



Sistemas Operativos

Sistemas Operativos: PUNTEROS

John Corredor Franco, PhD
Departamento de Ingeniería de Sistemas

Julio, 2023

- Definición de punteros
 - Porque usar los punteros
 - Ventajas de los punteros
 - Operadores especiales
 - Punteros con vectores (arrays)
- Punteros Dobles (Puntero a Puntero)



- Puntero es la variable que almacena la dirección de memoria de otra variable.
- Declaración de puntero:
 - datatype *nombre puntero
- Ejemplo
 - **int *punteroA**
 - **char *ptrWord**



Asignando datos a la variable puntero

```
punteroNombre = &nombreVariable;
```

```
int *a, b = 92;
```

```
a = &b;
```

```
int *P, Quantity = 86;
```

```
P = &Quantity;
```

Variable	Value	Address
Quantity	20	500
P	500	5048



➤ Punteros

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

int main(int argc, char *argv[]){
    int variable = 123;
    int *ptr = &variable;
    printf("Valor de la Variable %d \n", variable);
    printf("Dirección de memoria de la Variable %p\n", &variable);
    printf("Dirección a la que apunta el puntero %p \n", (void *)ptr);
    printf("Valor de la Variable %i \n", *ptr);

    return 0;
}
```



➤ Porque usar los punteros

- Para retornar más de un valor de una función
- Para pasar vectores (arrays) y strings entre funciones
- Para manipular arrays fácilmente moviendo los punteros
- Para asignar memoria y accederla (Dynamic Memory Allocation)
- Para crear estructuras de datos complejas tales como listas enlazadas, en donde una ED debe contener referencias a otras ED



➤ Punteros

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

#define size 10

int main(int argc, char *argv[]){

    int *vectorPunteros[3]; //--> vector de punteros de tipo int
    int p=40, q=80, r=120; //--> variables tipo entero

    vectorPunteros[0] = &p; //--> Apuntar a la dirección de p;
    vectorPunteros[1] = &q; //--> Apuntar a la dirección de q;
    vectorPunteros[2] = &r; //--> Apuntar a la dirección de r;

    printf("\nForma de acceso al vector de punteros: \n");
    for(int i=0; i<3; i++){
        printf("Para la dirección: %p \t el valor = %d\n", vectorPunteros[i], *vectorPunteros[i]);
    }

    return 0;
}
```



Ventajas

- Un puntero permite acceder a una variable que está definida fuera de la función
- Los punteros reducen la complejidad de un programa
- Los punteros incrementan el rendimiento en los algoritmos
- Los punteros usan menos recursos de memoria
- Permite acceder a una variable sin referirse a la variable directamente



- & → Retorna la dirección de memoria del operando

- * → Retorna el valor contenido en la ubicación de memoria apuntada por el valor de la variable del puntero. Es el complemento de &.





Punteros

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

#define size 10

int main(int argc, char *argv[]){
    //***** Suma de dos valores ingresados por pantalla usando punteros *****/
    int valor1, valor2;          //--> variables de tipo entero
    int *ptrV1, *ptrV2;          //--> punteros de tipo entero
    ptrV1 = &valor1;             //--> apunte a la variable
    ptrV2 = &valor2;             //--> apunte a la variable

    printf("\nIngrese dos valores a ser sumados \n");
    scanf("%d %d", &valor1, &valor2);

    printf("\nLa suma es = %d\n", *ptrV1+*ptrV2);

    return 0;
}
```



La dirección de un elemento en un vector pueden ser expresadas de 2 maneras:

- Escribiendo el elemento actual del vector precedido del signo Ampersand (&).
- Escribiendo una expresión en la que el subíndice se añade al nombre del vector.



➤ Punteros con vectores (arrays)

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

#define size 10
Punteros

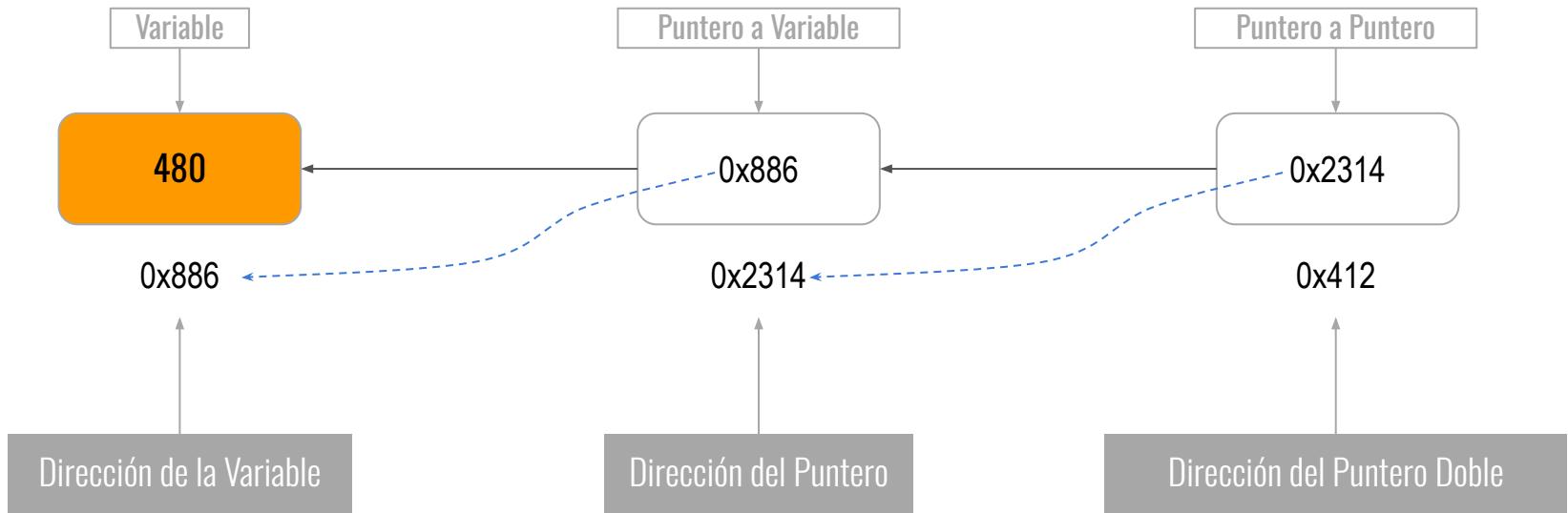
int main(int argc, char *argv[]){
    //***** Suma de dos valores ingresados por pantalla usando punteros *****/
    int vector[] = {40, 80, 120, 230};    //--> vector de enteros e inicializado
    int *ptrV1;                          //--> puntero de tipo entero
    ptrV1 = vector;                      //--> apunte a primera posición del vector
    //

    printf("\n Impresión por Indices\n");
    printf("-----\n");
    for(int i=0; i<4; i++){
        printf("Por Indices: El valor del vector[%d] = %d\n", i, ptrV1[i]);
    }

    printf("\n Impresión por Sucesor\n");
    printf("-----\n");
    for(int i=0; i<4; i++){
        printf("Por Sucesor: El valor del vector[%d] = %d\n", i, *ptrV1);
        ptrV1++;                           //--> Apunta a la siguiente posición
    }

    return 0;
}
```

➤ Doble Puntero (Puntero a Puntero)



➤ Doble Puntero (Puntero a Puntero)

En programación C, un puntero doble es un puntero que apunta a otro puntero. También se denomina puntero-a-puntero. Un puntero en C es una variable que representa la ubicación de un elemento, como una variable o una matriz. Utilizamos punteros para pasar información de ida y vuelta entre una función y su punto de referencia.



Doble Puntero (Puntero a Puntero)

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

int main(int argc, char **argv){

    int var = 480;
    int *varptr = &var;
    int **doubleptr = &varptr;
    printf("Valor de la Variable \t\t= %d\n", var );
    printf("Valor del puntero \t\t\t= %d\n", *varptr );
    printf("Valor del puntero doble \t\t= %d\n", **doubleptr);

    printf("Dirección de la Variable \t\t= %p\n", &var );

    printf("Dirección del puntero \t\t\t= %p\n", &varptr );
    printf("Valor en el puntero \t\t\t= %p\n", varptr );

    printf("Dirección del puntero doble \t\t= %p\n", *doubleptr);
    printf("Valor en el puntero doble \t\t= %p\n", doubleptr);

    return 0;
}
```

```
macbook-pro de Santiago: programas corredor$ ./doble  
Valor de la Variable = 480  
Valor del puntero = 480  
Valor del puntero doble = 480  
Dirección de la Variable = 0x16f2735ec  
Dirección del puntero = 0x16f2735e0  
Valor en el puntero = 0x16f2735ec  
Dirección del puntero doble = 0x16f2735ec  
Valor en el puntero doble = 0x16f2735e0
```



➤ Doble Puntero (Puntero a Puntero)

```
int main(int argc, char **argv){  
    int rows, cols, i, j;  
    int **matrix;  
  
    rows = (int) atof(argv[1]);  
    cols = (int) atof(argv[2]);  
  
    // Asignación de Memoria para la Matriz  
    matrix = (int **)malloc(rows * sizeof(int *));  
    for (i = 0; i < rows; i++) {  
        matrix[i] = (int *)malloc(cols * sizeof(int));  
    }  
  
    // Llenado de matriz con valores iniciales  
    for (i = 0; i < rows; i++) {  
        for (j = 0; j < cols; j++) {  
            matrix[i][j] = i * j;  
        }  
    }  
  
    // Impresión de Matriz  
    for (i = 0; i < rows; i++) {  
        for (j = 0; j < cols; j++) {  
            printf("%d ", matrix[i][j]);  
        }  
        printf("\n");  
    }  
  
    // Liberación o entrega de memoria al SO  
    for (i = 0; i < rows; i++) {  
        free(matrix[i]);  
    }  
    free(matrix);  
  
    return 0;  
}
```



➤ Doble Puntero (Puntero a Puntero)

Crear un programa que reciba como argumentos 5 datos, los cuales deben ser del tipo (entero y double).

Con los datos ingresados como argumento, han de hacer cálculos con fórmulas y anunciar que tipo de cálculo se está haciendo.





Usos de Puntero a Puntero

- **Asignación dinámica de memoria:** Los punteros dobles se utilizan a menudo para asignar memoria a las matrices, en particular a las matrices multidimensionales. Al utilizar un puntero doble, puede asegurarse de que la asignación de memoria está protegida incluso fuera de una llamada a función. Para asignar espacio a una matriz o a matrices multidimensionales de forma dinámica, necesitará un puntero doble.
- **Pasar manejadores entre funciones:** Esto es particularmente útil si necesita pasar memoria reubicable entre funciones.
- **Lista de Caracteres:** Otro uso es que si quieres tener una lista de caracteres (una palabra), puedes usar char *palabra, para una lista de palabras (una oración), puedes usar char **sentencia, y para lista de oraciones (un párrafo), puedes usar char ***párrafo.
- **Parámetros de la función principal:** Uno de los usos más comunes (que todo programador de C debería haber encontrado) como parámetros de la función main(), es decir, int main(int argc, char **argv). Aquí argv es la matriz de argumentos pasados a este programa.
- **Listas enlazadas:** Los punteros dobles se utilizan con frecuencia en las operaciones con listas enlazadas, como la inserción y la eliminación.





Sistemas Operativos

Procesos

John Corredor Franco, PhD
Departamento de Ingeniería de Sistemas

Julio, 2023

➤ Agenda

- Procesos en Sistemas Operativos
 - Estados del proceso
 - Bloque de Control de Procesos
 - Procesos versus programas
 - Planificador de procesos
 - Planificación de colas
 - Culminación de procesos
 - Tipos de planificadores
 - Control de procesos: modos de ejecución
 - Cambio de contexto
 - Operaciones sobre procesos
- Round-Robin Scheduling
- FCFS Scheduling

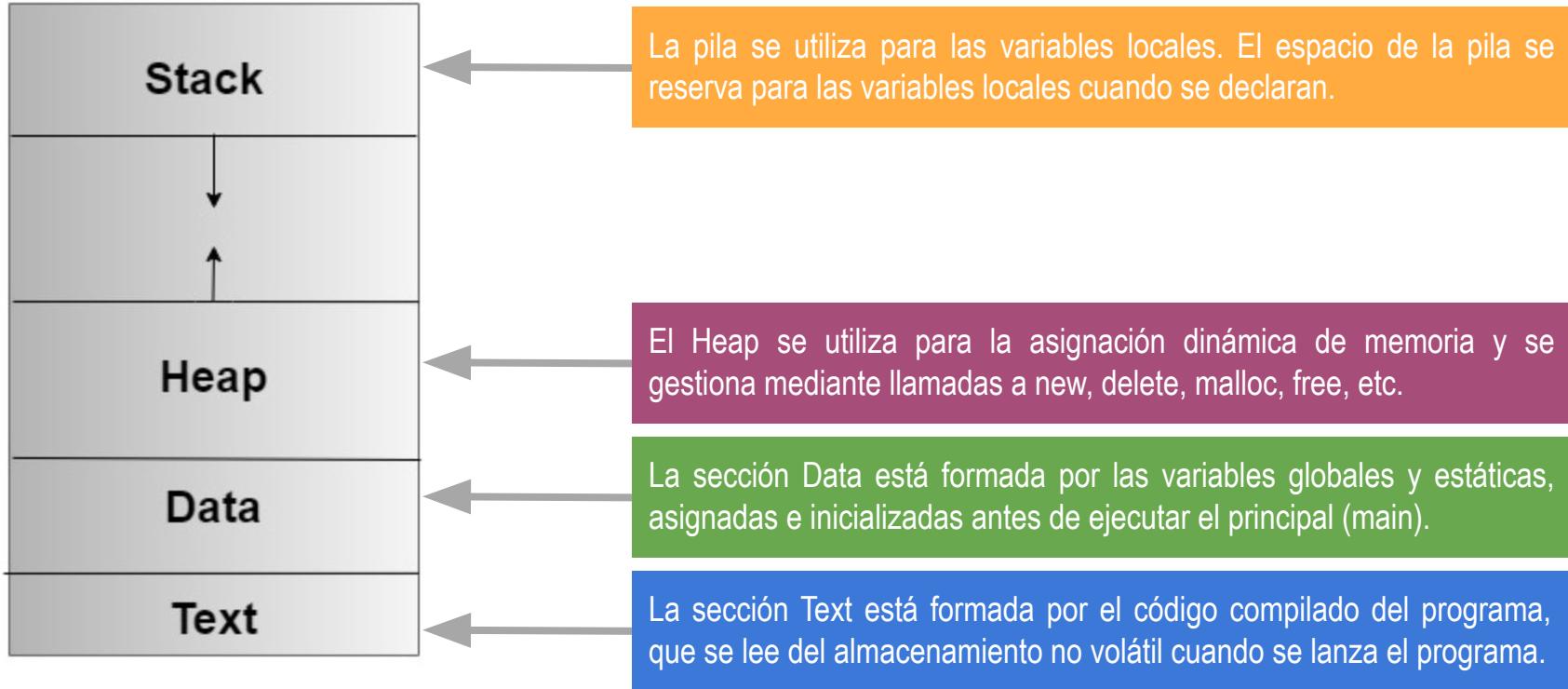


- Un proceso es un programa en ejecución que constituye la base de toda la computación. El proceso no es lo mismo que el código de un programa, sino mucho más que él. Un proceso es una entidad "activa", a diferencia del programa, que se considera una entidad "pasiva". Entre los atributos que posee el proceso se incluyen el estado del hardware, la memoria, la CPU, etc.

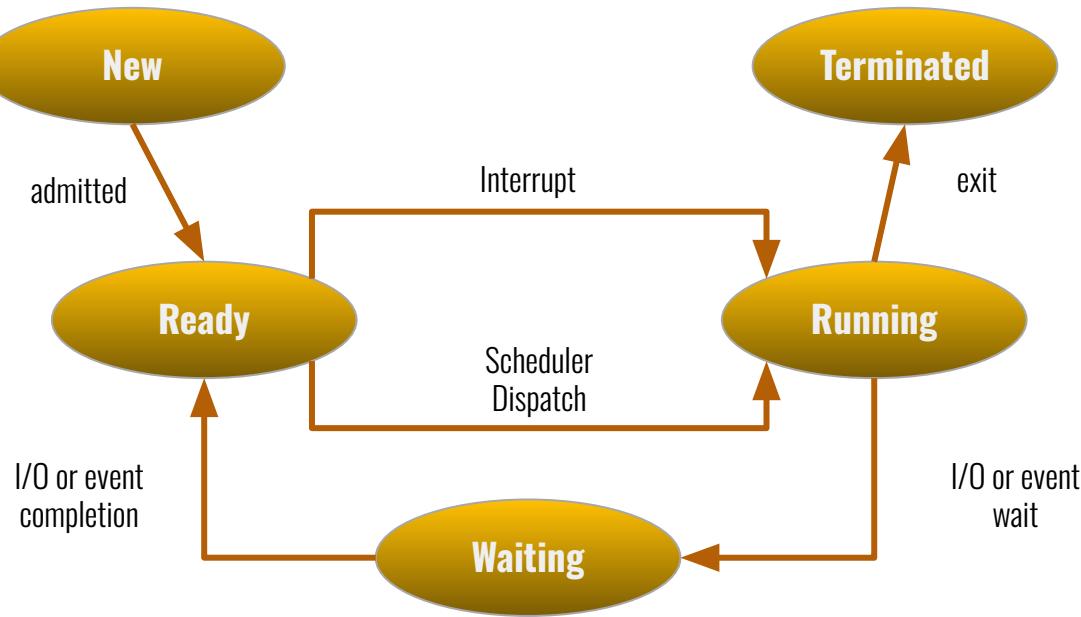




max



➤ Los diferentes estados del proceso



New: El proceso ha sido creado

Ready: El proceso está esperando ser asignado a un procesador

Running: Las instrucciones están siendo ejecutadas

Waiting: El proceso está esperando algún evento ocurra

Terminated: El proceso ha finalizado la ejecución.

➤ Bloque de control de procesos

Para cada proceso existe un Bloque de Control de Proceso, que contiene toda la información sobre el proceso. También se conoce como bloque de control de tareas. Es una estructura de datos que contiene:

Process State

Process ID y el ID del Proceso Padre

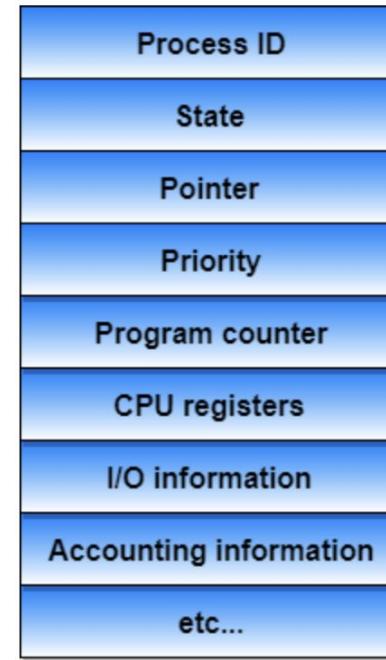
Registros del CPU y el Program Counter. El contador del programa tiene la dirección de la siguiente instrucción a ser ejecutada por ese proceso.

CPU Scheduling que tiene la información: prioridad, y punteros a colas de programación.

Memory Management information: Página de tablas o segmentos

Accounting Information: Tiempo consumido por el kernel CPU, Usuario, número de cuentas, etc.

I/O Status information: asignación de dispositivos, archivos, etc.



> Bloque de control de procesos

- Identificación del Proceso: identificador del proceso, identificador del proceso que creó al proceso actual, identificador del usuario y/o grupo propietarios del proceso.
- El Estado del Proceso: se refiere a si el actualmente se está ejecutando, está bloqueado esperando por un evento, está listo para ejecutarse, ha terminado (zombie), etc.
- El PC y registros de Control (el PSW): el PC contiene la dirección de comienzo de la próxima instrucción a ejecutar y el PSW es uno o varios registros que contiene: resultado de las más recientes operaciones aritméticas o lógicas, bits de modo de ejecución o que indican si las interrupciones estaban habilitadas o deshabilitadas.
- Los registros del CPU: Son registros visibles al usuario (de 8 a 64 en arquitecturas RISC). Incluyen acumuladores, registros índices, apuntadores a la pila, registros de propósito general, etc. Cuando ocurre un cambio de proceso, el valor del PC y de los registros se almacenan en estos campos de la PCB.
- Información de Scheduling: Esta información incluye la prioridad de un proceso (actual, default, más alta permitida, etc), información importante para el algoritmo de scheduling como por ejemplo la cantidad de tiempo que el proceso ha estado esperando por el CPU y la cantidad de tiempo que el proceso utilizó el CPU la última vez que se ejecutó; el evento que suspendió al proceso y por el cual el proceso podrá estar de nuevo listo.
- Apuntadores a otras estructuras de datos: por ejemplo al próximo proceso en la lista de procesos esperando por el disco. Zonas de memoria del proceso.



➤ Procesos vs Programas

Procesos	Programas
Es básicamente una instancia del programa informático que se está ejecutando.	Es básicamente una colección de instrucciones que realizan principalmente una tarea específica cuando son ejecutadas por el ordenador.
Tiene una vida útil más corta.	Tiene una vida útil más larga.
Requiere recursos como memoria, CPU, dispositivos de Entrada-Salida.	Se almacena en el disco duro y no necesita recursos.
Tiene una instancia dinámica de código y datos.	Tiene código estático y datos estáticos.
Básicamente, es la instancia en ejecución del código.	Es el código ejecutable.





Planificador de Procesos

- Cuando hay dos o más procesos ejecutables, el sistema operativo decide cuál se ejecuta primero.
- Un planificador se utiliza para tomar decisiones mediante el uso de algún algoritmo de programación.
- A continuación se indican las propiedades de un buen algoritmo de programación:

- El tiempo de respuesta debe ser mínimo para los usuarios.
- El número de trabajos procesados por hora debe ser máximo, es decir, un buen algoritmo de programación debe proporcionar el máximo rendimiento.
- La utilización de la CPU debe ser del 100%.
- Cada proceso debe obtener una parte justa de la CPU.



➤ Planificador de Procesos

El acto de determinar qué proceso se encuentra en estado **Ready** y debe pasar al estado en **Running** se conoce como Process Scheduling.

El objetivo principal del sistema del Process Scheduling es mantener la CPU ocupada todo el tiempo y ofrecer un tiempo de respuesta mínimo para todos los programas. Para conseguirlo, el programador debe aplicar las reglas adecuadas para el intercambio de procesos **IN** y **OUT** de la CPU.

La programación se divide en dos categorías generales:

- Programación no preventiva: Cuando el proceso en ejecución cede voluntariamente la CPU.
- Programación preferente: Cuando el sistema operativo decide favorecer a otro proceso, adelantándose al proceso actualmente en ejecución.



➤ Planificación de colas

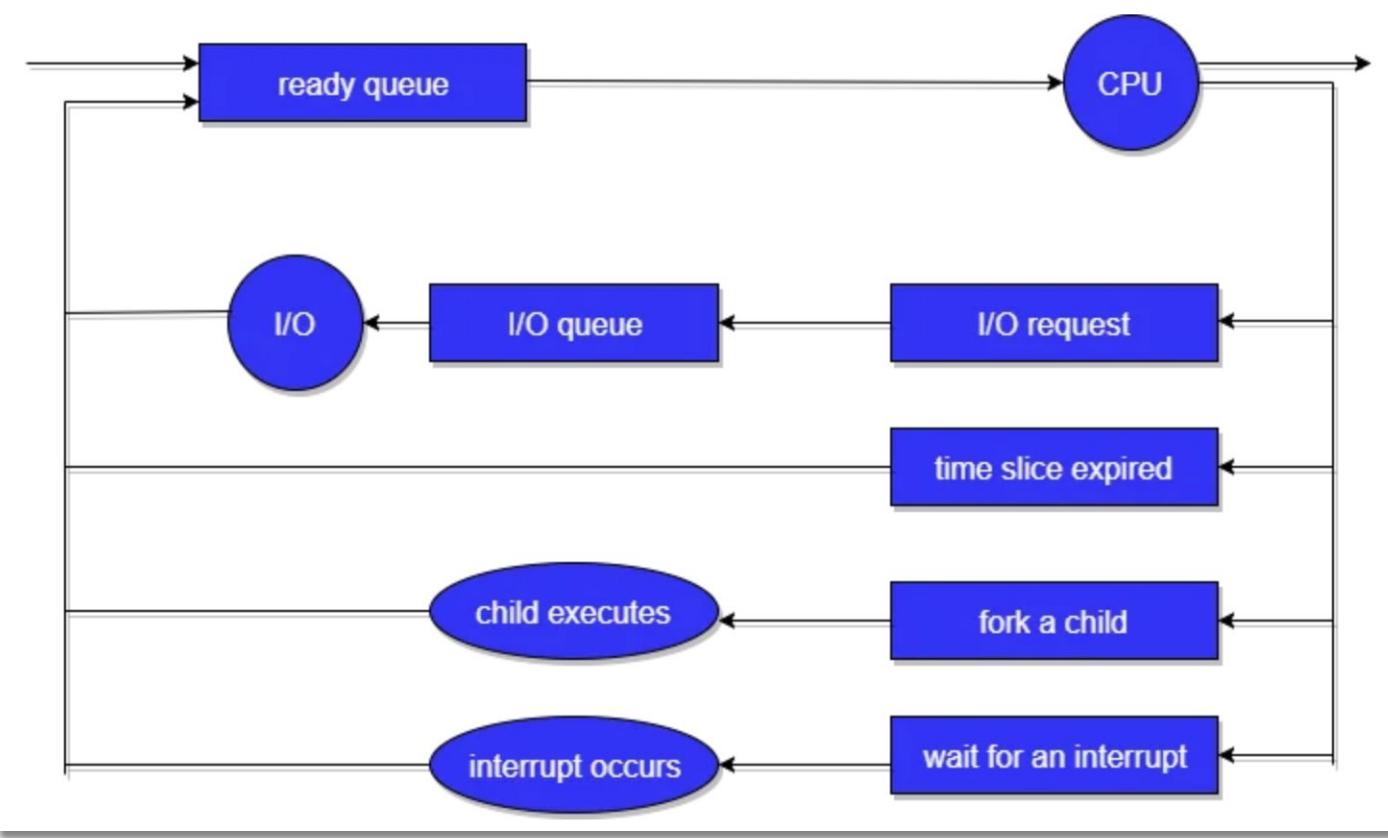
- Todos los procesos, al entrar en el sistema, se almacenan en **Job Queue**.
- Los procesos en estado Ready se colocan en la **Ready Queue**.
- Los procesos en espera de que un dispositivo esté disponible se colocan en **Device Queues**. Hay colas de dispositivos únicas disponibles para cada dispositivo de E/S.

Un nuevo proceso se coloca inicialmente en la cola de Listos. Espera en la cola de listos hasta que es seleccionado para su ejecución (o despachado). Una vez que el proceso se asigna a la CPU y se está ejecutando, puede ocurrir uno de los siguientes eventos:

- El proceso podría emitir una solicitud de E/S, y luego ser colocado en la cola de E/S.
- El proceso podría crear un nuevo subproceso y esperar su terminación.
- El proceso podría ser removido forzosamente de la CPU, como resultado de una interrupción, y ser puesto de nuevo en la cola de listos.



➤ Planificación de colas



- El proceso A está en ejecución, cuando se le termina el tiempo asignado; el despachador toma el control y manda a ejecutar el proceso B que estaba listo. El A queda ahora suspendido pero listo.
- Durante la ejecución de B se encuentra con que un requerimiento de E/S no puede ser satisfecho por lo tanto, el *dispatcher* toma el control y el proceso B es puesto en bloqueado. De esta manera el proceso C es puesto a correr. Cuando se le termina el tiempo asignado el dispatcher manda a A y luego a C porque B continúa bloqueado.

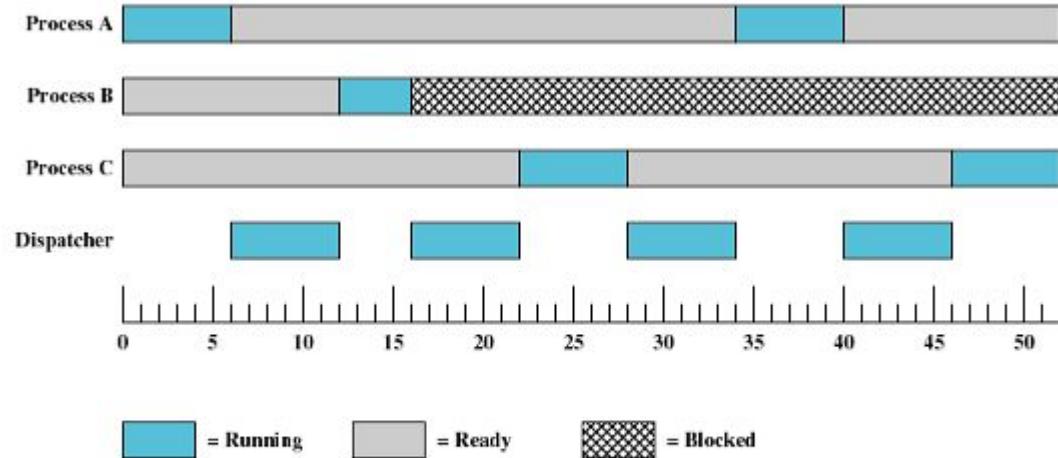


Figure 3.7 Process States for Trace of Figure 3.4

1. Long Term Scheduler:

- LTS se ejecuta con menos frecuencia.
- LTS deciden qué programa debe entrar en la job queue (cola de trabajos).
- Desde job queue, el job processor, selecciona los procesos y los carga en la memoria para su ejecución.
- El objetivo principal del job scheduler es mantener un buen grado de Multiprogramación.
- Un grado óptimo de Multiprogramación significa que la tasa media de creación de procesos es igual a la tasa media de salida de procesos de la memoria de ejecución.



2. Short Term Scheduler:

- También se conoce como **cpu scheduler** y se ejecuta con mucha frecuencia.
- El objetivo principal de este programador es mejorar el rendimiento de la CPU y aumentar la velocidad de ejecución de los procesos.

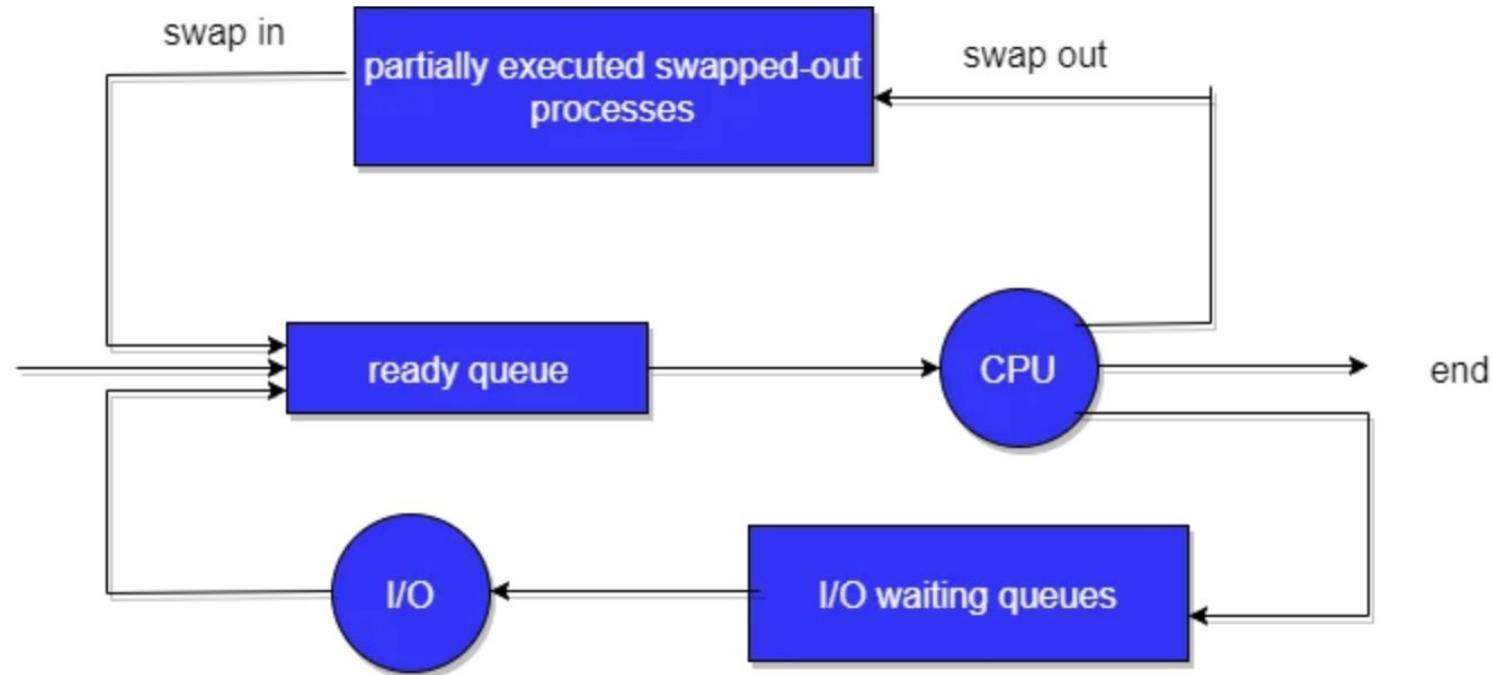


3. Medium Term Scheduler:

- Este planificador retira los procesos de la memoria (y de la contención activa por la CPU), y reduce así el grado de multiprogramación.
- En un momento posterior, el proceso puede re-introducirse en memoria y su ejecución puede continuar donde se quedó. Este esquema se denomina **swapping**.
- El proceso se intercambia hacia fuera, y más tarde se intercambia hacia dentro, por el **MTS**.
- El intercambio puede ser necesario para mejorar la combinación de procesos, o porque un cambio en los requisitos de memoria ha comprometido en exceso la memoria disponible, lo que requiere liberar memoria.



3. Medium Term Scheduler:



Incorporación de MTS al diagrama de colas

John Corredor Franco, PhD

➤ Políticas de Planificación: Round Robin

Pensado principalmente para uso en sistemas de tiempo compartido. Este enfoque es similar a la programación FCFS; sin embargo, la programación Round Robin(RR) incorpora la preferencia, permitiendo al sistema cambiar entre procesos. La preferencia permite que el proceso sea sacado a la fuerza de la CPU.

- A cada solicitud se le asocia un intervalo de tiempo fijo, conocido como **Quantum** de tiempo.
- El **job scheduler** guarda el progreso del trabajo que se está ejecutando en ese momento y pasa al siguiente trabajo presente en la cola cuando se ejecuta un proceso concreto durante un determinado quantum de tiempo.
- Ningún proceso retendrá la CPU durante mucho tiempo, lo que hace que la programación de RR sea eficiente.



- **Burst Time**
 - Todo el tiempo que necesita el proceso para ejecutarse en la CPU.
 - Todo proceso en un sistema informático necesita tiempo para su ejecución.
 - Este tiempo incluye tanto el tiempo de CPU como el de E/S.
 - El tiempo de CPU es el tiempo que tarda la CPU en realizar el proceso.
 - El tiempo de E/S, por otro lado, es el tiempo que tarda el proceso en realizar una actividad de E/S.
 - Por lo general, no tenemos en cuenta el tiempo de E/S y evaluamos únicamente el tiempo de CPU de un proceso.
- **Arrival Time**
 - Es cuando un proceso entra al **ready state** y está listo a ser ejecutado
- **Exit Time/Completion Time**
 - El momento en que un proceso finaliza su ejecución y sale del sistema se denomina tiempo de salida.



- **Turn Around Time**
 - El tiempo total empleado por el proceso desde que entra en estado listo por primera vez hasta su finalización se denomina tiempo de respuesta.
 - **TurnAround Time = Completion Time - Arrival Time**
- **Waiting Time**
 - El tiempo acumulado que pasa el proceso en estado listo esperando a la CPU se denomina tiempo de espera.
 - **Waiting Time = TurnAround Time – Burst Time**



➤ Planificador Round Robin: Ejemplo

Time Quantum: 2 ms

ID Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

PASO 1

Queue:



Gantt chart:



0 2

A los 2 seg debe haber llegado otro proceso. Según la tabla:
P2(AT = 1) y P3(AT = 2)



P1 aún no se ha completado porque ha superado el intervalo de tiempo; se moverá al final de la cola y se ejecutarán los demás procesos.



Time Quantum: 2 ms

# Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Gantt chart:



Habría llegado otro proceso. Según la tabla: $P4(AT = 3)$



Al proceso $P2$ le quedan aún 2ms restantes; se moverá al final de la cola y se ejecutarán los demás procesos.



Time Quantum: 2 ms

# Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Gantt chart:



P3: BT = QT, se completa la ejecución. Se descarta de la cola



Time Quantum: 2 ms

# Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Gantt chart:



P1: $TQ > BT$ restante, se considera el menor tiempo. Simultáneamente es descartado de la cola



Time Quantum: 2 ms

# Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Gantt chart:



P4: $TQ > BT$, se considera el menor tiempo. Simultáneamente es descartado de la cola



Time Quantum: 2 ms

# Process	Arrival Time (AT)	Burst Time (BT)
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Gantt chart:

*La cola queda vacía*

- **Concurrencia explícita**

- Contienen pasos de procesamiento independientes (a nivel de bloque, sentencia o nivel de expresión) que pueden ejecutarse en paralelo;
- Desencadenan operaciones de dispositivos que pueden proceder en paralelo con la ejecución del programa.

- **Concurrencia implícita**

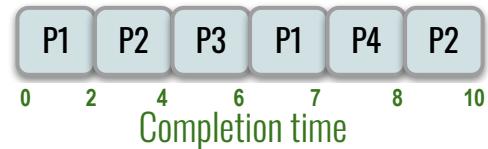
- El comportamiento concurrente es especificado por el diseñador del programa



- Un **programa concurrente** define acciones que pueden ser ejecutadas simultáneamente
- Un **programa paralelo** es un **programa concurrente** que está diseñado para ser ejecutado en hardware paralelo
- Un **programa distribuido** es un **programa paralelo** diseñado para ser ejecutado sobre una red de procesadores autónomos que no comparten memoria principal (a nivel hardware).



➤ Planificador Round Robin: Ejemplo



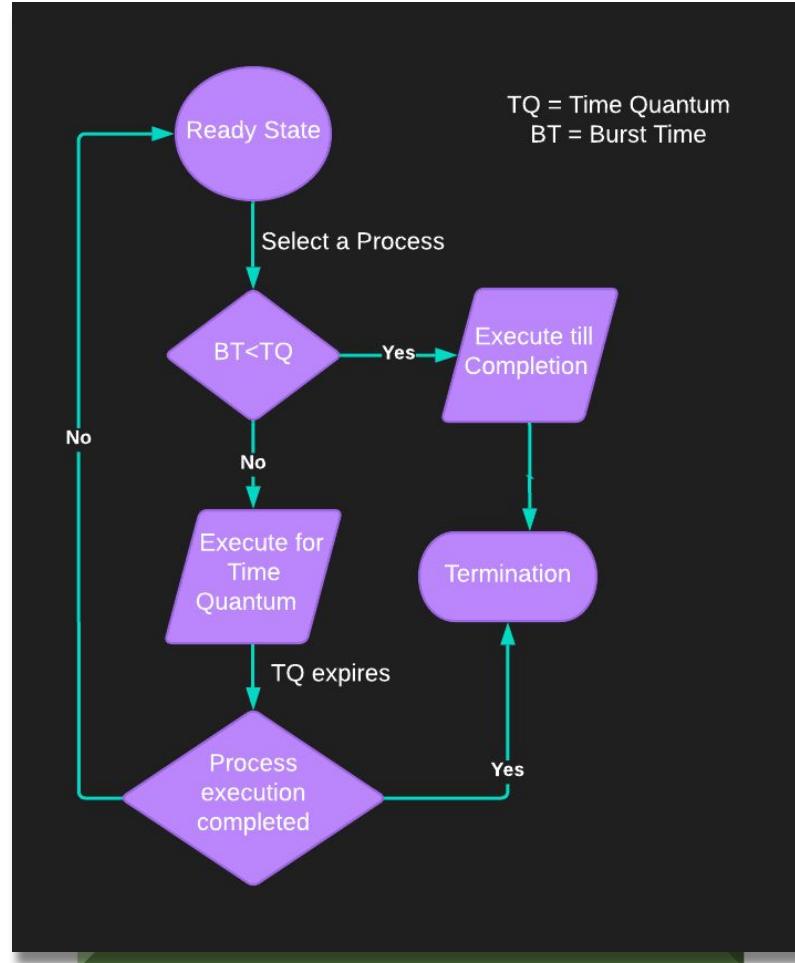
Time Quantum: 2 ms

# P	AT	BT
P1	0	3
P2	1	4
P3	2	2
P4	3	1

Completion time:	Turn Around Time: (CT - AT)	Waiting Time: (TAT - BT)
P1: P2: P3: P4:	P1: P2: P3: P4:	P1: P2: P3: P4:

# Proceso	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turn Around Time (TAT)	Waiting Time (WT)
P1	0	3	7	7	4
P2	1	4	10	9	5
P3	2	2	6	4	2
P4	3	1	8	5	4

➤ Planificador Round Robin: Flow chart



Ventajas RR	Desventajas RR
<ul style="list-style-type: none">• Dado que la CPU sirve a cada proceso durante un quantum de tiempo fijo, a todos los procesos se les asigna la misma prioridad.• Como cada ciclo round-robin da a cada proceso un tiempo predeterminado para ejecutarse, no se produce inanición. No se pasa por alto ningún proceso.	<ul style="list-style-type: none">• La elección de la longitud del <i>Quantum Time</i> tiene un impacto considerable en el rendimiento en RR. Si el quantum de tiempo es mayor de lo necesario, se comporta de forma similar a FCFS.• Si el quantum de tiempo es menor de lo necesario, aumenta el número de veces que la CPU cambia de un proceso a otro. Como resultado, afecta la eficiencia de la CPU.



- ¿Qué ocurre cuando el intervalo de tiempo es grande?
 - En un sistema de tiempo compartido con intervalos de tiempo muy amplios, el algoritmo de programación round-robin se transforma en el algoritmo de programación First come, First served.
- En términos de tiempo de respuesta, ¿es round-robin superior a FCFS? (Gate 2010)
 - Sí, una programación round-robin dará un mejor tiempo de respuesta que FCFS porque en FCFS, durante el tiempo de ejecución del proceso, éste se ejecuta sólo cuando se completa su tiempo de ráfaga. En cambio, cada proceso se ejecutará hasta el quantum de tiempo en un round-robin.
- ¿Por qué la política de programación round-robin es la más adecuada para los sistemas operativos de tiempo compartido?
 - La programación Round Robin funciona sobre el quantum de tiempo; después de un cierto tiempo, cada proceso recupera las unidades CPU para su finalización, el mismo fenómeno utilizado en un sistema de tiempo compartido. Por lo tanto, Round Robin es lo mejor para un sistema de tiempo compartido.



FCFS: First-Come-First-Served (o FIFO: Primero en entrar, primero en salir)

- El planificador ejecuta los trabajos hasta su finalización por orden de llegada.
- En los primeros planificadores FCFS, el trabajo no renunciaba a la CPU incluso cuando estaba haciendo E/S.
- Supondremos un planificador FCFS que se ejecuta cuando los procesos están bloqueados en E/S, pero que no es preferente, es decir, el trabajo mantiene la CPU hasta que se bloquea (digamos en un dispositivo de E/S).



Ventajas RR	Desventajas RR
<ul style="list-style-type: none">• Simple	<ul style="list-style-type: none">• El tiempo medio de espera es muy variable, ya que los trabajos cortos pueden esperar detrás de los largo trabajos largos.• Puede dar lugar a un solapamiento deficiente de E/S y CPU, ya que los procesos ligados a la CPU obligarán a los procesos ligados a E/S a esperar a la CPU. obligan a los procesos de E/S a esperar a la CPU, dejando ociosos los dispositivos de E/S.



5 jobs, 100 seconds each, QT (time slice) 1 second, context switch time of 0 sec

Job	Length	Completion Time		Waiting Time	
		FSFC	RR	FCFS	RR
1					
2					
3					
4					
5					
AVERAGE					



➤ Planificador Ejercicios

- A) 5 jobs, 50, 40, 30, 20 y 10 seconds each, QT (time slice)
1 second, context switch time of 0 sec. Entregar la Tabla Llena

Job	Length	Completion Time		Waiting Time	
		FSFC	RR	FCFS	RR
1					
2					
3					
4					
5					
AVERAGE					

B) Round Robin
TQ = 2
Cantidad procesos = 5
Arrival Time = 1, 2, 3, 4, 5
Burst Time = 4, 6, 7, 8, 5

Hacer una tabla que presente
Completion Time
Turn Around Time
Waiting Time



➤ Planificador Ejercicios

C) 5 jobs, CPU SCHEDULING POLICY is Round Robin with time quantum = 3, arrival time and burst time are given below

PID	Arrival Time	Burst Time	Exit Time	Turn Around Time	Waiting Time
P1	5	5	32	27	22
P2	4	6	27	23	17
P3	3	7	33	30	23
P4	1	9	30	29	20
P5	2	2	6	4	2
P6	6	3	21	15	12
Average time			21.33	16	

Ready queue: P3, P1, P4, P2, P3, P6, P1, P4, P2, P3, P5, P4



❖ Modo de usuario

- Modo menos privilegiado
- Los programas de usuario ejecutan normalmente en ese modo

❖ Modo de sistema, control, o kernel

- Modo más privilegiado
- Kernel del sistema operativo



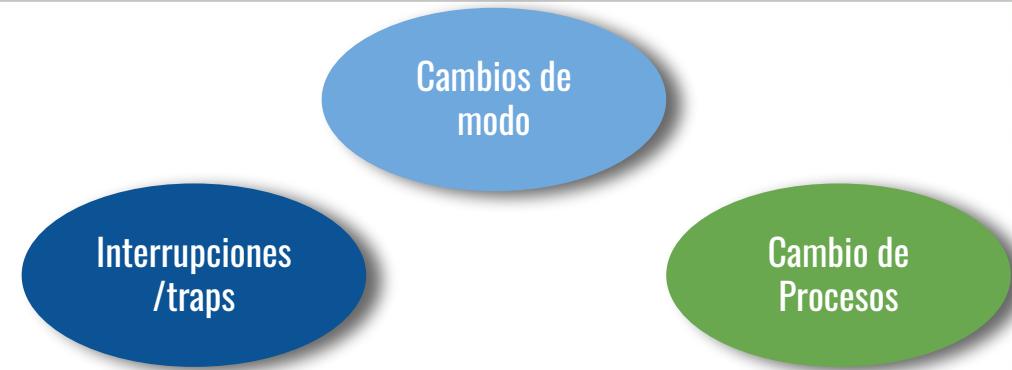
¿Cuando cambiar de modo?

Cambios de modo

Interrupciones /traps

Cambio de Procesos

- ❖ **Interrupción de reloj:**
 - Si el SO determina que el proceso que está en ejecución, se ha estado ejecutando durante la fracción máxima de tiempo permitida, el proceso debe pasar al estado *Ready* y se debe expedir otro proceso (CAMBIO de PROCESO)
- ❖ **Interrupción de E/S:**
 - Si la acción constituye un suceso que están esperando uno o más procesos, entonces el SO traslada todos los procesos bloqueados correspondientes al estado *Ready* o *Ready* suspendido (no necesariamente se cambia al proceso actual)
- ❖ **Fallo de memoria:**
 - Una referencia a una dirección de memoria virtual no está en memoria principal. El SO puede llevar a cabo un cambio de proceso; el proceso que cometió el fallo de memoria se pasa a estado Bloqueado.



- ❖ **Trap:**
 - Error de excepción
 - Si es fatal puede causar que el proceso pase al estado terminado (CAMBIO DE PROCESO)
- ❖ **Trap - System Call**
 - Ejemplo: Leer datos de un archivo. El proceso de usuario puede pasar al estado bloqueado. No todas las llamadas al sistema producen bloqueo del proceso (SI EL PROCESO SE BLOQUEA, HAY CAMBIO DE PROCESO)



¿Cuando cambiar de proceso?

Un cambio de proceso puede ocurrir en cualquier momento en que el SO haya obtenido el control del proceso que se está ejecutando en ese momento. Los posibles eventos que dan el control al SO son:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

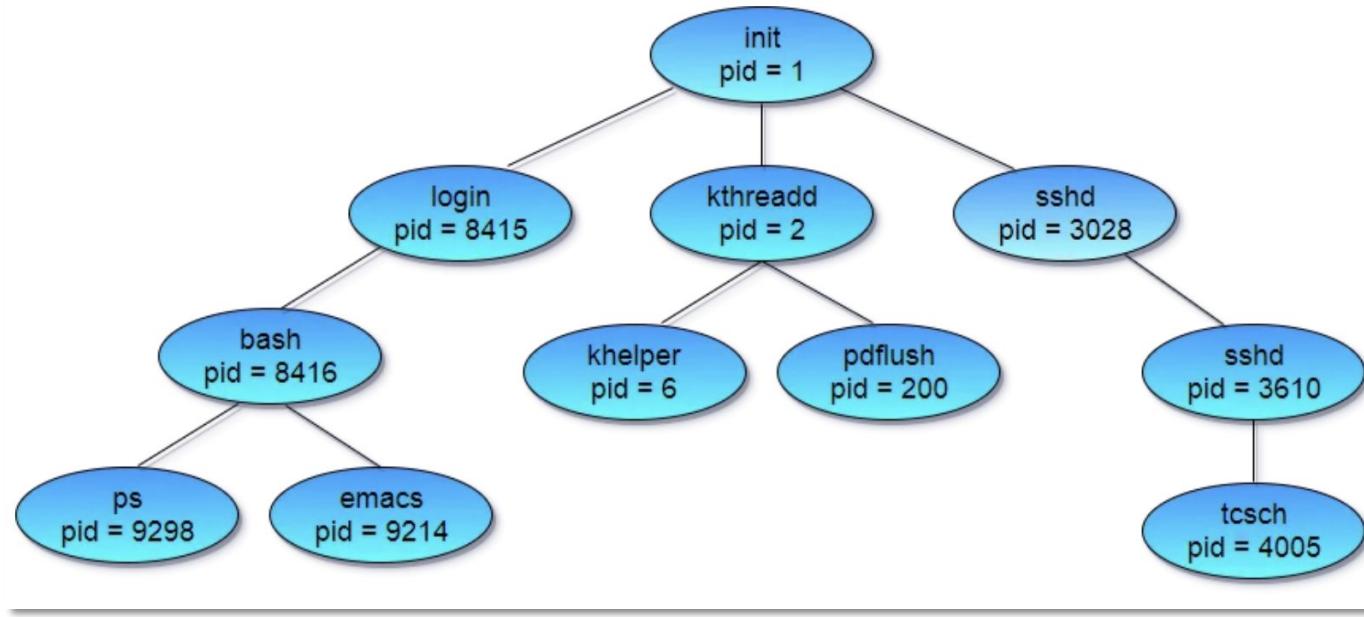
➤ Cambio de Contexto

1. Cambiar la CPU a otro proceso.
 - Se requiere guardar el estado del proceso anterior y cargar el estado guardado para el nuevo proceso.
2. El contexto de un proceso se representa en el **Bloque de Control de Proceso (PCB)** de un proceso; incluye el valor de los registros de la CPU, el estado del proceso y la información de gestión de memoria. Cuando se produce un cambio de contexto, el Kernel guarda el contexto del proceso anterior en su **PCB** y carga el contexto guardado del nuevo proceso programado para ejecutarse.
3. El tiempo de cambio de contexto es **overhead**, porque el sistema no realiza ningún trabajo útil mientras se produce el cambio. Su velocidad varía de una máquina a otra, en función de la velocidad de la memoria, el número de registros que deben copiarse y la existencia de instrucciones especiales (como una única instrucción para cargar o almacenar todos los registros). Las velocidades típicas oscilan entre 1 y 1000 microsegundos.
4. El cambio de contexto se ha convertido en un cuello de botella para el rendimiento, hasta el punto de que los programadores utilizan nuevas estructuras (hilos) para evitarlo siempre que sea posible.



- El proceso que crea otro proceso se denomina **padre** del otro proceso, mientras que el subprocesso creado se denomina **hijo**.
- A cada proceso se le asigna un identificador entero, denominado **identificador de proceso o PID**.
 - El PID padre (PPID) también se almacena para cada proceso.
- En un sistema UNIX típico, el programador de procesos se denomina **sched**, y se le asigna el PID 0.
 - Lo primero que hace al arrancar el sistema es lanzar **init**, que le da a ese proceso el PID 1.
 - Después, **init** lanza todos los procesos del sistema.
 - Además, **init** lanza todos los demonios del sistema y los inicios de sesión de usuario, y se convierte en el **padre último** de todos los demás procesos.





Un proceso hijo puede recibir cierta cantidad de recursos compartidos con su padre dependiendo de la implementación del sistema. Para evitar que los procesos hijo consuman la totalidad de un determinado recurso del sistema, los procesos hijo pueden estar limitados o no a un subconjunto de los recursos asignados originalmente al proceso padre.

Hay dos opciones para el proceso padre después de crear el proceso hijo :

- Esperar a que el proceso hijo termine antes de continuar. El proceso padre hace una llamada al sistema **wait()**, ya sea para un proceso hijo específico o para cualquier proceso hijo en particular, lo que hace que el proceso padre se bloquee hasta que vuelva **wait()**. Los shells UNIX normalmente esperan a que sus hijos finalicen antes de emitir un nuevo **prompt**.
- Ejecutar concurrentemente con el hijo, continuando el proceso sin esperar. Cuando un shell UNIX ejecuta un proceso como tarea en segundo plano, esta es la operación vista. También es posible que el padre se ejecute durante un tiempo y espere al hijo más tarde, lo que podría ocurrir en una especie de operación de procesamiento paralelo.

También hay dos posibilidades en cuanto al espacio de direcciones del nuevo proceso:

- El proceso hijo es un duplicado del proceso padre.
- El proceso hijo tiene un programa cargado en él.



Haciendo la llamada al sistema **exit**(system call), típicamente devolviendo un int, los procesos pueden solicitar su propia terminación. Este int se pasa al proceso padre si está haciendo un **wait()**, y es típicamente cero en caso de finalización exitosa y algún código distinto de cero en caso de algún problema.

Los procesos también pueden ser terminados por el sistema por una variedad de razones, incluyendo :

- La incapacidad del sistema para entregar los recursos necesarios del sistema.
- En respuesta a un comando **KILL** u otras interrupciones de proceso no manejadas.
- Un padre puede matar a sus hijos si la tarea asignada a ellos ya no es necesaria, es decir, si la necesidad de tener un hijo termina.
- Si el padre sale, el sistema puede o no permitir que el hijo continúe sin un padre (En los sistemas UNIX, los procesos huérfanos son generalmente heredados por init, que entonces procede a matarlos).



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

int main(int argc, char *argv[]) {
    int x;
    for(x = 0; x < 10; x++){
        fork();
        printf("The process ID (PID): %d \n",getpid());
    }
    return 0;
}
```

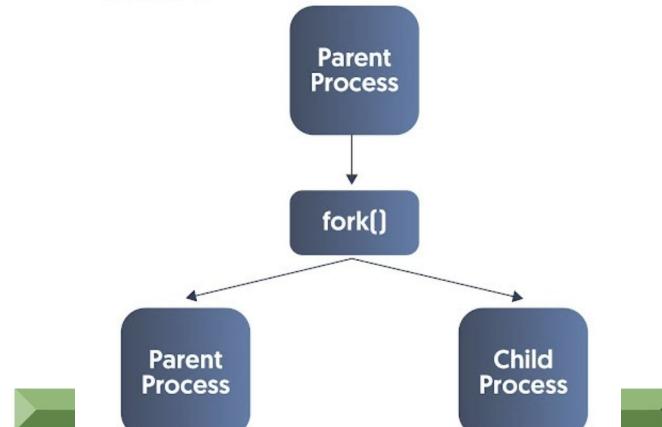
➤ Fork() function

En el contexto de lenguajes como C, C++, shell script, etc., fork se refiere a: Crear un proceso hijo duplicando un proceso padre, y entonces ambos se ejecutan concurrentemente.

La función fork()

La llamada al sistema **fork()** crea un nuevo proceso a partir del proceso que lo llama duplicándolo. El proceso padre hace la llamada al sistema **fork()**, y su proceso hijo se forma como resultado de esa llamada si tiene éxito.

Fork in C



➤ Fork() function

La función fork() no toma argumentos. Simplemente crea un proceso hijo y devuelve un ID de proceso.

Si una llamada a fork tiene éxito:

- El SO hará dos copias idénticas de los espacios de direcciones para los procesos padre e hijo. Así que los procesos padre e hijo tienen diferentes espacios de direcciones.
- Una variable **local** es:
 - Declarada dentro del proceso
 - Creada cuando se inicia el proceso
 - Se pierde cuando el proceso termina
- Una variable **global** es:
 - Declarada fuera del proceso
 - Creada al iniciarse el proceso
 - Se pierde al finalizar el programa



Si una llamada a fork tiene éxito (continuación):

-
- El ID del proceso, es decir, el PID del proceso hijo creado, se devuelve al proceso padre.
 - (En caso de fallo, se devuelve -1 al proceso padre).
- Se devuelve cero al proceso hijo.
 - (Si falla, el proceso hijo no se crea.) Si un proceso hijo sale en ese instante o se interrumpe, se envía una señal SIGCHLD al proceso padre.
- Ambos procesos, padre e hijo, ejecutan independientemente los comandos subsiguientes después de la llamada al sistema fork().
- Procesos entre padre e hijo, a través del valor que devuelve fork
 - **Valor negativo:** La llamada a fork ha fallado.
 - **Valor cero:** Este valor se devuelve al hijo que se ha creado recientemente.
 - **Valor positivo:** El padre recibió el PID del proceso hijo como valor de retorno.



> Fork() function

```
int main(int argc, char *argv[]) {  
    int processID= fork();  
  
    if(processID>0) {  
        printf("fork() devuelve un valor +ve. Este es el 'proceso padre' con ID: %d \n",getpid());  
    }  
    else if(processID==0) {  
        printf("fork() devuelve un valor 0. Este es un 'proceso hijo' recien creado con ID: %d \n",getpid());  
        printf("El 'proceso padre' de este 'proceso hijo' tiene el ID: %d\n",getppid());  
    }  
    else {  
        printf("fork() devuelve un valor -ve, por lo que la llamada al sistema fork() falló\n");  
    }  
  
    printf("\n\nEsta es una única impresión. Si la llamada al sistema fork() ha tenido éxito \n");  
    printf("tanto el 'proceso padre' como el 'proceso hijo' se ejecutarán simultáneamente, \n");  
    printf("y esta impresión aparecerá dos veces.\n");  
  
    return 0;  
}
```

Transición de diapositivas

> □ Cubo

Animaciones de objetos

+ Selecciona un objeto

Reproducir

> Fork() function

```
int main(int argc, char *argv[]) {
    //function

    int processID= fork();

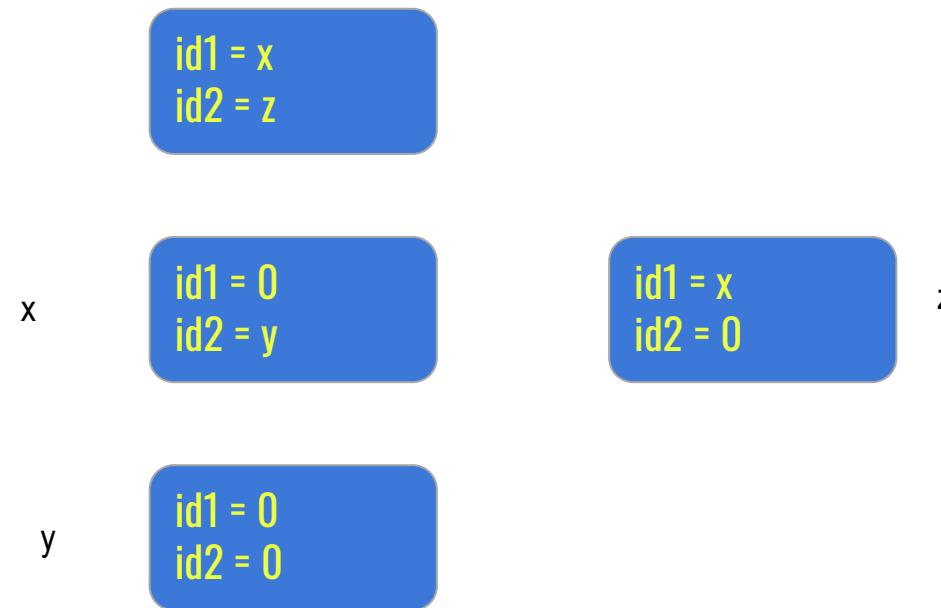
    if(processID>0) {
        printf("\n 'proceso padre'.... \n");
    }
    else if(processID==0) {
        printf("\n 'proceso hijo' recien creado \n");
    }
    else {
        printf("\n llamada al sistema fork() falló\n");
    }

    printf("\n\n Imprimiendo ... \n");
    for (int i = 0; i < 5; i++)
        printf(" %d ", i);
    printf("\n Fin \n");

    return 0;
}
```

in Corredor Franco, PhD

➤ Fork() function



fork() - it creates a child process, this child process has a new PID and PPID.

```
// No toma ningún parámetro, devuelve  
// valores enteros. Puede devolver valores enteros negativos,  
// positivos o cero valores enteros.
```

pipe() is a Unix, Linux system call that is used for inter-process communication.

```
int pipe(int pipefd[2]);
```



Pipes

```
int main(void){  
    int fd[2]; //Se usa para almacenar los 2 extremos del pipe  
    int nbytes;  
    pid_t childpid;  
    char frase_enviada[] = "Hola desde Entrada a Pipe\n";  
    char buffer_lectura[80];  
  
    pipe(fd);  
  
    if((childpid = fork()) == -1){  
        perror("fork");  
        exit(1);  
    }  
  
    if(childpid == 0){  
        /* El proceso hijo cierra el lado de entrada del pipe */  
        close(fd[0]);  
  
        /* Envía "frase_enviada" a la salida del pipe */  
        write(fd[1], frase_enviada, (strlen(frase_enviada)+1));  
        exit(0);  
    }  
    else {  
        /* El proceso padre cierra la salida del pipe */  
        close(fd[1]);  
  
        /* Lee en un string el buffer_lectura */  
        nbytes = read(fd[0], buffer_lectura, sizeof(buffer_lectura));  
        printf("Opción recibida: %s", buffer_lectura);  
    }  
  
    return(0);  
}
```

John Corredor Franco, PhD



Pipes

```
9 int main() {
10     // Se usan dos pipes: Primero para enviar una oración al proceso padre
11     // Segundo para la oración concatenada al proceso hijo
12     int fd1[2];
13     int fd2[2];
14
15     char input_str00[100];
16     char input_str01[100];
17
18     pid_t p;
19
20     printf("Ingrese primera frase (enter para continuar) \n");
21     scanf("%s", input_str00);
22     printf("Ingrese segunda frase (enter para continuar) \n");
23     scanf("%s", input_str01);
24
25     if ((pipe(fd1) == -1) || (pipe(fd2) == -1)) {
26         fprintf(stderr, "Pipe Failed");
27         return 1;
28     }
29
30     p = fork();
31
32     if (p < 0) {
33         fprintf(stderr, "fork Failed");
34         return 1;
35     }
36
37     // Parent process
38     else if (p > 0) {
39         char concat_str[100];
40
41         close(fd1[0]);
42
43         write(fd1[1], input_str01, strlen(input_str01) + 1);
44         close(fd1[1]);
```

```
46     wait(NULL);
47
48     close(fd2[1]);
49
50     read(fd2[0], concat_str, 100);
51     printf("Concatenated string %s\n", concat_str);
52     close(fd2[0]);
53 }
54
55 // child process
56 else {
57     close(fd1[1]);
58
59     char concat_str[100];
60     read(fd1[0], concat_str, 100);
61
62     int k = strlen(concat_str);
63     int i;
64     for (i = 0; i < strlen(input_str0); i++)
65         concat_str[k++] = input_str0[i];
66
67     concat_str[k] = '\0'; // string finaliza '\0'
68
69     close(fd1[0]);
70     close(fd2[0]);
71
72     write(fd2[1], concat_str, strlen(concat_str) + 1);
73     close(fd2[1]);
74
75     exit(0);
76 }
77 }
```





Sistemas Operativos

Concurrencia: POSIX

John Corredor Franco, PhD
Departamento de Ingeniería de Sistemas

Julio, 2023

➤ Agenda

- Conceptos básicos
- Threads
- Pthreads
- Diseño de Programas con Pthreads
- Pthreads API
- Compilación de pthreads en las diferentes plataformas
- Rutinas básicas
- Codificación



- **Concurrencia explícita**

- Contienen pasos de procesamiento independientes (a nivel de bloque, sentencia o nivel de expresión) que pueden ejecutarse en paralelo;
- Desencadenan operaciones de dispositivos que pueden proceder en paralelo con la ejecución del programa.

- **Concurrencia implícita**

- El comportamiento concurrente es especificado por el diseñador del programa



➤ Conceptos básicos

- Un **programa concurrente** define acciones que pueden ser ejecutadas simultáneamente
- Un **programa paralelo** es un **programa concurrente** que está diseñado para ser ejecutado en hardware paralelo
- Un **programa distribuido** es un **programa paralelo** diseñado para ser ejecutado sobre una red de procesadores autónomos que no comparten memoria principal (a nivel hardware).



- Vista desde SO
 - Flujo de instrucciones independiente que es planificado y ejecutado por el SO
- Vista desde el desarrollador de software
 - Un *thread* puede ser considerado como un ‘*procedimiento*’ que se ejecuta independiente del programa principal.
 - Programa secuencial
 - Un único flujo de instrucciones en un programa
 - Programa Multi-*thread*
 - Un programa con múltiples flujos
 - Multiple threads usan multiple cores/CPUs



- Computer games
 - each thread controls the movement of an object.
- Scientific simulations
 - Hurricane movement simulation: each thread simulates the hurricane in a small domain.
 - Molecular dynamic: each thread simulates a subset of particulars.
 -
- Web server
 - Each thread handles a connection.
 -



- **Process context**

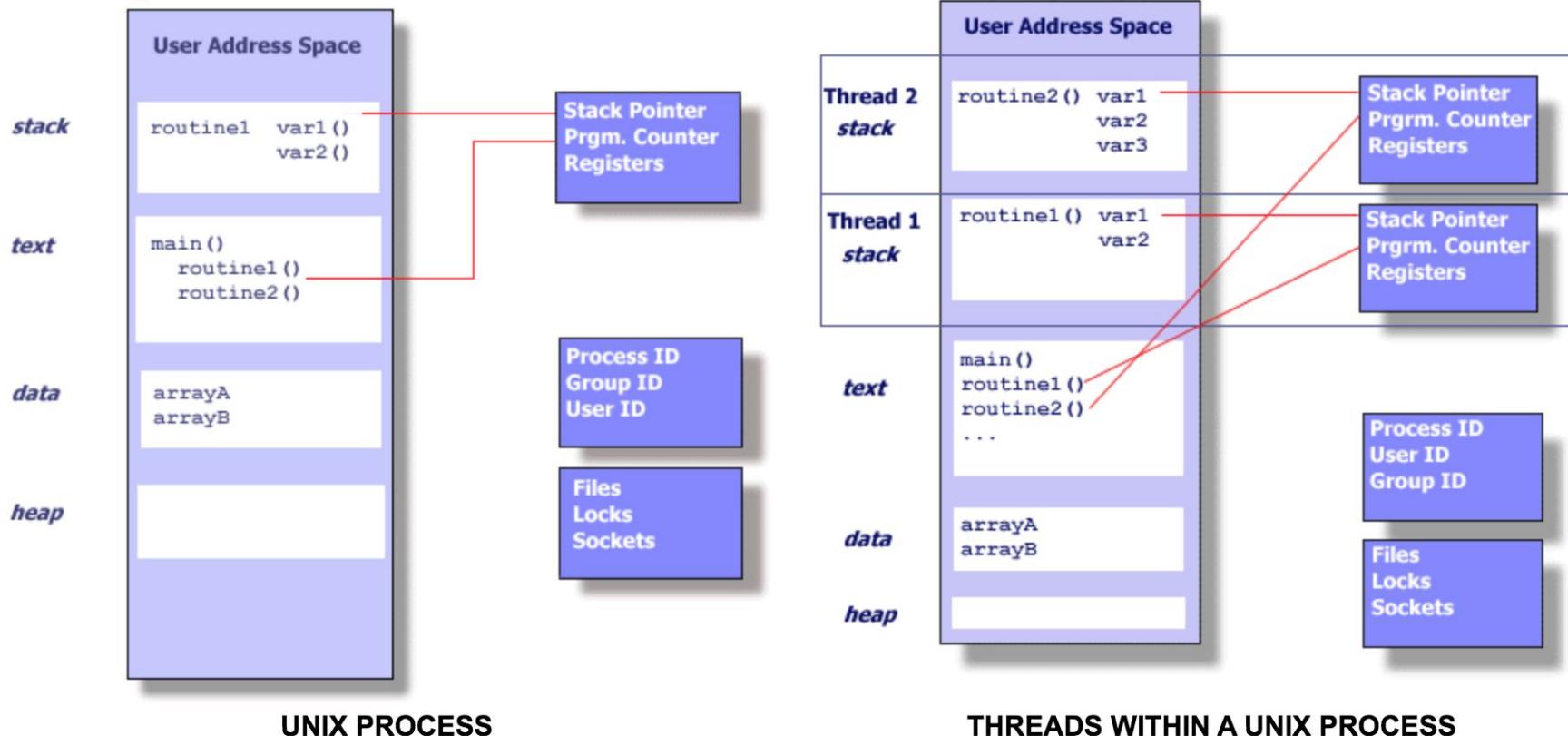
- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers (including PC)
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools
- Two parts in the context: self-contained domain (protection) and execution of instructions.



- What are absolutely needed to support a stream of instructions, given the process context?
 - Process ID, process group ID, user ID, and group ID
 - Environment
 - Working directory.
 - Program instructions
 - Registers (including PC)
 - Stack
 - Heap
 - File descriptors
 - Signal actions
 - Shared libraries
 - Inter-process communication tools



➤ Procesos e hilos



> ¿Qué es un *thread*?

- Técnicamente, un hilo es definido como un flujo de instrucciones independiente del programa principal, que es planificado y ejecutado por el sistema operativo. But what does this mean?
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (*a.out*) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.



> ¿Qué es un *thread*?

- Existe con los procesos
- Muere si el proceso muere
- Usa los recursos del proceso
- Duplica solo los recursos esenciales para que el SO los planifique independientemente.
- Cada *thread* mantiene
 - Stack
 - Registers
 - Scheduling properties (e.g. priority)
 - Set of pending and blocked signals (to allow different react differently to signals)
 - Thread specific data



- En resumen, en un entorno UNIX un *thread*:
 - Existe dentro de un proceso y usa los recursos del proceso.
 - Tiene su propio flujo de control independiente siempre que exista su proceso padre y el SO lo soporte.
 - Duplica sólo los recursos esenciales que necesita para ser planificado de forma independiente.
 - Puede compartir los recursos con otros *threads* que actúan igualmente independientes (y dependientes).
 - Muere si el proceso padre muere - o algo similar
 - Es “ligero” porque la mayor parte del *overhead* ya se ha realizado mediante la creación del proceso.
- Porque los hilos dentro del mismo proceso comparten recursos:
 - Los cambios realizados por un *thread* en los recursos compartidos en el sistema (ejemplo: cierre de un fichero) serán vistos por todos los demás *threads*.
 - Dos punteros con el mismo valor apuntan a los mismos datos.
 - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.



- **Estado compartido**
 - Las variables globales se comparten entre los threads. Cambios accidentales pueden ser fatales.
- **Muchas funciones de la library no son seguras para los threads**
 - Funciones de la Library que devuelven punteros a la memoria interna estática. E.g. gethostbyname()
- **Falta de robustez**
 - Crash in one thread will crash the entire process.



- Históricamente, los fabricantes de hardware implementan sus propias versiones propietarias de *threads*
 - Dificulta a los programadores el desarrollo de aplicaciones portables con threads.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
 - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
 - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
 - Most hardware vendors now offer Pthreads in addition to their proprietary API's.

- *Pthreads* se definen como un conjunto de tipos de programación y llamadas a procedimientos en lenguaje C, implementados con la interfaz #include <pthread.h> a una *library* de *threads*.



- Light-weight
 - Lower overhead for thread creation
 - Lower Context Switching Overhead
 - Fewer OS resources



➤ Pthreads: comparativa fork() pthread_created()

Tiempos en segundos de 50k creaciones procesos/*threads*, sin flags de optimización, sobre múltiples núcleos, trabajando en un problema al mismo tiempo.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6



> Pthreads Ventajas (2)

- **Comunicaciones eficientes/intercambio de datos:**
 - La motivación principal para considerar el uso de ***pthreads*** en un entorno de HPC es lograr un rendimiento óptimo. En particular, si una aplicación está usando MPI para comunicaciones en el nodo, existe la posibilidad de que el rendimiento pueda ser mejorado usando ***pthreads*** en su lugar.
 - Las bibliotecas MPI suelen implementar la comunicación de tareas en el nodo a través de la memoria compartida, lo que implica al menos una operación de copia en memoria (proceso a proceso).
 - En el caso de los ***pthreads*** no se requiere una copia de memoria intermedia porque los hilos comparten el mismo espacio de direcciones dentro de un solo proceso. No hay transferencia de datos en sí. Puede ser tan eficiente como simplemente pasar un puntero.
 - En el peor de los casos, las comunicaciones de los ***pthreads*** se convierten más en un problema de ancho de banda de caché a CPU o de memoria a CPU. Estas velocidades son mucho más altas que las comunicaciones de memoria compartida MPI.

> Pthreads Ventajas (3) Otras razones

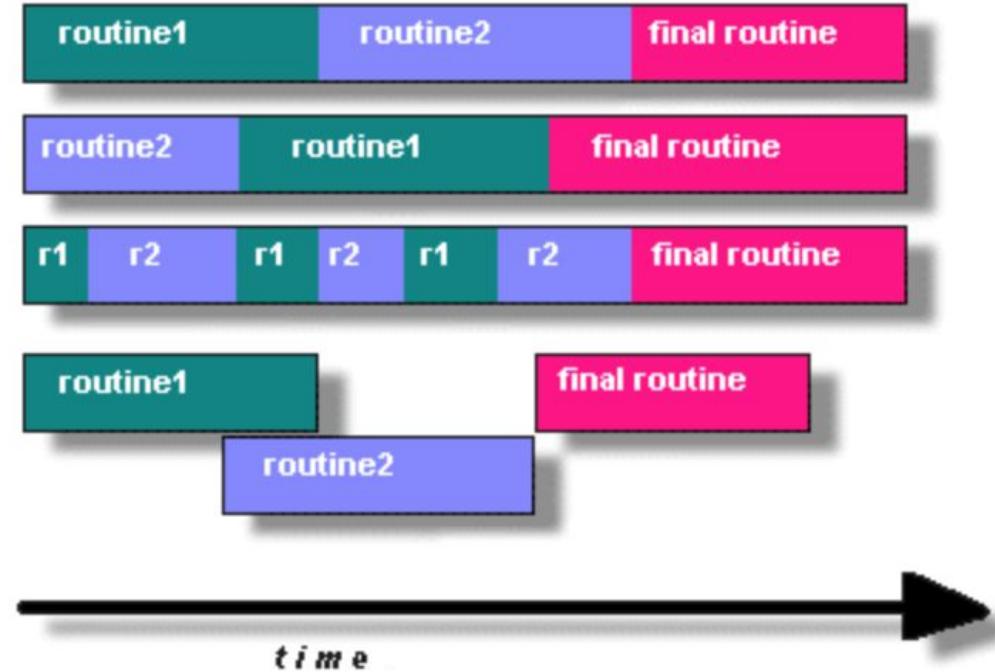
- Las aplicaciones que usen *pthreads* ofrecen mayor ganancia de rendimiento y ventajas prácticas sobre las aplicaciones sin *pthreads*:
 - Trabajo solapado CPU con E/S
 - Ej.: un programa puede tener secciones en las que realiza operaciones de E/S prolongadas. Mientras un *pthread* espera la completitud de la llamada E/S, los demás *pthreads* hacen uso del CPU.
 - Prioridad/Planificación en tiempo real
 - Las tareas más importantes pueden ser planificadas para sustituir o interrumpir las tareas con menor prioridad.
 - Manejo de eventos asincrónicos:
 - Las tareas que dan servicio a eventos de frecuencia y duración indeterminadas pueden ser intercaladas. Ej.: un servidor web puede transferir datos de peticiones anteriores, en tanto gestiona la llegada de nuevas peticiones.
 - Ejemplo perfecto es un **navegador web**: múltiples tareas intercaladas pueden estar ejecutándose al mismo tiempo, cada tarea puede variar en prioridad.



- Algunas consideraciones para el diseño de programas paralelos:
 - ¿Qué tipo de modelo de programación paralela utilizar?
 - Problemas de particionamiento
 - Balanceo de cargas
 - Comunicaciones
 - Dependencia de datos
 - Sincronización y condiciones de carrera
 - Memoria
 - E/S
 - Complejidad del programa
 - Esfuerzo, costos, tiempo del programador
 - among others....



➤ Diseño de programas paralelos



- Gestión de *pthreads*: Rutinas que trabajan directamente en los hilos: crear, separar, unir, etc. También incluyen funciones para establecer/consultar los atributos de los hilos (unirse, programar, etc.)
- Mutexes: Rutinas que se ocupan de la sincronización, denominadas "mutex", que es una abreviatura de "exclusión mutua". Las funciones mutex permiten crear, destruir, bloquear y desbloquear los mutex. Éstas se complementan con funciones de atributo mutex que establecen o modifican los atributos asociados a los mutex.
- Variables de condición: Rutinas que dirigen las comunicaciones entre los hilos que comparten un mutex. Basadas en condiciones especificadas por el programador. Este grupo incluye funciones para crear, destruir, esperar y señalar basadas en valores de variables especificadas. También se incluyen funciones para establecer/consultar los atributos de las variables de condición.
- Sincronización: Rutinas que gestionan los bloqueos y barreras de lectura/escritura.



> API Pthreads

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers



➤ Compilación de Pthreads en las diferentes plataformas

Compiler / Platform	Compiler Command	Description
INTEL Linux	<code>icc -pthread</code>	C
	<code>icpc -pthread</code>	C++
PGI Linux	<code>pgcc -lpthread</code>	C
	<code>pgCC -lpthread</code>	C++
GNU Linux, Blue Gene	<code>gcc -pthread</code>	GNU C
	<code>g++ -pthread</code>	GNU C++
IBM Blue Gene	<code>bgxlC_r / bgcc_r</code>	C (ANSI / non-ANSI)
	<code>bgxlC_r, bgxlC++_r</code>	C++



➤ Rutinas básicas *pthreads*

- **Creation:** `pthread_create`
- **Termination:**
 - **Return**
 - `Pthread_exit`
- **Wait (parent/child synchronization):** `pthread_join`
- **Pthread header file** `<pthread.h>`
- **Compiling pthread programs:** `gcc -lpthread aaa.c`



Ejercicio



Pontificia
U
JAVE

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 void *funcion(void *arg){
8     printf("Función Hilo: INICIO \n");
9     sleep(2); //espera 2 segundos (dormir)
10    printf("Función Hilo: FIN\n");
11    return NULL;
12 }
13
14 int main(int argc, char **argv){
15     pthread_t hilo; //se identifica el hilo
16     //Se crea el hilo, se envía la función
17     int hiloCreacion = pthread_create(&hilo, NULL, &funcion, NULL);
18     //Se verifica que el hilo se creo exitosamente
19     if (hiloCreacion) {
20         printf("Fallo de creación del hilo :: %d \n", hiloCreacion);
21     } else {
22         printf("Hilo creado con ID ::%d \n", hiloCreacion);
23     }
24
25     printf("Espera de salida del hilo...\n");
26
27     hiloCreacion = pthread_join(hilo, NULL); //Se espera el proceso del hilo termine
28     //Se verifica sale exitosamente
29     if (hiloCreacion) {
30         printf("Fallo de salida del hilo :: %d \n", hiloCreacion);
31     }
32     printf("Final de programa principal\n");
33     return 0;
34 }
```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <pthread.h>
6
7 void * threadFunc(void * arg)
8 {
9 std::cout << "Thread Function :: Start" <<
10 sleep(2);
11 std::cout << "Thread Function :: End" << std::endl;
12 }
13
14 int main()
15 {
16 pthread_t hilo; // Thread id
17 int err = pthread_create(&hilo, NULL, &threadFunc, NULL);
18 if (err != 0){
19 std::cout << "Error creating thread: " << err << std::endl;
20 }
21 sleep(2);
22 std::cout << "Final de programa principal" << std::endl;
23 }

redor Franco, PhD

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 #define N 5
8
9 static void *funcion(void *arg) {
10     size_t job = *(size_t*)arg;
11     printf("Job %zu\n", job);
12     return NULL;
13 }
14
15 int main(int argc, char *argv[]) {
16     size_t jobs[N];           //vector de trabajos
17     pthread_t hilos[N];      //Vector de hilos
18     for (size_t i=0; i<N; i++) {
19         jobs[i] = i;
20         pthread_create(&hilos[i], NULL, funcion, jobs+i);
21     }
22
23     for (size_t i=0; i<N; i++) {
24         pthread_join(hilos[i], NULL); //Se espera por cada hilo finalice
25     }
26
27     return 0;
28 }
```

➤ Ejercicio: productor - consumidor

```
35 void * productor(){          23     for(i=0; i<5; i++)  
36     while(1) {                  24     {  
37         pthread_mutex_lock(&condp_mutex);    25         pthread_create(&prothr[i],NULL,&prodfun,NULL);  
38         pthread_cond_wait(&condVarProd,&condp_mutex); 26         pthread_create(&conthr[i],NULL,&consfun,NULL);  
39         counter++;                27     }  
40         pthread_cond_signal(&condVarCons); 28     }  
41     }                                29     }  
42     pthread_mutex_unlock(&condp_mutex); 30     }  
43  
44     pthread_mutex_lock(&counter_mutex); 31     }  
45     counter++;                      32     }  
46     pthread_cond_signal(&condVarCons); 33     }  
47     printf("Soy productor %d valor contador = %d\n", pthread_self(), counter); 34     }  
48     pthread_mutex_unlock(&condp_mutex); 35     }  
49     pthread_mutex_unlock(&counter_mutex); 36     }  
50     if(counter==5) {                37     }  
51         sleep(1);                  38     }  
52     }                                39     }  
53 }
```

John Corredor Franco, PhD



> Ejercicio: productor - consumidor

```
55 void * consumidor() {  
56     while(1) {  
57         sleep(1);  
58         pthread_mutex_lock(&condc_mutex);  
59         while(counter<=0)  
60         {  
61             pthread_cond_signal(&condVarProd);  
62             pthread_cond_wait(&condVarCons,&condc_mutex);  
63         }  
64         pthread_mutex_unlock(&condc_mutex);  
65         pthread_mutex_lock(&counter_mutex);  
66         if(counter>0) {  
67             printf("Soy consumidor %d valor contador = %d \n", pthread_self(), counter);  
68             counter--;  
69             pthread_cond_signal(&condVarProd);  
70         }  
71         //pthread_mutex_lock(&counter_mutex);  
72         counter++;  
73         read_cond_signal(&condVarCons);  
74         printf("I am producer %ld counter value=%d\n",pthread_self(),counter);  
75     }  
76 }
```

```
28     for(i=0; i<5; i++)  
29     {  
30         pthread_join(prothr[i],NULL);  
31         pthread_join(conthr[i],NULL);  
32     }  
33 }  
34 }  
35 }  
36 }  
37 }  
38 }  
39 }  
40 }  
41 }  
42 }  
43 }  
44 }  
45 }  
46 }  
47 }  
48 }  
49 }  
50 }  
51 }  
52 }  
53 }  
54 }  
55 }  
56 }  
57 }  
58 }  
59 }
```

➤ Ejercicio: productor_consumidor

```
 6 int counter = 0;
 7 int max      = 4;
 8
 9
10 pthread_mutex_t counter_mutex;
11 pthread_mutex_t condp_mutex;
12 pthread_mutex_t condc_mutex;
13 pthread_cond_t  condVarProd;
14 pthread_cond_t  condVarCons;
15
16 void *productor();
17 void *consumidor();
18
19
20 int main(){
21
22     pthread_t proHilo[max], conHilo[max];
23     int i;
24     for(i=0; i<5; i++) {
25         pthread_create(&proHilo[i], NULL, &productor, NULL);
26         pthread_create(&conHilo[i], NULL, &consumidor, NULL);
27     }
28     for(i=0; i<5; i++){
29         pthread_join(proHilo[i], NULL);
30         pthread_join(conHilo[i], NULL);
31     }
32     return 0;
33 }
```

```
14 void *prodfun();
15 void *consfun();
16
17
18 main()
19 {
20     pthread_mutex_init(&counter_mutex, NULL);
21     pthread_mutex_init(&condp_mutex, PTHREAD_MUTEX_INITIALIZER);
22     pthread_mutex_init(&condc_mutex, PTHREAD_MUTEX_INITIALIZER);
23     pthread_cond_init(&condVarProd, PTHREAD_COND_INITIALIZER);
24     pthread_cond_init(&condVarCons, PTHREAD_COND_INITIALIZER);
25
26     for(i=0; i<5; i++)
27     {
28         pthread_join(prothr[i], NULL);
29         pthread_join(conthr[i], NULL);
30     }
31
32     void * prodfun()
33     {
34         while(1)
35         {
36             pthread_mutex_lock(&counter_mutex);
37             if(counter == max)
38                 pthread_cond_wait(&condVarProd, &condp_mutex);
39             else
40                 pthread_cond_signal(&condVarCons);
41         }
42     }
43
44     void * consfun()
45     {
46         while(1)
47         {
48             pthread_mutex_lock(&condc_mutex);
49             if(counter == 0)
50                 pthread_cond_wait(&condVarCons, &condc_mutex);
51             else
52                 counter--;
53             pthread_mutex_unlock(&condc_mutex);
54             printf("I am producer %d counter value\n", counter);
55         }
56     }
57 }
```

