Preguntas de parciales:

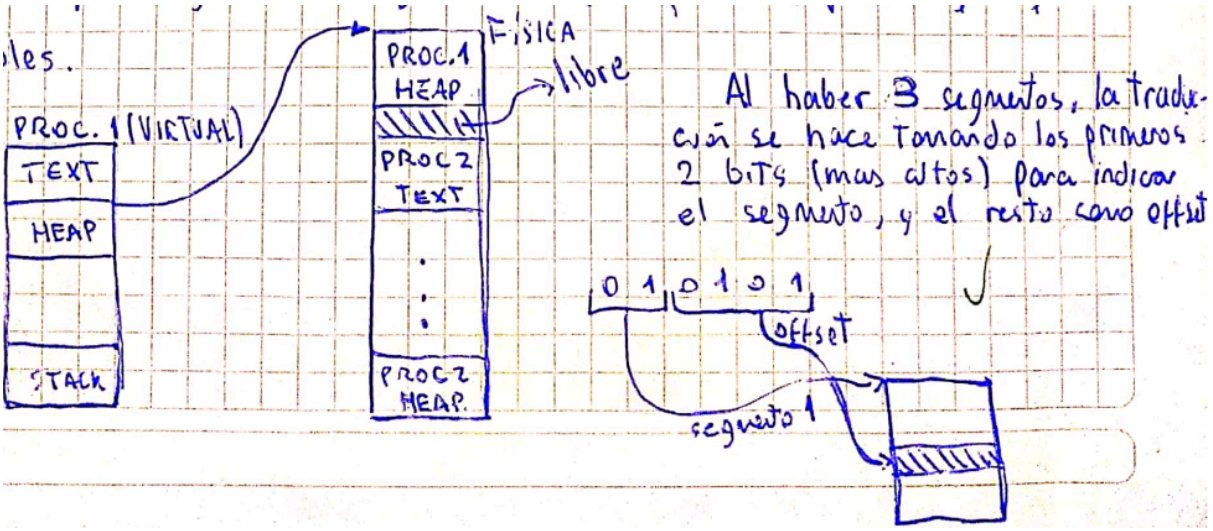
MEMORIA:

“¿Qué es la memoria virtual? ¿Qué mecanismos conoce?: describa los tres que a ud. le parezcan más relevantes.”

La memoria virtual es una técnica utilizada por los sistemas operativos para la abstracción y la gestión de la memoria física. Permite que los procesos tengan esa “ilusión” de disponer de toda la memoria del sistema, cuando solo utilizan una parte de ella. Facilita la ***protección*** entre procesos, ya que cada proceso solo puede ver y utilizar la memoria que le corresponde sin interferir en la de los demás procesos, ***transparencia***, la virtualización debe ser invisible para el programa en ejecución, y eficiente, la virtualización debe ser ***eficiente*** en términos de tiempo y espacio. Algunos mecanismos son:

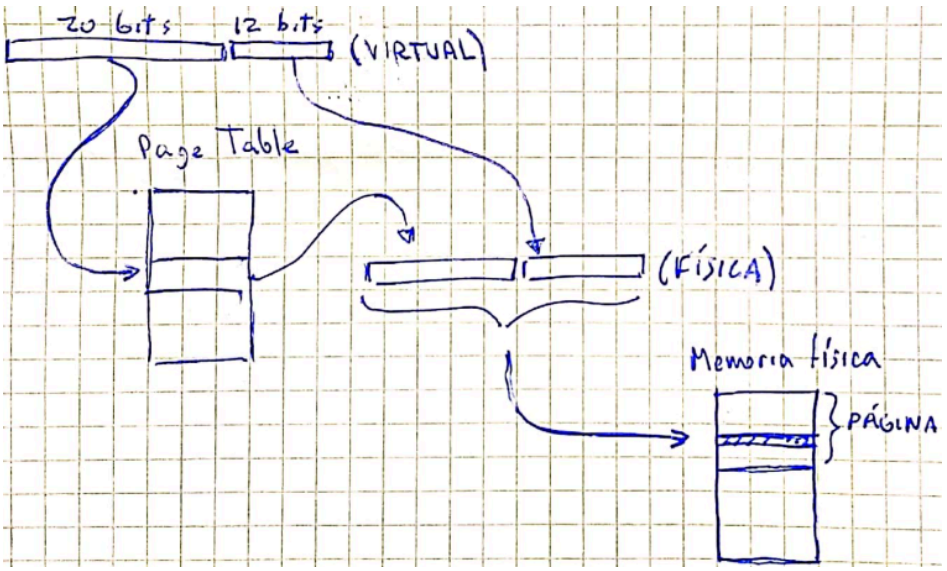
-Base and bound: Se asigna en memoria un bloque contiguo. Este bloque comienza en un índice base y termina en el índice base-bound. Se tienen dos registros en los cuales se guardan estos índices. Cualquier acceso fuera de estos índices produce un error.

-Segmentación: Divide la memoria en segmentos, utilizando base and bound, por lo que cada segmento tiene su inicio y longitud. Se tiene una tabla de segmentos donde contiene la base, el límite y los permisos de cada segmento. En los bits más altos se encuentra el índice de la tabla de segmentos, el resto es el offset. Esto produce fragmentación externa ya que pueden quedar bloques sin utilizar debido a su tamaño.



-Paginación: Consiste en dividir la memoria según un tamaño fijo llamados páginas, o page frames. Por cada proceso hay una tabla de páginas cuya entrada son direcciónes virtuales las cuales se pueden traducir a direcciones físicas, estas direcciones suelen ser de 32 bits. Mediante las tablas de páginas podemos realizar estas traducciones ya que los primeros 20 bits se utilizan como índice de la page table, el cual dentro de este índice podemos encontrar la dir virtual gracias a los 12 bits restantes de offset.

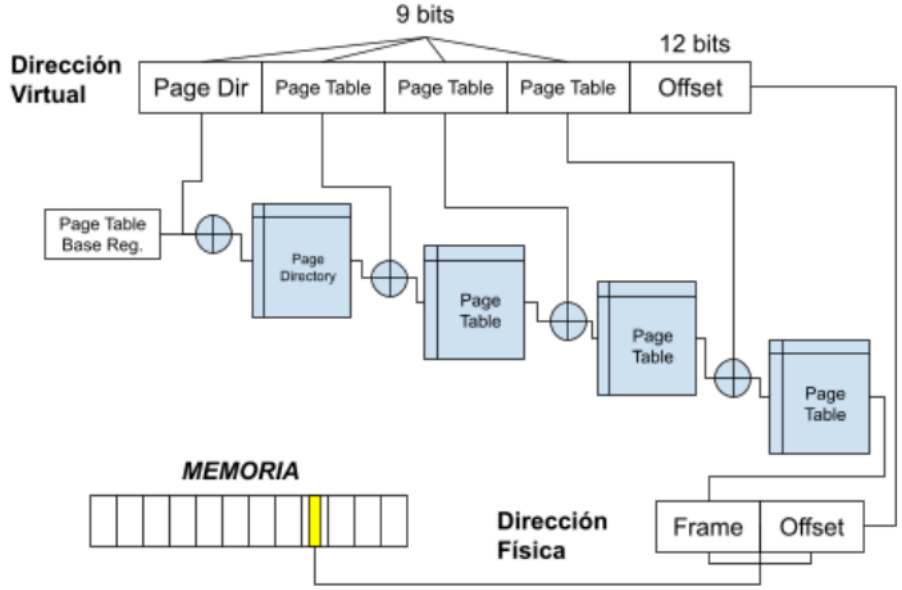
También existen las de paginación multinivel, (con el page directory)



“Explicar el mecanismo de address translation memoria virtual paginada de tres niveles de indirección de 32 bits. Indique la cantidad de direcciones de memoria que provee, una virtual address: [7 bits, 7 bits, 6 bits, 12 bits] con tablas de registros de 4 bytes.”

El address translation es muy simple, el mecanismo es muy similar al de 2 niveles solo que le agregamos como un nivel de páginas más. Se utilizan los tres primeros fragmentos como índices jerárquicos para buscar en las 3 tablas de páginas. Por último se utilizan los 12 bits restantes como el offset para encontrar la dirección física real.

Se tienen 2^7 entradas en el primer nivel, 2^7 en el segundo nivel, 2^6 en el tercer nivel y por último 2^{12} del offset (el desplazamiento dentro de la página). Por lo que nos da un total de 2^{32} de espacio de direcciones virtuales.

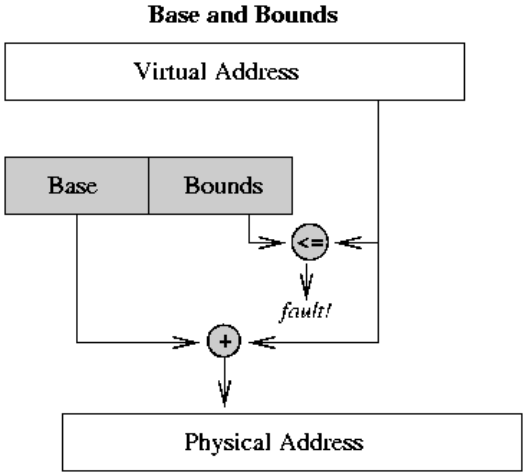


“¿Cuál es la cantidad de Kbytes que se pueden almacenar en un esquema de memoria virtual de 48 bits con 4 niveles de indirección, en la cual una dirección de memoria se describe como sigue: 9 bits page dir., 9 bits para cada page table y 12 bits para el offset? Explicar.”

En el esquema propuesto se analiza de la siguiente manera: tenemos 2^{48} bytes y asumimos que cada dir es un byte. y 1 Kbyte es 2^{10} por lo que podríamos almacenar $2^{48}/2^{10}$ Kbytes.

“Describa cómo fue variando la estructura del address space respecto la memoria física en: base y bound, tabla de registros y paginación. Explique con diagramas.”

La estructura del adress space respecto a la memoria física fue cambiando significativamente. Primero se utilizó base and bound el cual a cada proceso se le asigna una porción de memoria contigua comenzando en base y terminando en base+bound.



Luego está la tabla de registros o de segmentos, que se basaba en dividir la memoria en segmentos con base and bound. No es necesario disponer de memoria contigua para cada proceso, aprovechando los distintos segmentos. La tabla contiene base, bound y permisos del archivo. (VER IMAGEN ARRIBA).

Por último, tenemos la paginación, en la cual consiste dividir la memoria en ciertos bloques de tamaño fijo llamados páginas. En esta hay direcciones virtuales, que gracias a la MMU se pueden convertir en direcciones físicas reales. Para optimizar esto , se puede utilizar una TLB que guarda las instrucciones más recientes utilizadas. Además, si tenemos una memoria virtual muy grande se puede usar una paginación multinivel. (VER DIBUJO ARRIBA).

“La siguiente es una representación del espacio de direcciones virtual de un proceso. La tabla representa la totalidad de las secciones del espacio de direcciones virtual de 32 bits.”

Sección	Tamaño en páginas de 4k
Trampoline	1
Stack	512
Sin asignar	*?*
Heap	1024
Data	20
Text	768

“-Recordando que la tabla abarca todo el espacio de direcciones virtual. ¿Cuánto ocupa en cantidad de páginas del espacio sin asignar (en regiones)?”

Páginas totales= 1+512+1024+20+758=2315. En un sistema de direcciones de 32 bits->

$2^{32}/4*4096(2^{12})=2^{20}$ por lo que son la cant de páginas totales. 2^{20}-2315=1046261 páginas a asignar.

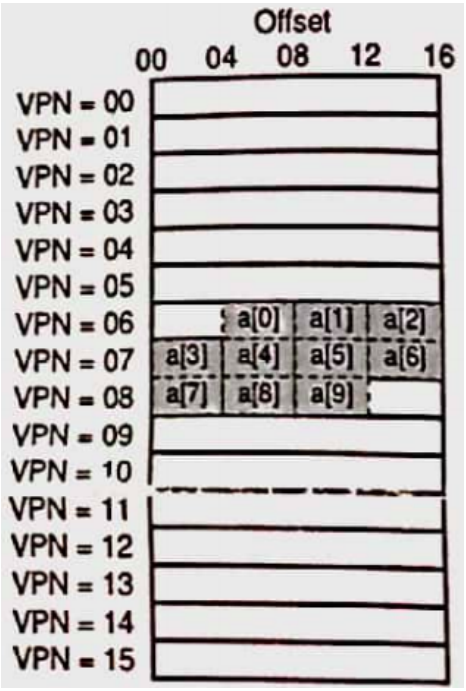
“Cuántas frames de memoria física son necesarias para mapear este espacio (no considere en este punto las tablas de páginas, solo las frames asignadas).”

Se necesita mapear 2315 según los frames asignados.

“Cuántas tablas de páginas en un esquema x86 de 2 niveles de paginado son necesarias para mapear este espacio de direcciones”

En un esquema x86 de 2 niveles de paginado se deben utilizar 2315/1024=3 tablas de páginas.

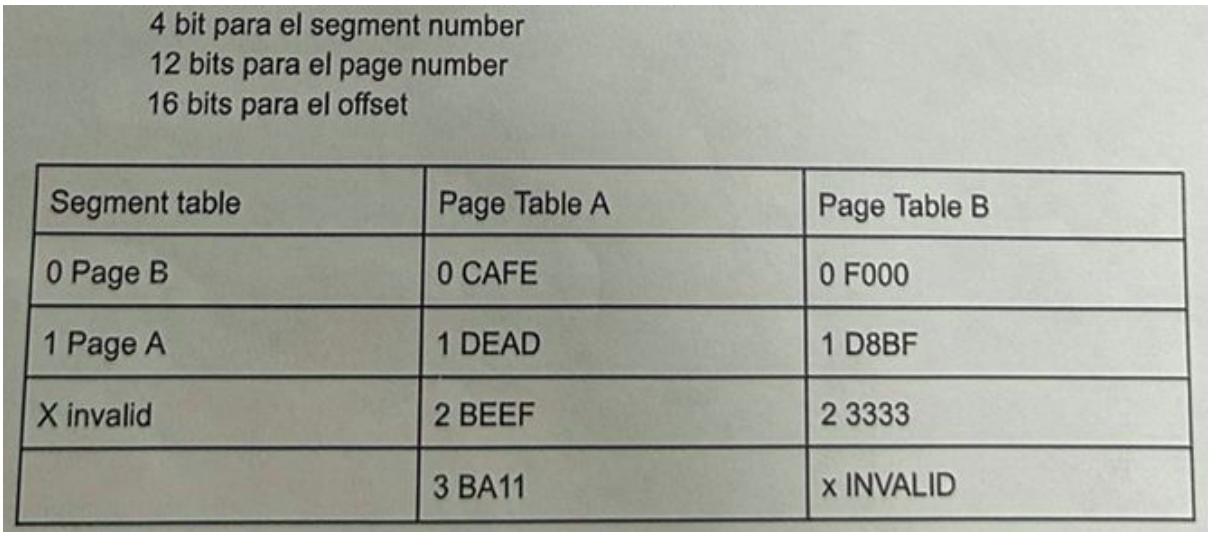
“Dado el siguiente esquema, explique cómo se realizan las traducciones recorriendo el arreglo en un modelo de memoria virtual con TLB y paginación de dos niveles. En el mismo esquema, especifique cuántos miss, hit, accesos a memoria y traducciones hay.”



Determine si son verdaderos o falsos:

- Hay 3 hits y 7 miss en la TLB. **FALSO**
- Hay 10 accesos a memoria. **FALSO**
- En las traducciones hay 7 hits y 3 miss en la TLB. **VERDADERO**
- Hay 3 traducciones completas de VA a PA. **VERDADERO**
- Hay 3 accesos a memoria en total. **FALSO**
- Hay 10 traducciones completas de VA a PA. **FALSO**

“Suponga una dirección virtual con las siguientes características:. Traducir las siguientes direcciones virtuales a físicas: 00000000, 20022002, 10022002, 00015555.”



Vamos a pasar a binario las direcciones virtuales y a separarlas en segment number, page number y offset.

Dirección Virtual	Binario	Segment Number	Page Number	Offset
00000000	0000 0000 0000 0000 0000 0000	0000	0000 0000 0000	0000 0000 0000 0000
20022002	0010 0000 0000 0010 0010 0000 0010	0010	0000 0000 0000	0010 0010 0000 0010
10022002	0001 0000 0000 0010 0010 0000 0010	0001	0000 0000 0000	0010 0010 0000 0010
00015555	0000 0000 0000 0001 0101 0101 0101	0000	0000 0000 0000	0001 0101 0101 0101

Pasado a número decimal, tenemos:

Dirección Virtual	Segment Number	Page Number	Offset
00000000	0	0	0
20022002	2	0	2002
10022002	1	2	2002
00015555	0	1	5555

Entonces, mappeados: 00000000 → CAFE → CAFE 0000
20022002 → Invalid 10022002 → BEEF → BEEF 2002
00015555 → DEAD → DEAD 5555

“Sea un disco que posee 2049 bloques de 4 KB y un sistema operativo cuyos inodos son de 256 bytes. Definir un sistema de archivos FFS. Explique las decisiones tomadas. Desarrolle.”

C/bloque contiene 4kb=4096 bytes/ 256bytes=16 inodos por bloque

2049/16=128.063=129 bloques de inodo

1 superbloque, 1 bitmap de datos, 1 bitmap de inodos, 129 de bloque de inodos. Por lo que bloque de datos tendrá: 2049-1-1-1-129=1917

SCHEDULING:

“Dado el siguiente scheduler, implemente la función `elegir` para conseguir una política Round Robin (puede agregar las variables globales que crea necesarias):”

```
void switch(struct p* proceso); // Asuma ya implementada
struct p {
    int status; // RUNNING, RUNNABLE, BLOCKED
};
struct p p[64]; // Tabla de procesos, máximo de 64
struct p* elegir() {
    // TODO: implementar aquí
}

void scheduler() {
    for (;;) {
        struct p* candidato = elegir();
        if (candidato == NULL) {
            idle();
        } else {
            candidato->status = RUNNING;
            switch(candidato);
        }
    }
}
```

Asuma que switch ya existe y recibe un struct p* con estado RUNNING y hace un cambio de contexto a ese proceso. Cuando el proceso se suspende, devuelve el control al scheduler (en este momento el proceso tendrá estado RUNNABLE o BLOCKED). Asuma que los procesos no terminan.

```
int ultimo_proceso = -1;
struct p* elegir() {
    int i;
```

```
   int total_procesos = 64;
   // Comenzar a buscar desde el siguiente proceso después del último seleccionado
   for (i = 1; i <= total_procesos; i++) {
      int indice = (ultimo_proceso + i) % total_procesos; // Circular
      if (procesos[indice].status == RUNNABLE) {
         ultimo_proceso = indice; // Actualizamos el último proceso elegido
         return &procesos[indice];
      }
   }
    return NULL;  // Si no hay ningún proceso RUNNABLE, retornar NULL
}
```

```c
int ultimo_proceso = -1;

struct p* elegir() {
    int i;
    int total_procesos = 64;

    // Comenzar a buscar desde el siguiente proceso después del último seleccionado
    for (i = 1; i <= total_procesos; i++) {
        int indice = (ultimo_proceso + i) % total_procesos; // Circular
        if (procesos[indice].status == RUNNABLE) {
            ultimo_proceso = indice; // Actualizamos el último proceso elegido
            return &procesos[indice];
        }
    }

    // Si no hay ningún proceso RUNNABLE, retornar NULL
    return NULL;
}
```

“¿Qué es un context switch? En un context switch, cuáles de las siguientes cosas no deben/deben ser guardadas y por qué?: a. registros de propósito general, b. translation lookaside buffer, c. program counter, d. Page Directory Entry, e. PCB entry, e. ninguna.”

Un context switch o un cambio de contexto es un proceso que consta de justamente cambiar de un proceso a otro. Para esto se guarda el estado actual del proceso saliente, para que luego en un futuro se pueda volver a reanudar cómo se encontraba previamente, y se carga el estado del proceso entrante, volviendo a cargar lo guardado por este proceso previamente, permitiendo que cada uno se vuelva a poder ejecutar como si nada hubiera pasado. Se guarda el contexto por eso context switch.

Los registros de propósito general no se deben guardar ya que justamente son de propósito general para todos los procesos en sí. El translation lookaside buffer o TLB no se guarda ya que es parte de la MMU. El PC o program counter es necesario ya que indica la próxima instrucción a ejecutar. El page directory entry también es necesario ya que nos ayuda a

encontrar los archivos necesarios para el programa (cada proceso tiene su propio espacio de direcciones). PCB entry se debe guardar para restaurar el proceso.

“Explique la política MLFQ detalladamente. Sea 1 proceso, cuyo tiempo de ejecución total es de 40 ms, el time slice por cola es de 2 ms/c pero el mismo se incrementa en 5 ms por cola. Cuantas veces se interrumpe y en qué cola termina su ejecución”

La política MLFQ consiste en tener un arreglo de colas, en este cada posición del arreglo tiene una cierta prioridad, que suele ir de 1 a 8, y cada una de estas prioridades son prioridades de ejecución. El scheduler tiene distintas opciones, si un nivel o posición del arreglo, tiene un solo elemento, se ejecuta ese proceso. Pero si ese nivel está compuesto por múltiples elementos, se ejecutan en Round Robin hasta que terminen, luego pudiendo pasar al siguiente nivel donde ejecuta las tareas o procesos de la misma forma. MLFQ tiene ciertas reglas que permiten priorizar las tareas enfocadas a las E/S reduciendo el response time, asegurando que las tareas largas también se ejecuten:

-Si A tiene mayor prioridad que B, A se ejecuta y B no.

-Si A y B tienen igual prioridad, se ejecutan en RR hasta que finalicen.

-Cada tarea nueva que llega al sistema se coloca en la cola de mayor prioridad o mejor nivel.

-Cada cierto tiempo S las tareas se mueven todas a la primera cola de prioridad.

-Cada vez que una tarea consuma su time slice completo en un nivel dado, su prioridad baja, es decir, baja un nivel en la cola de prioridad.

Para el caso del ejemplo ocurre lo siguiente (ver hoja)

“Explique cuál es la idea central de MLFQ y porque es mejor que otras políticas de scheduling. Justifique su respuesta”

La idea central de MLFQ consiste en tener distintos niveles de prioridad, es decir distintas colas con distintas prioridades que generalmente llegan hasta 8. Logra evitar el starvation pero no funciona correctamente con tareas de tiempo variable. Los procesos se van moviendo entre las colas de prioridades según sus tiempos de ejecución. Esto se dicta por ciertas reglas como: si A tiene mayor prioridad que B, A se ejecuta y B no, cada proceso nuevo que llega al scheduler se coloca en la cola de mayor prioridad. Si hay dos o más procesos con la misma prioridad, es decir la cola tiene más de un elemento, se ejecutan en RR hasta que terminen. Si un proceso termina consumiendo todo su timeslice en cierto nivel dado, su prioridad baja. Cada cierto tiempo S todos los procesos se restauran el la cola de mayor prioridad (esto ayuda al starvation). También mejora el Turnaround time, al contrario de SJF o STCF. Por ejemplo:

- Cola 0: Quantum de 8ms, procesos nuevos y procesos I/O-bound.
- Cola 1: Quantum de 16ms, procesos que no se completaron en la cola 0.
- Cola 2: Quantum de 32ms, procesos que no se completaron en la cola 1.
- Cola 3: Quantum de 64ms, procesos que no se completaron en la cola 2.

“Cuál es la mejora respecto a MLFQ que introduce el scheduler de linux”

Linux introduce otro scheduler llamado CFS o completely fair scheduler. Este elige los procesos con menor vruntime para que se ejecute luego de un context switch. Este contiene algunos elementos para mejorar su implementación. Uno es la implementación de un red-black-tree, el cual optimiza para obtener el proceso de menor vruntime ya que es ir todo

hacia la izquierda del árbol, así como las inserciones de elementos es mucho más eficiente, sin olvidar mencionar que siempre mantiene los procesos ordenados. También, trata de ser equitativo tratando de dar un tiempo justo para todos los procesos, es decir, tratar de mantener los vruntime de los procesos iguales, eligiendo siempre el proceso que quedó "más atrás" o con menos vruntime.

"Dado el siguiente scheduler, implemente la función `elegir` para conseguir una política que seleccione el proceso que menos veces se haya seleccionado hasta el momento (puede modificar `struct p` agregando los campos que crea necesarios):"

```
void switch(struct p* proceso); // Asuma ya implementada
struct p {
    int status; // RUNNING, RUNNABLE, BLOCKED
};
struct p p[64]; // Tabla de procesos, máximo de 64
struct p* elegir() {
    // TODO: implementar aquí
}
void scheduler() {
    for (;;) {
        struct p* candidato = elegir();
        if (candidato == NULL) {
            idle();
        } else {
            candidato->status = RUNNING;
            switch(candidato);
        }
    }
}
```

¿Qué diferencias encuentra entre el scheduler del punto anterior y el Completely Fair Scheduler de Linux?

```
struct p {
    int status; // RUNNING, RUNNABLE, BLOCKED
    int veces_elegido;
};
struct p* elegir() {
        int min_veces_elegido = INT_MAX;
    int indice_min_veces_elegido = -1;
    for (i = 0; i <64; i++) {
```

```
                if (procesos[i].status == RUNNABLE || procesos[i].status == RUNNING &&
procesos[i].veces_elegido < min_veces_elegido) {
        min_veces_elegido = procesos[i].veces_elegido;
        indice_min_veces_elegido = i;
    }
  }
  if (indice_min_veces_elegido != -1) {
    procesos[indice_min_veces_elegido].veces_elegido++;
    return &procesos[indice_min_veces_elegido];
  }
  return NULL;
}
```

```c
struct p {
    int status; // RUNNING, RUNNABLE, BLOCKED
    int veces_elegido;
};

struct p* elegir() {
    int total_procesos = 64;
    int min_veces_elegido = INT_MAX;
    int indice_min_veces_elegido = -1;

    for (i = 0; i < total_procesos; i++) {
        if (procesos[i].status == RUNNABLE || procesos[i].status == RUNNING && procesos[i].veces_elegido < min_veces_elegido) {
            min_veces_elegido = procesos[i].veces_elegido;
            indice_min_veces_elegido = i;
        }
    }

    if (indice_min_veces_elegido != -1) {
        procesos[indice_min_veces_elegido].veces_elegido++;
        return &procesos[indice_min_veces_elegido];
    }

    return NULL;
}
```

A diferencia del CFS de Linux, este selecciona el proceso con menor vruntime para correr luego de un context switch. Utiliza un red-black-tree como estructura auxiliar que optimiza la ejecución y selección del proceso con menor vruntime. Además de utilizar un valor nice (que va de -20 a 19), que permite ajustar la prioridad a los procesos. A diferencia, CFS toma en cuenta el tiempo real de ejecución de los procesos según su prioridad, en lugar de estimarlos.

PROCESOS:

“¿Qué es el stack? Explique el mecanismo de funcionamiento del stack para x86 de la siguiente función: int read(void *buff, size_t len, size_t num, int fd);. Cómo se pasan los parámetros, dirección de retorno, etc.”

El stack es una región del espacio de direcciones que funciona similar a una pila. Se utiliza para almacenar variables locales, argumentos de funciones, direcciones de retorno, etc. Mediante las funciones de push y pop se pueden agregar o quitar elementos del stack, en su defecto también se podría hacer modificando el stack pointer que apunta al stack.

La función mencionada por la consigna funciona de la siguiente manera: se almacenan los registros caller saved, se almacenan los argumentos de la función en orden inverso, se llama a la función, se obtienen los argumentos del stack, dentro de la función se almacenan los registros caller saved en el stack, se realiza la operación de la función, se almacena el valor de retorno en el eax y termina la función, por último se restauran los registros caller saved del stack.

“¿Qué es el Address Space? ¿Qué partes tiene? ¿Para qué sirve? Describa el/los mecanismos para crear procesos en unix, sus syscalls, ejemplifique. Adicional: wait.”

El address space o espacio de direcciones, fue un concepto que se introdujo luego de la virtualización de la memoria para poder operar entre direcciones físicas y la abstracción del programa. En este se tiene un espacio de direcciones, que son las direcciones virtuales vistas por el programa, por lo que el address space tiene todo el estado de la memoria de un programa en ejecución. Está compuesto por text (código del programa), data (variables globales), heap (memoria dinámica) y stack (variables locales, trace a llamadas, etc). Sirve para que un proceso utilice sólo la memoria de este y no afecte a la de los demás procesos. Si se desea utilizar memoria compartida se puede tener una memoria compartida dada por los permisos de los procesos.

Los mecanismos para crear procesos en unix el sistema operativo se encarga de crear el espacio de direcciones, copiando las instrucciones del programa en text, las var globales y las mapea en data, reserva espacio para el heap y stack que a su vez setea el stack pointer. Para crear un proceso se utiliza la syscall fork(), que crea un proceso hijo idéntico al padre copiando su espacio de direcciones. Luego está exec() que reemplaza el programa que se está ejecutando en el proceso actual por otro, setea nuevo heap, stack y cambia el text.

“¿Cuáles son los requerimientos mínimos de hardware para poder construir un kernel?”

Los requerimientos mínimos son:

- Modo dual: permitir la ejecución tanto en modo usuario como en modo kernel. Permitiendo que hayan instrucciones privilegiadas que sean ejecutadas sólo por el modo kernel y bloquearlas al usuario.

-Instrucciones Privilegiadas: que hayan ciertas instrucciones aptas para el kernel y bloqueadas para el usuario. Estas instrucciones privilegiadas pueden actuar en secciones críticas y a nivel hardware.

-Timer interrupts: Soportar temporizadores a nivel hardware, generando interrupciones periódicas para que el kernel pueda recuperar el control del sistema.

-Protección de memoria: soporte de hardware para proteger y separar áreas de memoria.

“Escriba un programa en C que simule el funcionamiento de una shell primitiva. ¿Qué debería agregarse para poder encadenar dos comandos por salidas y entradas estándar?”

```
#define BUF 256

#define CMD 32

void tokenize(char *cmd[16], char *buffer) {
    char *token;
    size_t i = 0;
```

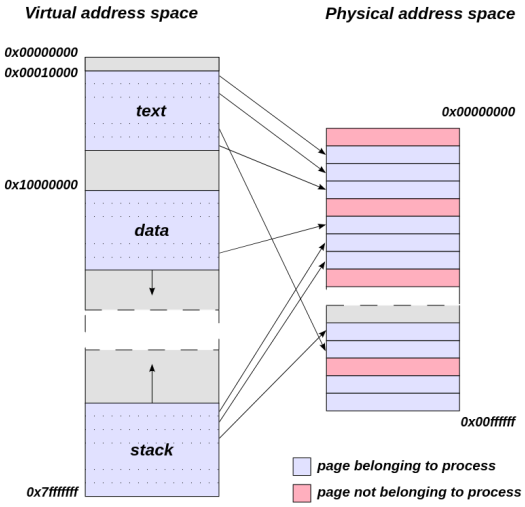
```
    token = strtok(buffer, " \t\n");
    while (token != NULL && i < CMD-1) {
        cmd[i++] = token;
        token = strtok(NULL, " \t\n");
    }
    cmd[i] = NULL;
}
int shell(void) {
    char buffer[BUF];
    char *cmd[CMD];
    while (1) {
        if (getcwd(buffer, BUF) == NULL) return 1;
        printf("%s%s ", buffer, "$");
        if (fgets(buffer, BUF, stdin) == NULL) break;
        if (!strcmp(buffer, "exit\n")) break;

        tokenize(cmd, buffer);
        if (!strcmp (cmd[0], "cd")) {
            if (chdir(cmd[1]) == -1) perror("chdir");
            continue;
        }
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            return 1;
        }
        if (pid == 0) {
            if (execvp(cmd[0], cmd) == -1) {
                perror("execvp");
                return 1;
            }
        } else {
            wait(NULL);
        }
    }
    free(cmd);
    return 0;
}
```

Para encadenar dos comandos por salidas y entradas estándar, se puede utilizar pipes al encontrarse con símbolos de redirección, haciendo por ejemplo que la salida de un comando sea la entrada del siguiente.

“¿Qué es el address space, cuál es su estructura y que función cumple? Explique detalladamente y con gráficos”

El **address space** (espacio de direcciones) es el conjunto de direcciones de memoria que un proceso puede usar. Cada proceso en un sistema operativo moderno tiene su propio espacio de direcciones, lo que le permite ejecutarse de manera aislada y segura. El espacio de direcciones de un proceso está dividido en segmentos. Cada segmento tiene un propósito específico. Algunos son: **Text segment**: contiene el código del programa. **Data segment**: contiene variables globales. **Heap**: zona de memoria dinámica. **Stack**: memoria para funciones y variables locales. Define qué regiones de memoria puede usar el proceso y para qué.



Describa qué es un proceso: qué abstrae, cómo lo hace, cuál es su estructura. Además, explique el mecanismo por el cual el proceso cree tener la memoria completa de la máquina cuando en realidad solo tiene lo necesario para su funcionamiento.

Un proceso abstrae la ejecución de un programa, es un programa en ejecución. Incluye el estado del programa, recursos y contexto de ejecución. Abstrae el uso exclusivo de la CPU, la memoria y dispositivos, permitiendo que varios programas se ejecuten en simultáneo y sin interferencias. El sistema operativo lo gestiona mediante planificación, aislamiento de memoria y conmutación de contexto. Su estructura incluye: código y datos del programa, stack (llamadas y variables locales), heap (memoria dinámica), contexto de ejecución (registros, PC, estado), tabla de archivos y PCB (estructura de control del proceso)

Un proceso cree tener la memoria completa de la máquina gracias a la abstracción de la memoria o memoria virtual. La memoria virtual permite tener una abstracción entre la memoria física real y la memoria virtual, a través de direcciones virtuales que se pueden traducir a direcciones físicas. Esto permite que un proceso vea solo su espacio de direcciones de memoria el cuál le hace creer que toda esa memoria es solo para el proceso

que se está ejecutando. Ve una memoria continua e independiente cuando físicamente comparte los recursos con los demás procesos.

“¿Cuál o cuáles mecanismos utiliza el kernel para garantizar el aislamiento entre procesos? Estos mecanismos están relacionados con el hardware. Explique por qué deben existir y dónde se soporta su funcionamiento.”

El kernel, para garantizar el aislamiento entre procesos, utiliza la memoria virtual y los diferentes modos de ejecución del procesador. Esto permite que gracias a la memoria virtual, la utilización de memoria entre procesos sea aislada y segura permitiendo tener a cada proceso su propio espacio de direcciones. La memoria virtual necesita de la MMU para hacer la traducción de direcciones virtuales a físicas. Los distintos modos sirven para que las instrucciones privilegiadas no se ejecuten siempre que se desee, sino que se ejecutan cuando el procesador está en modo kernel para que se ejecuten de manera controlada. Esto no permite que los procesos accedan a recursos críticos del sistema. Se requiere que el CPU pueda soportar los dos modos, y las instrucciones privilegiadas y no privilegiadas.

CONCURRENCIA

Atomicidad: En el contexto de los threads (hilos), "atómico" significa que una operación se completa sin interrupciones, es decir, se realiza como una unidad indivisible. Si una operación es atómica, no puede ser interrumpida a mitad de su ejecución por otro hilo.

“¿Qué es un deadlock? Describa por lo menos tres casos diferentes en el que puede suceder.”

Un deadlock es un tipo de situación que se da cuando dos o más locks se bloquean mutuamente entre sí. Esto sucede cuando los threads quieren utilizar recursos que están siendo utilizados por otros threads, bloqueandolos para su uso. Un ejemplo se da cuando el proceso a tiene recurso 1 y necesita recurso 2 pero B tiene recurso 2 y necesita recurso 1. Otro caso es cuando proceso X toma un scanner y quiere utilizar la impresora, luego proceso Y tiene la impresora pero quiere usar el scanner. También T1 bloquea tabla clientes y quiere acceder a tabla pedidos, T2 bloquea tabla pedidos y quiere acceder a tabla clientes.

“Describa qué es un thread y use su API para crear un programa que use 5 threads para incrementar una variable compartida por todos en 7 unidades/thread hasta llegar a 1000.”

Un thread es un hilo de ejecución atómica dentro de un proceso. Permite la ejecución concurrente de tareas dentro de un proceso, compartiendo el espacio de memoria y los recursos del proceso principal permitiendo trabajar con datos compartidos de manera eficiente. Se utiliza para optimizar tareas concurrentes o paralelas, es mejor en varios CPUs.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void *increment(void *_value) {
    int *value = (int *)_value;
    while (*value < 1000) {
        pthread_mutex_lock(&mutex);
```

```
        if (*value < 1000) {
            *value += 7;
        }
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
int main(void) {
    int value = 0;
    pthread_t threads[5];
    for (int i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, increment, &value);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Value: %d\n", value);
}
```

“¿Es correcta la siguiente implementación de este programa que actualiza el saldo de una cuenta? Explique qué problema tiene y cómo solucionarlo:”

```
void add(int incremento) {
    int v;
    v = cuenta_leer_saldo();
    v += incremento;
    cuenta_escribir_saldo(v);
}
```

En el contexto de threads, no es correcta ya que puede haber un race condition. Genera un problema de concurrencia ya que si dos threads se ejecutan al mismo tiempo puede haber una mezcla entre las ejecuciones. Se puede agregar un mecanismo de bloqueo para que haya exclusión mutua. En este caso, ambos threads leerían el saldo, ambos lo incrementarían y el resultado va a ser pisado por el último que escriba. En caso de querer arreglarlo quedaría:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void add(int incremento) {
    pthread_mutex_lock(&mutex);
    int v;
    v = cuenta_leer_saldo();
    v += incremento;
```

```
    cuenta_escribir_saldo(v);
    pthread_mutex_unlock(&mutex);
}
```

“Defina y dé ejemplos: race-condition, heisenbug, dead-lock, interleave.”

Race-condition: se da cuando el resultado del programa depende en qué orden se ejecutan las instrucciones.Los threads “compiten” entre ellos. Ej: t1=x=x*2 y t2=x=x+1

Heisenbug: Un bug que desaparece o cambia de comportamiento cuando se intenta depurarlo. Por ejemplo, un bug que solo aparece cuando se ejecuta el programa en modo debug.

Deadlock: bug que pueden utilizar los locks para sincronizar la ejecución entre ellos. Es cuando dos o más threads se bloquean mutuamente cuando intentan utilizar recursos que otro proceso los tiene bloqueados.Por ejemplo, si el proceso A tiene el recurso 1 y quiere el recurso 2, y el proceso B tiene el recurso 2 y quiere el recurso 1.

Interleave: Ocurre cuando dos o más procesos se ejecutan de forma concurrente y sus instrucciones se mezclan entre sí. Por ejemplo, si el proceso A elige un valor de una variable compartida y el proceso B incrementa este valor, el resultado puede ser inesperado. Se enlazan instrucciones de distintos threads, generando múltiples posibles órdenes de ejecución.

“Describa detalladamente con un esquema cuál es la diferencia estructural entre un proceso y un thread. ¿Cuál es la curiosa implementación de éstos en Linux?”

La diferencia estructural entre un proceso y un thread radica en la gran diferencia que hay entre ellos en cuestión de compartir recursos. Los threads a diferencia de los procesos pueden compartir memoria, registros, file descriptors, el manejo de señales, el contexto del filesystem. En cambio un proceso no puede compartir todos estos recursos entre procesos, ya que cada uno tiene su propio código y datos estáticos. Un proceso puede contener uno o muchos threads. La implementación de estos en linux es la misma ya que no se diferencia un proceso de un thread, ambos se tratan como tareas ejecutables y no más que eso. Fork y pthread_create son wrappers de create pero con distintos argumentos.

“¿Cuáles son las diferencias entre un spin lock y un sleep lock? ¿Cuándo usaría cada uno?”

Un spin lock consta de un mecanismo de bloqueo que consiste en un bucle donde los threads esperan activamente a que un recurso esté disponible, es decir, que se libere un lock. Sirve para casos donde el recurso se libera rápidamente sin hacer cambios de contexto. En cambio, un sleep lock se da cuando los threads que no contienen el lock en el momento, se van a dormir o se desactivan, es decir, espera hasta que el recurso esté disponible. Es muy útil en los casos que los tiempos de espera son muy largos y no se quiere consumir CPU innecesariamente.

“¿Es correcta la siguiente implementación de este spinlock? Explique qué problema tiene y cómo solucionarlo:

```
void spin_lock(int* lock) {
    while (*lock != 0) {
    }
    *lock = 1;
```

}”

La siguiente implementación de spinlock no es correcta por problemas de atomicidad. La verificación y asignación se hacen en diferentes operaciones. Dos hilos pueden leer la asignación del lock y luego colocarlos en 1, no cumpliendo la exclusión mutua.

Para solucionarlo, se puede utilizar una instrucción atómica como `__sync_lock_test_and_set` o `__sync_bool_compare_and_swap` para realizar la verificación y la asignación en una sola operación atómica.

```
void spin_lock(int* lock) {
   while (__sync_lock_test_and_set(lock, 1) == 1) {
   }
}
```

“¿Cuáles de los siguientes mecanismos son compartidos entre threads de un mismo programa?” Stack Segment **File descriptors** Registros de CPU **Heap** **Code Segment** METADATA del Thread **Data Segment** **Signals.**

✅❌ **Resumen rápido por ítem:**

Mecanismo	¿Compartido entre threads?
Stack Segment	❌ No
File Descriptors	✅ Sí
Registros de CPU	❌ No
Heap	✅ Sí
Code Segment	✅ Sí
METADATA del Thread	❌ No
Data Segment	✅ Sí
Signals	✅ Parcial (depende del manejo)

FILESYSTEM

“El superbloque de un sistema de archivos indica que el (3) inodo correspondiente al directorio raíz es el #43. En la siguiente secuencia de comandos, y siempre partiendo de ese directorio raíz, se pide indicar la cantidad de inodos y bloques de datos a los que se precisa acceder (leer) para resolver la ruta dada a `cat(1)` o `stat(1)`.

 # mkdir /dir /dir/s /dir/s/w

 # touch /dir/x /dir/s/y

 # stat /dir/s/w/x Inodos: 4 Blq. datos: 4

 # stat /dir/s/y Inodos: 4 Blq. datos:3

 # ln /dir/s/x /dir/h

 # ln -s /dir/s/y /dir/y

 # cat /dir/h Inodos: 3 Blq. datos: 3

 # cat /dir/y Inodos: 7 Blq. datos: 7

Ayuda: Todos los directorios ocupan un bloque. La idea es que describan cómo `stat` llega a los archivos." hecho en hoja

"Describa VFS."

VFS (Virtual File System) es un subsistema del kernel de Linux que le da una interfaz común a todos los sistemas de archivos, para que los programas accedan a archivos sin importar si el sistema es ext4, FAT, etc. Algunos de sus componentes son: **Superbloque**: representa el sistema de archivos. **Inodo**: representa un archivo, con sus metadatos. **Dentry**: representa una entrada de directorio, parte de una ruta. **File**: representa un archivo abierto por un proceso.

"Describa la estructura de un i-nodo."

La estructura de un i-nodo es:

Tamaño del archivo, id del propietario, permisos del archivo, tiempo de acceso, y puntero al bloque de datos.

"Describir el proceso de acceso (lectura de inodos y bloques) de /home/darthmendez/opt/tool/sisop.txt suponiendo que el archivo está vacío. Además describa para qué sirven todas las estructuras involucradas."

El ejercicio resuelto en mi hoja. Las estructuras involucradas son: inodos, contienen la metadata del archivo no el archivo en sí, bloques de datos, contienen los datos, dentries, estructura de datos que representa una entrada de directorio del sistema de archivos.

Mencione una ventaja de los hard links sobre los symbolic links y una ventaja de los symbolic links sobre los hard links.

Una ventaja de los hard links sobre los symbolic links es que no requieren ninguna estructura interna para guardar ese link. No se rompen al mover o renombrar el archivo original. Hardlink: Referencia directa a un inodo, el archivo original y el hard link son iguales. Los symbolic links permiten ser implementados en sistemas generales, ya que se tiene el path del archivo y puede ser referenciado desde cualquier parte ya que el path es global, pueden apuntar a directorios. Symbolic link: Es un archivo que contiene la ruta de otro archivo.

"Describir en un esquema y detallar los componentes de un VFS en un disco de 8192 bloques, cada bloque posee una longitud de 4096 bytes. Los inodos del disco tienen un tamaño de 128 bytes."

SUPERBLOQUE=1, BITMAPS DE INODOS=1, BITMAPS DE DATOS=1, INODOS=256, BLOQUE DE DATOS=7933

Cantidad de inodos=4096/128=32, por cada bloque de inodos pueden haber 32 bloques de datos. Por lo que 8192/32=256 Inodos. Para calcular la cantidad de bloque de datos simplemente hacemos 8192-1-1-1-256=7933

MULTIPLE CHOICE

“En el CFS, ¿cómo se determina el "vruntime" (tiempo de ejecución virtual) de una tarea y cuál es su importancia?”

a. El "vruntime" se incrementa en función del tiempo de ejecución real de la tarea y de la carga del sistema; es importante porque ayuda a mantener la equidad en la asignación de CPU. b. El "vruntime" se incrementa en función de la prioridad de la tarea; es importante porque determina la prioridad de la tarea en el sistema. c. El "vruntime" se incrementa solo cuando la tarea está esperando; es importante porque reduce la latencia de las tareas en tiempo real. d. El "vruntime" se mantiene constante para todas las tareas; es importante porque simplifica la programación.

“¿Cuál es el propósito principal del VFS (Virtual File System) en Linux?”

a. Proporcionar una interfaz para la comunicación entre dispositivos. b. Permitir a los usuarios ejecutar múltiples aplicaciones en paralelo. **c. Abstraer las operaciones del sistema de archivos y permitir la coexistencia de múltiples sistemas de archivos**. d. Es una estructura lógica que organiza, gestiona y almacena archivos en dispositivos de almacenamiento.

“En el contexto de la gestión de memoria en sistemas operativos, ¿por qué los programadores generalmente utilizan malloc en lugar de brk directamente para la asignación de memoria dinámica?”

a. brk es una función de bajo nivel que solo puede asignar memoria en bloques de tamaño fijo, mientras que malloc puede asignar bloques de cualquier tamaño. **b. malloc proporciona una abstracción más alta que incluye gestión de memoria, reutilización de bloques libres, y manejo de fragmentación, lo cual no es directamente manejado por brk.** c. brk sólo puede ser utilizado por el kernel del sistema operativo y no está disponible para las aplicaciones de usuario. d. malloc es más rápido que brk porque no requiere llamadas al sistema.

“En un sistema concurrente, dos hilos están intentando actualizar un contador compartido sin utilizar ninguna forma de sincronización. ¿Cuál de los siguientes problemas describe mejor el fenómeno que puede ocurrir y cómo puede ser mitigado?”

a. Interbloqueo (deadlock): puede ser mitigado utilizando un algoritmo de detección de interbloqueo.
b. Inanición (starvation): puede ser mitigado utilizando un planificador de hilos justo.
c. Condición de carrera (race condition): puede ser mitigado utilizando mecanismos de sincronización como mutexes o locks.
d. Fragmentación de memoria: puede ser mitigado utilizando memoria dinámica mediante malloc (en lugar de brk).

“En el capítulo 1 de "Operating Systems: Principles and Practice" de Anderson y Dahlin, los autores describen tres roles principales de los sistemas operativos: "referee", "ilusionista" y "pegamento". ¿Cuál de las siguientes opciones define correctamente estos roles?”

a. Referee: Gestiona el acceso a recursos compartidos para evitar conflictos; Ilusionista: Crea la ilusión de recursos dedicados y abundantes; Pegamento: Facilita la comunicación y coordinación entre aplicaciones.
b. Referee: Asigna recursos de manera justa; Ilusionista: Mejora el rendimiento del sistema mediante optimizaciones; Pegamento: Gestiona la memoria virtual.
c. Referee: Proporciona mecanismos de seguridad y aislamiento; Ilusionista: Simplifica la interfaz de usuario; Pegamento: Administra la entrada/salida de datos.
d. Referee: Asegura que todas las aplicaciones se ejecuten en tiempo real; Ilusionista: Proporciona acceso directo al hardware; Pegamento: Coordina los procesos en tiempo compartido.

“En el contexto de la llamada al sistema fork en Unix/Linux, ¿cuál de las siguientes afirmaciones describe mejor lo que sucede cuando fork es ejecutado por un proceso?”

a. fork crea un nuevo hilo dentro del proceso existente, compartiendo el mismo espacio de direcciones.
b. fork duplica el proceso llamante, creando un nuevo proceso hijo con un espacio de direcciones independiente pero idéntico al del proceso padre en el momento de la llamada.
c. fork crea un nuevo proceso hijo que comparte el mismo espacio de direcciones con el proceso padre, pero tiene su propio contador de programa y puntero de pila.
d. fork inicia un nuevo programa especificado por el proceso llamante, cargando un nuevo ejecutable en el espacio de direcciones del proceso hijo.

“El concepto de "Copy-On-Write" (COW) es utilizado en sistemas operativos para optimizar el uso de memoria. ¿Cuál de las siguientes afirmaciones describe mejor cómo funciona Copy-On-Write y sus beneficios?”

a. COW crea inmediatamente copias duplicadas de los datos en memoria cuando se realiza un fork, mejorando la velocidad de acceso.
b. COW posterga la creación de copias duplicadas de los datos en memoria hasta que uno de los procesos intenta modificar los datos, optimizando el uso de memoria.
c. COW permite a los procesos compartir los descriptores de archivos, reduciendo la sobrecarga de apertura y cierre de archivos.
d. COW desactiva la paginación de memoria para los procesos hijo, permitiendo un acceso más rápido a la memoria

“¿Qué estructura de datos utiliza el CFS (Completely Fair Scheduler de Linux) para mantener el orden de ejecución de las tareas?”

a. Cola FIFO
b. Árbol AVL (Adelson-Velsky and Landis)
c. Árbol Rojo-Negro
d. Lista doblemente enlazada

“¿Qué estructura de datos en el VFS de Linux representa un archivo abierto?”

a. inode
b. dentry

c. file
d. superblock

“En una implementación típica de `malloc`, ¿qué estructura de datos se utiliza comúnmente para administrar los bloques de memoria libre?”

a. Árbol AVL
b. Tabla hash
c. Lista enlazada
d. Pila (stack)

“Durante el proceso de ejecución de una llamada al sistema (system call) en un sistema operativo, ¿cuál es el orden correcto de los siguientes eventos y cuál es su propósito principal?”

1. Cambio de modo de usuario a modo kernel.
2. Ejecución del manejador de la llamada al sistema en el kernel.
3. Validación de los parámetros de la llamada al sistema.
4. Retorno al modo de usuario con el resultado de la llamada al sistema.

a. 1 → 3 → 2 → 4: El propósito principal es asegurar que los parámetros sean válidos antes de ejecutar la operación del kernel.
b. 1 → 2 → 3 → 4: El propósito principal es permitir que el kernel ejecute la operación antes de verificar la validez de los parámetros.
c. 1 → 3 → 4 → 2: El propósito principal es validar los parámetros en modo usuario antes de cambiar al modo kernel.
d. 1 → 2 → 4 → 3: El propósito principal es ejecutar la operación del kernel lo más rápido posible y validar los parámetros después.

“¿Cuál de las siguientes afirmaciones describe mejor una condición de carrera (race condition) en el contexto de programación concurrente?”

a. Ocurre cuando dos procesos intentan acceder a una variable global, pero el sistema operativo serializa los accesos para evitar conflictos.
b. Es una situación en la que el resultado del programa depende del orden no controlado de la ejecución de hilos o procesos.
c. Es una técnica de optimización donde múltiples hilos comparten el uso de la CPU para mejorar el rendimiento del sistema.
d. Es un problema que solo ocurre en sistemas de tiempo real debido a la necesidad de cumplir con estrictos plazos de tiempo

“¿Cuál de las siguientes afirmaciones describe mejor la diferencia entre un sistema operativo monolítico y un sistema operativo microkernel?”

a. Los sistemas monolíticos dividen el kernel en múltiples servicios pequeños, mientras que los microkernels tienen un kernel unificado.
b. Los sistemas monolíticos implementan la mayoría de los servicios del sistema operativo en el espacio de usuario, mientras que los microkernels los implementan en el espacio del kernel.
c. Los sistemas monolíticos ejecutan todos los componentes del sistema operativo en el espacio del kernel, mientras que los microkernels ejecutan la mayoría de los

servicios del sistema operativo en el espacio de usuario.
d. Los sistemas monolíticos son más adecuados para dispositivos embebidos, mientras que los microkernels son más adecuados para sistemas de escritorio y servidores.

“¿Cuál de las siguientes afirmaciones describe mejor el propósito y funcionamiento del Translation Lookaside Buffer (TLB) en un sistema de memoria virtual?”

a. La TLB almacena copias de datos de la memoria principal para reducir los tiempos de acceso, similar a una caché de CPU.
b. La TLB almacena las traducciones de direcciones virtuales a físicas más frecuentemente utilizadas, reduciendo la necesidad de acceder a las tablas de páginas en la memoria principal.
c. La TLB almacena las páginas más frecuentemente utilizadas, reduciendo la necesidad de acceder a las páginas en la memoria principal.
d. La TLB almacena las tablas de páginas más frecue

“En el contexto de la política de scheduling Multi-Level Feedback Queue (MLFQ) en sistemas operativos, ¿cuál de las siguientes afirmaciones describe correctamente cómo funciona MLFQ y cuál es una de sus ventajas principales?”

a. MLFQ asigna a cada proceso un nivel de prioridad fijo y utiliza un time slice hasta que termina, proporcionando equidad de tiempo entre todas las prioridades para todos los procesos.
b. MLFQ divide los procesos con diferentes niveles de prioridad, y los procesos se mueven entre las colas basándose en su comportamiento y tiempo de ejecución, adaptándose a las necesidades de los procesos.
c. MLFQ asigna a cada proceso a la cola más baja de prioridad y lo mantiene allí independientemente de su comportamiento.
d. MLFQ asigna siempre el quantum de tiempo más largo disponible al proceso interactivo, asegurando que estos procesos tengan más tiempo de CPU que los procesos en segundo plano.

“En un sistema operativo moderno, ¿cuál es la principal razón para mantener separado el espacio de usuario del espacio de kernel, y qué técnica utiliza el kernel para permitir que los programas en espacio de usuario accedan a recursos privilegiados sin comprometer la seguridad del sistema?”

a. La separación protege al kernel de posibles errores en aplicaciones de usuario, y el acceso se permite mediante llamadas al sistema que validan las solicitudes antes de ejecutar operaciones privilegiadas.
b. La separación permite que el espacio de usuario ejecute código más rápido, y el acceso al kernel se realiza a través de interrupciones de hardware directas que garantizan la eficiencia.
c. La separación evita que las aplicaciones de usuario accedan a memoria restringida, y el kernel concede acceso mediante variables globales que almacenan información sensible.
d. La separación permite al kernel proteger la privacidad de los datos, y el acceso se permite mediante funciones públicas en el espacio de usuario que verifican permisos.

“En un sistema operativo, la función `malloc` se utiliza para asignar memoria dinámica en el espacio de usuario. ¿Qué sucede internamente cuando se solicita un bloque de memoria que excede el tamaño del heap disponible actual, y qué técnica emplea el sistema para gestionar este tipo de solicitudes?”

a. `malloc` falla y devuelve un puntero nulo, ya que no es capaz de asignar más memoria que la disponible en el heap actual.
b. `malloc` inicia la recolección de basura para liberar memoria y reutilizar los bloques existentes, sin necesidad de ampliar el heap.
c. `malloc` amplía el heap solicitando más memoria al sistema mediante la llamada `sbrk` o `mmap`, ajustando así el tamaño del espacio de memoria disponible dinámicamente.
d. `malloc` intercambia el contenido del heap con la memoria secundaria (disco) para liberar espacio, asignando el bloque solicitado sin afectar el tamaño del heap.

“El Completely Fair Scheduler (CFS) de Linux utiliza un árbol rojo-negro. ¿Qué ventaja ofrece esta estructura al objetivo del CFS de asignar CPU de forma justa?”

a. Asigna CPU en orden de llegada, priorizando los procesos más antiguos en la cola de ejecución.
b. Almacena prioridades de procesos y ejecuta primero los de menor prioridad para una distribución justa.
c. Permite actualizar el proceso con menor tiempo de ejecución virtual en tiempo logarítmico, permitiendo así encontrarlo y seleccionarlo rápidamente durante un cambio de contexto.
d. Divide procesos en grupos y asigna CPU proporcional al tamaño de cada grupo.

“¿Qué papel juega el descriptor de archivo en un sistema Linux, y cuál de las siguientes afirmaciones es correcta respecto a cómo un proceso utiliza un descriptor de archivo?”

a. El descriptor de archivo almacena información sobre los permisos del sistema de archivos y permite que diferentes procesos accedan a archivos sin restricciones.
b. Un descriptor de archivo es un índice en una tabla que gestiona el sistema operativo, permitiendo a un proceso realizar operaciones de lectura/escritura sobre un archivo abierto.
c. Los descriptores de archivo se almacenan en la memoria principal y contienen datos del archivo completo para reducir los tiempos de lectura.
d. Un descriptor de archivo almacena la ubicación física de un archivo en disco, facilitando el acceso directo a sectores específicos.

“¿Qué técnica de planificación de procesos es utilizada por los sistemas operativos para evitar que un proceso acapare indefinidamente la CPU?”

a. Planificación sin desalojo.
b. Planificación circular.
c. Planificación por turnos con un quantum.
d. Planificación basada en eventos.

“¿Qué sucede en el sistema cuando un proceso en estado "zombie" es eliminado?

a. Su entrada en la tabla de procesos se libera completamente una vez que el proceso padre lee el estado de salida del proceso hijo.
b. El sistema lo pasa a un estado suspendido hasta que pueda ser reasignado.
c. El proceso es automáticamente reasignado a otros procesos activos.
d. Su estado permanece en un "daemon" y se mantiene activo.

¿Cuál de las siguientes técnicas es utilizada por el kernel de Linux para optimizar el acceso a los sistemas de archivos?

a. Utiliza la memoria swap para almacenar partes del sistema de archivos en caso de falta de memoria.
b. Almacena partes del sistema de archivos en la memoria para acceder repetidamente al disco por la misma información.
c. Divide los archivos de grandes tamaños en fragmentos, minimizando la fragmentación del disco.
d. Realiza desfragmentación periódica del sistema de archivos durante el tiempo de inactividad.

EJERCICIOS FALOPA DE CHAT

🧠 **EJERCICIO 1: Condición de carrera simple**

```
#include <stdio.h>
#include <pthread.h>

int contador = 0;

void* incrementar(void* arg) {
    for (int i = 0; i < 100000; i++) {
        contador++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, incrementar, NULL);
    pthread_create(&t2, NULL, incrementar, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Contador: %d\n", contador);
    return 0;
}
```

Preguntas:

a) ¿Por qué este código tiene condición de carrera?
b) ¿Cuál sería un rango esperable de resultados para `contador` al ejecutarlo varias veces?
c) Corregí el código usando `pthread_mutex_t`.

🔐 EJERCICIO 2: Lock mal usado

```
#include <stdio.h>
#include <pthread.h>

int valor = 0;
pthread_mutex_t lock;

void* sumar(void* arg) {
    pthread_mutex_lock(&lock);
    for (int i = 0; i < 10000; i++) {
        valor++;
    }
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t hilos[4];
    pthread_mutex_init(&lock, NULL);

    for (int i = 0; i < 4; i++) {
        pthread_create(&hilos[i], NULL, sumar, NULL);
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(hilos[i], NULL);
    }

    printf("Valor final: %d\n", valor);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Preguntas:

a) ¿Está bien ubicado el `pthread_mutex_lock`? ¿Qué pasaría si estuviera **dentro del for**?
b) ¿Qué pasaría si se olvidara llamar a `pthread_mutex_unlock`?
c) ¿Qué sucede si dos hilos usan el mismo mutex y uno entra en estado de espera infinita?

🔄 EJERCICIO 3: Lecturas sin proteger

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int valor = 0;
pthread_mutex_t lock;

void* lector(void* arg) {
    while (1) {
        int copia = valor;
        if (copia % 2 != 0) {
            printf("Valor inconsistente leído: %d\n", copia);
        }
    }
    return NULL;
}

void* escritor(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        valor++;
        valor++;
        pthread_mutex_unlock(&lock);
        usleep(100);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, lector, NULL);
    pthread_create(&t2, NULL, escritor, NULL);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

Preguntas:

a) ¿Por qué es posible leer valores impares, si siempre se suma de a dos?
b) ¿Cómo se puede corregir el acceso de lectura para evitar race conditions?
c) ¿Qué estructura podría usarse para mejorar la concurrencia entre lectores y escritores?

💥 EJERCICIO 4: Deadlock

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int saldo;
    pthread_mutex_t lock;
} Cuenta;

void transferir(Cuenta* origen, Cuenta* destino, int monto) {
    pthread_mutex_lock(&origen->lock);
    pthread_mutex_lock(&destino->lock);

    origen->saldo -= monto;
    destino->saldo += monto;

    pthread_mutex_unlock(&destino->lock);
    pthread_mutex_unlock(&origen->lock);
}
```

Preguntas:

a) ¿Cómo se puede generar un **deadlock** con este código?
b) Proponé una solución para evitarlo (pista: orden de adquisición de locks).
c) Reescribí la función `transferir` para prevenir deadlocks.