53. Algorytm przydzielania nowych bloków dyskowych w systemie ext2

Ewelina Grzesik i Jakub Wroniecki

20 grudnia 2000

Spis treści

1 Opis działania algorytmu ext2_new_block

1.1 Wstęp

Przydzielanie nowych bloków dyskowych w systemie ext2 realizowane jest przez algorytm ext2_new_block. Działanie tego algorytmu można podzielić na trzy najważniejsze zadania :

1. Znaleźć wolny blok dyskowy i przekazać jego numer procesowi wołającemu.
2. Uczynić to w taki sposób, aby zminimalizować fragmentację dysku
3. Pre-alokować bloki dla pliku żądającego nowego bloku

1.2 Źródła

Źródła algorytmu można znaleźć w pliku *fs/ext2/balloc.c*. Źródła procedury ext2_alloc_block() [1] można znaleźć w pliku *fs/ext2/inode.c*. Kod procedur obsługujących bitmapy zajętości bloków dyskowych jest również umieszczony w pliku *fs/ext2/balloc.c*.

1.3 Znajdowanie wolnego bloku

Przed przystąpieniem do szukania wolnego bloku dyskowego algorytm sprawdza, czy operacja taka jest możliwa. Możliwa jest tylko wtedy, gdy na dysku znajdują się jeszcze wolne bloki, oraz liczba bloków wolnych nie jest mniejsza od liczby bloków zarezerwowanych dla super-użytkownika (*root*) [2].

Przed rozpoczęciem poszukiwania wolnych bloków, ładowane są do bufora bitmapy zajetości bloków dyskowych (odzielna bitmapa dla każdej przeszukiwanej grupy). Wszystkie opisywane poniżej operacje wykonywane są na tym buforze. Ilość operacji dyskowych wykonywanych przez algorytm jest więc zminimalizowana do ilości przeszukiwanych grup [3].

Następnie sprawdza się czy wolny jest blok najlepiej pasujący do rozszerzanego pliku (numer tego bloku przekazywany jest jako parametr *goal*). Jeśli tak to blok ten zwracany jest jako wynik funkcji. Jeśli nie, szuka się wolnego bloku w bezpośrednim sąsiedztwie bloku *goal* (w odległości nie większej niż 32 bloki) i po znalezieniu takiego bloku zwraca się go jako wynik funkcji.

Gdy najbliższe otoczenie bloku o numerze *goal* jest zajęte, przeszukiwana jest reszta dysku, zaczynając od pozostałej części grupy, w której znajduje się blok *goal*, kończąc na przeszukaniu wszystkich pozostałych grup na dysku. Sposób w jaki wykonywane jest to szukanie opisany jest poniżej.

1.4 Minimalizacja fragmentacji dysku

Przeszukiwanie kolejnych grup dyskowych odbywa się tak, by jak najbardziej uspójnić obszary dysku zajmowane przez pliki. Aby to zapewnić, dla każdej grupy wykonywany jest następujący algorytm :

[1] Procedura wołana przez VFS, używająca ext2_new_block().

[2] Sprawdzenie quoty dla aktualnego użytkownika odbywa się dopiero po znalezieniu wolnego bloku.

[3] + jedna operacja na zapisanie zmian.

1. Szukanie pierwszych ośmiu wolnych bloków występujących jeden po drugim[4]. Gdy takie bloki istnieją - cofnięcie się maksymalnie 7 bloków, w celu znalezienia początku wolnego obszaru na dysku. Zwracany jest pierwszy blok wolnego obszaru.

2. Szukanie pierwszego wolnego bloku w grupie.

3. Zwrócenie znalezionego bloku, lub przejście do następnej grupy.

Oczywiste jest, że algorytm ten jest daleki od ideału, gdyż kolejne wywołania funkcji, *mogą* zwracać kolejne bloki dyskowe, tym samym prowadząc do fragmentacji dysku. Duże znaczenie odgrywa więc parametr *goal*, którego wartość jest wyznaczana poza opisywanym algorytmem. W praktyce, zapobieganie fragmentacji jest realizowane przez dwa mechanizmy: opisany wyżej algorytm i pre-alokację bloków.

1.5 Pre-alokacja bloków dyskowych

Pre-alokacja jest rodzajem rezerwacji bloków dla potrzeb pliku. Algorytm jej działania jest bardzo prosty : bezpośrednio po przyznanym bloku szuka się wolnych bloków, aż do natrafienia na blok zajęty, lub przekroczenia pewnej ilości[5].

Wyniki pre-alokacji zapisywane są w strukturze i-węzła, ale pre-alokowane bloki nie są zaznaczane jako zajęte na mapach bitowych zajętości bloków. Zajmowaniem pre-alokowanych bloków zajmuje się już funkcja *ext2_alloc_block()*, która wołana jest przez VFS. Funkcja ta sprawdza czy istnieje pre-alokowany blok, i jesli taki blok istnieje i jest wolny - przyznaje go konkretnemu plikowi. W przeciwnym przypadku - wołana jest funkcja *ext2_new_block()*. Taki mechanizm umożliwia zajmowanie poprzednio pre-alokowanych bloków przez inne pliki, gdy na dysku zaczyna brakować miejsca.

Pomimo ewidentnych korzyści płynących z pre-alokacji, nie zawsze jest ona stosowana. Na przykład, partycje na których przechowuje się duże ilości małych plików ($<$ wielkości jednego bloku) mogą wydajniej działać bez pre-alokowania.

1.6 Błędy zwracane przez funkcję

Możliwe jest oczywiście, że funkcja zostanie przerwana w wyniku wystąpienia błędu. Oto lista wszystkich możliwych błedów zwracanych przez *ext2_new_block()* :

- ENOSPC zwracane gdy na dysku zabrakło miejsca na przydzielenie nowego bloku.
- EDQUOT zwracane gdy przekroczona została quota właściciela pliku.
- EIO zwracane gdy nastąpił błąd wejścia/wyjścia podczas odczytywania/zapisywania danych na dysku.

[4] Szukanie 8 bloków jest b.łatwe do zaimplementowania - szuka się bajtu o wartości 0 na mapie bitowej.

[5] Wartość ustawiana indywidualnie dla każdej partycji, lub stała systemowa. Przeważnie równa 8.

W opisanych powyżej sytuacjach parametr *err* ustawiany jest na daną wartość, a wynikiem funkcji jest zero.

Jeżeli wywołanie funkcji się powiedzie, zwraca ona numer przyznanego bloku (> 0).

2 Algorytm krok po kroku

int ext2_new_block (const struct inode * inode, unsigned long goal,
u32 * prealloc_count, u32 * prealloc_block, int * err)

Funkcja przyjmuje nastepujące parametry

inode wskaźnik na i-węzeł pliku dla którego przydzielamy blok dyskowy

goal numer bloku, który funkcja powinna próbować przydzielić jako pierwszy

prealloc_count wskaźnik na zmienną przez ktorą funkcja zwróci ilość prealokowanych bloków

prealloc_block wskaźnik na zmienna przez która funkcja zwróci numer pierwszego prealokowanego bloku

err zmienna której wartość będzie ustawiona na kod błędu

Wartością zwróconą przez funkcję jest numer przyznanego bloku. Jeżeli wartość ta jest 0, oznacza to, że operacja przyznawania bloku się nie powiodła i zmienna *err* ustawiona jest na kod błędu.

```
{
        struct buffer_head * bh;
        struct buffer_head * bh2;
        char * p, * r;
        int i, j, k, tmp;
        int bitmap_nr;
        struct super_block * sb;
        struct ext2_group_desc * gdp;
        struct ext2_super_block * es;
#ifdef EXT2FS_DEBUG
        static int goal_hits = 0, goal_attempts = 0;
#endif
        *err = -ENOSPC;
        sb = inode->i_sb;[6]
        if (!sb) {
                printk ("ext2_new_block: nonexistent device");
                return 0;
        }
        lock_super (sb);[7]
        es = sb->u.ext2_sb.s_es;
        if (le32_to_cpu(es->s_free_blocks_count) <=
            le32_to_cpu(es->s_r_blocks_count) &&
            ((sb->u.ext2_sb.s_resuid != current->fsuid) &&
            (sb->u.ext2_sb.s_resgid == 0 ||
```

[6] Algorytm pobiera (do zmiennej sb) strukturę opisującą super-blok aktualnej grupy. Jeżeli operacja ta się nie powiedzie, wypisywany jest komunikat o błędzie i funkcja zwraca 0.

[7] Super-blok jest blokowany (procedura *lock_super*), aby zapobiec sytuacji, gdzie dwa procesy próbują jednocześnie alokować te same bloki. Jest to konieczne, gdyż wynikiem alokacji jest modyfikacja struktur dyskowych i pamięci.

```
        !in_group_p (sb->u.ext2_sb.s_resgid)) &&
        !capable(CAP_SYS_RESOURCE)))[8]
    {
        unlock_super (sb);
        return 0;
    }

repeat:[9]

    if (goal < le32_to_cpu(es->s_first_data_block) ||
        goal >= le32_to_cpu(es->s_blocks_count))[10]
            goal = le32_to_cpu(es->s_first_data_block);[11]
    i = (goal - le32_to_cpu(es->s_first_data_block)) /
            EXT2_BLOCKS_PER_GROUP(sb);[12]
    gdp = ext2_get_group_desc (sb, i, &bh2);[13]
    if (!gdp)
            goto io_error;

    if (le16_to_cpu(gdp->bg_free_blocks_count) > 0) {[14]
            j = ((goal - le32_to_cpu(es->s_first_data_block)) %
                EXT2_BLOCKS_PER_GROUP(sb));[15]
            bitmap_nr = load_block_bitmap (sb, i);[16]
            if (bitmap_nr < 0)
                    goto io_error;

            bh = sb->u.ext2_sb.s_block_bitmap[bitmap_nr];[17]

            ext2_debug ("goal is at %d:%d.\n", i, j);

            if (!ext2_test_bit(j, bh->b_data))[18] {
                    goto got_block;
```

[8] Powyższy warunek jest prawdziwy, gdy ilość wolnych bloków jest mniejsza od ilości bloków zarezerwowanych dla super-użytkownika i (oczywiście) aktualny użytkownik nim nie jest. W takiej sytuacji funkcja się kończy z kodem błędu.

[9] Etykieta pętli, wykonywanej do momentu zaalokowania bloku, lub wystąpienia błędu. Na samym początku zmienna *goal* ustawiona jest na numer bloku preferowanego przez proces wołający.

[10] Warunek ten sprawdza, czy *goal* nie wykracza poza bloki dostępne dla danych w aktualnej grupie.

[11] Jeżeli *goal* wykracza poza dostępne dla danych bloki, jest ustawiany na pierwszy z tych bloków.

[12] Obliczany jest numer grupy w której znajduje się blok o numerze *goal*. Potrzebny on będzie do załadowania deskryptora grupy.

[13] Ładowany jest deskryptor grupy, ktory z kolei wykorzystany będzie do załadowania odpowiedniej mapy bitowej zajętości bloków.

[14] Jeżeli są jeszcze wolne bloki w grupie *'i'*.

[15] Tu obliczany jest offset bloku '*goal*' w aktualnej grupie i przypisywany do zmiennej *j*.

[16] jest bitmapa zajętości bloków w aktualnej grupie. Bitmapa ta jest ładowana do bufora na pozycji *bitmap_nr*.

[17] Na zmienną *bh* przypisywany jest wskaźnik na interesującą nas bitmapę.

[18] Za pomocą funkcji *ext2_test_bit* sprawdzamy czy *j-ty* bit jest nieustawiony. Jezeli tak to nasze poszukiwania zakończyly się i skaczemy do *got_block*.

```
		}
		if (j) {[19]
			int end_goal = (j + 63) & ~63;[20]
			  j = ext2_find_next_zero_bit(bh->b_data, end_goal,
j);[21]
			if (j < end_goal)[22]
				goto got_block;
		}[23]


		[24]p = ((char *) bh->b_data) + (j >> 3);
		r = memscan(p, 0,
			(EXT2_BLOCKS_PER_GROUP(sb) - j + 7) >> 3);[25]
		k = (r - ((char *) bh->b_data)) << 3;
		if (k < EXT2_BLOCKS_PER_GROUP(sb)) {[26]
			j = k;
			goto search_back;[27]
		}

	      [28]k = ext2_find_next_zero_bit ((unsigned long *) bh->b_data,
					EXT2_BLOCKS_PER_GROUP(sb),
					j);
		if (k < EXT2_BLOCKS_PER_GROUP(sb)) {[29]
			j = k;
			goto got_block;
		}
	}

	/*
	 * Now search the rest of the groups.  We assume that
	 * i and gdp correctly point to the last group visited.
	 */
	for (k = 0; k < sb->u.ext2_sb.s_groups_count; k++) {[30]
```

[19] W tym miejscu wiadomo jest, że blok o numerze *goal* jest zajęty. W tej sytuacji algorytm szuka wolnego bloku w aktualnej grupie za *j-tym* blokiem w zakresie [*j..end_goal*].

[20] Wartosc *end_goal* jest ustalana w zasadzie pseudolosowo. Jedynym warunkiem jest, by nie przekraczała EXT2_BLOCKS_PER_GROUP.

[21] Znajdujemy następny zerowy bit pomiędzy bitami [*j..end_goal*]

[22] Jeżeli się udało to poszukiwania są zakończone

[23] W tym miejscu wiemy już, że blok *goal* jest zajęty i nie ma innych wolnych bloków w jego pobliżu. W takiej sytuacji algorytm przeszukuje resztę dysku wprzód, szukając najpierw całego wolnego bajtu (8 bloków) w mapach bitowych, a następnie wolnego bitu.

[24] Najpierw przeszukujemy pozostałe bloki aktualnej grupy

[25] Szukanie wolnego bajtu

[26] Znaleziono wolny bajt (8 wolnych bloków) w aktualnej grupie

[27] Ponieważ znaleziono wolny bajt, powinniśmy skoczyć do procedury *got_block*, ale aby zapobiec niepotrzebej fragmentacji wolnego miejsca na dysku, najpierw wywołujemy procedurę *search_back*, która cofa się maksymalnie 7 bitów, jeżeli są one wolne.

[28] Skoro się nie udalo znaleźć wolnego bajtu, szukamy wolnego bitu

[29] Jeżeli znaleziono wolny bit to kończymy poszukiwania

[30] Teraz przeszukujemy pozostałe grupy. Niezmiennikiem pętli jest warunek, że *i* i *gdp* pokazują na ostatnio przeszukiwaną grupę. Procedura jest identyczna jak dla grupy, w której znajduje się goal, poza częścią, gdzie przeszukiwane jest bezpośrednie otoczenie bloku

```
                i++;[31]
                if (i >= sb->u.ext2_sb.s_groups_count)
                        i = 0;
                gdp = ext2_get_group_desc (sb, i, &bh2);
                if (!gdp) {
                        *err = -EIO;
                        unlock_super (sb);
                        return 0;
                }
                if (le16_to_cpu(gdp->bg_free_blocks_count) > 0)
                        break;
        }
        if (k >= sb->u.ext2_sb.s_groups_count) {
                unlock_super (sb);
                return 0;
        }
        bitmap_nr = load_block_bitmap (sb, i);
        if (bitmap_nr < 0)
                goto io_error;

        bh = sb->u.ext2_sb.s_block_bitmap[bitmap_nr];
        r = memscan(bh->b_data, 0,
                   EXT2_BLOCKS_PER_GROUP(sb) >> 3);
        j = (r - bh->b_data) << 3;
        if (j < EXT2_BLOCKS_PER_GROUP(sb))
                goto search_back;
        else
                j = ext2_find_first_zero_bit ((unsigned long *) bh->b_data,
                                        EXT2_BLOCKS_PER_GROUP(sb));
        if (j >= EXT2_BLOCKS_PER_GROUP(sb)) {
                ext2_error (sb, "ext2_new_block",
                            "Free blocks count corrupted for block group %d", i);
                unlock_super (sb);
                return 0;
        }

    search_back:[32]
        for (k = 0; k < 7 && j > 0 && !ext2_test_bit (j - 1, bh->b_data);
k++, j–);

    got_block:[33]
```

goal. Najpierw szukamy wolnego bajtu i jeśli taki istnieje alokujemy 8 bloków (skok do search_back), a jeśli nie ma wolnego bajtu szukamy bitu (jeśli jest - skok do got_block).

[31] Następna grupa.

[32] Udało się znaleźć wolny bajt w bitmapie zajetości blokow. Teraz przesuwamy się maksymalnie 7 bitów wstecz, aby znaleźć początek tej grupy wolnych blokow i przechodzimy do *got_block*

[33] W tym miejscu j pokazuje na znaleziony wolny blok

```
        if(DQUOT_ALLOC_BLOCK(sb, inode, 1)) {[34]
                unlock_super(sb);
                *err = -EDQUOT;[35]
                return 0;
        }

        tmp = j + i * EXT2_BLOCKS_PER_GROUP(sb) +
                le32_to_cpu(es->s_first_data_block);[36]

        if (test_opt (sb, CHECK_STRICT) &&
            (tmp == le32_to_cpu(gdp->bg_block_bitmap) ||
             tmp == le32_to_cpu(gdp->bg_inode_bitmap) ||
             in_range (tmp, le32_to_cpu(gdp->bg_inode_table),
                     sb->u.ext2_sb.s_itb_per_group)))[37]
                ext2_panic (sb, "ext2_new_block",
                            "Allocating block in system zone - "
                            "block = %u", tmp);

        if (ext2_set_bit (j, bh->b_data)) {[38]
                ext2_warning (sb, "ext2_new_block",
                              "bit already set for block %d", j);
                DQUOT_FREE_BLOCK(sb, inode, 1);[39]
                goto repeat;
        }

        ext2_debug ("found bit %d\n", j);

#ifdef EXT2_PREALLOCATE[40]
        if (prealloc_block) {
                int     prealloc_goal;[41]

              prealloc_goal = es->s_prealloc_blocks ? es->s_prealloc_blocks
:
                              EXT2_DEFAULT_PREALLOC_BLOCKS;[42]

                *prealloc_count = 0;
                *prealloc_block = tmp + 1;
```

[34] Tu sprawdzamy, czy zajęcie tego bloku nie spowoduje przekroczenia quoty dyskowej przez aktualnego użytkownika. DQUOT_ALLOC_BLOCK to makro informujące system quot o alokacji bloku.

[35] Przekroczono quotę - wychodzimy z odpowiednim błędem

[36] Obliczany jest numer zaalokowanego bloku.

[37] Blok *tmp* jest blokiem systemowym, a nie blokiem danych. Nie powoduje to jednak przerwania działania algorytmu.

[38] Jednak coś nie poszło najlepiej i *j-ty* blok jest zajęty. Trzeba odkręcić wszystkie zmiany i spróbować jeszcze raz.

[39] Informujemy system quot o zwolnieniu bloku (przez makro DQUOT_FREE_BLOCK)

[40] Tu następuje prealokacja blokow, jeżeli została ustawiona taka opcja

[41] prealloc_goal = maksymalna ilosc blokow jaka chcemy zaalokowac

[42] Jezeli ustalona jest jakaś indywidualna liczba prealokowanych bloków to ją uwzględniamy, jeżeli nie - bieżemy systemową wartość domyślną.

```
        [43]for (k = 1; k < prealloc_goal &&
                (j + k) < EXT2_BLOCKS_PER_GROUP(sb);
                 k++) {
              if (DQUOT_PREALLOC_BLOCK(sb, inode, 1))
                    break;
              if (ext2_set_bit (j + k, bh->b_data)) {
                    DQUOT_FREE_BLOCK(sb, inode, 1);
                    break;
              }
              (*prealloc_count)++;
        }
        [44]gdp->bg_free_blocks_count =
             cpu_to_le16(le16_to_cpu(gdp->bg_free_blocks_count)
-
                    *prealloc_count);
        [45]es->s_free_blocks_count =
              cpu_to_le32(le32_to_cpu(es->s_free_blocks_count) -
                    *prealloc_count);
        ext2_debug ("Preallocated a further %lu bits.\n",
                  *prealloc_count);
  }
#endif[46]

  j = tmp;

  mark_buffer_dirty(bh, 1);[47]
  if (sb->s_flags & MS_SYNCHRONOUS) {[48]
        ll_rw_block (WRITE, 1, &bh);
        wait_on_buffer (bh);
  }

  if (j >= le32_to_cpu(es->s_blocks_count)) {[49]
        ext2_error (sb, "ext2_new_block",
                  "block >= blocks count - "
                  "block_group = %d, block=%d", i, j);
        unlock_super (sb);
        return 0;
  }
  if (!(bh = getblk (sb->s_dev, j, sb->s_blocksize)))[50] {
        ext2_error (sb, "ext2_new_block", "cannot get block %d", j);
```

[43]Prealokujemy bloki nastepujące po *tmp-ym* bloku, az napotkamy blok zajęty, lub zaprealokujemy *prealloc_goal* bloków. Przy okazji modyfikujemy dane systemu quoty.

[44]Uaktualniamy struktury dyskowe aktualnej grupy.

[45]Uaktualniamy struktury dyskowe całej partycji.

[46]Tu się kończy prealokacja.

[47]Oznaczamy bufor z mapą bitowa zajetości bloków jako '*dirty*' -> zmieniony.

[48]Uaktualniamy mapę bitową, jeśli zachodzi taka potrzeba, czekamy aż będzie gotowa do zapisu.

[49]Tu następuje ostatnie sprawdzenie, czy przeznaczony do alokacji blok jest w dozwolonym zasięgu. Jeżeli nie - kończymy funkcję z kodem błędu i komunikatem.

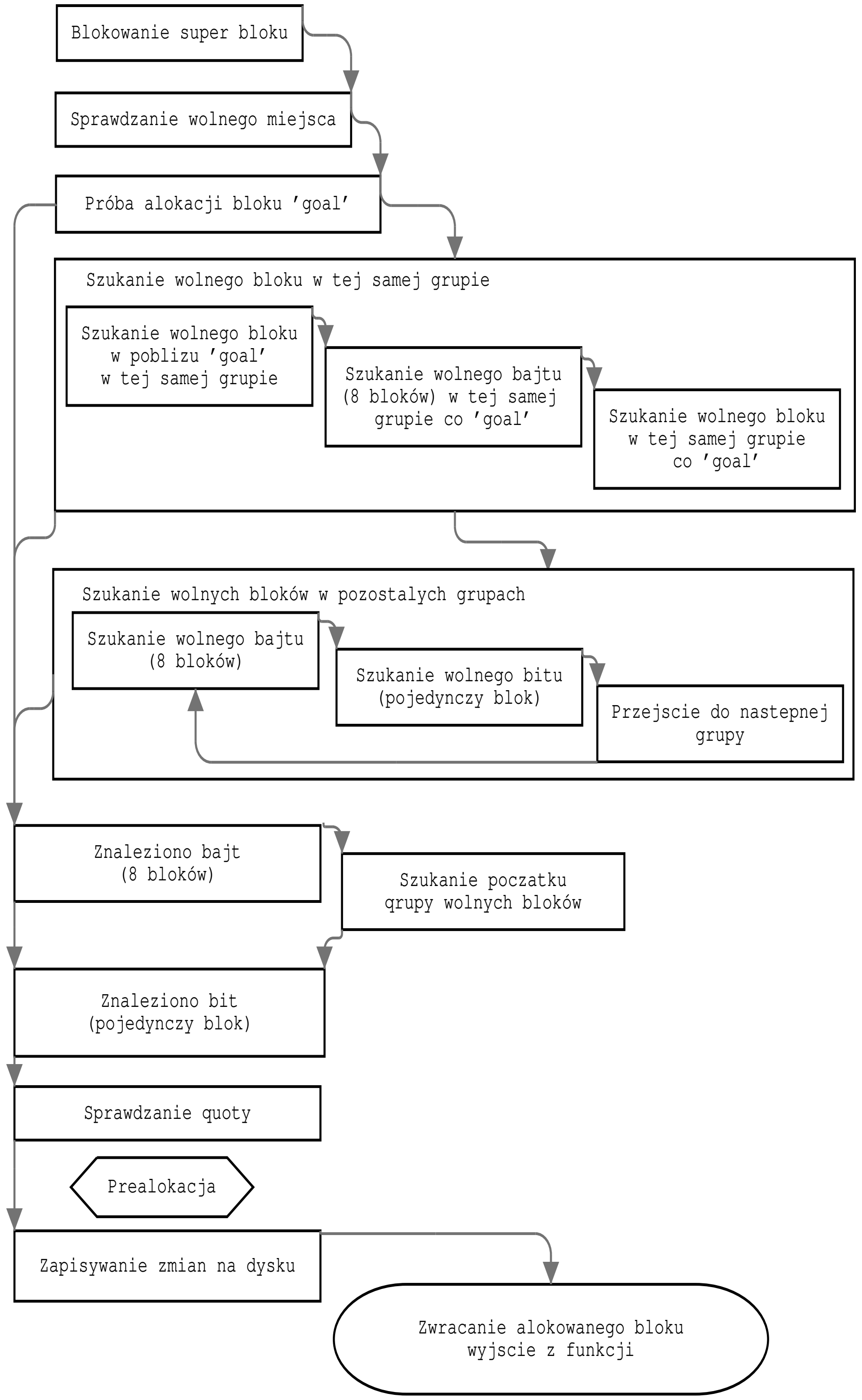
[50]Tutaj następuje fizyczne zaznaczenie bloku jako zajętego, bezpośrednio na tablicy partycji.

```
            unlock_super (sb);
            return 0;
        }
        if (!buffer_uptodate(bh))[51]
            wait_on_buffer(bh);
        memset(bh->b_data, 0, sb->s_blocksize);
        mark_buffer_uptodate(bh, 1);
        mark_buffer_dirty(bh, 1);
        brelse (bh);

        ext2_debug ("allocating block %d. "
                "Goal hits %d of %d.\n", j, goal_hits, goal_attempts);

        gdp->bg_free_blocks_count =[52]
                cpu_to_le16(le16_to_cpu(gdp->bg_free_blocks_count)
- 1);
        mark_buffer_dirty(bh2, 1);
       [53]es->s_free_blocks_count = cpu_to_le32(le32_to_cpu(es->s_free_blocks_count)
- 1);
        mark_buffer_dirty(sb->u.ext2_sb.s_sbh, 1);
        sb->s_dirt = 1;
        unlock_super (sb);
        *err = 0;[54]
        return j;

    io_error:
        *err = -EIO;
        unlock_super (sb);
        return 0;

    }
```

[51] W tym miejscu uaktualniany, zapisywany i zwalniany jest bufor z mapą bitową zajetości bloków.
[52] Zmniejszamy ilość wolnych bloków w aktualnej grupie.
[53] Zmniejszamy ilość wolnych bloków na dysku.
[54] Wszystko się powiodło, zwracamy j - zaalokowany blok

3 Uproszczony schemat działania algorytmu

3.1 Główna część algorytmu



3.2 Schemat mechanizmu pre-alokacji