Sistemas Operativos

Tutorías Ex. de Título 2024-2

Universidad Diego Portales
Prof. Víctor Reyes R.

Objetivos

- Gestión de procesos y scheduling
- Técnicas de sincronización
- Gestión de Memoria

Procesos

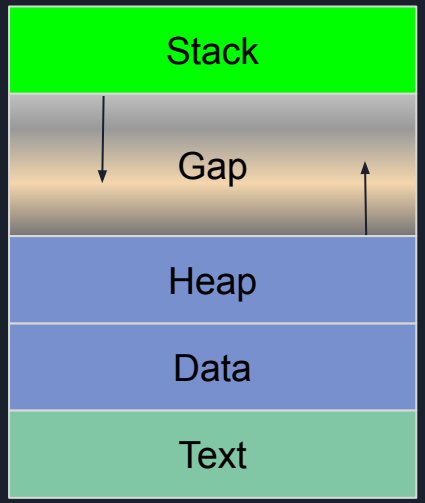
- Concepto central en SO ¿Por qué?
- El objetivo de un computador es el de operar sobre datos y entregar resultados o efectuar acciones.
- Proceso: programa en ejecución (junto con todos los recursos y datos asociados)

Procesos

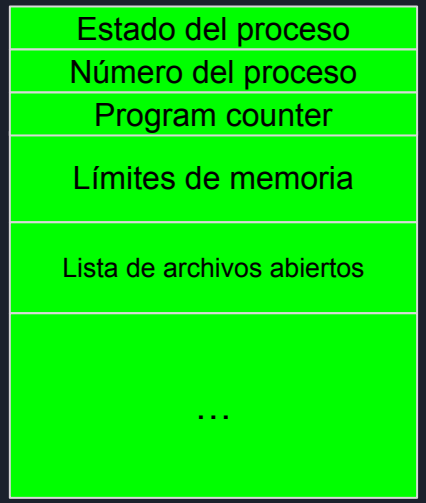
- Al usar su computador: ¿Únicamente trabajan con un solo proceso?
- Uno sabe que lo anterior no es verdad (chequear top, administrador de tareas)
- ¿No se supone que hay una única CPU? No hace demasiado que las CPU son multicore, ¿Cómo se hacía antes?, ¿Qué implica el multicore?

Procesos

- Datos del programa y estado de actividad



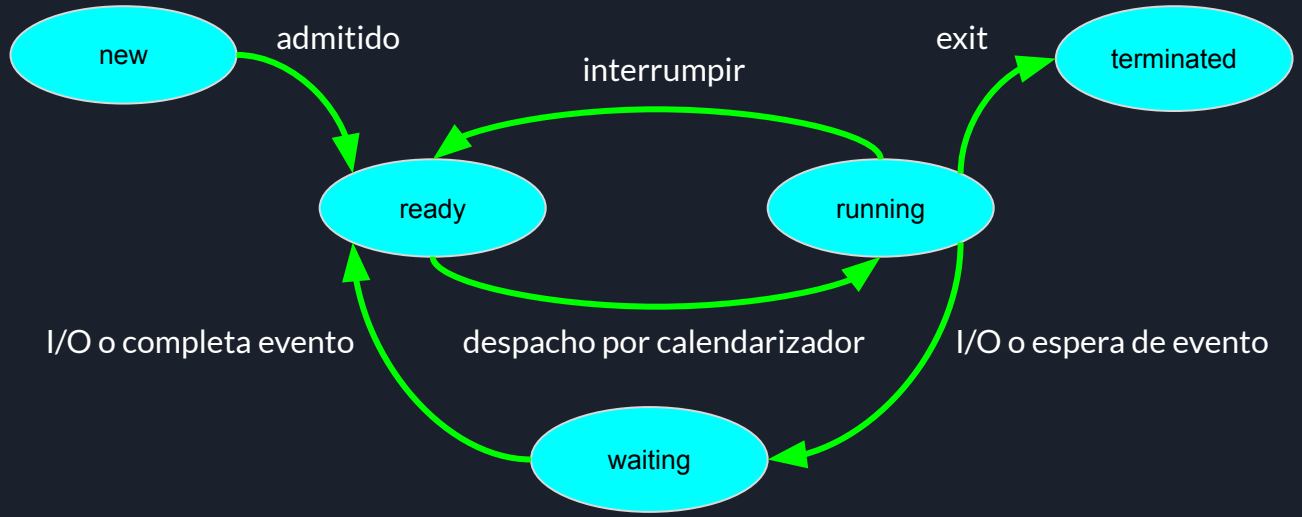
Representación de un proceso en memoria



Process Control Block (PCB)

Procesos

- Datos del programa y estado de actividad



Grafo de estados de un proceso

Procesos

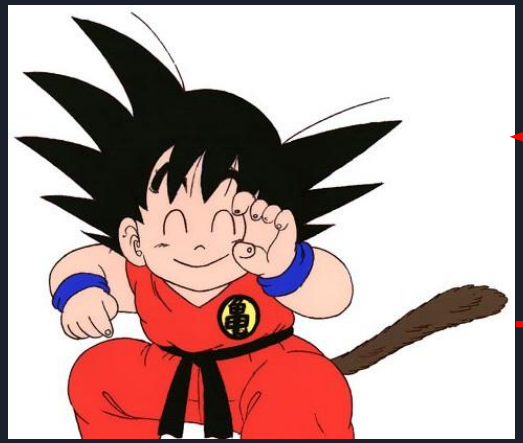
- ¿Cómo se gestionan dichos procesos?

  Varias colas de procesos intervienen, las más importantes son:
  - Job queue (procesos que entran al sistema)
  - Ready queue (procesos listos y en RAM)

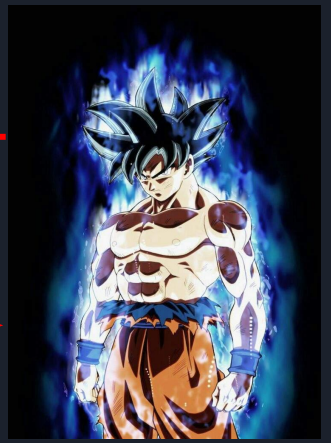
- Los estados de los procesos se rigen de acuerdo al grafo mostrado previamente.

Procesos

- Los programas funcionan en **modo dual**



syscalls

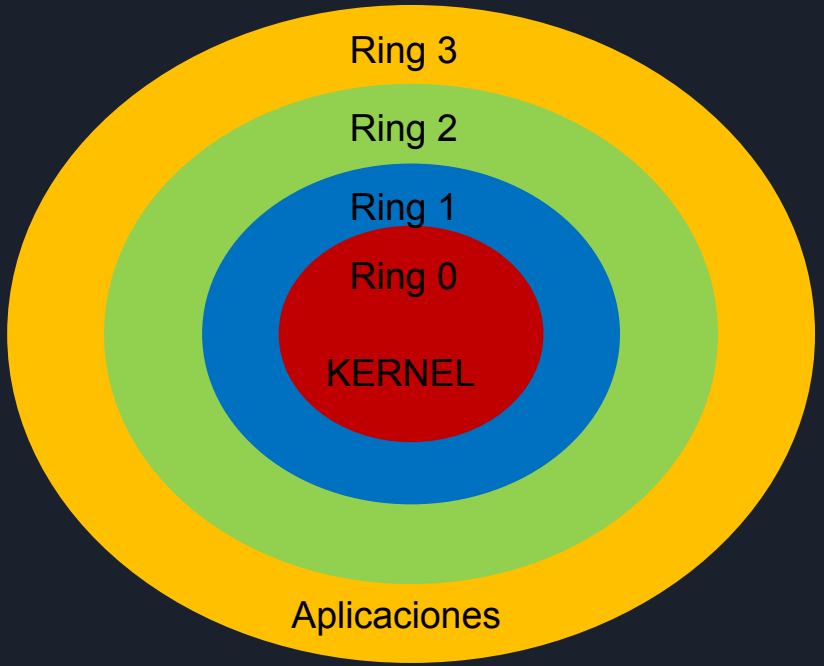


Modo usuario

Modo kernel

Procesos

- El SO tiene varios niveles de protección



Procesos

- ¿Qué es una syscall? Método utilizado por los programas de aplicación para comunicarse con el kernel.
- Punto de enlace entre el modo usuario y el modo kernel.
- Proporciona una interfaz entre un proceso y el sistema operativo para permitir que los procesos a nivel de usuario soliciten servicios del sistema operativo.

Syscalls ejemplos (Unix)

- 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().

- Linux: http://man7.org/linux/man-pages/man2/syscalls.2.html
- MacOS: https://jameshfisher.com/2017/01/31/macos-system-calls.html
- Windows: https://j00ru.vexillium.org/syscalls/nt/32/

Syscalls ejemplos (Unix)

```
int main() {

        pid_t t = fork();
        if (t>0){   //padre
                fork();
        }

        if (t==0){ //hijo
        }
}
```

Procesos

- Interrupciones: son señales que indican una necesidad de detener un proceso en ejecución para dar lugar a otro requerimiento que necesita atención inmediata.
- Multitasking -> Cambio de contexto.
- Dos tipos de interrupciones: Interrupciones de HW, Interrupciones de SW (trap)

Procesos

- Para poder manejar las interrupciones y en caso de que lleguen varias concurrentes, existe una cola de interrupciones. Estas interrupciones tienen también un “nivel de prioridad”. Son manejadas por un Interrupt Handler.

- Al espacio de memoria del Interrupt Handler se le denomina Interrupt Vector.

Procesos

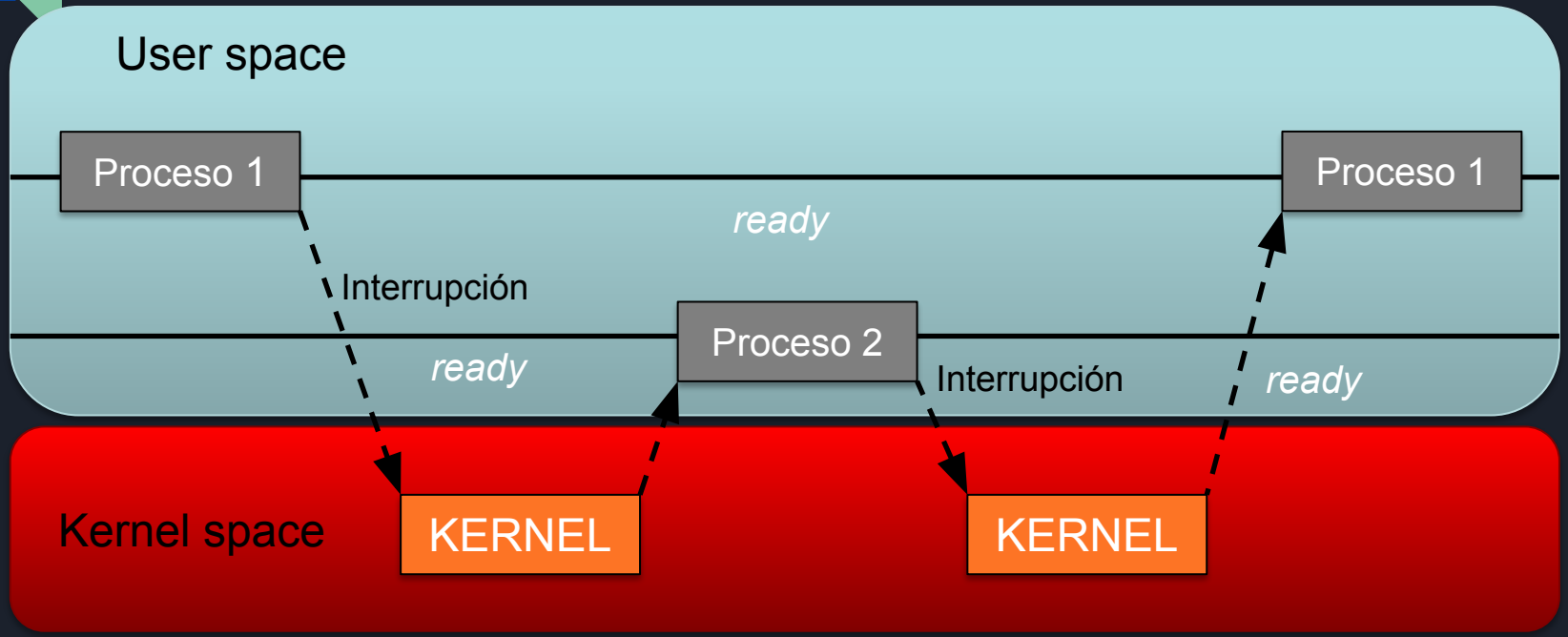
- Las interrupciones de HW son generadas por dispositivos I/O al terminar su operación.
- Las interrupciones de SW también se generan al finalizar la ejecución de programas o bien al requerir tareas por parte del SO.
- Las interrupciones generan cambio de contexto.

Procesos

- Cambios de contexto involucran almacenaje y carga de los datos desde/hacia el PCB.

- Esto se debe a que los procesos que salen de la CPU deben guardar su estado para poder ser restaurado una vez que vuelvan a ser ejecutados en CPU.

Procesos



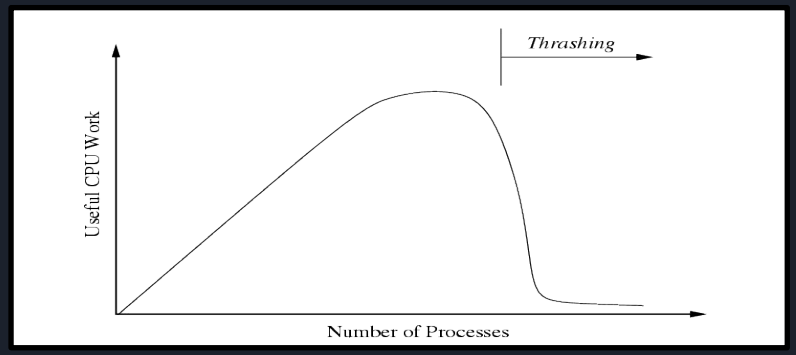
Almacena estado del Proceso 1 en PCB1 y recupera estado de Proceso 2 desde PCB2

Almacena estado del Proceso 2 en PCB2 y recupera estado de Proceso 1 desde PCB1

Procesos

- El scheduling es importante para poder organizar la multiprogramación, sin embargo puede irse para la otra punta…

- Y si ocurre mucho scheduling y cambio de contexto? → Thrashing



Procesos

- Hay dos tipos principales de tareas a las que están asociados los procesos:
  - Uso de CPU
  - Espera de I/O

- Generalmente los procesos alternan entre esas dos fases y predomina alguna.

- El tiempo de espera por I/O es tiempo que no se usa la CPU→Necesitamos usarla al 100%!!!

Procesos

- Así entonces, se infiere que la decisión del scheduler está gobernada por un algoritmo.

- Existen varios de ellos, y se clasifican a su vez por las dimensiones según las cuales toman sus decisiones y/u operan.

Procesos

- Scheduling expropiativo
- Scheduling no-expropiativo
- Batch scheduling
- Interactive scheduling
- Real-time scheduling

En todos los casos, se debe buscar que todo proceso tenga oportunidad de utilizar la CPU durante un periodo razonable.

Procesos

- A continuación se describen algunos algoritmos de scheduling de los tipos mencionados:
  - First come first served
  - Shortest job first
  - Round-robin
  - Priority
  - Earliest deadline first

First come first served, Shortest job first → Batch

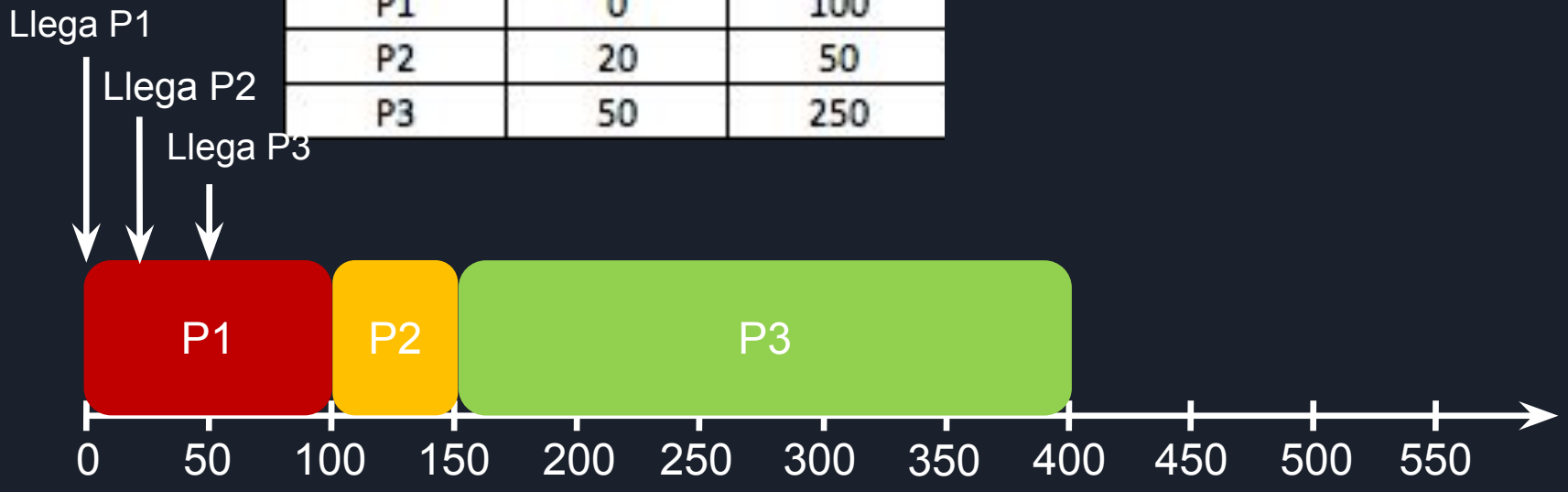
Round-robin, Priority → Interactive

Earliest deadline first → Real-time

Procesos

- First come first served (FCFS)

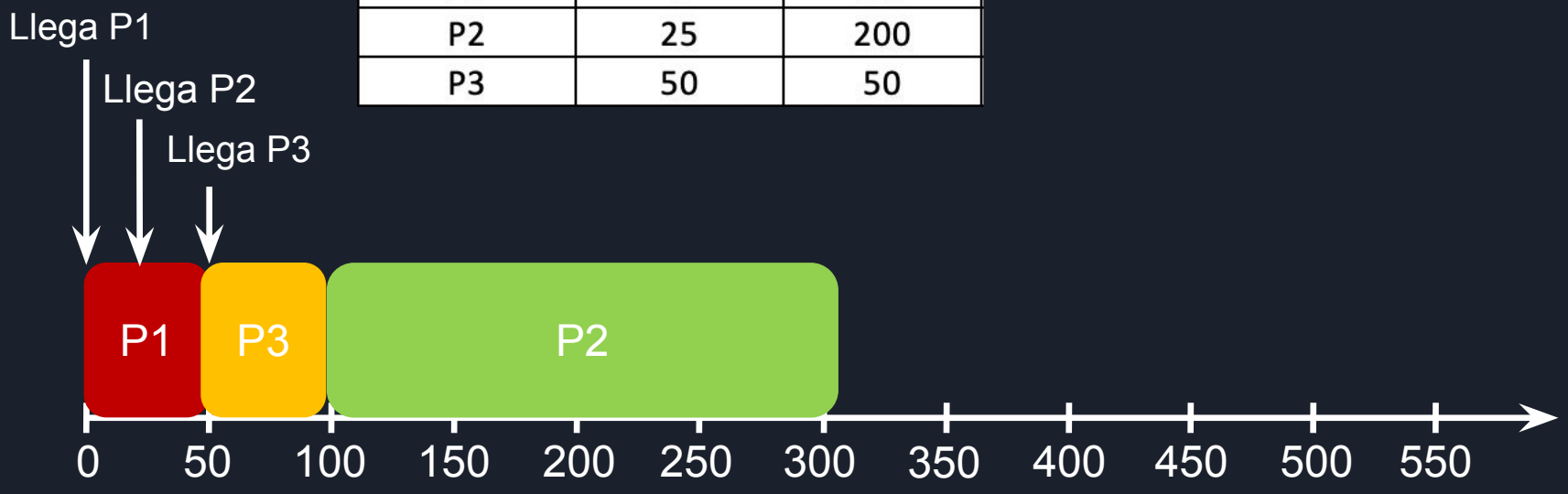
Process	Arrival time	CPU burst	End time	Turnaround
P1	0	100		
P2	20	50		
P3	50	250		



Procesos

- Shortest job first (SJF)

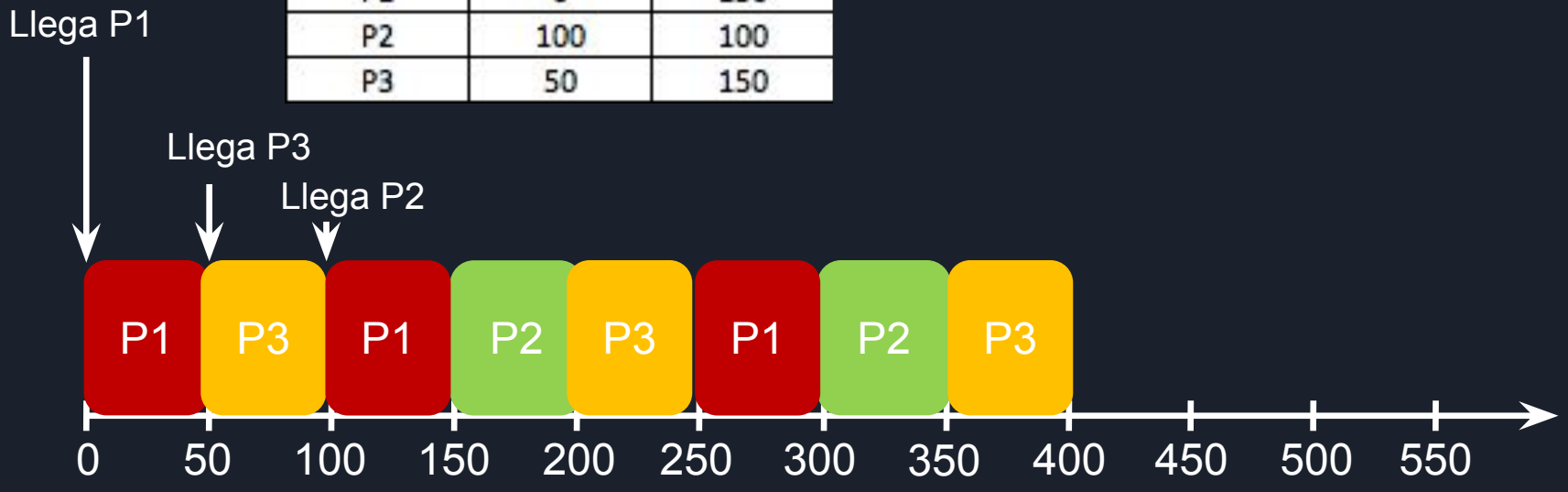
Process	Arrival Time	CPU Burst	End Time	Turnaround
P1	0	50		
P2	25	200		
P3	50	50		



Procesos

- Round-Robin (q=50)

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time
P1	0	150				
P2	100	100				
P3	50	150				

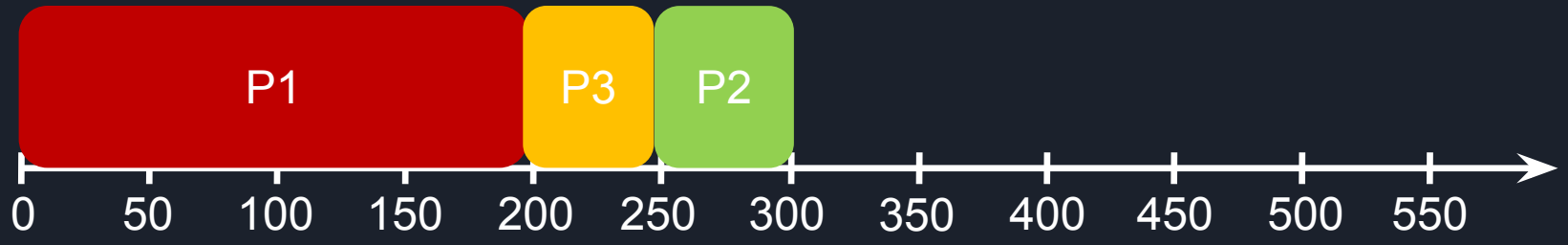


Procesos

- Priority scheduling

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Priority
P1	0	200					63
P2	0.1	50					24
P3	0.2	50					50

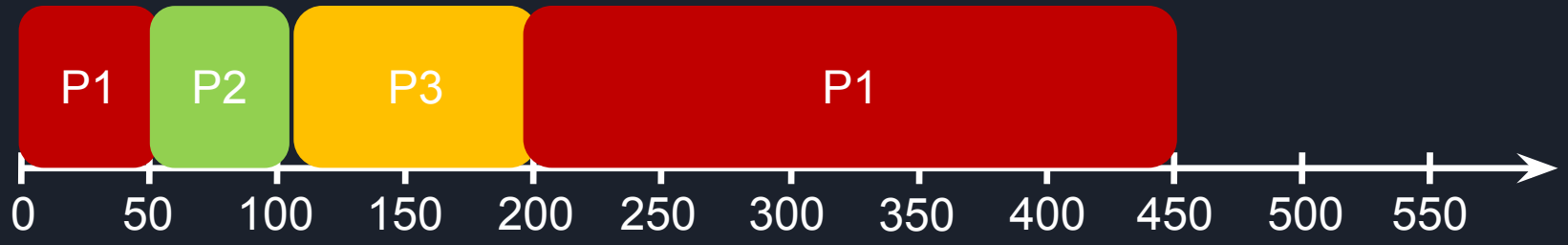
http://osr507doc.xinuos.com/en/PERFORM/calc_proc_priorities.html



Procesos

- Earliest deadline first

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

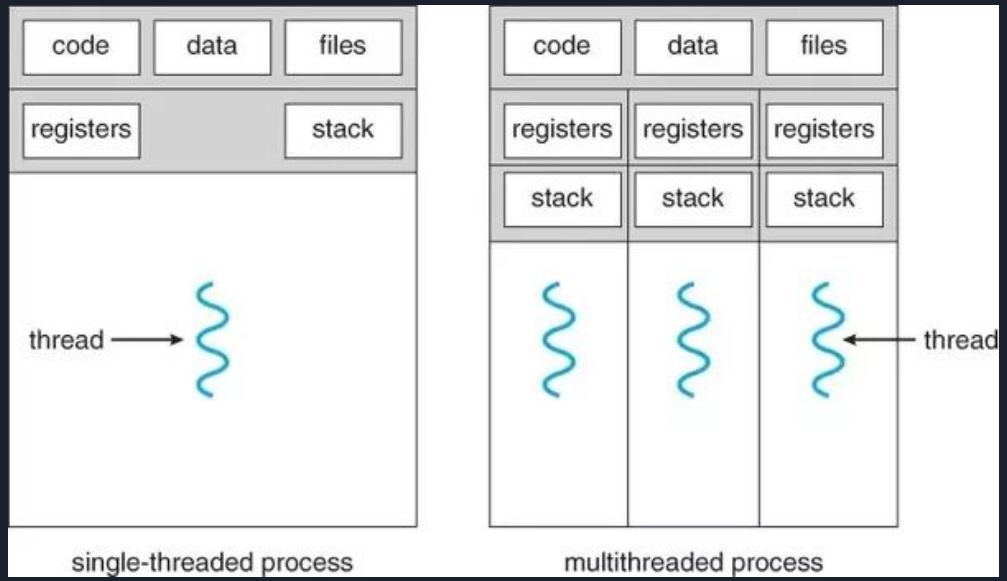


Procesos

- Threads son procesos livianos que, al igual que los procesos, requieren de datos específicos (PCB) para ejecutar.
- ¿Diferencias?
  - Un thread siempre está asociado a un proceso y efectúa una tarea específica. Cada proceso puede tener asociados uno o más threads.
  - Las operaciones asociadas a los threads son menos costosas que las de los procesos.
  - Procesos: agrupan recursos y ejecución Threads: unidades de ejecución

Procesos

- Un thread comparte ciertos datos con su proceso padre, pero tiene también datos propios.



Procesos

- Las operaciones típicas (en POSIX threads) que se usan son las siguientes:

  - thread_create()
  - thread_exit()
  - thread_join()
  - thread_yield()

Sincronización

- Al usar threads se presenta un problema: Race Conditions.

- Las 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.

Sincronización

- Supongamos lo siguiente:

```c
void *mythread1(void *arg)
{
    char *letter = arg;
    int i; // stack (privado por thread)
    printf("%s: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
        counter = counter + 1; // compartido: solo uno
    }
    printf("%s: terminado\n", letter);
    return NULL;
}
```

```c
void *mythread2(void *arg)
{
    char *letter = arg;
    int i; // stack (privado por thread)
    printf("%s: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
        counter = counter * 2; // compartido: solo uno
    }
    printf("%s: terminado\n", letter);
    return NULL;
}
```

¿Qué resultado(s) entregaría el programa al imprimir counter (inicialmente con counter 1 y con max = 2, ambas variables globales)?

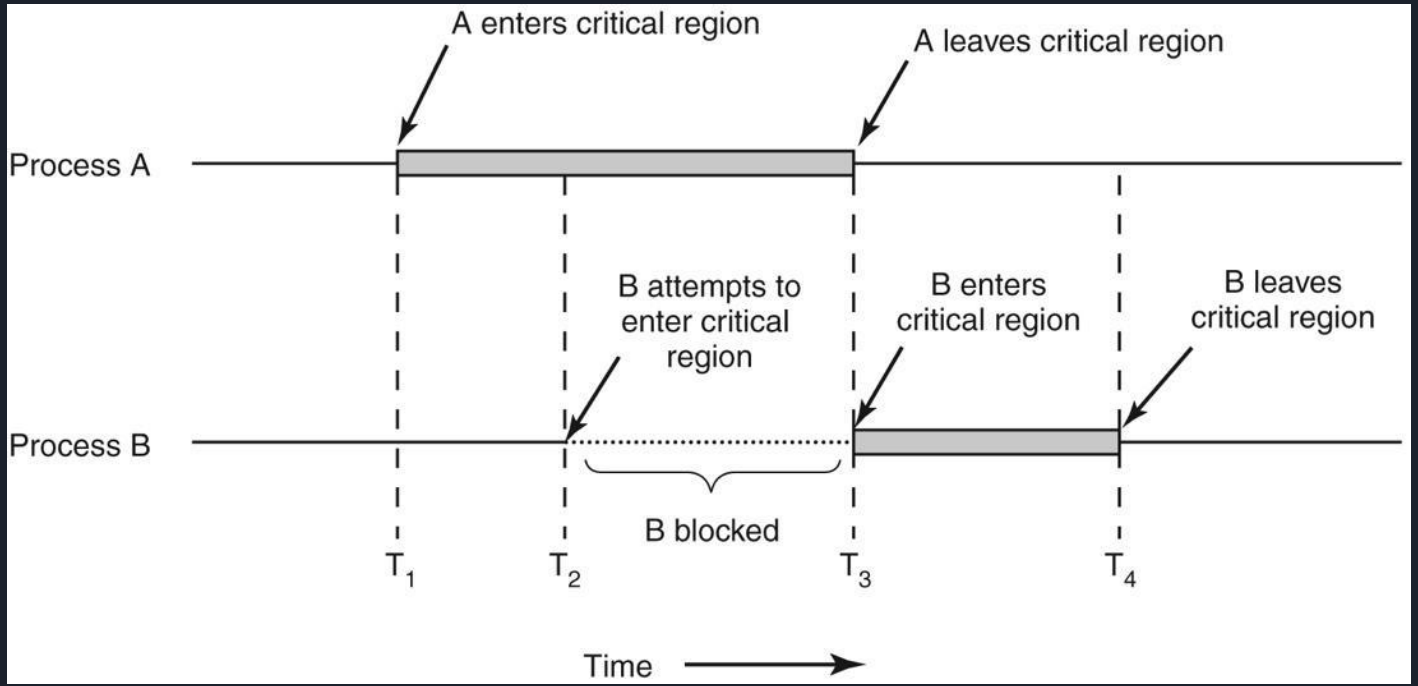
Sincronización

- Sección crítica (SC)
    - Se refiere a una región de memoria de acceso compartido sobre la que trabaja código.
    - Requiere que únicamente un proceso o thread esté trabajando sobre dicha región al mismo tiempo.
    - Se deduce de la necesidad de mecanismos para asegurar que el acceso sea secuencial y único a dicha región de memoria.

Sincronización

- Se desprenden los siguientes conceptos de la definición de SC:
  - **Exclusión mutua**: a lo más un thread/proceso está en su SC.
  - **Progreso**: al menos un thread/proceso puede entrar a su SC. Si no hay threads/procesos en su SC y alguno(s) quiere(n) entrar, deben decidir cuál en un tiempo acotado.
  - **Ausencia de inanición**: Si un thread/proceso quiere entrar a su SC, podrá hacerlo después de un tiempo acotado.

Sincronización



Ejecución deseada con acceso controlado a la SC

Sincronización

- Locks de MUTEX
  - Se abstrae a una variable que garantiza la atomicidad de su manejo y exclusión mutua de acceso a la SC.

  - Usa dos primitivas (que pueden variar en su nombre):
    - Acquire
    - Release

Sincronización

```
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;
}
```

Ejemplos completos en mutex_mal.c y mutex_bien.c

Sincronización

- Semáforos
  - Son mecanismos de sincronización inventados por Edsger Dijkstra (ese mismo, el del algoritmo)
  - En vez de tratarse de un lock que restringe el acceso de un único thread a una SC, permite que accedan N (> 0, con N limitado) threads a la SC.
  - Se implementa mediante un contador que se incrementa/decrementa atómicamente

Sincronización

- Semáforos
  - Para un semáforo S, se implementan dos funciones:
    - P(), wait(): intenta decrementar el valor del contador para simbolizar que ha entrado un thread en la SC.
    - V(), signal(): intenta incrementar el valor del contador. Representa una ubicación libre más para que otro thread pueda entrar a la SC.
  - Cuando el contador del semáforo llega a 0, no pueden entrar más threads a la SC

Sincronización

```c
sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntro a la SC\n");

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

    //signal
    printf("\nSalgo de la SC\n");
    sem_post(&mutex);
}


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;
}
```

Implementación de sincronización de threads usando semáforos

Sincronización

- Variables de condición
  - Son mecanismos de sincronización en que la condición no viene 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 el thread
    - signal() que despierta a un thread bloqueado en caso que lo haya

Sincronización

```c
pthread_mutex_t fill_mutex;
int arr[10];
int flag=0;
pthread_cond_t cond_var=PTHREAD_COND_INITIALIZER;

void *llenar()
{

    int i=0;
    printf("\n Ingrese valores \n");
    for(i=0;i<4;i++)
    {
        scanf("%d",&arr[i]);
    }
    pthread_mutex_lock(&fill_mutex);
    flag = 1;
    pthread_cond_signal(&cond_var);
    pthread_mutex_unlock(&fill_mutex);
    pthread_exit(NULL);
}
void *leer()
{
    int i=0;
    pthread_mutex_lock(&fill_mutex);
    while(!flag)
    {
        pthread_cond_wait(&cond_var,&fill_mutex);
    }
    pthread_mutex_unlock(&fill_mutex);
    printf("Los valores ingresados en el arreglo son:");
    for(i=0;i<4;i++)
    {
        printf("\n %d \n",arr[i]);
    }
    pthread_exit(NULL);
}
```

```c
int main()
{
    pthread_t thread_id,thread_id1;
    int ret;
    void *res;
    ret=pthread_create(&thread_id,NULL,&llenar,NULL);
    ret=pthread_create(&thread_id1,NULL,&leer,NULL);
    printf("\n Threads creados \n");
    pthread_join(thread_id,&res);
    pthread_join(thread_id1,&res);
    return 0;
}
```

Ejemplo de programa que usa threads para leer y escribir en un espacio compartido en forma de arreglo.

Administración de memoria

- Ahora vamos a ver cómo es que el SO gestiona otro de sus recursos centrales, la memoria principal.

  Preguntas:
  - ¿Qué es la memoria principal?
  - ¿Para qué se emplea?
  - ¿Cómo se representa?
  - ¿Bajo qué condiciones debe administrarse?

Administración de memoria

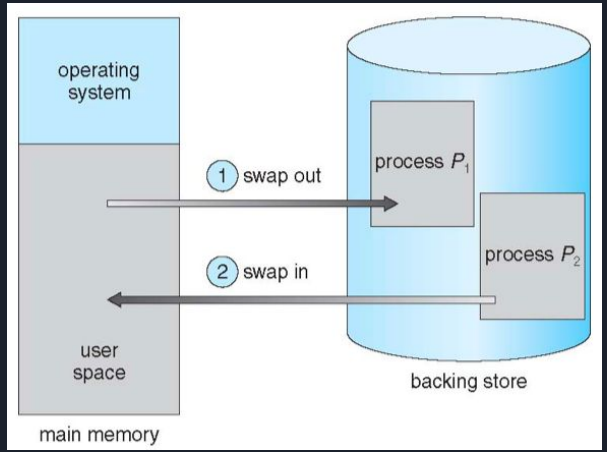
- Así pues, además hay que describir estrategias para ubicar los procesos en memoria al haber dinamismo.
    - Compactación (desfragmentación)
    - First-fit
    - Best-fit
    - Worst-fit
- En este punto surge el concepto de fragmentación externa.

Administración de memoria

- ¿Qué ocurre si se pasa en la cantidad de memoria a emplear en el proceso?
  - No queda entonces espacio físico en la memoria principal (RAM).
  - Debemos recurrir a otro tipo de almacenamiento (secundario).
  - Se aplica swapping, en que ciertos datos de la RAM son movidos a HDD temporalmente.

Administración de memoria

- Dos operaciones de swap:
  - Swap-out (RAM → HDD)
  - Swap-in (HDD → RAM)

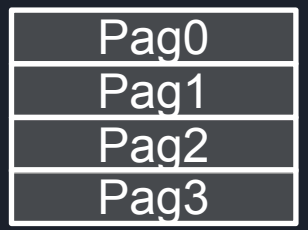


Administración de memoria

- Soluciones: Segmentación - **Paginación**.
- La MMU (Memory Management Unit) lleva registro de la asignación según sea el caso.

Administración de memoria

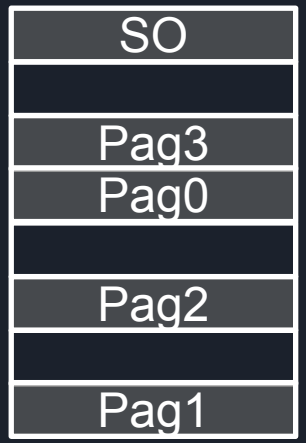
- La relación entre frame y página será 1:1 (tienen el mismo tamaño)
- Veamos el siguiente ejemplo:



0B

64B

Virtual



0B

128B

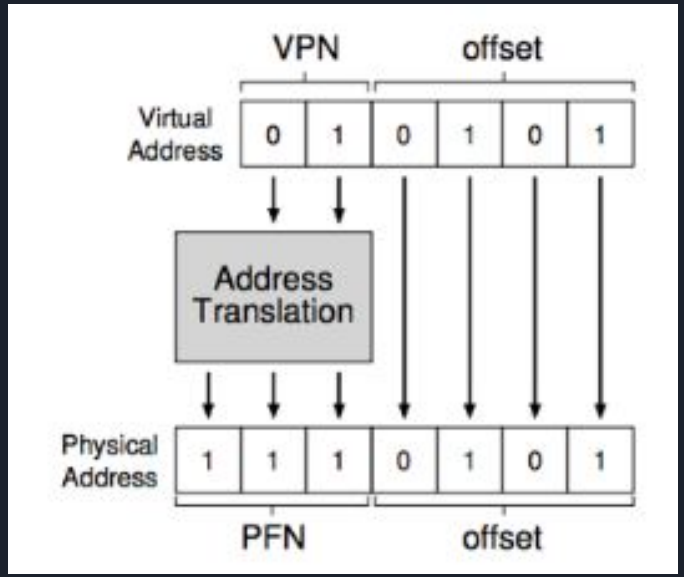
Física

Administración de memoria

- Del ejemplo anterior, calculemos tamaños: Espacio virtual de 64B, físico de 128B y páginas de 16B:
  - Direcciones posibles en una página: 16→4 bits (esto da el mayor offset posible en una página)
  - Espacio virtual de 64B→6 bits.
  - Dado que 4 bits determinan el offset, los 2 restantes indican el número de páginas (hay 4).
  - Espacio físico de 128B → 7 bits. Al igual que para el caso anterior, el offset está determinado por 4 bits, por lo que hay 3 bits para los marcos (son 8).

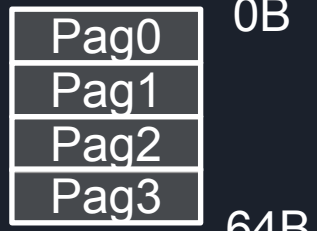
Administración de memoria

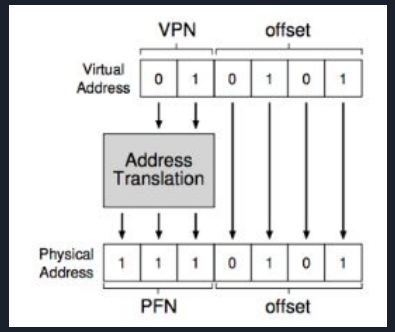
- Se vería así:

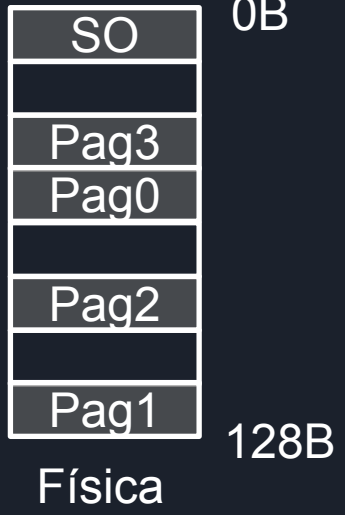


Administración de memoria

- Si ahora hacemos traducción:







Page	Frame
00 (0)	011 (3)
01 (1)	111 (7)
10 (2)	101 (5)
11 (3)	010 (2)

- Tomemos por ejemplo la dirección virtual 21 (010101):
  - Página 1 (01)
  - Offset: 5 (0101)
  - Según la page table, página 1 (01) corresponde a frame 7 (111)
- Luego la dirección física es 117 (1110101)

Administración de memoria

- Por último, recordemos que el swap se refiere a un intercambio en la ubicación (física) donde se encuentran los datos a ser usados por el sistema.

- Surgen un par de preguntas importantes: ¿Qué dato se manda a HDD? ¿Hay alguna política para determinar cuándo, cómo y adónde?

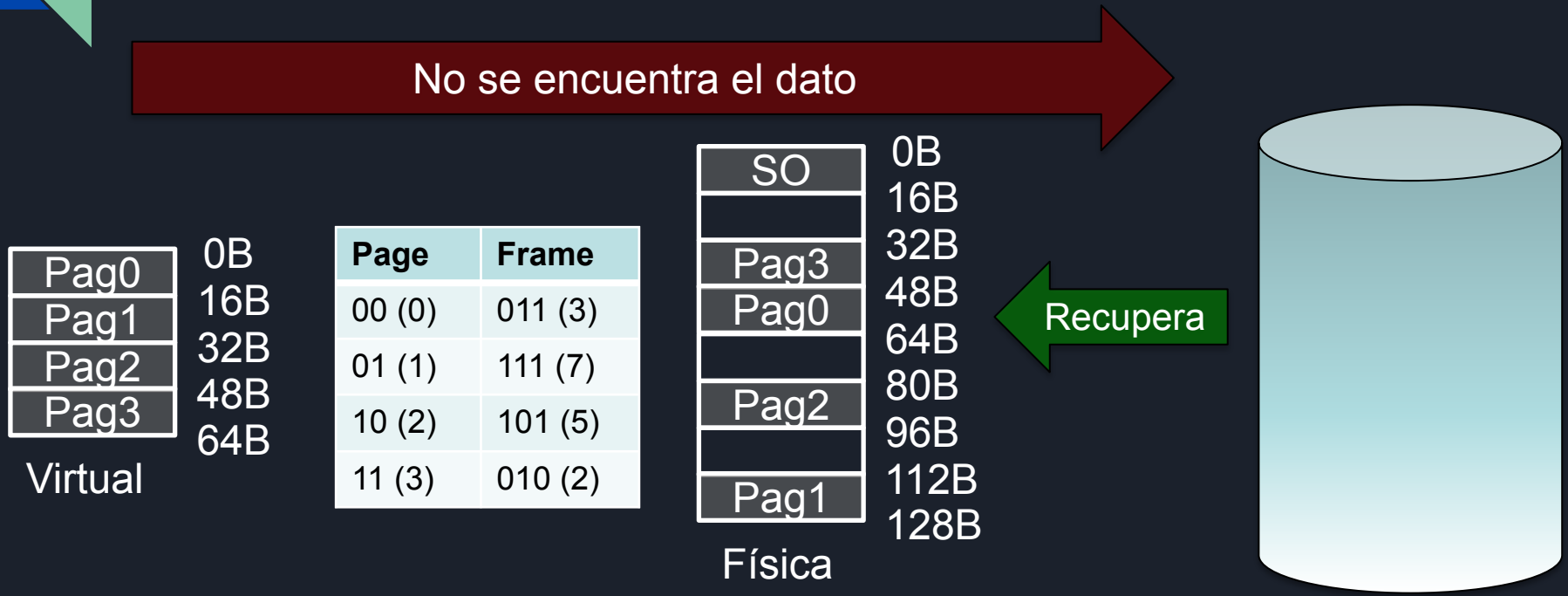
Administración de memoria

- El espacio de memoria que se utiliza es el de swap.
- Se debe tener identificado cuál página está presente en memoria (frame) y cuál no → se traduce a un “present bit”.
- En caso de que se busque la página y ésta no se encuentre en memoria, se genera un Page Fault.

Administración de memoria

- La Page Fault, activa una recuperación de la página a través del SO.
- Esta recuperación pasa de memoria principal a memoria secundaria (HDD).
- Una Page Fault pone en espera al proceso que está ejecutando hasta que se haya resuelto el requerimiento.

Administración de memoria



Administración de memoria

- La ocurrencia de una Page Fault es algo malo… pues implica el acceso a HDD. La idea es evitar al máximo que ocurran.

- ¿Qué pasa si hay un recambio reiterado de las mismas páginas? → RIP ;(

- Para esto, se necesitan algoritmos y políticas que dirijan la dinámica.

Administración de memoria

- Algoritmo óptimo (MIN): Conocer el futuro!!! Cambiar la página que se usa el mayor tiempo después.

- Supongamos la siguiente secuencia de solicitudes:
  7 0 1 2 0 3 0 4 2 3 0 3 2

- Hay 4 espacios de memoria

Administración de memoria

- Contemos los page faults y apliquemos algoritmo (7 0 1 2 0 3 0 4 2 3 0 3 2 y 4 espacios):
  - 4 page faults de inicio (7 0 1 2)
  - 0 page faults del 0
  - 1 page fault del 3 → reemplaza al 7
  - 0 page faults del 0
  - 1 page fault del 4 → reemplaza al 1
  - 0 page faults hasta el final
  - Total: 6 page faults

Administración de memoria

- El problema es que no se sabe la secuencia a priori.
- El algoritmo es óptimo, sí, pero impracticable.
- El objetivo es demarcar un benchmark para comparar otros algoritmos.

Administración de memoria

- Algoritmo FIFO: Se elige la página que lleva más tiempo en memoria.
- Supongamos la secuencia de solicitudes:
  0 1 2 0 1 3 0 3 1 2 1
- Supongamos ahora 3 frames de memoria

Administración de memoria

- Contemos los page faults y apliquemos algoritmo (0 1 2 0 1 3 0 3 1 2 1 y 3 espacios):
  - 3 page faults de inicio (0 1 2)
  - 0 page faults del 0 y del 1
  - 1 page fault del 3 → reemplaza al 0
  - 1 page fault del 0 → reemplaza al 1
  - 0 page faults del 3
  - 1 page fault del 1 → reemplaza al 2
  - 1 page fault del 2 → reemplaza al 3
  - 0 page faults del 1
  - Total: 7 page faults

Administración de memoria

- Algoritmo random: Se elige alguna página aleatoria en memoria para reemplazar.

- Usualmente funciona mejor que FIFO.

Administración de memoria

- Algoritmo LRU (Least Recently Used): Parecido a MIN, pero con datos que sí conocemos.

- Supongamos la misma secuencia de solicitudes: 7 0 1 2 0 3 0 4 2 3 0 3 2

- Suponemos 4 espacios de memoria

Administración de memoria

- Contemos los page faults y apliquemos algoritmo (7 0 1 2 0 3 0 4 2 3 0 3 2 y 4 espacios):
  - 4 page faults de inicio (7 0 1 2)
  - 0 page faults del 0
  - 1 page fault del 3 → reemplaza al 7
  - 0 page faults del 0
  - 1 page fault del 4 → reemplaza al 1
  - 0 page faults hasta el final
  - Total: 6 page faults