

# INFORME PRÁCTICA 3

Alejandro Santorum Varela - alejandro.santorum@estudiante.uam.es

David Cabornero Pascual - david.cabornero@estudiante.uam.es

Sistemas Operativos - Pareja 7

Universidad Autónoma de Madrid

16-04-2018

## Contents

1	Introducción	2
2	Ejercicio 2	2
3	Ejercicio 2 mejorado	3
4	Ejercicio 3	3
5	Ejercicio 4	5
6	Ejercicio 5	6
7	Conclusión	7

# 1 Introducción

Ese documento recoge los ejercicios realizados en la tercera práctica de Sistemas Operativos.

Para cada ejercicio se comentará su diseño, su funcionalidad detalla y/o un análisis de las diferentes decisiones tomadas a la hora de enfrentarse a los problemas o dificultades encontradas.

## 2 Ejercicio 2

El primer ejercicio se basa en el concepto de **condición de carrera**: el resultado final de un ejercicio con múltiples procesos ejecutándose depende de la secuencia de acción de dichos procesos, provocando que el resultado pueda variar si las variables compartidas no están bien protegidas.

Se nos pide realizar un ejercicio que cree tantos procesos como indique el usuario por línea de comandos. Cada proceso una vez creado dormirá durante un tiempo aleatorio, y a continuación solicitará al usuario que dé de alta a un cliente (nombre), incrementará la variable compartida en una unidad y enviará al proceso padre la señal SIG\_USR1 para indicar que el proceso ha acabado.

Tal como se plantea este ejercicio podemos sostener que tiene algunas fujas de corrección. Por un lado, el tiempo de creación de los procesos hijos es insignificante, humanamente hablando, por lo que todos los procesos pedirán al usuario dar de alta un cliente a la vez. Por otro lado, no hay nada protegiendo la variable compartida; si no fuera por el hecho que lo único que hacen es incrementarla (podrían decrementarla algunos y incrementarla otros procesos), podríamos tener diferentes resultados.

A continuación mostramos una salida que ejemplifica lo dicho anteriormente:

```
AlejandroSantorum@DESKTOP-GC6HCIA: /mnt/c/users/Alejandro_Santorum/Desktop/SOPER-PI $ ./ejercicio2 3
Alta del cliente del proceso 1: Alta del cliente del proceso 2: Alta del cliente del proceso 3: cliente1
Nombre: cliente1
Identificador: 1
cliente2
Nombre: cliente2
Identificador: 2
cliente3
Nombre: cliente3
Identificador: 3
AlejandroSantorum@DESKTOP-GC6HCIA: /mnt/c/users/Alejandro_Santorum/Desktop/SOPER-PI $
```

Cada vez que introducimos un nombre, un proceso capta el nombre y lo muestra por pantalla.

La función fgets se encola y podemos inscribir todos los clientes. Scanf no se encola, y solo se daría de alta un cliente de los N procesos

Los tres procesos dan de alta un cliente al mismo tiempo

Se comenta en la imagen, pero podemos enfatizarlo: en nuestro programa, aunque todos los procesos nos pidan un nombre, la función fgets(...) se encola y nos permite introducirlos todos tal y como vemos en la imagen. Si utilizásemos la función scanf(...) esto no pasaría y solo podríamos introducir un nombre, y  $N - 1$  procesos fallarían estrepitosamente

### 3 Ejercicio 2 mejorado

Como continuación del ejercicio 2, tenemos que mejorarlo para que no se produzcan los percances comentados, utilizando lo que sabemos de memoria compartida y semáforos. Esta solución la podemos encontrar en el `ejercicio2_solved`.


Básicamente lo que hacemos es crear un semáforo. Mientras un proceso pida un cliente, ningún otro proceso podrá hacerlo. Así pues, cuando un proceso llega a la zona crítica baja el semáforo que será levantado cuando el proceso padre haya dado de alta el nuevo cliente.

Comentar que inicialmente se producían algunos errores con los semáforos (tal y como te comentamos en clase). En un principio era debido a que el sistema no funciona muy bien cuando operamos sobre un semáforo en procesos distintos (por ejemplo, cuando uno lo baja y otro lo sube). No obstante, después de leer varios artículos y posts por internet junto con el man de C, descubrimos, con la colaboración de otras parejas, que la flag **SEM\_UNDO** no funciona tal y como se muestra en la documentación y que cuando un proceso hijo finaliza envía una señal al padre, `System call`, produce ciertos fallos en los semáforos.

Comentar que este percance ya se había tenido en la práctica anterior con el ejercicio 9, y que había sido solucionado con una espera activa y la bandera **IPC\_NOWAIT**. En esta hemos mejorado y podemos controlar más robustamente los errores ya que no usamos espera activa, y usamos en el campo flag **0**.

Ahora mostramos la salida del programa mejorado:

```
AlejandroSantorum@DESKTOP-GC6HCIA: /mnt/c/users/Alejandro_Santorum/Desktop/SOPER-P3$ ./ejercicio2_solved 3
Alta del cliente del proceso 1: cliente1
Nombre: cliente1
Identificador: 1
Alta del cliente del proceso 2: cliente2
Nombre: cliente2
Identificador: 2
Alta del cliente del proceso 3: cliente3
Nombre: cliente3
Identificador: 3
AlejandroSantorum@DESKTOP-GC6HCIA: /mnt/c/users/Alejandro_Santorum/Desktop/SOPER-P3$
```

 Ahora cada proceso pide un cliente de una forma ordenada: de uno en uno.

### 4 Ejercicio 3

El ejercicio 3 es el problema del productor-consumidor. En este caso solo necesitamos dos procesos: un productor que introducirá caracteres y los introducirá en un array, y un consumidor, que retirará los caracteres y los mostrará por pantalla.

Necesitamos tres semáforos: uno para indicar cuando el array (que simula una estantería de productos) está lleno, otro para indicar cuando el array o estantería está vacío, y por último un semáforo mutex para proteger las variables conflictivas (sección crítica) y para

permitir que un único proceso tenga acceso al array en un instante determinando.

· **PRODUCTOR:** down(vacio) > down(mutex) > sección crítica-producir\_producto()  
> up(mutex) > up(lleno)

· **CONSUMIDOR:** down(lleno) > down(mutex) > sección crítica-consumir\_producto()  
> up(mutex) > up(vacio)

Comentar que la implementación de este ejercicio se ha generalizado: el usuario debe introducir el tamaño del array por parámetro de entrada del programa. El resultado va a ser el mismo, pero un tamaño menor hará que los semáforos que controlan el tamaño del array trabajen más duramente para que un productor no produzca si la estantería está llena o un consumidor no consuma si no hay productos.

Por otro lado, los productos son producidos circularmente, empezamos con la A hasta la Z y después continuamos con los números desde el 0 al 9. Una vez completado el ciclo, volvemos a empezar por la A. No obstante, para ceñirse al enunciado hemos puesto una constante (define) a 36, que es el número de productos a producir y posteriormente a consumir. Si quisésemos producir-consumir más que un ciclo, solo tendríamos que cambiar esta constante.

Ahora abajo mostramos la salida del programa:

```
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/SOPER-P3$ ./ejercicio3 10
Producto recogido: A
Producto recogido: B
Producto recogido: C
Producto recogido: D
Producto recogido: E
Producto recogido: F
Producto recogido: G
Producto recogido: H
Producto recogido: I
Producto recogido: J
Producto recogido: K
Producto recogido: L
Producto recogido: M
Producto recogido: N
Producto recogido: O
Producto recogido: P
Producto recogido: Q
Producto recogido: R
Producto recogido: S
Producto recogido: T
Producto recogido: U
Producto recogido: V
Producto recogido: W
Producto recogido: X
Producto recogido: Y
Producto recogido: Z
Producto recogido: 0
Producto recogido: 1
Producto recogido: 2
Producto recogido: 3
Producto recogido: 4
Producto recogido: 5
Producto recogido: 6
Producto recogido: 7
Producto recogido: 8
Producto recogido: 9
AlejandroSantorum@DESKTOP-GC6HCIA:/mnt/c/Users/Alejandro_Santorum/Desktop/SOPER-P3$
```



## 5 Ejercicio 4

El ejercicio 4 podríamos considerarlo como una instancia muy sencilla del problema de lectores-escriptores. Sin embargo, no podemos decir al 100% que sea así, ya que el lector y el escritor no trabajan a la vez (tal y como indica el enunciado), por lo que no tenemos que utilizar semáforos ni tener cuidado con la concurrencia.

El obetivo principal de este ejercicio es iniciarse con la función `mmap(...)` de C.

Se nos pide iniciar un hilo que escriba en un fichero un número aleatorio entre 1000 y 2000 de números aleatorios entre 100 y 1000. Una vez que haya acabado, el proceso principal iniciará otro hilo, el cual utilizará `mmap(...)` para leer el contenido del fichero.

Comentar que este ejercicio se podría haber hecho de varias formas: en escritor también podría haber escrito ayudándose de la función `mmap(...)`. Nosotros no lo hemos hecho así. Por otro lado, nosotros hemos abierto el fichero en el proceso padre y al segundo hilo le hemos pasado el descriptor del fichero por parámetro, pero se podría haber pasado el nombre del fichero y en el hilo obtener el descriptor del fichero con `open(...)`.

Independientemente de lo anterior, el resultado final es idéntico, y aquí mostramos la salida:

```
AlejandroSantorum@DESKTOP-GC6HCIA: ~/C:/Users/Alejandro_Santorum/Desktop/sofer-P/$ ./ejercicio4
455 805 819 921 277 402 792 350 599 530 666 428 562 957 925 672 746 227 646 114 318 223 824 241
827 928 162 955 573 177 273 697 902 414 157 118 512 156 314 974 912 866 340 586 438 784 561 701
630 692 873 496 932 458 834 716 920 534 294 956 929 233 893 677 489 658 353 808 377 502 303 268
213 814 247 771 167 956 147 569 258 316 818 760 691 971 676 784 184 221 568 170 162 284 515 838
3 597 622 507 719 189 578 782 374 993 619 890 773 666 131 773 850 933 886 848 982 770 913 986 700
2 748 202 524 633 950 506 403 863 491 102 410 639 850 310 708 535 534 374 741 264 660 136 472 727
0 425 619 634 700 360 798 359 397 270 986 103 845 398 269 493 963 927 789 729 209 717 445 797 949
6 755 225 643 543 855 752 260 299 549 209 224 424 392 939 918 660 853 837 546 401 455 693 648 333
2 484 847 455 320 393 757 675 987 404 908 222 470 104 805 797 364 203 879 749 144 504 988 737 290
6 374 900 529 253 649 573 657 637 310 847 163 189 932 252 534 303 844 362 421 891 410 834 693 132
0 113 442 444 375 764 334 685 597 928 718 829 728 381 681 105 580 860 657 679 567 461 426 747 822
2 897 267 241 553 846 708 914 272 455 736 882 593 765 940 310 934 596 941 545 597 946 820 833 635
7 681 585 233 621 129 731 566 850 564 201 541 559 143 833 446 674 507 229 472 322 466 115 746 616
1 400 583 297 530 955 520 896 971 265 512 802 790 915 332 786 968 398 462 605 599 660 272 530 424
1 794 899 940 255 503 539 816 675 970 239 363 894 429 910 773 528 345 952 210 879 661 747 933 265
4 319 922 610 271 131 489 833 778 421 999 132 571 281 696 730 395 901 682 407 145 790 823 729 714
8 252 626 529 834 934 574 624 757 302 338 670 584 115 939 413 285 570 461 376 712 681 499 342 733
0 964 763 409 424 139 121 104 539 363 738 838 557 521 170 272 535 931 139 176 320 740 650 183 966
6 944 785 187 983 861 407 723 511 491 689 391 641 217 173 440 223 694 203 893 624 289 702 576 381
1 156 142 574 259 936 198 449 637 675 731 586 466 841 620 930 299 811 437 444 187 827 448 773 941
2 195 900 192 532 343 279 359 692 953 299 556 801 943 228 365 605 680 886 309 707 667 850 832 796
5 459 430 455 345 640 161 912 489 893 708 515 524 363 302 321 619 371 213 775 532 537 273 873 220
3 741 465 274 855 239 706 391 412 579 511 676 746 514 587 105 342 272 724 500 312 688 297 414 563
1 198 883 243 823 382 456 510 579 771 974 963 179 118 147 908 910 135 477 265 298 801 661 166 515
6 180 839 972 558 104 269 358 665 335 774 132 750 415 885 356 598 708 962 662 674 490 107 997 427
3 239 846 300 201 507 875 591 515 872 919 448 961 256 269 322 424 926 665 430 654 566 441 552 725
1 447 800 216 111 230 771 578 571 322 302 588 908 860 312 475 766 529 183 517 948 893 676 340 293
9 956 391 619 139 809 566 932 484 806 225 735 309 637 107 838 526 570 812 214 250 799 933 601 362
1 227 870 451 940 984 602 738 917 202 100 239 377 128 340 131 683 531 742 629 340 491 382 616 172
0 639 503 902 381 132 241 773 414 758 845 836 454 723 230 442 946 406 557 136 933 611 210 160 403
3 298 105 139 756 141 973 367 252 133 670 353 425 766 584 324 682 285 763 101 789 583 454 533 176
2 149 130 797 813 132 586 396 487 118 472 607 954 886 329 746 460 685 737 940 180 482 562 548 495
4 876 652 498 612 592 579 995 153 127 489 389 976 567 652 750 994 526 574 551 198 211 141 355 145
81 146 994 307 621 544 405 733 586 661 778 605 192 229 207 754 772 524 290 967 338 339 753 632 382
50 521 327 974 711 293 311 951 947 844 332 540 796 146 261 143 861 663 439 667 372 355 919 386 903
52 516 950 515 856 617 787 210 535 172 113 291 133 342 390 762 126 939 911 136 560 770 341 394 579
93 102 207 231 913 244 692 683 485 986 262 877 957 784 959 400 494 893 260 154 341 876 331 476 702
94 206 864 987 367 918 327 242 248 703 845 191 386 385 938 311 907 978 238 636 249 980 510 112 502
15 511 319 492 650 856 642 629 366 654 131 134 516 219 938 183 676 935 857 997 919 898 333 229 180
34 898 762 769 755 758 687 653 992 817 733 710 861 842 906 205 362 616 861 102 719 267 599 463 201
80 329 619 796 190 621 514 357 219 878 459 580 640 527 837 216 681 928 592 597 786 722 270 332 504
rum@DESKTOP-GC6HCIA: ~/C:/Users/Alejandro_Santorum/Desktop/sofer-P/$
```

## 6 Ejercicio 5

Finalmente llegamos al ejercicio final de la práctica. Este se basa en el aprendizaje de las funciones de las colas de mensajes.

El objetivo del ejercicio es: un proceso A lee de un fichero de entrada texto e introduce en la cola de mensajes trozos del texto de longitud máxima de 2KB. El proceso B coge de la cola de mensajes los trozos de texto, los transforma escribiendo la siguiente letra de cada caracter y se lo envía al proceso C a través de la cola de mensajes. Finalmente, el proceso C lee de la cola de mensajes el texto modificado por B y lo escribe en un fichero de salida.

Comentar en primer lugar que para que el proceso B sepa que ya ha finalizado de computar todos los trozos de texto el proceso A le envía una señal cuando A haya acabado. Sin embargo, A puede haber acabado pero B no, por lo que B tiene que comprobar que, además de que A haya acabado, la cola de mensajes este vacía. Esto ocurre simétricamente para el proceso C y el proceso B.

Para conocer el número de mensajes restantes en la cola se ha utilizado `msgctl` con la flag `textbfIPC_STAT` que nos devuelve información de la cola de mensajes. Debido a que el número obtenido de mensajes en la cola con esta función es la suma de todos los mensajes de diferente tipo, hemos tenido que crear dos colas independientes: una para pasar mensajes de A a B y otra para B y C. Esto es para poder saber cuando finalizar los procesos B y C.

Aunque utilicemos dos colas independientes, los mensajes entre A y B son del tipo 1, y los mensajes entre B y C son de tipo 2. Esto es así para demostrar que sabemos usar mensajes de diferente tipo, aunque para este ejercicio con esta implementación no sea necesario.

Este ejercicio no produce ninguna salida por terminal, pero adjuntamos dos ficheros de texto, "origen.txt" y "destino.txt", que ya han sido utilizados por el programa, por lo que tienen la salida esperada, la cual se puede comprobar que es del mismo tamaño y que cada letra del origen.txt es la anterior a la del destino.txt.

**NOTA:** En la entrega anticipada se detectó un warning que no había saltado en nuestros ordenadores, a pesar de usar la flag `-Wall`. Después de buscar información exhaustivamente, descubrimos que esto se debe a un *bug* de aparición reciente. A continuación mostramos los reportes de *StackOverflow* y de una página especializada en *bugs*:

Yes, this appears to be related to [GCC bug 53119](#). It goes away if you change the C declaration to `{{0}}`.

GCC Bugzilla - Bug 53119
-Wmissing-braces wrongly warns about universal zero initializer {0}
Last modified: 2017-04-19

Home | New | Browse | Search |  Search [?] | Reports | Help | New Account | Log In | Forgot Password

Bug 53119 - -Wmissing-braces wrongly warns about universal zero initializer {0}

**Status:** RESOLVED FIXED  
**Alias:** None  
**Product:** gcc  
**Component:** c ([show other bugs](#))  
**Version:** unknown  
**Importance:** P3 enhancement  
**Target Milestone:** ---  
**Assignee:** Not yet assigned to anyone

**Reported:** 2012-04-25 17:04 UTC by Rich Felker  
**Modified:** 2017-04-19 12:23 UTC ([History](#))  
**CC List:** 10 users ([show](#))  
**See Also:**  
**Host:**  
**Target:**  
**Build:**  
**Known to work:**  
**Known to fail:**  
**Last reconfirmed:** 2012-05-03 00:00:00

Utilizando lo anterior hemos modificado nuestro código a esto. La compilación a nosotros nos sigue resultando perfecta, pero no hemos podido comprobar al 100% que se soluciona el problema. Aún así, *StackOverflow* suele proporcionar muy buenas soluciones, y no fueron pocos los usuarios que aconsejaron la mostrada solución.

```
struct msqid_ds buf = {{0}};
```

## 7 Conclusión

Esta ha sido una práctica corta pero bastante productiva. Hemos obtenido muy buenas herramientas para el proyecto final y hemos aprendido nuevas formas de comunicar procesos.

Posiblemente esta práctica este menos refinada que las dos anteriores debido al poco tiempo que hemos tenido, pero creemos que ha salido todo exitosamente y hemos hecho una buena práctica.

Agradecer el trabajo del profesor, que a pesar de tener el tiempo muy limitado, le ha echado un ojo a la entrega de la práctica anticipada.