Processes

Operating Systems
Grado en Informática.
Departamento de Computación
Facultad de Informática
Universidad de Coruña

Contents I

Contents II

Contents III

Contents IV

Contents V

Processes

What is a process?

- A key concept in an operating system is the idea of process.
- A program is a set of instructions. For example: the command `ls` is an executable file in some system directory. typically `/bin` or `/usr/bin`.

Definition

A process is an abstraction that refers to each execution of a program.

- IMPORTANT: a process has not got to be always executing.
  A process life cycle has several stages, *execution* is one of them.

Concept of process

- We can think of a process as
  - each instance of an executing program
  - the entity the O.S. creates to execute a program
- A process consists of
  - Address space: the set of addresses the process may reference
  - Control point: next instruction to be executed
    - Some systems allow for a process to have more than one control point: this process is executing concurrently at various places within itself, this is what we call threads.

Concept of process

- Example: open two terminals and execute `ls -lR /` in one and `ls -lR /usr` in the other. We have two processes executing the same program `/bin/ls`.
- In this process execution is sequential.
- Concurrency: these two processes seem to be executing simoultaneously. Actually, the CPU keeps changing between them (just like our cheff does). This is what we call multitasking.
- At any given instant ONLY ONE OF THEM is actually executing
- Should we have serveral CPUs, we could have real parallel execution. But each CPU can only be executing one process at a time. Usually, number of processes > number of CPUs.
- There are processes that are also internally concurrent. They use *threads* of execution.

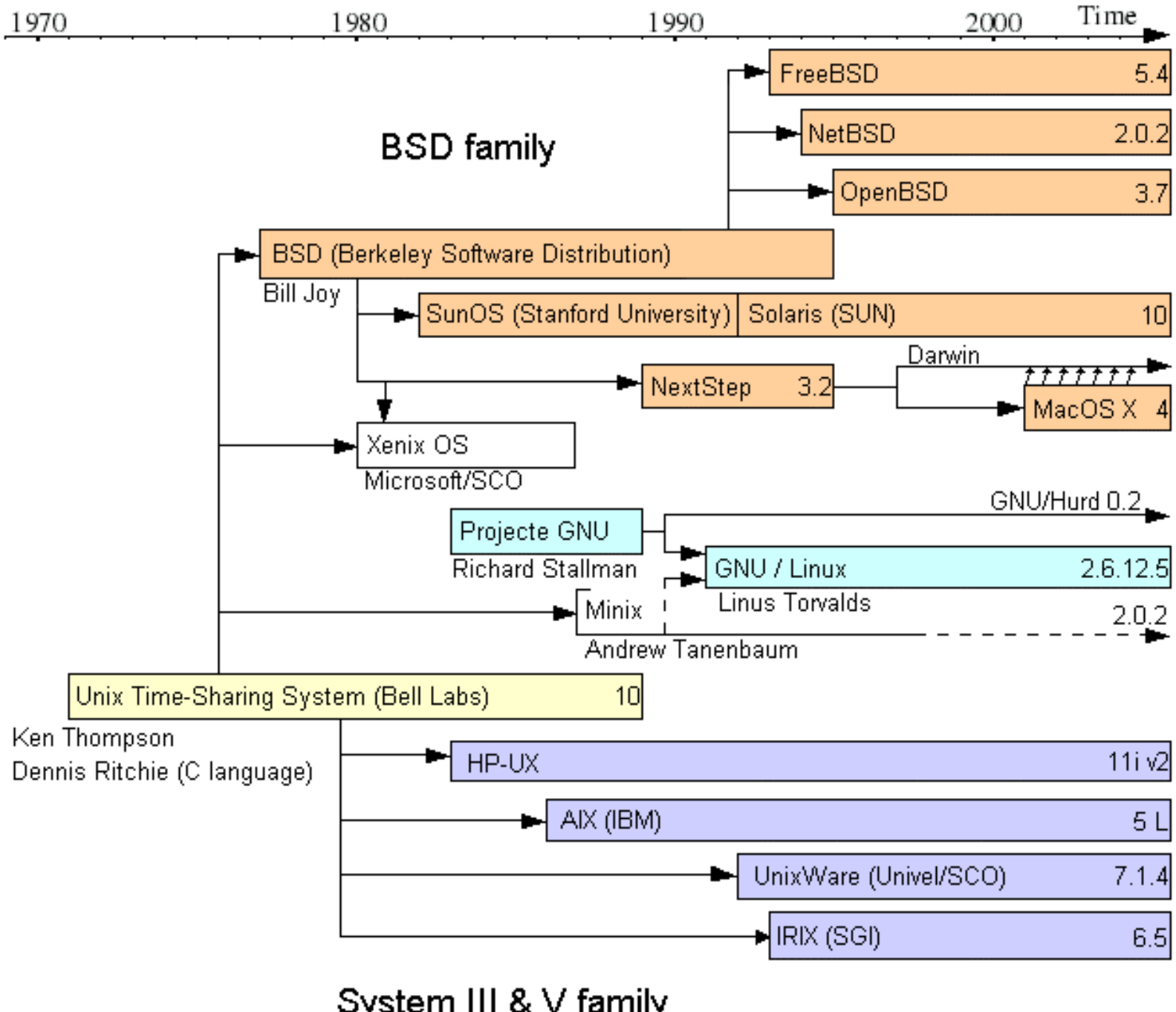
Concept of process

- In its simplest form, a process address space has three regions
  - **code** Code of all functions in the program that the process is running
  - **data** Golbal variables of the program. Dynamically assigned memory (*heap*) in usually a part of this region.
  - **stack** Used for parameter passing and to store the return address once a function is called. It is also used by the function being called to store its local variables.
- Actually there are more regions: shared memory regions, dynamically linked libraries, mapped files ...
- Unix command `pmap` shows us the address space of a process. This space has holes in it and the addresses shown are virtual (logical addresses of the process, not real physical addresses)

Processes in UNIX

What is UNIX?

- The term UNIX is a generic term that refers to many different O.S.s
- There are many *flavours* of unix: some commercial and some free (linux, solaris, aix, freeBSD . . .)
- Each of them may, or may not, comply to different standards, which apply to the funcionality, implentation or interface with the O,S.
- The main standards are
  - System V
  - BSD
  - POSIX


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

What is the kernel?

- The kernel is the term with which we usually refer to the Operating System itself
- The kernel resides in a file (/unix, /vmunix, /vmlinuz, /bsd, /vmlinuz, /kernel.GENERIC..) which gets loaded by the boot loader during the machine bootstrap procedure
- The kernel, initializes the system and creates the environment to execute processes. It creates some processes, which, in turn, will create the rest of the processes in the system
- INIT (pid 1) is the first user process and the ancestor of every user process in the system
- UNIX (kernel) interacts with the hardware
- User processes interact with the kernel using the system call interface

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 -
root 107    1  107 c085cd80    0 Is    ??  /usr/sbin/sy
root 112    1  112 c0874500    0 Is    ??  mountd -r
root 115    1  115 c0874600    0 Is    ??  nfsd: master
root 117  115  115 c0874600    0 I     ??  nfsd: server
```

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 ?     keven
1 S   0     3    1  0  79  19 -     0 ksofti ?     ksoft
1 S   0     4    1  0  69   0 -     0 kswapd ?     kswap
1 S   0     5    1  0  69   0 -     0 bdflus ?     bdflu
1 S   0     6    1  0  69   0 -     0 kupdat ?     kupda
4 S   0   229    1  0  67  -4 -   369 select ?     udevd
1 S   0   375    1  0  69   0 -     0 down_i ?     knode
1 S   0   492    1  0  69   0 -     0 ?      ?     khubd
1 S   0  1571    1  0  69   0 -   561 select ?     syslo
5 S   0  1574    1  0  69   0 -   547 syslog ?     klogd
1 S   0  1592    1  0  69   0 -   637 select ?     dirmn
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

What does the kernel do?

- Unix kernel is the only program to run directly on the system hardware
- User processes do not interact directly with the hardware but use the system call interface instead
- Unix kernel also receives requests from de external devices via interrupts

kernel mode and user mode

Operating systems need more than one running mode to operate: *kernel mode* and *user mode*

- **user mode** user code runs in this mode
- **kernel mode** kernel runs in this mode
  1. **system call:** A user process explicitly requests some service from the kernel via the system call interface
  2. **Exceptions:** Exceptional situations (division by 0, addressing errors . . .) cause hardware traps that require kernel intevention
  3. **Interrrupts:** Devices use interrupts to notify the kernel of certain events (i/o completion, change in the state of a device . . .)
- Some processor instructions can only be executed when running in kernel mode

Executing in kernel mode: examples

- The `time` command shows CPU times (in both user and kernel mode)
- Let's consider the following program

```
main()
{
  while (1);
}
```

- When running for 25 seconds, `time` shows

```
real 0m25.417s
user 0m25.360s
sys 0m0.010s
```

Executing in kernel mode: examples

- With the following *getpid* loop

```
main()
{
  while (1)
    getpid();
}
```

- When running for 25 seconds, `time` shows

```
real 0m24.362s
user 0m16.954s
sys 0m7.380s
```

Executing in kernel mode: examples

- Porcess sending a signal to itself

```
main()
{
  pid_t pid=getpid();

  while (1)
     kill (pid, 0);
}
```

- When running for 25 seconds, `time` shows

```
real 0m25.434s
user 0m11.486s
sys 0m13.941s
```

Executing in kernel mode: examples

- Sending SIGINT to the **init** process

```
main()

{
  while (1)
     kill (1, SIGINT);
}
```

- After executing for 25 seconds `time` shows

```
real 0m25.221s
user 0m9.199s
sys 0m16.014s
```

Executing in kernel mode: examples

- We produce an addressing exception (by dereferencing a NULL pointer) (we have installed a handler for SIGSEGV)

```
void vacio(int sig)
{
}
main()
{
  int *p;

  sigset(SIGSEGV,vacio);
  p=NULL;
  *p=3;
}
```

Executing in kernel mode: examples

- After executing for 25 seconds `time` shows

```
real 0m25.853s
user 0m10.331s
sys 0m15.509s
```

Executing in kernel mode: examples

- We repeat the first example but we move the mouse and press the keyboard at the same time

```
main()
{
  while (1);
}
```

- After executing for 25 seconds `time` shows

```
real 0m25.453s
user 0m25.326s
sys 0m0.039s
```

Executing in kernel mode: examples

- Let's consider now the following program

```
main()
{
  struct timespec t;
  t.tv_sec=0;t.tv_nsec=1000;  /*1 milisegundo*/

  while (1)
    nanosleep (&t, NULL);
}
```

- After running for 25 seconds, `time` shows

```
real 0m25.897s
user 0m0.006s
sys 0m0.022s
```

Threads and processes

- In a traditional Unix system a process is defined by
  - **address space**: Set of memory addresses the process can reference
  - **control point**: Indicates which is the next istruction to execute (Program Counter)
- In a modern Unix system a process can have several *control points* (*threads*)
  - *threads* share address space

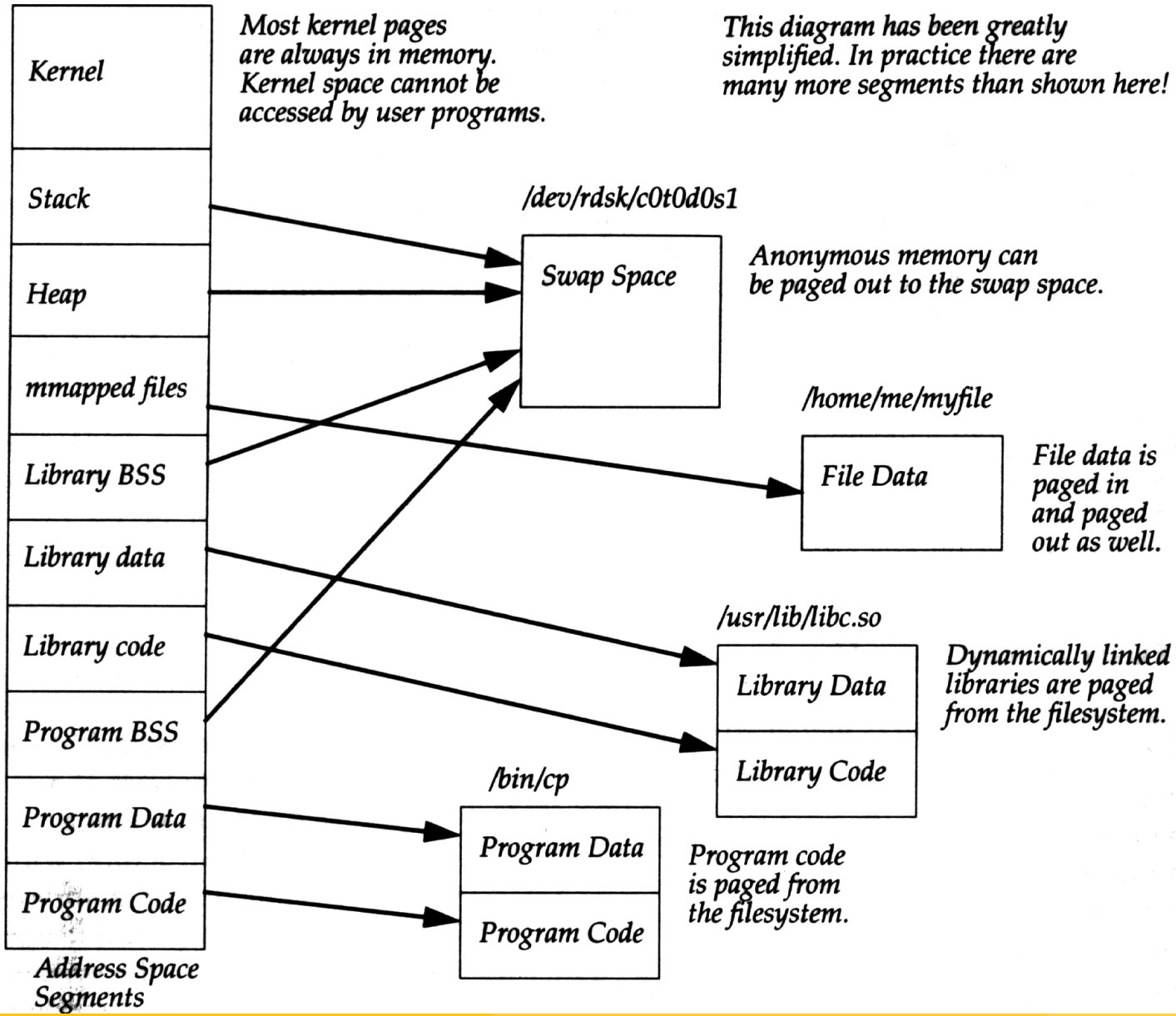
Address space

- Processes use **virtual addresses**. A part of their virtual address space corresponds to the kernel code and data. It is called *system space* or *kernel space*
- *system space* can only be reached when in kernel mode
- The kernel keeps
  - Global data structures
  - Per process data structures
- Currently running process address space is directly accesible because the MMU registers point to it

Linux 32 but memory map

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

Sparc Solaris memory map



kernel unix

- Unix kernel is a C program, and as such, it has
  - **kernel code**: what is run when the system is executing in kernel mode: system calls code, interrupt and execption handlers
  - **kernal data**: global variables of the kernel, accesible for all the pocesses in the system (process table, inode table, open file table ...)
  - **kernel stack**: part of memory used as stack when executing in kernel mode: parameter passing inside the kernel, local variables of kernel functions ...

Reentrant kernel

- unix kernel is reentrant
  - Several processes can be running simoultaneously several kernel functions
  - Several processes can be running simoultaneously the same kernel function
- For the kernel to be reentrant:
  - **kernel code** must be read only
  - **kernal data** (global kernel variables) must be protected from concurrent access
  - Each process has its own **kernel stack**

Reentrant kernel: Kernel data protection

- Traditional approach (*non preemptible kernel*)
  - A process running in kernel mode can not be preempted, it only leaves the CPU if it ends, blocks or returns to user mode
  - Only certain kernel data structures need to be protected (the ones that might be used by processes that block)
  - Protecting these structures is simple, just a flag in use/not in use
- Modern approach (*preemptible kernel*)
  - A process running in kernel mode can be preempted if a higher priority process apears ready
  - **ALL** kernel data structures must be protected by more sophisticated means (for example, semaphores)
- More complex mechanisms are needed in multiprocessor systems

Data structures

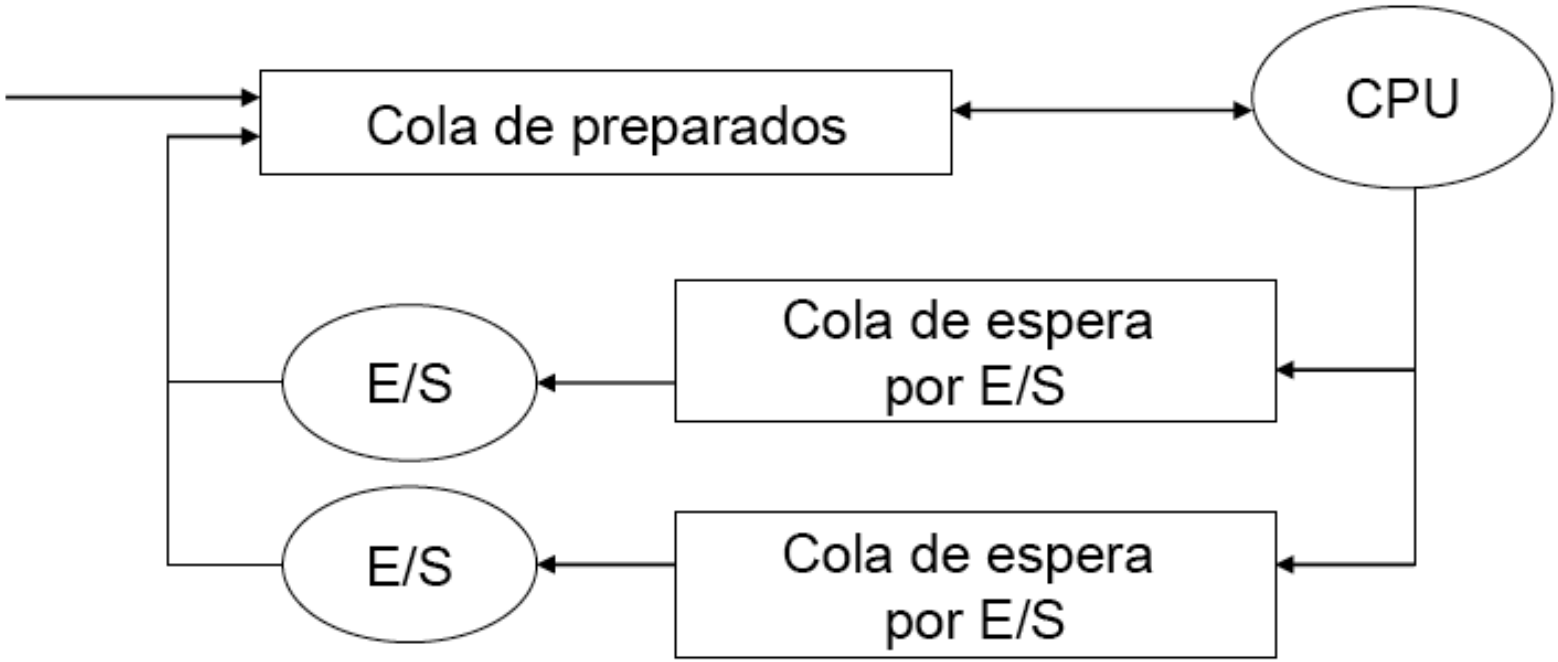
Data structures

What does the O.S. need to manage processes?

- First of all it has to assign memory for the program to be loaded
- As several processes can run the same program, the O.S. has to identify them: Process Descriptor or Process Identifier
- When the process is not being executed, the O.S. needs to keep execution information: registers, memory, resources.

Data Structures

- The O.S. needs to know the list of processes in the system and the state in which each of them is. Usually, it uses lists of Process Descriptors, one for each state or even one for each i/o device.



Let's see some typical O.S. structures...

System Control Block

Some data common for all the processes in the system

Definition (System Control Block)

(SCB) is a set of data structures used by the O.S. to control the execution of processes in the system.

It usually includes:

- List of all Proccess Descriptors.
- Pointer to the process currently in CPU (its Process Descriptor).
- Pointer to lists of processes in different states: list of runnable processes, list of i/o blocked processes...
- Pointer to a list of resource descriptors.
- References to the hardware and software interrupt routines and to the error handling routines.

System Control Block

The O.S. runs when

- An (exception) ocurs. (Some process tries to do something it can't: divide by 0, illegal reference to memory...)
- Some process asks the O.S. to do something (system call) (Example: open a file, create a process...)
- Some externat device requires atention (interrupt) (Example: a disks finishes a pending write request)
- Periodically (clock interrupt)

Process Control Block

- Th O.S. has a table of processes where it keeps information of each process in the system.
- Each entry in this table has information on ONE PROCCESS
- The Process Control Block keeps the data relevant to one process that the OS. uses to manage it (PCB)

Process control block

The PCB usually includes:

- Identification:
  - Process identifier
  - Parent process identifier (if the O.S supports process hierarchy)
  - user and group identifiers
- Scheduling:
  - state of the process
  - If the process is blocked, event the process is waiting for
  - Scheduling parameters: process priority and other information relevant to the scheduling algorithm

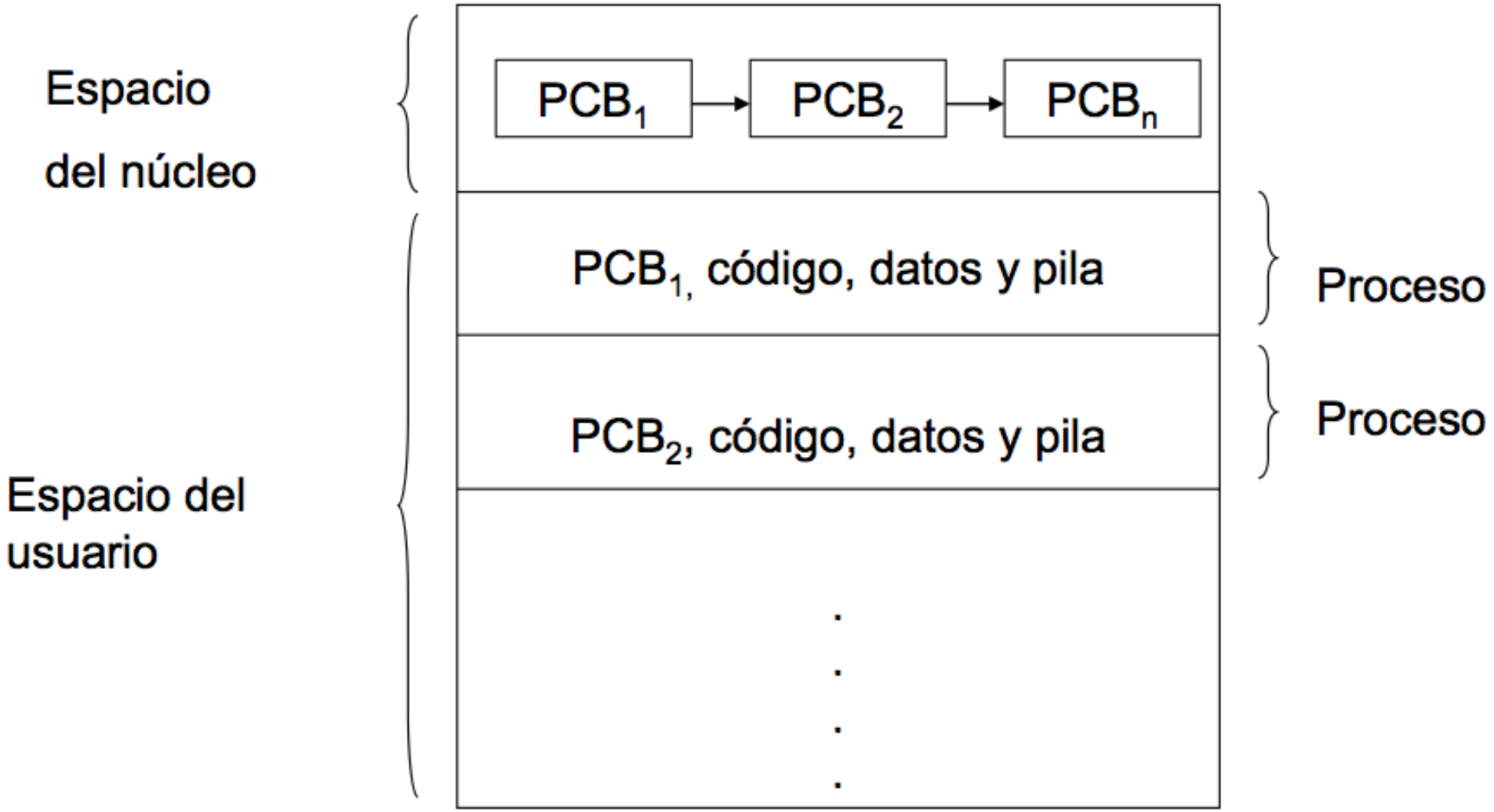
⋮

Process Control Block

⋮

- References to the assigned memory regions:
  - data region
  - code region
  - stack region
- assigned resources:
  - open files: file descriptor table or “file handlers” table.
  - assigned communication ports
- Pointers to arrange the processes in lists.
- Inter-process communication information: message queues, semaphores, signals . . . .

Process Control Block



Data structures in UNIX

Implementing a process I

- To implement the concept of process, unix uses some concepts and structures that allow a program to execute
  - **User address space**. Consists of code, data, stack, shared memory regions, mapped files . . .
  - **Control information**.
    - `proc` structure
    - `u_area`
    - kernel stack
    - address translation maps
  - **credentials**. Indicate which user is behind the execution of that process.
  - **environment variables**. An alternate method to pass information to the process
  - **hardware context**. The contents of the hardware registers (PC, PSW, SP, FPU and MMU registers . . .). When a context switch happens these are stored in a part of the `u_area` called PCB (Process Control Block)

Implementing a process II

- Some of these entities, although conceptually different share implementation: for example, the *kernel stack* of a process is usually implemented as part of the `u_area`, and the credentials go in the `proc` structure
- In linux, instead of `u_area` and `proc` structure, there exists the `task_struct` struct

proc structure

- The kernel keeps an array of proc structures called *process table*
- It is in the kernel space
- The proc structure of a process is always directly accesible, even when the process is not in CPU
- It contains the information on the process that the kernel needs at all times

Some relevant information in the proc structure

- Process IDentifier
- Location of the u_area (address maps)
- Process state
- Pointers to ready queues, wait queues ...
- Priority and related information
- *sleep channel*
- Information on signals (masks)
- Information on memory management
- Pointers to keep the proc sturcture into avaliable queues, zombie queues ...
- Pointers to hash queues based on PID
- Hierarchy information
- flags

u_area

- It is in user space but it is only accesible when in kernel mode and when the process is in CPU
- Always at the same virtual address
- Contains information only needed when the process is running

Some relevant information in the u_area

- PCB
- pointer to `proc` structure
- Parameters to, and return values from system calls
- Information on signals: handlers
- UFDT (User File Descriptor Table)
- Pointers to *vnodes* of `root` directory, current working directory and associated terminal
- Kernel stack for the process

Credencials

- Credentials of a process allow the system to determine what privileges a process has relating to files and to other processes in the system
  - Each user in the system is identified by a number: *user id* or *uid*
  - Each group in the system is identified by a number: *group id* or *gid*
  - There's a special user in the system: *root* (*uid=0*)
    - Can access all files
    - Can send signals to every process
    - Can make privileged system calls

Accessing to file

- Access to a file is conditioned by:
  - Owner: (file *uid*)
  - Group: (file *gid*)
  - Permissions: (file *mode*)

Credencials

- A process has its **credentials**, which specify what files it can access (and how) and what processes it can send signals to (and from what processes it can get signals sent)
  - User credential (process *uid*)
  - Group credential (process *gid*)
- When a process tries to access to a file, the following procedure applies
  - If the process uid matches the file uid: owner premissions apply
  - If the process gid matches the file gid: group permissions apply
  - Otherwise *rest of the world* permissions apply

Credentials

- A process has actually three pairs of credentials: real, effective and saved
  - effective: rule access to files
  - real and effective: rule sending and receiving signals: a signal is received if the real or effective *uid* of the sending process matches the real *uid* of the receiving process
  - real and saved: rule what changes to the effective credential can be done via the *setuid* and *setgid* system calls

Change of credentials

- Only **three** system calls can change a process credentials
  - *setuid()* Changes the *uid* of the calling process
  - *setgid()* Changes the *gid* of the calling process
  - *exec()*: Executes a program

Change of credentials

- *setuid()* Changes the effective credential of the calling process.
  - The only changes allowed are *effective:=real* or *effective:=saved*.
  - If the process has the effective credential of `root`, *setuid* changes the three credentials
- *setgid()* Analogous to *setuid()* but for the group credentials

Change of credentials

- The *exec* system calls (`execl, execv, execlp, execve...`) can change the credentials of the calling process if the file to be executed has the adecuate premissions
  1. *exec()* on an executable file with mode **s******, changes the effective and saved *uid* of the calling process to that of the file being executed
  2. *exec()* on an executable file with mode *****s***, changes the effective and saved *gid* of the calling process to that of the file being executed

Credentials: example

- The following example uses the *setuid()* system call to allow copying of files (and directories) betweenn two different unix acounts, both having only rwx—— (0700) permisions
- For that executable to work properly it must
  - be owned by the user we are copying files from
  - have the setuid bit set (rwsr-xr-x, 04755)
  - be executed by the user copying the files

Credentials: example I

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <libgen.h>
#include <pwd.h>

/***************ReaglaFicheros.c**************************/
/*Copia ficheros y/o directorios de una cuenta a  a otra*/
/*usando ambas  credenciales*/
/*el ejecutable debe ser setuid del que deja copiar de su cuenta*/

#define MAXNAME        1024                    /*longitud maxima de un nombre*/
#define BASEDIR        "/"                     /*por si queremos limitar la parte del arbol de dir

int EsDirectorio (char * dir)
{
  struct stat s;
  if (lstat(dir,&s)==-1)
     return 0;
  return (S_ISDIR(s.st_mode));
}
```

Credentials: example II

```
int EsFichero (char * dir)
{
  struct stat s;
  if (lstat(dir,&s)==-1)
     return 0;
  return (S_ISREG(s.st_mode));
}

ssize_t TamanoFichero(char * f)
{
   struct stat s;
   if (lstat(f,&s)==-1)
      return -1;
   return (ssize_t) s.st_size;
}

ssize_t CopyFile (char * source, char *dest, pid_t euid, pid_t ruid, int verb)
{
  void * buff;
  ssize_t siz,n1,n2;
  int err, df1, df2;

  if (verb)
     printf ("    ------------------Copiando fichero %s -> %s\n", source,dest);

  /*se supone que se llama con credencial efectiva la del origen, euid*/
  if ((siz=TamanoFichero(source))==-1 || (df1=open(source, O_RDONLY))==-1)
```

Credentials: example III

```
      return -1;

   setuid (ruid);      /*cambiamos a la real, destino*/
   if ((df2=open(dest,O_WRONLY |O_CREAT |O_EXCL,0777))==-1){
      err=errno; close (df1); errno=err;
      return -1;
   }
   if ((buff=malloc (siz))==NULL){     /*una vez abierto, la credencial no influye en la lectura o e
      err=errno; close(df1); close(df2); errno=err;
      return -1;
   }
   if ((n1=read(df1,buff,siz))==-1 || (n2=write (df2,buff,n1))==-1){
      err=errno; close (df1);close(df2); free(buff); errno=err;
      return -1;
   }
   close (df1);close(df2); free(buff);
   setuid (euid);                /*volvemos a la credencial efectiva origen*/
   return n2;
}

char * CheckValidAccess (char * dir)          /*comprueba que  no hay .. en el path*/
{
    static char aux[MAXNAME];
    char *tr[MAXNAME/2];
    int i=1;

    if (dir==NULL)
       return NULL;
```

Credentials: example IV

```
   strncpy (aux,dir, MAXNAME);
   if ((tr[0]=strtok(aux,"/"))==NULL)
      return 0;
   while ((tr[i]=strtok(NULL,"/"))!=NULL)
      i++;
   for (i=0; tr[i]!=NULL; i++)
      if (!strcmp(tr[i],".."))    /*algun componente es ".." */
         return NULL;
   if (!strcmp(BASEDIR,"/"))
      strncpy (aux,dir,MAXNAME);
   else
      snprintf (aux,MAXNAME,"%s/%s",BASEDIR,dir);
   return aux;
}

int CopyDirectory (char * source, char *dest, pid_t euid, pid_t ruid,int rec, int verb)
{
    DIR *p;
    struct dirent *d;
    int res,err;
    char aux[MAXNAME], aux2[MAXNAME];

    if (verb)
        printf ("***************Copiando directorio %s -> %s\n", source,dest);

    if ((p=opendir(source))==NULL)    /*se supone que se llama con credencial efectiva la del orig
       return -1;
```

Credentials: example V

```
    if (setuid (ruid)==-1 || mkdir (dest,0777)){ /*cambiamos a la real, destino*/
       err=errno; closedir (p); errno=err;
       return -1;
    }
    setuid (euid);   /*volvemos a la credencial efectiva origen*/
    while ((d=readdir(p))!=NULL){
       if (!strcmp(d->d_name,".") || !strcmp(d->d_name,".."))
          continue;

       snprintf (aux,MAXNAME,"%s/%s",source,d->d_name);snprintf(aux2,MAXNAME,"%s/%s",dest,d->d_nam

       if (EsFichero (aux))       /*los enlaces simbolicos no se siguen*/
          res=CopyFile(aux,aux2,euid,ruid,verb);
       if (rec && EsDirectorio(aux))     /*solo se copia los ficheors y los directorios si es recu
          res=CopyDirectory(aux,aux2,euid,ruid,rec,verb);  /*no los dispositivos ni los fifos...*/
       if (res==-1)
          printf("Error copiando %s->%s:%s\n", aux,aux2,strerror(errno));
    }
    setuid (euid);                          /*volvemos a la credencial efectiva origen*/
    return closedir(p);
}

void Usage(char *ejec){
    printf ("uso: %s [-r] [-v] origen destino\n",ejec);
    printf ("     copia origen en destino, -r para recursivo, -v para verboso \n");
    printf ("     si no se especifica destino, copia en directorio actual\n");
    printf ("     destino especificado a partir de (y limitado por)  %s\n",BASEDIR);
    printf ("     directorio destino no debe existir previamente\n");
```

Credentials: example VI

```
}

void CheckCredentials (pid_t euid, pid_t ruid)
{
  if (euid==ruid){
     printf ("Para copiar con la misma credenciar real y efectiva usar el comando cp\n");
     exit (1);
  }
  if (euid==0 || ruid==0){
     printf ("Este programa no funciona con el root\n");
     exit (1);
  }
}

char * Nombre(uid_t u)
{
  struct passwd *p=getpwuid(u);

  if (p!=NULL)
     return p->pw_name;
  return "???????";
}

int main (int argc, char *argv[])
{
  int i,pos=1,res=-1,recurse=0,verbose=0;
  char *source=NULL, dest[MAXNAME];
  uid_t ruid=getuid(), euid=geteuid();
```

Credentials: example VII

```
for (i=1; argv[i]!=NULL; i++)
   if (!strcmp(argv[i],"-r")) {recurse=1; pos++;}     /*copia recursiva */
   else if (!strcmp(argv[i],"-v")) {verbose=1;pos++;}   /*imprime info*/
   else break;      /*ni -r ni -v, i contienen el directorio origen */

if ((source=CheckValidAccess(argv[pos]))==NULL){      /*no se especifica que copiar o no es váli
   Usage(argv[0]);
   exit (1);
}
CheckCredentials (euid, ruid);

if (argv[pos+1]!=NULL)
   strcpy(dest,argv[pos+1]);
else
      snprintf(dest,MAXNAME,"%s/%s",".",basename(source));

if (verbose){
   printf ("Copiando %s -> %s\n", source,dest);
   printf ("  con credencial efectiva %d (%s) ", euid,Nombre(euid));
   printf (" real %d (%s)\n",ruid, Nombre(ruid));
}
if (EsDirectorio(source))
   res=CopyDirectory(source,dest,euid,ruid,recurse,verbose);  /*no los dispositivos ni los fifos
if (EsFichero (source))       /*los enlaces simbolicos no se siguen*/
   res=CopyFile(source,dest,euid,ruid,verbose);
if (res==-1)
   printf("Error al intentar copiar %s -> %s: %s\n", source,dest,strerror(errno));
```

Credentials: example VIII

```
  return 0;
}
```

Environment variables

- Strings of characters
- Usually in the form "VARIABLENAME=value"
- At the bottom of the user stack
- Several ways to access them
  - Third argument to *main()*: NULL terminated array of the environment variables
  - `extern char ** environ`: NULL terminated array of the environment variables
  - Library functions. *putenv(), getenv(), setenv(), unsetenv()*

Environment variables

The following examples show both the code and output of several programs

- Example 1
  1. Shows the command line arguments
  2. Shows the enviroment variables reached through *main* third argument
  3. Shows both the value and the storing address for main's third argument and the external variable *environ*
  4. Shows the enviroment variables reached through *environ*
  5. Shows both the value and the storing address for main's third argument and the external variable *environ*

Environment variables: example 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");
```

Environment variables: example 1

```
%./entorno.out uno dos tres
0xbfbfffba0->argv[0]=(0xbfbffc8c) ./entorno.out
0xbfbfffba4->argv[1]=(0xbfbffc9a) uno
0xbfbfffba8->argv[2]=(0xbfbffc9e) dos
0xbfbfffbac->argv[3]=(0xbfbffca2) tres
0xbfbfffbb0->argv[4]=(0x0) ---------
0xbfbfffb5c->argv=0xbfbfffba0
0xbfbfffb58->argc=4
0xbfbfffbb4->env[0]=(0xbfbffca7) USER=visita
0xbfbfffbb8->env[1]=(0xbfbffcb4) LOGNAME=visita
0xbfbfffbbc->env[2]=(0xbfbffcc4) HOME=/home/visita
0xbfbfffbc0->env[3]=(0xbfbffcd7) MAIL=/var/mail/visita
0xbfbfffbc4->env[4]=(0xbfbffcee) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr
0xbfbfffbc8->env[5]=(0xbfbffd5c) TERM=xterm
0xbfbfffbcc->env[6]=(0xbfbffd67) BLOCKSIZE=K
0xbfbfffbd0->env[7]=(0xbfbffd73) FTP_PASSIVE_MODE=YES
0xbfbfffbd4->env[8]=(0xbfbffd88) SHELL=/bin/csh
0xbfbfffbd8->env[9]=(0xbfbffd97) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbdc->env[10]=(0xbfbffdb8) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbe0->env[11]=(0xbfbffdec) SSH_TTY=/dev/ttyp0
0xbfbfffbe4->env[12]=(0xbfbffdff) HOSTTYPE=FreeBSD
0xbfbfffbe8->env[13]=(0xbfbffe10) VENDOR=intel
0xbfbfffbec->env[14]=(0xbfbffe1d) OSTYPE=FreeBSD
0xbfbfffbf0->env[15]=(0xbfbffe2c) MACHTYPE=i386
0xbfbfffbf4->env[16]=(0xbfbffe3a) SHLVL=1
0xbfbfffbf8->env[17]=(0xbfbffe42) PWD=/home/visita/c
0xbfbfffbfc->env[18]=(0xbfbffe56) GROUP=users
0xbfbfffc00->env[19]=(0xbfbffe62) HOST=gallaecia.dc.fi.udc.es
0xbfbfffc04->env[20]=(0xbfbffe7e) REMOTEHOST=portatil
0xbfbfffc08->env[21]=(0xbfbffe92) EDITOR=vi
0xbfbfffc0c->env[22]=(0xbfbffe9c) PAGER=more
```

Environment variables: example 1

```
0xbfbfffbb4->environ[0]=(0xbfbfffca7) USER=visita
0xbfbfffbb8->environ[1]=(0xbfbfffcb4) LOGNAME=visita
0xbfbfffbbc->environ[2]=(0xbfbfffcc4) HOME=/home/visita
0xbfbfffbc0->environ[3]=(0xbfbfffcd7) MAIL=/var/mail/visita
0xbfbfffbc4->environ[4]=(0xbfbfffcee) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:
0xbfbfffbc8->environ[5]=(0xbfbfffd5c) TERM=xterm
0xbfbfffbcc->environ[6]=(0xbfbfffd67) BLOCKSIZE=K
0xbfbfffbd0->environ[7]=(0xbfbfffd73) FTP_PASSIVE_MODE=YES
0xbfbfffbd4->environ[8]=(0xbfbfffd88) SHELL=/bin/csh
0xbfbfffbd8->environ[9]=(0xbfbfffd97) SSH_CLIENT=192.168.0.99 33208 22
0xbfbfffbdc->environ[10]=(0xbfbfffdb8) SSH_CONNECTION=192.168.0.99 33208 193.144.51.154 22
0xbfbfffbe0->environ[11]=(0xbfbfffdec) SSH_TTY=/dev/ttyp0
0xbfbfffbe4->environ[12]=(0xbfbfffdff) 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
%
```

Environment variables: example 2

- Example 2
  1. Shows the command line arguments
  2. Shows the enviroment variables reached through *main* third argument
  3. Shows the enviroment variables reached through *environ*
  4. Shows both the value and the storing address for main's third argument and the external variable *environ*
  5. Creates a new variable using the library function *putenv*: `putenv("NUEVAVARIABLE=XXXXXXXXXXX")`
  6. Repeats steps 2, 3 and 4

Environment variables: example 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=XXXXXXXXXXX");

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

Environment variables: example 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
```

Environment variables: example 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=XXXXXXXXXX
0x80498d8->environ=0x804c000
0xbfbffb70->env=0xbfbffbc0
%
```

Environment variables

- Example 3
  1. Shows the command line arguments
  2. Shows the enviroment variables reached through *main* third argument
  3. Shows the enviroment variables reached through *environ*
  4. Shows both the value and the storing address for main's third argument and the external variable *environ*
  5. Creates a new variable using the library function *putenv*: `putenv("NUEVAVARIABLE=XXXXXXXXXXX")`
  6. Repeats steps 2, 3 and 4
  7. Makes an *exec* system call on the program in example 1

Environment variables: example 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);
```

Environment variables: example 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
```

Environment variables: example 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
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
```

Environment variables: example 3

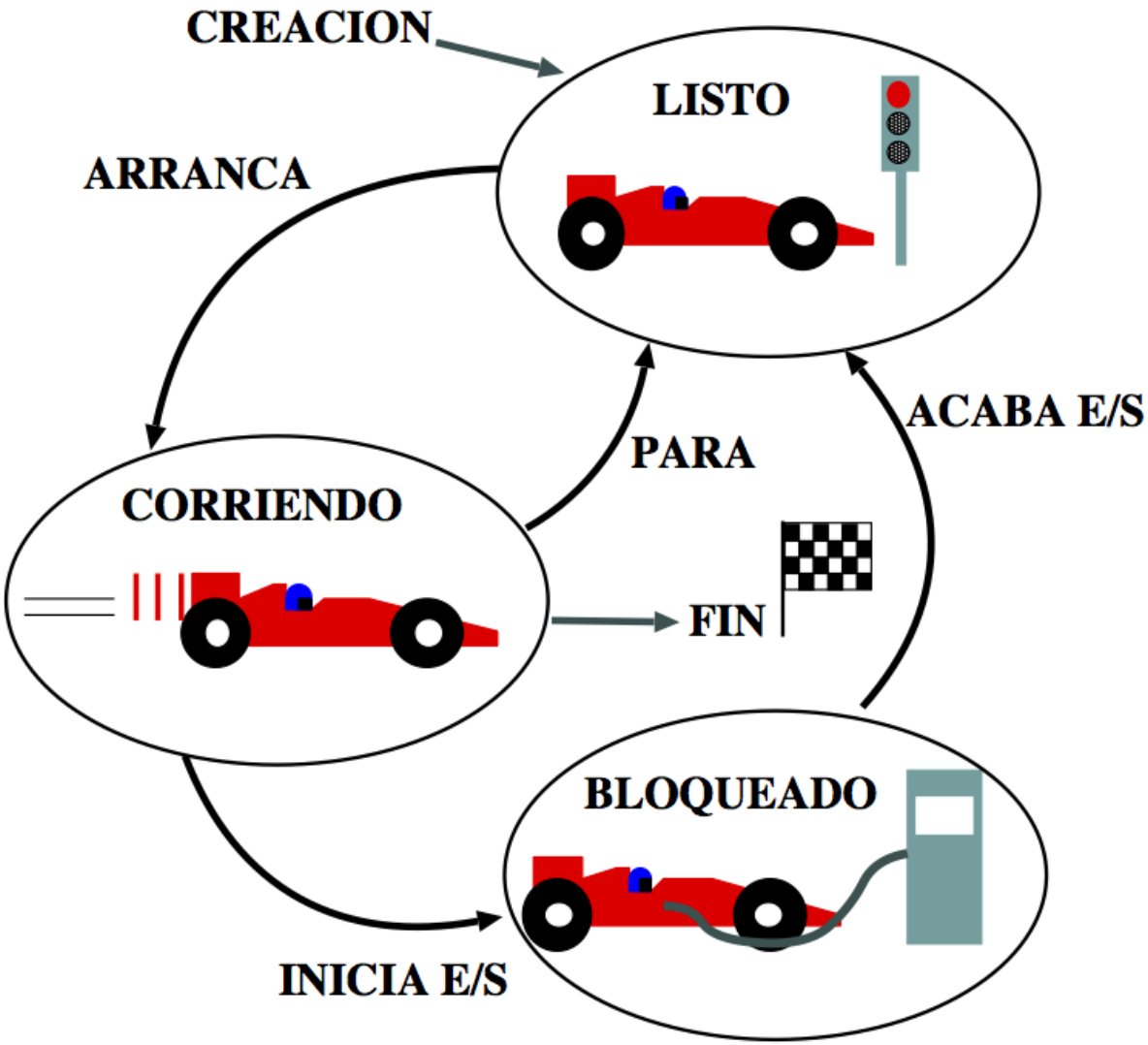
```
0xbfbfffba4->environ[0]=(0xbfbfffc8c) USER=visita
0xbfbfffba8->environ[1]=(0xbfbfffc99) LOGNAME=visita
0xbfbfffbac->environ[2]=(0xbfbfffca9) HOME=/home/visita
0xbfbfffbb0->environ[3]=(0xbfbfffcbc) MAIL=/var/mail/visita
0xbfbfffbb4->environ[4]=(0xbfbfffcd3) PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:
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
```

Process life cycle

Process life cycle

- Every process in the system starts its life with a create process system call. (create process is an O.S. service). The process that makes that system call is called parent process of the created process
  - In some O.S., (Windows) we must provide the system call to create process with the executable file we want the created process to run. In unix-like system we get different system calls to *create process* and to *execute program*
- During its life cicle a process goes throught different states: running, ready to run, blocked ...
- Every process in the system ends with the terminate process system call. (terminate process is an O.S. service)

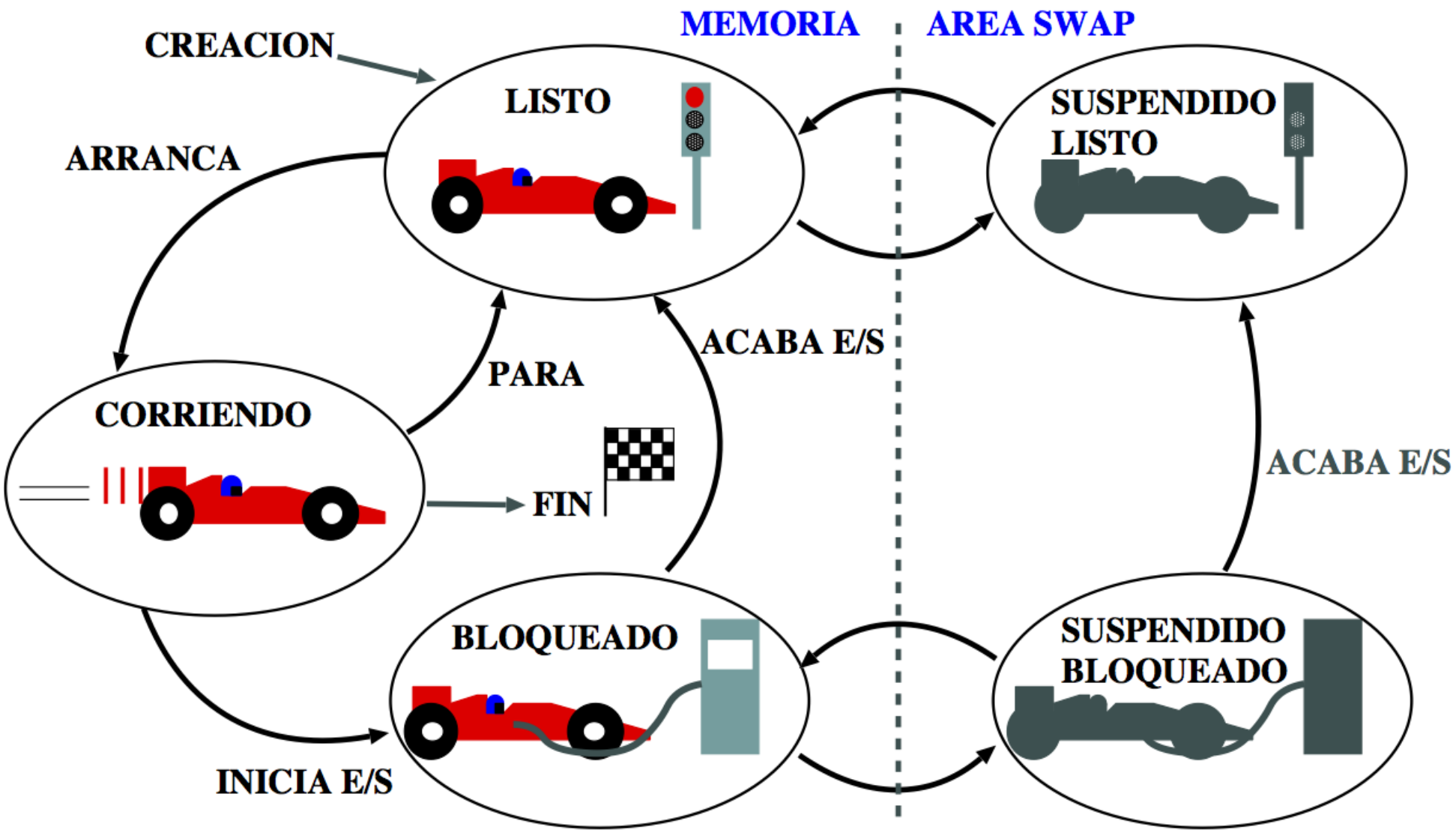
Process states



Process life cycle

- On older systems, some part of the secondary memory (disk) was used to swap out processes from the primary memory so that the degree of multiprogramming could be increased
  - whole processes were swapped out
  - This part of secondary memory si called swap
  - Now we have a new process state: swapped out or swapped. Sometimes refered to as suspended
  - The transitions associated with this new state are swap in and swap out
  - These systems are refered to as swapping systems

Process states



Process life cycle

- On modern systems, some part of the secondary memory (disk) is used to swap out PIECES of processes from the primary memory so that the degree of multiprogramming can be increased. This also allows for a process that is not loaded completely into memory to be executed
  - PIECES of processes (typically pages) are swapped out
  - This part of secondary memory si called swap. It can be ether a partition or a file on a filesystem
  - Processes can be executed without being completely loaded into memory. This also allows for executing processes larger than the physical memory installed in the system
  - This is what we call virtual memory
  - This systems are often refered to as paging systems

Process states

Terminology:

- CPU, running or executing
- ready to run or runnable (*ready*)
- blocked, asleep or waiting.
- swapped out, suspended (a process in this state can either be runnable or blocked)

State transitions

- Entering the running state: the first in the ready to run queue is scheduled to run
- Entering the ready to run state: there are 4 different scenarios
  - A new process has been created and it enters the runnable queue
  - From CPU: Another process is scheduled to run via a context switch. We say the process has been preempted.
  - From blocked: The event the process was waiting for (some i/o operation or whatever) has ocurred. We call this transition unblock or wake up. The transition is the same if the process is also swapped out
  - From ready to run/swapped out: The O.S. decides to bring it to primary memory. This transition is called swap in.

State transitions

- Entering the blocked state: two different scenaios
  - From CPU: The process makes some system call (for example asks for some i/o to be done) that cannot be complete at the time so it blocks
  - From blocked/swapped out: The O.S. swaps in a blocked/swapped process. (Not every O.S. accepts this transition)
- The blocked/swapped out and ready to run/swapped out states can be entered when the O.S. decides to swap out a process (ready or blocked) to free some primary memory

Process creation

- create process is an O.S: service
- When a process makes a create process system call, the O.S. must:
  1. Assign an Identifier to the new process
  2. Create and initialize its PCB
  3. Update the SCB to include the new process
  4. Assign memory to it and, if needed, load the program the new process is going to execute
  5. Put it into the ready to run queue.

Process creation

- It may seem that a process can be created in different ways
  - System initialization: a lot of processes are created when a system boots:
    - Processes that interact directly with the user (graphical environment, shells).
    - Processes that DO NOT interact directly with the user. They are in charge of maintaining some system services (mail reception, printer management . . .). They are usually called *daemons*,
  - system call inside some process to create anew process. Example: we make a program that creates new process to fill a buffer with data.
  - explicit user request. Example: the user explicitly creates a new process from the *shell* or by clicking (once or twice) in the appropiate graphic menu. (in this case it is the *shell* code or the code from the graphical enviroment that creates the new process)
  - As part of a batch processing
- In all these scenarios, the proceses are actually created the same way: by a system call

Termination of a process

- Terminate process: is an O.S. service.
- When a process is terminated its PCB is deleted. The O.S. reclaims all the resources assigned to that process.
- If the process has some children processes: it may wait for them to end, terminate them or leave them be
- Two ways of termination
  1) normal termination: The process calls voluntarily the terminate process system call
  2) abnormal termination: Not provided for in the process code. The process is *forced* to make the terminate process system call

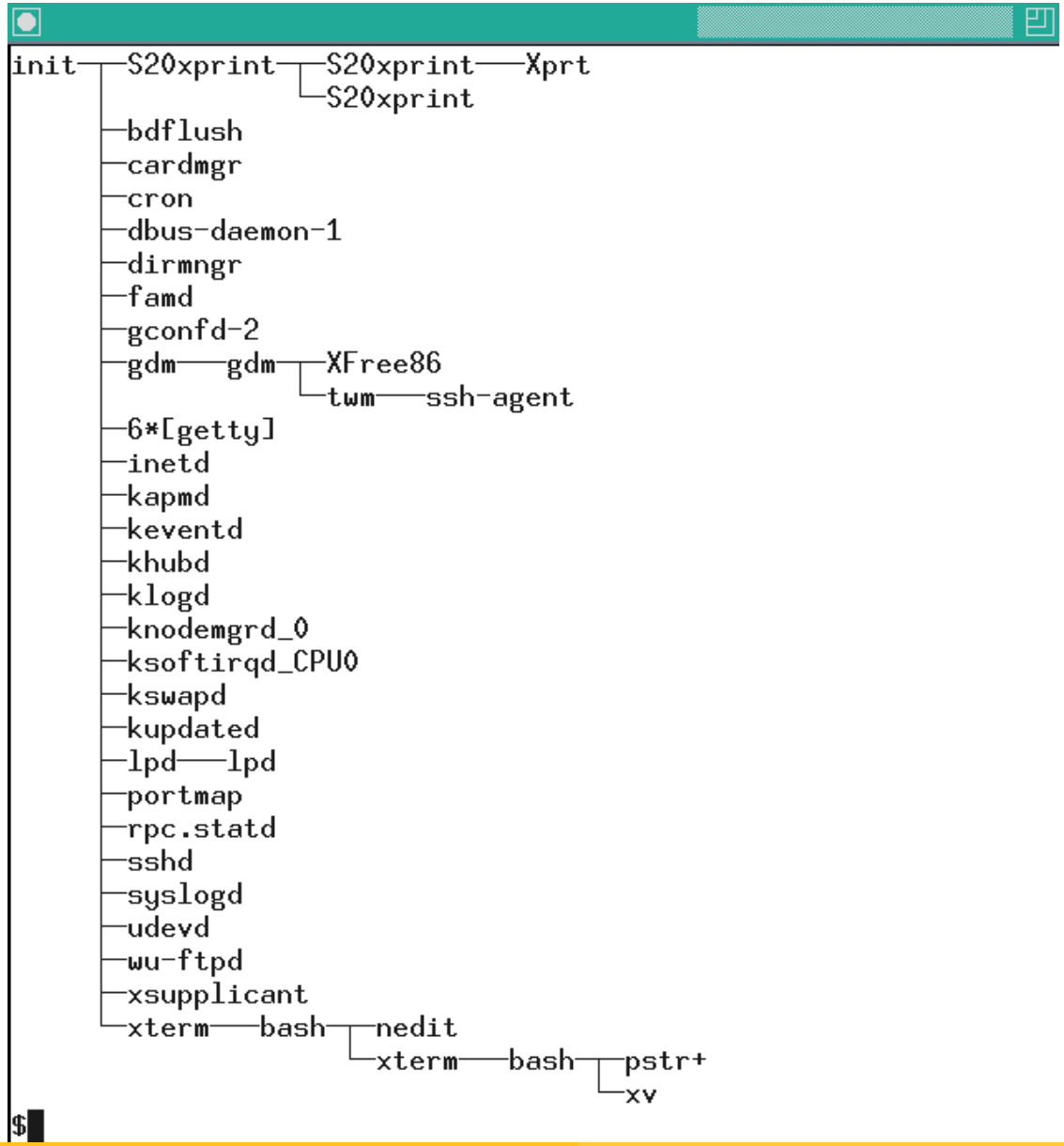
Process life cycle in UNIX

Process life cycle in UNIX

- process: *instance of a program executing*
- process: *entity that the O.S. creates to execute a program and that provides an environment for the program to execute: address space and one (or several) control point*
- A process has a specific life span
  - It is created by the *fork()* (or *vfork()*) system call
  - Ends with the *exit()* system call
  - Can execute a program with one of the *exec()* system calls

- Every process has a parent process
- Can have one (or more than one) child processes
- Tree like structure with *init* the common ancestor to (almost) all processes in the system
- When a process ends its children processes are inherited by *init*

Process tree shown by the command *pstree*

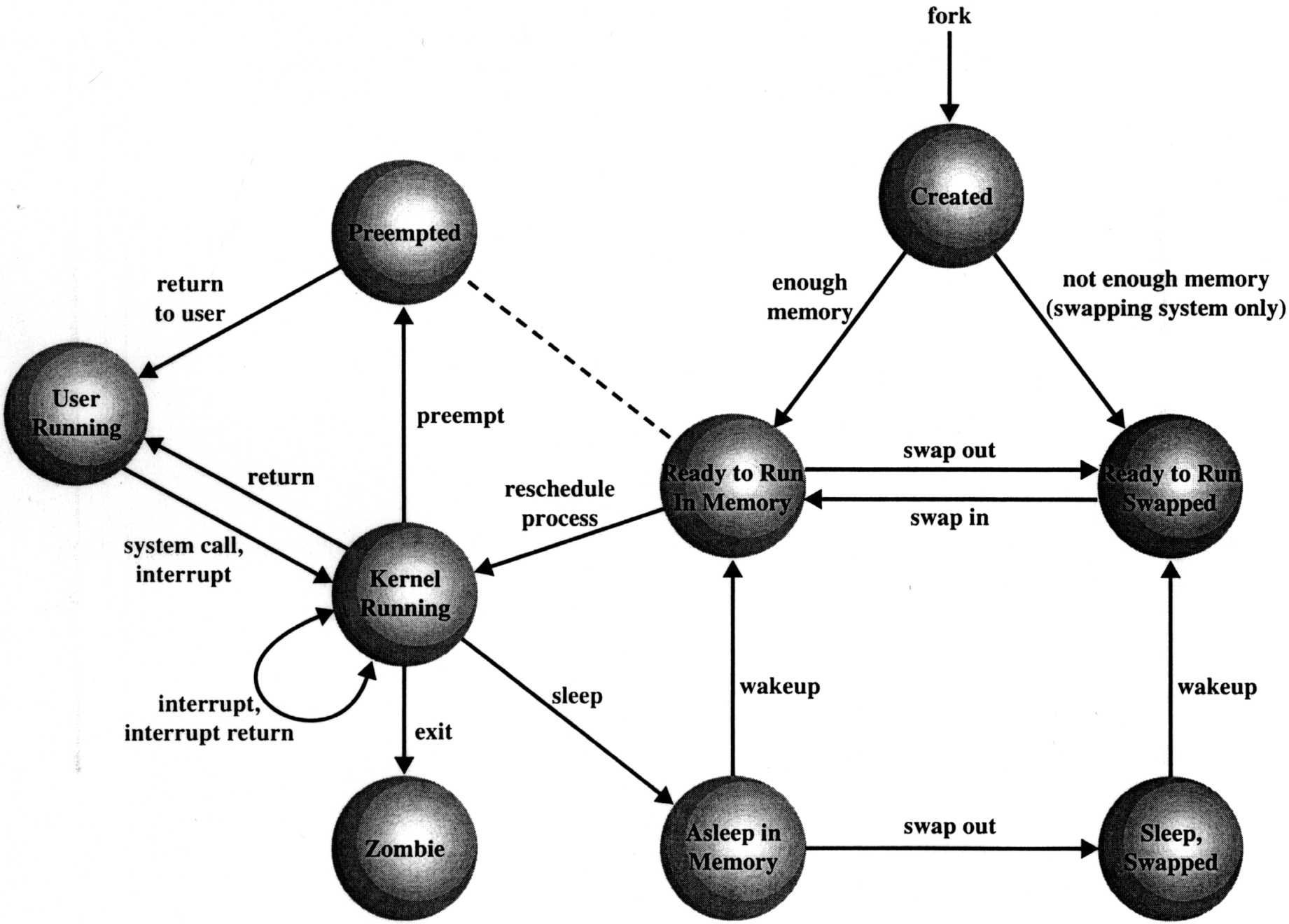


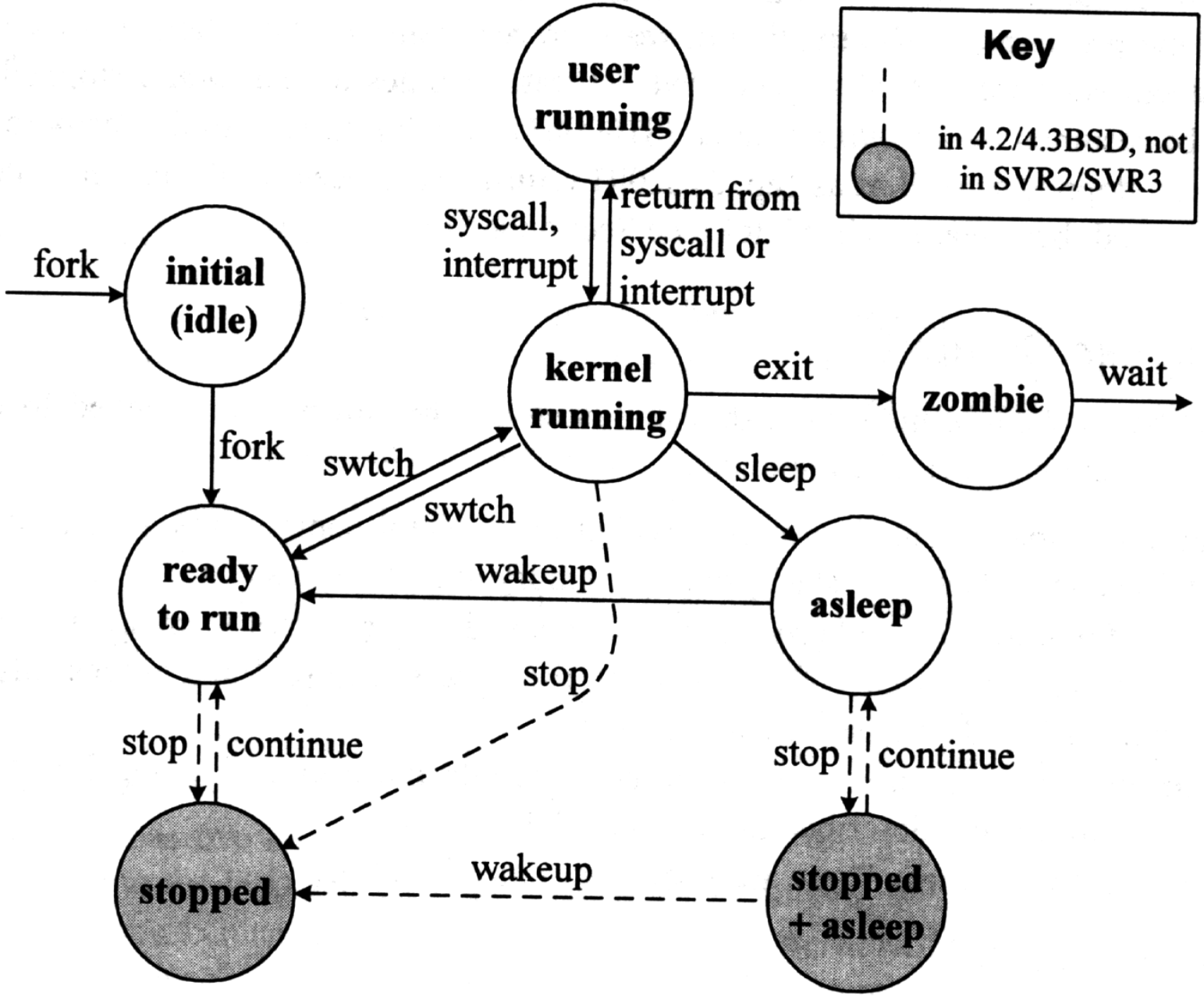
Process states in System V R2

- **idle**: The process is being created but it is not yet ready to run
- **runnable, ready to run**
- **blocked, asleep**. In this state, as with the *runnable* state, the process can be either in main memory or in the swap area (*swapped*)
- ***user running***
- **kernel running**
- **zombie**: The process has terminated but the parent process has not yet performed one of the *wait* system calls on it: its *proc structure* has not been emptied so, for the system, the process still exists.

- From 4.2BSD on there's a new state: **stopped**
- It can be either *runnable stoped* or *blocked stopped*
- It can be reached by receiving one of these signals
  - **SIGSTOP**
  - **SIGTSTP** ctrl-Z
  - **SIGTTIN**
  - **SIGTTOU**
- SIGCONT takes a process out of this state

- Execution starts in kernel mode
- Transition to *blocked* is from kernel mode running
- Transitions to and from runnable are from kernel mode running
- Execution ends in kernel mode
- When a process ends it goes into *zombie* state until its parent process performs one of the *wait* system calls on it


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


user running
Key
in 4.2/4.3BSD, not in SVR2/SVR3
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

fork() and exec() system calls

- unix system calls to create processes and execute programs are:

fork() Creates a process. The created process is a "klon" of the parent process, its address space is a replica of the parent process' address space. The only difference is the value returned by *fork()*: 0 to the child process and the child's pid to the parent process

exec() (*execl(), execv(), execle(), execve(), execlp(), execvp()*) Makes an already created process execute a program: it replaces the calling process address space (code, data, stack . . .) with that of the program to execute

exit() Ends a process

fork() and exec() calls: example

```
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*/
```

Tasks performed by *fork()*

1. Allocate swap space
2. Assign *pid* and allocate `proc` structure
3. Initialize `proc` structure
4. Assign address translation maps for child process
5. Allocate child process's `u_area` and copy data from parent process
6. Update fields in `u_area`
7. Add child process to set of processes sharing code region of parent process
8. Duplicate data and stack segments from parent process and update tables
9. Initialize hardware context
10. Change state of chiuld process to *ready to run*
11. Return 0 to child process
12. Return child's pid to parent process

Optimizing *fork()*

- Among the tasks preformed by *fork()*, *'duplicate data and stack segments from parent process and update tables'* implies
  - allocate memory for child's process data and stack
  - copy parent process's data and stack
- It often happens that a process just created by *fork()* executes another program

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

- The *exec()* calls discard current address space and allocate a new one
- In this case, we have allocated memory, copied data on it and then ended up discarding all that memory

Optimizing *fork()*

- Two optimizations: *copy on write* and *vfork()* system call
- ***copy on write***
  - Data and stack are not copied: they are shared between parent and child processes
  - Data and stack are marked read only
  - When an attempt is made to modify any them, as thay are marked read only, an exception is produced
  - The execption handler copies *ONLY THE PAGE* that is being modified. Only modified pages of data and stack are copied
- ***vfork()* system call**
  - used only if a call to exec *exec()* is to be made in a short time
  - Child process *borrows* parent process' space address until a call to *exec()* or *exit()* is made. At this moment the parent process is awaken and returned its address space
  - Nothing gets copied

Executing programs: *exec()*

- An already created process executes a program; its address space is replaced by that of the program to be executed
  - If the program was created by *vfork()*, *exec()* returns the address space to the parent process
  - If the program was created by *fork()*, *exec()* releases the address space
- A new address space is created and loaded with the new programs
- When *exec()* ends, execution starts and the new program's first instruction
- *exec()* **DOES NOT CREATE** a new process. The process is the same: same *pid*, same `proc` structure, same *u_area*, ...
- If the program to be executed has the adequate *mode* (*setuid* and/or *setgid* bits), *exec()* changes the efective and saved user and/or group credentials of the process calling exec to that of the executable file *uid* and/or *gid*

Executingprograms: *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[]);
```

Tasks performed by *exec()* I

1. Get executable file from path (*namei*)
2. Check for execute access
3. Inspect file header an check it is a valid executable
   - If its a text file starting with `!#`, call the interpreter and pass the file as argument
4. If the bits *setuid* and/or *setgid* are set (–s—— or ——s—) change the efecctive (and saved) *uid* or *gid* of the process
5. Save environment and arguments to *exec()* in kernel space (user space is being discarded)
6. Allocate new swap space (data and stack)
7. Release address space (if the process was created by *vfork()* return it to the parent process)

Tasks performed by *exec()* II

8. Allocate a new address space. If the code is already in use, share it, if not, load it from the executable file.
9. Copy environment variables and arguments to *exec()* into new user stack
10. Restore signal handlers to the default action
11. Initialize hardware context, all registers to 0, except Program Counter, to entry point of program

Terminating a process: *exit()*

- A process can end using the *exit()* system call
- *exit()* performs the following tasks
  - Deactivate all signals
  - close all process's open files
  - free from Vnode Table vnodes of the code file, control terminal, root directory and current working directory (*iput*)
  - Save resource usage statistics and *exit state* into `proc` structure
  - Change process state to SZOMB and place `proc` structure into *zombie* list
  - Make *init* inherit all of process's children processes
  - Deallocate address space, *u_area*, and swap space ...
  - Send SIGCHLD to parent process (usually ignored)
  - If parent is waiting for child process then awake parent process
  - Call *swtch* to initiate context switch
- IMPORTANT: as `proc` structure is not deallocated, the process still sxists, although in *zombie* state

Waiting for a child process to end: *wait()* system calls

- If a process needs to know how a child process has terminated, it can use one of *wait()* system calls

```
#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);
```

Waiting for a child process to end: *wait()* system calls

- *wait()* Checks whether a child process has ended
  - If it has, *wait()* returns inmediately
  - If it has not, the process calling *wait()* waits until any of its children processes has ended
- The exit value that the child process passed to *exit()* is *transfered* to the variable *wait* uses as parameter
- Deallocates child processes's *proc* structure
- Reurns *pid* of the child process that has ended
- *waitpid(), waitid(), wait3* y *wait4()* do admit options
  - **WNOHANG** Does not wait for the child process to end
  - **WUNTRACED** In addition to reporting ending processes, stopping of a child process is also reported
  - **WCONTINUED** In addition to reporting ending processes, continuing of a stopped child process is also reported (in linux, only from kernel 2.6.10 on)
  - **WNOWAIT** Does not deallocate child process's`proc` structure (solaris)

Waiting for a child process to end: *wait()* system calls

- `proc` structure is not deallocated until one of the *wait()* system calls is used
- Creation of *zombie* processes can be avoided using flag SA_NOCLDWAIT in *sigaction* for the SIGCHLD signal.
- Should that be the case *wait()* would return -1 setting errno to ECHILD

Waiting for a child process to end: *wait()* system calls

- Value obtained with `stat_loc` in *wait (int * stat_loc)* is interpreted as follows
  - If child process terminated normally calling *exit()*
    - 8 least significative bits are 0
    - 8 more significative bits are the 8 less significative bits of argument passed to *exit()*
  - If child process was terminated by a signal
    - 8 more significative bits are 0
    - 8 least significative bits contain the signal number
  - If the process was stopped
    - 8 least significative bits contain WSTOPFLG
    - 8 more significative bits contain the number of the signal that stopped the processs

Waiting for a child process to end: *wait()* system calls

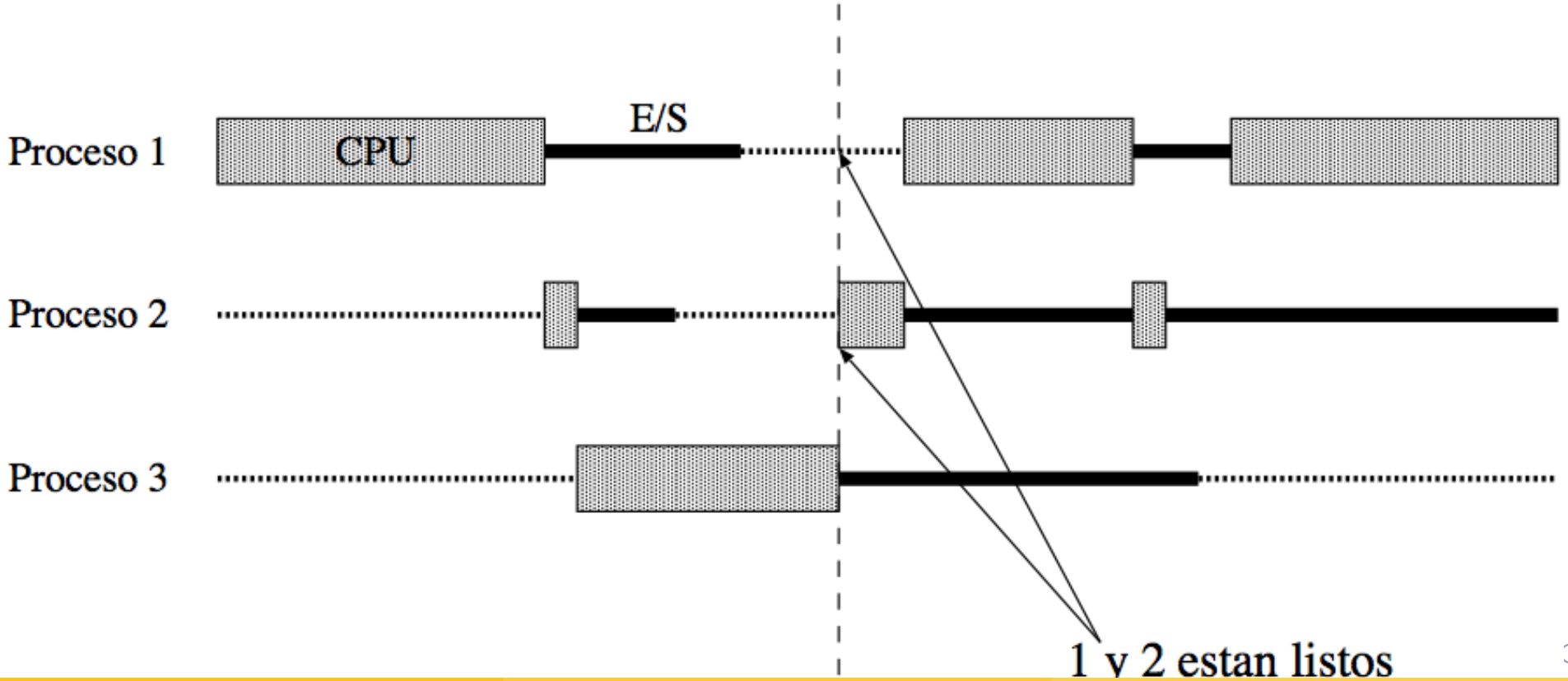
To determine the meaning of the value obtained with *wait()*, there exist the following macros:

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

WIFEXITED(status)
WEXITSTATUS(status)

WIFSIGNALED(status)
WTERMSIG(status)

WIFSTOPPED(status)
WSTOPSIG(status)

WIFCONTINUED(status)
```

CPU Scheduling

CPU Scheduling

- In a multiprogrammed O.S. several processes and/or threads compete for CPU.
- Each process is formed by a sucession of CPU ad I/O bursts.
- Each process starts and ends with a CPU burst



CPU Scheduling

- The (scheduler) es the part of the O.S. which decides which process (among the runnable processes) obtains the CPU.
- There are two kinds of scheduling algorithms.
  - non-preemptive algorithms: The currently running process stays in the CPU until it ends its CPU burst (*voluntarily* relinquishing the CPU: start of an I/O operation, wait for a child to terminate, terminating . . .)
  - preemptive algorithms: The scheduler can move out from the CPU the currently running process before it ends its CPU burst (*preemption*)

Types of scheduler

- short term scheduler: Decides which process enters the CPU among the runnable processes
- medium term scheduler: In *swapping* systems: decides which swapped out processes will be swapped in
- long term scheduler: in *batch systems*. Decides which process(es) in the *spool* device will be loaded into main memory. It controls the degree of multiprogramming

Scheduler goals

- the goals of a scheduler will vary depending on the environment it is used:
  - Batch environments: Typically non-preemptive scheduling. long term scheduler sometimes refered to as *job scheduler*. It decides the order in which the jobs are processed and the degree of multiprogramming. It's main goal is to be efficient and have great throughput
  - Interactive environments such as graphical systems, servers. Preemptive scheduling, usually time sharing is used. Its main goal is to give at least some CPU to all processes in a timely manner
  - Real time environments: Some processes in the system have very specific time constraints that need to be met. Typically priority based scheduling is used and those processes with special needs are assigned the greatest priorities in the system

Typical scheduler goals

- In every system the scheduler has to provide
  - fairness: every process has to get a fair (or reasonable) share of the CPU.
  - Policy: meet a certain criteria previously stablished.
  - Balance: Different parts of the system share similar workloads.
- Batch Systems:
  - Throughput: Number of jobs per time unit. We try to maximize it
  - Turnaround time: Time elapsed since the submitting of a job and its ending. We try to minimize it.
  - CPU usage rate: We try to keep the CPU busy all the time.

Typical scheduler goals

- Time Sharing:
  - Response time: Time elapsed since the user submits something for execution until it produces some response.
  - Proportionality: Keep user expectations (simple tasks=small response time).
  - Maximize the number of active interactive clients.
- Real Time:
  - Reliability: avoid data loss; react before time limit, etc.
  - Predictability

Scheduling Evaluation

- Try to determine which scheduling algorithm works best on a given system
- Goals vary depending on the type of system.
- There are three methods to evaluate how an algorithm behaves on a given system
  - Analytical methods (both deterministic and non deterministic)
  - Simulation
  - Implantation

Time measurement

- Turnaround time (t_R) = time elapsed since the process is initiated (Ti) until it ends (Tf).

$$t_R \stackrel{def}{=} Tf - Ti$$

  It includes:
  - Time to be loaded into main memory
  - Time in the ready to run queue
  - Time executing in CPU t_{CPU}
  - Time blocked on I/O $t_{I/O}$
- Waiting time (t_W) is the time obtained substracting from the turnaround time the time in CPU and the time in i/o, $t_E \stackrel{def}{=} t_R - t_{CPU} - t_{I/O}$.

Time measurement

- Service time (t_S) = Is the time the process would need if it were the only process in the system and it didn't need to be loaded into main memory. That's to say the turnaround time minus the waiting time.

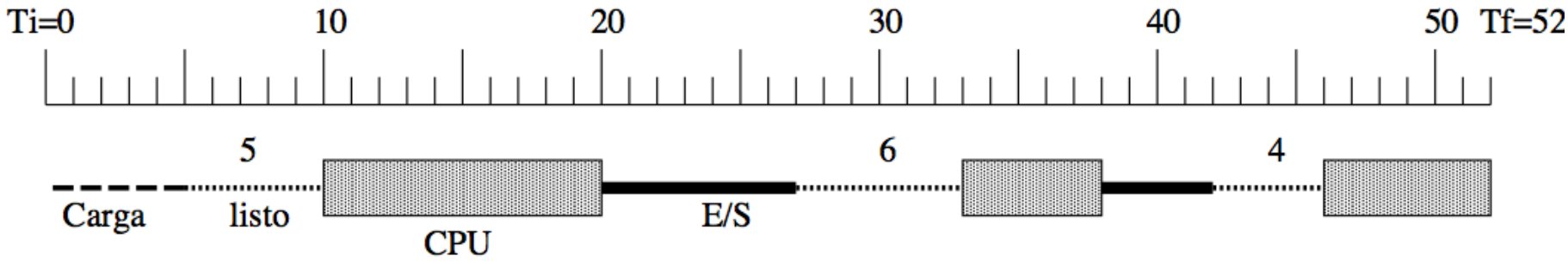
$$t_S \stackrel{def}{=} t_R - t_E = t_{CPU} + t_{I/O}$$

- Service index (i)

$$i_S \stackrel{def}{=} t_S / t_R$$

Time measurement

An example ...



- Turnaround time: $t_R = Tf - Ti = 52 - 0 = 52$
- CPU time: $t_{CPU} = 10 + 5 + 6 = 21$
- I/O time: $t_{I/O} = 7 + 4 = 11$
- Service time: $t_S = t_{CPU} + t_{I/O} = 32$
- Waiting time: $t_E = t_R - t_S = 52 - 32 = 20$
- Service index: $i_S = 32/52 = 0.615$

Scheduling Evaluation

Deterministic Models

- We take a sample workload and evaluate how the system behaves. Important: the workload must be representative.
- We use some of the time measurements to assess th algorithm's peformance (for example. mean turnaround time, throughput, etc).
- Pros: Simple. It yields exact measurements.
- Cons: Misleading results if the workload is not correctly selected.

Scheduling Evaluation

Nondeterministic models (queuing theory)

- On many systems, the arrival time and length of the jobs connot be predicted, so it is not possible to use a deterministic model.
- We use probability distribution functions to model the CPU bursts and arrival times for the jobs in the system.
- With those two distributions we can estimate the mean values of throughput, turaround time, waitting time . . . .

Scheduling Evaluation

Nondeterministic Models (queuing theory)

- The computer system is described as a series of “servers”. Each server has a queue of waiting jobs. The CPU is a server for its ready to run processes list. The same stands for each i/o device.
- If we know the rate at which new processes arrive and how long they are, we can calculate each *server* usage, mean value for the length of each of the servers queues . . . .

Scheduling Evaluation

Simulation

- Another option is to simulate the system behaviour.
- Data for processes are either ramdomly generated or sampled from a real system.
- This method gives a real glimpse on how an scheduling algorithm actually performs.
- High computing cost (getting the data, simulation times, meassurements, etc).

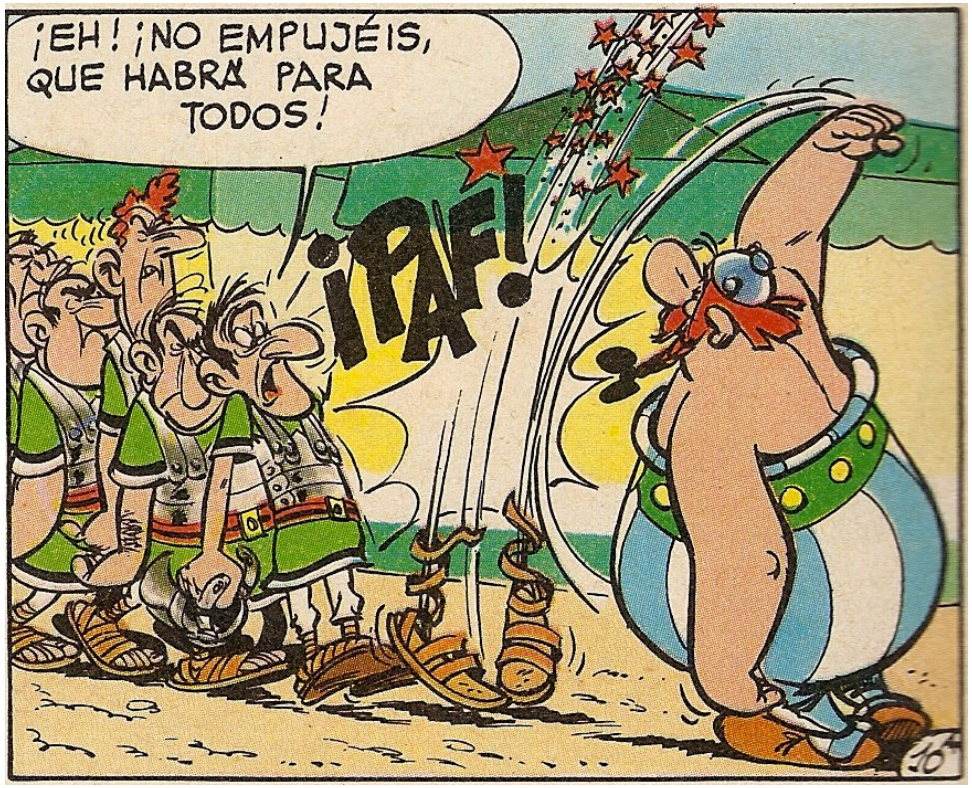
Scheduling Evaluation

Implantation

- The algorithm is implemented on a running system to be evaluated
- Data obtained correspond to actual processes in a real system.
- The mere implatation of some specific algorithm in a running system can condition user behaviour so that the results thus obtained may be not as *authentic* as they should

Non preemptive: FCFS

First-Come-First-Served (FCFS):



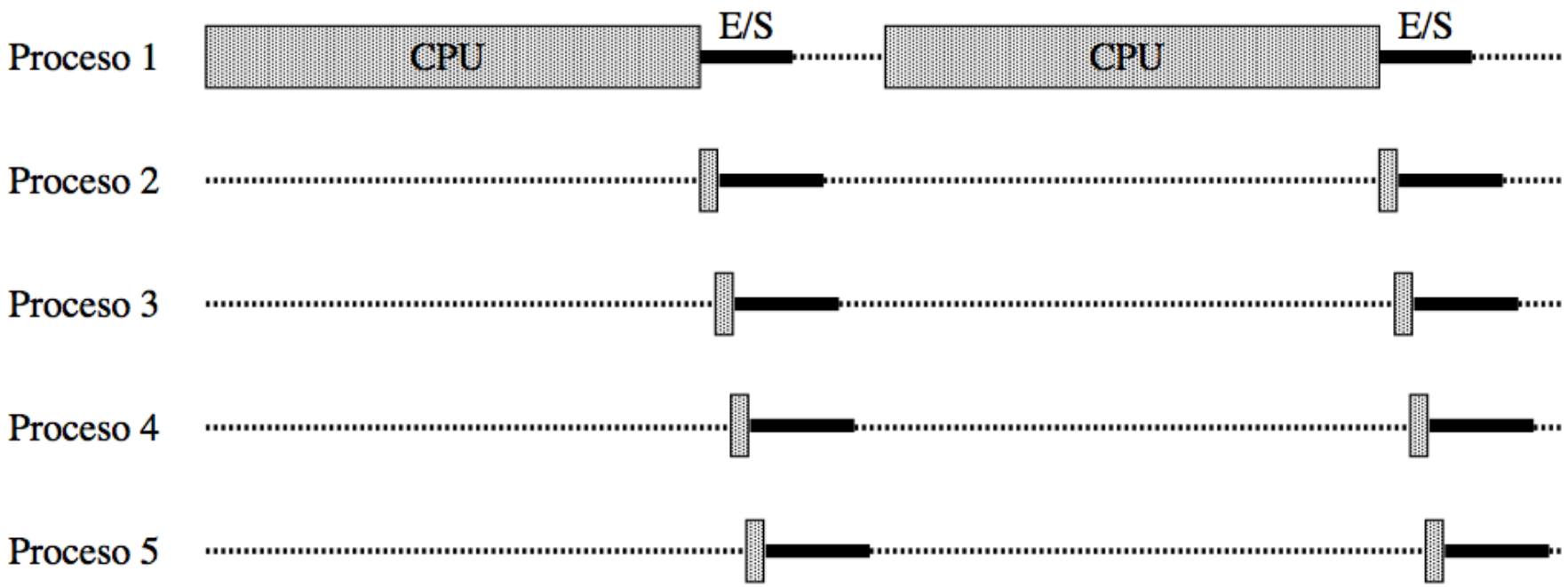
Advantages:

- Easy to implement. A FIFO queue is enough
- fair

Non preemptive: FCFS

Algorithm First-Come-First-Served (FCFS)

- Drawback: risk of low throughput; "convoy" effect
- Example: one CPU bound process and many i/o bound processes.



Non preemptive: SJF

Algorithm Shortest Job First (SJF).

- Only theoretical usage, it needs to know beforehand the length of the CPU burts.
- Produces the shortests turnaround times with various processes arriving simultaneously.
- When two CPU bursts are the same length FCFS is used.

Non preemptive: SJF

Algorithm Shortest Job First (SJF): Example.

- We have 4 processes with respective CPU bursts 8, 4, 4 y 4 ms. Using FCFS:

8	4	4	4

- Turnaroud times are P1: 8, P2: $8 + 4 = 12$, P3: $12 + 4 = 16$, P4: $16 + 4 = 20$.
- Mean value of turnaround time: $(8 + 12 + 16 + 20)/4 = 14$

Using SJF

4	4	4	8

- Mean value of turnaround time: $(4 + 8 + 12 + 20)/4 = 11$
- It can be proved that this algotithm produces the best possible results. Example: with times a, b, c, d, the mean value for the return time is $(4 \cdot a + 3 \cdot b + 2 \cdot c + d)/4$. Clearly it improves if a is the shortest, b is the next shortest ....

Non preemptive: SJF

Shortest Job First (SJF)

- When processes appear at different times its doesn't neccessarily produce the, best results. Check the following example

Proccess	CPU burst	Arrival time
A	2	0
B	4	0
C	1	3
D	1	3
E	1	3

- Compute mean turnaround time for SJF with processes arriving in the order: B,C,D,E,A.

Non preemptive: SJF

Shortest Process Next

- SJF is usually implemented estimating the next CPU burt from the previous ones

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$$

where:

$$\begin{aligned} \tau_{n+1} &= \text{estimated value} \\ t_n &= \text{last burst} \\ \tau_n &= \text{previous estimated value} \\ \alpha &\in [0, 1] \text{ adjustement factor} \end{aligned}$$

Non preemptive: Priorities

Prioirity scheduling

- SJF is just an example of prioirity scheduling.
- Priority scheduling can be both preemptive and non preemptive. In non preemptive priority scheduling, when the process in CPU voluntarily relinquishes CPU, the scheduler selects the highest priority process among all ready to run processes

Definition (Priority)

Priority is a numeric value used to decide whether a process gets to use CPU before other processes.

- Depending on the system, higher numbers may represent higher or lower priorities
- The range of the numeric values also depends on the system

Non preemptive; Priorities

Depending on how they are assigned, priorities can be

- Internal: Assigned by the O.S. from information on the processes: CPU and memory usage, open files, i/o bursts ....
- External: Assigned to de processes by the users or de System Administrator.
- Mixed: Combination of internal and external

Priorities can also be considered

- static: The priority of a process does not change (unless the system administrator or some user explicitly changes it)
- dynamic: The system recalculates (periodically or not) the processes' priorities

Non preemptive: Priorities

- Main drawback: starvation: A process with low priority waits forever.
- Usually solved with dynamic priorities. Two examples:
  - Use as priority q/t_{CPU} where t_{CPU} was the last CPU burst.
  - Aging: Priority of a process increases as time goes by without the process getting to use the CPU.

Preemptive: priorities

Preemptive priority scheduling

- Similar to the non preemptive priorities algorithm
- When a process with higher priority than the one in CPU becomes ready, it takes the CPU from the one using it, which goes into the *preempted state* (*ready to run*).
- Previous classification (internal, external, static . . .) also applies.
- Performance of this scheduling algorithm depends, as on the previous case, on how priorities are assigned

Preemptive: SRTF

Shortest Remaining Time First (SRTF)

- It is the preemptive implementation of SJF.
- Every time new jobs appear ready, their CPU bursts are compared with the remaining time of the one in CPU.
- If one of the new jobs has a CPU burts shorter than the remaining time of the one in CPU, the new job gets the CPU.
- Again, we assume we know the CPU bursts beforehand. A real implementation must estimate them.

Preemptive: RR

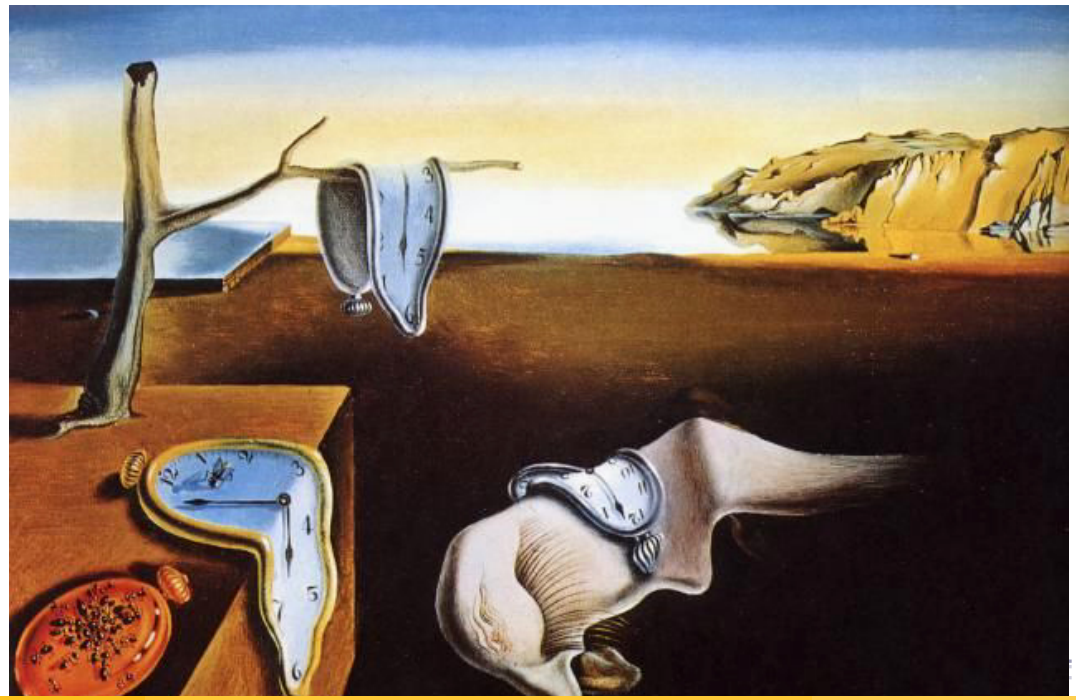
Round-Robin (RR)

- each process has a time limit in its CPU time called quantum (q).
- Ready to run processes are organiced in a FIFO queue.

Preemptive: RR

Round-Robin (RR)

- If *A* is executing and reaches the quantum $\Rightarrow$ a context switch occurs.
- The first process in the ready to run queue gets the CPU and *A* entes the queue (last).
- A timer (clock interrupt) takes care of waking up the scheduler.



Preemptive: RR

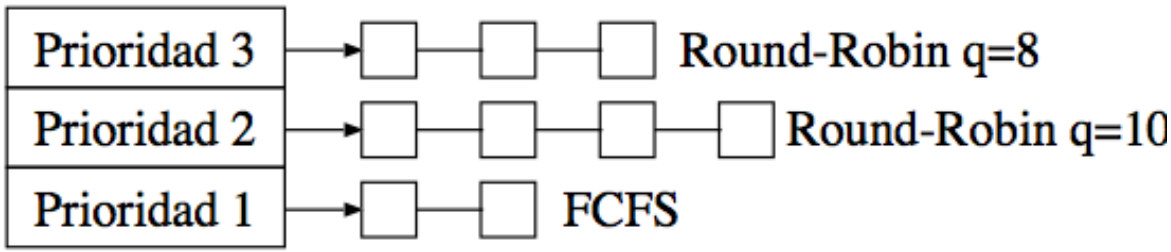
Algorithm Round-Robin (RR)

- Advantages: easy to implement. Fairness: every process gets its slice of CPU time.
- Drawback: finding the right q value
  - Smaller q values cause many context switches: loss of time.
  - Too larger a q value, and the algorithm degenerates into FCFS.
- It has been found that when $\approx 80\%$ of the CPU bursts are lower than q, the algorithm yields the best results . Tipycal value $20\ ms \leq q \leq 50\ ms$.

Preemptive: Multilevel Queue

Multilevel Queue

- It's an evolution from the priority scheduling.
- We have one queue for each priority level. Each queue can have its own scheduling algorithm.
- Moreover, to avoid starvation, dynamic priorities are used, allowing for a change of queue.



Preemtive: Multilevel Queues

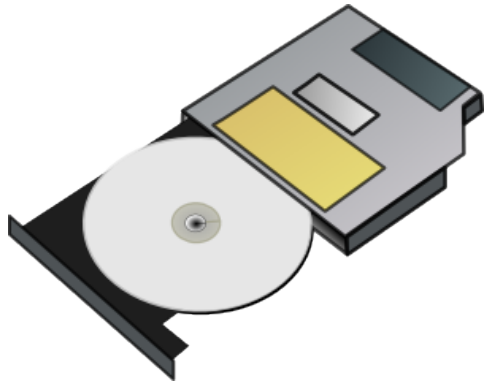
Multilevel Queues

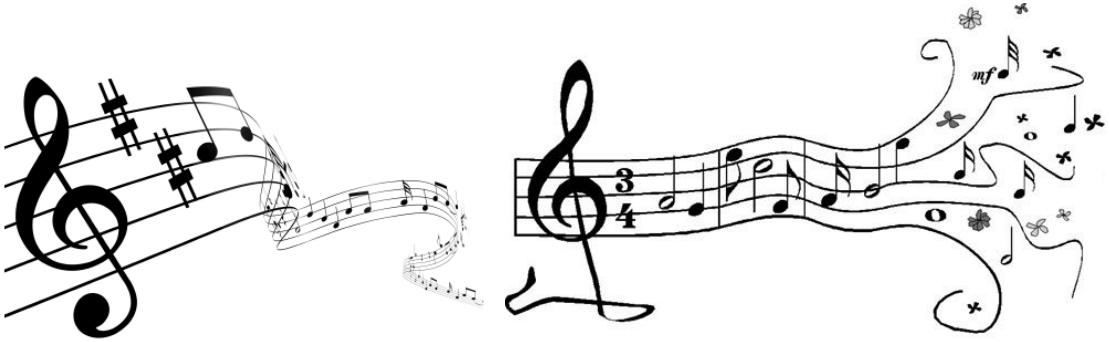
Example

- Higher priorities for system processes, interactive foreground processes, or I/O bound processes. RR.
- Lower priorities for non interactive background processes. FCFS.
- Another example: two queues and we split the time between queues (ex. 80% for RR y 20% for the FCFS queue).

Real time scheduling

- On real time systems, time is of critical importance.
- One or more physical devices generates stimuli and the system must react to them within limited time.
- Example (old): a CD player reads data from the media which must converted into music within limited time.

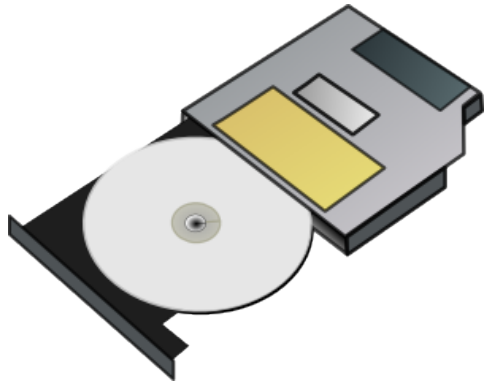


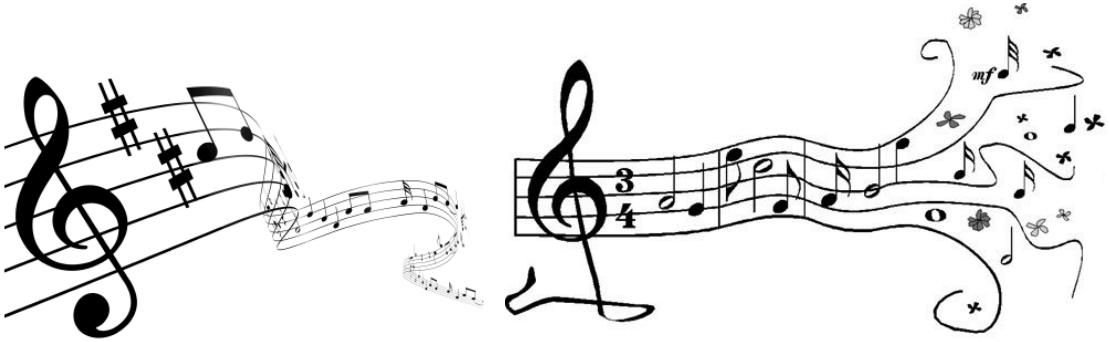


- It it is not done properly: quality loss or wierd sounds.

Real time scheduling

- On real time systems, time is of critical importance.
- One or more physical devices generates stimuli and the system must react to them within limited time.
- Example (old): a CD player reads data from the media which must converted into music within limited time.





- It it is not done properly: quality loss or wierd sounds.

Real time scheduling

- Hard real time: All time limits MUST be met
- Soft real time: missing one hit, thougt not desirable, is tolerable.
- A program is usually divided into short and predictable bursts whose duration is known in advance.
- The scheduler must organize the processes so the time limits are met.
- On a real time system we distinguish between these two kinds of events
  - Periodical: occur at regular intervals
  - Non periodical: happen unpredictably.

Real time scheduling

- A real time system may have to respond to several periodical event streams. If each event requieres too much processing it can becomme unmanageable.
- Let's think of $1, \ldots, m$ periodical event streams, and for each stream i:

$$\begin{aligned} P_i &= \text{period at which the event occurs} \\ C_i &= \text{CPU time needed to process the event} \end{aligned}$$

Definition (Schedulable real time system)

We define a real time system with m streams to be schedulable if it satisfies:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Real time scheduling

- Example: 3 streams with periods $P_1 = 100$, $P_2 = 200$ y $P_3 = 500$ and CPU consumptions of $C_1 = 50$, $C_2 = 30$ y $C_3 = 100$ (everything is in *ms*).
- the sum is $0,5 + 0,15 + 0,2 < 1$. which makes the system schedulable
- If we were to have another stream with period $P_4 = 1000$. What is the maximum value of its CPU consumption C_4 to keep the system schedulable ?

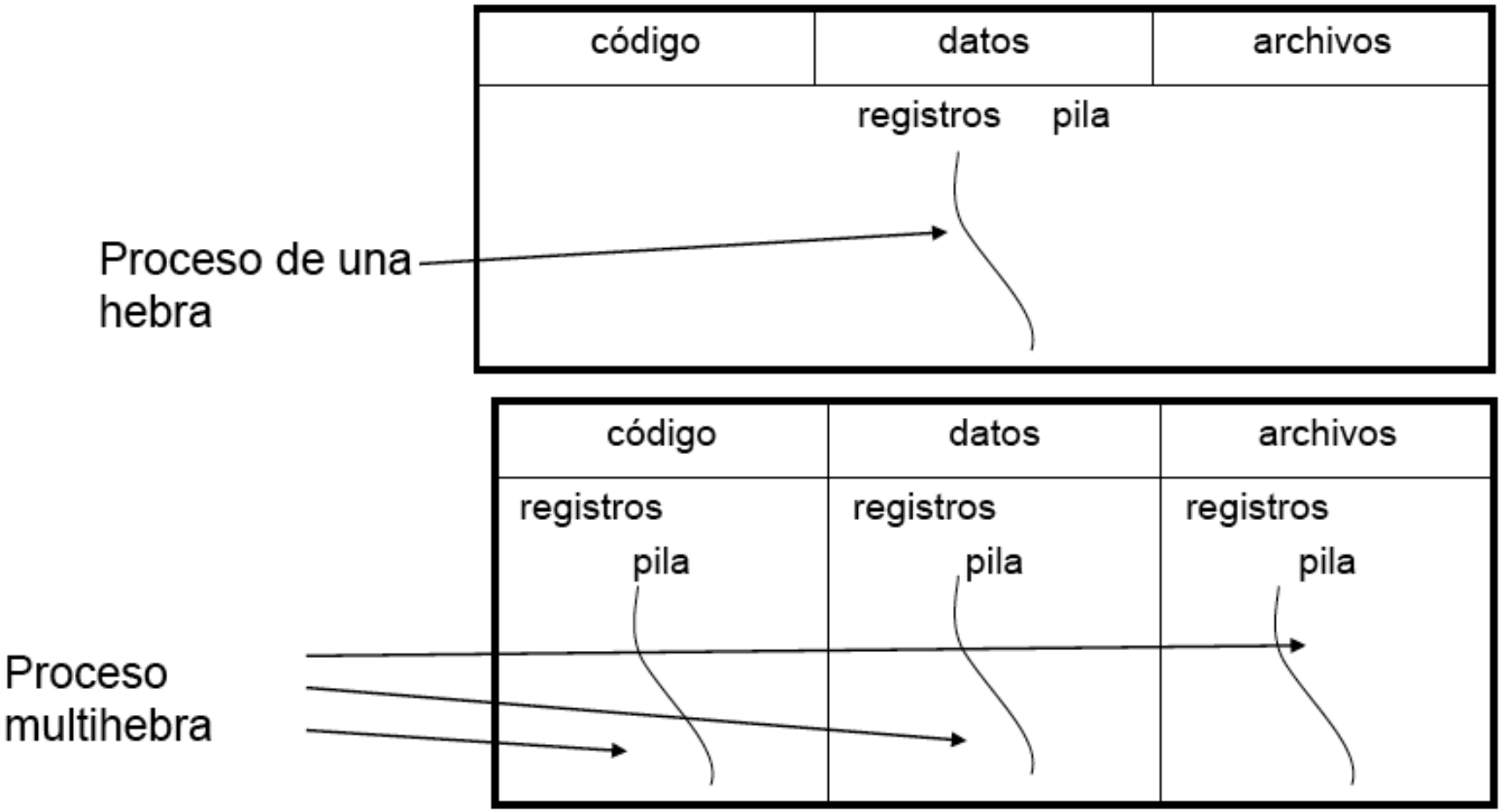
Thread scheduling

- A thread can be defined as the basic unit of CPU usage.
- Every process has at least one thread. If it has more than one thread it con perform several tasks concurrently.
- The difference between a process with several (*threads*) and various processes, is that threads in the same process share the same address space

Thread scheduling

- Threads inside a process share: code segment, data segment, resources (open files, signals . . .).
- For each thread: identifier, program counter, registers, stack.

Thread scheduling



Thread scheduling

Advantages

- higher response capability: if one thread blocks, other threads can continue to execute.
- There may be several threads sharing the same resources (memory, files . . .).
- less expensive than creating processes. Context switch is also lighter.
- Can take advantage of multiprocessor architectures.

Thread scheduling

- We can think of scheduling at two levels: processes and threads.
- A process scheduler chooses a process. Then a thread scheduler chooses the thread.
- There's no preemption among threads. If a thread uses up all the *quantum* another process is selected. When it returns to CPU the same thread will continue.
- If the thread does not use all the *quantum*, the thread scheduler can select another thread inside the same process.

Multiprocessor Scheduling

- Scheduling gets more complicated when we have several processors .
- Assigning different sized jobs to several processors in a optimal way is a combinatory problem (NP complexity).
- They are usualy multithreaded as well.
- Asymmetric Multiprocessing: One processor is in charge, the others just execute the processes they are assigned.
- Symmetric Multiprocessing: Each processor has its own scheduling. Sometimes they share the ready to run queue, in this case extra care must be taken that one process doesn't end up in more than one processor.

Multiprocessor Scheduling

- Even when we have several identical cores (or processors), not all of the are equally convinient for a given thread.
- If thread *A* has been executing longer in CPU1, its cache will be filled with data from *A*. We call this affinity.
- Affinity algorithms work at two levels:
  1. First they assign a group of threads to each processor
  2. Then they make the internal scheduling for each CPU
- Advantage: greatest cache affinity. *Possible drawback*: leave some CPU idle.

Multiprocessor Scheduling

- Load balancing: we try to keep the activity balanced among the different CPUs.
- Forced migration: the work load of the processors is checked periodically and a migration of processes is imposed when there's a need to balance the work loads.
- Requested migration: an idle processor extracts a process from another processor's queue.

Unix process scheduling

Traditional unix scheduling

- Preemptive priorities recalculated dynamically
  - Always is run the process with the highest priority
  - Smaller numbers indicate greater priorities
  - Priority of a process decreases as the process uses CPU
  - Priority of a process increases as it spends time in the ready to run queue
  - When a process with higher priority than the one in CPU appears ready, in preempts the one in CPU (which goes into the ready to run state),unless the one in CPU is running in kernel mode, in which case it will be preempted when it returns to user mode (unless it blocks or ends)
- Processes of the same priority share the CPU in *round robin*.
- user mode priority is recalculated attending to
  - *nice* factor: controlled by the *nice()* system call
  - CPU usage: higher CPU usage (recent) means lower priority

Traditional unix scheduling

- The `proc`structure has the following members related to priority recalculation:

`p_cpu`	cpu usage for the purpose of priority recalculation
`p_nice`	*nice*
`p_usrpri`	user mode priority, recalculated periodically from cpu usage and nice factor *nice*
`p_pri`	process priority, this is the one used for scheduling

- When the process runs in user mode `p_pri` is identical to `p_usrpri`
- After a process has blocked, when it is awaken, `p_pri` is asigned a value depending on the reason the process was bocked. It is called a *kernel priority* or *sleep priority*
  - This *kernel priorities* are smaller numbers thus higher priorities than user mode priorities `p_usrpri`
  - The goal is to make processes complete the system calls faster

Traditional unix scheduling

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

Recalulation of user mode priorities

- Every clock *tic*, the *handler* increments `p_cpu` of the currently running process
- Every second, the routine *shedcpu()*
  1. adjusts `p_cpu` using
     - BSD: $p\_cpu = \frac{2*systemload}{2*systemload+1} * p\_cpu$
     - System V R3: $p\_cpu = \frac{p\_cpu}{2}$
  2. and then it recalculates the user mode priorities
     - BSD: $p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2 * p\_nice$
     - System V R3: $p\_usrpri = PUSER + \frac{p\_cpu}{2} + p\_nice$
- PUSER is a number added, so that the user mode priorities are lower (represented by higher numbers) than the *kernel priorities*

Example of scheduling in SVR3

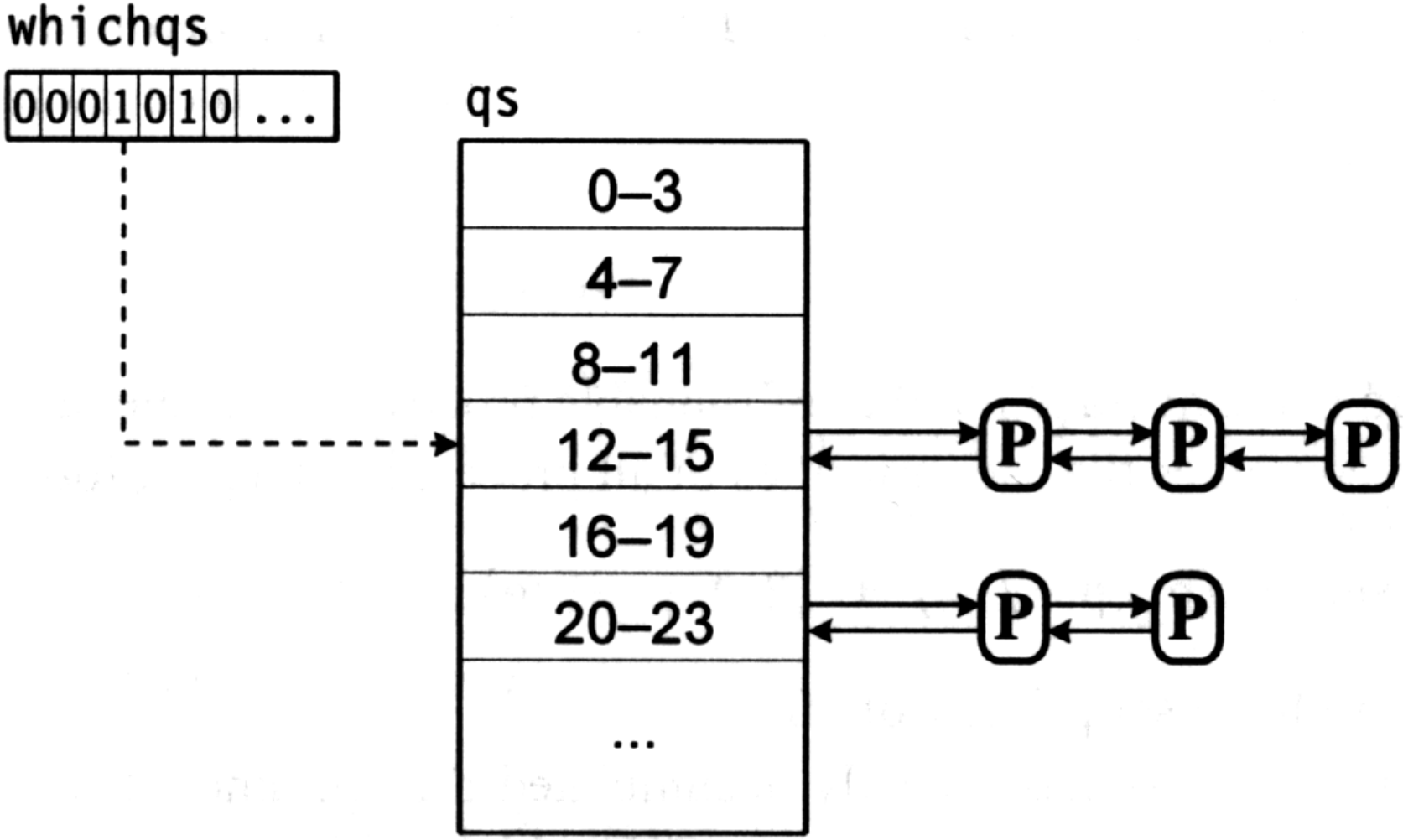
- In this example, the clock *tic* happens 60 times per second, PUSER is 40, and the three processes in the example have a value of 20 as *p_nice*.

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

Traditional unix scheduling: Implementation

- Is implemented as a array of multilevel queues
- Usually 32 queues. Each of them with several adjacent priorities
- After recalculating its priority, a process is moved to the appropiate queue
- *swtch()* just loads the first process of the first non empty queue
- Each 100ms the *roundrobin()* routine changes to the next process in the same queue

Traditional unix scheduling: Implementation



BSD scheduler data structures.

Traditional unix scheduling: Context switch

Theres a context switch when

a. Currently running process blocks or ends
b. A as result of priority recalculation there appears ready a process with higher priority than the currently running process
c. An interrupt handler (or the currently running process) wakes up (*unblocks*) a higher priority process
    - a *voluntary* context switch. *swtch()* is called from *sleep()* o *exit()*
    - b,c *involuntary*, context switch, it happens in kernel mode: the kernel uses a flag (*runrun*) to indicate that a context switch should be done (by calling *swtch()*) when returning to user mode

Traditional unix scheduling: shortcommings

This kind of scheduling has the following shortcommings

- It does not scale well
- There's no means to guarantee a certain amount of CPU to a process (or group of processes)
- There's no guarantee on the response time
- Posibility of *priority inversion*

Priority inversion

- *priority inversion* is an anomaly, present in some scheduling implementations where a lower priority process prevents a higher priority from using the CPU. Example
  - P_1 Very low priority process, has allocated a resource
  - P_2 Higher priority process, blocked on the resource allocated to P_1
  - P_3 Low priority process, higher than P_1 and lower than P_2
  - As P_2 is blocked, the higher runnable process is P_3, which gets CPU before P_2, which is a higher priority process, but P_2 is blocked until P_1 releases the resource, which dependes on P_1 getting the CPU. But this won't happen as P_1 is lower priority than P_3. As a result: P_2 is in fact waiting for P_3
- The way to avoid this situation is by *priority inheritance*. In this case, as P_1 holds a resource that blocks P_2, P_1 *would inherit* P_2's priority while holding that resource.

System V R4 scheduling

- It includes real time applications
- It separates scheduling policy from implementation mechanisms
- New scheduling policies can be implemented
- It limits applications latency
- Priorities can be *"inherited"* to avoid *priority inversion*

System V R4 scheduling

Some *Scheduling classes* are defined and they determine the policies applied to the processes belonging to them

- Class independent routines
  - Queue manipulation
  - Context switch
  - Preemption
- Class dependent routines
  - Priority recalculation
  - Inheritance

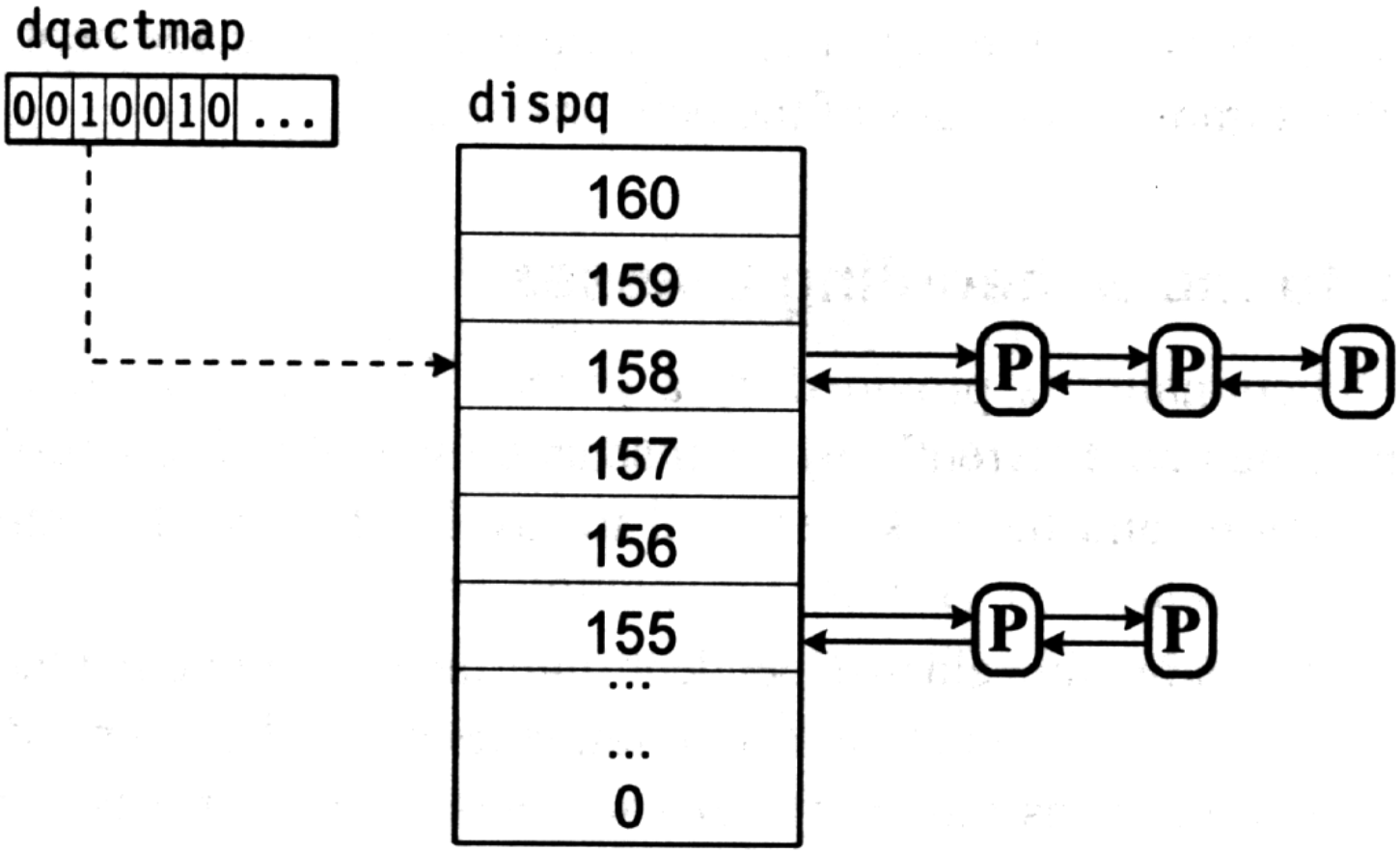
System V R4 scheduling

- Priorities range from 0 to 159: the higher the number the higher the priority
  - 0-59 *time sharing class*
  - 60-99 *system priority*
  - 100-159 *real time*
- In the `proc` structure

`p_cid` class identifier
`p_clfuncs` pointer to class functions
`p_clproc` pointer to class dependent data

- Is implemented as an array of multilevel queues

System V R4 scheduling: Implementation



SVR4 dispatch queues.

System V R4 scheduling

- Several predefined classes
  - clase *time sharing*
  - clase *system*
  - clase *real time*

Sample listing of classes available on a running system

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

System V R4 scheduling: *time sharing* class

- User mode priorities are recalculated dynamically
- When a process blocks it is assigned a *sleep priority* depending on the reason it blocked. When it returns to user mode, its user mode priority is used
- Quantum depends on priority: higher priority $\Rightarrow$ shorter quantum
- ONLY the process leaving the CPU gets its priority recalculated
  - **used up all of its quantum**: Its user mode priority gets lower
  - **blocked before using all of its quantum**: its priority raises

System V R4 scheduling: *time sharing* class

- Class dependent data
  - `ts_timeleft` remaining time of quantum
  - `ts_cpupri` part of the user mode priority imposed by the system (what gets actually recalculated)
  - `ts_upri` part of the user mode priority assigned by the user (with the *priocntl()* system call)
  - `ts_umdpri` user mode priority (`ts_cpupri` +`ts_upri`)
  - `ts_dispwait` seconds elapsed since the process started its quantum

System V R4 scheduling: *time sharing* class

- The system has a table that relates quantum, priorities and how the priorities get recalculated (*ts_cpupri*)
- items in that table are

`quantum` time quantum corresponding to each priority level

`slpret` new `ts_cpupri` if the quantum is not expired (process blocks before expiring time quantum)

`tqexp` new `ts_cpupri` if quantum expires

`maxwait` seconds form the quantum start to use *lwait* as new *cpupri*

`lwait` new *cpupri* if more than *maxwait* elapsed since the start of its quantum

`pri` priority, used both to relate *umdpri* with the quantum and to recalculate *cpupri*

System V R4 scheduling: *time sharing* class. *TS class* scheduling table

```
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
```

System V R4 scheduling: *time sharing* class. *TS class* scheduling table

```
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
```

Example of scheduling in the *time sharing* class

- Lets consider a process with ts_upri=3 and ts_cpupri=5
- Its user mode priority is ts_umdpri=ts_upri + ts_cpupri= 3+5=8
- When it gets to the CPU it would get a *quantum* de 200ms
- If the process uses up all of its *quantum* ts_cpupri would be assigned a value 0. Its user mode priority would be in this case 3, which gets a *quantum* of 200
- If the process does not use all of its *quantum*, ts_cpupri would be reevaluated to 50 and its new user mode priority would be 53, which gets a *quantum* of 40

Planificación en System V R4: *real time* class

- Uses priorities 100-159
- Fixed priorities and quantums. Can only be changed with the *priocntl()* system call
- kernel maintains a table with the corresponding time quantums for each priority (although they can be changed with the *priocntl()* system call)
- the higher the priority, the shorter the default quantum
- after using up its quantum, the process returns to the same queue (at the end)

Planificación en System V R4: *real time* class

To make real time processes compatibles with non preemptible kernels:

- When a process is running in kernel mode, and a real time process appears ready, it cannot preempt inmediately (kernel might not be in a consistent state). The real time process will get the CPU when the current running process
  - blocks
  - returns to user mode
  - reaches a *preemption point*
- A realtime process ready is indicated by the flag *kprunrun*
- A preemption point is a point inside kernel code where its safe to preempt (kernel data are in a consistent state) so the *kprunrun* is checked and context switch is initiated if needed
- Solaris uses fully preemptible kernel (all kernel data structures are protected by semaphores) son no preemption points are necessary

System V R4 scheduling: *real time class*. *RT class* scheduling table

```
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
```

System V R4 scheduling: *real time class*. *RT class* scheduling table

```
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
```

System V R4 scheduling: *system* class

- Not accesible in all installations
- Used for special system processes such as *pageout*, *sched* or *fsflush*
- Fixed priorities
- In the 60-99 range

```
bash-2.05$ ps -lp 0,1,2
 F S   UID   PID  PPID  C PRI NI     ADDR      SZ     WC
19 T     0     0     0  0   0 SY        ?       0
 8 S     0     1     0  0  41 20        ?      98
19 S     0     2     0  0   0 SY        ?       0
```

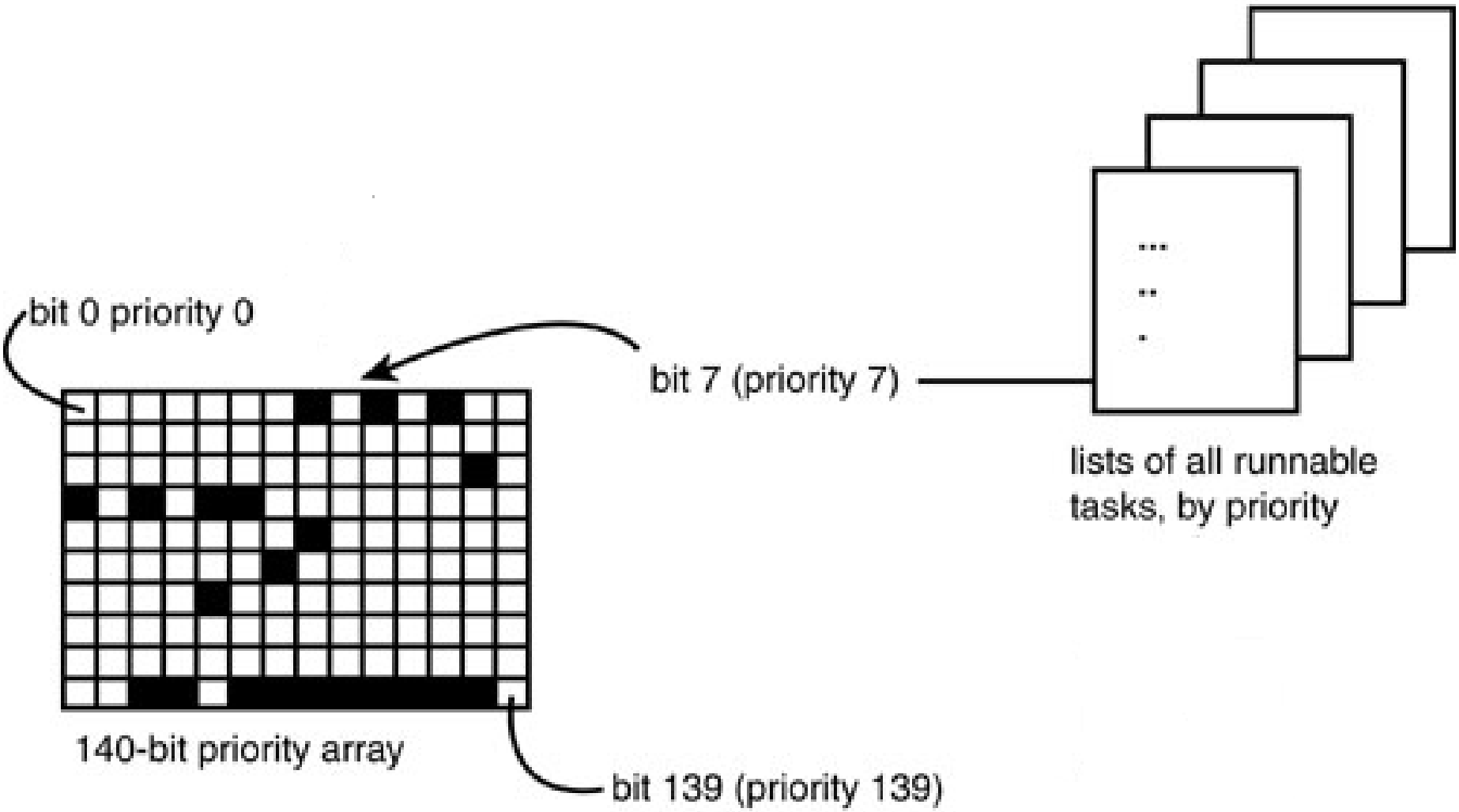
Linux scheduling

- Linux distinguishes between two types of processes
  - Real time processes: Fixed static priority between 1 y 99. Its priority doesn't change and the higher priority runnable process gets the CPU. Two kinds of real time processes; RR and FIFO
  - Normal processes: correspond to a static rpiority of 0. They execute if no real time process is ready to run. For them a preemptive dynamic priority algorithm is used. The system recalculates their priorities and quantums according to the values specified by *nice* and/or *setpriority*.

Linux scheduling

- CPU scheduling is done for time intervals called *epochs*
- For each *epoch* every process has its time *slice* depending on its priority
- The system has a *runqueue* for each processor and each process can only be in one *runqueue* at a time
- Each *runqueue* has two structures: the *active array* and the *expired array*. Each array has a process queue for each priority level
- When a process uses up all of its *slice*, it slice gets recalculated and the process is moved to the *expired array*. When all processes have used up all of their *slices* the *expired array* becomes *active array*
- A array of bits indicates the non empty queues

Array de colas en linux



system calls for priority management

- In unix there exist several system calls and library functions to control process priorities
- Not all calls are available in every system
  - *nice()*
  - *setpriority() y getpriority()*
  - *rtprio()*. Specific to some BSD systems
  - *priocntl()*. Specific to System V R4
  - POSIX: *sched_setscheduler(), sched_getscheduler(),. . .*

Unix scheduling: *nice()*

- Avalilable in all systems: *nice()*

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

- Changes the *niceness* of the calling process. For traditional unix systems that is the *p_nice* factor
- It takes the *nice* increment as its argument
- return the *niceness* minus 20. *niceness* is a number between 0 and 40. The call returns a number between -20 (maximun priority) and 20

Unix scheduling: *getpriority()* y *setpriority()*

getpriority() and *setpriority()*

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

- Available in almost every system
- They change the same scheduling parameters as the *nice* system call
- Better interface to priority than *nice()*, as a process, with the right credentials, can check and/or modify other processes' priorities

Unix scheduling: *rtprio*

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
```

Unix scheduling: *rtprio*

rtprio

- Available in HP/UX and some BSD system (freeBSD, dragonfly ..)
- `RTP_PRIO_REALTIME` processes have static priorities higher than other processes in the system
- `RTP_PRIO_NORMAL` processes have dynamic priorities that get recalculated attending to the *nice* factor and CPU usage
- `RTP_PRIO_IDLE` processes have static priorities LOWER than other processes in the system

Unix scheduling: *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.                    */
```

Unix scheduling: *priocntl()*

cmd

`PC_GETCID`

`PC_GETCLINFO`

`PC_SETPARMS`

`PC_GETPARMS`

`PC_GETCID` and `PC_GETCLINFO` parameters

```
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 */
```

Unix scheduling: *priocntl()*

PC_GETPARMS and PC_SETPARMS parameters

```
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;
```

Unix scheduling: POSIX calls

POSIX calls

```
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

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;
```

Unix scheduling: POSIX calls

- It distiguishes there scheduling policies with three different static priorities
  - SCHED_OTHER: Normal processes. They have a static priority of 0 so they only get to execute if there in no `SCHED_FIFO` or `SCHED_RR` ready to run. They schedule using dynamic preemptive priorities recalculated from *niceness* and CPU usage.
  - SCHED_RR: Real time processes, static priorities, *round robin* scheduling
  - SCHED_FIFO: Real time processes, static priorities, FIFO scheduling
  - SCHED_BATCH: Static priority 0. Similar to SCHED_OTHER, execpt that the system assumes they are CPU intensive
  - SCHED_IDLE: Similar to the ones in BSD systems. Not available in every installation.

Unix Processes: Executing in kernel mode

Executing in kernel mode

- These three events change the execution into kernel mode
  - **Device interrupt:** Asynchronous to the running process. An external device needs to communicate with the O.S. It can happen at any time: process runing in user mode, process running in kernel mode, even when another interrupt service routine is being run.
  - **Exception:** Synchronous to the running process and caused by it (division by 0, invalid addressing, illegal instruction . . .)
  - **System call:** Synchronous to the running process. The process in CPU explicitly ask the O.S. for something

Executing in kernel mode

- In any of these cases, the O.S. kernel takes the control
  - Saves the process context in its kernel stack
  - Executes the function corresponding to the event it is dealing with
  - When the routine is completed, restores the process to its previous state (as does with the running mode)

Executing in kernel mode: interrupt

- An interrupt can happen at any time, even if the O.S. is already dealing with another interrupt
- Each interrupt is assigned an Interrupt Priority Level (IPL-**I**nterrupt **P**riority **L**evel)
- When an interrupt occurs its *ipl* is compared with the current *ipl*. If it is higher the corresponding handler is invoked, if not, execution of its handler is postponed until *ipl* drops enough
- All user code and most of kernel code (except interrupt service routines and little fragments of code in some system calls) is run at *ipl* minimum
- Ipl goes from 0 to 7 in traditional unix systems and from 0 to 31 in BSD

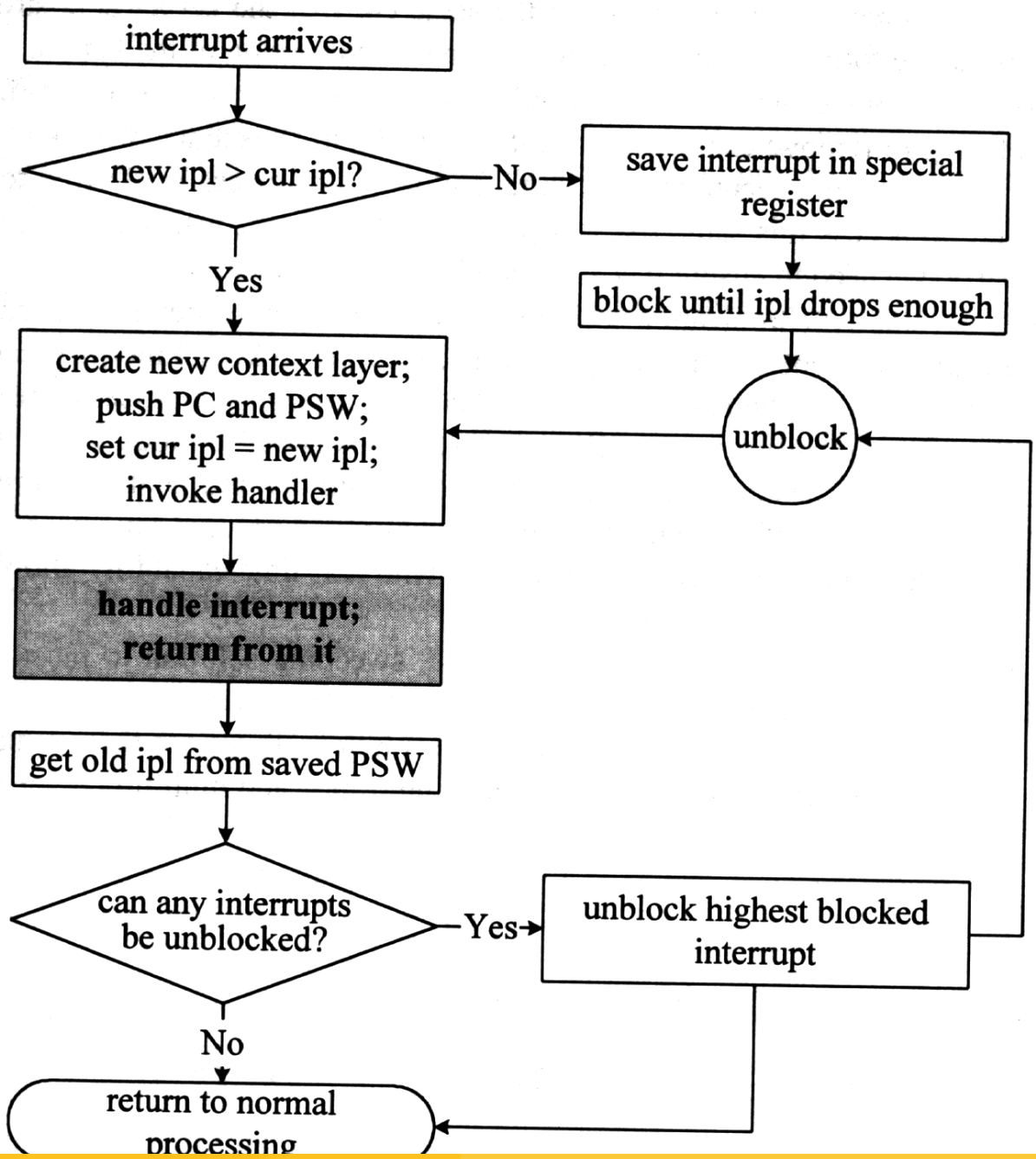

interrupt arrives
new ipl > cur ipl?
No
save interrupt in special register
block until ipl drops enough
Yes
create new context layer;
push PC and PSW;
set cur ipl = new ipl;
invoke handler
unblock
handle interrupt;
return from it
get old ipl from saved PSW
can any interrupts be unblocked?
Yes
unblock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

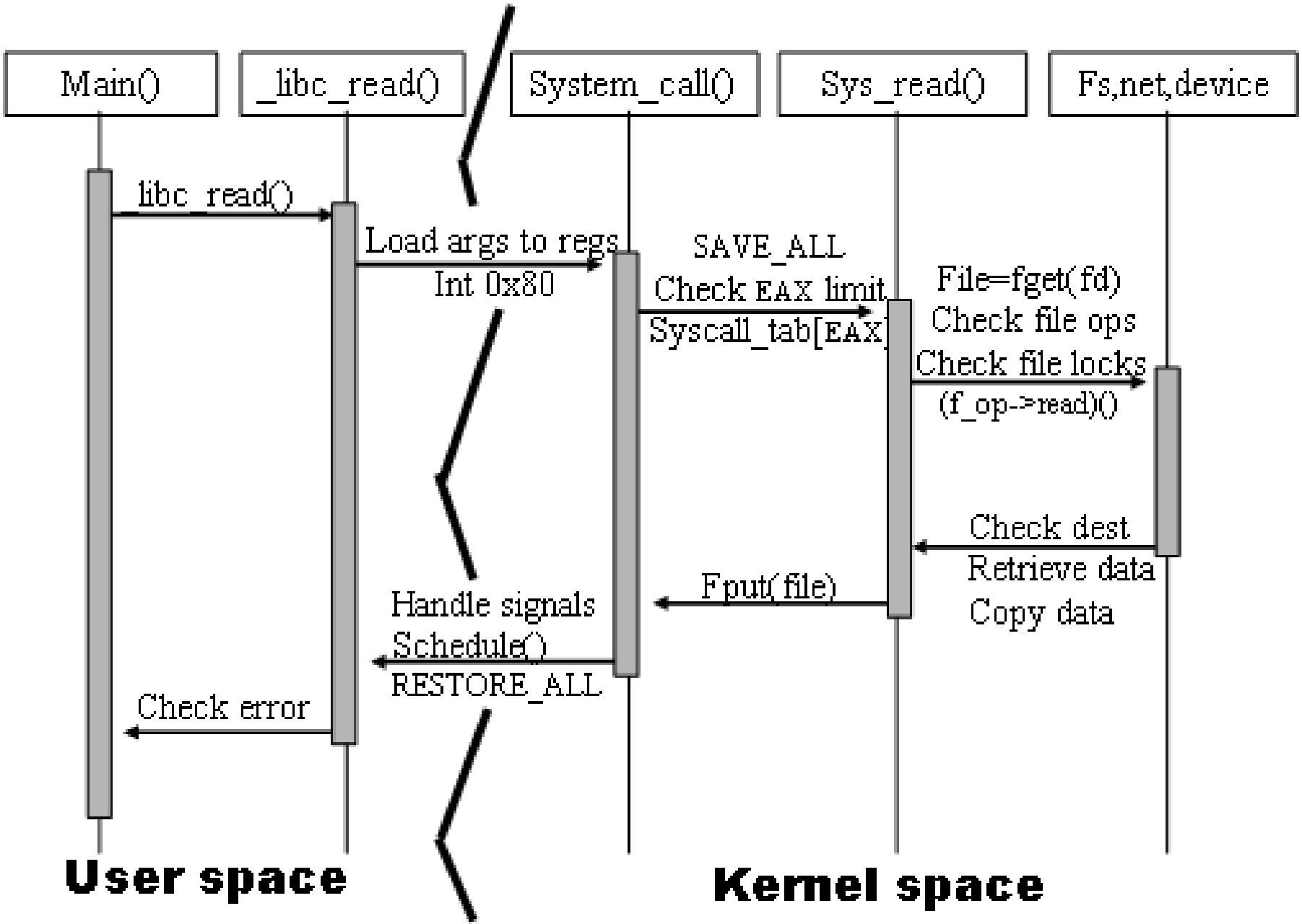
Executing in kernel mode: system call

What we see is a wrapper library function (*open(), read(), fork() . . .*)

- **library function (for example `read()`)**
- It gets its parameters in the user stack
- Pushes the service number onto the stack (or at an specific processor register)
- It executes a special intruction (*trap, chmk, int . . .*) that changes execution into kernel mode. This instruction, besides changing execution mode, transfers control to the system call handler *syscall()*
  - `syscall()`

Executing in kernel mode: system call

- - **`syscall()`**
  - Copies the arguments to the `u_area`
  - Saves process context in its kernel stack
  - Uses the service number as an index into an array (`sysent[]`) which indicates which kernel function should be called (for example `sys_read()`)
    - **function called by `syscall()`: f.e. `sys_read()`**
    - Is the one providing the service
    - Should it have to call other functions inside the kernel, it uses the kernel stack
  - It puts the return (or error) values in the corresponding register
  - Restores the process context and returns to user mode returning control to the library function
- Returns control and values to the calling function


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

System call numbers in 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
```

System call numbers in 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 *" */
```

System call numbers in 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
```

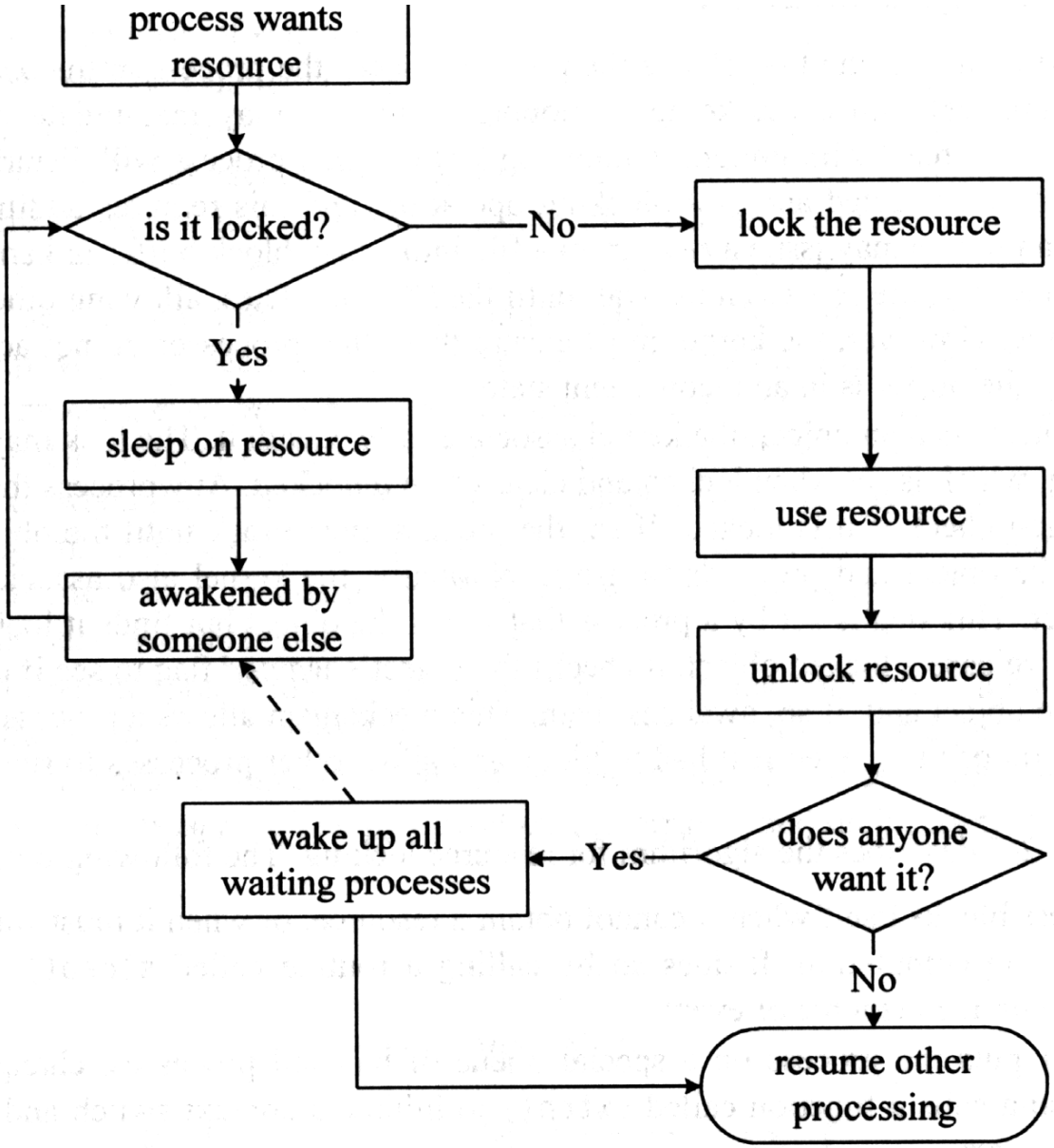
Executing in kernel mode: resources

- Elementary protection of kernel data: a process running in kernel mode cannot be preempted until it returns to user mode waits or ends
- As it can not be preempted it can manipulate kernel data without risking to create inconsistencies
- When a process using a resource goes to sleep it must mark the resource as busy: before using a resource a process must check whether it is busy, it it is, it marks the resource as *wanted* and calls *sleep())*.
  - *sleep()* puts the process to sleep and calls *swtch()* to initiate the context switch

Executing in kernel mode: resources

- When a resource is released, it is marked as non-busy and, if it is also maked as *wanted* **ALL** processes that are *sleeping on it* (waiting for it to be marked not busy) are waken up
  - *wakeup()* finds all processes sleeping on a resource, changes their state to *ready to run* and places them in the runnable queue
- An awaken process may not be the first one to obtain the CPU, so the first thing it has to do is to re-check if the resource is in fact available (another process may have got it)

Executing in kernel mode: resources



Executing in kernel mode: resources

- Even though a process running in kernel mode can not be preempted, an interrupt can occur at any time
- When accessing kernel data that might be accessed by some interrupt service routine, that interrupt should be disabled by rising the *ipl*
- Some care must be taken
  - Interrupt require fast servicing: disabling time should be kept to a minimum
  - Disabling some interrupt also disables those with a lower *ipl*
- If we want the kernel to be fully preemptible, kernel data structures must be protected with semaphores
- More complex mechanisms should be used in multiprocessors systems

Unix processes: Signals

Signals

- Kernel uses signals to notify processes or asynchronous events. Examples:
  - When cntrl-C is pressed, the kernel sends SIGINT
  - When a communication line goes down, the kernel sends SIGHUP
- User processes can send each other signals using the *kill* system call
- Processes respond to signals when they return to user mode. Except for SIGKILL and SIGSTOP several actions are possible
  - Terminate the process
  - Ignore the signal: nothing is done
  - User defined action: signal handler

Signals

- Upon receiving a signal, the kernel sets a bit in the correspondent member of the `proc` structure
  - If the process is running in kernel mode, nothing gets done until it returns to user mode
  - If the process is blocked
    - If it is an interruptible wait, the kernel interrupts the system call (which would return -1 setting errno to EINTR), and the signal is dealt with when the process returns to user mode
    - If it is an non-interruptible wait, the signal is dealt with when the process returns to user mode after it has completed the system call in which it was waiting

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.

Signals in System V R2

- 15 available signals
- signal related system calls
  - *kill (pid_t pid, int sig)*: to send signals
  - *signal (int sig, void (*handler)(int))*: to set a signal handler. *handler* can be:
    - SIG_DFL: signal default action (which is either ignore the signal or terminate the process)
    - SIG_IGN: signal is ignored (nothing is done)
    - Address of the functio nwhich be called upon receiving the signal. This function returns *void* , and gets an interger argument (the number of the signal which cause its calling)
- In the process u_area there is an array, indexed by signal number, with the signal handlers for each signal (or SIG_IGN o SIG_DFL should the signal be ignored or at its default action)

Signals System V R2

- Sending the signal is setting the corresponding bit in a member of the `proc` structure
- When the process is about to return to user mode, if there is some signal pending for this process, the kernel clears that bit; if the signal si ignored nothing is done, but if there is a handler installed, the kernel does the following :
  - Creates a context layer in the user stack
  - Restores the signal to its default action
  - Sets PC (program counter) to the addres of the handler, so the first thing to execute upon returning to user mode, is the signal handler,
- Signal handlers are **NO PERMANENT**
- If a signal arrives during the execution of its signal handler, the current associated action is performed

Señales en System V R2

- The following code could be interrupted by pressing Cntrl-C twice

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

- To make the handler permanent we could reinstall it

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

- The program is also terminated if we manage to press control-C for a second time before the signal system call inside `manejador` is executed

Signals in System V R3

- System V R3 introduces reliable signals
  - permanent handlers (the kernel does not restore the signal default action once received the signal)
  - the handler for a signal runs with that signal masked
  - ability to mask and unmask signals
  - information on signals received, masked or ignored is now in the `proc` structure
- System V R3 has these new system calls
  - *`sigset (int senal, void (*handler)(int))`*. Install a handler for signal *senal*. This handler is permanent an cannot be interrupted by *senal*
  - *`sighold (int senal)`*. Masks (blocks) a signal
  - *`sigrelse (int senal)`*. Unmasks (unblocks) a signal
  - *`sigpause (int senal)`*. Unmasks *senal* and blocks the process until a signal is received

Signals in BSD

- It allows to mask signals in groups (in System V R3 signals could be masked only one at a time)
- If a system call is interrupted by a signal, it gets restarted automatically (this behaviour is configurable with *siginterrupt*)
- BSD has these new system calls
  - `sigsetmask(int mask)` Sets the set of masked signals (mask can be manipulated with *int sigmask(int signum)*)
  - `sigblock(int mask)` Masks (blocks) the signals in *mask*
  - `sigpause(int mask)` Sets the current signal mask and blocks the calling process until a signal is received
  - `sigvec(int sig, struct sigvec *vec, struct sigvec *ovec)` Installs a handler for signal *sig*
  - `int sigstack(struct sigstack *ss, struct sigstack *oss)` Allows to specify an alternative stack for running signal handlers.

Signals in System V R4

- In includes previous system's functionality
- It is *standard* on nowadays systems
- System calls
  - `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)`

Signals in System V R4

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

- Establishes the set of a process masked (blocked) signals depending on *how*
  - SIG_BLOCK signals on *set* are added to the set of masked (blocked) signals of the process
  - SIG_UNBLOCK signals on *set* are removed from the set of masked (blocked) signals of the process
  - SIG_SETMASK signals on *set* become the set of masked (blocked) signals of the processl
- *oldset* shows the previous set
- Signal sets are of type *sigset_t* and can be manipulated with

```
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)
```

Signals in System V R4

int sigsuspend(const sigset_t *mask)

- Sets the mask of blocked signals and blocks the process until a signal (neither blocked nor ignored) is received.

int sigpending(sigset_t *set)

- Checks whether a signal has been received

sigaltstack(const stack_t *ss, stack_t *oss)

- Allows to specify an alternate stack for the execution of handlers

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

- More shophisticated than *kill* calls to send signals

Signals in System V R4

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

- Installs a handler for signal *sig*
- struct *sigaction* has the following members
  - **sa_handler** SIG_DFL, SIG_IGN or the address of the signal handler
  - **sa_mask** signals to block DURING execution of the handler
  - **sd_flags** tunes the handler behaviour. Bitwise OR of the following
    - SA_ONSTACK handler runs on alternate stack
    - SA_RESETHAND non permanet handler (signal returns to its default action once handler is called)
    - SA_NODEFER signal is not blocked during execution of its handler
    - SA_RESTART if signal interrupts a system call, it is restarted automatically
    - other flags: SA_SIGINFO, SA_NOCLDWAIT, SA_NOCLDSTOP, SA_WAITSIG

SIgnals in System V R4

This is an infinite loop if cntrl-C is pressed within 5 seconds

```
#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

This produces a stack overflow if cntrl-C is pressed within 5 seconds

```
#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);
}
```

Signals in System V R4: implementation

- In the `u_area`
  - `u_signal[]` array of handlers
  - `u_sigmask[]` signal mask for each handler
  - `u_sigaltstack` alternate stack
  - `u_sigonstack` signals that run on alternate stack
  - `u_oldsig[]` signals with 'old' (System V R2) behaviour
  - form system V R4 u_area:

```
.....
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 */
.........
```

Signals System V R4: implementation

- In struct `proc`
  - `p_cursig` signal being handled
  - `p_sig` pending signals
  - `p_hold` blocked signals
  - `p_ignore` ignored signals
  - from `proc` structure in 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 */
........
```

Unix processes: Inter Process Communication

Interprocess Communication

The main mechanisms to intercommunicate processes in unix are

- *pipes*
- shared memory
- semaphores
- message queues

Inter Process Communication: *pipes*

- They are temporary files created with the *pipe()* system call

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

- This call returns two file descriptors (fildes[0] and fildes[1])
- On some systems they can be both used with read and write system calls
- On other systems however, fildes[0] is for reading and fildes[1] for writing (historical standard)
- When the *pipe* is empty, *read()* blocks and when the pipe is full *write()* blocks
- Data in the *pipe* are discarded as they are being read

Inter Process Communication: *shared memory*

As IPC resources are shared by several processes, it becomes necessary that different processes can refer to the same resource: Every IPC resource in the system is identified by a number (its key).

1. First it is necessary to get a memory block (creating it or using one already created)

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

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

   - **key**: number identifying the resource on the system
   - **size**: size of the shared memory region (some systems impose a minimum)
   - **shmflg**: Bitwise OR of the permissions and one or more flags

flags on IPC resources *get* system calls

- Available flags are:
  - `IPC_CREAT`
  - `IPC_EXCL`
- Used as follows
  - **0** If the resource already exists it returns an identifier, otherwise error is returned.
  - **IPC_CREAT** If the resource already exists it returns an identifier, otherwise it is created and an identifier for the created resource is returned.
  - **IPC_CREAT | IPC_EXCL** If the resource already exists an error is returned, otherwise it is created and an identifier for the created resource is returned.

Inter Process Communication: *shared memory*

2 Once created, to be accessible, shared memory must be placed in the process's address space.

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

void *shmat(int shmid, const void *shmaddr, int flg
```

- **shmid**: id returned by *shmget()*
- **shmaddr**: virtual address to place the shared memory segment onto (NULL to get it assigned by the system)
- **flg**: `SHM_RND`, `IPC_RDONLY`, `SHM_SHARE_MMU` (Solaris) `SHM_REMAP` (linux)

shmat() returns the virtual address where the shared memory can be accessed

Inter Process Communication: *shared memory*

3 when it is no longer needed shared memory can be detached from the process address space

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

4 There also exist a control system call, that allows, among other things, to remove a shared memory region from the system

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

- **shmid**: id returned by *shmget()*
- **cmd**: action to perform: `IPC_RMID`, `SHM_LOCK`, `SHM_UNLOCK`, `IPC_STAT`, `IPC_SET`...
- **buf**: information

Inter Process Communication: *shared memory*

- The following function obtains a shared memory address from the key and the size (NULL in case of some error). If the option to create is specified the memory will be created unless it already exists, in which case in returns an error

```
void * ObtenerMemoria (key_t clave, off_t tam, int crear)
{
 int id;
 void * p;
 int flags=0666;

 if (crear)
   flags=flags | IPC_CREAT | IPC_EXCL;

 if ((id=shmget(clave, tam, flags))==-1)
   return (NULL);

 if ((p=shmat(id,NULL,0))==(void*) -1){
   if (crear)
      shmctl(id,IPC_RMID,NULL);
   return (NULL);
   }

 return (p);
}
```

Inter Process Communication: *semaphores*

Semaphores are useful to sync up processes. System V IPC's interface provides arrays of semaphores

1. As with the shared memory, the first step is to get the resource (either creating it or using one already created)

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

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

   - **key**: number identifying the resource on the system
   - **nsems**: number of semaphores in the array
   - **semflg**: as the previously described shmflag

Inter Process Communication: *semaphores*

2 Once created, operations can be performed on one (or more than one) semaphores in the semaphore array

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

int semop(int semid, struct sembuf *sops, size_t ns
```

- **semid**: id obtained with *semget()*
- **sops**: pointer to a (or some) struct sembuf, each of which describes an operation to be performed on the array

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

- **nsops**: number of operations to perform

Inter Process Communication: *semaphores*

3 There also exists a control system call, that allows, among other things, to remove the resource, initialize the semaphores . . .

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

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

- **shmid**: identifier returned by *semget()*
- **semnum**: semaphore number onto which operation is to be performed
- **cmd**: action to perform : `IPC_RMID, IPC_STAT, IPC_SET, GETALL, SETALL, GETVAL, SETVAL ...`
- **fourth argument**: information

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

Inter Process Communication: *message queues*

A message queue allows the passing of messages among processes (much more sophisticated than a *pipe*)
A message is any piece of information sent together. It does not have a predefined format.
Its first 32 bits define the *type* of message. The system call to receive messages can specify the *type* of the message to receive.

1. As in ther IPC resources, it is first necessary to get the resouce

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

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

   - **key**: number that identifies the resource on the system
   - **msgflg**: as shmflag seen before

Inter Process Communication: *message queues*

2 Once created the queue, messages can be sent and received

```
#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**: id returned by *msgget()*
- **msgp**: pointer to message
- **msgsz**: message size (*msgsnd()*) or maximun number of bytes to transfer (*msgrcv()*)
- **msgtyp**: type of message to get
  - 0 first in the queue
  - -n first of type less or equal than `n`
  - n first of type `n`
- **msgflg**:`IPC_NOWAIT`, `MSG_EXCEPT` (linux), `MSG_NOERROR` (linux)

Inter Process Communication: *message queues*

3 There also exists a control system call, that allows, among other things, to remove the resource, get information on the queue ...

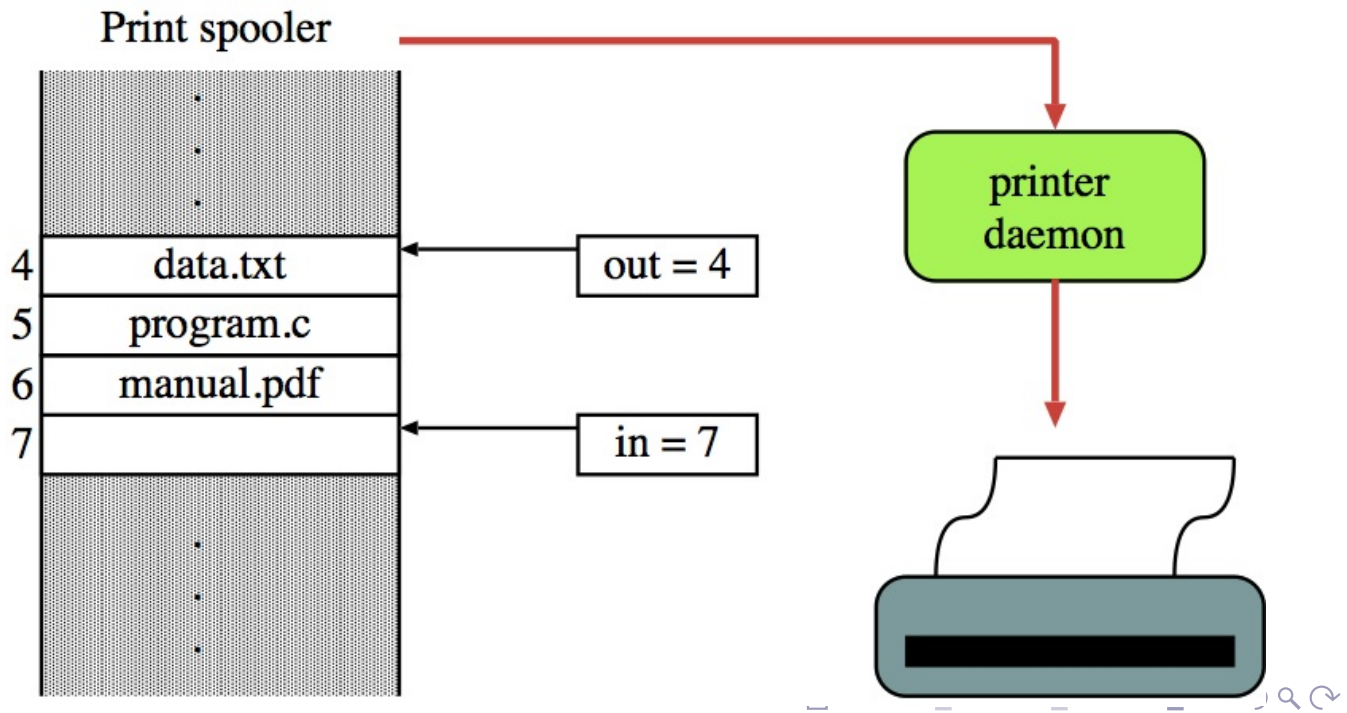
```
#include <sys/msg.h>

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

- **msqid**: id returned by *msgget()*
- **cmd**: action to perform: `IPC_RMID, IPC_STAT, IPC_SET`
- **buf**: information

Concurrence

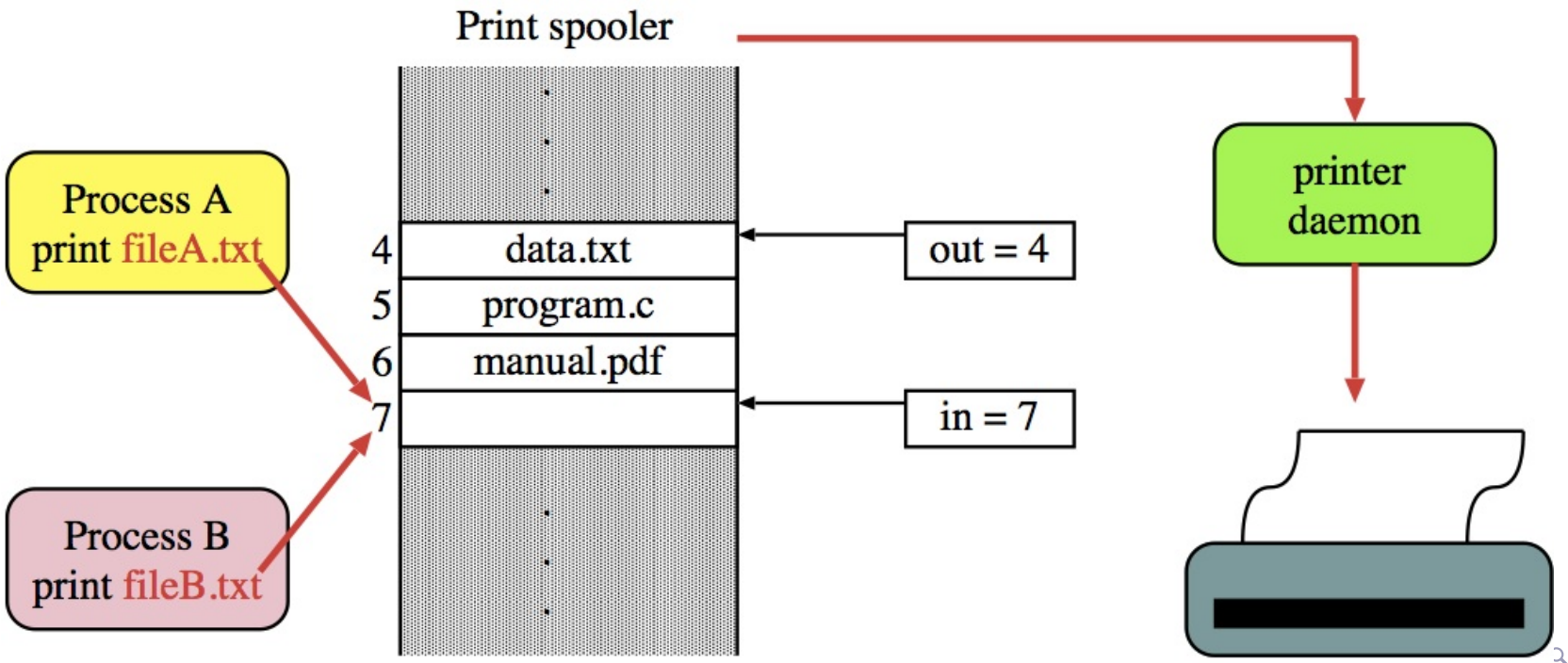
Example: printer spooler

- Spool: Simultaneous Peripheral Operations On-Line. The printer daemon consults the spooler and prints the jobs.
- *out* = next to print; *in* = next free slot. .



Example: printer spooler

- Spool: Simultaneous Peripheral Operations On-Line. The printer daemon consults the spooler and prints the jobs.
- *out* = next to print; *in* = next free slot. .
- Lets think of two processes *A* y *B* which try to print simoultaneously



Example: printer spooler

- The folloing scenario is possible:

time	Process A	Process B
0	regA:=in (regA=7)	
1		regB:=in (regB=7)
2		spooler[regB]:= "fileB.txt"
3		in:=regB+1 (in=8)
4	spooler[regA]:= "fileA.txt"	
5	in:=regA+1 (in=8)	

- Error: printing "fileB.txt" is not happening. Printer daemon is only finding "fileA.txt" in the `spooler[7]`.

Race conditions

- There's some inconsistency in the shared values. *A* does not know that *B* changed $in := 8$ and still thinks $regA = 7 = in$ (wrong)
- This is called a race condition: its result depends on the order of interleaving of processes' instructions.
- They happen with shared resources (variables *in*, *out* and *spooler*).
- Complex to debug. Errors can be infrequent, but possible

Critical section

- How do we avoid *race conditions*??
- Finding critical sections: parts of a process code that manipulate shared resources (in such a way that could produce *race conditions*).
- The solution must provide:
  - Mutual exclusion: at most one process is executing its critical section
  - Independence of the speed or number of processors
  - Progress: A process which is not executing its critical section must not prevent other processes from doing so
  - Limited wait: A process can not wait forever to enter its critical section

Atomicity

- There exist many solutions
- Most of them make use of atomicity. A secuence of operations is said to be atomic if the O.S. guarantees no context switch will occur during its execution. Its execution is indivisible.
- Examples of solutions with atomicity: semaphores (Dijkstra 1965), monitors (Hansen 1973, Hoare 1974)
- The most spread solution in UNIX, is the use of semaphores.

Semaphores

- A semaphore is an integer variable *sem* which, appart from its initialization, can only be accesesed with two operations:
  1. *P*(*sem*) o *Wait*(*sem*). Which executes atomically:

```
wait until sem>0;
sem--;
```

  2. *V*(*sem*) o *Signal*(*sem*). Executes atomically:

```
sem++;
```

- A process blocked at `wait until sem>0` does not use CPU (blocked state)
- When other process executes *Signal*(*sem*), the O.S. takes one of the ones waitting on *sem* and wakes it up. This is done atomically.
- The process waked form *Wait*(*sem*) decrements the counter atomically.
- A binary semaphore can only have the values $\{0, 1\}$.

Solution to the Spooler problem

- We use a binary semaphore *mutex*. Its initial value $mutex = 1$.
- Each time a process prints

```
void printFile(char *fname) {
  wait(mutex);
  load shared var "in" into reg;
  spooler[reg]=fname;
  write reg+1 in shared var "in";
  signal(mutex);
}
```

- To print, the printer daemon also has to access the spooler using mutex

Apéndices

Sample struct `proc` in 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 */
```

Sample struct proc in 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;
```

Sample struct proc in 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;
```

Sample struct proc in 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) */
```

Sample struct `proc` in 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 */
```

Sample struct `proc` in 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
};
```

Sample struct `proc` in 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 */
```

Sample struct proc in 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 */
```

Sample struct proc in 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 */
```

Sample struct proc in 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;
```

Sample u_area in 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.
```

Sample u_area in 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 distribution */
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 */
```

Sample u_area in 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 */
```

Sample u_area in 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:   */
/*     0-15  see x_renv (x.out.h) */
/*    16-23  see x_cpu  (x.out.h) */
/*    24-31  see below   */
```

Sample u_area in 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 */
daddr_t u_limit; /* maximum write address */
```

Sample u_area in 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;
```

Sample u_area in 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 */
```

Sample u_area in 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 */
```

Sample u_area in 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 */
};
```

Sample u_area in 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? */
```

Sample u_area in 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 */
```

Sample u_area in 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;
```

Sample linux 2.6 task_struct I

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;    /* per process flags, defined below */
    unsigned int ptrace;

    int lock_depth;        /* BKL lock depth */

#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
    int oncpu;
#endif
#endif

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;

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

    /*
     * fpu_counter contains the number of consecutive context switches
```

Sample linux 2.6 task_struct II

```
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
     * so that after 256 times the counter wraps and the behavior turns
     * lazy again; this to deal with bursty apps that only use FPU for
     * a short time
     */
    unsigned char fpu_counter;
#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif

    unsigned int policy;
    cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
    int rcu_read_lock_nesting;
    char rcu_read_unlock_special;
    struct list_head rcu_node_entry;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TREE_PREEMPT_RCU
    struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */
#ifdef CONFIG_RCU_BOOST
    struct rt_mutex *rcu_boost_mutex;
#endif /* #ifdef CONFIG_RCU_BOOST */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
    struct sched_info sched_info;
```

Sample linux 2.6 task_struct III

```
#endif

    struct list_head tasks;
#ifdef CONFIG_SMP
    struct plist_node pushable_tasks;
#endif

    struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
    unsigned brk_randomized:1;
#endif
#if defined(SPLIT_RSS_COUNTING)
    struct task_rss_stat    rss_stat;
#endif
/* task state */
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    /* ??? */
    unsigned int personality;
    unsigned did_exec:1;
    unsigned in_execve:1;    /* Tell the LSMs that the process is doing an
                 * execve */
    unsigned in_iowait:1;


    /* Revert to default priority/policy when forking */
    unsigned sched_reset_on_fork:1;
```

Sample linux 2.6 task_struct IV

```
    pid_t pid;
    pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif

    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->real_parent->pid)
     */
    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    /*
     * children/sibling forms the list of my natural children
     */
    struct list_head children;    /* list of my children */
    struct list_head sibling;    /* linkage in my parent's children list */
    struct task_struct *group_leader;    /* threadgroup leader */

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

Sample linux 2.6 task_struct V

```
    struct list_head ptraced;
    struct list_head ptrace_entry;

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

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

    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
#ifndef CONFIG_VIRT_CPU_ACCOUNTING
    cputime_t prev_utime, prev_stime;
#endif
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;          /* monotonic time */
    struct timespec real_start_time;    /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
    unsigned long min_flt, maj_flt;

    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];

/* process credentials */
    const struct cred __rcu *real_cred; /* objective and real subjective task
                    * credentials (COW) */
```

Sample linux 2.6 task_struct VI

```
    const struct cred __rcu *cred;    /* effective (overridable) subjective task
                     * credentials (COW) */
    struct cred *replacement_session_keyring; /* for KEYCTL_SESSION_TO_PARENT */

    char comm[TASK_COMM_LEN]; /* executable name excluding path
                     - access with [gs]et_task_comm (which lock
                       it with task_lock())
                     - initialized normally by setup_new_exec */
/* file system info */
    int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
    struct sysv_sem sysvsem;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
    unsigned long last_switch_count;
#endif
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
```

Sample linux 2.6 task_struct VII

```
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask;    /* restored if set_restore_sigmask() was used */
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;
    struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
    uid_t loginuid;
    unsigned int sessionid;
#endif
    seccomp_t seccomp;

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

#ifdef CONFIG_GENERIC_HARDIRQS
    /* IRQ handler threads */
    struct irqaction *irqaction;
```

Sample linux 2.6 task_struct VIII

```
#endif

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

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

#ifdef CONFIG_DEBUG_MUTEXES
    /* mutex deadlock detection */
    struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int irq_events;
    unsigned long hardirq_enable_ip;
    unsigned long hardirq_disable_ip;
    unsigned int hardirq_enable_event;
    unsigned int hardirq_disable_event;
    int hardirqs_enabled;
    int hardirq_context;
    unsigned long softirq_disable_ip;
    unsigned long softirq_enable_ip;
    unsigned int softirq_disable_event;
    unsigned int softirq_enable_event;
```

Sample linux 2.6 task_struct IX

```
    int softirqs_enabled;
    int softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
    u64 curr_chain_key;
    int lockdep_depth;
    unsigned int lockdep_recursion;
    struct held_lock held_locks[MAX_LOCK_DEPTH];
    gfp_t lockdep_reclaim_gfp;
#endif

/* journalling filesystem info */
    void *journal_info;

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

/* VM state */
    struct reclaim_state *reclaim_state;

    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use.  */
    struct task_io_accounting ioac;
```

Sample linux 2.6 task_struct X

```
#if defined(CONFIG_TASK_XACCT)
    u64 acct_rss_mem1;    /* accumulated rss usage */
    u64 acct_vm_mem1;    /* accumulated virtual memory usage */
    cputime_t acct_timexpd;    /* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
    nodemask_t mems_allowed;    /* Protected by alloc_lock */
    int mems_allowed_change_disable;
    int cpuset_mem_spread_rotor;
    int cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
    /* Control Group info protected by css_set_lock */
    struct css_set __rcu *cgroups;
    /* cg_list protected by css_set_lock and tsk->alloc_lock */
    struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
    struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
    struct compat_robust_list_head __user *compat_robust_list;
#endif
    struct list_head pi_state_list;
    struct futex_pi_state *pi_state_cache;
#endif
#ifdef CONFIG_PERF_EVENTS
    struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
    struct mutex perf_event_mutex;
```

Sample linux 2.6 task_struct XI

```
    struct list_head perf_event_list;
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;    /* Protected by alloc_lock */
    short il_next;
#endif
    atomic_t fs_excl;    /* holding fs exclusive resources */
    struct rcu_head rcu;

    /*
     * cache last used pipe for splice
     */
    struct pipe_inode_info *splice_pipe;
#ifdef    CONFIG_TASK_DELAY_ACCT
    struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
    int make_it_fail;
#endif
    struct prop_local_single dirties;
#ifdef CONFIG_LATENCYTOP
    int latency_record_count;
    struct latency_record latency_record[LT_SAVECOUNT];
#endif
    /*
     * time slack values; these are used to round up poll() and
     * select() etc timeout values. These are in nanoseconds.
     */
```

Sample linux 2.6 task_struct XII

```
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;

    struct list_head    *scm_work_list;
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
    /* Index of current stored address in ret_stack */
    int curr_ret_stack;
    /* Stack of return addresses for return function tracing */
    struct ftrace_ret_stack    *ret_stack;
    /* time stamp for last schedule */
    unsigned long long ftrace_timestamp;
    /*
     * Number of functions that haven't been traced
     * because of depth overrun.
     */
    atomic_t trace_overrun;
    /* Pause for the tracing */
    atomic_t tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
    /* state flags for use by tracers */
    unsigned long trace;
    /* bitmask of trace recursion */
    unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_CGROUP_MEM_RES_CTLR /* memcg uses this to do batch job */
    struct memcg_batch_info {
        int do_batch;    /* incremented when batch uncharge started */
```

Sample linux 2.6 task_struct XIII

```
        struct mem_cgroup *memcg; /* target memcg of uncharge */
        unsigned long bytes;          /* uncharged usage */
        unsigned long memsw_bytes; /* uncharged mem+swap usage */
    } memcg_batch;
#endif
};
```