Es gibt zahlreiche Techniken zur lokalen Kommunikation und Synchronisation:

- *pipe*: unidirektional, Prozesse müssen miteinander verwandt sein.
- Benannte Pipes: über das Dateisystem, die Pipe-Datei muss explizit angelegt werden.
- UNIX-Domain-Sockets: bidirektional, deutlich einfacher im Vergleich zu benannten Pipes.
- Message Queues mit *msgsnd*, *msgrcv* etc. – sehr unhandlich wie alle System-V-IPC-Mechanismen
- Gemeinsame Speicherbereiche mit *mmap* – da fehlt noch die Synchronisierung...

- Speicherbereiche können auch von nicht miteinander verwandten Prozessen gemeinsam genutzt werden.
- Hierzu genügt es, mit *mmap* eine Datei in den eigenen Speicherbereich abzubilden.
- Vorteil: Das Hin- und Herkopieren kann minimiert werden.
- Nachteile:
  - Die Größe des gemeinsamen Speicherbereichs wird zu Beginn festgelegt. Dieser kann später nicht ohne weiteres wachsen.
  - Die Synchronisierung muss auf irgendeine andere Weise erreicht werden.

Zur Synchronisation von Prozessen auf dem gleichen Rechner bieten sich u.a. folgende Techniken an:

- Über das Dateisystem, etwa mit *link* (siehe erstes *mutexd*-Beispiel) oder mit *flock*. Nachteil: Wie warten wir darauf, dass uns der Partner etwas mitgeteilt hat?
- Semaphores aus dem System-V-IPC-Mechanismen (ebenso sehr unhandlich). Nachteil wie oben.
- Andere Kommunikation mit impliziter Synchronisierung
- Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle

Letzteres ist vielleicht überraschend. Interessanterweise können Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle auch von mehreren Prozessen mit Hilfe gemeinsamer Speicherbereiche genutzt werden.

shared_mutex.h

```
#include <stdbool.h>
#include <pthread.h>

typedef pthread_mutex_t shared_mutex;

bool shared_mutex_create(shared_mutex* mutex);
bool shared_mutex_free(shared_mutex* mutex);
bool shared_mutex_lock(shared_mutex* mutex);
bool shared_mutex_unlock(shared_mutex* mutex);
```

- Da besondere Vorkehrungen zu treffen sind, damit POSIX-Mutex-Variablen in gemeinsamen Speicherbereichen funktionieren, lohnt sich eine besondere Schnittstelle.
- Die Funktionen *shared_mutex_create* und *shared_mutex_free* dürfen nur von einem einzigen Prozess aufgerufen werden – typischerweise demjenigen, der den gemeinsamen Speicher vorbereitet, bevor die anderen Prozesse ihn in ihren Adressraum abbilden.

shared_mutex.c

```
bool shared_mutex_create(shared_mutex* mutex) {
   pthread_mutexattr_t mxattr;
   pthread_mutexattr_init(&mxattr);
   bool ok = true;
   if (pthread_mutexattr_setpshared(&mxattr, PTHREAD_PROCESS_SHARED)) {
      ok = false;
   }
   if (ok && pthread_mutex_init(mutex, &mxattr)) {
      ok = false;
   }
   pthread_mutexattr_destroy(&mxattr);
   return ok;
}
```

- Mit Hilfe von Mutex-Variablen können mehrere Parteien sichergehen, dass nur ein Prozess Zugang zu einer Ressource hat.
- Eine Mutex-Variable wird mit *pthread_mutex_init* initialisiert.
- Als einziges Attribut wird hier *PTHREAD_PROCESS_SHARED* gesetzt. Dies muss gesetzt sein, wenn die Mutex-Variable von mehreren Prozessen gemeinsam genutzt wird.

shared_mutex.c

```
bool shared_mutex_free(shared_mutex* mutex) {
   return pthread_mutex_destroy(mutex) == 0;
}

bool shared_mutex_lock(shared_mutex* mutex) {
   return pthread_mutex_lock(mutex) == 0;
}

bool shared_mutex_unlock(shared_mutex* mutex) {
   return pthread_mutex_unlock(mutex) == 0;
}
```

- Die weiteren Operationen können direkt übernommen werden.
- Wenn alles in Ordnung läuft, wird jeweils 0 zurückgegeben. Sonst handelt es sich um einen Fehlercode.
- Durch den Aufruf von *pthread_mutex_lock* wird der Aufrufer blockiert, bis die Mutex-Variable frei ist.
- Danach ist sie vom Aufrufer belegt, bis sie mit *pthread_mutex_unlock* wieder freigegeben wird.

Wenn ein Prozess auf eine Datenstruktur im gemeinsamen Speicherbereich zugreifen möchte, dann sollte das nur über die in der Datenstruktur integrierte Mutex-Variable erfolgen:

- Zu Beginn ist *shared_mutex_lock* aufzurufen,
- dann kann im sogenannten *kritischen Bereich* ein Zugriff auf die Datenstruktur erfolgen, wonach
- mit *shared_mutex_unlock* der Lock wieder freizugeben ist.

Es muss hier darauf geachtet werden, dass der kritische Bereich nicht versehentlich ohne eine Freigabe des Locks verlassen wird.

shared_cv.h

```
#include <pthread.h>
#include <afblib/shared_mutex.h>

typedef pthread_cond_t shared_cv;

bool shared_cv_create(shared_cv* cv);
bool shared_cv_free(shared_cv* cv);

bool shared_cv_wait(shared_cv* cv, shared_mutex* mutex);
bool shared_cv_notify_one(shared_cv* cv);
bool shared_cv_notify_all(shared_cv* cv);
```

- Bedingungsvariablen erlauben es, auf ein Ereignis zu warten, das mit einem der *notify*-Funktionen signalisiert wird.
- Wie bei Mutex-Variablen darf das Anlegen und Abbauen nur von einem einzigen Prozess vorgenommen werden.

shared-counter.c

```
struct shared_counter {
   shared_mutex mutex;
   unsigned int counter;
};
```

Das folgende triviale Beispiel zeigt, wie

- ein gemeinsamer Speicherbereich angelegt wird für diese Datenstruktur,
- wie dieser über *fork* an Kindprozesse weitervererbt wird,
- die dann konkurrierend darauf zugreifen und über die Mutex-Variable sich jeweils einen exklusiven Zugang sichern.

Beispiel zu Mutex-Variablen

shared-counter.c

```
/* create shared memory region */
void* sm = mmap(0, sizeof(struct shared_counter),
   PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0);
if (sm == MAP_FAILED) {
   perror("mmap"); exit(1);
}

/* initialize shared counter */
struct shared_counter* scnt = (struct shared_counter*) sm;
if (!shared_mutex_create(&scnt->mutex)) {
   perror("mutex"); exit(1);
}
scnt->counter = 0;
```

- Mit *MAP_ANON* wird implizit */dev/zero* als abzubildende Datei gewählt. Dies gehört nicht zum Umfang von POSIX, wird aber weitgehend unterstützt (einschließlich Linux und Solaris).

shared-counter.c

```
/* create some processes who inherit the shared memory region */
for (unsigned int i = 0; i < 10; ++i) {
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork"); break;
   }
   if (pid == 0) {
      srand(getpid() ^ time(0));
      for (unsigned int i = 0; i < 5; ++i) {
         shared_mutex_lock(&scnt->mutex);
         unsigned int increment = (unsigned int) (1 + rand() % 10);
         printf("[%5d] counter = %u, incremented by %u\n",
            (int)getpid(), scnt->counter, increment);
         scnt->counter += increment;
         shared_mutex_unlock(&scnt->mutex);
      }
      exit(0);
   }
}
```

shared-counter.c

```
        shared_mutex_lock(&scnt->mutex);
        unsigned int increment = (unsigned int) (1 + rand() % 10);
        printf("[%5d] counter = %u, incremented by %u\n",
           (int)getpid(), scnt->counter, increment);
        scnt->counter += increment;
        shared_mutex_unlock(&scnt->mutex);
```

- Der „kritische Bereich“ zwischen *shared_mutex_lock* und *shared_mutex_unlock* wird immer nur von maximal einem der Prozesse betreten.
- Weitere Prozesse werden ggf. in eine Warteschlange eingereiht, bis der Vorgänger *shared_mutex_unlock* aufruft.

shared-counter.c

```
/* wait for all childs to finish */
int wstat;
while (wait(&wstat) > 0);

printf("final value of the counter: %u\n", scnt->counter);

/* finish everything */
shared_mutex_free(&scnt->mutex);
munmap(sm, sizeof(struct shared_counter));
```

- Mit *wait*-Schleife synchronisieren wir uns mit dem Abschluss aller Kindprozesse.
- Danach wird die Mutex-Variable abgebaut und der gemeinsame Speicherbereich freigegeben.

Anlegen einer POSIX-Bedingungsvariablen

shared_cv.c

```
bool shared_cv_create(shared_cv* cv) {
   pthread_condattr_t condattr;
   pthread_condattr_init(&condattr);
   bool ok = true;
   if (pthread_condattr_setpshared(&condattr,
         PTHREAD_PROCESS_SHARED)) {
      ok = false;
   }
   if (ok && pthread_cond_init(cv, &condattr)) {
      ok = false;
   }
   pthread_condattr_destroy(&condattr);
   return ok;
}
```

- Wie bei den Mutex-Variablen müssen Bedingungsvariablen, die von mehreren Prozessen gemeinsam genutzt werden, mit dem Attribut *PTHREAD_PROCESS_SHARED* angelegt werden.

shared_cv.c

```
bool shared_cv_wait(shared_cv* cv, shared_mutex* mutex) {
   return pthread_cond_wait(cv, mutex) == 0;
}
```

- Die Operation *pthread_cond_wait* erfolgt immer in Verbindung mit einer Mutex-Variablen, die bereits mit *pthread_mutex_lock* reserviert sein muss.
- In einer atomaren Operation wird dann die Mutex-Variable freigegeben und der aufrufende Prozess (bzw. Thread) in die zugehörige Warteschlange eingereiht.

Weitere Operationen für POSIX-Bedingungsvariablen 324

shared_cv.c

```
bool shared_cv_free(shared_cv* cv) {
   return pthread_cond_destroy(cv) == 0;
}

bool shared_cv_notify_one(shared_cv* cv) {
   return pthread_cond_signal(cv) == 0;
}

bool shared_cv_notify_all(shared_cv* cv) {
   return pthread_cond_broadcast(cv) == 0;
}
```

- Bei *pthread_cond_signal* wird genau ein Prozess bzw. Thread aufgeweckt und aus der Warteschlange entfernt.
- Bei *pthread_cond_broadcast* wird die gesamte Warteschlange geleert und alle wartenden Prozesse bzw. Threads aufgeweckt.

Bei Ringpuffern handelt es sich um eine klassische Datenstruktur, die den konkurrierenden Lese- und Schreibzugriff erlaubt. Die Datenstruktur besteht aus einem festdimensionierten Array von Nachrichten und den folgenden Variablen:

- *read_index*: hier ist die nächste Nachricht zu lesen
- *write_index*: hier ist das nächste Nachricht zu schreiben
- *filled*: der aktuelle Füllgrad

Hierbei gilt, dass

- beim Schreiben darauf gewartet werden muss, bis der Füllgrad unter dem Maximum liegt und
- bei Lesen darauf zu warten ist, dass der Füllgrad positiv ist.

shared-ringbuffer.c

```
#define RINGBUF_SIZE (4)

struct shared_ringbuffer {
   shared_mutex mutex;
   shared_cv ready_for_reading;
   shared_cv ready_for_writing;
   struct message ringbuf[RINGBUF_SIZE];
   unsigned int filled;
   unsigned int write_index;
   unsigned int read_index;
};
```

- *mutex* sichert den exklusiven Zugang auf den Ringpuffer.
- Mit der Bedingungsvariablen *ready_for_reading* kann gewartet werden, bis der Füllgrad positiv ist und
- mit *ready_for_writing* kann darauf gewartet werden, dass der Füllgrad unter dem Maximum liegt.

shared-ringbuffer.c

```
static bool init_ringbuffer(struct shared_ringbuffer* rb) {
   if (!shared_mutex_create(&rb->mutex) ||
         !shared_cv_create(&rb->ready_for_reading) ||
         !shared_cv_create(&rb->ready_for_writing)) {
      return false;
   }
   rb->filled = rb->write_index = rb->read_index = 0;
   return true;
}
```

- Wie zuvor darf nur ein Prozess, die Mutex- und die Bedingungsvariablen anlegen.

Versenden einer Nachricht

shared-ringbuffer.c

```
static void send_message(struct shared_ringbuffer* rb,
      struct message* mp) {
   shared_mutex_lock(&rb->mutex);
   while (rb->filled == RINGBUF_SIZE) {
      shared_cv_wait(&rb->ready_for_writing, &rb->mutex);
   }
   rb->ringbuf[rb->write_index] = *mp;
   rb->write_index = (rb->write_index + 1) % RINGBUF_SIZE;
   ++rb->filled;
   shared_mutex_unlock(&rb->mutex);
   shared_cv_notify_one(&rb->ready_for_reading);
}
```

- Alle Operationen finden nur mit exklusivem Zugang statt. Einzige Ausnahme sind die *notify*-Funktionen bei Bedingungsvariablen, die ohne Vorkehrungen konkurrierend genutzt werden können.
- Solange der Ringpuffer voll ist, warten wir darauf, dass jemand eine Nachricht daraus empfängt.
- Sobald wir die Nachricht abgelegt haben, wecken wir mit *shared_cv_notify_one* maximal einen Prozess auf, der darauf wartete, etwas zu empfangen.

shared-ringbuffer.c

```
static void receive_message(struct shared_ringbuffer* rb,
      struct message* mp) {
   shared_mutex_lock(&rb->mutex);
   while (rb->filled == 0) {
      shared_cv_wait(&rb->ready_for_reading, &rb->mutex);
   }
   *mp = rb->ringbuf[rb->read_index];
   rb->read_index = (rb->read_index + 1) % RINGBUF_SIZE;
   --rb->filled;
   shared_mutex_unlock(&rb->mutex);
   shared_cv_notify_one(&rb->ready_for_writing);
}
```

- Solange der Ringpuffer leer ist, warten wir darauf, dass ein anderer Prozess eine Nachricht sendet.
- Sobald wir eine Nachricht entnommen haben, wecken wir mit *shared_cv_notify_one* maximal einen Prozess auf, der darauf wartete, etwas zu senden.

Die POSIX-Funktion *pthread_cond_wait* umfasst drei Operationen:

1. Freigabe der angegebenen Mutex-Variablen.
2. Warten bis auf das Eintreffen einer Notifikation über die Bedingungsvariablen mit *pthread_cond_signal* oder *pthread_cond_broadcast*.
3. Warten, bis die Mutex-Variable wieder gelockt ist.

Hierbei sind die beiden ersten Operationen atomar, d.h. der Mutex wird erst freigegeben, wenn der wartende Prozess in der Warteschlange eingetragen ist.

Denkbares Szenario, wenn *pthread_cond_wait* diese Atomizität nicht zusichern würde. Es sind die Prozesse P_1 und P_2 beteiligt und der Ringpuffer sei zu Beginn im initialen Zustand:

P_1	P_2
Aufruf von *receive_message* und Sichern des exklusiven Zugangs. **while** (*rb*−>*filled* == 0){	
	Aufruf von *send_message* und Warten auf den exklusiven Zugang.
shared_cv_wait(&*rb*−>*ready_for_reading*, &*rb*−>*mutex*); *rb*−>*mutex* wird freigegeben	
	wacht auf und hat exklusiven Zugang, fügt ein Element hinzu, erhöht *rb*−>*filled* um 1, gibt den Lock wieder frei *shared_cv_notify_one*(&*rb*−>*ready_for_reading*); (dies verpufft wirkungslos, da die Warteschlange noch leer ist)
Eintrag in die Warteschlange und langes Warten, obwohl der Ringpuffer nicht mehr leer ist.	

Gelegentlich wird **if** statt **while** verwendet. Warum jedoch **while** notwendig ist, zeigt folgendes Szenario mit den Prozessen P_1, P_2 und P_3:

P_1	P_2	P_3
Aufruf von *receive_message*, Gewinnung des exklusiven Zugangs. Feststellung, dass *filled* gleich 0 ist. Aufruf von *shared_cv_wait*, wobei der Lock freigegeben wird und P_1 in der Warteschlange ist.		
	Aufruf von *send_message*, Gewinnung des exklusiven Zugangs, Eintrag einer Nachricht, Freigabe des Locks.	
		Aufruf von *receive_message*, Gewinnung des exklusiven Zugangs, Feststellung, dass *filled* positiv ist. Entnahme der Nachricht und Freigabe des Locks.
	Aufruf von *shared_cv_notify_one*	
Aufwachen und Feststellung, dass *filled* immer noch 0 ist.		