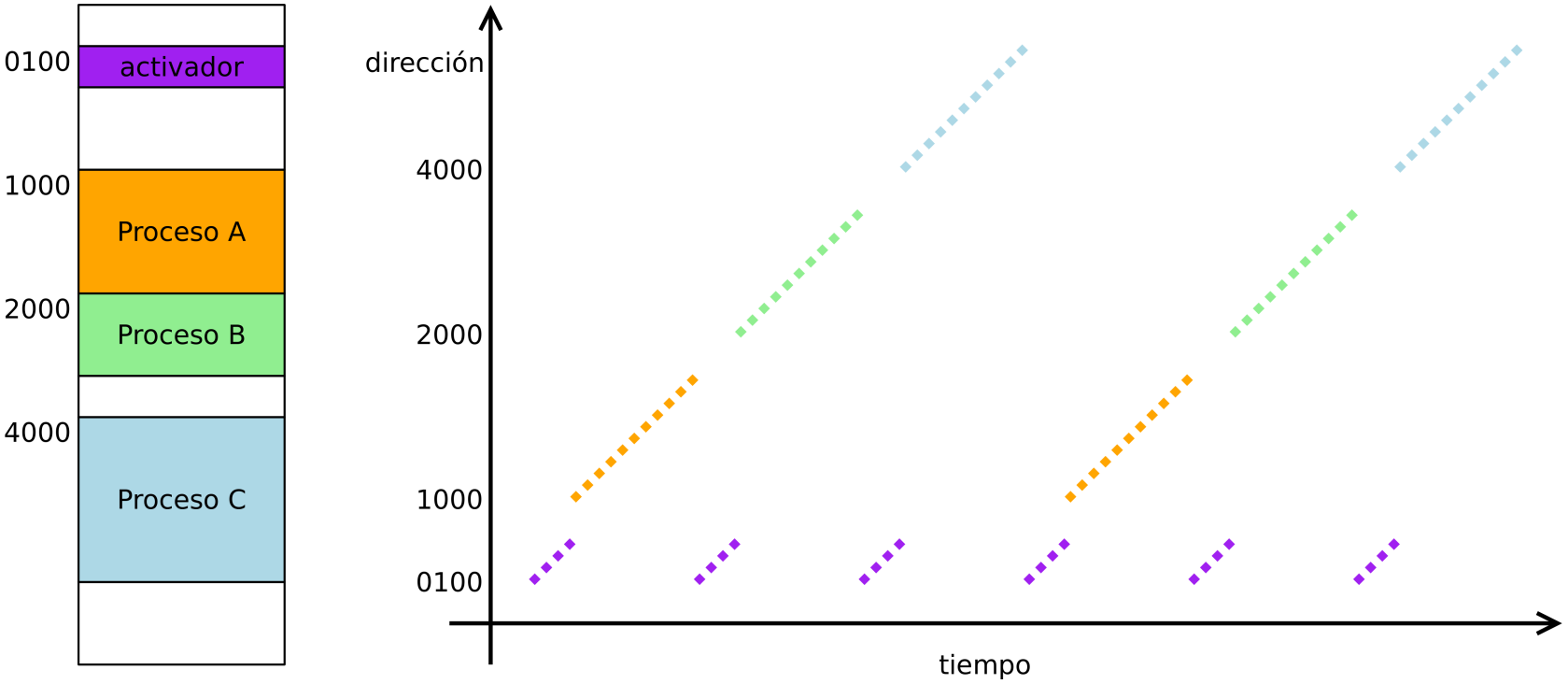
Sistemas operativos basados en **procesos**.

- El SO se implementa como una colección de procesos.
- Ventajas:
  - modularidad y facilidad de programación.
  - mejora del rendimiento en sistemas multiprocesador.
- Inconvenientes:
  - cada evento cuesta varios cambios de proceso y modo.



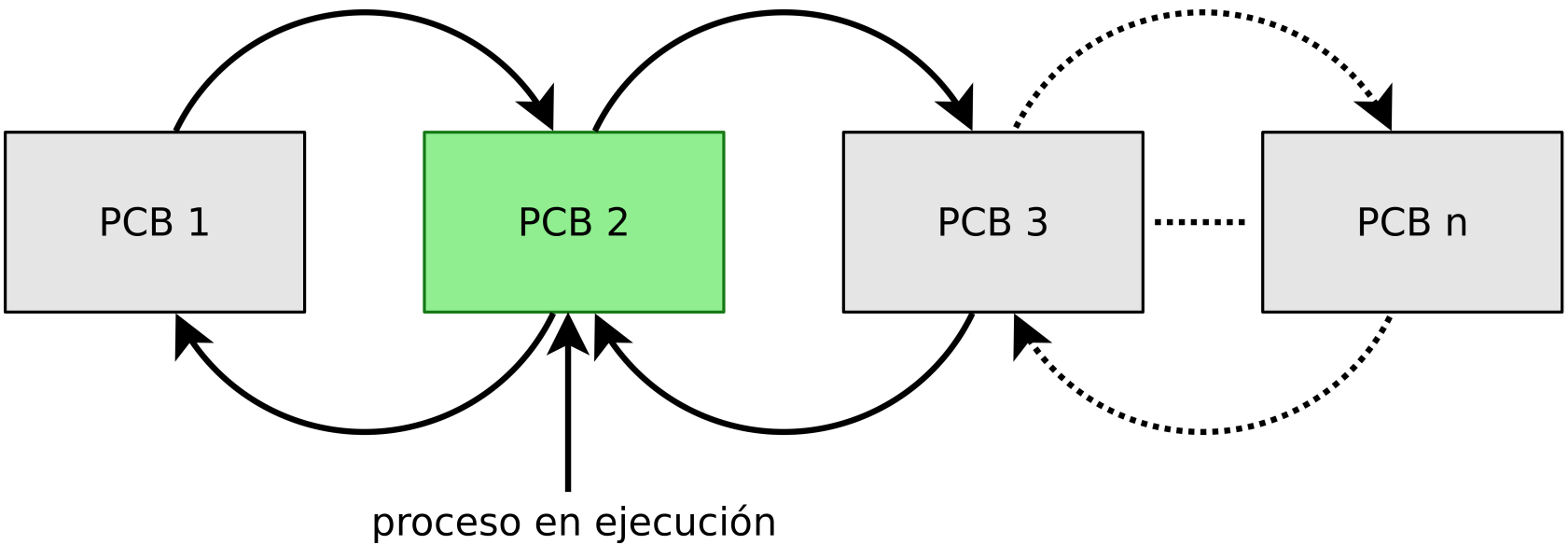
Traza de un proceso

- Comportamieto de un proceso = lista de instrucciones que ejecuta $\Longrightarrow$ **traza**.
- **Activador**: programa encargado de cambiar entre los PCBs de los procesos para ejecutar un proceso u otro.

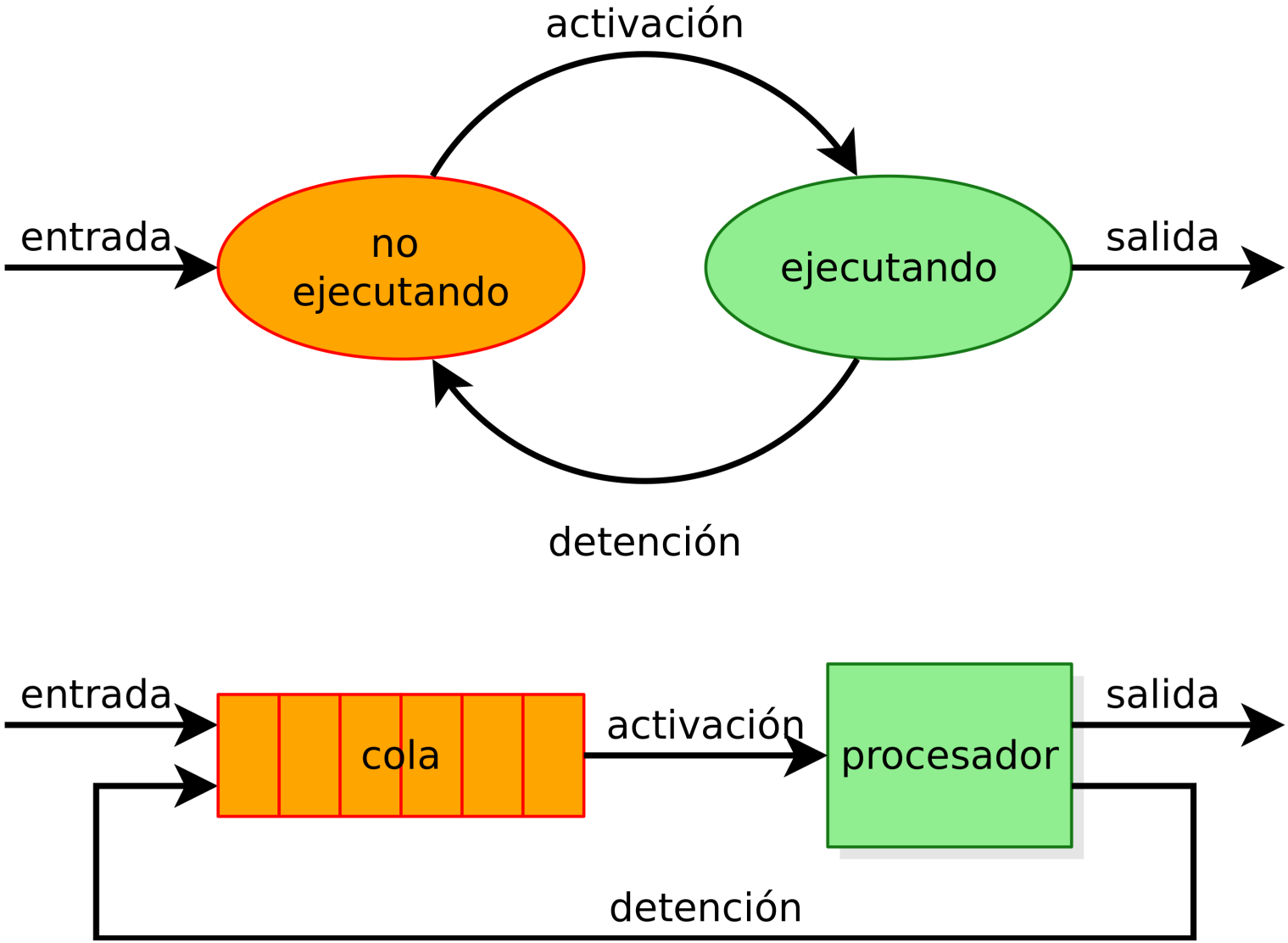


Cola de procesos

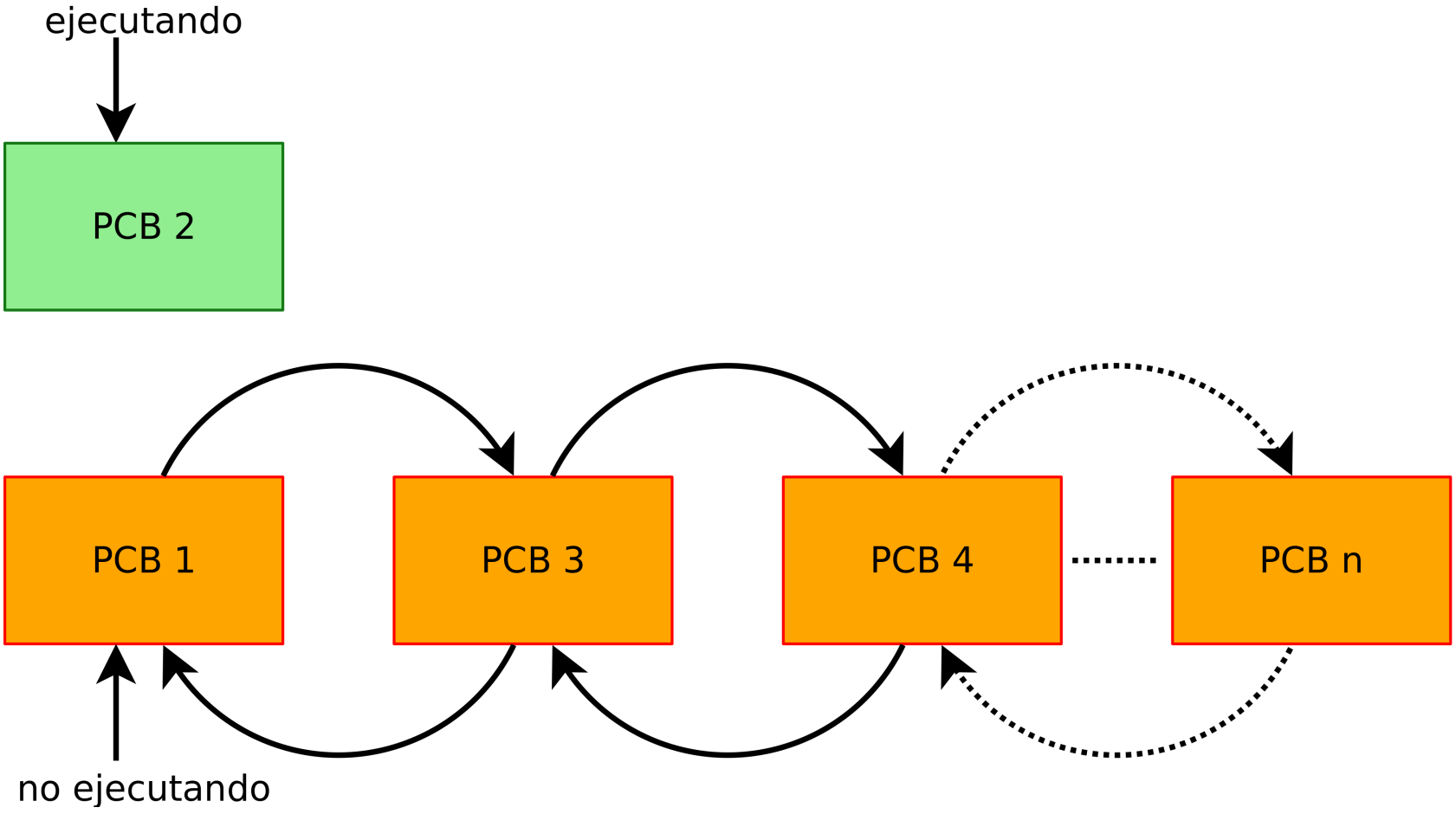
- Creación de un proceso = crear PCB + cargar imagen.
- Lista de procesos = lista de PCBs.
- **Planificador** (*"scheduler"*):
  - Parte del SO que escoge el siguiente proceso a ejecutar.
  - Gestor de las colas de planificación.
- **Activador** (*"dispatcher"*): parte del planificador que realiza el intercambio de procesos.
- Ejecución = encolar + activar.



Modelo de 2 estados



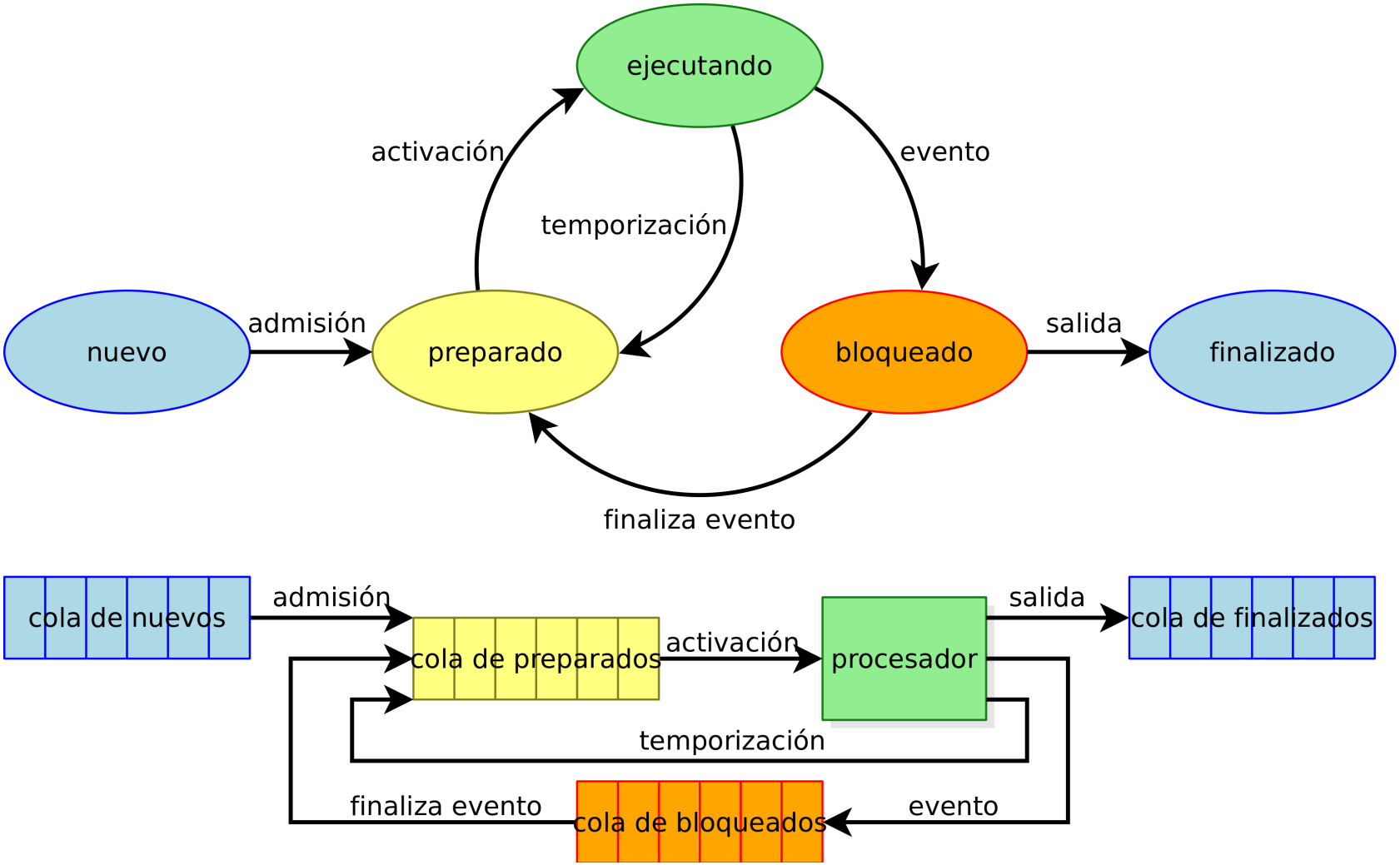
Modelo de 2 estados



Modelo de 2 estados

- Estados:
  - **Ejecutando**: proceso en ejecución.
  - **No ejecutando**: proceso que no se está ejecutando.
- Transiciones:
  - **Ejecutando** $\rightarrow$ **no ejecutando**: evento o temporización.
  - **No ejecutando** $\rightarrow$ **ejecutando**: temporización o fin de evento.
- Inconvenientes:
  - No permite discriminar fácilmente la razón por la que un proceso no se encuentra en ejecución.
  - Solución: subdividir el estado "no ejecutando" para reflejar el motivo.

Modelo de 5 estados



Modelo de 5 estados

- Estados:
  - **Nuevo**: el proceso ha sido creado.
  - **Preparado**: proceso a la espera de que se le asigne un procesador.
  - **Ejecutando**: proceso actualmente en ejecución.
  - **Bloqueado**: proceso que no puede continuar hasta que finalice un evento.
  - **Finalizado**: proceso finalizado.

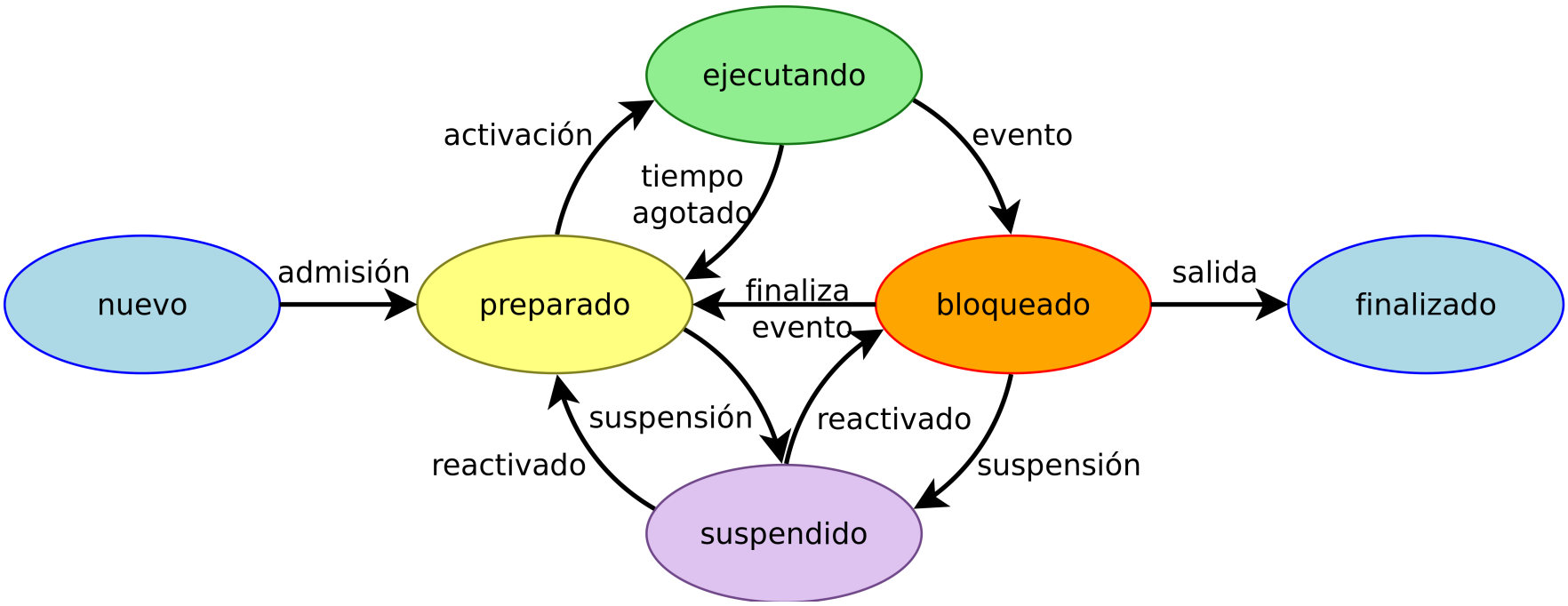
Modelo de 5 estados

- Transiciones:
  - **Nuevo** → **preparado**: se admite un nuevo proceso en el sistema.
  - **Preparado** → **ejecutando**: el planificador selecciona el proceso para su ejecución.
  - **Preparado** → **finalizado**: padre termina hijo.
  - **Ejecutando** → **finalizado**: proceso finalizado.
  - **Ejecutando** → **preparado**: tiempo de procesador agotado.
  - **Ejecutando** → **bloqueado**: se produce un evento.
  - **Bloqueado** → **preparado**: finalización de evento.
  - **Bloqueado** → **finalizado**: padre termina hijo.

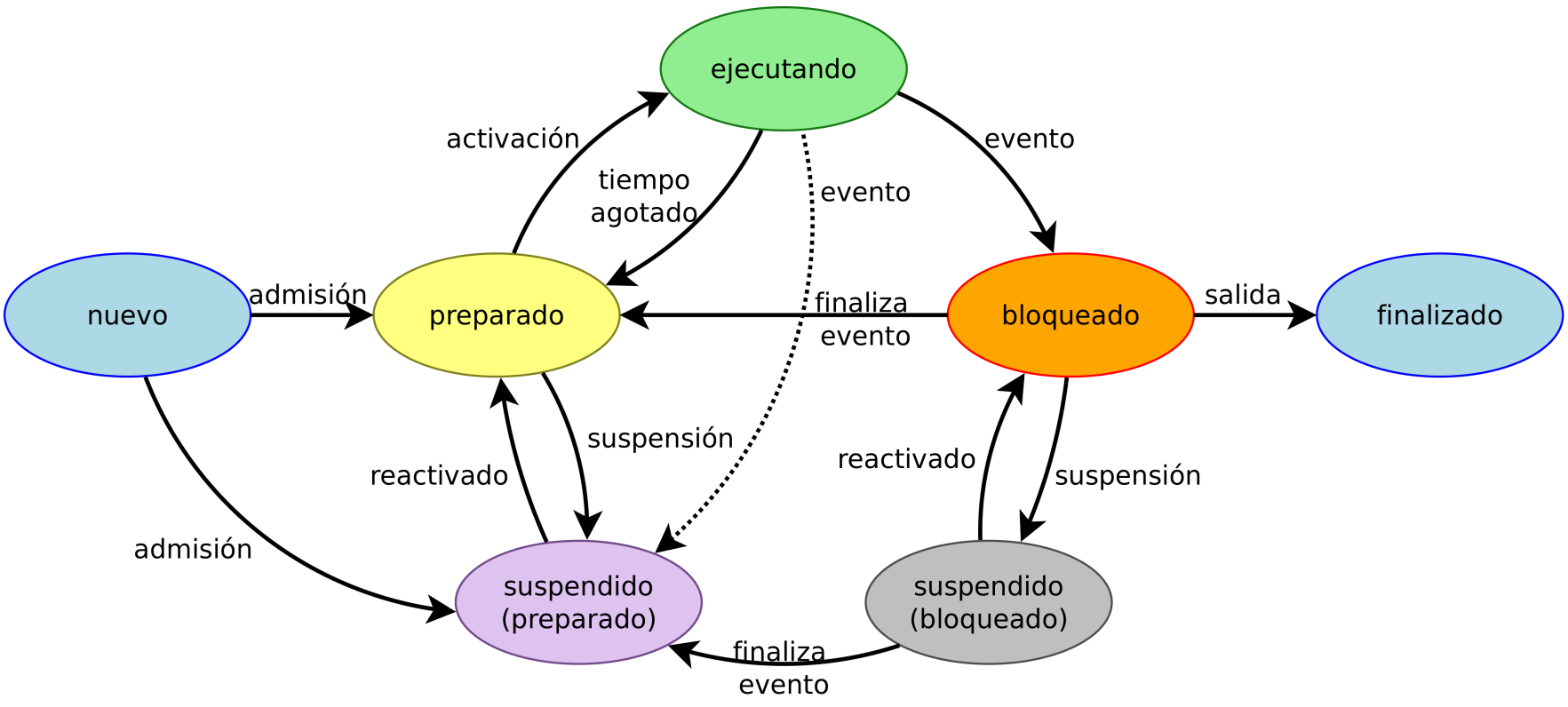
Modelos de +5 estados

- Existe una buena razón para añadir al menos un nuevo estado: **suspendido**.
- Objetivo de la multiprogramación: aprovechar al máximo el procesador debido a la lentitud de las operaciones de E/S.
- ¿Resuelve el problema el modelo de 5 estados? $\Longrightarrow$ no.
  - Causa: diferencia de velocidad procesador/dispositivos de E/S.
  - Todos los procesos podrían llegar a estar bloqueados en espera de E/S, un cierto recurso o la finalización de un subproceso.
  - Solución: añadir más procesos.
  - Problema: falta de memoria.
- Circulo vicioso de difícil solución:
  - Solución cara: añadir más memoria
  - Solución barata: memoria de intercambio (*"swap"*).
- **Intercambio** (*"swapping"*): proceso de expulsión de un

Modelo de 6 estados



Modelo de 7 estados



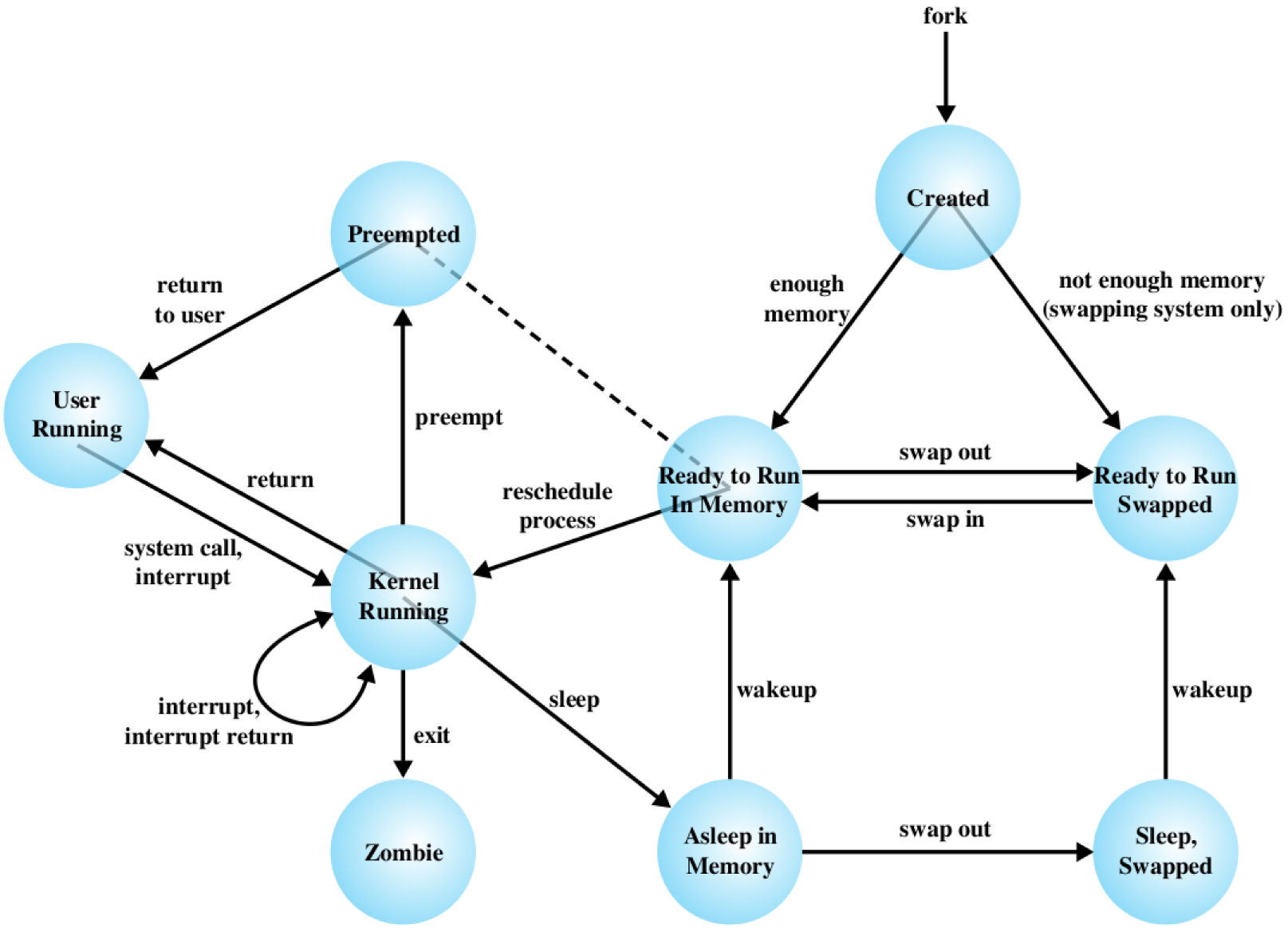
Modelo de 7 estados

- Motivación: el reactivar un proceso sólo para descubrir que está bloqueado es muy costoso.
- Nuevos estados:
  - **Suspendido (bloqueado)**: proceso en área de intercambio y esperando un evento.
  - **Suspendido (preparado)**: proceso en área de intercambio y a la espera de espacio en memoria principal.

Modelo de 7 estados

- Nuevas transiciones:
  - **bloqueado → suspendido (bloqueado)**: no hay procesos preparados o estos consumen demasiada memoria.
  - **suspendido (bloqueado) → suspendido (preparado)**: sucede el evento por el que estaba bloqueado.
  - **suspendido (preparado) → preparado**: no quedan procesos preparados en el sistema o tiene mayor prioridad que los preparados.
  - **preparado → suspendido (preparado)**: liberar memoria o dejar sitio para un proceso bloqueado de mayor prioridad.
  - **nuevo → suspendido (preparado)**: control de carga del sistema.
  - **suspendido (bloqueado) → bloqueado**: queda memoria libre o proceso de alta prioridad.
  - **ejecutando → suspendido (preparado)**: un proceso agota su tiempo y hay que liberar memoria para un proceso suspendido de mayor priridad.

Diagrama de transiciones entre estados en UNIX



Planificación

- Los procesos pueden cambiar varias veces de cola de planificación a lo largo de su vida.
- La parte del SO encargada de realizar estos cambios es el planificador.
- Tipos de planificadores:
  - Corto plazo: selecciona entre los procesos preparados uno para ejecutar.
    - Ejecución muy frecuentemente, ej: cada 10..100ms.
  - Medio plazo: decide que procesos pasar al área de intercambio y así controla el grado de multiprogramación.
  - Largo plazo: selecciona que procesos poner en ejecución, ej: sistema por lotes.
    - ejecución en función de la carga del sistema, cada varios minutos o cuando finaliza un proceso.

El planificador en acción...

```
/usr/bin/time -v find / -iname *.h -exec grep '^
struct task_struct {$' {} +
```

```
User time (seconds): 1.79
System time (seconds): 1.75
Percent of CPU this job got: 98%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.59
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 20004
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 33152
Voluntary context switches: 65
Involuntary context switches: 568
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 1
```

Comunicación entre procesos

- Los procesos que se ejecutan concurrentemente pueden ser...
  - **Independientes**: no afecta ni es afectado por otros procesos (no comparten datos).
  - **Cooperantes**: puede afectar y ser afectado por otros procesos (si comparten datos).
- El SO debe proporcionar mecanismos para crear, comunicar y terminar procesos.
- Motivos para cooperar:
  - Compartir información: capacidad de acceso y economía de recursos.
  - Acelerar los cálculos: realizar las tareas más rápidamente.
  - Modularidad: facilidad de creación de programas.
  - Conveniencia: multitarea.

Comunicación entre procesos

- Métodos de comunicación:
  - Memoria compartida:
    - los procesos comparten un área de memoria.
    - comunicación responsabilidad de los procesos.
  - Paso de mensajes.
    - los procesos intercambian mensajes.
    - comunicación responsabilidad del sistema operativo.