CS162 Midterm 1

Dual mode operation: hardware supports kernel mode and user mode. 3 types of mode transfer:
(1) syscall: process requests system service
(2) interrupt: external asynchronous event
(3) trap/exception: protection violation, error

Processes operate in an address space distinct from other processes - simple implementation is base & bound (add base to address and check it is less than bound).

Processes represented in kernel by PCB with status (running, ready, blocked, finished), registers, PID, priority, execution time, translation tables, etc.

Example of shell implementation:

```
int status;
pid_t cpid = fork();
if (cpid == 0) { /* child */      // only returns if an error occurs
    execv(prog-path, prog-args);
    exit(0);
} else {          /* parent */
    wait(&status);          // same as waitpid(-1, &status, 0)
}
```

A child process contains an exact copy of the parent's address space and open file descriptor table, but henceforth operates independently. Note: exec family preserves file descriptors.

Signals
- int kill(pid_t pid, int signum)
- int signal(int signum, void (*handler)(int))

→ The 2nd argument can be SIG_IGN to drop the received signal, or SIG_DFL to do the default behavior.

→ For ex, SIGINT = 2 is keyboard interrupt. signal(SIGINT, SIG_IGN) tells program to ignore this signal.

File IO: A file descriptor is an index into a table stored in kernel. We can read/write to them. Initially, there are 3: stdin (0), stdout (1), and stderr (2).

- int open(const char *path, int oflags)
  open new file, return file descriptor. Offset initialized to zero.
- size_t read(int fd, void *buf, size_t nbytes)
  read n bytes into buf. File offset incremented by nbytes. * offsets are associated with the fd, and there may be multiple fds for file
- size_t write(int fd, void *buf, size_t nbytes)
  write n bytes from buf into fd at file offset. File offset incremented by nbytes.
- size_t lseek(int fd, off_t offset, int whence)
  moves offset of file. Options for whence:
  1) SEEK_SET: offset set to offset
  2) SEEK_CUR: offset set to cur_offset + offset.
  3) SEEK_END: offset set to size of file + offset
- int dup2(int oldfd, int newfd)
  duplicates oldfd using newfd (and closes newfd if was in use). Useful for output redirection, e.g. dup2(fd, stdout);

The file IO mentioned earlier are low-level kernel syscalls. fopen(), provided in stdio.h, returns a FILE * and can use many utility functions like fscanf(), fprintf(). These are streaming IO functions where buffering is handled automatically. fflush() or close() are needed to write buffer to stream.

Threads: Smallest unit of sequential instructions. Multiple threads of a process occupy the same address space, but operates on their own stack. Note: other thread stacks still accessible.

pthreads are POSIX-compliant thread implementations.

- pthread_create(pthread_t *t, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
  immediately creates & starts new child thread in same addr space running function specified
- pthread_join(pthread_t t, void **retval)
  wait for specific thread to terminate ← get retval of the thread's function
- pthread_yield(void)
  calling thread goes back into ready queue for a (possibly) new thread to run

Note: If parent thread terminates (i.e. return from main()) before children threads, the entire process is terminated. So parent should wait (via join) if necessary.

Synchronization

Atomic operation: execute to completion or not at all.

test_and_set is an atomic operation implemented in hardware. In C, it does

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

It can be used to implement a lock, or a primitive that protects a critical section from race conditions:

```
int locked = 0;

void lock() {                          // * better to busy wait on memory read than run test-and-set a lot
    do {
        while (locked == 1);
    } while (test_and_set(&locked));
}
void unlock() {
    locked = 0;
}
```

This is more desirable than implementing a lock by disabling interrupts, as user programs can use them.

A semaphore is a generalized lock. It is a non-negative integer that supports P(): atomic operation that waits for semaphore to become positive, then decrements it by 1; and V(): atomic operation that increments semaphore by 1 and waking up a waiter thread.

There are 2 general uses of semaphores:
(1) Mutual exclusion: initial value = 1
    used to wrap critical section
(2) Scheduling constraints: initial value = 0
    allows one thread to wait for another event to occur

Example: bounded buffer producer/consumer

```
semaphore fullslots = 0;
semaphore emptyslots = BUFSIZE;
semaphore mutex = 1;

Producer(item) {
    emptyslots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullslots.V();
}

Consumer(item) {
    fullslots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptyslots.V();
    return item;
}
```

A monitor is a lock and zero or more condition variables. Example usage in a queue:

```
void queue_init(queue *q) {
    pthread_mutex_init(q->lock, NULL);
    pthread_cond_init(q->cond, NULL);
    ...
}

queue_item queue_pop(queue *q) {
    pthread_mutex_lock(q->lock);
    while (queue_empty(q)) {
        pthread_cond_wait(&q->cond,
                          &q->lock);
    }   * atomically release lock and go to sleep. re-acquire lock before returning.
    ...
    pthread_mutex_unlock(&q->lock);
}

void queue_push(queue *q) {
    pthread_mutex_lock(q->lock);
    ...
    pthread_cond_signal(&q->cond);   // * wake up a waiter; use broadcast to wake all waiters
    pthread_mutex_unlock(&q->lock);
}
```

We use 'while' instead of 'if' in condition check because of Mesa monitors: the waiter gets placed back on ready queue, so another thread might run first & change condition.

Example: thread pool in a http server:

```
...
queue_init(&wq);
for (int i = 1; i <= n_threads; i++)
    pthread_create(&threads[i], NULL,
                   thread_loop, request_handler);
...
while (1) {
    queue_push(&wq, client_socket)
```

where thread_loop is defined as

```
void thread-loop (void (* request-handler)(int)) {
    int client_socket_fd;
    while(1) {
      client_socket_fd = queue_pop(&wq);
      request-handler(client-socket-fd);
      close(client-socket-fd);
}
```

Priority inversion occurs when a high priority thread is blocking on a resource that a lower priority thread holds access to. This can be solved w/ priority donation.

Deadlock occurs when two or more processes are waiting in turn on each other. An example is

```
void transfer(acct *donor, acct *rec, float amount)
  pthread_mutex_lock(&donor->lock);
  pthread_mutex_lock(&rec->lock);
  ...
```

Simultaneous & opposite transactions can result in deadlock. To resolve this, we could do *in general, deadlock occurs when threads acquire locks in a different order

```
  ...
  if(donor->id <= rec->id)
     pthread_mutex_lock(&donor->lock);
     pthread_mutex_lock(&rec->lock);
  else
     pthread_mutex_lock(&rec->lock);
     pthread_mutex_lock(&donor->lock);
...
```

Socket Programming socket descriptors allow us to read/write data over a network just like regular file descriptors.

Server: * this is server fd

```
int fd = socket(PF_INIT, SOCK_STREAM, 0);
bind(fd, &addr, sizeof(addr));
listen(fd, 0);
...
/* inside thread pool loop */
    int client_fd = accept(fd);
    handle(client_fd);
...
```

Client:

```
...
connect(fd, &addr, sizeof(addr))   // same fd & addr as above
...
```

Scheduling Various algorithms include

- FIFO - subject to convoy effect
- Shortest remaining time first (SRTF) - optimal but impossible, unfair
- Priority
- Round Robin (RR) - cycle between all jobs for a fixed quanta (CPU time)
- MLFQS - multiple queues of different priorities / algorithms
- Real-time scheduling

Example: *RR quanta = 1 *priority preemptive *SRTF break ties w/ priority

Process	CPU Burst	Arrives at	Priority
A	4	1	1
B	1	2	2
C	2	4	4
D	3	5	3

Time	1	2	3	4	5	6	7	8	9	10
FIFO	A	A	A	A	B	C	C	D	D	D
RR	A	A	B	A	C	A	D	C	D	D
SRTF	A	B	A	C	C	A	A	D	D	D
Priority	A	B	A	C	C	D	D	D	A	A

Total turnaround time
= $\sum_{\text{processes}}$ finish time + 1 − arrival time

Exam Trivia / Pintos Specifics

How to implement a "process pool" with limited number of processes & error message:

```
int count = 0;
int status;
while (1) {
   while (count > MAX_PROCESSES) {
      if (wait(&status) > 0) count--;
   }
   client_fd = accept(...)
   count++;
   if (fork() == 0) {
      if (count == MAX_PROCESSES + 1)
         send(client_fd, "Overloaded");
      else
         ... // communication
   }
   ...
}
```

Linux Syscalls: pass syscall number into register and then use a known interrupt. The interrupt handler gets the request number and looks up the function in a table.

Pintos kernel stack: a portion of the TCB is reserved for a thread's associated kernel stack.

Hyperthreading is a hardware technique utilizing superscalar processors to execute multiple threads simultaneously.

Therac-25 case study: series of race conditions in code led to machine malfunction, leading to deaths of several patients.

Transition from thread-running → waiting state: call sleep; lock-aquire; semaphore; monitor; make I/O call; call wait() on child process.

Bounded synchronized queue: use previous implementation, but add new cond variable space_available to queue structure. Wait on this variable with condition (queue->len == queue->capacity) in queue_push() and signal the space_available cond var in queue_pop().

- In Pintos, each process consists of a single thread. There is no multithreading within a process.
- Idle thread serves as placeholder so that scheduling logic is simple; e.g. switch_threads() still works.
- When interrupts are disabled, interrupts do not trigger the corresponding interrupt handler. This creates an atomic section. Note: does not work for multiple processors.
- schedule() removes threads in dying state and cleans up their memory.
- Space shuttle failed to launch in 1981 because of time synchronization issue with PASS and BFS.
- Threads in Pintos are page-aligned (last 12 bits). So masking off last 12 bits of stack pointer gives beginning of TCB (thread struct)

Readers / Writers Solution

Correctness Constraints: (1) readers access database only when no writers (2) writers access database only when no readers or writers (3) only one thread manipulates state variable at a time

State vars: int AR, WR, AW, WW;
(active reader, waiting reader, active writer, etc.)

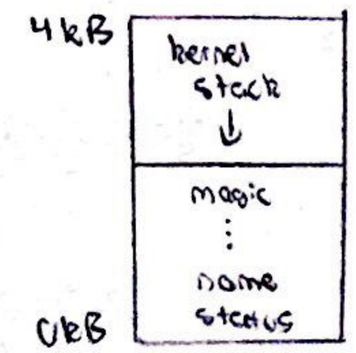
Reader()

```
lock.acquire();
while ((AW+WW) > 0) {
   WR++;
   okToRead.wait(&lock);
   WR--;
}
AR++;
lock.release();
// read database
lock.acquire();
AR--;
if (AR == 0 && WW > 0)
   okToWrite.signal()
lock.release()
```

Writer

```
lock.acquire();
while((AW+AR) > 0) {
   WW++;
   okToWrite.wait(&lock);
   WW--;
}
AW++;
lock.release();
// read/write data
lock.acquire();
AW--;
if (WW > 0)
   okToWrite.signal();
else if (WR > 0)
   okToRead.broadcast();
lock.release();
```

Pintos thread:



CS162 Midterm 2

Size prefixes (for bytes)

Value	Metric		Value	IEC	
10^3	kB	kilobyte	2^{10}	KiB	kibibyte
10^6	MB	megabyte	2^{20}	MiB	mebibyte
10^9	GB	gigabyte	2^{30}	GiB	gibibyte
10^{12}	TB	terabyte	2^{40}	TiB	tebibyte
10^{15}	PB	petabyte	2^{50}	EiB	pebibyte
10^{18}	EB	exabyte	2^{60}	PiB	exibyte

Note: in context of memory, kB, MB, and GB often refer to KiB, MiB, and GiB.

Powers of 2

$2^6 = 64$ $2^8 = 256$ $2^{10} = 1024$
$2^7 = 128$ $2^9 = 512$ $2^{11} = 2048$

Number conversions (Dec → Bin → Hex)

0 = 0000 = 0
1 = 0001 = 1
2 = 0010 = 2
3 = 0011 = 3
4 = 0100 = 4
5 = 0101 = 5
6 = 0110 = 6
7 = 0111 = 7
8 = 1000 = 8
9 = 1001 = 9
10 = 1010 = a
11 = 1011 = b
12 = 1100 = c
13 = 1101 = d
14 = 1110 = e
15 = 1111 = f

Advanced scheduling

Real-time scheduling is about enforcing predictability (minimizing worst-case response-time) rather than maximizing throughput.

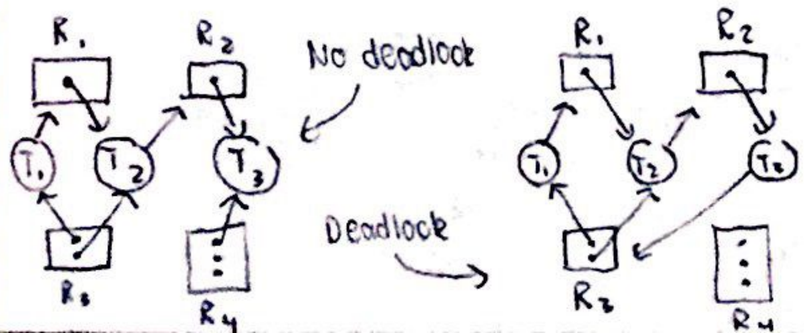
Earliest-deadline-first (EDF) is a hard real-time scheduler that always schedules the active task with the closest absolute deadline. Schedulability test: n periodic processes with deadlines equal to their periods are schedulable if $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$, where C_i = computation time and D_i = period.

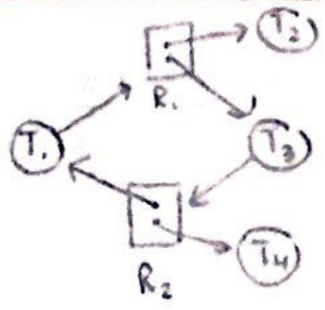
Starvation occurs when a thread waits indefinitely (e.g. when it waits for resources constantly in use by higher priority threads).

Deadlock occurs when threads wait on each other. To prevent, we need to ensure resources are accessed in the same order. 4 conditions for deadlock: (1) mutual exclusion - only one thread can use a resource at a time; (2) hold and wait - a thread holding a resource is waiting to acquire additional resources; (3) no preemption - resources are released voluntarily after being used; (4) circular wait - there is a circular dependency.

Resource Allocation Graphs

Request edge: $T_i \rightarrow R_j$
Assignment edge: $R_j \rightarrow T_i$





Resource allocation graph with cycle, but no deadlock because T_2 and T_4 could release their resources.

Banker's Algorithm is a deadlock avoidance algorithm that tests for 'safety'. We need the total # of available resources, and the current and maximum allocation of resources for each process.

Example: resources ABC and threads T_1, T_2, T_3, T_4.

Total

A	B	C
7	8	9

Current

	A	B	C
T_1	0	2	2
T_2	2	2	1
T_3	3	0	4
T_4	1	3	1

Max

A	B	C
4	3	3
3	6	9
3	1	5
3	3	4

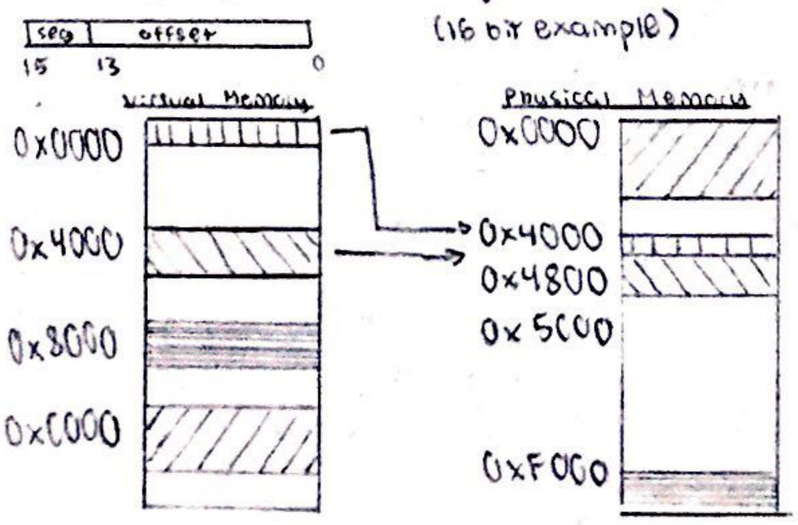
A state is considered safe if it is possible for all processes to finish executing.

1) Calculate available resources (in above example, sum up currently held resources and subtract from total)
2) Calculate needed resources matrix (max - current)
3) If there is a process for which all needed resources are available, run it and return its held resources to the available pool. If no such process, state is unsafe.
4) Repeat step 3 until all processes terminate (state is safe) or was declared unsafe.

Address Translation

Base and Bound (simple): add PHYS_BASE and check < BOUND to convert Virtual Address to Physical address. This works, but leads to fragmentation, inefficiency for sparse address spaces, and no support for address sharing.

Segmentation Separate logical segments into their own smaller contiguous blocks:

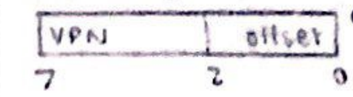


Segment Map (stored in CPU)

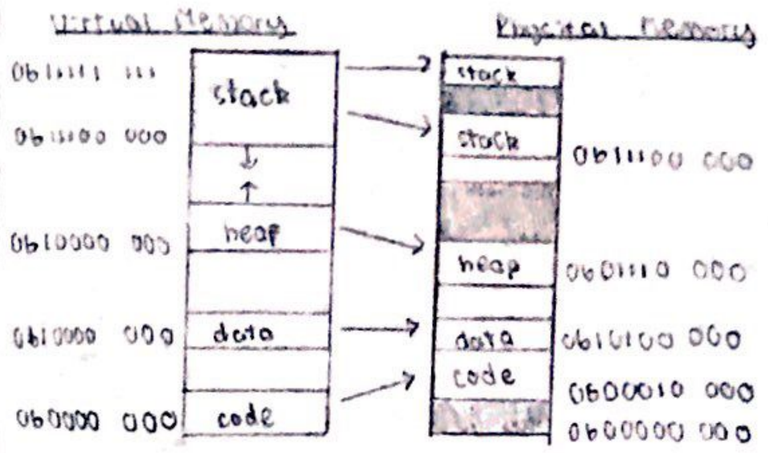
Segment ID		Base	Limit
0	code	0x4000	0x0800
1	data	0x4800	0x1400
2	shared	0xF000	0x1000
3	stack	0x0000	0x3000

- efficient for sparse address spaces
- allows for sharing & different permissions

Paging Solution to fragmentation from large contiguous segmentation. Allocate "pages", or small fixed-size chunks of memory.

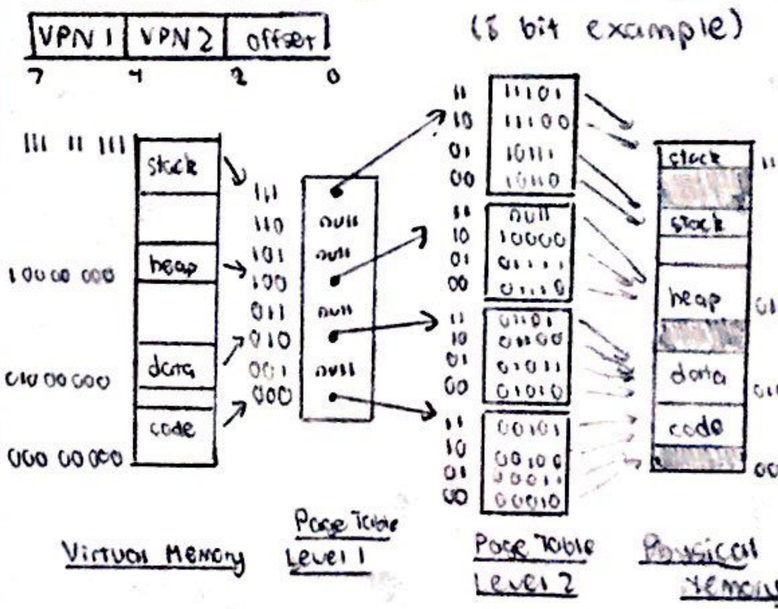


$\log_2$(page size in bytes) bits long (8 bit example)



Each VPN indexes into a page table, where each entry contains the PPN, page dirty, page valid, and permission bits. To context switch, simply change PTBR to point to that processes's page table (in physical memory), and invalidate TLB entries.

Multilevel Tables Problem with above is that page tables are too large (millions of entries). New idea: add another level, and only bring in second-level tables if accessed. Make tables to be 1 page in size for easy swaps.



Demand paging is the technique of bringing pages into physical memory from disk only when an access is attempted.

Memory access steps: Search TLB (a cache of PTEs) for VPN. If hit, we get the PPN. Otherwise, walk the Page table(s) to PTE. If valid, we get the PPN and update the TLB. If invalid, trap into OS (called page fault). Bring page into free frame or evict a page (writing to disk if dirty bit set), invalidating that page's PTE and TLB entry. Update TLB and PTE of new page with it's new PPN. Obtain address with PPN + offset. Thread then resumes from original faulting location (OS puts it on wait queue until finished).

Note: Physical pages are often shared across different processes (e.g. code for Chrome tabs). In Linux, children processes forked from parent share same pages until a write is attempted - then page is duplicated.

Caching Review

Caches exploit temporal and spatial locality of access.

cache tag	set index	block offset
Bits: remaining	$\log_2 S$	$\log_2 B$

B = size of cache blocks (in bytes), S = number of sets. Let K be the number of cache blocks (cache size / block size). Then

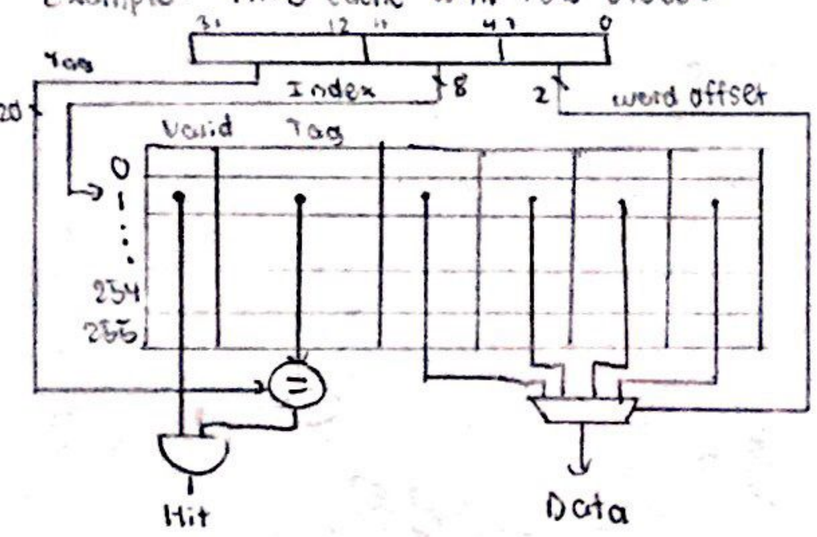
Fully Associative: 1 set with K ways

Direct Mapped: K sets with 1 way

N-way Set Associative: K/N sets with N ways

Total cache capacity = associativity × S × B

Example: 4KiB cache with 16B blocks:



Associative caches look the same, but have multiple 'ways' (copies of the above). A fully associative cache can never have a conflict miss, but they also have more overhead due to comparators.

Miss types

Compulsory: first access

Capacity: cache isn't large enough

Conflict: memory addresses map to same location. Solution: increase associativity or capacity.

Coherence: other processes invalidating cache entries

Average access time: (hit rate × hit time) + (miss rate × miss time)

Note: miss rate + hit rate = 1

Page / Cache Replacement Policies (for associative caches)

FIFO - evict oldest page

MIN - evict page that won't be used for the longest time. Optimal, but theoretical.

LRU - evict page least recently used. attempts to approximate MIN, but can perform badly for looping access patterns.

The clock algorithm is an approximation of LRU that replaces an old page. It keeps a circular list of pages in memory, with the "hand" pointing to the last examined page. When a page fault occurs, examine the use bit. If it is 0, the new page replaces it and the hand increments. Else, set to 0 and clock hand is incremented until we find a page with use bit 0.

Every time a page is accessed, the use bit is set to 1. This essentially gives pages "2 chances". There are generalizations of this with "Nth chance". A large N is closer to LRU while a small N is closer to FIFO - but large N is less efficient.

Bélády's Anomaly: phenomenon in which increasing the number of page frames / cache capacity actually increases the number of page faults. FIFO, clock, Nth chance are subject to this. LRU is not. Occurs only with specific access patterns.

I/O

Interrupts: Hardware triggers OS interrupt to handle incoming data stream. Invokes interrupt handler to transfer large amount of data. Good for high-bandwidth, low frequency devices (network card, disk drive). High overhead!

Polling: regularly check for new data. Good for mouse and keyboard, but infeasible for most things.

The CPU interacts with a device controller, which contains its own registers and memory for buffering requests. With I/O instructions, there are special assembly instructions to use these registers; in memory-mapped I/O, regular instructions are used because I/O forms extended part of address space.

Methods of data transfer with controllers:

Programmed I/O: each byte processed with loads and store in memory-mapped I/O.

Direct Memory Access (DMA): I/O devices read/write directly to memory without kernel supervision.

Miscellaneous

- Consider a machine with 8GB physical memory, 8KB pages, and PTE size of 4B. How many levels of page tables are needed to map a 46-bit virtual address space if page tables are single pages?
  - Answer: 2^{13} (page size) / 2^2 (PTE size) = 2^{11} entries per page. With 13 offset bits, there are 33 VPN bits $\Rightarrow 2^{33}$ virtual pages. One level addresses 2^{11} virtual pages; 2 levels addresses $2^{11} \cdot 2^{11} = 2^{22}$ pages; 3 levels addresses $2^{22} \cdot 2^{11} = 2^{33}$ pages. So 3 levels are needed.

Inverted page tables for systems with large virtual address spaces and small physical memory, page tables take up enormous space. Multilevel paging mitigates the problem, but requires more memory accesses. (slow) An alternative approach is an inverted page table, which is a global table. To translate a virtual address, the process ID and virtual page number are hashed to get an entry in the hash anchor table. The entry's pointer to a page table entry is followed, and the PID and VPN are compared against that stored there. If they don't match, the PTE's next pointer is followed. The process repeats until reaching a matching entry or null pointer (page fault). The PPN is obtained as the index of the entry.

- A system being in an unsafe state in the Banker's algorithm does not necessarily mean deadlock will occur.
- Dining philosopher solution: Number the resources (chopsticks) 1-5. Have each philosopher always pick up the lower numbered chopstick, then the higher numbered chopstick of the two they will use. (Alternatively, never give the last chopstick to a philosopher who doesn't already have one.)

Pintos Trivia

User VM ranges from virtual address 0 up to PHYS-BASE, which is 0xC0000000 (3GB). Kernel VM ranges from PHYS-BASE to 4GB. It is mapped one-to-one with physical memory, starting at PHYS-BASE.

Syscall.c: (+ implementation of SYS-INFO, which sets user processes name and returns tid)

```
static void syscall_handler (struct intr_frame *f)
{
  uint32_t *args = ((uint32_t *) f->esp);
  exit_thread_if_invalid(args, 4);
  if (args[0] == SYS_INFO) {
    exit_thread_if_invalid(&args[1], 4);
    exit_thread_if_invalid(args[1], 16);
    memcpy(args[1], thread_current()->name, 16);
    f->eax = thread_current()->tid;
  }
  ...
}
```

- Data caches use physical addresses, not virtual addresses. The TLB lookup and cache lookup occur simultaneously, since the untranslated offset is all that's needed before the comparator.
- Precise exceptions are those where the state of the machine is preserved up to the offending instruction. They are useful as they allow the OS to easily restart.

CIS2 Midterm 3

More I/O: A block device accesses large chunks of data (called blocks) while a character device accesses individual bytes.

DMA is appropriate for transferring large amounts of data without CPU usage (e.g. access disk sector) while memory-mapped I/O is appropriate for accessing devices directly from CPU (e.g. program interrupt vector)

The top half of a device driver is used by kernel to start I/O while bottom half services interrupts produced by the device.

Storage Devices

Latency is the time to perform an operation, throughput is the rate of operations, and startup (overhead) is the time to initiate.

Disk latency = Queuing time + controller + seek + rotation + transfer.

The average rotational delay is $\frac{1}{2} \cdot \frac{60S/min}{RPM}$.

SSDs have, in general, writes 10x reads and erasures 10x writes. They have no seek or rotational latency, but have a limited lifetime.

* Effective Bandwidth for n operations $= n/(S + n/B)$ (S: fixed overhead; B: time per op)

Queuing Theory

N = avg service rate
S = avg service time $= 1/N$
λ = avg arrival rate
U = utilization, $0 \le U \le 1$, $= \lambda S$
W = avg time spent in queue
R = response time $= W + S$
Q = avg length of queue $= \lambda W$ (Little's Law)
N = avg number of jobs in system $= \lambda R$

The graph of utilization vs response time is linear at first, but grows asymptotically to ∞. Why? Idle time accumulates & queue grows.

C is the squared coefficient of variance (σ^2/S^2) of service time. Assuming steady state (arrival rate = departure rate):
C=0 means regular arrival rate
C=1 means arrival rate is Poussin and interval between arrivals is exponential.

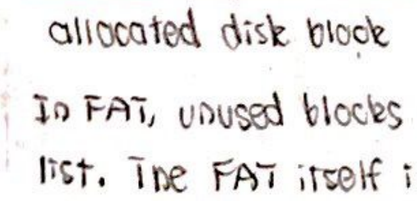
$$W = S \cdot \frac{1}{2}(1 + C) \cdot \frac{U}{1 - U}$$

C=1 called M/M/1 queue, general C called M/G/1 queue

File Systems

Blocks refer to logical view of data while sectors refer to physical disk locations.
A file system handles (1) naming: interface to find files by name; (2) disk management: interface to map disk blocks into files; (3) protection: some files should have access control; (4) reliability: persistent storage even if crash occurs.

- UNIX block size is 4KB while most disk sectors are 512B.
- In general, a file starts small and is grown according to how it is used.
- A file path (string) is translated through a directory structure. This gives a file number, or inumber, which is an index to the file header data structure (or inode).

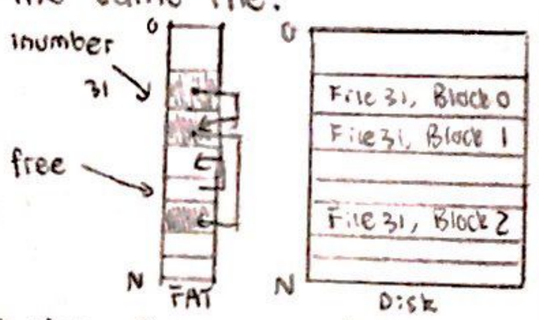


Typically users only deal with soft links.

FAT File System

The inumber is an index to the FAT, which is a 1-to-1 array of pointers from one block to the next in the same file.

The FAT entries are pointers; 0 means end of file and -1 means free.



External fragmentation: Small, divided regions of storage resulting from allocations and deallocations of varying size that are too small.
Internal fragmentation: Wasted space inside allocated disk block

In FAT, unused blocks are tracked with a free list. The FAT itself is stored in disk, but cached in memory on boot.

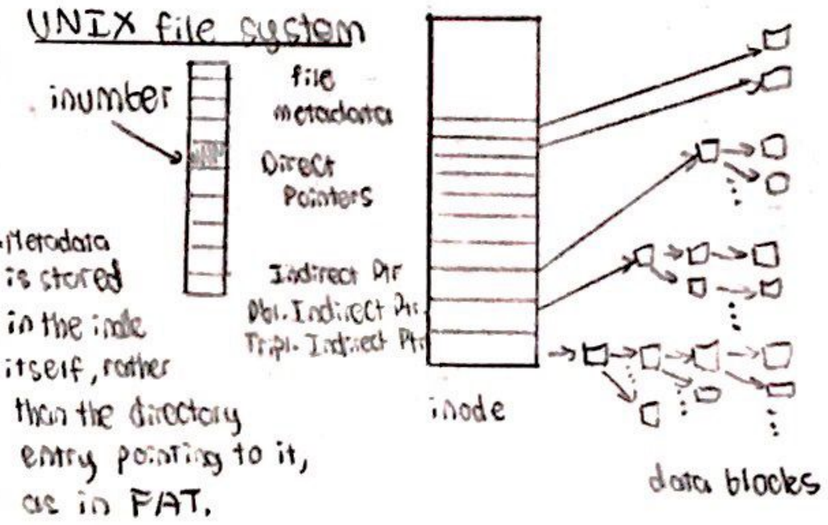
- Formatting a FAT system means resetting the FAT only, or optionally zeroing out blocks.

Pros: common so good compatibility; no external fragmentation, simple.

Cons: linear time lookup for random access; blocks are stored increasingly non-contiguously; FAT may not fit in RAM; no security at all.

Directories are treated as files whose disk block contents contain name, file number, and next file pointer of files within the directory.

UNIX file system



Metadata is stored in the inode itself, rather than the directory entry pointing to it, as in FAT.

There are 12 direct pointers; for 4KB blocks, this is sufficient for 48KB files. An indirect pointer points to a block of only pointers. Assuming 4B pointers, this is 1024·4KB = 4MB w/ indirect; 4GB w/ doubly indirect; 4TB w/ triply indirect.

Bitmap vs Free List:
Bitmap is proportional to disk size, which means wasted space when disk is full. However contiguous allocation is easier to perform.

Fast File System (modern UNIX):
- Bitmap instead of free list
- Contiguous blocks for most part
- 10% reserved disk space
- Skip-sector partitioning and/or read-ahead

Locality is achieved by dividing the file system volume into block groups - circular regions with data blocks, inodes, metadata, and free bitmaps interleaved. 10% of a block group is reserved to allow for larger contiguous runs. Because free bitmap always assigns first free bit, contiguous free space is left at the end for large files.

What about large directories? File systems like OpenBSD use B-trees: hash filename to get branch to take. Log lookup time.

NTFS File System (Windows)

Variable-length extents rather than fixed-size blocks. The Master File Table (MFT) contains data of small files directly, and points to nonresident extents for large files. Hence smaller internal fragmentation, but high external fragmentation.

Memory-Mapped Files: segment of virtual memory that has been assigned a direct byte-to-byte correlation with some portion of a file. Process can treat is like main memory.

```
void * mmap(void *addr, size_t len, int prot,
    int flags, int fd, off_t offset);
```

mmap() can also be used to share a file between processes. Pros of mmap: allow for in-place updates without seeking overhead. Cons: paging files in & out may be worse than overhead of traditional syscall-based I/O.

Availability: probability a system can process a request at a given time, e.g. 99.9%.
Durability: ability to recover from fault
Reliability: ability to perform function for specified period of time

RAID 1: each disk is fully duplicated

High-bandwidth reads, fast recovery, but slow writes and 100% capacity overhead

RAID 5:

	D0	D1	D2	D3	P0
Parity	D4	D5	D6	P1	D7
blocks	D8	D9	P2	D10	D11
spread	⋮	⋮	⋮	⋮	⋮
across	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
disks.					

Ex: $P0 = D0 \oplus D1 \oplus D2 \oplus D3$

Copy-on-write systems never update-in-place, but instead write new version and move a pointer. If crash results in partial write, revert to prev state. COW does this efficiently with tree based approach.

In general, transactions ensure atomic updates.

ACID properties of transactions:

Atomicity: all actions occur, or none at all

Consistency: maintain integrity, e.g. never have negative balance in bank

Isolation: No concurrency problems

Durability: If transaction commits, it persists

Idempotent operations can be repeated without effect after first operation.

In a journaling file system, a log of actions is used for recovery. In a fully log-structured system, writes go to an append-only log. Once changes are in log, the disk controller optimizes the order and actually performs the operations. This system is good for random writes but slower for reads or sequential access.

Distributed Systems: physically separate computers working together (a cluster).

The goal of a distributed system is transparency (hiding complexity) of location, migration, replication, concurrency, parallelism, and fault tolerance.

IPv6 = 128 bits

Networking

A network card is associated with a 48 bit MAC address and 32-bit (for IPv4) IP address, which is assigned dynamically.

A port number is a 16-bit identifier assigned by the OS.

IETF defines 5 layers:

Physical Layer: optical fiber, coaxial, etc.

Datalink Layer: exchange of packets on same physical link - uses MAC addresses. This is the LAN layer, using broadcast channels (e.g. wifi) or switches (e.g.

Internet Layer: Deliver IP packets across many datalinks over a WAN via routers. Routers forward based on IP addresses. Protocol is "best-effort" - packets can be dropped, corrupted, or recieved out of order.

Transport Layer: TCP & UDP most common

Application Layer: HTTP, SSH, Skype, etc.

- End-to-end principle: some functionality like reliability and security can only be implemented on end hosts.

"moderate" interpretation: If hosts can implement functionality correctly, implement it in a lower layer only as performance enhancement, if it does not impose a burden on applications not needing it.

TCP Details

3-way handshake: (1) Client sends SYN, seqnum = x. (2) Server returns SYN-ACK, seqnum = y, ack = x+1, (3) Client sends ACK, seqnum = x+1, ack = y+1. From then on, recieved ack becomes new seqnum and ack is recieved seqnum + length.

Timing diagrams: Sender tracks LastByteAcked, LastByteSent, AdvWin. Reciever tracks LastByteRecieved (not necessarily in sequence), LastByteRead (start of buffer) and AdvWin = MaxRcvBuffer - (LastByteRecieved - LastByteRead). For given data, sender begins sending packets of max-packet-size, up to advWin. The reciever ACKs each packet with ACK = next expected byte and AdvWin according to formula. Packet is then consumed (buffer moves right). Upon recieving ACK, the sender waits to send new data if LastByteSent - LastByteAcked ≥ AdvWin. Otherwise, continue from last byte sent.

The sender uses an adaptive window for congestion control. The window size increases by 1 for each ACK recieved, and cut window in half when we recieve a timeout. "Additive increase, multiplicative decrease"

Remote Procedure Calls

In distributed systems, a RPC calls a procedure on another computer with same syntax as normal call. A client "stub" "marshalls" arguments and "unmarshalls" return values. (and vice-versa for server stub). Stub generators are compilers that automate this code.

Two-Phase Commit

The General's Paradox states that messages over an unreliable network can never guarantee two entities do something simultaneously.

2-phase commit is a protcol for distributed transactions. One coordinator, N workers.

- Coordinator asks all workers if they can commit
- If all N workers "VOTE-COMMIT" before timeout, coordinator broadcasts "GLOBAL-COMMIT"
- Otherwise, coordinator broadcasts "GLOBAL-ABORT"
- Workers acknowledge, commit/abort. They log all messages.

In event of crash, worker can use log to recover state. Coordinator crash is more tricky - if fail while workers are in ready state, they must block until recovery

Byzantine General's Problem: distributed decision making with malicious failures.

- One general, n-1 lieutenants, of whom f are malicious
- All non-malicious lieutenants must come to same decision.
- Non-malicious general must be followed
- Only solvable if $n \geq 3f+1$.

Byzantine fault tolerance means protected against Byzantine malicious results.

Key-Value Stores

Simple interface: put(key, value), get(key)

Common, scalable "database" (does not satisfy ACID properties) in practice.

In directory-based architecture, a directory maps keys to nodes storing their values.

Scale by adding nodes & duplicating directories. Large variety of complex consistency algorithms.

Quorum Consensus: Define replica set of size N. put() waits for acknowledgements from at least W replicas and get() waits for R responses. If W+R > N, we are guaranteed at least one node has the update

Security: computing in presence of adversary

Requirements: (1) authentication, (2) data integrity, (3) confidentiality, (4) non-repudiation (use digital signatures)

- Hash H is cryptographically secure if
(1) infeasible to find m2 s.t. H(m2) = H(m1)
(2) infeasible to locate two colliding messages
(3) small change produces big random change in output hash.

Kerberos Key Exchange uses central auth server to generate keys between clients, encrypted with server's secret key.