Guillem Escriba Molto - u188331

Sistemas Operativos

Informe Práctica 5

El objetivo principal de esta última práctica era combinar lo aprendido referente a la comunicación entre procesos y los distintos métodos de llevarla a cabo con la sincronización que hemos ido viendo a lo largo del curso.

En el primer apartado, debemos sincronizar con semáforos launch_file.c y launch_shm.c, ambos métodos (file y shared memory) de comunicación entre procesos requieren de un método de sincronización explicita entre los procesos que lo utilizan ya que si no pueden haber problemas de sincronización tales como sobreescrituras, en el primer caso, sobreescribir el file y en el segundo la posición de memoria en cuestión. Para evitarlo se puede hacer uso de semáforos binarios (inicializados en 1) para controlar el acceso a los recursos comunes para los distintos procesos. En el primer método, el de file, colocaríamos el sem_wait() en el pids_file.c justo antes de hacer el primer acceso de lectura “read(fd,&n, sizeof(int));” y el sem_signal() tras realizar el último acceso de escritura en “write(fd,&pid,sizeof(int));” delimitando como zona de exclusión mutua la que hay entre ambas funciones del semáforo, ya que deben asegurarse de que al acabarse el quantum de un proceso, otro no sobreescriba el mismo recurso. Por otro lado, en shared memory tan solo debemos incluir en la zona mutex aquellas instrucciones que hacen acceso al array “a” que es la que usan los procesos para comunicarse. por lo que el sem_wait() va en “a[0]++;” y el sem_signal() en “n = a[0];”. Por otro lado, ambos códigos finalizan con un sem_close() y empiezan con un sem_unlink() para reutilizar los semáforos.

Por otro lado en el segundo apartado debemos implementar los métodos de comunicación entre procesos que no requieren sincronización que son las pipes y los sockets. Con las pipes, todo el código, desde la creación de procesos hasta el uso de pipes se hace en el mismo archivo. En cambio, en el de los sockets se debe hacer un .c para el cliente y otro para el servidor. En este caso, no se necesita crear múltiples procesos ya que lo que se hace es acceder al servidor con distintos clientes. En el servidor es donde se genera de manera aleatoria esta semilla que se utilizará para la simulación. Se puede comprobar a simple vista tras ejecutar ambos métodos que el uso de pipes es considerablemente más rápido que el de sockets, ya que al fin y al cabo, con sockets el cliente debe hacer una solicitud online, esperar a que el servidor la acepte y luego acceder a la semilla, mientras que con pipes es tan fácil como acceder al canal de lectura y cogerla. Por otro lado, los sockets son bidireccionales lo que permite una comunicación fluida en ambos sentidos y lo que es más importante, la comunicación entre procesos no se limita simplemente al mismo dispositivo, sino que se puede comunicar con cualquier proceso de cualquier dispositivo a través del servidor y del puerto.