Sistemas Operativos I

“A computer is a state machine. Threads are for people who can't program state machines.”

Alan Cox

Clase: planificación de procesos - sincronización

Rafael Ignacio Zurita <rafa@fi.uncoma.edu.ar>

Advertencia: Estos slides traen ejemplos.

No copiar (ctrl+c) y pegar en un shell o terminal los comandos aquí presentes.

Algunos no funcionarán, porque al copiar y pegar tambien van caracteres “ocultos” (no visibles pero que están en el pdf) que luego interfieren en el shell.

Sucedió en vivo :)

Conviene “escribirlos” manualmente al trabajar.

Sistemas Operativos I - Procesos

Contenido

- Cambio de contexto
- Planificación de Procesos
- Sincronización
  - productor-consumidor — semáforos
  - región crítica — mutex

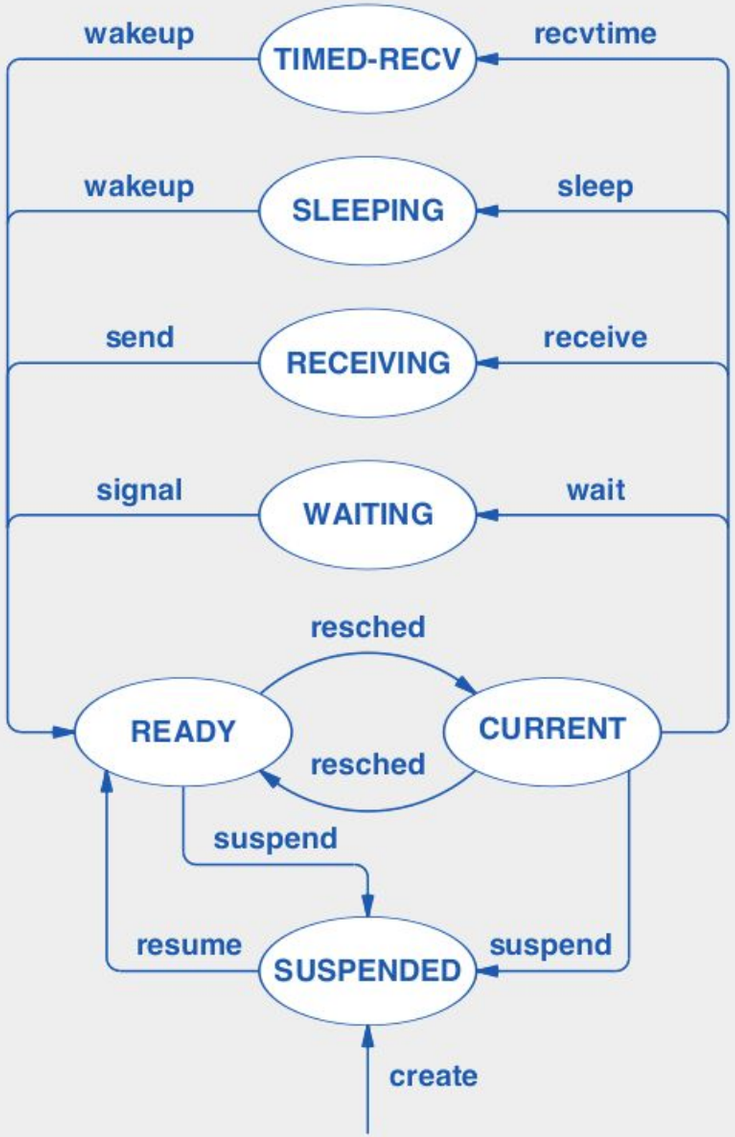
Estados de un proceso

Multiprogramación apropiativa (tiempo compartido)

- Slice o QUANTUM
- Soporte del hardware: timer
- Cambio de contexto

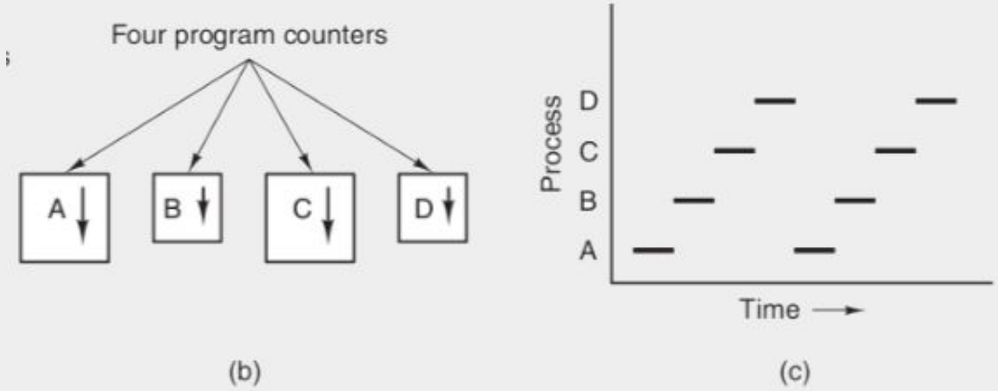
Estados de un proceso en Xinu (EJEMPLO)

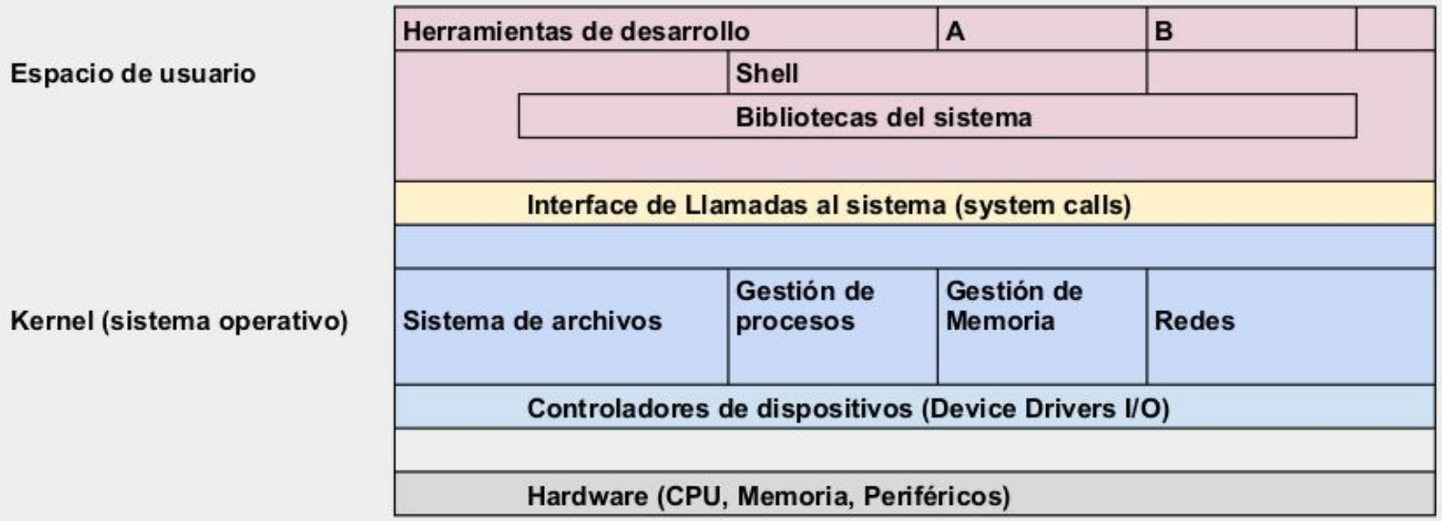
```
#include <xinu.h>

void test(void)
{
    int pid;

    pid = create(procA, 8192, 20, "process 1", 0) );
    resume(pid);
    sleep(10);
    kill(pid);
}

/* program procA  --  repeatedly emit 'A' on the console *
void procA(void)
{
        while (1) {
                putc(CONSOLE, 'A');
                sleep(1);
        }
}
```

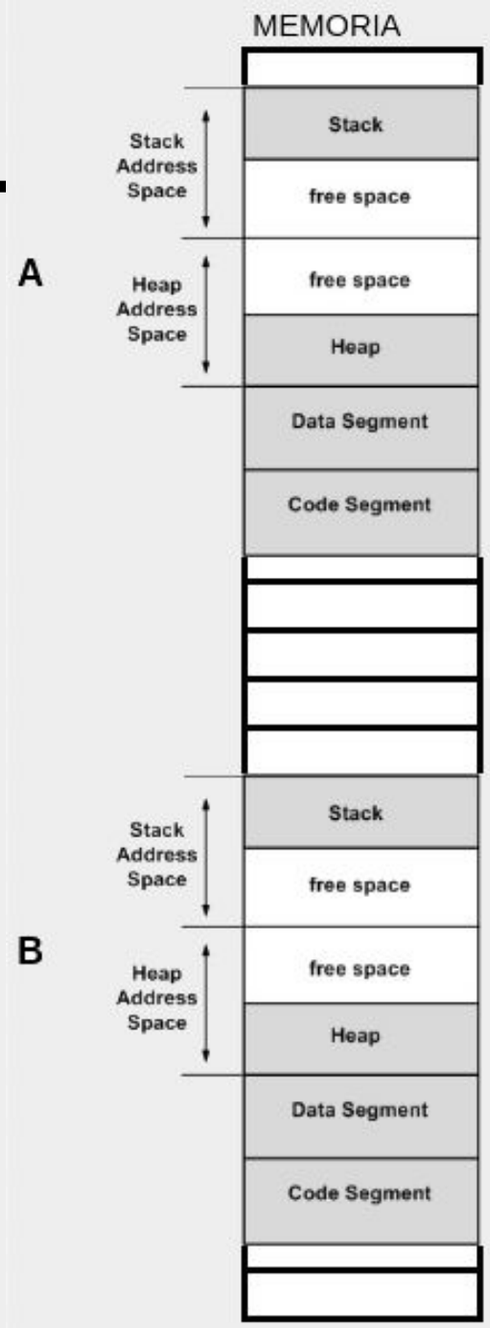


Sistemas Operativos I - cambio de contexto

Ejecución concurrente - cambio de contexto





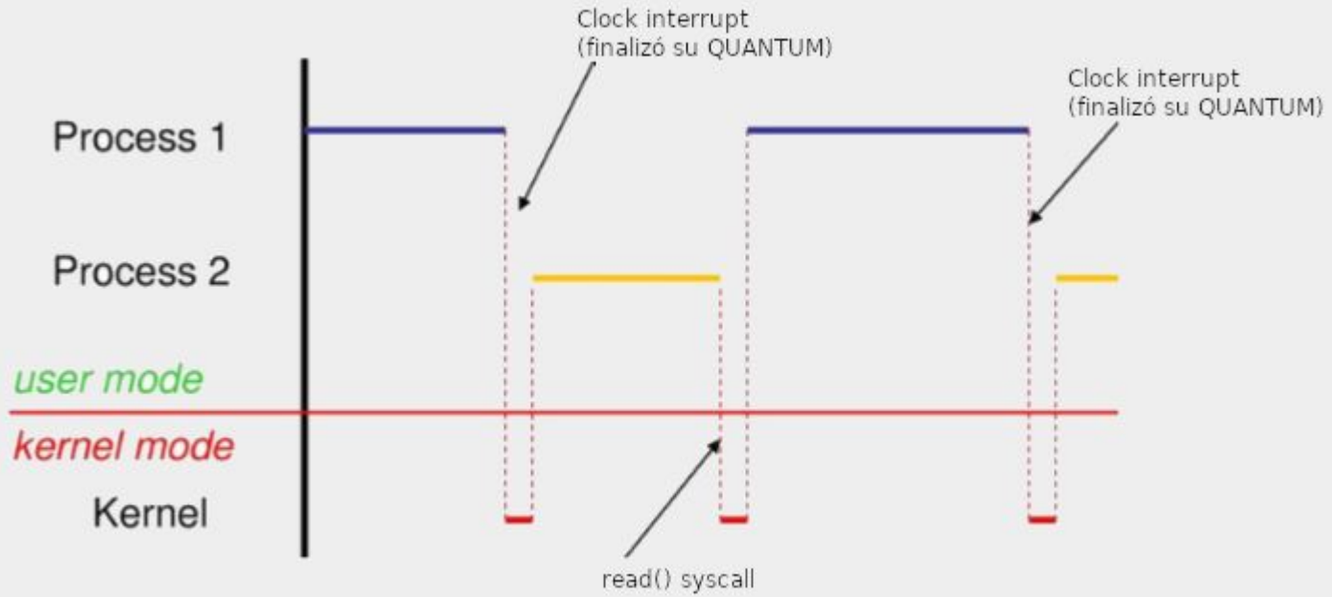


Sistemas Operativos I - cambio de contexto

Implementación de procesos concurrentes

Un proceso "libera" la CPU:

1. solicita un servicio al SO
2. finalizó su **QUANTUM**



1. multiprogramación
1. y 2. multiprogramación y tiempo compartido
(requiere reloj por hardware [clock/timer]) - interrupción

Los SO de tiempo compartido son "apropiativos" (preemptive)

Sistemas Operativos I - cambio de contexto

Implementación de procesos concurrentes

Cambio de contexto

Arreglo de estructuras PCBs.

1. Resguardar el estado del procesador del proceso A
2. Cargar el estado anterior del procesador del proceso B
3. "Poner" la CPU en modo usuario

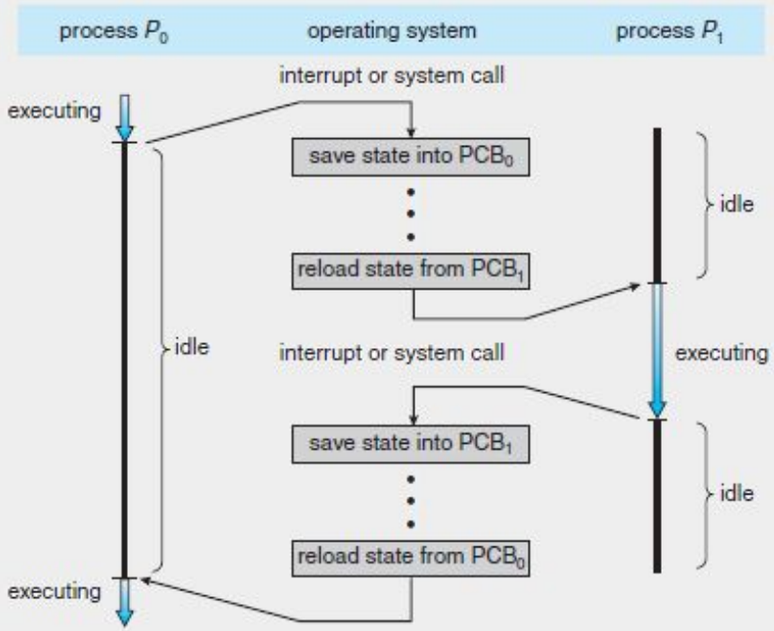
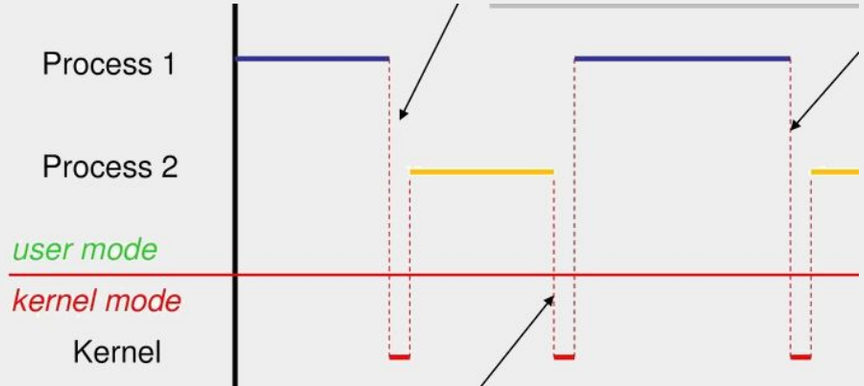


Figure 3.4 Diagram showing CPU switch from process to process.

Estado del procesador:

Registros (pc, stack pointer, otros registros).
El código se implementa utilizando lenguaje ensamblador.
Tiene una implementación diferente para cada arquitectura (ISA).

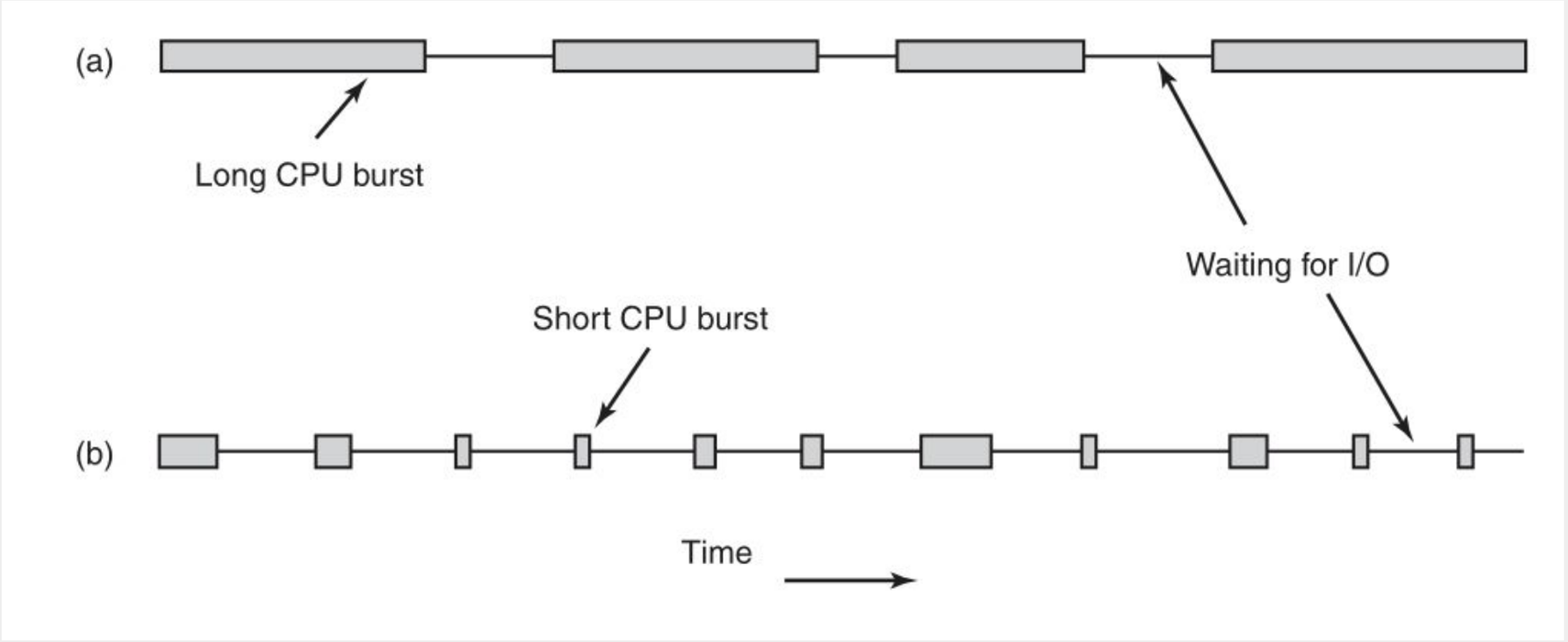


Sistemas Operativos I - Procesos : planificación

Planificación de procesos

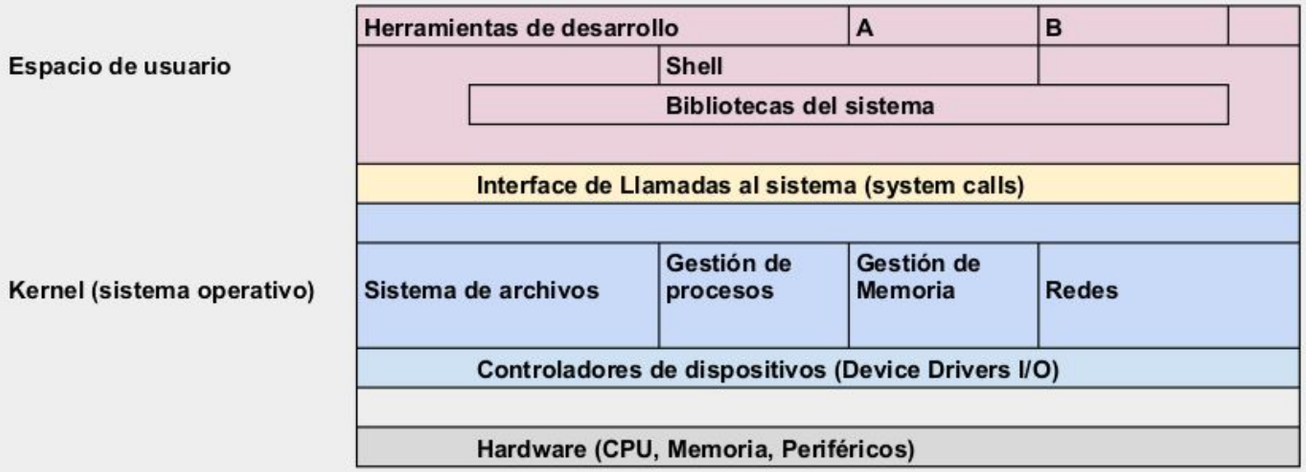
- La multiprogramación permite maximizar la utilización de la CPU.
- Los procesos poseen secuencias alternadas de ráfagas de CPU y de E/S.

- **(a) Proceso cargado en CPU**
- **(b) Proceso cargado en E/S**



Sistemas Operativos I - Procesos : planificación

Planificación de procesos



- **PLANIFICADOR** (SCHEDULER): Es el componente del kernel que selecciona un proceso para asignarle la CPU

- **algoritmo de planificación:** Es el algoritmo que toma la decisión de seleccionar el proceso que debe utilizar la CPU

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Metas

- Para todos
  - Justo
  - Que aplique la política del administrador o programador
  - Balanceado (todos los recursos en uso)

- Throughput: Maximizar la cantidad de trabajos por hora — Sistemas Batch
- Tiempo de turnaround: Minimizar tiempos desde arribo a finalizado
- Mantener la CPU en uso mayormente

- Minimizar los tiempos de respuesta ante un requisito o evento — Sistemas interactivos
- Proporcional: cumplir expectativas de los usuarios

- Cumplir tiempos de respuesta límites — Sistemas de Tiempo Real
- Predecible

Planificación de procesos

- **Métricas básicas para evaluar algoritmos de planificación**

  Tiempo de turnaround
  Tiempo de turnaround promedio:

  ```
  sumatoria de turnarounds / n
  ```

  Tiempo de espera
  Tiempo de espera promedio:

  ```
  sumatoria de tiempos de espera / n
  ```

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación de CPU

Los algoritmos básicos son los siguientes:

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRTF (Shortest Remaining Time First)
- RR (Round Robin)
- Prioridades
- Combinado (round-robin y prioridades)
- Colas multinivel

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - FCFS (First Come First Served)

Atiende a los procesos según el orden de arribo a la cola de listos.

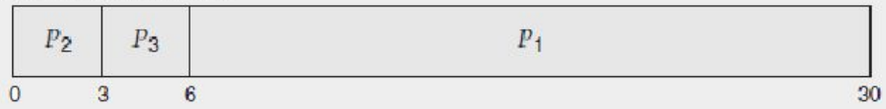
Es un algoritmo no preemptive: cuando un proceso obtiene la CPU no la deja hasta finalizar su ráfaga.

Proceso	Ráfaga de CPU
P1	24
P2	3
P3	3

- Suponer que los procesos arribaron a la cola de listos en el orden P1, P2 y P3.
- El diagrama de Gantt correspondiente a la planificación es:

- Tiempo de turnaround: P1=24; P2=27; P3=30.
- Tiempo de turnaround promedio: (24+27+30)/3= 27
- Tiempo de espera: P1=0; P2 =24; P3=27
- Tiempo de espera promedio: (0+24+27)/3 =17

- Suponer que los procesos arribaron a la cola de listos en el orden P2, P3, P1.
- El diagrama de Gantt correspondiente a la planificación es:



- Tiempo de turnaround: P1= 30; P2=3; P3=6
- Tiempo de turnaround promedio: (30+3+6)/3=13
- Tiempo de espera: P1= 6; P2=0; P3=3
- Tiempo de espera promedio: (6+0+3)/3=3

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - SHORTEST JOB FIRST (SJF)

Selecciona de la cola de listos aquel proceso cuyo próximo intervalo de CPU sea el más corto.
SJF es un algoritmo no preemptive:

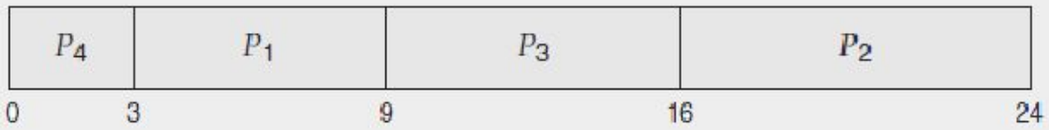
Una vez que un proceso obtiene la CPU,
no puede ser desalojado de ella hasta que finalice su intervalo.

Es óptimo, se obtiene el menor tiempo de espera promedio
para un conjunto de procesos.

Problema: conocer por adelantado la duración de la siguiente ráfaga de CPU.

Proceso	Ráfaga de CPU
P1	6
P2	8
P3	7
P4	3

- Suponer que los procesos arribaron a la cola de listos en el orden P1, P2, P3 y P4.
- El diagrama de Gantt correspondiente a la planificación es:



- Tiempo de turnaround: P1= 9; P2=24; P3=16; P4= 3
- Tiempo de turnaround promedio: (9+24+16+3)/4=13
- Tiempo de espera: P1= 3; P2=16; P3=9; P4=0
- Tiempo de espera promedio: (3+16+9+0)/4=7
- Tiempo de espera promedio para FCFS: (0+6+14+21)/4 = 10,25

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - SHORTEST REMAINING TIME FIRST (SRTF)

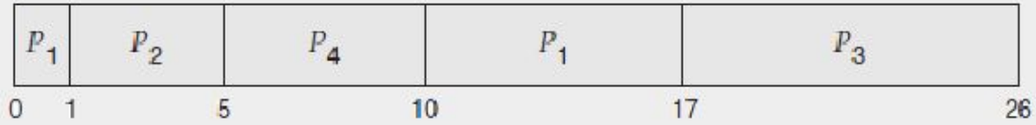
Es una implementación preemptive de SJF.

Cuando un proceso arriba a la cola de listos,
el planificador analiza los intervalos de CPU.

De todos los procesos; si el nuevo proceso tiene el menor intervalo,
desaloja al que estaba ejecutándose y le asigna la CPU.

Proceso	Arribo	Ráfaga de CPU
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- El diagrama de Gantt correspondiente a la planificación es:



- Tiempo de turnaround: P1=17; P2=4; P3=24; P4=7
- Tiempo de turnaround promedio: (17+4+24+7)/4=13
- Tiempo de espera: P1=9; P2=0; P3=15; P4=2
- Tiempo de espera promedio: (9+0+15+2)/4=6,5
- Tiempo de espera promedio en SJF: (0+7+15+9)/4=7,75

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - SHORTEST JOB FIRST (SJF) - SHORTEST REMAINING TIME FIRST (SRTF)

SJF y SRTF son algoritmos teóricos pues no se puede saber exactamente cuál es la longitud de la próxima ráfaga de CPU.

Se puede implementar realizando una estimación en base a las longitudes de las ráfaga de CPU previas.

Se elige el proceso con la siguiente ráfaga de CPU que se ha estimado como más corta.

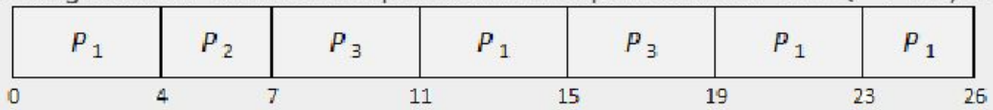
Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - ROUND ROBIN (RR)

Utiliza una **unidad de tiempo de CPU (quantum q)** *"usualmente de entre 10 y 100 milisegundos"*.
Cada proceso se ejecuta un quantum. Se lo desaloja y se lo coloca al final de la cola de listos. RR es preemptive.
Si hay n procesos en la cola de listos, y el quantum es q, entonces cada proceso ejecutará 1/n del tiempo total de la CPU en intervalos de como máximo q unidades.
Ningún proceso esperará por la CPU más que (n-1)q unidades de tiempo.

Proceso	Ráfaga de CPU
P1	15
P2	3
P3	8

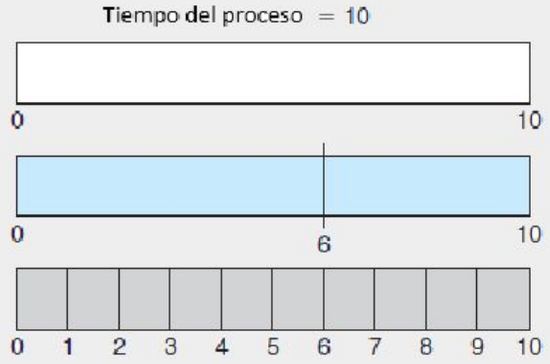
- Suponer que los procesos arribaron a la cola de listos en el orden P1, P2 y P3.
- El diagrama de Gantt correspondiente a la planificación con Q=4 ms, es:



- Tiempo de turnaround: P1= 26; P2=7; P3=19
- Tiempo de turnaround promedio: (26+7+19)/3=17,33
- Tiempo de espera: P1= 11; P2=4; P3=11
- Tiempo de espera promedio: (11+4+11)/3=8,66
- En general, RR produce un mayor tiempo de turnaround promedio que SJF ((26+3+11)/3=13,33), aunque el tiempo de respuesta suele ser mucho mejor.

SJF

- La perfomance depende del tamaño del quantum:
  - q grande ✉ RR tiende a FCFS.
  - q pequeño ✉ si q está en el orden del tiempo de cambio de contexto, el overhead del sistema será muy alto.



quantum	Cambios de contexto
12	0
6	1
1	9

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - POR PRIORIDADES

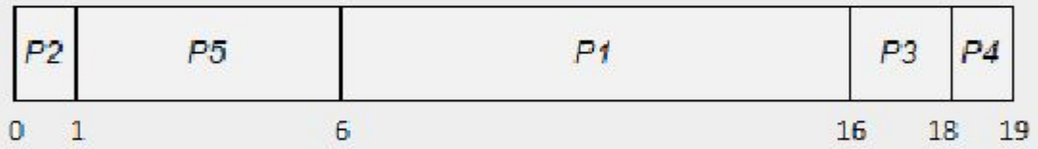
- Cada proceso tiene asociado un número (entero) de **prioridad**.
- La CPU se asigna al proceso con la mayor prioridad (a menor número mayor prioridad).
  - Preemptive
  - Nonpreemptive
- El problema que presenta es **starvation**
  - Solución posible : aging
- Por prioridades es combinado generalmente con Round Robin (ejemplo Xinu)

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - POR PRIORIDADES

Proceso	Prioridad	Ráfaga de CPU
P1	3	10
P2	1	1
P3	4	2
P4	5	1
P5	2	5

- El diagrama de Gantt correspondiente a la planificación es:



- Tiempo de turnaround: P1=16; P2=1; P3=18; P4=19; P5=6
- Tiempo de turnaround promedio: (16+1+18+19+6)/5=12
- Tiempo de espera: P1=6; P2=0; P3=16; P4=18; P5=1
- Tiempo de espera promedio: (6+0+16+18+1)/5=8,2

Variante

- Por prioridades es combinado generalmente con Round Robin (ejemplo Xinu)
- XINU
- Linux y Windows "se asemejan"

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación - COLAS MULTINEVEL

- También debe realizarse la planificación entre las colas. Dos posibles estrategias:
  - **Por nivel de cola**: cada cola tiene prioridad absoluta sobre las colas de prioridad más baja. Posibilidad de starvation.
  - **Por tiempo**: cada cola tiene una cierta cantidad de tiempo disponible para planificar sus procesos.

Un proceso puede moverse entre las distintas colas.
Los procesos son asignados a las distintas colas según las características de sus ráfagas de CPU.

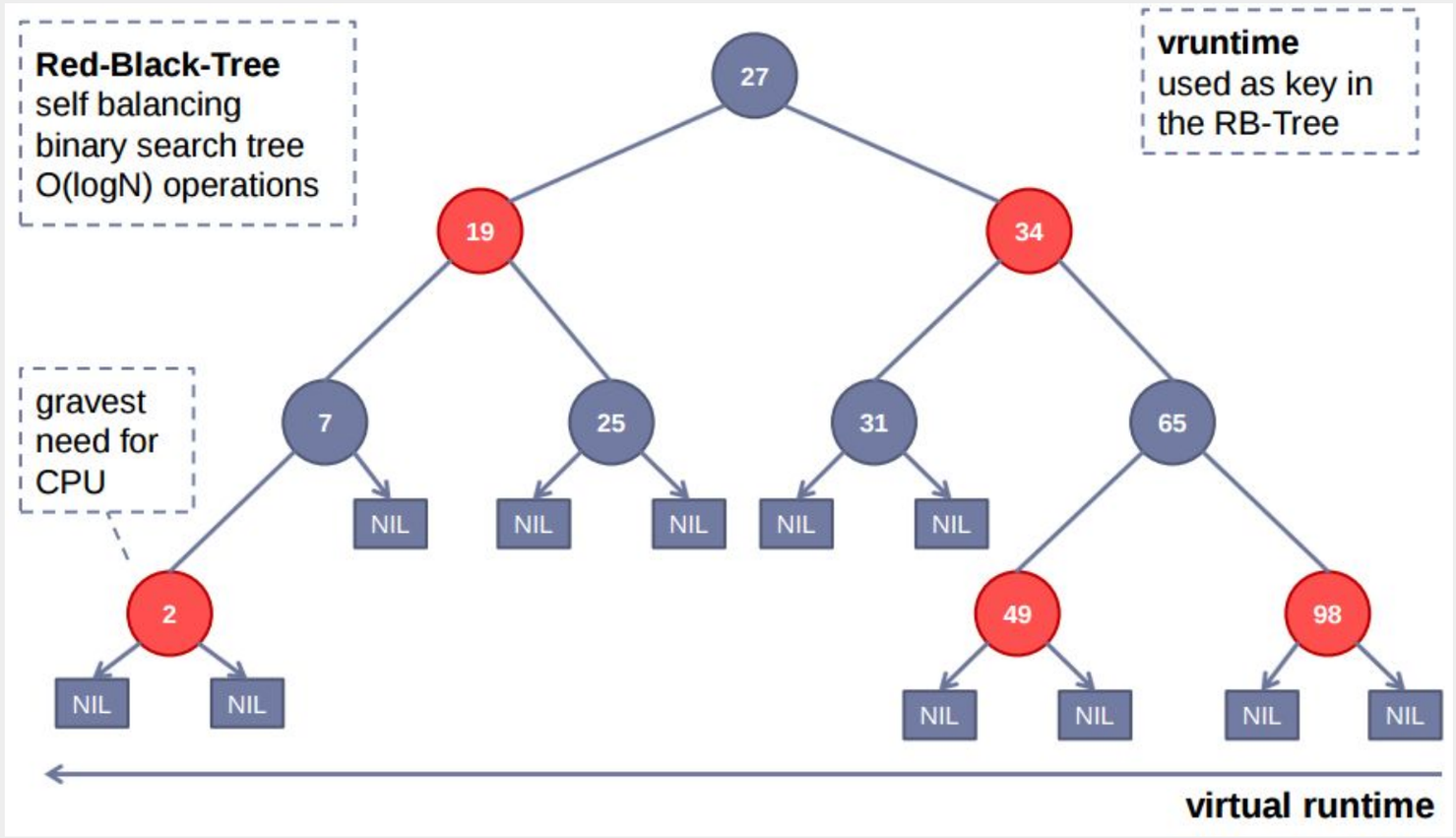
Los procesos
de menor prioridad van "promocionando"
de mayor prioridad van "degradando" de cola.
Es una forma de implementar aging y sirve para evitar starvation.

Este esquema de planificación está definido por los siguientes parámetros:

- Cantidad de colas;
- Algoritmo de planificación de cada cola;
- Método para determinar cuando se promociona un proceso;
- Método para determinar cuando se degrada un proceso;
- Método para determinar a que cola ingresará un proceso.

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación en Linux y Anroid CFS (completely fair scheduling)



vruntime += t ms * (peso basado en nice y prioridad)

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación en Windows (dispatcher - por prioridades)

Fila (prioridad relativa) - Columna (prioridad base)

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 5.28 Windows thread priorities.

- REALTIME_PRIORITY_CLASS—24
- HIGH_PRIORITY_CLASS—13
- ABOVE_NORMAL_PRIORITY_CLASS—10
- NORMAL_PRIORITY_CLASS—8
- BELOW_NORMAL_PRIORITY_CLASS—6
- IDLE_PRIORITY_CLASS—4

Prioridades tienen dos clases:
1. real-time (16-31)
2. variable (1-15)

incrementa prioridad cuando pasa de esperando a listo (mejora UI)
decrementa prioridad cuando utiliza todo el quantum (evita starvation)

```
/* TRES PROCESOS QUE CANTAN en XINU */

#include <xinu.h>

#define CANT_FRASES 3   /* cantidad de frases a cantar */

char * lucia[] = {"- Quien es?",
                  "- Que vienes a buscar?",
                  "- Ya es tarde.",
                  "- Porque ahora soy yo la que quiere estar sin
ti."
                 };

char * joaquin[] = {"- Soy yo.",
                    "- A ti.",
                    "- Por que?",
                    "- ."
                   };


void    main(void)
{
        /* creamos dos procesos en Xinu (threads) */
        resume( create(canta_lucia, 1024, 20, "lucia", 0) );
        resume( create(canta_joaquin, 1024, 20, "joaquin", 0) );
}
```

```
/* Canta Joaquin Galan */
void    canta_joaquin(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", joaquin[i]);
                sleep(3);
        }
}



/* Canta Lucia Galan */
void    canta_lucia(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", lucia[i]);
                sleep(3);
        }
}
```

Sistemas Operativos I

```
#include <xinu.h>

#define CANT_FRASES 3   /* cantidad de frases a cantar */

char * lucia[] = {"- Quien es?",
                  "- Que vienes a buscar?",
                  "- Ya es tarde.",
                  "- Porque ahora soy yo la que quiere estar sin ti."
                 };

char * joaquin[] = {"- Soy yo.",
                    "- A ti.",
                    "- Por que?",
                    "- ."
                   };


void    main(void)
{
        /* creamos dos procesos en Xinu (threads) */
        resume( create(canta_lucia, 1024, 20, "lucia", 0) );
        resume( create(canta_joaquin, 1024, 20, "joaquin", 0) );
}
```

¿Está todo en orden en este código fuente?.

¿Probamos?

```
/* Canta Joaquin Galan */
void    canta_joaquin(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", joaquin[i]);
                sleep(3);
        }
}



/* Canta Lucia Galan */
void    canta_lucia(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", lucia[i]);
                sleep(3);
        }
}
```

Problemas comunes con procesos cooperativos concurrentes

- **Falta de sincronización**
  - Productor consumidor
- **Falta de protección de acceso a un recurso**
  - Mutual exclusion
  - Región crítica
    - Recurso compartido
    - Condición de carrera

Sistemas Operativos I

```
#include <xinu.h>

#define CANT_FRASES 3   /* cantidad de frases a cantar */

char * lucia[] = {"- Quien es?",
                  "- Que vienes a buscar?",
                  "- Ya es tarde.",
                  "- Porque ahora soy yo la que quiere estar sin ti."
                 };

char * joaquin[] = {"- Soy yo.",
                    "- A ti.",
                    "- Por que?",
                    "- ."
                   };


void    main(void)
{
        /* creamos dos procesos en Xinu (threads) */
        resume( create(canta_lucia, 1024, 20, "lucia", 0) );
        resume( create(canta_joaquin, 1024, 20, "joaquin", 0) );
}
```

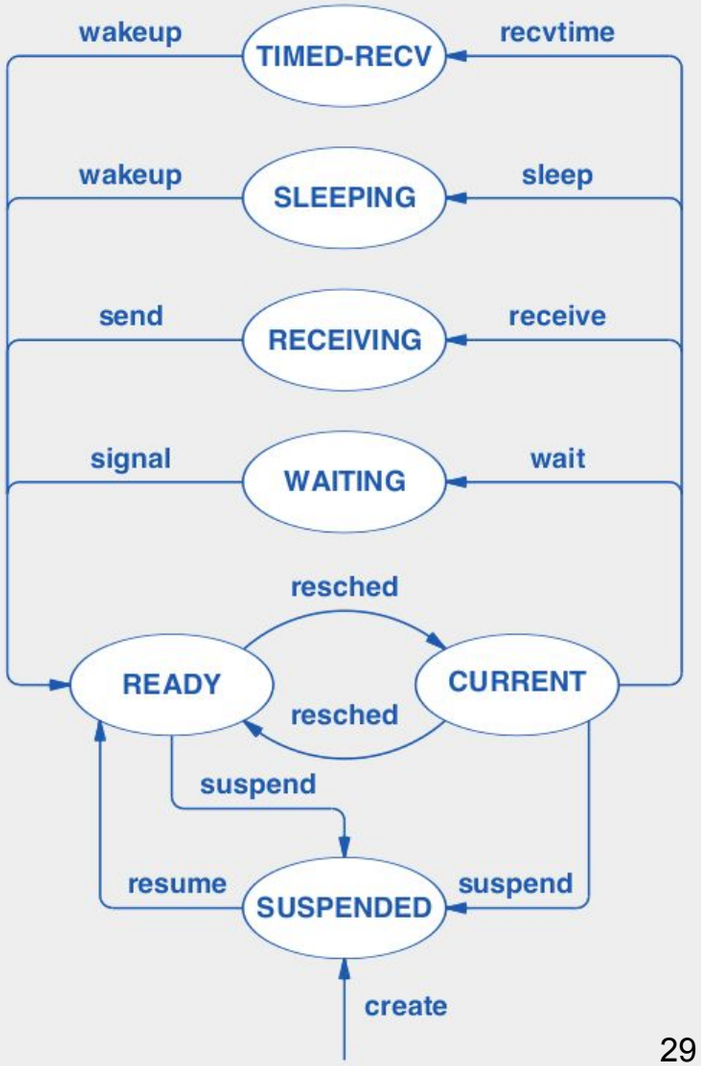
¿Cómo podemos solucionar el problema?

```
/* Canta Joaquin Galan */
void    canta_joaquin(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", joaquin[i]);
                sleep(3);
        }
}



/* Canta Lucia Galan */
void    canta_lucia(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                printf("%s\n", lucia[i]);
                sleep(3);
        }
}
```

Sistemas Operativos I - Sincronización entre procesos

Semáforo: recurso lógico para sincronización implementado por el sistema operativo

- Las aplicaciones pueden acceder al recurso a través de **system calls.**
- Un Semáforo S es una variable entera.
  - Semáforo Contador: valor entero.
  - Semáforo Binario: el valor entero entre 0 y 1
- Solo puede accederse a través de 2 operaciones atómicas:
  - **wait() decrementa el contador. Bloquea si menor que 0**
  - **signal() incrementa el contador. No desbloquea si mayor que 0**



Sistemas Operativos I - Sincronización entre procesos

```
#include <xinu.h>
sid32   luc_sem, joaq_sem;


#define CANT_FRASES 3           /* cantidad de frases a cantar */

char * lucia[] = {      "- Quien es?",
                        "- Que vienes a buscar?",
                        "- Ya es tarde.",
                        "- Porque ahora soy yo la que quiere estar sin ti."
                 };

char * joaquin[] = {    "- Soy yo.",
                        "- A ti.",
                        "- Por que?",
                        "- ."
                   };


/* Canta Lucia Galan */
void    canta_lucia(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                /* lucia espera que le indiquen cuando cantar */
                wait(luc_sem);

                printf("%s\n", lucia[i]);
                sleep(3);

                /* lucia le indica a joaquin que le toca cantar */
                signal(joaq_sem);
        }
}
```

```
/* Canta Joaquin Galan */
void    canta_joaquin(void)
{
        int   i;

        /* cantamos .. */
        for( i=0 ; i <= CANT_FRASES ; i++ ) {

                /* joaquin espera que le indiquen cuando cantar */
                wait(joaq_sem);

                printf("%s\n", joaquin[i]);
                sleep(3);

                /* joaquin le indica a lucia que le toca cantar */
                signal(luc_sem);
        }
}

void    cancion(void)
{

        /* dos semaforos para sincronización */
        luc_sem = semcreate(1);
        joaq_sem = semcreate(0);

        /* creamos dos procesos en Xinu (threads) */
        resume( create(canta_lucia, 1024, 20, "lucia", 0) );
        resume( create(canta_joaquin, 1024, 20, "joaquin", 0) );

}
```

Semáforo: recurso lógico para sincronización

Implementación

- **De software:**
  - Algoritmos de Sección Crítica -
    CUIDADO, puede fallar diría el mago Black
- **De Hardware**
  - Inhibición de interrupciones
  - Instrucciones del ISA atómicas (sin interrupción):
    - Test-And-Set
    - Compare-and-swap()

In 1981, G. L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Fig. 2-24. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show prototypes here or later.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Recursos compartidos

```
/* Programa de usuario : contador */

void    imprimir_factura(void)
{


     /* Imprimir factura .. */

     cp("factura.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

```
/* Programa de Usuario : productor de cine */

void    imprimir_propaganda(void)
{


     /* Imprimir afiche .. */

     cp("afiche.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

Recursos compartidos

```
/* Programa de usuario : contador */

void    imprimir_factura(void)
{


     /* Imprimir factura .. */

     cp("factura.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

```
/* Programa de Usuario : productor de cine */

void    imprimir_propaganda(void)
{


     /* Imprimir afiche .. */

     cp("afiche.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

¿ Nota algo raro ?

Recursos compartidos

```
/* Programa de usuario : contador */

void    imprimir_factura(void)
{


     /* Imprimir factura .. */

     cp("factura.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

```
/* Programa de Usuario : productor de cine */

void    imprimir_propaganda(void)
{


     /* Imprimir afiche .. */

     cp("afiche.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

¿ Nota algo raro ? CONDICIÓN DE CARRERA REGIÓN CRÍTICA

Recursos compartidos

```
/* Programa de usuario : contador */

void    imprimir_factura(void)
{


     /* Imprimir factura .. */

     cp("factura.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

```
/* Programa de Usuario : productor de cine */

void    imprimir_propaganda(void)
{


     /* Imprimir afiche .. */

     cp("afiche.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     close(fd);
}
```

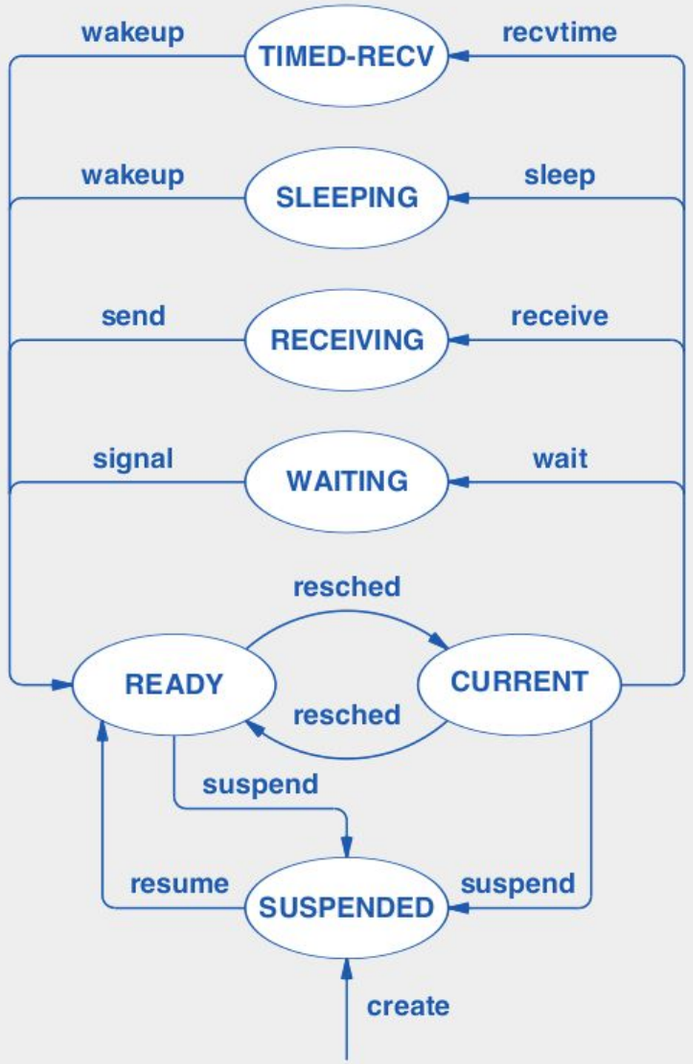
¿ Cómo lo solucionamos? CONDICIÓN DE CARRERA REGIÓN CRÍTICA

Mutex

recurso lógico "posiblemente" implementado
por el sistema operativo para sincronización de procesos.

- Tiene una variable booleana cuyo valor indica si el lock esta disponible o no.
- Acceso protegido a la región crítica:
  - Primero se debe ejecutar acquire() sobre un lock.
    - Es exitosa si el lock está disponible.
    - Si el lock no está disponible, el proceso se bloquea (hasta que se libera el lock).
  - Luego release() el lock.

- Puede ser implementado mediante un semáforo binario. CON PRECAUCIÓN.
- En algunas implementaciones los syscalls pueden llamarse simplemente mutex_lock() y mutex_unlock()



Recursos compartidos

```
/* Programa de usuario : contador */

void    imprimir_factura(void)
{


     /* Imprimir factura .. */

     cp("factura.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);

     mutex_lock();
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     mutex_unlock();

     close(fd);
}
```

```
/* Programa de Usuario : productor de cine */

void    imprimir_propaganda(void)
{


     /* Imprimir afiche .. */

     cp("afiche.pdf", "/var/spool/");

     /* Actualizar cantidad de trabajos a imprimir */
     fd = open("/var/spool/cant_trabajos", RW);

     mutex_lock();
     n = read(fd, 1);
     n = n + 1;
     write(fd, &n, 1);
     mutex_unlock();

     close(fd);
}
```

CONDICIÓN DE CARRERA PROTEGIDA

REGIÓN CRÍTICA