

Exercises, mostly on pthread

1 Semaphores

A semaphore is an old fashioned synchronisation primitives that generalises the mutex: the semaphore is given a *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 `wait_semaphore(s)` (analog of `lock_mutex`) to `post_semaphore(s)` (analog of `unlock_mutex`). The semaphore uses an internal counter `nfree` to count the number of threads allowed to enter critical section. The counter is initialised to c at semaphore creation time, then:

- `wait_semaphore(s)` checks that `nfree` is non-null and decrements it. If `nfree` is null, the thread suspends.
- `post_semaphore(s)` increments `nfree` and release waiting threads.

One may write a semaphore with a mutex (to protect the modifications of `nfree`) and a condition variable (to wait on). Complete the following code:

```
/* 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 nfree ;  
    pthread_mutex_t *mutex ;  
    pthread_cond_t *cond ;  
} semaphore_t ;
```

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

```
void free_semaphore(semaphore_t *p) { ... }
```

```

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 *p = _p ;
    common_t *q = p->common ;
    for (int k = q->size-1 ; k >= 0 ; k--) {
        wait_semaphore(q->sem) ;
        printf("+") ;
        printf("-") ;
        post_semaphore(q->sem) ;
        wait_mybarrier(q->b) ;
        if (p->id == 0) printf("\n") ;
        wait_mybarrier(q->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 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:

```

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.

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 task¹.

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.

¹Threads can be cached by another component so as to amortised thread creation costs. We neglect this issue.

Pool

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

```
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) ;
    }
    ...
}
```

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:

```
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:

```
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:

```

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 of 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:

```
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_%i\n", id, v) ;
    }
    if (verbose) fprintf(stderr, "POPPER<%i>_OUT\n", 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:

```
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_%i\n", id, 1) ;
        push(c->stack, alloc_boxed_int(1)) ;
    }
    return NULL ;
}
```

Hence, the `nprocs` pushers will push the integer “1” $1 + 2 + \dots + nprocs$ times. As a result, reading the accumulating sum once all pushers and poppers have finished, should yield the value $1 + 2 + \dots + nprocs$. For instance, with default value 2 for `nprocs` and 100 for `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.

4.1 Concurrent push and pop

Write blocking `push` and `pop` function.

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

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

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

Here is for instance the wrong code for `push`:

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

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 `is_empty` and `is_full` that are already present in the stack structure definition (defined in `stack.h`) and properly initialised by `allocate_stack` (defined in `stack.c`). You may draw inspiration from the bounded FIFO of class 01.

Once you have written correct `push` and `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
PUSHER<1> PUT 1
PUSHER<1> PUT 1
POPPER<0> GOT 1
SUM=1, OK=3
pthread_mutex_destroy: Device or resource busy
```

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:

```
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:

```
:
/* 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++) {
        pusher_t *w = alloc_pusher_t(id,c) ;
        create_thread(&th[id],pusher,w) ;
    }
    for (int id = 0 ; id < n ; id++) join_thread(&th[id]);
    return c ;
}

```

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:

```

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` (popper's common argument) where we do is safe.

Once you have completed your kill functionality, try:

```
% ./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
```

And:

```
% ./run.out
27394666
% ./run.out 18
1540820542
```

Solutions

1 Semaphores

1.1 Coding a semaphore

```
/* Semaphore */

typedef struct {
    volatile int nfree ;
    pthread_mutex_t *mutex ;
    pthread_cond_t *cond ;
} semaphore_t ;

semaphore_t *alloc_semaphore(int capacity) {
    semaphore_t *p = malloc_check(sizeof(*p)) ;
    p->nfree = capacity ;
    p->mutex = alloc_mutex() ;
    p->cond = alloc_cond() ;
    return p ;
}

void free_semaphore(semaphore_t *p) {
    free_mutex(p->mutex) ;
    free_cond(p->cond) ;
    free(p) ;
}

void wait_semaphore(semaphore_t *p) {
    lock_mutex(p->mutex) ;
    while (p->nfree <= 0) wait_cond(p->cond,p->mutex) ;
    p->nfree-- ;
    unlock_mutex(p->mutex) ;
}

void post_semaphore(semaphore_t *p) {
    lock_mutex(p->mutex) ;
    p->nfree++ ;
    broadcast_cond(p->cond) ;
    unlock_mutex(p->mutex) ;
}
```

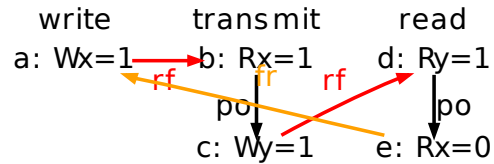
1.2 Semaphore usage

The constraints on output are the same as for T1 plus the additional constraint that no more than $c = 2$ “+” will ever follow with no “-” in-between.

1. ++--++-- is legal, cf. above.
2. ++++---- is illegal, as there are 3 “+” in a row, hence 3 threads should be in critical section at the same time.
3. -+++---- is illegal, no “-” can be without a matching “+” before.
4. +++--++- is legal, and is still legal with $c = 1$.
5. ++++++++ is illegal and is a joke.

2 Transitive visibility

Here is the diagram:



The diagram follows from:

1. As the **reader** reads value 1 in *y* (written by the **transmitter**), it must be that the **transmitter** has read value 1 in *x* (written by **writer**). Hence the two \xrightarrow{rf} arrows.
2. As the **reader** reads initial value 0 in *x*, it must be that the read is in \xrightarrow{fr} with the sole write to *x* by **writer**.

We see first that the execution is not sequentially consistent, as we have a cycle in $\xrightarrow{po} \cup \xrightarrow{fr} \cup \xrightarrow{rf} \cup \xrightarrow{co}$. The execution is not TSO either, given that no $W \xrightarrow{po} R$ edge is present in the cycle.

3 Processor farm

The solutions are given in files `monitor.sol.c` and `pool.sol.c` in directory `pool`.

We here reproduce our solution:

Pool

```
void leave_pool(pool_t *p) {
    lock_mutex(p->lock) ;
    p->nrun-- ;
    if (p->waiting) signal_cond(p->cond) ;
    unlock_mutex(p->lock) ;
}

void look_pool(pool_t *p) {
    lock_mutex(p->lock) ;
    p->waiting = 1 ;
    while (p->nrun >= p->maxrun) {
        wait_cond(p->cond, p->lock) ;
    }
    p->waiting = 0 ;
    p->nrun++ ;
    unlock_mutex(p->lock) ;
}
```

The solution is quite straightforward: `look_pool` increases the number of running workers, while `leave_pool` decreases it.

The master (cf. `look_pool`) suspends when `maxrun` workers are already running and is awoken by any worker that leaves the pool. Observe that we guard against spurious wakeups in `look_pool` by the means of the classical **while** loop on the sleeping condition.

Monitor

```

/* Called by master */
void enter_monitor(monitor_t *m) {
    lock_mutex(m->lock) ;
    m->nrun++ ;
    unlock_mutex(m->lock) ;
}

/* Called by worker */
void leave_monitor(monitor_t *m, uintmax_t y) {
    lock_mutex(m->lock) ;
    m->nrun-- ;
    m->ncompleted++ ;
    m->r = m->compose(y, m->r) ;
    if (m->waiting && m->nrun == 0) signal_cond(m->cond) ;
    unlock_mutex(m->lock) ;
}

/* Called by master */
uintmax_t wait_monitor(monitor_t *m, int ntasks) {
    uintmax_t r ;
    lock_mutex(m->lock) ;
    m->waiting = 1 ;
    while (m->ncompleted < ntasks) wait_cond(m->cond, m->lock) ;
    assert(m->nrun == 0) ;
    m->waiting = 0 ;
    r = m->r ;
    unlock_mutex(m->lock) ;
    return r ;
}

```

The only difficult point is the handling of master sleep and wakeup. Quite logically, the master sleeps when strictly less than `ntasks` have been completed. The worker that is last to complete should awake the master in `leave_monitor`. A worker can know it is the last to complete by checking the condition `m->nrun == 0`, because the *master* called `enter_monitor` and thus has incremented `m->nrun`.

If workers were calling `enter_monitor`, then there would be no guarantee that all tasks are completed when `m->nrun == 0`. Consider a scenario where the master leaves the `for` loop while two tasks have been allocated but the workers have not started yet and all other workers have terminated. The value of `m->nrun` is then 0. The master then suspends. One worker start, increasing `m->nrun` to 1, performs its work and then decreases `m->nrun` to 0. As a result the master is awoken, while one task is still pending.

Notice that `enter_monitor` can be called by workers (in fact suppressed) by a different design: the call `wait_monitor(m, ntasks)` would record `ntasks` into the monitor structure, for workers that call `leave_monitor()` to compare `ntasks` with `m->ncompleted`. Then, the last worker to complete would know it is last by the condition `ntasks == m->ncompleted`.

Finally as noticed by a student, the original version of the two examples of had a bug: the master handles program termination as follows:

```

...
uintmax_t r = wait_monitor(...) ;
free_pool(c.pool) ;
free_monitor(c.monitor) ;
...
}

```

That is, once `wait_monitor` has returned, the master de-allocates the pool and monitor structures. While the buggy worker code was as follows:

```

void *worker(void *p) {

```

```

    worker_t *w = (worker_t *)p ;
    common_t *c = w->common ;
...
    leave_monitor(c->monitor,y) ;
    leave_pool(c->pool) ;
    return NULL ;
}

```

Thus, it may be that a worker attempts to access the pool structure after (while!) it has been freed. The bug can be corrected by swapping the calls to `leave_monitor` and `leave_pool` in `worker`, so as to perform the call to `leave_monitor` last:

```

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

```

The issue is solved (up to weak memory effects...). Nevertheless, our minimal “process farm” framework appears quite brittle. This can be alleviated by grouping the pool and monitor structures in one single component.

4 Controlling workers with a stack

4.1 Concurrent push and pop

The solution is in file `stack.partial.c`. The code is standard and handles spurious wakeups:

```

void push(stack_t *p, void *q) {
    lock_mutex(p->lock) ;
    while (p->sp >= p->sz) wait_cond(p->is_full,p->lock) ;
    int was_empty = p->sp <= 0 ;
    p->t[p->sp] = q ; p->sp++ ;
    if (was_empty) broadcast_cond(p->is_empty) ;
    unlock_mutex(p->lock) ;
}

```

```

void *pop(stack_t *p) {
    void *r ;
    lock_mutex(p->lock) ;
    while (p->sp <= 0) wait_cond(p->is_empty,p->lock) ;
    int was_full = p->sp >= p->sz ;
    p->sp-- ; r = p->t[p->sp] ;
    if (was_full) broadcast_cond(p->is_full) ;
    unlock_mutex(p->lock) ;
    return r ;
}

```

4.2 Controlling termination

The solution is in file `stack.sol.c`.

The easiest alteration is for `push`. We simply crash the program in case of a push attempt on a killed stack:

```

static void stack_error(char *msg) {
    fprintf(stderr, "stack: %s\n", msg) ;
    exit(2);
}

void push(stack_t *p, void *q) {
    lock_mutex(p->lock) ;
    if (p->killed) stack_error("pushing on a killed stack");
    while (p->sp >= p->sz) wait_cond(p->is_full, p->lock) ;
    int was_empty = p->sp <= 0 ;
    p->t[p->sp] = q ; p->sp++ ;
    if (was_empty) broadcast_cond(p->is_empty) ;
    unlock_mutex(p->lock) ;
}

```

Now, here is the code for kill:

```

void kill(stack_t *p, int nprocs) {
    lock_mutex(p->lock) ;
    if (p->killed) stack_error("killing the stack more than once");
    p->killed = 1 ;
    broadcast_cond(p->is_empty) ;
    while (p->seen < nprocs) wait_cond(p->wait, p->lock) ;
    unlock_mutex(p->lock) ;
}

```

After the lock has been acquired:

- Double kill is checked, crashing the program if occurring.
- The `p->killed` flag is set, in effect killing the stack.
- Some poppers may be suspended, waiting for the stack to become non-empty (or to be killed!). We awake them all (with `broadcast_cond`).
- Then we wait for `nprocs` acknowledgements, suspending on the condition variable `p->wait` in a quite standard manner.

The pop function is responsible for making acknowledgements:

```

void *pop(stack_t *p) {
    void *r ;
    lock_mutex(p->lock) ;
    while (!p->killed && p->sp <= 0) wait_cond(p->is_empty, p->lock) ;
    if (p->sp > 0) {
        int was_full = p->sp >= p->sz ;
        p->sp-- ; r = p->t[p->sp] ;
        if (was_full) broadcast_cond(p->is_full) ;
    } else {
        assert (p->killed) ;
        p->seen++ ;
        signal_cond(p->wait) ;
        r = NULL ;
    }
    unlock_mutex(p->lock) ;
    return r ;
}

```

In effect, `pop` must take some action when the stack is non-empty or killed, or equivalently `pop` must block when the stack is not killed and empty — see the condition `(!p->killed && p->sp <= 0)` above.

If some action have to be taken, observe that retrieving an item from a non-empty stack has priority — see `if (p->sp) {...}` above. If the action is a kill acknowledgement, the thread that performed the kill is waiting on the condition variable `p->wait`, we awake it (by instruction `signal_cond (p->cond)`) *after* having acknowledged the kill (by instruction `p->seen++`).