1)

a. Debe leer los datos de entrada por la entrada estándar (file descriptor 0) y escribir los resultados por salida estándar (file descriptor 1) ya que al componerse dos programas lo que se hace es enviar la salida estándar del proceso del primer programa a la entrada estándar del proceso del segundo programa.
Los errores deberían escribirse por error estándar (file descriptor 2) para no contaminar la salida estándar.

b. En primer lugar el SO provee syscalls para crear procesos y cargar programas (fork() y exec()), que sirven para correr los programas a componer.
No es memoria
Para hacer la composición de entrada y salida estándar el SO provee las syscalls pipe() y dup2(). Con el primero se crea un archivo anónimo al cual se puede escribir y leer. Con el segundo se pueden reemplazar los file descriptors de los procesos antes y, así, reemplazar la salida estándar del primer programa por el file descriptor que se usa para escribir en el archivo anónimo y reemplazar la entrada estándar del segundo proceso por el file descriptor que se usa para leer del archivo anónimo.

c.

```
int pipe[2];
pipe(pipe);  // creo el pipe

// corro ls -l
if (fork() == 0) { // soy el hijo
    dup2(pipe[1], 1);  // redirijo la salida estándar
    close(pipe[0]); close(pipe[1]);  // cierro descriptores que no uso
    execvp("ls", {"ls", "-l"});  // ejecuto ls -l
}

// corro wc -l
if (fork() == 0) { // soy el hijo
    dup2(pipe[0], 0);  // redirijo la entrada estándar
    close(pipe[0]); close(pipe[1]);
    execvp("wc", {"wc", "-l"});  // ejecuto wc -l
}

// cierro descriptores que no uso
close(pipe[0]); close(pipe[1]);

exit(0);
```

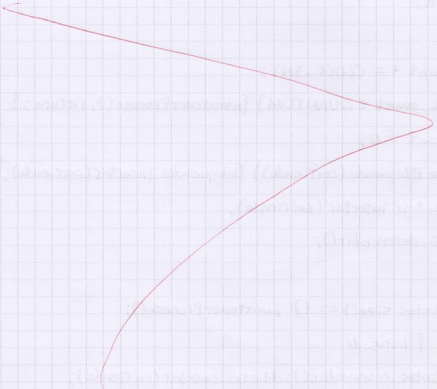
NOTA

2)

```
#define QUANTUM 50
#define CLOCK_INT 10

cola<int> listos;
lista<int> bloqueados;
int clock_count = 0;
int corriendo = -1;

void sched() {
    if (corriendo == -1) {
        if (listos.size() == 0) noHayNingúnProcesoParaCorrer();
        else ponerACorrer(corriendo = listos.proximo());
        return;
    }
    clock_count += CLOCK_INT;
    if (clock_count ≤ QUANTUM) {mantenerProceso(); return;}
    clock_count = 0;
    if (procesoBloqueado(corriendo)) {
        bloqueados.insertar(corriendo);
        listos.desencolar();
    } else {
        if (listos.size() == 1) mantenerProceso();
        else {
            listos.encolar(listos.desencolar());
            ponerACorrer(corriendo = listos.proximo());
        }
    }
}
```

```
void ingresoProceso(int pid) {
    listos.encolar(pid);
    if (corriendo == -1 && listos.size() == 1) {
        ponerACorrer(corriendo = pid);
        clock_count = 0;
}

void volvioProceso(int pid) {
    bloqueados.eliminar(pid);
    ingresoProceso(pid);
}
```

Se asume que los procesos no terminan y que sched, ingresoProceso y volvioProceso no se interrumpen.



(3)

a. Verdadero. La memoria virtual le permite al SO mapear la dirección que usa un proceso para llamar a una función de la glibc a un marco en RAM que se encuentran las instrucciones de esta función y poder así implementar linkeo dinámico. Esto hace que los programas sean más compactos y que sólo sea necesario cargar las funciones una vez en memoria, en vez de tener varias copias.

b. Verdadero. La memoria virtual permite:

- Cargar el código y los datos de los procesos en frames arbitrarios de la memoria física, lo que permite hacer un mejor uso de la memoria sin alterar la semántica del código.
- Tener un mecanismo de protección de la memoria de los procesos creados.
- Hacer copy-on-write a la hora de crear procesos con fork(), lo que resulta en una creación rápida de los procesos y un uso eficiente de la memoria.

c. Falso. El bit se mantiene ya que es un recurso más que puede usar el sistema operativo para implementar funcionalidades. Por ejemplo, es útil para la implementación de políticas de desalojo de páginas como segunda chance y not recently used.

d.

Tres marcos:

Acceso	Faults	Cola de desalojo	Páginas
1	1	1	1
2	2	1 2	1 2
3	3	1 2 3	1 2 3
4	4	2 3 4	4 2 3
1	5	3 4 1	4 1 3
2	6	4 1 2	4 1 2
5	7	1 2 5	5 1 2
1	7	2 5 1	5 1 2
2	7	5 1 2	5 1 2
3	8	1 2 3	3 1 2
4	9	2 3 4	3 4 2
5	10	3 4 5	3 4 5

Cuatro marcos

Acceso	Faults	Cola de desalojo	Páginas
1	1	1	1
2	2	1 2	1 2
3	3	1 2 3	1 2 3
4	4	1 2 3 4	1 2 3 4
1	4	2 3 4 1	1 2 3 4
2	4	3 4 1 2	1 2 3 4
5	5	4 1 2 5	1 2 5 4
1	5	4 2 5 1	1 2 5 4
2	≤ 6		
3	≤ 7		
4	≤ 8		
5	≤ 9		

No presenta la anomalía.

4)

a.

- Agrego un mutex global a todos los agentes: semaphore mutex=1. Se usará para registrar una transacción sin que se produzcan condiciones de carrera sobre los portfolios afectados. Se usa un semáforo y no un spinlock ya que los agentes podrían esperar una cantidad arbitraria de tiempo para conseguir el lock si otros agentes lo obtienen antes constantemente. En esos casos los spinlocks desperdician tiempo de CPU. → Pero lo hace más rápido.
- Agrego un semáforo por agente semaphore turno[n]=0; sobre el que esperarán los agentes para verificar si les toca hacer la transacción. Agrego también un semáforo semaphore generar = 1 para que el generador espere a que un agente haya tomado el turno antes de cambiar el agente seleccionado para que no se pierdan turnos.
  En ambos casos se usan semáforos porque el tiempo de espera puede ser largo.
  Modifico el generador para que haga generar.wait() después de llamar RuidoCósmico() y antes de asignar agenteSeleccionado. Así puede ejecutar RuidoCosmico apenas un agente toma el turno sin esperar a que termine la transacción. (*)
  Modifico AgenteLi) reemplazando el primer if (y todo lo de adentro) por lo siguiente:

(*) Hago que después de asignar agenteSeleccionado haga ∀ 0≤i<n turno[i].signal(); para que los agentes verifiquen si les toca.

```
while (true) {
    turno[i].wait();
    if (agenteSeleccionado == miNumeroDeAgente) break;
}
generar.signal();
mutex.wait();
if (portfolioComprador.puede_comprar(monto) &&
    portfolioVendedor.puede_vender(monto)) {
    portfolioComprador.comprar(monto);
    portfolioVendedor.vender(monto);
}
mutex.signal();
```

- Para que los demás agentes no se queden esperando si le toca el turno a un agente que está esperando a un compradors/vendedor? agrego para cada uno: bool ocupado[i] = true; (**)

Asumo que una función dameAgente(int nro) devuelve en índice en este arreglo del agente con número nro ó -1 si no existe.

Luego, en lugar de hacer generar.wait() el generador hace

```
int index = dameAgente(azar);
if ((index != -1 && !ocupado[index]) || libres.get() == 0)
    generar.wait();
```

Antes del segundo while en Agentes hago

```
libres.getAndInc();
ocupado[i] = false;
```

Después de este while y antes de generar.signal() hago

```
ocupado[i] = true; libres.getAndDec();
```

(*)(*) Agrego un atomic<int> libres = 0; que cuenta la cantidad de agentes listos para tomar su turno.

Modifico también Portfolio:

```
bool puede_comprar (int num) {
    return num > 0;
}
bool puede_vender (int num) {
    return cantidadAcciones >= num && num > 0;
}
bool comprar (int num) { cantidadAcciones += num }
bool vender (int num) { cantidadAcciones -= num; }
```

b. Haría que los agentes tengan que esperar más para verificar si les toca el turno. Esto no afecta la solución de (a) ya que los agentes esperan usando semáforos, que no desperdician tiempo del CPU como los spinlocks.

c. Ninguna solución tiene ecuanimidad ya que es posible que el número de un agente nunca sea generado por RuidoCosmico();