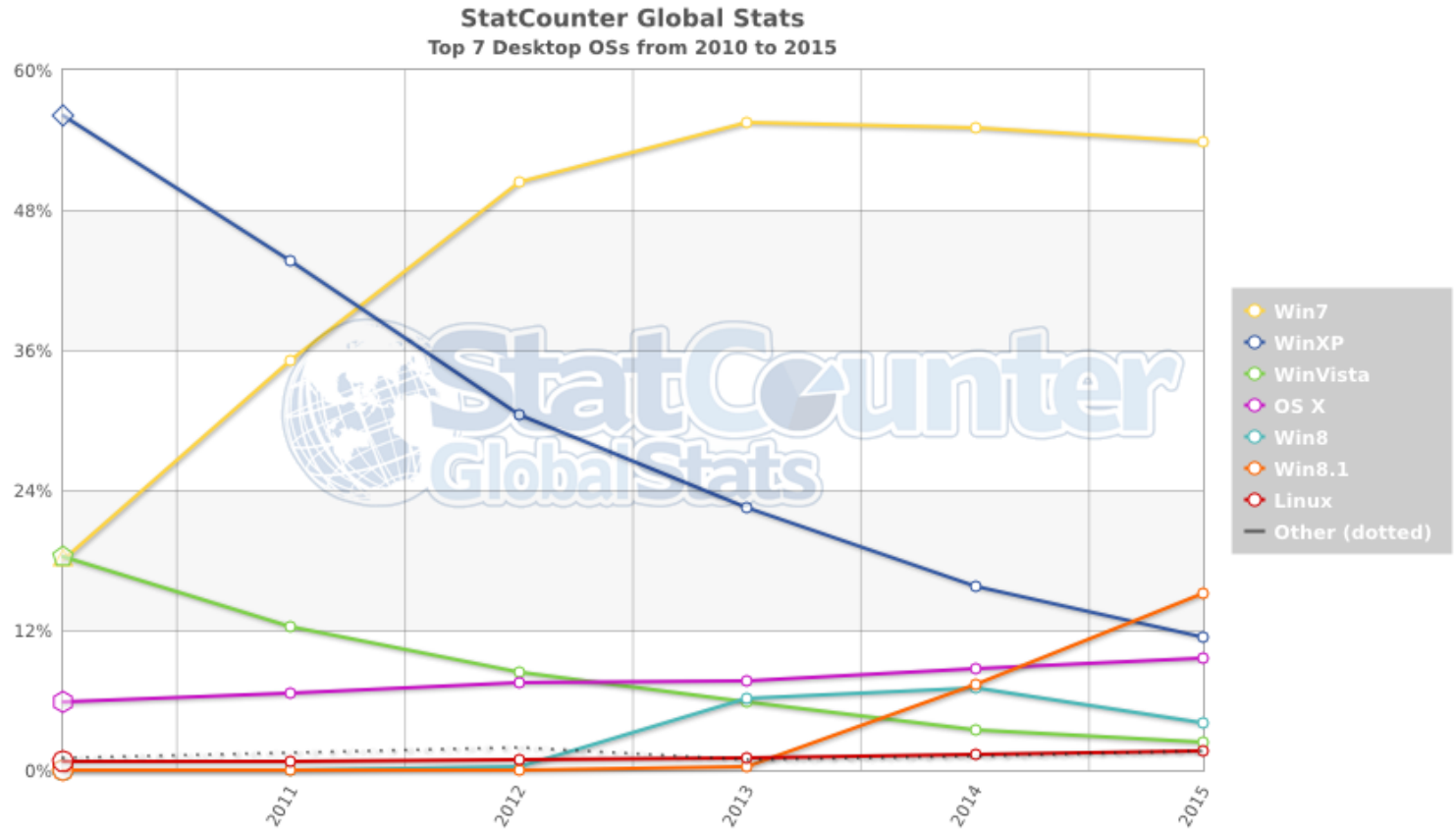
Sistemas Operativos

Dr. Luis Gerardo de la Fraga

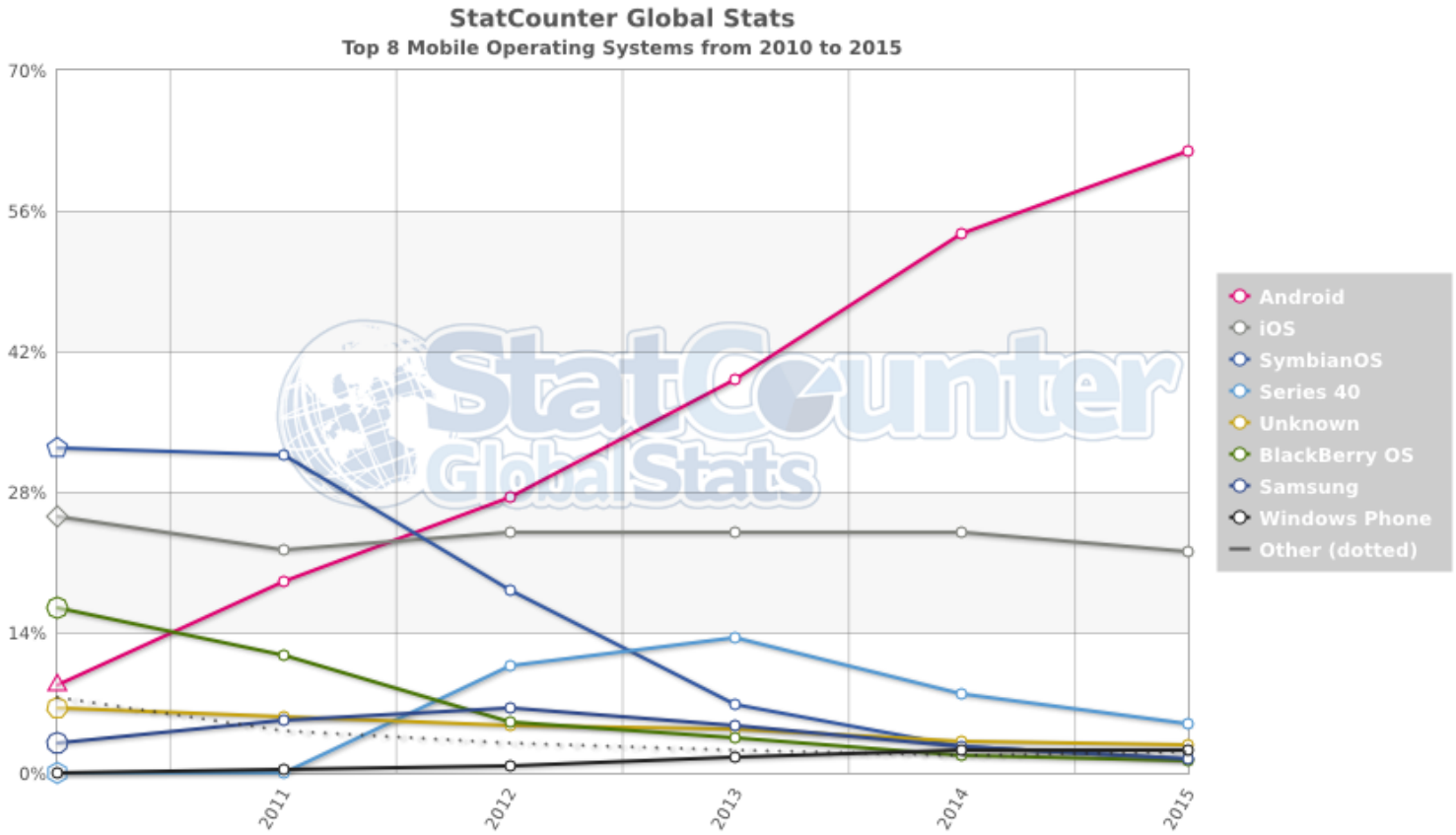
E-mail: fraga@cs.cinvestav.mx
http://cs.cinvestav.mx/~fraga

Departamento de Computación
Cinvestav

25 de mayo, 2015

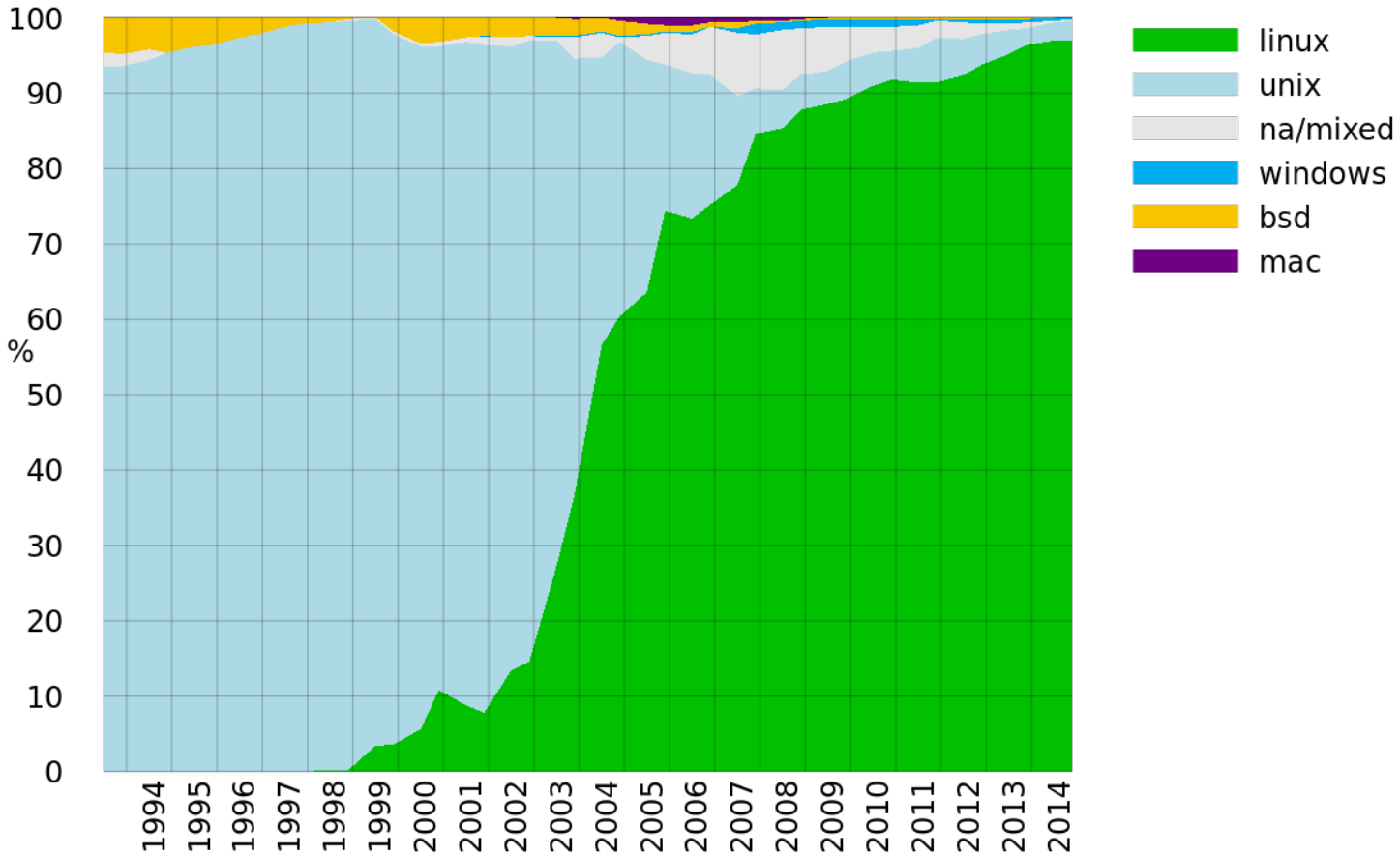


http://gs.statcounter.com



http://gs.statcounter.com

SO en supercomputadoras de 1994 a 2014 de acuerdo con TOP500



`http://en.wikipedia.org/wiki/Usage_share_of_operating_systems`

Procesos

Creación de procesos

- Existen básicamente cuatro formas de crear procesos en UNIX:

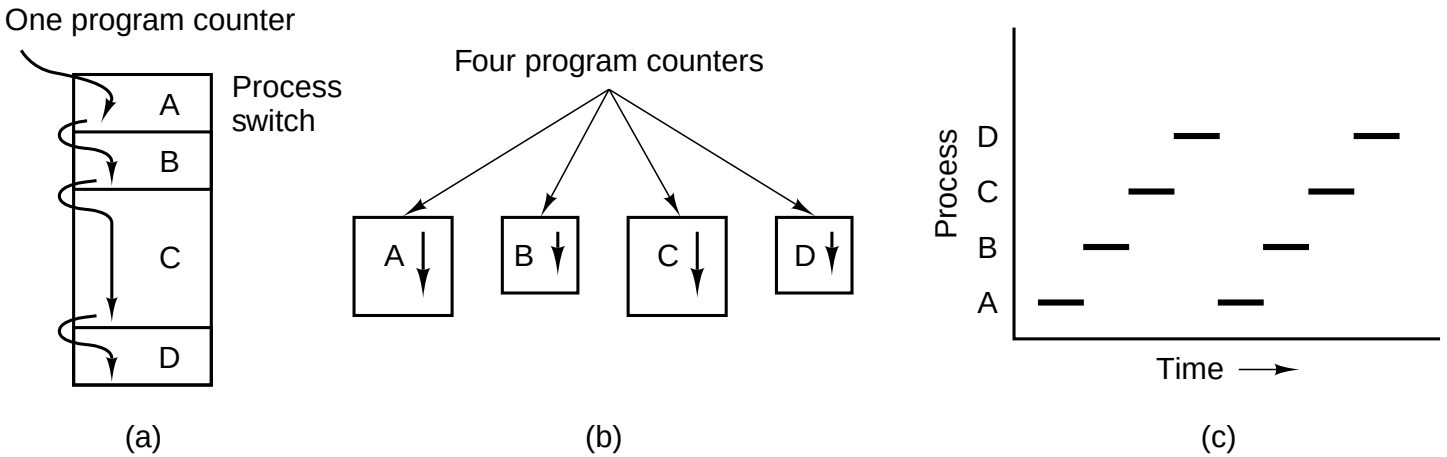
1. En la inicialización del sistema (init)
2. Creación de un proceso por una llamada al sistema (fork) de un proceso que ya está corriendo.
3. Un usuario crea un nuevo proceso (en el shell)
4. Inicialización de un trabajo en lotes (cron y at)

- Los procesos que se quedan en el fondo se les llama *demonios* (de impresión, por ejemplo).

- Ya vimos que un proceso, de forma muy general, es un programa en ejecución.
- Un proceso contiene:
  - un espacio de direcciones (desde 0 hasta un máximo en el que el proceso puede leer y escribir)
  - un conjunto de registros (que incluye el contador de programa, el apuntador a la pila y otros más).
  - Y toda la información necesaria para ejecutar el programa.
- Toda esta información se almacena en una **tabla de procesos**

- En la computadora de están ejecutando varias aplicaciones: un visor web, un editor de texto, un acceso a disco, etc.
- En una computadora con un solo microprocesador, estrictamente hablando solo se ejecuta un programa a la vez, pero en el transcurso de un segundo se ejecutan varios programas dando al usuario la ilusión de paralelismo.
- El cambio rápido del CPU entre varios procesos se le llama **multiprogramación** (y a veces se le llama **pseudoparalelismo**).

- Conceptualmente como puede visualizarse la ejecución de los procesos:
- todos ejecutándose en paralelo como en máquinas separadas, con contadores de programa, registros y variables independientes, con una máquina por proceso.



(a) Multiprogramación de cuatro programas. (b) Modelo conceptual para cuatro procesos independientes y secuenciales. (c) Solo un programa está activo en cualquier instante.

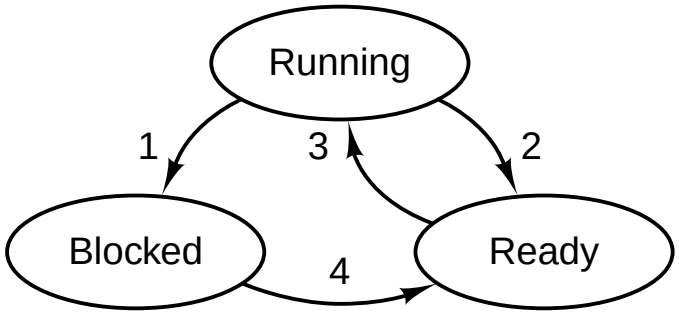
- Un proceso es una actividad, es un programa en ejecución, entonces un proceso contiene: un **programa**, su **entrada** y **salida**, y su **estado**.
- En minix, el proceso inicial se llaman **init**, y se encuentra en la imagen que botea.
- Cuando *init* comienza su ejecución lee un archivo (`\etc\inittab` que le dice cuántas terminales hay y lanzo un proceso (con fork()) por terminal
- Estos procesos esperan que el usuario ingrese a la máquina.
- Si el acceso es exitoso, entonces se lanza un shell.

El despachador

- Estados en los que puede estar un proceso: ejecutándose, bloqueado y listo:

1. Ejecutándose (está usando el CPU en este instante).
2. Listo (puede ejecutarse, pero está parado por otro proceso ejecutándose).
3. Bloqueado (sin poder ejecutarse porque está esperando que un evento externo suceda).

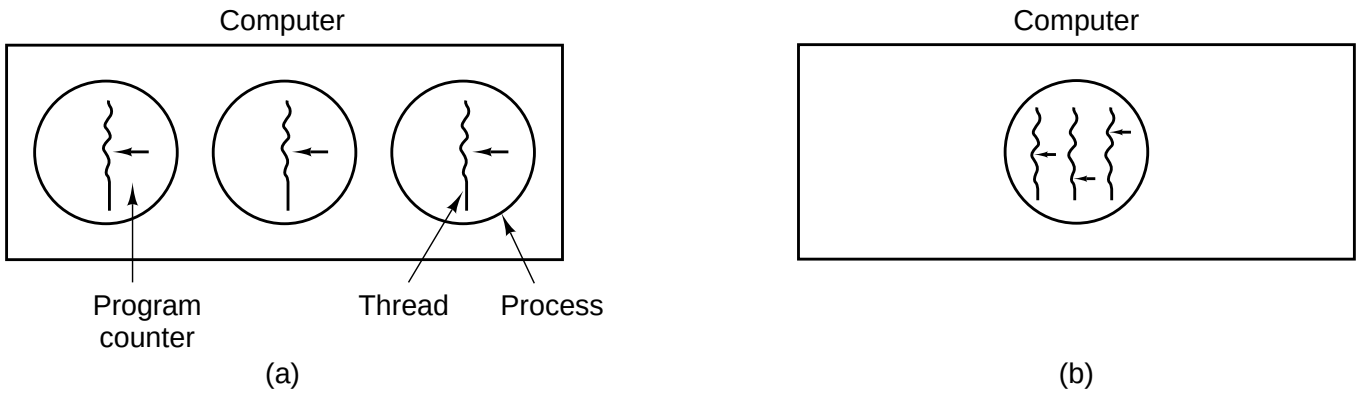
- Las cuatro transiciones posibles se muestran en la siguiente figura:



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Usando el modelo de procesos es mucho más fácil pensar en lo que hace el sistema
- En el nivel más bajo del sistema operativo se encuentra el **despachador**, con todos los procesos sobre de él.
- El despachador oculta todo el manejo de interrupciones y los detalles sobre cómo iniciar y parar procesos.
- Las rutinas del despachador tienen que escribirse en ensamblador: salvar los registros y cambiar el apuntador a la pila no pueden expresarse en C.

Hilos



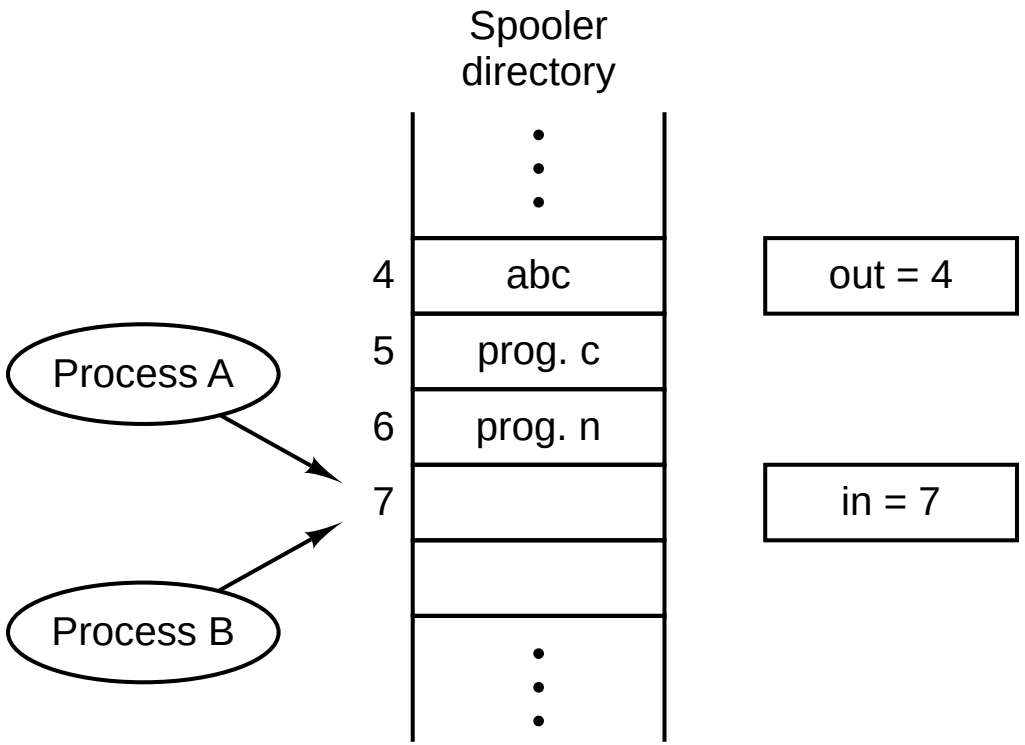
- POSIX threads (p-threads) pueden funcionan en el espacio del usuario
- El cambio entre hilos es más rápido en el espacio de usuario
- Pero cuando un hilo en el espacio del usuario se bloquea (esperado una E/S o el manejo de una falla de página) el proceso completo también se bloquea.
- Como consecuencia ambos sistemas se usan, tanto en el espacio de usuario como en el núcleo.

Comunicación interprocesos

1. Condiciones de carrera
2. Secciones críticas
3. Exclusión mutua con esperas ocupadas
4. Dormir y despertar
5. Semáforos
6. Monitores
7. Paso de mensajes

1. Condiciones de carrera

Se le llama así cuando dos o más procesos están leyendo o escribiendo a algún dato compartido y el resultado final depende de quién ejecuta (corre) algo en que preciso momento.



```
print_file( filename )
{
   next_free_slot = in;
                              <- Aquí se interrumpe A debido
                              <- a una interrupción de reloj

   write_file_name_in_position( filename, next_free_slot );

   next_free_slot = next_free_slot + 1;

   in = next_free_slot;
}
```

Resultado: B nunca se imprime!

2. Secciones críticas

- Una **sección crítica** es la parte del programa donde se accesa a la memoria compartida o a cualquier recurso compartido.
- Lo que se necesita para evitar las condiciones de carrera es la **exclusión mutua**: los procesos no deben de leer y escribir en las secciones críticas al mismo tiempo.

- Esto no es una suficiente para evitar una condición de carrera.
- Se necesitan cuatro condiciones para tener una buena solución:
  1. Dos procesos no deben entrar simultáneamente en sus regiones críticas
  2. No deben realizarse suposiciones sobre la velocidad o número de CPUs
  3. Ningún proceso fuera de su sección crítica puede bloquear a otro proceso.
  4. Ningún proceso debería esperar para siempre para entrar a su región crítica.

3. Exclusión mutua con esperas ocupadas

- Una **espera ocupada** es cuando se prueba continuamente una variable hasta que aparece algún valor.
- Debe evitarse porque consume gran cantidad de tiempo de CPU.

3.1 Deshabilitación de las interrupciones

```
disable_interrupts();
critical_section();
enable_interrupts();
```

- Esto es posible para un programa ejecutándose en el modo del núcleo pero es muy mala idea para programas en el espacio de usuario.
- Un solo programa de usuario que deshabilita y vuelve a habilidad las interrupciones puede tirar a todo el sistema.

3.2 Variables con candados

```
lock0 = 0; // A global variable

process_A( )
{
  if( lock0 == 0 ) {
  <I>
    lock0 = 1;
    critical_section( );
    lock0 = 0;
  }
}
```

```
process_B( )
{
  if( lock0 == 0 ) {

    lock0 = 1;
    critical_section( );
    lock0 = 0;
  }
}
```

Los dos procesos pueden entrar al mismo tiempo en sus secciones críticas si se interrumpe A en el punto <I>

3.3 Alternación estricta

```
int turn = 0;
```

```
while (TRUE){
  /* loop */
  while(turn != 0);
  critical_region();
  turn  = 1;
  noncritical_region();
}
```

```
while (TRUE) {
  /* loop */
  while(turn != 1);
  critical_region();
  turn  = 0;
  noncritical_region();
}
```

Esta solución viola el punto 3.

3.4 Solución de Peterson

```
#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 */
}
```

```
process_0( )
{
    enter_region( 0 );
    critical_section( );
    leave_region( 0 );

    do_some_work( );
}
```

```
process_1( )
{
    enter_region( 1 );
    critical_section( );
    leave_region( 1 );

    do_some_work( );
}
```

```
/* Variables globales! */
int turn;
int interested[2];

enter_region(0)                    enter_region(1)
  other = 1                          other = 0
  interested[0] = TRUE               interested[1] = TRUE
  turn = 0                           turn = 1
  <I> turn = 1 !!!!!                 while (turn==1 &&
  while (turn==0 &&                         interested[0]==TRUE);
         interested[1]==TRUE);
```

El proceso 0 entra en su sección crítica. El proceso 1 entrará en su sección critica una vez que el proceso 0 ejecute `leave_region(0)`, cuando haga `interested[0] = FALSE` y saldrá del ciclo de espera `while`.

3.5 La solución TSL

- La instrucción: `TSL RX,LOCK`
- (Test and Set Lock) funciona así: lee el contenido de memoria apuntado por `LOCK` en el registro `RX` y almacena un valor distinto de cero en el localidad de memoria `LOCK`.
- Las operaciones para leer el contenido de la memoria y almacenarla se garantizan indivisibles por el hardware
- Se pone un candado en el bus de la memoria para prohibir que otro CPU accese la memoria hasta que la instrucción termine.

```
enter_region:
    TSL REGISTER,LOCK   |copy LOCK to register and set LOCK to 1
    CMP REGISTER,#0     |was LOCK zero?
    JNE ENTER_REGION    |if it was non zero, LOCK was set, so loop
    RET                 |return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        |store a 0 in LOCK
    RET                 |return to caller
```

Soluciones correctas:

- Tanto la solución de Peterson como la que usa la instrucción TSL son correctas
- Pero son malas soluciones porque usan la espera ocupada,
- que consume tiempo de CPU
- También pueden causar que un proceso nunca entre a su sección crítica si el otro proceso es más pesado.

4. Dormir y despertar

- El problema de productores y consumidores

```
#define N 100  /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */

void producer(void)
{
    int item;
    while (TRUE){                  /* repeat forever */
        item = produce_item();     /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item);         /* put item in buffer */
        count = count + 1;         /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE){                  /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        item = remove_item();      /* take item out of buffer */
        count = count - 1;         /* decrement count of items in buffer */
        if (count == N-1 ) wakeup(producer); /* was buffer full? */
        consume_item(item);      /* print item */
    }
}
```

```
#define N 100  /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */

void producer(void)
{
    int item;
    while (TRUE){                  /* repeat forever */
        item = produce_item();     /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item);         /* put item in buffer */
        count = count + 1;         /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE){                  /* repeat forever */
        if (count == 0) <I> sleep(); /* if buffer is empty, got to sleep */
        item = remove_item();      /* take item out of buffer */
        count = count - 1;         /* decrement count of items in buffer */
        if (count == N-1 ) wakeup(producer); /* was buffer full? */
        consume_item(item);      /* print item */
    }
}
```

5. Semáforos

- Resolviendo el problema de productores y consumidores con semáforos
- Los semáforos los propuso Dijkstra en 1965.
- Las primitivas que propuso son `up` y `down` (generalizaciones de despertar y dormir)

- La operación `down` sobre un semáforo checa si su valor es más grande que 0.
- Si es así, decrementa su valor y continúa.
- Si su valor es 0, el proceso se pone a dormir sin completar el down por el momento.
- Checar el valor del semáforo, cambiarlo y posiblemente enviar el proceso a dormir debe realizarse en una operación indivisible.
- Esta atomicidad es indispensable para prevenir alguna condición de carrera.

- La operación `up` incrementa el valor del semáforo.
- Si uno a más procesos estaban durmiendo por ese semáforo, sin poder completar una operación `down` anterior, uno de ellos se escoge por el sistema (tal vez aleatoriamente) y se le permite completar su `down`.
- Entonces, después de un `up` sobre un semáforo con procesos durmiendo por él, el semáforo seguirá valiendo 0, pero habrá un proceso menos durmiendo por él.
- Esta operación también debe ser indivisible.

```
#define N 100          /* number of slots in the buffer */
typedef int semaphore; /* semaphores are a special kind of int */
semaphore empty = N;   /* counts empty buffer slots */
semaphore full = 0;    /* counts full buffer slots */

void producer(void)
{
  int item;
  while (TRUE){
    item = produce_item();  /* generate something to put in buffer */
    down(&empty);           /* decrement empty count */
    insert_item(item);      /* put new item in buffer */
    up(&full);              /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;
  while (TRUE){
    down(&full);            /* decrement full count */
    item = remove_item();   /* take item from buffer */
    up(&empty);             /* increment count of empty slots */
    consume_item(item);     /* do something with the item  */
  }
}
```

```
#define N 100          /* number of slots in the buffer */
typedef int semaphore; /* semaphores are a special kind of int */
semaphore mutex = 1;   /* controls access to critical region */
semaphore empty = N;   /* counts empty buffer slots */
semaphore full = 0;    /* counts full buffer slots */

void producer(void)
{
  int item;
  while (TRUE){
    item = produce_item();  /* generate something to put in buffer */
    down(&empty);           /* decrement empty count */
    down(&mutex);           /* enter critical region */
    insert_item(item);      /* put new item in buffer */
    up(&mutex);             /* leave critical region */
    up(&full);              /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;
  while (TRUE){
    down(&full);            /* decrement full count */
    down(&mutex);           /* enter critical region */
    item = remove_item();   /* take item from buffer */
    up(&mutex);             /* leave critical region */
    up(&empty);             /* increment count of empty slots */
    consume_item(item);     /* do something with the item  */
  }
}
```

- Un uso de los semáforos es garantizar la exclusión mutua
- El otro uso es para sincronización
- Los semáforos full y empty se necesitan para garantizar que cierta secuencia de procesos ocurra, o no ocurra.
- En este caso, se garantiza que el productor pare de correr cuando el buffer está lleno,
- y el consumidor pare de correr cuando el buffer este vacío.

Mutexes

- Un **mutex** es una versión simplificada de un semáforo
- Mutex viene del inglés “mutal exclusion”
- Se usan para manejar la exclusión mutua sobre un recurso compartido o sobre una pieza de código.

6. Monitores

- Fueron propuestos por Brinch Hansen (1973) and Hoare (1974)

```
monitor example
  integer i;
  condition c;

  procedure producer (x);
  .
  .
  .
  end;

  procedure consumer (x);
  .
  .
  .
  end;
end monitor;
```

- Solo las funciones dentro del monitor pueden acceder a las variables del monitor (un concepto de programación con objetos)
- Solo un proceso dentro del monitor puede estar activo en cualquier instante.
- Los monitores son un constructor en un lenguaje de programación, de forma que el compilador sabe como manejar los procedimientos dentro del monitor.
- Si se llama un procedimiento del monitor, se debe de checar si existe otro procedimiento ejecutándose y si es así se suspende la llamada.
- Si no hay otro procedimiento ejecutándose, entonces puede ejecutarse.
- Para realizar la exclusión mutua se usan un mutex o un semáforo binario.
- El compilador es el que ordena la exclusión mutua.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

   count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

7. Paso de mensajes

- Los semáforos y monitores están pensando para proteger regiones críticas en computadoras con memoria compartida.
- Si se tienen varias computadoras estas construcciones no sirven.
- Se resuelve el problema con estas dos primitiva que serían llamadas al sistema:

```
send(destination, &message);
receive(destination, &message);
```

- El proceso que recibe, si no hay un mensaje disponible, se bloquea.
- O podría regresar inmediatamente y generar un error.
- Para garantizar que los mensajes no se pierde, se debe generar un mensaje de *reconocimiento*.
- Si el proceso que enviar no recibe el reconocimiento, entonces retransmite el mensaje.
- Si se pierde el mensaje de reconocimiento el mensaje se enviará dos veces. Se resuelve con un contador de mensajes.
- También existen problemas de autenticación.

```
#define N 100      /* number of slots in the buffer */

void producer(void)
{
  int item;
  message m;                    /* message buffer */
  while (TRUE) {
    item = produce_item();      /* generate something to put in buffer */
    receive(consumer, &m);      /* wait for an empty to arrive */
    build_message(&m, item);    /* construct a message to send */
    send(consumer, &m);         /* send item to consumer */
  }
}

void consumer(void)
{
  int item, i;
  message m;
  for (i = 0; i < N; i++) send(producer, &m);   /* send N empties */
  while (TRUE) {
    receive(producer, &m);      /* get message containing item */
    item = extract_item(&m);    /* extract item from message */
    send(producer, &m);         /* send back empty reply */
    consume_item(item);         /* do something with the item */
  }
}
```

- Muchas variantes son posibles usando paso de mensajes.
- Una posibilidad es usar una dirección única a cada proceso y enviar los mensajes a esa dirección única.
- Otra en usar la estructura de datos **buzón**.
- Un buzón es un buffer para poner cierto número de mensajes.
- En las llamadas a send y receive se usan las direcciones a los buzones, no a procesos
- Cuando un proceso intenta enviar un mensaje a un buzón lleno, este se suspende hasta que se remueva algún mensaje y se haga espacio para uno nuevo.

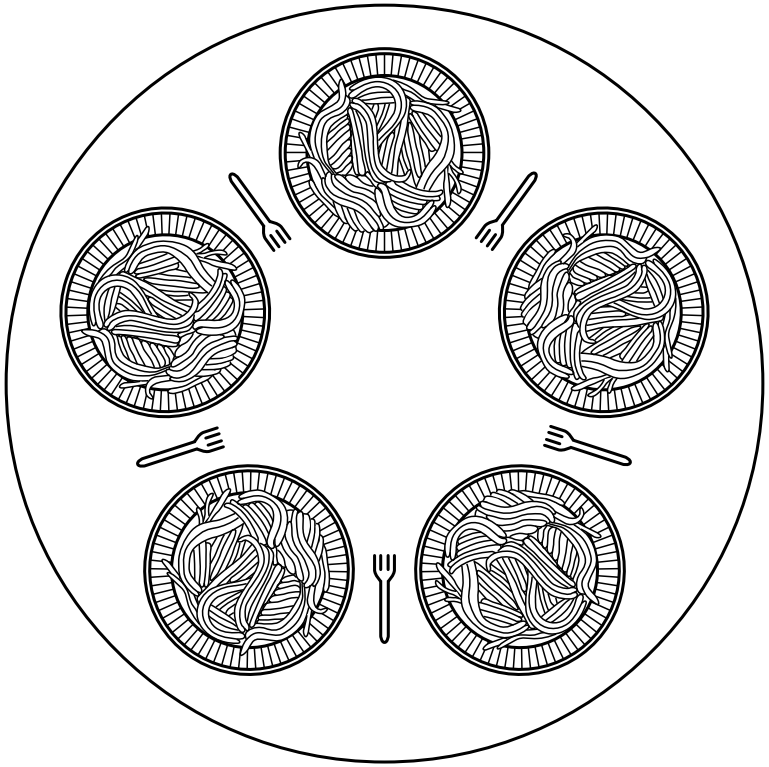
- Para el problema de productores y consumidores se podrían usar buzones.
- De tamaño para mantener N mensajes.
- El productor podría enviar mensajes con datos al buzón del consumidor y
- el consumidor podría enviar mensajes vacíos al buzón del productor.
- El mecanismo del buffer es claro: se envían mensajes al destinatario que aún no han sido aceptados.
- También se podría remover los buffers: el envío se hace antes de la recepción,
- el que envía se bloquea hasta que se recibo suceda y así se copia directamente el mensaje.
- Si se hace primero el recibo, este se bloquea hasta que el envío suceda.
- Esto se conoce como **encuentro** (rendezvous).

Problemas clásicos de comunicación interprocesos

1. Problemas de los filósofos cenando
2. Lectores y escritores

Los filósofos cenando

- Fue propuesto por Dijkstra en 1965
- Cinco filósofos están sentados alrededor de una mesa circular.
- Cada filósofo tiene enfrente un plato de espagueti.
- El espagati está tan resbaladizo que se necesitan dos tenedores para comerlo.



- La vida de un filósofo consiste en periódos alternativos de comer y pensar.
- Cuando un filósofo tiene hambre, intenta adquirir sus dos tenedores, el de la izquierda y el de la derecha, uno a la vez, en cualquier orden.
- Si los obtiene, se sienta a comer por un rato, cuando termina regresa los tenedores y continúa pensando.

```
#define N 5      /* number of philosophers */

void philosopher( int i )   /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();              /* philosopher is thinking */
        take_fork( i );       /* take left fork */
        take_fork( (i+1)%N ); /* take right fork; % is modulo operator */
        eat();                /* yum-yum, spaghetti */
        put_fork( i );        /* put left fork back on the table */
        put_fork( (i+1)%N );  /* put right fork back on the table */
    }
}
```

Una no solución al problema.

- Se puede modificar el programa de tal manera que:
- Se toma el tenedor a la izquierda,
- se checa si el tenedor a la derecha está disponible
- Si no está disponible se regresa el tenedor de la izquierda, espera por un rato y después se repite todo el proceso.
- En esta versión se puede llegar a la **inanición**

- Se puede tener una solución sin bloqueos de muerte ni inaniciones así:
- Se usa un semáforo binario justo después de pensar.
- Entonces cada filófoso hace un down sobre un mutex.
- Después de regresar los tenedores, se hace un up sobre el mutex.
- En esta solución solo un filósofo puede comer en cualquier instante.

```
#define N         5
#define LEFT      (i+N-1)%N
#define RIGHT     (i+1)%N
#define THINKING  0
#define HUNGRY    1
#define EATING    2

typedef int semaphore; /* semaphores are a special kind of int */
int state[N];          /* array to keep track of everyone's state */
semaphore mutex = 1;   /* mutual exclusion for critical */
semaphore s[N];        /* one semaphore per philosopher */

void philosopher(int i)   /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE){
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

```
void take_forks(int i)  /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);       /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i);            /* try to acquire 2 forks */
    up(&mutex);         /* exit critical region */

    down(&s[i]);        /* block if forks were not acquired */
}

void put_forks(int i)         /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);         /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);           /* see if left neighbor can now eat */
    test(RIGHT);          /* see if right neighbor can now eat */

    up(&mutex);           /* exit critical region */
}
```

```
void test(int i)   /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

El problema de lectores y escritores

- El problema de los filósofos cenando es útil para modelar procesos que están compitiendo para tener acceso exclusivo a un número limitado de recursos, tal como a dispositivos de E/S.
- El problema de lectores y escritores modela el acceso a una base de datos.

```
typedef int semaphore;   /* use your imagination */
semaphore mutex = 1;     /* controls access to 'rc' */
semaphore db = 1;        /* controls access to the database */
int rc = 0;              /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE){                /* repeat forever */
        down(&mutex);            /* get exclusive access to 'rc' */
        rc = rc + 1;             /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);              /* release exclusive access to 'rc' */
        read_data_base();        /* access the data */

        down(&mutex);            /* get exclusive access to 'rc' */
        rc = rc - 1;             /* one reader fewer now */
        if (rc == 0) up(&db);    /* if this is the last reader ... */
        up(&mutex);              /* release exclusive access to 'rc' */
        use_data_read();         /* noncritical region */
    }
}
```

```
void writer(void)
{
    while (TRUE){              /* repeat forever */
        think_up_data();       /* noncritical region */
        down(&db);             /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db);               /* release exclusive access */
    }
}
```