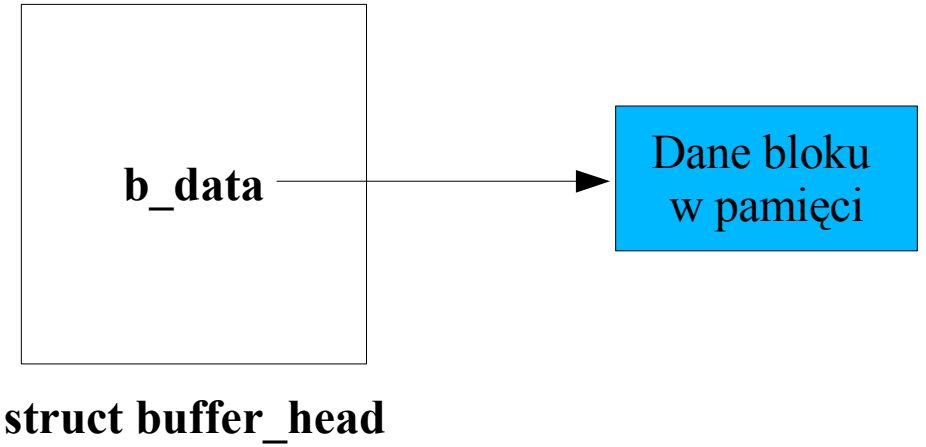
Wykład 7
Podręczna pamięć buforowa (ang. buffer cache) w systemie Linuks

Wstęp

- Przyczyną wprowadzenia pamięci buforowej są ogromne różnice w czasie dostępu do pamięci operacyjnej i dyskowej.
  - Przykładowa prędkość transmisji: dysk 30 MB/s pamięć 500 MB/s
  - Przykładowy czas dostępu : pamięć 50 ns, dysk 20 ms (***milion razy dłużej !!!***)
- W celu zwiększenia przepustowości systemu pamięć buforowa wykorzystuje dwa mechanizmy.
  - Przechowywanie najczęściej używanych bloków. Dzięki temu kolejne żądanie odczytu bloku może zostać zaspokojone z pamięci RAM.
  - Opóźniony zapis. Bloki przeznaczone do zapisania są zapisywane z pewnym opóźnieniem. Dzięki temu możliwe jest wielokrotne (np. Naprzemienne) wykonania operacji zapisu i odczytu tego samego bloku bez dostępu do urządzenia.
- Implementacja pamięci buforowej w Linuksie bardzo przypomina implementację w systemie UNIX System V R2 opisaną w rozdziale 3 książki M. Bacha “Budowa systemu operacyjnego UNIX”.
  - Linus korzystał z tej książki.
  - Identyczne są nazwy podstawowych funkcji (getblk, bread, brelse)

Reprezentacja bufora



- Jeden bufor przechowuje pojedynczy blok.
- Struktura buffer_head (nagłówek bufora) przechowuje informacje wszelkie informacje związane z buforem.
- Dane w pamięci wskazywane są przez pole b_data.
- Od tej chwili poprzez “bufor” rozumiemy egzemplarz struktury buffer_head

Przypomnienie: struktura buffer_head

```
struct buffer_head {
   unsigned long b_blocknr;         /* block number */
   kdev_t b_dev;                    /* device (B_FREE = free) */
   kdev_t b_rdev;                   /* Real device */
   unsigned long   b_rsector;       /* Real buffer location on disk */
   struct buffer_head * b_next;     /* Hash queue list */
   struct buffer_head * b_this_page; /* circular list of buffers in one page */
   unsigned long b_state;           /* buffer state bitmap (see above) */
   struct buffer_head * b_next_free;
   unsigned int b_count;            /* users using this block */
   unsigned long b_size;            /* block size */
   char * b_data;                   /* pointer to data block (1024 bytes) */
   unsigned int b_list;             /* List that this buffer appears */
   unsigned long b_flushtime;
   unsigned long b_lru_time;
   struct wait_queue * b_wait;
   struct buffer_head * b_prev;
   struct buffer_head * b_prev_free;
   struct buffer_head * b_reqnext;
};
```

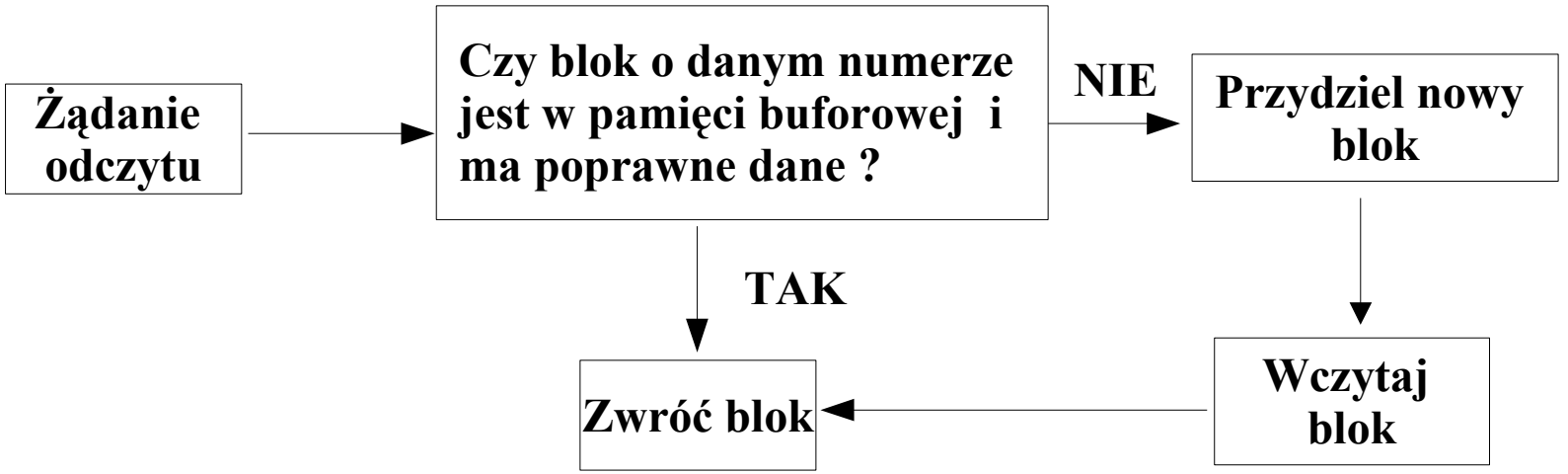
Stan bufora - b_state

- Jest to pole bitowe będące kombinacją poniżych flag, z których każda może być niezależnie włączona bądź wyłączona.
  - BH_Uptodate – bufor zawiera aktualne dane dla odpowiedniego bloku
  - BH_Dirty – bufor został zmodyfikowany i jego zawartość trzeba zapisać na dysk.
  - BH_Lock – bufor zablokowany, wykorzystany w sytuacji gdy trwa operacja zapisu/odczytu dla tego bufora. Bufory związane z elementami kolejki żądań urządzenia blokowego są zablokowane.
- API do manipulacji stanami bufora udostępniane innym podsystemom jądra (plik ./include/linux/fs.h)
  - mark_buffer_uptodate(struct buffer_head *buffer, int on). Dla on=1 ustawia flagę BH_Uptodate, 0 kasuje.
  - mark_buffer_clean(struct buffer_head *buffer). Kasuje flagę BH_Dirty
  - mark_buffer_dirty(struct buffer_head *buffer, int flag). Ustawia flagę BH_Dirty. Pole flag służy do ustawienia czasu zapisu 1 oznacza blok specjalny (5s) a zero blok zwykły (30s).

b_count - licznik odniesień (ang. reference counter) bufora

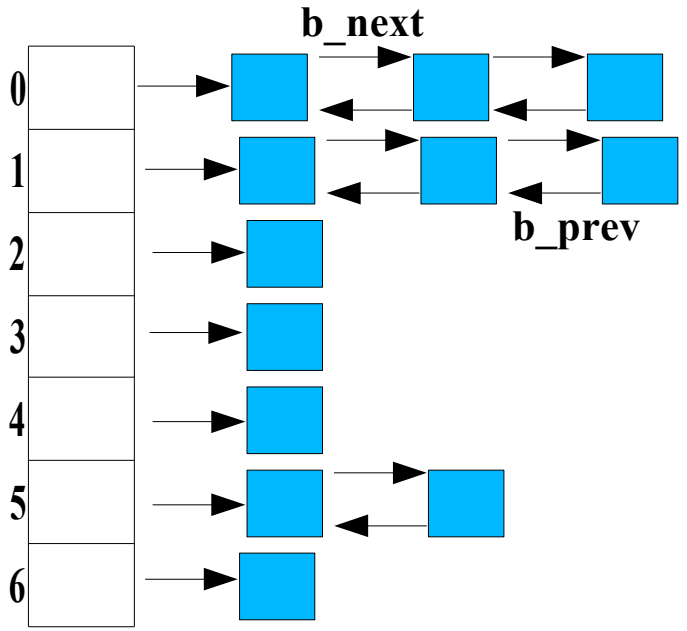
- Przy zarządzaniu czasem życia bufora, jądra stosuje technikę zwaną zliczaniem referencji.
- Ten sam bufor może być jednocześnie wykorzystany z różnych miejsc jądra (np. katalog przeglądany przez kilka procesów). Zliczanie referencji rozwiązuje problem zwolnienia bufora (np. w celu przyszłego ponownego wczytania danych innego bloku), dopiero wtedy gdy mamy pewność, że żaden z podsystemów jądra nie wykorzystuje bufora.
  - Nowo utworzony bufor ma licznik referencji równy 0.
  - Przy pobieraniu bufora (funkcja getblk) licznik odniesień b_count jest zwiększany o 1.
  - Podsystem jądra, który pobrał bufor ma obowiązek wywołania funkcji zwalniającej bufor (brelse albo bforget). Funkcja ta zmniejsza wartość licznika odniesień o jeden.
  - Tak długo jak licznik odniesień jest większy od zera, bufor pozostaje w pamięci.

Żądanie odczytu



- W pamięci buforowej mogą być przechowywane tysiące bloków.
  - Musimy być w stanie szybko odpowiedzieć, czy dany blok jest w pamięci.
  - Linuks w tym celu stosuje haszowanie.
- Musi istnieć mechanizm przydzielenia nowego bufora.
  - Kilka możliwości: przydziel dodatkową pamięć, odzyskaj (recykling) inny ***nieużywany*** bufor.

Tablice mieszające



- Oblicz wartość funkcji mieszającej

i=hash(device,block)= (device xor block) mod size.

  - Device - numer urządzenia
  - Block - numer bloku
  - Size - rozmiar tablicy.
- Przeszukaj listę rozpoczynającą się na pozycji i w tablicy
- Przykład: tablica ma 100 pozycji, 1000 buforów, lista ma 10 elementów.
- Oddzielna tablica dla każdej długości bufora

- Przy założeniu, że liczba elementów w tablicy jest ograniczona przez α*Size, gdzie α jest stałą średni koszt wyszukiwania jest stały (rzędu O(1)).

Listy LRU

- Każdy z buforów będących w tablicy mieszającej znajduje się na jednej z sześciu list LRU (Last Recently Used) ostatnio używany. Do organizacji listy służą pola b_next_free oraz b_prev_free. Istnieje sześć list rozróżnianych za pomocą wartości pola b_list. Są to:
  - BUF_CLEAN zwykły bufor
  - BUF_DIRTY bufor przeznaczony do zapisu.
  - BUF_LOCKED, BUF_LOCKED1 – trwa zapis bufora
  - BUF_SHARED bufor współdzielony
  - BUF_UNSHARED bufor przestał być współdzielony.
- Listy LRU wykorzystywane są do odzyskiwania buforów.
  - Jeżeli jądro zażąda bufora to jest on przesuwany na koniec listy
  - Odzyskiwane są bufory z początku listy

Lista wolnych buforów - freelist

- Zawiera bufory nie znajdujące się w żadnych kolejkach mieszających.
- Do jej organizacji służą również pola b_next_free oraz b_prev_free.
- Bufory na tej liście mają b_dev ustalone na B_FREE.
- Jeżeli system potrzebuje nowego bufora, to pobierany jest bufor z tej listy.
- Problem, gdy lista ta jest pusta !!! Linuks wtedy w kolejności:
  - Próbuje przydzielić pamięć nowym buforom (ale może brakować wolnej pamięci).
  - Przegląda listy lru i zwalnia bufory, dla których b_count==0 oraz BH_Dirty=0
  - Budzi wątek jądra bdflush, aby zapisać bufory brudne (BH_Dirty =1) i powtórzyć poprzedni krok.
  - Funkcja refill_free_list.
- Proces bdflush jest demonem jądra. Zapisuje on bufory z listy BUF_DIRTY na dysk, po zapisie kasując flagę BH_Dirty.

Funkcja getblk – pobranie bufora

```
struct buffer_head *getblk(kdev_t dev, int block, int size) {
   struct buffer_head *bh=Znajdź_w_tab_mieszaj(dev,block,size);
   if (bh) {
        bh->b_count++;
        if (!buffer_dirty(bh) && buffer_uptodate(bh)) {
             put_last_lru(bh);
        return bh;
   }
   bh=Przydziel_nowy_bufor(dev,block,size); // BH_Uptodate skasowane
   bh->b_count=1;
   return b;
}
```

- Mamy dwie możliwe sytuacje:
  - Trafienie (ang. hit) pamięci podręcznej. Bufor jest już w tablicy mieszającej – wystarczy go zwrócić.
  - Chybienie (ang. miss) pamięci podręcznej. Bufora nie ma w tablicy - należy przydzielić nowy.
- Zarządzanie listą LRU: bufor zawierający poprawne dane i nie wymagający zapisu przeniesiony na koniec. W przypadku braku pamięci bufory są odzyskiwane z początku

Funkcja bread – odczyt bufora

```
struct buffer_head * bread(kdev_t dev, int block, int size)
{
        struct buffer_head * bh=getblk(dev, block, size);
       if (buffer_uptodate(bh))
                return bh;    // Scenariusz 1
        ll_rw_block(READ, 1, &bh);
        wait_on_buffer(bh);
        if (buffer_uptodate(bh))
                return bh;  // Scenariusz 2

        brelse(bh);
        return NULL;          // Scenariusz 3

}
```

- Scenariusz 1. Bufor z aktualnymi danymi znaleziony w pamięci podręcznej.
- Scenariusz 2. Bufor z nieaktualnymi danymi lub nowo przydzielony
  - ll_rw_block: Dodaj żądanie odczytu do kolejki żądań urządzenia, jeżeli kolejka była pusta wywołaj procedurę strategii.
  - wait_on_buffer: Uśpij proces na kolejce b_wait. Proces jest budzony przez sterownik urządzenia blokowego (funkcja end_request)
- Scenariusz 3. Podobnie jak 2, ale odczyt się nie powiódł.
  - brelse zwalnia bufor

Zwolnienie bufora - funkcja brelse

```
void brelse(struct buffer_head * buf)
{
        wait_on_buffer(buf);

        set_writetime(buf, 0);
        refile_buffer(buf);

        if (buf->b_count)
                buf->b_count--;

}
```

- brelse == buffer release. Zasadniczo należy zmniejszyć licznik odniesień bufora.
- Bufor może być w danej chwili wczytywany (bądź zapisywany) na dysk.
  - Należy poczekać na zakończenie tej operacji.
-

Zwolnienie bufora - funkcja bforget

```
void bforget(struct buffer_head * buf)
{
        wait_on_buffer(buf);
        mark_buffer_clean(buf);
        clear_bit(BH_Protected, &buf->b_state);
        buf->b_count--;
        remove_from_hash_queue(buf);
        buf->b_dev = NODEV;
        refile_buffer(buf);
}
```

- bforget jest bardziej “brutalną” wersją brelse.
- Bufor jest usunięty z tablicy mieszającej i przemieszczony na listę buforów wolnych.
  - Ponowne żądanie odczytu bufora zostanie skierowane do urządzenia.
  - Użycie bforget ma sens wtw. jeżeli jesteśmy pewni, że bufor nie będzie potrzebny przez bardzoooo długi czas
- Uwaga !!! Kasowany jest bit BH_Dirty !!! bforget nie można używać w przypadku zmodyfikowanych buforów !!!

Odczyt bufora – funkcja breada

- Buffer Read Ahead

```
struct buffer_head * breada(kdev_t dev, int block, int bufsize,
   unsigned int pos, unsigned int filesize)
```

- Funkcja ta odczytuje bufor (pierwsze trzy argumenty takie jak w bread).
- Dodatkowo generuje żądanie odczytu dodatkowych bloków.
  - Funkcja czeka tylko na odczyt pierwszego bloku, nie czeka na odczyt pozostałych bloków.
  - Dodatkowe bloki będą miały licznik odniesień ustalony na zero.
- Liczba przeczytanych bloków jest obliczona tak, że filesize-pos ma być liczbą przeczytanych bajtów.
  - Jednakże liczba bloków nie może być większa od 16

Wykorzystanie pamięci podręcznej – scenariusz 1

- Chcemy przeczytać blok z dysku.

```
struct buffer head *bh=bread(device,buffer,size);

if (bh=NULL) Błąd();


// Teraz korzystamy z bufora, dane są w bh->b_data

// Zwalniamy bufor
brelse(bh);
// Teraz nie wolno korzystać danych
```

- Przy braku brelse bufor pozostałby na stale w pamięci
- Zamiast brelse można użyć bforget – wtedy bufor trafi na listę wolnych i ponowne wywołanie bread na pewno będzie wymagało odczytu z użądzenia

Wykorzystanie pamięci podręcznej – scenariusz 2

- Chcemy przeczytać blok z dysku, zmodyfikować i później zapisać

```
struct buffer head *bh=bread(device,buffer,size);

if (bh=NULL) Błąd();

// Teraz korzystamy z bufora, dane są w bh->b_data
// Modyfikujemy te dane

mark_buffer_dirty(bh, flag) // flag =1 5sekund, 0 - 30 sekund
// Zwalniamy bufor
brelse(bh);
// Teraz nie wolno korzystać danych
```

- Nie ma funkcji zapisującej bufor.
  - Bufor zaznaczamy jako zmodyfikowany.
  - System sam zadba o zapis (z opóźnieniem)

Wykorzystanie pamięci podręcznej – scenariusz 3

- Chcemy zapisać blok na dysku, bez uprzedniego czytania np. wypełnić jakiś blok zerami

```
// Przydziel (nie odczytaj blok)

struct buffer head *bh=getblk(device,buffer,size);

// Wypełniamy dane w bh->b_data

mark_buffer_uptodate(bh,1); // Bufor zawiera poprawne dane
mark_buffer_dirty(bh, flag) // flag =1 5sekund, 0 – 30 sekund
// Zwalniamy bufor
brelse(bh);
// Teraz nie wolno korzystać danych
```

Dygresja – wątki jądra (ang. kernel thread)

- Wątek jądra – proces “nie mający części użytkownika”
  - Nie posiada pamięci użytkownika – zatem wykonuje się w trybie jądra i jest częścią jądra.
  - Posiada własny task_struct i podlega planowaniu przez planistę jak zwykłe procesy.
  - Widoczny jest po wykonaniu polecenia ps.
- Tworzymy go przy pomocy funkcji:

```
pid_t kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

- Wątek jądra rozpoczyna się od funkcji fn.
- argument arg zostanie przekazany funkcji fn.
- Jeżeli wątek jądra zostanie stworzony z kontekstu zwykłego procesu (a nie innego wątku jądra), to pamięć tego procesu zostanie zwolniona dopiero wtedy, gdy wątek oraz proces zakończą pracę !!!

Wątek bdflush – zapis bloków na dysk

```
struct buffer_head *bh;

for(;;) {

    for some bh from BUF_DIRTY LRU list {

        // wstaw żądanie zapisu do kolejki urządzenia

        ll_rw_block(WRITE,bh,1);

        wake_up(&bdflush_done);

        interruptible_sleep_on(&bdflush_wait);

    }
```

- Jest on odpowiedzialny za zapis „brudnych" (zmodyfikowanych) buforów na dysk, w sytuacji, gdy takich buforów jest dużo.
- Algorytm wyboru liczby żądań zapisu do wygenerowania jest skomplikowany i nie będziemy go omawiać
  - Zbyt mało żądań -> brak wolnej pamięci, długie oczekiwanie na zapis, zmniejszenie wydajności systemu (trzeba pamięć podkradać procesom)
  - Zbyt dużo żądań -> duże obciążenie dysku zmniejszenie wydajności systemu.
- Drugi mechanizm – w postaci procesu (zwykłego demona) update jest odpowiedzialny za zapis brudnych stron długo pozostających w pamięci.

Budzenie wątku bdflush

```
static void wakeup_bdflush(int wait)
{
        if (current == bdflush_tsk)
                return;
         wake_up(&bdflush_wait);
         if (wait) {
                 run_task_queue(&tq_disk);
                 sleep_on(&bdflush_done);
         }
}
```

- Wątek bdflush czeka na kolejce bdfush_wait.
- Mamy opcję uśpienia procesu (kolejka zadań bdflush_done) , do momentu gdy bdflush zakończy pracę (Nie jest to równoważne zapisaniu buforów !!!).

Podsumowanie

- Jest oczywiste, że pamięć buforowa zwiększa przepustowość systemu.

- Pamięć buforowa wymaga dwukrotnego kopiowania danych.
  - Dane kopiowane są raz z urządzenia do bufora, a następnie z urządzenia do pamięci procesu.
  - Funkcja systemowa mmap pozwala na uniknięcie tego problemu.
  - Linuks (wersja 2.0.x) nie posiada mechanizmu surowego (ang. raw) wejścia-wyjścia pozwalającego na przesyłanie danych bezpośrednio pomiędzy urządzeniem a pamięcią procesu.

- Zastosowanie mechanizmu opóźnionych zapisów może prowadzić do problemów ze spójnością systemu plików w przypadku awarii