

## Unix: Procesos Información en tiempo de ejecución Señales

## Procesos

- Redirección de Salida:
  - Cuando se abre un archivo, su identificador será el menor file descriptor disponible
  - Para redirigir la salida estándar a un archivo podríamos utilizar:

```
close(1); open("/tmp/output", O_WRONLY);
(Ejemplo: salida.c)
```

- Existe la rutina “dup()”, que tiene un funcionamiento similar al comando “ln”:
  - cuando se borra un archivo (rm), el contenido sigue accesible por medio del nombre asociado con ln
  - cuando el contador de enlaces del i-nodo es cero, se borrará el i-nodo y su contenido

## Procesos

- La rutina “dup” duplica un file descriptor abierto (archivo o pipe) utilizando el menor file descriptor disponible
- La siguiente instrucción redirige la salida estándar a un canal de un pipe:

```
pipe(fd); close(1); dup(fd[1]); close(fd[1]);
```

- Ejemplo: dup.c

## Información en tiempo de ejecución

- Existe una gran cantidad de información disponible para ser usada por un programa:
  - Información de los headers .h: <limits.h>, etc
  - Es “construida” en la compilación del programa
- Existe otro tipo de información, como el nombre del usuario, la cual se conoce sólo en tiempo de ejecución
- Para identificar procesos, existe el “PID”, número entero positivo y único
- Para averiguar el PID del proceso actual se utiliza la función: “getpid()”

## Información en tiempo de ejecución

- La función “getpid()” no recibe argumentos y retorna un short o long correspondiente al proceso
- Con la función “getppid()” se puede saber el PID del proceso padre (tampoco recibe argumentos)
- También es posible obtener información del usuario:
  - Cada usuario que ingresa a un sistema tiene un login name, user ID (uid), group id (gid)
  - uid y gid son manejados como números enteros positivos, los cuales se pueden convertir a nombres
  - Un usuario al menos está en un grupo

## Información en tiempo de ejecución

- Funciones para obtener información del usuario:

```
s=getlogin(); /* retorna puntero al nombre del
              usuario, o NULL */
uid=getuid(); /* retorna real UID */
uid=geteuid(); /* retorna effective UID */
gid=getgid(); /* real group ID */
gid=getegid(); /* effective group ID */
```

## Información en tiempo de ejecución

- Dado un uid es posible obtener un “record” de información asociado al usuario:

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

struct passwd *pwptr;
pwptr = getpwuid(uid);
```

- Función “getpwuid()” retorna NULL si el uid no fue encontrado

## Información en tiempo de ejecución

- Estructura “passwd”:

```
struct passwd {
    char *pw_name;           /* Username. */
    __uid_t pw_uid;         /* User ID. */
    __gid_t pw_gid;         /* Group ID. */
    char *pw_dir;           /* Home directory. */
    char *pw_shell;         /* Shell program. */
};
```

- En algunos sistemas podría tener más o menos elementos, dependiendo de la implementación

## Información en tiempo de ejecución

- En la mayoría de los sistemas la función “getpwuid()” buscará en el archivo “/etc/passwd”
- También existe la función:
  - busca por nombre y retorna un record al puntero respectivo
- Para obtener información a partir del “gid”, se utiliza la función:

```
struct group *getgrgid(gid_t gid);
```

- Esta estructura está definida en <grp.h>

## Información en tiempo de ejecución

- La estructura group:

```
struct group {
    char *gr_name; /* nombre del grupo */
    gid_t gr_gid; /* ID del grupo */
    char **gr_mem; /* miembros del grupo */
};
```

- Al igual que la estructura “passwd”, puede tener más campos dependiendo del sistema

## Información en tiempo de ejecución

- Identificación del sistema:
  - La función “uname()”, entrega una mínima información sobre el sistema en el cual se está ejecutando:

```
int uname(struct utsname *buf);
```

- La estructura “utsname” está definida en <sys/utsname.h>
- En caso de éxito, uname() retornará un valor no negativo, en caso de error retornará -1
- El formato de cada miembro de la estructura depende de la implementación del sistema

## Información en tiempo de ejecución

- Un ejemplo de estructura utsname:

```
struct utsname {
    char sysname[SYS_NMLN]; /* nombre del OS */
    char nodename[SYS_NMLN]; /* nombre en la red*/
    char release[SYS_NMLN];
    char version[SYS_NMLN];
    char machine[SYS_NMLN]; /* tipo de HW */
    char domainname[SYS_NMLN];
};
```

- Como depende de la implementación no hay manera de hacer un programa portable en su totalidad

## Información en tiempo de ejecución

- Una parte importante del “ambiente” de ejecución es la fecha hora actual
- La función “time()” retorna el número de segundos transcurridos desde 1970-01-01 a las 00:00:00

```
#include <time.h>
```

```
time_t time(time_t *t);
```

- Si “t” no es nulo, el resultado también se guarda en lo apuntado por “t”
- Generalmente es un “unsigned long”

## Información en tiempo de ejecución

- La función “time” basta para saber todo lo que se necesita del tiempo transcurrido:
  - con el número de segundos desde una fecha conocida, es posible calcular cualquier otra fecha que se desee
  - De todos modos existen las funciones “gmtime()” y “localtime()”, que convierten el número de segundos a una estructura “tm”
  - gmtime() retorna el tiempo UTC, localtime() retorna según el horario local

```
struct tm *gmtime(time_t *t);
struct tm *localtime(time_t *t);
```

## Información en tiempo de ejecución

- Estructura tm:

```
struct tm{
    int    tm_sec;    /* segundos 0..61 */
    int    tm_min;    /* minutos 0..59 */
    int    tm_hour;   /* horas 0..23 */
    int    tm_mday;   /* día del mes 1..31 */
    int    tm_mon;    /* mes 0..11 */
    int    tm_year;   /* año desde 1900 */
    int    tm_wday;   /* día de la semana 0..6 */
    int    tm_yday;   /* día del año 0..365 */
    int    tm_isdst;  /* verano/invierno */
};
```

## Información en tiempo de ejecución

- La función “mktime” permite convertir la estructura “tm” al número de segundos:

```
time_t mktime(struct tm *timeptr);
```

- mktime() ignora los valores de tm\_wday y tm\_yday
- Si los tiempos dados como entrada son inválidos, mktime los normaliza:
  - el 40 de octubre se cambiará al 9 de noviembre

## Información en tiempo de ejecución

- Para generar una fecha con formato amistoso:

```
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timep);
```

- Generan string de la forma "Wed Sep 16 21:49:08 2004\n"
- Con la función "strftime" se puede formatear fechas y horas

## Información en tiempo de ejecución

- Para medir el tiempo de ejecución de un proceso se utiliza la función:

```
clock_t clock();
```

- Indica el tiempo de procesador utilizado por el programa
- Para obtener ese número en segundos, es necesario dividirlo por la macro: CLOCKS\_PER\_SEC
- Si no se puede obtener el tiempo utilizado, retornará -1
- Ejemplo: time.c

## Señales

- Informan a un proceso la ocurrencia de algún evento:
  - Errores: división por cero, referencia inválida de memoria, instrucciones no permitidas
  - Evento Asíncrono: Finalización de un proceso
- Un proceso tiene que realizar una acción en respuesta a la señal.
- El tiempo entre que la señal se activa y el proceso la atiende se dice que la señal está pendiente.
- Un proceso puede bloquear algunas señales, mediante un "signal mask", la cual se hereda.

## Señales

- Posibilidades de proceso de una señal:
  - Terminar el proceso.
  - Ignorar la señal.
  - Detener el proceso.
  - Reanudar el proceso.
  - Atrapar la señal mediante una función del programa.

- Existen varias señales estándares:

SIGALRM	Alarma
SIGFPE	División por cero
SIGINT	^C
SIGKILL	Terminación (no se puede atrapar)
SIGPIPE	Broken Pipe
SIGTERM	Terminación

## Señales

- Para contarle al sistema cómo se debe manejar una señal, se usa:

```
signal(señal, accion);
```

- La acción puede ser una función creada por el usuario, entregando un puntero a la función
- Existen macros para acciones:

```
SIG_DFL:   Default action
SIG_IGN:   Ignore
```

## Señales

- Por ejemplo, para ignorar el Control-C (interrupt):

```
signal(SIGINT, SIG_IGN);
```

- Llamar función “errorMemoria”, cuando se detecte un referencia ilegal de memoria:

```
signal(SIGSEGV, errorMemoria);
```

- También se puede enviar una señal a si mismo con:

```
int raise (int sig);
```

## Señales

- “raise()” retornará cero en caso de éxito y no-cero en caso de error

- También se puede enviar señal a un proceso haciendo:

```
int kill(pid, señal);
```

- La señal debe ser válida y ejecutará acciones dependiendo del “pid”:
  - Positivo: Enviará la señal al proceso dueño del PID
  - Cero: Envía la señal a todos los procesos del grupo
  - -1: En la mayoría de los sistemas, envía la señal a todos los procesos

## Señales

- Para esperar una señal se utiliza:

```
pause();
```

- El programa no realiza ninguna acción hasta que llegue una señal
- El principal uso de pause es con la función sleep
  - La función “sleep()”, es una llamada a la función pause() con un tiempo
  - Si ocurre una señal, sleep retornará el número de segundos transcurridos, si no hay señal, retornará cero

## Señales

- Para programar una alarma:

```
alarm(num_segundos);
```

- Genera SIGALRM en “num\_segundos” segundos
- Si “num\_segundos” es cero, se cancelan las alarmas pendientes
- Ejemplo: lectura con timeout (timeout.c)