Do all your work on these exam sheets.
(There is a total of 7 pages including this cover page, function signature page/work area)

This is a closed-book exam. No papers or electronics.

You have **one hour and fifteen minutes** to answer the questions.

Please write down your name and student id on the top of this page first before you start answering any question. *Answer all questions directly on the exam papers.* If you need additional sheets, please let us know.

**Partial credit is possible for an answer.** Please show your work steps and make your exam as readable as possible. *We must* be able to read your handwriting in order to be able to grade your exam. *Spend your time in accordance with the point allocations.*
Problem 1: Concepts (40 Points, Please be concise)

a) [4 points] A thread in a critical section cannot be context-switched. True or False?

b) [4 points] What is a race condition?

c) [6 points] Why is sleep() acceptable to use with a kernel thread implementation, but not within a user-space thread implementation? Hint: what happens when a process sleeps.

d) [10 points] List one advantage of shared-memory IPC and one advantage of message-passing IPC? Briefly sketch how you could implement message-passing IPC using shared-memory – a diagram is fine.
e) [5 points] Multicast is an IPC mechanism where a sender sends the same message to a group of M receivers. A sender can multicast to different overlapping groups, e.g. send message M1 to processes {A, B, C} and send M2 to processes {B, C, E}. Briefly explain how you could use message queues to implement multicast assuming all processes are on the same machine.

f) [6 points] List two main differences between Unix Pipes and Unix Message Queues.
Problem 2: Analyze (20 Pts)

Consider a server on a uniprocessor that serves a request by executing N sequential stages each taking time $T$ (where is $1 \ldots N$). Assume the stages can be fully overlapped (such that when one blocks another can run). i) What is the time taken to execute a single request? ii) what is the server throughput for assuming 1 or $N$ threads? (iii) there is a limit to the number of threads we should use, explain why.
Problem 3: Signals (20 Pts)

The SIGINT signal indicates that a ^C is sent to your process. The default action is to terminate the process. If your program has children, then they will become orphans. We will prevent this. Your main program will call create_children which will create N children as given to you, and you will modify the main program to prevent a ^C from terminating it if children are still running. If a ^C is hit while children are running, you should print: NOT YET: “k children still running” (where k is the number of children still running). ^C should work as usual after all children are finished.

Hint: setting the handler function in the call to sigaction to SIG_DFL re-establishes the default signal behavior. Do NOT worry about ANY errors.

#define N 10

void main () {

    // This function creates N children
    create_children (N);
}
Problem 4: Multithreaded program (25 Pts)

Write a simple multithreaded file-server in Posix. The main program receives a request for a file (by calling get_next_request) and then creates a thread (handle_request) to handle that request. Multiple requests MUST be handled concurrently. The thread checks if the request can be found in the cache (get_cache), if the file is not there (returns NULL), it goes to disk (get_disk). After it gets the file from disk (if not in the cache), it stores the file in the cache (put_cache). Be careful to prevent race conditions on cache access only. You may simply call the functions provided below as needed.

#include <pthread.h>
// global declarations here if needed

// assume these are callable by your program
void put_cache (file_t *); // put file into disk_cache
file_t *get_cache(request_t *); // lookup file in disk_cache, NULL if not there
file_t *get_disk(request_t *); // returns file from disk
void send_file (file_t *); // send file to client
request_t* get_next_request (); //returns next request

void handle_request (void *arg) {
    request_t *R;
    file_t *file; // the file that will be sent to the user
    R = (request_t *) arg; / now you can use the argument!
    // fill in code here

    free (arg);
    send_file (file); // sends file to client
    pthread_exit (NULL);
}

void main () {
    request_t *R; // more declarations below

    while (1) {
        R = get_next_request (); // request is allocated on the heap
        // fill in code here
    }
}
System call signatures (not all may be needed):

**Threads:**
- `pthread_mutex_init (pthread_mutex_t *, NULL);`
- `pthread_mutex_lock (pthread_mutex_t *);`
- `pthread_mutex_unlock (pthread_mutex_t *);`
- `pthread_create (pthread_t *, NULL, thread_func, void *);`
- `void *thread_func (void *);`
- `int pthread_join (pthread_t, NULL);`
- `int pthread_detach (pthread_t);`
- `pthread_exit (NULL);`
- `void thr_yield ();`

**Signals:**
- `int sigemptyset (sigset_t *);`
- `int sigfillset (sigset_t *);`
- `int sig{add/del}set (sigset_t *, int signo);`
- `int sigprocmask ({SIG_SETMASK or SIG_UNBLOCK}, const sigset_t *, NULL);`
- `int sigaction (int signo, const struct sigaction *, NULL);`
  `sigaction contains sa_mask and sa_handler fields`
- `void sig_handler (int signo);`
- `int pause ();`
- `int alarm (int);`

**Memory management:**
- `void *malloc (size_t nbytes);`
- `void free (void *ptr);`
- `void *memcpy (void *buf1, const void *buf2, size_t);`
size_t sizeof (<type> or <var>);