U12 12. Übung

- Besprechung 9. Aufgabe (mt-httpd)
- PV-chunk Semaphore
- Semaphore vs. Mutex- und Condition-Variablen

U12-1 Semaphore vs. Mutexes und Conditions

- Semaphore sind der "abstraktere" Koordinierungsmechanismus
- Vorteile:
  - ➤ Koordinierungscode in der Anwendung ist schlanker
  - ➤ Semantik von P- und V-Operationen ist allgemein bekannt, Koordinierung damit unmittelbar verständlich
  - ➤ Anwendungsprogramm ist besser lesbar, weil eigentliche Funktionalität nicht zu sehr durch Koordinierungscode unterbrochen ist
- Nachteile:
  - ➤ weniger Flexibilität als beim expliziten Umgang mit Mutex-Locks und Condition-Variablen
    - – z. B. wenn man anwendungsabhängig entscheiden will, ob man aktiv wartet (Spin-Lock) oder den Thread schlafen legt (cond_wait)

U12-2 Lösungsskizze zum Ringpuffer

! Fehlerbehandlungen sind z. T. verkürzt oder vernachlässigt

1 Semaphor-Modul

- Semaphor-Datenstruktur

```
struct SEM {
    int s;                  /* Zustand */
    pthread_mutex_t m;
    pthread_cond_t c;
};
```

1 … Semaphor-Modul

- Initialisierungs-Funktion

```
SEM *sem_init(int n) {
    SEM *s;
    if ((s = malloc(sizeof SEM)) == NULL) {
        return NULL;
    }
    s->s = n;
    pthread_mutex_init(&s->m, NULL);
    pthread_cond_init(&s->c, NULL);
    return s;
}
```

- Lösch-Funktion

```
int sem_del(SEM *s) {
    pthread_mutex_destroy(&s->m);
    pthread_cond_destroy(&s->c);
    free(s);
    return 0;
}
```

1 … Semaphor-Modul

- P- Operation

```
void P(SEM *s) {
    pthread_mutex_lock(&s->m);
    while (s->s <= 0) {
        pthread_cond_wait(&s->c, &s->m);
    }
    s->s --;
    pthread_mutex_unlock(&s->m);
}
```

- V- Operation

```
void V(SEM *s) {
    pthread_mutex_lock(&s->m);
    s->s ++;
    pthread_cond_broadcast(&s->c);
    pthread_mutex_unlock(&s->m);
}
```

2 Ringpuffer: Datenstruktur

```
/* Grundidee: alle Puffer-relevanten Daten werden in einer
    Datenstruktur zusammengefasst, die dann an die Threads
    uebergeben wird.
    Dadurch könnte es auch mehrere Ringpuffer in einer
    Anwendung geben.
*/

struct BNDBUF {
    int *buf;
    unsigned int size;
    volatile unsigned int w_index;
    volatile unsigned int r_index;

    SEM *free_slots;
    SEM *filled_slots;
    SEM *w_mtx;
    SEM *r_mtx;
};
```

2 Ringpuffer-Initialisierung

```
BNDBUF *bb_init(unsigned int size) {
    BNDBUF *bbuf = calloc(1, sizeof(struct BNDBUF));
    bbuf->buf = malloc(sizeof(int) * size);
    bbuf->size = size;

    /* Zählende Semaphore zum Warten auf freie (Schreiber)
       bzw. gefüllte (Leser) Plätze */
    bbuf->free_slots = sem_init(size);
    bbuf->filled_slots = sem_init(0);

    /* Mutexes zu Leser/Leser bzw.
       Schreiber/Schreiber-Synchronisation */
    bbuf->r_mtx = sem_init(1);
    bbuf->w_mtx = sem_init(1);

    return bbuf;
}
```

2 Ringpuffer: Einfügen eines Elements

```
void *bb_add(BNDBUF *bbuf, int fd) {
    /* Warten auf freien Slot */
    P(bbuf->free_slots);

    /* Synchronisation mit anderen Schreibern (lock) */
    P(bbuf->w_mtx);

    /* Eintragen des Elements */
    bbuf->buf[bbuf->w_index] = fd;
    w_index = (w_index+1) % bbuf->size;

    /* Synchronisation mit anderen Schreibern (unlock) */
    V(bbuf->w_mtx);

    /* Bereitstellen eines gefüllten Slots */
    V(bbuf->filled_slots);
}
```

2 Ringpuffer: Entnahme eines Elements

```
int *bb_get(BNDBUF *bbuf) {
    int fd;

    /* Warten auf gefüllten Slot */
    P(bbuf->filled_slots);

    /* Synchronisation mit anderen Lesern (lock) */
    P(bbuf->r_mtx);

    /* Eintragen des Elements */
    fd = bbuf->buf[bbuf->r_index];
    r_index = (r_index+1) % bbuf->size;

    /* Synchronisation mit anderen Lesern (unlock) */
    V(bbuf->r_mtx);

    /* Bereitstellen eines freien Slots */
    V(bbuf->free_slots);

    return fd;
}
```

3 Producer (Haupt-Thread)

```
int main(int argc, char **argv) {
    int s;
    BNDBUF *bbuf;

    /* ...Initialisierung... */
    bbuf = bb_init(128);

    while(1) {
        /* neue Verbindung annehmen */
        s = accept(...);

        /* den Arbeitern den socket zur Verfügung stellen */
        bb_add(bbuf, s);
    }

}
```

4 Consumer (Arbeiter-Threads)

```
static void *consumer(void *arg) {
    BNDBUF *bbuf = (BNDBUF *) arg;

    while(1) {
        /* neuen Auftrag entnehmen */
        int s = bb_get(bbuf);

        /* ... Auftrag bearbeiten ... */

        /* Verbindung schliessen */
        close(s);
    }
}
```