



Sistemas Operativos

Dr. Léonard Janer y Dr. Pere Tuset-Peiró
Grado en Ingeniería de Telecomunicación
Estudios de Informática, Multimedia y Telecomunicación
Universitat Oberta de Catalunya

Enero de 2020

Índice general

Lista de figuras	v
Lista de tablas	xv
Lista de código	xvii
1. Introducción	1
2. Programación en C para sistemas GNU/Linux	5
2.1. Introducción	5
2.2. Instalación de <code>git</code> y obtención del repositorio con el código fuente	6
2.3. Instalación de <code>gcc</code> y compilación del programa <i>Hello world!</i>	7
2.4. Detalle del proceso de compilación de un programa con <code>gcc</code>	10
2.5. Automatización del proceso de compilación con <code>make</code>	17
3. Comunicación con <i>sockets</i>	25
3.1. Introducción	25
3.2. Comunicaciones orientadas a conexión	26
3.2.1. Desarrollo del programa cliente (satélite)	28
3.2.2. Desarrollo del programa servidor (estación base)	37
3.2.3. Pruebas de funcionamiento	44
3.3. Comunicaciones no orientadas a conexión	63
3.3.1. Desarrollo del programa cliente (satélite)	65
3.3.2. Desarrollo del programa servidor (estación base)	71
3.3.3. Pruebas de funcionamiento	74
4. Gestión de concurrencia con procesos y <i>mutex</i>	83
4.1. Introducción	83

4.2. Procesos	84
4.2.1. Servidor orientado a conexión concurrente	95
4.3. <i>Threads</i>	108
4.4. Conclusiones	115
5. Mecanismos de sincronización	117
5.1. Introducción	117
5.2. Semáforos	117
5.3. <i>Mutex</i>	176
5.4. <i>Conclusiones</i>	187
6. Mecanismos de comunicación entre procesos	193
6.1. Introducción	193
6.2. Memoria Compartida	193
6.3. Tuberías	198
6.4. Tuberías sin nombre	198
6.5. Tuberías con nombre	208
6.6. Ficheros	218
6.7. Conclusiones	230
7. Conclusiones	231

Lista de figuras

1.1. Sistema de comunicación entre un satélite y la estación base.	1
2.1. Instalación de <code>Git</code> , un sistema de control de versiones.	6
2.2. Creando directorio de referencia.	6
2.3. Clonar el repositorio <code>uoc-eimt-xx-510/ejemplos</code>	6
2.4. Contenidos del repositorio <code>ejemplos</code>	7
2.5. Instalación de <code>gcc</code>	7
2.6. Versión de <code>gcc</code>	8
2.7. Ejecución del programa de ejemplo <code>a.out</code>	9
2.8. Ejecución del programa de ejemplo <code>hello1.c</code> compilado con nombre <code>hello1</code>	9
2.9. Compilación del programa de ejemplo <code>hello2.c</code> con <code>-Wall</code>	10
2.10. Proceso de compilación de un programa en C.	11
2.11. Proceso de preprocesado de un programa en C.	11
2.12. Primeras 25 líneas del fichero resultado del preprocesado.	12
2.13. Últimas 25 líneas del fichero resultado del preprocesado.	12
2.14. Resultado en código ensamblador de nuestro ejemplo.	14
2.15. Resultado del ensamblado de nuestro ejemplo. Primeras y últimas 25 líneas.	14
2.16. Linkado de nuestro ejemplo.	15
2.17. Listado de las librerías en nuestro ejemplo.	15
2.18. Información sobre los diferentes ficheros del proceso de compilado de nuestro ejemplo.	16
2.19. Símbolos referenciados en <code>hello2.o</code>	16
2.20. Símbolos referenciados en <code>hello2</code>	17
2.21. Generación del fichero de salida en nuestro ejemplo.	18
2.22. Generación del fichero de salida en nuestro ejemplo, sin actualizar ningún componente.	18

2.23. Eliminación de los ficheros intermedios en nuestro ejemplo, para poder volver a generarlos.	18
2.24. Actualización del fichero, para volver a compilar.	19
2.25. Generación del fichero <code>hello3</code>	20
2.26. Generación del fichero <code>hello4</code>	20
2.27. Generación del fichero <code>hello5</code>	24
3.1. Sistema de comunicación entre un satélite y la estación base.	25
3.2. Ayuda del sistema <code>'man socket'</code>	31
3.3. Ayuda del sistema <code>'man socket'</code> para ver cuando usamos <code>'SOCK_STREAM'</code> como tipo	31
3.4. Campos de la estructura de tipo <code>'sockaddr_in'</code>	32
3.5. Funciones de gestión de <i>endianess</i>	34
3.6. Definición del tipo <code>'in_addr'</code>	35
3.7. Definición de la función <code>'inet_addr'</code>	35
3.8. Manual de ayuda con la definición de la función <code>'connect()'</code>	36
3.9. Manual de ayuda con la definición de la función <code>'close()'</code>	37
3.10. Manual de ayuda con la definición de la función <code>'bind()'</code>	41
3.11. Manual de ayuda con la definición de la función <code>'accept()'</code>	42
3.12. Manual de ayuda con la definición de la función <code>'listen()'</code>	43
3.13. Compilación del ejemplo <code>'tcp_client1.c'</code> con el fichero de configuración <code>'tcp_client.Makefile'</code>	44
3.14. Compilación del ejemplo <code>'tcp_server1.c'</code> con el fichero de configuración <code>'tcp_server.Makefile'</code>	44
3.15. Ejecución de <code>'tcp_server1.c'</code> y <code>'tcp_client1.c'</code> sin parámetros para que el programa nos los indique.	47
3.16. Ejecución de <code>'tcp_server1'</code> esperando peticiones de conexión al puerto 6000.	47
3.17. Escaneado de puertos TCP en nuestro sistema local con el comando <code>'nmap -sT 127.0.0.1'</code>	48
3.18. Instalación de la utilidad <code>'nmap'</code> en nuestro sistema.	48
3.19. Instalación de la utilidad <code>'netstat'</code> en nuestro sistema.	48
3.20. Comprobación que tenemos el puerto 6000 en estado <code>'LISTEN'</code> con el comando <code>'netstat'</code>	48
3.21. Comprobación del PID del proceso <code>'tcp_server1'</code> ejecutándose en nuestro sistema con el comando <code>'ps -aux'</code>	49

3.22. Podemos matar cualquier proceso del sistema con el comando 'kill'. . .	49
3.23. Ejecución del servidor en modo <i>background</i> con './tcp_server1 6000 &'. . .	49
3.24. Comprobación que nuestro servidor ejecutándose en modo <i>background</i> tiene PID=24167.	49
3.25. Comprobación que nuestro servidor con PID=24167 está a la espera de peticiones de conexión en el puerto 6000.	49
3.26. Ejecución del cliente en modo <i>background</i> con el comando './tcp_client1 127.0.0.1 hello 6000 &'. . .	49
3.27. Comprobación de la ejecución de cliente y servidor en modo <i>background</i>	50
3.28. Comprobación de las conexiones establecidas en nuestro sistema.	51
3.29. Comprobación de las direcciones asignadas en nuestro terminal.	51
3.30. Ejecución de una segunda instancia del cliente en dirección IP '192.168.122.254'. . .	51
3.31. Comprobación de las conexiones establecidas en nuestro sistema. Ahora vemos dos conexiones cliente-servidor.	51
3.32. Instalación de 'netcat' en nuestro sistema.	52
3.33. Conexión con nuestro servidor al puerto 6000 con la utilidad 'netcat'.	52
3.34. Comprobación de las conexiones establecidas en nuestro sistema. Ahora vemos tres conexiones cliente-servidor.	52
3.35. Mensaje de error de 'netcat' al intentar una petición de conexión al puerto 8000 que está cerrado.	53
3.36. Comprobación del número PID de los procesos en ejecución.	53
3.37. Eliminación de los procesos pendientes con el comando 'kill -9'.	53
3.38. Compilación de 'tcp_client2.c' y 'tcp_server2.c'.	55
3.39. Ejecución de varias peticiones de 'tcp_client2.c'.	55
3.40. Información del manual de la función 'write()'.	56
3.41. Información del manual de la función 'read()'.	56
3.42. Información del manual de la utilidad de sistema 'write'.	58
3.43. Generación de los ficheros ejecutables 'tcp_client3' y 'tcp_server3'.	58
3.44. Ejecución del servidor 'tcp_server3' escuchando en el puerto 6000 en modo <i>foreground</i>	59
3.45. Paramos la ejecución del servidor 'tcp_server3' con la combinación de teclas 'CTRL+C'.	59
3.46. Paramos la ejecución del servidor 'tcp_server3' con la combinación de teclas 'CTRL+Z' y luego pasamos el proceso a modo <i>background</i> con el comando 'bg'.	59

3.47. Volvemos a poner la ejecución del servidor 'tcp_server3' con el comando 'fg 1' a modo <i>foreground</i>	59
3.48. Control de varios procesos con los comandos 'fg' y 'bg'.	60
3.49. Ejecución del servidor 'tcp_server3' y varias instancias seguidas del cliente 'tcp_client3'.	60
3.50. Identificación del primer sistema con 'PROMPT' 'UOC1:' y dirección IP '192.122.122.203'.	62
3.51. Identificación del segundo sistema con 'PROMPT' 'UOC2:' y dirección IP '192.122.122.254'.	62
3.52. Generación del ejecutable 'tcp_server4' en el sistema con dirección IP '192.122.122.203'.	63
3.53. Generación del ejecutable 'tcp_client4' en el sistema con dirección IP '192.122.122.254'.	63
3.54. Ejecución del servidor 'tcp_server4' y respuesta de la conexión.	63
3.55. Ejecución del cliente 'tcp_client4' y respuesta de la conexión.	63
3.56. Ayuda del sistema 'man socket' para ver cuando usamos 'SOCK_DGRAM' como tipo	67
3.57. Ayuda del sistema 'man sendto' para ver los parámetros de la función	68
3.58. Ayuda del sistema 'man recvfrom' para ver los parámetros de la función.	70
3.59. Generación de los ficheros ejecutables 'udp_server1' y 'udp_client1'.	75
3.60. Ejecución de los programas 'udp_server1' y 'udp_client1' sin parámetros para saber cuáles son.	76
3.61. Ejecución del programa 'udp_server1'.	76
3.62. Comprobación del puerto UDP 6000 abierto en nuestro sistema con la utilidad 'nmap'.	78
3.63. Comprobación del puerto UDP 6000 abierto en nuestro sistema con la utilidad 'netstat'.	79
3.64. Ejecución del servidor 'udp_server1' en el sistema 1 con IP '192.168.122.140'.	79
3.65. Ejecución del cliente 'udp_client1' en el sistema 2 con IP '192.168.122.88'.	79
3.66. Mensaje recibido por el servidor 'udp_server1' donde vemos que no se imprime correctamente.	80
3.67. Mensaje recibido por el servidor 'udp_server3' donde vemos que ahora sí se imprime correctamente.	81
3.68. Estadísticas de paquetes en nuestro sistema 1 con el comando 'netstat -s'.	81
3.69. Simulación de una petición de cliente utilizando el comando 'nc -u 192.168.122.140 6000' en el sistema 2.	82

3.70. Mensaje de recibido en el servidor con la petición simulada con el comando ' <code>nc -u 192.168.122.140 6000</code> ' en el sistema 2.	82
4.1. Formato de la función <code>fork()</code>	84
4.2. Generación del ejecutable <code>fork1</code> con el comando <code>make fork1 -f fork.Makefile</code>	88
4.3. Ejemplos de ejecución del programa <code>fork1</code>	89
4.4. Ejemplos de ejecución del programa <code>fork2</code>	89
4.5. Ejemplos de ejecución del programa <code>fork3</code>	90
4.6. Eliminación del proceso en ejecución del programa <code>fork3</code>	90
4.7. Ejemplo de ejecución del programa <code>fork4</code>	91
4.8. Ejemplo de ejecución del programa <code>fork5</code>	93
4.9. Ejemplo de ejecución del programa <code>fork5</code> con 100 iteraciones sin cambio de contexto en nuestro sistema.	94
4.10. Generación de los ficheros ejecutables <code>server1</code> y <code>client1</code>	99
4.11. Ejecución del servidor <code>server1</code> y de dos instancias del cliente <code>client1</code>	100
4.12. Eliminación del servidor <code>server1</code> y de rebote de los dos procesos hijos creados que se quedaron en estado <i>zombie</i>	100
4.13. Generación de los ficheros ejecutables <code>server2</code> y <code>client2</code>	101
4.14. Ejecución del servidor <code>server2</code> y de tres instancias del cliente <code>client2</code>	103
4.15. Generación de los ficheros ejecutables <code>server3</code> y <code>client3</code>	103
4.16. Ejecución del servidor <code>server3</code> y de tres instancias del cliente <code>client3</code>	104
4.17. Generación de los ficheros ejecutables <code>server_iteration</code> y <code>client_iteration</code>	105
4.18. Ejecución del servidor <code>server_iteration</code> y dos instancias del cliente <code>client_iteration</code>	106
4.19. Introducción y eco de los dos primeros mensajes de la primera instancia del cliente <code>client_iteration</code>	107
4.20. Intercambio de mensajes por las dos instancias de cliente <code>client_iteration</code>	107
4.21. Últimos mensajes de la segunda instancia del cliente <code>client_iteration</code>	108
4.22. Generación de los ficheros ejecutables <code>server_iteration</code> y <code>client_iteration</code>	108
4.23. Manual de ayuda de la función <code>pthread_create</code> donde vemos la impor- tancia de utilizar la opción <code>-pthread</code>	109
4.24. Generación de los ejecutables <code>thread_server1</code> y <code>thread_client1</code> con el fichero de configuración <code>thread.Makefile</code>	112
4.25. Ejecución del servidor <code>thread_server1</code> y de dos instancias del cliente <code>thread_client1</code>	113

4.26. Generación de los ejecutables <code>thread_server2</code> y <code>thread_client2</code> con el fichero de configuración <code>thread.Makefile</code>	113
4.27. Ejecución del servidor <code>thread_server2</code> y de dos instancias del cliente <code>thread_client2</code>	114
4.28. Comprobación de que sólo tenemos un proceso servidor <code>thread_server2</code> en ejecución en el sistema.	114
4.29. Comprobación del número de hilos de ejecución asociados al proceso <code>thread_server2</code> en ejecución en el sistema.	115
4.30. Comprobación del número de hilos de ejecución asociados al proceso <code>thread_server2</code> en ejecución en el sistema.	115
4.31. Comprobación del número de hilos de ejecución asociados al proceso <code>thread_server2</code> en ejecución en el sistema.	115
5.1. Formato de la función ' <code>sem_open()</code> ' con la llamada al manual del sistema.	142
5.2. Formato del nombre a utilizar en el caso de semáforos con nombre de acuerdo con la llamada al manual del sistema para la función ' <code>sem_overview</code> '.142	
5.3. Generación del fichero ejecutable ' <code>sem_1a</code> '.	150
5.4. Ejemplo de ejecución del programa ' <code>./sem_1a</code> '.	150
5.5. Generación del fichero ejecutable ' <code>sem_2a</code> '.	151
5.6. Ejemplo de ejecución del programa ' <code>./sem_2a</code> '.	151
5.7. Generación del fichero ejecutable ' <code>sem_cleaning_name</code> '.	153
5.8. Ejemplo de ejecución del programa ' <code>./sem_cleaning_name</code> '.	153
5.9. Generación del fichero ejecutable ' <code>sem_3a</code> '.	154
5.10. Ejemplo de ejecución del programa ' <code>./sem_3a</code> '.	155
5.11. Manual de ayuda de la utilidad de sistema ' <code>./ipcs</code> '.	155
5.12. Ejecución de la utilidad de sistema ' <code>./ipcs</code> '.	155
5.13. Vemos el fichero creado para el semáforo ' <code>sem_1</code> ' con nombre ' <code>sem_sem_1</code> ' en el directorio del sistema ' <code>/dev/shm</code> '.	155
5.14. Ejecución del programa ' <code>sem_3a</code> ' con PID=18381.	156
5.15. Información del proceso 18381 con el comando ' <code>more /proc/18381/maps</code> '. 156	
5.16. Ejecución de la segunda instancia de ' <code>sem_3a</code> ' con PID = 18384.	156
5.17. Instalación del paquete ' <code>lsof</code> ' con el comando ' <code>apt-get install lsof</code> '. 156	
5.18. Ejecución del comando ' <code>lsof /dev/shm/sem.sem1</code> ' para ver los procesos asociados con el semáforo ' <code>sem1</code> '.	157
5.19. Instalación del paquete ' <code>ltrace</code> ' con el comando ' <code>apt-get install ltrace</code> '.157	

5.20. Monitorización de los semáforos asociados a un proceso con el comando 'pmap'.	158
5.21. Instalación del paquete 'parallel' con el comando 'apt-get install parallel'.	158
5.22. Generación de los ejecutables 'sem_4a' y 'sem_4b'.	163
5.23. Creación del semáforo 'sem_2' y puesta a valor 1 al ejecutar 'sem_4a' y 'sem_4a'.	163
5.24. Ejecución de dos instancias en paralelo de 'sem_4b' con el comando 'parallel'.	164
5.25. Ejecución de dos instancias en paralelo de 'sem_4b' ambas en modo <i>back-</i> <i>ground</i> .	164
5.26. Generación de los ejecutables para 'sem_5a', 'sem_5b' y 'sem_5c'.	173
5.27. Inicializamos el problema con el semáforo 'semaphore1' a 1, y el semáforo 'semaphore2' a 0 con el comando './sem_5a 1 0'.	174
5.28. Ejecución de 'sem_5b' y 'sem_5c' en alternancia empezando por el proceso 1.	174
5.29. Generación de los ejecutables para 'sem_6a', 'sem_6b' y 'sem_6c'.	175
5.30. Inicializamos el problema con el semáforo 'semaphore1' a 1, y el semáforo 'semaphore2' a 0 con el comando './sem_6a 1 0'.	175
5.31. Ejecución de 'sem_6b' y 'sem_6c' en alternancia empezando por el proceso 1.	176
5.32. Ejecución de 'sem_6b' y 'sem_6c' en alternancia empezando por el proceso 2.	177
5.33. Generación del ejecutable para 'mutex1'.	180
5.34. Ejecución del programa 'mutex1'.	181
5.35. Generación del ejecutable para 'mutex2'.	181
5.36. Ejecución del programa 'mutex2'.	182
5.37. Generación del ejecutable para 'mutex3'.	182
5.38. Ejecución del programa 'mutex3'.	182
5.39. Ejecución del programa 'mutex3': empezamos primer hilo de ejecución.	183
5.40. Ejecución del programa 'mutex3': conmutación al segundo hilo de ejecución.	183
5.41. Ejecución del programa 'mutex3': volvemos al primer hilo de ejecución.	184
5.42. Ejecución del programa 'mutex3': finalización del primer hilo de ejecución.	184
5.43. Generación del ejecutable para 'mutex4'.	184
5.44. Ejecución del programa 'mutex4'.	185
5.45. Generación del ejecutable para 'mutex5'.	186
5.46. Ejecución del programa 'mutex5'.	186
5.47. Generación del ejecutable para 'mutex6'.	186
5.48. Ejecución del programa 'mutex6'.	187

6.1. Generación de los ejecutables para 'shm_1a', 'shm_1b' y 'shm_1c'.	195
6.2. Inicializamos el problema con el semáforo 'semaphore1' a 1, y el semáforo 'semaphore2' a 0, y creamos la zona de memoria compartida con el comando './shm_1a 1 0'.	196
6.3. Ejecución de 'shm_1b' y 'shm_1c' en alternancia empezando por el proceso 1.	196
6.4. Ejecución del comando 'ipcs' para ver la información asociada a la zona de memoria compartida.	197
6.5. Ejecución del comando 'ipcs' para ver la información asociada a la zona de memoria compartida con un proceso asociado.	197
6.6. Límites del sistema asociados a las zonas de memoria compartida.	197
6.7. Información sobre la creación y acceso a la zona de memoria compartida.	198
6.8. Generación del ejecutable para 'pipe1'.	202
6.9. Ejecución del primer ejemplo './pipe1'.	202
6.10. Generación del ejecutable para 'pipe2'.	203
6.11. Ejecución del segundo ejemplo './pipe2' donde tenemos dos procesos que acceden a la tubería sin nombre.	203
6.12. Generación del ejecutable para 'pipe3'.	204
6.13. Ejecución del tercer ejemplo './pipe3' con el proceso padre bloqueado a la espera de recibir el número que debería haber enviado el proceso hijo.	204
6.14. Generación del ejecutable para 'pipe4'.	205
6.15. Ejecución del cuarto ejemplo './pipe4' con el proceso padre bloqueado a la espera de recibir el número que debería haber enviado el proceso hijo, e indicando que ha leído 16 caracteres en el mensaje.	205
6.16. Generación del ejecutable para 'pipe5'.	207
6.17. Ejecución del cuarto ejemplo './pipe5' de forma correcta.	207
6.18. Generación del ejecutable para 'pipe6'.	207
6.19. Ejecución del cuarto ejemplo './pipe6' de forma correcta.	208
6.20. Ayuda del libro 2 del manual de ayuda de la función 'mknod()'.	212
6.21. Generación del ejecutable para 'mknod1'.	213
6.22. Ejecución del primer ejemplo './mknod1' con los procesos padre e hijo accediendo a la tubería con nombre.	213
6.23. El fichero '/tmp/FIFO_FILE' está creado pero sin contenido.	214
6.24. Error al volver a intentar crear la tubería con nombre en el sistema. . . .	214
6.25. Generación del ejecutable para 'mknod_cleaning'.	214

6.26. Ejecución del programa <code>'./mknod_cleaning'</code> para eliminar del sistema de ficheros la tubería con nombre.	215
6.27. Generación de los ejecutables para <code>'mknod2a'</code> y <code>'mknod2b'</code>	215
6.28. Ejecución de los programas <code>'./mknod2a'</code> y <code>'./mknod2b'</code>	216
6.29. Generación de los ejecutables para <code>'mknod3a'</code> , <code>'mknod3b'</code> y <code>'mknod3c'</code> . . .	217
6.30. Ejecución de los programas <code>'./mknod3a'</code> , <code>'mknod3b'</code> y <code>'./mknod3c'</code>	218
6.31. Generación de los ejecutables para <code>'mknod4a'</code> , <code>'mknod4b'</code> y <code>'mknod4c'</code> . . .	218
6.32. Ejecución de los programas <code>'./mknod4a'</code> , <code>'mknod4b'</code> y <code>'./mknod4c'</code>	219
6.33. Generación del ejecutable para <code>'file1'</code>	222
6.34. Ejecución del programa <code>'./file1'</code>	222
6.35. Generación del ejecutable para <code>'file2'</code>	223
6.36. Ejecución del programa <code>'./file2'</code>	223
6.37. Comprobación del contenido del fichero <code>'./source_file.txt'</code> y de su ta- maño con 33 bytes.	223
6.38. Comprobación del contenido del fichero <code>'./source_file.txt'</code> y de su ta- maño con 33 bytes, con la utilidad <code>'xxd'</code>	224
6.39. Instalación de la utilidad <code>'xxd'</code> con el comando <code>'apt-get install xxd'</code> . .	224
6.40. Comprobación del contenido del fichero <code>'./source_file.txt'</code> y de su ta- maño con 33 bytes, con la utilidad <code>'hexdump'</code>	224
6.41. Instalación de la utilidad <code>'hexdump'</code> con el paquete <code>bsdmainutils</code> , con el comando <code>'apt-get install bsdmainutils'</code>	225
6.42. Generación del ejecutable para <code>'file3'</code>	225
6.43. Ejecución del programa <code>'./file3'</code>	225
6.44. comprobación que el fichero origen <code>'./source_file.txt'</code> y el fichero des- tino <code>'destination _file.txt'</code> son idénticos con el comando <code>'diff source_file.txt destina</code>	
6.45. Generación del ejecutable para <code>'file4'</code>	226
6.46. Ejecución del programa <code>'./file4'</code>	227
6.47. comprobación que el fichero origen <code>'./source_file.txt'</code> y el fichero des- tino <code>'destination _file.txt'</code> son ahora diferentes con el comando <code>'diff source_file.txt d</code>	
6.48. Generación del ejecutable para <code>'file5'</code>	229
6.49. Ejecución del programa <code>'./file5'</code>	229
6.50. comprobación que el fichero origen <code>'./source_file.txt'</code> y el fichero des- tino <code>'destination _file.txt'</code> son ahora nuevamente iguales con el co- mando <code>'diff source_file.txt destination_file.txt'</code>	229

Lista de tablas

3.1. Modelo de comunicación orientado a conexión.	28
3.2. Modelo de comunicación no orientado a conexión.	64
5.1. Problema de sincronización.	119
5.2. Problema de sincronización: Solución 1 = 22.	120
5.3. Problema de sincronización: Solución 2 = 25.	122
5.4. Problema de sincronización: Solución 3 = 28.	123
5.5. Problema de sincronización: Solución 4 = 40.	124
5.6. Problema de sincronización: Solución 5 = 34.	126
5.7. Problema de sincronización: Solución 6 = 31.	127
5.8. Problema de sincronización: Solución 7 = 14.	129
5.9. Problema de sincronización: Solución 8 = 19.	130
5.10. Problema de sincronización: Solución 9 = 16.	132
5.11. Problema de sincronización: Solución 10 = 14.	133
5.12. Problema de sincronización: Solución 11.	135
5.13. Problema de sincronización: Solución 12 = 17.	136
5.14. Posibles combinaciones con operaciones sobre la variable 'op'.	139
5.15. Problema de sincronización con un semáforo 'M1' para controlar acceso a zona crítica.	140
5.16. Posibles combinaciones con operaciones sobre la variable 'op'. controladas con semáforo 'M1'.	141
5.17. Problema de sincronización con un semáforo 'M2' para evitar la espera activa en el proceso 'C'.	141

Lista de código

1.	Código fuente del fichero <code>hello1.c</code>	8
2.	Código fuente del fichero <code>hello2.c</code>	10
3.	Código ensamblador del fichero <code>hello2.s</code>	13
4.	Fichero <code>hello2.Makefile</code>	17
5.	Código fuente del fichero <code>hello3.c</code>	19
6.	Fichero <code>hello3.Makefile</code>	20
7.	Fichero <code>hello4.Makefile</code>	21
8.	Fichero <code>hello5a.c</code>	21
9.	Fichero <code>hello5b.c</code>	22
10.	Fichero <code>hello5a.h</code>	22
11.	Fichero <code>hello5b.h</code>	22
12.	Fichero <code>hello5.Makefile</code>	23
13.	Fichero <code>'tcp_client1.c'</code>	30
14.	Definición del descriptor de <i>socket</i>	31
15.	Definición de la función <code>'err_sys()'</code>	32
16.	Inicialización del espacio de memoria con la función <code>'memset()'</code>	33
17.	Asignación del número de puerto leído como argumento de entrada con <code>'htons(atoi())'</code>	34
18.	Definición tipo de <i>socket</i>	35
19.	Petición de conexión del cliente al servidor con la llamada a la función <code>'connect()'</code>	37
20.	Código del servidor <code>'tcp_server1.c'</code>	39
21.	Creamos el descriptor de <i>socket</i> con la función <code>'socket()'</code>	39
22.	Aceptaremos peticiones de conexión desde cualquiera de las interfaces del sistema.	40
23.	Configuración tipo del <i>socket</i> en el lado servidor.	40
24.	Asociación del <i>socket</i> a nivel de sistema operativo.	41

25.	Aceptación de la petición de conexión por parte de un cliente con la función <code>'accept()'</code>	42
26.	Gestión del número de peticiones de conexión pendientes de ser atendidas en el lado servidor con la función <code>'listen()'</code>	43
27.	Fichero de configuración <code>'tcp_server.Makefile'</code> de <code>'make'</code> para todos los servidores	45
28.	Fichero de configuración <code>'tcp_client.Makefile'</code> de <code>'make'</code> para todos los servidores	46
29.	Control de los parámetros de entrada al programa <code>'tcp_client1'</code>	47
30.	Control de los parámetros de entrada al programa <code>'tcp_server1'</code>	47
31.	Bucle infinito para bloquear la finalización del cliente <code>'tcp_client1'</code>	50
32.	Nuevo programa cliente sin bucle infinito <code>'tcp_client2.c'</code>	54
33.	Nuevo programa servidor sin bucle infinito <code>'tcp_server2.c'</code>	55
34.	Nuevo programa servidor como servidor de eco <code>'tcp_server3.c'</code>	56
35.	Nuevo programa cliente como petición de eco <code>'tcp_client3.c'</code>	57
36.	Nuevo programa servidor como generación de respuesta a la petición de eco <code>'tcp_server4.c'</code>	61
37.	Nuevo programa cliente <code>'tcp_client4.c'</code> para imprimir la respuesta recibida en forma de eco por el servidor	62
38.	Programa cliente <code>'udp_client1.c'</code> la comunicación no orientada a conexión con un servidor.	67
39.	Creación del descriptor de <i>socket</i> en una comunicación no orientada a conexión tipo.	68
40.	Inicialización de la dirección del servidor en una comunicación no orientada a conexión tipo.	68
41.	Envío de información al servidor en una comunicación no orientada a conexión tipo.	69
42.	Recepción de información del servidor en una comunicación no orientada a conexión tipo.	71
43.	Impresión del mensaje devuelto por el servidor.	71
44.	Código del servidor <code>'udp_server1.c'</code>	73
45.	Código del fichero de configuración <code>'udp_server.Makefile'</code>	74
46.	Código del fichero de configuración <code>'udp_client.Makefile'</code>	75
47.	Código del servidor <code>'udp_server2.c'</code>	78
48.	Código del servidor <code>'udp_server3.c'</code>	80

49.	Fichero de configuración de <code>make fork.Makefile</code>	87
50.	Base para los ejemplos de la función <code>fork()</code> . <code>fork1.c</code>	88
51.	Código añadido en <code>fork2.c</code> para imprimir PID del proceso y de su proceso padre.	89
52.	Código añadido en <code>fork3.c</code> para bloquear la finalización del proceso. . .	89
53.	Modificaciones de código en <code>fork4.c</code> para crear un nuevo proceso hijo. .	91
54.	Modificaciones de código en <code>fork5.c</code> para crear un nuevo proceso hijo e imprimir información.	93
55.	Ejecución de <code>./fork5 2000</code> . Primer contexto proceso padre.	95
56.	Ejecución de <code>./fork5 2000</code> . Primer cambio de contexto entre el proceso padre y el proceso hijo. Y algunos cambios de contexto sucesivos.	96
57.	Ejecución de <code>./fork5 2000</code> . Último contexto de ejecución con el proceso hijo.	97
58.	Fichero de configuración <code>server.Makefile</code>	97
59.	Fichero de configuración <code>client.Makefile</code>	98
60.	Parte del código del <code>client1.c</code> donde vemos la petición y recepción de eco al servidor.	98
61.	Parte del código del <code>server1.c</code> donde vemos la gestión de la bifurcación de procesos al recibir la petición de conexión.	99
62.	Parte del código del <code>client2.c</code> donde vemos la petición y recepción de eco al servidor, sin impresión de mensajes.	101
63.	Parte del código del <code>server2.c</code> donde vemos la impresión de PID de proceso padre e hijo.	102
64.	Parte del código del <code>server3.c</code> donde vemos el bucle infinito para que el proceso hijo no finalice nunca.	104
65.	Parte del código del <code>server_iteration.c</code> donde vemos la gestión de tres mensajes por cliente en la función <code>HandleClient()</code>	105
66.	Parte del código del <code>client_iteration.c</code> donde vemos la gestión de tres mensajes.	106
67.	Parte del código del <code>thread_server1.c</code> donde vemos la creación del nuevo hilo de ejecución.	110
68.	Parte del código del <code>thread_server1.c</code> donde vemos la función <code>handleThread()</code> para atender la conexión con cada nuevo cliente.	111
69.	Fichero de configuración <code>thread.Makefile</code>	112

70.	Parte del código del <code>thread_server1.c</code> donde vemos la función <code>handleThread()</code> para atender la conexión con cada nuevo cliente con un bucle infinito que impide su finalización.	114
71.	Parte del código del <code>'sem_1a.c'</code> donde vemos la función <code>'sem_open()'</code> con el nombre del semáforo <code>'sem1'</code> que vamos a utilizar.	143
72.	Parte del código del <code>'sem_1a.c'</code> donde vemos el bucle para poner a cero el valor del semáforo al inicio del programa.	143
73.	Parte del código del <code>'sem_1a.c'</code> donde vemos el bucle para incrementar el valor del semáforo al inicio del programa.. . . .	144
74.	Fichero de configuración de <code>'make'</code> de nombre <code>'sem.Makefile'</code>	146
75.	Código del ejemplo <code>'sem_1a.c'</code>	150
76.	Parte del código del <code>'sem_2a.c'</code> donde vemos como primero ejecutamos la llamada <code>'exit()'</code> y por lo tanto nunca ejecutaremos la llamada <code>'unlink()'</code> . 151	
77.	Código del <code>'sem_cleaning_name_name.c'</code>	153
78.	Parte del código del <code>'sem_3a.c'</code> donde vemos como primero ejecutamos la llamada <code>'exit()'</code> y por lo tanto nunca ejecutaremos la llamada <code>'unlink()'</code> . 154	
79.	Código para manejar las señales de <code>'SIGSTOP'</code> en las llamadas a la función <code>'sem_wait()'</code>	157
80.	Código <code>'sem_4a.c'</code> para crear e inicializar el semáforo <code>'/sem_2'</code>	160
81.	Código <code>'sem_4b.c'</code> para las diferentes instancias de nuestro programa de sincronización.	163
82.	Código <code>'sem_5a.c'</code> para crear e inicializar los dos semáforos de nuestro problema de sincronización.	168
83.	Código <code>'sem_5b.c'</code> para la instancia del proceso 1 de nuestro problema de sincronización.	170
84.	Código <code>'sem_5c.c'</code> para la instancia del proceso 2 de nuestro problema de sincronización.	173
85.	Bucle del Código <code>'sem_6b.c'</code> para la instancia del proceso 1 de nuestro problema de sincronización.	175
86.	Bucle del Código <code>'sem_6c.c'</code> para la instancia del proceso 2 de nuestro problema de sincronización.	176
87.	Código del ejemplo <code>'mutex1.c'</code>	180
88.	Código del ejemplo <code>'mutex2.c'</code> donde vemos como paralelizamos varios hilos de ejecución.	181

89.	Código del ejemplo 'mutex3.c' donde vemos como ya no pedimos pulsación de tecla.	183
90.	Código del ejemplo 'mutex4.c' donde vemos como comparten la variable global 'value'.	185
91.	Código de la rutina para el primer hilo de ejecución en el ejemplo 'mutex5.c'.189	
92.	Código de la rutina para el segundo hilo de ejecución en el ejemplo 'mutex5.c'.190	
93.	Código de la función principal del ejemplo 'mutex5.c'.	191
94.	Código de inicialización de la variable global 'current_thread' del ejemplo 'mutex6.c'.	192
95.	Estructura para la información a guardar en la zona de memoria compartida en el ejemplo 'shm_1a.c'.	194
96.	Identificador de la zona de memoria compartida en el ejemplo 'shm_1a.c'. 194	
97.	Asignación del puntero para acceder a la zona de memoria 'shm_1a.c'. . 194	
98.	Identificador de la zona de memoria compartida en el ejemplo 'shm_1b.c' que ya se creó anteriormente.	195
99.	Acceso a la zona de memoria crítica en el ejemplo 'shm_1c.c'.	195
100.	Fichero de configuración 'pipe.Makefile'.	200
101.	Código ejemplo 'pipe1.c'.	201
102.	Código ejemplo 'pipe2.c' donde vemos la bifurcación de procesos.	202
103.	Código ejemplo 'pipe3.c' donde vemos la bifurcación de procesos y el intento de procesar a través de la tubería un texto y a continuación un número.	203
104.	Código ejemplo 'pipe4.c' donde vemos la función para imprimir el número de caracteres leídos en el mensaje por el proceso padre.	205
105.	Código ejemplo 'pipe5.c' donde vemos como la lectura y escritura de mensajes ahora se hace con primero el número de caracteres y luego el mensaje.	206
106.	Código ejemplo 'pipe6.c' donde vemos la parte del proceso hijo con una parada de 5 segundos.	208
107.	Código ejemplo 'pipe6.c' donde vemos la parte del proceso padre con una impresión de hora, para ver la parada forzada por la espera en la escritura en el proceso hijo.	209
108.	Fichero de configuración 'mknod.Makefile'.	211
109.	Código ejemplo 'mknod1.c' donde vemos la parte del proceso hijo que se encarga de escribir la información en la tubería con nombre.	212

110. Código ejemplo 'mknod1.c' donde vemos la parte del proceso padre que se encarga de leer la información de la tubería con nombre.	213
111. Código ejemplo 'mknod_cleaning.c' para eliminar del sistema de ficheros la entrada a la tubería con nombre.	214
112. Código ejemplo 'mknod2a.c' para escribir en la tubería con nombre. . . .	215
113. Código ejemplo 'mknod2b.c' para leer de la tubería con nombre.	216
114. Código ejemplo 'mknod3b.c' donde hemos eliminado la última lectura. . .	217
115. Código ejemplo 'mknod3c.c' para hacer la última lectura de la tubería con nombre.	218
116. Código ejemplo 'mknod4a.c' que queda bloqueado antes de salir.	219
117. Fichero de configuración 'file.Makefile'.	221
118. Código ejemplo 'file1.c'.	222
119. Código ejemplo 'file2.c'.	223
120. Contenido del fichero, de texto, 'source_file.txt'.	224
121. Código ejemplo 'file3.c'.	226
122. Código ejemplo 'file4.c' donde vemos que sólo leemos una línea del fichero de origen 'source_file.txt'.	227
123. Código ejemplo 'file5.c' donde vemos diversas lecturas del fichero de entrada 'source_file.txt'.	229

Capítulo 1

Introducción

Un sistema de telecomunicación tiene por objetivo poder enviar (y recibir) datos entre dos (o más) puntos distantes con el objetivo de intercambiar información que resulte útil para los usuarios de dicho sistema. Imaginemos por ejemplo un satélite en la órbita terrestre que dispone de un conjunto de sensores que captan imágenes de la superficie para estudiar el calentamiento del planeta. Para recibir comandos operativos y enviar la información de los sensores, el satélite se comunica con la estación base mediante un conjunto de tecnologías y protocolos de comunicación según la tipología de datos a intercambiar (texto, imágenes, etc.) y los requerimientos de la comunicación (ancho de banda mínimo, retraso máximo, tolerancia a la pérdida de paquetes, etc.).

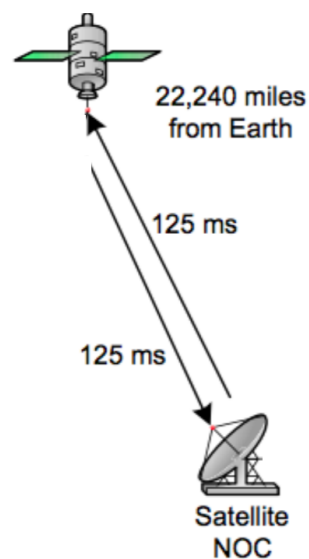


Figura 1.1: Sistema de comunicación entre un satélite y la estación base.

Por ejemplo, la telemetría del satélite requerirá del envío constante de información para el control de su estado (altitud, velocidad, orientación, etc.), requiriendo por tanto minimizar los retrasos en la comunicación (latencia entre la entrega de dos paquetes consecutivos) pero sin importar tanto el ancho de banda (se envía un número pequeño de información) ni las pérdidas (si se pierde un paquete se recibirá el siguiente con la información actualizada). En cambio, las imágenes tomadas por los distintos instrumentos del satélite no tendrán requerimientos de tiempo real pero será imprescindible garantizar un ancho de banda mínimo (en caso contrario no se podría descargar todas las fotografías que se tomen a lo largo de un día) y que todos los datos se reciben de manera correcta (si se pierde o se alteran uno o más bits la fotografía no podría abrirse correctamente o bien estaría corrompida, resultando inútil para su objetivo).

Pero más allá de las tecnologías y protocolos de comunicación utilizados para el intercambio de datos, tanto la estación base como el satélite dispondrán de uno o más sistemas de computación que se encargaran de controlar el estado del satélite y los instrumentos de medida para llevar a cabo la misión. Debido a su complejidad y a que estos sistemas pueden realizarán más de una tarea de manera simultánea, lo más probable es que estén gobernados por un sistema operativo que se encargue de abstraer y gestionar los recursos del sistema (*hardware*) y de facilitar a los programadores el uso de dichos recursos (*software*). De hecho, desde finales de los años 90 uno de los sistemas operativos que se usan para el desarrollo de satélites es GNU/Linux debido a su gran flexibilidad y el amplio soporte, tanto hardware (diferentes arquitecturas como x86, MIPS, ARM, etc.) como software (sistema de ficheros, protocolos de comunicación, etc.). Por ejemplo, GNU/Linux se ha utilizado en los proyectos FlightLinux de la NASA (*National Aeronautics and Space Administration*) y TerraSAR-X de la ESA (*European Space Agency*).

Teniendo en cuenta esto, en el siguiente documento vamos a estudiar cómo funciona la programación de un sistema basado en el sistema operativo GNU/Linux y que hoy en día se utilizan de manera ubicua en todos los ámbitos de la computación, desde satélites hasta supercomputadores, pasando por los ordenadores personales y otros dispositivos cotidianos (teléfonos móviles, televisores, etc.). En concreto, nos centraremos en aquellos aspectos de la programación que permiten la comunicación con otros sistemas, la ejecución concurrente de diferentes procesos y el almacenaje de datos en ficheros. Para ello utilizaremos el lenguaje de programación C y el conjunto de herramientas que permiten desarrollar programas escritos en este lenguaje y ejecutarlos en un dispositivo que utilice el sistema operativo GNU/Linux.

Para darle un enfoque más cercano al ámbito del Grado de Tecnologías de Telecomu-

nicación empezaremos por aprender a enviar datos entre dos dispositivos a través de un sistema de telecomunicaciones basado en los protocolos de Internet. Así pues, tendremos un programa servidor (estación base) que atiende las peticiones de un programa cliente (satélite). El funcionamiento de dicho mecanismo de comunicación puede basarse en el establecimiento de una conexión entre ambos extremos (como si fuera una llamada telefónica, que es interesante cuando hay que intercambiar mucha información, o una cantidad variable de información, y donde lo importante es no perder información) o sin tal establecimiento (cuando la comunicación es rápida, del tipo petición respuesta, y donde el parámetro más importante es la velocidad de comunicación).

A partir de este punto, nos centraremos en los mecanismos del sistema operativo que permiten ejecutar tareas de manera concurrente, tanto procesos como hilos de ejecución (*thread*), y que permiten al programador realizar diversas acciones a la vez (por ejemplo, la adquisición de datos de los instrumentos con el envío y recepción de la información a través del canal de comunicación) y también aumentar el rendimiento de una tarea concreta (por ejemplo, paralelizar el procesamiento de una imagen a través del uso simultáneo de diversas unidades de cómputo). Además, también exploramos el uso de mecanismos de sincronización, principalmente semáforos y *mutex*, y que nos permitirán generar una secuencia de ejecución que garantice el correcto funcionamiento de nuestra aplicación a pesar del paralelismo inherente introducido por los mecanismos de concurrencia.

Finalmente, también exploramos los mecanismos que permiten almacenar información de manera permanente y recuperarla posteriormente (gestión de ficheros), así como los mecanismos que permiten intercambiar información entre diferentes procesos o hilos (tuberías con nombre y sin nombre, y memoria compartida) que se ejecutan en un sistema operativo.

Nota importante: Para el correcto seguimiento de este documento se supone unos conocimientos básicos del lenguaje de programación C (funciones, variables, estructuras de control y estructuras de datos) y también unos conocimientos básicos de los comandos del sistema operativo GNU/Linux (creación, cambio y listado de directorios, así como creación, edición y eliminación de ficheros, etc.). En caso de no disponer de dichos conocimientos se recomienda seguir algún tutorial básico de programación en C y de uso del sistema operativo GNU/Linux.

Capítulo 2

Programación en C para sistemas GNU/Linux

2.1. Introducción

El primer paso para ejecutar un programa para el sistema operativo GNU/Linux es obtener el código fuente. Para ello utilizaremos `git`, un sistema de control de versiones ampliamente utilizado y que nos permitirá trabajar con el código fuente.

A continuación tendremos que compilarlo para convertirlo en un programa que sea ejecutable. Para ello utilizaremos el compilador `gcc`¹, que se encarga de generar el fichero binario que se ejecuta en el procesador a partir del código fuente.

Finalmente aprenderemos a automatizar el proceso de compilación de un programa escrito en el lenguaje de programación C para que el proceso sea reproducible. Para ello utilizaremos la herramienta `make` y sus ficheros de configuración `Makefile`, que se encargan de gestionar las dependencias entre los diferentes módulos que forman un programa y de realizar los diferentes pasos necesarios (compilación y linkado) para generar el fichero ejecutable.

¹Es importante recordar que el propio sistema operativo GNU/Linux está escrito principalmente en el lenguaje de programación C y que para ello utiliza el compilador GCC (*GNU C Compiler*). La primera versión de GCC (<http://gcc.gnu.org>) la realizó Richard Stallman en el año 1984 y actualmente se utiliza la versión 9 del compilador.

2.2. Instalación de git y obtención del repositorio con el código fuente

Antes de empezar vamos a instalar `git`. Para asegurar que `git` se encuentra instalado en el sistema hay que abrir un terminal y ejecutar el comando `sudo apt-get install git` o `apt-get install git` en función de los privilegios del usuario que lo ejecute. Después de introducir la contraseña se procederá a la instalación, o se nos indicará que `Git` ya se encuentra instalado en el sistema.

```
UOC: apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.20.1-2).
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
UOC:
UOC: █
```

Figura 2.1: Instalación de `Git`, un sistema de control de versiones.

Una vez instalado `git` en nuestro sistema podemos clonar el repositorio de ejemplos de la asignatura que se encuentra en `GitHub`.

En nuestro caso creamos una carpeta primero con el comando `mkdir /home/UOC`:

```
UOC: mkdir /home/UOC
UOC:
UOC: cd //home/UOC
UOC: pwd
//home/UOC
UOC: █
```

Figura 2.2: Creando directorio de referencia.

Para ello hay que abrir un terminal de comandos y ejecutar el comando `git clone https://github.com/uoc-eimt-xx-510/ejemplos` desde la carpeta donde nos interese clonar el repositorio (se nos pedirán credenciales de acceso a `GitHub`).

```
UOC: git clone https://github.com/uoc-eimt-xx-510/ejemplos
Cloning into 'ejemplos'...
Username for 'https://github.com': █
```

Figura 2.3: Clonar el repositorio `uoc-eimt-xx-510/ejemplos`.

Una vez clonado el repositorio podremos entrar en el directorio con el comando `cd ejemplos` y ver la estructura de directorios con el comando `ls -la`. Como vemos, la carpeta contiene los directorios `00-intro` y `01-sockets`, entre otros.

```
JOC: cd ejemplos/
JOC: ls
00-intro 01-sockets 02-concurrency 03-synchronization README.md
JOC: ls -laF | awk '{print $1,$9}'
total
drwxr-xr-x ./
drwxr-xr-x ../
drwxr-xr-x .git/
-rw-r--r-- .gitignore
drwxr-xr-x 00-intro/
drwxr-xr-x 01-sockets/
drwxr-xr-x 02-concurrency/
drwxr-xr-x 03-synchronization/
-rw-r--r-- README.md
JOC: █
```

Figura 2.4: Contenidos del repositorio ejemplos.

Podemos utilizar el comando `ls -laF|awk 'print 1,9'` para imprimir los permisos del fichero y su nombre (sin ninguna información adicional).

2.3. Instalación de gcc y compilación del programa *Hello world!*

Una vez obtenido el repositorio con el código fuente podemos empezar a trabajar. Empezaremos con el proceso de generación de un programa ejecutable a partir de un código escrito en el lenguaje de programación C utilizando el compilador `gcc`.

Para asegurar que el compilador `gcc` se encuentra instalado en el sistema hay que abrir un terminal y ejecutar el comando `apt-get install gcc`. Después de introducir la contraseña se procederá a la instalación o se nos indicará que `gcc` ya se encuentra instalado en el sistema.

```
UOC: apt-get install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
gcc is already the newest version (4:8.3.0-1).
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
UOC: █
```

Figura 2.5: Instalación de gcc.

Una vez instalado `gcc` podemos ver con qué versión trabajamos en nuestro sistema abriendo una `shell` y ejecutando el comando `gcc -version`. Como vemos, en el caso del ejemplo la versión de `gcc` instalada en nuestro entorno de trabajo es la 9.1.0.

Una vez instalado podemos empezar a utilizar `gcc` con el ejemplo `Hello World!`, que estará grabado en un fichero de texto llamado `hello1.c` en el directorio `00-intro`.

```
UOC: gcc --version
gcc (GCC) 9.1.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
UOC: █
```

Figura 2.6: Versión de gcc

Como vemos, el programa imprime por pantalla el mensaje **Hello World 1!** y finaliza su ejecución de manera ordenada (devolviendo 0).

```
1  /*
2   * Filename: hello1.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      printf("Hello world 1!\n");
10
11     return 0;
12 }
```


Código 1: Código fuente del fichero `hello1.c`

Para compilar el fichero podemos utilizar el comando `gcc hello1.c` y si el resultado es exitoso (no hay problemas en el proceso de compilación) obtendremos un fichero ejecutable llamado `a.out`. Es importante tener en cuenta que en sistemas operativos GNU/Linux la extensión del fichero (en este caso `.out`) es irrelevante para determinar si se trata de un programa u otro tipo de fichero. De hecho, la capacidad de ejecución de un fichero se basa en las propiedades del mismo, de modo que la mayoría de programas en un sistema operativo GNU/Linux no tienen extensión. En cambio, en sistemas operativos Windows la mayoría de programas tienen extensión (`.exe`).

Recordemos también que los parámetros de entrada del programa se definen con las variables `argv` y `argc` de la función `main()`, tal como se muestra en la línea 7. Por un lado `argc` será un valor entero con el número de argumentos pasados en la ejecución del programa. Por otro lado, `argv` será un vector de punteros a las cadenas de caracteres (*string*) que representan los diferentes parámetros pasados en la ejecución del programa. De este modo el parámetro `argv[0]` será siempre el nombre del programa tal y como lo

hemos introducido en la línea de comandos, mientras que de `argv[1]` a `argv[argc-1]` tendremos el resto de parámetros.


Una vez compilado satisfactoriamente, podemos ejecutar el programa escribiendo el comando `./a.out` y vemos como la salida del programa es **Hello World 1!**. Es importante tener en cuenta que se ha incluido el prefijo `./` al nombre del programa para indicar que este se encuentra en el propio directorio. Para evitarlo se podría incluir el directorio actual en la variable `PATH` de la consola, pero esto podría llevar a problemas en la búsqueda de programas del sistema y no es recomendable. Por tanto, es un buen hábito indicar siempre que vamos a ejecutar el fichero del directorio actual utilizando el prefijo `./` antes del nombre del programa.



```
UOC: gcc hello1.c
UOC: ./a.out
Hello world 1!
UOC:
UOC: █
```

Figura 2.7: Ejecución del programa de ejemplo `a.out`.

Si bien el nombre por defecto de un programa compilado con `gcc` es `a.out`, podríamos definir su nombre en el momento del compilado (es lo habitual) utilizando el comando `gcc -o hello1 hello1.c`, donde el parámetro `-o` indica el nombre del fichero de salida. Como vemos, el resultado es que nuestro programa ahora se llama `hello1`, y podemos ejecutarlo con `./hello1`.



```
UOC: gcc -o hello1 hello1.c
UOC: ./hello1
Hello world 1!
UOC:
UOC: █
```

Figura 2.8: Ejecución del programa de ejemplo `hello1.c` compilado con nombre `hello1`.

El proceso de desarrollo de un programa utilizando el lenguaje de programación C es complejo y plagado de errores propios de la condición humana. Por ejemplo, escribir mal el nombre de una variable o olvidarse un `;` al final de una línea. A la vista de esto, es un buen hábito compilar el programa de manera frecuente (para ir detectando posibles errores de manera incremental) y también activar las opciones de alerta (*warning*) del compilador. Esto último se puede hacer activando los parámetros `-Wall` o `-Werror`. El primero activa todos los mensajes de aviso de compilación, de modo que facilita la detección de errores cometidos por el programador. El segundo convierte los mensajes de aviso de compilación en errores, de modo que garantiza que el programador los subsane antes de generar el programa. A pesar de la molestia que puede suponer, el uso de `-Werror` maximiza la

probabilidad que el programa funcione correctamente una vez se encuentre en ejecución, por lo que se recomienda su utilización.

Para probar el funcionamiento de estos parámetros vamos a añadir la declaración de una variable de tipo entero que no vamos a utilizar en nuestro programa, tal como se muestra en el Código 2. Como vemos en la Figura 2.9, con la opción `-Wall` activada el compilador nos avisa que la variable `i` se encuentra declarada en la línea 9 pero no se utiliza a lo largo del programa.

```
1  /*
2   * Filename: hello2.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      int i;
10
11     printf("Hello world 2!\n");
12
13     return 0;
14 }
```

Código 2: Código fuente del fichero `hello2.c`

```
UOC: gcc -Wall -Werror -o hello2 hello2.c
hello2.c: In function 'main':
hello2.c:9:9: error: unused variable 'i' [-Werror=unused-variable]
    9 |     int i;
      |         ^
cc1: all warnings being treated as errors
UOC:
UOC: █
```

Figura 2.9: Compilación del programa de ejemplo `hello2.c` con `-Wall`.

2.4. Detalle del proceso de compilación de un programa con gcc

Ahora ya sabemos cómo compilar un programa escrito en el lenguaje de programación C utilizando el compilador `gcc` en un sistema GNU/Linux. Pero la mayoría de programas

no se componen de un único fichero, sino que están formados por diferentes módulos (unidades de compilación) que son compilados de manera independiente y que son unidas (linkadas) para formar el programa ejecutable final. Además, la mayoría de programas escritos en C cuentan con inclusiones (directivas `include`) y definiciones (directivas `define`) que deben ser resueltas antes del proceso de compilación de cada unidad de compilación. Así pues, el proceso desde que tenemos nuestro código fuente hasta que lo podemos ejecutar lo podríamos descomponer en las fases que se muestran en la Figura 2.10 y que son descritas a continuación.

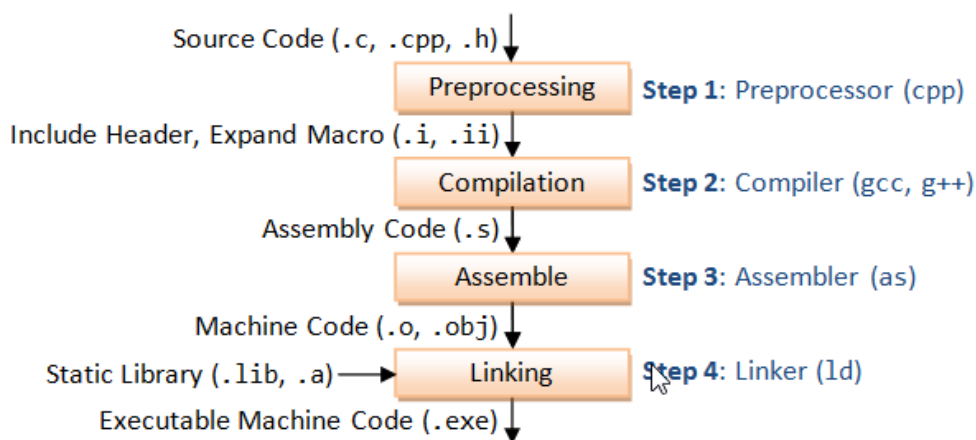


Figura 2.10: Proceso de compilación de un programa en C.

Paso 1: El preprocesador

El procesador se encarga de incorporar los ficheros de cabecera a través de la directiva `include` (ficheros con extensión `.h`), de expandir las posibles macros de nuestro código a través de la directiva `define` (por ejemplo valores constantes) y de eliminar los comentarios del usuario (ya que éstos no son interpretados por el compilador). Podemos generar el fichero intermedio generado por el preprocesador con el comando `cpp hello2.c hello2.i`, donde `hello2.i` será el fichero de salida.

```

UOC: cpp hello2.c hello2.i
UOC:
UOC:
  
```

Figura 2.11: Proceso de preprocesado de un programa en C.

Podemos ver el contenido del fichero expandido `hello2.i` con el comando `cat hello2.i`. Utilizaremos el comando `head -25 hello2.i` para mostrar sólo las primeras 25 líneas de la salida (para facilitar la lectura).

```
UOC: head -25 hello2.i
# 1 "hello2.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello2.c"

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 442 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 443 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 444 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
UOC:
UOC: █
```

Figura 2.12: Primeras 25 líneas del fichero resultado del preprocesado.

También podemos mostrar las últimas 25 líneas del mismo fichero con el comando `tail -25 hello2.i`

```
UOC: tail -25 hello2.i
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 864 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 879 "/usr/include/stdio.h" 3 4

# 6 "hello2.c" 2

# 7 "hello2.c"
int main(int argc, char* argv[])
{
    int i;

    printf("Hello world 2!\n");

    return 0;
}
UOC: █
```

Figura 2.13: Últimas 25 líneas del fichero resultado del preprocesado.

Cómo vemos, la directiva `#include <stdio.h>` ha incorporado un gran número de líneas entre las que se incluyen rutas a las librerías y definición de funciones.

```
1  .file "hello2.c"
2  .text
3  .section .rodata
4  .LC0:
5  .string "Hello world 2!"
6  .text
7  .globl main
8  .type main, @function
9  main:
10 .LFB0:
11 .cfi_startproc
12 pushq %rbp
13 .cfi_def_cfa_offset 16
14 .cfi_offset 6, -16
15 movq %rsp, %rbp
16 .cfi_def_cfa_register 6
17 subq $16, %rsp
18 movl %edi, -4(%rbp)
19 movq %rsi, -16(%rbp)
20 movl $.LC0, %edi
21 call puts
22 movl $0, %eax
23 leave
24 .cfi_def_cfa 7, 8
25 ret
26 .cfi_endproc
27 .LFE0:
28 .size main, .-main
29 .ident "GCC: (GNU) 9.1.0"
30 .section .note.GNU-stack,"",@progbits
```

Código 3: Código ensamblador del fichero `hello2.s`.

Paso 2: El compilador

A partir de la salida del preprocesador el compilador se encarga de generar un código en ensamblador para la arquitectura con la que vayamos a trabajar. Para generar el código en ensamblador a partir de la salida del preprocesador utilizamos el comando `gcc -S hello2.i`.

Cómo la salida del preprocesador no es demasiado útil para el usuario podemos realizar el proceso de compilación directamente a partir del fichero original utilizando el comando `gcc -S hello2.c`. En este caso el compilador se encargará de llamar al preprocesador

de manera transparente.

```
UOC: gcc -S hello2.i
UOC: gcc -S hello2.c
UOC:
UOC: █
```

Figura 2.14: Resultado en código ensamblador de nuestro ejemplo.

El resultado del proceso de compilación es un código en ensamblador tal como se muestra en el Código 3. Como vemos, el código de alto nivel se ha convertido en un conjunto de instrucciones en ensamblador para una arquitectura genérica.

```
UOC: hexdump hello2.o | head -10
00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000010 0001 003e 0001 0000 0000 0000 0000 0000
00000020 0000 0000 0000 0000 0290 0000 0000 0000
00000030 0000 0000 0040 0000 0000 0040 000d 000c
00000040 4855 e589 8348 10ec 7d89 48fc 7589 bff0
00000050 0000 0000 00e8 0000 b800 0000 0000 c3c9
00000060 6548 6c6c 206f 6f77 6c72 2064 2132 0000
00000070 4347 3a43 2820 4e47 2955 3920 312e 302e
00000080 0000 0000 0000 0000 0014 0000 0000 0000
00000090 7a01 0052 7801 0110 0c1b 0807 0190 0000
UOC: hexdump hello2.o | tail -10
00005400 0008 0000 0000 0000 0018 0000 0000 0000
00005500 0009 0000 0003 0000 0000 0000 0000 0000
00005600 0000 0000 0000 0000 01c8 0000 0000 0000
00005700 0014 0000 0000 0000 0000 0000 0000 0000
00005800 0001 0000 0000 0000 0000 0000 0000 0000
00005900 0011 0000 0003 0000 0000 0000 0000 0000
00005a00 0000 0000 0000 0000 0228 0000 0000 0000
00005b00 0061 0000 0000 0000 0000 0000 0000 0000
00005c00 0001 0000 0000 0000 0000 0000 0000 0000
00005d00
UOC: █
```

Figura 2.15: Resultado del ensamblado de nuestro ejemplo. Primeras y últimas 25 líneas.

Paso 3: El ensamblador

El ensamblador se encarga de convertir un código en ensamblador en lenguaje máquina de la arquitectura de nuestro procesador. Es decir, convierte el conjunto secuencial de instrucciones (y sus respectivos parámetros) en instrucciones que pueden ser ejecutadas por el procesador.

La salida del ensamblador es un fichero que ya no es editable con un editor de texto normal, pues puede contener información no imprimible por pantalla (códigos ASCII por debajo del 0x20). Por tanto, el fichero `hello2.o` debe visualizarse con un editor hexadecimal como `hexdump` como se muestra en la Figura 2.15. Para ello utilizamos el comando `hexdump hello2.o`. En caso que el comando no se encuentre en el sistema se puede instalar utilizando el comando `apt-get install bsdmainutils`.

Igual que en el caso de preprocesador, la salida intermedia no es demasiado útil para las personas, de modo que también podemos utilizar la herramienta de ensamblar. Para ello utilizamos el comando `as hello2.s -o hello2.o`.

Paso 4: El linkador

Por último, el linkador toma el código en lenguaje máquina junto con las librerías para producir el fichero ejecutable por el sistema. Para ello utilizamos el comando `ld -o hello2 hello2.o -lc -entry main`. Para no tener errores que nos impiden generar el fichero de salida tenemos que linkar nuestro programa con la librería estándar del lenguaje de programación C (parámetro `-lc`) e indicar el punto de entrada en ejecución del programa (parámetro `-entry main`).

En caso contrario el linkador nos indicará que no encuentra las referencias a las funciones utilizadas o bien que no encuentra el punto de entrada en ejecución, tal y como se muestra en la Figura 2.16.

```
UOC: ld -o hello2 hello2.o
ld: warning: cannot find entry symbol _start; defaulting to 000000000401000
ld: hello2.o: in function `main':
hello2.c:(.text+0x15): undefined reference to `puts'
UOC: ld -o hello2 hello2.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000401020
UOC: ld -o hello2 hello2.o --entry main
ld: hello2.o: in function `main':
hello2.c:(.text+0x15): undefined reference to `puts'
UOC: ld -o hello2 hello2.o -lc --entry main
UOC: █
```

Figura 2.16: Linkado de nuestro ejemplo.

Una vez linkado el programa podemos ver las librerías que se han utilizado para el linkado de nuestro ejecutable utilizando el comando `ldd hello2`.

```
UOC: ldd hello2
      linux-vdso.so.1 (0x00007ffe35d8b000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2202b6f000)
      /lib/ld64.so.1 => /lib64/ld-linux-x86-64.so.2 (0x0000564802533000)
UOC:
UOC: █
```

Figura 2.17: Listado de las librerías en nuestro ejemplo.

Cómo vemos en la Figura 2.17, en nuestro caso se ha linkado el programa con la librería `linux-vdso.so.1`, `libc.so.6` y `ld64.so.1`. La primera es una librería (dinámica) que se encarga de mapear el espacio de direcciones del programa de manera automática para aumentar el rendimiento de las llamadas al sistema. La segunda es la librería estándar del lenguaje de programación C para el sistema operativo utilizado, en nuestro caso GNU/Linux. La tercera es la librería que se encarga de realizar el linkaje dinámico

requeridos por el programa en el momento de la ejecución del mismo, de cargarlo en memoria y de ejecutarlo.

Lógicamente podemos realizar todos los cuatro pasos (preprocesado, compilado, ensamblado y linkado) de manera encadenada y transparente para el usuario utilizando el comando `gcc -o hello2 hello2.c`. Como en el caso anterior, esto generará un fichero ejecutable con el nombre `hello2`.

Una vez compilado el programa podemos utilizar herramientas del sistema operativo para analizar el tipo de los diferentes ficheros generados, como vemos en la Figura 2.18. Ejecutando el comando `file` para cada fichero podemos ver que `hello2.i` es un fichero de texto (ASCII) que contiene código fuente en C y `hello2.s` es un fichero de texto (ASCII) que contiene código fuente en ensamblador. Por su parte, `hello2.o` es un fichero objeto pendiente de linkado y `hello2` es un fichero ejecutable para la arquitectura x86-64.

```
UOC: file hello2.c
hello2.c: C source, ASCII text
UOC: file hello2.i
hello2.i: C source, ASCII text
UOC: file hello2.s
hello2.s: assembler source, ASCII text
UOC: file hello2.o
hello2.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
UOC: file hello2
hello2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib/ld64.so.1, not stripped
UOC:
UOC: █
```

Figura 2.18: Información sobre los diferentes ficheros del proceso de compilado de nuestro ejemplo.

Además, también podemos utilizar la utilidad `nm` para tener la tabla de símbolos referenciados en nuestro programa. En el caso del fichero `hello2.o`, tal y como se muestra en la Figura 2.19 vemos cómo los únicos símbolos son `main` y `puts`.

```
UOC: nm hello2.o
0000000000000000 T main
                 U puts
UOC: █
```

Figura 2.19: Símbolos referenciados en `hello2.o`.

En cambio, para el fichero ejecutable `hello2` como vemos en la Figura 2.20 la lista de símbolos y funciones se incrementa debido al proceso de linkado.

```
UOC: nm hello2
0000000000403eb0 d _DYNAMIC
0000000000404000 d _GLOBAL_OFFSET_TABLE_
0000000000404020 D __bss_start
0000000000404020 D __edata
0000000000404020 D __end
0000000000401020 T main
                U puts@@GLIBC_2.2.5
UOC: █
```

Figura 2.20: Símbolos referenciados en `hello2`.

2.5. Automatización del proceso de compilación con `make`

Como vemos, el proceso de compilación de un programa en lenguaje C es bastante tedioso y propenso a errores humanos debido a todos los pasos que hay que realizar para cada fichero que contiene código fuente. Para solucionar este problema existen multitud de herramientas encargadas de automatizar el proceso de compilación y de generar el fichero ejecutable resultante. Por ejemplo, tenemos herramientas como **CMake** y **SCons** que son multiplataforma, pero para proyectos pequeños para el sistema operativo GNU/Linux lo más habitual es utilizar la herramienta **make** y sus ficheros de configuración **Makefile**.

```
1 all: hello2
2
3 hello2: hello2.o
4 gcc -o hello2 hello2.o
5
6 hello2.o: hello2.c
7 gcc -c hello2.c
8
9 clean:
10 rm hello2.o hello2
```

Código 4: Fichero `hello2.Makefile`.

A continuación, se muestra el fichero de configuración **Makefile** para el programa `hello2`, que hemos renombrado como `hello2.Makefile`. Como vemos en el Código 4, el fichero se compone de reglas (*targets*), que tienen dependencias y un conjunto de comandos a ejecutar. Como es de esperar, para una regla concreta primero se ejecutan todas las reglas para cumplir las dependencias y luego se ejecutan los comandos asociados a la propia regla. Por ejemplo, la regla `all` depende del fichero `hello2` de modo que se

intenta resolver esta dependencia antes de seguir. La regla `hello2` depende del fichero `hello2.o` de modo que se ejecuta la regla `hello2.o` que depende del fichero `hello2.c`. Como el fichero `hello2.c` ya existe se ejecuta el comando asociado a la regla, en este caso `gcc -c hello2.c`. Este comando genera el fichero `hello2.o`, de modo que se satisface la regla `hello2` y se ejecuta el comando asociado a la regla, en este caso `gcc -o hello2 hello2.o`. Este proceso se ejecuta de manera recurrente, como vemos en la Figura 2.21, hasta que se satisfacen todas las dependencias y se ejecutan todos los comandos, garantizando que el programa se encuentra actualizado.

```
UOC: make -f hello2.Makefile
gcc -c hello2.c
gcc -o hello2 hello2.o
UOC: █
```

Figura 2.21: Generación del fichero de salida en nuestro ejemplo.

Es importante recordar que si no se especifica un *target* el programa `make` se ejecuta con el *target* `all` por defecto. Como ya compilamos anteriormente y no hemos actualizado ningún fichero, nos indica el mensaje todo está actualizado, como vemos en la Figura 2.22. Ya tenemos la última versión resultante de compilar la última versión de código.

```
UOC: make all -f hello2.Makefile
make: Nothing to be done for 'all'.
UOC: make -f hello2.Makefile
make: Nothing to be done for 'all'.
UOC:
UOC: █
```

Figura 2.22: Generación del fichero de salida en nuestro ejemplo, sin actualizar ningún componente.

Otro *target* común de un fichero `Makefile` es `clean`, que se encarga de eliminar todos los ficheros intermedios generados por el proceso de compilación. Como vemos en la Figura 2.23, el *target* `clean` no tiene dependencias y ejecuta el comando `rm hello2.o hello2`, de modo que eliminará los ficheros `hello2.o` y `hello2` que se han generado durante el proceso de compilación del programa.

```
UOC: make clean -f hello2.Makefile
rm hello2.o hello2
UOC:
UOC: █
```

Figura 2.23: Eliminación de los ficheros intermedios en nuestro ejemplo, para poder volver a generarlos.

Si ahora volvemos a ejecutar el comando `make` se volverá a ejecutar todos los comandos

descritos en el fichero `hello2.Makefile` en el orden adecuado para generar el programa `hello2`.

Es importante tener en cuenta que para decidir si un fichero se ha modificado y hay que volver a compilarlo la utilidad `make` se basa en la hora de modificación del propio fichero. Así pues, si simulamos una actualización del fichero utilizando el comando `touch` `hello2.c` y volvemos a ejecutar el comando `make` vemos cómo este vuelve a compilar y se vuelve a generar el programa `hello2`, como vemos en la Figura 2.24.

```
UOC: make -f hello2.Makefile
make: Nothing to be done for 'all'.
UOC: touch hello2.c
UOC: make -f hello2.Makefile
gcc -c hello2.c
gcc -o hello2 hello2.o
UOC: █
```

Figura 2.24: Actualización del fichero, para volver a compilar.

Ahora que ya conocemos el proceso de compilación de un programa en C y de su automatización mediante la utilidad `make`, vamos a ver un ejemplo un poco más complejo.

Retomamos nuestro ejemplo, pero le volvemos a quitar la variable declarada como vemos en el Código 5.

```
1  /*
2   * Filename: hello3.c
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char* argv[])
8  {
9      printf("Hello world 3!\n");
10
11     return 0;
12 }
```

Código 5: Código fuente del fichero `hello3.c`.

Para ello vamos a escribir un fichero de configuración `hello3.Makefile` genérico, que vemos en Código 6 que no tengamos que modificar cada ejecución en su totalidad, cambiando los nombres y las etiquetas. Como vemos, los comandos toman valores de la etiqueta (`$@`) y de los requisitos de compilación (`$<`).

```

1 all: hello3
2
3 hello2: hello3.o
4   gcc -o $@ $<
5
6 hello2.o: hello3.c
7   gcc -c $<
8
9 clean:
10  rm hello3.o hello3

```

Código 6: Fichero hello3.Makefile.

Si ejecutamos el comando `make -f hello3.Makefile` vemos en la Figura 2.25 como los diferentes ficheros se compilan y el programa se linka de manera satisfactoria.

```

UOC: make -f hello3.Makefile
cc -c -o hello3.o hello3.c
cc hello3.o -o hello3
UOC:
UOC: █

```

Figura 2.25: Generación del fichero hello3.

Pero aún podemos ir un paso más allá para hacer más genérico nuestro fichero de configuración Makefile. Para ello vamos a definir en el Código 7 unas variables de `make` que representen el nombre del programa `PROGRAM_NAME` y los ficheros objeto (`hello4.o`) necesarios para generarlo. Estas variables son sustituidas antes de empezar el proceso de compilación, de modo que se obtiene el mismo resultado que en el caso anterior.

Como vemos en la Figura 2.26 si ejecutamos el comando `make -f hello4.Makefile` observamos cómo los diferentes ficheros se compilan y el programa se linka de manera satisfactoria igual que en el caso anterior.

```

UOC: make -f hello4.Makefile
gcc -c hello4.c
gcc -o hello4 hello4.o
Finished!
UOC:
UOC: █

```

Figura 2.26: Generación del fichero hello4.

Por último vamos a separar el fichero fuente `.c` en dos ficheros llamados `hello5a.c` y `hello5b.c`. Tal como vemos en Código 8, el primer fichero contiene la función `main`,

```
1 PROGRAM_NAME = hello4
2 PROGRAM_OBJS = hello4.o
3
4 REBUIDABLES = $(PROGRAM_OBJS) $(PROGRAM_NAME)
5
6 all: $(PROGRAM_NAME)
7     @echo "Finished!"
8
9 $(PROGRAM_NAME): $(PROGRAM_OBJS)
10    gcc -o $@ $<
11
12 %.o: %.c
13    gcc -c $<
14
15 clean:
16    rm -f $(REBUIDABLES)
17    @echo "Clean done"
```

Código 7: Fichero hello4.Makefile.

mientras en el Código 9 vemos que el segundo fichero contiene la función `printer`. Además, también hemos separado los ficheros de cabecera llamados `hello5a.h` y `hello5b.h`, tal y como vemos en el Código 10 y Código 11

```
1 /*
2  * Filename: hello5a.c
3  */
4
5 #include "hello5a.h"
6 #include "hello5b.h"
7
8 int main(int argc, char* argv[])
9 {
10     return printer();
11 }
```

Código 8: Fichero hello5a.c.

Como vemos en el Código 12 tenemos la variable `PROGRAM_NAME` que representa el nombre del programa a construir y la variable `PROGRAM_OBJS` que representa los diferentes objetos que hay que generar para poder linkar el programa `hello5`. A partir de aquí se definen los diferentes *targets* de compilación. El *target* `all` depende del programa pero

```
1  /*
2   * Filename: hello5b.c
3   */
4
5  #include "hello5b.h"
6
7  int printer(void)
8  {
9      printf("Hello world 5!\n");
10     return 0;
11 }
```

Código 9: Fichero hello5b.c.

```
1  /*
2   * Filename: hello5a.h
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
```

Código 10: Fichero hello5a.h.

```
1  /*
2   * Filename: hello5b.h
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int printer(void);
```

Código 11: Fichero hello5b.h.

no realiza ninguna acción, ya que de esto se encargan los *targets* `%.o` y `$(PROGRAM_NAME)` respectivamente. Así pues, el primero se encarga de compilar todos los ficheros `.c` a objetos `.o`, mientras que el segundo se encarga de linkar todos los objetos `.o` indicados por la variable `$(PROGRAM_OBJS)` en un programa con el nombre definido por la variable `$PROGRAM_NAME`. De este modo se consigue generar el programa `hello5` a partir de la

compilación y el linkaje de los diferentes módulos que lo conforman.

```
1 PROGRAM_NAME = hello5
2 PROGRAM_OBJS = hello5b.o hello5a.o
3
4 REBUIDABLES = $(PROGRAM_OBJS) $(PROGRAM_NAME)
5
6 all: $(PROGRAM_NAME)
7     @echo "Finished!"
8
9 $(PROGRAM_NAME): $(PROGRAM_OBJS)
10    gcc -Wall -Werror -o $@ $^ -I ./
11
12 %.o: %.c
13    gcc -Wall -Werror -c $< -I ./
14
15 clean:
16    rm -f $(REBUIDABLES)
17    @echo "Clean done"
```

Código 12: Fichero `hello5.Makefile`.

También es importante indicar que el parámetro `-I ./` sirve para indicar los directorios donde el compilador va a buscar los ficheros de cabecera (`.h`) para resolver las directivas `include` durante el pre-procesado. En este caso los ficheros de cabecera están el mismo directorio, pero en caso que no fuese así habría que indicarlo de manera explícita. Esto se suele hacer mediante rutas relativas (es decir, respecto el directorio raíz del programa) para garantizar que el proceso de compilación funcione a pesar que los ficheros se encuentren en otro directorio.

Como vemos en la Figura 2.27, si ejecutamos el comando `make -f hello5.Makefile` se genera el programa `hello5` a partir de la compilación de las diferentes unidades de compilación (ficheros `hello5a.c` y `hello5b.c`) que lo componen. El resultado de la compilación es exitoso y el programa `hello5` funciona como se espera. En este nuevo fichero de configuración también hemos cambiado la opción `$<` por `$^` para que se procesen todos los ficheros de la lista.

Como vemos en los diferentes ejemplos los mensajes resultantes del comando `echo` están precedidos por un carácter `@` para evitar que se impriman dos veces.

```
UOC: make -f hello5.Makefile
gcc -Wall -Werror -c hello5b.c -I ./
gcc -Wall -Werror -c hello5a.c -I ./
gcc -Wall -Werror -o hello5 hello5b.o hello5a.o -I ./
Finished!
UOC: █
```

Figura 2.27: Generación del fichero `hello5`.

Capítulo 3

Comunicación con *sockets*

3.1. Introducción

Retomando el ejemplo de la introducción, en este primer apartado vamos a poner en marcha un entorno de comunicación entre el satélite de observación terrestre y la estación base utilizando los protocolos de Internet (IP, TCP/UDP). Estos protocolos son independientes de la tecnología de comunicación y el medio de transporte, y ofrecen funcionalidades adaptadas a los requerimientos de la aplicación, de modo que son ampliamente utilizados en diferentes entornos de comunicación.

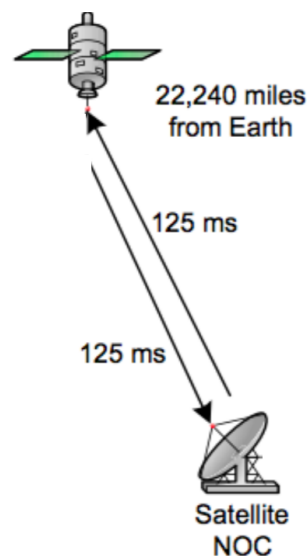


Figura 3.1: Sistema de comunicación entre un satélite y la estación base.

Por ejemplo, UDP es un protocolo de comunicación no orientado a conexión de modo

que proporciona un mecanismo de transporte ligero, es decir, incorpora mecanismos de detección de errores pero no de corrección de los mismos. En cambio, TCP es un protocolo orientado a conexión e incorpora mecanismos de recuperación de errores y de control de congestión, de modo que asegura la recepción ordenada de todos los datos y un funcionamiento justo en caso de que existan múltiples usuarios. De hecho, incluso existen adaptaciones de TCP para su utilización en la comunicación de satélites que tienen en cuenta que los retardos de propagación (2x125 milisegundos) pueden ser superiores a los retardos de transmisión.

Así pues, vamos a crear dos programas usando el lenguaje de programación C y las librerías de *sockets* que proporciona el sistema operativo GNU/Linux y que implementan los protocolos de Internet (IP, TCP/UDP). El primer programa (servidor) representará la estación base y el segundo programa (cliente) representará el satélite de observación terrestre, de modo que podremos cumplir nuestro objetivo de enviar los datos de los sensores a través de la red de comunicación.

3.2. Comunicaciones orientadas a conexión

La idea es similar a lo que pasaría si quisiéramos hablar por teléfono con alguien que conozcamos. Deberemos de conocer en primer lugar su número de teléfono; a continuación deberemos intentar establecer una llamada con el destinatario de la llamada, lo que llamaremos establecer una conexión. Es un proceso que conocemos bien. Primero intentamos conectar con el destinatario; en el momento que somos capaces de llegar al teléfono destinatario a través de la red telefónica (y siempre que por lo menos el teléfono esté conectado) entonces suena en dicho terminal un tono de llamada que en cierta forma avisa que se está intentando establecer un conexión. El receptor de la llamada debe aceptarla. Es en ese preciso momento que queda establecida la conexión entre emisor de la llamada y receptor. A partir de entonces, podemos empezar a hablar. Mientras dura la conversación sabemos que no hace falta indicar quien es el interlocutor (sólo hay dos, por lo tanto si uno habla sabe que el destinatario es el otro); la comunicación es bidireccional, los dos interlocutores pueden hablar en cualquier momento. . . . y la conexión quedará establecida hasta que uno de los dos interlocutores cuelgue el terminal y cierra la conexión.

De forma similar vamos a ver cómo funcionan las comunicaciones entre procesos/-programas a través de la red Internet, para poder establecer comunicaciones a través de mecanismos de comunicación conocidos como *sockets*, en un entorno que denominaremos orientado a conexión.

La alternativa a las comunicaciones orientadas a conexión, son las comunicaciones no orientadas a conexión, en las que no hay establecimiento de conexión. Pensemos en una comunicación del tipo *push-to-talk*. Cada vez que alguien quiere hablar, lo único que debe hacer es apretar el botón y empezar a hablar. Para que se sepa quien es el destinatario de la comunicación, siempre se indica al principio del mensaje (el emisor se informa de forma automática). Por lo tanto si apretamos el botón, y decimos “Juan:¿cómo estás?” todo el mundo entenderá que éste es un mensaje para “Juan”, y que el mensaje es “¿cómo estás?”. Al recibir el mensaje Juan, tendrá también la información del emisor (supongamos “Pedro”), y por lo tanto podrá contestar, apretando nuevamente el botón “Pedro: Yo bien, ¿y tú?”, donde se generará un mensaje destinado a “Pedro” con mensaje “Yo bien, ¿y tú?”.

De forma similar vamos a ver cómo funcionan las comunicaciones entre procesos/-programas a través de la red Internet, para poder establecer comunicaciones a través de mecanismos de comunicación conocidos como *sockets*, en un entorno que denominaremos no orientado a conexión. Por lo tanto, para las comunicaciones orientadas a conexión en un entorno Internet deberemos de fijarnos en el emisor (solemos denominarlo cliente) y en el receptor (solemos denominarlo servidor), y las funciones que cada uno debe realizar que vemos en la Tabla 3.1.

EMISOR (CLIENTE)		RECEPTOR(SERVIDOR)
3.- Emitir una petición de conexión a la dirección (IPs) del servidor, y al puerto (Ps) del servidor		1.- Informar al sistema operativo de que cuando reciba alguna llamada (a la dirección indicada (IPs), y al puerto indicado (Ps) se ponga en contacto con el proceso/programa que gestiona el servicio
		2.- Esperar una petición de conexión (IPs,Ps)
		4.- Aceptar la petición de conexión (se establece la conexión entre cliente-servidor)
Tabla 3.1 continua en la página siguiente...		

EMISOR (CLIENTE)		RECEPTOR(SERVIDOR)
	Conexión establecida entre cliente y servidor (ya pueden enviarse mensajes de forma bidireccional) (puede empezar a hablar tanto el cliente como el servidor, pues lo importante es que la conexión está establecida)	
5.- Cliente envía un mensaje al servidor		6.- Servidor recibe el mensaje del cliente
8.- Cliente recibe el mensaje de eco del servidor		7.- Servidor envía el mismo mensaje de vuelta al cliente (actuando como servidor de eco)
	Se puede continuar con la conversación en la forma que se desee	
	Ambos extremos de la comunicación (conexión) cierran la conexión (sin orden prefijado)	
8.- Cierra la conexión con el servidor		8.- Cierra la conexión con el cliente

Tabla 3.1: Modelo de comunicación orientado a conexión.

Por lo tanto, deberemos usar las funciones (llamadas al sistema) adecuadas para implementar los pasos descritos previamente.

Primero presentaremos los códigos del cliente y del servidor, para a continuación analizar cada paso.

3.2.1. Desarrollo del programa cliente (satélite)

Empecemos con el código del cliente, que tenemos en el Código 13.

```
1  /*
2   * Filename: tcp_client1.c
3   */
4
5  #include <stdio.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #define BUFFSIZE 5
14
15 void err_sys(char *mess) { perror(mess); exit(1); }
16
17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver;
19     char buffer[BUFFSIZE];
20     unsigned int echolen;
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 4) {
26         fprintf(stderr, "Usage: %s <ip_server> <word> <port>\n", argv[0]);
27         exit(1);
28     }
29
30     /* Try to create TCP socket */
31     sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
32     if (sock < 0) {
33         err_sys("Error socket");
34     }
35
36     /* Set information for sockaddr_in */
37     memset(&echoserver, 0, sizeof(echoserver)); /* reset memory */
38     echoserver.sin_family = AF_INET; /* Internet/IP */
39     echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
40     echoserver.sin_port = htons(atoi(argv[3])); /* server port */
41 }
```

```
42  /* Try to have a connection with the server */
43  result = connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver));
44  if (result < 0) {
45      err_sys("Error connect");
46  }
47
48  /* Just wait */
49  while (1);
50
51  /* Close socket */
52  close(sock);
53
54  exit(0);
55 }
```

Código 13: Fichero 'tcp_client1.c'.

Estructura con la información del socket

Lo primero que debemos entender es la información necesaria para poder comunicarse utilizando un *socket*. En el caso de las comunicaciones en Internet, la estructura que utilizamos es del tipo 'struct sockaddr_in'. Cuando vamos a trabajar con un *socket* (un conector por Internet), deberemos obtener un descriptor (de tipo 'int'), que nos va a permitir realizar posteriormente todas las operaciones necesarias de comunicación. Podemos pedir información a nuestro sistema ('man socket') como vemos en la Figura 3.2.

Como podemos ver vamos al tomo (2) de nuestro manual donde tenemos los módulos de *Linux Programmer's Manual*. Aquí encontraremos la información relevante de las funciones del sistema.

Vemos que en el momento de crear un *socket*, debemos fijarnos en la parte de la *SYNOPSIS*; en la que nos indica que necesitamos trabajar con los ficheros de cabecera `#include <sys/types.h>` y `#include <sys/socket.h>`, y que la función tiene tres parámetros de entrada (el dominio, el tipo y el protocolo), y uno de retorno (*int socket(int domain, int type, int protocol);*) tal y como vemos en el Código 14.

El descriptor de *socket* será un índice a un espacio en el que se almacenará toda la información importante del mismo para su correcto funcionamiento y operación.

Analicemos los parámetros de entrada:

- 'Domain': Nos indicará el tipo de "entorno" de comunicación que vamos a utilizar. En nuestro caso, vamos a utilizar un entorno Internet, con direcciones IP.

```

UOC: man socket|head -20
SOCKET(2)                                Linux Programmer's Manual          SOCKET(2)

NAME
    socket - create an endpoint for communication

SYNOPSIS
    #include <sys/types.h>           /* See NOTES */
    #include <sys/socket.h>

    int socket(int domain, int type, int protocol);

DESCRIPTION
    socket() creates an endpoint for communication and returns a file descriptor that
    refers to that endpoint. The file descriptor returned by a successful call will be
    the lowest-numbered file descriptor not currently open for the process.

    The domain argument specifies a communication domain; this selects the protocol fam-
    ily which will be used for communication. These families are defined in
    <sys/socket.h>. The currently understood formats include:

UOC: █

```

Figura 3.2: Ayuda del sistema 'man socket'.

```

30  /* Try to create TCP socket */
31  sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

```

Código 14: Definición del descriptor de *socket*.

Aquí como vemos en el código utilizamos la opción 'PF_INET'. que nos indicará 'Protocol_Family_INET (InterNET)'.

- 'Type': Para trabajar en entornos orientados a conexión utilizaremos 'SOCK_STREAM', como vemos en la Figura 3.3 donde capturamos un trozo del manual de ayuda, para resaltar la definición del tipo.
- 'Protocol': Donde definiremos el protocolo de transporte TCP, para las comunicaciones orientadas a conexión.

```

UOC: man socket|head -38|tail -2
    SOCK_STREAM    Provides sequenced, reliable, two-way, connection-based byte streams.
                   An out-of-band data transmission mechanism may be supported.
UOC: █

```

Figura 3.3: Ayuda del sistema 'man socket' para ver cuando usamos 'SOCK_STREAM' como tipo

Por lo tanto, en el momento de la creación del descriptor del *socket* con el que vamos a trabajar, deberemos fijar siempre los mismos valores para las comunicaciones orientadas a conexión en entornos internet, como en nuestro ejemplo:

```
'sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);'
```

La llamada al sistema, nos dará un número mayor que cero en el caso de haber podido crear el descriptor; en caso contrario, podremos ver el error si nos interesara. En nuestro caso utilizamos la función `'err_sys(mess)'` tal y como vemos en el Código 15 para imprimir un mensaje identificativo, que luego hace una llamada a `'perror(mess)'` que nos va a indicar la causa del error, y además nos va a finalizar el programa (utilizando la llamada `'exit(1)'`).

```
15 void err_sys(char *mess) { perror(mess); exit(1); }
```

Código 15: Definición de la función `'err_sys()'`.

Una vez hemos creado de forma exitosa el descriptor de nuestro *socket*, deberemos inicializar los valores adecuados para poder configurar las direcciones del mismo. Para empezar, en el lado cliente, deberemos configurar la dirección del servidor con el que deseamos conectarnos, para poder llevar a cabo la petición de conexión. Para ello utilizaremos la estructura para la definición de las direcciones `'struct sockaddr_in echoserver;'`.

En nuestro caso, al tratarse de un entorno internet utilizaremos la declaración específica al tipo `'sockaddr_in'`, pero en cualquier caso las funciones están definidas con el tipo genérico `'sockaddr'` para que puedan utilizarse para cualquier situación: El tipo genérico define un tamaño de la dirección, una familia, y los valores en una cadena de 14 bytes. En el caso específico de las direcciones en entornos Internet, como vemos en la Figura 3.4 las direcciones deberán tener un puerto identificativo y una dirección IP. En el caso de IPv4 el puerto son dos bytes, y la dirección IP cuatro, con lo que nos quedan 8 bytes libres (que se recomienda siempre poner a cero).

```
UOC: cat /usr/include/linux/in.h |tail -73|head -12
/* Structure describing an Internet (IP) socket address. */
#if __UAPI_DEF_SOCKADDR_IN
#define __SOCK_SIZE__ 16          /* sizeof(struct sockaddr) */
struct sockaddr_in {
    __kernel_sa_family_t  sin_family; /* Address family */
    __be16                sin_port;   /* Port number */
    struct in_addr         sin_addr;   /* Internet address */

    /* Pad to size of `struct sockaddr'. */
    unsigned char          __pad[__SOCK_SIZE__ - sizeof(short int) -
                                sizeof(unsigned short int) - sizeof(struct in_addr)];
};
UOC: █
```

Figura 3.4: Campos de la estructura de tipo `'sockaddr_in'`.

Antes de declarar los valores adecuados de la estructura se recomienda siempre primero inicializar toda la zona de memoria a cero, para ello utilizaremos la función `'memset'`, donde tendremos la dirección de memoria desde la que vamos a poner ceros, el número de ceros. Fijaros cómo `'&echoserver'` indica la dirección de memoria donde tenemos la variable `'echoserver'`, `'0'` indica el valor con el que vamos a inicializar todas las posiciones de memoria y `'sizeof(echoserver)'` indica el número de valores que vamos a poner (en este caso, nos indicará el tamaño de la variable `'echoserver'`) tal y como vemos en el Código 16

```
37      memset(&echoserver, 0, sizeof(echoserver));      /* reset memory */
```

Código 16: Inicialización del espacio de memoria con la función `'memset()'`.

Una vez inicializada la variable `'echoserver'` el primero de los valores que vamos a configurar es la familia de direcciones, en nuestro caso siempre pondremos `'AF_INET'` (`'echoserver.sin_family = AF_INET;'`).

El segundo parámetro será el número del puerto, en este caso, en el que el servidor estará preparado para recibir la petición de conexión por parte del cliente. Lo que deberemos introducir en este caso es un valor de tipo entero, con el número de puerto.

Algunas consideraciones al respecto importantes:

- Los valores enteros deben tener en cuenta, siempre, en el caso de las funciones asociadas con *socket* el orden de los bytes. Lo que conocemos como `'little endian'`, o `'big endian'`. Para estar seguros que siempre se ponen los valores en el orden correcto, se definen cuatro funciones para convertir los formatos. No es necesario saber el orden con el que trabaja nuestro sistema, ni el orden del formato de las comunicaciones en Internet. Sencillamente hay que utilizar sistemáticamente las funciones, para garantizar que se hagan las conversaciones en caso de necesidad. Las cuatro funciones las tenemos en la Figura 3.5. Cada vez que tengamos un valor entero (en el lado máquina) que debemos utilizar en el lado *socket*, deberemos usar la función de conversión *Host-to-Network*, para `'short'` (2 bytes) (`'htons()'`) o para `'longs'` (`'htonl()'`). de forma similar siempre que recibamos un valor a través del *socket* que debemos utilizar en el sistema, deberemos usar las funciones *Network-to-Host*, para `'short'` (2 bytes) (`'ntohs()'`) o para `'long'` (4 bytes) (`'ntohl()'`).

- Si el valor del puerto lo obtenemos como parámetro de entrada del programa (a través de los argumentos de entrada), como éstos son cadenas de caracteres deberemos convertir la cadena en un valor entero. La forma más fácil de hacerlo es con la función `'atoi()'`, como en nuestro Código 17.

```
UOC: man byteorder|head -28
BYTEORDER(3)                                Linux Programmer's Manual                                BYTEORDER(3)

NAME
    htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS
    #include <arpa/inet.h>

    uint32_t htonl(uint32_t hostlong);

    uint16_t htons(uint16_t hostshort);

    uint32_t ntohl(uint32_t netlong);

    uint16_t ntohs(uint16_t netshort);

DESCRIPTION
    The htonl() function converts the unsigned integer hostlong from host byte order to network byte order.

    The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.

    The ntohl() function converts the unsigned integer netlong from network byte order to host byte order.

    The ntohs() function converts the unsigned short integer netshort from network byte order to host byte order.

UOC: █
```

Figura 3.5: Funciones de gestión de *endianess*.

40

```
echoserver.sin_port = htons(atoi(argv[3]));    /* server port */
```

Código 17: Asignación del número de puerto leído como argumento de entrada con `'htons(atoi())'`.

Siguiendo con los parámetros de configuración del *socket*, el siguiente parámetro será la dirección IP, en este caso del servidor con el que nos deseemos conectar, en formato `'long'`. Para ello se trabaja con la estructura `'sin_addr'`, que tendrá el formato adecuado. En el caso de las comunicaciones internet, tiene un único campo `'s_addr'`, tal y como vemos en la Figura 3.6, para la dirección IP (4 bytes) en formato `'long int'`.

Normalmente las direcciones IP las tendremos en formato cadena de caracteres, del tipo `'A.B.C.D'`, y por lo tanto deberemos convertirlas de tipo. Para ello utilizaremos la


```
UOC: cat /usr/include/linux/in.h|head -87|tail -4
/* Internet address. */
struct in_addr {
    __be32 s_addr;
};
UOC: █
```

Figura 3.6: Definición del tipo 'in_addr'.

función 'inet_addr', que convierte de cadena, a entero, en formato *network*, con lo que ya no habrá que hacer uso de la función 'htonl()', tal y como explica en la Figura 3.7.

```
UOC: man inet_addr|head -70|tail -6
The inet_addr() function converts the Internet host address cp from IPv4 numbers-and-
dots notation into binary data in network byte order. If the input is invalid, IN-
ADDR_NONE (usually -1) is returned. Use of this function is problematic because -1
is a valid address (255.255.255.255). Avoid its use in favor of inet_aton(),
inet_pton(3), or getaddrinfo(3), which provide a cleaner way to indicate error re-
turn.
UOC: █
```

Figura 3.7: Definición de la función 'inet_addr'.

Por lo tanto, para inicializar la dirección del servidor, primero a ceros, y después a los valores que nos interesa de puerto y dirección el código siempre será del tipo representado en el Código 18.

```
36  /* Set information for sockaddr_in */
37  memset(&echoserver, 0, sizeof(echoserver));          /* reset memory */
38  echoserver.sin_family = AF_INET;                     /* Internet/IP */
39  echoserver.sin_addr.s_addr = inet_addr(argv[1]);     /* IP address */
40  echoserver.sin_port = htons(atoi(argv[3]));        /* server port */
```

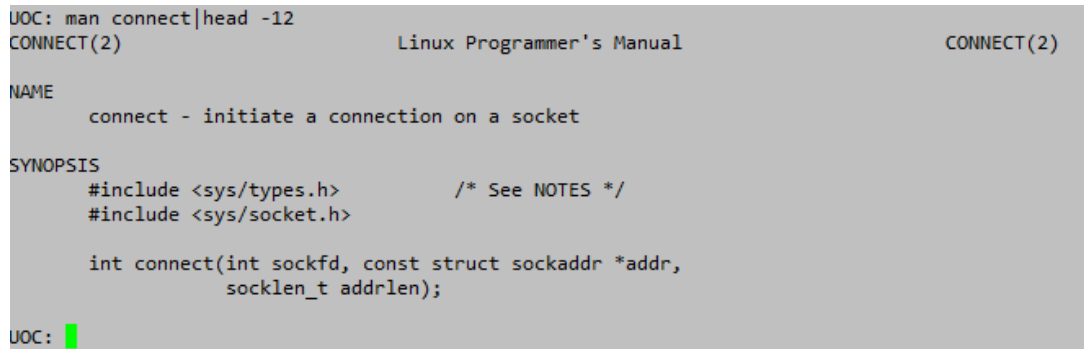
Código 18: Definición tipo de *socket*.

Un *socket*, es una estructura del sistema operativo que tiene tres parámetros de configuración que deben ser conocidos en ambos extremos de la comunicación. Hasta que dichos parámetros no estén todos correctamente asignados o configurados el *socket* no es operativo. Los tres parámetros de un *socket* son:

- El protocolo (de comunicación).
- La dirección de uno de los extremos del *socket*, con todos sus parámetros.
- La dirección del otro extremo del *socket*, con todos sus parámetros.

Cuando declaramos el *socket*, en el lado cliente, definimos el protocolo en el lado cliente.

Para establecer la petición de conexión con el servidor, debemos configurar los parámetros de dirección (dirección, puerto, familia) del servidor, y hacer una llamada a la función '`connect()`' que tenemos definida en la Figura 3.8.



```

UOC: man connect|head -12
CONNECT(2)                                Linux Programmer's Manual                                CONNECT(2)

NAME
    connect - initiate a connection on a socket

SYNOPSIS
    #include <sys/types.h>                /* See NOTES */
    #include <sys/socket.h>

    int connect(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);

UOC:
  
```

Figura 3.8: Manual de ayuda con la definición de la función '`connect()`'.

Los tres parámetros de esta función son:

- El descriptor del *socket* que hemos inicializado.
- La dirección de memoria donde tenemos almacenada la información de la dirección remota (en nuestro caso el servidor) que hemos inicializado previamente.
- El tamaño de la dirección, que en un entorno internet, siempre es fijo y viene dado por la macro '`sizeof()`'.

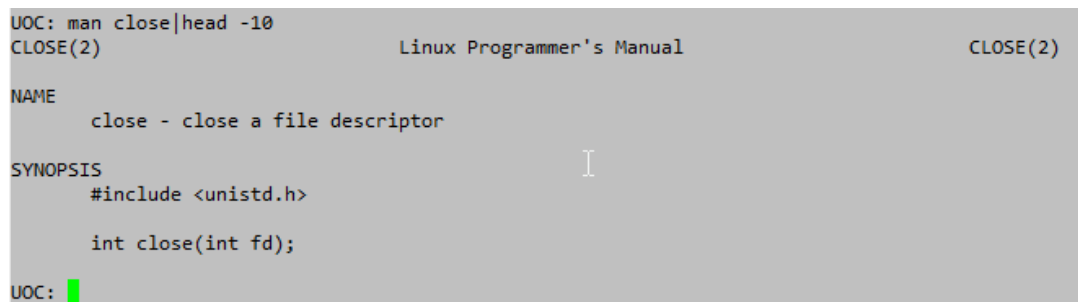
Los parámetros del lado cliente, se toman de forma automática por parte del sistema operativo. La dirección IP es una de las posibles en nuestro sistema (alguna de las existentes), y el número de puerto, en toda comunicación el lado cliente lo elige de forma pseudoaleatoria el sistema operativo de entre los números disponibles.

Cuando la llamada a '`connect()`' retorne un valor correcto, como vemos en el Código 19, en nuestro cliente ya tendremos todos los parámetros de configuración del *socket*, y por lo tanto, ya lo podremos utilizar.

A partir de este punto, el cliente ya tiene la conexión establecida con el servidor, y por lo tanto va a poder enviar y recibir información, a través de un canal bidireccional. El funcionamiento de la comunicación lo veremos posteriormente. Ahora sólo nos quedará en el momento que ya no queramos mantener el *socket*, cerrarlo con una llamada a la función '`close()`' que tenemos definida en la Figura 3.9.

```
42  /* Try to have a connection with the server */
43  result = connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver));
44  if (result < 0) {
45      err_sys("Error connect");
46  }
```

Código 19: Petición de conexión del cliente al servidor con la llamada a la función 'connect()'.



The screenshot shows a terminal window with the command 'UOC: man close | head -10' executed. The output displays the first ten lines of the manual page for 'close(2)'. The header 'Linux Programmer's Manual' is centered, and 'CLOSE(2)' is on the right. The 'NAME' section states 'close - close a file descriptor'. The 'SYNOPSIS' section shows the header file '#include <unistd.h>' and the function signature 'int close(int fd);'. The prompt 'UOC:' is visible at the bottom left.

```
UOC: man close | head -10
CLOSE(2)                                Linux Programmer's Manual                                CLOSE(2)

NAME
    close - close a file descriptor

SYNOPSIS
    #include <unistd.h>

    int close(int fd);

UOC: █
```

Figura 3.9: Manual de ayuda con la definición de la función 'close()'.

3.2.2. Desarrollo del programa servidor (estación base)

Pasaremos ahora a analizar el código del servidor que tenemos en el Código 20.

```
1  /*
2   * Filename: tcp_server1.c
3   */
4
5  #include <stdio.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #define MAXPENDING 5    /* Maximum number of simultaneous connections */
14 #define BUFFSIZE 5     /* Size of message to be received */
15
16 void err_sys(char *mess) { perror(mess); exit(1); }
17
18 void handle_client(int sock) {
```

```
19     char buffer[BUFSIZE];
20     int received = -1;
21
22     /* Just wait */
23     while (1);
24
25     /* Close socket */
26     close(sock);
27 }
28
29 int main(int argc, char *argv[]) {
30     struct sockaddr_in echoserver, echoclient;
31     int serversock, clientsock;
32     int result;
33
34     /* Check input arguments */
35     if (argc != 2) {
36         fprintf(stderr, "Usage: %s <port>\n", argv[0]);
37         exit(1);
38     }
39
40     /* Create TCP socket */
41     serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
42     if (serversock < 0) {
43         err_sys("Error socket");
44     }
45
46     /* Set information for sockaddr_in structure */
47     memset(&echoserver, 0, sizeof(echoserver));          /* we reset memory */
48     echoserver.sin_family = AF_INET;                     /* Internet/IP */
49     echoserver.sin_addr.s_addr = htonl(INADDR_ANY);      /* ANY address */
50     echoserver.sin_port = htons(atoi(argv[1]));         /* server port */
51
52     /* Bind socket */
53     result = bind(serversock, (struct sockaddr *) &echoserver, sizeof(echoserver));
54     if (result < 0) {
55         err_sys("Error bind");
56     }
57
58     /* Listen socket */
59     result = listen(serversock, MAXPENDING);
60     if (result < 0) {
```

```
61     err_sys("Error listen");
62 }
63
64 /* loop */
65 while (1) {
66     unsigned int clientlen = sizeof(echoclient);
67
68     /* Wait for a connection from a client */
69     clientsock = accept(serversock, (struct sockaddr *) &echoclient, &clientlen);
70     if (clientsock < 0) {
71         err_sys("Error accept");
72     }
73     fprintf(stdout, "Client: %s\n", inet_ntoa(echoclient.sin_addr));
74
75     /* Call function to handle socket */
76     handle_client(clientsock);
77 }
78 }
```

Código 20: Código del servidor 'tcp_server1.c'.

En el lado servidor, aprovecharemos muchos de los comentarios del lado cliente. En primer lugar declaramos un descriptor de *socket* para su uso en la inicialización del canal de comunicación entre cliente y servidor. Utilizaremos las mismas opciones, para declarar un *socket* orientado a conexión en un entorno internet como vemos en el Código 21.

```
40 /* Create TCP socket */
41 serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
42 if (serversock < 0) {
43     err_sys("Error socket");
44 }
```

Código 21: Creamos el descriptor de *socket* con la función 'socket()'.

A continuación hemos de inicializar los parámetros del *socket* en el lado servidor, con el número de puerto en el que esperaremos las peticiones de conexión, y la dirección IP en la que esperaremos las mismas.

Para el valor del puerto utilizaremos el mismo procedimiento que en el lado cliente.

Siempre es conveniente inicializar primero toda la estructura a cero, y luego rellenar los valores adecuados.

Para la dirección IP. Debemos tener algunas consideraciones. Es importante entender que un sistema (una máquina) puede tener más de una interfaz con el mundo “exterior”. Para cada una de ellas puede/debe tener una dirección IP diferente (sin la configuración de la dirección IP la interfaz no es operativa a nivel de *socket*). Si nuestro sistema tiene más de una interfaz, podemos configurar el *socket*, para que pueda aceptar las peticiones que le lleguen de cualquier de las posibles conexiones (puertas, interfaces). Es el caso de nuestro ejemplo. En estas situaciones cuando configuramos la dirección IP del *socket*, configuramos para que funcione con cualquier interfaz, como vemos en el Código 22.

```
49 echoserver.sin_addr.s_addr = htonl(INADDR_ANY);    /* ANY address */
```

Código 22: Aceptaremos peticiones de conexión desde cualquiera de las interfaces del sistema.

Si por contra quisiéramos que sólo funcionase con una dirección IP concreta, configuraríamos ésta con `echoserver.sin_addr.s_addr = inet_addr("192.168.0.1");` como hicimos en el lado cliente. (si sólo queremos que responda a peticiones a la dirección '192.168.0.1').

La configuración depende de la especificación que deseemos tener del servidor.

Por lo tanto, la configuración en el lado servidor, de los parámetros del *socket* serán siempre similares a las del Código 23.

```
46 /* Set information for sockaddr_in structure */
47 memset(&echoserver, 0, sizeof(echoserver));    /* we reset memory */
48 echoserver.sin_family = AF_INET;                /* Internet/IP */
49 echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /* ANY address */
50 echoserver.sin_port = htons(atoi(argv[1]));    /* server port */
```

Código 23: Configuración tipo *delsocket* en el lado servidor.

Como hemos dicho un *socket*, es una estructura del sistema operativo que tiene tres parámetros de configuración y que deben ser conocidos en ambos extremos de la comunicación. Hasta que dichos parámetros no estén todos correctamente asignados o configurados el *socket* no es operativo. Los tres parámetros de un *socket* son:

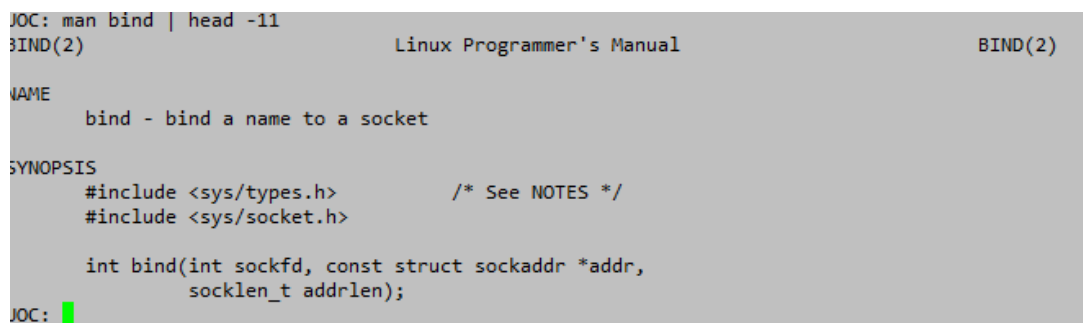
- El protocolo (de comunicación).
- La dirección de uno de los extremos del *socket*, con todos sus parámetros.
- La dirección del otro extremo del *socket*, con todos sus parámetros.

Cuando declaramos el *socket*, en el lado servidor, definimos el protocolo en el lado servidor.

La primera de las configuraciones necesarias en el *socket*, es en el lado servidor, para indicarle al sistema operativo que esté atento a cualquier intento de conexión a la dirección (direcciones) IP que nos convenga y al número de puerto del servicio. Para ello, primero inicializamos los parámetros del *socket* en el lado servidor, y llevamos a cabo una llamada a la función 'bind()', para asociar estos parámetros con el proceso que atenderá el servidor, como vemos en el Código 24.

```
52  /* Bind socket */
53  result = bind(serversock, (struct sockaddr *) &echoserver, sizeof(echoserver));
54  if (result < 0) {
55      err_sys("Error bind");
56  }
```

Código 24: Asociación del *socket* a nivel de sistema operativo.



The image shows a terminal window displaying the manual page for the `bind(2)` system call. The title bar indicates it's from the 'Linux Programmer's Manual'. The content includes the command 'bind - bind a name to a socket', the synopsis with headers and the function signature, and the start of the 'SEE ALSO' section.

```
JOC: man bind | head -11
BIND(2)                                Linux Programmer's Manual                                BIND(2)

NAME
    bind - bind a name to a socket

SYNOPSIS
    #include <sys/types.h>           /* See NOTES */
    #include <sys/socket.h>

    int bind(int sockfd, const struct sockaddr *addr,
              socklen_t addrlen);

JOC: █
```

Figura 3.10: Manual de ayuda con la definición de la función 'bind()'.

Esta función tiene tres parámetros, tal y como vemos en la Figura 3.10:

- El descriptor de *sockets* que hemos creado con la llamada a la función 'socket()'.
- La estructura de dirección con los parámetros del *socket* (de dirección, puerto, familia) que hemos inicializado previamente.

- El tamaño de la dirección que en el caso de entorno Internet es fijo, y siempre se utiliza la macro `'sizeof()'`.

Al finalizar la llamada, tendremos los parámetros de protocolo y dirección local (en nuestro caso servidor) configurados. Sólo nos faltarán los parámetros del lado remoto (en nuestro caso el cliente).

Para poder establecer la conexión con un cliente potencial, debemos tener algunas consideraciones importantes:

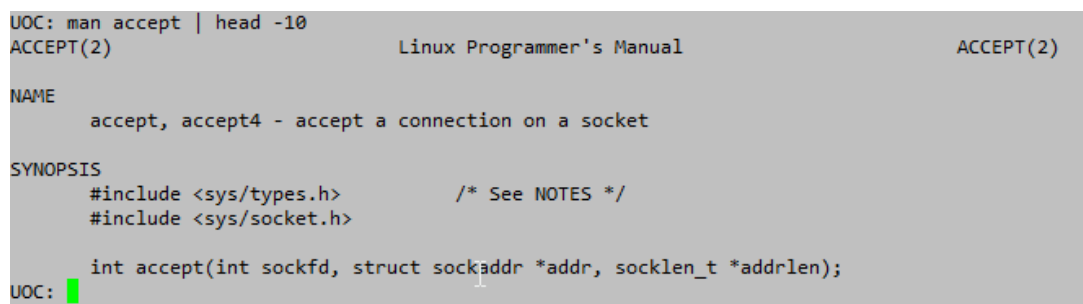
- En un entorno orientado a conexión, las comunicaciones pueden durar mucho tiempo. Para poder tener el control de la misma, el sistema operativo tendrá un *socket* (un canal de comunicación) para cada una de las conexiones. Para ello cuando se ejecute la llamada a la función `'accept()'`, el sistema operativo creará un nuevo *socket*, que será el que tenga todos los parámetros operativos. Es muy importante entender que cada conexión tendrá su propio descriptor de *socket*, como vemos en el Código 25.

```

68      /* Wait for a connection from a client */
69      clientsock = accept(serversock, (struct sockaddr *) &echoclient, &clientlen);
70      if (clientsock < 0) {
71          err_sys("Error accept");
72      }

```

Código 25: Aceptación de la petición de conexión por parte de un cliente con la función `'accept()'`.



```

UOC: man accept | head -10
ACCEPT(2)                                Linux Programmer's Manual                ACCEPT(2)

NAME
    accept, accept4 - accept a connection on a socket

SYNOPSIS
    #include <sys/types.h>                /* See NOTES */
    #include <sys/socket.h>

    int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
UOC:

```

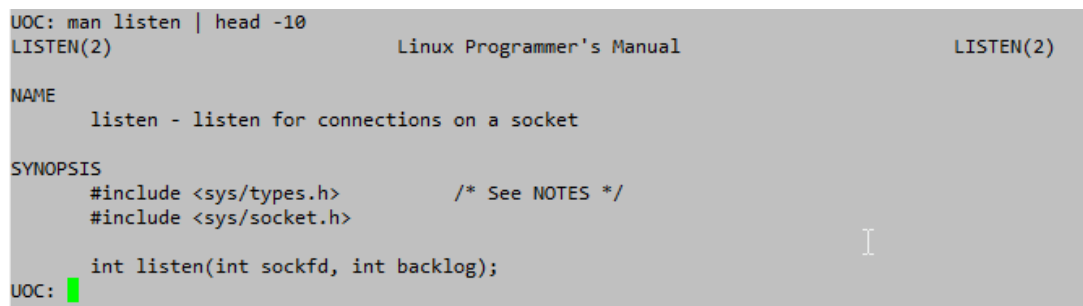
Figura 3.11: Manual de ayuda con la definición de la función `'accept()'`.

La función tiene tres parámetros de entrada, como vemos en la Figura 3.11:

- El descriptor del *socket* que el servidor tiene preparado y asociado con el sistema operativo para recibir las peticiones de conexión.

- La dirección de memoria donde se va a almacenar la información del lado remoto del *socket*.
- La dirección con el tamaño de dicha dirección. En el caso de las comunicaciones en entorno Internet este valor será fijo, pero aún así nos lo tiene que devolver el sistema operativo. Aunque se trata de un parámetro de retorno de la función, debemos siempre inicializarlo previamente, para evitar posibles disfunciones del programa.

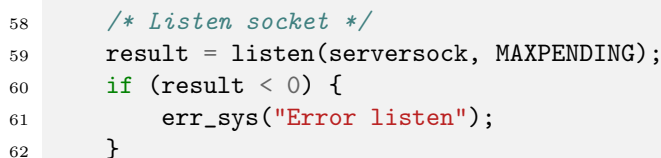
La función tiene un parámetro de retorno, que es el nuevo descriptor de *socket* con el que podremos operar, y que será el único que tenga todos los parámetros configurados, tanto en el lado local como en el lado remoto.



The screenshot shows the manual page for the `listen(2)` function. At the top, it says "UOC: man listen | head -10" and "LISTEN(2)". The title "Linux Programmer's Manual" is in the center, and "LISTEN(2)" is on the right. The "NAME" section states "listen - listen for connections on a socket". The "SYNOPSIS" section shows the header files `<sys/types.h>` and `<sys/socket.h>`, followed by the function signature `int listen(int sockfd, int backlog);`. The prompt "UOC:" is visible at the bottom left.

Figura 3.12: Manual de ayuda con la definición de la función '`listen()`'.

- En un entorno orientado a conexión podemos llegar a limitar el número de peticiones de conexión pendientes de ser atendidas (sería como llamadas en espera), con la función '`listen()`', como vemos en el Código 26, definida en la Figura 3.12. En algunas implementaciones del sistema operativo esta limitación no es operativa pues el límite inferior lo marca el propio sistema operativo, en valores de 1024/2048. Aún así es bueno utilizar la función.



```
58  /* Listen socket */
59  result = listen(serversock, MAXPENDING);
60  if (result < 0) {
61      err_sys("Error listen");
62  }
```

Código 26: Gestión del número de peticiones de conexión pendientes de ser atendidas en el lado servidor con la función '`listen()`'.

Ahora ya podemos utilizar el nuevo descriptor de *socket* para la comunicación bidireccional entre cliente y servidor. Cuando deseemos finalizarla, podemos cerrar el *socket* con una llamada a la función `'close()'`.

3.2.3. Pruebas de funcionamiento

Primero vamos a ver cómo nuestro *socket* se establece, entre cliente y servidor, y posteriormente veremos cómo intercambiar información. Trabajaremos con los dos programas de la carpeta `'01-tcp'`.

Para su compilado utilizaremos los ficheros `'tcp_server.Makefile'`, que tenemos en el Código 27 y `'tcp_client.Makefile'`, que tenemos en el Código 28.

Para compilar el cliente `'tcp_client1.c'`, como vemos en la Figura 3.13, utilizaremos el comando `'make tcp_client1 -f tcp_client.Makefile'`.

```
UOC: make tcp_client1 -f tcp_client.Makefile
gcc -c tcp_client1.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_client1 tcp_client1.o -I ./
UOC: █
```

Figura 3.13: Compilación del ejemplo `'tcp_client1.c'` con el fichero de configuración `'tcp_client.Makefile'`.

Para compilar el servidor `'tcp_server1.c'`, como vemos en la Figura 3.14, utilizaremos el comando `'make tcp_server1 -f tcp_server.Makefile'`.

```
UOC: make tcp_server1 -f tcp_server.Makefile
gcc -c tcp_server1.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_server1 tcp_server1.o -I ./
UOC: █
```

Figura 3.14: Compilación del ejemplo `'tcp_server1.c'` con el fichero de configuración `'tcp_server.Makefile'`.

Para ejecutar tanto el cliente como el servidor, primero vemos los parámetros necesarios, ejecutando el programa sin argumentos, para que los controles que hemos implementado en el lado cliente como vemos en el Código 29 y en el lado servidor como vemos en el Código 30 nos indique los parámetros, tal y como vemos en la Figura 3.15.

Fijaremos el número de puerto del servidor, en el valor 6000. Por lo tanto lo primero que vamos a hacer es lanzar el servidor esperando conexiones al puerto 6000, como vemos en la Figura 3.16. Ahora el servidor estará esperando la petición de conexión por parte de algún cliente.

Podemos comprobar cómo tenemos el puerto abierto, esperando la petición. Para ello utilizaremos la herramienta `'nmap'`, que permite escanear los puertos TCP y UDP

```
1 PROGRAM_NAME_1 = tcp_server1
2 PROGRAM_OBJS_1 = tcp_server1.o
3
4 PROGRAM_NAME_2 = tcp_server2
5 PROGRAM_OBJS_2 = tcp_server2.o
6
7 PROGRAM_NAME_3 = tcp_server3
8 PROGRAM_OBJS_3 = tcp_server3.o
9
10 PROGRAM_NAME_4 = tcp_server4
11 PROGRAM_OBJS_4 = tcp_server4.o
12
13 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
14 ↪ $(PROGRAM_NAME_4)
15 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
16 ↪ $(PROGRAM_OBJS_4)
17
18 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
19
20 all: $(PROGRAM_NAME_ALL)
21 ↪ @echo "Finished!"
22
23 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
24 ↪ gcc -o $$ $^ -I ./
25
26 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
27 ↪ gcc -o $$ $^ -I ./
28
29 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
30 ↪ gcc -o $$ $^ -I ./
31
32 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
33 ↪ gcc -o $$ $^ -I ./
34
35 %.o: %.c
36 ↪ gcc -c $< -Wall -Wno-unused-variable -I ./
37
38 clean:
39 ↪ rm -f $(REBUIDABLES)
40 ↪ @echo "Clean done"
```

Código 27: Fichero de configuración 'tcp_server.Makefile' de 'make' para todos los servidores

abiertos en un ordenador. En concreto utilizaremos el comando 'nmap -sT 127.0.0.1' que escaneará los puertos (TCP) abiertos en nuestra propia máquina ('localhost') como vemos en la Figura 3.17.

```

1 PROGRAM_NAME_1 = tcp_client1
2 PROGRAM_OBJS_1 = tcp_client1.o
3
4 PROGRAM_NAME_2 = tcp_client2
5 PROGRAM_OBJS_2 = tcp_client2.o
6
7 PROGRAM_NAME_3 = tcp_client3
8 PROGRAM_OBJS_3 = tcp_client3.o
9
10 PROGRAM_NAME_4 = tcp_client4
11 PROGRAM_OBJS_4 = tcp_client4.o
12
13 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
14 ↪ $(PROGRAM_NAME_4)
15 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
16 ↪ $(PROGRAM_OBJS_4)
17
18 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
19
20 all: $(PROGRAM_NAME_ALL)
21 ↪ @echo "Finished!"
22
23 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
24 ↪ gcc -o $$ $^ -I ./
25
26 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
27 ↪ gcc -o $$ $^ -I ./
28
29 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
30 ↪ gcc -o $$ $^ -I ./
31
32 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
33 ↪ gcc -o $$ $^ -I ./
34
35 %.o: %.c
36 ↪ gcc -c $< -Wall -Wno-unused-variable -I ./
37
38 clean:
39 ↪ rm -f $(REBUIDABLES)
40 ↪ @echo "Clean done"

```

Código 28: Fichero de configuración 'tcp_client.Makefile' de 'make' para todos los servidores

Nunca debe utilizarse esta herramienta en entornos de producción (por la afectación que podría tener en los servicios en producción) ni sin la autorización explícita por escrito del titular de la máquina que vamos a escanear.

```
UOC: ./tcp_server1
Usage: ./tcp_server1 <port>
UOC: ./tcp_client1
Usage: ./tcp_client1 <ip_server> <word> <port>
UOC: █
```

Figura 3.15: Ejecución de 'tcp_server1.c' y 'tcp_client1.c' sin parámetros para que el programa nos los indique.

```
24  /* Check input arguments */
25  if (argc != 4) {
26      fprintf(stderr, "Usage: %s <ip_server> <word> <port>\n", argv[0]);
27      exit(1);
28  }
```

Código 29: Control de los parámetros de entrada al programa 'tcp_client1'.

```
34  /* Check input arguments */
35  if (argc != 2) {
36      fprintf(stderr, "Usage: %s <port>\n", argv[0]);
37      exit(1);
38  }
```

Código 30: Control de los parámetros de entrada al programa 'tcp_server1'.

```
UOC: ./tcp_server1 6000
█
```

Figura 3.16: Ejecución de 'tcp_server1' esperando peticiones de conexión al puerto 6000.

Si no tenemos la utilidad instalada en nuestro sistema lo haremos con el comando 'apt-get install nmap', como vemos en la Figura 3.18.

Como vemos, el puerto 6000/TCP está abierto y se corresponde con el servicio 'X11' tal cómo define la *IANA* (*Internet Assigned Numbers Authority*).

La *Internet Assigned Numbers Authority* es una entidad privada sin ánimo de lucro de Estados Unidos, gestionada por *ICANN* (*Internet Corporation for Assigned Names and Numbers*) que supervisa de forma global la asignación de direcciones IP, la gestión de los servidores DNS raíz y otras asignaciones en los protocolos de Internet.

Del mismo modo, podemos utilizar el comando 'netstat', que nos permite inspeccionar las estadísticas de red del *kernel* del sistema operativo. Si no lo tuviéramos instalado

```
UOC: nmap -sT 127.0.0.1
Starting Nmap 7.70 ( https://nmap.org ) at 2019-08-18 12:17 UTC
Client: 127.0.0.1
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000087s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
6000/tcp  open  X11

Nmap done: 1 IP address (1 host up) scanned in 0.11 seconds
UOC: █
```

Figura 3.17: Escaneado de puertos TCP en nuestro sistema local con el comando 'nmap -sT 127.0.0.1'.

```
UOC: apt-get install nmap
Reading package lists... Done
```

Figura 3.18: Instalación de la utilidad 'nmap' en nuestro sistema.

en nuestro sistema lo haríamos con 'apt-get install net-tools' como vemos en la Figura 3.19.

```
UOC: apt-get install net-tools
Reading package lists... Done
```

Figura 3.19: Instalación de la utilidad 'netstat' en nuestro sistema.

En concreto, si ejecutamos el comando 'netstat -tulpn | grep LISTEN | grep 6000' podemos ver el puerto 6000 en estado 'LISTEN', tal y como vemos en la Figura 3.20.

```
UOC: netstat -tulpn | grep LISTEN | grep 6000
tcp        0      0 0.0.0.0:6000          0.0.0.0:*             LISTEN      23679/./tcp_ser
ver1
UOC: █
```

Figura 3.20: Comprobación que tenemos el puerto 6000 en estado 'LISTEN' con el comando 'netstat'.

Podemos comprobar que en nuestro sistema tenemos el programa 'tcp_server1' ejecutándose con el número de proceso (PID=23679), si ejecutamos el comando 'ps -aux', como vemos en la Figura 3.21.

Podemos matar todo proceso con el comando 'kill -9' y su número de PID, como vemos en la Figura 3.22.

Ahora volveremos a ejecutar el servidor, pero en modo *background*, para poder ejecutar en el mismo terminal el cliente. Cuando se ejecuta en modo *background* un proceso perdemos el control del teclado (pero en nuestro caso no lo necesitamos). Por lo tanto, ejecutaremos el servidor en modo *background* con el comando './tcp_server1 6000 &', como vemos en la Figura 3.23.

```
UOC: ps -aux|grep tcp
root    23679 71.3  0.0   2276   744 ?        R   12:06   27:51 ./tcp_server1 6000
root    24114  0.0  0.0   3084   896 ?        S+  12:45    0:00 grep tcp
UOC: █
```

Figura 3.21: Comprobación del PID del proceso 'tcp_server1' ejecutándose en nuestro sistema con el comando 'ps -aux'

```
UOC: kill -9 23679
UOC: ps -aux|grep tcp
root    24116  0.0  0.0   3084   880 ?        S+  12:46    0:00 grep tcp
[1]+  Killed                  ./tcp_server1 6000
UOC: █
```

Figura 3.22: Podemos matar cualquier proceso del sistema con el comando 'kill'.

```
UOC: ./tcp_server1 6000 &
[1] 24167
UOC: █
```

Figura 3.23: Ejecución del servidor en modo *background* con './tcp_server1 6000 &'.

El sistema ha asignado el número de proceso 24167, como podemos comprobar con el comando 'ps -aux', y vemos en la Figura 3.24.

```
UOC: ps -aux|grep tcp
root    24167  0.0  0.0   2144   748 ?        S   16:22    0:00 ./tcp_server1 6000
UOC: █
```

Figura 3.24: Comprobación que nuestro servidor ejecutándose en modo *background* tiene PID=24167.

Vemos como el servidor vuelve a estar a la escucha en el puerto 6000, como antes (con el comando 'netstat') tal y como vemos en la Figura 3.25. En esta ejecución para no volver a teclear el comando entero hemos ejecutado '!net', que ha recuperado el último comando que empezaba por 'net' de los que habíamos ejecutado en nuestro terminal.

```
UOC: !net
netstat -tulpn | grep LISTEN | grep 6000
tcp        0      0 0.0.0.0:6000          0.0.0.0:*           LISTEN      24167/./tcp_ser
ver1
UOC: █
```

Figura 3.25: Comprobación que nuestro servidor con PID=24167 está a la espera de peticiones de conexión en el puerto 6000.

Ahora que hemos comprobado que el servidor está escuchando por el puerto 6000/TCP vamos a ejecutar el cliente para que realice una petición. Para ello vamos a ejecutar el comando './tcp_client1 127.0.0.1 hello 6000 &', que vemos en la Figura 3.26 también en modo *background*. Vemos que el cliente ha tenido como número de proceso

el PID=24180. y como el servidor ha imprimido por pantalla (si tenemos control de la pantalla en modo escritura aunque estemos ejecutando el servidor en modo *background*) el mensaje 'Client: 127.0.0.1' que identifica que el servidor ha atendido una petición de conexión de un cliente desde la dirección '127.0.0.1' que es la que se le ha asignado al cliente al hacer la petición al servidor a la dirección '127.0.0.1'.

```
UOC: ./tcp_client1 127.0.0.1 hello 6000 &
[2] 24180
UOC: Client: 127.0.0.1
UOC: █
```

Figura 3.26: Ejecución del cliente en modo *background* con el comando './tcp_client1 127.0.0.1 hello 6000 &'.

Como tenemos un bucle infinito que bloquea la finalización del cliente como vemos en el Código 31, podremos ver el proceso cliente (y el servidor) en ejecución con el comando 'ps -aux' como vemos en la Figura 3.27, con PID 24167 para el servidor y 24180 para el cliente.

```
48  /* Just wait */
49  while (1);
```

Código 31: Bucle infinito para bloquear la finalización del cliente 'tcp_client1'.

```
UOC: ps -aux|grep tcp
root    24167 20.4  0.0   2276   748 ?        R   16:22   4:53 ./tcp_server1 6000
root    24180 49.4  0.0   2144   684 ?        R   16:36   4:53 ./tcp_client1 127.0.0.1 hello
6000
root    24182  0.0  0.0   3084   888 ?        S+  16:46   0:00 grep tcp
UOC: █
```

Figura 3.27: Comprobación de la ejecución de cliente y servidor en modo *background*.

Ahora con el comando 'ss -4 state established', podemos ver los *sockets* que están en estado 'established' para IPv4. Vemos que tenemos dos con el puerto 40070 (son la conexión que acabamos de establecer) y el puerto 'X11' que es el puerto 6000 donde tenemos el servidor. Vemos que todo *socket* al establecerse dispone de dos conexiones (en nuestro caso una del 40070 al 6000, y una segunda del 6000 al 40070), tal y como vemos en la Figura 3.28.

Ahora vamos a comprobar primero cual es la dirección (no 'localhost') de nuestro sistema con el comando 'ip addr show'. Tal y como vemos en la Figura 3.29 nuestro


```
UOC: ss -4 state established
```

Netid	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
tcp	0	0	127.0.0.1:40770	127.0.0.1:x11
tcp	0	0	127.0.0.1:x11	127.0.0.1:40770
tcp	0	0	192.168.122.254:33666	192.168.56.1:microsoft-ds

```
UOC: █
```

Figura 3.28: Comprobación de las conexiones establecidas en nuestro sistema.

sistema tiene la dirección IP '192.168.122.254'.

```
UOC: ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
10: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 76:31:1e:5c:4d:7f brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.254/24 brd 192.168.122.255 scope global eth0
        valid_lft forever preferred_lft forever
UOC: █
```

Figura 3.29: Comprobación de las direcciones asignadas en nuestro terminal.

Podemos pues, volver a ejecutar un cliente pero ahora haciendo la petición a la dirección IP '192.168.122.254', como vemos en la Figura 3.30.

```
UOC: ./tcp_client1 192.168.122.254 bye 6000 &
[3] 24186
UOC:
UOC: █
```

Figura 3.30: Ejecución de una segunda instancia del cliente en dirección IP '192.168.122.254'.

Ahora con el comando 'ss -4 state established', podemos ver los *sockets* que están en estado 'established' para IPv4. Vemos que tenemos las dos conexiones anteriores, y dos nuevas con el puerto 39100 (son la conexión que acabamos de establecer en la IP '192.168.122.254') y el puerto 'X11' que es el puerto 6000 donde tenemos el servidor, tal y como vemos en la Figura 3.31.

```
UOC: ss -4 state established
```

Netid	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
tcp	0	0	127.0.0.1:40770	127.0.0.1:x11
tcp	0	0	127.0.0.1:x11	127.0.0.1:40770
tcp	0	0	192.168.122.254:39100	192.168.122.254:x11
tcp	0	0	192.168.122.254:x11	192.168.122.254:39100
tcp	0	0	192.168.122.254:33666	192.168.56.1:microsoft-ds

```
UOC: █
```

Figura 3.31: Comprobación de las conexiones establecidas en nuestro sistema. Ahora vemos dos conexiones cliente-servidor.

Podemos comprobar la conectividad con nuestro servidor con la utilidad `'netcat'`. Si no la tuviéramos instalada, siempre lo podemos hacer con el comando `'apt-get install netcat'` como vemos en la Figura 3.32.

```
UOC: apt-get install netcat
Reading package lists... Done
```

Figura 3.32: Instalación de `'netcat'` en nuestro sistema.

Ahora ya podemos hacer la prueba de conexión al puerto 6000, como vemos en la Figura 3.33. En el momento de la ejecución, en modo *background* vemos que el mensaje de `'netcat'` de establecimiento de la conexión `'(UNKNOWN) [127.0.0.1] 6000 (x11) open'`. Si comprobamos las conexiones vemos que tenemos dos nuevas, que son las asignadas a `'netcat'` entre los puertos 41368 y 6000, como vemos en la Figura 3.34.

```
UOC: nc -vn 127.0.0.1 6000 &
[4] 24292
UOC: (UNKNOWN) [127.0.0.1] 6000 (x11) open

[4]+ Stopped nc -vn 127.0.0.1 6000
UOC:
```

Figura 3.33: Conexión con nuestro servidor al puerto 6000 con la utilidad `'netcat'`.

```
UOC: !ss
ss -t state established
Netid Recv-Q Send-Q Local Address:Port Peer Address:Port
tcp 0 0 127.0.0.1:40770 127.0.0.1:x11
tcp 0 0 127.0.0.1:x11 127.0.0.1:40770
tcp 0 0 192.168.122.254:39100 192.168.122.254:x11
tcp 0 0 127.0.0.1:x11 127.0.0.1:41368
tcp 0 0 192.168.122.254:x11 192.168.122.254:39100
tcp 0 0 127.0.0.1:41368 127.0.0.1:x11
tcp 0 0 192.168.122.254:33666 192.168.56.1:microsoft-ds
UOC: ^C
UOC: █
```

Figura 3.34: Comprobación de las conexiones establecidas en nuestro sistema. Ahora vemos tres conexiones cliente-servidor.

Si probáramos la petición de conexión a un puerto cerrado (por ejemplo el 8000) veríamos como `'netcat'` nos indica que la petición de conexión ha sido rechazada, con el mensaje `'(UNKNOWN) [127.0.0.1] 8000 (?) : Connection refused'` como vemos en la Figura 3.35.

Podemos para finalizar el apartado matar todos los procesos que tenemos en ejecución. Primero los buscaremos con el comando `'ps -aux'` como vemos en la Figura 3.36 y después los mataremos con el comando `'kill -9'` tal y como vemos en la Figura 3.37.

```
UOC: nc -vn 127.0.0.1 8000 &
[5] 24301
UOC: (UNKNOWN) [127.0.0.1] 8000 (?): Connection refused

[5]- Exit 1          nc -vn 127.0.0.1 8000
UOC: █
```

Figura 3.35: Mensaje de error de 'netcat' al intentar una petición de conexión al puerto 8000 que está cerrado.

```
UOC: ps -aux|grep tcp & ps -aux|grep nc
[5] 24310
root    24167 31.3  0.0   2276   748 ?        R   16:22   20:09 ./tcp_server1 6000
root    24180 40.0  0.0   2144   684 ?        R   16:36   20:09 ./tcp_client1 127.0.0.1 hello
6000
root    24186 32.8  0.0   2144   744 ?        R   16:58   9:17 ./tcp_client1 192.168.122.254
bye 6000
root    24310  0.0  0.0   3084   820 ?        S   17:26   0:00 grep tcp
root    24292  0.0  0.0   2372  1840 ?        T   17:11   0:00 nc -vn 127.0.0.1 6000
root    24312  0.0  0.0   3084   888 ?        S+  17:26   0:00 grep nc
[5]- Done          ps -aux | grep tcp
UOC: █
```

Figura 3.36: Comprobación del número PID de los procesos en ejecución.

```
UOC: kill -9 24167
UOC: kill -9 24180
[1] Killed          ./tcp_server1 6000
UOC: kill -9 24186
[2] Killed          ./tcp_client1 127.0.0.1 hello 6000
UOC: kill -9 24292
[3]- Killed         ./tcp_client1 192.168.122.254 bye 6000
UOC:
[4]+ Killed         nc -vn 127.0.0.1 6000
UOC: █
```

Figura 3.37: Eliminación de los procesos pendientes con el comando 'kill -9'.

Ahora eliminamos el bucle infinito después de la petición de conexión por parte del cliente, y hacemos que funcione de acuerdo con lo previsto, y que por lo tanto su ejecución sea muy rápida. Sólo ponemos la parte de código con la modificación en el cliente 'tcp_client2.c' en el Código 32 y en la rutina de gestión de la conexión con el cliente en el servidor con nombre 'tcp_server2.c' Código 33.

Podremos ver como podemos ir ejecutando diversas veces el cliente, con el mismo servidor, que irá atendiendo los diferentes clientes de forma secuencial. Primero aceptará una petición de conexión, la atenderá y al finalizar atenderá a un nuevo cliente. Primero compilaremos tanto el cliente como el servidor utilizando nuestros ficheros de configuración 'tcp_client.Makefile' y 'tcp_server.Makefile' tal y como vemos en la Figura 3.38.

Ejecutamos el servidor en modo *background* en el puerto 6000, y 5 veces el cliente, con diferentes configuraciones de mensaje y direcciones IP. Vemos como se ha ejecutado el servidor, como se van ejecutando los clientes y el mensaje en el servidor. No es necesario ejecutar el cliente en modo *background* pues hemos eliminado el bucle infinito, y su

```

17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver;
19     char buffer[BUFFSIZE];
20     unsigned int echolen;
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 4) {
26         fprintf(stderr, "Usage: %s <ip_server> <word> <port>\n", argv[0]);
27         exit(1);
28     }
29
30     /* Try to create TCP socket */
31     sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
32     if (sock < 0) {
33         err_sys("Error socket");
34     }
35
36     /* Set information for sockaddr_in */
37     memset(&echoserver, 0, sizeof(echoserver));           /* reset memory */
38     echoserver.sin_family = AF_INET;                       /* Internet/IP */
39     echoserver.sin_addr.s_addr = inet_addr(argv[1]);      /* IP address */
40     echoserver.sin_port = htons(atoi(argv[3]));          /* server port */
41
42     /* Try to have a connection with the server */
43     result = connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver));
44     if (result < 0) {
45         err_sys("Error connect");
46     }
47     fprintf(stdout, " done \n");
48
49     /* Close socket */
50     close(sock);
51
52     exit(0);
53 }

```

Código 32: Nuevo programa cliente sin bucle infinito 'tcp_client2.c'.

ejecución finaliza de forma inmediata, como vemos en la Figura 3.39. Vemos para cada cliente el mensaje correspondiente en el servidor informando de una nueva conexión.

Ahora vamos a implementar con este servidor/cliente base, un servidor de eco. El cliente le enviará al servidor el parámetro entrado ('hello') y el servidor lo imprimirá por pantalla.

En el lado servidor hacemos las modificaciones en la rutina de gestión de la conexión

```
18 void handle_client(int sock) {
19     char buffer[BUFFSIZE];
20     int received = -1;
21
22     /* Close socket */
23     close(sock);
24 }
```

Código 33: Nuevo programa servidor sin bucle infinito 'tcp_server2.c'.

```
UOC: make tcp_server2 -f tcp_server.Makefile
gcc -c tcp_server2.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_server2 tcp_server2.o -I ./
UOC: make tcp_client2 -f tcp_client.Makefile
gcc -c tcp_client2.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_client2 tcp_client2.o -I ./
UOC:
UOC: █
```

Figura 3.38: Compilación de 'tcp_client2.c' y 'tcp_server2.c'.

```
JOC: ./tcp_server2 6000 &
[1] 24345
JOC:
JOC: ./tcp_client2 127.0.0.1 mess1 6000
Client: 127.0.0.1
done
JOC: ./tcp_client2 127.0.0.1 mess2 6000
Client: 127.0.0.1
done
JOC: ./tcp_client2 192.168.122.254 mess3 6000
Client: 192.168.122.254
done
JOC: ./tcp_client2 127.0.0.1 mess4 6000
Client: 127.0.0.1
done
JOC: ./tcp_client2 192.168.122.254 mess5 6000
Client: 192.168.122.254
done
JOC: █
```

Figura 3.39: Ejecución de varias peticiones de 'tcp_client2.c'.

en una nueva versión 'tcp_server3.c' como vemos en el Código 34.

La función de lectura 'read()', tiene tres parámetros de entrada, de acuerdo con la entrada del manual que vemos en la Figura 3.41.

Los tres parámetros de entrada son:

1. El descriptor del *socket* sobre del que vamos a leer. De dónde vamos a leer.
2. Una dirección de memoria, donde almacenaremos la información que vamos a recibir.

```

18 void handle_client(int sock) {
19     char buffer[BUFFSIZE];
20     int received = -1;
21
22     /* Read from socket */
23     read(sock, &buffer[0], BUFFSIZE);
24     printf("Message from client: %s\n", buffer);
25
26     /* Close socket */
27     close(sock);
28 }

```

Código 34: Nuevo programa servidor como servidor de eco 'tcp_server3.c'.

```

UOC: man 2 write | head -10
WRITE(2)                                Linux Programmer's Manual                                WRITE(2)

NAME
    write - write to a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);

UOC: █

```

Figura 3.40: Información del manual de la función 'write()'.

```

UOC: man read | head -10
READ(2)                                Linux Programmer's Manual                                READ(2)

NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);

UOC: █

```

Figura 3.41: Información del manual de la función 'read()'.

3. El número máximo de bytes de información que queremos recibir.

La función nos devolverá el número de bytes realmente leídos, que será el parámetro que utilizaremos para controlar la información a escribir. Este valor siempre será igual o inferior al máximo indicado. Si el valor es negativo nos indicará un error en la lectura, y deberemos gestionarlo.

En el lado cliente, modificaremos el programa de la nueva versión 'tcp_client3.c'

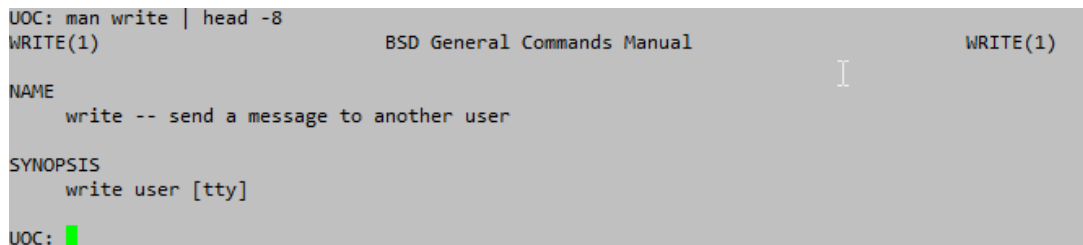
como vemos en el para incluir el envío de la información al servidor en el Código 35.

```
17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver;
19     char buffer[BUFFSIZE];
20     unsigned int echolen;
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 4) {
26         fprintf(stderr, "Usage: %s <ip_server> <word> <port>\n", argv[0]);
27         exit(1);
28     }
29
30     /* Try to create TCP socket */
31     sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
32     if (sock < 0) {
33         err_sys("Error socket");
34     }
35
36     /* Set information for sockaddr_in */
37     memset(&echoserver, 0, sizeof(echoserver));           /* reset memory */
38     echoserver.sin_family = AF_INET;                       /* Internet/IP */
39     echoserver.sin_addr.s_addr = inet_addr(argv[1]);      /* IP address */
40     echoserver.sin_port = htons(atoi(argv[3]));         /* server port */
41
42     /* Try to have a connection with the server */
43     result = connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver));
44     if (result < 0) {
45         err_sys("Error connect");
46     }
47
48     /* Write to socket */
49     write(sock, argv[2], strlen(argv[2]) + 1);
50     fprintf(stdout, " done \n");
51
52     /* Close socket */
53     close(sock);
54
55     exit(0);
56 }
```

Código 35: Nuevo programa cliente como petición de eco 'tcp_client3.c'.

La función de escritura 'write()', tiene tres parámetros de entrada, de acuerdo con la entrada del manual que vemos en la Figura 3.40. Es importante destacar la importancia de poner la opción 2 cuando hacemos la consulta. Si no la pusiéramos, veríamos que no

tenemos la entrada de manual de la función `'write()'` sino de una utilidad de sistema que nos permite enviar mensajes a otros usuarios de nuestro mismo sistema, como vemos en la Figura 3.42.



```
UOC: man write | head -8
WRITE(1)                                BSD General Commands Manual                                WRITE(1)

NAME
    write -- send a message to another user

SYNOPSIS
    write user [tty]

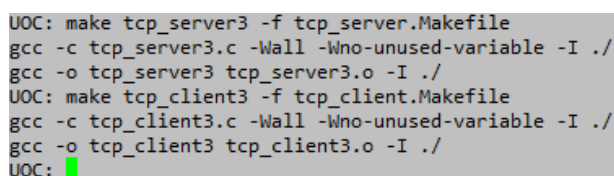
UOC: █
```

Figura 3.42: Información del manual de la utilidad de sistema `'write'`.

Los tres parámetros de entrada son:

1. El descriptor del *socket* sobre el que vamos a escribir. Dónde vamos a escribir.
2. Una dirección de memoria, donde localizaremos la información que vamos a enviar.
3. El número de bytes de información que queremos enviar. En este caso vamos a utilizar la función `'strlen()'` que nos devolverá el número de caracteres de la palabra al que sumaremos 1 para incluir el final de *string* (`'\0'`).

Como siempre, lo primero será generar los ficheros ejecutables para `'tcp_client3.c'` y `'tcp_server3.c'` como vemos en la Figura 3.43.




```
UOC: make tcp_server3 -f tcp_server.Makefile
gcc -c tcp_server3.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_server3 tcp_server3.o -I ./
UOC: make tcp_client3 -f tcp_client.Makefile
gcc -c tcp_client3.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_client3 tcp_client3.o -I ./
UOC: █
```

Figura 3.43: Generación de los ficheros ejecutables `'tcp_client3'` y `'tcp_server3'`.


Lanzamos el servidor, ahora en modo *foreground*, para tener control total sobre el proceso (tanto entrada como salida). Como vemos en la Figura 3.44 el programa se ha quedado con el control del terminal y por lo tanto en dicho terminal no podemos realizar otras acciones.

Ahora podríamos parar el programa servidor con la combinación `'CTRL+C'`, con lo que finalizaría su ejecución (y obviamente volveríamos a tener control del terminal, pero sin el programa servidor en ejecución), como vemos en la Figura 3.45.



```
UOC: ./tcp_server3 6000
```


Figura 3.44: Ejecución del servidor 'tcp_server3' escuchando en el puerto 6000 en modo *foreground*.



```
UOC: ./tcp_server3 6000
^C
UOC: 
```

Figura 3.45: Paramos la ejecución del servidor 'tcp_server3' con la combinación de teclas 'CTRL+C'.


Si lo que queremos es colocar el proceso que hemos ejecutado en modo *foreground* en modo *background*, para poder ejecutar nuevos comandos en el terminal, como vemos en la Figura 3.46 podremos hacerlo con la combinación de teclas 'CTRL+Z' para en primer lugar parar el proceso, y a continuación con el comando 'bg' enviar el proceso a modo *background*. De hecho como vemos, es cómo si hubiéramos ejecutado directamente el servidor en modo *background* con el comando './tcp_server3 6000 &'.



```
UOC: ./tcp_server3 6000
^Z
[1]+  Stopped                  ./tcp_server3 6000
UOC: bg
[1]+ ./tcp_server3 6000 &
UOC:
UOC: 
```

Figura 3.46: Paramos la ejecución del servidor 'tcp_server3' con la combinación de teclas 'CTRL+Z' y luego pasamos el proceso a modo *background* con el comando 'bg'.

Con esta solución vemos como podemos enviar a modo *background* cualquier proceso que esté en ejecución, si queremos tener el control del terminal (aunque sea temporalmente): primero lo paramos ('CTRL+Z') y luego lo enviamos a modo *background* ('bg'). Es importante conocer también que siempre podemos volver el proceso al modo *foreground* para poder hacer uso del terminal en el programa con el comando 'fg 1' como vemos en la Figura 3.47.




```
UOC: fg 1
./tcp_server3 6000
```

Figura 3.47: Volvemos a poner la ejecución del servidor 'tcp_server3' con el comando 'fg 1' a modo *foreground*.

Para entender mejor el parámetro del comando 'fg', vamos a ejecutar, como vemos

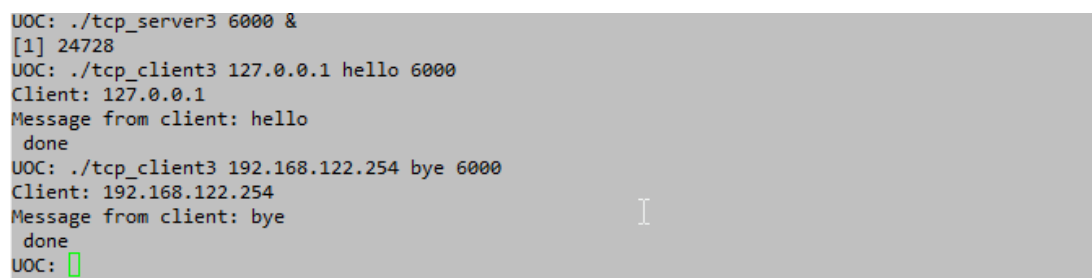
en la Figura 3.48, una segunda instancia del servidor pero ahora escuchando en el puerto 7000, también en modo *background*. Como vemos el primer proceso que pusimos en modo *background* tenía asignado el número de proceso 1 ('[1]+ ./tcp_server3 6000 &'), y el segundo el número de proceso 2 ('[2]+ ./tcp_server3 7000 &') (la segunda instancia de nuestro servidor). Lo primero que haremos será cambiar a modo *foreground* la primera instancia de nuestro servidor con el comando 'fg 1', como anteriormente; luego lo volvemos a mover a *background* y en segundo lugar cambiaremos a modo *foreground* la segunda instancia con el comando 'fg 2'. Por lo tanto en todo momento podemos control los procesos.



```
UOC: ./tcp_server3 7000 &
[2] 24717
UOC: fg 1
./tcp_server3 6000
^Z
[1]+  Stopped                  ./tcp_server3 6000
UOC: bg
[1]+ ./tcp_server3 6000 &
UOC: fg 2
./tcp_server3 7000
^Z
[2]+  Stopped                  ./tcp_server3 7000
UOC: bg
[2]+ ./tcp_server3 7000 &
UOC: █
```

Figura 3.48: Control de varios procesos con los comandos 'fg' y 'bg'.

Ahora ya podemos ejecutar como vemos en la Figura 3.49 el servidor y varias instancias del cliente para ver la impresión de los mensajes recibidos por parte del servidor, para la primera conexión del cliente 'Client: 127.0.0.1', 'Message from client: hello', 'done', y para la segunda conexión 'Client: 192.168.122.254', 'Message from client: bye', 'done'.



```
UOC: ./tcp_server3 6000 &
[1] 24728
UOC: ./tcp_client3 127.0.0.1 hello 6000
Client: 127.0.0.1
Message from client: hello
done
UOC: ./tcp_client3 192.168.122.254 bye 6000
Client: 192.168.122.254
Message from client: bye
done
UOC: █
```

Figura 3.49: Ejecución del servidor 'tcp_server3' y varias instancias seguidas del cliente 'tcp_client3'.

Para finalizar con este primer ejemplo podemos implementar ahora el retorno del texto del servidor al cliente, para disponer de un sencillo servidor de eco. Hacemos pues

las variaciones en la rutina de gestión de la conexión en el servidor `'tcp_server4.c'` como vemos en el Código 36. Para devolver el mensaje, en lugar de tomar el número de caracteres leídos, utilizamos la función `'strlen()'` para tomar el tamaño de la palabra.

```
19 void handle_client(int sock) {
20     char buffer[BUFSIZE];
21     int received = -1;
22
23     /* Read from socket */
24     read(sock, &buffer[0], BUFSIZE);
25     printf("Message from client: %s\n", buffer);
26
27     /* Write to socket */
28     write(sock, buffer, strlen(buffer) + 1);
29
30     /* Close socket */
31     close(sock);
32 }
```

Código 36: Nuevo programa servidor como generación de respuesta a la petición de eco `'tcp_server4.c'`.

En el lado cliente, como vemos en el Código 37, después de enviar el mensaje al servidor, imprimimos un mensaje de confirmación de envío (`'fprintf(stdout, " sent \n ");'`) a la salida estándar. Esperamos el eco del servidor (`'read(sock, buffer, BUFSIZE);'`) y lo imprimimos como muestra de correcto funcionamiento de nuestro sistema de eco a la salida estándar nuevamente (`'fprintf(stdout, " s ...done \n ", buffer);'`).

Para la validación de este último ejemplo, vamos a ejecutar el servidor en un sistema y los clientes en diferentes sistemas.

El primero de nuestros sistemas tiene IP `'192.168.122.203'` y está identificado con el *PROMPT* de sistema `'U0C1:'`, como vemos en la Figura 3.50. Por contra el segundo está identificado con el *PROMPT* de sistema `'U0C2:'` y tiene IP `'192.168.122.254'` como vemos en la Figura 3.51.

En el primer sistema generaremos el fichero ejecutable para el servidor `'tcp_server4'` como vemos en la Figura 3.52 y en el segundo sistema el fichero ejecutable para el cliente `'tcp_client4'` como vemos en la Figura 3.53.

Como ahora funcionamos en dos sistemas y terminales diferentes, podemos ejecutar tanto el servidor como el cliente en modo *foreground*.

Primero ejecutamos el servidor `'tcp_server4'` en el primer sistema, y vemos como

```

42  /* Try to have a connection with the server */
43  result = connect(sock, (struct sockaddr *) &echoserver, sizeof(echoserver));
44  if (result < 0) {
45      err_sys("Error connect");
46  }
47
48  /* Write to socket */
49  write(sock, argv[2], strlen(argv[2]) + 1);
50  fprintf(stdout, " sent \n");
51
52  /* Read from socket */
53  read(sock, buffer, BUFSIZE);
54  fprintf(stdout, " %s ...done \n", buffer);
55
56  /* Close socket */
57  close(sock);

```

Código 37: Nuevo programa cliente 'tcp_client4.c' para imprimir la respuesta recibida en forma de eco por el servidor .

```

UOC1: ip addr show dev eth0
9: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 3a:38:93:c1:4c:d7 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.203/24 brd 192.168.122.255 scope global eth0
        valid_lft forever preferred_lft forever
UOC1: ip addr show dev eth0|grep inet|awk '{print $2}'
192.168.122.203/24
UOC1: █

```

Figura 3.50: Identificación del primer sistema con 'PROMPT' 'UOC1:' y dirección IP '192.122.122.203'.

```

UOC2: ip addr show dev eth0
10: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 76:31:1e:5c:4d:7f brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.254/24 brd 192.168.122.255 scope global eth0
        valid_lft forever preferred_lft forever
UOC2: ip addr show dev eth0|grep inet|awk '{print $1}'
inet
UOC2: █

```

Figura 3.51: Identificación del segundo sistema con 'PROMPT' 'UOC2:' y dirección IP '192.122.122.254'.

recibimos la respuesta de conexión y mensaje de un cliente desde la IP '192.168.122.254' en la Figura 3.54. En el segundo sistema ejecutamos el cliente y vemos como se recibe el mensaje de eco en la Figura 3.55.

```
UOC1: make tcp_server4 -f tcp_server.Makefile
gcc -c tcp_server4.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_server4 tcp_server4.o -I ./
UOC1: █
```

Figura 3.52: Generación del ejecutable 'tcp_server4' en el sistema con dirección IP '192.122.122.203'.

```
UOC2: make tcp_client4 -f tcp_client.Makefile
gcc -c tcp_client4.c -Wall -Wno-unused-variable -I ./
gcc -o tcp_client4 tcp_client4.o -I ./
UOC2: █
```

Figura 3.53: Generación del ejecutable 'tcp_client4' en el sistema con dirección IP '192.122.122.254'.

```
UOC1: ./tcp_server4 6000
Client: 192.168.122.254
Message from client: hello
█
```

Figura 3.54: Ejecución del servidor 'tcp_server4' y respuesta de la conexión.

```
UOC2: ./tcp_client4 192.168.122.203 hello 6000
sent
hello ...done
UOC2: █
```

Figura 3.55: Ejecución del cliente 'tcp_client4' y respuesta de la conexión.

3.3. Comunicaciones no orientadas a conexión

Una vez hemos visto el funcionamiento con *sockets* para la puesta en marcha de soluciones de comunicación orientadas a conexión, pasemos al funcionamiento de los sistemas no orientados a conexión.

En estos sistemas no se hace ningún establecimiento de conexión previo al envío de información entre los participantes de la comunicación, sino que directamente, el cliente envía mensajes al servidor, que los recibirá sin el previo establecimiento.

El funcionamiento es más sencillo, pues sólo tenemos envío y recepción de mensajes. Al utilizar el protocolo de comunicación UDP (como veremos más adelante) para las comunicaciones no hay ningún procedimiento implementado de control de errores, ni ninguna garantía de recepción de los mensajes.

En nuestros ejemplos por simplicidad no implementaremos tales mecanismos de control, pues supondremos que no tenemos pérdidas en la comunicación, al utilizar nuestra red local, que nos da alta fiabilidad (al tener buen ancho de banda y muy poco tráfico).

Por lo tanto, para las comunicaciones no orientadas a conexión (en un entorno Internet), deberemos de fijarnos en el emisor (solemos denominarlo cliente) y en el receptor

(solemos denominarlo servidor), y las funciones que cada uno debe realizar que tenemos en la Tabla 3.2.

EMISOR (CLIENTE)		RECEPTOR(SERVIDOR)
3.- Emitir una emisión de información a la dirección (IPs) del servidor, y al puerto (Ps) del servidor		1.- Informar al sistema operativo de que cuando reciba alguna llamada (a la dirección indicada (IPs), y al puerto indicado (Ps) se ponga en contacto con el proceso/programa que gestiona el servicio
		2.- Esperar una petición de envío de información (IPs,Ps)
		4.- Recepcionar el mensaje enviado. Se conoce la dirección IP y el puerto del emisor, con lo que existe la posibilidad de enviar una respuesta
6.- El cliente recibe la respuesta del servidor		5.- Servidor puede enviar una respuesta al cliente
	Se puede continuar con la conversación en la forma que se desee, pero no hay establecimiento de conexión. Cuidado que podríamos recibir peticiones de otro cliente, y ello provocaría problemas en la sincronización cliente-servidor	

Tabla 3.2: Modelo de comunicación no orientado a conexión.

Por lo tanto, deberemos usar las funciones (llamadas al sistema) adecuadas para

implementar los pasos descritos previamente.

Primero presentaremos los códigos del cliente y del servidor, para a continuación analizar cada paso.

3.3.1. Desarrollo del programa cliente (satélite)

El cliente en una comunicación no orientada a conexión es el primero en enviar mensaje, pues quien está identificado (publicado, es conocido) es el servidor. Al enviar un mensaje, al servidor, éste tendrá la información necesaria de identificación de su interlocutor y le podría enviar un mensaje de respuesta.

```
1  /*
2   * Filename: udp_client1.c
3   */
4
5  #include <stdio.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #define BUFFSIZE 5
14
15 void err_sys(char *mess) { perror(mess); exit(1); }
16
17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver, echoclient;
19     unsigned int echolen, clientlen;
20     char buffer[BUFFSIZE];
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 4) {
26         fprintf(stderr, "Usage: %s <ip_server> <word> <port>\n", argv[0]);
27         exit(1);
28     }
29
```

```

30  /* Create UDP socket */
31  sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
32  if (sock < 0) {
33      err_sys("Error on socket creation");
34  }
35
36  /* Configure/set socket address for the server */
37  memset(&echoserver, 0, sizeof(echoserver));      /* Erase the memory area */
38  echoserver.sin_family = AF_INET;                 /* Internet/IP */
39  echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
40  echoserver.sin_port = htons(atoi(argv[3]));     /* Server port */
41
42  /* Try to send the word to server */
43  echolen = strlen(argv[2]);
44  result = sendto(sock, argv[2], echolen, 0, (struct sockaddr *) &echoserver,
45      ↪ sizeof(echoserver));
46  if (result != echolen) {
47      err_sys("error writing word on socket");
48  }
49
50  /* Set the maximum size of address to be received */
51  clientlen = sizeof(echoclient);
52
53  /* Wait for echo from server */
54  fprintf(stdout, "Server echo: ");
55  received = recvfrom(sock, buffer, BUFSIZE, 0, (struct sockaddr *) &echoclient,
56      ↪ &clientlen);
57  if (received != echolen) {
58      err_sys("Error reading");
59  }
60  buffer[received] = '\0';
61
62  /* Print word on screen */
63  fprintf(stdout, "%s", buffer);
64  fprintf(stdout, "\n");
65
66  /* Close socket */
67  close(sock);
68
69  exit(0);
70 }

```


Código 38: Programa cliente `'udp_client1.c'` la comunicación no orientada a conexión con un servidor.

En el Código 38 podemos ver el código completo del cliente `'udp_client1.c'`

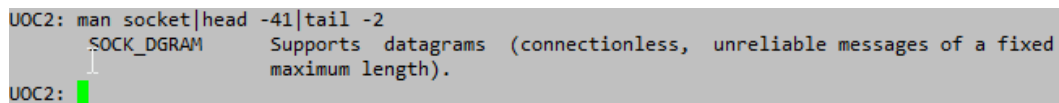
Estructura con la información del socket

Mantiene la estructura utilizada en el caso de las comunicaciones orientadas a conexión.

Debemos generar el descriptor de *socket* que vamos a utilizar para la comunicación. Que tendrá los mismos parámetros que en el caso de las comunicaciones orientadas a conexión.

Analicemos los parámetros de entrada:

1. **'Domain'**: nos indicará el tipo de “entorno” de comunicación que vamos a utilizar. En nuestro caso, vamos a utilizar un entorno Internet, con direcciones IP. Aquí como vemos en el código utilizamos la opción `'PF_INET'`. que nos indicará `'Protocol_Family_INET (InterNET)'`.
2. **'Type'**: Para trabajar en entornos no orientados a conexión utilizaremos `'SOCK_DGRAM'`, como vemos en la Figura 3.56.



```
UOC2: man socket | head -41 | tail -2
      SOCK_DGRAM      Supports datagrams (connectionless, unreliable messages of a fixed
                        maximum length).
UOC2: 
```

Figura 3.56: Ayuda del sistema `'man socket'` para ver cuando usamos `'SOCK_DGRAM'` como tipo

3. **'Protocol'**: Donde definiremos el protocolo de transporte UDP, para las comunicaciones no orientadas a conexión.

Por lo tanto, en el momento de la creación del descriptor del *socket* con el que vamos a trabajar, deberemos fijar siempre los mismos valores para las comunicaciones no orientadas a conexión en entornos internet, como vemos en el Código 39.

Tras la creación del descriptor de *socket* como hicimos en el caso de las comunicaciones orientadas a conexión, en el caso de las comunicaciones no orientadas a conexión, necesitaremos inicializar la dirección del servidor, para poder comunicarnos con él, como vemos en el Código 40.

La función para el envío de información que vamos a utilizar en las comunicaciones no orientadas a conexión `'sendto()'` tiene seis parámetros de entrada, tal y como vemos en la Figura 3.57.

```

30  /* Create UDP socket */
31  sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
32  if (sock < 0) {
33      err_sys("Error on socket creation");
34  }

```

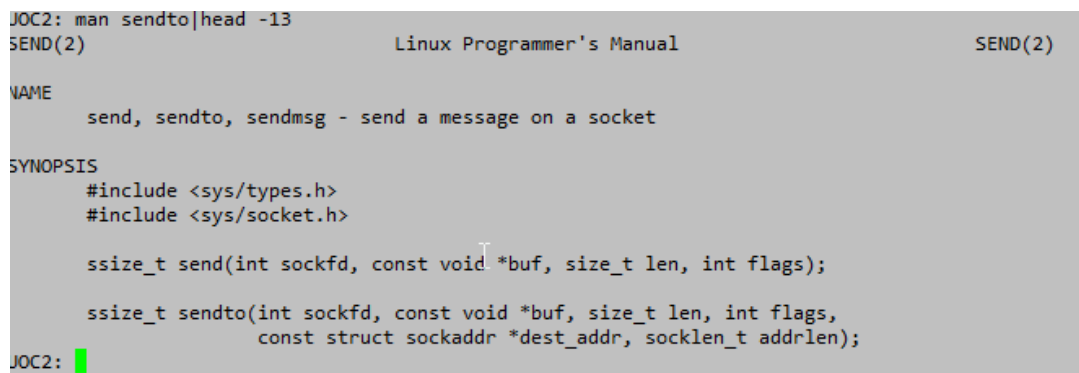
Código 39: Creación del descriptor de *socket* en una comunicación no orientada a conexión tipo.

```

36  /* Configure/set socket address for the server */
37  memset(&echoserver, 0, sizeof(echoserver)); /* Erase the memory area */
38  echoserver.sin_family = AF_INET; /* Internet/IP */
39  echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
40  echoserver.sin_port = htons(atoi(argv[3])); /* Server port */

```

Código 40: Inicialización de la dirección del servidor en una comunicación no orientada a conexión tipo.



```

JOC2: man sendto|head -13
SEND(2)                                Linux Programmer's Manual                                SEND(2)

NAME
    send, sendto, sendmsg - send a message on a socket

SYNOPSIS
    #include <sys/types.h>
    #include <sys/socket.h>

    ssize_t send(int sockfd, const void *buf, size_t len, int flags);

    ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                   const struct sockaddr *dest_addr, socklen_t addrlen);
JOC2: █

```

Figura 3.57: Ayuda del sistema 'man sendto' para ver los parámetros de la función

Los parámetros en cuestión:

1. El descriptor del *socket* a través del cual vamos a enviar la información.
2. La dirección de memoria en la que tenemos almacenada la información que vamos a enviar.
3. El número máximo de bytes que vamos a enviar.

Estos tres primeros parámetros son los mismos que en su momento utilizamos en la

función `'write()'` para escribir (enviar información) en una comunicación orientada a conexión una vez establecida la conexión.

4. Un parámetro que de momento pondremos siempre a 0, que nos permitirá configurar algunas opciones de la comunicación no orientada a conexión. Para más información podemos consultar la página del manual `'man sendto|head -105|tail -49'` para ver la información sobre el parámetro `'flags'`.
5. La dirección de memoria, donde tenemos almacenada la información del lado servidor del *socket*. Es el mismo tipo de información que utilizamos en la función `'connect()'` en el caso de las comunicaciones orientadas a conexión.
6. El número de bytes de la información de la dirección (como en el caso de la función `'connect'`).

De hecho como vemos en el Código 41 los parámetros de la función `'sendto'` son la agregación de los parámetros de la función `'connect()'` y la función `'write()'` en el caso de las comunicaciones orientadas a conexión. Necesitamos por un lado la información del servidor con el que nos vamos a comunicar (como en la función `'connect()'`) y por otro lado la información del mensaje que queremos enviar (como en la función `'write()'`).

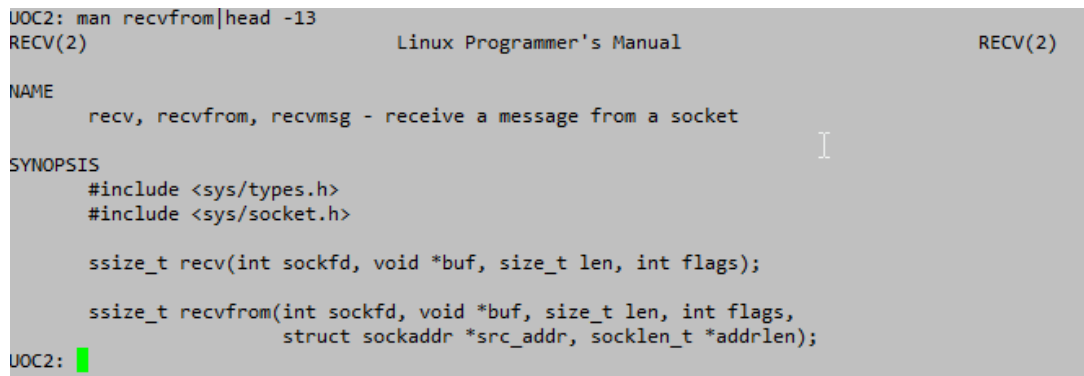
```
42  /* Try to send the word to server */
43  echolen = strlen(argv[2]);
44  result = sendto(sock, argv[2], echolen, 0, (struct sockaddr *) &echoserver,
45               ↪ sizeof(echoserver));
46  if (result != echolen) {
47      err_sys("error writing word on socket");
  }
```

Código 41: Envío de información al servidor en una comunicación no orientada a conexión tipo.

Los parámetros del lado cliente, se toman de forma automática por parte del sistema operativo, como sucedía en el caso de la función `'connect()'` en las comunicaciones orientadas a conexión. La dirección IP es una de las posibles en nuestro sistema (alguna de las existentes), y el número de puerto, en toda comunicación el lado cliente lo elige de forma pseudoaleatoria el sistema operativo de entre los números disponibles.

La llamada a la función `'sendto()'` nos devolverá (como en el caso de la función `'write()'`) el número de bytes realmente enviados hacia el servidor. Lo cual nos permitiría garantizar que hemos enviado la información que deseábamos enviar.

Como en este caso, hemos implementado directamente el programa con el eco en el servidor y la impresión del mensaje de retorno en el cliente, deberemos estar preparados para la recepción del mensaje de eco. Para ello utilizaremos la función `'recvfrom()'` que también tiene seis parámetros de entrada como vemos en la Figura 3.58.



```

UOC2: man recvfrom | head -13
RECV(2)                                Linux Programmer's Manual                                RECV(2)

NAME
    recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS
    #include <sys/types.h>
    #include <sys/socket.h>

    ssize_t recv(int sockfd, void *buf, size_t len, int flags);

    ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                     struct sockaddr *src_addr, socklen_t *addrlen);
UOC2:

```

Figura 3.58: Ayuda del sistema `'man recvfrom'` para ver los parámetros de la función.

Los parámetros en cuestión:

1. El descriptor del *socket* a través del cual vamos a recibir la información.
2. La dirección de memoria en la que vamos a almacenar la información que vamos a recibir.
3. El número máximo de bytes que vamos a intentar leer/recibir.

Estos tres primeros parámetros son los mismos que en su momento utilizamos en la función `'read()'` para leer (recibir información) en una comunicación orientada a conexión una vez establecida la conexión.

4. Un parámetro que de momento pondremos siempre a 0, que nos permitirá configurar algunas opciones de la comunicación no orientada a conexión. Para más información podemos consultar la página del manual `'man sendto | head -105 | tail -49'` para ver la información sobre el parámetro `'flags'`.
5. La dirección de memoria, donde vamos a almacenar la información del lado servidor del *socket*. Es el mismo tipo de información que utilizamos en la función `'accept()'` en el caso de las comunicaciones orientadas a conexión.
6. La dirección en la que vamos a almacenar el número de bytes de la dirección del servidor (como en el caso de la función `'accept'`). Como entonces, aunque sea un

parámetro de retorno, y por eso hacemos el paso de parámetro por referencia se recomienda inicializarlo a un valor máximo antes de hacer la llamada a la función.

De hecho como vemos en el Código 42 los parámetros de la función `'recvfrom'` son la agregación de los parámetros de la función `'accept()'` y la función `'read()'` en el caso de las comunicaciones orientadas a conexión. Necesitamos por un lado la información del servidor del que vamos a recibir el mensaje (como en la función `'accept()'`) y por otro lado la información del mensaje que vamos a recibir (como en la función `'read()'`).

```
52  /* Wait for echo from server */
53  fprintf(stdout, "Server echo: ");
54  received = recvfrom(sock, buffer, BUFSIZE, 0, (struct sockaddr *) &echoclient,
    ↪  &clientlen);
55  if (received != echolen) {
56      err_sys("Error reading");
57  }
58  buffer[received] = '\0';
```

Código 42: Recepción de información del servidor en una comunicación no orientada a conexión tipo.

Una vez hemos recibido el mensaje, garantizamos que tiene un final de *string* al final, y lo imprimimos, como vemos en el Código 43.

```
58  buffer[received] = '\0';
59
60  /* Print word on screen */
61  fprintf(stdout, "%s", buffer);
62  fprintf(stdout, "\n");
```

Código 43: Impresión del mensaje devuelto por el servidor.

Ahora sólo nos quedará en el momento que ya no queramos mantener el *socket*, cerrarlo con una llamada a la función `'close(sock)'`.

3.3.2. Desarrollo del programa servidor (estación base)

Para la puesta en marcha del servidor, vemos en Código 44, la parte de programa para la configuración de un servidor de eco. Éste va a estar en un bucle infinito, esperando un

mensaje de un posible cliente, y devolviéndole el mismo mensaje en forma de eco.

```

1  /*
2   * Filename: udp_server1.c
3   */
4
5  #include <stdio.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #define BUFFSIZE 255
14
15 void err_sys(char *mess) { perror(mess); exit(1); }
16
17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver, echoclient;
19     unsigned int echolen, clientlen, serverlen;
20     char buffer[BUFFSIZE];
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 3) {
26         fprintf(stderr, "Usage: %s <ip_server> <port>\n", argv[0]);
27         exit(1);
28     }
29
30     /* Create UDP socket */
31     sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
32     if (sock < 0) {
33         err_sys("Error socket");
34     }
35
36     /* Prepare sockaddr_in structure for server address */
37     memset(&echoserver, 0, sizeof(echoserver)); /* Erase the memory area */
38     echoserver.sin_family = AF_INET; /* Internet/IP */
39     echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* we will receive message only
    ↪ to that IP on server */

```

```
40  //echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /* We can receive requests from
    ↪ any IP address valid for server*/
41  echoserver.sin_port = htons(atoi(argv[2]));      /* Server port */
42
43  serverlen = sizeof(echoserver);
44
45  /* Bind that socket with the OS, to be able to receive messages on that socket
    ↪ */
46  result = bind(sock, (struct sockaddr *) &echoserver, serverlen);
47  if (result < 0) {
48      err_sys("Error bind");
49  }
50
51  /* as a server we are in an infinite loop, waiting forever */
52  while (1) {
53      /* Set the maximum size for address */
54      clientlen = sizeof(echoclient);
55
56      /* we receive a message from a particular client */
57      received = recvfrom(sock, buffer, BUFSIZE, 0, (struct sockaddr *)
    ↪ &echoclient, &clientlen);
58      if (received < 0) {
59          err_sys("Error receiveing word from client");
60      }
61
62      /* Print client address */
63      fprintf(stderr, "Client: %s, Message: %s\n", inet_ntoa(echoclient.sin_addr),
    ↪ buffer);
64
65      /* Try to send echo word back to the client */
66      result = sendto(sock, buffer, received, 0, (struct sockaddr *) &echoclient,
    ↪ sizeof(echoclient));
67      if (result != received) {
68          err_sys("Error writing message back to the client");
69      }
70  }
71 }
```

Código 44: Código del servidor 'udp_server1.c'.

En el lado servidor vamos a aprovechar los comentarios de las funciones de envío

(`'sendto()'`) y recepción (`'recvfrom()'`) y de la codificación del servidor para las comunicaciones orientadas a conexión en la parte de declaración del *socket* y de su publicación con la función `'bind()'` en el sistema operativo, y por lo tanto, no vamos a hacer un comentario detallado de todas ellas.

3.3.3. Pruebas de funcionamiento

Para las pruebas de funcionamiento, en primer lugar utilizaremos los ficheros de configuración que tenemos en Código 45 para el `'udp_server1.c'` y en Código 46 para el `'udp_server1.c'`.

```
1 PROGRAM_NAME_1 = udp_server1
2 PROGRAM_OBJS_1 = udp_server1.o
3
4 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1)
5 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1)
6
7 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
8
9 all: $(PROGRAM_NAME_ALL)
10    @echo "Finished!"
11
12 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
13    gcc -o $@ $^ -I ./
14
15 %.o: %.c
16    gcc -c $< -Wall -Wno-unused-variable -I ./
17
18 clean:
19    rm -f $(REBUIDABLES)
20    @echo "Clean done"
```

Código 45: Código del fichero de configuración `'udp_server.Makefile'`.

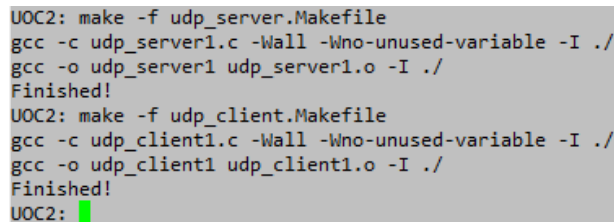
```
1 PROGRAM_NAME_1 = udp_client1
2 PROGRAM_OBJS_1 = udp_client1.o
3
4 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1)
```



```
5 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1)
6
7 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
8
9 all: $(PROGRAM_NAME_ALL)
10     @echo "Finished!"
11
12 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
13     gcc -o $@ $^ -I ./
14
15 %.o: %.c
16     gcc -c $< -Wall -Wno-unused-variable -I ./
17
18 clean:
19     rm -f $(REBUIDABLES)
20     @echo "Clean done"
```

Código 46: Código del fichero de configuración 'udp_client.Makefile'.

Tras generar los dos ficheros ejecutables con el comando 'make' como vemos en la Figura 3.59.



```
UOC2: make -f udp_server.Makefile
gcc -c udp_server1.c -Wall -Wno-unused-variable -I ./
gcc -o udp_server1 udp_server1.o -I ./
Finished!
UOC2: make -f udp_client.Makefile
gcc -c udp_client1.c -Wall -Wno-unused-variable -I ./
gcc -o udp_client1 udp_client1.o -I ./
Finished!
UOC2: █
```

Figura 3.59: Generación de los ficheros ejecutables 'udp_server1' y 'udp_client1'.

Para ejecutar tanto el cliente como el servidor, primero lo haremos sin parámetros para que el control de parámetros que tenemos nos informe de los parámetros para la correcta ejecución de ambos programas como vemos en la Figura 3.60.

Como vemos el servidor tendrá dos parámetros de entrada, que serán la dirección IP a la que podrá recibir mensajes, y el puerto. En esta implementación no aceptación mensajes de cualquier cliente, sino sólo aquellos que nos los envían a la dirección configurada. El cliente tiene los mismos tres parámetros que en los ejemplos de comunicación orientada a conexión (la dirección del servidor, el mensaje que le vamos a enviar y el puerto en el que el servidor estará operativo).

```
JOC2: ./udp_server1
Usage: ./udp_server1 <ip_server> <port>
JOC2: ./udp_client1
Usage: ./udp_client1 <ip_server> <word> <port>
JOC2:
JOC2: █
```

Figura 3.60: Ejecución de los programas 'udp_server1' y 'udp_client1' sin parámetros para saber cuáles son.

Para empezar con las ejecuciones, lanzamos el servidor en modo *background* a la dirección del terminal y en el puerto 6000. (como en la Figura 3.61 vemos el servidor se ejecutará en el sistema 1, esperando en la dirección localhost '127.0.0.1').

```
UOC1: ./udp_server1 127.0.0.1 6000 &
[1] 602
UOC1: █
```

Figura 3.61: Ejecución del programa 'udp_server1'.

Ahora el servidor estará esperando mensajes por parte de algún cliente.

Podemos comprobar cómo tenemos el puerto abierto con la herramienta de escaneo de puertos 'nmap'.

Como en nuestra implementación del servidor no hacemos la comprobación correctamente de la recepción del mensaje, al recibir el mensaje de la utilidad 'nmap' que no tiene ningún contenido, se nos va a producir un error que puede bloquear nuestro sistema. Para corregirlo, vamos a implementar para la prueba con 'nmap' una segunda versión del servidor que en caso de recibir un mensaje sin contenido, no va a generar el eco, como vemos en el Código 47.

```
1  /*
2   * Filename: udp_server2.c
3   */
4
5  #include <stdio.h>
6  #include <sys/socket.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #define BUFFSIZE 255
14
```

```
15 void err_sys(char *mess) { perror(mess); exit(1); }
16
17 int main(int argc, char *argv[]) {
18     struct sockaddr_in echoserver, echoclient;
19     unsigned int echolen, clientlen, serverlen;
20     char buffer[BUFFSIZE];
21     int sock, result;
22     int received = 0;
23
24     /* Check input arguments */
25     if (argc != 3) {
26         fprintf(stderr, "Usage: %s <ip_server> <port>\n", argv[0]);
27         exit(1);
28     }
29
30     /* Create UDP socket */
31     sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
32     if (sock < 0) {
33         err_sys("Error socket");
34     }
35
36     /* Prepare sockaddr_in structure for server address */
37     memset(&echoserver, 0, sizeof(echoserver)); /* Erase the memory area */
38     echoserver.sin_family = AF_INET; /* Internet/IP */
39     echoserver.sin_addr.s_addr = inet_addr(argv[1]); /* we will receive message only
    ↪ to that IP on server */
40     //echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /* We can receive requests from
    ↪ any IP address valid for server*/
41     echoserver.sin_port = htons(atoi(argv[2])); /* Server port */
42
43     serverlen = sizeof(echoserver);
44
45     /* Bind that socket with the OS, to be able to receive messages on that socket
    ↪ */
46     result = bind(sock, (struct sockaddr *) &echoserver, serverlen);
47     if (result < 0) {
48         err_sys("Error bind");
49     }
50
51     /* as a server we are in an infinite loop, waiting forever */
52     while (1) {
53         /* Set the maximum size for address */
```

```

54     clientlen = sizeof(echoclient);
55
56     /* we receive a message from a particular client */
57     received = recvfrom(sock, buffer, BUFSIZE, 0, (struct sockaddr *)
        ↪ &echoclient, &clientlen);
58     if (received < 0) {
59         err_sys("Error receiveing word from client");
60     }
61
62     if (received != 0) {
63         /* Print client address */
64         fprintf(stderr, "Client: %s, Message: %s\n", inet_ntoa(echoclient.sin_addr),
        ↪ buffer);
65
66         /* Try to send echo word back to the client */
67         result = sendto(sock, buffer, received, 0, (struct sockaddr *) &echoclient,
        ↪ sizeof(echoclient));
68         if (result != received) {
69             err_sys("Error writing message back to the client");
70         }
71     }
72 }
73 }

```

Código 47: Código del servidor 'udp_server2.c'.

Ahora podremos compilar el servidor ('gcc -o udp_server2 udp_server2.c', en este caso sin utilizar la utilidad 'make'). Lo ejecutamos en la dirección *localhost*, y podremos comprobar con el comando 'nmap -sU -p 6000 127.0.0.1' que tenemos el puerto UDP 6000 preparado como vemos en la Figura 3.62.

```

UOC1: gcc -o udp_server2 udp_server2.c
UOC1: ./udp_server2 127.0.0.1 6000 &
[1] 77
UOC1: nmap -sU -p 6000 127.0.0.1
Starting Nmap 7.70 ( https://nmap.org ) at 2019-08-19 12:07 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up.

PORT      STATE      SERVICE
6000/udp  open|filtered X11

Nmap done: 1 IP address (1 host up) scanned in 2.05 seconds
UOC1:
UOC1: █

```

Figura 3.62: Comprobación del puerto UDP 6000 abierto en nuestro sistema con la utilidad 'nmap'.

Recordemos que nunca debemos utilizarla en entornos de producción ni sin la autorización explícita y por escrito del titular del sistema, como ya indicamos antes.

Si utilizamos el comando `netstat`, podemos ver el puerto 6000, preparado (con las opciones adecuadas `'netstat -tulpn'` del comando para ver los números de puerto) como vemos en la Figura 3.63.

```
UOC1: netstat -tulpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
udp        0      0 127.0.0.1:6000         0.0.0.0:*               LISTEN      777/./udp_server2
UOC1: █
```

Figura 3.63: Comprobación del puerto UDP 6000 abierto en nuestro sistema con la utilidad `'netstat'`.

Recordemos que no podremos utilizar la utilidad `'ss'` para ver las conexiones establecidas, pues en este caso no hay establecimientos de conexión.

Para comprobar ahora sí el funcionamiento de nuestro cliente `'udp_client1'` y de nuestro servidor `'udp_server1'` vamos a ejecutar cada uno en un sistema diferente.

El servidor lo ejecutaremos en el sistema 1 (que tiene IP `'192.168.122.140'`) en modo *background* y en el puerto 6000, como vemos en la Figura 3.64.

```
UOC1: ip addr show | grep inet
inet 127.0.0.1/8 scope host lo
inet6 ::1/128 scope host
inet 192.168.122.140/24 brd 192.168.122.255 scope global eth0
UOC1: ./udp_server1 192.168.122.140 6000 &
[1] 109
UOC1: █
```

Figura 3.64: Ejecución del servidor `'udp_server1'` en el sistema 1 con IP `'192.168.122.140'`.

El cliente lo ejecutaremos en el sistema 2 (que tiene IP `'192.168.122.88'`) en modo *background* y atacaremos al servidor que está en el sistema 1 (que tiene IP `'192.168.122.140'`) esperando en el puerto 6000, como vemos en la Figura 3.65. Como vemos el cliente ha recibido el mensaje de eco por parte del servidor (`'Server echo: hello'`).

```
UOC2: ip addr show | grep inet
inet 127.0.0.1/8 scope host lo
inet6 ::1/128 scope host
inet 192.168.122.88/24 brd 192.168.122.255 scope global eth0
UOC2: ./udp_client1 192.168.122.140 hello 6000
Server echo: hello
UOC2: █
```

Figura 3.65: Ejecución del cliente `'udp_client1'` en el sistema 2 con IP `'192.168.122.88'`.

En el lado servidor, vemos como el mensaje que se imprime en la Figura 3.66 es `'hell'` en lugar de `'hello'`.

```
UOC1:
UOC1: Client: 192.168.122.88, Message: hell
```

Figura 3.66: Mensaje recibido por el servidor 'udp_server1' donde vemos que no se imprime correctamente.

Para corregir el error en la presentación del mensaje en el lado servidor, pues vemos que el mensaje de eco es correcto en el cliente, vamos a añadir la misma inserción del carácter de final de *string* que ponemos al recibir el eco en el cliente, en una nueva implementación del servidor 'udp_server3.c' que vemos en el Código 48.

```
52  while (1) {
53      /* Set the maximum size for address */
54      clientlen = sizeof(echoclient);
55
56      /* we receive a message from a particular client */
57      received = recvfrom(sock, buffer, BUFSIZE, 0, (struct sockaddr *)
58          ↪ &echoclient, &clientlen);
59      if (received < 0) {
60          err_sys("Error receiveing word from client");
61      }
62      buffer[received]='\0';
63
64      /* Print client address */
65      fprintf(stderr, "Client: %s, Message: %s\n", inet_ntoa(echoclient.sin_addr),
66          ↪ buffer);
67
68      /* Try to send echo word back to the client */
69      result = sendto(sock, buffer, received, 0, (struct sockaddr *) &echoclient,
70          ↪ sizeof(echoclient));
71      if (result != received) {
72          err_sys("Error writing message back to the client");
73      }
74  }
```

Código 48: Código del servidor 'udp_server3.c'.

Ahora al ejecutar nuevamente el servidor, y recibir una petición de eco, el mensaje ya es correcto en la Figura 3.67. Con las estructuras de *string* siempre hay que vigilar a la hora de manejar, entre otras cosas los mensajes de impresión. Cuando usamos la impresión de un 'string' imprimiremos todos los caracteres que vayamos encontrando hasta que localizamos un final de 'string'. Si por el camino encontramos ruido"que incluye caracteres de control como podría ser 'retrocede un carácter' el mensaje que

se imprime es engañoso. Una solución es incluir al final de los mensajes recibidos siempre un final de 'string' como hemos hecho en la implementación de 'udp_server3.c'.

```
UOC1:
UOC1: Client: 192.168.122.88, Message: hell
```

Figura 3.67: Mensaje recibido por el servidor 'udp_server3' donde vemos que ahora sí se imprime correctamente.

Podemos ver algunas estadísticas de paquetes recibidos y enviados en nuestro sistema con el comando 'netstat -s', como vemos en la Figura 3.68.

```
UOC1: netstat -s |head -32
Ip:
  Forwarding: 1
  506 total packets received
  0 forwarded
  0 incoming packets discarded
  500 incoming packets delivered
  360 requests sent out
Icmp:
  0 ICMP messages received
  0 input ICMP message failed
  ICMP input histogram:
  0 ICMP messages sent
  0 ICMP messages failed
  ICMP output histogram:
Tcp:
  4 active connection openings
  0 passive connection openings
  0 failed connection attempts
  0 connection resets received
  0 connections established
  480 segments received
  341 segments sent out
  0 segments retransmitted
  0 bad segments received
  4 resets sent
Udp:
  20 packets received
  0 packets to unknown port received
  0 packet receive errors
  16 packets sent
  0 receive buffer errors
  0 send buffer errors
UOC1: █
```

Figura 3.68: Estadísticas de paquetes en nuestro sistema 1 con el comando 'netstat -s'.

Para finalizar, vamos a ejecutar el comando 'nc -u 192.168.122.140 6000' para simular una petición de un cliente. Nos pedirá como vemos en la Figura 3.69. Esto nos puede ser interesante para validar que nuestro servidor trata las peticiones UDP de clientes de forma correcta.

En el lado servidor, como vemos en la Figura 3.70 tenemos el mensaje recibido, y la identificación de la IP del cliente simulado con la petición con la utilidad 'nc'.

```
UOC2: nc -u 192.168.122.140 6000
HELLO
HELLO
^C
UOC2: █
```

Figura 3.69: Simulación de una petición de cliente utilizando el comando 'nc -u 192.168.122.140 6000' en el sistema 2.

```
UOC1: Client: 192.168.122.88, Message: HELLO
UOC1: █
```

Figura 3.70: Mensaje de recibido en el servidor con la petición simulada con el comando 'nc -u 192.168.122.140 6000' en el sistema 2.

Capítulo 4

Gestión de concurrencia con procesos y *mutex*

4.1. Introducción

En el capítulo anterior hemos aprendido cómo enviar datos entre un cliente y un servidor utilizando los protocolos de Internet, TCP (orientado a conexión) y UDP (no orientado a conexión). Típicamente un cliente mantiene un *socket* para comunicarse con un servidor, pero un servidor debe mantener abiertos múltiples *sockets* de manera simultánea, uno para cada cliente.

En el caso de las comunicaciones no orientadas a conexión mantener más de una conversación al mismo tiempo requiere de una complicada gestión de las mismas. No somos capaces de diferenciar la conversación más que por el hecho que en cada mensaje recibimos la identificación de nuestro interlocutor. Si queremos poder mantener varias al mismo tiempo, en cada recepción de mensaje deberemos de identificar el interlocutor (IP y puerto) y entonces mantener una lista (por decir un tipo) de conversaciones de acuerdo con ambos parámetros. Ello complica mucho la gestión en el lado servidor, y por lo tanto habitualmente las comunicaciones no orientadas a conexión se reservan para protocolos de rápida respuesta. El cliente lanza una petición que es fácil de resolver y por lo tanto no es necesario ninguna gestión especial en el servidor. Como en el caso de nuestro servidor de eco, se recibirá un mensaje, y se atenderá. Ningún cliente enviará más de un mensaje enlazado; si enviara dos mensajes se considerarían como dos comunicaciones independientes. Aquí lo importante es que la gestión de la respuesta sea rápida.

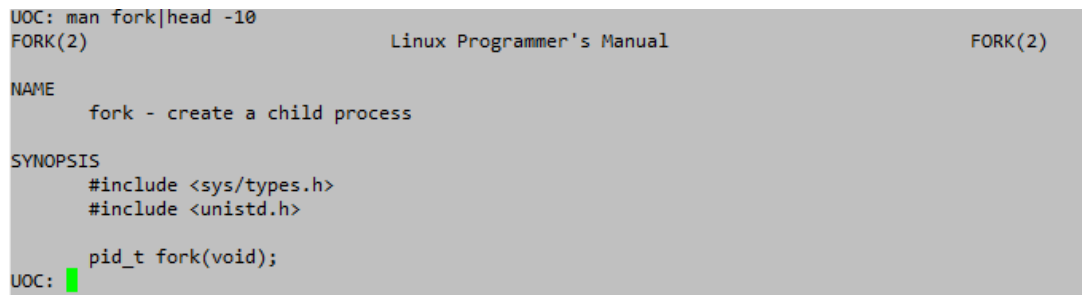
¿Qué pasa en el caso de comunicaciones orientadas a conexión? Podríamos entender el mecanismo similar a la telefonía cuando se establecía la conexión a través de centralita

(operador/operadora). El cliente llamaba a la centralita, allí era recepcionado y transferido a destino final. En nuestro caso, el cliente hará petición de conexión al *socket* abierto por el servidor, pero una vez establecida la conexión, en el lado servidor generaremos un nuevo proceso para poder atender al cliente de forma independiente, mientras el propio proceso servidor volverá a esperar una nueva petición de conexión.

Teniendo en cuenta esto, en este capítulo vamos a estudiar los mecanismos que permiten crear programas con múltiples hilos de ejecución. Concretamente estudiaremos la creación de procesos y *threads* a partir de uno en ejecución, así como compartir recursos entre diferentes procesos y como gestionar la ejecución concurrente de los mismos en nuestro programa.

4.2. Procesos

La función (llamada al sistema) que nos permite duplicar un proceso es `fork()`, tal y como vemos en la Figura 4.1. La función `fork()` es especial debido a que puede devolver dos veces un valor de retorno y en ningún caso podemos conocer cuál va a ser el primero en ser devuelto. Así pues, si la llamada al sistema tiene éxito y duplica el proceso devolverá valor para el proceso padre, y también para el proceso hijo.



```
UOC: man fork|head -10
FORK(2)                                Linux Programmer's Manual                                FORK(2)

NAME
    fork - create a child process

SYNOPSIS
    #include <sys/types.h>
    #include <unistd.h>

    pid_t fork(void);
UOC: █
```

Figura 4.1: Formato de la función `fork()`.

Si el valor de retorno es positivo (mayor que cero), nos indicará que estamos nuevamente en el proceso padre, y dicho valor será el PID del nuevo proceso hijo creado. Conocer el PID de su nuevo hijo puede ser de utilidad para el proceso padre, por ejemplo para esperar que finalice su ejecución antes de seguir.

Por contra, cuando el valor de retorno sea 0 nos indicará que la bifurcación ha tenido éxito y estamos en el proceso hijo, es decir, el nuevo proceso creado. En este caso podemos utilizar una llamada a la función `getppid()` para conocer el PID del proceso padre.

Finalmente, si se produjera un error en la llamada, devolvería un -1 sólo en el proceso padre, pues no se crearía entonces el nuevo proceso.

Así pues, si la función `fork()` tiene éxito tendremos dos procesos en el sistema: el proceso que ya teníamos en ejecución (que denominaremos proceso padre) y el nuevo proceso resultante de la duplicación (que denominaremos hijo).

Evidentemente cada proceso del sistema debe tener su propio PID (identificador de proceso). Cada proceso dispone de una zona de memoria (indexada por PID) en la que guardará toda la información relevante del mismo: el código, el punto de ejecución del mismo, las variables de memoria, los descriptores de los ficheros y *sockets* con los que está trabajando, y otra información). Por tanto, el nuevo proceso tendrá su propio PID, y por lo tanto también su propia zona de información. Un proceso una zona de forma biunívoca.

El proceso llamado padre, no se crea de nuevo, y por lo tanto, es el mismo proceso que se estaba ejecutando previamente a la llamada a la función. Habremos añadido en su información un nuevo proceso hijo (los procesos padre/hijo tienen ambos información de la existencia del otro).

Por su parte el proceso, como nuevo proceso que es no tenemos histórico y por lo tanto todos los contadores están inicializados a cero.

En definitiva a partir de la existencia de ambos procesos no comparten nada, son dos procesos cada uno con su zona de información y por lo tanto cuando uno hace modificaciones (porque modificada una variable) el otro proceso no tiene conocimiento del cambio. Eso sí las variables (incluyendo los descriptores) que el proceso padre tenía en el momento de la llamada, el código y el punto de ejecución del mismo se han compartido. Por lo tanto, el nuevo proceso hijo tendrá el mismo código y el mismo estado (en cuanto a variables, descriptores, ...) que el proceso padre. Se encontrará en el mismo punto de ejecución (en el punto de retorno de la función `fork()` -a continuación detallaremos un poco más este punto) y tendrá las mismas variables y con el mismo valor.

Vamos a ver, pues, como implementar ahora una bifurcación en el proceso servidor para poder atender varias conexiones con clientes de forma concurrente.

En nuestro caso lo interesante será que al compartir el mismo código, ambos (el proceso padre y el proceso hijo) tendrán implementada la rutina de gestión de la conexión. Ambos tendrán información del *socket* establecido pues haremos la bifurcación justo después de aceptar la petición de conexión.

Por lo tanto, veamos cómo bifurcar un nuevo proceso en el servidor, para atender la nueva petición de conexión. Debemos implementar las siguientes funcionalidades:

1. Después de la conexión crear un nuevo proceso para atender la conexión.
2. En el proceso padre, cerrar el nuevo descriptor de *socket*, el que gestionará la conexión, pues el padre no debe gestionarla y de esta forma se gestiona de forma más eficiente la memoria.
3. En el proceso padre, volver al bucle de espera de petición de conexión.
4. En el proceso hijo, podemos cerrar el descriptor de *socket* inicial, el que se publicó en el sistema operativo para esperar la petición de conexión, pues el proceso hijo no ha de esperar peticiones de conexión sino gestionar la conexión que se acaba de establecer con el nuevo descriptor de *socket* obtenido.
5. En el proceso hijo deberemos gestionar la comunicación.
6. En el proceso hijo deberemos finalizar la conexión con el cliente.

Para implementar nuestro servidor concurrente, retomaremos nuestro servidor orientado a conexión, pero antes de nada debemos entender el funcionamiento de la bifurcación (creación) de procesos con la función `fork()`. Para ello hemos preparado un fichero de configuración de `make fork.Makefile` para la generación de nuestros ejecutables, que tenemos en el Código 49.

```
1 PROGRAM_NAME_1 = fork1
2 PROGRAM_OBJS_1 = fork1.o
3
4 PROGRAM_NAME_2 = fork2
5 PROGRAM_OBJS_2 = fork2.o
6
7 PROGRAM_NAME_3 = fork3
8 PROGRAM_OBJS_3 = fork3.o
9
10 PROGRAM_NAME_4 = fork4
11 PROGRAM_OBJS_4 = fork4.o
12
13 PROGRAM_NAME_5 = fork5
14 PROGRAM_OBJS_5 = fork5.o
15
16 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
   ↪ $(PROGRAM_NAME_4) $(PROGRAM_NAME_5)
17 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
   ↪ $(PROGRAM_OBJS_4) $(PROGRAM_OBJS_5)
```

```
18
19 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
20
21 all: $(PROGRAM_NAME_ALL)
22     @echo "Finished!"
23
24 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
25     gcc -o $@ $^ -I ./
26
27 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
28     gcc -o $@ $^ -I ./
29
30 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
31     gcc -o $@ $^ -I ./
32
33 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
34     gcc -o $@ $^ -I ./
35
36 $(PROGRAM_NAME_5): $(PROGRAM_OBJS_5)
37     gcc -o $@ $^ -I ./
38
39 %.o: %.c
40     gcc -c $< -Wall -Wno-unused-variable -I ./
41
42 clean:
43     rm -f $(REBUIDABLES)
44     @echo "Clean done"
```

Código 49: Fichero de configuración de `make fork.Makefile`.

Primero empezamos con el Código 50 que no tiene ninguna bifurcación, pero que nos servirá de base para los ejemplos posteriores.

```
1 /*
2  * Filename: fork1.c
3  */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
```

```

8  #include <unistd.h>
9
10 void err_sys(char *mess) { perror(mess); exit(1); }
11
12 int main(int argc, char *argv[]) {
13     int pid, ppid;
14     int i, top;
15
16     /* Check input arguments */
17     if (argc != 2) {
18         fprintf(stderr, "Usage: %s <value>\n", argv[0]);
19         exit(1);
20     }
21
22     top = atoi(argv[1]);
23
24     for (i = 0; i < top; i++) {
25         fprintf(stdout, "Value: %d\t", i);
26     }
27
28     fprintf(stdout, "\n");
29
30     exit(0);
31 }

```

Código 50: Base para los ejemplos de la función `fork()`. `fork1.c`.

Para la generación del ejecutable ejecutamos el comando `make fork1 -f fork.Makefile` como vemos en la Figura 4.2.

```

UOC1: make fork1 -f fork.Makefile
gcc -c fork1.c -Wall -Wno-unused-variable -I ./
gcc -o fork1 fork1.o -I ./
UOC1: █

```

Figura 4.2: Generación del ejecutable `fork1` con el comando `make fork1 -f fork.Makefile`.

Si ejecutamos el programa sin argumentos, el control de argumentos nos indica que debemos pasarle como argumento de entrada un número. Si lo introducimos como vemos en la Figura 4.3 en un bucle va imprimiendo por pantalla los valores de forma consecutiva, hasta imprimir tantos como el argumento introducido.

En el siguiente ejemplo que generaremos con el comando `make fork2 -f fork.Makefile`

```

UOC1: ./fork1
Usage: ./fork1 <value>
UOC1: ./fork1 3
Value: 0      Value: 1      Value: 2
UOC1: ./fork1 6
Value: 0      Value: 1      Value: 2      Value: 3      Value: 4      Value: 5
UOC1:
UOC1:

```

Figura 4.3: Ejemplos de ejecución del programa `fork1`.

imprimiremos el valor del PID del proceso que estamos ejecutando, y el PID de su proceso padre. Todos los procesos del sistema tienen un proceso padre, añadiendo el Código 51 al programa base.

```

25  fprintf(stdout, "My PID is %d, my parent PID is %d\n", (int)getpid(),
    ↪  (int)getppid());

```

Código 51: Código añadido en `fork2.c` para imprimir PID del proceso y de su proceso padre.

Como vemos en la Figura 4.4 el proceso en ejecución tiene $PID = 11669$, y su proceso padre tiene $PID = 1$.

```

UOC1: make fork2 -f fork.Makefile
gcc -c fork2.c -Wall -Wno-unused-variable -I ./
gcc -o fork2 fork2.o -I ./
UOC1: ./fork2
Usage: ./fork2 <value>
UOC1: ./fork2 3
My PID is 11669, my parent PID is 1
Value: 0      Value: 1      Value: 2
UOC1:

```

Figura 4.4: Ejemplos de ejecución del programa `fork2`.

Para poder comprobar el resultado, vamos a incluir un bucle `while(1)` antes de la finalización, que impida la finalización del proceso en una nueva versión del programa `fork3.c` que vemos en el Código 52.

```

33  /* Wait forever */
34  while(1);

```

Código 52: Código añadido en `fork3.c` para bloquear la finalización del proceso.

Como vemos en la Figura 4.5 tenemos un proceso **bash** con $PID = 1$ que es la **shell** con la que en nuestro caso estamos trabajando, y no tenemos ningún proceso **fork** en ejecución como vemos al ejecutar el comando **ps -aux** con los argumentos adecuados para filtrar la salida. Al ejecutar el programa **fork3** en modo *background* (pues recordemos que se bloquea la finalización del programa) y volver a mostrar los procesos vemos como el proceso **fork3** en ejecución tiene ahora $PID = 11704$, y su proceso padre sigue teniendo $PID = 1$.

```
UOC1: make clean -f fork.Makefile
rm -f fork1 fork2 fork3 fork4 fork5 fork1.o fork2.o fork3.o fork4.o fork5.o
Clean done
UOC1: make fork3 -f fork.Makefile
gcc -c fork3.c -Wall -Wno-unused-variable -I ./
gcc -o fork3 fork3.o -I ./
UOC1: ps -aux | head -2
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  3996  3292 pts/0    Ss   Aug19    0:00 bash
UOC1: ps -aux | grep fork
root    11696  0.0  0.0   3084   820 pts/0    S+   07:23    0:00 grep fork
UOC1: ./fork3 3 &
[1] 11704
UOC1: My PID is 11704, my parent PID is 1
Value: 0      Value: 1      Value: 2

UOC1: ps -aux | grep fork
root    11704 87.5  0.0   2276   744 pts/0    R    07:23    0:06 ./fork3 3
root    11706  0.0  0.0   3084   880 pts/0    S+   07:23    0:00 grep fork
UOC1: █
```

Figura 4.5: Ejemplos de ejecución del programa **fork3**.

Recordemos que hemos dejado un proceso en modo *background* y que deberíamos eliminarlo de nuestro sistema con el comando **kill -9** como vemos en la Figura 4.6.

```
UOC1: kill -9 11704
UOC1:
[1]+  Killed                  ./fork3 3
UOC1: █
```

Figura 4.6: Eliminación del proceso en ejecución del programa **fork3**.

Ahora ya sabemos mostrar el número de un proceso (su **PID**) y el número del **PID** de su proceso padre. Es pues, el momento de bifurcar (crear) un proceso hijo en nuestro ejemplo. Ahora introduciremos la llamada a la función **fork()** y para simplificar la salida, vamos a eliminar la impresión de los valores de nuestro bucle como vemos en el Código 53.

Analicemos el resultado de la ejecución que vemos en la Figura 4.7:

1. La función **fork()** como ya dijimos se ejecuta una vez y devuelve resultado dos veces, una para el proceso padre y otra para el proceso hijo.


```

24  /* Create a new process and print */
25  returnedpid = fork();
26  fprintf(stdout, "fork() returned %d. My PID is %d, my parent PID is %d\n",
    ↪   returnedpid, (int)getpid(), (int)getppid());
27
28  /*
29  top =atoi(argv[1]);
30  for (i=0; i< top;i++) {
31      fprintf(stdout, "value: %d\t",i);
32  }
33  fprintf(stdout, "\n");
34  */

```

Código 53: Modificaciones de código en `fork4.c` para crear un nuevo proceso hijo.

```

UOC1: ps -aux |grep fork
root    11711  0.0  0.0   3084   892 pts/0    S+   07:35   0:00 grep fork
UOC1: make fork4 -f fork.Makefile
gcc -c fork4.c -Wall -Wno-unused-variable -I ./
gcc -o fork4 fork4.o -I ./
UOC1: ./fork4 3 &
[1] 11719
UOC1: fork() returned 11720. My PID is 11719, my parent PID is 1
fork() returned 0. My PID is 11720, my parent PID is 11719

UOC1: ps -aux |grep fork
root     11719 51.0  0.0   2276   676 pts/0    R    07:35   0:02 ./fork4 3
root     11720 51.2  0.0   2276    68 pts/0    R    07:35   0:02 ./fork4 3
root     11722  0.0  0.0   3084   828 pts/0    S+   07:35   0:00 grep fork
UOC1:

```

Figura 4.7: Ejemplo de ejecución del programa `fork4`.

- Una de las dos veces el valor retornado es 0, indicando que estamos en el proceso hijo, que tiene por PID 11720, y por padre un proceso con el PID=11719.
- La segunda de las dos veces el valor retornado es 11720, que es el PID del nuevo proceso creado. Estamos en el proceso padre que tiene PID=11719 y un proceso padre con PID=1 (bash)

Tal y como podemos ver (con el comando `ps -aux` con los argumentos adecuados para filtrar la salida), ambos procesos 11719 y 11720 son la misma ejecución (`./fork4 3`).

Hemos de tener muy claro que cuando hacemos una llamada a la función `fork()` creamos un nuevo proceso (si se realiza con éxito). Por lo tanto si tenéis bucles o iteraciones cuidado con el control del proceso porque podéis acabar generando muchos más procesos de los deseados, y podéis dejar el sistema sin recursos si el número de procesos que

creáis es muy alto. Mucha precaución al trabajar con `fork()`, sobre todo en los primeros ejemplos que hagáis.

Ahora vamos a analizar las variables de los procesos, y vamos a mirar de entender el funcionamiento en concurrencia de ambos procesos, con una última versión de nuestro ejemplo `fork5.c`.

Una vez se haya creado el nuevo proceso, el sistema operativo debe gestionar la conmutación de procesos, a través de dos procesos del sistema el **scheduler** que se encargará de seleccionar el siguiente proceso que deberá de ejecutarse cuando se produzca una conmutación de proceso, y el **dispatcher** que se encargará de conmutar los contextos del proceso saliente y entrante en el sistema para mantener la coherencia en las ejecuciones. Es importante recordar que dicho funcionamiento concede los recursos a un proceso durante un cierto tiempo, después a otro, para volver al primero y así hasta que los procesos se acaben, en una secuencia que no es determinista, y por lo tanto en la que no tenemos control de la ejecución de los procesos, ni de cómo va a poder acabar.

Si las iteraciones que hemos de hacer para imprimir son pocas, probablemente se ejecute completamente uno de los procesos y a continuación el otro. Lo más fácil es que el proceso padre que hace la llamada a la función `fork()` sea el primero en continuar con la ejecución (pero no siempre tiene porqué ser así, y esto debe tenerse en consideración en procesos con sincronización).

En el bucle de impresión vamos a incluir la impresión del proceso que se está ejecutando, con la sentencia `'returnedpid?'P':'C'` como vemos en la nueva versión del programa `fork5.c` en el Código 54. Es un operador ternario para ejecutar una sentencia condicional `if`. Preguntamos el valor de la variable `returnedpid`. Si el valor no es nulo, devolveremos la primera asignación, en nuestro caso el carácter `P` indicando que estamos ejecutando el proceso padre. Si el valor fuera nulo, entonces devolveríamos el carácter `C` para indicar que estamos ejecutando el proceso hijo. Es una manera eficiente de simplificar el código. Además hemos añadido un salto de línea en la impresión cada 10 valores, para hacer más fácil la lectura de la salida cuando se imprimen muchos valores.

Como vemos en la Figura 4.8, en nuestro caso, primero ejecutamos las iteraciones del proceso padre, y después las del proceso hijo. Ambas empiezan en valor 0, pues se copia los valores de las variables antes de la llamada a la función `fork()`. Como vemos ahora los procesos ya no los ejecutamos en modo *background* y por lo tanto ni antes, ni después de la ejecución del comando `./fork5 4` tenemos ningún proceso `fork` ejecutándose en el sistema.

¿Qué pasaría si el número de iteraciones fuera mucho mayor, y tuviéramos que con-

```

13 int main(int argc, char *argv[]) {
14     int pid, ppid;
15     int i, top;
16     int returnedpid;
17
18     /* Check input arguments */
19     if (argc != 2) {
20         fprintf(stderr, "Usage: %s <value>\n", argv[0]);
21         exit(1);
22     }
23
24     top = atoi(argv[1]);
25
26     /* Create a new process and print */
27     returnedpid = fork();
28     fprintf(stdout, "fork() returned %d. My PID is %d, my parent PID is %d\n",
29         ↪ returnedpid, (int)getpid(), (int)getppid());
30
31     for (i = 0; i < top; i++) {
32         fprintf(stdout, "%c: value: %d\t", returnedpid ? 'P' : 'C', i);
33     }
34     if (i%10 == 0) fprintf(stdout, "\n");
35
36     fprintf(stdout, "\n");
37
38     exit(0);

```

Código 54: Modificaciones de código en `fork5.c` para crear un nuevo proceso hijo e imprimir información.

```

UOC1: ps -aux |grep fork
root    11758  0.0  0.0   3084   892 pts/0    S+   07:55   0:00 grep fork
UOC1: make fork5 -f fork.Makefile
gcc -c fork5.c -Wall -Wno-unused-variable -I ./
gcc -o fork5 fork5.o -I ./
UOC1: ./fork5 4
fork() returned 11767. My PID is 11766, my parent PID is 1
P: value: 0    P: value: 1    P: value: 2    P: value: 3
UOC1: fork() returned 0. My PID is 11767, my parent PID is 1
C: value: 0    C: value: 1    C: value: 2    C: value: 3

UOC1: ps -aux |grep fork
root    11769  0.0  0.0   3084   884 pts/0    S+   07:55   0:00 grep fork
UOC1:

```

Figura 4.8: Ejemplo de ejecución del programa `fork5`.

mutar del proceso padre al hijo, y posteriormente nuevamente del hijo al padre?

Primero ejecutaremos el comando `./fork5 100` como vemos en la Figura 4.9 y no tenemos conmutación de contexto. Aumentaremos el número de iteraciones para forzar

cambios de contexto, hasta 2000 iteraciones con el comando `./fork5 2000`.

```

UOC1: ./fork5 100
fork() returned 11773. My PID is 11772, my parent PID is 1
P: value: 0    P: value: 1    P: value: 2    P: value: 3    P: value: 4    P: value: 5    P
: value: 6    P: value: 7    P: value: 8    P: value: 9    P: value: 10   P: value: 11   P
: value: 12   P: value: 13   P: value: 14   P: value: 15   P: value: 16   P: value: 17   P
: value: 18   P: value: 19   P: value: 20   P: value: 21   P: value: 22   P: value: 23   P
: value: 24   P: value: 25   P: value: 26   P: value: 27   P: value: 28   P: value: 29   P
: value: 30   P: value: 31   P: value: 32   P: value: 33   P: value: 34   P: value: 35   P
: value: 36   P: value: 37   P: value: 38   P: value: 39   P: value: 40   P: value: 41   P
: value: 42   P: value: 43   P: value: 44   P: value: 45   P: value: 46   P: value: 47   P
: value: 48   P: value: 49   P: value: 50   P: value: 51   P: value: 52   P: value: 53   P
: value: 54   P: value: 55   P: value: 56   P: value: 57   P: value: 58   P: value: 59   P
: value: 60   P: value: 61   P: value: 62   P: value: 63   P: value: 64   P: value: 65   P
: value: 66   P: value: 67   P: value: 68   P: value: 69   P: value: 70   P: value: 71   P
: value: 72   P: value: 73   P: value: 74   P: value: 75   P: value: 76   P: value: 77   P
: value: 78   P: value: 79   P: value: 80   P: value: 81   P: value: 82   P: value: 83   P
: value: 84   P: value: 85   P: value: 86   P: value: 87   P: value: 88   P: value: 89   P
: value: 90   P: value: 91   P: value: 92   P: value: 93   P: value: 94   P: value: 95   P
: value: 96   P: value: 97   P: value: 98   P: value: 99
UOC1: fork() returned 0. My PID is 11773, my parent PID is 11772
C: value: 0    C: value: 1    C: value: 2    C: value: 3    C: value: 4    C: value: 5    C
: value: 6    C: value: 7    C: value: 8    C: value: 9    C: value: 10   C: value: 11   C
: value: 12   C: value: 13   C: value: 14   C: value: 15   C: value: 16   C: value: 17   C
: value: 18   C: value: 19   C: value: 20   C: value: 21   C: value: 22   C: value: 23   C
: value: 24   C: value: 25   C: value: 26   C: value: 27   C: value: 28   C: value: 29   C
: value: 30   C: value: 31   C: value: 32   C: value: 33   C: value: 34   C: value: 35   C
: value: 36   C: value: 37   C: value: 38   C: value: 39   C: value: 40   C: value: 41   C
: value: 42   C: value: 43   C: value: 44   C: value: 45   C: value: 46   C: value: 47   C
: value: 48   C: value: 49   C: value: 50   C: value: 51   C: value: 52   C: value: 53   C
: value: 54   C: value: 55   C: value: 56   C: value: 57   C: value: 58   C: value: 59   C
: value: 60   C: value: 61   C: value: 62   C: value: 63   C: value: 64   C: value: 65   C
: value: 66   C: value: 67   C: value: 68   C: value: 69   C: value: 70   C: value: 71   C
: value: 72   C: value: 73   C: value: 74   C: value: 75   C: value: 76   C: value: 77   C
: value: 78   C: value: 79   C: value: 80   C: value: 81   C: value: 82   C: value: 83   C
: value: 84   C: value: 85   C: value: 86   C: value: 87   C: value: 88   C: value: 89   C
: value: 90   C: value: 91   C: value: 92   C: value: 93   C: value: 94   C: value: 95   C
: value: 96   C: value: 97   C: value: 98   C: value: 99

```

Figura 4.9: Ejemplo de ejecución del programa `fork5` con 100 iteraciones sin cambio de contexto en nuestro sistema.

Como vemos en la salida del Código 55, el proceso padre se ha ejecutado en primer lugar y sigue ejecutándose hasta cuando llega al valor 1100.

En ese momento el sistema operativo realiza un cambio de contexto y empieza a ejecutar el proceso hijo por primera vez, como vemos en la salida del Código 56. A partir de aquí vemos como se producen varios cambios de contexto padre-hijo e hijo-padre para conseguir que ambos vayan avanzando en su ejecución y consigan terminar.

Al final acabamos con la ejecución (en nuestro caso) de las últimas iteraciones del proceso hijo, como vemos en la salida del Código 57.

```
1 UOC1: ./fork5 2000
2 fork() returned 11886. My PID is 11885, my parent PID is 1
3 P: value: 0
4 P: value: 1      P: value: 2      P: value: 3      P: value: 4      P: value: 5      P:
  ↪ value: 6      P: value: 7      P: value: 8      P: value: 9      P: value: 10
```

Código 55: Ejecución de `./fork5 2000`. Primer contexto proceso padre.

4.2.1. Servidor orientado a conexión concurrente

Estamos pues ya en disposición de ver cómo implementar nuestro servidor de eco orientado a conexión y con capacidad de ejecución concurrente para atender múltiples clientes.

Para ello retomamos los ejemplos del servidor `tcp_server4.c` y del cliente `tcp_client4.c` que tenemos en el Código 36 y Código 37 respectivamente. Para simplificar ahora estos ficheros los hemos renombrado como `server1.c` y `client1.c`.

Además también tenemos dos ficheros de configuración de `make` para el servidor (`server.Makefile`) y el cliente (`client.Makefile`) que tenemos en el Código 58 y en el Código 59 para la generación de los diferentes programas de ejemplo.

```
1 PROGRAM_NAME_1 = server1
2 PROGRAM_OBJS_1 = server1.o
3
4 PROGRAM_NAME_2 = server2
5 PROGRAM_OBJS_2 = server2.o
6
7 PROGRAM_NAME_3 = server3
8 PROGRAM_OBJS_3 = server3.o
9
10 PROGRAM_NAME_4 = server_iteration
11 PROGRAM_OBJS_4 = server_iteration.o
12
13 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
  ↪ $(PROGRAM_NAME_4)
14 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
  ↪ $(PROGRAM_OBJS_4)
15
16 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
17
```

```

113 P: value: 1091 P: value: 1092 P: value: 1093 P: value: 1094 P: value: 1095 P:
    ↳ value: 1096 P: value: 1097 P: value: 1098 P: value: 1099 P: value: 1100
114 fork() returned 0. My PID is 11886, my parent PID is 11885
115 C: value: 0
116 C: value: 1      C: value: 2      C: value: 3      C: value: 4      C: value: 5      C:
    ↳ value: 6      C: value: 7      C: value: 8      C: value: 9      C: value: 10
117 C: value: 11     C: value: 12     C: value: 13     C: value: 14     C: value: 15     C:
    ↳ value: 16     C: value: 17     C: value: 18     C: value: 19     C: value: 20
118 C: value: 21     C: value: 22     C: value: 23     C: value: 24     C: value: 25     C:
    ↳ value: 26     C: value: 27     C: value: 28     C: value: 29     C: value: 30
119 P: value: 1101 P: value: 1102 P: value: 1103 P: value: 1104 P: value: 1105 P:
    ↳ value: 1106 P: value: 1107 P: value: 1108 P: value: 1109 P: value: 1110
120 C: value: 31     C: value: 32     C: value: 33     C: value: 34     C: value: 35     C:
    ↳ value: 36     C: value: 37     C: value: 38     C: value: 39     C: value: 40
121 P: value: 1111 P: value: 1112 P: value: 1113 P: value: 1114 P: value: 1115 P:
    ↳ value: 1116 P: value: 1117 P: value: 1118 P: value: 1119 P: value: 1120
122 C: value: 41     C: value: 42     C: value: 43     C: value: 44     C: value: 45     C:
    ↳ value: 46     C: value: 47     C: value: 48     C: value: 49     C: value: 50
123 P: value: 1121 P: value: 1122 P: value: 1123 P: value: 1124 P: value: 1125 P:
    ↳ value: 1126 P: value: 1127 P: value: 1128 P: value: 1129 P: value: 1130

```

Código 56: Ejecución de ./fork5 2000. Primer cambio de contexto entre el proceso padre y el proceso hijo. Y algunos cambios de contexto sucesivos.

```

18 all: $(PROGRAM_NAME_ALL)
19 @echo "Finished!"
20
21 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
22 gcc -o $@ $^ -I ./
23
24 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
25 gcc -o $@ $^ -I ./
26
27 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
28 gcc -o $@ $^ -I ./
29
30 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
31 gcc -o $@ $^ -I ./
32
33 %.o: %.c
34 gcc -c $< -Wall -Wno-unused-variable -I ./
35
36 clean:

```

```
400 C: value: 1941 C: value: 1942 C: value: 1943 C: value: 1944 C: value: 1945 C:
    ↳ value: 1946 C: value: 1947 C: value: 1948 C: value: 1949 C: value: 1950
401 C: value: 1951 C: value: 1952 C: value: 1953 C: value: 1954 C: value: 1955 C:
    ↳ value: 1956 C: value: 1957 C: value: 1958 C: value: 1959 C: value: 1960
402 C: value: 1961 C: value: 1962 C: value: 1963 C: value: 1964 C: value: 1965 C:
    ↳ value: 1966 C: value: 1967 C: value: 1968 C: value: 1969 C: value: 1970
403 C: value: 1971 C: value: 1972 C: value: 1973 C: value: 1974 C: value: 1975 C:
    ↳ value: 1976 C: value: 1977 C: value: 1978 C: value: 1979 C: value: 1980
404 C: value: 1981 C: value: 1982 C: value: 1983 C: value: 1984 C: value: 1985 C:
    ↳ value: 1986 C: value: 1987 C: value: 1988 C: value: 1989 C: value: 1990
405 C: value: 1991 C: value: 1992 C: value: 1993 C: value: 1994 C: value: 1995 C:
    ↳ value: 1996 C: value: 1997 C: value: 1998 C: value: 1999
406
407 UOC1:
```

Código 57: Ejecución de `./fork5 2000`. Último contexto de ejecución con el proceso hijo.

```
37 rm -f $(REBUIDABLES)
38 @echo "Clean done"
```

Código 58: Fichero de configuración `server.Makefile`.

```
1 PROGRAM_NAME_1 = client1
2 PROGRAM_OBJS_1 = client1.o
3
4 PROGRAM_NAME_2 = client2
5 PROGRAM_OBJS_2 = client2.o
6
7 PROGRAM_NAME_3 = client3
8 PROGRAM_OBJS_3 = client3.o
9
10 PROGRAM_NAME_4 = client_iteration
11 PROGRAM_OBJS_4 = client_iteration.o
12
13 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
    ↳ $(PROGRAM_NAME_4)
14 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
    ↳ $(PROGRAM_OBJS_4)
15
16 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
```

```
17
18 all: $(PROGRAM_NAME_ALL)
19     @echo "Finished!"
20
21 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
22     gcc -o $@ $^ -I ./
23
24 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
25     gcc -o $@ $^ -I ./
26
27 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
28     gcc -o $@ $^ -I ./
29
30 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
31     gcc -o $@ $^ -I ./
32
33 %.o: %.c
34     gcc -c $< -Wall -Wno-unused-variable -I ./
35
36 clean:
37     rm -f $(REBUIDABLES)
38     @echo "Clean done"
```

Código 59: Fichero de configuración `client.Makefile`.

Como vemos, el código del Cliente no se modifica puesto que la concurrencia la añadimos en el lado servidor, y no en el lado cliente. A modo de recordatorio, en el Código 60 vemos la parte de código para la petición y recepción de eco de `client1.c`.

```
40 write(sock,argv[2],strlen(argv[2])+1);
41 fprintf(stdout, " sent \n");
42 read(sock,buffer, BUFFSIZE);
43 fprintf(stdout, " %s ...done \n",buffer);
```

Código 60: Parte del código del `client1.c` donde vemos la petición y recepción de eco al servidor.

En el lado servidor, añadiremos la gestión de la bifurcación de procesos al recibir la petición de conexión por parte de un nuevo cliente, como vemos en el Código 61.


```
52 /* loop */
53 while (1) {
54     unsigned int clientlen = sizeof(echoclient);
55     /* we wait for a connection from a client */
56     if ((clientsock =
57         accept(serversock, (struct sockaddr *) &echoclient,&clientlen)) < 0) {
58         err_sys("error accept");
59     }
60     fprintf(stdout, "Client: %s\n",inet_ntoa(echoclient.sin_addr));
61     if (fork()==0) {
62         //child process
63         close (serversock);
64         HandleClient(clientsock);
65         exit(0);
66     }
67     else {
68         //parent process
69         close(clientsock);
70     }
71 }
```

Código 61: Parte del código del `server1.c` donde vemos la gestión de la bifurcación de procesos al recibir la petición de conexión.

Al ejecutarlo el servidor estará siempre pendiente de recibir las peticiones de conexión pero ahora para cada nueva petición tendremos un nuevo proceso para atenderla.

Así pues, generamos los dos ficheros ejecutables como vemos en la Figura 4.10.

```
UOC1: make server1 -f server.Makefile
gcc -c server1.c -Wall -Wno-unused-variable -I ./
gcc -o server1 server1.o -I ./
UOC1: make client1 -f client.Makefile
gcc -c client1.c -Wall -Wno-unused-variable -I ./
gcc -o client1 client1.o -I ./
UOC1: █
```

Figura 4.10: Generación de los ficheros ejecutables `server1` y `client1`.

Para comprobar su funcionamiento ejecutaremos en modo *background* el servidor esperando peticiones de conexión en el puerto 6000 `./server1 6000 &`, y dos instancias del cliente, `./client1 127.0.0.1 hello 6000 &` y `./client1 127.0.0.1 done 6000 &`, como vemos en la Figura 4.11.

Como vemos los mensajes correspondientes al funcionamiento de la petición/respuesta. Al finalizar los dos procesos de cliente, vemos que en el lado servidor, tenemos dos procesos con PID 11946 y 11948 en estado *zombie* tal (al no tener ninguna gestión de sus

señales de finalización por parte del proceso padre) y como muestra el mensaje correspondiente `[server1] <defunct>` al ejecutar el comando `ps -aux` con el filtrado adecuado.

```
UOC1: ps -aux|grep server1
root      11943  0.0  0.0   3084   892 pts/0    S+   10:35   0:00 grep server1
UOC1: ./server1 6000 &
[1] 11944
UOC1: ./client1 127.0.0.1 hello 6000 &
[2] 11945
UOC1: Client: 127.0.0.1
sent
message from client: hello
hello ...done

[2]+ Done                ./client1 127.0.0.1 hello 6000
UOC1: ./client1 127.0.0.1 done 6000 &
[2] 11947
UOC1: Client: 127.0.0.1
sent
message from client: done
done ...done

[2]+ Done                ./client1 127.0.0.1 done 6000
UOC1: ps -aux|grep server1
root      11944  0.0  0.0   2276   684 pts/0    S    10:35   0:00 ./server1 6000
root      11946  0.0  0.0      0      0 pts/0    Z    10:36   0:00 [server1] <defunct>
root      11948  0.0  0.0      0      0 pts/0    Z    10:36   0:00 [server1] <defunct>
root      11950  0.0  0.0   3084   892 pts/0    S+   10:36   0:00 grep server1
UOC1: █
```

Figura 4.11: Ejecución del servidor `server1` y de dos instancias del cliente `client1`.

Vemos en la Figura 4.12 como al finalizar el proceso padre con el comando `kill -9` las dos instancias de los procesos hijos creadas para atender las dos peticiones de clientes que hemos recibido también han finalizado (ahora ya sí completamente) como vemos con la ejecución del comando `ps -aux` con las opciones adecuadas para un filtrado de la salida.

```
UOC1: kill -9 11944
UOC1:
[1]+ Killed                ./server1 6000
UOC1: ps -aux|grep server1
root      11952  0.0  0.0   3084   824 pts/0    S+   10:38   0:00 grep server1
UOC1: █
```

Figura 4.12: Eliminación del servidor `server1` y de rebote de los dos procesos hijos creados que se quedaron en estado *zombie*.

Vamos a modificar los programas del cliente y del servidor. En el lado cliente, con el programa `client2.c`, vamos a eliminar la impresión de mensajes (para facilitar la lectura en concurrencia de los clientes y servidores en un mismo terminal), como vemos en el Código 62.

En el lado servidor, con el programa `server2.c`, vamos a imprimir los PID en el proceso padre, y en el proceso hijo, para ver cómo los vamos creando, como vemos en el Código 63.

```
40 write(sock,argv[2],strlen(argv[2])+1);
41 //fprintf(stdout, " sent \n");
42 read(sock,buffer, BUFFSIZE);
43 //fprintf(stdout, " %s ...done \n",buffer);
```

Código 62: Parte del código del `client2.c` donde vemos la petición y recepción de eco al servidor, sin impresión de mensajes.

Generamos los dos ficheros ejecutables como vemos en la Figura 4.13.

```
UOC1: make server2 -f server.Makefile
gcc -c server2.c -Wall -Wno-unused-variable -I ./
gcc -o server2 server2.o -I ./
UOC1: make client2 -f client.Makefile
gcc -c client2.c -Wall -Wno-unused-variable -I ./
gcc -o client2 client2.o -I ./
UOC1: █
```

Figura 4.13: Generación de los ficheros ejecutables `server2` y `client2`.

Ejecutaremos en modo *background* el servidor esperando peticiones de conexión en el puerto 6000 `./server2 6000 &`, y tres instancias del cliente, `./client2 127.0.0.1 HELLO 6000 &`, `./client2 127.0.0.1 done 6000 &` y `./client2 127.0.0.1 bye 6000 &`, como vemos en la Figura 4.14. Vemos los mensajes correspondientes al funcionamiento de la petición/respuesta. Vemos como se han creado los tres nuevos procesos (hijos) para atender cada nueva petición de cliente con PID 11981, 11983 y 11985 respectivamente.

Para entender la utilidad de la bifurcación vamos a hacer que los procesos que atienden las conexiones después de devolver el eco, y por lo tanto permitiendo que el proceso cliente finalice, se queden en un bucle infinito. El objetivo es ver cómo podemos seguir atendiendo a nuevas peticiones, porque para cada nueva petición tendremos un proceso para atenderla, independientemente del tiempo que se tarde en atenderla. Para ello hemos implementado una versión de servidor `server3.c` en la que hemos añadido en el proceso hijo (el que gestiona la conexión) un bucle infinito al final, para que el proceso no finalice nunca como vemos en el Código 64.

El programa cliente `client3.c` es el mismo que el anterior `client2.c`.

Generamos los dos ficheros ejecutables como vemos en la Figura 4.15.

Ejecutaremos en modo *background* el servidor esperando peticiones de conexión en el puerto 6000 `./server3 6000 &`, y tres instancias del cliente, `./client3 127.0.0.1 HELLO 6000 &`, `./client3 127.0.0.1 done 6000 &` y `./client3 127.0.0.1 bye 6000 &`, como vemos en la Figura 4.16. Vemos los mensajes correspondientes al funcionamiento

```
53 /* loop */
54 while (1) {
55     unsigned int clientlen = sizeof(echoclient);
56     /* we wait for a connection from a client */
57     if ((clientsock =
58         accept(serversock, (struct sockaddr *) &echoclient,&clientlen)) < 0) {
59         err_sys("error accept");
60     }
61     fprintf(stdout, "Client: %s\n",inet_ntoa(echoclient.sin_addr));
62     if ((returnedpid=fork())==0) {
63         //child process
64         fprintf(stdout,"%c, fork() returned %d. My PID is %d, my parent PID is %d\n",
65             ↪ returnedpid?'P':'C',returnedpid, (int)getpid(), (int)getppid());
66         close (serversock);
67         HandleClient(clientsock);
68         exit(0);
69     }
70     //parent process
71     fprintf(stdout,"%c, fork() returned %d. My PID is %d, my parent PID is %d\n",
72         ↪ returnedpid?'P':'C',returnedpid, (int)getpid(), (int)getppid());
73     close(clientsock);
74 }
```

Código 63: Parte del código del `server2.c` donde vemos la impresión de PID de proceso padre e hijo.

de la petición/respuesta. Vemos como se han creado los tres nuevos procesos (hijos) para atender cada nueva petición de cliente con PID 12045, 12047 y 12049 respectivamente. Al finalizar la recepción de los mensajes, vemos como todos los procesos siguen en ejecución, y nuestro servidor de eco puede atender diversas peticiones concurrentes por parte de clientes sin quedarse bloqueado.

Para finalizar con nuestros ejemplos de bifurcación vamos a implementar un bucle de tres lecturas/escrituras en ambos programas, cliente y servidor.

En el lado servidor, en el programa `server_iteration.c` hemos introducido en la función que gestiona las conexiones `HandleClient()` un bucle para gestionar tres mensajes por cliente como vemos en el Código 65.

En el lado cliente, en el programa `client_iteration.c` hemos introducido un bucle para gestionar tres mensajes por cliente como vemos en el Código 66.

Generamos los dos ficheros ejecutables como vemos en la Figura 4.17.

Ejecutamos en primer lugar el servidor en modo *background* esperando peticiones de

```

UOC1: ps -aux|grep server2
root      11978  0.0  0.0   3084   824 pts/0    S+   10:53   0:00 grep server2
UOC1: ./server2 6000 &
[1] 11979
UOC1: ./client2 127.0.0.1 HELLO 6000 &
[2] 11980
UOC1: Client: 127.0.0.1
P, fork() returned 11981. My PID is 11979, my parent PID is 1
C, fork() returned 0. My PID is 11981, my parent PID is 11979
message from client: HELLO

[2]+ Done                  ./client2 127.0.0.1 HELLO 6000
UOC1: ./client2 127.0.0.1 done 6000 &
[2] 11982
UOC1: Client: 127.0.0.1
P, fork() returned 11983. My PID is 11979, my parent PID is 1
C, fork() returned 0. My PID is 11983, my parent PID is 11979
message from client: done

[2]+ Done                  ./client2 127.0.0.1 done 6000
UOC1: ./client2 127.0.0.1 bye 6000 &
[2] 11984
UOC1: Client: 127.0.0.1
P, fork() returned 11985. My PID is 11979, my parent PID is 1
C, fork() returned 0. My PID is 11985, my parent PID is 11979
message from client: bye

[2]+ Done                  ./client2 127.0.0.1 bye 6000
UOC1: ps -aux|grep server2
root      11979  0.0  0.0   2276   744 pts/0    S    10:54   0:00 ./server2 6000
root      11981  0.0  0.0      0      0 pts/0    Z    10:54   0:00 [server2] <defunct>
root      11983  0.0  0.0      0      0 pts/0    Z    10:54   0:00 [server2] <defunct>
root      11985  0.0  0.0      0      0 pts/0    Z    10:54   0:00 [server2] <defunct>
root      11987  0.0  0.0   3084   896 pts/0    S+   10:54   0:00 grep server2
UOC1:

```

Figura 4.14: Ejecución del servidor `server2` y de tres instancias del cliente `client2`.

```

UOC1: make server3 -f server.Makefile
gcc -c server3.c -Wall -Wno-unused-variable -I ./
gcc -o server3 server3.o -I ./
UOC1: make client3 -f client.Makefile
gcc -c client3.c -Wall -Wno-unused-variable -I ./
gcc -o client3 client3.o -I ./
UOC1:

```

Figura 4.15: Generación de los ficheros ejecutables `server3` y `client3`.

clientes en el puerto 6000 con el comando `./server_iteration 6000 &`. A continuación lanzamos dos peticiones por parte de dos instancias del cliente, también en modo *background*, con los comandos `./client_iteration 127.0.0.1 6000 &` y `client_iteration 192.168.122.140 6000 &`, como vemos en la Figura 4.18, y vemos que los dos clientes están a la espera que les introduzcamos el texto a enviar al servidor.

Para poder introducir un mensaje, recuperamos el control del primer cliente, volviendo el proceso a modo *foreground* con el comando `fg 2`, e introducimos dos mensajes `1_message_1` y `1_message_2`, que vemos que son recepcionados por el servidor, con los mensajes `ECHO: 1_message_1` y `ECHO: 1_message_2` como vemos en la Figura 4.19. Al

```

62  if ((returnedpid=fork())==0) {
63      //child process
64      fprintf(stdout,"%c, fork() returned %d. My PID is %d, my parent PID is %d\n",
        ↪ returnedpid?'P':'C',returnedpid, (int)getpid(), (int)getppid());
65      close (serversock);
66      HandleClient(clientsock);
67      //LOOP
68      while(1);
69      exit(0);
70  }

```

Código 64: Parte del código del `server3.c` donde vemos el bucle infinito para que el proceso hijo no finalice nunca.

```

UOC1: ps -aux | grep server; ps -aux | grep client
root    12040  0.0  0.0  3084  892 pts/0    S+   11:45   0:00  grep server
root    12042  0.0  0.0  3084  884 pts/0    S+   11:45   0:00  grep client
UOC1: ./server3 6000 &
[1] 12043
UOC1: ./client3 127.0.0.1 HELLO 6000 &
[2] 12044
UOC1: Client: 127.0.0.1
P, fork() returned 12045. My PID is 12043, my parent PID is 1
C, fork() returned 0. My PID is 12045, my parent PID is 12043
message from client: HELLO
UOC1: ./client3 127.0.0.1 done 6000 &
[3] 12046
[2]  Done                  ./client3 127.0.0.1 HELLO 6000
UOC1: Client: 127.0.0.1
P, fork() returned 12047. My PID is 12043, my parent PID is 1
C, fork() returned 0. My PID is 12047, my parent PID is 12043
message from client: done
UOC1: ./client3 127.0.0.1 bye 6000 &
[4] 12048
[3]  Done                  ./client3 127.0.0.1 done 6000
UOC1: Client: 127.0.0.1
P, fork() returned 12049. My PID is 12043, my parent PID is 1
C, fork() returned 0. My PID is 12049, my parent PID is 12043
message from client: bye
UOC1: ps -aux | grep server; ps -aux | grep client
root    12043  0.0  0.0  2276  680 pts/0    S    11:45   0:00  ./server3 6000
root    12045 61.8  0.0  2276   80 pts/0    R    11:45   0:14  ./server3 6000
root    12047 42.4  0.0  2276   80 pts/0    R    11:45   0:05  ./server3 6000
root    12049 34.6  0.0  2276   80 pts/0    R    11:46   0:02  ./server3 6000
root    12051  0.0  0.0  3084  880 pts/0    S+   11:46   0:00  grep server
[4]+  Done                  ./client3 127.0.0.1 bye 6000
root    12053  0.0  0.0  3084  888 pts/0    S+   11:46   0:00  grep client
UOC1:

```

Figura 4.16: Ejecución del servidor `server3` y de tres instancias del cliente `client3`.

finalizar volvemos a enviar el proceso al modo *background* con el comando `bg`.

Para poder introducir un mensaje, recuperamos el control del segundo cliente, volviendo el proceso a modo *foreground* con el comando `fg` 3, e introducimos un mensaje `2_message_1`, que vemos que es recepcionado por el servidor, con el mensaje `ECHO`:

```

17 void HandleClient(int sock) {
18     char buffer[BUFSIZE];
19     int received = -1;
20     int contador;
21
22     for (contador=0; contador<3; contador++) {
23         /* new message from client */
24         buffer[0]='\0';
25         if ((received = recv(sock, buffer, BUFSIZE, 0)) < 0) {
26             err_sys("error on reading message");
27         }
28         buffer[received]='\0';
29         /* send back echo */
30         // while (received > 0) {
31             /* send back echo */
32             if (send(sock, buffer, received, 0) != received) {
33                 err_sys("error writign echo message");
34             }
35             /* quedan datos por recibir */
36             // if ((received = recv(sock, buffer, BUFSIZE, 0)) < 0) {
37             //     err_sys("error lectura adicional");
38             // }
39         // }
40         printf("(%d) text : %s", contador, buffer);
41     }
42     close(sock);
43 }

```

Código 65: Parte del código del `server_iteration.c` donde vemos la gestión de tres mensajes por cliente en la función `HandleClient()`.

```

UOC1: make server_iteration -f server.Makefile
gcc -c server_iteration.c -Wall -Wno-unused-variable -I ./
gcc -o server_iteration server_iteration.o -I ./
UOC1: make client_iteration -f client.Makefile
gcc -c client_iteration.c -Wall -Wno-unused-variable -I ./
gcc -o client_iteration client_iteration.o -I ./
UOC1:

```

Figura 4.17: Generación de los ficheros ejecutables `server_iteration` y `client_iteration`.

2_message_1. Al finalizar volvemos a enviar el proceso al modo *background* con el comando `bg`. Para poder introducir un mensaje, recuperamos el control del primer cliente, volviendo el proceso a modo *foreground* con el comando `fg` 2, e introducimos el tercer mensaje 1_message_3, que vemos que es recepcionado por el servidor, con el mensaje `ECHO: 1_message_3`. Al finalizar volvemos a enviar el proceso al modo *background* con

```

44 for (contador=0; contador<3; contador++){
45     buffer[0] = '\0';          /* \0 */
46     printf("(%d) which word?\n", contador);
47     fgets(buffer,BUFSIZE-1,stdin);
48     echolen = strlen(buffer);
49     if (send(sock, buffer, echolen, 0) != echolen) {
50         err_sys("error writing");
51     }
52     /* echo */
53     fprintf(stdout, "ECHO: ");
54     buffer[0] = '\0';          /* \0 */
55     //while (received < echolen) {
56     bytes = 0;
57     if ((bytes = recv(sock, buffer, BUFSIZE-1, 0)) < 1) {
58         err_sys("error reading");
59     }
60     buffer[bytes] = '\0';      /* \0 */
61     fprintf(stdout, "%s", buffer);
62     //}
63     fprintf(stdout, "\n");
64 }
65 }

```

Código 66: Parte del código del `client_iteration.c` donde vemos la gestión de tres mensajes.

```

UOC1: ./server_iteration 6000 &
[1] 12140
UOC1: PARENT PROCESS: waiting for ACCEPT

UOC1: ./client_iteration 127.0.0.1 6000 &
[2] 12141
UOC1: PARENT: I have already forked a new child process
PARENT PROCESS: waiting for ACCEPT
(0) which word?
Client: 127.0.0.1

[2]+ Stopped                  ./client_iteration 127.0.0.1 6000
UOC1: ./client_iteration 192.168.122.140 6000 &
[3] 12143
UOC1: PARENT: I have already forked a new child process
PARENT PROCESS: waiting for ACCEPT
(0) which word?
Client: 192.168.122.140

[3]+ Stopped                  ./client_iteration 192.168.122.140 6000
UOC1:

```

Figura 4.18: Ejecución del servidor `server_iteration` y dos instancias del cliente `client_iteration`.


```
UOC1: fg 2
./client_iteration 127.0.0.1 6000
1_message_1
(0) text : 1_message_1
ECHO: 1_message_1

(1) which word?
1_message_2
(1) text : 1_message_2
ECHO: 1_message_2

(2) which word?
^Z
[2]+ Stopped                  ./client_iteration 127.0.0.1 6000
UOC1: bg
[2]+ ./client_iteration 127.0.0.1 6000 &
UOC1: █
```

Figura 4.19: Introducción y eco de los dos primeros mensajes de la primera instancia del cliente `client_iteration`.

el comando `bg`, como vemos en la Figura 4.20.

```
UOC1: fg 3
./client_iteration 192.168.122.140 6000
2_message_1
(0) text : 2_message_1
ECHO: 2_message_1

(1) which word?
^Z
[2]- Stopped                  ./client_iteration 127.0.0.1 6000

[3]+ Stopped                  ./client_iteration 192.168.122.140 6000
UOC1: bg
[3]+ ./client_iteration 192.168.122.140 6000 &
UOC1: fg 2
./client_iteration 127.0.0.1 6000
1_message_3
(2) text : 1_message_3
end of child process: Success
ECHO: 1_message_3

[3]- Stopped                  ./client_iteration 192.168.122.140 6000
UOC1: bg
[3]+ ./client_iteration 192.168.122.140 6000 &
UOC1: █
```

Figura 4.20: Intercambio de mensajes por las dos instancias de cliente `client_iteration`.

Para finalizar, recuperamos el control del segundo cliente, volviendo el proceso a modo *foreground* con el comando `fg 3`, e introducimos los dos mensajes que faltan `2_message_2` y `2_message_3`, que vemos que son recepcionados por el servidor, con los mensajes `ECHO: 2_message_2` y `ECHO: 2_message_3`. Al finalizar volvemos a enviar el proceso al modo *background* con el comando `bg`, como vemos en la Figura 4.21.

Al final, comprobamos que ya hemos finalizado (como era de esperar al enviar los tres mensajes) las dos instancias del cliente `client_iteration` y nos queda el proceso

```
UOC1: fg 3
./client_iteration 192.168.122.140 6000
2_message_2
(1) text : 2_message_2
ECHO: 2_message_2

(2) which word?
2_message_3
(2) text : 2_message_3
end of child process: Success
ECHO: 2_message_3
UOC1: █
```

Figura 4.21: Últimos mensajes de la segunda instancia del cliente `client_iteration`.

servidor en ejecución `server_iteration` a la espera de nuevas conexiones por parte de otras instancias del cliente, y los dos procesos (hijo) que atendieron a nuestros dos clientes, que como ya explicamos se quedan en estado *zombie* a la espera de la finalización del proceso padre, como vemos en la Figura 4.22.

```
UOC1: ps -aux | grep server; ps -aux | grep client
root    12140  0.0  0.0  2276   740 pts/0    S   12:32   0:00 ./server_iteration 6000
root    12142  0.0  0.0      0      0 pts/0    Z   12:33   0:00 [server_iteratio] <defunct>
root    12144  0.0  0.0      0      0 pts/0    Z   12:33   0:00 [server_iteratio] <defunct>
root    12146  0.0  0.0  3084   884 pts/0    S+  12:37   0:00 grep server
root    12148  0.0  0.0  3084   888 pts/0    S+  12:37   0:00 grep client
UOC1: █
```

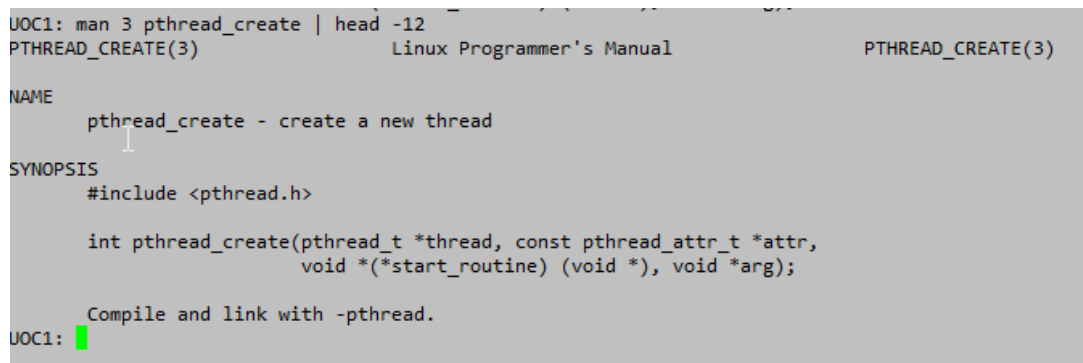
Figura 4.22: Generación de los ficheros ejecutables `server_iteration` y `client_iteration`.

4.3. *Threads*

En lugar de trabajar con nuevos procesos cada vez que hemos de atender a un cliente podemos hacerlo con *threads* o hilos de ejecución. Los *threads* son un mecanismo alternativo a la bifurcación de procesos para poder tener varios hilos de ejecución de un programa. La principal ventaja de utilizar *threads* es que son más óptimos en cuanto al consumo de recursos, por lo que son una mejor alternativa siempre y cuando se puedan utilizar. Como veremos a lo largo de la sección la optimización se fundamenta en gran medida porque ambos hilos de ejecución comparten más información de la que hacen los procesos y por lo tanto cuando se bifurca procesos necesitamos por un lado más tiempo para la creación del nuevo proceso, y necesitaremos también más recursos del sistema para guardar la información del estado y del proceso pues hay menos información compartida de la que nos encontramos en hilos de ejecución: si compartimos información la gran ventaja es que sólo la tendremos almacenada una vez en nuestro sistema y ello supone un importante ahorro de recursos.

Así como para bifurcar/crear procesos utilizábamos la función `fork()` para los hilos de ejecución, utilizaremos la función `pthread_create()`

Para la creación de un nuevo hilo de ejecución deberemos usar la función `pthread_create()` de la que vemos la ayuda en la Figura 4.23. Sobre todo recordar la opción adicional `-pthread` que debe indicarse en el momento de la compilación, como veremos posteriormente.



```
UOC1: man 3 pthread_create | head -12
PTHREAD_CREATE(3)          Linux Programmer's Manual          PTHREAD_CREATE(3)

NAME
  pthread_create - create a new thread

SYNOPSIS
  #include <pthread.h>

  int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start_routine) (void *), void *arg);

  Compile and link with -pthread.
UOC1: █
```

Figura 4.23: Manual de ayuda de la función `pthread_create` donde vemos la importancia de utilizar la opción `-pthread`.

Para nuestro servidor de eco, en la primera versión `thread_server1.c`, vamos a sustituir la función de gestión de la conexión por una función de un *thread* de ejecución que se encargará sólo de la conexión. Después de aceptar la conexión, en lugar de hacer una llamada a `fork()` para crear un nuevo proceso vamos a crear un nuevo hilo de ejecución con una llamada a la función `pthread_create()`, como vemos en el Código 67.

Con la llamada al sistema `pthread_create()` vamos a crear el nuevo hilo de ejecución que tendrá asociado un nuevo identificador de *thread* (`handleThreadId`), llamará la función `handleThread()` para ser ejecutada por el *thread*, y tendrá como argumento de entrada el identificador del *socket* que tendremos establecido para la conexión con el cliente.

Podemos imprimir en el lado servidor el identificador del *thread* (por si fuera de utilidad), con `'fprintf(stdout, "Thread ID for handler %lu\n ", handleThreadId);'` por ejemplo como hacemos en nuestro ejemplo.

La función que en caso de producirse una petición de conexión gestionará la conexión por cada nuevo cliente (para un funcionamiento concurrente), será la que hemos asociado al *thread* con identificador `handleThreadId` `handleThread` que hemos asociado anteriormente.

Como vemos en el Código 68, la función de gestión de la conexión es similar a la que

```
65  /* loop */
66  while (1) {
67      unsigned int clientlen = sizeof(echoclient);
68      /* we wait for a connection from a client */
69      if ((clientsock =
70          accept(serversock, (struct sockaddr *) &echoclient, &clientlen)) < 0) {
71          err_sys("error accept");
72      }
73      fprintf(stdout, "Client: %s\n", inet_ntoa(echoclient.sin_addr));
74      pthread_create(&handleThreadId, NULL, handleThread, (void *)&clientsock);
75      fprintf(stdout, "Thread ID for handler %lu\n", handleThreadId);
76      // pthread_join(handleThreadId, NULL);
77
78      /*
79      if (fork()==0) {
80          //child process
81          close (serversock);
82          HandleClient(clientsock);
83          exit(0);
84      }
85      else {
86          //parent process
87          close(clientsock);
88      }
89      */
90  }
```

Código 67: Parte del código del `thread_server1.c` donde vemos la creación del nuevo hilo de ejecución.

utilizamos con procesos bifurcados, con el cambio asociado a la lectura del parámetro de entrada que nos permitirá pasar el identificador del *socket* a la función.

También podemos imprimir en la propia función de gestión del thread el identificador, con el comando `'fprintf(stdout, "Thread ID in handler %lu\n ", pthread_self());'` con la llamada `pthread_self()`.

La parte de cliente mantiene el código de nuestro cliente con un mensaje, que ya utilizamos anteriormente. La hemos renombrado a `thread_client1.c`.

Para la generación de los ficheros ejecutables, utilizaremos el fichero de configuración de `make thread.Makefile` que vemos en el Código 69.

```
1 PROGRAM_NAME_1 = thread_client1
2 PROGRAM_OBJS_1 = thread_client1.o
3
```

```
22 void *handleThread(void *vargp)
23 {
24     int *mysock = (int *)vargp;
25     char buffer[BUFSIZE];
26     int received = -1;
27
28     printf("Executing handleThread with socket descriptor ID: %d\n", *mysock);
29     fprintf(stdout, "Thread ID in handler %lu\n", pthread_self());
30
31     read(*mysock,&buffer[0],BUFSIZE);
32     printf("message from client: %s\n", buffer);
33     write(*mysock,buffer,strlen(buffer)+1);
34     close(*mysock);
35     return((void*)NULL);
36 }
```

Código 68: Parte del código del `thread_server1.c` donde vemos la función `handleThread()` para atender la conexión con cada nuevo cliente.

```
4 PROGRAM_NAME_2 = thread_server1
5 PROGRAM_OBJS_2 = thread_server1.o
6
7 PROGRAM_NAME_3 = thread_client2
8 PROGRAM_OBJS_3 = thread_client2.o
9
10 PROGRAM_NAME_4 = thread_server2
11 PROGRAM_OBJS_4 = thread_server2.o
12
13 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
14 ↪ $(PROGRAM_NAME_4)
15 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
16 ↪ $(PROGRAM_OBJS_4)
17
18 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
19
20 all: $(PROGRAM_NAME_ALL)
21     @echo "Finished!"
22
23 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
24     gcc -o $$@ $$^ -pthread -I ./
```

```

25 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
26 gcc -o $@ $^ -pthread -I ./
27
28 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
29 gcc -o $@ $^ -pthread -I ./
30
31 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
32 gcc -o $@ $^ -pthread -I ./
33
34 %.o: %.c
35 gcc -c $< -pthread -Wall -Wno-unused-variable -I ./
36
37 clean:
38 rm -f $(REBUILDABLES)
39 @echo "Clean done"

```

Código 69: Fichero de configuración `thread.Makefile`.

Procedemos a generar los dos ejecutables, del cliente `thread_client1` y del servidor `thread_server1` con el fichero de configuración `thread.Makefile` como vemos en la Figura 4.24.

```

UOC1: make thread_server1 -f thread.Makefile
gcc -c thread_server1.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_server1 thread_server1.o -pthread -I ./
UOC1: make thread_client1 -f thread.Makefile
gcc -c thread_client1.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_client1 thread_client1.o -pthread -I ./
UOC1:

```

Figura 4.24: Generación de los ejecutables `thread_server1` y `thread_client1` con el fichero de configuración `thread.Makefile`.

Podemos pues, ahora ejecutar una instancia del servidor (como en muchas otras ocasiones en modo *background* para tener control del terminal) con el comando `./thread_server1 6000 &` esperando peticiones de conexión al servidor de eco en el puerto 6000. A continuación ejecutaremos dos instancias del cliente `thread_client1`. La primera a la dirección local 127.0.0.1 con el comando `./thread_client1 127.0.0.1 message1 6000 &` y la segunda a la dirección del terminal con el comando `./thread_client1 192.168.122.140 message2 6000 &`. Como vemos en la Figura 4.25 el servidor atiende ambas conexiones.

Para mostrar el funcionamiento concurrente del servidor, vamos a introducir en la rutina de gestión de la interrupción, la que ejecutará cada nuevo hilo de ejecución un

```
UOC1: ./thread_server1 6000 &
[1] 17280
UOC1: ./thread_client1 127.0.0.1 message1 6000 &
[2] 17281
UOC1: Client: 127.0.0.1
Thread ID for handler 140614833612544
sent
Executing handleThread with socket descriptor ID: 4
Thread ID in handler 140614833612544
message from client: message1
message1 ...done

[2]+ Done ./thread_client1 127.0.0.1 message1 6000
UOC1: ./thread_client1 192.168.122.140 message2 6000 &
[2] 17283
UOC1: Client: 192.168.122.140
Thread ID for handler 140614825219840
sent
Executing handleThread with socket descriptor ID: 5
Thread ID in handler 140614825219840
message from client: message2
message2 ...done

[2]+ Done ./thread_client1 192.168.122.140 message2 6000
UOC1: █
```

Figura 4.25: Ejecución del servidor `thread_server1` y de dos instancias del cliente `thread_client1`.

bucle infinito para impedir salir de la función.

Procedemos a generar los dos ejecutables, del cliente `./thread_client2` y del servidor `thread_server2` con el fichero de configuración `thread.Makefile` como vemos en la Figura 4.26.

```
UOC1: make thread_server2 -f thread.Makefile
gcc -c thread_server2.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_server2 thread_server2.o -pthread -I ./
[1]+ Killed ./thread_server1 6000
UOC1: make thread_client2 -f thread.Makefile
gcc -c thread_client2.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_client2 thread_client2.o -pthread -I ./
UOC1: █
```

Figura 4.26: Generación de los ejecutables `thread_server2` y `thread_client2` con el fichero de configuración `thread.Makefile`.

Podemos pues, ahora ejecutar una instancia del servidor (como en muchas otras ocasiones en modo *background* para tener control del terminal) con el comando `./thread_server2 6000 &` esperando peticiones de conexión al servidor de eco en el puerto 6000. A continuación ejecutaremos dos instancias del cliente `thread_client2`. La primera a la dirección local 127.0.0.1 con el comando `./thread_client2 127.0.0.1 message1 6000 &` y la segunda a la dirección del terminal con el comando `./thread_client2 192.168.122.140 message2 6000 &`. Como vemos en la Figura 4.27 el servidor atiende ambas conexiones.

```

23 void *handleThread(void *vargp)
24 {
25     int *mysock = (int *)vargp;
26     char buffer[BUFSIZE];
27     int received = -1;
28
29     printf("Executing handleThread with socket descriptor ID: %d\n", *mysock);
30     fprintf(stdout, "Thread ID in handler %lu\n", pthread_self());
31
32     read(*mysock, &buffer[0], BUFSIZE);
33     printf("message from client: %s\n", buffer);
34     write(*mysock, buffer, strlen(buffer)+1);
35     while(1);
36     close(*mysock);
37     return((void*)NULL);
38 }

```

Código 70: Parte del código del `thread_server1.c` donde vemos la función `handleThread()` para atender la conexión con cada nuevo cliente con un bucle infinito que impide su finalización.

Como no se va cerrando el descriptor de *socket*, los descriptors asignados son siempre diferentes. No eliminamos el descriptor, y por lo tanto el sistema operativo va asignando un número correlativo.

```

UOC1: make thread_server2 -f thread.Makefile
gcc -c thread_server2.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_server2 thread_server2.o -pthread -I ./
[1]+  Killed                  ./thread_server1 6000
UOC1: make thread_client2 -f thread.Makefile
gcc -c thread_client2.c -pthread -Wall -Wno-unused-variable -I ./
gcc -o thread_client2 thread_client2.o -pthread -I ./
UOC1:

```

Figura 4.27: Ejecución del servidor `thread_server2` y de dos instancias del cliente `thread_client2`.

Como vemos en la Figura 4.28 no vamos generando nuevos procesos para atender a cada nuevo cliente, pues sólo tenemos un proceso en ejecución en el sistema.

```

UOC1: ps -aux|grep thread
root    17304 93.4  0.0  18812   724 pts/0    S1   12:14   3:09 ./thread_server2 6000
root    17310  0.0  0.0   3084    884 pts/0    S+   12:17   0:00 grep thread
UOC1:

```

Figura 4.28: Comprobación de que sólo tenemos un proceso servidor `thread_server2` en ejecución en el sistema.

Como vemos en la Figura 4.29 tenemos tres hilos de ejecución asociados al proceso con PID 17304 que es el proceso del servidor.

```
UOC1: ps -o thcount 17304
THCNT
3
UOC1: █
```

Figura 4.29: Comprobación del número de hilos de ejecución asociados al proceso `thread_server2` en ejecución en el sistema.

Como vemos en la Figura 4.30 tenemos tres hilos de ejecución asociados al proceso con PID 17304 que es el proceso del servidor mirando el fichero de registro de información del proceso en el sistema con el comando `cat /proc/17304/status`.

```
UOC1: cat /proc/17304/status | grep Threads
Threads:      3
UOC1: █
```

Figura 4.30: Comprobación del número de hilos de ejecución asociados al proceso `thread_server2` en ejecución en el sistema.

Como vemos en la Figura 4.31 tenemos tres hilos de ejecución asociados al proceso con PID 17304 que es el proceso del servidor, con los identificadores de hilos de ejecución 17304 para el servidor que lanzamos en el terminal, 17306 para el hilo de ejecución que se creó para atender a la primera petición de conexión, y 17308 para el hilo de ejecución que se creó para atender a la segunda instancia de petición de conexión, gracias a la opción `-f` del comando `ps`.

```
UOC1: ps -elf | grep thread
root  17304  1 17304  0   3 12:14 pts/0    00:00:00 ./thread_server2 6000
root  17304  1 17306 49   3 12:14 pts/0    00:06:51 ./thread_server2 6000
root  17304  1 17308 49   3 12:14 pts/0    00:06:42 ./thread_server2 6000
root  17323  1 17323  0   1 12:28 pts/0    00:00:00 grep thread
UOC1: █
```

Figura 4.31: Comprobación del número de hilos de ejecución asociados al proceso `thread_server2` en ejecución en el sistema.

4.4. Conclusiones

Para resumir es importante en este punto recapitular la diferencia entre los procesos y los *threads*:

- Ambos representan una secuencia de ejecución independiente del proceso que los creó. En el caso de los *threads* se trabaja en un espacio de memoria compartida, mientras que en los procesos se trabaja con un espacio de memoria independiente. Por lo tanto los *threads* son más eficientes, y rápidos de crearse.
- Un proceso puede generar (tener) uno o más hilos de ejecución. Podríamos decir que un proceso es una tarea en ejecución.
- Los procesos sólo pueden interactuar a través de mecanismos de comunicación entre procesos (IPC) pues no comparten información (variables/memoria). Por contra los *threads* comparten variables globales con lo que es más sencillo compartir información entre ellos. Eso sí, no puede compartirse información sin un mecanismo de sincronización.
- Los *threads* comparten código, variables globales y ficheros y dispositivos abiertos. Evidentemente los *threads* no comparten el contador del programa (cada hilo está en un punto de ejecución diferente), registros de la CPU, pila para las variables locales de los procedimientos que se pudieran llamar después de crear el hilo, o el estado de ejecución del hilo.

Capítulo 5

Mecanismos de sincronización

5.1. Introducción

En este capítulo vamos a ver como podemos sincronizar diferentes procesos o hilos de ejecución que estén ejecutándose de forma paralela (concurrente) en un mismo sistema. La idea principal es disponer de algún mecanismo de aviso que pueda indicar a los diferentes elementos en ejecución que es su turno. La decisión no puede dejarse en manos del *scheduler* del sistema operativo, pues entonces la conmutación de proceso no es determinista.

5.2. Semáforos

Si deseamos que dos o más procesos o hilos de un sistema se puedan sincronizar, uno de los mejores mecanismos son los semáforos. Los semáforos vienen a ser como un contador (como si fuera una variable compartida entre todos los procesos que se van a sincronizar), sobre el que se definen dos operaciones principales. La primera es la operación de 'SIGNAL' (que incrementará siempre en una unidad el valor del contador) que suele utilizarse para avisar que una operación se ha finalizado. La segunda es la operación de 'WAIT' (que intentará decrementar en una unidad el valor del contador) que es una operación de bloqueo. Si el proceso que intenta decrementar el valor, no puede hacerlo (porque el valor del mismo es 0) entonces quedará bloqueado hasta recibir una notificación de que algún otro proceso lo ha incrementado con lo que podríamos nuevamente intentar decrementarlo. Es una operación que se suele hacer para indicar a un proceso que espere la ejecución de otro código por parte de otro proceso.

Uno de los usos habituales de los semáforos es para controlar el acceso a un recurso compartido. Para garantizar que sólo un proceso está accediendo al recurso compartido al mismo tiempo, y por lo tanto garantizar que se tiene control sobre lo que se hace con tal recurso una de las formas más simples de sincronización radica en tener un semáforo que comparten todos los procesos. Antes de acceder al recurso compartido todo proceso hará una operación de `'WAIT(semáforo)'`, si ha podido decrementar el semáforo indicará que tendrá el control del recurso compartido, con lo que lo podrá utilizar. Los demás procesos si ahora intentaran acceder al recurso, intentarían hacer un `'WAIT(semáforo)'`, pero como el valor del contador sería 0 ahora no lo podrían decrementar y estarían bloqueados a la espera que nuestro proceso acabara de utilizar el recurso compartido. En ese momento llevaría a cabo una operación de `'SIGNAL(semáforo)'` con lo que se incrementaría el valor del contador a uno lo que habilitaría la posibilidad de una operación de `'WAIT'`.

Las operaciones de `'WAIT'` y `'SIGNAL'` sobre semáforos se definen como operaciones atómicas, que significa que una vez empiezan a ejecutarse las instrucciones no pueden quedar interrumpidas a medias. Con lo que si vamos a incrementar el valor del contador, siempre lo incrementaremos. Y si lo quisiéramos intentar decrementar, que implica una operación de comprobación del valor del semáforo (contador) para saber si puede decrementarse y en caso que se pueda llevar a cabo la operación de decremento, todo ello siempre se ejecutará en una única ventana de ejecución del proceso. Es una forma de garantizar que no hacemos la comprobación de que podemos decrementar y entonces pasamos la CPU a otro proceso, porque entonces el mecanismo de sincronización no sería efectivo.

Para poder entender la necesidad de un mecanismo de sincronización, imaginemos el siguiente problema:

Suponed que tenemos el siguiente código para los procesos 'A', 'B' y 'C' de una aplicación concurrente. `'doA'` y `'doB'` son dos tareas con tiempo de ejecución indeterminado (por ejemplo porque esperan una respuesta del usuario) (nos garantizarán que nunca podremos predefinir la secuencia de ejecución de las mismas, pues a veces durarán más y a veces durarán menos). Las variables `'op'`, `'n'`, `'fa'` y `'fb'` se encuentran en memoria compartida y pueden ser accedidas/modificadas por los tres procesos, inicialmente valen 4.

Tenemos los tres procesos en la Tabla 5.1.

Intentemos explicar la salida por pantalla que mostrará la ejecución de esta aplicación concurrente. (Si hubiese más de una posibilidad las marcaremos todas).

SOLUCIÓN 1 = 22

PROCESO A	PROCESO B	PROCESO c
<pre>while (fa>2){ doA(); op = op * 2; fa--; n++;}</pre>	<pre>while (fb>1){ doB() op = op + 3; fb-=2; n++; }</pre>	<pre>while(n<8); printf(“ %d,“,op); exit(0);</pre>

Tabla 5.1: Problema de sincronización.

Haciendo un análisis del resultado del programa. Aquí haremos la suposición de que se ejecuta primero 'A', hasta el final, y entonces 'B', y finalmente 'C'. Con 'C' nunca tendremos conflicto pues está bloqueado hasta que deje de cumplirse la condición 'n<8'.

Los problemas de zona crítica los provocan las variables 'op' y 'n', que de hecho son las compartidas.

En la Tabla 5.2 'Reg' representa un registro interno del procesador para hacer las operaciones. Al inicio de cada proceso su valor dependerá de operaciones anteriores. Pero una vez fijado, el valor se mantendrá en cada proceso.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg	8				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
A6	while(fa>2)	8	5	3	4	5
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
A8-3	op = Reg	16				
A9	fa-=1			2		
A10-1	Reg = n		5			

Tabla 5.2 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A10-2	Reg++		6			
A10-3	n = Reg					6
A11	while(fa>2)	16	6	2	4	6
B1	while(fb>1)	16	?	2	4	6
B2	doB()					
B3-1	Reg = op		16			
B3-2	Reg = Reg +3		19			
B3-3	op = Reg	19				
B4	fb-=2				2	
B5-1	Reg = n		6			
B5-2	Reg++		7			
B5-3	n = Reg					7
B6	while(fb>1)	19	7	2	2	7
B7	doB()					
B8-1	Reg = op		19			
B8-2	Reg = Reg +3		22			
B8-3	op = Reg	22				
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B120-3	n = Reg					8
B11	while(fb>1	22	8	2	0	8
C1	while(n<8);	22	?	2	0	8
C2	printf(op)	22				

Tabla 5.2: Problema de sincronización: Solución 1 = 22.

SOLUCIÓN 2 = 25

Haciendo un análisis del resultado del programa, como vemos en la Tabla 5.3. Ahora ejecutaremos un bucle de 'A', uno de 'B', uno de 'A' y otro de 'B'.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			

Tabla 5.3 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A3-2	Reg = Reg * 2	8	8	3		
A3-3	op = Reg					
A4	fa-=1					
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
B1	while(fb>1)	8	?	3	4	5
B2	doB()	11			2	
B3-1	Reg = op		8			
B3-2	Reg = Reg + 3		11			
B3-3	op = Reg					
B4	fb-=2					
B5-1	Reg = n		5			
B5-2	Reg++	11	6			
B5-3	n = Reg					
A6	while(fa>2)		5	3	2	6
A7	doA()			2		
A8-1	Reg = op		11			
A8-2	Reg = Reg * 2		22			
A8-3	op = Reg					
A9	fa-=1					
A10-1	Reg = n		6			
A10-2	Reg++	22	7			
A10-3	n = Reg					
A11	while(fa>2)		7	2	2	2
B6	while(fb>1)		6	2	2	7
B7	doB()				0	
B8-1	Reg = op		22			
B8-2	Reg = Reg + 3		25			
B8-3	op = Reg					
B9	fb-=2					
B10-1	Reg = n		7			
B10-2	Reg++		8			

Tabla 5.3 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
B10-3	n = Reg					8
B11	while (fb>1)	25	8	2	0	8
C1	while(n<8);	25	?	2	0	8
C2	printf(op)	25				

Tabla 5.3: Problema de sincronización: Solución 2 = 25.

SOLUCIÓN 3 = 28

Ahora ejecutaremos un bucle de 'A', dos de 'B' y otro de 'A', como vemos en la Tabla 5.4.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg	8				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
B1	while(fb>1)	8	?	3	4	5
B2	doB()					
B3-1	Reg = op		8			
B3-2	Reg = Reg + 3		11			
B3-3	op = Reg	11				
B4	fb-=2				2	
B5-1	Reg = n		5			
B5-2	Reg++		6			
B5-3	n = Reg					6
B6	while(fb>1)	11	6	3	2	6
B7	doB()					
B8-1	Reg = op		11			
B8-2	Reg = Reg + 3		14			
B8-3	op = Reg	14				
B9	fb-=2				0	

Tabla 5.4 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
B10-1	Reg = n		6			
B10-2	Reg++		7			
B10-3	n = Reg					7
B11	while(fb>1)	14	7	3	0	7
A6	while(fa>2)	14	5	3	0	7
A7	doA()					
A8-1	Reg = op		14			
A8-2	Reg = Reg * 2		28			
A8-3	op = Reg	28				
A9	fa-=1			2		
A10-1	Reg = n		7			
A10-2	Reg++		8			
A10-3	n = Reg					8
A11	while(fa>2)	28	8	2	0	8
C1	while(n<8);	28	?	2	0	8
C2	printf(op)	28				

Tabla 5.4: Problema de sincronización: Solución 3 = 28.**SOLUCIÓN 4 = 40**

Primero ejecutaremos 'B' (dos veces) y después 'A' (también dos veces), como vemos en la Tabla 5.5.

HILO	Instrucción	op	Reg	fa	fb	n
B1	while(fb>1)	4	?	4	4	4
B2	doB()					
B3-1	Reg = op		4			
B3-2	Reg = Reg + 3		7			
B3-3	op = Reg	7				
B4	fb-=2				2	
B5-1	Reg = n		4			
B5-2	Reg++		5			
B5-3	n = Reg					5
B6	while(fb>1)	7	5	4	2	5
B7	doB()					

Tabla 5.5 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
B8-1	Reg = op	10	7			
B8-2	Reg = Reg + 3		10			
B8-3	op = Reg					
B9	fb-=2				0	
B10-1	Reg = n		5			
B10-2	Reg++		6			
B10-2	n = Reg					6
B11	while(fb>1)	10	6	4	0	6
A1	while(fa>2)	10	?	4	0	6
A2	doA()					
A3-1	Reg = op	20	10			
A3-2	Reg = Reg * 2		20			
A3-3	op = Reg					
A4	fa-=1			3		
A5-1	Reg = n		6			
A5-2	Reg++		7			
A5-3	n = Reg					7
A6	while(fa>2)	20	7	3	0	7
A7	doA()					
A8-1	Reg = op	40	20			
A8-2	Reg = Reg * 2		40			
A8-3	op = Reg					
A9	fa-=1			2		
A10-1	Reg = n		7			
A10-2	Reg++		8			
A10-3	n = Reg					8
A11	while(fa>2)	40	8	2	0	8
C1	while(n<8);	40	?	2	0	8
C2	printf(op)	40				

Tabla 5.5: Problema de sincronización: Solución 4 = 40.

SOLUCIÓN 5 = 34

Haciendo un análisis del resultado del programa. Ahora ejecutaremos un bucle de 'B', otro de 'A', de nuevo un bucle de 'B' y finalmente 'A', como vemos en la Tabla 5.6.

HILO	Instrucción	op	Reg	fa	fb	n
B1	while(fb>1)	4	?	4	4	4
B2	doB()					
B3-1	Reg = op		4			
B3-2	Reg = Reg + 3		7			
B3-3	op = Reg	7				
B4	fb-=2				2	
B5-1	Reg = n		4			
B5-2	Reg++		5			
B5-3	n = Reg					5
A1	while(fa>2)	7	?	4	2	5
A2	doA()					
A3-1	Reg = op		7			
A3-2	Reg = Reg * 2		14			
A3-3	op = Reg	14				
A4	fa-=1			3		
A5-1	Reg = n		5			
A5-2	Reg++		6			
A5-3	n = Reg					6
B6	while(fb>1)	14	5	3	2	6
B7	doB()					
B8-1	Reg = op		14			
B8-2	Reg = Reg + 3		17			
B8-3	op = Reg	17				
B9	fb-=2				0	
B10-1	Reg = n		6			
B10-2	Reg++		7			
B10-3	n = Reg					7
B11	while(fb>1)	17	7	3	0	7
A6	while(fa>2)	17	6	3	0	7
A7	doA()					
A8-1	Reg = op		17			
A8-2	Reg = Reg * 2		34			
A8-3	op = Reg	34				

Tabla 5.6 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A9	fa-=1			2		
A10-1	Reg = n		7			
A10-2	Reg++		8			
A10-3	n = Reg					8
A11	while(fa>2)	34	8	2	0	8
C1	while(n<8);	34	?	2	0	8
C2	printf(op)	34				

Tabla 5.6: Problema de sincronización: Solución 5 = 34.**SOLUCIÓN 6 = 31**

Ahora ejecutaremos un bucle de 'B', dos de 'A' y otro de 'B', como vemos en la Tabla 5.7.

HILO	Instrucción	op	Reg	fa	fb	n
B1	while(fb>1)	4	?	4	4	4
B2	doB()					
B3-1	Reg = op		4			
B3-2	Reg = Reg + 3		7			
B3-3	op = Reg	7				
B4	fb-=2				2	
B5-1	Reg = n		4			
B5-2	Reg++		5			
B5-3	n = Reg					5
A1	while(fa>2)	7	?	4	2	5
A2	doA()					
A3-1	Reg = op		7			
A3-2	Reg = Reg * 2		14			
A3-3	op = Reg	14				
A4	fa-=1			3		
A5-1	Reg = n		5			
A5-2	Reg++		6			
A5-3	n = Reg					6
A6	while(fa>2)	14	6	3	2	6
A7	doA()					
A8-1	Reg = op		14			

Tabla 5.7 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A8-2	Reg = Reg * 2	28	28	2		
A8-3	op = Reg					
A9	fa-=1					
A10-1	Reg = n		6			
A10-2	Reg++		7			
A10-3	n = Reg	28		2	2	7
A11	while(fa>2)		7			7
B6	while(fb>1)		5			7
B7	doB()					
B8-1	Reg = op		28			
B8-2	Reg = Reg + 3	31	31		0	
B8-3	op = Reg					
B9	fb-=2					
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg	31		2	0	8
B11	while(fb>1)		8			8
C1	while(n<8);		?			8
C2	printf(op)	31				

Tabla 5.7: Problema de sincronización: Solución 6 = 31.

Ahora haremos que las operaciones sobre las variables 'op' y 'n' no sean atómicas, y que por lo tanto pueda haber una conmutación de proceso antes de guardar. Primero provocaremos errores en la variable 'op'. Para provocar modificaciones en el resultado sólo podremos tener afectación si hacemos A y luego B, o B y después A. Es obvio que si la secuencia es A y A no puede haber error, pues el contexto de cada proceso siempre se guarda de forma correcta.

SOLUCIÓN 7 = 14

Variante de la solución 1. Aquí haremos 'A-A-B-B', como vemos en la Tabla 5.8.

Sólo podremos tener problemas entre el segundo 'A' y el primer 'B'.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					

Tabla 5.8 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A3-1	Reg = op	8	4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg					
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
A6	while(fa>2)	8	5	3	4	5
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
<p>Antes de guardar el resultado en op, el proceso 'B' lo leerá.</p> <p>Dos procesos acceden al mismo recurso al mismo tiempo.</p> <p>Problema de sincronización.</p>						
B1	while(fb>1)	8	?	3	4	5
B2	doB()					
B3-1	Reg = op		8			
B3-2	Reg = Reg + 3		11			
A8-3	op = Reg	16				
A9	fa-=1			2		
A10-1	Reg = n		5			
A10-2	Reg++		6			
A10-3	n = Reg					6
A11	while(fa>2)	16	6	2	4	6
Cambiamos el valor de op						
B3-3	op = Reg	11				
B4	fb-=2				2	
B5-1	Reg = n		6			
B5-2	Reg++		7			
B5-3	n = Reg					7
B6	while(fb>1)	11	7	2	2	7
B7	doB()					
B8-1	Reg = op		11			

Tabla 5.8 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
B8-2	Reg = Reg + 3	14	14			
B8-3	op = Reg					
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	14	8	2	0	8
C1	while(n<8);	14	?	2	0	8
C2	printf(op)	14				

Tabla 5.8: Problema de sincronización: Solución 7 = 14.**SOLUCIÓN 8 = 19**

Variante de la solución 1. Aquí haremos 'A-A-B-B', como vemos en la Tabla 5.9.

Sólo podremos tener problemas entre el segundo 'A' y el primer 'B'.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg	8				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
A6	while(fa>2)	8	5	3	4	5
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
<p>Antes de guardar el resultado en op, el proceso 'B' lo leerá.</p> <p>Dos procesos acceden al mismo recurso al mismo tiempo.</p> <p>Problema de sincronización.</p>						
B1	while(fb>1)	8	?	3	4	5
B2	doB()					

Tabla 5.9 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
B3-1	Reg = op	11	8			
B3-2	Reg = Reg +3		11			
B3-3	op = Reg					
Cambiamos el valor de op						
A8-3	op = Reg	16				
A9	fa-=1			2		
A10-1	Reg = n		5			
A10-2	Reg++		6			
A10-3	n = Reg					6
A11	while(fa>2)	16	6	2	4	6
B4	fb-=2	16			2	
B5-1	Reg = n		6			
B5-2	Reg++		7			
B5-3	n = Reg					7
B6	while(fb>1)	16	7	2	2	7
B7	doB()					
B8-1	Reg = op		16			
B8-2	Reg = Reg + 3		19			
B8-3	op = Reg	19				
B9	fb-=2					0
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	19	8	2	0	8
C1	while(n<8);	19	?	2	0	8
C2	printf(op)	19				

Tabla 5.9: Problema de sincronización: Solución 8 = 19.

SOLUCIÓN 9 = 16

Variante de la solución 1. Aquí hacemos 'A-A-B-B', como vemos en la Tabla 5.10.

Sólo podremos tener problemas entre el segundo 'A' y el primer 'B'.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4

Tabla 5.10 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A2	doA()	8				
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg					
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
A6	while(fa>2)	8	5	3	4	5
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
<p>Antes de guardar el resultado en op, el proceso 'B' lo leerá.</p> <p>Dos procesos acceden al mismo recurso al mismo tiempo.</p> <p>Problema de sincronización.</p>						
B1	while(fb>1)	8	?	3	4	5
B2	doB()					
B3-1	Reg = op		8			
B3-2	Reg = Reg + 3		11			
B3-3	op = Reg	11				
B4	fb-=2				2	
B5-1	Reg = n		5			
B5-2	Reg++		6			
B5-3	n = Reg					6
B6	while(fb>1)	11	6	3	2	6
B7	doB()					
B8-1	Reg = op		11			
B8-2	Reg = Reg + 3		14			
B8-3	op = Reg	14				
Cambiamos el valor de op						
A8-3	op = Reg	16				
A9	fa-=1			2		
A10-1	Reg = n		6			

Tabla 5.10 continua en la página siguiente...

HILO	Instrucción	op	Reg	fa	fb	n
A10-2	Reg++		7			
A10-3	n = Reg					7
A11	while(fa>2)	16	7	2	2	7
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	16	8	2	0	8
C1	while(n<8);	16	?	2	0	8
C2	printf(op)	16				

Tabla 5.10: Problema de sincronización: Solución 9 = 16.**SOLUCIÓN 10 = 14**

Variante de la solución 1. Aquí hacemos 'A-A-B-B', como vemos en la Tabla 5.11.

Sólo podremos tener problemas entre el segundo 'A' y el primer 'B'.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
A3-3	op = Reg	8				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
A6	while(fa>2)	8	5	3	4	5
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
<p>Antes de guardar el resultado en op, el proceso 'B' lo leerá.</p> <p>Dos procesos acceden al mismo recurso al mismo tiempo.</p> <p>Problema de sincronización.</p>						
Tabla 5.11 continua en la página siguiente...						

HILO	Instrucción	op	Reg	fa	fb	n
B1	while(fb>1)	8	?	3	4	5
B2	doB()					
B3-1	Reg = op		8			
B3-2	Reg = Reg + 3		11			
B3-3	op = Reg	11				
B4	fb-=2				2	
B5-1	Reg = n		5			
B5-2	Reg++		6			
B5-3	n = Reg					6
B6	while(fb>1)	11	6	3	2	6
B7	doB()					
B8-1	Reg = op		11			
B8-2	Reg = Reg + 3		14			
Cambiamos el valor de op						
A8-3	op = Reg	16				
Cambiamos el valor de op						
B8-3	op = Reg	14				
A9	fa-=1			2		
A10-1	Reg = n		6			
A10-2	Reg++		7			
A10-3	n = Reg					7
A11	while(fa>2)	14	7	2	2	7
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	14	8	2	0	8
C1	while(n<8);	14	?	2	0	8
C2	printf(op)	14				

Tabla 5.11: Problema de sincronización: Solución 10 = 14.**SOLUCIÓN 11 = 19**

Ahora haremos variantes de la solución 2.

Haciendo un análisis del resultado del programa. Ahora ejecutaremos un bucle de 'A',

uno de 'B', uno de 'A', uno de 'B', como vemos en la Tabla 5.12.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
<p>Antes de guardar el resultado en op, el proceso 'B' lo leerá.</p> <p>Dos procesos acceden al mismo recurso al mismo tiempo.</p> <p>Problema de sincronización.</p>						
B1	while(fb>1)	4	?	4	4	5
B2	doB()					
B3-1	Reg = op		4			
B3-2	Reg = Reg + 3		7			
B3-3	op = Reg	7				
Cambiamos el valor de op						
A3-3	op = Reg	8				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
B4	fb-=2				2	
B5-1	Reg = n		5			
B5-2	Reg++		6			
B5-3	n = Reg					6
A6	while(fa>2)	8	5	3	2	6
A7	doA()					
A8-1	Reg = op		8			
A8-2	Reg = Reg * 2		16			
A8-3	op = Reg	16				
A9	fa-=1			2		
A10-1	Reg = n		6			
A10-2	Reg++		7			
A10-3	n = Reg					7
A11	while(fa>2)	16	7	2	2	7
Tabla 5.12 continua en la página siguiente...						

HILO	Instrucción	op	Reg	fa	fb	n
B6	while(fb>1)	16	6	7	2	7
B7	doB()					
B8-1	Reg = op		16			
B8-2	Reg = Reg + 3		19			
B8-3	op = Reg	19				
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	19	8	2	0	8
C1	while(n<8);	19	?	2	0	8
C2	printf(op)	19				

Tabla 5.12: Problema de sincronización: Solución 11.**SOLUCIÓN 12 = 17**

Hacemos un análisis del resultado del programa. Ahora ejecutaremos un bucle de 'A', uno de 'B', uno de 'A' y otro de 'B', como vemos en la Tabla 5.13.

HILO	Instrucción	op	Reg	fa	fb	n
A1	while(fa>2)	4	?	4	4	4
A2	doA()					
A3-1	Reg = op		4			
A3-2	Reg = Reg * 2		8			
Antes de guardar el resultado en op, el proceso 'B' lo leerá. Dos procesos acceden al mismo recurso al mismo tiempo. Problema de sincronización.						
B1	while(fb>1)	4	?	4	4	5
B2	doB()					
B3-1	Reg = op		4			
B3-2	Reg = Reg + 3		7			
Cambiamos el valor de op						
A3-3	op = Reg	8				
Cambiamos el valor de op						
Tabla 5.13 continua en la página siguiente...						

HILO	Instrucción	op	Reg	fa	fb	n
B3-3	op = Reg	7				
A4	fa-=1			3		
A5-1	Reg = n		4			
A5-2	Reg++		5			
A5-3	n = Reg					5
B4	fb-=2				2	
B5-1	Reg = n		5			
B5-2	Reg++		6			
B5-3	n = Reg					6
A6	while(fa>2)	7	5	3	2	6
A7	doA()					
A8-1	Reg = op		7			
A8-2	Reg = Reg * 2		14			
A8-3	op = Reg	14				
A9	fa-=1			2		
A10-1	Reg = n		6			
A10-2	Reg++		7			
A10-3	n = Reg					7
A11	while(fa>2)	14	7	2	2	7
B6	while(fb>1)	14	6	2	2	7
B7	doB()					
B8-1	Reg = op		14			
B8-2	Reg = Reg + 3		17			
B8-3	op = Reg	17				
B9	fb-=2				0	
B10-1	Reg = n		7			
B10-2	Reg++		8			
B10-3	n = Reg					8
B11	while(fb>1)	17	8	2	0	8
C1	while(n<8);	17	?	2	0	8
C2	printf(op)	17				

Tabla 5.13: Problema de sincronización: Solución 12 = 17.

Para poder poner todos los casos, haremos una tabla con las operaciones sobre 'op'

del proceso 'A' ('RA' indicando lectura por parte del proceso 'A', 'WA' indicando escritura por parte del proceso 'A') y del proceso 'B' ('RB' indicando lectura por parte del proceso 'B', 'WB' indicando escritura por parte del proceso 'B'). Las diferentes opciones las tenemos en la Tabla 5.14.

1	op	2	op	3	op	4	op	5	op	6	op	7	op	8	op
Combinación 'A-A-B-B'															
RA1	4	WA1	8	RA2	8	WA2	16	RB1	16	WB1	19	RB2	19	WB2	22
RA1	4	WA1	8	RA2	8	RB1	8	WA2	16	WB1	11	RB2	11	WB2	14
RA1	4	WA1	8	RA2	8	RB1	8	WB1	11	WA2	16	RB2	16	WB2	19
RA1	4	WA1	8	RA2	8	RB1	8	WB1	11	RB2	11	WA2	16	WB2	14
RA1	4	WA1	8	RA2	8	RB1	8	WB1	11	RB2	11	WB2	14	WA2	16
Combinación 'A-B-A-B'															
RA1	4	WA1	8	RB1	8	WB1	11	RA2	11	WA2	22	RB2	22	WB2	25
RA1	4	WA1	8	RB1	8	WB1	11	RA2	11	RB2	11	WA2	22	WB2	14
RA1	4	WA1	8	RB1	8	WB1	11	RA2	11	RB2	11	WB2	14	WA2	22
RA1	4	WA1	8	RB1	8	RA2	8	WB1	11	WA2	16	RB2	16	WB2	19
RA1	4	WA1	8	RB1	8	RA2	8	WA2	16	WB1	11	RB2	11	WB2	14
RA1	4	WA1	8	RB1	8	RA2	8	WB1	11	RB2	11	WA2	16	WB2	14
RA1	4	WA1	8	RB1	8	RA2	8	WB1	11	RB2	11	WB2	14	WA2	16
RA1	4	RB1	4	WA1	8	WB1	7	RA2	7	WA2	14	RB2	14	WB2	17
RA1	4	RB1	4	WB1	7	WA1	8	RA2	8	WA2	16	RB2	16	WB2	19
RA1	4	RB1	4	WA1	8	WB1	7	RA2	7	RB2	7	WA2	14	WB2	10
RA1	4	RB1	4	WB1	7	WA1	8	RA2	8	RB2	8	WB2	11	WA2	16
RA1	4	RB1	4	WA1	8	RA2	8	WB1	7	RB2	7	WA2	16	WB2	10
RA1	4	RB1	4	WA1	8	RA2	8	WA2	16	WB1	7	RB2	7	WB2	10
RA1	4	RB1	4	WA1	8	RA2	8	WB1	7	WA2	16	RB2	16	WB2	19
RA1	4	RB1	4	WA1	8	RA2	8	WB1	7	RB2	7	WA2	16	WB2	19
RA1	4	RB1	4	WA1	8	RA2	8	WB1	7	RB2	7	WB2	10	WA2	16
Combinación 'A-B-B-A'															
RA1	4	WA1	8	RB1	8	WB1	11	RB2	11	WB2	14	RA2	14	WA2	28
RA1	4	RB1	4	WA1	8	WB1	7	RB2	7	WB2	10	RA2	10	WA2	20
RA1	4	RB1	4	WB1	7	WA1	8	RB2	8	WB2	11	RA2	11	WA2	22
RA1	4	WA1	8	RB1	8	WB1	11	RB2	22	RA2	11	WB2	25	WA2	22
Tabla 5.14 continua en la página siguiente...															

Tabla 5.14 continua en la página siguiente...

1 op	2 op	3 op	4 op	5 op	6 op	7 op	8 op
RA1 4	RB1 4	WA1 8	WB1 7	RB2 7	RA2 7	WB2 10	WA2 14
RA1 4	RB1 4	WB1 7	WA1 8	RB2 8	RA2 8	WB2 11	WA2 16
RA1 4	WA1 8	RB1 8	WB1 11	RB2 22	RA2 11	WA2 22	WB2 25
RA1 4	RB1 4	WA1 8	WB1 7	RB2 7	RA2 7	WA2 14	WB2 10
RA1 4	RB1 4	WB1 7	WA1 8	RB2 8	RA2 8	WA2 16	WB2 11
RA1 4	RB1 4	WB1 7	RB2 7	WA1 8	WB2 10	RA2 10	WA2 20
RA1 4	RB1 4	WB1 7	RB2 7	WA1 8	RA2 8	WB2 10	WA2 16
RA1 4	RB1 4	WB1 7	RB2 7	WA1 8	RA2 8	WA2 16	WB2 10
RA1 4	RB1 4	WB1 7	RB2 7	WB2 20	WA1 8	RA2 8	WA2 16
Combinación 'B-B-A-A'							
RB1 4	WB1 7	RB2 7	WB2 10	RA1 10	WA1 20	RA2 20	WA2 40
RB1 4	WB1 7	RB2 7	RA1 7	WB2 10	WA1 14	RA2 14	WA2 28
RB1 4	WB1 7	RB2 7	RA1 7	WA1 14	WB2 10	RA2 10	WA2 20
RB1 4	WB1 7	RB2 7	RA1 7	WA1 14	RA2 14	WB2 10	WA2 28
RB1 4	WB1 7	RB2 7	RA1 7	WA1 14	RA2 14	WA2 28	WB2 10
Combinación 'B-A-B-A'							
RB1 4	WB1 7	RA1 7	WA1 14	RB2 14	WB2 17	RA2 17	WA2 34
RB1 4	WB1 7	RA1 7	WA1 14	RB2 14	RA2 14	WB2 17	WA2 28
RB1 4	WB1 7	RA1 7	WA1 14	RB2 14	RA2 14	WA2 28	WB2 17
RB1 4	WB1 7	RA1 7	RB2 7	WA1 14	WB2 10	RA2 10	WA2 20
RB1 4	WB1 7	RA1 7	RB2 7	WB2 10	WA1 14	RA2 14	WA2 28
RB1 4	WB1 7	RA1 7	RB2 7	WA1 14	RA2 14	WB2 10	WA2 28
RB1 4	WB1 7	RA1 7	RB2 7	WA1 14	RA2 14	WA2 28	WB2 10
RB1 4	RA1 4	WB1 7	WA1 8	RB2 8	WB2 11	RA2 11	WA2 22
RB1 4	RA1 4	WA1 8	WB1 7	RB2 7	WB2 10	RA2 10	WA2 20
RB1 4	RA1 4	WB1 7	WA1 8	RB2 8	RA2 8	WB2 11	WA2 16
RB1 4	RA1 4	WA1 8	WB1 7	RB2 7	RA2 7	WA2 14	WB2 10
RB1 4	RA1 4	WB1 7	RB2 7	WA1 8	RA2 8	WB2 10	WA2 16
RB1 4	RA1 4	WB1 7	RB2 7	WB2 10	WA1 8	RA2 8	WA2 16
RB1 4	RA1 4	WB1 7	RB2 7	WA1 8	WB2 10	RA2 10	WA2 20
RB1 4	RA1 4	WB1 7	RB2 7	WA1 8	RA2 8	WB2 10	WA2 16
RB1 4	RA1 4	WB1 7	RB2 7	WA1 8	RA2 8	WA2 16	WB2 10
Tabla 5.14 continua en la página siguiente...							

1 op	2 op	3 op	4 op	5 op	6 op	7 op	8 op
Combinación 'B-A-A-B'							
RB1 4	WB17	RA1 7	WA1 14	RA2 14	WA2 28	RB2 28	WB2 31
RB1 4	RA1 4	WB17	WA1 8	RA2 8	WA2 16	RB2 16	WB2 19
RB1 4	RA1 4	WA1 8	WB17	RA2 7	WA2 14	RB2 14	WB2 17
RB1 4	WB17	RA1 7	WA1 14	RA2 14	RB2 14	WA2 28	WB2 17
RB1 4	RA1 4	WB17	WA1 8	RA2 8	RB2 8	WA2 16	WB2 11
RB1 4	RA1 4	WA1 8	WB17	RA2 7	RB2 7	WA2 14	WB2 10
RB1 4	WB17	RA1 7	WA1 14	RA2 14	RB2 14	WB2 17	WA2 28
RB1 4	RA1 4	WB17	WA1 8	RA2 8	RB2 8	WB2 11	WA2 16
RB1 4	RA1 4	WA1 8	WB17	RA2 7	RB2 7	WB2 10	WA2 14
RB1 4	RA1 4	WA1 8	RA2 8	WB17	WA2 16	RB2 16	WB2 19
RB1 4	RA1 4	WA1 8	RA2 8	WB17	RB2 7	WA2 16	WB2 10
RB1 4	RA1 4	WA1 8	RA2 8	WB17	RB2 7	WB2 10	WA2 16
RB1 4	RA1 4	WA1 8	RA2 8	WA2 16	WB17	RB2 7	WB2 10

Tabla 5.14: Posibles combinaciones con operaciones sobre la variable 'op'.

En el caso de la variable 'n', si en alguna de las operaciones de incremento no se guarda antes de volver a leer, el contador no llegará a 8, y el programa no imprimirá por pantalla.

Por lo tanto, las posibles respuestas son:

1. Ninguna impresión por pantalla (el programa no finaliza).
2. Impresión del valor 10, 11, 14, 16, 17, 19, 20, 22, 25, 28, 31, 34, 40 y finalización del programa.

Como vemos cuando nos encontramos ante procesos concurrentes, la gestión del resultado se hace absolutamente inviable, si no podemos garantizar que la ejecución es determinista (que podemos predecir el resultado).

Para que una aplicación concurrente sea determinista y tenga un único resultado debemos forzar la sincronización (secuencia de ejecución) de la misma. Para evitar las condiciones de carrera, utilizaremos semáforos para la sincronización de los procesos en la zona crítica. Una posible solución sería, con un semáforo inicializado a 1 ('M1').

Tenemos los tres procesos en la Tabla 5.15.

PROCESO A	PROCESO B	PROCESO c
<pre>while (fa>2){ doA(); sem_wait(M1); op = op * 2; fa--; n++; sem_signal(M1); }</pre>	<pre>while (fb>1){ doB() sem_wait(M1); op = op + 3; fb-=2; n++; sem_signal(M1); }</pre>	<pre>while(n<8); printf(“ %d“,op); exit(0);</pre>

Tabla 5.15: Problema de sincronización con un semáforo 'M1' para controlar acceso a zona crítica.

Una vez realizados los cambios del apartado anterior, ¿Cuál será el resultado que mostrará la aplicación por pantalla?

Ahora ya no tendremos condiciones de carrera y podremos tener como posibles soluciones: 22, 25, 28, 31, 34, 40.

No tenemos un único resultado pues no marcamos el acceso a la zona crítica, podemos tener diferentes secuencias AABB, ABAB...Lo que controlamos es que si un proceso está accediendo el otro no puede hacerlo, pero no controlamos el orden de los procesos, como vemos en la Tabla 5.15.

1	op	2	op	3	op	4	op	5	op	6	op	7	op	8	op
Combinación 'A-A-B-B'															
RA1	4	WA1	8	RA2	8	WA2	16	RB1	16	WB1	19	RB2	19	WB2	22
Combinación 'A-B-A-B'															
RA1	4	WA1	8	RB1	8	WB1	11	RA2	11	WA2	22	RB2	22	WB2	25
Combinación 'A-B-B-A'															
RA1	4	WA1	8	RB1	8	WB1	11	RB2	11	WB2	14	RA2	14	WA2	28
Combinación 'B-B-A-A'															
RB1	4	WB1	7	RB2	7	WB2	10	RA1	10	WA1	20	RA2	20	WA2	40
Combinación 'B-A-B-A'															
RB1	4	WB1	7	RA1	7	WA1	14	RB2	14	WB2	17	RA2	17	WA2	34
Combinación 'B-A-A-B'															
RB1	4	WB1	7	RA1	7	WA1	14	RA2	14	WA2	28	RB2	28	WB2	31
Tabla 5.16 continua en la página siguiente...															

1	op	2	op	3	op	4	op	5	op	6	op	7	op	8	op
---	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----

Tabla 5.16: Posibles combinaciones con operaciones sobre la variable 'op'. controladas con semáforo 'M1'.

Otro de los problemas de la solución actual es la espera activa en el proceso 'C'. Está continuamente comprobando el valor de la variable para saber si puede o no imprimir. Sería mejor disponer de un mecanismo de sincronización para que los otros procesos avisarán.

Para evitar la espera activa, activaremos un semáforo con valor inicial 0 ('M2') y haremos un 'SIGNAL' para cada una de las cuatro operaciones que hemos de hacer. Al finalizar haremos un 'WAIT(4)' en el proceso 'C' (o encadenaremos 4 'WAIT' que es la solución indicada).

Tenemos los tres procesos en la Tabla 5.17.

PROCESO A	PROCESO B	PROCESO c
<pre>while (fa>2){ doA(); sem_wait(M1); op = op * 2; fa--; n++; sem_signal(M1); sem_signal(M2); }</pre>	<pre>while (fb>1){ doB(); sem_wait(M1); op = op + 3; fb-=2; n++; sem_signal(M1); sem_signal(M2); }</pre>	<pre>sem_wait(M2); sem_wait(M2); sem_wait(M2); sem_wait(M2); printf("%d",op); exit(0);</pre>

Tabla 5.17: Problema de sincronización con un semáforo 'M2' para evitar la espera activa en el proceso 'C'.

Por lo tanto hemos visto, como cuando tenemos concurrencia la sincronización con semáforos es imprescindible. Veamos pues como se utilizan los semáforos en un programa en C.

Lo primero que debemos hacer para trabajar con semáforos es disponer del identificador del mismo. Será el puntero que nos permitirá acceder de forma compartida (entre todos los procesos que lo definan) al semáforo; lo obtendremos con una llamada a la función 'sem_open()' como vemos en la Figura 5.1 con la llamada a 'man sem_open'.

Es importante recordar que nuevamente debemos compilar con la opción '-pthread'.

Como en cualquier función que retorne un descriptor, podemos indicar opciones de creación del mismo como 'O_CREAT' o 'O_EXCL'.

```
UOC1: man sem_open | head -15
SEM_OPEN(3)                                Linux Programmer's Manual          SEM_OPEN(3)

NAME
    sem_open - initialize and open a named semaphore

SYNOPSIS
    #include <fcntl.h>                /* For O_* constants */
    #include <sys/stat.h>            /* For mode constants */
    #include <semaphore.h>

    sem_t *sem_open(const char *name, int oflag);
    sem_t *sem_open(const char *name, int oflag,
                    mode_t mode, unsigned int value);

    Link with -pthread.
UOC1: █
```

Figura 5.1: Formato de la función 'sem_open()' con la llamada al manual del sistema.

El primero de los procesos que defina el semáforo lo debe crear y los demás apuntarán al descriptor ya existente.

El nombre que debemos indicar en la función 'sem_open()' sigue las indicaciones de 'sem_overview()' como vemos en la Figura 5.2.

```
UOC1: man sem_overview | head -30
SEM_OVERVIEW(7)                            Linux Programmer's Manual          SEM_OVERVIEW(7)

NAME
    sem_overview - overview of POSIX semaphores

DESCRIPTION
    POSIX semaphores allow processes and threads to synchronize their actions.

    A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

    POSIX semaphores come in two forms: named semaphores and unnamed semaphores.

    Named semaphores
        A named semaphore is identified by a name of the form /somenam; that is, a null-terminated string of up to NAME_MAX-4 (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to sem_open(3).

        The sem_open(3) function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using sem_post(3) and sem_wait(3). When a process has finished using the semaphore, it can use sem_close(3) to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using sem_unlink(3).
UOC1: █
```

Figura 5.2: Formato del nombre a utilizar en el caso de semáforos con nombre de acuerdo con la llamada al manual del sistema para la función 'sem_overview'.

En nuestros ejemplos trabajaremos con semáforos con nombre que de hecho son los

más genéricos pues no se necesita relación entre los procesos que acceden al mismo, como vemos en el Código 71.

```
61  /* Open psem1 */
62  psem1 = (sem_t*) sem_open("/sem1", O_CREAT, 0644, 0);
63  if (psem1 == SEM_FAILED) {
64      err_sys("Open psem1");
65  }
```

Código 71: Parte del código del 'sem_1a.c' donde vemos la función 'sem_open()' con el nombre del semáforo 'sem1' que vamos a utilizar.

Cuando se trabaja con semáforos, uno de los peligros es que mientras estamos depurando (acabando de hacer el programa) los semáforos no se vuelven a inicializar al volver a lanzar el programa.

Una de las soluciones es comprobar que el valor del semáforo es el que nos interesa. Una posible solución sería la que vemos en el Código 72.

```
67  /* Read and print semaphore value */
68  result = sem_getvalue(psem1, &sem_value);
69  if (result < 0) {
70      err_sys("Read psem1");
71  }
72  printf("PROCESS 1(SEM1): %d\n", sem_value);
73
74  while (sem_value > 0) {
75      sem_wait(psem1);
76      sem_value--;
77  }
```

Código 72: Parte del código del 'sem_1a.c' donde vemos el bucle para poner a cero el valor del semáforo al inicio del programa.

Tomamos el valor del semáforo, y si éste es superior a cero, le vamos haciendo 'wait' (decrementos) hasta que el valor sea nulo. Esto se hace en el programa que lanzamos primero.

La función de decrementar (intentar decrementar el valor) la acabamos de ver ('sem_wait()') con parámetro el identificador del semáforo. Para incrementar el valor del mismo ('sem_post()') que vemos en el Código 73.

```
86  /* Increment the value of semaphore to initialize it to 1 */
87  result = sem_post(psem1);
88  if (result < 0) {
89      err_sys("Post psem1");
90  }
```

Código 73: Parte del código del 'sem_1a.c' donde vemos el bucle para incrementar el valor del semáforo al inicio del programa..

Para la generación de los diferentes ficheros ejecutables de este apartado tenemos el fichero de configuración 'sem.Makefile' que vemos en el Código 74.

```
1  PROGRAM_NAME_1 = sem_1a
2  PROGRAM_OBJS_1 = sem_1a.o
3
4  PROGRAM_NAME_2 = sem_2a
5  PROGRAM_OBJS_2 = sem_2a.o
6
7  PROGRAM_NAME_3 = sem_3a
8  PROGRAM_OBJS_3 = sem_3a.o
9
10 PROGRAM_NAME_4 = sem_4a
11 PROGRAM_OBJS_4 = sem_4a.o
12
13 PROGRAM_NAME_5 = sem_4b
14 PROGRAM_OBJS_5 = sem_4b.o
15
16 PROGRAM_NAME_6 = sem_5a
17 PROGRAM_OBJS_6 = sem_5a.o
18
19 PROGRAM_NAME_7 = sem_5b
20 PROGRAM_OBJS_7 = sem_5b.o
21
22 PROGRAM_NAME_8 = sem_5c
23 PROGRAM_OBJS_8 = sem_5c.o
24
25 PROGRAM_NAME_9 = sem_6a
26 PROGRAM_OBJS_9 = sem_6a.o
27
28 PROGRAM_NAME_10 = sem_6b
```

```
29 PROGRAM_OBJS_10 = sem_6b.o
30
31 PROGRAM_NAME_11 = sem_6c
32 PROGRAM_OBJS_11 = sem_6c.o
33
34 PROGRAM_NAME_12 = sem_cleaning
35 PROGRAM_OBJS_12 = sem_cleaning.o
36
37 PROGRAM_NAME_13 = sem_cleaning_name
38 PROGRAM_OBJS_13 = sem_cleaning_name.o
39
40 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
    ↪ $(PROGRAM_NAME_4) $(PROGRAM_NAME_5) $(PROGRAM_NAME_6) $(PROGRAM_NAME_7)
    ↪ $(PROGRAM_NAME_8) $(PROGRAM_NAME_9) $(PROGRAM_NAME_10) $(PROGRAM_NAME_11)
    ↪ $(PROGRAM_NAME_12) $(PROGRAM_NAME_13)
41 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
    ↪ $(PROGRAM_OBJS_4) $(PROGRAM_OBJS_5) $(PROGRAM_OBJS_6) $(PROGRAM_OBJS_7)
    ↪ $(PROGRAM_OBJS_8) $(PROGRAM_OBJS_9) $(PROGRAM_OBJS_10) $(PROGRAM_OBJS_11)
    ↪ $(PROGRAM_OBJS_12) $(PROGRAM_OBJS_13)
42
43 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
44
45 all: $(PROGRAM_NAME_ALL)
46     @echo "Finished!"
47
48 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
49     gcc -o $$@ $$^ -I ./ -pthread
50
51 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
52     gcc -o $$@ $$^ -I ./ -pthread
53
54 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
55     gcc -o $$@ $$^ -I ./ -pthread
56
57 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
58     gcc -o $$@ $$^ -I ./ -pthread
59
60 $(PROGRAM_NAME_5): $(PROGRAM_OBJS_5)
61     gcc -o $$@ $$^ -I ./ -pthread
62
63 $(PROGRAM_NAME_6): $(PROGRAM_OBJS_6)
64     gcc -o $$@ $$^ -I ./ -pthread
```

```
65
66 $(PROGRAM_NAME_7): $(PROGRAM_OBJS_7)
67 gcc -o $@ $^ -I ./ -pthread
68
69 $(PROGRAM_NAME_8): $(PROGRAM_OBJS_8)
70 gcc -o $@ $^ -I ./ -pthread
71
72
73 $(PROGRAM_NAME_9): $(PROGRAM_OBJS_9)
74 gcc -o $@ $^ -I ./ -pthread
75
76 $(PROGRAM_NAME_10): $(PROGRAM_OBJS_10)
77 gcc -o $@ $^ -I ./ -pthread
78
79 $(PROGRAM_NAME_11): $(PROGRAM_OBJS_11)
80 gcc -o $@ $^ -I ./ -pthread
81
82 $(PROGRAM_NAME_12): $(PROGRAM_OBJS_12)
83 gcc -o $@ $^ -I ./ -pthread
84
85 $(PROGRAM_NAME_13): $(PROGRAM_OBJS_13)
86 gcc -o $@ $^ -I ./ -pthread
87
88 %.o: %.c
89 gcc -c $< -Wall -Wno-unused-variable -I ./ -pthread
90
91 clean:
92 rm -f $(REBUIDABLES)
93 @echo "Clean done"
```

Código 74: Fichero de configuración de 'make' de nombre 'sem.Makefile'.

En el primero de los ejemplos que tenemos en el Código 75 Vamos iterando (las veces que nos indica el argumento de entrada del programa) la impresión de un mensaje. Antes de la impresión pedimos permiso para acceder al recurso compartido ('WAIT'). Entonces esperamos que el usuario pulse cualquier tecla (con una llamada a la función 'inkey()') y damos permiso ('POST') para que el mismo proceso u otro tenga acceso al recurso compartido.


```
1  /*
2   * Filename: sem_1a.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD  0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 char inkey(void) {
27     char c;
28     struct termio param_ant, params;
29
30     ioctl(STDINFD, TCGETA, &param_ant);
31
32     params = param_ant;
33     params.c_lflag &= ~(ICANON | ECHO);
34     params.c_cc[4] = 1;
35
36     ioctl(STDINFD, TCSETA, &params);
37
38     fflush(stdin); fflush(stderr); fflush(stdout);
39     read(STDINFD, &c, 1);
40
41     ioctl(STDINFD, TCSETA, &param_ant);
```

```
42     return c;
43 }
44
45 int main(int argc, char *argv[])
46 {
47     int    input;
48     char    buffer[10 + 2];
49     sem_t*  psem1;
50     int     sem_value;
51     char    key;
52     int     count;
53     int     result;
54
55     /* Parse program arguments */
56     if (argc != 2) {
57         fprintf(stderr, "Usage: %s <iterations>\n", argv[0]);
58         exit(1);
59     }
60
61     /* Open psem1 */
62     psem1 = (sem_t*) sem_open("/sem1", O_CREAT, 0644, 0);
63     if (psem1 == SEM_FAILED) {
64         err_sys("Open psem1");
65     }
66
67     /* Read and print semaphore value */
68     result = sem_getvalue(psem1, &sem_value);
69     if (result < 0) {
70         err_sys("Read psem1");
71     }
72     printf("PROCESS 1(SEM1): %d\n", sem_value);
73
74     while (sem_value > 0) {
75         sem_wait(psem1);
76         sem_value--;
77     }
78
79     /* Read and print semaphore value */
80     result = sem_getvalue(psem1, &sem_value);
81     if (result < 0) {
82         err_sys("Read psem1");
83     }
}
```

```
84     printf("PROCESS 1(SEM1): %d\n", sem_value);
85
86     /* Increment the value of semaphore to initialize it to 1 */
87     result = sem_post(psem1);
88     if (result < 0) {
89         err_sys("Post psem1");
90     }
91
92     /* Repeat */
93     for (count = 0; count < atoi(argv[1]); count++) {
94         /* Entering critical zone */
95         result = sem_wait(psem1);
96         if (result < 0) {
97             err_sys("Wait psem1");
98         }
99
100        /* Critical zone */
101        fprintf(stdout, "Critical zone, process A, %d\n", count);
102
103        /* Wait user type any key to continue */
104        inkey();
105
106        /* Exiting critical zone */
107        result = sem_post(psem1);
108        if (result < 0) {
109            err_sys("Post psem1");
110        }
111    }
112
113    /* Close de semaphore ID */
114    result = sem_close(psem1);
115    if (result != 0) {
116        err_sys("Close psem1");
117    }
118
119    /* Unlink the semaphore (no one is using neither will be able to use again the
120    ↪ semaphore */
121    result = sem_unlink("/sem1");
122    if (result != 0) {
123        err_sys("Unlink /sem1");
124    }
```

```
125     exit(0);  
126 }
```

Código 75: Código del ejemplo 'sem_1a.c'.

En este primer ejemplo sólo tenemos un proceso, pero aún así vamos a ir sincronizando las operaciones sobre la variable (que podríamos considerar nuestro recurso compartido).

Generamos el fichero ejecutable con la llamada a 'make sem_1a -f sem.Makefile' como vemos en la Figura 5.3.

```
UOC1: make sem_1a -f sem.Makefile  
gcc -c sem_1a.c -Wall -Wno-unused-variable -I ./ -pthread  
gcc -o sem_1a sem_1a.o -I ./ -pthread  
UOC1: █
```

Figura 5.3: Generación del fichero ejecutable 'sem_1a'.

Como vemos en la Figura 5.4 al principio del programa nos aseguramos que el valor del semáforo es 0. Según el número de iteraciones que pasemos como argumento al programa, nos imprimirá un número u otro de mensajes del tipo 'critical zone, process A, 0' y esperará al final de cada mensaje que el usuario pulse una tecla para continuar.

```
Usage: ./sem_1a <iterations>  
UOC1: ./sem_1a 2  
PROCESS 1(SEM1): 0  
PROCESS 1(SEM1): 0  
Critical zone, process A, 0  
Critical zone, process A, 1  
UOC1: ./sem_1a 5  
PROCESS 1(SEM1): 0  
PROCESS 1(SEM1): 0  
Critical zone, process A, 0  
Critical zone, process A, 1  
Critical zone, process A, 2  
Critical zone, process A, 3  
Critical zone, process A, 4  
UOC1: █
```

Figura 5.4: Ejemplo de ejecución del programa './sem_1a'.

Como al finalizar el programa antes hacemos un 'unlink()' del semáforo cuando empezamos una segunda ejecución del programa el semáforo siempre se crea nuevo y vale 0.

Para poder ver el efecto de no hacer la llamada a la función 'unlink()' haremos una segunda versión del programa 'sem_2a.c' donde saldremos del programa antes de la llamada como vemos en el Código 76.

```
118     exit(0);
119
120     /* Unlink the semaphore (no one is using neither will be able to use again the
121        ↪ semaphore */
122     result = sem_unlink("/sem_1");
123     if (result != 0) {
124         err_sys("Unlink /sem_1");
125     }
```

Código 76: Parte del código del 'sem_2a.c' donde vemos como primero ejecutamos la llamada 'exit()' y por lo tanto nunca ejecutaremos la llamada 'unlink()'.

Generamos el fichero ejecutable con la llamada a 'make sem_2a -f sem.Makefile' como vemos en la Figura 5.5.

```
UOC1: make sem_2a -f sem.Makefile
gcc -c sem_2a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_2a sem_2a.o -I ./ -pthread
UOC1:
```

Figura 5.5: Generación del fichero ejecutable 'sem_2a'.

Como vemos en la Figura 5.6 al principio del programa nos aseguramos que el valor del semáforo es 0. Según el número de iteraciones que pasemos como argumento al programa, nos imprimirá un número u otro de mensajes del tipo 'critical zone, process A, 0' y esperará al final de cada mensaje que el usuario pulse una tecla para continuar.

```
UOC1: ./sem_2a
Usage: ./sem_2a <iterations>
UOC1: ./sem_2a 2
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process A, 0
Critical zone, process A, 1
UOC1: ./sem_2a 5
PROCESS 1(SEM1): 1
PROCESS 1(SEM1): 0
Critical zone, process A, 0
Critical zone, process A, 1
Critical zone, process A, 2
Critical zone, process A, 3
Critical zone, process A, 4
UOC1:
```

Figura 5.6: Ejemplo de ejecución del programa './sem_2a'.

Como salimos del programa antes de ejecutar el 'unlink()', ahora al principio de la segunda ejecución el valor del semáforo será 1 debido al último 'POST' que se ejecutó en la

primera ejecución (aunque lo corregimos con nuestro control), y vemos lo conveniente de nuestro bucle inicial. Si algún programa no acabara bien, garantizaríamos que el semáforo tiene el valor que nos interesa.

Otra solución que siempre funciona bien, es hacer un pequeño programa de eliminación de los semáforos, como el del Código 77 que podemos ejecutar siempre que queramos volver a la situación inicial, pasando como argumento el nombre del semáforo que queremos eliminar.

```
1  /*
2   * Filename: sem_cleaning_name.c
3   */
4
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9
10 #include <sys/stat.h>
11 #include <fcntl.h>
12
13 #include <sys/ipc.h>
14 #include <sys/sem.h>
15 #include <semaphore.h>
16
17
18 void err_sys(const char* text)
19 {
20     perror(text);
21     exit(1);
22 }
23
24
25 int main(int argc, char* argv[])
26 {
27
28     /* Check input arguments */
29     if (argc != 2) {
30         fprintf(stderr, "Usage: %s <name>\n", argv[0]);
31         exit(1);
32     }
33
```

```
34  if (sem_unlink(argv[1])!=0) err_sys(strcat("UNLINK ",argv[1]));
35  exit(0);
36 }
```

Código 77: Código del 'sem_cleaning_name.c'.

Generamos el fichero ejecutable con la llamada a 'make sem_cleaning_name -f sem.Makefile' como vemos en la Figura 5.7.

```
UOC1: make sem_cleaning_name -f sem.Makefile
gcc -c sem_cleaning_name.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_cleaning_name sem_cleaning_name.o -I ./ -pthread
UOC1: █
```

Figura 5.7: Generación del fichero ejecutable 'sem_cleaning_name'.

Como vemos en la Figura 5.8 primero ejecutamos 'sem_2a' donde vemos que el valor inicial del semáforo es 0; hacemos una segunda instancia del mismo programa para ver que ahora el valor inicial del semáforo antes de la corrección es 1. Ejecutamos nuestro programa de eliminación del semáforo ('sem_cleaning_name "/sem_1"') para eliminar del sistema del semáforo, y ahora al volver a ejecutar el programa 'sem_2a' vemos que nuevamente el valor inicial del semáforo es 0.

```
UOC1: ./sem_2a 1
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process A, 0
UOC1: ./sem_2a 1
PROCESS 1(SEM1): 1
PROCESS 1(SEM1): 0
Critical zone, process A, 0
UOC1: ./sem_cleaning_name "/sem_1"
UOC1: ./sem_2a 1
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process A, 0
UOC1: █
```

Figura 5.8: Ejemplo de ejecución del programa './sem_cleaning_name'.

Ahora vamos a tener más de un proceso acceso al recurso compartido. Lo que haremos será no eliminar el semáforo al final del programa, y lanzar varias instancias del mismo, como vemos en el Código 78

Generamos el fichero ejecutable con la llamada a 'make sem_3a -f sem.Makefile' como vemos en la Figura 5.9.

```
112     }
113
114     exit(0);
115
116     /* Close psem1 */
117     result = sem_close(psem1);
118     if (result != 0) {
119         err_sys("Close psem1");
120     }
121
122     /* Unlink the semaphore (no one is using neither will be able to use again the
123        ↪ semaphore */
124     result = sem_unlink("/sem1");
125     if (result != 0) {
126         err_sys("Unlink /sem1");
127     }
```

Código 78: Parte del código del 'sem_3a.c' donde vemos como primero ejecutamos la llamada 'exit()' y por lo tanto nunca ejecutaremos la llamada 'unlink()'.

```
UOC1: make sem_3a -f sem.Makefile
gcc -c sem_3a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_3a sem_3a.o -I ./ -pthread
UOC1: █
```

Figura 5.9: Generación del fichero ejecutable 'sem_3a'.

Como vemos en la Figura 5.10 para lanzar varias iteraciones podemos ejecutar './sem_3a 2 & ./sem_3a 3' con lo que lanzamos dos procesos (dos instancias del mismo proceso, una con parámetro de entrada 2 y que ejecutaremos en modo *background* y la segunda con parámetro de entrada 3 y que dejaremos en modo *foreground*).

El primero de los procesos (PID=18267) queda en modo *background*, y por lo tanto no podemos tener acceso a teclado, y sólo se ejecuta el segundo proceso. Éste proceso es el que toma el control del recurso compartido hasta que pasan las 3 iteraciones. Como vemos podemos retomar el control del primer proceso ('fg') y finalizarlo.

Si trabajamos en un entorno con semáforos System V, podemos ver información de los mismos en el sistema con el comando ('ipcs') del que vemos la página de ayuda en la Figura 5.11

En la Figura 5.12 vemos como el comando no nos da información del semáforo que tenemos al ser un semáforo en un entorno 'POSIX'.

En 'POSIX' podemos localizar la información en el directorio en el que se montan (cargan) los semáforos (se montan siempre en '/dev/shm')


```

UOC1: ./sem_3a 2 & ./sem_3a 3
[1] 18267
PROCESS 1(SEM1): 2
PROCESS 1(SEM1): 0
Critical zone, process 18268, 0
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process 18267, 0
Critical zone, process 18268, 1
Critical zone, process 18268, 2

[1]+  Stopped                  ./sem_3a 2
UOC1: fg
./sem_3a 2
Critical zone, process 18267, 1
UOC1: █

```

Figura 5.10: Ejemplo de ejecución del programa './sem_3a'.

```

UOC1: man ipcs | head -13
IPCS(1)                                User Commands                                IPCS(1)

NAME
    ipcs - show information on IPC facilities

SYNOPSIS
    ipcs [options]

DESCRIPTION
    ipcs shows information on the inter-process communication facilities for which the
    calling process has read access. By default it shows information about all three re-
    sources: shared memory segments, message queues, and semaphore arrays.

UOC1: █

```

Figura 5.11: Manual de ayuda de la utilidad de sistema './ipcs'.

```

UOC1: ipcs

----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages

----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status

----- Semaphore Arrays -----
key      semid      owner      perms      nsems

UOC1: █

```

Figura 5.12: Ejecución de la utilidad de sistema './ipcs'.

En la Figura 5.13 vemos como en el directorio '/dev/shm' tenemos un fichero 'sem.sem_1' que es el que tiene la información del semáforo de nombre 'sem_1'.

```

UOC1: ls /dev/shm/
sem.sem_1
UOC1: █

```

Figura 5.13: Vemos el fichero creado para el semáforo 'sem_1' con nombre 'sem_sem_1' en el directorio del sistema '/dev/shm'.

Volvemos a ejecutar el proceso 'sem_3a' en modo *background*, para dejarlo parado. Hemos ejecutado el proceso (con PID 18381) y lo tenemos parado como vemos en la Figura 5.14.

```
UOC1: ./sem_3a 3 &
[1] 18381
UOC1: PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process 18381, 0

[1]+  Stopped                  ./sem_3a 3
```

Figura 5.14: Ejecución del programa 'sem_3a' con PID=18381.

Ahora podemos ver información del mismo con el comando 'more /proc/18381/maps' como vemos en la Figura 5.15. Vemos (con el filtro adecuado) como tiene acceso al semáforo '/dev/shm/sem.sem1'.

```
UOC1: more /proc/18381/maps |grep /dev
7efd1f502000-7efd1f503000 rw-s 00000000 00:28 4                /dev/shm/sem.sem1
UOC1: █
```

Figura 5.15: Información del proceso 18381 con el comando 'more /proc/18381/maps'.

Ahora lanzamos una segunda instancia del proceso como vemos en la Figura 5.16 con PID=18384.

```
UOC1: ./sem_3a 2 &
[2] 18384
UOC1: PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 0
Critical zone, process 18384, 0

[2]+  Stopped                  ./sem_3a 2
UOC1: █
```

Figura 5.16: Ejecución de la segunda instancia de 'sem_3a' con PID = 18384.

Vamos a instalar la utilidad 'lsof' con el comando 'apt-get install lsof' como vemos en la Figura 5.17.

```
UOC1: apt-get install lsof
Reading package lists... Done
Building dependency tree
```

Figura 5.17: Instalación del paquete 'lsof' con el comando 'apt-get install lsof'.

Al ejecutar el comando 'lsof /dev/shm/sem.sem1' como vemos en la Figura 5.17 vemos que tenemos dos procesos asociados con el semáforo 'sem1'; ambos son ejecu-

nes del programas 'sem_3a' uno con PID=18381, y el segundo con PID=18384 que son nuestras dos instancias en ejecución.

```
UOC1: lsof /dev/shm/sem.sem1
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
sem_3a   18381 root mem    REG  0,40      32    4 /dev/shm/sem.sem1
sem_3a   18384 root mem    REG  0,40      32    4 /dev/shm/sem.sem1
UOC1:
```

Figura 5.18: Ejecución del comando 'lsof /dev/shm/sem.sem1' para ver los procesos asociados con el semáforo 'sem1'.

El comando 'ltrace', que si es necesario tal y como vemos en la Figura 5.19 instalaremos con el comando 'apt-get install ltrace', nos permite seguir el estado de un proceso (saber si está bloqueado, parado...).

```
UOC1: apt-get install ltrace
Reading package lists... Done
Building dependency tree
```

Figura 5.19: Instalación del paquete 'ltrace' con el comando 'apt-get install ltrace'.

Nuestros dos procesos están ambos parados pues al ejecutar el comando 'ltrace -p 18381' y 'ltrace -p 18384' nos aparece el mensaje '--- SIGTTOU (Stopped (tty output)) ---' (de forma continuada en nuestro terminal).

De todos modos no es muy recomendable utilizar este comando pues envía señales de 'SIGSTOP', que provoca que las funciones que manejan semáforos devuelvan una condición de -1 con error tipo 'EINTR'. La solución es incluir en todas las llamadas una condición similar a la que tenemos en el Código 79.

```
1 do {
2   r = sem_wait(myem); //return -1 with errno=EINTR if interrupted
3 } while ((r==-1) && (errno=EINTR));
```

Código 79: Código para manejar las señales de 'SIGSTOP' en las llamadas a la función 'sem_wait()'.

En la Figura 5.20 vemos como podemos ver el semáforo 'sem1' asociado al proceso con PID=18381 con el comando 'pmap -d 18381'

Si queremos ejecutar varias instancias de un programa en paralelo, también podemos utilizar el comando 'parallel' que instalaremos (si fuera necesario) con el comando 'apt-get install parallel' tal y como vemos en la Figura 5.21.

```
UOC1: pmap -d 18381
18381: ./sem_3a 3
Address      Kbytes Mode Offset      Device      Mapping
0000000000400000 4 r---- 0000000000000000 008:00001 sem_3a
0000000000401000 4 r-x-- 0000000000000100 008:00001 sem_3a
0000000000402000 4 r---- 0000000000000200 008:00001 sem_3a
0000000000403000 4 r---- 0000000000000200 008:00001 sem_3a
0000000000404000 4 rw--- 0000000000000300 008:00001 sem_3a
00000000013ad000 132 rw--- 0000000000000000 000:00000 [ anon ]
00007efd1f312000 12 rw--- 0000000000000000 000:00000 [ anon ]
00007efd1f315000 136 r---- 0000000000000000 008:00001 libc-2.28.so
00007efd1f337000 1312 r-x-- 0000000000002200 008:00001 libc-2.28.so
00007efd1f47f000 304 r---- 0000000000016a00 008:00001 libc-2.28.so
00007efd1f4cb000 4 ---- 000000000001b600 008:00001 libc-2.28.so
00007efd1f4cc000 16 r---- 000000000001b600 008:00001 libc-2.28.so
00007efd1f4d0000 8 rw--- 000000000001ba00 008:00001 libc-2.28.so
00007efd1f4d2000 16 rw--- 0000000000000000 000:00000 [ anon ]
00007efd1f4d6000 24 r---- 0000000000000000 008:00001 libpthread-2.28.so
00007efd1f4dc000 60 r-x-- 0000000000006000 008:00001 libpthread-2.28.so
00007efd1f4eb000 24 r---- 0000000000001500 008:00001 libpthread-2.28.so
00007efd1f4f1000 4 r---- 0000000000001a00 008:00001 libpthread-2.28.so
00007efd1f4f2000 4 rw--- 0000000000001b00 008:00001 libpthread-2.28.so
00007efd1f4f3000 24 rw--- 0000000000000000 000:00000 [ anon ]
00007efd1f502000 4 rw-s- 0000000000000000 000:00028 sem.sem1
00007efd1f503000 4 r---- 0000000000000000 008:00001 ld-2.28.so
00007efd1f504000 120 r-x-- 0000000000000100 008:00001 ld-2.28.so
00007efd1f522000 32 r---- 0000000000001f00 008:00001 ld-2.28.so
00007efd1f52a000 4 r---- 0000000000002600 008:00001 ld-2.28.so
00007efd1f52b000 4 rw--- 0000000000002700 008:00001 ld-2.28.so
00007efd1f52c000 4 rw--- 0000000000000000 000:00000 [ anon ]
00007ffe34180000 132 rw--- 0000000000000000 000:00000 [ stack ]
00007ffe341e7000 8 r---- 0000000000000000 000:00000 [ anon ]
00007ffe341e9000 8 r-x-- 0000000000000000 000:00000 [ anon ]
fffffffff6000000 4 r-x-- 0000000000000000 000:00000 [ anon ]
mapped: 2424K writeable/private: 340K shared: 4K
UOC1: █
```

Figura 5.20: Monitorización de los semáforos asociados a un proceso con el comando 'pmap'.

```
UOC1: apt-get install parallel
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figura 5.21: Instalación del paquete 'parallel' con el comando 'apt-get install parallel'.

Ahora tenemos un primer programa ('sem_4a.c') para crear el semáforo ('sem_2') e inicializarlo a valor 1 (para que podamos entrar en el bucle en una de las instancias) tal y como vemos en el Código 80.

```
1  /*
2   * Filename: sem_4a.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
```

```
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD  0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 int main(int argc, char *argv[])
27 {
28     int    input;
29     char    buffer[10 + 2];
30     sem_t*  psem1;
31     int     sem_value;
32     char     key;
33     int     count;
34     int     result;
35
36     /* Parse program arguments */
37     if (argc != 1) {
38         fprintf(stderr, "Usage: %s\n", argv[0]);
39         exit(1);
40     }
41
42     /* Create psem1 */
43     psem1 = (sem_t*)sem_open("/sem_2", O_CREAT, 0600, 0);
44     if (psem1 == SEM_FAILED) {
45         err_sys("Open psem1");
46     }
47
48     /* Read and print psem1 */
49     result = sem_getvalue(psem1, &sem_value);
```

```

50     if (result < 0) {
51         err_sys("Read psem1");
52     }
53     printf("PROCESS 1(SEM1): %d\n", sem_value);
54
55     /* Wait for sem_value to be 0 */
56     while (sem_value > 0) {
57         sem_wait(psem1);
58         sem_value--;
59     }
60
61     /* Increment the value of semaphore to initialize it to 1 */
62     result = sem_post(psem1);
63     if (result < 0) {
64         err_sys("Post psem1");
65     }
66
67     /* Read and print psem1 */
68     result = sem_getvalue(psem1, &sem_value);
69     if (result < 0) {
70         err_sys("Read psem1");
71     }
72     printf("PROCESS 1(SEM1): %d\n", sem_value);
73
74     /* Close psem1 */
75     result = sem_close(psem1);
76     if (result != 0) {
77         err_sys("Close psem1");
78     }
79 }

```

Código 80: Código 'sem_4a.c' para crear e inicializar el semáforo '/sem_2'.

Entonces tenemos el programa que hace el bucle ('sem_4b.c'), y finalmente el programa para eliminar el semáforo ('sem_cleaning_name.c') si fuera necesario reinicializar los semáforos.

```

1  /*
2   * Filename: sem_4b.c
3   */

```

```
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9 #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD 0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 char inkey(void) {
27     char c;
28     struct termio param_ant, params;
29
30     ioctl(STDINFD, TCGETA, &param_ant);
31
32     params = param_ant;
33     params.c_lflag &= ~(ICANON | ECHO);
34     params.c_cc[4] = 1;
35
36     ioctl(STDINFD, TCSETA, &params);
37
38     fflush(stdin); fflush(stderr); fflush(stdout);
39     read(STDINFD, &c, 1);
40
41     ioctl(STDINFD, TCSETA, &param_ant);
42     return c;
43 }
44
45 int main(int argc, char *argv[])
```

```
46 {
47     int    input;
48     char    buffer[10 + 2];
49     sem_t*  psem1;
50     int     sem_value;
51     char    key;
52     int     count;
53     int     result;
54
55     /* Parse program arguments */
56     if (argc != 3) {
57         fprintf(stderr, "Usage: %s <iterations> <ID>\n", argv[0]);
58         exit(1);
59     }
60
61     /* Create psem1 */
62     psem1 = (sem_t*) sem_open("/sem_2", O_CREAT, 0600, 0);
63     if (psem1 == SEM_FAILED) {
64         err_sys("Open psem1");
65     }
66
67     for (count = 0; count < atoi(argv[1]); count++) {
68         fprintf(stdout, "Ready to execute WAIT, process %s, %d\n", argv[2], count);
69
70         /* Enter critical zone */
71         result = sem_wait(psem1);
72         if (result < 0) {
73             err_sys("Wait psem1");
74         }
75
76         /* In the critical zone */
77         fprintf(stdout, "Critical zone, process %s, %d\n", argv[2], count);
78
79         /* Exiting critical zone */
80         result = sem_post(psem1);
81         if (result < 0) {
82             err_sys("Post psem1");
83         }
84     }
85
86     /* Close psem1 */
87     result = sem_close(psem1);
```



```
88     if (result != 0) {
89         err_sys("Close psem1");
90     }
91 }
```

Código 81: Código 'sem_4b.c' para las diferentes instancias de nuestro programa de sincronización.

Con los comandos 'make sem_4a -f sem.Makefile' y 'make sem_4b -f sem.Makefile' generaremos los dos ejecutables nuevos ('sem_4a', 'sem_4b') respectivamente tal y como vemos en la Figura 5.22.

```
UOC1: make sem_4a -f sem.Makefile
gcc -c sem_4a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_4a sem_4a.o -I ./ -pthread
UOC1: make sem_4b -f sem.Makefile
gcc -c sem_4b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_4b sem_4b.o -I ./ -pthread
UOC1: █
```

Figura 5.22: Generación de los ejecutables 'sem_4a' y 'sem_4b'.

Empezamos creando el semáforo y le ponemos valor 1, como vemos en la Figura 5.23 con la ejecución de './sem_4a'.

```
UOC1: ./sem_4a
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 1
UOC1: █
```

Figura 5.23: Creación del semáforo 'sem_2' y puesta a valor 1 al ejecutar 'sem_4a' y 'sem_4a'.

Ahora, lanzamos dos instancias en paralelo como vemos en la Figura 5.24 con el comando 'parallel ::: './sem_4b 2 1" './sem_4b 2 2"'. El primer argumento del programa 'sem_4b' representa el número de iteraciones que haremos de impresión del mensaje y de acceso a la zona crítica, y el segundo de los parámetros es un argumento para indicar en el mensaje para identificar los procesos. En una de las llamadas lo pondremos a 1 y en la otra a 2. Hemos eliminado la petición de tecla al usuario para agilizar el programa.

De hecho si no tenemos varios cores en nuestro sistema no es efectivo el comando 'parallel' y por lo tanto es mejor lanzar los dos procesos en modo *background* como vemos en la Figura 5.25 donde vemos la alternancia entre ambas instancias.

```
UOC1: parallel ::: "./sem_4b 2 1" "./sem_4b 2 2"
Ready to execute WAIT, process 1, 0
Critical zone, process 1, 0
Ready to execute WAIT, process 1, 1
Critical zone, process 1, 1
Ready to execute WAIT, process 2, 0
Critical zone, process 2, 0
Ready to execute WAIT, process 2, 1
Critical zone, process 2, 1
UOC1: █
```

Figura 5.24: Ejecución de dos instancias en paralelo de 'sem_4b' con el comando 'parallel'.

```
UOC1: ./sem_4b 40 1 & ./sem_4b 40 2 &
[1] 19973
[2] 19974
UOC1: Ready to execute WAIT, process 2, 0
Ready to execute WAIT, process 1, 0
Critical zone, process 1, 0
Ready to execute WAIT, process 1, 1
Critical zone, process 1, 1
Ready to execute WAIT, process 1, 2
Critical zone, process 1, 2
Ready to execute WAIT, process 1, 3
Critical zone, process 2, 0
Ready to execute WAIT, process 2, 1
Critical zone, process 1, 3
Ready to execute WAIT, process 1, 4
Critical zone, process 2, 1
Ready to execute WAIT, process 2, 2
Critical zone, process 1, 4
```

Figura 5.25: Ejecución de dos instancias en paralelo de 'sem_4b' ambas en modo *background*.

Ahora vamos a trabajar con varios semáforos para poder decidir la ejecución de los programas. Si queremos alternar dos procesos en ejecución trabajaremos con dos semáforos, uno para controlar el proceso 1 y otro para controlar el proceso 2. Uno de los dos procesos se inicializa a 1, por ejemplo el del proceso 1. Antes de iniciar el acceso al recurso compartido el proceso 1 hará 'WAIT(1)' accederá al recurso y dará paso al proceso dos 'SIGNAL(2)' al finalizar. El proceso dos esperará la asignación 'WAIT(2)', accederá al recurso y dará paso al proceso 1 'SIGNAL(1)' al finalizar.

Ahora tendremos tres programas.

El que inicializa los dos semáforos 'sem_5a.c' que vemos en el Código 82. El programa tiene un argumento para el valor inicial del semáforo 'semaphore1' que controlará el acceso al recurso compartido para el proceso 1, y un segundo argumento para dar valor inicial al semáforo 'semaphore2' que controlará el acceso para el proceso 2. Los pondremos en alternancia: uno a valor 1 y el otro a valor 0, para decidir cuál de los dos procesos 1 o 2 es el primero en poder acceder al recurso compartido.

```
1  /*
2   * Filename: sem_5a.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD  0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 int main(int argc, char *argv[])
27 {
28     int    input;
29     char    buffer[10 + 2];
30     sem_t*  psem1;
31     sem_t*  psem2;
32     int     sem_value;
33     char     key;
34     int     count;
35     int     result;
36
37     /* Parse program arguments */
38     if (argc != 3) {
39         fprintf(stderr, "Usage: %s value_sem1 value_sem2\n", argv[0]);
40         exit(1);
41     }
```

```
42
43  /* Create psem1 */
44  psem1 = (sem_t*)sem_open("/semaphore1", O_CREAT, 0600, 0);
45  if (psem1 == SEM_FAILED) {
46      err_sys("Open psem1");
47  }
48
49  /* Read and print psem1 */
50  result = sem_getvalue(psem1, &sem_value);
51  if (result < 0) {
52      err_sys("Read psem1");
53  }
54  printf("PROCESS 1(SEM1): %d\n", sem_value);
55
56  /* Wait for sem_value to be 0 */
57  while (sem_value > 0) {
58      sem_wait(psem1);
59      sem_value--;
60  }
61
62  /* Repeat */
63  count = 0;
64  while (count < atoi(argv[1])) {
65      /* Increment the value of semaphore to initialize it to set argument */
66      count++;
67      if (sem_post(psem1) < 0) {
68          err_sys("Post psem1");
69      }
70  }
71
72  /* Read and print psem1 */
73  result = sem_getvalue(psem1, &sem_value);
74  if (result < 0) {
75      err_sys("Read psem1");
76  }
77  printf("PROCESS 1(SEM1): %d\n", sem_value);
78
79  /* Create psem2 */
80  psem2 = (sem_t*)sem_open("/semaphore2", O_CREAT, 0600, 0);
81  if (psem2 == SEM_FAILED) {
82      err_sys("Open psem2");
83  }
```

```
84
85  /* Read and print psem2 */
86  result = sem_getvalue(psem2, &sem_value);
87  if (result < 0) {
88      err_sys("Read psem2");
89  }
90  printf("PROCESS 1(SEM2): %d\n", sem_value);
91
92  /* Wait for sem_value to be 0 */
93  while (sem_value > 0) {
94      sem_wait(psem2);
95      sem_value--;
96  }
97
98  /* Repeat */
99  count = 0;
100 while (count < atoi(argv[2])) {
101     /* Increment the value of semaphore to initialize it to set argument */
102     count++;
103
104     /* Post psem2 */
105     result = sem_post(psem2);
106     if ( result < 0) {
107         err_sys("Post psem2");
108     }
109 }
110
111 /* Read and print psem2 */
112 result = sem_getvalue(psem2, &sem_value);
113 if (result < 0) {
114     err_sys("Read psem2");
115 }
116 printf("PROCESS 1(SEM2): %d\n", sem_value);
117
118 /* Close psem1 */
119 result = sem_close(psem1);
120 if (result != 0) {
121     err_sys("Close psem1");
122 }
123
124 /* Close psem2 */
125 result = sem_close(psem2);
```

```
126     if (result != 0) {
127         err_sys("Close psem2");
128     }
129 }
```

Código 82: Código 'sem_5a.c' para crear e inicializar los dos semáforos de nuestro problema de sincronización.

El primero de los procesos 'sem_5b.c' que vemos en el Código 83 en el bucle nos pedirá acceso a través del semáforo 'semaphore1' con la llamada 'result = sem_wait(psem1);' y al salir del bucle dará acceso al otro proceso con la llamada 'result = sem_post(psem2);'.

```
1  /*
2   * Filename: sem_5b.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD  0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 char inkey(void) {
27     char c;
28     struct termio param_ant, params;
```

```
29
30     ioctl(STDINFD, TCGETA, &param_ant);
31
32     params = param_ant;
33     params.c_lflag &= ~(ICANON | ECHO);
34     params.c_cc[4] = 1;
35
36     ioctl(STDINFD, TCSETA, &params);
37
38     fflush(stdin); fflush(stderr); fflush(stdout);
39     read(STDINFD, &c, 1);
40
41     ioctl(STDINFD, TCSETA, &param_ant);
42     return c;
43 }
44
45 int main(int argc, char *argv[])
46 {
47     int     input;
48     char    buffer[10 + 2];
49     sem_t*  psem1;
50     sem_t*  psem2;
51     int     sem_value;
52     char    key;
53     int     count;
54     int     result;
55
56     /* Parse program arguments */
57     if (argc != 3) {
58         fprintf(stderr, "USAGE: %s <iterations> <ID>\n", argv[0]);
59         exit(1);
60     }
61
62     /* Create psem1 */
63     psem1 = (sem_t*) sem_open("/semaphore1", O_CREAT, 0600, 0);
64     if (psem1 == SEM_FAILED) {
65         err_sys("Open psem1");
66     }
67
68     /* Create psem2 */
69     psem2 = (sem_t*) sem_open("/semaphore2", O_CREAT, 0600, 0);
70     if (psem2 == SEM_FAILED) {
```

```
71     err_sys("Open psem2");
72 }
73
74 /* Repeat */
75 for (count = 0; count < atoi(argv[1]); count++) {
76     fprintf(stdout, "Ready to execute WAIT, process %s, %d\n", argv[2], count);
77
78     /* Entering critical zone */
79     result = sem_wait(psem1);
80     if (result < 0) {
81         err_sys("Wait psem1");
82     }
83
84     /* Just in the critical zone */
85     fprintf(stdout, "Critical zone, process %s, %d\n", argv[2], count);
86
87     /* Exiting critical zone */
88     result = sem_post(psem2);
89     if (result < 0) {
90         err_sys("Signal psem2");
91     }
92 }
93
94 /* Close psem1 */
95 result = sem_close(psem1);
96 if (result != 0) {
97     err_sys("Close psem1");
98 }
99
100 /* Close psem2 */
101 result = sem_close(psem2);
102 if (result != 0) {
103     err_sys("Close psem2");
104 }
105 }
```

Código 83: Código 'sem_5b.c' para la instancia del proceso 1 de nuestro problema de sincronización.

El segundo de los procesos 'sem_5c.c' que vemos en el Código 84 en el bucle nos pedirá acceso a través del semáforo 'semaphore2' con la llamada 'result = sem_wait(psem2);'

y al salir del bucle dará acceso al otro proceso con la llamada `'result = sem_post(psem1);'`.

```
1  /*
2   * Filename: sem_5c.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #define STDINFD  0
19
20 void err_sys(const char* text)
21 {
22     perror(text);
23     exit(1);
24 }
25
26 char inkey(void) {
27     char c;
28     struct termio param_ant, params;
29
30     ioctl(STDINFD, TCGETA, &param_ant);
31
32     params = param_ant;
33     params.c_lflag &= ~(ICANON | ECHO);
34     params.c_cc[4] = 1;
35
36     ioctl(STDINFD, TCSETA, &params);
37
38     fflush(stdin); fflush(stderr); fflush(stdout);
39     read(STDINFD, &c, 1);
40 }
```

```
41     ioctl(STDINFD, TCSETA, &param_ant);
42     return c;
43 }
44
45 int main(int argc, char *argv[])
46 {
47     int     input;
48     char    buffer[10 + 2];
49     sem_t*  psem1;
50     sem_t*  psem2;
51     int     sem_value;
52     char    key;
53     int     count;
54     int     result;
55
56     /* Parse program arguments */
57     if (argc != 3) {
58         fprintf(stderr, "Usage: %s <iterations> <ID>\n", argv[0]);
59         exit(1);
60     }
61
62     /* Create psem1 */
63     psem1 = (sem_t*)sem_open("/semaphore1", O_CREAT, 0600, 0);
64     if (psem1 == SEM_FAILED) {
65         err_sys("Open psem1");
66     }
67
68     /* Create psem2 */
69     psem2 = (sem_t*)sem_open("/semaphore2", O_CREAT, 0600, 0);
70     if (psem2 == SEM_FAILED) {
71         err_sys("Open psem2");
72     }
73
74     /* Repeat */
75     for (count = 0; count < atoi(argv[1]); count++) {
76         fprintf(stdout, "Ready to execute WAIT, process %s, %d\n", argv[2], count);
77
78         /* Entering critical zone */
79         result = sem_wait(psem2);
80         if (result < 0) {
81             err_sys("Wait psem2");
82         }
```

```
83
84     /* Just in the critical zone */
85     fprintf(stdout, "Critical zone, process %s, %d\n", argv[2], count);
86
87     /* Exiting critical zone */
88     result = sem_post(psem1);
89     if (result < 0) {
90         err_sys("Post psem1");
91     }
92 }
93
94 /* Close psem1 */
95 result = sem_close(psem1);
96 if (result != 0) {
97     err_sys("Close psem1");
98 }
99
100 /* Close psem2 */
101 result = sem_close(psem2);
102 if (result != 0) {
103     err_sys("Close psem2");
104 }
105 }
```

Código 84: Código 'sem_5c.c' para la instancia del proceso 2 de nuestro problema de sincronización.

Generamos los tres ejecutables 'sem_5a', 'sem_5b' y 'sem_5c' con el comando 'make' y las opciones adecuadas como vemos en la Figura 5.26.

```
UOC1: make sem_5a -f sem.Makefile
gcc -c sem_5a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_5a sem_5a.o -I ./ -pthread
UOC1: make sem_5b -f sem.Makefile
gcc -c sem_5b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_5b sem_5b.o -I ./ -pthread
UOC1: make sem_5c -f sem.Makefile
gcc -c sem_5c.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_5c sem_5c.o -I ./ -pthread
UOC1: █
```

Figura 5.26: Generación de los ejecutables para 'sem_5a', 'sem_5b' y 'sem_5c'.

Inicializamos los semáforos, el primero a 1 (empezará el proceso 1) y el segundo a 0, como vemos en la Figura 5.27 con el comando './sem_5a 1 0'.

```
UOC1: ./sem_5a 1 0
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 1
PROCESS 1(SEM2): 0
PROCESS 1(SEM2): 0
UOC1: █
```

Figura 5.27: Inicializamos el problema con el semáforo 'semaphore1' a 1, y el semáforo 'semaphore2' a 0 con el comando './sem_5a 1 0'.

Lanzamos los dos procesos en modo *background* para que hagan tres iteraciones, el primero con el comando './sem_5b 3 1 &' para identificarlo como proceso 1, y el segundo con el comando './sem_5c 3 2 &' para identificarlo como proceso 2. Como vemos en la Figura 5.28 los dos procesos se ejecutarán en alternancia, tres veces, empezando por el proceso 1.

```
UOC1: ./sem_5b 3 1 & ./sem_5c 3 2 &
[1] 20081
[2] 20082
UOC1: Ready to execute WAIT, process 2, 0
Ready to execute WAIT, process 1, 0
Critical zone, process 1, 0
Ready to execute WAIT, process 1, 1
Critical zone, process 2, 0
Ready to execute WAIT, process 2, 1
Critical zone, process 1, 1
Ready to execute WAIT, process 1, 2
Critical zone, process 2, 1
Ready to execute WAIT, process 2, 2
Critical zone, process 1, 2
Critical zone, process 2, 2

[1]- Done          ./sem_5b 3 1
[2]+ Done          ./sem_5c 3 2
UOC1: █
```

Figura 5.28: Ejecución de 'sem_5b' y 'sem_5c' en alternancia empezando por el proceso 1.

Ahora modificamos el código para dejar sólo un mensaje en la zona crítica, para que quede clara la alternancia de procesos

El programa 'sem_6a.c' es idéntico al programa 'sem_5a.c'.

En el primer proceso simplificamos el bucle con una sola impresión como vemos en el Código 85.

En el segundo proceso simplificamos el bucle con una sola impresión como vemos en el Código 86.

Generamos los tres ejecutables 'sem_6a', 'sem_6b' y 'sem_6c' con el comando 'make' y las opciones adecuadas como vemos en la Figura 5.29.

Inicializamos los semáforos, el primero a 1 (empezará el proceso 1) y el segundo a 0, como vemos en la Figura 5.30 con el comando './sem_6a 1 0'.

```

74  /* Repeat */
75  for (count = 0; count < atoi(argv[1]); count++) {
76      /* Entering critical zone */
77      result = sem_wait(psem1);
78      if (result < 0) {
79          err_sys("Wait psem1");
80      }
81
82      /* Critical zone */
83      fprintf(stdout, "Critical zone, process %s, %d\n", argv[2], count);
84
85      /* Exiting critical zone */
86      result = sem_post(psem2);
87      if (result < 0) {
88          err_sys("Signal psem2");
89      }
90  }

```

Código 85: Bucle del Código 'sem_6b.c' para la instancia del proceso 1 de nuestro problema de sincronización.

```

UOC1: make sem_6a -f sem.Makefile
gcc -c sem_6a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_6a sem_6a.o -I ./ -pthread
UOC1: make sem_6b -f sem.Makefile
gcc -c sem_6b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_6b sem_6b.o -I ./ -pthread
UOC1: make sem_6c -f sem.Makefile
gcc -c sem_6c.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o sem_6c sem_6c.o -I ./ -pthread
UOC1:

```

Figura 5.29: Generación de los ejecutables para 'sem_6a', 'sem_6b' y 'sem_6c'.

```

UOC1: ./sem_6a 1 0
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 1
PROCESS 1(SEM2): 0
PROCESS 1(SEM2): 0
UOC1:

```

Figura 5.30: Inicializamos el problema con el semáforo 'semaphore1' a 1, y el semáforo 'semaphore2' a 0 con el comando './sem_6a 1 0'.

Lanzamos los dos procesos en modo *background* para que hagan tres iteraciones, el primero con el comando './sem_6b 3 1 &' para identificarlo como proceso 1, y el segundo con el comando './sem_6c 3 2 &' para identificarlo como proceso 2. Como vemos en la Figura 5.31 los dos procesos se ejecutarán en alternancia, tres veces, empezando por el

```

66     for (count = 0; count < atoi(argv[1]); count++) {
67         if (sem_wait(psem2) < 0) err_sys("WAIT psem2");
68         //just in the critical zone
69         fprintf(stdout, "critical zone, process %s, %d\n", argv[2], count);
70         //exiting critical zone
71         if (sem_post(psem1) < 0) err_sys("SIGNAL psem1");
72     }

```

Código 86: Bucle del Código 'sem_6c.c' para la instancia del proceso 2 de nuestro problema de sincronización.

proceso 1.

```

UOC1: ./sem_6b 3 1 & ./sem_6c 3 2 &
[1] 20134
[2] 20135
UOC1: Critical zone, process 1, 0
critical zone, process 2, 0
Critical zone, process 1, 1
critical zone, process 2, 1
Critical zone, process 1, 2
critical zone, process 2, 2

[1]- Done          ./sem_6b 3 1
[2]+ Done          ./sem_6c 3 2
UOC1: █

```

Figura 5.31: Ejecución de 'sem_6b' y 'sem_6c' en alternancia empezando por el proceso 1.

Ahora los ejecutamos en el orden cambiado como vemos en la Figura 5.32. Primero inicializamos los semáforos con el comando './sem_6a 0 1', para que empiece el proceso 2. Y lanzamos las dos instancias en modo *background* para que hagan 4 iteraciones con los comandos './sem_6b 4 1 &' y './sem_6c 4 2 &'.

5.3. *Mutex*

Ahora podemos trabajar como mecanismo de sincronización con *mutex* (pensado para manejar *threads*). Son el mecanismo que permite garantizar el acceso a un recurso en forma de exclusión mutua.

De forma similar a los semáforos sobre los *mutex* se definen dos operaciones:

- 'LOCK': Esta petición es similar a la operación 'WAIT'. Se realiza en el momento que un *thread* quiera acceder a un recurso compartido. Si nadie lo está utilizando bloqueará el recurso y lo utilizará. En caso que otro *thread* esté accediendo al recurso

```

UOC1: ./sem_6a 0 1
PROCESS 1(SEM1): 1
PROCESS 1(SEM1): 0
PROCESS 1(SEM2): 0
PROCESS 1(SEM2): 1
UOC1: ./sem_6b 4 1 & ./sem_6c 4 2 &
[1] 20137
[2] 20138
UOC1: critical zone, process 2, 0
Critical zone, process 1, 0
critical zone, process 2, 1
Critical zone, process 1, 1
critical zone, process 2, 2
Critical zone, process 1, 2
critical zone, process 2, 3
Critical zone, process 1, 3

[1]- Done          ./sem_6b 4 1
[2]+ Done          ./sem_6c 4 2
UOC1:

```

Figura 5.32: Ejecución de 'sem_6b' y 'sem_6c' en alternancia empezando por el proceso 2.

el *thread* que ha hecho la petición quedará bloqueado hasta que el recurso quede liberado.

- 'UNLOCK': Es la operación similar al 'SIGAL(POST)' de los semáforos. Se hace uso cuando el *thread* que está utilizando el recurso compartido va a dejar de utilizarlo para avisar a los demás *thread* que podrían utilizarlo.

Evidentemente ambas operaciones son atómicas.

Para poder utilizar el *mutex*, lo primero será inicializarlo, con la llamada a la función 'pthread_mutex_init()'. Si vamos a utilizar los parámetros por defecto, podemos utilizar la macro 'pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;' como haremos en nuestros ejemplos. Como los *threads* comparten variables globales, la definiremos como tal en nuestro programa.

En nuestro primer ejemplo, que tenemos en el Código 87, vamos a replicar el contador de los ejemplos de semáforos pero ahora con *mutex*. Para ello tendremos un *thread*, que iterará con bloqueo ('pthread_mutex_lock (&mutex);') y desbloqueo en cada iteración con ('pthread_mutex_unlock(&mutex);') de *mutex*.

```

1  /*
2   * Filename: mutex1.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>

```

```
8  #include <fcntl.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <semaphore.h>
12 #include <unistd.h>
13 #include <sys/types.h>
14
15 #include <sys/ioctl.h>
16 #include <termio.h>
17
18 #include <pthread.h>
19
20 #define STDINFD  0
21
22 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
23
24 void err_sys(const char* text)
25 {
26     perror(text);
27     exit(1);
28 }
29
30 char inkey(void) {
31     char c;
32     struct termio param_ant, params;
33
34     ioctl(STDINFD, TCGETA, &param_ant);
35
36     params = param_ant;
37     params.c_lflag &= ~(ICANON | ECHO);
38     params.c_cc[4] = 1;
39
40     ioctl(STDINFD, TCSETA, &params);
41
42     fflush(stdin); fflush(stderr); fflush(stdout);
43     read(STDINFD, &c, 1);
44
45     ioctl(STDINFD, TCSETA, &param_ant);
46     return c;
47 }
48
49 void *handle_thread(void *vargp)
```

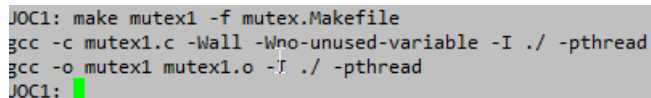


```
50 {
51     int *myiterations = (int *)vargp;
52     int count;
53
54     /* Print thread information */
55     printf("Executing handle_thread for %d iterations \n", *myiterations);
56     fprintf(stdout, "Thread ID in handler %lu\n", pthread_self());
57
58     /* Repeat */
59     for (count = 0; count < (*myiterations); count++) {
60         /* Aquire mutex */
61         pthread_mutex_lock (&mutex);
62
63         /* In critical zone */
64         fprintf(stdout, "critical zone, thread %lu %d\n", pthread_self(), count);
65
66         /* Wait user type any key to continue */
67         inkey();
68
69         /* Release mutex */
70         pthread_mutex_unlock(&mutex);
71     }
72
73     return NULL;
74 }
75
76 int main(int argc, char *argv[])
77 {
78     int    input;
79     char    buffer[10 + 2];
80     char    key;
81     int     iterations;
82     int     errno = 0;
83     int     result;
84
85     pthread_t handle_thread_id;
86
87     /* Parse program arguments */
88     if (argc != 2) {
89         fprintf(stderr, "Usage: %s <iterations>\n", argv[0]);
90         exit(1);
91     }
```

```
92
93     iterations = atoi(argv[1]);
94
95     /* Create and print thread */
96     pthread_create(&handle_thread_id, NULL, handle_thread, (void *)&iterations);
97     fprintf(stdout, "Thread ID for handler %lu\n", handle_thread_id);
98
99     /* Wait for thread to complete */
100    result = pthread_join(handle_thread_id, NULL);
101    if (result > 0) {
102        fprintf(stderr, "Joining thread handle_thread error: errno = %d\n", errno);
103        exit(1);
104    }
105
106    exit(0);
107 }
```

Código 87: Código del ejemplo 'mutex1.c'.

Generamos el fichero ejecutable 'mutex1' con el comando 'make mutex1 -f mutex.Makefile' como vemos en la Figura 5.33.



```
JOC1: make mutex1 -f mutex.Makefile
gcc -c mutex1.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex1 mutex1.o -f ./ -pthread
JOC1: █
```

Figura 5.33: Generación del ejecutable para 'mutex1'.

Ejecutamos una instancia del hilo de ejecución, con tres iteraciones de impresión como vemos en la Figura 5.34. Después de cada impresión (acceso a la zona crítica) se nos pide que pulsemos una tecla como anteriormente en los ejemplos de semáforos. En cada mensaje imprimimos el identificador del *thread* y el valor del contador.

Ahora vamos a crear varios *threads*, que van a ejecutarse en paralelo, como vemos en el Código 88.

Generamos el fichero ejecutable 'mutex2' con el comando 'make mutex2 -f mutex.Makefile' como vemos en la Figura 5.35.

Ejecutamos una instancia del hilo de ejecución, con dos iteraciones de impresión, y 4 hilos de ejecución como vemos en la Figura 5.36. Después de cada impresión (acceso a la zona crítica) se nos pide que pulsemos una tecla. En cada mensaje imprimimos

```

UOC1: ./mutex1
Usage: ./mutex1 <iterations>
UOC1: ./mutex1 3
Thread ID for handler 139741824304896
Executing handle_thread for 3 iterations
Thread ID in handler 139741824304896
critical zone, thread 139741824304896 0
critical zone, thread 139741824304896 1
critical zone, thread 139741824304896 2
UOC1:

```

Figura 5.34: Ejecución del programa 'mutex1'.

```

98  /* Create and print threads */
99  for (count = 0; count < threads; count++) {
100      pthread_create(&handle_thread_id[count], NULL, handle_thread, (void
    ↪ *)&iterations);
101      fprintf(stdout, "Thread ID for handler %lu\n", handle_thread_id[count]);
102  }

```

Código 88: Código del ejemplo 'mutex2.c' donde vemos como paralelizamos varios hilos de ejecución.

```

UOC1: make mutex2 -f mutex.Makefile
gcc -c mutex2.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex2 mutex2.o -I ./ -pthread
UOC1:

```

Figura 5.35: Generación del ejecutable para 'mutex2'.

el identificador del *thread* y el valor del contador. Como vemos creamos 4 hilos de ejecución con los identificadores 140155888285440, 140155879892736, 140155871500032 y 140155863107328, que vemos que se ejecutan de forma secuencial (en orden hilo de ejecución 1, 2, 3 y 4, y para cada uno iteración 0, 1).

Para poder gestionar los *threads*, de forma sencilla hemos creado un 'array' para hasta 10 *threads*. No pueden ejecutarse más de 10 *threads* en esta implementación. Por sencillez no hacemos comprobaciones al respecto.

Ahora vamos a eliminar la petición de tecla por parte del usuario, para poder iterar en mayor cantidad el bucle.

Ahora vamos a crear varios *threads*, que van a ejecutarse en paralelo, como vemos en el Código 89

Generamos el fichero ejecutable 'mutex3' con el comando 'make mutex3 -f mutex.Makefile' como vemos en la Figura 5.37.

Ejecutamos una instancia del hilo de ejecución, con tres iteraciones de impresión,

```
UOC1: ./mutex2
Usage: ./mutex2 <threads> <iterations>
UOC1: ./mutex2 4 2
Thread ID for handler 140155888285440
Thread ID for handler 140155879892736
Thread ID for handler 140155871500032
Thread ID for handler 140155863107328
Executing handle_thread for 2 iterations
Thread ID in handler 140155863107328
Critical zone, thread 140155863107328 0
Executing handle_thread for 2 iterations
Thread ID in handler 140155871500032
Executing handle_thread for 2 iterations
Thread ID in handler 140155879892736
Executing handle_thread for 2 iterations
Thread ID in handler 140155888285440
Critical zone, thread 140155863107328 1
Critical zone, thread 140155879892736 0
Critical zone, thread 140155879892736 1
Critical zone, thread 140155871500032 0
Critical zone, thread 140155871500032 1
Critical zone, thread 140155888285440 0
Critical zone, thread 140155888285440 1
UOC1: █
```

Figura 5.36: Ejecución del programa 'mutex2'.

```
UOC1: make mutex3 -f mutex.Makefile
gcc -c mutex3.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex3 mutex3.o -I ./ -pthread
UOC1: █
```

Figura 5.37: Generación del ejecutable para 'mutex3'.

y 2 hilos de ejecución como vemos en la Figura 5.38. En cada mensaje imprimimos el identificador del *thread* y el valor del contador. Como vemos creamos 2 hilos de ejecución con los identificadores 140131542288128 y 140131533895424, que vemos que se ejecutan de forma secuencial (en orden hilo de ejecución 1 y 2, y para cada uno iteración 0, 1 y 2).

```
UOC1: ./mutex3 2 3
Thread ID for handler 140131542288128
Thread ID for handler 140131533895424
Executing handle_thread for 3 iterations
Thread ID in handler 140131533895424
Critical zone, thread 140131533895424 0
Critical zone, thread 140131533895424 1
Critical zone, thread 140131533895424 2
Executing handle_thread for 3 iterations
Thread ID in handler 140131542288128
Critical zone, thread 140131542288128 0
Critical zone, thread 140131542288128 1
Critical zone, thread 140131542288128 2
UOC1: █
```

Figura 5.38: Ejecución del programa 'mutex3'.

Ahora vamos a aumentar el número de iteraciones. Con un número suficientemente alto (300 en este ejemplo) veremos como conmutamos de un *thread* a otro en alguna

```
38     for (count = 0; count < (*myiterations); count++) {
39         /* Acquire mutex */
40         pthread_mutex_lock(&mutex);
41
42         /* In critical zone */
43         fprintf(stdout, "Critical zone, thread %lu %d\n", pthread_self(), count);
44
45         /* Release mutex */
46         pthread_mutex_unlock(&mutex);
47     }
```

Código 89: Código del ejemplo 'mutex3.c' donde vemos como ya no pedimos pulsación de tecla.

ocasión, como vemos en Figura 5.39, Figura 5.40, Figura 5.41 y Figura 5.42.

```
UOC1: ./mutex3 2 300
Thread ID for handler 139716166403840
Thread ID for handler 139716158011136
Executing handle_thread for 300 iterations
Thread ID in handler 139716158011136
Critical zone, thread 139716158011136 0
Critical zone, thread 139716158011136 1
Critical zone, thread 139716158011136 2
Critical zone, thread 139716158011136 3
```

Figura 5.39: Ejecución del programa 'mutex3': empezamos primer hilo de ejecución.

```
Critical zone, thread 139716158011136 174
Critical zone, thread 139716158011136 175
Executing handle_thread for 300 iterations
Thread ID in handler 139716166403840
Critical zone, thread 139716158011136 176
Critical zone, thread 139716158011136 177
Critical zone, thread 139716158011136 178
Critical zone, thread 139716166403840 0
Critical zone, thread 139716166403840 1
Critical zone, thread 139716166403840 2
Critical zone, thread 139716166403840 3
```

Figura 5.40: Ejecución del programa 'mutex3': conmutación al segundo hilo de ejecución.

Dado que los *threads* comparten memoria global, en el programa 'mutex4.c' vamos a hacer que imprimen al valor de una variable global que iremos incrementando a cada vuelta del bucle, tal y como vemos en el Código 90.

Generamos el fichero ejecutable 'mutex4' con el comando 'make mutex4 -f mutex.Makefile' como vemos en la Figura 5.43.

Ejecutamos una instancia del hilo de ejecución, con dos iteraciones de impresión, y 4 hilos de ejecución como vemos en la Figura 5.44. En cada impresión (acceso a la

```
Critical zone, thread 139716166403840 297
Critical zone, thread 139716166403840 298
Critical zone, thread 139716166403840 299
Critical zone, thread 139716158011136 179
Critical zone, thread 139716158011136 180
Critical zone, thread 139716158011136 181
Critical zone, thread 139716158011136 182
Critical zone, thread 139716158011136 183
Critical zone, thread 139716158011136 184
Critical zone, thread 139716158011136 185
```

Figura 5.41: Ejecución del programa 'mutex3': volvemos al primer hilo de ejecución.

```
Critical zone, thread 139716158011136 294
Critical zone, thread 139716158011136 295
Critical zone, thread 139716158011136 296
Critical zone, thread 139716158011136 297
Critical zone, thread 139716158011136 298
Critical zone, thread 139716158011136 299
UOC1: █
```

Figura 5.42: Ejecución del programa 'mutex3': finalización del primer hilo de ejecución.

```
UOC1: make mutex4 -f mutex.Makefile
gcc -c mutex4.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex4 mutex4.o -I ./ -pthread
UOC1: █
```

Figura 5.43: Generación del ejecutable para 'mutex4'.

zona crítica) vamos mostrando el valor de la variable global 'value' que como vamos se va incrementando de 0 a 7, en cada uno de los procesos (instancias de nuestros dos hilos de ejecución). Como vemos creamos 4 hilos de ejecución con los identificadores 140106502268672, 140106493875968, 140106485483264 y 140106477090560, que vemos que se ejecutan de forma secuencial (en orden hilo de ejecución 1, 2, 3 y 4, y para cada uno iteración 0, 1).

Como vemos los *mutex* son un mecanismo de exclusión mutua, no tanto de sincronización, con sus dos operaciones de bloqueo y desbloqueo del acceso a la zona crítica. No podemos ir alternando las ejecuciones, pues el proceso que bloquea el *mutex* debe ser el que lo tiene que desbloquear.

Si queremos alternancia, algo más parecido a la sincronización que obtenemos con semáforos podemos utilizar variables condicionales.

Las variables condicionales tienen dos operaciones básicas:

- 'C_WAIT': bloqueará el *thread* que ejecuta la llamada, lo expulsa del *mutex* en el que está asociada la variable condicional y permite que otro *thread* adquiera el control del *mutex*. Todo ello se realiza de forma atómica.
- 'C_SIGNAL': desbloquea el *thread* bloqueado en la variable condicional, compiten-

41

/* Repeat */

Código 90: Código del ejemplo 'mutex4.c' donde vemos como comparten la variable global 'value'.

```
UOC1: ./mutex4 4 2
Thread ID for handler 140106502268672
Thread ID for handler 140106493875968
Thread ID for handler 140106485483264
Thread ID for handler 140106477090560
Executing handle_thread for 2 iterations
Thread ID in handler 140106477090560
critical zone, thread 140106477090560 0---0
critical zone, thread 140106477090560 1---1
Executing handle_thread for 2 iterations
Thread ID in handler 140106485483264
critical zone, thread 140106485483264 0---2
critical zone, thread 140106485483264 1---3
Executing handle_thread for 2 iterations
Thread ID in handler 140106502268672
critical zone, thread 140106502268672 0---4
critical zone, thread 140106502268672 1---5
Executing handle_thread for 2 iterations
Thread ID in handler 140106493875968
critical zone, thread 140106493875968 0---6
critical zone, thread 140106493875968 1---7
UOC1: █
```

Figura 5.44: Ejecución del programa 'mutex4'.

do de nuevo por el *mutex*.

Ahora incorporaremos una variante, con el programa 'mutex5.c' con dos *threads* controlados en las funciones 'handle_thread1()' y 'handle_thread2()', variables condicionales para garantizar la ejecución en alternancia de los dos *threads* 'cond_var1' y 'cond_var2'.

El uso de las variables condicionales nos bloquea el proceso sin consumir CPU. Pero debemos añadir como vemos en el ejemplo normalmente una validación: Tenemos un *mutex* y dos variables condicionales.

La rutina del primer *thread*, que vemos en el Código 91 comprobará el valor de 'current_thread'; si es 2 le pasará el control al segundo *thread* para garantizar la alternancia. El programa inicializa la variable global para que empiece el primer *thread* con 'int current_thread = 1;', al crear la variable global.

La rutina del segundo *thread*, que vemos en el Código 92 tendrá la condición cambiada, y alternará también las dos variables condicionales.

El programa principal lo vemos en el Código 93

Generamos el fichero ejecutable 'mutex5' con el comando 'make mutex5 -f mutex.Makefile' como vemos en la Figura 5.45.

```
UOC1: make mutex5 -f mutex.Makefile
gcc -c mutex5.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex5 mutex5.o -I ./ -pthread
UOC1: █
```

Figura 5.45: Generación del ejecutable para 'mutex5'.

Ejecutamos una instancia del hilo de ejecución, con cuatro iteraciones de impresión, y nuestros dos hilos de ejecución como vemos en la Figura 5.46. En cada impresión (acceso a la zona crítica) vamos mostrando el valor de la variable global 'value' que como vamos se va incrementando de 0 a 7, en cada uno de los procesos (instancias de nuestros dos hilos de ejecución). Como vemos creamos 2 hilos de ejecución con los identificadores 139784647284480 y 139784638891776, que vemos que se ejecutan de forma alternada (en orden hilo de ejecución 1-2, para cada iteración 0, 1, 2, 3).

```
UOC1: ./mutex5
Usage: ./mutex5 <iterations>
UOC1: ./mutex5 4
Thread ID for handler 139784647284480
Thread ID for handler 139784638891776
Executing handle_thread for 4 iterations
Thread ID in handler 139784638891776
Executing handle_thread for 4 iterations
Thread ID in handler 139784647284480
Critical zone, thread 139784647284480 0---0
Critical zone, thread 139784638891776 0---1
Critical zone, thread 139784647284480 1---2
Critical zone, thread 139784638891776 1---3
Critical zone, thread 139784647284480 2---4
Critical zone, thread 139784638891776 2---5
Critical zone, thread 139784647284480 3---6
Critical zone, thread 139784638891776 3---7
UOC1: █
```

Figura 5.46: Ejecución del programa 'mutex5'.

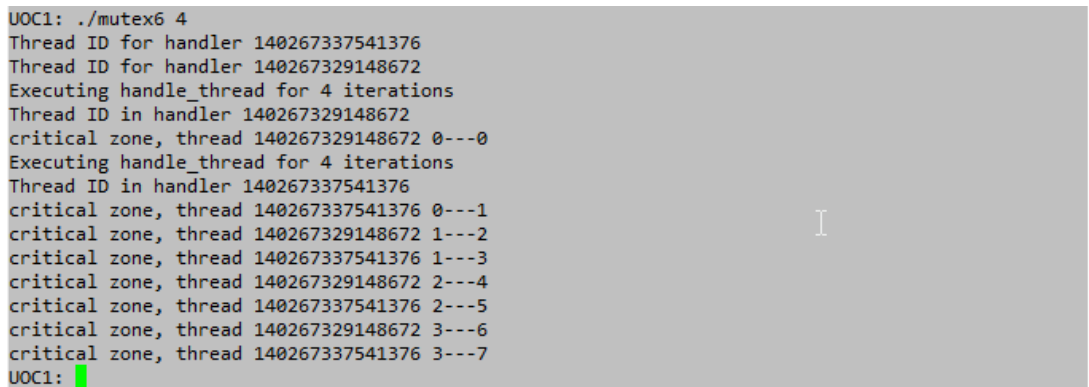
Ahora cambiamos el orden de ejecución de los hilos de ejecución en el programa 'mutex6.c', como vemos en el Código 94

Generamos el fichero ejecutable 'mutex6' con el comando 'make mutex6 -f mutex.Makefile' como vemos en la Figura 5.47.

```
UOC1: make mutex6 -f mutex.Makefile
gcc -c mutex6.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mutex6 mutex6.o -I ./ -pthread
UOC1: █
```

Figura 5.47: Generación del ejecutable para 'mutex6'.

Ejecutamos una instancia del hilo de ejecución, con cuatro iteraciones de impresión, y nuestros dos hilos de ejecución como vemos en la Figura 5.48. En cada impresión (acceso a la zona crítica) vamos mostrando el valor de la variable global 'value' que como vamos se va incrementando de 0 a 7, en cada uno de los procesos (instancias de nuestros dos hilos de ejecución). Como vemos creamos 2 hilos de ejecución con los identificadores 140267337541376 y 140267329148672, que vemos que se ejecutan de forma alternada (en orden hilo de ejecución 2-1, para cada iteración 0, 1, 2, 3).



```
UOC1: ./mutex6 4
Thread ID for handler 140267337541376
Thread ID for handler 140267329148672
Executing handle_thread for 4 iterations
Thread ID in handler 140267329148672
critical zone, thread 140267329148672 0---0
Executing handle_thread for 4 iterations
Thread ID in handler 140267337541376
critical zone, thread 140267337541376 0---1
critical zone, thread 140267329148672 1---2
critical zone, thread 140267337541376 1---3
critical zone, thread 140267329148672 2---4
critical zone, thread 140267337541376 2---5
critical zone, thread 140267329148672 3---6
critical zone, thread 140267337541376 3---7
UOC1: █
```

Figura 5.48: Ejecución del programa 'mutex6'.

5.4. Conclusiones

Hemos visto como forzar la sincronización entre procesos con el uso de semáforos (en un entorno 'POSIX') y *mutex*.

Algunas consideraciones importantes a la hora de trabajar con semáforos que debemos recordar, a modo de conclusión del capítulo son:

1. Todos los ejemplos que hemos puesto son en un entorno de semáforos en sistema 'POSIX'.
2. Todos los ejemplos que hemos puesto son de semáforos con nombre, que son el entorno habitual para sincronizar procesos/programas diferentes, pues el nombre es lo que nos va a permitir identificar el mismo semáforo en los dos programas.
3. Siempre es conveniente disponer de una solución para verificar el valor inicial de los semáforos en nuestros programas, y de volver a ponerlos al valor que nos interesa. En caso contrario un programa correcto puede no parecerlo. Recordemos que los

semáforos vienen a ser variables globales, pero no de nuestro proceso que se reinicializa a cada ejecución, sino del sistema operativo, y por lo tanto sólo se reinicializa con la reinicialización del sistema, o con una llamada a una función que lo haga, como hemos visto en las diferentes soluciones que hemos implantado.

4. Un único semáforo sólo garantiza que dos o más procesos no acceden a una zona crítica al mismo tiempo, pero no garantiza el orden de ejecución/acceso a la zona crítica de los diferentes procesos; para ello necesitaremos más semáforos, normalmente uno por cada proceso que deseemos sincronizar.
5. Hemos visto diferentes soluciones para ver los procesos asociados a semáforos.

A continuación hemos visto como los *mutex* son un mecanismo de bloqueo de acceso a una zona crítica, destacando las siguientes características:

1. Los *mutex* son un mecanismo más de bloqueo que de sincronización en alternancia.
2. La operación de bloqueo sobre un *mutex* la debe desbloquear el mismo hilo de ejecución que lo bloqueó, por eso no pueden utilizarse para funcionando en alternancia por si solos.

Para poder trabajar en alternancia con los *mutex* deben trabajar con variables condicionales, que tienen las siguientes características principales:

1. Las variables condicionales permiten trabajar la alternancia con *mutex*.
2. Las variables condicionales tienen las operaciones de espera no activa con la llamada a la función `'pthread_cond_wait(&cond_var1, &mutex);'`.
3. Las variables condicionales tienen las operaciones de despertar a un hilo de ejecución de una espera no activa con la llamada a la función `'pthread_cond_signal(&cond_var1);'`.

Los mecanismos de sincronización pueden utilizarse según la conveniencia del programador. Normalmente cuando se trabaja con procesos diferentes se suelen utilizar los semáforos, y cuando se trabaja con hilos de ejecución con *mutex* y si es necesario variables condicionales.

```
35 void *handle_thread1(void *vargp)
36 {
37     int *myiterations = (int *)vargp;
38     int count;
39
40     /* Print thread information */
41     printf("Executing handle_thread for %d iterations \n", *myiterations);
42     fprintf(stdout, "Thread ID in handler %lu\n", pthread_self());
43
44     for (count = 0; count < (*myiterations); count++) {
45         /* Acquire mutex */
46         pthread_mutex_lock(&mutex);
47
48         /* Check current_thread */
49         if (current_thread == 2) {
50             /* Wait cond_var1 */
51             pthread_cond_wait(&cond_var1, &mutex);
52         }
53
54         /* In critical zone */
55         fprintf(stdout, "Critical zone, thread %lu %d---%d\n", pthread_self(), count,
56             ↪ value++);
57
58         /* Upadte current_thread */
59         current_thread = 2;
60
61         /* Signal cond_var2 */
62         pthread_cond_signal(&cond_var2);
63
64         /* Release mutex */
65         pthread_mutex_unlock(&mutex);
66     }
67     return NULL;
68 }
```

Código 91: Código de la rutina para el primer hilo de ejecución en el ejemplo 'mutex5.c'.

```

70 void *handle_thread2(void *vargp)
71 {
72     int *myiterations = (int *)vargp;
73     int count;
74
75     /* Print thread information */
76     printf("Executing handle_thread for %d iterations \n", *myiterations);
77     fprintf(stdout, "Thread ID in handler %lu\n", pthread_self());
78
79     /* Repeat */
80     for (count = 0; count < (*myiterations); count++) {
81         /* Acquire mutex */
82         pthread_mutex_lock(&mutex);
83
84         /* Check current_thread */
85         if (current_thread == 1) {
86             /* Wait for cond_var1 */
87             pthread_cond_wait(&cond_var2, &mutex);
88         }
89
90         /* In critical zone */
91         fprintf(stdout, "Critical zone, thread %lu %d---%d\n", pthread_self(), count,
92             ↵ value++);
93
94         /* Upadte current_thread */
95         current_thread = 1;
96
97         /* Signal cond_var1 */
98         pthread_cond_signal(&cond_var1);
99
100        /* Release mutex */
101        pthread_mutex_unlock(&mutex);
102    }
103    return NULL;
104 }

```

Código 92: Código de la rutina para el segundo hilo de ejecución en el ejemplo 'mutex5.c'.

```
106 int main(int argc, char *argv[])
107 {
108     int    input;
109     char    buffer[10 + 2];
110     char    key;
111     int     iterations;
112     int     errno = 0;
113     int     count;
114     int     result;
115
116     pthread_t handle_thread_id1;
117     pthread_t handle_thread_id2;
118
119     /* Parse program arguments */
120     if (argc != 2) {
121         fprintf(stderr, "Usage: %s <iterations>\n", argv[0]);
122         exit(1);
123     }
124
125     iterations = atoi(argv[1]);
126     value = 0;
127
128     /* Create and print thread1 */
129     pthread_create(&handle_thread_id1, NULL, handle_thread1, (void *)&iterations);
130     fprintf(stdout, "Thread ID for handler %lu\n", handle_thread_id1);
131
132     /* Create and print thread2 */
133     pthread_create(&handle_thread_id2, NULL, handle_thread2, (void *)&iterations);
134     fprintf(stdout, "Thread ID for handler %lu\n", handle_thread_id2);
135
136     /* Wait for thread1 */
137     result = pthread_join(handle_thread_id1, NULL);
138     if (result > 0) {
139         fprintf(stderr, "Joining thread handle_thread error: errno = %d\n", errno);
140         exit(1);
141     }
142
143     /* Wait for thread2 */
144     result = pthread_join(handle_thread_id2, NULL);
145     if (result > 0) {
146         fprintf(stderr, "Joining thread handle_thread error: errno = %d\n", errno);
147         exit(1);
148     }
149
150     exit(0);
151 }
```

Código 93: Código de la función principal del ejemplo 'mutex5.c'.

```
27 int current_thread = 2;
```

Código 94: Código de inicialización de la variable global 'current_thread' del ejemplo 'mutex6.c'.

Capítulo 6

Mecanismos de comunicación entre procesos

6.1. Introducción

Cuando queremos comunicar (enviar información entre diferentes procesos/*threads*) de un sistema podemos utilizar diferentes mecanismos de comunicación IPC (*InterProcess Communication*).

En el caso de *threads* recordemos que la forma más fácil es el uso de variables globales. Por lo que nos centraremos en la comunicación entre procesos que no comparten zonas de memoria, y para los que es imprescindible el uso de mecanismos de comunicación para compartir información.

Veremos diferentes soluciones (no todas las disponibles) que resumiremos en la sección de conclusiones.

6.2. Memoria Compartida

Para el caso de la comunicación entre procesos que hemos implementado antes con semáforos, anteriormente ahora vamos a utilizar un espacio de memoria compartida para poder compartir una variable.

Igual que en el caso de las variables dinámicas donde asignamos un puntero con la función '`malloc()`' y liberamos el espacio con la función '`free()`' en el caso de la memoria compartida, deberemos asignar el espacio de memoria a un puntero para su utilización en nuestros programas.

Uno de los procesos crea la zona de memoria compartida. En nuestro caso para poder generalizar el ejemplo la vamos a definir compuesta de tres tipos de informaciones con la estructura 'shmseg' como vemos en el Código 95.

```
30 struct shmseg {
31     int number;
32     int process;
33     char memory[100];
34 };
```

Código 95: Estructura para la información a guardar en la zona de memoria compartida en el ejemplo 'shm_1a.c'.

Cada espacio de memoria compartida, debe tener primero un identificador que lo representará en el sistema, y debe obtenerse con una llamada a la función 'shmget'. Cada zona se puede representar en el sistema a través de un número de clave (en nuestro caso a través de la macro '*##define SHM_KEY 0x1234*').

```
54     shmids = shmget(SHM_KEY, sizeof(struct shmseg), 0666|IPC_CREAT);
55     if (shmids == -1) err_sys("Shared Memory Error");
```

Código 96: Identificador de la zona de memoria compartida en el ejemplo 'shm_1a.c'.

Cuando ya tenemos el identificador a la zona de memoria compartida, lo podemos convertir en un puntero, para acceder a la información en el resto del programa con la función 'shmat' como vemos en el Código 97, que como en el caso de la función 'malloc' nos devuelve un puntero a memoria.

```
57 //attach to the shmp pointer
58 shmp = shmat(shmids, NULL, 0);
59 if (shmp == (void*)(-1)) err_sys("Shared Memory attachment error");
```

Código 97: Asignación del puntero para acceder a la zona de memoria 'shm_1a.c'.

El resto de procesos (por ejemplo 'shm_1b.c'), ya no deben crear la zona de memoria compartida, sino acceder a ella como vemos en el Código 98.


```
73     shmids = shmget(SHM_KEY, sizeof(struct shmseg), 0666);  
74     if (shmids == -1) err_sys("Shared Memory Error");
```

Código 98: Identificador de la zona de memoria compartida en el ejemplo 'shm_1b.c' que ya se creó anteriormente.

Ahora en el bucle (por ejemplo de 'shm_1c.c') ya podemos acceder a la información compartida entre los procesos como vemos en el Código 99.

```
87     for (count=0; count < atoi(argv[1]); count++) {  
88         fprintf(stdout, "ready to execute WAIT, process %s, %d\n", argv[2], count);  
89         if (sem_wait(psem2)<0) err_sys("WAIT psem2");  
90         //just in the critical zone  
91         shmp->number=shmp->number+1;  
92         shmp->process=2;  
93         fprintf(stdout, "critical zone, process %s, %d, %d::%s\n", argv[2], shmp->process,  
94             ↪ shmp->number, (char*)shmp->memory);  
95         //exiting critical zone  
96         if (sem_post(psem1)<0) err_sys("SIGNAL psem1");  
97     }
```

Código 99: Acceso a la zona de memoria crítica en el ejemplo 'shm_1c.c'.

Nuestros tres programas 'shm_1a.c', 'shm_1b.c' y 'shm_1c.c' son la repetición de los códigos que teníamos con los semáforos 'sem_6a.c', 'sem_6b.c' y 'sem_6c.c' pero añadiendo la parte de acceso a la zona de memoria compartida.

Generamos los tres ejecutables 'shm_1a', 'shm_1b' y 'shm_1c' con el comando 'make' y las opciones adecuadas como vemos en la Figura 6.1.

```
JOC1: make shm_1a -f shm.Makefile  
gcc -c shm_1a.c -Wall -Wno-unused-variable -I ./ -pthread  
gcc -o shm_1a shm_1a.o -I ./ -pthread  
JOC1: make shm_1b -f shm.Makefile  
gcc -c shm_1b.c -Wall -Wno-unused-variable -I ./ -pthread  
gcc -o shm_1b shm_1b.o -I ./ -pthread  
JOC1: make shm_1c -f shm.Makefile  
gcc -c shm_1c.c -Wall -Wno-unused-variable -I ./ -pthread  
gcc -o shm_1c shm_1c.o -I ./ -pthread  
JOC1: █
```

Figura 6.1: Generación de los ejecutables para 'shm_1a', 'shm_1b' y 'shm_1c'.

Inicializamos los semáforos, el primero a 1 (empezará el proceso 1) y el segundo a 0, y creamos la zona de memoria compartida, como vemos en la Figura 6.2 con el comando `./shm_1a 1 0`.

```
UOC1: ./shm_1a
USAGE: ./shm_1a value_sem1 value_sem2
UOC1: ./shm_1a 1 0
PROCESS 1(SEM1): 0
PROCESS 1(SEM1): 1
PROCESS 1(SEM2): 1
PROCESS 1(SEM2): 0
UOC1: █
```

Figura 6.2: Inicializamos el problema con el semáforo `'semaphore1'` a 1, y el semáforo `'semaphore2'` a 0, y creamos la zona de memoria compartida con el comando `./shm_1a 1 0`.

Lanzamos los dos procesos en modo *background* para que hagan tres iteraciones, el primero con el comando `./shm_1b 3 1 &` para identificarlo como proceso 1, y el segundo con el comando `./shm_1c 3 2 &` para identificarlo como proceso 2. Como vemos en la Figura 6.3 los dos procesos se ejecutarán en alternancia, tres veces, empezando por el proceso 1.

```
UOC1: ./shm_1b
USAGE: ./shm_1b <iterations> <ID>
UOC1: ./shm_1c
USAGE: ./shm_1c <iterations> <ID>
UOC1: ./shm_1b 3 1 & ./shm_1c 3 2 &
[1] 20341
[2] 20342
UOC1: ready to execute WAIT, process 2, 0
ready to execute WAIT, process 1, 0
critical zone, process 1, 1, 2::Sentence
ready to execute WAIT, process 1, 1
critical zone, process 2, 2, 3::Sentence
ready to execute WAIT, process 2, 1
critical zone, process 1, 1, 4::Sentence
ready to execute WAIT, process 1, 2
critical zone, process 2, 2, 5::Sentence
ready to execute WAIT, process 2, 2
critical zone, process 1, 1, 6::Sentence
critical zone, process 2, 2, 7::Sentence

[1]- Done          ./shm_1b 3 1
[2]+ Done          ./shm_1c 3 2
UOC1: █
```

Figura 6.3: Ejecución de `'shm_1b'` y `'shm_1c'` en alternancia empezando por el proceso 1.

Podemos ver la información de la zona de memoria compartida, al ser un mecanismo de `'System V'` con el comando `'ipcs'` como vemos en la Figura 6.4. Donde vemos la clave utilizada para la creación, el identificador, los permisos, y el tamaño. Como los procesos ya acabaron ahora no la tenemos en ningún proceso.

```

UOC1: ipcs
----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes       nattch     status
0x00001234  65536      root       666        108         0
----- Semaphore Arrays -----
key          semid      owner      perms      nsems
UOC1: █

```

Figura 6.4: Ejecución del comando 'ipcs' para ver la información asociada a la zona de memoria compartida.

Vamos a ejecutar el proceso 1 ('shm_1b'), pero no el 2, con lo que quedaremos bloqueados, y podremos ver la información sobre la zona de memoria compartida donde ahora vemos que está asociada a un proceso, como vemos en la Figura 6.5.

```

UOC1: ipcs -m
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes       nattch     status
0x00001234  65536      root       666        108         1
UOC1: █

```

Figura 6.5: Ejecución del comando 'ipcs' para ver la información asociada a la zona de memoria compartida con un proceso asociado.

Podemos ver los límites en el sistema con el comando 'ipcs -lm' como vemos en la Figura 6.6.

```

UOC1: ipcs -lm
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1
UOC1: █

```

Figura 6.6: Límites del sistema asociados a las zonas de memoria compartida.

Podemos ver la información del proceso que creó la zona de memoria (con PID 20338) y el último que accedió a la zona (con PID 20345, que corresponde al proceso 'shm_1b' que ejecutamos) con el comando 'ipcs -pm' como vemos en la Figura 6.7.

```
UOC1: ipcs -pm
----- Shared Memory Creator/Last-op PIDs -----
shmid  owner  cpid  lpid
65536   root    20338 20345

UOC1: ps -aux |grep shm
root    20345  0.0  0.0  2432  776 pts/0    S   11:33   0:00 ./shm_1b 3 1
root    20351  0.0  0.0  3084  888 pts/0    S+  11:39   0:00 grep  shm
UOC1: █
```

Figura 6.7: Información sobre la creación y acceso a la zona de memoria compartida.

6.3. Tuberías

Un camino alternativo de comunicar información entre procesos son las tuberías.

Podemos trabajar con tuberías sin nombre y tuberías con nombre.

Las primeras sólo se pueden utilizar entre procesos (*threads*) que tienen relación parental (no necesitamos al nombre, pues utilizaremos la herencia para conocer en todos las tuberías). Las que se puedan generar en un único programa. Las segundas se pueden utilizar en cualquier caso.

Las tuberías son canales de comunicación unidireccionales. La información que se escribe en un extremo de la tubería es accesible en el otro extremo de la misma.

6.4. Tuberías sin nombre

Veamos primero las tuberías sin nombre para comunicar dos procesos con relación parental.

Una tubería tiene dos descriptores (de tubería) asociados, Uno para las operaciones de lectura y otro para las operaciones de escritura. Si más de un proceso accede a la tubería todos comparten la entrada y la salida de la misma.

Respecto a las tuberías (como de hecho sucede también con los *sockets*), si leemos de una tubería vacía que ya está (esto sí es importante abierta en ambos extremos, que es lo normal en tuberías sin nombre) el proceso que lee quedará bloqueado hasta que haya información a leer en la tubería. Por lo tanto, como los propios *sockets* puede utilizarse como mecanismo de sincronización.

El segundo elemento importante es que la lectura de información de tamaño variable puede provocar consecuencias inesperadas. Si escribimos un texto y luego un número y en lectura leemos un texto (de tamaño variable y desconocido excepto su tamaño máximo) y luego el número, probablemente si el número ya está escrito cuando leamos el texto

leamos ambos.

Para evitar problemas es mejor (en determinadas situaciones al menos) cuando se escriba un texto de tamaño variable, primero escribir el tamaño (como un número) y luego el texto, para tener control en la lectura. Se lee primero el número y luego el número exacto de caracteres del texto. Veremos luego un ejemplo.

Para trabajar con los ficheros de ejemplo, tendremos un fichero de configuración 'pipe.Makefile' que tenemos en el Código 100 para utilizar con las opciones adecuadas para generar los ejecutables de cada ejemplo, como veremos a continuación.

```
1 PROGRAM_NAME_1 = pipe1
2 PROGRAM_OBJS_1 = pipe1.o
3
4 PROGRAM_NAME_2 = pipe2
5 PROGRAM_OBJS_2 = pipe2.o
6
7 PROGRAM_NAME_3 = pipe3
8 PROGRAM_OBJS_3 = pipe3.o
9
10 PROGRAM_NAME_4 = pipe4
11 PROGRAM_OBJS_4 = pipe4.o
12
13 PROGRAM_NAME_5 = pipe5
14 PROGRAM_OBJS_5 = pipe5.o
15
16 PROGRAM_NAME_6 = pipe6
17 PROGRAM_OBJS_6 = pipe6.o
18
19 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
   ↪ $(PROGRAM_NAME_4) $(PROGRAM_NAME_5) $(PROGRAM_NAME_6)
20 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
   ↪ $(PROGRAM_OBJS_4) $(PROGRAM_OBJS_5) $(PROGRAM_OBJS_6)
21
22 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
23
24 all: $(PROGRAM_NAME_ALL)
25     @echo "Finished!"
26
27 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
28     gcc -o $$ $^ -I ./ -pthread
29
```

```
30
31 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
32 gcc -o $@ $^ -I ./ -pthread
33
34 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
35 gcc -o $@ $^ -I ./ -pthread
36
37 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
38 gcc -o $@ $^ -I ./ -pthread
39
40 $(PROGRAM_NAME_5): $(PROGRAM_OBJS_5)
41 gcc -o $@ $^ -I ./ -pthread
42
43 $(PROGRAM_NAME_6): $(PROGRAM_OBJS_6)
44 gcc -o $@ $^ -I ./ -pthread
45
46 %.o: %.c
47 gcc -c $< -Wall -Wno-unused-variable -I ./ -pthread
48
49 clean:
50 rm -f $(REBUIDABLES)
51 @echo "Clean done"
```

Código 100: Fichero de configuración 'pipe.Makefile'.

El primer ejemplo ('pipe1.c') va a crear una tubería sin nombre (con la función 'pipe()') como vemos en el Código 101; la llamada nos va a devolver dos descriptores de la tubería 'pipefd[0]' que podremos utilizar para la lectura de la tubería (como si fuera un descriptor de fichero, o de *socket* orientado a conexión) y 'pipefd[1]' que utilizaremos para escritura.

El ejemplo va a escribir un texto fijo en la tubería de 12 caracteres (con la llamada 'write(pipefd[1], "hello world\n ", 12)'); lo va a leer sin conocer previamente el tamaño, con lo que vamos a intentar leer tantos caracteres como podamos de la tubería (con la llamada '(n=read(pipefd[0], buffer, sizeof(buffer)))'), y vamos a imprimir el resultado en la salida estándar (con la llamada 'write(1, buffer, n);').

Los descriptores de un proceso son números secuenciales que va asignando el sistema operativo a medida que los va necesitando, como ya vimos en ejemplos anteriores. Por defecto todo proceso tiene abiertos tres canales. La entrada estándar con el descriptor

```
1  /*
2   * Filename: pipe1.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8
9  void err_sys(const char* cadena)
10 {
11     perror(cadena);
12     exit(1);
13 }
14
15 int main()
16 {
17     int pipefd[2],n;
18     char buffer[100];
19
20     if (pipe(pipefd) < 0)
21         err_sys("error with pipe");
22     printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
23     if (write(pipefd[1],"hello world\n", 12)!=12)
24         err_sys("write_error");
25     if ( (n=read(pipefd[0], buffer, sizeof(buffer))) <= 0)
26         err_sys("read_error");
27
28     write(1, buffer, n);
29     exit(0);
30 }
31 }
```

Código 101: Código ejemplo 'pipe1.c'.

0; la salida estándar con el descriptor 1 (por eso el último 'write()' de nuestro ejemplo utiliza el descriptor 1), y la salida de mensajes de error estándar que tiene el descriptor 2. A partir de aquí cuando se necesita un nuevo descriptor se le asignará el 3 y después el 4...

Cuando creamos la tubería sin nombre, se nos asignan los descriptors 3 para lectura (siempre es el primero) y el 4 para escritura.

Generamos el fichero ejecutable 'pipe1' con el comando 'make pipe1 -f pipe.Makefile' y las opciones adecuadas como vemos en la Figura 6.8.

Ejecutamos el programa, como vemos en la Figura 6.9 con el comando './pipe1', y vemos como tenemos los dos descriptors abiertos para la tubería sin nombre con número

```
UOC1: make pipe1 -f pipe.Makefile
gcc -c pipe1.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe1 pipe1.o -I ./ -pthread
UOC1: █
```

Figura 6.8: Generación del ejecutable para 'pipe1'.

de descriptor 3 y 4, y como se imprime por pantalla el mensaje original 'hello world'.

```
UOC1: ./pipe1
read fd = 3, write fd = 4
hello world
UOC1: █
```

Figura 6.9: Ejecución del primer ejemplo './pipe1'.

Evidentemente tener un mismo proceso para escribir y leer no parece muy útil. Por lo que vamos a crear dos procesos en el programa 'pipe2.c', uno leerá y el otro escribirá, utilizando la bifurcación de procesos que ya conocemos a través de la llamada a la función 'fork()'.

```
21 if (pipe(pipefd) < 0)
22     err_sys("error with pipe");
23 printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
24 pid=fork();
25 if (pid==0) {
26     //child process
27     if (write(pipefd[1], "hello world\n", 12)!=12) err_sys("write_error");
28 }
29 else {
30     //parent process
31     if ( (n=read(pipefd[0], buffer, sizeof(buffer))) <= 0) err_sys("read_error");
32     write(1, buffer, n);
33 }
34 exit(0);
```

Código 102: Código ejemplo 'pipe2.c' donde vemos la bifurcación de procesos.

Como vemos en el Código 102 el proceso padre creará la tubería con los descriptors de lectura y escritura. Creará un nuevo proceso (hijo), que heredará los descriptors. El proceso hijo escribirá el mensaje en la tubería y el proceso padre lo leerá e imprimirá por la salida estándar.

Generamos el fichero ejecutable 'pipe2' con el comando 'make pipe2 -f pipe.Makefile' y las opciones adecuadas como vemos en la Figura 6.10.


```
UOC1: make pipe2 -f pipe.Makefile
gcc -c pipe2.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe2 pipe2.o -I ./ -pthread
UOC1: █
```

Figura 6.10: Generación del ejecutable para 'pipe2'.

Ejecutamos el programa, como vemos en la Figura 6.11 con el comando './pipe2', y vemos como tenemos los dos descriptors abiertos para la tubería sin nombre con número de descriptor 3 y 4, y como se imprime por pantalla el mensaje original 'hello world'.

```
UOC1: ./pipe2
read fd = 3, write fd = 4
hello world
UOC1: █
```

Figura 6.11: Ejecución del segundo ejemplo './pipe2' donde tenemos dos procesos que acceden a la tubería sin nombre.

El resultado será el mismo que en el caso anterior, visualmente, pero ahora tendremos dos procesos implicados en la escritura y lectura (lo cual parece más razonable).

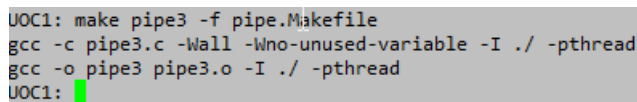
Ahora vamos a escribir un texto, y luego un número, para ver los problemas que puede ocasionar no hacerlo de forma correcta.

```
21  if (pipe(pipefd) < 0) err_sys("error with pipe");
22  printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
23  pid=fork();
24  if (pid==0) {
25      //child process
26      if (write(pipefd[1], "hello world\n", 12)!=12) err_sys("write_error");
27      n = 2;
28      if (write(pipefd[1], &n, sizeof(int))!=sizeof(int)) err_sys("write_error");
29  }
30  else {
31      //parent process
32      fprintf(stdout, "the number is %d\n", n);
33      if ( (n=read(pipefd[0], buffer, sizeof(buffer))) <= 0) err_sys("read_error");
34      write(1, buffer, n);
35      if ( (n=read(pipefd[0], &n, sizeof(int))) <= 0) err_sys("read_error");
36      fprintf(stdout, "now the number is %d\n", n);
37
38  }
```

Código 103: Código ejemplo 'pipe3.c' donde vemos la bifurcación de procesos y el intento de procesar a través de la tubería un texto y a continuación un número.

Como vemos en el Código 103 el proceso padre creará la tubería con los descriptors de lectura y escritura. Creará un nuevo proceso (hijo), que heredará los descriptors. El proceso hijo escribirá el mensaje en la tubería (`'write(pipefd[1], "hello world\n ", 12)'`) y a continuación el número (`'write(pipefd[1], &n, sizeof(int))'`), y el proceso padre los leerá, primero el mensaje (del que no conoce su tamaño) (con lo que intenta el máximo de capacidad) (`'(n=read(pipefd[0], buffer, sizeof(buffer)))'`) y luego el número (`'(n=read(pipefd[0], &n, sizeof(int)))'`) y si todo es correcto los imprimirá por la salida estándar.

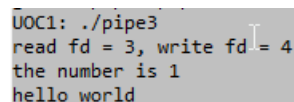
Generamos el fichero ejecutable `'pipe3'` con el comando `'make pipe3 -f pipe.Makefile'` y las opciones adecuadas como vemos en la Figura 6.12.



```
UOC1: make pipe3 -f pipe.Makefile
gcc -c pipe3.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe3 pipe3.o -I ./ -pthread
UOC1: █
```

Figura 6.12: Generación del ejecutable para `'pipe3'`.

Ejecutamos el programa, como vemos en la Figura 6.13 con el comando `'./pipe3'`, y vemos como tenemos los dos descriptors abiertos para la tubería sin nombre con número de descriptor 3 y 4; imprimimos el mensaje (`'the number is 1'`) en el proceso padre, para indicar que estamos esperando el texto a través de la tubería sin nombre, que nos tiene que enviar el proceso hijo; pero luego imprimimos el mensaje (`'hello world'`) que parece el mensaje que nos tenía que enviar el proceso hijo (el mismo que en los ejemplos anteriores); pero no tenemos el mensaje de impresión del proceso padre conforme se ha recibido el número del proceso hijo. Estamos pues, como vemos bloqueados en el proceso padre esperando la recepción del número.



```
UOC1: ./pipe3
read fd = 3, write fd = 4
the number is 1
hello world
█
```

Figura 6.13: Ejecución del tercer ejemplo `'./pipe3'` con el proceso padre bloqueado a la espera de recibir el número que debería haber enviado el proceso hijo.

Parece que tenemos el proceso de lectura bloqueado esperando el número. Para entender qué está pasando creamos una variante de nuestro programa (`'pipe4.c'`) en la que como vemos en la Código 104 imprimiremos el número de caracteres que el proceso padre ha leído en el momento de leer el mensaje.

```
35  fprintf(stdout, "the number of characters on text are %d\n", n);
```

Código 104: Código ejemplo 'pipe4.c' donde vemos la función para imprimir el número de caracteres leídos en el mensaje por el proceso padre.

Generamos el fichero ejecutable 'pipe4' con el comando 'make pipe4 -f pipe.Makefile' y las opciones adecuadas como vemos en la Figura 6.14.

```
UOC1: make pipe4 -f pipe.Makefile
gcc -c pipe4.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe4 pipe4.o -I ./ -pthread
UOC1: █
```

Figura 6.14: Generación del ejecutable para 'pipe4'.

Ejecutamos el programa, como vemos en la Figura 6.15 con el comando './pipe4', y vemos como tenemos los dos descriptores abiertos para la tubería sin nombre con número de descriptor 3 y 4; imprimimos el mensaje ('the number is 1') en el proceso padre, para indicar que estamos esperando el texto a través de la tubería sin nombre, que nos tiene que enviar el proceso hijo; pero luego imprimimos el mensaje ('hello world') que parece el mensaje que nos tenía que enviar el proceso hijo (el mismo que en los ejemplos anteriores); a continuación vemos como imprimimos el mensaje con el número de caracteres leídos ('the number of characters on text are 16') pero no tenemos el mensaje de impresión del proceso padre conforme se ha recibido el número del proceso hijo. Estamos pues, como vemos bloqueados en el proceso padre esperando la recepción del número.

```
UOC1: ./pipe4
read fd = 3, write fd = 4
the number is 1
hello world
the number of characters on text are 16
█
```

Figura 6.15: Ejecución del cuarto ejemplo './pipe4' con el proceso padre bloqueado a la espera de recibir el número que debería haber enviado el proceso hijo, e indicando que ha leído 16 caracteres en el mensaje.

Efectivamente vemos que el número de caracteres leídos cuando vamos a leer el texto es de 16: 12+4 (los 4 *bytes* del tamaño de un 'integer'). Por lo tanto, al ya tener en la tubería tanto la información del mensaje como la del número el proceso lo lee todo en una única lectura pensando que es el mensaje.

Cuando se trabaja con tuberías es importante sincronizar la escritura y la lectura, por ejemplo conociendo en todo momento los tamaños de las informaciones a leer.

Para evitar esta situación al escribir el texto, primero pondremos el número de caracteres y luego el texto.

Creamos una variante de nuestro programa ('pipe5.c') en la que como vemos en la Código 105 el hijo primero escribirá el número de caracteres del mensaje, y luego el mensaje, para que el proceso padre haga la operación pareja de leer primero el número de caracteres que deberá leer, y a continuación intentar leer un número exacto de caracteres y no el máximo asociado a la zona de memoria en la que se va a guardar el mensaje leído.

```

22  if (pipe(pipefd) < 0) err_sys("error with pipe");
23  printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
24  pid=fork();
25  if (pid==0) {
26      //child process
27      n = strlen("hello world\n")+1;
28      if (write(pipefd[1], &n, sizeof(int)) != sizeof(int)) err_sys("write_error");
29      if (write(pipefd[1], "hello world\n", n) != n) err_sys("write_error");
30      number = -2;
31      if (write(pipefd[1], &number, sizeof(int)) != sizeof(int)) err_sys("write_error");
32  }
33  else {
34      //parent process}
35      fprintf(stdout, "the number is %d\n", number);
36      if ( (n=read(pipefd[0], &number, sizeof(int))) <= 0) err_sys("read_error");
37      if ( (n=read(pipefd[0], buffer, number)) != number) err_sys("read_error");
38      write(1, buffer, n);
39      fprintf(stdout, "the number of characters on text are %d\n", n);
40      if ( (n=read(pipefd[0], &number, sizeof(int))) <= 0) err_sys("read_error");
41      fprintf(stdout, "now the number is %d\n", number);
42
43  }

```

Código 105: Código ejemplo 'pipe5.c' donde vemos como la lectura y escritura de mensajes ahora se hace con primero el número de caracteres y luego el mensaje.

Generamos el fichero ejecutable 'pipe5' con el comando 'make pipe5 -f pipe.Makefile' y las opciones adecuadas como vemos en la Figura 6.16.

Ejecutamos el programa, como vemos en la Figura 6.17 con el comando './pipe5', y vemos como tenemos los dos descriptores abiertos para la tubería sin nombre con número de descriptor 3 y 4; imprimimos el mensaje ('the number is 1', que es el valor que tiene la variable en la declaración del programa) en el proceso padre, para indicar

```
UOC1: make pipe5 -f pipe.Makefile
gcc -c pipe5.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe5 pipe5.o -I ./ -pthread
UOC1: █
```

Figura 6.16: Generación del ejecutable para 'pipe5'.

que estamos esperando el texto a través de la tubería sin nombre, que nos tiene que enviar el proceso hijo; pero luego imprimimos el mensaje ('hello world') que parece el mensaje que nos tenía que enviar el proceso hijo (el mismo que en los ejemplos anteriores); a continuación vemos como imprimimos el mensaje con el número de caracteres leídos ('the number of characters on text are 13', que corresponde con los 12+1 enviados por el proceso hijo), y a continuación el mensaje con el valor del número entero ('now the number is -2' que el proceso hijo cambio a valor -2 antes de enviarlo por la tubería).

```
UOC1: ./pipe5
read fd = 3, write fd = 4
the number is 1
hello world
the number of characters on text are 13
now the number is -2
UOC1: █
```

Figura 6.17: Ejecución del cuarto ejemplo './pipe5' de forma correcta.

Vemos como escribimos un texto de 13 caracteres (los 12 del texto) incluyendo el de final de 'string' ('\0').

Ahora el funcionamiento es el correcto.

Para finalizar, Vamos a incluir, en el programa 'pipe6.c' ahora una llamada a la función 'sleep(5)' en el proceso (hijo), como vemos en el Código 106 que escribe de 5 segundos, para ver como el proceso lector (padre) queda bloqueado a la espera de la información en el otro extremo de la tubería.

Hemos incluido las funciones necesarias para imprimir la hora, para ver el tiempo de espera de 5 segundos, en el proceso padre, como vemos en el Código 107.

Generamos el fichero ejecutable 'pipe6' con el comando 'make pipe6 -f pipe.Makefile' y las opciones adecuadas como vemos en la Figura 6.18.

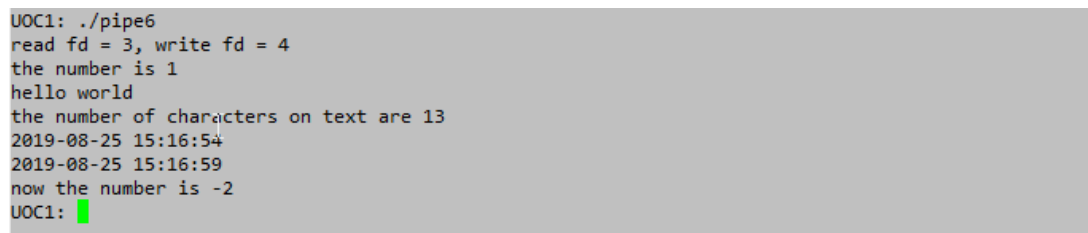
```
UOC1: make pipe6 -f pipe.Makefile
gcc -c pipe6.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o pipe6 pipe6.o -I ./ -pthread
UOC1: █
```

Figura 6.18: Generación del ejecutable para 'pipe6'.

```
30 if (pid==0) {
31     //child process
32     n = strlen("hello world\n")+1;
33     if (write(pipefd[1],&n, sizeof(int))!=sizeof(int)) err_sys("write_error");
34     if (write(pipefd[1],"hello world\n", n) !=n) err_sys("write_error");
35     number = -2;
36     sleep(5);
37     if (write(pipefd[1],&number, sizeof(int))!=sizeof(int)) err_sys("write_error");
38 }
```

Código 106: Código ejemplo 'pipe6.c' donde vemos la parte del proceso hijo con una parada de 5 segundos.

Ejecutamos el programa, como vemos en la Figura 6.19 con el comando './pipe6', y como vemos el proceso padre imprime antes del número la hora ('15:16:54') y después la hora con un retraso de 5 segundos ('15:16:59'), confirmando que la lectura en las tuberías (tanto con nombre como sin nombre es una operación bloqueante).



```
UOC1: ./pipe6
read fd = 3, write fd = 4
the number is 1
hello world
the number of characters on text are 13
2019-08-25 15:16:54
2019-08-25 15:16:59
now the number is -2
UOC1: █
```

Figura 6.19: Ejecución del cuarto ejemplo './pipe6' de forma correcta.

6.5. Tuberías con nombre

Ahora podemos traspasar el ejemplo a tuberías con nombre. El funcionamiento es similar, pero ahora deberemos abrir la tubería dos veces, una para escritura y otra para lectura. Como además tendrá nombre (será como si fuera un fichero) lo podremos utilizar con procesos que no tengan relación parental.

Para trabajar con los ficheros de ejemplo, tendremos un fichero de configuración 'mknod.Makefile' que tenemos en el Código 108 para utilizar con las opciones adecuadas para generar los ejecutables de cada ejemplo, como veremos a continuación.

```
39  else {
40      //parent process
41      fprintf(stdout,"the number is %d\n", number);
42      if ( (n=read(pipefd[0], &number, sizeof(int))) <= 0) err_sys("read_error");
43      if ( (n=read(pipefd[0], buffer, number)) != number) err_sys("read_error");
44      write(1, buffer, n);
45      fprintf(stdout,"the number of characters on text are %d\n",n);
46
47
48      time(&timer);
49      tm_info = localtime(&timer);
50      strftime(buffer, 26, "%Y-%m-%d %H:%M:%S", tm_info);
51      puts(buffer);
52
53      if ( (n=read(pipefd[0], &number, sizeof(int))) <= 0) err_sys("read_error");
54
55      time(&timer);
56      tm_info = localtime(&timer);
57      strftime(buffer, 26, "%Y-%m-%d %H:%M:%S", tm_info);
58      puts(buffer);
59
60      fprintf(stdout,"now the number is %d\n", number);
61
62  }
```

Código 107: Código ejemplo 'pipe6.c' donde vemos la parte del proceso padre con una impresión de hora, para ver la parada forzada por la espera en la escritura en el proceso hijo.

```
1  PROGRAM_NAME_1 = mknod1
2  PROGRAM_OBJS_1 = mknod1.o
3
4  PROGRAM_NAME_2 = mknod2a
5  PROGRAM_OBJS_2 = mknod2a.o
6
7  PROGRAM_NAME_3 = mknod2b
8  PROGRAM_OBJS_3 = mknod2b.o
9
10 PROGRAM_NAME_4 = mknod3a
11 PROGRAM_OBJS_4 = mknod3a.o
12
13 PROGRAM_NAME_5 = mknod3b
14 PROGRAM_OBJS_5 = mknod3b.o
15
```

```
16 PROGRAM_NAME_6 = mknod3c
17 PROGRAM_OBJS_6 = mknod3c.o
18
19 PROGRAM_NAME_7 = mknod4a
20 PROGRAM_OBJS_7 = mknod4a.o
21
22 PROGRAM_NAME_8 = mknod4b
23 PROGRAM_OBJS_8 = mknod4b.o
24
25 PROGRAM_NAME_9 = mknod4c
26 PROGRAM_OBJS_9 = mknod4c.o
27
28 PROGRAM_NAME_10 = mknod_cleaning
29 PROGRAM_OBJS_10 = mknod_cleaning.o
30
31 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
    ↪ $(PROGRAM_NAME_4) $(PROGRAM_NAME_5) $(PROGRAM_NAME_6) $(PROGRAM_NAME_7)
    ↪ $(PROGRAM_NAME_8) $(PROGRAM_NAME_9) $(PROGRAM_NAME_10)
32 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
    ↪ $(PROGRAM_OBJS_4) $(PROGRAM_OBJS_5) $(PROGRAM_OBJS_6) $(PROGRAM_OBJS_7)
    ↪ $(PROGRAM_OBJS_8) $(PROGRAM_OBJS_9) $(PROGRAM_OBJS_10)
33
34 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
35
36 all: $(PROGRAM_NAME_ALL)
37     @echo "Finished!"
38
39 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
40     gcc -o $@ $^ -I ./ -pthread
41
42 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
43     gcc -o $@ $^ -I ./ -pthread
44
45 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
46     gcc -o $@ $^ -I ./ -pthread
47
48 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
49     gcc -o $@ $^ -I ./ -pthread
50
51 $(PROGRAM_NAME_5): $(PROGRAM_OBJS_5)
52     gcc -o $@ $^ -I ./ -pthread
53
```



```
54 $(PROGRAM_NAME_6): $(PROGRAM_OBJS_6)
55 gcc -o $@ $^ -I ./ -pthread
56
57 $(PROGRAM_NAME_7): $(PROGRAM_OBJS_7)
58 gcc -o $@ $^ -I ./ -pthread
59
60 $(PROGRAM_NAME_8): $(PROGRAM_OBJS_8)
61 gcc -o $@ $^ -I ./ -pthread
62
63 $(PROGRAM_NAME_9): $(PROGRAM_OBJS_9)
64 gcc -o $@ $^ -I ./ -pthread
65
66 $(PROGRAM_NAME_10): $(PROGRAM_OBJS_10)
67 gcc -o $@ $^ -I ./ -pthread
68
69
70 %.o: %.c
71 gcc -c $< -Wall -Wno-unused-variable -I ./ -pthread
72
73 clean:
74 rm -f $(REBUIDABLES)
75 @echo "Clean done"
```

Código 108: Fichero de configuración 'mknod.Makefile'.

Cuando manejamos tuberías con nombre, lo primero será crear una entrada en la tabla de 'i-nodos' del sistema de ficheros, de tipo 'S_IFIFO' (tubería con nombre), con una llamada a la función 'mknod()' de la que vemos la información de la página de ayuda (del libro 2 con el comando 'man 2 mknod') en la Figura 6.20.

- El primero de los parámetros es el nombre de la tubería con nombre que deberán utilizar todos los procesos que quieran acceder a la misma.
- El segundo parámetro nos indicará que vamos a crear una tubería con nombre y los permisos de acceso que queremos que tenga
- En el caso de las tuberías con nombre el tercer parámetro siempre es 0.

A continuación cada uno de los procesos debe abrir un descriptor de fichero, en un caso para escribir y en el otro para leer. En nuestro

```
UOC1: man 2 mknod | head -13
MKNOD(2)                                Linux Programmer's Manual                                MKNOD(2)

NAME
    mknod, mknodat - create a special or ordinary file

SYNOPSIS
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <unistd.h>

    int mknod(const char *pathname, mode_t mode, dev_t dev);

UOC1: █
```

Figura 6.20: Ayuda del libro 2 del manual de ayuda de la función 'mknod()'.

En nuestro primer ejemplo, con un único programa 'mknod1.c' tendremos dos procesos.

Vamos a incluir en el programa 'mknod1.c' el proceso hijo; como vemos en el Código 109 se hará la parte de escritura en la tubería con nombre.

```
31  if (pid==0) {
32      //child process
33      //open S_IFIFO for writing
34      outputFile = open(FIFO_NAME, O_WRONLY /*| O_CREAT | O_TRUNC*/, 0644);
35      if (outputFile == (-1)) err_sys("Error: could not open FIFO for Writing");
36      n = strlen("hello world\n")+1;
37      if (write(outputFile,&n, sizeof(int))!=sizeof(int)) err_sys("write_error");
38      if (write(outputFile,"hello world\n", n) !=n) err_sys("write_error");
39      number = -2;
40      if (write(outputFile,&number, sizeof(int))!=sizeof(int)) err_sys("write_error");
41  }
```

Código 109: Código ejemplo 'mknod1.c' donde vemos la parte del proceso hijo que se encarga de escribir la información en la tubería con nombre.

Vamos a incluir en el programa 'mknod1.c' el proceso padre; como vemos en el Código 110 se hará la parte de lectura en la tubería con nombre. Además el proceso padre antes de la bifurcación creará la tubería con nombre en el sistema de ficheros de nuestro sistema con una llamada a la función 'mknod(FIFO_NAME, S_IFIFO|0666, 0)'.

Como vemos la tubería con nombre la vamos a crear en '/tmp/FIFO_FILE'.

Generamos el fichero ejecutable 'mknod1' con el comando 'make mknod1 -f mknod.Makefile' y las opciones adecuadas como vemos en la Figura 6.21.

Ejecutamos el programa, como vemos en la Figura 6.22 con el comando './mknod1',

```
42 else {
43     //parent process
44     inputFile = open(FIFO_NAME, O_RDONLY);
45     if (inputFile == (-1)) err_sys("Error: could not open FIFO for Reading");
46     fprintf(stdout, "the number is %d\n", number);
47     if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
48     if ( (n=read(inputFile, buffer, number)) != number) err_sys("read_error");
49     write(1, buffer, n);
50     fprintf(stdout, "the number of characters on text are %d\n", n);
51     if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
52     fprintf(stdout, "now the number is %d\n", number);
53 }
```

Código 110: Código ejemplo 'mknod1.c' donde vemos la parte del proceso padre que se encarga de leer la información de la tubería con nombre.

```
UOC1: make mknod1 -f mknod.Makefile
gcc -c mknod1.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod1 mknod1.o -I ./ -pthread
UOC1: █
```

Figura 6.21: Generación del ejecutable para 'mknod1'.

y como vemos el proceso padre imprime el valor de la variable tal y como está en su declaración ('the number is 1'), a continuación el mensaje ('hello world'), seguido del número de caracteres del mensaje ('the number of characters on text are 13'), y finalmente el nuevo valor de la variable después de su modificación y envío por parte del proceso hijo, y la lectura por parte del proceso padre ('now the number is -2').

```
UOC1: ./mknod1
the number is 1
hello world
the number of characters on text are 13
now the number is -2
UOC1: █
```

Figura 6.22: Ejecución del primer ejemplo './mknod1' con los procesos padre e hijo accediendo a la tubería con nombre.

En esta primera implementación los dos procesos tienen relación parental. De hecho hemos aprovechado el programa de las tuberías sin nombre, donde ya teníamos dos procesos creados en un mismo programa.

Como vemos en la Figura 6.23 que muestra la salida del comando 'ls -laF /tmp/FIFO_FILE' tenemos el fichero creado, sin información pues ya se leyó todo lo que se escribió en la tubería.

```
UOC1: ls -laF /tmp/FIFO_FILE
prw-r--r-- 1 root root 0 Aug 25 16:18 /tmp/FIFO_FILE|
UOC1: █
```

Figura 6.23: El fichero '/tmp/FIFO_FILE' está creado pero sin contenido.

Ahora vamos a separar los dos procesos en dos programas. Y los ejecutaremos en paralelo.

Si no eliminamos el fichero en la segunda ejecución nos dará un error, como vemos al intentar ejecutar nuevamente nuestro programa 'mknod1' en la Figura 6.24.

```
UOC1: ./mknod1
Error: could not create FIFO: File exists
UOC1: █
```

Figura 6.24: Error al volver a intentar crear la tubería con nombre en el sistema.

Para solucionar el problema podemos hacer un pequeño programa 'mknod_cleaning.c' para eliminar del sistema de fichero la tubería con nombre.

```
13 int main(int argc, char** argv) {
14
15     unlink(FIFO_NAME);
16 }
```

Código 111: Código ejemplo 'mknod_cleaning.c' para eliminar del sistema de ficheros la entrada a la tubería con nombre.

Generamos el fichero ejecutable 'mknod_cleaning' con el comando 'make mknod_cleaning -f mknod' y las opciones adecuadas como vemos en la Figura 6.25.

```
UOC1: make mknod_cleaning -f mknod.Makefile
gcc -c mknod_cleaning.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod_cleaning mknod_cleaning.o -I ./ -pthread
UOC1: █
```

Figura 6.25: Generación del ejecutable para 'mknod_cleaning'.

Ejecutamos el programa, como vemos en la Figura 6.26 con el comando './mknod_cleaning', y como vemos el fichero de la tubería con nombre ha sido eliminado del sistema.

Separamos el proceso de lectura y escritura en dos programas.

El programa 'mknod2a.c' como vemos en el Código 112 se encarga de la escritura en la tubería con nombre.

```
UOC1: ./mknod_cleaning
UOC1:
UOC1: ls -laF /tmp/FIFO_FILE
ls: cannot access '/tmp/FIFO_FILE': No such file or directory
UOC1: █
```

Figura 6.26: Ejecución del programa './mknod_cleaning' para eliminar del sistema de ficheros la tubería con nombre.

```
20 int main()
21 {
22     int number=1,n;
23     int inputFile, outputFile;
24
25     char buffer[100];
26     int pid;
27
28     if (mknod(FIFO_NAME, S_IFIFO|0666, 0) == (-1)) err_sys("Error: could not create
    ↪ FIFO");
29     //open S_IFIFO for writing
30     outputFile = open(FIFO_NAME, O_WRONLY /*| O_CREAT | O_TRUNC*/, 0644);
31     if (outputFile == (-1)) err_sys("Error: could not open FIFO for Writing");
32     n = strlen("hello world\n")+1;
33     if (write(outputFile,&n, sizeof(int))!=sizeof(int)) err_sys("write_error");
34     if (write(outputFile,"hello world\n", n) !=n) err_sys("write_error");
35     number = -2;
36     if (write(outputFile,&number, sizeof(int))!=sizeof(int)) err_sys("write_error");
37     exit(0);
38 }
```

Código 112: Código ejemplo 'mknod2a.c' para escribir en la tubería con nombre.

El programa 'mknod2b.c' como vemos en el Código 113 se encarga de la lectura de la tubería con nombre.

Generamos los dos ficheros ejecutables 'mknod2a' y 'mknod2b' con el comando 'make' y las opciones adecuadas como vemos en la Figura 6.27.

```
JOC1: make mknod2a -f mknod.Makefile
gcc -c mknod2a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod2a mknod2a.o -I ./ -pthread
JOC1: make mknod2b -f mknod.Makefile
gcc -c mknod2b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod2b mknod2b.o -I ./ -pthread
JOC1: █
```

Figura 6.27: Generación de los ejecutables para 'mknod2a' y 'mknod2b'.

Ejecutamos ambos programas, en modo *background*, como vemos en la Figura 6.28 y

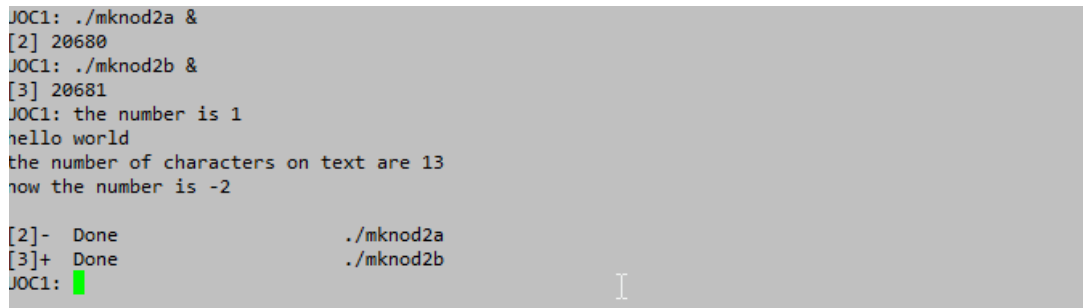
```

20 int main()
21 {
22     int number=1,n;
23     int inputFile, outputFile;
24
25     char buffer[100];
26     int pid;
27
28     inputFile = open(FIFO_NAME, O_RDONLY);
29     if (inputFile == (-1)) err_sys("Error: could not open FIFO for Reading");
30     fprintf(stdout,"the number is %d\n", number);
31     if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
32     if ( (n=read(inputFile, buffer, number)) != number) err_sys("read_error");
33     write(1, buffer, n);
34     fprintf(stdout,"the number of characters on text are %d\n",n);
35     if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
36     fprintf(stdout,"now the number is %d\n", number);
37
38     exit(0);
39 }

```

Código 113: Código ejemplo 'mknod2b.c' para leer de la tubería con nombre.

vemos que se imprimen correctamente los mensajes en el receptor.



```

JOC1: ./mknod2a &
[2] 20680
JOC1: ./mknod2b &
[3] 20681
JOC1: the number is 1
hello world
the number of characters on text are 13
now the number is -2

[2]- Done          ./mknod2a
[3]+ Done          ./mknod2b
JOC1: █

```

Figura 6.28: Ejecución de los programas './mknod2a' y './mknod2b'.

En el tercer ejemplo 'mknod3b.c' vamos a eliminar la última lectura de la tubería para ver el efecto en el sistema.

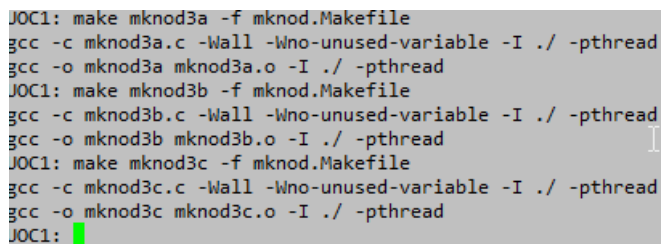
El programa 'mknod3b.c' como vemos en el Código 114 se encarga de la lectura de la tubería con nombre, pero no del último valor.

Ahora haremos un pequeño programa para leer lo que queda en la tubería. El programa 'mknod3c.c' como vemos en el Código 115 se encarga de la lectura de la tubería con nombre del último valor.

```
28  inputFile = open(FIFO_NAME, O_RDONLY);
29  if (inputFile == (-1)) err_sys("Error: could not open FIFO for Reading");
30  fprintf(stdout, "the number is %d\n", number);
31  if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
32  if ( (n=read(inputFile, buffer, number)) != number) err_sys("read_error");
33  write(1, buffer, n);
34  fprintf(stdout, "the number of characters on text are %d\n", n);
35  /*
36  if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
37  fprintf(stdout, "now the number is %d\n", number);
38  */
```

Código 114: Código ejemplo 'mknod3b.c' donde hemos eliminado la última lectura.

Generamos los tres ficheros ejecutables 'mknod3a', 'mknod3b' y 'mknod3c' con el comando 'make' y las opciones adecuadas como vemos en la Figura 6.29.



```
JOC1: make mknod3a -f mknod.Makefile
gcc -c mknod3a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod3a mknod3a.o -I ./ -pthread
JOC1: make mknod3b -f mknod.Makefile
gcc -c mknod3b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod3b mknod3b.o -I ./ -pthread
JOC1: make mknod3c -f mknod.Makefile
gcc -c mknod3c.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod3c mknod3c.o -I ./ -pthread
JOC1: █
```

Figura 6.29: Generación de los ejecutables para 'mknod3a', 'mknod3b' y 'mknod3c'.

Ejecutamos los programas, en modo *background*, como vemos en la Figura 6.30 y vemos que se imprimen correctamente los mensajes en el receptor, excepto el último valor que debía leer el programa 'mknod3c'.

En el cuarto ejemplo 'mknod4a.c' vamos a bloquear la finalización del proceso de escritura en la tubería con nombre, para garantizar que cuando se ejecute 'mknod4c' la tubería aún está abierta para escritura. El programa 'mknod4a.c' incorpora una llamada a la función 'while(1);'.

Generamos los tres ficheros ejecutables 'mknod4a', 'mknod4b' y 'mknod4c' con el comando 'make' y las opciones adecuadas como vemos en la Figura 6.31.

Ejecutamos los programas, en modo *background*, como vemos en la Figura 6.32 y vemos que se imprimen correctamente, ahora sí, todo los mensajes en el receptor.

Como vemos para el correcto funcionamiento de la tubería no deben cerrarse los extremos si no se quiere perder información.

```

20 int main()
21 {
22     int number=1,n;
23     int inputFile, outputFile;
24
25     char buffer[100];
26     int pid;
27
28     inputFile = open(FIFO_NAME, O_RDONLY);
29     if (inputFile == (-1)) err_sys("Error: could not open FIFO for Reading");
30     if ( (n=read(inputFile, &number, sizeof(int))) <= 0) err_sys("read_error");
31     fprintf(stdout,"now the number is %d\n", number);
32     exit(0);
33 }

```

Código 115: Código ejemplo 'mknod3c.c' para hacer la última lectura de la tubería con nombre.

```

JOC1: ./mknod_cleaning
JOC1:
JOC1: ./mknod3a &
[2] 20712
JOC1: ./mknod3b &
[3] 20713
JOC1: the number is 1
hello world
the number of characters on text are 13

[2]- Done                ./mknod3a
[3]+ Done                ./mknod3b
JOC1: ./mknod3c &
[2] 20714
JOC1:
JOC1:

```

Figura 6.30: Ejecución de los programas './mknod3a', 'mknod3b' y './mknod3c'.

```

UOC1: make mknod4a -f mknod.Makefile
gcc -c mknod4a.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod4a mknod4a.o -I ./ -pthread
UOC1: make mknod4b -f mknod.Makefile
gcc -c mknod4b.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod4b mknod4b.o -I ./ -pthread
UOC1: make mknod4c -f mknod.Makefile
gcc -c mknod4c.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o mknod4c mknod4c.o -I ./ -pthread
UOC1:

```

Figura 6.31: Generación de los ejecutables para 'mknod4a', 'mknod4b' y 'mknod4c'.

6.6. Ficheros

Por último para finalizar, vamos a trabajar con ficheros. El funcionamiento es muy diferente al de las tuberías, pues no hay bloqueo de lectura. Si intentamos leer de un fichero


```

28 if (mknod(FIFO_NAME, S_IFIFO|0666, 0) == (-1)) err_sys("Error: could not create
    ↪ FIFO");
29 //open S_IFIFO for writing
30 outputFile = open(FIFO_NAME, O_WRONLY /*| O_CREAT | O_TRUNC*/, 0644);
31 if (outputFile == (-1)) err_sys("Error: could not open FIFO for Writing");
32 n = strlen("hello world\n")+1;
33 if (write(outputFile,&n, sizeof(int))!=sizeof(int)) err_sys("write_error");
34 if (write(outputFile,"hello world\n", n) !=n) err_sys("write_error");
35 number = -2;
36 if (write(outputFile,&number, sizeof(int))!=sizeof(int)) err_sys("write_error");
37 while (1);
38 exit(0);

```

Código 116: Código ejemplo 'mknod4a.c' que queda bloqueado antes de salir.

```

UOC1: ./mknod_cleaning
UOC1: ./mknod4a &
[3] 20737
UOC1:
UOC1: ./mknod4b &
[4] 20738
UOC1: the number is 1
hello world
the number of characters on text are 13

[4]+ Done ./mknod4b
UOC1: ./mknod4c &
[4] 20739
UOC1: now the number is -2

[4]+ Done ./mknod4c
UOC1: █

```

Figura 6.32: Ejecución de los programas './mknod4a', 'mknod4b' y './mknod4c'.

y no tenemos información, no leemos nada, no nos quedamos bloqueados esperando, como sí sucede con las tuberías, o los *sockets*.

Para trabajar con los ficheros de ejemplo, tendremos un fichero de configuración 'file.Makefile' que tenemos en el Código 117 para utilizar con las opciones adecuadas para generar los ejecutables de cada ejemplo, como veremos a continuación.

```

1 PROGRAM_NAME_1 = file1
2 PROGRAM_OBJS_1 = file1.o
3
4 PROGRAM_NAME_2 = file2
5 PROGRAM_OBJS_2 = file2.o
6

```

```
7 PROGRAM_NAME_3 = file3
8 PROGRAM_OBJS_3 = file3.o
9
10 PROGRAM_NAME_4 = file4
11 PROGRAM_OBJS_4 = file4.o
12
13 PROGRAM_NAME_5 = file5
14 PROGRAM_OBJS_5 = file5.o
15
16 PROGRAM_NAME_ALL = $(PROGRAM_NAME_1) $(PROGRAM_NAME_2) $(PROGRAM_NAME_3)
   ↪ $(PROGRAM_NAME_4) $(PROGRAM_NAME_5)
17 PROGRAM_OBJS_ALL = $(PROGRAM_OBJS_1) $(PROGRAM_OBJS_2) $(PROGRAM_OBJS_3)
   ↪ $(PROGRAM_OBJS_4) $(PROGRAM_OBJS_5)
18
19 REBUIDABLES = $(PROGRAM_NAME_ALL) $(PROGRAM_OBJS_ALL)
20
21 all: $(PROGRAM_NAME_ALL)
22     @echo "Finished!"
23
24 $(PROGRAM_NAME_1): $(PROGRAM_OBJS_1)
25     gcc -o $@ $^ -I ./ -pthread
26
27 $(PROGRAM_NAME_2): $(PROGRAM_OBJS_2)
28     gcc -o $@ $^ -I ./ -pthread
29
30 $(PROGRAM_NAME_3): $(PROGRAM_OBJS_3)
31     gcc -o $@ $^ -I ./ -pthread
32
33 $(PROGRAM_NAME_4): $(PROGRAM_OBJS_4)
34     gcc -o $@ $^ -I ./ -pthread
35
36 $(PROGRAM_NAME_5): $(PROGRAM_OBJS_5)
37     gcc -o $@ $^ -I ./ -pthread
38
39 %.o: %.c
40     gcc -c $< -Wall -Wno-unused-variable -I ./ -pthread
41
42 clean:
43     rm -f $(REBUIDABLES)
44     @echo "Clean done"
```

Código 117: Fichero de configuración 'file.Makefile'.

Empezaremos con un ejemplo, 'file1.c' para mostrar como escribir información en un fichero ('destination_file.txt'). Cerrarlo. Volverlo a abrir para leer la información del fichero, como vemos en el Código 118. Será como entender que en dos bloques, tratamos el fichero como un elemento de comunicación. En el primer bloque escribiremos en el fichero; y en el segundo accederemos a dicha operación con una lectura, casi como si fuera una tubería con nombre.

```
1  /*
2   * Filename: file1.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  #include <sys/stat.h>
9  #include <sys/types.h>
10 #include <fcntl.h>
11
12 #include <unistd.h>
13
14 void err_sys(const char* cadena)
15 {
16     perror(cadena);
17     exit(1);
18 }
19
20 int main()
21 {
22     int file_write,file_read,n;
23     char buffer[100];
24
25     if ((file_write=open("./destination_file.txt",O_WRONLY|O_CREAT|O_TRUNC,00700)) < 0)
26         ↪ err_sys("error with destination file");
27     printf("write fd = %d\n", file_write);
28     if (write(file_write,"hello world\n", 12)!=12) err_sys("write_error");
29     if (close(file_write)!=0) err_sys("error close write");
30
31     if ((file_read=open("./destination_file.txt",O_RDONLY)) < 0) err_sys("error with
32         ↪ source (destination) file");
```

```
31 printf("read fd = %d\n", file_read);
32 if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
33 if (close(file_read)!=0) err_sys("error close read");
34
35 write(1, buffer, n);
36 exit(0);
37
38 }
```

Código 118: Código ejemplo 'file1.c'.

Generamos el fichero ejecutable 'file1' con el comando 'make file1 -f file.Makefile' y las opciones adecuadas como vemos en la Figura 6.33.

```
gcc -c file1.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o file1 file1.o -I ./ -pthread
UOC1: █
```

Figura 6.33: Generación del ejecutable para 'file1'.

Ejecutamos el programa como vemos en la Figura 6.34. Hemos primero abierto el fichero para escritura, y hemos obtenido el primer descriptor libre (3). Luego hemos escrito y cerrado. Al volver a abrirlo, volvemos a obtener el mismo descriptor, pero ahora para hacer operaciones de lectura en el fichero.

```
UOC1: ./file1
write fd = 3
read fd = 3
hello world
UOC1: █
```

Figura 6.34: Ejecución del programa './file1'.

En el siguiente ejemplo, 'file2.c', abriremos el descriptor de lectura antes de cerrar el de escritura, como vemos en el Código 119.

Generamos el fichero ejecutable 'file2' con el comando 'make file2 -f file.Makefile' y las opciones adecuadas como vemos en la Figura 6.35.

Ejecutamos el programa como vemos en la Figura 6.36. Ahora vamos a abrir el fichero como escritura, y después antes de cerrarlo, otro vez, como lectura. Ahora veremos que los dos descriptors de fichero son el 3, y el 4.

Ahora vamos a mirar de acceder a un fichero con información. Nuestro fichero de origen, 'source_file.txt', que vemos en el Código 120, tiene 33 bytes, tal y como vemos

```

18 int main()
19 {
20     int file_write,file_read,n;
21     char buffer[100];
22
23     if ((file_write=open("./destination_file.txt",O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU)) <
        ↪ 0) err_sys("error with destination file");
24     if ((file_read=open("./destination_file.txt",O_RDONLY)) < 0) err_sys("error with
        ↪ source (destination) file");
25     printf("write fd = %d\n", file_write);
26     printf("read fd = %d\n", file_read);
27     if (write(file_write,"hello world\n", 12)!=12) err_sys("write_error");
28     if (close(file_write)!=0) err_sys("error close write");
29     if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
30     if (close(file_read)!=0) err_sys("error close read");
31     write(1, buffer, n);
32     exit(0);
33 }

```

Código 119: Código ejemplo 'file2.c'.

```

UOC1: make file2 -f file.Makefile
gcc -c file2.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o file2 file2.o -I ./ -pthread
UOC1: █

```

Figura 6.35: Generación del ejecutable para 'file2'.

```

UOC1: ./file2
write fd = 3
read fd = 4
hello world
UOC1: █

```

Figura 6.36: Ejecución del programa './file2'.

en la Figura 6.37.

```

JOC1: cat source_file.txt
0123456789
ABCDEFGHIJ
abcdefghij
JOC1: ls -laF source_file.txt
-rw-r--r-- 1 root root 33 Aug 26 06:50 source_file.txt
JOC1: █

```

Figura 6.37: Comprobación del contenido del fichero './source_file.txt' y de su tamaño con 33 bytes.

Podemos utilizar la utilidad 'xxd' para ver el contenido del fichero tal y como está

```
1 0123456789
2 ABCDEFGHIJ
3 abcdefghij
```

Código 120: Contenido del fichero, de texto, 'source_file.txt'.

almacenado en el sistema como vemos en la Figura 6.38.

```
UOC1: xxd source_file.txt
00000000: 3031 3233 3435 3637 3839 0a41 4243 4445  0123456789.ABCDE
00000010: 4647 4849 4a0a 6162 6364 6566 6768 696a  FGHIJ.abcdefghij
00000020: 0a                                     .
UOC1: █
```

Figura 6.38: Comprobación del contenido del fichero './source_file.txt' y de su tamaño con 33 bytes, con la utilidad 'xxd'.

Si fuera necesario instalaríamos la utilidad 'xxd' con el comando 'apt-get install xxd' como vemos en la Figura 6.39.

```
UOC1: apt-get install xxd
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  xxd
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 140 kB of archives.
```

Figura 6.39: Instalación de la utilidad 'xxd' con el comando 'apt-get install xxd'.

Podemos utilizar la utilidad 'hexdump' para ver el contenido del fichero tal y como está almacenado en el sistema como vemos en la Figura 6.40.

```
UOC1: hexdump source_file.txt
00000000 3130 3332 3534 3736 3938 410a 4342 4544
00000010 4746 4948 0a4a 6261 6463 6665 6867 6a69
00000020 000a
00000021
UOC1: █
```

Figura 6.40: Comprobación del contenido del fichero './source_file.txt' y de su tamaño con 33 bytes, con la utilidad 'hexdump'.

Si fuera necesario instalaríamos la utilidad 'hexdump' con el comando 'apt-get install bsdmainutils' como vemos en la Figura 6.41.

En el siguiente ejemplo, 'file3.c', enviaremos al fichero de destino 'destination_file.txt' la información del fichero origen 'source_file.txt', como vemos en el Código 121.

```
UOC1: apt-get install bsdmainutils
Reading package lists... Done
Building dependency tree
Reading state information... Done
bsdmainutils is already the newest version (11.1.2+b1).
```

Figura 6.41: Instalación de la utilidad 'hexdump' con el paquete bsdmainutils, con el comando 'apt-get install bsdmainutils'.

Generamos el fichero ejecutable 'file3' con el comando 'make file3 -f file.Makefile' y las opciones adecuadas como vemos en la Figura 6.42.

```
UOC1: make file3 -f file.Makefile
gcc -c file3.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o file3 file3.o -I ./ -pthread
UOC1: █
```

Figura 6.42: Generación del ejecutable para 'file3'.

Ejecutamos el programa como vemos en la Figura 6.43. Hemos imprimido los descriptores de los tres ficheros (3, 4 y 5). Vamos a leer un máximo de 100 caracteres del fichero de entrada (tiene 33, por lo tanto lo leeremos todo), lo escribiremos en un fichero temporal, lo leeremos de dicho fichero, y finalmente lo imprimiremos por la salida estándar.

```
UOC1: ./file3
write fd = 3
read fd = 4
input fd = 5
0123456789
ABCDEFGHIJ
abcdefghij
UOC1: █
```

Figura 6.43: Ejecución del programa './file3'.

Podemos comparar el fichero de entrada con el temporal, para ver que son iguales, como vemos en la Figura 6.44 con el comando 'diff source_file.txt destination_file.txt'.

```
UOC1: diff source_file.txt destination_file.txt
UOC1: █
```

Figura 6.44: comprobación que el fichero origen './source_file.txt' y el fichero destino 'destination_file.txt' son idénticos con el comando 'diff source_file.txt destination_file.txt'.

En el siguiente ejemplo, 'file4.c', sólo vamos a leer una parte del fichero de entrada (conceptualmente una línea, 11 caracteres), como vemos en el Código 122.

Generamos el fichero ejecutable 'file4' con el comando 'make file4 -f file.Makefile' y las opciones adecuadas como vemos en la Figura 6.45.

```

22 int main()
23 {
24     int file_write,file_read,n;
25     int input;
26     char buffer[100];
27
28     if ((file_write=open("./destination_file.txt",O_WRONLY|O_CREAT|O_TRUNC,0700)) < 0)
29         ↪ err_sys("error with destination file");
30     if ((file_read=open("./destination_file.txt",O_RDONLY)) < 0) err_sys("error with
31         ↪ source (destination) file");
32     if ((input=open("./source_file.txt",O_RDONLY)) < 0) err_sys("error with
33         ↪ source_file");
34
35     printf("write fd = %d\n", file_write);
36     printf("read fd = %d\n", file_read);
37     printf("input fd = %d\n", input);
38
39     if ( (n=read(input, buffer, sizeof(buffer))) <= 0) err_sys("read_error input");
40     if (write(file_write,buffer, n)!=n) err_sys("write_error");
41     if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
42     write(1, buffer, n);
43
44     if (close(file_write)!=0) err_sys("error close write");
45     if (close(file_read)!=0) err_sys("error close read");
46     if (close(input)!=0) err_sys("error close input");
47
48     exit(0);
49 }

```

Código 121: Código ejemplo 'file3.c'.

```

UOC1: make file4 -f file.Makefile
gcc -c file4.c -Wall -Wno-unused-variable -I ./ -pthread
gcc -o file4 file4.o -I ./ -pthread
UOC1: █

```

Figura 6.45: Generación del ejecutable para 'file4'.

Ejecutamos el programa como vemos en la Figura 6.46. Como vemos sólo imprimimos una línea a la salida, la que leemos.

Podemos comparar el fichero de entrada con el temporal, para ver que son diferentes, como vemos en la Figura 6.47 con el comando 'diff source_file.txt destination_file.txt'. Vemos que hemos copiado sólo una línea y las diferencias entre ambos ficheros.

Ahora leeremos tres veces del fichero de entrada, para leer los 36 caracteres. Los escribiremos en el fichero temporal, y los leeremos para enviarlos a la salida estándar.


```
37 if ( (n=read(input, buffer, 11)) <= 0) err_sys("read_error input");
38 if (write(file_write,buffer, n)!=n) err_sys("write_error");
39 if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
40 write(1, buffer, n);
```

Código 122: Código ejemplo 'file4.c' donde vemos que sólo leemos una línea del fichero de origen 'source_file.txt'.

```
UOC1: ./file4
write fd = 3
read fd = 4
input fd = 5
0123456789
UOC1: █
```

Figura 6.46: Ejecución del programa './file4'.

```
UOC1: diff source_file.txt destination_file.txt
2,3d1
< ABCDEFGHIJ
< abcdefghij
UOC1: █
```

Figura 6.47: comprobación que el fichero origen './source_file.txt' y el fichero destino 'destination_file.txt' son ahora diferentes con el comando 'diff source_file.txt destination_file.txt'.

Mientras la secuencia sea primero leer de la entrada, escribir en el fichero temporal (pensemos que sólo tenemos un espacio de memoria para almacenar la información). Podemos hacer varias operaciones en el fichero temporal de escritura, y luego hacer las de lectura. Recordemos lo importante es primero escribir y luego leer, pues la operación de lectura no es bloqueadora si no hay información.

En el siguiente ejemplo, 'file5.c', haremos la lectura del fichero de entrada en varias iteraciones, como vemos en el Código 123.

```
1  /*
2   * Filename: file5.c
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/stat.h>
8  #include <sys/types.h>
9  #include <fcntl.h>
```

```
10 #include <unistd.h>
11
12 void err_sys(const char* cadena)
13 {
14     perror(cadena);
15     exit(1);
16 }
17
18 int main()
19 {
20     int file_write,file_read,n;
21     int input;
22     char buffer[10+1];
23
24     if ((file_write=open("./destination_file.txt",O_WRONLY|O_CREAT|O_TRUNC,0700)) < 0)
25         ↪ err_sys("error with destination file");
26     if ((file_read=open("./destination_file.txt",O_RDONLY)) < 0) err_sys("error with
27         ↪ destination (source) file");
28     if ((input=open("./source_file.txt",O_RDONLY)) < 0) err_sys("error with source
29         ↪ file");
30
31     printf("write fd = %d\n", file_write);
32     printf("read fd = %d\n", file_read);
33     printf("input fd = %d\n", input);
34
35     if ( (n=read(input, buffer, sizeof(buffer))) <= 0) err_sys("read_error input");
36     if (write(file_write,buffer, n)!=n) err_sys("write_error");
37     if ( (n=read(input, buffer, sizeof(buffer))) <= 0) err_sys("read_error input");
38     if (write(file_write,buffer, n)!=n) err_sys("write_error");
39     if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
40     write(1, buffer, n);
41     if ( (n=read(input, buffer, sizeof(buffer))) <= 0) err_sys("read_error input");
42     if (write(file_write,buffer, n)!=n) err_sys("write_error");
43     if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
44     write(1, buffer, n);
45     if ( (n=read(file_read, buffer, sizeof(buffer))) <= 0) err_sys("read_error");
46     write(1, buffer, n);
47
48     if (close(file_write)!=0) err_sys("error close write");
49     if (close(file_read)!=0) err_sys("error close read");
50     if (close(input)!=0) err_sys("error close input");
51 }
```

```
49  exit(0);  
50  
51 }
```

Código 123: Código ejemplo 'file5.c' donde vemos diversas lecturas del fichero de entrada 'source_file.txt'.

Generamos el fichero ejecutable 'file5' con el comando 'make file5 -f file.Makefile' y las opciones adecuadas como vemos en la Figura 6.48.

```
UOC1: make file5 -f file.Makefile  
gcc -c file5.c -Wall -Wno-unused-variable -I ./ -pthread  
gcc -o file5 file5.o -I ./ -pthread  
UOC1: █
```

Figura 6.48: Generación del ejecutable para 'file5'.

Ejecutamos el programa como vemos en la Figura 6.49. Como vemos imprimimos todo el fichero.

```
UOC1: ./file5  
write fd = 3  
read fd = 4  
input fd = 5  
0123456789  
ABCDEFGHIJ  
abcdefghij  
UOC1: █
```

Figura 6.49: Ejecución del programa './file5'.

Podemos comparar el fichero de entrada con el temporal, para ver que son nuevamente iguales, con el comando 'diff source_file.txt destination_file.txt', como vemos en la Figura 6.50 .

```
UOC1: diff source_file.txt destination_file.txt  
UOC1: █
```

Figura 6.50: comprobación que el fichero origen './source_file.txt' y el fichero destino 'destination_file.txt' son ahora nuevamente iguales con el comando 'diff source_file.txt destination_file.txt'.

6.7. Conclusiones

En este capítulo hemos hecho un repaso de algunos de los mecanismos de comunicación entre procesos. El objetivo principal es disponer de una solución para intercambiar información entre diferentes procesos o hilos de ejecución de un mismo sistema.

En el caso de los hilos de ejecución sabemos que siempre podemos utilizar las variables globales, que comparten los diferentes hilos de ejecución de un proceso.

El primero de los mecanismos que hemos visto, las zonas de memoria compartida, es similar a las variables globales pero para procesos que no comparten las mismas. Lo importante de trabajar tanto con una zona de memoria compartida como con las variables globales para los hilos de ejecución es que necesitamos un mecanismo de sincronización para que el acceso a la zona de memoria (zona crítica) sea controlado. Sin ello tendremos conflictos entre los diferentes accesos y nuestra solución probablemente no funcionará de forma correcta.

El segundo mecanismo de comunicación que hemos visto, es un poco más lento que la zona de memoria compartida pues accede al sistema de ficheros del sistema, pero tiene la ventaja que no necesita de elemento de sincronización. Las tuberías, con nombre o sin nombre, permiten el traspaso de información entre procesos o hilos de ejecución, con un mecanismo de bloqueo en caso de aún no disponer la información a leer en la tubería.

El último de los mecanismos que hemos visto, los ficheros, tiene todos los inconvenientes de los anteriores. Necesita de un mecanismo de sincronización adicional, pues el acceso a ficheros no es autoblocante, y accede al sistema de ficheros con lo que es poco eficiente.

Capítulo 7

Conclusiones

A lo largo de este documento hemos podido ver de manera práctica cómo utilizar el lenguaje de programación C y el sistema operativo GNU/Linux para desarrollar los diferentes módulos de un sistema de telecomunicación para el envío de datos entre un satélite de observación terrestre y la estación base, permitiendo realizar el control del mismo y obtener la información de los diferentes sensores embarcados. Esto nos ha llevado a explicar cómo comunicar procesos a través de *sockets* (tanto orientados como no orientados a conexión), así como generar nuevos procesos o hilos de ejecución (*threads*) y los mecanismos de sincronización (semáforos, *mutex* y variables condicionales) para poder paralelizar de manera segura la ejecución de tareas en nuestro sistema. Finalmente también hemos visto los mecanismos para comunicar los diferentes procesos/*threads* en un mismo sistema (tuberías con nombre y sin nombre, y memoria compartida) e incluso cómo guardar y leer información de manera persistente en ficheros.

Todos los conceptos tratados a lo largo del documento son ampliamente utilizados hoy en día para el desarrollo de sistemas, tanto de telecomunicación como informáticos, por lo que su estudio va más allá del interés puramente académico y te resultarán aplicables durante el desarrollo de tu actividad como ingeniero.

