# OSMP Runner – Technische Dokumentation

Projekt: Betriebssysteme Praktikum – OSMP

Autor: Konrad Skwarski, Christopher Jung, Erik Kolke

Datum: 2025-04-02

Dateien: osmpRun.c, osmpRun.h

Ziel: Initialisierung und Ausführung einer verteilten, message-passing-basierten IPC-Laufzeitumgebung unter UNIX

1. Zweck des Programms

Der osmpRunner ist ein Initialisierungs- und Steuerprogramm, das eine vordefinierte Anzahl an OSMP-Prozessen erzeugt. Diese Prozesse kommunizieren über gemeinsamen Speicher (shared memory) mit Message Passing. Der Runner übernimmt:  
- das Setup des Shared Memorys,  
- die Initialisierung von Semaphoren, Mailboxen, Synchronisationsstrukturen,  
- das Starten der Prozesse via fork() und execvp(),  
- das Logging aller relevanten Aktionen und Fehlermeldungen,  
- das koordinierte Warten auf Prozessende (waitpid()).

2. Kompilierung

gcc -o osmpRunner osmpRun.c ../osmpLibrary/osmpLib.c -lpthread -lrt

3. Ausführungsparameter

./osmpRunner -p <prozessanzahl> -l <logfile> -v <verbosity> -e <pfad\_zur\_exe> [args...]

4. Globale Variablen

- logfile\_path: Pfad zur Logdatei  
- verbosity\_level: Verbositätslevel (1: Standard, 2: erweitert, 3: Debug)  
- buffer[256]: Buffer zur temporären Formatierung von Logausgaben  
- osmp\_shared: Zeiger auf den gemeinsam genutzten Speicherbereich

5. Funktionsübersicht

5.1 setup\_shared\_memory(int process\_count)

Zweck:  
Initialisiert den Shared Memory, alle Mailboxen, Slot-Queues, Semaphore und die Barrierestruktur für process\_count Prozesse.  
  
Ablauf:  
- Speichergröße berechnen  
- Shared Memory anlegen (shm\_open + ftruncate)  
- Mapping mit mmap  
- Initialisierung:  
 - pid\_map: Zuweisung Rank → PID  
 - Mailboxen: Semaphore, Mutex, Slot-Index-Array  
 - FreeSlotQueue  
 - Logging-Mutex  
 - Barriere  
  
Rückgabe:  
- 0 bei Erfolg  
- EXIT\_FAILURE bzw. OSMP\_FAILURE bei Fehler

5.2 main(int argc, char \*argv[])

Zweck:  
Hauptfunktion zum Starten des OSMP-Systems. Parst Argumente, richtet das Umfeld ein und startet Kindprozesse mit execvp().  
  
Ablauf:  
1. Argumente parsen via getopt  
2. setup\_shared\_memory aufrufen  
3. Logfile vorbereiten  
4. Prozesse mit fork() starten  
5. execvp() im Kindprozess  
6. PID-Map im Elternprozess  
7. Logging & waitpid() auf alle Kinder  
8. Aufräumen  
  
Rückgabe:  
- EXIT\_SUCCESS bei Erfolg  
- EXIT\_FAILURE bei Fehler

6. Initialisierte Strukturen (aus osmpLib.h)

6.1 MailboxTypeManagement

- sem\_free\_mailbox\_slots: Steuerung freier Nachrichtenslots  
- sem\_msg\_available: Steuerung empfangsbereiter Nachrichten  
- mailbox\_mutex: Schutz für kritische Abschnitte  
- slot\_indices[OSMP\_MAX\_SLOTS]: Index-Mapping für Nachrichten  
- in, out: Zeiger für zirkuläre Mailboxen

6.2 FreeSlotQueue

- sem\_slots: Zähler für verfügbare Nachrichtenslots  
- free\_slots\_mutex: Schutz der Queue  
- free\_slots[OSMP\_MAX\_SLOTS]: Slot-ID-Puffer  
- head, tail: FIFO-Verwaltung

6.3 osmp\_shared\_info\_t

- process\_count  
- logfile\_path[256]  
- verbosity\_level  
- log\_mutex  
- pid\_map[]  
- mailboxes[]  
- fsq (FreeSlotQueue)  
- barrier (Synchronisation)

7. Logging & Fehlermanagement

Alle Vorgänge wie Prozessstarts, fork-/exec-Fehler und Exit-Codes werden mit OSMP\_Log() protokolliert.  
Das Logging erfolgt synchronisiert über sem\_wait(&log\_mutex).

8. Beispielaufruf

./osmpRunner -p 4 -l ./osmp.log -v 2 -e ./echoAll A B C  
  
→ Startet 4 Prozesse mit dem Programm ./echoAll,  
 schreibt Logeinträge in osmp.log und setzt das Logging-Level auf 2.

9. Weiterführende Literatur & Referenzen

- POSIX man pages: man 2 fork, man 3 sem\_init, man 2 mmap, man 3 pthread\_mutex\_init  
- Vorlesungsfolien Betriebssysteme – FH Münster (Prof. Dr.-Ing. Malysiak)  
- OSMP Praktikumsbeschreibung: BS-Praktikumsbeschreibung2025.pdf

10. Funktionsparameter & Rückgabewerte

### setup\_shared\_memory(int process\_count)

Initialisiert Shared Memory, alle Mailboxen und Synchronisationsmechanismen.

* Parameter:
* int process\_count – Anzahl der zu initialisierenden Prozesse

Rückgabewert: int – 0 bei Erfolg, EXIT\_FAILURE oder OSMP\_FAILURE bei Fehler

### main(int argc, char \*argv[])

Startpunkt des Programms. Parst Kommandozeilenargumente, initialisiert Umgebung und startet Prozesse.

* Parameter:
* int argc – Anzahl der Kommandozeilenargumente
* char \*argv[] – Kommandozeilenargumente (z. B. -p, -l, -v, -e)
* Rückgabewert: int – EXIT\_SUCCESS bei Erfolg, EXIT\_FAILURE bei Fehler

# OSMP Library – Technische Dokumentation

Projekt: Betriebssysteme Praktikum – OSMP

Autor: Konrad Skwarski, Christopher Jung, Erik Kolke

Dateien: osmpLib.c, osmpLib.h

Stand: 2025-04-02

## 1. Zweck der Bibliothek

Die OSMP Library implementiert eine message-passing-basierte Interprozesskommunikation über Shared Memory. Sie nutzt Semaphoren und Mutexes zur Synchronisation und unterstützt blockierende Kommunikation, Barrieren sowie kollektive Operationen.

## 2. Globale Variablen

- osmp\_shared: Zeiger auf das zentrale Shared Memory Segment  
- osmp\_rank: Rang des aktuellen Prozesses  
- mailboxes: Mailboxverwaltung für Prozesse  
- fsq: Verwaltung freier Nachrichtenslots  
- slots: Nachrichtenspeicher (Slots)

## 3. Initialisierung

Die Initialisierung erfolgt über OSMP\_Init() und richtet alle Strukturen ein, ermittelt den Prozess-Rang und stellt den Zugriff auf geteilten Speicher her.

## 4. Funktionen

### OSMP\_Init(const int \*argc, char \*\*\*argv)

Initialisiert die OSMP-Umgebung und ermittelt den Rank.

* Parameter:
* argc: Kommandozeilenparameteranzahl (nicht verwendet)
* argv: Kommandozeilenargumente (nicht verwendet)

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Finalize(void)

Beendet den Prozess und gibt Ressourcen frei.

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Send(const void \*buf, int count, OSMP\_Datatype datatype, int dest)

Sendet eine Nachricht an einen anderen Prozess.

* Parameter:
* buf: Zeiger auf Sendepuffer
* count: Anzahl der Elemente
* datatype: Datentyp der Elemente
* dest: Zielprozess (Rang)

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Recv(void \*buf, int count, OSMP\_Datatype datatype, int \*source, int \*len)

Empfängt eine Nachricht aus der eigenen Mailbox.

* Parameter:
* buf: Zeiger auf Empfangspuffer
* count: maximale Anzahl
* datatype: erwarteter Datentyp
* source: Adresse für Senderang
* len: Adresse für empfangene Bytes

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Size(int \*size)

Gibt die Anzahl gestarteter Prozesse zurück.

* Parameter:
* size: Rückgabewert

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Rank(int \*rank)

Gibt den Rang des aktuellen Prozesses zurück.

* Parameter:
* rank: Rückgabewert

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_SizeOf(OSMP\_Datatype datatype, unsigned int \*size)

Liefert die Größe eines OSMP-Datentyps.

* Parameter:
* datatype: Typ der Daten
* size: Rückgabewert für Byte-Größe

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Barrier(void)

Synchronisiert alle Prozesse an einer Barriere.

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Gather(void \*sendbuf, int sendcount, OSMP\_Datatype sendtype, void \*recvbuf, int recvcount, OSMP\_Datatype recvtype, int root)

Sammelt Daten von allen Prozessen beim Root.

* Parameter:
* sendbuf: lokaler Sendepuffer
* sendcount: Anzahl zu sendender Elemente
* sendtype: Datentyp
* recvbuf: Empfangspuffer (nur Root)
* recvcount: Anzahl je Prozess
* recvtype: Empfangs-Datentyp
* root: Rank des Root-Prozesses

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

### OSMP\_Log(OSMP\_Verbosity verbosity, char \*message)

Schreibt eine Nachricht in die Logdatei.

* Parameter:
* verbosity: Log-Level
* message: Nachrichtentext

Rückgabewert: OSMP\_SUCCESS bei Erfolg, OSMP\_FAILURE bei Fehler

## 5. Datenstrukturen

### MailboxTypeManagement

- Steuerung des Nachrichtenpuffers eines Prozesses mit Semaphoren und Mutex

### MessageType

- Struktur einer Nachricht mit Typ, Quelle, Länge und Payload

### FreeSlotQueue

- Globale Queue zur Verwaltung freier Nachrichtenslots

### osmp\_shared\_info\_t

- Struktur für globalen Shared-Memory-Status, inkl. Logging, PID-Map und Barriere

# barrier.c / barrier.h – Technische Dokumentation

Projekt: Betriebssysteme Praktikum – OSMP

Autor: Konrad Skwarski, Christopher Jung, Erik Kolke

Datum: 19.05.2025

Dateien: barrier.c, barrier.h

Ziel: Prozessübergreifende Barriere für Synchronisation mittels POSIX-Mutex und -Condition

## 1. Zweck der Barriere

Die Datei `barrier.c/h` implementiert eine prozessübergreifende Barriere zur Synchronisation. Eine definierte Anzahl an Prozessen/Threads wartet an der Barriere, bis alle anderen sie erreicht haben. Dann werden alle gleichzeitig freigegeben.

## 2. Datenstruktur: barrier\_t

- pthread\_mutex\_t mutex – schützt den Zugriff auf interne Variablen  
- pthread\_cond\_t convar – wartet auf Erreichen der Bedingung  
- int valid – Kennzeichnung, ob Barriere initialisiert wurde  
- int threshold – Anzahl erwarteter Threads  
- int counter – verbleibende Threads  
- int cycle – Barrierenummer zur Wiedererkennung neuer Zyklen

## 3. Funktionen & Funktionsweise

### barrier\_init(barrier\_t \*barrier, int count)

Initialisiert eine Barriere im Shared Memory mit einem erwarteten Teilnehmer-Zähler.

* Parameter:
* barrier – Zeiger auf die zu initialisierende Barriere
* count – Anzahl der erwarteten Teilnehmer (muss > 0 sein)

Rückgabewert: 0 bei Erfolg, sonst Fehlercode (z. B. EINVAL)

Funktionsweise:  
- Initialisiert prozessübergreifende Mutex- und Condition-Attribute  
- Setzt Zähler, Zustand und Zyklusvariablen  
- Verwendet: pthread\_mutexattr\_setpshared, pthread\_condattr\_setpshared, pthread\_mutex\_init, pthread\_cond\_init

### barrier\_destroy(barrier\_t \*barrier)

Zerstört die Synchronisationsmechanismen der Barriere.

* Parameter:
* barrier – Zeiger auf die Barriere

Rückgabewert: 0 bei Erfolg, sonst Fehlercode

Funktionsweise:  
- Prüft, ob Barriere gültig ist  
- Zerstört Mutex und Condition Variable  
- Setzt Barriere als ungültig (valid = 0)  
- Verwendet: pthread\_mutex\_destroy, pthread\_cond\_destroy

### barrier\_wait(barrier\_t \*barrier)

Synchronisiert alle Teilnehmerprozesse an der Barriere.

* Parameter:
* barrier – Zeiger auf die aktive Barriere

Rückgabewert: 0 bei Erfolg, sonst Fehlercode

Funktionsweise:  
- Sperrt Mutex und dekrementiert counter  
- Letzter Teilnehmer: setzt neuen Zyklus und broadcasted  
- Andere warten per pthread\_cond\_wait  
- Cancel-Status wird vorübergehend deaktiviert (pthread\_setcancelstate)  
- Verwendet: pthread\_cond\_broadcast, pthread\_cond\_wait, pthread\_mutex\_unlock