SO: Notas Segundo Parcial

Galileo Cappella

2c 2022

1 Inter Process Communication

1.1 Signals

- `int kill(pid_t pid, int sig)`
- `void* signal(int signum, void *handler)`
- `pid_t fork(void)`
- `int execv(const char *pathname, char *const argv[])`
- `pid_t wait(int *wstatus)`
- `pid_t waitpid(pid_t pid, int *wstatus, int options)`
- `pid_t waitpid(pid_t pid, int *wstatus, int options)`
- `uint sleep(uint seconds)`

1.2 Pipes

- `int pipe(int fildes[2])`
- `int close(int fd)`
- `ssize_t write(int fd, const void *buf, size_t count)`
- `ssize_t read(int fd, const void *buf, size_t count)`

1.3 Sockets

- **Server** `socket()` → `bind()` → `listen()` → `accept()`
- **Client** `socket()` → `skip` → `skip` → `connect()`
- `int socket(int domain, int type, int protocol)`, crea un nuevo socket.
- `int bind(int fd, sockaddr* a, socklen_t len)`, asigna una dirección (nombre o IP y puerto) al socket.
- `int listen(int fd, int backlog)`, setea al socket como pasivo que recibirá conexiones. Se maneja una cola para poder recibir varias conexiones.
- `int accept(int fd, sockaddr* a, socklen_t* len)`, extrae de la cola una solicitud de conexión y establece la comunicación entre sockets. Se bloquea en caso de no existir conexioens pendientes. Devuelve un nuevo fd para conexión.

- `int connect(int fd, sockaddr* a, socklen_t* len)`, se conecta a un socket remoto que debe estar escuchando.
- `ssize_t send(int s, void *buf, size_t len, int flags)`
- `ssize_t recv(int s, void *buf, size_t len, int flags)`

1.4 Ejemplo

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>
#include <time.h>
#include <sys/wait.h>
int pipes[N][2], pidHijos[N], hijo;
void crearPipes() {
  for (int i = 0; i < N; ++i) pipe(pipes[i]);
}
void cerrarPipesParaPadre() {
  for (int i = 0; i < N; ++i) close(pipes[i][1]);
}
void cerrarPipesParaHijo() {
  for (int i = 0; i < N; ++i)
    if (i != hijo) {
      close(pipes[i][0]);
      close(pipes[i][1]);
    }
  close(pipes[hijo][0]);
}
void handler() {
  printf("Gane!\n");
  exit(EXIT_SUCCESS);
}
void ejecutarHijo() {
  signal(SIGTERM, handler);
  srand(getpid());
  cerrarPipesParaHijo();
  while (1) {
    int numero = rand() % 100;
    write(pipes[hijo][1], &numero, sizeof(numero));
    sleep(1);
  }
}
void crearHijos() {
  for (hijo = 0; hijo < N; hijo++) {
    pidHijos[hijo] = fork();
    if (pidHijos[hijo] == 0)
      ejecutarHijo();
  }
}
void jugarConHijos() {
  int numeroSecreto = rand() % 50, hijosVivos = N, i = 0;
  while (hijosVivos > 1) {
    if (pidHijos[i] != 0) {
      int numero;
      read(pipes[i][0], &numero, sizeof(numero));
      if(numero > numeroSecreto) {
        hijosVivos--;
        kill(pidHijos[i], SIGKILL);
        waitpid(pidHijos[i], NULL, 0);
        pidHijos[i] = 0;
```

```
55        }
56      }
57      i = (i+1) % N;
58    }
59    while(pidHijos[i] == 0) {
60      i = (i+1) % N;
61    }
62    kill(pidHijos[i], SIGTERM);
63  }
64  int main(void) {
65    srand(time(NULL));
66    crearPipes();
67    crearHijos();
68    cerrarPipesParaPadre();
69    jugarConHijos();
70    return 0;
71  }
```

2 Scheduling

2.1 Métodos

- **First In, First Out (FIFO)** o **First Come, First Served (FCFS)**.
- **Shortest Job First (SJF)**.
- **Shortest Remaining Time First (SRTF)** = SJF + tiempos de llegada y desalojo.
- **Priority Scheduling (PS)**.
- **Round Robin (RR)**, single/multiple queue.
- **Multilevel Queue Scheduling (MQS)**.
- **Multilevel Feedback Queue Scheduling (MFQS)**, los procesos van llegando a la cola más baja, si no termina en el tiempo dado se mueve a la siguiente cola. Se corre en órden, $\text{queue}_1 \rightarrow \text{queue}_2 \rightarrow \ldots \rightarrow \text{queue}_n \rightarrow \text{queue}_1$.
- **Earliest Deadline First (EDF)** para *Real Time (RT)* systems.
- **Linux CFS**, usa un árbol rojo-negro ordenado por *vruntime*, y prioriza la de menor valor.

2.2 Misc.

- **Diagrama de Gantt** muestra el tiempo que ocupan los procesos.
- **CPU burst (ráfaga)**, tiempo de CPU que necesita un proceso.
- El scheduler toma decisiones cuando:
  1. Un proceso cambia de estado running a waiting.
  2. Un proceso cambia de estado running a ready.
  3. Un proceso cambia de estado waiting a ready.
  4. Un proceso termina.

  Sólo considerar 1 y 4 lo hace **nonpreemtive**, y las 4 es **preemptive**.
- El **Dispatcher** da el control al proceso seleccionado luego de:
  1. Context switch.
  2. Cambiar a modo usuario.
  3. Saltar a la ubicación correcta en el programa.
- **Turnaround** = tiempo de finalización - tiempo de llegada.
- **Waiting Time** = turn around - CPU burst.
- Determinar la longitud del siguiente **CPU Burst**: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$ con t_n longitud real, τ_n longitud estimada, $0 \leq \alpha \leq 1$ usualmente $\frac{1}{2}$

2.3 Ejemplo

t	Proceso	ejecución (t)	Burst restante (t)
0-1ms	P4	1ms	2ms
1-2ms	P4	1ms	1ms
	P3	0ms	8ms
2-3ms	P4	1ms	0ms
	P3	0ms	8ms
	P1	0ms	6ms
3-4ms	P3	0ms	8ms
	P1	1ms	5ms
4-5ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	1ms	4ms
5-7ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	0ms	4ms
	P2	2ms	0ms
7-10ms	P3	0ms	8ms
	P1	0ms	5ms
	P5	4ms	0ms
10-15ms	P3	0ms	8ms
	P1	5ms	0ms
15-23ms	P3	8ms	0ms

Proceso	Llegada	CPU Burst	Finalización	Turnaround	Waiting
P1	2ms	6ms	15ms	15ms - 2ms = 13ms	15ms - 6ms = 7ms
P2	5ms	2ms	7ms	7ms - 5ms = 2ms	2ms - 2ms = 0ms
P3	1ms	8ms	23ms	23ms - 1ms = 22ms	22ms - 8ms = 14ms
P4	0ms	3ms	3ms	3ms - 0ms = 3ms	3ms - 3ms = 0ms
P5	4ms	4ms	10ms	10ms - 4ms = 6ms	6ms - 4ms = 2ms
Promedio	-	-	-	9.2ms	4.6ms

3 Sync

- **Race condition** es cuando el resultado depende del orden en que se ejecuten las cosas.
- **Sección crítica** es un cacho de código que sólo hay un proceso ejecutándolo a la vez, y cualquiera que quiera ejecutarlo espera a que se libere.
- Una operación **Atómica** es indivisible a nivel CPU.
- **TestAndSet (TAS)** es una instrucción atómica que cambia el valor de una variable y devuelve el valor que tenía. `while(testAndSet(x)) {}` hace **busy waiting** hasta poder usarla.
- Un **Semáforo** es una variable entera que sólo se la puede manipular con dos operaciones: `s.wait()` que espera hasta que se mande una signal, y `s.signal(n)` que despierta a n procesos esperando en wait. Nos ahorra del busy waiting.
- El **Mutex** es un tipo de semáforo binario.
- El **Spin lock (o TASLock)** hace busy waiting `(TASLock) mtx.lock()`, y luego libera con `mtx.unlock()`. Puede tener menos overhead que un semáforo.
- El **Local spinning (o TTASLock)** tiene menos overhead, hace busy waiting
```
lock() {
  while (true) {
    while(mtx.get()) {};
    if (!mtx.testAndSet()) return;
  }
}.
```
- **Registros Read-Modify-Write** atómicos: `int getAndInc()`, `int getAndAdd(int)`, `T compareAndSwap(T u, T v) { if (u == reg) reg = v }`
- **Cola atómica**:
```
enqueue(T item) { mtx.lock(); q.push(item); mtx.unlock(); }
bool dequeue(T *pitem) {   mtx.lock();
  bool success = !q.empty();
  if (success) pitem = q.pop();
  else pitem = null;
  mtx.unlock();
  return success;
}
```
- **Mutex recursivo** se salva de deadlock por recursión:
```
void create() { owner.set(-1); calls = 0; }
void lock() {
  if (owner.get() != self)
    while(owner.compareAndSwap(-1, self) != self) {}
  calls++;
}
void unlock() { if (- -calls == 0) owner.set(-1); }
```
- Las **Condiciones de Coffman** para un deadlock, todas son necesarias:

utual exclusion Un recurso está en un modo no-compartido.

Hold and wait Un proceso tiene al menos un recurso y está pidiendo más recursos que están siendo tenidos por otros procesos.

No preemtion Un recurso sólo puede ser liberado voluntariamente por el proceso que lo tiene.

Circular wait Hay un ciclo de esperas.

3.1 Ejemplo

```
//************************************************************
//S: Barrier
int n = 0, N;
semaphore mtx(1), step[2] = {0, 0};
void wait(void) {
  mtx.wait();
  if (++n == N) step[0].signal(N); //A: Dejo pasar a N
  mtx.signal();
  step[0].wait();
  mtx.wait();
  if (--n == 0) step[1].signal(N);
  mtx.signal();
  step[1].wait();
}
//************************************************************
//S: Programa
SHARED:
  semaphore permisoLobos(0), permisoCabras(0),
            mtx(1);
  barrier barrera(4);
LOCAL:
  bool remero = false;
void lobo() { //NOTA: Las cabras son similares
  mtx.wait();
  if (++lobos == 4) {
    lobos = 0;
    permisoLobos.signal(4);
    remero = true;
  } else if (lobos == 2 && cabras => 2) {
    lobos = 0;
    cabras -= 2;
    permisoLobos.signal(2);
    permisoCabras.signal(2);
    remero = true;
  } else mtx.signal();
  permisoLobos.wait();
  abordar();
  barrera.wait();
  if (remero) {
    remar(); remero = false;
    mtx.signal();
  }
}
```

4 Memory Management

4.1 Remoción

- **First In, First Out (FIFO)**
- **Least Recently Used (LRU)**
- **Segunda Oportunidad**, si fue referenciada le doy otra oportunidad
- **Not Recently Used**, primero desalojo las que no fueron referenciadas ni modificadas, luego las referenciadas, y luego las modificadas.

4.2 Allocation

- **First fit**
- **Best fit**, agarro el bloque más chico posible.
- **Worst fit**, agarro el bloque más grande posible.
- **Quick fit**, se usan listas de bloques de tamaño fijo.

5 Drivers

- Tres tipos de interración:
  - **Polling:** El driver se fija si el dispositivo se comunicó.
    Pros: Sencillo, cambio de contexto controlado.
    Cons: Consume CPU.
  - **Interrupciones (push):** El dispositivo avisa.
    Pros: Eventos asincrónicos poco frecuentes.
    Cons: Cambios de contexto impredecibles.
  - **Direct Memory Access (DMA):** Para transmitir grandes volúmenes (la CPU no interviene). Requiere un componente de HW, el controlador de DMA. Cuando finaliza interrumpe al CPU.
- Subsistema de I/O, API bajo nivel: `open, close, read, write, seek`. API de alto nivel: `fopen, fclose, fread, fwrite, fgetc, fputc, fgets, fputs, fscanf, fprintf`.
- **Char device**, se transmite byte a byte secuencialmente. No soportan acceso aleatorio y no usan cache.
- **Block device**, se transmite la información en bloque. Acceso aleatorio y buffer (cache).
- Además de las políticas normales de scheduling, se usa: **Shortest Seek Time First (SSTF)** que toma el pedido que menos mueve a la cabeza.
- Procedimientos de un driver:
  - *int driver_init()*, invocada durante la cargada del OS
  - *int driver_open()*, invocada al solicitarse un open
  - *int driver_close()*, invocada al solicitarse un close
  - *int driver_read(int *data)*, invocada al solicitarse un read
  - *int driver_write(int *data)*, invocada al solicitarse un write. **USAR** *copy_from_user()* y *copy_to_user()*.
  - *int driver_remove()*, invocada durante la descarga del OS
- Syscalls que puede acceder un driver:
  - *void OUT(int IO_address, int data)* escribe data en el registro de I/O
  - *int IN(int IO_address)* devuelve el valor almacenado en el registro de I/O
  - *int request_irq(int irq, void *handler)* permite asociar el procedimiento handler a la interrupción IRQ. Devuelve IRQ_ERROR si ya está asociada a otro handler.
  - *int free_irq(int irq, void *handler)* libera la interrupción IRQ del procedimiento asociado.
- Cuidarse de las race conditions.

5.1 Ejemplo

```
//S: Variables del programa
static char input_mem[3][100], buffer_lectura[3][1000];
static atomic_int buffer_start[3], buffer_end[3], procesos_activos[3];
static mutex mtx;
static void driver_handler(void) {
  int reg_data = IN(KEYB_REG_DATA);
  int keycode = reg_data & 0x3FFF; //A: Primeros 14 bits
```

```
  int id = (reg_data >> 14) & 0x3; //A: Bits 14 y 15
  bool res;
  if (id == 0) res = write_to_all_buffers(keycode2ascii(keycode));
  else res = write_to_buffer(id, keycode2ascii(keycode));
  OUT(KEYB_REG_CONTROL, res ? READ_OK : READ_FAILED);
}
static int driver_open(void) {
  int id = -1;
  mtx.lock();
  for (int i = 0; i < 3; i++)
    if (!procesos_activos[i]) {
      procesos_activos[i] = true;
      OUT(KEYB_REG_STATUS, i);
      OUT(KEYB_REG_AUX, APP_UP);
      id = i;
      break;
    }
  mtx.unlock();
  return id;
}
static void driver_close(int id) {
  mtx.lock();
  procesos_activos[id] = false;
  buffer_start[id] = buffer_end[id];
  OUT(KEYB_REG_STATUS, id);
  OUT(KEYB_REG_AUX, APP_DOWN);
  mtx.unlock();
}
static int driver_read(int id, char *buffer, int length) {
  char *buf = kmalloc(length);
  while (get_buffer_length(id) < length); //TODO: Semaphore
  copy_from_buffer(id, buf, length);
  copy_to_user(buf, buffer, length);
  kfree(buf);
  return length;
}
static int driver_write(char *input, int size, int proceso) {
  int _size = min(size, 100);
  copy_from_user(input_mem[proceso], input, _size);
  return _size;
}
//S: Datos del driver
static struct file_operations driver_operaciones = {
  .owner = THIS_MODULE,
  .read = driver_read,
  .write = driver_write,
  .open = driver_open,
  .close = driver_close
};
static struct class *driver_class;
static struct cdev cdev;
static dev_t num = 0;
static int minor = 0;
static int count = 1;
static int __init driver_init(void) {
  printk(KERN_INFO "[driver_init]\n");
  //A: Obtener major y minor, crear device, registrarlo...
  cdev_init(&cdev, &driver_operaciones);
  int code = alloc_chrdev_region(&num, minor, count, "driver");
  if (code != 0) { printk(KERN_ALERT "\t* alloc_chrdev_region code=%i\n", code);
   return 1; }
  printk(KERN_INFO "\t* alloc_chrdev_region num=%i\n", num);
  code = cdev_add(&cdev, num, count);
  if (code != 0) { printk(KERN_ALERT "\t* cdev_add code=%i\n", code); return 1; }
```

```
71   driver_class = class_create(THIS_MODULE , "driver");
72   device_create(driver_class, NULL, num, NULL , "driver");
73   //A: Iniciar variables
74   for (int i = 0; i < 3; i++) mem_map(input_mem[i], INPUT_MEM[i], 100);
75   request_irq(IRQ_KEYB, driver_handler);
76   //TODO: Errors
77   printk(KERN_INFO "\t* SUCCESS\n");
78   return 0;
79 }
80 static void __exit driver_exit(void) {
81   printk(KERN_INFO "[driver_exit]\n");
82   //A: Liberar estructuras del kernel, remover device
83   device_destroy(driver_class, num);
84   class_destroy(driver_class);
85   unregister_chrdev_region(num, count);
86   cdev_del(&cdev);
87   //A: Liberar variables
88   for (int i = 0; i < 3; i++) mem_unmap(input_mem[i], INPUT_MEM[i], 100);
89   free_irq(IRQ_KEYB);
90 }
91 module_init(driver_init);
92 module_exit(driver_exit);
93 MODULE_LICENSE("GPL");
94 MODULE_AUTHOR("La banda de SO");
95 MODULE_DESCRIPTION("Una suerte de '/dev/null'");
```

6 File Systems

6.1 FAT

- Lista enlazada de bloques, armada con una tabla que para cada bloque dice el bloque que continúa al archivo
- Un directorio indica el índice del primer bloque de cada archivo, y los metadatos

6.2 ext2/3

- **fsck** chequea para cada bloque cuántos inodos le apuntan y cuántas veces aparece en la lista de bloques libres. Y se fija que tenga sentido.
- El fs tiene un bit que indique apagado normal.
- Se lleva un **journal**
  - Un buffer circular con un registro de los cambios que habría que hacer.
  - Si se llena, se baja el caché al disco, y se marca el buffer indicando qué cambios ya se reflejaron.
  - Impacta la performance, pero poco
  - Cuando el sistema se levanta se aplican cambios aún no aplicados.

6.3 Network File System (NFS)

- Utilizando **Remote Procedure Call (RPC)** permite acceder a FS remotos como si fueran locales.

6.4 Redundant Array of Inexpensive Disks (RAID)

Se copia en multiples discos para qe si se pierde alguno se tienen los otros.

Lvl	Descripción	Capacidad	Tolerancia a fallas	Aplicaciones típicas
0	Bandas, alterno los bloques entre cada uno de los discos	n discos	0	Data logging, caché
1	Espejo de 1 disco repetido n veces	1 disco	$n-1$ discos	Backups domésticos, DBs tradicionales
0+1	Bandas espejadas	$\frac{n}{2}$ discos	1 disco	Servers de archivos, servers de aplicaciones, DBs veloces
2	3 discos de paridad por cada 4 de datos	60%	$\frac{n}{2}$ discos	Sin uso comercial
3	1 discos de paridad por cada 4 de datos	1 disco	1 disco	Sin uso comercial
4	Paridad con intercalación de bloques	$n-1$ discos	1 disco	Sin uso comercial
5	Paridad con intercalación distribuida de bloques	$n-1$ discos	1 disco	Data warehouse, web servers
6	Paridad con intercalación doblemente distribuida de bloques	$n-2$ discos	2 disco	Backups comerciales

6.5 Ejemplo

	Contiguo	FAT			inodos		
	Disco	Disco	Lect	Esc	Disco	Lect	Esc
add_0	2561	1	0	1	1	1280	1281
$\text{add}_{\frac{n}{2}}$	1281	1	640	2	1	640	641
add_n	1	1	1279	2	1	0	1
del_0	2558	0	1	0	0	1279	1279
$\text{del}_{\frac{n}{2}}$	1278	0	641	1	0	639	639
del_n	0	0	1278	1	0	0	0

```
void printEntry(struct ext2_dir *entry) {
  struct ext2_inode *inode = load_inode(entry->inode);
  char *typeNPerm = tipos_y_permisos(inode->i_mode);
  printf("%u %s %u %u %u\t%u %u %s\n", entry->inode, typeNPerm, inode->
    i_links_count, inode->i_uid, inode->i_gid, inode->i_size, inode->i_mtime, entry
    ->name); //TODO: Alinear
  free(typeNPerm);
```

```
  //NOTA: Asumo que inode no tiene que ser liberado
}
void lsExt2(const char *path) {
  struct ext2_inode *inode = inode_for_path(path);
  if ((inode->i_mode & EXT2_TYPE_DIR) == 1) { //A: Directory
    char *buffer = malloc(BLOCK_SIZE);
    for (u32 i = 0, block; (block = get_block_address(i)) != 0; i++) { //A: Until
   the last block //NOTE: 0 is invalid
      read_block(block, buffer);
      size_t accum = 0;
      while (accum < BLOCK_SIZE) {
        printInode(entry);
        accum += entry->size;
        entry = (struct ext2_dir*)(((char*)entry) + entry->size);
      }
    }
    free(buffer);
  } else {} //TODO: Single file
}
```

7 Distributed

- Varias máquinas conectadas en red. Un procesador con varias memorias.
- **Pros**
  - Paralelismo
  - Replicación
  - Descentralización
- **Cons**
  - Dificultad para la sincronización
  - Dificultad para mantener coherencia
  - No suelen compartir clock
  - Información parcial
- Memoria compartida por Hardware: **Uniform Memory Access (UMA)**, **Non-Uniform Memory Access (NUMA)**
- **Telnet**, es un protocolo y un programa que permite que un equipo controle remotamente a otro.
- **RPC** es un mecanismo que permite hacer procedure calls de manera remota. Es un mecanismo *sincrónico*
- **Pasaje de mensajes**, *asincrónico*, **MPI** (una API). Problemas a considerar: Manejar codificación de los datos; Hay que dejar de procesar para atender mensajes; Lento; Se pueden perder mensajes; Hay un costo en $ por cada mensaje. Problemas ignorados en la materia: Los nodos pueden morir; La red se puede partir.
- No hay TestAndSet atómico. Soluciones:
  - Un nodo coordinador controla los recursos. Este nodo tiene proxies representantes de los procesos remotos. Cuando un proceso necesita un recurso se lo pide a su proxy.
    Cons: Punto único de falla; Cuello de botella;
  - **Orden parcial entre eventos**: Cada procesador tiene un reloj. Cada mensaje se marca con el tiempo t de envío. Cuando se recibe y t es mayor al valor actual del reloj se actualiza el reloj a $t + 1$.
    Los empates se ordenan arbitrariamente (p.e.: con el pid).
- **Acuerdo bizantino**, no hay algoritmo si hay fallas en la comunicación. Sí si pueden fallar a lo sumo $k < n$ procesos. Con $\mathcal{O}((k + 1)n^2)$ mensajes.
- **Cluster**, un conjunto de computadoras conectadas con un scheduler de trabajos en común. **Grid**, conjunto de clusters.
- **Scheduling**
  - Nivel **local**: Dar el procesador a un proceso listo.
  - Nivel **global** (mapping): Asignar un proceso a un procesador.
    Compartir la carga: De manera *estática (affinity)*, en el momento de la creación del proceso.
    O *dinámica (migration)*, variando durante la ejecución.
    Migración: sender initiated. Receiver initiated / Work Stealing.
  - Política:

    - Transferencia: Cuándo se migra
    - Selección: Qé proceso se migra
    - Ubicación: A dónde se envía el proceso
    - Información: Cómo se difunde el estado

- **Two Phase Commit (2PC)** *locking distribuído*, un nodo líder que no falla
  - **Phase Commit Request**:
    1. El líder le pregunta a todos los nodos si se puede realizar la operación, el mensaje firmado con un commit.
    2. Los nodos receptores ejecutan lo pedido, lockeando los recursos que tienen que utilizar. Pero no hacen commit, sino que guardan en una tabla cómo deshacer los cambios hechos.
    3. Cada nodo contesta "OK", o "ABORT" si hubo fallo.
  - **Phase Complete**:
    1. El líder envía "COMMIT"/"ROLLBACK" a todos los nodos
    2. Cada nodo completa/deshace la operación liberando los recursos.
    3. Los nodos devuelven COMMIT_OK
    4. EL lider da por finalizado cuando recibe "COMMIT" de todos los nodos

- **Three Phase Commit (3PC)** *locking distribuído*, un nodo líder que puede fallar.
  Si el líder falla en la primera o segunda fase: Se elije un nuevo líder y se puede abortar o reiniciar la ejecución. Si falla en la última fase: Se elije un nuevo líder y se le pregunta a los nodos en qué estado están, si un sólo nodo dice que está en "COMMIT_INITIALIZED" entonces el líder anterior había psado a la última fase, se continúa sin abortar.
  - **Phase Commit Request**:
    1. El líder le pregunta a todos los nodos si se puede realizar la operación, el mensaje firmado con un commit.
    2. Los nodos receptores ejecutan lo pedido, lockeando los recursos que tienen que utilizar. Pero no hacen commit, sino que guardan en una tabla cómo deshacer los cambios hechos.
    3. Cada nodo contesta "OK", o "ABORT" si hubo fallo.
  - Si el líder recibió algún "ABORT" manda "ROLLBACK" a todos, sino manda "PREPARE TO COMMIT". Los nodos responden si están preparados o no.
  - **Phase Complete**:
    1. El líder envía "COMMIT"/"ROLLBACK" a todos los nodos
    2. Cada nodo completa/deshace la operación liberando los recursos.
    3. Los nodos devuelven COMMIT_OK
    4. El lider da por finalizado cuando recibe "COMMIT" de todos los nodos

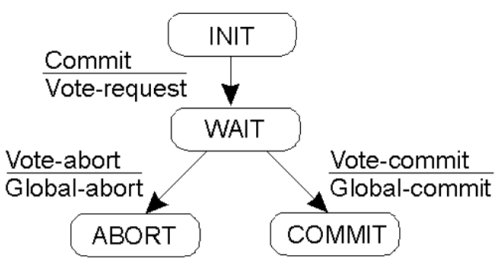
- Elegir líder:

- **LCR** (anillo), $\mathcal{O}(n)$
  Flood max (como el anillo, pero no es un anillo), $\mathcal{O}(n \cdot \text{diámetro})$.
  Bully Algorithm (grafo completamente conexo), cada nodo le manda a todos los nodos con ID mayor que quiere ser lider, si nadie le contesta se declara lider. Cada nodo que le llega repite el mismo esquema, pero también contestando al nodo que él se va a encargar de ser líder. Si nadie le contesta, se declara líder y broadcastea un mensaje declarándose victorioso. $\mathcal{O}(n^2)$

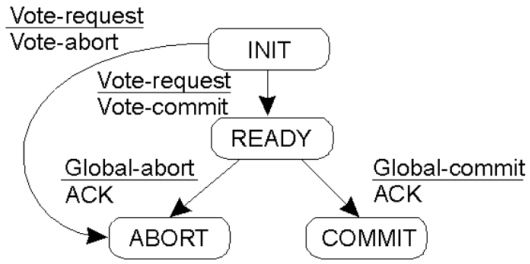
7.1 Distributed File System (DFS)

- El cliente lo ve como un FS normal.
- **Modelo cliente-servidor**:
  - El servidor almacena los archivos y su metadata en almacenamiento conectado al servidor.
  - Los clientes le piden archivos al servidor.
  - El servidor es responsble de la autenticación, el chequeo de permisos, y de devolver el archivo.
  - Los cambios que el cliente hace en archivos deben ser propagados al servidor.
  - Ejemplo: **NFS**
  - Unico punto de falla, si el sv se cae. Cuello de botella.
- **Modelo basado en cluster**:
  - Los clientes se conectan al sv de metadata, y hay varios svs de datos con chunks (porciones) de archivos.
  - El sv de metadata mantiene un mapeo de qué servidores tienen qué chunks de cada archivo.
  - Los chunks se replican n veces.
  - **Google FS (GFS)**. Inflenciado por las siguientes observaciones
    - La falla en componente de hardware es la norma.
    - Los archivos son muy grandes
    - Muchso archivos son modificados mediante appending.
    - Se puede rediseñar las aplicaciones y la API del FS para incrementar flexibilidad del sistema.
    - Se puede usar **MapReduce** sobre GFS para ejecutar computaciones paralelas a gran escala.
  - **Hadoop DFS (HDFS)**
  - Más resiliente a fallas y más escalable.
  - **Consistencia**: HDFS sólo permite operaciones de escritura de tipo append-only un sólo escritor por archivo. GFS permite escrituras arbitrarias y escritores concurrentes.
- **Mapeo multinivel**: oculta los detalles de cómo y dónde en el disco está almacenado el archivo.
- Un DFS **transparente** oculta la ubicación en la red.
- **Mapeo estático** independiente de ubiación.
- **Nombres**, enfoques:
  - Los archivos se nombran combinando el nombre del host con el nombre local. No es independiente ni transparente de ubicación.
  - Montar directorios remotos en locales. Dando la apariencia de un árbol de directorios coherentes. Sólo los directorios previamente montados son accesibles de manera transparente.
  - Única estructura global de nombre abarca a todos los archivos del sistema. Si un sv no está disponible, un conjunto arbitrario de directorios en distintas máquinas también está indisponible.

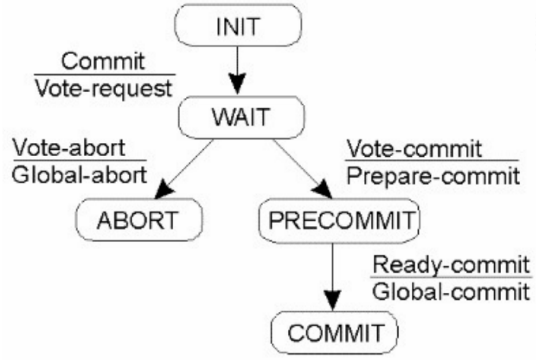
7.2 Ejemplo



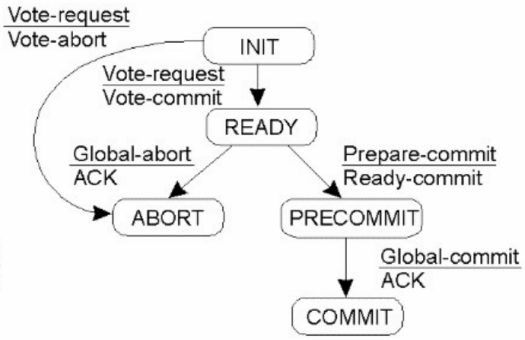
(a) Nodo líder



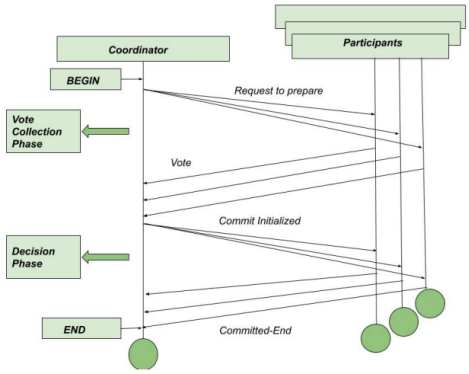
(b) Otros nodos



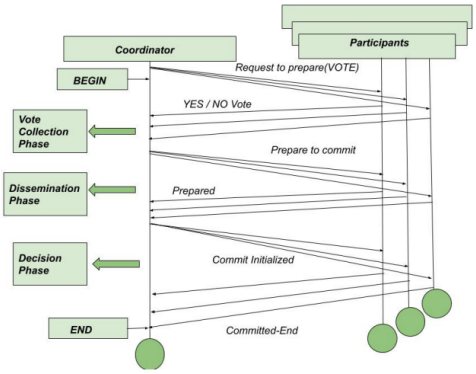
(a) Nodo líder



(b) Otros nodos



(a) 2 Phase Commit



(b) 3 Phase Commit

Asumo que

8 Security

- **Protección**: Los mecanismos para que nadie pueda tocar los datos de otro. Qué usuario puede hacer cada cosa.
- **Seguridad**: Se asegura que todes sean quien dicen ser. Impedir la adulteración de datos.
- Confidencialidad, Integridad, Disponibilidad.
- Sujetos, Objetos, Acciones. Se define qué Sujetos pueden realizar qué Acciones sobre qué Objetos.
- **A+1**, Authentication, Authorization, Accounting (registro).
- Criptografía *simétrica* una clave (Caesar, DES, Blowfish, AES), y *asimetrica* clave de encriptación, y otra de decriptación (RSA)
- **Hash**
  - Resistencia a la preimagen, dado h es difícil encontrar m tal que $h = hash(m)$
  - Resistencia a la segunda preimagen, dado m_1 es difícil encontrar m_2 tal que $m_1 \neq m_2 \wedge hash(m_1) = hash(m_2)$
  - En el Unix original los hashes se almacenan en `/etc/passwd`
  - **RSA**
    1. Tomamos p, q primos de (approx) 200 dígitos.
    2. $n = pq, n' = (p-1)(q-1)$
    3. Tomamos $e \in \mathbb{Z}/2 \leq n' - 1 \wedge e \perp n'$
    4. Computamos $d/d \cdot e \equiv 1 \ (mod\ n')$
    5. (e, n) clave pública, (d, n) clave privada.
    6. Encripto letra m calculando $r_n(m^e)$
    7. Desencripto letra encriptada c calculando $r_n(c^d)$
    8. La clave pública desencripta a la clave privada y viceversa
    9. Firma digital, calculo hash de un documento y firmo el cash con mi clave privada.
- **Discretionary Access Control**, cuando se crea un objecto se definen los permisos.
- **Mandatory Access Control (MAC)**, los objetos heredan el grado del último sujeto que los modificó, sólo se pueden acceder a objetos de grado menor o igual qe el tuyo.
- **SETUID** y **SETGID** permite correrlo con privilegios elevaos.
- Problemas comunes:
  - **Format String**
    Problema: Input de usuario no sanitizado.
    Impacto: Escalamiento de privilegios. Se puede ejecutar un shell de root.
    Solución: Validar input antes de ejecutarlo.
    ~~whitelist~~/allowlist: validar que tenga el formato requerido.
    ~~blacklist~~/blocklist: Sanitizar caracteres peligrosos o inválidos (ej: - ' " ; & , etc)
  - **Environment Variables**
    Problema: No se provee el path completo a la aplicación que se quiere ejecutar.
    Impacto: Escalamiento de privilegios. Se puede modificar el path y agregar un comando de igual nombre.
    Solución: Utilizar el path completo al llamar al programa.

- **Buffer Overflow**
  Problema: Se ingresa input de usuario directamente sobre un buffer de tamaño limitado.
  Impacto: Autenticación indebida. Se puede escribir input suficientemente largo para pisar valores. Esto genera escalamiento de privilegios.
  Solución: Limitar el tamaño del buffer de input.

- **Stack Overflow** Como el buffer, pero escribe sobre el stack.
- **Integer Overflow/Underflow**
- **Permisos**
  Impacto: Ejecución de código arbitrario.
  Solución: Principio del mínimo privilegio.

- **Denial of Service** (fork bomb)

- **SQL Injection** Similar al format string.