Respuestas Orientadoras e incompletas (en algunos casos) a las preguntas de revisión. Favor tomarlas como orientadoras.

Revisión – Unidad III – Sincronización entre Procesos

1. ¿Podría aplicarse la exclusión mutua sobre una sección crítica?

Si, es lo que debe hacerse; si se aplicara una exclusión mutua sobre una sección no crítica, ésta no tendría mayor sentido.

2. Unir con flechas:

Inanición	mutual exclusion
Condición de Carrera	livelock
Exclusión Mutua	deadlock
Interbloqueo	race condition
Circulo Vicioso	startvation

Inanición--> startvation
Condición de Carrera-->race condition
Exclusión Mutua--> mutual exclusion
Interbloqueo-->deadlock
Circulo Vicioso-->livelock

3. ¿Puedo coordinar n procesos teniendo en cuenta las velocidades de ejecución de cada proceso?

No, no puede predecirse la velocidad relativa de ejecución de los proceos.

4. ¿Qué relación Ud. Encuentra entre la dificultad de encontrar errores y la condición de carrera?

La dificultad reside en que los resultados obtenidos -cuando hay una condición de carrera- depende del orden de ejecución de las intrucciones de n procesos (tal vez, con n hilos cada uno) y ésto a su vez esta determinado por el planificador de procesos y otros factores, con lo cual, no es fácilmente reproducible, se rompe con el modelo determinístico al cual debería responder la ejecución de procesos.

5. ¿Qué tipo de problemas puede provocar la implementación de la exclusión mutua?

Interbloqueo (deadlock) e Inanición (startvation)

6. ¿Qué sucede si un proceso queda indefinidamente dentro de una sección crítica?

El resto de los procesos -que también ejecutan esta sección crítica- quedarán bloqueados intentando entrar en ella.

7. Responda V o F: La espera activa es una buena solución de software, puesto que no consume tiempo de CPU.

Falso. La espera activa consume tiempo de cpu.

8. Responda V o F: El algoritmo de Dekker es mejor que el algoritmo de Peterson, porque tiene menor carga de procesamiento.

Falso. El algoritmo de Dekker es mas complejo, menos elegante, pero no puede afirmarse que tenga una menor carga de procesamiento.

9. ¿La deshabilitación de interrupciones es una técnica de hardware que da soporte a la exclusión mutua en un SMP?

Falso, porque en un SMP hay n cpu's que pueden estar ejecutando n procesos al mismo tiempo y un SMP no posee un mecanismo de interrupción de n cpu's que comparten la misma memoria.

10. Responda V o F: Deshabilitar interrupciones en un SMP implica evitar el entrelazado de procesos.

Falso, idem anterior.

11. Responda V o F: Deshabilitar interrupciones en un computador con una sola CPU implica evitar el entrelazado de procesos.

Verdadero.

12. ¿Las soluciones de hardware son mejores que las soluciones de software, en cuanto al problema de la exclusión mutua?

No, tanto las soluciones de software como las de hardware son insuficientes para tratar el problema de la exclusión mutua, se requiere soporte del SO.

13. ¿Las instrucciones de CPU atómicas surgen como una mejora a la deshabilitación de interrupciones?

Si, mas que nada para permitir implementar un mecanismo de exclusión mutua por hardware en un SMP. Si bien permiten mejorar la eficiencia de la cpu permitiendo el entrelazado de procesos, la mejora se ve disminuida debido a que las operaciones atómicas requieren de espera activa.

14. Responda V o F: Tanto las soluciones de software como las instrucciones atómicas de hardware evitan la espera activa.

Falso, en ambos casos hay espera activa.

15. Responda V o F: Las instrucciones de CPU atómicas permiten implementar algoritmos de exclusión mutua de forma más eficiente que las soluciones de software.

Falso. Idem anterior.

16. ¿Por qué razón Ud. Cree que el SO es el que puede dar el mejor soporte a la exclusión mutua?

Porque el SO es quien controla al dispatcher, a la ejecución de todos los procesos del computador, tiene una posición privilegiada para tener una visión global en cuanto a todos los procesos y facilitar mecanismos de exclusión entre ellos.

17. Responda V o F: Según Dijkstra, la exclusión mutua no puede resolverse usando semáforos.

Falso.

18. Responda V o F: semSignal(s) es una instrucción atómica, no interrumpible, que envía una señal al semáforos.

Verdadero.

19. Responda V o F: semWait(s) incrementa s.

Falso, semWait(s) decrementa s.

20. Responda V o F: semSignal(s) decrementa s.

Falso, semSignal(s) incrementa s.

21. Responda V o F: Una de las ventajas de los semáforos es que su valor se puede cambiar sin necesidad de usar semWait() o semSignal().

Falso, no se puede manipular el valor de un semáforo fuera de semSignal() y semWait().

22. Responda V o F: Un semáforo binario es más rápido y potente que un semáforo no binario.

Falso, solo son más faciles de usar, pero ello no implica que sean más potentes.

23. ¿Qué es un proceso productor?

Proceso que genera algún tipo de dato, poniéndolo en un buffer del que luego será extraído por un proceso consumidor.

24. ¿Qué es un proceso consumidor?

Proceso que extrae datos del buffer cargado previamente por el productor.

25. ¿Por qué razón se desarrollaron los monitores?

Debido a la dificultad y a la probabilidad de error que implica el uso de semáforos distribuidos en distintas partes del sistema. No es fácil seguir la secuencia de ejecución de semWait() y semSignal() en distintas partes del sistema para los distintos semáforos.

26. ¿Qué relación hay entre los monitores y la OOP?

Dos características comunes de los monitores y la OOP: 1. variables locales, solo accesibles a través

de procedimientos del monitor y no por otros procedimientos externos (encapsulación); 2. Un proceso entra en un monitor invocando el uso de sus procedimientos (se accede al interior del objeto solo a través de sus métodos).

27. ¿Qué es una variable de condición?

Son variables al interior del monitor que permiten la sincronización entre distintos procesos: cwait(c) bloquea el proceso esperando por la condicion c; csignal(c) retoma la ejecución de cualquier proceso bloqueado a la espera de la condicion c.

28. En cuanto a cola de mensajes, ¿Qué tipo de primitivas receive() existen?

Las categorías más comunes son: bloqueante (queda el proceso bloqueado, en espera no activa, hasta que llegue algún mensaje), no bloqueante (no queda bloqueado esperando, devuelve algun valor indicando la llegada o no de un mensaje). También existe el receive() como comprobación de llegada de mensaje (al estilo de poll() o select()). Otra clasificación puede ser receive() explícito (se indica de qué proceso se espera recibir un mensaje) e implícito (recibe mensaje de cualquier proceso).

En cuanto a los trabajos prácticos:

1. ¿Es posible aplicarle cierto time-out a una operación bloqueante read()?

Si, a través del uso de las funciones poll(), select() o bien usando la señal de alarma.

2. ¿Las señales se lanzan como eventos sincrónicos o asincrónicos?

Las señales pueden ser lanzadas en forma sincrónica o asincrónica. Generalmente, en forma sincrónica cuando están vinculadas a acciones dentro y bajo el control del programa. En forma asincrónica cuando se lanzan en respuesta a eventos que están fuera del control del programa que las recibe (ejemplo, el proceso A responde asincrónicamente a la señal SIGUSR1 que fue enviada por el proceso B).

3. ¿Qué sucede si un programa entra en loop y comienza a lanzar en forma repetitiva una misma señal a otro proceso?

Supongamos que el proceso A entra en loop enviado la señal SIGUSR1 al proceso B en forma repetitiva, en tal caso, el SO, encolará una sola vez la señal SIGUSR1 para el proceso B.

4. ¿Una señal de interrupción tiene mayor prioridad que una señal de usuario?

No, todas las señales tienen la misma prioridad, no hay señales más prioritarias que otras.

5. ¿Qué cuidados debe tener el programador que utiliza la función pause()?

El cuidado es que no se produza una señal en forma temprana, antes de lo previsto por el programador, antes de que se ejecute pause(), porque, si esta fuese la única señal generada, el proceso quedará bloqueado indefinidamente, ya que no arribarán más señales a este proceso, impidiendo la finalización de la función pause().

6. ¿La llamada a la función sleep() implica una espera activa del proceso que la invocó?

No, sleep() bloquea el proceso actual, no es una espera activa.

7. En un contexto de multithreading, ¿Tiene sentido escribir sentencias luego de invocar a la función pthread_exit()?

No, estas sentencias no serán ejecutadas.

8. ¿Qué sucede cuando un thread invoca a la función pthread_join()?

Si el thread t1 invoca la función join(t2) implica que t1 quedará bloqueado hasta que finalice t2; si t2 ya estaba finalizado al momento de invocar el join(), join() no hace nada y t1 continua con su ejecución.

9. ¿Qué tipo de atributo debe tener un thread sobre el cual se ejecuta una operación de join?

El thread debe ser de tipo "joineable" (atributo PTHREAD_CREATE_JOINABLE). Puesto que un thread de tipo "joineable" requiere de una serie de controles, por defecto, todo thread es "no joineable" mientras no se declare lo contrario.

10. ¿Qué diferencia hay entre un semáforo y una variable de condición?

Un semáforo actúa de forma distinta a una variable de condicion. Por ejemplo, un semáforo binario o mutex simplemente es algo de tipo "pasa" o "no pasa", incrementando o decrementando el valor del semaforo, permitiendo la implementación de una exclusión mutua. Una variable de condición se usa en combinación con mutex (una especie de semáforo binario, como se explicó anteriormente), una vez que se ingresó en la sección crítica, se pregunta por una determinada condición y en tal caso, se señala una variable de condición (csignal(c)) ; se supone que hay otro proceso esperando por la señalización de esa condicion (cwait(c)). El csginal(c) provoca el bloqueo del proceso actual y la liberación del mutex actual, se adquiere el mutex y se desbloquea el proceso que estaba esperando por la condicion c (cwait(c)).

11. ¿Para qué sirve el comando "time"? ¿Cómo se usa?

El comando unix time sirve para contabilizar los recursos utilizados por un proceso
Forma de uso: time [opciones] <proceso> Ejemplo: $ time ./tp81
En el ejemplo anterior se obtendría una salida que indicaría (expresado en horas, minutos, segundos) el Real Time (tiempo real, wall clock) transcurrido en la ejecución del programa, el User Time (tiempo que el proceso corrió en modo usuario, no privilegiado) y el System Time (tiempo que el proceso corrió en modo kernel o núcleo o privilegiado).

12. Supongamos que tengo la sección crítica C y hay dos funciones f1() y f2() que acceden a dicha sección crítica. La función f1() tarda unos 200 milisegundos, f2() tarda unos 500 milisegundos; entonces ejecuto f1() y luego f2(); al comienzo de f2() pongo una demora de unos 300 milisegundos. Ambas funciones se lanzan con pthread_create(). Responda:

a) Ud. es el Analista a cargo del desarrollo de la aplicación X y un programador vino con este planteo para un proceso que Ud. le encargó ¿Aprueba Ud. este diseño?

Falso, no puede aprobarse este tipo de sincronización basado en supuestos tiempos de duración de los procesos.

b) ¿Funciona el diseño planteado en el punto 12?

Eventualmente podría funcionar, pero no se puede asegurar un funcionamiento libre de errores ya que esta supeditado al tiempo real que dure la ejecución de cada proceso, lo cual puede variar por múltiples factores (planificador, tiempos de I/O, carga actual del sistema, etc.).

c) ¿Qué otro diseño podría aplicarse al proceso del punto 12?

La forma correcta de diseñar esto sería implementar un mutex para las funciones f1() y f2() sin importar los tiempos de ejecución de los procesos, sin implementar demoras que impacten en el rendimiento de este proceso. Sería un diseño sólido y de mejor rendimiento que el propuesto.

13. ¿Por qué razón Ud. cree que las prácticas proponen devolver valores distintos en la función main() para las distintas situaciones de error detectadas?

Se pueden capturar los valores de retorno en el shell del SO y saber con exactitud que tipo de error se ha dado en la aplicación. Esto facilita la ubicación del error y su corrección.

14. ¿Qué es una señal nula? ¿Para qué sirve?

La señal nula es aquella cuyo valor es 0 (nulo decimal) y generalmente se utiliza para determinar si un proceso esta en este momento en ejecución o no, si la función kill() devuelve -1 y errno = = ESRCH indica que el proceso en cuestión no existe.

15. ¿Qué significa que una función no es re-entrante?

Una función no re-entrante es una función que utiliza datos que estan fuera del stack, por ejemplo una variable estática o global o dinámicamente asignada (usando malloc(), calloc(), realloc()) la cual es accedida desde el código de esta función (apuntando siempre a la misma dirección) y que habiendo dos o más llamadas concurrentes a dicha función pudiera haber una interferencia en su funcionamiento. La interferencia se produce sobre el dato que esta fuera del stack (su copia privada de datos) y sobre el cual no hay un mecanismo de exclusión mutua implementado.

16. ¿Una variable de tipo volatile implica que su valor vuelve a su valor por defecto luego de cierto tiempo o bajo determinada condición?

No, una variable debe declararse como volatile si la misma es accedida en forma asincrónica desde una o más funciones; esto inhibe ciertas optimizaciones del compilador y permite que las actualizaciones de la variable se hagan de forma atómica.

17. ¿Es posible cambiar la imagen del proceso actual por la de otro proceso?

Si, es lo hace el shell cuando combina la función fork() con la función exec*().

18. ¿Qué significa que un programa no se comporte acorde con un "Modelo Determinista"?

Que, dada una misma entrada, no siempre se obtiene la misma salida.

19. ¿Qué es un coproceso?

Se trata de dos procesos que colaboran de forma muy estrecha, como es el caso de la implementación de pipes half-duplex entre procesos emparentados, en donde el proceso padre escribe en el stdin del proceso hijo y lee del stdout del proceso hijo.

20. Tengo un programa interactivo (requiere que el operador tipee datos por teclado y obtenga salidas por pantalla), ¿Puedo -a través de otro proceso- reemplazar al operador (obviamente suponiendo un comportamiento determinístico)?

Si, a través del uso de pipes sin nombre (unnamed pipes) entre proceso emparentados, generamos un proceso padre que ejecute a otro proceso hijo usando fork() + exec*() y luego el proceso padre será quien simule la interacción humana.

21. Desde el punto del vista del SO, ¿Cómo se identifica cada recurso de tipo IPC y Cómo podemos consultarlo y/o administrarlo?

Se identifica a través de un número entero positivo, este número es de tipo key_t y esta definido dentro del archivo de cabecera sys/types.h . Si dos o más programas pretenden compartir este recurso IPC, los procesos deberán conocer este número para poder accederlo de forma unívoca. El comando ipcs permite la administración (por fuera de las aplicaciones) de los recursos IPC actualmente existentes en el kernel del SO.

22. De la misma forma que un SO tiene su API (system calls), cualquier aplicación que desarrollemos también podría proveer (hacia el exterior) su propia API, ¿En qué archivo se especificaría esta API?

La api externa de la aplicacion se indica en los archivos de cabecera (headers) *.h. Puesto que, si la API luego se distribuirá como una librería, quien pretenda usarla debe conocer su interfase y la misma esta descripta en cada uno de los prototipos de las funciones a utilizar -desde el exterior- dentro del archivo .h y su forma de uso debe indicarse en el manual del sistema. Si una función no será ejecutada desde el exterior, sino que su reuso será interno, simplemente puede definirse su prototipo dentro de los archivos .c que tienen su implementacion (se debe quitar su prototipo del archivo de cabecera .h para que no sea "visible" desde el exterior, puesto que C no tiene los cualificadores de control de acceso de los lenguajes orientados a objetos, tales como: friend, public, protected, private, etc. Para C todas las funciones son public).

23. ¿Qué otra razón -además de la indicada en el TP VII Cola de Mensajes- se le ocurre a Ud. que un Analista puede tener para definir una API implementada en la librería X para ser reusada en las aplicaciones A,B,C,D?

Existen varias razones, definir una API implica definir la interfase de una capa de software con la cual interacturán las aplicaciones A,B,C,D; de esta forma, el Analista puede implementar cambios hacia el interior de la capa de software, mientras que no altere la API previamente definida, las aplicaciones seguirán funcionando como siempre, sin necesidad de cambio. Facilita el training del personal, es cuestion de aprender a utilizar esta API para poder dar soporte a las aplicaciones A,B,C,D sin necesidad de conocer los detalles de implementación de esta capa de software. Si hubiese algún error dentro de esta capa de software, será cuestión de modificar una llamada de esta API y todas las aplicaciones asimilarán el cambio. ...

24. Responda Verdadero o Falso y justifique: "La memoria compartida es la forma más lenta de IPC”.

Falso, se considera la forma más rápida de IPC porque no se requiere que los datos de comunicación entre procesos requieran ser copiados a areas de memoria locales a los procesos que se comunican.

25. Enumere algunos ejemplos de funciones bloqueantes vinculadas con IPC.

msgrcv() intenta recibir un mensaje de la cola de mensajes, si ésta esta vacía el proceso quedará bloqueado. Otro ejemplo puede ser la función semop() usada para incrementar/decrementar los valores de un semaforo, si un proceso pretente decrementar el semaforo actualmente usado por otro usuario, éste quedará bloqueado hasta que el semafóro sea liberado.

26. Responda Verdadero o Falso y justifique: “El programa cliente que se conecta a un area de memoria compartida debe especificar (en la conexión) el tamaño de la misma”.

Falso. Razon por la cual la API de la librería myipc tiene dos llamadas distintas: una para crear la memoria (proceso server) y otra para conectarse a la memoria compartida (proceso cliente) y en este último caso, no se requiere indicar el tamaño de la memoria compartida.

27. Responda Verdadero o Falso y justifique: “Una de las principales ventajas de la memoria compartida es que nos libera de la preocupación de la compartición de datos, puesto que implementa automáticamente mecanismos de exclusión mutua”.

Falso. La memoria compartida sólo permite el acceso controlado (a través de los permisos que indique el proceso owner de dicha memoria compartida) a un area de memoria comun, mantenida como un recurso centralizado en el kernel del SO, pero no implementa ningún tipo de exclusión mutua sobre la misma.

28. ¿Qué es lo que hace esta sentencia C: `for(i=0;i<10;i++,p++) *p *= 1.1;` ?

Si p es un puntero a un arreglo de números de tipo double o float, este código lo que hace es incrementar en un 10 % los primeros 10 valores contenidos en dicho arreglo. Luego de esta instrucción, p estará apuntado al elemento 11; si el arreglo solo tiene 10 elementos, p estará apuntando más allá de los límites del arreglo.

29. ¿Qué relación hay entre el estado "zombie" de un proceso y la llamada al sistema waitpid() ?

Un proceso *h* queda en estado "zombie" (una especie de estado suspendido) cuando *h* termina y su proceso padre *p* aun no lo estaba esperando (a través de la ejecución de la llamada al sistema waitpid(), por ejemeplo); cuando por fin el proceso *p* llama a waitpid(), el proceso *h* deja el estado "zombie" y termina [Tanenbaum, Sistemas Operativos Modernos, 3ra. Ed, Pag 744].

30. ¿Cuántas llamadas pendientes de alarma puede tener un proceso?

Una sola [Tanenbaum, Sistemas Operativos Modernos, 3ra. Ed, Pag 745].

31. ¿Qué sucede si se solapan dos llamadas de alarm() en un proceso? Por ejemplo, supongamos que un proceso hace una llamada a alarm() con 10 segundos y 3 segundos más tarde, se hace otra llamada a alarm() con 30 segundos.

Como todo proceso no puede tener mas de una alarma pendiente, lo que sucede es que la segunda llamada a alarm() cancela a la primer llamada y provocará que se genere una señal de alarma 20 segundos después de la segunda llamada a alarm() [Tanenbaum, Sistemas Operativos Modernos, 3ra. Ed, Pag 745].

En cuanto a Concurrencia, Interbloqueo e Inanición:

1. Supongamos que dos procesos (A,B) que se ejecutan concurrentemente, requieren los recursos R1 y R2. Respnda Verdadero o Falso:
   a. Siempre -sin excepción- se producirá el interbloqueo

Falso, pueden existir varios caminos seguros en las trayectorias de los proceso A y B.

   b. Es factible que exista una o más trayectorias (en la ejecución concurrente de A y B) que eviten el interbloqueo

Verdadero.

c. El interbloqueo se evita cuando los procesos entran en la "región fatal"

Falso, el interbloqueo se produce cuando se ingresa en una región fatal.

d. Evitar el interbloqueo implica encontrar "un camino seguro"

Verdadero, se evita el interbloqueo no entrado a una región fatal, manteniendo al sistema es un estado seguro.

e. "Un camino seguro" es un camino que no pasa por una "región fatal"

Verdadero.

2. Responda Verdadero o Falso: El interbloqueo siempre ocurre por competencia de recursos.

Falso. Tambien puede ocurrir por comunicación entre procesos.

3. ¿Cuáles son las condiciones necesarias (aunque no suficientes) para la ocurrencia del interbloqueo?

Exclusión mutua. Retención y Espera (un proceso tiene recursos asignados mientras espera por otros). Sin Expropiación (no se le puede forzar la expropiación de un recurso a un proceso que lo posee).

4. ¿Para qué sirve un "Grafo de Asignación de Recursos"?

Permite graficar interbloqueos.

5. ¿Qué forma tiene un "Grafo de Asignación de Recursos" cuando estamos en presencia de una "espera circular"?

Son grafos cerrados, circulares, cíclicos, en donde todos los procesos tienen asignados recursos y al menos uno de ellos pretende la asignación de un recurso tomado por otro proceso, no habiendo más unidades disponibles de ese recurso para satisfacer su demanda.

6. ¿Cuáles son las estrategias utilizadas para el tratamiento del problema del interbloqueo? ¿Qué tratan de lograr estas estrategias?

Las estrategias tratan de encontrar un camino seguro en la asignación de recursos a procesos, evitando, de esta forma, el interbloqueo.

Las estrategias son: Prevención, Predicción, Detección.

7. Teniendo en cuenta el grado de concurrencia entre los procesos que permiten las estrategias para el tratamiento del interbloqueo ¿Cuál es la estrategia más conservadora, la menos conservadora y la más moderada?

La más conservadora es la Prevención, la más moderada es la Predicción y la menos conservadora es la Detección.

8. Supongamos que no puedo determinar -por anticipado- los recursos que requerirá cada proceso, en este caso, ¿Puedo utilizar la estrategia de "Predicción del Interbloqueo"?

Si no se puede determinar por anticipado todos los recursos que requerirá cada proceso entonces no puede aplicarse Prevención ni tampoco puede aplicarse Predicción a través del Algoritmo del Banquero (uso de matrices de necesidad y asignación de recursos, permite anticipar la posibilidad de interbloqueo).

En 1965 Dijstra ideó el algoritmo del Banquero, si al mismo lo aplicamos sobre un solo recurso –y solo en este caso- podemos decir que podría aplicarse aunque no supiéramos de antemano el requerimiento de este recurso por parte de cada proceso[1]. Cuando generalizamos el Algoritmo del Banquero para manejar varios recursos entre varios procesos, se requiere conocer por anticipado (antes de su ejecución) la cantidad máxima requerida de cada recurso por parte de cada proceso.[2] Debido a esta limitación, este algoritmo en la práctica es muy poco utilizado para evitar interbloqueos.

9. ¿Cómo funciona el "Algoritmo del Banquero" y en qué tipo de estrategia se usa?

Usa la estrategia de Predicción. El sistema posee un estado (la asignación actual de recursos a

[1] "Sistemas Operativos Modernos", 3ra.Ed. en Español, Tanenbaum, Pag.451-452.
[2] Idem anterior, Pag. 452-454.

procesos), el cual puede ser un estado seguro (al menos puede otorgarse una petición de asignación de recurso sin ocasionar un interbloqueo) o no seguro. El algoritmo requiere 2 vectores uno indicando los recursos existentes y la cantidad de cada uno y otro vector indicando la cantidad de disponible de cada recurso, atento a la asignación actual de recursos. La asignación actual de recursos se implementa en una matriz de recursos y procesos (matriz de asignación); por otro lado, hay otra matriz identica que guarda la necesidad de recurso por parte de cada proceso (matriz de neccesidad); el algoritmo hace la diferencia entre ambas matrices y teniendo en cuenta la disponibilidad de recurso, cuando un proceso solicita un conjunto de recursos, suponiendo que éstos se conceden, se actualizan las matrices y el sistema puede determinar si ello configura un estado seguro.

10. ¿Cuál es la estrategia que requiere de un algoritmo que se ejecute periódicamente para detectar la "condición de espera circular"?

Detección.