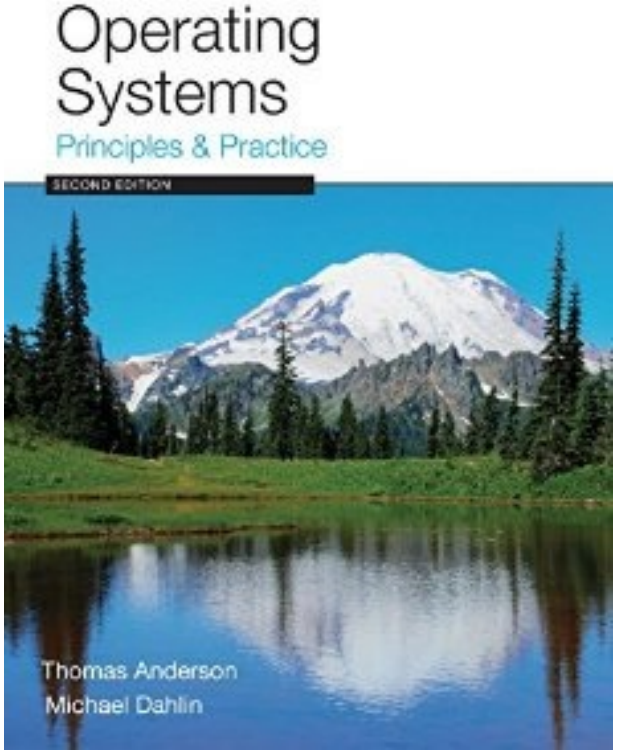
Sistemas Operativos

Capítulo 3 La Interfaz de Programación

Prof. Javier Cañas R.

Estos apuntes están tomados en parte del texto: “Operating System: Principles and Practice” de T. Anderson y M. Dahin



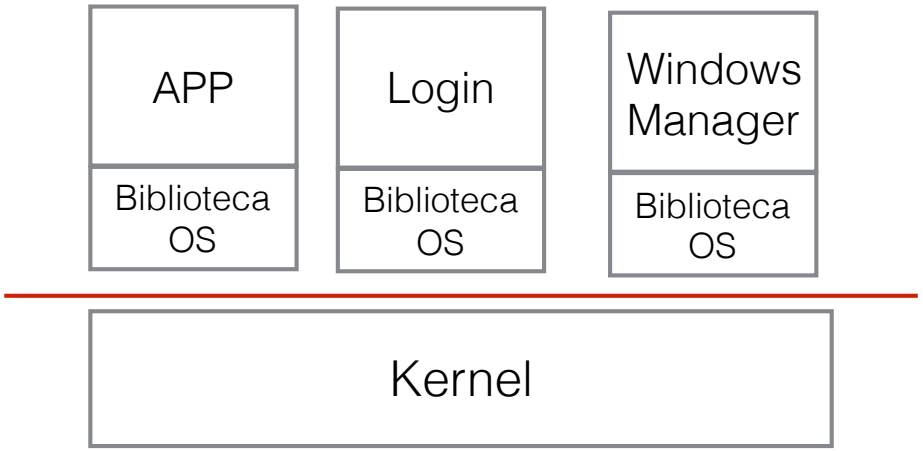
Introducción

- ¿**Qué** funciones necesita el SO proporcionar a las aplicaciones?
  - **Manejo de procesos**
  - **Entrada y Salida**
  - Manejo de Threads
  - Manejo de memoria
  - Sistema de archivos y almacenamiento
  - Redes
  - Gráfica y manejo de ventanas
  - Autenticación y seguridad

… Introducción

- ¿**Dónde** están estas funcionalidades?

- Pueden estar en
  - Programas nivel usuarios
  - Bibliotecas nivel usuario
  - Kernel: se accesan por *llamadas al sistema*



Puntos Principales

- Creación y manejo de procesos
  - fork, exec y wait
- Entrada y Salida
  - open, read, write y close
- Comunicación entre procesos
  - pipe, dup, select, connect
- Ejemplo: implementación de un shell

Temario

1. Manejo de Procesos
2. Entrada y Salida
3. El Shell
4. Comunicación entre procesos
5. Estructura de SO

1 Manejo de Procesos

- ¿Por qué se necesita manejar procesos?. Una primera motivación fue crear shells.
- Un shell es una interfaz con el SO. Se conoce también como Job Control System
  - Permite a los programadores crear y manejar un conjunto de programas para realizar alguna tarea.
  - Window, MacOS y Linux tienen shells

… Procesos

- Ejemplo: compilar un programa C

```
$ gcc -c file1.c
$ gcc -c file2.c
$ gcc -o output file1.o file2.o
```

- gcc -c file.c genera código objeto en file.o

- gcc -o output file1.o file2.o liga los archivos objetos en el ejecutable output

- Este conjunto de comandos se puede poner en un archivo y el shell lee línea por línea creando procesos para compilar y ligar.

Creación de Procesos

- Un compilador crea una imagen ejecutable de un programa.
- Para correr un programa ejecutable, es necesario asignarle recursos. Esta acción se realiza por llamadas al sistema.

La Process Control Block (PCB)

La PCB la almacena el kernel

Creación de procesos en Windows

- Se hace a través de una llamada al sistema que permite crear y correr un programa. Las acciones son:
  - Crear e inicializar la PCB en el kernel.
  - Crear e inicializar un nuevo espacio de direcciones.
  - Cargar el programa en este nuevo espacio de direcciones.
  - Copiar los argumentos del programa dentro de la memoria en el espacio de direcciones.
  - Inicializar el contexto del hardware para comenzar la ejecución en el punto de inicio.
  - Informar al Scheduler que el nuevo proceso está listo para correr.

Función de creación en Windows

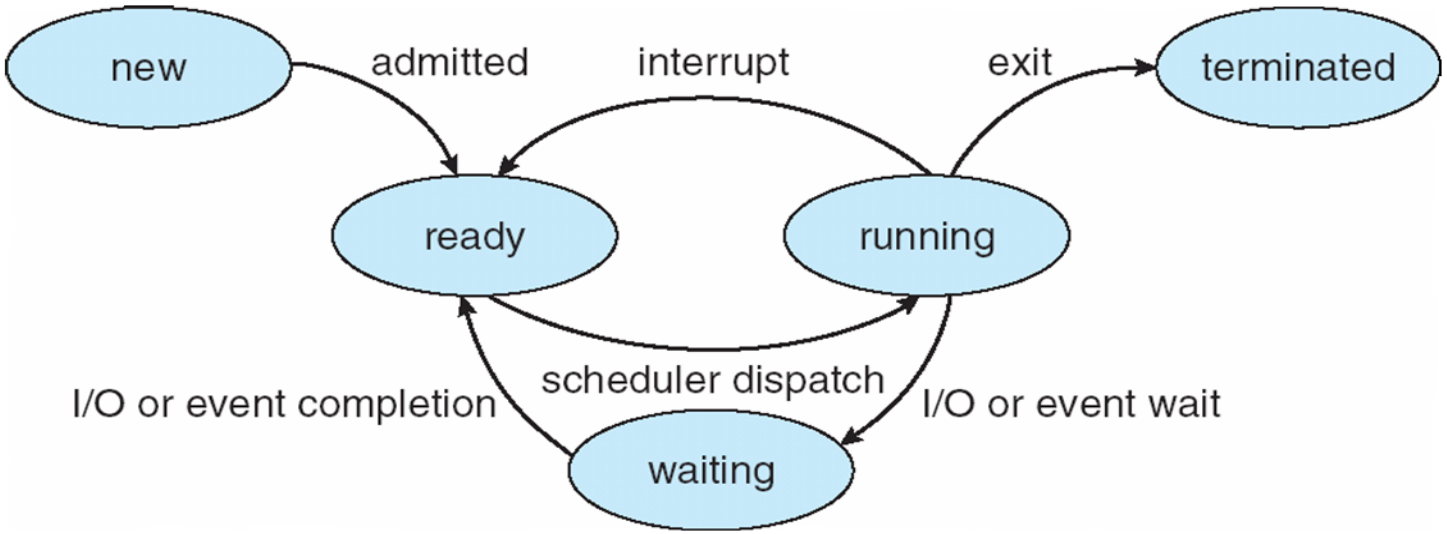
```
if (!CreateProcess(
    NULL,      // No module name (use command line)
    argv[1],   // Command line
    NULL,      // Process handle not inheritable
    NULL,      // Thread handle not inheritable
    FALSE,    // Set handle inheritance to FALSE
    0,         // No creation flags
    NULL,    // Use parent's environment block
    NULL,   // Use parent's starting directory
    &si,     // Pointer to STARTUPINFO structure
    &pi )    // Pointer to PROCESS_INFORMATION structure
)
```

El proceso que crea se denomina ***proceso padre***
El proceso creado se denomina ***proceso hijo***

Manejo de procesos en UNIX

- Se utilizan las siguientes llamadas al sistema:
  - **fork**: crea una copia del proceso que invoca y comienza su ejecución. No tiene argumentos.
  - **exec**: cambia el programa que está corriendo por el proceso que se quiere activar. Cargar el programa en este nuevo espacio de direcciones.
  - **wait**: espera a que termine un proceso.
  - **signal**: notifica a otro proceso de algún evento.

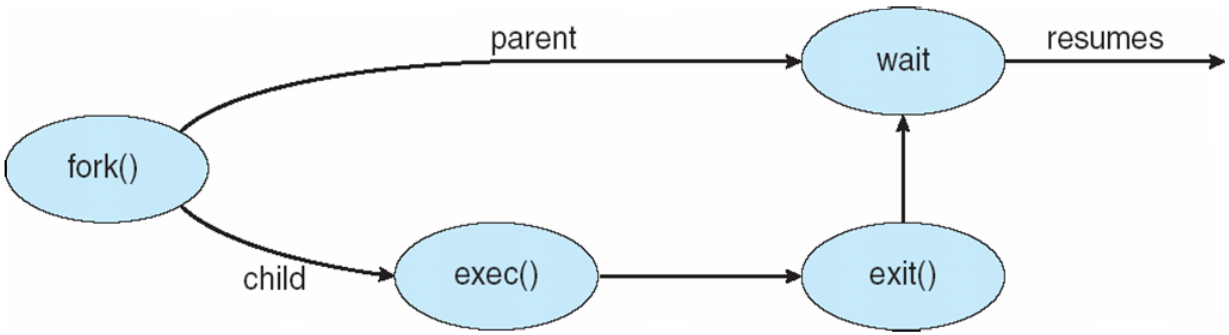
Diagrama de estado de proceso/thread



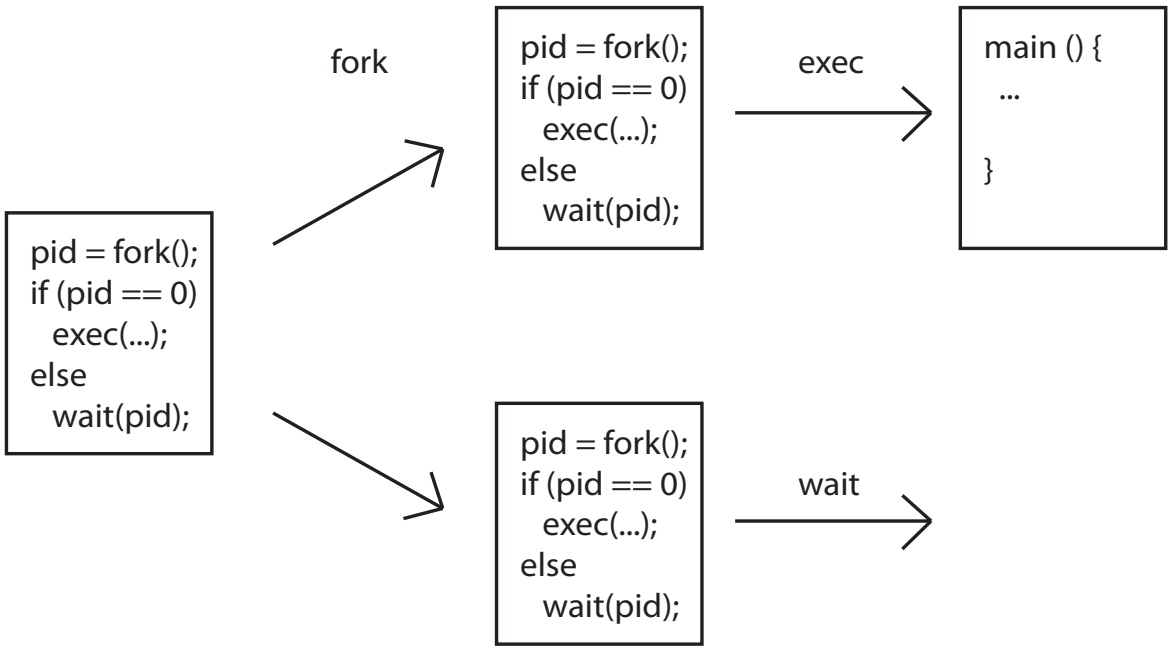
fork()

Retorna:

1. Un número negativo cuando la creación no ha sido exitosa.
2. Cero al proceso hijo.
3. Un valor positivo (el pid del hijo) al proceso padre.



fork()


pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);
fork
pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);
exec
main () {
...
}
pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);
wait

¿Qué se imprime?

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

int main()
{
    int child_pid = fork();
    if (child_pid == 0) {    // I'm the child process
        printf("I am process #%d\n", getpid());
        return 0;
    }
    else {                  // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
}
```

¿Qué se imprime?

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

int main()
{
    int child_pid = fork();
    if (child_pid == 0) {    // I'm the child process
        printf("I am process #%d\n", getpid());
        return 0;
    }
    else {                   // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
}
```

```
I am parent of process #36992
I am process #36992
```

UNIX fork

- Pasos del kernel para la implementación del fork:
  1. Crear e inicializar la PCB en el kernel.
  2. Crear un nuevo espacio de direcciones.
  3. Inicializar el espacio de direcciones con una copia completa del espacio de direcciones del proceso padre.
  4. Heredar el contexto de ejecución del proceso padre (por ejemplo, archivos abiertos).
  5. Informar al Scheduler que un nuevo proceso está listo para ejecución

UNIX exec

- Pasos del kernel para la implementación de exec:
  1. Cargar el programa en el espacio de direcciones actual.
  2. Copiar argumentos en memoria dentro del espacio de direcciones.
  3. Inicializar el contexto del hardware para comenzar la ejecución en el punto de inicio.

UNIX wait y UNIX signal

- Wait suspende al padre hasta que el hijo termina, se cae (crash) o es terminado externamente. Como un padre puede haber creado muchos hijos, su parámetro es el pid del proceso hijo.

- Signal (kill) se utiliza para enviar a otro proceso una notificación o "*upcall*". Se utiliza para terminar una aplicación, suspenderla temporalmente para debuging y reasumir después de una suspensión.

2 Entrada y Salida en UNIX

- Los sistemas computacionales aceptan una diversidad de dispositivos de E/S: teclados, mouse, discos, USB, WiFi, GPS, etc…

- Antiguamente se creaban aplicaciones especializadas para cada dispositivo. Uno de los grandes aportes de UNIX fue estandarizar la E/S de diversos dispositivos detrás de una misma interfaz.

- UNIX va más allá y utiliza la misma interfaz para leer y escribir archivos, y comunicar procesos.

Ideas básicas de E/S en UNIX

- Uniformidad
  - Toda operación, en cualquier archivo utiliza el mismo conjunto de llamadas al sistema: open, close, read y write.
- Abrir antes de usar
  - Open retorna un “file descriptor” para ser utilizado en futuras llamadas sobre el archivo.
- Toda estructura es orientada al Byte
- El kernel maneja un buffer para read/write
- Cerrar en forma explícita. Así se recuperan los descriptores.

File Descriptor

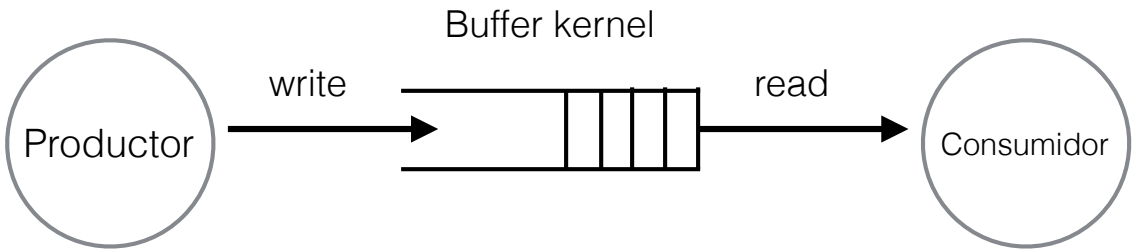
```
int fd; //file descriptor

fd=open(“nombre archivo”, permisos);

read(fd, buffer , numbytes);
```

o para escribir:

```
write(fd, buffer , numbytes);
```

Pipes

- Un pipe en UNIX es un buffer del kernel con dos descriptores: escritura y lectura.
- El pipe termina cuando cualquiera de los extremos cierra el pipe o hace un exit().



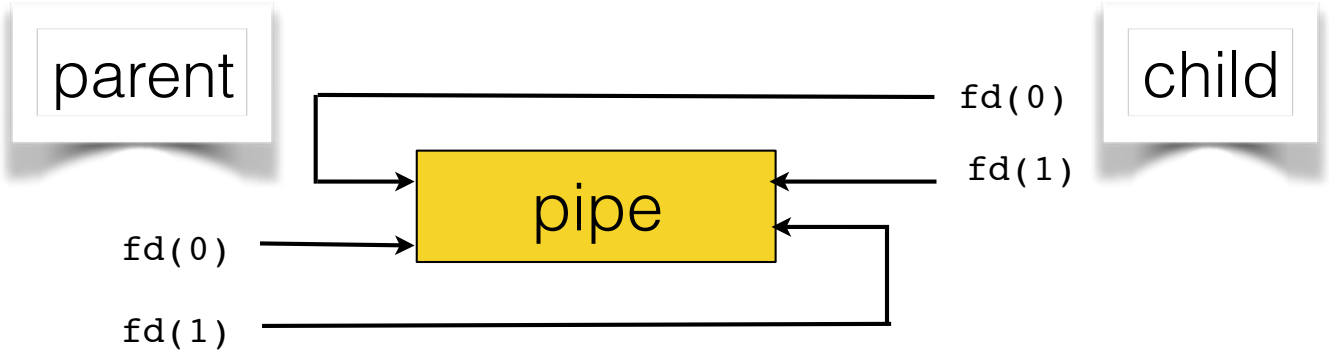
Descriptores de archivos para pipes

- En UNIX se crea un pipe usando:

```
pipe(int fd[]);
```

- Esta función crea un pipe que puede ser accesado usando el descriptor:
  - **`fd[0]`** para leer el pipe
  - **`fd[1]`** para escribir en el pipe
  - Se utilizan las llamadas al sistema **`read()`** y **`write()`**

Descriptores



Ejemplo

```c
/* IPC Ordinary pipes in UNIX*/
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFF_SIZE 25
#define READ 0
#define WRITE 1

int main(void)
{
    char write_msg[BUFF_SIZE] = "Viva Chile\n";
    char read_msg[BUFF_SIZE];
    int fd[2];
    pid_t pid;
    /*crear el pipe*/
    pipe(fd);

    /* se crea un hijo */
    pid = fork();
    if (pid > 0) {
        close(fd[READ]); /*no se usa*/
        write(fd[WRITE], write_msg, strlen(write_msg)+1);
        close(fd[WRITE]); /* Ya no se usa*/
    }
    else { /*proceso hijo*/
        close(fd[WRITE]); /*no se usa*/
        read(fd[READ], read_msg, BUFF_SIZE);
        printf("El mensaje dice: %s", read_msg);
        close(fd[READ]);
    }
    return 0;
}
```

Resumen llamadas al sistema

Creación y manejo de procesos

fork()	Crea proceso hijo como un clon del padre
exec(prg, args)	Corre el programa prg en el actual proceso
exit()	informa al kernel que el proceso terminó
wait(pid)	Pausa hasta que el hijo termina
kill(pid, tipo)	envía una interrupción tipo al proceso pid

Resumen llamadas al sistema

Operaciones de E/S

fileDesc open (name)	Abre archivo, retorna descriptor
pipe(fd[2])	Crea pipe unidireccional
dup2 (desde fd, a fd)	reemplaza descriptores con una copia
int read(fd, buffer; size)	Lee hasta size bytes en buffer desde fd
int write(fd, buffer; size)	escribe hasta size bytes en buffer a fd
close(fd)	El proceso con archivo fd terminó

3 El Shell

- Un shell lee una línea de comando desde la entrada y hace un fork para crear el proceso que ejecuta el comando.
- Al hacer un fork, UNIX automáticamente duplica todos los archivos abiertos en el padre.
- El padre espera que el hijo termine antes de leer el siguiente comando.

Estructura de un Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

4 IPC: Inter Process Communication

- Aplicaciones complejas se construyen a partir de módulos simples.
- Resulta muy conveniente crear aplicaciones que se especialicen en tareas específicas. Combinándolas se construyen estructuras más complejas.
- Por ejemplo UNIX utiliza un servidor de impresión especializado en manejar colas de documentos a imprimir.
- Por todo esto, es necesario que los procesos puedan comunicarse unos con otros. Esto se denomina IPC

Estructuras de IPC

- **Productor-consumidor**: programas aceptan como entradas las salidas de otros programas (ej. pipes). La comunicación es SDX (simplex o one way).

- **Cliente-servidor**: Comunicación FDX (full duplex o two way). El servidor implementa un servicio y los clientes lo solicitan mediante solicitudes.

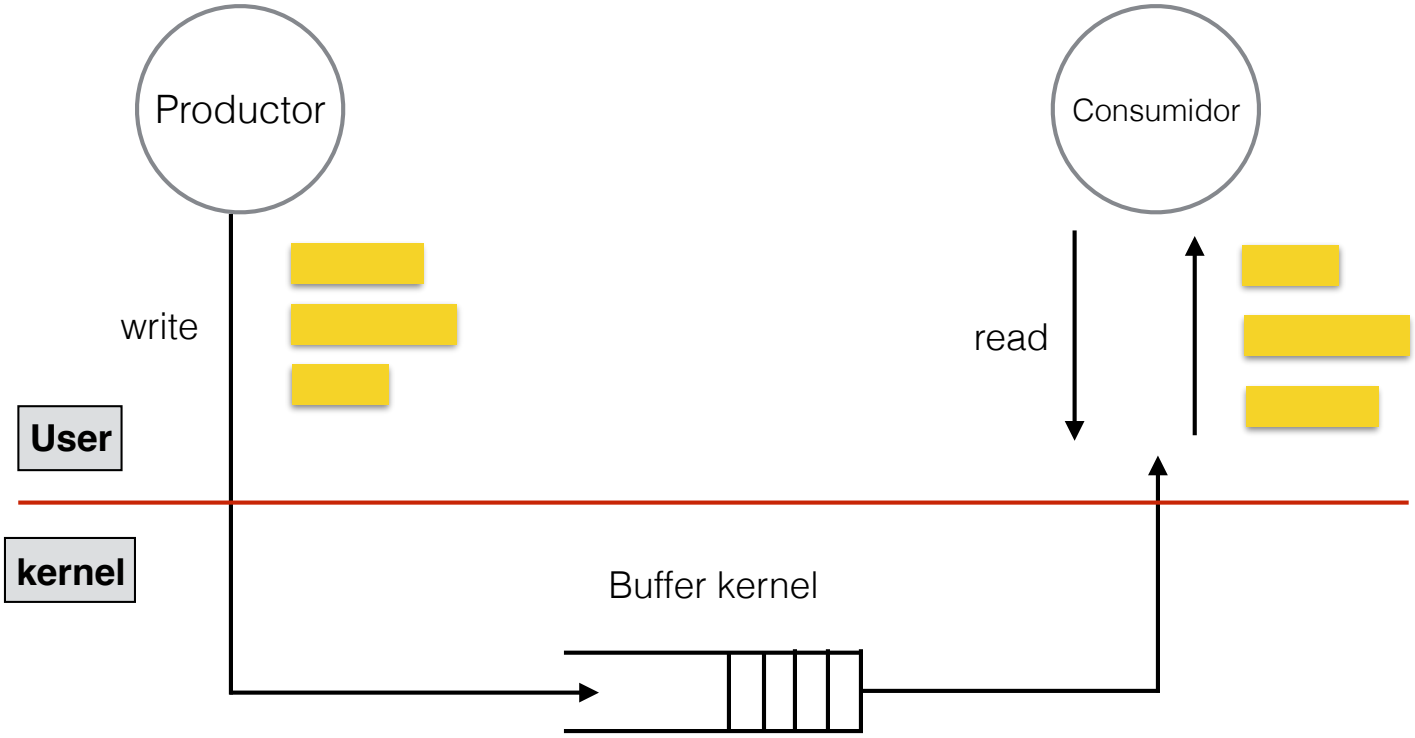
- **Sistema de archivos**: los programas se conectan escribiendo y leyendo desde archivos. A diferencia de los anteriores, esta comunicación puede ser diferida en tiempo.

Ejemplos de IPC

- Google MapReduce opera sobre la red como productor-consumidor. Las salidas de la función *map* es enviada a máquinas que corren la función *reduce*.

- La web es un ejemplo de cliente-servidor.

Productor Consumidor



En un sistema multicore, Productor y Consumidor pueden funcionar al mismo tiempo

Ejemplo de interacción Cliente-Servidor

```
Cliente:
    char request[RequestSize];
    char reply[ReplySize];

    // diversos calculos
    // poner requerimiento en buffer
    // enviar buffer al server
    write(output, request, RequestSize);
    // Espera respuesta
    read(input, reply, RelySize);
    // cálculo

Server:
    char request[RequestSize];
    char reply[ReplySize];
    //loop esperando requerimiento
    while(1) {
       // leer comandos que llegan
       read(input, request, RequestSize);

       //hacer la operacion

       //enviar resultados
       write(output, reply, ReplytSize);
    }
```

Servidor con múltiples clientes

```
Server:
    char request[RequestSize];
    char reply[ReplySize];
    FileDescriptor clientInput[NumClients];
    FileDescriptor clientOutput[NumClients];
    //loop esperando requerimiento desde cualquier cliente
    while(fd==select(clientInput,NumClients)) {
       // leer comandos que llegan desde clientes especificos
       read(ClientInput[fd], request, RequestSize);

       //hacer la operacion

       //enviar resultados
       write(clientOutput[fd], reply, ReplytSize);
    }
```

5 Estructura de un SO

- Un SO es un sistema de SW grande y complejo. Su ingeniería debe ser cuidadosa para que funcione correctamente y pueda ser modificado con facilidad.

- Básicamente, los SO de propósito general obedecen a dos estructuras:
  - **Kernel monolíticos**
  - **Micro kernels**

Dependencias

- Independiente de su estructura, los SO se organizan con módulos. Existen muchas dependencias entre los módulos internos. Las más relevantes son:
  - Muchas partes dependen de primitivas de sincronización para acceso a estructuras del kernel.
  - La memoria virtual depende del soporte de HW para conversión de direcciones.
  - El sistema de archivos y memoria virtual comparten un pool común de bloques de memoria física y dependen del driver del disco.
  - El sistema de archivos depende de protocolos de red para acceso a discos en máquinas distintas.

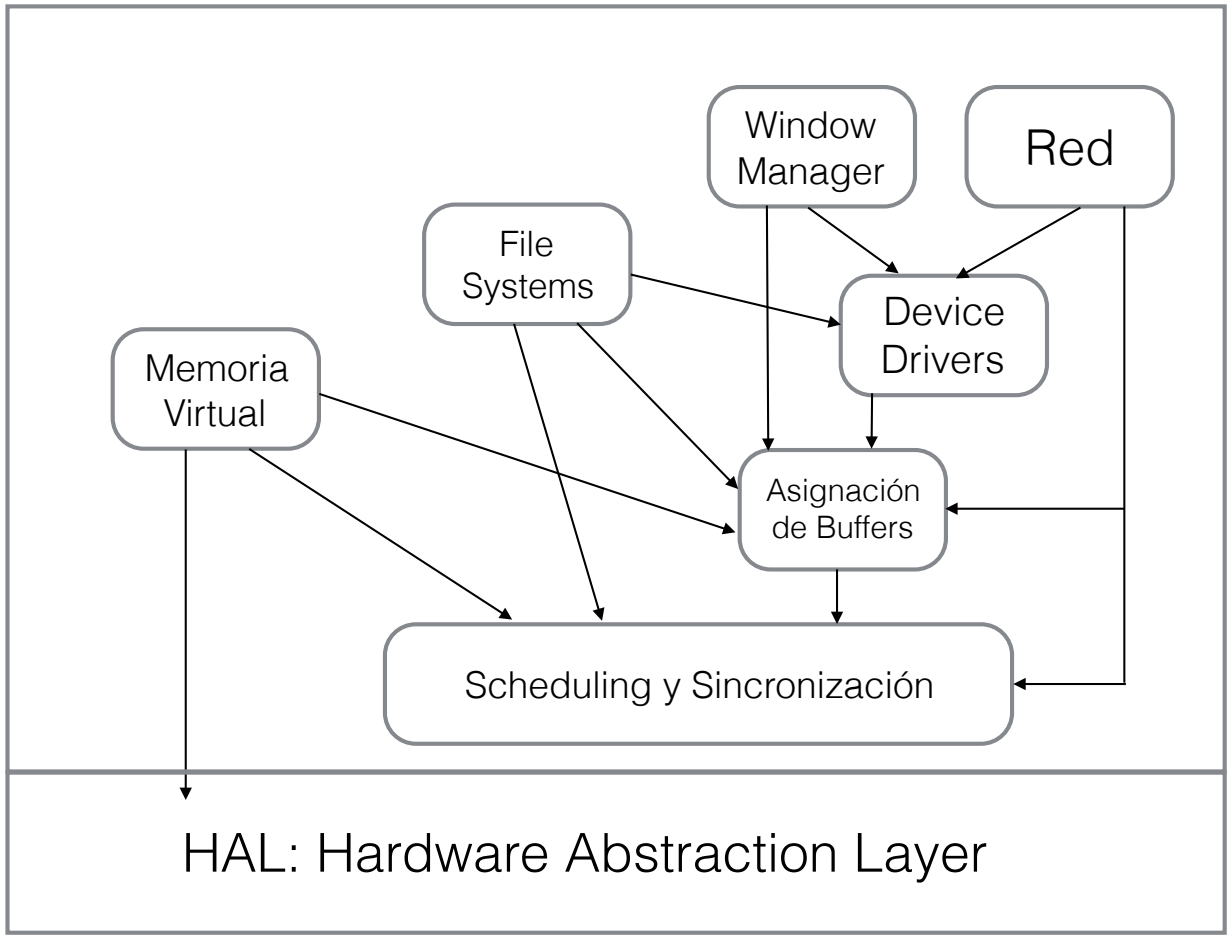
Tradeoff

- Las dependencias llevan a los diseñadores a enfrentar diversas decisiones de compromiso.
- Al centralizar funciones del kernel, se mejora el desempeño y facilita una estrecha integración de los distintos módulos pero quita flexibilidad, los cambios se vuelven difíciles y se adaptan menos a diversos requerimientos

Kernels Monolíticos

- La mayoría de los actuales SO tienen esta concepción: Windows, MacOSX y Linux.
- La mayoría de sus funcionalidades corren dentro del kernel.
- Sin embargo en estos sistemas muchas de sus partes están fuera del kernel: shells, bibliotecas de sistema, gráfica, …

Kernel Monolítico



Portabilidad de kernels monolíticos

- La portabilidad de kernels monolíticos involucran dos aspectos:
  - Capa de Abstracción de Hardware (HAL)
  - Instalación dinámica de drivers de dispositivos

HAL

- Un objetivo de diseño es la portabilidad del SO sobre distintas plataformas de HW.
- La HAL es una interfaz para configuraciones y operaciones específicas de procesadores. Por ejemplo en x86 distintos fabricantes de computadores requieren códigos específicos diferentes para el manejo de interrupciones y HW de timers.
- Con una capa HAL bien definida, los SO son independientes del procesador. Portar un SO significa modificar rutinas de bajo nivel de HAL y re-ligar módulos

Instalación dinámica de drivers

- Un SO necesita instalar una gran variedad de dispositivos de E/S como por ejemplo nuevas impresoras.

- Existe una gran diversidad de interfaces de HW para dispositivos. Un 70% del código del kernel de Linux es específico de dispositivos.

- Los drivers de carga dinámica permiten el manejo de dispositivos. Los fabricantes proporcionan el código utilizando una interfaz estándar soportada por el kernel.

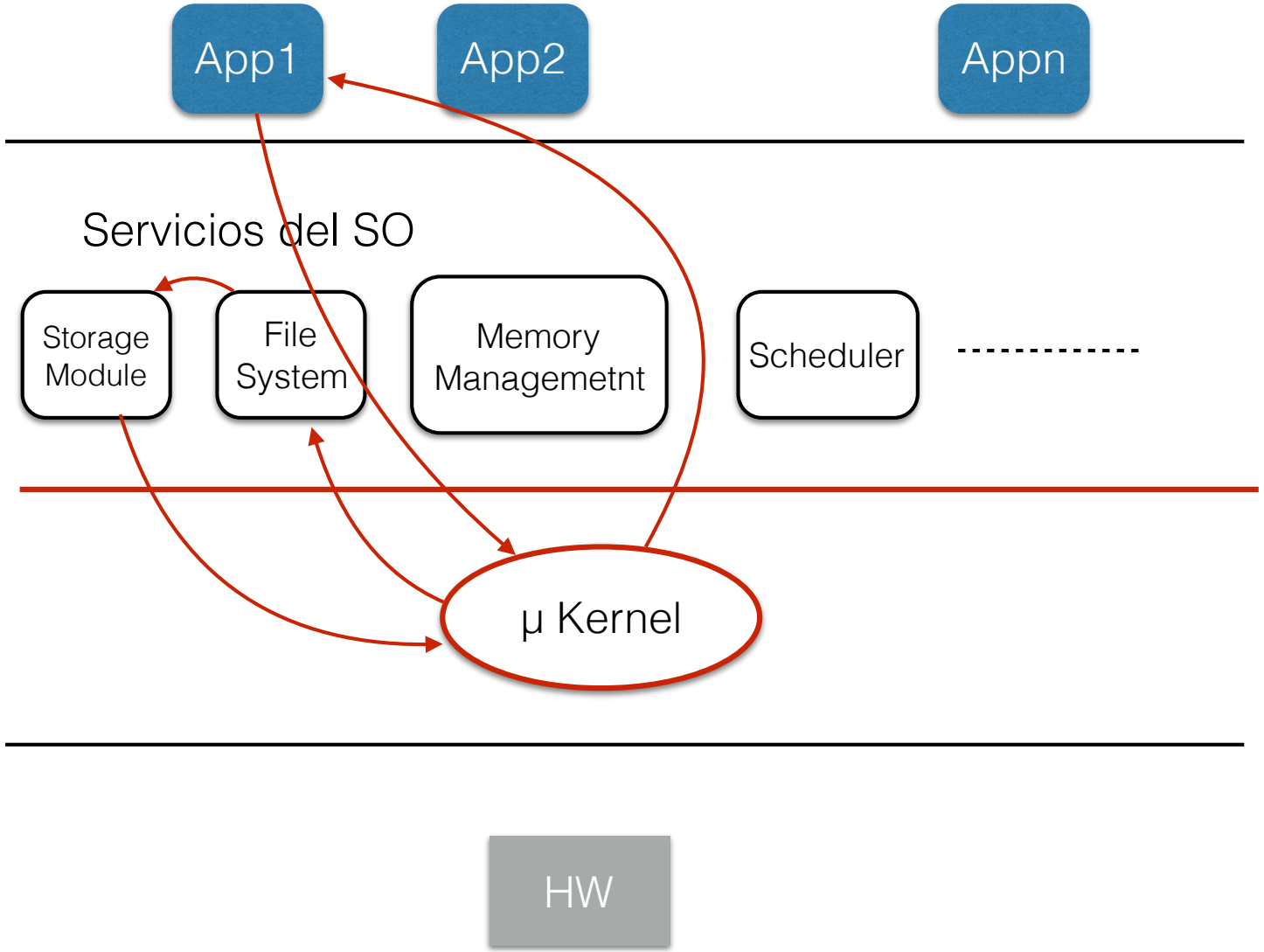
… Instalación dinámica de drivers

- Cuando el SO se inicia, pregunta al bus de E/S por los dispositivos conectados y carga los drivers correspondientes desde archivos en disco.
- Si los dispositivos están en red, por ejemplo una impresora, el SO carga los drivers desde Internet.

Microkernel

- La idea es mover todo lo posible desde el kernel al espacio de usuario.
- La comunicación se desarrolla entre los módulos usuarios usando mecanismos de comunicación (IPC).
- La función principal es proporcionar facilidades de comunicación entre el programa cliente y un conjunto de servicios que también corren en espacio usuario.

Microkernel



… Microkernel

- Pros:
  - Fácil de extender
  - Fácil de portar a nuevas arquitecturas
  - Más confiable
  - Más seguro
- Contra:
  - Overhead producto de comunicación entre espacios de kernel y usuario

Sistemas Operativos

Capítulo 3 La Interfaz de Programación

Prof. Javier Cañas R.