**skróty:** przadr - przestrzeń adresowa

# Zarządzanie pamięcią

## Pojęcie przestrzeni adresowej

Aby umożliwić wielu aplikacjom przebywanie w pamięci w tym samym czasie w taki sposób, by wzajemnie sobie nie przeszkadzały, trzeba rozwiązać dwa problemy: ochrony i relokacji. Prymitywne rozwiązanie problemu: w komputerze IBM 360 wszystkie komórki pamięci były oznaczone kluczem zabezpieczającym; dla każdego słowa pobranego z pamięci klucz uruchamiającego procesu był porównywany z kluczem słowa pamięci. Nie rozwiązywało to problemu relokacji. Lepsza metoda: opracowanie nowej abstrakcji pamięci - przestrzeni adresowej.

**przestrzeń adresowa** - zbiór adresów, które proces może wykorzystać do zaadresowania pamięci. Każdy proces ma swoją własną przadr, która jest niezależna od przadrów należących do innych procesów (poza sytuacjami, gdy procesy chcą współdzielić swoje przadry).

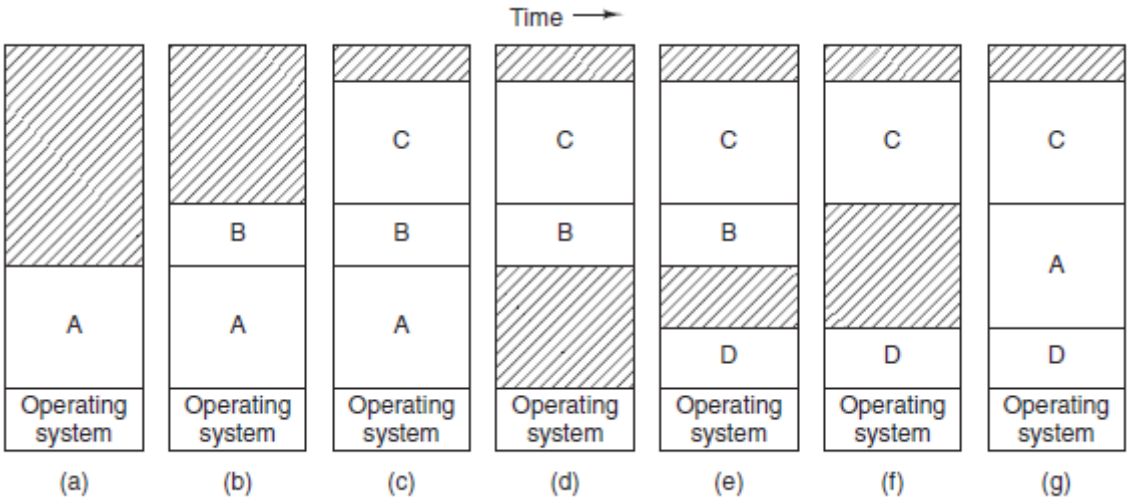
Trudnym zadaniem jest przydzielenie każdemu programowi właśnego przadra, tak aby adres 28 w jednym programie oznaczał inną lokalizację fizyczną niż adres 28 w innym programie. Prosty sposób rozwiązania tego problemu - rejestry bazy i limitu. Stosuje się w nim prostą wersję dynamicznej relokacji. Jej działanie polega na zmapowaniu przadra każdego procesu na różne części pamięci fizycznej. Wyposażene każdego procesora CPU w dwa specjalne rejestry sprzętowe: bazowy i limitu. Kiedy proces się uruchamia, rejestr bazowy jest ładowany wartością adresu początku programu w pamięci fizycznej, natomiast rejestr limitu jest ładowany wartością rozmiaru programu. Za każdym razem, gdy proces odwołuje się do pamięci w celu pobrania instrukcji albo oczytania czy zapisania słowa danych, sprzet procesora, zanim wyśle adres na szynę pamięci, automatycznie dodaje wartość bazową do adresu wygenerowanego przez proces i sprawdza, czy oferowany adres jest większy równy od wartości w rejestrze limitu. Jeśli tak, to generowany jest błąd, a operacja przerywana. Wady rozwiązania: konieczność wymuszania dodawania i porównywania przy każdym odwołaniu do pamięci.

### Wymiana pamięci

Poważne współczesne aplikacje użytkowe, takie jak Photoshop, często zużywają dużo pamięci na samo uruchomienie i przetwarzanie danych. W konsekwencji utrzymywanie wszystkich procesów w pamięci przez cały czas wymaga olbrzymich ilości pamięci i nie może być zrfealizowane, jeśli rozmiar pamięci na to nie pozwala. Dwa ogólne rozwiązania problemu przeładowania pamięci:

- wymiana (ang. *swapping*) - polega na załadowaniu określonego procesu w całości, uruchiomieniu go przez pewien czas, a następnie umieszczeniu z powrotem na dysku. Bezczynne procesy zwykle są zapisane na dysku, zatem wtedy, gdy nie działają, w ogóle nie zajmują pamięci (ale niektóre okresowo budzą się w celu wykonania swojej pracy, a następnie przechodzą w stan uśpienia).
- pamięć wirtualna - umożliwia programom działanie nawet wtedy, gdy częściowo są zapisane w pamięci głównej.

Wymiana: alokacja pamięci zmienia się, w miarę jak procesy wchodzą do pamięci i ją




opuszczają.

**kompaktowanie pamięci** - kiedy w wyniku wymiany w pamięci tworzą się duże luki, można je scalić w jeden ciągły blok poprzez przeniesienie wszystkich prcoesów tak daleko w dół, jak się da. Zwykle się tego nie robi, ponieważ operacja ta wymaga dużo czasu procesora.

**scalanie** - złączanie dwóch sąsiednich wolnych bloków pamięci. Gdy aplikacja zwalnia pamięć, może powstać luka wolnej pamięci pomiędzy zajętymi blokami. Scalanie jest używane w celu zmniejszania fragmentacji zewnętrznej, ale nie zawsze jest efektywne. Scalanie może być poczynione natychmiast, odroczone lub niewykonane w ogóle.

To avoid this unnecessary coalescing and splitting, the lazy buddy system defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible. In general, the lazy buddy system tries to maintain a pool of locally free blocks and only invokes coalescing if the number of locally free blocks exceeds a threshold. If there are too many locally free blocks, then there is a chance that there will be a lack of free blocks at the next level to satisfy demand. Most of the time, when a block is freed, coalescing does not occur, so there is minimal bookkeeping and operational costs.

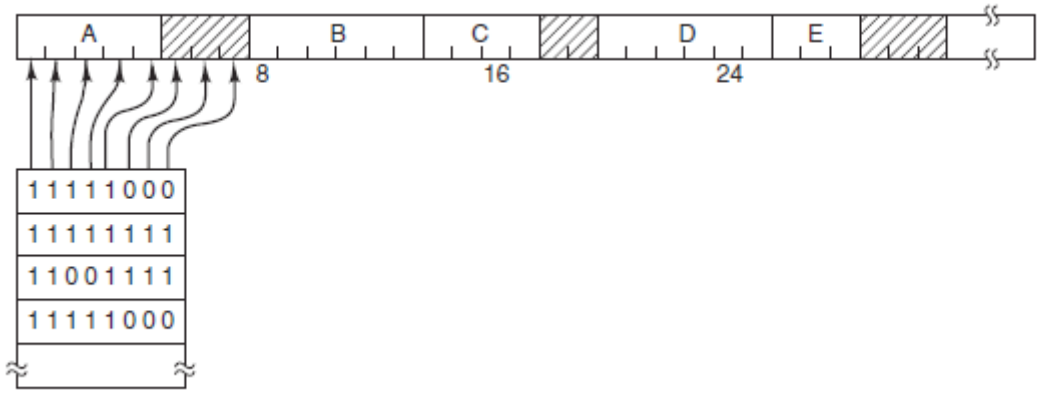
Warto zwrócić uwagę na to, ile pamięci należy przydzielić procesowi podczas jego tworzenia lub wymiany z dyskiem. Jeśli są tworzone procesy o stałym rozmiarze, który nigdy się nie zmienia, alokacja jest prosta - sysopek alokuje dokładnie tyle pamięci, ile trzeba. Jeśli jednak segmenty danych procesów mogą się rozrastać, np. poprzez dynamiczną alokację pamięci ze sterty, przy każdej próbie wzrostu rozmiaru procesu występuje problem. Jeśli obok procesu jest blok wolnej pamięci - ok, można zaalokować i proces rozrasta się, zajmując ten blok. Jeśli obok procesu jest inny proces, trzeba ten rozrastający przenieść do bloku wolnej pamięci o odpowiednim rozmiarze albo kilka procesów będzie musiało zostać wymienionych z dyskiem, tak aby powstał wystarczająco duży blok w pamięci. Jeśli prcoes nie będzie miał możliwości zwiększenia swojego rozmiaru w pamięci, a obszar wymiany na dysku jest zapełniony, proces będzie musiał być zawieszony do czasu zwolnienia miejsca lub może zostać zniszczony, Jeśli spodziewamy się, że większość procesów będzie się rozrastała podczas swojego działania, warto zaalokować niecowięcej pamięci w czasie wymiany procesu z dyskiem lub jego przenoszenia. Podczas wymiany procesów na dysk należy pamiętać, że wymianie powinna podlegać tylko pamięć faktycznie używana. Wymiana przy okazji dodatkowej pamięci to marnotrawstwo.

## Zarządzanie wolną pamięcią

Dwa modele śledzenia wykorzystania pamięci: mapy bitowe i listy wolnych bloków.

### Mapy bitowe

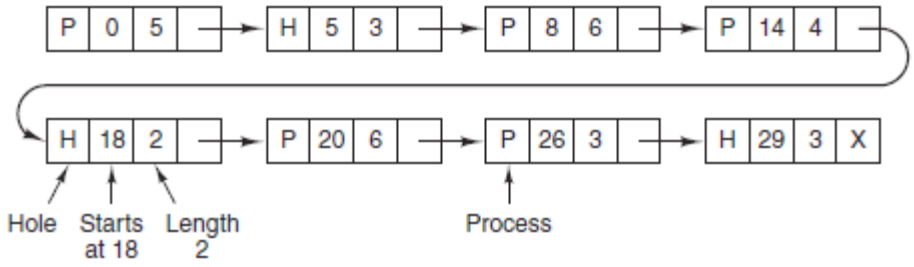
Pamięć jest podzielona na jednostki alokacji o rozmiarze od kilku słów do kilku kilobajtów. Każdej jednostce alokacji odpowiada bit na mapie bitowej. Ten bit ma wartość 0, jeśli jednostka alokacji jest wolna, i 1, gdy jest zajęta. Rozmiar jednostki alokacji jest ważnym problemem projektowym. Im mniejszy rozmiar jednostki alokacji, tym większa mapa bitowa. W przypadku wyboru jednostki alokacji o dużym rozmiarze mapa bitowa będzie mniejsza, ale spora część pamięci może być zmarnowana w ostatniej jednostce alokacji procesu, gdy rozmiar procesu nie jest dokładną wielokrotnością jednostki




alokacji. Prob
lem z mapą bitową: w przypadku podjęcia decyzji o załadowaniu procesu składającego się z k jednostek menedżer pamięci musi przeszukać mapę bitową w celu znalezienia k kolejnych bitów o wartości 0 - jest to bardzo wolne.

### Listy jednokierunkowe

Innym sposobem śledzenia pamięci jest utrzymywanie list jednokierunkowych dla zaalokowanych i wolnych segmentów pamięci, przy czym segment albo zawiera proces, albo jest pustym miejscem pomiędzy dwoma procesami. Wygodniej jest posługiwać się listą dwukierunkową niż jednokierunkową, ponieważ miejsce w tabeli procesów dla procesu końcowego będzie standardowo wskazywało na pozycję na liście odpowiadającą samemu




procesowi.

### Algorytmy alokacji pamięci dla utworzonego procesu lub wymiany istniejącego procesu z dyskiem

#### First-fit

Menedżer pamięci skanuje listę segmentów tak długo, aż znajdzie wolny blok o odpowiedniej wielkości. Ten blok jest następnie dzielony na dwie części - jedna zostaje przeznaczona na proces, natomiast druga na nieużywaną pamięć, chyba że wystąpi

mało prawdopodbny przypadek, w którym wolny blok będzie dokładnie odpowiadał rozmiarowi procesu. Wydajność: szybki, ponieważ operacje wyszukiwania są w nim ograniczone do minimum.

#### Next-fit

Algorytm działa w taki sam sposób, jak first-fit, poza tym, że zapamiętuje miejsce, w którym został znaleziony wolny blok o odpowiedniej wielkości. Kiedy algorytm zostanie wywołany następnym razem w celu znalezienia wolnego bloku, rozpoczyna wyszukiwanie od miejsca, w którym zakończył szukanie ostatnim razem, a nie zawsze od początku, jak w przypadku first-fit. Wydajność: nieco gorszy od first-fit.

#### Best-fit

Polega na przeszukaniu całej listy od początku do końca i wybraniu najmniejszego wolnego bloku, który umożliwia zamieszczenie procesu. Wydajność: wolniejszy od pierwszy pasujący, przy każdym wywołaniu musi być przeszukana cała lista. Skutkuje większym marnotrawstwem pamięci niż first i next-fit, ponieważ wtedy pamięć wypełnia się nieprzydatnymi do niczego wolnymi blokami o niewielkich rozmiarach.

#### Worst-fit

Aby obejść problem dzielenia niemal dokładnie dopasowanego bloku na proces i niewielki wolny blok, można wymyślić algorytm najgorszy pasujący - zawsze wybierać największy możliwy wolny blok. Powstający wolny blok będzie na tyle duży, że ma szansę być użyteczny. Nie jest jednak dobrym pomysłem.

#### Quick-fit

Szybkie dopasowanie - utrzymywane są oddzielne listy dla częściej wykorzystywanych rozmiarów bloków. Algorytm może wykorzystywać tabelę zawierającą n pozycji, w której peirwsza pozycja jest wskaźnikiem na początek listy 4-kB bloków, druga pozycja jest wskaźnikiem na listę 8-kB bloków, itd. Bloki wolne o rozmiarze np. 21 kB mogłyby być umieszczone na liście bloków o rozmiarze 20 kB lub na specjalnej liście bloków o nietypowych rozmiarach. Wydajność: znalezienie wolnego bloku o pożądanym rozmiarze jest bardzo szybkie. Ma jednak takie same wady jak wszystkie mechanizmy, które sortują bloki według ich rozmiaru. Jeśli zatem proces zakończy działanie lub zostanie przeniesiony do pliku wymiany na dysk, to znalezienie sąsiadów w celu sprawdzenia, czy jest możliwe scalenie, okazuje się kosztowne. Jeśli scalenie nie zostanie wykonane, pamięć ulegnie szybkiej fragmentacji na wiele bloków wolnych o małych rozmiarach, w których procesy nie będą mogły się mieścić.

Możliwe optymalizacje pierwszych czterech algorytmów

- Wszystkie algorytmy można przyspieszyć poprzez utrzymywanie osobnych list dla procesów i bloków wolnych. W ten sposób cała energia może być skupiona na przeszukiwaniu listy wolnych bloków. Wydajność poprawek: dodatkowa złożoność i spowolnienie podczas zwalniania pamięci, ponieważ zwolniony segment trzeba usunąć z listy procesów i umieścić na liście wolnych bloków.
- Gdy dla procesów i bloków wolnych są utrzymywane osobne listy, listę bloków wolnych można posortować względem rozmiaru. Dzięki temu best-fit może działać szybciej. Wtedy first i next-fit są jednakowo szybkie, a worst-fit nie ma sensu.

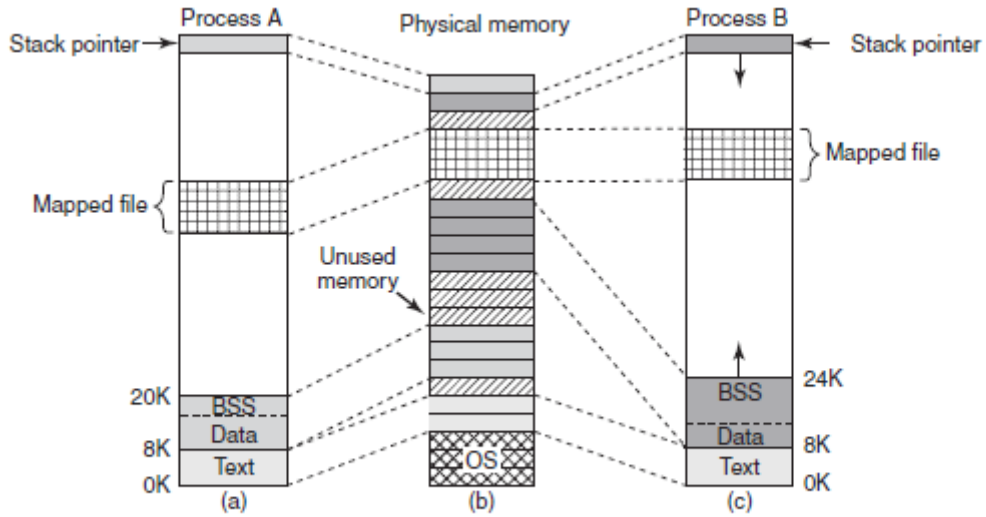
- Tyci optymalizacja: zamiast utrzymywać oddzielny zbiór struktur danych do przechowywania listy wolnych bloków, informacje mogą być zapisane w wolnych blokach.

# Zarządzanie pamięcią w systemie Linux

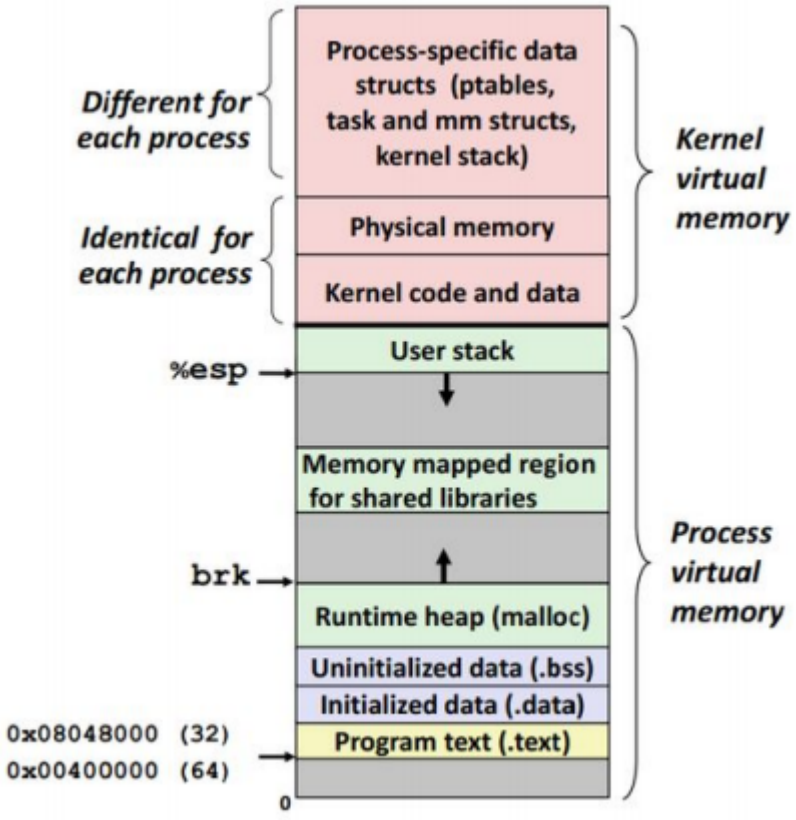
## Podstawowe pojęcia

Każdy proces systemu Linux dysponuje przadrem podzielonym na trzy logiczne segmenty:

- tekst
    - obejmuje rozkazy maszynowe składające się na kod wykonywalny programu
    - jest generowany przez kompilator i asembler, które tłumaczą kod C na kod maszynowy
    - zwykle jest dostępny tylko do odczytu
    - nie rozrasta się, nie kurczy się ani nie jest zmieniany w żaden inny sposób
- dane
    - jest miejscem składowania wszystkich zmiennych, łańcuchów, tablic i innych danych programu
    - składa się z dwóch części: danych inicjalizowanych (zmienne i stałe kompilatora, które w czasie uruchamiania programu wymagają przypisania konkretnych wartości początkowych) i danych nieinicjalizowanych BSS (zmienne należące do bloku BSS są inicjalizowane wartością zero po załadowaniu program)
    - może być zmieniany - programy stale modyfikują swoje zmienne, wiele programów musi dynamicznie przydzielać w czasie wykonywania programu niezbędnego przadra
        - `brk` - wywsys, za pomocą którego program może ustawiać rozmiar swojego segmentu danych. Aby uzyskać więcej pamięci, program może zwiększyć rozmiar swojego segmentu danych.
        - `malloc` - procedura stosowana do przydzielania pamięci, korzysta z `brk`. Deskryptor przadra procesu zawiera informacje o obszarach pamięci dynamicznie przydzielonych danemu procesowi, tzw. stercie (ang. *heap*)
- stos
    - rozpoczyna się na szczycie wirtualnego przadra i rośnie w dół w kierunku adresu zerowego
    - jeśli stos rozrośnie się poniżej dolnej granicy tego segmentu, wystąpi błąd sprzętowy, a sysopek obniży granicę o jedną stronę
    - same programy wprost nie zarządzają rozmiarem segmentu stosu




a) wirtualny przadr procesu A; b) pamięć fizyczna; c) wirtualny przadr procesu B



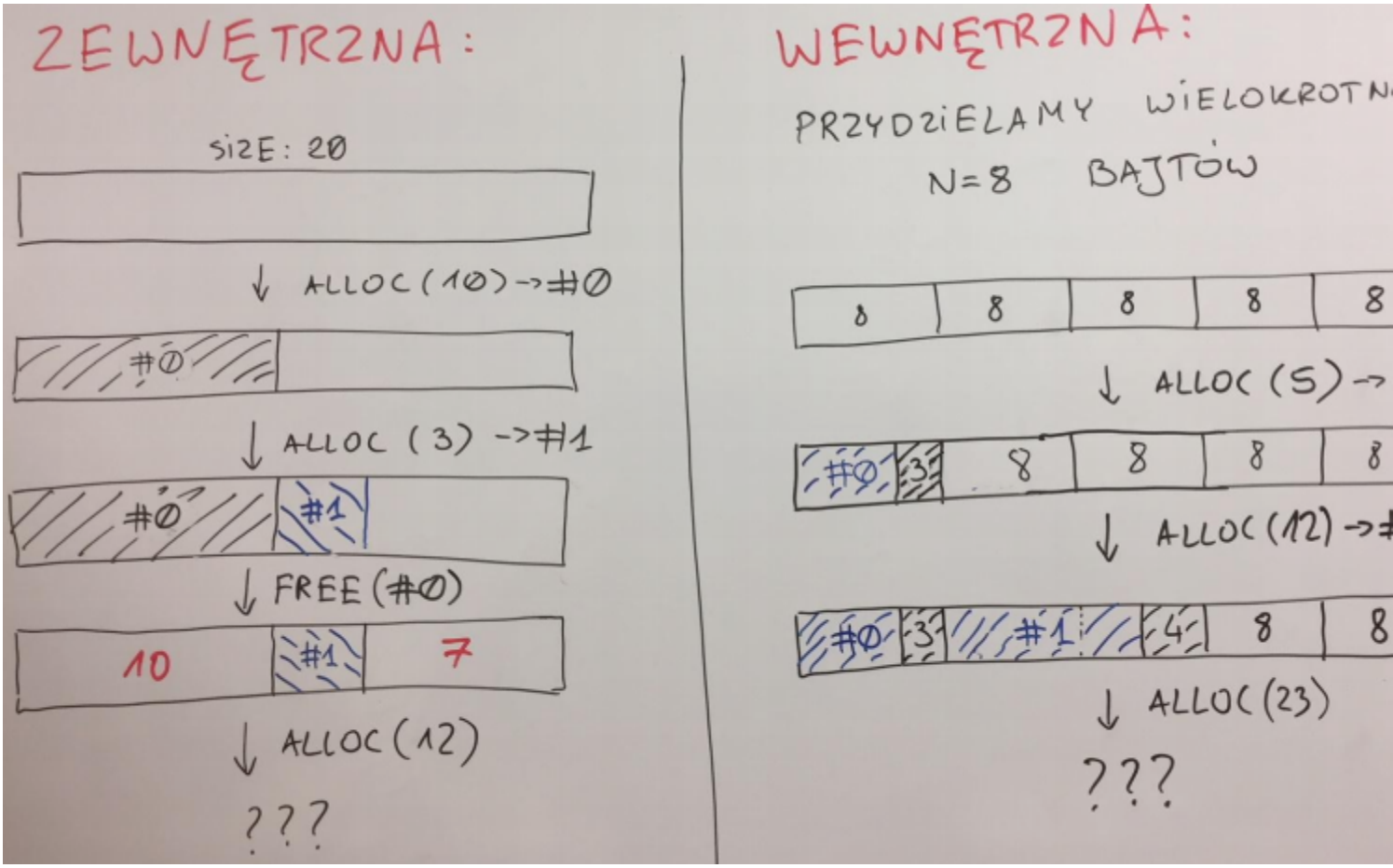

## Mechanizmy przydzielania pamięci

System Linux obsługuje wiele mechanizmów przydzielania pamięci. Główny mechanizm przydzielania nowych ramek stron pamięci fizycznej, nazywany dyspozytorem stron (ang. *page allocator*), wykorzystuje doskonale znany algorytm bliźniaków (ang. *buddy algorithm*). Fragmentację zewnętrzną w algo bl. niwelujemy translacją adresów.

**fragmentacja wewnętrzna** - występuje wtedy, kiedy mamy ustalony rozmiar porcji pamięci (strony np. 4kB), a programy nie wypełniają tych stron. Dla przykładu: cztery procesy po 1kB pamięci będą używać 4 stron zamiast 1, przez co 12kB pada ofiarą fragmentacji wewnętrznej.

**fragmentacja zewnętrzna** - występuje, kiedy nie mamy ustalonego rozmiaru porcji pamięci, a małe procesy zwalniają małe porcje pamięci, które idą w nieużytki. Dla

przykładu mamy 3 procesy z przydzielonymi po sobie pamięciami równymi 10kB, 1kB, 5kB. Drugi proces się kończy, zwalnia pamięć, a następne procesy chcą co najmniej 2kB pamięci, więc 1kB pada ofiarą fragmentacji zewnętrznej.

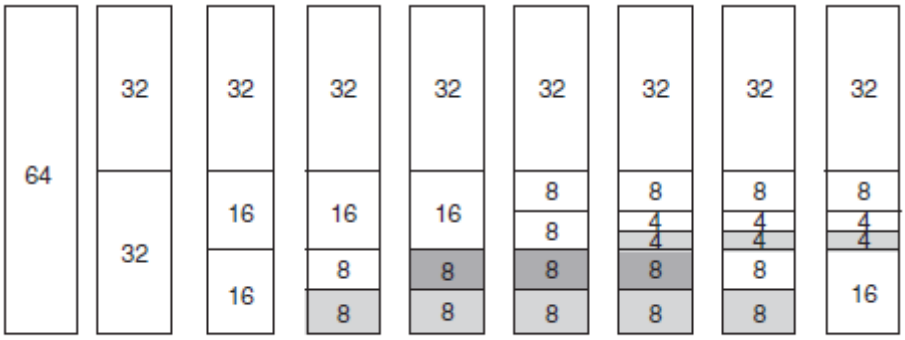



Linux zarządza pamięcią, stosując algorytm bliźniaków wzbogacony o dodatkową tablicę, której pierwszy element reprezentuje początek listy bloków zajmujących po jednej jednostce, drugi element reprezentuje początek listy bloków zajmujących po dwie jednostki, itd. Takie rozwiązanie umożliwia błyskawiczne odnajdywanie bloków o rozmiarach równych potędze 2. Opisany algorytm prowadzi do sporej fragmentacji wewnętrznej, ponieważ w odpowiedzi na żądanie obszaru zajmującego 65 stron otrzymujemy obszar zajmujący aż 128 stron.

`kmalloc`: aby złagodzić skutki tego problemu twórcy systemu Linux stworzyli drugi mechanizm przydzielania pamięci, tzw. dyspozytor płytowy (ang. *slab allocator*), który przydziela pamięć, stosując standardowy algorytm bliźniaków, by następnie wycinać z nich plastry (mniejsze jednostki) i zarządzać tymi drobnymi obszarami niezależnie od siebie.
`vmalloc`: trzeci dyspozytor pamięci, jest wykorzystywany w sytuacji, gdy żądana pamięć musi być ciągła tylko w przestrzeni wirtualnej (kiedy nie jest wymagana ciągłość w pamięci fizycznej). Powoduje spadek wydajności.

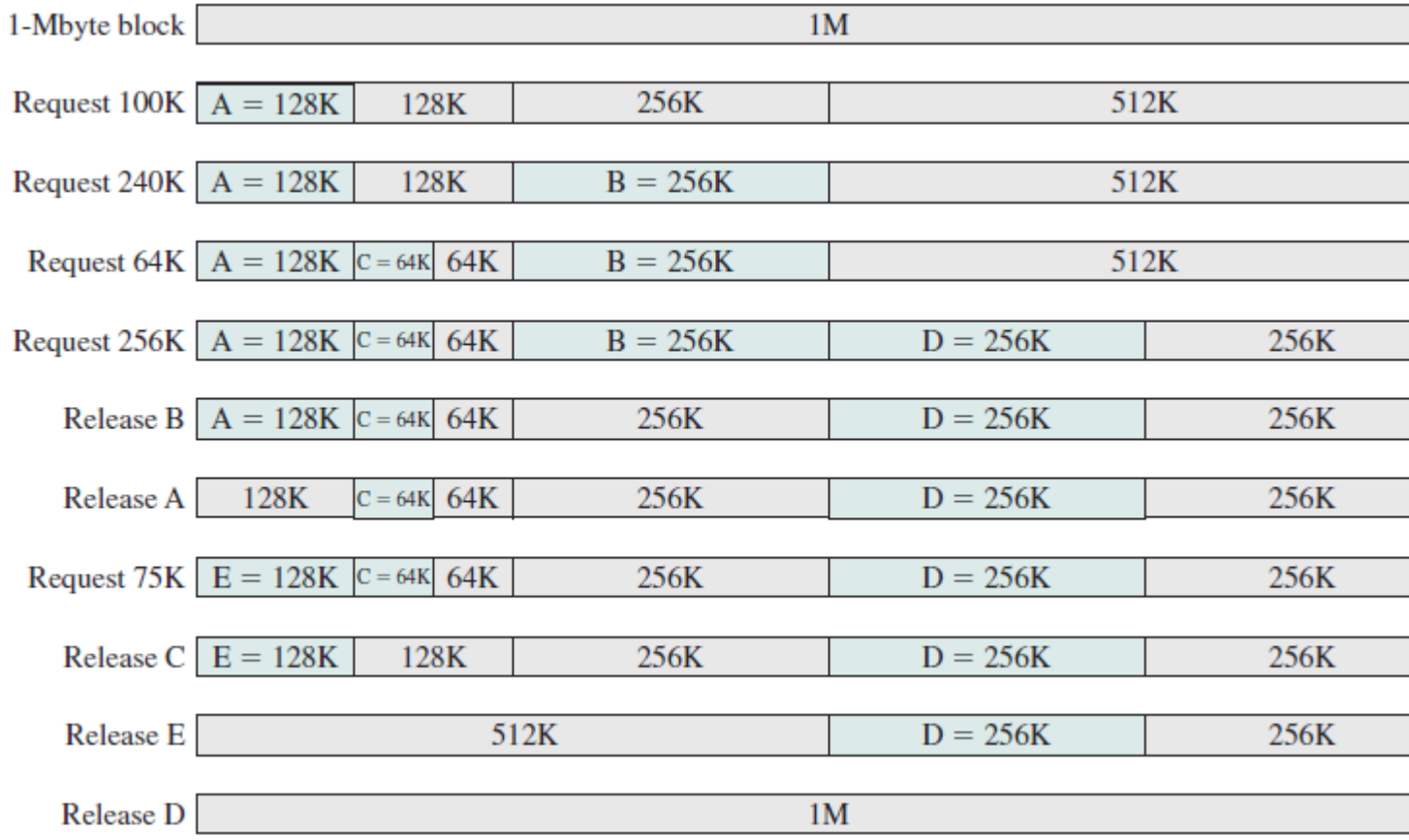
#### Algorytm bliźniaków

Początkowo pamięć składa się z pojedynczego, ciągłego obszaru - przyjmijmy, że zajmuje 64 strony. Kiedy dyspozytor stron otrzymuje żądanie pamięci, zaokrągla żądaną liczbę stron do potęgi dwójki.

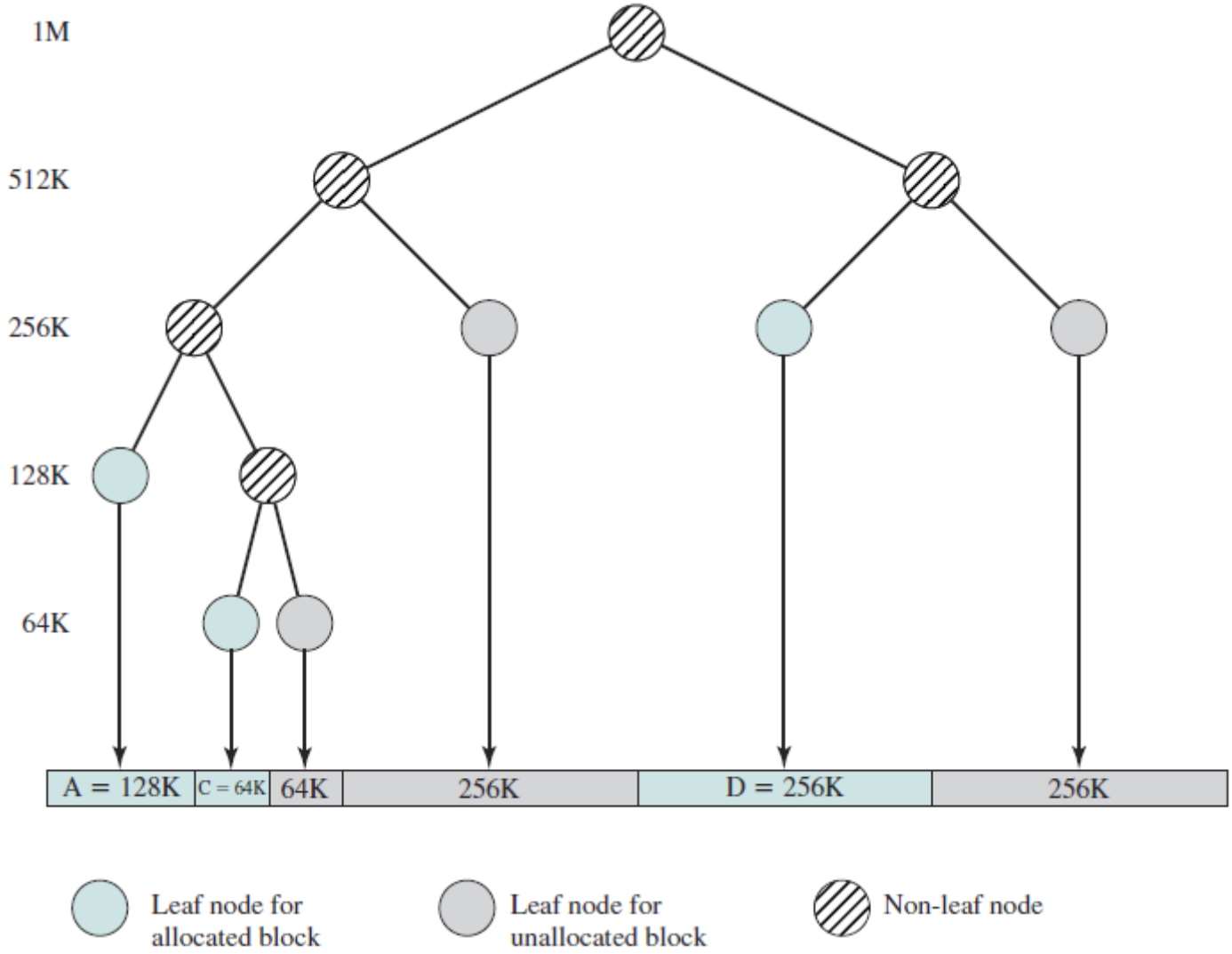



1. 8 stron b: cały obszar pamięci zostaje podzielony na pół c: każda z tych części okazuje się zbyt duża, pierwsza z nich jest ponownie dzielona na pół d: podział pierwszej z otrzymanych połówek Dopiero teraz dysponujemy obszarem właściwych rozmiarów (szary kolor), zatem można go przydzielić procesowi, który wysłał żądanie.
2. 8 stron e: można je zrealizować od razu
3. 4 strony f: najmniejszy możliwy obszar jest dzielony g: jego połowa jest przydzielana żądającemu procesowi
4. Zwolnij drugi obszar 8 stron
5. Zwolnij pierwszy obszar 8 stron h: i: dwa zwolnione obszary należały do tego samego fragmentu obejmującego łącznie 16 stron, dyspozytor może je ponownie scalić

Inny przykład:



Na drzewie:


1M
512K
256K
128K
64K
A = 128K
C = 64K
64K
256K
D = 256K
256K
Leaf node for allocated block
Leaf node for unallocated block
Non-leaf node