CS 162 Fall 2023 — Operating Systems and System Programming

Midterm 3

INSTRUCTIONS

Please do not open this exam until instructed to do so.

Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

Preliminaries

This is a proctored, closed-book exam. You are allowed 3 pages of notes (both sides). You may not use a calculator. You have 75 minutes to complete as much of the exam as possible. This exam is out of 75 points. Make sure to read all the questions first, as some are substantially more time-consuming.

If there is something about the questions you believe is open to interpretation, please ask us about it.

We will overlook minor syntax errors when grading coding questions. **There is a reference sheet at the end of the exam that you may find helpful.**

(a) Name

(b) Student ID

(c) Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use three 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

Write your full name below to acknowledge that you've read and agreed to this statement.

1. (14 points) True/False

Please explain your answer in TWO SENTENCES OR LESS. Longer explanations will GET NO CREDIT. Answers without an explanation or that just rephrase the question will GET NO CREDIT.

(a) **i.** **(2 pt)** Consider two adjacent tracks of a magnetic disk. Assume the outer track contains sectors 11-20, and the inner track contains sectors 21-30. Aligning the tracks such that sectors 20 and 21 are directly adjacent will minimize the time it takes to consecutively read blocks 11-30.

○ True.

[✓ ● False.]

ii. Explain.

Using track skewing can result in a more optimal consecutive access time. When sector 21 is offset from sector 20 such that the seek time is offset by rotation distance, contiguous access will be minimized.

(b) (2 points)

i. For infrequent, large data transfers, DMA lowers CPU utilization compared to polling.

[✓ ● True.]

○ False.

ii. Explain.

With DMA, the CPU offloads work to the DMA controller, so utilization drops compared to polling, where the CPU must burn cycles while waiting for more data to become available.

(c) **i.** **(2 pt)** Consider a file system which stores file metadata (access permissions) in the corresponding directory entry. Assume the entry for `/src/a.txt` does not provide the current user read/write permissions, but permits the current user to create arbitrary links. The current user could bypass the read/write permissions by creating a soft link to the file.

○ True.

[✓ False.]

ii. Explain.

Creating a *hard link* to the file would be sufficient to bypass the metadata. However, attempting to read the soft link would result in 1) Directory traversal to `/src/`, 2) Attempt to read `a.txt` which fails because of the metadata.

(d) **(2 points)**

i. FFS takes advantage of spatial locality on disk.

[✓ True.]

○ False.

ii. Explain.

FFS takes advantage of spatial locality by placing a directory and its files near each other.

(e) (2 points)

i. UDP has higher overhead than TCP.

○ True.

[✓ False.]

ii. Explain.

TCP needs to send additional packets for retries in order to ensure reliability, increasing its overhead.

(f) i. (2 pt) An RPC is no more expensive than a local procedure.

○ True.

[✓ False.]

ii. Explain.

RPCs do result in large performance overheads (particularly the serialization/deserialization process). This has led to extensive work and research trying to reduce this overhead and reduce this bottleneck in (distributed) systems.

(g) i. **(2 pt)** One advantage of taking a stateless approach in NFS is that crash recovery is much easier as compared to stateful protocols.

[✓ True.]

○ False.

ii. Explain.

Since you aren't maintaining state in NFS, when you are recovering from crashes, you do not have to worry about resolving partially written state or potential consistency issues. You can just continue sending RPCs as before.

2. (12 points) Multiple Select

(a) **(2 pt)** Select the following statements about memory and port-mapped I/O that are true.

- ■ Memory-mapped I/O simplifies CPU design.
- ■ Port-mapped I/O operates in a separate address space than main memory.
- ☐ The device controller incurs more complexity in port-mapped I/O.
- ☐ Memory-mapped I/O incurs higher memory overhead.
- ☐ None of the above.

A: True. Memory-mapped I/O does not necessitate the use of specialized instructions, thus simplifying CPU design.

B: True. One of the key differences between port-mapped I/O and memory-mapped I/O is that the devices work out of a different address space than main memory and are located in a specialized address location for handling the device.

C: False. Port-mapped I/O offsets more of the complexity of memory management and handling to the CPU than memory-mapped I/O does.

D: False. Memory overhead is not affected by either method of I/O.

(b) **(2 pt)** Select the following statements about disks that are true.

- ☐ Evenly sparing sectors across the disk reduces transfer time.
- ☐ When there is a bad sector, it is optimal to remap only that sector.
- ■ Sectors contain error-correcting codes to detect corruption of data.
- ■ Reading sequential sectors is faster than reading random sectors.
- ☐ None of the above.

A: False, spreading out spare sectors reduces seek/rotation time, not transfer time.

B: False, the idea of slip sparing is that we want to remap all sectors to keep them sequential for faster read/writes

C: True, see lecture 18.

D: True, sequential sectors reduce seek/rotation time since we are going directly to the next sector.

(c) **(2 pt)** Select the following statements about NTFS that are true.

- ■ NTFS provides support for hard links.
- ■ Each entry in the MFT will contain the human-readable name of the file along with the file number of its parent directory.
- ☐ NTFS is the default file system for both Windows and Linux systems.
- ■ The directory structure is implemented as a B-Tree.
- ☐ None of the above.

A: True. NTFS provides support for hard links, even if we support multiple file name attributes in the MFT.

B: True. Every entry in the MFT will include a file name attribute, which includes both the human-readable name of the file and the number of its parent directory.

C: False. NTFS is the default file system for Windows, while other OSs might be able to support NTFS device drivers, they will be read-only, and they won't be able to write to them.

D: True. The way NTFS creates the directory structure results in an implementation that uses a B-Tree.

(d) **(2 pt)** Select the following statements about the buffer cache that are true.

- ■ Prefetching subsequent disk blocks can improve disk access time.
- ■ Buffer cache can choose LRU as the main replacement policy.
- ☐ Because buffer cache writes back dirty blocks periodically (30 seconds in Linux), even for blocks used recently, we can never crash with dirty blocks in the cache.
- ☐ In a buffer cache with the Clock Algorithm for replacement, the replacement of a block is determined by considering both the access bit and the dirty bit associated with that block.
- ☐ None of the above.

A: True. Typical file access patterns read sequential blocks.

B: True. Buffer cache can afford the overhead of full LRU implementation.

C: False. Even with periodically writing back dirty blocks, we can still crash with dirty blocks in the cache.

D: False. The replacement of a block is only determined by considering the access bit, while the dirty bit is for determining whether we have to persist changes to the block to disk before evicting.

(e) **(2 pt)** Select the following statements about NFS that are true.

- ■ NFS implements weak consistency.
- ☐ NFS immediately broadcasts file changes to all clients who have the file open.
- ■ The cache becomes a bottleneck when a large number of clients connect to the same NFS volume.
- ☐ There is always only ever one version of a file in NFS.
- ☐ None of the above.

A: True. Any other viewer will eventually have their view of the file updated.

B: False. It waits for some timeout.

C: True. NFS does not scale to a large number of clients, as the server becomes a bottleneck due to polling traffic.

D: False. There can be multiple for a period of time and one is chosen arbitrarily.

(f) (2 pt) Select the following statements about the FAT file system that are true.

- ☐ Because linked list insertion and removal are O(1) operations, FAT filesystem supports very fast random access compared to inode based file systems.
- ☐ In FAT file system, if we know the file number is 31, we then can say for sure disk block number 31 is one of the blocks for that file but not necessarily the first block for that file.
- ☐ FAT file system supports hard links.
- ☐ In the traditional FAT file system, the File Allocation Table is stored in memory and maintained by the OS, while only the disk blocks are stored on disk.
- ■ None of the above.

A: False. FAT file system is not efficient for random access by any means because it's fundamentally a linked list 1-1 with blocks, and thus accessing a specific offset in a file requires traversing through the linked list. The "layout" fragmentation in FAT also exacerbates the situation.

B: False. In FAT file system, the file number is the index of the root of block list for the file.

C: False. FAT cannot support hard links as all metadata for a file is stored in the directory entry.

D: False. The FAT is stored on disk.

3. (24 points) Short Answer

(a) **(4 pt)** Suppose we have a hard drive with the following specifications:

- An average seek time of 6 ms.
- A rotational speed of 5000 revolutions per minute
- A controller that can transfer data at a rate of 80 MB/s

What will be the access time of a hard drive with the above specifications, when it tries to read two consecutive 4 KB sectors from a random location on disk? Ignore the queueing delay. *Show your work.*

Rotational speed in ms = 60000 (ms/min) / 5000 (rev/min) = 12 ms
Average rotation time = 0.5 * 12 ms = 6 ms
Transfer rate = 4 KB / 80 MB/s = 0.05 ms for 1 sector
Total time to read both sections = Time to read 1 random sector + Time to read a consecutive sector (sector on the same track)
Time to read 1 random sector = Seek + avg rotation time + transfer rate = 6 ms + 6 ms + 0.05 ms = 12.05 ms
Time to read a consecutive sector (one in the same track) = transfer rate = 0.05 ms
Total time to read both sections = 12.05 ms + 0.05 ms = 12.1 ms

(b) **(4 pt)** Name one issue with SCAN that Circular Scan (CSCAN) resolves.

SCAN is biased towards pages in the middle, whereas CSCAN is not. Alternatively, using CSCAN means that you don't have to decelerate the disk in order to reverse the direction.

(c) (4 pt) How does using the flash translation layer allow flash memory to last longer?

Wear leveling: physical pages that have been used less often can be prioritized in being mapped, to get an even amount of wear across all the physical pages.

(d) (4 pt) Suppose I want to write a PintOS user program which reads the first byte in the file `/src/main/test.c`. How many disk blocks will be read by this request? *Show your work.*

You may assume the following:

- The inode array is *not* resident in memory, but the root directory has its inode saved at a known location on disk.
- The entries for `src`, `main`, and `test.c` are in the first block of the relevant directories.

We need to access: 1) The first block of the root directory, 2) The first block of the src directory, 3) The first block of the main directory, and 4) The first block of test.c.
All accesses simply require reading the first data block of the corresponding file/directory. This involves 1) reading the inode, 2) reading the data block pointed to by the first direct pointer.
Thus, # of blocks read = 4*2 = 8

(e) **(4 pt)** In a distributed system, nodes in the system might be running different programs written in different languages running on different operating systems. How do RPCs allow for communication in spite of these differences?

RPCs convert everything to/from some canonical form which describes how to encode/decode entries. Thus, the only thing that needs to be shared is this canonical representation.

(f) **(4 pt)** Explain how the `map` and `reduce` functions work in a MapReduce program.

The *map* function takes in an input pair (key/value) and produces a set of intermediate key/value pairs. Then, all of the intermediate values associated with the same intermediate key are grouped together through the MapReduce library. The *reduce* function takes in an intermediate key and a set of values for that key, and then it merges all of them together to form a possibly smaller set of values.

4. (12 points) Two-Phase Commit and Extensions

In lecture, you learned about the *two-phase commit* (2PC) algorithm. Recall that the algorithm proceeds as follows:

Prepare:

P1. The coordinator logs a message with the transaction ID to indicate that a transaction has started and sends out a `VOTE-REQ` message to all workers.

P2. Upon receiving a `VOTE-REQ` message, each worker logs their response and sends a `VOTE-COMMIT` or a `VOTE-ABORT` to the coordinator.

Commit:

C1. The coordinator waits for all workers to send either a `VOTE-COMMIT` or at least one worker to send a `VOTE-ABORT`. If all workers voted to commit, the coordinator logs the outcome and sends out a `GLOBAL-COMMIT` to all workers. Else, the coordinator logs the outcome and sends out a `GLOBAL-ABORT` to all workers.

C2. Upon receiving a `GLOBAL-COMMIT`, each worker logs the result and commits the transaction. Upon receiving a `GLOBAL-ABORT`, each worker logs the result and aborts the transaction.

We will consider different failure cases, optimizations, and extensions to the 2PC algorithm.

(a) (2 points)

i. Suppose that a worker sends `VOTE-COMMIT` to the coordinator. Then, that worker must wait until the coordinator responds with the `GLOBAL-COMMIT` message before committing, in order to ensure correctness of the algorithm.

● True.

○ False.

ii. Explain.

It is possible for some other worker to send a `VOTE-ABORT`, in which case the current worker should not complete their part of the transaction.

(b) (2 points)

i. Suppose that a worker sends `VOTE-ABORT` to the coordinator. Then, that worker must wait until the coordinator responds with the `GLOBAL-ABORT` message before aborting, in order to ensure correctness of the algorithm.

○ True.

● False.

ii. Explain.

Note that if a single worker votes to abort, then it is guaranteed that the transaction will be aborted and thus it can start releasing resources or doing any necessary cleanup without waiting for confirmation from the coordinator.

(c) (2 pt) Suppose that we additionally stipulate that after each worker completes step C2, it should send an `ACK` message to the coordinator. How can the coordinator use these additional messages to reduce the amount of disk space used by the log?

Once the coordinator receives all `N` ACKs, then it knows it is able to clear all the log messages related to this transaction from its persistent storage.

(d) (3 pt) Now suppose that we use the *presumed abort* optimization, where transactions missing from the coordinator's log are presumed to have been aborted. (Namely, if a worker queries the coordinator for the status of a transaction whose ID does not appear in the coordinator's logs, then the coordinator responds with a `GLOBAL-ABORT`). How can the coordinator use this to reduce the amount of disk space used up by the log?

Upon seeing a `VOTE-ABORT`, the coordinator can immediately delete any logs related to that transaction, and does not need to log that it will submit a `GLOBAL-ABORT`.

(e) (3 pt) Explain why in the presumed abort optimization (defined above), the coordinator does not need to flush the initial log message (indicating that voting has started) to disk and may instead keep the dirty block in main memory.

If the coordinator crashes and the message indicating that voting has started is lost, then the coordinator will indicate to the workers that the transaction should be aborted.

5. (13 points) Journaled FFS

Carlos' project group is trying to implement some functions for an inode-based journaled file system on a 32-bit machine.

The group is given the following functions and types to use. Assume that all operations are synchronized properly.

```
typedef uint32_t block_t;

typedef enum {
  FILE,
  DIR
} filetype_t;

typedef struct {
  filetype_t filetype;  // Type of the file
  size_t filesize;      // Size of the file
  block_t dp;           // Direct pointer
  uint8_t unused[4084]; // Padding to match block size
} inode_t;

// Returns the next free index in the bitmap,
// If not successful,  returns -1
int next_bitmap_free();
// Returns true if the index in the bitmap is free
bool is_bitmap_free(uint8_t idx);
// Sets the index in the bitmap to is_free
void mark_bitmap_free(uint8_t idx, bool is_free);

// Returns the next empty inumber
// If not successful, returns -1
int next_empty_inode();
// Returns the inode at inumber
// If not successful, returns NULL
inode_t* get_inode(int inumber);
// Sets the inumber to point to the inode at block address
void set_inode(int inumber, block_t block);

// Returns the block at block_num
// If not successful, returns NULL
block_t get_block(int block_num);
// Returns a pointer to the block's data in memory
uint8_t* block_read(block_t block);
// Writes the full block-sized data to block
void block_write(block_t block, uint8_t* data);

// Returns the inumber from a directory inode given filename and filename size
// If not successful, return -1
int get_dir_entry(inode_t* inode, uint8_t* filename, size_t filename_size);
// Adds a directory to the given directory inode pointing the inode to the file's inumber
void add_dir_entry(inode_t* inode, uint8_t* filename, size_t filename_size, int inumber);
// Returns true if another directory entry can be added to the inode
// If not successful, returns false
bool can_add_dir_entry(inode_t* inode);
```

```
// Deletes all entries within a list
void list_clear(struct list* list);

typedef enum {
  START_COMMIT,
  OPERATION,
  END_COMMIT,
} entrytype_t;

typedef struct {
  struct list_elem journal_e;
  struct list_elem pending_e;
  entrytype_t type;
} journal_entry_t;

static struct list elem journal;

void init_journal();    // Initializes the journal
void start_commit();    // Adds a start_commit entry to the journal
void end_commit();      // Adds an end_commit entry to the journal

// Logs an operation entry with X arguments to the journal
#define log_op0(void* func)
#define log_op1(void* func, void* arg1)
#define log_op2(void* func, void* arg1, void* arg2)
#define log_op3(void* func, void* arg1, void* arg2, void* arg3)
#define log_op4(void* func, void* arg1, void* arg2, void* arg3, void* arg4)

// Performs pending operations onto disk
void journal_perform_ops(struct list* pending);
// Frees and clears journal entries starting from tail to head (inclusive).
// If empty, nothing happens.
void journal_free(struct list_elem* tail, struct list_elem* head);
```

(a) (7 points)

Carlos wants to implement a function that creates an empty file in the root inode using the journal. Instead of storing all the changes necessary to create the file in the journal, he decides to only journal metadata updates. The function should return `true` if the file can be successfully created and `false` otherwise.

Assume that `journal_touch_rootfile` will only be called by one thread at a time.

Help Carlos complete his implementation:

```
bool journal_touch_rootfile(uint8_t* filename, size_t filename_size) {
  inode_t* root_inode = get_inode(2);
  int free_block_idx = next_bitmap_free();

  _____________[Ax4]_____________

  inode_t inode;

  _____________[Bx4]_____________

  start_commit();

  _____________[C]_____________
  int inumber = next_empty_inode();

  _____________[Dx8]_____________

  end_commit();
  return true;
}
```

i. (1 pt) [A] (max 4 lines)

```
if (free_block_idx == -1) {
  return false;
}
block_t block = get_block(free_block_idx);
```

ii. (1 pt) [B] (max 4 lines)

```
inode.filetype = FILE;
inode.filesize = 0;
inode.dp = NULL;

block_write(block, &inode);
```

iii. (1 pt) [C] (max 1 line)

```
log_op2(mark_bitmap_free, free_block_idx, false);
```

iv. (4 pt) [D] (max 8 lines)

```
if (inumber == -1) {
  return false;
}
log_op2(set_inode, inumber, block);

if (!can_add_dir_entry(root_inode)) {
  return false;
}
log_op4(add_dir_entry, root_inode, filename, filename_size, inumber);
```

(b) (6 points)

Steven swaps with Carlos to work on a function that flushes the journal.

Assume that `journal_flush` will only be called by one thread at a time.

Help Steven complete his implementation:

```
void journal_flush() {
  struct list pending;
  list_init(&pending);
  struct list_elem* tail = list_begin(&journal);
  struct list_elem* e = tail;
  while(e != list_end(&journal)) {

    journal_entry_t* j = list_entry(e, journal_entry_t, journal_e);

    switch (j->type) {
      case START_COMMIT:
        _____________[A]_____________
        break;
      case OPERATION:
        _____________[B]_____________
        e = list_next(e);
        break;
      case END_COMMIT:
        struct list_elem* head = e;
        e = list_next(e);

        _____________[Cx4]_____________
        break;
    }
  }

  if (_____________[D]_____________) {
    _____________[E]_____________
  }
}
```

i. (0.5 pt) [A] (max 1 line)

```
e = list_next(e);
```

ii. **(1 pt)** [B] (max 1 line)

```
list_push_back(&pending, &j->pending_e);
```

iii. **(3 pt)** [C] (max 4 lines)

```
journal_perform_ops(&pending);
list_clear(&pending);
journal_free(tail, head);
tail = e;
```

iv. **(0.5 pt)** [D] (max 1 expression)

```
!list_empty(&journal)
```

v. **(1 pt)** [E] (max 1 line)

```
journal_free(list_begin(&journal), list_end(&journal));
```

6. (1 points) Extra Credit

(a) (1 pt) Give the full names of two out of our five tutors. Spelling counts!

Ashwin Chugh, Edrees Saied, Junhee Park, Steven Cai, Vivek Verma

7. Empty Page

8. Empty Page

9. Reference Functions

```
/* Process */
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/* pthreads */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
      void* (*start_routine (void *)), void *arg);
int pthread_join(pthread_t thread, void **retval);
void pthread_exit(void *retval);

/* pthread Semaphore interface */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem); /* The p() or down() operation */
int sem_post(sem_t *sem); /* The v() or up() operation */

/* pthread Lock/mutex operations */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* pthread Condition Variable */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/* PintOS locks */
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* PintOS semaphore interface */
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

/* PintOS condition variables */
void cond_init(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
void cond_broadcast(struct condition *cond, struct lock *lock);

/* PintOS Readers/Writers Locks */
void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);
```

```
/* PintOS List */
void list_init(struct list *list);
struct list_elem *list_head(struct list *list);
struct list_elem *list_tail(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_insert_ordered(struct list* list, struct list_elem* elem, list_less_func* less, void* aux);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem* list_pop_front(struct list* list);
struct list_elem* list_pop_back(struct list* list);

/* Strings */
char *strcpy(char *dest, char *src);
char *strdup(char *src);

/* Interrupt enable/disable */
enum intr_level {};
enum intr_level intr_get_level(void)
enum intr_level intr_set_level(enum intr_level level)
enum intr_level intr_enable(void)
enum intr_level intr_disable(void)

/* High-Level IO */
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
fprintf(FILE * restrict stream, const char * restrict format, ...);

/* Low-Level IO */
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

/* Socket */
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

```
/* Memory */
void *memcpy(void *dest, const void * src, size_t n)
```

10. Powers of 2

$2^5 = 32$

$2^6 = 64$

$2^8 = 256$

$2^{10} = 1024$

$2^{20} = 1048576$