Sistemas operativos II

Procesos en UNIX

- Introducción sistema operativo UNIX
- Procesos en UNIX
- Planificación
- Creación y terminación de procesos
- Señales
- Comunicación entre procesos

Procesos en UNIX

- Introducción sistema operativo UNIX
- Procesos en UNIX
- Planificación
- Creación y terminación de procesos
- Señales
- Comunicación entre procesos

Antecedentes

- **UN**iplexed **I**nformation **C**omputing **S**ervice (versión monousuario de MULTICS)
- Primera implementación sobre SEC PDP-7 (1969) por Ken Thomson y Dennis Ritchie
- portada a PDP-11/20 con *runoff*. Adoptada por Laboratorios BELL como procesador de textos en 1970
- 1972 2nd Edition
- 1973 Implementación en C (Thompson y Ritchie)
- 1974 4th Edition
- 1977 5th & 6th Edition
- 1979 7th Edition (ascendiente directo de las versiones actuales)

System V

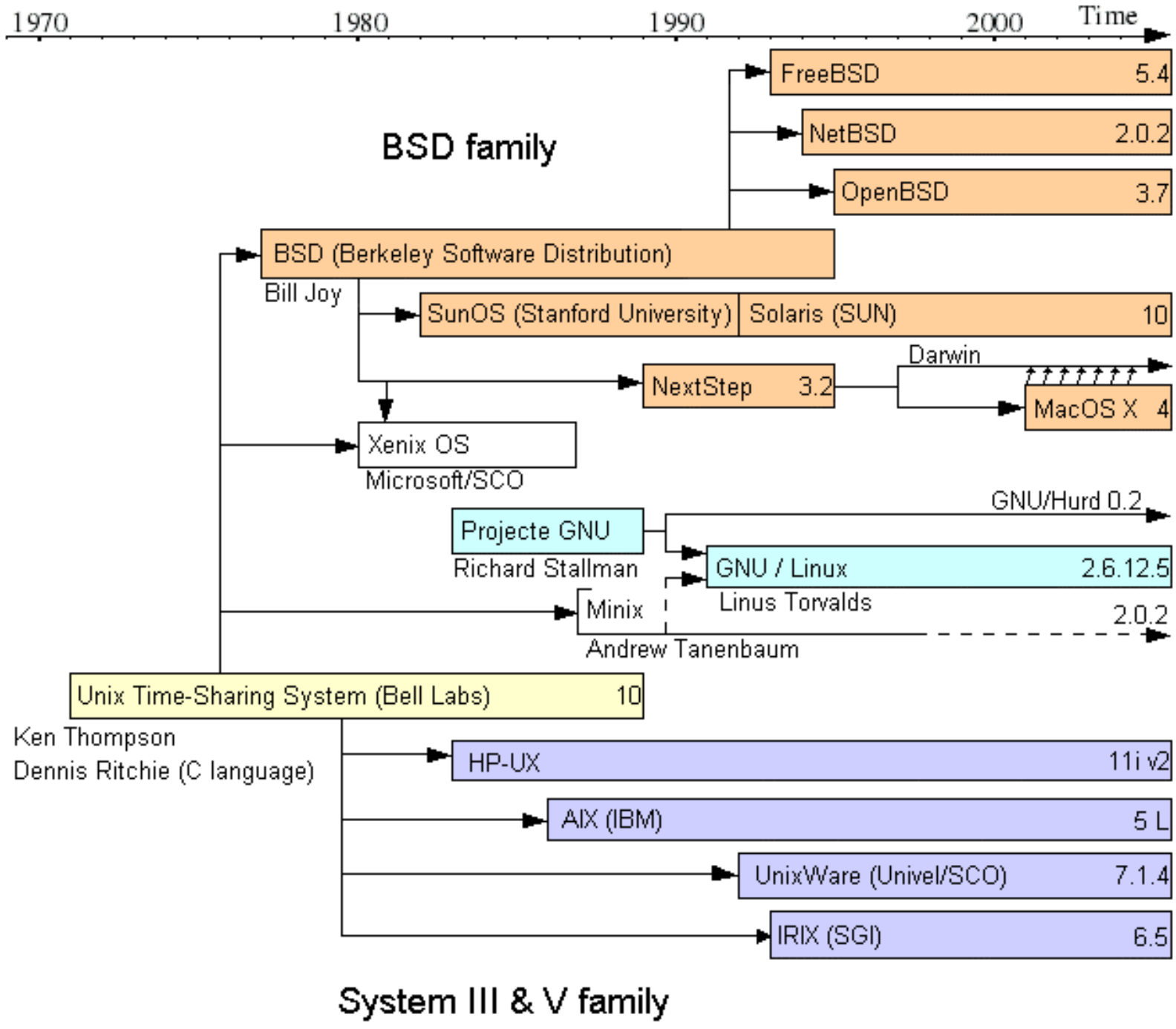
- Introducido por ATT en 1983. System V release 1. Compromiso para mantener compatibilidad ascendente
- System V release 1: vi, biblioteca curses
- System V release 2 (1985): protección y bloqueo de archivos
- System V release 2.1: paginación bajo demanda
- System V release 3 (1987): redes
- System V release 4: unifica versiones de distintos fabricantes

BSD

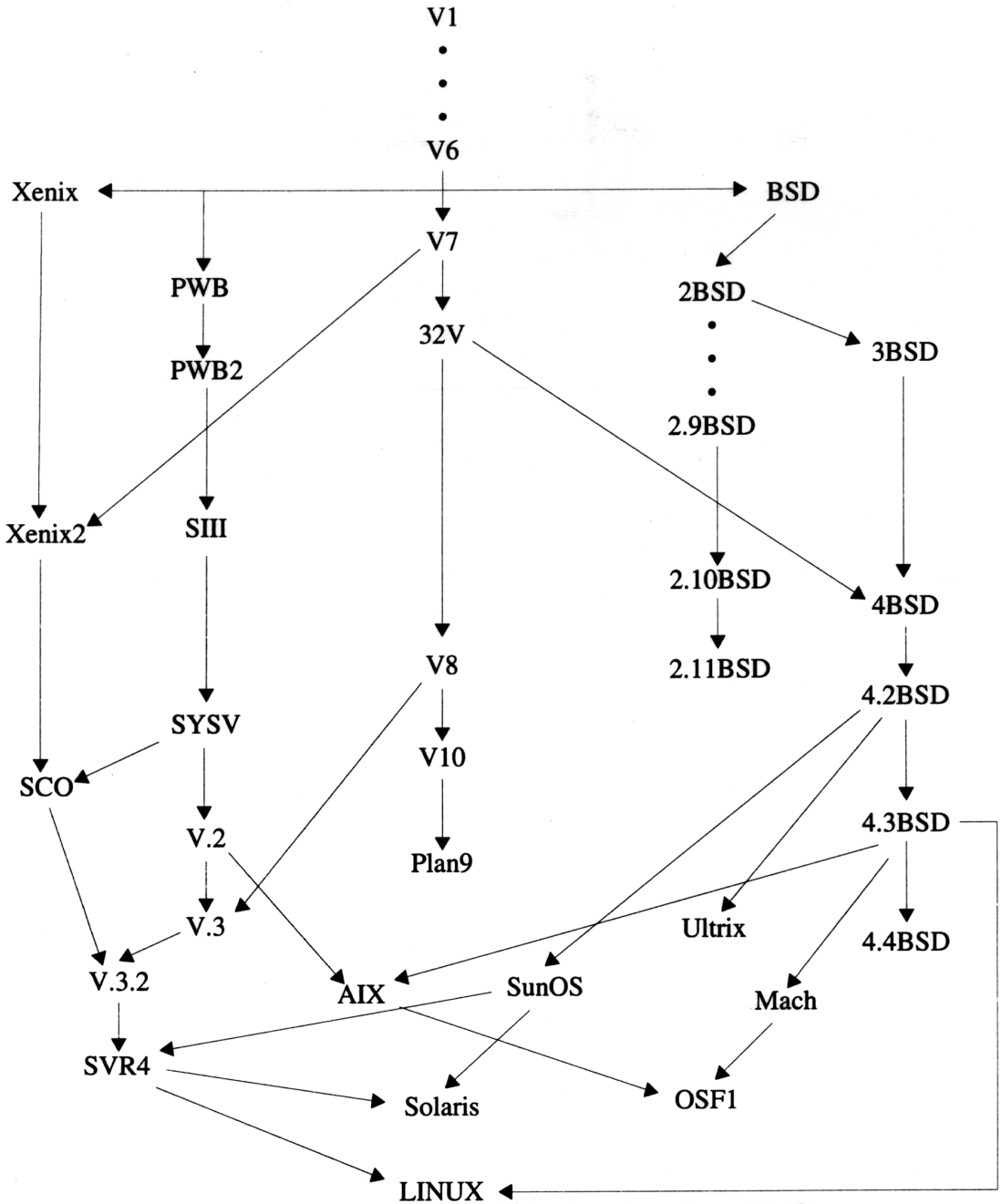
- Universidad de Berkley entró en contacto un UNIX 4th edition
- Bill Joy en 1977 diseño un añadido para las 6th edition llamado Berkeley Software Distribution (compilador pascal, cshell, ex..)
- 1978 2BSD
- 1979 3BSD basada en 2BSD y 7th edition: memoria virtual (DEV VAX-11/780)
- DARPA(Defence Advanced Research Project Agency) consolida la 4BSD
- 1983 4.1 BSD
- 4.2BSD: nuevo sistema de archivos
- 1987 4.3BSD
- Sun Microsystems añadió NFS a 4.3 BSD
- Actualmente freeBSD, openBSD netBSD

POSIX

- Término acuñado por Richard Stallman
- **P**ortable **O**perating **S**ystem **I**nterface for uni**X**
- Designada con IEEE 1003, ISO/IEC 9945
- Familia de estándares que definen una API para software compatible con distintas variantes de UNIX
- Varias extensiones
  - POSIX.1: Incorpora estándar ANSI C
  - POSIX.1b: Tiempo real
  - POSIX.1c: Threads


1970
1980
1990
2000
Time
BSD family
FreeBSD 5.4
NetBSD 2.0.2
OpenBSD 3.7
BSD (Berkeley Software Distribution)
Bill Joy
SunOS (Stanford University)
Solaris (SUN) 10
Darwin
NextStep 3.2
MacOS X 4
Xenix OS
Microsoft/SCO
GNU/Hurd 0.2
Projecte GNU
Richard Stallman
GNU / Linux 2.6.12.5
Linus Torvalds
Minix
2.0.2
Andrew Tanenbaum
Unix Time-Sharing System (Bell Labs) 10
Ken Thompson
Dennis Ritchie (C language)
HP-UX 11i v2
AIX (IBM) 5 L
UnixWare (Univel/SCO) 7.1.4
IRIX (SGI) 6.5
System III & V family

Relación entre las distintas variedades de UNIX



- El núcleo reside en un fichero (/unix, /vmunix /vmlinuz /kernel.GENERIC..) que se carga al arrancar la máquina (procedimiento bootstrap)
- El núcleo (kernel) inicializa el sistema y crea el entorno para que se ejecuten los procesos y crea unos pocos procesos que a su vez crearán el resto.
- INIT (proceso con pid 1) es el primer proceso de usuario y antecesor del resto de procesos de usuario en el sistema
- El núcleo (kernel) de UNIX interactua con el hardware
- Los procesos interactuan con el núcleo a través de la interfaz de llamadas al sistema

freebsd 4.9

```
USER PID PPID PGID   SESS  JOBC STAT  TT  COMMAND
root   0    0    0 c0326e60    0 DLs   ??  (swapper)
root   1    0    1 c08058c0    0 ILs   ??  /sbin/init --
root   2    0    0 c0326e60    0 DL    ??  (taskqueue)
root   3    0    0 c0326e60    0 DL    ??  (pagedaemon)
root   4    0    0 c0326e60    0 DL    ??  (vmdaemon)
root   5    0    0 c0326e60    0 DL    ??  (bufdaemon)
root   6    0    0 c0326e60    0 DL    ??  (syncer)
root   7    0    0 c0326e60    0 DL    ??  (vnlru)
root  90    1   90 c08509c0    0 Ss    ??  /sbin/natd -n ed0
root 107    1  107 c085cd80    0 Is    ??  /usr/sbin/syslogd
root 112    1  112 c0874500    0 Is    ??  mountd -r
root 115    1  115 c0874600    0 Is    ??  nfsd: master (nfs
root 117  115  115 c0874600    0 I     ??  nfsd: server (nfs
```

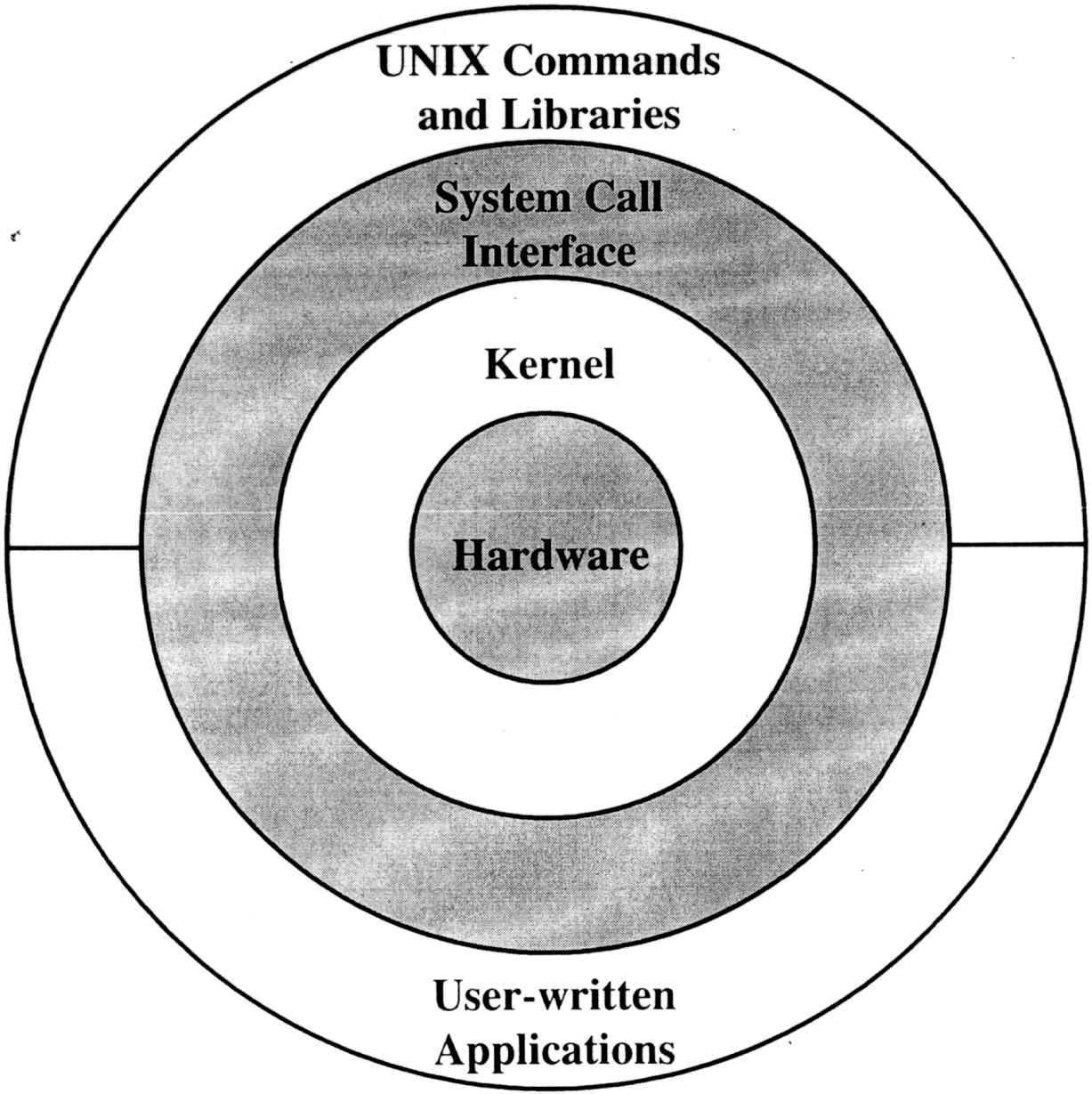
linux 2.4

```
F S UID   PID PPID  C PRI  NI ADDR SZ WCHAN  TTY  CMD
4 S   0     1    0  0  68   0 -   373 select ?    init
1 S   0     2    1  0  69   0 -     0 contex ?    keventd
1 S   0     3    1  0  79  19 -     0 ksofti ?    ksoftirqd
1 S   0     4    1  0  69   0 -     0 kswapd ?    kswapd
1 S   0     5    1  0  69   0 -     0 bdflus ?    bdflush
1 S   0     6    1  0  69   0 -     0 kupdat ?    kupdated
4 S   0   229    1  0  67  -4 -   369 select ?    udevd
1 S   0   375    1  0  69   0 -     0 down_i ?    knodemgrd
1 S   0   492    1  0  69   0 -     0 ?      ?    khubd
1 S   0  1571    1  0  69   0 -   561 select ?    syslogd
5 S   0  1574    1  0  69   0 -   547 syslog ?    klogd
1 S   0  1592    1  0  69   0 -   637 select ?    dirmngr
5 S   0  1604    1  0  69   0 -   555 select ?    inetd
```

solaris 7 sparc

F	S	UID	PID	PPID	C	PRI	NI	SZ	TTY	CMD
19	T	0	0	0	0	0	SY	0	?	sched
8	S	0	1	0	0	41	20	98	?	init
19	S	0	2	0	0	0	SY	0	?	pageout
19	S	0	3	0	0	0	SY	0	?	fsflush
8	S	0	282	279	0	40	20	2115	?	Xsun
8	S	0	123	1	0	41	20	278	?	rpcbind
8	S	0	262	1	0	41	20	212	?	sac
8	S	0	47	1	0	45	20	162	?	devfseve
8	S	0	49	1	0	57	20	288	?	devfsadm
8	S	0	183	1	0	41	20	313	?	automoun
8	S	0	174	1	0	40	20	230	?	lockd
8	S	0	197	1	0	41	20	444	?	syslogd
8	S	0	182	1	0	41	20	3071	?	in.named
8	S	0	215	1	0	41	20	387	?	lpsched
8	S	0	198	1	0	51	20	227	?	cron
8	S	0	179	1	0	41	20	224	?	inetd
8	S	0	283	279	0	46	20	627	?	dtlogin

estructura de un sistema UNIX



En UNIX son necesarios dos modos de ejecución

- **modo usuario** se ejecuta el código de usuario
- **modo kernel** se ejecutan las funciones del kernel
  1. **Llamadas al sistema:** Los procesos de usuario solicitan servicios explicitamente a través de la interfaz de llamadas al sistema.
  2. **Excepciones:** Situaciones excepcionales (división por 0, errores de direccionamiento..) causan excepciones hardware que requieren intervención del kernel.
  3. **Interrupciones:** Los dispositivos periféricos interrumpen para notificar al kernel de diversos sucesos (terminación de e/s, cambio de estado..)
- Algunas instrucciones hardware solo pueden ser ejecutadas en modo kernel.

En un sistema UNIX tradicional un proceso viene definido por

- **espacio de direcciones:** Conjunto de direcciones de memoria que el proceso puede referenciar.
- **punto de control del proceso** que indica cual es la siguiente instrucción a ejecutar utilizando un registro hardware que se llama C.P.

En un sistema UNIX moderno puede haber varios puntos de control (*threads*).
Los procesos manejan direcciones **virtuales** de memoria. Una parte de este espacio corresponde al código y los datos del kernel. Se llama *system space* o *kernel space*
El *system space* solo puede ser accedido en modo kernel
El kernel mantiene

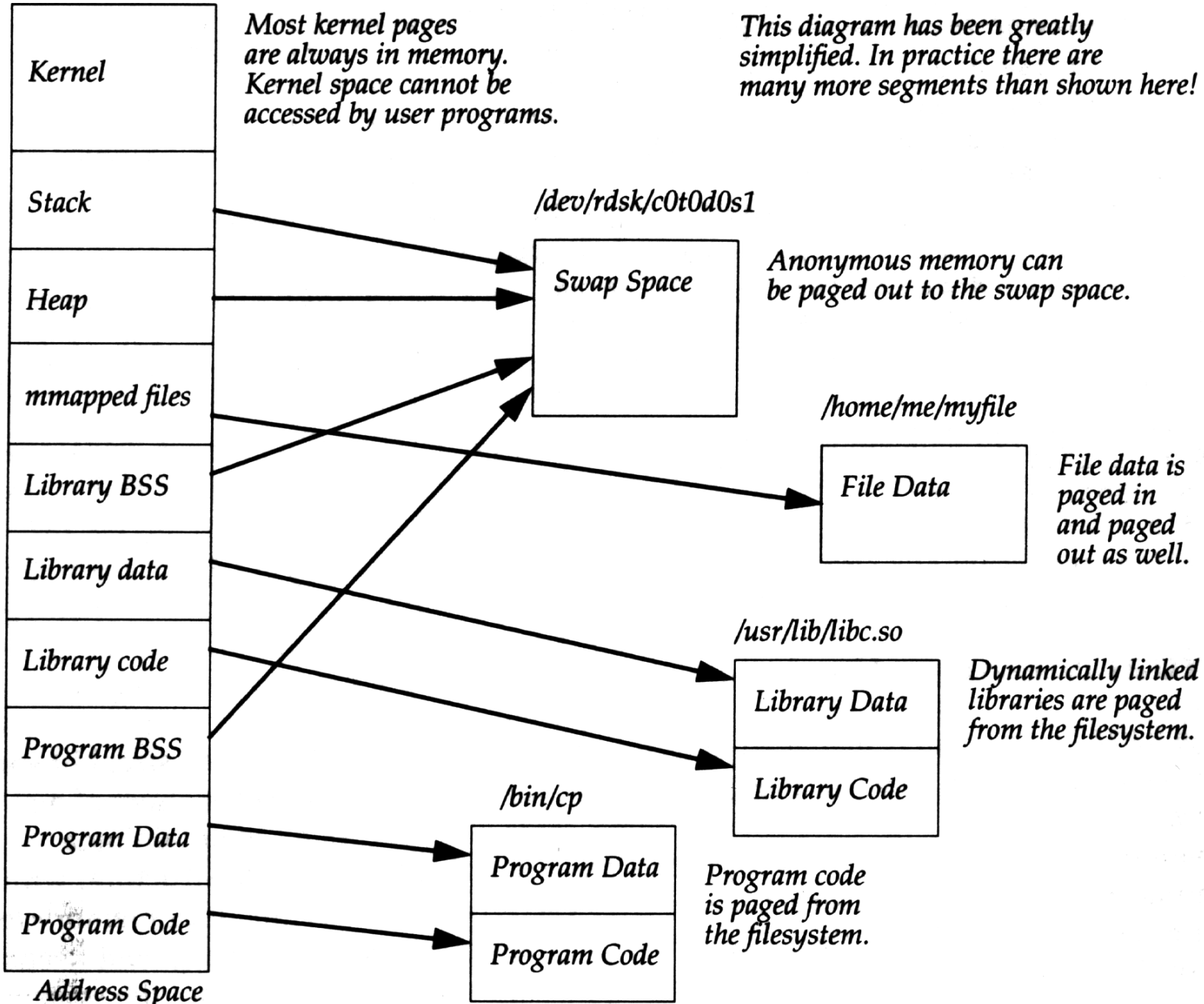
- estructuras de datos globales
- estructuras de datos específicas de cada proceso

El espacio de direcciones del proceso actual es accesible directamente pues los registros de la MMU tienen la información necesaria.

Mapa de memoria en linux

0xc0000000	the invisible kernel
	initial stack
	room for stack grouth
0x60000000	shared libraries
brk	unused
	malloc memory
`end_data`	uninitialized data
`end_code`	initialized data
0x00000000	text

Mapa de memoria en solaris

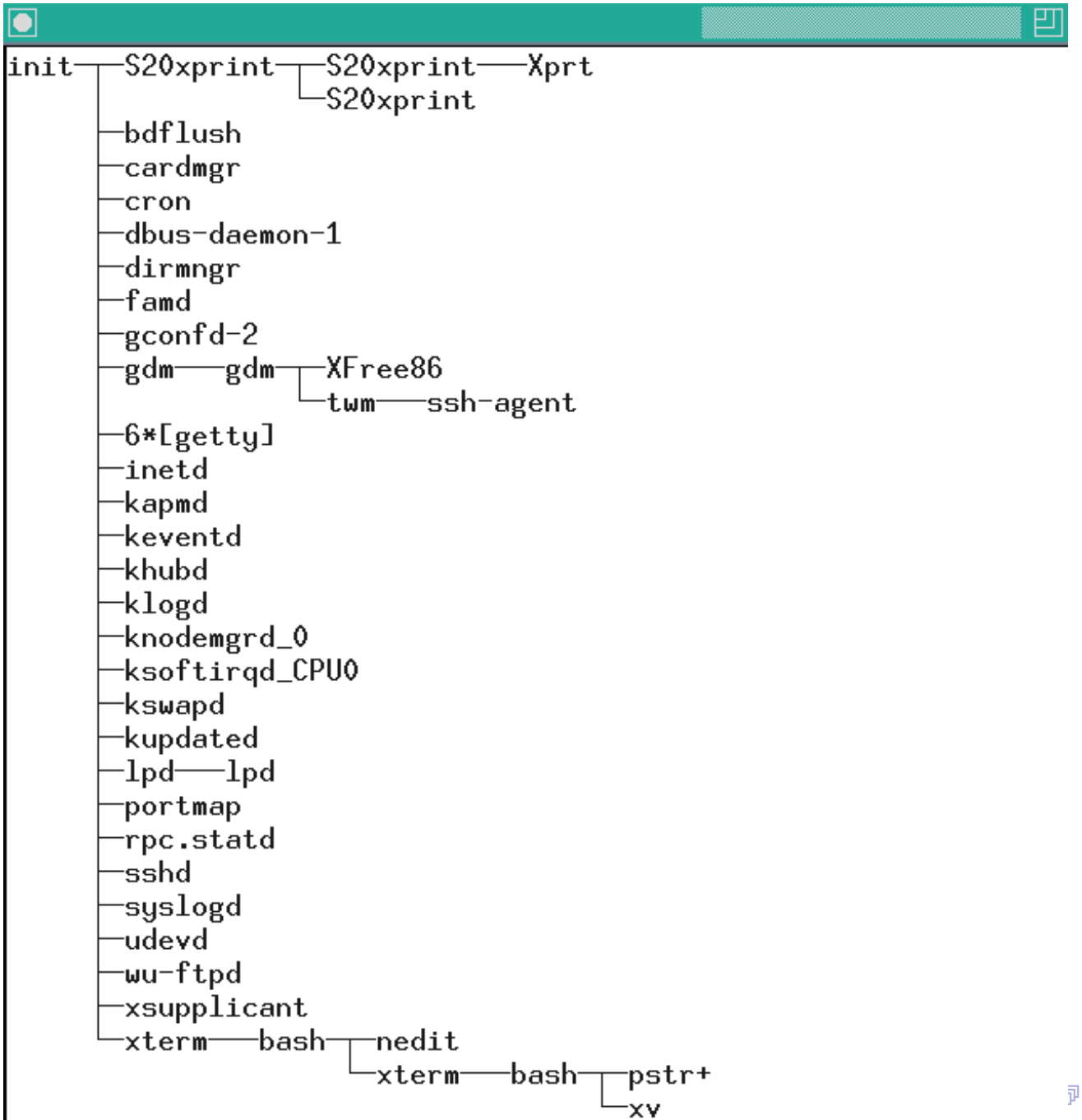


- kernel de unix reentrante
- varios procesos pueden estar ejecutando simultáneamente distintas funciones del kernel
- varios procesos pueden estar ejecutando simultáneamente la misma funcion del kernel
  - **código del kernel** es de solo lectura
  - **datos** (variables globales) **del kernel** protegidos de accesos cocurrentes
  - cada proceso tiene su propia **pila del kernel**

- proceso: *instancia de un programa en ejecución*
- proceso: *entidad que ejecuta un programa y proporciona un entorno de ejecición para él; en concreto un espacio de direcciones y uno (o varios) puntos de control*
- un proceso tiene un tiempo de vida definido
  - se crea mediante la llamada *fork()* (o *vfork()*)
  - termina mediante *exit()*
  - puede ejecutar un programa mediante alguna de las llamadas de la familia *exec()*

- todo proceso tiene un proceso padre
- un proceso puede tener varios procesos hijos
- estructura jerárquica en forma de arbol con el proceso *init* como tronco
- si un proceso termina antes que sus procesos hijos estos pasan a ser heredados por *init*

Ejemplo de arbol de procesos

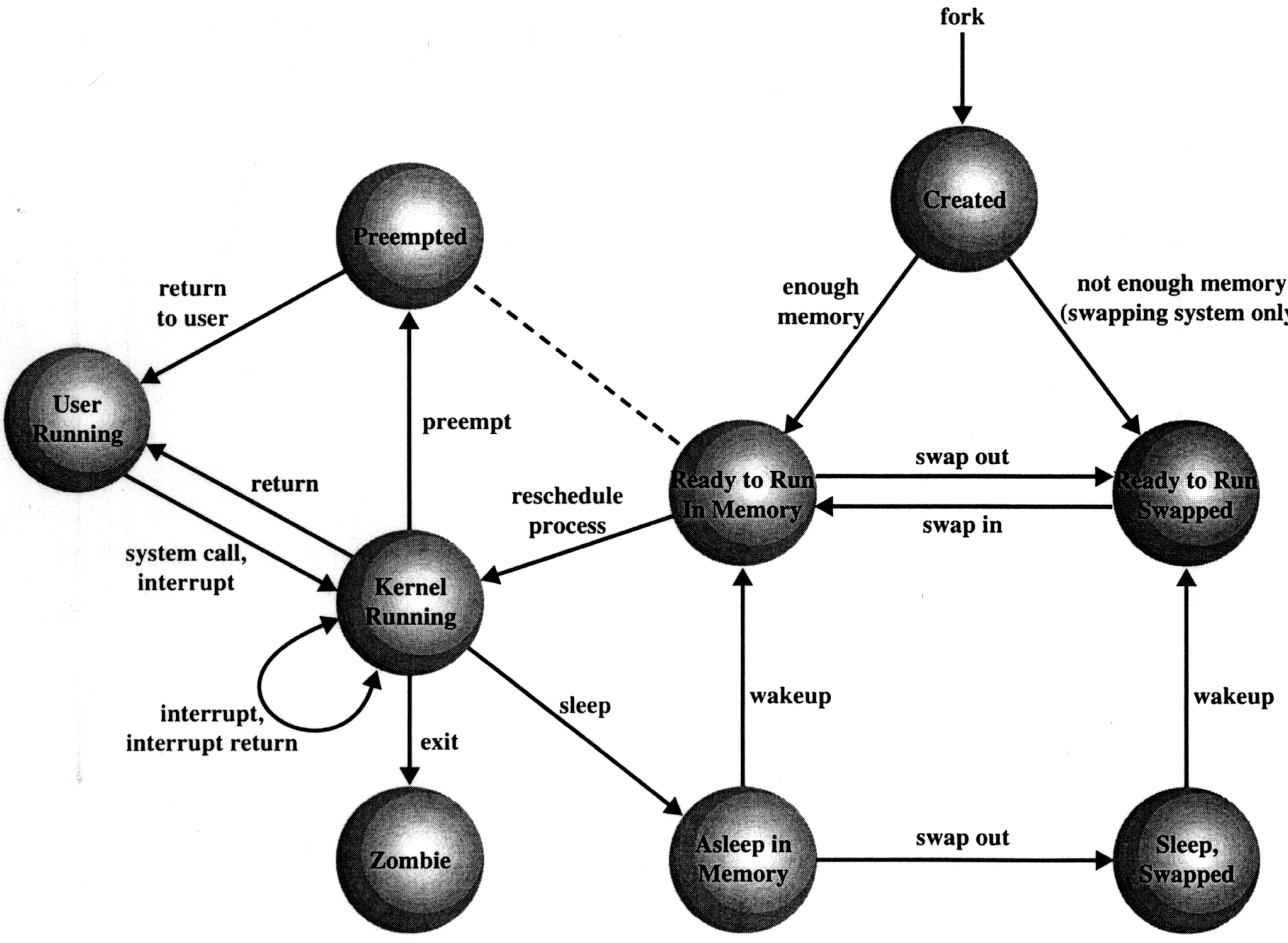


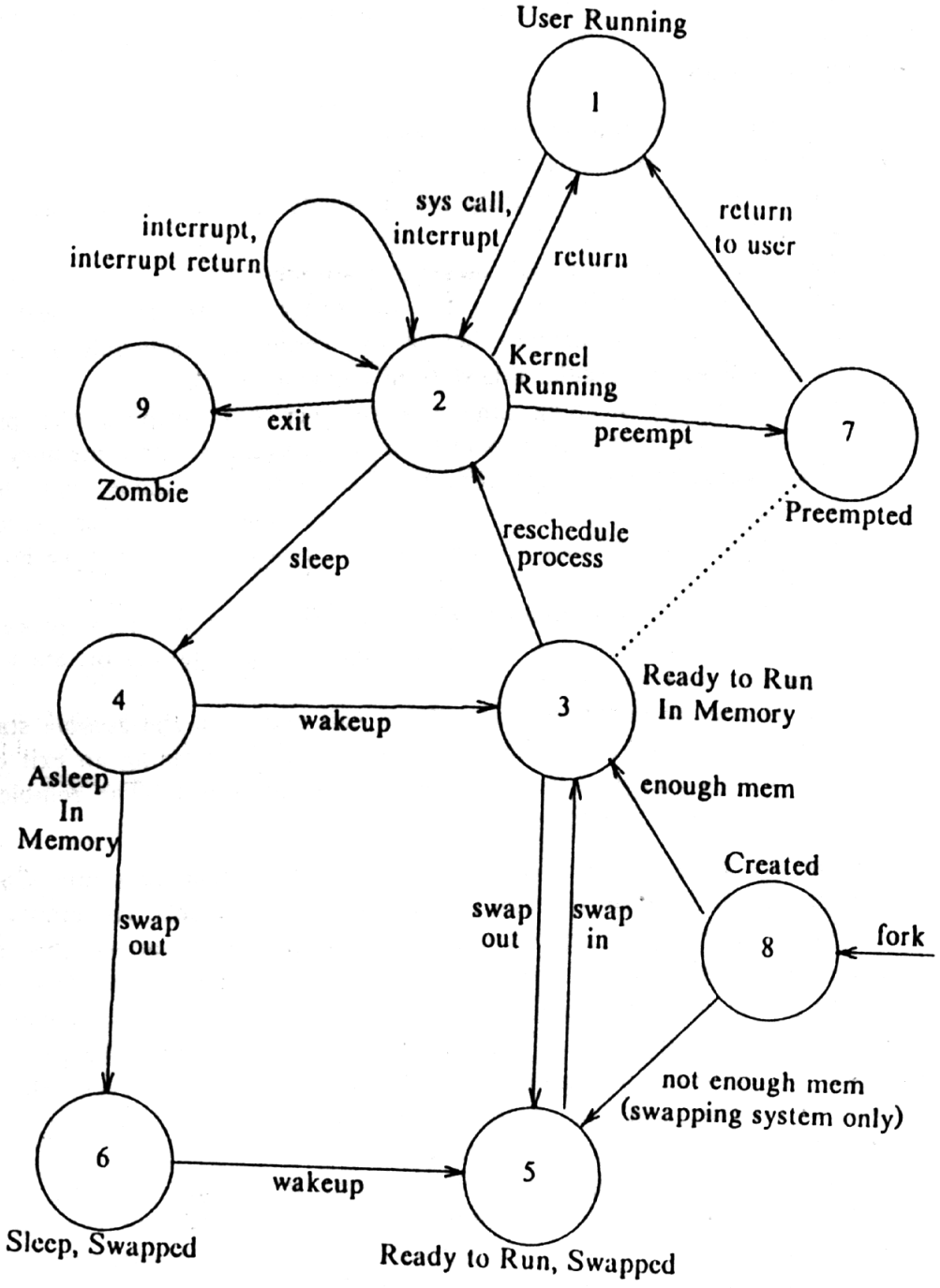
Estados de un procesos en SystemV R2

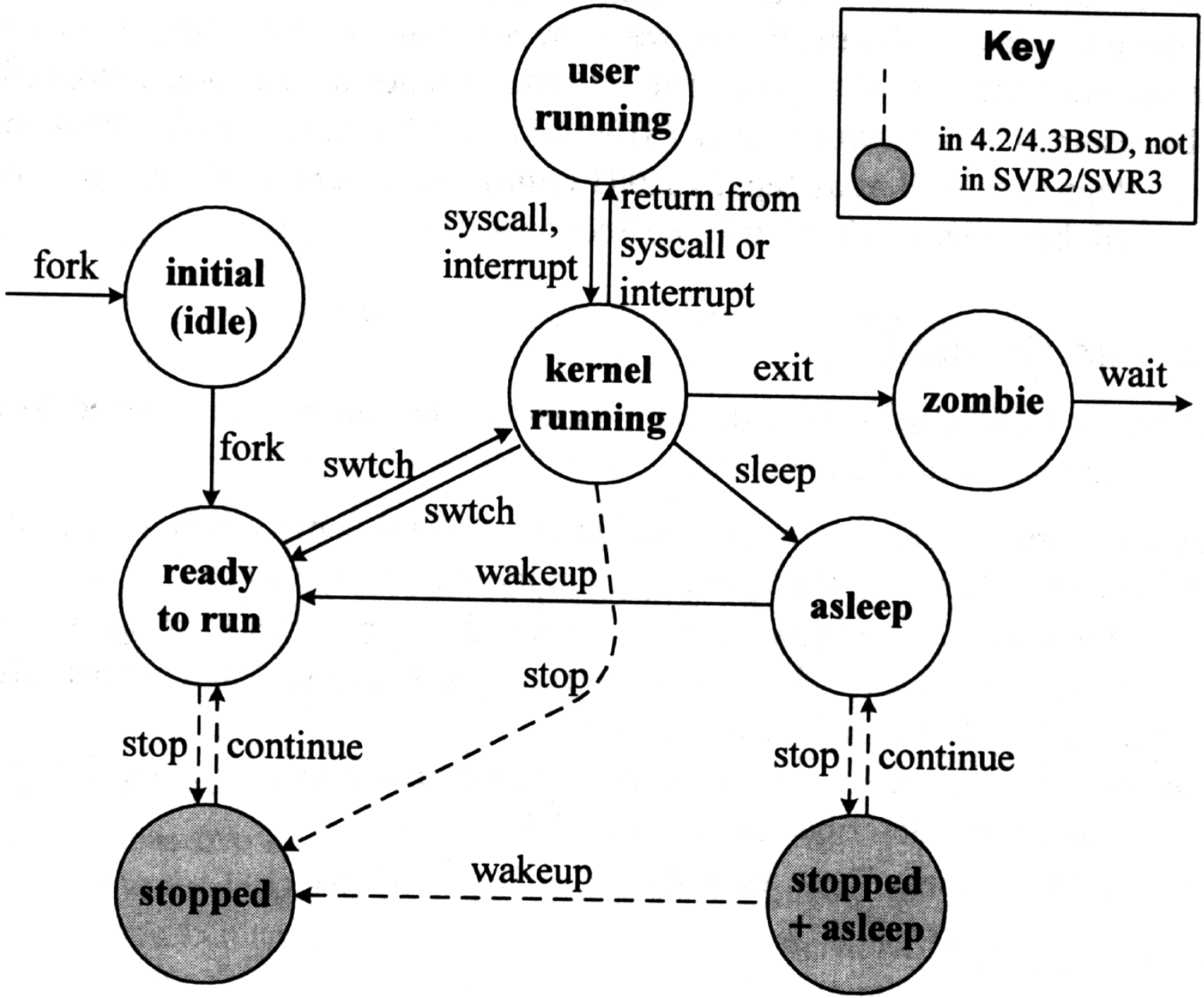
- **inicial** (*idle*): El proceso está siendo creado pero todavía no está listo para ejecición
- **listo** (*runnable, ready to run*)
- **en espera** (*blocked, asleep*). Tanto en este estado, como en el anterior, el proceso puede estar en la memoria principal o en el intercambio (*swapped*)
- **ejecución modo usuario** (*user running*)
- **ejecución modo kernel** (*kernel running*)
- **zombie**: El proceso ha terminado pero su proceso padre no ha hecho *wait()*, con lo que no se ha vaciado su entrada en la tabla de procesos y para el sistema el proceso sigue existiendo.

- A partir de 4.2BSD hay un nuevo estado: **parado** (*stopped*)
- Puede ser parado en espera o parado listo
- Se llega al estado de parado al recibir una de las siguientes señales
  - **SIGSTOP**
  - **SIGTSTP** ctrl-Z
  - **SIGTTIN**
  - **SIGTTOU**
- Se sale de él mediante la señal SIGCONT

- La ejecución de un proceso comienza en modo kernel
- Las transiciones de ejecución a espera son desde ejecución en modo kernel
- Las transiciones de ejecución a listo son desde ejecición en modo kernel
- Un proceso termina desde ejecución en modo kernel
- Cuando un proceso termina queda en estado *zombie* hasta que su padre hace una de las llamadas *wait*


fork
Created
Preempted
return
to user
enough
memory
not enough memory
(swapping system only
User
Running
preempt
swap out
Ready to Run
In Memory
Ready to Run
Swapped
return
reschedule
process
swap in
system call,
interrupt
Kernel
Running
wakeup
wakeup
interrupt,
interrupt return
exit
sleep
Zombie
Asleep in
Memory
swap out
Sleep,
Swapped


User Running
1
interrupt,
interrupt return
sys call,
interrupt
return
return
to user
Kernel
Running
2
9
exit
Zombie
preempt
7
Preempted
sleep
reschedule
process
4
wakeup
3
Ready to Run
In Memory
Asleep
In
Memory
enough mem
Created
swap
out
swap
out
swap
in
8
fork
not enough mem
(swapping system only)
6
wakeup
5
Sleep, Swapped
Ready to Run, Swapped


Key
in 4.2/4.3BSD, not in SVR2/SVR3
user running
syscall, interrupt
return from syscall or interrupt
fork
initial (idle)
kernel running
exit
zombie
wait
fork
swtch
swtch
sleep
ready to run
wakeup
asleep
stop
stop
continue
stop
continue
stopped
wakeup
stopped + asleep

Un proceso se ejecuta dentro de un determinado contexto que contiene la información necesaria para ejecutar dicho proceso. Dicho contexto está formado por:

- **Espacio de direcciones de usuario**. Usualmente formado por texto (código), datos, pila, regiones de memoria compartida ...
- **Información de control**.
  - estructura `proc`
  - `u_area`
  - pila del kernel
  - mapas de traslación de direcciones
- **credenciales**
- **variables de entorno**
- **contexto hardware** Los contenidos de los registros hardware (PC, PSW, SP, registros de FPU y MMU ...). Cuando hay un cambio de contexto se guardan en una parte de la `u_area` llamada PCB (Process Control Block)

estructura `proc`

- El kernel mantiene un array de estructuras `proc` llamado *proccess table*
- Está en el espacio de datos del kernel
- La estructura `proc` de un proceso es siempre visible para el kernel, incluso cuando el proceso no está en CPU
- Contiene la información del proceso que es necesaria en todo momento

estructura `proc`

- identificación del proceso
- localización de la `u_area` (mapa de direcciones)
- estado del proceso
- punteros para cola de planificación, espera ...
- prioridad e información relacionada
- *sleep channel*
- información sobre las señales (máscaras)
- información manejo de memoria
- punteros mara mantener estructura en lista de activas, espera, zombies ...
- punteros para cola hash basada en PID
- información de jerarquía
- flags

ejemplo estructura `proc` en SCO system V R3

```
typedef struct  proc {
        char    p_stat;                 /* status of process */
        char    p_pri;                  /* priority  */
        char    p_cpu;                  /* cpu usage for scheduling */
        char    p_nice;                 /* nice for cpu usage */
        uint    p_flag;                 /* flags defined below */
        ushort  p_uid;                  /* real user id */
        ushort  p_suid;                 /* saved (effective) uid from exec */
        pid_t   p_sid;                  /* POSIX session id number */
        short   p_pgrp;                 /* name of process group leader */
        short   p_pid;                  /* unique process id*/
        short   p_ppid;                 /* process id of parent*/
        ushort  p_sgid;                 /* saved (effective) gid from exec */
        sigset_t        p_sig;          /* signals pending to this process */
        struct  proc    *p_flink;       /* forward link */
        struct  proc    *p_blink;       /* backward link */
        union {                         /* wait addr for sleeping processes */
                caddr_t p_cad;          /* Union is for XENIX compatibility */
                int     p_int;
        } p_unw;
              /* current signal */
```

ejemplo estructura proc en SCO system V R3

```
#define p_wchan p_unw.p_cad             /* Map MP name to old UNIX name */
#define p_arg   p_unw.p_int             /* Map MP name to old UNIX name */
        struct  proc    *p_parent;      /* ptr to parent process */
        struct  proc    *p_child;       /* ptr to first child process */
        struct  proc    *p_sibling;     /* ptr to next sibling proc on chain */
        int     p_clktim;               /* time to alarm clock signal */
        uint    p_size;                 /* size of swappable image in pages */
        time_t  p_utime;                /* user time, this process */
        time_t  p_stime;                /* system time, this process */
        struct  proc *p_mlink;          /* linked list of processes sleeping
                                         * on memwant or swapwant
                                         */
        ushort  p_usize;                /* size of u-block (*4096 bytes) */
        ushort  p_res1;                 /* Pad because p_usize is replacing
                                         * a paddr_t (i.e., long) field, and
                                         * it is only a short.
                                         */

        caddr_t p_ldt;                  /* address of ldt */
        long    p_res2;                 /* Pad because a 'pde_t *' field was
                                         * removed here.  Its function is
                                         * replaced by p_ubptbl[MAXUSIZE].
                                         */

        preg_t  *p_region;              /* process regions */
        ushort  p_mpgneed;              /* number of memory pages needed in
                                         * memwant.
                                         */
        char    p_time;                 /* resident time for scheduling */
        unchar  p_cursig;
```

ejemplo estructura proc en SCO system V R3

```
        short   p_epid;                 /* effective pid; normally same as
                                         * p_pid; for servers, the system that
                                         * sent the msg
                                         */
        sysid_t p_sysid;                /* normally same as sysid; for servers,
                                         * the system that sent the msg
                                         */
        struct  rcvd  *p_minwd;         /* server msg arrived on this queue */
        struct  proc  *p_rlink;         /* linked list for server */
        int     p_trlock;
        struct  inode *p_trace;         /* pointer to /proc inode */
        long    p_sigmask;              /* tracing signal mask for /proc */
        sigset_t        p_hold;         /* hold signal bit mask */
        sigset_t        p_chold;        /* deferred signal bit mask; sigset(2)
                                         * turns these bits on while signal(2)
                                         * does not.
                                         */
        short   p_xstat;                /* exit status for wait */
        short   p_slot;                 /* proc slot we're occupying */
        struct  v86dat *p_v86;          /* pointer to v86 structure */
        dbd_t   p_ubdbd;                /* DBD for ublock when swapped out */
        ushort  p_whystop;              /* Reason for process stop */
        ushort  p_whatstop;             /* More detailed reason */
        pde_t   p_ubptbl[MAXUSIZE];     /* u-block page table entries */
        struct  sd *p_sdp;              /* pointer to XENIX shared data */
        int     p_sigflags[MAXSIG];     /* modify signal behavior (POSIX) */
} proc_t;
```

ejemplo estructura proc en SunOs 4.1

```
struct  proc {
        struct  proc *p_link;   /* linked list of running processes */
        struct  proc *p_rlink;
        struct  proc *p_nxt;    /* linked list of allocated proc slots */
        struct  proc **p_prev;  /* also zombies, and free procs */
        struct  as *p_as;       /* address space description */
        struct seguser *p_segu; /* "u" segment */
        /*
         * The next 2 fields are derivable from p_segu, but are
         * useful for fast access to these places.
         * In the LWP future, there will be multiple p_stack's.
         */
        caddr_t p_stack;        /* kernel stack top for this process */
        struct user *p_uarea;   /* u area for this process */
        char    p_usrpri;       /* user-priority based on p_cpu and p_nice */
        char    p_pri;          /* priority */
        char    p_cpu;          /* (decayed) cpu usage solely for scheduling */
        char    p_stat;
        char    p_time;         /* seconds resident (for scheduling) */
        char    p_nice;         /* nice for cpu usage */
        char    p_slptime;      /* seconds since last block (sleep) */
```

ejemplo estructura proc en SunOs 4.1

```
char    p_cursig;
int     p_sig;          /* signals pending to this process */
int     p_sigmask;      /* current signal mask */
int     p_sigignore;    /* signals being ignored */
int     p_sigcatch;     /* signals being caught by user */
int     p_flag;
uid_t   p_uid;          /* user id, used to direct tty signals */
uid_t   p_suid;         /* saved (effective) user id from exec */
gid_t   p_sgid;         /* saved (effective) group id from exec */
short   p_pgrp;         /* name of process group leader */
short   p_pid;          /* unique process id */
short   p_ppid;         /* process id of parent */
u_short p_xstat;        /* Exit status for wait */
short   p_cpticks;      /* ticks of cpu time, used for p_pctcpu */
struct  ucred *p_cred;  /* Process credentials */
struct  rusage *p_ru;   /* mbuf holding exit information */
int     p_tsize;        /* size of text (clicks) */
int     p_dsize;        /* size of data space (clicks) */
int     p_ssize;        /* copy of stack size (clicks) */
int     p_rssize;       /* current resident set size in clicks */
int     p_maxrss;       /* copy of u.u_limit[MAXRSS] */
int     p_swrss;        /* resident set size before last swap */
caddr_t p_wchan;        /* event process is awaiting */
long    p_pctcpu;       /* (decayed) %cpu for this process */
```

ejemplo estructura proc en SunOs 4.1

```
        struct  proc *p_pptr;   /* pointer to process structure of parent */
        struct  proc *p_cptr;   /* pointer to youngest living child */
        struct  proc *p_osptr;  /* pointer to older sibling processes */
        struct  proc *p_ysptr;  /* pointer to younger siblings */
        struct  proc *p_tptr;   /* pointer to process structure of tracer */
        struct  itimerval p_realtimer;
        struct  sess *p_sessp;  /* pointer to session info */
        struct  proc *p_pglnk;  /* list of pgrps in same hash bucket */
        short   p_idhash;       /* hashed based on p_pid for kill+exit+... */
        short   p_swlocks;      /* number of swap vnode locks held */
        struct aiodone *p_aio_forw; /* (front)list of completed asynch IO's */
        struct aiodone *p_aio_back; /* (rear)list of completed asynch IO's */
        int     p_aio_count;    /* number of pending asynch IO's */
        int     p_threadcnt;    /* ref count of number of threads using proc */
#ifdef  sun386
        struct  v86dat *p_v86;  /* pointer to v86 structure */
#endif  sun386
#ifdef  sparc
/*
 * Actually, these are only used for MULTIPROCESSOR
 * systems, but we want the proc structure to be the
 * same size on all 4.1.1psrA SPARC systems.
 */
        int     p_cpuid;        /* processor this process is running on */
        int     p_pam;          /* processor affinity mask */
#endif  sparc
};
```

ejemplo de estructura proc en System V R4

```
typedef struct proc {
/*
 * Fields requiring no explicit locking
 */
clock_t p_lbolt; /* Time of last tick processing */
id_t p_cid; /* scheduling class id */
struct vnode *p_exec; /* pointer to a.out vnode */
struct as *p_as; /* process address space pointer */
#ifdef XENIX_MERGE
struct sd *p_sdp; /* pointer to XENIX shared data */
#endif
o_uid_t p_uid; /* for binary compat. - real user id */
kmutex_t p_lock; /* proc struct's mutex lock */
kmutex_t p_crlock; /* lock for p_cred */
struct cred *p_cred; /* process credentials */
/*
 * Fields protected by pidlock
 */
int p_swapcnt; /* number of swapped out lwps */
char p_stat; /* status of process */
char p_wcode; /* current wait code */
int  p_wdata; /* current wait return value */
pid_t p_ppid; /* process id of parent */
struct proc *p_link; /* forward link */
struct proc *p_parent; /* ptr to parent process */
struct proc *p_child; /* ptr to first child process */
struct proc *p_sibling; /* ptr to next sibling proc on chain */
struct proc  *p_next; /* active chain link */
struct proc  *p_nextofkin; /* gets accounting info at exit */
```

ejemplo de estructura proc en System V R4

```
struct proc  *p_orphan;
struct proc  *p_nextorph;
struct proc *p_pglink; /* process group hash chain link */
struct sess *p_sessp; /* session information */
struct pid  *p_pidp; /* process ID info */
struct pid  *p_pgidp; /* process group ID info */
/*
 * Fields protected by p_lock
 */
char p_cpu; /* cpu usage for scheduling */
char p_brkflag; /* serialize brk(2) */
kcondvar_t p_brkflag_cv;
kcondvar_t p_cv; /* proc struct's condition variable */
kcondvar_t p_flag_cv;
kcondvar_t p_lwpexit; /* waiting for some lwp to exit */
kcondvar_t p_holdlwps; /* process is waiting for its lwps */
/* to to be held.  */
u_int p_flag; /* protected while set. */
/* flags defined below */
clock_t p_utime; /* user time, this process */
clock_t p_stime; /* system time, this process */
clock_t p_cutime; /* sum of children's user time */
clock_t p_cstime; /* sum of children's system time */
caddr_t *p_segacct; /* segment accounting info */
caddr_t p_brkbase; /* base address of heap */
u_int p_brksize; /* heap size in bytes */
```

ejemplo de estructura proc en System V R4

```
/*
 * Per process signal stuff.
 */
k_sigset_t p_sig; /* signals pending to this process */
k_sigset_t p_ignore; /* ignore when generated */
k_sigset_t p_siginfo; /* gets signal info with signal */
struct sigqueue *p_sigqueue; /* queued siginfo structures */
struct sigqhdr *p_sigqhdr; /* hdr to sigqueue structure pool */
u_char p_stopsig; /* jobcontrol stop signal */
/*
 * Per process lwp and kernel thread stuff
 */
int p_lwptotal; /* total number of lwps created */
int  p_lwpcnt; /* number of lwps in this process */
int p_lwprcnt; /* number of not stopped lwps */
int p_lwpblocked; /* number of blocked lwps. kept  */
/*    consistent by sched_lock() */
int p_zombcnt; /* number of zombie LWPs */
kthread_t *p_tlist; /* circular list of threads */
kthread_t *p_zomblist; /* circular list of zombie LWPs */
/*
 * XXX Not sure what locks are needed here.
 */
k_sigset_t p_sigmask; /* mask of traced signals (/proc) */
k_fltset_t p_fltmask; /* mask of traced faults (/proc) */
struct vnode *p_trace; /* pointer to primary /proc vnode */
struct vnode *p_plist; /* list of /proc vnodes for process */
```

ejemplo de estructura proc en System V R4

```
struct proc  *p_rlink; /* linked list for server */
kcondvar_t p_srwchan_cv;
int  p_pri; /* process priority */
u_int p_stksize; /* process stack size in bytes */
/*
 * Microstate accounting, resource usage, and real-time profiling
 */
hrtime_t p_mstart; /* hi-res process start time */
hrtime_t p_mterm; /* hi-res process termination time */
hrtime_t p_mlreal; /* elapsed time sum over defunct lwps */
hrtime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
struct lrusage p_ru; /* lrusage sum over defunct lwps */
struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
int p_rprof_timerid; /* interval timer's timeout id */
u_int p_defunct; /* number of defunct lwps */
/*
 * profiling. A lock is used in the event of multiple lwp's
 * using the same profiling base/size.
 */
kmutex_t p_pflock; /* protects user pr_base in lwp */

/*
 * The user structure
 */
struct user p_user; /* (see sys/user.h) */

/*
 * C2 Security  (C2_AUDIT)
 */
caddr_t p_audit_data; /* per process audit structure */
} proc_t;
```

u_area

- está en el espacio de usuario: solo está accesible cuando el proceso está en CPU
- siempre en la misma dirección virtual (el cambio de contexto realiza esta traslación)
- contiene información que es necesaria solamente cuando el proceso está en CPU

u_area

- PCB
- puntero a la estructuta `proc`
- argumentos y valores devueltos por la llamada al sistema
- información de señales: manejadores
- TDFU
- punteros a *vnodos* de directorio raíz, directorio actual y terminal asociada al proceso.
- pila del kernel del proceso

ejemplo de u_area en SCO unix System V R3

```
typedef struct user
{
char u_stack[KSTKSZ]; /* kernel stack */

union u_fps u_fps;

long    u_weitek_reg[WTK_SAVE]; /* bits needed to save weitek state */
/* NOTE: If the WEITEK is actually  */
/* present, only 32 longs will be   */
/* used, but if it is not, the      */
/* emulator will need 33.           */

struct tss386 *u_tss; /* pointer to user TSS */
ushort u_sztss; /* size of tss (including bit map) */

char u_sigfault; /* catch general protection violations
   caused by user modifying his stack
   where the old state info is kept */
char u_usigfailed; /* allows the user to know that he caused
   a general protection violation by
   modifying his register save area used
   when the user was allowed to do his own
   signal processing */

ulong u_sub; /* stack upper bound.
   The address of the first byte of
   the first page of user stack
   allocated so far */
char u_filler1[40]; /* DON'T TOUCH--this is used by
 * conditionally-compiled code in iget.c
 * which checks consistency of inode locking
 * and unlocking.  Name change to follow in
 * a later release.
 */
```

ejemplo de u_area en SCO unix System V R3

```
int u_caddrflt; /* Ptr to function to handle */
/* user space external memory */
/* faults encountered in the */
/* kernel. */
char u_nshmseg; /* Nbr of shared memory */
/* currently attached to the */
/* process. */
  struct rem_ids { /* for exec'ing REMOTE text */
  ushort ux_uid; /* uid of exec'd file */
  ushort ux_gid; /* group of exec'd file */
  ushort ux_mode; /* file mode (set uid, etc. */
  } u_exfile;
char *u_comp; /* pointer to current component */
char *u_nextcp; /* pointer to beginning of next */
/* following for Distributed UNIX */
ushort  u_rflags; /* flags for distripution */
int u_sysabort; /* Debugging: if set, abort syscall */
int u_systrap; /* Are any syscall mask bits set? */
int  u_syscall; /* system call number */
int u_mntindx; /* mount index from sysid */
struct sndd *u_gift; /* gift from message      */
long  u_rcstat;  /* Client cache status flags */
ulong u_userstack;
struct response *u_copymsg; /* copyout unfinished business */
struct msgb *u_copybp; /* copyin premeditated send    */
char  *u_msgend; /* last byte of copymsg + 1    */
/* end of Distributed UNIX */
long u_bsize; /* block size of device */
char  u_psargs[PSARGSZ]; /* arguments from exec */

int u_pgproc; /* use by the MAU driver */
time_t u_ageinterval; /* pageing ageing countdown counter */
label_t u_qsav; /* label variable for quits and */
/* interrupts */
```

ejemplo de u_area en SCO unix System V R3

```
char u_segflg; /* IO flag: 0:user D; 1:system; */
/*          2:user I */
unchar u_error; /* return error code */
ushort u_uid; /* effective user id */
ushort u_gid; /* effective group id */
ushort u_ruid; /* real user id */
ushort u_rgid; /* real group id */
struct lockb u_cilock; /* MPX process u-area synchronization */
struct proc *u_procp; /* pointer to proc structure */
int *u_ap; /* pointer to arglist */
union { /* syscall return values */
struct {
int r_val1;
int r_val2;
}r_reg;
off_t r_off;
time_t r_time;
} u_r;
caddr_t u_base; /* base address for IO */
unsigned u_count; /* bytes remaining for IO */
off_t u_offset; /* offset in file for IO */
short u_fmode; /* file mode for IO */
ushort u_pbsize; /* Bytes in block for IO */
ushort u_pboff; /* offset in block for IO */
dev_t u_pbdev; /* real device for IO */
daddr_t u_rablock; /* read ahead block address */
short u_errcnt; /* syscall error count */
struct inode *u_cdir; /* current directory */
struct inode *u_rdir; /* root directory */
caddr_t u_dirp; /* pathname pointer */
struct direct u_dent; /* current directory entry */
struct inode *u_pdir; /* inode of parent directory */
/* of dirp */
```

ejemplo de u_area en SCO unix System V R3

```
char *u_pofile; /* Ptr to open file flag array.  */
struct inode *u_ttyip; /* inode of controlling tty (streams) */
int u_arg[6]; /* arguments to current system call */

unsigned u_tsize; /* text size (clicks) */
unsigned u_dsize; /* data size (clicks) */
unsigned u_ssize; /* stack size (clicks) */

void (*u_signal[MAXSIG])(); /* disposition of signals */
void (*u_sigreturn)(); /* for cleanup */

time_t u_utime; /* this process user time */
time_t u_stime; /* this process system time */
time_t u_cutime; /* sum of childs' utimes */
time_t u_cstime; /* sum of childs' stimes */

int *u_ar0; /* address of users saved R0 */

/*
 *    The offsets of these elements must be reflected in ttrap.s and misc.s
 */

struct { /* profile arguments */
short *pr_base; /* buffer base */
unsigned pr_size; /* buffer size */
unsigned pr_off; /* pc offset */
unsigned pr_scale; /* pc scaling */
} u_prof;

short  *u_ttyp; /* pointer to pgrp in "tty" struct */
dev_t u_ttyd; /* controlling tty dev */

ulong u_renv; /* runtime environment.    */
/*   for meaning of bits:   */
```

ejemplo de u_area en SCO unix System V R3

```
/*
 * Executable file info.
 */
struct exdata {
struct    inode  *ip;
long      ux_tsize; /* text size    */
long      ux_dsize; /* data size    */
long      ux_bsize; /* bss size     */
long      ux_lsize;   /* lib size     */
long      ux_nshlibs;  /* number of shared libs needed */
short     ux_mag;    /* magic number MUST be here */
long      ux_toffset; /* file offset to raw text      */
long      ux_doffset; /* file offset to raw data      */
long      ux_loffset; /* file offset to lib sctn      */
long      ux_txtorg; /* start addr. of text in mem   */
long      ux_datorg; /* start addr. of data in mem   */
long      ux_entloc; /* entry location               */
ulong   ux_renv; /* runtime environment */
} u_exdata;

long   u_execsz;

char u_comm[PSCOMSIZ];

time_t u_start;
time_t u_ticks;
long u_mem;
long u_ior;
long u_iow;
long u_iosw;
long u_ioch;
char u_acflag;

short u_cmask; /* mask for file creation */
```

ejemplo de u_area en SCO unix System V R3

```
short u_lock; /* process/text locking flags */
/* floating point support variables */
char    u_fpvalid;              /* flag if saved state is valid     */
char    u_weitek;               /* flag if process uses weitek chip */
int     u_fpintgate[2];         /* fp intr gate descriptor image    */

/* i286 emulation variables */
int    *u_callgatep;            /* pointer to call gate in gdt  */
int     u_callgate[2];          /* call gate descriptor image   */
int     u_ldtmodified;          /* if set, LDT was modified     */
ushort u_ldtlimit; /* current size (index) of ldt */

/* Flag single-step of lcall for a system call. */
/* The signal is delivered after the system call*/
char    u_debugpend;            /* SIGTRAP pending for this proc */

/* debug registers, accessible by ptrace(2) but monitored by kernel */
char    u_debugon;              /* Debug registers in use, set by kernel */
int     u_debugreg[8];
long u_entrymask[SYSMASKLEN]; /* syscall stop-on-entry mask */
long u_exitmask[SYSMASKLEN]; /* syscall stop-on-exit mask */
/*
 * New for POSIX
 */
sigset_t u_sigmask[MAXSIG];      /* signals to be blocked */
sigset_t u_oldmask; /* mask saved before sigsuspend() */
gid_t *u_groups; /* Ptr to 0 terminated */
/*  supplementary group array */

struct file *u_ofile[1]; /* Start of array of pointers */
/* to file table entries for */
/* open files. */
/* NOTHING CAN GO BELOW HERE!!!!*/
} user_t;
```

ejemplo de u_area en SunOs 4.1

```
struct user {
struct pcb u_pcb;
struct proc *u_procp; /* pointer to proc structure */
int *u_ar0; /* address of users saved R0 */
char u_comm[MAXCOMLEN + 1];

/* syscall parameters, results and catches */
int u_arg[8]; /* arguments to current system call */
int *u_ap; /* pointer to arglist */
label_t u_qsave; /* for non-local gotos on interrupts */
union { /* syscall return values */
struct {
int R_val1;
int R_val2;
} u_rv;
off_t r_off;
time_t r_time;
} u_r;

char u_error; /* return error code */
char u_eosys; /* special action on end of syscall */

label_t u_ssave; /* label for swapping/forking */

/* 1.3 - signal management */
void (*u_signal[NSIG])(); /* disposition of signals */
int u_sigmask[NSIG]; /* signals to be blocked */
int u_sigonstack; /* signals to take on sigstack */
int u_sigintr; /* signals that interrupt syscalls */
int u_sigreset; /* signals that reset the handler when taken */
int u_oldmask; /* saved mask from before sigpause */
int u_code; /* ``code'' to trap */
char *u_addr; /* ``addr'' to trap */
struct sigstack u_sigstack; /* sp & on stack state variable */
```

ejemplo de u_area en SunOs 4.1

```
/* 1.4 - descriptor management */
/*
 * As long as the highest numbered descriptor that the process
 * has ever used is < NOFILE_IN_U, the u_ofile and u_pofile arrays
 * are stored locally in the u_ofile_arr and u_pofile_arr fields.
 * Once this threshold is exceeded, the arrays are kept in dynamically
 * allocated space.  By comparing u_ofile to u_ofile_arr, one can
 * tell which situation currently obtains.  Note that u_lastfile
 * does _not_ convey this information, as it can drop back down
 * when files are closed.
 */
struct file **u_ofile; /* file structures for open files */
char *u_pofile; /* per-process flags of open files */
struct file *u_ofile_arr[NOFILE_IN_U];
char u_pofile_arr[NOFILE_IN_U];
int u_lastfile; /* high-water mark of u_ofile */
struct ucwd  *u_cwd; /* ascii current directory */
struct vnode *u_cdir; /* current directory */
struct vnode *u_rdir; /* root directory of current process */
short u_cmask; /* mask for file creation */
* 1.5 - timing and statistics */
struct rusage u_ru; /* stats for this proc */
struct rusage u_cru; /* sum of stats for reaped children */
struct itimerval u_timer[3];
int u_XXX[3];
long u_ioch; /* characters read/written */
struct timeval u_start;
short u_acflag;

struct uprof { /* profile arguments */
short *pr_base; /* buffer base */
u_int  pr_size; /* buffer size */
u_int  pr_off; /* pc offset */
u_int  pr_scale; /* pc scaling */
```

ejemplo de u_area en SunOs 4.1

```
/* 1.6 - resource controls */
struct rlimit u_rlimit[RLIM_NLIMITS];

/* BEGIN TRASH */
union {
struct exec Ux_A; /* header of executable file */
char ux_shell[SHSIZE]; /* #! and name of interpreter */
#ifdef sun386
struct exec UX_C; /* COFF file header */
#endif
} u_exdata;
#ifdef sun386
/*
 * The virtual address of the text and data is needed to exec
 * coff files.  Unfortunately, they won't fit into Ux_A above.
 */
u_int u_textvaddr; /* virtual address of text segment */
u_int u_datavaddr; /* virtual address of data segment */
u_int u_bssvaddr; /* virtual address of bss segment */

int u_lofault; /* catch faults in locore.s */
#endif sun
/* END TRASH */
};
```

ejemplo de u_area en System V R4

```
typedef struct user {
/*
 * Fields that require no explicit locking
 */
int u_execid;
long u_execsz;
uint u_tsize; /* text size (clicks) */
uint u_dsize; /* data size (clicks) */
time_t u_start;
clock_t u_ticks;
kcondvar_t u_cv; /* user structure's condition var */
/*
 * Executable file info.
 */
struct exdata u_exdata;
auxv_t u_auxv[NUM_AUX_VECTORS]; /* aux vector from exec */
char u_psargs[PSARGSZ]; /* arguments from exec */
char u_comm[MAXCOMLEN + 1];
/*
 * Initial values of arguments to main(), for /proc
 */
int u_argc;
char **u_argv;
char **u_envp;
/*
 * Updates to these fields are atomic
 */
struct vnode *u_cdir; /* current directory */
struct vnode *u_rdir; /* root directory */
struct vnode *u_ttyvp; /* vnode of controlling tty */
mode_t u_cmask; /* mask for file creation */
long u_mem;
char u_systrap; /* /proc: any syscall mask bits set? */
```

ejemplo de u_area en System V R4

```
/*
 * Flag to indicate there is a signal or event pending to
 * the current process.  Used to make a quick check just
 * prior to return from kernel to user mode.
 */
char u_sigevpend;

/*
 * WARNING: the definitions for u_ttyp and
 * u_ttyd will be deleted at the next major
 * release following SVR4.
 */

o_pid_t *u_ttyp; /* for binary compatibility only ! */
o_dev_t  u_ttyd; /*
 * for binary compatibility only -
 * NODEV will be assigned for large
 * controlling terminal devices.
 */
/*
 * Protected by pidlock
 */
k_sysset_t u_entrymask; /* /proc syscall stop-on-entry mask */
k_sysset_t u_exitmask; /* /proc syscall stop-on-exit mask */
k_sigset_t u_signodefer; /* signals defered when caught */
k_sigset_t u_sigonstack; /* signals taken on alternate stack */
k_sigset_t u_sigresethand; /* signals reset when caught */
k_sigset_t u_sigrestart; /* signals that restart system calls */
k_sigset_t u_sigmask[MAXSIG]; /* signals held while in catcher */
void (*u_signal[MAXSIG])(); /* Disposition of signals */
```

ejemplo de u_area en System V R4

```
/*
 * protected by u.u_procp->p_lock
 */
char u_nshmseg; /* # shm segments currently attached */
char u_acflag; /* accounting flag */
short u_lock; /* process/text locking flags */

/*
 * Updates to individual fields in u_rlimit are atomic but to
 * ensure a meaningful set of numbers, p_lock is used whenever
 * more than 1 field in u_rlimit is read/modified such as
 * getrlimit() or setrlimit()
 */
struct rlimit u_rlimit[RLIM_NLIMITS]; /* resource usage limits */

kmutex_t u_flock; /* lock for u_nofiles and u_flist */
int u_nofiles; /* number of open file slots */
struct ufchunk u_flist; /* open file list */
} user_t;
```

credenciales

- cada usuario en el sistema es identificado por un número *user id* o *uid*
- cada grupo en el sistema es identificado por un número *group id* o *gid*
- hay un usuario especial en el sistema *root uid=0*
  - puede acceder a todos los ficheros
  - puede enviar señales a todos los procesos
  - puede realizar las llamadas al sistema privilegiadas

credenciales

un fichero tiene tres atributos que condicionan como pueden acceder a él los distintos procesos

- propietario (*uid* del fichero)
- grupo (*gid* del fichero)
- permisos (*modo* del fichero)

un proceso tiene las **credenciales**, que especifican a qué ficheros puede acceder y para qué y a que procesos puede enviar señales (y de qué procesos las puede recibir)

- dos pares de credenciales: real y efectivo
- *uid* y *gid* efectivos: condicionan el acceso a los ficheros
- hay sólo **tres** llamadas que cambian la credencial
  - *setuid()*
  - *setgid()*
  - *exec()*:
    1. *exec()* sobre un ejecutable con permisos **s****** cambia el *uid* efectivo del proceso que hace *exec()*
    2. *exec()* sobre un ejecutable con permisos *****s*** cambia el *gid* efectivo del proceso que hace *exec()*

credenciales

Consideremos el siguiente código

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <dirent.h>

#define MAX 60

main()
{
 uid_t u1, u2;
 char dir1[MAX], dir2[MAX];
 char real[MAX], efec[MAX];
 char err[2*MAX];
 DIR *d;

 u1=getuid(); u2=geteuid();
 strcpy(dir1,getpwuid(u1)->pw_dir);
 strcpy(dir2,getpwuid(u2)->pw_dir);
 strcpy(real,getpwuid(u1)->pw_name);
 strcpy(efec,getpwuid(u2)->pw_name);
```

credenciales

```
  if ((d=opendir(dir1))==NULL){
    sprintf (err,"ERROR dir=%s, real=%s, efectiva=%s", dir1, real, efec)
    perror(err);
    }
  else {
    printf ("ABIERTO dir=%s, real=%s, efectiva=%s\n", dir1, real, efec);
    closedir(d);
    }
  if ((d=opendir(dir2))==NULL){
    sprintf (err,"error dir=%s, real=%s, efectiva=%s", dir2, real, efec)
    perror(err);
    }
  else {
    printf ("ABIERTO dir=%s, real=%s, efectiva=%s\n", dir2, real, efec);
    closedir(d);
    }
} /*main*/
```

credenciales

ejecutándolo el usuario antonio y si los permisos del fichero son

```
% ls -o
total 20
-rw-r--r--  1 antonio   840 2006-10-03 12:20 credenciales.c
-rwxr-xr-x  1 visita  12726 2006-10-03 12:20 credenciales.out
%
```

produce la salida

```
% ./credenciales.out
ABIERTO dir=/home/antonio, real=antonio, efectiva=antonio
ABIERTO dir=/home/antonio, real=antonio, efectiva=antonio
%
```

credenciales

cambiando los permisos del ejecutable a 4755

```
% ls -o
total 20
-rw-r--r--  1 antonio   840 2006-10-03 12:20 credenciales.c
-rwsr-xr-x  1 visita  12726 2006-10-03 12:20 credenciales.out
%
```

la salida es

```
% ./credenciales.out
ERROR dir=/home/antonio, real=antonio, efectiva=visita: Permission deni
ABIERTO dir=/home/visita, real=antonio, efectiva=visita
%
```

variables de entorno

- son cadenas de caracteres
- usualmente tienen la forma "NOMBREVARIABLE=valorvariable"
- colocadas al final de la pila de usuario
- varias maneras de acceder
  - tercer argumento de *main()*: array de punteros a las variables de entorno. El último puntero es NULL
  - `extern char ** environ`: array de punteros a las variables de entorno. El último puntero es NULL
  - funciones de libreria. *putenv()*, *getenv()*, *setenv()*, *unsetenv()*

variables de entorno: ejemplo 1

```
/**entorno.c**/
#include <stdio.h>

extern char ** environ;

void MuestraEntorno (char **entorno, char * nombre_entorno)
{
 int i=0;

 while (entorno[i]!=NULL) {
    printf ("%p->%s[%d]=(%p) %s\n", &entorno[i],
      nombre_entorno, i,entorno[i],entorno[i]);
    i++;
  }
}
main (int argc, char * argv[], char *env[])

{
  int i;

  for (i=0; i<argc; i++)
    printf ("%p->argv[%d]=(%p) %s\n",
        &argv[i], i, argv[i], argv[i]);
  printf ("%p->argv[%d]=(%p) ---------\n",
        &argv[argc], argc, argv[argc]);
  printf ("%p->argv=%p\n%p->argc=%d \n", &argv, argv, &argc, argc);

  MuestraEntorno(env,"env");
  printf("%p->environ=%p\n%p->env=%p \n", &environ, environ, &env, env);

  MuestraEntorno(environ,"environ");
  printf("%p->environ=%p\n%p->env=%p \n", &environ, environ, &env, env);
}
```

variables de entorno: ejemplo 1

```
%./entorno.out uno dos tres
0xbfbfffba0->argv[0]=(0xbfbfffc8c) ./entorno.out
0xbfbfffba4->argv[1]=(0xbfbfffc9a) uno
0xbfbfffba8->argv[2]=(0xbfbfffc9e) dos
0xbfbfffbac->argv[3]=(0xbfbfffca2) tres
0xbfbfffbb0->argv[4]=(0x0) ---------
0xbfbfffb5c->argv=0xbfbfffba0
0xbfbfffb58->argc=4
0xbfbfffbb4->env[0]=(0xbfbfffca7) USER=visita
0xbfbfffbb8->env[1]=(0xbfbfffcb4) LOGNAME=visita
0xbfbfffbbc->env[2]=(0xbfbfffcc4) HOME=/home/visita
0xbfbfffbc0->env[3]=(0xbfbfffcd7) MAIL=/var/mail/visita
0xbfbfffbc4->env[4]=(0xbfbfffcee) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/l
0xbfbfffbc8->env[5]=(0xbfbfffd5c) TERM=xterm
0xbfbfffbcc->env[6]=(0xbfbfffd67) BLOCKSIZE=K
0xbfbfffbd0->env[7]=(0xbfbfffd73) FTP_PASSIVE_MODE=YES
0xbfbfffbd4->env[8]=(0xbfbfffd88) SHELL=/bin/csh
0xbfbfffbd8->env[9]=(0xbfbfffd97) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbdc->env[10]=(0xbfbfffdb8) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbe0->env[11]=(0xbfbfffdec) SSH_TTY=/dev/ttyp0
0xbfbfffbe4->env[12]=(0xbfbfffdff) HOSTTYPE=FreeBSD
0xbfbfffbe8->env[13]=(0xbfbfffe10) VENDOR=intel
0xbfbfffbec->env[14]=(0xbfbfffe1d) OSTYPE=FreeBSD
0xbfbfffbf0->env[15]=(0xbfbfffe2c) MACHTYPE=i386
0xbfbfffbf4->env[16]=(0xbfbfffe3a) SHLVL=1
0xbfbfffbf8->env[17]=(0xbfbfffe42) PWD=/home/visita/c
0xbfbfffbfc->env[18]=(0xbfbfffe56) GROUP=users
0xbfbfffc00->env[19]=(0xbfbfffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbfffc04->env[20]=(0xbfbfffe7e) REMOTEHOST=portatil
0xbfbfffc08->env[21]=(0xbfbfffe92) EDITOR=vi
0xbfbfffc0c->env[22]=(0xbfbfffe9c) PAGER=more
0x80497fc->environ=0xbfbfffbb4
0xbfbfffb60->env=0xbfbfffbb4
```

variables de entorno: ejemplo 1

```
0xbfbfffbb4->environ[0]=(0xbfbffca7) USER=visita
0xbfbfffbb8->environ[1]=(0xbfbffcb4) LOGNAME=visita
0xbfbfffbbc->environ[2]=(0xbfbffcc4) HOME=/home/visita
0xbfbfffbc0->environ[3]=(0xbfbffcd7) MAIL=/var/mail/visita
0xbfbfffbc4->environ[4]=(0xbfbffcee) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/lo
0xbfbfffbc8->environ[5]=(0xbfbffd5c) TERM=xterm
0xbfbfffbcc->environ[6]=(0xbfbffd67) BLOCKSIZE=K
0xbfbfffbd0->environ[7]=(0xbfbffd73) FTP_PASSIVE_MODE=YES
0xbfbfffbd4->environ[8]=(0xbfbffd88) SHELL=/bin/csh
0xbfbfffbd8->environ[9]=(0xbfbffd97) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbdc->environ[10]=(0xbfbffdb8) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbe0->environ[11]=(0xbfbffdec) SSH_TTY=/dev/ttyp0
0xbfbfffbe4->environ[12]=(0xbfbffdff) HOSTTYPE=FreeBSD
0xbfbfffbe8->environ[13]=(0xbfbffe10) VENDOR=intel
0xbfbfffbec->environ[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0xbfbfffbf0->environ[15]=(0xbfbffe2c) MACHTYPE=i386
0xbfbfffbf4->environ[16]=(0xbfbffe3a) SHLVL=1
0xbfbfffbf8->environ[17]=(0xbfbffe42) PWD=/home/visita/c
0xbfbfffbfc->environ[18]=(0xbfbffe56) GROUP=users
0xbfbfffc00->environ[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbfffc04->environ[20]=(0xbfbffe7e) REMOTEHOST=portatil
0xbfbfffc08->environ[21]=(0xbfbffe92) EDITOR=vi
0xbfbfffc0c->environ[22]=(0xbfbffe9c) PAGER=more
0x80497fc->environ=0xbfbfffbb4
0xbfbfffb60->env=0xbfbfffbb4
%
```

variables de entorno: ejemplo 2

```
/**entorno2.c**/
#include <stdio.h>
#include <stdlib.h>

extern char ** environ;

void MuestraEntorno (char **entorno, char * nombre_entorno)
{
 ...........
}
main (int argc, char * argv[], char *env[])

{
  int i;

  for (i=0; i<argc; i++)
    printf ("%p->argv[%d]=(%p) %s\n",
        &argv[i], i, argv[i], argv[i]);
  printf ("%p->argv[%d]=(%p) ---------\n",
        &argv[argc], argc, argv[argc]);
  printf ("%p->argv=%p\n%p->argc=%d \n", &argv, argv, &argc, argc);

  MuestraEntorno(env,"env");
  MuestraEntorno(environ,"environ");
  printf("%p->environ=%p\n%p->env=%p \n\n\n", &environ, environ, &env, env);

  putenv ("NUEVAVARIABLE=XXXXXXXXXX");

  MuestraEntorno(env,"env");
  MuestraEntorno(environ,"environ");
  printf("%p->environ=%p\n%p->env=%p \n", &environ, environ, &env, env);
}
```

variables de entorno: ejemplo 2

```
%./entorno2.out
0xbfbffbb8->argv[0]=(0xbfbffc98) ./entorno2.out
0xbfbffbbc->argv[1]=(0x0) ---------
0xbfbffb6c->argv=0xbfbffbb8
0xbfbffb68->argc=1
0xbfbffbc0->env[0]=(0xbfbffca7) USER=visita
............
0xbfbffbf8->env[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0xbfbffbfc->env[15]=(0xbfbffe2c) MACHTYPE=i386
0xbfbffc00->env[16]=(0xbfbffe3a) SHLVL=1
0xbfbffc04->env[17]=(0xbfbffe42) PWD=/home/visita/c
0xbfbffc08->env[18]=(0xbfbffe56) GROUP=users
0xbfbffc0c->env[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbffc10->env[20]=(0xbfbffe7e) REMOTEHOST=portatil
0xbfbffc14->env[21]=(0xbfbffe92) EDITOR=vi
0xbfbffc18->env[22]=(0xbfbffe9c) PAGER=more
0xbfbffbc0->environ[0]=(0xbfbffca7) USER=visita
............
0xbfbffbf8->environ[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0xbfbffbfc->environ[15]=(0xbfbffe2c) MACHTYPE=i386
0xbfbffc00->environ[16]=(0xbfbffe3a) SHLVL=1
0xbfbffc04->environ[17]=(0xbfbffe42) PWD=/home/visita/c
0xbfbffc08->environ[18]=(0xbfbffe56) GROUP=users
0xbfbffc0c->environ[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbffc10->environ[20]=(0xbfbffe7e) REMOTEHOST=portatil
0xbfbffc14->environ[21]=(0xbfbffe92) EDITOR=vi
0xbfbffc18->environ[22]=(0xbfbffe9c) PAGER=more
0x80498d8->environ=0xbfbffbc0
0xbfbffb70->env=0xbfbffbc0
```

variables de entorno: ejemplo 2

```
0xbfbfffbc0->env[0]=(0xbfbffca7) USER=visita
..............
0xbfbfffbf8->env[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0xbfbfffbfc->env[15]=(0xbfbffe2c) MACHTYPE=i386
0xbfbfffc00->env[16]=(0xbfbffe3a) SHLVL=1
0xbfbfffc04->env[17]=(0xbfbffe42) PWD=/home/visita/c
0xbfbfffc08->env[18]=(0xbfbffe56) GROUP=users
0xbfbfffc0c->env[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbfffc10->env[20]=(0xbfbffe7e) REMOTEHOST=portatil
0xbfbfffc14->env[21]=(0xbfbffe92) EDITOR=vi
0xbfbfffc18->env[22]=(0xbfbffe9c) PAGER=more
0x804c000->environ[0]=(0xbfbffca7) USER=visita
..............
0x804c038->environ[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0x804c03c->environ[15]=(0xbfbffe2c) MACHTYPE=i386
0x804c040->environ[16]=(0xbfbffe3a) SHLVL=1
0x804c044->environ[17]=(0xbfbffe42) PWD=/home/visita/c
0x804c048->environ[18]=(0xbfbffe56) GROUP=users
0x804c04c->environ[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0x804c050->environ[20]=(0xbfbffe7e) REMOTEHOST=portatil
0x804c054->environ[21]=(0xbfbffe92) EDITOR=vi
0x804c058->environ[22]=(0xbfbffe9c) PAGER=more
0x804c05c->environ[23]=(0x804a080) NUEVAVARIABLE=XXXXXXXXXXX
0x80498d8->environ=0x804c000
0xbfbfffb70->env=0xbfbfffbc0
%
```

variables de entorno: ejemplo 3

```
/**entorno3.c**/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

extern char ** environ;
void MuestraEntorno (char **entorno, char * nombre_entorno)
{
. ............
}
main (int argc, char * argv[], char *env[])

{
  int i;

  for (i=0; i<argc; i++)
    printf ("%p->argv[%d]=(%p) %s\n",
        &argv[i], i, argv[i], argv[i]);
  printf ("%p->argv[%d]=(%p) ---------\n",
        &argv[argc], argc, argv[argc]);
  printf ("%p->argv=%p\n%p->argc=%d \n", &argv, argv, &argc, argc);

  MuestraEntorno(env,"env");
  MuestraEntorno(environ,"environ");
  printf("%p->environ=%p\n%p->env=%p \n\n\n", &environ, environ, &env, env);

  putenv ("NUEVAVARIABLE=XXXXXXXXXXX");

  MuestraEntorno(env,"env");
  MuestraEntorno(environ,"environ");
  printf("%p->environ=%p\n%p->env=%p \n", &environ, environ, &env, env);

  execl("./entorno.out","entorno.out", NULL);
}
```

variables de entorno: ejemplo 3

```
%./entorno3.out
0xbfbffbb8->argv[0]=(0xbfbffc98) ./entorno3.out
0xbfbffbbc->argv[1]=(0x0) ---------
0xbfbffb6c->argv=0xbfbffbb8
0xbfbffb68->argc=1
0xbfbffbc0->env[0]=(0xbfbffca7) USER=visita
0xbfbffbc4->env[1]=(0xbfbffcb4) LOGNAME=visita
............
0xbfbffc14->environ[21]=(0xbfbffe92) EDITOR=vi
0xbfbffc18->environ[22]=(0xbfbffe9c) PAGER=more
0x8049944->environ=0xbfbffbc0
0xbfbffb70->env=0xbfbffbc0


0xbfbffbc0->env[0]=(0xbfbffca7) USER=visita
..............
0xbfbffc14->env[21]=(0xbfbffe92) EDITOR=vi
0xbfbffc18->env[22]=(0xbfbffe9c) PAGER=more
0x804c000->environ[0]=(0xbfbffca7) USER=visita
0x804c004->environ[1]=(0xbfbffcb4) LOGNAME=visita
..............
0x804c054->environ[21]=(0xbfbffe92) EDITOR=vi
0x804c058->environ[22]=(0xbfbffe9c) PAGER=more
0x804c05c->environ[23]=(0x804a080) NUEVAVARIABLE=XXXXXXXXXXX
0x8049944->environ=0x804c000
0xbfbffb70->env=0xbfbffbc0
```

variables de entorno: ejemplo 3

```
0xbfbfffb9c->argv[0]=(0xbfbffc80) entorno.out
0xbfbfffba0->argv[1]=(0x0) ---------
0xbfbfffb4c->argv=0xbfbfffb9c
0xbfbfffb48->argc=1
0xbfbfffba4->env[0]=(0xbfbffc8c) USER=visita
0xbfbfffba8->env[1]=(0xbfbffc99) LOGNAME=visita
0xbfbfffbac->env[2]=(0xbfbffca9) HOME=/home/visita
0xbfbfffbb0->env[3]=(0xbfbffcbc) MAIL=/var/mail/visita
0xbfbfffbb4->env[4]=(0xbfbffcd3) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/l
0xbfbfffbb8->env[5]=(0xbfbffd41) TERM=xterm
0xbfbfffbbc->env[6]=(0xbfbffd4c) BLOCKSIZE=K
0xbfbfffbc0->env[7]=(0xbfbffd58) FTP_PASSIVE_MODE=YES
0xbfbfffbc4->env[8]=(0xbfbffd6d) SHELL=/bin/csh
0xbfbfffbc8->env[9]=(0xbfbffd7c) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbcc->env[10]=(0xbfbffd9d) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbd0->env[11]=(0xbfbffdd1) SSH_TTY=/dev/ttyp0
0xbfbfffbd4->env[12]=(0xbfbffde4) HOSTTYPE=FreeBSD
0xbfbfffbd8->env[13]=(0xbfbffdf5) VENDOR=intel
0xbfbfffbdc->env[14]=(0xbfbffe02) OSTYPE=FreeBSD
0xbfbfffbe0->env[15]=(0xbfbffe11) MACHTYPE=i386
0xbfbfffbe4->env[16]=(0xbfbffe1f) SHLVL=1
0xbfbfffbe8->env[17]=(0xbfbffe27) PWD=/home/visita/c
0xbfbfffbec->env[18]=(0xbfbffe3b) GROUP=users
0xbfbfffbf0->env[19]=(0xbfbffe47) HOST=gallaecia.dc.fi.udc.es
0xbfbfffbf4->env[20]=(0xbfbffe63) REMOTEHOST=portatil
0xbfbfffbf8->env[21]=(0xbfbffe77) EDITOR=vi
0xbfbfffbfc->env[22]=(0xbfbffe81) PAGER=more
0xbfbfffc00->env[23]=(0xbfbffe8c) NUEVAVARIABLE=XXXXXXXXXXX
0x80497fc->environ=0xbfbfffba4
0xbfbfffb50->env=0xbfbfffba4
```

variables de entorno: ejemplo 3

```
0xbfbfffba4->environ[0]=(0xbfbfffc8c) USER=visita
0xbfbfffba8->environ[1]=(0xbfbfffc99) LOGNAME=visita
0xbfbfffbac->environ[2]=(0xbfbfffca9) HOME=/home/visita
0xbfbfffbb0->environ[3]=(0xbfbfffcbc) MAIL=/var/mail/visita
0xbfbfffbb4->environ[4]=(0xbfbfffcd3) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/lo
0xbfbfffbb8->environ[5]=(0xbfbfffd41) TERM=xterm
0xbfbfffbbc->environ[6]=(0xbfbfffd4c) BLOCKSIZE=K
0xbfbfffbc0->environ[7]=(0xbfbfffd58) FTP_PASSIVE_MODE=YES
0xbfbfffbc4->environ[8]=(0xbfbfffd6d) SHELL=/bin/csh
0xbfbfffbc8->environ[9]=(0xbfbfffd7c) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbcc->environ[10]=(0xbfbfffd9d) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbd0->environ[11]=(0xbfbfffdd1) SSH_TTY=/dev/ttyp0
0xbfbfffbd4->environ[12]=(0xbfbfffde4) HOSTTYPE=FreeBSD
0xbfbfffbd8->environ[13]=(0xbfbfffdf5) VENDOR=intel
0xbfbfffbdc->environ[14]=(0xbfbffe02) OSTYPE=FreeBSD
0xbfbfffbe0->environ[15]=(0xbfbffe11) MACHTYPE=i386
0xbfbfffbe4->environ[16]=(0xbfbffe1f) SHLVL=1
0xbfbfffbe8->environ[17]=(0xbfbffe27) PWD=/home/visita/c
0xbfbfffbec->environ[18]=(0xbfbffe3b) GROUP=users
0xbfbfffbf0->environ[19]=(0xbfbffe47) HOST=gallaecia.dc.fi.udc.es
0xbfbfffbf4->environ[20]=(0xbfbffe63) REMOTEHOST=portatil
0xbfbfffbf8->environ[21]=(0xbfbffe77) EDITOR=vi
0xbfbfffbfc->environ[22]=(0xbfbffe81) PAGER=more
0xbfbfffc00->environ[23]=(0xbfbffe8c) NUEVAVARIABLE=XXXXXXXXXXX
0x80497fc->environ=0xbfbfffba4
0xbfbfffb50->env=0xbfbfffba4
```

ejecución en modo kernel

- Cualquiera de estos tres sucesos hace que el sistema pase a modo kernel
  - **Interrupción de dispopsitivo:** Es asíncrona al proceso en CPU, producida por un dispositivo externo que necesita comunicarse con el S.O.. Puede ocurrir en cualquier momento: cuando el proceso actual está en modo usuario, en modo kernel ejecutando una llamada al sistema o incluso cuando se está ejecutando la rutina de servicio de otra interrupción.
  - **Excepción:** Es síncrona al proceso en CPU y está causada por éste (división entre 0, referencia a una dirección de memoria no válida, instrucción ilegal . . .)
  - **Llamada al sistema:** El proceso en CPU solicita algo explicitamente al S.O.
- En cualquiera de estos casos, cuando el kernel recibe el control
  - salva el estado del proceso actual en su pila de kernel
  - ejecuta la rutina correspondiente al suceso del que se trate
  - cuando la rutina se completa, el kernel restaura el estado del proceso y el modo e ejecución a su valor anterior.

ejecución en modo kernel: interrupción

- una interrupción puede ocurrir en cualquier momento, incluso si se está tratando otra
- a cada interrupción se le asigna un Nivel de Prioridad de Interrupción (IPL-**I**nterrupt **P**riority **L**evel)
- cuando ocurre una interrupción se compara su *ipl* con el *ipl* actual. Si es mayor se invoca al manejador correspondiente, si nó, no se atiende inmediatamente y la ejecución de su manejador se pospone hasta que el ipl descienda lo suficiente.
- todo el código de usuario y la mayor parte del código del kernel (todo salvo las rutinas de servicio de interrupciones y pequeños fragmentos en algunas llamadas al sistema) se ejecuta con *ipl* mínimo
- en sistemas tradicionales va de 0 a 7 y en BSD de 0 a 31

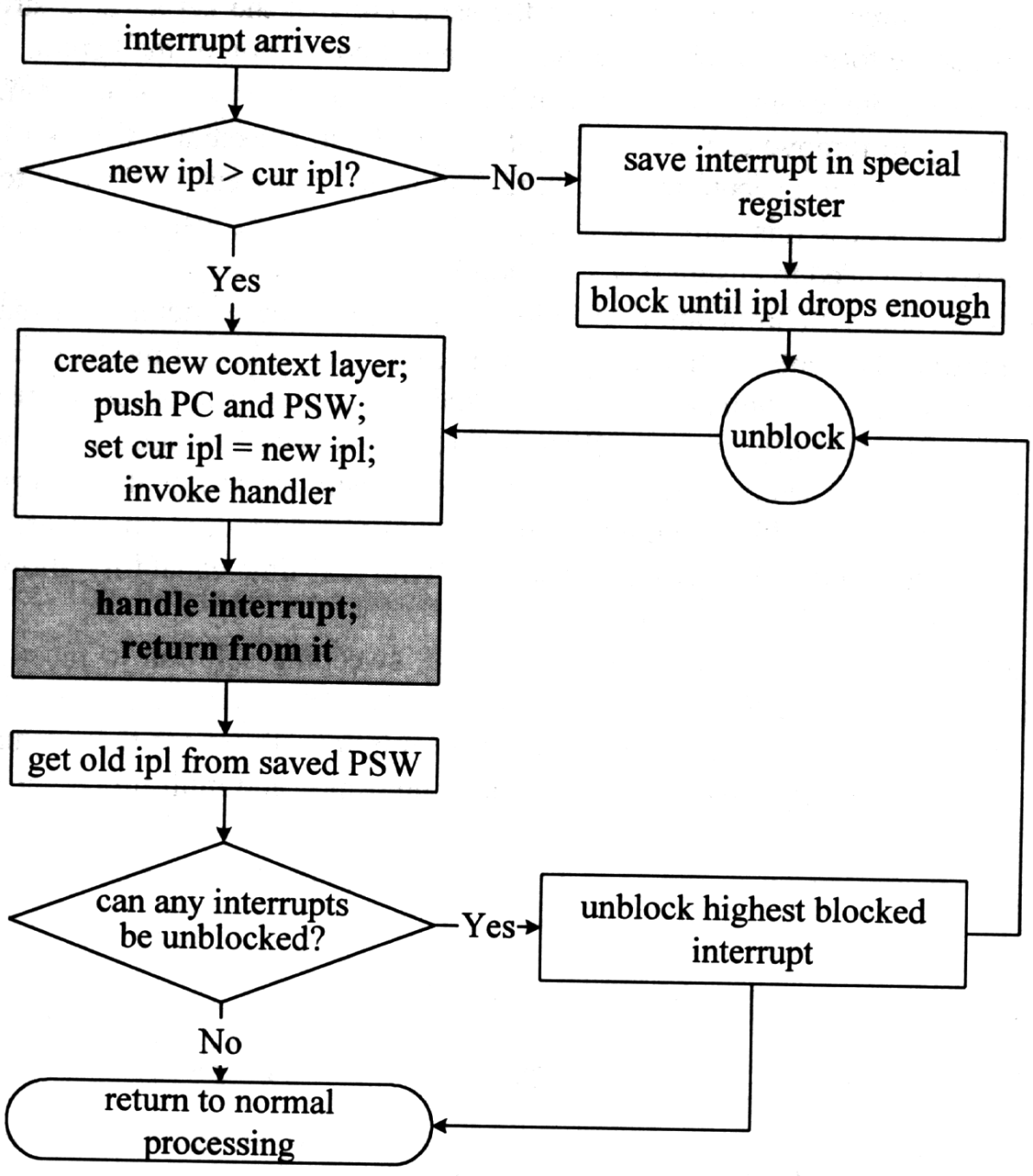

interrupt arrives
new ipl > cur ipl?
No
save interrupt in special register
block until ipl drops enough
Yes
create new context layer;
push PC and PSW;
set cur ipl = new ipl;
invoke handler
unblock
handle interrupt;
return from it
get old ipl from saved PSW
can any interrupts be unblocked?
Yes
unblock highest blocked interrupt
No
return to normal processing

Table 2-1. Setting the interrupt priority level in 4.3BSD and SVR4

4.3BSD	SVR4	Purpose
`spl0`	`spl0` or `splbase`	enable all interrupts
`splsoftclock`	`spltimeout`	block functions scheduled by timers
`splnet`		block network protocol processing
	`splstr`	block STREAMS interrupts
`spltty`	`spltty`	block terminal interrupts
`splbio`	`spldisk`	block disk interrupts
`splimp`		block network device interrupts
`splclock`		block hardware clock interrupt
`splhigh`	`spl7` or `splhi`	disable all interrupts
`splx`	`splx`	restore *ipl* to previously saved value

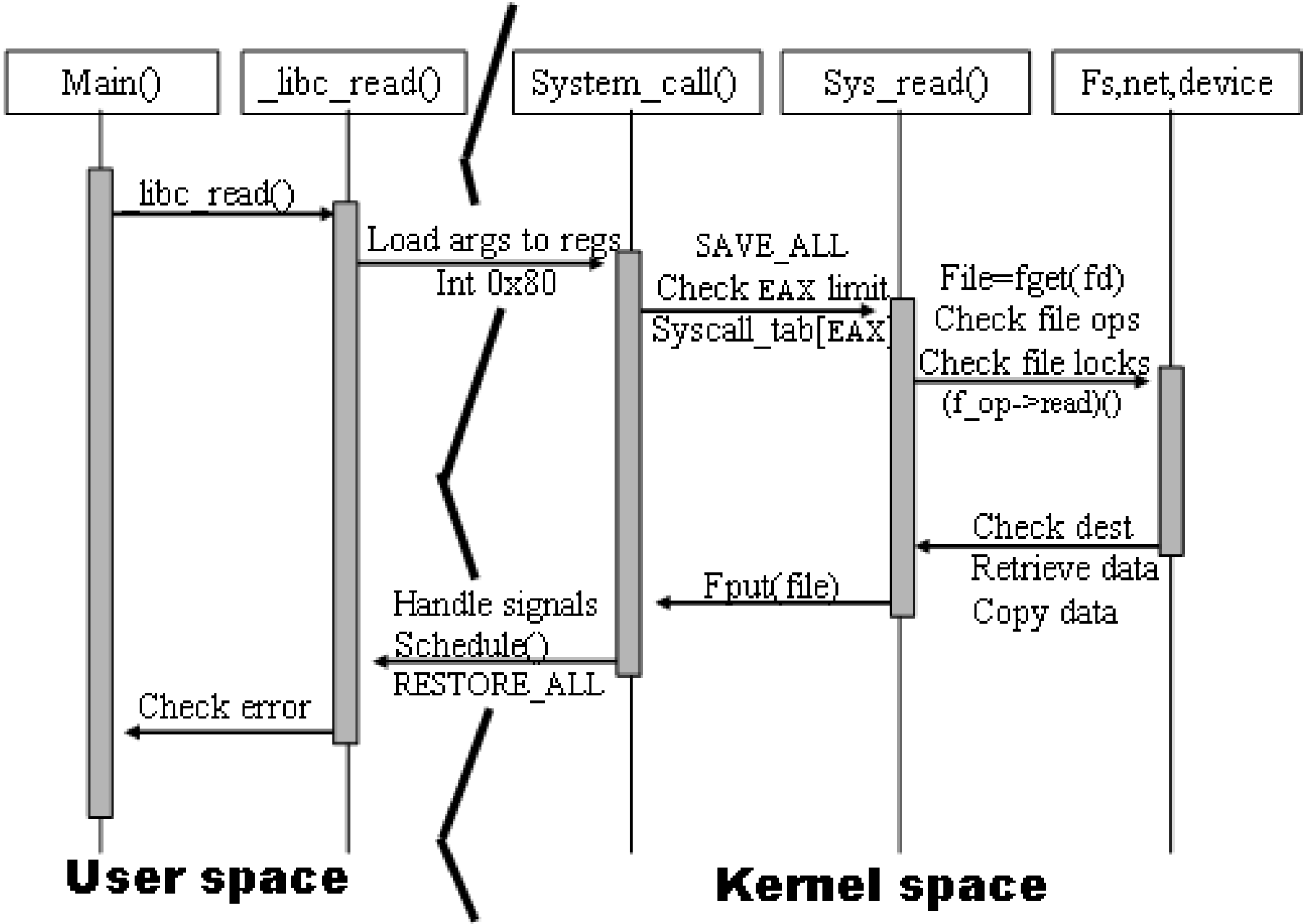
ejecución en modo kernel: llamada al sistema

vemos una funcion envoltorio de librería (*open()*, *read()*, *fork()* . . .)

- **función de librería (p. e. `read()`)**
- recibe los parámetros en la pila de usuario
- pone el número de servicio asociado en la pila (o un registro específico del microprocesador)
- ejecuta una instrucción especial (*trap*, *chmk*, *int* . . .) que cambia a modo kernel. Esta instrucción especial, además de cambiar a modo kernel transfiere el control al *handler* de llamadas al sistema *syscall()*
  - `syscall()`

ejecución en modo kernel: llamada al sistema

-
  - `syscall()`
  - copia los argumentos de la pila de usuario a la `u_area`
  - salva el estado del proceso en su pila del kernel
  - utiliza el número de servicio como un índice en un array (`sysent[]`) que le dice que función del kernel debe llamar (p.e. `sys_read()`)
    - **función llamada por `syscall()`: p.e. `sys_read()`**
    - es la que proporciona el servicio
    - si tiene que llamar a otras funciones dentro del kernel utiliza la pila del kernel
  - pone los valores de retorno (o error) en los registros adecuados
  - restaura estado del proceso y vuelve a modo usuario devolviendo el control a la rutina de librería
- devuelve el control y los valores correspondientes a la función que la llamó


Main()
_libc_read()
System_call()
Sys_read()
Fs,net,device
libc_read()
Load args to regs
Int 0x80
SAVE_ALL
Check EAX limit
Syscall_tab[EAX]
File=fget(fd)
Check file ops
Check file locks
(f_op->read)()
Check dest
Retrieve data
Copy data
Fput(file)
Handle signals
Schedule()
RESTORE_ALL
Check error
User space
Kernel space

números llamada al sistema en linux

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_oldstat            18
#define __NR_lseek              19
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_setuid             23
#define __NR_getuid             24
#define __NR_stime              25
#define __NR_ptrace             26
#define __NR_alarm              27
```

números llamada al sistema en openBSD

```
/*      $OpenBSD: syscall.h,v 1.53 2001/08/26 04:11:12 deraadt Exp $    */

/*
 * System call numbers.
 *
 * DO NOT EDIT-- this file is automatically generated.
 * created from;        OpenBSD: syscalls.master,v 1.47 2001/06/26 19:56:52 dugsong Exp
 */
/* syscall: "syscall" ret: "int" args: "int" "..." */
#define SYS_syscall     0
/* syscall: "exit" ret: "void" args: "int" */
#define SYS_exit        1
/* syscall: "fork" ret: "int" args: */
#define SYS_fork        2
/* syscall: "read" ret: "ssize_t" args: "int" "void *" "size_t" */
#define SYS_read        3
/* syscall: "write" ret: "ssize_t" args: "int" "const void *" "size_t" */
#define SYS_write       4
/* syscall: "open" ret: "int" args: "const char *" "int" "..." */
#define SYS_open        5
/* syscall: "close" ret: "int" args: "int" */
#define SYS_close       6
/* syscall: "wait4" ret: "int" args: "int" "int *" "int" "struct rusage *" */
#define SYS_wait4       7
                                /* 8 is compat_43 ocreat */
/* syscall: "link" ret: "int" args: "const char *" "const char *" */
#define SYS_link        9
/* syscall: "unlink" ret: "int" args: "const char *" */
#define SYS_unlink      10
                                /* 11 is obsolete execv */
/* syscall: "chdir" ret: "int" args: "const char *" */
#define SYS_chdir       12
```

números llamada al sistema en solaris

```
/*
 * Copyright (c) 1991-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 */
#ifndef _SYS_SYSCALL_H
#define _SYS_SYSCALL_H
#pragma ident   "@(#)syscall.h  1.77    01/07/07 SMI"
#ifdef  __cplusplus
extern "C" {
#endif

/*
 *      system call numbers
 *              syscall(SYS_xxxx, ...)
 */
        /* syscall enumeration MUST begin with 1 */
        /*
         * SunOS/SPARC uses 0 for the indirect system call SYS_syscall
         * but this doesn't count because it is just another way
         * to specify the real system call number.
         */
#define SYS_syscall     0
#define SYS_exit        1
#define SYS_fork        2
#define SYS_read        3
#define SYS_write       4
#define SYS_open        5
#define SYS_close       6
#define SYS_wait        7
#define SYS_creat       8
#define SYS_link        9
#define SYS_unlink      10
#define SYS_exec        11
#define SYS_chdir       12
```

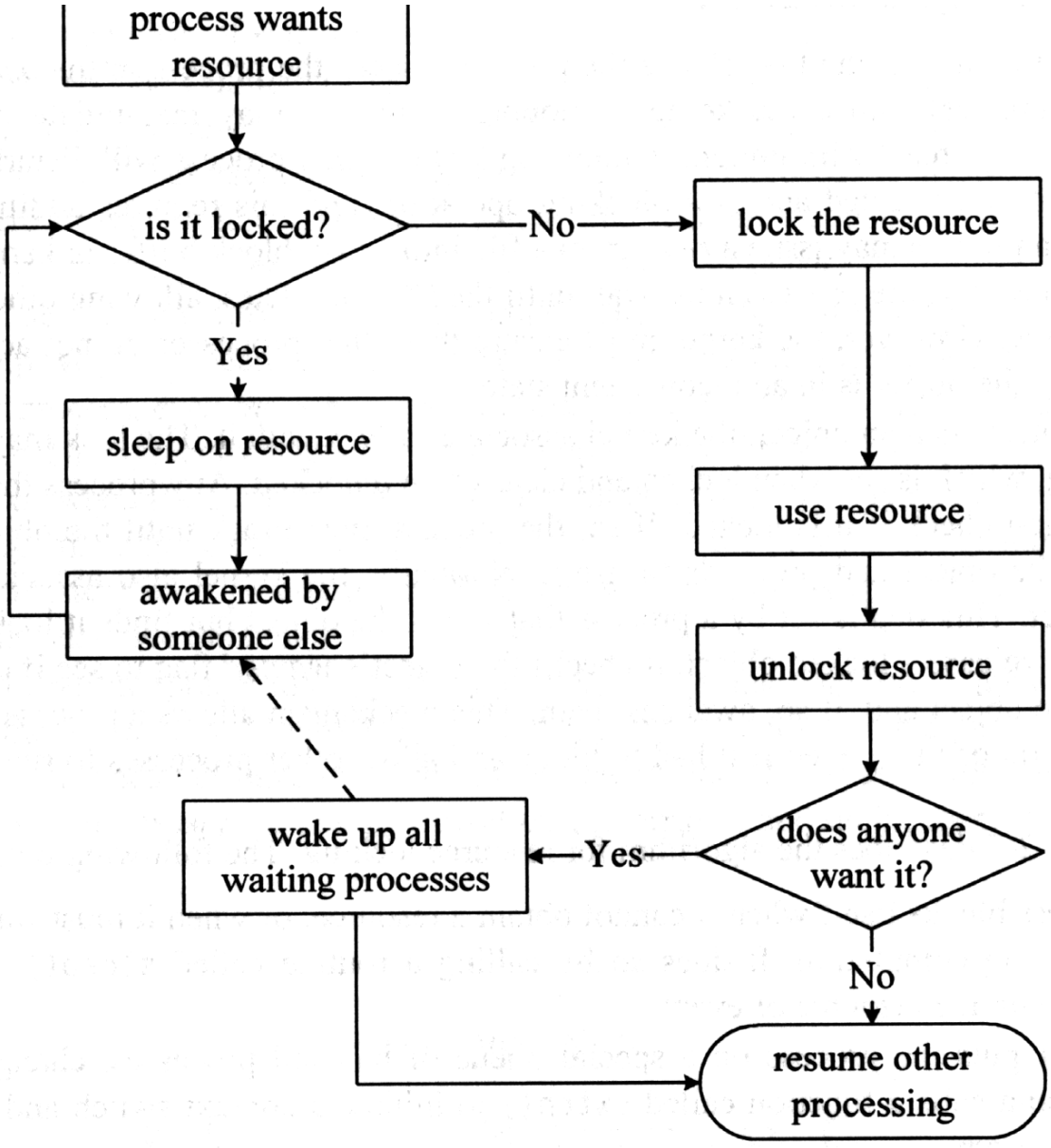
ejecución en modo kernel: recursos

- protección elemental datos del kernel: un proceso en modo kernel no puede ser apropiado: no abandona la CPU hasta que vuelva a modo usuario, pase a espera o termina
- dado que no puede ser apropiado puede manajar los datos del kernel sin riesgo de provocar inconsistencias.
- si está utilizando un recurso y pasa a espera debe dejarlo marcado como ocupado: antes de utilizar un recurso hay que comprobar si está ocupado. Si lo está se marca el recurso como demandado y se llama a *sleep()*).
  - *sleep()* pone el proceso en espera y llama a *swtch()* que inicia el cambio de contexto

ejecución en modo kernel: recursos

- cuando se deja de utilizar un recurso se libera, es decir se marca como no ocupado y si el recurso está demandado se despierta a **TODOS** los procesos en espera por ese recurso
  - *wakeup()* encuentra todos los procesos que esperan por un recurso, cambia su estado a listos y los coloca en la cola de listos
- como desde que un proceso es despertado hasta que obtiene la CPU puede pasar tiempo, al obtener la CPU debe volver a comprobar si el recurso por el que esperaba está libre

ejecución en modo kernel: recursos



ejecución en modo kernel: recursos

- aunque no pueda apropiarse la CPU si está ejecutando en modo kernel una interrupción puede ocurrir en cualquier momento
- al acceder a datos del kernel susceptibles de ser accedidos por alguna rutina de servicio de alguna interrupción debe inhabilitarse dicha interrupción (elevando *ipl*)
- hay que tener en cuanta además
  - las interrupciones requieren servicio rápido, debe restringirse su inhabilitación al mínimo posible
  - inhabilitar una interrupción inhabilita las de *ipl* menor
- si queremos que el modo kernel sea apropiable las estructuras deben protegerse con semáforos
- en sistemas multiprocesador los mecanismos de protección son más complejos

Planificación en sistemas unix tradicionales

- Prioridades apropiativas que se recalculan dinámicamente
  - siempre se ejecuta el proceso de más prioridad
  - menor número indica mayor prioridad
  - la prioridad del proceso que no obtiene la CPU aumenta
  - la prioridad del proceso en CPU disminuye
  - si un proceso con más prioridad que el que está en CPU pasa al estado de listo, expulsa al que está en CPU, salvo que éste último se estuviese ejecutando en modo kernel
- procesos de la misma prioridad se reparten la CPU en *round robin*.
- la prioridad en modo usuario se recalcula atendiendo a dos factores
  - factor *nice*: cambiable mediante la llamada al sistema *nice()*
  - uso de CPU: mayor uso (reciente) de CPU implica menor prioridad

Planificación en sistemas unix tradicionales

- en la estructura `proc` hay los siguientes miembros referidos al la prioridad

`p_cpu`	uso de cpu a efectos del cálculo de prioridades
`p_nice`	factor *nice*
`p_usrpri`	prioridad en modo usuario, recalculada periodicamente partir del uso de CPU y del factor *nice*
`p_pri`	prioridad del proceso, es la que se usa para planificar

- si el proceso está en modo usuario `p_pri` es idéntico a `p_usrpri`
- si el proceso pasa a espera, cuando es despertado se asigna a `p_pri` un valor que depende del motivo por el que el proceso estaba en espera. Es lo que se llama una *kernel priority* o *sleep priority*
  - estas *kernel priorities* son menores en valor absoluto, y por tanto representan mayor prioridad, que las prioridades en modo usuario en `p_usrpri`
  - con esto se consigue que las llamadas al sistema se completen antes

Planificación en sistemas unix tradicionales

Table 2-2. Sleep priorities in 4.3BSD UNIX

Priority	Value	Description
PSWP	0	swapper
PSWP + 1	1	page daemon
PSWP + 1/2/4	1/2/4	other memory management activity
PINOD	10	waiting for inode to be freed
PRIBIO	20	disk I/O wait
PRIBIO + 1	21	waiting for buffer to be released
PZERO	25	baseline priority
TTIPRI	28	terminal input wait
TTOPRI	29	terminal output wait
PWAIT	30	waiting for child process to terminate
PLOCK	35	advisory resource lock wait
PSLEP	40	wait for a signal

Planificación en sistemas unix tradicionales
Recálculo de las prioridades en modo usuario

- con cada *tic* de reloj, el *handler* incrementa `p_cpu` del proceso que está en cpu
- cada segundo la rutina *shedcpu()*
  1. ajusta el tiempo de cpu
     - en BSD $p\_cpu = \frac{2*cargamedia}{2*cargamedia+1} * p\_cpu$
     - en System V R3 $p\_cpu = \frac{p\_cpu}{2}$
  2. recalcula las prioridades
     - en BSD $p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2 * p\_nice$
     - en System V R3 $p\_usrpri = PUSER + \frac{p\_cpu}{2} + p\_nice$
- PUSER es un número que se suma para que las prioridades en modo usario sean menores (mayores en valor absoluto) que las *kernel priorities*

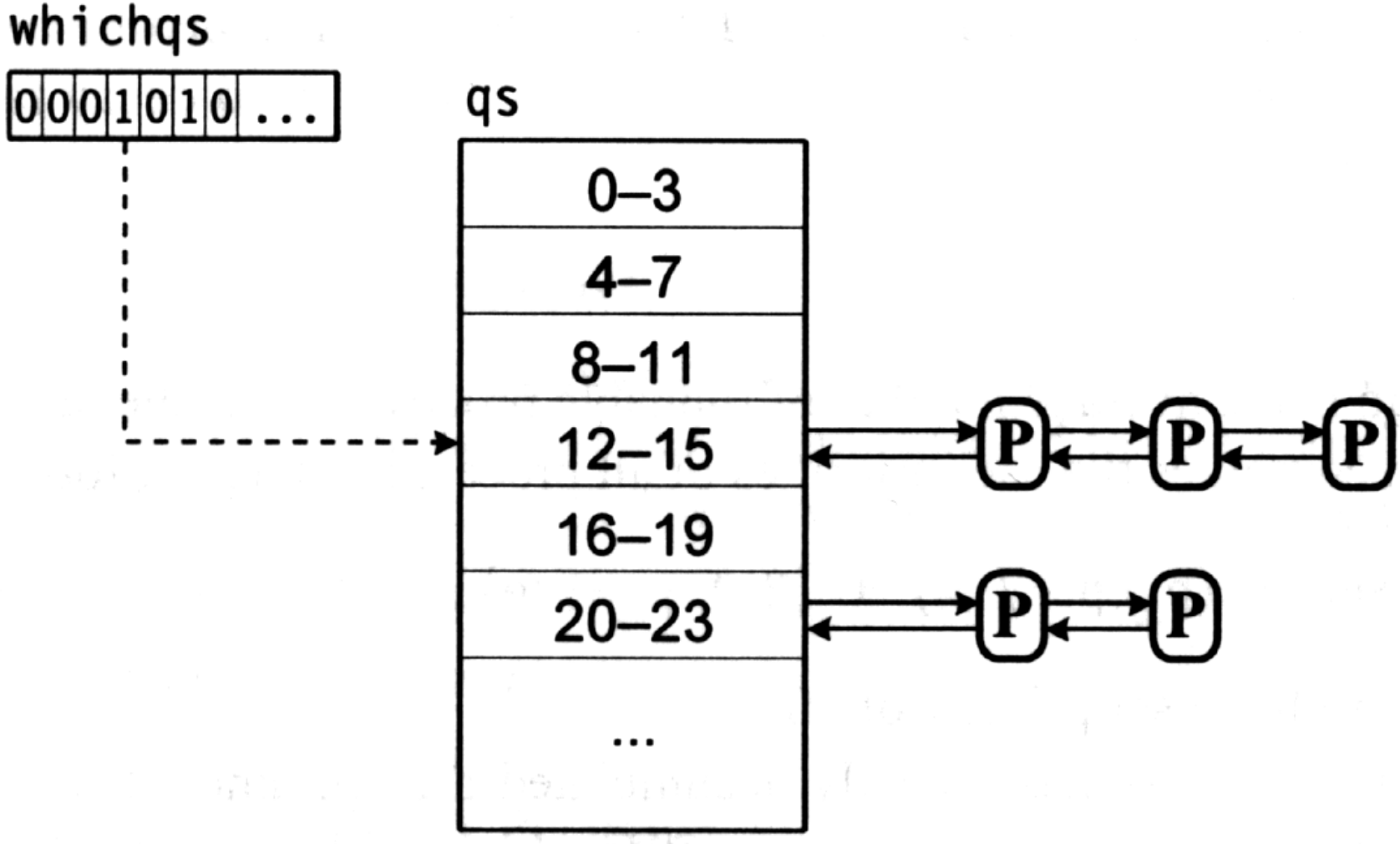
Ejemplo de planificación en SVR3 donde el *tic* ocurre 60 veces por segundo. *priority* representa la parte fija de la prioridad (`PUSER + p_nice`)

Time	Process A		Process B		Process C	
	Priority	CPU Count	Priority	CPU Count	Priority	CPU Count
0	60	0, 1, 2, ..., 60	60	0	60	0
1	75	30	60	0, 1, 2, ..., 60	60	0
2	67	15	75	30	60	0, 1, 2, ..., 60
3	63	7, 8, 9, ..., 67	67	15	75	30
4	76	33	63	7, 8, 9, ..., 67	67	15
5	68	16	76	33	63	7

Planificación en sistemas unix tradicionales
Implementación

- se implementa como un array de múltiples colas
- típicamente 32 colas. Cada cola tiene varias prioridades adyacentes
- al recalcular las prioridades se coloca el proceso en la cola correspondiente
- *swtch()* sólo tiene que cargar el proceso de la primera cola no vacia
- cada 100ms la rutina *roundrobin()* cambia de un proceso a otro de la misma cola

Planificación en sistemas unix tradicionales
Implementación



BSD scheduler data structures.

Planificación en sistemas unix tradicionales
Cambio de contexto

Se produce cambio de contexto si

a. El proceso actual pasa a espera o termina.
b. El recálculo de prioridades hace que aparezca listo un proceso de más prioridad
c. Un manejador de interrupciones o el proceso actual despierta a un proceso de mayor prioridad

a. Cambio de contexto *voluntario*. *swtch()* es invocada desde *sleep()* o *exit()*
b,c. Cambio de contexto *involuntario*, ocurre en modo kernel: el kernel pone un flag *(runrun)* indicando que ha de hacerse el cambio de contexto (llamar a *swtch()*) al volver a modo usuario

Planificación en sistemas unix tradicionales
Limitaciones

Este tipo de planificación presenta las siguientes limitaciones

- no escala bien
- no hay medio de garantizar una porción de la CPU a un proceso o grupo de procesos
- no hay garantía del tiempo de respuesta
- posibilidad de *inversión de prioridades*: procesos con menos prioridad pueden obtener más CPU que otros con más prioridad

Planificación en System V R4

- incluye aplicaciones en tiempo real
- separa la política de planificación de los mecanismos que la implementan
- se pueden incorporar nuevas políticas de planificación
- limita la latencia de las aplicaciones

Planificación en System V R4

Se definen una serie de clases de planificación (*scheduling class*) que determinan la política que se aplica a los procesos que pertenecen a ellas

- rutinas independientes de las clases
  - manipulación de colas
  - cambio de contexto
  - apropiación
- rutinas dependientes de las clases
  - cálculo de prioridades
  - herencia

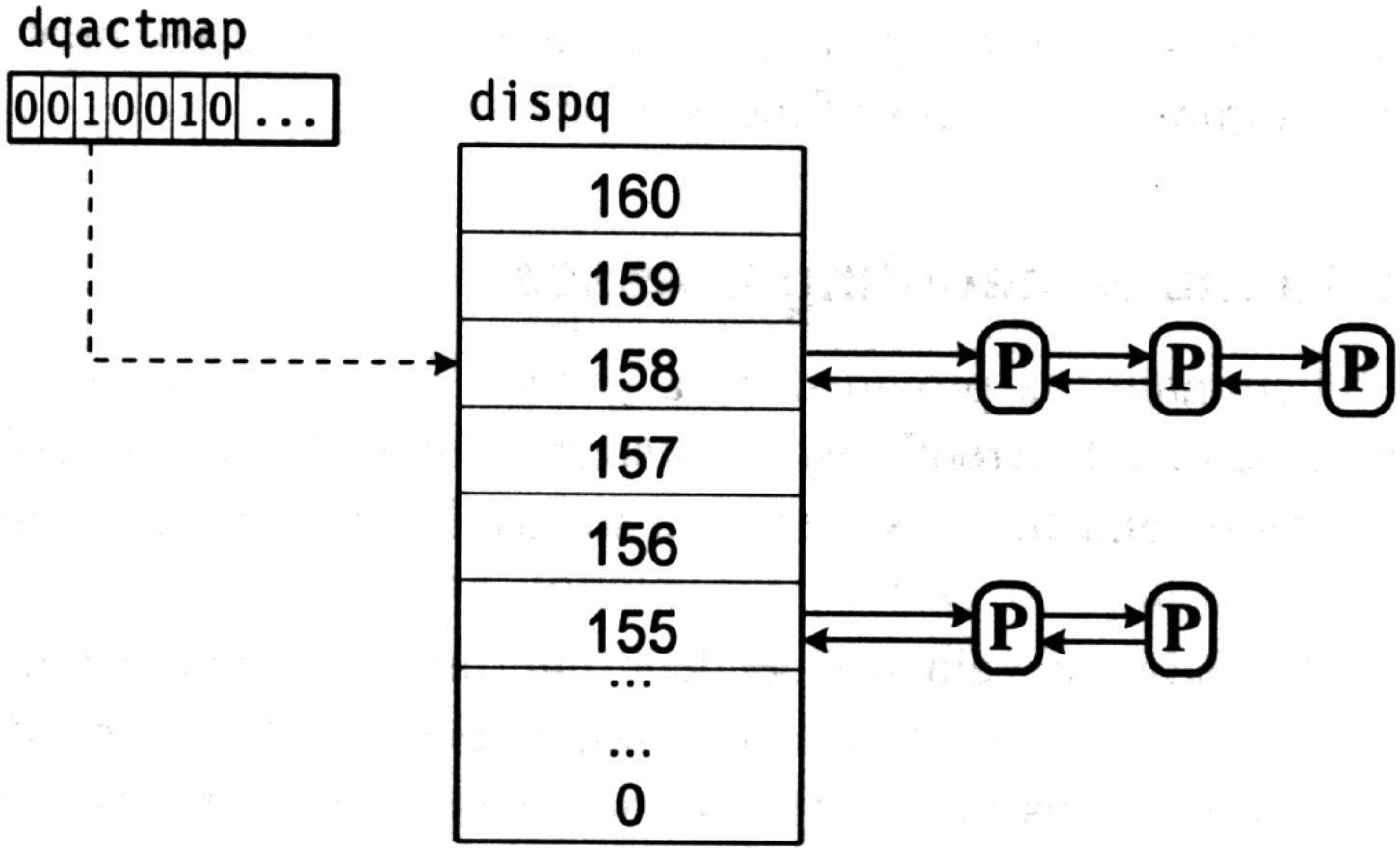
Planificación en System V R4

- prioridades entre 0 y 159: mayor número mayor prioridad
  - 0-59 *time sharing class*
  - 60-99 *system priority*
  - 100-159 *real time*
- en la estructura `proc`

`p_cid`	identificador de la clase
`p_clfuncs`	puntero a las funciones de la clase
`p_clproc`	puntero a datos dependientes de la clase

- se implementa como un array de múltiples colas

Planificación es System V R4
Implementación



SVR4 dispatch queues.

Planificación en System V R4

- varias clases predefinidas
  - clase *time sharing*
  - clase *system*
  - clase *real time*

```
%dispadmin -l
CONFIGURED CLASSES
==================
SYS      (System Class)
TS       (Time Sharing)
IA       (Interactive)
RT       (Real Time)
%
```

Planificación en System V R4: clase *time sharing*

- las prioridades en modo usuario se recalculan dinámicamente
- si un proceso pasa a espera se le asigna una *sleep priority* dependiendo del motivo por el que ha ido a espera. Cuando vuelve a modo usuario se utiliza su priordad en modo usuario
- cuanto dependiente de la prioridad: mayor prioridad $\Rightarrow$ menor cuanto
- no se recalculan todas las prioridades, solo las del proceso en CPU al abandonar la CPU
  - **usó todo su quanto**: su prioridad en modo usuario disminuye
  - **pasó a espera antes de usar todo su quanto**: su prioridad aumenta

Planificación en System V R4: clase *time sharing*

- los datos dependientes de la clase
  - `ts_timeleft` lo que le queda de cuanto
  - `ts_cpupri` parte de la prioridad en modo usuario puesta por el sistema (lo que se recalcula dinámicamente)
  - `ts_upri` parte de la prioridad en modo usuari puesta por el usuario (mediante la llamada *priocntl()*
  - `ts_umdpri` prioridad en modo usuario (`ts_cpupri` +`ts_upri`)
  - `ts_dispwait` segundos desde que comienza el cuanto

Planificación en System V R4: clase *time sharing*

- el sistema tiene una tabla que relaciona la prioridad con el cuanto y determina como se recalcula *cpupri*
- los campos en dicha tabla son

`quantum` el cuanto que corresponde a cada prioridad

`slpret` nueva `ts_cpupri` si no se agota el cuanto (se va a espera)

`tqexp` nueva `ts_cpupri` si se agota el cuanto

`maxwait` segundos que marca el límite donde se usa *lwait* como nueva *cpupri*

`lwait` nueva *cpupri* si transcurren mas de *maxwait* segundos

`pri` la prioridad, se usa tanto para relaccionar *umdpri* con el cuanto como para recalcular *cpupri*

Planificación en System V R4: clase *time sharing*. Ejemplo de tabla de la clase TS

```
bash-2.05$ dispadmin -c TS -g
# Time Sharing Dispatcher Configuration
RES=1000
# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
       200          0         50           0        50        #     0
       200          0         50           0        50        #     1
       200          0         50           0        50        #     2
       200          0         50           0        50        #     3
       200          0         50           0        50        #     4
       200          0         50           0        50        #     5
       200          0         50           0        50        #     6
       200          0         50           0        50        #     7
       200          0         50           0        50        #     8
       200          0         50           0        50        #     9
       160          0         51           0        51        #    10
       160          1         51           0        51        #    11
       160          2         51           0        51        #    12
       160          3         51           0        51        #    13
       160          4         51           0        51        #    14
       160          5         51           0        51        #    15
       160          6         51           0        51        #    16
       160          7         51           0        51        #    17
       160          8         51           0        51        #    18
       160          9         51           0        51        #    19
       120         10         52           0        52        #    20
       120         11         52           0        52        #    21
       120         12         52           0        52        #    22
       120         13         52           0        52        #    23
       120         14         52           0        52        #    24
       120         15         52           0        52        #    25
```

Planificación en System V R4: clase *time sharing*. Ejemplo de tabla de la clase TS

```
        120      16      52       0      52      #      26
        120      17      52       0      52      #      27
        120      18      52       0      52      #      28
        120      19      52       0      52      #      29
         80      20      53       0      53      #      30
         80      21      53       0      53      #      31
         80      22      53       0      53      #      32
         80      23      53       0      53      #      33
         80      24      53       0      53      #      34
         80      25      54       0      54      #      35
         80      27      54       0      54      #      37
         80      28      54       0      54      #      38
         80      29      54       0      54      #      39
         40      30      55       0      55      #      40
         40      31      55       0      55      #      41
         40      32      55       0      55      #      42
         40      33      55       0      55      #      43
         40      34      55       0      55      #      44
         40      35      56       0      56      #      45
         40      36      57       0      57      #      46
         40      37      58       0      58      #      47
         40      38      58       0      58      #      48
         40      39      58       0      59      #      49
         40      40      58       0      59      #      50
         40      41      58       0      59      #      51
         40      42      58       0      59      #      52
         40      43      58       0      59      #      53
         40      44      58       0      59      #      54
         40      45      58       0      59      #      55
         40      46      58       0      59      #      56
         40      47      58       0      59      #      57
         40      48      58       0      59      #      58
         20      49      59   32000      59      #      59
bash-2.05$
```

Planificación en System V R4: clase *real time*

- usa prioridades 100-159
- prioridades fijas y cuantum fijo, sólo cambianles mediante *priocntl()*
- el kernel mantiene una tabla que relacciona las prioridades con los cuantos (en el caso de que no se especfique uno)
- en principio a mayor prioridad menor cuanto
- al terminar el cuanto vuelve al final de la cola de esa misma prioridad

Planificación en System V R4: clase *real time*

- si un proceso se está ejecutando en modo kernel y aparece listo un proceso en tiempo real, no puede apropiarle inmediatamente (el kernel puede no estar en estado consistente). Lo hará en cuanto
  - pase a espera
  - vuelva a modo usuario
  - llegue a un punto de apropiación (*preemption point*)
- el kernel indica que hay un proceso en tiempo real listo para ejecución mediante el flag *kprunrun*
- un punto de apropiación es un punto dentro del código del kernel en donde el sistema está en un estado consistente y en donde se comprueba el flag *kprunrun* y se inicia el cambio de contexto en caso necesario
- en Solaris todas la estructuras del kernel están protegidas por semáforos: la ejecución en modo kernel es apropiable

Planificación en System V R4: clase *real time*. Ejemplo de tabla de la clase RT

```
bash-2.05$ dispadmin -c RT -g
# Real Time Dispatcher Configuration
RES=1000

# TIME QUANTUM                    PRIORITY
# (rt_quantum)                      LEVEL
      1000                  #         0
      1000                  #         1
      1000                  #         2
      1000                  #         3
      1000                  #         4
      1000                  #         5
      1000                  #         6
      1000                  #         7
      1000                  #         8
      1000                  #         9
       800                  #        10
       800                  #        11
       800                  #        12
       800                  #        13
       800                  #        14
       800                  #        15
       800                  #        16
       800                  #        17
       800                  #        18
       800                  #        19
       600                  #        20
       600                  #        21
       600                  #        22
       600                  #        23
       600                  #        24
       600                  #        25
```

Planificación en System V R4: clase *real time*. Ejemplo de tabla de la clase RT

```
        600                     #       26
        600                     #       27
        600                     #       28
        600                     #       29
        400                     #       30
        400                     #       31
        400                     #       32
        400                     #       33
        400                     #       34
        400                     #       35
        400                     #       36
        400                     #       37
        400                     #       38
        400                     #       39
        200                     #       40
        200                     #       41
        200                     #       42
        200                     #       43
        200                     #       44
        200                     #       45
        200                     #       46
        200                     #       47
        200                     #       48
        200                     #       49
        100                     #       50
        100                     #       51
        100                     #       52
        100                     #       53
        100                     #       54
        100                     #       55
        100                     #       56
        100                     #       57
        100                     #       58
        100                     #       59
bash-2.05$
```

Planificación en System V R4: clase *system*

- no es accesible en todas las instalaciones
- a ella pertenecen los procesos como *pageout sched fsflush*
- prioridades fijas
- en el rango 60-99

Planificación en unix: llamada *nice()*

- la llamada disponiple en todos los istemas es la llamada *nice()*

```
#include <unistd.h>
int nice(int incr);
```

- modifica el factor nice del proceso
- se le pasa el incremento
- devuelve el factor nice resultante menos 20. El factor nice vale entre 0 y 20, la llamada devuelve entre -20 (máxima prioridad) y 20

Planificación en unix: llamadas *getpriority()* y *setpriority()*

funciones de librería getpriority() y setpriority()

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int priority);
```

- disponibles en la práctica totalidad de los sistemas
- lo que se cambia es el factor *nice*
- interfaz algo más sofisticada que la llamada *nice()* pues un proceso, con las credenciales adecuadas, puede modificar la prioridad de otros procesos

Planificación en unix: llamada *rtprio*

llamada rtprio

```
#include <sys/types.h>
#include <sys/rtprio.h>

int rtprio(int function, pid_t pid, struct rtprio *rtp);
func:RTP_LOOKUP
     RTP_SET

struct rtprio {
    ushort type;
    ushort prio;
    }
type:RTP_PRIO_REALTIME
     RTP_PRIO_NORMAL
     RTP_PRIO_IDLE
prio:0..RTP_PRIO_MAX
```

Planificación en unix: llamada *rtprio*

llamada rtprio

- disponible en HPUX y algonos sistemas BSD (freeBSD, dragonfly ..)
- los procesos `RTP_PRIO_REALTIME` tienen prioridades estáticas y mayores que cualquier otro proceso en el sistema
- los procesos `RTP_PRIO_NORMAL` tienen prioridades dinámicas que se recalculan a partir del factor *nice* y el uso de cpu
- los procesos `RTP_PRIO_IDLE` tienen prioridades estáticas y menores que cualquier otro proceso en el sistema

Planificación en unix: llamadas POSIX

llamadas POSIX

```
int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *p);

int sched_getscheduler(pid_t pid);

int sched_setparam(pid_t pid, const struct sched_param *p);

int sched_getparam(pid_t pid, struct sched_param *p);

int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);

struct sched_param {
          int sched_priority;
      };
```

Planificación en unix: llamadas POSIX

- Se distinguen tres políticas de planificación con distintas prioridades estáticas
  - SCHED_OTHER para los procesos normales, éstos tiene una prioridad estática 0 por lo que solo se ejecutan si no hay ninguno de `SCHED_FIFO` y `SCHED_RR`. Entre ellos se planifican dinamicamente según el uso de cpu y el factor nice
  - SCHED_RR procesos en tiempo real, se planifican por *roundrobin*
  - SCHED_FIFO procesos en tiempo real, se planifican por fifo

Planificación en unix: llamada *priocntl()*

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
long priocntl(idtype_t  idtype, id_t  id, int  cmd, /*arg */...);
/*idtype:*/
P_PID,          /* A process identifier.                 */
P_PPID,         /* A parent process identifier.          */
P_PGID,         /* A process group (job control group)   */
                /* identifier.                           */
P_SID,          /* A session identifier.                 */
P_CID,          /* A scheduling class identifier.        */
P_UID,          /* A user identifier.                    */
P_GID,          /* A group identifier.                   */
P_ALL,          /* All processes.                        */
P_LWPID         /* An LWP identifier.                    */
```

Planificación en unix: llamada *priocntl()*

cmd

PC_GETCID

PC_GETCLINFO

PC_SETPARMS

PC_GETPARMS

parámetros para PC_GETCID y PC_GETCLINFO

```
typedef struct pcinfo {
  id_t  pc_cid;                  /* class id */
  char  pc_clname[PC_CLNMSZ];    /* class name */
  int   pc_clinfo[PC_CLINFOSZ]; /* class information */
} pcinfo_t;

typedef struct tsinfo {
 pri_t  ts_maxupri;  /*configured limits of priority range*/
} tsinfo_t;
typedef struct rtinfo {
 pri_t  rt_maxpri;   /* maximum configured rt priority */
} rtinfo_t;
typedef struct iainfo {
  pri_t  ia_maxupri; /* configured limits of user priority range */
} iainfo_t;
```

Planificación en unix: llamada *priocntl()*

parámetros para PC_GETPARMS y PC_SETPARMS

```
typedef struct pcparms {
  id_t pc_cid;                 /* process class */
  int pc_clparms[PC_CLPARMSZ];/*class  parameters */
} pcparms_t;

typedef struct tsparms {
   pri_t   ts_uprilim;     /* user priority limit */
   pri_t   ts_upri;        /* user priority */
} tsparms_t;

typedef struct rtparms {
   pri_t   rt_pri;         /* real-time priority */
   uint_t  rt_tqsecs;      /* seconds in time quantum */
   int  rt_tqnsecs;/*additional nanosecs in time quant */
} rtparms_t;
typedef struct iaparms {
   pri_t  ia_uprilim;      /* user priority limit */
   pri_t  ia_upri;         /* user priority */
   int    ia_mode;         /* interactive on/off */
   int    ia_nice;         /* present nice value */
} iaparms_t;
```

- las llamadas relacionadas con la creación de procesos en unix:

fork() crea un proceso. El proceso creado es un "klon" del proceso padre, su espacio de direcciones es una réplica del espacio de direcciones del padre, solo se distinguen por el valor devuelto por *fork()*: 0 para el hijo y el pid del hijo para el padre

exec() (*execl(), execv(), execle(), execve(), execlp(), execvp()*) hace que un proceso ya creado ejecute un programa: reemplaza el código (datos, pila..) de un proceso con el programa que se le especifica

exit() termina un proceso

```
if ((pid=fork())==0){ /*hijo*/
  if (execv("./programilla",args)==-1){
     perror("fallo en exec");
     exit(1);
     }
  }
else
  if (pid<0)
     perror("fallo en fork")
  else /*el padre sigue*/
```

tareas de la llamada *fork()*

1. reservar espacio de intercambio
2. asignar *pid* y estructura `proc`
3. inicializar estructura `proc`
4. asignar mapas de traducción de direcciones para el proceso hijo
5. asignar `u_area` del hijo y copiar en ella la del padre
6. actualizar campos necesarios en la `u_area`
7. añadir hijo al conjunto de procesos que comparten región de código del programa que ejecuta el proceso padre
8. duplicar datos y pila del padre y actualizar tablas
9. inicializar contexto hardware del hijo
10. marcar hijo como listo
11. devolver 0 al hijo
12. devolver pid del hijo al padre

optimizaciones de *fork()*

- entre las tareas de fork vistas anteriormente la tarea *duplicar datos y pila del padre y actualizar tablas* supone
  - asignar memoria para datos y pila del hijo
  - copiar datos y pila del padre en datos y pila del hijo
- en muchas ocasiones un proceso recien creado con *fork()* se utiliza para ejecutar otro programa

```
if ((pid=fork())==0){ /*hijo*/
  if (execv("./programilla",args)==-1){
     perror("fallo en exec");
     exit(1);
     }
  }
```

- al hacer una de las llamadas *exec()* el espacio de direcciones se descarta y se asigna uno nuevo
- se ha asignado memoria y se han copiado datos para luego descartarlos

optimizaciones de *fork()*

- hay dos optimizaciones: *copy on write* y la llamada *vfork()*
- **copy on write**
  - no se copian los datos ni la pila del padre: se comparten entre padre e hijo
  - se marcan como de solo lectura
  - al intentar modificarlos, como están marcados de solo lectura se genera una excepción
  - el manejador de esa excepción copia solamente *la página* que se quiere modificar. Solo se duplican las páginas que se modifican
- **llamada vfork()**
  - se utiliza si se espera lamar a *exec()* en poco tiempo
  - el proceso padre *presta* su espacio de direcciones al proceso hijo hasta que el hijo hace *exec()* o *exit()*, momento en que se despierta al proceso padre
  - no se copia nada

ejecución de programas *exec()*

- un proceso ya creado ejecuta un programa; se reemplaza el espacio de direcciones del proceso con el del nuevo programa
  - si el proceso fue creado mediante *vfork()*, *exec()* devuelve el espacio de diecciones al proceso padre
  - si el proceso fue creado mediante *fork()*, *exec()* libera el espacio de direcciones
- se crea un nuevo espacio de direcciones y se carga en él el nuevo programa
- cuando *exec()* termina comienza la ejecución en la primera instrucción del nuevo programa
- *exec()* **NO CREA** un nuevo proceso: el proceso es el mismo, conserva su *pid*, estructura `proc` *u_area*
- si el ejecutable que pretende ejecutarse tiene los permisos adecuados (bits *setuid* y *setgid*), *exec()* cambia las credenciales de usuario y/o de grupo

ejecución de programas *exec()*

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);

int execv(const char *path, char *const argv[]);

int execlp(const char *file, const char *arg, ...);

int execvp(const char *file, char *const argv[])

int execle(const char *path, const char *arg, ...
                              char *const envp[])

int execve(const char *path, char *const argv[],
                              char *const envp[]);
```

tareas de la llamada *exec()*

1. seguir el path y obtener el fichero ejecutable (*namei*)
2. comprobar permiso de ejecución (credencial efactiva)
3. leer cabecera y comprobar que es un ejecutable válido
4. si en los permisos del ejecutable están activados los bits *setuid* y/o *setgid* (–s——— ó ——s—) cambiar la credencial efectiva de usuario y/o de grupo según corresponda
5. salvar argumentos a *exec()* y entorno el el espacio del kernel (el espacio de usuario va a ser destruido)
6. asignar nuevo espacio de intercambio (para datos y pila)
7. liberar espacio previo de direcciones y de intercambio (si el proceso fue creado con *vfork()* devolverselo al padre)
8. asignar nuevo espacio de direcciones. Si el código ya está siendo usado por otro proceso, compartirlo, en caso contrario cargarlo del fichero ejecutable
9. copiar argumentos a *exec()* variables de entorno en la nueva pila de usuario
10. restaurar los manejadores de señales a su acción por defecto
11. inicializar contexto hardware, todos a 0, excepto PC al punto

terminación de un proceso *exit()*

- un proceso puede terminar mmediante la llamada al sistema *exit()*
- *exit()* realiza lo siguiente
  - desactivar las señales
  - cerrar ficheros abiertos por el proceso
  - liberar inodos de la T.I.M. del fichero de codigo y directorios actual y raíz (*iput*)
  - salvar estadísticas de uso de recursos y estado de salida en estructura `proc`
  - cambiar estado a SZOMB y colocar estructura `proc` en lista de procesos *zombies*
  - hacer que *init* herede los hijos de este proceso
  - liberar espacio de direcciones, *u_aera*, intercambio ...
  - enviar al padre SIGCHLD (normalment ignorada por el padre)
  - despertar al proceso padre (si este último en espera por la terminación de un hijo)
  - llamar *swtch* para iniciar cambio de contexto
- nótese que no se desasigna la estructura `proc`, el proceso sigue existiendo, aunque en estado *zombie*

espera por terminación de un proceso hijo *wait()*

- si un proceso necesita conocer si ha terminado un proceso hijo, utiliza una de las llamadas *wait()*

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);

/*POSIX.1*/
pid_t waitpid(pid_t pid, int *stat_loc, int options);

/*BSD*/
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statusp, int options,
            struct rusage *rusage);
pid_t wait4(pid_t pid, int *statusp, int options,
            struct rusage *rusage);

/*System VR4*/
#include <wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

espera por terminación de un proceso hijo *wait()*

- *wait()* mira si hay un hijo que haya terminado
  - si lo hay vuelve inmediatamente
  - si no lo hay deja al proceso que llama a *wait()* en espera hasta que un hijo termine
- *transfiere* el estado que el proceso hijo le pasó a *exit()* a la variable que recibe
- desasigna la estructuta *proc* del hijo
- devuelve el pid del hijo
- las variantes *waitpid()*, *waitid()*, *wait3* y *wait4()* admiten opciones

WNOHANG	no se espera a que el hijo termine
WUNTRACED	se informa también si el hijo para o se reanuda
WNOWAIT	no desasigna estructura `proc` (solaris)

espera por terminación de un proceso hijo *wait()*

- la estructura `proc` no se desasigna hasta que se llama a *wait()*
- *init* hereda hijos *activos*
  - si el proceso padre termina antes que sus procesos hijos, estos son heredados por *init*
  - si los procesos hijos terminan antes que el proceso padre quedarán en estado zombie hasta que el proceso padre haga *wait()*. Como *init* sólo hereda procesos activos, si el padre no hace *wait()* antes de terminar, permanecerán en estado *zombie* reteniendo una estructura`proc`
- puede evitarse la creacion de *zombies* utilizando el flag SA_NOCLDWAIT en la llamada *sigaction*. En ese caso la llamada *wait()* devolveria -1 con el error ECHILD

espera por terminación de un proceso hijo *wait()*

- el valor que se obtenido con `stat_loc` en *wait (int * stat_loc)* se interpreta de la siguiente manera
  - el proceso hijo terminó con *exit()*
    - los 8 bits menos significativos 0
    - los 8 bits más significativos contienen los 8 bits menos significativos del argumento pasado a *exit()*
  - el proceso hijo terminó debido a una señal
    - los 8 bits menos significativos contienen el número de la señal
    - los 8 bits más significativos 0
  - el proceso hijo se paró
    - los 8 bits menos significativos WSTOPFLG
    - los 8 bits más significativos el número de la señal que par'o al proceso hijo

espera por terminación de un proceso hijo *wait()*

Para analizar en estado obtenido con *wait()*, existen las siguientes macros:

```
#include <sys/types.h>
#include <sys/wait.h>

WIFEXITED(status)

WEXITSTATUS(status)

WIFSIGNALED(status)

WTERMSIG(status)

WIFSTOPPED(status)

WSTOPSIG(status)
```

Señales

- el kernel usa señales para notificar a los procesos de sucesos asíncronos. Ejemplos:
  - si se pulsa cntrl-C el kernel envía SIGINT
  - si se corta la comunicación el kernel envía SIGHUP
- los procesos pueden enviarse señales entre ellos mediante la llamada al sistema *kill*
- se responde a las señales al volver a modo usuario. Varias acciones posibles (excepto para SIGKILL y SIGSTOP):
  - terminar el proceso
  - ignorar la señal: no se realiza ninguna acción
  - acción definida por el usuario: manejador
- la acción no se realiza inmediatamente. El kernel pone un bit indicando la señal que ha llegado en el campo correspondiente de su estructura `proc`, si el proceso está en una espera interrumpible, lo saca de la espera (la llamada al sistema que esté en curso devolverá -1 y errno será EINTR). En cualquier caso la señal se manejará al volver a modo usuario

Table 4-1. UNIX signals

Signal	Description	Default Action	Available In	Notes
SIGABRT	process aborted	abort	APSB	
SIGALRM	real-time alarm	exit	OPSB	
SIGBUS	bus error	abort	OSB	
SIGCHLD	child died or suspended	ignore	OJSB	6
SIGCONT	resume suspended process	continue/ignore	JSB	4
SIGEMT	emulator trap	abort	OSB	
SIGFPE	arithmetic fault	abort	OAPSB	
SIGHUP	hang-up	exit	OPSB	
SIGILL	illegal instruction	abort	OAPSB	2
SIGINFO	status request (control-T)	ignore	B	
SIGINT	tty interrupt (control-C)	exit	OAPSB	
SIGIO	async I/O event	exit/ignore	SB	3
SIGIOT	I/O trap	abort	OSB	
SIGKILL	kill process	exit	OPSB	1
SIGPIPE	write to pipe with no readers	exit	OPSB	
SIGPOLL	pollable event	exit	S	
SIGPROF	profiling timer	exit	SB	
SIGPWR	power fail	ignore	OS	
SIGQUIT	tty quit signal (control-\)	abort	OPSB	
SIGSEGV	segmentation fault	abort	OAPSB	
SIGSTOP	stop process	stop	JSB	1
SIGSYS	invalid system call	exit	OAPSB	
SIGTERM	terminate process	exit	OAPSB	
SIGTRAP	hardware fault	abort	OSB	2
SIGTSTP	tty stop signal (control-Z)	stop	JSB	
SIGTTIN	tty read from background process	stop	JSB	
SIGTTOU	tty write from background process	stop	JSB	5
SIGURG	urgent event on I/O channel	ignore	SB	
SIGUSR1	user-definable	exit	OPSB	
SIGUSR2	user-definable	exit	OPSB	
SIGVTALRM	virtual time alarm	exit	SB	
SIGWINCH	window size change	ignore	SB	
SIGXCPU	exceed CPU limit	abort	SB	
SIGXFSZ	exceed file size limit	abort	SB	

Availability:
O Original SVR2 signal
A ANSI C
B 4.3 BSD
S SVR4
P POSIX.1
J POSIX.1, only if job control is supported

Notes:
1 cannot be caught, blocked, or ignored.
2 Not reset to default, even in System V implementations.
3 Default action is to exit in SVR4, ignore in 4.3BSD.
4 Default action is to continue process if suspended, else to ignore. Cannot be blocked.
5 Process can choose to allow background writes without generating this signal.
6 Called SIGCLD in SVR3 and earlier releases.

Señales en System V R2

- hay 15 señales disponibles
- llamadas relacionadas con la señales
  - *kill (pid_t pid, int sig)*: permite a un proceso enviar señales a otro
  - *signal (int sig, void (*handler)(int))*: especifica que se hace al recibir la señal. *handler* puede ser:
    - SIG_DFL: se realiza la acción por defecto asociada a la señal
    - SIG_IGN: la señal está ignorada (no se hace nada)
    - la dirección de la función que se ejecutará al recibir la señal (manejador). Esta función es de tipo *void*
- en la u_area del proceso hay un array, indexado por número de señal, que contiene en cada posicón, la dirección del manejador asociado a dicha señal (o SIG_IGN o SIG_DFL para indicar que una señal está ignorada o asociada a su acción por defecto)

Señales en System V R2

- el envío de la señal supone poner a 1 el bit correspondiente en un campo de la estructura `proc`
- cuando el proceso va a volver a modo usuario, si hay alguna señal pendiente, el kernel limpia el bit; si la señal está ignorada no se realiza acción alguna, en caso contrario el kernel procede de la siguiente manera:
  - crea una capa de contexto en la pila (de usuario)
  - restaura la señal a su acción por defecto
  - establece contador de programa igual a la dirección del manejador, con lo cual lo primero que se ejecuta al volver a modo usuario es el manejador
- los manejadores son, por tanto **NO PERMANENTES**
- además, si llega la señal a mitad de la ejecución de un manejador se realiza la acción que en ese momento esté asociada a la señal

Señales en System V R2

- el siguiente código pordría interrumpirse pulsando 2 veces cntrl-C

```
#include <signal.h>
void manejador ()
{
      printf ("Se ha pulsado control-C\n");
}
main()
{
      signal (SIGINT, manejador);
      while (1);
}
```

- para hacer el handler permanente podemos reinstalarlo.

```
#include <signal.h>
void manejador ()
{
      printf ("Se ha pulsado control-C\n");
      signal (SIGINT,manejador);
}
main()
{
      signal (SIGINT, manejador);
      while (1);
}
```

- el programa tambien terminaría si control-C es pulsado la segunda vez antes de que se ejecute la llamada *signal*.

Señales en System V R3

- introduce las señales fiables
  - manejadores permanentes (no se restaura la acción por defecto una vez recibida la señal)
  - el manejador de uns señal se ejecuta con dicha señal enmascarada
  - posibilidad de enmascarar y desenmascarar señales.
  - la información de la señales recibidas, enmascaradas e ignoradas está ahora en la estructura `proc`
- System V R3 intriduce la siguientes llamadas
  - *`sigset (int senal, void (*handler)(int))`*. Instala un manejador para la señal *senal*. El manejador es permanente y no puede ser interrumpido por la propia señal
  - *`sighold (int senal)`*. Enmascara una señal
  - *`sigrelse (int senal)`*. Desenmascara una señal
  - *`sigpause (int senal)`*. Desenmascara *senal* y deja al proceso en espera hasta que llega una señal.

Señales en BSD

- permite enmascarar señales por grupos (no de una en una. como en System V R3)
- si una llamada es interrumpida por una señal, es reiniciada automaticamente (aunque este comportamiento es configurable mediante *siginterrupt*)
- BSD introduce las sigientes llamadas
  - *sigsetmask(int mask)* Establece el conjunto de señales enmascaradas (dicho conjunto se manipula con *int sigmask(int signum)*)
  - *sigblock(int mask)* Enmascara el conjunto de señales en *mask*
  - *sigpause(int mask)* Establece el conjunto de señales enmascaradas y deja al proceso en espera a que llegue una señal
  - *sigvec(int sig, struct sigvec *vec, struct sigvec *ovec)* Instala un manejador para la señal *sig*
  - *int sigstack(struct sigstack *ss, struct sigstack *oss)* Permite especificar una pila para ejecutar los manejadores de la señales.

Señales en System V R4

- Incluye las funcionalidades de los otros conjuntos de llamadas
- Es el *standard* de los sistemas acutales
- Llamadas
  - `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
  - `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`
  - `int sigsuspend(const sigset_t *mask)`
  - `int sigpending(sigset_t *set)`
  - `int sigaltstack(const stack_t *ss, stack_t *oss)`
  - `int sigsendset(procset_t *psp, int sig)`
  - `int sigsend(idtype_t idtype, id_t id, int sig)`

Señales en System V R4

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)

- establece el conjunto de señales enmascaradas para el proceso dependiendo del valor de *how*
  - SIG_BLOCK las señales en *set* se añaden al conjunto de señales enmascaradas del proceso
  - SIG_UNBLOCK las señales en *set* se quitan del conjunto de señales enmascaradas del proceso
  - SIG_SETMASK las señales en *set* pasan a ser el conjunto de señales enmascaradas del proceso
- *oldset* nos dice como estaba el conjunto
- los conjuntos son del tipo *sigset_t* y para manipularlos DEBEN ser usadas las siguientes funciones

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum)
```

Señales en System V R4

int sigsuspend(const sigset_t *mask)

- establece el conjunto de señales enmascaradas y deja al proceso en espera hasta que llega una señal (que no esté ignorada ni enmascarada).

int sigpending(sigset_t *set)

- indica si se ha recibido alguna señal

sigaltstack(const stack_t *ss, stack_t *oss)

- permite especificar una pila alternativa para la ejecución de los manejadores

int sigsendset(procset_t *psp, int sig)
int sigsend(idtype_t idtype, id_t id, int sig)

- permiten enviar señales de manera más sofisticada que *kill*

Señales en System V R4

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)

- instala un manejador para la señal *sig*
- la estructura sigaction tiene los siguientes miembros
  - **sa_handler** SIG_DFL, SIG_IGN o la dirección de la función manejador
  - **sa_mask** el conjunto de señales enmascaradas DURANTE la ejecución del manejador
  - **sd_flags** condiciona el funcionamiento del manejador. Se construye con un OR bit a bit de los siguientes valores
    - SA_ONSTACK el manejador se ejecuta en la pila alternativa
    - SA_RESETHAND el manejador es temporal (la señal vuelve a su acción por defecto una vez se ejecuta el manejador)
    - SA_NODEFER el manejador se ejecuta con la señal para la cual es manejador NO enmascarada
    - SA_RESTART si la señal interrumpe alguna llamada al sistema, dicha llamada es reiniciada automaticamente
    - SA_SIGINFO, SA_NOCLDWAIT, SA_NOCLDSTOP, SA_WAITSIG

Señales en System V R4

El siguiente código es un bucle infinito si se pulsa ctrl-C antes de 5 segundos

```
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
void manejador (int s)
{
        static int veces=0;
        printf ("Se ha recibido la SIGINT (%d veces) en %p\n",++veces,&s);
        kill (getpid(),s);
}
int InstalarManejador (int sig, int flags, void (*man)(int))
{
    struct sigaction s;
        sigemptyset(&s.sa_mask);
        s.sa_flags=flags;
        s.sa_handler=man;
        return (sigaction(sig,&s,NULL));
}
main()
{
        InstalarManejador (SIGINT,0, manejador);
        sleep(5);
}
```

Señales en System V R4

El siguiente código desborda la pila si se pulsa ctrl-C antes de 5 segundos

```
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
void manejador (int s)
{
    static int veces=0;
        printf ("Se ha recibido la SIGINT (%d veces) en %p\n",++veces,&s);
        kill (getpid(),s);
}
int InstalarManejador (int sig, int flags, void (*man)(int))
{
    struct sigaction s;
        sigemptyset(&s.sa_mask);
        s.sa_flags=flags;
        s.sa_handler=man;
        return (sigaction(sig,&s,NULL));
}
main()
{
        InstalarManejador (SIGINT,SA_NODEFER, manejador);
        sleep(5);
}
```

Señales en System V R4: implementación

- en la `u_area`
  - `u_signal[]` array de los manejadores
  - `u_sigmask[]` máscara con cada manejador
  - `u_sigaltstack` pila alternativa
  - `u_sigonstack` señales cuyo manejador se ejecuta en la pila alternativa
  - `u_oldsig[]` señales que exhiben el comportamiento antiguo
- en la estructura `proc`
  - `p_cursig` señal que está siendo manejada
  - `p_sig` señales pendientes
  - `p_hold` señales enmascaradas
  - `p_ignore` señales ignoradas

Comunicación entre procesos

Los principales mecanismos de comunicación entre procesos

- ficheros tuberías (*pipes*)
- memoria compartida
- semáforos
- colas de mensajes

Comunicación entre procesos: *pipes*

- son ficheros temporales. Se crean con la llamada *pipe()*

```
#include <unistd.h>
int pipe(int fildes[2]);
```

- la llamada devuelve dos dscriptores (fildes[0] y fildes[1]) que pueden usarse con las llamadas read y write.
- en algunos sistemas fildes[0] es para lectura y fildes[1] para escritura
- si el *pipe* está vacio la llamada *read()* queda en espera y si está lleno la llamada *write()* queda en espera
- los datos desaparecen del *pipe* a medida que se van leyendo

Comunicación entre procesos: *memoria compartida*

Dado que los recursos IPC son compartidos por varios procesos, es necesario que distintos procesos puedan referirse al mismo recurso: todos los recursos IPC se identifican en el sistema por un número.

1. es necesario obtener el bloque de memoria compartida (bien creándolo, o bien usando uno ya creado)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

   - **key**: número que identifica el recurso en el sistema
   - **size**: tamaño de la zona de memoria compartida (existe un mínimo)
   - **shmflg**: or bit a bit de `IPC_CREAT`, `IPC_EXCL` y los permisos
     - `IPC_CREAT`: si no existe el recurso lo crea
     - `IPC_EXCL` usado en conjunto con `IPC_CREAT`: si el recurso existe produce un error

Comunicación entre procesos: *memoria compartida*

2 una vez creada, para poder acceder a ella hay que colocarla en el espacio de direcciones del proceso, a partir de ese momento es accesible como el resto de la memoria del proceso

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg)
```

- **shmid**: identificador obtenido con *shmget()*
- **shmaddr**: dirección de memoria virtual donde se quiere colocar la memoria compartida (NULL para que la asigne el sistema)
- **shmflg**: `SHM_RND`, `IPC_RDONLY`, `SHM_SHARE_MMU` (Solaris) `SHM_REMAP` (linux)

shmat() devuelve la dirección de memoria donde se encuentra la región de memoria compartida

Comunicación entre procesos: *memoria compartida*

3. cuando no se necesita acceder a él puede desencadenarse del espacio de direcciones del proceso

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

4. finalmente existe una llamada de control que permite, entre otras cosas, eliminar del sistema una zona de memoria compartida

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

   - **shmid**: identificador obtenido con *shmget()*
   - **cmd**: acción a realizar: `IPC_RMID`, `SHM_LOCK`, `SHM_UNLOCK`, `IPC_STAT`, `IPC_SET` ...
   - **buf**: paso de información

Comunicación entre procesos: *semáforos*

Lo semáforos sirven para sincronizar procesos, el interfaz system V ipc, proporciona arrays de semáforos

1. como en el caso de la memoria compartida, primero es necesario obtener el recurso (bien creándolo, o bien usando uno ya creado)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int shmflg);
```

   - **key**: número que identifica el recurso en el sistema
   - **nsems**: número de semáforos en el array
   - **shmflg**: or bit a bit de `IPC_CREAT`, `IPC_EXCL` y los permisos
     - `IPC_CREAT`: si no existe el recurso lo crea
     - `IPC_EXCL`: usado en conjunto con `IPC_CREAT` si el recurso existe produce un error

Comunicación entre procesos: *semáforos*

2 una vez creado el array de semáforos se pueden hacer operaciones sobre uno o varios de los semáforos que componen el array

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops)
```

- **semid**: identificador obtenido con *semget()*
- **sops**: puntero a estructuras sembuf, cada una de las cuales representa una operación a realizar en el array

```
struct sembuf {
  ushort_t  sem_num;   /* semaphore # */
  short     sem_op;    /* semaphore operation */
  short     sem_flg;   /* operation flags */
};                     /* SEM_UNDO, IPC_NOWAIT*/
```

- **nsops**: número de operaciones a realizar

Comunicación entre procesos: *semáforos*

3 finalmente existe una llamada de control que permite, entre otras cosas, eliminar del sistema un array de semáforos, inicializarlos, ...

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

- **shmid**: identificador obtenido con *semget()*
- **semnum**: número de semáforo sobre el que realizar la acción
- **cmd**: acción a realizar: `IPC_RMID`, `IPC_STAT`, `IPC_SET`, `GETALL`, `SETALL`, `GETVAL`, `SETVAL` ...
- **cuarto argumento**: paso de información

```
union semun {
  int    val;
  struct semid_ds *buf;
  ushort_t        *array;
 } arg;
```

Comunicación entre procesos: *colas de mansaje*

Con la cola de mensajes se puede enviar información entre procesos de forma mas sofisticada que mediante el *pipe*
Un mensaje es cualquier bloque de información que se envía conjuntamente. No tiene formato predefinido.
Los primeros 32 bits del mensaje representan el *tipo* de mensaje.
La llamada que recibe mensajes puede especificar el *tipo* del mensaje que quiere recibir.

1. como en el caso de la memoria compartida y los semáforos, primero es necesario obtener el recurso (bien creándolo, o bien usando uno ya creado)

```
#include <sys/msg.h>


int msgget(key_t key, int shmflg);
```

   - **key**: número que identifica el recurso en el sistema
   - **shmflg**: or bit a bit de `IPC_CREAT`, `IPC_EXCL` y los permisos
     - `IPC_CREAT`: si no existe el recurso lo crea
     - `IPC_EXCL` usado en conjunto con `IPC_CREAT`: si el recurso existe produce un error

Comunicación entre procesos: *colas de mensajes*

2 una vez creada la cola de mensajes se pueden enviar y recibir mensajes

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);
int msgrcv(int msqid, void *msgp, size_t msgsz,
           long msgtyp, int msgflg);
```

- **msqid**: identificador obtenido con *msgget()*
- **msgp**: puntero al mensaje
- **msgsz**: tamaño del mensaje en *msgsnd()* o número máximo de bytes a transferir en *msgrcv*
- **msgtyp**: tipo de mensaje que se quiere recibir
  - 0 el primero en la cola
  - -n el primero del tipo de mensaje mas bajo que sea menor o igual que n
  - n el primero de tipo n
- **msgflg**:`IPC_NOWAIT`, `MSG_EXCEPT` (linux), `MSG_NOERROR` (linux)

Comunicación entre procesos: *colas de mensajes*

3 finalmente existe una llamada de control que permite, entre otras cosas, eliminar del sistema una cola de mensajes, obtener información ...

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd,struct msqid_ds *buf);
```

- **msqid**: identificador obtenido con *msgget()*
- **cmd**: acción a realizar: `IPC_RMID`, `IPC_STAT`, `IPC_SET`
- **buf**: paso de información