

Progetto di sistemi operativi

Transito navale di merci

Bassignana Francesco - numero matricola: 977217 - Turno T1

Kostadinov Iliyan - numero matricola: 979118 - Turno T2

24 gennaio 2023

Indice:

1. Shared memory e strutture coinvolte	3
2. Distribuzione delle risorse:	4
3. Gestione merce scaduta:	6
4. Algoritmo carico/scarico delle navi:	7
5. Dump:	9
6. Organizzazione Meteo:	10
7. Librerie di supporto:	11
8. Master	15

Processi principali utilizzati:

- Processo MASTER
- Processo NAVE
- Processo PORTO
- Processo METEO

Scelte progettuali :

- *Metodi per mantenere in shared memory i porti e le navi*
- *Distribuzione delle risorse*
- *Il dump*
- *Strutture delle navi e dei porti*
- *Organizzazione del meteo*
- *Algoritmo carico/scarico delle navi*
- *Librerie di supporto*
- *Gestione merce scaduta*

1. Shared memory e strutture coinvolte

Struttura Porto:

```
struct port {  
  
    // int requestsID;  
    int requests[SO_MERCI];  
    unsigned short swell;  
    unsigned short weatherTarget;  
    Supplies supplies;  
    double x;  
    double y;  
    int deliveredGoods;  
    int sentGoods;  
};  
  
typedef struct supplies {  
  
    int magazine[SO_DAYS*SO_MERCI];  
    int expirationTimes[SO_DAYS * SO_MERCI];  
}Supplies
```



```
struct port {  
  
    int requestsID;  
    unsigned short swell;  
    unsigned short weatherTarget;  
    Supplies supplies;  
    double x;  
    double y;  
    int deliveredGoods;  
    int sentGoods;  
};  
  
typedef struct port* Port;  
  
typedef struct supplies {  
  
    int magazineID;  
    int expirationTimesID;  
}Supplies;
```

Abbiamo avuto la necessità di sostituire tutte le strutture array con degli ID che si riferiscono alla struttura IPC in shared memory corrispondente.

Ogni volta che si deve leggere/scrivere da *domande/offerte/expiration-times* di un determinato porto, si deve fare l'attach e la detach della risorsa identificata dal suo ID;

Struttura Nave:

```
struct ship {  
    int shipID;  
    double x;  
    double y;  
  
    int pid;  
    int weight;  
    loadShip loadship;  
    unsigned short storm;  
    unsigned short dead;  
    unsigned short weatherTarget;  
    unsigned short inSea;  
    int nChargesOptimal; /* da rimuovere*/  
};  
typedef struct ship* Ship;
```

```
struct productNode_ /* nodo utilizzato nella lista */  
{  
    int product_type; /* tipo di merce nella lista */  
    int weight;  
    int expirationTime;  
    int distributionDay;  
    int portID;  
    struct productNode_* next;  
};  
typedef struct productNode_* Product;  
  
struct load /* lista implementata per il carico della nave */  
{  
    Prod* head;  
    Prod* tail;  
    int load::length  
};  
int length;  
};  
typedef struct load* loadShip
```

Ci siamo permessi di aggiungere una lista dinamica alla struttura della nave in shm perché non serve che sia visibile da altri processi.

2. Distribuzione delle risorse:

Ogni porto al giorno 0 della simulazione dispone di $\text{SO_FEEL}/\text{SO_DAYS} + \text{il resto della divisione fatta di risorse}$. Mentre dal giorno 1 in poi dispone di $\text{SO_FEEL}/\text{SO_DAYS}$ risorse considerando solo la parte intera della divisione.

Questo perchè SO_FILL potrebbe non essere divisibile per SO_DAYS e quindi in questo modo, la somma del numero di merci generate ogni giorno ammonterebbe a SO_DAYS .

Esempio: SO_FILL: 10 SO_DAYS: 3

```
intList* distributeV1(int quantity, int parts) {
    int media;
    int scarto;
    intList* l;
    int i;
    media = quantity / parts;
    scarto = media / 5; /*per avere un coefficiente di variazione massimo del 20% (1/5 della media)*/

    l = intInit();
    for (i = 0; i < parts - 1; i++) {
        intPush(l, random_int(media - scarto, media + scarto));
    }
    /*la quantità restante viene messa nell'ultimo elemento*/
    intPush(l, quantity - sum(l));
    return l;
}
```

Al giorno 0 i porti hanno (3+1) quantità di merce richiesta, mentre al giorno 1 e 2 ne hanno 3.

```
//ES: SO_PORTI=3, SO_FILL/SO_DAYS=30, DAY=1
intList* risorsePorti = distributeV1(30,3);
//Esempio: per distribuire la merce tra gli slot dei tipi di merci:
/*
ES: il porto 0 deve distribuire le risorse giornaliere che gli
sono state assegnate negli slot del giorno di distribuzione corrispondente
*/
int quantitaGiorno1porto0 = elementAt(risorsePorti,0);
intList* richiesteGiorno1 = distributeV1(quantitaGiorno1porto0,SO_MERCI);
```

Esempio per distribuire la merce tra i porti:

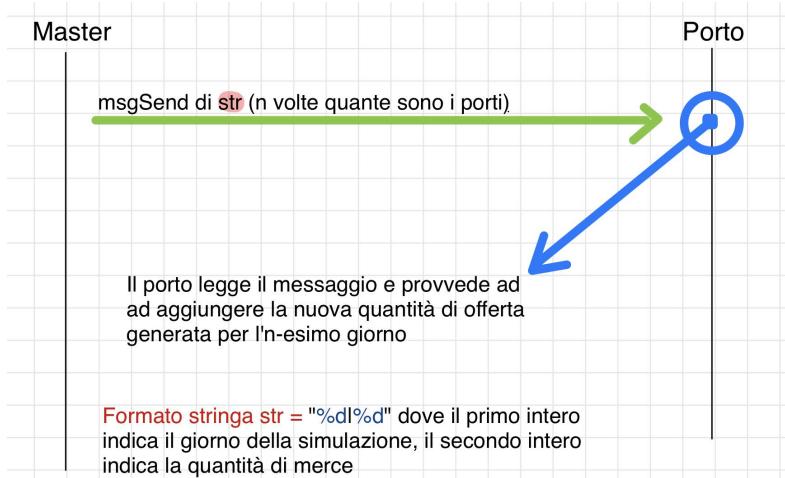
Risultato

```
[17251]Risorse porto 0:
DOMANDE:
2975,
0,
0,
0,
0,
SUPPLIES:
GIORNO 0: [ 183, 231, 215, 181, 197, ]
GIORNO 1: [ -1, -1, -1, -1, -1, ]
GIORNO 2: [ -1, -1, -1, -1, -1, ]
GIORNO 3: [ -1, -1, -1, -1, -1, ]
GIORNO 4: [ -1, -1, -1, -1, -1, ]
GIORNO 5: [ -1, -1, -1, -1, -1, ]
GIORNO 6: [ -1, -1, -1, -1, -1, ]
GIORNO 7: [ -1, -1, -1, -1, -1, ]
GIORNO 8: [ -1, -1, -1, -1, -1, ]
GIORNO 9: [ -1, -1, -1, -1, -1, ]
END TIME=
```

“-1” perché quello spazio è ancora vuoto/da riempire, non è ancora arrivato quel giorno.

Le domande sono distribuite a partire dalla quantità SO_FILL/SO_PORTI (e non SO_FILL/SO_DAYS/SO_PORTI anche se ovvio).

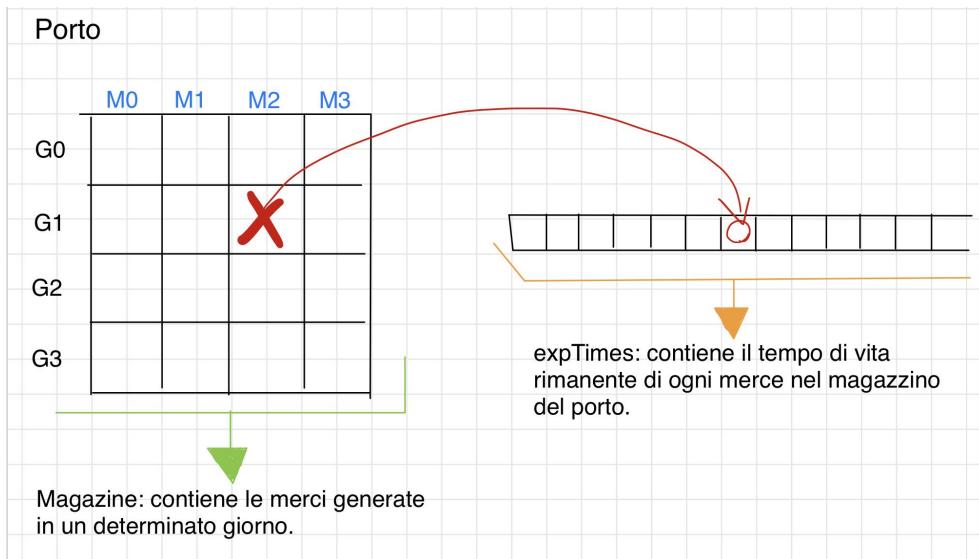
Dal giorno 1 in poi il master si serve di una message queue per comunicare a tutti i porti della nuova quantità di merce generata, questo compito è attribuito ad un processo figlio del master. Il porto invece anche lui utilizza un child che ascolta di continuo sulla coda di messaggi, quando arriverà il messaggio, esegue una funzione la quale ha il compito di rifornire i magazzini dei porti.



3. Gestione merce scaduta:

Ogni porto ha un array lungo (`SO_DAYS*SO_MERCI`) chiamato `expTimes` tale per cui `expTimes[i] = data di scadenza del lotto magazine[i/SO_MERCI][i%SO_MERCI]` (ogni lotto `magazine[k][j]` ha come data di scadenza `expTimes[(SO_MERCI * k) + j]`)

Ogni giorno il master decrementa i giorni rimanenti della merce che fino ad allora è stata distribuita e quando arriva a 0, viene azzerata anche la quantità di merce



corrispondente.

Come gestisce la merce scaduta la nave?

A riguardo la nave può fare due cose:

1. Interrompere un'azione in caso vedesse che il prodotto coinvolto è scaduto
2. Eliminare tutta la merce scaduta dal suo carico

Come fa la nave a sapere quando scade un prodotto?

Ogni prodotto della nave memorizza anche il porto dal quale è stato offerto, il giorno in cui è stato distribuito, e il tipo di merce ovviamente. Così facendo ogni volta che vuole sapere che è scaduto va a vedere la data di scadenza del porto in cui è stato distribuito che corrisponde a quel tipo di merce e a quel giorno di distribuzione.

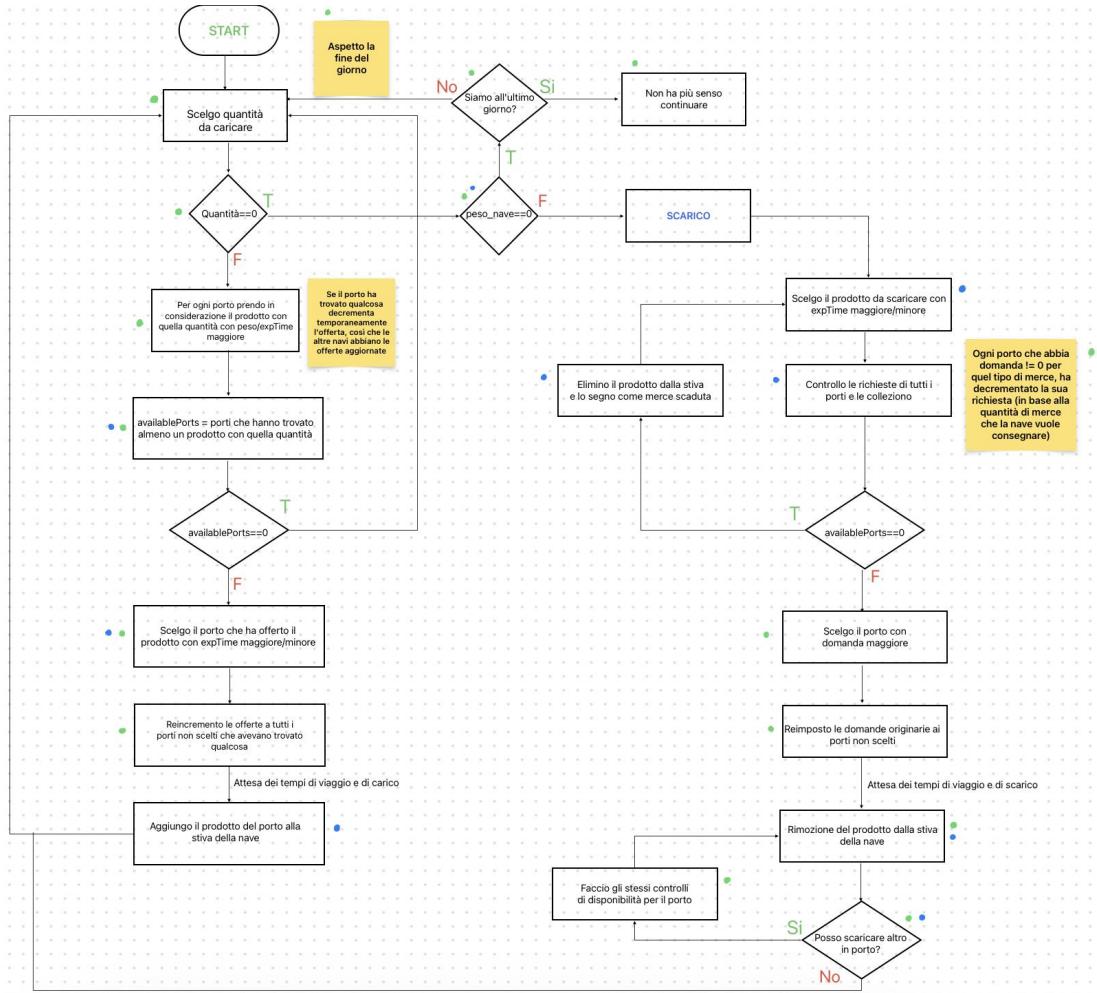
4. Algoritmo carico/scarico delle navi:

Carico:

- 1) Nave sceglie la quantità da richiedere, essa è sempre il $\min\{A,B\}$ dove
 - a) A = massima quantità offerta da un qualsiasi porto di cui esiste anche la richiesta in altri porti
 - b) B = capacità rimanente della nave

Così facendo se la quantità da richiedere è = 0 vuol dire che non c'è più offerta di tipi di merce che ha senso consegnare, oppure che la nave è piena

Se la quantità è 0 allora la nave **scarica**

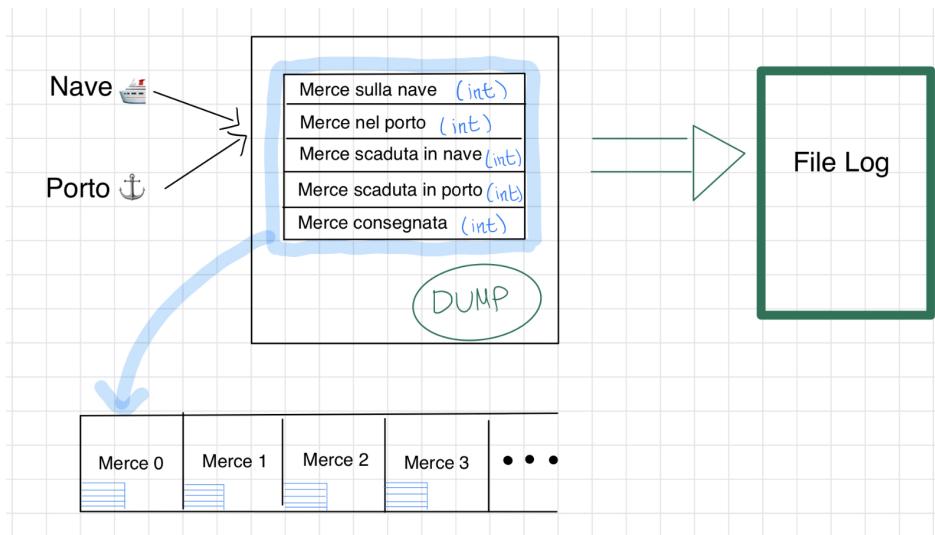


Pallino verde: progettato da Bassignana

Pallino blu: progettato da Kostadinov

5. Dump:

Dump è una shared memory utilizzata dai processi nave e porto per segnalare lo stato delle proprie risorse.



Risultato :

```
-----Day 0 -----  
✓ Tipo merce 0:  
✓   - Non scaduta:  
✓     a) nei porti: 1045  
✓     b) in nave: 0  
✓   - Scaduta:  
✓     a) nei porti: 0  
✓     b) in nave: 0  
✓   - Consegnata: 0  
✓ Tipo merce 1:  
✓   - Non scaduta:  
✓     a) nei porti: 1124  
✓     b) in nave: 0  
✓   - Scaduta:  
✓     a) nei porti: 0  
✓     b) in nave: 0  
✓   - Consegnata: 0  
✓ Tipo merce 2:  
✓   - Non scaduta:  
✓     a) nei porti: 1043  
✓     b) in nave: 0  
✓   - Scaduta:  
✓     a) nei porti: 0  
✓     b) in nave: 0  
✓   - Consegnata: 0  
✓ Tipo merce 3:  
✓   - Non scaduta:  
✓     a) nei porti: 976  
✓     b) in nave: 0  
✓   - Scaduta:  
✓     a) nei porti: 0  
✓     b) in nave: 0  
✓   - Consegnata: 0  
✓ Tipo merce 4:  
✓   - Non scaduta:  
✓     a) nei porti: 812  
✓     b) in nave: 0  
✓   - Scaduta:  
✓     a) nei porti: 0  
✓     b) in nave: 0  
✓   - Consegnata: 0
```

```
typedef struct goodTypeInfo {  
    int goods_on_ship;  
    int goods_on_port;  
    int delivered_goods;  
    int expired_goods_on_ship;  
    int expired_goods_on_port;  
} GoodTypeInfo;  
  
typedef struct dumpArea {  
  
    int typesInfoID;  
    double expTimeVariance;  
    double tempoScaricamentoTot;  
} DumpArea;
```

Tramite le seguenti procedure si possono aggiornare i diversi campi del dump, per ogni tipo di merce presente:

```
void addExpiredGood(...);  
void addNotExpiredGood(...);  
void addDeliveredGood(...);
```

e poi con la funzione printDump() si stampa tutto quanto su un file asincronicamente.

6. Organizzazione Meteo:

```
void launchStorm() {
    int pid = fork();
    if(pid == -1){
        throwError("fork in launchStorm", "launchStorm");
        exit(EXIT_FAILURE);
    } else if(pid == 0){
        stormRoutine();
        exit(EXIT_SUCCESS);
    }
}
```

Periodicamente il processo Meteo lancia una tempesta tramite il metodo launchStorm(), forkandosi e assegnando al figlio il compito.

```
void launchSwell(){
    int pid = fork();
    if(pid == -1){
        throwError("fork in launchSwell", "launchSwell");
        exit(EXIT_FAILURE);
    } else if(pid == 0){
        swellRoutine();
        exit(EXIT_SUCCESS);
    }
}
```

Stesso discorso per il lancio dell'evento mareggiata.

```
void malestormHandler() {
    int pid;
    pid = fork();

    if (pid == -1) {
        throwError("fork in launchStorm", "malestormHandler");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        malestormRoutine();
        exit(EXIT_FAILURE);
    }
}
```

In questo caso il processo figlio del meteo (colui che si occupa del malestorm) resta in vita per tutta la durata dell'esecuzione.
Stessa cosa avviene con le mareggiate, launchSwell().

Quando le navi o i porti vengono *colpiti da un evento atmosferisco*, nelle loro strutture dati si imposta un valore (nel nostro caso 1) che indica il coinvolgimento di quest'ultime in una *tempesta/malestorm* (per le navi) oppure una *mareggiata* (per i porti).

```
struct port {
    int requestsID;
    unsigned short swell;
    unsigned short weatherTarget;
    Supplies supplies;
    double x;
    double y;
    int deliveredGoods;
    int sentGoods;
};
```

Se il porto è stato colpito da una mareggiata, nel campo swell viene inserito il valore 1.

```

struct ship {
    int shipID;
    double x;
    double y;

    int pid;
    int weight;
    loadShip loadship;
    unsigned short storm;
    unsigned short dead;
    unsigned short weatherTarget;
    unsigned short inSea;
};

typedef struct ship* Ship;

```

Se la nave è stata colpita da una tempesta, nel campo `storm` viene inserito il valore 1.

Osservazione: seppur il campo di entrambe le strutture, `weatherTarget`, possa essere ambiguo alla gestione meteo, viene utilizzato per un altro scopo (utile al master per il dump).

7. Librerie di supporto:

(msg/sem/shm)_utility.c

Sono degli insiemi di funzioni utili che servono per semplificare l'uso delle classiche syscall degli oggetti ipc, di semplificato hanno:

1. Gestione degli errori (spiegerò nella sezione della gestione degli errori)
2. Eliminazione della ripetizione di codice per i parametri

```

mex* msgRecv(int msgqID, long type, void (*errorHandler)(int err, char* errMsg), void (*callback)(long type, char text[MEXBSIZE]), int mod, char* errMsg) {
    mex* m = (mex*)malloc(sizeof(mex));
    if (msgrecv(msgqID, m, MEXBSIZE, type, 0) == -1) {
        if (errorHandler != NULL) {
            errorHandler(MERRRCV,errMsg);
            exit(EXIT_FAILURE);
        }
        else {
            throwError("msgRecv -> msgrecv", "msgRecv");
            exit(EXIT_FAILURE);
        }
    }

    if (mod == SYNC) {
        return m;
    }
    else if (mod == ASYNC) {
        int pid = fork();

        if (pid == -1) {
            if (errorHandler == NULL) {
                throwError("msgRecv -> fork" , "msgRecv");
                exit(EXIT_FAILURE);
            }
            else {
                errorHandler(errno, errMsg);
                exit(EXIT_FAILURE);
            }
        }
        if (pid == 0) {

            callback(m->mtype, m->mtext);
            free(m);
            exit(EXIT_SUCCESS);
        }
        else {
            free(m);
        }
        return NULL;
    }
    else {
        fprintf(stderr, "mod must be SYNC or ASYNC\n");
        exit(EXIT_FAILURE);
    }
}

```

errorHandler.c

Questo file contiene 2 funzioni:

1. `errorHandler()`: viene passata come argomento opzionale a tutte le funzioni che abbiamo fatto per gli oggetti ipc, a seconda del tipo di errore che ha scaturito la funzione in questione al suo interno, chiama `errorHandler()` passandole un identificatore dell'errore incontrato.
Quando `errorHandler()` viene eseguita a seconda dell'identificatore dell'errore incontrato chiama `throwError()` con un messaggio di errore dedicato all'identificatore.
2. `throwError()`: chiamata all'interno di `errorHandler()` (e non); scrive sul file di log degli errori l'errore in questione specificando il contesto dell'errore, l'operazione che ha scaturito l'errore e il messaggio di errno.

```
void throwError(char* myerr, char* errCtx) {
    int semid;
    FILE *fp;
    int hash = rand();
    semid = useSem(ERRFILESEMID, errorHandler, "throwError");
    mutex(semid, LOCK, errorHandler, "LOCK throwError");
    printf("XXXXXXXXXXXXXX HASH %d\n", hash);
    fp = fopen("./logs/errorLog.log", "a+");
    if (fp == NULL) {
        perror("fopen throwError");
        exit(1);
    }
    fprintf(fp, "*****\n");
    fprintf(fp, "ERROR: %s error handler\nERRNO: %s\nCTX: %s\nHASH: %d\n", myerr, strerror(errno), errCtx, hash);
    fprintf(fp, "*****\n");
    fclose(fp);
    mutex(semid, UNLOCK, errorHandler, "UNLOCK throwError");
}

void errorHandler(int err, char* errCtx) {
    switch (err) {
        case SERRCTL:
            throwError("sem ctl", errCtx);
            break;
        case SERRGET:
            throwError("sem get", errCtx);
            break;
        case SERROP:
            throwError("sem op", errCtx);
            break;
        case MERRCTL:
            throwError("msg ctl", errCtx);
            break;
    }
}
```

support.c

Funzioni utili che non riguardano entità particolari come la nave o il porto

```
double mediaTempoViaggioFraPorti() {
    long c;
    int i;
    int j;
    double sum = 0;
    int so_porti = SO_("PORTI");

    Port p = getPort(0);
    c = 0;
    for (i = 0; i < so_porti-1; i++) {
        for (j = i + 1; j < so_porti; j++) {

            sum += getTempoDiViaggio(p[i].x, p[i].y, p[j].x, p[j].y);
            c++;
        }
    }
    detachPort(p, 0);
    return sum / c;
}
```

(port/ship/master)_utility.c e supplies.c

Funzioni utili a tutti i compiti presi in carico dalle entità porto, nave e master rispettivamente.

(supplies per le offerte dei porti)

Es:

```
intList* tipiDiMerceRichiesti(Port p) {
    intList* ret;
    int* reqs;
    int so_merci = SO_("MERCIA");
    reqs = getShmAddress(p->requestsID, 0, errorHandler, "tipiDiMerceRichiesti");
    ret = findIdxs(reqs, so_merci, filterIdxs);
    shmDetach(reqs, errorHandler, "tipiDiMerceRichiesti");
    return ret;
}
```

loadShip.c

Qui troviamo le funzioni utili a manipolare la lista rappresentante le merci trasportate sulla nave, in poche parole il suo carico.

```
loadShip initLoadShip();
Product findProduct(loadShip list, int product_type);
int getProductID(loadShip list, int product_type);

void printLoadShip(loadShip list, FILE* stream);

void freeLoadShip(loadShip list);
Product productAt(loadShip l, int idx);
Product initProduct(int weight, int type, int expTime, int portID, int dd);
int weightSum(loadShip l);
```

supplies.c

In supplies.c abbiamo funzioni comode per gestire il magazzino dei porti (ricordiamo struttura a matrice) e l'array con le date di scadenza.

```
/*
 *      riempie di date di scadenza casuali il vettore di date di scadenza
 */
void fillExpirationTime(Supplies* S);

/*
 *      riempie di risorse il magazzino alla riga della matrice corrispondente all'indice
 */
void fillMagazine(int* magazine, int day, int* supplies);

/*
 *      dato il tipo della merce (indice colonna) e il giorno in cui è stata distribuita (
 */
int getExpirationTime(Supplies S, int tipoMerce, int giornoDistribuzione);
```

Queste erano solo alcune delle funzioni, giusto come esempio.

vettorInt.c

Funzioni fondamentali in molti casi per avere array dinamici

Es:

```
intList* intFindAll(intList* l, int(*f)(int el, int idx)) {
    intNode* aux = l->first;
    intList* lRet = intInit();
    int idx = 0;
    while (aux != NULL) {
        if (f(aux->numero, idx)) {
            intPush(lRet, aux->numero);
        }
        aux = aux->next;
        idx++;
    }

    return lRet;
}
```

8. Master

```
int main(){
    configuredMaster(codiceMaster);
}

void codiceMaster(...vari id delle risorse ipc configurate){
/*
... aspetta che tutti abbiano finito di configurarsi tramite semafori...
... da il via a tutti i processi per partire contemporaneamente tramite semaforo...
*/
while(1){
    /*routine giornaliera del master*/
    refillPorts();
    printDump();
    //ecc...
}
}
```

All'avvio del programma il Master inizia subito con la creazione di tutte le strutture IPC utilizzate durante l'esecuzione, in particolare si occupa di: creare semafori e inizializzarli, creare una coda di messaggi, creare le memorie condivise. Terminato ciò procede al lancio dei processi Nave, Porto e Meteo. Il Master attende che tutti i processi siano configurati e pronti per iniziare la simulazione e successivamente *“da il via libera”* per iniziare la loro esecuzione (tramite l'utilizzo di opportuni semafori).

Ogni giorno passato (1 secondo) il processo Master si occupa di fare i print necessari a visualizzare lo stato generale della simulazione, rifornire i porti con nuove quantità di merci e a decrementare le date di scadenza di quelle già presenti.

```
void configuredMaster(void(*codiceMaster)(...parametri)){
    ...
codiceMaster(...parametri configurati);
...
}
```