



UNIVERSIDAD DE GUADALAJARA  
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS  
DIVISIÓN DE ELECTRÓNICA Y COMPUTACIÓN



---

# PROGRAMA 4 – ALGORITMO DE PLANIFICACIÓN FCFS (FIRST COME, FIRST SERVED)

---

SEMINARIO DE SOLUCIÓN DE PROBLEMAS DE SISTEMAS OPERATIVOS

**SECCIÓN:** D01

**CÓDIGO:** 215468696

**PROFESOR:** VIOLETA DEL ROCIO BECERRA VELAZQUEZ

**ALUMNO:** DOMÍNGUEZ DURAN OSCAR ALEJANDRO

**CARRERA:** INGENIERÍA EN COMPUTACIÓN



**12 DE MAYO DE 2021**

## ACTIVIDAD DE APRENDIZAJE 8 – ALGORITMO DE PLANIFICACIÓN FCFS (FIRST COME, FIRST SERVED) CONTINUACIÓN

### OBJETIVO

---

Como el mismo nombre de la actividad lo indica, esta resulta ser una continuación directa de lo visto en la primera actividad correspondiente a la implementación del algoritmo FCFS. Esta continuación supone la adición de nuevas funcionalidades al programa pasado, tales como la posibilidad de crear un nuevo proceso al oprimir la tecla “N” o el poder mostrar en todo momento la tabla de los procesos, es decir, el BCP de todos los procesos con los tiempos correspondientes que lleven a la hora de hacer la pulsación de la tecla “B”.

Podemos decir que, a priori, suenan como un par de añadidos bastante más sencillos a comparación de lo que se ha estado realizando en actividades previas, pero es importante recalcar que este programa depende en gran medida de que los programas previos se hayan hecho bien, si no, se nos pueden presentar muchas dificultades; esto último ya que, pese a la simpleza de las nuevas especificaciones, se deben tomar en consideración bastantes aspectos. Por ejemplo, si queremos añadir un nuevo proceso al pulsar la tecla correspondiente, necesitamos tener en consideración que hay que generar todos sus campos, tales como el ID, la operación, el resultado e inicializar sus correspondientes tiempos acorde a como es presentado. De esta manera, el añadir un nuevo proceso en cualquier momento requiere de una correcta gestión de los procesos que están listos, los nuevos y los procesos bloqueados, además de que hay que tener en cuenta que se debe respetar la cantidad máxima de procesos que pueden estar en memoria (es decir, 5 procesos).

Respecto a la tabla de procesos con sus respectivos BCP, se deberán mostrar los tiempos de todos los procesos, incluyendo a los procesos que estén en la cola de procesos nuevos. Evidentemente, dependiendo de en que momento se muestre dicha tabla habrá procesos que no han terminado aún y, por ende, no podrán tener todos sus tiempos calculados; lo cual implica una serie de validaciones para saber que tiempos se pueden mostrar según el estado en el que se encuentre el proceso y que tiempos se deben considerar como nulos. Finalmente, cabe mencionar, que el programa debe contar con las mismas funcionalidades que el programa pasado y por ende respetar las especificaciones planteadas en dicho programa.

### DESARROLLO

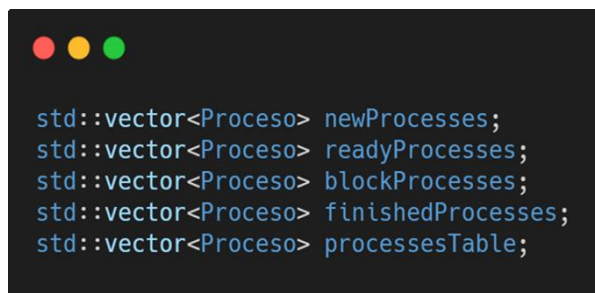
---

**Lenguaje y herramientas utilizadas:**

Para este programa seguí trabajando con el lenguaje de C++ y el *framework* de Qt, ya que, al ser una continuación inmediata del programa pasado, me resultó más sencillo y práctico trabajar con lo que ya tenía hecho con estas herramientas. De igual manera, considero que, llegados a este punto, me siento bastante seguro y cómodo con el uso de este lenguaje y *framework*. En el seminario he estado trabajando con C# y Visual Studio para el desarrollo de un par de actividades relacionadas con la gestión de procesos, pero he de admitir que aún no me siento del todo desenvuelto con estas y pese a las similitudes que existen entre Qt y Visual Studio para el desarrollo de aplicaciones con interfaz gráfica, las diferencias respecto al manejo de componentes y/o la estructura de las clases me parecen un tanto extrañas por decirlo de alguna manera, no son complejas, pero estoy tan acostumbrado a C++ y Qt de momento que el estar aprendiendo a usar C# a la par, hace que pueda apreciar estos detalles y por ende, saber que desarrollar en C# tiene una curva de aprendizaje más elevada para mí de momento como para querer trasladar o empezar de 0 con el simulador. No obstante, sigo abierto a la posibilidad de usarlo en algún programa futuro puesto que sigo aprendiendo a usarlo y en mi opinión, considero que nunca es bueno amarrarse o limitarse al uso de una herramienta en específico, al contrario, soy partidario de usar lo que mejor me sirva para cada situación.

### Funcionamiento del programa:

Lo primero que realice en este programa fue cambiar algunas de las estructuras de datos que se emplearon en el programa pasado, debido a que en el caso de las colas, pese a que el añadir y eliminar elementos de estas resulta ser más intuitivo y legible, lo cierto que recorrerlas para poder mostrar su contenido resulta poco eficiente debido a los constantes *pops* y *push* que se hacen de un mismo elemento; además de que el eliminar y agregar de esa manera a elementos que en la lógica del programa no deberían ser eliminados aún puede desencadenar en *bugs* potenciales para el desarrollo de esta actividad a la hora de crear nuevos procesos o de mostrar estado de la tabla de los procesos. Debido a eso, opté por dejar que las 4 estructuras de datos en las que se moverían los procesos fuesen todos vectores estándar propios del lenguaje C++ ya que nos permiten un mejor control de los elementos y sobre todo son bastantes flexibles, por lo que sin problemas se le puede dar el comportamiento lógico de una cola a un vector. De esta manera ahora tenemos estas 4 estructuras básicas para el funcionamiento de nuestro programa:



```
std::vector<Proceso> newProcesses;  
std::vector<Proceso> readyProcesses;  
std::vector<Proceso> blockProcesses;  
std::vector<Proceso> finishedProcesses;  
std::vector<Proceso> processesTable;
```

Ilustración 1 - Estructuras de datos principales

Respecto a los procesos, la clase que los moldea fue cambiada un poco respecto al programa anterior. Le agregué un atributo publico que hace referencia al estado lógico que posee el proceso, es decir, para saber en que estructura está en ese momento. También se le incorporaron nuevas sobrecargas de operadores (el mayor que y menor que) para poder ordenar a los procesos en base a su ID, de momento estas sobrecargas son utilizadas solo para poder mostrar la tabla de procesos ordenada, aunque en futuro quizá sean necesarios para poder comparar algún criterio de prioridad. La razón de que el estado de cada proceso fuera público, es que considero que el estado de cada proceso debe ser de fácil acceso para el planificador y poder manipular su valor según en que estructura de datos se almacenado el proceso.

A screenshot of a code editor with a dark background and light-colored text. The code defines a namespace named 'States' which contains an enumeration. The enumeration lists five states: 'Nuevo', 'Listo', 'Bloqueado', 'Finalizado', and 'Count'. The code is as follows:

```
namespace States {  
    enum {  
        Nuevo,  
        Listo,  
        Bloqueado,  
        Finalizado,  
        Count  
    };  
}
```

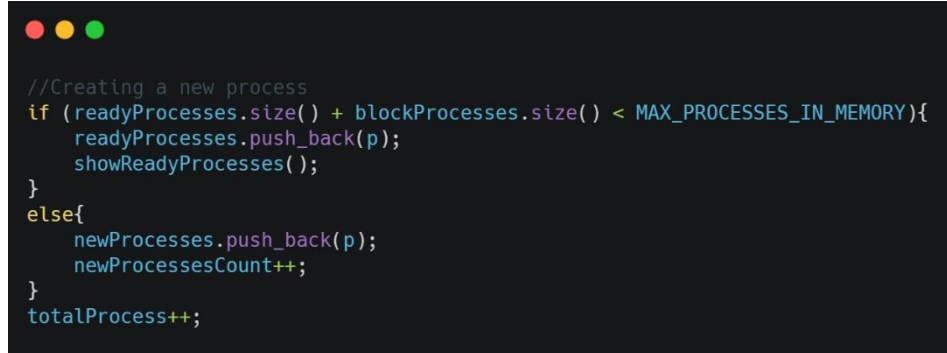
*Ilustración 2 - Posibles estados de cada proceso*

Los estados son asignados a cada proceso justo antes de que sean asignados a cualquier estructura de datos. Por ejemplo, cuando todos los procesos son creados, justo antes de ser añadidos al vector de procesos nuevos su estado se establece como nuevo, cuando el proceso pasa al vector de procesos nuevos su estado se establece en listo y así por cada uno de las estructuras y estados restantes.

Aclarado lo anterior, podemos proseguir a explicar la incorporación de las nuevas funcionalidades requeridas para esta actividad. Decidí comenzar por la implementación de la creación de nuevos procesos mediante la pulsación de la tecla “N”. Para detectar la tecla presionada, se sigue haciendo uso de la clase *QKeyPressEvent*, la cual se encarga de detectar toda tecla presionada cuando el *focus* del programa está localizado en la ventana principal, hago hincapié en este detalle puesto que más adelante se describirá uno de los problemas surgidos en la realización de esta actividad debido al cambio del *focus* del programa entre algunos componentes.

Cada el usuario teclea “N” se detecta, al igual que con las otras teclas, el valor en ASCII y procede a llamar a la función de “*createNewProcess*”, en esta función se generan todos los datos necesarios para el proceso y se verifica la cantidad de procesos que estén en memoria, es decir, verificamos si la cantidad de procesos que están listos más los procesos que están en bloqueados no rebasan la cantidad máxima de procesos en memoria, o sea, 5. En caso de que ya

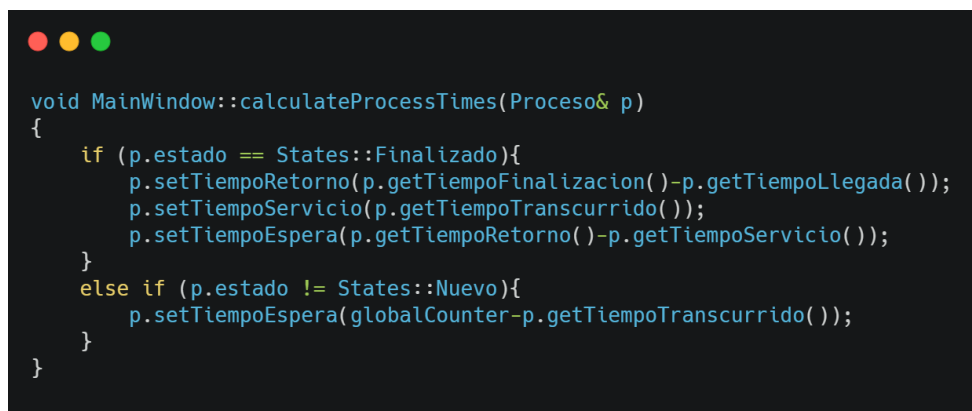
haya 5 procesos en memoria y se cree un nuevo proceso, este deberá irse directamente a la cola de procesos nuevos tras ser creado. La siguiente validación se realiza cada que un nuevo proceso es creado mediante la pulsación de la tecla correspondiente, anexo la parte del código correspondiente:



```
//Creating a new process
if (readyProcesses.size() + blockProcesses.size() < MAX_PROCESSES_IN_MEMORY){
    readyProcesses.push_back(p);
    showReadyProcesses();
}
else{
    newProcesses.push_back(p);
    newProcessesCount++;
}
totalProcess++;
```

*Ilustración 3 - Validación para añadir un nuevo proceso en ejecución*

Una vez explicada la parte de como se crean nuevos procesos, podemos revisar la siguiente funcionalidad: mostrar la tabla de procesos con sus respectivos tiempos. Para llevar a cabo lo anterior fue necesario crear una función que calcule los tiempos de un proceso determinado en donde, si su estado es finalizado, podrá obtener los tiempos de retorno, servicio y espera; en caso contrario, solo calculará el tiempo de espera en base al contador global siempre y cuando el proceso no sea nuevo. Esto último debido a que sabemos que todos los procesos nuevos solo deben mostrar su ID, su operación y su tiempo máximo estimado. Anexo la parte del código que describe la función mencionada:



```
void MainWindow::calculateProcessTimes(Proceso& p)
{
    if (p.estado == States::Finalizado){
        p.setTiempoRetorno(p.getTiempoFinalizacion()-p.getTiempoLlegada());
        p.setTiempoServicio(p.getTiempoTranscurrido());
        p.setTiempoEspera(p.getTiempoRetorno()-p.getTiempoServicio());
    }
    else if (p.estado != States::Nuevo){
        p.setTiempoEspera(globalCounter-p.getTiempoTranscurrido());
    }
}
```

*Ilustración 4 - Método para calcular los tiempos de un proceso*

Los demás tiempos se asignan en distintas partes del programa y no requieren ser calculados como tal, en cualquier otro estado que no sea nuevo se pueden obtener tiempos de manera parcial. Este método es llamado cada que se va a crear la tabla de procesos, ya que cuando se presiona la tecla “P”, procede a crear un arreglo con todos los procesos en el

programa, es decir, un arreglo que contenga los procesos nuevos, los procesos listos, los procesos bloqueados y los procesos finalizados. Una vez tenemos estos procesos en el vector que representa la tabla, se procede a ordenar este arreglo mediante un método estándar de C++ cuyo único requisito es poder comparar los objetos que componen al arreglo por ordenar, Cómo el estado de la tabla siempre será diferente cada que se quiera mostrar, es importante tener este vector limpio antes de que se vaya a crear la nueva tabla. Anexo el código del método descrito anteriormente:

```
void MainWindow::createProcessesTable()
{
    int i;
    if (!processesTable.empty()){
        processesTable.clear();
    }
    for (i = 0; i < totalProcess; i++){
        if (i < (int)readyProcesses.size()){
            processesTable.push_back(readyProcesses[i]);
        }
        if (i < (int)blockProcesses.size()){
            processesTable.push_back(blockProcesses[i]);
        }
        if (i < (int)newProcesses.size()){
            processesTable.push_back(newProcesses[i]);
        }
        if (i < (int)finishedProcesses.size()){
            processesTable.push_back(finishedProcesses[i]);
        }
    }
    //Sorting the elements of the processes table
    std::sort(processesTable.begin(), processesTable.end());
}
```

*Ilustración 5 - Método para crear la tabla de procesos y ordenarla*

Ya con la tabla de procesos creada y ordenada, podemos mostrar la información de cada proceso dependiendo el estado en el que se encuentre cuando se quiere mostrar la tabla de procesos. Sabemos que independientemente del estado el proceso siempre se va a mostrar su ID, la operación que va a realizar y el tiempo estimado del proceso. Partiendo de eso, se comienza a validar el estado del proceso para saber cuales tiempos de este se pueden mostrar y cuales no porque no han sido calculados o no aplican para el proceso. Por ejemplo, sabemos que, si el estado finalizó, entonces podemos mostrar el resultado ya sea el resultado de la operación o el mensaje de error, según sea su finalización; por ende, todos los procesos que no tengan este estado de finalización no pueden mostrar el resultado y por ende se mostraría un NULL. En el caso de que los procesos sean nuevos tanto su tiempo de llegada como su tiempo de servicio deberán permanecer en NULL si o sí, puesto que no han sido ejecutados. Para mostrar los tiempos de retorno y finalización estos deben de haber sido terminados, por lo que si el proceso no lo está también mostrará un NULL en estos tiempos. Finalmente, el tiempo de

espera y de respuesta es algo que solo aquellos procesos que no sean nuevos pueden mostrar, así que también se valida que esto se cumpla. Además de los tiempos mostrados que se necesitan en otra columna también se muestra el estado actual del proceso. La manera en la que se decide que tiempos y datos mostrar depende del estado de cada proceso, por lo que cada que vayamos a mostrar un proceso es necesario determinar que datos se pueden mostrar; mi primer acercamiento fue una serie de estructuras condicionales que evaluaban el estado y a partir de este, se mostraban los tiempos que les correspondían. El problema con ese acercamiento es que eran demasiadas líneas de código repetidas y, además, había validaciones que se podían factorizar, por lo que me entretuve bastante en obtener la manera correcta de seleccionar la información a mostrar por cada proceso. En términos generales, la lógica que se siguió fue la siguiente: si el estado de un proceso es “Nuevo”, lo que se mostrará en su resultado, tiempo de finalización, tiempo de retorno, tiempo de llegada, tiempo de servicio, tiempo de respuesta y tiempo de espera serán todos NULL. En caso de que no sea un proceso “Nuevo”, y no haya terminado, podemos mostrar los tiempos de llegada, servicio, respuesta y espera; si el proceso ha terminado, entonces podemos mostrar el resultado, el tiempo de finalización y el tiempo de retorno, en cualquier otro caso estos apartados serán NULL. Por si no se entendió del todo bien, anexo la parte del código correspondiente:

```

1  switch (p.estado){
2      case States::Finalizado:
3      case States::Listo:
4      case States::Bloqueado:
5      case States::Ejecutandose:
6          arrived = QString::number(p.getTiempoLlegada());
7          service = QString::number(p.getTiempoServicio());
8          request = QString::number(p.getTiempoRespuesta());
9          waiting = QString::number(p.getTiempoEspera());
10         switch (p.estado) {
11             case States::Finalizado:
12                 finish = QString::number(p.getTiempoFinalizacion());
13                 retrn = QString::number(p.getTiempoRetorno());
14                 if (p.getFinalizacion()){
15                     result = QString::number(p.getResultadoOperacion());
16                 }else{
17                     result = "ERROR";
18                 }
19                 break;
20             default:
21                 result = finish = retrn = "NULL";
22         }
23         break;
24     default:
25         result = finish = retrn = arrived = service = request = waiting = "NULL";
26         break;
27 }

```

*Ilustración 6 - Condiciones para decidir qué tiempos y datos mostrar de cada proceso*



Una vez obtenidas las cadenas de la información que voy a mostrar en la tabla, procedo a realizar lo mismo para obtener el valor a mostrar en la columna del estado del proceso:

```
//Showing the process state
switch (p.estado){
    case States::Listo:      stateProcess = "LISTO";      break;
    case States::Nuevo:      stateProcess = "NUEVO";      break;
    case States::Ejecutandose: stateProcess = "EJECUCION";  break;
    case States::Bloqueado:   stateProcess = "BLOQUEADO";  break;
    case States::Finalizado:  stateProcess = "FINALIZADO"; break;
    default: ;
}
```

*Ilustración 7 - Verificando el estado del proceso a mostrar*

Llegados a este punto fue donde tuve el mayor problema de esta actividad, como ya había mencionado anteriormente, fijar el *focus* del programa en la ventana principal es de vital importancia para que se puedan escuchar los eventos de presión de una tecla, quiero suponer que esto se debe al funcionamiento interno de la clase *QKeyPressEvent*. De cualquier manera, el punto es que cada que la tabla era mostrada, cambiaba de página (es decir, mostraba la página que contenía la tabla), y si seleccionaba algún elemento de la tabla o alguna parte de la tabla, el *focus* del programa cambiaba de estar en la ventana principal a la *QTableWidget*. Leyendo un poco al respecto me topé con que este *focus* cambia entre los componentes de la aplicación según son utilizados, y la única manera de quitar el *focus* de una tabla como tal era destruir dicha tabla; por obvias razones no podía eliminar la tabla para poder detectar el tecleo, así que, adentrándome dentro de la documentación de Qt, pude darme cuenta de que este *focus* puede ser desactivado en el componente que quiera como si de una propiedad se tratase. Desactivando este aspecto de la tabla y fijando el *focus* del programa en la venta principal antes de mostrar la tabla todo funcionó como era debido. Cuando se tecldea la “C” el programa valida en que página está para saber si fue una pausa normal, o si, por el contrario, podemos dejar de mostrar la tabla y volver a la página en la que se muestra la ejecución de todos los procesos.

```
else if (event->key() == Qt::Key_C and state == PAUSED){
    qDebug() << "Continue";
    state = RUNNING;
    if (ui->stackedWidget->currentIndex() == SHOW_TIMES_PROCESSES){
        ui->stackedWidget->setCurrentIndex(SHOW_PROCESSES);
    }
    pause.quit();
}
else if (event->key() == Qt::Key_B and state == RUNNING){
    qDebug() << "BCP";
    state = PAUSED;
    ui->stackedWidget->setCurrentIndex(SHOW_TIMES_PROCESSES);
    showProcessesBcp();
    this->setFocus();
    pause.exec();
}
```

*Ilustración 8 - Parte del código en la que se detecta la tecla "B" y su ...*



PROGRAMA 4 – ALGORITMO FCFS (CONTINUACIÓN)

Descritas las funcionalidades principales solicitadas para esta actividad, considero necesario mostrar algunas capturas de pantalla del programa siendo ejecutado y mostrando el correcto funcionamiento de las nuevas funcionalidades implementadas. En la primera captura a mostrar veremos lo que pasa cuando tecleamos la “N” y aún espacio en la memoria de procesos:

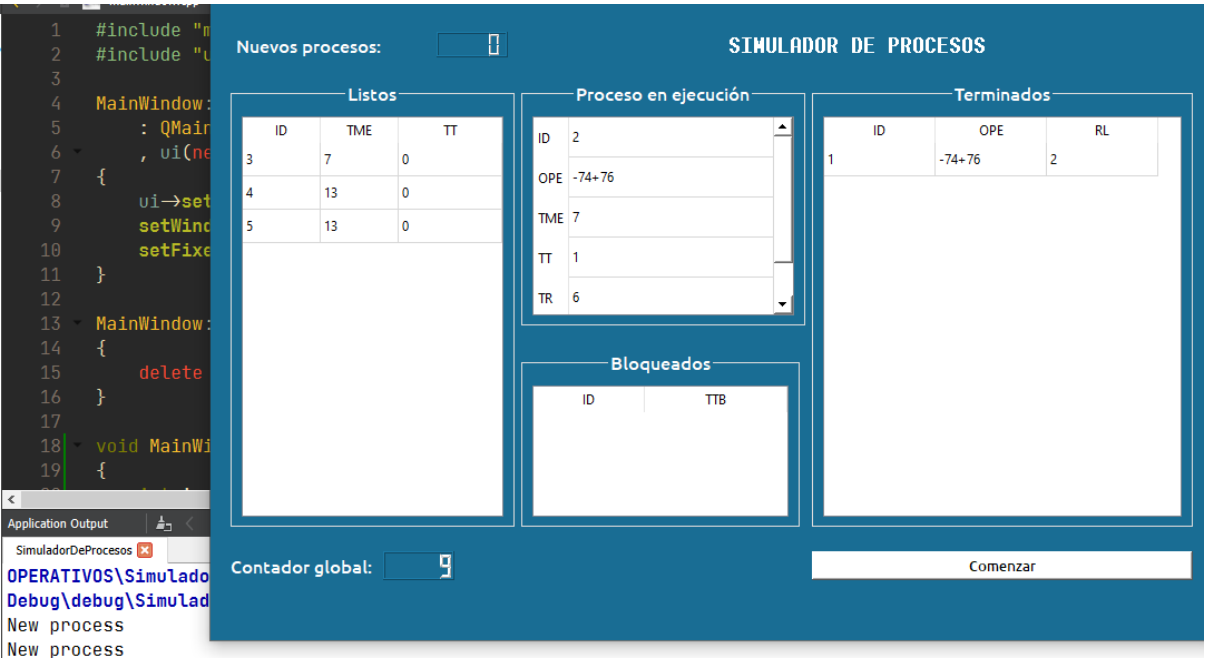
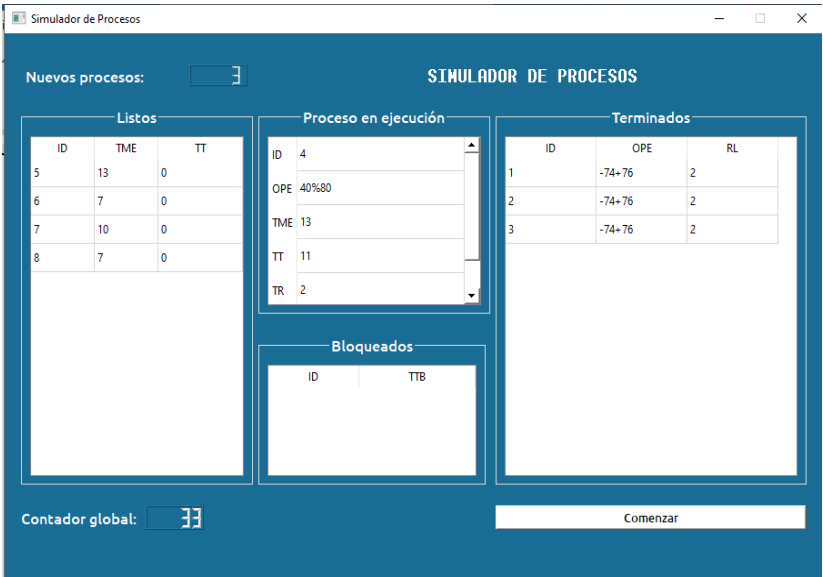


Ilustración 9 - Agregando 2 procesos cuando hay espacio en la memoria

En el caso de que ya haya 5 procesos en memoria, si se crean nuevos procesos entonces estos pasan a estar en la cola de procesos nuevos, por ende, el contador de procesos nuevos aumentaría:

Ilustración 10 - Nuevos procesos en la cola de nuevos



A continuación, se muestra la captura de pantalla correspondiente a la tabla de procesos cuando hay bloqueados, listos, nuevos y en ejecución, pero aún no finalizados:



	T. Estimado	T. Llegada	T. Finalizacion	T. Retorno	T. Respuesta	T. Espera	T. Servicio	Estado
1	9	0	NULL	NULL	0	3	1	BLOQUEADO
2	9	0	NULL	NULL	0	4	0	BLOQUEADO
3	9	0	NULL	NULL	0	4	0	EJECUCION
4	9	0	NULL	NULL	0	4	0	LISTO
5	9	0	NULL	NULL	0	4	0	LISTO
6	9	NULL	NULL	NULL	NULL	NULL	NULL	NUEVO
7	9	NULL	NULL	NULL	NULL	NULL	NULL	NUEVO
8	9	NULL	NULL	NULL	NULL	NULL	NULL	NUEVO

Ilustración 11 - Tabla de procesos sin procesos finalizados

En la siguiente imagen se puede apreciar como se muestra la tabla tras una serie de acciones que permiten observar la tabla con todos los estados posibles para los procesos:



	T. Estimado	T. Llegada	T. Finalizacion	T. Retorno	T. Respuesta	T. Espera	T. Servicio	Estado
1	11	0	NULL	NULL	0	22	3	BLOQUEADO
2	13	0	NULL	NULL	0	24	1	EJECUCION
3	14	0	6	6	0	6	0	FINALIZADO
4	13	0	8	8	0	7	1	FINALIZADO
5	14	0	NULL	NULL	0	21	4	LISTO
6	7	6	20	14	0	8	6	FINALIZADO
7	15	8	NULL	NULL	0	24	1	BLOQUEADO
8	6	20	NULL	NULL	0	25	0	LISTO
9	12	NULL	NULL	NULL	NULL	NULL	NULL	NUEVO

Ilustración 12 - Tabla de procesos con todos los estados y sus tiempos

## CONCLUSIÓN

---

Esta actividad, pese a ser una continuación de la actividad anterior, me costó un poco más de trabajo que el programa pasado. No fueron problemas tan complejos con ya había experimentado en otras ocasiones pero si tuve que estar *debuggeando* bastantes más cosas, tuve que cambiar las estructuras de datos en las que almacenaba los procesos ya que aunque las colas nos permiten entender lo que se hace de manera más sencilla, lo cierto es que a la hora de iterar sobre los elementos de las colas el hecho de que tengamos que sacar el primer elemento y añadirlo al final de esta suponía que en ciertos puntos del programa se perdía dicha referencia. Es por eso por lo que decidí dejar todas las estructuras de los procesos como simples vectores y darles el comportamiento lógico de una cola, también tuve que revisar algunos métodos en los que trabajaba con las referencias directas de un elemento puesto que al insertar un nuevo elemento a la cola de procesos listos el incremento de este vector podría ocasionar la pérdida de la referencia del proceso con el cual estaba trabajando.

Además de esos pequeños inconvenientes, sin duda lo que más me costó en esta actividad fue el poder mostrar la tabla de procesos con los datos correctos y poder seguir con la ejecución del programa y no fue tanto un problema de lógica sino de lograr comprender como se manejaba el *focus* del programa y como es que se relacionaba con la clase que permite detectar el cuando una tecla es presionada. Viendo el lado bueno, pude aprender a como quitarles esta propiedad a los componentes que no la requieren para la simulación, de manera que siempre sea la ventana principal del programa quién tiene el *focus* y por ende siempre poder escuchar los eventos que necesite. De ahí en más, no hubo mayor complicación, bien es cierto que abstraer de la mejor manera posible el como y cuales tiempos deberían mostrarse según el estado del proceso fue algo tardado, más no complicado, a decir verdad, considero que añadiendo el estado al proceso es algo que me hubiera servido en actividades pasadas y de seguro esta misma clase me será de utilidad en futuros programas con sus respectivas adecuaciones claro está.

Respecto al resultado obtenido con el programa, he de decir que quedé bastante satisfecho con como se ejecuta y también con los cambios que realicé en el código fuente ya que con la semana adicional de tiempo que se nos dio para esta actividad pude darme a la tarea de mejorar algunos aspectos de este y factorizar algunas partes para que fuesen menos líneas y de cierta manera fuera más entendible. Por último, me gustaría mencionar que pese a las complicaciones que surgieron a lo largo de la actividad, me alegra que estas fueran más relacionadas con el cómo utilizar la herramienta del *Qt* y no tanto con la lógica que estoy siguiendo a la hora de codificar. Además, considero que esta actividad me ha servido para reforzar algunos aspectos esenciales de las herramientas que estoy usando y, claro está, para seguir aprendiendo a como usarlas, ya que ahora pude aprender bastante aspectos relacionados con el concepto del *focus* del programa y el cómo este se puede relacionar con otras clases que interactúan con el programa en manera de componentes. Sin duda alguna la semana adicional que se nos brindó me sirvió no solo para mejorar mi implementación, sino que también para

poder estar menos presionado con otros proyectos y entregables, es un aspecto que me permitió cumplir el objetivo de la actividad en tiempo y forma y que, de cierta manera, me hace sentir satisfecho con el resultado y motivado de seguir adelante con los programas posteriores.

Link del vídeo del programa:

- [\(223\) Programa 4 \(Continuación de FCFS\) - SSP Sistemas Operativos - YouTube](#)