

2. Asignación enlazada:

- Almacena punteros, toman espacios de almacenamiento
- no requiere tamaños, puede crecer.
- Acceso secuencial, tiene un problema, ya que debe recorrer los bloques para llegar a un bloque en particular.
- método vulnerable a falla de un bloque, si pierdo info, pierdo de todos los recorridos, pero aun así es mejor que la asignación contigua.

3. Asignación Indexada:

- cada archivo contiene un index block, que contiene los bloques del archivo.
- El ultimo cluster almacena EOF
- Acceso aleatorio sin fragmentación externa.
- Usa espacio por el index block, en este se tienen todas las referencias de donde esta el archivo.
- No hay fragmentación externa y existe un acceso aleatorio, ya que tengo info de los bloques.
- problema: gasta espacio, punto unico de falla.

→ Tipos de asignación indexada:

1. Esquema enlazado: ultima entrada de un index block, apunta a otro index block; otro bloque guarda otro index block.

2. Multi-level: funciona como multi-level Page Tables, se tienen muchos index block asociados a muchos index block.

3. Combinado: se podría tener de todos. (index block + niveles)

→ diferencia entre:

acceso secuencial: necesito recorrer todo para llegar a un bloque en particular

acceso aleatorio: se donde esta guardado el archivo en el disco

Virtualización

→ Las MV nacen ya que la interacción entre los diversos SO deben especificarse de manera exclusiva para cada gestor.

→ Las MV frente a esto, sugieren una abstracción para el acceso independiente a recursos.

→ Beneficios:

- Aislamiento entre sistemas involucrados.
- manejo temporal de ejecución
- concurrencia y diversidad de SO.
- pruebas e investigación sobre SO volátiles

Sistemas Operativos / PARTE I

→ sistema operativo es un software encargado de administrar los recursos disponibles de la máquina con intención mínima del usuario.

→ forma parte de un sistema operativo: Kernel, suide, bajo nivel, navegador web, reproductor de video, GUI, etc.

→ **S.O = Kernel + programa de sistemas**

→ **Procesos:**

- un programa en ejecución, ocupa un cierto espacio en memoria, en donde se tiene la capacidad para leer, escribir datos y además contiene un stack de ejecución.

- tiene acceso a una serie de recursos.

- existen múltiples ejecutándose, existe comunicación entre ellos.

→ **memoria:**

- espacio de memoria, deberá ser compartido por los distintos procesos.

- se debe trabajar con la memoria física y administrar su espacio correctamente.

→ **archivos:**

- espacios de almacenamiento en memoria secundaria, manejados por S.O a través de syscalls.

- organizados en forma de árbol.

→ **I/O:**

- dispositivos que ingresan datos al S.O y producen salidas.

→ **Shell:**

- interfaz de comandos UNIX.

→ **kernel:**

- núcleo del S.O, software que corre en todo momento.

- se ejecuta en modo privilegiado.

- incluye manejo de errores, logging, ipc, protección y seguridad, i/o, etc.

Procesos

→ Existe paralelismo, para cumplir esto hay que compartir recursos y administrar el uso de ellos por lo que el sistema operativo es clave.

→ un proceso contiene, código del programa, datos del programa y estado de actividad.

Threads

→ asociados a una tarea específica.

→ asociados a un proceso, de modo que un proceso pueda efectuar múltiples tareas.

→ La información se almacena en el PCB.

→ Para ejecutar varios procesos al mismo tiempo, se utilizan scheduling de procesos.

→ se usan colas de procesos para intervenir: job queue (procesos que entran al sistema), ready queue (procesos listos y en RAM).

→ se comunican entre ellos, mediante IPC (Interprocess Communication).

→ se llevan a cabo de dos maneras (para mensajes (pipes) o memoria compartida).

→ **syscall**

- es pedirle al kernel que haga algo por nosotros
- métodos utilizados por los programas de aplicación para comunicarse con el kernel
- proporciona una interfaz entre un proceso y el sistema operativo para permitir que los procesos a nivel de usuario soliciten servicios del S.O.

• ejemplos de syscalls:

- control de procesos: `fork()`, `exit()`, `wait()`
- manipulación de archivos: `open()`, `read()`, `write()`, `close()`
- mantenimiento de la información: `getpid()`, `sleep()`
- protección: `chmod()`, `umask()`, `chown()`
- comunicación: `pipe()`, `shmget()`, `mmap()`

- **general protection fault**: interrupción definida por un mecanismo de protección en respuesta a una violación de acceso causada por algún código de ejecución.

- **IPC**: mecanismo que permite a los procesos comunicarse entre ellos, necesaria para que los procesos puedan enviar resultados y peticiones entre ellos.

Los pipes son canales de comunicación entre procesos, de tal manera que el output de un proceso sea el input de otro. consideran múltiples variables (unidireccionales, bidireccionales, half/full duplex, parentesco entre procesos).

→ **FORK**: orfización, crea procesos hijos, copia del mismo proceso. una generalidad del `fork()` es que el proceso padre genera al proceso hijo y espera que este termine.

- **contexto**: es mínimo conjunto de datos utilizados por un proceso que debe ser guardado para permitir su interrupción en un momento dado.

- **cambio de contexto**: detener la ejecución de un proceso para dar paso a la ejecución de otro distinto.

Codigos Fork()

FORK()

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
    pid_t pid = fork();
    if(pid == -1){
        printf("Error en el fork\n");
        exit(0);
    }
    else
        if(pid==0){
            printf("soy luke skywalker y mi pid es: %d\n", getpid());
            exit(0);
        }
        else {
            // sleep(2);
            printf("soy darth vader y mi pid es: %d\n", getpid());
        }
}
```

proceso: (línea de ejecución, pero es un proceso)

se almacena un identificador, que crea una copia y guarda la salida del fork en p.

obtener el id

hijo

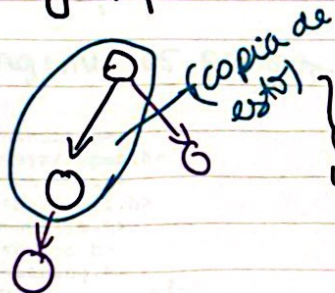
padre

P1
p=0

P2
p=0

- Siempre el padre termina antes que el hijo.

(* dentro del if del padre se crea otro hijo del mismo padre)

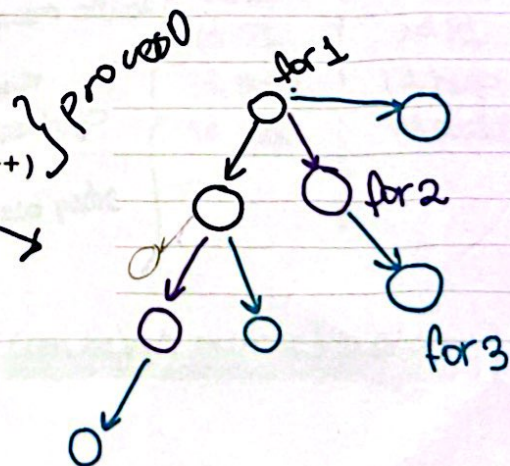


pid_t pid = fork();
se se hace otro fork para esto

?

14

```
int main(){
    for (int i = 0 ; i < 3 ; i++){
        fork();
        printf("hola\n");
    }
}
```



SYSCALLS que se llaman y se pueden relacionar con procesos a medida que trabajamos con procesos

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>

int main(){

    pid_t pid = vfork();

    if(pid == -1){
        printf("Error en el fork\n");
        exit(0);
    }
    else
        if(pid==0){
            printf("soy luke skywalker y mi pid es: %d\n",getpid());
            exit(0);
        }
        else {
            printf("soy darth vader y mi pid es: %d\n",getpid());
        }
}
```

El padre espera que los hijos terminen y hace el lo que necesita.

multitasking: muchos procesos en memoria.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
    pid_t pid;
    printf("%d\n",getpid());
    pid = fork();

    if (pid == 0){
        printf("%d\n",getpid());
    }
    else if (pid > 0){
        printf("%d\n",getpid());
    }

    return 0;
}
```

FORK()

no se sabe cual viene primero.

libreria	o
id P1	id P1
id padre	id hijo
id hijo	id padre

(con break imprimir hijo primero)


```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
```

```
    pid_t pid;
    pid = fork();
```

```
    if (pid == 0){
        printf("Proceso hijo utilizando la primitiva exec()\n");
        execlp("./clase5_hola","",NULL);
        printf("y por que no imprime esto???\n");
    }
```

```
    else if (pid > 0){
        printf("El padre sigue aca...\n");
    }
```

```
    return 0;
```

```
}
```

```
#include<stdio.h> clase5_hola
```

```
int main(){
```

```
    printf(" sup dude ;D \n");
    return 0;
```

```
}
```

EXEC()

→ cambiar las línea de instruccion.

→ imprime esa línea
→ cambia a ejecutar lo
de clase 5_hola
→ esto no lo imprime
y luego imprime esto.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
```

```
    pid_t pid;
    pid = fork();
```

```
    if (pid == 0){
        printf("Proceso hijo utilizando la primitiva exec()\n");
        execlp("./clase5_hola","",NULL);
    }
```

```
    else if (pid > 0){
        wait(NULL);
        printf("El padre sigue aca...\n");
    }
```

```
    return 0;
```

```
}
```

WAITC()

→ espera que el hijo
termine la ejecución;
(siempre se pone en proceso padre).

→ esto imprime
si se imprime el
hijo y termina el padre.

→ esto se ejecuta ultimo ya que
el padre espera que esto se ejecute


```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
```

```
    pid_t pid;
    pid = fork();
```

```
    if (pid == 0){
        for (int i = 0 ; i < 5 ; i++){
            printf("Que tal, soy el proceso hijo\n");
            sleep(1);
        }
    }
```

```
    else if (pid > 0){
        for (int i = 0 ; i < 5 ; i++){
            printf("Que tal, soy el proceso padre\n");
            sleep(1);
        }
    }
```

```
    return 0;
```

```
}
```

(En sleep,
padre o hijo todos
primeros)

SLEEP() → detiene el proceso por un segundo según código.

duerme por un segundo y se imprime.

que tal, soy el proceso hijo
que tal, soy el proceso padre.

(se imprime 5 veces cada uno,
de manera desordenada,
esperando un segundo cada uno).

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
```

```
int main(){
```

```
    pid_t pid;
    int status;
    pid = fork();
```

```
    if (pid == 0){
        for (int i = 0 ; i < 15 ; i++){
            printf("Que tal, soy el proceso hijo\n");
            sleep(1);
        }
    }
```

```
    else if (pid > 0){
        for (int i = 0 ; i < 5 ; i++){
            printf("Que tal, soy el proceso padre\n");
            sleep(1);
        }
    }
```

```
    kill(pid, SIGKILL);
    printf("vuela alto proceso hijo\n");
    return 0;
```

```
}
```

(usa señales)

KILL() → envía una señal al hijo, para que termine su proceso.
Siokill → terminar su proceso.

se pasa el id del hijo
kill(id, señal = SIGKILL)

KILLALL()

Elementos de control de procesos

- **Interrupciones**: generan cambio de contexto señales que indican una necesidad de detener un proceso en ejecución para dar lugar a otra requiriendo que necesite atención inmediata.
- **multitasking = cambio de contexto**.
- con una cola de interrupciones se manejan las interrupciones, además cuentan con un nivel de prioridad.
- interrupciones de HW se generan por dispositivos de I/O, y las de SW se generan al finalizar la ejecución de programas.

→ Scheduling (planificación)

- **Long term scheduling**: selección de procesos que quedan en la ready queue.
- **medium term scheduling**: modifica temporalmente el grado de multiprogramación y hace swapping (RAM → HDD / HDD → RAM).
- **short term scheduling**: manda procesos de la ready queue a CPU y efectúa el cambio de contexto.
- scheduling importante para poder organizar la multiprogramación.
- thrashing = ocurrencia + cambio de contexto.
- **tipos de scheduling**

1. **scheduling no-expropiativo**: este termina con el proceso al momento de pasarlo a la CPU.

2. **Scheduling expropiativo**: este no necesariamente termina con el proceso al momento de pasarlo a la CPU de una vez.

3. **batch**: no interactúan con el usuario.

4. **Interactive**: interactúan con el usuario.

5. **Real time**: los procesos tienen un plazo (deadline).

Algoritmos

→ **FIRST COME FIRST SERVED**: (no expropiativo)

• turnaround = endtime - arrival time.

→ **SHORTEST JOB FIRST**: (no expropiativo)

• el más corto en CPU va primero.

→ **ROUND ROBIN**: (expropiativo)

• q = tiempo de CPU que le asigna máximo a un proceso en CPU.

• response time = execution start - arrival time

→ **PRIORITY SCHEDULING**: (expropiativo)

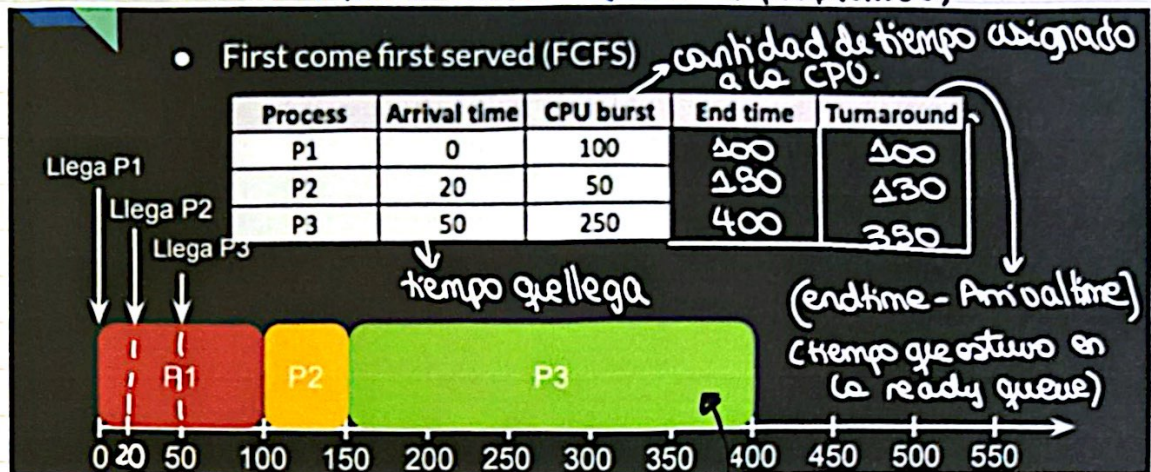
• priority: mientras más alta sea, se le da el proceso con más prioridad.

→ **EARLIEST DEADLINE FIRST**: (expropiativo)

• deadline: (prioriza a procesos con deadline menor), que se ejecute antes o igual que el deadline.

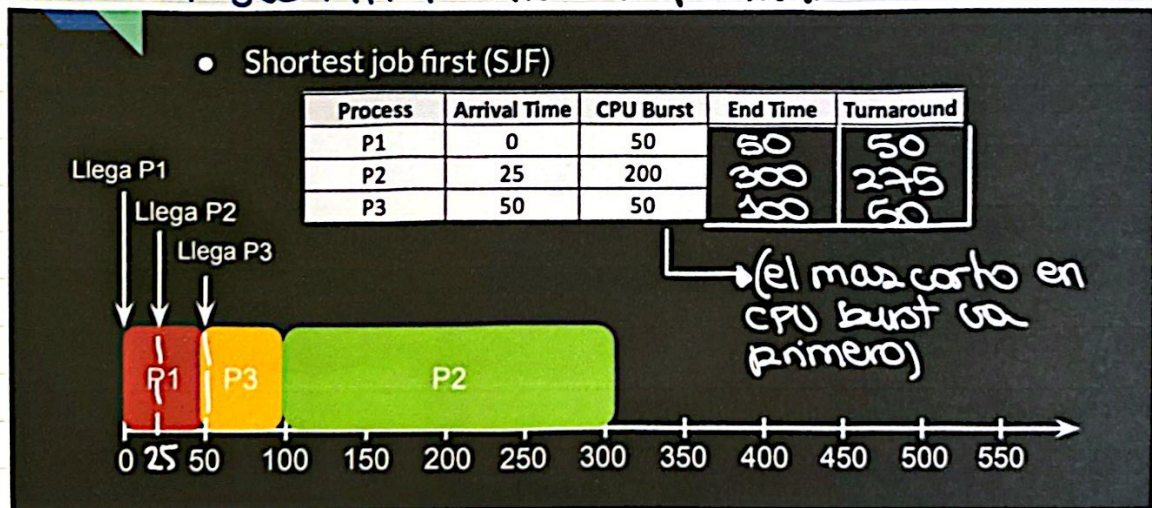
ALGORITMOS:

FIRST COME FIRST SERVED: (no expropiativo) → (batch)



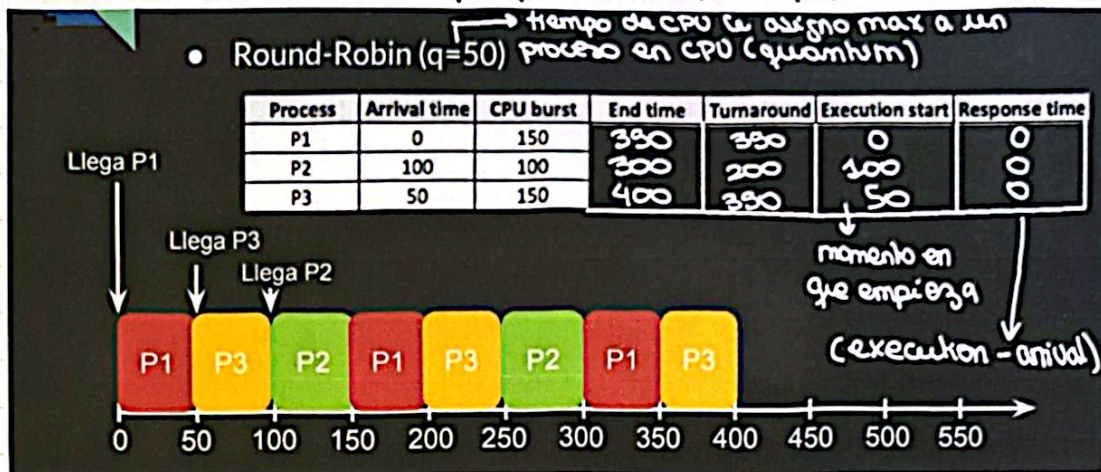
(llega el 1, luego el 2, luego el 3)

SHORTEST JOB FIRST: (no expropiativa) → (batch)

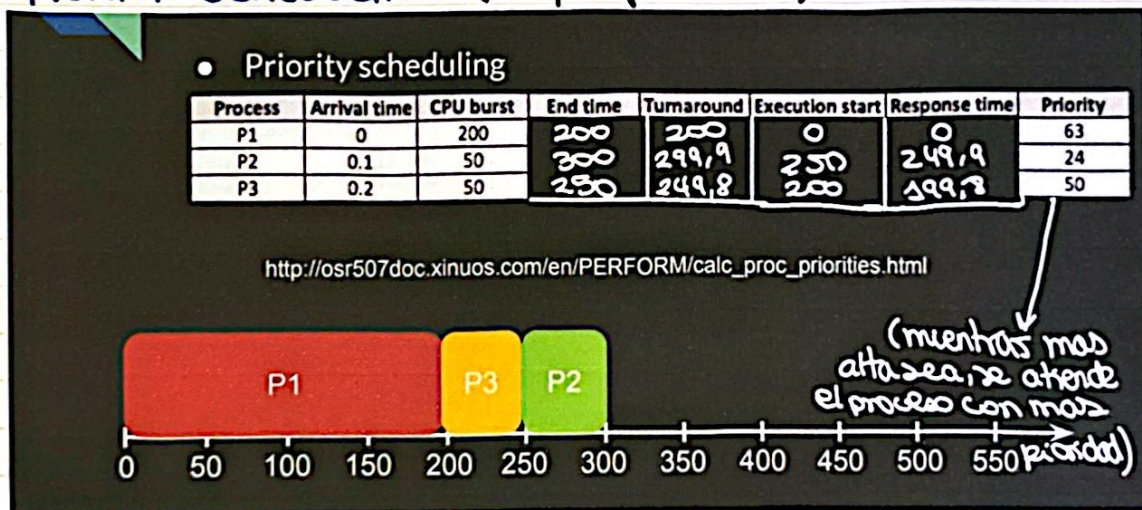


(el 1 dura menos y llega primero, luego el 3 dura menos que el 2 y luego el 3).

ROUND ROBIN: (expropiativo) ^(interactivo) → P1 → P3 → P2



PRIORITY SCHEDULING: (expropiativo) ^(interactivo)

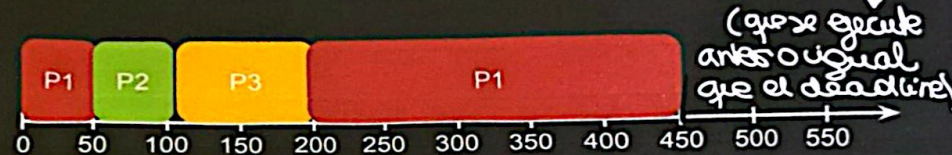


EARLIST DEADLINE FIRST: (expropiativo)

- Earliest deadline first

(prioriza a procesos con un deadline menor)

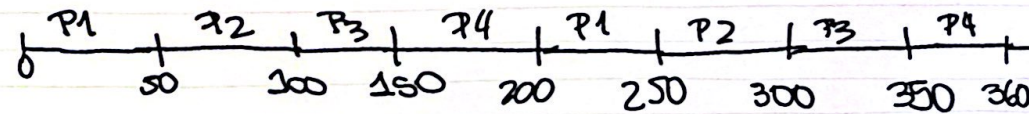
Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Deadline
P1	0	300	450	450	0	0	500
P2	50	50	100	50	50	0	350
P3	75	100	200	125	100	25	450



EJERCICIO:

	Arrival	CPU BURST	End time	turnaround	Execution start	response time
P1	0	100	250	250	0	0
P2	30	120	380	350	50	20
P3	60	250	430	370	100	40
P4	120	60	360	240	150	30

2) round robin $q=50$. $\rightarrow (P1 \rightarrow P2 \rightarrow P3 \rightarrow P4)$



Sincronización

Threads: procesos livianos que al igual que los procesos requieren de datos específicos para ejecutar.

→ diferencias:

- un thread siempre está asociado a un proceso y ejecuta una tarea específica.
- cada proceso tiene asociado uno o más threads.
- menos cosas que las de los procesos.
- un thread comparte datos con su proceso padre, pero tiene datos propios.

→ tipos:

- **user level threads**: threads asociados a los procesos de usuario y no dependen del S.O.
- **kernel level threads**: threads que operan en modo kernel (osea dependen del S.O.) y que representan la ejecución de una tarea.

→ Los procesos multithread podrían ejecutar más lentamente que procesos secuenciales.

• operación típicas:

1. **thread create()**: crea un thread.
2. **thread_exit()**: termina un thread.
3. **thread_join()**: espera que otro thread termine.
4. **thread_yield()**: cede el paso a otro thread.

• Problemas de las syscalls:

→ existe un orden no especificado de ejecución entre los distintos procesos.

• Sincronización:

→ objetivo es sacar el mayor provecho a la concurrencia pero sin afectar la lógica de los programas.

→ existe un interleaving de las operaciones en el código.

→ **race conditions**: representan una dependencia de las salidas de una operación en base al orden temporal de ejecución de las instrucciones internas sin control por parte del programador.

• Sección crítica:

→ región de memoria de acceso compartido sobre la que trabaja código.

→ requiere que únicamente un proceso/thread esté trabajando sobre dicha región al mismo tiempo.

→ se tiene mecanismos para que el acceso sea secuencial y único a dicha sección de memoria.

→ conceptos sección crítica:

1. **exclusión mutua**: uno o más thread trabaja solo un thread trabajando con variable y luego el otro.

PROARTE®

2. Progreso: cual thread pasa, para un tiempo acortado.

3. Ausencia de inanición: todos deben pasar por la sección crítica.

Herramientas para asegurar que un proceso este realizándose a la vez en la SC.

→ deshabilitar los temporales.

• **RIESGO DE COLGARSE:** si se deshabilitan los temporales y no se vuelven a habilitar posterior a la ejecución de la SC, la CPU queda asignada al proceso actual sin poder cambiarse al otro.

• **BUSYWAITING:** significa que la CPU esta evaluando las variables de sincronización, pero sin avanzar en el procesamiento.

• **ATOMICIDAD:** no se asegura que los cambios en los valores que sincronizan trabajen atomicamente; se requieren de instrucciones atomicas.

→ Existen los siguientes que se muestra el código en las siguientes paginas.

1. Spinlock:

→ pueden entrar 2 a la sección crítica.

2. Peterson:

→ es para algoritmos con 2 procesos

3. Locks de mutex:

→ garantiza exclusión mutua

→ se abstrae una variable y se garantiza la atomicidad

4. Semáforos

→ permite que accedan $N > 0$ pero limitados threads a la SC.

→ $P()$, $wait()$: decrementa el valor del contador, entra un thread de la SC.

→ $V()$, $signal()$: incrementa el valor del contador, solo un thread de la SC.

5. Variables de condición

→ limita quien entra primero a la S.C.

→ mecanismo con condición, no con limitación de threads.

• SPINLOCK()

```

/*
  Compiler:
  gcc spinlock.c -lpthread
*/

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

int max;
volatile int counter = 0; // Variable compartida global
int lock = 0; // 0: SC está libre, 1: SC está ocupada

void *mythread(void *arg)
{
    while(lock);
    lock = 1;
    char *letter = arg;
    int i; // atLock (privado por thread)
    printf("as: inicio (direccion de i: %p)\n", letter, &i);
    for (i = 0; i < max; i++)
    {
        counter = counter + 1; // compartido: solo uno
    }
    printf("as: terminado\n", letter);
    lock = 0;
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "uso: ./spinlock <n_ciclos>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: inicio [contador: %d] [i: %d]\n", counter, (unsigned long) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join esperan a que los threads terminen
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: terminado [contador: %d] [esperado: %d]\n", counter, max*2);
    return 0;
}

```

para porque lock vale 0.

cae en un ciclo.

termina y luego entra el otro proceso.

PROBLEMA A ESTO: - pueden entrar los dos a la SC.
 sino se pone el candado antes
 entre un proceso y otro.
 - desperdicia CPU. **Busy waiting.**

• TURN() → PETERSON → ES PARA 2 PROCESOS

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

int max;
volatile int counter = 0; // Variable compartida global
int turn = 1; // A que thread le toca entrar en la SC
int flag[2] = {0,0}; // Arreglo que indica si algun thread quiere entrar a la SC

void *mythread1(void *arg)
{
    int me = 0;
    int other = 1 - me;
    char *letter = arg;
    int i; // stack (privado por thread)
    flag[me] = 1; // Quiero entrar a la SC
    turn = other;
    while(flag[other] && turn == other); // Espero si no me toca
    printf("ts: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
        counter = counter + 1; // compartido: solo uno
    }
    printf("ts: terminado\n", letter);
    flag[me] = 0; // Ya no necesito ni quiero entrar a la SC
    return NULL;
}

void *mythread2(void *arg)
{
    int me = 1;
    int other = 1 - me;
    char *letter = arg;
    int i; // stack (privado por thread)
    flag[me] = 1; // Quiero entrar a la SC
    turn = other;
    while(flag[other] && turn == other); // Espero si no me toca
    printf("ts: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
        counter = counter + 1; // compartido: solo uno
    }
    printf("ts: terminado\n", letter);
    flag[me] = 0; // Ya no necesito ni quiero entrar a la SC
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "uso: ./turn <n_ciclos>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: inicio [contador = %d] [%lx]\n", counter, (unsigned long) &counter);
    pthread_create(&p1, NULL, mythread1, "A");
    pthread_create(&p2, NULL, mythread2, "B");
    // join esperan a que los threads terminen
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: terminado\n [contador: %d]\n [esperado: %d]\n", counter, max*2);
    return 0;
}
```

} identificadores.

thread1

arreglo, que proceso o thread entra a x

} identificadores

thread2

garantiza que uno pase, no dejando a ambas en espera (siempre pasa al que va primero)

TAREA: si se cumple el algoritmo de exclusividad mutua en PETERSON, con una CPU y un pr

→ (se garantiza exclusión mutua)

LOCKS DE MUTEX: (no genera busy waiting)
se abstrae a una variable que garantiza la atomicidad de su manejo y exclusión mutua de acceso a la ec.

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* funcionThread(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

Sección crítica

MUTEX MALO:

```
/*
  Compilar:
  gcc -o mutex_mal mutex_mal.c -lpthread
*/

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;

void* funcionThread(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
    while(i < 2)
    {
        err = pthread_create(&tid[i], NULL, &funcionThread, NULL);
        if (err != 0)
            printf("\nNo se puede crear el thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

MUTEX BUENO.

```
/*
  Compilar:
  gcc -o mutex_bien mutex_bien.c -lpthread
*/

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* funcionThread(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n Falla inicializacion de mutex \n");
        return 1;
    }
    while(i < 2)
    {
        err = pthread_create(&tid[i], NULL, &funcionThread, NULL);
        if (err != 0)
            printf("\nNo se puede crear el thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```


→ (no genera busywaiting)
→ (se ejecuta un thread a la vez)

SEMAFOROS → (define cuantos threads entran a la SC).

```
sem_t mutex;
void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("¡¡Entró a la SC!\n");

    //critical section
    printf("¡¡ Como soy tan bacán... me voy a dormir durante la SC... \n");
    sleep(4);
    printf("¡¡ Buenos días!!! \n");

    //signal
    printf("¡¡Salgo de la SC!\n");
    sem_post(&mutex);
}
```

→ (si el semáforo llega a 0
no pueden entrar más
procesos)

se comporta como mutex → **la variable es 1** → **N: cantidad de procesos que pueden entrar.**
→ **siempre será 0** → **inicializar el semáforo**

```
int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread, NULL);
    sleep(2);
    pthread_create(&t2, NULL, thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&mutex);
    return 0;
}
```

- Son mecanismos de sincronización inventados por Dijkstra.
- En vez de tratarse de un lock que restringe el acceso de un único thread a una SC, permite que accedan $N(>0, N(\text{limitado}))$ threads a la SC.
- Para esto se implementan 2 funciones:
 - **PC, wait()**: intenta decrementar el valor del contador para simbolizar que ha entrado un thread a la SC.

*** V(), signal()**: intenta incrementar el valor del contador
representa una liberación libre más
para que otro thread pueda entrar a la SC

VARIABLES DE CONDICIÓN:

[illegible]

(Quiero que primero se llene y luego sea y muestre las variables)

si se ejecuta leer primero: se estanca hasta que se le mande una señal para poder pasar.

```
(pthread_cond_wait(&cond_var, &fill_mutex);
```

- Son mecanismos de sincronización en que la condición no tiene de limitar la cantidad de threads que acceden a la SC, sino que se trata de condiciones arbitrarias.

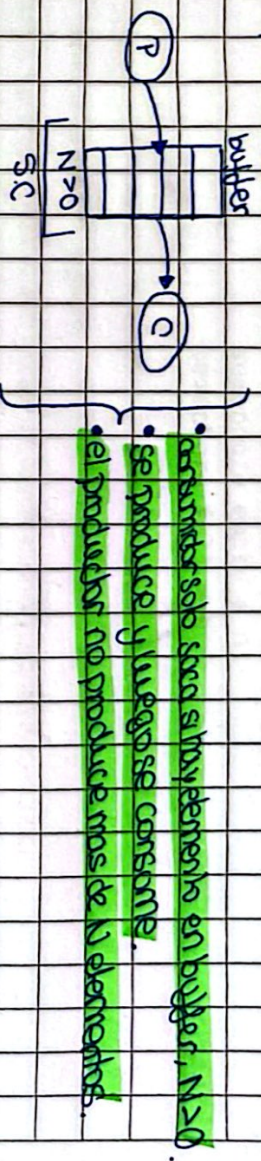
* Trabaja con dos operaciones:

- `wait()` que siempre bloquea al thread
- `signal()` que despierta a un thread bloqueado en caso que lo haya.

Fabulas / sincronización

Fábula 1: Fábula del gusón parrillero

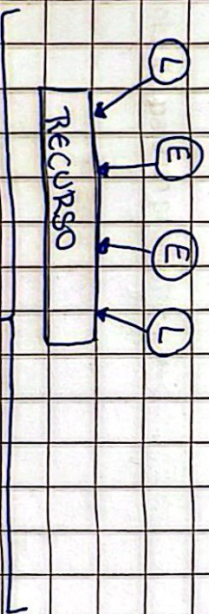
- Estudiantes hambrientos, comida gratis en una mesa, se sirven platos de carne, los gobiernos un huevito carne y lo cocinan ordenadamente y en secuencia sobre una mesa para los estudiantes.
- problema: productor - consumidor



- se usa un semáforo para C y P y un mutex para el buffer.

Fábula 2: Fábula de los estudiantes y ayudantes (escritas)

- modo educacional queda así: un profesor habla y el escriba anota mientras que el escriba anota, el preguntado queda a su disposición únicamente, una o más estudiantes pueden escribir del preguntado, pero el escriba no puede escribir lo que dice el ~~profesor~~ el profesor hasta que el último estudiante haya terminado de escribir.



- profesor habla, ayudante escribe
- solo se puede acceder como estudiante, si el escriba deja de escribir y si los estudiantes dejan de escribir accede el escriba
- variables de condición:
 - si hay lectores, no hay escritores y viceversa en el recurso.
- semáforos limita la cantidad de lectores sobre el recurso, y un mutex para los escritores al mismo tiempo en el recurso.

Ejemplos de sincronización

Problemas clásicos: el problema de los productores y consumidores (*The Bounded-Buffer Problem*)

- Se usa comúnmente para ilustrar el poder de las primitivas de sincronización.
- Los procesos de productor y consumidor comparten las siguientes estructuras de datos:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

- **n**: número de *slots* en el *buffer*.
- **mutex**: semáforo binario que brinda exclusión mutua para la región crítica.
- **empty**: número de *slots* disponibles.
- **full**: número de elementos que existen en el *buffer*.

SOYO 2020 Gustavo C. Diehl

Ejemplos de sincronización

Problemas clásicos: el problema de los productores y consumidores

- Solución para un productor y un consumidor:

Productor

Consumidor

```
while (true) {
    item = produce_item();
    wait(empty);
    insert_item(item);
    signal(full);
}

while (true) {
    wait(full);
    item = remove_item();
    signal(empty);
    consume_item(item);
}
```

SOYO 2020 Gustavo C. Diehl

Ejemplos de sincronización

Problemas clásicos: el problema de los productores y consumidores

- Solución para múltiples productores y consumidores:

Productor

Consumidor

```
while (true) {  
    item = produce_item();  
    wait(empty);  
    wait(mutex);  
    insert_item(item);  
    signal(mutex);  
    signal(full);  
}  
  
while (true) {  
    wait(full);  
    wait(mutex);  
    item = remove_item();  
    signal(mutex);  
    signal(empty);  
    consume_item(item);  
}
```

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el problema de los lectores-escritores (*The Readers-Writers Problem*)

- Considerar una base de datos compartida entre varios procesos concurrentes.
- Algunos de estos procesos leen la base de datos (lectores), mientras que otros la actualizan (escritores).
- No ocurre ningún problema si dos lectores acceden a los datos compartidos simultáneamente.
- Sin embargo, si un escritor y algún otro proceso (ya sea lector o escritor) acceden simultáneamente, podrían producirse errores.
- Para garantizar que no surjan estas dificultades se requiere que los escritores tengan acceso exclusivo a la base de datos mientras la escriben.
- El problema de los lectores y escritores tiene distintas variaciones y todas ellas involucran prioridades.
- Por ejemplo: supongamos que en una clase hay un pizarrón, donde solo una persona puede escribir a la vez, pero al mismo tiempo muchos lectores pueden leer el pizarrón.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el problema de los lectores-escritores

- El primer problema de lectores y escritores
 - Requiere que ningún lector espere a menos que un escritor ya haya obtenido permiso para usar el objeto compartido.
 - Es decir, ningún lector debe esperar a que otros lectores terminen simplemente porque un escritor está esperando.
- El segundo problema de lectores-escritores
 - Requiere que, una vez que un escritor esté listo, ese escritor realice su escritura lo antes posible.
 - Si un escritor está esperando acceder al objeto ningún lector nuevo puede comenzar a leer.
- Una solución a cualquiera de ambos problemas puede resultar en inanición.
- En el primer caso, los escritores pueden caer en inanición mientras que en el segundo caso son los lectores.
- Por esta razón, se han propuesto otras variantes; como el tercer problema de lectores-escritores.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el problema de los lectores-escritores

Escritor

```
while(true) {  
    wait(mutex);  
    // Escritura  
    signal(mutex);  
}
```

Lector

```
while(true) {  
    wait(mutex)  
    // Lectura  
    signal(mutex)  
}
```

- Solución **incorrecta o subóptima**:
 - Es posible que un hilo lector T_1 tenga el *lock*, y luego otro hilo lector T_2 solicite acceso.
 - Sería ineficiente que T_2 espere hasta que T_1 estuviera listo antes de comenzar su propia operación de lectura; en cambio,
 - Se debe permitir que T_2 lea el recurso junto con T_1 porque las lecturas no modifican los datos, es decir, las lecturas concurrentes son seguras.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el primer problema de los lectores-escritores

- Solución al primer problema de lectores y escritores; "preferencia lectores".

```
semaphore resource = 1;  
semaphore rmutex = 1;  
int read_count = 0;
```

- Variable `read_count`: cantidad de hilos que están leyendo el objeto actualmente.
- Semáforo `rmutex`: utilizado para la exclusión mutua de la variable `read_count`.
- Semáforo `resource`: exclusión mutua para escritores.
 - También lo utiliza el primer o el último lector que ingresa o sale de la sección crítica.
 - No se utiliza por lectores que entran o salen mientras otros lectores están en sus secciones críticas.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el primer problema de los lectores-escritores

Escritor

```
while(true) {  
    wait(resource);  
    // Escritura  
    signal(resource);  
}
```

Lector

```
while(true) {  
    wait(rmutex);  
    read_count++;  
    if (read_count == 1)  
        wait(resource);  
    signal(rmutex);  
    // Lectura  
    wait(rmutex);  
    read_count--;  
    if (read_count == 0)  
        signal(resource);  
    signal(rmutex);  
}
```

SOYD 2020 Gustavo C. Distel

11

Ejemplos de sincronización

Problemas clásicos: el segundo problema de los lectores-escritores

- Solución al segundo problema de lectores y escritores; "preferencia escritores".

```
semaphore rmutex = 1;
semaphore wmutex = 1;
semaphore readTry = 1;
semaphore resource = 1;
int read_count = 0;
int write_count = 0;
```

- Variables `read_count/write_count`: cantidad de hilos que están leyendo/escribiendo el objeto actualmente.
- Semáforo `rmutex/wmutex`: para la exclusión mutua de la variable `read_count/write_count`.
- Semáforo `resource`: exclusión mutua para escritores y primer lector.
- Semáforo `readTry`: otorga la preferencia a escritores ya que cada lector debe bloquearlo y liberarlo; sin embargo, solo el primer escritor debe esperarlo y solo el último escritor lo libera.

1

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el segundo problema de los lectores-escritores

Escritor

```
while(true) {
    wait(wmutex);
    write_count++;
    if (write_count == 1)
        wait(readTry);
    signal(wmutex);
    wait(resource);

    // Escritura

    signal(resource);
    wait(wmutex);
    write_count--;
    if (write_count == 0)
        signal(readTry);
    signal(wmutex);
}
```

Lector

```
while(true) {
    wait(readTry);
    wait(rmutex);
    read_count++;
    if (read_count == 1)
        wait(resource);
    signal(rmutex);
    signal(readTry);

    // Lectura

    wait(rmutex);
    read_count--;
    if (read_count == 0)
        signal(resource);
    signal(rmutex);
}
```

1

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el tercer problema de los lectores-escritores

- Solución al tercer problema de lectores y escritores; "solución justa para lectores y escritores".
 - Se agrega la restricción de no permitir que ningún hilo caiga en inanición; es decir, la acción de obtener el *lock* de los datos compartidos siempre terminará en un período de tiempo acotado.

```
semaphore resource_Access = 1;  
semaphore rmutex = 1;  
semaphore service_Queue = 1;  
int read_count = 0;
```

- Variable `read_count`: cantidad de lectores accediendo actualmente al recurso.
- Semáforo `resource_Access`: control de acceso lectura/escritura al recurso.
- Semáforo `rmutex`: para sincronizar cambios en la variable compartida `read_count`.
- Semáforo `service_Queue`: aplica justicia, conservando el orden de las solicitudes (los *signals* deben ser *FIFO*)

1

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el tercer problema de los lectores-escritores

- Construcción, primer paso: Justicia.

Escritor	Lector
<pre>while(true) { wait(service_Queue); signal(service_Queue); }</pre>	<pre>while(true){ wait(service_Queue); signal(service_Queue); }</pre>

- Como primer paso se necesita crear una cola justa entre lectores y escritores para evitar la inanición (*starvation*).
- Para lograr eso, se utiliza el semáforo llamado `service_Queue` que dará el orden de llegada.
- Cualquier hilo que solicite acceso al recurso tomará este semáforo y lo liberará tan pronto como ese hilo obtenga acceso al recurso.

1.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el tercer problema de los lectores-escritores

- Construcción, segundo paso: código del escritor (el más sencillo de ambos).

```
Escritor
while(true) {
    wait(service_Queue);
    wait(resource_Access);
    signal(service_Queue);

    // Escritura
    signal(resource_Access);
}
```

```
Lector
while(true){
    wait(service_Queue);
    signal(service_Queue);
}
```

- El escritor debe tener acceso exclusivo al recurso.
- El escritor solicitará el semáforo `resource_Access` antes de modificar el recurso.

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el tercer problema de los lectores-escritores

- Construcción, tercer paso: final.
- Múltiples lectores acceden simultáneamente al recurso.
- El primer lector que tenga acceso al recurso lo bloqueara, así ningún escritor acceda al mismo tiempo.
- El último lector liberará el recurso.

```
Escritor
while(true) {
    wait(service_Queue);
    wait(resource_Access);
    signal(service_Queue);
    // Escritura
    signal(resource_Access);
}
```

```
Lector
while(true){
    wait(service_Queue);
    wait(rmutex);
    if (read_count == 0)
        wait(resource_Access);
    read_count++;
    signal(service_Queue);
    signal(rmutex);

    // Lectura
    wait(rmutex);
    read_count--;
    if (read_count == 0)
        signal(resource_Access);
    signal(rmutex);
}
```

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: el problema de los lectores-escritores

- En algunos sistemas los problema de lectores y escritores y sus soluciones se han generalizado para proporcionar *locks* de lector y escritor.
- La adquisición de un *lock* de lector-escritor requiere especificar su modo: acceso de lectura o escritura.
- Cuando un proceso solo desea leer datos compartidos, solicita el *lock* en modo lectura.
- Un proceso que desee modificar los datos compartidos debe solicitar el *lock* en modo escritura.
- Está permitido que múltiples procesos adquieran simultáneamente un *lock* lector-escritor en modo de lectura, pero solo un proceso puede adquirir el bloqueo para escribir, ya que se requiere acceso exclusivo para los escritores.

SOYD 2020 Gustavo C. Distel

1

Ejemplos de sincronización

Problemas clásicos: el problema de los lectores-escritores

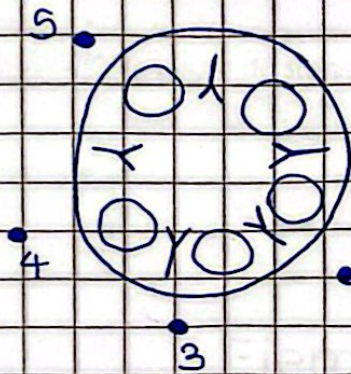
- Los *locks* de lector-escritor son útiles en las siguientes situaciones:
 - Aplicaciones en las cuales es fácil identificar qué procesos solo leen datos compartidos y cuales solo escriben datos compartidos.
 - En aplicaciones que tienen más lectores que escritores.
 - Esto se debe a que los *locks* de lector-escritor generalmente requieren más sobrecarga que los semáforos o *locks* de exclusión mutua.
 - La mayor concurrencia de permitir múltiples lectores compensa la sobrecarga involucrada en la configuración de *locks* lector-escritor.

SOYD 2020 Gustavo C. Distel

1

Fabula 3: Fabula de los alumnos gladiadores

• un profesor no encuentra nada más que implementar la selección natural en el curso y somete a sus alumnos a este proceso para ver si aprueban. Para esto el profesor se coloca en la ventana de la facultad y sus estudiantes en el patio de ahí sentados en círculos cerrados de 5 personas. En cada círculo se pone una mesa entre cada uno de los estudiantes, llega un momento en que se levantan se ponen a pelear pero para eso el estudiante necesita una maza en cada mano. El sobreviviente final pasa el ramer / Fabula de filósofos comensales



- todos comen en un periodo de tiempo.
- si todos toman un tenedor a la vez se produce un deadlock, ya que todos quedan a la espera de que se desocupe 1.
- tomar en una SC ambos tenedores a la vez, reviso a los vecinos, si están comiendo ambos tenedores ocupados, uno está desocupado.
- variable global, arreglo de todos de 0 y 1 para controlar quienes comieron y quienes no.

DEADLOCKS..

- situación en la que dos o más threads quedan bloqueados para siempre, esperándose entre ellos.
- se dan en presencia de las sigles condiciones: exclusión mutua, captura y espera, espera circular etc.
- Para mejorar los deadlocks, se tiene como estrategia: prevención/evitamiento de deadlocks, detección y recuperación, ignorar el problema.

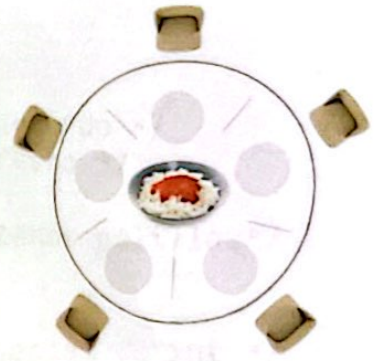
LIVELLOCKS.

- los procesos no progresan, aunque los estados están cambiando constantemente.
- caso especial de inanición.
- son resultado de un intento de manejo de deadlock.

Ejemplos de sincronización

Problemas clásicos: El problema de la cena de los filósofos (*The Dining-Philosophers Problem*)

- Consideremos cinco filósofos que pasan sus vidas pensando y comiendo.
- Los filósofos comparten una mesa circular con cinco sillas, cada una de las cuales pertenece a un filósofo. En el centro hay un tazón de arroz, y alrededor se colocan con cinco palillos.
- De vez en cuando, un filósofo tiene hambre y trata de recoger los dos palillos que están más cerca (los palillos que se encuentran entre él y sus vecinos izquierdo y derecho).
- Un filósofo puede recoger solo un palillo a la vez. Obviamente, no puede recoger un palillo que ya está en la mano de un vecino.
- Cuando un filósofo está hambriento y tiene dos palillos, come sin soltarlos. Cuando termina de comer, deja los palillos y comienza a pensar de nuevo.
- Esta es una representación simple de la necesidad de asignar varios recursos de una manera libre de *deadlocks* y libre de *starvation*.



1'

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: El problema de la cena de los filósofos

- Una solución simple es representar cada palillo con un semáforo.
- Un filósofo intenta agarrar un palillo ejecutando una operación `wait()` y suelta sus palillos ejecutando la operación `signal()`.

Datos compartidos

```
semaphore chopstick[5] = 1;
```

- Aunque esta solución garantiza que no haya dos vecinos comiendo simultáneamente, podría causar *deadlock*:
 - Si los cinco filósofos tienen hambre al mismo tiempo y cada uno agarra su palillo izquierdo todos los elementos del palillo serán = 0.
 - Cuando cada filósofo intente agarrar su palillo derecho, esperará por siempre.

```
filósofo i
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    /* eat for a while */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* think for awhile */
    . . .
}
```

2'

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: El problema de la cena de los filósofos

- Solución Tanenbaum:

```
#define N 5 /* número de filósofos */
#define LEFT (i+N-1) % N /* número del i-ésimo vecino izquierdo */
#define RIGHT (i+1) % N /* número del i-ésimo vecino derecho */
#define THINKING 0 /* filósofo está pensando */
#define HUNGRY 1 /* filósofo está tratando de obtener palillos */
#define EATING 2 /* filósofo está comiendo */

typedef int semaphore; /* los semáforos son un tipo especial de int */
int state[N]; /* arreglo para realizar un seguimiento del estado de todos */
semaphore mutex = 1; /* exclusión mutua para regiones críticas */
semaphore s[N]; /* un semáforo por filósofo */
```

2

SOYD 2020 Gustavo C. Distel

Ejemplos de sincronización

Problemas clásicos: El problema de la cena de los filósofos

```
void philosopher(int i){
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i){
    wait(&mutex);
    state[i] = HUNGRY;
    test(i);
    signal(&mutex);
    wait(&s[i]);
}

void put_forks(i){
    wait(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(&mutex);
}

void test(i){ /* i: philosopher number, from 0 to N-1 */
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        state[i] = EATING;
        signal(&s[i]);
    }
}
```

Para acceder a state o invocar a test(), un hilo debe mantener el mutex. Por lo tanto, la operación de chequear y actualizar el arreglo es atómica.

2

SOYD 2020 Gustavo C. Distel

• memoria

- objetivo principal S.O. gestionar y administrar los recursos de la memoria principal.
- RAM: (memoria principal): forma de almacenamiento de datos temporal y volátil que se utiliza para cargar y almacenar los programas y datos de ejecución.
- se emplea para cargar y ejecutar programas, almacenamiento de datos e intercambio de datos entre distintos procesos.
- se debe administrar bajo condiciones, en donde se debe asignar y liberar espacio de los programas y datos en ejecución utilizando técnicas de administración de memoria, como paginación y segmentación, para asignar y reutilizar la memoria de manera eficiente.
- se utiliza para ejecutar un proceso, ~~se ejecuta~~ en ese caso lo llamamos desde la memoria.
- mediante la administración de las direcciones virtuales y físicas con el uso del módulo del sistema MMU (Memory Management Unit), el cual se encarga de la conversión de direcciones virtuales a físicas y viceversa gracias a esto se puede saber cuáles son las direcciones de un programa cualquiera.
- Si no queda espacio físico en la RAM, se recurre a tipos de almacenamiento secundario. Se aplica swapping, movidos temporalmente a HDD. Swap-out (RAM $\rightarrow$ HDD), swap-in (HDD $\rightarrow$ RAM).

Fragmentación:

- se produce debido a las inserciones en memoria y que los procesos van abandonando cuando se terminan generando divisiones en la memoria en bloques con tamaños insuficientes para agregar nuevos procesos.
 - $\rightarrow$ fragmentación interna: fragmentación de un bloque de memoria (dentro)
 - $\rightarrow$ fragmentación externa: fragmentación de la memoria en general (entre bloques)
- estrategias para evitar el dinamismo de la memoria.
 - $\rightarrow$ compactación: se agrupan todos los procesos, es el más costoso.
 - $\rightarrow$ First-fit: El proceso entra al primer espacio disponible, es el más rápido.
 - $\rightarrow$ Best-fit: El proceso entra al mejor espacio posible, no muy bueno a largo plazo.
 - $\rightarrow$ Worst-fit: El proceso entra al peor espacio posible, bueno a largo plazo.
- En worst-fit siempre va a quedar espacio, por ende albergar procesos más pequeños, y en best-fit elige el espacio más pequeño, pero puede que no sobre espacio y luego no quede para otros procesos.

- **Segmentación**: proceso de dividir la memoria asignada a un proceso en segmentos más pequeños.
 - persige que la asignación de espacios en memoria sea más fácil.
 - si se pasa en la cantidad de memoria el proceso: no queda espacio físico en la RAM, se recurre a otro tipo de almacenamiento o se aplica swapping.

¿Cómo se implementaría la segmentación entonces? (si parte en 0, se suma normal)

Segment table

Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

- Si buscamos la dirección 100 (virtual):
- Sabemos que está en el segmento Code
- Se calcula como base + dirección: $32768 + 100 = 32868$
- Dirección física: 32868

(se suma la dirección física con la virtual).

(base + dirección virtual = dirección física).

Administración de memoria

¿Cómo se implementaría la segmentación entonces? (si no parte del cero, se obtiene el offset)

Segment table

Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

- Si buscamos la dirección 4200 (virtual):
- Sabemos que está en el segmento Heap
- Se calcula como base + dirección: $34816 + 4200 = 39016$
- Dirección física: 39016 → Segmentation Fault!!

→ error de segmentación.

Segment table

Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

- Si buscamos la dirección 4200 (virtual):
- Sabemos que está en el segmento Heap Offset dentro de Heap: $4200 - 4096 = 104$ → ese es el offset: $4 \cdot 1024$
- Se calcula como base + dirección: $34816 + 104 = 34920$
- Dirección física: 34920

(dirección virtual - offset = resultado)

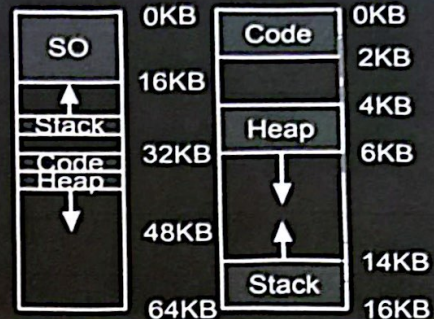
(base + resultado = dirección física)

- ¿Y en este caso?

Segment table

Segment	Base	Size
code	32768 (32KB)	2048 (2KB)
heap	34816 (34KB)	2048 (2KB)
stack	28672 (28KB)	2048 (2KB)

- Si buscamos la dirección 16300 (virtual):
- Segmento Stack (segmento 2) Base + offset: $28672 + 4012 = 32684$
- Dirección física: **32684**

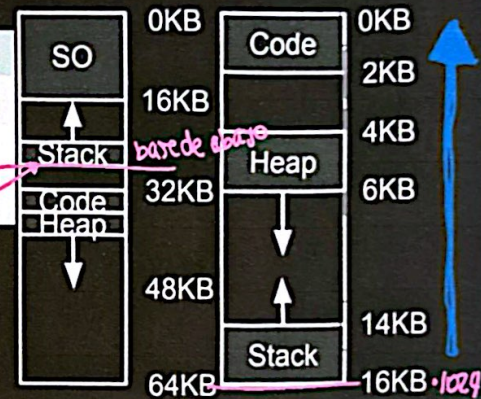


- Debemos considerar lo siguiente:

Segment table

Segment	Base	Size	Up
code	32768 (32KB)	2048 (2KB)	1 ↓
heap	34816 (34KB)	2048 (2KB)	1 ↓
stack	28672 (28KB)	2048 (2KB)	0 ↑

- Si buscamos la dirección 16300 (virtual): Segmento Stack (segmento 2)
- Offset: complemento de 4096 (4KB máximo de offset):
- Dirección física: 28588



$$16.300 - 16.384 = -84$$

y eso se suma con la base de stack.

$$28.672 - 84 = 28.588$$

(dirección virtual - offset (invertido) = resultado.)

(base + resultado = dirección física)

Paginación

- segmentos del mismo tamaño y se denominan como páginas, estas se ubican dentro del espacio virtual.
- Para la parte física se denominan frames, relación entre espacios 1 a 1, o sea que ambos (frames y páginas) tienen el mismo tamaño.
- La MMU lleva un registro de la asignación de páginas a procesos y se encarga de la traducción de página virtual a marco físico.

Ej:

PAGO	0B	0B
	16B	16B
PAG 1		32B
	32B	48B
PAG 2		64B
	48B	80B
PAG 3		96B
	64B	112B
VIRTUAL		128B

FÍSICA.

• de este ejemplo tenemos:

- espacio virtual 64B, espacio físico 128B, páginas 16B

Por ende obtenemos:

- direcciones posibles de 1 página: $16 = 2^4 = 4$ bits
- espacio virtual: $64B = 2^6 = 6$ bits
- son 4 bits de offset, y los dos restantes indican el n.º páginas $2^2 = 4$
- espacio físico: $128B = 2^7 = 7$ bits
- por ende son 4 bits de offset, los tres restantes para los frames $2^3 = 8$.

Políticas de swapping

- swapping: intercambio en la ubicación física donde se encuentran los datos que se usan por el sistema operativo.
- se debe identificar cual página está presente en memoria y cual no (present bit).
- **Page Fault**: ocurre cuando se busca una página y esta no se encuentra, esto solo activa una recuperación de la página a través del S.O. y esta recuperación pasa de memoria principal a memoria secundaria.

→ Para evitar los **Page Fault** usamos algoritmos

1. MIN: predice el futuro, no es implementable.

2. FIFO: hace un swap de la página que lleva más tiempo en memoria

3. Random: elige una página al azar.

4. LRU: reemplaza la última página que se ha llamado.

5. Reloj: da segundos oportunidades, las páginas tienen un bit de referencia, si es 1 la página no corresponde a la que se busca y este cambia a 0 y se lee la otra página en memoria, si al leer la página su bit es 0, se intercambia.

PROARTE®

$$1KB = 2^{10}$$

$$1MB = 2^{20}$$

$$1GB = 2^{30}$$

$$1TB = 2^{40}$$

¡EJERCICIO Solemne!

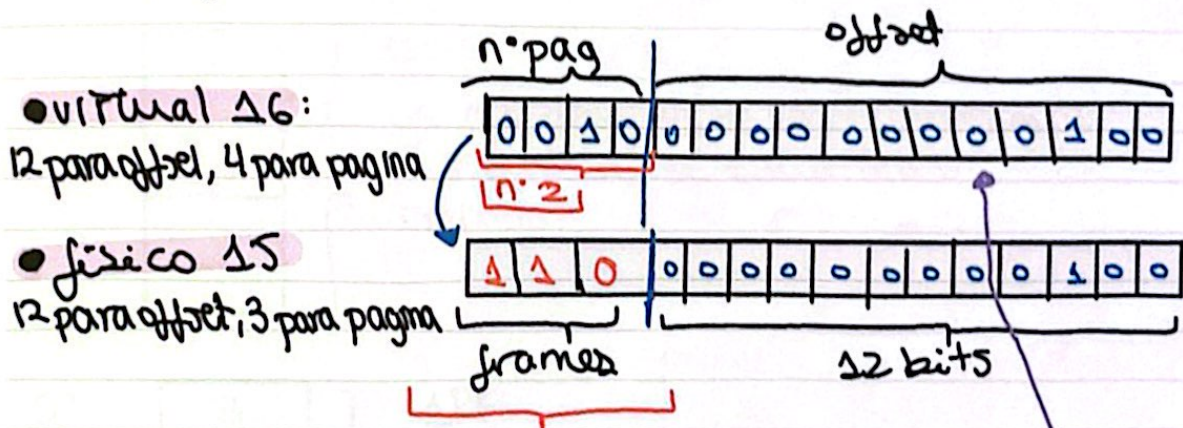
- Trabajen ahora sobre este ejemplo:
 - Páginas de 4KB, dir. virtual 16 bits, dir. física 15 bits
 - Tamaños de espacio virtual y físico y no existen en tamaño de página #Bits de offset, frame y página
 - Transformar dirección virtual 8196

$$1KB \rightarrow 2^{10}$$

$$4KB \rightarrow 2^{12}$$

cada Byte es una dirección.

• cada pag es de 4KB. $\rightarrow 2^{12}$ Bytes direcciones por pagina.



se debe ver a que frame corresponde la pagina.

numero a transformar es 8196 en binario ya que esta en direccion virtual.

y esto para este caso se representa 8196 en 16 bits

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

• Los bits de la arquitectura de una maquina marcan el direccionamiento posible de la memoria principal.

ALGORITMO OPTIMO (Min): (ningun algoritmo puede ser menor a este)

- es optimo pero no se puede realizar, ya que se debe conocer el futuro y así cambiar la pagina que se usa en un tiempo mayor.

Ej: se tiene la siguiente secuencia

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2.

y se tienen 4 espacios de memoria:

1°

7
0
1
2

4 PF,

ya que ninguna de las 4 primeras de la secuencia se encuentra en memoria.

2°

3
0
1
2

1 PF

se cambia el 7 ya que se conoce que este no volverá a aparecer por ende se cambia por el 3.

3°

3
0
4
2

1 PF

se cambia el 1 ya que se conoce que no volverá a aparecer, mientras que el 0, 3, 2 si.

total PF = 6. //

ALGORITMO FIFO:

se elige la pagina que tiene mas tiempo en memoria.

Ej: Se tiene la secuencia.

0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1
y se tienen 3 frames de memoria.

0
1
2

3PF

3
1
2

1PF

3
0
2

1PF

3
0
1

1PF

2
0
1

PTF

Para este algoritmo se va cambiando en orden desde arriba hacia abajo, una vez completadas las posiciones de los frames.

ALGORITMO RANDOM:

se elige alguna pagina aleatoria en memoria para reemplazar. Funciona mejor que FIFO.

ALGORITMO LRU: (Least Recent Used).
se reemplaza la ultima pagina que se ha llamado.

← miro hacia atras y saco el mas antiguo.

Problemas de este algoritmo es que es caro.

Ej: se tiene la siguiente secuencia.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 y
4 espacios en memoria.

no mas estan
todas ya en
memoria.

7
0
1
2

4PF

3
0
1
2

1PF

3
0
4
2

1PF

- se va cambiando el que menos se ha usado
si se meten todos los datos a la memoria, el primero que se metio fue el que menos se ha usado y asi sucesivamente

Ej: se tiene 1,2,3, llega 4 queda 4,2,3

Ej: se tiene 5,1,4,3, llega 6 no hago nada, luego llega 6
me quedaria 5,6,4,3.

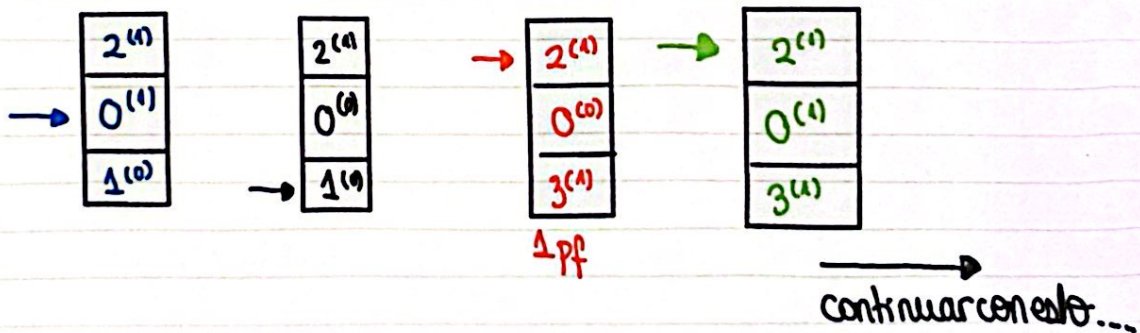
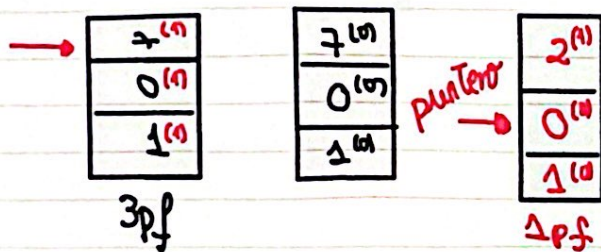
Ej: tengo 5,1,4,3, llega un 5 no hago nada, luego llega un 6
me queda 5,6,4,3, me vuelve a llegar un 6 no hago nada y
me llega un 1, me quedaria 5,6,1,3.

ALGORITMO DEL RELOJ:

las paginas tienen un bit de referencia, si el bit de referencia es 1 y la pagina no es la que se busca cambia a 0 y se lee la sgte pagina en memoria. Si al leer la pagina su bit es 0, se intercambia.

Ej: se tiene la siguiente secuencia

7, 0, 1, 2, 0, 3, 0, 4, 2, 4, 0, 3, 2.



• entidades MV

→ VCPU (CPU Virtual), no es un elemento de procesamiento, sino que se refiere al estado de CPU que la VM huésped cree que tiene.

→ El procesamiento sigue haciendo a nivel de CPU física y con la ayuda de su contexto.

→ 2 técnicas que usan las MV:

• **Trap and emulate.** La VM emula la operación en modo kernel, consiste en la definición del modo dual virtual y su vínculo con el modo dual físico. La VM corre sobre el modo usuario de la máquina física, por ende, no puede acceder al modo kernel directamente. Así al entrar en modo kernel virtual e intentar ejecutar una instrucción en modo privilegio, genera un error. Producido este error, se rebota el control al VMX del SO guest, luego se ~~emula~~ emula lo que se intentó ejecutar desde el kernel guest. Completado se rebota el control a la VM.

• **Binary translation.** traduce la operación de modo kernel a una operación segura.

Surge de la pobre separación que existe entre instrucciones privilegiadas y no privilegiadas.

Se establece si una VCPU está en modo usuario, la VM guest puede correr sus instrucciones en CPU física. Si la VCPU está en modo kernel, entonces se hace creer al guest que corre en modo kernel. Las instrucciones en question son examinadas por VMX. Aquellas instrucciones especiales que lo requieren son traducidas a otro conjunto de instrucciones que ejecutan una tarea equivalente.

SCHEDULING (en virtualiz)

• cada CPU física debe actuar como un sistema multi procesador.

• Los threads al ejecutar pueden provenir de los guest como de la VM.

• si hay suficientes CPUs físicas para atender los requerimientos de la VM, entonces la CPU trabaja como si estuviera dedicada totalmente a esa VM.

Para este caso, se debe repartir el uso de la CPU entre los threads de la VM y la misma VM requiere también de ciclos de CPU.

• **overcommitment:** escenario en que los guest están configurados de forma que esperan más CPUs que las que tiene el sistema.

Gestión de memoria:

• cuando se puede suscitar la sobrecarga de memoria, sucede que ~~se~~ se le asigne más memoria RAM a las VM de la que el sistema físico realmente tiene.

- **mecanismos de gestión:**

- Tablas de paginación doble

- **balloon memory manager**: pseudo-driver que instala el VMM en el guest para que el kernel lo reconozca como un dispositivo y tenga acceso directo a modo kernel.

Usa una técnica que fija marcos de memoria y con ello juega con las expectativas de la VM quien utiliza su propio sistema de gestión de memoria.

Así la gestión comprende distintas opciones de mecanismos:

- * técnica de indexación en que si un marco ha sido cargado más de una vez, se eliminan las copias y se referencia a una única copia.

Aspectos más importantes a considerar:

- **I/O**: Los hipervisores son los encargados y la tienen algo más fácil.

Al ocurrir que los drivers de I/O establecen una conexión con el SO, provee una flexibilidad que es aprovechada por el entorno de virtualización.

- **administración de almacenamiento:**

depende del tipo de hipervisor:

- **Tipo 1 (HW)**: almacenan la info del HDD raíz del guest en un conjunto de archivos en el VMM.

- **Tipo 2 (SW)**: almacenan la misma info, pero en el SO host.