Introduction to Computer Systems Cheat Sheet

Data representation

Byte = 8 bits

32-bit Machine

Char: 1 byte; Short: 2 bytes; Int: 4 bytes; Long: 4 bytes; Pointers: 4 bytes; Float: 4 bytes; Double: 8 bytes

64-bit Machine

Char: 1 byte; Short: 2 bytes; Int: 4 bytes; Long: 8 bytes; Pointers: 8 bytes; Float: 4 bytes; Double: 8 bytes

x86-64 systems

Char: 1 byte; Short: 2 bytes; Int: 4 bytes; Long: 8 bytes; Pointer: 8 bytes; Float: 4 bytes; Double: 8 bytes

- Decimal- Binary- Octal- Hexadecimal

In C, numeric constants starting with "0x" or "0X" are interpreted as being in **hexadecimal**. The letters A to F is not Case-sensitive.

The most common computer representation of signed numbers is known as **two's-complement form**. This is defined by interpreting the most significant bit of the word to have negative weight. When the sign bit is set to 1, the represented value is negative, and when set to 0 the value is nonnegative.

A useful equality: $-x = \sim x + 1$.

Consider the range of values that can be represented as a w-bit two's complement number. The least representable value is given by bit vector 10 ... 0 (set the bit with negative weight, but clear all others), having integer value **TMin(w bits)** $= -2^{w-1}$.

The greatest value is given by bit vector 01 ... 1 (clear the bit with negative weight, but set all others as "1"), having integer value **TMax(w bits)** $= 2^{w-1} - 1$.

So you can see: |TMin| = |TMax| + 1, UMax = 2TMax + 1, -1 has the same bit representation as UMax---a string of all ones.

$$U2T_w(u) = \begin{cases} u, & u < 2^{w-1} \\ u - 2^w, & u \ge 2^{w-1} \end{cases} \quad T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \ge 0 \end{cases}$$

When do calculation or comparison between unsigned and signed values, signed values implicitly cast to unsigned.

To convert an unsigned number to a larger data type, we can simply add leading zeros to the representation; this operation is known as **zero extension**.

For converting a two's- complement number to a larger data type, the rule is to perform a **sign extension**, adding copies of the most significant bit to the representation.

Shift operations

- Left shift
- **Logical right shift**: zero-extend
- **Arithmetic right shift**: sign-extend

Different effects for signed (2's complement) and unsigned numbers when simply throwing away the MSB in the overflow results

Array

- **Multi-Dimensional Arrays**
  - Row-major ordering in C
  - Each row is allocated contiguously, and all rows are allocated contiguously
- **Multi-Level Arrays**
  - The second-level arrays are not necessarily contiguous in memory

Satisfy alignment in structures with padding:

- Within structure: each field satisfies its own alignment requirement
- Overall structure: align to the largest alignment requirement of all fields

Big Endian: Least significant byte has highest address.

Little Endian: Least significant byte has lowest address.

Array not affected by byte ordering.

Big endian

	0x100	0x101	0x102	0x103	
. . .	01	23	45	67	. . .

Little endian

	0x100	0x101	0x102	0x103	
. . .	67	45	23	01	. . .

Floating-point

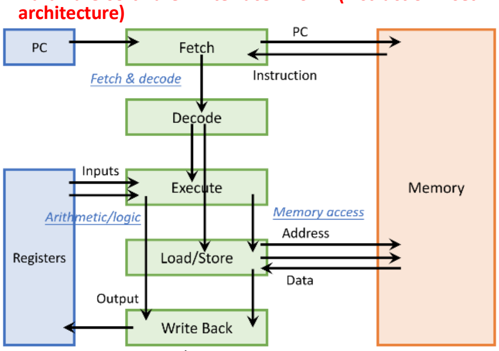
The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of 2.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit s directly encodes the sign s.
- The k-bit exponent field $\mathtt{exp} = e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field $\mathtt{frac} = f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

Hardware-software interface: ISA (instruction set architecture)



Optimization 1: Pipelining

Pipelining is a form of instruction-level parallelism

Optimization 2: Superscalar

Optimization 3: Multi-Core

PC and registers are private per-core; memory is shared

RISC-V ISA: States and Instructions

Program counter (PC): 32-bit

32 general-purpose registers (x0 to x31), each 32-bit

x0 is always fixed at 0; writing to x0 has no effect

Memory: 1 byte address

Word address is 4-byte aligned

Arithmetic/Logic Instructions

<op> rd, rs1, rs2 # inputs are 2 registers

<op>i rd, rs1, imm # inputs are 1 register and 1 constant

xor, add, sub, addi, subi

slli instruction: shift logical left by an immediate value

Load a 32-bit word (memory to register): lw rd, offset(rs1)

Store a 32-bit word (register to memory): sw rs2, offset(rs1)

Displacement addressing

- Address is given by a base register and a signed offset immediate
- Effective address is a byte address = register value + offset

Conditional branch: based on a condition

- Mark the target instruction of branch/jump with a label
- **beq rs1, rs2, L # branch to L if rs1 == rs2**
- **bne rs1, rs2, L # branch to L if rs1 != rs2**
- **blt rs, rt, label # branch to label if rs<rt**
- **bge rs, rt, label # branch to label if rs>=rt**

Sometimes we can invert the condition for convenience. if (a == d) => bne

Unconditional jump: always change flow

- **j L # jump to L**

Passing control: jump from caller to callee; return back

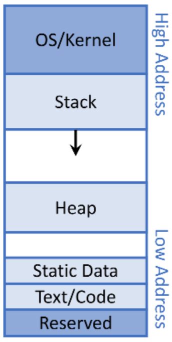
- **jal L**: jump-and-link, jump to L and save return position to register ra (x1)
- **jr ra**: jump to the address stored in ra

Test and set: **t&s reg, addr**

- reg = *addr; /* test */ if (reg == 0) *addr = 1; /* set */

SIMD: Single-Instruction, Multiple-Data

Memory Layout of a Program



Heap: dynamic data allocation, managed by OS

Stack: to support (recursive) procedure calls in hardware

- Arguments, **local** variables, temporary variables, ...
- last in, first out (LIFO), to match nested procedure call
- Stack grows downwards (low address) and shrinks upwards

Primary storage, a.k.a., main memory, or simply memory

- Fast but volatile (lose contents after powered off)

Secondary storage, a.k.a., external storage (treated as I/O devices)

- Slow but non-volatile

Tradeoff: large memory capacity vs. fast memory access speed

Locality

Temporal locality (locality in time)

- If an item has been referenced, it will tend to be referenced again soon

Spatial locality (locality in space)

- If an item has been referenced, nearby items will tend to be referenced soon

Exploiting Locality: Memory Hierarchy

Caches

Data are always copied back and forth between level k and level k + 1 in block-size transfer units

- **Cache hit**: data found in the cache; serve with short latency
- **Cache miss**: data not found in the cache; need to fetch the block from memory, and may replace a block in the cache

An empty cache is sometimes referred to as a cold cache, and misses of this kind are called compulsory misses or **cold misses**.

Restrictive placement policies of this kind lead to a type of miss known as a **conflict miss**, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing.

When the size of the working set exceeds the size of the cache, the cache will experience what are known as **capacity misses**.

Hit rate: percentage of accesses that hit

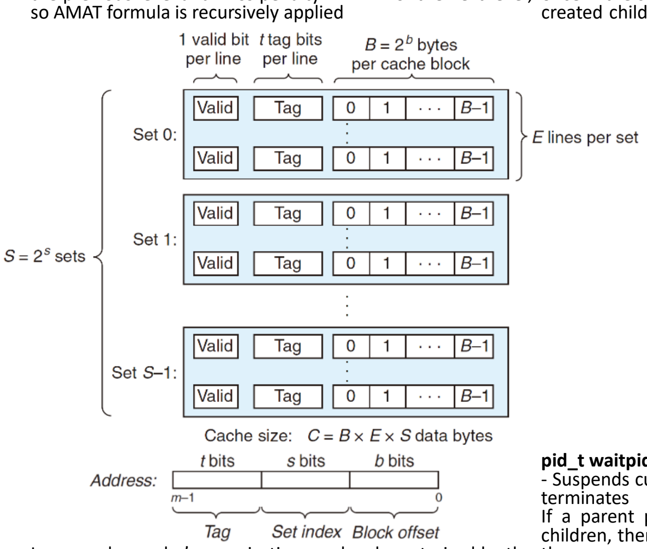
Miss rate = 1 – hit rate

Hit latency = data access latency on a hit = time to access cache

Miss penalty = overhead of fetching data from memory on a miss = time to access memory + time to deliver to cache (+ time to replace in cache)

AMAT (average memory access time) = hit latency + miss rate × miss penalty

At each level, Miss rate = local miss rate = # misses in this level / # accesses to this level = # misses in this level / # misses from the previous level and Miss penalty = AMAT of the next level, so AMAT formula is recursively applied



In general, a cache's organization can be characterized by the tuple (S, E, B, m). The size (or capacity) of a cache, C, is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus, **C = S × E × B**.

The s set index bits in A form an index into the array of S sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the **set index bits** tell us which set the word must be stored in. Once we know which set the word must be contained in, the **tag bits** in A tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the **b block offset bits** give us the offset of the word in the B-byte data block.

The process that a cache goes through of determining whether a request is a hit or a miss and then extracting the requested word consists of three steps: **(1) set selection, (2) line matching, and (3) word extraction.**

Fully Associative Caches

A fully associative cache consists of a single set (i.e., E = C/B) that contains all of the cache lines.

Notice that there are no set index bits.

Direct-Mapped Caches

A cache with exactly one line per set (E = 1) is known as a direct-mapped cache.

Set-Associative Caches

A cache with 1<E<C/B is often called an **E-way set associative cache**.

A **least frequently used (LFU)** policy will replace the line that has been referenced the fewest times over some past time window. A **least recently used (LRU)** policy will replace the line that was last accessed the furthest in the past.

write-through, is to immediately write w's cache block to the next lower level.

write-back, defers the update as long as possible by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm. The cache must maintain an additional dirty bit for each cache line that indicates whether or not the cache block has been modified.

write allocate, loads the corresponding block from the next lower level into the cache and then updates the cache block.

Associativity->conflict misses; Blocksize->compulsory misses; Capacity->capacity misses

Processes and Threads

A process is an instance of a executing program

A thread is a single unique execution context including registers, PC, stack pointer, memory, ...

a process = one or multiple threads + an address space

Process executions are time-interleaved on the processor

Usually one process has one address space, which is shared by all threads of that process

Base and bound (B&B)

OS kernel represents each process as a process control block (PCB), an in-memory data structure that contains process states

hardware provides at least two modes

- **Kernel mode** (or system mode, supervisor mode, protected mode): high privilege
- **User mode**: limited privilege

Three Ways of User To Kernel Transition:

- **System call (syscall)**: user process purposely requests a system service; proactive
- **Interrupt**: an **external** event triggers a context switch to kernel; reactive
- **Exception and trap**: an **internal** event triggers a context switch to kernel; reactive; internal reasons

Context switch could be triggered by

- An interrupt of a timer, e.g., every 100 microseconds
- A syscall to voluntarily yield the processor, e.g., sleep

Syscalls for Process Management

- **exit** – terminate a process

exit is called but never returns (to the program)

- **fork** – copy the current process

Parent process creates a new running child process by calling fork. The newly created child process is almost, but not quite, identical to the parent. The child gets an identical (but separate) copy of the parent's user-level virtual address space, including the code and data segments, heap, shared libraries, and user stack. The child also gets identical copies of any of the parent's open file descriptors, which means the child can read and write any files that were open in the parent when it called fork. The most significant difference between the parent and the newly created child is that they have different PIDs.

Call once, return twice. The fork function is interesting (and often confusing) because it is called once but it returns twice: once in the calling process (the parent), and once in the newly created child process. **In the parent, fork returns the PID of the child. In the child, fork returns a value of 0.**

Concurrent execution. The parent and the child are separate processes that run concurrently. The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way. In general, as programmers we can never make assumptions about the interleaving of the instructions in different processes.

- **exec** – change the program being executed by the current process

One call, no return if succeed.

- **wait** – wait for a process to finish

A terminated process that has not yet been reaped is called a **zombie**.

A process waits for its children to terminate or stop by calling the wait or waitpid function.

pid_t wait(int* wstatus)

- Suspends current process until one of its children terminates
- **Return value is the PID of the child process that terminated**

pid_t waitpid(pid_t pid, int* wstatus, int options)

- Suspends current process until the specific child process pid terminates

If a parent process terminates without reaping its zombie children, then the kernel arranges for the init process to reap them.

- **Signals** – small message to notify a process that some event has occurred
Sending/receiving signals: kill, sigaction

Scheduling

Waiting time: time when the job waits in the ready queue
Response time = waiting time + execution time
Throughput: number of jobs completed per unit of time
Fairness: all jobs use resource in some equal way
Convoy effect: short jobs are blocked behind long jobs
Two classes
- **Non-preemptive**: once giving a job some period of time to execute (not necessarily to completion), cannot take processor back before the period ends
- **Preemptive**: we can take processor back at any time, and use it for other jobs; preempted job goes back to ready queue and can resume later

FCFS (First-Come, First-Served)
Round Robin (RR): each job gets a small unit of time (time quantum, q) to execute, and then gets preempted and added back to end of queue
SJF (shortest job first): run the job with the shortest execution time first
SRTF (shortest remaining time first): preemptive version of SJF
SJF/SRTF is the optimal policy to minimize average response time

Basic shell command

1. Basic Commands
ls: Lists directory contents.
cd: Change directory.
mkdir: Make directory.
pwd: Print working directory.
cat: Concatenate and display files.
less / more: Pager programs to view long text files.
2. Managing Files and Directories
cp: Copy files and directories.
mv: Move or rename files and directories.
rm: Remove files or directories.
3. Redirection and Pipes
Redirection (>, >>, <): Redirects output to a file or reads input from a file.
Example: echo "Hello, World!" > hello.txt (writes "Hello, World!" to hello.txt).
Pipe (|): Sends the output of one command to another command as input.
Example: ls -l | grep "Jan" (lists files modified in January).
4. File Content and Size Management
wc (word count): Prints newline, word, and byte counts for each file.
Example: wc -l file.txt (counts the number of lines in file.txt).
df (disk free): Reports file system disk space usage.
Example: df -kh (shows disk space in human-readable form, including GB, MB).
5. Sorting and Manipulating Text
sort: Sorts lines of text files.
uniq: Reports or omits repeated lines.
6. Detailed File and Directory Manipulation
find: Searches for files in a directory hierarchy.
Example: find /home -name "*.txt" (finds all .txt files in the /home directory).
grep: Searches for patterns in files.
Example: grep "pattern" file.txt(Searches for pattern in file.txt and prints lines containing the pattern).
7. System Monitoring and Configuration
ps (process status): Reports a snapshot of current processes.
Example: ps -ef (shows every process running on the system).
top: Displays an up-to-date list of running processes.
kill: Sends a signal to a process, usually related to stopping the process.
nohup: Allows a command to continue running after logging out.
8. Shell globbing
?: Matches any single character.
Example: ls file?.txt would match files named file1.txt, file2.txt, etc., but not file10.txt.
*****: Matches zero or more characters.
Example: ls *.txt would match any file ending in .txt, such as file.txt, document.txt, etc.
Brace Expansion Syntax: {item1,item2,...}
Expands into multiple items that can be used to generate strings for file names or command arguments.

Regex

[] The string of characters inside the braces specifies a disjunction of characters to match.
- In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (-) to specify any one character in a range.

^ If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated.
? This means "the preceding character or nothing".
***** The Kleene star means "zero or more occurrences of the immediately previous character or regular expression".
+ This is the Kleene +, which means "one or more occurrences of the immediately preceding character or regular expression".
. A wildcard expression that matches any single character (except a carriage return).
| Means disjunction
Anchors:

Regex	Match
^	start of line
$	end of line
\b	word boundary
\B	non-word boundary

Regex	Expansion	Match
\d	[0-9]	any digit
\D	[^0-9]	any non-digit
\w	[a-zA-Z0-9_]	any alphanumeric/underscore
\W	[^\w]	a non-alphanumeric
\s	[\r\t\n\f]	whitespace (space, tab)
\S	[^\s]	Non-whitespace

{n} exactly n occurrences of the previous char or expression
{n,m} from n to m occurrences of the previous char or expression
{n,} at least n occurrences of the previous char or expression
{,m} up to m occurrences of the previous char or expression

Concurrency and Synchronization

Concurrent threads are a very useful abstraction
- Allow transparent overlapping of computation and I/O
- Allow use of parallel processing when available

Concurrent threads introduce problems when accessing shared data
- Programs must be insensitive to arbitrary interleavings
- Without careful design, shared variables can become completely inconsistent

Atomic Operation: an operation that always runs to completion or not at all
Synchronization: using atomic operations to ensure cooperation between threads
Critical Section: piece of code that only one thread can execute at once
Mutual Exclusion: ensuring that only one thread executes critical section
Semaphores are a kind of generalized locks
P(s): If s is nonzero, then P decrements s and returns immediately. If s is zero, then suspend the thread until s becomes nonzero and the thread is restarted by a V operation. After restarting, the P operation decrements s and returns control to the caller.
V (s): The V operation increments s by 1. If there are any threads blocked at a P operation waiting for s to become nonzero, then the V operation restarts exactly one of these threads, which then completes its P operation by decrementing s.
The basic idea is to associate a semaphore s, initially 1, with each shared variable (or related set of shared variables) and then surround the corresponding critical section with P(s) and V (s) operations.
A semaphore that is used in this way to protect shared variables is called a **binary semaphore** because its value is always 0 or 1. Binary semaphores whose purpose is to provide mutual exclusion are often called **mutexes**. Performing a P operation on a mutex is called **locking the mutex**. Similarly, performing the V operation is called **unlocking the mutex**. A thread that has locked but not yet unlocked a mutex is said to be **holding the mutex**. A semaphore that is used as a counter for a set of available resources is called a **counting semaphore**.
Mutual Exclusion (initial value = 1)
Scheduling Constraints (initial value = 0)
- Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given constrained is satisfied

Producer-Consumer Problem
A producer and consumer thread share a bounded buffer with n slots. The producer thread repeatedly produces new items and inserts them in the buffer. The consumer thread repeatedly removes items from the buffer and then consumes (uses) them.
Since inserting and removing items involves updating shared variables, we must guarantee mutually exclusive access to the buffer. But guaranteeing mutual exclusion is not sufficient. We also need to schedule accesses to the buffer. If the buffer is full (there are no empty slots), then the producer must wait until a slot becomes available. Similarly, if the buffer is empty (there are no available items), then the consumer must wait until an item becomes available.

```
Semaphore fullSlots = 0;  // Initially, no coke
Semaphore emptySlots = bufSize;  // Initially, num empty slots
Semaphore mutex = 1;  // No one using machine
Producer(item) {
	emptySlots.P();  // Wait until space
	mutex.P();  // Wait until machine free
	Enqueue(item);
	mutex.V();
	fullSlots.V();  // Tell consumers there is more coke
}
Consumer() {
	fullSlots.P();  // Check if there's a coke
	mutex.P();  // Wait until machine free
	item = Dequeue();
	mutex.V();
	emptySlots.V();  // tell producer need more
	return item;
}
```

The order of P's is important, the order of V's is not important.
Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
Condition Variable: a queue of threads waiting for something inside a critical section
Operations:
- **Wait(&lock)**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
- **Signal()**: Wake up one waiter, if any
- **Broadcast()**: Wake up all waiters

```
Lock lock;
Condition dataready;
Queue queue;
AddToQueue(item) {
	lock.Acquire();  // Get Lock
	queue.enqueue(item);  // Add item
	dataready.signal();  // Signal any waiters
	lock.Release();  // Release Lock
}
RemoveFromQueue() {
	lock.Acquire();  // Get Lock
	while (queue.isEmpty()) {
		dataready.wait(&lock);  // If nothing, sleep
	}
	item = queue.dequeue();  // Get next item
	lock.Release();  // Release Lock
	return(item);
}
```

Hoare monitors
Signaler gives up lock and CPU to waiter; waiter runs immediately.
Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again.
In Hoare Monitors, when a process signals a condition, it immediately transfers control to a waiting process. This means the signaling process stops its execution and the waiting process resumes, using the resource or condition that has just been signaled.
Mesa monitors
Signaler keeps lock and processor
Waiter placed on a local "e" queue for the monitor
In Mesa Monitors, when a process signals a condition, it does not immediately transfer control to a waiting process. Instead, the signaling process continues, and the waiting process is simply moved to the ready queue.
The waiting process in Mesa Monitors must recheck the condition upon waking up because there's no guarantee that the condition is still true by the time it gets processor control. This is known as "spurious wake-ups" and requires extra handling in the code. So we should use "while()" instead of "if()" with Mesa monitors.
Readers-Writers Problem
Writers must have exclusive access to the object, but readers may share the object with an unlimited number of other readers. In general, there are an unbounded number of concurrent readers and writers.
The second readers-writers problem, which favors writers, requires that once a writer is ready to write, it performs its write as soon as possible.

```
Reader() {
	// check into system
	lock.Acquire();
	while ((AW + WW) > 0) {
		WR++;
		okToRead.wait(&lock);
		WR--;
	}
	AR++;
	lock.release();
	// read-only access
	AccessDbase(ReadOnly);
	// check out of system
	lock.Acquire();
	AR--;
	if (AR == 0 && WW > 0)    okToWrite.signal();
	lock.release();
}
Writer() {
	// check into system
	lock.Acquire();
	while ((AW + AR) > 0) {
		WW++;
		okToWrite.wait(&lock);
		WW--;
	}
	AW++;
	lock.release();
	// read/write access
	AccessDbase(ReadWrite);
	// check out of system
	lock.Acquire();
	AW--;
	if (WW > 0){
		okToWrite.signal();
	} else if (WR > 0) {
		okToRead.broadcast();
	}
	lock.release();
}
int readcnt; /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */
void reader(void){
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Critical section *//* Reading happens */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
void writer(void){
    while (1) {
        P(&w);
        /* Critical section *//* Writing happens */
        V(&w);
    }
}
```

starvation, where a thread blocks indefinitely and fails to make progress.
deadlock, where a collection of threads is blocked, waiting for a condition that will never be true.
Four requirements for Deadlock
- **Mutual exclusion**
Only one thread at a time can use a resource
- **Hold and wait**
Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
There exists a set {T1, …, Tn} of waiting threads
T1 is waiting for a resource that is held by T2; T2 is waiting for a resource that is held by T3⋯Tn is waiting for a resource that is held by T1

Methods for Handling Deadlocks
- Allow system to enter deadlock and then recover
- Deadlock prevention: ensure that system will never enter a deadlock
- Ignore the problem and pretend that deadlocks never occur in the system

Techniques for Preventing Deadlock
- Infinite resources
    - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
- No Sharing of resources (totally independent threads)
- Don't allow waiting

Mutex lock ordering rule: Given a total ordering of all mutexes, a program is deadlock-free if each thread acquires its mutexes in order and releases them in reverse order.