Lesson 6

Threads and Synchronization:

Using notes from:
http://www.csl.mtu.edu/cs4411.ck/www/NOTES/threads/ and the Technion.

When to use threads?
1. When the program does tasks that do not depend on one another.
2. Servers who need to serve many requests/connections.
3. Parallel computations/algorithms.

As you've learnt in class, Solaris uses a hybrid approach: combining user level threads with kernel level threads.

Using Threads:
- Compilation: gcc ***-lthread*** prog.c
- #include <thread.h>
- Many of the routines include many arguments. In many cases you can/should use the default values for the arguments, which are generally NULL or 0.

Creation of a thread: ***thr_create:***
int thr_create(void *stack_base, size_t stack_size, void *(*start_func)(void *), void *arg, long flags,
                    thread_t * ID);

Calling **thr_create()** will create a *child thread* which will execute concurrently with the *parent thread* (*i.e.*, the caller) and other threads created by the *parent thread*. A newly created thread shares all of the calling process' global data with the other threads in this process; however, it has its own set of attributes and private execution stack. The new thread inherits the calling thread's signal mask, possibly, and scheduling priority.
The following is the meaning of each argument:
- Function **start_func()** is called and run as a thread. Function **start_func()** has only one argument, a pointer to **void** and returns a pointer to **void**.
- **arg** is a pointer to **void** and is supplied to function **start_func()** as its argument. Thus, if you want to send a pointer to a object whose type is not **void**, you need to cast that pointer to a pointer to **void**. Then, in function **start_func()**, you should cast the pointer of the argument back to a pointer to the original type. If more than one argument needs to be passed to **start_func**, the arguments can be packed into a structure, and the address of that structure can be passed to **arg**.
- **ID** is a pointer to a variable of type **thread_t**. ***ID** is where the new thread's system ID is stored. The **ID** is only valid within the calling process.
- **thr_create()** returns an integer. Any non-zero value indicates that an error has occurred and in this case no thread is created.

With thr_create(), the new thread will use the stack starting at the address specified by stack_base and continuing for stack_size bytes. If stack_base is NULL then thr_create() allocates a stack for the new thread with at least stack_size bytes. If stack_size is zero, then a default size is used.

The lifetime of a thread begins with the successful return from thr_create(), which calls start_func() and ends with either:
o the normal completion of start_func(),
o the normal completion of the main thread (the whole process terminats),
o the return from an explicit call to thr_exit(), or
o the conclusion of the calling process (exit()).

RETURN VALUES
0 indicates a successful return and a non-zero value indicates an error.

USAGE

```
int thr_create(
          NULL,            /* use this default            */
          0,               /* use this default            */
          void *(*start_func)(void *),  /* thread funct   */
          void *arg,  /* argument passed to start_func() */
          0,          /* use default                  */
          thread_t *ID);   /* use NULL works fine    */
```

<u>Termination of a thread</u>: *thr_exit:*

SYNOPSIS

#include <thread.h>

size_t status;
void thr_exit(void *status);

After the termination of a thread, the memory and its execution become unavailable.

Do not use exit() to terminate a thread or the main program. Otherwise, the whole program, including all threads the main program has, terminates.

<u>Waiting for a thread completion</u>: *thr_join:*

int thr_join(thread_t target_thread, thread_t *departed, void **status);

In many cases, a thread has to wait until some other threads terminate.

The thr_join() function suspend processing of the calling thread until the target **target_thread** (the 6th argument in **thr_create**) completes. **target_thread** must be a member of the current process.
Several threads cannot wait for the same thread to complete; one thread will complete successfully and the others will terminate with an error of ESRCH.
If **thr_join()** is reached *before* the completion of the indicated thread, the caller waits until the completion of the indicated thread. After this, the caller executes its next statement.
If **thr_join()** is reached *after* the completion of the indicated thread, nothing will happen to the caller and the caller executes its next statement.

If a thr_join() call returns successfully with a non-null status argument, the value passed to thr_exit() by the terminating thread will be placed in the location referenced by status.
If the target **target_thread** ID is 0, thr_join() waits for any undetached thread in the process to terminate.
If departed is not NULL, it points to a location that is set to the ID of the terminated thread if thr_join() returns successfully.

RETURN VALUES

If successful, thr_join() would return 0; otherwise, an error number is returned to indicate the error.

NOTES

Using thr_join() in the following syntax,

```
while (thr_join(NULL, NULL, NULL) == 0);
```

will wait for the termination of all other undetached and non-daemon threads; after which, EDEADLK will be returned.

USAGE

```
   int thr_join(
          thread_t wait_for,   /* thread we are waiting  */
          0,                   /* 0 is fine with use     */
          void **status);      /* status in thr_create() */
```

<u>Yielding the control of execution</u>: *thr_yield:*

void thr_yield(void);

thr_yield() causes the current thread to yield its execution in favor of another thread with the same or greater priority.It is equivalent to say that the calling thread is put back to the ready queue and a thread from the ready queue is picked as the candidate for execution. Note that it is possible that the calling thread is picked if the ready queue is empty when **thr_yield()** is called.
Since the calling thread is temporarily suspended, one may consider it entering a "sleeping" state for an unspecified period. In this way, it "simulates" **sleep()**.

<u>Get calling thread's ID</u>*: thr_self:*
SYNOPSIS
```
#include <thread.h>

thread_t thr_self(void);
```

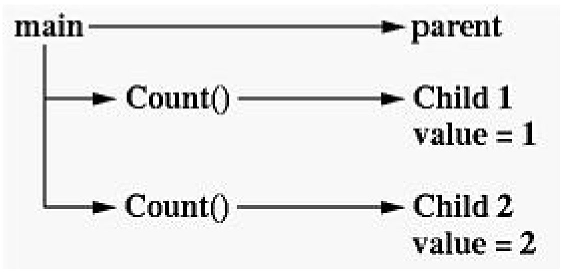
<u>Global Variables:</u>
All threads and the main program run in the same address space allocated to the main program. This implies that names declared as external (global) can be accessed by all threads. However, names declared local to a function are still local to that function. If a variable is shared by threads, it is very difficult to predict its value. In other words, the behavior of a multithreaded program is *dynamic*.

Examples:

(thread6_1.c)

```
#include <stdio.h>
#include <thread.h>

void *count(void *JunkPTR)
{
      int *valuePTR = (int *) JunkPTR;  /* convert to integer ptr.  */
      int value     = *valuePTR;              /* then take the value      */

      printf("Child: value = %d\n", value);
}

int main(void)
{
      thread_t  ID1, ID2;          /* for thread IDs                */
      int       v1 = 1;                /* argument for the 1st thr */
      int       v2 = 2;               /* argument for the 2nd thr */

      thr_create(NULL, 0, count, (void *) (&v1), 0, &ID1);
      thr_create(NULL, 0, count, (void *) (&v2), 0, &ID2);
      printf("Parent\n");
      sleep(2);                          /* why is sleep() here?     */
}
```

The above program creates two threads, each of which is a copy of function **count()**.
The sixth argument can be a **NULL** if we do not care about the ID of the created thread. If we do want the ID, the sixth argument must be a pointer to a variable of type **thread_t**. In the above example, **ID1** stores the thread ID of the first thread.
count() receives a pointer to **void**. To retrieve the value passed by this pointer, the argument, **JunkPTR** must be first converted back to the right type. In this case, it is a pointer to **int**.
If the thread creations are successful, we have three threads running: (1) the main program, (2) the first copy of **count()** and the second copy of **count()**.
After creating two copies of **count()**, the main program displays a message and sleeps for two seconds. Note that **sleep()** is a Unix system call. But, why is a call to **sleep()** required here? Can we remove it?
Well, we cannot remove this **sleep()**. Since the two copies of **count()** are child threads of the main program which is the parent. If parent exits, all of its child threads exit as well. Thus, if **sleep()** is not there and if the main runs faster than any one copy of **count()**, it is possible that before **count()** can display anything, the main program exits. As a result, you won't see anything displayed from **count()**. Therefore, the call to **sleep()** here is to delay the main program a little until the messages from copies of **count()** can be displayed.
Using **sleep()** is definitely not a good practice. A better way to do this is with **thr_join()**.
The following diagram shows the three concurrently executing threads, one parent (*i.e.*, the main program) and two children (*i.e.*, the two copies of **count()**).



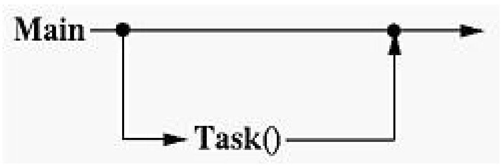
(thread6_2.c)

```
#include <thread.h>
#include <sys/wait.h>

void *Task(void *Junk)
{
          …
       thr_exit((void *) 5);
}
```

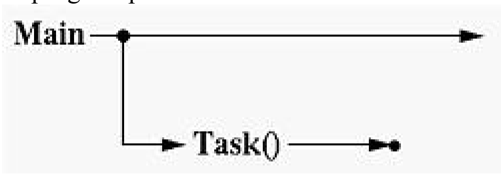
```
int main(void)
{
        thread_t TaskID;
        size_t     TaskStatus;

        thr_create(NULL,0,Task,(void *) NULL,0,&TaskID);
             …
        thr_join(TaskID, 0, (void *) &TaskStatus);
        printf("Thread %d exited with status %d\n", TaskID, TaskStatus);
}
```

```
1 > gcc -lthread -o thread6-2 thread6_2.c
2 > thread6-2
Thread 4 exited with status 5
```

If **thr_join()** in the main program is reached before **Task()** terminates, the main program waits there until **Task()** completes. Then, the execution of main resumes:



On the other hand, if **Task()** has already terminated when the main program reaches **thr_join()**, there is nothing to "join" and the main program proceeds:



(thread6_3.c)

```
#include <thread.h>
#include <stdio.h>

#define MAX_ITERATION 200

void *ThreadA(void *DontNeedIt)
{
    int  i;

    for (i = 1; i <= MAX_ITERATION; i++)
        printf("Thread A speaking: iteration %d\n", i);
    thr_exit(NULL);
}

void *ThreadB(void *DontNeedIt)
{
    int  i;

    for (i = 1; i <= MAX_ITERATION; i++)
        printf("     Thread B speaking: iteration %d\n", i);
    thr_exit(NULL);
}

int main(void)
{
    thread_t  FirstThread, SecondThread;
    size_t    StatusFromA, StatusFromB;

    thr_create(NULL, 0, ThreadA, (void *) NULL, 0, &FirstThread);
    thr_create(NULL, 0, ThreadB, (void *) NULL, 0, &SecondThread);

    thr_join(FirstThread, 0, (void *) &StatusFromA);
    thr_join(SecondThread, 0, (void *) &StatusFromB);
}
```

(thread6_4.c)

```
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int main(int argc, char *argv[])
{
        for ( i = 0; i < NUM_THREADS; i++)
                thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0, &tid[i]);
        while (thr_join(NULL, NULL, NULL) == 0)
                ;

        printf("main() reporting that all %d threads have terminated\n", i);
}  /* main */

void *sleeping(void *arg)
{
        int sleep_time = (int)arg;
        printf("thread %d sleeping %d seconds ...\n", thr_self(), sleep_time);
        sleep(sleep_time);
        printf("\nthread %d awakening\n", thr_self());
        return (NULL);
}
```

If main() had not waited for the completion of the other threads (using thr_join()), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely.

<u>Output:</u>

```
thread 4 sleeping 10 seconds ...
thread 5 sleeping 10 seconds ...
thread 6 sleeping 10 seconds ...
thread 7 sleeping 10 seconds ...
thread 8 sleeping 10 seconds ...

thread 5 awakening

thread 4 awakening

thread 6 awakening

thread 7 awakening

thread 8 awakening
main() reporting that all 5 threads have terminated
```

Running many threads that share a global:
The following program creates many threads running concurrently. All of them share the same global counter variable. Of these many threads, one of them is a counting thread that keeps increasing the counter, while all the others keep retrieving and displaying the value of the shared counter.

(thread6_5.c)
```c
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>

#define MAX_THREADS     10

// global (shared) variables:
int      MaxIteration;
int      Counter = 0;

void  *Display(void *ID)
{
    int  *intPTR = (int *) ID;
    int  MyID    = *intPTR;
    int  i;

    for (i = 1; i <= MaxIteration/2; i++)
        printf("Thread %d reporting --> %d\n", MyID, Counter);
    printf("Thread %d is done....\n", MyID);
    thr_exit(0);
}

void  *Counting(void *ID)
{
    int  i;

    while (1) {    /* this is an infinite loop */
        Counter++;
        if (Counter % 10 == 0)
            printf("\t\tFrom Counting(): counter = %d\n", Counter);
    }
}

int main(int argc,  char *argv[])
{
    thread_t  ID[MAX_THREADS];
    size_t    Status;
    int       NoThreads, i, No[MAX_THREADS];

    NoThreads    = atoi(argv[1]);
    MaxIteration = atoi(argv[2]);

    thr_create(NULL, 0, Counting, (void *) NULL, 0, NULL);
    for (i = 0; i < NoThreads; i++) {
        printf("Parent: about to create thread %d\n", i);
        No[i] = i;
        thr_create(NULL, 0, Display, (void *) (&(No[i])), 0, &(ID[i]));
    }
    for (i = 0; i < NoThreads; i++)
        thr_join(ID[i], 0, (void *) &Status);
}
```

Output Example:
```
% thread6_5 2 2

Parent: about to create thread 0
Parent: about to create thread 1
            From Counting(): counter = 10
            From Counting(): counter = 20
            From Counting(): counter = 30
            From Counting(): counter = 40
           …
            From Counting(): counter = 4640
            From Counting(): counter = 4650
Thread 0 reporting --> 4540
Thread 0 is done....
            From Counting(): counter = 4660
            From Counting(): counter = 4670
Thread 1 reporting --> 4540
Thread 1 is done....
            From Counting(): counter = 4680
            From Counting(): counter = 4690
            From Counting(): counter = 4700
           …
            From Counting(): counter = 4910
            From Counting(): counter = 4920
            From Counting(): counter = 4930
```

main program just waits for the completion of all display threads. Note that the counting thread has an infinite loop. Note that once the main exits, the whole program, including the counting thread, exits.

This program looks straightforward. But, it has a subtle problem. Consider the loop for creating all display threads, which is repeated below:

```
for (i = 0; i < NoThreads; i++) {
   printf("Parent: about to create thread %d\n", i);
   No[i] = i;
   thr_create(NULL, 0, Display, (void *) (&(No[i])), 0, &(ID[i]));
}
```

Why is an array **No[]** used in the call to **thr_create()**? Could we just use a variable? In fact, we ***cannot*** use a variable. This is because before the newly created thread takes the value of **i**, the **for** loop could come back and increases the value of **i**. As a result, the newly created thread will receive an incorrect value. This is why an array rather than a variable is used in the call.

(thread6_10.c)

```
#include <thread.h>
#include <sys/wait.h>

typedef struct
{
	int id;
	char name[10];
} customers;

void *Display(void *cust)
{
	customers *d_cust = (customers *) cust;
	printf("customer id: %d, name: %s\n", d_cust->id, d_cust->name);
}

int main(void)
{
	thread_t TaskID;
	customers cust;

	// initialize data
	cust.id = 123;
	strcpy(cust.name,"none");
	// create thread
	thr_create(NULL,0,Display,&cust,0,&TaskID);
	// wait till thread completes
	thr_join(TaskID, 0, NULL);
}
```

```
1 > gcc -lthread -o thread6-10 thread6_10.c
2 > thread6-10
customer id: 123, name: none
```

Note the differences:

```
1 > cat thread.c

#include <thread.h>
#include <sys/types.h>

void *sleeping(void *DontNeedIt);

main()
{
	thr_create(NULL, 0, sleeping, NULL, 0, NULL);
	thr_join(NULL, NULL, NULL);
	printf("main done\n");
}

void *sleeping (void *DontNeedIt)
{
        sleep(20);
        printf("sleeping done\n");
}

2 > gcc -lthread -o threadExample thread.c
3 > threadExample &
[1] 24255
4 > ps -l
F   UID   PID  PPID %C PRI NI   SZ  RSS    WCHAN S TT     TIME COMMAND
8  8385 22112 22107  0  48 20 2536 2216 auth_knc S pts/10 0:00 -tcsh
8  8385 24255 22112  0  58 20 1136  928 E_Syslim S pts/10 0:00 a.out
sleeping done
main done
[1]    Exit 1                    threadExample
```

Only one pid, as opposed to:

```
5 > cat process.c

#include <sys/types.h>

main()
{
        fork();
        sleep(20);
        printf("sleeping done\n");
}

6 > gcc -o processExample process.c
7 > processExample &
[1] 25199
8 > ps -l
F  UID   PID  PPID %C PRI NI   SZ  RSS    WCHAN S TT     TIME COMMAND
8 8385 22112 22107  0  48 20 2536 2216 auth_knc S pts/10 0:00 -tcsh
8 8385 25199 22112  0  48 20  864  568 auth_knc S pts/10 0:00 processExample
8 8385 25200 25199  0  58 20  864  536 auth_knc S pts/10 0:00 processExample
sleeping done
sleeping done

[1]    Exit 1                    a.out
```

two pid's.

Synchronization:

Problem:
The thread library doesn't provide any mechanism for synchronization between the threads.

Example (taken from Berg and Lewis):
(thread6_6.c)
```
#include <stdio.h>
#include <thread.h>

int int_val[5];

/* threaded routine */
void *add_to_value(void *arg)
{
        int inval = (int) arg;
        int i;

        for (i = 0; i < 10000; i++)
                int_val[i % 5] += inval; /* !!! changing global array without synchronization */

        return (NULL);
}

main()
{
        int i;

        /* initialize the data */
        for (i = 0; i < 5; i++)
                int_val[i] = 0;

        for (i = 0; i < 5; i++)
                thr_create(NULL, 0, add_to_value, (void *)(2 * i), THR_BOUND, NULL);

        /* wait till all threads have finished */
        for (i = 0; i < 5; i++)
                thr_join(0, 0, 0);

        /* print the results */
        printf("final values…\n");

        for (i = 0; i < 5; i++)
                printf("integer value [%d] = \t %d\n", i, int_val[i]);

        return(0);
}
```

```
1 > gcc -lthread -o example6 thread6_6.c
2 > repeat 3 example6
final values                   final values                   final values
integer value [0] =   34374    integer value [0] =   34276    integer value [0] =   34274
integer value [1] =   39180    integer value [1] =   39444    integer value [1] =   39280
integer value [2] =   35454    integer value [2] =   35850    integer value [2] =   35514
integer value [3] =   37464    integer value [3] =   37710    integer value [3] =   37664
integer value [4] =   35078    integer value [4] =   35276    integer value [4] =   34768
```

Solution:
Since all the threads belong to a single process they all have access to the global variables. Thus, we need to prevent collisions when accessing data → need for synchronization → using ***locks***. All the

threads, which use the locks, need to cooperate. The kernel doesn't prevent illegal access in case of a lock that wasn't checked or was checked but the checked value was disregarded.

1. Mutex Locks:

Mutex lock enable only *one* thread to hold the lock. All the other threads which try to hold the lock will be blocked until the lock is released → only one thread will be at the critical section at a time.
Mutex locks are the fastest and the most efficient in aspect of memory than all other synchronization mechanisms.

SYNOPSIS
```
#include <thread.h>
#include <synch.h>
int mutex_init(mutex_t *mp, int type, void * arg);
int mutex_lock(mutex_t *mp);
int mutex_trylock(mutex_t *mp);
int mutex_unlock(mutex_t *mp);
int mutex_destroy(mutex_t *mp);
```

DESCRIPTION
All mutexes must be global. A successful call for a mutex lock via mutex_lock() will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it via mutex_unlock().mutex_trylock() is the same as mutex_lock(), except that if the mutex object referenced by *mp* is locked (by any thread, including the current thread), the call returns immediately with an error.
Threads within the same process or within other processes can share mutexes.
Creation of a mutex lock is done using mutex_init() and they are freed using mutex_destroy(). Only the thread which holds the lock can free it.
Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

USAGE
```
#include <synch.h>

mutex_t   Lock;
int  mutex_init(
             mutex_t  *Lock,    /* pointer to a lock */
             USYNC_THREAD,      /* use this          */
             (void *) NULL);    /* always use this   */
```

The second parameter, *type*, can be either:

- USYNC_THREAD: means that the mutex can synchronize threads only in this process (arg is ignored).
- USYNC_PROCESS: means that the mutex can synchronize threads in this process and other processes (arg is ignored).

Example: **(thread6_7.c)**
```
#include <thread.h>
#include <sys/types.h>

mutex_t m;
int count;

/* using mutex to ensure that the update of count is done atomically. return new value */
int increment_count()
{
	int value;

	mutex_lock(&m);
		value = count++;
	mutex_unlock(&m);
```

```
        return value;
}
```

```
/* using mutex to ensure that the memory is synchronized while accessing count */
int get_count()
{
        int c;

        mutex_lock(&m);
                c = count;
        mutex_unlock(&m);
        return c;
}
main()
{
        mutex_init(&m, USYNC_THREAD, NULL);
        … // threads executing increment_count and get_count
        mutex_destroy(&m);
}
```

Question:
Why not write increment_count() like this:

```
int increment_value()
{
        int value;

        mutex_lock(&m);
                count++ ;
        mutex_unlock(&m);
        return count;
}
```

Suppose the **increment_value()** was called when the value of **count** is 2. Then, the value of **count** is changed to 3. At this moment, we would expect **increment_count()** returns 3. Unfortunately, it may not be the case. Before executing the **return** statement, another thread calls **increment_count()** and has the value of **count** changed. So the call to **increment_count()** will not return 3 but some other unexpected values. This is why the new counter value is immediately saved to **value**, which is returned.

Note:

1. If there are several threads locked on a mutex, after the mutex is released, the scheduling policy of the OS.
2. As opposed to semaphores, mutex has ***ownership***, i.e., only the thread who locked the mutex can release it (whereas in semaphores, every thread can increase/decrease the value of the semaphore).
3. If a process that locked the mutex try to lock it again before releasing it, it will be ***deadlocked***.

2. Counting Semaphores:

```
#include <synch.h>

int sema_init(sema_t *sp, unsigned int count, int type, void *arg);
int sema_wait(sema_t *sp);
int sema_trywait(sema_t *sp);
int sema_post(sema_t *sp);
int sema_destroy(sema_t *sp); // free semaphore
```

Creating and Initializing Semaphores:*sema_init()*

```
#include <synch.h>

sema_t   Semaphore;

int  sema_init(
        sema_t  *Semaphore,
        unsigned int   value,   // can't get negative values
        USYNC_THREAD,/* use this */
        (void *) NULL /* default */
        );
```

A call to **sema_init()** will initialize the given semaphore with the specified **value** (the second argument).

The function returns **0** if successful.

As a good programming practice, a semaphore should be initialized once at the very beginning of your program and before the threads which use it are created.

Example:

```
sema_t Semaphore;
int count = 1;
sema_init(&Semaphore, count, USYNC_THREAD, NULL);
```

Semaphore Wait (decreasing semaphore value):*sema_wait():*

```
#include <synch.h>

sema_t   Semaphore;
int  sema_wait(sema_t *Semaphore);
```

If the semaphore counter is zero, the calling thread is blocked.
If the semaphore counter is greater than zero, the counter is subtracted by 1 (atomically) and the calling thread continues.

The function returns **0** if successful.

Semaphore Signal (Increasing semaphore value):*sema_post():*

```
#include <synch.h>

sema_t   Semaphore;
int  sema_post(sema_t *Semaphore);
```

If there are any threads blocked on the semaphore, one is released.
When no threads are blocked, the counter is increased by one.

The function returns **0** if successful.

In addition to be used as locks, semaphores can block the execution of a thread until it is notified by other threads (bound buffer, alternate execution and etc.).

sema_trywait():

```
#include <synch.h>

sema_t   Semaphore;
int  sema_trywait(sema_t *Semaphore);
```

sema_trywait() atomically decrements the semaphore count pointed by Semaphore, if the count is greater than zero. Otherwise, it returns an error.

The function returns **0** if successful.

Example: ***(thread6_8.c)***

Protecting a shared counter:
The following program creates a number of threads running concurrently. All of them share the same global counter variable. They lock the counter and update and display the value of the counter. The lock mechanism is done by a semaphore. This shows the first use of semaphores: a lock!

```
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>
#include <synch.h>

#define  NO_THREADS   5
sema_t   Lock;                 /* the protecting semaphore */
int      Counter, Max_Run;     /* the shared counter       */

void  *Counting(void *voidPTR)
{
   int   *intPTR = (int *) voidPTR;
   int   Name    = *intPTR;
   int   i;

   printf("Thread %d started\n", Name);
   for (i = 0; i < Max_Run; i++) {
      thr_yield();              /* rest for unspecified time*/
      sema_wait(&Lock);         /* enter critical section  */
         Counter++;             /* do updating and printing */
         printf("Thread %d reports: new counter value = %d\n",
                Name, Counter);
      sema_post(&Lock);         /* leaving critical section */
   }
   printf("Thread %d ends\n", Name);
   thr_exit(0);
}

int main(int  argc,  char *argv[])
{
   thread_t   ID[NO_THREADS];             /* thread IDs          */
   size_t   Status[NO_THREADS];           /* thread status       */
   int      Argument[NO_THREADS];   /* thread argument       */
   int      i;

   Max_Run = atoi(argv[1]);

   printf("Parent started ...\n");

   Counter = 0;
   sema_init(&Lock, 1, USYNC_THREAD, (void *) NULL);  /* init sem. */

   printf("Parent is about to create %d threads\n", NO_THREADS);

   for (i = 0; i < NO_THREADS; i++) {  /* create all threads       */
      Argument[i] = i;
      thr_create(NULL, 0, Counting, (void *) &(Argument[i]), 0, (void *) &(ID[i]));
   }

   for (i = 0; i < NO_THREADS; i++) {  /* wait for all threads     */
      thr_join(ID[i], 0, (void *) &(Status[i]));
      printf("Parent found thread %d done\n", i);
   }
```

Output Example:

```
% thread6_8 2

Parent started ...
Parent is about to create 5 threads
Thread 0 started
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 0 reports: new counter value = 1
Thread 1 reports: new counter value = 2
Thread 2 reports: new counter value = 3
Thread 3 reports: new counter value = 4
Thread 4 reports: new counter value = 5
Thread 0 reports: new counter value = 6
Thread 0 ends
Thread 1 reports: new counter value = 7
Thread 1 ends
Thread 2 reports: new counter value = 8
Thread 2 ends
Thread 3 reports: new counter value = 9
Thread 3 ends
Thread 4 reports: new counter value = 10
Thread 4 ends
Parent found thread 0 done
Parent found thread 1 done
Parent found thread 2 done
Parent found thread 3 done
Parent found thread 4 done
Parent exits …
```

```
    printf("Parent exits ...\n");
}
```

Since the semaphore will be used by all threads, it is declared as a global variable. Because only one thread is allowed in the critical section that protects the counter, the initial value of **Lock** is 1. Note that this is done *before* any thread starts.
For each thread, it iterates **Max_Run** times. In each iteration, this thread yields the control of execution to other threads. This is very similar to sleep for an unspecified time. When this thread is rescheduled to run, it tries to enter the critical section with **sema_wait()**. If it succeeds, the value of the counter is increased by one and the new value is displayed. Finally, it exits the critical section with **sema_post()** and goes back for the next iteration.

More examples:
http://www.csl.mtu.edu/cs4411.ck/www/NOTES/threads/buffer.html

3. Reader/Writer Locks:

Reader/Writer locks enable reading of protected object by several threads concurrently. They also enable for only one thread to write when no other thread is reading.
Reader/Writer locks are useful for protection of data that is often read and written occasionally. Many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time.

```
int rwlock_init(rwlock_t *rwlp, int type, void * arg);
int rwlock_destroy(rwlock_t *rwlp);
int rw_rdlock(rwlock_t *rwlp);
int rw_wrlock(rwlock_t *rwlp);
int rw_unlock(rwlock_t *rwlp);
int rw_tryrdlock(rwlock_t *rwlp);
int rw_trywrlock(rwlock_t *rwlp);
```

Creation: ***rwlock_init()*:**
Readers/writer locks must be initialized prior to use. *type* can be either USYNC_PROCESS or USYNC_THREAD. We'll always use USYNC_THREAD. *arg* is currently not used.

Example:
```
rwlock_t rwlp;
rwlock_init(&rwlp, USYNC_THREAD, NULL);
```

Freed: *rwlock_destroy()*.

Get read lock: ***rw_rdlock():***
Gets a **read** lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for **writing**, the calling thread blocks until the write lock is freed. Multiple threads may simultaneously hold a read lock on a readers/writer lock.

rw_tryrdlock():
Trys to get a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is locked for writing, it returns an error; otherwise, the read lock is acquired.

Get write lock: ***rw_wrlock():***
Gets a **write** lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for **reading** or **writing**, the calling thread blocks until all the read and write locks are freed. At any given time, only **one** thread may have a write lock on a readers/writer lock.

rw_trywrlock():
Trys to get a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is currently locked for reading or writing, it returns an error.

<u>Free lock:</u> *rw_unlock()*
Unlocks a readers/writer lock pointed to by *rwlp*, if the readers/writer lock is locked and the calling thread holds the lock for either reading or writing. One of the other threads that is waiting for the readers/writer lock to be freed will be unblocked.
If the calling thread does not hold the lock for either reading or writing, no error status is returned, and the behavior of the program is unknown.

<u>Note:</u>
If multiple threads are waiting for a readers/writer lock, the acquisition order is random by default. However, some implementations may bias acquisition order to avoid depriving writers. The current implementation favors writers over readers.

<u>Example:</u> **(thread6_9.c)**

```
/* many threads can read the balance, but only one thread can change it */
#include <thread.h>
#include <synch.h>

rwlock_t account_lock;
float checking_balance = 100;
float saving_balance = 100;

float get_balance();
void transfer_checking_to_saving(float amount);

main()
{
        rwlock_init(&account_lock, 0, NULL);
        printf("%f", get_balance());
        transfer_checking_to_saving(5);
        rwlock_destroy(&account_lock);
}

float get_balance()
{
        float bal;

        rw_rdlock(&account_lock);
        bal = checking_balance + saving_balance;
        rw_unlock(&account_lock);
        return bal;
}

void transfer_checking_to_saving(float amount)
{
        rw_wrlock(&account_lock);
        checking_balance = checking_balance - amount;
        saving_balance = saving_balance + amount;
        rw_unlock(&account_lock);
}
```