Informe del Obligatorio - Segunda Parte

Ramón A. Lorenzo y Nicolás A. Cartalla

Universidad Católica del Uruguay

Índice

Resumen

En este trabajo se creará un programa siguiendo las instrucciones y cumpliendo los requerimientos que nos fueron proporcionados por los profesores de nuestro curso de Sistemas Operativos. El mismo será un planificador de corto plazo, que se programará en lenguaje JAVA, en su versión 18. El planificador implementará un algoritmo de planificación de colas multinivel con 3 colas, las cuales serán una para cada tipo de proceso, los cuales son: proceso de tiempo real, proceso interactivo, y proceso batch. A cada una de estas colas se le aplicará un algoritmo de planificación distinto, siendo estos, en orden, Round Robin, Planificación de prioridades con envejecimiento y FCFS.

A su vez, el programa tendrá una interfaz, que estará hecha con JAVA Swing, en la que el usuario podrá visualizar muy fácilmente que es lo que está sucediendo en ese momento. Para cumplir con los requerimientos, el usuario podrá, desde la interfaz: crear procesos (de manera individual o en lotes), bloquear y desbloquear procesos a voluntad y cambiar la prioridad a cualquier proceso en cualquier momento.

Abstract

In this work we will create a program following the instructions and fulfilling the requirements that were provided to us by the professors of our Operating Systems course. It will be a short term scheduler, which will be programmed in JAVA language, version 18. The scheduler will implement a multilevel queue planning algorithm with 3 queues, which will be one for each type of process, which are: real time process, interactive process, and batch process. A different planning algorithm will be applied to each of these queues, being, in order, Round Robin, Priority Planning with Aging and FCFS.

At the same time, the program will have an interface, which will be made with JAVA Swing, in which the user will be able to visualize very easily what is happening at that moment. To meet the requirements, the user will be able to, from the interface: create processes (individually or in batches), lock and unlock processes at will and change the priority of any process at any time.

Marco teórico

Proceso: informalmente, un proceso es un programa en ejecución. Hay que resaltar que un proceso es algo más que el código de un programa. Además del código, un proceso incluye también la actividad actual, que queda representada por el valor del *contador del programa*, y por los contenidos de los registros del procesador. Generalmente, un proceso influye también en la *pila* del proceso, que contiene datos temporales como los parámetros de las funciones, las direcciones de retorno y las variables locales) y *una sección de datos*, que contiene las variables globales. El proceso puede incluir, asimismo, *un cúmulo de memoria*, que es la memoria que se asigna dinámicamente al proceso en tiempo de ejecución.

Un programa, por sí mismo, no es un proceso; un programa es una entidad pasiva, un archivo que contiene una lista de. instrucciones almacenadas en disco (a menudo denominado *archivo ejecutable*), mientras que un proceso es una entidad activa, con un contador de programa que especifica la siguiente instrucción que hay que ejecutar y un conjunto de recursos asociados. Un programa se convierte en un proceso cuando se carga en memoria un archivo ejecutable. Dos técnicas habituales para cargar archivos ejecutables son: hacer doble clic sobre un icono que represente el archivo ejecutable e introducir el nombre del archivo ejecutable en la línea de comandos (como por ejemplo, *prog.exe*).

Aunque puede haber dos procesos asociados con el mismo programa, estos procesos se consideran dos secuencias de ejecución separadas. Por ejemplo, varios usuarios pueden estar ejecutando copias diferentes del programa de correo, o el mismo usuario puede invocar muchas copias del explorador web. Cada una de estas copias es un proceso distinto y, aunque las secciones de texto sean equivalentes, las secciones de datos, del cúmulo (heap) de memoria y de la pila variarán de unos procesos a otros. También es habitual que un proceso cree muchos otros procesos a medida que se ejecuta.

En nuestro programa, y para conseguir el efecto de la simplificación a la hora de la simular el planificador, nuestros procesos no tendrán las mismas características que las especificadas anteriormente, sino que tendrán las siguientes:

- ID
- Nombre
- Prioridad
- Tiempo de Creación
- Tiempo de Finalización
- Tiempo de ejecución actual
- Especificaciones de entrada y salida
- Tipo
- Estado

El tipo del proceso hace referencia a si el proceso es de *tiempo real, interactivo,* o *batch;* estando cada tipo ligado estrictamente ligado a su prioridad.

Los detalles de cada atributo y de la relación tipo-prioridad serán especificados en la sección de **procedimiento.**

Estado del proceso: A medida que se ejecuta un proceso, el mismo va cambiando de *estado*. El estado de un proceso se define, en parte, según la actividad actual de dicho proceso. Cada proceso puede estar en uno de los estados siguientes:

1. *En ejecución:* Se están ejecutando las instrucciones.
2. *Listo:* El proceso está a la espera de que le asignen a un procesador.
3. *Finalizado:* Ha terminado la ejecución del proceso.

4. *Bloqueado:* El proceso está esperando a que se produzca un suceso (como la terminación de una operación E/S).

Solo puede haber UN proceso *ejecutándose* en cualquier procesador en cada instante concreto. Sin embargo, pueden haber muchos procesos *Listos y Bloqueados*.

En los sistemas de un solo procesador, nunca habrá más de un proceso en ejecución: si hay más procesos, tendrán que esperar hasta que la CPU esté libre y se pueda asignar a otro proceso.

Colas de planificación: a medida que los procesos entran en el sistema, se colocan en una(s) *cola de trabajos* que contiene todos los procesos del sistema. Los procesos que residen en la memoria principal y están *Listos* se mantienen en una lista denominada *cola de procesos listos*. Generalmente, esta cola se almacena en forma de lista enlazada.

En nuestro caso, hay varias listas enlazadas para cada tipo de proceso, además de otra para los bloqueados y finalizados, dando un total de 5 colas.

Planificación: En un sistema de un único procesador, como el nuestro, sólo puede ejecutarse un proceso cada vez. Cualquier otro proceso tendrá que esperar hasta que la CPU quede libre y pueda volver a planificarse. El objetivo de la *multiprogramación* es tener continuamente varios procesos en ejecución, con el fin de maximizar el uso de la CPU.

La idea es bastante simple: un proceso se ejecuta hasta que tenga que esperar, normalmente porque es necesario completar alguna solicitud de E/S. En un sistema informático simple, la CPU permanece entonces inactiva y todo el tiempo de espera se desperdicia; no se realiza ningún trabajo útil. Con la multiprogramación, se intenta usar ese tiempo de forma productiva. En este caso, se mantienen varios procesos en memoria a la vez. Cuando un proceso tiene que esperar, el sistema operativo retira el uso de la CPU a ese proceso y se lo

cede a otro proceso. Este patrón se repite continuamente y cada vez que un proceso tiene que esperar, otro proceso puede hacer uso de la CPU.

La adecuada planificación de la CPU depende de una propiedad observada de los procesos: la ejecución de un proceso consta de un ciclo de ejecución en la CPU, seguido de una espera de E/S; los procesos alternan entre estos dos estados. La ejecución del proceso comienza con una ráfaga de CPU. Ésta va seguida de una ráfaga de E/S, a la cual sigue otra ráfaga de CPU, luego otra ráfaga de E/S, etc. Finalmente, la ráfaga final de CPU concluye con una solicitud al sistema para terminar la ejecución.

Planificadores: Durante su tiempo de vida, los procesos se mueven entre las diversas colas de planificación. El sistema operativo, como parte de la tarea de planificación, debe seleccionar de alguna manera los procesos que se encuentran en estas colas. El proceso de selección se realiza mediante un *planificador apropiado*.

A menudo, en un sistema de procesamiento por lotes, se envían más procesos de los que pueden ser ejecutados de forma inmediata. Estos procesos se guardan en cola en un dispositivo de almacenamiento masivo (normalmente, un disco), donde se mantienen para su posterior ejecución. El **planificador a largo plazo** o planificador de trabajos selecciona procesos de esta cola y los carga en memoria para su ejecución. El **planificador a corto plazo** o planificador de la CPU selecciona de entre los procesos que ya están listos para ser ejecutados y asigna la CPU a uno de ellos. La principal diferencia entre estos dos planificadores se encuentra en la frecuencia de ejecución, siendo el segundo ejecutado mucho más frecuentemente que el primero.

Planificador de corto plazo: cuando la CPU queda inactiva, el sistema operativo debe seleccionar uno de los procesos que se encuentran en la cola de procesos listos para ejecución. El *planificador a corto plazo* lleva a cabo esa selección del proceso. El

planificador elige uno de los procesos que están en memoria *listos* para ejecutarse y asigna la CPU a dicho proceso.

Ahora bien, la cola de procesos *listos* puede ser implementada de muchas maneras distintas, pudiendo ser desde una cola FIFO, a una cola prioritaria, un árbol, o incluso una lista enlazada, como en nuestro programa. Todo depende de los *algoritmos de planificación* utilizados.

La planificación de la CPU puede ser *no apropiativa* o *apropiativa*, siendo la primera la que deja ejecutar el proceso en CPU hasta que éste para por *bloqueo* o *finaliza*. La segunda, por otro lado, es la que utilizan los tres algoritmos que se implementan en nuestro programa, y se describe a continuación.

Planificación apropiativa: con este tipo de planificación, el planificador puede desalojar al proceso en CPU durante su ejecución y cambiarlo por otro; necesita una interrupción de reloj para poder ejecutarse en períodos regulares de tiempo (*quantum*).

Objetivos del planificador:

Justicia (fairness): que el proceso obtenga una porción de CPU "justa" o razonable.

Política: que se satisfaga un determinado criterio establecido (ej. prioridades).

Equilibrio: que todas las partes del sistema estén ocupadas haciendo algo

Algoritmos de planificación: hay una gran variedad de algoritmos de planificación, y los mismos se dividen en dos categorías fundamentales: *algoritmos apropiativos* y *no apropiativos.* Tomando en cuenta lo que ya se sabe de ambas planificaciones, uno puede deducir las características generales de cada categoría.

A continuación, algunos ejemplos de algoritmos de planificación:

Algoritmos de planificación no apropiativos:

- Planificación FCFS: es el algoritmo de planificación más simple, en el First Come First Served, como su nombre lo indica, se toma el primer proceso que llegue a la cola de listos, y los que lleguen posteriormente esperarán para entrar luego de él en su órden de llegada. Es un algoritmo simple de escribir y fácil de comprender, basta con una cola FIFO. Sin embargo, el tiempo medio de espera es a menudo bastante largo, puede provocar el efecto "convoy" (el CPU se encuentra ejecutando un proceso que consumirá mucho más tiempo que los procesos en espera).
- Planificación SJF: cumple una selección del trabajo más corto(Shortest Job First). Este algoritmo asocia con cada proceso la duración de la siguiente ráfaga de CPU del proceso. Cuando la CPU está disponible, se asigna al proceso que tiene la siguiente ráfaga de CPU más corta. Si las siguientes ráfagas de CPU de dos procesos son iguales, se usa la planificación *FCFS* para romper el empate. El algoritmo de planificación SJF es probablemente *óptimo*, en el sentido de que proporciona el tiempo medio de espera mínimo para un conjunto de procesos dado. Anteponer un proceso corto a uno largo disminuye el tiempo de espera del proceso corto en mayor medida de lo que incrementa el tiempo de espera del proceso largo. Consecuentemente, el tiempo medio de espera disminuye. Aunque el algoritmo SJF es óptimo, *no se puede implementar* en el nivel de la planificación de la CPU a corto plazo, ya que no hay forma de conocer la duración de la siguiente ráfaga de CPU.

Planificación por prioridades: el algoritmo SJF es un caso especial del *algoritmo de planificación por prioridades* general. A cada proceso se le asocia una prioridad y la CPU se asigna al proceso que tenga la prioridad más alta. Si dos procesos tienen la misma prioridad

se usa FCFS. Cuanto más larga sea la ráfaga de CPU, menor será la prioridad. Generalmente, las prioridades se indican mediante un rango de números fijo, como por ejemplo de 0 a 7, o de 0 a 1000. No existe un acuerdo general para indicar si 0 es la prioridad más baja o la más alta. La planificación por prioridades puede ser apropiativa, o no apropiativa. Un problema importante de los algoritmos de planificación por prioridades es el bloqueo indefinido o la muerte por inanición. Un proceso que está listo para ejecutarse pero está esperando a acceder a la CPU puede considerarse bloqueado; un algoritmo de planificación por prioridades puede dejar a algunos procesos de baja prioridad esperando indefinidamente. Una manera de resolver este problema es aplicando el mecanismo de *envejecimiento*. Esta técnica consiste en aumentar gradualmente la prioridad de los procesos que estén esperando en el sistema durante mucho tiempo.

Algoritmos de planificación apropiativa:

- Planificación por turnos (Round Robin): está diseñado específicamente para S.O. multiprogramación. Es similar a la planificación FCFS, pero añade la técnica de *desalojo* para *conmutar* entre procesos. En este tipo de sistema se define una pequeña unidad de tiempo, denominada *quantum*. La cola de procesos listos se trata como una cola circular. El planificador de la CPU recorre la cola de procesos listos,asignando la CPU a cada proceso durante un intervalo de tiempo de hasta 1 quantum. Para implementar la planificación por turnos, mantenemos la cola de procesos listos como una cola FIFO de procesos. Los procesos nuevos se añaden al final de la cola de procesos listos. El planificador de la CPU toma el primer proceso de la cola de procesos listos, configura un temporizador para que interrumpa pasado 1 quantum y despacha el proceso. Puede ocurrir una de dos cosas. El proceso puede tener una ráfaga de CPU cuya duración sea menor que 1 quantum; en este caso, el propio

proceso liberará voluntariamente la CPU. El planificador continuará entonces con el siguiente proceso de la cola de procesos listos. En caso contrario, si la ráfaga de CPU del proceso actualmente en ejecución tiene una duración mayor que 1 quantum, se producirá un fin de cuenta del temporizador y éste enviará una interrupción al sistema operativo;.entonces se ejecutará un cambio de contexto y el proceso se colocará al *final* de la cola de procesos listos. El planificador de la CPU seleccionará a continuación el siguiente proceso de la cola de procesos listos. El rendimiento del algoritmo Round Robin depende enormemente de su quantum. Si es muy largo, el algoritmo se asemeja demasiado al FCFS, y si es muy pequeño, los procesos pueden demorar demasiado en terminarse al estar conmutando entre ellos con demasiada frecuencia.

- Planificación mediante colas multinivel: otra clase de algoritmos de planificación es la que se ha desarrollado para aquellas situaciones en las que los procesos pueden clasificarse fácilmente en grupos diferentes(por ej: interactivos, lotes, etc). Estos grupos de procesos tienen requisitos diferentes de tiempo de respuesta y distinta prioridad, por tanto, pueden tener distintas necesidades de planificación. Un algoritmo de planificación mediante colas multinivel divide la cola de procesos listos en varias colas distintas. Los procesos se asignan permanentemente a una cola, generalmente en función de alguna propiedad del proceso, como por ejemplo el tamaño de memoria, la prioridad del proceso o el tipo de proceso. *Cada cola tiene su propio algoritmo de planificación.* Cada cola tiene prioridad absoluta sobre las colas de prioridad más baja. Si se está ejecutando un proceso de una cola de baja prioridad y llega uno a una cola de alta prioridad, el proceso será desalojado y pasará a ejecutarse el recién llegado.

En nuestro planificador, se utilizará una planificación mediante colas multinivel, en la que habrá 3 colas de procesos listos. La configuración de las mismas y sus detalles serán especificados en la sección de **Procedimiento.**

Hilos: Un hilo es una unidad básica de utilización de la CPU; comprende un ID de hilo, un contador de programa, un conjunto de registros y una pila. Comparte con otros hilos que pertenecen al mismo proceso la sección de código, la sección de datos y otros recursos del sistema operativo, como los archivos abiertos y las señales. Un proceso tradicional (o proceso pesado) tiene un solo hilo de control y se le llama proceso *monohilo*. Si un proceso tiene, por el contrario, múltiples hilos de control, puede realizar más de una tarea a la vez y pasa a ser un proceso *multihilo.*

Ventajas de usar hilos:

- Capacidad de respuesta: El uso de múltiples hilos en una aplicación interactiva permite que el programa continúe ejecutándose incluso aunque parte de él esté bloqueado o realizando una operación muy larga, lo que incrementa la capacidad de respuesta al usuario.
- Compartición de recursos: Por definición, los hilos comparten la memoria y los recursos del proceso al que pertenecen. La ventaja de compartir el código y los datos es que permite que una aplicación tenga varios hilos de actividad diferentes dentro del mismo espacio de direcciones.
- Economía: La asignación de memoria y recursos para la creación de procesos es costosa. Dado que los hilos comparten recursos del proceso al que pertenecen, es más económico crear y realizar cambios de contexto entre unas y otras.
- Utilización sobre arquitecturas multiprocesador: Las ventajas de usar configuraciones multihilo pueden verse incrementadas significativamente en un sistema con

arquitectura multiprocesador, donde los hilos pueden ejecutarse en paralelo en los diferentes procesadores.

NetBeans: entorno de desarrollo integrado (IDE) para cualquier lenguaje de programación que permite desarrollar y modificar aplicaciones. Es el IDE utilizado para este trabajo. Windows, MacOS, Linux y Solaris permiten ejecutar NetBeans en su sistema. Está escrito en Java y desarrollado por Apache Software Foundation y Oracle Corporation. Es de código abierto y permite escribir aplicaciones de escritorio Java en el IDE. Cualquier desarrollo de aplicaciones web, empresariales, de escritorio y móviles puede ser desarrollado en NetBeans IDE. Además de Java, que es el principal lenguaje para el que está hecho este IDE, soporta C, C++, XML, HTML, PHP, Groovy y lenguajes de scripting.

Además, es un framework que simplifica el desarrollo de aplicaciones para Java Swing. El paquete de NetBeans IDE para Java SE contiene lo que se necesita para empezar a desarrollar plugins y aplicaciones basadas en la plataforma NetBeans; no se requiere un SDK adicional.

La plataforma ofrece servicios reusables comunes para las aplicaciones de escritorio, permitiendo a los desarrolladores centrarse en la lógica de sus aplicaciones. NetBeans IDE es libre, código abierto, multiplataforma con soporte integrado para el lenguaje de programación Java.

La versión de NetBeans IDE usada en este caso es la 13.

JAVA: Java es un lenguaje de programación de alto nivel, basado en clases y orientado a objetos, diseñado para tener el menor número posible de dependencias de implementación y es el que utilizamos en esta ocasión para hacer el planificador. Es un lenguaje de programación de propósito general que permite a los programadores escribir una vez y ejecutar en cualquier lugar, lo que significa que el código Java compilado puede ejecutarse en

todas las plataformas que soportan Java sin necesidad de recompilar. Las aplicaciones Java suelen compilar en código de bytes que puede ejecutarse en cualquier máquina virtual Java (JVM), independientemente de la arquitectura informática subyacente. En 2019, Java era uno de los lenguajes de programación más populares en uso según GitHub, particularmente para aplicaciones web cliente-servidor, con un informe de 9 millones de desarrolladores.

Java fue desarrollado originalmente por James Gosling en Sun Microsystems y lanzado en mayo de 1995 como un componente central de la plataforma Java de Sun Microsystems.

En nuestro trabajo, se utiliza JAVA en su versión 18.

JAVA Swing: Swing es una biblioteca gráfica para Java. Incluye widgets para interfaz gráfica de usuario tales como cajas de texto, botones, listas desplegables y tablas. Es la interfaz en la que está basada nuestra aplicación.

Es un framework para desarrollar interfaces gráficas para Java con independencia de la plataforma. Sigue un simple modelo de programación por hilos, y posee las siguientes características principales:

- Independencia de plataforma.
- Extensibilidad: es una arquitectura altamente particionada: los usuarios pueden proveer sus propias implementaciones modificadas para sobrescribir las implementaciones por defecto. Se puede extender clases existentes previendo alternativas de implementación para elementos esenciales.
- Personalizable: dado el modelo de representación programático del framework de Swing, el control permite representar diferentes estilos de apariencia (desde apariencia MacOS hasta apariencia Windows XP, pasando por apariencia GTK+, IBM UNIX o HP UX, entre otros). Además, los usuarios pueden proveer su propia

implementación de apariencia, que permitirá cambios uniformes en la apariencia existente en las aplicaciones Swing sin efectuar ningún cambio al código de aplicación.

Procedimiento

1) Elección del planificador a desarrollar y características del mismo

Lo primero que se hizo antes de siquiera pensar en programar el planificador fue tener una discusión entre los integrantes del equipo con el objetivo de decidir que algoritmo de planificación íbamos a utilizar, y no solo eso, sino también idear sobre que S.O. iba a "correr" nuestro planificador; que tipo de procesos pueden haber, que características tienen los mismos, etc.

Para empezar, definimos nuestra implementación de *procesos*. Para nosotros, los procesos son un objeto con las siguientes propiedades:

- ID: entero, único para cada proceso. Su asignación es automática al momento de la creación del proceso.
- Nombre: String, pueden haber varios procesos con el mismo nombre.
- Prioridad: entero, su rango es de 1 a 99, siendo 1 la prioridad más alta y 99 la más baja.
- Duración: entero, cantidad de segundos que requiere el programa para finalizar.
- Tiempo de Creación: tipoTiempoLocal, se le asigna el tiempo de creación del proceso automáticamente una vez se ejecuta su constructor. El formato es hh:mm:ss:ms
- Tiempo de Finalización: tipoTiempoLocal, variable donde irá guardado el tiempo de finalización de un proceso.
- Tiempo de ejecución actual: entero, marca cuantos segundos va ejecutándose el proceso actual. Se utilizará para verificar si en el segundo actual se tiene que ejecutar una interrupción por Entrada y Salida.
- Especificaciones de entrada y salida: Diccionario(HashMap) de tipo <Entero, Entero>, en la primera entrada se encuentra el segundo donde el proceso realizará la

interrupción por entrada y salida, y en la segunda entrada, cuantos segundos durará esa interrupción.

- Tipo: enum, se utilizará para marcar el tipo de proceso. El enum consta con 3 valores, uno para cada tipo existente de proceso, los cuales son *procesos de tiempo real*, *procesos interactivos,* o *procesos batch*.
- Estado: enum, aquí se guardará el estado actual del proceso. Los valores posibles son: *LISTO, EJECUCIÓN, BLOQUEADO, y FINALIZADO.*

Existe una relación entre las propiedades Tipo - Prioridad que cumple las siguientes características:

- Si un proceso tiene prioridad 1 a 20, ese proceso será de Tipo Tiempo Real
- Si un proceso tiene prioridad 21 a 70, ese proceso será de Tipo Interactivo
- Si un proceso tiene prioridad 71 a 99, ese proceso será de Tipo Batch

En base a estos procesos, se llegó a la conclusión de construir un planificador a corto plazo (como lo indica la consigna) que implemente un algoritmo de planificación de *colas multinivel* con 3 colas de procesos *listos* (habrá programación concurrente), en donde cada una de ellas corresponda a un tipo de proceso(Tiempo Real, Interactivos, Batch), con 2 colas más para los procesos *bloqueados* y *finalizados*. A su vez, sobre cada una de las 3 colas de procesos listos, corre un algoritmo de planificación distinto, siendo la configuración de estos como se describe a continuación:

1. Cola de procesos de Tiempo Real: sobre esta cola, corre un algoritmo Round Robin o de planificación por turnos, con un quantum de 1 segundo para cada proceso. Se decidió este algoritmo para este tipo de procesos debido a que, desde nuestro punto de vista, cualquier proceso de Tiempo Real (prioridad muy alta) necesita atención inmediata y todos deberían ser igual de importantes. Con este algoritmo de

planificación, nos aseguramos que todos los procesos de esta categoría tengan un acceso equitativo a tiempo de CPU.

2. Cola de procesos interactivos: para realizar la planificación de los procesos interactivos, se utilizará un algoritmo de planificación por prioridades, el cual contará con un mecanismo de envejecimiento para asegurarnos de que al CPU se le asignen estos procesos de la manera más equitativa posible. Otro detalle que vale la pena destacar de la implementación de este algoritmo es que el mismo es apropiativo, y que tiene un quantum de 2 segundos para cada proceso actual. Cada vez que se termina el quantum, si el proceso no se terminó, se implementa el mecanismo de envejecimiento disminuyendo en 1 su prioridad actual de forma permanente. Este algoritmo fue pensado y elegido en base a nuestro deseo de querer que todos los procesos tengan el tiempo de CPU que requieran de manera equitativa, sin perder la importancia de la prioridad específica de cada uno de ellos.
3. Cola de procesos batch: para este tipo de procesos, se utilizará un sencillo algoritmo FCFS, con la excepción de que es apropiativo, con un quantum de 4 segundos. Se diferencia con el algoritmo Round Robin en que aquí, los procesos se ordenan por su Tiempo de Creación (tiempo de llegada). Decidimos no crear un algoritmo de planificación complejo para los procesos batch debido a la poca importancia que tienen dentro del Sistema Operativo comparado con los otros procesos.

Un detalle no menor es nuestra implementación del CPU, que, para nosotros, será un objeto que tenga como único atributo un proceso, el proceso *en ejecución* actual y un método que sea el que simule el procesamiento.

Por último, la interfaz gráfica será hecha con JAVA Swing, como se indicó en el marcó teórico, desde ahí, el usuario podrá:

- Cargar conjuntos de procesos
- Cargar procesos individualmente
- Observar en tiempo real el proceso de Planificación, el proceso que se está ejecutando actualmente, y todos los demás con todos los datos pertinentes(ID, nombre, tipo, estado, prioridad, tiempo de creación).
- Bloquear manualmente cualquier proceso
- Cambiar la prioridad de cualquier proceso de manera manual.

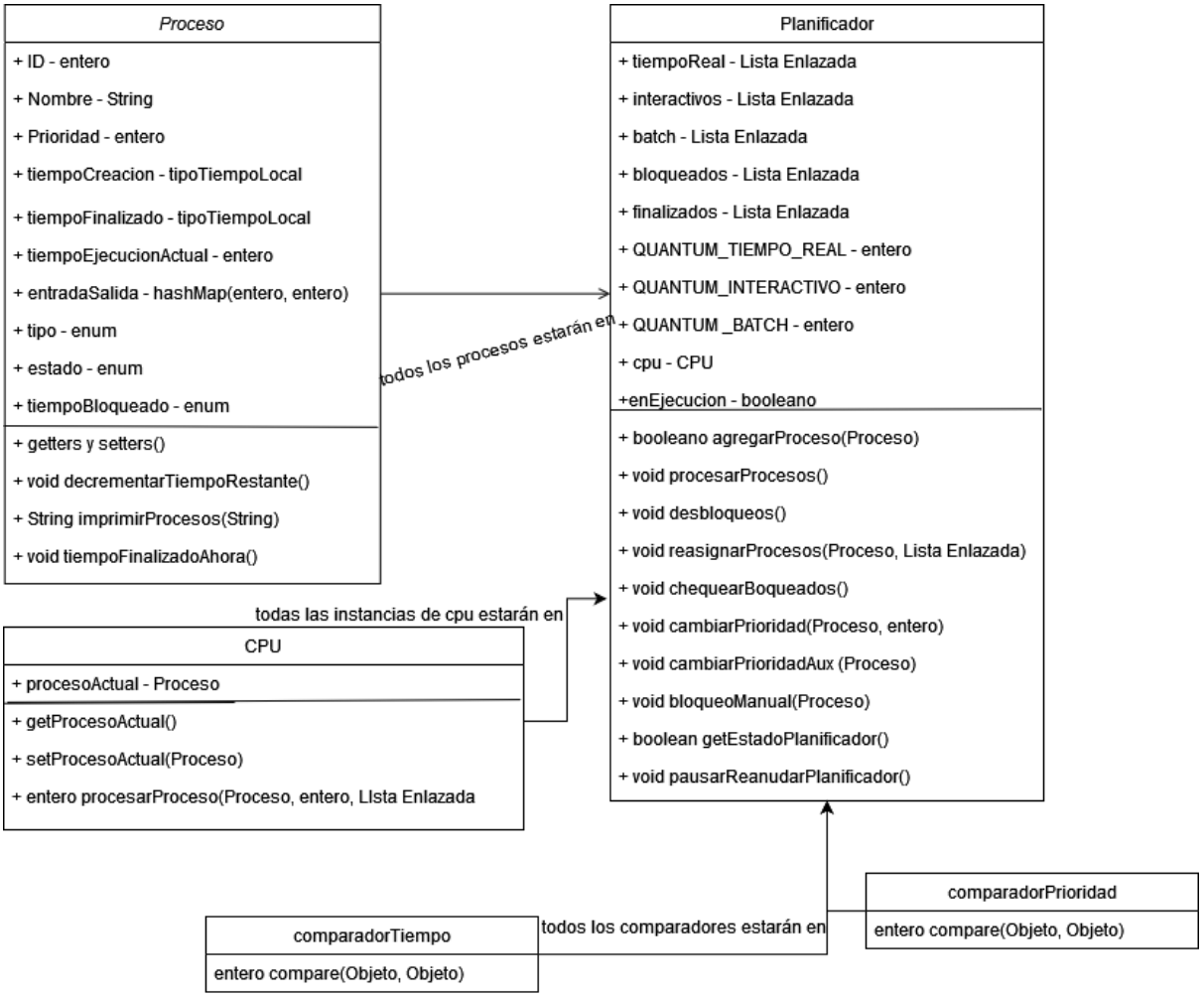
A continuación se especificará el funcionamiento general (básico) del Planificador una vez que se ejecuta:

- Se requerirá cargar al menos un proceso para que el planificador inicie
- Una vez cargados los procesos, los mismos se distribuirán automáticamente a su lista correspondiente según su tipo.
- El Planificador verificará si hay elementos disponibles en cada una de las listas con este orden: tiempo real, interactivos, batch
- El Planificador elegirá el proceso que corresponda y lo asignará como proceso actual (se pondrá en EJECUCIÓN)
- El CPU procesará el proceso actual hasta que
    - se termine
    - se bloquee
    - finalice el quantum actual
- Si el proceso finalizó, su estado será ahora FINALIZADO y será trasladado a la lista de finalizados. Se guardará su tiempo de finalización.
- Si el proceso se bloqueó, su estado será ahora BLOQUEADO y será trasladado a la lista de bloqueados.

2) Creación del Planificador

A partir de lo obtenido en la sección anterior, es hora de utilizarlo para la creación del Planificador per sé.

En primer lugar, se realizó un diagrama UML de clases en el que, para cada clase, se especifican las variables y métodos que se necesitarán para el correcto funcionamiento del planificador. El mismo, es el siguiente:



nota: el formato de los métodos es el siguiente: + *devuelve* nombreDelMetodo *(parámetros)*

A continuación, se detalla todo el proceso que conlleva la planificación de procesos en nuestro programa, desde la creación de un proceso hasta la finalización de todos los procesos.

Se describirán en pseudocódigo todos los métodos utilizados en el proceso, así como también se incluirá una descripción de qué es lo que hacen y una justificación de su existencia.

1. Lo primero que se debe hacer es crear procesos, y nuestro programa admite dos maneras de hacerlo, tal y como se describe en el **manual de usuario**. Ambas maneras van a utilizar, tarde o temprano, el mismo constructor de objetos Proceso. El mismo es el que se especifica a continuación:

`Proceso(String nombre, entero prioridad, entero duracion, diccionario<entero, entero> entradaSalida)`

Si se decide crear un proceso con la prioridad incorrecta, saltará error. Aquí se observa de primera manera la relación prioridad - Tipo.

```
Proceso(String nombre, entero prioridad, entero duracion, diccionario<entero, entero> entradaSalida)
COM
this.id = contador + 1
this.nombre = nombre
SI(prioridad < 1 OR prioridad > 99) ENTONCES
    Imprimir error
    SALIR
SINO SI(prioridad >= 1 AND prioridad <= 20) ENTONCES
    this.prioridad = prioridad
    this.tipo = TIEMPOREAL
SINO SI(prioridad >= 21 AND prioridad <= 70) ENTONCES
    this.prioridad = prioridad
    this.tipo = INTERACTIVO
SINO SI (prioridad >= 71 AND prioridad <= 99) ENTONCES
    this.prioridad = prioridad
    this.tipo = BATCH
FINSI
this.tiempoRestante = duracion
this.entradaSalida = entradaSalida
this.estado = LISTO
FIN
```

Como se puede observar, se evita claramente que la prioridad deseada no sea menor a 1 o mayor a 99; en cualquiera de los dos casos, el programa NO permitirá al usuario continuar diciéndole que la prioridad debe situarse entre ese par de valores.

2. Luego de creados él/los procesos, se crea una instancia del objeto Planificador, dentro del cual ya se crea una instancia del objeto CPU (ambos constructores vacíos, si se quiere ver que variables tiene cada clase, observar el diagrama UML). Una vez se tiene el planificador instanciado, se llama al método *agregarProcesos*, el cuál se

encarga de distribuir los procesos a las colas correspondientes. El mismo se detalla a continuación:

```
agregarProceso(Proceso proc) : devuelve booleano
COM
SI(proc.tipo = nulo) ENTONCES
    DEVUELVE falso
SINO
    SI(proc.tipo = INTERACTIVO) ENTONCES
        totalInteractivos <- totalInteractivos + 1
        Interfaz.imprimir("El proceso … se agregó a la cola de interactivos")
        interactivos.añadir(proc)
        interactivos.ordenarPor(prioridad)
        SALIR
    SI(proc.tipo = BATCH) ENTONCES
        totalBatch <- totalBatch + 1
        Interfaz.imprimir("El proceso … se agregó a la cola de batch")
        batch.añadir(proc)
        SALIR
    SI(proc.tipo = TIEMPOREAL) ENTONCES
        totalTiempoReal <- totalTiempoReal + 1
        Interfaz.imprimir("El proceso … se agregó a la cola de tiempo real")
        tiempoReal.añadir(proc)
        SALIR
    SINO
        DEVUELVE falso
    FINSI
DEVUELVE verdadero
FIN
```

Este método es muy sencillo y no conlleva mayor análisis, pero sí se puede resaltar que la cola de interactivos recibe una línea adicional a sus contrapartes, pues cada vez que se modifique su cola, es necesario volver a ordenarlas por prioridad.

3. Luego, se procede a arrancar el proceso de planificación per sé. Esto se hace llamando al método del Planificador *procesarProcesos*, pero como este método es muy extenso y contiene muchos métodos auxiliares, se irá construyendo de a poco, analizando cada paso en su construcción.

Método completo:

```
procesarProcesos() : devuelve void
COM
enEjecucion <- verdadero
MIENTRAS (enEjecucion AND (tiempoReal <> vacía OR interactivos <> vacía OR batch <> vacía)
    SI(tiempoReal <> vacía) ENTONCES
        procesoActual <- tiempoReal.agarrarPrimero()
        Interfaz.imprimir("El proceso … ingresó al CPU")
        cpu.procesarProceso(procesoActual, QUANTUM_TIEMPO_REAL, bloqueados)
        SI(procesoActual.estado = EJECUCION) ENTONCES
            tiempoReal.remover(procesoActual)
            tiempoReal.añadirAlFinal(procesoActual)
            procesoActual.estado <- LISTO
        FINSI
        reasignarProcesos(procesoActual, tiempoReal)
        chequearBloqueados()
        CONTINUAR
    SINO SI(interactivos <> vacía)
        procesoActual <- interactivos.agarrarPrimero()
        Interfaz.imprimir("El proceso … ingresó al CPU")
        cpu.procesarProceso(procesoActual, QUANTUM_INTERACTIVOS, bloqueados)
        reasignarProcesos(procesoActual, interactivos)
        chequearBloqueados()
        SI(procesoActual.estado = EJECUCION) ENTONCES
            procesoActual.prioridad = procesoActual.prioridad + 1
    interactivos.ordenarPor(prioridad)
    SI(procesoActual.estado <> FINALIZADO OR BLOQUEADO) ENTONCES
        SI(interactivos.agarrarPrimero() <> procesoActual)
            procesoActual.estado <- LISTO
            FINSI
        FINSI
        CONTINUAR
        SINO SI(batch <> vacía)
            procesoActual <- batch.agarrarPrimero()
            Interfaz.imprimir("El proceso … ingresó al CPU")
            cpu.procesarProceso(procesoActual, QUANTUM_BATCH, bloquSeados)
            reasignarProcesos(procesoActual, batch)
```

```
            chequearBloqueados()
            batch.ordenarPor(tiempoDeCreacion)
    SI(procesoActual.estado <> FINALIZADO OR BLOQUEADO) ENTONCES
        SI(batch.agarrarPrimero() <> procesoActual)
            procesoActual.estado <- LISTO
        FINSI
    FINSI
    CONTINUAR
FINMIENTRAS
enEjecucion <- falso
desbloqueos()
FIN
```

4. Primero que nada, se marca el valor *enEjecucion* apenas arranca el método, esto marca que efectivamente el planificador está en funcionamiento.

5. Siguiendo, es deseable que este método se corra SOLO mientras haya procesos en las colas, por lo casi todas las acciones del método estarán dentro un bucle MIENTRAS que chequee constantemente esa condición.
6. Luego, se comienza por verificar la condición de vacío o no en la cola de tiempo real, la de más alta prioridad.
    a. Si lo está, el algoritmo seguirá de largo y chequeará la siguiente cola.
    b. Si no lo está, se llamará a un objeto CPU para que tome al primer proceso de la cola como *proceso actual* y se ejecutará el método de CPU *procesarProceso*, que está descrito a continuación:

```
procesarProceso(Proceso proc, entero quantum, ListaEnlazadaDeProcesos bloqueados) : devuelve entero
COM
MIENTRAS (quantum > 0) HACER
    SI(proc.estado = LISTO OR proc.estado = EJECUCION) ENTONCES
    proc.estado <- EJECUCION
    proc.decrementarTiempoRestante()
    PARA CADA Proceso en bloqueados HACER
        Proceso.tiempoBloqueado = Proceso.tiempoBloqueado - 1
    FIN PARA CADA
    SI(proc.tiempoRestante <= 0) ENTONCES
        proc.estado = FINALIZADO
        DEVOLVER quantum
    SI(proc.entradaSalida contiene la clave proc.tiempoEjecucionActual) ENTONCES
        proc.estado = BLOQUEADO
        aux <- proc.entradaSalida.obtenerDatoPorClave(proc.tiempoEjecucionActual)
        proc.tiempoBloqueado <- aux
        DEVOLVER quantum
    FINSI
FIN MIENTRAS
DEVOLVER quantum
FIN
```

El método se encarga de “procesar” el proceso pasado por parámetro durante el quantum pasado por parámetro, de detectar si el proceso debe ser bloqueado (y setear cuánto tiempo debe de estar en BLOQUEADO) o si el mismo finalizó. Además, se encarga de decrementar *quantum* unidades de tiempo restante de bloqueado a los procesos que se encuentran bloqueados(por eso se le pasa la lista por parámetro). Esto es así debido a que es el CPU el que simula el paso del tiempo en nuestro programa.

c. Ahora mismo, luego de terminado el método descrito anteriormente, hay tres opciones
    i. El proceso finalizó
    ii. El proceso entró en estado bloqueado
    iii. El proceso no ha terminado ni se bloqueó.
d. En caso de que no haya ni terminado ni se haya bloqueado, eso quiere decir que el programa debe recolocarse al final de la lista, para cumplir con el algoritmo Round Robin.
e. Para cubrir las otras dos opciones se llama al método *reasignarProcesos*, que se encarga de chequear cada uno de esos estados y de realizar las acciones pertinentes a cada estado. Si se bloqueó, se irá a la cola de bloqueados, y se removerá de la cola de procesos tiempo real. Si finalizó, lo mismo, pero yéndose a la lista de finalizados.
f. Se llamará al método *chequearBloqueados,* que, como el nombre lo dice, se fijará en todos los procesos en la cola de bloqueados. Por cada uno de ellos, chequeará que tipo de proceso es, y si ya está listo para desbloquear.
    i. Si ya está listo para desbloquear (tiempoBloqueo <= 0), entonces lo removerá de la lista de bloqueados, lo añadirá a la cola correspondiente, y lo pondrá en estado LISTO.
    ii. Si no, no hace nada y se fija si es del siguiente tipo de proceso.
g. Vuelve al bucle MIENTRAS

Métodos *reasignarProcesos y chequearBloqueados:*

```
reasignarProcesos(Proceso proc, ListaEnlazadaDeProcesos listaProcesos) : devuelve void
COM
SI(proc.estado = FINALIZADO) ENTONCES
    listaProcesos.remover(proc)
    proc.tiempoFinalizado <- ahora
    Interfaz.imprimir("Proceso ... finalizó")
    finalizados.añadir(proc)
SINO SI(proc.estado = BLOQUEADO) ENTONCES
    listaProcesos.remover(proc)
    Interfaz.imprimir("Proceso ... ingresó a la cola de bloqueados")
    bloqueados.añadir(proc)
FINSI
FIN
```

```
chequearBloqueados() : devuelve void
COM
SI(bloqueados <> vacía)
    PARA CADA procesoBloqueado EN bloqueados
        SI(procesoBloqueado.tipo = TIEMPOREAL) ENTONCES
            SI(procesoBloqueado.tiempoBloqueado <= 0) ENTONCES
                bloqueados.remover(proceso)
                procesoBloqueado.estado <- LISTO
                tiempoReal.añadir(procesoBloqueado)
            FINSI
        SI(procesoBloqueado.tipo = INTERACTIVO) ENTONCES
            SI(procesoBloqueado.tiempoBloqueado <= 0) ENTONCES
                bloqueados.remover(proceso)
                procesoBloqueado.estado <- LISTO
                interactivos.añadir(procesoBloqueado)
            FINSI
        SI(procesoBloqueado.tipo = BATCH) ENTONCES
            SI(procesoBloqueado.tiempoBloqueado <= 0) ENTONCES
                bloqueados.remover(proceso)
                procesoBloqueado.estado <- LISTO
                batch.añadir(procesoBloqueado)
            FINSI
        FINSI
    FINPARACADA
FINSI
FIN
```

7. Continúa comprobando la condición de esVacío con las otras colas. De no serlo, se repite el mismo en las otras colas que la cola de tiempo real, con la excepción de que para las colas de interactivos y batch, una vez se terminen de realizar todas las posibles modificaciones a las colas, las mismas se vuelven a reordenar (interactivos - por prioridad, batch - por tiempo de creación). Es gracias a esta modificación que se

debe de agregar una comprobación que le pregunte a las listas "¿sigue siendo el *procesoActual* el primero en la lista?"(¿lo debo procesar de nuevo?):

a. Si no es así, se debe quitar el estado "en ejecución" al mismo, y ponerlo en LISTO.

Además, cabe aclarar que en la cola de interactivos, una vez terminado el método de CPU *procesarProceso* con el proceso actual, si este todavía no se ha terminado ni se ha bloqueado, entonces se le aumentará en uno su prioridad(en el punto de vista de nuestro programa, la hará más baja ya que mientras más alto el número más baja es la prioridad en sí).

8. Una vez se sale del bucle MIENTRAS, se pone el booleano *enEjecucion* en falso.
9. Una consecuencia directa de nuestro modo de implementar el planificador es que, si en un caso hipotético, un proceso se queda en la lista de bloqueados mientras que las otras tres colas estén vacías, el mismo se quedará atascado ahí para siempre. Así que para solucionarlo, se creó el método *desbloqueos*, que simulará el paso del tiempo igual que el CPU lo hacía por nosotros cuando se estaba procesando un proceso.

   El método hace lo descrito a continuación:

```
desbloqueos() : devuelve void
COM
MIENTRAS(bloqueados <> vacia)
    chequearBloqueados()
    PARA CADA procesoBloqueado EN bloqueados
        SI(procesoBloqueado.tiempoBloqueado >0) ENTONCES
            procesoBloqueado.tiempoBloqueado <- procesoBloqueado.tiempoBloqueado - 1
        SINO
            enEjecucion <- true
    procesarProcesos()
FIN
```

   En caso de haber quedado procesos bloqueados, se desbloquean y se llama al método de *procesarProcesos* de nuevo.

En este punto, cuando oficialmente no quedan más procesos en ninguna cola, se da por finalizado el proceso de Planificación. Sin embargo, todavía quedan por desarrollar dos características del planificador que la consigna nos solicitaba.

1. El método *bloqueoManual:* como la letra nos decía, se debe de implementar una funcionalidad que permita que el usuario tenga la capacidad de bloquear a su antojo cualquier proceso en cualquier momento. En nuestro programa, esa funcionalidad vino en forma del método *bloqueoManual*. El mismo será descrito a continuación:

```
bloqueoManual(Proceso proc, booleano bloquear) : devuelve void
COM
SI(bloquear = verdadero)
    proc.estado <- BLOQUEADO
    SI(tiempoReal.contiene(proc)) ENTONCES
        tiempoReal.remover(proc)
    SI(interactivos.contiene(proc)) ENTONCES
        interactivos.remover(proc)
    SI(batch.contiene(proc)) ENTONCES
        batch.remover(proc)
    FINSI
    proc.tiempoBloqueado = INFINITO
    SI(bloqueados NO contiene (proc)) ENTONCES
        bloqueados.añadir(proc)
    FINSI
SINO
    proc.tiempBloqueado = 0
    chequearBloqueados()
FINSI
FIN
```

Como se puede apreciar, es un método muy sencillo, al que se le pasa por parámetro el proceso que se desea bloquear(o desbloquear, dependiendo del valor *bloquear* que se le pase por parámetro). Lo que hace es chequear todas las colas en busca de ese proceso, cuando lo encuentre, lo saca de ahí y lo mete a la cola de bloqueados, seteando su tiempo de bloqueado en infinito, así no se verá afectado por el constante decremento del tiempo de bloqueo del método de CPU *procesarProceso*.

En caso de pasarle un booleano falso como parámetro, el proceso sencillamente pondrá su tiempo de bloqueo en 0 y llamará al método encargado de los bloqueados que ya se vió anteriormente, *chequearBloqueados()*.

2. Él método *cambiarPrioridad:* según la letra, el usuario debe ser capaz de cambiar la prioridad de un proceso en cualquier momento si así lo desea, por lo tanto, se creó este método que cumple con lo que se requiere.

El mismo es muy parecido a lo que se vió en el constructor de un Proceso, en el sentido de las limitaciones en el número *prioridad*. En este caso, el método se diferencia del constructor en el sentido de que no se puede cambiar la prioridad de un proceso que no es de tiempo real a una prioridad que sea acorde a ese tipo de procesos. De igual manera, no se puede modificar un proceso de tiempo real para que ya no lo sea. Esto fue ideado por nosotros teniendo en mente la noción de que no tiene sentido que se le permita al usuario situar cualquier proceso, por más útil/importante (o no) que sea por delante de los procesos de tiempo real que el S.O. situó ahí.

3) Generalidades de la interfaz

Tal y como se describe en la letra, el programa está implementado con una interfaz gráfica. La misma fue creada con JAVA Swing, como se mencionó en el **marco teórico**.

Debido a que el foco de este trabajo es la lógica del programa y no la interfaz, no entraremos en detalles acerca de la creación de la misma. Sin embargo, queríamos mencionar algunos detalles que pueden ser importantes:

1. Casi todos los métodos descritos anteriormente están descritos como eventos de presionar *botones* en la interfaz.
2. Todos los detalles acerca de las funcionalidades de la interfaz y sobre cómo ejecutar el programa se encuentran en el documento **Manual de usuario** incluído en este trabajo.
3. El programa será ejecutable desde un archivo .jar.

Conclusión

Una vez finalizada la tarea, nos quedará como resultado un programa ejecutable .jar que, al ejecutarse, correrá todo el código que corresponde al de un planificador de corto plazo que implementa colas multinivel, y se presentará en pantalla la interfaz del mismo, donde el usuario podrá cargar procesos (de manera individual o por lotes), visualizar el trabajo actual del planificador a través de tablas de procesos, en donde se verán todos los procesos que están listos o en ejecución, junto con su ID, su nombre, su tiempo de creación y su tiempo restante. Por otro lado, también se verá una lista de bloqueados, en donde se podrán ver los procesos bloqueados ya sea por operaciones de entrada y salida o los bloqueados manualmente por el usuario y donde podrá también desbloquearlos a voluntad. A efectos de cumplir con los requerimientos de la tarea, el usuario podrá cambiar la prioridad de un proceso a voluntad. Además, se incluyó un log de eventos para que el usuario pueda ver en cada momento qué es exactamente lo que está haciendo el planificador.

Referencias

Pedro Cabalar (s.f.). Sistemas Operativos, TEMA III: PROCESOS. Universidade da Coruña. Recuperado de: https://www.dc.fi.udc.es/~so-grado/SO-Procesos-planif.pdf

Andrew Tanenbaum (2009). *Sistemas operativos modernos. Tercera edición.* Pearson Educación. Ciudad de México, México.

Abraham Silberschatz. (2006). *Fundamentos de sistemas operativos. 7ma edición.* McGraw-Hill / Interamericana de España, S.A.U. Madrid, España.

NetBeans. (s.f.). Recuperado de https://es.wikipedia.org/wiki/NetBeans

What is Java and what is it used for?. (s.f.). Recuperado de https://codeinstitute.net/global/blog/what-is-java/

Personal del curso de Algoritmos y Estructuras de Datos. (s.f.). *Apuntes sobre seudocódigo.* UCU. Montevideo, Uruguay. Recuperado de: https://webasignatura.ucu.edu.uy/pluginfile.php/581725/mod_folder/content/0/Apuntes%20sobre%20seudoc%C3%B3digo.pdf?forcedownload=1