1. In Section 4.3, in what order did processes complete? Explain exactly how and

when the scheduler chose to switch from one process to another.

Process prB (priority 11) completed first, then the process prA (priority 10). The scheduler always selects the highest priority ready process to run. So the prB was executed first before the prA was given to the CPU.

2. In Section 4.4, in what order did processes complete? Explain how and when the

scheduler chose to switch from one process to another.

Again the process prB finished first and then the prA, the difference here was that the compl() function confirmed the termination order, when the scheduler chose prB first and after prB terminated the kernel automatically sent a message to main and comp(receive()) report completion.

3. In Section 4.5, in what order did processes complete? Explain how the scheduler

made decisions to change from one process to another.

When prB priority was increased to 21, it still finished first and the prA, but this time the scheduler gave absolute preference to prB because of the big difference in the priority.

4. In Section 4.6, in what order did processes complete? Explain how the scheduler

made decisions to change from one process to another.

This time was different, both prA and prB had the same priority (10), so the scheduler trated them equally and switched between them in a round robin fashion, The output showed alternating chunks of A and B.

5. In Section 4.7, state the sequence of events that occur, including process exit, and

explain how the resulting output is affected.

1 prA and prB were started with equal priority (12), so they lternated each other.

2 then prC ws created with higher priority (15).

3 the scheduler immediately switched to prC, printing 100 C.

4 after prC finished, the scheduler resumed round robin the execution of prA and prB until both finished.

5 finally compl() reported the completion of all three process.

Resuling output showed a block of C appearing in the middle of the interleaved A and C, and then the termination messages.

/\* Incluimos el archivo de cabecera de Xinu, que contiene las definiciones y funciones necesarias

para programar en el sistema operativo Xinu, como la creación de procesos y la impresión en la consola. \*/

#include <xinu.h>

/\* Declaramos tres variables para almacenar los identificadores de los procesos (PIDs).

Un PID es como el número de identificación único de un proceso en el sistema. \*/

pid32 pidA;

pid32 pidB;

pid32 pidC;

/\* Definimos una función llamada 'prch' (abreviatura de "print character").

Esta función toma un carácter 'ch' y un número 'ntimes' como argumentos.

Su propósito es imprimir el carácter 'ch' en la consola la cantidad de veces especificada por 'ntimes'. \*/

void prch(char ch, int32 ntimes) {

int i; // Declaramos un contador para nuestro bucle.

// Usamos un bucle 'for' para repetir la impresión del carácter.

for(i=0; i<ntimes; i++) {

// 'kputc' es una función de Xinu que imprime un solo carácter.

kputc(ch);

}

}

/\* Definimos una función llamada 'compl' (abreviatura de "complete").

Esta función toma el PID de un proceso y determina cuál de los procesos que creamos ha terminado.

Luego, imprime un mensaje descriptivo en la consola. \*/

void compl(pid32 pid) {

// Comprobamos si el PID recibido es igual al de nuestro primer proceso.

if(pid == pidA) {

kprintf("\nprocess prA has completd\n");

// Si no es el primero, comprobamos si es el segundo.

} else if (pid == pidB) {

kprintf("\nprocess prB has completed\n");

// Si no es ninguno de los dos, comprobamos si es el tercero.

} else if (pid == pidC) {

kprintf("\nprocess prC has completed\n");

// Si el PID no coincide con ninguno de los procesos que esperamos, algo salió mal.

} else {

kprintf("\nsomething is wrong\n", pid);

}

}

/\* Esta es la función principal del programa. En Xinu, 'main' es un proceso en sí mismo

y es el punto de entrada desde donde se inicia la ejecución. \*/

process main(void) {

// Creamos nuestro primer proceso, 'prA'.

// 'create' es la función que crea un nuevo proceso con sus propios recursos.

// - (void \*)prch: La función que el nuevo proceso ejecutará.

// - 4096: El tamaño de la pila de memoria para el proceso.

// - 12: La prioridad del proceso. A mayor número, mayor importancia.

// - "prA": El nombre del proceso.

// - 2, 'A', 800: Son los dos argumentos que se pasan a la función 'prch' (el carácter y las veces a imprimir).

pidA = create((void \*)prch, 4096, 12, "prA", 2, 'A', 800);

// 'resume' inicia la ejecución del proceso que acabamos de crear.

resume(pidA);

// Creamos y reanudamos el segundo proceso, 'prB', de manera similar.

pidB = create((void \*)prch, 4096, 12, "prB", 2, 'B', 800);

resume(pidB);

// El proceso 'main' se detiene por 1 milisegundo para permitir que los procesos 'prA' y 'prB'

// comiencen a ejecutarse y compartan el tiempo del procesador.

sleepms(1);

// Creamos y reanudamos el tercer proceso, 'prC', con una prioridad más alta (15).

// Esto significa que el sistema le dará más tiempo de CPU en comparación con 'prA' y 'prB'.

pidC = create((void \*)prch, 4096, 15, "prC", 2, 'C', 100);

resume(pidC);

// Imprimimos un mensaje para indicar que el proceso 'main' está a punto de esperar.

kprintf("\nmain is waiting\n");

// 'receive' bloquea el proceso 'main' y lo pone a esperar un mensaje.

// En Xinu, cuando un proceso hijo termina, le envía un mensaje a su padre (en este caso, 'main').

// El valor que recibe es el PID del proceso que acaba de terminar.

// Llamamos 'receive' tres veces para esperar a que los tres procesos que creamos terminen.

// Luego, pasamos el PID a la función 'compl' para que identifique y anuncie cuál proceso ha finalizado.

compl(receive());

compl(receive());

compl(receive());

// Una vez que todos los procesos hijos han terminado, 'main' continúa su ejecución.

kprintf("\nmain exiting\n\n");

// 'return OK' indica que el programa ha finalizado correctamente.

return OK;

}