PThreads-Merkblatt

1. Übersicht der Funktionen

Prozess-Management		
	int getpid()	Liefert die Prozessnummer (PID) des aktuellen Prozesses.
	int getppid()	Liefert die Prozessnummer (PID) des Elternprozesses ('p' für “parent”).
	int fork()	Erstellt eine Kopie des aktuellen Prozesses. Kopie und Original setzen ihre Ausführung genau an der Aufrufstelle fort und unterscheiden sich nur im Rückgabewert von fork(): 0 in der Kopie (Kindprozess) und >0 im Original (Elternprozess).
	int wait(int *status)	Wartet auf die Beendigung eines beliebigen Kindprozesses und speichert dessen Beendigungs-Status in den durch den Zeiger *status* bezeichneten int (falls nicht NULL). Kann u.U. auch Fehler zurückgeben (z.B. wenn keine Kindprozesse existieren).
	int waitpid(int pid, int *status, int flags	Wartet auf die Beendigung des spezifischen Kindprozesses *pid*. Sonst im wesentlichen analog zu wait().
	int kill(int pid, int sig)	Schickt das Signal (nicht zu verwechseln mit dem “Signal”-Typ aus dem Prozess-Interaktions-Kapitel, hier geht es um UNIX-Signale, vgl. die beiden “signal”-Manpages, 'man 2 signal' und 'man 7 signal') mit der Nummer *sig* an den Prozess *pid*. Häufigster Use Case ist die Zerstörung eines Prozesses mit dem Signal SIGTERM oder SIGKILL.
Thread-Management		
	Die C-Funktionen der PThreads-Bibliothek benutzen folgende Datentypen: • pthread_t Identifikator für einen Thread • void *func(void *) Funktionssignatur für Thread-main()-Funktionen • pthread_mutex_t ein Mutex • pthread_cond_t eine Bedingungsvariable (“Condvar”) Im Folgenden sind diese aus Platzgründen mit “PT”, “FN”, “mutex” und “cond” abgekürzt.	
	PT pthread_self()	Liefert den Thread-Identifikator des aktuellen Threads.
	int pthread_create(PT *tid, NULL, FN mainfunc, void *arg)	Erstellt einen neuen Thread im aktuellen Prozess. Das zweite Argument erlaubt das Setzen von abweichenden Thread-Attributen (nicht VL-relevant). Der Thread führt die angegebene Funktion *mainfunc* als “main()-Ersatz” aus. Das Argument *arg* wird dieser Funktion übergeben. Eine Rückkehr von dieser Funktion gleicht einem Aufruf von pthread_exit() (s.u.), der Thread terminiert dann also.
	int pthread_join(PT tid, void **ret)	Wartet auf die Beendigung des mit *tid* bezeichneten Threads. Der Rückgabewert der Thread-”main()” wird in die

		mit *ret* bezeichnete Variable gespeichert (wenn nicht NULL); diese Funktion arbeitet also analog zu waitpid(). Achtung: im Gegensatz zu fork/wait bei Prozessen existiert bei Threads kein Eltern-/Kind-Konzept, es gibt also keinen ausgezeichneten “Haupt-Thread”, jeder Thread kann auf die Terminierung jedes anderen warten.
	void pthread_exit(void *ret)	Beendet den aktuellen Thread mit Rückgabewert *ret* (ist also äquivalent dazu, dass der Thread von seiner *mainfunc* mit “return *ret*” zurückkehrt).
Mutexes		
	pthread_mutex_init(mutex *m, NULL)	Initialisiert den Mutex. Das zweite Argument erlaubt wieder die Angabe von abweichenden Mutex-Attributen; nicht VL-relevant.
	pthread_mutex_lock(mutex *m)	Ergreift (sperrt) den Mutex. Andere Threads, die diesen Mutex ergreifen wollen, solange er von einem Thread gehalten wird, werden in den Zustand BLOCKIERT versetzt, bis der Mutex wieder frei ist.
	pthread_mutex_unlock (mutex *m)	Gibt den Mutex wieder frei.
Bedingungs-Synchronisation (mit “Condvars”)		
	pthread_cond_init(cond *c, NULL)	Initialisiert die Bedingungsvariable. Das zweite Argument erlaubt wieder die Angabe von abweichenden Attributen; nicht VL-relevant.
	pthread_cond_signal(cond *c)	Signalisiert einem der Threads, die in pthread_cond_wait() blockieren, dass die Bedingung dieser Variable eingetreten ist und die Ausführung fortgesetzt werden kann.
	pthread_cond_wait(cond *c, mutex *m)	Wartet darauf, dass ein anderer Thread das Eintreten der gewünschten Bedingung signalisiert. Das Warten wird durch Versetzen in den Zustand BLOCKIERT erreicht. Diese Funktion muss mit gesperrtem Mutex *m* aufgerufen werden; sie gibt für die Zeit des Blockierens den Mutex atomar frei (so dass, sobald der Thread blockiert worden ist, andere Threads in von diesem Mutex geschützte kritische Bereiche eintreten können). **Achtung**: Der POSIX-Standard erlaubt es, dass diese Funktion auch “ohne Benachrichtigung” wieder aufwacht. Deshalb muss immer die zugrundeliegende logische Bedingung geprüft werden, bevor der Thread die Ausführung tatsächlich fortsetzt. Siehe Beispiel unten.
	pthread_cond_broadcast(cond *c)	Weckt alle Threads, die auf das Eintreten einer Bedingung warten, auf.

2. Anwendungsbeispiel

Grundsätzliche Überlegung bei der Verwendung von Mutexes und Condvars muss sein, dass alle möglichen verzahnten Ausführungen mehrerer Threads (und das können sehr

sehr viele Möglichkeiten sein!) korrekt ablaufen. Im folgenden ist noch einmal das Beispiel für den Bounded-Buffer-Producer/Consumer dargestellt und umfangreich kommentiert.

```
 1 #define N    16
 2
 3 struct bounded_buffer {
 4     void             *objects[N];
 5     int               head;
 6     int               tail;
 7     int               count;
 8     pthread_mutex_t   mutex;
 9     pthread_cond_t    cond_notempty;
10     pthread_cond_t    cond_notfull;
11 };
12
13 void buffer_init(struct bounded_buffer *bb) {
14     pthread_mutex_init(&bb->mutex, NULL);
15     pthread_cond_init(&bb->cond_notempty, NULL);
16     pthread_cond_init(&bb->cond_notfull, NULL);
17     bb->head = 0;
18     bb->tail = 0;
19     bb->count = 0;
20 }
21
22 void deposit(struct bounded_buffer *bb, void *obj) {
23     pthread_mutex_lock(&bb->mutex);
24     while (bb->count == N) {
25         pthread_cond_wait(&bb->cond_notfull, &bb->mutex);
26     }
27     bb->objects[bb->head] = obj;
28     bb->head = (bb->head + 1) % N;
29     bb->count++;
30     pthread_cond_signal(&bb->cond_notempty);
31     pthread_mutex_unlock(&bb->mutex);
32 }
33
34 void *fetch(struct bounded_buffer *bb) {
35     void *ret;
36
37     pthread_mutex_lock(&bb->mutex);
38     while (bb->count == 0) {
39         pthread_cond_wait(&bb->cond_notempty, &bb->mutex);
40     }
41     ret = bb->objects[bb->tail];
42     bb->tail = (bb->tail + 1) % N;
43     bb->count--;
44     pthread_cond_signal(&bb->cond_notfull);
45     pthread_mutex_unlock(&bb->mutex);
46
47     return ret;
48 }
```

	Z. 1-11 Die Struktur ist ein Ringpuffer: sie enthält *N* Speicherzellen, Objekte werden immer am *head* gespeichert (der danach eins weitergezählt wird) und vom *tail* ausgegeben (der ebenfalls hinterher weitergezäht wird). *head* zeigt also immer auf die Zelle, die als nächstes belegt werden wird, und *tail* auf die, die als nächste ausgegeben wird.
	Z 13-19 Bei der Initialisierung werden nur Mutex und Condvars mit den entsprechenden PThreads-Bibliotheksfunktionen standardmäßig initialisiert.
	Z. 22 / Z. 34 deposit() erlaubt das Hinzufügen eines Objekts *obj* zum Puffer. fetch() erlaubt das Entnehmen eines Objekts aus dem Puffer.
	Z. 23 / Z. 37 Da die Funktionen Modifkationen am Array objects[] und den Zählern head bzw. tail und count vornehmen, müssen sie ausschließen, dass andere Threads nebenläufig eine weitere deposit()- oder eine fetch()-Operation ausführen. Beide Operationen bilden also kritische Bereiche auf derselben Datenstruktur. Dazu dient der Mutex.
	Z. 24-26 / Z. 38-40 Für den Fall, dass der Puffer derzeit maximal gefüllt bzw. leer ist, muss der aktuelle Thread warten: in deposit() muss gewartet werden, bis durch eine fetch()-Operation wieder “Luft” für ein neues Objekt entstanden ist, und in fetch() muss gewartet werden, bis durch eine deposit()-Operation wieder ein Objekt im Puffer vorhanden ist. Damit dies aber überhaupt stattfinden kann, muss der Schutz-Mutex für die Dauer des Wartens vorübergehend freigegeben werden. Dies wird durch den Aufruf von pthread_cond_wait() erreicht. Da diese Funktion jedoch zurückkehren kann, auch wenn die zugrundeliegende logische Bedingung (“Puffer nicht mehr voll”) nicht (oder nicht mehr) erfüllt ist, muss der Aufruf jeweils in eine Schleife eingebunden werden, die die eigentliche logische Bedingung überprüft.
	Z. 27-29 / Z. 41-43 Nachdem der Puffer nun nicht mehr voll bzw. leer ist (der Thread hat die vorige while-Schleife verlassen und hält den Mutex, so dass auch keine nebenläufigen Änderungen stattfinden können), kann nun die eigentliche deposit()- bzw. fetch()-Operation stattfinden.
	Z. 30 / Z.44 Da es sein kann, dass Threads in der jeweils anderen Operation auf eine Änderung des Puffer-Zustands (nicht mehr leer bzw. nicht mehr voll) warten, wird nun eine pthread_cond_signal()-Operation auf der entsprechenden Condvar ausgelöst, die einen dieser wartenden Threads aufweckt (wenn denn welche existieren).
PThreads FAQ	
	Q: Warum findet pthread_cond_signal() noch unter dem Mutex statt? A: Das muss nicht zwingend so sein. Die Manpage stellt dies ausdrücklich frei. Fest steht natürlich, dass alle Änderungen an der Datenstruktur vom Mutex geschützt ablaufen müssen — man könnte also allenfalls das pthread_cond_signal() hinter das pthread_mutex_unlock() ziehen. Ob die eine oder die andere Reihenfolge zu besserer Performance (oder “besser vorhersagbarem Scheduling”, wie die Manpage angibt) führt, liegt an der ganz konkreten Implementierung der PThreads-Bibliothek.