Inlämningsuppgift 3: The great kernel debacle

Ken och Dennis utvecklar ett operativsystem, men är inte helt överens om hur vissa delar av operativsystemet ska fungera. Mer specifikt bråkar de om vilken metod som ska användas för att allokera minne.

Material

- Kapitel 7 och 12 i *The Elements of Computing Systems*, specifikt de delar som avhandlar minneshantering.
- Kapitel 7 i *Operating Systems Internals and Design Principles* för deluppgift 3.
- Uppgiftskod på https://github.com/koddas/heapsim.
- Java version 5 eller nyare.

Instruktioner

För att bli godkänd på inlämningen måste både deluppgift 1 och 2 vara godkända. För betyget väl godkänd fordras att även deluppgift 3 lämnas in och godkännas. Alla deluppgifter lämnas in samtidigt på It's learning. Låt varje deluppgift vara en egen zip-fil. För samtliga uppgifter gäller att du ska skicka in de .java-filer som du skriver.
Uppgifterna får lösas och redovisas i grupper om maximalt två studenter.

Exempelkoden

På Github ligger den kod som ni utgår från. Den är uppdelad i två paket, *memory* och *batches*.
Paketet *memory* innehåller tre grundläggande filer som du kan använda som de är, samt tre filer kopplade till de tre deluppgifterna. Dessa filer är:

- Pointer.java, en klass som representerar en pekare. Med hjälp av denna kan du läsa från och skriva till minnet. **Den här filen ska du inte ändra något i.**
- RawMemory.java, en klass som representerar ett minnesområde. Varje minnescell, eller ord, kan hålla ett heltal (integer). **Den här filen ska du inte ändra i eller använda direkt**. Dess funktionalitet kommer du åt via *Pointer*.
- **Memory.java, en abstrakt klass som definierar metoderna *alloc()*, *release()* och *printLayout()*.** Den är superklass till de tre filer som du faktiskt ska arbeta med, nämligen
    - **FirstFit.java, som används i deluppgifter 1 och 3. Använd klassen Pointer och Javas standardbibliotek för att implementera metoderna i filen.**

- **BestFit.java, som används i deluppgift 2. Använd klassen Pointer och Javas standardbibliotek för att implementera metoderna i filen.**
- Buddy.java, som används i deluppgift 3. Använd klassen Pointer och Javas standardbibliotek för att implementera metoderna i filen.

I *batches* finns två batch-program som kan köras för att testa dina implementationer. Programmen gör en serie allokeringar och avallokeringar som du kan följa med hjälp av dina implementationer av *printLayout()* i klasserna nämnda ovan. Kör dem från kommandoraden med *java ClassName* eller direkt från din IDE. För båda batch-programmen gäller att du anger vilken fil du vill testa på rad 25.

Deluppgift 1 (G)
Ken förespråkar tekniken *first-fit* för att minneshanteringen ska vara snabb. Hjälp honom att implementera *alloc()* och *release()* med *first-fit*. Utgå från filen FirstFit.java.
Implementera även *printLayout()* i samma fil för att kunna visualisera minnet.

Deluppgift 2 (G)
Dennis tror att *best-fit* är bättre i längden, eftersom det sparar minne. Hjälp honom att implementera *alloc()* och *release()* med *best-fit*. Utgå från filen BestFit.java.
Implementera även *printLayout()* i samma fil för att kunna visualisera minnet.

Deluppgift 3 (VG)
Ken och Dennis inser båda två att både *first-fit* och *best-fit* fungerar, men att de tenderar att lämna en massa utspritt, fragmenterat, fritt utrymme som inte kan återanvändas. Därför vill de testa två metoder för att lösa problematiken:

- Kompaktering, eller defragmentering, av det lediga minnet. Din uppgift är att skriva en metod *void compact()* i FirstFit.java. Metoden ska ta bort alla de hål som uppstår efter upprepade anrop till *alloc()* och *release()*. Efter en genomförd körning av metoden är allt allokerat minne koncentrerat i den första delen av minnesområdet. Glöm inte att eventuella pekare som pekar på minnesplatserna också måste uppdateras.
- Buddy-allokering av minnet. Implementera *alloc()* och *release()* i Buddy.java enligt buddy-metodiken. Implementera även *printLayout()* i samma fil för att kunna visualisera minnet.