Exercises, mostly on \textit{pthread}

1 Semaphores

A semaphore is an old fashioned synchronisation primitives that generalises the mutex: the semaphore is given a \textit{capacity} and at most capacity threads can be in critical section simultaneously — hence, a mutex is a semaphore with capacity 1.

1.1 Coding a semaphore

Given a semaphore $s$ initialised to capacity $c$, critical sections are defined from a call to \texttt{wait_semaphore ($s$)} (analog of \texttt{lock_mutex}) to \texttt{post_semaphore ($s$)} (analog of \texttt{unlock_mutex}). The semaphore uses an internal counter \texttt{nfrr} to count the number of threads allowed to enter critical section. The counter is initialised to $c$ at semaphore creation time, then:

- \texttt{wait_semaphore ($s$)} checks that \texttt{nfrr} is non-null and decrements it. If \texttt{nfrr} is null, the thread suspends.
- \texttt{post_semaphore ($s$)} increments \texttt{nfrr} and release waiting threads.

One may write a semaphore with a mutex (to protect the modifications of \texttt{nfrr}) and a condition variable (to wait on). Complete the following code:

\begin{verbatim}
/* Signature of mutex and condition variable primitives */

pthread_mutex_t *alloc_mutex(void) ;
void free_mutex(pthread_mutex_t *p) ;
void lock_mutex(pthread_mutex_t *p) ;
void unlock_mutex(pthread_mutex_t *p) ;

pthread_cond_t *alloc_cond(void) ;
void free_cond(pthread_cond_t *p) ;
void wait_cond(pthread_cond_t *c, pthread_mutex_t *m) ;
void signal_cond(pthread_cond_t *c) ;
void broadcast_cond(pthread_cond_t *c) ;

/* Semaphore structure */
typedef struct {
    volatile int nfrr ;
    pthread_mutex_t *mutex ;
    pthread_cond_t *cond ;
} semaphore_t ;

semaphore_t *alloc_semaphore(int capacity) { ... }

void free_semaphore(semaphore_t *p) { ... }
\end{verbatim}
void wait_semaphore(semaphore_t *p) { ... }
void post_semaphore(semaphore_t *p) { ... }

1.2 Semaphore usage

We consider nprocs threads running function T2 below, with argument described by ctx_t below:

typedef struct {
    int size ;
    mybarrier_t *b ;
    semaphore_t *sem ;
} common_t ;

typedef struct {
    int id ;
    common_t *common ;
} ctx_t ;

void *T2(void *p) {
    ctx_t *ctx = p;
    common_t *cm = ctx->common;
    for (int k = cm->size-1; k >= 0; k--) {
        wait_semaphore(cm->sem);
        printf("+");
        printf("-");
        post_semaphore(cm->sem);
        wait_mybarrier(cm->b);
        if (ctx->id == 0) printf("\n");
        wait_mybarrier(cm->b);
    }
    return NULL;
}

With a semaphore of capacity 2, q->size = 1 and nprocs == 4. Classify the following outputs as legal or illegal, giving a short explanation in each case:

1. ++++-+
2. +++---
3. -+++++
4. ++++++
5. ++++++

2 A concurrent component

We aim at building a concurrent component on top of POSIX threads. The component and_t operates for nprocs participant threads, the number of participant threads being fixed at component allocation time:

and_t *alloc_and(int nprocs);

Each thread will call the following function:

int wait_and(and_t *p, int b);
where $b$ is some integer encoding a boolean (i.e. 0 is false, while 1 is true). The call `wait_and` returns only when all participants have submitted their boolean and returns the conjunction (and) of all submitted booleans. Hence the `and_t` component looks very much like a synchronisation barrier that additionally returns a boolean.

It is important to notice that participants can call `wait_and` several times, just as they can call a POSIX synchronisation barrier several times.

### 2.1 Barrier encoding

We first write the component by the means of a POSIX synchronisation barrier (described in slides 8–9 of lesson 2). Here is the type of component and the `alloc_and` function:

```c
typedef struct {
    pthread_barrier_t *b ; // Posix synchronisation barrier
    int v ;               // You'll need that field to compute result
} and_t ;

and_t *alloc_and(int nprocs) {
    and_t *p = malloc_check(sizeof(*p)) ;
    p->v = 1 ;         // This is the conjunction of zero boolean.
    p->b = alloc_barrier(nprocs) ;
    return p ;
}
```

Write the `wait_and` function. You’ll probably have to call `wait_barrier (p->b)` several times.

### 2.2 Direct coding

We now write the component by the means of the basic POSIX synchronisation primitives: locks and condition variables.

Here is an incomplete definition of type `and_t` and an incomplete `alloc_and` function:

```c
typedef struct {
    pthread_cond_t *cond ;
    pthread_mutex_t *mutex ;
    /* Hum the following should remind you of something... */
    int nprocs,count ;
    int turn ;
    ...;
} and_t ;

and_t *alloc_and(int nprocs) {
    and_t *p = malloc_check(sizeof(*p)) ;
    p->cond = alloc_cond();
    p->mutex = alloc_mutex() ;
    p->nprocs = p->count = nprocs ;
    p->turn = 0 ;
    ...;
    return p ;
}
```

Complete the above definitions and write the `wait_and` function. You can start from the code of `wait_barrier` on slide 18 of lesson 2.

### 3 A process farm

We aim at building a simple “process farm” framework:
A worker will perform a computation. More precisely given a task $x$, a worker computes $y = F(x)$ and accumulate in a “running” result $r$ by calling a function $C (r = C(y, r))$.

A master will allocate some tasks to workers and control their execution.

Additionally there cannot be more than $nprocs$ workers running concurrently.

We have already seen such a framework in class 01 based upon a FIFO. Here we aim at another solution based upon two components: a pool that will manage worker allocation, checking that no more than $nprocs$ are running concurrently, and a monitor that will manage computation of partial results and termination. Notice that our process farm will create one (POSIX) thread per task$^1$.

In practice, you have to write C code for those two components from the templates in directory pool. The pool directory also contains two examples the simple tst.c example and the more sophisticated run.c example: The associated Makefile builds the executables tst.out and run.out.

- The tst example computes the sum of the first $n$ integers (i.e. $F(x) = x$ and $C(y, r) = y + r$).
- The run example computes the number of polyominoes of size $n$, using the code presented in class 01. That is, given $x$ the description of a partial polyomino of size $n - d$, $F$ returns the number of polyominoes of size $n$ that contain $x$; and $C(y, r) = y + r$ again.

**Important:** You have to complete the source files pool.c and monitor.c. Once you are done, you can test your components as follows:

```
% make
% ./tst.out
5050
% ./run.out
27394666
```

You can also try ./tst.out nprocs $n$, to sum the $n$ first integers using $nprocs$ cores; or ./run.out -j nprocs $n$ to compute the number of polyominoes of size $n$ using $nprocs$ cores. Both examples will output some information on what happens if you give them the command-line option -v, which can be repeated for more diagnostics.

We now describe the simple example tst.out, so as to demonstrate the pool and monitor components usage.

**Pool**

The master simply executes a loop from 1 to $n$, spawning a worker for each loop indice value:

```c
typedef struct {
    pool_t *pool ;
    monitor_t *monitor ;
} common_t ;

void master(int nprocs, int n) {
    common_t c ;
    ...
    c.pool = alloc_pool(nprocs) ;
    for (int k = 1 ; k <= n ; k++) {
        look_pool(c.pool) ;
        spawn_worker(k,&c) ;
    }
    ...
```

$^1$Threads can be cached by another component so as to amortised thread creation costs. We neglect this issue.
More precisely, `look_pool` will suspend if `nprocs` or more workers are already running. Otherwise, `look_pool` returns immediately having altered the pool structure that will remember that a worker is running.

It will be the worker responsibility to inform the pool when it becomes available again. In the simple example, it works as follows:

```c
typedef struct {
  int arg ;
  common_t *common ;
} worker_t ;

void *worker(void *p) {
  worker_t *w = (worker_t *)p ;
  common_t *c = w->common ;
  ...
  leave_pool(c->pool) ;
  return NULL ;
}

void spawn_worker(int arg, common_t *c) {
  worker_t *w = alloc_worker_t(arg,c) ;
  create_thread_detached(worker,w) ;
}
```

That is, the worker thread is created detached (i.e. we shall not join on it) to execute `worker` with the appropriate argument that includes the task (here `arg`) and a pointer to `common` that in turn records pointers to the pool and monitor components. The `worker` code performs the allocated work (not shown yet...), and finally informs the pool that a new worker gets available by calling `leave_pool` just before exiting. In case the master is suspended, `leave_pool` should awake it.

Here are the signatures of the two functions you have to write:

```c
typedef struct {
  int maxrun,nrun ; /* Max number of running workers, running workers */
  int waiting ;    /* flag, true when master is waiting */
  pthread_mutex_t *lock ;
  pthread_cond_t *cond ;
} pool_t ;
...

/* To be called by worker: tell pool a worker is free, should awake master if suspended */
void leave_pool(pool_t *p) ;

/* To be called by master: allocate a worker, suspend when none is available */
void look_pool(pool_t *p) ;
```

Monitor

The monitor component manages result computation and program termination. Result computation is performed incrementally by accumulating partial results by the mean of the `C` function that will be hidden in the monitor.

We first examine its interface with the worker:

```c
void *worker(void *p) {
  worker_t *w = (worker_t *)p ;
  common_t *c = w->common ;
  int arg = w->arg ;
```
int y = compute(arg);
...
leave_monitor(c->monitor,y);
return NULL;
}

Hence, the worker computes. It then passes the partial result y to the monitor, for it to accumulate partial results into the final result.

The interface with the master is as follows:

uintmax_t add(uintmax_t y, uintmax_t r) { return y+r; }

void master(int nprocs, int n) {
    common_t c;
    c.monitor = alloc_monitor(add,0);
    c.pool = alloc_pool(nprocs);
    for (int k = 1; k <= n; k++) {
        look_pool(c.pool);
        enter_monitor(c.monitor);
        spawn_worker(k,&c);
    }
    int r = wait_monitor(c.monitor,n); 
}

The master first creates the monitor with alloc_monitor(add,0), arguments are the C function (here a simple addition function) and the initial value of result (here 0). Then, the master create all tasks (and spawn all workers) with the for loop.

Observe that the master calls enter_monitor before spawning the worker. It does so to inform the monitor that a new task is being computed. In practice, the monitor will record the number of tasks being computed with some internal counter. Of course leave_monitor (called by workers) should now also decrease this internal counter.

Finally the master wait on the monitor, passing it the number of generated tasks as an argument. The function wait_monitor should behave as follows:

- If n tasks are completed then return the accumulated result.
- Otherwise suspend.

Hence, if the master suspends, someone should awake it. This will be the responsibility of the last worker that calls leave_monitor. The internal counter of tasks being computed may help workers to know when they are this last worker.

Here are the signatures of the three functions you have to write:

typedef struct {
    int nrun;          /* number of tasks being executed */
    int ncompleted;    /* number if tasks being completed */
    int waiting;       /* flag set if master is waiting */
    pthread_mutex_t *lock;
    pthread_cond_t *cond;
    compose_t compose; /* compose function */
    uintmax_t r;       /* result of computation */
} monitor_t;

/*
   To be called by worker :
*/
1. Pass partial result y, so as to update result of computation
   \[ m->r = m->compose(y,m->r) \]
2. Signals a task is completed

*/
void leave_monitor(monitor_t *m, uintmax_t y);

/* To be called by master to signal a task is being executed */
void enter_monitor(monitor_t *m);

/* To be called by master to wait for ntasks being completed.
   returns computation result */
uintmax_t wait_monitor(monitor_t *m, int ntasks);

4 Controlling workers with a stack

We aim at controlling a set of nprocs worker threads by the mean of a stack. The stack will be a concurrent, bounded, blocking stack. This means (“bounded”) that the stack is of limited capacity (from now sz) and (“blocking”) that attempting to push on a full stack or to pop from an empty stack will block the calling thread.

The exercise has two steps: first (4.1) write push and pop operations that are blocking; and second (4.2) write a kill functionality that controls termination.

We provide a starting point for you to write the code, in sub-directory stack, with two testing applications tst.out and run.out. The former application tst.out is a simple test than spawn nprocs “popper” threads:

```c
void *popper(void *p) {
    // Various initialisation from p
    ...
    void *item;
    while (((item = pop(c->stack)) != NULL) {
        boxed_int_t *q = item;
        int v = q->v;
        free_boxed_int(q);
        (void)__sync_fetch_and_add(&c->sum,v);
        if (verbose) fprintf(stderr,"POPPER%i GOT%in",id,v);
    }
    if (verbose) fprintf(stderr,"POPPER%i OUT",id);
    return NULL;
}
```

Hence, a popper pops items from the stack, until NULL is returned. The popped item is a boxed integer, whose contents is added atomically to a running sum, which is common to all poppers.

Moreover there are nprocs “pusher” threads that will push items on the stack:

```c
void *pusher(void *p) {
    // Various initialisations from p
    ...
    // push 1 id+1 times, id is pushed id in 0,...,nprocs-1
    for (int k = 0 ; k <= id ; k++) {
        if (verbose) fprintf(stderr,"PUSHER%i PUT%in",id,1);
        push(c->stack,alloc_boxed_int(1));
    }
    return NULL;
}
```
Hence, the \texttt{nprocs} pushers will push the integer “1” \(1 + 2 + \cdots + \texttt{nprocs}\) times. As a result, reading the accumulating sum once all pushers and poppers have finished, should yield the value \(1 + 2 + \cdots + \texttt{nprocs}\). For instance, with default value 2 for \texttt{nprocs} and 100 for \texttt{sz} the size of the stack, we should get:

\%
./tst.out -v
nprocs=2, sz=100
PUSHER<1> PUT 1
PUSHER<0> PUT 1
PUSHER<1> PUT 1
POPPER<0> GOT 1
POPPER<0> GOT 1
POPPER<0> GOT 1
POPPER<0> OUT
SUM=3, OK=3
POPPER<1> OUT

The second test computes the number of polyominoes of size \(p\), as we have seen in the first class. With default value of 15 for \(p\), we get:

\%
./run.out
27394666

Running “./run.out -v” gives additional information.

\subsection{4.1 Concurrent push and pop}

Write blocking \texttt{push} and \texttt{pop} function.

The testing source (sub-directory \texttt{stack}) includes starting code for the stack (files \texttt{stack.h} and incomplete \texttt{stack.c}), our wrappers around POSIX thread operations (\texttt{basic.h} and \texttt{basic.c}), and complete code for the test applications \texttt{tst.c} and \texttt{run.c}.

The starting code in \texttt{stack.c} contains complete \texttt{alloc_stack} and \texttt{free_stack} functions, and wrong attempts for \texttt{push} and \texttt{pop}. As a result attempting to run \texttt{./tst.out} (or \texttt{./run.out}) may fail:

\%
./tst.out
Segmentation fault (core dumped)

Here is for instance the wrong code for \texttt{push}:

\begin{verbatim}
void *pop(stack_t *p) {
  void *r ;
  while (p->sp <= 0) ;
  p->sp-- ; r = p->t[p->sp] ;
  return r ;
}
\end{verbatim}

Notice that the stack includes an array \(p->t\) of size \(p->sz\) and that \(p->sp\) is the stack pointer. As usual, \(p->sp\) is the indice of the next free position in the stack.

Correct code will probably use the mutex \(p->lock\) and the two condition variables \texttt{is_empty} and \texttt{is_full} that are already present in the stack structure definition (defined in \texttt{stack.h}) and properly initialised by \texttt{allocate_stack} (defined in \texttt{stack.c}). You may draw inspiration from the bounded FIFO of class 01.

Once you have written correct \texttt{push} and \texttt{pop} functions, you still may get wrong sums, as termination is not handled properly yet:

\%
./tst.out -v
nprocs=2, sz=100
PUSHER<0> PUT 1
You may have to run the experiment more than once to get a wrong result (i.e. SUM different from OK=3). Also notice that the de-allocation of resources is not properly performed.

4.2 Controlling termination

She shall now enrich our stack with a “kill” functionality that behaves as follows:

- `kill(stack_t *p, int nprocs)` should be called at most once and “kills” the stack. The call to `kill` is blocking and will return once the kill has been acknowledged `nprocs` times (see `pop` below).

- Once `kill` has been called, calling `push` is an error.

- Attempting to pop a stack that is both killed and empty should return `NULL` and acknowledge the kill once.

Hence, you should alter your working `push` and `pop` functions from 4.1 and write the `kill` function. To that aim, you may use new fields for the stack structure: the flag `killed` (to register the kill), the integer `seen` (to count acknowledgements), and the condition variable `wait` (for the killer to suspend on, waiting for acknowledgements).

So as to describe the kill functionality in greater detail, here are the relevant code snippets from `tst.c`. First we recall that poppers exit when `pop` returns `NULL`:

```c
void *popper(void *p) {
    // Various initialisation from p
    ...
    void *item ;
    while ((item = pop(c->stack)) != NULL) {
        ...
    }
    if (verbose) fprintf(stderr,"POPPER<%i>\OUT\n",id) ;
    return NULL ;
}
```

Then, here is the code that creates poppers and pushers:

```c
/* Create n poppers */
common_popper_t *spawn_poppers(stack_t *stack, int n) {
    common_popper_t *c = alloc_common_popper(stack);
    for (int id = 0 ; id < n ; id++) {
        popper_t *w = alloc_popper_t(id,c) ;
        create_thread_detached(popper,w) ;
    }
    return c ;
}

/* Create n pushers */
common_pusher_t *spawn_pushers(stack_t *stack, int n) {
    common_pusher_t *c = alloc_common_pusher(stack);
    pthread_t th[n] ;
    for (int id = 0 ; id < n ; id++) {
```
It can be noticed:

- Both functions allocate specific “common” arguments for poppers and pushers, noticeably to hold a pointer to the common stack. Those arguments are returned so as to be de-allocated once termination is ensured.
- While poppers are created detached (their termination is handled through the kill functionality), the pushers are joined. As a result, when spawn_pushers returns, we can be sure that all pushes have been performed.

Finally, here is the overall thread control:

```c
void zyva(int nprocs, int sz) {
    if (verbose) fprintf(stderr, "nprocs=%i, sz=%i\n", nprocs, sz);
    // Allocate stack and start all threads
    stack_t *stack = alloc_stack(sz);
    common_popper_t *pop = spawn_poppers(stack, nprocs);
    common_pusher_t *push = spawn_pushers(stack, nprocs);
    // Kill stack
    kill(stack, nprocs);
    // Get and check result
    int sum = __sync_fetch_and_add(&pop->sum, 0);
    int ok = 0;
    for (int k = 1; k <= nprocs; k++) ok += k;
    printf("SUM=%i, OK=%i\n", sum, ok);
    // Free all data structures
    free_common_popper(pop);
    free_common_pusher(push);
    free_stack(stack);
}
```

Observe:

- The stack is killed only after spawn_pushers has returned. As a consequence, and because we know that all pushers have terminated before spawn_pushers returns, we know that no further push will ever occur.
- The function kill will return only once the nprocs poppers have acknowledged the kill. As a result, pop->sum is valid. Further notice how pop->sum is read, for greater safety — however it can be argued that the kill/pop synchronisation suffices to allow an ordinary read of pop->sum.
- Furthermore, (see pop code), no popper will access its “common” argument, nor the stack once it has acknowledged the kill. Hence, freeing the pop (poppers common argument) where we do is safe.

Once you have completed you kill functionality, try:

```
% ./tst.out -v
nprocs=2, sz=100
PUSHER<1> PUT 1
PUSHER<0> PUT 1
PUSHER<1> PUT 1
10
```
5 Transitive visibility

One of your friends works at Intel and argues that, on processors, “stores obey transitive visibility”. As you wonder what “transitive visibility” is, he writes the following three functions, to be executed concurrently:

```c
int x=0, y=0;

void writer(void) {
  x = 1;
}

void transmitter(void) {
  int r = x;
  y = r;
}

void reader(void) {
  int ry = y;
  int rx = x;
}
```

He then argues that the reader thread must see `rx == 1` whenever it sees `ry == 1`. Said otherwise, an execution where `ry == 1` and `rx == 0` is not possible.

Is your friend right? To answer, you can draw a diagram for the test, similar to the ones of lesson 03, and consider that Intel processors are TSO.

6 A memory model zoo

Here are the definitions of the SC, TSO and PSO (Partial Store Order) memory models in the axiomatic formalism we used in class (see class 03 slides 20 and 62):

```plaintext
(* SC Model *)
acyclic po | rf | fr | co

(* TSO Model *)
acyclic po-loc | rf | fr | co
acyclic (po \ (W*R)) | rfe | fr | co

(* PSO Model *)
acyclic po-loc | rf | fr | co
acyclic (po \ (W*M)) | rfe | fr | co
```
In the above languages expressions are either event sets (such as $M$) or relations (such as $po$, $rf$ etc.). Binary operators used are union “$|$”, difference “$\setminus$” and Cartesian product “$*$”. Some sets are pre-defined: write events “$W$” read events “$R$” and all memory events “$M$” — Notice that $M$ can be defined as $R|W$. Hence, for instance, $po \setminus (W*R)$ is the program-order relation minus write-to-read pairs.

Consider the four tests of Figure 1. Those tests are written in pseudo-code: $x$, $y$ are memory locations, $r0$, $r1$ are registers, all locations are initialised to zero.

A test is valid on a model (written $Ok$), when the final condition of the test can be observed to be true, once a machine that implements the model has run the test. Otherwise the test is invalid, which we write $No$.

Fill the cells of following table with $Ok$ or $No$, depending upon the result of each test on each model.

<table>
<thead>
<tr>
<th></th>
<th>Test $2+2W$</th>
<th>Test MP</th>
<th>Test $R$</th>
<th>Test LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T0$</td>
<td>$x \leftarrow 2</td>
<td>y \leftarrow 2$</td>
<td>$x \leftarrow 1</td>
<td>r0 \leftarrow y$</td>
</tr>
<tr>
<td>$T1$</td>
<td>$y \leftarrow 1</td>
<td>x \leftarrow 1$</td>
<td>$y \leftarrow 1</td>
<td>r1 \leftarrow x$</td>
</tr>
<tr>
<td></td>
<td>$x = 2 \land y = 2$</td>
<td>$r0 = 1 \land r1 = 0$</td>
<td>$r0 = x</td>
<td>r1 \leftarrow y$</td>
</tr>
</tbody>
</table>

Then argue that any test valid on TSO is also valid on PSO.