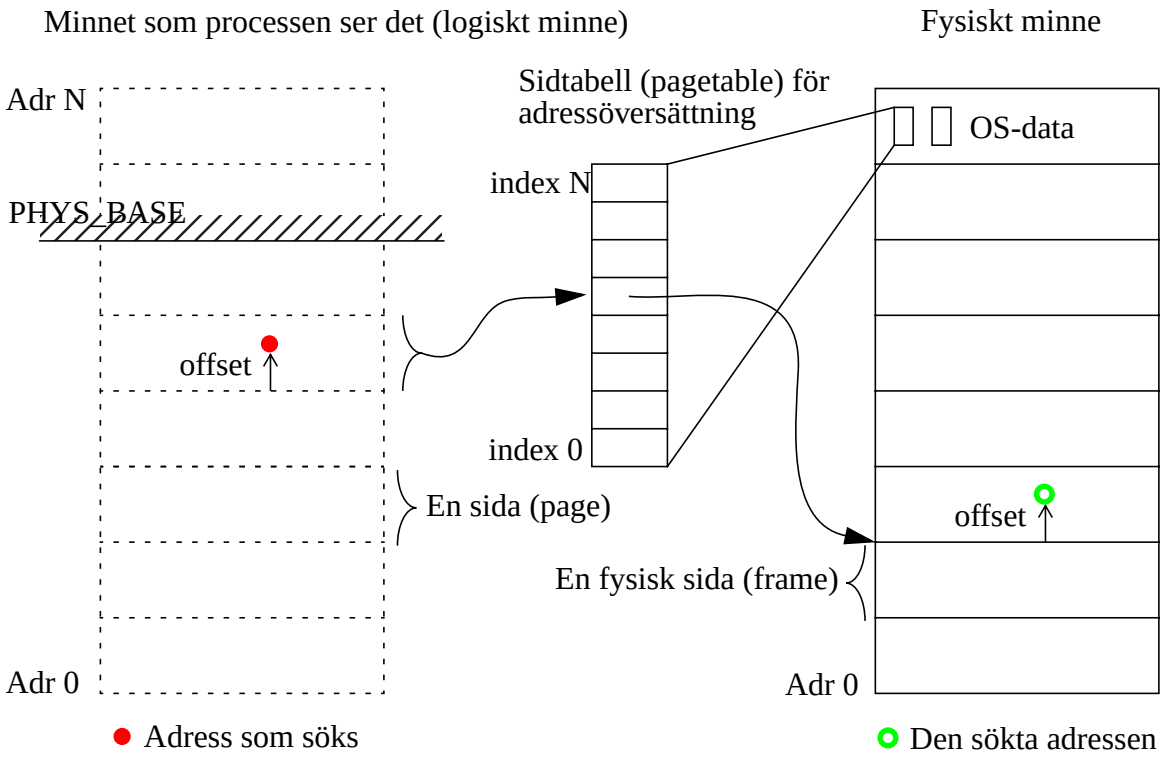
Deluppgift 5 Accesskontroll

Saker som kan vara bra att känna till

De flesta operativsystem, t.ex. PintOS, använder *paging* för att tilldela minne till processer på ett flexibelt sätt. Då ett program har placerats i minnet och börjat exekvera kallas det enligt gängse terminologi för en process, en term som används i fortsättningen. Med minnesmodellen *paging* översätter CPU alla adresser processen specificerar (user-adresser i processens logiska minnesrymd) till riktiga adresser (kernel-adresser i systemets fysiska minnesrymd). Detta görs genom att OS/CPU delar in allt minne i en uppsättning lika stora sidor (pages). Genom att titta på vilken sida en logisk adress finns i kan motsvarande frame i fysiskt minne slås upp i en tabell, och genom att titta på hur långt från starten på sidan adressen finns (offset) kan rätt adress inom den fysiska framen enkelt räknas fram.



Givet en adress USER_ADR i processen kan den fysiska adressen PHYS_ADR räknas ut enligt:

```
PAGE_NUMBER = USER_ADR / PAGE_SIZE; // heltalsdivision
PAGE_OFFSET = USER_ADR % PAGE_SIZE; // resten vid divisionen
FRAME_NUMBER = PAGETABLE[PAGE_NUMBER];
PHYS_ADR = FRAME_NUMBER + PAGE_OFFSET;
```

Dessa beräkningar är särskilt enkla och effektiva då sidstorleken är en jämn multipel av talsystemets bas eftersom man vid divisionen och rest-beräkningen helt enkelt kan stryka respektive behålla de N sista siffrorna, och vid additionen helt enkelt kan skriva ihop talen. Eftersom datorn räknar binärt kommer därför sidstorleken alltid vara en jämn multipel av 2 i ett riktigt system, d.v.s. 2^N.

Figuren ovan visar en översiktlig bild av hur det ser ut. Flera detaljer som inte är viktiga i denna uppgift har utelämnats. Normalt använder processen endast en liten del av adressrymden den tilldelats. I typfallet placeras programkod, gobala variabler, och konstanter i sidorna längst ned i processens minne (nära address 0), och processens exekveringsstack placeras i sidor högst upp i processens minne. Däremellan brukar många sidor vara helt oanvända. För varje sida som processen behöver i logiska minnet reserveras en frame i fysiskt minne och information om vilken frame som reserverades placeras på sidans plats i sidtabellen. Oanvända sidor markeras i sidtabellen som ogiltiga, och har ingen frame i fysiskt minne knuten till sig.

Ett problem som kan uppstå med denna minnesmodell är att processen kan råka försöka använda en adress där innehållet i översättningstabellen är ogiltigt. Detta händer ofta om processen är felaktigt skriven (t.ex. råkar använda en oinitierad pekare). Om detta fel uppstår då processen exekverar är det ingen större fara: CPU ger ifrån sig ett interrupt (pagefault) när uppslagningen misslyckas, interruptet hanteras av operativsystemet som då kan terminera processen med ett felmeddelande som brukar lyda något i stil med “page fault”, “segmentation fault” eller “bus error”. Om det däremot händer då OS exekverar är det allvarligt. Det betyder ju då att OS försöker använda en adress som “inte finns”, och OS terminerar sig själv, d.v.s. kraschar. Det får inte hända. OS får aldrig krascha!

Tyvärr är det nu så att när en process behöver en tjänst från OS måste OS få information om vad saken gäller. Denna information måste processen lagra i sitt eget minne (andras minne har den ju inte tillgång till). OS får reda på adressen dit, och OS använder adressen för att läsa och ev. modifiera minnet innan processen slutligen kan hitta resultatet av tjänsten där. Vad händer nu om processen avsiktligt eller oavsiktligt ger OS en ogiltig adress? Antingen kraschar OS, eller ännu värre, så lyckas OS (som ju har full access till allt minne) läsa/skriva adressen, vilket kan förstöra data för en annat process eller för OS självt.

För att hindra att något sådant kan uppstå måste operativsystem-programmeraren verifiera att alla adresser som kommer från en process är giltiga innan de används av OS. För att kontrollera att en address är giltig slår OS upp adressen i översättningstabellen för att se om det går bra. Hela poängen är alltså att utföra kontrollen *innan* läsning/skrivning sker till adresserna, så att OS kan undvika att krascha p.g.a. en ogiltig adress. En sådan uppslagning är dock långsam, och det är ofta *många* adresser att kontrollera då t.ex. läsning och skrivning från resp. till en fil kan behöva många megabyte (varje byte har en adress!) för att lagra all data. Det gäller därför att kontrollera precis så mycket (lite) som behövs. *Det går här att utnyttja att det som egentligen kontrolleras är innehållet i översättningstabellen. Det innehållet är identiskt för* alla *adresser inom en given sida, eftersom* alla *adresser inom en sida ger samma index i tabellen*. Resultatet för en av adresserna i en sida är alltså giltigt för alla adresser inom samma sida. Utnyttjas detta kan prestanda ökas avsevärt. En annan metod som ofta görs i praktiken är att nyttja processorns automatiska uppslagning som både är implementerad i hårdvara och (förmodligen) nyttjar en TLB.

När adresser ändå kontrolleras finns (senare och i PintOS) ytterligare två saker att kontrollera. Processer får på inga villkor använda adresser större än PHYS_BASE, då minnet ovanför den adressen är reserverat för OS kernel. PHYS_BASE är en konstant som bestämts av operativsystem-programmeraren. Slutligen är alla adresser som är NULL ogiltiga. I denna uppgift kan du slopa dessa två sista kontroller då de är förhållandevis enkla att hantera senare. Koncentrera dig på att kontrollera adresserna enligt ovan stycke.

Uppgift

Du skall skriva logiken som verifierar att allt minne inom ett visst intervall är giltigt. Till din hjälp har du beskrivningen ovan, ett antal funktioner och konstanten `PGSIZE` (se header-filen):

`void* pg_round_down(const void* adr);`
Returnerar första adressen i samma sida som `adr`.

`unsigned pg_no(const void* adr);`
Returnerar numret på sidan som innehåller `adr`. Sidnummer börjar räkna på 0, som allt annat i C.

`void *pagedir_get_page (void *pd, const void *adr);`
Använder översättningstabellen `pd` för att slå upp fysisk adress motsvarande `adr`. Om översättningen misslyckas returneras `NULL`. I denna uppgift används ett simulerat testsystem där `pd` kan anges till `NULL`.

`bool is_end_of_string(char* adr);`
Returnerar `true` om adressen `adr` innehåller ett noll-tecken, '\0'. Eftersom koden endast simulerar systemet går inga adresser att läsa eller skriva. Därför måste du använda denna funktion för att avgöra om en sträng är slut.
Funktionen ersätter alltså testet `(*adr == '\0')`.

De funktioner du måste implementera är:

`bool verify_fix_length(void* start, int length);`
Kontrollera alla adresser från och med `start` till och *inte* med `(start+length)`. Returnerar `true` om alla adresser i intervallet är giltiga.

`bool verify_variable_length(char* start);`
Kontrollera alla adresser från och med `start` till och med den adress som först innehåller ett noll-tecken, '\0'. (C-strängar lagras på detta sätt.) Returnerar `true` om alla adresser som används av strängen är giltiga.

Ett givet testprogram, filen `verify_adr.c` samt `pagedir.h` och `pagedir.o`, är förberett för dina ändringar. Tänk på att dessa filer inte implementerar ett verkligt system. Adresserna går därför inte att läsa eller skriva, och det behöver du inte heller eftersom du endast skall kontrollera om de är giltiga. Då du behöver testa om en adress innehåller ett noll-tecken (för att hitta slutet på en sträng) finns en funktion som simulerar just den läsningen given ovan. Testprogrammet skall inte skriva ut några fel, och bör avsluta inom en minut när du gjort rätt (gör du fel kommer det ta avsevärt längre tid eftersom varje uppslagning är mycket långsam i det simulerade systemet).

För att kompilera koden använder du som vanligt `gcc` med alla käll- och objektfiler som skall ingå:

```
gcc -Wall -Wextra -std=c99 -g verify_adr.c pagedir.o
```