Sistemas Operativos I

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

Alan Cox

Clases 2, 3, y 4
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 y Threads

Contenido general de estos slides

- Procesos
- Comunicación interprocesos
- Threads
- Planificación de procesos
- Sincronización de procesos
- Deadlocks

Sistemas Operativos I - Procesos y Threads

Concepto de Procesos

- Creación de procesos
- Jerarquía de procesos
- Finalización de procesos
- Ejemplos de comandos del sistema y llamadas al sistema

- Implementación de procesos
- Cambio de contexto
- Uso de Xinu

Sistemas Operativos I - Procesos y Threads

Procesos

- Desde el punto de vista del usuario y llamadas al sistema
- Desde el punto de vista del OS:

  multiprogramación y tiempo compartido

  mapa de memoria de un proceso (segmentos)

  memoria general del sistema (kernel y procesos)

  estados de un proceso

  PCB - tabla de procesos

  quantum - reloj/timer - interrupciones

  cambio de contexto

Concepto de Proceso

El kernel tiene la capacidad de poner en ejecución a los programas que se encuentran almacenados en el sistema.

Cuando un programa está en ejecución, lo llamamos un proceso.

El sistema operativo controla la creación, ejecución y finalización de los procesos.

Tarea del SO al crear un proceso

- El sistema operativo obtiene una porción de memoria para el proceso (segmentos de memoria de un proceso)
- Crear una estructura de datos administrativos para el proceso (PCB)
- Asignar un process id (PID)
- Colocar al proceso en estado de listo o suspendido

Creación de procesos (cuando)

- En la secuencia de inicio del sistema
- Cuando una aplicación en ejecución ejecuta un system call para crear un proceso
- Cuando un usuario solicita ejecutar un programa (ej: en el shell)

```c
/* Creación de proceso en LINUX */


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


void  main(void)
{
     int  pid;
     int x = 0;

     pid = fork();

     if (pid == 0)
        printf("Proceso hijo %d\n", x++);
     else
        printf("Proceso padre. Mi hijo es el pid=%d \n", pid);
}

/* otras funciones de la biblioteca de C
 * (que realizan llamadas al sistema)
 *
 * wait()
 * exit()
 * execv()
 * getpid()
 */
```

```c
/* Creación de proceso en XINU */

#include <xinu.h>

void    sndA(void);

void    main(void)
{
      int pid;

     pid = create(sndA, 2048, 20, "process 1", 0) );
     resume(pid);

}


/* proceso sndA */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}
```

Creación de procesos (code)

Sistemas Operativos I - Procesos y Threads

En sistemas de tipo UNIX

- Sistema **jerárquico de procesos** (árbol).
- El proceso padre puede esperar al hijo.
- El proceso hijo puede reemplazar sus segmentos con el de un nuevo programa (Ej. en Stallings).

  En Linux, comandos útiles: ps, pstree, top, kill, killall

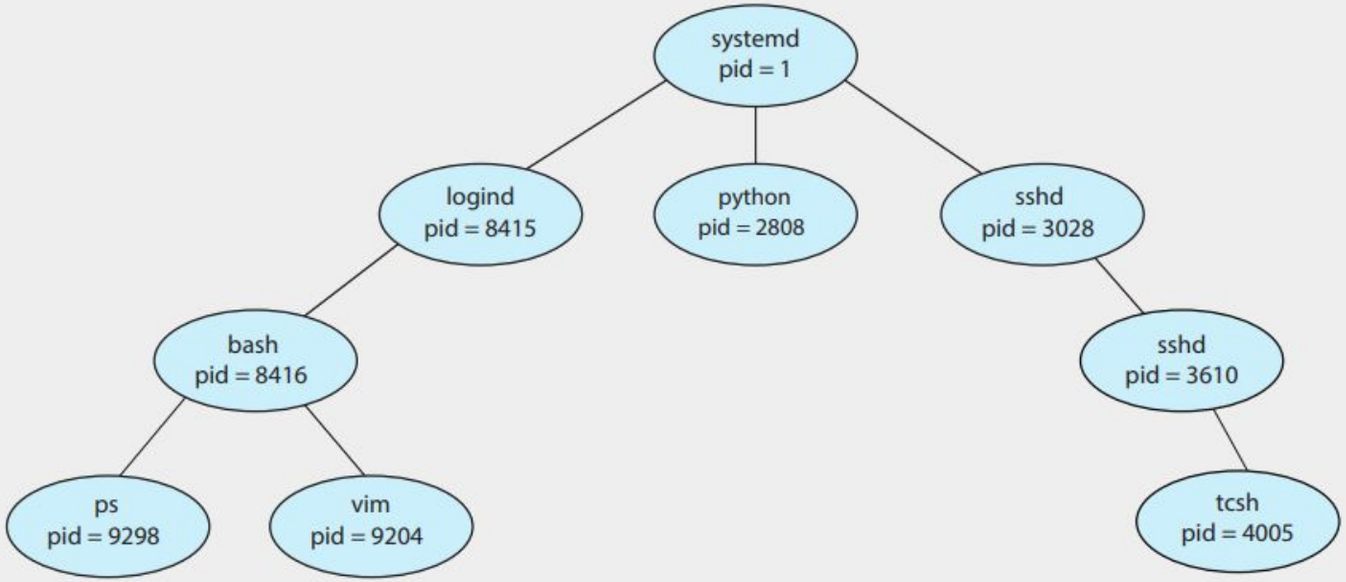


Figure 3.7 A tree of processes on a typical Linux system.

Finalización de procesos (cuando)

- Finalización normal (voluntario).
- Salida con Error (voluntario).
- Error detectado por el OS (involuntario).
- Finalizado por otro proceso (ej: kill, involuntario)

Usualmente existen los system calls kill() y exit()

```c
/* Creación y finalización de proceso en Linux */

#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <signal.h>


void  main(void)
{
     int  pid;
     int x = 0;

     pid = fork();

     if (pid == 0)
        for(;;)
                printf("Proceso hijo %d\n", x++);

     else {
        sleep(5);
        kill(pid, SIGKILL);
        printf("Maté a mi propio hijo (suena horrible)\n");

     }
}
```

```c
/* Creación y finalización de proceso en XINU */

#include <xinu.h>

void    sndA(void);

void    main(void)
{
     int pid;

     pid = create(sndA, 2048, 20, "process 1", 0) );
     resume(pid);
     sleep(5);
     kill(pid);
     printf("Maté a mi propio hijo (suena horrible)\n");
}


/* proceso sndA */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}
```

Finalización de procesos (code)

Sistemas Operativos I - Procesos y Threads

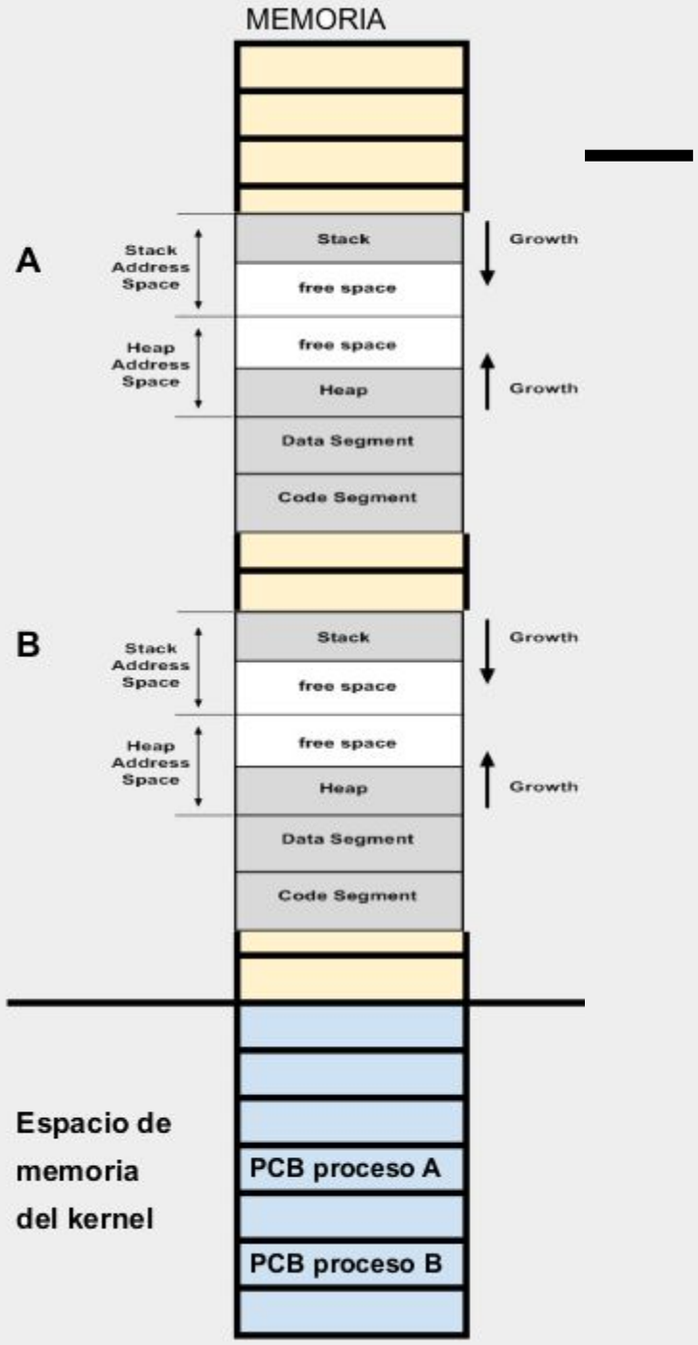
Implementación de procesos

¿Qué mantener? : espacio de direcciones, registros del estado del proceso, lista de archivos abiertos, semáforos que espera, etc

El kernel mantiene un Arreglo/Lista de Estructuras, donde
Cada elemento es una Tabla o **Bloque de Control proceso (PCB)**

Cada **PCB** Contiene:

- PID, el PID del padre
- Espacio para resguardar el contenido de los Registros de la CPU
  - (pc, stack pointer, otros registros)
- Estado del proceso
- El espacio de direcciones de memoria del proceso
- Archivos abiertos
- Recursos en uso (semáforos, dispositivos E/S, etc)



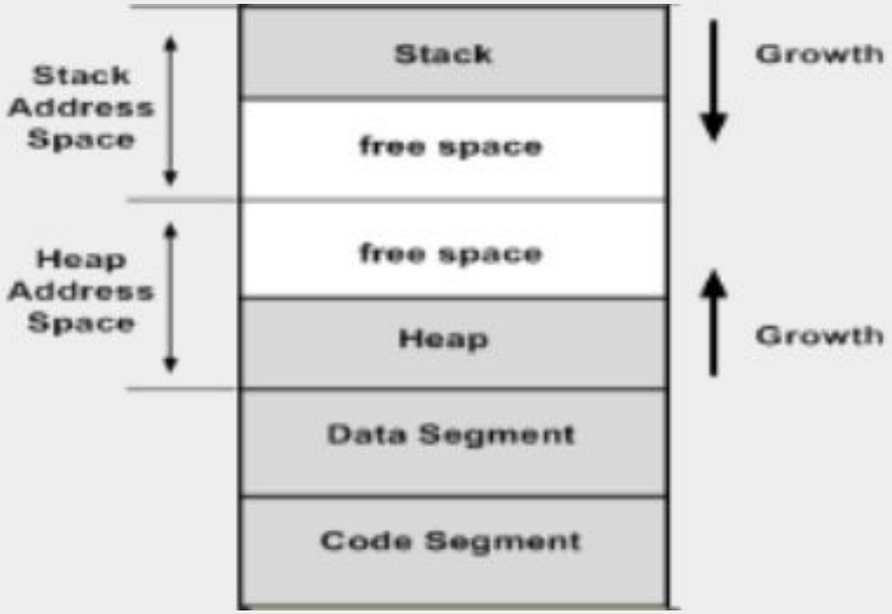
Sistemas Operativos I - Procesos y Threads

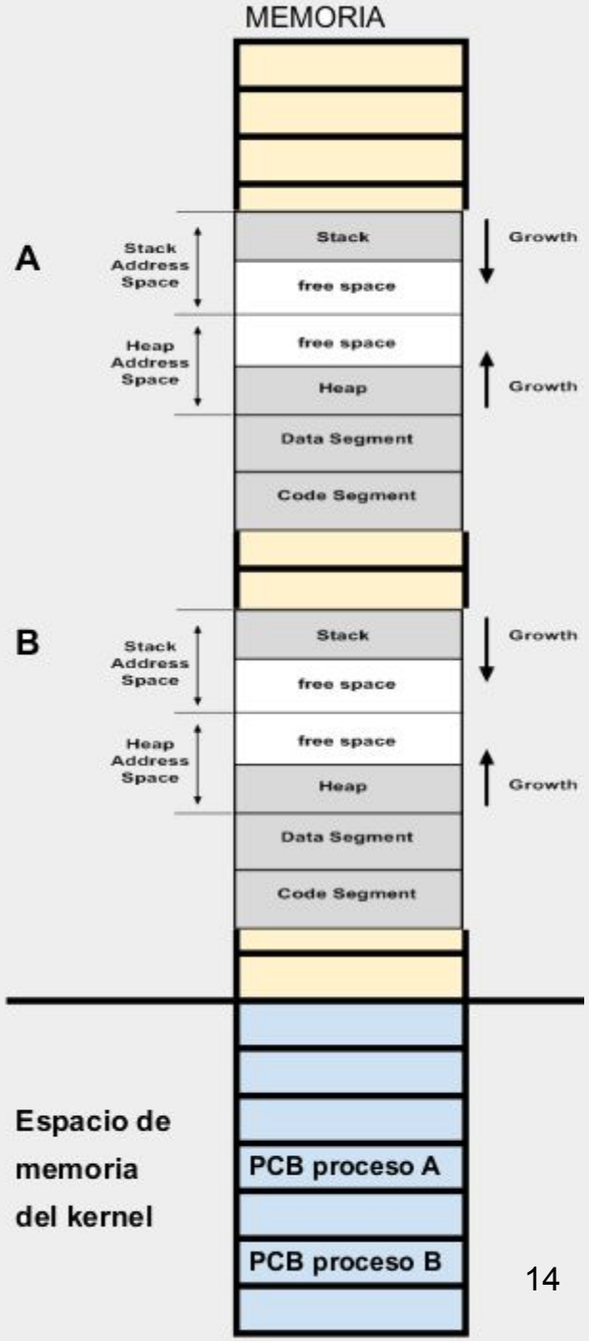
Implementación de procesos

mapa de memoria de un proceso (segmentos)

memoria general del sistema (kernel y procesos)

```
char edad = 45;
int DNI = 34563112;
char nota[] = "Isidoro Caniones";
char encrip[] = "Los Elefantes de Asia";

main () {

        int i = 0;
        int c = 0;

        for (i=0; i<60; i++)
          c = add_elem(i);

}

int add_elem(int n)
{
    int val;

    val = nota[n] + encrip[n];
    return val;
}
```

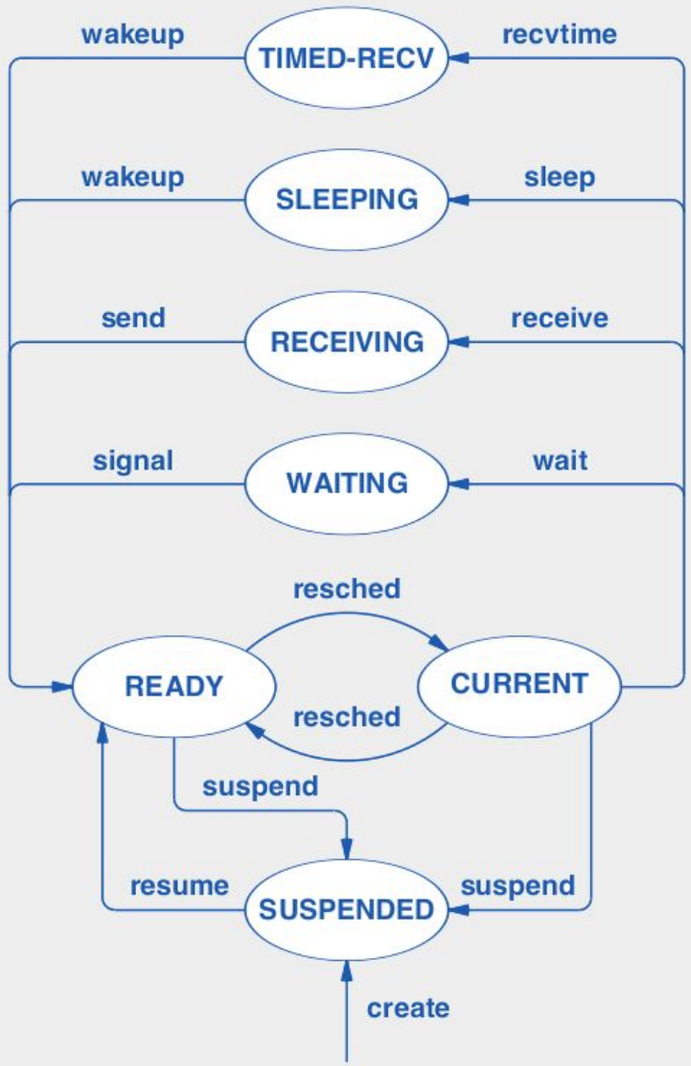


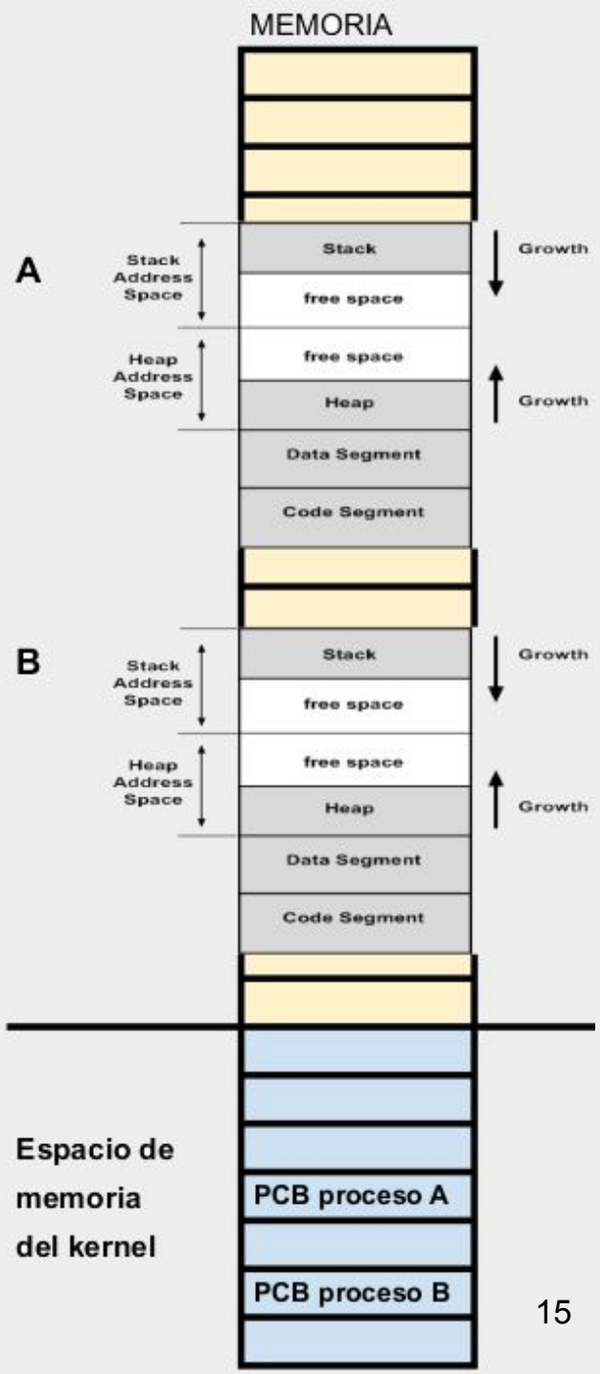


Sistemas Operativos I - Procesos : planificación

Estados de un proceso en Xinu

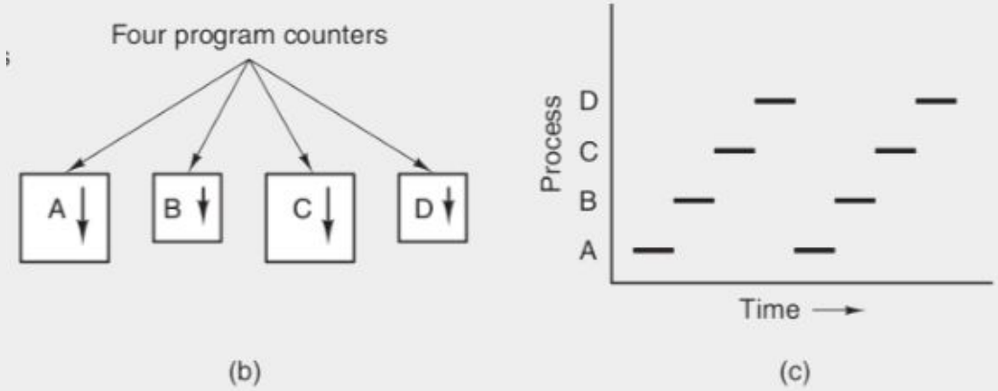
```
/* Creación de proceso en XINU */

#include <xinu.h>

void    sndA(void);

/*------------------------------------------------------------------------
 * main  --  example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
       int pid;

       pid = create(sndA, 128, 20, "process 1", 0) );
       resume(pid);
       sleep(10);
       kill(pid);
}

/*------------------------------------------------------------------------
 * sndA  --  repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
       while( 1 )
              putc(CONSOLE, 'A');
}
```

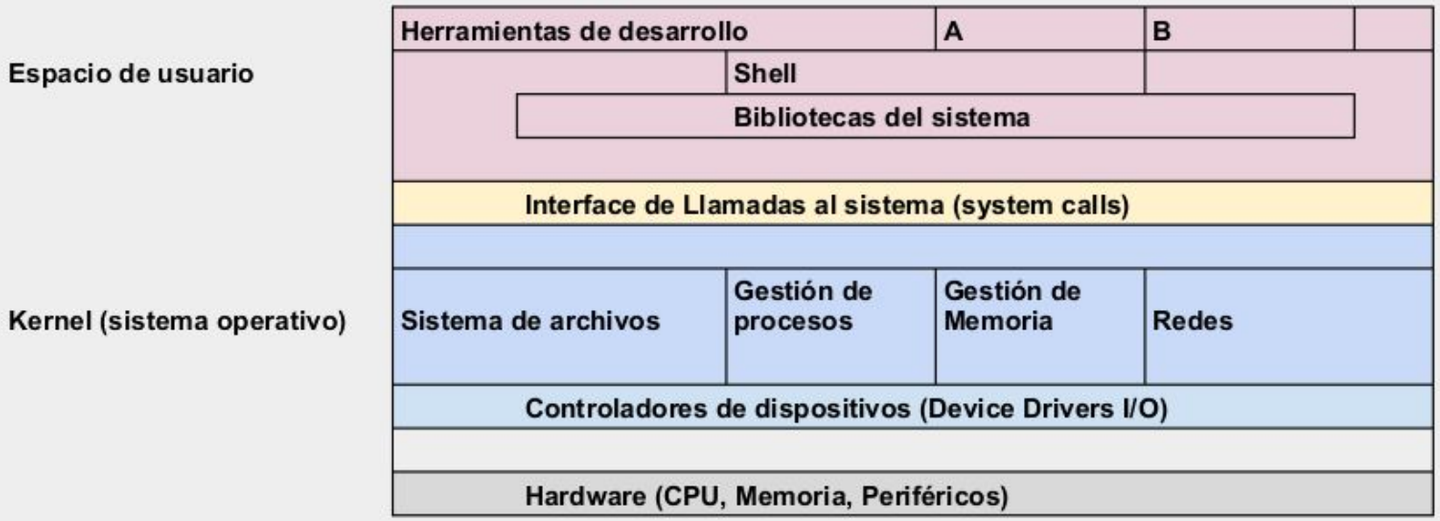


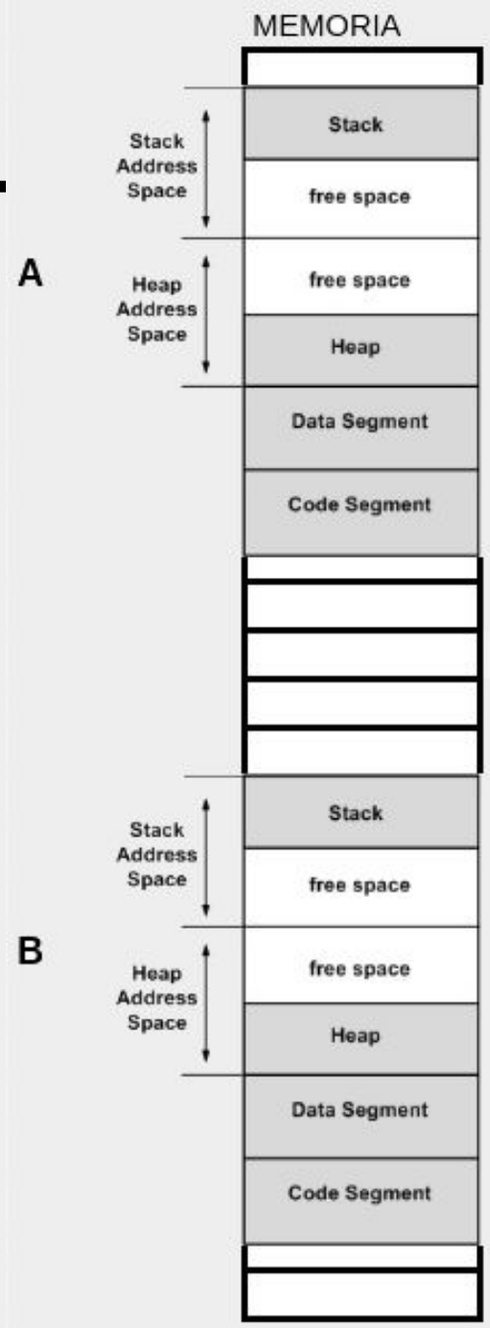


Sistemas Operativos I - Procesos y Threads

Ejecución concurrente - cambio de contexto







Sistemas Operativos I - Procesos y Threads

Implementación de procesos concurrentes

Cambio de contexto

Arreglo de estructuras.
Tabla de procesos (PCB)

Resguardar el estado del procesador
para el proceso A

Cargar el estado anterior del procesador
para el proceso B

Estado del procesador:
Registros (pc, stack pointer, otros registros)

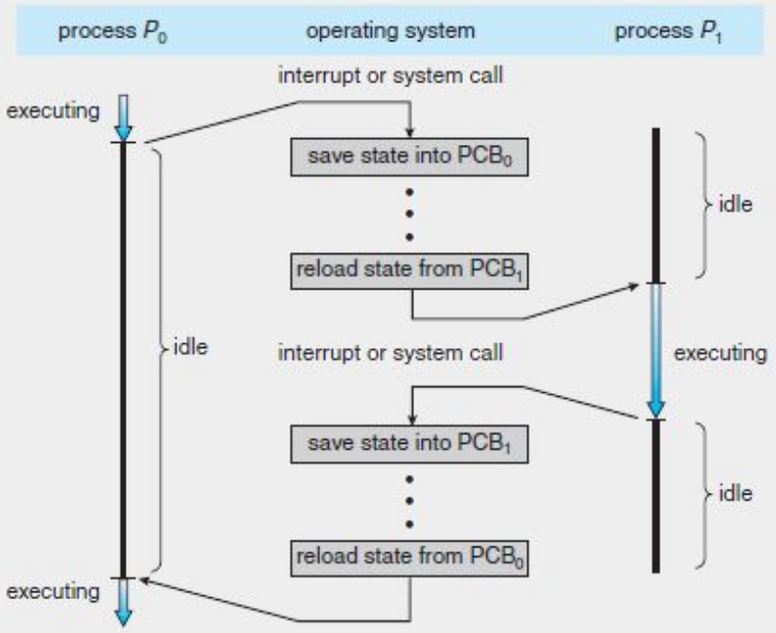
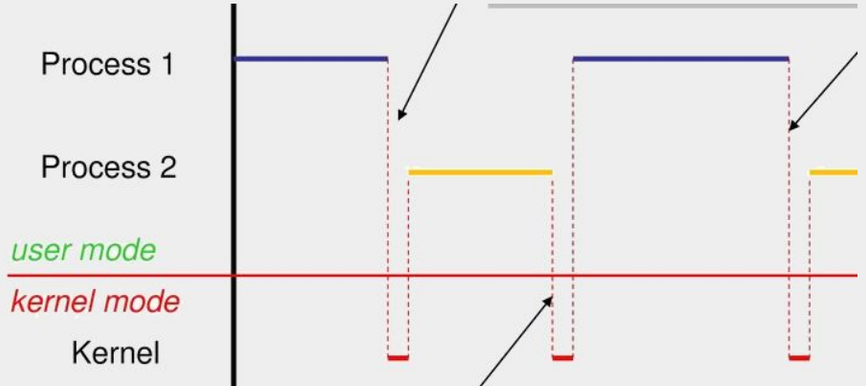
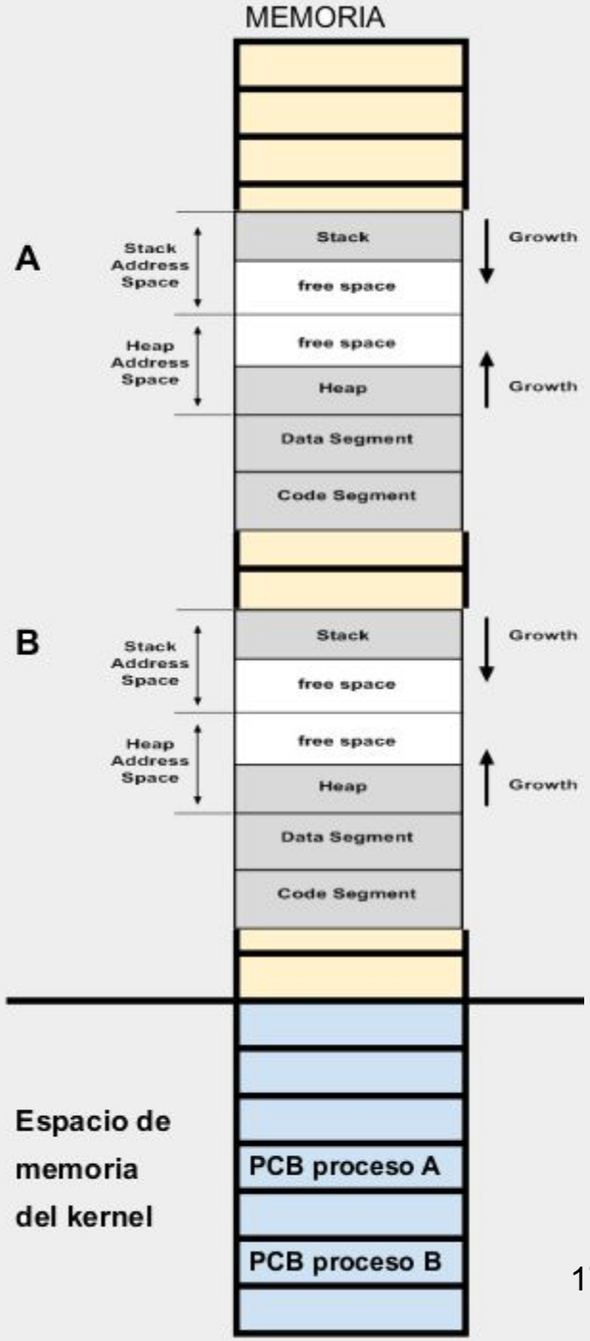


Figure 3.4 Diagram showing CPU switch from process to process.



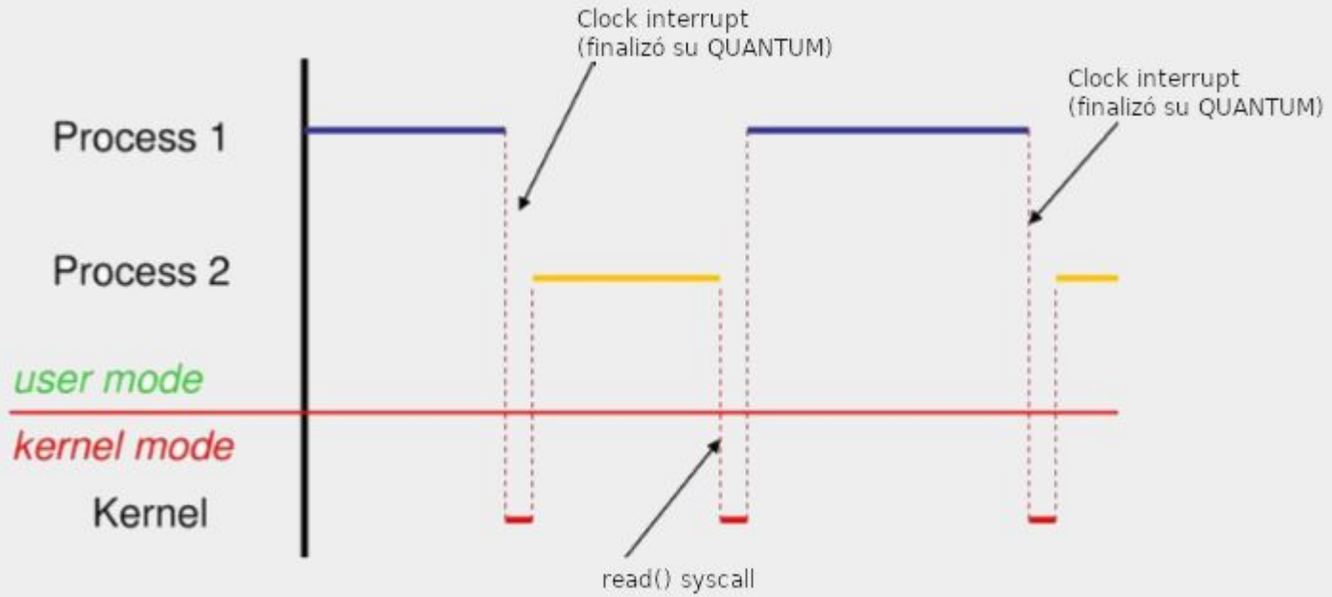


Sistemas Operativos I - Procesos y Threads

Implementación de procesos concurrentes

Un proceso "libera" el uso de CPU cuando:

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)

Clase 3 - Temario

- Estados de un Proceso
- Conceptos de planificación de procesos
- Sincronización entre procesos

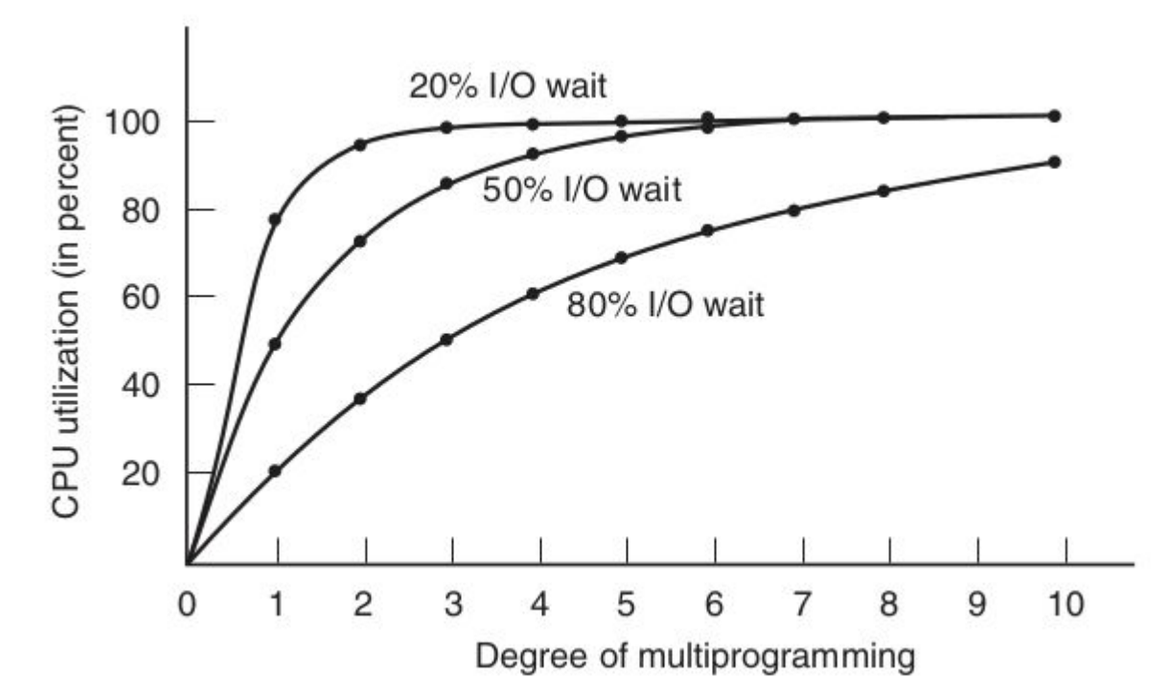
Grado de multiprogramación - Modelo sencillo

Utilización de CPU = $1 - p^n$

p: fracción de tiempo de un proceso esperando E/S
n: cantidad de procesos

Ejemplo: Sistema con 8GB de RAM.
2GB para el SO
2GB para cada proceso.
80% del tiempo esperando E/S.

¿Conviene tener mas RAM?



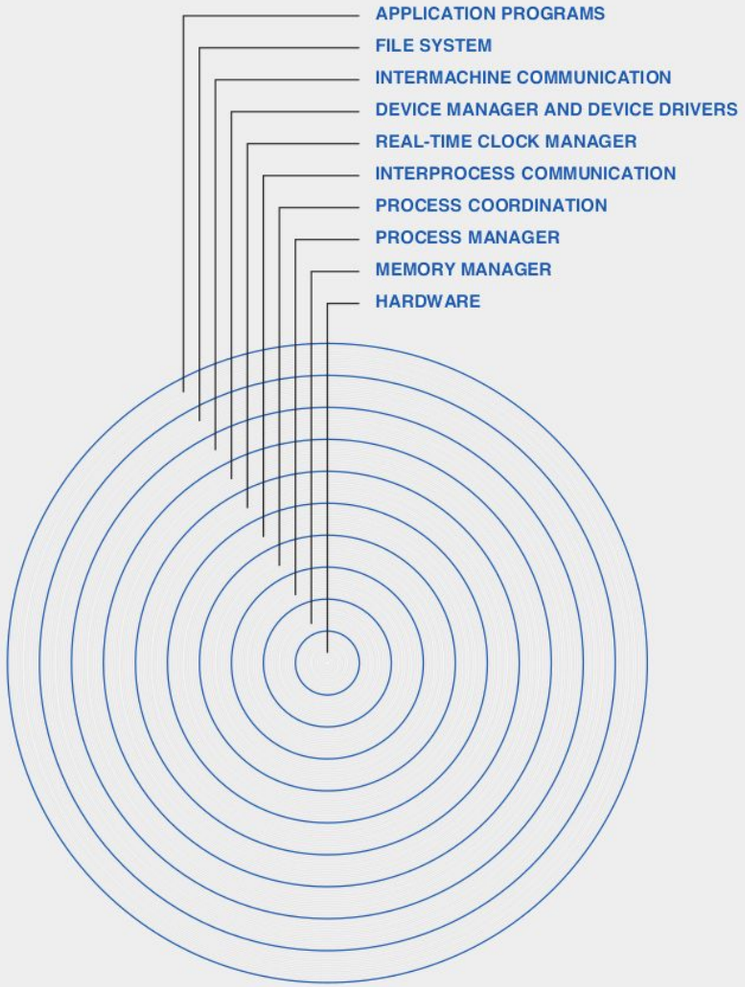
Sistemas Operativos I - Procesos : planificación

Clase 4 - Temario

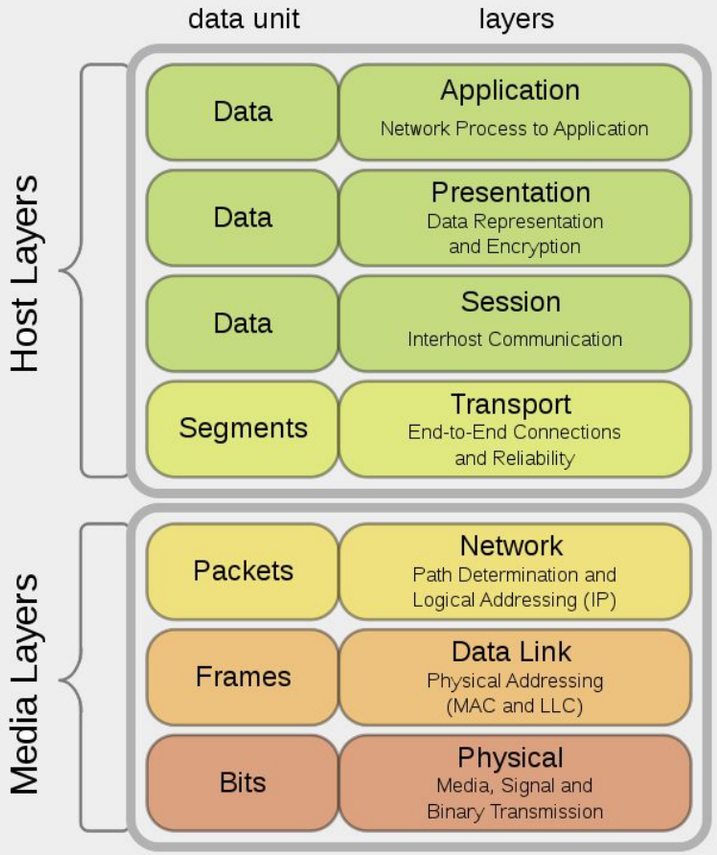
- Comparativa de jerarquía multinivel vs sistema en capas
- Estados de los procesos
- Algoritmos de Planificación de CPU
- Sincronización entre procesos
    - Race condition
    - Semáforos
    - Exclusión mutua
    - Monitores

Sistemas Operativos I - Estructura de un OS

Dos tipos de diseño: Jerarquía multinivel (Xinu) **vs** **Sistema en Capas (TCP/IP - ISO/OSI)**



¿Linux?



Sistemas Operativos I - Procesos

Estados de un proceso

A medida que un proceso se ejecuta va cambiando su estado.

Modelo con algunos posibles estados:

- **Nuevo**: Se crea (nace) un proceso.
- **Ejecutando**: el proceso tiene asignada la CPU
- **Esperando**: el proceso se encuentra en espera que suceda algú
  evento.
- **Listo**: el proceso está esperando para obtener la CPU.
- **Finalizado**: el proceso terminó y sale del sistema (muere).

Estados de un proceso

Sistemas Operativos I - Procesos

Estados de un proceso

A medida que un proceso se ejecuta va cambiando su estado.

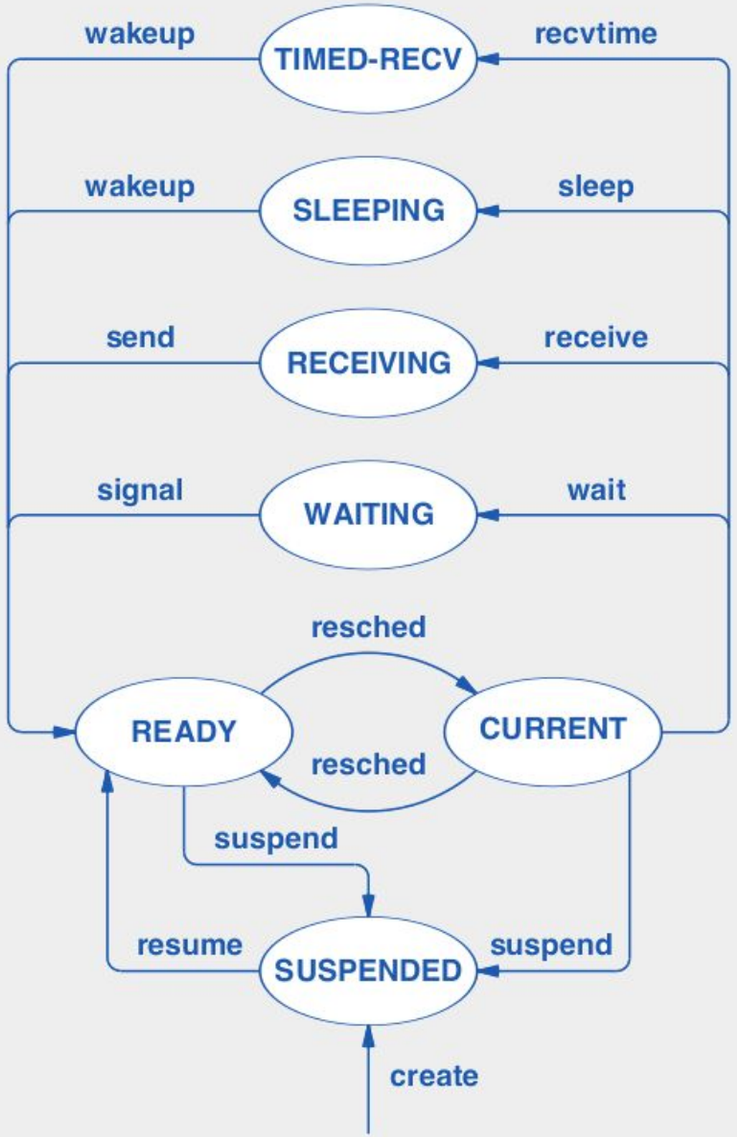
Modelo con varios estados posibles:

- **Nuevo**: Se crea (nace) un proceso.
- **Ejecutando**: el proceso tiene asignada la CPU
- **Bloqueado**: el proceso se encuentra en espera que suceda algún evento.
- **Listo**: el proceso está esperando obtener la CPU.
- **Listo suspendido**: el proceso está Listo, pero el OS lo colocó en me secundaria (swap).
- **Bloqueado suspendido**: el proceso se encuentra en espera a que s algún evento. El OS lo colocó en memoria secundaria (swap)

Sistemas Operativos I - Procesos

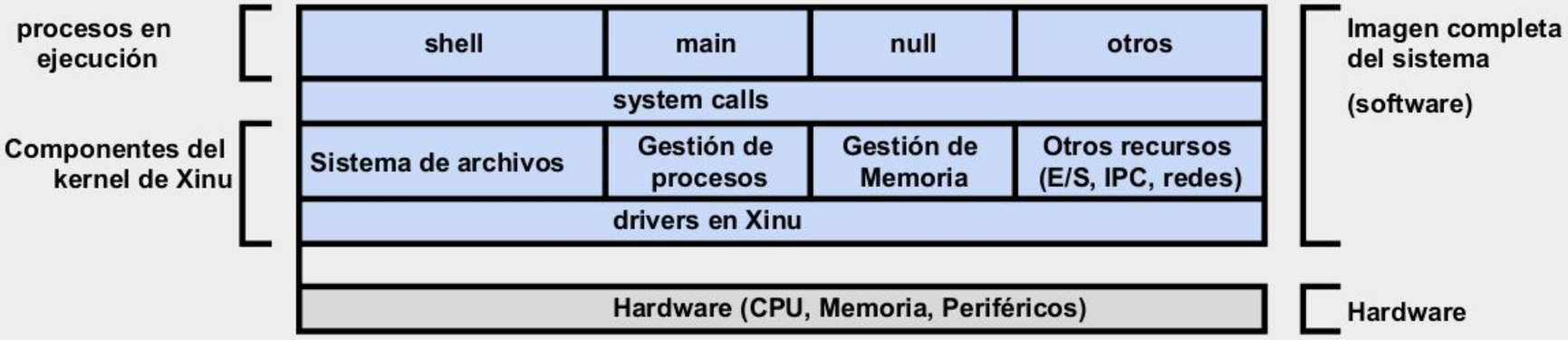
Estados de un proceso en Xinu (EJEMPLO)

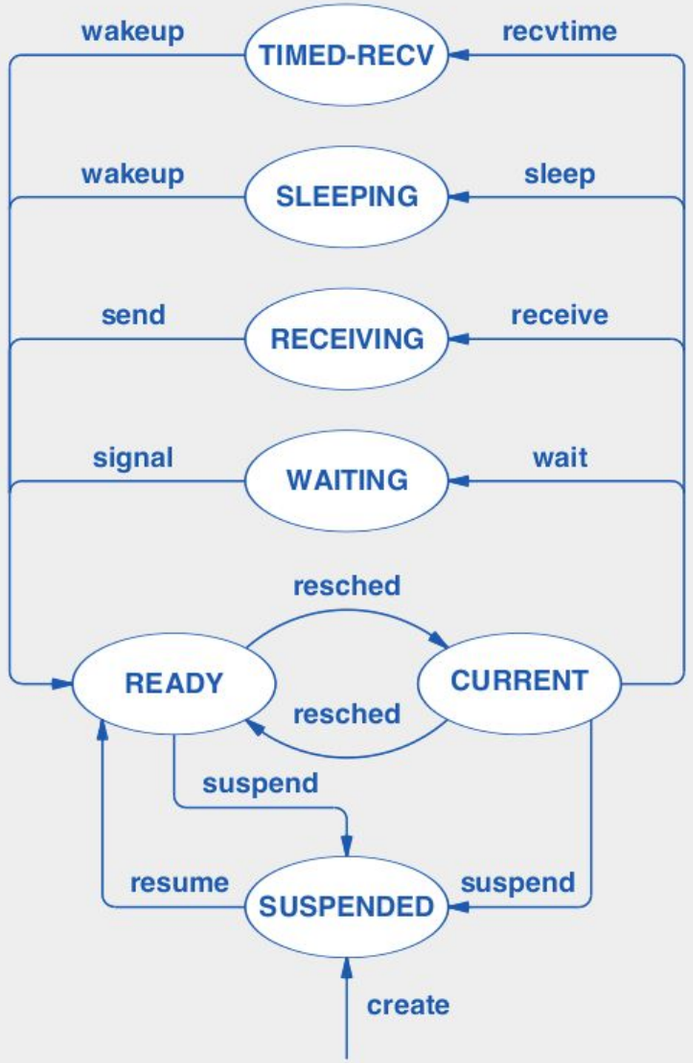
```
/* Creación de proceso en XINU */

#include <xinu.h>

void sndA(void);

void main(void)
{
     int pid;

     pid = create(sndA, 128, 20, "process 1", 0) );
     resume(pid);
     sleep(10);
     kill(pid);
}


/* program sndA  --  repeatedly emit 'A' on the console */

void sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}
```



Sistemas Operativos I - Procesos

Estados de un proceso en Xinu y Diagrama de componentes





Repaso: ¿Qué es un sistema operativo? ¿Qué recursos ofrece?

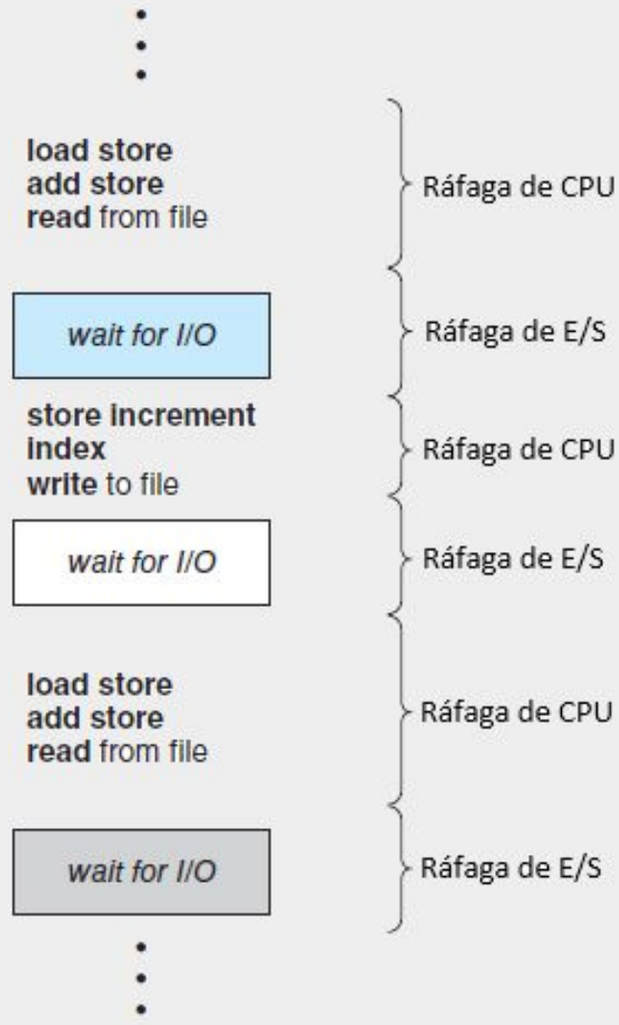
Conceptos (terminología) a conocer y relacionar:

Procesos cooperativos. Apropiativo. Multiprogramación. Sistema Compartido. Servicios. LLamadas al sistemas. Administrar recursos. Ofrecer servicios (de recursos físicos o nuevos/logicos). CPU modo supervisor/usuario, interrupciones, estados de un proceso. Entrega voluntaria de la CPU (sleep). Llamadas al sistema bloqueantes. Espacio del kernel. Espacio de usuario. PCB. Procesos.

Sistemas Operativos I - Procesos : planificación

Planificación de procesos

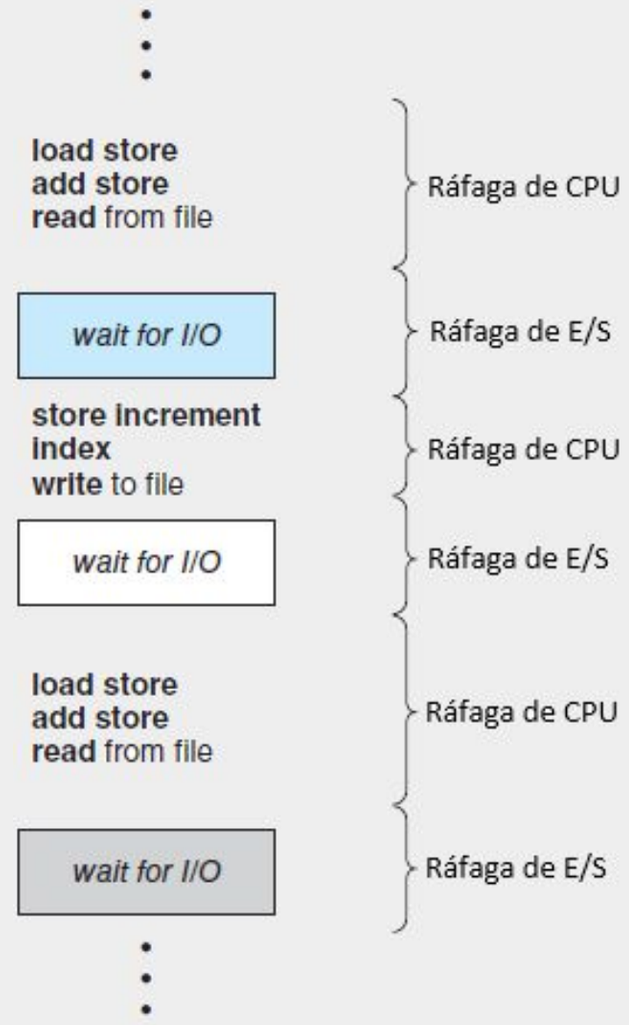
- La multiprogramación permite maximizar la utilización de la CPU.
- Los procesos poseen secuencias alternadas de ráfagas de CPU y de E/S.
- **Proceso cargado en E/S**: tiene muchas ráfagas pequeñas de CPU. El tiempo de ejecución depende principalmente del tiempo que realiza operaciones de E/S.
- **Proceso cargado en CPU**: ráfagas grandes de CPU. Realiza mucho cómputo y ocasionalmente realiza operaciones de E/S



Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Conceptos distintos categorizados

- Metas
- Mecanismos (Algoritmos de planificación de CPU)
- Políticas



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)
- Utilización de CPU : No mantenerla ociosa — Batch
- Throughput: Cantidad de trabajos por unidad de tiempo
- Tiempo de turnaround: Minimizar tiempos desde arribo a finalizado
- Tiempo de espera

- Tiempo de respuesta: ante un requisito o evento — Sistemas interactivos
- Proporcional: a todos los usuarios

- Cumpliendo tiempos de respuesta — Tiempo Real
- Predecible

Sistemas Operativos I - Procesos : planificación

Planificación de procesos - Algoritmos de planificación

Los principales son los siguientes:

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRTF (Shortest Remaining Time First)
- RR (Round Robin)
- Prioridades

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.

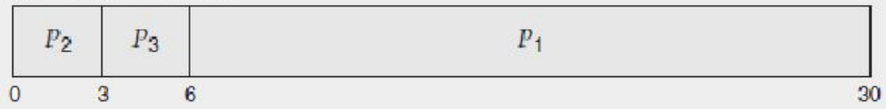
Es un algoritmo no preemptive: cuando un proceso obtiene la CPU no la deja hasta finalizar su ráfaga.

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:

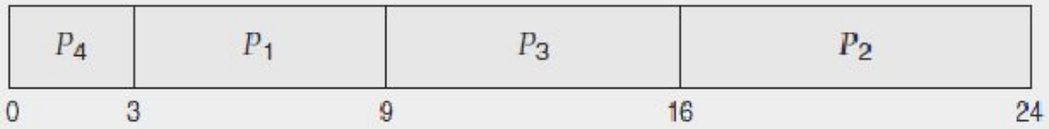
Una vez que un proceso obtiene la CPU,
no puede ser desalojado de ella hasta que finalice su intervalo.

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)

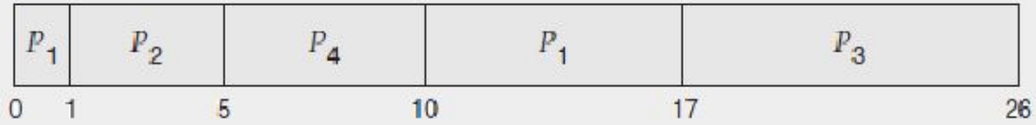
Es una implementación preemptive de SJF.

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

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.

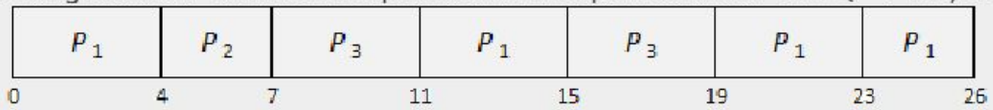
Sistemas Operativos I - Procesos : planificación

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

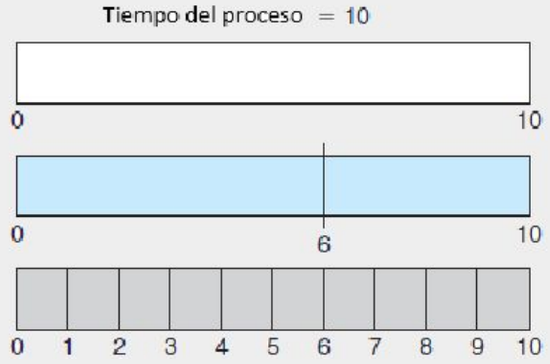
- Suponer que los procesos arribaron a la cola de listos en el orden P1, P2 y P3.
- El diagrama de Gantt correspondiente a la planificación con Q=4 ms, es:



- 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: los procesos de baja prioridad pueden demorarse mucho en ser atendidos o no ser atendidos nunca.

Una solución es implementar un mecanismo de aging:
a medida que pasa el tiempo, se aumenta la prioridad de los procesos (los de baja prioridad aumentan sus chances de ser elegidos para CPU).

Una variación del algoritmos es dar alta prioridad (dinámicamente por el planificador) a procesos con mucha carga de E/S.

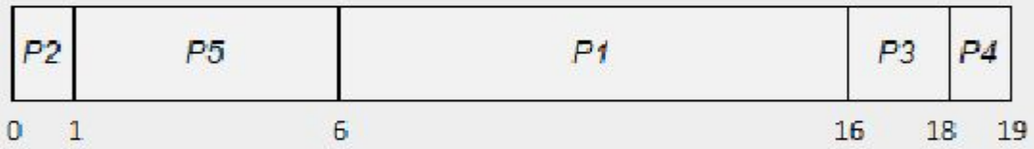
Por prioridades es combinado generalmente con Round Robin para sistemas preemptive (ejemplo Xinu)

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

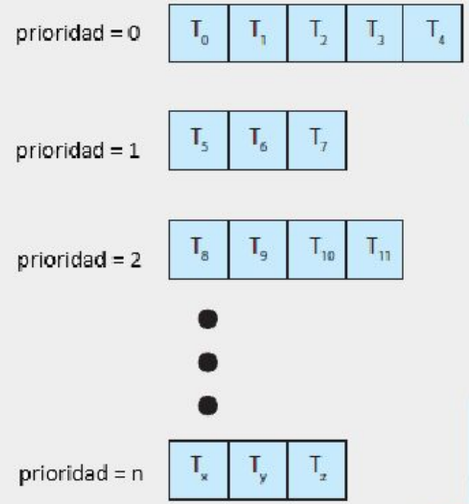
- El diagrama de Gantt correspondiente a la planificación es:



- 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

Otra variante

- La planificación por prioridades se combina con RR.
- La cola de listos está dividida en colas separadas según el tipo de proceso:
  - cola para los procesos interactivos.
  - cola para los procesos batch.
- Cada cola tiene su propio algoritmo de planificación:
  - RR para cola de procesos interactivos.
  - FCFS para cola de procesos batch.



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.

```
/* 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 con procesos cooperativos concurrentes

Condición de carrera y falta de sincronización

- Condición de carrera
- Recurso compartido
- Sección crítica

```
#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

```
#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);
        }
}
```

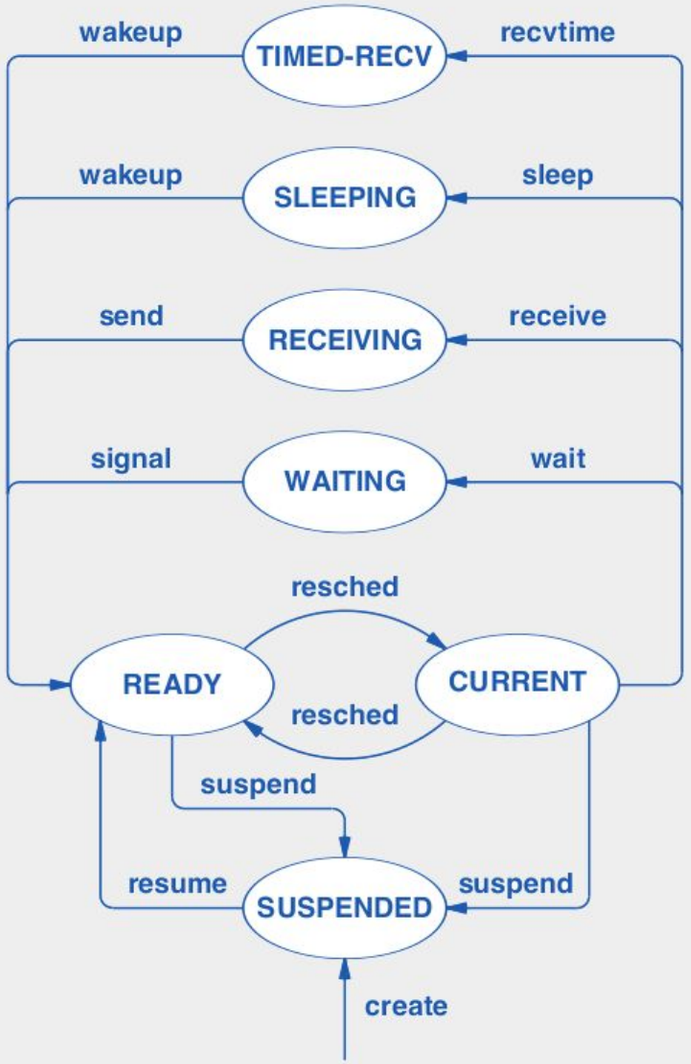
```
/* Canta Joaquin Galan */
void    canta_joaquin(void)
{
        int   i;

        /* 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    main(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 implementado por el sistema operativo para sincronización de procesos.

- Las aplicaciones pueden acceder al recurso a través de **system calls.**

- Un Semáforo S es una variable entera.
- Solo puede accederse a través de 2 operaciones atómicas: **wait() y signal()**.
    - Semáforo Contador: valor entero.
    - Semáforo Binario: el valor entero entre 0 y 1 (similar a mutex lock).



Semáforo

recurso lógico implementado por el sistema para sincronización de procesos.

Implementación

- **De software:**
  - Algoritmos de Sección Crítica - *CUIDADO, puede fallar diría el mago Black*
- **De Hardware**
  - Inhibición de interrupciones
  - Instrucciones de máquina que se ejecutan atómicamente (sin interrupción):
    - Test-And-Set: testear y modificar el contenido de un word.
    - Compare-and-swap(): intercambiar el contenido de dos words

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);
}
```

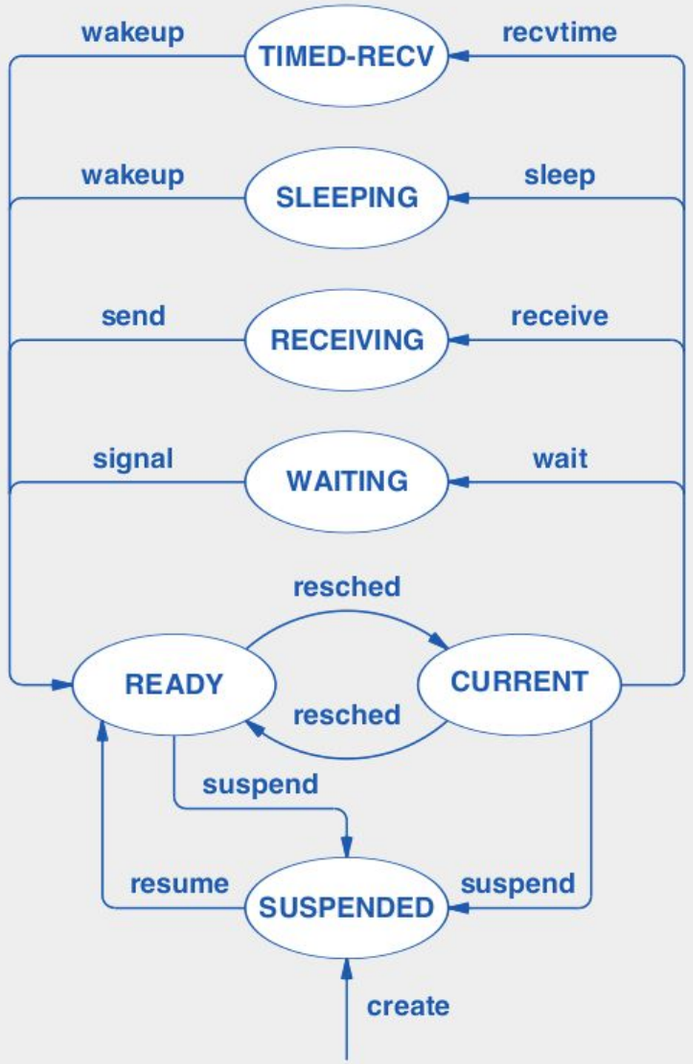
```
/* Programa de Usuario : productor de cine */

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 sección crítica:
  - Primero se debe ejecutar acquire() 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.



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

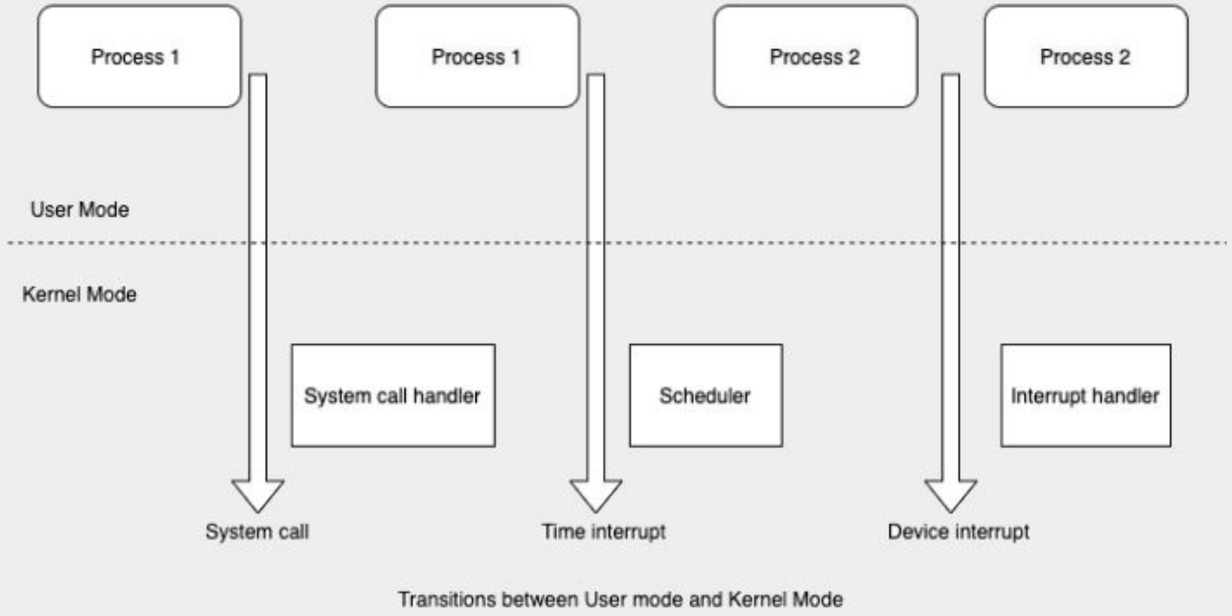
Clase 5 - Temario

- Cuando desaloja un proceso la CPU

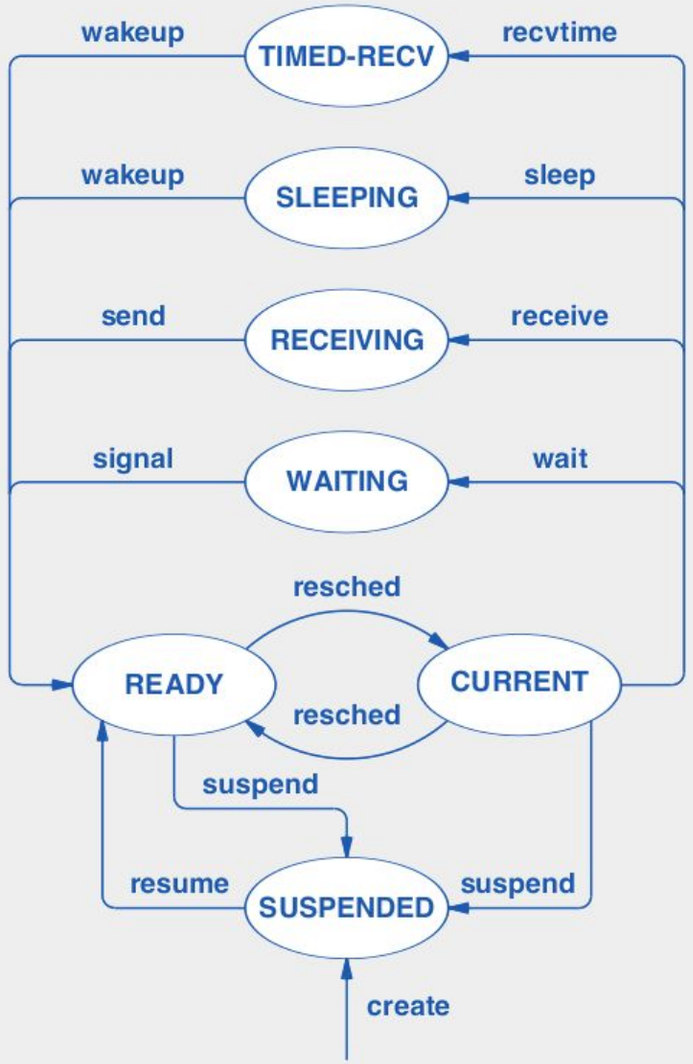
- Interrupciones o Excepciones (de E/S o de system call). Caso de Linux/UNIX vs Xinu.
- Ejemplo de Llamada al sistema bloqueante.
- Llamadas al sistema indirectas. El caso de printf(). ¿Bloqueante?
- Llistado de system calls en Xinu vs Llbc, donde se encuentran

- Interbloqueos
- Inversión de prioridades

Sistemas Operativos I - Repaso de algunos conceptos



Transitions between User mode and Kernel Mode

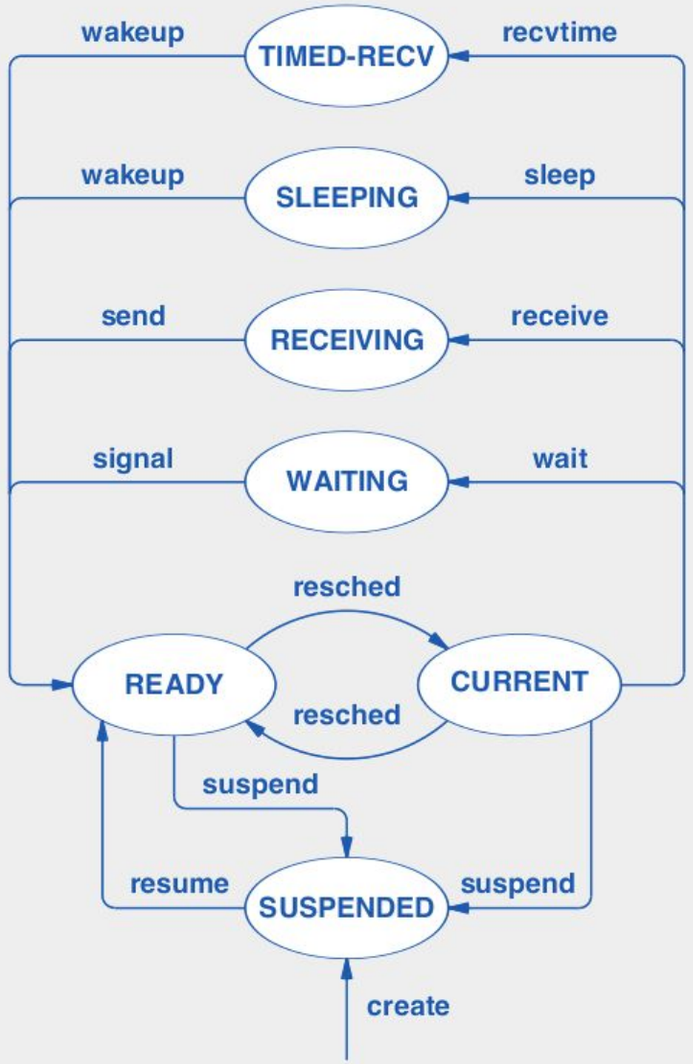


Interrupciones o Excepciones. Caso de Linux/UNIX vs Xinu.
Ejemplo de Llamada al sistema bloqueante.
Llamadas al sistema indirectas. El caso de printf(). ¿Bloqueante?

Sistemas Operativos I - Repaso de desalojo de CPU

```
/* Ejemplo de programa en XINU */

#include <xinu.h>

void    sndA(void);

char saludo[] = “Hola mundo”;

/*------------------------------------------------------------------------
 * main  --  example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
       int pid;

       pid = create(sndA, 128, 20, "process 1", 0) );
       resume(pid);
       resume(create(sndA, 128, 20, "process 2", 0) ));
       sleep(10);
       kill(pid);

}

/*------------------------------------------------------------------------
 * sndA  --  repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
       char t[80];
       memcpy(t, saludo, strlen(saludo)+1);
       while( 1 )
                printf(“%s! \n”, t);

}
```

¿Cuántos procesos hay en ejecución? - ¿Y cuando termina main, finaliza process 2?
¿Cuáles son llamadas al sistema y cuales funciones de la biblioteca de C?

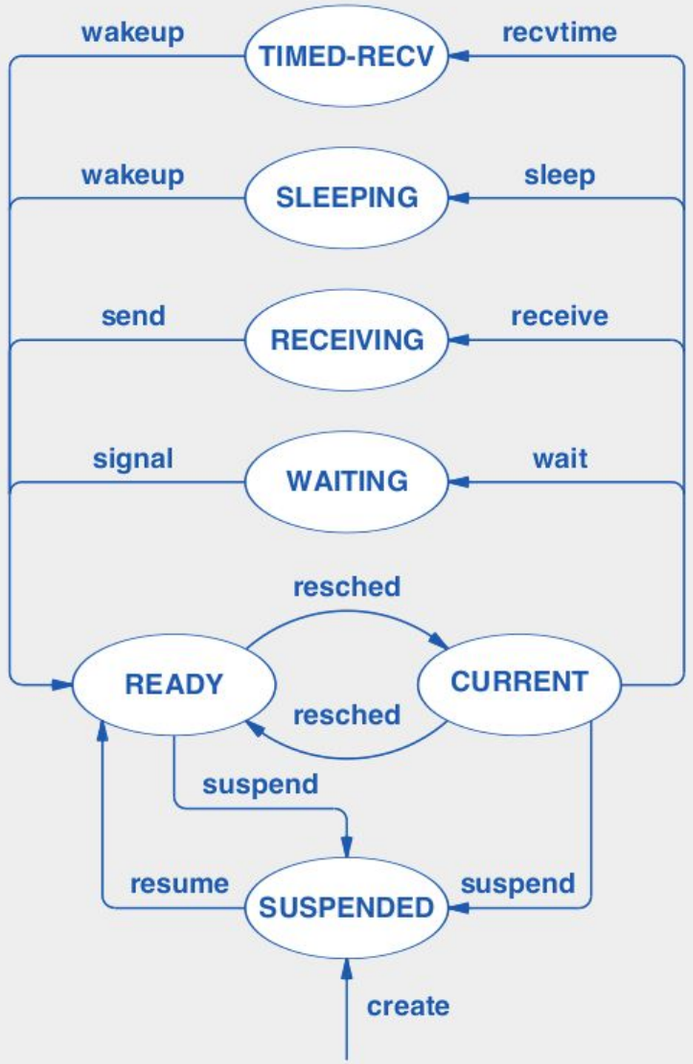


Sistemas Operativos I - Sincronización entre procesos

Deadlock : Dos o mas procesos están esperando indefinidamente por un evento que solo puede ser causado por uno de los procesos en espera.

Ejemplo: P0 y P1 son 2 procesos. S y Q son dos semáforos inicializados con el valor 1.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

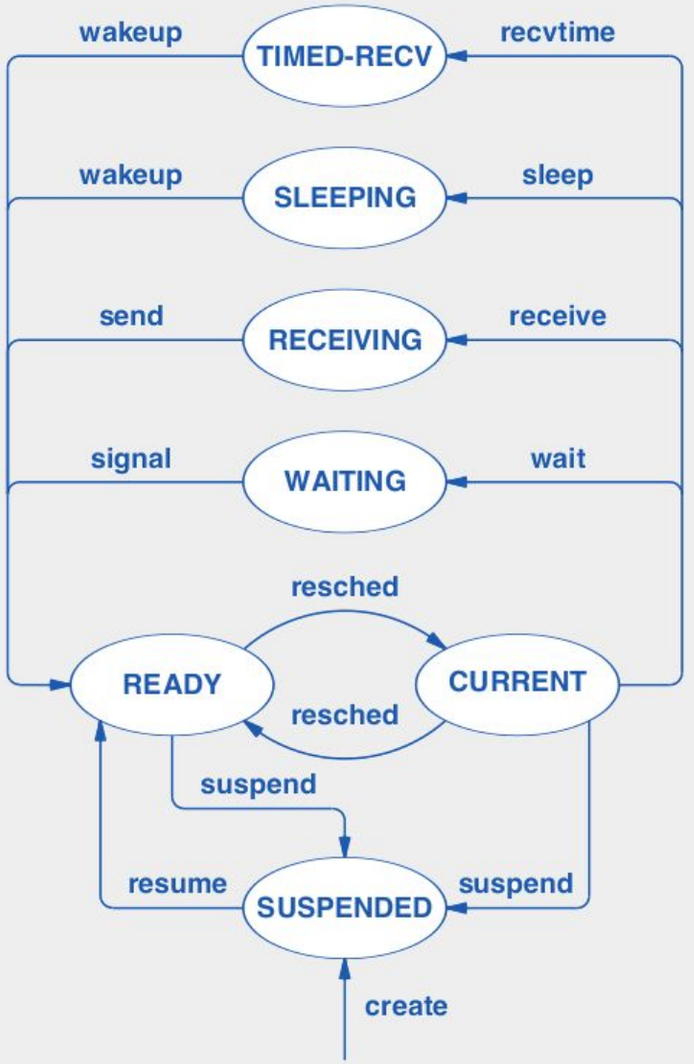


Deadlock : Condiciones - Coffman et al. (1971)

- **Exclusión Mutua**
- **Retención y espera (Hold and wait)**
- **No desalojo (no preemption)**
- **Espera circular**

Cómo gestionar deadlocks:

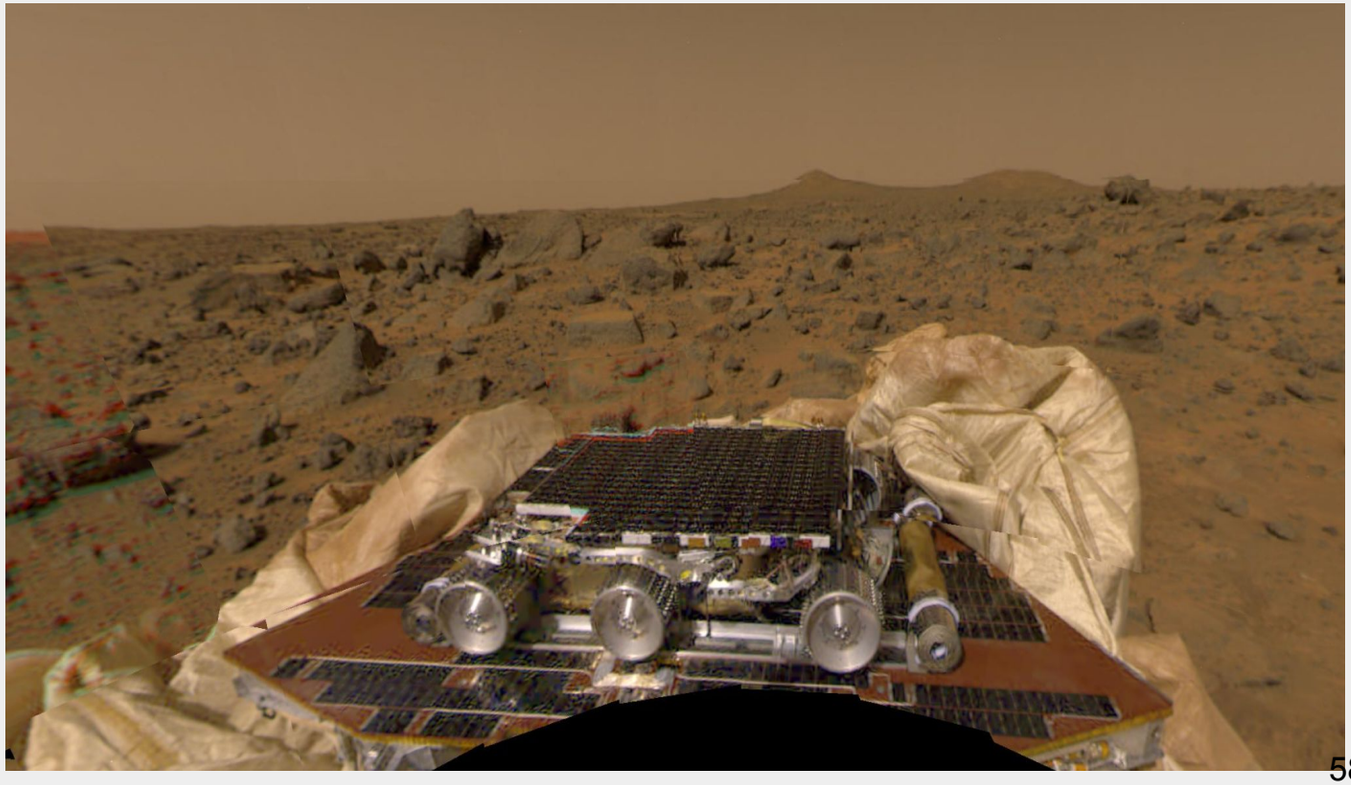
• Prevenir que ocurra una de las cuatro condiciones. En la práctica, la última.
• Evitar: utilizando the banker's algorithm
• Con un algoritmo de detección (analizar procesos y recursos), y algoritmo de recuperacióin (abortar un proceso, o apropiarse de un recurso).



Sistemas Operativos I - Sincronización entre procesos

Inversión de Prioridades : Ejemplo Mars Pathfinder (1997)

- Al menos 3 procesos en un sistema con planificación por prioridades.
- Un proceso con baja prioridad retiene un recurso
- El proceso de más alta prioridad requiere el recurso
- Un proceso con prioridad intermedia utiliza la CPU



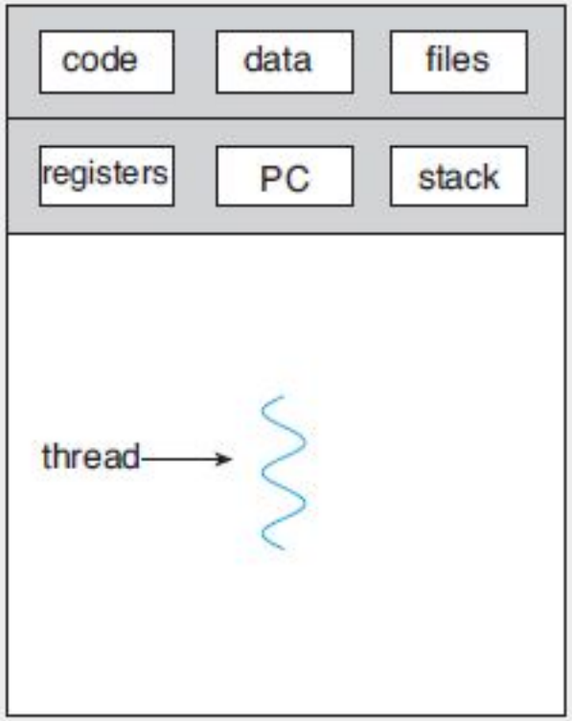
Threads

- Unidad básica de utilización de la CPU.
- Tiene su propio ID, PC, registros, y stack.
- Comparte la sección de código, datos y otros recursos (como archivos abiertos).

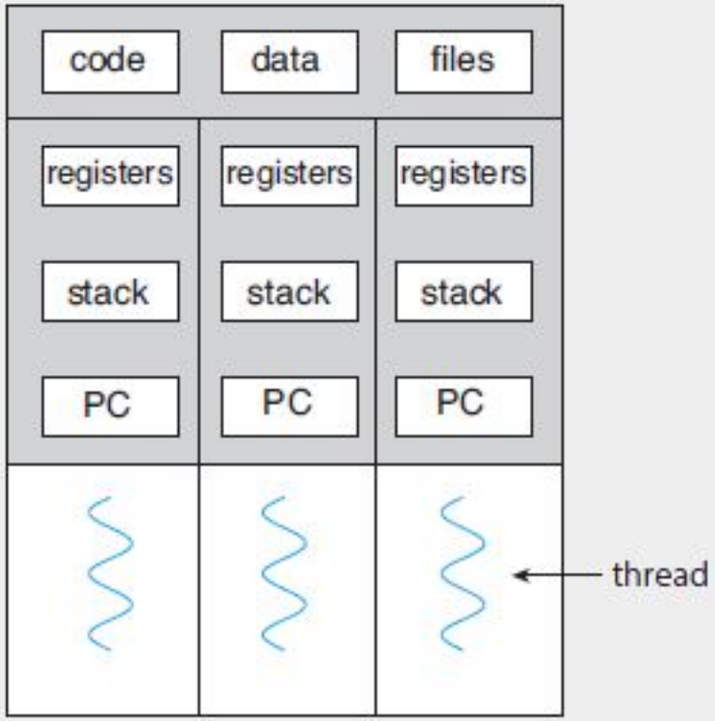
Otras denominaciones:

- Hilo de control
- Proceso de peso liviano (LWP)
- Thread de control

Un thread NO ES UN PROCESO HIJO



single-threaded process



multithreaded process

Threads

Si...

- n procesos en un sistema con m CPU son concurrentes/paralelos
  - comparten recursos del SO (dispositivos lógicos, semáforos, etc)

Si...

- n threads en un proceso con m CPU son concurrentes/paralelos
  - comparten recursos del proceso

Threads

Si...

- n procesos en un sistema con m CPU son concurrentes/paralelos
  - comparten recursos del SO (dispositivos lógicos, semáforos, etc)

Si...

- n threads en un proceso con m CPU son concurrentes/paralelos
  - comparten recursos del proceso

Entonces...¿tiene algún sentido tener THREADS?????

Threads

Si...

- n procesos en un sistema con
  - comparten recursos del S

Si...

- n threads en un proceso con m
  - comparten recursos del p

Entonces...¿tiene algún sentido te

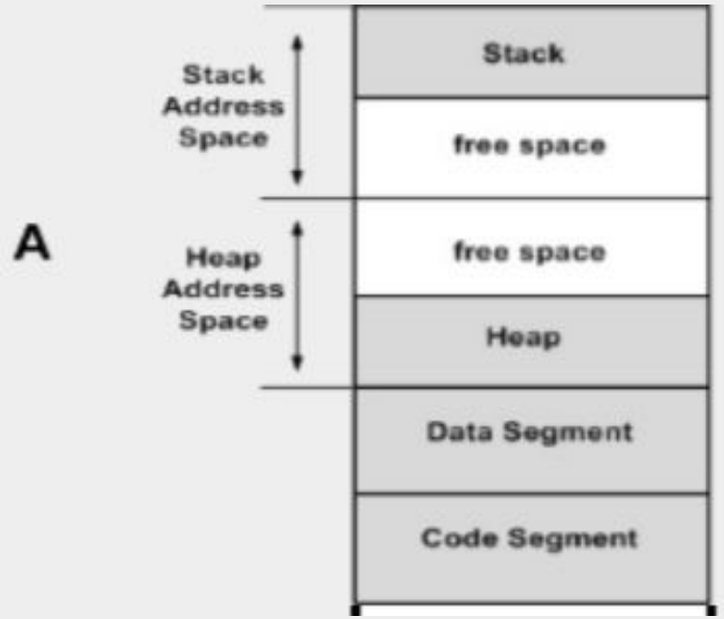
Threads

- Es la unidad básica de utilización de la CPU.
- Tiene su propio ID, PC, registros, y stack.
- Comparte la sección de código, datos y otros recursos (como archivos abiertos).

Otras denominaciones:

- Hilo de control
- Proceso de peso liviano (LWP)
- Thread de control

Repaso: ¿Cómo implementa un SO el control de los procesos?



Threads

Algunos posibles beneficios:

- Capacidad de respuesta
- Compartir recursos
  - Menor overhead en:
    - la creación y finalización de threads
    - el cambio de contexto.
- Escalabilidad

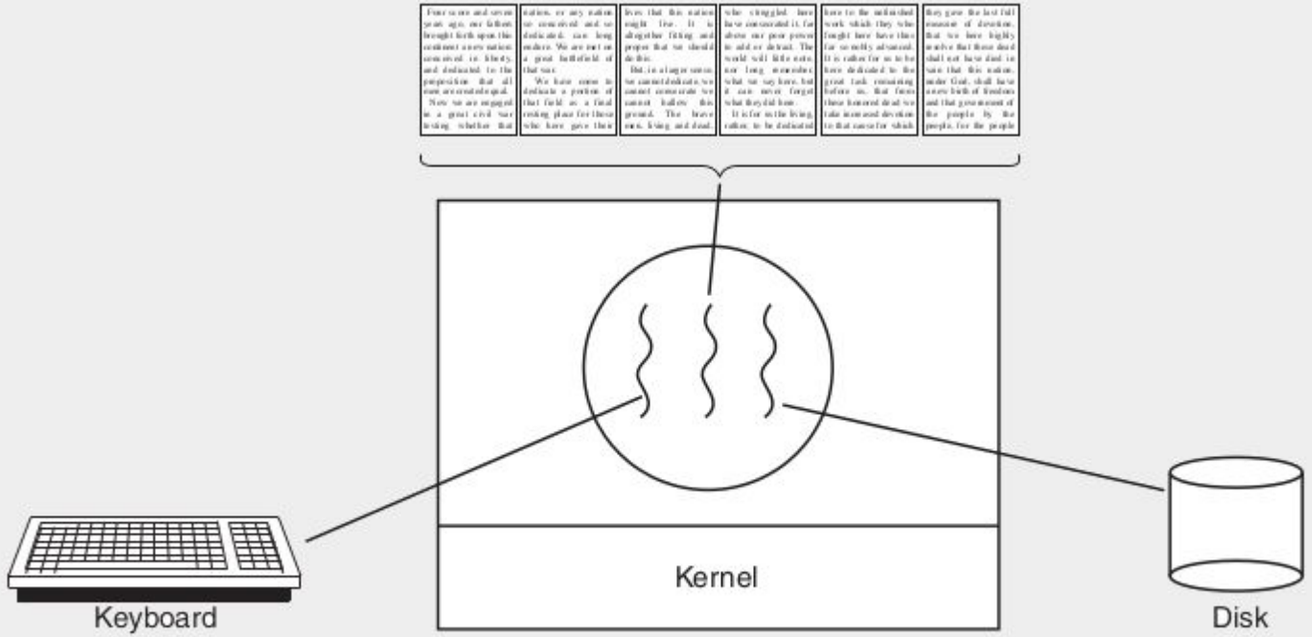


Figure 2-7. A word processor with three threads.

Threads

Algunos posibles beneficios:

- Capacidad de respuesta: un thread espera I/O, otro utiliza CPU, etc.
- Compartir recursos: compartir codigo y datos, archivos abiertos, etc.
  Menor overhead en:
  la creación y finalización de threads
  el cambio de contexto.
- Escalabilidad: aprovechar multiples CPU y tener paralelismo real.

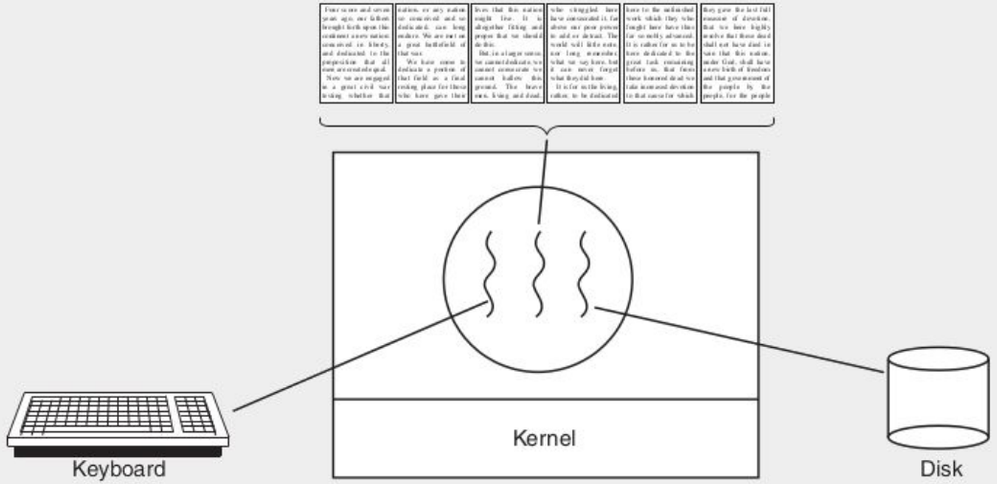
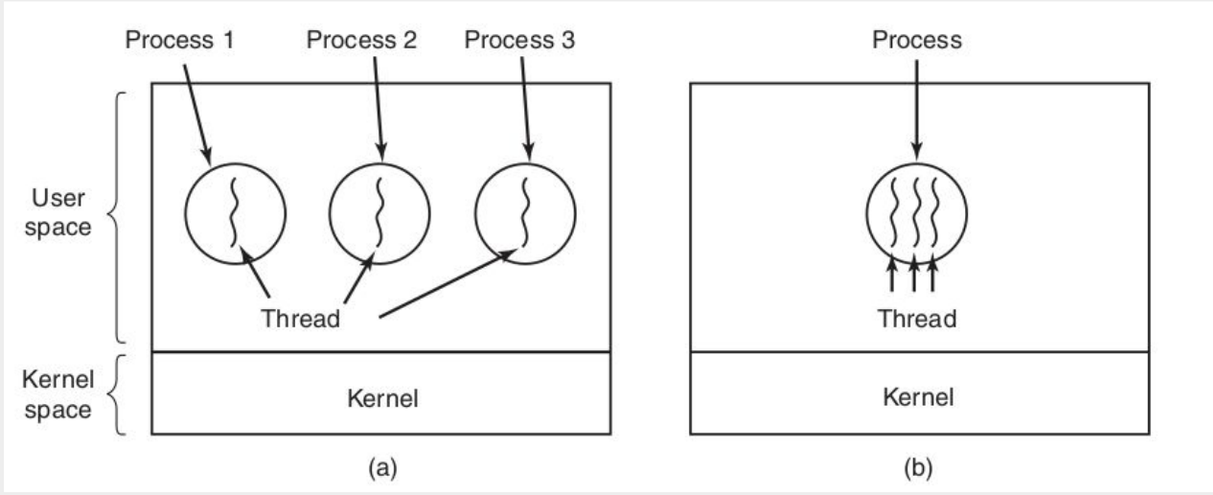


Figure 2-7. A word processor with three threads.

Threads - Implementación de threads



Con el fin de ser portables, la IEEE definió un estándar:

POSIX threads 1003.1c. Pthreads.
(Casi todos los UNIX soportan el estándar.)

Repaso: ¿Qué era POSIX y para qué es útil?

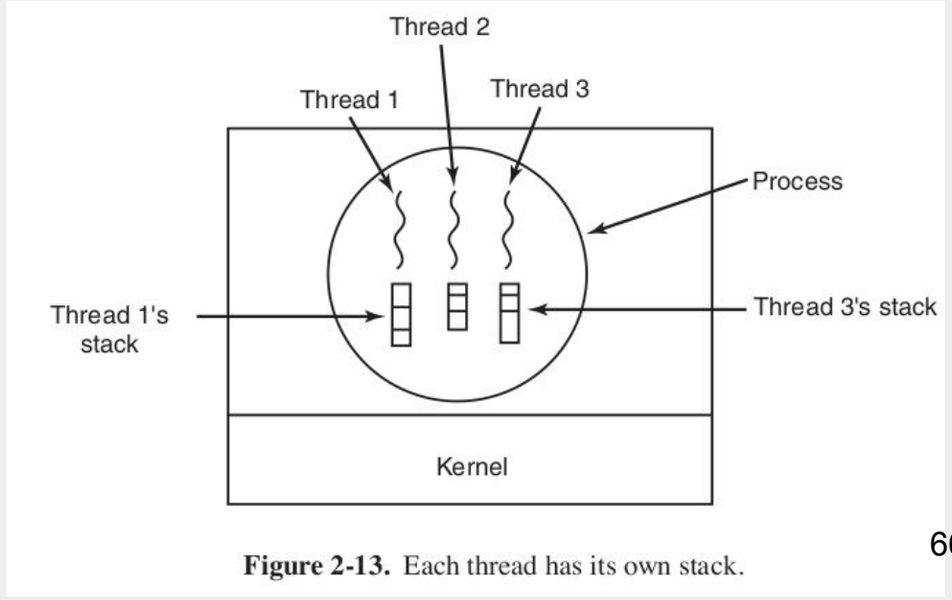


Figure 2-13. Each thread has its own stack.

Threads - Implementación de threads

- En espacio de usuario.
  En el ambiente de tiempo de ejecución (run-time system)
- En espacio del kernel

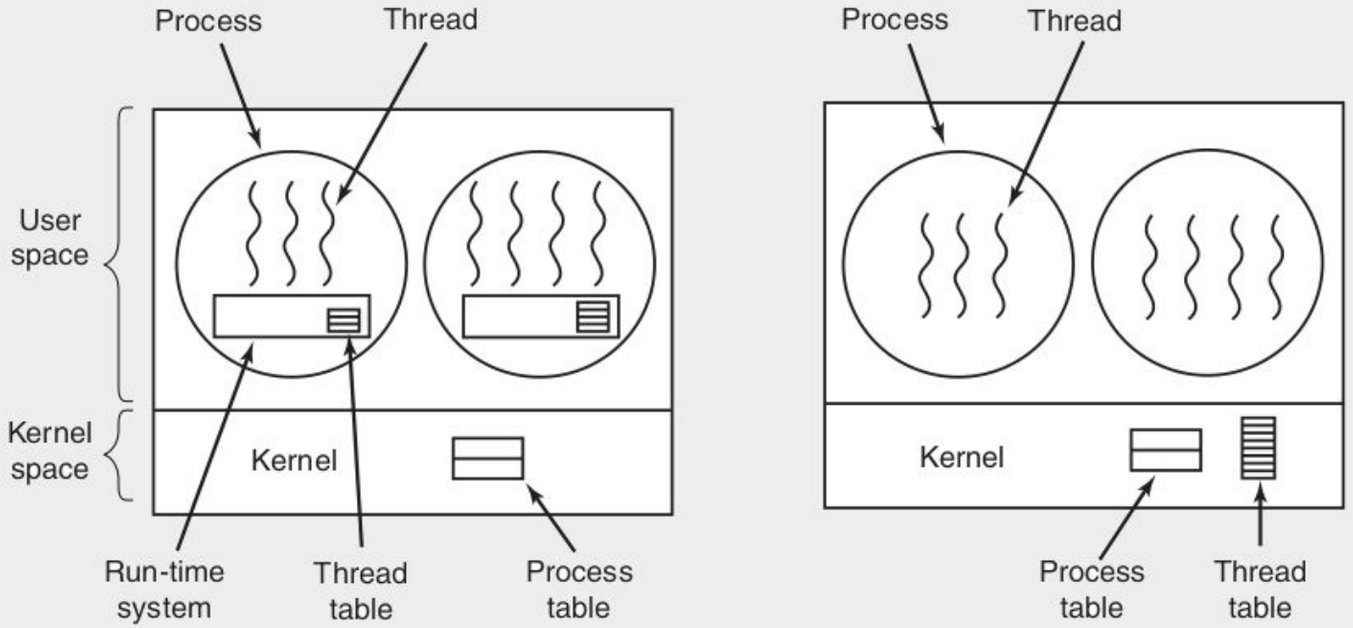


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Threads - Implementación de threads

Implementación en espacio de usuario

Ventajas:

- No es necesario pasar a modo kernel ni ejecutar código del SO. Mejor rendimiento que kernel threads.
- Aún mejor que planificar diferentes procesos (el planificador de threads tiene mejor rendimiento)
- Cada proceso puede tener un algoritmo de planificación diferente para sus threads

Desventajas

- ¿Cómo gestionar las llamadas al sistema bloqueantes?
  Posibilidad : cambiar la API de read, etc. O usar select en UNIX.
- Si existe un page fault el proceso es bloqueado por el SO.
- El thread toma la CPU y no la libera.
  Posibilidad: implementar una señal de clock periódica.
- Las aplicaciones más destinadas a ser resueltas con threads realizan muchas llamadas al sistema bloqueantes.

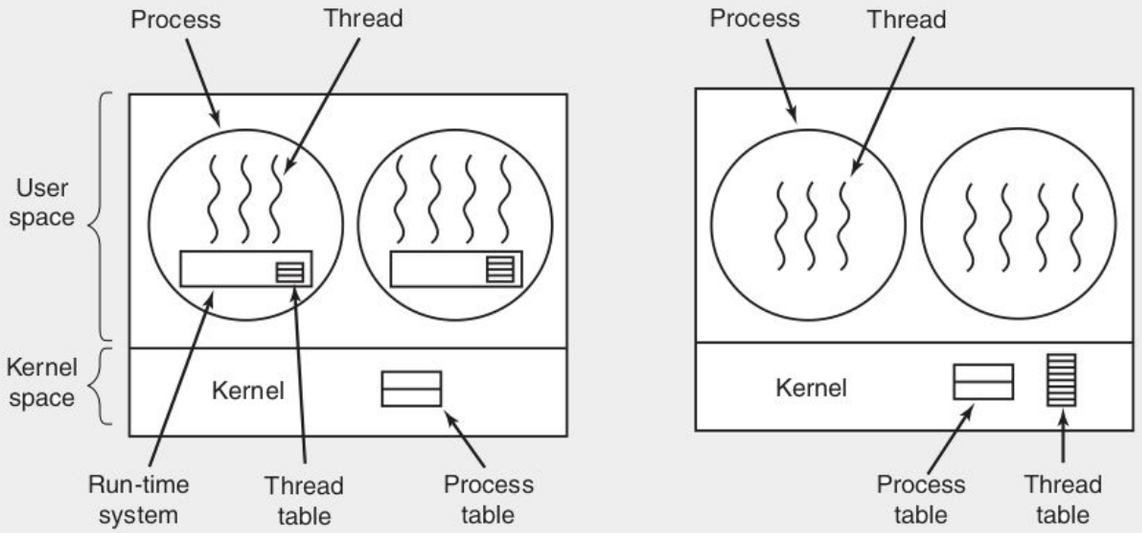


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Threads - Implementación de threads

Implementación en espacio del kernel

Ventajas:

- Soluciona casi todos los problemas presentes en la implementación de threads en espacio de usuario.

Desventajas

- La creación y finalización de threads es “mas costosa”.
  Posibilidad : al finalizar un thread no liberar su gestión y reusarla luego.
- Ante un system call desde un thread el SO podría ejecutar otro thread (bien), pero también podría switchear a otro proceso (ouch).
- Un system call sigue siendo complejo.
- ¿Cómo tratar las señales recibidas por el proceso?
- ¿Cómo tratar a un fork()?

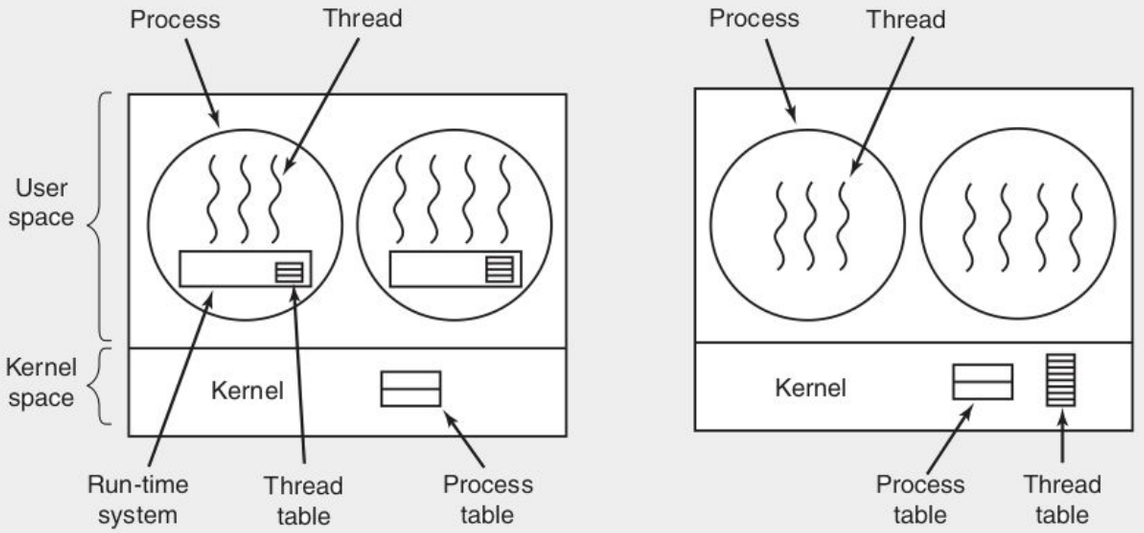


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Sistemas Operativos I - Procesos y Threads

Threads - Implementación de threads

Implementación en Sistemas Operativos

- Linux, Windows, Apple OS y otros sistemas operativos tradicionales de PC y servidores contienen implementación en espacio del kernel.
- También es posible usar threads implementados en espacio de usuario (Java, biblioteca de C posix threads, etc).
- Xinu y otros RTOS: el modelo de procesos es el de threads.
  Desventaja:
  Los procesos no tienen memoria protegida e independiente, todos comparten el segmento de código y de datos.
  Las pilas son independientes, pero podrían llegar a solaparse y corromper el sistema.

```
/* Ejemplo Linux. Compilar con: gcc -o p p.c -lpthread */

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

void *p_msg( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *msg1 = "Thread 1";
     char *msg2 = "Thread 2";
     int  r1, r2;

    /* Create independent threads each of which will execute function */
     r1 = pthread_create( &thread1, NULL, p_msg, (void*) msg1);
     r2 = pthread_create( &thread2, NULL, p_msg, (void*) msg2);

     /* Wait till threads are complete before main continues.  */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",r1);
     printf("Thread 2 returns: %d\n",r2);
     exit(0);
}

void *p_msg( void *ptr )
{
     char *m;
     m = (char *) ptr;
     printf("%s \n", m);
}
```

Documentación de system calls y de la biblioteca de C en Linux

man 2 intro
man 2 syscalls
man 2 ipc
man 7 libc

man 5 proc # ver /proc/pid/status
man 5 sysfs

man 7 standards