Sintaxis básica: tipos de datos simples

Tipo	Bytes min - hab	Rango	Formato
char	1 - 1	$[-128, 127]$ o $[0, 255]$	%c
unsigned char	1 - 1	$[0, 255]$	%uc
short (int)	2 - 2	$[-(2^{15}-1), (2^{15}-1)]$	%hd
int	2 - 2	$[-(2^{15}-1), (2^{15}-1)]$	%d
	2 - 4	$[-(2^{31}-1), (2^{31}-1)]$	
long (int)	4 - 8	$[-(2^{31}-1), (2^{31}-1)]$	%ld
long long (int)	8 - 8	$[-(2^{63}-1), (2^{63}-1)]$	%lld
float	4	$\pm$[1.2E-38, 3.4E+38]	%f
double	8	$\pm$[2.3E-308, 1.7E+308]	%lf
long double	10	$\pm$[3.4E-4932, 1.1E+4932]	%Lf

Usar siempre `sizeof(<tipo_de_dato>)` y definiciones de `limits.h` (p.ej. `INT_MAX`)

Implementadas por la biblioteca estándar de C

- Requiere la inclusión del fichero de cabecera (`strings.h`)
- Copia: strcpy, strncpy
- Concatenado: strcat, strncat
- Comparación: strcmp, strncmp
- Longitud: strlen
- Duplicado: strdup
- Creación con formato: sprintf, snprintf

API stdlib para ficheros

```
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *pathname, const char *mode,
              FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
              FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
int fflush(FILE *stream);
void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

API POSIX para directorios

En POSIX un directorio se maneja con el descriptor obtenido al abrirlo:

```
DIR* opendir(char *dirname);
struct dirent* readdir(DIR* dirp);
int closedir(DIR *dirp);
```

- El descriptor es una estructura `DIR`
- Su almacenamiento en memoria lo gestiona la biblioteca del sistema
- La función `readdir` devuelve la siguiente entrada de directorio como una estructura `struct dirent`
  - Almacenamiento gestionado por la biblioteca del sistema
  - Implementación dependiente del sistema
  - POSIX fija que debe tener al menos dos campos:
    - `d_name`: nombre del fichero/directorio
    - `d_ino`: número de nodo-i del fichero
  - Definida en el fichero `dirent.h`

Se recomienda consultar las páginas de manual de estas funciones.

- **mkdir**: crea un directorio con un nombre y protección
- **rmdir**: borra el directorio vacío con un nombre
- **rewinddir**: sitúa el puntero de posición en la primera entrada
- **chdir**: cambia el directorio actual
- **getcwd**: obtener el directorio actual
- **rename**: cambiar el nombre de una entrada del directorio

Comandos básicos

Comando	Descripción
ls	Lista los ficheros del directorio actual
pwd	Muestra en qué directorio nos encontramos
cd directory	Cambia de directorio
mv files dest	Mueve/Cambia la ruta de los ficheros
diff file1 file2	Muestra las diferencias de dos ficheros
patch file patchfile	Aplica un parche (diff) a un fichero (ver op. -
cp files dest	Copia ficheros a una nueva ruta
tail file	Muestra las últimas líneas de un fichero
head file	Muestra las primeras líneas de un fichero
mkdir dirname	Crea un nuevo directorio
rm files	Borra ficheros (elimina su nombre, unlink)
file filename	Muestra el tipo de fichero dado

Comandos básicos

Comando	Descripción
cat text_file	Muestra el contenido del archivo en pantalla
man command	Muestra la página de manual para el comando da
apropos string	Busca la cadena en la base de datos whatis
exit/logout	Abandona la sesión
grep	Busca en archivos líneas que contengan un patrón
echo	Muestra una línea de texto
env	Guarda información en el entorno
export	Cambia el valor de una variable de entorno

Ejemplo: búsqueda de fichero en ./

```
int busca(char* name)
{
  struct dirent *dp;
  DIR* dirp = opendir(".");

  if (dirp == NULL)
  return ERROR;

  while ((dp = readdir(dirp)) != NULL) {
  if (strcmp(dp->d_name, name) == 0) {
    closedir(dirp);
    return FOUND;
  }
  }
  closedir(dirp);
  return NOT_FOUND;
}
```

Señales recibidas (POSIX)

- En POSIX los procesos pueden recibir señales
  - de otros procesos: `int kill(pid_t pid, int sig)`
  - del propio SO
- El SO guarda una máscara de señales pendientes por atender para cada proceso (no encola varias señales del mismo tipo)
- Un proceso puede:
  - bloquearse a la espera de la recepción de alguna señal:
    `int pause(void)`
  - registrar un manejador para tratar la señal:
    `int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);`
- Cuando el SO cede la CPU al proceso, si tiene señales pendientes serán tratadas primero.
  - Supone una analogía software de una interrupción

Variantes de exec

Variantes:

- Formato largo (l): un puntero por argumento
- Formato vector (v): un array con los parámetros
- Con entorno (e)
- Buscando en la variable PATH (p)

Cabeceras:

- `int execl(const char *path, const char *arg0, ... , NULL)`
- `int execle(const char *path, const char *arg0, ... , NULL, char *const envp[])`
- `int execlp(const char *file, const char *arg0, ... , NULL)`
- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`
- `int execvpe(const char *file, char *const argv[] char *const envp[])`

Esperar la terminación de hijos

Un proceso puede bloquearse hasta que finalice alguno de sus hijos:

- `pid_t wait(int *status)`
  espera la finalización de cualquiera de sus procesos hijo
- `pid_t waitpid(pid_t pid, int *status, int options)`
  - `pid` determina por qué proceso se espera:
    - $<$ -1: espera a que termine cualquier proceso hijo que pertenezca al grupo de procesos indicado por el valor aboluto de `pid`
    - -1: espera a la finalización de cualquier proceso hijo
    - 0: espera a la finalización de cualquier proceso hijo que pertenezca al mismo grupo de procesos que el proceso padre
    - $>$ 0: espera por la terminación del proceso hijo con este valor de `pid`
  - `status` es un parámetro de salida que codifica el motivo de la terminación del hijo.
    - Macros: WIFEXITED, WIFSIGNALED, ...
  - `options` es una máscara de bits que permite determinar las situaciones que terminan la espera:
    - `WNOHANG`: no ha terminado ningún proceso hijo
    - `WUNTRACED`: un hijo ha sido parado
    - `WCONTINUED`: un hijo se ha reanudado

Si un proceso hijo termina, queda en un estado **zombie** hasta poder entregar el valor de retorno al proceso padre (a través **wait**)

Utilidades: procesos y trabajos

Se recomienda consultar las páginas de manual de:

- **ps**: lista los procesos activos del sistema
- **pgrep**: equivalente a **ps** | **grep**
- **top**: lista ordenada e interactiva de los procesos del sistema
- **kill**: envio de una señal a un proceso
  - o trabajo con %N donde N es el número de trabajo
- **killall**: envio de una señal a un proceso por nombre
- **pidof**: obtiene el pid de un proceso por nombre (filtra la salida ps)
- **pkill**: permite mandar una señal a un proceso por el nombre del commando ejecutado (usa grep y pidof)
- **pstree**: muestra el árbol de procesos
- **jobs**: lista trabajos activos en el shell
- **wait**: permite esperar la finalización de un proceso o un trabajo
- **nohup**: permite lanzar un proceso que ignore la señal SIGHUP

Llamada al sistema mmap

```
void* mmap(void *addr, size_t len, int prot, int
    flags, int fd, off_t offset);
```

Crea una región nueva en el mapa de memoria del proceso, dónde:

- `addr`, dirección sugerida para el mapeo
- `len`, tamaño de la región. Si no es múltiplo del tamaño de página se toma el siguiente múltiplo.
- `prot`, máscara de los bits de protección de la región (permisos de acceso): `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`
- `flags`, máscara de bits que determina el tipo de región con la combinación (OR) de los flags `MAP_PRIVATE` y `MAP_SHARED` con `MAP_ANONYMOUS`
- `fd`: el descriptor de fichero del fichero a proyectar si es un mapeo de fichero, o -1 si es un mapeo anónimo.
  - Debe abrirse siempre con permiso de lectura, y si `prot` tiene `PROT_WRITE` y `flags` tiene `MAP_SHARED` deben incluirse también permisos de escritura.
- `offset`: el offset en el fichero a partir de donde se realiza la proyección. Debe ser 0 si el flag `MAP_ANONYMOUS` está activo.

mmap tipos de regiones

En función de la combinación de los flags `MAP_SHARED` y `MAP_PRIVATE` con `MAP_ANONYMOUS` tenemos cuatro tipos de regiones:

- `MAP_PRIVATE`: mapeo de fichero privado. Se usa el contenido del fichero para inicializar la región. No sirve como mecanismo IPC. Es una región *copy-on-write*.
- `MAP_PRIVATE | MAP_ANONYMOUS`: mapeo anónimo, sirve para crear una nueva región de memoria inicializada a 0. No sirve como mecanismo IPC. Es una región *copy-on-write*.
- `MAP_SHARED`: mapeo de fichero compartido. Se usa el contenido del fichero para inicializar la memoria. Todos los procesos que tengan el mismo mapeo usan las mismas páginas físicas. El fichero se actualiza con los cambios. Sirve como mecanismo IPC, con respaldo en el sistema de ficheros.
- `MAP_SHARED | MAP_ANONYMOUS`: mapeo anónimo compartido. La región se inicializa a 0. Los procesos hijo heredan este mapeo y usan las mismas páginas físicas. Sirve como mecanismo IPC entre procesos con relación de parentesco.

POSIX: munmap y msync

```
int munmap(void *addr, size_t len);
```

- Elimina las regiones mapeadas entr addr y addr + len
- En una región compartida con respaldo en fichero deberíamos hacer primero una llamada a msync para asegurarnos que se actualiza el fichero antes de eliminar la región
- Todos los locks (mlock y mlockall) se eliminan al desmapear la región
- En exec todas las regiones son eliminadas de forma automática

```
int msync(void *addr, size_t len, int flags);
```

- Los ficheros de respaldo de las regiones MAP_SHARED los actualiza automáticamente el sistema, pero no se especifica cuándo se hará
- msync permite forzar la actualización:
  - MS_SYNC: bloqueando el hilo hasta que se complete la operación
  - MS_ASYNC: actualizando la cache de bloques, de forma que los reads al fichero obtendrán el nuevo valor, pero dejando la escritura a disco para el futuro
  - MS_INVALIDATE: se invalidan otros mapeos del mismo fichero para que se cargen de nuevo en el siguiente acceso

Uso de Variables de Condición

- Hilo que espera mientras se cumpla una condición:

```
lock(mutex);
while (<conditional expresion>)
     wait(cond_var, mutex);
<acciones restantes en sección crítica>
unlock(mutex);
<acciones deseadas fuera de sección crítica>
```

- Señalización desde otro hilo:

```
lock(mutex);
<operaciones que afectan a la expresión condicional>
signal(cond_var);
<otras operaciones protegidas>
unlock(mutex);
```

- Es decir:
  - wait: **siempre** en un bucle **while**
  - wait: **siempre** con el mutex cogido
  - signal: **mejor** con el mutex cogido

POSIX: Variables de Condición

- Inicialización de variable condicional

```
int pthread_cond_init(pthread_cond_t*cond,
    pthread_condattr_t*attr);
```

- Destrucción de variable condicional

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Operación `signal`

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Operación `broadcast`

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Operación `wait`

```
int pthread_cond_wait(pthread_cond_t*cond, pthread_mutex_t*mutex);
```

Semáforos (Dijkstra'65)

- Mecanismo de sincronización entre hilos o entre procesos
- Un contador (`s.count`) mantenido por el kernel
- Dos operaciones **atómicas**
  - **wait**: si el contador es 0 bloquea al proceso, si no decrementa el contador
  - **post**: si hay procesos bloqueados se desbloquea uno, si no se incrementa el contador

```
wait(s) { //P
  if (s.count == 0)
     //Bloquear al proceso
  else
     s.count = s.count - 1;
}
post(s) { //V
  if (hay procesos bloqueados)
     //Desbloquear a un proceso bloqueado
  else
      s.count = s.count + 1;
}
```

API Semáforos POSIX

- Semáforos anónimos
  - Inicialización (shared memoria compartida)
    `int sem_init(sem_t *sem, int shared, int val);`
  - Destrucción
    `int sem_destroy(sem_t *sem);`
- Semáforos con nombre
  - Creación o apertura de semáforo creado
    `sem_t* sem_open(char*name,int flag,mode_t mode,int val);`
  - Cierre
    `int sem_close(sem_t *sem);`
  - Borrado
    `int sem_unlink(char *name);`
- Ambos:
  - Operación `wait`
    `int sem_wait(sem_t *sem);`
  - Operación `signal`
    `int sem_post(sem_t *sem);`