# 1 Treść zadania

Zmienić domyślny algorytm przydziału pamięci w systemie minix na dający możliwość wyboru pomiędzy *worst fit* a *first fit*. Zademonstrować i zinterpretować różnice w działaniu algorytmów.

# 2 Założenia

Modyfikowane są tylko następujące pliki:

- `/usr/src/mm/proto.h`
- `/usr/src/mm/alloc.c`
- `/usr/src/mm/table.c`
- `/usr/src/fs/table.c`
- `/usr/include/minix/callnr.h`

# 3 Realizacja

## 3.1 Funkcje systemowe

Aby zaimplementować funkcje systemowe konieczne były następujące zmiany:

- `mm/table.c` - oraz `fs/table.c` dodanie funkcji do tablicy
- `mm/proto.h` - dodanie prototypów funkcji w sekcji `alloc.c`
- `/usr/include/minix/callnr.h` - zwiększenie tablicy wywołań
- `mm/alloc.c` - dodanie kodu obu funkcji na końcu pliku

### 3.1.1 hole_map

Funkcja zapisuje do bufora adres i wielkość kolejnych bloków pamięci, po czym kopiuje je do zadanego bufora za pomocą systemowej funkcji `sys_copy`.

```
int do_hole_map(){

phys_clicks buffer[1024];
struct hole * hp;
int i = 0;
int counter;

int nbytes = mm_in.m1_i1;
char * usr_buff = mm_in.m1_p1;
int usr_pid = mm_in.m_source;

hp = hole_head;
```

```
if(hp == NIL_HOLE) return -1;

if(nbytes%2==0) nbytes-=2; /* sprawdz czy jest miejsce dla 0 */
counter = nbytes/2;

do{
buffer[i++]=hp->h_len; /* dlugosc */
buffer[i++]=hp->h_base; /* adres */
hp=hp->h_next; /* nastepny */
counter--;
}
while(hp != NIL_HOLE && counter > 0);
buffer[i]=0; /* dopisz zero */
sys_copy(0,D,(phys_bytes) buffer, usr_pid, D, (phys_bytes) usr_buff, nbytes);
return 0;

}
```

#### 3.1.2 worst_fit

Funkcja ustawia wartość zmiennej `worstFit`

```
int do_worst_fit(){
if(mm_in.m1_i1 == 0 || mm_in.m1_i1 == 1)
worstFit = mm_in.m1_i1;
return mm_in.m1_i1;
}
```

### 3.2 Algorytm worst fit

Implementacja polegała na modyfikacji funkcji `alloc_mem` w pliku `alloc.c` i modyfikacji istniejącego algorytmu przydzielania pamięci w następujący sposób: Dodania warunku, który w zależności od wartości zmiennej `worstFit` wybierał odpowiedni algorytm Realizacji algorytmu *worst fit*, który przeszukiwał wszystkie dostępne bloki pamięci i porównywał je z największym jaki dotąd znalazł. Po dotarciu do końca listy, zwracał największy znaleziony blok i zmniejszał jego długość i początkowy adres w liście (analogicznie dla istniejącego już algorytmu).

```
register struct hole * max_hole, *prev_max_hole;
if(hp!=NIL_HOLE){ /* jest miejsce? */
max_hole = hp;
prev_max_hole = NIL_HOLE;
}
else{
return(NO_MEM); /* nie ma */
}

while(hp!=NIL_HOLE){
if(hp->h_len > max_hole->h_len){
prev_max_hole = max_hole;
```

```
max_hole = hp;
}
hp = hp->h_next;
} /* while */
if(max_hole->h_len >= clicks){
old_base = max_hole->h_base;
max_hole->h_base += clicks;
max_hole->h_len -= clicks;

if(max_hole->h_len == 0 && prev_max_hole!=NIL_HOLE)
del_slot(prev_max_hole, max_hole);
return(old_base);
}
```

# 4 Testy

Poprawność wykonywanego zadania przetestowałem za pomocą programów przygotowanych w skrypcie laboratoryjnym: `t.c`, `w.c`, `x.c` oraz `skrypt_testowy`.

# 5 Wnioski

Wyniki działania programu `skrypt_testowy` są zgodne z przewidywaniami teoretycznymi. Po ich przeanalizowaniu widoczne stają się różnice w działaniu dwóch algorytmów.

W przypadku algorytmu *first fit* kolejno uruchamiane procesy programu `x`, zajmują pamięć z pierwszego wystarczająco dużego, wolnego bloku, a po 10 sekundach, kiedy pierwszy uruchomiony program kończy działanie, po kolei zwalniają zajęte bloki pamięci, co widoczne jest na listingu.

Przy użyciu algorytmu *worst fit*, widoczne jest, że każdy kolejny uruchomiony program `x`, zajmuje blok pamięci będący częścią ostatniego wolnego bloku, ktory jest największy.