```
/* ================================================================================================
 * OpSys_FinalExam_Questions
 * ================================================================================================
 * 01 -- Multiple Choice;               02 -- Term Matching;                   12 -- Bonus Question
 * ------------------------------------------------------------------------------------------------
 * 03 -- Memory and registers value with x86.py simulator
 * 04 -- Coordination with mutex and condition variable
 * 05 -- Drawing RAG and deadlock detection, deadlock avoidance
 * 06 -- Flash operations in log-structured SSD
 * 07 -- Multi-level index allocation and bitmap size in file system
 * 08 -- File system update w/ vsfs.py simulator
 * 09 -- Timelines of file access operations
 * 10 -- Allocation under ffs
 * 11 -- Journaling and meta journaling
 * ================================================================================================
 * (1) CONCURRENCY ( FinalReview_Part_1 )
 * ================================================================================================
 * 1.01 -- ( i) Process VS Thread
 * 1.01 -- (ii) PCB VS TCB
 * ------------------------------------------------------------------------------------------------
 * 1.02 -- Race Condition
 * ------------------------------------------------------------------------------------------------
 * 1.03 -- Implementation of Lock (HW solution, yielding, sleeping)
 * ------------------------------------------------------------------------------------------------
 * 1.04 -- Synchronization via Mutex & Condition Variable
 * ------------------------------------------------------------------------------------------------
 * 1.05 -- Synchronization via Semaphore
 * ------------------------------------------------------------------------------------------------
 * 1.06 -- Necessary Conditions for Deadlock
 * ------------------------------------------------------------------------------------------------
 * 1.07 -- ( i) Draw RAG
 * 1.07 -- (ii) Deadlock Detection
 * ------------------------------------------------------------------------------------------------
 * 1.08 -- Deadlock Prevention VS Deadlock Avoidance
 * ================================================================================================
 * (2) PERSISTENCE ( FinalReview_Part_2 )
 * ================================================================================================
 * 2.1  -- I/O Canonical Devices & Protocols
 * ------------------------------------------------------------------------------------------------
 * 2.2  -- ( i) Polling VS Interrupt VS DMA
 * 2.2  -- (ii) Programmed I/O (PIO)
 * ------------------------------------------------------------------------------------------------
 * 2.3  -- Hard Drive Disk (HDD) VS Solid State Disk (SSD)
 * ------------------------------------------------------------------------------------------------
 * 2.4  -- Disk Access Steps: Seeks, Rotation, Transfer
 * ------------------------------------------------------------------------------------------------
 * 2.5  -- Sequential Access VS Random Access
 * ------------------------------------------------------------------------------------------------
 * 2.6  -- Disk Scheduling (SSTF, C-SCAN/Elevator)
 * ------------------------------------------------------------------------------------------------
 * 2.7  -- Log-Structured FTL
 * ------------------------------------------------------------------------------------------------
 * 2.8  -- Reading/Writing Performance & Reliability for RAIDs 0,1,4/5
 * ------------------------------------------------------------------------------------------------
 * 2.9  -- Structure of VSFS
 * ------------------------------------------------------------------------------------------------
 * 2.10 -- ( i) Inode
 * 2.10 -- (ii) Multi-Level Inode
 * ------------------------------------------------------------------------------------------------
 * 2.11 -- Extend-Based VS Linux-Based VS FAT VS Index-Based Allocation
 * ------------------------------------------------------------------------------------------------
 * 2.12 -- Timelines for Operations: open(), creat(), mkdir(), read(), write() in VSFS
 * ------------------------------------------------------------------------------------------------
 * 2.13 -- How does FFS handle Long-Traveling issue?
 * ------------------------------------------------------------------------------------------------
 * 2.14 -- How does Journaling handle Crash issue?
 * ------------------------------------------------------------------------------------------------
 * 2.15 -- Meta-Journaling? How does it reduce writes?
 * ------------------------------------------------------------------------------------------------
 */
```

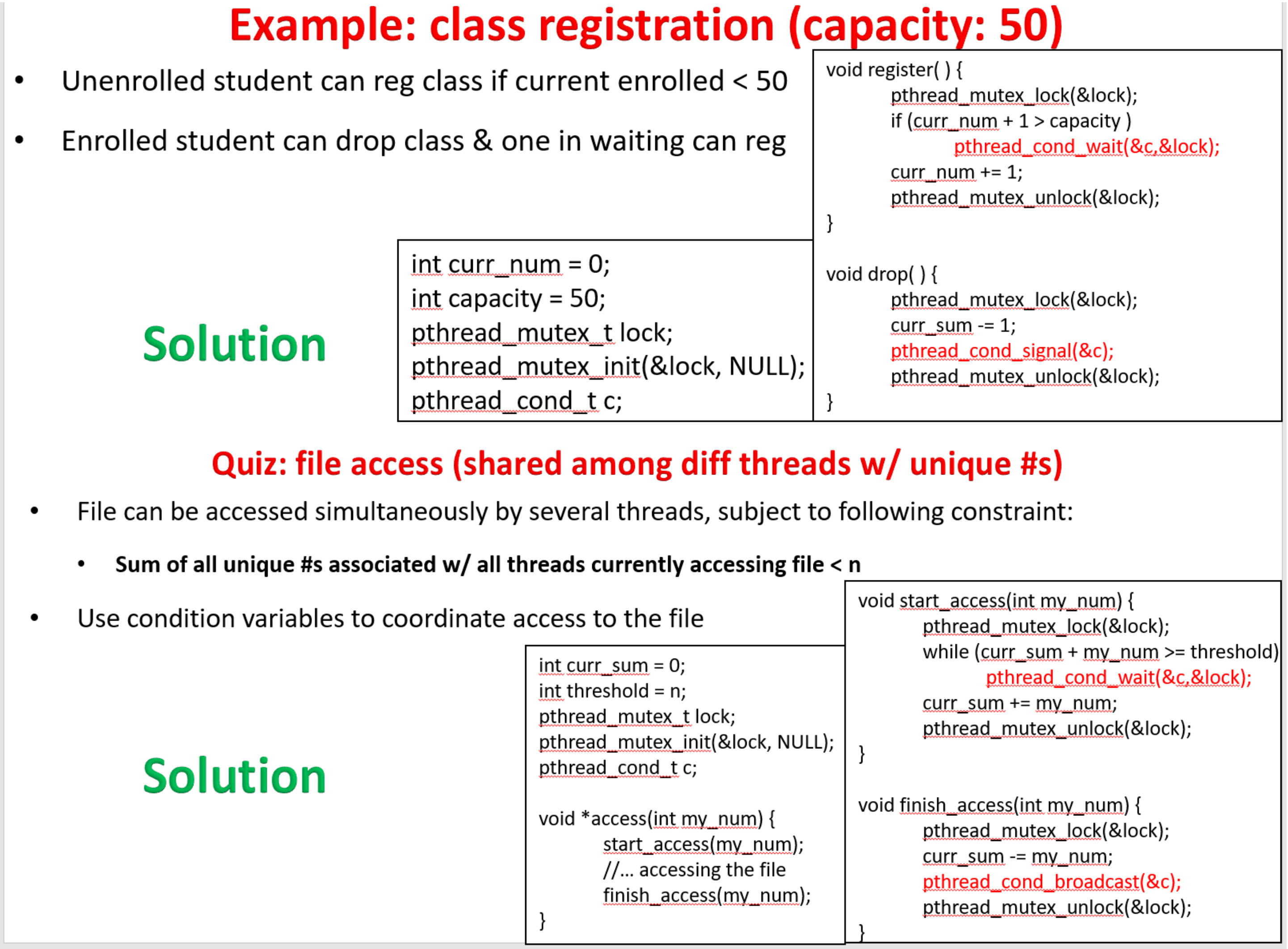
// 0.03 -- Memory and registers value with x86.py simulator

```
       // simulator x86.py to study concurrency w/ multithreading, fill in [ blanks ]

       2000         ax            bx     Thread 0                   Thread 1
          0          0             2
       [  0]         [0]           [2]    1000 mov  2000, %ax
       [  0]         [1]           [2]    1001 add  $1  , %ax
       [  0]         [0]           [2]    ---- Interrupt ----        ---- Interrupt ----
       [  0]         [0]           [2]                               1000 mov  2000, %ax
       [  0]         [1]           [2]                               1001 add  $1  , %ax
       [  0]         [1]           [2]    ---- Interrupt ----        ---- Interrupt ----
       [  1]         [1]           [2]    1002 mov  %ax , 2000
       [  1]         [1]           [1]    1003 sub  $1  , %bx
       [  1]         [1]           [2]    ---- Interrupt ----        ---- Interrupt ----
       [  1]         [1]           [2]                               1002 mov  %ax , 2000
       [  1]         [1]           [1]                               1003 sub  $1  , %bx
           1           1             1    ---- Interrupt ----        ---- Interrupt ----
           1           1             1    1004 test $0 , %bx
           1           1             1    1005 jgt .top
           1           1             1    ---- Interrupt ----        ---- Interrupt ----
           1           1             1                               1004 test $0  , %bx
           1           1             1                               1005 jgt  .top
           1           1             1    ---- Interrupt ----        ---- Interrupt ----
```

```
// Note that value at memory address 2000 is remains after each interrupt
// Counter++ has 3 instructions, which may result in interleaved operation!
// Interleaved operation is where a second op starts before the first one ends
// 1) moc 0x0MemAddr,      %eax          // Load val of counter from mem into reg
// 2) add $0x1      ,      %eax          // Increment counter
// 3) mov %eax      ,      0x0memAddr    //  Store it back into mem
// Since the instructions do not happen atomically (all at once) weird things can happen
```

// Results in race condition

```
// 0.04 -- Coordination with mutex and condition variable

/* KNOW WORKER PRODUCER CONSUMER PROBLEMS:
 * ----------------------------------------------------------------------------------------------------
 * lock_t mutex;
 * lock ( & mutex );
 * ctr++; // CRITICAL_SECTION
 * unlock ( & mutex );
 * ----------------------------------------------------------------------------------------------------
 * Conditional Variable:
 * =====================
 * Locks are not the only primitives that are needed to build concurrent programs
 * Before a thread continues its execution, most cases require checking for condition first
 *
 * These put threads in waiting state of its execution until condition is met
 * Then other threads may be "woken up" by previously waiting threads after execution
 * (this is called "signaling on the condition" )
 *
 * Condition Variuable has two operations associated w/ it:
 * -- wait()
 * A call that is executed when a thread wishes to put itself to sleep
 *
 * -- signal()
 * A call that is executed when a thread has changed something in the program & wants to wake a
 * sleeping thread waiting on this condition
 *
 * Condition Variables always used in conjunction w/ a mutex lock in pthread
 * pthread_cond_wait( & condition_var, & lock );
 * ----------------------------------------------------------------------------------------------------
 */
```



Example: class registration (capacity: 50)

- Unenrolled student can reg class if current enrolled < 50
- Enrolled student can drop class & one in waiting can reg

Solution

```
int curr_num = 0;
int capacity = 50;
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
pthread_cond_t c;
```

```
void register( ) {
        pthread_mutex_lock(&lock);
        if (curr_num + 1 > capacity )
                pthread_cond_wait(&c,&lock);
        curr_num += 1;
        pthread_mutex_unlock(&lock);
}

void drop( ) {
        pthread_mutex_lock(&lock);
        curr_sum -= 1;
        pthread_cond_signal(&c);
        pthread_mutex_unlock(&lock);
}
```

Quiz: file access (shared among diff threads w/ unique #s)

- File can be accessed simultaneously by several threads, subject to following constraint:
  - **Sum of all unique #s associated w/ all threads currently accessing file < n**
- Use condition variables to coordinate access to the file

Solution

```
int curr_sum = 0;
int threshold = n;
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
pthread_cond_t c;

void *access(int my_num) {
        start_access(my_num);
        //... accessing the file
        finish_access(my_num);
}
```

```
void start_access(int my_num) {
        pthread_mutex_lock(&lock);
        while (curr_sum + my_num >= threshold)
                pthread_cond_wait(&c,&lock);
        curr_sum += my_num;
        pthread_mutex_unlock(&lock);
}

void finish_access(int my_num) {
        pthread_mutex_lock(&lock);
        curr_sum -= my_num;
        pthread_cond_broadcast(&c);
        pthread_mutex_unlock(&lock);
}
```

lock()

- Calling the routine lock(&mutex) tries to acquire the lock mutex:
  - If no other thread holds the lock (i.e., it is free), the thread will acquire the lock (owner) & enter the critical section
  - If another thread then calls lock(&mutex) on that same lock mutex, it will not return while the lock is held by another threa

```
lock(&m
// critica
unlock(&
```

unlock()

- Once the owner of the lock calls unlock(&mutex), the lock is now available again:
  - If no other threads are waiting for the lock, the state of the lock is simply changed to free
  - If there are waiting threads (stuck in lock(&mutex)), one will eventually notice a change of the lock's state, acquire the lock, & enter the critical section

```
lock(&m
// critica
unlock(&
```

Example: Pthread Locks

- pthread_mutex_t mutex
- pthread_mutex_init(mutex, attr) // attr, mutex object attributes (default: NULL)
- pthread_mutex_destroy(mutex)
- pthread_mutex_lock(mutex) // Attempts to lock a mutex; thread blocks if already locked by another thread
- pthread_mutex_trylock(mutex) // Attempts to lock a mutex; returns w/ "busy" err code if already locked
- pthread_mutex_unlock(mutex); // Unlock a mutex if called by the owning thread

```
// CORRECTED EXAMPLE: "thread.c"
pthread_mutex_t lock;

void *worker(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
      pthread_mutex_lock(&lock);
      counter++;
      pthread_mutex_unlock(&lock);
   }
   return NULL;
}
```

```
// Bank Account Example:
// Use 1 lock for 2 critical sections!
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

void withdraw(int amount) {
        assert(amount < 0);
        pthread_mutex_lock(&lock);
        balance = balance + amount;
        pthread_mutex_unlock(&lock);
}

void deposit(int amount) {
        assert(amount < 0);
        pthread_mutex_lock(&lock);
        balance = balance - amount;
        pthread_mutex_unlock(&lock);
}
```

```
// CORRECTED EXAMPLE
pthread_mutex_t lock;

void *worker(void *arg)
   int i;
```

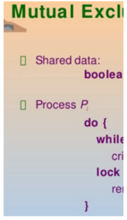
Locks: SW Solution

1st attempt w/o HW support

- A simple var to indicate whether some thread has possession of a lock

```
typedef struct __lock_t {
        int flag;
} lock_t;
void init(lock_t *lock) {
        lock->flag = 0;  // 0: unlocked, 1: locked
}
```

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = 0;            // 0: unlocked, 1: locked
}

void lock(lock_t *lock) {
        while (lock->flag == 1)  // TEST the flag
                ;                  // spin-wait (do nothing)
        lock->flag = 1;            // now SET it!
}

void unlock(lock_t *lock) {
        lock->flag = 0;
}
```



Mutual Excl

Shared data:
boolea

Process P
do {
while
cri
lock
re
}

oblem 2: spin waiting

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = 0;            // 0: unlocked, 1: locked
}

void lock(lock_t *lock) {
        while (lock->flag == 1)  // TEST the flag
                ;                  // spin-wait (do nothing)
        lock->flag = 1;            // now SET it!
}

void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

Solving spin-waiting: yielding

- Simple approach: when going to spin, instead give up the CPU to another thread.

```
void lock(lock_t *lock) {
        while (lock->flag == 1)
        yield(); // give up the CPU
}
```

- What happens? : OS switches the threads between running and ready/runnable states

Problem with yielding under many threads: it is still costly

Locks: HW Solution – Controlling interrupts

- Early solution used to provide mutual exclusion was to disable interrupts for critical sections – invented for single-processor systems

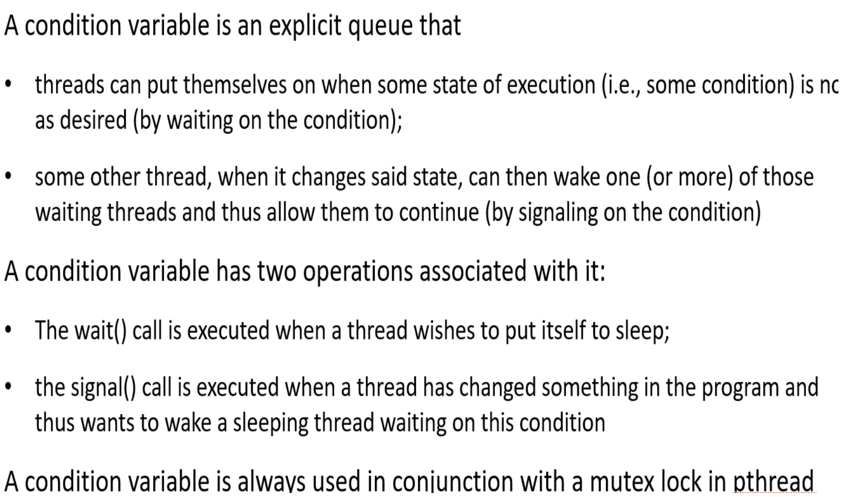
```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Features

- Pros: Simplicity
- Cons:
  - Requires allowing any calling thread to perform privileged operation (turning interrupts on/off), & thus trust that this facility is not abused
  - Inefficient: code that masks or unmasks interrupts tends to be executed slowly by modern CPUs
  - It does not work on multiprocessors

Other hardware primitives

- Test-And-Set (Atomic Exchange)
- Compare-And-Swap



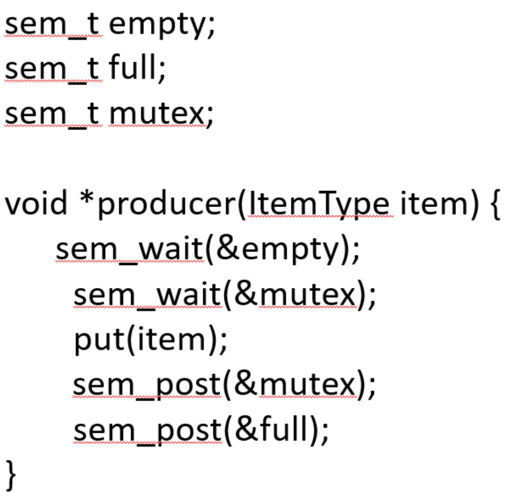
A condition variable is an explicit queue that

- threads can put themselves on when some state of execution (i.e., some condition) is nc as desired (by waiting on the condition);
- some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)

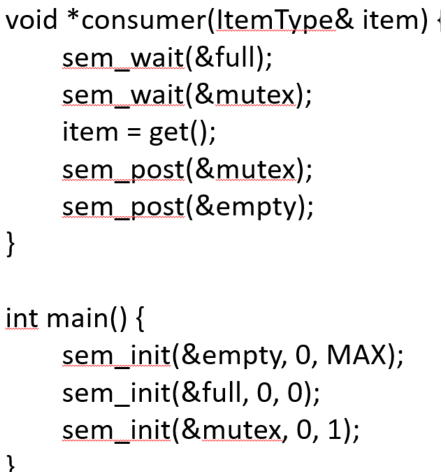
A condition variable has two operations associated with it:

- The wait() call is executed when a thread wishes to put itself to sleep;
- the signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

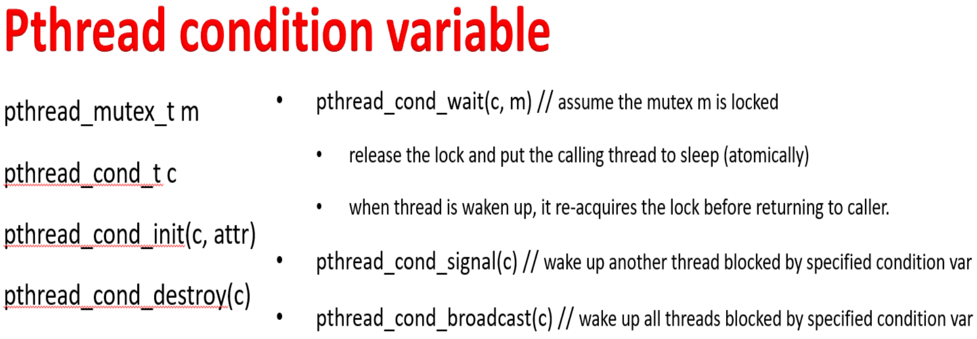
A condition variable is always used in conjunction with a mutex lock in pthread



```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(ItemType item) {
     sem_wait(&empty);
       sem_wait(&mutex);
       put(item);
       sem_post(&mutex);
       sem_post(&full);
}
```



```
void *consumer(ItemType& item) {
      sem_wait(&full);
      sem_wait(&mutex);
      item = get();
      sem_post(&mutex);
      sem_post(&empty);
}

int main() {
      sem_init(&empty, 0, MAX);
      sem_init(&full, 0, 0);
      sem_init(&mutex, 0, 1);
}
```



Pthread condition variable

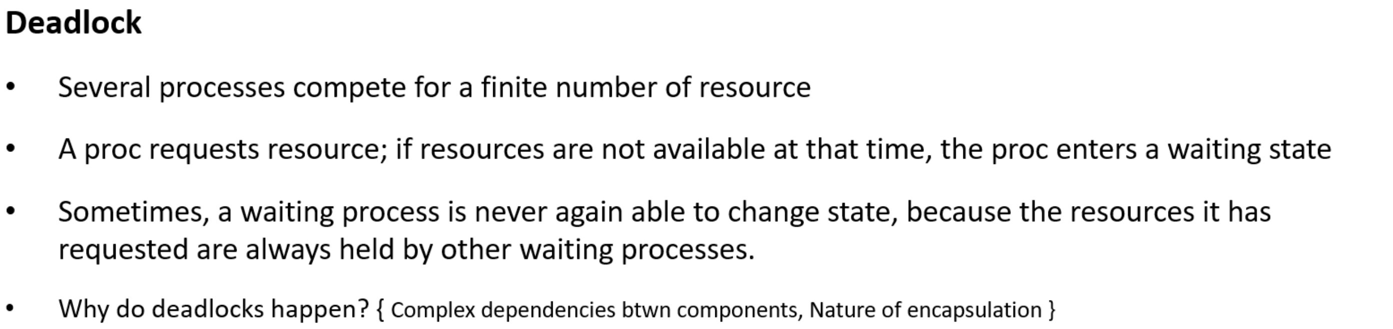
pthread_mutex_t m

pthread_cond_t c

pthread_cond_init(c, attr)

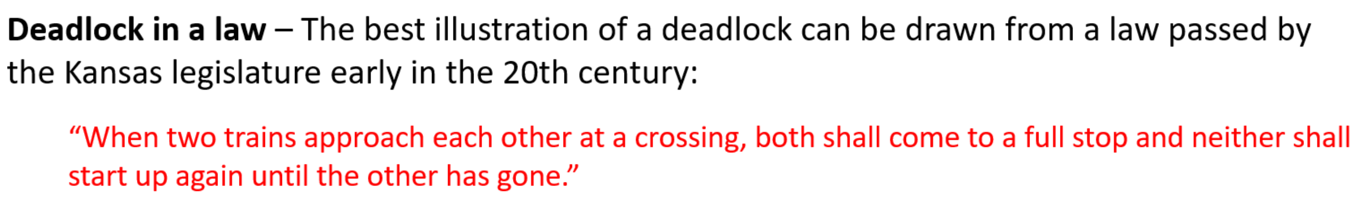
pthread_cond_destroy(c)

- pthread_cond_wait(c, m) // assume the mutex m is locked
  - release the lock and put the calling thread to sleep (atomically)
  - when thread is waken up, it re-acquires the lock before returning to caller.
- pthread_cond_signal(c) // wake up another thread blocked by specified condition var
- pthread_cond_broadcast(c) // wake up all threads blocked by specified condition var



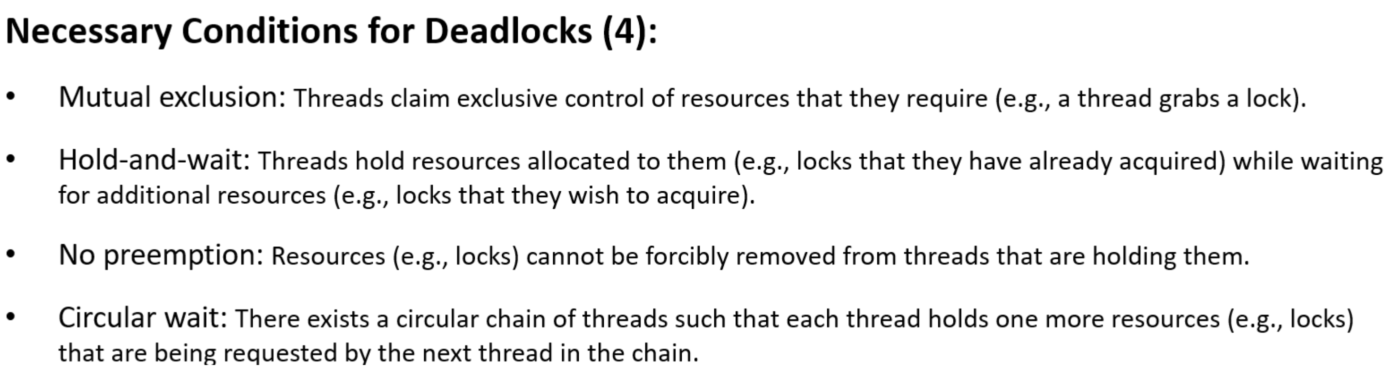
Deadlock

- Several processes compete for a finite number of resource
- A proc requests resource; if resources are not available at that time, the proc enters a waiting state
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are always held by other waiting processes.
- Why do deadlocks happen? { Complex dependencies btwn components, Nature of encapsulation }



Deadlock in a law – The best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century:

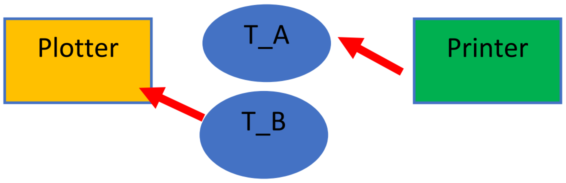
> "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."



Necessary Conditions for Deadlocks (4):

- Mutual exclusion: Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- Hold-and-wait: Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- No preemption: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- Circular wait: There exists a circular chain of threads such that each thread holds one more resources (e.g., locks) that are being requested by the next thread in the chain.

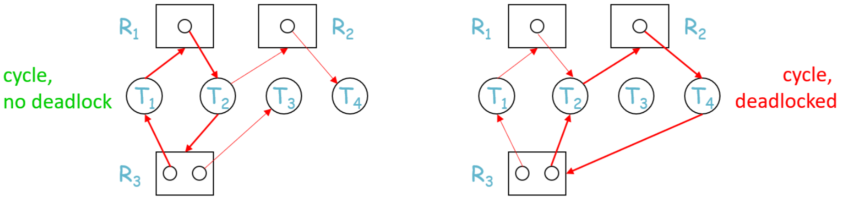
```
// 0.05 -- Drawing RAG and deadlock detection, deadlock avoidance
/* -------------------------------------------------------------------------------------------------
 * Resource Allocation Graph (RAG): for two threads accessing a printer & a plotter at position T
 * V = ( T = { Thread_A , Thread_B } , R = { R_Printer , R_Plotter } ) = Vertices
 * Thread A is assigned Printer Resource while Thread B is requesting Plotter Resource
 *
 * ( T_i, R_j ) in G -- Thread T_i is waiting for Resource R_j (request edge)  || (T_i) ----> [R_j]
 * ( R_j, T_i ) in G -- Resource R_j is allocated to T_i (assignment edge)     || (T_i) <---- [R_j]
 * -------------------------------------------------------------------------------------------------
 */
```



```
/* Necessary_Conditions_For_Deadlock: ( all four conditions must occur for deadlock )
 * -------------------------------------------------------------------------------------------------
 * i   ) Mutual Exclusion -- A resource accessed by multiple processes simultaneously
 * ii  ) Hold & Wait -- A resource holds while waiting requests for mult processes simultaneously
 * iii ) No Preemption -- Resources cannot be forcibly removed from processes that are holding them
 * iv  ) Circular Wait – Not requesting resources in an increasing order ( busy-waiting )
 * -------------------------------------------------------------------------------------------------
 */

/* Deadlock Prevention: ( use any one method to block one of the four necessary conditions )
 * -------------------------------------------------------------------------------------------------
 * i   ) Prevention of Mutual Exclusion – Preventing a resource from being accessed by multiple
 * processes simultaneously ( solution is to have the thread grab a lock )
 * ii  ) Prevention of Hold & Wait – Preventing resource holds while waiting requests for multiple
 * processes simultaneously  ( solution : lock all needed resources for thread )
 * iii ) Preemption (or prevention of no preemption) – Allow resources to be forcibly removed from
 * processes that are holding them ( solution is to not allow locks to be interrupted )
 * iv  ) Prevention of Circular Wait – Request resources in an increasing order so that no circular
 * wait condition may occur ( busy-waiting )
 * -------------------------------------------------------------------------------------------------
 */

/* Deadlock_Prevention vs Deadlock_Avoidance:
 * ---------------------------------------------------------------------------------------
 * Deadlock Prevention – Use a law opposing 1 of 4 necessary conditions ( mutual exclusion, hold &
 * wait, no preemption, circular wait). However, this leads to lower device utilization
 *
 * Deadlock Avoidance – Use scheduling to require global status of resource usage, & schedule all
 * requests accordingly (may even delay some requests)
 * ---------------------------------------------------------------------------------------
 */
```

RAG and deadlocks

- Observation 1: If a RAG does not have a cycle, then no process is deadlocked
- Observation 2: If a RAG has a cycle, then a deadlock may exist

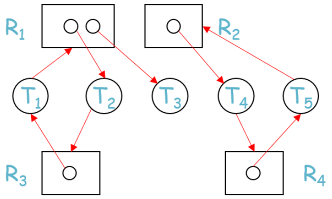


The existence of cycles in the RAG is necessary but not sufficient for a deadlock

Quiz: Any thread is deadlocked?

Why?

Answer: T4 & T5 are deadlocked

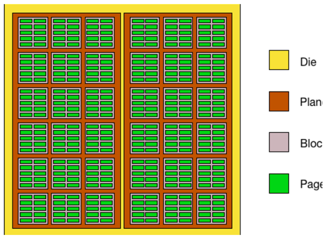


```
// 0.06 -- Flash operations in log-structured SSD
/* -------------------------------------------------------------------------------------
 * Solid State Drive (SSD): Benefits of DRAM (random access) & HDD (persistent)
 * -- Flash chips used to store bit(s) in single transistor (charge mapped to bin val)
 * -- Flash chips are organized into Planes (or banks), which contain block groupings
 * -- Each block has a number of pages (typical pg size: 2KB or 4KB)
 * FTL: Logical Blocks (pages) [LA] -> Physical Blocks (pages) [PA]
 * -------------------------------------------------------------------------------------
 * Flash Operations: { read(pg) - random access, write(pg) - erase block 1st all bits=1 }
 * Operation Format: { read(page=0), erase(block=0), & program(page=0,’a’) }
 * States of a Page: { INVALID(I), VALID(V), ERASED(E) }
 * Note that you can only program an erased page, & initial pages in block are set to I
 * -------------------------------------------------------------------------------------
 * Log-Structured FTL: uses mapping table to store phys addr of each logical pg in sys
 * -- for write to logical pg N, append write to next free spot of that specific pg
 * -- for reads to logical pg N, use mapping table (in memory)
 * -------------------------------------------------------------------------------------
 * Flash block size: 16KB && flash page size: 4KB --> 4 pages per block on flash chips
 * File system block size is 4 KB
 */
/* File Sys Writes & Sequence of Flash Operations:
  * -------------------------------------------------------------------------------------
-
 * write(int,char): syntax(starting flash block fsblock, data)
  * -------------------------------------------------------------------------------------
-
 * write(1000,’a’);     | erase(block=0);       // Erase block 0, set state='E' for pg0-3
 *                      | program(page=0,'a');  // program page 0 with data content 'a'
 * write(100, ’b’);     | program(page=1,'b');
 * write(500, ’c’);     | program(page=2,'c');
 * write(  1, ’d’);     | program(page=3,'d');
 * write(  2, ’e’);     | erase(block=1); program(page=4,'e');
 * write(300, ’f’);     | program(page=5,'f');
 * write(100, ’g’);     | program(page=6,'g');
 * write(500, ’h’);     | program(page=7,'h');
 * write(1000,’i’);     | erase(block=2); program(page=8,'i');
 */
```

Mapping Table (stores phys addr of each logical page) for Flash Transition Layer Device																
BEFORE																
fsblock (PA)	0				1				2				3			
fspage (PA)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
fsstate	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
AFTER																
fsblock (PA)	0				1				2							
fspage (PA)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
fsstate	V	V	V	V	V	V	V	V	V	E	E	E	I	I	I	I
data content	a	b	c	d	e	f	g	h	i							
mempage (LA)	1000	100	500	1	2	300	100	500	1000							

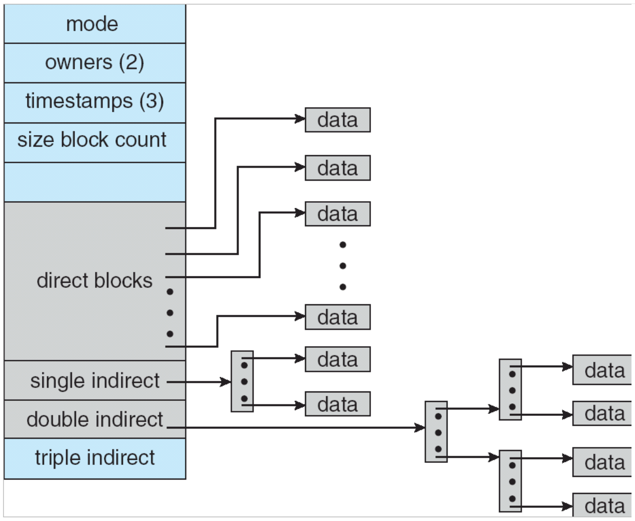
From bits to banks/planes

- Flash chips are organized into planes (or banks)
- Within each bank there are a large number of blocks
  - Block size: 128 KB or 256 KB
- Within each block, there are a large number of pages
  - Page size: 2KB or 4KB



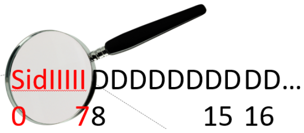
http://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases

```
// 0.07 -- Multi-level index allocation and bitmap size in file system

/* Multiple-Level Index in an inode:
 * ------------------------------------------------------------------------------------------
 * Direct Pointers
 * Indirect Pointers -- points to a block that contains multiple direct pointers
 * Double Indirect Pointers -- points to block that contains multiple indirect pointers
 * Tripple Indirect Pointers -- points to block that contains multiple double indirect pointers
 */
```

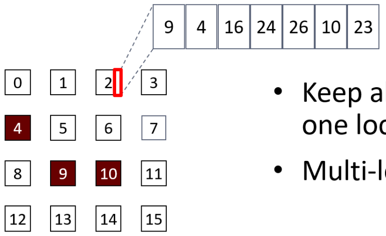


inode: example

- Allocation
  - The superblock starts at 0KB (1st block);
  - The inode bitmap is at address 4KB (2nd block);
  - The data bitmap at 8KB (3rd block).
  - inode region starts at 12KB (4th block).
- inodes
  - Size of each inode: 128B
  - Overall number of inodes: 160

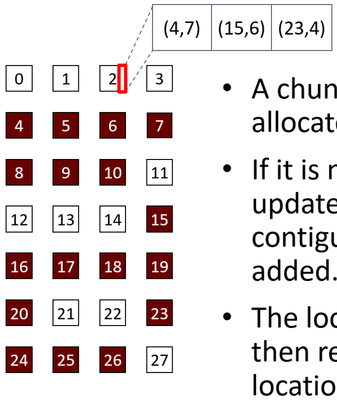


```
// Allocation_Methods:
```



- Keep all pointers to blocks in one location
- Multi-level index.

```
// Indexed-Based
```

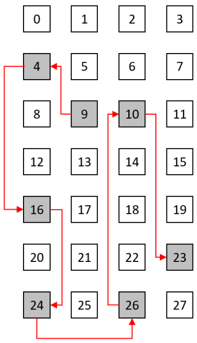


- A chunk of contiguous space is allocated initially.
- If it is not enough when the file is updated, another chunk of contiguous space (extent) is added.
- The location of a file's blocks is then recorded as a (first block location, block count) pair.

Extent-based (cont'd)

- Advantages
  - Minimizes head movements
  - Simplicity of both sequential and direct access
  - Particularly applicable to applications where entire files are scanned
- Disadvantages for limited available extents
  - When the disk is pretty full, the number of big extents are limited.

```
// Extent-Based (first_blk_loc, blk_cnt)
```



- Scatter logical blocks throughout the disk.
- Link each block to next one by forward pointer.
- May need a backward pointer for backspacing.

Linked-based (cont'd)

- Pros:
  - Flexible to use any block.
- Cons:
  - Poor direct access
  - Not reliable
- Improvement
  - Maintain all pointers as a separate linked list, preferably in main memory.
  - Example: File-Allocation Tables (FAT) in MS-DOS, OS/2.

```
// Linked-Based
// Extend-Based VS Linux-Based VS FAT VS Index-Based Allocation

/* Extent-Based allocation method:
 * Initially start w/ chunk of contiguously allocated space, if need more room when file
 * is updated, another chunk of contiguous apace (extent) is added.
 *
 * Location of a file's blocks recorded as a pair: (start_block_addr, block_count)
 *
 * pros: Minimal head movement, simple (sequential & direct access)
 * cons: Number of big extends available are limited when disk is near full
 * -------------------------------------------------------------------------------
 * File Allocation Table (FAT) allocation method:
 * Allocating new file block: { unused block (0), EOF block (-1) }
 * -- Get 1st '0' blk, replace '-1' blk w/ '0' blk addr, set '0' blk content to '-1'
 *
 * pros: Fast access
 * cons: Lacks scalability (whole table must always be in memory to work)
 */

/* *** HW4_PROBLEM_05: VERY_SIMPLE_FILE_SYSTEM (VSFS)
 * Find the physical address of the inode whose number is 101 in vsfs
 * inode region starting addr = 3200 KB
 * inumber of target inode = 101
 * inode object size = 128 B
 * ------------------------------------------------------------------------------------------
 * inode (index node): file meta data, implicitly referred to by its inumber
 * The reference num (inumber, e.g., 0,1,2,...) is the file’s low-level name
 */

offset = inode_size * inumber <==> 128 B * 101 = 12928 B

/* *** HW4_PROBLEM_06: FILE_SYSTEM_ORGANIZATION
 * Find max file size, & number of disk blocks required for 32GB disk bitmap
 * file sys which has: { disk block size = 2KB, disk block ptr = 4B }
 * file sys inode has: { 6 direct, 1 single, & 1 double indirect } disk blocks
 * ------------------------------------------------------------------------------------------
 * Bitmap    ____________________________________________________________________________
 *          |_S_|_i_|_d_|_I_|_I_|_I_|_I_|_I_|_D_|_D_|_D_|_D_|_D_|_D_|_D_|_D_|_D_|_D_|_D_|
 *          |_0_|___|___|___|___|___|___|_7_|_8_|_________________________15|_16|___|_18|
 *
 * Bitmap (free space management): gives status of blocks that divide up disk
 * Bitmap bit used to indicate if object/block is { free (0) or in-use (1) }
 * Bitmap Types     : i-bitmap (inode object), d-bitmap (data block)
 * Pros             : Bitmap is simple & efficient
 * Cons             : Bitmap requires extra space
 * An empty dir has        : 2 entries { itself “.” (dot), parent “..” (dot-dot) }
 * File descriptor         : int (fd) representing a “file” object
 * Disk Pointer     : Direct ptr to disk address
 */

// POINTERS_PER_BLCOK = sizeof( disk_block ) / sizeof( disk_ptr )
// 2K / 4 = 512 --> ln( 512 ) / ln( 2 ) = 9 ==> 2^9 [ pointers / block ]
POINTERS_PER_BLOCK: ppb = 2^9

// MAX_FILE_SIZE = blk_sz*(dir + ind_S*ppb^1 + ind_D*ppb^2 + ind_T*ppb^3)
// 2KB * ( 6 + 2^9 + 2^9 * 2^9 ) <==> ( 518 + 2^18 ) * 2KB = 525324000B
MAX_FILE_SIZE: mfs = 525MB

// NUMBER_OF_DISK_BLOCKS = sizeof( disk ) / sizeof( disk_block )
// 32GB / 2KB <==> 2^35 / 2^11 = 2^10 blocks ( or 16M blocks )
// 32*2^30 / (2^11*2^3*2^11) = 2^10
NUMBER_OF_DISK_BLOCKS = 10M disk blocks
```

```
// 0.08 -- File system update w/ vsfs.py simulator
// VSFS_FILE_SYSTEM_STATES: Init file sys states before operation are provided, update state after
// (1) ===========================================================================================
// inode bitmap    11110000
// inodes          [d a:0 r:3] [d a:1 r:2] [f a:-1 r:1] [f a:2 r:1][][][][]
// data bitmap     11100000
// data            [(.,0) (..,0) (c,1) (f,2) (b,3)][(.,1) (..,0)][a][][][][][]

mkdir("/c/y"); // makes new dir

// inode bitmap    11111000
// inodes          [d a:0 r:3] [d a:1 r:3] [f a:-1 r:1] [f a:2 r:1] [d a:3 r:2] [][][]
// data bitmap     11110000
// data            [(.,0)(..,0)(c,1)(f,2)(b,3)][(.,1)(..,0)(y,4)][a][(.,4)(..,1)][][][][]
// (2) ===========================================================================================
// inode bitmap    11000000
// inodes          [d a:0 r:3] [d a:1 r:2] [][][][][][]
// data bitmap     11000000
// data            [(.,0)(..,0)(s,1)] [(.,1)(..,0)] [][][][][][]

creat("/s/v"); // creats empty file

// inode bitmap    11100000
// inodes          [d a:0 r:3] [d a:1 r:2] [f a:-1 r:1] [][][][][]
// data bitmap     11000000
// data            [(.,0)(..,0)(s,1)] [(.,1)(..,0)(v,2)] [][][][][][]
// (3) ===========================================================================================
// inode bitmap    11110000
// inodes          [d a:0 r:3] [f a:1 r:1] [d a:2 r:2] [f a:-1 r:1] [][][][]
// data bitmap     11100000
// data            [(.,0)(..,0)(j,1)(y,2)] [k] [(.,2)(..,0)(f,3)] [][][][][]

fd=open("/y/f", O_WRONLY | O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
// write(int fd, const void * buf, size_t cnt); // write up to cnt bytes from buf2file
// O_WRONLY ensures it is only written to // O_APPEND ensures offset is EOF b4 each write

// inode bitmap    11110000
// inodes          [d a:0 r:3] [f a:1 r:1] [d a:2 r:2] [f a:3 r:2] [][][][]
// data bitmap     11110000
// data            [(.,0)(..,0)(j,1)(y,2)] [k] [(.,2)(..,0)(f,3)] [f][][][][]
// (4) ===========================================================================================
// inode bitmap    11001000
// inodes          [d a:0 r:3] [d a:1 r:2] [] [] [f a:2 r:1] [] [] []
// data bitmap     11100000
// data            [(.,0)(..,0)(n,1)(j,4)] [(.,1)(..,0)] [k] [][][][][]

link("/j", "/n/w"); // link("old/path","new/path");

// inode bitmap    11001000
// inodes          [d a:0 r:3] [d a:1 r:2] [] [] [f a:2 r:2] [][][]
// data bitmap     11100000
// data            [(.,0)(..,0)(n,1)(j,4)] [(.,1)(..,0)(w,4)] [k] [][][][][]

/* -------------------------------------------------------------------------------------------
 * NOTES (for vsfs file system state changes):
 * -------------------------------------------------------------------------------------------
 * Data_Organization: (entry_name, inode_number)
 * inode: [ f(type) a:-1(block) r:1(reference countr) ] <==> file, empty, 1 ref
 * [f a:10 r:1] <==> file, block 10, 1 ref
 * [d a:2  r:2] <==> dir,  block 2, 2 ref
 * -------------------------------------------------------------------------------------------
 * Reference Count:
 * -- Dir : Num of references to its inumber in all directories including itself
 * -- File: Num of references to its inumber in all directories
 * -------------------------------------------------------------------------------------------
 */
```

```
// 0.09 -- Timelines of file access operations (open()/creat()/mkdir()/read()/write() in VSFS)

/* inode_bitmap                    : indicates allocation status of inodes (1 - used, 0 - available)
 * inodes                          : table of inodes & their content
 * data_bitmap                     : indicates data block allocation status (e.g., 1110 - 1st 3 used)
 * data                            : indicates data block contents (e.g., [(.,0)(..,0)] for new dir)
 * ----------------------------------------------------------------------------------------------
 * mkdir()                         : creates a new dir
 * creat()                         : creates a new empty file
 * open(), write(), close()        : appends block to a file
 * link()                          : creates hard link to file (cmd: "ln" in Unix)
 * unlink()                        : unlinks file (removing it if link_count == 0)
 */
```

(Part-A): create dir w/ 2 data blocks: "/OS19W/Homework"										
	data bitmap	inode bitmap	root inode	OS19W inode	Homework inode	root data	OS19W data	Homework data [0]	Homework data [1]	comments
mkdir("/OS19W\ /Homework");			r							r root inode
						r				r root data block to get dir inode
				r						r dir inode addr to get dir data
							r			r dir data block
		r								r inode bitmap, find available inode #
		w								w to mark new inode as allocated
							w			w dir data (link child dir name-inode)
					r					r child dir inode dst into mem
					w					w to initialize child dir inode
				w						w to update parent dir inode

	data bitmap	inode bitmap	root inode	OS19W inode	Homework inode	HW1 inode	root data	OS19W data	Homework data [0]	Homework data [1]	HW1 data [0]	HW1 data [1]
(Part-B): create empty file : "HW1.txt"												
create("/OS19W\ /Homework\ /HW1.txt");			r									
							r					
				r								
								r				
					r							
									r			
		r										
		w										
									w			
						r						
						w						
					w							
(Part-C): write to HW1 file w/ 2 new data blocks (starting after opening file)												
write()						r						
	r											
	w											
											w	
						w						
write()						r						
	r											
	w											
												w
						w						
(Part-D): read two data blocks from HW1 file (starting after opening file)												
read()						r						
											r	
						w						
read()						r						
												r
						w						

```
// 0.10 -- Allocation under FFS (Fast_File_System)

// FFS_MAIN_GOAL: keep related stuff close & unrelated stuff far apart (spatial locality)

/* NOTES:
 * --------------------------------------------------------------------------------------
 * FFS Disk: divided up into a number of Cylinder Groups (set of N consecutive cylinders)
 * Cylinder: set of tracks on diff surfaces of hard drive, equidistant from HDs center
 * CylGroup: has { Super Block, Inode Bitmap, Data Bitmap, Inodes, Data Blocks }
 * BlkGroup: organizational structure typically used for file system, which actually
 * are just consecutive portions of a disk's logical addr space (not real groups)
 */

/* FFS_Directory_Policy:
 * Directory data & inode are allocated to a cylinder group w/ few number of directories
 * (to balance dir load across groups) & high number of free inodes (to allocate files)
 */

/* FFS_Sml_File_Policy:
 * --------------------------------------------------------------------------------------
 * (1) Allocate file's data blocks within cylinder group of its inodes (no long seeks)
 * (2) Place all files of same dir within cylinder group of the dir they are in
 */

/* FFS_Lrg_File_Policy:
 * --------------------------------------------------------------------------------------
 * Problem: Need to avoid lrg file filling up block group it was first placed in, as it
 * prevents related files from being able to be placed within that block group, as well
 * as hinders file-access locality.
 *
 * Solution: After N number of direct pointers within an inode (= number of blocks) have
 * been allocated into the first block group, the following chunk of the file will be
 * placed into a diff block group which is pointed to by the first indirect block.
 *
 * following portions (or chunks) of the file are placed into different block groups,
 * pointed to by indirect pointers
 */

/* LRG_FILE_PTR_STRUCT:
 * --------------------------------------------------------------------------------------
 * Larger files require structures within inodes. An indirect pointer is used to point to
 * more pointers which point directly to a data block. This type of inode structure
 * provides a single indirect ptr, as well as a small fixed number of direct pointers.
 *
 * Once a file in lrg enough, the indirect ptr will point to a newly allocated indirect
 * block (provided from a disk's data-block region), containing pointers to data blocks.
 *
 * Consequently, double & triple indirect pointers give further support as need be. A dbl
 * indirect ptr would reference a block of single indirect pointers. The single indirect
 * pointers would then reference indirect blocks with pointers to data blocks. Etc.
 */
```

```
// 0.11 -- Journaling & meta journaling

/* DATA_JOURNALING_NOTES: (How does Journaling handle crash issue?)
 * -----------------------------------------------------------------------------------------------
 * Write-Ahead Logging ("called journaling for file systems for historical reasons") was
 * the idea that when updating a disk, before overwriting structures, write a note first
 * that indicates what you are about to do. The written note is the "write-ahead", and it
 * is written to a structure which is organized as a "log"
 *
 * Updating (overwriting) structures may cause a crash, & if so, the journaling of which
 * structures were being updated before the even took place will greatly ease the amount
 * of work required for recovery afterwards (no need to scan disk, since it was logged)
 *
 * inode (I[v2]), bitmap (B[v2]), data block (Db)
 * Transaction Begin Block: TxBegin --> TxB
 * Transaction End Block: TxEnd --> TxE
 *          _______________________________________
 * Journal: |_TxB_(id=1)_|_I[v2]_|_B[v2]_|_Db_|_TxE_|
 *
 * Typically this mode of journaling is called Data Journaling since it logs all
 * of the user data in addition to the metadata of the file system
 * -----------------------------------------------------------------------------------------------
 * 4-Major_Steps_of_Data_Journaling:
 * =================================
 * (1) Journal Write           -- write contents of transaction (TxB, & update contents) to log
 *                             -- wait4write2complete
 *
 * (2) Journal Commit          -- Write transaction commit block (containing TxEnd) to the log
 *                             -- wait4write2complete (transaction is now committed)
 *
 * (3) Checkpoint              -- Write update contents to their final locations within file system
 *
 * (4) Free                    -- Mark transaction free in journal (via updating journal superblock)
 * -----------------------------------------------------------------------------------------------
 */

/* METADATA_JOURNALING_NOTES: (How does it reduce writes?)
 * -----------------------------------------------------------------------------------------------
 * Due to cost of writing every data block to the hard disk twice, a simpler form of
 * journaling called ordered journaling (or metadata journaling) was created.
 *
 * User data block (Db) is not written to the log in this case, it is written to the file
 * system proper (to avoid extra writing) - substantially reduces I/O load of journaling
 *
 * Crash_Consistency_Rule: "write pointed-to object before the object that points to it"
 * Need to write the Db (of reg files) to disk BEFORE the related metadata(I[v2] & B[v2])
 * -----------------------------------------------------------------------------------------------
 * 5-Major_Steps_of_Metadata_Journaling:
 * =====================================
 * (1) Data Write              -- write the final location
 *                             -- wait4write2complete (optional)
 *
 * (2) Journal Metadata Write  -- write the transaction begin block (TxB) & the metadata to the log
 *                             -- wait4write2complete
 *
 * (3) Journal Commit          -- Write transaction commit block (containing TxEnd) to the log
 *                             -- wait4write2complete (transaction & data is now committed)
 *
 * (4) Checkpoint Metadata     -- Write metadata update contents to final locations in file sys
 *
 * (5) Free                    -- Mark transaction free in journal (via updating journal superblock)
 * -----------------------------------------------------------------------------------------------
 */
```

- Interrupts thus allow for overlap of computation and I/O, which is key for improved utilization.

1: process 1
2: process 2
p: polling

Polling

```
CPU    1111111111pppppppppp11111111111
Disk   ----------1111111111-----------
```

Interrupt

```
CPU    1111111111222222222211111111111
Disk   ----------1111111111-----------
```

Direct Memory Access (DMA)

- When the main CPU is involved with the data movement, we refer to it as programmed I/O (PIO).
- Problem with PIO when transferring a large chunk of data:
  - The CPU is again overburdened with a rather trivial task.
  - Waste a lot of time and effort that could better be spent running other processes.

c: initiates the I/O copying the data from memory to the device explicitly, one word (each c) at a time.

PIO

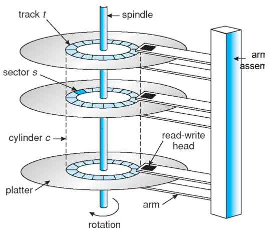
```
CPU    1111111111cccccc222222222222211111111111
Disk   ----------------1111111111-----------
```

DMA

```
CPU    11111111112222222222222222222211111111111
DMA    ----------cccccc----------------------
Disk   ----------------1111111111-----------
```

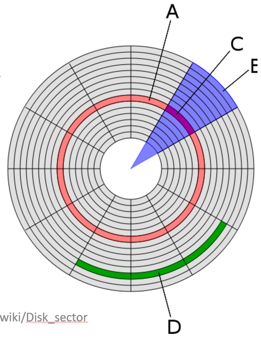
Basic geometry for magnetic disks

- Platter
  - Two surfaces of magnetic layer
  - Tracks, sectors
- Disk head
  - Attached to disk arm
- Spindle
  - 7,200 RPM to 15,000 RPM
- Access process
  - First, move disk arm to desired cylinder.
  - Then, rotates the desired sector into the position under the head.



Disk structure

- A: Track
- B: Geometrical sector
- C: Track sector
- D: Cluster



http://en.wikipedia.org/wiki/Disk_sector

Optimize I/O performance

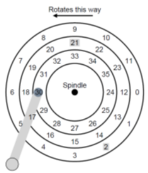
- I/O time
  - $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$
- I/O rate
  - $R_{I/O} = Size_{Transfer}/T_{I/O}$
- Objective
  - Large $R_{I/O}$ => short $T_{I/O}$

Performance evaluation

- Compare time for service for given request sequence, distinguish only by track.
- The seek time highly depends on locality.

SSTF: Shortest Seek Time First

- SSTF orders the queue of I/O requests by track, and picks the request on the nearest track to complete first.



Requests 21 and 26

Issues in SSTF

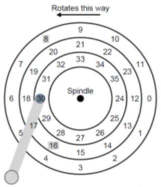
- Problem 1: the drive geometry is not available; rather, it sees an array of blocks.
  - Instead of SSTF, an OS can implement nearest-block-first (NBF)
- Problem 2: starvation
  - How can we implement a SSTF-like scheduling algorithm but avoid starvation?

SCAN and C-SCAN

- SCAN (or Elevator) simply moves back and forth across the disk servicing requests in order across the tracks.
  - We call a single pass across the disk (from outer to inner tracks, or inner to outer) a sweep.
- C-SCAN (circular SCAN)
  - Instead of sweeping in both directions across the disk, the algorithm only sweeps from outer-to-inner, and then resets at the outer track to begin again.
  - Doing so is a bit more fair to inner and outer tracks.

How to account for disk rotation costs?

- The head is currently positioned over sector 30 on the inner track.
- The scheduler thus has to decide:
  - Should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request.
- So which should it service next?
  - It depends.



Redundant Array of Independent Disks (RAID)

- RAID
  - A technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system.
- The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley
  - Led by Professors David Patterson and Randy Katz and then student Garth Gibson.

Features of RAID

- Transparency
- Capacity
- Reliability
- Performance

Interface and internals of RAID

- Interface
  - To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk.
- Internals
  - A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host.
  - At a high level, a RAID is very much a specialized computer system: it has a processor, memory, disks, and specialized software.
  - When a file system issues a logical I/O request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more physical I/Oes to do so.

How to evaluate a RAID

- Transparency
- Capacity
- Reliability
- Performance

Fault model

- Fail-stop: working or failed
- When a disk has failed, we assume that this is easily detected.

RAID designs

- RAID Level 0
  - striping
- RAID Level 1
  - mirroring
- RAID Levels 4/5
  - parity-based redundancy

RAID level 0: striping

- Spread the blocks of the array across the disks in a round-robin fashion.
  - Each round allocation is called chunk.
  - Each chunk could be multi-blocks.
- Chunk size
  - Small chunk size implies better parallelism of reads and writes to a single but longer the positioning time.
  - A big chunk size implied shorter positioning time to disks.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Chuck size = 1 block (4KB)

Disk 0	Disk 1	Disk 2	Disk 3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

Chuck size = 2 blocks (8KB)

RAID level 1: mirroring

- Duplicate chunks in multiple disks.
- The following arrangement is a common one and is sometimes called RAID-01 because it striped data (RAID-0) on top of mirrored pairs.
- Another common arrangement is RAID-10, which mirrors on top of two arrays that each perform striping internally.

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

RAID level 4: saving space with parity

- We added a single parity block that stores the redundant information for that stripe of blocks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

RAID level 5: rotating parity

- It rotates the parity block across the drives.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Outline

- Hard disk basics
- Hard disk scheduling
- RAID
- Flash-based SSD

Solid state disk (SSD)

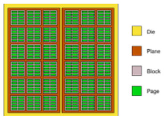
- NAND-based flash
  - Simply built out of transistors, much like memory (DRAM)
  - No mechanical or moving parts like hard drive disk (HDD)
- Benefits
  - Combination of both DRAM (random access) and HDD (persistent)

Storing bits

- Flash chips are designed to store one or more bits in a single transistor
  - The level of charge trapped within the transistor is mapped to a binary value
- Single-level cell (SLC) flash:
  - only a single bit is stored within a transistor (i.e., 1 or 0)
- Multi-level cell (MLC) flash:
  - More bits are encoded into different levels of charge
  - e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels

From bits to banks/planes

- Flash chips are organized into planes (or banks)
- Within each bank there are a large number of blocks
  - Block size: 128 KB or 256 KB
- Within each block, there are a large number of pages
  - Page size: 2KB or 4KB



http://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases

Basic flash operations

READ (A PAGE)

- Random access regardless of location on the device

Similar to DRAM

WRITE (A PAGE)

- Erase the whole block first
  - setting all bits in the block to 1
- Program some of the 1's within a page to 0's as desired

Different from DRAM

States of a page

- Invalid(i); Valid(V); Erased(E)
- You can only program an erased page

Operation		State	Description
		iiii	Initial: pages in block are invalid (i)
Erase()	->	EEEE	State of pages in block set to erased (E)
Program(0)	->	VEEE	Program page 0; state set to valid (V)
Program(0)	->	error	Cannot re-program page after programming
Program(1)	->	VVEE	Program page 1
Erase()	->	EEEE	Contents erased; all pages programmable

Erase (a block)

- Make sure that any data you care about in the block has been copied elsewhere (to memory, or to another flash block) before executing the erase.

A detailed example: modify Page 0

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

Page 0	Page 1	Page 2	Page 3
00000011	11001110	00000001	00111111
VALID	VALID	VALID	VALID

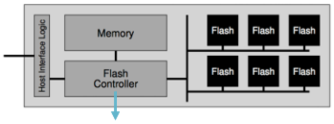
Reliability

- Wear out

P/E: Program/Erase

  - MLC-based block: 10,000 P/E cycles
  - SLC-based block: 100,000 P/E cycles
- Program disturbance
  - When programing a particular page within a flash, it is possible that some bits get flipped in neighboring pages

From raw flash to flash-based SSDs



- Flash translation layer (FTL)
  - logical blocks (pages) -> physical blocks (pages)

Design concerns for FTL

- Wear out
- Program disturbance
- Write amplification:
  - It is defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) issued to the SSD.

Design goals for FTL

- Reducing wear out
  - Wear leveling: try to spread writes across the blocks of the flash as evenly as possible.
- Reducing program disturbance
  - Program pages within an erased block in order, from low page to high page.
- Reducing write amplification
  - Reduce write traffic to SSD.

FTL organization approach: direct mapped

- A read to logical page N is mapped directly to a read of physical page N.
- A write to logical page N:
  - the FTL first has to read in the entire block that page N is contained within;
  - it then has to erase the block;
  - finally, the FTL programs the old pages as well as the new one.

Not good design!

A log-structured FTL

- Upon a write to logical page N, the device appends the write to the next free spot in the currently-being-written-to page.
- To allow for subsequent reads of page N, the device keeps a mapping table in its memory, and persistent, in some form, on the device
  - This table stores the physical address of each logical page in the system.

An example

Each time write/read a whole page (page size: 4K)

1. Write(100) with contents a1
2. Write(101) with contents a2
3. Write(2000) with contents b1
4. Write(2001) with contents b2

Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	i	i	i	i	i	i	i	i	i	i	i	i

An example

1. Write(100) with contents a1
2. Write(101) with contents a2
3. Write(2000) with contents b1
4. Write(2001) with contents b2

Erase Block 0

Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	E	E	E	E	i	i	i	i	i	i	i	i

An example

1. Write(100) with contents a1 — Program Page 0
2. Write(101) with contents a2 — Program Page 1
3. Write(2000) with contents b1 — Program Page 2
4. Write(2001) with contents b2 — Program Page 3

Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	V	V	V	V	i	i	i	i	i	i	i	i
	a1	a2	b1	b2								

Table 100->0; 101->1; 2000->2; 2001->3

An example

5. Write(100) with contents c1
6. Write(101) with contents c2

Erase Block 2
Program Page 4
Program Page 5

Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	V	V	V	V	V	V	E	E	i	i	i	i
	a1	a2	b1	b2	c1	c2						

Table 100->0; 101->1; 2000->2; 2001->3; 100->4; 101->5;

An example

- Garbage collection
  - Both Pages 0 & 1

Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	V	V	V	V	V	V	E	E	i	i	i	i
	a1	a2	b1	b2	c1	c2						

Table 100->0; 101->1; 2000->2; 2001->3; 100->4; 101->5;

An example

- Garbage collection
  - Both Pages 0 & 1
  - Erase Block 0 and Program Pages 2 & 3 into Pages 6 & 7

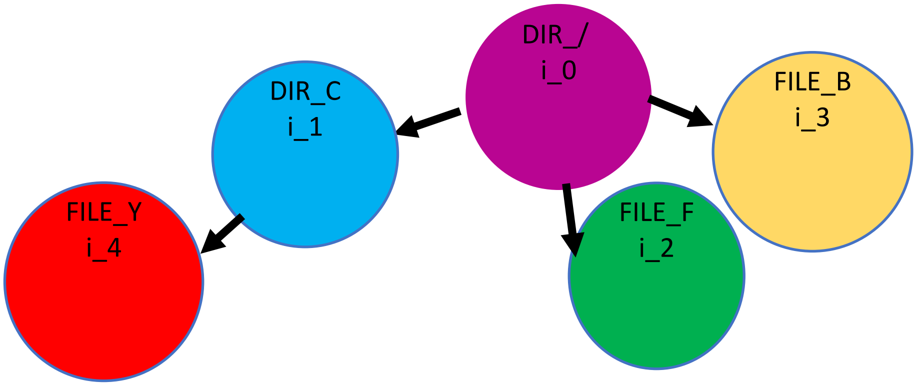
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
State	E	E	E	E	V	V	V	V	i	i	i	i
					c1	c2	b1	b2				

Table 100->4; 101->5; 2000->6; 2001->7;

```
// ================================================================================================
// (1) CONCURRENCY ( FinalReview_Part_1 )
// ================================================================================================
/* PROCESS_VS_THREAD:
 * -----------------------------------------------------------------------------------------------
 * Process (isolated memory):
 * -- Single point of execution within a program where instructions are fetched & executed from
 * -- Memory isolation for protection & reliability (e.g., google chrome browser)
 * -----------------------------------------------------------------------------------------------
 * Thread (shared addr space):
 * -- Threads within a process share the same address space (i.e., code, data, heap, & open files)
 * -- Each thread will have it's own unique program counter (PC), register set, & stack
 * -- Fast thread creation, fast context switching, saves memory by sharing (e.g., web servers)
 * -----------------------------------------------------------------------------------------------
 */

/* MULTI-THREAD_VS_MULTI-PROCESS:
 * -----------------------------------------------------------------------------------------------
 * Scenario__i (Multi-Thread Solution):
 * web server network: When too many processes are concurrently accessed.
 * -- The best method is a multi-threaded solution as they are accessing a shared memory which
 * provides opportunity for parallel execution of lighter-weight and more responsive processes.
 * Another opportunity for multi-threading optimization is provided when crunching large data set
 * computations, where you can transform a single-threaded program into a multi-threaded program
 * that can split up the work between multiple CPUs (parallelization)
 * -----------------------------------------------------------------------------------------------
 * Scenario_ii (Multi-Process Solution):
 * Internet Browsing: With only a single process requiring unique access to a memory resource.
 * -- The best method is a multi-process solution as it provides isolation (no other processes have
 * access to this memory space) between processes – ensuring a high degree of reliability.
 * However, this is more time & resource intensive due to added context switching btwn processes
 * -----------------------------------------------------------------------------------------------
 */

/* PCB_VS_TCB:
 * -----------------------------------------------------------------------------------------------
 * PROCESS_CONTROL_BLOCK (PCB):
 * Contains info on the containing process: Addr space, code, general registers, OS resources
 * -- But NO processor state, will have one or more TCB's linked to it
 * -----------------------------------------------------------------------------------------------
 * THREAD_CONTROL_BLOCK (TCB):
 * Contains specific info abt a single thread: PC, SP, thread state/status, reg values, ptr to PCB
 * -- Just processor state & pointer to corresponding PCB (of the process that the thread is in)
 * -----------------------------------------------------------------------------------------------
 */

// 1.02 -- Race Condition *** Talk about interleaved operations and the 3 instructions of a ctr
/* If two threads execute ctr++, sharing the ctr var, a race condition occurs. The ctr may not
 * always get incremented by both threads depending on the context switching times during execution
 * Race Condition (or data race) – Where the outcome of the code is dependent upon the timing of
 * the execution and therefore results in an indeterministic computation
 * Mutual_Exclusion – Prevent 2 concurrent processes from accessing critical section simultaneously
 */

// 1.03 -- Implementation of Lock (HW solution, yielding, sleeping)
/* Difference between yielding & sleeping methods used to overcome spinning waiting:
 * -----------------------------------------------------------------------------------------------
 * Since the sleeping method will not cause threads to compete for CPU resource, as they are placed
 * in blocked/waiting state, the sleeping method is more efficient for CPU utilization
 * Yielding Method: yield(); – Allows the thread to give up CPU to another thread, instead of
 * spinning (doing nothing). Causes OS to switch threads btwn running & ready/runnable proc states
 * Sleeping Method: sleep(); – Allows the thread to be placed into a sleep-queue. This will put the
 * sleeping thread into a blocked/waiting process state, where it will not compete for CPU resource
 * -----------------------------------------------------------------------------------------------
 * TestAndSet (Atomic Exchange) -- An atomic (non-interruptible) operation that enables “testing”
 * of old val (which returns int) while simultaneously “setting” the ptr (mem loc) to the new val.
 * This is used to implement lock as it can enables you to continuously test the lock's “flag”.
 * Thus, can force thread to wait, & set flag var to "locked" immediately after it becomes unlocked
 * New_t accesses crit sect w/ ensured mutual exclusion (e.g.: while(TestAndSet(&lock->flag,1));
 */
```

```
// ==========================================================================================
// (2) PERSISTENCE ( FinalReview_Part_2 )
// ==========================================================================================
// 2.01 -- I/O Canonical Devices & Protocols

/* Canonical_Device_Registers (3):
 * ---------------------------------------------------------------------------------
 * Status Register  : read to check current status of a given device
 * Command Register : used to tell the device to perform a given task
 * Data Register    : transmits or receives data to/from a given device
 */

// 2.02 -- ( i) Polling VS Interrupt VS DMA
/* Polling -- Good for fast devices
 * ------------------------------------------------------------------------------------------
 * Inefficient CPU time wasted for checking polling flags, however if you have more important tasks
 * allowed to interrupt the CPU & don't want to let unimportant tasks flood the CPU message stream,
 * polling is a good solution. E.g., PS/2 port interrupts from older keyboards vs USB bus polling
 * from modern keyboards. Or polling your e-mail periodically rather than getting spammed
 * interrupts throughout the day every time you receive an email.
 */

/* Interrupt -- Good for slow devices
 * ------------------------------------------------------------------------------------------
 * When you need something handled immediately, like say when the power button is hit on the \
 * computer & the hardware triggers an IRQ for an ISR to preserve the most important registers
 * before shutting down
 */

/* Direct Memory Access (DMA) -- Good for large data transfer
 * ------------------------------------------------------------------------------------------
 * Allowing the peripherals total control of the memory buses (privileged) to transport data back
 * and forth from the I/O devices to the main memory unit. E.g., GPUs capable of processing
 * graphics data may want to access main memory unit without the slowdown oversight management of
 * the CPU
 */

// 2.02 -- (ii) Programmed I/O (PIO)
// 2.04 -- Disk Access Steps: Seeks, Rotation, Transfer
// 2.05 -- Sequential Access VS Random Access
// 2.06 -- Disk Scheduling (SSTF, C-SCAN/Elevator)

// 2.08 -- Reading/Writing Performance & Reliability for RAIDs 0,1,4/5
/* ------------------------------------------------------------------------------------------
 * RAID-0 (striping): Spliting array of blocks across multiple disks (via RR)
 * pros: faster r/w speeds
 * cons: no data redundancy
 * ------------------------------------------------------------------------------------------
 * RAID-1 (mirroring): Duplicating full chunks of data onto multiple disks
 * pros: redundancy of data (not striping)
 * cons: slower (double the writing)
 * ------------------------------------------------------------------------------------------
 * Single-Disk vs. Two-Disk RAID-0 :
 * ================================
 * 2Disk_RAID0: higher chance of data loss, no redundancy, very fast for r/w
 * Single_Disk: better data integrity, slower due to only single disk for r/w
 * ------------------------------------------------------------------------------------------
 * Single-Disk vs. Two-Disk RAID-1 :
 * ================================
 * 2Disk_RAID1: fault tolerance (redundancy) but halves storage, slower r/w
 * Single_Disk: faster for r/w, equivalent storage space, zero data backup
 * ------------------------------------------------------------------------------------------
 */

// 2.10 -- ( i) Inode
// 2.10 -- (ii) Multi-Level Inode
```

```
HW_4
// 7: ALLOCATION_METHODS

/* Extent-Based allocation method:
 * -------------------------------------------------------------------------------------------------
 * Initially start w/ chunk of contiguously allocated space, if need more room when file
 * is updated, another chunk of contiguous apace (extent) is added.
 * -- Location of a file's blocks recorded as a pair: (start_block_addr, block_count)
 * pros: Minimal head movement, simple (sequential & direct access)
 * cons: Number of big extends available are limited when disk is near full
 */
/* File Allocation Table (FAT) allocation method:
 * -------------------------------------------------------------------------------------------------
 * Allocating new file block: { unused block (0), EOF block (-1) }
 * -- Get 1st '0' blk, replace '-1' blk w/ '0' blk addr, set '0' blk content to '-1'
 * pros: Fast access
 * cons: Lacks scalability (whole table must always be in memory to work)
 */
```



HW_3

1) *Describe scenarios to show that multi-process is better than multi-thread & vice versa*

Scenario i – Busy web server network: When too many processes are concurrently accessed. In this scenario, the better method is a multi-threaded solution as they are accessing a shared memory which provides opportunity for parallel execution of lighter-weight and more responsive processes. Another opportunity for multi-threading optimization is provided when crunching large data set computations, where you can transform a single-threaded program into a multi-threaded program that can split up the work between multiple CPUs (parallelization)

Scenario ii – Internet Browsing: When there is only a single process requiring unique access to a memory resource, the better method is a multi-process solution as it provides an avenue of isolation (no other processes have access to this memory space) between processes – ensuring a high degree of reliability. However, this is more time & resource intensive due to the added context switching between processes and the creation of new processes

2) *If two threads execute ctr++, sharing the ctr var, a race condition occurs. The ctr may not always get incremented by both threads depending on the context switching times during execution*

3)

Race Condition (or data race) – Where the outcome of the code is dependent upon the timing of the execution and therefore results in an indeterministic computation

Mutual Exclusion – Preventing two concurrent processes from accessing critical section simultaneously

4) *Since the sleeping method will not cause the threads to compete for CPU resource, as they are placed in the blocked/waiting state, the sleeping method is more efficient for CPU utilization*

Yielding Method: yield(); – Allows the thread to give up CPU to another thread, instead of spinning (doing nothing). This causes OS to switch the threads between running & ready/runnable process states

Sleeping Method: sleep(); – Allows the thread to be placed into a sleep-queue. This will put the sleeping thread into a blocked/waiting process state, where it will not compete for CPU resource

5) *TestAndSet to implement lock (0: unlocked, 1: locked): int TestAndSet(int * ptr, int new)*

TestAndSet (Atomic Exchange) – An atomic (non-interruptible) operation that enables "testing" of the old value (which is return int) while simultaneously "setting" the ptr (memory location) to the new value

This is used to implement lock as it can enables you to continuously test the "lock" ptr's Boolean member variable, "flag", passed in by reference. Therefore, it can force a thread to wait, and then set the flag variable to a new locked position immediately after it becomes unlocked, allowing new thread to access the critical section (ensuring mutual exclusion). E.g., while(TestAndSet(&lock->flag, 1));

6) *xv6 Locking – CPU & memory system communication w/ each other to guarantee operations take place on a memory location that xchg is currently reading to or writing to, in order to avoid data inconsistency. xchg is an atomic operation responsible for swapping words in memory w/ contents of a register. Thus, it can freeze CPU memory activities for a specified address while performing a context switch, and then unfreezes the memory activities once swapped*