**Trabajo Práctico N°2 – ISO.**

1. **Conceptos**
2. Un sistema operativo es un **software** diseñado para controlar y ejecutar otro software y que el mismo utilice el hardware.

* Actúa como intermediario entre el usuario y el hardware.
* Gestiona el hardware
* Brinda comodidad, eficiencia y una interfaz entre las aplicaciones y el hardware
* Controla la ejecución de los **procesos**

1. Los **componentes/aspectos** del hardware que se precisan para cumplir los objetivos de un S.O son:

* **CPU** (Unidad Central de Procesamiento) -> Procesador
  + Es el "cerebro" de la computadora. El SO debe gestionar y asignar tiempo de CPU a los diferentes procesos y programas (planificación de procesos), manejar interrupciones y proporcionar multiprocesamiento
* Memoria **RAM**
  + Es el espacio de trabajo **volátil** donde se cargan los programas y datos en ejecución. Uno de los objetivos clave del SO es la **gestión de memoria**: **asignar** y **liberar** espacio, **proteger** la memoria de un proceso de otro, y utilizar **técnicas** como la memoria virtual.
* Memoria **Secundaria** (HDD, SSD, Etc…)
  + Proporcionan almacenamiento **persistente** (**no volátil**). El SO los gestiona a través de su sistema de archivos, que se encarga de almacenar, organizar, recuperar y proteger la información (archivos y directorios) a largo plazo
* Dispositivos de **E/S**
  + Son la interfaz para la **comunicación** con el **mundo** **exterior**. El SO debe controlar estos dispositivos a través de controladores (drivers), manejar las solicitudes de E/S y abstraer su complejidad para el usuario
* BUS del Sistema
  + Es el canal de comunicación que **conecta todos los componentes internos del hardware** (CPU, memoria, dispositivos de E/S). El SO debe interactuar con este bus para coordinar la transferencia de datos entre los distintos componentes.

1. Componentes:
2. **Kernel**
   * Es el corazón del SO. Es la parte más crítica y privilegiada, que se carga en memoria al iniciar la computadora y permanece allí. Gestiona directamente el hardware y proporciona servicios esenciales a los demás componentes y a las aplicaciones
3. **Shell**: GUI / CUI o CLI
   * Es la interfaz entre el usuario y el kernel
4. **Herramientas**: Editores, compiladores, librerias, etc…
5. **Administrador de memoria**: Módulo (a menudo parte del kernel) dedicado exclusivamente a la memoria
   * Lleva un registro de cada byte en memoria, decidiendo qué procesos se cargan en memoria y dónde. Se encarga de técnicas como paginación, segmentación y memoria virtual
6. **Sistema de Archivos:** El componente responsable de la organización y manipulación de los datos almacenados
   * Gestiona cómo se almacenan y recuperan los datos en los discos. Crea la estructura de directorios (carpetas) y se encarga de los permisos de acceso, el nombre de los archivos y sus atributos
7. **Administrador de Procesos:** El módulo que trabaja con la creación, ejecución y eliminación de procesos e hilos
   * Asigna recursos a los procesos, sincroniza la comunicación entre ellos (IPC - Comunicación entre Procesos) y evita condiciones de carrera o interbloqueos (**deadlocks**)
8. **Drivers**: Programas especiales que actúan como traductores entre el hardware específico y el sistema operativo
   * Permiten que el kernel se comunique con un dispositivo de hardware (una impresora, una tarjeta gráfica, etc.) sin necesidad de conocer sus detalles específicos. Cada dispositivo requiere su propio driver
9. Una llamada al sistema (**System** **Call**) es la forma en la que los programas de usuario acceden a los servicios del sistema operativo (Es una RUTINA o FUNCIÓN).
   * Los **parámetros** asociados a las llamadas pueden pasarse de varias maneras: por registros, bloques, tablas de memoria o stack (pila)
   * Se ejecutan en **modo kernel**o **supervisor**
   * Los S.O cambian su SYSTEM CALLs dependiendo la arquitectura del procesador

Es posible implementar los llamados al sistema mediante instrucciones de código. Ejemplo: El **read** en assembler es una librería que ejecuta un proceso, el cuál dentro produce una interrupción para pasar al modo kernel. Allí se verifica quien precisa del uso del kernel mediante el uso de un registro y se procede a ejecutar la instrucción (SysCall Handler). *\*Falta detallar el retorno al llamado… Está en carpeta*

1. Un **programa** es un archivo con código dentro que está almacenado en memoria secundaria esperando a ser ejecutado.

* Es estático
* No tiene PC (Program Counter)
* Existe desde que se edita hasta que se borra **(Ciclo de vida)**

Un **proceso** es un programa en **ejecución**. Es una entidad de abstracción, es decir, tiene un “algo” en CPU pero falta algo…

* Es dinámico, es quién ejecuta el programa
* Tiene un PC (Program Counter)
* Existe desde que se solicita hasta que se termina **(Ciclo de vida)**

1. La información mínima que debe tener el Kernel sobre un proceso es:

* **PID** del proceso (Identificador)
* **Estado** del proceso (Si está en *listo****,*** *ejecutándose, bloqueado o esperando*)
* **PC**
* **Registros** de la CPU
* **Espacio de direcciones de memoria** (Memoria asignada al proceso)
* **Prioridad** o **Información de planificación** (Tiempo dedicado al proceso)
* **Información de E/S y archivos**
* **Privilegios** (Información de su propiedad. UID, GID, usuario, etc..)

La estructura de datos asociada al proceso donde se almacena dicha información es la **PCB** (Process Contol Block). El PCB es la "*cédula de identidad*" completa de un proceso dentro del sistema operativo. Contiene toda la información mínima y necesaria para que el kernel pueda detener un proceso y luego reanudarlo más tarde sin que este se percate de la interrupción, permitiendo así la multitarea.

1. Los **algoritmos de planificación (scheduling)** son el componente del kernel responsable de decidir qué proceso en estado "Listo" se asignará a la CPU a continuación.

Sus objetivos se pueden clasificar en dos categorías principales: **orientados al usuario** (visibles desde fuera del sistema) y **orientados al sistema** (internos y de eficiencia).

* Orientados al usuario:
  + Minimizar el Tiempo de Retorno (Turnaround Time)
  + Minimizar el Tiempo de Espera
  + Minimizar el Tiempo de Respuesta
* Orientados al sistema
  + Maximizar el rendimiento (Throughput)
  + Maximizar la utilización del CPU
  + Equidad
  + Previsibilidad

1. Los algoritmos **apropiativos (Preemptive)** existen en situaciones en los que el proceso en ejecución es expulsado por la **CPU**. Los algoritmos **no apropiativos (Non- Preemptive)** son cuando los procesos se ejecutan hasta que el mismo, por **cuenta** **propia**, abandona la **CPU**.

* El **SHORT-TERM SCHEDULER** determina cuál de todos los procesos que estén listos para ejecutarse se ejecutará a continuación en un ambiente multrporgramado.
* El **LONG-TERM SCHEDULER** controla el grado de multi-programación, es decir, la cantidad de procesos en memoria. *\*Puede no existir y absorber esta tarea el short-term.*
* El **MEDIUM-TERM SCHEDULER** reduce el grado de multiprogramación (Solo si ES NECESARIO). “Saca” temporalmente de la memoria los procesos que sea necesario para mantener el equilibro del sistema.
* Los NOMBRES de los SCHEDULER provienen de la frecuencia de ejecución

1. El **DISPATCHER** se encarga de realizar el cambio de contexto, cambio de modo de ejecución, “Despacha” el proceso elegido por el **short term**.

El **LOADER** carga en memoria el proceso elegido por el **long term**.

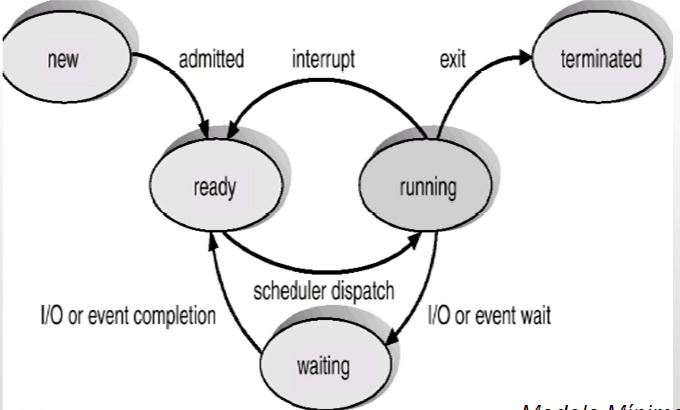
1. Que un proceso sea **CPU-Bound** significa que el mismo se pasa la mayor parte del tiempo utilizando la CPU. Que un proceso sea **I/O-Bound** significa que pasa la mayor parte del tiempo esperando por I/O (E/S)
2. Los estados posibles por los que puede atravesar un proceso son:

* **NEW:** Un usuario “dispara” el proceso. Un proceso es creado por otro, su padre

**En este estado** se crean las estructuras asociadas (PCB), el proceso queda en la *cola de procesos*, normalmente en espera de ser cargado en memoria

* **READY:** El proceso queda en estado listo, solo necesita que se le asigne CPU
* **RUNNING:** Se le asignó CPU al proceso y este lo tendrá hasta que se termine el período de tiempo asignado (**quantum o time slice**), termine o hasta que se necesite alguna operación de E/S
* **WAITING**: El proceso precisa que se cumpla un evento esperado para continuar. Este puede ser una E/S solicitada o la llegada de una señal por parte de otro proceso.

**Sigue en memoria** pero sin CPU. Al cumplirse el evento para a LISTO (Ready)



**Transiciones:**

* **New-Ready**: Por elección del scheduler de largo plazo, el **loader** carga en memoria el programa.
* **Ready-Running**: Por elección del scheduler de corto plazo, el **dispatcher** asigna la CPU.
* **Running-Waiting**: El proceso se “pone a dormir”, esperando por un evento.
* **Waiting**-**Ready**: Terminó la espera y compite nuevamente por la CPU.

1. De los schedulers anteriormente mencionados el que se encarga de las transiciones en los estados es el **LONG-TERM** (Encargado de elegir el proceso para cargarlo en memoria, queda en LISTO) y el **SHORT-TERM** (Elige el proceso para asignarle CPU, queda en RUNNING)
2. **Tiempo de Retorno (Tr):** Tiempo que transcurre entre que el proceso llega al sistema hasta que completa su ejecución.

**Tiempo de Espera (Te):** Tiempo que el proceso se encuentra en el sistema esperando, es decir, que pasa sin ejecutarse (Tr - Tcpu).

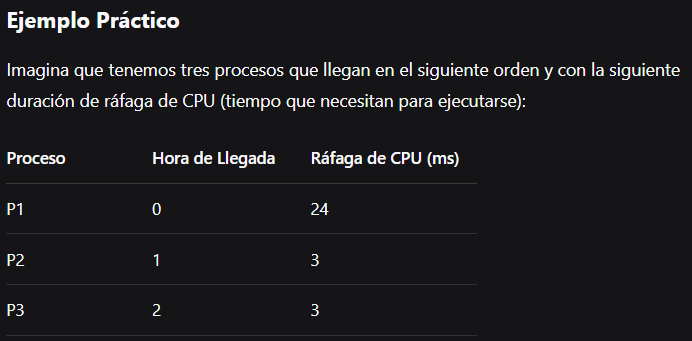
1. **Tiempo Promedios (TPR y TPE):** Tiempos promedio de retorno y espera. Promedio calculado de los tiempos individuales de cada proceso de lote.
2. **Tiempo de Respuesta:** Es el LAPSO DE TIEMPO que transcurre desde que un usuario (o proceso) emite una solicitud o comando hasta que se produce la primera respuesta perceptible o útil.

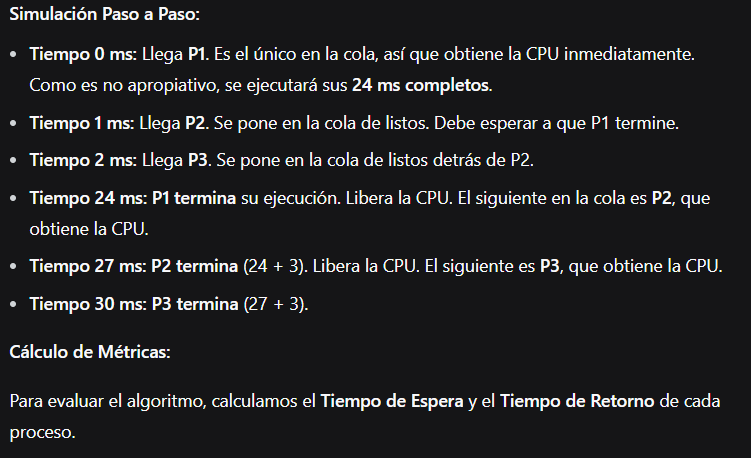
2) **Algoritmos de Scheduling**

1. **First Come First Served (FCFS):** Su funcionamiento se basa en una cola **FIFO** (First In, First Out).

* Los procesos se atienden en el orden estricto de llegada a la cola de LISTO
* Quien llega primero obtiene la CPU primero, la mantiene hasta que termina
* Solo cuando el proceso en ejecución libera la CPU, se asigna la CPU al siguiente proceso de la cola
* No son apropiativos

**Ejemplo:**





**SJF – Shortest Job First**: Es un algoritmo de planificación que **da prioridad al proceso con la ráfaga de CPU (tiempo de ejecución) más corta siguiente**

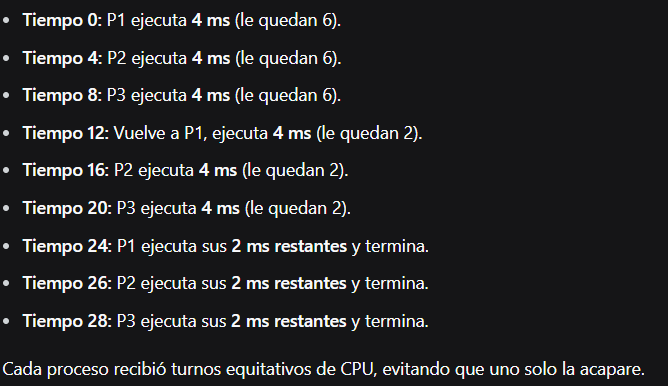
* Cuando la CPU se libera, se selecciona del estado “READY” el proceso que **tenga el tiempo de ráfaga siguiente más corto**
* No se interrumpe el proceso hasta que termine su tiempo en la CPU
* No apropiativo

**Ejemplo:** Llegan P1 (duración 6), P2 (duración 8), P3 (duración 7). El planificador SJF elige el orden: P1 (6), P3 (7), P2 (8), porque **6 < 7 < 8.**

**Round Robin:** Es un **algoritmo apropiativo** diseñado para sistemas de tiempo compartido. Asigna la CPU a cada proceso por un **quantum** **de** **tiempo** **fijo** (ej: 10-100 ms). Si un proceso no termina en ese tiempo, **es expulsado de la CPU** y se vuelve a colocar al final de la cola de procesos listos.

* El QUANTUM (Q) es una porción de tiempo fijo. Es el **parámetro** **más importante** del algoritmo
* Los nuevos procesos se agregan al final de la cola. Un proceso, al ser expulsado, se agrega al final
* Su **objetivo** es proporcionar buenos **tiempos de respuesta y equidad (fairness)**, ya que todos los procesos reciben una porción de CPU de forma cíclica

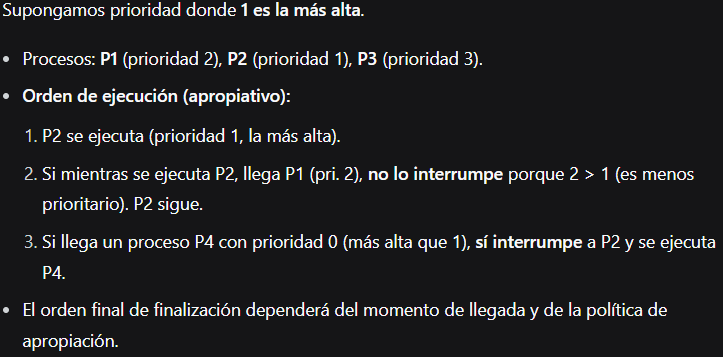
**Ejemplo:** Tres procesos (P1, P2, P3) con **ráfaga de 10 ms** cada uno y un **quantum** de **4 ms**.



**Prioridades:** A cada proceso se le asigna un **número de prioridad** (entero). La CPU se asigna al proceso que esté en la cola de listos y que tenga **la prioridad más alta** (que puede ser el número mayor o el menor, dependiendo de la implementación del sistema).

* La asignación de prioridades puede ser **estática** (Fijas durante toda la vida del proceso) o **dinámicas** (Cambian durante la ejecución)
  + Ejemplo dinámica: Prioridad baja si usa mucha CPU, alta si está esperando por E/S
* Pueden ser **No Apropiativos**, el proceso con mayor prioridad obtiene la CPU y la mantiene hasta terminar
* Pueden ser **Apropiativos**, un proceso con mayor prioridad llega al estado “READY” y **le quita inmediatamente la CPU** al proceso de menor prioridad en ejecución
* Si varios procesos tienen la misma prioridad, se suele usar otro algoritmo para desempatar (**FCFS**)

**Ejemplo**:



1. De los 4 algoritmos quien tiene el **parámetro** más claro y crítico a la hora de su funcionamiento es el **Round Robin (RR)** con el **Quantum** (Time Slic o Quanta). FCFS, SJF **no tienen un parámetro directo de configuración**. El algoritmo de prioridades tiene como parámetro la **propia prioridad asignada a cada proceso** (Valor específico para cada proceso).
2. No existe un único algoritmo adecuado para todos los casos. La elección del más adecuado depende del tipo de sistema y de los objetivos que se prioricen:

* **FCFS**: Es más adecuado para sistemas por lotes (Batch) simples donde la predictibilidad y la simplicidad son más importantes que el rendimiento interactivo
* **SJF**: Más adecuado para entornos especializados donde se puede predecir con alta precisión el tiempo de ejecución de los procesos
* **PRIORIDADES:** Es más adecuado para sistemas donde los procesos tienen **claros niveles de importancia o urgencia** distintos. Muy usado en **sistememas de tiempo real y kernels modernos**
* **Round Robin (RR):** Más adecuado para **sistemas de tiempo compartido e interactivos** (Escritorios, servidores de terminales, shells multiusuario). Es el **algoritmo por excelencia para este fin**. Proporciona un buen equilibrio entre el **TR** y **EQUIDAD**.

El “Más común” para sistemas de propósitos general (Windows, Linux, MacOs) es la combinación de **Round Robin y Prioridades** con **envejecimiento**. Los sistemas modernos **nunca usan un solo algoritmo**, utilizan esquemas **híbridos** o de **múltiples** colas:

1. **Ventajas** **FCFS**:

* Muy simple de implementar. Solo requiere COLA FIFO
* Facil de entender
* Justo por orden de llegada

**Desventajas FCFS:**

* Bajo rendimiento para métricas de tiempo de espera promedio
* No es apropiativo, lo que hace que no sea adecuado para entornos interactivos
* Poco eficiente si los procesos tienen grandes diferencias en sus tiempos de ejecución
* Efecto “**convoy”**; Un proceso largo retrasa al resto

\*

**Ventajas** **SJF**:

* Minimiza el tiempo de espera promedio, es **optimo**
* **Eficiente** para reducir el tiempo total de ejecución de un lote de procesos

**Desventajas SJF:**

* **Inanición (Starvation):** Los procesos largos pueden postergarse indefinidamente
* **Impredecible**: Es imposible saber con certeza la duración de la siguiente ráfaga
* Dificultad en la práctica

\*

**Ventajas** **Round Robin**:

* **Excelente** tiempo de respuesta
* **Equitativo**, todos los procesos reciben una porción de CPU
* **Apropiativo**, evita que un proceso robe la CPU

**Desventajas Round Robin:**

* **Alto overhead** (sobrecarga de consumo) por cambios de contexto si el quantum es pequeño
* **TR** puede ser alto para procesos largos
* El rendimiento depende **criticamente** de la elección del **quantum**

\*

**Ventajas Prioridades:**

* **Flexible**; Permite tratar procesos de distinta importancia de forma diferenciada
* Puede priorizar tareas críticas
* Se puede combinar fácilmente con otros algoritmos, ejemplo RR

**Desventajas Prioridades:**

* Los procesos de baja prioridad pueden no ejecutarse nunca
* Si no se usa **envejecimiento**, el sistema puede volverse injusto
* La asignación de prioridades puede ser compleja o subjetiva

4) En Excel

5) En Excel

6) En Excel

7)

a) **La Inanición (Starvation)** es un problema que ocurre en la gestión de recursos de un sistema donde un proceso **no puede obtener los recursos necesarios para progresar en su ejecución** debido a que otros procesos siempre tienen preferencia sobre él. El proceso afectado queda **“bloqueado”** indefinidamente esperando un recurso que nunca llega a obtener.

**En resumen**, la inanición es la situación indeseable donde un proceso es ignorado indefinidamente por el planificador debido a políticas de asignación injustas.

Surge típicamente en algoritmos que toman decisiones basadas en **prioridades** o en la naturaleza de los procesos.

b) Los algoritmos que pueden provocar Inanición son: **Algoritmo con Prioridades** y **SJF**.

c) La solución más común en los S.O Modernos es una técnica llamada **envejecimiento (Aging)**. Consiste en **aumentar gradualmente** la prioridad de los procesos que llevan mucho tiempo esperando en la cola.

Cómo resultado un proceso de prioridad baja que espero bastante eventualmente verá su prioridad aumentada hasta un nivel donde **obligatoriamente será seleccionado** para ejecutarse**.**

8)

9) Algunos algoritmos pueden presentar ciertas desventajas cuando en el sistema se cuenta con procesos ligados a CPU y procesos ligados a entrada salida. Analice las mismas para los siguientes algoritmos:

1. **Round Robin:** Para procesos que son CPU Bound podría presentar cómo **desventajas** que los tiempos de espera se vuelven bastante grandes, se genera cierto desperdicio de tiempo de CPU si usamos Timer Variable (TV) y un proceso termina antes de terminar su Quantum. También podemos incluir como desventaja que al usar este algoritmo en procesos CPU Bound se puede producir un Overhead de context switch que reduzca la eficiencia del sistema
2. **SRTF (Shortest Remaining Time First):** Este algoritmo **BENEFICIA** a los procesos **I/O Bound**, pero para los procesos CPU Bound este algoritmo podria llegar a generar inanición de procesos con larga duración y también podría generar una sobrecarga de cambios de contexto si la mayoria de procesos que están en la cola de listos tienen un tiempo de ejecución similar.

11) Si, podria suceder que nunca el quantum de un proceso llegue a cero si el proceso solamente requiere 1 ráfaga de CPU y termina antes que el quantum asignado. En condiciones normales y con un S.O bien diseñado esto no ocurriría

12) Ni en pedo lo hago.

13)

1. Para colas multinivel, teniendo dos tipos de procesos (Interactivos y Batch), utilizaría el algoritmo de **Round Robin** para procesos interactivos y el **FCFS** para procesos Batch. En el primer caso opté por RR dado que se necesita proporcionalidad en ellos y un tiempo de respuesta con rapidez. En el segundo caso opte por FCFS dado que se debe priorizar el mayor tiempo de CPU a cada proceso.
2. Utilizaría un Algoritmo de PRIORIDADES.

14) Realicé el **punto A** en excel, queda el B (Apropiativo)

15) NO sé cómo contar o llevar a cabo las UT, **preguntar**

16) **Pendiente**

17)

18) Los tipos de trabajos que benefician a **a) prioridad determinada estáticamente con el ḿetodo del más corto primero (SJF)** son:

* Cortos acotados por CPU
* Cortos acortados por E/S
* Largos acortados por CPU se ven **penalizados** siempre que haya procesos más cortos que estos
* Largos acrotados por E/S se ven **penalizados** por lo mismo que los anteriores más el tiempo de espera para usar la E/S

Mientras que los tipos de trabajos que benefician a **b)** **prioridad dinámica inversamente proporcional al tiempo transcurridoo desde la ultima operación de E/S** pueden ser aquellos que usan la E/S ya que serian los únicos capaces de escalar en prioridad.

18) Si el QUANTUM en RR se incrementa sin límite, el método de planificación se aproxima a **FIFO** dado que va a llegar un momento en el que el QUANTUM sea tan grande que todos los procesos terminarian o saldrian de la CPU sin ser expulsados por el algoritmo, sino que por sus propios medios. De esta manera la cola se convertiria en una cola circular del estilo **FIFO**.

19)

* **CASO 1:** En este caso si X tuviera un valor igual a cero significa que el proceso padre fue quien invocó al fork, lo que decantaria que X es la copia de un proceso padre siendo asi el HIJO. Este ejecutaria execv para reemplazar su espacio de memoria por las instrucciones del comando **ls** y luego imprimiria **“MUNDO”.** Si X fuera un valor negativo, se imprimiría un error.
* **CASO 2:**
  + La primer línea imprime **“Hola”**
  + **IDEM** que en caso 1
  + Luego se imprimiría “FIN”
* **CASO 3:**
  + La primer línea imprime **“Anda a rendir el primer parcial de promo!”**
  + Si el valor que retorna fork() asignada a la variable NEWPID es igual que 0, quiere decir que este es la copia de un proceso padre, siendo así el hijo. Al ingresar al IF, se imprime “**Estoy comenzando el Examen**”, luego se ejecuta la instrucciń **execv** reemplazando el espacio de memoria del proceso hijo por las instrucciones del comando **PS**, para así luego imprimir “**Termine el examen**”
  + Luego se imprimiría “¿Cómo te fue?”
  + Se finaliza el proceso, se retorna al estado del mismo
  + Imprime “Ahora anda a descansar”

20)

1. Figuran 8 lineas con la palabra “Poceso”
2. SI, el número de lineas es la cantidad de procesos en ejecución. No entiendo bien por qué pero la cantidad de lineas que se imprimen no respetan la cantidad de veces que itera el for

21)

1. Los valores que imprime la consola son los numeros de los procesos en ejecución
2. Deberian tener valores distintos, en mi caso se imprimieron todos del mismo valor y no sé por qué
3. -
4. -

22) La MMU (Memory Managment Unit) es un dispositivo de hardware encargado de mapear direcciones virtuales a físicas. Es parte de la CPU.

23) El espacio de direcciones de un proceso es un RANGO de direcciones (A memoria) posibles que un proceso puede utilizar para direccionar sus instrucciones y datos. Su tamaño depende de la arquitectura de la CPU: Si es de **32 bits** seria de **0..232 – 1, IDEM** para 64 bits.

24) **- Dirección Lógica**: Es una referencia a una localidad de memoria independiente de la asignación actual de los datos en la memoria. Representa una dirección en el “Espacio de direcciones del proceso”

**- Dirección Física**: Referencia a una localidad en la memoria fisica (**RAM).** Es una dirección **absoluta**

La relación entre estas 2 direcciones no es solo que ambas referencian a una localidad, sino que en caso de usar DIRECCIONES LÓGICAS es necesaria algún tipo de conversión a DIRECCIONES FÍSICAS.

25)

1. **-Particiones FIJAS:** La memoria es dividida en regiones de tamaño FIJO (Todas estas pueden ser del mismo tamaño o no)>

> Alojan un proceso c/u

> Cada proceso se coloca de acuerdo a algún criterio (Primer ajuste, mejor ajuste, peor ajuste y siguiente ajuste)

**-Particiones DINÁMICAS:** La memoria es dividida en regiones de tamaño DINÁMICO, varían en tamaño y en número.

> Alojan un proceso c/u

> Cada proceso se genera en forma dinámica del tamaño justo que necesita el proceso

**VENTAJAS Y DESVENTAJAS:**

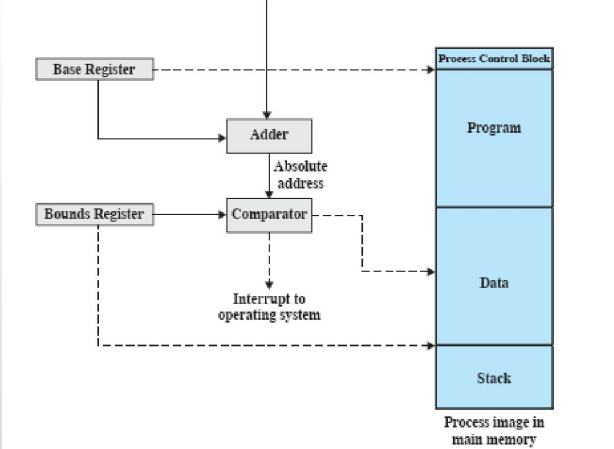
Las PARTICIONES FIJAS son más faciles de administrar a diferencia de las DINÁMICAS

Las PARTICIONES FIJAS pueden desperdiciar espacio si un proceso ocupa menos del que tiene la región de memoria

Las PARTICIONES FIJAS generan FRAGMENTACIÓN INTERNA

Las PARTICIONES DINÁMICAS generan FRAGMENTACIÓN EXTERNA

1. -
2. **Gráfico:**



26) Ventajas citadas en PUNTO 25.A

27)

1. La fragmentación se produce cuando una localidad de memoria no puede ser utilizada por no encontrarse en forma contigüa.

La **FRAGMENTACIÓN INTERNA**: Se produce en particiones fijas. Es la porción de la partición que queda sin utilizar

La **FRAGMENTACIÓN EXTERNA:** Se produce en el esquema de particiones dinámicas. Son **huecos** qe van quedando en la memoria a medida que los procesos finalizan

NO se encuentran en forma contigüa (Ordenado) por lo que puede darse que tengamos memoria libre para alocar un proceso pero no se pueda utilizar

1. Se acude a la compactación, pero es muy costosa.

28)

* **BEST FIT (MEJOR AJUSTE):** Busca la partición más pequeña que puede contener el proceso
* **WORST FIT (PEOR AJUSTE):** Asigna el proceso a la partición libre más grande disponible
* **FIRST FIT (PRIMER AJUSTE):** Asigna el proceso a la primera partición libre que sea lo suficientemente grande
* **NEXT FIT (SIGUIENTE AJUSTE):** Similar al FIRST FIT pero comienza la busqueda desde la última posición asignada

La i**nformación que debe mantener el kernel** para implementar estas técnicas son: La **tabla de particiones** con su estado, dirección base, tamaño, proceso asignado y una **estructura de listas** para gestionar particiones libres; **Best/Worst FIT** es una lista ordenada por tamaño y **First/Next FIT** es una lista ordenada por dirección de memoria.

Para las **particiones fijas** la mejor técnica es la del **primer ajuste** dado que el tamaño de las particiones es predeterminado. La fragmentación interna en esta política es inevitable pero FIRST FIT la minimiza al encontrar rápidamente la partición adecuada. De todas maneras tiende a dejar particiones muy pequeñas e inutilizables.

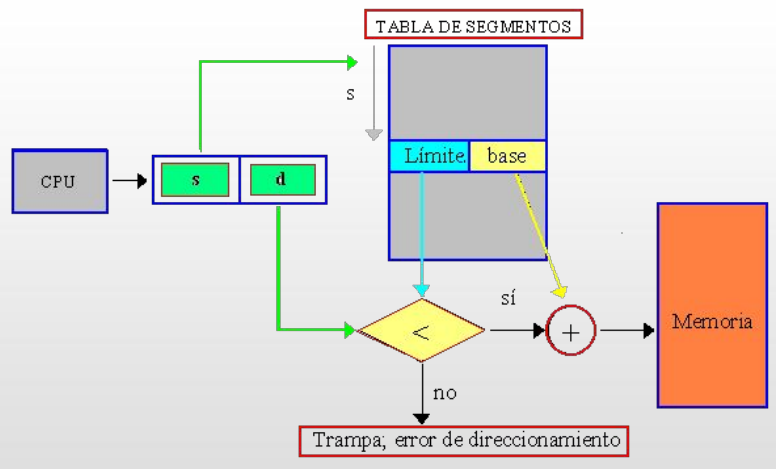
Para las **particiones dinámicas** la mejor técnica es la de **mejor ajuste** dado que se le asigna la partición más pequeña que puede ocupar un proceso, es decir, ocupa lo necesario. Minimiza la fragmentación externa al dejar los fragmentos más pequeños posibles, es decir, aprovecha mejor la memoria.

29)

1. La **segmentación** es un esquema que se asemeja a la ‘visión del usuario’. El programa **se divide en partes/secciones**.

Hay que tener en cuenta que un proceso es una colección de segmentos. Un **segmento es una unidad lógica** como lo es un programa principal, procedimientos, funciones, variables locales, globales, stack, etc.

1. Las estructuras necesarias que debe tener el kernel son; una **TABLA DE SEGMENTOS** por cada **PROCESO**. Esta permite mapear la dirección lógica en física, donde cada entrada contiene una dirección BASE (Dirección física de comienzo del segmento en RAM y una dirección LÍMITE (Longitud del segmento). Además debe contar con la **Segment-table base register (STBR)**, quien apunta a la ubicación de la tabla de segmentos y con la **Segment-table lenght register (STLR)**, el cuál contiene la cantidad de segmentos de un programa.
2. Gráfico:



1. SI, la **segmentación** produce **fragmentación externa** dado que cada segmento tiene un tamaño diferente según las necesidades del proceso, lo que impideuna asignación uniforme de la memoria. Los huecos que quedan entre segmentos son de tamaños irregulares. Además, la asignación de segmentos no necesariamente son contigüas.

Ejemplo → **Memoria**: [*S1] [Hueco] [S2][S3] [Hueco] [S4] [Hueco] [S5]*

1. La técnica de **MEJOR AJUSTE** es la adecuada para la segmentación, coincide con la naturaleza de la misma. Me refiero a que la segmentación maneja tamaños **variables** de segmentos y BEST FIT está **especificamente diseñado para manejar asignación de tamaños. variables**, busca el “hueco” perfecto para cada segmento.

**Ejemplo**:

Memoria disponible: [Hueco 50K][Hueco 30K][Hueco 100K]

Segmento a cargar: 45K

- **Best Fit**: Usa Hueco 50K → deja 5K

**- Worst Fit:** Usa Hueco 100K → deja 55K (desperdicio mayor)

**- First Fit:** Usa Hueco 50K → deja 5K (pero no siempre óptimo)

30)