OS Final: Concurrency & Persistence

Fall 2022 (Lecture: Remzi Arpaci-Dusseau, Textbook: OSTEP)

Ruixuan Tu, Feijun Chen
{ruixuan.tu, fchen222}@wisc.edu
University of Wisconsin-Madison

Concurrency

- **APIs** (all return 0 on success or errno on error)

```c
void *mythread(void *arg) { ... }; // example of start_routine
int pthread_create(pthread_t *thread, NULL, void *(*start_routine)(void *), void *arg); // e.g., (&p1, NULL, mythread, "A")
int pthread_join(pthread_t *thread, NULL); // wait for thread to finish; e.g., (p1, NULL)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // init a lock
int pthread_mutex_lock/unlock(pthread_mutex_t *mutex);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; // init a condition variable
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex); // assume mutex is locked, release mutex and put caller to sleep (not ready); when signaled, reacquire mutex before returning
int pthread_cond_signal(pthread_cond_t *cond);
sem_t sem; // semaphore
int sem_init(sem_t *sem, 0, unsigned int value); // 0: shared between threads in same process
int sem_wait(sem_t *sem); // sem->value--; wait if sem->value < 0
int sem_post(sem_t *sem); // sem->value++; wake up one or more waiting threads
```

- **Threads** (Ch. 26)
  - **thread**: very much like a separate process, except for that they *share* the same address space and thus can access the same data
    - **states**: private PC, set of registers, contexts (with switch except for page table)
    - **state saving** to **process control block** (PCB) for process, **thread control block** (TCB) for thread
    - **multiple stacks** for multi-threaded process: variables, parameters, return values, etc. in **thread-local** storage (the stack of relevant thread)
    - **reasons** [**parallelism**] *single-threaded* program to multiple CPUs. [**avoid slow I/O blocking**] enables *overlap* of I/O with other activities within a single program, much like *multiprogramming* did for processes across programs
  - **scheduler**: what runs next is determined by the OS *scheduler*, and it is hard to know what will run at any given moment in time; a new thread may run immediately or put in "ready" but not "running" state
  - **concurrent issues**
    - **critical section**: a piece of code that accesses a shared resource, usually a variable or data structure
    - **race condition/data race**: arises if multiple threads of execution enter the *critical section* at roughly the same time; both attempt to update the shared data structure, leading to an *indeterminate* (and perhaps undesirable) outcome
    - an **indeterminate** program consists of one or more *race conditions*; the output of the program is not *deterministic*, depending on which threads ran when
    - to avoid these, threads should use **mutual exclusion** primitives to guarantee that only a single thread ever enters a *critical section*, thus avoiding races, and resulting in *deterministic* program outputs
  - **atomicity**: "as a unit", or "all or none" for a series of actions called a *transaction*, no in-between state visible
  - **synchronization primitives**: hardware provides a few useful instructions upon which we can build a general set of what we call *synchronization primitives*, to build multi-threaded code that accesses critical sections in a synchronized and controlled manner

- **Locks** (Ch. 28)
  - **criterias**
    - **mutual execution**: basic task, lock called *mutex* in POSIX library
    - **fairness**: does any thread contending for the lock starve while doing so, thus never obtaining it?
    - **performance**: the time overheads added by using the lock (in single/multiple threads)
  - **coarse-grained** (big lock that is used any time any critical section is accessed); **fine-grained** (protect different data structures with different locks, allowing more threads in locked code at once)
  - **controlling interrupts**. *good*: simplicity. *bad*: priviledged operation with trust (monopolize CPU over OS), does not work on multiprocessors (enter on another CPU), lost interrupts (e.g., disk), inefficient

```c
void lock() { DisableInterrupts(); }
void unlock() { EnableInterrupts(); }
```

  - **spin lock**: use CPU cycles until lock available, requires a *preemptive scheduler* (i.e., interrupt via timer). *good*: correctness (*mutex*). *bad*: fairness (no guarantee), performance (overhead on single CPU, good when # threads ≈ # CPUs)
    - **test-and-set** or **atomic exchange** (`XCHG`)

```c
int TestAndSet(int *old_ptr, int new_value) {
  int old_value = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new_value; // store 'new_value' into old_ptr
  return old_value; // return the old value
}
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *lock) {
  // 0: lock is available, 1: lock is held
  lock->flag = 0;
}
void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1)
    ; // spin-wait (do nothing)
}
void unlock(lock_t *lock) { lock->flag = 0; }
```

    - **compare-and-swap/compare-and-exchange** (`CMPXCHG`)

```c
int CompareAndSwap(int *ptr, int expected, int new_value) {
  int original = *ptr;
  if (original == expected)
    *ptr = new_value;
  return original;
}
void lock(lock_t *lock) {
  while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; // spin
}
```

      - identical to `TestAndSet` when using spin lock, but provides lock-free synchronization
    - **load-linked** and **store-conditional** (RISC)

```c
int LoadLinked(int *ptr) { return *ptr; }
int StoreConditional(int *ptr, int value) {
  if (no update to *ptr since LoadLinked to this address) {
    *ptr = value;
    return 1; // success
  } else return 0; // failed to update
}
void lock(lock_t *lock) {
  while (1) {
    while (LoadLinked(&lock->flag) == 1)
      ; // spin until it is 0
    if (StoreConditional(&lock->flag, 1) == 1)
      return; // if set-it-to-1 succeeded: all done
              // otherwise: try again
  }
}
void unlock(lock_t *lock) { lock->flag = 0; }
```

  - **ticket locks**
    - store `ticket`, `turn` (which process to enter critical section)
      - **good**: ensure progress for all threads (once assigned ticket value, scheduled in the future)
      - **bad**: [without yield] waste time slice if wait for lock which will not be available, e.g., N threads contending a lock, $N - 1$ time slice wasted
    - **fetch-and-add** (`XADD`) and **yield**

```c
int FetchAndAdd(int *ptr) { int old = *ptr; *ptr = old + 1; return old; }
typedef struct __lock_t { int ticket; int turn; } lock_t;
void lock_init(lock_t *lock) { lock->ticket = 0; lock->turn = 0; }
void lock(lock_t *lock) {
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn)
    yield(); // spin(): discussed above
}
void unlock(lock_t *lock) { lock->turn = lock->turn + 1; }
```

- **test-and-set** and **yield**

```c
void init() { flag = 0; }
void lock() {
  while (TestAndSet(&flag, 1) == 1)
    yield(); // give up CPU
}
void unlock() { flag = 0; }
```

  - **yield**: *deschedules* caller itself by moving from *running* state to *ready* state

- **queues**: sleeping instead of spinning

```c
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
void lock_init(lock_t *m) { m->flag = 0; m->guard = 0; queue_init(m->q); }
void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; // acquire guard lock by spinning
  if (m->flag == 0) {
    m->flag = 1; // lock is acquired
    m->guard = 0;
  } else {
    queue_add(m->q, gettid());
    setpark(); // if then interrupted, then park() will return immediately, avoid wakeup race
    m->guard = 0;
    park(); // deschedule caller
  }
}
void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; // acquire guard lock by spinning
  if (queue_empty(m->q)) m->flag = 0; // let go of lock; no one wants it
  else unpark(queue_remove(m->q)); // hold lock for and wake up next thread
  m->guard = 0;
}
```

  - **good**: no waste, avoid starvation
  - **bad**: (limited) if interrupted in acquiring/releasing lock, then other threads spin-wait for this to run again; (without `setpark()` – about to sleep) **wakeup race** if another thread released the lock, the `park()` by this thread sleep forever
  - `park()` and `unpark()` switch state between *running* and *waiting* or *sleep* (not *ready*)
  - Linux-based *futex* locks

```c
void futex_wait(void *address, int expected); // if *address != expected, return immediately, else sleep caller
void futex_wake(void *address); // wake up one thread sleeping on queue
```

- **two-phase lock**
  - **reason**: spinning can be useful, particularly if the lock is about to be released
  - **procedure**: (1) the lock spins for a while, hoping that it can acquire the lock; (2) if could not acquire, the caller is put to sleep, and only woken up when the lock becomes free later by *futex* lock

- **Locked Data Structures** (Ch. 29)
  - **concurrent counter**

```c
typedef struct __counter_t {
  int global; // global count
  pthread_mutex_t glock; // global lock
  int local[NUMCPUS]; // per-CPU count
  pthread_mutex_t llock[NUMCPUS]; // ... and locks
  int threshold; // update frequency
} counter_t;
void init(counter_t *c, int threshold); // record threshold, init locks, init values of all local counts and global count
void update(counter_t *c, int threadID, int amt); // usually, just grab local lock and update local amount; once local count has risen 'threshold', grab global lock and transfer local values to it
int cpu = threadID % NUMCPUS; // map thread ID to CPU ID
int get(counter_t *c); // grab global lock and return global amount (approximate)
```

    * **naive** bad: only one thread can increment the counter at a time
    * **approximate** *idea*: have per-thread counters; periodically merge counter values. *good*: multiple threads (scalable). *bad*: (1) only approximate value; (2) read-heavy workloads can still cause lock contention

  - **concurrent queue**

```c
typedef struct __node_t { int value; struct __node_t *next; } node_t;
typedef struct __queue_t { node_t *head; node_t *tail; pthread_mutex_t head_lock; pthread_mutex_t tail_lock; } queue_t;
void Queue_Enqueue(queue_t *q, int value); // new tmp node; lock tail; add to tail; unlock tail
int Queue_Dequeue(queue_t *q, int *value); // lock head; remove from head (or empty); unlock head
```

    * **dummy node** (0th): to make accesses to head and tail pointers independent (for 2 small locks), value is not used. *good*: no need 1 big lock

  - **concurrent linked list** *hand-over-hand locking/lock coupling*: a lock per node, grab next node's lock and release current node's lock

- **Condition Variables** (Ch. 30)
  - **condition variable**: an explicit queue that threads can put themselves on when some condition is not desired (by *waiting* on condition); when some other thread changes state, can wake one or multiple waiting threads (might not all) and allow them to continue (by *signaling* on condition)
    * **good**: allow not only mutual execution, but also ordering of thread execution
  - **rules**
    1. keep state in addition to condition variables. if state is already as needed, thread does not call wait on CV
    2. protect shared state in concurrent programs. hold the lock while changing the shared variable and calling `signal()` to *avoid race conditions*
    3. always check state after waking up
       * **problem**: spurious wake-ups (system threads might wake up even if `signal()` not called; `signal()` may wake up more than one thread)
       * **solution**: (1) verify the state has changed as expected before continuing; (2) use `while`, not `if` when waiting on a condition variable, and `wait()` when not satisfied

  - **`join()` implementation**

```c
void thread_exit(thread_t *t) {
  mutex_lock(&t->mutex);
  t->done = 1; // might already terminated before join()
  cond_signal(&t->cond);
  mutex_unlock(&t->mutex);
}
void thread_join(thread_t *t) {
  mutex_lock(&t->mutex);
  while (t->done == 0) // rule (3)
    cond_wait(&t->cond, &t->mutex);
  mutex_unlock(&t->mutex);
}
```

  - **producer/consumer (bounded buffer) problem**
    * **put and get routines**

```c
int buffer[MAX];
int fill_ptr = 0, use_ptr = 0, count = 0;
void put(int value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1) % MAX;
  count++;
}
int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1) % MAX;
  count--;
  return tmp;
}
```

    * **producer/consumer synchronization**

```c
cond_t empty, fill; mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == MAX)
      pthread_cond_wait(&empty, &mutex);
    put(i);
    pthread_cond_signal(&fill);
    pthread_mutex_unlock(&mutex);
  }
}
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
```

```
    while (count == 0)
      pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

  * **Mesa semantics**: when you call `signal()`, you do not immediately switch to a waiting thread but a waiting thread will instead be marked as ready
  * **problems**: (1) no data when consumer awake (after another consumer), solve by `while`; (2) all sleep (after producer filled data and a consumer exhausted data, then wake another consumer), solve by that a consumer/producer should not wake other consumers/producers, by `fill` and `empty`; (3) only one thread can fill or use a buffer at a time, solve by unlock when fill or use next buffer, lock before update `count`

- **Semaphores** (Ch. 31)
  - **value**: if negative, equal to # waiting threads, init value equal to # resources
  - **binary semaphores/locks**: init value 1, `sem_wait()` as `lock()`, `sem_post()` as `unlock()`
  - **semaphores for ordering**: waiting/signaling – *ordering primitive* (like condition variables); init value 0, parent runs and calls `sem_wait()` to sleep (value == -1), child runs and calls `sem_post()` to wake parent (value == 0)
  - **producer/consumer (bounded buffer) problem**: no need `count`, instead semaphores `empty` and `full`

```
sem_t empty, full, mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);
    sem_wait(&mutex); // not outer to avoid deadlock
    put(i);
    sem_post(&mutex); sem_post(&full);
  }
}
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&full); sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex); sem_post(&empty);
    printf("%d\n", tmp);
  }
}
int main() {
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0); // 0 are full
  sem_init(&mutex, 0, 1); // lock
}
```

  - **reader-writer locks** *good*: safe to have multiple readers in the critical section without writer (if a writer exists, no reader and other writers); *bad*: often add overhead
  - **throttling**: init value max # threads, to avoid too many threads acquiring large memory
  - **implementation**: 1 lock, 1 condition variable, 1 state variable for value

- **Bugs** (Ch. 32)
  - **atomicity violation**: a code region is intended to be atomic, but the atomicity is not enforced during execution; *solution* by adding locks
  - **order-violation**: A should always be executed before B, but the order is not enforced during execution; *solution* by condition variables
  - **deadlock**: no progress can be made because two or more threads are each waiting for another to take some action and thus none ever does
    * **reasons**: (1) complex dependencies; (2) encapsulation
    * **conditions**: happens when all hold: (1) mutual exclusion, (2) hold-and-wait, (3) no preemption, (4) circular wait
    * **solution**: eliminate any condition: (1) atomic but lock-free/wait-free, (2) acquire all locks at once, no more acquire until all released [less encapsulation or concurrency], (3) trylock and release another lock on failure [*livelock*: states constantly change without progress, solve by exponential random backoff], (4) partial order instead of total order
    * **avoidance**: (1) schedule so that no lock wait, (2) detect deadlock and restart

Persistence

1 s == 10^3 ms == 10^6 µs

- **Hardware**
  - **I/O Devices** (Ch. 36)
    * **reasons for OS controlling device**: (1) security; (2) virtualization [different kinds of hardware, concurrency]
    * **interface registers**: (1) *command register* stores commands for device (e.g., r/w block); (2) *data register* stores data to exchange between device and

outside; (3) *status register* keepss track of status of the register (e.g., if device busy)

· **access**: (1) *special I/O instructions* addition to CPU's instruction set (e.g., `IN` and `OUT` in x86); (2) *memory-mapped I/O*, device registers mapped into memory, != mmap

∗ **access protocols**

· **pooling**

– **procedure**: (1) spin until device is not busy (*pooling*); (2) write into the data and command registers; (3) do polling again until request done

– **analysis**: *good*: simple and working. *bad*: uses CPU excessively, data transfer uses a lot of CPU

· **interrupt**:

– **procedure**: change spin in pooling to `sem_wait(device_ready)`, when ready use `sem_post(device_ready)` to issue interrupt

– **analysis**: *good*: go to sleep instead of spin. *bad*: if device fast, very frequent interrupts; leads to context switch overhead

∗ **direct memory access**: bypass CPU using DMA (memory – DMI interface – I/O chip – storage). *analysis* faster than copying to CPU then disk; CPU can do other things when data moving; requires specialized hardware

– **HDDs** (Ch. 37) block device, read/write a block of data (typically 512 bytes/4 KB)

∗ **access physically**: *location* (ϕ, r) at platter p, *cylinder* has r, *track* has r, p, sector has ϕ, r, p; e.g. `surface 3, track 5, sector 7`; *platters* spin to ϕ by spindle (rpm), *arms assembly* moves to r simultaneously, only one *head* R/W at one time

∗ **access (R/W) time** = seek time (arm move to track) + rotational delay (block rotate under arm head) + transfer time (actual data move) [sorted from long to short, transfer very short] (causes random time >> sequential time)

∗ **throughput** = $\frac{\text{amount of data}}{T_{\text{access}}}$

∗ **interface**: linear array of blocks/sectors, can perform read/write

∗ **internals**

· 1+ **platters** that can spin around at a fixed rate

· an **arm** that can move along different tracks (a circle on a platter) with a read/write head

· **controller**: execute commands in buffer, write output to status and data registers. keep track of multiple actions at once (allows higher throughput, schedule actions to optimize delay)

∗ **track skew**: add some offset between tracks to tolerate rotational delay so that when doing sequential read, the arm can catch up without waiting for another full rotation cycle

∗ **track skew** another explaination: sectors on different tracks are offset on most disks, e.g., the "gap" between sectors 11 and 12. *good*: change tracks without stopping the platter rotation, allows for faster sequential reads

∗ **policies** for disk scheduling

· **SSTF/SSF** Shortest Seek Time First: pick requests on nearest track first, OS uses **nearest-block-first (NBF)** as no geometry. *bad*: not account for rotation → disk arm stay on same track for long time → starvation

· **SCAN/Elevator**: scan back and forth from outer track to inner track (called a *sweep*) to solve starvation problem. *bad*: not account for rotation, only seek

– **F-SCAN** for Freeze: executes a fixed number of operations in one batch (other operations later, fair to requests that are on other parts of the platter)

– **C-SCAN** for Circular: only moves into one direction (does not favor middle tracks)

· **SPTF/SATF** Shortest Positioning Time First/Shortest Access Time First: *best*: minimize both seek and rotation times (close). need geometry like track boundaries, perform inside drive

∗ **multi-zoned disk drives**: outer tracks have more sectors than inner tracks

– **RAID** (Ch. 38)

∗ **comparison**: N disks each with B blocks, S sequential bandwidth of a disk, R random bandwidth of a disk, T time a request to a single disk would take

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 for sure; $\frac{N}{2}$ if lucky	1	1
Sequential read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Read latency	T	T	T	T
Write latency	T	T	$2T$	$2T$

∗ **reasons** for multiple drives: (1) disk failure might occur, (2) capacity is not enough, (3) improve performance

* **RAID-0 – striping (no redundancy)**: store the data evenly across the disks
  - **layout** with chunk size = 1

disk 0	disk 1	disk 2	disk 3
0	1	2	3
4	5	6	7

  - logical address `A`, `disk_id = A % disk_count`, `offset = A / disk_count`
  - **read / write**: direct read / write, each issues 1 I/O
* **RAID-1 – mirroring**: have 2 copies of each block on different disks; (!) issuing large I/O requests to different parts of each mirror could achieve full bandwidth
  - **layout**

disk 0	disk 1	disk 2	disk 3
0	0	1	1
2	2	3	3

  - **read**: directly read one of the copies, 1 I/O; **write**: write to all copies in parallel, M (mirroring level) I/O; **recovery**: when a disk fails, there is another copy of data to be used
* **RAID-4 – saving space with parity**: use a disk as a parity disk, each bit stores the parity information about the other bits in that position on other disks
  - **layout**

disk 0	disk 1	disk 2	disk 3
0	1	2	P0
3	4	5	P1
6	7	8	P2

  - **read**: direct read, issue 1 I/O; parallel read at most $N-1$ since one disk is parity disk
  - **write** (use either): (1) read other blocks and compute parity; write the block to be changed and the new parity block. (2) read old data and old parity, then compute new parity and write 2 blocks. (!) both need 2 reads and 2 writes (with subtractive parity), or $N-1$ reads and 2 writes (with additive parity)
  - **parity computation**: parity of a row is the XOR of all the bits in that row
* **RAID-5 – rotated parity**: store the parity block on different disks sequentially in a rotated manner (e.g. first parity block on last disk, second one on second last)
  - **layout**

disk 0	disk 1	disk 2	disk 3
0	1	2	P0
3	4	P1	5
6	P2	7	8

  - **read**: same as RAID-4, random better as used all disks; **write**: same as RAID-4, random much better as allows request parallelism

- **File Systems**
  - **Files & Dirs** (Ch. 39)
    * **file**: array of bytes (low-level name *inode number*)
    * **directory/dir**: a special type of file; array of records (human-readable names of files / dirs), map the names to inode nums
    * **operations**
      - **create**: calls `creat()` system call
      - **read**: use the file descriptor (int). OS map the FD to the file
      - **grow**: calls `write()` system call; calls `lseek()` to the end of file (set the offset to point to the end of file) then write there
      - **truncate**: truncate the file to start a new one; one way to do this is to use `O_TRUNC` flag in `open()`
      - **remove**: calls `unlink()`. *user-level cmd* `rm`
      - **rename**: calls `rename()`. *user-level cmd* `mv`
      - **link**
        - **hard link**: make another name refer to the same file (points to the same inode num), with the same stats; need to delete all linked files in order to delete the file. *user-level cmd* `ln`
        - **soft/symbolic link**: create a file of a special file type, the content has what it is linked to; can leave daggling pointers (when the file pointed to is removed). *user-level cmd* `ln -s`
      - **mount**: make a file system seems to be under a dir of another file system; allows us to create one big FS from many disks
      - **metadata**: stores the file info (name, size, blocks, inode number, # links, access time, modification time, etc.)
      - **path traversal** with root dir `/`
        - **absolute pathnames**: start at root, go down until getting to desired file / dir; ignore redundant slushs, i.e `//////` == `/`
        - **relative pathnames**: relative to current working dir (CWD); `.` refers to current dir; `..` refers to parent dir
  - **Implementation** (Ch. 40)
    * **disk interface**: an array of blocks `SidIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD`
      - 1 **super block** (`S`): contains info about the entire FS; tells where the other block regions are
      - 2 **bitmap blocks** (`i` for inode, `d` for data): tracks if a block is free
      - 1 **inode block** (`I`, for 32 inodes with 128 bytes each): stores type (regular file, dir, sym link), ownership, access rights, size, # blocks, pointers to data blocks (*direct ptrs* having a fixed number of blocks that points to the address of the data

block, thus have a max file size limit; *indirect ptrs* usually is the last ptr in the array of direct ptrs, pointing to another data block that is full of direct ptrs)
    · 32 **data blocks** (D): store data only
  * **make a FS `mkfs()`**: creates an empty file system (just a root directory)
  * **opening a file with absolute path**: (1) read root directory inode (usually a "well known" number such as 2) and then read root dir data; (2) check (2-1) right (if it's ok for current process to do specified operations), (2-2) unique (does file already exist); (3) read inode bitmap and find a free spot, then write to bitmap; (4) write dir data and dir inode
  * **writing to a file `write(fd, buffer, size)`**: (1) allocate a data block: (1-1) read the data bitmap and find a free block; (1-2) write to data bitmap. (2) update inode: (2-1) read reelevant inode block and update inode; (2-2) write inode block back. (3) write data to data block
  * **efficient access by page cache**: in OS memory, keep freq/recently accessed FS data. *good* for (1) reduces reads; (2) writes, allow to wait to write
- **Journaling** (Ch. 42): write-ahead logging
  * **log** - a special part of disk: before update, write info to log about update; want to update blocks atomically (all or nothing)
  * **FSCK** (File System Checker): scan the entire file system and fix inconsistencies
    · **checks**: pointed data block allocated? superblock match? dir contain . and ..? dir points to valid inodes? inode `size` and `nblocks` match? free bitmap? # dir entries == inode link count (update link count + mv to `/lost+found`)? different inodes point to same block (duplicate block)? bad ptrs (remove ref)?
    · **problems**: (1) slow; (2) no info about correct state, only know consistency
  * **journaling**: blocks designated to store notes
    · **assumptions**: issue many writes, some may complete (but not all) in case of crash / power loss and may complete in any order; issue a single 512-byte (sector) write atomically; want a transaction be atomic
    · **protocol**: (1) write all updates to log; (2) wait for I/O to complete; (3) issue updates to in-place final locations
    · **content**: a transaction begin block (T_b) info about the update; update info follows T_b; a transaction end block (T_e), which will be written after waiting for all the data block to be transferred
- **NFS** (Ch. 49):
  * **idempotent**: same when retry (e.g., lost packet)
  * **UDP**: could happen: messages arrive out-of-order at the client; messages are lost; NOT HAPPEN message content is corrupted, but still delivered
  * **operations**: `GETATTR`, `SETATTR`, `LOOKUP`, `READ`, `WRITE`, `CREATE`, `REMOVE`, `MKDIR`, `RMDIR`, `READDIR`; could accelerate by client-side caching (inconsistency in 3 sec before cache timeout, must flush-on-close), but not server-side write buffering
- **SSD** (Ch. 44): Flash-based Solid-State Disk
  * a blocked based storage device build upon flash chips; (!) a page in flash chip interface – 2-4 KB, a block – a chunk of pages, 128-256 KB
  * **operations**: read page, erase block (clears entire block), program page (can only program erased page and only once)
  * **properties**: [**performance**] *read I/O*: 10 μs (1000x faster than HDD); *erase*: a few ms; *program*: 100 μs. [**reliability**] erase/program a block too many (10k/100k, depending on density) times may *wear out* the chip
  * **flash translation layer (FTL)**
    · **goals**: convert logical blocks to physical blocks+pages; parallelism for multiple chips; reduces write amplification (less copying for block-level erases); implement wear leveling (distributes writes equally to all blocks)
    · **approaches**
      - **directly mapped**
        * **read**: just read the physical address as is in the drive interface
        * **write**: (1) identify block that write is within; (2) read other data out of the block; (3) erase the entire block; (4) program both the old data and the new data in; (5) write the block back to chip
        * **problems**: (1) wear out (needs to do unnecessary overwrites); (2) performance (needs to read and write the entire block)
      - **log structuring**
        * **copy-on-write**: not overwrite in place
        * always write new data to the end of log
        * **cleaning/garbage collection**: (1) pick a block; (2) identify live pages; (3) copy live pages (not dead pages) to the end of log; (4) erase the block; (!) defer at background
        * **wear leveling**: periodically erase long-lived blocks and rewrite elsewhere, to avoid no rewritten/garbage collection