Hoja 1 | Ejercicio 1 | Eric Brandwein 349/16

a) Para poder comunicarse con otros programas con un pipe de la manera descripta, un programa debe escribir el resultado por la salida estándar y recibir los argumentos por la entrada estándar. A su vez, no debe escribir ningún otro dato en la salida estándar, como ser ~~errores~~ mensajes de error.

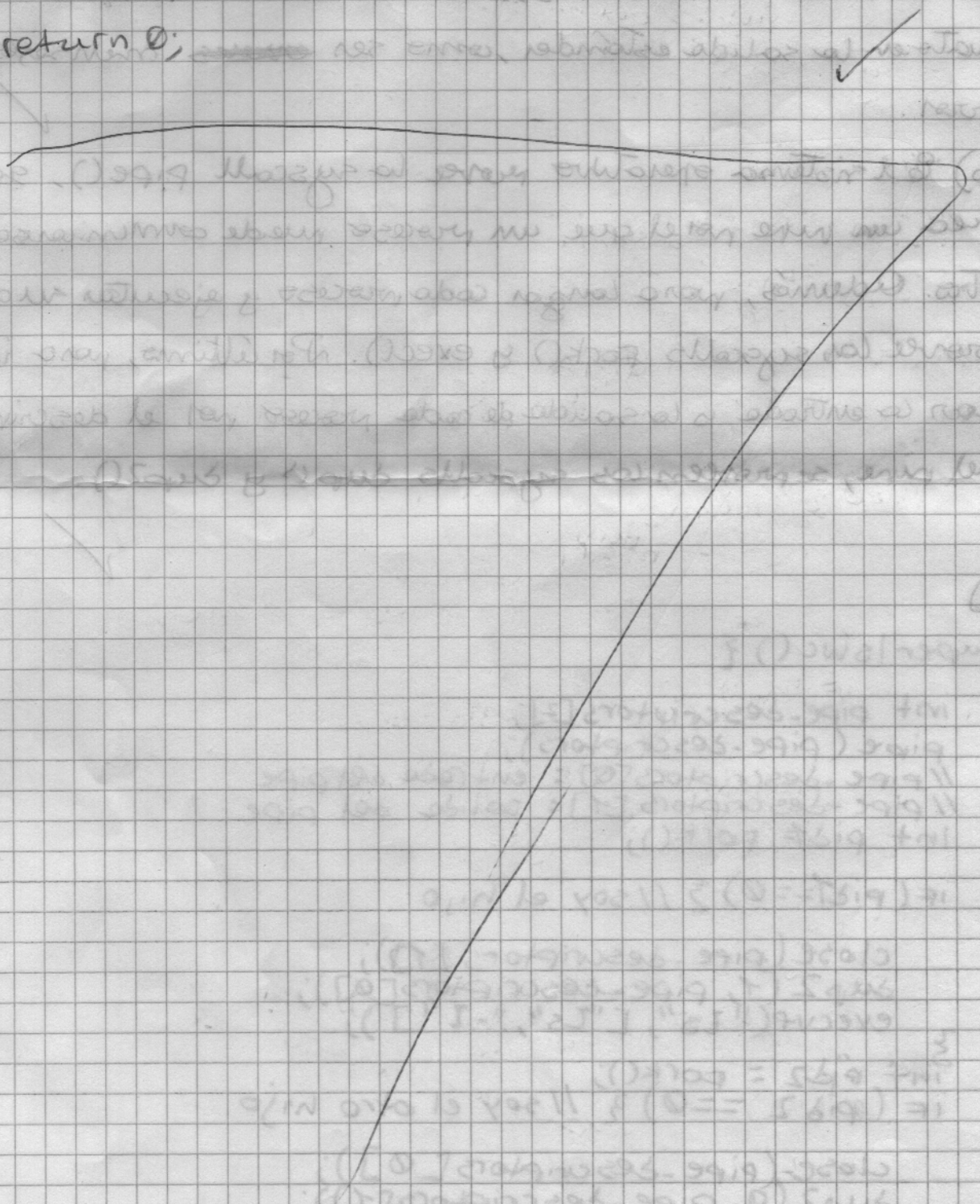
b) El sistema operativo provee la syscall pipe(), que crea un pipe por el que un proceso puede comunicarse con otro. Además, para lanzar cada proceso y ejecutar su código provee las syscalls fork() y execl(). Por último, para intercambiar la entrada y la salida de cada proceso por el descriptor del pipe, se proveen las syscalls dup() y dup2().

c)

```
superlswc() {
    int pipe_descriptors[2];
    pipe(pipe_descriptors);
    //pipe_descriptors[0] : entrada del pipe
    //pipe_descriptors[1] : salida del pipe
    int pid1= fork();

    if(pid1==0) { //soy el hijo

        close(pipe_descriptor;[1]);
        dup2(1, pipe_descriptors[0]);
        execvp("ls", ["ls", "-l"]);
    }
    int pid2 = fork();
    if(pid2 ==0) { //soy el otro hijo

        close(pipe_descriptors[0]);
        dup2(0, pipe_descriptors[1]);
        execvp("wc", ["wc", "-l"]);
    }
```

(sigue →)

```
close(pipe_descriptors[0]);
close(pipe_descriptors[1]);
// cierro los file descriptors por las dudas
for (int i=0; i<2; i++) { // espero a los dos
    wait();
}
// el wc ya imprimió el stdout, asi que termino
return 0;
}
```



Hoja 2 | Ejercicio 2 | Eric Brandwein 349/16

```
List<int> procesos = [];
int ms_actuales = 0;
int proceso_actual = -1;

void ingresoProceso(int pid){
    procesos.push_back(pid);
}

void sched (){
    if(proceso_actual == -1){
        if( procesos.size() > 0){
            nuevo_quantum( 0);
            return;
        } else {
            noHayNingunProcesoParaCorrer();
        }
    } else {
        if( ms_actuales == 50) {
            nuevo_quantum ((proceso_actual +1) % procesos.size())
            return;
        } else if(procesoBloqueado( procesos[proceso_actual])){
            noHayNingunProcesoParaCorrer();
        } else {
            mantenerProceso();
        }
    }
    ms_actuales += 10;
}

void nuevo_quantum(int indice_proceso){
    ms_actuales = 0;
    proceso_actual = indice_proceso;
    ponerACorrer( procesos[proceso_actual]);
}
```

PODRÍA ESTAR BLOQUEADO

```
void volvioProceso(int pid){
    if(procesos[proceso_actual]==pid){
        ponerACorrer(pid);
    }
}
```

¿QUÉ PASA CON EL QUE ESTABA CORRIENDO?

Se asume que los procesos nunca terminan, lo que hace que no haga falta mantener un índice más sofisticado de qué proceso es el actual.

Hoja 3 — Ejercicio 3 — Eric Brandwein 319/16

a) Verdadero. Las operaciones que se realizan por medio de la glibc en memoria se hacen referenciando direcciones virtuales, no físicas. Esto es necesario para hacer uso de la paginación, entre otras cosas. ✓

b) Verdadero. ~~Fals~~ Cuando un nuevo proceso se crea, por ejemplo, mediante un fork(), la memoria virtual del proceso hijo y de del proceso padre apuntan a la misma memoria física, pero cuando alguno escribe en memoria, la página escrita se duplica para no pisar la información del otro proceso, y las direcciones virtuales pasan a apuntar a direcciones físicas diferentes. Sin la memoria virtual no se podría "compartir" la memoria física de esta forma. ✓

d) Para ver si es verdadero o falso, haremos tablas de seguimiento de cada caso, y compararemos los fallos.

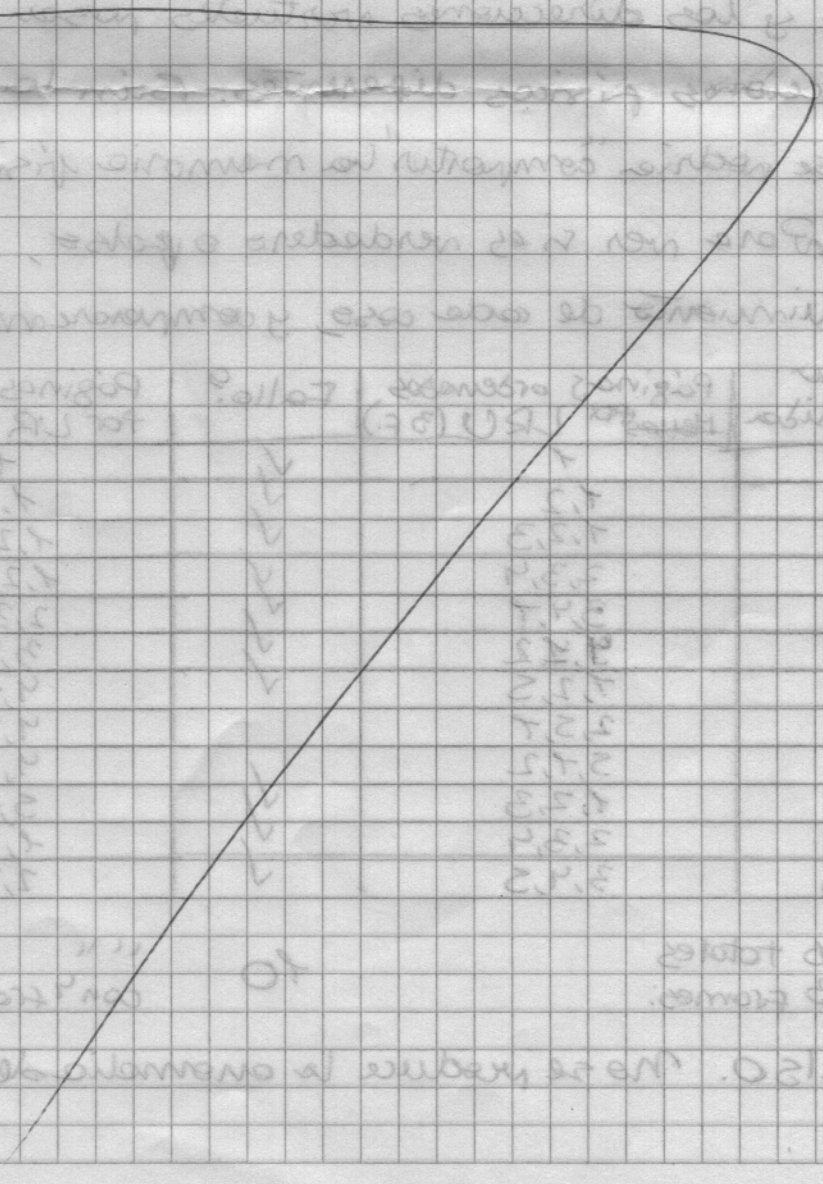
Página Accedida	Páginas ordenadas ~~Mayor~~ por LRU (3 F.)	Fallo?	Páginas ordenadas por LRU (4 F.)	Fallo?
1	1	✓	1	✓
2	1,2	✓	1,2	✓
3	1,2,3	✓	1,2,3	✓
4	2,3,4	✓	1,2,3,4	✓
1	3,4,1	✓	2,3,4,1	
2	4,1,2	✓	3,4,1,2	
5	1,2,5	✓	4,1,2,5	✓
1	2,5,1		4,2,5,1	
2	5,1,2		4,5,1,2	
3	1,2,3	✓	5,1,2,3	✓
4	2,3,4	✓	1,2,3,4	✓
5	3,4,5	✓	2,3,4,5	✓

Fallos totales con 3 frames: 10 — " " con 4 frames: 8

Falso. No se produce la anomalía de Belady, porque 8 ≠ 10 ✓

c) Falso. Se utiliza para saber si la página está siendo referenciada por algún proceso, o si simplemente no hay ningún proceso que la utilice, ~~y por ende~~ En este caso, se podría retirar de la memoria cuando se pide

d) Falso. Se utiliza para saber si la página fue pedida por un proceso en algún momento, y, por ende, si se encuentra actualmente en memoria principal. Con esto se determina, cuando un proceso referencia a una dirección de memoria, si se debería producir un page fault.



Hoja 4 | Ejercicio 4 | Eric Brandwein 314116

a) Se asume que un mismo cliente puede estar realizando una compra y una venta a la vez, pero no más de una cada una. También se asume que un solo agente puede realizar una transacción al mismo tiempo, y que si cuando un agente es seleccionado, el mismo no está listo, entonces puede operar el siguiente.

~~Como no pueden ser~~ Lo primero que haremos es convertir la variable agenteSeleccionado en atómica, para que no ocurra que el generador la modifique mientras un agente la consulta.

Luego, encapsularemos la sección del agente en la que se realiza la transacción para que sea una sección crítica, y que no haya dos agentes realizando transacciones al mismo tiempo. Para esto, utilizaremos un mutex global llamado puedo_operar que comenzará en 1, queriendo decir que el primer agente que lo tome podrá realizar la transacción. El código crítico del agente quedaría entonces así:

```
...
while(agenteSeleccionado.get() != miNumeroDeAgente);
puedo_operar.wait();
if(portfolioVendedor.vender(monto)){
    portfolioComprador.comprar(monto);
}
puedo_operar.signal();
...
```

Se cambia en el agente el if por un while, para que el agente espere a que le toque su turno. Como la función RuidoCosmico() es rápida, probablemente no se quede esperando mucho tiempo, y no se gastarán tantos ciclos de clock haciendo busy waiting.
Como solamente una transacción puede realizarse al mismo tiempo, en ningún momento estarán dos procesos operando sobre el mismo portfolio, así que no surgirán problemas de sincronización. ✓

b) Si RuidoCosmico() tardase mucho, cada Agente podría estar mucho tiempo haciendo busy waiting. Esto ocasionaría que RuidoCosmico() tarde más todavía, porque la CPU se estaría utilizando para esperar.
Para remediar esto, en vez de utilizar la variable agente seleccionado, podríamos utilizar una lista de mutex, uno por cada Agente, a los que los Agentes escuchen para saber si fueron seleccionados. El código resultante sería así:

```
mutex seleccionados[1000]; // comienzan en 0.

//Generador cuántica
while (true){
   double azar = RuidoCosmico();
   seleccionados[(azar * 1000) % 1000].signal();
}

//Agente
while (true){
   ...
   seleccionados[miNumeroDeAgente].wait();
   //solamente cambiamos el while anterior por esta linea
   ...
```

Ahora, en vez de que cada proceso ocupe la CPU esperando, el sistema operativo le informará cuando puede proceder con la ejecución.

c) Si asumimos que RuidoCosmico() tiene una distribución uniforme, y que el scheduler es uno relativamente justo entre procesos, todos los agentes deberían tener una oportunidad de realizar su porción crítica en el infinito en las dos versiones del sistema. Sin embargo, en los dos sistemas podría ocurrir inanición, sea porque nunca un agente es seleccionado, o porque siempre que quiere realizar su transacción hay otro agente realizando la suya, ocupando el recurso compartido.

Estos nos llevan a ~~dos~~ varios casos posibles, dependiendo de cómo interpretemos fairness:

- Si ~~fairness~~ decimos que fairness es la condición de que todos los procesos puedan intentar entrar a la sección crítica, o, más bien, que puedan ejecutar su código no crítico, entonces se cumple.
- Si decimos que es la condición de que, en el infinito, todos los procesos corran su sección crítica, entonces también se cumple.
- Si, en cambio, fairness lo tomamos como el hecho de que no puede haber inanición, entonces no es así.