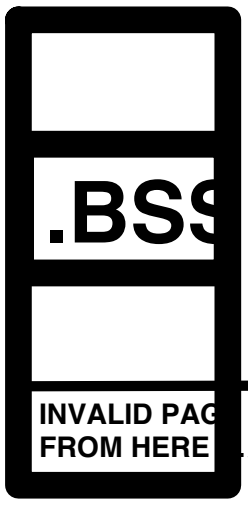
Program-break (BRK) in sistemi UNIX

Che cos'è il PROGRAM-BREAK all'interno di un address space su un'applicazione POSIX?

Esso rappresenta il LIMITE di MEMORIA LOGICA valida (quindi stiamo parlando dell'address space) oltre la fine della sezione .BSS (Dati non inizializzati).
Il BRK non è altro che la quantità di memoria che noi realmente possiamo raggiungere a partire dal BSS sino al punto in cui troviamo sotto delle pagine non valide.



Il BRK dipende dal BSS. Ed indica qual è il limite su cui possiamo arrivare a partire dal BSS fino a trovare pagine valide.

BRK

Questo BRK ci identifica un punto in cui poi inizierà una zona di pagine NON VALIDE.

Address Space

Abbiamo la possibilità di validare pagine utilizzando quest BRK?

Ciò implicherebbe dire di spostarlo più giù ed avere una zona in più.
In modo tale da ottenere un insieme di pagine che prima non erano utilizzabili.
Quindi, oltre al servizio MMAP, abbiamo un alternativa per mappare pagine (quindi mappare memoria) all'interno del contenitore logico che stiamo utilizzando?

Va ad indicare una nuova posizione di questo BRK rispetto alla posizione che avevamo originariamente.

int brk(void *addr)	
Descrizione	definisce il punto di rottura del programma
Argomenti	*addr: nuovo punto di rottura
Restituzione	-1 in caso di fallimento

Nuovo BRK

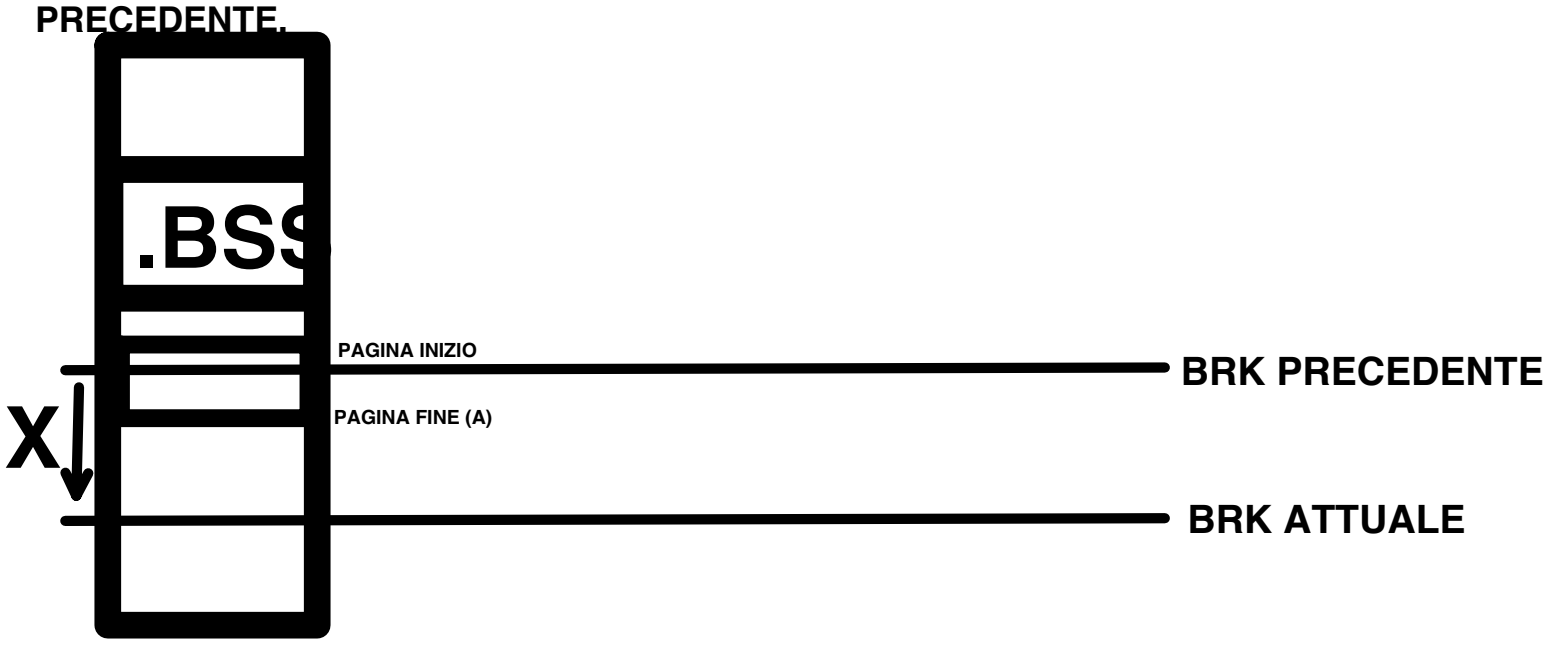
void* sbrk(intptr_t increment)

Nuovo BRK

Spostamento del BRK

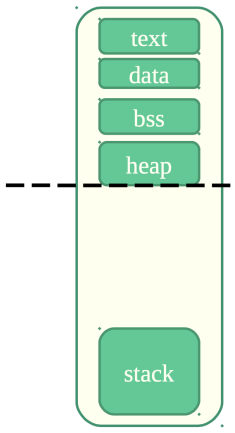
L'API sbrk ci permette di passare un parametro che non va a definire il nuovo valore del BRK che deve essere messo in pratica, bensì un incremento positivo o negativo del valore corrente del BRK. Il che implica dire che noi stiamo ampliando o riducendo il BRK.
Nel momento in cui cerchiamo di incrementare il BRK di una certa quantità di byte X, quindi vogliamo spostare più avanti (più giù) questo BRK di una quantità X, ci viene restituito il nuovo indirizzo che ci va a definire qual è il punto di rottura nell'address space.
Questo punto di rottura non necessariamente è allineato alla fine di una pagina, infatti queste API permettono di incrementare il BRK di un unico byte, il che implica dire che il BRK continua a cadere nella pagina dove correntemente cadeva.
Se noi analizziamo lo spostamento di X byte più avanti, magari questi X byte non necessariamente determinano il posizionamento di un'ulteriore pagina all'interno dell'address

space, magari si magari no: Dipende da qual è il punto in cui arriviamo nel nuovo BRK, rispetto alla posizione che inizialmente avevamo (A)che era l'ultima pagina valida CHE OSPITAVA IL BRK PRECEDENTE.



Queste API non svolgono un lavoro alla GRANULARITÀ della pagina.

Possiamo "MAPPARE/DEMAPPARE" PAGINE se spostiamo il BRK in maniera tale che questo spostamento vada a coinvolgere pagine inizialmente NON MAPPATE se lo aumentiamo o pagine MAPPATE se lo decrementiamo.



NOTA: il BRK non è necessariamente allineato alla fine di una pagina logica

Il BRK viene utilizzato in maniera interessante dalle librerie per la gestione della memoria, la libreria malloc che cosa fa quando ha bisogno di andare a reperire più memoria all'interno dell'address space?

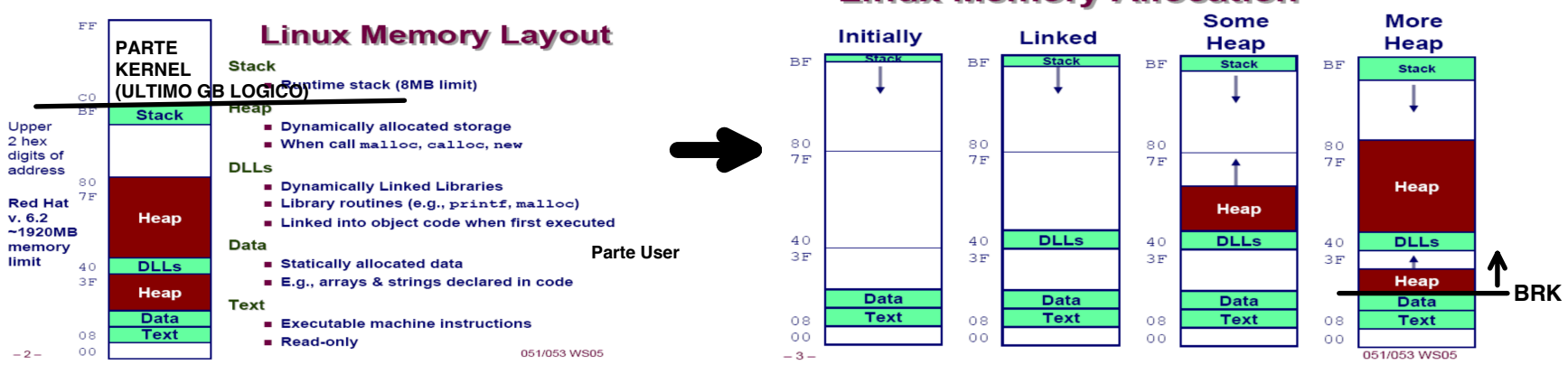
Non c'è verso.

Da questo address space non si esce, non possiamo uscire anche perché all'interno di un processore non abbiamo neanche i bit che ci possano permettere di esprimere un indirizzo che possa uscire.

Se il BRK corrente è quello, la MALLOC può spostare il BRK più giù utilizzando quel servizio semplice sopra descritto, riservando delle pagine che sono mappate all'interno dell'address space.

La seguente è una rappresentazione che noi abbiamo di un classico address space utilizzabile su processori X86 a 32 bit. Abbiamo un address space che è composto da 4 GB, con 32 bit possiamo indirizzare 2^{32} possibili locazioni di memoria - possibili byte - partendo dall'indirizzo 0 e poi salendo con gli indirizzi logici all'interno di questo contenitore abbiamo che il contenitore è usato in questo modo qua.

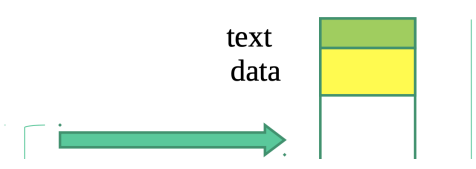
Linux (up to 2.4 kernel) – processori x86 a 32 bit



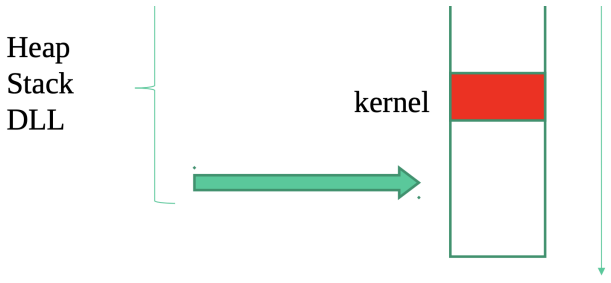
Quando noi lanciamo un'applicazione, in questo contenitore il sistema operativo ci permette di ospitare in una zona le pagine del TESTO e una zona per i DATI, poi eventualmente in un'altra zona la stack area (in alto primo AB nell'immagine a DX). Noi potremmo aver compilato anche utilizzando il gestore delle DLL delle librerie dinamiche (e quindi le informazioni che all'interno di questo address space competono la gestione delle DLL, quindi il modulo che carica le DLL, è caricato all'interno di un'altra zona di memoria). Essa è lontana rispetto alla zona dei dati, questo perché eventualmente noi possiamo con il BRK spostarci più su e ottenere della pagine (quindi la zona dell'heap nell'ultimo AB a DX) nel momento in cui c'è una richiesta a MALLOC per spostare questo limite all'interno dell'address space ed ottenere le zone di memoria utilizzabili.
Sopra la zona che avevamo riservato per le DLL abbiamo ulteriore possibilità di utilizzare MMAP per andare a mappare nuove pagine e chiaramente con la MMAP possiamo anche andare sotto a fare cose (sotto la DLL) perché MMAP ci permette di lavorare ovunque all'interno dell'address space, mentre invece il BRK ci permette di partire dal limite corrente che noi abbiamo e poi spostarci più SU.
Tipicamente questo spostamento viene fatto prima che il main prenda il controllo dagli starters, alcune volte gli starters hanno bisogno di riservare (mappare pagine) pagine all'interno dell'address space per fare delle cose: questo viene fatto spostando il BRK prima che il main poi prenda il controllo. L'ultimo GB logico, ossia la parte KERNEL, non è utilizzata direttamente dal nostro software, ma utilizzata indirettamente attraverso il passaggio in modalità kernel che viene offerto attraverso le funzioni della libreria di sistema, quindi all'interno di queste funzioni ci sono queste istruzioni macchina particolari che passano il controllo al kernel, che comincerà ad utilizzare gli indirizzi logici nella sua zona.

Andiamo a vedere la versione 64 bit.

... up to 2.6 (or piú recenti) – processori x86-64



Questo address space su processori a 64 bit non è 2^{64}, ma per dettagli che verranno specificati al corso avanzato di



sistemi operativi, abbiamo 2^{48} possibilità.
Abbiamo una zona iniziale dove abbiamo il testo, i dati e un po' dopo la metà il kernel, quindi la zona non utilizzabile direttamente nel momento in cui con un pointer vogliamo andare in quel punto dell'address space a leggere e/o scrivere, ma dobbiamo chiamare il kernel, e poi tutto il resto viene utilizzata per heap, stack e DLL.