Computer Science 162
David E. Culler
University of California, Berkeley
Practice Final Exam
December XX, 2019

Name	
TA Name and Section Time	

This is **practice** for a closed book exam with two 2-sided pages of notes permitted. It is intended to be a 100 minute exam. You have 170 minutes to complete it. The number at the beginning of each question indicates the points for that question. Write all of your answers directly on this exam. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, please raise your hand to request clarification. When told to open the exam, put your Student ID on every page and check that you have them all. The final pages are for reference. [* You can expect the topical coverage, length, and format to be similar. The places where we ask you to demonstrate working knowledge through innovation versus recall may differ, but the mix will be similar. *]

By my signature below, I swear that this exam is my own work. I have not obtained answers or partial answers from anyone. Furthermore, if I am taking the exam early, I promise not to discuss it with anyone prior to completion of the regular exam, and otherwise I have not discussed it with anyone who took the early alternate exam.

X ______________________________________

Grade Table (for instructor use only)

Question:	1	2	3	4	5	6	7	8	Total
Points:	1	20	20	10	10	20	20	0	101
Score:									

Student ID: ____________________

1. (1 point) **Write your Student ID on every page of the exam.**

2. (20 points) **Operating System Concepts: True/False**

   Choose True or False for 20 of the 25 questions below. Clearly cross out the five that should not be graded. You do not need to justify your answers.

   (a) (1 point) If a process writes past the end of an open file the `write` system call will succeed and extend the file with no reported error.

   ◯ True ◯ False

   (b) (1 point) If a fair scheduler, like Linux CFS, strives to meet a responsiveness target T by giving each task a quanta of T divided by the number of tasks, the overhead may get excessive with a large number of tasks.

   ◯ True ◯ False

   (c) (1 point) Changing a cache from direct-mapped to fully-associative will *never* decrease its hit rate, under any size, workload, and eviction policy.

   ◯ True ◯ False

   (d) (1 point) A single memory access may result in a TLB miss and a cache hit.

   ◯ True ◯ False

   (e) (1 point) As a system's workload is increased, below the half-power point, response time typically increases due to queuing delays.

   ◯ True ◯ False

   (f) (1 point) LRU is an appropriate policy to evict page frames for demand paging.

   ◯ True ◯ False

   (g) (1 point) In the x86 architecture, the MMU clears the use bit of page table entries that have not been accessed in a long time.

   ◯ True ◯ False

   (h) (1 point) A device driver may interact with I/O devices by reading and writing to memory addresses reserved for memory-mapped I/O.

   ◯ True ◯ False

   (i) (1 point) In a typical file system, large files account for most in-use disk space.

   ◯ True ◯ False

   (j) (1 point) The compiled kernel code contains the file system root (`/`) at a known inumber.

   ◯ True ◯ False

   (k) (1 point) In the FAT file system, file attributes are kept in the directory and not in the file or its header.

   ◯ True ◯ False

   (l) (1 point) In a Unix FFS-style inode-based file system, there is no limit on how large a file can be, except for the size of the underlying storage device.

   ◯ True ◯ False

   (m) (1 point) In the Windows NTFS file system, unlike FFS, sequential allocation on disk is enabled by variable extents.

   ◯ True ◯ False

   (n) (1 point) Low-level file API (`read`, `write`) is accelerated by the kernel's buffer cache.

   ◯ True ◯ False

(o) (1 point) The Transmission Control Protocol (TCP) allows each endpoint of a connection to always have multiple outstanding unACKed packets at a time.

◯ True ◯ False

(p) (1 point) In the Network File System (NFS), a user's modifications to a file *might* become visible to other users in the system as soon as the `write` operation completes.

◯ True ◯ False

(q) (1 point) A group of machines can use the Two-Phase Commit (2PC) protocol to guarantee that they will perform a task simultaneously.

◯ True ◯ False

(r) (1 point) For consistent hashing each node chooses a random value in the range of a hash function on the key and shares it with the other nodes to partition the key space into equal-sized blocks.

◯ True ◯ False

(s) (1 point) Storing hashes of passwords, even without adding a salt, is not prone to dictionary attacks if a sufficiently robust hash function is used and users are encouraged to change their passwords often.

◯ True ◯ False

(t) (1 point) For virtual machines, the shadow page table is the bottom half of the two layer address translation, and maps guest physical frames to host physical frames.

◯ True ◯ False

(u) Containers handle package dependences for collections of processes, whereas Kubernetes introduces Pods as a mechanism for co-locating a related set of containers on a set of (virtualized) reasources.

◯ True ◯ False

(v) (1 point) Containers provide the illusion of a dedicated file system for a well-defined collection of executables and libraries and the processes that instantiate them.

◯ True ◯ False

(w) (1 point) The socket abstraction is dictated by the transport layer of the network stack to allow a client and a server to communicate over HTTP.

◯ True ◯ False

(x) (1 point) The total number of physical page frames is always greater than or equal to the total number of resident virtual pages (assuming the size of physical and virtual pages are the same).

◯ True ◯ False

(y) (1 point) A key/value store can provide some, but not all, aspects of a file system by structuring the keys as pathnames, (e.g., `/usr/bin/ps`) and the values as file contents. `GET` acts as reading the whole file.

◯ True ◯ False

3. (20 points) **Operating System Concepts: Short Answer**

   Do 20 of the following 25 points and cross out the ones that are not to be graded.

   Fill in the blanks correctly for the questions below. You do not need to provide justifications.

   (a) (1 point) A process may request the operating system to perform a privileged action on its behalf by issuing a/an ____________________.

   (b) (1 point) A/An ____________________ is a programming pattern in which a lock is used with one or more condition variables.

   (c) (1 point) The subset of its virtual address space that a process is actively using over a given time interval is called a/an ____________________.

   (d) (1 point) Running a virtual machine requires support in the host operating system in the form of a kernel module or driver called a/an ____________________.

   (e) (1 point) An operation is said to be ____________________ if performing it multiple times in a row has the same effect as performing it exactly once.

   (f) (1 point) A group of n machines can maintain safety and liveness in the face of fail-stop failures as long as at most ____________________ node(s) are faulty, whereas with Byzantine failures a __________ fraction of nodes must be non-faulty.

   (g) (2 points) What is the difference between a directory and an index?

   (h) (2 points) Why can't distributed file systems, like AFS and NFS, keep client-side caches consistent using techniques from multiprocessor cache coherence (e.g., snoopy caches on a bus)? What do they do instead?

   (i) (2 points) The concept of a *namespace* arises in a variety of places in operating systems. Give two examples.

(j) (2 points) Google's Internet search service receives approximately 64000 search queries per second, on average. Let's assume that the average latency for Google to process each query is approximately 400 ms. How many search queries, on average, are being processed by Google's system at any particular instant? Show your work.

(k) (2 points) What is the difference between a virtual machine and a container?

(l) (2 points) Assume that a service is able to perform an operation in 100ms. If it has no internal parallelism, what is its rate of performing operation? If the service is 50% utilized and it takes 50ms to get to and from the services, how long should you expect your request to take.

(m) (2 points) For a server listening on a TCP socket, what must take place for it to complete the `accept` call? What does `accept` return?

(n) (3 points) With a two level page table on a 32-bit machine with 4kb pages and 4-byte page table entries, list every page table and memory access involved in fetching and executing an instruction that loads a value into a register assuming no page faults occur. How many TLB accesses occur during this instruction? [* You can expect a more detailed work-em-out variant of this question on the exam. Try to give a complete answer here. *]

(o) (2 points) Consider an IPC framework used to allow one process to provide machine learning services to other processes. The client has the following object that it wants to pass to the server for it to make a prediction.

```
struct predictor_obj {
    char *str;
    int num_weights;
    int weights[];
};
```

Explain why it cannot pass this object directly. What does it need to do instead?

4. (10 points) **Pintos**

   Choose True or False for the questions below according to Pintos, the operating system you used in the projects for this class. You do not need to justify your answers.

   (a) (1 point) Accesses to kernel memory (`PHYS_BASE` and above) in the system call handler and interrupt handlers operate directly on physical addresses, bypassing all page tables.

   ◯ True ◯ False

   (b) (1 point) It is inherently unsafe to call `sema_down` on a zero-value semaphore in external interrupt context.

   ◯ True ◯ False

   (c) (1 point) It is inherently unsafe to call `sema_down` on a zero-value semaphore in the page fault handler.

   ◯ True ◯ False

   (d) (1 point) Interrupts are disabled while Pintos switches threads (i.e., while the `switch_threads` function executes).

   ◯ True ◯ False

   (e) (1 point) On a successful call to `pagedir_set_page`, Pintos flushes the TLB.

   ◯ True ◯ False

   (f) (1 point) When a page is allocated using `palloc_get_page`, Pintos must modify the page table of every user process so that the page is accessible in kernel mode.

   ◯ True ◯ False

   (g) (1 point) The set of free blocks is stored on disk as a linked list.

   ◯ True ◯ False

   Answer the question below on the provided line according to Pintos, the operating system you used in the projects for this class. You do not need to justify your answers.

   (h) (1 point) What scheduling algorithm did you implement in Pintos in Task 2 of Project 2?

   (h) ____________________

   Answer the questions below in the box provided according to Pintos, the operating system you used in the projects for this class.

   (i) (2 points) Suppose the scheduling timer expires while interrupts are disabled. When will the `timer_interrupt` function (i.e., the external interrupt handler) be invoked next?

5. (10 points) **Inode Design**

   Alice, Benjamin, Catherine, and David are in a group for their CS 162 project. For Project 3, they choose an inode design with the following property: **the first 162 bytes of a file are stored directly within the inode.**

   (a) (1 point) What does the "i" in "inode" stand for?

   (a) ______________________

   (b) (2 points) List one advantage of storing some file data directly in the inode.

   (c) (2 points) List one disadvantage of storing some file data directly in the inode.

   (d) (3 points) Write out a definition of `struct inode_disk` with this property. Be sure to include 10 direct pointers, 1 singly indirect pointer, and 1 doubly indirect pointer, in addition to the first 162 bytes. Your inode structure must support directories, as required in Task 3 of the project. Do *not* assume useful data is stored in the `magic` or `unused` fields.

```
struct inode_disk {

    ______________________________

    ______________________________

    ______________________________

    ______________________________

    ______________________________

    ______________________________

    unsigned magic;
    uint8_t unused[______________________________];
};
```

   (e) (2 points) If the group decides to implement hard links in their file system, how would their `struct inode_disk` have to change?

6. (20 points) **Footex**

   Assume that shared pages are implemented in Pintos—that is, a physical page may be mapped into the address spaces of multiple processes. This provides the opportunity to synchronize two processes in *userspace* by placing a lock on a shared page.

   In class, we studied the `futex` ("fast userspace mutex") system call. In this problem, you will implement `footex`, a simplified version of `futex`, for Pintos. It consists of two system calls, which each return `true` on success and `false` on failure. [* You can expect a more familiar variant of this question. *]

```
/* Checks if *ADDR == VAL and atomically blocks the calling process if so. */
bool footex_wait(int* addr, int val);
/* Wakes up NUM threads waiting on the provided address ADDR. */
bool footex_wake(int* addr, int num);
```

   You may call the following function from userspace in Pintos:

```
/* Atomically sets *ADDR = VAL and returns the old value of *ADDR. */
int test_and_set(int* addr, int val);
```

   As a warm-up, let's do a programming exercise with `footex` in a Pintos **user program**.

   (a) (4 points) Implement a userspace lock in Pintos using `footex_wait`, `footex_wake`, and `test_and_set`. This code should run in a **user program** (i.e., not in the Pintos kernel). Your implementation should be syscall-free if the lock is uncontended. You may assume that `footex_wait` and `footex_wait` always return `true` for this question.

```
struct user_lock {
    int state;

    _______________________________________________
};
void user_lock_init(struct user_lock* lock) {
    lock->state = 0; // unlocked state

    _______________________________________________
}
void user_lock_acquire(struct user_lock* lock) {

    _______________________________________________

    _______________________________________________

    _______________________________________________

    _______________________________________________

    _______________________________________________
}
```

```
void user_lock_release(struct user_lock* lock) {
    ASSERT(lock->state == 1); // locked state

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________
}
```

Now, we will implement the `footex_wait` and `footex_wake` system calls in the Pintos **kernel**.

(b) (4 points) Extend the system call handler to support the two new system calls. You may call the following two functions:

```
void validate_user_buffer(void* pointer, size_t length);
void validate_user_string(const char* string);
```

These functions check if the provided buffer (or string) exists entirely within valid user-accessible memory. If not, they terminate the calling process with exit code `-1`.

```
static void syscall_handler (struct intr_frame* f) {
    uint32_t* args = ((uint32_t*) f->esp);
    validate_user_buffer(args, sizeof(uint32_t));

    switch (args[0]) {
        /* Pre-existing cases (not shown) */
        ...

    case SYS_FOOTEX_WAIT:

        ___________________________________________________________

        ___________________________________________________________

        ___________________________________________________________

        f->eax = (uint32_t) syscall_footex_wait((int*) args[1], (int) args[2]);
        break;

    case SYS_FOOTEX_WAKE:

        ___________________________________________________________

        ___________________________________________________________

        ___________________________________________________________

        f->eax = (uint32_t) syscall_footex_wake((int*) args[1], (int) args[2]);
        break;
```

```
            /* Additional pre-existing cases (not shown) */
            ...
        }
    }
```

(c) (1 point) Suppose process P calls `footex_wait` on the address `addr1` and is put to sleep as a result. Then process Q calls `footex_wake` on the address `addr2`. How must `addr1` and `addr2` be related, for Q's `footex_wake` to wake up P? (Hint: consider that a single shared page may be mapped at different virtual addresses in different address spaces.)

(d) (1 point) Keep track of the threads waiting on a particular address using `struct footex`.

```
struct footex {
    int* addr;
    struct list_elem elem;

    ______________________________________________

    ______________________________________________

    ______________________________________________
};
```

(e) (1 point) Determine what additional global variables must be added to `syscall.c`, and extend `syscall_init` to do any necessary initialization work.

```
/* Add additional global variables on the lines below. */

______________________________________________

______________________________________________

______________________________________________

void syscall_init(void) {
    intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");

    /* Perform any necessary initialization work. */

    ______________________________________________

    ______________________________________________

    ______________________________________________
}
```

Student ID: ______________________________

(f) (2 points) Complete the `footex_get` helper function below. Its intended behavior is to find the `struct footex` for the given address. If none exists and the `create` flag is `true`, then one is created. The provided `addr` is the virtual address in the calling process' address space.

```
static struct footex* footex_get(int* addr, bool create) {
    struct thread* t = thread_current();
    addr = ____________________________________________________________;
    for (struct list_elem* e = list_begin(&footex_list);
                       e != list_end(&footex_list); e = list_next(e)) {
        struct footex* curr = list_entry(e, struct footex, elem);
        if (curr->addr == addr) {
            return curr;
        }
    }
    struct footex* f = NULL;
    if (create) {
        f = malloc(sizeof(struct footex));
        if (f != NULL) {
            f->addr = addr;
            list_push_back(&footex_list, &f->elem);

            ____________________________________________________________

            ____________________________________________________________

            ____________________________________________________________
        }
    }
    return f;
}
```

Question continues on the next page ⟶

(g) (4 points) Implement the system call handlers for `footex_wait`.

```
bool syscall_footex_wait(int* addr, int val) {

    ________________________________________

    ________________________________________

    if (*addr == val) {
        struct footex* f = footex_get(addr, true);
        if (f == NULL) {

            ________________________________________

            ________________________________________

            ________________________________________

        }

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

        ________________________________________

    }

    ________________________________________

    ________________________________________

    return true;
}
```

Question continues on the next page ⟶

(h) (4 points) Implement the system call handlers for `footex_wake`.

```
bool syscall_footex_wake(int* addr, int num) {

    ________________________________________________

    ________________________________________________

    struct footex* f = footex_get(addr, false);
    if (f == NULL) {

        ________________________________________________

        ________________________________________________

        ________________________________________________

    }
    for (int i = 0; i < num; i++) {

        ________________________________________________

        ________________________________________________

        ________________________________________________

    }

    ________________________________________________

    ________________________________________________

    return true;
}
```

7. (20 points) **Journaling**

You are writing code late into the night to finish Project 3 for 162 when your laptop's battery dies. You plug in and restart your computer, and the operating system consults the file system's journal to determine how to recover from the unexpected shutdown. Fortunately, you're running a journaling file system that you wrote so there is a log of file system operations in persistent storage. The running state of the kernel is lost on reboot (of course) so a consistent file system state must be reestablished from the log on recovery. Its `head` pointer gives the next available entry and `tail` is the last completed operation. The contents of the journal are as follows. All prior log entries have been freed. The directory for `/home/oski` has been set to include an entry with name `filesys.c` and inode number 40.

`head`	`tail`
117	112

Entry #	**Transaction ID**	**Operation**
101	10	BEGIN
102	10	Initialize inode 18
103	10	Add a new entry to the directory `/home/oski` with name `inode.c` and inode number 18.
104	11	BEGIN
105	11	Mark blocks 4, 8, and 15 as *allocated* in the free map.
106	11	Write `<data buf>` to blocks 4, 8, and 15
107	10	COMMIT
108	11	Update inode 18 and set its first three direct pointers to refer to blocks 4, 8, and 15.
109	12	BEGIN
110	12	Mark blocks 16, 23, 42, and 108 as *allocated* in the free map.
111	11	COMMIT
112	12	Update inode 40 and set its first four direct pointers to refer to blocks 16, 23, 42, and 108.
113	13	BEGIN
114	13	Mark block 108 as *free* in the free map.
115	12	COMMIT
116	13	Clear the direct pointer to block 108 in inode 40.

(a) (2 points) Knowing that file system operations take a long time, in your implementation you decided to use threads to do all the updates to the disk for a transaction concurrently. What would the implementation of the COMMIT operation need to do?

(b) (4 points) What is the status of the file `/home/oski/inode.c` *on disk* and what happens to the log on recovery?

1. Blocks 4, 8, and 15 allocated.
   - ◯ Completed, free log entry
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard
2. Direct pointers in inode 18 are set to blocks 4, 8, and 15. A directory entry for `inode.c` added to `/home/oski` and refers to inode 18.
   - ◯ Completed, free log entry
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard

Explain your answers.

Question continues on the next page ⟶

(c) (7 points) What is the status of the file `/home/oski/filesys.c` *on disk*, what happens to the log on recovery, and what is the status of the file after recovery?

1. Blocks 16, 23 and 42 allocated
   - ◯ Completed and free log entry
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard
2. inode 40 initialized, its first four direct pointers set
   - ◯ Completed and log entry can be freed
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard
3. Commit Transaction 12.
   - ◯ Completed and log entry can be freed
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard
4. Block 108 is marked as free
   - ◯ Completed and log entry can be freed
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard

5. Direct pointer to block 108 cleared in inode 40
   - ◯ Completed and log entry can be freed
   - ◯ Completed but log entry cannot be freed
   - ◯ Maybe completed, could be replayed
   - ◯ Maybe completed, must be replayed
   - ◯ Not complete, replay
   - ◯ Maybe completed, discard
   - ◯ Not completed, discard
   - ◯ Completed, discard

Explain your answers.

(d) (1 point) How would your answers change if item 115 were not present in the log?

(e) (1 point) Instead assume there as an additional entry in the log, as below. What does recovery do?

Entry #	**Transaction ID**	**Operation**
117	13	COMMIT

(f) (5 points) Logs to provide proper recovery show up in many places, besides file systems. For each of the following, describe the proper action for a worker to take in a two-phase commit protocol.

A GLOBAL-COMMIT Message is received when log contains:
Transaction ID: 17, State: Ready, Last Sent Msg: VOTE-COMMIT. What action do you take?

You have just rebooted and are in recovery. The log contains:
Transaction ID: 93, State: Init, Last Sent Msg: <na>.
What action do you take?

You have just rebooted and are in recovery. The log contains:
Transaction ID: 117, State: Ready, Last Sent Msg: VOTE-COMMIT.
What action do you take?

8. (0 points) **Optional Questions**

   (a) (0 points) Having finished the exam, how do you feel about it? Check **all** that apply:

   ☐ 🙂 ☐ ☹️ ☐ 😐 ☐ 😕 ☐ 😑 ☐ 😵 ☐ 😎 ☐ 🤕

   ☐ 😴 ☐ 🐱 ☐ Other (please draw):

   (b) (0 points) If there's anything you'd like to tell the course staff (e.g., feedback about the class or exam, suspicious activity during the exam, new logo suggestions, etc.) you can write it on this page.

```
/******************************** Threads ********************************/
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);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
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);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
/******************************* Processes *******************************/
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[]);
void exit(int status);
/***************************** High-Level I/O *****************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
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);
int fclose(FILE *stream);
/***************************** Low-Level I/O ******************************/
int open(const char *pathname, int flags); (O_APPEND|O_CREAT|O_TMPFILE|O_TRUNC)
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
/***************************** Pintos Lists ******************************/
void list_init(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);
void list_insert(struct list_elem *before, struct list_elem *elem);
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_remove(struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_pop_back(struct list *list);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
```

```
/************************** Pintos Threads ****************************/
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
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);
enum intr_level intr_get_level(void);
enum intr_level intr_set_level(enum intr_level);
enum intr_level intr_enable(void);
enum intr_level intr_disable(void);
bool intr_context(void);
void intr_yield_on_return(void);
tid_t thread_create(const char *name, int priority, void (*fn)(void *), void *aux);
void thread_block(void);
void thread_unblock(struct thread *t);
struct thread *thread_current(void);
void thread_exit(void) NO_RETURN;
void thread_yield(void);
struct thread {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
/************************** Pintos Memory ******************************/
void *palloc_get_page(enum palloc_flags); (PAL_ASSERT|PAL_ZERO|PAL_USER)
void *palloc_get_multiple(enum palloc_flags, size_t page_cnt);
void palloc_free_page(void *page);
void palloc_free_multiple(void *pages, size_t page_cnt);
#define PGBITS 12 /* Number of offset bits. */
#define PGSIZE (1 << PGBITS) /* Bytes in a page. */
unsigned pg_ofs(const void *va);          uintptr_t pg_no(const void *va);
void *pg_round_up(const void *va);        void *pg_round_down(const void *va);
#define PHYS_BASE 0xc0000000
uint32_t *pagedir_create (void);          void pagedir_destroy(uint32_t *pd);
bool pagedir_set_page(uint32_t *pd, void *upage, void *kpage, bool rw);
void *pagedir_get_page(uint32_t *pd, const void *upage);
void pagedir_clear_page(uint32_t *pd, void *upage);
bool pagedir_is_dirty(uint32_t *pd, const void *upage);
void pagedir_set_dirty(uint32_t *pd, const void *upage, bool dirty);
bool pagedir_is_accessed(uint32_t *pd, const void *upage);
void pagedir_set_accessed(uint32_t *pd, const void *upage, bool accessed);
void pagedir_activate(uint32_t *pd);
/*********************** Pintos File Systems *************************/
bool filesys_create(const char *name, off_t initial_size);
struct file *filesys_open(const char *name);
bool filesys_remove(const char *name);
struct file *file_open(struct inode *inode);
struct file *file_reopen(struct file *file);
void file_close(struct file *file);
struct inode *file_get_inode(struct file *file);
off_t file_read(struct file *file, void *buffer, off_t size);
off_t file_write(struct file *file, const void *buffer, off_t size);
bool inode_create(block_sector_t sector, off_t length);
struct inode *inode_open(block_sector_t sector);
block_sector_t inode_get_inumber(const struct inode *inode);
void inode_close(struct inode *inode);
void inode_remove(struct inode *inode);
off_t inode_read_at(struct inode *inode, void *buffer, off_t size, off_t offset);
off_t inode_write_at(struct inode *inode, const void *buffer, off_t size, off_t offset);
off_t inode_length(const struct inode *inode);
bool free_map_allocate(size_t cnt, block_sector_t *sectorp);
void free_map_release(block_sector_t sector, size_t cnt);
size_t bytes_to_sectors(off_t size);
struct inode_disk {
    block_sector_t start;              /* First data sector. */
    off_t length;                      /* File size in bytes. */
    unsigned magic;                    /* Magic number. */
    uint32_t unused[125];              /* Not used. */
};
/************************* Pintos Devices ***************************/
typedef uint32_t block_sector_t;
#define BLOCK_SECTOR_SIZE 512
void block_read(struct block *block, block_sector_t sector, void *buffer);
void block_write(struct block *block, block_sector_t sector, const void *buffer);
```