\_\_GFP\_HIGH

\_\_GFP\_HARDER

enum {

ZONELIST\_FALLBACK,

};

\_\_zone\_watermark\_ok(z,order,mark,classzone\_idx,alloc\_flags,free\_pages)

{

bool alloc\_harder = (alloc\_flags&(ALLOC\_HARDER | ALLOC\_OOM));

if(alloc\_flags &ALLOC\_HIGH)

min -= min/2;

if(!alloc\_harder) {

}

}

zone\_watermark\_fast(z,order,mark,classzone\_idx,alloc\_flags)

{

}

## node\_zonelists

enum {

ZONELIST\_FALLBACK,

MAX\_ZONELISTS

};

struct pglist\_data {

struct zonelist node\_zonelists[MAX\_ZONELISTS];

};

start\_kernel->build\_all\_zonelists->\_\_build\_all\_zonelists->build\_zonelists

build\_zonelists(pgdat)

{

int nr\_zones;

nr\_zones = build\_zonerefs\_node(pgdat,zonerefs);

}

build\_zonerefs\_node(pgdat,zonerefs)

{

struct zone \*zone;

enum zone\_type = zone\_type = MAX\_NR\_ZONES;

do {

zone\_type - -;

zone = pgdat->node\_zones + zone\_type;

if(managed\_zone(zone)) {

zoneref\_set\_zone(zone,&zonerefs[nr\_zones++]);

check\_highest\_zone(zone\_type);

}while(zone\_type);

}while(zone\_type);

}

enum zone\_watermarks {

WMARK\_MIN ,

WMARK\_LOW,

WMARK\_HIGH,

};

enum migrate\_moe {

MIGRATE\_ASYNC,

};

struct address\_space \*page\_mapping(struct page \*page)

{

struct address\_space \*mapping;

mapping = page->mapping;

if((unsigned long)mapping &PAGE\_MAPPING\_ANON)

return NULL;

return (void \*)((unsigned long)mapping &~PAGE\_MAPPING\_FLAGS);

}

/\*

page\_mapcount:If the page can be mapped to userspace,encodes the number of times this page is referenced by a page table.

\*/

int page\_mapcount(page)

{

return atomic\_read(&page->\_mapcount)+1;

}

int page\_count(page)

{

return atomic\_read(&compound\_head(page)->\_refcount);  
}

isolate\_migrate\_t isolate\_migratepages(zone,cc)

{

unsigned long low\_pfn;

low\_pfn = fast\_find\_migrateblock(cc);

}

fast\_find\_migrateblock(cc)

{

if(cc->order <= PAGE\_ALLOC\_COSTLY\_ORDER)

return pfn;

}

isolate\_migratepages\_block(cc,low\_pfn,end\_pfn,isolate\_mode)

{

bool skip\_on\_failure =false;

unsigned long nr\_scanned =0,nr\_isoated =0;

unsigned long next\_skip\_pfn = 0;

bool skip\_updated = false;

bool locked = false;

struct page \*page = NULL,\*valid\_page = NULL;

for(;low\_pfn <end\_pfn;low\_pfn++) {

/\*

migration will fail if an anonymous page is pinned in memory.

\*/

if(!page\_mapping(page)&&

page\_count(page) > page\_mapcount(page))

/\*

Only allow to migrate anonymous pages in GFP\_NOFS context

because those do not depend on fs locks.

\*/

lruvec = mem\_cgroup\_page\_lruvec(page,pgdat);

if(\_\_isolate\_lru\_page(page,isolate\_mode)!=0)

goto isolate\_fail;

del\_page\_from\_lru\_list(page,lruvec,page\_lru(page));

inc\_node\_page\_state(page,

NR\_ISOLATED\_ANON+ page\_is\_file\_cache(page));

isolate\_success:

list\_add(&page->lru,&cc->migratepages);

cc->nr\_migratepages++;

nr\_isolated++;

if(cc->nr\_migratepages==COMPACT\_CLUSTER\_MAX&&

!cc->rescan&&!cc->contended)

{

++low\_pfn;

break;

}

continue;

}

}

\_\_isolate\_lru\_page(page,isolate\_mode)

{

int ret = -EINVAL;

if(!PageLRU(page))

return ret;

/\*

get\_page\_unless\_zero:if page \_refcount为0，则返回false；否则page \_refcount加1并返回true;

\*/

if(get\_page\_unless\_zero(page)) {

ClearPageLru(page);

ret =0;

}

return ret;

}

fragmentation\_index(zone,order)

{

unsigned long requeted = 1UL << order;

1000/info->free\_blocks\_total +(info->free\_pages\*1000ULL/requested)/info->free\_blocks\_total;

}

/\*

碎片剖析:总的page /(1<<order)，当前可以分配出来的order。

1000

\*/

\_\_compact\_finished(cc)

{

/\*

Compaction run completes if

\*/

if(compact\_scanners\_met(cc))

{

}

}

/\*

migrate\_pfn-> <-free\_pfn

[free][occupied][occuped][free] ……..[free][occupied][free]

\*/

bool compact\_scanners\_met(cc)

{

return (cc->free\_pfn >> pageblock\_order)

<= (cc->migrate\_pfn >> pageblock\_order);  
}

isolate\_free\_pages()

struct zone {

unsigned long compact\_cached\_free\_pfn;

unsigned long compact\_cached\_migrate\_pfn[2];

unsigned long compact\_init\_migrate\_pfn;

unsigned long compact\_init\_free\_pfn;

};

\_\_compact\_finished(cc)

{

}