Sistemas Operativos

Introducción

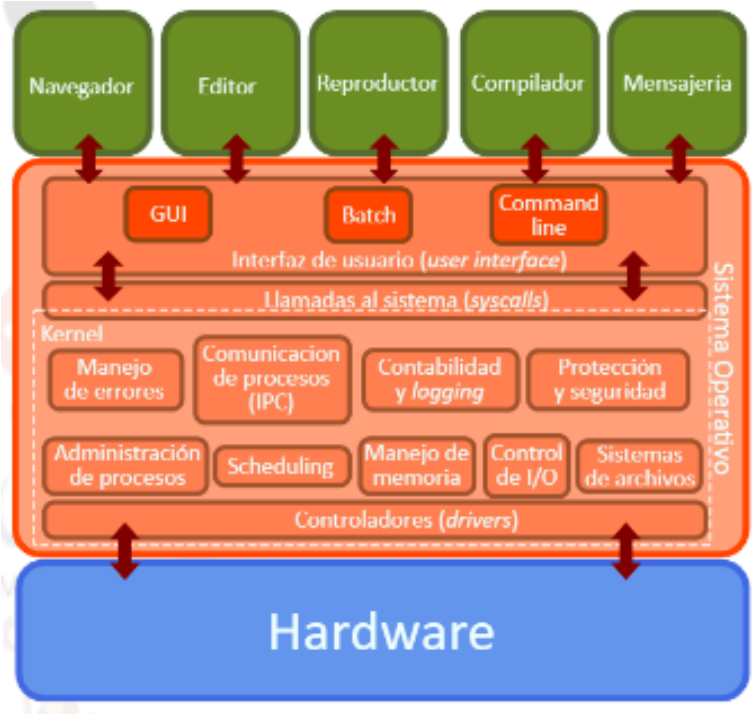
Un sistema operativo es un hardware que se encarga de gestionar todos los recursos del sistema informático (*software* que habla con el *hardware*), tanto de *hardware* como de *software*, permitiendo así la comunicación entre el usuario y el ordenador mediante la abstracción del *hardware*.

Un SO debe realizar las siguientes tareas:

1. **Administrar eficientemente los recursos**, es decir, no gastarlos (CPU, memoria, disco, etcétera) al ir decidiendo cuál programa debe ejecutar ahora en tal CPU, o bien, en qué parte del disco debo almacenar la información, entre otros.
2. **Proteger los recursos** para que sea seguro y confiable, es decir, evitar que procesos sobrescriban en los datos de otros procesos, saber cuál usuario está usando la CPU, manejo de permisos, etcétera.
3. **Abstraernos de los detalles del hardware** para que el usuario no deba preocuparse de éste y poder usar el ordenador de manera fluida.

Estructura

Esquema de un SO:



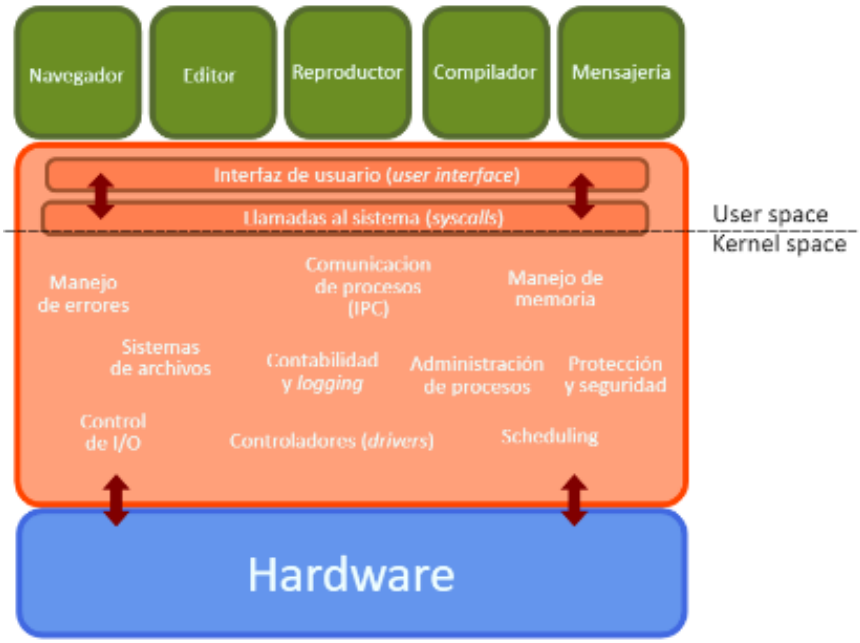
Los SO proveen **interfaces de usuario (user interface)** que enmascara las **llamadas al sistema (SYSCALLS)** para ofrecer un entorno más amigable para usar el computador. Estas *SYSCALLS* les permiten a los programas delegarle tareas al SO para que ejecute, por ejemplo, Spotify le indica al SO que active los parlantes.

El **kernel** o núcleo es un *software* que se define como la parte que se ejecuta en **modo privilegiado (superuser)**, y es el responsable de facilitar a los distintos programas acceso seguro al *hardware* de la computadora a través de las *SYSCALLS*, básicamente maneja los recursos. Por otra parte, todo lo que no es parte del *kernel* opera en **modo usuario (user mode)**, por lo que un SO es el conjunto de *kernel* con programas del sistema.

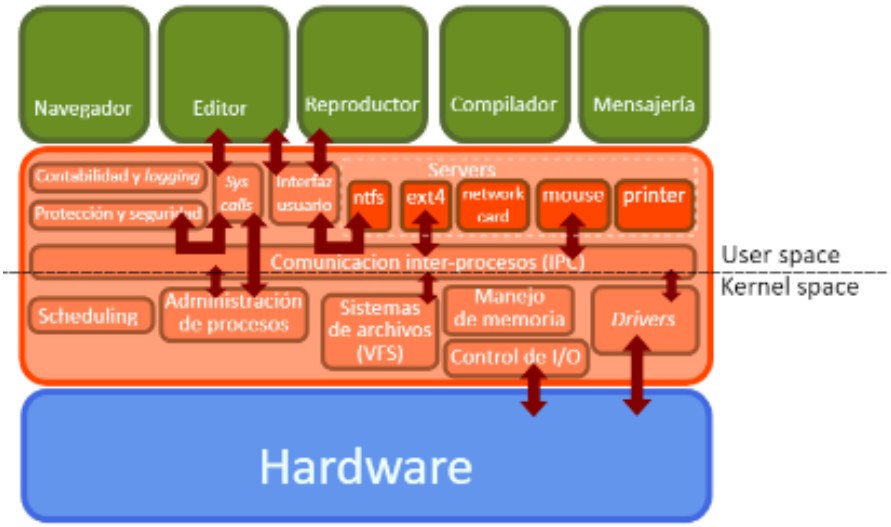
Arquitecturas

Hay tres tipos de arquitecturas:

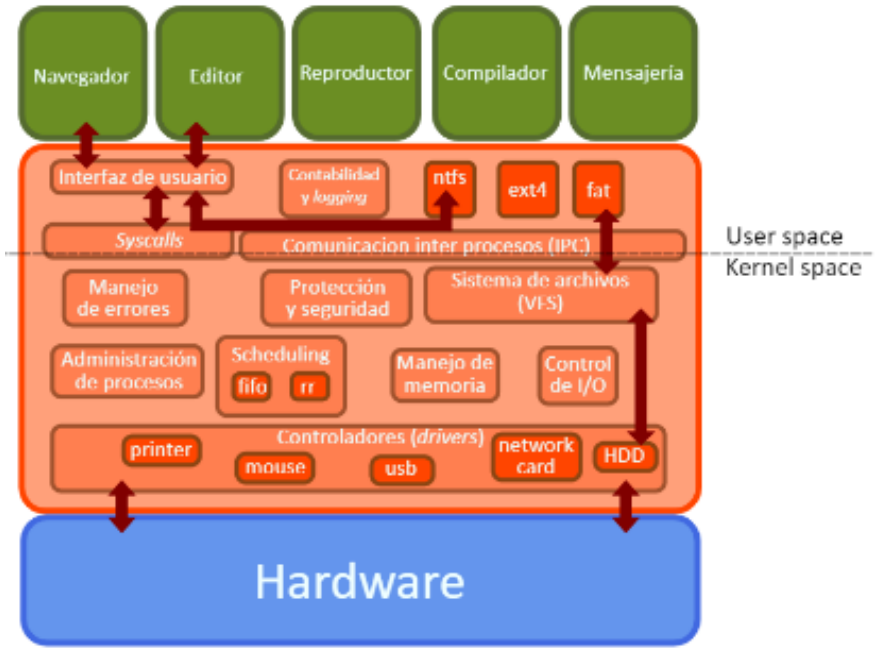
1. Monolíticas: Todo está dentro de un único programa, es decir, servicios para el sistema y usuario están en el mismo espacio por lo que todos los servicios se ejecutan en modo *kernel* causando que sea complejo, difícil de extender, de mayor tamaño y que sobre espacio en el **user space**. Si bien su ejecución es más rápida, una falla de un servicio puede comprometer al *kernel*. Cabe destacar que también hay monolíticos con módulos.



2. Microkernel: Mejor distribución de recursos entre *user* y **kernel space** ya que solo los servicios básicos se encuentran en el *kernel*, mientras que los otros se encuentran en el *user space*. Lo anterior causa que el *kernel* sea más sencillo, pequeño y fácil de portar por lo que es más extensible que una arquitectura monolítica. Además, como es modulado, los procesos deben comunicarse de manera especiar mediante el paso de mensajes lo que se denomina como **Inter – Process Communication (IPC)**. Si bien, su ejecución es con **overhead** por comunicación (exceso de tiempo de computación, memoria, u otros recursos), una falla no compromete al sistema completo.



3. Híbrido: Son las arquitecturas actuales que combinan lo mejor de cada uno. Básicamente consiste en un *kernel* monolítico con módulos que tiene servicios/funcionalidades que se ejecutan en el user space.



Funcionamiento

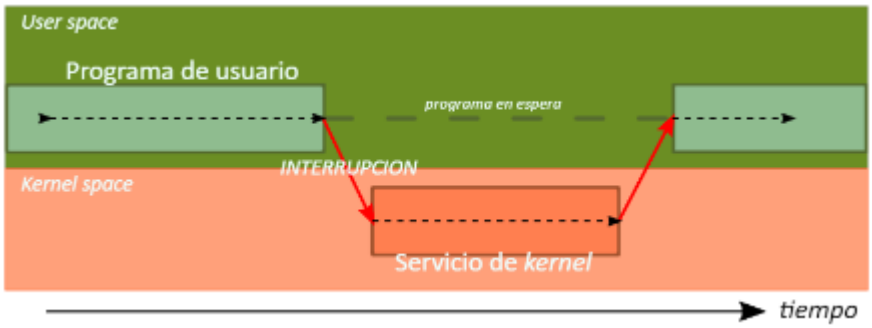
Al iniciar el computador, se ejecuta el código de la tarjeta madre, es decir el **BIOS (Basic I/O System)** que tiene la misión de inicializar e identificar el *hardware*, y ubicar y arrancar mediante un **bootloader** (*software* encargado de activar el SO) el *kernel* en modo privilegiado.

Como el *kernel* es el único que puede ejecutar una instrucción en modo privilegiado, si en *user mode* se trata de hacer lo mismo, el *hardware* generará una interrupción **(protection fault)**, o bien, ignorará la instrucción. Algunas de estas instrucciones son las siguientes:

1. Modificar el vector de interrupciones
2. Acceder a dispositivos I/O
3. Modificar el *timer* del computador
4. Detener el computador **(halt)**
5. Cambiar el bit de modo (*user* a *kernel*, o viceversa)
6. Modificar tablas de acceso a memoria
7. Almacenar el estado de la máquina

Los SO son manejados por **interrupciones**, por lo que mientras nadie lo llame, el SO no hará nada. El seguimiento es el siguiente:

1. Programa de usuario genera una interrupción
2. Control pasa al sistema operativo: *kernel space*
3. SO ejecuta el servicio solicitado
4. SO regresa el control al programa de usuario



SYSCALLS: Llamadas al Sistema

Es una interfaz para solicitar servicios al SO mediante una manera precisa que es altamente dependiente del *software* en el que participa un proceso ejecutable de usuario, una biblioteca (estática o dinámica) que invoca la *SYSCALL*, un vector de interrupciones configurado por el SO, y un código de *SYSCALLS* del SO que usa el *hardware*. Algunos *SYSCALLS* son los siguientes:

1. **Fork:** Crea otro proceso como copia del creador, pero con distinto número de proceso (`if (PID == 0) proceso es hijo; else es el padre`). Por lo tanto, se le asigna un nuevo espacio de memoria para almacenar toda la información que tiene el padre, por lo que sus variables no apuntan a las mismas direcciones de memoria. Para poder compartir memoria deben emplear IPC mediante mensajes con (`ftok`, `shmget` y `shmat`), al finalizar debe ejecutar **`exit(status)`**.
2. **`Exec`**: Carga un binario en memoria reemplazando el código de quién lo llamó, por lo que el nuevo programa se "roba" el proceso (la memoria se sobrescribe). Retorna solo si falla.
3. **`Wait`**: Espera a que termine un proceso hijo. Si no se especifica el PID, entonces va a esperar a que cualquier proceso hijo termine.

4. **`Exit`**: Termina el proceso actual y devuelve toda la memoria al SO.
5. **`Kill`**: Para indicarle a otro proceso que debe terminar. Existen varios tipos de *Kill*, pero por default se envía la señal **`SIGTERM`** que le indica a un proceso que debe terminar, sin embargo, también se puede usar **`SIGKILL`** que elimina al proceso "sin piedad".
6. **`Sleep`**: "Duerme" al proceso dejándolo bloqueado durante un tiempo definido.

Procesos

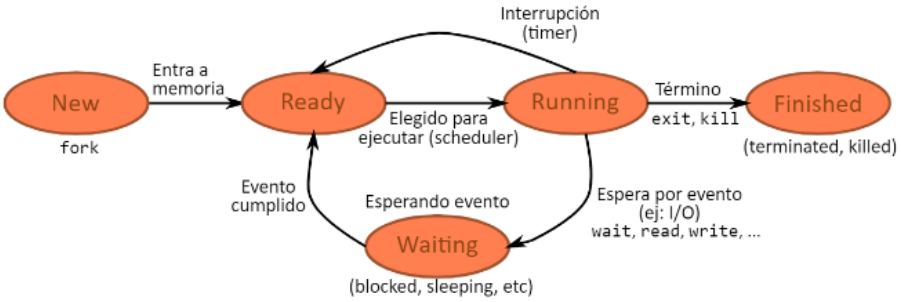
Son abstracciones para un programa en ejecución que están compuestos por un código (programa) y los recursos que utiliza (memoria, archivos, dispositivos I/O, entre otros). A partir de esta definición nacen las siguientes:

1. **Multiprogramación:** Capacidad de mantener múltiples procesos en memoria, esto es que dicho proceso está preparado para ser ejecutado en la misma CPU.
2. **Multitasking:** Capacidad de atender "simultáneamente" múltiples procesos usando la misma CPU.

Un proceso está compuesto por:

1. Código (información estática)
2. Datos (variables globales)
3. Stack, cuyos elementos representan un llamado a función que contiene parámetros, variables locales, lugar de retorno (donde estaba la ejecución anterior, PC)
4. Heap que corresponde a la memoria asignada dinámicamente durante la ejecución

Para tratar con los procesos el SO cuenta con el **Process Control Block (PCB)** que almacena toda la información de los procesos (estado, PID, PC, memoria, *scheduling*, registros, I/O, contabilidad, entre otros). Un proceso puede estar en diversos estados: **new** (en creación), **running** (ejecutándose), **waiting** (esperando I/O, señales), **ready** (listo para ejecutar), y **finished** (finalizado).



Para mantener la multiprogramación y el *multitasking* el SO debe realizar cambios de contextos **(Context Switch)** que siguen el siguiente procedimiento:

1. SO actúa luego de una interrupción (*SYSCALL*, *timer*, evento, etcétera)
2. SO almacena el estado de un proceso en su PCB y queda en pausa
3. SO restaura el estado del próximo proceso
4. Próximo proceso continúa su ejecución

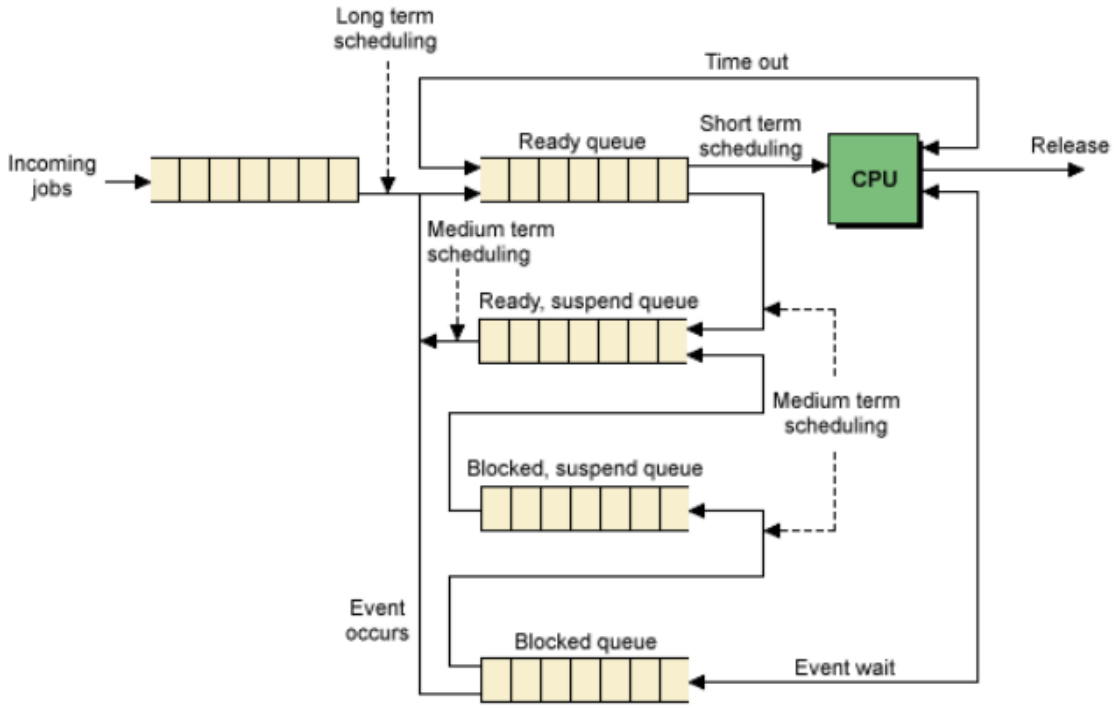
**Para averiguar el estado de un proceso resulta útil ejecutar 'ps aux'/'htop'/'top'.*

Creación de procesos

Todo proceso tiene un proceso padre **(parent)**, es más, todo proceso puede tener un proceso hijo **(child)** mediante `fork`. Tanto padre como hijo continúan ejecutando concurrentemente y existen en la memoria del computador. Sin embargo, puede suceder que un proceso padre termine/muera antes que sus procesos hijos, en estos casos los procesos hijos que aún están activos quedan **huérfanos** y pasan a ser hijos de `init` quien hará `wait` periódicamente por sus hijos. Por otra parte, puede ocurrir que cuando un proceso se convierta en **zombie** cuando termina y su padre no hace `wait`, por lo que el proceso terminado no se borra de la tabla de procesos (ni ejecuta) hasta que su padre haga `wait`.

Scheduling

Es el método empleado por el SO para elegir cuál de todos los procesos en estado *ready* debe ejecutar a continuación cuyo objetivo es maximizar el uso de la CPU en user time, y menos tiempo en *systime* (tiempo que ejecuta el SO), sin embargo, implementarlo junto al *context switch* producen *overhead*. El responsable de emplear **scheduling** es el **scheduler** que puede ser visto como un sistema de manejo de colas:



Hay distintos niveles de **scheduling**:

1. **Long – term Scheduler:** Admite procesos en la cola *ready* y determina el grado de multiprogramación; al usar `fork` se puede rechazar el llamado si queda poca memoria.
2. **Short – term Scheduler:** Selecciona un proceso de la cola *ready* para ejecutar y ejecuta el cambio de contexto.

3. **Medium – term Scheduler:** También conocido como el *dispatcher* ya que modifica temporalmente el grado de multiprogramación y ejecuta **swapping** copiando memoria RAM a disco, y de disco a RAM.

**Swapping es el procedimiento de sacar un proceso de la RAM y ponerlo en el disco (swap – out). Sirve para liberar memoria cuando el SO necesita más. Cuando se vuelve a ejecutar dicho proceso es necesario traerlo de vuelta (swap – in).*

Tipos de Scheduling

Conceptos importantes:

1. **Turnaround time:** Tiempo total que le toma al proceso terminar su ejecución, es decir, el tiempo que le toma pasar a estado *finished* desde que ingresa a la cola por primera vez.
2. **Response time:** Tiempo que le toma al proceso ser atendido por primera vez desde que ingresa a la cola en estado *ready*.
3. **Waiting time:** Tiempo total que el proceso estuvo en espera, ya sea en la cola esperando ser atendido, o bien, en estado *waiting*.

```
[t = 12] El proceso GERMY ha sido creado.
[t = 15] El proceso GERMY ha pasado a estado RUNNING.
[t = 17] El proceso GERMY ha pasado a estado WAITING.
[t = 18] El proceso GERMY ha pasado a estado READY.
[t = 19] El proceso GERMY ha pasado a estado RUNNING.
[t = 20] El proceso GERMY ha pasado a estado FINISHED.
```

Turnaround time

$$T_{Término} - T_{Llegada} = 20 - 12 = 8$$

Response time

$$T_{Atendido} - T_{Llegada} = 15 - 12 = 3$$

Waiting time

$$\sum_i (T^i_{RUNNING} - T^i_{READY}) + \sum_j (T^j_{READY} - T^j_{WAITING})$$

$$= ((15 - 12) + (19 - 18)) + ((18 - 17)) = 5$$

Hay diversos tipos de scheduling, entre estos destacan los siguientes:

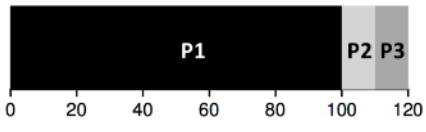
1. **Por interrupciones:**
   - **Preemptive (expropiativo):** Utiliza un *timer* para decidir el tiempo máximo que un proceso puede estar ocupando la CPU continuamente (**quantum**).
   - **Non - preemptive (no expropiativo):** Permite que un proceso ejecute hasta que deja voluntariamente la CPU, se bloquea en I/O, o termina.
2. **Por objetivo:**
   - **Batch:** Corresponde al trabajo por lotes y sin interacción que busca minimizar el *turnaround time* mientras maximiza el *throughput*.
   - **Interactive:** Busca minimizar tiempos de respuesta para satisfacer a los usuarios.
   - **Real time:** Diseñado para alcanzar *deadlines*. Su tiempo de respuesta debe ser predecible.
   - **Fairness:** Que todos los procesos tengan un tiempo razonable de ejecución.

Algoritmos de Scheduling

Hay muchos algoritmos de *scheduling* que cuyo desempeño dependerá netamente de los procesos a los que está atendiendo y de su implementación. Algunos son los siguientes:

1. **Batch:**
   - **First – Come, First – Served (FCFS):** Es por orden de llegada (*FIFO*), por lo que es *non – preemptive* y bastante simple, pero es poco predecible (*convoy effect)* ya que procesos de *CPU – burst* cortos pueden estar esperando a que uno muy largo termine, alentando todo el proceso.

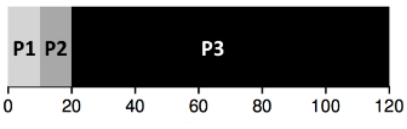
	Llegada	CPU-burst	Término	Turnaround
P1	0	100	100	100
P2	1	10	110	109
P3	2	10	120	118



Turnaround time promedio: 109
Si P2 hubiese llegado en $t = 0$, y P1 hubiese llegado en $t = 1$, entonces *turnaround time promedio* $\rightarrow 79$

   - **Shortest Job First (SJF):** El más corto primero, que es bastante óptimo. Su versión *preemptive* (**Shortest Remaining Time Next**) elige al que le queda menos tiempo. Lo negativo es que no se sabe cuánto demora cada *CPU – burst* y que los procesos largos pueden llevar a la **inanición (starvation)**, es decir, a que se le puede denegar el acceso al recurso compartido.

	Llegada	CPU-burst	Término	Turnaround
P1	0	10	10	10
P2	2	10	20	18
P3	1	100	120	119



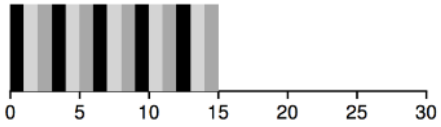
Turnaround time promedio: 49

**Optimiza el turnaround time*

2. **Interactive:**
   - **Round – Robin (RR):** Un turno (*quantum*) para cada proceso, por lo que es un algoritmo *fair*. Por lo tanto, se tiene que, si son n procesos, cada uno recibe 1/n de CPU, y ninguno esperará más de (n – 1) x *quantum* para ejecutar. De lo anterior se rescata que depende mucho del tamaño del *quantum*.

Ejemplo con $q = 1$

	Llegada	CPU-burst	Término	Turnaround	Inicio ejec.	T.Respuesta
P1	0	5	13	13	0	0
P2	0.1	5	14	13.9	1	0.9
P3	0.2	5	15	14.8	2	1.8

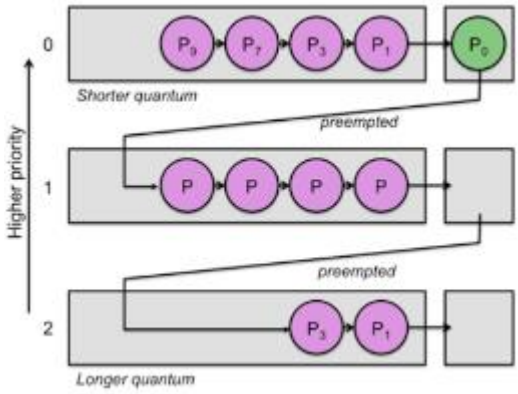


- Turnaround time promedio: 13.9
- Response time promedio: 0.9

**Optimiza el response time*

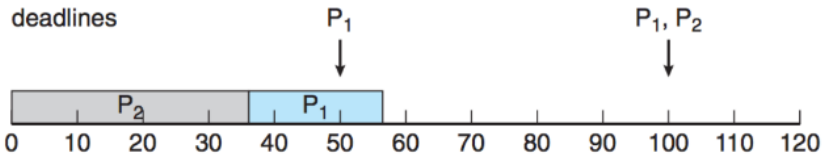
   - **Priority scheduling:** Cada proceso tiene asociada una prioridad que pueden ser estáticas o dinámicad, sin embargo, si las prioridades son iguales tendremos un scheduling del tipo *FCFS* o *RR* según el *quantum*. Por lo anterior, se puede producir inanición. Sin embargo, es posible incrementar la prioridad de procesos que llevan más tiempo (*aging*).

- **Multilevel Feedback Queue (MLFQ):** Basado en múltiples colas con distintas prioridades, por lo que optimiza el *turnaround time* mientras minimiza el *response time*. Funciona como sigue:
  - R1. Si $\text{priority}(A) > \text{priority}(B)$, ejecutar A
  - R2. Si $\text{priority}(A) = \text{priority}(B)$, ejecutar A y B con RR
  - R3. Procesos entran en la cola con **mayor** prioridad
  - R4. Si un proceso usa su q (acumulado en todos sus turnos), su prioridad se reduce
  - R5. Después de un tiempo S, todos los procesos se mueven a la cola con mayor prioridad



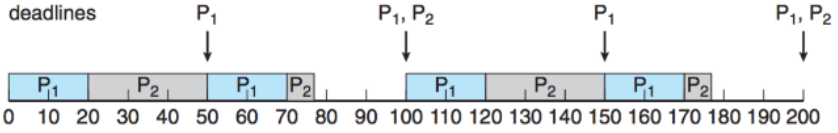
- Otros como: *Big O of 1 Scheduler O(1)*, *Completely Fair Scheduler*, y *Brain Fuck Scheduler (BFS)*.

3. **Real Time:**
   Son aquellos que poseen *deadlines* y periodos de ejecución, por lo que el SO debe determinar si dado un *deadline* d, un periodo p, y un tiempo de ejecución t, es capaz de incorporar un proceso a la ejecución.
   - **Rate Monotonic:**



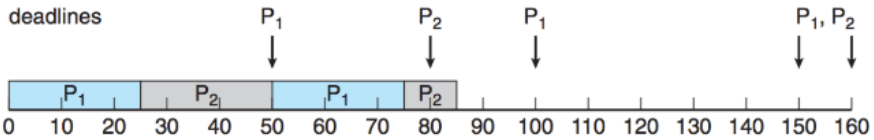
Cada proceso recibe prioridad: $\frac{t_i}{p_i}$

Con $P_1 : \{p_1 = 50, t_1 = 20\}$, y $P_2 : \{p_2 = 100, t_2 = 35\}$



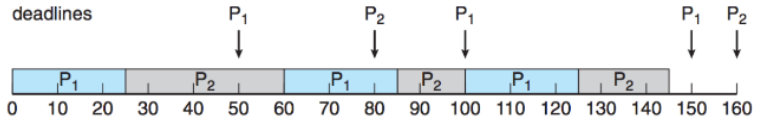
Rate Monotonic Scheduling es **estático**. Podría perder *deadlines*

Con $P_1 : \{p_1 = 50, t_1 = 25\}$, y $P_2 : \{p_2 = 80, t_2 = 35\}$



- **Earliest Deadline First (EDF):**

Earliest Deadline First Scheduling es **dinámico**. Elige siempre el que tiene *deadline* más cercano.
Con $P_1 : \{p_1 = 50, t_1 = 25\}$, y $P_2 : \{p_2 = 80, t_2 = 35\}$



Threads

Los *threads* son como procesos, pero más livianos, ya que solo tienen su propio TID, PC, registros y stack. Estos son útiles ya que nos permiten ejecutar distintas partes de un código de manera concurrente (competencia por uso de CPU) y distribuir tareas entre múltiples núcleos en sistemas *multicore* (paralelismo), ambos mientras comparten espacios de memoria, variables globales, archivos abiertos, procesos hijo, señales y manejadores de estas.

**Información privada tiene que ver con ejecución mientras que información compartida tiene que ver con recursos.*

Funciones típicas de la librería `pthreads`:

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

**Volatile: Indica el valor de una variable en memoria que puede ser modificado en un contexto que el compilador no puede determinar, así evita optimizaciones del sistema sobre esa variable. Es útil cuando la variable puede ser modificada "por fuera", es decir, cuando un proceso comparte explícitamente su memoria con otro.*

Implementación de Threads

Hay tres formas de implementar *threads*:

1. **User Space:** Son implementados y manejados por una biblioteca de usuario en donde el *kernel* no ve *threads,* sino que ve procesos *single threaded*. Algunas implementaciones usan variantes no bloqueantes de *SYSCALLS*. Cada proceso tiene su *thread table*.
   - Ventajas: Es portable en cualquier SO, tiene mejor escalabilidad ya que usa la memoria del proceso y ejecuta *context switch* y *scheduling* más rápido.
   - Desventajas: Llamadas a *SYSCALLS* pueden bloquear el proceso y requiere *scheduling* cooperativo sin *timers*.

2. **Kernel Space:** Son implementados y manejados nativamente por estructuras del *kernel* por lo que no necesita bibliotecas de usuario. El *kernel* tiene una *thread table* única para todos los procesos, además, usa *thread pools* que son patrones de diseño implementados para lograr concurrencia mediante múltiples *threads* esperando a recibir tareas, por lo que se evita la creación y destrucción de estos.
   - Ventajas: No requiere de bibliotecas (soporte nativo), *SYSCALLS* no bloquean el proceso, pero son más costosas ya que se requieren llamadas al *kernel*.
   - Desventajas: Escalabilidad está limitada por el SO, la semántica de señales y de `fork` no son claras.
3. **Hybrid:** Son implementaciones de *kernel threads* multiplexados entre *user threads*, en donde el *kernel* solo ve sus *threads*, mientras que varios *user threads* son asignados a un *kernel thread*.
   - Ventajas: Más livianos de crear, *SYSCALLS* no bloquean el proceso y es escalable como los *user threads*.

Sincronización

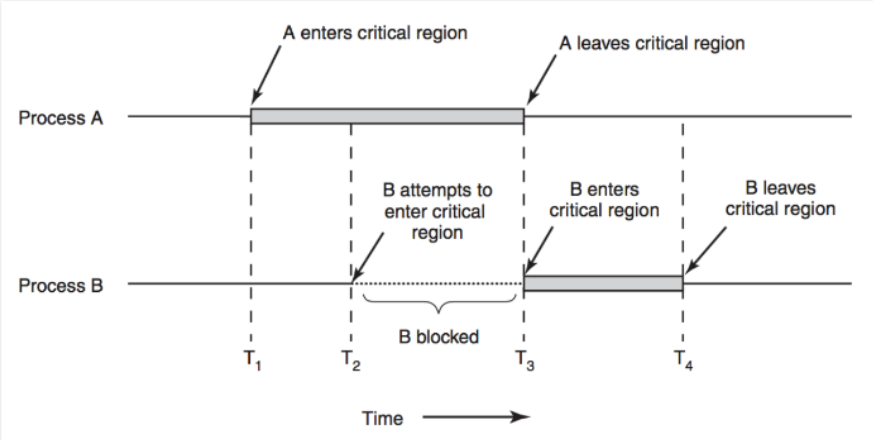
Race Conditions: Situación en que la salida de una operación depende del orden temporal de sus operaciones internas, la cual no está bajo control del programador.

Estas se pueden generar cuando hay más de un *thread* dentro del proceso que acceden a secciones críticas al mismo tiempo. Estas situaciones son evitables usando `lock` o algún mecanismo de sincronización.

Sección crítica (SC): Segmento de código en que el *thread* accede a recursos compartidos.

Una solución al problema de la sección crítica debe cumplir lo siguiente:

1. Exclusión mutua: A lo más un *thread* está en su SC
2. Progreso: Al menos un *thread* puede entrar a su SC. Si múltiples *threads* quieren entrar, se debe decidir quién entrara en un tiempo acotado
3. Espera acotada: Si un proceso quiere entrar a su SC, podrá hacerlo luego de una cantidad finita de tiempo



1. **Variable compartida**: `lock`

```
int lock = 0;  // 0==free, 1==busy
while(TRUE) {
    while(lock);     // busy waiting (spinlock)
    lock = 1;
    /* ... SC ... */
    lock = 0;
    /* ... out of SC ... */
}
```

- Cumple:
  - Progreso: Siempre hay al menos 1 *thread* que puede entrar
- No cumple:
  - Exclusión Mutua: La lectura y escritura de *lock* no pueden ser hechas de forma atómica (ambos podrían entrar a la SC).
  - Espera Acotada: Puede pasar un *thread* que ponga `lock = 0` y de inmediato vuelva a ejecutar el *loop* y entrar a la SC, mientras que otro *thread* que estaba esperando no lo podrá hacer, ergo, su espera podría no ser finita.

2. Solución compartida: `turn`

```
int turn = 1;  // o 0 ...
/* Thread 0 */
while(TRUE) {
    while(turn != 0){
       pass
    }
    /* ... SC ... */
    turn = 1;
    /* ... out of SC ... */
}
```

```
/* Thread 1 */
while(TRUE) {
    while(turn != 1){
        pass
    }
    /* ... SC ... */
    turn = 0;
    /* ... out of SC ... */
```

- Cumple:
  - Exclusión Mutua: Solo se entra a la SC si el dato es igual a un valor único por *thread* y `turn` no puede tomar dos valores simultáneamente.
- No cumple:
  - Progreso: Es necesario esperar que un *thread* dé acceso a otro para poder entrar, por lo que el tiempo necesario para que se cumpla no es acotado por lo que se menciona en espera acotada.
  - Espera Acotada: Si `turn = 3`, entonces solo el *thread* 3 podrá entrar, pero este no ha entrado por lo que no hay nadie en la SC y no se podrá decidir si entrará alguno de los *threads* que están esperando entrar, por lo que van a tener que esperar a que el *thread* 3 se ejecute y le dé permiso a otro que esté esperando entrar, sino se repetirá la misma situación.

3. Solución G.L. Peterson

```
int turn = 1;                 // o 0 ...
int flag[2] = {false, false}; // indica que thread está interesado en entrar

int me = // ... 0 ó 1;    /* Peterson es para dos threads */
int other = 1-me;
while(TRUE) {
    flag[me] = true;  /* Indico que quiero entrar*/
    turn = other;   /* Digo que es el turno del otro*/
    while(flag[other] && turn == other); /* Espero mientras el otro quiere entrar y es su turno*/
    /* ... SC ... */
    flag[me] = false; /* Indicó que no quiero entrar*/
    /* ... out of SC .../ */
}
```

- Cumple:
  - Exclusión Mutua: Al cambiar el `turn` a `other` y verificar el booleano del otro asegura que es posible entrar solo cuando él termine la SC, mientras que el otro no podrá.
  - Progreso: Siempre entra uno o el otro, y su uno no quiere entrar a la SC, su `flag` será `false` y por ello, si o si el otro podrá entrar.
  - Espera Acotada: No hay dependencia de que otro *thread* cambie algún valor para volver a ejecutar, por lo que un *thread* esperará entrar a la SC de forma finita a que se libere.

Sincronización por Hardware

Las soluciones por *software* dependen de que existan instrucciones atómicas (concisas/indivisibles) provistas por el *hardware*. Algunas arquitecturas proveen instrucciones atómicas del tipo `test_and_set` que altera el booleano que recibe a `true`, pero retorna el valor que tenía antes, así cuando es `false` se permite que un *thread* entre, pero antes le pone `lock = true` para que los demás se queden en un *loop*:

```
bool SOFTWARE_test_and_set (bool *target) {
    bool old_value = *target;
    *target = true;

    return old_value;
}
```

Veamos la SC para N procesos:

```
bool lock = false;  // false==free, true==busy
bool waiting[N];    // todos inicializados en false

while(True) {
  /* ... ... ... */
  waiting[i] = key = true;          /* I'm waiting and it's locked */
  while (waiting[i] && key)
    key = test_and_set(&lock);
  waiting[i] = false;               /* I'm no longer waiting */

  /* ... ... SC ... ... */

  j = (i+1)%N;                      /* Find waiting proc (clockwise circle) */
  while((j != i) && !waiting[j]){
    j = ++j%N;}
  if(j == i)                          /* -Nobody was waiting */
    lock = false;
  else                                /* -proc_j should go (stop waiting) */
    waiting[j] = false;
  /* ... out of SC ... ... */'
}
```

Acá el *thread* está esperando y `key` indica si logró tomar el *lock* o no. Cuando el `lock = false` deja de esperar y comienza a ejecutar, y como después `lock = true` todos entran en un *loop*. Luego de ejecutar, el *thread* original buscará a alguien que esté esperando para que deje de hacerlo y entre a la SC, pero si no hay nadie esperando, entonces pondrá el `lock = false` para que otro *thread* pueda llegar.

- Cumple:
  - Exclusión Mutua: El *lock* cambia de forma atómica y por ello, cuando los demás verifican, siempre lo verán como true y no podrán acceder hasta que el *thread* que entró le dé el paso a otro, o bien, ponga `lock = false`.
  - Progreso: Siempre puede entrar 1, y cuando se libera, se asegura de dejar a otro adentro ejecutando en la SC o dejar `lock = false`.
  - Espera Acotada: No depende de que otro *thread* cambie algún valor para volver a ejecutar, es más, éste dejará todo habilitado para que cualquiera entre, incluso él.
- Lo malo:
  - Implementación de *busy waiting*

**Spinlock: Similar a busy waiting pero con se espera un tiempo de bloqueo muy corto. Puede ser más eficiente que un lock regular cuando el tiempo de espera es suficientemente pequeño.*

Problema de **inversión de prioridad** entre dos procesos A y B que utilizan *busy waiting* ocurre cuando A tiene mayor prioridad que B, pero B posee un recurso que A necesita, por lo que B no podrá liberarlo ya que A está en estado *ready* y su prioridad no permite que B ejecute. Para solucionar esto podemos usar **priority inheritance** que permite que un proceso con menor prioridad pueda aumentar su prioridad temporalmente para poder liberar el recurso que tiene asignado.

Son soluciones que no se emplean directamente por *hardware* ya que se construyen o implementan acorde al *software*, por ejemplo, los **Mutex Locks** que realizan `lock.acquire()` para tomar el *lock* y `lock.release()` para liberarlo (error si no está tomado), ergo, se usan para proteger recursos compartidos.

Otro ejemplo son los **semáforos** que permite un número limitado de *threads* ejecutar una SC, por lo que se usan para **signaling**. El clásico ejemplo es el de producto – consumidor. Estos incluyen un contador S y dos operaciones:

1. **wait P()/down()**: Intenta disminuir el valor del contador, alude a pedir el *lock*
2. **signal()/up()**: Incrementa el valor del contador, alude a soltar el *lock*

Esto ayuda a sincronizar procesos y la implementación de exclusión mutua entre estos. Además, pueden usarse para que un proceso notifique a otro cuando se libere un recurso. Su implementación puede ser mediante *busy waiting* o bloqueo de *threads*.

```
mutex buffer_mutex; // similar to "semaphore
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer()
{
    while (true)
    {
        item = produceItem();
        down(emptyCount);
        down(buffer_mutex);
        putItemIntoBuffer(item);
        up(buffer_mutex);
        up(fillCount);
    }
}

procedure consumer()
{
    while (true)
    {
        down(fillCount);
        down(buffer_mutex);
        item = removeItemFromBuffer();
        up(buffer_mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

Aquí se ve un semáforo para solucionar el problema de múltiples consumidores y productores. Notar, que el semáforo y *mutex* funcionan distinto, ya que este último tiene el concepto de propiedad, en donde solo el productor o consumidor que disminuyó su valor puede volver a subirlo.

Las variables de condición son primitivas de sincronización que permite implementar una construcción que en sincronización de procesos se conoce como **Monitor**. En estos, puede haber más de un proceso dentro de una sección crítica, pero solo uno de ellos puede estar ejecutando código a la vez. Una variable de condición se ejecuta dentro de una región protegida (monitoreada) por un *lock*. Al momento de decidir esperar (*wait*), el proceso se bloquea, pero antes libera el *lock* para que otro proceso pueda entrar a la región monitoreada. Por otra parte, la acción despertar a un proceso (*signal*) hace que un proceso despertado intente tomar el *lock* de nuevo; si lo logra, podrá entrar de manera exclusiva a la región protegida; de lo contrario, se mantendrá esperando.

Administración de Memoria

Direcciones de memoria: Grandes arreglos de bytes