FH MÜNSTER
University of Applied Sciences

Fachbereich
Elektrotechnik und Informatik

Dokumentation und Bericht zur OSMP-Bibliothek

Praktikum Betriebssysteme,
SoSe 2024, PG 1, Gruppe 3

18. Juni 2024
Thomas Fidorin, Alina Rölver
fidorin.thomas@fh-muenster.de
alina.roelver@fh-muenster.de

Inhaltsverzeichnis

1 Einleitung

In diesem Dokument erläutern wir die Funktionsweise unserer Implementierung der OSMP-Bibliothek im Rahmen des Betriebssysteme-Praktikums im Sommersemester 2024. Dieser Bericht erläutert die folgenden drei Hauptkomponenten der Bibliothek:

- den Runner, implementiert in `osmp_runner/osmp_run.c`
- die eigentliche Implementierung der OSMP-Funktionalitäten in `osmp_library/osmplib.c` und `osmp_library/osmplib.h`
- den Logger, implementiert in `osmp_library/logger.c`

Der folgende Bericht fokussiert sich auf die Datenstrukturen und die Synchronisationskonzepte, die unserer Implementierung zugrunde liegen. Die Code-Dokumentation der einzelnen Funktionen findet sich in der Doxygen-Dokumentation im Anhang.

2 Runner

Im Runner werden zunächst die Kommandozeilenargumente geparst und die entsprechenden Einstellungen (Anzahl der Prozesse, Executable, ggfs. Argumente für das Executable, ggfs. Log-Datei und -Verbosität) gesetzt. Auf Basis der Anzahl der Prozesse kann die Größe des Shared Memory berechnet werden. Der Runner öffnet den Shared Memory und initialisiert ihn mit den entsprechenden Anfangswerten und Einstellungen (vgl. dazu Abschnitt 3.1).

Nach der erfolgreichen Initialisierung werden die einzelnen Executable-Prozesse mittels `fork()` gestartet (vgl. die Funktion `start_all_executables()`) und mit `exec()` wird das Executable in den neuen Prozess geladen. Dafür erhält der Runner einen Lock auf das Init-Mutex (vgl. Abschnitt 3.1), den die einzelnen Prozesse in ihrer eigenen Initialisierung (`OSMP_Init()`) ebenfalls locken müssen. Mit einer dazugehörigen Conditon-Variable können sie passiv darauf warten. So wird sichergestellt, dass der Runner erst alle Prozesse und die dazugehören Informationen im Shared Memory vollständig initialisiert hat, bevor ein Prozess darauf zugreifen kann. Falls der `fork()`-Call für einen Prozess fehlschlägt, killt der Runner alle Prozesse beendet das Programm.

Im Erfolgsfall wartet der Runner dann auf das Ende der einzelnen Prozesse, um danach Datenstrukturen im Shared Memory (insbesondere Mutexe und Semaphoren) wieder zu zerstören und den Shared Memory wieder freizugeben.

3 OSMPLib

Die OSMPLib bietet die Funktionalität für die Nutzer unserer Bibliothek, darunter haben wir die Funktionen: OSMP_Send, OSMP_Recv, OSMP_ISend, OSMP_IRecv, OSMP_Barrier und OSMP_Gather sowie die Struktur des Shared Memory.

3.1 Struktur des Shared Memory

Die OSMP-Bibliothek soll dazu dienen, Nachrichten zwischen Prozessen auszutauschen. Für die Implementierung ist zunächst die Unterscheidung zwischen Nachrichtenslot und Postfach wichtig: In einem Nachrichtenslot wird die eigentliche Nachricht gespeichert. Im Postfach eines Prozesses finden sich Verweise auf die Nachrichtenslots, in denen Nachrichten für den jeweiligen Prozess bereit liegen.

An den Anfang des Shared Memory (vgl. Abb. 3.1 auf S. 4)[1] wird die Größe des Memory geschrieben (**Size**, Z. 1). Durch diese Position am Anfang des Speicherbereichs können die Prozesse beim Initialisierungsvorgang, auch schon ohne die Gesamtgröße des Shared Memory zu kennen, diese Information auslesen und so die Gesamtgröße des Shared Memorys berechnen. Weiterhin liegen im Shared Memory ein **Logging-Mutex** für den Zugriff auf die Logdatei (Z. 1) sowie ein **Init-Mutex** und eine **Init-Condition-Variable** für den Initialisierungsvorgang (beides Z. 1; vgl. die Erläuterung dazu in 2).

Weiterhin enthält das Shared Memory eine Liste bzw. Array mit `OSMP_MAX_SLOTS` Elementen (**Free Slots**, Z. 2), in dem die aktuell freien Nachrichtenslots verzeichnet sind. Der **Free-Slots-Index** (Z. 2) indiziert die Stelle in **Free Slots**, an der das nächste freie Postfach verzeichnet ist. (Vgl. dazu auch Abschnitt 3.1.1.) Mit dem Semaphore **sem_shm_free_slots** (Z. 2) wird die Anzahl der freien Nachrichtenslots gezählt. Bei Belegen eines Slots wird die Anzahl mit `sem_wait()` dekrementiert, bei Freigeben eines Slots mit `sem_post()` inkrementiert. Der Mutex **mutex_shm_free_slots** (Z. 2) synchronisiert den Zugriff auf die **Free-Slots**-Liste und ihren Index.

An diese Elemente, die der Verwaltung des Shared Memory und der Nachrichtenslots dienen, schließen sich die `OSMP_MAX_SLOTS` **Message-Slots** an (Z. 3-6). Ein Message-Slot besteht jeweils aus den folgenden Elementen:

- Absender
- Länge

[1] Im weiteren Verlauf dieses Abschnitts referenzieren wir einzelne Stellen der Abb. 3.1 durch Zeilenangaben. Diese beziehen sich auf die farblich markierten Zeilen im Schema. Die Länge bzw. Größe der einzelnen Elemente steht nicht im Zusammenhang mit der tatsächlichen Größe im Speicherbereich. Es handelt sich dabei also nicht um ein *byte alignment*, sondern die Aufteilung in Zeilen dient nur der besseren Darstellbarkeit.

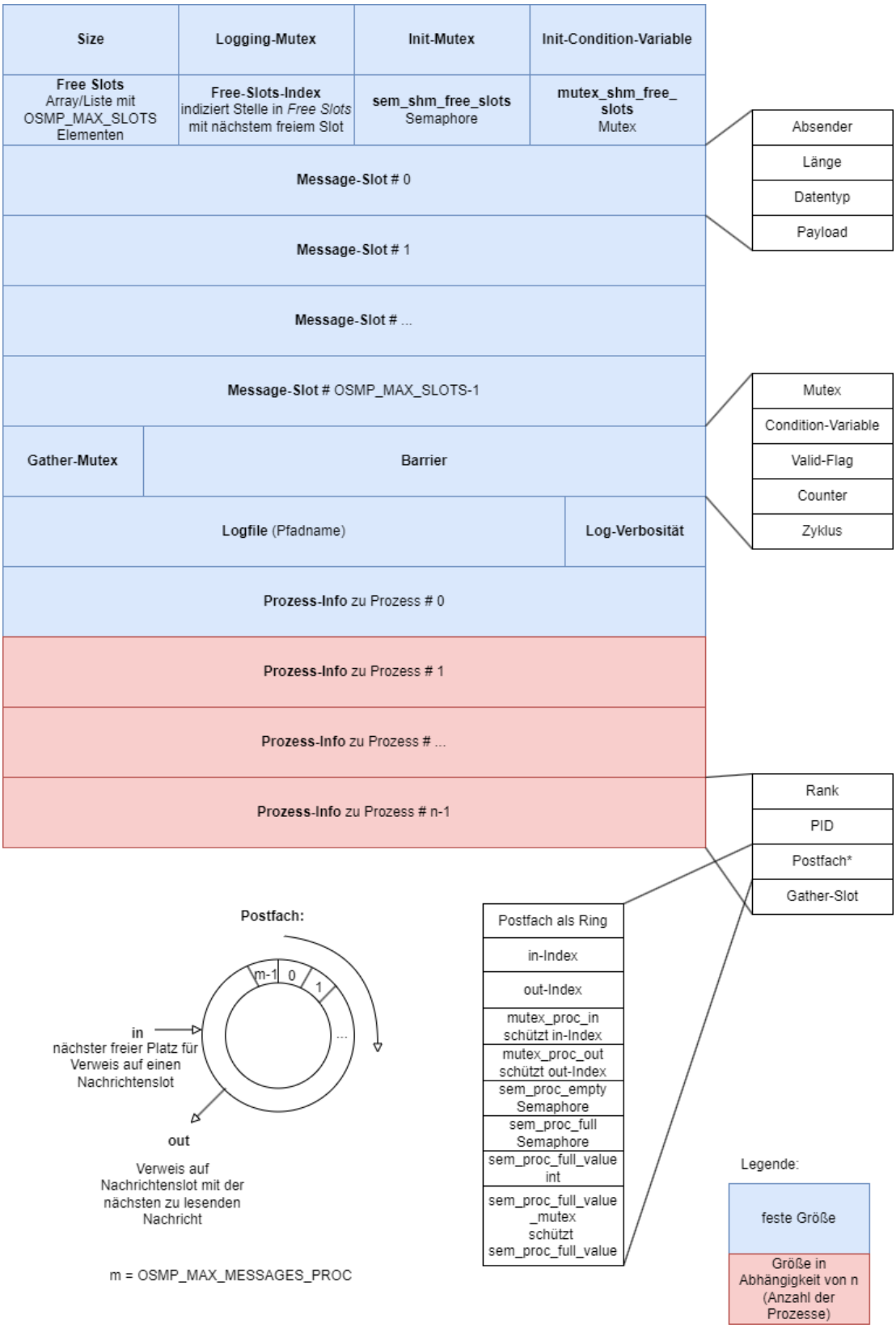

Abbildung 3.1: Schematische Darstellung des Shared Memory und der darin verwendeten Strukturen.

- Datentyp (`OSMP_Datatype`)
- Payload

Mit Hilfe des **Gather-Mutex** (Z. 7) wird der Zugriff auf die Gather-Postfächer synchronisiert (mehr dazu weiter unten). Die **Barrier**-Struktur (Z. 7) dient der Implementierung der Barrier-Funktionalität. Sie besteht aus folgenden Elementen:

- einem Mutex, mit dem der Zugriff auf die weiteren Barrier-Elemente synchronisiert wird
- einer Condition-Variable, mit der passives Warten an der Barriere ermöglicht wird
- einem Valid-Flag, mit dem signalisiert wird, dass die Barriere vollständig initialisiert ist
- einem Counter, mit dem die Anzahl der Prozesse gezählt wird, die die Barriere noch erreichen müssen
- einer Zyklus-Variable, mit der verschiedene aufeinanderfolgende Aufrufe der Barriere unterschieden werden können

Das Shared Memory enthält außerdem den Pfad zum **Logfile** sowie die **Log-Verbosität** (beides Z. 8), da auch auf diese Informationen von allen Prozessen zugegriffen werden muss.

Der Rest des Shared Memory besteht aus **Informationen zu den einzelnen Prozessen** (Z. 9-12). Zu jedem OSMP-Prozess werden die folgenden Informationen verwaltet:

- der Rang des Prozesses
- seine PID
- sein Postfach (vgl. Abschnitt 3.1.2), dies besteht wiederum aus
  - dem eigentlichen Postfach mit `OSMP_MAX_MESSAGES_PROC` Elementen, in denen jeweils ein Verweis auf einen freien Nachrichtenslot stehen kann
  - einem in-Index, der angibt, an welcher Stelle im Postfach ein freier Platz ist, um auf einen Nachrichtenslot zu verweisen
  - einem out-Index, der auf die Stelle im Postfach verweist, die wieder angibt, in welchem Nachrichtenslot sich die nächste zu lesende Nachricht für den Prozess befindet
  - je einem Mutex und einem Semaphore pro in- und out-Index, mit denen die freien Plätze im Postfach synchronisiert (Mutex) bzw. gezählt (Semaphore) werden

  - einen Zähler (`sem_proc_full_value`), der den aktuellen Wert des Semaphores mitzählt, d.h. wie viele Nachrichten(verweise) im Postfach liegen
  - einen Mutex, der den Zugriff auf diesen Zähler schützt

- sein Gather-Slot.

Die Anzahl der Prozess-Informationen ist natürlich abhängig von der Anzahl der Prozesse, die als Argument an das Programm übergeben wird. Daher ist auch die gesamte Größe des Shared Memory variabel. Es gibt jedoch einen festen Anteil: Alle Elemente bis einschließlich der ersten Prozess-Info (Z. 1-9) sind fest, da es immer mindestens einen Prozess geben muss. Die festen Elemente sind im Schema blau gekennzeichnet, die variablen rot.

3.1.1 Funktionalität der Free-Slots-Liste

Beim Start des Programms, bzw. genauer nach der Initialisierung des Shared Memory, sind zunächst alle Nachrichtenslots frei. Daher sind in der Free-Slots-Liste zunächst auch alle Slots verzeichnet (vgl. Abb. 3.2). Der Index bewegt sich nach rechts (d.h. er wird inkrementiert), wenn Slots belegt werden, und nach links (d.h. er wird dekrementiert), wenn Slots „zurückgegeben“ werden. Wenn ein Slot „entnommen“ wird, wird in der Liste die Konstante `NO_SLOT` geschrieben. Alle freien Postfächer sind „rechts“ des Indexes (und an der Stelle des Indexes selbst) verzeichnet (vgl. Abb. 3.3).

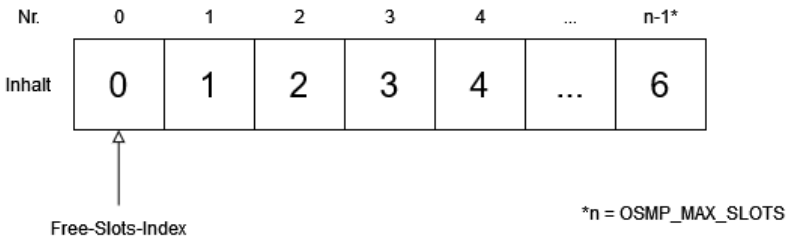

Abbildung 3.2: Die Free-Slots-Liste unmittelbar nach der Initialisierung des Shared Memory.

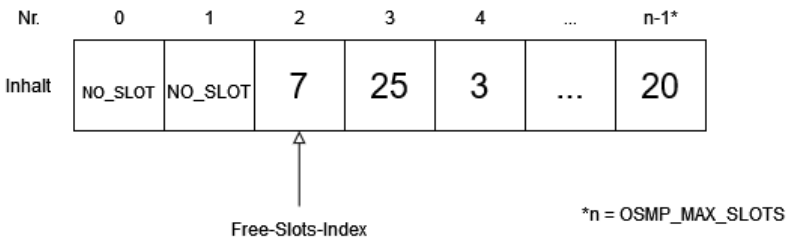

Abbildung 3.3: Ein fiktiver Zustand der Free-Slots-Liste im Laufe des Programms.

3.1.2 Funktionalität der Postfächer

Das Postfach (vgl. den unteren linken Teil von Abb. 3.1 auf S. 4) ist als Ring angelegt. Der in- und der out-Index zeigen initial auf die gleiche Stelle. Sobald per in-Index eine Stelle belegt wird, wird der in-Index inkrementiert. Der out-Index indiziert aber noch die Stelle, die gerade belegt wurde; diese enthält nun den Verweis auf einen Slot, aus dem gelesen werden kann. Nicht belegte Fächer im Postfach werden auf die Konstante `NO_MESSAGE` gesetzt.

3.2 OSMP-Funktionen

3.2.1 Blockierende Funktionen

OSMP_Send

Bei `OSMP_Send()` wird geprüft, ob der empfangene Prozess schon gestartet und fertig initialisiert wurde. Wenn nicht, wird gewartet. Falls der empfangende Prozess voll mit Nachrichten ist, wird mit Hilfe eines Semaphore gewartet. Außerdem wird ggfs. wird auf einen freien Slot mit Hilfe eines Semaphore gewartet, falls es keine freie Slots vorhanden sind. In den freien Slot wird die Nachricht geschrieben und die Adresse der Nachricht wird in das Postfach des empfangenen Prozesses geschrieben, so kann der Prozess später die Nachricht lesen. Es wird dazu mit Hilfe von einem Semaphore signalisiert, dass der Prozess eine Nachricht empfangen kann. Folgender Pseudocode fasst die Synchronisierung von `OSMP_Send()` zusammen. Für eine bessere Übersicht sind die unterschiedlichen Zugriffe auf Mutexe und Semaphoren farblich gekennzeichnet. So wird im Vergleich mit `OSMP_Recv` (3.2.1) ersichtlich, dass Schreiben und Lesen nahezu symmetrisch zueinander funktionieren.

```
1:  semwait(sem_proc_empty)                ▷ freier Platz im Postfach (Ring)
2:  semwait(sem_shm_free_slots             ▷ freier Nachrichtenslot
3:  mutex_lock(mutex_shm_free_slots
4:  entnimm Slot aus Liste
5:  mutex_unlock(mutex_shm_free_slots
6:  schreibe Nachricht in Slot
7:  mutex_lock(mutex_proc_in)
8:  lies in-Index
9:  schreibe Slot in Postfach
10: hole nächsten Index
11: mutex_unlock(mutex_proc_in)
12: semsignal(sem_proc_full)
```

OSMP_Recv

Bei OSMP_Recv wird zunächst mit Hilfe eines Semaphore gewartet, bis eine Nachricht da ist. Es wird ein Index von der nächsten Nachricht im Postfach-Ring (vgl. 3.1.2) benutzt, um herauszufinden, welcher der Slot der nächsten Nachricht ist. Dann wird diese Nachricht in den Buffer des empfangenen Prozesses kopiert und mit Hilfe von eines Semaphore wird signalisiert, dass es wieder einen freien Platz für Nachrichten gibt. Analog zu Abschnitt 3.2.1 fasst der folgende Pseudocode das Schreiben zusammen:

1: semwait(sem_proc_full)
2: mutex_lock(mutex_proc_out)
3: lies out-Index
4: lies Postfach (Index für Slot), lösche Index
5: inkrementiere out-Index
6: mutex_unlock(mutex_proc_out)
7: semsignal(sem_proc_empty)
8: kopiere Nachricht in User-Space
9: mutex_lock(mutex_shm_free_slots)
10: schreibe freien Slot in Liste
11: mutex_unlock(mutex_shm_free_slots)
12: semsignal(sem_shm_free_slots)

OSMP_Barrier

Bei `OSMP_Barrier()` wird ein Counter vom Anfangswert OSMP_Size runter gezählt, bis ein Prozess null erreicht. Mögliche Race-Conditions werden mit Hilfe eines Mutex verhindert. Wenn ein Prozess nicht der letzte ist, der die Barriere, wird durch eine Condition Variable gewartet, bis der Zyklus der Barriere erhöht wird. Um *spurious wakeups* zu verhindern, wird das Warten in einer while-Schleife mit einem Prädikat (Prüfung, ob Zyklus immer noch der gleiche ist) durchgeführt. Der letzte Prozess muss nicht mehr warten; er erhöht den Zyklus und signalisiert mit Hilfe eines Broadcasts, dass es für alle weitergehen kann. In Pseudocode ausgedrückt:

1: prüfe, ob Barrier initialisiert ist (wenn nicht: Fehler)
2: mutex_lock(barrier_mutex)
3: speichere aktuellen Zyklus
4: dekrementiere Counter ▷ *User-Threads ausschließen!*
5: **if** letzter Prozess **then**
6: re-initialisiere Barrier für nächsten Durchlauf und benachrichtige alle wartenden Prozesse per Condition-Variable
7: **else**

8: warte an Condition-Variable, bis Zyklusnummer nicht mehr der gespeicherten Zyklusnummer entspricht
9: mutex_unlock(barrier_mutex)

OSMP_Gather

Bei `OSMP_Gather()` schreibt jeder Prozess eine Nachricht in seinen Gather-Slot. Dann wird mit Hilfe von `OSMP_Barrier()` (s.o.) blockiert, bis alle Prozesse ihre Nachricht in ihren Slot geschrieben haben. So wird sichergestellt, dass der Prozess, der die Nachrichten sammelt, nicht frühzeitig anfängt, sie zu lesen. Wenn der Root (der empfangende Prozess) anfängt zu lesen, werden alle andere mit Hilfe vom Barrier 3.2.1 nochmal blockiert, so können wir gewährleisten, dass es erst weitergeht, wenn der Root das Lesen beendet hat, sodass keine Probleme mit Race Conditions auftreten. In Pseudocode ausgedrückt:

1: kopiere eigene zu sendende Nachricht in den eigenen Gather-Slot
2: warte, bis alle Prozesse in ihren eigenen Gather-Slot geschrieben haben (Barrier)
3: **if** empfangender Prozess (Root) **then**
4: mutex_lock(gather_mutex)
5: kopiere Nachrichten aus allen Gather-Slots in User-Space
6: mutex_unlock(gather_mutex)
7: warte, bis Root gelesen hat (Barrier)

3.2.2 Nicht blockierende Funktionen

OSMP_ISend

In `OSMP_ISend()` wird das Struct IParams(s.u.) benutzt, um die asychrone Funktionalität des Sendens zu ermöglichen. Am Anfang werden alle Parameter in das Struct geschrieben. Dann wird ein Thread zu einer Linked List hinzugefügt, um später alle gestarteten Threads zu verwalten. Dieser Thread startet asynchron das Senden der Nachricht mit Hilfe von OSMP_Send (s.o.). Wenn der Thread fertig ist, wird das Flag im IParams-Struct auf `OSMP_DONE` gesetzt, um dem Aufrufer signalisieren zu können, dass das Senden abgeschlossen ist. Er muss dies aber selbstständig mit `OSMP_Test()` (nicht blockierend) oder `OSMP_Wait()` (blockierend) prüfen.

OSMP_IRecv

In `OSMP_IRecv()` wird analog zu `OSMP_ISend()` das Struct IParams (s.u.) benutzt, um die asychrone Funktionalität des Empfangens zu ermöglichen. Am Anfang werden alle Parameter in das Struct geschrieben. Dann wird ein Thread zu einer Linked List hinzugefügt, um später alle gestarteten Threads zu verwalten. Dieser Thread startet asynchron das Empfangen der Nachricht mit Hilfe von `OSMP_Recv` (s.o.). Wenn der Thread fertig ist, wird das Flag im IParams-Struct auf `OSMP_DONE` gesetzt, um dem Aufrufer signalisieren

zu können, dass das Empfangen abgeschlossen ist. Er muss dies aber selbstständig mit `OSMP_Test()` (nicht blockierend) oder `OSMP_Wait()` (blockierend) prüfen.

IParams

IParams (vgl. Abb. 3.4) hat alle Attribute, um Send/Recv zu ermöglichen sowie die Attribute für die synchronisation zwischen den Threads. Dazu zählt ein Mutex für den Zugriff auf die Elemente des Structs, eine Condition-Variable, um auf das Fertigstellen des mit diesem Struct assoziierten Prozesses zu warten (in `OSMP_Wait()`), und done, dass signalisiert, ob der mit diesem Struct assoziierte blockierende Vorgang abgeschlossen ist.

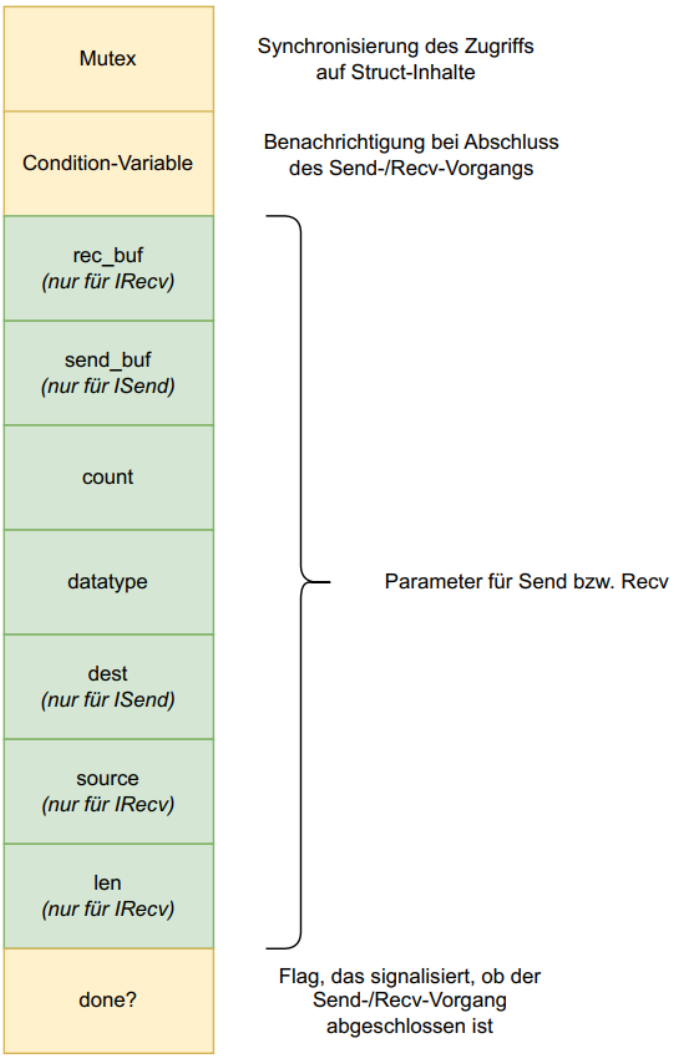

Abbildung 3.4: Die Struktur der IParams.

3.3 Logger

Der Logger bietet die Funktionalität abhängig von der Verbosität zu loggen. Die Verbosität beträgt drei Stufen (vgl. Praktikumsbeschreibung) Beim Loggen wird die Zeitpunkt zusammen mit der PID gelogt. Es kann eine Log-Datei beim Starten des Runner gewählt werden. Wenn es keine gewählt wird, wird es in log.log geloggt. Die Dateiname wird im Shared Memory gespeichert und lokal beim Initialsieren kopiert. Am Ende des Programms wird mit free sicher gestellt, dass alle Resourcen freigegeben werden. Das Loggen wird mit einem Logging-Mutex gemacht, um zu gewährleisten, dass keine Probleme vom Typ Race Conditions auftreten. Nach dem erfolgreichen Lock des Mutex wird die Datei mit `fopen()` und `append` geöffnet, dann wird mit `fprintf()` in die Datei geschrieben und am Ende mit `fclose()` geschlossen.